A Surf-Hippo Tutorial

Lyle J Graham
Neurophysiology of Visual Computation, CNRS
Université Paris Descartes
45 rue des Saint-Peres
75006 Paris, France

1 Introduction

This is a short tour on using the neuron simulation package Surf-Hippo. We assume that you have a running Surf-Hippo in front of you (and that the system is found at /usr/local/surf-hippo). Otherwise, consult the READ.ME.FIRST file in the Surf-Hippo home directory and go from there.

Here we shall describe some of the basic concepts for defining circuit elements, and how to simulate a neuron model. You don't have to have any experience with Lisp for this tour (there are some brief notes below on Lisp for the tyro), but if you are running Surf-Hippo under Emacs (preferably via SLIME), then it is a good idea to first read the tutorial that is included in Emacs. To ensure consistency, please enter all the example commands as presented in the text below. For convenience, these can be simply cut and pasted from the /usr/local/surf-hippo/doc/SH_tutorial.lisp file.

To run Surf-Hippo from the Linux shell, cd to the Surf-Hippo home directory and enter

```
unix-prompt> surf-hippo
```

You should now see something like this:

```
Starting /usr/local/surf-hippo/surf-hippo ...
CMU Common Lisp release x86-linux 2.4.20 10 March 2000 build 498
Send bug reports and questions to surf-hippo-bugs@ai.mit.edu or cmucl-help@cons.org.
Loaded subsystems:
    Python 1.0, target Intel x86
    CLOS based on PCL version:  September 16 92 PCL (f)
    CLX X Library MIT R5.02
    Defsystem Mar 13 1995
    Garnet Version 3.0
    Surf-Hippo Version 3.0alpha, 4/14/02 02:12 pm [-1]
               http://www.cnrs-gif.fr/iaf/iaf9/surf-hippo.html


To start the Surf-Hippo menus enter (SURF) at the Lisp prompt.
To start the Surf-Hippo demo circuit menu enter (DEMO) at the Lisp prompt.
For help, read the User Manual in the surf-hippo/doc directory.
To quit Lisp, enter (QUIT) at the Lisp prompt.

*
```

where `*` is the Lisp prompt.

If you are running from Emacs, and SLIME has been installed (see surf-hippo/quick-start.txt, or surf-hippo/doc/installation/installation.doc.txt), type "M-x slime" (type the ESC key, and then type `x`, then type `slime`, followed by RETURN or ENTER).

SLIME will start up a Lisp buffer (typically named `*slime-repl surf-hippo*`), where `SH>` is the Lisp prompt.

In either case, you are now talking to the Lisp interpreter (for the rest of this tutorial we will assume this is from the Linux shell). First, to convince you that Lisp is alive and waiting for your commands, type `(+ 1 1)` followed by the RETURN (or ENTER) key:

```
* (+ 1 1 2)
4
*
```

Here, `+` is the add function, followed by (in this case) three numeric arguments.

Now try invoking the main menu by typing `(surf)` at the prompt and hitting RETURN (or ENTER):

```
* (surf)

 ** The Surf-Hippo Neuron Simulation System, Version X.X **

   [Menu appears]
```

Although much of what we will do below can be done by the menus (starting with this one), for now we will mainly enter commands directly into Lisp. So click on "OK" and in response to the message:

```
Do you want to quit LISP? (Hit RETURN/ENTER for NO, type yes/YES for YES):
```

just hit the RETURN (or ENTER) key.

To make sure that everything is initialized for this exercise, run the `CIRCUIT-LOAD` macro (here we don't need to worry about the distinction between Lisp functions and macros):

```
* (circuit-load)
NIL
*
```

The main way to load complete circuit defining files or functions into Surf-Hippo is with `CIRCUIT-LOAD`, but if this form is entered without any arguments, as here, any definitions that were previously loaded are cleared.

2 Lisp

Just a few notes about Lisp for now.

When typing directly to the Lisp prompt (thus, the Lisp interpreter), you can enter either *atoms* or *forms*. Atoms include strings ("hello world"), numbers (3.14156) or symbols ('abc). Forms are lists delineated by parentheses, that may include zero or more atoms and/or forms (thus forms may be embedded in other forms).

Note that when entering symbols they must be indicated by a quote. On the other hand, when Lisp prints out a symbol, there is no quote; furthermore, any characters are capitalized.

```
* 'abc
ABC
*
```

If you type in a form that is preceeded by a quote, then Lisp takes that to be a literal list of things. If there is no preceeding quote, then the first element of the list is taken to be a symbol which is the name of a function (or macro), followed by the function's (or macro's) arguments, if any.

Finally, symbols are case insensitive, whether referring to functions or not. Thus,

```
* (circuit-load)
NIL
*
```

is equivalent to

```
* (CIRCUIT-LOAD)
NIL
*
```

2.1 Errors

Errors generally put you into the debugger, for example something that Lisp does not understand:

```
* grok

Error in KERNEL::UNBOUND-SYMBOL-ERROR-HANDLER:  the variable GROK is unbound.
  [Condition of type UNBOUND-VARIABLE]

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

Debug  (type H for help)

(EVAL GROK)
Source: Error finding source:
Error in function DEBUG::GET-FILE-TOP-LEVEL-FORM:  Source file no longer exists:
 target:code/eval.lisp.
0]
```

Under SLIME you will see something similar. Typically you will then ABORT (in this case, by typing 0), which will bring you back to the Lisp prompt. In other cases, the debugger can be an important tool for finding out problems.

Probably most errors arise from misspelling, e.g. a function name:

```
* (circuit-lade)

; Warning: This function is undefined:
;   CIRCUIT-LADE

Error in KERNEL:%COERCE-TO-FUNCTION:  the function CIRCUIT-LADE is undefined.
   [Condition of type UNDEFINED-FUNCTION]

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

Debug  (type H for help)

(KERNEL:%COERCE-TO-FUNCTION CIRCUIT-LADE)
Source: Error finding source:
Error in function DEBUG::GET-FILE-TOP-LEVEL-FORM:  Source file no longer exists:
  target:code/fdefinition.lisp.
0] 0

* (circuit-load)
;
NIL
*
```

3 Elements

All the parts of a circuit - cells, somas, segments; membrane elements such as sources, channels, or synapses; parts of membrane elements such as gating particles; the element types such as cell types, channel types, particle types - comprise the Surf-Hippo family of circuit elements. All elements and element types have

associated low-level CREATE functions (CREATE-CELL-TYPE, CREATE-CELL, CREATE-SEGMENT, CREATE-CHANNEL, CREATE-PARTICLE, etc.) which you may use to build up a circuit; alternatively a variety of higher-level routines (in particular, CREATE-ELEMENT, described below) may be more appropriate. It is important to understand that there may be several ways to define the same circuit, the idea being that different syntaxes may be more convenient in different circumstances. This may be appreciated by looking over the example files in the /usr/local/surf-hippo/circuits/demos/ directory.

All elements and element types have names, which can be a string, integer or symbol. When an element or element type is created, you can assign a specific name (the typical case for element types), otherwise a default name is automatically generated (the usual case for individual cell elements).

Elements or element types may be referenced by a pointer to the actual element or element type object, or its name. Furthermore, for each kind of element or element type there is a global variable that points to the most recently created instance (*CELL-TYPE*, *CELL*, *SEGMENT*, *CHANNEL*, *PARTICLE*, etc.).

Many functions operate on all types of elements, in a quasi object-oriented way. The most fundamental operation, obtaining the object pointer from an element's name, is accomplished by the function ELEMENT. As an example, let us create a cell with the CREATE-CELL function:

```
* (create-cell "Simple" :cell-type 'cortical :soma-diameter 30)
<Cell Simple: type CORTICAL>
*
```

This function returns the cell object, displaying the *printed representation* (always delineated as <...>) as <Cell Simple: type CORTICAL>. Conforming to the specification in the CREATE-CELL call, the printed representation confirms that the name of the cell is the string "Simple", and that the name of the associated cell type is the symbol CORTICAL (here the cell type is automatically created by CREATE-CELL, as opposed to an explicit call of CREATE-CELL-TYPE).

Note that is is completely arbitrary that we have named the cell with a string, and the cell type with a symbol. Note also that there is a distinction between an element's name and its printed representation. Printed representations only provide information to the user; if you tried to type one of these into Lisp, Lisp wouldn't understand it.

Given a circuit element's name, the ELEMENT function will get the (pointer to that) object:

```
* (element "Simple")
<Cell Simple: type CORTICAL>
* (element 'cortical)
<Cell Type CORTICAL>
*
```

We can verify that the associated global variables point to these elements:

```
* *CELL*
<Cell Simple: type CORTICAL>
* *CELL-TYPE*
<Cell Type CORTICAL>
*
```

Note that global variable names are symbols, but you do not use a preceeding quote (otherwise Lisp will simply return the symbol itself).

The inverse function of ELEMENT is ELEMENT-NAME. Because two different elements of different types may have the same name, ELEMENT has an optional TYPE argument. On the other hand, all elements of a given type must have distinct names, and typically Surf-Hippo takes care of this constraint automatically. Indeed, if you try to create an element with the same name of one already created, by default Surf-Hippo will modify the name as follows:

```
* (create-cell "Simple" :cell-type 'cortical :soma-diameter 30)
<Cell Simple-1: type CORTICAL>
```

For clarity, let's get rid of that interloper with ERASE-ELEMENT:

```
* *cell*
<Cell Simple-1: type CORTICAL>
* (erase-element *cell*)
NIL
* *cell*
<Cell Simple: type CORTICAL>
*
```

You can print out information about an element with PRINT-ELEMENT:

```
* (print-element "Simple")
Locating segments...

 Cell Simple: 0 branch points and 0 segments processed.
Cell Simple (soma @ [0.0 0.0 0.0])
    Membrane Area 2.83e+3um^2
    Passive somatic R-in (actual|cable model) = 1.41d+3|1.41d+3 Mohms

NIL
*
```

In this example, the "Locating segments... Cell Simple: 0 branch points and 0 segments processed." message is generated because the cell structure had to be processed before printing out its characteristics.

The CREATE-ELEMENT function creates and adds various circuit elements, such as channels and synapses, to different cell elements (somas and segments). Here we will add a Na^+ and a K^+ channel that are fitted to the classical Hodgkin-Huxley squid axon channels to the soma (in this example using the *SOMA* global variable), referenced by the symbols NA-HH-EXT and DR-HH-EXT:

```
* (create-element *soma* 'na-hh-ext 'dr-hh-ext)
(<Channel Simple-soma-NA-HH-EXT: type NA-HH-EXT>
 <Channel Simple-soma-DR-HH-EXT: type DR-HH-EXT>)
*
```

When CREATE-ELEMENT creates more than one element it returns them as a list, as in this example. Note that the definitions of the NA-HH-EXT and DR-HH-EXT channel types (and many others) are pre-loaded in Surf-Hippo.

The ELEMENT-TYPE function returns the element type of a given element. Here it returns a channel type object since the argument refers to a specific instance of a channel:

```
* (element-type "Simple-soma-DR-HH-EXT")
<Channel Type DR-HH-EXT>
NIL
*
```

Now we can use the return value of the ELEMENT-TYPE function as an argument for another function, EDIT-ELEMENT:

```
* (edit-element (element-type "Simple-soma-DR-HH-EXT"))

    [Menu appears]

NIL
*
```

Here the value returned by `ELEMENT-TYPE` was passed directly to the generic `EDIT-ELEMENT` function, which here in turn generates an editing menu for the channel type `DR-HH-EXT`.

Edit menus for different sorts of circuit elements allow access to most important object parameters. Of course, all element properties may be set programatically with the appropriate functions (see the User Manual).

In this example, we can examine the gating particle for this channel - from the bottom of the "Parameters of Channel Type DR-HH-EXT" menu, try plotting the "Steady State" and "Tau" of the `N-HH-EXT` (as always, type "OK" when ready). Now edit the particle parameters by selecting "Edit parameters for the particle types". This particular particle model is based on an extended Hodgkin-Huxley formulation, whose most important parameters are listed in the upper part of the menu. You can try modifying some of these (for example $V_{1/2}$ or Tau-0), and then see the effect by replotting the particle kinetic curves.

3.1 Lists of Elements

There are a variety of ways to access all the elements in a circuit of a given type. A simple way is to use one of a series of functions which return a list of the given kind of element:

```
* (channels)
(<Channel Simple-soma-DR-HH-EXT: type DR-HH-EXT>
 <Channel Simple-soma-NA-HH-EXT: type NA-HH-EXT>)
* (particle-types)
(<Particle Type N-HH-EXT> <Particle Type H-HH-EXT> <Particle Type M-HH-EXT>)
*
```

Similar functions exist for cells, cell types, segments, channel types, synapses, synapse types, particles, concentration-dependent particles, etc.

The `ELEMENT` function, as well as most functions which operate on an element (and have the word `ELEMENT` in their name, e.g. `PRINT-ELEMENT`) can take, as their first argument, either a single name or pointer to an element, or a list of names or pointers. Thus, these sorts of functions may be used on a list returned by a function such as `CHANNELS`:

```
* (enable-element-plot (channels))
NIL
*
```

Here we are flagging all the channels to plot their current after a simulation. Again, related functions can take a list of elements as arguments, or in many cases can take a symbol that represents some class of elements:

```
* (print-element 'particle-type)
Particle-type M-HH-EXT (class :HH-EXT):
    z 2.7 gamma 0.4 base-rate 1.2 V-1/2 -40.0 tau-0 (additive) (applied to entire kinetics) 0.
    There is 1 M-HH-EXT PARTICLE.
    Source file: /usr/local/surf-hippo/bin/parameters/hodgkin-huxley.fasl
Particle-type H-HH-EXT (class :HH-EXT):
    z -3.7 gamma 0.4 base-rate 0.07 V-1/2 -62.0 tau-0 (additive) (applied to entire kinetics)
    There is 1 H-HH-EXT PARTICLE.
    Source file: /usr/local/surf-hippo/bin/parameters/hodgkin-huxley.fasl
Particle-type N-HH-EXT (class :HH-EXT):
    z 1.5 gamma 0.9 base-rate 0.14 V-1/2 -51.0 tau-0 (additive) (applied to entire kinetics) 1
    There is 1 N-HH-EXT PARTICLE.
    Source file: /usr/local/surf-hippo/bin/parameters/hodgkin-huxley.fasl
```

4 Define a Simple Cell

Let us now look at a somewhat more complex cell definition in the file surf-hippo/circuits/demos/hh-demo.lisp. The cell has a soma and two dendritic cables As above, the Hodgkin-Huxley I_{DR} and I_{Na} channels are added

to the soma. In addition, a current source is added and configured, and various elements are flagged for plotting.

Note that the `LET` form binds a local variable `CELL` to the value returned by the function `CREATE-CELL`. The `CELL` variable may be then used later within the defining form to reference the created cell object, in particular to access the cell's soma with the function `CELL-SOMA`.

The functions used in the example may have more possible arguments than used here; for a complete description of any documented function see the Reference Manual, or for all functions, (under SLIME) type `C-c C-d d`, or call the `DESCRIBE` function on the function name. For example:

```
* (describe 'pulse-list)
PULSE-LIST is an internal symbol in the SURF-HIPPO package.
Function: #<Function PULSE-LIST {124C39B9}>
Function arguments:
  (source &optional (pulse-list nil pulse-list-supplied))
Function documentation:
  For adding a PULSE-LIST to SOURCE, where the format of PULSE-LIST is either:

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

4.1 Running Demo-Cell

Now load a predefined circuit from a file with `CIRCUIT-LOAD`:

```
* (circuit-load "circuits/demos/hh-demo.lisp")
Reading in circuit hh-demo...
; Loading #p"/usr/local/surf-hippo/circuits/demos/hh-demo.lisp".
Locating segments...

Cell Demo-Cell: 0 branch points and 8 segments processed.
NIL
*
```

This demo draws the cell (and the specified plotted elements) automatically. Note that if you give a filename to `CIRCUIT-LOAD` (it can also accept function names), then you can give either the full pathname, or if the file is under the Surf-Hippo home directory, just the path without specifying the home directory explicitly (as in the example above).

Try examining the soma characteristics on the new circuit, this time by referencing the global variable `*CELL*`:

```
* (print-element *cell*)
Cell Demo-Cell (soma @ [100.0 -300.0 50.0])  -  Created from hh-demo.lisp
    8 Segments, 2 Trunks, 2 Terminals, Membrane Area 5.85e+4um^2
    Passive somatic R-in (actual|cable model) = 7.43d+1|7.28d+1 Mohms
    R-Soma (passive) = 1.04d+3 Mohms
    Tree R-in (passive) (actual|cable model) = 8.00e+1|7.83e+1 Mohms
    Coupling R's from individual compartments [single-leg]
NIL
*
```

Now let us redraw the cell, but with some colors to the channel types:

```
* (element-parameter 'na-hh-ext 'color 'red)
RED
* (element-parameter 'dr-hh-ext 'color 'blue)
BLUE
* (just-draw :scale 5 :mark-elements :all :mark-all-nodes t)
```

You can now run the simulation either from the menus or from the interpreter using the function `RUN`:

```
* (run)
     [Simulation runs, output plotted]
NIL
*
```

At the end of the simulation several plots will appear. These data plots (and histology windows) allow a wide variety of point-and-click actions, as well as an extensive menu interface. Menus are brought up by typing Control-m over a window. For help with any given type of output window, type "h" over the window.

You can remove output windows by typing "d" over a window, which brings up a menu. Othewise, a simple way to clear all output windows is by running the function:

```
* (caows)
```

4.2 Loops - The Heart of Automatic Runs

Here is a simple loop that simulates the loaded cell with different current steps - you can try out this code directly from the interpreter. In this example the `LOOP` form is within the scope of a `LET` form. This is so that we can temporarily make local bindings (value assignments) of two global variables `*OVERLAY-PLOTS*` and `*ACCOMODATE-OVERLAYS*`, whose values will be restored after the `LET` form is completed. Note that comments are preceeded by a comma (or two).

```
* (let ((*overlay-plots* nil)             ; First simulation clears plots.
       (*accomodate-overlays* t))         ; Everything can fit.
    ;; Iterate using a local variable CURRENT.
    (loop for current from 0 to 1 by 0.2 do
          (add-pulse-list *isource* (list 10 20 current))   ; 10 to 20ms pulse
          (run)                                             ; Run simulation
          (setq *overlay-plots* t)))   ; Subsequent simulations overlay on each other.
```

If you kept the same plotting parameters as defined earlier, this will make a bit of a mess for the voltage plots. You can get a better view by using a waterfall format on the final voltage plot. With the mouse over the Voltages window, type Control-m to get a menu for the plot window. Click on "More edit and analysis options" and then "OK". Click both "Reorder/suppress traces" and "Waterfall" (under "Plot layout:") and then "OK". Rearrange the order of the traces as you like - typically in sequence from the soma out is more clear. Type "OK". Choose "Use automatic waterfall setup" etc. Hit "OK".

5 More Demos

There are several demo files in circuits/demos - you can load these quickly from the menus by entering the function (DEMO). Continue the tour by loading these files, and looking at them in an emacs buffer. For most of these circuits, loading the file will define the circuit and generate a histology plot. You should then enter (RUN) in general to run the defined simulations, or, use the menu system started by entering (SURF).

6 The Surf-Hippo Distribution: Important Directories

In the top level Surf-Hippo directory (surf-hippo/) these entries are particularly important:

- READ.ME.FIRST, Surf-Hippo.README (files) Obvious.
- anatomy (directory) Holds anatomy files, both originals and those processed for Surf-Hippo input.
- circuits (directory) Useful place to put your circuit definition code. Also includes various demo files under the demos/ directory.
- customs.lisp (file) This user edited file is loaded when Surf-Hippo is initialized.
- data (directory) Where simulation data is written by default.
- doc (directory) The documentation directory.
- lib/patches.lisp (file) This file should be updated with any system patches.
- logs (directory) Automatic logging of all simulations normally goes here.
- plot (directory) Postscript files from graphics windows normally go here.
- src (directory) All the source code.
- surf-hippo (file) The shell file which invokes the proper cmucl for your system.

7 Summary

This completes the short tour of Surf-Hippo. The next step is to consult the Users Manual, especially the early material introducing Lisp.

Surf-Hippo is available from the website surf-hippo.neurophysics.eu. If you are using the Surf-Hippo software, please send mail to surf-hippo@gmail.com to get on the mailing list.

Good luck!

Lyle J Graham (lyle@biomedicale.univ-paris5.fr)
Neurophysiology of Visual Computation, CNRS
Université Paris Descartes
45 rue des Saint-Peres
75006 Paris, France