Chapter 6

How to build and use models of individual cells

In **Chapter 2** we remarked that a conceptual model is an absolute prerequisite for the scientific application of computational modeling. But if a computational model is to be a fair test of our conceptual model, we must take special care to establish a direct correspondence between concept and implementation. To this end, the research use of NEURON involves all of these steps:

1. Implement a computational model of the biological system
2. Instrument the model
3. Set up controls for running simulations
4. Save the model with instrumentation and run controls
5. Run simulation experiments
6. Analyze results

These steps are often applied iteratively. We first encountered them in **Chapter 1**, and we will return to each of them repeatedly in the remainder of this book.

GUI vs. `hoc` code: which to use, and when?

At the core of NEURON is an interpreter which is based on the `hoc` programming language (Kernighan and Pike 1984). In NEURON, `hoc` has been extended by the addition of many new features, some of which improve its general utility as a programming language, while others are specific to the construction and use of models of neurons and neural circuits in particular. One of these features is a graphical user interface (GUI) which provides graphical tools for performing most common tasks. We have already seen that many of these tools are especially useful for model development and exploratory simulations (**Chapter 1**).

Prior to the advent of the GUI, the only way to use NEURON was by writing programs in `hoc`. For many users, convenience is probably reason enough to use the GUI. We should also mention that several of the GUI tools are quite powerful in their own right, with functionality that would require significant effort for users to recreate by writing their own `hoc` code. This is particularly true of the tools for optimization and electrotonic analysis.

But sooner or later, even the most inveterate GUI user may encounter situations that call for augmenting or replacing the default implementations provided by the GUI. Traditional programming allows maximum control over model specification, simulation

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

control, and display and analysis of results. It is also appropriate for noninteractive simulations, such as "production" runs that generate large amounts of data for later analysis.

So the answer to our question is: use the GUI *and* write hoc code, in whatever combination gets the job done with the greatest conceptual clarity and the least human effort. Each has its own advantages, and the most productive strategy for working with NEURON is to combine them in a way that exploits their respective strengths. One purpose of this book is to help you learn what these strengths are.

Hidden secrets of the GUI

There aren't any, really. All but one of the GUI tools are implemented in hoc, and all of the hoc code is provided with NEURON (see nrn-x.x/share/nrn/lib/hoc/ under UNIX/Linux, c:\nrnxx\lib\hoc\ in MSWindows). Thus the CellBuilder, the Network Builder, and the Linear Circuit Builder are all implemented in hoc, and each of them works by executing hoc statements in a way that amounts to creating hoc programs "on the fly." It can be instructive to examine the source code for these and NEURON's other GUI tools. A recurring theme in many of them is a sequence of hoc statements that construct a string, followed by a hoc statement that executes this string (if it is a valid hoc statement) or uses it as an argument to some other hoc function or procedure. We will return to this idea in **Chapter 14: How to modify NEURON itself**, which shows how to create new GUI tools and add new functions to NEURON.

The only GUI tool that is not implemented in hoc is the Print & File Window Manager, which is written in C. The source code for it is included with the UNIX distribution of NEURON.

Anything that can be done with a GUI tool can be done directly with hoc. To underscore this point, we will now use hoc statements to replicate the example that we built with the GUI in **Chapter 1**. Our code follows the same broad outline as before, specifying the model first, then instrumenting it, and finally setting up controls for running simulations. For clarity of presentation, we will consider this code in the same sequence: model implementation, instrumentation, and simulation control.

If you want to work along with this example, it would be a good idea to create an empty directory in which to save the file or files that you will make. These will be plain text files, which are also sometimes known as ASCII files. Begin by using a text editor to create a file called example.hoc that will contain the code.

Implementing a model with hoc

The properties of our conceptual model neuron are summarized in Fig. 6.1 and Tables 6.1 and 6.2. For the most part, the steps required to implement a computational model of this cell with hoc statements parallel what we did to build the model with NEURON's GUI; differences will be noted and discussed as they arise. In the following program listings, single line comments begin with a pair of forward slashes // and multiple line

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

comments begin with `/*` and are terminated by `*/`. For a discussion of `hoc` syntax, see **Chapter 12**.

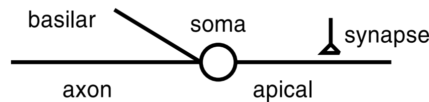

Fig. 6.1. The model neuron. The conductance change synapse can be located anywhere on the cell.

Table 6.1. Model cell parameters

	Length µm	Diameter µm	Biophysics
soma	30	30	HH g_{Na}, g_K, and g_{leak}
apical dendrite	600	1	passive with Rm = 5,000 Ω cm^2, E_{pas} = -65 mV
basilar dendrite	200	2	same as apical dendrite
axon	1000	1	same as soma

Cm = 1 µf / cm^2

cytoplasmic resistivity = 100 Ω cm

Temperature = 6.3 ^oC

Table 6.2. Synaptic mechanism parameters

g_{max}	0.05 µS
τ_s	0.1 ms
E_s	0 mV

Topology

Our first task is to map the branched architecture of this conceptual model onto the topology of the computational model. We want each unbranched neurite in the conceptual model to be represented by a corresponding section in the computational model, and this is done with a `create` statement (top of Listing 6.1). The `connect` statements attach these sections to each other so that the conceptual and computational models have the same shape. As we noted in **Chapter 5**, each section has a normalized position parameter which ranges from 0 at one end to 1 at the other. The `basilar` and `axon` sections arise from one end of the cell body while the `apical` section arises from the other, so they are attached by `connect` statements to the 0 and 1 ends of the `soma`, respectively.

This model is simple enough that its geometry and biophysical properties can be specified directly in `hoc` without having to resort to sophisticated strategies. Therefore we will not bother with subsets of sections, but proceed immediately to geometry.

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

```
////////////////////////
/* model specification */
////////////////////////

/////// topology ////////

create soma, apical, basilar, axon
connect apical(0), soma(1)
connect basilar(0), soma(0)
connect axon(0), soma(0)

/////// geometry ////////

soma {
  L = 30
  diam = 30
  nseg = 1
}

apical {
  L = 600
  diam = 1
  nseg = 23
}

basilar {
  L = 200
  diam = 2
  nseg = 5
}

axon {
  L = 1000
  diam = 1
  nseg = 37
}

/////// biophysics //////

forall {
  Ra = 100
  cm = 1
}

soma {
  insert hh
}

apical {
  insert pas
  g_pas = 0.0002
  e_pas = -65
}
```

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

```
basilar {
  insert pas
  g_pas = 0.0002
  e_pas = -65
}

axon {
  insert hh
}
```

Listing 6.1. The first part of `example.hoc` specifies the anatomical and biophysical attributes of our model.

Geometry

Each section of the model has its own length `L`, diameter `diam`, and discretization parameter `nseg`. The statements inside the block `soma { }` pertain to the `soma` section, etc. (the "stack of sections" syntax--see **Which section do we mean?** in **Chapter 5**). Since the emphasis here is on elementary aspects of model specification with `hoc`, we have assigned specific numeric values to `nseg` according to what we learned from prior use of the CellBuilder (see **Chapter 1**). A more general approach would be to wait until `L`, `diam`, and biophysical properties (`Ra` and `cm`) have been assigned, and then compute values for `nseg` based on a fraction of the AC length constant at 100 Hz (see **The d_lambda rule** in **Chapter 5**).

Biophysics

The biophysical properties of each section must be set up individually because we have not defined subsets of sections. Cytoplasmic resistivity `Ra` and specific membrane capacitance `cm` are supposed to be uniform throughout the model, so we use a `forall` statement to assign these values to each section.

The Hodgkin-Huxley mechanism `hh` and the passive mechanism `pas` are distributed mechanisms and are specified with `insert` statements (see **Distributed mechanisms** in **Chapter 5**). No further qualification is necessary for `hh` because our model cell uses its default ionic equilibrium potentials and conductance densities. However, the parameters of the `pas` mechanism in the `basilar` and `apical` sections differ from their default values, and so require explicit assignment statements.

Testing the model implementation

Testing is always important, especially when project development involves writing code. If you are working along with this example, this would be an excellent time to save what you have written to `example.hoc` and use NEURON to test it. Then, if you're using a Mac, just drag and drop `example.hoc` onto `nrngui`. Under MSWindows use Windows Explorer (the file manager, not Internet Explorer) to go to the directory where you saved `example.hoc` and double click on the name of the file. Under UNIX or Linux, type the command `nrniv example.hoc -` at the system prompt (we're

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

deliberately *not* typing `nrngui example.hoc`, to avoid having NEURON load its GUI library).

This will launch NEURON, and NEURON's interpreter will then process the contents of `example.hoc` and generate a message that looks something like this:

```
NEURON -- Version 5.6 2004-5-19 23:5:24 Main (81)
by John W. Moore, Michael Hines, and Ted Carnevale
Duke and Yale University -- Copyright 2001

oc>
```

The NEURON Main Menu toolbar will not appear under MSWindows, UNIX, or Linux. This happens because NEURON did not load its GUI library, which contains the code that implements the NEURON Main Menu. We're roughing it, remember? We trust that Mac users will pretend they don't see the toolbar, because dropping a `hoc` file on the `nrngui` icon automatically loads the GUI library.

Since we aren't using the CellBuilder, there isn't see a nice graphical summary of the model's properties. However a couple of `hoc` commands will quickly help you verify that the model has been properly specified.

We can check the branched architecture of our model by typing `topology()` at the `oc>` prompt (see **Checking the tree structure with topology()** in **Chapter 5**). This confirms that `soma` is the root section (i.e. the section that has no parent; note that this is *not* the same as the default section). It also shows that `apical` is attached to the 1 end of `soma`, and `basilar` and `axon` are connected to its 0 end.

```
oc>topology()

|-|       soma(0-1)
   `----------------------|       apical(0-1)
 `----|       basilar(0-1)
 `------------------------------------|       axon(0-1)

   1
oc>
```

The command `forall psection()` generates a printout of the geometry and biophysical properties of each section. The printout is in the form of `hoc` statements that, if executed, will recreate the model.

```
oc>forall psection()
soma { nseg=1  L=30  Ra=100
   /*location 0 attached to cell 0*/
   /* First segment only */
   insert morphology { diam=30}
   insert capacitance { cm=1}
   insert hh { gnabar_hh=0.12 gkbar_hh=0.036 gl_hh=0.0003 el_hh=-54.3}
   insert na_ion { ena=50}
   insert k_ion { ek=-77}
}
```

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

```
apical { nseg=23  L=600  Ra=100
   soma connect apical (0), 1
   /* First segment only */
   insert capacitance { cm=1}
   insert morphology { diam=1}
   insert pas { g_pas=0.0002 e_pas=-65}
}
basilar { nseg=5  L=200  Ra=100
   soma connect basilar (0), 0
   /* First segment only */
   insert capacitance { cm=1}
   insert morphology { diam=2}
   insert pas { g_pas=0.0002 e_pas=-65}
}
axon { nseg=37  L=1000  Ra=100
   soma connect axon (0), 0
   /* First segment only */
   insert capacitance { cm=1}
   insert morphology { diam=1}
   insert hh { gnabar_hh=0.12 gkbar_hh=0.036 gl_hh=0.0003 el_hh=-54.3}
   insert na_ion { ena=50}
   insert k_ion { ek=-77}
}
oc>
```

After verifying that the model specification is correct, exit NEURON by typing `quit()` in the interpreter window.

An aside: how does our model implementation in `hoc` compare with the output of the CellBuilder?

The `hoc` code we have just written is supposed to set up a model that has the same anatomical and biophysical properties as the model that we created in **Chapter 1** with the CellBuilder. We can confirm that this is indeed the case by starting a fresh instance of NEURON, using it to load the session file that we saved in **Chapter 1**, and then typing `topology()` and `forall psection()`. But the CellBuilder can also create a file containing `hoc` statements that, when executed, recreate the model cell. How do the statements in this computer-generated file compare with the `hoc` code that we wrote for the purpose of specifying this model?

To find out, let us retrieve the session file from **Chapter 1**, and then select the Management page of the CellBuilder. Next we click on the Export button (Fig. 6.2), and save all the topology, subsets, geometry, and membrane information to a file called `cell.hoc`. Executing the `hoc` statements in this file will recreate the model cell that we specified with the CellBuilder.

It is instructive to briefly review the contents of `cell.hoc`, which are presented in Listing 6.2. At first glance this looks quite complicated, and its organization may seem a bit strange--after all, `cell.hoc` is a computer-generated file, and this might account for its peculiarities. But let him who has never written an idiosyncratic line of code cast the first stone! Actually, `cell.hoc` is fairly easy to understand if, instead of attempting a line-by-line analysis from top to bottom, we focus on the flow of program execution.

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

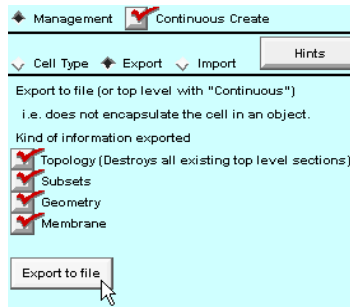

Figure 6.2. The Management page of the CellBuilder. We have clicked on the Export radio button, and are about to export the model's topology, subsets, geometry, and membrane information to a `hoc` file that can be executed to recreate the model cell.

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}

create soma, apical, basilar, axon

proc topol() { local i
  connect apical(0), soma(1)
  connect basilar(0), soma(0)
  connect axon(0), soma(0)
  basic_shape()
}

proc basic_shape() {
  soma {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(15, 0, 0, 1)}
  apical {pt3dclear() pt3dadd(15, 0, 0, 1) pt3dadd(75, 0, 0, 1)}
  basilar {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(-29, 30, 0, 1)}
  axon {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(-74, 0, 0, 1)}
}

objref all, has_HH, no_HH

proc subsets() { local i
  objref all, has_HH, no_HH
  all = new SectionList()
    soma all.append()
    apical all.append()
    basilar all.append()
    axon all.append()

  has_HH = new SectionList()
    soma has_HH.append()
    axon has_HH.append()
```

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

```
  no_HH = new SectionList()
    apical no_HH.append()
    basilar no_HH.append()
}

proc geom() {
  forsec all {  }
  soma {  L = 30  diam = 30  }
  apical {  L = 600  diam = 1  }
  basilar {  L = 200  diam = 2  }
  axon {  L = 1000  diam = 1  }
}

proc geom_nseg() {
  soma area(.5) // make sure diam reflects 3d points
  forsec all { nseg = int((L/(0.1*lambda_f(100))+.9)/2)*2 + 1  }
}

proc biophys() {
  forsec all {
    Ra = 100
    cm = 1
  }
  forsec has_HH {
    insert hh
      gnabar_hh = 0.12
      gkbar_hh = 0.036
      gl_hh = 0.0003
      el_hh = -54.3
  }
  forsec no_HH {
    insert pas
      g_pas = 0.0002
      e_pas = -65
  }
}

access soma

celldef()
```

Listing 6.2. The contents of `cell.hoc`, a file generated by exporting data from the CellBuilder that was used in **Chapter 1** to implement the model specified in Table 6.1 and 2 and shown in Fig. 6.1.

So we skip over the definition of `proc celldef()` to find the first statement that is executed:

```
create soma, apical, basilar, axon
```

Nothing too obscure about this. Next we jump over the definitions of two more `procs` (the temptingly simple `topol()` and the slightly puzzling `basic_shape()`) before encountering a declaration of three `objrefs` (see **Chapter 13: Object oriented programming**)

```
objref all, has_HH, no_HH
```

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

that are clearly used by the immediately following `proc subsets()` (what does it do? patience, all will be revealed . . .).

Finally at the end of the file we find a declaration of the default section, and then the procedure `celldef()` is called.

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}
```

This is the master procedure of this file. It invokes other procedures whose names remind us of that familiar sequence "topology, subsets, geometry, biophysics" before it ends with the eponymic `geom_nseg()`. Using `celldef()` as our guide, we can skim through the rest of the procedures.

- `topol()` first connects the sections to form the branched architecture of our model, and then it calls `basic_shape()`. The latter uses `pt3dadd` statements that are based on the shape of the stick figure that we saw in the CellBuilder itself. This establishes the orientations (angles) of sections, but the lengths and diameters will be superseded by statements in `geom()`, which is executed later.
- `subsets()` uses `SectionLists` to implement the three subsets that we defined in the CellBuilder (`all`, `has_HH`, `no_HH`).
- `geom()` specifies the actual physical dimensions of each of the sections.
- `biophys()` establishes the biophysical properties of the sections.
- `geom_nseg()` applies the discretization strategy we specified, which in this case is to ensure that no segment is longer than 0.1 times the length constant at 100 Hz (see **The d_lambda rule** in **Chapter 5**). This procedure is last to be executed because it needs to have the geometry and biophysical properties of the sections.

Instrumenting a model with hoc

The next part of `example.hoc` contains statements that set up a synaptic input and create a graphical display of simulation results (Listing 6.3). The synapse and the graph are specific instances of the `AlphaSynapse` and `Graph` classes, and are managed with object syntax (see **Chapter 13**). The synapse is placed at the middle of the `soma` and is assigned the desired time constant, peak conductance, and reversal potential. The graph will be used to show the time course of `soma.v(0.5)`, the somatic membrane potential.

> The strategy for dealing with synapses depends on the nature of the model. They are treated as part of the instrumentation in cellular and subcellular models, and there is indeed a sense in which they can be regarded as "physiological extensions" of the stimulating apparatus. However, synapses between cells in a network model are clearly intrinsic to the biological system. This difference is reflected in the GUI tools for constructing models of individual cells and networks.

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

```
/////////////////////////
/*   instrumentation   */
/////////////////////////

///// synaptic input ////

objref syn
soma syn = new AlphaSynapse(0.5)
syn.onset = 0.5
syn.tau = 0.1
syn.gmax = 0.05
syn.e = 0

/// graphical display ///

objref g
g = new Graph()
g.size(0,5,-80,40)
g.addvar("soma.v(0.5)", 1, 1, 0.6, 0.9, 2)
```

Listing 6.3. The second part of `example.hoc` specifies the instrumentation used to stimulate and monitor our model.

Setting up simulation control with `hoc`

The code in the last part of `example.hoc` controls the execution of simulations. This code must accomplish many tasks. It must define the size of the time step and the duration of a simulation. It also has to initialize the simulation, which means setting time to 0, making membrane potential assume its proper initial value(s) throughout the model, and ensuring that all gating variables and ionic currents are consistent with these conditions. Furthermore, it has to advance the solution from beginning to end and plot the simulation results on the graph. Finally, if interactive use is important, initializing and running simulations should be as easy as possible.

Setting up simulation control is a recurring task in developing computational models, and much effort can be wasted trying to reinvent the wheel. For didactic purposes, in this example we create our own simulation control code *de novo*. However, it is always far more efficient to use the powerful, customizable functions and procedures that are built into NEURON's standard run system (see **Chapter 7**).

The code in Listing 6.4 accomplishes these goals for our simple example. Simulation initialization and execution are generally performed by separate procedures, as shown here; the sole purpose of the final procedure is to provide the minor convenience that simulations can be initialized and executed by merely typing the command `go()` at the `oc>` prompt.

The first three statements in Listing 6.4 specify the default values for the time step, simulation duration, and initial membrane potential. However, initialization doesn't actually happen until you invoke the `initialize()` procedure, which contains statements that set time, membrane potential, gating variables and ionic currents to their proper initial values. The main computational loop that executes the simulation (`while`

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

`(t<tstop) { }`) is in the `integrate()` procedure, with additional statements that make the plot of somatic membrane potential appear in the graph.

```
/////////////////////////
/* simulation control  */
/////////////////////////

dt = 0.025
tstop = 5
v_init = -65

proc initialize() {
   t = 0
   finitialize(v_init)
   fcurrent()
}

proc integrate() {
   g.begin()
   while (t<tstop) {
      fadvance()
      g.plot(t)
   }
   g.flush()
}

proc go() {
   initialize()
   integrate()
}
```

Listing 6.4. The final part of `example.hoc` provides for initialization and execution of simulations.

Testing simulation control

Use NEURON to execute `example.hoc` (a graph should appear) and then type the command `go()` (this should launch a simulation, and a trace will appear in the graph). Change the value of `v_init` to -60mV and repeat the simulation (at the `oc>` prompt type `v_init=-60`, then type `go()`). When you are finished, type `quit()` in the interpreter window to exit NEURON.

Evaluating and using the model

Now that we have a working model, we are almost ready to put it to practical use. We have already checked that its sections are properly connected, and that we have correctly specified their biophysical properties. Although we based the number of segments on `nseg` generated by the CellBuilder using the d_lambda rule, we have not really tested discretization in space or time, so some exploratory simulations to evaluate the spatial and temporal grid are advisable (see **Chapter 4** and **Choosing a spatial grid** in **Chapter 5**). Once we are satisfied with its accuracy, we may be interested in improving simulation speed, saving graphical and numerical results, automating simulations and

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

data collection, curve fitting and model optimization. These are somewhat advanced topics that we will examine later in this book. The remainder of this chapter is concerned with practical strategies for working with models and fixing common problems.

Combining `hoc` and the GUI

The GUI tools are a relatively "recent" addition to NEURON (recent is a relative term in a fast-moving field--would you believe 1995?) so many published models have been implemented entirely in `hoc`. Also, many long-time NEURON users continue to work quite productively by developing their models, instrumentation, and simulation control exclusively with `hoc`. Often the resulting software is elegantly designed and implemented and serves its original purpose quite well, but applying it to new research questions can be quite difficult if significant revision is required.

Some of this difficulty can be avoided by generic good programming practices such as modular design, in particular striving to keep the specifications of the model, instrumentation, and simulation control separate from each other (see **Elementary project management** below). There is also a large class of problems that would require significant programming effort if one starts from scratch, but which can be solved with a few clicks of the mouse by taking advantage of existing GUI tools. But what if you don't see the NEURON Main Menu toolbar, or (as often happens when you first start to work with a "legacy" model) you do see it but many of the GUI tools don't seem to work?

No NEURON Main Menu toolbar?

This is actually the easiest problem to solve. At the `oc>` prompt, type the command `load_file("nrngui.hoc")` and the toolbar should quickly appear. If you add this statement to the very beginning of the `hoc` file, you'll never have to bother with it again.

`nrngui` also loads the standard run library

The toolbar will always appear if you use `nrngui` to load a `hoc` file. On the Mac this is what happens when you drag and drop a `hoc` file onto the `nrngui` icon. Under MSWindows you would have to start NEURON by clicking on its desktop `nrngui` icon (or on the `nrngui` item in the Start menu's NEURON program group), and then use NEURON Main Menu / File / load hoc to open the the `hoc` file. UNIX/Linux users can just type `nrngui` *`filename`* at the system prompt.

However, even if you see the toolbar, many of the GUI tools will not work if the `hoc` code didn't define a default section.

Default section? We ain't got no default section!

No badges, either. But to make full use of the GUI tools, you do need a default section. To see what happens if there isn't one, let's add a second synapse to the instrumentation of our example as if we were modeling feedforward inhibition. We could do this by writing `hoc` statements that define another point process, but this time let's use the GUI (see **4. Instrument the model. Signal sources** in **Chapter 1**).

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

First, change example.hoc by adding the statement

```
load_file("nrngui.hoc")
```

at the very beginning of the file. Now when NEURON executes the commands in example.hoc, the first thing that happens is the GUI library is loaded and the NEURON Main Menu toolbar appears.

UNIX/Linux users can go back to typing nrngui example.hoc.

But NEURON Main Menu / Tools / Point Processes / Managers / Point Manager doesn't work. Instead of a PointProcessManager we get an error message that there is "no accessed section" (Fig. 6.2). What went wrong, and how do we fix it?

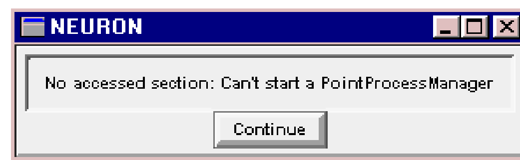

Fig. 6.2. A useful error message.

Many of the GUI tools, such as voltage graphs, shape plots, and point processes, must refer to a particular section at the moment they are spawned. This is because sections share property names, such as L and v. Remember the statement we used to create a point process in example.hoc:

```
soma syn = new AlphaSynapse(0.5)
```

This placed the newly created synapse at the 0.5 location on a particular section: the soma. But we're not writing hoc statements now; we're using a graphical tool (the NEURON Main Menu) to create another graphical tool that we will use to attach a point process to a section, and the NEURON Main Menu has no way to guess which section we're thinking about.

The way to fix this problem is to add the statement

```
access soma
```

to our model description, right after the create statement. The access statement defines the *default* section (see **Which section do we mean?** in **Chapter 5**). If we assign membrane properties or attach a point process to a model, the default section is affected unless we specify otherwise. And if we use the GUI to create a plot of voltage vs. time, v at the middle of the default section is automatically included in the list of things that are plotted.

If there are many sections, which one should be the default section? A good rule of thumb is to pick a conceptually privileged section that will get most of the use. The soma is generally a good choice.

So click on the "Continue" button to dismiss the error message, quit NEURON, add the access soma statement to example.hoc, and try again. This time it works. Configure the PointProcessManager to be an AlphaSynapse with onset = 0.5 ms, tau = 0.3 ms, gmax = 0.04 µS, and e = -70 mV and type

Scientific question: can you explain the effect of the inhibitory synapse's tau on cell firing?

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

go() to run a simulation. Run a couple more simulations with tau = 1 ms and 3 ms. Then exit NEURON.

Strange Shapes?

The barbed wire model

In **Chapter 1** we mentioned that the 3-D method for specifying geometry can be used to control the appearance of a model in a Shape plot. The benefits of the 3-D method for models based on detailed morphometric data are readily appreciated: the direct correspondence between the anatomy of the cell as seen under a microscope, and its representation in a Shape plot, can assist conceptual clarity when specifying model properties and understanding simulation results. Perhaps less obvious, but no less real, is the utility of the 3-D method for dealing with more abstract models, whose geometry is easy enough to specify in terms of L and diam. We hinted at this in the walkthrough of the hoc code exported by the CellBuilder, but a few examples will prove its value and at the same time help prevent misapplication and misunderstanding of this approach.

Suppose our conceptual model is a cell with an apical dendrite that gives rise to 10 oblique branches along its length. For the sake of visual variety, we will have the lengths of the obliques increase systematically with distance from the soma. Listing 6.5 presents an implementation of such a model using L and diam to specify geometry. The apical trunk is represented by the proximal section apical and the sequence of progressively more distal sections ap[0] - ap[NDEND-1]. With our mind's eye, aided perhaps by dim recollection of Ramon y Cajal's marvelous drawings, we can visualize the apical trunk stretching away from the soma in a more or less straight line, with the obliques coming off at an angle to one side.

```
/////// topology ////////

NDEND = 10

create soma, apical, dend[NDEND], oblique[NDEND]
access soma

connect apical(0), soma(1)
connect ap[0](0), apical(1)
connect oblique[0](0), apical(1)

for i=1,NDEND-1 {
  connect ap[i](0), ap[i-1](1)
  connect oblique[i](0), dend[i-1](1)
}

/////// geometry ////////

soma { L = 30 diam = 30 }

apical { L = 3 diam = 5 }
```

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

```
for i=0,NDEND-1 {
  ap[i] { L = 15 diam = 2 }
  oblique[i] { L = 15+5*i diam = 1 }
}
```

Listing 6.5. Implementation of an abstract model that has a moderate degree of dendritic branching using `L` and `diam` to specify geometry.

But executing this code and bringing up a Shape plot (e.g. by NEURON Main Menu / Graph / Shape plot) produces the results shown in Figure 6.3. So much for our mind's eye. Where did all the curvature of the apical trunk come from?

This violence to our imagination stems from the fact that stylized specification of model geometry says nothing about the orientation of sections. At every branch point, NEURON's internal routine for rendering shapes makes its own decision, and in doing so it follows a simple rule: make a fork with one child pointing to the left and the other to the right by the same amount relative to the orientation of the parent. Models with more complex branching patterns can look even stranger; if the detailed architecture of a real neuron is translated to simple `hoc` statements that assert nothing more than connectivity, length, and diameter, the resulting Shape may resemble a tangle of barbed wire.

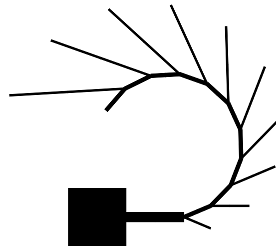

Fig. 6.3. Shape plot rendering of the model produced by the code in Listing 6.5. To help indicate the location of the soma section, Shape Style: Show Diam was enabled.

To gain control of the graphical appearance of our model, we must specify its geometry with the 3-D method. This is illustrated in Listing 6.6, where we have meticulously used absolute (x,y,z) coordinates, based on the actual location of each section, as arguments for the pt3dadd() statements. Now when we bring up a Shape plot, we get what we wanted: a nice, straight apical trunk with oblique branches coming off to one side (Fig. 6.4).

```
/////// geometry ////////

forall pt3dclear()

soma {
  pt3dadd(0, 0, 0, 30)
  pt3dadd(30, 0, 0, 30)
}

apical {
  pt3dadd(30, 0, 0, 5)
  pt3dadd(60, 0, 0, 5)
}
```

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

```
for i=0,NDEND-1 {
  ap[i] {
    pt3dadd(60+i*15, 0, 0, 2)
    pt3dadd(60+(i+1)*15, 0, 0, 2)
  }
  oblique[i] {
    pt3dadd(60+i*15, 0, 0, 1)
    pt3dadd(60+i*15, -15-5*i, 0, 1)
  }
}
```

Listing 6.6. Specification of model geometry using the 3-D method. This assumes the same model topology as shown in Listing 6.5.

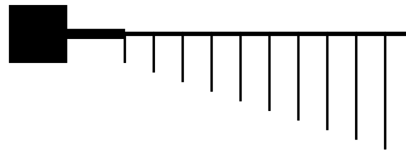

Fig. 6.4. Shape plot rendering of the model when the geometry is specified using the 3-D method shown in Listing 6.6.

Although we scrupulously used absolute (x,y,z) coordinates for each of the sections, we could have saved some effort by taking advantage of the fact that the root section is treated as the origin of the cell with respect to 3-D position. When any section's 3-D shape or length changes, the 3-D information of all child sections is translated to correspond to the new position. Thus, if the soma is the root section, we can move an entire cell to another location just by changing the location of the soma. Another useful implication of this feature allows us to simplify our model specification: the only `pt3dadd()` statements that must use absolute coordinates are those that belong to the root section. We can use relative coordinates for all child sections, instead of absolute (x,y,z) coordinates, as long as they result in proper length and orientation (see Listing 6.7).

```
/////// geometry ////////

forall pt3dclear()

soma {
  pt3dadd(0, 0, 0, 30)
  pt3dadd(30, 0, 0, 30)
}

apical {
  pt3dadd(0, 0, 0, 5)
  pt3dadd(30, 0, 0, 5)
}
```

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

```
for i=0,NDEND-1 {
  ap[i] {
    pt3dadd(0, 0, 0, 2)
    pt3dadd(15, 0, 0, 2)
  }
  oblique[i] {
    pt3dadd(0, 0, 0, 1)
    pt3dadd(0, -15-5*i, 0, 1)
  }
}
```

Listing 6.7. A simpler 3-D specification of model geometry that relies on the absolute coordinates of the root section and relative coordinates of all child sections. Compare the (x,y,z) coordinates in the `pt3dadd()` statements for `apical`, `ap`, and `oblique` with those in Listing 6.6.

The case of the disappearing section

In **Chapter 5** we mentioned that it is generally a good idea to attach the 0 end of a child section to its parent, in order to avoid confusion. For an example of one particularly vexing problem that can arise when this recommendation is ignored, consider Listing 6.8. The `access dend[0]` statement and the arguments to the `pt3dadd()` statements suggest that the programmer's conceptual model had the sections arranged in the left to right sequence `dend[0]` - `dend[1]` - `dend[2]`. Note that the 1 end of `dend[0]` is connected to the 0 end of `dend[1]`, and the 1 end of `dend[1]` is connected to the 0 end of `dend[2]`. This means that `dend[2]`, which is not connected to anything, is the root section. From a purely computational standpoint this is perfectly fine, and if we simulate the effect of a current step applied to the 0 end of `dend[0]`, there will be an orderly spread of charge and potential along each section from its 0 end to its 1 end, with the largest membrane potential shift in `dend[0]` and the smallest in `dend[2]`.

```
/////// topology ////////

NDEND = 3

create dend[NDEND]
access dend[0]

connect dend[0](1), dend[1](0)
connect dend[1](1), dend[2](0)

/////// geometry ////////

forall pt3dclear()

dend[0] {
  pt3dadd(0, 0, 0, 1)
  pt3dadd(100, 0, 0, 1)
}

dend[1] {
  pt3dadd(100, 0, 0, 1)
  pt3dadd(200, 0, 0, 1)
}
```

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

```
dend[2] {
  pt3dadd(200, 0, 0, 1)
  pt3dadd(300, 0, 0, 1)
}
```

Listing 6.8. The programmer's intent seems to be for `dend[0]`, `dend[1]`, and `dend[2]` to line up from left to right. However, the connect statements make `dend[2]` the root section, and thereby hangs a tale.

However, we're in for a surprise when we bring up a PointProcessManager (NEURON Main Menu / Tools / Point Processes / Managers / Point Manager) and try to place an `IClamp` at different locations in this model. No matter where we click, we can only put the `IClamp` on `dend[0]` or `dend[2]` (Fig. 6.5). Try as we might to find it, there just doesn't seem to be any `dend[1]`!

But `dend[1]` really does exist, and we can easily prove this by invoking the `topology()` function, which generates this diagram:

```
|-|       dend[2](0-1)
 `|       dend[1](1-0)
   `|       dend[0](1-0)
```

This not only confirms the existence of `dend[1]`, but also shows that `dend[2]` is the root section, with the 1 end of `dend[1]` connected to its to the 0 end, and the 1 end of `dend[0]` connected to the 0 end of `dend[1]`. Exactly as we expected, and just as specified by the code in Listing 6.8.

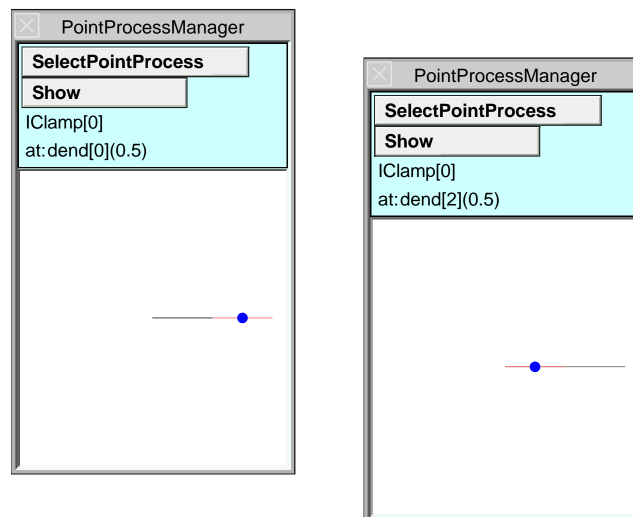

Fig. 6.5. The code in Listing 6.8 produces a model that seems not to have a `dend[1]`--or at least, we can't find `dend[1]` when we try to use a PointProcessManager to attach an `IClamp` to it.

But isn't something terribly wrong with the appearance of our model in the Shape plot? Not at all. Although we might not like it, the model looks exactly as it should, given the statements in Listing 6.8.

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

Here's why. As we mentioned above in **The barbed wire model**, the location of the root section determines the placement of all other sections. The root section is `dend[2]`, and the `pt3dadd()` statements in Listing 6.8 place its 0 end at (200, 0, 0) and its 1 end at (300, 0, 0) (Fig. 6.6).

Since `dend[1]` is attached to the 0 end of `dend[2]`, the first 3-D data point of `dend[1]` is mapped to (200, 0, 0) (see **3-D specification** in **Chapter 5**). According to the `pt3dadd()` statements for `dend[1]`, its last 3-D data point lies 100 µm to the right of its first 3-D point. This means that the 1 end of `dend[1]` is at (200, 0, 0) and its 0 end is at (300, 0, 0) (Fig. 6.6)--precisely the locations of the left and right ends of `dend[2]`! So `dend[1]` and `dend[2]` will appear as the same line in the Shape plot. When we try to select one of these sections by clicking on this line, the section we get will depend on the inner workings of NEURON's GUI library. It just happens that, for the particular `hoc` statements in Listing 6.8, we can only select points on `dend[2]`. This is as if `dend[1]` is hidden from view and shielded from our mouse cursor.

Finally we consider `dend[0]`, whose 1 end is connected to the 0 end of `dend[1]`. Thus its first 3-D data point is drawn at (300, 0, 0), and, following its `pt3dadd()` statements, its last 3-D data point lies 100 µm to the right, i.e. at (400, 0, 0). Thus `dend[0]` runs from (400, 0, 0) (its 0 end) to (300, 0, 0) (its 1 end), which is just to the right of `dend[2]` and the hidden `dend[1]` (Fig. 6.6).

So the mystery is solved. All three sections are present, but two are on top of each other.

The first lesson to take from this sad tale is the usefulness of `topology()` as a means for diagnosing problems with model architecture. The second lesson is the importance of following our recommendation to avoid confusion by connecting the 0 end of a child section to its parent. The strange appearance of the model in the Shape plot happened entirely because this advice was not followed. There are probably occasions in which it makes excellent sense to violate this simple rule; please be sure to let us know if you find one.

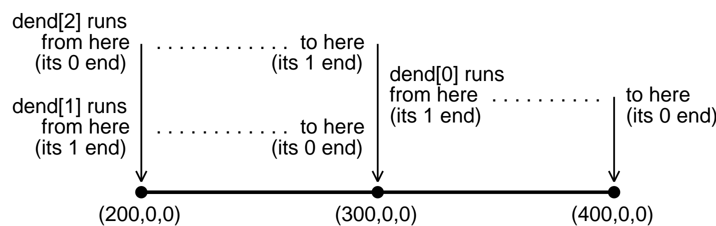

Fig. 6.6. Deciphering the `pt3dadd()` statements in Listing 6.8 leads us to realize that we only see two sections in the Shape plot because two of them (`dend[1]` and `dend[2]`) are drawn in the same place. This figure shows the (x,y,z) coordinates of the sections and indicates their 0 and 1 ends.

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

Graphs don't work?

If there is no default section, new graphs created with the GUI won't work properly. You've already seen how to declare the default section, so everything should be OK, right? Let's see for ourselves.

Make sure that `example.hoc` starts with `load_file("nrngui.hoc")` and contains an `access soma` statement, and then use NEURON to execute it. Then follow the steps shown in Fig. 1.27 (see **Signal monitors** in **Chapter 1**) to create a space plot that will show membrane potential along the length of the cell. Now type `go()`. What happens?

The graph of `soma.v(0.5)` shows an action potential, but the trace in the space plot remains a flat line. Is there something wrong with the space plot, or does the problem lie elsewhere?

To find out, use NEURON Main Menu / Tools / RunControl to bring up a RunControl window. Click on the RunControl's Init & Run button. Result: this time it's the space plot that works, and the graph of `soma.v(0.5)` that doesn't (Init & Run should have erased the trace in the latter and drawn a new one).

So there are actually two problems. The simulation control code in our `hoc` file can't update new graphs that we create with the GUI, and the GUI's own simulation control code can't update the "old" graph that is created by our `hoc` file. Of the many possible ways to deal with these problems, one is ridiculously easy and another requires a little effort (but only a very little).

The ridiculously easy solution is to use the GUI to make a new graph that shows the same variables, and ignore or throw away the old graph. In this example, resorting to NEURON Main Menu / Graph / Voltage axis gets us a new graph. Since the `soma` is the default section, the `v(.5)` that appears automatically in our new graph is really `soma.v(0.5)`.

What if a lot of programming went into one or more of the old graphs, so the GUI tools offer nothing equivalent? This calls for the solution that requires a little effort: specifically, we add a single line of `hoc` code for each old graph that needs to be fixed. In this example we would revise the code that defines the old graph by adding the line shown here in bold:

```
/// graphical display ///

objref g
g = new Graph()
addplot(g, 0)
g.size(0,5,-80,40)
g.addvar("soma.v(0.5)", 1, 1, 0.6, 0.9, 2)
```

Listing 6.9. Fixing an old graph so it works with NEURON's standard run system.

This takes advantage of NEURON's *standard run system*, a set of functions and procedures that orchestrate the execution of simulations (see **Chapter 7**). The statement `addplot(g, 0)` adds `g` to a list of graphs that the standard run system automatically

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

updates during the course of a simulation. Also, the x-axis of our graph will be adjusted automatically when we change `tstop` (Tstop in the RunControl panel). NEURON's GUI relies heavily on the standard run system, and every time we click on the RunControl's Init & Run button we are actually invoking routines that are built into the standard run system.

The standard run system has many powerful features and can be used in any simulation, with or without the GUI. The statement `load_file("stdrun.hoc")` loads the `hoc` code that implements the standard run system, without loading the GUI.

Does this mean that we have to abandon the simulation control code in our `hoc` program, and does it matter if we do? The control code in `example.hoc` performs a "plain vanilla" initialization and simulation execution, so abandoning it in favor of the standard run system only makes things better by providing additional functionality. But what if we want a customized initialization or some unusual flow of simulation execution? As we shall see in **Chapter 7**, the standard run system was designed and implemented so that only minor changes are required to accommodate most special needs.

Conflicts between hoc code and GUI tools

Many of the GUI tools specify properties of the model or the interface, and this leads to the possibility of conflicts that cause a mismatch between what you *think* is in the computer, and what *actually* is in the computer. For example, suppose you use the CellBuilder to construct a model cell with a section called `dend` that has `diam` = 1 µm, `L` = 300 µm, and passive membrane, and you turn Continuous create ON. Then typing `dend psection()` at the `oc>` prompt will produce something like this

```
oc>dend psection()
dend { nseg=11  L=300  Ra=80
         . . .
   insert pas { g_pas=0.001 e_pas=-70}
}
```

(a few lines were omitted for clarity), which confirms the presence of the `pas` mechanism.

A bit later, you decide to make `dend` active and get rid of its `pas` mechanism. You could do this with the CellBuilder, but let's say you find it quicker just to type

```
oc>dend {uninsert pas insert hh}
```

and then confirm the result of your action with another `psection()`

```
oc>dend psection()
dend { nseg=11  L=300  Ra=80
         . . .
   insert hh { gnabar_hh=0.12 gkbar_hh=0.036 gl_hh=0.0003 el_hh=-54.3}
   insert na_ion { ena=50}
   insert k_ion { ek=-77}
}
```

So far, so good.

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

But check the Biophysics page of the CellBuilder, and you will see that the change you accomplished with hoc did not track back into the GUI tool, which still shows dend as having pas but not hh. This is particularly treacherous, because it is all too easy to become confused about what is the actual specification of the model. If these new biophysical properties lead to particularly interesting simulation results, you might save "everything" to a session file, thinking that you would be able to reproduce those results in the future--but the session file would only contain the state of the GUI tools. Completely absent would be any reflection of the fact that you had executed your own hoc statement to override the CellBuilder's model specification.

And still more surprises are in store. Using the CellBuilder, with Continuous create still ON, change dendritic diameter to 2 µm. Now use psection() to check the properties of dend

```
oc>dend psection()
dend { nseg=7  L=300  Ra=80
          . . .
    insert hh { gnabar_hh=0.12 gkbar_hh=0.036 gl_hh=0.0003 el_hh=-54.3}
    insert na_ion { ena=50}
    insert k_ion { ek=-77}
    insert pas { g_pas=0.001 e_pas=-70}
}
```

and you see that *both* pas and hh are present, despite the previous use of uninsert to get rid of the pas mechanism.

Similar conflicts can arise between hoc statements and other GUI tools (e.g. the PointProcessManager) All of these problems have a common source: changes you make at the hoc level are not propagated to the GUI tools, so if you then make any changes with the GUI tools, it is likely that all the changes you made with hoc statements will be lost. The lesson here is to exercise great caution when combining GUI tools and hoc statements, in order to avoid introducing potentially confusing conflicts.

Conflicts may also occur between the CellBuilder and older GUI tools for managing section properties.

Elementary project management

The example used in this chapter is simple so all of its code fits in a single, small file that can be quickly understood. Nonetheless, we were careful to organize example.hoc in a way that separates specification of the model *per se* from the specification of the interface, i.e. the instrumentation and control procedures for running simulations. This separation maximizes clarity and reduces effort, and it should begin while the model is still in the conceptual stage.

Designing a model starts by answering the questions: what anatomical features are necessary, and what biophysical properties should be included? The answers to these questions govern key decisions about what what kinds of stimuli to apply, what kinds of measurements to make, and how to display, record, and analyze these measurements. When it is finally time to implement the computational model, it is a good idea to try to

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

keep these questions separate. This is the way NEURON's graphical tools are organized, and this is the way models specified with hoc should be organized.

- First you create a model, specifying its topology, geometry, and biophysics, either with the CellBuilder or with hoc code. This is a representation of selected aspects of a biological system, and you might think of it as a virtual experimental preparation.
- Then you instrument that model. This is analogous to applying stimulating and recording electrodes and other apparatus to a real neuron or neural circuit in the laboratory.
- Finally, you set up controls for running simulations.

Instrumentation and simulation controls are the user interface for exercising the model. Metaphorically speaking, they amount to a virtual experimental rig. In a wet lab, noone would ever confuse a brain slice with the microscope or instrumentation rack. The physical and conceptual distinction between biological preparation and experimental rig them is an inescapable fact and has a strong bearing on the and execution of experiments. NEURON lets you carry this separation over into modeling. Why confound the code that defines the properties of a model cell with the code that generates a stimulus or governs the sequence of events in a simulation?

One way to help separate model specification from user interface is to put the code that defines them into separate files. One file, which we might call cell.hoc, would contain the statements that specify the properties of the model: its topology, geometry, and biophysics. The code that defines point processes, graphs, other instrumentation, and simulation controls would go into a second file that we might call rig.hoc. Finally, we would use a third file for purely administrative purposes, so that a single command will make NEURON execute the other files in proper sequence. This file, which we might call init.hoc, would contain only the statements shown in Listing 6.10. Executing init.hoc with NEURON will make NEURON load its GUI and standard run libraries, bring up a NEURON Main Menu toolbar, execute the statements in cell.hoc to reconstitute the model cell, and finally execute the statements in rig.hoc to reconstitute our user interface for exercising the model.

```
load_file("nrngui.hoc")
load_file("cell.hoc")
load_file("rig.hoc")
```

Listing 6.10. Contents of init.hoc.

For instance, we could recast example.hoc in this manner by putting its model specification component into cell.hoc, while the instrumentation and simulation control components would become rig.hoc. This would allow us to reuse the same model specification with different instrumentation configurations rig1.hoc, rig2.hoc, etc.. To make it easy to select which rig is used, we could create a corresponding series of init files (init1.hoc, init2.hoc, etc.) that differ only in the argument to the third load_file() statement. This strategy is not limited to hoc files, but can also be used to retrieve cells and/or interfaces that have been constructed with the GUI and saved to session (ses) files.

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

Iterative program development

A productive strategy for program development in NEURON is to revise and reinterpret `hoc` code and/or GUI tools repeatedly during the same session. Bugs afflict all nontrivial programs, and the process of making incremental changes, saving them to intermediate `hoc` or `ses` files, and testing at each step, reduces the difficulty of trying to diagnose and eliminate them. In this way it is possible begin with a program skeleton that consists of one or two `hoc` files with a handful of `load_file()` statements and function stubs, and quickly refine it until everything works properly. However, two caveats do apply.

First, a variable cannot be declared with a new type during the same session. In other words, "once a scalar, always a scalar" (or double, or string, or object reference). Attempting to redeclare a variable will produce an error message, e.g.

```
oc>x = 3
first instance of x
oc>objref x
/usr/local/nrn/i686/bin/nrniv: x  already declared near line 2
objref x
        ^
oc>
```

Trying to redefine a double, string, or object reference as something else will likewise fail. This is generally of little consequence, since it is rarely absolutely necessary to change the type assigned to a particular variable name. When this does happen, you just have to exit NEURON, make your changes to the `hoc` code, and restart.

The second caveat is that, once the `hoc` interpreter has parsed the code in a template (see **Chapter 13: Object-oriented programming**), the class that it defines is fixed for that session. This means that any changes to a template require exiting NEURON and restarting. The result is some inconvenience when developing and testing new classes, but this is still easier than having to recompile and link a program in C or C++.

References

Kernighan, B.W. and Pike, R. Appendix 2: Hoc manual. In: *The UNIX Programming Environment*. Englewood Cliffs, NJ: Prentice-Hall, 1984, p. 329-333.

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

Chapter 6 Index

3-D specification of geometry 15
coordinates
absolute vs. relative 16, 17
A
access 14
B
biophysical properties
specifying 5
C
CellBuilder
hoc output
exported cell 7
CellBuilder GUI
Continuous create 22, 23
Management page
Export 7
computational model
implementing with hoc 2
conceptual clarity 2, 15
connect 3
create 3
D
diam 5
distributed mechanism 5
E
error message
no accessed section 14
G
good programming style
iterative development 25
modular programming 13

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

program organization 23
separate model specification from user interface 24
GUI
combining with hoc 13
conflicts with hoc or other GUI tools 22
tools
are implemented in hoc 2
work by constructing hoc programs 2
vs. hoc 1
H
hoc 1
can do anything that a GUI tool can 2
combining with GUI 13
conflicts with GUI 22
idiom
forall psection() 6
load_file("nrngui.hoc") 13
implementing a computational model 2
vs. GUI 1
hoc syntax
comments 2
variables
cannot change type 25
I
initialization 11
custom 22
insert 5
instrumentation 23
L
L 5
M
model

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

computational
essential steps 1
correspondence between conceptual and computational 1, 3
testing 12
model specification 23
as virtual experimental preparation 24
N
NEURON
starting with a specific hoc file 5
NEURON Main Menu
creating 13, 24
nrngui 6
loads GUI and standard run library 13
nrniv 5
nseg 5
P
plain text file 2
PointProcessManager
creating 19
project management 23
Q
quantitative morphometric data 15
R
RunControl
creating 21
RunControl GUI
Init & Run 22
Tstop 22
S
section
child
connect 0 end to parent 18

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

currently accessed
default section 14
orientation 10, 16, 17
root section 6
is 3-D origin of cell 17, 20
vs. default. section 6
SectionList class 10
Shape plot
creating 16
Shape plot GUI
Shape Style
Show Diam 16
simulation control 11, 23
standard run system 21
addplot() 21
tstop 22
stylized specification of geometry 5
strange shapes 15
synapse
as instrumentation 10
T
template
cannot be redefined 25
topology
checking 6, 20
specifying 3
topology, subsets, geometry, biophysics 10
topology() 20
troubleshooting
disappearing section 18
Graphs don't work 21
legacy code 13

Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved

no default section 13
no NEURON Main Menu toolbar 13
U
uninsert 23
user interface 23
as virtual experimental rig 24

 Copyright © 2001-2004 N.T. Carnevale and M.L. Hines, all rights reserved