The Surf-Hippo User Manual
Version 3.0 β

Lyle J Graham*
Laboratoire de Neurophysique et Physiologie du Système Moteur, CNRS UMR 8119
UFR Biomdicale de l'Université René Descartes
45 rue des Saint-Peres
75006 Paris, France

July, 2002

Abstract

This document comprises the user manual for the Surf-Hippo Neuron Simulation System, a public domain circuit simulation package written in Lisp for Unix workstations and PCs that is used to investigate morphometrically and biophysically detailed compartmental models of single neurons and networks of neurons. Surf-Hippo allows ready construction of cells and networks using various file formats or a variety of built-in functions. Surf-Hippo has an extensive user interface, including menus, 3D graphics of dendritic trees (with point-and-click measurement, examination and modification of circuit elements), and data plotting (reformatable, with mouse-oriented measurement).

Companion volume to the Surf-Hippo Reference Manual.

http://www.cnrs-gif.fr/iaf/iaf9/surf-hippo.html

⋆ ⋆ Beta release: Comments and corrections will be appreciated. ⋆ ⋆

*lyle@biomedicale.univ-paris5.fr

Contents

List of Figures

List of Tables

1 Introduction

The Surf-Hippo neuron simulator is used to study morphologically and biophysically detailed compartmental models of single neurons and networks of neurons. Surf-Hippo allows ready construction of cells and networks using built-in functions and various anatomical file formats (Neurolucida, NTS and others). Surf-Hippo is a public domain program, written in Lisp, and runs under Unix and Linux.

Cell models may have complicated 3-dimensional dendritic trees with distributed non-linearities and synaptic inputs driven by arbitrary inputs or other cells. The number of circuit nodes (cell compartments) is in practice limited by the memory in the machine; simulations with thousands of nodes are routine. The retention of XYZ coordinates for each circuit node may be exploited, for example, by simulations which use spatially-coded input, or with network simulations where synaptic connectivity is defined by spatial proximity. Channels (Hodgkin-Huxley, an extended Hodgkin-Huxley model (Borg-Graham, 1991, 1999), Markovian (Borg-Graham, 1999), and others), synapses (event-driven, voltage-dependent, light-dependent, and others), current and voltage sources may be added at arbitrary locations within a neuron. Stationary or moving two dimensional input (e.g. light, for retinal (Borg-Graham and Grzywacz, 1992; Borg-Graham, 2001) or visual cortex (Gazères et al., 1998) simulations) is also provided. Multi-compartment concentration systems are provided for modelling mechanisms such as calcium-dependent processes.

Surf-Hippo has an extensive graphical user interface (GUI), including menus, 3D graphics of dendritic trees (with point-and-click measurement, examination and modification of circuit elements), and data plotting (reformatable, with mouse-oriented measurement). Publication quality postscript files of all graphical output are easily generated. Data files may also be saved for analysis with external tools. Surf-Hippo allows automatic saving of edited model parameters into files which are both loadable and human readable, a feature which helps to avoid errors in saving results during a simulation session.

For integrating the circuit equations, Surf-Hippo uses a variant of the Crank-Nicholson method described by Hines (1984; Borg-Graham, 1999, see Section 37). A major difference in the method used by Surf-Hippo is a variable time step option, where step size is adjusted according to an estimate of the linear truncation error for all state variables (e.g. node voltages, channel particles). The adaptive time step can give much faster run times for typical simulations, with the option of verifying selected results using the more conservative fixed time step integration. Another difference is that Surf-Hippo allows for either ideal or non-ideal voltage clamp at arbitrary nodes in the circuit.

A more detailed reference for many of the cellular mechanisms available in Surf-Hippo may be found in Borg-Graham, 1999.

1.1 Motivations, and Why Lisp?

In general, the application of Surf-Hippo overlaps programs such as GENESIS (Bower and Beeman, 1994) and NEURON (Hines, 1992), as well as several other simulators designed for the computational neuroscience community (De Schutter, 1992). The fact that Surf-Hippo is the only complete system written in Lisp has at least the following advantages.

To begin with, the necessity for the numerical analysis of the nonlinear dynamical systems that characterize compartmental neuron models means that formal verification of even moderately complicated neuron models is not possible. A practical approach to verification is by the cross-validation of models using independent tools, e.g. simulators written in different languages and with major or minor differences in algorithms. Thus, in an ideal world (someday!) neuron models will be evaluated with more than one software system.

Another advantage follows from the ability of Lisp to readily handle both symbolic as well as numeric representations and relationships. Formulation of complex models of physical systems - of which the brain is perhaps the canonical example - is by necessity a symbolic task. Thus, access to model components in Surf-Hippo is quasi object-oriented. For example, a large family of functions are based on the concept of circuit elements, which apply to different types of physical models (somas, dendritic segments, channels, particles, channel types, synapses, etc.; see Section 6). Coupled with the natural language-like syntax of Lisp, this makes for extremely efficient writing of model construction and analysis, without demanding a highly sophisticated knowledge of Lisp by the user. Simulation scripts written by the user, a necessity for serious parameter searching, are also in Lisp, whose flexible and powerful syntax is arguably more transparent than most other languages.

Evaluation of those models, on the other hand, is a numerical task. Despite its flexibility, the numerical performance of Lisp can be similar to C or Fortran. Our results using the Rallpack benchmark suite (Bhalla et al., 1992) show that the speed and accuracy of Surf-Hippo is comparable to GENESIS and NEURON (see Section 37.14).

Lisp has the advantage that the user communicates directly with the Lisp interpreter environment, and thus has complete access to all components of a simulation. In practice, it is often convenient to move in and out of the Surf-Hippo GUI loop and access simulation data directly from the Lisp interpreter. The data analysis and graphical capablities of Surf-Hippo are sufficiently sophisticated so that we use this simulator for working with real physiological data (and for directly comparing that data with simulation data).

Another advantage of Lisp is that all functions defined in the system may be executed from the interpreter either individually or within scripts, which makes for a very flexible working environment. In fact, the running Lisp system includes the compiler and the interpreter along with the runtime executable. New code (including bug fixes) may be (incrementally) compiled and used as needed, without recompiling the entire executable. These features, plus those such as the integrated documentation of Lisp and flexibility regarding types, makes the development and maintenance of a large system like Surf-Hippo much more efficient.

1.2 History and Acknowledgements

The initial motivation for Surf-Hippo came from the neuron simulation programs BULLFROG and NEURON (not the same as that of Michael Hines), developed by Christof Koch and Patrick O'Donnell on Symbolics Lisp Machines at the MIT AI Laboratory. A new simulator, HIPPO (as in hippocampus), was developed on the Lisp Machine, and later Surf-Hippo was written with some features based on the SURF (as in circuit node WAVEforms) circuit simulator, written by Don Webber, then of the VLSI CAD Group at the University of California at Berkeley. The name "Surf-Hippo" has the advantage of not being already used in other common contexts, e.g. biblical or biological, nor is it an attempt at a tortured acronym. The PLOT-HACK name and many of the ideas in the plot utilities that I wrote for Surf-Hippo were inspired by Patrick. Surf-Hippo was developed first within the former Center for Biological Information Processing (Tomaso Poggio and Ellen Hildreth, directors), Department of Brain and Cognitive Sciences, MIT. This project continues at the Laboratoire de Neurophysique et Physiologie du Système Moteur, CNRS, Paris, France. We would greatly appreciate reprints or pointers to work that uses Surf-Hippo. Please use the address found in the surf-hippo/lib/SNAIL-MAIL file or `lyle@biomedicale.univ-paris5.fr`.

To cite this package, please use:

```
Graham, L., The Surf-Hippo Neuron Simulation System, v3.0, 2002
(http://www.cnrs-gif.fr/iaf/iaf9/surf-hippo.html)
```

With thanks to the Garnet group at CMU - the system organization for Surf-Hippo is derived directly from Garnet, and to the CMUCL group at CMU. The work of both groups contributed enormously to the realization of this project. Also a tip 'o the hat to the CMUCL net community and their ongoing help. I would also like to acknowledge the important contributions to this project from Leonardo Topa (MIT), Cyril Monier (CNRS) and Nicolas Gazeres (CNRS, who also contributed to Appendix A).

2 Help!

Before delving further into the system, we will describe a few basic conventions, and where to find Help!.

2.1 CONTROL Key Sequences, and Other, Commands

By convention, the key sequence denoted by "C-<*key*>" means holding down the CONTROL key while pressing <*key*>. Likewise, "C-<key_1> <key_2>" means follow the CONTROL <key_1> combination with pressing <key_2>. <*key*> may refer to a mouse button, assuming a 3-button mouse: thus, LEFT, MIDDLE and RIGHT. More complex chords include the SHIFT and ESC keys.

2.2 Bugging Out and Debugging in the Lisp Environment

It is unusual to crash Lisp completely - more often than not if an error is encountered you will find yourself in the Lisp Debugger (see Appendix B). The Debugger provides many tools for discovering the precise source of a problem. Often, the Debugger conserves program state, allowing for a complete recovery or even continuation from the error condition. If for one reason or another (usually by mistyping something) you find yourself in the Debugger, in many cases it may be sufficient to simply abort and return to the top level by typing "q". For example, an error is returned when a unbound symbol is evaluated:

```
* grok

Error in KERNEL::UNBOUND-SYMBOL-ERROR-HANDLER:  the variable GROK is unbound.

Restarts:
  0: [ABORT] Return to Top-Level.
  .
  .
  .
0] q
*
```

Aborting to Top-Level allows you to continue without restarting the entire system. Otherwise, you can use the powerful diagnostic information available with the Debugger to find the problem (see also Section 29.2).

2.3 The System Seems Dead

During simulations or other evaluations (like the Main Menu loop) the Lisp interpreter and, nornally, all Surf-Hippo windows, will be "dead". For example, if there is a buried Surf-Hippo window is raised during a simulation, the window will be blank until the simulation is finished. This annoyance can be circumvented by temporarily interrupting the Lisp process (Section 4.8).

From time to time Lisp will stop what it is doing and run a "garbage collection" (GC) for managing the memory space. This will cause the system to appear dead until the GC is finished (normally within a second or so.

2.4 How to Find Help!

There are several ways to get help while running Surf-Hippo:

- Printed help
  - The User Manual (this document)
  - The Reference Manual
  - Common Lisp: The Language, 2nd edition (a.k.a. CLTL2)
  - A large collection of Lisp references

- On-line help
  - Window help
  - The Lisp function `APROPOS`
  - The Lisp function `DESCRIBE` (and related Ilisp commands)
- Internet help
  - comp.lang.lisp
  - ALU Lisp Tools page (http://www.elwood.com/alu/table/tools.htm)

2.5 Printed Help

2.6 On-line Help

2.6.1 Help regarding Surf-Hippo windows

Each Surf-Hippo window, including menus, support various commands entered by the mouse and keyboard. A summary of these commands for most types of windows is displayed by typing "h" with the mouse over the window in question.

Typing C-q over any Surf-Hippo output window will deiconify and raise any active menus. Destroying Surf-Hippo output windows is done by typing C-d with the mouse over the target window. Note that text entry windows in the menus work similarily to Emacs (Appendix F).

If menus do not respond to the mouse, make sure that the Caps Lock key is OFF.

Note: avoid destroying a Surf-Hippo window with the X window manager: instead, type C-d over the window (see Section 40.1).

2.6.2 Help from Lisp

All Surf-Hippo Lisp functions that the user would typically use have associated documentation; the general way to see this is to use the `DESCRIBE` function, described in Section 31. Otherwise, the Surf-Hippo Reference Manual includes all documented functions and variables. For finding defined functions and variables from only part of their name, the `APROPOS` function is used (see Appendix A.7).

2.6.3 Help from Emacs and Ilisp

Emacs, in particular when using the special Ilisp mode, has an extensive documentation system. The caonical help access is found by typing C-h in the Emacs window, and following the instructions that appear at the bottom of the window. Of particular interest is the Emacs tutorial (C-h t).

The Ilisp mode provides an excellent interface to the documentation for the running Surf-Hippo. A summary of these commands may be obtained by typing C-h m into an Ilisp (e.g. Surf-Hippo) window. For example, typing C-z i (describe-lisp) with the cursor over a Lisp function, macro or variable, will be equivalent to using `DESCRIBE`.

2.7 Internet Help

2.8 Where Else to Look

For typical error messages, bugs and other problems, see Sections 40 and 41 or the Bugs section in many of the sections. Typical Debugger messages will be made throughout the text, and there is more discussion in Section B.

3 Surf-Hippo is a Lisp Program

Surf-Hippo is loaded as an integral part of a complete, running Lisp, and interaction with Surf-Hippo, at least initially, is accomplished by typing directly to Lisp, specifically at the so-called Lisp top level (see Section 4.1). This interaction is made through the "Lisp window" (alternatively, the "Lisp interpreter" or "Lisp Listener"), a text window that the Lisp process uses as standard output and input, and that displays some sort of:

```
lisp-prompt> (some-stuff)
```

which normally is an `*`, thus:

```
* (some-stuff)
```

This is in contradistinction to:

```
unix-prompt>
```

In this manual, we will generally assume that the `lisp-prompt>` is "`*`" (default for CMUCL). Strictly speaking, when you enter commands directly to Lisp, you are communicating with the Lisp interpreter (the Lisp *read-eval-print* loop) via the appropriate window, e.g. a terminal window, an Ilisp emacs buffer (see Sections 38 and 39.3), etc.

It is important to emphasize that essentially all the syntactical rules for Surf-Hippo functions, macros, constants and variables are the same as for Lisp - Surf-Hippo does not use some meta-language distinct from Lisp. A running Surf-Hippo system is thus a Lisp system, onto which a large set of additional functions, variables and other definitions have been loaded. The overall directory structure of the Surf-Hippo system, including source code, documentation and data directories, among others, is described in Section 30.3.

Of the approximately 3700 Surf-Hippo functions and macros, and 3400 other Surf-Hippo symbols (including variables and constants), about 950 include documentation strings as a part of their definition (documentation/comments for the remaining definitions can be found in the source code, the location of which is provided by using the `DESCRIBE` function - see below). These documented functions, macros, variables and constants are described in the Reference Manual, and should provide a sufficient set of definitions for writing any user code.

3.1 Conventions in this Manual, and a Miniature Lisp Lesson

In this manual we will use `typewriter style` for examples of input typed directly to Lisp, for Lisp output, for program source code, and for Lisp *symbols* (in this last case, normally in uppercase, e.g. `FOO`). Lisp symbols may refer to defined functions, macros, model types, or variables. A given symbol can have multiple definitions, where the appropriate meaning is taken according to the syntactical environment. Various defined Surf-Hippo symbols are referenced here, displayed in a format that indicates their definition as functions, macros or variables, and lists, as appropriate, their arguments or values. For the full documentation of the Surf-Hippo functions, macros and variables of interest to users, see the Surf-Hippo Reference Manual. For a further introduction to Lisp, see Appendix A.

There is a distinction between Lisp *strings* and Lisp symbols. Strings, which are framed by double quotes, are actually sequences of (character) data, just like a Lisp *list* or an array represents a sequence. Lists are framed by parentheses, and Lisp arrays are printed out framed by parentheses and preceded by a `#`. Thus,

```
"A string with 27 characters"
(A LIST WITH 5 MEMBERS)
#(AN ARRAY WITH 5 MEMBERS)
```

Comments are preceded by a semicolon, thus:

```
(+ A (* B C) D)  ; Add the values of A and D to the product of B and C
```

The list is the basic (but not only!) type of structure in Lisp. Lists may represent data, for example a list of voltage values, or a list of simulation times. Lists also represent operations of some sort or another, based on infix notation. In this case the list represents some *expression*, where the first element of the list is taken as the operator (typically a *function*) that is applied to the remaining members of the list. Expression precedence is unambiguous with the infix notation of Lisp, since everything is evaluated from the inside out in nested lists: e.g. the Lisp expression `(+ A (* B C) D)` is equivalent to $A + BC + D$.

All the *elements* that define a circuit are of one model type or another. Each model type is associated with a defined data structure and other characteristics, including certain functions and variables. Some model types have an associated *element type* (which is itself a model type). For example, each synapse in a circuit is an instance of the `SYNAPSE` model, whose characteristics are defined by a synapse type (which itself is an instance of the `SYNAPSE-TYPE` model).

Lisp variables may be either *global*, meaning they may be referenced (and altered) by different functions at different times, or they may be *local*, relevant only during the evaluation of some function. By convention, global variables are framed with asterisks, e.g. `*USER-STOP-TIME*`. Various important predefined global variables of Surf-Hippo, such as `*USER-STOP-TIME*`, are described throughout this manual, or in the Reference Manual.

Sometimes, one is interested in the value associated with (bound to) a symbol (if any), while other times it is the symbol itself which is needed. The first case requires that Lisp "evaluate" the symbol. For the second case, in order to *prevent* evaluation, the symbol must be preceded by a quote (note that a leading quote prevents evaluation of any Lisp form, not just a symbol). For example, if the symbol `*USER-STOP-TIME*` is entered into Lisp with a leading quote, then Lisp just returns the symbol, without evaluating it:

```
* '*user-stop-time*
*USER-STOP-TIME*
```

Note that the *printed* representation of an unevaluated (quoted) symbol does not include the leading quote.

If the symbol is read by Lisp without the quote, then it is evaluated (according to the current context, which in this case implicates it as a variable), and its current value is returned. Thus, the symbol `*USER-STOP-TIME*` is normally bound to some number:

```
* *user-stop-time*
100.0
```

If you forget what some symbol is supposed to mean, Lisp provides the `DESCRIBE` function:

```
* (describe '*user-stop-time*)
*USER-STOP-TIME* is an internal symbol in the SURF-HIPPO package.
It is a special variable; its value is 100.0.
   100.0 is a SINGLE-FLOAT.
Special documentation:
  The time to end the simulation, in milliseconds.
```

Note that in Lisp all symbol are defined in the context of some "package" (in this case the `SURF-HIPPO` package), which allows the same symbol to be used in different packages for different things, as long as the symbols are not "exported" to make them available for other packages. However, for most cases, you don't need to be concerned about this distinction.

If a symbol is evaluated but does not have an associated value, then an error is returned (see below about Bugging Out). Setting the value of a variable is done with the `SETQ` form:

```
* (setq *user-stop-time* 200.0)
200.0
```

Some variables are used as boolean values, and accordingly the values `T` and `NIL` represent true or false, respectively:

```
* (setq *plot-standard-windows* T)
T
```

Note that both `T` and `NIL` are symbols, not strings, but unlike conventional (non-keyword) symbols when typing a leading quote is not necessary. Note also that in a boolean evaluation in Lisp any value other than `NIL` is considered "true"; using `T` tends to emphasize that a given variable is intended only as a boolean variable (the symbols `T` and `NIL` are reserved symbols in Lisp, whose meanings are immutable).

Lisp symbols may include a leading colon, in which case the symbol is called a *keyword*. Keywords are often used to denote specific arguments in function definitions (see below). Keyword symbols are never evaluated - the colon prevents evaluation just like the quote does for other Lisp symbols and forms (this also means that keywords cannot be used as variables). The printed representation of a keyword, however, retains the colon:

```
* :mumble
:MUMBLE
```

Note as well that symbols, with or without leading colons, are by default case-insensitive in Lisp, and are represented internally with upper-case characters.

Lisp functions calls are written as lists with the general format of:

```
(function-name arg0 arg1 ...)
```

where the number and format of the arguments is given by the function definition. Function (or macro) arguments referenced within the text will be written in CAPITALS. Function arguments include *required* arguments, *optional* arguments and *keyword* arguments. Required arguments, if any, always immediately follow the function name, and must always be included. Optional arguments, if any, come next, and they may be included as desired. Keyword arguments in a function call must be preceeded by the keyword itself, and the keyword keyword-value pairs may occur in any order after any required or optional arguments. Note that if keyword arguments are to be included in a function call, then *all* defined optional arguments must be explicitly included (after any required arguments) as well.

For example, consider the phony function `FOO` - the following format is typical for *describing* the syntax of Lisp functions and macros (this is `not` how you would write this function in actual code):

foo *bar baz* &optional *(boo 14)* &key *(bee :ignore) bop* [*Function*]

In this presentation, `FOO` is the name of a function, with arguments BAR, BAZ, BOO, BEE and BOP. BOO is an optional argument of `FOO`, with a default value of 14, and BEE and BOP are keyword arguments, whose default value is :IGNORE and NIL. The arguments BAR, BAZ must always appear in this order, and BOO, if included, must be next. The keyword arguments follow, if included.

As mentioned, the above format gives the syntactical summary of the function, which then guides how the function is written in actual Lisp code. Thus, the following would all be legal formats for writing a call to the function `FOO`:

```
(foo t 3.14159)
(foo t 3.14159 11)
(foo t 3.14159 11 :bee :normalize :bop 'sphere)
(foo t 3.14159 11 :bop 'sphere :bee :normalize)
```

Although in the function description above keywords (such as BOP) do not include a colon, in an actual function call of `FOO` the BOP argument (if used) would be preceeded by the :BOP keyword. In this example the symbol 'SPHERE is used as the value of the keyword argument :BOP. It includes a leading quote because it is desired to pass the symbol itself to the function, not any value bound to it. Note also that keyword symbols (here :NORMALIZE) may be used as function arguments themselves, even for keyword arguments, depending on what a function expects.

References will be made in this manual to source code that comes with the Surf-Hippo distribution. Non-essential code, such as optimization forms (e.g. lines starting with `(declare (optimize ...))`), or type declarations (e.g. `(the sf foo)`), will usually be left out for clarity and thus examples may not exactly match the original source. For example, Lisp does not *a priori* require type declarations (see Section 3.2), but they may be very important to generate efficient compiled code. In fact, a key advantage of Lisp is that

programs may be written with hardly any attention to such details, such that prototyping (and most sorts of useful simulation scripts) can be written "quick and dirty".

Circuit elements in Surf-Hippo are referenced either in terms of their print names or the structure pointers, depending on the context. Many Surf-Hippo functions are written to allow either technique for convenience (see Section 6). Element names are either strings, symbols, or numbers:

```
"Hippo-1"
'CA1
234
```

The *printed* representation of data structures are all something like this:

```
<Segment Hippo-1: prox node Hippo-soma>
<Cell Type CA1>
<Synapse 234: type AMPA-AUTO>
```

Thus, and surrounded by "<>", is the type of data structure, followed by the name of the structure and, possibly, some minimal information. The printed representation of any data structure is intended for human readability only. Surf-Hippo won't understand it if you try to enter something like "<Segment Hippo-1: prox node Hippo-soma>", for example in order to reference a particular data structure (here a segment). On the other hand, Surf-Hippo allows you to reference data structures using their names with the `ELEMENT` function, described in Section 6, for example `(element "Hippo-1")`, `(element 'ca1)` or `(element 234)`.

Finally, from time to time we shall refer to circuits and and mechanisms that are included in the Surf-Hippo distribution. One example that will be used often is a single cell model of a hippocampal pyramidal cell (Borg-Graham, 1999) that may be referenced in the text as simply the "Working Model" (the Lisp function that sets this model up is `WORKING-HPC`).

3.2 Number Types

As mentioned, Lisp allows varying degrees of *type specification*, in particular for numbers. For most documented Surf-Hippo functions that you will have occasion to use, any necessary type conversion is done automatically (e.g. with respect to a numeric function argument, a number is a number, rather than having to be a single float, integer, etc.). This will be seen in many of the examples in this manual where, for example, function calls are shown with integer values even though the values used internally in the function will be floating point numbers. However, for more efficient code certain documented Surf-Hippo functions do restrict the type of their numeric parameters (e.g. fixnum, single or double float), if any. This applies mainly to "low-level" utility functions, for example:

gaussian *x mean variance* [*Function*]

where the X, MEAN and VARIANCE arguments must be a single float number.

If for some reason a type violation is encountered the Debugger will be invoked automatically: In most cases you can then back up the execution stack to identify the offending function call or variable assignment (see Appendix B). Depending on whether the function has been compiled with strong optimization (often the case if the argument types are specified), the initial Debugger message may be quite cryptic: the Debugger backtrace will indicate only the function which punted, without explicit information as the precise violation (this is usually manifested as a "segmentation violation"). The key in these cases is to `DESCRIBE` the offending function in order to see what it expects, or if necessary examine the function source code (which will be indicated by `DESCRIBE`).

3.3 Unix and Lisp Environment

In general, references in this manual to the Unix environment will assume that csh is the shell. Note that Linux implementations usually use bash instead. In any event, appropriate modifications should be made according to the shell in use.

We shall assume that the Surf-Hippo system is installed under a directory called "surf-hippo" - this directory may be referred to as either the top-level Surf-Hippo directory or the Surf-Hippo home directory. Note that the name is arbitrary; it only must match the setting of the SURFHOME environment variable (see Section 4.10). By default, unless the full pathname is given, in this manual any references to directories will assume that they are under the Surf-Hippo home directory. That is, "...(as found in circuits/)..." refers to the directory "surf-hippo/circuits/". Likewise, filenames without any directory reference are assumed to be in the surf-hippo/src/sys/ directory (the system source file directory) (see also Section 30).

3.4 The Emacs Ilisp Environment

Although Surf-Hippo may be run directly from a Unix terminal window, it is *much* more convenient to use the ILisp mode under Emacs. Likewise, it is really essential to become familar with at least the basic Ilisp commands, for example for loading a file, compiling an expression, and so forth. Here are just a small selection of the commands available under Ilisp mode:

`C-z H`	hyperspec-lookup
`C-z l`	load-file-lisp
`C-z s`	status-lisp
`C-z c`	compile-defun-lisp
`C-z C-b`	ilisp-compile-buffer
`C-z M`	macroexpand-lisp
`C-z d`	documentation-lisp
`C-z i`	describe-lisp
`C-z )`	find-unbalanced-lisp
`C-z ;`	comment-region-lisp

See Sections 38 and 39.3 for more information on Ilisp.

4 Running Surf-Hippo

We will now describe the basics for running Surf-Hippo. We assume that Surf-Hippo is loaded and running, and you are communicating with Lisp via the shell or Emacs ILISP buffer that is displaying the Lisp prompt. If not, refer to Section 38.

4.1 Surf-Hippo Menus vs. the Top Level Lisp Interpreter

An important concept is that interactions with Surf-Hippo may be either via commands (that is Lisp functions) entered directly into the Lisp interpreter or via the Surf-Hippo menu system, and that one may switch back and forth between the menus and the top level as needed. In general, there is no loss of "state" when switching between the modes, until you load in a new circuit.

The Main Menu loop is invoked by the function `SURF`, and starts off with the "Surf-Hippo Main Menu". While this loop is running, the interpreter is essentially "turned off" - there will be no response when typing something into the Lisp window. Hitting OK from the "Surf-Hippo Main Menu" with no other selections returns control to the interpreter.

Menu-based operation is sufficient for exploiting much of Surf-Hippo, assuming that the appropriate circuit descriptions (from files or functions) are available. This means that the system may be used with only a rudimentary knowledge of Lisp. For example, to run a simulation from the menus you click the "Run simulation (immediately)" option on the "Surf-Hippo Main Menu".

In contrast, the full flexibility of the system is realized when the user can write his or her own Lisp code and switch back and forth between interaction modes as needed. Using Surf-Hippo in interpretive mode, either by typing functions directly to the Lisp window or by loading Lisp files (that include the appropriate functions), is necessary for parametric simulations, typically centered on running individual simulations with the function `GOFERIT`:

goferit &optional *(stop−time *user−stop−time*)* [*Function*]

The execution of `GOFERIT` will run the simulation for STOP-TIME milliseconds - if the optional STOP-TIME argument is omitted then the current value of the global variable `*USER-STOP-TIME*` will be used.

4.2 Running Surf-Hippo From The Menus

Now run the Surf-Hippo main menus by entering:

```
* (surf)

 ** The Surf-Hippo Neuron Simulator, Version X.X **
```

The "Surf-Hippo Main Menu" should now appear. If not, one possibility is that the X display Unix environment variable is pointing to another screen (see Section 40.4).

Start out by simulating a hippocampal action potential with the buit-in function `WORKING-HPC`. The following sequence assumes that you are starting Surf-Hippo from scratch (exit each menu by hitting "OK"):

1. From the "Surf-Hippo Main Menu", select "Overall parameters, load circuit or files"
2. Select "Load circuit function/file"
3. Select "Catalog Function"
4. Select "WORKING-HPC"

The Lisp window will show something like:

```
Reading in circuit WORKING-HPC...
Locating segments...

 Cell HPC: 0 branch points and 5 segments processed.
```

And the "Surf-Hippo Main Menu" will reappear, now with the loaded circuit displayed at the bottom of the menu. To draw the cell:

1. Select "Histology"
2. Just hit OK from the "Setting Up Cell Drawing" menu

After the histology window appears, the "Surf-Hippo Main Menu" will reappear. Click "OK" to return to the Lisp command line. The following message will then appear in the Lisp window:

```
Do you want to quit Lisp? (RETURN for NO, yes/YES for YES):
```

Answering in the affirmative (not now!) will close Lisp elegantly. Now set up the plotting and stimulus. You can do this by point-and-clicking on the histology window, or from the Main Menu loop, but for now, enter the following from the command line:

```
* (add-isource *soma*)
<Current Source HPC-soma-isrc>
* (pulse-list *isource* '(10 25 1))
NIL
* (setq *user-stop-time* 100)
100
* (enable-element-plot *soma*)
NIL
* (enable-element-plot *isource*)
NIL
```

Now go back to the menus by typing the function:

```
* (surf)
```

and select "Run simulation (immediately)". When the simulation is finished the soma voltage and the current stimulus will be plotted in various plotting windows. The "Surf-Hippo Main Menu" will then reappear. The main menu has a "Information management" option. This is a convenient place to save (archive) the results of important simulations to files. Simulations may be repeated on the loaded cell description, or a new circuit may be specified.

The loaded circuit may be modified however you want with the appropriate menus, clicking the OK button when done with each menu. For example, membrane elements including sources, channels, and synapses may be added to the loaded cell(s) by choosing locations from a histology window (LEFT mouse), and then bringing up a menu for the chosen location (CONTROL-SHIFT-LEFT mouse).

4.3 Pre-defined and User-defined Global Variables

As mentioned, global variables (usually represented by symbols that begin and end with `*`) are referenced throughout the simulation environment. There are on the order of 3000 pre-defined global variables (the important ones are listed in the Reference Manual, and some are described in this manual). Normally, you should only need to work with the documented global variables in your code, for example changing their values with `SETQ`. Often, the menu system will allow setting a global variable, and in these cases the actual variable name will be shown in the menu to facilitate knowing which is the appropriate variable to reference in explicit user-written code.

To define a new global variable, you first should make sure that the symbol that you want to use has not already been defined by the system, and for this you can use the `DESCRIBE` function on the (quoted) symbol (you must use the quote so Lisp acts on the symbol itself, not on any value assigned to it) . Thus, suppose you want to use the symbol `*FOO*` as a global variable in your own code. If `*FOO*` was not already assigned, then `DESCRIBE` will say only that the symbol is indeed a symbol:

```
* (describe '*foo*)
*FOO* is an internal symbol in the SURF-HIPPO package.
```

You can then use the DEFVAR form to define your variable and its initial value (note that here you *do not* include the quote):

```
* (defvar *foo* 1.23)
*FOO*
```

Now if you apply DESCRIBE on the symbol *FOO*, DESCRIBE will reflect the new assignment:

```
* (describe '*foo*)
*FOO* is an internal symbol in the SURF-HIPPO package.
It is a special variable; its value is 1.23.
   1.23 is a SINGLE-FLOAT.
```

If you try to use DEFVAR on a symbol that was already assigned to a global variable, it will have no effect.

Surf-Hippo keeps track of any variables defined by the user. For example, as soon as any new variables are defined, the main menu will include an option to allow editing these variables. You can get this menu directly by calling the function GLOBAL-VARIABLE-MENU. This menu will include any documentation string, if defined, associated with each variable. Of course, you can also change any global variable explicitly with the SETQ form:

```
* (setq *foo* 2.34)
2.34
```

Note that SETQ returns the new value, while DEFVAR returns the new symbol. One limitation of the automatic menu for editing user-defined globals is that the allowed data type (number, string, or boolean) for each variable in the menu is inferred by the current value of the variable. Thus, if you want to set a new value for some variable that is of a different type than the current value, you must use SETQ.

User-defined global variables may also be documented automatically - see Section 23.1.

4.4 Some More Examples

One can try the other circuit functions that are preloaded as part of the "Catalog Function" list mentioned above. For example, to see a moderately complicated simulation, try the built-in function STAR-AMACRINE-DS.

In addition, there are several circuit model files in the surf-hippo/circuits/demos/ directory, which on the most part can be CIRCUIT-LOADed (see Section 5.3) directly and run (some include a GOFERIT, so the simulation runs on its own). For a colorful example, try:

```
* (circuit-load "/usr/local/surf-hippo/circuits/demos/colorized-n120.lisp")
```

For other cells with complete detailed anatomy, try loading files from the surf-hippo/anatomy/ directory (the colorized-n120.lisp file references an anatomical file here). For information about how these sorts of circuit files are made, see Section 8. Some of these files define a circuit when they are loaded; others must be loaded, and functions defined by them must then be invoked to actually load the circuit. See Section 5 for more information on building and adding to circuits.

4.5 Result Output

Unless the global variable *KILL-ALL-OUTPUT* is T [default NIL], at the beginning and end of each simulation, various circuit and simulation information may be printed out to the Lisp Listener (default), a special Information Window, or a file. For file documentation, see Section 23. Note that *KILL-ALL-OUTPUT* will also suppress data plotting and automatic data file writing.

4.6 Loading Circuit Definitions

Circuits are defined by either compiled functions, files (see Section 5), or combinations which can in turn reference either functions or files (e.g. compiled functions which load files). Try simulating the circuit file file-hippo.lisp (circuits/demos/ directory) by loading it from the menu sequence as before, but choosing a file as the circuit definition source.

In general, more complicated circuits should be put into a DEFUN form and compiled from Lisp, or the file that defines the cell should be compiled and the resulting binary version (e.g. .sparcf or .fasl file) loaded when loading the circuit.

For Lisp, since they are interpreted as symbols function names may be entered in mixed case (e.g. working-hpc, WORKING-hpc, working-HPC, etc.), while for file names case matters. For example, for the circuit file "fOo", the file name must be entered as "fOo", not "FOO" or "foo", etc. If you enter in a function name that does not exist, the following error will be generated:

```
Reading in circuit bogus-function-name...

Error in KERNEL:%COERCE-TO-FUNCTION:  the function BOGUS-FUNCTION-NAME is undefined.

Restarts:
  0: [ABORT] Return to Top-Level.

Debug  (type H for help)

(KERNEL:%COERCE-TO-FUNCTION BOGUS-FUNCTION-NAME)
0]
```

If this happens, simply enter Q or 0 (ABORT) to the Debugger, and then enter (surf) to start things off again.

4.7 Element Library Parameters

Parameters for specific membrane element models (channels, synapses, etc.) are generally found in the src/parameters/ directory. Many of these parameters are taken from other published models. There are no guarentees on the correctness of any of the parameters listed here; thus the original references should always be checked. On the other hand *numerous* errors in the originals have been found - see comments in the various parameter files. The general concepts behind element parameter libraries are described in Section 11.

4.8 Interrupting The Simulation

It is often useful, especially for long simulations, to check things out in the middle, or to gain access to (or perhaps simply refresh) the output windows. As above, you can BREAK into the running simulation by typing C-c C-c into the Lisp window. If desired, from the Lisp Debugger, enter the function SIM-OUTPUT to see the result obtained so far:

sim-output [*Function*]

For example:

```
            .
            .
            .
Starting transient solution
```

Enter C-c C-c →

```
Interrupted at #x716DEB8.

Restarts:
  0: [CONTINUE] Return from BREAK.
  1:            Return NIL from load of "/usr/local/surf-hippo/src/visual/j43ds.lisp".
  2: [ABORT   ] Return to Top-Level.
```

```
Debug  (type H for help)

(UNIX::SIGINT-HANDLER #<unavailable-arg> #<unavailable-arg> #.(SYSTEM:INT-SAP #xEFFFEC68))
0] (sim-output)
Total time points/iterations 13180/20379

Simulation duration: 200.0 ms
File /usr/local/surf-hippo/data/j43d-11-5/11_12_1994/j43d-11-5-731552.info written
Done.

NIL
0] 0
```

Typing 0 to the Debugger resumes the simulation. Note that, according to the "Restart" options, "0" was entered to `CONTINUE` from the `BREAK`, as opposed to returning (`ABORT`) to the Lisp top level.

Another function which may be useful to find out the progress of the simulation is:

print-simulation-stats &optional *complete* [*Function*]

Here, various numerical parameters will be printed out as well if the optional COMPLETE argument is T.

4.9 Customized Setup

If there is a file named `customs.lisp` in either the user directory (set by the shell variable SURFUSER-HOME), or in the Surf-Hippo lib directory (a subdirectory of the top level Surf-Hippo directory, by default set by the shell variable SURFHOME), then this file is loaded during the Surf-Hippo initialization. This lisp file can contain various declarations, function definitions, or directives for loading other files as needed to define your particular environment. Another file that is automatically loaded during initialization, if it exists, is `patches.lisp`, found in the Surf-Hippo lib directory. This file can hold various bug fixes that may be supplied for a particular version of the code.

Check Section 38 for more information on the setting of various environment variables.

4.10 Referencing the Surf-Hippo Home Directory and Others

For loading a file that is somewhere under the top-level Surf-Hippo directory, you can use:

load-surf-home-file *filename* [*Function*]

This function loads a FILENAME which must be in the Surf-Hippo home directory (as specified by the global variable `*SURF-HOME*`, which in turn is defined initially by the SURFHOME enviroment variable). For example, suppose the Surf-Hippo home was "/usr/local/surf-hippo". In that case, then

```
(load-surf-home-file "src/hippocampus/warman-durand-yuen.lisp")
```

will load the file "/usr/local/surf-hippo/src/hippocampus/warman-durand-yuen.lisp". A related function is (for a file under the Surf-Hippo user directory):

load-surf-user-file *filename* [*Function*]

where the user directory is specified by the global variable `*SURF-USER-HOME*`, which in turn is initially defined by the HOME enviroment variable.

Finally, the global variable `*SURF-USER-DIR*` defines the default path where simulation data is stored. This variable is initially set by the value of the SURFUSERHOME enviroment variable, if defined; otherwise, `*SURF-USER-DIR*` is set to the HOME enviroment variable. Once `*SURF-USER-DIR*` is defined, then the subdirectories "data" and "plot" are automatically created (see also below for the `system-variables.lisp`

file. Note that if SURFUSERHOME has the same value as SURFHOME, then the relevant data and plot subdirectories are found along with the standard Surf-Hippo directories (see Section 30.3).

4.11 Quitting Lisp

To quit Lisp, you can answer affirmatively when exiting the main menu loop, as above, or you can use the function:

```
* (quit-sh)
```

If necessary, typing C-c C-c into the Lisp window will usually interrupt Surf-Hippo, and put you into the Debugger. Avoid doing this in the middle of a GC, however, because this hang the system. Alternatively (from ILISP), M-x panic-lisp, followed by C-c C-c, may be necessary.

4.12 Saving and Restoring State

At the end of a Surf-Hippo session, that is when you quit Lisp, a file named `system-variables.lisp` is automatically written which captures the values of several global variables (given by the value of `*SYSTEM-VARIABLES*`). This file is written in the `lib` subdirectory under the current value of `*SURF-USER-DIR*`, and of course this file may be edited manually. When the global variable `*READ-SYSTEM-VARIABLES-FILE-ON-STARTUP*` is true (the default), the variable values in this file are restored at the initialization of the next Surf-Hippo session.

5 Circuit Definitions - Functions and Files

See also Section 19.

5.1 What Makes A Circuit

A "circuit" is the electrical/chemical representation that defines the differential equations simulated by the program. Normally, a circuit describes one or more neurons, each of which has a soma, and, perhaps, a dendritic tree consisting of compartments called segments, and a variety of membrane mechanisms may be included with any anatomical compartment.

In Surf-Hippo, all individual parts of the circuit are called "elements", each of which is represented by a specific type of Lisp structure object.

References in the documentation to "cell elements" mean `SOMA` and `SEGMENT` structures, and "membrane elements" `CHANNEL`, `SYNAPSE`, `PUMP`, `BUFFER`, `ISOURCE`, `VSOURCE`, `CONC-INT` (concentration integrators) structures (and others to be defined). Many elements in a circuit have an associated element "type", including channel types for channels, synapse types for synapses, etc. The element type associated with segments and somas is a cell type (and each soma and segment is associated with a cell structure which is itself an "element"). Each loaded element in the cirucit The element system is described further in Section 6.

Circuit are defined by a series of Lisp forms (functions or macros) that may be grouped together in either some user-defined function, or in a Lisp file (or files).

5.1.1 Somas and Segments

The `SOMA` circuit object in Surf-Hippo is represented as a single node with a capacitance and (leak) resistance connected to ground, with the values of these elements calculated under the assumption of a spherical anatomical form. Furthermore, the method for evaluating the electrical circuit of a cell requires that the root of the tree topology is a `SOMA` circuit object. On the other hand, the soma diameter may be 0 (see for example Section 8.5.1).

The `SEGMENT` circuit object in Surf-Hippo is represented as a single node with a capacitance and (leak) resistance connected to ground, and a linear resistance connected to the node of either another segment or the soma (naturally leading to a tree topology). The values of these elements are calculated under the assumption of a cylindrical anatomical form. A given cell may or may not have segments.

5.1.2 Defining a Neuron Geometry by Anatomy Files or Explicit Cell Element Creation

Cell morphology may be established by loading in a converted anatomy file (Section 8), or by explicit functions that create cell elements (Section 7), including:

```
CREATE-SOMA (necessary)
CREATE-CELL (usually taken care of by CREATE-SOMA)
CREATE-SEGMENT, CREATE-TREE, SEGMENT-CHAIN, TREE-CONTROL (optional)
```

Thus, all cells require at least a call to `CREATE-SOMA.` Cell types and cells may be explicitly defined, if necessary, with calls to `CREATE-CELL-TYPE` and `CREATE-CELL`. Dendrite geometries may be defined one segment at a time using `CREATE-SEGMENT`, or by using `CREATE-TREE` or `SEGMENT-CHAIN.` These functions are described in Section 7.

5.1.3 Processing the Circuit Structure

Once all the cell elements are created, further processing is necessary to complete the circuit's geometry. This is accomplished by

process-circuit-structure &optional *force circuit−element−to−update* [*Function*]

which is normally called automatically. However, this function may also be called explicitly from within a circuit definition, which allows subsequent morphologically-based references to the new cell elements, for

example placing synapses onto segments according to some function of the spatial coordinates of the dendritic segments.

5.1.4 Adding Membrane Elements to Segments and Somas

After the cell morphology has been defined (and, if appropriate, after an explicit call to PROCESS-CIRCUIT-STRUCTURE), the membrane elements may be added to the cell elements. The most general function for adding membrane elements (e.g. channels, synapses, sources, etc.) to the cell elements is with the function CREATE-ELEMENT (Section 6.2). Loaded circuits may also be modified using a histology window (see Section 25); cell segments or somas may be selected with the mouse and various membrane elements may be added/deleted/edited as desired.

5.2 Loading the Circuit Defining Function or File

Loading a circuit function or a circuit file are options given by the circuit input menu sequence. Otherwise, from the interpretor, one uses the general circuit loading macro CIRCUIT-LOAD (described below), which can handle circuits defined by files or functions.

5.3 CIRCUIT-LOAD - The Basic Wrapper for Circuit Loading

The CIRCUIT-LOAD macro is used for loading circuit descriptions:

circuit-load &body *body* [*Macro*]

Normally CIRCUIT-LOAD initializes and clears everything before loading a circuit, unless it is called recursively (see below) or if the global variable *INITIALIZE-ON-CIRCUIT-LOAD* is NIL (default T). CIRCUIT-LOAD tries to figure out what the forms in BODY refer to, which could include a file name, the name of a compiled function, or a set of explicit lisp forms that define a circuit. The global variables *CIRCUIT-SOURCE*, *CIRCUIT-FILENAME* and *CIRCUIT* are set accordingly, and the circuit is loaded. If the BODY refers to a compiled function, then this can be either a symbol, e.g.:

```
* (circuit-load 'working-hpc)
```

a string, e.g.:

```
* (circuit-load "working-hpc")
```

or a regular Lisp form (e.g. a function form with appropriate arguments):

```
* (circuit-load (working-hpc :name "foo"))
```

If BODY refers to a file, then this must be a pathname string, e.g.:

```
* (circuit-load "/home/bozo/surf-hippo/circuits/a-channels-test.lisp")
```

Thus, in this case CIRCUIT-LOAD behaves analogously to the Lisp function LOAD, but with the (possible) initialization of the loaded circuit parameters.

The C shell "~/" convention for the user's home directory will be parsed by CIRCUIT-LOAD (note, but not by the standard Lisp function LOAD): thus, if the home directory (given by the global variable *SURF-USER-HOME*) is "/home/bozo/", then the preceeding command could also be:

```
* (circuit-load "~/surf-hippo/circuits/a-channels-test.lisp")
```

CIRCUIT-LOAD can also reference a path that is implicitly under, in order of precedence, the current working directory (which is set on start up by the value of *SURF-HOME*) or the value of *CIRCUIT-DIRECTORY*. Thus, assuming that *SURF-HOME* was equal to "/usr/local/surf-hippo/" (and had not been changed since startup), then the following are equivalent:

```
* (circuit-load "/usr/local/surf-hippo/circuits/demos/file-hippo.lisp")
```

and

```
* (circuit-load "circuits/demos/file-hippo.lisp")
```

The value of *CIRCUIT-DIRECTORY* is automatically set to the directory of the last circuit file (if any). See also Section 19. As stated, CIRCUIT-LOAD can also work on Lisp forms directly. Here the BODY of CIRCUIT-LOAD is a LOOP form that defines two cells:

```
(circuit-load
 (loop for i from -2 to 2 by 4
       for name in '(a b) do
       (let ((*cell-name-suffix* name)))
        (move-cell (dead-hippo) (list (* i 100.0) 0.0 0.0))))))
```

In this case the function DEAD-HIPPO returns the created cell which is then moved by MOVE-CELL (see Section 7.2) to the coordinates given by the second LIST argument. Unique cell and segment names are ensured by local binding of the global variable *CELL-NAME-SUFFIX*, (see Sections 7.1, 8.2 and 29.4).

CIRCUIT-LOAD may also be used recursively. This may be useful when a part of the circuit definition is given in an anatomy file, or a circuit is composed of several other files which might include CIRCUIT-LOAD forms so that they may be used independently. For example, a complete circuit function could be:

```
(defun adp-c12861 ()
  (circuit-load "anatomy/misc/c12861.ca1.lisp")
  (std-setup)                          ; Add somatic current source and plot soma voltage.
  (pulse-list *isource* '((1.0 2.9 1.0) (2.9 3.30 -1.2))) ; Set two pulse stimulus.
  (setq *absolute-voltage-error* 0.001 ; We want high resolution for these traces.
        *save-data-step* 1             ; See every step of the simulation.
        *user-stop-time* 12.0))
```

Here, the call to CIRCUIT-LOAD loads a compiled circuit anatomy file that was generated by ntscable and then edited to set both the cell type and name to "c12861.ca1" (see Section 8.2). Files generated by ntscable and then loaded by CIRCUIT-LOAD (or by the input circuit menu) create cell soma(s) whose names are the concatenation of the cell name and "-soma". Thus, in this case a soma named "c12861-soma" is created. The function STD-SETUP (called here without its optional ELEMENT argument) adds a current source to the soma, and causes the soma voltage to be plotted (see also Section 20). The current source (referenced here automatically by the global variable *ISOURCE*) stimuluation is set by PULSE-LIST (Section 10). Finally, the values of a few global variables are set to fine tune the simulation. Note that once this function is defined, you could CIRCUIT-LOAD it directly (again, since CIRCUIT-LOAD may be called recursively):

```
* (circuit-load 'adp-c12861)
```

The same circuit could be accomplished by writing a Lisp file, let us say "/home/your-name/adp-c12861.lisp", that includes:

```
(circuit-load "anatomy/misc/c12861.ca1.lisp")
(std-setup)                          ; Add somatic current source and plot soma voltage.
(pulse-list *isource* '((1.0 2.9 1.0) (2.9 3.30 -1.2))) ; Set two pulse stimulus.
(setq *save-data-step* 1             ; See every step of the simulation.
      *user-stop-time* 12.0)
```

Reading in this file, for example via the circuit loading menu with the circuit file browser, or from lisp using CIRCUIT-LOAD:

```
* (circuit-load "/home/your-name/adp-c12861.lisp")
```

would be the same as calling the function ADP-C12861 defined above.

5.3.1 CIRCUIT-LOAD Always Initializes the System

Unless the global variable *INITIALIZE-ON-CIRCUIT-LOAD* is T, any loaded circuit is cleared by calling the function INITIALIZE-GLOBALS-FOR-CIRCUIT, even if BODY is not included or is NIL. Among other things, this will remove any current circuit elements.

5.4 Menu Entry of Circuit Functions

A circuit function may be loaded from the load circuit menu sequence by either typing in the name or choosing a name that has been added to the circuit function catalog, *CIRCUIT-FUNCTIONS*. For example, suppose that you include in your system loading sequence a file that defines a circuit function:

```
(defun interneuron-234 ()
  (create-cell-type "interneuron-234")
  ...
   )
```

If your load files include the statement:

```
(push 'interneuron-234 *circuit-functions*)
```

then the circuit loading menu will include this choice. Multiple functions may be selected from this menu, and all will be loaded to make up the simulated circuit. In this case, an additional menu will appear that asks for a name of the new circuit.

Note that the same thing may be achieved by defining a function which then calls the composite circuit functions. For example, suppose that the following functions have been defined and listed in *CIRCUIT-FUNCTIONS*:

```
(STAR-AMACRINE-DS N120-MAX-RED THREE-HIPPOS DEAD-HIPPO WORKING-HPC)
```

In the menu, clicking on STAR-AMACRINE-DS and THREE-HIPPOS will load those two functions and the simulated circuit will be made up of both. On the other hand, you could:

```
(defun star-dead ()
  (STAR-AMACRINE-3)
  (DEAD-HIPPO))
```

CIRCUIT-LOADing STAR-DEAD by itself will do the same thing.

If a circuit function to be loaded is typed in, then the function name is also added to the function catalog (for the current session only).

5.5 Restrictions on Duplicate Circuit Element Names

The same name may be assigned to two different circuit elements, but only if they are not the same type (e.g. two segments, two somas, two channels, etc.), no matter whether they are part of the same cell or different cells. The simplest way to avoid duplicate names is to ensure that all circuit element names are consecutive integers (the default - see below Section 5.6). Otherwise, for circuits that include more than one instance of a cell description, it is normally sufficient to assign a unique name to each cell, since element names (if not explicitly defined) are derived from the name of the cell of which it is a part. For example, if two HIPPO based functions are clicked in the circuit catalog function menu, an error occurs as soon as the second incarnation of the HIPPO cell is loaded:

```
  Reading in circuits DEAD-HIPPO_BASIC-HIPPO...

  Error in function HIPPO:
create-soma: soma Hippo-soma  already defined, ignoring

  Restarts:
    0: [CONTINUE] continue
```

```
    1: [ABORT   ] Return to Top-Level.

Debug  (type H for help)

(HIPPO "Hippo" :SYNAPSE-TYPES NIL :SYNAPSE-SEGS ...)
Source:
; File: /usr/local/surf-hippo/src/hippocampus/hippos.lisp
(CREATE-SOMA :CELL CELL-NAME :DIAMETER SOMA-DIAMETER)
0] q
```

Here, the solution would be to explicitly define a new function that calls the two desired cell functions, with, however, unique cell names associated with each:

```
(defun dead-basic-hippos ()
  (dead-hippo "Dead-Hippo")
  (working-hpc))
```

One advantage of using automatically generated strings for element names (as the case when *USE-SIMPLE-NAMES* is NIL, the default - see below) is that when passing a name to the ELEMENT function (and all those that depend on it; see Section 6.1), the optional MODEL-TYPE argument usually won't be needed. Otherwise, with integer simple names, passing an integer (name) to ELEMENT with specifying the model type will return the first element with that integer, in priority given by:

```
(SOMA SEGMENT CHANNEL SYNAPSE CELL BUFFER PUMP ISOURCE CONC-INT PARTICLE
 CONC-PARTICLE AXON VSOURCE EXTRACELLULAR-ELECTRODE ISOURCE-TYPE VSOURCE-TYPE
 BUFFER-TYPE PUMP-TYPE CHANNEL-TYPE CONC-INT-TYPE PARTICLE-TYPE
 CONC-PARTICLE-TYPE SYNAPSE-TYPE AXON-TYPE CELL-TYPE NODE ELECTRODE)
```

5.5.1 Duplicate Elements on Same Node - Naming Membrane Elements

If automatic name string generation is enabled, and you try to create a duplicate membrane element on the same soma or segment, a message like:

```
CREATE-SYNAPSE: synapse Hippo-3-WILD already defined
```

will be printed and the element creation function will quit. In some cases, however, it may be useful to have duplicate instances of the same type of membrane element on the same node, for example light synapses (e.g. with different RFs). In these cases, if the global variable *PROMPT-FOR-ALTERNATE-ELEMENT-NAMES* is T (the default), then the user is prompted before the additional element is created - otherwise, the element is automatically created with a new name that is created from the standard name with an numeric extension (see the function CHECK-ELEMENT-NAME).

Specifically, synapses are created with the CREATE-SYNAPSE function (see Section 15.3). Referring to the arguments of this function, duplicate synapses are allowed if the POST-SYNAPTIC-ELEMENT already has a synapse of the same type, and the PRE-SYNAPTIC-ELEMENT is either different or the synapse type has no PRE-SYNAPTIC-ELEMENT.

If *ALLOW-DUPLICATE-SYNAPTIC-CONNECTIONS* is T (default), then more than one synapse of the same type may be created with the same pre and post synaptic element.

5.6 "Simple" (Numbered) Element Names

While in principle every created circuit element may be explicitely named, normally one of two automatic name generating algorithms are used, according to the global variable *USE-SIMPLE-NAMES*. When *USE-SIMPLE-NAMES* is T, circuit element are named with integers (unique sequence for each element type).

If *USE-SIMPLE-NAMES* is NIL, name strings for each element will be generated which include information to tell where and what the element is by inspection. However, for large circuits these names can use significant memory. Previously created (and named) circuit elements can be renamed using this scheme by a set of RENAME-xx-SIMPLE functions, where xx stands for SYNAPSES, CHANNELS, PARTICLES, CONC-PARTICLES, AXONS, PUMPS and BUFFERS.

5.7 Recommended Strategy for Defining Cells

The safe way to build a cell is to *first* create the cell with the `CREATE-CELL` function, and then use the returned cell object (or the name of the returned cell object) as the cell reference for subsequent cells to `CREATE-SOMA` and `CREATE-SEGMENT` (or `CREATE-TREE`, etc.). In this way, if the cell-building code is used as part of a multiple cell circuit (even when the multiple cells are copies of the same cell structure, that is each reuse the same code), if a call to `CREATE-CELL` requires an automatic modification of the requested name, then the later calls to the other cell elements will still have a target cell to work with.

This can be done by locally binding the created cell to a local variable, and then passing that local variable to subsequent functions, e.g.:

```
(let ((cell (create-cell CELL-NAME ...)))
  (create-soma :cell cell ...)
  (create-tree cell tree-list ...)
  ...)
```

Or, the call to `CREATE-CELL` can be directly embedded within other functions, e.g.:

```
(create-tree
 (create-soma :cell (create-cell CELL-NAME ...) ...)
 tree-list ....)
```

5.8 Adding or Subtracting Cells, Somas or Segments to a Loaded Circuit

Cells, somas or segments may be added to a `CIRCUIT-LOAD`ed circuit if desired. However, the function `PROCESS-CIRCUIT-STRUCTURE` must be run after the new elements are created to complete their initialization. Note under most circumstances this function will be called automatically when simulating either a new or modified circuit.

6 Elements in the Circuit

A key concept in Surf-Hippo is that of being able to access any *element* of the circuit by either the name of that element, or with a pointer to the element structure. The former case is normally appropriate when there is an explicit literal reference to a some element, for example when you must supply the element reference by actually typing it. The latter case, although more abstract from the point of view of the user, is often appropriate since many functions return element structure pointers. In any event both methods are employed internally as necessary.

6.1 The ELEMENT Function

The basic function for accessing circuit elements is:

element *elt−reference* &optional *model−type fast* [*Function*]

This function takes either an explicit pointer to a structure or the name of a structure, and returns a pointer to a structure. If ELT-REFERENCE is a name, which may be a string, number or symbol, the mapping is not necessarily unique since it is legal for two elements of different types to have the same name. Thus, the type of the returned data structure is either MODEL-TYPE, if included, or chosen from all model types according to the following precedence:

```
NODE SEGMENT SOMA CHANNEL SYNAPSE CELL BUFFER PUMP ISOURCE CONC-INT PARTICLE
CONC-PARTICLE AXON VSOURCE EXTRACELLULAR-ELECTRODE BUFFER-TYPE PUMP-TYPE
CHANNEL-TYPE CONC-INT-TYPE PARTICLE-TYPE CONC-PARTICLE-TYPE SYNAPSE-TYPE
AXON-TYPE CELL-TYPE ELECTRODE
```

The ELT-REFERENCE argument to ELEMENT may be either a list or an atom (that is, a single string, symbol or number). If ELT-REFERENCE is a list, a list of all element structure pointers associated with all the atoms in the list is returned. If ELT-REFERENCE is an atom, just the single element structure assocciated with the atom is returned. In general, the functions which are based on ELEMENT have a similar option, that is they operate on and possibly return a list of circuit element structures if they are given a list argument, otherwise they do the same on an individual circuit element. For example (see Section 6.3):

```
* (segments)
(<Segment Hippo-3: prox node Hippo-2>
<Segment Hippo-2: prox node Hippo-1>
<Segment Hippo-1: prox node Hippo-soma>
<Segment Hippo-5: prox node Hippo-4>
<Segment Hippo-4: prox node Hippo-3>)
* (element "Hippo-1")
<Segment Hippo-1: prox node Hippo-soma>
* (element (list "Hippo-2" *soma* "Hippo-1"))
(<Segment Hippo-2: prox node Hippo-1>
<Soma Hippo-soma>
<Segment Hippo-1: prox node Hippo-soma>)
*
```

In almost every case in which a documented Surf-Hippo function operates on a circuit element or elements, the ELEMENT function is used internally to convert the element-specifying argument to the actual element data structure or structures. Exceptions include data structure accessor functions, which are automatically created when an data structure is defined, and in those cases when the function documentation specifies "...(requires explicit structure)".

6.2 Generic Membrane Element Creation: The CREATE-ELEMENT Function

The following function:

create-element *thing* &rest *others* [*Function*]

is the generic create function for elements added to somas or segments, including membrane elements (channels, synapses) and other circuit elements (sources, axons, etc.). This function takes any number of arguments (this is what the &rest keyword, followed by the dummy variable others means), and considers all the atoms in the arguments. Given any atom which references an element type, an element of that type is added to all the cell elements which are referenced in the arguments. If there are no references to cell elements, then any element types referenced in the arguments that do not already exist are created. All created elements or element types are returned as a list if more than one, or as an atom if only one. For example:

```
* (create-element 'NA-hh)
<Channel Type NA-HH>

* (create-element 'NA-hh *soma*)
<Channel Hippo-soma-NA-HH: type NA-HH>

* (create-element 'NA-hh *soma* 'DR-hh)
(<Channel Hippo-soma-NA-HH: type NA-HH>
 <Channel Hippo-soma-DR-HH: type DR-HH>)
```

If the keyword `:NO-DUPLICATES` is included in the arguments, then no duplicate elements (for example the same synapse type on the same cell element) will be created.

Note that all elements have their own specific `CREATE-*` functions, whose more detailed argument options may be necessary in certain cases.

6.3 Referencing Circuit Elements of a Given Class

6.3.1 Referencing the Most Recently Created Elements

Each of these global variables point to the most recent created instance of the given model type:

```
*SOMA* *SEGMENT* *CELL*
*VSOURCE* *ISOURCE* *AXON*
*SYNAPSE* *CHANNEL* *PUMP* *BUFFER*
*ELECTRODE* *PARTICLE* *CONC-PARTICLE*
*CELL-TYPE* *AXON-TYPE* *SYNAPSE-TYPE*
*CHANNEL-TYPE* *PUMP-TYPE* *BUFFER-TYPE*
*PARTICLE-TYPE* *CONC-PARTICLE-TYPE*
```

If the element referenced by one of these variables is destroyed, then that variable is updated to the next most recently created instance.

6.3.2 Referencing all Elements

A series of functions, including:

nodes &optional *element* [*Function*]

as well as

```
SEGMENTS, SOMAS, CELLS, CELL-TYPES, ISOURCES, VSOURCES, SYNAPSES,
CHANNELS, SYNAPSE-TYPES, CHANNEL-TYPES, PARTICLES, CONC-PARTICLES,
PARTICLE-TYPES, CONC-PARTICLE-TYPES, CONC-INT-TYPES, CONC-INTS, PUMPS,
PUMP-TYPES, BUFFERS, BUFFER-TYPES, AXONS, AXON-TYPES, ELECTRODES,
EXTRACELLULAR-ELECTRODES
```

return a list of all the nodes (segments, etc.) associated with ELEMENT, which may be a cell element (soma or segment) or a cell, or an element type. If ELEMENT is not included, then a list of all nodes (segments, etc.) in circuit is returned. For example:

```
* (conc-particle-types)
(<Conc Particle Type KCTX-HPC-CA>
 <Conc Particle Type KAHPO-HPC>)
* (conc-particles)
(<Conc Particle HPC-soma-KCT-HPC-KCTX-HPC-CA: type KCTX-HPC-CA>
 <Conc Particle HPC-soma-KAHP-HPC-KAHPO-HPC: type KAHPO-HPC>)
* (conc-particles 'KCTX-HPC-CA)
(<Conc Particle HPC-soma-KCT-HPC-KCTX-HPC-CA: type KCTX-HPC-CA>)
* (conc-particles *segment*)
NIL
* (conc-particles *soma*)
(<Conc Particle HPC-soma-KAHP-HPC-KAHPO-HPC: type KAHPO-HPC>
 <Conc Particle HPC-soma-KCT-HPC-KCTX-HPC-CA: type KCTX-HPC-CA>)
```

6.4 "Cell Elements" are Somas or Segments

All functions which refer to a soma or segment, for example as a destination for a newly created channel or synapse, allow the CELL-ELEMENT (soma or segment) argument to be either the pointer to the actual Lisp data structure or, as is often more convenient especially for calling functions from the interpreter, by the name of the CELL-ELEMENT. In general, the translation for all circuit elements (including CELL-ELEMENTs) is with the `ELEMENT` function. Thus, suppose we want to add a type `NA-HH` channel to a dendritic segment (compartment) that is called "Hippo-3". This could be done with

```
(create-channel "Hippo-3" 'na-hh)
```

or

```
(create-channel (element "Hippo-3") 'na-hh)
```

Obviously the latter case is overspecified - the point is that there will be situations where it will be more convenient to pass the structure pointer to a function like `CREATE-CHANNEL`, rather than a structure name.

In fact, in most cases functions with element arguments are written so that the element structure pointer or the element name may be used interchangeably, typically by using the `ELEMENT` function at the start to access the correct data structure.

Note that in these two examples `CREATE-ELEMENT` could have been used instead of `CREATE-CHANNEL`, since the more specific arguments for `CREATE-CHANNEL` were not used.

6.5 Element Parameters

Each element in the circuit is represented by a structure, as defined by the model type. All structures have a `:PARAMETERS` slot, as well as those slots that are unique to the model type. The `:PARAMETERS` slot is a general purpose location to store extra parameters in an association list. Accessing, loading, or changing specific values in the `:PARAMETERS` of a given element is done with:

element-parameter *element parameter* &optional *(value nil value−supplied−p) update* [*Function*]

which returns the value or values associated with the key PARAMETER for elements in ELEMENT. If VALUE is supplied, the parameter is set to this new value. For some types of elements and parameters, the UPDATE flag will cause the parameter to be fully processed. Note that any parameter stored in the `:PARAMETERS` slot is referenced to a key, which is a Lisp symbol (for example both `'FOO-LEVEL` and `:FOO-LEVEL` could be used as *distinct* keys). It is possible that a user-chosen key could conflict with one already used in the Surf-Hippo code. To insure against this, either grep the src/sys directory to check for a candidate key or choose highly personalized keys that would be unlikely to be used by the creative Surf-Hippo staff.

Whether or not element `:PARAMETERS` keys are quoted symbols or true keywords (that is preceded by a colon) is arbitrary in Surf-Hippo. Some functions (for example, the related function `IV-TYPE-PARAMETER` - see Section 12.4) rely on keyword symbols, while others may refer to quoted symbols.

6.6 Editing Individual Elements

In general, parameters of segments, channels and synapses are inherited from the appropriate cell type, channel type, and synapse type, respectively. However, for some of the element parameters, specific instances of these elements may be assigned unique parameter values by `SETF`ing the appropriate slot, or by using the function:

edit-element *element* &optional *model−type* [*Function*]

for example,

```
(EDIT-ELEMENT "Hippo-1" 'segment)
```

If individual parameters of a given element are to be not overwritten by the relevant element type parameters, then the `:INHERIT-PARAMETERS-FROM-TYPE` slot of the element must be set to NIL (this is provided for in the menus above). The initial value of `:INHERIT-PARAMETERS-FROM-TYPE` for a given element is set to the value of the relevant type `:INHERIT-PARAMETERS-FROM-TYPE` when the element is created.

The menus for individual element parameters are also accessible via the histology element menu (select a soma or segment first with mouse LEFT and then the menu for the chosen element with SHIFT-CONTROL-LEFT). A quick way to see some salient characteristics of a given element is by using:

print-element *element* &optional *model−type (stream *standard−output*)* [*Function*]

plot-element *element* [*Function*]

When the argument of `PRINT-ELEMENT` is a circuit element (soma, segment, channel, particle, etc.) the current state of the element will also be printed as long as the circuit has been initialized (by the function `INITIALIZE-SIMULATION`, called at the beginning of every simulation)

6.7 Cell Element Lengths and Diameters

element-length *element* &optional *new−length* [*Function*]

element-diameter *element* &optional *new−diameter* [*Function*]

These functions return the total length or diameter, respectively, of the cell elements associated with ELEMENT in microns - note that `ELEMENT-LENGTH` will return NIL if the assocatied cell element is a soma and not a segment (see also Section 7). These functions provide access to segment and soma slot values `:LENGTH` (for segments only) and `:DIAMETER`.

In addition, each function has an optional argument (NEW-LENGTH or NEW-DIAMETER) which, when supplied, will become the new length or diameter of the cell element. In that case, the global variable `*CIRCUIT-PROCESSED*` will be set to NIL, so that `PROCESS-CIRCUIT-STRUCTURE` will be called appropriately to process the changed dimensions.

6.8 Cell Element Areas and Volumes

element-area *element* &optional *consider−virtual−elements model−type* [*Function*]

element-volume *element* &optional *consider−virtual−elements model−type* [*Function*]

These functions return the total area or volume, respectively, of the cell elements associated with ELEMENT in square microns or cubic microns, respectively (see also Section 7). Segment areas do not include the cylinder ends (i.e. only the lateral areas are considered). If a cell is given by ELEMENT, then the total cell area or volume is considered. Related functions include:

tree-area &optional *(cell *cell*)* [*Function*]

This returns the area in square microns of the dendritic (and axonal) tree attached to the soma of CELL. If CELL not supplied, uses the last cell created in the current circuit.

tree-length &optional *(cell *cell*)* [*Function*]

This returns the total length in microns of the dendritic (and axonal) tree attached to the soma of CELL. If CELL not supplied, uses the last cell created in the current circuit.

6.9 Anatomical Distance Functions

Note that the reference point for a segment is the location of its distal node.

as-the-crow-flies *location−1 location−2* [*Function*]

Returns the straight line distance between two elements, two explicit locations, of the combination, in microns. Elements can be either elements or names of elements, and explicit locations are lists of 3 numbers (X Y Z), e.g.:

```
* (AS-THE-CROW-FLIES '(200 -300 50) "Hippo-5")
1514.1003
```

distance-to-soma *element* [*Function*]

Given an element or the name of an element, returns the distance along the tree to the soma in microns. For example:

```
* (distance-to-soma "Hippo-3")
720.0
```

neighbors *target radius* &optional *restrict−to−cell−of−target* [*Function*]

This returns a list of all elements of the same type as TARGET which lie at most RADIUS microns away from the TARGET. For example:

```
* (neighbors "1-1-38" 20.0)
(<Segment 1-1-39: prox node 1-1-38>
 <Segment 1-1-38: prox node 1-1-37>
 <Segment 1-1-35: prox node 1-1-34>
 <Segment 1-1-34: prox node 1-1-33>
 <Segment 1-1-37: prox node 1-1-36>
 <Segment 1-1-36: prox node 1-1-35>
 <Segment 1-1-41: prox node 1-1-40>
 <Segment 1-1-40: prox node 1-1-39>)
```

element-cloud *reference−element cloud−radius* &optional *restrict−to−reference−element−cell returned−model−type* [*Function*]

This returns a list of segments and or somas who are within a radius of CLOUD-RADIUS microns from REFERENCE-ELEMENT (which can be any cell or membrane element). If RESTRICT-TO-REFERENCE-ELEMENT-CELL is T, then the returned somas/segments are constrained to be from the same cell as the REFERENCE-ELEMENT.

distals-without *distal−border* &optional *(model−type 'segment) cell* [*Function*]

Returns all elements of MODEL-TYPE that are further from the soma than DISTAL-BORDER (microns).

proximals-within *proximal−border* &optional *(model−type 'segment) cell* [*Function*]

Returns all elements of MODEL-TYPE that are closer to the soma than PROXIMAL-BORDER (microns).

6.10 Branch-Related Functions

A branch is defined as a set of singly connected segments whose proximal and distal ends are nodes with either more than 2 segments, or are a termination (soma or distal tip) point. The name of a branch is the name of its proximal segment.

branch *element* &optional *type* [*Function*]

Given an ELEMENT, returns the branch (list of segments) that the segment associated with ELEMENT is a part of. For example:

```
* (branch "Hippo-2")
(<Segment Hippo-5: prox node Hippo-4>
 <Segment Hippo-4: prox node Hippo-3>
 <Segment Hippo-3: prox node Hippo-2>
 <Segment Hippo-2: prox node Hippo-1>
 <Segment Hippo-1: prox node Hippo-soma>)
```

branch-ends *element* [*Function*]

Returns a list of the proximal and distal segments of the branch referenced by ELEMENT. For example:

```
* (branch-ends "Hippo-2")
(<Segment Hippo-1: prox node Hippo-soma>
 <Segment Hippo-5: prox node Hippo-4>)
```

branch-elements *branch−element element−type* &optional *total−segments ends* [*Function*]

Returns a list of elements of type ELEMENT-TYPE from the branch identified by BRANCH-ELEMENT. If the TOTAL-SEGMENTS is included (should be a number), then elements are chosen from a total of (approximately) TOTAL-SEGMENTS segments, evenly distributed along the branch. Note that the order of the elements in the list corresponds to the order of the segments in the branch, that is from proximal to distal. For example:

```
(branch-element "11-5" 'segment 5)
```

will return 5 (approximately) evenly distributed segments from branch "11-5". Related functions include

branch-synapses-of-type *branch−element type* &optional *total−segments ends* [*Function*]

branch-channels-of-type *branch−element type* &optional *total−segments ends* [*Function*]

For example:

```
(branch-synapses-of-type "11-5" "AUTO-INH")
```

will return all the synapses of type AUTO-INH along the branch "11-5", from proximal to distal.

6.11 Tree-Related Functions

segments-out *element* &optional *(segment−skip 0) previous−segs* [*Function*]

Given a SEGMENT, returns a list of all the segments moving distally, skipping by SEGMENT-SKIP. If a loop is encountered (a segment distal to SEGMENT is found in the optional argument PREVIOUS-SEGS, which is used on recursive calls to `SEGMENTS-OUT`), then an error is signaled.

segments-in *element* &optional *(segment−skip 0)* [*Function*]

Given a SEGMENT, returns an inclusive list of all the segments on the path to the soma, skipping by SEGMENT-SKIP.

loop-check &optional *exclude−segments* [*Function*]

Find any loops in the circuit trees by sucessive calls to SEGMENTS-OUT. If a loop is found, then `SEGMENTS-OUT` signals an error.

distal-segments *element* &optional *include−electrodes* [*Function*]

Returns a list of all the segments directly attached to the distal node of segment associated with ELEMENT, or the trunk segments if ELEMENT is associated with the soma.

proximals &optional *(model−type 'segment) (proximal−distal−distance−cutoff 0.5) cell* [*Function*]

distals &optional *(model−type 'segment) (proximal−distal−distance−cutoff 0.5) cell* [*Function*]

These functions return lists of elements, according to the specified ELEMENT-TYPE, that are either proximal or distal on the dendritic tree with respect to whether their distance to the soma (along the most direct path on the tree) is less than or greater than, respectively, the longest path in the tree times the PROXIMAL-DISTAL-DISTANCE-CUTOFF coefficient. If CELL is included, then the returned elements are only from that cell; otherwise elements are returned from all cells in the current circuit.

soma-segments &optional *(target *cell*)* [*Function*]

Returns a list of segments which are conceptually assigned to the actual soma for the cell associated with TARGET. (see Section 7.3).

trunk-segment *element* [*Function*]

Given a segment or a segment name, returns the associated proximal segment that abuts onto the soma.

6.12 Element Parameter Histograms

element-param-distribution *model−type parameter* &key *parameter−function cell note param−max param−min type−for−title x−label y−label x−min x−max x−inc (x−axis−tick−skip 0) x−are−fns y−min y−max y−inc (y−axis−tick−skip 0) y−are−fns bin−width include−simulation−name (width 350) (height 300) font title−position create−new−window* [*Function*]

Generates a histogram of the distribution of various element parameters, including:

```
'AREA         =>  (element-area elt)
'DISTANCE     =>  (distance-to-soma elt)
'DIAMETER     =>  (element-diameter elt)
'CAPACITANCE  =>  (element-capacitance elt)
'GBAR         =>  (element-gbar elt)
```

For example,

```
(ELEMENT-PARAM-DISTRIBUTION 'SEGMENT'DISTANCE :bin-width 20)
```

These histograms may be modified somewhat with the histogram menu (Control-m with the mouse over the histogram window).

6.13 Units of Measurement

Table 1 lists the default units used in Surf-Hippo for all elements of the circuit.

Quantity	Units
Distance	micrometers
Surface area	micrometers^2
Temperature	degrees Celcius
Time	milliseconds
Rate (e.g. particle kinetics)	milliseconds^{-1}
Voltage	millivolts
Current	nanoamperes
Capacitance	nanofarads
Specific capacitance	microfarads centimeters^{-2}
Resistance	megaohms
Resistivity	ohms centimeters^2
Conductance (ohmic conduction)	microsiemans
Conductance Density	picosiemans micrometers^{-2}
Permeability (constant field conduction)*	centimeters^3 seconds^{-1}
Concentration	millimolar

Table 1: The default units used in Surf-Hippo for all elements of the circuit. $*$ Used in the `:CONDUCTANCE` slot.

6.14 More Functions Based on ELEMENT

element-name *element* &optional *model−type* [*Function*]

element-cell *element* &optional *model−type* [*Function*]

element-cell-element *element* &optional *model−type* *fast* *(disable−element−slot−core−typecase−error t)* [*Function*]

These functions return the names, cells, or cell elements, respectively, of the elements associated with ELEMENT of TYPE. For example:

```
* (element-name (segments))
("Hippo-3" "Hippo-2" "Hippo-1" "Hippo-5" "Hippo-4")
* (element-cell "Syn-11-6-17-AUTO-FAST-EX")
```

```
<Cell j43d: type V1-pyramidal>
* (element-cell-element "Syn-11-6-17-AUTO-FAST-EX")
<Segment 11-6-17: prox node 11-6-16>
```

Many elements can be turned on or off, in other words enabled or disabled, using:

enable-element *element* &optional *model−type* [*Function*]

disable-element *element* &optional *model−type* [*Function*]

For example, individual channel or synapses types, or specific channels or synapses, may be blocked (disabled), as well as membrane pumps, buffers, and concentration integrators. Likewise, current and voltage sources may be turned on or off, etc.

element-location *element* &optional *model−type* [*Function*]

This returns the XYZ coordinates (microns) of the cell element node(s) associated with ELEMENT of TYPE.

elements-of-type *element−reference* &optional *cell−elements−reference* [*Function*]

If ELEMENT-REFERENCE is a model type symbol (e.g. 'channel or 'channel-type), then returns all instances of the model (e.g. all channels or all channel types). Otherwise, if ELEMENT-REFERENCE refers to a specific instance of an element parent type (a synapse type, a channel type, etc.), returns all the child instances (synapses of that synapse type, or channels of that channel type, etc). If CELL-ELEMENTS-REFERENCE is included, returned elements are restricted to cell elements associated with CELL-ELEMENTS-REFERENCE; otherwise all elements are returned.

```
* (elements-of-type 'NA-4STATE-EXP-GEN)
(<Channel 119-NA-4STATE-EXP-GEN: type NA-4STATE-EXP-GEN>
 <Channel 118-NA-4STATE-EXP-GEN: type NA-4STATE-EXP-GEN>
...
 <Channel 11-NA-4STATE-EXP-GEN: type NA-4STATE-EXP-GEN>)
```

cell-elements &optional *element model−type* [*Function*]

Returns a list of all segments and somas associated with the cell or cells in CELLS (can be a single cell or a list) [default all cells in circuit].

7 Functions for Creating Cell Elements and Defining Cell Geometries

The most general way to create cell geometries is by explicit creation of somas and segments using the functions described in this section. Alternatively, various anatomical file formats may be processed to create complete cells, as described in Section 8.

7.1 Explicitly Creating Somas, Cells and Segments

At minimum, all neurons must have with a soma; the basic function for this step is

create-soma &key *cell cell−type name (location '(0.0 0.0 0.0)) length soma−cylinder−diameter (diameter *default−soma−diameter*) parameters adjust−area−for−trunks shunt (enable−automatic−cell−names *enable−automatic−cell−names*) (automatic−name−fixing *prompt−for−alternate−element−names*)* [*Function*]

where DIAMETER is in microns. The CELL argument is either a cell structure or a string - if not supplied, a cell is created. SHUNT (in ohms, default NIL), when supplied, is a non-specific somatic shunt. LOCATION gives the xyz coordinates of the soma in microns. When ADJUST-AREA-FOR-TRUNKS is T (default nil), then the soma area (as returned by the `ELEMENT-AREA` and `ELEMENT-AREA-CM2` functions) is adjusted for the areas of the faces of any abutting segments. If desired, the CELL argument may be supplied, for example, by the result of a call to

create-cell *cell−name* &key *cell−type tree−list soma−diameter (segment−diameter 0.5) (origin '(0.0 0.0 0.0)) (name−suffix *cell−name−suffix*) (enable−automatic−cell−names *enable−automatic−cell−names*) (automatic−name−fixing *prompt−for−alternate−element−names*)* [*Function*]

This function creates a new cell, if not already defined, and returns the cell. If cell is not already defined, a soma is created when SOMA-DIAMETER is supplied (microns), and if SOMA-DIAMETER and TREE-LIST are supplied, TREE-LIST is used in a call to `CREATE-TREE`, with a `:DEFAULT-DIAMETER` argument given by SEGMENT-DIAMETER (microns, default 0.5). If the global variable `*NEXT-CELL-NAME*` is non-NIL, then this will be used instead of CELL-NAME. Always sets `*NEXT-CELL-NAME*` to NIL. If NAME-SUFFIX is non-NIL (default `*CELL-NAME-SUFFIX*`), it is automatically added as a suffix to the name of a cell, even if this is supplied by `*NEXT-CELL-NAME*`. Thus, the values of either `*NEXT-CELL-NAME*` and/or `*CELL-NAME-SUFFIX*` may be set prior to the `CREATE-CELL` call in order to ensure unique cell names in a multiple cell circuit.

The CELL-TYPE argument may be either an explicit cell type or a type symbol used by a previously loaded `CELL-TYPE-DEF` macro (see Section 11). If a type symbol does not correspond to an entry in the parameter library, then the cell type parameters will be taken from various global variables, including `*R-MEM*`, `*R-I*`, `*CAP-MEM*`, `*CAP-MEM-DENDRITE*`, `*SOMA-SHUNT*`, `*E-LEAK*`, `*E-NA*`, `*E-K*`, `*E-CA*`, `*E-CL*`, in addition to default specifications for reversal potentials (`:FIXED`) and concentrations (`:FOLLOWS-GLOBAL`).

Note that each of these functions updates the global variables `*SOMA*`, `*CELL*` and `*CELL-TYPE*`, as appropriate (Section 6.3.1).

Dendrite geometries may be built up with repeated calls of

create-segment *name proximal−element* &optional *cell* &key *(diameter 0.0) (length 0.0) (theta 0.0) (phi (* −0.5 pi−single)) (relative−location '(0.0 0.0 0.0)) relative−location−is−float absolute−location absolute−location−is−float dummy−proximal−element−location dummy−proximal−element−location−is−float (ri−coefficient 1.0) parameter−a−list* [*Function*]

which returns the created segment. NAME is a string. PROXIMAL-NODE may be either a string (which may point to an already created node), a segment (whose distal node will be used as the new segment's

proximal node), a soma or a node. The distal node name is the same as the segment name. Note that the membrane properties will be set again with the call to `SET-SEGMENTS-MEMBRANE-PARAMETERS` since segment dimensions may have to be figured out later when the segment is defined in terms of its coordinates. RELATIVE-LOCATION is of the segment's distal node relative to cell soma (in microns). The location of the distal node relative to cell soma may also be determined by including PHI, THETA (radians) and LENGTH (microns) keyword arguments (see the function `TREE-CONTROL` below). Precedence of location information when segment node locations are determined is defined in the function `LOCATE-DISTAL-NODE`. A coefficient for the intracellular resistivity will be taken from RI-COEFFICIENT, respectively, if either is not equal to 1.0 (the default). Note that if the global variable `*USE-SIMPLE-NAMES*` is T the NAME will be ignored (can be NIL).

A straight line of segments may be created with

segment-chain *proximal−cell−element chain−name total−segs seg−length seg−diam* [*Function*]
&key *(proximal−phi 0.0) (proximal−theta 0.0)*

which adds a straight chain of segments of the same dimensions to the PROXIMAL-CELL-ELEMENT, returning the last (distal) segment in the created chain. If CHAIN-NAME is nil, then the segment names are derived from the name of the associated cell. The PROXIMAL-PHI and PROXIMAL-THETA arguments specify the angle of the branch chain with respect to the PROXIMAL-CELL-ELEMENT, in radians. The default values of 0.0 for PROXIMAL-PHI and PROXIMAL-THETA generate a chain that extends from the PROXIMAL-CELL-ELEMENT in the positive X direction. For a chain of segments that extends in the positive Y direction, include the key argument:

```
:PROXIMAL-THETA (* -0.5 pi)
```

The orientations of the PROXIMAL-PHI and PROXIMAL-THETA arguments are given in Figure 1. See for example the definition of the Working Model geometry, given by the function `HIPPO` (`src/hippocampus/hippo.lisp`).

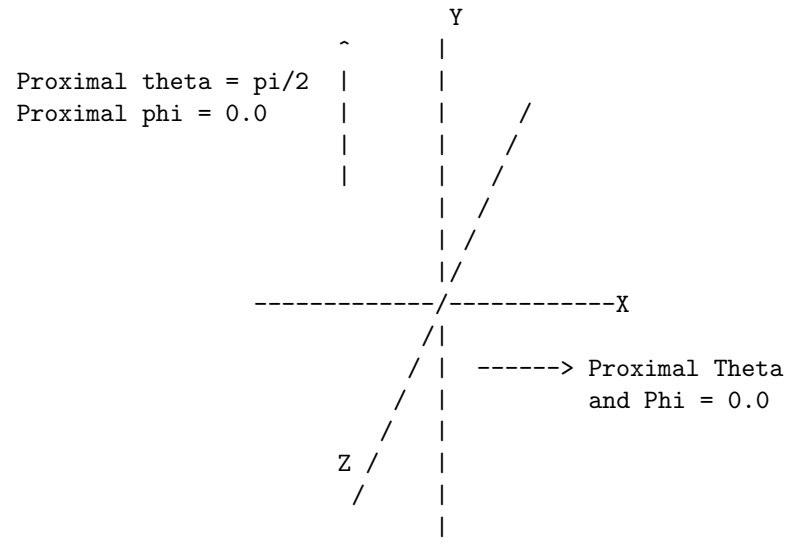

Figure 1: Relative orientations of the chain of segments generated by `SEGMENT-CHAIN`, according to the function arguments PROXIMAL-THETA and PROXIMAL-PHI.

Note that with limited internal precision, the calculated locations of segments derived from angular arguments may be slightly different than the ideal case (for example, the single-precision result of `(cos (/ pi-single 2))` is -4.3711388e-8, not 0.0).

More sophisticated dedrite geometries may be defined by

create-tree *cell−reference segment−list* &key *(xy−factor 1.0) (z−factor 1.0) (add−cell−name−to−segs *add−cell−name−to−segs* add−cell−name−to−segs−supplied−p) (default−soma−diameter *default−soma−diameter*) (default−diameter 0.5)* [*Function*]

which creates a segment tree according to SEGMENT-LIST, adding the tree to the soma associated with CELL. The associated cell is returned. SEGMENT-LIST is a list of lists, where the sublist format is as follows:

```
(prox-elt-name seg-name x y z &optional diameter extras)
```

The PROX-ELT-NAME refers to the proximal segment or soma, the SEG-NAME is for the segment to be created, and x, y, and z refer to the coordinates of the distal node of the segment to be created. EXTRAS is a list of lists for adding channels or synapses to a segment. If GLOBAL-EXTRAS-LIST is supplied, then this list is used in addition to any extras specific to a given segment. XY-FACTOR and Z-FACTOR are scaling factors for node coordinates, which may be useful when translating histological renderings into the sublists. The PROX-ELT-NAME of the first sublist will refer to the soma, which has been created already with `CREATE-SOMA`. For example, the segment sublist:

```
(soma 1a 7 -1 -5 1.2)
```

specifies a segment named "1a" whose proximal end connects to the node named "soma", whose distal node has coordinates (7×XY-FACTOR, -1×XY-FACTOR, -5×Z-FACTOR), and whose diameter is 1.2 microns. Likewise, the segment sublist:

```
(1a 1b 12 -3 -7 0.6 '(KA-HPC))
```

specifies a segment named "1b" whose proximal end connects to the distal node of segment "1a", whose distal node has coordinates (12×XY-FACTOR, -3×XY-FACTOR, -7×Z-FACTOR), and whose diameter is 0.6 microns. In addition, an `KA-HPC` type channel is included at the segment's distal node. If the PROX-ELT-NAME and SEG-NAME are the same, then this is the cell's soma, and the diameter is the soma diameter (which overrides the previous diameter). This entry will be used to reference the coordinates of the segments, so that they are created in relative coordinates. The soma origin is set elsewhere if it is to be other than (0 0 0). If the ADD-CELL-NAME-TO-SEGS keyword is set, then the cell name is prepended to the segment names specified in the segment sublists.

For an example of `CREATE-TREE`, see the definition of `STAR-AMACRINE` (`src/retina/star-amacrine.lisp`, derived from data of Tauchi and Masland, 1984; this is the cell referenced in `STAR-AMACRINE-DS`).

7.2 Moving Cells and Modifying Cell Geometry

move-cell *cell new−origin* [*Function*]

shift-cell *cell* &key *(x−shift 0.0) (y−shift 0.0) (z−shift 0.0)* [*Function*]

These functions allow moving a CELL (referenced by either the cell or the name of the cell) relatively (`SHIFT-CELL`) or absolutely (`MOVE-CELL`), with all dimensions given in μm.

set-proximal-thetas *seg* &optional *(total−fan−angle 30.0) (spreadmoredistal 0)* [*Function*]

For spreading out a branch. Fans out segments working outward from SEG by distal recursion to the end of the associated branchs, such that the total fan angle at each branch point is equal to:

$$TOTAL - FAN - ANGLE \quad + \quad \frac{SPREADMOREDISTAL}{\#segmentstosoma} \tag{1}$$

$$\text{TOTAL-FAN-ANGLE} + \frac{\text{SPREADMOREDISTAL}}{\text{number of segments to soma}}$$

with all angles given in degrees.

warp-cell *cell* &key *(x−factor 1.0) (y−factor 1.0) (z−factor 1.0)* [*Function*]

Multiplies the relative location vector of each node of CELL by the appropriate key arguments, and then calls PROCESS-CIRCUIT-STRUCTURE to reevaluate the segment dimensions and electrical parameters.

7.3 Soma Segments

In certain cases, a cell model will describe the anatomical soma as some combination of one or more connected cylindrical segments, perhaps explicitly including a terminal spherical part. Here, the segments assigned conceptually to the soma are referred to as "soma segments". As far as the simulator is concerned, the main distinction for a soma segment SEGMENT object is that it will be highlighted appropriately when drawn in a histology window. On the other hand, the classification of soma segments may be convenient when assigning channels, synapses etc. to various locations in the cell.

Some anatomical file formats (Neurolucida, as far as I know, for one - see Section 8.5.1) describes somas only in terms of cylindrical sections. In this case, the (necessary) SOMA object has negligible physical dimensions, and it is understood that any additional elements to be attached to the soma should instead be attached to one (or more) of the soma segments. For example, the much smaller branch elements to ground implied by a very small soma can lead to numerical instabilities when channels or synapses are added directly to the same node.

Conceptual soma segments may assigned by the function

add-soma-segment *soma segment* &optional *initialize* [*Function*]

which assigns the segments associated with SEGMENT to the SOMA. If INITIALIZE is T [default NIL], then first remove any already defined soma segments. A soma segment may be explicitly be removed (that is, dis-assigned from the soma, not removed from the circuit) with

remove-soma-segment *soma seg* [*Function*]

A list of the soma segments assigned to the soma may be retrieved by the function

soma-segments &optional *(target *cell*)* [*Function*]

and the predicate

soma-segment-p *element* [*Function*]

indicates whether the segment associated with ELEMENT has been designated as a soma segment.

8 Processing Anatomy Files

This section describes conversions of various anatomy file formats into Surf-Hippo format.

8.1 Anatomy File Types

As explained below, a hack of the Unix program ntscable is used to translate certain anatomical file formats into Lisp files that then may be input into Surf-Hippo. These formats include:

- NTS (Neuron Tracing System, Eutectic Electronics)
- Rodney Douglas
- Rocky Nevin

Neurolucida format files, on the other hand, are processed directly by Surf-Hippo.

When loading anatomy files using explicit functions such as `CIRCUIT-LOAD` or `ADD-CIRCUIT`, the global variable `*CIRCUIT-FILE-TYPE*` is set automatically as appropriate for each file. The relevent values are:

```
:NEUROLUCIDA
:LISP
```

In this section, only the first two values are pertinent. If anatomy files are loaded with the menus, this variable is set automatically.

8.2 Using The Surf-Hippo Hack of ntscable to Translate Neuronal Anatomy Files

The geometry of the soma and dendritic tree for a cell model may be derived directly from various anatomical reconstruction file formats. The basic process is:

```
    anatomy.file that can be digested by ntscable

                     |
                     | ntscable (under Unix)
                     |
                     V

  Surf-Hippo lisp circuit file

                     |
                     | editing (optional)
                     |
                     V

    Load circuit file into the simulation with menus
    or via scripts using a function such as CIRCUIT-LOAD
```

8.3 Cell Type Specification For Anatomy File-Based Cells

In order to explicitely define the cell type for anatomyfile-based cells, it is necessary to set the value of the global variable `*DEFAULT-CELL-TYPE-NAME*` prior to loading the anatomy file, according to some definition given by a `CELL-TYPE-DEF` form.

8.3.1 ntscable and Surf-Hippo

ntscable is a C program written by JC Wathey, then at the Computational Neurobiology Laboratory of the Salk Institute. From the ntscable.doc file:

> "This program translates a digitized morphological description of a neuron into files which can be used directly by the simulation programs "CABLE" and "NEURON", by Michael Hines of Duke University (Hines, 1989). In its original incarnation, ntscable could only read data files in the syntax of the Neuron Tracing System (Eutectic Electronics) and could only produce output in CABLE syntax; hence the name "ntscable". The latest version can also read data files generated by the digitizing systems of Rodney Douglas and Rocky Nevin. All of these digitizing systems are similar in concept. Since the Eutectic NTS system is the most widely used, it is described in some detail under INPUT FILE SYNTAX, below. Special considerations for using Douglas and Nevin data files are also described there."

We have hacked ntscable a bit so that it will generate files that Surf-Hippo can digest, using functions found in `src/sys/ntscable.lisp`. All the files associated with ntscable itself are found in the `misc/ntscable/` directory The file `/misc/ntscable/ntscable.doc` is the original documentation for ntscable, while the file that you are reading now is just relevant to Surf-Hippo.

NOTE: ntscable must be (re)compiled with the Surf-Hippo hack of write.c (surf-hippo-write.c) and the associated makefile. A recompiled executable "ntscable" file is found in the `/misc/ntscable/ntscable/` directory.

8.3.2 Running ntscable for Making Surf-Hippo Anatomy Files

To translate an anatomy file into a Surf-Hippo (lisp) file, you run the modified ntscable from UNIX (*not* from Lisp!!):

```
unix-prompt> ntscable -x n input-anatomy-file output-lisp-file
```

In other words, compiling ntscable with surf-hippo-write.c causes the "-x n" option (originally used to generate NEURON format file) to mean "generate a Surf-Hippo format file".

ntscable assigns each point of the cell the original anatomical coordinates, which may be referenced to an arbitrary origin. Surf-Hippo, on the other hand, assigns the (relative) origin of each cell to that cell's soma center. Thus, the `:RELATIVE-LOCATION` for the segment nodes are with respect to the soma origin. The `:ABSOLUTE-LOCATION` of the nodes for the segments and the soma of a cell are calculated by adding the relative coordinates to the value in the cell's `:ORIGIN`. Since ntscable assigns essentially absolute coordinates to the cell elements, Surf-Hippo adjusts these coordinates, according the location of the soma as defined by ntscable, so that the soma relative coordinate is at the origin, and everything in the cell is referenced from there, as explained.

It is possible that the input anatomy files will have more than one (connected) point with the same coordinates. The processing by the ntscable.lisp functions kills the resulting zero-length segments.

8.3.3 ntscable Somas → Surf-Hippo

The soma dimensions have two characteristics, as determined by ntscable. First, this program calculates the diameter of an equivalent spherical soma, and it is this quantity which Surf-Hippo uses for the electrical representation of the soma. This is reflected in a Lisp form near the beginning of the ntscable output file that is something like this:

```
(setq *nts-radius* (sqrt (/ 1453.39 (* 3.14159 4))))
```

This may be edited if necessary before loading the file into Surf-Hippo. In addition, the soma diameter may be redefined by the Surf-Hippo user menu.

ntscable also generates a list of parallel circles (slices) representing the soma, and a point list that follows an outline which is orthogonal to the slices. Surf-Hippo uses these data for generating the graphical picture of the soma. This shape is *irrelevant* with respect to the circuit representation of the soma. The assumption is that ntscable has calculated a reasonable spherical approximation to the anatomical points.

8.3.4 Surf-Hippo Addendum to the ntscable Output Files

The Surf-Hippo version of ntscable produces an output file that has, at the end, several Lisp forms that may also be edited as desired. These include (with their default values):

```
(setq *nts-cell-type* "ntscable")
(setq *nts-cell-name* "ntscable")
(setq *nts-r-mem* 40000.0)
(setq *nts-soma-r-mem* 40000.0)
(setq *nts-r-a* 200.0)
```

You also may want to add a form like the following:

```
(setq *nts-cell-type-notes*
      "Sub-genius pyramidal neuron, area Z, digitized by Ramon Y. Cajal")
```

However, if the global variable `*DEFAULT-CELL-TYPE-NAME*` is non-NIL, then the referenced cell type will override these special NTS global variables.

For large cell files, it is much more efficient to load compiled files into Surf-Hippo than the source (ascii) files. The file browser that appears when a file is to be loaded into Surf-Hippo tags both *.lisp files and *.sparcf (compiled) files.

When the global variable `*ADD-CELL-NAME-TO-SEGS*` is T, then the value of `*NTS-CELL-NAME*` is prepended on the segment names derived from the original anatomy list. This is necessary if you are going to use the same anatomy file for more than one cell.

8.4 Some notes on Rocky Nevin format files

A point labeled "P" generates:

```
DG-m49-4-22-92: invalid point type 'P' in line 12
```

Changing the "P" to a "B" seems to do the right thing. Apparently, the EOF must occur at the end of the last line, i.e. no extra CRs or LFs. Otherwise (?) -

```
read_nevin: syntax error in DG-m49-4-22-92 at line 736:
```

8.5 Neurolucida File Format

Neurolucida files are processed by the function:

read-neurolucida-file *filename* &optional *cell−name* [*Function*]

However, in general you will not have to call this function explicitly, since `CIRCUIT-LOAD` can automatically detect the Neurolucida file type.

The proximal location of each segment is taken (inferred) from the neurolucida file format. There may be cases where this location is only referred to once in the file, in the sense that a segment is defined as starting from a location which is not shared by any other cell location. In these cases, the proximal location (from the file) is assigned to the segment's `:DUMMY-PROXIMAL-NODE-LOCATION` slot, and the proximal node is chosen as the closest node in the rest of the circuit. The location specified by the `:DUMMY-PROXIMAL-NODE-LOCATION` slot is used for calculating the length and drawing geometry of the segment, while the actual proximal node (the segment's `:NODE-1` slot) is the connection used by the electrical circuit.

8.5.1 Neurolucida Somas

Neurolucida files are processed under the assumption that only cylinderical sections are described, whether corresponding to somas, dendrites, axons or spines. Thus, only `segments` are explicitly defined. One or more may be assigned to the "soma" by the neurolucida format (e.g. a "minor code" of 41 or 42), and are designated as "soma segments" (see Section 7.3).

Since, however, a `soma` node is required by the method for solving the circuit equations, a soma is created whose diameter is given by the global variable `*NEUROLUCIDA-SOMA-DIAMETER*` (microns, default 0.0), generally placed at the proximal end of the first soma segment encountered in the Neurolucida file. Note that with the default diameter of 0.0 microns, this soma has no capacitive nor leak resistance connection to ground. It is advisable to avoid assigning membrane elements directly to this node.

8.5.2 Debugging Neurolucida Files

Problems that show up in the graphical rendition of a cell or otherwise can be traced to the neurolucida file by using the following form (for example referencing segment 527):

```
* (element-parameter 527 'file-line-number)
6793
*
```

This gives the line number in the original file which defined the distal location and radius of the segment. In this example, it was determined that segment 527 did not look right, so with the above form it was discovered that the 6793rd line in the anatomy file defined the segment.

Now look at the file -

```
[1,2]  (37.21, 18.54, 16.00)  0.58
[1,2]  (37.60, 18.54, 16.00)  0.58
[1,2]  (37.60, 19.30, 16.00)  0.58
[1,2]  (37.98, 19.30, 16.00)  0.58   <- line 6793
[10,5]  (37.60, 19.30, 16.00)  0.58
[1,2]  (37.60, 19.30, 16.00)  0.58
[1,2]  (39.13, 18.92, 16.00)  0.58
[1,2]  (39.51, 18.92, 16.00)  0.58
```

The problem is that there is a loop back after the [10,5] line which connects a segment to the segment at location (37.60, 19.30, 16.00). Two solutions are possible: either block the creation of the segment at (37.98, 19.30, 16.00), or specify a branch point at (37.60, 19.30, 16.00). The second solution requires adding the following line:

```
[1,2]  (37.60, 18.54, 16.00)  0.58
[1,2]  (37.60, 19.30, 16.00)  0.58
[1,2]  (37.98, 19.30, 16.00)  0.58
[10,5]  (37.60, 19.30, 16.00)  0.58

;; adding this line because otherwise we have a loop
[2,1]  (37.60, 19.30, 16.00)  0.58

[1,2]  (37.60, 19.30, 16.00)  0.58
[1,2]  (39.13, 18.92, 16.00)  0.58
[1,2]  (39.51, 18.92, 16.00)  0.58
```

The preceeding discussion assumes that the described process does not in fact bend back and pass through a point in space that was already assigned to another point on the process. In the translation algorithm, it is assumed that there should be a unique assignment of points in space to anatomical points.

The parsing of these files by Surf-Hippo allows lines to be commented out by a leading ";".

8.6 Zero Length Segments in Trees

If the tree definition parameters specify a segment of zero length, then that segment is not included in the final tree. During the processing of the cell geometry, a message will appear that you can refer to later in order to modify the original parameters, if desired:

```
Reading in circuit CELLULE2...
Destroying zero length segment BAS4-3
Destroying zero length segment BAS3-3
Locating segments...
```

9 Cell, Cable and Linear Parameters

There are a variety of functions for evaluating common cell and cable parameters and various linear parameters of cells.

9.1 Basic Linear Properties of Cell Types

The `CELL-TYPE-PARAMETER` function is similar to the `IV-TYPE-PARAMETER` function, but for examining or changing cell type parameters:

cell-type-parameter *element param* &optional *(value nil value−supplied−p) (update t)* [*Function*]

PARAM can be:

```
:RI [ohms-cm]
```

These set both the soma and the dendritic values -

```
:RM [ohms-cm2]
:V-LEAK [mV]
:CM [uF/cm2]
```

These set only the dendritic values-

```
:RM-DENDRITE [ohms-cm2]
:V-LEAK-DENDRITE [mV]
:CM-DENDRITE [uF/cm2]
```

These set only the somatic values-

```
:RM-SOMA [ohms-cm2]
:V-LEAK-SOMA [mV]
:CM-SOMA [uF/cm2]
:SOMA-SHUNT [ohms]
```

9.2 Basic Linear Properties of Segments and Somas

Other functions for examining or setting segment and soma parameters individually include the following.

set-segment-absolute-parameters *seg capacitance g−axial g−leak* [*Function*]

segment-v-leak *segment* [*Function*]

segment-g-axial *segment* [*Macro*]

segment-g-leak *segment* [*Macro*]

segment-capacitance *segment* [*Macro*]

set-soma-absolute-parameters *soma capacitance g−leak* [*Function*]

soma-v-leak *soma* [*Function*]

soma-g-leak *soma* [*Macro*]

soma-capacitance *soma* [*Macro*]

9.2.1 Somatic Shunt

A non-specific soma shunt is considered when the `:INCLUDE-SHUNT` slot of a soma is non-NIL (see also Section 7). The value of the shunt is given by either a non-nil value of `(element-parameter soma 'soma-shunt)` or by the `:SOMA-SHUNT` slot of the soma's cell-type (see Section 7).

9.3 Geometry Parameters of Segments and Somas

For lengths and diameters of cell elements, see Section 6.7 for a description of the functions `ELEMENT-LENGTH` and `ELEMENT-DIAMETER`. For the membrane surface areas and the total volume of these elements, you may use `ELEMENT-AREA` and `ELEMENT-VOLUME`, as described in Section 6.8.

9.4 More Linear Properties of Segments, Somas and Cells

cell-cap &optional *(cell *cell*) (exclude−electrodes t)* [*Function*]

This returns the total capacitance of the CELL in nF. If CELL not supplied, uses the first cell created in the current circuit. When EXCLUDE-ELECTRODES is T, any contribution by attached electrodes will be ignored.

lambda-cable *ri rm a−um* [*Function*]

This returns cable electrotonic space constant in cm. Intracellular resistivity RI is in Ω cm, membrane resistivity RM is in Ω cm^2, and cable radius A-UM is in microns.

length-from-lambda *ri rm a−um l* [*Function*]

This returns cable length in um given intracellular resistivity RI (Ω cm), membrane resistivity RM (Ω cm^2), cable radius A-UM (microns), and electrotonic length L (dimensionless!).

segment-electrotonic-length *segment* [*Function*]

This returns electrotonic length of segment SEG.

electrotonic-length *length diameter ri rm* &optional *(ri−coefficient 1.0)* [*Function*]

This returns electrotonic length of segment given explicit parameters LENGTH (microns), DIAMETER (microns), CELL-TYPE-RI (Ω cm), RI-COEFFICIENT (dimensionless), and CELL-TYPE-RM (Ω cm^2).

g-inf-in *ri rm a−um* &optional *lambda−cable* [*Function*]

This returns the input conductance of semi-infinite cable, in μS. Intracellular resistivity RI is in Ω cm, membrane resistivity RM is in Ω cm^2, and cable radius A-UM is in microns.

z-cable-in *ri rm a−um l−um* &optional *(g−end 0.0)* [*Function*]

This returns input resistance (MΩ) to sealed-end (open circuit) cable of length L-UM (microns). Intracellular resistivity RI is in Ω cm, membrane resistivity RM is in Ω cm^2, and cable radius A-UM is in microns. Optional G-END is in μS.

z-cable-in-seg *segment* &key *store−segment−z−cable−in* [*Function*]

This returns input resistance (MΩ) of SEGMENT, taking into account the tree distal to the segment, using the cable parameters.

z-cable-in-cell &optional *(cell *cell*) z−tree* [*Function*]

This returns input resistance (MΩ) of cell, using the cable parameters for the dendritic tree if Z-TREE is not supplied. Otherwise, the input resistance is calculated from the soma resistance and the Z-TREE argument (MΩ). If CELL not supplied, uses the first cell created in the current circuit.

z-tree-cable-in-cell &optional *(cell *cell*) include−virtual−soma* [*Function*]

This returns input resistance (MΩ) of dendritic tree of CELL, using the cable parameters. If no tree, returns NIL. If CELL not supplied, uses the first cell created in the current circuit. If INCLUDE-VIRTUAL-SOMA is T, include any segments assigned to the soma.

z-discrete-in-cell &optional *(cell *cell*) z−tree* [*Function*]

This returns input resistance (MΩ) of cell, using the compartmental network parameters. If CELL not supplied, uses the first cell created in the current circuit.

z-tree-discrete-in-cell &optional *(cell *cell*) include−virtual−soma* [*Function*]

This returns input resistance (MΩ) of dendritic tree of CELL, using the compartmental network parameters. If no tree, returns NIL. If CELL not supplied, uses the first cell created in the current circuit. If INCLUDE-VIRTUAL-SOMA is T, include any segments assigned to the soma.

g-element *element g−density* [*Function*]

This returns the absolute conductance of CELL-ELEMENT in μS. G-DENSITY is in pS per square micron.

10 Current and Voltage Sources

10.1 Adding Sources

Sources may be added with the histology window menus, or by calling the functions:

add-isource &optional *(element *soma*)* &key *name pulse−list (type 'autonomous) (reference−magnitude *isource−default−reference−magnitude*)* [*Function*]

add-vsource &optional *(element *soma*)* &key *name (ideal t) pulse−list (reference−magnitude *vsource−default−reference−magnitude*)* [*Function*]

For example:

```
(add-isource "Hippo-soma")
```

or

```
(add-vsource "Hippo-soma")
```

If the optional NAME argument is not supplied, the name of the source is the node name concatenated with either "isrc" or "vsrc", as appropriate.

A more detailed method is to use the functions:

create-pwl-isource *cell−element* &key *type name node−1−cell−element (use−pulse−list t) pulse−list (enable−individual−pulses t) enable−pulse−train pulse−train−args waveform−spec (waveform−time−interval *default−waveform−step*)* [*Function*]

create-pwl-vsource *cell−element* &key *(waveform−time−interval *default−waveform−step*) name pulse−train−args enable−pulse−train (ideal t) (enable−individual−pulses t) pulse−list (use−pulse−list t) waveform−spec* [*Function*]

In these functions, if both a WAVEFORM sequence and WAVEFORM-ARGS are included, then WAVEFORM-ARGS is ignored.

A given circuit element may have more than one current source (with different names), but only one voltage source (even though the voltage source has a non-zero source impedance).

For editing properties of individual sources:

edit-source *source* [*Function*]

edit-source-stimulus *source* [*Function*]

where SOURCE is either a voltage or current source, or the name of one.

For convenience, the most recently created current source or voltage source is assigned to the global variables `*ISOURCE*` and `*VSOURCE*`, respectively.

10.2 Current and Voltage Source Driving Functions

Current and voltage sources generate piece-wise linear waveforms which are derived either from user-specified pulse sequences or from a full waveform array assigned to the source. In the latter case, the values in the array may be filled from a choice of various canonical functions or with an arbitrary waveform, using the `ADD-WAVEFORM` function, described below.

Pulses are used as the driving function when the `:USE-PULSE-LIST` slot of the source is T; otherwise a waveform defines the source output. The parameters of the pulses or waveforms are stored in the source `:PARAMETERS`, e.g.:

```
(element-parameters "Hippo-soma-vsrc'') ->

 ((PULSE-TRAIN-ARGS (:START . 50.0) (:STOP . 200.0) (:DELAY . 10.0)
   (:DURATION . 10.0) (:PERIOD . 30.0) (:AMPLITUDE . -30.0))
  (ENABLE-PULSE-TRAIN . T)
  (PULSE-LIST (50.0 90.0 -20.0)))
```

Similar arguments may be included in the `CREATE-PWL-ISOURCE` or `CREATE-PWL-VSOURCE` functions as seen above. See discussion on `PULSE-LIST` below.

For pulse-based waveforms that actually drive a source, each pulse is approximated in a piece-wise linear fashion between the breakpoints of the pulse train, where the slope of each transition is determined by the variables

pwl-isource-di-dt *100.0* [*Variable*]

pwl-vsource-dv-dt *1000.0* [*Variable*]

These values (in nA/msec and mV/msec, respectively) may be edited with the menus.

For waveforms that are derived from array values, the time base for the array is specified in the case of the canonical functions as the STEP parameter (milliseconds), or in the WAVEFORM-TIME-INTERVAL argument of `ADD-WAVEFORM` (default 0.2ms). In either case, the array time base is inverted and assigned to the `:WAVEFORM-TIME-INTERVAL-INVERSE` slot of both current and voltage sources. The output value of a source at a given time point is a piece-wise linear interpoloation between the array values that frame the current time point. This two-point-based interpolation should be kept in mind when assigning a value of the array time base (the time resolution of the array), so that the synthesized waveform is an adequate approximation of the desired source output function.

10.3 Specifying Pulse Sequences and Pulse Trains

Pulses may be defined as either descriptions of individual pulses and/or a description of a pulse train. The final driving function pulse sequence may be a combination of both, depending on the values of `'ENABLE-INDIVIDUAL-PULSES` and `'ENABLE-PULSE-TRAIN` in the source `:PARAMETERS`.

pulse-list *source* &optional *(pulse−list nil pulse−list−supplied)* [*Function*]

This adds a `'PULSE-LIST` entry to the SOURCE `:PARAMETERS`, where the format of PULSE-LIST is either:

```
(pulse-1 pulse-2 ...)
```

or for just a single pulse:

```
pulse
```

and the format of each specific pulse is as follows:

```
(start-time stop-time magnitude), e.g. (4 6 .1)
```

where the magnitude for current sources is in nA, and for voltage sources is in mV. For example,

```
(pulse-list "9-pyr-soma-isrc" '((120.0 170.0 0.5) (220.0 270.0 -0.5)))
(pulse-list *isource* '(10 20 1))
```

The first example sets the current source named "9-pyr-soma-isrc" to give a 50ms 0.5nA pulse starting at 120ms, and a 50ms -0.5nA pulse starting at 220ms. The second example specifies a 1nA pulse from 10ms to 20ms for the last created current source, assigned to `*ISOURCE*`.

This function will also set the `:USE-PULSE-LIST` slot for the source. As above, the magnitude for current sources is in nA, and for voltage sources the magnitude is in mV. The time units are in milliseconds. This function also has no effect on whether a defined pulse train is referenced. If the only argument is the source, then the current pulse list specification is returned.

The value of the source voltage for voltage sources during times outside of the specified waveform is given by the global variable `*VCLAMP-DEFAULT-MAGNITUDE*` (mV). The analagous value for current sources is 0.0.

In general, pulse sequences should be defined so that they are "well-behaved", that is the durations do not overlay (they can be contiguous, however). Surf-Hippo will attempt to synthesize waveforms that "make sense" given various pulse sequences, but if there is any doubt you should verify the resulting waveform by the plotted output of the source (this is an option also when you edit the stimulus).

For example, the following pulse sequence is ok:

```
((10.0 20.0 2.0) (20.0 22.0 -5.0))
```

If a pulse is defined or edited (e.g. via the menus) with the stop time less than or equal to the start time, then that pulse is ignored. This is convenient, for example, when a multiple pulse stimulus is defined. Selected pulses in the train may be temporarily disabled simply by making the stop time negative, without the necessity of removing the pulse or changing any other pulse parameters. If the stop time of a pulse is greater than the stop time of the simulation, then that pulse is ignored.

Pulse trains may be examined or added by:

pulse-train *source* &optional *(start nil start−supplied) stop delay duration period amplitude* [*Function*]

10.4 Specifying Waveforms

Waveforms may be added by:

add-waveform *destination* &key *waveform−spec (waveform−time−interval *default−waveform−step*) delay use−menu float−input breakpoints* [*Function*]

This loads a source with a waveform and its timebase information. WAVEFORM-SPEC is either a sequence of numbers or a function specification (lambda list). If not included, or if WAVEFORM-SPEC is a function spec and USE-MENU is T, the function WAVEFORM-MENU is called. Resulting waveform array is put into :WAVEFORM-ARRAY slot of SOURCE. WAVEFORM-TIME-INTERVAL (ms, default `*DEFAULT-WAVEFORM-STEP*`) must be appropriate for a WAVEFORM-SPEC that is either a number sequence or function spec. DELAY, when not NIL (default) is in milliseconds, and sets the source :DELAY slot directly. SOURCE may also be an electrode, in which case the actual source is extracted with the function `ELECTRODE-SOURCE`. If WAVEFORM-SPEC is a function spec, then it is expected that this function will return a number sequence. For example:

```
(add-waveform *isource* :waveform-spec '(sinewave 1 10 .10 :phase 35 :step 0.2)
```

Note that the waveform function specification is echoed when you print the source characteristics:

```
* (print-element *isource*)
Isource Hippo-soma-isrc (Rint 0.0Mohms, slope 100.0nA/msec)
    SINEWAVE args: AMPLITUDE 1, DURATION 10, FREQUENCY 0.1, OFFSET 0.0, START 0.0, PHASE 35, STEP 0.2
```

On the other hand, if WAVEFORM-SPEC is an explicit number sequence:

```
* (add-waveform *isource*
                :waveform-spec '(0.0 1.0 2.0 1.0 0.0 -1.0 0.0)
                :WAVEFORM-TIME-INTERVAL 1.0)
 NIL
* (PRINT-ISOURCE *isource*)
Isource Hippo-soma-isrc (Rint 0.0Mohms, slope 100.0nA/msec)
  Explicit waveform
```

Here are some waveform functions that come with Surf-Hippo - in general, the units for wave parameters are milliseconds for time related units (e.g. STEP, DELAY, DURATION, TAU), and nA or mV for amplitude related units (e.g. AMPLITUDE, OFFSET):

sinewave &optional *(amplitude 1.0) (duration *user−stop−time*) (frequency 1.0)* &key *(phase 0.0) (offset 0.0) (step 0.2) (start 0.0) zero−before−start* [*Function*]

exponential-array &optional *(tau 1.0) (step 1.0) (length 0) (offset 0.0) (amplitude 1.0) (start 0.0)* [*Function*]

impulse-array &optional *(amplitude 1.0) (duration 1) (delay 0) (step 1.0)* [*Function*]

alpha-array &optional *(tau 1.0)* &key *(time−exponent 1) (adjustment :normalize) (step 1.0) (duration 0.0) (offset 0.0) (amplitude 1.0) (delay 0.0)* [*Function*]

double-alpha-array &optional *(tau1 1.0) (tau2 1.0) (alpha−proportion 1.0)* &key *(offset 0.0) (step 1.0) (tau1−alpha−area 1.0) (start 0.0)* [*Function*]

double-exponential-array &optional *(tau−rise 1.0) (tau−fall 1.0)* &key *(amplitude 1.0) normalize (step 1.0) (length 0) (offset 0.0) (start 0.0)* [*Function*]

10.5 Source Resistance - Ideal Voltage Sources

Both current and voltage sources may have a non-zero internal resistance, which as a first approximation represents the electrode resistance. For nodes with current sources, the voltage measured at the node will be offset by the IR drop across this resistance (given by the variable `*ISOURCE-ELECTRODE-RESISTANCE*` [Mohms, default 0] when the source is created, and which may be edited with the menus). For voltage sources, the source resistance (given by the variable `*VSOURCE-RESISTANCE*` [Mohms, default 0.001 or 1Kohms] when the source is created, and which may be edited with the menus) models the non-ideal voltage clamp. The voltage source resistance for non-ideal sources must be ¿0, since the circuit integration treats voltage sources as a (typically very large) membrane conductance in series with a controlled (time-varying) battery.

For a more sophisticated model of sources, see the discussion on Electrode Model below.

Ideal voltage sources (created by default), on the other hand, by definition have zero internal resistance. To toggle a voltage source between ideal and non-ideal, use the functions:

ideal-vsource *vsource* [*Function*]

non-ideal-vsource *vsource* &optional *resistance* [*Function*]

Circuits with nodes that have ideal voltage sources are solved by creating a new circuit matrix in which these nodes have been removed. The nodes adjacent to these nodes still have a current and jacobian (from the connecting segment axial resistance) contribution from the removed nodes, as if those nodes were still there. Any membrane elements on the removed nodes are evaluated with reference to the value of the associated ideal voltage source (see also Sections 35 and 37).

10.6 Sources May Be En/Disabled

Sources may be enabled or disabled either via the stimulus menus, or using the functions `ENABLE-ELEMENT` or `DISABLE-ELEMENT` (see Section 6.14).

10.7 Voltage Source Current Data

The function `GET-VSOURCE-CURRENT` obtains the voltage source current in various ways:

- If the source is non-ideal, and the global variable `*VSOURCE-INTRINSIC-CURRENT*` is T, then the current is calculated using the voltage drop between the source and the source node, considering the source resistance. Otherwise, the current is summed over all the elements connected to the source node.
- If `*INCLUDE-VSOURCE-CURRENT-LINEAR-AND-NON-LOCAL-COMPONENT*` is NIL (default T), then the set of elements for summing the current does not include the source node leak resistance, the source node capacitance, and the segments and/or soma that are connected to the source node. If the source node is the only node with non-linear elements, then the resulting voltage source current is similar to what is obtained experimentally when the "linear component" of the voltage clamp current is subtracted from the record.
- If `*INCLUDE-LOCAL-CAP-CURRENT-IN-VSOURCE*` is NIL (default T), then the source node capacitance current is ignored.

10.8 Stability of Voltage Sources

There are some subtleties vis-a-vis the integration and the (non-ideal) voltage source model that require further debugging. This shows up as (apparently) bounded oscillations in the voltage of the node with the source, and the source current. These oscillations are more pronounced with non-pulse source waveforms and variable time step integration. The oscillations are also larger with smaller source resistance.

For the present we recommend conservative setting of the numerical parameters, in particular a small value for the global variable `*USER-MAX-STEP*` (default is 0.15 ms, but a better value in this case could be 0.01ms) when using a variable time step, or a similar value for a fixed time step, e.g.:

```
(setq *INTEGRATION-TIME-REFERENCE* :FIXED
      *USER-STEP* 0.01)
```

See also the discussion "Choosing Parameters for Numerical Integration" in the Section 29. Also see the discussion "Voltage Errors" in Section 37.

Since the effective source resistance for ideal voltage sources is of the same order of magnitude as the tree structure (since the source "resistance" as seen by the circuit is taken from the appropriate segment axial resistance), we have not as yet observed stability problems with ideal voltage sources. Thus, in practice, the inclusion of an ideal voltage source poses no additional constraints on the time step parameters.

10.9 Electrode Model

For adding a simple electrode model with a current or voltage source, respectively, to a node:

add-ielectrode *element* &key *(capacitance 0.001) (resistance 1.0e+7) (source−resistance 0.0) (leak−resistance 1.0d+16) name* [*Function*]

add-velectrode *element* &key *(capacitance 0.001) (resistance 1.0e+7) (source−resistance 0.0) (leak−resistance 1.0d+16) name* [*Function*]

These functions add the electrode model illustrated in Figure 2 to CELL-ELEMENT. RESISTANCE and SOURCE-RESISTANCE are in ohms, and CAPACITANCE is in nanofarads. If CELL-ELEMENT is a segment, then the electrode is added to the distal node of the segment. If NAME is not given, the name of the electrode is taken from the name of the CELL-ELEMENT, with "-electrode" added. In fact, the actual circuit includes a linear conductance (1.0^{-10} uS) in parallel with the capacitance, but this is small enough so that it may be ignored. SOURCE-RESISTANCE, the internal resistance of the current or voltage source, is also included in the electrode model (essentially the source added to the electrode here is no different from any other source added to a cell). Thus, it may be important to adjust the source intrinsic resistance accordingly. In some situations, the electrode will be lumped together with the cell segments, for example with the plot output. This is because the electrode model is a special case of the segment model.

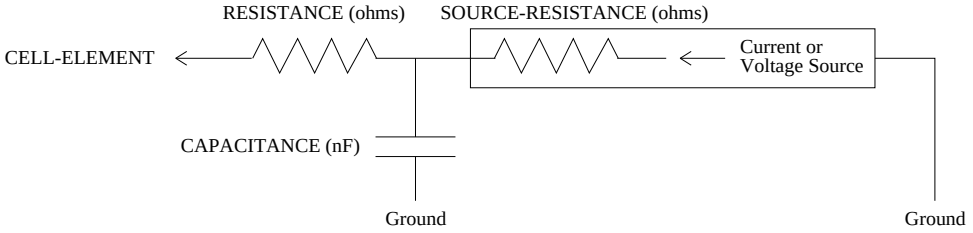

Figure 2: The electrode model generated by `ADD-IELECTRODE` and `ADD-VELECTRODE`.

edit-electrodes &optional *electrode* [*Function*]

This function allows direct menu editing of the electrode(s) axial and "membrane" resistance, and capacitance.

set-electrode-capacitance *c−electrode* &optional *(electrode *electrode*)* [*Function*]

This sets the capacitance of the optional ELECTRODE argument to C-ELECTRODE (nF). If ELECTRODE is not supplied, then the value of `*ELECTRODE*` will be used.

set-electrode-resistance *r−electrode* &optional *(electrode *electrode*)* [*Function*]

Sets the resistance of the optional ELECTRODE argument to R-ELECTRODE (Mohms). If ELECTRODE is not supplied, then the value of `*ELECTRODE*` will be used.

Here is an example of a script that tests the effect of various electrode parameters of the voltage measured at the electrode (assuming that the proper plotting parameters have already been set up, perhaps with the menus):

```
(loop for r-electrode in '(0.01 5.0 10.0 50.0) do
      (loop for c-electrode in '(0.0001 0.001 0.01) do
            (set-electrode-capacitance c-electrode)
            (set-electrode-resistance r-electrode)
            (goferit)))
```

10.10 Voltage Clamp Initialization

Under certain constraints, simulations with voltage sources may be initialized so that the node voltages correspond to a steady-state condition where the voltage source or sources are set to their initial value. This initialization avoids a transient at the start of the simulation whenever the initial voltage source voltage is different from the normal initial voltage of the circuit.

The steady-state algorithm sets all the node voltages in a single step, under the condition that all channels and synapses are blocked. This algorithm works for cells with a single voltage source that is either at the soma, or part of an electrode which in turn is connected to the soma. Future versions of this algorithm will be appropriate for the more general case, i.e. more than one voltage source, or sources at arbitrary locations in the dendritic tree.

The variable

steady-state-linear-vclamp-enable *t* [*Variable*]

enables this option.

10.11 Some Examples

This form adds a current source (not the electrode model) to all the somas in the circuit, and gives each of them a pulse of 1nA starting from 0 milliseconds until 0.5 milliseconds:

```
(loop for soma in (somas) do (pulse-list (add-isource soma) '(0.0 0.5 1.0)))
```

10.12 Adding a Constant Current Source to Somas or Segments

The function

add-constant-current *element* &optional *current* [*Function*]

adds a constant CURRENT [nA] to the somas or segments associated with ELEMENT for the duration of the simulation. This is equivalent to including a current source at the element that has a fixed DC value. If CURRENT is not supplied or NIL, then this will remove any assigned constant current. Likewise, the function

add-constant-current *element* &optional *current* [*Function*]

returns the value for the constant current term for ELEMENT, if any, and the function

clear-constant-currents [*Function*]

removes the constant current term from all the circuit nodes, respectively.

10.13 The Voltage Recorded at Current Source Nodes: Bridge Balance

Current sources have two additional parameters - an internal source resistance (MΩ, in the `:RESISTANCE` slot) and a `:BRIDGE-BALANCE` setting (MΩ, stored in the source's `:PARAMETERS`). These parameters, which may be changed from the `EDIT-ISOURCE` menu, effect *only* the voltage measured at the source node, as shown in Table 2. If there is more than one current source associated with NODE, then these corrections are disabled.

10.14 Steady State Voltage Clamp at the Soma

Steady-state voltage clamp experiments are more conveniently run using the function:

:ENABLE-ISOURCE-DROP	:ENABLE-BRIDGE-BALANCE	$V_r(t)$
NIL	NIL	$V(t)$
T	NIL	$V(t) + (I(t) \times R_{int})$
NIL	T	$V(t) - (I(t) \times R_{bridge})$
T	T	$V(t) + (I(t) \times R_{int}) - (I(t) \times R_{bridge})$

Table 2: Calculation of recorded current source node voltage, taking into account source resistance and bridge balance. $V_r(t)$ is the measured node voltage, $V(t)$ is the actual node voltage, R_{int} (MΩ) is the current source internal :RESISTANCE, R_{bridge} (MΩ) is the current source :PARAMETERS :BRIDGE-BALANCE. Note that the source node may be represent a soma, segment, or electrode. If there is more than one current source associated with the source node, then these corrections are disabled.

steady-state-vclamp *v−holding* &optional *(vsource *vsource*)* [*Function*]

When the vsource is at the soma, the initial voltage for all cell nodes is set according to an imposed holding potential at the soma, and only linear cable/membrane properties. With the latter assumption the starting conditions (potentials) may be found quickly by first calculating (iteratively, from distal to proximal) and storing the equivalent terminating (end) resistance connected to the distal node of each segment. With this information, the segment voltages may be assigned (iteratively, from proximal to distal) in one pass through the dendritic tree. These steps are performed by the function:

steady-state-linear-voltage-clamp *vsource* &optional *holding−potential* [*Function*]

A future revision will apply this algorithm for voltage sources located anywhere in the cell.

10.15 Multiple Current Source Targets

The output of a given current source will be applied to more than one circuit node when there is a circuit element or list of circuit elements stored as a 'TARGETS element parameter for the source. For example,

```
(element-parameters *isource* 'targets (segments))
```

will cause the output of the last defined current source to be applied to all the segments in the circuit. Note that if the 'TARGETS specification includes the original circuit element of the source, then that soma or segment will receive twice the actual output of the source.

11 Libraries for Element Type Types and Parameter Saving

The parameters for the various element type types - cell types, channel types, synapse types, particle types, concentration dependent particle types, concentration integrator types, axon types, pump types, and buffer types - are referenced from parameter libraries specific to each type. This section describes the system for accessing/storing parameter libraries. See also Sections 12, 13, 17 and 6.

11.1 Global Parameter Lists As Element Type Reference Libraries

Each of the libraries for the element type types (organized as association lists for each type) are stored in each element type model `:PARAMETER-TYPE-LIBRARY`. The format and necessary and optional entries for a given element type can be found in the appropriate section of this manual.

The current entries of each library can be found with

library-catalog *reference* &key *only−in−circuit (synapse−control :all) verbose ionic−type* [*Function*]

which returns a list of all types in the type library referenced by ELEMENT.

11.2 Element Type Definition (`Type-Def`) Macros For Updating Parameter Libraries

Normally, the user need not directly manipulate these parameters libraries. Rather, adding (and updating) a new entry of a given element type to the appropriate library is done with the various `Type-Def` macros, including:

channel-type-def *()* &body *body* [*Macro*]

as well as

```
PARTICLE-TYPE-DEF CONC-PARTICLE-TYPE-DEF
SYNAPSE-TYPE-DEF  CONC-INT-TYPE-DEF
BUFFER-TYPE-DEF   PUMP-TYPE-DEF
AXON-TYPE-DEF     CELL-TYPE-DEF
```

The BODY of each `Type-Def` macro is a quoted list whose first element is the name (typically a symbol) of an element type, followed by an association list of parameters specific to that sort of element type. Thus:

```
(FOO-TYPE-DEF
 '(type-name
   (parameter . value)
   (parameter . value)
   ...
   (parameter . value)))
```

where the `TYPE-NAME` symbol is that used by any calls to `CREATE-FOO-TYPE` or `CREATE-FOO` (or in general, using something like `(CREATE-ELEMENT cell-element TYPE-NAME)`). Unless specifically mentioned in this manual, the ordering of the parameter lists in the `Type-Def` form is not important. On the other hand, the "." in each parameter sub-list *is* important, since these lists are considered elements of an association list (this is essentially a detail of Lisp).

For example, the following form:

```
(synapse-type-def
 '(basic-depressing
   (parent-type . auto-fast-ex-double-exp-abs)
   (1st-order-depressing-dynamics . t)
   (tau-recovery . 800) ;ms
   (release-fraction . 0.2)))
```

removes any current entry in `:PARAMETER-TYPE-LIBRARY` slot of the `SYNAPSE-TYPE` model structure whose `CAR` (first element) is the symbol `BASIC-DEPRESSING`, and then adds the new entry to this list. Subsequently, a call to:

```
(create-element 'BASIC-DEPRESSING)
```

will search the `:PARAMETERS-TYPE-LIBRARY` slot for an entry which starts with `BASIC-DEPRESSING`, and then use the associated parameters to create the synapse type (Section 6.2). Likewise,

```
(create-element 'BASIC-DEPRESSING ``J43D-201'')
```

will create a BASIC-DEPRESSING synapse on the cell element named "J43D-201" - if the BASIC-DEPRESSING synapse type doesn't exist yet, then it will be created first, as above.

11.3 `Type-Def` Parameter Keywords and Structure Slots

It is often the case that a parameter keyword in the `Type-Def` form of a particular element type corresponds directly with a slot of the same name in that element type's structure. Thus, in the text of this Manual the significance of a given `Type-Def` parameter may usually interpreted as being assigned directly to the created type structure. Otherwise, the relation between the `Type-Def` parameter and the characteristics of the created element type should be clear. In case of ambiguity, one should refer to the structure definitions in `src/sys/structures.lisp`, or use the `DESCRIBE` on a given instance of a type structure.

For example, the `PARTICLE-TYPE-DEF` for a gating particle associated with a model of the T-type calcium channel (`src/parameters/working-hpc.lisp`) is given by:

```
(particle-type-def
 `(CA-TM-HPC
   (class . :HH-EXT)
   (valence . 3.0)
   (gamma . 0.0)
   (base-rate . 1.0)
   (v-half . -36.0)
   (tau-0 . 1.5)
   (IGNORE-TAU-VOLTAGE-DEPENDENCE . T)
   (reference-temp . 27.0)
   (qten . 1.0)))
```

Many of these parameters are directly reflected in the `PARTICLE-TYPE` structure:

```
* (describe (element 'CA-TM-HPC))
<Particle Type CA-TM-HPC> is a structure of type PARTICLE-TYPE.
NAME: "CA-TM-HPC".
CLASS: :HH-EXT.
Q10: 1.0.
REFERENCE-TEMP: 27.0.
NUMBER-OF-STATES: NIL.
STATE-TRANSITION-ARRAY: NIL.
OPEN-STATE-ARRAY: NIL.
TAU-ARRAY: #(1.5 1.5 1.5 1.5 1.5...).
INF-ARRAY: #(1.7983902e-6 1.819381e-6 1.8406183e-6 1.8621017e-6 1.8838377e-6...).
Z: 3.0.
GAMMA: 0.0.
BASE-RATE: 1.0.
V-HALF: -36.0.
TAU-0: 1.5.
IGNORE-TAU-VOLTAGE-DEPENDENCE: T.
ALPHA-FUNCTION: NIL.
```

```
BETA-FUNCTION: NIL.
TAU-COEFFICIENT: 1.0.
TAU-FUNCTION: NIL.
SS-FUNCTION: NIL.
FIRST-PARTICLE: <Particle HPC-soma-CA-T-HPC-CA-TM-HPC: type CA-TM-HPC>.
PARAMETERS: ((SOURCE . "/usr/local/surf-hippo/bin/parameters/working-hpc.sparcf")
     (QTEN . 1.0) (REFERENCE-TEMP . 27.0)
     (IGNORE-TAU-VOLTAGE-DEPENDENCE . T) (TAU-0 . 1.5) ...).
```

11.4 Source Files are Automatically Registered

The `Type-Def` macros above also store the name of the source (or object) file of the definition, in an entry denoted by `SOURCE`. This information (typically displayed in the `EDIT-ELEMENT` or `PRINT-ELEMENT` routines) may be useful for tracking down some obscure element long after it is loaded.

11.5 Inheritance of Types

The following element type types may be created with reference to a "parent type":

```
CHANNEL-TYPE
PARTICLE-TYPE
CONC-PARTICLE-TYPE
CONC-INT-type
BUFFER-TYPE
PUMP-TYPE
SYNAPSE-TYPE
AXON-TYPE
```

This mechanism requires that the appropriate `Type-Def` form include a `PARENT-TYPE` entry:

```
(thing-type-def
 '(NEW-THING
   (some-parameter . value)
   ...
   (parent-type . SOME-OTHER-TYPE)
   ...
   (another-parameter . value)))
```

Thus, when an instance of `'NEW-THING` is created, it is built upon the parameters defined (in another `Type-Def` form) for `'SOME-OTHER-TYPE`. This base is then modified with any parameters included in the `Type-Def` form for `'NEW-THING`. The parent-type mechanism may be non-recusively nested. Note that for a given element type, the required parameters in the associated `Type-Def` form must be satisfied at minimum in the lowest parent-type of the new type.

When a parent-type is included in a `Type-Def` form, the parent type is *not* created explicitly - there will be no instance of the parent type unless it was created by a top-level create call specific to the parent type.

For a given `TYPE` instance, the `PARENT-TYPES` entry in the `:PARAMETERS` slot, e.g. as returned by:

```
(element-parameters TYPE 'parent-types)
```

has a list of the ancestor parent type symbols, if any, with the immediate parent first. These symbols may them be referenced to the element type library in order to see the original chain of definitions.

11.6 Saving Loadable Element Type Parameters

The following element types have routines, called by:

document-element *element* &optional *model−type* *(circuit−dump* [*Function*]
**document−elements−for−circuit−dump*)*

which will write loadable `Type-Def` forms as above which include the current values of a type's parameters:

```
CHANNEL-TYPE
PARTICLE-TYPE
CONC-PARTICLE-TYPE
SYNAPSE-TYPE
CONC-INT-TYPE
```

The following element types also have a documentation routine which writes out Lisp-loadable forms based on `CREATE`-thing type of functions that reference the current circuit:

```
CHANNEL
SYNAPSE
VSOURCE
ISOURCE
```

Edits of an element type during a simulation may be saved for use later from the menus (select "Information management" from the main menu) or using the function:

dump-elements-file &optional *elements−or−select−each−element* [*Function*]

This function writes a loadable file with `Type-Def` forms for selected (loaded) elements which are element types, and `CREATE` forms for selected elements such as channels, synapses, or sources. Selected elements are determined by the ELEMENT-OR-SELECT-EACH-ELEMENTS argument - this arg can be either a single element, a list of elements, non-NIL (generating a selection menu, or NIL (default) which will select all loaded elements.

Loadable `Type-Def` forms are written to the /data directory with a filename constructed from the simulation name and the date (see Section 24), with the extension ".elts".

11.7 Updating Existing Element Type Types from the Parameter Libraries

It is important to remember that once an element type is created, any subsequent references to that type (such as creating a synapse of a given type) will refer to the existing structure, *not* the current value of the parameter library entry. If you want to update the library (by loading the appropriate `Type-Def` form) and have that change reflected in subsequent references to the type, then you must call:

update-type-from-definition *element* [*Function*]

which updates the ELEMENT type from the most recently loaded library definition. Note that if it is desired to set to NIL a parameter included in a previous library entry, then that parameter must be explicitly included (and given a NIL value) in the updated library (that is in the new `Type-Def` form).

For example, assume that an original particle type `Type-Def` was as follows:

```
(particle-type-def
 '(n-foo
   (class . :hh)
   (tau-function . 30)     ; Constant fixed value for time constant in ms.
   (alpha-function . (lambda (voltage)
                       (let ((v+3 (- voltage -3)))
                         (/ (* 0.003 v+3) (- 1 (exp (/ v+3 -8)))))))
   (beta-function . (lambda (voltage)
                      (let ((v+30 (- voltage -30)))
                        (/ (* -0.0002 v+30) (- 1 (exp (/ v+30 80)))))))))
```

Now, suppose we want to redefine this particle type so that the tau is defined completely by the alpha and beta functions. Normally, this is accomplished by leaving out the `tau-function` entry in the `Type-Def`. However, in order to revamp an existing particle type `N-FOO`, it is necessary to include an explicit NIL entry for `tau-function`:

```
(particle-type-def
 `(n-foo
   (class . :hh)
   (tau-function . NIL)
   (alpha-function . (lambda (voltage)
                       (let ((v+3 (- voltage -3)))
                         (/ (* 0.003 v+3) (- 1 (exp (/ v+3 -8)))))))
   (beta-function . (lambda (voltage)
                      (let ((v+30 (- voltage -30)))
                        (/ (* -0.0002 v+30) (- 1 (exp (/ v+30 80)))))))))
```

Once this new `Type-Def` is loaded, a call to

```
(UPDATE-TYPE-FROM-DEFINITION 'N-FOO)
```

will set up the desired behaviour.

12 Membrane Elements

This section describes properties of all circuit elements that affect the electrical properties of the membrane, including the passive membrane components of somas and segments, channels and synapses. Much of this text is taken from Borg-Graham 1999. See also Sections 11, 15 and 13.

12.1 Models of Pore Conduction

In this section we will describe various models for non-linear channel (and synapse) conduction. These models are at an intermediate level of biophysical detail, appropriate for describing whole cell currents.

For a given current I_X, the most common models for the underlying channel are based on the assumption of a gated, membrane-spanning ionophore, where the mechanism of conduction $f(V, \Delta[X])$ (flow of ions down a voltage gradient), and that of gating $h(V, t, ...)$ (voltage and/or ligand dependent modulation of conduction) are separable:

$$I_X = h(V, t, ...)\ f(V, \Delta[X])$$

Where V is the membrane voltage, t is time and $\Delta[X]$ represents the concentration gradients for the permeable ions of the channel or synapse. The ellipsis in the argument of $h()$ stands for the various ligand-dependent processes (e.g. Ca^{2+}-dependence). The gating term $h()$ for channels is discussed in Section 13.

Here we review two models of the conduction term $f(V, \Delta[X])$: ohmic (thermodynamic equilibrium conduction) and constant-field permeation (non-equilibrium conduction).

12.1.1 Linear Conductance: Driving Force From Equilibrium Thermodynamics

In most cell models channel conduction is taken as ohmic (linear), with the driving force determined by the membrane voltage, $\Delta[X]$, and the relative permeabilities of the ions that pass through the channel. The general form of the ohmic model is:

$$f(V, \Delta[X]) \ = \ \overline{g}_X \ (V - E_X)$$

where, assuming that the units of $f()$ are in nanoamperes, $\overline{g}_X$ is the absolute conductance in μS and E_X is the reversal potential for the channel, the latter given by either the Nernst equation if only one ion is involved, or the Goldman-Hodgkin-Katz (GHK) voltage equation (e.g. Hille, 1992) for more than one ion (voltages in millivolts). In this model, the non-linearity of the conductance term arises implicitely from the Nernst or GHK equation-based estimate of the driving force E_X coupled with a non-zero change in $[X]$ with respect to I_X. Thus, the more current, the more the concentration gradient is reduced, resulting in negative feedback. In practice, many cell models that use the ohmic description neglect changes in concentration, under the assumptions that these activity-driven changes are small.

The use of the Nernst equation for the driving force in the ohmic model is based on the assumption of thermodynamic equilibrium; strictly speaking this corresponds to the situation in which no current is flowing. Thus this description is more appropriate for those ions in which the concentration gradient is not very high and the integrated currents are small compared to the relevant concentrations. These conditions are approximated for K^+ and Na^+, which is similar to saying that these ions are closer to equilibrium. Although widely used (as in all the non-Ca^{2+} channel descriptions we shall present here), the validity of this approximation for describing non-equilibrium conditions (that is, when the neuron is *doing* something electrically) has not been systematically tested.

12.1.2 Setting E_{rev} for Ohmic Channel Models

The most basic way to determine the ions associated with a particular channel experimentally is by estimating the reversal potential and relating that value to that expected from the Nernst or GHK voltage equations. Manipulating the concentrations of specific ions and comparing the change in the reversal potential with that predicted by these equations may also be done. Especially for the case of K^+ channels, there is a dichotomy between the experimental values for E_{rev} and that used by many models, in that in the former case the measured E_{rev} is rarely that predicted for a pure K^+ conductance, whereas in the latter case models often

assume that all K^+ channels share the same $E_{rev} = E_K$. A more flexible approach is to allow the E_{rev} for nominally K^+ channels to be set to within a maximum of 20 millivolts or so above the Nernst E_K (typically this is -90 to -95mV), a range typically seen in the data. We have found that this additional free parameter allows for some useful fine tuning in the shape of the afterpotentials.

We may note as well that the value of E_K (and sometimes E_{Na}) in some models is not given explicitly, despite the fact that there are no reliable canonical values, and variations in these parameters can have a significant effect on cell behaviour.

12.1.3 Constant Field Conduction Model

As a particular ion moves farther from equilibrium (specifically the case for Ca^{2+}), the ohmic model becomes less accurate. One of the most widely used non-equilibrium models of pore conduction permeation is the constant field model, described by the Goldman-Hodgkin-Katz (GHK) current equation. In this equation (see Jack et al., 1983 and Hille, 1992), the non-linearity of the permeation term is explicit:

$$f(V, \Delta[X]) = \overline{p}_X \frac{Vz^2F^2}{RT} \frac{[X]_{in} - [X]_{out} \exp(-zFV/RT)}{1 - \exp(-zFV/RT)}$$

where, again assuming that the units of $f()$ are in nanoamperes, R is the gas constant, F is Faraday's constant, T is temperature in degrees Kelvin, $\overline{p}_X$ is the permeability (*not* the conductance) of the channel in cm^3/second, $[X]_{in}$ and $[X]_{out}$ are the intracellular and extracellular concentrations in millimolar and z is the valence of the permeant ion - here we assume that channel conduction is mediated by only one type of ion. The non-linear behaviour of this term manifests itself as negative feedback for inward currents, particularly Ca^{2+} (e.g. Hess and Tsien, 1984), both because of the sublinear characteristic at typical values of $[Ca^{2+}]$ and because the driving force is reduced as the gradient of $[X]$ is reduced with current flow. We note, however, that the dynamic change of this characteristic due to typical concentration changes is not very large.

12.2 Deriving the Reversal Potential for Channels and Synapses

Reversal potentials are determined by either fixed values or in reference to intra- and extracellular concentrations of ions whose relative contributions are given in the :ION-PERMEABILITIES slot.

In a `CHANNEL-TYPE-DEF` or `SYNAPSE-TYPE-DEF` form, the presence of `E-REV` and/or `USE-DEFINED-E-REV` entries set up the method for the reversal potential calculation:

E-REV	USE-DEFINED-E-REV	Method	:USE-DEFINED-E-REV slot
number	no entry	Reference fixed value	T
no entry	N/A	Reference ion permeabilities	nil
N/A	nil	Reference ion permeabilities	nil

In the last case, there must be an `ION-PERMEABILITIES` (or `ION-PERMS`) entry in the Type-Def form. This entry is either a single symbol corresponding to the ion type (if the pore is only permeable to that ion);

```
(channel-type-def
 '(bar
...
   (ion-perms . na)
...
))
```

or a list of ions with associated relative permeabilities:

```
(channel-type-def
 '(foo
...
   (ion-perms . ((K 0.85) (NA 0.15)))
...
))
```

The relevant concentrations (as specified by the ion permeabilities) are taken from associated concentration integrators or the appropriate cell type concentrations, or global values for concentrations.

Thus, for a given channel or synapse type, when :USE-DEFINED-E-REV is nil and :ION-PERMEABILITIES is set to a ion permeability list, reversal potential is set with the function EFFECTIVE-REVERSAL-POTENTIAL:

effective-reversal-potential *ion−perms* &optional *element* [*Function*]

Calculate reversal potential based on ion-permeabilities and the associated reversal potentials, which in turn reference the appropriate cell type or the DEFAULT-ION-REVERAL-POTENTIAL:

default-ion-reversal-potential *species* &optional *value* [*Function*]

This sets the default (fixed) reversal potential for ion SPECIES (NA, K, CL, CA, MG) if VALUE [mV] supplied. Returns the current value.

12.3 Deriving the Maximum Conductance for Channels and Synapses

The :GBAR, or maximum (peak) conductance of an individual channel or synapse may be determined either by the area of the associated cell element and conductance density (:GBAR-DENSITY), or by a fixed constant value (either :GBAR-REF or :GBAR). The method used depends on the :GBAR-SOURCE slot, which should be either :DENSITY or :ABSOLUTE. In either case, the final value of :GBAR is calculated by reference to the relevant type (channel type or synapse type) if the :INHERIT-PARAMETERS-FROM-TYPE slot for the channel or synapse is T, or from the channel or synapse itself. In summary, given a channel or synapse ELT, the calculation of the final :GBAR is given in the following table.

:INHERIT-PARAMETERS-FROM-TYPE slot of ELT	T	T	NIL	NIL
:GBAR-SOURCE slot of ELT type	:DENSITY	:ABSOLUTE	n/a	n/a
:GBAR-SOURCE slot of ELT	n/a	n/a	:DENSITY	:ABSOLUTE
Gbar calculated with reference to	:GBAR-DENSITY of type	:GBAR of type	:GBAR-DENSITY of ELT	:GBAR-REF of ELT

The final value for the ELT :GBAR is then taken from the density or absolute reference, taking into account the appropriate conductance :QTEN of the type (not that for the gating kinetics), and multiplied by the 'GBAR-MODULATION entry in the type's :PARAMETERS. If there is no 'GBAR-MODULATION entry, then this is taken to be 1.0. Note that if a synapse or channel type includes a 'GBAR-MODULATION, then this factor applies to the synapses or channels of that type whether or not the specific instances have their :INHERIT-PARAMETERS-FROM-TYPE slot set.

For example, with light synapses in which the receptive fields of the synapses are mapped retinotopically with respect to their location on the cell (e.g. retina), then it may be more appropriate to use synapse types in which the peak conductance is defined in terms of cell element area. For synapses who are activated independently of each other, then it may be more appropriate to use a type for which there is a fixed peak conductance, and then define (create) the appropriate number of individual instances of that synapse type to achieve the desired synaptic (conductance) density.

The channel or synapse type definition stored with CHANNEL-TYPE-DEF or SYNAPSE-TYPE-DEF can include a GBAR-SOURCE entry (otherwise the default in :DENSITY). Also, these parameters may be edited in the EDIT-ELEMENT menu, called on the channel, channel type, synapse or synapse type.

12.4 Setting Membrane Element Parameters

The following function specifies various parameters for channel and synapse types:

iv-type-parameter *element param* &optional *(value nil value−supplied−p) (update t)* [*Function*]

which is similar to ELEMENT-PARAMETER, but for examining/setting specific parameters of the synapse or channel type associated with ELEMENT. PARAM can be:

```
:IV-RELATION-SOURCE (e.g. :ABSOLUTE or :DENSITY)
:IV-RELATION-REF [for :ABSOLUTE gbar (uS) or permeability (cm3/sec)]
:IV-RELATION-DENSITY [pS/um2 (0.1mS per square cm) for gbar, 1.0e-6 cm3/sec/um2 for permeability]
:IV-RELATION-MODULATION [applied to all type children, regardless of inheritance]
:E-REV [mV]
:BLOCKED [T or NIL]
```

If no new VALUE follows the PARAM, then the current value of the slot corresponding to PARAM is returned. Supplying a non-nil value for UPDATE will cause the change to propagate to the appropriate elements of the type associated with ELEMENT.

The following function specifies an individual conductance reference for channels or synapses:

set-element-absolute-iv-relation-ref *element iv−reference* [*Function*]

The ELEMENT argument can be either an instance of a synapse or a channel, or the name of a synapse or a channel. In SET-ELEMENT-ABSOLUTE-GBAR-REF, GBAR-REF is in μS. This function also resets the :INHERIT-PARAMETERS-FROM-TYPE slot of ELEMENT to NIL, so that the newly assigned values are not overridden by the corresponding element type parameters. It returns the numerical second argument. To get the current value, use ELEMENT-GBAR.

12.5 Updating Membrane Element Parameters that are Dependent on Temperature and Membrane Area

At various points in the program flow, prior to the actual simulation, the function SET-CIRCUIT-ELEMENTS-PARAMETERS is called. This function in turn calls functions which update element parameters which may depend on membrane area or temperature. To see exactly what is updated for a given type of membrane or cell element, see the appropriate function definition as indicated in SET-CIRCUIT-ELEMENTS-PARAMETERS (each update function is found in the corresponding element type file, e.g. the function SET-CHANNELS-PARAMETERS is defined in channel.lisp). Many of these functions are written so that element parameters are inherited from the appropriate element types, depending on the :INHERIT-PARAMETERS-FROM-TYPE slot as described above.

If you want to disable this update for a certain class of elements, SETQ the appropriate global enable variable:

```
*ENABLE-SEGMENT-MEMBRANE-PARAMETER-UPDATE*
*ENABLE-SOMA-MEMBRANE-PARAMETER-UPDATE*
*ENABLE-CONC-INTEGRATOR-MEMBRANE-PARAMETER-UPDATE*
*ENABLE-CHANNEL-TYPE-MEMBRANE-PARAMETER-UPDATE*
*ENABLE-CHANNEL-MEMBRANE-PARAMETER-UPDATE*
*ENABLE-AXON-MEMBRANE-PARAMETER-UPDATE*
*ENABLE-SYNAPSE-MEMBRANE-PARAMETER-UPDATE*
```

The default for these variables is T. It is likely that you will not need to change these variables.

12.6 Aribitrary Conductance Functions

For channels and synapses an arbitrary conductance coefficient may be applied to the standard evaluated conductance (or permeability) value by including

```
(conductance-function .  function-name)
```

in the CHANNEL-TYPE-DEF or SYNAPSE-TYPE-DEF. This function should take a single channel or synapse argument, as appropriate, and return a single float number. This value will then be used as a coefficient for the evaluated conductance value of the channel or synapse.

12.7 Static Voltage Dependence of Conductances

Likewise, the conductance of a channel or synapse can also have a static dependence on the current voltage of the associated node. Thus, once the channel or synapse conductance is determined by the standard methods, the result is then multiplied by the value obtained by the static voltage dependence. This dependence is specified when the type definition parameters stored with CHANNEL-TYPE-DEF or SYNAPSE-TYPE-DEF includes:

```
(static-voltage-dependence-function .  (seq-function args ...))
```

The form in SEQ-FUNCTION should return a sequence used as a look up table, whose index runs from -200 (given by *PARTICLE-LOOK-UP-TABLE-MIN-VOLTAGE*) to 200 mV (given by *PARTICLE-LOOK-UP-TABLE-MAX-VOLTAGE* - this is the range for the particle voltage-dependent function arrays as well - check the value of *PARTICLE-LOOK-UP-TABLE-VOLTAGE [normally 400]), with a resolution given by *PARTICLE-LOOK-UP-TABLE-PRECISION* [normally 0.1]. For example, the function:

sigmoid-array *v−half slope vmin vmax vres* [*Function*]

could be used, specified with:

```
(synapse-type-def
 `(nmda
   (parent-type . fast-ex)
   (static-voltage-dependence-function .  (SIGMOID-ARRAY
                                           -50.0 0.5
                                          ,*PARTICLE-LOOK-UP-TABLE-MIN-VOLTAGE*
                                          ,*PARTICLE-LOOK-UP-TABLE-MAX-VOLTAGE*
                                          ,*PARTICLE-LOOK-UP-TABLE-PRECISION*))))
```

Note that if explicit references to global variables are used as function arguments, as here, they must by evaluated (here, by using the backquote list syntax, and adding a comma prior to each variable). Of course, if the values are known in advance (usually the case), a simpler format would be:

```
(synapse-type-def
 `(nmda
   (parent-type . fast-ex)
   (static-voltage-dependence-function . (SIGMOID-ARRAY -50.0 0.5 -200.0 200.0 0.1))))
```

The function and arguments supplied as a STATIC-VOLTAGE-DEPENDENCE-FUNCTION entry are evaluated only once, when the channel or synapse type is first created. Alternatively, an explicit numeric sequence may be used, again with the characteristics determined as above. In principle, the CONDUCTANCE-FUNCTION parameter described above could accomplish the same task, but the use of an explicit, pre-calculated look up table of voltage may be much more efficient than calling a function with every channel or synapse evaluation.

12.8 Miscellaneous

channel-types-of-ion-type *ion−type* &optional *(only−loaded t) (only−in−circuit t) exclude−conc−dependent−types* [*Function*]

synapse-types-of-ion-type *ion−type* &optional *only−loaded* *only−in−circuit* *exclude−conc−dependent−types* [*Function*]

For both of these functions, ION-TYPE is a symbol that is matched to entries in the `:ION-PERMEABILITIES` slot of channel or synapse types. This function searchs through all the entries of defined by `CHANNEL-TYPE-DEF` or `SYNAPSE-TYPE-DEF` as appropriate and returns a list of all the channel types referenced there whose `:ION-PERMEABILITIES` includes ION-TYPE. This function will create instances of channel or synapse types that satisfy this criteria if the types don't already exist. For example:

```
* (CHANNEL-TYPES-OF-ION-TYPE 'na nil nil)
(<Channel type NA-HH-EXT> <Channel type NA-HH> <Channel type NA3>
 <Channel type NA2> <Channel type NA1> <Channel type NA-TRB>
 <Channel type NA-FG> <Channel type NA-RHO> <Channel type NA-SGB>
 <Channel type NA-WDY>)
```

Note that the `:ION-PERMEABILITIES` specification is not required for channel or synapse types, i.e. it may be NIL.

13 Channels and Gating Particles

This section describes the channel and particle models. Much of this text is taken from Borg-Graham 1999.

13.1 Description of Channel and Particle Models

Channel conductances are governed by the general equation:

$$g_{ch}() = \prod {v_n}^{x_n} \prod {c_m}^{y_m} \bar{g_{ch}}$$

where v_n are x_n voltage dependent gating particles and c_m are y_m concentration dependent gating particles associated with each channel protein.

Hodgkin and Huxley (1952b) described channel gating as an interaction between different, independent two-state (open and closed) elements or "particles", all of which must be in the open state for channel conduction. The state dynamics of each particle are described with first order kinetics:

$$x_C \overset{\alpha_x(V),\beta_x(V)}{\rightleftharpoons} x_O \tag{2}$$

where x_C and x_O represent the closed and open states of gating particle x, respectively. $\alpha_x(V)$ and $\beta_x(V)$ are the forward and backward rate constants of the particle as a function of voltage, respectively. Taking x to be the probability of the open state, the associated differential equation is:

$$\frac{dx}{dt} = \frac{x_\infty(V) - x}{\tau_x(V)}$$

with the steady state value of x, $x_\infty(V)$, and the time constant, $\tau_x(V)$ given by $\alpha_x(V)$ and $\beta_x(V)$ (see below). A useful generalization of this kinetic scheme is to assume that the rate constants can be functions of other signals (e.g. $[Ca^{2+}]$); the general form of the gating term in this scheme is the product of the open probability of the channel's gating particles:

$$h(V,t,...) \;=\; \Pi x_i^{n_i}(V,t,...)$$

where x_i is the open probability of a given type of gating particle, and n_i is the number of particles of a given type associated with the channel. This system is a specific type of Markovian model (see Section 14), constrained by the independence condition; the general Markovian model allows arbitrary transitions between the states describing the channel kinetics.

Two-state voltage dependent gating particles in Surf-Hippo follow the Hodgkin-Huxley model of first-order kinetics, where the rate constants (and thus the time constant and steady-state value) are voltage dependent. The equations for the rate constants (or alternatively, the tau and steady-state values) may be supplied explicitly (particle type class :HH and others). Otherwise, they may be derived from a single-barrier, extended Hodgkin-Huxley model (particle type classes :HH-EXT or :HH-EXT-OLD), modified somewhat from the model described in Borg-Graham 1991. In the latter case, the parameters of the particle rate equations are Z (effective valence of the gating particle), GAMMA (symmetry), BASE-RATE (reference rate constant, called alpha-0 in the above reference, in 1/ms), V-HALF (midpoint for voltage dependence, in mV) and TAU-0 (minimum time constant for particle transitions, in ms), and voltage independent additive forward and backward rate constants (ALPHA_0 and BETA_0, in 1/ms). The precise application of TAU-0 depends on whether the particle type class is :HH-EXT-OLD (the version described in Borg-Graham '91) or :HH-EXT:

For :HH-EXT particle types:

$$\texttt{Tau(V)} = \frac{1}{\texttt{alpha + beta}} + \texttt{TAU-0}$$

For :HH-EXT-OLD particle types:

$$\text{Tau(V)} = \text{Maximum}\left[\ \frac{1}{\text{alpha} + \text{beta}}\ ,\ \text{TAU-0}\ \right]$$

In both cases alpha and beta are both functions of z, gamma, base-rate, v-half, alpha_0 and beta_0 (respectively) and a temperature term (see below). The newer version of :HH-EXT model correctly follows the original hypothetical basis for the TAU-0 parameter, namely that there is a voltage-independent rate-limiting step for the gating particle transitions.

For particle types in which either there is no data on the voltage dependence, or there is no observed voltage dependence, `:IGNORE-TAU-VOLTAGE-DEPENDENCE` is T, and the `:TAU-0` value is the constant time constant for this particle type (see below).

Thus, the definition of a new channel consists of providing parameters for the channel type, and the parameters for the particle types that the channel type must refer to. Channel type, voltage-dependent particle type, and concentration-dependent particle type parameters are stored in a list formats, which are in turn passed to the macros `CHANNEL-TYPE-DEF`, `PARTICLE-TYPE-DEF`, or `CONC-PARTICLE-TYPE-DEF` as appropriate.

Reversal potential calculation is discussed in Section 12.

Channel type parameter lists must have a value for either `GBAR-DENSITY` or `GBAR`.

For example, a calcium-dependent potassium channel of the type described in Borg-Graham (1991) may be defined as follows:

```
(channel-type-def
 '(c
   (gbar-density . 40.0)
   (e-rev . -85.0)
   (ion-permeabilities . ((K 1.0)))
   (QTEN . 1.5)
   (reference-temp . 30.0)
   (v-particles . ((cx 3)(cy 1)))
   (conc-particles . ((cw 1)))))
```

This expression supplies parameters for a 'C channel, with the following characteristics:

```
Conductance density = 40 pS per square micron or 40.0e-4 S/cm2
Reversal potential = -85 mV
Permeable to K^+ ions only
Q_{10} = 1.5, referenced to 30 degrees C
2 types of voltage dependent particles - 3 of the CX type,
and 1 of the CY type
1 type of concentration dependent particle - 1 of the CW type
```

For the Hodgkin-Huxley DR channel in the squid:

```
(channel-type-def
 '(dr-hh
   (gbar-density . 360.0)
   (e-rev . -77.0)
   (ion-permeabilities . ((K 1.0)))
   (v-particles . ((n-hh 4)))))
```

This implies:

```
Conductance density = 360 pS per square micron
Reversal potential = -77 mV
Permeable to K^+ ions only
1 types of voltage dependent particle - 4 of the N-HH type,
```

The particle type parameters referenced by the C channel type above are defined with the following expressions:

```
(particle-type-def
 '(cx
   (class . :hh-ext)
   (VALENCE . 25)
   (GAMMA . 0.2)
   (BASE-RATE . 0.007)
   (V-HALF . -65.0)
   (TAU-0 . 0.25)
   (QTEN . 3)
   (reference-temp . 30.0)))

(particle-type-def
 '(cy
   (class . :hh-ext)
   (VALENCE . -20)
   (GAMMA . 0.8)
   (BASE-RATE . 0.01)
   (V-HALF . -60.0)
   (TAU-0 . 15.0)
   (QTEN . 3)
   (reference-temp . 30.0)))

(conc-particle-type-def
 '(cw
   (class . :nth-order)
   (alpha . 200.0)
   (beta . 0.125)
   (power . 3)
   (QTEN . 3)
   (conc-int-type . ca-in)
   (reference-temp . 30.0)
   (Fixed-boltzmann-reference-temperature . 30.0)
   (shell . 1)))
```

For the single N particle (here called N-HH) in the Hodgkin-Huxley DR channel:

```
(particle-type-def
 `(n-hh
   (class . :hh)
   (alpha-function . (lambda (voltage)
                       (let ((v-55 (- voltage -55.0)))
                         (/ (* -0.01 v-55)
                            (1- (exp (/ v-55 -10.0)))))))
   (beta-function . (lambda (voltage)
                      (* 0.125 (exp (/ (- voltage -65.0) -80.0)))))))
```

Thus, descriptions for particles of the :HH type require a `LAMBDA` form or function name for either both `ALPHA-FUNCTION` and `BETA-FUNCTION`, or both `TAU-FUNCTION` and `SS-FUNCTION` entries, in the form above. For these particle type functions, either the `LAMBDA` forms or the functions referenced by name must process a single required argument, membrane voltage (in mV). See also Section A.11.

Note that the use of `ALPHA-FUNCTION` here refers to the voltage dependence of the forward rate constant for gating particle transitions, and should not be confused with the alpha function form often used as empirical descriptions of synaptic conductance waveforms, etc. Also, the use of the *symbol* `ALPHA-FUNCTION`

in the context of gating particles does not interfere with the Lisp function `ALPHA-FUNCTION` - the usages are independent.

The final voltage dependent steady-state curves for the particles are rectified to avoid a negative values from the tails of the defining steady-state or rate contant functions.

13.2 Voltage Independent Components and Fixed Time Constants for :HH-EXT Particle Types

Note that there are explicit voltage independent components of the :HH-EXT formulation: `ALPHA_0`, `BETA_0` assume that there is a parallel voltage-independent transition between the two states, while `TAU-0` assumes that there is a voltage-independent rate-limiting component to the voltage dependent rates.

If either the `BASE-RATE` entry `PARTICLE-TYPE-DEF` is missing for a :HH-EXT particle, or `(IGNORE-TAU-VOLTAGE-DEPENDENCE . T)` is included, then the `:IGNORE-TAU-VOLTAGE-DEPENDENCE` of the type is set T.

In these cases, the time constant for the particle type is determined by the value of `TAU-0.`

13.3 Summary of Extended HH Model Conventions

Given `ALPHA(V)` as tThe forward rate constant of a particle as a function of voltage, that is the transition of a particle from the closed to the open state, and `BETA(V)` as the backward rate constant of a particle as a function of voltage, that is the transition of a particle from the open to the closed state,. in Surf-Hippo the conventions for the parameters of the HH extended model are as follows:

- `K` The leading coefficient (1/ms) for the voltage dependent components of both `ALPHA(V)` and `BETA(V)`. Also referred to as the `BASE-RATE`.
- `Z` Valence of the gating particle, where a positive value means that the particle tends to the open position with depolarization - i.e. it is an activation particle. Likewise, a negative value of the valence means that the particle tends to the open position with hyperpolarization - i.e. it is an inactivation particle. Note that this definition of the sign convention is opposite to that found in Borg-Graham (1991), for no profound reason. However, the convention used here is the same as that used in Jack, Noble, and Tsien (1983 , eq. 8.33-34, p. 242), and depends only on the (arbitrary) polarity of the gating particle with respect to the membrane inner and outer surface.
- `GAMMA` Asymmetry of the gating particle voltage sensor within the membrane - the value of `GAMMA` is a coefficient of the (shifted) voltage in the argument of the exponental term for `ALPHA(V)`, whereas `(1 - GAMMA)` is a coefficient of the (shifted) voltage in the argument of the exponental term for `BETA(V)`.
- `ALPHA_0` Voltage independent additive forward rate constant.
- `BETA_0` Voltage independent additive backward rate constant

$$\alpha'_x(V) = K \exp\left(\frac{z\ \gamma\ (V - V_{1/2})\ F}{R\ T}\right)$$

$$\beta'_x(V) = K \exp\left(\frac{-z\ (1-\gamma)\ (V - V_{1/2})\ F}{R\ T}\right)$$

An additional parameter, τ_0, (not the same as the linear membrane time constant) represents a rate-limiting step in the state transition, for example "drag" on the particle conformation change.[1] This parameter is crucial for fitting the expressions to the original Hodgkin-Huxley equations. There are two equivalent ways

[1]The formulation in Borg-Graham (1991) used this parameter as a lower bound for $\tau_x(V)$. The form defined above not only avoids discontinuities in the characteristic but also seems more plausible. Similar considerations have been explored for other, more general, kinetic schemes, e.g. Patlak, 1991.

of incorporating τ_0 into the kinetics. The more intuitive approach is to include τ_0 explicitly in the expression for the time constant $\tau_x(V)$ of the particle state differential equation:

$$\tau_x(V) = \frac{1}{\alpha'_x(V) \ + \ \beta'_x(V)} \ + \ \tau_0$$

We can also derive the appropriate forms for $\alpha_x(V)$ and $\beta_x(V)$ that characterize the kinetics of Equation 2:

$$\alpha_x(V) = \frac{\alpha'_x(V)}{\tau_0(\alpha'_x(V) + \beta'_x(V)) + 1}$$

$$\beta_x(V) = \frac{\beta'_x(V)}{\tau_0(\alpha'_x(V) + \beta'_x(V)) + 1}$$

Note that when τ_0 is 0, $\alpha_x(V)$ is equal to $\alpha'_x(V)$ and $\beta_x(V)$ is equal to $\beta'_x(V)$. Finally, the expression for $x_\infty(V)$ is:

$$x_\infty(V) = \frac{\alpha'_x(V)}{\alpha'_x(V) \ + \ \beta'_x(V)} = \frac{\alpha_x(V)}{\alpha_x(V) \ + \ \beta_x(V)} \tag{3}$$

Two additional parameters, α_0 and β_0, may be considered in some cases, though they are not necessary in reproducing the original Hodgkin-Huxley equations. These parameters are forward and backward rate constants, respectively, of parallel state transitions which are voltage-independent, but nonetheless figure into the total kinetics. If considered, α_0 and β_0 alter the expressions for $\alpha_x(V)$, $\beta_x(V)$, and thus $\tau_x(V)$ and $x_\infty(V)$, which we will not consider here.

13.4 Creating Channels and Channel Types

The basic function to create a channel is:

create-channel *element type* &key *pre−synaptic−element conc−int−delta* [*Function*]

The TYPE argument references a type description that was defined with a `CHANNEL-TYPE-DEF` entry, as described above, and the channel will be added to the cell element associated with ELEMENT. Note that if the keyword arguments PRE-SYNAPTIC-ELEMENT or CONC-INT-DELTA are not required, then `CREATE-ELEMENT` may be used just as well to create the channel.

The keyword argument CONC-INT-DELTA is for adding a coefficient (less than or equal to 1) that will adjust the channel current integrated by an associated concentration integrator. The delta term is used when a channel which is in real life distributed over several (say, N) elements is assigned in the circuit model to a fewer (< N) number of elements, possibly only one. In this case delta, which is less than or equal to one, compensates for the overestimate of the integrated current in the higher density channel distribution. For example, if the total membrane area of the elements for which a given channel is postulated to be assigned to is area-true, and the area of the element(s) that the channel is actually assigned to is some smaller value area-actual, then

```
delta = area-actual/area-true
```

Other heuristics for adjusting this parameter are possible. The default value for delta is 1.0 (see also Section 17.9).

When supplied, PRE-SYNAPTIC-ELEMENT specifies the cell element which controls the channel, otherwise taken as the cell element associated with ELEMENT (see Section 15.1).

13.5 Channel Maximum Conductance

See Section 12.

13.6 Q_{10}'s for Channels and Particles

The Q_{10} parameter (denoted by the symbol `Q10` in `CHANNEL-TYPE-DEF`, `PARTICLE-TYPE-DEF` and `CONC-PARTICLE-TYPE-DEF`) refers to the temperature dependence of the channel base conductance or permeability (gbar), and the temperature dependence of the particle kinetics. The `REFERENCE-TEMP` entries are the reference temperatures for the various Q_{10} factors.

```
(defvar *ignore-q10* nil)
```

When this flag is true then the only temperature dependence in the (:HH-EXT class) particle kinetics (rate constant calculation) occurs via the `1/T` factor in the exponential argument of the Boltzmann term (see also Section 16). Thus, for gating particles that use the extended HH model, there is an "intrinsic" temperature dependence in the Boltzmann term of the rate constants, which as a default refers to the current simulation temperature. The complete temperature dependence of a particle of this class arises from both this term and the explicit Q_{10} coefficient. The Boltzmann term may be nullified, however, by including the following in the particle type's parameter list used in the `PARTICLE-TYPE-DEF`:

```
(Fixed-boltzmann-reference-temperature . 30.0)  degrees celcius
```

In this case, the temperature term in the Boltzmann term is taken from this value (which may be changed with the menus, or by calling:

```
(element-parameter PARTICLE-TYPE 'fixed-boltzmann-reference-temperature VALUE)
(element-parameter PARTICLE-TYPE 'use-fixed-boltzmann-reference-temperature t)
```

on particle type PARTICLE-TYPE, using a single-float VALUE (in degress celcius). The second call sets a flag which enables the use of this reference temperature.

13.7 Modifying Channel and Particle Parameters

A series of menus are available for editing the properties of channel types and their associated particle types. These menus also include plots of particle rate equations as functions of voltage, and steady-state IV curves for the cell.

When trying to fit parameters of channels/particles, it is convenient to save updated parameters using the:

```
"Dump loadable circuit element definitions:"
```

option from the save data menu. This creates a file for circuit element types that can be reloaded back into Lisp. This is especially handy for simulation sessions in which element type parameters have been changed, since it allows for almost automatic restoral of circuit state in a new session.

13.8 Integration of Particle States

All the voltage-dependent time constant and steady-state functions for the particles are precomputed and stored in arrays for fast reference during the integration. These arrays are automatically re-evaluated whenever particle parameters are changed. For details on the numerical method used for integrating particle states, see Section 33.

13.9 Specifying Explicit Initial Conditions for Particles

Normally, the initial condition for a gating particle is determined by the initial voltage, concentrations, etc. that determine the dynamics of the particle. On the other hand, an explicit value for the initial state may be defined by including a number, between 0 and 1 (naturally), in the particle type or particle parameters, e.g.:

```
(element-parameter SOME-PARTICLE-TYPE 'initial-state 0.1)
```

If a particle has a parameter value defined for `'initial-state`, this number will supercede that defined, if any, for the particle type.

13.10 Precision Of Voltage Dependent Kinetics Lookup Tables

The precision of the table lookup for voltage-dependent rate constants (or equivalently, time constant and steady-state values) is determined by the constant *PARTICLE-LOOK-UP-TABLE-PRECISION*, which is set to 0.1mV. If necessary, interpolation is applied depending on the value of

interpolate-particle-arrays *nil* [*Variable*]

which if true, results in an interpolation between lookup table entries during the particle evaluations.

13.11 Summary of CHANNEL-TYPE-DEF Format

See also Section 11.
Gbar Parameters:

```
(gbar-density . 130.0)     <-   ps/um2 (pS/um2 = 1.0e-12 S / 1.0e-8 cm2 = 1.0e-4 S/cm2)
   or
(gbar . 3.0)               <-   microsiemens
```

Ion Permeabilities and Reversal Potential:

```
(e-rev . 50.0)                      <-   mV
(ion-permeabilities . ((NA 1.0)))  <- ((ion-1 relative-perm) (ion-2 relative-perm) ...)
(use-variable-e-rev . t)
   or
(use-defined-e-rev . t)
```

Conductance Intrinsic Temperature Dependence:

```
(QTEN . 1.5)
(reference-temp . 24.0)            <-   degrees C
```

Concentration Integrator Parameters:

```
(conc-int-type-params . ((ca-in1-wdy (1 .7))
                         (ca-in2-wdy (1 0.024))))
(conc-int-type-e-rev-params . ((ca-in1-wdy (1 1))))
```

Gating Particles:

```
(v-particles . ((m1 2) (H1 1)))
(conc-particles . ((ahpw 1)))
```

13.12 Summary of PARTICLE-TYPE-DEF Format

- :HH Class

```
(particle-type-def
 '(m-hh
   (class . :hh)
   (alpha-function . (lambda (voltage)
                       (let ((v-40 (- voltage -40.0)))
                         (/ (* -0.1 v-40)
                            (1- (exp (/ v-40 -10.0)))))))
   (beta-function . (lambda (voltage)
                      (* 4.0 (exp (/ (- voltage -65.0) -18.0)))))))
```

These functions may be optimized, if desired, as follows:

```
  (lambda (voltage)
   (declare (optimize (speed 3) (space 0))
            (single-float voltage))
   (* 4.0 (exp (/ (- voltage -65.0) -18.0))))
```

Instead of explicit functions for alpha and beta, there may be instead explicit functions for:

```
(particle-type-def
 `(foom-bar
   (class . :hh)
   (tau-function . 'foom-bar-tau)
   (ss-function . 'foom-bar-ss)))
```

When :HH class particle type kinetics are derived from explicit functions of alpha and beta, then the voltage depdendent time constant may have a scaling factor *tau-coefficient*:

```
   (tau-coefficient . 0.8)
```

- :HH-EXT Class

```
(particle-type-def
 '(m-hh-fit
   (class . :hh-ext)
   (valence . 2.7)
   (gamma . 0.4)
   (base-rate . 1.2)
   (v-half . -40.0)
   (tau-0 . 0.07)))

(particle-type-def
 `(kdx-gen
   (class . :hh-ext)
   (valence . 3.0)
   (gamma . 0.0)
   (base-rate . 1.0)
   (v-half . -63.0)
   (tau-0 . 1.0)
   (ignore-tau-voltage-dependence . T)
   (reference-temp . 35.0)
   (fixed-boltzmann-reference-temperature . 27.0)
   (qten . 1.0)))
```

13.12.1 Changing explicit TAU and SS specifications

Two functions allow changing the specification for the time constant and steady-state of two state gating particles, in general as a function of voltage:

set-particle-type-tau *type tau−form* [*Function*]

set-particle-type-ss *type ss−form* [*Function*]

Each can accept either a number, representing a fixed time constant in milliseconds, or fixed steady-state value between 0 and 1, respectively, or a function name or lambda form, with single voltage arguments in millivolts, and that return either a tau value in milliseconds or a steady-state value between 0 and 1, respectively.

13.13 Summary of CONC-PARTICLE-TYPE-DEF Format

- :NTH-ORDER Class -

```
(conc-particle-type-def
 '(kahpo-gen
   (class . :nth-order)
   (alpha . 2.0e+14)
   (beta . 0.01)
   (tau-0 . 100.0d0)
   (power . 4)
   (qten . 1.0)
   (reference-temp . 30.0)
   (shell . 2)
   (conc-int-type . ca-in-gen)))
```

13.14 Related Functions and Files

See also Section 17.

14 Markov Particles

Particle kinetics may be defined in terms of a Markov model, as described in Destexhe et al.(1994). The definition for a Markov state particle type is a bit more complicated than that for :HH or :HH-EXT particle types. The description follows the following format.

14.1 Defining Global Variables Used in the Markov Particle Type Definition

For this example, various components of the particle type definition will be assigned to global variables (e.g. `*4STATE-DEMO-STATES*`, etc.). On the other hand, note that all components can be incorporated directly in the `PARTICLE-TYPE-DEF` form, described later.

First, define two lists which hold all the state names and the open state names, respectively:

```
'(C1 C2 O I)
'(O)
```

State transition rates can be either dependent on the voltage controlling the particle, in which case the rates are precomputed and stored in look up arrays, and/or the rates can be arbitrary functions of the particles during the simulation. In the latter case, the functions are referenced by the particle type and called when the particle is evaluated.

To setup the transition rates, define one or two variables which reference the states and assigns transition functions dependent on voltage or on the particle in general to the available transitions.

This is a list of lists, one for each specific state transition:

```
'((from to rate-def particle-arg-p) ...)
```

`FROM` and `TO` reference specific states of the particle. They can be either symbols corresponding to the symbols used in the states list, or indices referring to the original ordering of the states list.

`RATE-DEF` may be either a function name (for functions that will be called during the particle evaluations), a function expression, or a number ($1/ms$, for voltage-independent transitions).

If a voltage-dependent function is to be used, then the function expression should be a form referencing a function that has a single required argument (corresponding to voltage in mV). As described below, using a function expression is advantageous if you want to easily edit function parameters and store the changes in an .elts file.

If a function name is used, this function is called during the particle evalation with the particle as the single argument. The result of the function is passed on by using the macro `RETURN-MARKOV-RATE`:

return-markov-rate *val* [*Macro*]

Thus, the function must be written with the `RETURN-MARKOV-RATE` macro, whose argument is an expression that returns a double-float rate value. For example:

```
(defun kco-markov-ca-activation-backward (prt)
  (return-markov-rate
   (let ((type (conc-particle-type (particle-conc-particle prt))))
     (d-flt ; So that a double float value is returned.
      (* (conc-particle-type-beta type)
         (conc-particle-type-q10-rate-factor type))))))
```

The `PARTICLE-ARG-P` entry flags whether or not the expression is dependent on voltage alone (`PARTICLE-ARG-P` = NIL, or is absent), in which case a look up array is precomputed, or whether the expression is a function with the particle as its argument (`PARTICLE-ARG-P` = T). The various optimization flags are not required, but can make a very large difference in efficiency.

In all cases, the calculated rate should be in units of $1/ms$.

14.2 Editing Markov Transition Function Parameters

Functions that define voltage-dependent transition rates may also have keyword arguments, with values specified in the expression. These expressions may be edited (in particular the values supplied to the keywords) from the menus later, while editing parameters of the particle type. Thus, it is advantageous to use functions with keyword parameters if you want to (easily) dynamically modify the transition rates during a Surf-Hippo session.

Note that the a function expression should have a dummy required argument - in the example below we use "VOLTAGE" just to emphasize that the expression will be evaluated with voltage arguments. The function in a function expression also must take a single-float voltage argument (mV), and return a value in units of $1/ms$.

Following from the example above, a complete list of the transitions would be:

```
'((C2 O (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -51.0 :K 1.0 :TAU-MIN 0.3333))
  (O C2 (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -57.0 :K -2.0 :TAU-MIN 0.3333))
  (O I 3)
  (O C1 (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -51.0 :K -2.0 :TAU-MIN 0.3333))
  (C1 O (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -42.0 :K 1.0 :TAU-MIN 0.3333))
  (I C1 (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -53.0 :K -1.0 :TAU-MAX 100.0 :TAU-MIN 1.0))
  (C1 C2 (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -60.0 :K -1.0 :TAU-MAX 100.0 :TAU-MIN 1.0)))
```

In this example of a particle with four states, seven of the possible twelve state transitions are allowed. six of these transitions are defined in terms of a voltage dependent function, with keyword parameters adjusted for each specific transition (see documentation for the `SQUEEZED-EXPONENTIAL` function). The seventh transition (state O to state I) is voltage independent and set to a constant value of 3.0ms^{-1}.

14.3 The Format of the `PARTICLE-TYPE-DEF` Form

Finally, the `PARTICLE-TYPE-DEF` form for a Markov state particle type references the lists describe above:

```
(particle-type-def
 `(NA-X-HPC
   (class . :MARKOV)
   (STATES . (C1 C2 O I))
   (OPEN-STATES . (O))
   (STATE-TRANSITIONS .
    ((C2 O (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -51.0 :K 1.0 :TAU-MIN 0.3333))
    (O C2 (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -57.0 :K -2.0 :TAU-MIN 0.3333))
    (O I 3)
    (O C1 (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -51.0 :K -2.0 :TAU-MIN 0.3333))
    (C1 O (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -42.0 :K 1.0 :TAU-MIN 0.3333))
    (I C1 (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -53.0 :K -1.0 :TAU-MAX 100.0 :TAU-MIN 1.0))
    (C1 C2 (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -60.0 :K -1.0 :TAU-MAX 100.0 :TAU-MIN 1.0))))
   (reference-temp . 27.0)
   (qten . 1.0)))
```

Note that we could have used the backquote syntax in order to represent the various lists as global variables; in some cases the resulting code will be clearer. The explicit format shown here is used, for example, if a `PARTICLE-TYPE-DEF` form is saved to file by the program, as below.

14.4 Saving Markov Particle Type Parameters

As described in Section 11, loadable Type-Def forms with the currently loaded element type parameters may be written to file using the `DUMP-ELEMENTS-FILE` function. For Markov particle types, this form will only be loadable if the state transistion rates are defined in terms of function expressions, as described above. In this case, the saved Type-Def form will include the original function expressions, along with the current

values for the keyword parameters, if they exist. On the other hand, if a state transitsion rate is defined in terms of a lambda expression, then the stored function will refer only to a lexical (local) definition, which is only valid during the Lisp session in which it was created.

14.5 Specifying Explicit Initial Conditions for Markov Particles

Normally, the initial conditions for a Markov particle are determined by the initial voltage, concentrations, etc. that determine the dynamics of the particle. On the other hand, as with HH-type particles, an explicit value for the initial state may be defined by including a list of numbers, summing to 1 (naturally), in the particle type or particle parameters, e.g.:

```
(element-parameter SOME-PARTICLE-TYPE 'initial-state '(0.1 0.2 0 0 0.7))
```

If a particle has a parameter value defined for `'INITIAL-STATE`, this list will supercede that defined, if any, for the particle type. The values in the list are assigned the the states coresponding to the definition in the particle type's parameters, e.g.:

```
* (element-parameter SOME-PARTICLE-TYPE 'states)
  (C1 C2 O I1 I2)
```

14.6 Linear Markov models based on the Hodgkin-Huxley model

The standard interpretation of the Hodgkin-Huxley gating model is that all particles associated with a channel must be in the open state for channel conduction. A generalization of this theme is to assume that for a channel with N Hodgkin-Huxley gating particles of the same type, the channel conducts when any M ($\leq N$) of the particles are in the open state (e.g. the model of retinal I_H of Barnes and Hille, 1989). We call this a linear Markov model since the Markov state diagram has no loops. Since all the particles are the same type, the computation of the channel gating is much more efficient than that for the general Markov case. Specifically, given N particles of type x, each with the same open probability p (since they are assumed to be identical and independent), then the contribution $h_x(V,t,...)$ of the ensemble of particles to the channel gating term $h(V,t,...)$ is given by a summation of binomial terms:

$$h_x(V,t,...) = \sum_{i=M}^{N} \left(\begin{array}{c} N \\ i \end{array} \right) p^i (1-p)^{(N-i)}$$

A linear Markov particle type is defined as a two-state particle (e.g. :HH, :HH-EXT) or concentration particle type, with an additional entry in the `PARTICLE-TYPE-DEF` or `CONC-PARTICLE-TYPE-DEF` form, for example (in `src/parameters/barnes-hille-cone-channels.lisp`):

```
(particle-type-def
 `(h-x-barnes-hille-89
   (class . :HH-EXT)
   (linear-markov . (4 2)) ; N = 4, M = 2
   ...
)
```

14.7 Sub-Particles

A Markov particle state transition can be concentration dependent by specifying a concentration "sub" particle. This is done, for example, for the Ca^{2+}-dependent gating of the KCT channel in the Working model:

```
(particle-type-def
 `(KCTX-HPC
   (class . :MARKOV)
   (STATES . (C O I))
```

```
(OPEN-STATES . (O))
(STATE-TRANSITIONS .
 ((C O KCTX-HPC-CA-ACTIVATION-FORWARD T)
  (O C KCTX-HPC-CA-ACTIVATION-BACKWARD T)
  (I C (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -120.0 :K -10.0 :TAU-MIN 10.0))
  (O I (SQUEEZED-EXPONENTIAL VOLTAGE :V-HALF -64.0 :K -3.5 :TAU-MIN 0.1))))
(reference-temp . 27.0)
(qten . 1.0)
(concentration-particle-type . KCTX-HPC-CA)))
```

By specifying a `CONCENTRATION-PARTICLE-TYPE` entry in the `PARTICLE-TYPE-DEF`, whenever a Markov particle of this type is created, an additional conc-dependent particle is also created. Importantly, the sub-particle is not evaluated as if it were a normal distinct gating particle, but rather the specified rate functions for the parent Markov particle may access the sub-particle as needed, for example:

```
(defun KCTX-HPC-ca-activation-backward (prt)
  (return-markov-rate (nthorder-conc-particle-backward-rate (particle-conc-particle prt))))
```

As explained above, the PRT argument in the state transition rate function `KCTX-HPC-CA-ACTIVATION-BACKWARD` is the Markov particle, and the sub-particle which imparts concentration dependence is referenced via that particle's :CONC-PARTICLE slot, thus `PARTICLE-CONC-PARTICLE`. Since the sub particle is not evaluated *per se*, normally there is no data associated with it (see Section 20.6).

15 Synapses

This section describes the synapse model.

15.1 Controlling Synapse Activation - Classes of Synapse Types

The fundamental difference between different classes of synapse types is how class is controlled. The options, specified by the synapse type slot `:CONTROL`, are as follows:

- `:LIGHT` - The conductance waveform for each synapse is precalculated before the simulation: the synapse spatial receptive field is integrated over the simulation time with a (2D) spatio-temporal pattern of "light" projected onto a specified anatomical plane, the result of which is convolved with the synapse type's impulse response and then passed through a non-linearity explicitly associated with the impulse response. If specified, there can be an additional (linear) convolution step.
- `:CHANNEL` - The synaptic conductance, calculated during the simulation, is modelled as a voltage and/or concentration dependent channel, where the reference node for the voltage/concentrations controlling the channel is specified by the `:PRE-SYNAPTIC-NODE` slot of the synapse.
- `:VOLTAGE` - The synaptic conductance, calculated during the simulation, is derived from summing delayed copies of a reference conductance waveform (defined for the synapse type). Additional copies are included in the summation (triggered) when the voltage at the `:PRE-SYNAPTIC-NODE` slot of the synapse satisfies the conditions specified for the synapse type. The pre-synaptic node may refer to any segment, soma, or axon.
- `:EVENT` - The synaptic conductance, calculated during the simulation, is derived from summing delayed copies of a reference conductance waveform (defined for the synapse type). Additional copies are included in the summation (triggered) at time points stored in the `:EVENT-TIMES` slot of the synapse.
- `:TONIC` - The synaptic conductance is constant for the entire simulation.

The control method for a given synaptic type is included in the list that defines the type parameters, e.g.:

```
(control . :voltage)
```

There is a `:EVENT-TIMES` slot that is associated with each synapse. For event synapses and voltage, the synapse type waveform is applied with a `:DELAY` added to the `:EVENT-TIMES`. For light dependent synapses, the precalulated conductance waveform is time shifted by `:DELAY`.

15.2 Evaluation of Different Types of Synapse Control

The sequence of events for the evaluation of each type of synapse control are as follows.

15.2.1 Evaluation of Light-Controlled Synapses

The conductance waveform, except for possible static conductance non-linearity, is computed before simulation:

1. Integrate spatial receptive field of synapse with the pattern of light over the duration of the simulation.
2. Convolve result of spatial integration with linear impulse response of synapse type.
3. Pass result of convolution through non-linearity associated with impulse response (default is a threshold at 0, i.e. half-wave rectification).
4. If specified, convolve waveform again with a 2nd linear impulse response associated with synapse type.
5. Store resulting waveform for use during simulation.

If the global variable `*REUSE-SYNAPSE-WAVEFORMS*` is T and the simulation duration is unchanged and there was a prior simulation that computed the light inputs, then the results of the last simulation's convolution(s) are reused.

During the simulation, this waveform is used (shifted by the delay for each synapse) for the conductance reference at each time step, after passing through (additional) static non-linearity (if definded for this synapse type). The result is then multiplied by the `:GBAR` of the synapse, and this is used as current value of synaptic conductance.

15.2.2 Evaluation of Channel-Controlled Synapses

During the simulation, the associated channel conductance is evaluated like a regular channel, with the controlling voltage and/or concentrations taken from the pre-synaptic element. The result is passed through a static non-linearity, if it exists for this synapse type, and then multiplied by the `:GBAR` of the synapse. This value is used as current value of synaptic conductance.

15.2.3 Evaluation of Voltage-Controlled Synapses

At each time step in the simulation:

1. Look at pre-synaptic element voltage (segment, soma, or axon).
2. Trigger conditions satisfied?
3. If so, add new trigger time (plus delay for each synapse) to list of trigger times.
4. Loop over all trigger times, summing appropriately shifted values of reference conductance waveform.
5. Pass result of summation through static non-linearity, if it exists for this synapse type.
6. Result is multiplied by the `:GBAR` of the synapse and this is used as current value of synaptic conductance.

15.2.4 Evaluation of Event-Controlled Synapses

At each time step in the simulation:

1. If time corresponds to one of the delay times for a given synapse, add new trigger time (plus delay for each synapse)to list of trigger times.
2. Loop over all trigger times, summing appropriately shifted values of conductance template.
3. Pass result of summation through static non-linearity, if it exists for this synapse type.
4. Result is multiplied by the `:GBAR` of the synapse and this is used as current value of synaptic conductance.

15.2.5 Evaluation of Tonic-Controlled Synapses

The synapse conductance is constant, taken from the `:GBAR` of the synapse.

15.3 Creating Synapses and Synapse Types

The basic function to create a synapse is:

create-synapse *post−synaptic−element type* &optional *pre−synaptic−element* [*Function*]
(add−pre−synaptic−element−name−to−name t) name

Returns a synapse of TYPE, installed on the cell element associated with POST-SYNAPTIC-ELEMENT. The TYPE argument references a type description defined with the macro `SYNAPSE-TYPE-DEF`. Synapse types that are controlled by the voltage of another node must include a PRE-SYNAPTIC-ELEMENT *elementornameassociatedwithasoma*, *segment*, *oraxon*. If the POST-SYNAPTIC-ELEMENT (can be NIL) already has a synapse of the same type, and the PRE-SYNAPTIC-ELEMENT is either different or not required for TYPE, then an alternate name will be created from the addition of a number at the end of the standard synapse name. If the global variable `*PROMPT-FOR-ALTERNATE-ELEMENT-NAMES*` is T, then the user is prompted before the additional synapse is created, otherwise, the synapse is created. The standard synapse name is either an integer, if `*USE-SIMPLE-NAMES*` is T (generated by GET-SYNAPSE-SIMPLE-NAME; see also Section 5.6), or given by NAME, if non-NIL, or composed from the TYPE and the post-synaptic element, including also the name of the PRE-SYNAPTIC-ELEMENT if there is one and if ADD-PRE-SYNAPTIC-ELEMENT-NAME-TO-NAME is T.

15.4 Synapse Presynaptic Cell Elements

Synapse types with `:CONTROL :CHANNEL` or `:VOLTAGE` require a presynaptic element for their control, which in turn can be either an axon, soma, or segment, as stated above. Thus, these types of synapses may only be defined *after* their presynaptic elements are defined. For example, this would mean that in a circuit with multiple interconnected cells, the cells would have to be defined first (with the appropriate functions that generate cell segments and somas, e.g. `CREATE-TREE`, etc.), and the synapses added afterwards.

Specifically, the PRE-SYNAPTIC-ELEMENT optional argument to `CREATE-SYNAPSE` must be supplied for `:CHANNEL` or `:VOLTAGE`-controlled synapse types - if you try to create a synapse whose type controlled by `:VOLTAGE` or `:CHANNEL`, and a pre-synaptic element is not specified, or does not exist, then an error is flagged, e.g.:

```
Warning:
  CREATE-SYNAPSE: Synapse Syn-Hippo-soma-FAST-EX must have complete presynaptic info

  ** Restarting from Top Level **
```

See the `TWO-HIPPOS` and `THREE-HIPPOS` examples in src/hippocampus/hippos.lisp.

It may be convenient to incorporate the pre-synaptic element creation within the call to `CREATE-SYNAPSE`. For example:

```
(create-synapse "CA1-12-ap18" 'fast-ex (create-axon "CA1-203-soma" 'SIMPLE-FAST))
```

In this example, a `FAST-EX` type synapse is added to a cell element whose name is "CA1-12-ap18" (assume that this type of synapse requires a presynaptic element). The embedded call to `CREATE-AXON` returns a `SIMPLE-FAST` type axon that originates in the cell element whose name is "CA1-203-soma".

When a pre-synaptic element is an axon, the `CREATE-SYNAPSE` function automatically assigns the created synapse to the axon so that the axon may be drawn properly from origin to its post-synaptic destination.

Event synapses may have a pre-synaptic element defined for them, which may in turn be incorporated in their name. However, in these cases there is no functional significance for the pre-synaptic element.

For accessing the pre- or post-synaptic elements of a created synapse, you would use `PRE-SYNAPTIC-ELEMENT` or `ELEMENT-CELL-ELEMENT`. For accessing the synapses who are driven by a given cell element use `DRIVEN-SYNAPSES`; for accessing the associated post-synaptic cell elements use `SYNAPTIC-TARGETS`. For accessing the synapses who impinge on a given cell element use `IMPINGING-SYNAPSES`.

15.5 Synapse Maximum Conductance

See Section 12.

15.6 Static Voltage Dependence of Synaptic Conductances

The conductance of a synapse can also be dependent on the current voltage of the post-synaptic node, as described in Section 12. Note that this non-linear step can be used for any type of synapse, and is conceptually but not mathematically distinct from the non-linearity possible after the (first) linear convolution step for light controlled synapses.

15.7 Inheritance Of Synapse Types

Synapse types may defined in reference to another by including a `PARENT-TYPE` entry in the type parameter list. For example:

```
(synapse-type-def
 `(nmda-alpha
   (parent-type . fast-ex-alpha)
   (static-voltage-dependence-function . (SIGMOID-ARRAY -50.0 0.5
                                          ,*PARTICLE-LOOK-UP-TABLE-MIN-VOLTAGE*
                                          ,*PARTICLE-LOOK-UP-TABLE-MAX-VOLTAGE*
                                          ,*PARTICLE-LOOK-UP-TABLE-PRECISION*))))
```

Here, the synapse type NMDA is identical to the `FAST-EX` type (which must have its own entry stored using `SYNAPSE-TYPE-DEF`), with the additional parameter `STATIC-VOLTAGE-DEPENDENCE-FUNCTION` (see Section 12.7). This process can have an arbitrary number of levels.

15.8 Derivation of Synapse Conductance Waveforms for Non-V-Dependent-Channel Type Synapses - Linear and Non-Linear Stages

As mentioned above, synapses may be driven by some event or controlling variable, that in turn either initiates the application of a canned waveform or a waveform created by the convolution of the synapse type's impulse response with the driving event. For light-driven synapses, the convolution(s) of the light stimulus and the impulse response(s) is computed before the simulation integration. The parameters of the impulse response are given as the `IMPULSE-FUNCTION-ARGS` entry in the appropriate type parameter list stored with `SYNAPSE-TYPE-DEF` (see `src/parameters/synapse-params.lisp`). For example, one synapse type parameter list stored with `SYNAPSE-TYPE-DEF` may be:

```
'(LIGHT-EX-1
  (GBAR-DENSITY . 10.0)
  (E-REV . 0.0)
  (CONTROL . :LIGHT)
  (IMPULSE-FUNCTION-ARGS . (DOUBLE-ALPHA 10 60 1)))
```

Here, the list `(DOUBLE-ALPHA 10 60 1)` is used to obtain the impulse response for synapse type `LIGHT-EX-1`. The function to be evaluated is `DOUBLE-ALPHA`, applied to the arguments 10, 60, and 1. Any function which returns a 1-d sequence (list or array) of numbers may be used here, with the appropriate arguments. The parameters of the impulse function may be edited with the menus [Main Simulation Menu: ¡Modify cell types, sources, channels or synapses¿ ¡Modify the synapses¿¡Modify parameters of synapse types¿...¡Edit impulse response or event waveform¿]. The waveforms/impulse responses may also be plotted here, or from the ¡Modify the synapses¿ option.

If the impulse/waveform function is compiled with a documentation string (see the `DEFUN` form for `DOUBLE-ALPHA` as an example), then this string will be displayed in the menus for editing the function arguments.

The conductance waveforms for light-driven synapse types are derived from the convolution of the light stimulus (integrated over the spatial RF of the synapse) and the impulse response of the type (see below).

The result of this convolution may be passed through a nonlinearity, given by the

```
(IMPULSE-NONLINEARITY . :RECTIFICATION)
```

or

```
(IMPULSE-NONLINEARITY . :THRESHOLD)
(IMPULSE-NONLINEARITY-PARAMETERS . 0.2)
```

entries in the SYNAPSE-TYPE-DEF parameter list argument. If not specified (e.g. not explicitly NIL), then the nonlinearity is a threshold at 0. The possible functions and their parameters may be found in definition of CONVOLVE-INPUT-ARRAY.

If there is a:

```
(LINEAR-IMPULSE-FUNCTION . (ALPHA 20))
```

entry in the SYNAPSE-TYPE-DEF parameter list argument, then the result of the first (possibly non-linear) convolution is then convolved again with a function specified by the LINEAR-IMPULSE-FUNCTION entry. The output of this second convolution is *not* passed through a subsequent nonlinearity at this point (but may be during the simulation if a "static non-linearity" as described previously is specified).

The concatenation of the linear convolution with a static nonlinearity is useful for smoothing the rectification (half-wave for ON or OFF synapse, full-wave for ON-OFF synapses) typically observed for light-driven synapses (see the functions NONLINEARITY and CONVOLVE-INPUT-ARRAY).

For synapse types that reference a canonical response waveform, the time base of the waveform is given in the WAVEFORM-TIME-INTERVAL entry in appropriate type parameter list stored with SYNAPSE-TYPE-DEF. For example:

```
'(AUTO-INHIBITORY
  (GBAR-DENSITY . 1000.0)
  (E-REV . -70.0)
  (CONTROL . :EVENT)
  (WAVEFORM-SPEC . (ALPHA-ARRAY 10 1 T 0.2))
  (WAVEFORM-TIME-INTERVAL . 0.2))
```

Depending on the function referenced in WAVEFORM-SPEC, the WAVEFORM-TIME-INTERVAL (always in milliseconds) must be consistent with the arguments to the waveform generating function. In the example above, the function ALPHA-ARRAY has an (optional) fourth argument which specifies the step size, relative to the first tau argument, of the returned array. As long as the step size and the WAVEFORM-TIME-INTERVAL are the same, then the tau specified will be evaluated during the simulation as being in units of milliseconds.

As above for the impulse function waveforms, the entry in the type parameter list for WAVEFORM-SPEC must be a function which returns a list or array of numbers. Note that in both cases, the sequences are converted to arrays when the synapse type is created.

15.9 Relationship between Light Stimulus and Light Synapses

15.9.1 Spatial RF Location: Relationship between Light Stimulus and Synapses

For light-driven synapses, the projection of the light (defined according to an XY coordinate system) is mapped onto a 2D projection of the cells, as described in Section 25.2. Thus, the light XY is mapped to the cell's XY (eg for retina or flattened cortex) or to the cell's XZ (eg for cortical neurons in radial mount). This mapping is determined by the global variable *LIGHT-STIMULUS-PLANE*, which would have the value :XY in the former case, and :XZ in the latter.

For mapping the light XY plane to the cell XZ plane, the projection of the anatomical view is taken "from the top". This means that the [X,Y] for the light goes to [X,-Z] for the cell.

The location of a synapse's receptive field in the plane of the stimulus can be either the physical location of the synapse (default), or an arbitrary location set earlier. The former case is analogous to a retinotopic mapping, that is the light stimulus is mapped directly onto the geometry of the tissue. Note that for a synapse on a segment, the location is given by the DISTAL node of the segment.

If the receptive field center (x or y) of a given synapse is not to be taken from the physical location of the synapse, then the association list in the :PARAMETERS slot of the synapse structure must include the entry

```
(LIGHT-INPUT-X . 30.5)
```

for defining the X location of the RF center, and/or

```
(LIGHT-INPUT-Y . 230.0)
```

for defining the Y location of the RF center. Both of these parameters are taken in the light stimulus plane.

If these valuess are set for a given synapse, the function GENERATE-LIGHT-INPUT-ARRAY uses the values for determining the RF location; otherwise the physical location of the synapse is used. For example (from GENERATE-LIGHT-INPUT-ARRAY, where SYN-PARAMETERS is taken from the :PARAMETERS slot of a synapse):

```
(or (element-parameter syn 'LIGHT-INPUT-X)
    (+ (first (node-absolute-location (synapse-node syn)))
       (* (or (element-parameter (synapse-type syn) 'LIGHT-OFFSET-DISTANCE) 0.0)
       (cos (or (element-parameter (synapse-type syn) 'LIGHT-OFFSET-ANGLE) 0.0))))))
  --> syn-rf-center-x
```

The local variables LIGHT-INPUT-OFFSET-DISTANCE and LIGHT-INPUT-OFFSET-ANGLE referenced above are taken from the appropriate synapse type :PARAMETERS, if defined (otherwise they are taken as 0).

For explicit mapping of a synapse's RF center, use the functions

set-synapse-rf-center-x *syn rf−center−x* [*Function*]

set-synapse-rf-center-y *syn rf−center−y* [*Function*]

where SYNAPSE is either a synapse structure or a synapse name. For example, the function:

map-light-inputs &key *(x−max 500) (x−min −500) (y−max 500) (y−min −500) type−name (use−menus t)* [*Function*]

uses SET-SYNAPSE-RF-CENTER-X and SET-SYNAPSE-RF-CENTER-Y to randomly map the receptive field centers of all the synapses of a given type (or types, if chosen with the menus) over a rectangular area defined by the X-MAX, X-MIN, Y-MAX, and Y-MIN arguments.

See Section 25 for notes on visualization of the receptive field centers.

15.9.2 Spatial RF Function

The IMPULSE and SPATIAL-RF array entries are computed in the setup sequence, according to the appropriate parameters. The SPATIAL-RF-FUNCTION-ARGS entry, for light-dependent synapses, specifies a function which returns a 2-D array that defines the spatial receptive field. If this entry does not exist for a light-dependent synapse, then the spatial receptive field is a 2D impulse. For the args to GAUSSIAN-RF, see the funspec. If the ADJUST-TO-RF-AREA entry is NIL or missing, then the spatial integration is done without compensating for the support of the of the SPATIAL-RF 2-D array. Otherwise, the integration is multiplied by the area of the support. The first case is used, for example, when the 2-D array sums up to 1.0 or 0.0 (e.g. if you have a DOG-RF, and the keyword :TOTAL-VOL 0 is included so that the array has zero overall weight).

Spatial integration of over each synapse RF is done for the entire duration of the simulation. For times prior to and after the stimulus duration, then the integration is over a constant light level set by the global variable *LIGHT-BACKGROUND*. Thus, L(t), the light input at time t for a synapse with a receptive field RF(x,y), is given by integrating over the support of RF():

```
RF(x,y) * [*LIGHT-BACKGROUND*  +  *LIGHT-STIMULUS-STRENGTH* x Stimulus(x,y,t)]
```

where Stimulus(x,y,t) is non-zero for t between *LIGHT-STIMULUS-START-TIME* and before *LIGHT-STIMULUS-STOP-TIME*. In general the function Stimulus(x,y,t) ranges between 0.0 and 1.0. For example, when *LIGHT-STIMULUS* is set to :ON-SPOT, then

```
Stimulus(x,y,t) = 0.0 before and after the stimulus "on" time, or outside the spot region
                = 1.0 during the stimulus "on" time and inside the spot region
```

Conversely, when *LIGHT-STIMULUS* is set to :OFF-SPOT, then

```
Stimulus(x,y,t) = 0.0 before and after the stimulus "on" time, or outside the spot region
                = -1.0 during the stimulus "on" time and inside the spot region
```

The convention for OFF type stimuli (0.0 when not present, -1.0 when present) is the same for :OFF-MOVING-BAR, :OFF-MOVING-SPOT, and :OFF-BAR. In these cases, the values of *LIGHT-BACKGROUND* and *LIGHT-STIMULUS-STRENGTH* have to be chosen to get the final desired light waveform.

As discussed above, the waveform L(t) is then convolved with the synapse type's impulse response, and the result passed through a non-linearity explicitly associated with the impulse response. Again, if specified, there can be an additional (linear) convolution step. Finally, the resulting waveform is multiplyed by the synapse :GBAR value during the simulation to obtain the final value of the synaptic conductance.

15.9.3 Spatial RF Location: Relationship between Moving Light Stimulus and Synapses

Light stimulus motion is defined with respect to stimulus frame (referred to above by *LIGHT-STIMULUS-PLANE*). The origin of the stimulus frame is shifted with respect to the anatomical 2d coordinate frame by *LIGHT-START-POSITION-X* and *LIGHT-START-POSITION-Y* (microns). Further, the stimulus frame is rotated around its origin counter clockwise by *LIGHT-THETA* (in radians). Thus, to translate from anatomical (2D) coordinates (XY) and stimulus frame coordinates (X'Y'):

```
X = (X' * cos[*LIGHT-THETA*] - Y' * sin[*LIGHT-THETA*]) - *LIGHT-START-POSITION-X*
Y = (X' * sin[*LIGHT-THETA*] + Y' * cos[*LIGHT-THETA*]) - *LIGHT-START-POSITION-Y*

X' = (X - *LIGHT-START-POSITION-X*) * cos[*LIGHT-THETA*]
     + (Y' - *LIGHT-START-POSITION-Y*) * sin[*LIGHT-THETA*]
Y' = (*LIGHT-START-POSITION-X* - X) * sin[*LIGHT-THETA*]
     + (Y' - *LIGHT-START-POSITION-Y*) * cos[*LIGHT-THETA*]
```

The straight line trajectory for moving bars and spots is along the positive(negative) Y' axis when *LIGHT-DIRECTION* is T(NIL), and the trajectory begins at (X'Y')=(0,0). Bar stimuli are defined such that they are oriented with their length parallel to the X' axis and the width parallel to the Y' axis.

15.9.4 Spatial RF Location: Relationship between Apparent Motion Light Stimulus and Synapses

As above with regular motion, apparent motion stimulus parameters are defined with respect to stimulus frame (referred to above by *LIGHT-STIMULUS-PLANE*) - the origin of the stimulus frame is shifted with respect to the anatomical 2d coordinate frame by *LIGHT-START-POSITION-X* and *LIGHT-START-POSITION-Y* (microns) and the stimulus frame is then rotated around its origin counter clockwise by *LIGHT-THETA* (radians). The global variables *BAR-A-POSITION-X*, *BAR-A-POSITION-Y*, *BAR-B-POSITION-X*, and *BAR-B-POSITION-Y* determine the final positions of the A and B bars in the shifted and rotated stimulus frame. As with single bar stimuli, the length and width of the bars are taken as the dimensions along the stimulus frame X' and Y' axes, respectively.

15.9.5 Stimulus Visualization

In all cases, it is a good idea to check stimulus parameters by seeing how they are rendered with the histology graphics.

15.9.6 Fast Light-Driven Synapse Convolutions

The first step for the pre-simulation calculation of light-driven synaptic inputs is the spatio-temporal integration of the stimulus over the duration of the simulation with the spatial receptive field of each synapse. This step yields a one-dimensional light input waveform that will then be passed through the linear and non-linear stages of the synapse type impulse response as described above.

If the global variable *COMPUTE-ALL-LIGHT-INPUTS* is T, then the processing of the light input waveform is done explicitly for every synapse. However, when *COMPUTE-ALL-LIGHT-INPUTS* is NIL (default), when the first synapse is processed its light input waveform is stored with the synapse type information. When the next synapse of the same type is processed, the light input waveform for this synapse is compared with the previously stored light input waveform. If the two waveforms are identical, given a scale factor and a shift for synapse types with no non-linear stage, and given a shift for those types with a non-linear stage, then the synapse input is not processed further, but rather a reference to the earlier computed light input waveform is stored. Otherwise, the new light input waveform is stored with the type as was done for the first synapse, and the final conductance waveform is computed. These steps of partial evaluation of the synapse input and subsequent comparison with earlier computed light input waveforms is repeated for the remaining synapses. Thus during the simulation run, the conductance input for each synapse is taken from a waveform that was either computed especially for that synapse, or from a waveform that is from a synapse whose light input waveform has the same time course (and magnitude for nonlinear synapse types), with the final conductance waveform shifted and scaled appropriately.

The motivation for this algorithm is to avoid unnecessary convolution of the light input waveform with synapse impulse responses prior to simulations.

Associated with this method is the global variable *SYNAPSE-NAMES-TO-DO-FIRST*. This variable is a list of either synapse names or node names, and the conductance waveforms for the synapses that correspond to these references are computed first. In this way, you can choose synapses who will have the longest responses, thereby reducing the number of reference waveforms that must be stored in order to account for all synapse responses. This variable is also set via the histology element menu.

If the light stimulus is a flashing spot which covers the receptive fields of all the synapses, then a similar short cut may be used in convolving the synaptic waveforms prior to the simulation. This short cut exploits the fact that for this stimulus the synaptic waveforms of synapses of the same type will be identical. This option may be enabled by choosing the "Do Fast Rf Spot convolution for full field spot" in the "Parameters For SPOT Stimulus" menu. *FAST-FULL-FIELD-SPOT* is the global variable which enables this feature.

15.9.7 Specifying Light Stimulus Parameters

Light stimulus parameters may be defined in files by setting the appropriate global variables. For example, a 50 by 500 μm bar moving from left to right at 2 μm/millisecond, starting at position X = -300μm, Y = 0μm at time = 0 would be set up by the following variable settings:

```
(setq
 *enable-light* t
 *light-stimulus* :ON-MOVING-BAR
 *bar-length* 500.0 *bar-width* 50.0    ; microns
 *light-speed* 2.0                      ; microns/millisecond
 *light-theta* (/ pi-single 2)          ; Use pi-single since we need a single float
 *light-direction* nil                  ; => Movement is opposite to *LIGHT-THETA*
 *light-stimulus-start-time* 0.0   ; Time to start bar moving, milliseconds
 *light-stimulus-stop-time* 100000.0    ; This is just larger than the simulation time
 *light-start-position-x* -300.0   ; Stimulus center at *motion-start-time* (microns)
 *light-start-position-y* 0.0)
```

The options for the *LIGHT-STIMULUS* include:

```
:APPARENT-MOTION :MOVING-SPOT
:ON-MOVING-BAR   :OFF-MOVING-BAR
:ON-SPOT         :OFF-SPOT
:ON-BAR          :OFF-BAR
:ANNULUS
```

Associated parameters include:

```
*enable-light*              T or NIL - when nil, light synapses are not evaluated
```

```
*light-stimulus-start-time* milliseconds
*light-stimulus-stop-time*

*light-stimulus-strength*   arbitrary units (default 1)
*light-background*          Light background level in arbitrary units (default 0)

*light-theta*               Orientation of stimulus (and its trajectory if
                            moving), with 0 being vertical up - in radians

*light-stimulus-plane*      :XY for retina, :XZ for radial mount cortical cells.
```

Center of stimulus (μm) at *MOTION-START-TIME* (also for static stimuli) -

```
*light-start-position-x* *light-start-position-y*
```

For motion stimuli:

```
*light-speed*         Microns per millisecond
*light-direction*     T (nil) => movement direction with (opposite) to *light-theta*
*motion-start-time*   Time to start moving, milliseconds
```

For bars (μm):

```
*bar-width* *bar-length*
```

For an aperture (when *USE-APERTURE* is T), in μm:

```
*aperture-radius* *aperture-center-x* *aperture-center-y*
```

For apparent motion stimulus (μm):

```
*bar-a-width* *bar-a-length*
*bar-a-position-x* *bar-a-position-y*
*bar-b-width* *bar-b-length*
*bar-b-position-x* *bar-b-position-y*
```

For apparent motion stimulus (milliseconds):

```
*bar-a-start-time* *bar-a-stop-time*
*bar-b-start-time* *bar-b-stop-time*
```

For apparent motion stimulus (arbitrary units, default 1):

```
*bar-a-intensity* *bar-b-intensity*
```

For spots (μm):

```
*spot-outside-diameter* *spot-inside-diameter*
```

15.10 Q_{10}'s For Synapses

The Q_{10} for synaptic conductances (not kinetics) is defined identically as that for channels (see Section 13).

15.11 Gap Junctions

(NOT ENABLED YET)

A gap junction between nodes i and j is modelled as a reciprocal pair of channels with an identical voltage and time independent conductance, one for each node. In addition, the reversal potential for the channel in node i is given by the voltage at node j, and vica-versa. This should work since the voltage used for the reversal potential will be out of step with the evaluation time. In other words, there will always be a non-zero delay for signal propagation through the gap junction.

15.12 Miscellaneous - Controlling Synapse Setups, etc.

For all synapse types, the boolean variable:

```
*ENABLE-SYNAPSES*
```

allows the evaluation of synapses during the simulation. Synapses of each control are setup prior to each simulation according to the following global variables:

```
*SETUP-VOLTAGE-SYNAPSES* *SETUP-LIGHT-SYNAPSES*
*SETUP-AUTO-SYNAPSES* *SETUP-TONIC-SYNAPSES*
```

The default for each of these is T. However, if there is no change in the synapses of a given control type (i.e. deletions, creationgs, blocking, timing, or stimuli) then the corresponding flag may be set NIL to speed things up. On the other hand, there is NO automatic checking to see that in fact there is no change with a given group of synapses; therefore it is the responsibility of the user to set these flags appropriately.

To plot the impulse responses of the synapses, use the function `PLOT-SYNAPSE-IMPULSE`, or use the menus (see above). To plot the light input to light controlled synapses, use:

plot-light-input &optional *syns* [*Function*]

where SYNS is a synapse name or pointer, or a list of names or pointers. When SYNS is not supplied, then a menu prompts for the plotted synapses.

15.13 Specifying Trigger Times (Events or Delays) for Event Synapses

The event times for event synapses (the `:EVENT-TIMES` slot, which is a list of single-float times in milliseconds) can be set directly using the function:

add-events *syns−or−types event−times* [*Function*]

where EVENT-TIMES (single number or a list of numbers) is added to all (event) synapses referenced by the atom or list SYNS-OR-TYPES. Delays may also be set with the menu called by:

edit-synapse-event-times *syn* [*Function*]

This function is accessible from the main menu sequence.

Other functions of interest are:

clear-events &optional *syns−or−types* [*Function*]

remove-events *syns−or−types event−times* [*Function*]

add-poisson-events *syns−or−types lambda−spec start stop* &key *(step 1.0) (time−offset 0.0) (lambda−coefficient 1.0) (min−interval−value 1.0) clear−events* [*Function*]

print-synapse-events &optional *type* [*Function*]

histogram-synapse-events &key *(bins 10) (min−time 0) type syn (max−time *user−stop−time*)* [*Function*]

plot-scatter-synapse-distances-event-times &key *(synapse−types (synapse−types))* *(white−is−maximum−p t)* *(x−are−fns t)* *(y−are−fns t)* *(width 400)* *(height 400)* *plot−type* *min−plot−time−given−by−events* *(plot−events−prior−to−0 t)* *title* *(3dplot−scale 30.0)* *(3dplot−aspect 0.2)* *(3dplot−time−bins (/ *user−stop−time* 100))* *(3dplot−distance−bins 20)* *(minimum−distance 0)* *maximum−distance* *(maximum−time *user−stop−time*)* *minimum−time* *(cell *cell*)* *distance−plot−increment* [*Function*]

When the global variable `*PRINT-SYNAPSE-TOTAL-EVENTS*` is T, then:

print-synapse-total-events [*Function*]

is called at the end of the simulation, printing out the total number of events of all the voltage and event controlled synapses.

When the global variable `*PRINT-SYNAPSE-EVENT-TIMES*` is T, then:

print-synapse-events &optional *type* [*Function*]

is called at the end of the simulation, printing out the individual events of all the voltage and event controlled synapses.

15.14 Depressing/Facilitating Synapses: Dynamic Synaptic Weights

Depressing synapses according to the model proposed by Markram and Tsodyks (1996) is enabled by including the following entries in a `SYNAPSE-TYPE-DEF` form:

```
(1st-order-depressing-dynamics . t)
(tau-recovery . 800) ;ms
(release-fraction . 0.2)
```

Likewise, the conductance waveform for the apriori events assigned to event synapses may be scaled per event by specifying numeric lists in either

```
(element-parameter synapse-type 'event-weights)
```

or

```
(element-parameter syn 'event-weights)
```

In the first case it is assumed that all synapses of a given type have the same number of events; in both cases the lists are matched one by one to the assigned synaptic events.

See

sort-scale-and-shift-event-times *syn* [*Function*]

generate-1st-order-depressing-weights-list *syn sorted−events* [*Function*]

15.15 Event Generators and Followers

Event generators reduce evaluations for axons and synapses [VOLTAGE, LIGHT and LIGHT-EVENT] whose control parameters are identical for a given simulation. Thus, for LIGHT and LIGHT-EVENT synapse types, computation of the light response (prior to the simulation), and for VOLTAGE synapse types, detection of synaptic events (during the simulation), is done only for those synapses of a given type which are event-generators. Likewise, for axons, detection of spikes (during the simulation), is done only for axons which are event-generators.

For VOLTAGE controlled synapse types, a synapse which is an event generator acts as an event detector for all other synapses of the same type that have the same pre-synaptic node - these synapses are the event followers. For LIGHT and LIGHT-EVENT controlled synapse types, the light response (for LIGHT and LIGHT-EVENT) and event generation (for LIGHT-EVENT) applied to an event generator is used by all other synapses of the same type that have the same receptive field characteristics, for example the XY coordinates of the RF center.

For axons, an event generator serves as an event detector for all axon of the same type and with the same proximal-node.

Thus, in all cases the event generator assigned to a synapse or axon must be of the same type.

When a synapse or axon is created, its `:EVENT-GENERATOR` is initialized to point to itself. Note that for synapses this is only relevant if the synapse type is LIGHT, LIGHT-EVENT, or VOLTAGE controlled.

The :EVENT-GENERATOR slot and the assignment of event followers is normally updated at the beginning of each simulation by the functions `SETUP-VOLTAGE-SYNAPSES` and `SETUP-LIGHT-SYNAPSES`, which are called depending on the values of `*SETUP-LIGHT-SYNAPSES*` and `*SETUP-VOLTAGE-SYNAPSES*`, respectively. These functions set up event generators and followers by appropriate calls of `SETUP-EVENT-GENERATORS-AND-FOLLOWERS-OF-TYPE` (see below).

Event generators may also be set by explicit (e.g. in a user script) invocation of either

user-setup-event-generators-and-followers *event−generator event−followers* [*Function*]

which allows an arbitrary assignment of generators and followers, or

setup-all-event-generators-and-followers [*Function*]

which calls `SETUP-EVENT-GENERATORS-AND-FOLLOWERS-OF-TYPE` for all axon and synapse types. Note that this function locally binds `*SETUP-EVENT-GENERATORS-AND-FOLLOWERS*` to T and `*USER-SPECIFIED-EVENT-ELEMENT-SETS*` to NIL.

The function

setup-event-generators-and-followers-of-type *event−elements* [*Function*]

chooses event generators out of the list of EVENT-ELEMENTS, all of which must be the same synapse or axon type, and assigns event followers for each event generator. This assignment is initially predicated on

```
(and *SETUP-EVENT-GENERATORS-AND-FOLLOWERS*
     (not *USER-SPECIFIED-EVENT-ELEMENT-SETS*)
     *ENABLE-EVENT-GENERATORS*)
```

Event generator based evaluations are enabled by

enable-event-generators *t* [*Variable*]

The variable

setup-event-generators-and-followers *t* [*Variable*]

enables the automatic assignment of event element sets at the beginning of every simulation, as long as `*USER-SPECIFIED-EVENT-ELEMENT-SETS*` is NIL. This variable may be set to NIL after a simulation for more efficiency in subsequent simulations, or may always be NIL as long as the function `SETUP-ALL-EVENT-GENERATORS-AND-FOLLOWERS` is explicitly called when the circuit is setup or changed.

When the variable

user-specified-event-element-sets *nil* [*Variable*]

is T, then the user has the responsibility to setup event generators and followers (e.g. with calls to `USER-SETUP-EVENT-GENERATORS-`
or `SETUP-ALL-EVENT-GENERATORS-AND-FOLLOWERS`) before a simulation.

Note: With arbitrary evaluation order of voltage synapses, during a given time step an event generator may be evaluated after one of its followers. This imposes a minimum intrinsic delay for that follower given by the duration of the next time step.

Future revisions will fix this if there are enough complaints.

16 Some Biophysical Details

This section describes some specific biophysical (and related) issues.

16.1 Temperature Dependence - Element Q_{10}s

See also Sections 13, 15 and 17.

Q_{10}s (and associated reference temperatures) may be specified for types of synapses, channels, particles, concentration particles, buffers, pumps and concentration integrators. The Q_{10} for a synapse or channel type defines the temperature dependence for associated elements' maximum conductances. The Q_{10} for a particle, concentration particle, buffer or pump type defines the temperature dependence for associated elements' kinetics. Finally, the Q_{10} for a concentration integrator type defines the temperature dependence for the effective diffusion coefficient (`:MULTI-SHELL` class), the time constant (`:FIRST-ORDER` class), or the `DCDT-FUNCTION` of `:GENERIC` class concentration integrators.

In all cases, Q_{10} adjustment is only made when the global variable `*IGNORE-Q10*` is NIL (default). The simulation temperature is set by the global variable `*TEMP-CELCIUS*` (in degrees Celcius, 27.0 default). If an element does not have a explicit Q10 factor, then the default is 1.0 (i.e. no temperature dependence).

Q_{10} coefficients are calculated by the particular Q_{10} constant raised to $(T - T_{ref})/10$, where T is the temperature of the simulation, and T_{ref} is the reference temperature of the measured Q_{10}.

It is important to remember the provision above for conductance temperature dependence, as well as the more commonly considered temperature dependence of particle gating kinetics.

Relevant functions include:

q10-tau-factor *reference−temp temp q10* &optional *(ignore−q10 *ignore−q10*)* [*Function*]

q10-rate-factor *reference−temp temp q10* &optional *(ignore−q10 *ignore−q10*)* [*Function*]

where the former calculates the Q_{10} factor for time constants (as temperature goes up, τ goes down), and the latter calculates the Q_{10} factor for rate constants (as temperature goes up, so does rate).

16.2 Changing Temperature in Scripts

The global variable `*TEMP-CELCIUS*` may be changed by the user. If the temperature is changed between simulations then the function `UPDATE-TEMPERATURE-DEPENDENT-PARAMETERS` is called automatically at the beginning of the simulation.

16.3 Somatic I-V Characteristic

The functions:

plot-iv &optional *cells* [*Function*]

plot-ivs &optional *cells (use−menu t) use−one−window* [*Function*]

plot simple steady-state somatic current-voltage (I-V) relationships for the loaded cells. If the optional arguments are not supplied then `PLOT-IV` plots the characteristic for the last created cell, and `PLOT-IVS` presents a menu to choose a cell or cells. I-V relations are calculated using the following assumptions:

1. Passive dendritic tree (any channels in the tree are disabled).
2. Steady-state calcium concentrations, assuming the somatic potential is at resting potential.
3. Measurement made under ideal (somatic) voltage clamp.

These functions can be invoked from the menus. If there are any non-somatic non-linearities, more realistic I-V curves require an explicit set of (somatic) voltage clamp simulations, and collection of the corresponding clamp voltages and currents.

16.4 Spines

The function `GROW-SPINES` adjusts membrane resistivities and capacitances to fold-in "virtual" spines, under the assumption that the voltage drop or other compartmentalization of the spine neck is negligible:

grow-spines *neck−length neck−diameter head−diameter density* &key *cell−type segments (update−linear−parameters t)* [*Function*]

The spine model here is of a cylindrical neck capped by a spherical head. Spine area is given by the area of the neck (not including ends) plus the area of the head minus the area of the neck end (to partially compensate for the junction between the head and neck).

If SEGMENTS is a list of segments, then a `'MEMBRANE-AREA-COEFFICIENT` entry is added to the `:PARAMETERS` of each segment, otherwise if CELL-TYPE is supplied, the `:CM-DENDRITE` and `:RM` of the referred cell type are adjusted, otherwise (if both keyword args are NIL) the `:CM-DENDRITE` and `:RM` of all the cell types are adjusted. NECK-LENGTH, NECK-DIAMETER, HEAD-DIAMETER are in microns. DENSITY is number of spines per square micron of dendrite (non-spine) membrane. The update-linear-parameters argument controls whether or not the membrane parameters for the individual segment structures and the linear input parameters for the cells are updated to reflect the addition of the virtual spines.

Using the SEGMENTS argument allows a variable spine density for a given cell, while reference to the CELL-TYPE (or all cell types) implies that the spine density is constant throughout the dendritic tree. This function *does not* add virtual spines to the soma.

More elaborate spines may be modeled by appropriate anatomical construction using calls to `CREATE-SEGMENT`.

17 Concentration Integrators, Pumps and Buffers

This section describes concentration integrators and associated pumps and buffers. This documentation is not complete. Refer also to source code in `conc-part.lisp`, `conc-int.lisp`, `buffer.lisp`, and `pump.lisp`. See also Section 26.

17.1 The Classes of Concentration Integrator Types

Concentration integrators may be either intracellular or extracellular, and are "fed" from channels or synapses whose ionic species include the species specific to the integrator in question. The `CONC-INT-TYPE-DEF` macro, discussed in Section 11, creates an entry in the parameter library for concentration integrator types.

At present there are three classes of concentration integrator types, including `:FIRST-ORDER`, `:MULTI-SHELL` and `:GENERIC.` `:FIRST-ORDER` class concentration integrator types have only a single compartment. Both `:MULTI-SHELL` and `:GENERIC` classes may have 1 to 3 variable concentration compartments, and a constant concentration "core", as described in Section 17.3.

17.1.1 Reference Volumes Associated With Concentration Integrators

The volumes of the concentration integrator compartments are determined either implicitly or explicitly, as explained below. In all cases of intracellular concentration integrators, the total volume associated with a given cell element (soma or segment) is given by the function:

element-concentration-volume *element* &optional *consider−virtual−elements* *model−type* [*Function*]

This returns the volume in cubic microns of the cell element associated with ELEMENT (which is given by the function `ELEMENT-VOLUME`), minus the volume of any nucleus associated with the cell element, as indicated by the element parameter `'NUCLEUS-DIAMETER` (microns).

17.2 `:FIRST-ORDER` Concentration Integrator Types

Concentation dynamics of :FIRST-ORDER concentration integrators are based on the following differential equation:

$$d[X]/dt = -\frac{[X]-[X]_0}{\tau} + KI_X - K'J_X \tag{4}$$

where $[X]_0$ is the equilibrium concentration of X, τ is the original characteristic time constant of the integrator adjusted for the integrator's Q_{10} rate coefficient factor $\alpha_{Q_{10}}$, thus $\tau = \tau_{original}/\alpha_{Q_{10}}$. I_X is the total membrane current of ion X from channels or synapses, and K is a constant converting that current to a concentration change, taking into account the valence of X and the volume of the integrator compartment. J_X represents the membrane pump flux of X per unit area, and K' is a constant converting that current to a concentration change taking into account the membrane area associated with the pump and the volume of the integrator compartment.

To solve for $[X]$ at time t_{n+1}, and using the approximations:

$$d[X]/dt \approx \frac{[X]_{n+1}-[X]_n}{\Delta t}$$

$$[X] \approx \frac{[X]_{n+1}+[X]_n}{2}$$

where $\Delta t = t_{n+1} - t_n$, we arrive at the following (implicit) integration formula:

$$[X]_{n+1} = \frac{[X]_n(\Delta t/2-\tau) - \Delta t[X]_0 + \Delta t\tau(KI_X - K'J_X)}{-\tau - \Delta t/2} \tag{5}$$

An example of a :FIRST-ORDER concentration integrator type definition is given in the following `CONC-INT-TYPE-DEF` (from `src/parameters/traub91-channels.lisp`, derived from Traub et al., 1991):

```
(conc-int-type-def
 '(ca-in-traub91
   (class . :first-order)
   (species . ca)
   (valence . 2)
   (intra-p . t)      ; Intracellular concentration
   (tau . 13.33)      ; milliseconds
   (juxtamembrane-shell-thickness . 1.0) ; microns
   (core-conc . 0.0e-5)))                ; mM
```

Another example of this class may be found in `src/parameters/warman94-channels.lisp`, derived from Warman et al., 1994. If there is no `TAU` entry in the `CONC-INT-TYPE-DEF` for a `:FIRST-ORDER` concentration integrator, then it is taken to be infinite, and removal of ion is accomplished by membrane pumps alone (see, for example, the definition of `CA-IN-JAFFE-94` in `src/parameters/jaffe94-chs.lisp`, derived from Jaffe et al., 1994).

17.2.1 Volumes associated with `:FIRST-ORDER` concentration integrator types

Unless an explicit definition is given (see Section 17.3.1), the volume of a `:FIRST-ORDER` type concentration integrator is calculated according to the value of the juxtamembrane shell thickness parameter of the type, and the surface area of the associated cell element, as given in the first two lines of Table 3 (for compartment "Shell 1").

17.3 Geometry of Multiple Compartment Concentration Integrators

As stated above, there can be up to three compartments and a core in `:MULTI-SHELL` or `:GENERIC` concentration integrator types, named shell 1, shell 2, shell 3, and core. All integrator types have a shell 1, and the inclusion of other compartments is determined by `SHELL-2-P`, `SHELL-3-P` and `CORE-P` entries in the `CONC-INT-TYPE-DEF` form (each of these default to NIL).

The functional geometry of the integrator compartments is defined by:

- The volumes of the compartments.
- The connectivity of the compartments.
- The compartments which communicate directly with the membrane (shell 1 and/or shell 2).

Compartments volumes, intercompartment diffusion areas and membrane areas may be defined explicitely or implicitly. Compartment connectivity is determined by the diffusion coefficients assigned to each pair of compartments, where a zero value of a given pair implies that they are not connected.

We shall first describe how explicit formulae for these parameters may be defined. We then will describe the implicit algorithm that is built in to the system.

17.3.1 Explicit definition of compartment volumes

Explicit definition of compartment volumes is accomplished by including a `VOLUMES` entry in the `CONC-INT-TYPE-DEF` form. This entry is a list of compartment names and functions for determining their volumes. These functions must take an instance of a concentration integrator as a single arg, and return the appropriate compartment volume in μm^3. For example:

```
(volumes . ((1 (lambda (cint) (* (element-volume cint) .01)))
            (2 (lambda (cint) (* (element-volume cint) .09)))
            (3 (lambda (cint) (* (element-volume cint) .4)))
            (core (lambda (cint) (* (element-volume cint) .5)))))
```

As described in Section 6.8, `ELEMENT-VOLUME` returns the total volume in cubic microns of the cell element associated with its argument (here, a concentration integrator CINT).

17.3.2 Explicit definition of inter-compartment areas

Explicit definition of inter-compartment diffusional areas is accomplished by including a DIFFUSION-AREAS entry in the CONC-INT-TYPE-DEF form. This entry is a list of pairs of compartment names and either a number or function for determining their diffusional areas. These functions must take an instance of a concentration integrator as a single arg, and return the appropriate membrane area in μm^2. For example:

```
(diffusion-areas . (((1 3) (lambda (cint) (* (element-area cint) 0.1)))
                    ((2 3) 1.342e2)
                    ((core 3) (lambda (cint) (* (element-area cint) 0.5)))))
```

There is no assumed ordering of the compartment pairs. As described in Section 6.8, ELEMENT-AREA returns the total area in square microns of the cell element associated with its argument (here, a concentration integrator CINT).

17.3.3 Explicit definition of compartment membrane areas

Explicit definition of inter-compartment diffusional areas is accomplished by including a MEMBRANE-AREAS entry in the CONC-INT-TYPE-DEF form. This entry is a list of compartment names and either a number or function for determining their membrane areas. Normally this will apply to shells 1 and 2 only, and are used for calculating membrane areas for any pumps associated with a given compartment (see Section 17.10). These functions must take an instance of a concentration integrator as a single arg, and return the appropriate membrane area in μm^2. For example:

```
(membrane-areas . ((1 (lambda (cint) (* (element-area cint) .1)))
                   (2 (lambda (cint) (* (element-area cint) .3)))))
```

17.3.4 Implicit definition of compartment volumes

If there are no VOLUMES specified for a concentration integrator type, then compartment volumes are determined with a formula that depends on the SHELL-2-P, SHELL-3-P and CORE-P boolean entries, and on numeric values of JUXTAMEMBRANE-SHELL-THICKNESS, ALPHA-S and INNER-SHELL-THICKNESS entries in the CONC-INT-TYPE-DEF form. The various layouts that result from this formula are shown in Figure 3. This implicit calculation of concentration integrator compartment volumes is summarized in Table 3. Note in particular that if JUXTAMEMBRANE-SHELL-THICKNESS is 0 (the default value) then there is only one compartment (shell 1), and it occupies the entire element concentration volume (given by the function ELEMENT-CONCENTRATION-VOLUME).

Referring to this Figure, we can make a distinction between physical shells and the name of the compartments: physical shells can include a "juxtamembrane" shell, and up to two inner shells.

As stated, there is always a shell 1, and it is always bounded on one side by the membrane; in other words it either makes up part or the entirety of the juxtamembrane shell. Shell 2 may be thought of as either bounded by the membrane, in which case it is also part of the juxtamembrane shell, or be bounded by shell 1, in which case shell 2 is really a shell. Note, however, that the JUXTAMEMBRANE-SHELL-THICKNESS parameters applies to both shells 1 and 2, even when shell 2 is not phenominologically part of the juxtamembrane shell.

Shell 3 may be bounded on the membrane side either by the juxtamembrane shell or, if shell 2 is bounded by shell 1, bounded by shell 2. The core compartment is always bounded by outermost (innermost for extracellular integrators) variable concentration shell; thus it is always arranged as the innermost (outermost for extracellular integrators) compartment.

In the case of intracellular integrators, the volume of compartment(s) closest to the center may be either be a function of their thickness (shell 1, shell 2, juxtamembrane combination of shells 1 and 2, or shell 3), or given by the total volume of the soma or segment minus the volumes of any comparments lying closer to the membrane (the previous shells or the core).

The volume of shell 1 is given by the product of the :ALPHA-S slot of the concentration integrator type, the surface area of the cell element, and the :JUXTAMEMBRANE-SHELL-THICKNESS of the concentration integrator type. The volume of shell 2 is given by the product of 1 minus the value in the :ALPHA-S slot of the concentration integrator type, the surface area of the cell element, and the thickness of the juxtamembrane

Compartment	D_{juxta}	D_{inner}	SHELL-2-P	SHELL-3-P	CORE-P	Compartment Volume	Layout
Shell 1	=0	n/a	NIL	n/a	n/a	V_{tot}	A*
	>0	n/a	NIL	n/a	n/a	$D_{juxta} \times A$	A,D
	=0	n/a	T	n/a	n/a	$\alpha_S \times V_{tot}$	B,G*
	>0	n/a	T	n/a	n/a	$\alpha_S \times D_{juxta} \times A$	C,E,F,H,I,J
Shell 2	=0	n/a	T	n/a	n/a	$(1-\alpha_S) \times V_{tot}$	B,G*
	>0	n/a	T	n/a	n/a	$(1-\alpha_S) \times D_{juxta} \times A$	C,E,F,H,I,J
Shell 3	=0	n/a	n/a	n/a	n/a	n/a	no shell 3
	>0	>0	n/a	T	n/a	$D_{inner} \times A$	C,F,H,J
	>0	=0	n/a	T	n/a	$V_{tot} - V_{juxta}$	C,H*
Core	=0	n/a	n/a	n/a	n/a	n/a	no core
	>0	=0	n/a	n/a	n/a	n/a	no core
	>0	n/a	n/a	NIL	T	$V_{tot} - V_{juxta}$	D,E,I*
	>0	>0	T	T	T	$V_{tot} - (V_{juxta} + V_{inner})$	F,J*

Table 3: Implicit calculation of concentration integrator compartment volumes. D_{juxta} and D_{inner} are the values of the JUXTAMEMBRANE-SHELL-THICKNESS and INNER-SHELL-THICKNESS entries of the CONC-INT-TYPE-DEF, respectively, in μm. SHELL-2-P, SHELL-3-P and CORE-P are boolean CONC-INT-TYPE-DEF keywords (all default to NIL). A = cell element surface area, V_{tot} = cell element volume, V_{juxta} = volume shell 1 + volume shell 2, V_{inner} = volume shell 3. "n/a" means not applicable. "Layout" refer to those shown in Figure 3. Note that each of these CONC-INT-TYPE-DEF keyword entries are names of concentration integrator type slots as well. Entries which include an "*" in the Layout column are only appropriate for intracellular concentrtation integrator types. Note that the volume of a core compartment has no effect on the concentration dynamics since by definition the concentration of the core is constant - this volume will only effect the measurement of the total concentration of the integrator (Section 17.6).

shell. The volume of the inner shell 3 is given by the product of the surface area of the cell element, and the :INNER-SHELL-THICKNESS of the concentration integrator type.

17.4 :MULTI-SHELL Concentration Integrators

For the :MULTI-SHELL class, diffusional flux between two compartments i and j is given by the standard Fick's relationship:

$$J_{X(ij)} = D_{ij} A_{ij} \frac{[X]_i - [X]_j}{\Delta x_{ij}} \tag{6}$$

where $J_X(ij)$ is the flux of X from compartment i to compartment j, D_{ij} is the diffusion coefficient between the compartments (given in cm^2 sec^{-1}), A_{ij} is the diffusional area between the compartments (given in μm^2), and $[X]_i$ and $[X]_j$ are the concentrations of X in compartments i and j, respectively. Δx_{ij} is the diffusion distance appropriate for compartments i and j. The total change in the concentration of X for a compartment i is then:

$$d[X]_i/dt = KI_X - K'J_X - 1/V_i \sum_{j \neq i} J_{X(ij)} \tag{7}$$

where V_i is the volume of compartment i. KI_X and $K'J_X$, accounting for the membrane pore current and membrane pump flux for compartment i, respectively, are as described in Equation 4. By definition, the change in concentration of a core compartment is always zero. Numerical integration for the associated system of differential equations is done either with an implicit (trapezoidal integration) or explicit (forward Euler) formula, depending on the value of the global variable *IMPLICIT-CONC-INT-INTEGRATION* (default T). Currently, implicit integration of :MULTI-SHELL concentration integrators is disabled when there is no shell 3 and there is diffusion between shells 1 and 2 and the core (corresponding to layout I in Figure 3).

17.4.1 :MULTI-SHELL Diffusion Coefficients

Diffusion constants are determined by the DIFFUSION-COEFFICIENT entry in the CONC-INT-TYPE-DEF form. This entry is either a list of compartment pair lists, followed by their respective diffusion coefficient, or a single number that applies to all compartment pairs, with all values given in 0.8e-5 cm^2 sec^{-1}. In the first case, for example:

```
(diffusion-coefficient . (((1 2) 0.8e-5)
                          ((2 3) 0.8e-5)
                          ((core 3) 0.20e-5)))
```

There is no assumed ordering of the compartment pairs. In the second case, for example:

```
(diffusion-coefficient . 0.8e-5)
```

all diffusion between compartments in this integrator type has a diffusion coefficient of 0.8e-5 cm^2 sec^{-1}. The effective diffusion coefficients during a simulation are the original coefficients, as just described, muliplied by the integrator's Q_{10} rate coefficient factor $\alpha_{Q_{10}}$.

17.4.2 :MULTI-SHELL Diffusion Distances

Diffusion distances are determined by the DIFFUSION-DISTANCES entry in the CONC-INT-TYPE-DEF form. This entry is either a list of compartment pair lists, followed by their respective diffusion distance, or a single number that applies to all compartment pairs, with all values given in μm. In the first case, for example:

```
(diffusion-distances . (((1 3) (lambda (cint) (/ (element-volume cint) (element-area cint))))
                        ((2 3) 0.1)
                        ((core 3) 0.2)))
```

There is no assumed ordering of the compartment pairs. In the second case, for example:

```
(diffusion-distances . 0.4)
```

all diffusion between compartments in this integrator type has a diffusion distance of 0.4μm.

17.4.3 :MULTI-SHELL Interdigitated Juxtamembrane Shells and Diffusional Areas

As given in Equation 6, the diffusion between compartments is calculated according to the shared area between the compartments and the diffusion constant for each pair of compartments.

When both shells 1 and 2 are defined to be juxtamembrane, they are said to be functionally interdigitated. The idea is that the entire shell adjacent to the membrane is divided up into many subregions, each of which belongs to one of two groups (called shell 1 or 2). Each group is distinguished by a unique set of channels or synapses which traverse the membrane over a given subregion. Thus, the concentration in the set of subregions comprising shell 1 or 2 is evaluated over the total volume of the subregions, including the contributions of all the channels or synapses assigned to the appropriate shell. Diffusion between shell 1 and 2 depends of an effective diffusion area which conceptually depends on the relative interdigitation or dispersion of the subregions.

This interdigitation, or partitioning, is quantified by the :INTERDIGITATION-COEFFICIENT of the concentration integrator type ($1/\mu m$), which transforms the surface area of the membrane, A, into an effective length or border between the subregions. Thus, if the interdigitation coefficient is $\alpha_{1:2}$, then the effective diffusional area $A_{1:2}$ between shells 1 and 2 is given by:

$$A_{1:2} = \alpha_{1:2} \times A \times D_{juxta}$$

where D_{juxta} is the :JUXTAMEMBRANE-SHELL-THICKNESS. The diffusion area between the juxtamembrane shell 1 and shell 3 is given by the product of the :ALPHA-S slot of the concentration integrator type and the surface area of the cell element. The diffusion area between the juxtamembrane shell 2 and shell 3 is given by the product of 1 minus the value in the :ALPHA-S slot of the concentration integrator type and the surface area of the cell element.

The calculation of concentration integrator inter-compartmental diffusion areas is summarized in Table 4.

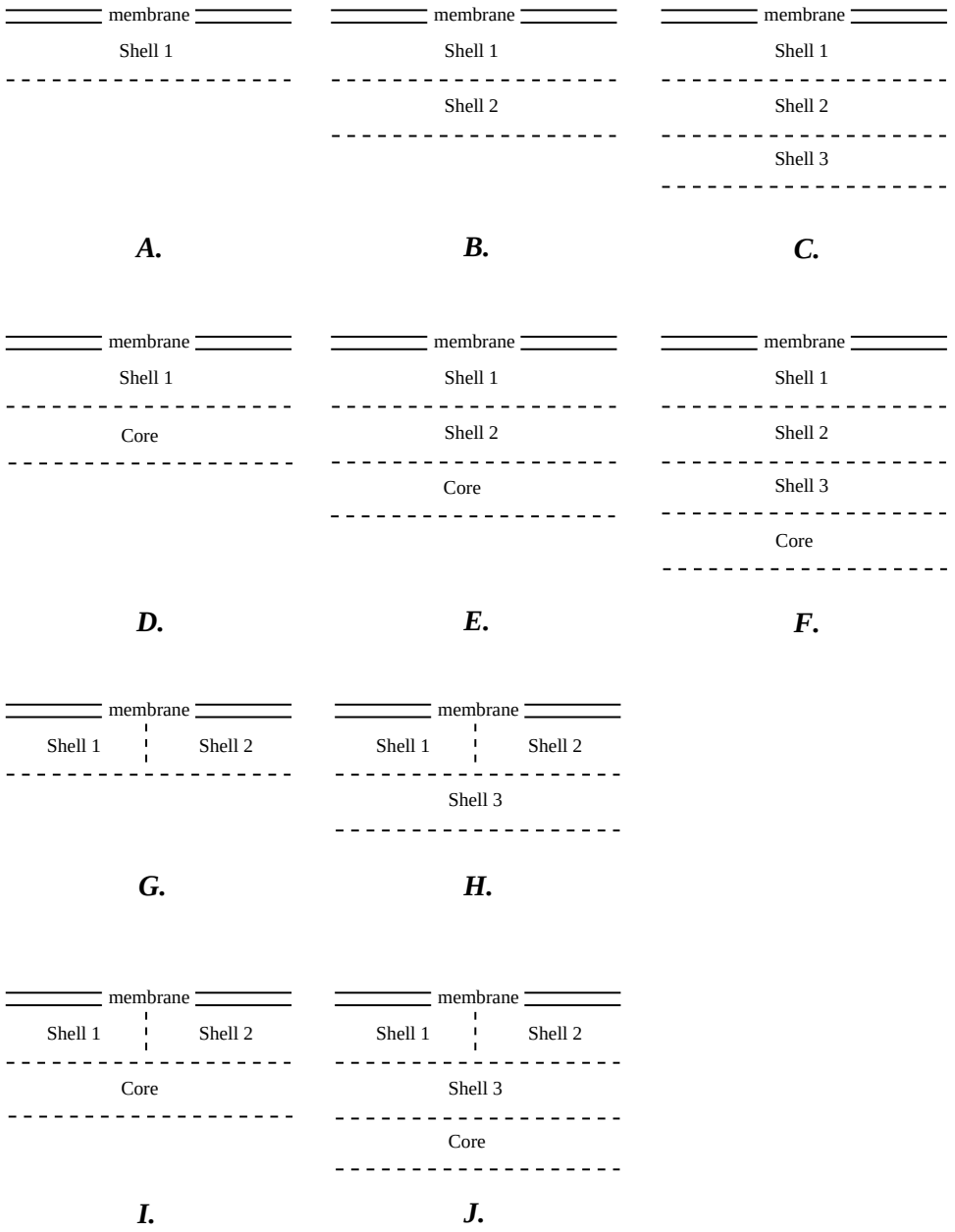

Figure 3: Possible arrangements of concentration integrator compartments, whether intra or extracellular. The juxtamembrane shell may be divided into two compartments, as seen in G - J, and specific channels or synapses may be assigned to either of these compartments. The layout used in the Working Model is shown in H. Core compartments have a constant concentration. Formulae for computing the compartment volumes depend on the layout, as described in Table 3. As suggested in Table 3, the existence of shell 2, shell 3, and the core is a function of the `SHELL-2-P`, `SHELL-3-P`, `CORE-P`, `JUXTAMEMBRANE-SHELL-THICKNESS` and `INNER-SHELL-THICKNESS` entries in the `CONC-INT-TYPE-DEF` (there is always a shell 1). However, for the `:MULTI-SHELL` integrator type class, a given compartment is only functional if the appropriate diffusion constants are non-zero.

		Diffusion Area Between Shells		
$\alpha_{1:2}$	D_{juxta}	1 and 2	1 and 3 (or Core)	2 and 3 (or Core)
>=0	>0	$\alpha_{1:2} \times D_{juxta} \times A$	$\alpha_S \times A$	(1 - α_S) $\times A$
<0	n/a	A	0	A
n/a	<=0	A	0	A

Table 4: Calculation of concentration integrator inter-compartmental diffusion areas. $\alpha_{1:2}$ and D_{juxta} are given by the INTERDIGITATION-COEFFICIENT (between shells 1 and 2) and JUXTAMEMBRANE-SHELL-THICKNESS (in μm) entries, respectively, in the CONC-INT-TYPE-DEF. A = cell element surface area. "n/a" means not applicable. Note that the last two combinations of $\alpha_{1:2}$ and D_{juxta} place shell 1 and 2 *en-face* with respect to each other (refer to layouts B, C, E and F in Figure 3). The diffusional areas between shell 1 and the core and shell 2 and the core apply when there is no shell 3 (layouts D, E and I in Figure 3). The diffusional area between shell 3 and the core, when they both exist, is given by A (layouts F, and J in Figure 3).

17.4.4 Defining a :MULTI-SHELL Concentration Integrator Type with CONC-INT-TYPE-DEF

An example of a :MULTI-SHELL type is given in the following CONC-INT-TYPE-DEF (taken from src/parameters/working-hpc.lisp as described in Borg-Graham 1999):

```
(conc-int-type-def
 '(CA-IN-HPC
   (class . :MULTI-SHELL)
   (species . CA)
   (intra-p . T)
   (shell-2-p . T)
   (shell-3-p . T)
   (juxtamembrane-shell-thickness . 1.0)
   (inner-shell-thickness . 0.0)
   (alpha-s . 10.0e-5)
   (interdigitation-coefficient . 1.0)
   (diffusion-coefficient . (((1 2) 8.0e-6)
                             ((1 3) 0.0)
                             ((2 3) 8.0e-6)))
   (transmembrane-concentration . 2.0)
   (core-conc . 0.00105)
   (pump-type-params . ((CA-HPC-MM 2)))
   (resting-free-conc . 5.0e-5)
   (instantaneous-buffer-enabled . T)
   (instantaneous-buffer-ratio . 20.0)))
```

This integrator type includes instantaneous buffering (*viz.* the INSTANTANEOUS-BUFFER-ENABLED and INSTANTANEOUS-BUFFER-RATI entries) and a membrane pump (*viz.* the PUMP-TYPE-PARAMS entry). It has juxtamembrane shells 1 and 2 and an inner shell 3 whose volume occupies the remaining volume of the cell element after accounting for the volume of shells 1 and 2. The shell thicknesses are in microns. Note that this model has no core: by convention, for the implicit calculation of compartment volumes, when the value for INNER-SHELL-THICKNESS is 0.0, then the inner compartment occupies the entire remaining cell element volume, as described in Table 3 (see the volumes for shell 3 and the core when $D_{inner} = 0$).

The RESTING-FREE-CONC entry is in mM, and is the concentration of free ion after taking into account any instantaneous buffer. The TRANSMEMBRANE-CONCENTRATION (referring to the concentration on the opposite side of the membrane from which the integrator is associated) and the CORE-CONC (normally for intracellular concentration integrators, referring to a fixed concentration compartment) are also in mM. These latter two concentrations are total concentrations of the ion; if there is no entry for RESTING-FREE-CONC, then the value for CORE-CONC is used, even though this is not intended to be the free concentration. The units for the diffusion coefficients are in $cm^2 sec^{-1}$.

The TRANSMEMBRANE-CONCENTRATION parameter of a concentration integrator is used to calculate reversal potentials for associated channels and synapses. If the TRANSMEMBRANE-CONCENTRATION parameter is not specified explicitly, the integrator will reference the DEFAULT-EXTRACELLULAR-CONCENTRATION or DEFAULT-INTRACELLULAR-CONCENTRATION value for the ionic species in question, as appropriate. These values are in turn supplied by global variables of the form *NA-CONC-EXTRA* and *NA-CONC-INTRA*, respectively (here for Na_+, likewise K for K_+, CL for Cl_-, CA for Ca_{2+}, MG for Mg_{2+}).

The PUMP-TYPE-PARAMS entry is a list of pump types and the associated integrator compartments (Section 17.10). In this example, the pump type CA-HPC-MM is assigned to the shell compartment 2.

17.5 :GENERIC Concentration Integrators

Concentration dynamics for the :GENERIC class of concentration are defined by user-defined functions, as described in Section 17.5.

For the :GENERIC class of concentration integrator type the evolution of the ionic concentration(s) is governed by explicit functions, whose single argument is a concentration itegrator, specified in the CONC-INT-TYPE-DEF. The possible calculations performed by these functions is reflected by the keyword used in the CONC-INT-TYPE-DEF:

- A C-N+1-FUNCTION, that returns the concentration of shell 1 (double float, in mM) of its concentration integrator argument at the next time step t_{n+1}. The result is then used to update the concentration in shell 1, thus:
$$[X]_{1,n+1} = \texttt{(C-N+1-FUNCTION CINT)}$$
- A DCDT-FUNCTION that returns the derivative of the concentration of shell 1 (double float, in mM/ms) of its concentration integrator argument at the the midpoint between the current time t_n and the next time step t_{n+1}. The result is then used to update the concentration in shell 1 by a simple forward Euler integration, thus:
$$d[X]_{1,n+1/2}/dt = \texttt{(DCDT-FUNCTION CINT)}$$
$$[X]_{1,n+1} = [X]_{1,n} + (\Delta t \times d[X]_{1,n+1/2}/dt \times \alpha_{Q_{10}})$$
where $\alpha_{Q_{10}}$ is the Q_{10} rate coefficient factor.
- A CONC-FUNCTION that specifically updates the compartment concentrations of its concentration integrator argmument for t_{n+1}. Note that with this method there is no implicit adjustement for Q_{10}.

For example (derived from Clay, 98; see src/parameters/clay-98.lisp), one might have:

```
(conc-int-type-def
 '(clay-98-k-ex
   (class . :generic)
   (c-n+1-function . clay-98-k-n+1-extra)
   ...
  )
```

where the C-N+1-FUNCTION is defined as:

```
(defun clay-98-k-n+1-extra (cint)
  (let* ((delta-t (*delta-t[n]*))
         (total-current (* 1.0e-6 (conc-int-membrane-current-component cint))) ; mA
         (total-current-per-unit-area (/ total-current (* 1.0e-8 (element-area cint)))) ; mA/cm2
         (accumulation-term (/ total-current-per-unit-area *CLAY-98-FARADAY-PHI*)) ; M/ms
         (conc-shell-extra (conc-int-shell-1-free-conc-n cint)) ; mM
         (f-of-cn (/ 1 *CLAY-98-TAU*)))
    (/ (+ (* 1000 accumulation-term)        ;mM/ms
          (* conc-shell-extra
             (- (/ 1 delta-t)
                (/ 1 (* 2 *CLAY-98-TAU*))))
```

```
          (* *K-CONC-EXTRA*
             f-of-cn))
       (+ (/ 1 delta-t)
          (/ 1 (* 2 *CLAY-98-TAU*))))))
```

Otherwise, we could have (also taken from `src/parameters/clay-98.lisp`),

```
(conc-int-type-def
 '(clay-98-k-ex
   (class . :generic)
   (dcdt-function . clay-98-dkdt-extra)
   ...
  )
```

where the `DCDT-FUNCTION` is defined as:

```
(defun clay-98-dkdt-extra (cint)
  (let* ((total-current (* 1.0e-6 (conc-int-membrane-current-component cint))) ; mA
         (total-current-per-unit-area (/ total-current (* 1.0e-8 (element-area cint)))) ; mA/cm2
         (accumulation-term (/ total-current-per-unit-area *CLAY-98-FARADAY-PHI*)) ; M/ms
         (conc-shell-extra (conc-int-shell-1-free-conc-n cint)) ; mM
         (difference-conc (* 0.001 (- conc-shell-extra *K-CONC-EXTRA*)))) ; M
    (* 1000.0                              ; M to mM
       (+ accumulation-term                ; M/ms
          (- (/ difference-conc *CLAY-98-TAU*))))))  ; M/ms
```

Note: with either the `C-N+1-FUNCTION` or `DCDT-FUNCTION` options, the evaluation of :GENERIC concentration integrators implicitly sets the concentration of shell 1 after calling the user-specified function just described. Multiple compartments may be defined for the `:GENERIC` class, but the user-specified function must take care of updating the concentrations in compartments other than shell 1.

17.6 Plotting Concentration Data

Any of the shell concentrations may be plotted with the `ENABLE-ELEMENT-PLOT` function, where the optional `DATA-TYPE` argument may be:

```
1      -> Shell 1
2      -> Shell 2
3      -> Shell 3
'TOTAL -> Average concentration over the entire
  volume of the associated cell element [default]
'ALL   -> Plot all of the above, as appropriate
          for concentration integrator type
```

If `'TOTAL` is specified, the average concentration of X is calculated with:

$$\frac{V_1 \times [X]_1 + V_2 \times [X]_2 + V_3 \times [X]_3 + V_{core} \times [X]_{core}}{V_1 + V_2 + V_3 + V_{core}}$$

Where V_a and $[X]_a$ refer to the volume of shell a and the concentration of $[X]$ in shell a, respectively. This "total" concentration may be compared to that measured by flourescent probes in which it is assumed that the concentration of some species is integrated over the entire volume of a section of neuron.

When the global variable `*PLOT-TOTAL-CONCENTRATIONS-SEPARATELY*` is T, then a separate plotting window is generated for the average (or "total") concentrations. This is useful when the magnitude of the juxtamembrane shell concentrations is much larger than that of the concentration averaged over the entire cell-element volume.

17.7 Pore Current and Concentration Integrator Shells

Specific channels or synapses on a cell element may be assigned to shell 1 or 2 (that is. both may be juxtamembrane). This assignment is defined according to the `:SHELL-1-PORES` and `:SHELL-2-PORES` slots of each concentration integrator. These slots in turn are set automatically when channels or synapses are created, according to the `:CONC-INT-TYPE-PARAMS` slots of the appropriate channel or synapse types (see also Sections 13 and s:synapse). The list in these slots is set for each ion that passes through the channel or synapse type, and includes an entry for a concentration integrator type, for the shell (1 and/or 2) and the proportion of the current associated with the shell, e.g.:

```
(CHANNEL-TYPE-DEF
 '(FOO
   ...
   (conc-int-type-params . '((CA-INTRA (1 0.7) (2 0.24)) (K-EXTRA (1 1)))
   ...
  )
```

Thus, for this example, the installation of every channel of type FOO in a soma or segment will automatically create a CA-INTRA and a K-EXTRA concentration integrator. The `:SHELL-1-PORES` slot of the CA-INTRA integrator will include `(<FOO channel> 0.7)` (it could also have other channels or synapses assigned to it), and likewise `:SHELL-2-PORES` slot will include `(<FOO channel> 0.24)`. Finally, the `:SHELL-1-PORES` slot of the K-EXTRA integrator will include `(<FOO channel> 1.0)`. Again, this entire process (integrator creation and assignment to channel or synapse) is automatic.

17.8 Concentration Integrators and Concentration Particles

When a concentration integrator is assigned to a node that already has concentration particle(s) associated with it, then that integrator is assigned to all the previously defined concentration particles. If a concentration particle is at a node without a concentration integrator, then the concentration used for evaluating that particle is given by the global variable `*X-CONC-INTRA*`, where `X` refers to the appropriate ionic species (e.g. `*CA-CONC-INTRA*`, `*K-CONC-INTRA*`, etc.). Note that normally these variables are constant during a simulation.

17.9 Concentration Integrator Channel or Synapse 'CONC-INT-DELTA Parameter

An additional parameter δ, assigned to specific pore elements (channels or synapses) is a scaling coefficient (less than 1) for the current integrated by associated integrators. This parameter is also discussed in Section 13.

This parameter may be appropriate when a model places all channels (or synapses) at some single position (typically the soma) even if those channels are in reality distributed throughout the dendritic tree. Such a distribution may give a reasonable first estimation as to the *electrical* image (more so under current clamp) from the soma of the channel behaviour. However, the concentration profile elicited by the "unnatural" concentration of channels could be overestimated, all other factors being equal (e.g. distribution of buffer and pump mechanisms). As a first approximation, δ is estimated by the ratio of the actual conductance of a channel at the location in the model, divided by the total conductance of the channel for the entire cell. For example, in the Working Model:

```
* (element-parameter "HPC-soma-CA-T-GEN" 'conc-int-delta)
0.34
```

This parameter may be specified in a `CHANNEL-TYPE-DEF` (Section 13), or assigned after channels are created, for example:

```
(loop for ch in (channels) when (element-of-ion-type-p ch 'ca)
      do (element-parameter ch 'conc-int-delta 0.34))
(set-conc-integrators-parameters))
```

The final call to `SET-CONC-INTEGRATORS-PARAMETERS` makes sure that the appropriate concentration integrators are setup properly. This parameter is also discussed in Section 13.

17.10 Pumps

There are several classes of pump types: `:MM` (Michaelis-Menton), `:FIRST-ORDER`, `:FIRST-ORDER-TAU-V` (where the tau depends on voltage of associated cell element), and `:GENERIC`. Each type is defined with the `PUMP-TYPE-DEF` macro, discussed in Section 11. At present only the `:MM` and `:GENERIC` classes of pump are well tested.

Note that pump types and pumps are typically created automatically, when a concentration integrator type or associated integrators are created whose parameter library definitions reference a particular pump type. Furthermore, pumps must be explicitly associated with specific compartments in `CONC-INT-TYPE-DEF` forms, for example as shown previously in the `(PUMP-TYPE-PARAMS . ((CA-HPC-MM 2)))` entry in the definition of the `CA-IN-HPC` type.

Thus, pump types generally have an ionic `:SPECIES`, but this is used mainly for documentation since the concentration references are done via the associated concentration integrators.

Pumps are initialized according to the resting free concentration in the associated concentration integrator compartment. This value is used *even* if concentation integrator values are initialized with `*CONC-INT-INITIALIZATIONS*` (see Section 26).

17.10.1 Michaelis-Menton Pump Types

The equation governing the ionic flux of a Michaelis-Menton pump type is:

$$J_X \;=\; V_{max}\frac{[X]}{K_d + [X]} - J_{leak}$$

where J_X is the removal rate of X per unit area, V_{max} is the maximum flux rate per unit area, $[X]$ is the concentration of X is the comparment associated with the pump, K_d is the half-maximal [X], and J_{leak} compensates for the resting pump rate. For example, as described in `src/parameters/pumps.lisp` (taken from Jaffe et al., 1994):

```
(pump-type-def
 '(CA-JAFFE-94
   (class . :MM)
   (v-max . 6.0e-11)     ; millimole ms^-1 cm^-2
   (kd . 0.01)           ; millimolar
   (species . CA)))
```

17.10.2 First Order, Voltage-Dependent τ Pump Types

(To be completed)....

As described in `src/parameters/pumps.lisp` (taken from Yamada et al., 1989):

```
(pump-type-def
 '(CA-YAMADA-89
   (species . ca)
   (class . :first-order-tau-v)    ; tau depends on voltage of associated cell element
   (equilibrium-conc . 50.0)       ; mM
   (tau-function . ,#'(lambda (voltage) (* 17.7 (exp (/ voltage 35.0)))))))
```

The `TAU-FUNCTION` entry must have a single voltage argument (in mV),and return a time constant value in milliseconds.

17.10.3 Generic Pump Types

Pump membrane flux of X is determined by a `PUMP-FUNCTION` entry in the `PUMP-TYPE-DEF`, which takes an instance of a pump and the concentration (in mM) of the compartment associated with the pump, and returns the flux of X in millimole/millisecond.

For example (derived from Clay, 98; see `src/parameters/clay-98.lisp`):

Pump Compartment	:SHELL-2-P	Pump Membrane Area
Shell 1	T	$\alpha_S \times A$
	NIL	A
Shell 2	T	$(1 - \alpha_S) \times A$

Table 5: Calculation of pump membrane area. Pumps, which are only associated with specific juxtamembrane concentration integrator compartments, are parameterized by the membrane area of those compartment.

```
(pump-type-def
 '(k-clay-98
   (species . k)
   (class . :generic)
   (pump-function . clay-98-pump)))
```

where the PUMP-FUNCTION is defined with:

```
(defun clay-98-pump (pump conc-shell-extra)
  (let* ((difference-conc (* 0.001 (- conc-shell-extra *k-conc-extra*))) ; M
         (result
          (* 1000.0                     ; mM/M
             (* *clay-98-gamma*         ; 1/ms
                (/ difference-conc      ; M
                   (expt (+ 1 (/ difference-conc *clay-98-Kd*)) 3))))))
    (* 0.001                            ; L/mL
       (pump-conc-int-compartment-volume pump) ; mL
       result)))                                ; millimole / L ms
```

17.10.4 Pump Membrane Areas For Concentration Integrators

The calculation of a pump's membrane area depends on its associated concentration integrator. This calculation may be done explicitely as described in Section 17.3.3, or implicitely, as summarized in Table 5.

17.11 Instantaneous Buffers

Instantaneous buffers for a concentration integrator type are defined in terms of buffer ratios, given by an INSTANTANEOUS-BUFFER-RATIO entry in the CONC-INT-TYPE-DEF form. This entry can either be a list of compartment names followed by buffer ratio, or a single number which implies that all compartments have an instantaneous buffer, whose ratio is given by the number. In the former case, for example:

```
(INSTANTANEOUS-BUFFER-RATIO . `((1 20.0) (3 100.0)))
```

Thus, the value given is the dimensionless ratio of $[X]_{bound}/[X]$ for a given compartment. Shells not included in this list, or with a ratio of NIL, are assumed not to have an instantaneous buffer. The INSTANTANEOUS-BUFFER-ENABLED entry in the CONC-INT-TYPE-DEF enables the instantaneous shell buffers.

17.12 Other Buffers

Not available yet.

17.13 Concentration Clamp

A convenient way to impose a concentration clamp (and turn it off) is with the functions

concentration-clamp *element* &optional *concentration* [*Function*]

concentration-clamp-off *element* &optional *concentration* [*Function*]

`CONCENTRATION-CLAMP` turns off all concentration integrators associated with ELEMENT. If CONCENTRATION is a number (mM, default NIL), then this sets the steady-state value of the integrators to this value. Returns the steady-state concentration(s). `CONCENTRATION-CLAMP-OFF` is the same, except that it turns all the appropriate concentration integrators on.

17.14 Functions of Interest

conc-int-shell-1-free-conc-n *cint* [*Function*]

conc-int-shell-1-free-conc-n+1 *cint* [*Function*]

conc-int-shell-2-free-conc-n *cint* [*Function*]

conc-int-shell-2-free-conc-n+1 *cint* [*Function*]

conc-int-shell-3-free-conc-n *cint* [*Function*]

conc-int-shell-3-free-conc-n+1 *cint* [*Function*]

These return the concentration (mM) of free ion in the appropriate shell of CINT at time t_n or t_{n+1}, as indicated by the function name.

conc-int-core-free-conc *cint* [*Function*]

Concentration (mM) of free ion in core compartment of CINT.

conc-int-membrane-current-component *cint* &optional *(shell 1)* [*Function*]

Returns the concentration derivative [mM/ms], for non `:GENERIC` integrators, or the current [nA] for `:GENERIC` integrators, for the compartment SHELL of CINT that is due to that compartment's associated `:SHELL-PORES` and pumps. The coefficient `:BETA-CURRENT-sh`, specific for a given CINT, converts channel current (nA) to d[x]/dt (mM/ms); `:BETA-CURRENT-sh` = 1 for `:GENERIC` integrators.

18 Consolidating Dendritic Trees

Assuming a defined branching structure, various algorithms may be employed to transform anatomical coordinate data into a compartmental model. The simplest approach, of course, is a one-to-one mapping, where there is one compartment (dendritic segment) for each anatomical location, with the location and dimensions of that compartment taken as some function of the associated anatomical point and either one or two adjacent points.

For most practical simulations this produces an over-detailed representation since typical anatomical studies include on the order of thousands of dendritic locations and associated diameters. Here we describe one method for reducing such a description into fewer compartments, with the minimum number given by twice the number of branch points in the dendritic tree, plus one. The motivation for this method is to preserve both the 3-dimensional shape of the neuron and the electrotonic properties of the tree as best as possible.

The following procedure can be evoked from the (Main Menu) "Edit circuit elements" -¿ "Examine/modify distribution of electrotonic lengths" option. Note that a histogram plot of the distribution of electrotonic lengths in the loaded cells is also available.

18.1 Tree Reduction Algorithm

The reduction is iterative and operates on a pair of segments at a time. Starting from the soma and working down each proximal trunk, two consecutive segments, call them A and B, are combined into one if there is no other segment common to both (i.e. the connection is not a branch point) *and* if the electrotonic length of the candidate replacement segment is less than that set by some *a-priori* criteria, i.e. a parameter L_{max}, set by the global variable:

maximum-electrotonic-length *0.25* [*Variable*]

The process continues distally - if A and B were combined in the previous step into a segment AB, the next pair consists of AB and the segment distal to the original segment B (call it C). Otherwise, the next pair to be considered is B and C.

A more stringent requirement is that the electrotonic length of the replacement segment be less than `*MAXIMUM-ELECTROTONIC-LENGTH*`. This is more stringent since the sum of the consecutive segments' electrotonic length will underestimate the electrotonic length of the replacement segment constructed with the criteria listed below. The tighter test is controlled by the global variable:

use-strict-lambda-criterium t [*Variable*]

The disadvantage is an increase in processing time. The parameters of the replacement segment are derived according to the following constraints:

1. The total axial series resistivity of the two original segments is conserved.
2. The total membrane impedance (area) of the two original segments is conserved.
3. The end points of the new segment correspond to the non-common end points of the original two segments.

In order to meet these constraints an additional parameter for each segment must be introduced, which we are choosing to be a coefficient, a, of the cytoplasmic resistivity that is used when determining the segment intracellular axial resistance (the default value for a is 1). This coefficient and the dimensions of the new segment are calculated as follows. Let l_i be the segment length, d_i the segment diameter, and a_i the coefficient for the segment axial resistivity, where i is p, d, or n for the proximal, distal, or new segment, respectively. As stated above, the length of the new segment, l_n is fixed by the far endpoints of the original proximal and distal segments. Conservation of membrane area (that is, conservation of membrane capacitance and resistance) gives:

$$d_n = \frac{(l_p \times d_p) + (l_d \times d_d)}{l_n}$$

Conservation of axial resistance gives:

$$a_n = \frac{(l_p \times a_p)/d_p^2 + (l_d \times a_d)/d_d^2}{l_n/d_n^2}$$

This technique yields a new segment whose cable parameters are similar to the concatenation of the two parents. However, the actual proximal input impedance of the new segment will always be greater than or equal to the input impedance of the former segment pair, since the membrane impedance is now farther away electrically from the proximal end of the new segment, relative to the membrane impedances of the original pair. Thus, the input impedance of a transformed (consolidated) tree will always be greater than or equal to the original tree, and the difference in the impedances will be greater the larger the consolidation (equivalently, the larger the L_{max}). As one example:

```
     * (surf)

      ** Surf-Hippo: The MIT CBIP Neuron Simulator (Version 2.2) **

     Reading in circuit j43d...
     ; Loading #p"/usr/local/surf-hippo/anatomy/j43d.sparcf".
     Destroying zero length segment 4-3-56
     Destroying zero length segment 11-1-1
     Destroying zero length segment 11-2-1
     Locating segments...

      76 branch points and 3369 segments processed.
     Sunday, 10/30/94 03:38:40 am EST

     Simulation 'j43d-619512' [File: /usr/local/surf-hippo/anatomy/j43d.sparcf]
     1 cell type, 1 cell, with 3370 nodes. Temperature 27.0 degrees(C)
     There is 1 soma, 1 current source, and 3369 segments.

       Cell-type V1-pyramidal:
       Rm 40000.0, Rm-sm 40000.0 (ohm-cm2)
       Soma shunt 1.0e+30 ohms, Ra 200.0 ohm-cm, Cm-sm 1.0, Cm-den 1.0 (uF/cm2)
       E-soma-leak -70.0 mV, E-dendrite-leak -70.0 mV
       Cells of type V1-pyramidal are:
 j43d (soma @ [0.0 0.0 0.0])
   Max G-in/Min R-in = 1.34e-2 uS/7.46e+1 Mohms
   Soma passive R-in = 2.75e+3 Mohms
   Dendritic tree passive R-in (actual model) = 9.80e+1 Mohms
       (cable model) = 9.79e+1 Mohms
   Passive total R-in from soma (actual model) = 9.47e+1 Mohms
(cable model) = 9.45e+1 Mohms

     Locating segments...

      76 branch points and 1427 segments processed.

     Sunday, 10/30/94 03:44:10 am EST

     Simulation 'j43d-619545' [File: /usr/local/surf-hippo/anatomy/j43d.sparcf]
     Trees have been consolidated with a maximum electrotonic length of 0.3
```

```
     1 cell type, 1 cell, with 1428 nodes. Temperature 27.0 degrees(C)
     There is 1 soma, 1 current source, and 1427 segments.

       Cell-type V1-pyramidal:
       Rm 40000.0, Rm-sm 40000.0 (ohm-cm2)
       Soma shunt 1.0e+30 ohms, Ra 200.0 ohm-cm, Cm-sm 1.0, Cm-den 1.0 (uF/cm2)
       E-soma-leak -70.0 mV, E-dendrite-leak -70.0 mV
       Cells of type V1-pyramidal are:
 j43d (soma @ [0.0 0.0 0.0])
   Max G-in/Min R-in = 1.34e-2 uS/7.46e+1 Mohms
   Soma passive R-in = 2.75e+3 Mohms
   Dendritic tree passive R-in (actual model) = 9.84e+1 Mohms
       (cable model) = 9.79e+1 Mohms
   Passive total R-in from soma (actual model) = 9.50e+1 Mohms
(cable model) = 9.45e+1 Mohms
```

This algorithm may be run from a menu with the function:

consolidate-cells-tree [*Function*]

See also hints_misc.doc for Saving Cell Geometries.

18.2 Naming Consolidated Segments

The new name of a replacement segment is derived from the original segment names by the function `CREATE-CONSOLIDATED-SEGMENT-NAME` as follows:

When neither of the two names contain the character "^", then the new name is the concatenation of the original names, separated by "^", for example:

```
"19" "2-34" --> "19^2-34"
"foo" "bar" --> "foo^bar"
```

When either name contains a "^", the new name is constructed from the characters to the left of the first "^" in the first (proximal) segment name, and the characters to the right of the last "^" in the first (proximal) segment name, separated by "^", for example:

```
"foo^bar" "bar2"                 ->  "foo^bar2"
"1-2-43^1-2-48" "1-2-49"         ->  "1-2-43^1-2-49"
"1-2-43^4-3-48" "2-1-9^5-9-54"   ->  "1-2-43^5-9-54"
```

The motivation is that the new name reflects the names of both the distal and proximal segments that contribute to it.

19 Scripts - Searching Parameter Spaces

See also Section 5.

19.1 Setting Up a Series of Simulations

For running a series of simulations, for example to explore some parameter space, it is useful to enter a looping form directly to the Lisp interpreter. The usual option for loading/modifying/running circuits is:

- Load in the circuit
- Run loop with appropriate function calls to modify the circuit properties.
- Call `GOFERIT` in the inner loop to run the simulation.

For example, see `surf-hippo/circuits/demos/parameter-loop.lisp`.

19.2 Functions to Modify Element Parameters

In general, any slot of an element structure may be `SETF`'d as needed - the `src/sys/structures.lisp` file holds all the element structures and documentation of each slot. However, it is usually more convenient to use various functions specialized to modify slots, which in particular may take care of some non-obvious bookeeping. It is also strongly recommended that structure slots *not* be referenced directly in your code, since this makes upward compatibility of the code to later revisions of Surf-Hippo more problematic. Some of the functions that handle this parameter adjustment include:

iv-type-parameter *element param* &optional *(value nil value−supplied−p) (update t)* [*Function*]

set-element-absolute-iv-relation-ref *element iv−reference* [*Function*]

described in Section 12, and `CELL-TYPE-PARAMETER`, described in Section 9.1.

19.3 Changing Soma or Segment Dimensions

If the dimensions of a soma or a segment are changed, then the function:

set-element-membrane-parameters *element* &optional *ignore−membrane−elements* [*Function*]

where ELEMENT is either the soma or segment, will update the appropriate membrane elements to reflect the new dimensions, but not the geometry of the circuit. Note that if the dimensions of a soma is changed with the menus, then this is update is done automatically.

A complete revision of all circuit properties, including cell geometry, is done by calling `PROCESS-CIRCUIT-STRUCTURE`. It is usually not necessary to call this function explicitly.

19.4 Suppressing Plot Output

For multiple runs for which either some on-line analysis is the goal, or where data is to be saved for each run, then it may be useful to suppress plot output by setting the global variable `*PLOT-STANDARD-WINDOWS*` to NIL. Note that this will *not* prevent the specified plotted variables from being accumulated (which is necessary for any output); this will only prevent the actual plot generation.

19.5 Plot Elements Utilities

Information about which elements to plot and the correct data type are kept in a set of string list global variables (e.g. `*PLOT-NODES*`, `*PLOT-CHANNEL-CONDUCTANCES*`), which contain element names. These lists may be edited explicitly, or (better) you may use the `ENABLE-ELEMENT-PLOT` function:

enable-element-plot *element* &optional *data−type model−type* [*Function*]

There is also

disable-element-plot *element* &optional *data−type model−type* [*Function*]

These functions are described in Section 22 The function:

plot-all-somas [*Function*]

is convenient for plotting all the soma voltages in the circuit. The definition of this function is simply:

```
(defun plot-all-somas ()
  (enable-element-plot (somas)))
```

19.6 Examples

Here is a simple loop that delivers a series of current steps (start time 50.0 ms, stop time 1050.0 ms), and plots the results to distinct windows, each of which will show a comment indicating the level of the current clamp. This loop assumes that a circuit is loaded with (at least) one current source (the last created current source is referenced by the variable `*ISOURCE*`). The local binding (within the LET form) of the variables `*CREATE-NEW-SIMULATION-PLOTS*` and `*SIMULATION-PLOT-WINDOW-COMMENT*` insure that later simulations will not inherit values for these variables during the loop.

```
(loop for current from 0.0 to 3.0 by 0.25 do    ; Current is in nA.
       (pulse-list *isource* (list 50.0 1050.0 current))
       (let ((*CREATE-NEW-SIMULATION-PLOTS* t)
             (*simulation-plot-window-comment* (format nil "~anA step" current)))
        (gotimed)))
```

Note that the pulse list argument to `PULSE-LIST` is constructed with the `LIST` function, since the value of the current is given by the local variable `CURRENT`. If that value was a constant, then the shorter quoted list form could have been used (e.g. '((50.0 1050.0 1.2)). `CC-STEPS` is a function which does more or less the same thing as the loop above:

cc-steps *start−current stop−current* [*Function*]
step &key *(isource *isource*) (current−start−time 10.0) (current−stop−time (* 0.9 *user−stop−time*)) individual−plots comment (extra−comment "") (overlay−plots t) (show−plot−windows t) timeit (include−comment t)*

The START-CURRENT, STOP-CURRENT and STEP arguments are in nanoamps, and the time arguments are in milliseconds.

19.7 Elapsed Time Window, GC Messages, or Not

During a simulation a timer window will appear. However, if you are running a lot of rapid simulations, it is important to disable this indicator by:

```
(setq *show-time-remaining* nil)
```

Otherwise, the frequent refreshing of this window may hangs things up.

If you want a record of the GC statistics intermingled with other program output in the LISP interpreter window, then set both `*PRINT-OUT-GC*` and `*GC-ANNOUNCE-TEXT*` to T. See also Sections 29.9 and 29.10.

20 Element Data

This section describes the formatting and manipulation of data generated by Surf-Hippo simulations. See also Sections 22, 21, 23, 23.6 and 25.8.

20.1 Format of Element Data and Time Lists

Data - voltages, currents, conductances, particle states, concentrations, etc. - describing various aspects of the circuit components may be saved during the simulation for later plotting or analysis, but only if the plotting is enabled for a given element (and its appropriate data type).

The format for element data is a list of single-float numbers. The associated times for these numbers is given in the list `*SIM-PLOT-TIME-LIST*`, a global variable which holds the plotted time values from the last simulation, in correct order. Note that for efficiency data lists that are actually stored in the circuit element structures are in reverse time order. However, in the usual case functions based on the `ELEMENT-DATA` function (see below) are used for accessing element data, which returns the stored data in the correct time order.

Simulation data is saved onto the appropriate lists (including the `*SIM-PLOT-TIME-LIST*` list) every `*SAVE-DATA-STEP*` time steps during the run. The default value for this global variable is 2, i.e. data is saved every other time step).

For variable time step simulations, referencing the current value of `*SIM-PLOT-TIME-LIST*` is obviously essential for interpreting the element data lists properly. For fixed time step simulations, where the time step is given by `*USER-STEP*` (in milliseconds), the element data time base can be reconstructed using both the value of `*USER-STEP*` and `*SAVE-DATA-STEP*`.

In general, to get the time points associated with the current simulation data, whether the simulation is finished or not, use:

current-sim-plot-time-list [*Function*]

Thus, the list of time points returned by this function will always be the ones associated with the result of `ELEMENT-DATA` (see below).

It is often important to analyze simulation data on a specific fixed sampling grid. The basic function for resampling element data (especially if a variable time step was used) is `ELEMENT-DATA-DTED` (see below).

20.2 Basic Handling of Plot Data

The basic functions for controlling and accessing the collection of simulation data are:

enable-element-plot *element* &optional *data−type model−type* [*Function*]

disable-element-plot *element* &optional *data−type model−type* [*Function*]

element-data *element* &optional *data−type model−type state−index* [*Function*]

The first two functions add (or remove) the appropriate element name or names to the appropriate plotting list. ELEMENT-DATA returns the plot data list (in correct time order from the last simulation, according to the list of times in `*SIM-PLOT-TIME-LIST*` - see below). These functions operate on elements in ELEMENT, which can be a list of element names or elements, a single instance of either or NIL, which generates a selection menu. TYPE applies to the element type. DATA-TYPE is used when there is more than one type of plot data for a given sort of element (e.g. current and conductance for a channel or synapse). If DATA-TYPE is not supplied, then the default type of plotted data for a given element type is used. Optionsf are shown in Table 6.

For markov state data, the optional STATE-INDEX integer argument must be supplied. If the DATA-TYPE or TYPE arguments are included, then they are used for all members of the ELEMENT arg. If ELEMENT

Element Type	Default Data Type	Other Data Types
SOMA	'VOLTAGE	'DVDT 'DENDRITE-CURRENT
SEGMENT	'VOLTAGE	'DVDT
AXON	'VOLTAGE	
CHANNEL, SYNAPSE	'CURRENT	'REVERSAL-POTENTIAL 'CONDUCTANCE
ISOURCE, VSOURCE	'CURRENT	
PARTICLE, CONC-PARTICLE	'STATE	'MARKOV-STATES
CONC-INT	'TOTAL	1 (shell 1) 2 (shell 2) 3 (shell 3)
EXTRACELLULAR-ELECTRODE	'FIELD-POTENTIAL	
BUFFER	'CONCENTRATION	
PUMP	'CURRENT	

Table 6: Element data types.

does not refer to a circuit element, if the DATA-TYPE is inconsistent with the referenced element, or if the data was not stored in the last simulation (i.e. was not earmarked for plotting), then `ELEMENT-DATA` returns NIL.

For example,

```
* (ENABLE-ELEMENT-PLOT *soma*)
NIL
```

or

```
* (element-data "Hippo-soma")
'(-70.0 -70.0 -70.0 -69.8123 -69.123 ...)
```

`ENABLE-ELEMENT-PLOT` also sets the enable plot flag for the appropriate class of plotted data.

element-data-dted *element* &optional *(delta−t 1.0) data−type model−type (time−base (current−sim−plot−time−list)) state−index* [*Function*]

Given an element or elements in ELEMENT or element type TYPE, returns a plot data list (or lists for more than one element) of type DATA-TYPE [defaults as is `ELEMENT-DATA`] sampled on an even time base as given by DT [milliseconds]. The time base for the original data is taken from TIME-BASE [default is the simulation time base list given by `*SIM-PLOT-TIME-LIST*`]; if a number, then the time base for the original data is taken from an even time grid with step TIME-BASE.

A string of segments along a path can be plotted by using:

plot-segments-to-soma *element* &optional *(segment−skip 0) clear−first* [*Function*]

Enables plotting on a separate window all the segments on the path from the ELEMENT to the soma, skipping path segments by SEGMENT-SKIP [default 0]. If CLEAR-FIRST [default NIL] is T, then any segments previously including in such a plot are cleared first.

20.3 On-Line Analysis - Creation Of Analysis Results File For Simulations

At the end of a simulation Surf-Hippo can run an analysis of any data that was saved during the simulation, which in turn is determined by the variables referenced in the names stored in the various plot and analysis lists. In the latter case, the appropriate functions to add/delete a node reference are:

enable-element-save-data *element* &optional *data−type model−type* [*Function*]

disable-element-save-data *element* &optional *data−type (abort−disable−if−plotted t) model−type* [*Function*]

This automatic analysis is performed from the function PRINT-ANALYSIS, which is called at the end of the simulation by the function SIM-OUTPUT. The following discussion of PRINT-ANALYSIS follows the version of this function as supplied with Surf-Hippo. For other types of analysis, it is straightforward to modify PRINT-ANALYSIS, and load your own definition:

```
(defun PRINT-ANALYSIS ()...)
```

after Surf-Hippo is loaded (it is probably a good idea *not* to edit the original source file, and rather load your own file that contains the new definition after Surf-Hippo is loaded). For the supplied definition of PRINT-ANALYSIS, the analyzed data is taken from all plotted variables, and the element names specified in

```
*ANALYSIS-NODES*   The nodes listed in this variable will not be plotted.
```

Again, this variable is a list of element names, e.g. the names of segments or somas.

The variable *PRINT-ANALYSIS* determines whether the plotted data is analyzed by the simulator when the simulation is complete by the function PRINT-ANALYSIS. The results of the data analysis at this level can be either printed out to the Lisp window (when *PRINT-OUT-TO-LISP* is T), added to the information file (when *SAVE-SIMULATION-INFO* is T), printed out to the Information Window (when *PRINT-OUT-TO-INFO-WINDOW* is T), and/or saved to a RESULTS_FILENAME.results file (when *DUMP-ANALYSIS-TO-FILE* and *PRINT-OUT-TO-LISP* are T). The RESULTS_FILENAME is made up of the data directory and the value of the string variable *SESSION-NAME*, if *SESSION-NAME* is not the NULL string, or *SIMULATION-NAME*, if *SESSION-NAME* the NULL string.

An example of the results form that is written (appended) to the results file is (for a simulation named J43DSBRANCH2-848441):

```
(push
 '(J43DSBRANCH2-848441
   (DISTAL-PROXIMAL BRANCH-11-5 0.25)
   (((NODE-11-30-54 (MAX . -57.37003) (MIN .-74.23389))
     (NODE-11-30-1 (MAX . -61.630474) (MIN . -73.27865))
     (NODE-11-30-15 (MAX . -59.089767) (MIN . -73.78816))
     (NODE-J43D-SOMA (MAX . -67.22821) (MIN . -71.92825)))
    ((NODE-11-30-54 (AVERAGE . 1.9533609) (BASE . -70.0))
     (NODE-11-30-1 (AVERAGE . 0.5855975) (BASE . -70.0))
     (NODE-11-30-15 (AVERAGE . 1.0426667) (BASE . -70.0))
     (NODE-J43D-SOMA (AVERAGE . 0.17151265) (BASE . -70.0)))))
 *archive-session-results*)
```

The results file will thus contain a form like the above for each simulation that comprises the current session. When this file is later loaded into the Lisp, then the variable *ARCHIVE-SESSION-RESULTS* can be processed as necessary.

As determined by PRINT-ANALYSIS, the basic structure of the list PUSHed onto the variable *ARCHIVE-SESSION-RESULTS* is:

```
(LIST *SIMULATION-NAME* *SIMULATION-RESULTS-ADDENDUM* RESULTS)
```

where the results form is built by consecutive PUSHs of the data returned by functions called by PRINT-ANALYSIS, for example (see INTEGRATE-PLOT-DATA to see the format of what this function returns):

```
(push (INTEGRATE-plot-DATA
       (retrieve-plot-data (list (list *analysis-nodes* 'voltage)))
       *analysis-nodes* *x-integrate-min* *x-integrate-max*)
      results)
```

The variable *SIMULATION-RESULTS-ADDENDUM* should be set to something which can facilitate later parsing of the entries in *ARCHIVE-SESSION-RESULTS*, e.g. some parameters which distinguish the individual simulation. For example, in the example above:

```
(setq *SIMULATION-RESULTS-ADDENDUM* '(DISTAL-PROXIMAL BRANCH-11-5 0.25))
```

In the supplied version, PRINT-ANALYSIS follows the value of various global flags, including *PRINT-LINEAR-ANALYSIS* (which enables the evaluation of INTEGRATE-PLOT-DATA) and *PRINT-NONLINEAR-ANALYSIS* (which enables the evaluation of MAX-MIN-PLOT-DATA). The result lists returned by INTEGRATE-PLOT-DATA and MAX-MIN-PLOT-DATA are collected in PRINT-ANALYSIS and then written to the results file when *DUMP-ANALYSIS-TO-FILE* is T. Again, these formats are determined by the definition of the PRINT-ANALYSIS function, and can be changed as desired.

20.4 Raster Plots

The function:

raster-plots &key *(start* [*Function*]
**user−start−time*) (stop *user−stop−time*) width height title−postfix win title font (event−width *default−raster−event−width*) (event−height *default−raster−event−height*) (times−in−fns t) (max−traces−per−plot *default−max−traces−per−raster−plot*) (raster−spacing *default−raster−spacing*) event−data−lists spiking−elements event−elements (only−event−generators t) (only−elements−with−activity *plot−only−elements−with−activity*) event−element−labels fixed−top−gap fixed−bottom−gap fixed−right−gap fixed−left−gap (include−labels t) default−label (raster−source−label "")*

plots event times of each of the EVENT-ELEMENTS (when ONLY-EVENT-GENERATORS is nil) or the original times of the event generators for the EVENT-ELEMENTS (ONLY-EVENT-GENERATORS T, the default). If not supplied, EVENT-ELEMENTS is taken as all the axons, voltage-controlled and event-driven synapses in the circuit. EVENT-WIDTH is the width of the mark for each event on the plot, in pixels. START-TIME and STP-TIME are in milliseconds. RASTER-SPACING is the vertical spacing between rasters in pixels. When ONLY-ELEMENTS-WITH-ACTIVITY is non-NIL (the default set to the value of *PLOT-ONLY-ELEMENTS-WITH-ACTIVITY*, whose default is NIL), then only elements that have events are plotted.

20.5 Data Folder

The Data Folder allows the storage of (plotted) simulation variables immediately after a simulation run (from the main menu, select the "Immediate information output" option). This folder keeps track of the simulation names and time bases, so that later on the traces may be replotted with reference to their original simulation name (from the main menu, select the "Edit plot parameters" option). This is convenient, for example, for comparing various traces after several simulations. For plotting, data in the folder may be organized according to the simulation run or according to the type of data (determined by the units of the data, e.g. mV).

If the variable *STORE-PLOT-RESULTS-TO-FOLDER* is set, then all the plotted traces will be stored automatically to the data folder after each simulation run.

20.6 Data for Markov Sub-Particles

Gating particles which are defined in order to control some state transition of a Markov gating particle are not fully evaluated in the sense of regular gating particles (Section 14.7). Thus, these "sub-particles" are not tracked by themselves, and although they exist as particle structures, it is not possible at present to store or plot their states.

21 Plot Data and Plotting Windows

Surf-Hippo includes an extensive library of plotting routines for both two and three-dimensional data, some of which have already been mentioned. Very extensive editing of plot windows may be done via the Plot Window Menu (CONTROL-m over the windows), and data can be directly written to Lisp files.

The full documentation for this library is still under construction - in this chapter we shall discuss how two dimensional simulation data is plotted automatically, and the basic use of several plotting functions that may be applied to arbitrary two dimensional data. For mouse and keyboard senstive actions for plot windows, see Appendix D. See also Sections 22, 33 and 25.8.

21.1 Plotting of Simulation Data: Basic Concepts

Many plot parameters relevant for the automatic plotting of simulation data may be set in the "Setting Up Plot Parameters" menu (choose "Modify plot parameters" from the Surf-Hippo main menu).

The data in a given plot is organized as a group of sets of traces, where each set is typically from a different simulation run. The default operation for plotting is that if the current plotting windows are to be reused, then the results from a new simulation replace the old data (see below Section 21.3.3).

21.1.1 Simulation Time Steps vs. Saved Data Time Steps: `*SAVE-DATA-STEP*`

Simulation data is saved onto lists every `*SAVE-DATA-STEP*` time steps (default 2, i.e. data is saved every other time step). The only exception to this rule is that the last time step of the simulation is always saved. The variable `*SAVE-DATA-STEP*` may be set explicitly, of course:

```
(setq *save-data-step* 1)
```

or in the "Setting Up Some More Plot Parameters" menu (choose "Change some more plot details?" in the "Setting Up Plot Parameters" menu), in the "Time Step and Numerical Integration Parameters" menu (from the "Overall parameters" menu), or in the "Modify Simulator Global Variables" menu. See also Section 29.

21.2 User Plot Functions

Clearly there will be cases in which you need to plot data independently from the automatic simulation plotting routines. There are several related functions for data plotting XY data. These include:

- `PLOT-TIMED-DATA`
- `PLOT-XY-DATA`
- `PLOT-POINTS`
- `PLOT-SCATTER`

The main difference between these functions is the format of the data that you supply to them. Each function accepts a large variety of arguments, including those for setting up various layout parameters. Here we will describe only a few of the more common parameters.

21.2.1 The `PLOT-TIMED-DATA` Function for Data Plots

The function `PLOT-TIMED-DATA` is primarily aimed at plotting time series data, in particular when several sequences share the same time reference ("time" of course can refer to some other variant):

plot-timed-data *data−sequences* &optional *label−list (time−base 1.0)* &key ... [*Function*]

The first three arguments to `PLOT-TIMED-DATA` are the most important: DATA-SEQUENCES is required, and LABEL-LIST and TIME-BASE (default 1.0) are optional. DATA-SEQUENCES consists of one or more sets of Y data point sequences, where each sequence may be either a list or an array of Y points. For example, to plot out a single list of numbers:

```
(plot-timed-data '(1 0 2 1 0 -1 -1 2))
```

To plot out multiple sequences in the same window, they are collected in a list:

```
(plot-timed-data (list (sinewave 1 15 0.2 :step 1) `(1 0 2 1 0 -1 -1 2)))
```

Note that numeric arrays and lists may be mixed together.

The LABEL-LIST argument PLOT-TIMED-DATA is a single label or a list of labels, corresponding to the number of sequences in DATA-SEQUENCES. The components of LABEL-LIST can be either symbols, strings or numbers:

```
(plot-timed-data '(1 0 2 1 0 -1 2) 'foo)

(plot-timed-data (list (sinewave 1 15 0.2 :step 1) `(1 0 2 1 0 -1 2)) '("Sin Wave" foo))
```

The format for the optional TIME-BASE argument is similar to DATA-SEQUENCES, where there may be a set of X data point sequences that are matched up with the Y data point sequences in DATA-SEQUENCES:

```
(plot-timed-data '(1 0 2 1 0 -1 -1 2) 'foo '(1 0 1 2 1 2 1))

(plot-timed-data '((3 2 1 3 4 2 2)(1 0 2 1 0 -1 2))
                 '(bar foo)
                 '((1 0 1 2 1 2 1) (0 1 0 2 1 0 1)))
```

If there is a single TIME-BASE sequence and multiple DATA-SEQUENCES, then the TIME-BASE sequence will provide the common X reference for all the Y data. Also, if TIME-BASE is a single number, then this is used as the common X increment (e.g. Δt) between all Y data points.

```
(plot-timed-data '((3 2 1 3 4 2 2)(1 0 2 1 0 -1 2))
                 '(bar foo)
                 5)
```

Just as an illustration, here is how you could plot out saved simulation data for some set of circuit elements (in this case segment voltages). Here we are assuming that the original simulation used a variable time step, and we want to plot the data resampled on a regular time grid of 0.1 milliseconds. Also, we assume that the segments in the circuit have been flagged (e.g. with ENABLE-ELEMENT-PLOT) to save their data:

```
(plot-timed-data (element-data-dted 'segment 0.1) (element-name 'segment) 0.1)
```

Recall that if the ELEMENT argument to ELEMENT-DATA-DTED refers to more than one element, then a list of data lists is returned (similarly for ELEMENT-NAME). The ability of PLOT-TIMED-DATA to handle not only single data sequences, but lists of data sequences, facilitates this operation.

21.2.2 The PLOT-XY-DATA Function for Data Plots

PLOT-XY-DATA is an alternate to PLOT-TIMED-DATA, when plotting more than one trace and there is a specific X sequence matched to a specific Y sequence:

plot-xy-data *xy−data−lists* &optional *label−list* &key ... [*Function*]

The required XY-DATA-LISTS is either a list of an X sequence and a Y sequence:

```
'((x x ... x)(y y ... y))
```

or a list of such lists:

```
'(((x1 x1 ... x1)(y1 y1 ... y1))
  ((x2 x2 ... x2)(y2 y2 ... y2))
   ...
  ((xn xn ... xn)(yn yn ... yn)))
```

LABEL-LIST can either be a single label (for one data set) or a list corresponding to the number of data sets in XY-DATA-LISTS. Thus:

```
(plot-xy-data '(((1 2 3 4)(1 0 1 0))))

(plot-xy-data '(((1 2 3 4)(1 0 1 0))((10 11 13 14)(1 0 1 0))) '(foo bar))
```

21.2.3 The PLOT-POINTS Function for Data Plots

PLOT-POINTS is used for plotting explicit pairs of XY points:

plot-points *list−of−point−lists* &optional *label−list* &key ... [*Function*]

The most important arguments are the required LIST-OF-POINT-LISTS argument and the optional LABEL-LIST argument. LIST-OF-POINT-LISTS can either be a list of XY pairs, for plotting out one set of data:

```
'((x y)(x y)(x y)(x y)...)
```

or a list of lists of XY pairs for more than one set:

```
'(((x1 y1)(x1 y1)...(x1 y1)) ((x2 y2)(x2 y2)...(x2 y2)) ...)
```

LABEL-LIST can either be a single label (for one data set) or a list corresponding to the number of data sets in LIST-OF-POINT-LISTS. Thus:

```
(plot-points '((1 2)(2 1)(3 4)) 'foo)

(plot-points '(((1 2)(2 1)(3 4)) ((2 5)(5 7))) '(foo bar))
```

21.2.4 The PLOT-SCATTER Function for Data Plots

PLOT-SCATTER is essentially identical to PLOT-POINTS, except that the default is to plot the points as a scatter plot, instead of connecting them:

plot-scatter *list−of−point−lists* &optional *label−list* &key ... [*Function*]

For example:

```
(plot-scatter '(((1 2)(2 1)(3 4)) ((2 5)(5 7))) '(foo bar))
```

21.2.5 User Plot Functions: Typical Keyword Arguments

As mentioned, the plot functions support a variety of keyword arguments. Some of the more common ones include:

- :X-MIN, :Y-MIN The minimum plotted X or Y values
- :X-MAX, :Y-MAX The maximum plotted X or Y values
- :X-INC, :Y-INC The tick mark increment along the X and Y axes
- :X-ORIGIN, :Y-ORIGIN The Y intercept on X axis, and the X intercept on Y axis, respectively
- :X-LABEL, :Y-LABEL The labels for the X and Y axes
- :X-ARE-FNS, :Y-ARE-FNS The tick marks are constrained to be integers
- :SCATTER Mark each point with a scatter symbol

- :CONNECT-DATA-POINTS Connect each successive point
- :TITLE Title of the plot window

These keyword arguments go after any required and optional arguments, for example:

```
(plot-xy-data '(((1 2 3 4)(1 0 1 0))((10 11 13 14)(-1 0 -1 0)))
              '(foo 232)
              :x-max 16 :y-min -2 :y-max 2 :y-inc .5
              :x-are-fns t :scatter t :x-min 0 :y-origin -2
              :title "Foo Bar Baz" :x-label "Kilometers" :y-label "Grams")
```

21.2.6 Trace Ordering

As implied above, the order of the traces in a plot is determined by the order of the data and label arguments to the plot functions. The order of the traces may also be set via the plotting menu sequence.

21.2.7 Keyword Arguments for Overall Plot Layout

In the default case, the plotting functions arrange the layout of the plot window in a reasonable way, generally maximizing the available space and allowing room for labels, etc. For various reasons, it may be preferable to set constraints on the layout. The gaps between the area in which the data appears and the window borders may be set with the following key arguments to `PLOT-TIMED-DATA`, `PLOT-POINTS`, `PLOT-SCATTER`, and `PLOT-XY-DATA`:

```
:UPDATE-FIXED-GAP-PARAMETERS

:USE-FIXED-TOP-GAP
:FIXED-TOP-GAP     [default 0 pixels]

:USE-FIXED-BOTTOM-GAP
:FIXED-BOTTOM-GAP  [default 0 pixels]

:USE-FIXED-RIGHT-GAP
:FIXED-RIGHT-GAP   [default 0 pixels]

:USE-FIXED-LEFT-GAP
:FIXED-LEFT-GAP    [default 0 pixels]
```

The fixed gaps may be negative. The keyword `:UPDATE-FIXED-GAP-PARAMETERS` must be T in order for the other arguments to be considered. These parameters may also be set via the plot window menus (select "Overlay and layout specifications").

21.3 Global Variables Affecting Plotting

Several global variables affect properties of data plots. We shall now discuss the more important ones.

21.3.1 Number of Traces per Plot

The global variable `*TRACES-PER-PLOT*` (default value 6) constrains the number of traces per plot window. If set to 0, then there will be no limit to the number of traces per plot. This may be set with the top level plot parameters menus.

21.3.2 Creating New Windows

When a new circuit is loaded, or if the global variable *CREATE-NEW-PLOT-WINDOWS* is T, then plot output will go to new windows, preserving any previous ones. A new window will also be created if an existing window to which the data would have been presented has been locked (CONTROL-L). A related variable is *CREATE-NEW-SIMULATION-PLOTS* , which makes sure to create a new set of plot windows for the next simulation.

21.3.3 Overlaying Data

If *CREATE-NEW-PLOT-WINDOWS* in NIL and the global variable *OVERLAY-ALL-PLOTS* is T (default nil), then new data is drawn over the old data, and the data is added to the plot windows. As a result subsequent zooming or unzooming applies to all the overlaid data. If the global variable *ACCOMODATE-ALL-OVERLAYS* is set (directly or via the "Setting Up Plot Parameters" menu), then the coordinates of the overlaid windows will be adjusted so that all overlaid data will be contained within the windows.

If the variable *PRESERVE-PLOT-LAYOUT* is set (default nil), then any new data sent to a plot window will be displayed without changing the scaling of the window.

21.3.4 Waterfall Plots

Waterfall plots may be generated with the plot window menu ("Edit miscellaneous parameters" option). The scale bar lengths in waterfall plot mode are set by the X and Y axis intervals. Offsets between the trace are set by the menu. Trace labels may be placed in the upper left corner, or adjacent to each trace on the right.

You may need to experiment a bit with various plot window parameters to get the proper layout. With the plot menu you can specify automatic waterfall layout, which may be satisfactory in some cases. In this case, some relevant global variables include:

```
*DEFAULT-Y-PLOT-TOP-GAP-EXTRA-WATERFALL*  -  Integer, in pixels
*X-TRACE-OFFSET*                          -  Float, in units of the data
*X-PLOT-LEFT-GAP-WATERFALL*               -  Integer, in pixels
*AUTO-WATERFALL-Y-TRACE-OVERLAP*          -  Between 0.0 and 1.0
```

Note that zooming does not work with waterfall plots. Also, the absolute coordinate values from the center mouse will be relative to the the first trace in the plot - the dy/dx values will be unchanged from the conventional plot.

Depending on the relative amplitudes of the waterfall traces, there is an alternative method for determining the plot window layout. From the "Waterfall Plot Parameters" menu (reached from the general edit plot menu) try enabling the "Use WATERFALL-Y-DATA-MAX" option, with different values of WATERFALL-Y-DATA-MAX (normally, 0, in units of the plotted data).

21.3.5 Window Comments

There are two categories of global comments that will be automatically to plot windows. For all plots generated by a circuit simulation, when the global variable

simulation-plot-window-comment *nil* [*Variable*]

is a string this comment will be added at the position given by the global variable

simulation-plot-window-comment-position *:lower−right* [*Variable*]

whose possible values given given by the variable *COMMENT-POSITIONS*. More generally, if the global variable

global-plot-comment *nil* [*Variable*]

is set to a string, this comment is added to any window produced by the plot functions at the position given by

global-plot-comment-position *nil* [*Variable*]

whose default value is given by the variable `*DEFAULT-COMMENT-POSITION*`).

21.4 Writing Plot Data to Files

The XY data of plotted traces may be written to ASCII files either by the plot window menu, or by directly calling:

grab-and-store-plot-data &key *filename win (trace :all) overlay−index force−menu (output−format nil output−format−supplied−p) suppress−comments include−which−comments (if−file−exists :supersede)* [*Function*]

In either case you have the option of storing the data in a Lisp or a columns format, in the case of the function call by the :OUTPUT-FORMAT keyword:

```
   :OUTPUT-FORMAT key    File format
-----------------------------------------
     :LISP [default]   A list of lists - ((x1 x2 ... xn)(y1 y2 ... yn))
     :COLUMNS          Two columns -
                          x1   y1
                          x2   y2
                             .
                             .
                             .
                          xn   yn
```

21.5 Miscellaneous

21.5.1 Re-Sizing Windows

After a plot window is resized with the window manager and mouse, the data can be redrawn with the R (replot) command, which will keep the current parameters, or CONTROL-RIGHT (restore) command, which will restore the original plot parameters. Otherwise, call up the main plot window menu with CONTROL-m and hit OK. This will make sure that the plot parameters are maintained from the previous window sizing.

21.5.2 Data Axes

Axes may be standard (abscissa and ordinate extending over the entire plot) or simple (horizontal and vertical bars, lengths according to the specified axes' increments, and located according to the plotting menu, default upper right hand corner).

21.5.3 Log Plots

Using the plot window menu, either the X or Y axes may be plotted on a logarithmic scale, with either the natural base or other, as specified. If an attempt is made to take the logarithm of an non-positive number, then an error is signaled, and you have the option of specifying an appropriate offset that allows the log.

The log operation is only applied to the data within the range specified by the X and Y minimum and maximum parameters (if defined).

21.5.4 Refreshing Plots

For the function REFRESH-ALL-PLOTS, the optional GRID argument can be :DRAW, :ERASE or nil (don't change). These functions operate on :STANDARD-PLOT windows, that is those produced by PLOT-TIMED-DATA, PLOT-XY-DATA, PLOT-SCATTER, PLOT-POINTS:

refresh-all-plots &optional *grid* [*Function*]

refresh-plot *win* &optional *(grid *refresh−plot−default−grid*)* [*Function*]

21.5.5 Hints for Plot Layout

Sometimes, axis limits or tick marks aren't quite right because of quantization error in the calculations for laying out the plot. Try twiddling various layout/axis parameters (e.g. changing a "maximum value" from 2.0 to 2.0001) to correct this.

21.5.6 Munged Dashed Lines

Dashed line styles (particularly for thin lines) do not display properly when there is a high density of points. This problem may be due to either a Garnet or CLX bug. A temporary workaround is a plot window slot :PLOT-POINT-SKIP, which may be either set directly or as a key argument to PLOT-TIMED-DATA, or via the plot window menu (currently via the "Overlay and layout specification and flag menu" option in plot window menu sequence). This parameter should be set so that an oversampled plot will only show every other N points, where N is given by the window's :PLOT-POINT-SKIP slot.

21.5.7 Plot Related Functions and Variables

USE-SIMULATION-NAME-FOR-SIMULATION-PLOT-TITLES

21.5.8 Data Folder

The Data Folder allows the storage of (plotted) simulation variables immediately after a simulation run (from the main menu, select the "Information management" option). This folder keeps track of the simulation names and time bases, so that later on the traces may be replotted with reference to their original simulation name (from the main menu, select the "Edit plot parameters" option). This is convenient, for example, for comparing various traces after several simulations. For plotting, data in the folder may be organized according to the simulation run or according to the type of data (determined by the units of the data, e.g. mV).

If the variable *STORE-PLOT-RESULTS-TO-FOLDER* is set, then all the plotted traces will be stored automatically to the data folder after each simulation run.

22 Analysis of Data

This section describes techniques for the analysis and display of Surf-Hippo data, both integral to the execution of simulations, for examination of data after a simulation is complete, and for later analysis of stored data. See also Sections 21 and 20.

22.1 Linear Analysis Functions

The following functions operate on lists of numbers, either implicitely via reference to a given element's data, on explicitely as an argument. Most are defined in the `waveforms.lisp` file.

element-integrated-data *element* &optional *data−type model−type* [*Function*]

According to the plot data and time points of the last simulation, returns the sum of the integrals of the data of type DATA-TYPE of each element in ELEMENT, of element type TYPE, where the default DATA-TYPE is given in the documentation for the ELEMENT-DATA function. ELEMENT can either be a single element or a list of elements.

integrate-wave *wave* &optional *(delta−t 1.0) (x−0 0.0)* [*Function*]

Given WAVE, an array or list of numbers assumed to be spaced evenly by DELTA-T with respect to the independent variable, returns a list which is the cumulative integral of WAVE, with the initial conditions given by the optional argument X-0.

differentiate-wave *wave* &optional *(time−spec 1.0)* [*Function*]

Returns a list which is a differentiated WAVE, an array or list of numbers which is assumed to be spaced evenly with respect to the independent variable, with a grid of DELTA-T. Note that the length of the result is one less than the length of WAVE. Thus, given a n-valued sequence WAVE with values:

```
[x1 x2 x3 ... xn]
```

returns an (n-1)-valued list with values:

```
[(x2-x1)/DELTA-T, (x3-x2)/DELTA-T, ... (x(i+1)-xi)/DELTA-T, ... (xn-x(n-1))/DELTA-T]
```

Also of interest:

element-data-dft *element* &key *data−type type state−index (delta−t 1.0) (reference−time−list (current−sim−plot−time−list)) (dc−offset 0.0)* [*Function*]

for plotting the frequency magnitude and phase of the data appropriate to ELEMENT (see arguments for the ELEMENT-DATA-DTED function).

22.2 Non-linear Analysis Functions

The following set of functions are based on the function:

data-extreme &key *(min−time 0.0) (max−time *user−stop−time*) dt maxp (what :value) data−list (time−list (current−sim−plot−time−list))* [*Function*]

which provides an extremum analysis of DATA-LIST, considered with respect to a time base of step DT (ms), if supplied, otherwise from times in TIME-LIST. The maximum (respectively minimum), depending on MAXP of WHAT (:SLOPE, 1ST-DERIVATIVE (same as :SLOPE), :2ND-DERIVATIVE, :VALUE), within a time window between MIN-TIME (ms) and MAX-TIME (ms). The function returns as values the extreme value and the time for which that value was detected. If no extreme was detected, then returns as values 0.0 and MIN-TIME. Data units are as appropriate for the type of data.

The function:

element-extreme *element* &key *data−type (min−time 0.0) (max−time *user−stop−time*) dt maxp (what :value) data−list model−type (time−list (current−sim−plot−time−list))* [*Function*]

calls DATA-EXTREME, where the data is (typically) provided by data of DATA-TYPE of ELEMENT of ELEMENT-TYPE.

The function:

element-amplitude *element* &key *data−type (min−time 0.0) (max−time *user−stop−time*) dt (time−list (current−sim−plot−time−list)) data−list model−type negative−p base−level* [*Function*]

and the functions ELEMENT-MIN, ELEMENT-MAX, ELEMENT-MIN-SLOPE and ELEMENT-MAX-SLOPE (which have similar arguments as ELEMENT-AMPLITUDE), are all based on ELEMENT-EXTREME. In addition, the reference level for the quantity of interest is given by BASE-LEVEL (assumed to be in the units corresponding to that of the data) if supplied, otherwise the reference is taken as the minimum (respectively maximum) when NEGATIVE-P is NIL, (respectively T). The measured event characteristic is either the maximum or minimum value thereafter, again depending on NEGATIVE-P.

The function:

data-amplitude &key *(min−time 0.0) (max−time *user−stop−time*) dt (time−list (current−sim−plot−time−list)) data−list negative−p base−level* [*Function*]

and the functions DATA-MIN, DATA-MAX, DATA-MIN-SLOPE and DATA-MAX-SLOPE (which have similar arguments as DATA-AMPLITUDE), are companion functions to the ELEMENT-based functions above, and may be used when you have an explicit data list. This can be more efficient if, for example, various analyses are to be done on the same sampled simulation data. In this case, it is advantageous to first derive and assign to a local variable a sampled version of some element's data, and then pass that list to succesive calls of the desired analysis functions.

22.3 Event Detection and Removal

The basic functions for detecting events are:

list-maxs *wave* &optional *(delta−t 1.0) (max 0.0) (min−max−time 0.0)* [*Function*]

list-mins *wave* &optional *(delta−t 1.0) (min 0.0) (min−min−time 0.0)* [*Function*]

These two functions find event epochs for the data in WAVE, with time steps DELTA-T. Both return two lists as values, the MAX (respectively MIN) positive and negative-going (respectively negative and positive-going) crossing times, whenever the duration framing a particular positive-negative pair of MAX (respectively negative-positive pair of MIN) crossings is greater than MIN-MAX-TIME (respectively MIN-MIN-TIME).

frame-min-maxs *wave max−min−wave* &optional *(delta−t 1.0) (max 0.0) (min 0.0) (min−min−max−time 0.0) messages* [*Function*]

This function strips epochs in WAVE (time step of DELTA-T) according to analysis applied to MAX-MIN-WAVE, and returns the processed wave. Epochs are detected by applying LIST-MINS and LIST-MAXS to WAVE, using MAX and MIN, respectively, and MIN-MIN-MAX-TIME as for the MIN-MIN-TIME and MIN-MAX-TIME arguments, respectively. Once an epoch is detected, the output data is held constant until the end of a given epoch. For example, events in a voltage waveform could be stripped based on extrema in the voltage derivative:

1. Get a list of data into WAVE with some DELTA-T

2. Get a DVDT with `(DIFFERENTIATE-WAVE WAVE DELTA-T)`.
3. Look at the derivative `(plot-timed-data dvdt nil nil :delta-t delta-t)` to get an idea of the "events" that you are looking for.
4. Call FRAME-MIN-MAXS with appropriate values for the max and min DVDT values.

A related function is:

find-zero-crossings *wave* &optional *(delta−t 1.0) (min−difference−from−0 0.0)* [*Function*]

where for data in WAVE with time step DELTA-T, a list of the zero-crossing times is returned.

22.4 Spike Detection

These functions are specialized for spike detection:

element-spike-times *element* &key *(spike−threshold −20.0) sub−threshold−time (supra−threshold−duration−min 0.0) model−type data−list (start−time 0.0) (time−base (current−sim−plot−time−list))* [*Function*]

Returns a list of detected spikes from the voltage of the soma or segment associated with ELEMENT of TYPE, according to the SPIKE-THRESHOLD (mV), SUPRA-THRESHOLD-DURATION-MIN, SUB-THRESHOLD-TIME. The voltage trace from ELEMENT is resampled at a time step DT with reference to REFERENCE-TIME-LIST. All times are in milliseconds.

element-firing-frequency *element* &key *(spike−threshold −20.0) (supra−threshold−duration−min 0.1) sub−threshold−time model−type data−list (time−base (current−sim−plot−time−list)) (start−time 0) (end−time *user−stop−time*)* [*Function*]

Returns the firing frequency in Hz from spikes detected from the voltage of the soma or segment associated with ELEMENT, between START-TIME (default 0ms) and END-TIME (default *USER-STOP-TIME*). Also takes key word arguments for spike detection as used in the function ELEMENT-SPIKE-TIMES.

22.5 Phase Plots

phase-plots *element−pairs* &key *title y−label x−label x−min y−min x−max y−max (prompt−for−overlay t)* [*Function*]

The argument ELEMENT-PAIRS is one of the following:

```
(element-1 element-2)
```

Where the data types for each element are defaults from:

```
(typecase element
         ((or axon segment soma node) 'voltage)
          ((or channel synapse isource vsource) 'current)
          ((or particle conc-particle) 'state)
          (conc-int 'concentration-1))
```

Or data types may be specified for any of the elements:

```
((element-1 data-type) element-2)
 (element-1 (element-2 data-type))
 ((element-1 data-type) (element-2 data-type))
```

Or a list of element pairs (with optional data types) may be given:

```
(((element-1 data-type) element-2)
  (element-3 element-4)
  (element-4 (element-5 data-type)) ... )
```

For example:

```
(PHASE-PLOTS `((("Hippo-soma-CA-IN-GEN" concentration-2) ,*soma*)
               ("Hippo-soma-CA-IN-GEN" ,*soma*)))
```

Note that in this example, the global variable *SOMA* is referenced. Using list construction with the backquote notation (`), the comma before *SOMA* allows it to be evaluated. This could have been done with the name of the *SOMA*:

```
(PHASE-PLOTS `((("Hippo-soma-CA-IN-GEN" concentration-2) "Hippo-soma")
                ("Hippo-soma-CA-IN-GEN" "Hippo-soma")))
```

Another example:

```
(PHASE-PLOTS `(("Hippo-soma-NA-RF2-NAM-RF2" "Hippo-soma-NA-RF2-NAH-RF2")
               ("Hippo-soma-NA-RF1-NAM-RF1" "Hippo-soma-NA-RF1-NAH-RF1"))
             :title "Na Particle States"
             :x-label "State" :x-max 1.0 :x-min 0.0
             :y-label "mV" :y-max 40 :y-min -80)
```

Note that a given element data type must have been enabled for plotting, for example with the menus or with the function ENABLE-ELEMENT-PLOT described in Section 20.2.

22.6 Plotting Archived Data

As described in Section 23, simulation data that has been archived to .dat files may be later reloaded and plotted via the menus. While this interface may be improved in later releases, the following describes how you may combine traces from different simulations in a single plot by entering plotting functions directly into the Lisp interpreter.

We will assume that three simulations were run previously, named AXON-298604580, AXON-298604630, AXON-298604648, and each were saved (see Section 32 for how simulations are named). We shall also assume that for each simulation a soma voltage and a current source were plotted. Remember that in order to save (archive) a simulation, you must do so before the next simulation run. In other words, the memory locations for the simulation data are overwritten with every subsequent simulation. Also, note that only circuit data that is plotted may be subsequently archived.

Now, after these simulations, or during a subsequent Surf-Hippo session (and as described in the file.doc section mentioned above), these archives may be loaded either from the menus or by entering LOAD commands to Lisp directly. For example, in the latter case (assuming that the directory "data" is a subdirectory of the current working directory - otherwise the full pathname is needed):

```
* (load "data/axon/8_16_1994/AXON-298604580.dat")
```

As described in Section 23, loaded archive data may be plotted out with the menus:

```
[Main Simulation Menu:<Modify plot parameters - plot loaded archive
data><Plot loaded archive data>[Choose Archived Simulation]]
```

Otherwise, enter the Lisp interpreter (QUIT from the Main Menu), and use the PLOT-TIMED-DATA and PLOT-XY-DATA functions to plot out the archive lists whose names are given by evaluating *ARCHIVE-VARIABLE-LIST*. For example, after the three archives are loaded (assuming nothing else was loaded from the archive), evaluating *ARCHIVE-VARIABLE-LIST* shows what is available:

```
* *ARCHIVE-VARIABLE-LIST*
  ((AXON-298604580
    AXON-298604580-TIME
    ((AXON-298604580-MARCH-SOMA-ISRC-ISOURCE-CURRENT-DATA ISOURCE-CURRENT-DATA)
     (AXON-298604580-MARCH-SOMA-NODE-VOLTAGE-DATA NODE-VOLTAGE-DATA)))
   (AXON-298604630
    AXON-298604630-TIME
    ((AXON-298604630-MARCH-SOMA-ISRC-ISOURCE-CURRENT-DATA ISOURCE-CURRENT-DATA)
     (AXON-298604630-MARCH-SOMA-NODE-VOLTAGE-DATA NODE-VOLTAGE-DATA)))
   (AXON-298604648
    AXON-298604648-TIME
    ((AXON-298604648-MARCH-SOMA-ISRC-ISOURCE-CURRENT-DATA ISOURCE-CURRENT-DATA)
     (AXON-298604648-MARCH-SOMA-NODE-VOLTAGE-DATA NODE-VOLTAGE-DATA))))
```

The list contains three lists, one for each simulation. The CAR of each simulation list is a symbol with the name of the simulation (e.g. AXON-298604580). The second component of each simulation list is a symbol which points to a list of time points (e.g. AXON-298604580-TIME). The last component of each simulation list is a list of lists, each of which is the cons of a symbol pointing to a data list (e.g. AXON-298604580-MARCH-SOMA-ISRC-ISOURCE-CURRENT-DATA) and a symbol (e.g. ISOURCE-CURRENT-DATA) which describes the type of data (although this is often apparent from the the data list symbol itself). For example, if we evaluate these symbols directly:

```
* AXON-298604580-TIME
  (1399.402 1398.402 1397.402 1396.402
               .
               .
               .
   .0103995 .0001147 0.0)
* AXON-298604580-MARCH-SOMA-ISRC-ISOURCE-CURRENT-DATA
  (0.200 0.200 0.200 0.200 0.200 0.200
               .
               .
               .
   0.0 0.0 0.0 0.0 0.0 0.0 0.0)
*
```

Note that the time list is in reverse order - however, the archived circuit data is also stored in reverse order, so there is always a direct match between the time list and the data lists in a given .dat file.

In general, the `PLOT-TIMED-DATA` function is more convenient when making a plot of one or more traces from the same simulation:

```
* (plot-timed-data AXON-298604580-MARCH-SOMA-NODE-VOLTAGE-DATA
                   ‘‘Soma Voltage, Control" AXON-298604580-TIME)
```

`PLOT-XY-DATA` must be used, however, when traces from different simulations are plotted, so that each data list may be matched with the proper time sequence:

```
* (plot-xy-data (list (list AXON-298604580-TIME AXON-298604580-MARCH-SOMA-NODE-VOLTAGE-DATA)
                      (list AXON-298604630-TIME AXON-298604630-MARCH-SOMA-NODE-VOLTAGE-DATA)
                      (list AXON-298604648-TIME AXON-298604648-MARCH-SOMA-NODE-VOLTAGE-DATA))
                (list "Soma Voltage, Control"
                      "Soma Voltage, Na Blocked"
                      "Soma Voltage, K Blocked"))
```

22.7 Sessions of Simulations

The session organization allows you to run a sequence of simulations with all the results (as generated by the call to `PRINT-ANALYSIS`, see below) being sent (appended) to the same *.results file. When `*SESSION-NAME*` is redefined, then the next call to `PRINT-ANALYSIS` (assuming that `*DUMP-ANALYSIS-TO-FILE*` and `*PRINT-OUT-TO-LISP*` are T) will write the results to a new *.results file, determined by the new value of `*SESSION-NAME*`.

The advantage of the *.results file format is that these files may be loaded directly into the Lisp, and the data manipulated more readily than is the case if you need to search and edit through the *.info files (which may have the same data, but in a more readable form).

The function `(WRITE-COMMENT-TO-ANALYSIS-FILE comment-string)` will write (append) its argument to the current *.results file.

22.8 Others

For organizing the way data is grouped, the following global variables may be useful (these may be set from the plot details menu):

```
*GROUP-PLOT-DATA-BY-CELL* [default T]
*GROUP-PLOT-DATA-BY-CELL-TYPE* [default T]

*PLOT-SYNAPSES-BY-MAJOR-ION* [default nil]
*PLOT-CHANNELS-BY-MAJOR-ION* [default nil]
*PLOT-CURRENTS-BY-MAJOR-ION* [default nil]
```

Depending on these variables, for a given type of data (e.g. voltage, conductance, current), all traces associated with each cell or cell type in the circuit, for the first two, or with a given ion, for the last three, are displayed in their own window(s).

23 Data and Information Files

There are several opportunities to write out either data files (simulation plot data or circuit element descriptions) or simulation information files.

The general level of detail for information output about the circuit is determined by the global variable:

simulation-print-detail *:terse* [*Variable*]

which may be set to :NONE, :TERSE, :MEDIUM, :FULL, :FULL_WITH_SEGMENTS. The destination of this information may be set from the Main Menu (select "Information Management").

The advantage of output to the Lisp window is efficiency, and if this is ILisp or other editor-based window, then the output can be readily edited. The advantage of the Information Window is that this can be hardcopied in the same manner as the Plotting windows. From within the Information Window (as well as Histology or Plotting Windows), Control-p will prompt for printing the selected window (and all others). The Info windows scroll, and may be resized by hand: the printed version includes only the part of the window that is visible on the screen.

You can also generate immediate information output either via the "Information management" or "Print out basic info" options from the Main Menu. The latter choice prints out a simple summary of the circuit to the Lisp window, and is convenient for quick checks of the circuit. Note that the time printed out and the simulation name is only changed when a simulation is actually run.

23.1 Documenting User Variables

(See also Section 4.3.) When the global variable `*DOCUMENTED-USER-VARIABLES*` is set to a list of globally bound symbols (i.e. global variables), the `PRINT-CIRCUIT` function will print out the symbols and their values both at the beginning of simulations, and as part of any info files written by the simulator (unless the optional argument to `PRINT-CIRCUIT` is :TERSE or :NONE). This is useful when you have defined your own set of global variables, and you wish to have an automatic documentation of their values. For example, if the user set:

```
(setq *DOCUMENTED-USER-VARIABLES* '(*NEURONS-PER-LAYER* *somatic-cxns* *maximum-gabab-density*))
```

then the simulation information might include:

```
(setq *NEURONS-PER-LAYER* 103
      *SOMATIC-CXNS* 4
      *MAXIMUM-GABAB-DENSITY* 2093)
```

depending on whatever value these variables happened to have. The `SETQ` format allows simple inclusion of this information in another Lisp file. Note that `*DOCUMENTED-USER-VARIABLES*` is cleared when a new circuit is loaded. Note as well in this example, all the global variable symbols are framed by `*`, just by convention.

If the global variable `*DOCUMENT-ALL-NEW-VARIABLES*` is T [default NIL], `PRINT-CIRCUIT` will print out *any* global variables that were defined after the initialization of Surf-Hippo in the `SURF` package or in `*DOCUMENTED-USER-VARIABLES*`.

If you want these values to be automatically written to a file at the end of a simulation, remember to use:

dump-documented-user-variables-file [*Function*]

23.2 Automatic Simulation File Creation

Automatic information and data file or window output can be enabled for each simulation via menus starting with the Information management option in the Main Menu. Surf-Hippo will dump simulation/circuit information to either an "Information Window" or the Lisp Listener. The advantage of output to a Lisp

Listener is efficiency, and if Ilisp or other editor-based window is used, then the output can be readily edited. The advantage of the Information Window is that it can be hardcopied in the same manner as the Plotting or Histology windows. The options in this series of menus control the documentation generated either immediately or with each and every simulation run.

23.3 Editing Lisp Files

You may want to edit a Lisp file that was created by Surf-Hippo just to add some comments, i.e. without really messing with the Lisp code or syntax. There are two ways that comments are delineated. First, anytime a semi-colon appears in a line, then the remainder of that line is a comment, e.g.:

```
(some lisp code)  ; This is the comment
```

A more convenient method for lengthy comments is to bracket the text with "`#|`" and "`|#`", as follows:

```
...
(lisp code)
(some more lisp code)
#|
The sharp sign followed by the vertical bar starts text that will be ignored
by Lisp when the file is read. The text ends with a vertical bar followed by a
sharp sign, like this:
|#
(more lisp code)
(and some more lisp code)
...
```

23.4 Loading Data, Circuit Description and Element Description Files

Data files are written as ASCII text which can be loaded into Lisp as is. Surf-Hippo files can also be compiled to save space and loading time. Whenever simulation data is written, the data file concludes with a Lisp statement which updates the global variable `*ARCHIVE-VARIABLE-LIST*` (when the archive file is loaded back into the same (or another) Surf-Hippo session). As the name suggests, `*ARCHIVE-VARIABLE-LIST*` keeps track of the loaded archived data.

Data, circuit element, and circuit definition Lisp files may all be loaded via the file loading menu (follow links from the main menu), or by temporarilly quitting Surf-Hippo (don't quit Lisp), and using the Lisp function `LOAD`, e.g. (assuming that the directory "data" is a subdirectory of the current working directory - otherwise the full pathname is needed):

```
* (load "data/basic-hippo/1_30_1994/basic-hippo-296895207.dat")
```

Note that information files (with file extension .info) are *not* Lisp files and cannot be loaded into Lisp: there are for reading by humans only.

Loaded archive data may be plotted out from the plotting menu series. Note that the archive option in the main menu only shows up if archived data has been loaded into Surf-Hippo.

Lisp files may also be loaded from the "Overall parameters, load circuit or other files" option of the Main menu. See the discussion on saving loadable Type-Def files in Section 11.

23.5 Random State Reference Files

For simulations that use the `RANDOM` function for generating a pseudo-random sequence, the seed may be initialized by calling `GET-REFERENCE-RANDOM-STATE`. This reads in a file that is created by calling the function `SAVE-REFERENCE-RANDOM-STATE`. The seed file referenced by both of these functions is /surf-hippo/random-state by default; you can supply an alternative filename as an optional argument is desired.

Note that for consistency, `SAVE-REFERENCE-RANDOM-STATE` should only be called once at a given site. The Surf-Hippo distribution contains a random-state file written under CMUCL 18, but there is no guarantee that this will work for other implementations.

23.6 The WRITE-ELEMENT-DATA Function

The WRITE-ELEMENT-DATA function can be called from the interpreter, or from a script file with arguments that limit which plotted traces are saved to file:

write-element-data &optional *elements−and−slots* &key *(output−format :lisp) filename suppress−comments* [*Function*]

The first optional argument is a list of elements or sublist containing the name of an element and symbol for the type of data saved. If there is only an element reference, then the DEFAULT-DATA-TYPE for that element will be used. For example:

```
(WRITE-ELEMENT-DATA '("11-5-6"
                      "11-5-46"
                      ("j43d-soma" dvdt)))
```

Note that data may only be written to file if it was saved from the last simulation (e.g. by using the appropriate call to ENABLE-ELEMENT-PLOT). In the case of :LISP OUTPUT-FORMAT, the written data file includes an assignment to special variables which may be referenced when the data file is loaded (see *ARCHIVE-VARIABLE-LIST*). For other options see the Reference Manual.

23.7 The WRITE-LISTS-MULTI-COLUMN Function

Multi-column data files may be written with the function:

write-lists-multi-column *lists* &key *(filename *results−filename*) (pathname−directory "") (announce−write t) indent comment (if−file−exists :append) (column−width 20) quit−on−first−null* [*Function*]

This writes the values in the sublists of LISTS in a multi-column format to FILENAME under PATHNAME-DIRECTORY. Note that the sublists may contain numbers, strings or symbols. If PATHNAME-DIRECTORY not supplied, then derive one under the surf-hippo data directory. If included, COMMENT is written to file first, followed by the data. Columns are tabbed by COLUMN-WIDTH spaces [default 20]. If a given sublist is empty while running through all the sublists of list, then a space is output for the corresponding column entry, unless QUIT-ON-FIRST-NULL is T [default NIL], in which case the file is closed.

24 File Names

Note that variables holding directory names in Surf-Hippo end with "/", contrary to the UNIX convention. For example, the global variable *SURF-HOME* will be a string with a final "/" (e.g. "/home/foo/surf-hippo/"), even though it is derived from the shell environment variables SURFHOME or HOME (which may or may not have a final "/", e.g. "/home/foo/surf-hippo").

In general, output filenames (plots, data, etc.) are constructed from the current time stamp and, if *ADD-SIMULATION-TO-FILENAMES* is T, the *SIMULATION-NAME*. This makes it convenient to keep track of simulation records. The time stamp mechanism described above is designed to generate a unique name for each simulation (as long as they occur at least 10 seconds apart). Also, the file writing functions will typically reference sub directories whose names are constructed from the circuit name and the date, for example (if *ADD-SIMULATION-TO-FILENAMES* is T):

```
surf-hippo/data/star-amacrine-3/3_29_1994/star-amacrine-3-297395627.dat
```

or if *ADD-SIMULATION-TO-FILENAMES* is NIL:

```
surf-hippo/data/star-amacrine-3/3_29_1994/297395627.dat
```

However, postscript filenames are generally derived from the title of the window that is being printed, so you must take care that this forms a legal (to Unix) filename.

There is also a global variable `*MAXIMUM-PS-FILENAME-LENGTH*` (default 32), which we added after some problems with VMS server based printers.

It is also possible that a filename, for example derived from a window title, will result in a file that is successfully written, but not printed (assuming that printing was requested). This may not generate an error message, so it is important to verify that a given file is actually printed. If there is problems, one solution is to change the filename and print the file from the UNIX shell.

For the moment, any automatically generated filename with a leading "_" or "-" has these characters removed. This is done by the function `MAKE-NICE-FILENAME`.

25 Histology Graphics

See also Section 33.

25.1 Basic Interface

The main histology graphics menu may be invoked from either the Main Menu, by the histology graphics window CONTROL-m option, or by invoking the function `HISTOLOGY` at the lisp prompt.

Note that if the cell(s) in a histology window is(are) no longer loaded into Surf-Hippo, then modification, selection of cell elements, etc, is disabled.

The options in this menu or in its submenus include variations on the following:

- "Viewing angle theta (degrees; For retina, 0 is for flat mount; 90 is radial mount)" -¿ See below.
- "Viewing angle phi (degrees; For retina 0 is flat/radial mount)" -¿ See below
- "Drawing scale (microns/pixel):"
- "Method to size the histology window to cells" -¿ Menu option allows for entering window parameters (see below).
- "Create new histology window?"
- "Change histology rendering details" -¿ this brings up a menu which may include the following:
  - "Label/Mark all cell nodes?"
  - "Label/Mark plotted nodes?" -¿ This can be used without drawing the cells to update newly chosen or deleted plotted nodes.
  - "Draw/Label sources on cells?"
  - "Channel graphics menu"
  - "Synapse stimulus/RF graphics menu"
  - "Draw only proximal segments?" -¿ prompts for the number of proximal segments out from the soma to draw, for each dendritic trunk.
  - "Modify element graphics"

If menu-based window sizing is chosen, then another menu appears with:

```
"Center of window along X direction [um]:"
"Center of window along Y direction [um]:"
"Histology window width [um]"
"Histology window height [um]"
```

For mouse and keyboard senstive actions for plot windows, see Section E.

25.2 Cell 2D Projections

Projecting 3D structures onto the histology window is a two part process: The first step is the calculation of the 2D projection of the structure, and the second step is the translation of the projection within the plane of the display window.

The anatomical information for each cell component is referenced to an XYZ coordinate system, with an implicit origin at (0,0,0). In this discussion we shall refer to the viewing plane as the X'Y' plane. The default orientation of this plane is such that X→¿X' and Y→Y'. The orientation of this XYZ system with respect a given brain structure is arbitrary, but as a general rule the typical anatomic/experimental view is corresponds to the default X'Y' projection. Typically, this means that Z is taken as the depth within tissue, oriented with respect to 2-dimensional sheets in brain. For example with retina, the XY plane is congruent with the plane of the retina, with the Z axis aligned along the radial dimension (for example, Z=0

at the ILM and increasing in the distal direction). Thus the default orientation of the X'Y' viewing plane corresponds to the retinal whole mount configuration, or retinotopic orientation. For flattened cortex, the XY plane is congruent with the surface of the brain, and the Z axis crosses the cortical layers. For cortical and hippocampal slice preparations, the XY plane is in the plane of the slice, since the slice is typically viewed "en-face". This is also the system for cortical neurons that are typically viewed perpendicular to the cortical surface.

Two dimensional projections onto the X'Y' viewing plane are taken as follows. Assume that the XYZ coordinates are placed with the XZ plane in the horizontal direction, with the Z axis emerging from the page and the X axis pointing to the right in the plane of the page. The Y axis points up in the plane of the page. Start with THETA = PHI = 0, where the X'Y' viewing plane is congruent with the XY plane. As the X'Y' plane is rotated about the Y' axis (now = Y axis), the azimuth angle PHI is the angle between the X and X' axises. Next, the X'Y' plane is rotated around the X' axis, and the elevation angle THETA is the angle between the Y and Y' axises. When a structure is drawn, the THETA and PHI values for the appropriate window are used to generate the 2D representation of each element in the structure.

Subsequent translation of the cell drawing in the viewing plane with respect to the histology window is done via the XY (really X'Y') center parameter menu described above. Rotation of the projected image, per se, is not implemented.

If either the viewing theta or phi is non-zero, the viewing angles will be displayed in the lower right corner. This label may be removed (or added to) via the (Control-t) text command described above.

25.3 Depth Relations are Not Rendered Faithfully

As a compromise to efficiency, there is no attempt to maintain depth relationships between overlapping sections of the dendritic tree or soma. For example, when two sections of the tree with two different color overlap, the choice of which section is drawn "in front of" the other is arbitrary. Somas are drawn either completely behind or completely in front of the dendritic trees, selectable from the Histology Menu, or not at all (also selectable from the Histology Menu).

25.4 Hints for Rotating Zoom Views

If you want to look at a zoomed view from a different angle (theta, phi), then bring up the Histology menu on the zoom window, specify the desired angles, and make sure to click the "Fix" or "Menu" option for "Method to size the histology window to cells".

25.5 Window Resizing

The current code does an incomplete job in sizing the graphics window properly, especially with respect to visualization of light stimulus or synapse receptive fields. Typically, though, use of the "Fix" or "Menu" options, instead of the "Automatic" option in the Drawing Menu will give you enough freedom to fix the picture. If the automatic window scaling is inappropriate you can adjust the size of the window by the following steps:

1. Resize the window with the mouse and the X window manager. The graphics will not readjust to this resizing, so you have to judge what the correct size will be.
2. Invoke the Cell Drawing menu (from Surf-Hippo Main Menu, or CONTROL-m on the graphics window).
3. Choose option "Fix" for "Method to size histology window to cells" and hit "OK".
4. The histology will now be redrawn centered on the resized window.

If an histology window is resized with the window manager, various graphics (including the highlighting of selected cell segments or somas) will be incorrect until the cell(s) are redrawn as just described.

The maximum size of a histology window is 90% of the screen width and height - thus if the chosen scale is too large then the cell drawing may be cut off.

25.6 Element Visualization - Sources, Channels, Synapses, Branches

Circuit elements can be highlighted on the histology rendering by choosing the appropriate options in the menus. For channels and synapses, different color markers may be assigned to different channel or synapse types (via "Change Histology Rendering Details" option).

With the "Mark specific branches" option in the histology rendering details, specific branches may be marked along their length with different colors. If you have chosen a segment prior to calling the Histology Menu, then the Mark Specific Branches option will default to the branch associated with that segment.

You can also assign colors to specific elements with the `ELEMENT-PARAMETER` function, where COLOR can be: 'RED, 'GREEN, 'BLUE, 'YELLOW, 'ORANGE, 'CYAN, 'PURPLE, 'BLACK, or 'WHITE. For example:

```
(element-parameter *synapse-type* 'color 'orange)

(loop for segment in (segments-to-soma 234) do (element-parameter segment 'color 'green))

(element-parameter 'DR-HH 'color 'cyan)
```

25.7 Light Synapse Receptive Field Graphics

The receptive fields for light synapses can be drawn in perspective by the appropriate selection in the Synapse Stimulus/RF Graphics submenu of the Drawing menu. The receptive field graphics shapes are determined by various simple approximations of the `'SPATIAL-RF-FUNCTION` component of the synapse type parameters (see the function `GET-SPATIAL-RF-ARRAY`). Connection lines are drawn from the synapse to its RF, at a height that can be specific to the synapse type (default 100.0 um). The style of these lines is determined by the global variables:

```
*SYN-RF-CONNECTION-DASH* (default '(10 10), see linestyles.doc for options)

*SYN-RF-CONNECTION-THICKNESS* (default 0, see linestyles.doc for options)

*SYN-RF-CONNECTION-SHADING* (default 100, see linestyles.doc for options)
```

These may also be set from the histology submenus. The color of the synapse marker on the soma or segment is also the color used for the RF, with a shading given by the global variable `*SYN-RF-SHAPE-SHADING*` (in percent, default 25). If `*SYN-RF-SHAPE-SHADING*` is 0, then only the outline of the RF is drawn.

See the description of `SET-TYPE-GRAPHICS` below.

Synapse stimuli are drawn either behind or in front of the cell drawing, according to the global variable `*WHERE-SYNAPSE-STIMULUS-GOES*` (:BACK or :FRONT). This may also be set in the synapse graphics menu for the current graphics window.

set-misc-histo-slots &key *(win *standard−graphics−output*) (scale (if win (gv win :scale) 3.0)) (colorize (when win (gv win :colorize))) mark−plotted−nodes label−plotted−nodes (background−color (if win (gv win :background−color) opal:white)) (mark−all−synapses (when win (gv win :mark−all−synapses))) (enable−marked−synapses (when win (gv win :enable−marked−synapses))) (draw−all−synapse−rfs (when win (gv win :draw−all−synapse−rfs))) (draw−synapse−rfs (when win (gv win :draw−synapse−rfs))) (phi−deg (if (and win (gv win :phi−deg)) (gv win :phi−deg) 0.0)) (theta−deg (if (and win (gv win :theta−deg)) (gv win :theta−deg) 0.0)) (soma−histology−fixed−diameter−p nil) (soma−histology−fixed−diameter−pixels 10) (soma−outline−p t) (draw−axons t) (draw−synapse−cxns t)* [*Function*]

This is used for setting some basic graphics parameters without using the menus. e.g.:

```
(SET-MISC-HISTO-SLOTS :scale 3.0 :phi-deg 90.0 :DRAW-AXONS nil)
```

Angle args are in degrees, and scale arg is in microns/pixel.

25.8 Colorization

Element values may be displayed as colors in histology windows, both during a simulation and afterwards. This mechanism in fact allows another way to specify simulation data storage and plotting, in a manner completely independent of the methods described in Sections 20 and 21.

Enabling colorization during a simulation requires setting `*COLORIZE-SIMULATION*` to `T`. Data for all segments and somas will also be saved on a sparse sampling grid when `*ENABLE-SPARSE-DATA*` (determined by `*COLORIZE-SIMULATION-STEP*`, and accessed by the function `CURRENT-SPARSE-DATA-TIMES`). If so, then the last simulation may be replayed using the function:

replay-colorized-simulation &key *(start−time 0) (stop−time *user−stop−time*) (time−step 0.1) include−colorizing−scale show−time−prefix win (repetitions 1) (display−time t) (elements (cell−elements)) (data−type 'voltage)* [*Function*]

The saved data may also be plotted with:

plot-element-sparse-data *elements* &key *(data−type 'voltage) (y−label "mv")* [*Function*]

and directly accessed with:

element-sparse-data *element* &optional *data−type* [*Function*]

This data is cleared at the beginning of a new simulation.

25.9 Bugs/Features

If you try to draw too big a picture, you may get a `DRAWABLE-ERROR` (Section 41.5.2. This could happen, for example, when the

```
"Size histology window to cells"
```

option is marked T for a

```
"Drawing scale (microns/pixel):"
```

value that is too small.

26 Initialization of Voltages and Concentrations

26.1 Initial Values for Voltages and Concentrations

The following mechanism is useful for imposing a "steady-state" condition on the circuit. Two variables may be used to set initial node voltages and concentration integrator concentrations:

node-voltage-initializations *'()* [*Variable*]

conc-int-initializations *'()* [*Variable*]

The first is a list of pairs, the CAR's of which are nodes, the CADR's of which are the initial voltage to set that node to, thus:

```
((NODE-1 value) (NODE-2 value) ... (NODE-N value))
```

where each NODE-X is either a node or an element (soma or segment, name or structure). `*CONC-INT-INITIALIZATIONS*` is a list of lists, the `CAR`'s being concentration integrators, and the `CADR`'s being lists of compartment initial concentrations to set that integrator to.

The function:

set-*node-voltage-initializations* [*Function*]

grabs the current node voltages and stores them in `*NODE-VOLTAGE-INITIALIZATIONS*`. Likewise, the function:

set-*conc-int-initializations* [*Function*]

grabs the current concentration integrator values and stores them in `*CONC-INT-INITIALIZATIONS*`.

The variables:

use-node-voltage-initializations *nil* [*Variable*]

use-conc-int-initializations *nil* [*Variable*]

enable the use of the initialization value lists. These options may be set by the Initialization Menu. Note that the initial state of voltage and concentration dependent gating particles are set by their steady state values determined by the initial values of the appropriate voltages or concentrations.

26.2 Virtual Holding Potentials

Normally, the voltage of each node at the begining of the simulation is set to the appropriate reversal potential of the leak conductance. Voltage dependent elements may also reference a "virtual" holding potential at the begining of the simulation.This is called a virtual holding potential since the actual voltage of the node is not changed. The effect is as if the node was voltage clamped from negative infinity time at the holding potential (and all other nodes clamped to either their virtual holding potential or their leak reversal potential), and then at time 0 there was an instantaneous and zero duration voltage clamp to the leak reversal potential for all nodes.

Virtual holding potentials may be set from the Initialization menu. Alternatively, to set the virtual holding potential for an element, use the function:

element-holding-potential *element* &optional *(value nil value−supplied−p)* [*Function*]

If VALUE (in mV) is supplied, this function sets the `'HOLDING-POTENTIAL` parameter of the circuit node associated with ELEMENT and returns VALUE (converted to double-float). Otherwise, it returns the current value of the `'HOLDING-POTENTIAL` parameter for the node, if that value has been set previously.

To set the holding potential for all somas and segments to their current voltage, use:

set-holding-potentials-to-current-values [*Function*]

To cancel the virtual holding potential for element, use:

```
(element-holding-potential element NIL)
```

or for all the cell elements at once:

```
(clear-holding-potentials)
```

Note that setting a virtual holding potential for an element will do the same for any other element at the same node.

26.3 Initial States for Gating Particles

Both two-state (HH-type) and Markov gating particles can have explicit initial conditions - see Sections 13.9 and 14.5.

27 Miscellaneous

This section covers various phenomena, functions and macros not discussed elsewhere. That's why its called "Miscellaneous".

27.1 Type Coercion Macros

A set of simple macros are provided in Surf-Hippo for type coercion of single numbers and numeric sequences. The following three macros coerce single number arguments to fixnums, single floats or double floats, respectively:

fix *arg* [*Macro*]

s-flt *arg* [*Macro*]

d-flt *arg* [*Macro*]

These macros take numeric sequence arguments and coerce them to either lists or arrays of the indicated type:

fix-list *arg* [*Macro*]

s-flt-list *arg* [*Macro*]

d-flt-list *arg* [*Macro*]

fix-array *arg* [*Macro*]

s-flt-array *arg* [*Macro*]

d-flt-array *arg* [*Macro*]

28 Output Windows, Postscript$_{TM}$ Output

28.1 Locking and Unlocking Graphics Windows

The global variable *LOCK-ALL-WINDOWS* (default NIL) when T causes all new windows (histology, plotting and information) to be locked so that they will not be overwritten. The following functions are related to this:

lock-all-windows [*Function*]

unlock-all-windows [*Function*]

lock-window &optional *(win *twin*)* [*Function*]

unlock-window &optional *(win *twin*)* [*Function*]

unstick-windows [*Function*]

See also Sections C, D, E and 41.

28.2 Window Visibility and Arranging

Normally, output windows (plotting and histology) will be automatically deiconified, raised and made visible when needed. For automatic runs, it sometimes convenient to suppress the showing of output so that you can do something else with the computer. Setting the following global variables (all of whom have a default value of T) may then help:

```
*RAISE-OUTPUT-WINDOWS* *DEICONIFY-OUTPUT-WINDOWS*
*UPDATE-OUTPUT-WINDOWS* *SHOW-OUTPUT-WINDOWS*
```

See also Section 29.10. For automatic tiling of graphics windows, try the following function:

arrange-windows &key *(windows−per−row *arrange−windows−per−row*)* [*Function*]
*windows use−menu (reassociate−windows *reassociate−windows*)*
(reassociate−windows−sublist−length
reassociate−windows−sublist−length)*
*(window−tile−x−gap *window−tile−x−gap*) (window−tile−y−gap*
window−tile−y−gap)*

This function may also be called from the Print Window Menu.

28.3 Generating Postscript$_{TM}$ Files

Plot and histology windows may be stored as Postscript$_{TM}$ files by typing CONTROL-p over the window, or by calling:

print-windows &key *(windows :all) (kill−ps−margins *kill−ps−margins*) (landscape *ps−landscape−p*) (include−title *print−windows−include−title*) (what−to−do *print−windows−what−to−do*) (printer *printer*) (print−together *print−together*) directory (filename−suffix *ps−filename−suffix*) arrange erase−files hard−copy−screen use−menu (exclusion−list *print−windows−exclusion−list*) (inclusion−list *print−windows−inclusion−list*)* [*Function*]

The printer is specified by the string global variable:

```
*PRINTER*
```

The default value for this is taken from the PRINTER Unix environment variable, although there may be some bugs with this. You can also set this variable explicitly, of course.

There are several global variables which effect the Postscript$_{TM}$ file format. When:

```
*INCLUDE-FILENAME-AND-DATE-IN-PS-FILES*
```

is T (default), then a small line of text is added to the printed page including the file name and the current date and time.

If you would like to remove this text from a Postscript file (e.g. for later inclusion as a text figure), then the offending lines (which may be removed from the file with an editor) are framed by

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Start of lower right corner filename and data....
   .
   .
   .
%% End of lower right corner filename and data....
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Other variables include (see Reference Manual for details):

```
*PS-POSITION-X* *PS-POSITION-Y*
*PS-SCALE-X* *PS-SCALE-Y*
*PS-LANDSCAPE-P* *PS-BORDERS-P* *PS-COLOR-P*
*PS-LEFT-MARGIN* *PS-RIGHT-MARGIN*
*PS-TOP-MARGIN* *PS-BOTTOM-MARGIN*
*LPR-PAPER-SIZE*
```

29 Random Hints

29.1 EXP-W-LIMITS Functions

The functions EXP-W-LIMITS EXP-W-LIMITS-DOUBLE EXP-W-LIMIT-GENERIC provide an in-lined version of EXP that saturates if the argument is too large or small. These functions have been used at various points in the Surf-Hippo code. When the global variable *NOTIFY-EXP-LIMIT* (default NIL) is non-NIL then a message is printed when one of the EXP-W-LIMITS functions punts.

29.2 Debugging Calls to SIM-ERROR

Sprinkled through the Surf-Hippo code are occasional explicit error checks, which may call the function:

sim-error &optional *(message "") (abort−on−error *abort−on−sim−error*)* [*Function*]

Normally this function will print out some information about the error and return to the Lisp top-level, avoiding a call to the Debugger (Appendix B). However, it may be useful to disable SIM-ERROR in order to exploit the Lisp Debugger and track down the error. To do this you must set the variable *ABORT-ON-SIM-ERROR* to NIL so that the call to SIM-ERROR does not go immediately to top-level.

29.3 Choosing Parameters for Numerical Integration

Try initial simulations (current clamp) with weak numerical constraints, for example start with a (large) fixed time step (e.g. 100 microseconds). The default values for the numerical method parameters are rather strict, with the aim of assuring very good accuracy.

Often, a fairly weak relative voltage error (the global variable *ABSOLUTE-VOLTAGE-ERROR*), for example on the order of 0.1mV, will give a good tradeoff between speed and accuracy. Sometimes, though, a too-large jump may be taken because of a very small 2nd derivative, even with non-linear elements. The symptom for this is an "unexpected" discontinuity in the voltage trace (obviously there is no strict criterium for this). In this case, the glitch may disappear with a small adjustment of *ABSOLUTE-VOLTAGE-ERROR* (either an increase or a decrease).

Voltage clamp (particularly with non-ideal voltage sources) simulations tend to be much more sensitive to the resolution of the numerical method. For example, under voltage clamp fixed time step integrations may be very susceptible to oscillations. These may only be visible if the plot resolution is taken at every time step - see the section on Plot Resolution. For example, if the plot resolution is at the default 2 time steps per plot sample, then it may appear that the clamp is "drifting" instead of oscillating. For circuits with voltage clamp(s) it is best to use variable time steps, and to try various error criteria to check for the convergence of the simulation (e.g. setting *ABSOLUTE-VOLTAGE-ERROR* for the LTE-based time step from 0.1mV to 0.001mV or less).

29.4 Naming Cells And Their Components

See also Sections 7.1 and 8.2.

The structure of the simulator is such that names of certain classes of objects, specifically cells, somas, segments, and all meta "X-type" classes must have unique names. Also in general problems are avoided if all other objects have unique names as well.

The root of this is with the name of the cell(s). When writing a function or a file to create a cell, a "good" (i.e. unused) name for the cell may be found at the beginning by the function:

check-cell-name *name* &key *(automatic−name−fixing t)* [*Function*]

This returns a string which is the original NAME argument is there is no cell already defined with that name. Otherwise, the function appends "-i", where i is an integer, to NAME in order to find a new name. If AUTOMATIC-NAME-FIXING is nil, then a menu prompt is used to approve the new name. For example:

```
(defun test-cell ()
  (let* ((cell-name (check-cell-name "Test-cell"))
         (cell (create-cell cell-name))
         (soma (create-soma :cell cell-name :diameter 31)))
    ;; Code that can explicitly access the CELL-NAME, CELL and SOMA local variables.
    (....)
    ))
```

In fact, by default the CREATE-CELL function runs CHECK-CELL-NAME. Thus, here is a more compact version (e.g. if you don't need to access the cell directly later in the function):

```
(defun test-cell ()
  (let ((soma (create-soma :cell (create-cell "Test-cell") :diameter 31)))
    ;; Code that can explicitly access the SOMA local variable.
    (....)
    ))
```

In this case, note that the CREATE-CELL function returns the created cell structure. The actual name of the created cell, because of the call to CHECK-CELL-NAME in CREATE-CELL, may be changed from the original "Test-cell" string. Note as well that CREATE-SOMA requires at least a single cell or cell name argument. All this means that in most cases, even with multiple invocations of the same cell circuit description, a unique cell name will be generated. This system is useful is you want to be able to add cell definitions together without taking care beforehand that the cell definitions all refer to unique names.

For objects that are added to a cell's membrane (channels, synapses, sources, etc.) the name of a given object is partly derived from the cell element that will receive the object. Therefore, if all the cell elements (somas and segments) have unique names, then the rest of the circuit objects will have unique names. Likewise, for both somas and segments, the name of their cell may be incorporated into the element name: in the case of segments when the global variable *ADD-CELL-NAME-TO-SEGS* is T. To help ensure that this incorporation is done when necessary (e.g. for unique element names) when there is more than one cell in the circuit, the following steps are taken by the simulator when new cells are read in and created.

Another feature to ensure good names for segments is also accomplished within CREATE-CELL. If this function creates and assigns a new name to the new cell, then the assumption is that this current cell is a copy of another cell, and the function automatically sets *ADD-CELL-NAME-TO-SEGS*. Subsequent calls to CREATE-SEGMENT then add the cell name to the segment name if the cell name is not at the beginning of the supplied segment name.

This procedure is not foolproof, since it is possible, for example, for two entirely different cell descriptions to use the same segment name. Thus the safest procedure is to always set *ADD-CELL-NAME-TO-SEGS* when mulitple cell circuits are constructed. This option appears in the circuit loading menus for convenience.

If you do try to load a circuit that will use an already assigned name, you will get an error similar to this:

```
Reading in circuit CA1-MAX-RED...
; Loading #p"/usr/local/surf-hippo/anatomy/c12861.ca1.sparcf".

Error in function TRANSLATE-NTSCABLE-LIST:
   create-segment: segment ca1_1-1-3  already defined, ignoring

Restarts:
  0: [CONTINUE] continue
  1: [ABORT   ] Return to Top-Level.

Debug  (type H for help)

(TRANSLATE-NTSCABLE-LIST <Soma ca1-soma>)
Source:
; File: /usr/local/surf-hippo/src/development/ntscable.lisp
```

```
  (CREATE-SEGMENT (CONSTRUCT-NTS-SEGMENT-NAME (NTH 0 SEGMENT-LIST))
        (COND (SOMA-PROXIMAL-LOCATION-AND-DIAMETER #)
(BP-PROXIMAL-LOCATION-NAME BP-PROXIMAL-LOCATION-NAME)
(T #)) CELL-NAME :RELATIVE-LOCATION ...)
  0] q
  * (surf)
```

A solution is to type "Q" as above, and restart (surf). Reload the circuit, but this time set `*ADD-CELL-NAME-TO-SEGS*` to T via the loading menu. See also discussion below re `*ADD-CELL-NAME-TO-SEGS-FOR-TREE-DUMP*.`

29.5 Initialize or Not When Loading New Circuit

The global variable `*INITIALIZE-ON-CIRCUIT-LOAD*` may be set NIL if you want to add subsequent circuits without erasing already loaded circuits. This is handled automatically if you use the menus for loading. Therefore, if you are loading multiple circuit definitions with a script, then be sure to execute:

```
(setq *INITIALIZE-ON-CIRCUIT-LOAD* t)
```

before loading the first circuit, and:

```
(setq *INITIALIZE-ON-CIRCUIT-LOAD* nil)
```

after loading the first circuit.

29.6 Saving Cell Geometries

In some cases, for example after consolidating a cell's dendritic tree, it is useful to save the new geometry. This may be done with the functions:

dump-tree-menu [*Function*]

dump-tree-list &key *(cells (cells))* *separate−files* *include−membrane−elts* *convert−to−simple−names* [*Function*]

where CELL is either a cell structure or the name of a cell. These functions write loadable Lisp files that define functions that which recreate the cell when loaded. Thus, the following procedure could be used:

1. Load original cell (Overall parameters, Load circuit menus).
2. Consolidate cell geometry (Edit circuit elements - Examine/modify distribution of electrotonic lengths menus).
3. Rename cell and/or cell type (Edit circuit elements - Edit names of circuit objects menus).
4. Dump transformed cell geometry to file (Information management - Dump cell geometry to file menus).
5. Load geometry file, which in turn defines (and puts in the compiled circuit catalog) a circuit function for that new geometry (Overall parameters, Load Lisp file menus).
6. Load the new circuit function to recreate the transformed cell (Overall parameters, Load circuit menus).

Note that the saved geometry files will only reproduce the cell geometry and the basic cell type parameters. Cell elements such as channels etc will have to be added either with the menus, or the geometry files may be edited before loading, or the files loaded as part of another script file.

Also, this method preserves segment `'RI-COEFFICIENT` values that are not equal to 1.0. This is important for dendritic trees that have been consolidated as described in Section 18.

To ensure that the saved file has unique segment names, the following variable is default T:

add-cell-name-to-segs-for-tree-dump *t* [*Variable*]

This may be changed in the menu that prompts for dumping the tree.

See also Sections 18, 19, and 5.

29.7 Plot Resolution

The default plot resolution, (the sample grid of computed time steps for producing plot data), is one plot point for every two time steps. In general, the fewer points plotted the faster the plotting and the less data that needs to be stored for later use. This value is determined by the global variable `*SAVE-DATA-STEP*` (default 2). In some cases, it may be important to verify that the integration is not oscillating with a period of 2 time steps. It will then be necessary to set `*SAVE-DATA-STEP*` to 1 to observe the oscillation, and not just the "envelope". `*SAVE-DATA-STEP*` may also be done with the plot parameters menu.

For simulations with a wide range of time scales, and in particular if there is the possibility of abrupt changes in the time step, it is a good idea to set `*SAVE-DATA-STEP*` to 1. Otherwise, it is possible to miss key transition points.

29.8 Resetting the Random Seed

A file named "random-state" is found in the lib directory, which supplies the random seed for the lisp function `RANDOM` (stored in the global variable `*RANDOM-STATE*`). This seed may be reset and accessed with the following functions, respectively:

save-reference-random-state &optional *filename* [*Function*]

get-reference-random-state &optional *filename* [*Function*]

If the optional FILENAME arguments for these functions are not supplied, then the afore-mentioned lib/random-state file is used.

29.9 Avoiding Bloat

It is often useful to define new global variables in the process of storing or analyzing data. However, after a while if these variables hold a substantial amount of data, they may overload the memory. Assuming that at some point the variables are no longer useful, they may be cleared out by:

clear-user-variables &optional *variables−to−keep* [*Function*]

29.10 Avoiding Plots

When running long simulations with a lot of output, it may be useful to supress the visibility of the plots (assuming that they are going to printed or the data otherwise saved). This can be done by setting `*HIDE-OUTPUT-WINDOWS*` to T. The hidden windows may still be printed (using the `PRINT-WINDOWS` function, either automatically in your script or interactively) as long as the variable `OPAL::*PRINT-NON-VISIBLE-WINDOWS*` is T (the default). Using `*HIDE-OUTPUT-WINDOWS*` set to T avoids display allocation errors which can kill a long-running session. Hiding plots with lots of data can also save significant time used when the windows are refreshed.

Note that in this case, you can still know that the simulation is running by keeping the clock visible (`*SHOW-TIME-REMAINING*` set to T).

30 System Issues

This section discusses some issues related to the interaction between Surf-Hippo and Unix.

30.1 Running Unix Shell Commands

Shell (csh) command strings may be executed from within Surf-Hippo with the function:

shell-exec *command* &optional *(show−result *show−csh−result*)* [*Function*]

30.2 Accessing Unix Environment Variables - A Few Common Pathnames

From Lisp you can access environment variables via the global association list variable `LISP::*ENVIRONMENT-LIST*`. For example,

```
* lisp::*environment-list*
((:XNLSPATH . "/home/netscape/nls/") (:XKEYSYMDB . "/home/netscape/XKeysymDB")
 ...
 (:BIBINPUTS . ".:/giga/lib/texmf/bibtex/bib") (:ARCH . "sun4"))
*
```

Thus, to get the value of SURFHOME, you would use:

```
* (cdr (assoc :SURFHOME lisp::*environment-list*))
"/usr/local/surf-hippo/"
*
```

Note that the global variable `*SURFDIR*` is set to the path for SURFHOME already:

```
* *SURFDIR*
"/usr/local/surf-hippo/"
*
```

As described in Section 4.10, there are a few globals like this, including:

surf-home *nil* [*Variable*]

surf-user-home *""* [*Variable*]

surf-user-dir *nil* [*Variable*]

For example,

```
* *Surf-user-dir*
"/usr/local/surf-hippo/"
* *circuit-directory*
"/usr/local/surf-hippo/circuits/"
* *data-directory*
"/usr/local/surf-hippo/data/"
* *plot-directory*
"/usr/local/surf-hippo/plot/"
```

30.3 Surf-Hippo Directories

The more important subdirectories in Surf-Hippo are as follows:

- `anatomy` Collection of original and converted cell anatomy files from a variety of external contributors.
- `bin` Binary Surf-Hippo Lisp files, under various catetegories. Normally these files are not necessary since the large `image` file includes them: they are used when the system is recompiled for generating a new image.
- `circuits` A convenient place for putting Lisp files of model circuits or of general code for later reference. Default directory for the `*CIRCUIT-DIRECTORY*` global variable. These files can be particularly useful when running a series of simulations directly from Lisp (i.e. via the interpreter), in which case they should be loadable directly into Lisp, or contain forms which are loadable. Of particular interest is:
  - `demos` Various demo files for running simulations.
- `cmucl` The lisp executables in this directory are specific to machine architectures and operating systems, as is the associated `image` file (in the `lib` directory). The file in this directory that is actually used is specified in the executable `surf-hippo` script in the top-level `surf-hippo` directory.
- `data` The default data directory. If the Unix environment variable SURFUSERHOME is defined differently than SURFHOME, then simulation data output will default to a `data` subdirectory under SURFUSERHOME (see above and Section 4.10).
- `plot` The default plot directory. Another `plot` directory may be referenced, according to the value of SURFUSERHOME, in the same way as explained for the `data` directory above.
- `doc` Various documentation, including this manual.
- `lib` Some files for system configuration information.
- `logs` Automatic system logs, for example generated when the global variable `*WRITE-LOG-FILE*` is set to `T`.
- `misc` Miscellaneous useful user information. `changes-x.doc` file(s) include non-backward compatible changes in the code. In particular, this directory includes:
  - `ilisp` Ilisp files.
  - `loaders` Files for system compiling. Not normally needed by the user.
  - `ntscable` Directory of ntscable code. Only needed if you need to recompile ntscable for your Unix box.
  - `rallpack_v1.1` The Rallpack benchmark files.
- `src` The Surf-Hippo source files. Important subdirectories are:
  - `cmucl-fixes`
  - `garnet-fixes`
  - `gui` The graphic user interface code for Surf-Hippo.
  - `hippocampus` Cell and mechanism files specific to hippocampus.
  - `parameters` Wide variety of membrane mechanism parameters (channels, synapses, etc.) taken from the literature.
  - `rallpack` Surf-Hippo files for running the Rallpack benchmarks.
  - `retina` Cell and mechanism files specific to retina.
  - `roylance-clmath` Copy of the Roylance numerical library distribution.
  - `sys` The Surf-Hippo system files.

31 Programming Hints

Random material on compiling, programming, and such.

31.1 Backward Compatibility

The Surf-Hippo source code defines on the order of 7000 defined symbols, including functions, macros and global variables. The policy of the development of Surf-Hippo is that all symbols with documentation strings are expected to preserve functional isomorphism in all future releases. Any exceptions to this policy are described in the file `doc/non-backward-compatible-changes`.

31.2 Relevant Functions for `CIRCUIT-LOAD`ed Forms

Circuit functions that define a new circuit, for example that will be loaded from within `CIRCUIT-LOAD`, should not call functions for drawing the circuit. These must be used only after a new circuit is loaded. For example, if `FOO-CIRCUIT` is

```
(defun foo-circuit ()
  (create-cell "Foo")
  (just-draw))
```

This won't get the circuit drawn:

```
* (circuit-load 'foo-circuit)
```

But by defining

```
(defun foo-circuit ()
  (create-cell "Foo"))
```

and

```
(defun foo ()
  (circuit-load 'foo-circuit)
  (just-draw))
```

Then calling `(FOO)` will both load the circuit and then draw it. Likewise, any functions which depend on element location should only be called after the circuit has been loaded (as above), and "processed".

31.3 Defining New Variables or Functions

When defining your own variables or functions, it is a good idea to make sure that Surf-Hippo does not already use the symbol that you want. The easiest way to check this by hand is to use DESCRIBE, e.g.:

```
  * (describe 'surf)
  SURF is an external symbol in the SURF-HIPPO package.
  Function: #<Function SURF {23E36A1}>
  Function arguments:
    (&optional circuit (automatic *automatic-run*) (load-only *load-only*)
     (keep-track-of-time-for-auto-run nil))
  Its defined argument types are:
    (&OPTIONAL T T T T)
  Its result type is:
    (MEMBER NIL T)
  On Monday, 1/29/96 05:10:36 pm EST it was compiled from:
  /usr/local/surf-hippo/src/sys/sim.lisp
    Created: Saturday, 1/27/96 04:55:00 pm EST
  * (describe '*user-stop-time*)
```

```
*USER-STOP-TIME* is an internal symbol in the SURF-HIPPO package.
It is a special variable; its value is 200.0.
   200.0 is a SINGLE-FLOAT.
Special documentation:
  The time to end the simulation, in milliseconds.
* (describe 'cons)
CONS is an external symbol in the COMMON-LISP package.
Function: #<Function CONS {1248819}>
Function arguments:
  (se1 se2)
Function documentation:
  Returns a list with se1 as the car and se2 as the cdr.
Its declared argument types are:
  (T T)
Its result type is:
  CONS
On Wednesday, 11/2/94 02:20:25 am EST it was compiled from:
target:code/list.lisp
  Created: Tuesday, 11/1/94 01:34:44 pm EST
  Comment: $Header: list.lisp,v 1.18 94/10/31 04:11:27 ram Exp $
* (describe 'foo)
FOO is an internal symbol in the SURF-HIPPO package.
*
```

In these examples, the symbol `'SURF` has been used to reference a function (which does not necessarily preclude its use as a global variable), and the symbol `'USER-STOP-TIME` is used as a special variable (which as well does not necessarily preclude its use as a function). The symbol `'CONS`, of course, is defined as a function in the COMMON-LISP package. `'FOO`, however, is not used by Surf-Hippo, so it would be safe to use it for something new. Note that you must use the single quote to denote a symbol.

Another, sure way to safely define new symbols is to do so under your own package, which in turn should use the SURF-HIPPO package. Such a package could be created as follows:

```
* (make-package "My-Package-Name" :use '("SURF-HIPPO"))
#<The My-Package-Name package, 0/9 internal, 0/9 external>
*
```

This case requires some familiarity with the concept of packages in Lisp.

31.4 Structure Slot Access

Most lisp code, for example files which define various circuits and elements, may be loaded into successive versions of Surf-Hippo without problem and without recompilation. The main exception to this is when code makes direct references to structure slots via accessor functions. For example:

```
(setf (channel-gbar-ref (element "Hippo-soma-NA1") 0.1))
```

`CHANNEL-GBAR-REF` is an accessor for CHANNEL structures, created with the `DEFSTRUCT` form. If your code uses structure accessors, then the code must be recompiled before loading into a new version of Surf-Hippo.

In general, we have included enough ways to access structure slots so that this mechanism is more or less transparent. These functions should be used in lieu of direct reference to structure slots in your code.

31.5 Compiling Individual Source Directories

Loading `surf-hippo/misc/loaders/main-compiler` by default compiles all the directories under `surf-hippo/src`. Loading the following files restricts the compile to specific directories:

```
surf-hippo/misc/loaders/compile-cmucl-fixes.lisp
surf-hippo/misc/loaders/compile-development.lisp
surf-hippo/misc/loaders/compile-garnet-fixes.lisp
surf-hippo/misc/loaders/compile-gui.lisp
surf-hippo/misc/loaders/compile-hippocampus.lisp
surf-hippo/misc/loaders/compile-parameters.lisp
surf-hippo/misc/loaders/compile-roylance-clmath.lisp
surf-hippo/misc/loaders/compile-sys.lisp
```

31.6 File Compiling Dependencies

There are three reasons that one file may depend (require) the prior compilation of another file - either the other file defines a structure (only `src/sys/structures.lisp`), defines a macro, or defines an inlined function. The various xx-loader files take into account the most typical examples of the first and second case, e.g. `sys-loader.lisp` forces a compile of all the source files if either `structures.lisp` or `structure-macros.lisp` has been changed. On the other hand, it is the responsibility of the user to make sure that any other dependencies are respected when recompiling Surf-Hippo.

31.7 Memory Diagnostics - Useful Functions

```
(vm::instance-usage :dynamic :top-n nil)

 INSTANCE-USAGE is an external symbol in the SPARC package.
 Function: #<Function SPARC:INSTANCE-USAGE {12FB6E1}>
 Function arguments:
   (space &key (top-n 15))
 Function documentation:
   Print a breakdown by instance type of all the instances allocated in
   Space.  If TOP-N is true, print only information for the the TOP-N types with
   largest usage.
 Its defined argument types are:
   ((MEMBER :STATIC :DYNAMIC :READ-ONLY) &KEY (:TOP-N (OR FIXNUM NULL)))
 Its result type is:
   (VALUES)
 On Wednesday, 11/2/94 02:41:34 am EST it was compiled from:
 target:code/room.lisp
   Created: Tuesday, 11/1/94 01:35:35 pm EST
   Comment: $Header: room.lisp,v 1.24 94/10/31 04:11:27 ram Exp $
```

32 Circuit and Simulation Names - Time Stamps

The circuit name (the global variable *CIRCUIT*) is determined by one of several ways, in descending priority:

1. Set to the first argument of the SURF function (if used).
2. Set to the function name that defines the circuit (specified in the circuit loading menu, or when the global variable *INPUT-IS-FUNCTION* is T).
3. Set to the filename of the file that defines the circuit (specified in the circuit loading menu, or when *INPUT-IS-FUNCTION* is NIL).

Simulation names (the global variable *SIMULATION-NAME*) are automatically generated or updated whenever a new simulation is run (for example, if using the menu interface, whenever "Run simulation (immediately)" is chosen from the main menu), as follows:

```
(setq *time-stamp* (round (/ (- (get-universal-time) *universal-time-conversion-factor*) 1d1)))
```

With the above assignment, the basic time stamp changes every 10 seconds. If there is an N^{th} subsequent simulation before this time stamp changes, the actual time stamp string will be comprised of the time stamp followed by "$+N$". Then, the name of the current simulation, given by the string variable *SIMULATION-NAME*, is comprised of the value of *CIRCUIT*, a "-", and the actual time stamp. The function DECODE-TIME-STAMP will decode the actual time stamp component of a simulation name, for example:

```
* (ENCODE-TIME-STAMP)
"17988444+4"
* (DECODE-TIME-STAMP)
Monday, 5/1/00 05:40:40 pm [-1] (fourth in this 10 second period)
```

The global variable *UNIVERSAL-TIME-CONVERSION-FACTOR* is used to reduce the length of the integer returned by the function GET-UNIVERSAL-TIME, since we don't need dates previous to 1994. Make sure not to run any simulations overnight on December 31, 1999 (;-}).

33 Graphics Window Names

The titles of graphics windows (e.g. plot and histology) are derived from the current simulation name when first created, for example "basic-tree-12487402: Voltages". Subsequent output to an existing window updates the window title, i.e. to reflect a new simulation name.

Additional Efficient Computation of Branched Nerve Equations: Adaptive Time Step and Ideal Voltage Clamp

This section is adapted from the article of the same name, published in the Journal of Computational Neuroscience (v8(3), pp.209-225, 2000).

Compartmental approximations of biophysically detailed neuron models, where the branching cable structure of the cell approximated by a series of isopotential compartments interconnected with resistors, are now a fundamental technique in computational neuroscience (Segev et al., 1998).

Many numerical methods spanning a wide range of complexity may be used to solve the resulting systems of partial differential equations (Mascagni and Sherman, 1998), However, the descriptions by Hines (1984) of $O(n)$ integration of tree circuit topologies, as naturally found in individual neurons, and mid-step solving of time-dependent non-linear elements (e.g. voltage-dependent gating particles), have been found to be quite effective despite their relative simplicity (Hines and Carnevale, 1995; implemented in the simulator package NEURON, Hines, 1992). These contributions by Hines have been crucial for the increasing application of biophysically and anatomically detailed compartmental neuron models.

In the following two sections I will present two straightforward improvements on the methods introduced by Hines. These methods include recasting the system equations to allow adaptive time step integration, and a reordering of the circuit matrix to allow ideal voltage clamp of arbitrary nodes.

Since analytical statements concerning the stability of adaptive time step integration are lacking (Vlach and Singhal, 1983), in the succeeeding section I will present several simulations demonstrating the stability and convergence of the presented methods, using a standard Hodgkin-Huxley axon model. Example code is given in Appendix H.

34 Adaptive Time Step

We first review the solution of the circuit equation as described by Hines. The method, which uses alternate implicit and explicit integration steps, and which is equivalent to the Crank-Nicolson method, is second order correct in time and space and numerically stable.

Given (see Figure 4):

- t_x, the time grid for the stored node voltages
- $t_{(n-1)}$, t_n, $t_{(n+1)}$; the last time, current time and prediction time, respectively.
- $\Delta t_n = t_{(n+1)} - t_n$, the current time step
- t'_x, the staggered time grid for the midstep evaluation of particle states, inputs and node voltages, where $t'_{(n+1)} = t_n + \Delta t_n/2$

In the following discussion, the stored values of state variables at the n^{th} time step will be notated with the subscript n, e.g. V_n $(= V(t_n))$ for voltages, and x_n $(= x(t'_n))$ for gating particles.

Consider a compartmental circuit model with N nodes. The implicit phase of the solution at each time step $t_n \rightarrow t_{(n+1)}$ consists of solving the following matrix equation for the node voltage vector $\mathbf{V}(t'_{(n+1)})$ at the midstep, given the known voltages at the current time step, $\mathbf{V}_n$:

$$\overbrace{(\mathcal{G}(t'_{(n+1)}) - (2/\Delta t_n \times \mathbf{C}\mathcal{I}))}^{\text{(Almost) Tri-diagonal matrix}} \times \overbrace{\mathbf{V}(t'_{(n+1)})}^{\text{Solve for}} = \begin{aligned} & -2/\Delta t_n \times \mathbf{C}^T \mathbf{V}_n \\ & + \mathbf{GE}(t'_{(n+1)}) \\ & + \mathbf{I}(t'_{(n+1)}) \end{aligned} \quad (8)$$

Where, for the node admittance matrix $\mathcal{G}$ (Desoer and Kuh, 1969),

$$\begin{aligned} G_{ij} &= -g_{ij},\ i \neq j \\ &= \sum_{k=1}^{N} g_{ik} + \sum g_{elt}(t'_{(n+1)}),\ i = j \end{aligned} \tag{9}$$

where g_{ij} is the cable compartment axial conductance between nodes i and j. Similarly, for the vector $\mathbf{GE}(t'_{(n+1)})$,

$$GE_i = \sum g_{elt}(t'_{(n+1)}) \times E_{g_{elt}}(t'_{(n+1)}) \tag{10}$$

Thus, g_{elt} refer to the (possibly time-dependent) membrane conductances (leak, channels, synapses) between a given node and ground, and $E_{g_{elt}}$ refer to the appropriate reversal potentials. The sums of g_{elt} in Equations 9 and 10 are over all membrane conductance elements connected to node i. Again, note that these circuit elements are evaluated at the staggered grid points $t'_{(n+1)}$. For the vector $\mathbf{C}$, C_i is the membrane capacitance of node i. $\mathcal{I}$ is the identity matrix. For the vector $\mathbf{I}$, I_i represents any grounded current source (or sum of sources) connected to node i, evaluated at time $t'_{(n+1)}$. Since the compartmental model of either a single neuron or a network of neurons (without gap junctions) describes a tree toplology, the solution of Equation 8 is an $O(N)$ operation (Hines, 1984).

The explicit phase of the solution to finish the time step evaluation is then given by:

$$\mathbf{V}_{(n+1)} = 2\mathbf{V}(t'_{(n+1)}) - \mathbf{V}_n$$

34.1 Solution of Non-linear Conductances on a Staggered Time Grid

The staggered time grid evaluation of the Hodgkin-Huxley gating variables is $O(\Delta t^2)$ accurate without interation because these variables are not instantaneous functions of voltage (Hines, 1984, Mascagni and Sherman, 1998). This condition holds for ohmic models of pore conduction (e.g. the classical Hodgkin-Huxley description): permeation models of pore conduction that are based on formulations such as the constant-field equation are another matter since the basic conduction term is an immediate non-linear function of voltage. For Hodgkin-Huxley ohmic channels, the contribution of a given channel to the circuit matrix is given by the product of the channel's maximum conductance and the ensemble of gating particles. In principle a small-signal linearization of a constant-field permeation channel could be incorporated into the matrix in a similar manner, taking care to adjust the apparent channel "reversal potential" accordingly in the matrix equation. Another approach is to treat the channel as a voltage-dependent current source, adding its contribution to the right-hand side of Equation 8.

34.2 Integration of Particle States

We now consider the solution of Equation 8 using an adaptive time step. The mid-step computation of voltage-dependent state variables may be recast as follows. The problem is to solve the differential equation for a gating particle x:

$$\dot{x} = \frac{x_\infty(V) - x}{\tau_x(V)} \tag{11}$$

on the staggered time grid t'_n, defined earlier. For second-order accuracy this solution requires the values of $x_\infty(V)$ and $\tau_x(V)$ at the midpoints of the staggered grid. If the time step is fixed, these midpoints map back to the original time grid: thus, solving for $x(t'_{(n+1)})$ references the (known) node voltage V_n. However, for an adaptive time step, the midpoint of the staggered grid in general is not equal to t_n, but must be calculated as a function of the present and last time steps of the original grid (Figure 4). We first derive the time step for the staggered grid:

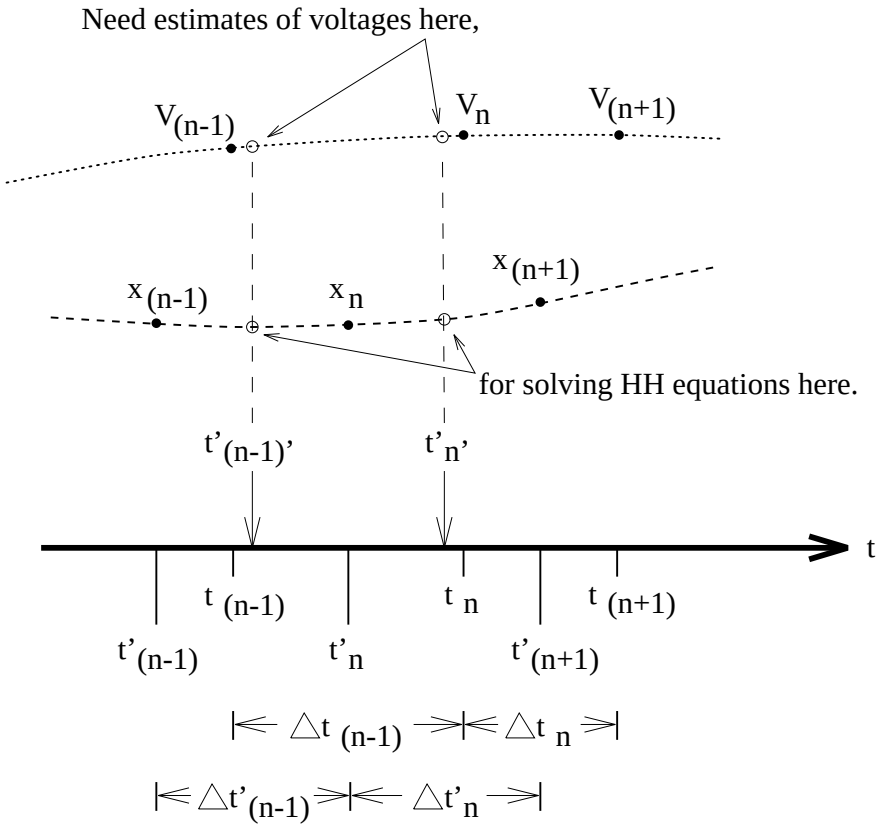

Figure 4: Time grids for evaluation of node voltages, gating particle states and inputs, as described in the text.

$$\Delta t'_n = t'_{(n+1)} - t'_n = \frac{\Delta t_n + \Delta t_{(n-1)}}{2}$$

Letting $n + \frac{1}{2} \equiv n'$, the staggered grid midpoint time may be expressed as:

$$t'_{n'} = t'_n + \frac{\Delta t'_n}{2}$$

The corresponding voltage can be obtained by simple regression from t'_n:

$$V(t'_{n'}) = V(t'_n) \ + \ \dot{V}(t'_n) \times \frac{\Delta t'_n}{2}$$

where

$$V(t'_n) = \frac{V_n + V_{(n-1)}}{2}$$

$$\dot{V}(t'_n) = \frac{V_n - V_{(n-1)}}{\Delta t_{(n-1)}} \tag{12}$$

Making the following discretizations:

$$x(t'_{n'}) = \frac{x_{(n+1)} + x_n}{2}$$

$$\dot{x}(t'_{n'}) = \frac{x_{(n+1)} - x_n}{\Delta t'_n} \tag{13}$$

We now solve Equation 11 at $t'_{n'}$. Letting $\tau_x = \tau_x(V(t'_{n'}))$ and $x_\infty = x_\infty(V(t'_{n'}))$, after some algebra we arrive at the following solution for $x_{(n+1)}$:

$$x_{(n+1)} = \frac{[x_\infty \times \Delta t'_n] - x_n[\tau_x - \Delta t'_n/2]}{\tau_x + \Delta t'_n/2} \tag{14}$$

As pointed out be Hines, the voltage-dependencies of the particle kinetics may be stored in lookup tables for efficiency - in this case, tables for $x_\infty(V)$ and $\tau_x(V)$ are stored. An equivalent formulation to Equation 14 may be made in terms of the rate constants α_x and β_x associated with Equation 11 (cf. Equation 9 in Hines, 1984), but the form given above is perhaps more clear since in the latter case tables for $\alpha_x(V)$ and $\frac{\alpha_x(V)-\beta_x(V)}{2}$ must be generated. In practice, additional computational savings may be made by calculating $V(t'_{n'})$ only for those circuit nodes which have voltage-dependent elements, and only once, at the beginning of each time step, such that the result may be used by multiple elements on a given node.

We may note that Equation 14 requires three additions, one multiplication and one division. This compares with the single multiplication and addition required in the case of a fixed time step (*ibid.*), where the (known) time step may be incorporated in the lookup tables. We will return to this point in the discussion.

34.3 Time Step Determined by LTE of Node Voltages and Particle States

The time step during the integration is determined according to its relation to the truncation error. Since the Crank-Nicolson method is first order this error is given by the second term of the Taylor series expansion of the solution, and is called the linear truncation error (LTE). Thus, after evaluation of the circuit at each time step, an estimate of the LTE for the circuit's state variables are determined. For each state variable an estimate of the second derivative at the midpoint between the current time and the two time steps back (note that for a fixed time step this corresponds to the last time step) is made by considering the known values of the first derivatives on the appropriate time grid.

In practice, the maximum of the the estimated errors for, respectively, all the considered node voltages (see below), all channel gating particle states, and all concentration integrator concentrations are examined. These maximum errors are compared with user-specified maximum errors, and if any are exceeded then the

time step is repeated with a smaller step, determined by the maximum LTE at the current step. If all errors are within the user-specified bounds, then the integration moves on, now with the next step determined by the maximum LTE.

Here we first describe the derivation for the LTE of the various state variables, and then we show how these estimates are translated into the next time step.

The LTE for the predicted node voltage $V_{(n+1)}$, LTE_V, is given by:

$$\begin{aligned} \text{LTE}_V &= \frac{\ddot{V}(t'_{n'})}{2}\Delta t_n^2 \\ &= \frac{\dot{V}(t'_{(n+1)}) - \dot{V}(t'_n)}{2(t'_{(n+1)} - t'_n)}\Delta t_n^2 \end{aligned}$$

where the expression for $\dot{V}(t'_n)$ was given in Equation 12. Letting the maximum *a-priori* allowed node voltage error be given by ϵ^V_{max}, the maximum relative voltage error ϵ^V_{rel} is given by:

$$\epsilon^V_{rel} = \frac{\text{LTE}^*_V}{\epsilon^V_{max}}$$

where LTE^*_V is the maximum LTE_V over all considered node voltages.

For concentration integrator systems, the corresponding LTE_C may be handled in an entirely analagous manner as for the node voltages, assuming that the concentrations are evaluated on the same time grid (this is the case for Surf-Hippo, with the default integration method for concentrations being a fully implicit method). In this case, the above equations may be applied as is, with the voltage variables replaced with concentration variables; given a user-specified maximum allowed concentration error ϵ^C_{max}, the important resulting measure is ϵ^C_{rel}.

For the integration of gating particles, the linear truncation error LTE_x for the predicted state value $x_{(n+1)}$ (occuring at time $t'_{(n+1)}$) is derived as follows. Letting t'' be midway between the last two midpoints $t'_{(n-1)'}$ and $t'_{n'}$ of the staggered grid, then:

$$\begin{aligned} \text{LTE}_x &= \frac{\ddot{x}(t'')}{2}\Delta t_n'^2 \\ &= \frac{\dot{x}(t'_{n'}) - \dot{x}(t'_{(n-1)'})}{2(t'_{n'} - t'_{(n-1)'})}\Delta t_n'^2 \end{aligned}$$

where the expression for $\dot{x}(t'_{n'})$ was given in Equation 13. Now, letting the maximum allowed particle error be given by ϵ^x_{max}, the maximum relative particle error ϵ^x_{rel} is given by:

$$\epsilon^x_{rel} = \frac{\text{LTE}^*_x}{\epsilon^x_{max}}$$

where LTE^*_x is the maximum LTE_x over all the gating particles in the circuit.

We now derive the maximum allowed time steps corresponding to the different LTEs. Given the maximum relative voltage error ϵ^V_{rel}, the maximum time step allowed by the node voltages, Δt^V_{max}, is given by:

$$\Delta t^V_{max} = \frac{\Delta t_n}{\sqrt{\epsilon^V_{rel}}} \tag{15}$$

A similar parameter, Δt^C_{max}, determined by the maximum relative error in the concentrations, ϵ^C_{rel}, may be estimated in exactly the same manner.

For the gating particle error, since LTE_x is a function of both Δt_n and $\Delta t_{(n-1)}$ (via $\Delta t'_n$), determining the corresponding maximum allowed time step Δt^x_{max} is a bit more complicated than Δt^V_{max}. Thus, we obtain:

$$\Delta t^x_{max} = \max\left(\frac{2\Delta t'_n}{\sqrt{\epsilon^x_{rel}}} - \Delta t_{prev},\ \Delta t^x_{min}\right) \tag{16}$$

where the first term on the right depends on whether or not the integration step will be repeated:

$$\Delta t_{prev} = \begin{cases} \Delta t_{(n-1)} & \text{Time step repeated} \\ \Delta t_n & \text{Integration advances} \end{cases}$$

Note, however, that when the time step must be repeated, the first term on the right of Equation 16 may be less than 0 (when $\sqrt{\epsilon^x_{rel}} \geq \frac{\Delta t_n + \Delta t_{(n-1)}}{\Delta t_{(n-1)}}$). Thus, a lower (positive) bound (Δt^x_{min}) must be made, as indicated.

Finally, if the largest of the maximum relative errors (ϵ^V_{rel}, ϵ^x_{rel} and ϵ^C_{rel}) is greater than one, then the current time step is repeated; otherwise the integration moves forward. In either case, the new time step is taken as:

$$\Delta t_n = \min\left(\Delta t^V_{max},\ \Delta t^C_{max},\ \Delta t^x_{max}\right) \tag{17}$$

34.4 Node Voltages for Consideration of LTE$_V$

At any given time during the simulation, the node with the largest error must be one that has some input associated with it, e.g. a source, channel, or synapse. All other nodes are driven by their neighboring nodes, and since the capacitance of each node is non-zero, then the response of a node driven by the voltage of a neighbor will always be slower than the neighbor's voltage. Therefore, for estimating the maximum LTE$_V$, we need only consider nodes with inputs; at the beginning of the simulation a list of all nodes with active membrane elements may be constructed for this purpose. One exception to this rule is in the case of ideal voltage clamp simulations, to be described below: here the node with the input is actually removed from the circuit, and copies of the voltage source are transferred to adjacent circuit nodes. Thus, these nodes may now be considered to have "inputs", and are included in the LTE$_V$ calculation.

34.5 Time Step Fudge

Although the determination of the time step as outlined above does the best possible (first-order) job based on the *past* behaviour of the circuit, clearly it may underestimate the resulting error for the next time step. This results in a tradeoff between the total number of time points during the integration, and any additional iterations that result from underestimating the LTE.

One method for improving the tradeoff is by introducing a fudge factor $\varepsilon_{\Delta t}$, less than or equal to one, which is used as a coefficient in the right hand side of Equations 15 and 16. In general, as $\varepsilon_{\Delta t}$ becomes smaller, the number of time points will increase and the number of iterations will be reduced. After a certain point for a small enough $\varepsilon_{\Delta t}$, the number of iterations will start to increase; in general $\varepsilon_{\Delta t}$ should be set to minimize the number of iterations. In Figure 5, this tradeoff suggests an optimal value for $\varepsilon_{\Delta t}$ to be about 0.8.

34.6 Breakpoints

For adaptive time step integration, breakpoints are time points that the simulation must incorporate in addition to those chosen by the LTE-based algorithm described above. In general a breakpoint is chosen because there is *a priori* knowledge of some input which begins or changes abruptly at that time. This makes less work for the adaptive time step, since without this information the initial stepping would more than likely encounter the input sometime after its initiation, and thus may be forced to backup if the input was too large at the first attempted time point that "saw" the input or the change in the input. The result (in general) with using breakpoints then is fewer overall iterations and a solution that is less likely to have "ringing". Breakpoints also avoid the situation in which an abrupt and short input might be completely missed by large time steps determined during a previous period of low activity.

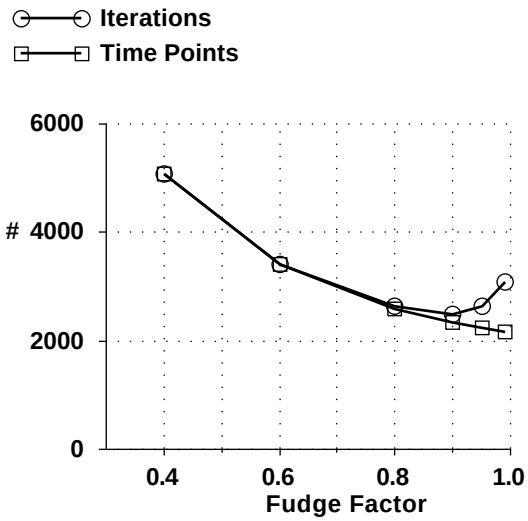

Figure 5: Total iterations and time points vs adaptive time step fudge factor for simulations of repetitive firing as shown in Figure 8. In all the adaptive time step simulations shown here the fudge factor $\varepsilon_{\Delta t}$ is set to 0.95; although in general a value about 0.8 is optimal, the higher value is taken in order to deemphasize this factor in comparing the adaptive versus fixed time step method.

Thus, prior to a simulation, breakpoints are added for any pulse-based source and for at the onsets of any autonomous processes, such as event driven synapses. During a simulation new breakpoints may also be generated with the evaluation of event-based elements, including axons and voltage-dependent synapses. When the simulation is evaluated at a breakpoint, the subsequent time step is taken to be an user-defined minimum step, unless this advance takes the integration beyond the next breakpoint, in which case the next breakpoint determines the time step.

In Surf-Hippo, element type definitions (see Appendix H) may optionally include a specification for the inclusion of breakpoints referenced to that element's onset time. For example, a synapse type definition, which includes the time course of the conductance change, may specify (a list of times in milliseconds, with the parameter keyword `SPECIFIED-WAVEFORM-BREAKPOINTS`) that whenever one of its synapses is triggered, then one or more extra breakpoints (referenced to the onset of the synaptic conductance change) are automatically added to the global breakpoint list of the current simulation. In addition, trigger events for event driven synapses cause a breakpoint to be added at a delay after the event that is one half the duration of the defined conductance waveform.

34.7 Other Time Step Constraints

Under some conditions it may be useful to impose an overall maximum time step. For example, a maximum could be imposed during the evaluation of some element types, in particular those that are driven by an *a-priori* waveform. In these cases (particularly if the waveform is not well-behaved), the global maximum time could be set to a value appropriate for the specific waveform.

35 Ideal Voltage Clamp

Ideal voltage clamp means that a given node j (or nodes) in the circuit includes a controlled voltage source, $V_j^s(t)$, connected to ground. However, when the circuit is described in the form given by Equation 8, this prevents a solution for $\mathbf{V}(t'_{(n+1)})$ since now there are more equations than unknowns (the value of $V_j(t'_{(n+1)})$, where node j is voltage clamped, is known and given by $V_j^s(t'_{(n+1)})$).

One solution is to make the voltage clamp non-ideal by adding a series resistance R_{Source} to V_j^s, so that the voltage source and resistor are handled like any other membrane element and associated reversal potential

on the right side of Equation 8. However, although this allows the evaluation of voltage clamp within the original circuit (and simulator) structure, this strategy has two disadvantages. First of all, the smaller the value of R_{Source} the more unstable the integration becomes, neccesitating smaller time steps. This arises because as R_{Source}^{-1} becomes much larger than the g_{elt}s, the stiffness of the resulting system of equations increases. Second, although voltage clamp circuits in real life are not ideal (e.g. they include a non-zero source impedance), it is useful to examine the conceptually simpler ideal case in simulations separately from that of the more realistic case.

These problems may be avoided by transfering copies of the voltage source V_j^s to a grounded membrane branch element of all nodes i connected to j, each of which includes a series conductance given by the axial conductance between i and j, g_{ij} (see Figure 6). In addition, the original connection between i and j via g_{ij} is eliminated. This procedure results in an invertible circuit matrix which is of order $N-1$. The stability (and stiffness) of the resulting circuit is similar to that of the original circuit, since there are no additional (large) terms to the node admittance matrix, contrary to the case for the non-ideal voltage clamp simulation.

With respect to the topology of the circuit, this procedure may be thought of as cutting the tree into two or more subtrees, depending on how many nodes are connected with j. Since the removal of any subgraph of a tree topology nevertheless results in one or more tree topologies, the original node ordering and efficient inversions as described by Hines is still applicable, in this case to the new, smaller trees.

The only remaining problem is the calculation of the currents through the membrane elements associated with node j (e.g. channels, synapses, membrane leak and capacitance) and the total current through the source V_j^s. In the former case, the circuit elements are evaluated as before, with the node voltage (or derivative, in the case of membrane capacitance) taken from the value of V_j^s at the appropriate time. In the latter case, the total current supplied by V_j^s is simply the sum of all the branch currents from the grounded membrane elements of node j, plus the axial branch currents across the appropriate g_{ij} with a driving force given by the difference between the voltage of the neighboring nodes i and V_j^s. Note that the evaluation of the voltage source current passing through membrane elements of node j is independent of the evalation of the new circuit matrix.

36 Results of Adaptive versus Fixed Time Step, and Ideal versus Non-Ideal Voltage Clamp

I will now present various current clamp and voltage clamp simulations to illustrate the methods just described. The cell model is based on the Rallpack 3 specification (Bhalla et al., 1992), which defines a single unbranched 1 micron diameter cable, 1000 microns long, with a nominally homogenous distribution of the canonical Hodgkin-Huxley I_{Na} and I_{DR} squid axon channels. The version used here is defined with 11 compartments, constructed so that the circuit is symmetrical (see Appendix H). The compartment on one end is defined to be the "soma"; the compartment on the other end is referred to as the "distal compartment". For detailed comparisons all plotted simulation results include every time step; normally plotting every other time step of the integration is sufficient for typical studies.

The current clamp protocol (Figure 7, left) is the response to a somatic current source of 0.1 nA, applied at 10 milliseconds and lasting until the end of the simulation. The initial holding potential for all compartments is -65mV, thus there is an initial transient as the cell moves towards the true resting potential of -72.7mV. This simulation represents a reasonable test of the adaptive time step algorithm since the activity in the circuit is somewhat dispersed in both space and time.

For the voltage clamp protocol (Figure 7, right), the voltage clamp is applied at the soma and all non-somatic channels are deactivated for simplicity. A somatic holding potential of -70mV is used, and the complete protocol consists of four repetitions, each with a 35 millisecond voltage step (to -70, -50, -30 and -10mV) applied at 10 milliseconds, followed by a return to -70mV. In all cases the voltage source has a fixed transition slope of 1000mV per millisecond.

The fixed time step simulation of the repetitive firing under current clamp converges as the time step goes from 0.01 (5,000 time steps) to 0.005 milliseconds (10,000 time steps) (Figure 8, top). Similar convergence occurs as the maximum allowed voltage error ϵ_{max}^V goes from 0.05 (1,662 time points, 1,982 total iterations) to 0.005mV (5,151 time points, 5,408 total iterations) in the adaptive time step case, without considering the LTE of the particle states (Figure 8, bottom).

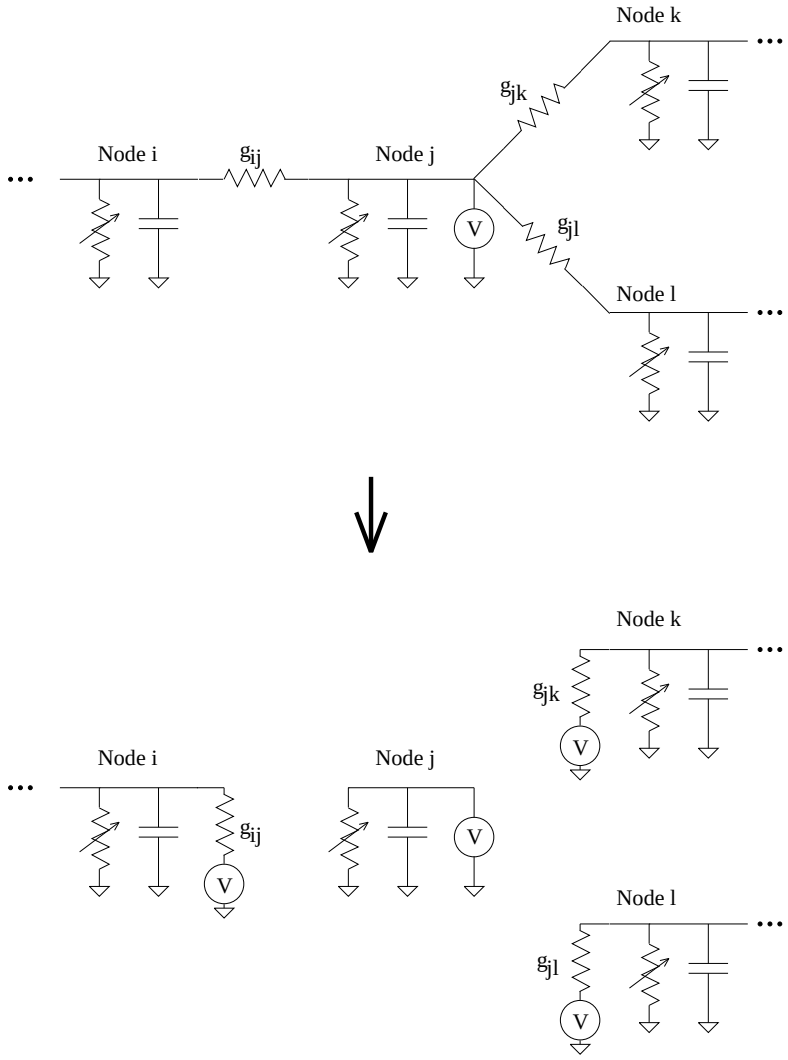

Figure 6: Method for splitting a circuit tree for ideal voltage clamp. In the example shown the voltage clamped node j is connected to three other nodes i, k and l, but the method is the same for an arbitrary number of connections. Here the original circuit tree is divided into three new trees (terminated by nodes i, k and l); node j is evaluated independently of the new circuit matrix defined by the three trees. The variable resistors represent membrane conduction elements (leak, channels and synapses) for each node and, implicitly, the associated voltage sources representing the appropriate reversal potentials.

There is an inherent tradeoff between the various LTE criteria, depending on the *a-priori* values given for ϵ^V_{max}, ϵ^x_{max} and ϵ^C_{max}. This is illustrated in Figures 10 through 12 in the case of ϵ^V_{max} and ϵ^x_{max}. These simulations show that there is a smooth and consistent exchange with respect to which type of LTE determines the time steps, as one criteria is made tighter than the other.

Under voltage clamp, the part of the response that is most sensitive to numerical stability and accuracy issues are probably at the pulse transitions, during which time the voltage source current is dominated by the capacitive transient. Under ideal voltage clamp, in Figure 13 we can see that the adaptive time step case follows the high-resolution fixed time step case ($\Delta t = 0.001$ms) quite well, without ringing. In the non-ideal voltage clamp case, there is a basic tradeoff between stability and accuracy as a function of R_{Source}. This is illustrated in Figure 14, where, under adaptive time step, a very low value of R_{Source} shows oscillations in the voltage source current.

The stability of the source current for non-ideal voltage clamp is much more sensitive to R_{Source} when a fixed time step is used, as seen in Figure 15. When $R_{Source} = 0.01$MΩ there is a huge oscillation (amplitude on the order of 50nA) in the voltage source current during the pulse transitions. When $R_{Source} = 0.1$MΩ, there is still a significant oscillation (amplitude on the order of 2nA). As a practical matter, although the voltage error for the non-ideal voltage clamp with $R_{Source} = 0.1$MΩ is not very significant (Figure 14), these oscillations make it more difficult to extract the correct peak values of the clamp currents.

36.1 Discussion

The basic performance issue with regards to the adaptive time step method is at which point the greater number of operations per time step for this method is countered by the smaller number of time steps and iterations for a given accuracy as compared to the fixed step method. In the current clamp simulation presented here the adaptive time was about 2.5 times faster than the fixed time step method. For the (ideal) voltage clamp simulation the difference was about 12 times. While it must be pointed out that there is no formal proof that the tested code is as efficient as possible (for either method), nevertheless this is a strong indication of the advantage of the adaptive method.

In the case of voltage clamp, I have shown how an adaptive time step allows a small enough value of R_{Source} ($= 0.1$MΩ) for the non-ideal voltage clamp so that voltage errors are small, that with a fixed time step gives transient oscillations that can complicate data analysis (Figure 15). Nevertheless, the ideal voltage clamp formulation is much more robust to this issues, as well as being conceptually simpler. The computational overhead of this method is actually less (though trivially so) than that for the non-ideal case, since in the ideal case the dimension of the circuit matrix is reduced by one.

For both adaptive time step and ideal voltage clamp, the programming of the method is straightforward, and should be easily ported to existing code that use the Hines method (e.g. NEURON, GENESIS (Bower and Beeman, 1994)).

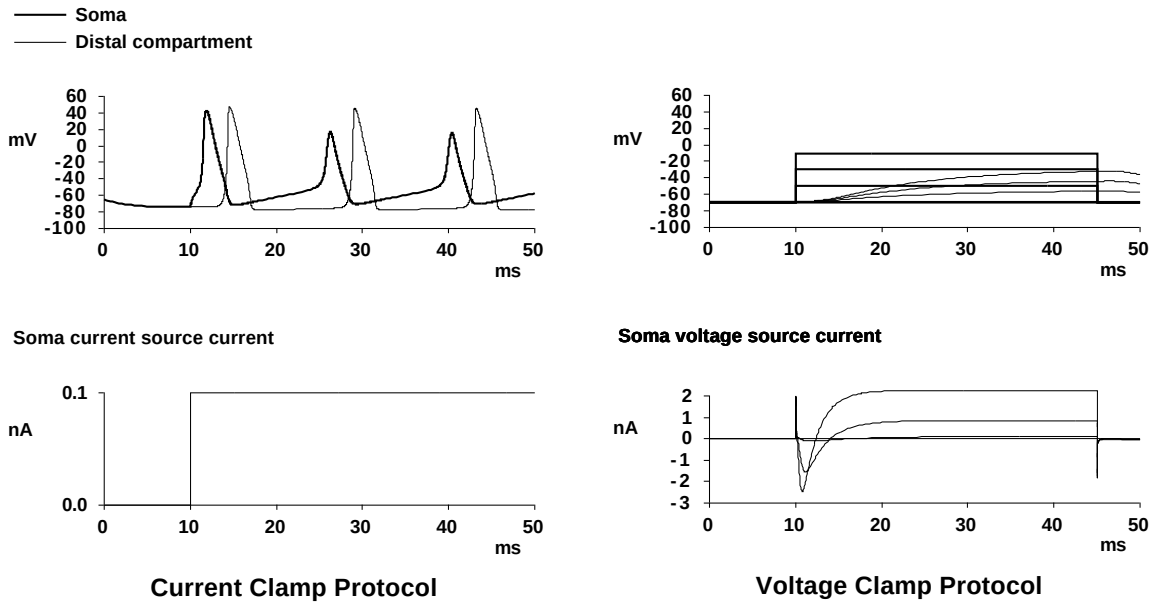

Figure 7: Basic current clamp (left) and (ideal) voltage clamp (right) protocols for the simulations presented in this paper, run here using an adaptive time step. For the current clamp protocol all compartments have a homogeneous distribution of Hodgkin-Huxley I_{Na} and I_{DR} channels (source code given in Appendix H). For the voltage clamp protocol only somatic channels are enabled.

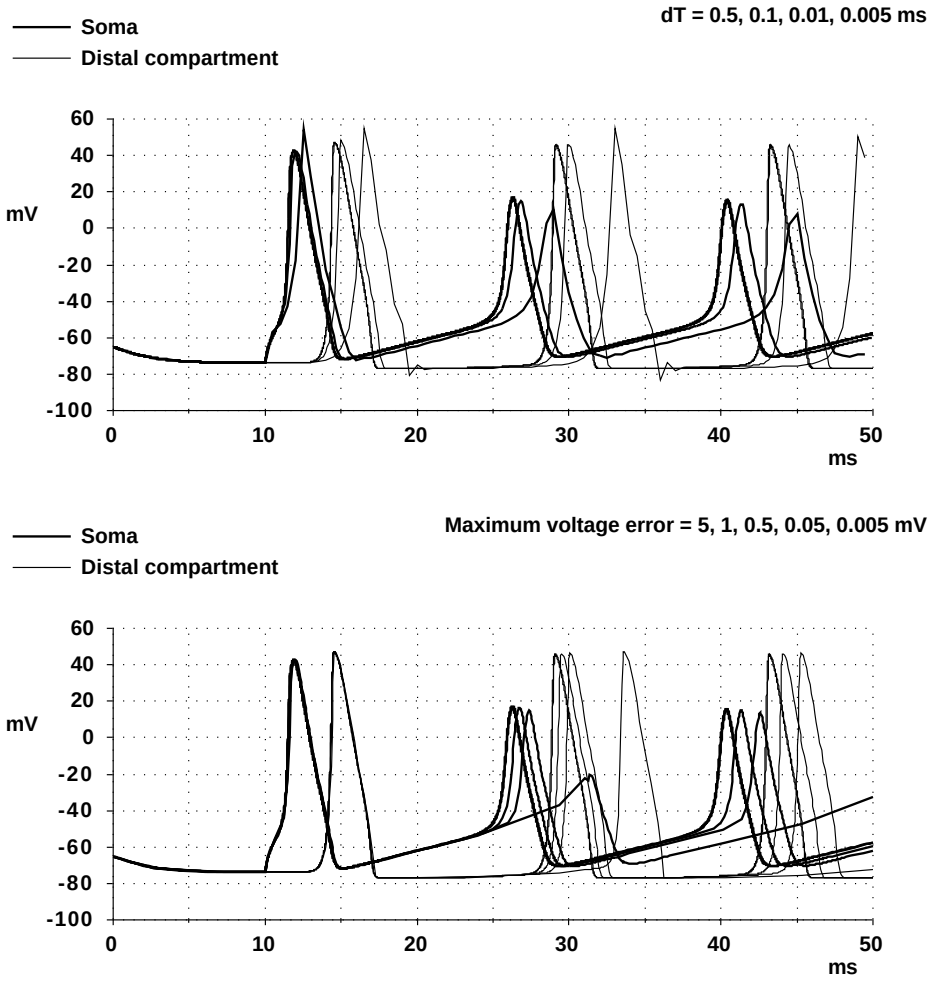

Figure 8: Comparison of convergence properties for repetitive firing during current clamp for different fixed time steps and adaptive time step error criteria. Top: Fixed time step simulations show convergence as the time step is reduced from 0.01 milliseconds (5,000 time points) to 0.005 milliseconds (10,000 time points). Bottom: Adaptive time step simulations, with only voltage error considered, show convergence as the maximum allowed voltage LTE ϵ^V_{max} is reduced from 0.05 mV (1,662 time points, 1,982 total iterations) to 0.005 mV (5,151 time points, 5,408 total iterations). Note that for both the fixed and adaptive time steps that the simulations are well-behaved (e.g. stable) for all parameter values.

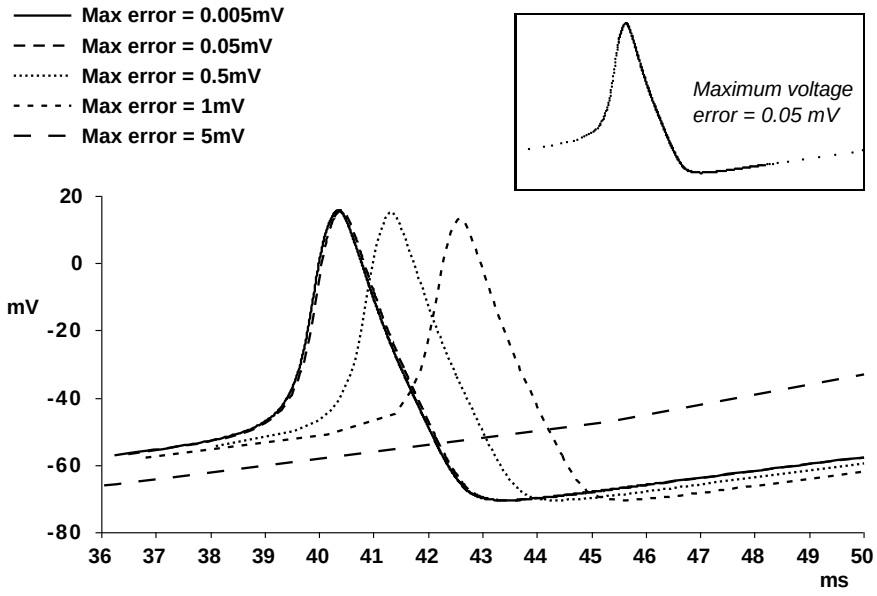

Figure 9: Detailed comparison of third somatic action potential in repetitive firing simulations shown in Figure 8, showing the results using various error criteria for the adaptive time step. Inset: Time points for adaptive time step with ϵ^V_{max} set to 0.05mV.

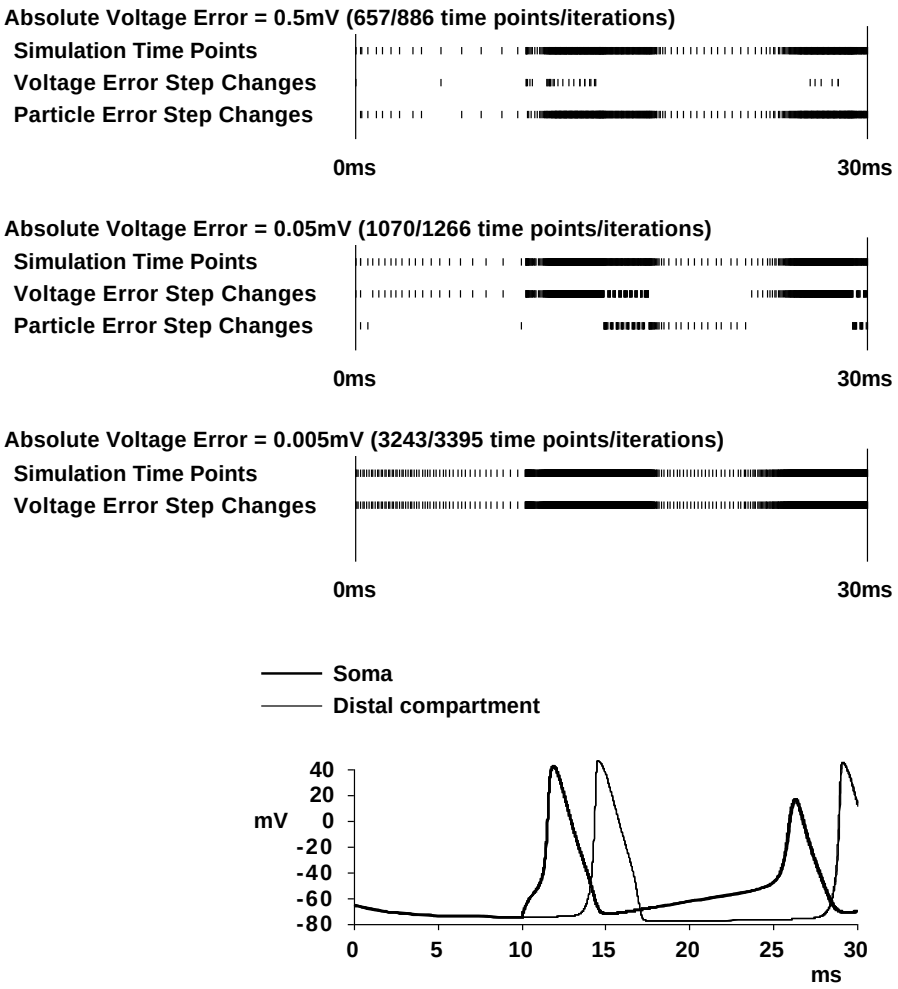

Figure 10: Adaptive time steps as a function of maximum allowed voltage LTE (ϵ^V_{max}), for a fixed maximum allowed particle LTE, ϵ^x_{max} ($= 0.001$), for the first 30 milliseconds of the current clamp protocol described in Figure 7. Raster plots in this and the following two figures compare the times of all time steps used in the simulations, with those time steps that are specifically determined by either the voltage error or the gating particle error, as appropriate. In addition, simulation output at the bottom is overlaid for each simulation described in the raster plots; in all cases these plots are indistinguishable.

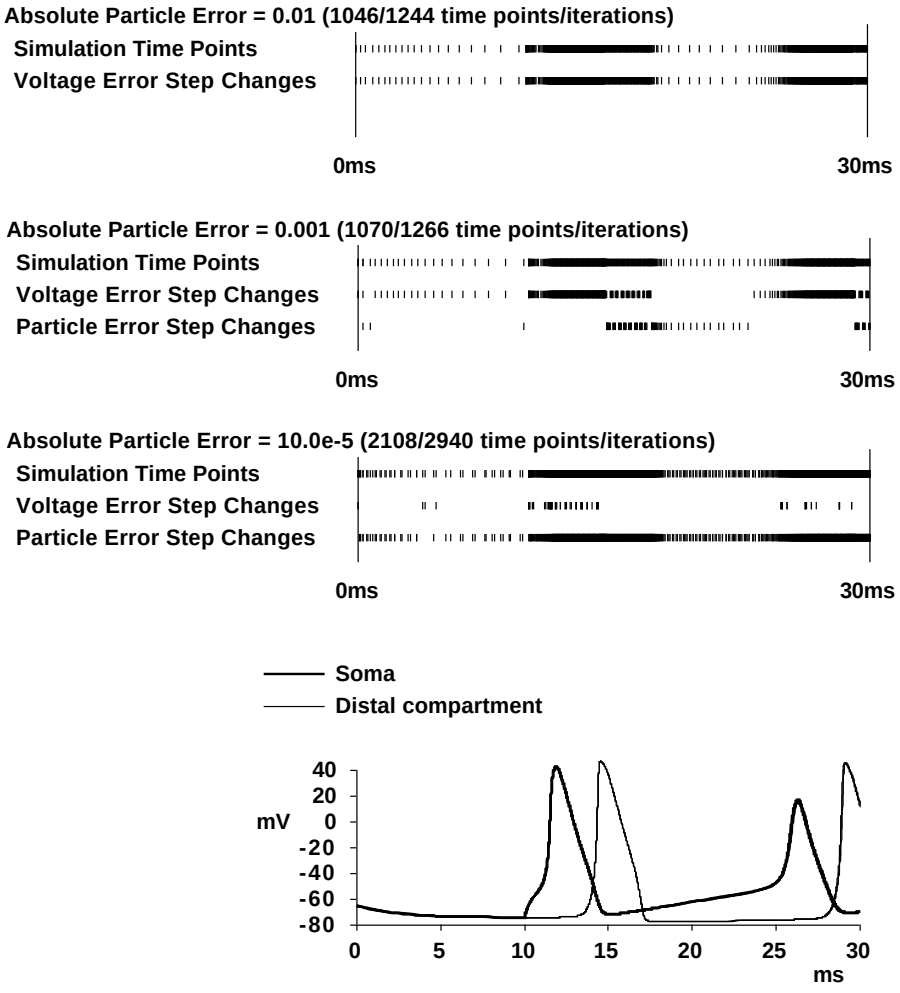

Figure 11: Adaptive time steps as a function of ϵ^x_{max}, for a fixed $\epsilon^V_{max} = 0.05$mV, for the first 30 milliseconds of the current clamp protocol described in Figure 7.

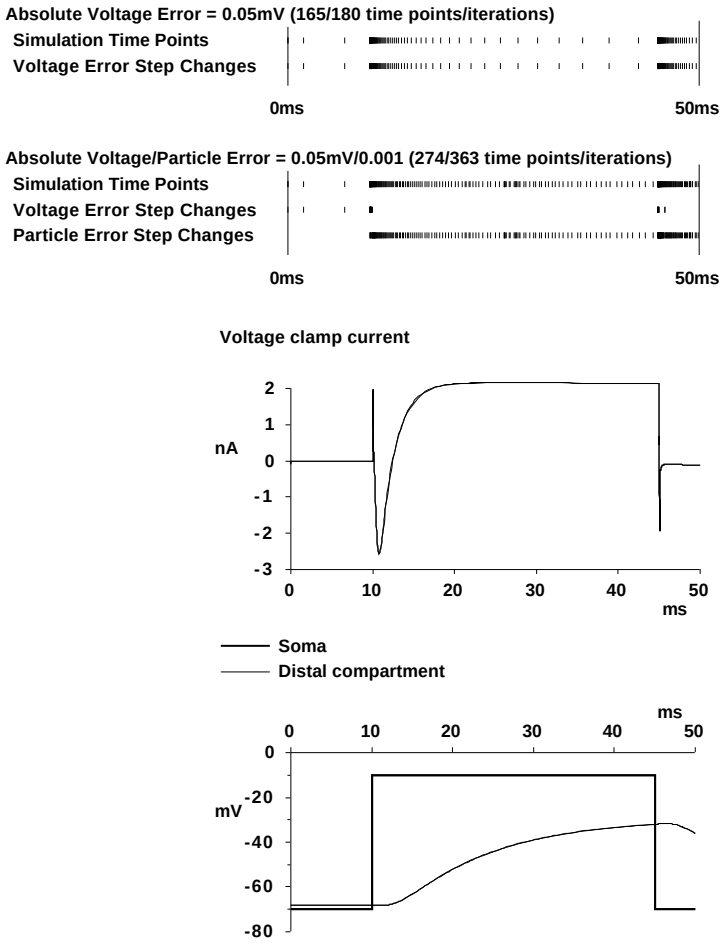

Figure 12: Time steps determined by voltage versus particle LTE for the -70mV to -10mV step of the (ideal) voltage clamp protocol described in Figure 7. In the top raster plot only voltage LTE ϵ^V_{max} ($= 0.05$mV) was used to determine time steps; as described in the text, the appropriate nodes for this calculation included both the clamped (soma) node and the adjacent node. This explains how the voltage LTE may be non-zero after the pulse transitions. In the lower raster plot, gating particle LTE ($\epsilon^x_{max} = 0.001$) were also considered in the determination of the time step. With the (default) values as given, the particle LTE determines all of the time steps subsequent to the pulse transition. Time course for the soma and distal compartment voltages and the clamp current are superimposed in the bottom plots for both error criteria, and are essentially identical.

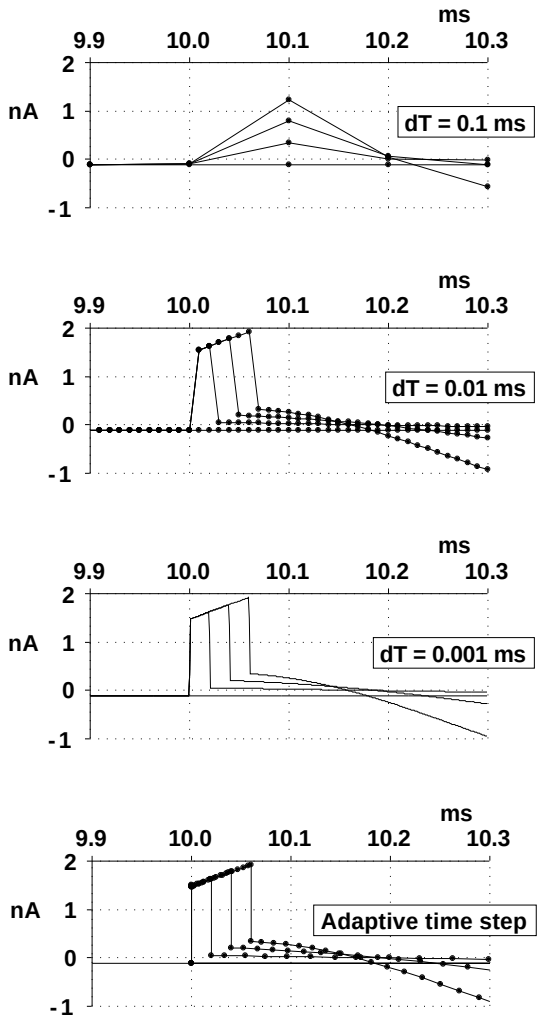

Figure 13: Capacitive transient currents under ideal voltage clamp with fixed and adaptive time step. In the adaptive time step case, breakpoints are used at the start and stop times of each transition of the voltage source pulse. The actual time steps are shown in each case except for the fixed step of 0.001 milliseconds. Note that the adaptive time step follows very well the transient response for the fixed step of 0.001 milliseconds. The time step prior to 10.0 milliseconds in the adaptive time step case occurred at 7.0 milliseconds.

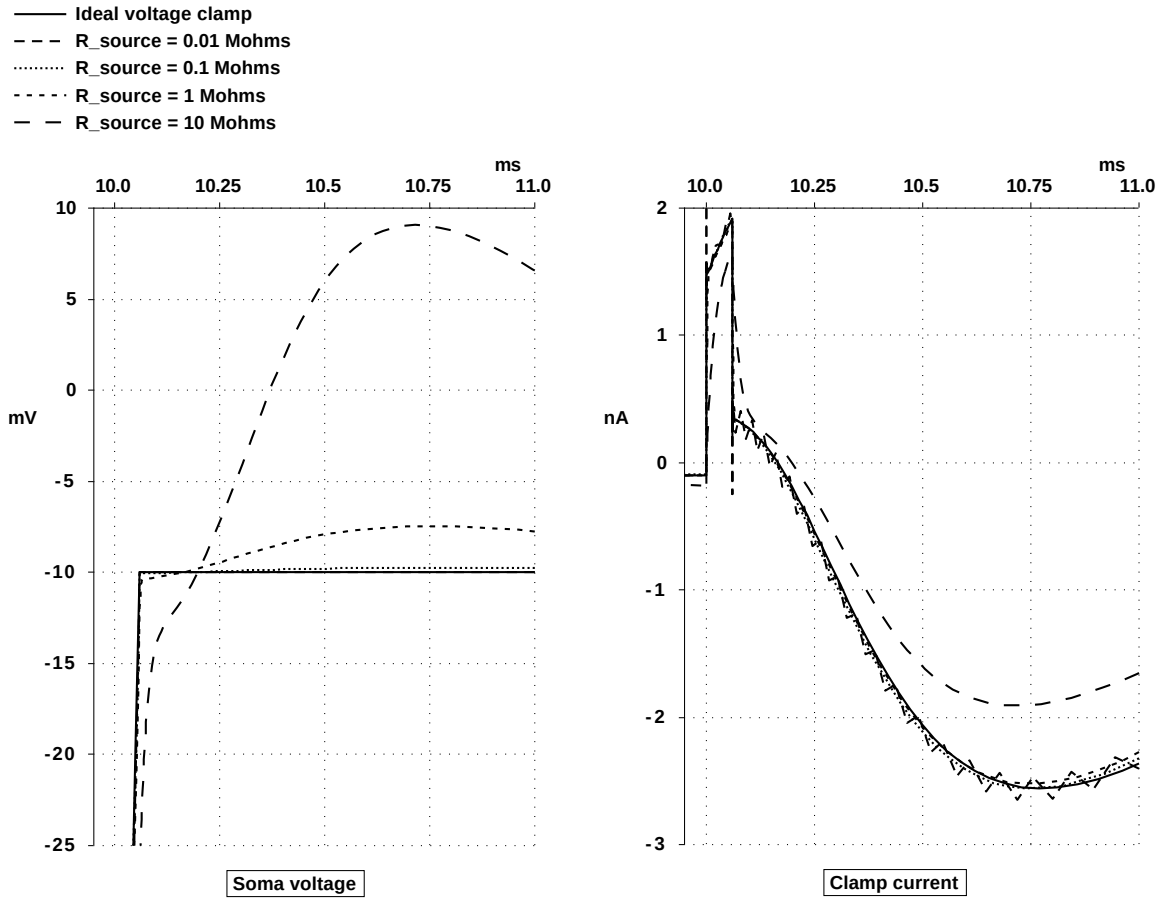

Figure 14: Voltages and currents under ideal and non-ideal voltage clamp, for the voltage step from -70 to -10mV, with an adaptive time step. The somatic voltage shown on the left in the non-ideal case for $R_{Source} = 0.01\text{M}\Omega$ is indistinguishable from the ideal case; however, this small value gives a small oscillation in the current record shown on the right.

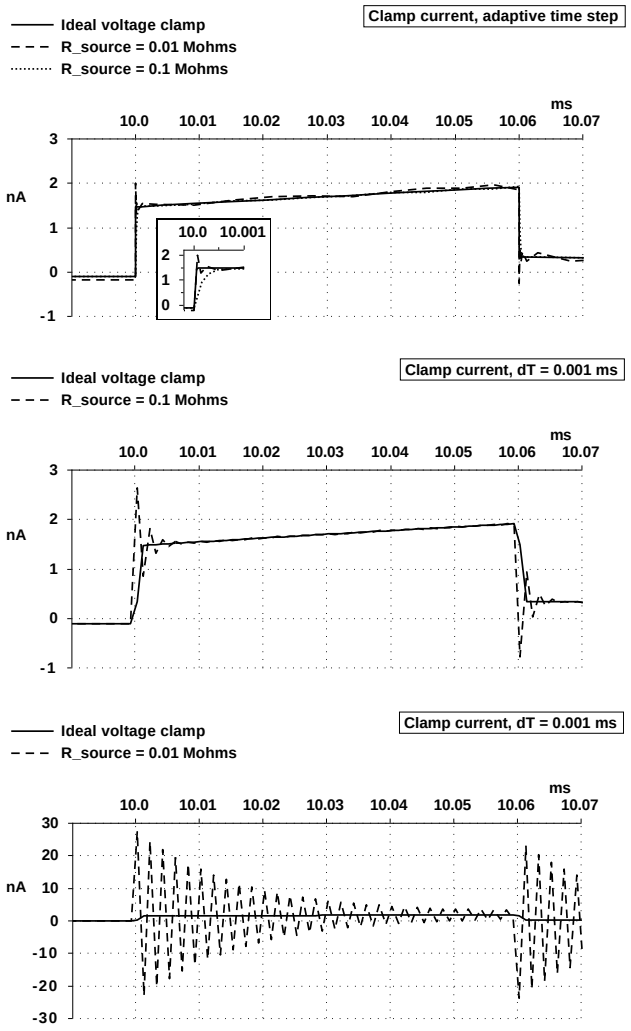

Figure 15: Capacitive transient under ideal and non-ideal voltage clamp, both fixed and adaptive time step. The traces at the top are a detailed view of those shown at the right in Figure 14; note again the oscillations seen when $R_{Source} = 0.01\text{M}\Omega$. In the inset at the top it can be seen that the current error for $R_{Source} = 0.1\text{M}\Omega$ during the transient is small. The sensitivity of the non-ideal voltage clamp current to R_{Source} is much more pronounced in the case of fixed time step integration, as seen in the middle and bottom traces.

Element	State	Dependent On	Controls
V-dep particles	Prob[open state] (0-1)	Segment/soma voltage	Channel conductance
Conc-dep particles	Prob[open state] (0-1)	Concentration Integrator	Channel conductance
Conc Integrator	[X] in Shells (mM)	Some current in soma or segment	Conc-dep particles

Table 7: The states of the above elements, along with the node voltages, correspond to the state variables of the circuit.

37 Surf-Hippo Integration - Numerical Methods

This section describes the numerical details of Surf-Hippo, in particular as related to the methods discussed in Sections 34 through 36.

37.1 Simulator Representation of Circuit Structure

Symbolically, circuits are composed of a set of object instances, of which there are several different types: nodes, segments, somata, channels, synapses, isources, particles, concentration integrators, etc. There are many different interrelationships between objects in a hierarchical fashion, e.g. an instance of a channel object references an instance of a node object (corresponding to the electrical circuit node that it is attached to), an instance of a segment or soma object that it is a part of, instances of particle and/or concentration particle objects that determine the channel's conductance, and, indirectly, instances of particle nodes which correspond to the particles' states.

Simulation of circuits is in terms of their nodes, which include the actual electrical nodes (corresponding to somata and segments) as well as elements whose states are both dependent on other nodes and in turn control either branch elements of the circuit or other (non-voltage) nodes. This latter class of elements are listed in Table 7.

37.2 Integration of the Circuit

For integrating the circuit equations, Surf-Hippo uses a variant of the method described by Hines (1984). Briefly, the method splits each time step into an implicit and explicit phase, and is equivalent to the Crank-Nicholson method. A major difference between the method described in this reference, and the Surf-Hippo method, is that Surf-Hippo uses a variable time step, whose size is adjusted during the integration by an estimate of the linear truncation error.

37.3 Determination of Time Step

There are three methods for determining the time steps, determined by the value of the global variable `*INTEGRATION-TIME-REFERENCE*`:

- :VARIABLE - Time step based on a LTE criteria
- :FIXED - Constant time step is set by the variable `*USER-STEP*`, in milliseconds
- :LIST - Time steps are taken from the list `*LAST-SIM-REVERSE-TIME-STEP-LIST*` of time steps (in milliseconds) generated by an earlier simulation

The :LIST method is used only if there are stored time steps which are valid for the current simulation (e.g. the stop times match). See the Section 37.9.

The adaptive time step method adjusts the step according to an estimate of the linear truncation error (LTE). The main parameters that control the mapping between LTE and the next time step include:

```
*ABSOLUTE-VOLTAGE-ERROR*  -> A reasonable range is 0.01 - 0.1 [mV]
*ABSOLUTE-PARTICLE-ERROR* -> A reasonable range is 0.001 - 0.01
*USER-MIN-STEP*           -> Can typically be set to 0.0 without danger [ms].
*USER-MAX-STEP*           -> A reasonable ragne is 0.5 - 5.0, which avoids
                             skipping over input events [ms].
```

These global variables may be set, starting from the Main Menu, via "Overall parameters, load circuit or files" → "Time step and numerical integration parameters".

The adaptive time step is discussed further in the sections Particle Errors, Voltage Errors, and LTE Estimation.

37.4 Breakpoints

Breakpoints are times for which the simulation is forced to use (for variable time steps only). In general a breakpoint is chosen because there is a priori knowledge of some input which begins or changes abruptly at that time. This makes less work for the adaptive time step, since without this information the initial stepping would more than likely enounter the input sometime after its initiation, and thus may be forced to backup if the input was too large at the first attempted time point that "saw" the input or the change in the input. The result (in general) with using breakpoints then is fewer overall iterations and a solution that is less likely to have "ringing".

Float or integer breakpoint times are collected prior to the simulation onto the global variable `*BREAKPOINT-LIST*` (the `PICK-TIME-STEP` function uses a version of this information in `*MRT-BREAKPOINT-LIST*`) with the following functions:

queue-breakpoint-time *time* [*Function*]

TIME is a breakpoint time in milliseconds.

queue-breakpoint-times *times* [*Function*]

TIMES is a list of breakpoint times in milliseconds.

queue-pwl-source-break-points *value−array period delay* [*Function*]

This is used for PWL (pulse-based) current and voltage sources.

In addition to breakpoints added for any pulse-based source, breakpoints are added (automatically) for the onsets of any autonomous processes, such as event-driven synapses.

Breakpoints are collected automatically. If there are additional breakpoints desired, then they must be put into the global variable `*USER-BREAKPOINT-LIST*`. `*USER-BREAKPOINT-LIST*` is cleared whenever a brand new circuit is read in. For example,

```
(push 15.0 *USER-BREAKPOINT-LIST*)
```

adds the time 15.0 ms to the breakpoints used by the next integration. New breakpoints may also be generated during a simulation with the evaluation of event-based elements, including axons and voltage-dependent synapses, depending on the value of:

enable-dynamic-breakpoint-generation *t* [*Variable*]

37.5 Time Step Global Variables

Note: To avoid consing, the single and double float globals which track the time steps are stored in arrays:

```
*time-single-float-variables*
*time-double-float-variables*
```

For example:

```
(defmacro *real-time* ()
  `(aref (the (simple-array single-float *) *time-single-float-variables*) 0))
```

Thus, the value for *REAL-TIME* is referenced as (*REAL-TIME*) in the code. This may not be much of a win.

During the integration, time is in units of the global variable *MRT*, or the Minimum Resolvable Time. This is done so that internally time may be kept as an integer; thus the value for *MRT* is used to convert floating times into integers. We should note that the original justification for the integer representation of time had to do with implementation in *LISP and concerns about truncation error, and it is not clear that there is a distinct advantage in the present code.

MRT is set at the beginning of the simulation:

```
(setf *MRT* (/ (float (max (abs *USER-START-TIME*) (abs *USER-STOP-TIME*)))
               MAX-INTEGER-TIME))
```

Note that *USER-START-TIME* is almost always equal to 0.

The constant MAX-INTEGER-TIME is set as:

```
(defconstant MAX-INTEGER-TIME (expt 2 (- TIME-WORD-LENGTH 2)))
```

and is the maximum integer used for internal times. From the original notes from Don Webber:

```
"The word length is subtracted by 2, one to allow for 2's
 complement math, and one for safety if a simulation runs
 over the stop time. I don't know if the second one is
 necessary."
```

Likewise, the constant TIME-WORD-LENGTH is defined as the number of bits in a integer used for internal times:

```
(defconstant TIME-WORD-LENGTH #-cmu 32 #+cmu 28)
```

This was also inherited from Webber's SURF package. This value is not guarranteed to be correct for all platforms. The +cmu figure of 28 was estimated by finding the type-of $2\hat{N}$, and N=28 was the largest value that gave a FIXNUM.

Now, we shall review the remaining global variables which are relevant to the time step. In general, only variables that include "USER" in the name should be set by the user.

- *USER-START-TIME* (default = 0.0), is the start time of the simulation, in milliseconds.
- *START-TIME* (default = 0), is the start time in units of *MRT*.
- *USER-STOP-TIME* is the time to end the simulation, in milliseconds.
- *INT-USER-STOP-TIME* is the integer part of the time to end the simulation, in milliseconds.
- *STOP-TIME* is the stop time in units of *MRT*.
- *USER-MAX-STEP* is the maximum time step allowed, in milliseconds. When 0, then MAX-STEP is bound by the simulation duration.
- *MAX-STEP* is the maximum time step in units of *MRT*.
- *USER-MIN-STEP* is the minimum time step allowed, in milliseconds. When 0, then *MIN-STEP* is set to *MIN-STEP-MRTS*.
- *MIN-STEP* is the minimum time step in units of *MRT*.
- *SIM-TIME-N+1* is the time for the step currently being computed, in units of *MRT*, i.e. t(n+1).
- *SIM-TIME-N* is the time for the step already computed, in units of *MRT*, i.e. t(n).
- *SIM-TIME-N-1* is the time for one step back, in units of *MRT*, i.e. t(n-1).
- *SIM-TIME-N-2* is the time for two steps back, in units of *MRT*, i.e. t(n-2).

- *TIME-STEP* is the current time step, in units of *MRT*.
- *LAST-TIME-STEP* is the last time step, in units of *MRT*.
- *REAL-TIME* is the time during simulation (msec), that is the end of the time step (corresponding to the prediction time of the data).
- (*T[N]*) is the value of *REAL-TIME* for the last time step. This value is used for, among others, axons and voltage dependent synapases who reference node voltages at the previous time step.
- (*INPUT-TIME*) is the time reference for inputs (msec). When the flag *EVALUATE-INPUTS-AT-MIDPOINT* is true, at each time step the circuit inputs (e.g. sources, driven synapses) are evaluated at the midpoint of the step - otherwise, the inputs are evaluated for the end of the step (the prediction time).
- *INTEGER-TIME* is the integer part (floor) of *REAL-TIME* (msec).
- *FRACTIONAL-TIME* is the fractional part of *REAL-TIME* (msec).
- *USER-TIME-STEP* is in units of *MRT*.

37.6 Particle Errors

The functions PARTICLE-ERROR-OK (if *CONSIDER-PARTICLE-ERROR* is T) and CALC-LTE-RATIO calculate the LTE from the gating particles and node voltages, respectively.

When *CONSIDER-PARTICLE-ERROR* is T, prior to the evaluation of the gating particles (both voltage and concentration dependent) the variable *MAXIMUM-PARTICLE-ERROR-NUMERATOR* is set to 0. As each particle is evaluated, *MAXIMUM-PARTICLE-ERROR-NUMERATOR* is set to the numerator of the 2nd order LTE (a difference between the current and last first derivatives of the particle state) when this value exceeds the current value of *MAXIMUM-PARTICLE-ERROR-NUMERATOR*. When all the gating particles have been evaluated, the value of *MAXIMUM-PARTICLE-ERROR-NUMERATOR* is used to estimate the LTE. The result is then used to calculate *PARTICLE-ERROR-MAX-TIME-STEP* (in units of *MRT*). If this value is greater than the current time step, a flag in DO-TIME-CONTROL (HINES-STEP-OK) is set to NIL.

37.7 Voltage Errors

After the node voltages have been computed, CALC-LTE-RATIO returns the ratio of the allowed error (*ABSOLUTE-VOLTAGE-ERROR*) divided by the estimate of max error. Note that in the equations below there are occasional references to the node structure slots that hold certain variables.

At any given time during the simulation, the node with the largest error must be one that has some input associated with it, e.g. a source, channel, or synapse. All other nodes are driven by their neighboring nodes, and since the capacitance of each node is non-zero, then the response of a node driven by the voltage of a neighbor will always be slower than the neighbor's voltage. Therefore, for estimating the maximum LTE, we need only consider nodes with inputs.

Thus at the beginning of the simulation the function MAKE-NODE-W/ELEMENTS-ARRAY sets the global array *NODE-W/ELEMENTS-ARRAY* to point to all nodes with active (e.g. gbar>0) membrane elements. It is these node that are considered in estimating the maximum LTE in CALC-LTE-RATIO. Note that CALC-LTE-RATIO also updates the voltage derivative, which is used for the particle evaluation. Since all nodes with particles (channels) are included in *NODE-W/ELEMENTS-ARRAY*, then these derivatives will be available for the proper nodes.

The inclusion of all nodes with sources may be modified if the global variable *INCLUDE-VSOURCE-NODES-IN-NODE-ERROR-EST* (default nil) is nil. In this case, if there is more than one node in the circuit the node with the voltage source is *not* included in the error estimation. This can avoid very small steps due to a fast voltage source. In practice, this can greatly speed up the simulations without effecting the result. However, it is a good idea to verify the simulation results by an occasional run with any voltage source node included in the error estimation. It is also a good idea to include any voltage source nodes in the error estimation if the source impedance is large.

37.8 LTE Estimation

The electrical circuit nodes (corresponding to somas and segments) which are considered in the LTE estimate are determined by the value of the global variable

lte-node-criterium *:all* [*Variable*]

Options for this variable include :ALL (default), :SOMAS, :CHANNELS, :SYNAPSES, :VSOURCES, :ISOURCES, :AXONS, or a list of circuit elements that may or may not include the afore-mentioned keywords. If :ALL, then include all circuit nodes with externally-driven elements (e.g. sources or synapses or channels). If :SOMAS, then include only somas. If :CHANNELS, :SYNAPSES, :VSOURCES, :ISOURCEs or :AXONS, include only those nodes with the appropriate elements.

Note that if a driving element on a cell element is specified via `*LTE-NODE-CRITERIUM*`, then the associated node will only be considered if the membrane element is enabled (i.e. not blocked), and has a non-zero absolute conductance (for synapses and channels).

Loop over the circuit nodes to be considered in the error estimate, finding the largest estimated O(dt$\hat{2}$) error using the following estimate for the second derivative of the voltage. If either the voltage LTE ratio is less than 1, or if the particle error to too large (`HINES-STEP-OK` is NIL) then we redo the last time step. Thus `PICK-TIME-STEP` will get as its `CURRENT-TIME` argument:

```
SIM-TIME-N+1  (Last step successful)
```

or

```
SIM-TIME-N    (Last step unsuccessful)
```

This argument is needed so that `PICK-TIME-STEP` can check for special time points (e.g. breakpoints) and other conditions which constrain what the next time step can be. The basic calculation by `PICK-TIME-STEP` is derived from the second order term in the Taylor expansion of the voltage:

```
NEW-TIME-STEP = PREV-STEP * (expt LTE-RATIO 0.5) * *PICK-TIME-STEP-FUDGE
```

If we are considering particle error, then NEW-TIME-STEP will then be limited by `*PARTICLE-ERROR-MAX-TIME-STEP*`. Note that `*PARTICLE-ERROR-MAX-TIME-STEP*` was calculated in `PARTICLE-ERROR-OK` by essentially the same formula as above.

For non-linear circuits the fudge factor (`*PICK-TIME-STEP-FUDGE*`, typically set to 0.8) helps ensure that a reduced time step (when LTE-RATIO is less than but close to 1) is small enough so that the next try (iteration) yields new LTEs that are within the error criteria. This is because the LTE for a non-linear circuit is not proportional to the time step, and thus the new step may not be small enough. In practice, this is a problem with voltage-dependent channels.

`*PICK-TIME-STEP-FUDGE*` must be less than or equal to 1. As an indication that `*PICK-TIME-STEP-FUDGE*` is not small enough, check the "Total time points/iterations" figure at the end of a simulation. If the LTE time step estimation is working properly, there should be no more than one (extra) iteration per time point. Typically, of course, there should be many time points in which the initial time step is fine, and in fact is increased for the next time point. But as a general rule of thumb, if the number of iterations is more than twice the total time points, then a possiblility is that `*PICK-TIME-STEP-FUDGE*` is too large.

For more detailed tracing of the time step iterations, try watching the evolution of the simulation with the `*DEBUG-TIME-TRACE*` flag enabled.

The circuit matrix integration and the particle state integrations described below take into account the variable time step, which is an extension of the method described by Hines 1984.

The `*USER-MAX-STEP*` constraint may be overruled by the evaluation of some element types, in particular those that are driven by an a-priori (external) waveform. In these cases, the global variable `*ELEMENT-TIME-STEP-MAXIMUM*` is set to a non-NIL step value (in units of `*MRT*`) that matches the (smallest) time interval of the waveform(s). For example, if `EVAL-AXON` finds itself in a spike, then the following form is evaluated:

```
(when (or (and *ELEMENT-TIME-STEP-MAXIMUM*
               (> *ELEMENT-TIME-STEP-MAXIMUM* waveform-interval))
          (not *ELEMENT-TIME-STEP-MAXIMUM*))
  (setf *ELEMENT-TIME-STEP-MAXIMUM* waveform-interval))
```

At the beginning of every time step, *ELEMENT-TIME-STEP-MAXIMUM* is set to NIL. This mechanism ensures that the complete detail of an applied waveform is captured by the simulation.

There are two lower limits on the internal (integer) time step - *USER-MIN-STEP* (in units of ms, when not set to 0) and 2. When the LTE estimate is too large despite using the minimum step, if

```
*PUNT-LTE-WHEN-MIN-STEP-REACHED* = T
```

then the simulation will keep going anyway, and the *LTE-WAS-PUNTED* flag will be set. Otherwise, the simulation will stop and a message will appear, e.g. (if *USER-MIN-STEP* = 0):

```
Integration stuck at time 4.353ms [internal integer time step = 2].
  Try either increasing the error criteria *ABSOLUTE-VOLTAGE-ERROR*, which is now 0.001,
  or reducing the simulation duration.
```

Or, (if *USER-MIN-STEP* is not = 0):

```
Integration stuck at 4.353ms, try reducing *USER-MIN-STEP*, which now is 0.1ms.
```

Note that the lower limit of 2 is imposed because the evaluation at each time step is done in two stages (which split the time step into two).

37.9 Shadowing Time Steps From a Previous Simulation

At the end of every simulation, the time steps (in units of *MRT*) are stored in reverse order in *SIM-REVERSE-TIME-STEP-LIST*. These time steps may be used, or "shadowed", by later simulations to direct the integration.

Time step shadowing is enabled when the global variable *INTEGRATION-TIME-REFERENCE* is set to :LIST, and certain conditions are met.

By default, shadowed time steps are taken from the list *LAST-SIM-REVERSE-TIME-STEP-LIST* when these values are consistent with the current simulation (primarily, the stop times are the same).

If *INTEGRATION-TIME-REFERENCE* is set to :LIST, but the time steps in *LAST-SIM-REVERSE-TIME-STEP-LIST* are not appropriate, and the stop time of the last simulation and the current simulation is the same, then *LAST-SIM-REVERSE-TIME-STEP-LIST* is set to the time steps from the last simulation which are subsequently used for the current simulation. Otherwise, time step shadowing is cancelled.

If *AUTO-REFRESH-LAST-SIM-REVERSE-TIME-LIST* is set then *LAST-SIM-REVERSE-TIME-STEP-LIST* is set to the time steps from the last simulation.

While the conditions describe above is designed to check for inconsistencies, it is not a bad idea to verify that the time steps used are the right ones.

Shadowing the time steps from a previous simulation is useful when doing precise quantitative comparisons between simulations using variable time steps. Although traces from any two simulations may be compared by using the interpolating function CONVERT-DATA-TIME-LISTS, there are some cases in which the inevitable noise generated by this method can be irritating (for example when analyzing small differences between two traces). See the voltage clamp protocol in src/sys/protocols.lisp for an example.

37.10 Ordering the Matrix

A fundamental contribution of Hines is recognizing that tree circuit topologies (valid for any arbitrary network of neurons, without gap junctions) can be ordered such that the resulting matrix is almost tri-diagonal with far off-diagonals that may be eliminated in one pass. The result is a matrix inversion that is $O(n)$ (see the description under HINES-SOLVE below).

The following description of branches is derived from Hines, 1984: A branch is composed of connected segments. The ends of the branch are determined by those segments which (at either end of the segment) connect to more than one segment. The branches of a cell, and in turn the branch segments, are numbered as follows: Choose any branch of the tree which is connected at one end to the soma. Number the segments of that branch so that the segment connected to the soma is the last segment. The first segment of this branch ('trunk') will be called the 'branch node'. Note that this branch node is not the same thing as a true circuit node (as referenced by Surf-Hippo).

The soma is also considered a 'branch node'.

All branches connecting to a branch node have their segments numbered so that their last segment connects to this node. Their first segments are also called branch nodes (as long as other segments are connected to them), and this segment-numbering process continues until all the segments are numbered. The last numbered branches, or 'twigs', all have one end (their first segments) unconnected. Each branch node becomes the center of a Wye network.

Continue this procedure for all other segments connected to the soma.

Branches are numbered as follows: Assume that there are N branches. Starting at soma, number all the branches connecting to the soma starting with N and decrementing. Continue numbering all the branches that are connected to the previous set of branches, decrementing the branch number. Continue working out on the tree until all the branches are labeled.

The two circuit nodes (proximal and distal) associated with each segment are ordered according to the branch number and the number of the attached segment in the proximal direction. Note that since the circuit is a tree, there will only be one such segment for each node, whereas there can be multiple segments attached to a node in the distal direction. For example, if branch 33 has 2 segments and branch 34 has 3 segments, then the nodes would be ordered: ...33-1, 33-2, 34-1, 34-2, 34-3... The soma node has the highest node index.

37.11 Solving the Circuit

Integration of the various gating particles which contribute product terms to G_{elts} is done prior to the computation of the matrix given in Equation 8 (see discussion of `EVAL-ALL-ELEMENTS` below).

Non-ideal voltage sources are handled as a very large membrane conductance G_{elt} (which corresponds to the series resistance of the electrode) in series with a controlled $E_{G_{elt}}$ (battery) (see also the Section 37.13).

Since the axial resistances are constant, we can fix the off diagonal terms before the simulation integration, where the g_{axials} are taken between the nodes implied by the matrix indices corresponding to the elements in the `*UPPER-DIAG*` and `*LOWER-DIAG*` arrays. These are referred in Equation 8 as the conductances which make up the sum G_{ik}, $(i \neq k)$.

```
 *upper-diag* and *lower-diag* = (-) G-axials    (EVAL-SEGMENT)
```

`V-AT-HALF-STEP` node structure slots and matrix arrays refer to $V(t + dt/2)$.

As pointed out by Hines, solving for $V(t + dt/2)$ in the implicit step, and then $V(t + dt)$ explicitely is second order correct with respect to both dt and the compartment length, and is numerically stable (Cooley and Dodge, 1966; Joyner et al., 1978).

The variable `*EVALUATE-INPUTS-AT-MIDPOINT*` controls the relationship between the input timing and the time step. When true, at each time step the circuit inputs (e.g. sources, driven synapses) are evaluated at the midpoint of the step - otherwise, the inputs are evaluated for the end of the step (the prediction time).

37.12 How The Code Does It, Briefly

(This description refers to an earlier release, but is still more or less relevant)

The main loop is in `DO-TIME-CONTROL`. The following functions are called directly from do-time-control at the start of the integration, and/or from within `HINES-STEP` during the integration. This is a condensed description; for the exact details please see the referenced code.

At the beginning of each time step:

```
(INITIALIZE-INTEGRATION)
```

Evaluate the various time step variables:

```
(setf (*delta-t[n-1]*) (*delta-t[n]*))      ; Advance time step.
(setf (*delta-t[n]*) (* *mrt* *time-step*)) ; New time step.

(setf (*delta-t-prime[n]*) (* 0.5 (+ (*delta-t[n-1]*) (*delta-t[n]*))))
(setf (*delta-t-prime[n]-squared*) (* (*delta-t-prime[n]*) (*delta-t-prime[n]*)))
(setf (*half-delta-t-prime[n]*) (* 0.5 (*DELTA-T-Prime[N]*)))
```

Now update the current sources.

```
(SET-SOURCES)
```

Voltage sources are handled by `EVAL-ALL-ELEMENTS`.

```
(INIT-ALL-NODES)
```

Clear tri-diag matrix, and initialize all the accumulator fields, including (NODE-JACOBIAN ND), (NODE-ALPHA-CHARGE ND) and (NODE-CURRENT ND), i.e.:

```
(setf (node-jacobian nd) (node-const-jacobian nd)
```

```
(UPDATE-NODES-V-INDEX)
```

The :PRT-V-INDEX slot in each node is derived by an estimate of the voltage at time (t) halfway between midpoints of last step and current step. Depending on size of these two steps, V(t) is either an interpolation between or an extrapolation beyond V(t-n-1) and V(t-n). This estimate is then translated to an index for the alpha and beta rate constant arrays.

```
(EVAL-ALL-ELEMENTS)
```

Run all the eval routines for the circuit elements, each of which accumulates the jacobian, alpha-charge, current entries of the appropriate node, and accumulates the near off-diagonal entries of the tri-diag matrix, as appropriate. In some cases, the order in which different types of circuit elements is evaluated is important, for example, particles must be evaluated before channels. The order is as follows:

```
(EVAL-ALL-AXONS)
(EVAL-ALL-CONC-PRTS)
(EVAL-ALL-PRTS)
(EVAL-ALL-CHS)
(EVAL-ALL-SYNS)
;; Otherwise, conc-ints are evaluated out of the inner loop, in DO-TIME-CONTROL.
(WHEN *EVAL-CONC-INTS-IN-INNER-LOOP*  (EVAL-ALL-CONC-INTS))
(EVAL-ALL-SOMAS)
(EVAL-ALL-SEGMENTS)
(EVAL-ALL-ISOURCES)
(EVAL-ALL-VSOURCES)
```

For example in `(EVAL-SEGMENT SEG)`: First, a local variable ALPHA-CAP is set -

```
alpha-cap <- (* 2/dt (segment-capacitance seg))
```

For a segment, the target node is the distal node (node-2). The :FLOATS slot of a node holds the variables that are continually updated during the integration (the :FLOATS slot of the various circuit element structures holds an array of the variables that are updated by the integration step - this is a kludge to avoiding CONSing):

```
node-2-floats <- (node-floats (segment-node-2 seg))
```

Then, the node's ALPHA-CHARGE, CURRENT, and JACOBIAN slots are updated:

```
(setf (node-floats-aref-alpha-charge node-2-floats)
        (+ (node-floats-aref-alpha-charge node-2-floats)
           (* (node-floats-aref-voltage-n node-2-floats) alpha-cap)))

(setf (node-floats-aref-current node-2-floats)
        (- (node-floats-aref-current node-2-floats)
           (* (segment-g-leak seg) (segment-v-leak seg))))

(setf (node-floats-aref-jacobian node-2-floats)
        (+ (node-floats-aref-jacobian node-2-floats)
           alpha-cap))
```

The particles are evaluated first: the particle states need to be integrated since the state kinetics are described by non-linear first-order differential equations. However, as pointed out by Hines, there is a non-zero delay between a change in a node voltage and the subsequent effect on a node channel conductance (since the time constants for the channel kinetics are always non-zero). Therefore, the particle state equations may be solved independently of the circuit matrix evaluation, with a completely (direct) explicit method. Particle states are evaluated at (t + dt/2).

When all the elements are evaluated, that is their contributions to their associated circuit nodes have been accumulated, then for each the circuit node:

```
The node (ALPHA-CHARGE - CURRENT) -> RHS (Right Hand Side [of matrix equation])
The node JACOBIAN -> diag
```

```
(EVAL-ALL-NODES)
```

Solve both implicit and explicit Hines step: Set the RHS, solve the matrix (implicit step), update the voltage estimate with the new V-at-half-steps (explicit step).

```
Implicit phase:

  For every electrical node -

(SET-DIAG-RHS-FLOATS) [calls SET-RHS]

    RHS = (2/dt * capacitance * V(t)) + (G-elements * E_G-elements)

    diag = Jacobian = (2/dt * capacitance) + G-axials + G-elements


(HINES-SOLVE)
```

Upper triangularization of the matrix is accomplished by repeated applications of the function `TRIANG`. Define a 'twig' as any branch whose distal end is *not* connected to an un-`TRIANG`ed branch (including no branch at all). Now upper triangularize the matrix by doing the following until there are only twigs left: Collect the set of all current twigs, except any twigs which have already been `TRIANG`ed. Apply `TRIANG` to the remaining set of twigs, in the order that they appear in the matrix.

```
(TRIANG)
```

Then, for all the somas:

$$V_{soma_i}(t + dt/2) = RHS_i/diag_i$$

where i refers to the matrix index for $soma_i$.

Now back substitute to complete the Gaussian elimination.

```
(BKSUB)

 Explicit phase:

(EVAL-NODE-FLOATS)
```

Over all nodes:

$$V(t + dt) = ((2 \times V(t + dt/2)) - V(t))$$

Now, if using a variable time step, check the LTE and choose a new time step.

```
(CALC-LTE-RATIO)

  Get LTE-RATIO =  *ABSOLUTE-VOLTAGE-ERROR* / estimated LTE

(PICK-TIME-STEP)
```

If the estimated LTE is 0, then CALC-LTE-RATIO returns 0 (instead of infinity), and PICK-TIME-STEP interprets this "special" value of LTE-RATIO appropriately (constraining the new time step by *MAX-STEP*).

If $LTE - RATIO > 1.0$ (or $= 0$), then advance integration, otherwise rerun last step with smaller time step.

If last step was ok, then:

```
(SAVE-DATA)
```

37.13 Ideal Voltage Sources

Relevant functions include the following. For stripping out ideal voltage source nodes from the branches which form the circuit matrix:

- BUILD-BRANCH-LIST

For evalation of the ideal voltage sources:

- CHECK-FIXED-VOLTAGE-NODES
- GET-VSOURCE-CURRENT
- EVAL-FIXED-VOLTAGE-NODES

For setting up the matrix:

- REORDER-CIRCUIT
- ADD-SEGS-TO-OFF-DIAGS
- INITIALIZE-SEG-NODE-JACOBIANS-AND-CURRENTS
- INITIALIZE-SOMA-NODE-JACOBIANS-AND-CURRENTS

For choosing the circuit elements that need to be evaluated:

- MAKE-SEGMENT-ARRAY
- MAKE-SOMA-ARRAY

For solving the matrix:

- TRIANG
- BKSUB
- HINES-SOLVE

37.14 Rallpacks - Neuron Simulator Benchmarks

Surf-Hippo code for running the Rallpack benchmarks (Bhalla et al., 1992; downloaded code in the rallpack_v1.1 directory) may be found in the src/rallpack directory.

38 Installation Notes

This section describes the basic installation of the Surf-Hippo package, under Unix (or a variant thereof). Here it is assumed that csh is the Unix shell. Thus, the "startup" file referred to below can be your .login file, your .cshrc file, or other file if you use another shell.

If you are running Surf-Hippo with the complete Lisp image file, the following instructions are complete. If you will be installing the Lisp and Garnet separately, refer also to Section 39.

38.1 File Installation

To install the Surf-Hippo code, create a directory called "surf-hippo" wherever you want the system to be, and go to it (this will be the Surf-Hippo home directory):

```
unix-prompt> mkdir surf-hippo
unix-prompt> cd surf-hippo
```

Now, ftp to either cogni.iaf.cnrs-gif.fr or ftp.ai.mit.edu. When asked to log in, use "anonymous", with your email address as the password:

```
unix-prompt>ftp ftp.ai.mit.edu
Connected to ftp.ai.mit.edu.
220 mini-wheats FTP server (SunOS 4.1) ready.
Name (ftp.ai.mit.edu:lyle): anonymous
331 Guest login ok, send ident as password.
Password:lyle@ai.mit.edu

230 Guest login ok, access restrictions apply.
```

Now, change to the surf-hippo directory at the ftp site and set binary transfer mode:

```
ftp> cd /pub/surf-hippo
250 CWD command successful.
ftp> bin
200 Type set to I.
```

The files have all been combined into compressed tar format files which will create the appropriate subdirectories automatically. Source code plus complete exeutable image files are in tar files that refer to the Surf-Hippo version, the machine architecture, and the operating system. If you are going to use a CMUCL and Garnet that is already installed at your site, just get the source code tar files, with names like surf-hippo.x.x.tar.Z. Here we will assume that you are installing the entire system, e.g.:

```
ftp> get surf-hippo.2.7e.x86.linux.17f.tgz
```

Now quit FTP:

```
ftp> quit
```

And uncompress and untar the files:

```
unix-prompt> uncompress surf-hippo.2.7e.x86.linux.17f.tgz
unix-prompt> tar -xvf surf-hippo.x.x.linux.17f.tar
```

These steps will create subdirectories will all the sources in them. You can now delete the original tar file:

```
unix-prompt> rm *.tar
```

38.2 Basic System Setup

Edit the "surf-hippo" executable shell script in the Surf-Hippo home directory to specify the lisp executable that is appropriate for your platform (see also surf-hippo/cmucl/README). The large image file is referenced by this script - make sure that the name (default "image") is correct in the script. Note that *both* the Lisp executable and the image file are specific to a given architecture and operating system.

38.2.1 Editing the startup file

Now edit your shell startup files - e.g. add the following to your .login or .cshrc file (this is for the Lisp):

```
unlimit datasize
unlimit stacksize
```

Add the pathname of the Surf-Hippo home directory to your startup file, via the SURFHOME environment variable. For example:

```
setenv SURFHOME /usr/local/surf-hippo
```

If needed, add the pathname of where data will be accessed (rw) (otherwise the value assigned to SURFHOME is used):

```
setenv SURFUSERHOME /home/someone-else/surf-hippo-stuff
```

If you do not set these environment variables, the defaults for both are:

```
~you/surf-hippo
```

If you want to use the ILISP interface of Emacs, and ILISP is not already installed on your system, then include a pointer to the ILISP files that come with Surf-Hippo, for example:

```
(setq load-path
      (cons (expand-file-name "/usr/local/surf-hippo/misc/ilisp/") load-path))
(load "/usr/local/surf-hippo/misc/ilisp/ilisp.el")
```

Note: when you run CMU Common Lisp, if you get an error similar to:

```
mapin: mmap: Invalid argument
ensure_space: Failed to validate 67108864 bytes at 0x01000000
```

then you might have to setenv CMUCL_EMPTYFILE in your startup file, e.g:

```
setenv CMUCL_EMPTYFILE /tmp/empty
```

See the CMUCL installation instructions for more information. Another common problem is not setting the DISPLAY environment variable properly, giving something like:

```
CMU Common Lisp 17f, running on neuro16
Send bug reports and questions to your local CMU CL maintainer, or to
cmucl-bugs@cs.cmu.edu.
Loaded subsystems:
    Python 1.0, target SPARCstation/Sun 4
    CLOS based on PCL version:  September 16 92 PCL (f)
    CLX X Library MIT R5.02
*** Restarting image created with opal:make-image ***
*** Image creation date: Aug 18, 1996, 5:58 AM ***

Error in function XLIB::HOST-ADDRESS:  Unknown host ""

Debug  (type H for help)

(XLIB::HOST-ADDRESS "" 0)
0]
```

Apparently, the CLX is not smart enough to choose an appropriate default if DISPLAY is not bound. Make sure it is set from the shell or startup file, for example:

```
unix-prompt>setenv DISPLAY `/usr/bin/hostname`:0.0
unix-prompt>echo $DISPLAY
cogni.iaf.cnrs-gif.fr:0.0
```

38.2.2 Editing the .emacs file

If you are going to run under ILISP, and you have downloaded the image, then set up a "surf-hippo" ILISP command in your Emacs setup file (e.g. /.emacs) to point to the image:

```
(autoload 'cmulisp "ilisp" "Inferior CMU Common LISP." t)
(defdialect surf-hippo "CMU Common LISP with Surf-Hippo" cmulisp)
(provide 'ilisp-surf-hippo)
(autoload 'surf-hippo "ilisp" "Inferior CMU Common LISP with Surf-Hippo." t)
(setq surf-hippo-program "/usr/local/surf-hippo/surf-hippo")  <----- Edit
                                                              this line to
                                                              wherever you
                                                              have installed
                                                              Surf-Hippo.
```

If you are running Xemacs, you may have to alter these initializations.

38.2.3 Other setup edits

In general, CMUCL GC (garbage collection) requires that the swap space be at least twice as large as the heap. Also, Lisp starts to thrash when the heap size gets significantly bigger than physical memory. For our machine (96MB), Lisp crashes when the heap gets around 41M (assuming that nothing else is running). Edit the following line in `misc/loaders/surf-hippo-loader.lisp` according to your setup:

```
(defvar surf::*gc-bytes-retained-warning-threshold* 40e6)
```

Check Section 39 and `misc/loaders/surf-hippo-setup.lisp` and `misc/loaders/surf-hippo-loader.lisp` for additional site initializations. Look for the `$$$$ CUSTOMIZE $$$$` string. CMU Lisp is industrial strength but not bullet-proof. Critical results should be saved at reasonable (roughly hourly) intervals.

38.3 Up and Running

In this section, which in some ways recapitulates Section 4, the Unix environment variable SURFHOME and surf-hippo/ will be used interchangeably as a reference to the Surf-Hippo home directory. There are two ways to load Surf-Hippo:

- (Fastest) Running the Image - If you have downloaded an image version, or made one yourself, run the Surf-Hippo CMUCL image using either the executable command:

  ```
  unix-prompt> surf-hippo
  ```

  (assuming that SURFHOME is in the Unix PATH), or from ILISP under Emacs: Type M-x surf-hippo in Emacs. Either method loads and executes the Surf-Hippo image file. Once Lisp starts, the Surf-Hippo environment is automatically initialized; you can then enter:

  ```
  lisp-prompt> (surf)
  ```

  to start the menus.

- Loading explicitly into Lisp - Run CMUCL from the Unix shell or from ILISP (to run CMUCL under ILisp, type M-x cmulisp in Emacs). Once Lisp has started, you then have to load Garnet and Surf-Hippo explicitly by loading the file `misc/loaders/main-loader.lisp`. For example:

  ```
  lisp-prompt> (load "/usr/local/surf-hippo/misc/loaders/main-loader.lisp")
  ```

  or from ILISP, use C-z l on this file (no need to compile it first). If you need to recompile the entire system, then load the file `$SURFHOME/misc/loaders/main-compiler.lisp` into Lisp (start by running CMUCL, not Surf-Hippo), for example:

```
lisp-prompt> (load "/usr/local/surf-hippo/misc/loaders/main-compiler.lisp")
```

or from ILISP, use C-z l on this file [No need to compile it first]. Note that both `main-loader.lisp` or `main-compiler.lisp` will load Garnet automatically. The `main-compiler.lisp` file will also transfer the new bin files to the $SURFHOME/bin/ directories. While compiling the supplied source, all warnings may be ignored, as far as I know. Note also that the supplied bin files are for a Sparc. If modified files are compiled individually (without using `main-compiler.lisp`), then you must transfer the .sparcf (or .fasl) binary file to the appropriate bin directory so that `main-loader.lisp` will load in the next session. Note that any modifications to structure definitions or in-line functions require that the entire system be recompiled for safety. Sometimes the compilation bugs out in the middle for no apparent reason; in this case try restarting Lisp and loading `main-compiler.lisp` again. To save a new image, see the more-installation.doc file in the surf-hippo/doc directory. Note that after you have installed your system, using an image thereafter is faster.

Further customization: If there is a `customs.lisp` file in the Surf-Hippo user home directory (defined according to either the SURFUSERHOME environment variable or set to the Surf-Hippo home directory), then this file is loaded on the initial start up. For example, this file may set certain global variables according to a user's preferences, or load other files.

39 More Installation Notes

See also Section 31.

39.1 Installing New Source

For updating the entire source code to a new version, download the compressed tar file that will be named something like:

```
surf-hippo.2.7c.tar.Z
```

Put this file in the surf-hippo home directory, for example:

```
unix-prompt> mv surf-hippo.2.7c.tar.Z /usr/local/surf-hippo/
unix-prompt> cd /usr/local/surf-hippo/
unix-prompt> uncompress surf-hippo.2.7c.tar.Z
unix-prompt> tar -xvf surf-hippo.2.7c.tar
```

These steps will update all subdirectories. You can now delete the original tar file:

```
unix-prompt> rm *.tar
```

Edit the surf-hippo executable file in the surf-hippo directory to point to the proper lisp executable (in the surf-hippo/cmucl directory).

To recompile the system, start up Surf-Hippo, and enter the following into the Lisp:

```
lisp-prompt> (setq user::*force-all-compile* t)
```

Now compile and make an image as described in the following section.

39.2 Image Saving Notes

If you want to save a new image from lisp, use the command

```
lisp-prompt> (opal:make-image "full-image-pathname")
```

If you want to use the surf-hippo/surf-hippo command to run the image (as is usually the case if you have set up things to run from Emacs with M-x surf-hippo), then the image filename should be "/usr/local/surf-hippo/imagename", according to what the Surf-Hippo home directory is, and "imagename" is used in the surf-hippo executable script file, for example:

```
cd $SURFHOME
exec $CMUCLLIB/../bin/lisp "$@" -core $SURFHOME/image
        -eval '(load "messages/initial-message.lisp" :verbose nil)
```

The supplied version of the surf-hippo executable script assumes that the image file is in $SURFHOME/image.

The simplest way to recompile all changed files and make a new image is to simply load:

```
$SURHOME/misc/loaders/image-maker
```

For example, with "C-z l /usr/local/surf-hippo/misc/loaders/image-maker" from the ILISP (*surf-hippo*) buffer. Note that this will store (overwrite) the resulting image as the file "image": make sure that the read permissions are set correctly for the new file.

If an image for a complete new source is compiled and loaded on top of a previous Surf-Hippo image, then the updated image will be about 10MB larger than before. Until I can figure out how to recover space allocated to redifined symbols in the static heap, the only way to avoid this is to make updated source images using a Garnet image to start with. Several versions of these are available at the Surf-Hippo ftp site.

39.3 ILISP Notes

If your emacs automatically include ILISP, then something like the following should be in your emacs setup file (e.g. /.emacs) so that the ILISP files supplied with Surf-Hippo are used:

```
(setq load-path (cons (expand-file-name "/usr/local/surf-hippo/misc/ilisp/") load-path))
(load "/usr/local/surf-hippo/misc/ilisp/ilisp.el")
```

If you want to run CMUCL from ILISP, set up an ILISP command in the emacs setup file to point to wherever the lisp is:

```
(setq cmulisp-program "/home/cbcl/lyle/cmu-17c/bin/lisp")
```

Otherwise, for just running Surf-Hippo, then add the following to the emacs setup file:

```
(defdialect surf-hippo "CMU Common LISP" cmulisp)
(provide 'ilisp-surf-hippo)
(autoload 'surf-hippo  "ilisp" "Inferior CMU Common LISP with Garnet." t)
(setq surf-hippo-program "/usr/local/surf-hippo/surf-hippo")
```

39.4 GARNET Notes

If you are going to install Garnet(not necessary for using Surf-Hippo), use the following settings for the load variables (edit the garnet-loader.lisp file as explained in garnetx.x_README file):

```
(unless (boundp '*Garnet-Going-To-Compile*)
  (defvar load-utils-p T)
  (defvar load-kr-p T)
  (defvar load-opal-p T)
  (defvar load-inter-p T)
  (defvar load-multifont-p NIL)
  (defvar load-gesture-p NIL)
  (defvar load-ps-p T)
  (defvar load-aggregadgets-p T)
  (defvar load-aggregraphs-p NIL)
  (defvar load-debug-p #+garnet-debug T #-garnet-debug NIL)
  (defvar load-gadgets-p t)
  (defvar load-demos-p NIL)
  (defvar load-lapidary-p NIL)
  (defvar load-gilt-p NIL)
  (defvar load-c32-p NIL))
```

39.5 UNIX Environment Variables Notes

For running LISP and GARNET explicitely (not necessary for using Surf-Hippo), set the CMUCLLIB and GARNETHOME environment variables in your startup file (e.g. .login or .cshrc), for example:

```
setenv CMUCLLIB /home/lisp/cmu-18a/lib/
setenv GARNETHOME ~lyle/systems/garnet/
```

39.6 Other Customizations

In the file $SURFHOME/misc/loaders/surf-hippo-loader.lisp the startup functions START (which leaves you talking to the Lisp interpreter) and SURF (which starts the menus) call INIT-SURF-DIRECTORIES which sets the program directories according to the environment variables SURFHOME, possibly SURFUSER-HOME, and as a last resort, by HOME. If you want the Surf-Hippo code and your data (input and output) files to reside on different directories, edit the following lines (marked with a "->") in surf-hippo-loader.lisp:

```
      (defun init-surf-directories ()
        (in-package "SURF")
        (setq *Surfdir*
              (if (assoc :SURFHOME lisp::*environment-list*)
                  (cdr (assoc :SURFHOME lisp::*environment-list*))
                  (concatenate 'string (cdr (assoc :HOME lisp::*environment-list*)) "/surf-hippo")))
        (surf::create-path *surfdir*)
        (setq *Surf-user-dir*
              (if (assoc :SURFUSERHOME lisp::*environment-list*)
                  (concatenate 'string (cdr (assoc :SURFUSERHOME lisp::*environment-list*)) "/")
                  *Surfdir*))
        (surf::create-path *surf-user-dir*)
->      (setq surf::*circuit-directory* (concatenate 'string *Surf-User-Dir* "circuits/"))
        (wh::create-path surf::*circuit-directory*)
->      (setq surf::*data-directory* (concatenate 'string *Surf-User-Dir* "data/"))
        (wh::create-path surf::*data-directory*)
->      (setq wh::*plot-directory* (concatenate 'string *Surf-User-Dir* "plot/"))
        (wh::create-path wh::*plot-directory*)
        (setq surf::*use-gc-announce-window* t) ; problems with save-image version.
        )
```

For example, you could change:

```
(setq surf::*circuit-directory* (concatenate 'string *Surfdir* "circuits/"))
```

to

```
(setq surf::*circuit-directory* "/home/jobobo/surf-hippo-input/my-circuits")
```

39.7 Source Tar File Procedure

Full source file updates are made with the following procedure, assuming SURFHOME is set correctly, where "x.x" refers to the version number.

```
unix-prompt> cd somewhere
unix-prompt> source $SURFHOME/misc/shell-scripts/cp-source
unix-prompt> tar -cvf ../surf-hippo.x.x.tar .
unix-prompt> compress ../surf-hippo.x.x.tar
```

Note that $SURFHOME/misc/shell-scripts/cp-source will remove all files recursively from the pwd (here the directory "somewhere"). In this example, the compressed tar file will be written on the parent directory of "somewhere".

40 Problems and Miscellaneous Simulation Issues

This file describes some of more common problems you can encounter with the simulator, including the following topics:

- Stuck Windows
- Getting Stuck During the Integration
- Some Bugs
- X Display Notes

Additional information may be found in Section 41 and Appendix B.

40.1 Stuck Windows

Normally you should avoid destroying a Surf-Hippo window with the X window manager (but see below). If you need to destroy a Surf-Hippo window, and Control "d" doesn't work (see below), then run the function:

```
* (clean-windows)
```

which will give a menu for destroying the current windows. This function will also free up "stuck" windows if there was a bug during the execution of some window menu operations. If this doesn't work, try:

```
* (mdw)
```

which stands for Mouse Destroy Window. A more drastic approach is:

```
* (caows)
```

which stands for Clear All Output Windows (but this will not get rid of menus - for this use (MDW)).

Manually clearing menus or other windows may be necessary, for example, when there are active menus visible, but some error has caused an exit from the local menu interaction loop. The symptom for this is a non-responsive "OK" button: clicking here does not remove the menu, and the following message appears in the Lisp window:

```
WARNING: Interaction-Complete called but not inside Wait-Interaction-Complete
```

It is possible that neither of these techniques will work, and you just get more Debugger messages, e.g.:

```
    Click or Type on any object or window...
    in Window #k<KR-DEBUG:*MENU-WIN*-45463>, :LEFTDOWN at x=306, y=84:

    Warning in Destroy: aggregate '#k<KR-DEBUG:AGGREGATE-45464>' has no parent,
        is in window '#k<KR-DEBUG:*MENU-WIN*-45463>', but is not that window's:aggregate.

    Error in function CHECK-SLOT-TYPE:
bad KR type: value *DESTROYED*(was #k<KR-DEBUG:AGGREGATE-45464>),
```

etc. If worse comes to worse, go to the Lisp top level, iconify (and ignore) the bad windows with the X window manager, and keep going. You will have to restart Lisp to get rid of the bad windows.

It may also be possible to run into a "destroyed" window in the wrong place, giving a Debugger message something like:

```
Error in function KR::VALUE-FN:
   Non-object *DESTROYED*(was #k<KR-DEBUG:PLOT-WINDOW-13654>) in g-value or get-value (slot is :DRAWABLE)
```

In this case, it may be sufficient to restart from the top-level. At worst, you may need to clear all the current windows with a combination of `(CAOWS)`, `(MDW)` and destroying with the window manager (if all else fails).

40.2 Getting Stuck During the Integration

It is possible that the integration will either be very slow, or actually get stuck, depending on circuit parameters. This should only happen for "unrealistic" values. As explained in numerical.doc, normally a message will appear if the integration get really stuck, e.g.:

```
Integration stuck at time 4.353ms [internal integer time step = 2].
Try either increasing the error criteria *ABSOLUTE-VOLTAGE-ERROR*,
which is now 0.001, or reducing the simulation duration.
```

Otherwise, if you suspect that Surf-Hippo has gotten a little choked, try breaking into the Debugger with C-c C-c, and then seeing what the value of the global variable `*REAL-TIME*` is. If `*REAL-TIME*` indicates something is wrong, then restart the Debugger with an ABORT signal (enter 0 or q). You can then restart with `(surf)`. Otherwise, if you want to continue the simulation, just enter the CONTINUE command to the Debugger (1):

```
            .
            .
Starting transient solution
```

Enter C-c C-c ⇒

```
Interrupted at #x2016B4.

Restarts:
   0: [CONTINUE] Return from BREAK.
   1: [ABORT   ] Return to Top-Level.

 Debug  (type H for help)

 (UNIX::SIGINT-HANDLER #<unavailable-arg>
#<unavailable-arg> #.(SYSTEM:INT-SAP #xEFFFECB0))
 0] *real-time*
 0.10192788   <-- Too small!
 0] q
 * (surf)

  ** Surf-Hippo: The MIT CBIP Neuron Simulator (Version 2.0) **
            .
            .
```

40.3 Some Bugs

REPEAT: CMU Lisp is not bullet-proof. Critical results should be saved at reasonable (roughly hourly?) intervals. Here are some bugs:

- If you get something similar to the following message:

  ```
  segv_handler: No mapping fault: 0x19a40004
  segv_handler: Recursive no mapping fault (stack overflow?)
  ```

  your Lisp session is lost. If this occurs consistently, please send us a note.

- With major Lisp crashes (e.g. segmentation violations), killing the Lisp job and restarting may not make things hunky-dory: you may need to boot the machine. It appears that these crashes somehow corrupt the file system. Probably there is a less drastic technique for recovery.

  Sometimes you will get a segmentation violation that will nonetheless put you back into Lisp (via quitting the Debugger). In this case, it is best to save everything and reload the system.

- With large simulations, or after long sessions, CMUCL may break into the monitor (typically during GC), with a message like:

```
GC lossage.  No transport function for object 0x191d36bf
LDB monitor
ldb>
```

  You may be able to keep going by typing

```
ldb> exit
```

  Otherwise, you have to:

```
ldb> quit
```

  and restart Lisp.

- Sometimes the system bugs during the initial loading, with cryptic error messages such as:

```
            .
            .
Loading Surf-Hippo Roylance-Clmath...

Argument X is not a REAL: NIL.

Restarts:
  0: [CONTINUE] Return NIL from load of #p"roylance-clmath-src:roylance-clmath-loader".
  1:            Return NIL from load of "/usr/local/surf-hippo/misc/loaders/surf-hippo-loader".
  2:            Return NIL from load of "/usr/local/surf-hippo/misc/loaders/main-loader".
  3:            Return NIL from load of "/usr/local/surf-hippo/misc/loaders/main-compiler".
  4: [ABORT   ] Return to Top-Level.

Debug  (type H for help)

(KERNEL:TWO-ARG-> NIL 3045846332)
0]
```

  In this case, all that you should do is quit the Debugger and Lisp, and start again.

40.4 X Display Notes

If menus or other graphics do not display on your screen, and there are no error messages printed in the Lisp window, then a probable cause is that the X display environment variable is pointing somewhere else. Use the `printenv` Unix command to check it:

```
unix-prompt> printenv DISPLAY
cogni.iaf.cnrs-gif.fr:0.0
unix-prompt>
```

Contrary to the Garnet README file, I have had problems (at least under OpenWindows) if the DISPLAY environment variable is set to ":0.0" or "unix:0.0". Rather, it appears that the DISPLAY variable must include the machine name, i.e.:

```
DISPLAY=cogni.iaf.cnrs-gif.fr:0.0
```

as if the machine running Garnet and the display machine are different. In my .openwin-init file I have the following lines:

```
# Start clients on screen 0
#
#SETDISPLAYSCREEN 0
xhost ‘hostname‘
DISPLAY=‘hostname‘:0.0
```

41 Features, Not Bugs

This section includes various situations which can leave you in the Debugger. Some cases may indicate code bugs, while others occur by "unreasonable" values in the parameters specific to a given simulation. Other bugs are noted in various sections. The topics covered here include:

- Bug fix files.
- Profiling Overflow
- Can't `QUIT-SH`
- `FLOATING-POINT-OVERFLOW`
- No response from menus or windows
- XLIB Related Errors
  - `Event code 0 not implemented...`
  - `DRAWABLE-ERROR`
  - `Type-error`
  - Window Color Allocation (ALLOC-ERROR)
  - Window Update
- Fast Mouse Click
- GC Announce Window
- Not Enough Core
- Printing Postscript$_{TM}$ Files
- Zooming Too Far
- Bad Plot Parameters
- Parameter Error Checking
- System Compiling
- Drawing Errors
- `COMMON-LISP::SUB-SERVE-EVENT` Hang

41.1 Bug Fix Files

If there is a bug-fix.lisp file in the Surf-Hippo user home directory (defined according to either the SURFUSERHOME environment variable or set to the Surf-Hippo home directory), then this file is loaded on the initial start up. Bug fix files are posted as necessary on the Surf-Hippo web site, or sent out to the user mailing list.

41.1.1 Profiling Overflow

Trying to profile code after a long session can sometimes give the following error:

```
(profile-all)
     .
     .
     .
Type-error in KERNEL::OBJECT-NOT-TYPE-ERROR-HANDLER:
671398848 is not of type (UNSIGNED-BYTE 29)

Restarts:

0: [ABORT] Return to Top-Level.

Debug  (type H for help)

("LAMBDA (PROFILE::NAME PROFILE::CALLERS-P)"
(-70.0 -70.0 -69.94982 -69.91509 -69.871124 ...)
(0.0 1.4930964e-5 5.033907 10.000012 10.0021105 ...) 0.1 2359472 ...)
Source: (- (PROFILE::TOTAL-CONSING) PROFILE::START-CONSED)
0]
```

The solution is to reset the gc (garbage collector) count:

```
(setq lisp::*total-bytes-consed* 0)
```

41.2 Hanging on QUIT-SH

If QUIT-SH puts you into the Debugger, try the more robust function SYSTEM::QUIT.

41.3 Arithmetic Error During a Simulation FLOATING-POINT-OVERFLOW

The simulator cannot handle arbitrary circuit dyamics - for example, unbounded node voltages. If, because of a mistake (or poor choice) for a circuit parameter or parameters, or because of the circuit architecture, the simulation may halt prematurely with an arithmetic error, for example:

```
Starting transient solution
         .
         .
         .
Arithmetic error FLOATING-POINT-OVERFLOW signalled.

Restarts:
  0: [ABORT] Return to Top-Level.

Debug  (type H for help)

(X86:SIGFPE-HANDLER #<unused-arg> #<unused-arg> #.(SYSTEM:INT-SAP #x3FFFEC68))
Source:
; File: target:code/float-trap.lisp

; File has been modified since compilation:
;   target:code/float-trap.lisp
```

```
; Using form offset instead of character position.
(ERROR (QUOTE FLOATING-POINT-OVERFLOW))
0]
```

One way to verify this is to try to see the simulation results so far. Thus, from within the Debugger (to keep useful context information current), try:

```
0] (sim-output)
```

A typical cause of a state variable (such as voltage) going wild is a time step that is too large.

41.4 Menus and Other Windows Do Not Respond to Mouse, or Do Not Refresh

If a window doesn't seem to respond to the mouse/keyboard commands, consider two possibilities. First, make sure that the caps lock key is not ON. Second, in plot and histology windows, various text fields (such as graph annotations and comments) may be edited when selected by the (left) mouse. In this case, further keyboard actions will be directed at editing the text field, until the (left) mouse is clicked again over the field.

Menu interactions and window refreshing are sometimes cranky. If the interpreter or menus do not respond, Lisp is probably garbage collecting ("GC'ing" - a fact of life with Lisp). If a new menu or other window is blank, you can try to refresh the window by:

- Hitting "r" or "R" with the mouse over the window.
- Resizing the window with the window manager (e.g. clicking on a corner and moving it with the mouse).
- Using the refresh option of your window manager.

41.5 XLIB Related Errors

41.5.1 `Event code 0 not implemented...`

Apparently if certain types of bad drawing commands are issued, a recursive error condition is set up,indicated by

```
Event code 0 not implemented for display #<XLIB:DISPLAY :0 (The XFree86 Project, Inc R40200000)>

Restarts:
  0: [CONTINUE] Ignore the event
  1: [ABORT   ] Return to Top-Level.

Debug  (type H for help)

(XLIB::GET-INTERNAL-EVENT-CODE
 #<XLIB:DISPLAY :0 (The XFree86 Project, Inc R40200000)>
 111)
Source: Error finding source:
Error in function DEBUG::GET-FILE-TOP-LEVEL-FORM:  Source file no longer exists:
  target:clx/input.lisp.
     .
     .
     .
```

which finally ends up with the debugger throwing up its hands:

```
Help! 12 nested errors.  KERNEL:*MAXIMUM-ERROR-DEPTH* exceeded.
```

As of this writing, unfortunately the only thing to do is quit Lisp (by destroying the Emacs buffer, for example, or by issuing the appropriate `kill` command from the shell) and restart.

41.5.2 DRAWABLE-ERROR

XLIB DRAWABLE-ERRORs arise once in a while, for example when Surf-Hippo trys to draw too far beyond the (virtual) window borders. For example:

```
DRAWABLE-ERROR in current request  Code 14.0 [GetGeometry] ID #xA0008D

Restarts:
  0: [ABORT] Return to Top-Level.

Debug  (type H for help)

(XLIB::X-ERROR XLIB:DRAWABLE-ERROR :DISPLAY
       #<XLIB:DISPLAY cogni.iaf.cnrs-gif.fr:0
              (X11/NeWS - Sun Microsystems Inc. R3000)> :ERROR-KEY ...)
   0]
```

For now, the only response is to punt from the Debugger to top level. In some cases, it may even be necessary to restart the Lisp if you want to access any graphics windows.

41.5.3 XLIB Related Errors: Type-error

An error may occur if mis-sized graphics are drawn on plot or histology windows. The error may show up as something like:

```
Type-error in KERNEL::OBJECT-NOT-TYPE-ERROR-HANDLER:
   43722 is not of type (SIGNED-BYTE 16)

Restarts:
  0: [ABORT] Return to Top-Level.

Debug  (type H for help)

(XLIB:DRAW-LINE 6
                #<XLIB:PIXMAP :0 33554579>
                #<XLIB:GCONTEXT :0 33554576>
                118
                ...)[:EXTERNAL]
0]
```

The key symptom is a reference to a basic XLIB drawing function, such as XLIB:DRAW-LINE or XLIB:DRAW-CIRCLE. Usually, it may be sufficient to destroy the culprit window (if Control-D doesn't work, try (MDW), or at worst, (CAOWS), which will destroy all output windows), and then repeat the operation.

41.5.4 Window Color Allocation Errors (ALLOC-ERROR)

Sometimes, you may get the following error in the Lisp window:

```
ALLOC-ERROR in current request  Code 84.0 [AllocColor]

Restarts:
  0: [ABORT] Return to Top-Level.

Debug  (type H for help)

(XLIB::X-ERROR XLIB:ALLOC-ERROR
                :DISPLAY
```

```
                #<XLIB:DISPLAY cogni.iaf.cnrs-gif.fr:0 (X11/NeWS - Sun Microsystems Inc. R3000)>
                :ERROR-KEY
                ...)
    0] q
```

You may be able to recover by typing "q" to the Debugger (as in above example) - otherwise you have to quit Lisp, quit the offending application, and restart Lisp. With regard to problems with colors, this error seems to be related to having too many colors allocated at once. This could happen, for example, when concurrently viewing a very multi-colored image with an application like "xv" or the like. In our experience, Netscape is a particularly bad "color hog" as far as the Lisp is concerned.

In the code that keeps track of colors (`src/gui/colors.lisp`) the idea of the `*COLOR-LIBRARY*` and `*COLOR-FILL-LIBRARY*` libraries is twofold: one to save memory and time by having new colors or filling styles made only when necessary, and two to avoid the color allocation error when there are too many colors in the X display. The maximum number of colors that can eventually be allocated is determined by the variable `*COLOR-LIBRARY-RESOLUTION*`, which sets the resolution for each of the red blue and green components (0 to 1) of the created colors. Thus, if this variable is 0.02 (the default), there can be a total of $50 \times 50 \times 50 (= 125000)$ colors in the library. It remains to be seen what is the maximum.

Another idea is the use of `*COLOR-LIBRARY-MAXIMUM*`, which punts the creation of new color if the number of entries in the color library reaches this level.

41.5.5 Window Update Errors

Another occasional error may look something like this (typically when you manipulate windows too quickly):

```
Asynchronous ACCESS-ERROR in request 21132 (last request was 21509)
                                          Code 30.0 [ChangeActivePointerGrab]

Restarts:
  0: [CONTINUE] Ignore
  1: [ABORT   ] Return to Top-Level.

Debug  (type H for help)

(XLIB::READ-ERROR-INPUT
 #<XLIB:DISPLAY cogni.iaf.cnrs-gif.fr:0 (X11/NeWS - Sun Microsystems Inc. R3000)>
 21132
 #S(XLIB::REPLY-BUFFER
      :SIZE 32
      :IBUF8 #(0 10 82 140 0...)
      :NEXT #S(XLIB::REPLY-BUFFER :SIZE 32 :IBUF8 # :NEXT # :DATA-SIZE 32)
      :DATA-SIZE 32)
 (NIL))

0]
```

If you have been running from the interpreter (entering commands directly into the Lisp window), then just enter 0 (for Ignore). With luck, you can recover from this error gracefully even while running Surf-Hippo - try to get back to the Main Menu, and hit Quit. You will be in good shape if the quit message appears in the Lisp window:

```
Do you want to quit LISP? (RETURN for NO, yes/YES for YES):

   T
   *
```

Now, you can continue Surf-Hippo from where you left off simply by entering:

```
* (surf)

  ** Surf-Hippo: The MIT CBIP Neuron Simulator (Version x.x) **
```

41.5.6 Fast Mouse Click Errors

Sometimes clicking too fast in a window gets an XLIB error, for example:

```
Asynchronous ACCESS-ERROR in request 2697 (last request was 2705)
                        Code 30.0 [ChangeActivePointerGrab]

Restarts:
0: [CONTINUE] Ignore
1: [ABORT   ] Return to Top-Level.

Debug  (type H for help)

(XLIB::READ-ERROR-INPUT
#<XLIB:DISPLAY substantia-nigra:0 (X11/NeWS - Sun Microsystems Inc. R3000)>
2697
#S(XLIB::REPLY-BUFFER
:SIZE 32
:IBUF8 #(0 10 10 137 0...)
:NEXT #S(XLIB::REPLY-BUFFER :SIZE 32 :IBUF8 # :NEXT # :DATA-SIZE 32)
:DATA-SIZE 32)
(NIL))
0] q
*
```

You can generally recover by just quitting from the Debugger, as shown above. However, if Surf-Hippo is running when this happens, get back to the main menu and quit (temporarily) before quitting from the Debugger. Then, restart Surf-Hippo with `(surf)`.

41.6 Not Enough Core Error

If you try to print or save too many windows at once, you may get the following error in the Lisp window:

```
Error in function #:G27:  Could not fork child process: Not enough core

Restarts:
0: [ABORT] Return to Top-Level.

Debug  (type H for help)

(#:G27)
0]
```

If this happens, just type "q" to get out of the Debugger. One solution is to write out the .ps file using the menu (i.e. do not select the "Print selected windows now?" option), and then print out the .ps file from the UNIX shell. In fact, if you get this error, the .ps file is probably residing in the specified plot directory, so that you can just reference it from the UNIX shell.

If this error occurs after selecting some action with the mouse, the windows may be left in a locked state - to unlock them:

```
* (unlock-windows)
NIL
*
```

Some notes on this via email:

```
Date: 27 Oct 94 10:07:33-0600
From: Ted Dunning <ted@crl.nmsu.edu>
To: lyle@cogni.iaf.cnrs-gif.fr
Subject: sparc 10, 4.1.3 patch recommedations

forking lisp could easily require enough space to completely clone
lisp (if they use fork instead of vfork).  this would make shell-exec
very different from running something from the shell.

the other piece of information that might help you is the output of
/etc/pstat -s when things are not working well.

18788k allocated + 5572k reserved = 24360k used, 177236k available

good luck!

Date: 27 Oct 94 11:57:34-0600
From: Ted Dunning <ted@crl.nmsu.edu>
To: lyle@cogni.iaf.cnrs-gif.fr
Subject: sparc 10, 4.1.3 patch recommedations

   Date: 27 Oct 94 19:32:07-0400
   From: Lyle Borg-Graham <lyle@cogni.iaf.cnrs-gif.fr>

   lisp hasn't crashed at the moment, but:

        lyle@cogni>/etc/pstat -s
        63676k allocated + 7680k reserved = 71356k used, 32240k available
                                                         ^^^^^
this means that forking lisp will probably be impossible (you had 18M
retained, x 2 for new and old help = 36 MB).

if you add 100MB more swap you are likely to be ok.

   wrt the shell-exec, is there a way to avoid the clone with fork and
   use vfork (I am no Unix hacker!).

no idea.  you might ask cmucl-bugs
```

41.7 Printing Postscript$_{TM}$ Files Crash

There is a problem with the Unix interface that can cause Lisp to crash when printing out more than one .ps file (using the print windows menu) - the message:

```
Oh no, got a PendingInterrupt while foreign function call was
active.
```

will appear in the Lisp window, and Lisp crashes. If this happens, you will have to load everything from scratch. To avoid this problem for now, the "Put all selected windows together?" option in the "Window Printing Options" menu is always set when you request the "Print selected windows now?" option, that is multiple selected windows will be put into a single .ps file. If you do need to print out several windows, either create the .ps files via a single call to the print windows menu, and then print out these files from the unix shell (independent of CMU Lisp), or print out each window with succesive calls to the print windows menu.

The global variable `*KILL-MULTIPLE-WINDOW-PRINTING*` will prevent Surf-Hippo from printing multiple ps files. The default for this variable is NIL.

41.8 Zooming Too Far

If you try to zoom in on too small an area, you may get the following error:

```
Error in function C::DO-CALL:
  Condition slot is not bound: CONDITIONS::OPERATION

Restarts:
  0: [ABORT] Return to Top-Level.

Debug  (type H for help)

(SPARC:SIGFPE-HANDLER #<unavailable-arg>
                      #<unavailable-arg>
                      #.(SYSTEM:INT-SAP #xEFFFEC40))
0]
```

Hit q to return to the top-level lisp, and left-mouse over the plot to restore the last zoom view. Try the zoom with a larger area. This bug should be fixed. Also, making the plot window larger (e.g. with the window-manager) and replotting (right-mouse) can allow a higher zoom.

41.9 Bad Plot Parameters

If a plot window is specified with bad parameters (e.g. via a menu or by an explicit function call), then the plot will punt, e.g.:

```
* Plot window FOO-BIPOLAR-23012195: Voltages was specified with bogus Y max (-21.64) min (-21.639078) valu


 Fatal Error in Function SETUP-PLOT:

  NIL

 ** Restarting from Top Level **
*
```

and you will be returned to the top level prompt. The offending window will be normally by stuck - it is necessary to call `(unstick-windows)`.

41.10 Parameter Error Checking

There is a *minimum* of error checking of entered parameters, particularly in the menus. It is up to the user to infer what are reasonable constraints on specific parameters (e.g. non-negative, non-zero, negative, etc.).

41.11 System Compiling

The system compile and image file saving script, `surf-hippo/misc/loaders/image-maker.lisp`, checks the write dates of all the source and corresponding binary files. If Emacs is used to edit a source file, and the temporary edit file is present, then something like the following bug will occur:

```
;;;; Loading #p"/usr/local/surf-hippo/lib/surf-hippo-version.lisp".

Argument X is not a REAL: NIL.
```

```
  Restarts:
    0: [CONTINUE] Return NIL from load of "/usr/local/surf-hippo/misc/loaders/main-loader".
    1:            Return NIL from load of "/usr/local/surf-hippo/misc/loaders/main-compiler".
    2:            Return NIL from load of "/usr/local/surf-hippo/misc/loaders/image-maker".
    3: [ABORT   ] Return to Top-Level.

  Debug  (type H for help)

  (KERNEL:TWO-ARG-> NIL 3170185043)
  Source:
  ; File: target:code/numbers.lisp

  ; File has been modified since compilation:
  ;   target:code/numbers.lisp
  ; Using form offset instead of character position.
  (TWO-ARG-</> TWO-ARG-> > CEILING FLOOR ...)
  0]
```

The solution is to remove the offending "#...#" file from one of the `surf-hippo/src/*` directories.

41.12 Drawing Errors

Plotting will choke if there is an attempt to draw something that is far from the designated plotting area of a given window. This will give an error something like:

```
Type-error in KERNEL::OBJECT-NOT-TYPE-ERROR-HANDLER:
   -77063 is not of type (SIGNED-BYTE 16)

Restarts:
  0: [ABORT] Return to Top-Level.

Debug  (type H for help)

(XLIB:DRAW-RECTANGLE 7 #<XLIB:PIXMAP :0 46137485> #<XLIB:GCONTEXT :0 46137486> -131 ...)[:EXTERNAL]
0] q
```

At least one solution is to exit the Debugger (as above), and destroy the window in question, and then try to replot.

41.13 Hanging on `COMMON-LISP::SUB-SERVE-EVENT`

There exists a so far unfixed hang on the Linux CMUCL, sometimes (but not always) provoked by heavy graphics activity. The first symptom is simply no response, but with the ILISP buffer status indicating "run". Breaking into the Debugger (CONTROL-C CONTROL-C), typing "back" and seeing the `SUB-SERVE-EVENT` call confirms the diagnosis:

```
(System running but nothing happening...)

(Type CONTROL-C CONTROL-C)

Interrupted at #x1289E38F.

Restarts:
  0: [CONTINUE] Return from BREAK.
  1: [ABORT   ] Return to debug level 1.
```

```
  2:            Return from BREAK.
  3:            Return to Top-Level.

Debug  (type H for help)

(UNIX::SIGINT-HANDLER #<unused-arg> #<unused-arg> #.(SYSTEM:INT-SAP #x3FFFDD10))
Source:
; File has been modified since compilation:
;   target:code/signal.lisp
; Using form offset instead of character position.
(DEFINE-SIGNAL-HANDLER SIGINT-HANDLER "Interrupted" BREAK)
0]] back
0: (UNIX::SIGINT-HANDLER #<unused-arg>
                         #<unused-arg>
                         #.(SYSTEM:INT-SAP #x3FFFDD10))
1: ("Foreign function call land")
2: ("Foreign function call land")
3: ("Foreign function call land")
4: ("Foreign function call land")
5: (COMMON-LISP::SUB-SERVE-EVENT NIL 0)
6: (SYSTEM:WAIT-UNTIL-FD-USABLE 4 :INPUT NIL)
7: (XLIB::BUFFER-INPUT-WAIT-DEFAULT
    #<XLIB:DISPLAY :0 (The XFree86 Project, Inc R335)>
    NIL)
...
```

Unfortunately, there is no solution as yet but to destroy the ILISP buffer (!) and start a new Lisp. Obviously, make sure to save anything important from the buffer before killing it. Hopefully, this hang will be solved in the not-so-distant future.

.

A A Short Introduction to Lisp

This section discusses the most elementary features of Lisp, in particular those necessary or especially helpful for understanding Surf-Hippo. There are numerous references on Lisp, the (non-official) standard being "Common Lisp: the Language, 2nd Edition" (aka "CLtL2"), by Guy L. Steele, Jr., published by Digital Press. The web is full of Lisp info - a good place to start is the unofficial CMUCL website (http://www.cons.org/cmucl).

In the Surf-Hippo distribution there are also references in the doc/outside_docs directory, including the faq-lisp and Good-Lisp-Style files. The faq-lisp file is especially useful for pointers to various on-line and published resources.

A.1 What is Lisp?

Lisp, is one of the older high-level languages, and was originally designed for artificial intelligence purposes. However, Lisp is also used for practically every kind of programming domain. For example, Surf-Hippo demonstrates that Lisp can be successfully applied to a "hard-core" numerical application, with essentially the same efficiency as C or other languages traditionally used in this area. While the fundamental data structure in Lisp is the list (the name "Lisp" came from "LISt Processing"), a concept which can be exploited quite nicely, it's perfectly possible to work with arrays and other data structures where these are more appropriate. In contrast to C, which has a very limited syntax, Lisp has a rich variety of defined symbols; one result is that you can write code that almost looks like real sentences (see e.g. the `LOOP` construct). Libraries (in the sense used by C) are not required to run a Lisp program and, if you don't need the fastest speed, you can use variables without having to declare their types.

In sum, and in stark contrast to languages like C, it is very easy to write "quick and dirty" Lisp code that may suit an application perfectly well, while it is also possible to write stricter Lisp code that fully exploits the processing power of your particular platform.

There are several varieties of Lisp - Surf-Hippo uses a high-quality, "industrial-strength", public domain Lisp from Carnegie-Mellon University: CMU Common Lisp (or CMUCL). "Common Lisp" means that this Lisp corresponds to the ANSI standard for Common Lisp.

A.2 The Lisp Command Line

The Lisp command line is like the UNIX command line. There's a prompt and you type in Lisp commands that will be processed by the interpreter. The default CMUCL prompt is a star (`*`). This is different from C where you have to compile functions to be able to use them. In Lisp, comment lines start with the semi-colon (`;`).

A.3 Compiling Versus Interpreting

A big advantage of Lisp is that you can evaluate the same code either directly by entering the code into the interpreter or by compiling the code running the resulting binary. Of course, the latter is executed much more efficiently, but for many tasks simply entering non-compiled code works fine. For example, running a batch of simulations in the form of a simple loop will run more or less the same speed whether the loop is interpreted or compiled.

Another advantage of Lisp is that if you do want to compile something, this may be done incrementally - you do not have to recompile the entire system.

A.4 List Processing

Data structures in Lisp were originally based on lists, and all expressions (called "forms)" that Lisp processes or evaluated (e.g. function calls) are in the list format. So, you can have lists of floats, lists of strings, lists of structures, etc. The beginning and the end of a list are defined by parentheses, and the empty list is equivalent to NIL. A synonym for list is "cons" (in fact the basic function for constructing lists is `CONS`). Some examples:

```
'()                        ; The empty list.
'(1 2 3 4)                 ; A list of 4 integers.
'("sodium" "potassium")    ; A list of two strings.
'(1 "aaa" 2 "b")           ; A mixed list.
```

You probably noticed the quote just before the opening parenthesis. It is there to ensure that the parenthesis is considered a list and not a function call (see below). A function call itself is written as a list whose first element is the name of the function. For example, suppose you want to calculate the square root of 2.0. The square root function in Lisp is called `SQRT`. Thus, the explicit command that you would type on the Lisp command line, after the star, to calculate the square root of 2.0 is:

```
* (sqrt 2.0)
1.4142135
```

Suppose you want to calculate the sum of 5.2 and 2.7. The operator for addition is the symbol `+`, so you will type:

```
 * (+ 5.2 2.7)
7.9
```

Now, we can understand why it is an error to write a list without the quote at the front. If, for example, one would write `(1 2 3)`, then the Lisp interpreter would take that as a function call for a function called 1, which would produce an error:

```
* (1 2 3)

In: 1 2
  (1 2 3)
Error: Illegal function call.
```

So, the correct way to handle the list is to put a quote:

```
* '(1 2 3)
(1 2 3)
```

Here, we typed in a list and the Lisp interpreter simply returned the value of the list, because no operation is required in this case.

The backquote character is almost the same, except that any element inside the backquoted list that is preceded by a comma will be evaluated. Confusing at first, but quite useful at times.

```
* (setq a 101)     ; Set a variable called A to 101.
101
* '(1 a b)         ; This quoted list is not evaluated.
(1 A B)
* `(1 a b)         ; The backquote also suppresses evaluation.
(1 A B)
* `(1 ,a b)        ; But a comma after the backquote causes evaluation of the symbol A as
(1 101 B)          ; the previously set variable.
```

Note that `SETQ` sets (binds) a variable (symbol) to a value.

There are a lot of commands to operate on lists: we can merge them, take parts of them, copy them, delete parts of them, make lists of lists, apply an operator to all the elements, etc. Here are a few examples:

```
* (append '(1 2) '(3 4))   ; Merge lists
(1 2 3 4)
* (car '(1 2 3 4))         ; Take the first element
1
* (cdr '(1 2 3 4))         ; Take all elements but the first
(2 3 4)
* (map 'list 'sqrt '(1.0 2.0 3.0 4.0))      ; Apply the SQRT operator
(1.0 1.4142135 1.7320508 2.0)
```

A.5 Nesting

Lisp allows the nesting of lists. For example:

```
* (+ (sqrt 3.0) (+ 2.0 -3.0) (sin (/ pi 4.0)) (log 5.0))
3.0485955001730587d0
* (append (cdr '(1 2 3)) (cdr '("a" "b" "c")))
(2 3 "b" "c")
```

A.6 All Those Parentheses!

If you examine the source code of Surf-Hippo, non-Lispers usually just see a jungle of parentheses. The secret through this mess is two-fold: first, with (a little) experience, the parentheses tend to fade into the background. Second, it is much easier to write and evaluate Lisp code in an environment that does most of the parentheses work (e.g. nice indenting, matching open and closed parentheses) for you. All of our work is done under Emacs (itself written in Lisp) which provides both a Lisp-mode for editing, and the ILISP environment for running programs. These sorts of tools are really essential for efficient travelling in the Lisp world.

A.7 Getting Help Directly from Lisp: the `DESCRIBE` and `APROPOS` Functions

For all objects that you can manipulate in Lisp, whether variables or functions, you can get helpful information using the `DESCRIBE` function. A few examples:

```
* (describe 1)
It is a composite number.

* (describe '(1 2 3))
(1 2 3) is a CONS.

* (describe my-vector)
#(0.0d0 0.0d0 0.0d0 0.0d0 0.0d0...) is a vector of length 10.
It has no fill pointer.
Its element type is DOUBLE-FLOAT.
```

This function is very useful for functions that you have written in the past and you don't really remember how you call it. Assuming you had earlier defined a function called `MY-FUNC`:

```
* (describe 'MY-FUNC)
MY-FUNC is an internal symbol in the SURF-HIPPO package.
Function: #<Function MY-FUNC {4840BED1}>
Function arguments:
  (x y)
Its defined argument types are:
  (T T)
Its result type is:
  NUMBER
On Tuesday, 7/2/00 08:48:51 am [-1] it was compiled from:
/usr/local/surf-hippo/circuits/foo.lisp
  Created: Tuesday, 7/2/00 08:48:49 am [-1]
```

This example may look a bit complex, but it gets better after some practice. Note that when `DESCRIBE` is called on a symbol that corresponded to a compiled function, as here, the information includes the source file for that function. The real usefulness of `DESCRIBE` appears in neural simulations, for example, when you suddenly want to know a very particular parameter of a very particular synapse somewhere on your cell, and the documentation or editing functions in Surf-Hippo do not readily give you the information you want.

Another useful function is `APROPOS`, which can help you find references to a given symbol:

```
* (apropos 'markov)
MARKOV
SURF-HIPPO::MARKOV-FLOW-ARRAYS
SURF-HIPPO::BOGUS-INITIALIZE-MARKOV-PARTICLE (defined)
SURF-HIPPO::MARKOV-STATE-VOLTAGE-TRANSITION-FUNCTIONS (defined)
SURF-HIPPO::GET-MARKOV-PARTICLE-P-ARRAY (defined)
SURF-HIPPO::PLOT-MARKOV-STATES
SURF-HIPPO::*MARKOV-TIME-STEP* (defined)
   .
   .
   .
SURF-HIPPO::EVAL-ALL-MARKOV-PARTICLES (defined)
:MARKOV, value: :MARKOV
```

In this example, `APROPOS` lists all the symbols that include `MARKOV`, in a format that includes (separated by `::`) the package under which each symbol was defined (that is `SURF-HIPPO`) (see Section 3.1) .

Of course, `DESCRIBE` knows about `APROPOS`, and vica-versa!

```
* (describe 'apropos)
APROPOS is an external symbol in the COMMON-LISP package.
Function: #<Function APROPOS {1212E99}>
Function arguments:
  (string &optional package external-only)
Function documentation:
  Briefly describe all symbols which contain the specified String.
  If Package is supplied then only describe symbols present in
  that package.  If External-Only is true then only describe
  external symbols in the specified package.
Its declared argument types are:
  ((OR BASE-STRING SYMBOL) &OPTIONAL (OR PACKAGE BASE-STRING SYMBOL) T)
Its result type is:
  (VALUES)
On Wednesday, 11/2/94 02:57:43 am EST it was compiled from:
target:code/package.lisp
  Created: Tuesday, 11/1/94 01:34:59 pm EST
  Comment: $Header: package.lisp,v 1.37 94/10/31 04:11:27 ram Exp $
*
```

Likwise,

```
* (apropos 'describe)
ILISP:ILISP-DESCRIBE (defined)
DEBUG::DESCRIBE-DEBUG-COMMAND (defined)
C::DESCRIBE-BYTE-COMPONENT (defined)
C::DESCRIBE-TN-USE (defined)
   .
   .
   .
COMMON-LISP::DEFAULT-DESCRIBE (defined)
COMMON-LISP::DESCRIBE-FUNCTION-NAME (defined)
DESCRIBE (defined)  <--- This is the function we are talking about.
```

A.8 Variables - Specials with Global Values

You can define variables in Lisp, exactly as in other languages. Variables that will be referred to at any time while running Lisp (that is, the associated symbol has dynamic scope) are defined with the `DEFVAR` form,

and called special variables. The assignment (or binding) of the symbol for a special variable to some value sets the global value of that variable. Despite this distinction between the terms special and global, we will refer to special variables at various times as either special or global. Thus a global variable is a symbol that can be referenced by any procedure later on after its definition. Variables that are defined and used locally within a form or program construct are described below. According to good Lisp style, almost all the global variables in Surf-Hippo begin, and usually end with, an asterisk.

You don't have to declare variable types at the time of definition; the Lisp interpreter will automatically handle it when you set a specific value. For example:

```
* (defvar a)
A
*
```

Here, we defined a variable called `A`, and the Lisp interpreter lets you know it has been created by echoing the name of the variable. You can now set the value of the variable to anything you want, single-float, string, list, array:

```
* (setq a 1)
1

* (setq a "This is a sentence in a list.")
"This is a sentence in a list."

* (setq a '(1 "bb" (3 4 5) "c"))
(1 "bb" (3 4 5) "c")
```

Here we bound the same variable symbol to a number, a string, and a mixed list, without caring to specify the type. This illustrates the flexibility of the language. It makes Lisp a little bit slower in this case, but Lisp has other features that allow you, once the code is working, to add optimization commands (where of course you specify the types of objects) and to recover the efficiency of languages such as C.

Note also in the third binding of `A` the initial quote for the list causes all elements of the list to be interpreted as is, in other words the inner list `(3 4 5)` is not taken as a function call to the function "3".

Note also that if a `SETQ` form is applied at top-level to a symbol which has not been defined with a `DEFVAR`, Lisp automatically declares that symbol to be a special (global) variable.

A.9 Loops

Loops are extremely important in any kind of programming. Lisp provides the powerful `LOOP` facility, which is equivalent to the `for( ..; ..; ..)` construct of C, but much easier to use. In Lisp, you can control loops by indices, you can loop over the elements of a list, or loop either as long as or until a condition is satisfied. You can also use the loop to do specific actions and/or to incrementally build a result list. Here are only the most frequently used variations - the first example does a particular action at each iteration, the second one builds a list.

```
* (loop for i from 1 to 5   ; Loop controlled by index i.
        do (print i))
1            ; Prints index varying between 1 and 5.
2            ; These are the side effects.
3
4
5
NIL            ; Form finally returns NIL because loop defines no return value.

* (loop for val in '(1.0 2.0 3.0)   ; Loop over a list.
        collect (sqrt val))
(1.0 1.4142135 1.7320508)      ; Return value is the collected list.
```

The MAPCAR function applies some function to successive elements of a list, and then returns a list of results. Thus, MAPCAR encapsulates the basic functionality of LOOP, often (but not always) with somewhat more "elegant" syntax. For example, recall the loop shown earlier in Section 10 for adding and setting up current sources to all the somas in a circuit: The

```
(loop for soma in (somas) do (pulse-list (add-isource soma) '(0.0 0.5 1.0)))
```

An equivalent version using MAPCAR is:

```
(mapcar #'(lambda (soma) (pulse-list (add-isource soma) '(0.0 0.5 1.0))) (somas))
```

The LAMBDA macro is used to define unamed functions for local use (see Section A.11 below).

A.10 Defining Named Functions

The construct to define new procedures or functions is the DEFUN macro. The simplest layout is as follows:

```
(defun function-name (arg1 arg2 ...)
   (something-here ...)
   (something-there ...)
   ...
)
```

Lisp functions can do two things: perform specific actions, and return a result. The first action is called a side effect; this could be printing characters, changing the value of a global variable, defining a new array, etc. The return value (or values) of a function can be any Lisp object - a string, a number, a list, even another function. The following example defines a function, called N-RAND, that takes one integer argument, NUM, and returns a list containing exactly NUM random numbers between 0.0 and 1.0. In this example no checking is performed on the arguments. The Lisp interpreter answers by echoing the name of the new function:

```
* (defun n-rand (num)
    (loop for i below num
          collect (random 1.0)))         ; Function definition
N-RAND
* (n-rand 5)                   ; First call
(0.008551478 0.5912522 0.4989798 0.18791544 0.28064716)
* (n-rand 5)                   ; Second call
(0.6307291 0.9514588 0.5839038 0.3242128 0.5458571)
```

In Lisp, it is possible to define functions with optional arguments; that is, the user will be free to provide or not values for some arguments, but the function will know what default values it should take. This is done by adding &OPTIONAL after the obligatory arguments, and before the optional ones. The argument name may be followed (within a list form) by the default value that the function should use. If there is no default value specified, then the default for that argument is NIL:

```
* (defun foo (x &optional (y 10))
    (+ x y))
FOO
* (foo 1)      ; Use default value of 10 for argument y
11
* (foo 1 2)      ; Everything is provided
3
```

With optional arguments, values supplied in the function call are bound to the function arguments in the order in which the optional arguments are listed. Similarly, you can have keyword arguments, denoted by &KEY, in which keywords defined in the DEFUN precede the appropriate values when the function is called:

```
* (defun foo (x &key (y 10) (z 20)
    (+ x y z))
FOO
```

In the following case, the default value of 10 is used for argument `y`. Note that keywords include a colon - here `:z` refers to the function variable `z`:

```
* (foo 1 :z 0)
11
```

In this example everything is provided - note that the keyword args don't have to be in the same order as in the original `DEFUN`:

```
* (foo 1 :z 100 :y 2)
103
```

A.11 Defining Unamed Functions

While the `DEFUN` macro creates a function with a name, in some cases it may be more convenient to create an unamed function. For example, this avoids having to decide on what a "good" name would be, in other words one that isn't or won't be used elsewhere. Unamed functions are created with a *lambda-expression*, whose syntax is similar to the `DEFUN` syntax. The particle type definitions for the canonical Hodgkin-Huxley squid axon kinetics (src/parameters/hodgkin-huxley.lisp), for example, use such expressions to define the forward and backward rate constants as given in the original Hodgkin and Huxley paper:

```
(particle-type-def
 `(m-hh
   (class . :hh)
   (alpha-function . (lambda (voltage)
                       (let ((v-40 (- voltage -40.0)))
                         (/ (* -0.1 v-40)
                            (1- (exp (/ v-40 -10.0)))))))
   (beta-function . (lambda (voltage)
                      (* 4.0 (exp (/ (- voltage -65.0) -18.0)))))))
```

Here both expressions have a single voltage argument - note that the main difference between these expressions and the `DEFUN` syntax is that the symbol `LAMBDA` replaces `DEFUN`, and there is no name that follows `LAMBDA`. Certainly the above definition could have been written by first defining some named functions, and then referring to these in the `PARTICLE-TYPE-DEF` form:

```
(defun m-hh-alpha-function (voltage)
  (let ((v-40 (- voltage -40.0)))
    (/ (* -0.1 v-40)
       (1- (exp (/ v-40 -10.0))))))

(defun m-hh-beta-function (voltage)
  (* 4.0 (exp (/ (- voltage -65.0) -18.0))))

(particle-type-def
 `(m-hh-test
   (class . :hh)
   (alpha-function . m-hh-alpha-function)
   (beta-function . m-hh-beta-function)))
```

You may note that in some cases in the text we have used the `LAMBDA` with a leading "`#'`", but not in the `TYPE-DEF` examples. Normally, a leading "`#'`" is *necessary* for a `LAMBDA` expression so that the Lisp reader knows that the expression should be expanded into a `(FUNCTION ...)` expression. However, for simplicity,

the Surf-Hippo code that parses TYPE-DEF expressions assumes that whenever a parameter entry starts with (LAMBDA ...), then this is taken to be a function definition (although, if you do use #'(LAMBDA ..).instead of (LAMBDA ..), all will be well)..

A.12 Structures

Lisp has built-in structures, as in C. These are defined with the DEFSTRUCT macro, which includes the name of the structure, followed by specifications for the different fields (slots) of the structure. The following example is adapted from Surf-Hippo and shows how the current source structure is defined. The fields of the structure can either be defined with simply the slot name (e.g. :NAME, :NODE-1 and :NODE-2), defined with a default value (e.g. :CELL-ELEMENT, :ENABLED, :CURRENT-DATA and :RESISTANCE), or defined with both a default value and a type specification (e.g. :CURRENT-DATA and :RESISTANCE):

```
(defstruct isource
   name
   (cell-element nil)      ; What the source is a part of (soma or segment)
   (current-data '() :type list)   ; used to store simulation data
   node-1
   node-2
   (enabled t)         ; Source is only considered when this is T
   (resistance 0.0 :type single-float)   ; Series resistance, in Mohms.
   ....
)
```

Here the DEFSTRUCT form generates a function for creating the structure (MAKE-ISOURCE) as well as accessor functions to retrieve and change values in the structure (e.g. ISOURCE-NAME, ISOURCE-NODE-1). The various structures in Surf-Hippo are defined in the sys/structures.lisp file. Typically, you will never need to define new structures in order to use Surf-Hippo, and in fact Surf-Hippo is written so that you never have to deal explicitely with structures or their associated accessor functions. In fact, in general it is not a good idea to modify the slots of specific structures directly (that is using SETF and the appropriate structure accessor form) since the simulator assumes that access to many structure slots are via specific functions which in turn take care of necessary side effects. However, since structures are the fundamental data structure of all the things that defines a circuit, including the specific elements and the element types, it is useful to be aware of the structure concept.

A.13 Variables - Local Bindings

It is frequently useful to define temporary, or local, variables within a form (that is, the associated symbol has lexical scope). For example, these can be variables local to a function, or may contain a complicated mathematical expressions calculated once but used many times within a form. The LET form is then used:

```
(let ((var-1 value-1)
      (var-2 value-2)         ; local definition of the variables
   ...
   )
   (command-1 var-1 var-2 ...)      ; commands in the LET construct
   (command-2 var-1 var-2 ...)      ; that use the local variables.
   ...
)                  ; end of LET construct
```

A local variable exists only within the LET form within which it is defined - as soon as the LET is complete, the variable disappears. Note that the symbols which represent arguments in function definitions are local variables, exactly as if they were defined in a LET.

A.14 Binding Special Variables Locally

As we have seen, there are a variety of special (global) variables that define aspects of the simulation environment. Sometimes it is useful to temporally assign values to these variables while executing some code, and then restore the original values of the variables once the code is finished. This could be done by expliciting saving the original values somewhere, and referencing them afterwards, but a cleaner method is to make a local binding of the global variables using `LET`.

For example, imagine that we want to run a series of simulations, charting the response to different levels of current inputs, all plotted together in the same window (or windows). Thus, the basic `LOOP` could be something like this:

```
(loop for current from 0.0 to 0.5 by 0.1
      do (pulse-list *isource* (list 10.0 200.0 current))
      (goferit))
```

Here, the `LOOP` iterates over values of a local variable `CURRENT` (from 0 to 0.5, with steps of 0.1), and this variable is used to form the argument to `PULSE-LIST` (see Section 10). Note that the function `LIST` returns a list of all of its arguments. Once the current source stimulation is set, `GOFERIT` is called (see Sections 4 or 19) which runs a simulation.

The global variable `*OVERLAY-ALL-PLOTS*` (see Section 21) determines whether or not old data is kept in plot windows when new data is plotted. In order to do what we want, the `LOOP` form could be augmented as follows:

```
(setq *OVERLAY-ALL-PLOTS* nil)  ;; Initially, clear out any previously plotted data.
(loop for current from 0.0 to 0.5 by 0.1
      do (pulse-list *isource* (list 10.0 200.0 current))
      (goferit)
      (setq *OVERLAY-ALL-PLOTS* t)) ;; For subsequent iterations, overlay the plots.
```

This works fine, but the manipulation of `*OVERLAY-ALL-PLOTS*` will survive after we are done. We could add another

```
(setq *OVERLAY-ALL-PLOTS* nil)
```

afterwards, but this would not necessarily restore the original value of `*OVERLAY-ALL-PLOTS*`. Rather, a more "hygenic" manipulation of `*OVERLAY-ALL-PLOTS*` is to locally bind it with a `LET` assignment:

```
(let ((*OVERLAY-ALL-PLOTS* nil))
  (loop for current from 0.0 to 0.5 by 0.1
        do (pulse-list *isource* (list 10.0 200.0 current))
        (goferit)
        (setq *OVERLAY-ALL-PLOTS* t)))
```

As before, for the first iteration of the `LOOP` the value of `*OVERLAY-ALL-PLOTS*` is NIL, and for the subsequent iterations it is T. As a side note, when setting a local variable to NIL, you can use the simpler syntax:

```
(let (*OVERLAY-ALL-PLOTS*)
     ...)
```

The important point is that once the `LET` form is finished, the original value of `*OVERLAY-ALL-PLOTS*` is restored, independent of the local assignment.

B On-line Debugger

CMUCL includes an on-line Debugger which is invoked either automatically in case of any error during execution, or explicitly by typing the sequence C-c C-c (entering "Control c" twice). The Debugger includes many commands (summarized by typing "h"), and it will usually reference the source file for the function where the error occured. Note that if the function was defined in the original CMUCL or Garnet source code, the file location will correspond to the system where these systems were compiled.

Execution errors can occur for a variety of reasons, ranging from parameter type errors, to syntax errors in your code, to arithmetic bugs in Surf-Hippo (never! `;-}`). In general, typing "q", "Q" or "0" (the number zero) to the Debugger will return you to the top-level of the Lisp interpreter. Depending on the error, the Debugger may immediately tell you what the problem is, for example:

```
* (/ 1 0)
Arithmetic error DIVISION-BY-ZERO signalled.
Operation was KERNEL::DIVISION, operands (1 0).

Restarts:
  0: [ABORT] Return to Top-Level.

Debug  (type H for help)

(KERNEL::INTEGER-/-INTEGER 1 0)
0] 0
*
```

In other cases it is useful (or necessary) to back up the Debugger execution stack to find legible information indicating where the problem lies, by displaying the stack with the Debugger's `BACKTRACE` command (or "back" for short).

If you landed in the Debugger for some unacceptable or unknown reason, please send a bug report.

B.1 Debugger Thrashing

In the current version of CMUCL for Linux, invoking the backtrace command takes a while, during which time there is a lot of hard drive thrashing. Hopefully this will be addressed in future releases.

B.2 Type Errors and Function Optimization

Consider a function in which the argument is defined to be a certain type:

```
(defun foo (x)
  (declare (single-float x))
  (+ x x))
```

If this `FOO` is then called with an argument that is not a single-float, then the Debugger will give copius information right off the bat:

```
* (foo 1)
Type-error in KERNEL::OBJECT-NOT-SINGLE-FLOAT-ERROR-HANDLER:
   1 is not of type SINGLE-FLOAT

Restarts:
  0: [ABORT] Return to Top-Level.

Debug  (type H for help)

(FOO 1 1)[:EXTERNAL]
```

```
Source:
; File: /usr/local/surf-hippo/src/sys/foo.lisp
(DEFUN FOO (X) (DECLARE (SINGLE-FLOAT X)) (+ X X))
0]
```

On the other hand, if FOO was compiled with various optimization options, for example:

```
(defun foo (x)
  (declare (single-float x)
   (optimize (safety 0) (speed 3) (space 0)))
  (+ x x))
```

Then the same bad call will give a more opaque Debugger message:

```
* (foo 1)

Error in function UNIX::SIGSEGV-HANDLER:  Segmentation Violation at #x483B2170.

Restarts:
  0: [ABORT] Return to Top-Level.

Debug  (type H for help)

(UNIX::SIGSEGV-HANDLER #<unused-arg>
#<unused-arg>
#.(SYSTEM:INT-SAP #x3FFFED08))
Source:
; File: target:code/signal.lisp

; File has been modified since compilation:
;   target:code/signal.lisp
; Using form offset instead of character position.
(DEFINE-SIGNAL-HANDLER SIGSEGV-HANDLER "Segmentation Violation")
0]
```

Backtracing up the debugger stack is the answer:

```
0] back
0: (UNIX::SIGSEGV-HANDLER #<unused-arg>
  #<unused-arg>
  #.(SYSTEM:INT-SAP #x3FFFED08))
1: ("Foreign function call land")
2: ("Foreign function call land")
3: ("Foreign function call land")
4: ("Foreign function call land")
5: (FOO #<unused-arg> #<unavailable-arg>)[:EXTERNAL]
6: (EXTENSIONS:INTERACTIVE-EVAL (FOO 1))
7: (COMMON-LISP::%TOP-LEVEL)
8: ("DEFUN MAKE-IMAGE")
9: (COMMON-LISP::RESTART-LISP)

0]
```

Note that the first line with intelligible information

```
5: (FOO #<unused-arg> #<unavailable-arg>)[:EXTERNAL]
```

tells you at least the offending function. Running `DESCRIBE` gives the information we saw before:

```
0] (describe 'foo)
FOO is an internal symbol in the SURF-HIPPO package.
Function: #<Function FOO {48761929}>
Function arguments:
  (x)
Its defined argument types are:
  (SINGLE-FLOAT)
Its result type is:
  SINGLE-FLOAT
On Thursday, 9/21/00 03:49:04 pm [-1] it was compiled from:
/usr/local/surf-hippo/src/sys/foo.lisp
  Created: Thursday, 9/21/00 03:49:02 pm [-1]

0]
```

If a global variable whose type is declared (not always the case) is assigned a value of the wrong type (for example in a user-defined circuit or script file), then you will get an error that looks something like this:

```
Error in function UNIX::SIGBUS-HANDLER:  Bus Error at #x71DEEC0.

Restarts:
  0: [ABORT] Return to Top-Level.

Debug  (type H for help)

(UNIX::SIGBUS-HANDLER #<unavailable-arg> #<unavailable-arg> #.(SYSTEM:INT-SAP #xEFFFE998))
0]
```

The error may also be flagged as:

```
Error in function UNIX::SIGSEGV-HANDLER:  Segmentation Violation at #x482C6518.
```

Sometimes it can be difficult to track down the offensive value of a global variable with the Debugger, since the error may be quite buried. The basic strategy is to check all variable values that you have explicitly changed or set in your code against their expected (proclaimed) types (if any) by using the `DESCRIBE` function.

For example, the above error would be generated if the Surf-Hippo variable `*TEMP-CELCIUS*` was set to an integer:

```
* (setq *temp-celcius* 100)
* 100
```

and then the simulation was run (note that an error is *not* flagged by the erroneous `SETQ`). If we examine this variable with `DESCRIBE`:

```
* (describe '*temp-celcius*)
*TEMP-CELCIUS* is an internal symbol in the SURF-HIPPO package.
It is a special variable; its value is 100.
Its declared type is SINGLE-FLOAT.
Special documentation:
Temperature of the simulation in degrees Celcius.
*
```

we see that the code *expects* this to be a single-float, and thus will choke if the type of `*TEMP-CELCIUS*` is otherwise. Note the quote in front of `*TEMP-CELCIUS*` when it is the argument to the `DESCRIBE` function. Otherwise `DESCRIBE` will evaluate the argument, and then describe the result of the evaluation:

```
* *TEMP-CELCIUS*
100
* (describe *TEMP-CELCIUS*)
It is a composite number.
```

C Mouse and Keyboard Actions for All Output Windows

The following are mouseable and keyable actions pertinent to all output windows (plots, histology, and info windows).

- If menus are buried under other windows, hitting CONTROL-q over any info, plot or histology window will deiconify and raise any active menus.
- CONTROL-m will prompt for a menu to change parmeters of the selected window
- CONTROL-p will prompt for printing the selected window (and the others).
- CONTROL-d will destroy the selected window (after prompting for verification). Avoid destroying any Surf-Hippo window with the X window manager.
- CONTROL-D will destroy the selected window immediately.
- CONTROL-a will allow alignment of other windows to the selected window.
- r will resurrect the window.

D Mouse and Keyboard Actions for Plot Windows

The following are mouseable and keyable actions pertinent to data plotting windows.

- R will replot the window.
- h, H, CONTROL-h or CONTROL-H will create an Information window with this section in it.
- CONTROL-LEFT will give a resizable rectangle for zooming.
- CONTROL-m will create a main plot menu for revising various parameters of the selected plot, including the font used by the labels. Various menus may be called from this executive menu.
- CONTROL-t gives a menu for adding/removing comments and titles on windows. Window text can be edited directly on the window by clicking the the left mouse on the text to start editing (typical emacs commands) and clicking with the mouse again to exit.
- RIGHT (unzoom) will replot at the last zoom view.
- CONTROL-RIGHT (restore) will replot the original view.
- CONTROL-s will prompt for saving the selected window to a Lisp file, allowing easy editing and re-creation of plots.
- CONTROL-SHIFT-LEFT will give a resizable rectangle for zooming to a new window.
- CONTROL-g, when invoked by itself, will toggle a grid on the plot. This may also be specified in the main plot menu below. Otherwise, CONTROL-g is the generic abort command. For example, when zooming or locating a cross hair, keyboard keys that invoked the command may be released while moving the mouse as long as the mouse button is pressed. Hitting CONTROL-g *before* releasing the mouse button will abort the operation.
- MIDDLE will put a cross hair on the window which can then be moved around by the mouse, with the current coordinates updated in the upper right corner, in the units of the data in the window. If there was a previous point marked on the window, then the slope between that point and the current one is also displayed. Normally, when the middle mouse button is released the xy coordinates of the last location of the cross hair is displayed in the units of the window data and the cross hair will remain. If CONTROL-g is typed while the middle mouse button is held down, then the cross hair will be removed.

  NOTE If the window has been resized, the cross hair result will not be correct until the window has been replotted/redrawn (e.g. for a data plot, doing a Restore (CONTROL-RIGHT)). If the window is a HISTOLOGY or BUILDER window, then the length between the current and last points is shown.
- CONTROL-MIDDLE will prompt for placing and editing a small cross marker, with optional label, at the any points that were selected with the cross hairs. This menu also allows for other editing options, such as removing the cross if you just want the text. The labels may also be edited directly by clicking the left mouse button on the text.
- CONTROL-f will remove the oldest zoom-rectangle or cross-hair that is visible.
- CONTROL-l will remove the newest zoom-rectangle or cross-hair that is visible.
- CONTROL-A will remove all the zoom-rectangles and markers.
- CONTROL-b will prompt for drawing a straight line on the plot.
- CONTROL-B will prompt for destroying or editing any added straight lines on the plot.
- CONTROL-G, will create a menu for a grid on the plot. This menu may also be accessed via the main plot menu below.

- CONTROL-e will erase any pointer comment (e.g. cross-hair coordinates).
- CONTROL-E will erase the data stored in a (standard) plot window, but not the displayed image. This may be useful if there are a lot of plots and memory is getting tight. While a window whose data has been erased may be printed and examined with the cross hairs, zooming, unzooming, restoring, or other rescaling will not be possible.
- CONTROL-L will toggle a lock on a plotting window which when set prevents that window from being used for subsequent plots. If CONTROL-L is activated on a non-locked window, then the lock is set. If the lock is already set, a menu will verify that you want to unlock the window.

E Mouse and Keyboard Actions for Histology Windows

The following are mouseable and keyable actions pertinent to histology windows.

- h, H, CONTROL-h or CONTROL-H will create an Information window with this section in it.
- LEFT will pick the closest soma or segment and print the name and xy coordinates in the upper left hand corner. For segments, this will identify the segment whose distal node is closest to the mouse.
- RIGHT will pick the closest soma and print its name and xy coordinates in the upper left hand corner. This may be convenient when a soma is surrounded by a lot of segments.
- CONTROL-SHIFT-LEFT will give a menu for various manipulations and examinations of the last soma or segment chosen by the LEFT mouse action, if there was one. This menu allows enabling/disabling plotting for all the plottable variables associated with the chosen circuit node, adding/removing various components, such as channels, sources, and synapses, and output of element and node parameters. Depending on the simulation, other options may also be available.
- CONTROL-g is the generic abort command. For example, when zooming or locating a cross hair, keyboard keys that invoked the command may be released while moving the mouse as long as the mouse button is pressed. Hitting CONTROL-g *before* releasing the mouse button will abort the operation.
- MIDDLE will put a cross hair on the window which can then be moved around - When the middle mouse button is released the xy coordinates of the last location of the cross hair is displayed in microns. If there was a previous point so delineated, or if a cell node was pointed out (mouse LEFT), then the length in microns between the current and last points is shown. If the window has been resized, the cross hair result will not be correct until the window has been redrawn.
- CONTROL-MIDDLE will prompt for placing a small cross marker, with optional label, at the last point that was selected with the cross hairs. This menu also allows for other editing options, such as removing the cross if you just want the text. These markers may be removed, or their fonts changed, via the main drawing menu (CONTROL-m), "Edit element graphics" option. The labels may also be edited directly by clicking the left mouse button on the text.
- CONTROL-m will create a menu for revising various parameters of the window, as described above.
- CONTROL-t will prompt for a single line comment in that is printed in the lower right corner of the window. This can also be used to erase any comment that is in either the lower or upper right corner of the selected window.
- CONTROL-e will erase any comment in the upper right hand corner (e.g. cross-hair coordinates).
- CONTROL-LEFT will give a resizable rectangle for zooming to a new window.
- CONTROL-f will remove the oldest marker (zoom-rectangle or cross-hair) that is visible.
- CONTROL-l will remove the latest marker (zoom-rectangle or cross-hair) that is visible.
- CONTROL-A will remove all the markers (zoom-rectangle or cross-hair).
- CONTROL-L will toggle a lock on a histology window which when set prevents that window from being used for subsequent output. If CONTROL-L is activated on a non-locked window, then the lock is set. If the lock is already set, a menu will verify that you want to unlock the window.

F Text Edit Commands

Text entry windows in the menus work similarily to emacs:

- C-a moves the cursor to the beginning of the window
- C-e moves the cursor to the end of the window
- C-f moves the cursor one step forward
- C-b moves the cursor one step backward
- C-d deletes the character in front of the cursor
- The "Delete" key deletes the character in back of the cursor
- "Return" or click another button (including the "OK") to enter the text

G Channel and Synapse Parameters

The Surf-Hippo distribution includes a variety of element models taken from the literature. Channel, gating particle and synapse models are listed in Tables 8- **??**.

Reference	File (*.lisp)	Channel
Clay, 1998	clay-98-na	NA-CLAY98
Deisz, 1996	deisz96	NA-DEISZ96
Destexhe et al., 1994	destexhe-3state-na	NA-3STATE-GEN
Destexhe et al., 1994	destexhe-vandenberg-bezanilla	NA-DVB94
Borg-Graham, 1987	hippo-TR1161	NA1-TR1161 NA2-TR1161 NA3-TR1161
Hodgkin and Huxley, 1952	hodgkin-huxley	NA-HH, NA-HH-FIT
Hoffman et al., 1997	hoffman-etal-97	NA-HOFFMAN-ETAL-97
Huguenard et al., 1988	huguenard-etal-chs	NA-Y-HUG NA-M-HUG
Jaffe et al., 1994	jaffe94-chs	NA-JAFFE94
Kuo and Bean, 1994	kuo-bean-na	NA-KB94 NA-KB94-V2
Lytton and Sejnowski, 1991	lytton-chs	NA-LYT
Huguenard and McCormick, 1992 McCormick and Huguenard, 1992	mccormick-huguenard-92	NA-MCC-HUG
Migliore et al., 1995	migliore95-chs	NA-MIG95
Patlak, 1991	patlak-na	NA-PATLAK91
Sah et al., 1988 French and Gage, 1982 French et al., 1990	sah-french-etal-na	NA-SGG NA-SGG-FIG10 NA-SGG-POWER-FIT NA-SGG-SHIFTED-POWER-FIT NA-SGG-MPOWER-FIT NA-2-IN-COMP-SGG NA-FSBG-SLICE NA-FSBG-DISS NA-FSBG-MINUS-WINDOW
Rhodes and Gray, 1994		NA-RHO
Traub et al., 1991	traub91-chs	NA-TRB91, NA-TRB91-FIT
Traub et al., 1994	traub94-chs	NA-TRB94 NA-AX-TRB94
Vandenberg and Bezanilla, 1991a/b	vandenberg-bezanilla	NA-VB91
Warman et al., 1994	warman94-chs	NA-WDY, NA-WDY-FIT
	working-fs	NA-FS
Borg-Graham, 1999	working-hpc	NA-HPC
Zador, 1993	zador-chs	NA-TZ-CA1-HH

Table 8: Parameters of Na^+ channel models supplied in the Surf-Hippo distribution. For this table as well as Tables 9 - 12, all files are found in the surf-hippo/src/parameters directory, with the extension ".lisp". Channel names that end in "-FIT" use :HH-EXT class particles whose parameters were chosen to fit the original channel model, as given in the first part of the name.

Reference	File (*.lisp)	Channel
Clay, 1998	clay-98-k	K-CLAY98
Borg-Graham, 1987	hippo-TR1161	DR-TR1161 M-TR1161 A-TR1161
Hodgkin and Huxley, 1952	hodgkin-huxley	DR-HH, DR-HH-FIT
Jaffe et al., 1994	jaffe94-chs	KDR-JAFFE94
Sah et al.,1988 Storm, 1990 Storm, 1988 Storm, 1988 Ficker and Heinemann, 1992	k-chs	KDR-SAH KDR-STO KA-STO88 KD-STO88 KTSLOW-FICKER-92 KTSLOW-FICKER-92-ACT-POWER2 KTSLOW-FICKER-92-ACT-POWER3
Lytton and Sejnowski, 1991	lytton-chs	KD-LYT M-LYT A-LYT
Huguenard and McCormick, 1992 McCormick and Huguenard, 1992	mccormick-huguenard-92	KA1-MCC-HUG KA2-MCC-HUG
Migliore et al., 1995	migliore95-chs	KDR-MIG95 KA-MIG95 KM-MIG95
Traub et al., 1991	traub91-chs	KDR-TRB91, KDR-TRB91-FIT KA-TRB91, KA-TRB91-FIT
Traub et al., 1994	traub94-chs	KDR-TRB94 KDR-AX-TRB94 KA-TRB94
Warman et al., 1994	warman94-chs	KM-WDY, KM-WDY-FIT KA-WDY, KA-WDY-FIT KDR-WDY, KDR-WDY-FIT
	working-fs	KDR-FS
Borg-Graham, 1999	working-hpc	KM-HPC KA-HPC KDR-HPC KD-HPC
Zador, 1993	zador-chs	DR-TZ-CA1-HH

Table 9: Parameters of K^+ channel models supplied in the Surf-Hippo distribution.

Reference	File (*.lisp)	Channel
Hines, 1992	NEURON-k-chs	KC-ML83
Borg-Graham, 1987	hippo-TR1161	C-TR1161 AHP-TR1161
Jaffe et al., 1994	jaffe94-chs	KC-JAFFE94
Köhler et al., 1996		KAHP-KOHLER-96
Huguenard and McCormick, 1992 McCormick and Huguenard, 1992	mccormick-huguenard-92	KC-MCC-HUG
Migliore et al., 1995	migliore95-chs	KAHP-MIG95 KC-MIG95
Traub et al., 1991	traub91-chs	KC-TRB91, KC-TRB91-FIT AHP-TRB91
Traub et al., 1994	traub94-chs	AHP-SOMA-TRB94 AHP-DENDRITE-TRB94 KC-SOMA-TRB94 KC-DENDRITE-TRB94
Warman et al., 1994	warman94-chs	KCT-WDY, KCT-WDY-FIT KAHP-WDY
Borg-Graham, 1999	working-hpc	KAHP-HPC KCT-HPC
Yamada et al., 1989	yamada-koch-adams-chs	KC-YKA89

Table 10: Parameters of Ca^{2+}-dep K^+ channel models supplied in the Surf-Hippo distribution.

Reference	File (*.lisp)	Channel
Borg-Graham, 1987	hippo-TR1161	CA-TR1161
Jaffe et al., 1994	jaffe94-chs	CA-T-JAFFE, CA-T-JAFFE-FIT CA-N-JAFFE, CA-N-JAFFE-FIT CA-L-JAFFE, CA-L-JAFFE-FIT
Lytton and Sejnowski, 1991	lytton-chs	CAT-LYT CAN-LYT
Huguenard and McCormick, 1992 McCormick and Huguenard, 1992	mccormick-huguenard-92	CAT-MCC-HUG
Migliore et al., 1995	migliore95-chs	CA-L-MIG95 CA-N-MIG95 CA-T-MIG95
Traub et al., 1991	traub91-chs	CA-TRB91, CA-TRB91-FIT
Traub et al., 1994	traub94-chs	CA-SOMA-TRB94 CA-DENDRITE-TRB94
Warman et al., 1994	warman94-chs	CA-WDY, CA-WDY-FIT
Borg-Graham, 1999	working-hpc	CA-L-HPC CA-T-HPC CA-N-HPC

Table 11: Parameters of Ca^{2+} channel models supplied in the Surf-Hippo distribution.

Reference	File (*.lisp)	Channel
Barnes and Hille, 1989	barnes-hille-cone-chs	H-BARNES-HILLE-89
Borg-Graham, 1987	hippo-TR1161	Q-TR1161
Huguenard and McCormick, 1992 McCormick and Huguenard, 1992	mccormick-huguenard-92	H-MCC-HUG, H-MCC-HUG-FIT
Madison et al., 1986	madison-etal-86	KCLV-MAD86
Borg-Graham, 1999	working-hpc	H-HPC

Table 12: Parameters of other channel models supplied in the Surf-Hippo distribution.

H Example: Rallpack Source Code

In this section we shall illustrate the Surf-Hippo source code used in some of the presented simulations in Section 33. This code a) defines the Hodgkin-Huxley I_{Na} and I_{DR} channel types, b) defines the soma/short-cable Rallpack 3 structure, and c) sets up and runs the current clamp simulation illustrated on the left in Figure 7.

The parameters for the various element types - cell types, channel types, synapse types, particle types, concentration dependent particle types, concentration integrator types, axon types, pump types, and buffer types - are referenced from parameter libraries specific to each type. Adding (and updating) a new entry of a given element type to the appropriate library is done with the Type-def macros, e.g. `CHANNEL-TYPE-DEF`, `CONC-INT-TYPE-DEF`, and so on. The body of each Type-def macro is a quoted list whose first element is the name (typically a symbol) of an element type, followed by an association list of parameters specific to that sort of element type. The names of the element types given by the Type-defs may then be referenced in the source code.

We start with the definitions for the Hodgkin-Huxley (Hodgkin and Huxley, 1952) I_{Na} and I_{DR} channel types (respectively, `NA-HH` and `DR-HH`). These definitions are made with the `CHANNEL-TYPE-DEF` macro:

```
(channel-type-def
 '(NA-HH
   (gbar-density . 1200)                  ; pS/um2
   (e-rev . 50)                           ; mV
   (v-particles . ((M-HH 3) (H-HH 1))))) ; There are 3 M-HH particles and 1 H-HH particle.

(channel-type-def
 '(DR-HH
   (gbar-density . 360)                ; pS/um2
   (e-rev . -77)                       ; mV
   (v-particles . ((N-HH 4)))))        ; There are 4 N-HH particles.
```

These channel type definitions reference various particle types, including `M-HH`, `H-HH` and `N-HH`, whose definitions are made using the `PARTICLE-TYPE-DEF` macro:

```
(particle-type-def
 `(M-HH
   (class . :HH)                        ; This particle is of the canonical HH form.
   (alpha-function . M-HH-ALPHA)        ; The forward rate constant is given by this function.
   (beta-function .  M-HH-BETA)))       ; The backward rate constant is given by this function.

(particle-type-def
 `(H-HH
   (class . :HH)
   (alpha-function . H-HH-ALPHA)
   (beta-function . H-HH-BETA)))

(particle-type-def
 `(N-HH
   (class . :HH)
   (alpha-function . N-HH-ALPHA)
   (beta-function . N-HH-BETA)))
```

The particle type definitions, in turn, reference various forward and backward rate functions (e.g. `M-HH-ALPHA` and `M-HH-BETA`, respectively, for the `M-HH` particle type). In the definitions for these functions, the voltage

argument is assumed to be in mV, and the functions return rates in 1/ms:

```
(defun m-hh-alpha (voltage)
  (/ (* -0.1 (- voltage -40))
     (1- (exp (/ (- voltage -40) -10)))))
```

Given the prefix notation of Lisp, this expression is equivalent to:

$$\alpha_m(V) = \frac{-0.1(V - -40)}{1 - \exp(\frac{V - -40}{-10})}$$

```
(defun m-hh-beta (voltage)
  (* 4 (exp (/ (- voltage -65) -18))))

(defun h-hh-alpha (voltage)
  (* 0.07 (exp (/ (- voltage -65) -20))))

(defun h-hh-beta (voltage)
  (/ 1.0 (1+ (exp (/ (- voltage -35) -10)))))

(defun n-hh-alpha (voltage)
  (/ (* -0.01 (- voltage -55))
     (1- (exp (/ (- voltage -55) -10)))))

(defun n-hh-beta (voltage)
  (* 0.125 (exp (/ (- voltage -65) -80))))
```

The macro `CELL-TYPE-DEF` defines the parameters for the cell type `HH-AXON`:

```
(cell-type-def
 '(HH-AXON
   (rm . 40000)      ; ohms cm2
   (ri . 100)        ; ohms cm
   (cm . 1)          ; uF/cm2
   (v-leak . -65)))  ; mV
```

Next we define a function which creates a soma/short-cable cell with 11 total compartments (1 soma and 10 segments), and adds I_{Na} and I_{DR} to all compartments.

In Surf-Hippo the electrical model for a segment is single-ended approximation to the cable section, as follows:

```
          g-axial
Prox o---/\/\/\-----+---o Distal
                    |
              memb-elements
                    |
                   Gnd
```

In addition, the soma circuit is simply a parallel RC. However, the Rallpack 3 specification assumes center-tapped approximation to cable sections for the entire circuit, which implies that the end nodes are half-versions of the remaining nodes. With a total length of 1000 microns and 9 middle compartments, in the

Surf-Hippo model this is accomplished by defining end compartments (soma and distal segment) with an equivalent cylinderical length of 50 microns. To maintain symmetry, we adjust the axial resistance of the distal segment by a factor of 2:

```
(defun rallpack-3-11 ()
  (let ((distal-segment
         (segment-chain  ; Create a chain of segments originating from the soma.
                         ; SEGMENT-CHAIN returns the last segment, which we assign to
                         ; a local variable DISTAL-SEGMENT for use below.

          (create-soma   ; This makes the soma.
           (create-cell 'AXON :cell-type 'HH-AXON)    ; This makes the cell.
           (sphere-diameter-from-area (* 50 pi 1)))   ; Diameter => area equals 1/2 segment area.
          NIL             ; Optional base name for segments - not used here.
          10              ; Number of segments.
          100 1)))        ; Default length and diameter of each segment in microns.

    (element-length distal-segment 50)                ; Shorten the distal segment.
    (element-parameter distal-segment 'RI-COEFFICIENT 2)    ; Double its effective Ri.

    ;; Add the Hodgkin-Huxley Na and DR channels to the soma and all the segments, as returned
    ;; from the function CELL-ELEMENTS.

    (create-element (cell-elements) 'NA-HH 'DR-HH)))
```

Now load the circuit definition:

```
(circuit-load 'RALLPACK-3-11)
```

We will now set up the details of a current clamp simulation. First, make sure that the variables defining the integration method and data plot resolution are set (these are the default values):

```
(setq *integration-time-reference* :VARIABLE ; Enable adaptive time step.
      *absolute-voltage-error* 0.05 ; mV
      *absolute-particle-error*  0.001 ; dimensionless
      *user-max-step* 5.0 ; ms
      *user-min-step* 0.0 ; ms
      *pick-time-step-fudge* 0.8 ; dimensionless
      *save-data-step* 2) ; Save every other time point.
```

Note that `*ABSOLUTE-VOLTAGE-ERROR*` and `*ABSOLUTE-PARTICLE-ERROR*` correspond to ϵ^V_{max} and ϵ^x_{max}, respectively, and `*PICK-TIME-STEP-FUDGE*` corresponds to $\varepsilon_{\Delta t}$ (Section 34).

Now add a current source to `*SOMA*` (this global variable references the last created soma in the circuit):

```
(add-isource *SOMA*)
```

The object-oriented nature of this code allows circuit elements to be referenced in a variety of ways. For example, given the above example, we could reference the soma by its name:

```
(add-isource "AXON-soma")
```

We now add a 0.1nA pulse from 10 to 50 milliseconds to the current source we just created, referenced by the global variable `*ISOURCE*`:

```
(pulse-list *ISOURCE* '(10 50 0.1))
```

Now enable element plotting:

```
(enable-element-plot *ISOURCE*)     ; Default for current sources is the current.
(enable-element-plot *SOMA*)        ; Default for somas is the voltage.
(enable-element-plot (distal-tips)) ; Default for segments is the voltage.
```

The function `DISTAL-TIPS` returns a list of the distal segments in the circuit - in this case this gives the same segment as referenced by the local variable `DISTAL-SEGMENT` in the `RALLPACK-3-11` function definition.

Set the simulation time (milliseconds):

```
(setq *USER-STOP-TIME* 50.0)
```

Finally, run the simulation:

```
(goferit)
```

Note that other than the setup of the cell anatomy, all of the steps in this code example may be done from the menus and point-&-click histology graphics.

References

[1] S. Barnes and B. Hille. Ionic channels of the inner segment of tiger salamander cone photoreceptors. *Journal of General Physiology*, 94:719–743, October 1989.

[2] U. S. Bhalla, D. H. Bilitch, and J. M. Bower. Rallpacks: a set of benchmarks for neuronal simulators. *Trends In Neurosciences*, 15(11), 1992. Available from ftp://genesis.bbb.caltech.edu/pub/genesis.

[3] L. Borg-Graham. Modelling the somatic electrical behavior of hippocampal pyramidal neurons. Master's thesis, Massachusetts Institute of Technology, 1987. Also appears as MIT AI Laboratory Technical Report 1161 (1989).

[4] L. Borg-Graham. Modelling the non-linear conductances of excitable membranes. In J. Chad and H. Wheal, editors, *Cellular Neurobiology: A Practical Approach*, chapter 13, pages 247–275. IRL/Oxford University Press, 1991.

[5] L. Borg-Graham. Interpretations of data and mechanisms for hippocampal pyramidal cell models. In P. S. Ulinski, E. G. Jones, and A. Peters, editors, *Cerebral Cortex*, volume 13, pages 19–138. Plenum Press, 1999.

[6] L. Borg-Graham. Additional efficient computation of branched nerve equations: Adaptive time step and ideal voltage clamp. *Journal of Computational Neuroscience*, 8(3):209–225, 2000.

[7] L. Borg-Graham. The computation of directional selectivity in the retina occurs prior to the ganglion cell. *Nature Neuroscience*, 4(2):176–183, 2001.

[8] L. Borg-Graham and N.M. Grzywacz. A model of the directional selectivity circuit in retina: Transformations by neurons singly and in concert. In T. McKenna, J. Davis, and S.F. Zornetzer, editors, *Single Neuron Computation*, pages 347–376. Academic Press, 1992.

[9] J. M. Bower and D. Beeman. *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System.* TELOS/Springer-Verlag, 1994.

[10] J. R. Clay. Excitability of the squid giant axon revisited. *Journal of Neurophysiology*, 80(2):903–913, August 1998.

[11] J. Cooley and F. Dodge. Digital computer solutions for excitation and propagation of the nerve impulse. *Biophysics Journal*, 6, 1966.

[12] E. D. De Schutter. A consumer guide to neuronal modeling software. *Trends In Neurosciences*, 15(11), 1992.

[13] R. A. Deisz. A tetrodotoxin-insensitive sodium current initiates burst firing of neocortical neurons. *Neuroscience*, 70(2):341–351, 1996.

[14] C. A. Desoer and E. S. Kuh. *Basic Circuit Theory.* McGraw-Hill, 1969.

[15] A. Destexhe, Z. F. Mainen, and T. Sejnowski. Synthesis of models for excitable membranes, synaptic transmission and neuromodulation using a common kinetic formalism. *Journal of Computational Neuroscience*, 1:195–230, 1994.

[16] E. Ficker and U. Heinemann. Slow and fast transient potassium currents in cultured rat hippocampal cells. *Journal of Physiology*, 445:431–445, 1992.

[17] C. French and P. Gage. A threshold sodium current in pyramidal cells in rat hippocampus. *Neuroscience Letters*, 56:289–293, 1985.

[18] C. French, P. Sah, K. J. Buckett, and P. Gage. A voltage-dependent persistent sodium current in mammalian hippocampal neurons. *Journal of General Physiology*, 95:1139–1157, 1990.

[19] N. Gazères, L. Borg-Graham, and Y. Frégnac. A model of non-lagged x responses to flashed stimuli in the cat lateral geniculate nucleus. *Visual Neuroscience*, 15:1157–1174, 1998.

[20] P. Hess and R. Tsien. Mechanism of ion permeation through calcium channels. *Nature*, 3, May 1984.

[21] B. Hille. *Ionic Channels of Excitable Membranes.* Sinauer Associates, Sunderland, Massachusetts, 1992. Second edition.

[22] M. Hines. Efficient computation of branched nerve equations. *Int. J. Bio-Medical Computing*, 15:69–76, 1984.

[23] M. Hines. NEURON — a program for simulation of nerve equations. In F. Eeckman, editor, *Neural Systems: Analysis and Modeling*, pages 127–136. Kluwer Academic Publishers, 1992. Vol. 2, http://www.neuron.yale.edu/neuron.html.

[24] M. Hines and N. T. Carnevale. Computer simulation methods for neurons. In M. Arbib, editor, *The Handbook of Brain Theory and Neural Networks.* MIT Press, 1995.

[25] A. L. Hodgkin and A. F. Huxley. Currents carried by sodium and potassium ions through the membrane of the giant axon of *Loligo*. *Journal of Physiology*, 116:449–72, 1952a.

[26] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500–44, 1952b.

[27] D. A. Hoffman, J. C. Magee, C. M. Colbert, and D. Johnston. K^+ channel regulation of signal propagation in dendrties of hippocampal pyramidal neurons. *Nature*, 87:869–875, June 1997.

[28] J. R. Huguenard, O. P. Hamill, and D. A. Prince. Developmental changes in Na^+ conductances in rat neocortical neurons: appearance of a slowly inactivating component. *Journal of Neurophysiology*, 59(3), March 1988.

[29] J. R. Huguenard and D. A. McCormick. Simulation of the currents involved rhythmic oscillations in thalamic relay neurons. *Journal of Neurophysiology*, 68(4), 1992. October.

[30] J. J. B. Jack, D. Noble, and R. W. Tsien. *Electric Current Flow In Excitable Cells*. Clarendon Press, Oxford, 1983.

[31] D. B. Jaffe, W. N. Ross, J. E. Lisman, N. Lasser-Ross, H. Miyakawa, and D. Johnston. A model for dendritic Ca^{2+} accumulation in hippocampal pyramidal neurons based on fluorescence imaging measurements. *Journal of Neurophysiology*, 71(3), March 1994.

[32] R. Joyner, M. Westerfield, J. Moore, and N. Stockbridge. A numerical method to model excitable cells. *Biophysics Journal*, 22:155–170, 1978.

[33] M. Köhler, B. Hirschberg, C. T. Bond, J. M. Kinzie, N. V. Marrion, J. Maylie, and J. P. Adelman. Small-conductance, calcium-activated potassium channels from mammalian brain. *Science*, 273:1709–1714, 1996.

[34] C. C. Kuo and P. B. Bean. Na^+ channels must deactivate to recover from inactivation. *Neuron*, 12:819–829, 1994.

[35] W. W. Lytton and T. J. Sejnowski. Simulations of cortical pyramidal neurons synchronized by inhibitory interneurons. *Journal of Neuroscience*, 66(3), September 1991.

[36] D. Madison, R. Malenka, and R. Nicoll. A voltage dependent chloride current in hippocampal pyramidal cells is blocked by phorbol esters. *Biophysical Journal*, 49, 1986.

[37] H. Markram and M. Tsodyks. Redistribution of synaptic efficacy between neocortical pyramidal neurons. *Nature*, 382:807–810, August 1996.

[38] M. Mascagni and A. S. Sherman. Numerical methods for neuronal modeling. In C. Koch and I. Segev, editors, *Methods in Neuronal Modeling*, chapter 13. MIT Press/Bradford Books, 1998. 2nd Edition.

[39] D. A. McCormick and J. R. Huguenard. A model of the electrophysiological properties of thalamic relay neurons. *Journal of Neurophysiology*, 68(4), 1992. October.

[40] M. Migliore, E. P. Cook, D. B. Jaffe, D. A. Turner, and D. Johnston. Computer simulations of morphologically reconstructed CA3 hippocampal neurons. *Journal of Neurophysiology*, 73(3), March 1995.

[41] J. Patlak. Molecular kinetics of voltage-dependent Na^+ channels. *Physiological Reviews*, 71(4), October 1991.

[42] P. A. Rhodes and C. M. Gray. Simulations of intrinsically bursting neocortical pyramidal neurons. *Neural Computation*, 6:1086–1110, 1994.

[43] P. Sah, A. J. Gibb, and P. W. Gage. The sodium current underlying action potentials in guinea pig hippocampal CA1 neurons. *Journal of General Physiology*, 91, March 1988.

[44] I. Segev, R. E. Burke, and M. Hines. Compartmental models of complex neurons. In C. Koch and I. Segev, editors, *Methods in Neuronal Modeling*, chapter 3, pages 63–96. MIT Press/Bradford Books, 1998. 2nd Edition.

[45] G. L. Steele. *Common Lisp: the Language*. Digital Press, 1990. (aka "CLtL2").

[46] J. F. Storm. Temporal integration by a slowly inactivating K^+ current in hippocampal neurons. *Nature*, 336, November 1988b.

[47] J. F. Storm. Potassium currents in hippocampal pyramidal cells. In J. Storm-Mathisen, J. Zimmer, and O.P. Ottersen, editors, *Progress in Brain Research*, volume 83, pages 161–187. Elsevier Science Publishers B. V. (Biomedical Division), 1990.

[48] M. Tauchi and R.H. Masland. The shape and arrangement of the cholinergic neurons in the rabbit retina. *Proc. R. Soc. London, B*, 223:101–119, 1984.

[49] R. D. Traub, J. G. R. Jefferys, R. Miles, M. A. Whittington, and K. Tóth. A branching dendritic model of a rodent CA3 pyramidal neurone. *Journal of Physiology*, 481(1), 1994.

[50] R. D. Traub, R. K. S. Wong, R. Miles, and H. Michelson. A model of a CA3 hippocampal pyramidal neuron incorporating voltage-clamp data on intrinsic conductances. *Journal of Neurophysiology*, 66(2), August 1991.

[51] C. A. Vandenberg and F. Bezanilla. Single-channel, macroscopic, and gating currents from sodium channels in the squid giant axon. *Biophysical Journal*, 60:1499–1510, 1991a.

[52] C. A. Vandenberg and F. Bezanilla. A sodium channel gating model based on single-channel, macroscopic ionic, and gating currents in the squid giant axon. *Biophysical Journal*, 60:1511–1533, 1991b.

[53] J. Vlach and K. Singhal. *Computer methods for circuit analysis and design*. Van Nostrand Reinhold, 1983.

[54] E. Warman, D. Durand, and G. Yuen. Reconstruction of hippocampal CA1 pyramidal cell electrophysiology by computer simulation. *Journal of Neurophysiology*, 71(6):2033–2045, June 1994.

[55] W. M. Yamada, C. Koch, and P. R. Adams. Multiple channels and calcium dynamics. In C. Koch and I. Segev, editors, *Methods in Neuronal Modeling*, chapter 4, pages 97–134. MIT Press/Bradford Books, 1989.

[56] A. M. Zador. *Biophysics of computation in single hippocampal neurons*. PhD thesis, Yale University, 1993.

Index