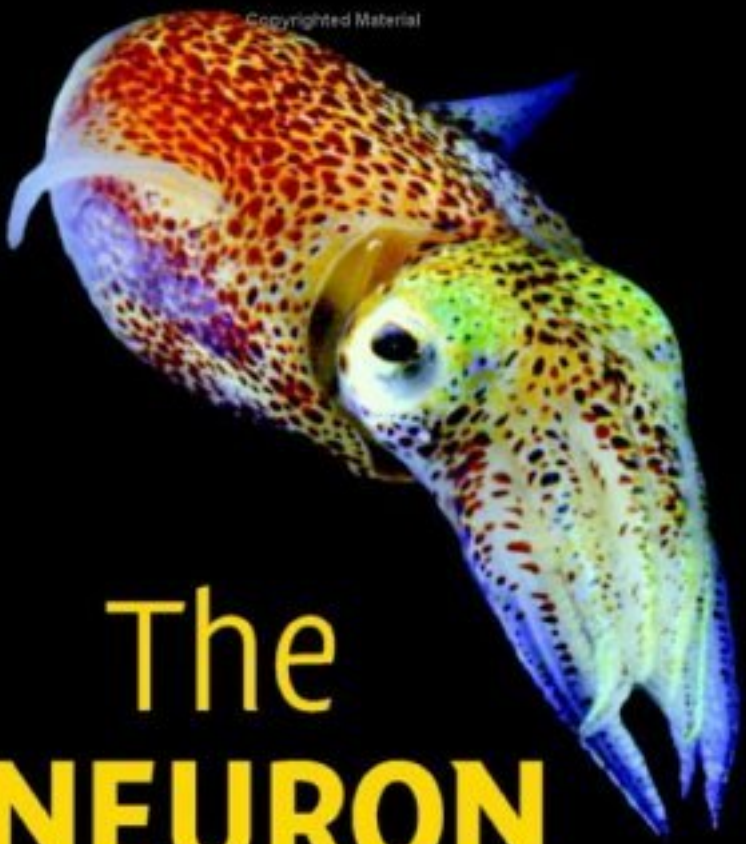


The NEURON Book

Nicholas T. Carnevale
and Michael L. Hines

CAMBRIDGE

Preface to The NEURON Book

N.T. Carnevale[1] and M.L. Hines[2]

Departments of [1]Psychology and [2]Computer Science

Yale University, New Haven, CT

ted.carnevale@yale.edu

michael.hines@yale.edu

Who should read this book

This book is about how to use the NEURON simulation environment to construct and apply empirically-based models of neurons and neural networks. It is written primarily for neuroscience investigators, teachers, and students, but readers with a background in the physical sciences or mathematics who have some knowledge about brain cells and circuits and are interested in computational modeling will also find it helpful. The emphasis is on the most productive use of NEURON as a means for testing hypotheses that are founded on experimental observations, and for exploring ideas that may lead to the design of new experiments. Therefore the book uses a problem-solving approach, with many working examples that readers can try for themselves.

What this book is, and is not, about

Formulating a *conceptual model* is an attempt to capture the essential features that underlie some particular function. This necessarily involves simplification and abstraction of real-world complexities. Even so, one may not necessarily understand all

implications of the conceptual model. To evaluate a conceptual model it is often necessary to devise a hypothesis or test in which the behavior of the model is compared against a prediction. *Computational models* are useful for performing such tests. The conceptual model and the hypothesis should determine what is included in a computational model and what is left out. This book is not about how to come up with conceptual models or hypotheses, but instead focuses on how to use NEURON to create and use computational models as a means for evaluating conceptual models.

What to read, and why

The first chapter conveys a basic idea of NEURON's primary domain of application by guiding the reader through the construction and use of a model neuron. This exercise is based entirely on NEURON's GUI, and requires no programming ability or prior experience with NEURON whatsoever.

The second chapter considers the role of computational modeling in neuroscience research from a general perspective. Chapters 3 and 4 focus on aspects of applied mathematics and numerical methods that are particularly relevant to computational neuroscience. Chapter 5 discusses the concepts and strategies that are used in NEURON to simplify the task of representing neurons, which (at least at the level of synapses and cells) are distributed and continuous in space and time, in a digital computer, where neither time nor numeric values are continuous. Chapter 6 returns to the topic of model construction, emphasizing the use of programming.

Chapters 7 and 8 provide "inside information" about NEURON's standard run and initialization systems, so that readers can make best use of their features and customize

them to meet special modeling needs. Chapter 9 shows how to use the NMODL programming language to add new biophysical mechanisms to NEURON. This theme continues in Chapter 10, which starts with mechanisms of communication between cells (gap junctions, graded and spike-triggered synaptic transmission), and moves on to models of artificial spiking neurons (e.g. integrate and fire cells). The first half of Chapter 11 is a tutorial on NEURON's GUI tools for creating simple network models, and the second half shows how to use the strengths of the GUI and hoc programming to create more complex networks.

Chapter 12 discusses the elementary features of the hoc programming language itself. Chapter 13 describes the object-oriented extensions that have been added to hoc. These extensions have greatly facilitated construction of NEURON's GUI tools, and they can also be very helpful in many other complex programming tasks such as creating and managing network models. Chapter 14 presents an example of how to use object oriented programming to increase the functionality of NEURON.

Appendix 1 presents a mathematical analysis of the IntFire4 artificial spiking cell mechanism, proving a result that is used to achieve computational efficiency when simulating this model. Appendix 2 summarizes the commands for NEURON's built-in text editor.

Acknowledgments

First and foremost, we want to thank our mentor and colleague John W. Moore for his vision, support, encouragement, and active participation in the development of NEURON, without which neither it nor this book would exist. Through his research and

teaching, he was introducing students to "computational neuroscience" long before that glorious term was invented. NEURON had its beginnings in John's laboratory at Duke University almost three decades ago, when he and one of the authors (MLH) started their collaboration to develop simulation software for neuroscience research. Users of NEURON on the Macintosh owe John a particular debt. He continues to participate in the development and dissemination of NEURON, concentrating most recently on educational applications in collaboration with Ann Stuart (Moore and Stuart 2004).

The list of those who have added in one way or another to the development of NEURON is far too long for this short preface. Zach Mainen, Alain Destexhe, Bill Lytton, Terry Sejnowski, and Gordon Shepherd deserve special mention for many contributions, both direct and indirect, that range from specific enhancements to the program, to fostering the wider acceptance of computational approaches in general, and NEURON in particular, by the neuroscience community at large. We also thank the countless NEURON users whose questions and suggestions continue to help guide the evolution of this software and its documentation. We hope that everyone else will forgive any omission and remind us, gently, in time for the second edition.

Finally, we thank our wives and children for their encouragement and patience while we completed this book.

References

Moore, J.W. and Stuart, A.E. *Neurons in Action: Computer Simulations with NeuroLab*. Sunderland, MA: Sinauer Associates, 2004.

The NEURON Book

Table of contents

Note: page numbers in each chapter start from 1, and correspond to double-spaced format.

Chapter 2. Principles of neural modeling

Chapter 3. Expressing conceptual models in mathematical terms

Chapter 4. Essentials of numerical methods for neural modeling

Chapter 5. Representing neurons with a digital computer

Chapter 6. How to build and use models of individual cells

Chapter 7. How to control simulations

Chapter 8. How to initialize simulations

Chapter 9. How to expand NEURON's library of mechanisms

Chapter 10. Synaptic transmission and artificial spiking cells

Chapter 11. Modeling networks

Chapter 12. hoc, NEURON's interpreter

Chapter 14. How to modify NEURON itself

Appendix A1. Mathematical analysis of IntFire4

Appendix A2. NEURON's built-in editor

Chapter 1

A tour of the NEURON simulation environment

Modeling and understanding

Modeling can have many uses, but its principal benefit is to improve understanding. The chief question that it addresses is whether what is known about a system can account for the behavior of the system. An indispensable step in modeling is to postulate a *conceptual model* that expresses what we know, or think we know, about a system, while omitting unnecessary details. This requires considerable judgment and is always vulnerable to hindsight and revision, but it is important to keep things as simple as possible. The choice of what to include and what to leave out depends strongly on the hypothesis that we are studying. The issue of how to make such decisions is outside the primary focus of this book, although from time to time we may return to it briefly.

The task of building a *computational model* should only begin after a conceptual model has been proposed. In building a computational model we struggle to establish a match between the conceptual model and its computational representation, always asking the question: would the conceptual model behave like the simulation? If not, where are the errors? If so, how can we use NEURON to help understand why the conceptual model implies that behavior?

Introducing NEURON

NEURON is a simulation environment for models of individual neurons and networks of neurons that are closely linked to experimental data. NEURON provides numerically sound, computationally efficient tools for conveniently constructing, exercising, and managing models, so that special expertise in numerical methods or programming is not required for its productive use. Increasing numbers of experimentalists and theoreticians are incorporating it into their research strategies. As of this writing, more than 460 scientific publications have reported work done with NEURON on topics that range from the molecular biology of voltage-gated channels to the operation of networks containing thousands of neurons (see **Research reports that have used NEURON** at `http://www.neuron.yale.edu/neuron/bib/usednrn.html`).

In the following pages we introduce NEURON by going through the development of a simple model from start to finish. This will require us to consider each of these steps:

1. State the question that we are interested in
2. Formulate a conceptual model
3. Implement the model in NEURON
4. Instrument the model, i.e. attach signal sources and set up graphs
5. Set up controls for running simulations
6. Save the model with instrumentation and run controls
7. Run simulation experiments

8. Analyze results

Since our aim is to provide an overview, we have chosen a simple model that illustrates just one of NEURON's strengths: the convenient representation of the spread of electrical signals in a branched dendritic architecture. We could do this by writing instructions in NEURON's programming language `hoc`, but for this example we will employ some of the tools that are provided by its graphical user interface. Later chapters examine `hoc` and the graphical tools for constructing models and managing simulations in more detail, as well as many other features and applications of the NEURON simulation environment (e.g. complex biophysical mechanisms, neural networks, analysis of experimental data, model optimization, customization of the user interface).

1. State the question

The scientific issue that motivates the design and construction of this model is the question of how synaptic efficacy is affected by synaptic location and the anatomical and biophysical properties of the postsynaptic cell. This has been the subject of too many experimental and theoretical studies to reference here. Interested readers will find numerous relevant publications in NEURON's on-line bibliography (cited above), and may retrieve working code for several of these from ModelDB (`http://senselab.med.yale.edu/senselab/modeldb/`).

2. Formulate a conceptual model

Most neurons have many branches with irregularly varying diameters and lengths (Fig. 1.1 A), and their membranes are populated with a wide assortment of ionic channels that have different ionic specificities, kinetics, dependence on voltage and second messengers, and spatial distributions. Scattered over the surface of the cell may be hundreds or thousands of synapses, some with a direct effect on ionic conductances (which may also be voltage-dependent) while others act through second messengers. Synapses themselves are far from simple, often displaying stochastic and use-dependent phenomena that can be quite prominent, and frequently being subject to various pre- and postsynaptic modulatory effects. Given all this complexity, we might well ask if it is possible to understand anything without understanding everything. From the very onset we are forced to decide what to include and what to omit.

Suppose we are already familiar with the predictions of the basic ball and stick model (Rall 1977; Jack et al. 1983), and that experimental observations motivate us to ask questions such as: How do synaptic responses observed at the soma vary with synaptic location if dendrites of different diameters and lengths are attached to the soma? What happens if some parts of the cell have active currents, while others are passive? What if a neuromodulator or shift of the background level of synaptic input changes membrane conductance?

Then our conceptual model might be similar to the one shown in Fig. 1.1 B. This model includes a neuron with a soma that gives rise to an axon and two dendritic trunks, and a single excitatory synapse that may be located at any point on the cell.

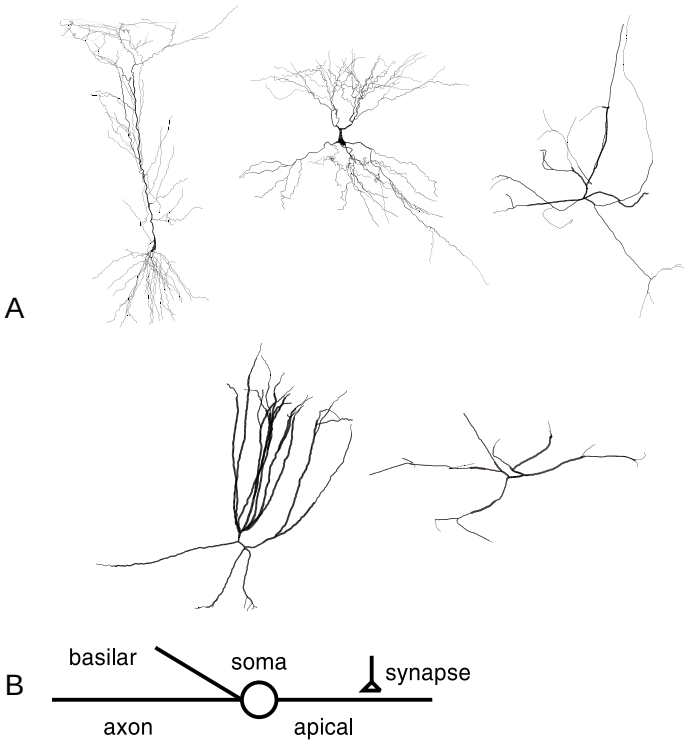


Figure 1.1. A. Clockwise from top left: Ca1 and Ca3 pyramidal neurons (from D.A. Turner); calbindin-, parvalbumin-, and calretinin-positive interneurons (from A.I. Gulyás). B. Our conceptual model neuron. The conductance change synapse can be located anywhere on the cell.

Although deliberately more complex than the prototypical ball and stick, the anatomical and biophysical properties of our model are much simpler than the biological original (Table 1.1). The axon and dendrites are simple cylinders, with uniform diameters and membrane properties along their lengths. The dendrites are passive, while the soma and axon have Hodgkin-Huxley sodium, potassium, and leak currents, and are capable of generating action potentials (Hodgkin and Huxley 1952). A single synaptic activation causes a localized transient conductance increase with a time course described by an alpha function

$$g_s(t) = \begin{cases} 0 \quad for \quad t < t_{act} \\ g_{max} \dfrac{(t - t_{act})}{\tau_s} \; e^{-\frac{(t - t_{act})}{\tau_s}} \quad for \quad t \geq t_{act} \end{cases} \qquad \text{Eq. 1.1}$$

where t_{act} is the time of synaptic activation, and g_s reaches a peak value of g_{max} at t = τ_s (see Table 1.2 for parameter values). This conductance increase mechanism is just slightly more complex than the ideal current sources used in many theoretical studies (Rall 1977; Jack et al. 1983), but it is still only a pale imitation of any real synapse (Bliss and Lømo 1973; Ito 1989; Castro-Alamancos and Connors 1997; Thomson and Deuchars 1997).

Table 1.1. 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 °C

Table 1.2. Synaptic mechanism parameters

$\mathbf{g_{max}}$	0.05 µS
τ_s	0.1 ms
$\mathbf{E_s}$	0 mV

3. Implement the model in NEURON

With a clear picture of our model in mind, we are ready to express it in the form of a computational model. Instead of writing instructions in NEURON's programming language hoc, for this example we will employ some of the tools that are provided by NEURON's graphical user interface.

We begin with the CellBuilder, a graphical tool for constructing and managing models of individual neurons. At this stage, we are not considering synapses, stimulating electrodes, or simulation controls. Instead we are focussing on creating a representation of the continuous properties of the cell. Even if we were not using the CellBuilder but instead were developing our model entirely with hoc code, it would probably be best for us to follow a similar approach, i.e. specify the biological attributes of the model cell separately from the specification of the instrumentation and control code that we will use to exercise the model. This is an example of modular programming, which is related to the "divide and conquer" strategy of breaking a large and complex problem into smaller, more tractable steps.

The CellBuilder makes it easier for us to create a model of a neuron by allowing us to specify its architecture and biophysical properties through a graphical interface. When we are satisfied with the specification, the CellBuilder will generate the corresponding hoc code for us. Once we have a model cell, we will be ready to use other graphical tools to attach a synapse to it and plot simulation results (see **4. Instrument the model** below).

The images in the following discussion were obtained under MSWindows; the appearance of NEURON under UNIX, Linux, and MacOS is quite similar.

Starting and stopping NEURON

No matter what a program does, the first thing you have to learn is how to start and stop it. To start NEURON under UNIX or Linux, just type `nrngui` on the command line and skip the remainder of this paragraph. Under MSWindows, double click on the nrngui icon on your desktop (Fig. 1.2 left); if you don't see one there, bring up the NEURON program group (i.e. use Start / Program Files / NEURON) and select the nrngui item (Fig. 1.2 right). If you are using MacOS, open the folder where you installed NEURON and double click on the nrngui icon.

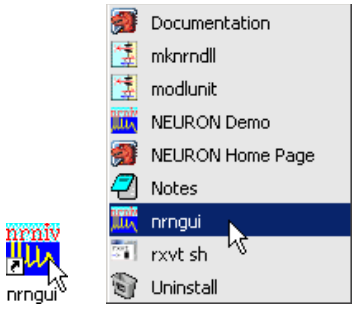


Figure 1.2. Under MSWindows, start NEURON by clicking on the nrngui icon on the desktop (left) or selecting the nrngui item in the NEURON program group (right).

You should now see the NEURON Main Menu (Fig. 1.3 top), which offers a set of menus for bringing up graphical tools for creating models and running simulations. If you are using UNIX or Linux, a "banner" that includes the version of NEURON you are running will be printed in the xterm where you typed `nrngui`, and the prompt will change to `oc>` to indicate that NEURON's `hoc` interpreter is running. Under MacOS and MSWindows, the "banner" and `oc>` prompt will appear in a new console window (Fig. 1.3 bottom).

There are three different ways to exit NEURON; you can use whichever is most convenient.

1. type `^D` (i.e. control D) at the `oc>` prompt
2. type `quit()` at the `oc>` prompt
3. click on File in the NEURON Main Menu, scroll down to Quit, and release the mouse button (Fig. 1.4)

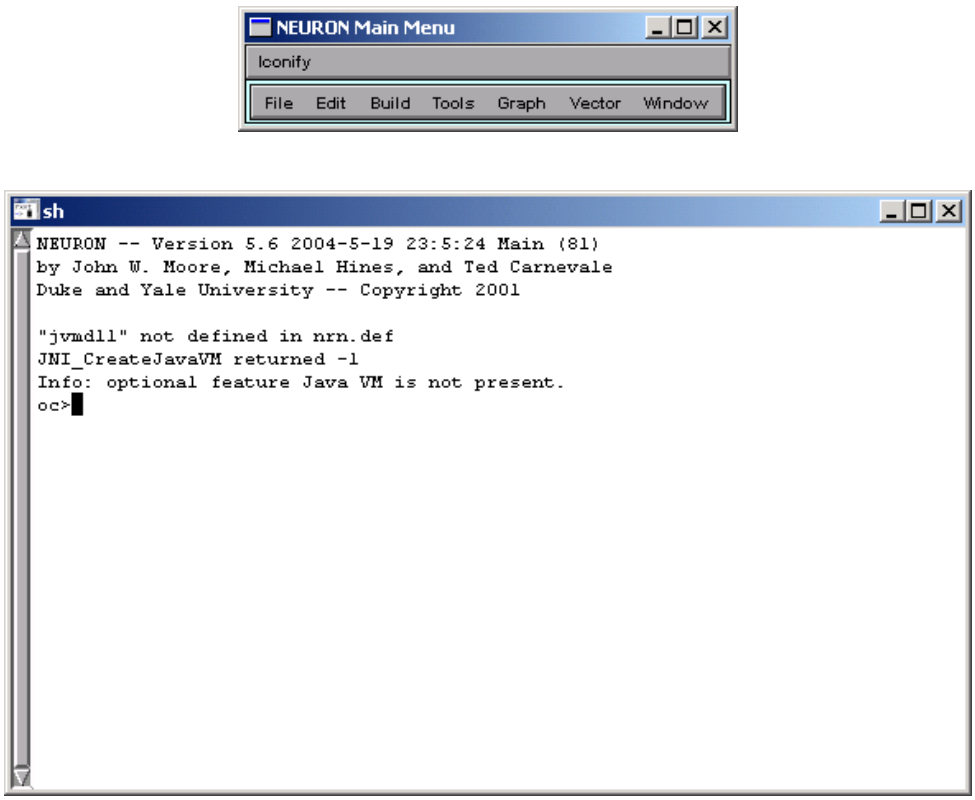


Figure 1.3. Top: The NEURON Main Menu toolbar. Bottom: NEURON's "banner" and `oc>` prompt in an MSWindows console window.

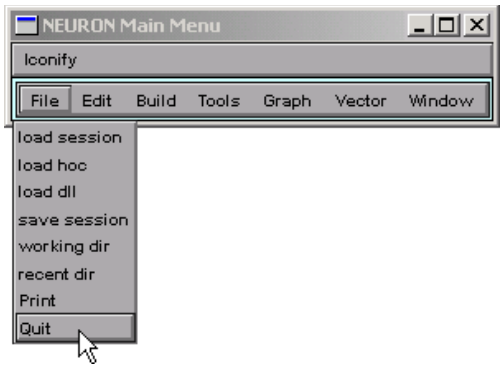


Figure 1.4. One way to exit NEURON is to click on File / Quit in the NEURON Main Menu toolbar.

Bringing up a CellBuilder

To get a CellBuilder just click on Build in the NEURON Main Menu, scroll down to the CellBuilder item, and release the mouse button (Fig. 1.5).

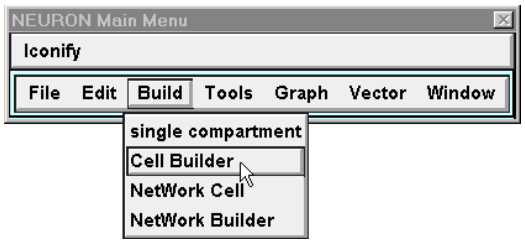


Figure 1.5. Using the NEURON Main Menu to bring up a CellBuilder.

Across the top of the CellBuilder is a row of radio buttons and a checkbox, which correspond to the sequence of steps involved in building a model cell (Fig. 1.6). Each radio button brings up a different page of the CellBuilder, and each page provides a view of the model plus a graphical interface for defining properties of the model. The first four pages (Topology, Subsets, Geometry, Biophysics) are used to create a complete specification of a model cell. On the Topology page, we will set up the branched

architecture of the model and give a name to each branch, without regard to diameter, length, or biophysical properties. We will deal with length and diameter on the Geometry page, and the Biophysics page is where we will define the properties of the membrane and cytoplasm of each of the branches.

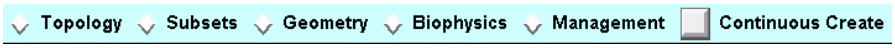


Figure 1.6. Top panel of the CellBuilder

The Subsets page deserves special comment. In almost every model that has more than one branch, two or more branches will have at least some biophysical attributes that are identical, and there are often significant anatomical similarities as well. Furthermore, we can almost always apply the d_lambda rule for compartmentalization throughout the entire cell (see below). We can take advantages of such regularities by assigning shared properties to several branches at once. The Subsets page is where we group branches into subsets, on the basis of shared features, with an eye to exploiting these commonalities on the Geometry and Biophysics pages. This allows us to create a model specification that is compact, efficient, and easily understood.

Enter the specifications of the model cell

- Topology

We start by using the Topology page to set up the branched architecture of the model. As Fig. 1.7 shows, when a new CellBuilder is created, it already contains a branch (or "section," as it is called in NEURON) that will serve as the root of the branched

architecture of the model (the root of a tree is the branch that has no parent). This root section is initially called "soma," but we can rename it if we desire (see below).

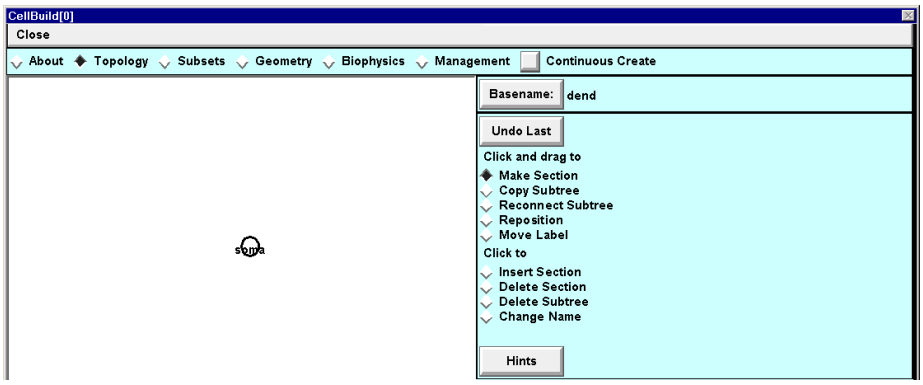


Figure 1.7. The Topology page. The left panel shows a simple diagram of the model, which is called a "shape plot." The right panel contains many functions for editing the branched architecture of a model cell.

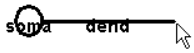
The Topology page offers many functions for creating and editing individual sections and subtrees. We can make the section that will become our apical dendrite by following the steps presented in Fig. 1.8. Repeating these actions a couple more times (and resorting to functions like Undo Last, Reposition, and Delete Section as needed to correct mistakes) gives us the basilar dendrite and axon.

Figure 1.8. Making a new section. Verify that the Make Section radio button is *on*, and then perform the following steps.

Place the cursor near one end of an existing section.

Click to start a new section. One end of the new section will automatically attach to the nearest end of an existing section; the other end is tethered to the cursor while the mouse button is held down.

Drag to the desired length and orientation.



Release the mouse button.

Our model cell should now look like Fig. 1.9. At this point some minor changes would improve its appearance: moving the labels away from the sections so they are easier to read (Fig. 1.10), and then renaming the apical and basilar dendrites and the axon (Figs. 1.11 and 12). The final result should resemble Fig. 1.13.

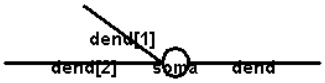
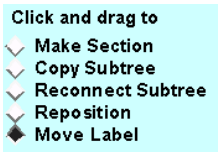
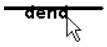


Figure 1.9. The model after all sections have been created.

Figure 1.10. To change the location of a label,

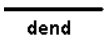


click on the Move Label radio button,



then click on the label,

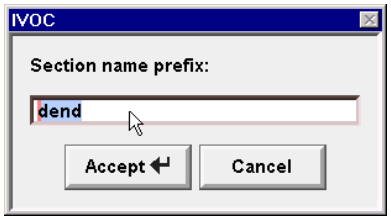
drag it to its new position,



and release the mouse button.

Figure 1.11. Preparing to change the name of a section. Each section we created was automatically given a name based on "dend." To change these names, we must first change the base name as shown here.

Click the Basename button.



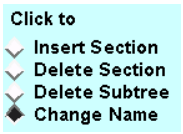
This pops up a Section name prefix window.

Click inside the text entry field of this new window, and type the desired name. It is important to keep the mouse cursor inside the text field while typing; otherwise keyboard entries may not have an effect.

After the new base name is complete, click on the Accept button. This closes the Section name prefix window, and the new base name will appear next to the Basename button.

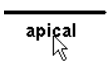
Figure 1.12. Changing the name of a section.

First make sure that the base name is what you want; if not, change the base name (see Fig. 1.11).



Click the Change Name radio button.

Place the mouse cursor over the section whose name is to be changed.



Click the mouse button to change the name of the section.

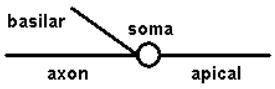


Figure 1.13. The shape plot of the model with labels positioned and named as desired.

- Subsets

As mentioned above, the Subsets page (Fig. 1.14) is for grouping sections that share common features. Well-chosen subsets can save a lot of effort later by helping us create very compact specifications of anatomical and biophysical properties.

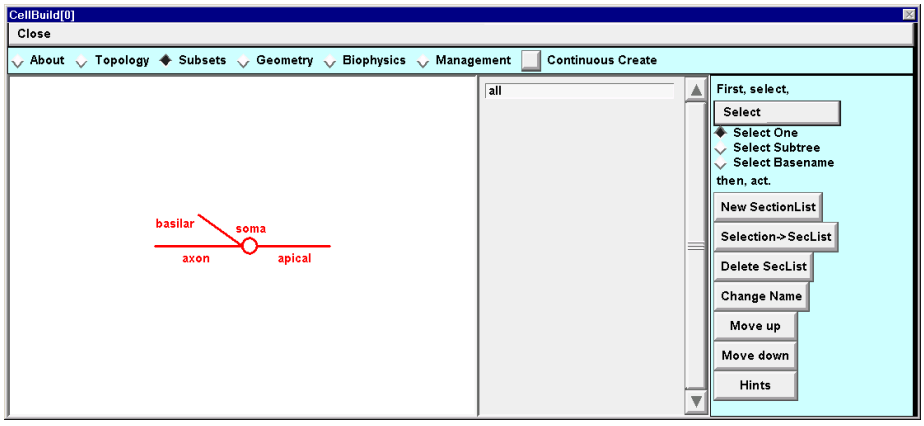
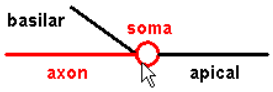


Figure 1.14. The Subsets page. The middle panel lists the names of all existing subsets. In the shape plot, the sections that belong to the currently selected subset are shown in red. When the Subsets page initially appears, it already has an all subset that contains every section in the model.

The properties of the sections in this particular example suggest that we create two subsets: one that contains the basilar and apical branches, which are passive, and another that contains the soma and axon, which have Hodgkin-Huxley spike currents. To make a subset called has_HH that contains the sections with HH currents, follow the steps in Fig. 1.15. Then make another subset called no_HH that contains the basilar and apical dendrites.

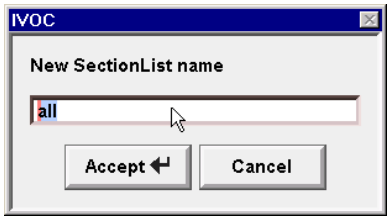
Figure 1.15. Making a new subset.



With the Select One radio button on (Fig. 1.14), click on the axon and soma sections while holding down the shift key. The selected sections will be indicated in red . . .

. . . and the list of subsets will change to show that all is not the same as the set {axon, soma}.

Next, click on the New SectionList button (a subset is a list of sections).



This pops up a window that asks you to enter a name for the new SectionList.

Click inside the text entry field of this new window and type the name of the new subset, then click on the Accept button.

The new subset name will appear in the middle panel of the CellBuilder.

- Geometry

In order to use the Geometry page (Fig. 1.16) to specify the anatomical dimensions of the sections and the spatial resolution of our model, we must first set up a strategy for assigning these properties. After we have built our (hopefully efficient) strategy, we will give them specific values.

The geometry strategy for our model is simple. Each section has different dimensions, so the length L and diameter diam of each section must be entered individually. However, for each section we will let NEURON decide how fine to make the spatial grid, based on a fraction of the length constant at 100 Hz (spatial accuracy and NEURON's tools for

adjusting the spatial grid are discussed in **Chapter 5**). Figure 1.17 shows how to set up this strategy.

Having set up the strategy, we are ready to assign the geometric parameters (see Figs. 1.18 and 19).

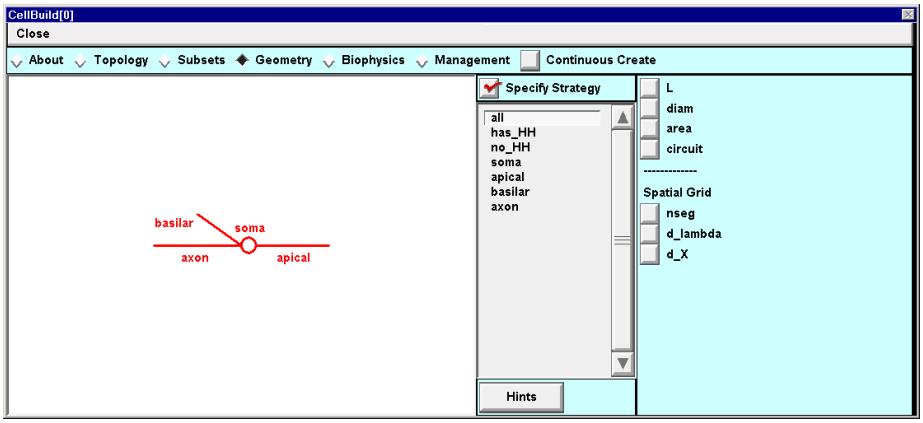
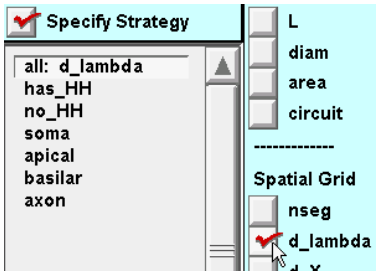


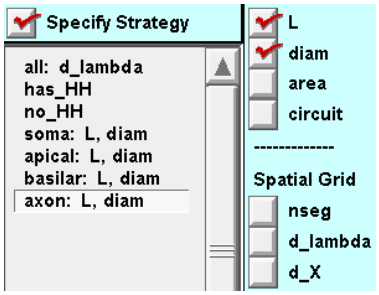
Figure 1.16. When the Geometry page in a new CellBuilder is first viewed, a red check mark should appear in the Specify Strategy checkbox. If not, clicking on the checkbox will toggle Specify Strategy *on*.

Figure 1.17. Specifying strategy for assignment of geometric parameters. First make sure that Specify Strategy contains a red check (see Fig. 1.16). Then proceed with the following steps.



For the all subset, toggle d_lambda on.

Select soma in the middle panel, and then toggle L and diam on.



Repeat for apical, basilar, and axon, and the result should resemble this figure.

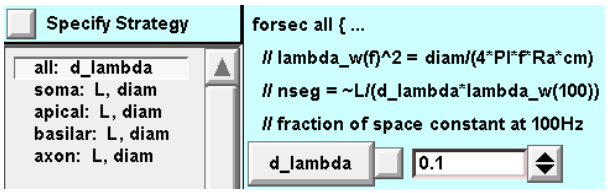
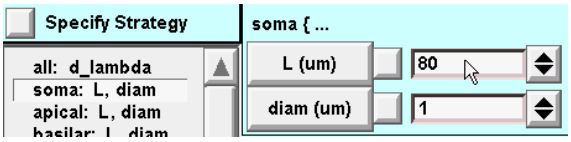


Figure 1.18. Assigning values to the geometric parameters. Toggling Specify Strategy *off* makes the middle panel show only the subsets and sections that we selected when setting up our strategy. Adjacent to each of these are the names of the parameters that are to be reviewed and perhaps changed. Here the subset all is selected; the right panel displays the current value of the parameter associated with it (d_lambda) and offers us the means to change this parameter if necessary. According to the d_lambda criterion for spatial resolution,

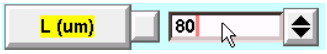
NEURON will automatically discretize the model, breaking each section into compartments small enough that none will be longer than d_lambda at 100 Hz. The default value of d_lambda is 0.1, i.e. 10% of the AC length constant. This is short enough for most purposes, so we do not need to change it.

Discretization is discussed in **Chapter 5**.

Figure 1.19. Assigning values to the geometric parameters *continued*.

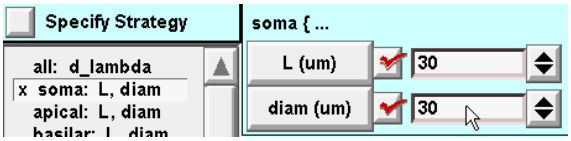


The length and diameter of each section must be changed from the default values.

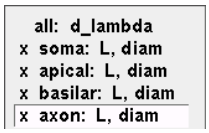


To set the length of the soma to 30 µm, first click inside the numeric field for L so that a red editing cursor appears.

Then use the backspace key to delete the old value, and finally type in the new value.



After doing the same for diam, the dimensions of soma should look like this. The checkboxes adjacent to the L and diam buttons indicate that these parameters have been changed from their default values. The x in the middle panel is another reminder that at least one of the parameters associated with soma has been changed.



After adjusting L and diam for the dendrites and the axon, the middle panel shows an x next to the name of each section.

- Biophysics

The Biophysics page (Fig. 1.20) is used to insert biophysical properties of membrane and cytoplasm (e.g. Ra, Cm, ion channels, buffers, pumps) into subsets and individual sections. As with the Geometry page, first we set up our strategy (Fig. 1.21), and then we review and adjust parameter values (Fig. 1.22). The CellBuilder will then contain a complete specification of our model.

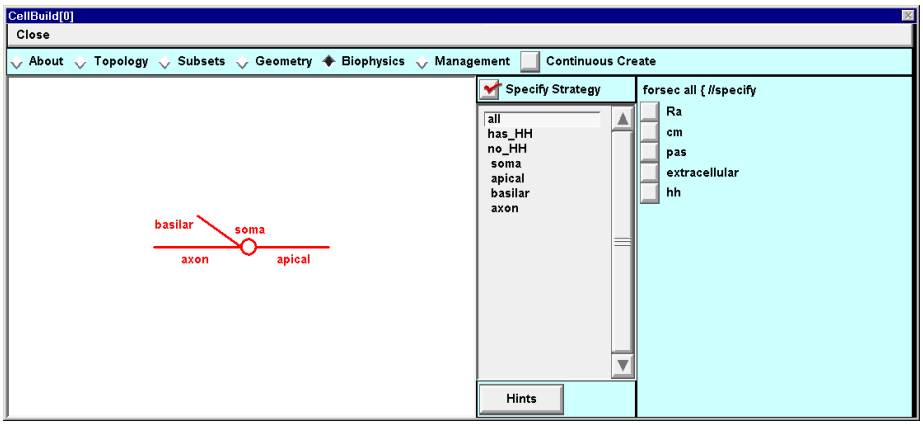
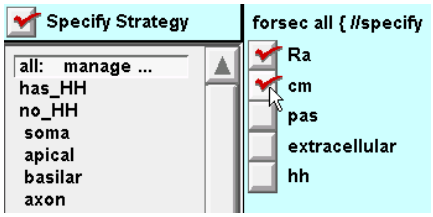
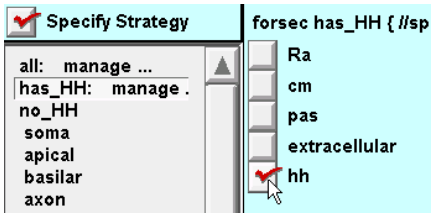


Figure 1.20. The Biophysics page, ready for specification of strategy. The right panel shows the mechanisms that are available to be inserted into our model. For this simple example, the number of mechanisms is deliberately small; adding new mechanisms is covered in **Chapter 9: How to expand NEURON's library of mechanisms**.

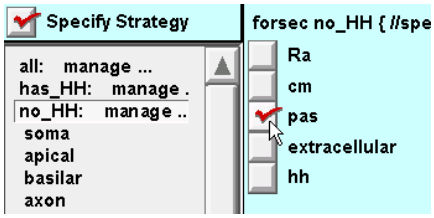
Figure 1.21. Specifying strategy for assignment of biophysical parameters. First make sure that Specify Strategy contains a red check, then proceed with the following steps.



For the all subset, toggle Ra (cytoplasmic resistivity) and cm (specific membrane capacitance) *on*.

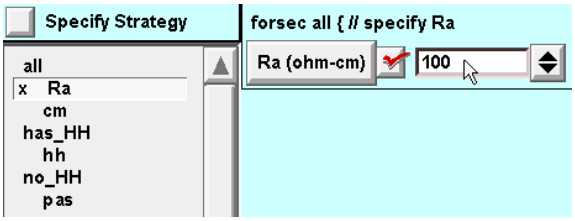


Select the has_HH subset in the middle panel, and then toggle HH *on*.

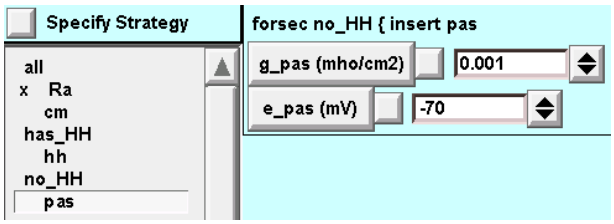


Finally select the no_HH subset and toggle pas *on*.

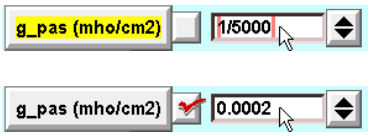
Figure 1.22. Assigning values to the biophysical parameters. Toggling Specify Strategy *off* shows a list of the names of the subsets that are part of the strategy. Beneath each subset are the names of the mechanisms that are associated with it. Clicking on a mechanism brings up a set of controls in the right panel for displaying and adjusting the parameters of the mechanism.



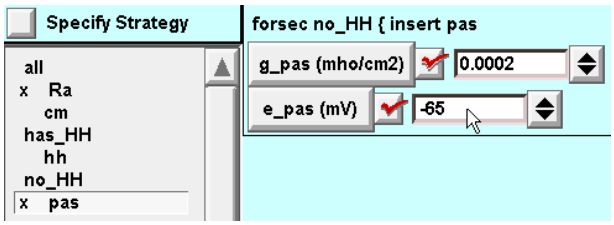
For the subset all, change the value of Ra from its default (80 Ω cm) to the desired value of 100 Ω cm.



The sections in the no_HH subset have a passive current whose parameters must be changed from their defaults (shown here).



The value of g_pas can be set by deleting the default and then typing 1/5000 (= 1/Rm).



The final values of g_pas and e_pas. Not shown: cm (all subset) and the parameters of the hh mechanism (has_HH subset), which have the desired values by default and do not need to be changed, although it is good practice to review them.

Save the model cell

After investing time and effort to set up our model, we would be wise to take just a moment to save it. The CellBuilder, like NEURON's other graphical windows, can be saved to disk as a "session file" for future re-use, as shown in Figures 1.23 and 1.24. For more information about saving and retrieving session files, including how to use the Print

& File Window Manager GUI tool to select and save specific windows, see **Using Session Files for Saving and Retrieving Windows** at `http://www.neuron.yale.edu/neuron/docs/saveses/saveses.html`

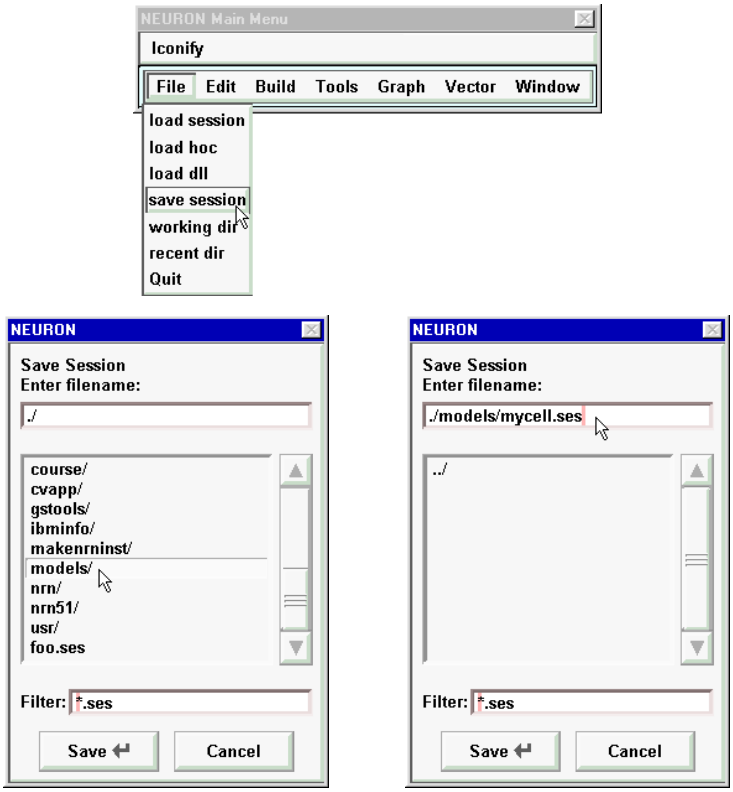


Figure 1.23. Top: To save all of NEURON's graphical windows into a session file, first click on File in the NEURON Main Menu and scroll down to save session. Bottom left: This brings up a directory browser that can be used to navigate to the directory where the session file will be saved. Bottom right: Click in the edit field at the top of the directory browser and type the name to use for the session file, then click on the Save button.

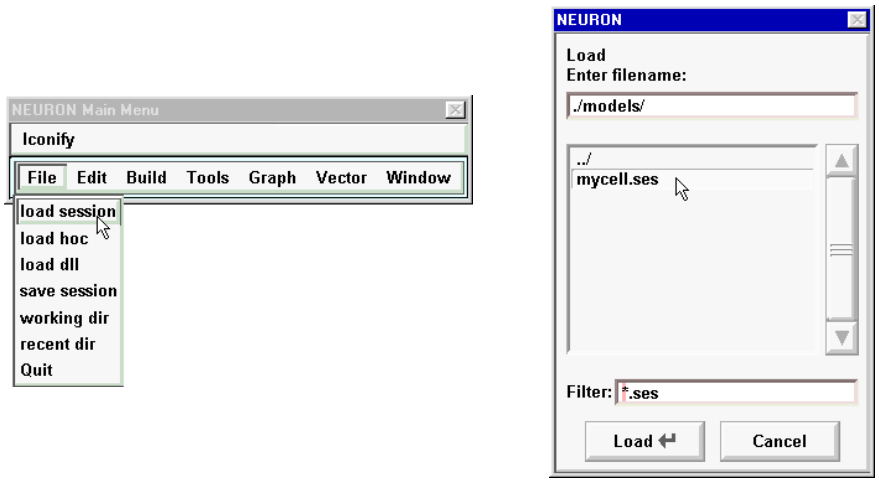


Figure 1.24. Left: To recreate the graphical windows that were saved to a session file, first click on File in the NEURON Main Menu and scroll down to load session. Right: Use the directory browser that appears to navigate to the directory where the session file was saved. Then double click on the session file that you want to retrieve.

Execute the model specification

Now that the CellBuilder contains a complete specification of the model cell, we could use the Export button on the Management page (see **Chapter 6**) to write out a hoc file that, when executed by NEURON, would create the model. However, for this example we will just turn Continuous Create *on* (Fig. 1.25). This makes the CellBuilder send its output directly to NEURON's interpreter without bothering to write a hoc file. The model cell whose specifications are contained in the CellBuilder is now available to be used in simulations.

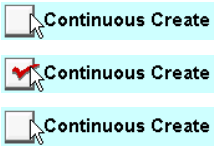


Figure 1.25. Continuous Create is initially *off*, but clicking on the adjacent button toggles it *on* and *off*.

If we make any changes to the model while Continuous Create is *on*, the CellBuilder will automatically send new code to the interpreter. This can be very convenient during model development, since it allows us to quickly examine the effects of any change. Automatic updates might bog things down if we were dealing with a large model on a slow machine. In such a case, we could just turn Continuous Create *off*, make whatever changes were necessary, and then cycle it *on* and *off* again.

4. Instrument the model

Signal sources

In the NEURON simulation environment, a synapse or electrode for passing current (current clamp or voltage clamp) is represented by a point source of current which is associated with a localized conductance. These signal sources are called "point processes" to distinguish them from properties that are distributed over the cell surface (e.g. membrane capacitance, active and passive ionic conductances) or throughout the cytoplasm (e.g. buffers), which are called "distributed mechanisms" or "density mechanisms."

We have already seen how to use one of NEURON's graphical tools for dealing with distributed mechanisms (the CellBuilder). To attach a synapse to our model cell, we turn to one of NEURON's tools for dealing with point processes: the PointProcessManager (Fig. 1.26). Using a PointProcessManager we can specify the type and parameters of the point process (Fig. 1.27) and where it is attached to the cell.

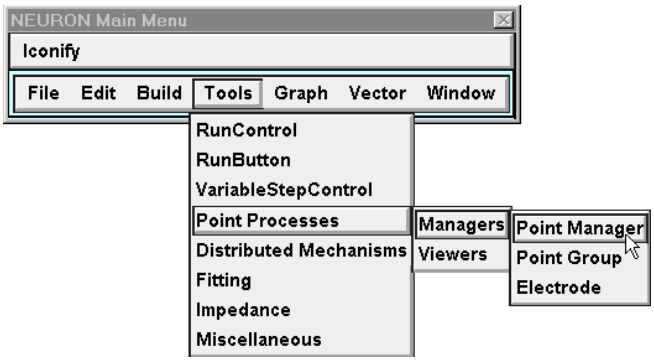
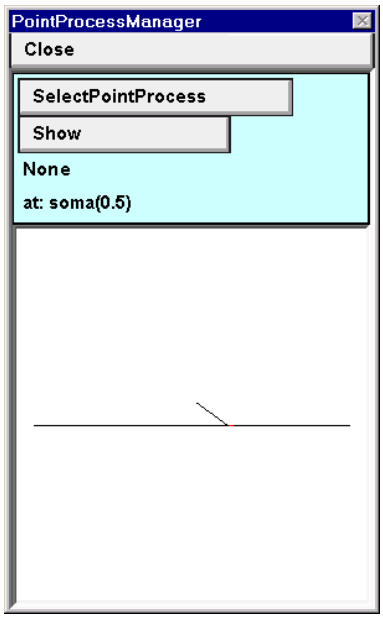
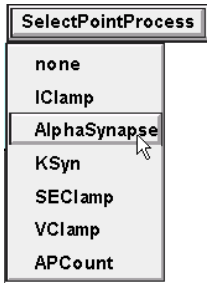


Figure 1.26. Bringing up a PointProcessManager in order to attach a synapse to our model cell. In the NEURON Main Menu, click on Tools / Point Processes / Managers / Point Manager, then proceed as shown in Fig. 1.27.

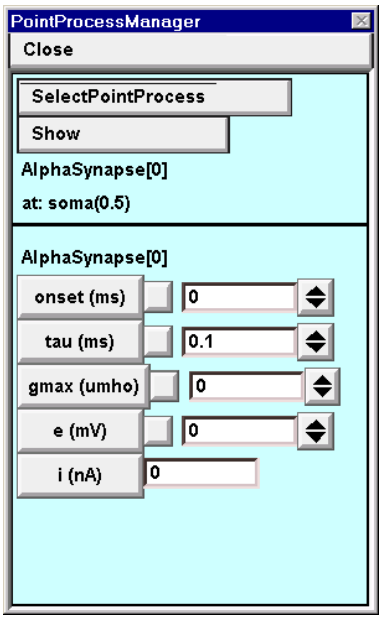
Figure 1.27. Configuring a new PointProcessManager to emulate a synapse.



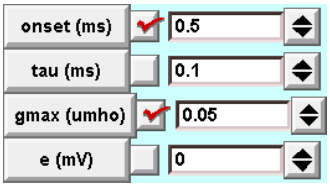
A. Note the labels in the top panel. None means that a signal source has not yet been created. The bottom panel shows a stick figure of our model cell.



B. SelectPointProcess / AlphaSynapse creates a point process that emulates a synapse with a conductance change governed by Eq. 1.1, and shows us a panel for adjusting its parameters.



C. The top panel of the PointProcessManager indicates what kind of point process has been specified, and where it is located (in this case, at the midpoint of the soma). The bottom panel shows the parameters of an AlphaSynapse: its start time onset and time constant tau (t_{act} and τ_s in Eq. 1.1), peak conductance gmax (g_{max} in Eq. 1.1), and reversal potential e (E_s in Table 1.2). The button marked i (nA) is just a label for the adjacent numeric field, which displays the instantaneous synaptic current.



D. For this example change onset to 0.5 ms and gmax to 0.05 µS; leave tau and e unchanged.

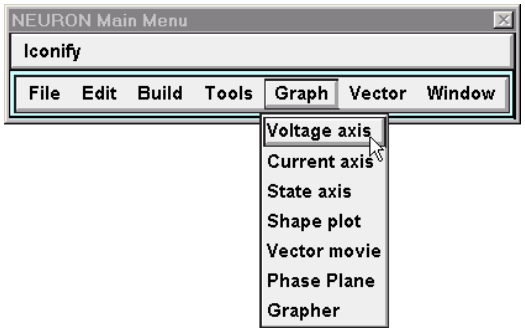
Signal monitors

Since one motivation for the model is to examine how synaptic responses observed at the soma vary with synaptic location, we want a graph that shows the time course of somatic membrane potential. In the laboratory this would ordinarily require attaching an electrode to the soma, so in a NEURON simulation it might seem to require a point process. However, the computer automatically evaluates somatic V_m in the course of a simulation. In other words, graphing V_m doesn't really change the system, unlike attaching a signal source, which adds new equations to the system. This means that a

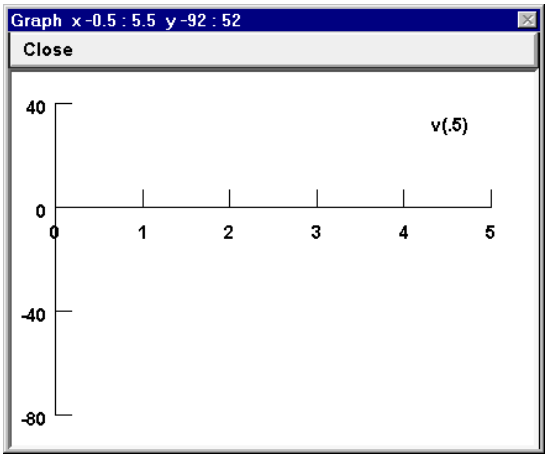
point process is not needed; instead, we just bring up a graph that includes somatic V_m in the list of variables that it plots (see Fig. 1.28).

We could monitor V_m at other locations by adding more variables to this graph, and bring up additional graphs if this one became too crowded. However, it can be more informative and convenient to create a "space plot" (Fig. 1.29), which shows V_m as a function of position along one or more branches of a cell. This graph will change throughout the simulation run, displaying the evolution of V_m as a function of space and time.

Figure 1.28. Creating a graph to display somatic membrane potential as a function of time.

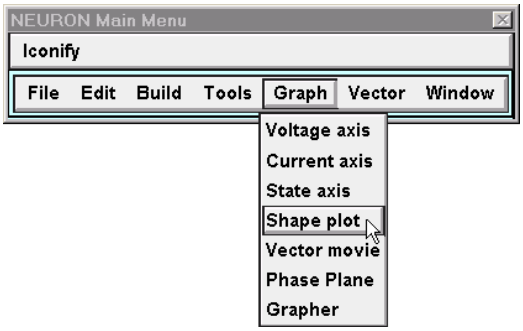


A. Click on Graph / Voltage axis in the NEURON Main Menu.

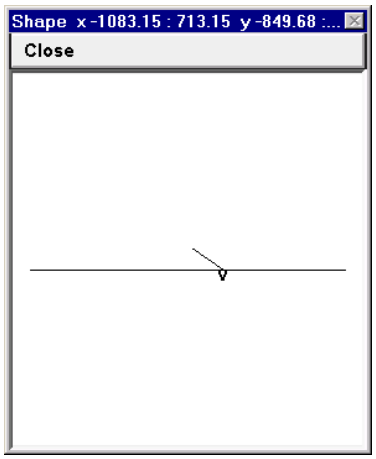


B. In the graph that appears, the horizontal axis is in milliseconds and the vertical axis is in millivolts. The label v(.5) signifies that this graph will show V_m at the middle of the default section. With the CellBuilder, this is always the root section, which in this example is the soma (the concepts of "root section" and "default section" are discussed in **Chapter 5**.

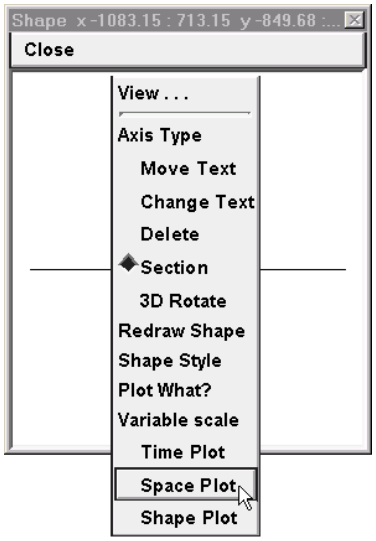
Figure 1.29. Setting up a space plot.



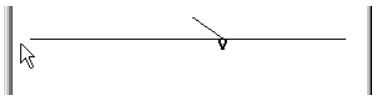
A. The first step is to create a Shape plot by clicking on Graph / Shape plot in the NEURON Main Menu.



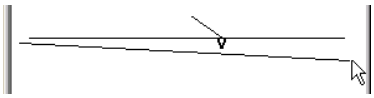
B. This brings up a Shape plot window, which is used to create the space plot.



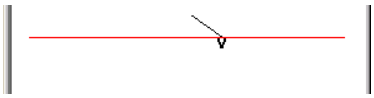
C. Right click in the Shape plot window to bring up its primary menu. While still pressing the mouse button, scroll down the menu to the Space Plot item, then release the button.



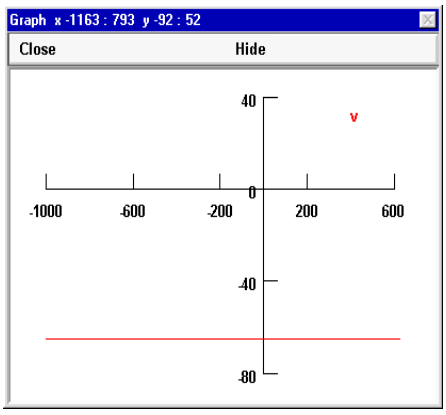
D. Place the cursor just to the left of the distal end of the axon and press the left mouse button.



E. While still holding the button down, drag the cursor across the window to the right, finally releasing the button when the cursor has passed the distal end of the apical dendrite.



F. The branches along the selected path (axon, soma, and apical dendrite) are now shown in red, and a new graph window appears (see G). If you like, you may now click on the Close button at the upper left corner of the shape plot window to conserve screen space



G. The x axis of the Space Plot window shows the distance from the 0 end of the default section, which in this example is the left end of the soma.

5. Set up controls for running the simulation

At this point we have a model cell with a synapse attached to the soma, and a graphical display of somatic V_m. All that is missing is a means to start and control the subsequent course of a simulation run. This is provided by the RunControl window (Fig. 1.30), which allows us to specify many more options than we will use in this example.

6. Save model with instrumentation and run control

After rearranging the RunControl, PointProcessManager, and graph window, our customized user interface for running simulations and observing simulation results should look something like Fig. 1.31. For the sake of safety and possible future convenience, it is a good idea to use NEURON Main Menu / File / Save Session to save this custom GUI to a session file.

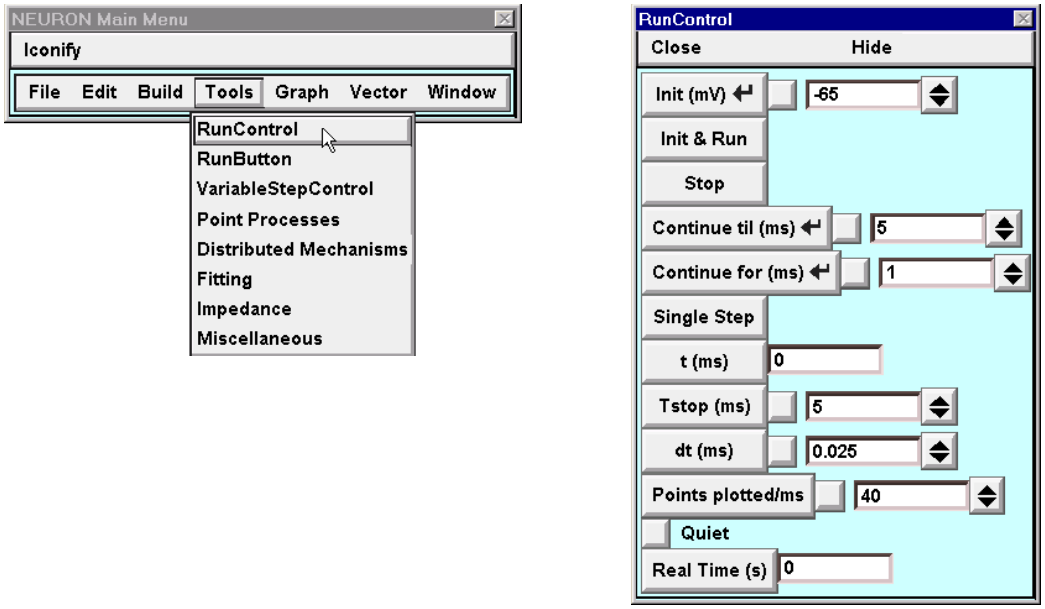


Figure 1.30. Left: To bring up a window with controls for running simulations, click on the RunControl button in NEURON Main Menu / Tools. Right: The RunControl window provides many options for controlling the overall time course of a simulation run. For this example, only three of these controls are relevant.

1. Init (mV) sets time t to 0, assigns the displayed starting value (-65 mV) to V_m throughout the model cell, and sets the ionic conductances to their their steady state values at this potential.

2. Init & Run performs the same initialization as Init (mV), and then starts a simulation run.

3. Points plotted/ms determines how often the graphical displays are updated during a simulation.

Three other items in this panel are of obvious interest, although we will not do anything with them in this example. The first is dt, which sets the size of the time intervals at which the equations that describe the model are solved. The second is Tstop, which specifies the duration of a simulation run. Finally, the button marked t doesn't actually do anything but is just a label for the adjacent numeric field, which displays the elapsed simulation time. Additional features of the RunControl window are discussed in **Chapter 7: How to control simulations**.

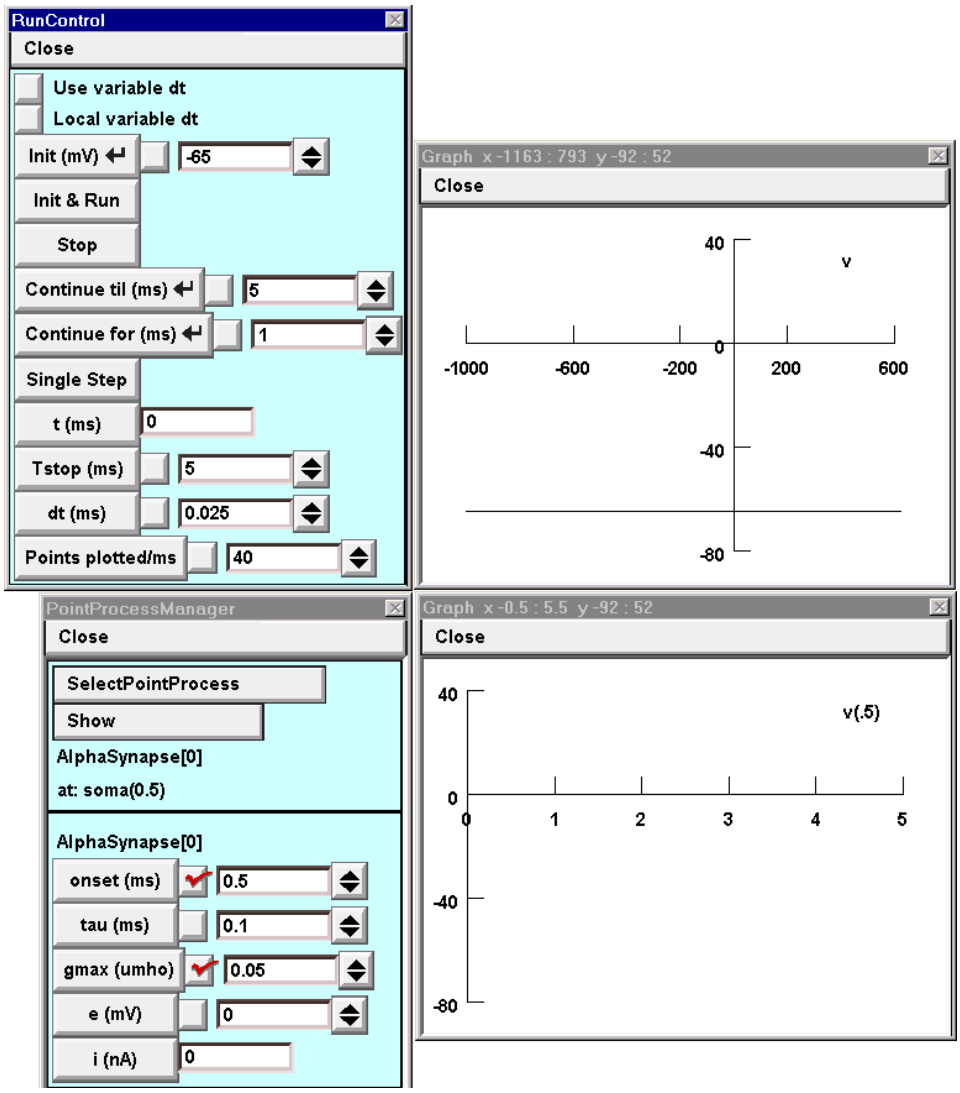


Figure 1.31. The windows we will use to run simulations and observe simulation results. Other windows that are present on the screen but not shown in this figure are the NEURON Main Menu and the CellBuilder.

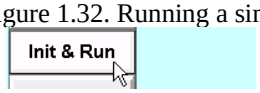
7. Run the simulation experiment

We are now ready to use our "virtual experimental rig" to exercise the model. When we run a simulation with the synapse located at the soma (Fig. 1.32 and 33), a spike is

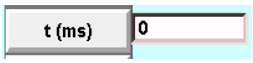
triggered. However, if we move the synapse even a small distance away from the soma along the apical dendrite (Fig. 1.34) and run a new simulation, the epsp is too small to evoke a spike (Fig. 1.35).

The utility of the space plot as a tool for understanding the temporal evolution of V_m throughout the cell can be enhanced by using it like a storage oscilloscope, as shown in Fig. 1.36. This allows us to compare the distribution of V_m at successive intervals during a run. It might be helpful to do something similar with the plot of somatic V_m vs. t if we wanted to compare responses to synaptic inputs with different parameters or locations.

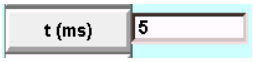
Figure 1.32. Running a simulation.



A. Press Init & Run in the RunControl window to launch a simulation.



B. This makes time t advance from 0 . . .



. . . to 5 ms in 0.025 ms increments. The response of the model is shown in Fig. 1.33.

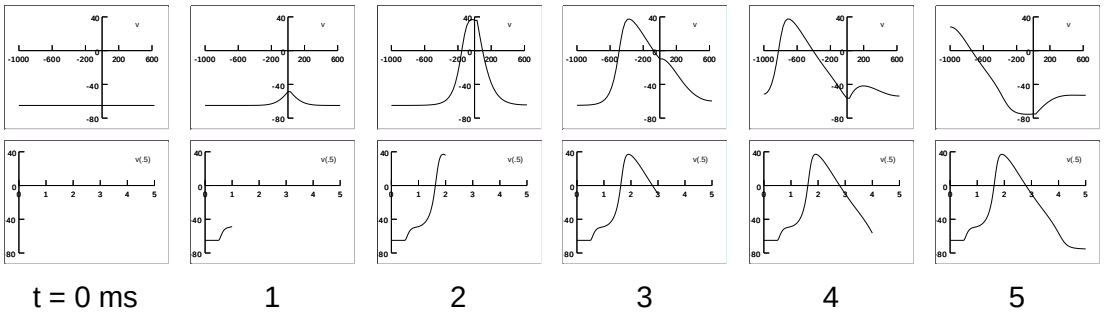
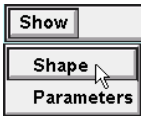
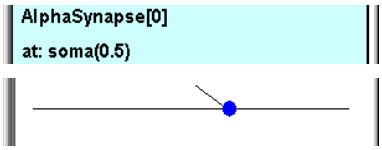


Figure 1.33. Snapshots of the space plot (top) and the graph of V_m vs. t at the soma (bottom) taken at 1 ms intervals. Synaptic input at the soma triggers a spike that propagates actively along the axon and spreads with passive decrement into the apical dendrite.

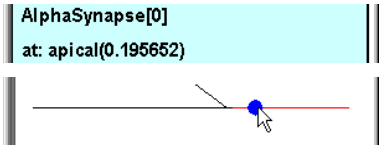
Figure 1.34. Changing synaptic location.



A. In the top panel of the PointProcessManager, click on Show and scroll down to Shape.



B. The top panel remains unchanged, but the bottom panel of the PointProcessManager now displays a shape plot of the cell, with a blue dot that indicates the location of the synapse.



C. Clicking on a different point in the shape plot moves the synapse to a new location. This change is reflected in the top and bottom panels of the PointProcessManager.

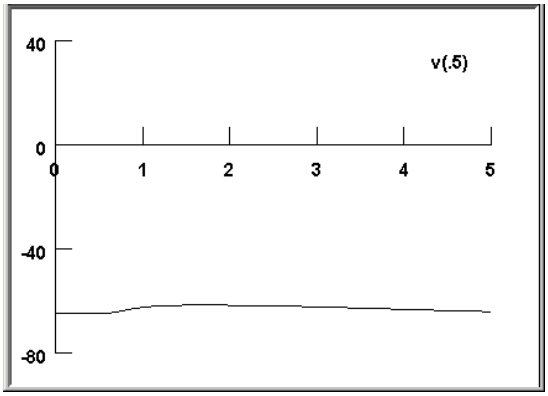


Figure 1.35. Pressing Init & Run starts a new simulation. Even though the synapse is still quite close to the soma, the somatic depolarization is now too small to trigger a spike (space plot not shown).

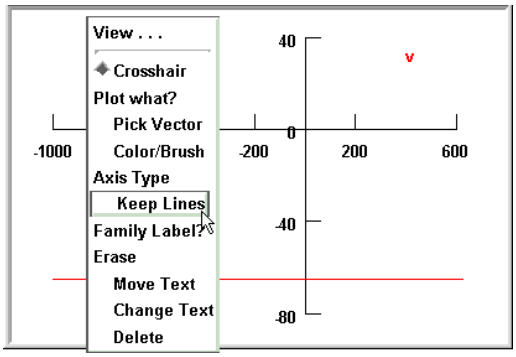
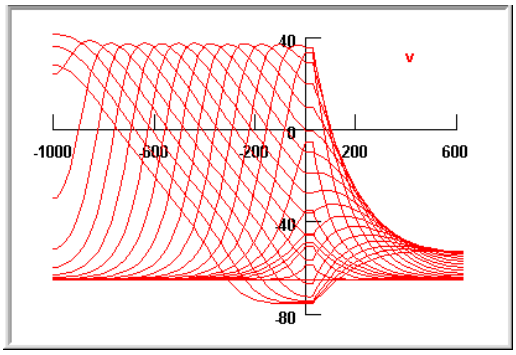


Figure 1.36. A. Activating "keep lines" can help visualize the evolution of V_m more clearly. Right click in the space plot window to bring up its primary menu, then scroll down to Keep Lines and release the mouse button. The next time the primary graph menu is examined, a red check mark will appear next to this item as an indication that keep lines has been toggled *on* (Fig. 1.37 A).

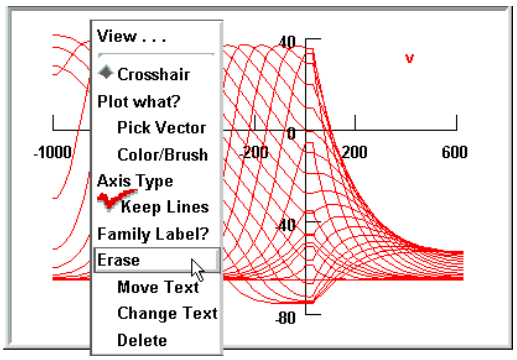
B. To keep the graph from filling up with an opaque tangle of lines, we should make sure the stored traces will be sufficiently different from each other. Plotting only 5 traces per millisecond will do the trick for this example (leave dt = 0.025 ms).



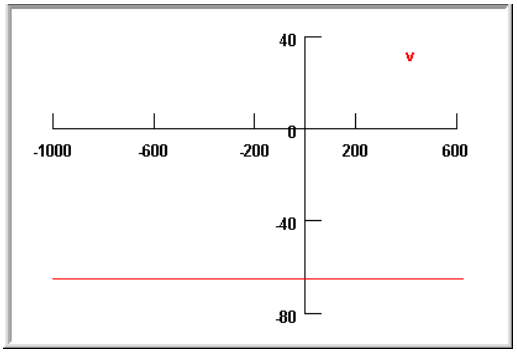
C. Now pressing Init & Run generates a set of traces that facilitate a close examination of the process of excitation and impulse conduction over the model.

For this example the synapse was at the middle of the soma (soma(0.5)). Before running another simulation with a different synaptic location, it would be a good idea to erase these traces (see Fig. 1.37).

Figure 1.37. How to erase traces.



A. Bring up the primary graph menu and scroll down to Erase.



B. The traces will disappear when the mouse button is released. Since keep lines is active, running another simulation will generate a new set of traces.

8. Analyze results

In this section we turn from our specific example to a consideration of the analysis of results. Models are generally constructed either for didactic purposes or as a means for testing a hypothesis. The design and analysis of any model are both strongly dependent on this original motivation, which determines what features are included in the model, what variables are regarded as important enough to measure, and how these measurements are to be interpreted.

While computational models are arguably simpler than any (interesting) experimental preparation, analysis of simulation results presents its own special problems. In the first

place, attempting to use a digital computer to mimic the behavior of a biological system introduces many potential complexities and artifacts. Some arise from the fact that neurons are continuous in space and time, but a digital computer can only generate approximate solutions for a finite number of discrete locations at particular instants. Even so, under the right conditions the approximation can be very good indeed. Furthermore, a well-designed simulation environment can reduce the difficulty of achieving good results.

Other difficulties can arise if there is a mismatch between the expectations of the user and the level of detail that has been included in a model. For example, the most widely used computational model of a conductance change synapse is designed to do the same thing each and every time it is "activated," yet most real synapses display many kinds of use-dependent plasticity, and many also have a high degree of stochastic variability. And even the venerable Hodgkin-Huxley model (Hodgkin and Huxley 1952), which is probably *the* classical success story of computational neuroscience, does not replicate all features of the action potential in the squid giant axon, because it does not completely capture the dynamics of the currents that generate the spike (Moore and Cox 1976; Fohlmeister et al. 1980; Clay and Shlesinger 1982). Such discrepancies are potentially a problem only if a user who is unaware of their existence attempts to apply a model outside of its original context.

The first analysis that is required of all computational modeling is actually the verification that what has been implemented in the computer is a faithful representation of the conceptual model. At the least, this involves checking to be sure that the intended anatomical and biophysical features have been included, that parameters have been assigned the desired values, and that appropriate initialization and integration methods

have been chosen. It may also be necessary to test the model's biophysical mechanisms to ensure that they show the correct dependence on time, membrane potential, ionic concentrations, and modulators. This means understanding the internals of the computational model, which in turn demands a nontrivial grasp of the programming language in which it is expressed. A custom graphical interface that includes well-designed menus and "variable browsers" can make it easier to answer the frequently occurring question "what are the names of things?" Even so, every simulation environment is predicated on a set of underlying concepts and assumptions, and questions inevitably arise that can only be answered on the basis of knowledge of these core concepts and assumptions.

Verification should also involve the qualitative, if not quantitative, comparison of simulation results with basic predictions obtained from experimental observations on biological preparations or generated with prior models. Discrepancies between prediction and simulation are usually caused by trivial errors in model implementation, but sometimes the fault lies in the prediction. Detecting these more interesting outcomes requires practical facility with the simulation environment, so that the level of effort does not obscure one's thinking about the problem.

Agreement between prediction and simulation is reassuring and suggests that the model itself may be useful for generating experimentally-testable predictions. Thus the effort shifts from verifying the model to characterizing its behavior in ways that extend beyond the initial test runs. Both verification and characterization of neural models may entail determining not only membrane potential but also rate functions, levels of modulators, and ionic conductances, currents, and concentrations at one or more locations

in one or more cells. Thus it is necessary to be able to gather and manage measurements, both within a single simulation run and across a family of runs in which one or more independent variables are assigned different values.

Similar concerns arise in connection with optimization, in which one or more parameters are adjusted until the behavior of the model satisfies certain criteria. Optimization also opens a host of new questions whose answers depend in part on the user's judgment, and in part on the resources provided by the simulation environment. Which parameters should remain fixed and which should be adjustable? What constitutes a "run" of the model? What are the criteria for goodness of fit? What constraints, if any, should be imposed on adjustable parameters, and what rules should govern how they are adjusted?

In summary, analysis of results can be the most difficult aspect of any experiment, whether it was performed on living neurons or on a computer model, yet it can also be the most rewarding. The issues raised here are critical to the informed use of any simulation environment, and in the following chapters we will reexamine them in the course of learning how to develop and exercise models with NEURON.

References

Bliss, T.V.P. and Lømo, T. Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetised rabbit following stimulation of the perforant path. *J. Physiol.* 232:331-356, 1973.

Castro-Alamancos, M.A. and Connors, B.W. Distinct forms of short-term plasticity at excitatory synapses of hippocampus and neocortex. *Proc. Nat. Acad. Sci.* 94:4161-4166, 1997.

Clay, J.R. and Shlesinger, M.F. Delayed kinetics of squid axon potassium channels do not always superpose after time translation. *Biophys. J.* 37:677-680, 1982.

Fohlmeister, J.F., Adelman, W.J.J., and Poppele, R.E. Excitation properties of the squid axon membrane and model systems with current stimulation. Statistical evaluation and comparison. *Biophys. J.* 30:79-97, 1980.

Hodgkin, A.L. and Huxley, A.F. A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol.* 117:500-544, 1952.

Ito, M. Long-term depression. *Ann. Rev. Neurosci.* 12:85-102, 1989.

Jack, J.J.B., Noble, D., and Tsien, R.W. *Electric Current Flow in Excitable Cells*. London: Oxford University Press, 1983.

Moore, J.W. and Cox, E.B. A kinetic model for the sodium conductance system in squid axon. *Biophys. J.* 16:171-192, 1976.

Rall, W. Core conductor theory and cable properties of neurons. In: *Handbook of Physiology, vol. 1, part 1: The Nervous System*, edited by E.R. Kandel. Bethesda, MD: American Physiological Society, 1977, p. 39-98.

Thomson, A.M. and Deuchars, J. Synaptic interactions in neocortical local circuits: dual intracellular recordings *in vitro*. *Cerebral Cortex* 7:510-522, 1997.

Chapter 1 Index

Chapter 2

The modeling perspective

This and the following chapter deal with concepts that are not NEURON-specific but instead pertain equally well to *any* tools used for neural modeling.

Why model?

In order to achieve the ultimate goal of understanding how nervous systems work, it will be necessary to know many different kinds of information:

- the anatomy of individual neurons and classes of cells, pathways, nuclei, and higher levels of organization
- the pharmacology of ion channels, transmitters, modulators, and receptors
- the biochemistry and molecular biology of enzymes, growth factors, and genes that participate in brain development and maintenance, perception and behavior, learning and forgetting, health and disease

But while this knowledge will be necessary for an understanding of brain function, it isn't sufficient. This is because the moment-to-moment processing of information in the brain is carried out by the spread and interaction of electrical and chemical signals that are distributed in space and time. These signals are generated and regulated by mechanisms

that are kinetically complex, highly nonlinear, and arranged in intricate anatomical structures. Hypotheses about these signals and mechanisms, and how nervous system function emerges from their operation, cannot be evaluated by intuition alone, but require empirically-based modeling. From this perspective, modeling is fundamentally a means for enhancing insight, and a simulation environment is useful to the extent that it maximizes the ratio of insight obtained to effort invested.

From physical system to computational model

Just what is involved in creating a computational model of a physical system?

Conceptual model: a simplified representation of a physical system

The first step is to formulate a *conceptual model* that attempts to capture just the essential features that underlie a particular function or property of the physical system. If the aim of modeling is to provide insight, then formulating the conceptual model necessarily involves simplification and abstraction (Fig. 2.1 left). When a physical system is already simple enough to understand, there is no point in further simplification because we won't learn anything new. If instead the system is complex, a conceptual model that omits excess detail can foster understanding.

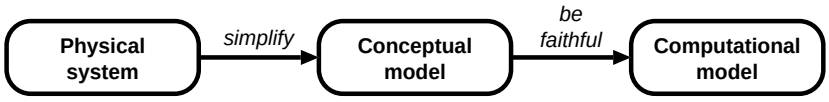


Figure 2.1. Creating a computational model of a physical system involves two steps. The first step deliberately omits real-world complexities to produce a conceptual model. In the second step, this conceptual model must be faithfully translated into a computational model, without any further subtractions or additions.

But some models contain essential irreducible complexities, and even conceptual models that are superficially simple can resist intuition. To evaluate such a model it is often necessary to devise a hypothesis or test in which the behavior of the model is compared against a prediction. *Computational models* are useful for performing such tests. The conceptual model, and the hypothesis behind it, determine what is included in the computational model and what is left out.

When we formalize our description of a biological system, the first language we use is mathematics. The conceptual model is usually expressed in mathematical form, although there are occasions when it is more convenient to express the concept in the form of a computer algorithm. **Chapter 3** is concerned with mathematical representations of chemical and electrical phenomena relevant to signaling in neurons.

Computational model: an accurate representation of a conceptual model

A computational model is a working embodiment of a conceptual model through the medium of computer simulation. It can assist hypothesis testing by serving as a virtual

laboratory preparation in which the functional consequences of the hypothesis can be examined. Such tests can be valid only if the computational model is as faithful to the conceptual model as possible. This means that the computational model must be implemented in a way that does not impose additional simplifications or introduce new properties that were not consciously chosen by the user; otherwise how can the user tell whether simulation results truly reflect the properties of the conceptual model, and are not a byproduct of distortions produced by trying to implement the model with a computer? This ideal is impossible to meet, and the proper use of any simulator requires judgment by the user as to whether discrepancies between concept and concrete representation are benign or vicious.

A useful simulation environment enables experimental tests of hypotheses by facilitating the construction, use, and revision of computational models that are faithful to the original idea and its subsequent evolution. NEURON is designed to meet this goal, and one of the aims of this book is to show you how to tell whether the model you have in mind is matched by the NEURON simulation you create.

An example

Suppose we are interested in how the cell of Fig. 2.2 A responds to current injected at the soma. We could imagine an enormously complicated conceptual model that attempts to mimic all of the detail of the physical system. But if we're really interested in insight, we might start with a much simpler conceptual model, like the ball and stick shown in Fig. 2.2 B. Most of the anatomical complexity of the physical system lies in the dendritic

tree, but our conceptual model approximates the entire dendritic tree by a very simple abstraction: a cylindrical cable.

So going from the physical system to the model involved simplification and abstraction. What about going from the conceptual model to a computational model?

The statements in Fig. 2.2 C specify the topology of the computational model using `hoc`, NEURON's programming language. Note that everything in the conceptual model has a direct counterpart in the computational model, and vice versa: the transition between concept and computational model involves neither simplification nor additional complexity. All that remains is to assign physical dimensions and biophysical properties, and the computational model can be used to generate simulations that reflect the behavior of the conceptual model.

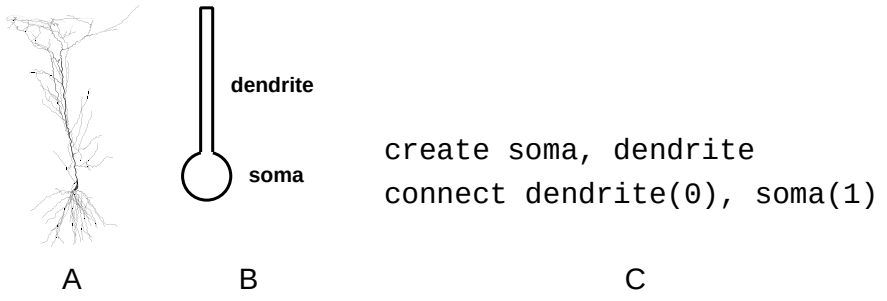


Figure 2.2. A. Detailed morphometric reconstruction of Ca1 pyramidal neuron (from D.A. Turner). B. "Ball and stick" conceptual model for studying charging properties of a neuron as seen from the soma. C. The computational implementation of the conceptual model in `hoc`, NEURON's programming language.

Chapter 2 Index

Chapter 3

Expressing conceptual models in mathematical terms

Computational neuronal modeling usually focusses on voltage and current in excitable cells, but it is often necessary to represent other processes such as chemical reactions, diffusion, and the behavior of electronic instrumentation. These phenomena seem quite different from each other, and each has evolved its own distinct "notational shorthand." Because these specialized notations have particular advantages for addressing domain-specific problems, NEURON has provisions that allow users to employ each of them as appropriate (see **Chapter 9: How to expand NEURON's library of mechanisms**). Apparent differences notwithstanding, there are fundamental parallels among these notations that can be exploited at the computational level: all are equivalent to sets of algebraic and differential equations. In this chapter, we will explore these parallels by examining the mathematical representations of chemical reactions, electrical circuits, and cables.

Chemical reactions

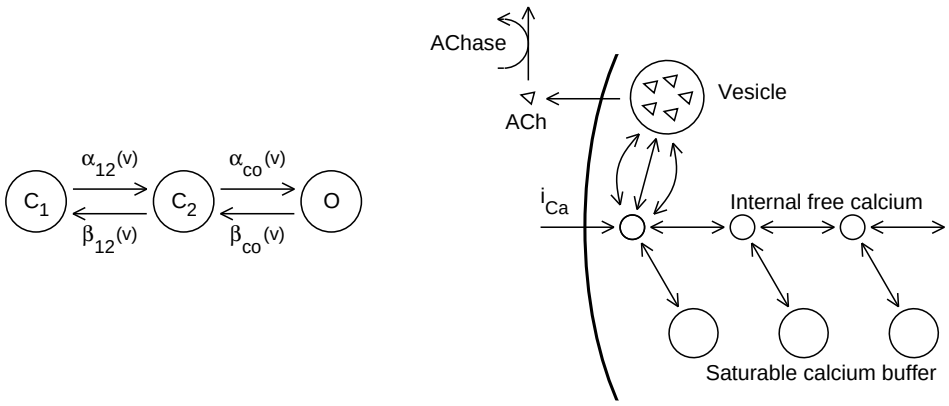


Fig. 3.1. Left: a voltage-gated channel modeled as a three-state kinetic scheme with voltage-dependent rate constants. Right: cartoon of a model of acetylcholine (ACh) release that involves the influx, buffering, and diffusion of calcium, exocytosis requiring binding of three calcium ions per vesicle, and enzymatic breakdown of ACh (rate constants omitted for clarity).

A natural first step in thinking about voltage-dependent or ligand-gated channel models or elaborate cartoons of dynamic processes is to express them with chemical reaction notation, i.e. kinetic schemes (Fig. 3.1). Kinetic schemes focus attention on conservation of material (in a closed set of reactions, material is neither created or destroyed) and flow of material from one state to another.

The notion of "state" is context-dependent: it may mean actual material quantity of a molecular species (sometimes moles, sometimes mass), the well-stirred molar concentration in a volume or the density on a surface, or the probability of a particle being in a particular state. Thus when we refer to "the value of state *A*" we mean a value expressed in the dimensions of *A*. When *A* is in units of concentration or density, "the

material in state *A*" is the product of *A* and the size of the compartment (volume or surface) in which *A* is distributed.

Flux and conservation in kinetic schemes

In a kinetic scheme, arrows that point toward or away from a state represent paths along which material enters or leaves the state. For each state there is a differential equation that expresses how the amount of material in the state is affected by fluxes that enter and leave it. These differential equations are specified by the states in the kinetic scheme and the paths along which material can move between them.

Thus

$$A \overset{k}{\rightarrow} B \qquad \text{Eq. 3.1}$$

means that material leaves state *A* at a rate that is proportional to the product of the value of *A* and a rate constant *k*, where *A* and *k* are understood to be nonnegative. From the standpoint of state *A*, the flux along this path is *-kA*, and this defines a term in the differential equation for this state.

$$\frac{dA}{dt} = -k\,A \qquad \text{Eq. 3.2a}$$

But the flux that leaves *A* in Eq. 3.1 is just the flux that enters *B*, so

$$\frac{dB}{dt} = k\,A \qquad \text{Eq. 3.2b}$$

Suppose we have a closed system in which Eq. 3.1 is the only chemical reaction that can occur. Adding Eqns. 3.2a and b together, we have

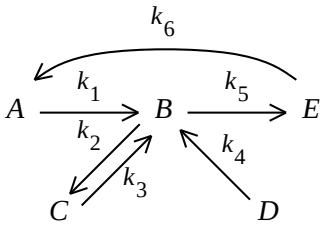
$$\frac{dA}{dt} + \frac{dB}{dt} = 0 \qquad \text{Eq. 3.3}$$

which we can integrate to get

$$A + B = \text{a constant} \qquad \text{Eq. 3.4}$$

Equation 3.4 is a statement of the principle of conservation of material: in a closed system with the reaction described by Eq. 3.1, the sum of *A* and *B* is conserved.

Any kinetic scheme is easily translated into a corresponding set of differential equations. Each differential equation expresses the rate of change of each state as the difference between the flux entering the state and the flux leaving the state. For example the kinetic scheme



Eq. 3.5

has five states, and is equivalent to five differential equations. Focussing on *B*, we see that the flux entering is the sum of k_1 *A*, k_3 *C*, and k_4 *D*, while the flux leaving is the sum of k_2 *B* and k_5 *B*, so the corresponding differential equation is

$$\frac{dB}{dt} = k_1\, A - (k_2 + k_5)\, B + k_3\, C + k_4\, D \qquad \text{Eq. 3.6a}$$

The differential equations for the other states are

$$\frac{dA}{dt} = -k_1\, A + k_6\, E \qquad \text{Eq. 3.6b-e}$$

$$\frac{dC}{dt} = k_2\ B - k_3\ C$$

$$\frac{dD}{dt} = -k_4\ D$$

$$\frac{dE}{dt} = k_5\ B - k_6\ E$$

To derive the conservation rules for a kinetic scheme, we just find linear combinations of these equations that add up to 0, and then integrate them. For the example of Eq. 3.6, we see that adding all of the equations together gives

$$\frac{dA}{dt} + \frac{dB}{dt} + \frac{dC}{dt} + \frac{dD}{dt} + \frac{dE}{dt} = 0 \qquad \text{Eq. 3.7}$$

which we integrate to obtain the conservation rule

$$A + B + C + D + E = \text{a constant} \qquad \text{Eq. 3.8}$$

i.e. the sum of the five states is conserved.

Stoichiometry, flux, and mole equivalents

In the reaction

$$A + B \underset{k_b}{\overset{k_f}{\rightleftarrows}} C \qquad \text{Eq. 3.9}$$

we see that producing one mole of *C* requires consumption of two moles: one mole of *A*, and one mole of *B*. That is, a change of *C* implies equal (but opposite) changes of *A*

and *B*. The forward flux is $k_f\ A\ B$ and the backward flux is $k_b\ C$, so this reaction translates to the differential equations

$$\frac{dA}{dt} = -k_f\ A\,B + k_b\ C$$

$$\frac{dB}{dt} = -k_f\ A\,B + k_b\ C \qquad \text{Eq. 3.10}$$

$$\frac{dC}{dt} = k_f\ A\,B - k_b\ C$$

from which we can generate several different linear combinations that add up to zero. Two obvious combinations are

$$\frac{dA}{dt} + \frac{dC}{dt} = 0 \qquad \text{Eq. 3.11a}$$

and

$$\frac{dB}{dt} + \frac{dC}{dt} = 0 \qquad \text{Eq. 3.11b}$$

from which we conclude that both $A + C$ and $B + C$ are conserved. Note that *A*, *B*, and *C* must have the same units (otherwise Eqns. 3.11a and b would involve the addition of dimensionally inconsistent values), while k_b has units of 1/time and k_f is in units of 1/time × units of *A*.

Confusion may occur with reactions like

$$A + A \underset{k_b}{\overset{k_f}{\rightleftarrows}} B \qquad \text{Eq. 3.12}$$

or the equivalent

$$2A \underset{k_b}{\overset{k_f}{\rightleftarrows}} B \qquad \text{Eq. 3.13}$$

if the underlying principle of conservation is overlooked. There is certainly no question that the equation for *B* is

$$\frac{dB}{dt} = k_f A^2 - k_b B \qquad \text{Eq. 3.14}$$

but what can we say about *dA/dt*?

To answer this question, we reexamine Eq. 3.13 and realize that it means that two moles of *A* produce one mole of *B*. So an increase of *B* implies twice as large a decrease of *A*, i.e.

$$\frac{dA}{dt} = 2\left(-k_f A^2 + k_b B\right) \qquad \text{Eq. 3.15}$$

From Eqns. 3.14 and 15 we see that, in a closed system described by Eq. 3.13,

$$\frac{dA}{dt} + 2\frac{dB}{dt} = 0 \qquad \text{Eq. 3.16}$$

and the conservation rule is that $A + 2B$ is constant.

Compartment size

Textbook treatments of kinetic schemes generally begin with the explicit assumptions that all states use identical dimensions (usually concentration) and are distributed in the same volume. Up to this point, we have tacitly made the same assumptions, because they

allow kinetic schemes to be translated into differential equations without having to take compartment size into account.

However, in neuronal modeling this is often too restrictive. Consider a model of the role of diffusion and active transport in regulating the amount of calcium in a thin shell adjacent to the cell membrane (Fig. 3.2). Some of the calcium is pumped out, and some diffuses between the shell and a bulk internal compartment ("core") at a rate that is proportional to a constant k_d.

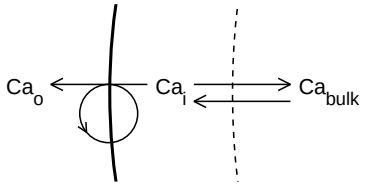


Fig. 3.2. In this model, $[Ca^{2+}]$ in a thin shell just inside the cell membrane is regulated by a pump in the cell membrane and by diffusional exchange with bulk stores of calcium in the core of the cell.

A kinetic scheme formulation of this model is

$$Ca_i \underset{k_d}{\overset{k_d}{\rightleftarrows}} Ca_{bulk} \qquad \text{Eq. 3.17a-c}$$

$$Ca_i + Pump \underset{k_2}{\overset{k_1}{\rightleftarrows}} Ca\ Pump$$

$$Ca\ Pump \underset{k_4}{\overset{k_3}{\rightleftarrows}} Ca_o + Pump$$

Here the active transport of calcium is represented by a pair of first order reactions between calcium ions in solution on either side of cell membrane, and a calcium pump that is restricted to the membrane. The states of this model are the amounts of calcium in the extracellular fluid (Ca_o), the shell (Ca_i), and the core of the cell (Ca_{bulk}), and the membrane-associated pump in its "free" and "calcium-bound" forms (*Pump* and *CaPump*, respectively). We want to translate these reactions into a corresponding set of differential equations, but the reactants occupy four regions, each of which has a different size. If the volume of the core (vol_{bulk}) is much larger than that of the shell (vol_{shell}), then a small amount of calcium could move from the core to the shell and have a significant effect on the concentration Ca_i while there is almost no change in the concentration Ca_{bulk}. And how do we deal with Eq. 3.17b and c, in which some reactants are described in terms of concentration, i.e. material/volume, while others are material densities, i.e. material/area?

In such situations, it is useful to realize that what we're trying to do is to write an equation for each state variable that expresses the rate of change of material as the difference between fluxes (material/time) into and out of the state. We start by defining the quantity of material as the product of the state variable and the size of its compartment, and then ensure that each term in the equation has the same units.

To see how this works, let's translate Eq. 3.17a-c into the corresponding differential equations. In order to avoid the distraction of scale factors, we start by assuming that areas and volumes and are in [cm^2] and [cm^3], respectively, while material densities (*Pump* and *CaPump*) are in micromoles per cm^2 [$\mu mole/cm^2$] and concentrations (Ca_o,

Ca_i, and Ca_{bulk}) are in [µmole/cm^3]. Later we will relax this assumption to see how scale factors enter into the picture.

We start with Ca_{bulk}, which has the simplest equation.

$$vol_{bulk} \frac{d\,Ca_{bulk}}{dt} = k_d\,Ca_i - k_d\,Ca_{bulk} \qquad \text{Eq. 3.18}$$

The total material in this state is $vol_{bulk}\,Ca_{bulk}$, the flux that enters it is $k_d\,Ca_i$, and the flux that leaves it is $k_d\,Ca_{bulk}$. The left hand side of Eq. 3.18 is the rate of change of material in this state, and it has units of [µmole/ms]. Since every term in this equation must have the same units, it is clear that k_d must be in [cm^3/ms].

The equation for Ca_o is

$$vol_o \frac{d\,Ca_o}{dt} = k_3\,CaPump - k_4\,Ca_o\,Pump \qquad \text{Eq. 3.19}$$

which, like Eq. 3.18, has units of [µmole/ms] on the left hand side. Since *CaPump* is in [µmole/cm^2], it follows that k_3 must have units of [cm^2/ms], and k_4 must be in [cm^5/ms µmole].

The state *CaPump* appears in two reactions, so its differential equation has more terms.

$$area_{pump} \frac{d\,CaPump}{dt} = k_1\,Ca_i\,Pump + k_4\,Ca_o\,Pump - (k_2 + k_3)\,CaPump \qquad \text{Eq. 3.20}$$

Once again the left hand side is in [μmole/ms], and it is clear that k_1 must have the same units as k_4, i.e. [cm^5/ms μmole], while the units of k_2 must be [cm^2/ms], identical to those of k_3.

The equation for *Pump* is

$$area_{pump}\frac{d\,Pump}{dt} = (k_2 + k_3)CaPump - (k_1 Ca_i Pump + k_4 Ca_o Pump) \quad \text{Eq. 3.21}$$

The terms on the right hand side of this equation are the same as those in Eq. 3.20 but with opposite signs, and units are obviously consistent throughout.

For Ca_i the equation is

$$vol_{shell}\frac{d\,Ca_i}{dt} = k_d Ca_{bulk} - k_d Ca_i - k_1 Ca_i Pump + k_2 CaPump \quad \text{Eq. 3.22}$$

and the units of all terms are consistent.

Scale factors

Up to this point we have used the same units for all calcium concentrations: [μmole/cm^3]. What if we prefer a more customary measure for Ca_o, e.g. [millimole/liter]? No problem--1 μmole/cm^3 is equivalent to 1 millimole/liter, i.e. the values are numerically equal, so we can use the same rate constants and equations as before, without having to insert scale factors into our equations.

Now suppose we decide that [cm^3] is too large a unit for intracellular volumes, and that we would prefer to use [μm^3] instead for vol_{bulk} and vol_{shell}? At first this seems perplexing, because the units of the left hand side of Eq. 3.18 would be [μm^3 μmole/ms cm^3], while the right hand side is still in [μmole/ms]. We are rescued from confusion by recalling that 1 μm = 10^{-4} cm, so [μm^3 μmole/ms cm^3] is equivalent to 10^{-12} [μmole/ms], and we have

$$vol_{bulk} \frac{d\,Ca_{bulk}}{dt} = 10^{12}\,(k_d Ca_i - k_d Ca_{bulk}) \qquad \text{Eq. 3.23}$$

The 10^{12} on the right hand side of Eq. 3.23 is a scale, or "conversion," factor, and if we wanted to be pedantic we would point out that it has units of [$cm^3/\mu m^3$]. In any case, its numeric value makes sense, because a small net movement of calcium will have a much larger effect on the concentration Ca_{bulk} if vol_{bulk} is 1 μm^3 rather than 1 cm^3.

Of course we also have to apply a scale factor in Eq. 3.22, the other equation that involves an intracellular volume. By identical reasoning we obtain

$$vol_{shell} \frac{d\,Ca_i}{dt} = 10^{12} \left(k_d Ca_{bulk} - k_d Ca_i - k_1 Ca_i Pump + k_2 CaPump \right) \qquad \text{Eq. 3.24}$$

Electrical circuits

An electrical circuit (Fig. 3.3) can be translated to an equivalent set of equations by combining Kirchhoff's current and voltage laws with the characteristics of the individual devices in the circuit. Here we present a brief heuristic approach to how this can be done. Space and time preclude discussion of related topics such as graph theory; for a more thorough development of circuit analysis, motivated readers are referred elsewhere (e.g. (Nilsson and Riedel 1996)).

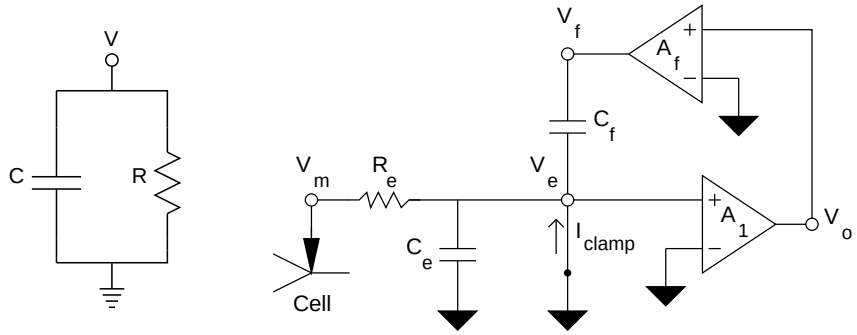


Fig. 3.3. Left: A simple parallel RC circuit. Right: Circuit for recording from a cell while passing current through the same electrode. Amplifier A_f and capacitor C_f are used to compensate for the electrode capacitance C_e.

To develop the equations that describe a circuit, we will employ Kirchhoff's current law which states that the algebraic sum of all currents entering a node (a connection between two or more device terminals) is always zero. Every node in a circuit has a voltage, and every connection between nodes ("branch" or "edge") has a current. In order for voltage throughout a circuit to be determined unambiguously, each node must be on a path that ultimately leads to ground. We can then write the current balance equation for

each node (the "node equations"), and solve these equations to find the potential at each node and the current through each element in the circuit.

Table 3.1 lists common circuit elements with their characteristic equations and schematic representations. The arrows in the figures of the resistor and capacitor indicate the direction of current flow when *i* given by the characteristic equation is positive. For the voltage source we have adopted the usual convention for the direction of positive current flow (away from the "positive" terminal, which is symbolized by the longer of the two transverse lines).

Table 3.1. Common circuit elements

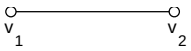
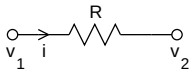
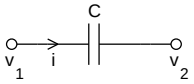
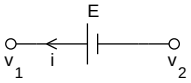
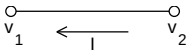
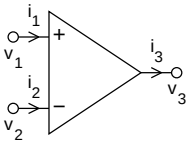
Type of element	Characteristic equation	Schematic representation
Ground	$v = 0$	
Wire	$v_1 = v_2$	
Linear resistor	$i = (v_1 - v_2)/R$	
Linear capacitor	$i = C\, d(v_1 - v_2)/dt$	
Voltage source	$v_1 - v_2 = E(t)$	
Current source	$i = I(t)$	
Ideal amplifier	$v_3 = G(v_1 - v_2)$ $i_1 = i_2 = 0$	

Figure 3.4 illustrates the application of Kirchhoff's current law to a circuit consisting of a capacitor in parallel with a resistor. There are two nodes, but one is grounded so its potential is 0. Since only one node has a potential that is unknown, this circuit can be described by a single node equation.

The current flow along all branches attached to the ungrounded node is indicated by the diagram on the right side of Fig. 3.4. To apply Kirchhoff's current law to this node, we must assume a positive direction for current flow along every edge that attaches to the node. We want to emphasize that these assumed directions are completely arbitrary, and no matter what we decide, the final equations will be the same. Here we have chosen the convention that current away from a node *adds* to the current balance equation, which gives us

$$I_C + I_R = 0 \qquad \text{Eq. 3.25}$$

Referring to Table 3.1 for the device properties of capacitors and resistors, we obtain the ordinary differential equation

$$C \frac{dV}{dt} + \frac{V}{R} = 0 \qquad \text{Eq. 3.26}$$

whose solution is

$$V(t) = V_0 e^{-t/RC} \qquad \text{Eq. 3.27}$$

where V_0 is the initial voltage on C.

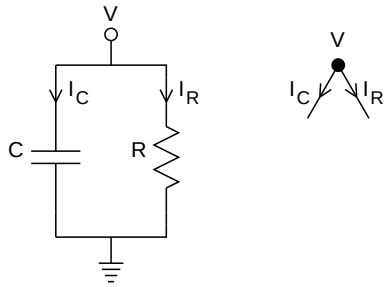


Fig. 3.4. Left: Schematic diagram of a simple parallel RC circuit, which has only one node at which potential is unknown. Right: Node diagram indicating the flow of current away from this single node.

The slightly more complex circuit of Fig. 3.5 has four nodes. There are only two nontrivial equations for the voltages at these nodes, since we already know that the grounded nodes have a voltage of 0. The potentials at the two ungrounded nodes are unknown, and we need to formulate the node equations for them. Once again, we can assign the directions of all currents arbitrarily, but once we have chosen the positive direction of current flow through R_3, we have committed ourselves to the positive direction of I_{R_3} relative to both node 1 and node 2. So if we assume that positive current in C_1, R_1, and R_3 flows away from node 1, and apply the convention that "current away from a node adds to the current balance equation," we have

$$I_{C_1} + I_{R_1} + I_{R_3} = 0 \qquad \text{Eq. 3.28a}$$

which is the current balance equation for node 1.

To get the other current balance equation, we will assume that the positive direction for current in C_2 and R_2 are away from node 2, so these currents add to its current

balance equation. However, we have already chosen a direction for positive current flow in R_3, and it happens to be *toward* node 2. The current flow diagrams for nodes 1 and 2 (Fig. 3.5 bottom right) underscores the fact that resistor R_3 makes equal but opposite contributions to current balance at nodes 1 and 2. Consequently the current I_{R_3} is subtracted from the current balance equation for node 2.

$$I_{C_2} + I_{R_2} - I_{R_3} = 0 \qquad \text{Eq. 3.28b}$$

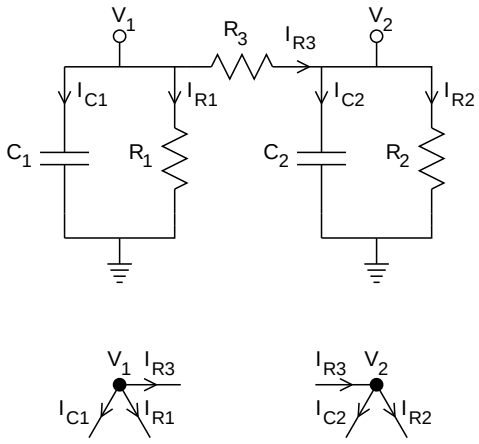


Fig. 3.5. Top: A circuit with three nodes. Bottom: Current flow diagram at each of the two nodes where potential is unknown. Note the direction of current flow in R_3.

Substituting device properties into these equations gives

$$C_1 \frac{dV_1}{dt} + \frac{V_1}{R_1} + \frac{(V_1 - V_2)}{R_3} = 0 \qquad \text{Eq. 3.29a}$$

$$C_2 \frac{dV_2}{dt} + \frac{V_2}{R_2} - \frac{(V_1 - V_2)}{R_3} = 0 \qquad \text{Eq. 3.29b}$$

Again note the - sign applied to the current in R_3 in the second node equation. This pair of coupled first order differential equations constitutes a second order initial value problem, which has a solution of the form

$$V_1(t) = A_1 e^{-t/\tau_a} + B_1 e^{-t/\tau_b} \qquad \text{Eq. 3.30a-b}$$

$$V_2(t) = A_2 e^{-t/\tau_a} + B_2 e^{-t/\tau_b}$$

where A_1, B_1, A_2, and B_2 are determined by the values of V_1 and V_2 at t = 0, and the time constants τ_a and τ_b are the eigenvalues of the matrix

$$\begin{bmatrix} -\frac{1}{C_1}\left(\frac{1}{R_1}+\frac{1}{R_3}\right) & \frac{1}{C_1 R_3} \\ \frac{1}{C_2 R_3} & -\frac{1}{C_2}\left(\frac{1}{R_2}+\frac{1}{R_3}\right) \end{bmatrix} \qquad \text{Eq. 3.31}$$

As a final example of the equivalence between an electrical circuit and a set of equations, let us consider a circuit that could be used to compensate for electrode capacitance. Anyone who has ever recorded from a cell with a microelectrode knows that electrode resistance and capacitance can interfere with experimental measurements. Figure 3.6 shows a simplified circuit of a common method used to compensate for electrode capacitance when recording with a sharp microelectrode under current clamp. This circuit includes a cell, a microelectrode whose electrical properties are represented by an equivalent circuit consisting of a series resistance R_e and a single lumped capacitance C_e located at the "amplifier" end of the electrode, a current source I_{clamp} for

injecting current into the cell, and a "headstage amplifier" A_1. It also has an amplifier A_f and capacitor C_f that provide positive feedback to compensate for the electrode capacitance.

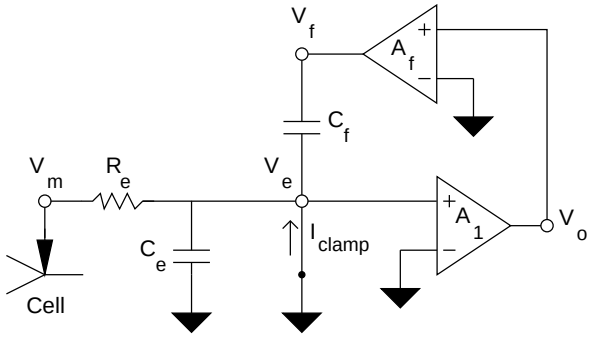


Fig. 3.6. Capacitance compensation under current clamp. The capacitance C_e of the microelectrode distorts recordings by slowing and attenuating the response of V_e to changes in V_m and I_{clamp}. Amplifier A_f and capacitor C_f compensate for this by supplying charging current to C_e.

The open circles mark the nodes that are not grounded. The first node is the site at which the electrode is attached to the cell, and the voltage at this node is V_m, the local membrane potential of the cell. As Fig. 3.7 suggests, the current balance equation for this node is

$$i_{inj} - i_{R_e} = 0 \qquad \text{Eq. 3.32}$$

i.e. the current i_{R_e} that flows through the electrode resistance equals the current i_{inj} that is injected into the cell.

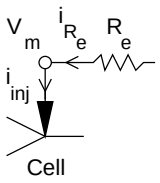


Fig. 3.7. The first node of the circuit in Fig. 3.6. The current injected into the cell equals the current that passes through the electrode resistance R_e.

The voltages at the remaining three nodes are unknown, so we will need three equations. Taking advantage of the characteristic equations for an amplifier (Table 3.1), we see immediately that the nodes at the outputs of the feedback and headstage amplifiers have voltages that are given by

$$V_f = G_f V_o \qquad \text{Eq. 3.33}$$

and

$$V_o = G_1 V_e \qquad \text{Eq. 3.34}$$

where G_f and G_1 are the "gains" or amplification factors of the feedback and headstage amplifiers, respectively. For the third equation, we apply Kirchhoff's current law to the remaining node, which is diagrammed in Fig. 3.8. The current balance equation for this node is

$$i_{R_e} + i_{C_e} - I_{clamp} - i_{C_f} + i_+ = 0 \qquad \text{Eq. 3.35}$$

Each device attached to this node contributes a term to Eq. 3.35, e.g. i_{C_e} is the current that charges the electrode capacitance, and i_{C_f} is the current supplied by the feedback

capacitor C_f. Referring to Fig. 3.6 and Table 3.1, we replace each term in Eq. 3.35 by the corresponding characteristic equation to get

$$\frac{V_e - V_m}{R_e} + C_e \frac{dV_e}{dt} - I_{clamp} - C_f \frac{d(V_f - V_e)}{dt} + 0 = 0 \qquad \text{Eq. 3.36}$$

which rearranges to

$$(C_e + C_f) \frac{dV_e}{dt} - C_f \frac{dV_f}{dt} = \frac{V_m - V_e}{R_e} + I_{clamp} \qquad \text{Eq. 3.37}$$

Interested readers may wish to combine Eqns. 3.33, 34, and 37 to derive a single differential equation that relates the "output" voltage V_o to the "input" voltage V_m.

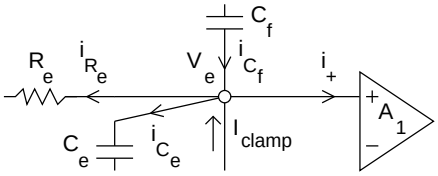


Fig. 3.8. The third node of the circuit in Fig. 3.6. Perfect compensation for electrode capacitance (which can never be achieved with real amplifiers and electrodes) requires that i_{C_f} balances i_{C_e} exactly.

Cables

The spread of electrical and chemical signals in a cable are described by equations that combine conservation laws with formulas that express how voltage and concentration gradients drive the movement of charge and mass. This discussion focusses

on electrical signals, since the basic form of these equations is identical for chemical signals (Rall 1977; Crank 1979; Carslaw and Jaeger 1980; Jack et al. 1983), and similar considerations arise in connection with their numerical solution.

The propagation of electrical signals along an unbranched cable is governed by the one-dimensional cable equation

$$\frac{\partial V}{\partial T} + F(V) = \frac{\partial^2 V}{\partial X^2} \qquad \text{Eq. 3.38}$$

where V and F are continuous functions of space and time, which are represented by X and T (with appropriate scaling) (Rall 1977; Jack et al. 1983). The branched architecture typical of most neurons is dealt with by combining partial differential equations of this form with appropriate boundary conditions. This is the approach taken in NEURON, whose programming language hoc and graphical user interface have special features that allow us to avoid the task of writing families of cable equations and puzzling out their boundary conditions. Instead, we construct models by specifying the properties of individual neurites and how they are interconnected. NEURON then applies the standard strategy of spatial and temporal discretization to convert our specification into algebraic difference equations, which it solves numerically (Rall 1964; Crank 1979; Carslaw and Jaeger 1980) (see **Chapter 4: Essentials of numerical methods for neural modeling**).

We can derive the cable equation by combining the physical principle of conservation of charge with Ohm's law. Focussing on these separately provides insight into the process of spatial discretization and the meaning of boundary conditions. In addition we can

easily handle issues of branching and spatially-varying diameter that were assumed away in the cable equation but are dominant physical features of real neurons.

Conservation of charge requires that the sum of currents flowing into any region from all sources must equal zero. For example, if Figure 3.9 represents part of a cell, conservation of charge means that

$$\sum i_a - \int_A i_m \, dA = 0 \qquad \text{Eq. 3.39}$$

where the first term is the sum of all axial currents i_a (in [mA]) flowing into the region through cross-section boundaries, and the second term is the total transmembrane current found by integrating the transmembrane current density i_m (in [mA/cm^2]) over the membrane area A (in [cm^2]) of the region. The usual sign convention is that outward transmembrane current is positive and axial current into a region is positive. If electrode current sources are present, they are treated exactly the same as membrane currents except for the sign convention, i.e. electrode current into a cell (depolarizing current) is positive. Including electrode current i_s in the conservation equation gives

$$\sum i_a - \int_A i_m \, dA + \int_A i_s \, dA = 0 \qquad \text{Eq. 3.40}$$

The physical size of electrode current sources is generally very small compared to the spatial extent of a region, so the mathematical form for i_s is usually a delta function of position i_s [mA] $\cdot$ δ(x-x$_0$, y-y$_0$, z-z$_0$) [cm^{-2}]. It becomes a matter of personal preference whether to keep electrode currents under an integral, analogous to distributed membrane

currents, or merely to add i_s [mA] to the sum of i_a in whatever region the electrode happens to be. In either case, the extra terms add nothing to the conceptual analysis, so we will omit them from the following equations to reduce irrelevant clutter.

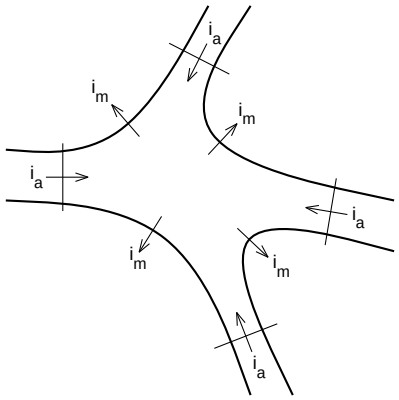


Fig. 3.9. The net current that flows into any region of a cell is 0. The arrows indicate the positive directions for transmembrane (i_m) and axial (i_a) currents.

A standard approach in computer simulation is to divide the neuron into regions or compartments small enough that the spatially-varying i_m in any compartment j is well approximated by its value at the center of the compartment. Equation 3.40 then becomes

$$i_{m_j} A_j = \sum_k i_{a_{kj}} \qquad \text{Eq. 3.41}$$

where A_j is the surface area of compartment j.

Up to this point we have relied entirely on the principle of conservation of charge. Ohm's law is invoked to resolve the axial currents between compartment j and its neighbors (right hand side of Eq. 3.41): each axial current is approximated by the voltage

drop between the centers of the compartments divided by the resistance of the path between them (the "axial resistance")

$$i_{a_{kj}} = (v_k - v_j) / r_{jk} \qquad \text{Eq. 3.42}$$

This transforms Eq. 3.41 into

$$i_{m_j} A_j = \sum_k (v_k - v_j) / r_{jk} \qquad \text{Eq. 3.43}$$

This automatically takes care of the direction of axial current flow, since $v_j < v_k$ implies that current flows into compartment *j*.

The total membrane current is the sum of capacitive and ionic components

$$i_{m_j} A_j = c_j \frac{dv_j}{dt} + i_{ion_j}(v_j, t) \qquad \text{Eq. 3.44}$$

where c_j is the membrane capacitance of the compartment and $i_{ion_j}(v_j, t)$ includes the effects of varying ionic channel conductances. In summary, the spatial discretization of branched cables yields a set of ordinary differential equations of the form

$$c_j \frac{dv_j}{dt} + i_{ion_j}(v_j, t) = \sum_k (v_k - v_j) / r_{jk} \qquad \text{Eq. 3.45}$$

As mentioned above, injected source currents would be added to the right hand side of this equation.

Equation 3.45 involves two approximations. First, axial current is specified in terms of the voltage difference between the centers of adjacent compartments. The second approximation is that spatially-varying membrane current is represented by its value at

the center of each compartment. This is much less drastic than the often heard statement that a compartment is assumed to be "isopotential." It is far better to picture the approximation in terms of voltage, membrane current, and axial current varying linearly between the centers of adjacent compartments. Indeed, the linear variation in voltage is implicit in the usual description of a cable in terms of discrete electrical equivalent circuits where all the membrane channels in a compartment have been pushed into a single point at the center of the compartment.

Two special cases of Eq. 3.45 deserve particular attention. The first of these allows us to recover the usual parabolic form of the cable equation. Consider the interior of an unbranched cable with constant diameter. The axial current consists of two terms involving compartments with indices j-1 and j+1, i.e.

$$c_j \frac{dv_j}{dt} + i_{ion_j}(v_j, t) = \frac{v_{j-1} - v_j}{r_{j-1,\,k}} + \frac{v_{j+1} - v_j}{r_{j+1,\,k}} \qquad \text{Eq. 3.46}$$

If each compartment has length Δx and diameter d, its capacitance is $C_m \pi d \Delta x$ and the axial resistance is $R_a \Delta x / \pi (d/2)^2$, where C_m is specific membrane capacitance and R_a is cytoplasmic resistivity. Equation 3.46 then becomes

$$C_m \frac{dv_j}{dt} + i_j(v_j, t) = \frac{d}{4 R_a} \frac{v_{j+1} - 2 v_j + v_{j-1}}{\Delta x^2} \qquad \text{Eq. 3.47}$$

where the total ionic current i_{ion_j} is replaced by the ionic current density i_j. As $\Delta x \to 0$, the right hand term becomes the second partial derivative of membrane potential with respect to distance at the location of the now infinitesimal compartment j, and we have

$$C_m \frac{\partial v}{\partial t} + i(v, t) = \frac{d}{4 R_a} \frac{\partial^2 v}{\partial x^2} \qquad \text{Eq. 3.48}$$

Multiplying both sides by R_m and recognizing that $i R_m = v$ gives

$$R_m C_m \frac{\partial v}{\partial t} + v = \frac{d R_m}{4 R_a} \frac{\partial^2 v}{\partial x^2} \qquad \text{Eq. 3.49}$$

Scaling t and x by the time and space constants $\tau_m = R_m C_m$ and $\lambda = \frac{1}{2}\sqrt{\frac{d R_m}{R_a}}$ (i.e. substituting $T = t / \tau_m$ and $X = x / \lambda$) transforms Eq. 3.49 into the form shown in Eq. 3.38.

The second special case of Eq. 3.45 allows us to recover the boundary conditions. This is an important issue since naive discretizations at the ends of the cable have destroyed the second order accuracy of many simulations. The boundary condition for the terminal end of a nerve fiber is that no axial current flows at the end of the cable, i.e. the end is sealed. This is implicit in Eq. 3.45, where the right hand side will consist only of the single term $(v_{j-1} - v_j) / r_{j-1, j}$ when compartment j lies at the end of an unbranched cable.

References

Carslaw, H.S. and Jaeger, J.C. *Conduction of Heat in Solids*. 2 ed. Oxford: Oxford University Press, 1980.

Crank, J. *The Mathematics of Diffusion*. 2 ed. London: Oxford University Press, 1979.

Jack, J.J.B., Noble, D., and Tsien, R.W. *Electric Current Flow in Excitable Cells*. London: Oxford University Press, 1983.

Nilsson, J.W. and Riedel, S.A. *Electric Circuits*. 5 ed. Reading, MA: Addison-Wesley, 1996.

Rall, W. Theoretical significance of dendritic tree for input-output relation. In: *Neural Theory and Modeling*, edited by R.F. Reiss. Stanford: Stanford University Press, 1964, p. 73-97.

Rall, W. Core conductor theory and cable properties of neurons. In: *Handbook of Physiology, vol. 1, part 1: The Nervous System*, edited by E.R. Kandel. Bethesda, MD: American Physiological Society, 1977, p. 39-98.

Chapter 3 Index

C

Chapter 4

Essentials of numerical methods for neural modeling

NEURON uses many strategies to achieve computational accuracy and efficiency, some of which are detailed elsewhere (Hines 1984). It also draws on several numerical methods libraries for vector (Press et al. 1992) and matrix (Stewart and Leyk 1994) methods, solving systems of sparse equations (Sparse 1.3 (Kundert 1986)), and adaptive integration (CVODES (Hindmarsh and Serban 2002) and IDA (Hindmarsh and Taylor 1999)). These all make their own special contributions to NEURON's performance, but they are already well documented elsewhere so this chapter will not discuss them in any detail. Instead, the emphasis will be on how NEURON deals with the fact that neurons are distributed analog systems that are continuous in time and space, but digital computation is inherently discrete. Because of this fundamental disparity, implementing a model of a neuron with a digital computer raises many purely numerical issues that have no relationship to the biological questions that are of primary interest, yet must be addressed if simulations are to be tractable and produce reliable results.

We saw in **Chapter 3** that the principle of conservation of charge can be expressed with a single ordinary differential equation

$$C \frac{dV}{dt} + I_{ion} = I_{inj} \qquad \text{Eq. 4.1}$$

so long as the transmembrane current density is nearly uniform over the surface of a cell. If current density varies too much, the computational representation must consist of two or more coupled compartments. These are described by a set of equations of the form

$$C_j \frac{dv_j}{dt} + I_{ion_j} = \sum_k \frac{v_k - v_j}{r_{jk}} + I_{inj_j} \qquad \text{Eq. 4.2}$$

where the second term on the right hand side is the sum of all axial currents from neighboring compartments. Additional terms and equations are necessary if extracellular fields (the `extracellular` mechanism) or electronic instrumentation (linear circuits) are to be included in the simulation.

Selection of a method for numerical integration of these equations is guided by concerns of stability, accuracy, and efficiency. In this chapter we review these important concepts and explain the rationale for the integrators used in NEURON. We start with a theoretical analysis of the errors that are introduced by discretizing the linear cable equation. Then we move on to a comparative analysis of methods for computing numerical solutions, which is illustrated by a series of case studies that bring up issues related to the practical concerns of empirically-based modeling.

Spatial and temporal error in discretized cable equations

A linear cable with uniform properties is described by the equation

$$c \frac{\partial V}{dt} + g V = \frac{a}{2 R_a} \frac{\partial^2 V}{\partial x^2} \qquad \text{Eq. 4.3}$$

where V is the membrane potential in volts, c the specific membrane capacitance in [F/cm^2], g the specific membrane conductance in [S/cm^2], a the radius in [cm], R_a the cytoplasmic resistivity in [Ω cm], and x the distance along the cable in [cm], so that each term in Eq. 4.3 has units of [A/cm^2]. We assume that the cable is L cm long, and that the axial current at each end is zero, i.e. "sealed end" boundary conditions, which implies that $\partial V / \partial x = 0$ at $x = 0$ and $x = L$. The membrane potential is a function of time and location $V(t,x)$, and the initial condition $V(0,x)$ can be any spatial pattern that satisfies the boundary conditions.

Analytic solutions: continuous in time and space

The spatial patterns that preserve their shape, changing only in amplitude, are the Fourier cosine terms $\cos(\pi n x/L)$. From Fourier theory, we know that any spatial pattern can be represented as an infinite sum of such cosine patterns (Strang 1986).

These cosine patterns always satisfy the boundary condition at x=0 because sin(0)=0. Satisfaction of the boundary condition at $x = L$, i.e. $\sin(\pi n) = 0$, requires that n be an integer. The pattern preserves its shape because substituting $V(t,x) = V_n(t) \cos(\pi n x/L)$ into Eq. 4.3 gives

$$c \frac{d V_n(t)}{dt} + g V_n(t) = - \frac{\pi^2 n^2 a}{2 R_a L^2} V_n(t) \qquad \text{Eq. 4.4}$$

which has the solution

$$V_n(t) = V_n(0)\,\mathrm{e}^{-k_n t} \qquad \text{Eq. 4.5a}$$

where n is the number of half waves in the cosine pattern, $V_n(0)$ is its initial amplitude, and the rate of decay is

$$k_n = \frac{g}{c} + \frac{\pi^2 n^2 a}{2 R_a L^2 c} \qquad \text{Eq. 4.5b}$$

When $n = 0$, voltage is independent of location along the length of the cable and decays with the membrane time constant $\tau_m = c/g$ seconds (top graph in Fig. 4.1). If n is large, i.e. when the spatial frequency of the cosine pattern is high, the second term on the right hand side of Eq. 4.5b is dominant, so the pattern decays very quickly at a rate that is proportional to the square of the number of half waves on the cable (see Fig. 4.1, especially the bottom graph). In a continuous cable, there is no limit to the spatial frequency, but high spatial frequencies decay extremely quickly.

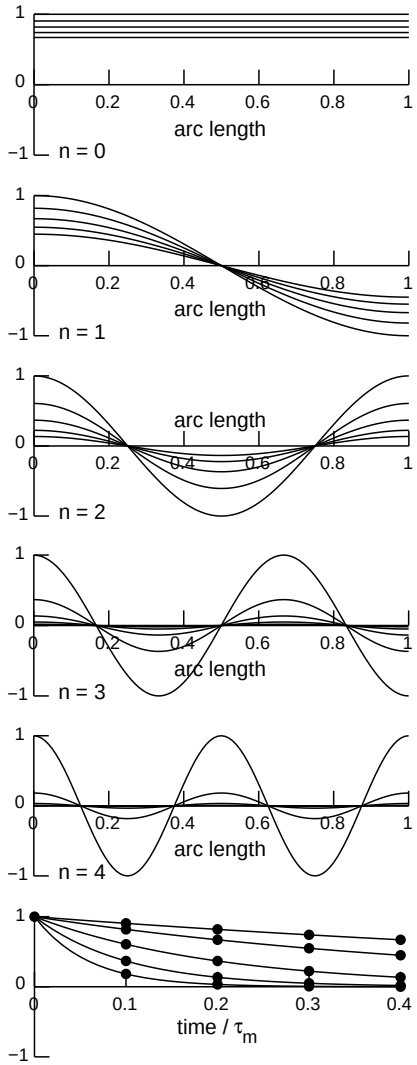


Figure 4.1. Top five graphs: These are the first five spatial patterns of *V* that preserve their shape along a uniform cylindrical passive cable. *V* is plotted as a function of normalized distance along the cable for *n* = 0, 1, 2, 3, and 4 half cycles. The decay of these patterns with time is illustrated by "snapshots" taken at *t* = 0, 0.1, 0.2, 0.3, and 0.4 times the membrane time constant τ_m. Note that larger *n* implies faster decay. Bottom graph: Amplitudes of these patterns plotted as functions of normalized time. Starting with the top trace and working down, *n* = 0, 1, 2, 3, and 4. Dots mark the amplitudes at the times of the snapshots shown in the upper graphs. These amplitudes assume cable

length is π times its DC length constant λ, so that $n = 1$ makes the first and second terms of Eq. 4.5b equal. Shorter cables have bigger k_n, hence decay is more rapid.

Adding a current stimulus to the equations is not difficult, but the detailed derivation is not necessary to our discussion of discretization error. Two points are worth mentioning, however. First, any stimulus can be represented as a Fourier sum. Second, a cosine stimulus with a specific spatial frequency excites a voltage response with the same spatial frequency and an amplitude that follows a single exponential decay, asymptotically approaching a steady state.

Spatial discretization

Now let us compare the continuous cable solution of Eq. 4.5 with the solution of a cable equation that has been discretized in space by replacing $\partial^2 V / \partial x^2$ with the second order correct approximation

$$\frac{\partial^2 V}{\partial x^2} \approx \frac{V(x+\Delta x) - 2V(x) + V(x-\Delta x)}{\Delta x^2} \qquad \text{Eq. 4.6}$$

For concreteness we need to specify precisely which values of x are allowed. The ordinary approach is to suppose m points with the first point at $x = 0$ and the last point at $x = L$, so that $\Delta x = L/(m\text{-}1)$. However, NEURON takes a different approach to discretization, in which there are m intervals of length $\Delta x = L/m$ and the m points are at the centers of these intervals. Thus the centers are at $x = (i + 0.5)L/m$ where $0 \le i < m$.

With either method, m is the number of points in space at which a numerical solution for V is computed, and $m = 1$ corresponds to a spatial frequency of 0, i.e. uniform membrane potential along the entire cable. Furthermore, for either approach the largest number of half waves that can be represented in the discretized system is $n = m$-1 so the highest spatial frequency is $(m\text{-}1)/2L$ cycles per unit length. This result is related to the Nyquist sampling theorem, which states that at least two samples must be captured per cycle in order to accurately measure the frequency of a signal (Strang 1986).

The ordinary method puts the ith point at $x = iL/(m\text{-}1)$, so $\cos(\pi nx/L) = \cos(\pi(m\text{-}1)iL/(m\text{-}1)L) = \cos(\pi i)$, and the value of V alternates sign at adjacent points. With NEURON's method, the largest n is also m-1 because, at $n = m$, $\cos(\pi nx/L) = \cos(\pi m(m+0.5)L/mL) = \cos(\pi(m+0.5)) = 0$.

With the ordinary method, the second difference at the ith point is most easily computed from the real part of

$$\begin{aligned} & e^{j\pi n(i+1)/(m-1)} - 2e^{j\pi ni/(m-1)} + e^{j\pi n(i-1)/(m-1)} \\ & = \left(e^{j\pi n/(m-1)} - 2 + e^{-j\pi n/(m-1)}\right) e^{j\pi ni/(m-1)} \\ & = 2\left(\cos\left(\pi n/(m-1)\right) - 1\right) e^{j\pi ni/(m-1)} \end{aligned} \qquad \text{Eq. 4.7}$$

which is

$$2\left(\cos\left(\pi n/(m-1)\right) - 1\right) \cos\left(\pi ni/(m-1)\right) \qquad \text{Eq. 4.8}$$

NEURON's method gives

$$2\left(\cos\left(\pi n/m\right)-1\right)\cos\left(\pi n\left(i+0.5\right)/\left(m-1\right)\right) \qquad \text{Eq. 4.9}$$

Therefore, for either method

$$\frac{dV_{nm}}{dt} = -k_{nm}V_{nm} \qquad \text{Eq. 4.10}$$

where

$$k_{nm} = \frac{g}{c} + \frac{\left(1-\cos\left(\pi n\Delta x/L\right)\right)a}{R_a c \Delta x^2} \qquad \text{Eq. 4.11}$$

The solution of Eq. 4.10 is

$$V_{nm}(t) = V_{nm}(0)\,\mathrm{e}^{-k_{nm}t} \qquad \text{Eq. 4.12}$$

Note that k_{nm} approaches k_n (Eq. 4.5b) when $n\Delta x/L$ is << 1 (because $\cos(\phi) \approx 1 - \phi^2/2$ when ϕ is small). This makes sense when one realizes that L/n is half of the wavelength of the spatial pattern, so "$n\Delta x/L$ is small" means that the discretization interval Δx is short compared to the wavelength of the spatial pattern. Thus the discrete system is "sampling" the spatial pattern at an interval that is fine enough to allow a smooth representation of the pattern. Restating this in more formal terms, the discretized system approximates the original continuous system more closely at those spatial frequencies for which the discretization interval Δx is short compared to the spatial wavelength.

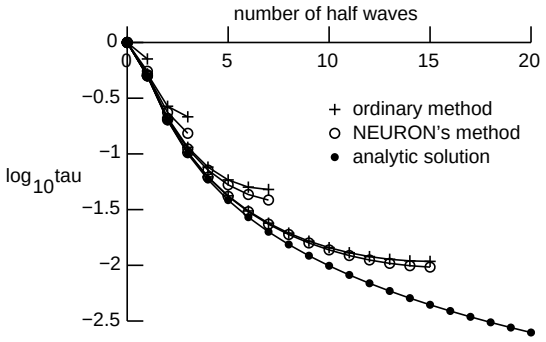


Figure 4.2. Normalized time constant for decay of spatial patterns vs. number of half waves along a uniform passive cylindrical cable (cable parameters as in Fig. 4.1).

Figure 4.2 shows the normalized time constant of decay $\tau = 1/k\tau_m$ as a function of the number of half waves for the continuous cable of Fig. 4.1 as well as for discretized models of this cable with 2, 4, 8, and 16 points. We must point out that, for both discretization methods, doubling the number of points reduces the error in the time constant for a given spatial frequency by a factor of 4. Also note that, for small numbers of compartments and at the highest spatial frequencies, the spatial error of NEURON's discretization method is significantly less than that of the ordinary method.

Adding temporal discretization

So far we have solved the spatially continuous and spatially discretized cables analytically with respect to time. Now we complete the discretization with respect to time. The numerical integration methods that have seen the widest use in empirically-based neural modeling are forward Euler, backward Euler, and Crank-Nicholson. Later in this chapter we will examine each of these individually and in more detail. For the

purpose of our present theoretical analysis, it is better to treat them all at once by introducing a parameter θ so that

$$\frac{dV}{dt} \approx \frac{V(t+\Delta t) - V(t)}{\Delta t} \qquad \text{Eq. 4.13a}$$

is evaluated at $t+\theta\Delta t$ by using V interpolated from its values at t and $t+\theta\Delta t$, i.e.

$$V(t+\theta\Delta t) = (1-\theta)V(t) + \theta V(t+\Delta t) \qquad \text{Eq. 4.13b}$$

Thus Eq. 4.10 becomes

$$\frac{V(t+\Delta t) - V(t)}{\Delta t} = -k_{nm} V(t+\theta\Delta t) \qquad \text{Eq. 4.14}$$

Drawing on Eq. 4.13b, we can write this as

$$\frac{V(t+\Delta t) - V(t)}{\Delta t} = -k_{nm}\left((1-\theta)V(t) + \theta V(t+\Delta t)\right) \qquad \text{Eq. 4.15}$$

When θ = 0 Eq. 4.15 is the forward Euler method, θ = 1 turns it into the backward Euler method, and θ = 0.5 gives us the Crank-Nicholson method.

From Eq. 4.15 we immediately get the iteration equation

$$V_{nm}(t+\Delta t) = \left(\frac{1-(1-\theta)k_{nm}\Delta t}{1+\theta k_{nm}\Delta t}\right) V_{nm}(t) \qquad \text{Eq. 4.16}$$

The first term on the right hand side of this equation is the iteration coefficient; if its magnitude for any spatial frequency is > 1, the iterations will diverge. With the forward Euler method (θ = 0), the iteration coefficient with the largest magnitude is for the spatial

frequency at which $n = m$. At this frequency, the $\cos(\pi n \Delta x/L)$ term in Eq. 4.11 is -1, making the decay rate constant

$$k_{mm} = \frac{g}{c} + \frac{2a}{R_a c \Delta x^2} \qquad \text{Eq. 4.17}$$

so we see that the magnitude of the iteration coefficient is > 1 when $k_{mm}\Delta t > 2$. If we want the discretized system to represent high spatial frequencies, Δx must be small, and this makes the second term in k_{mm} dominant. Substituting $\theta = 0$ and $k_{mm} \approx 2a/R_a c \Delta x^2$ into Eq. 4.16 and rearranging, we find that, for the forward Euler method to avoid numerical instability, the combination of Δt and Δx must obey the constraint

$$\frac{\Delta t}{\Delta x^2} < \frac{R_a c}{a} \qquad \text{Eq. 4.18}$$

With the backward Euler method ($\theta = 1$), there is no constraint on Δt because k_{nm} is always positive and so the iteration coefficient is greater than 0 and less than 1. For the Crank-Nicholson method ($\theta = 0.5$), the iteration coefficient never becomes less than -1, so this method is formally stable for all Δt.

Numerical integration methods

Now we continue our comparative analysis of numerical methods for integrating Eq. 4.1 and 4.2 by examining them in the context of practical examples. We start with the simplest approach: explicit or forward Euler, which is *not* used in NEURON for reasons

that will become clear. Then we consider the implicit or backward Euler method, Crank-Nicholson, CVODE, and DASPK, which are all available in NEURON.

Forward Euler: simple, inaccurate and unstable

Suppose we are modeling a neuron that has nearly uniform transmembrane current density. For our conceptual model of this cell, we also assume that its resting potential is 0 mV, its membrane conductance g is constant and linear, and that we are not injecting any current into it. The techniques we use to understand and control error in simulations of this linear, passive model are immediately generalizable to active and nonlinear cases.

Conservation of charge in this model is described by Eq. 4.1, which simplifies to

$$\frac{dV}{dt} + kV = 0 \qquad \text{Eq. 4.19}$$

where the rate constant k is the inverse of the membrane time constant $\tau_m = g/c$. The analytic solution of Eq. 4.19 is

$$V(t) = V(0)\mathrm{e}^{-kt} \qquad \text{Eq. 4.20}$$

Let us compare this to a numeric solution computed with the forward Euler method.

The forward Euler method is based on a simple approximation. From the initial conditions we know the starting value of the dependent variable ($V(0)$), and the differential equation that describes the model (Eq. 4.19) gives us the initial slope of the solution ($-kV(0)$). The approximation assumes that the slope of the solution is constant for a short period of time. Then we can extrapolate from the value of V at time 0 to a new value a brief interval into the future. Now we see why this is called the "forward" Euler

method: we are starting from something that is already known and projecting into the future. The forward Euler method is one of many integrators that calculate future values entirely on the basis of present, and possibly also past, values; these are called "explicit" integrators to distinguish them from "implicit" integrators, such as backward Euler and Crank-Nicholson (see below), which involve future values in the calculation.

In general terms, if a system is described by the differential equation

$$\frac{dV}{dt} = f(V, t) \qquad \text{Eq. 4.21}$$

then the forward Euler method approximates a solution by repeatedly applying

$$V(t + \Delta t) = V(t) + f(V(t), t)\,\Delta t \qquad \text{Eq. 4.22}$$

For this example, Eq. 4.22 becomes

$$V(t + \Delta t) = V(t) - k\,V(t)\,\Delta t \qquad \text{Eq. 4.23}$$

(cf. Eq. 4.16 with $\theta = 0$).

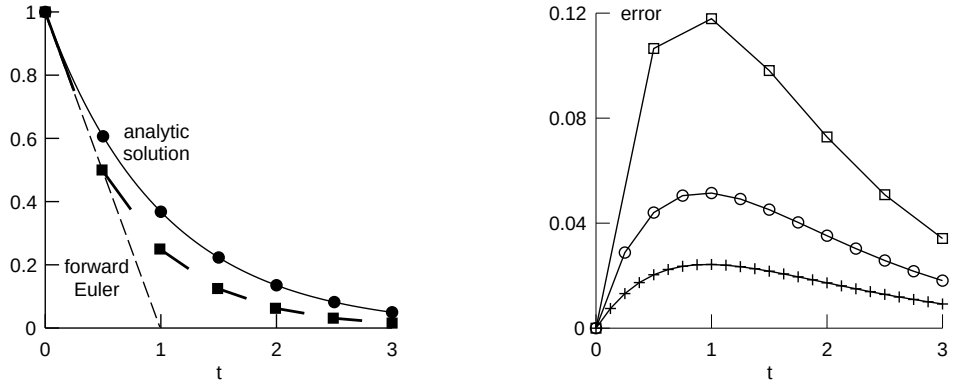


Figure 4.3. Left: analytic solution to Eq. 4.19 (solid line with circles) and results of the forward Euler method (squares) for $V(0) = 1$, $k = 1/s$, and $\Delta t = 0.5$ s (modified from (Hines and Carnevale 1997)). Right: absolute error of the forward Euler method with $\Delta t = 0.5$ (squares), 0.25 (circles), and 0.125 s (+).

The left panel of Fig. 4.3 shows the forward Euler solution obtained for rate parameter $k = 1/s$ (i.e. 1/second), initial condition $V(0) = 1$, and time interval Δt over which we extrapolate, assuming the transmembrane ionic current is constant within each interval. The current that is used for a given interval is found from the value of the voltage at the beginning of the interval (filled squares). This current determines the slope of the line segment that leads to the voltage at the next time step. The dashed line shows the value of the voltage after the first time step as a function of Δt. Corresponding values for the analytic solution (solid line) are indicated by filled circles.

The issue of accuracy in numerical simulation is complex, and we discuss it more thoroughly later in this chapter (see **Error**). For the moment we only mention that the forward Euler method has "first order accuracy," which means that the local error is proportional to Δt. This is demonstrated in the right panel of Fig. 4.3, where the absolute

difference between the analytic solution and the results of the forward Euler method is plotted for Δt = 0.5, 0.25, and 0.125 s (squares, circles, and +, respectively). Cutting Δt by a factor of 2 reduced error by very nearly half (Δt was comparable to the model's time constant (1 s) so slight deviations from strict proportionality are to be expected).

Numerical instability

We have already broached this topic from a theoretical standpoint in the setting of a uniform cable model (see **Adding temporal discretization** above), but it is still useful to consider stability of numerical integration in the context of "simpler" compartmental models. What would happen if the forward Euler method were applied to Eq. 4.19 using a very large time step, e.g. Δt = 3 s? The simulation would become numerically unstable, with the first step extrapolating down to V = -2, the second step going to V = -2 + 6 = 4, and each successive step oscillating with geometrically increasing magnitude.

Simulations of the two compartment model on the left of Fig. 4.4 demonstrate an important aspect of instability. Suppose the initial condition is V = 0 in one compartment and V = 2 in the other. According to the analytic solution, at first the potentials in the two compartments converge rapidly toward each other (time constant = 1/41 s), and later they decay slowly toward 0 (time constant = 1 s).

If we use the forward Euler method with Δt = 0.5 s, we realize that there will be a great deal of trouble during the time where the voltages are changing rapidly. We might imagine that we can deal with this by choosing a Δt that will carefully follow the time

course of the voltage changes, i.e. let Δt be small when they are changing rapidly, and larger when they are changing slowly.

The results of this strategy are shown on the right of Fig. 4.4. After 0.2 s with Δt = 0.001 s, the two voltages have nearly come into equilibrium. Then we changed Δt to 0.2 s, which is small enough to follow the slow decay closely. Unfortunately, no matter how small the difference between the voltages, the difference grows geometrically at each time step. This happens even if the difference consists only of roundoff error, because the time step used in the forward Euler method must never be more than twice the smallest time constant in the system.

Linear algebra clarifies the notion of "time constant" and its relationship to stability. For a linear system with N compartments, there are exactly N spatial patterns of voltage over all compartments such that only the amplitude of the pattern changes with time, while the shape of the pattern is preserved. The amplitudes of these patterns or "eigenfunctions" are given by $e^{t\lambda_i}$, where λ_i is called the eigenvalue of the ith eigenfunction. The real part of each eigenvalue is the reciprocal of one of the time constants of the solutions to the differential equations that describe the system. The ith pattern decays exponentially to 0 if the real part of λ_i is negative; if the real part is

Earlier in this chapter (see **Spatial and temporal error in discretized cable equations**) we saw that the eigenfunctions for a uniform cylindrical cable with sealed ends took the form of cosine waves. The decay rates k_n and k_{nm} of that theoretical discussion equal -1 times the corresponding eigenvalues.

positive, the amplitude grows catastrophically. If λ_i has an imaginary component, the pattern oscillates with frequency $\omega_i = \mathrm{Im}(\lambda_i)$. In a passive electrical system that contains only resistance and capacitance, all λ_i are real and negative.

Our two compartment model has two such patterns. In one, the voltages in the two compartments are identical; this pattern decays with the time course e^{-t}. The other pattern, in which the voltages in the two compartments are equal but have opposite sign, decays with the much faster time course e^{-41t}.

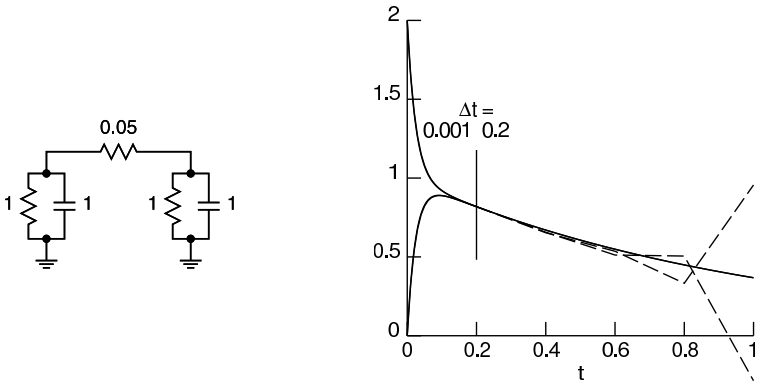


Figure 4.4. Left: The two compartments of this model are connected by a small axial resistance, so the membrane potentials are normally in quasi-equilibrium with each other while at the same time decaying fairly slowly toward 0. Right: The forward Euler method (dashed lines) is numerically unstable whenever Δt is greater than twice the smallest time constant. The analytic solution (solid lines) is the sum of two exponentials with time constants 1 s and 1/41 s. The solution step size was 0.001 s for the first 0.2 s, after which it increased to 0.2 s. Modified from (Hines and Carnevale 1997).

The key idea is that a problem involving N coupled differential equations can always be transformed into a set of N independent equations, each of which is solved separately. Numerical solution of these equations must use a time step Δt that is small enough for the solution of each equation to be stable. This why stability criteria that involve Δt depend on the smallest time constant.

If the ratio between the slowest and fastest time constants is large, the system is said to be stiff. Stiffness is a serious problem because a simulation may have to run for a very long time in order to show changes governed by the slow time constant, yet a small Δt has to be used to follow changes due to the fast time constant.

Signal sources may change the stability properties of a system by altering the time constants that describe it. A current source (perfect current clamp) does not affect stability because it does not affect the time constants. Any other signal source imposes a load on the compartment to which it is attached, changing the time constants and the corresponding eigenfunctions. The more closely it approximates a voltage source (perfect voltage clamp), the greater this effect will be.

Backward Euler: inaccurate but stable

The numerical stability problems of the forward Euler method can be avoided if the equations are evaluated at time $t + \Delta t$, i.e. the approximate solution is found from

$$V(t + \Delta t) = V(t) + f(V(t + \Delta t), t + \Delta t)\, \Delta t \qquad \text{Eq. 4.24}$$

which is called the implicit or backward Euler method. This equation can be derived from Taylor's series truncated at the Δt term but with $t + \Delta t$ in place of t.

For our example with one compartment, the backward Euler method gives

$$V(t + \Delta t) = \frac{V(t)}{1 + k\,\Delta t} \qquad \text{Eq. 4.25}$$

(cf. Eq. 4.16 with $\theta = 1$). Figure 4.5 shows several iterations of Eq. 4.25. Each step moves to a new point $(t_{i+1}, V(t_{i+1}))$ such that the slope there points back to the previous point $(t_i, V(t_i))$. If Δt is very large, the solution does not oscillate with geometrically increasing amplitude like the forward Euler method, but instead converges geometrically toward the steady state.

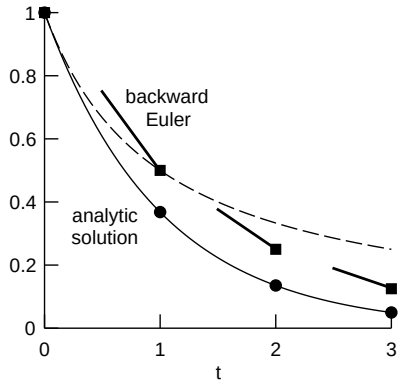


Figure 4.5. Comparison of analytic solution to Eq. 4.19 (solid line with circles) with results from the backward Euler method (Eq. 4.25, squares) for $V(0) = 1$, $k = 1/s$, and $\Delta t = 1$ s. At the end of each step, the slope at the new value (heavy lines) points back to the beginning of the step. The dashed line shows the voltage after the first time step as a function of Δt. Modified from (Hines and Carnevale 1997).

The robust stability of the backward Euler method are readily demonstrated by applying it to the two compartment model (Fig. 4.6). Notice that a large Δt gives a reasonable qualitative understanding of model behavior, even though the solution does not follow the early rapid voltage changes. Furthermore the step size can be changed according to how quickly the state variables are changing, yet the solution remains stable.

The backward Euler method requires solution of a set of nonlinear simultaneous equations at each step. To compensate for this extra work, the step size needs to be as large as possible while preserving good quantitative accuracy. Like the forward Euler method, backward Euler has first order accuracy (see **Error** below), but it is more practical for initial exploratory simulations since reasonable values of Δt produce fast simulations that are almost always qualitatively correct, and, as we have seen here, tightly coupled compartments do not generate large error oscillations but instead come quickly into equilibrium because of its excellent stability.

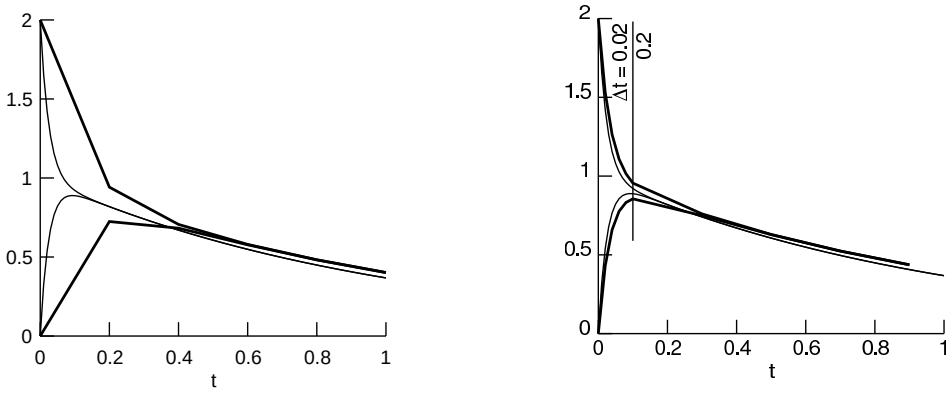


Figure 4.6. Simulation of the two compartment model of Fig. 4.4 using the backward Euler method. Left: Δt = 0.2 s, much larger than the fast time constant. Right: Δt was initially 0.02 s, small enough to follow the first time constant closely. After 0.1 s, Δt increased to 0.2 s but the solution remained stable. Thin lines are analytic solution, thick lines are backward Euler solution. Modified from (Hines and Carnevale 1997).

Crank-Nicholson: stable and more accurate

The central difference or Crank-Nicholson method (Crank and Nicholson 1947) is an implicit integrator that is equivalent to advancing by one half step with backward Euler and then advancing by another half step with forward Euler (Fig. 4.7). The value at the end of each step is along a line determined by the estimated slope at the midpoint of the step. The local error of this method is proportional to the square of the step size, so for a given Δt we can expect a large accuracy increase. In fact, simulation of our one compartment model with Δt = 1 s (Fig. 4.7) is much more accurate than the forward Euler simulation with Δt = 0.5 s (Fig. 4.3).

A most convenient feature of the central difference method is that the amount of computational work for the extra accuracy beyond the backward Euler method is trivial, since after computing $V(t + \Delta t/2)$ with backward Euler, we just have

$$V(t + \Delta t) = 2V(t + \frac{\Delta t}{2}) - V(t) \qquad \text{Eq. 4.26}$$

so the extra accuracy does not cost extra computations of the model functions.

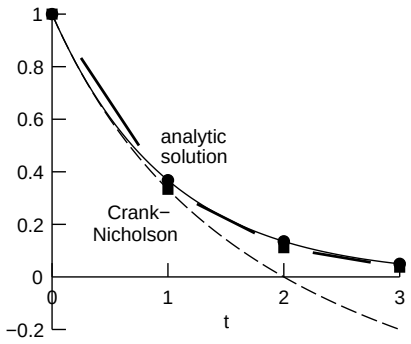


Figure 4.7. Simulations of the one compartment model with the Crank-Nicholson method, which uses the slope at the midpoint of the step (short thick lines) to determine the new value (squares). These are almost indistinguishable from the analytic solution (solid line with circles). The dashed line shows the voltage after the first time step as a function of Δt. Modified from (Hines and Carnevale 1997).

One might well ask what effect the forward Euler half step has on numerical stability. The left panel in Fig. 4.8 shows the solution for the two compartment model of Fig. 4.4 computed using the central difference method with Δt much larger than the fast time constant. The sequence of a backward Euler half step followed by a forward Euler half step approximates an exponential decay by

$$V(t + \Delta t) = V(t) \frac{1 - 0.5k\Delta t}{1 + 0.5k\Delta t} \qquad \text{Eq. 4.27}$$

(cf. Eq. 4.16 with $\theta = 0.5$). As Δt becomes very large, the step multiplier approaches -1 from above, so the solution oscillates with decreasing amplitude. Technically speaking the Crank-Nicholson method is stable because the error oscillations decay with time.

To prevent oscillations in the numeric solution for a model of a cylindrical cable, the normalized increments in time ($\Delta T = \Delta t / \tau_m$) and space ($\Delta X = \Delta x/\lambda$, where Δx is the distance between adjacent nodes and λ is the DC length constant) must satisfy the relationship $\Delta T/\Delta X \leq 1/2$ (see chapter 8 in Crank (1979)).

This example demonstrates that artifactual large amplitude oscillations may result if the time step is too large. Such oscillations can affect simulations of models that involve voltage clamps or in which very small resistances couple adjacent segments. However, in some cases oscillations can be minimized by using small Δt while the solution contains a large amplitude component that is changing rapidly, and increasing Δt after the slower components dominate the solution (Fig. 4.8 right).

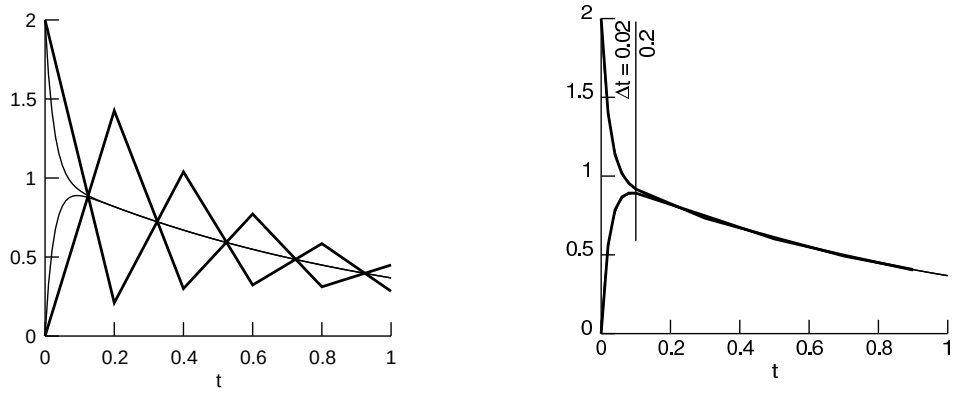


Figure 4.8. Simulations of the two compartment model using the Crank-Nicholson method. Left: Significant error oscillations can appear when the simulation has a large amplitude component with a time constant much smaller than Δt. However, the simulation is numerically stable because the oscillation amplitude decreases at each step. Right: Δt was initially 0.02 s, i.e. smaller than the fastest time constant (~ 0.0244 s), so the simulation followed the rapid collapse of the fast component. After 0.1 s, Δt increased to 0.2 s; this provoked oscillations, but their amplitude is only a small fraction of the total response and decays rapidly, so the trajectories appear smooth. Thin lines are analytic solution, thick lines are Crank-Nicholson solution. Modified from (Hines and Carnevale 1997).

Efficient handling of nonlinearity

Nonlinear equations generally need to be solved iteratively to maintain second order accuracy. However, voltage-dependent membrane properties, which are typically formulated in analogy to Hodgkin-Huxley (HH) type channels, allow the cable equation to be cast in a linear form that can be solved without iterations yet is still second order correct. A direct solution of the voltage equations at each time step $t \rightarrow t + \Delta t$ using the linearized membrane current $I(V,t) = g\ (V - E)$ is sufficient as long as the slope

conductance g and the effective reversal potential E are known to second order at time $t + 0.5\Delta t$. HH type channels are easy to solve at $t + 0.5\Delta t$ since the conductance is a function of state variables that can be computed using a separate time step offset by $0.5\Delta t$ with respect to the voltage equation time step. That is, to integrate a state from $t - 0.5\Delta t$ to $t + 0.5\Delta t$ we only require a second order correct value for the voltage-dependent rates at the midpoint time t.

Figures 4.9 and 10 illustrate the differences between the unstaggered and staggered time step approaches. The left panel of Fig. 4.9 shows membrane potential v and the gating variable m from an action potential simulation computed with the ordinary, i.e. unstaggered, implementation of the Crank-Nicholson method. The superior accuracy achieved with staggered time steps is apparent in Fig. 4.10. The middle panels of these two figures zoom in on the solutions between 2.0 and 2.2 ms to reveal the sequence of calculations. The right panels demonstrate that using staggered time steps turns a system of differential equations with nonlinear coupling into a linear system of decoupled equations, so that second order accuracy is achieved without having to resort to iterations.

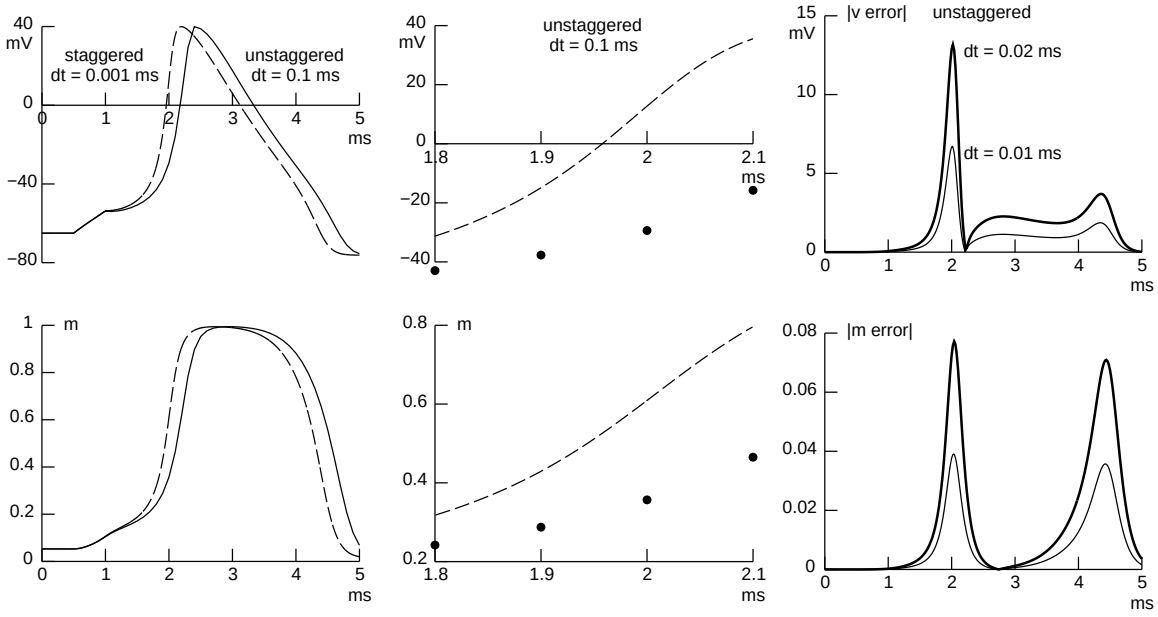


Figure 4.9. Simulated response of a 100 μm^2 patch of membrane with HH channels to a 0.025 nA current lasting 0.5 ms, computed with the ordinary (unstaggered) Crank-Nicholson method using time step Δt = 0.1 ms. Left: The spike was noticeably delayed compared to the standard for accuracy (dashed traces, computed with Crank-Nicholson using staggered time steps and Δt = 0.001 ms). Similar errors were observed in *h* and *n* (traces omitted for clarity). Middle: A magnified view of these solutions from 2.0 to 2.2 ms. Dots mark the individual values computed by the unstaggered Crank-Nicholson method. The unstaggered method advances the solution in two stages. First the new membrane potential $v(t + \Delta t)$ is computed from the values of *v*, *m*, *h*, and *n* at *t*. Then the new values of *m*, *h*, and *n* are computed analytically from their values at *t* and the average of the old and new membrane potentials $(v(t) + v(t+\Delta t)) / 2$. Right: The absolute error of *v* and *m* is proportional to the integration time step Δt, i.e. the solution has only first order accuracy.

For HH equations in a single compartment, using staggered time steps converts four simultaneous nonlinear equations at each step to four independent linear equations that have the same order of accuracy. Since the voltage-dependent rates use the voltage at the midpoint of the integration step, integration of channel states can be done analytically with just a single addition and multiplication and two table lookup operations. While this efficient scheme achieves second order accuracy, the tradeoff is that the tables depend on the value of Δt and must be recomputed whenever Δt changes.

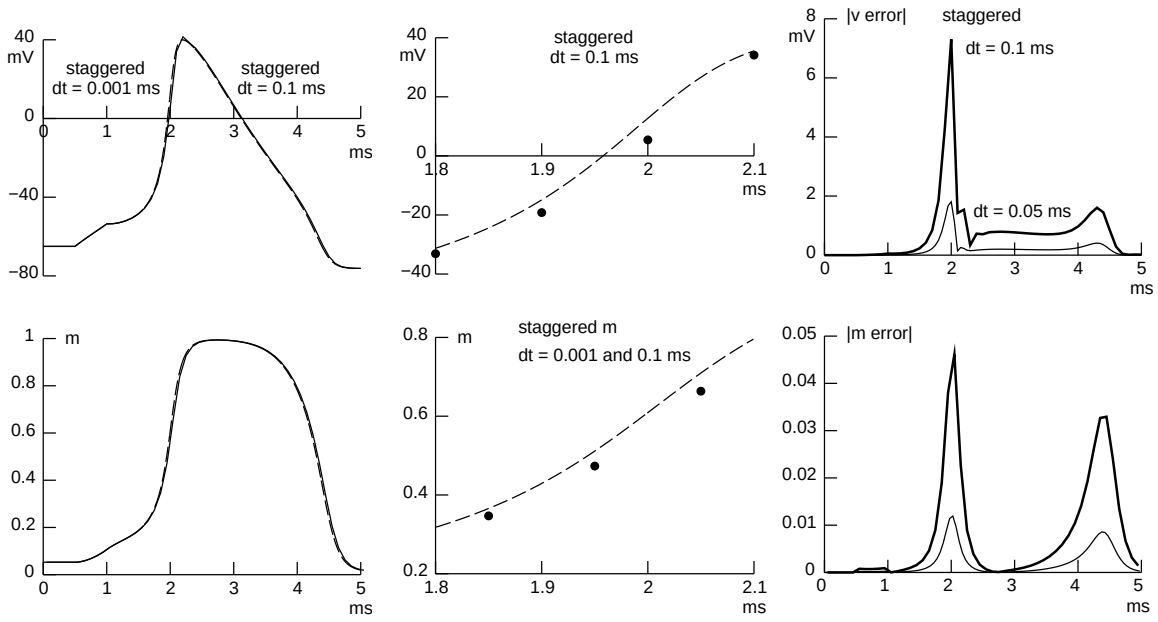


Figure 4.10. Simulated action potential from the same model as in Fig. 4.9, but computed with Crank-Nicholson using staggered time steps. Left: The solution with Δt = 0.1 ms was almost indistinguishable from the standard for accuracy. Similar improvements were observed in *h* and *n*. Right: An expanded view of these solutions, with dots marking the values computed with Δt = 0.1 ms. First the values of *m*, *h*, and *n* at $t + 0.5\Delta t$ are computed analytically from their values at $t - 0.5\Delta t$ and the membrane potential *v* at *t*. Then the values of *m*, *h*, and *n* at $t + 0.5\Delta t$ are used to update *v* from *t* to $t + \Delta t$. Right: Plots of the absolute error of *v* and *m* show that the error is proportional to the square of the integration time step Δt, i.e. using staggered time steps increases solution accuracy to second order.

Adaptive integration: fast or accurate, occasionally both

There is a wide variety of problems for which an adaptive time step method might have much higher performance than a fixed step method, e.g. Δt could grow very large when all states are varying slowly, as during interspike intervals. On the other hand, in problems involving propagating action potentials or networks of cells, it may happen that some state somewhere in the system is always changing quickly. In such cases Δt is always small in order to follow whichever state is varying fastest. Thus it is often not clear in advance that the increased overhead of an adaptive time step method will be repaid by an occasional series of long time steps.

Implementational considerations

The variable order variable time step integrator CVODE was written by Cohen and Hindmarsh (Cohen and Hindmarsh 1994, 1996) to solve ordinary differential equation (ODE) initial value problems of the form

$$\mathbf{y}' = \boldsymbol{f}(\mathbf{y}, t) \qquad \text{Eq. 4.28a-c}$$

$$\mathbf{y}(0) = \mathbf{y}_0$$

$$\mathbf{y} \in \mathbb{R}^N$$

where $\mathbf{y}'$ is the first derivative of $\mathbf{y}$ with respect to t, and bold face is used to signify vectors (lower case) and matrices (upper case). Since there are many different adaptive integrators, it is worthwhile to review the reasons why CVODE is particularly relevant to NEURON.

1. CVODE employs Backward Differentiation Formula (BDF) methods suitable for stiff problems, which are common in neuronal modeling.

2. CVODE was easily interfaced to the existing NEURON structure. It would be neither convenient nor efficient to gather *all* of the equations for every compartment and every membrane mechanism into one huge bag and throw it at a solver. The interface between ODE solver and the definition and setup of equations that are already distributed among membrane mechanisms requires a map between the internal NEURON states and the ODE state vector **y**, as well as a map between the internal computations for $\boldsymbol{f}$ and the ODE state derivative vector $\mathbf{y}'$. Programming an efficient map between the distributed internal Jacobian ($\mathbf{J} = \partial \boldsymbol{f}/\partial \mathbf{y}$) evaluation and a sparse matrix representation is possible but complex. The CVODE solver obviates this problem since it allows programmers to define their own problem-dependent linear solvers. This means NEURON can exploit the existing block structure of the Jacobian matrix and reuse the local block solvers that are already distributed within the membrane mechanism objects.

3. CVODE (and DASPK--see below) allows a sophisticated balance between accuracy of solution of $\mathbf{M}\,\mathbf{y} = \mathbf{b}$ and solution time by supporting the preconditioned iterative Krylov method, which requires one to only supply a solver for $\mathbf{P}\,\mathbf{y} = \mathbf{b}$, where **P** is in some sense an approximation to **M** such that $\mathbf{P}^{-1}\,\mathbf{M}$ is approximately the identity matrix and is chosen so that computation of the inverse of **P** is much faster than computation of the inverse of **M**. Small off-diagonal elements in the Jacobian are usually ignored for Gaussian elimination efficiency, but can occasionally have an

adverse effect on stability and thereby limit the effective time step. It is not yet clear which method is more robust when such off-diagonal terms are ignored in the context of nerve simulations: the Krylov method, or direct use of the approximate Jacobian in CVODE.

4. Finally, CVODE was implemented using encapsulated data structures, so it was conceptually simple to place it in an object-oriented class wrapper for use in implementing a local variable time step method. An important pre-existing feature of CVODE that helped support local variable time steps was the ability to efficiently retreat to any time within the previous integration interval.

Unfortunately, models that contain linear circuits and extracellular fields cannot be expressed, or at least are not easy to express, in the form shown in Eq. 4.28. Such models take the form

$$\mathbf{C}\,\mathbf{y}' = \boldsymbol{f}(\mathbf{y}, t) \qquad \text{Eq. 4.29a-c}$$

$$\mathbf{y}(0) = \mathbf{y}_0$$

$$\mathbf{y} \in \mathbb{R}^N$$

where some rows in the **C** matrix may be 0 (introduction of algebraic equations), and the nonzero rows may have off-diagonal elements (capacitors between nodes). In principle one could use the singular value decomposition of **C** to recast the system as

$$\mathbf{z}' = \boldsymbol{g}(\mathbf{z}, \mathbf{x}, t) \qquad \text{Eq. 4.30a and b}$$

$$\mathbf{0} = \boldsymbol{h}(\mathbf{z}, \mathbf{x}, t)$$

and satisfy the latter constraint directly whenever $\boldsymbol{f}$ is calculated. This is what NEURON does with the zero area nodes at the ends of sections, where membrane potential is governed by an algebraic equation rather than an ODE, without too much trouble and with no loss of efficiency. However, in practice $\boldsymbol{f}(\mathbf{y}, t)$ is given by an algorithm which one cannot multiply by a matrix. Also the sparse structure of $\boldsymbol{f}$ is generally lost in the transformation, making $\boldsymbol{g}$ much more dense and hence less efficient to solve.

For these reasons, when extracellular or linear circuit mechanisms are present and a variable step integration method is requested, the fast CVODE method is replaced by the slower but more robust DASPK method of Brown, Hindmarsh, and Petzold (Brown et al. 1994), which is available from `http://netlib.org`.

The user's perspective

A key feature of using CVODE is that one does not set the integration step size, but instead specifies tolerance criteria for local relative and absolute errors. The solver then adjusts Δt and the local error order of the implicit difference approximation (from first order up to $O(\Delta t^6)$) so that the local error for each state is less than the sum of its relative and absolute error tolerances.

Figure 4.11 illustrates the performance of CVODE in simulations of the two compartment model using two different values for the local absolute error tolerance. CVODE is capable of a high degree of accuracy, but caution must be exercised in setting the error tolerance, and it is a good idea to compare results against fixed time step methods during (and even after) model development.

For a more biologically relevant example of how CVODE can reduce the time necessary to produce accurate simulations, let us compare simulations of a neocortical layer V pyramidal cell model (Mainen and Sejnowski 1996) generated with the Crank-Nicholson and CVODE integrators. The model was subjected to a 900 ms depolarizing current applied to the soma, which evoked two bursts of spikes (Fig. 4.12 top). A series of simulations was run with the Crank-Nicholson method using progressively smaller Δt until the time at which the last action potential crossed above 0 mV converged to a constant value; this occurred for $\Delta t \leq 0.01$ ms, and a simulation performed with Δt = 0.01 ms took 340 seconds to complete on a 2.2 GHz Pentium 4 PC with 512 K cache. Solutions computed with CVODE converged to the same zero crossing time of the last spike, i.e. same global error, when absolute tolerance was $2.5 \cdot 10^{-3}$ for all states except $[Ca^{2+}]_i$, which had an absolute tolerance of $2.5 \cdot 10^{-7}$; using these tolerances, the solution runtime was 19 seconds. Thus CVODE achieved the same accuracy as the most accurate fixed time step solution, but with a runtime that was almost 20 times faster.

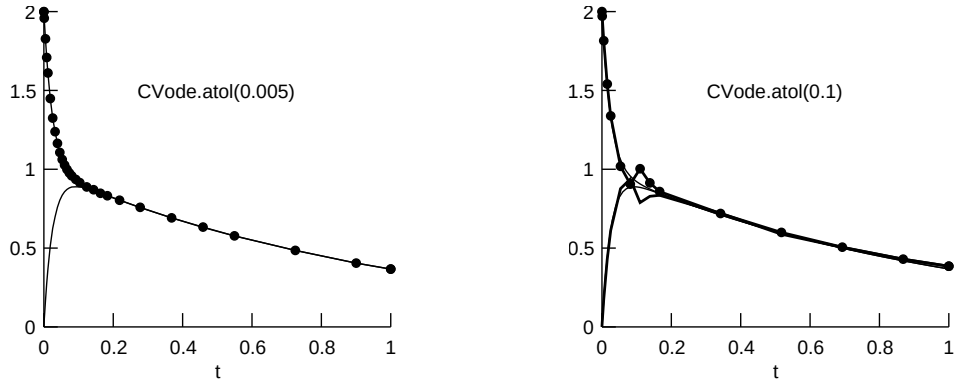


Figure 4.11. Simulations of the two compartment model using CVODE. Left: Filled circles on one of the traces mark the times at which CVODE calculated solutions. When the solution is changing rapidly, Δt is very small, but it grows quite large when the solution changes slowly. If the local absolute error tolerance is sufficiently strict (0.005 for this example), there is no visible difference between the computed and analytic solutions. Right: Thin lines are the analytic solution, thick lines the CVODE solution. Increasing the error tolerance allows CVODE to take larger steps, but spurious transients may occur if the criterion is too lax.

The bottom panel of Fig. 4.12 demonstrates the control that CVODE exerted over Δt throughout the entire simulation. When states were changing most rapidly, Δt fell to values much smaller than 0.01 ms, but during the long interburst interval it increased to a maximum of ~4.4 ms. The smallest steps were restricted to the onset and offset of the injected current (t = 5 and 905 ms) and brief intervals starting just before the threshold and ending shortly after the depolarized peak of each spike, as can be seen in an expanded view of the transition from the interburst interval to the beginning of the second

burst (Fig. 4.13). The remarkable speedup by CVODE is due to the fact that Δt was much larger than 0.01 ms for most of the simulation.

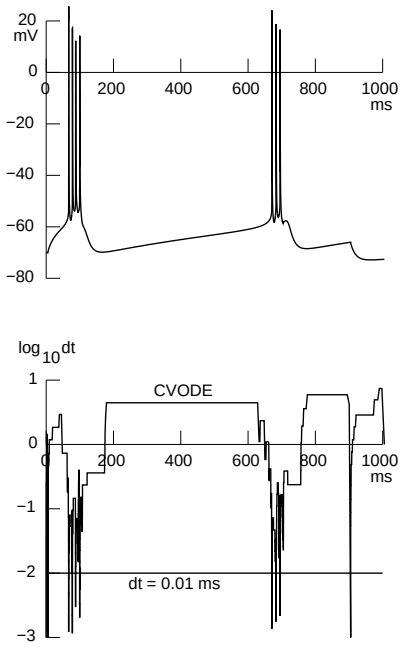


Figure 4.12. Top: CVODE was used to compute somatic membrane potential in a model of a neocortical layer V pyramidal cell subjected to a long depolarizing current pulse; Crank-Nicholson method with Δt = 0.01 ms produced results that are indistinguishable at the scale of this figure. Bottom: For most of the simulation, CVODE used time steps much larger than 0.01 ms. The order of integration (not shown) ranged from 2 to 5, most steps being second or third order. Figure from (Hines and Carnevale 2001).

The only difficulty that CVODE introduced is an excessive literalness required for interpretation of discrete functions. To see what this means, consider this strategy for emulating a "ramp clamp": filling the elements of a `Vector` with a linearly increasing sequence of values and then using the `Vector` class's `play()` method to drive the command potential of a voltage clamp. Figure 4.14 shows this technique applied to a

single compartment model with HH currents that was clamped by an `SEClamp` (series resistance `rs` = 10^6 Ω). The elements of a `Vector` were assigned the series of values -65+0.125i for $0 \leq i \leq 401$, i.e. a linear ramp that swept from -65 to -15 mV over the course of 10 ms, assuming Δt = 0.025 ms. A second `Vector` filled with the corresponding times (0.025i) was used to insure that each command potential in the sequence was applied at the proper time.

Simulations of this model using the implicit Euler method with a 0.025 ms time step display smoothly varying membrane potential and clamp current, even when examined at the scale of individual time steps (Fig. 4.14 right). This is because the stream of values delivered by the `Vector` is equivalent to a second order piecewise linear function, i.e. command potential itself varies smoothly with time.

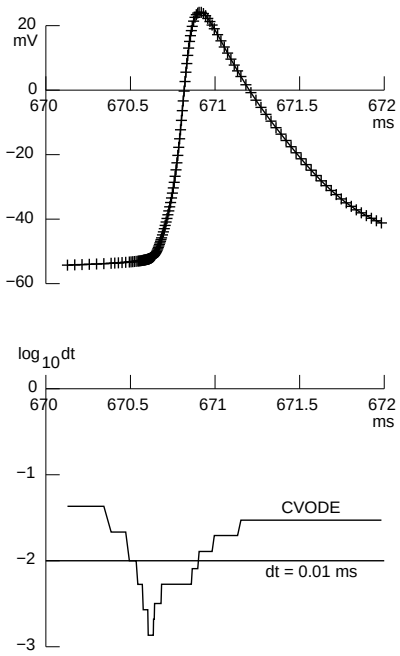


Figure 4.13. Top: An expanded view of the first spike in the second burst from Fig. 4.12. The times of computed solutions are marked by + symbols. Bottom: Δ*t* fell below 0.01 ms from just before the threshold of each spike until shortly after its peak. Figure from (Hines and Carnevale 2001).

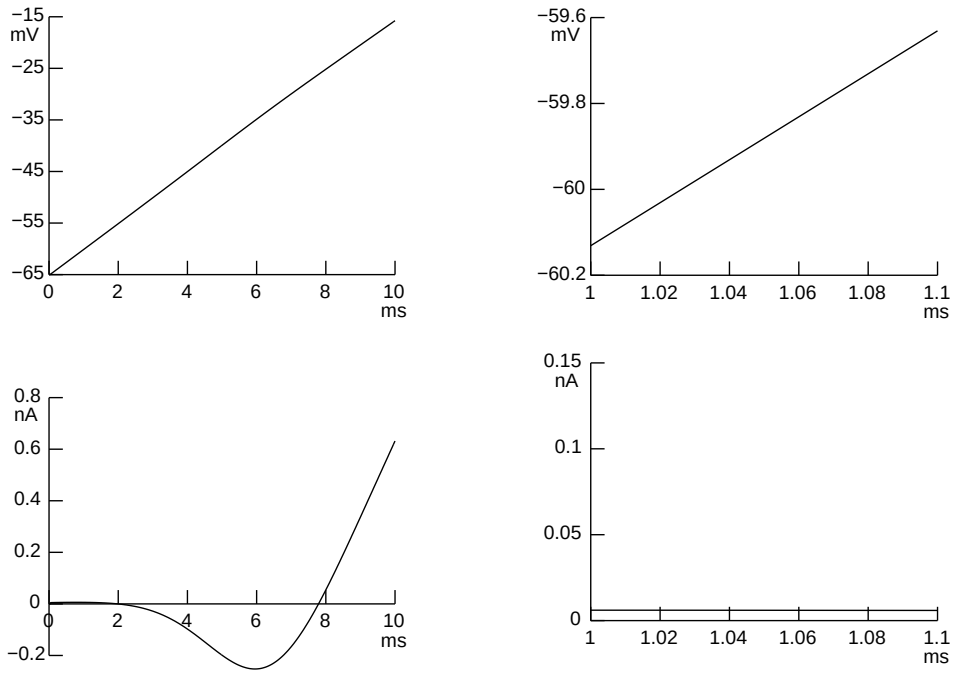


Figure 4.14. Ramp clamp using the `Vector` class's `play()` method works well with fixed Δt integration because command potential is effectively a continuous function of time. Top traces are membrane potential, bottom traces are clamp current.

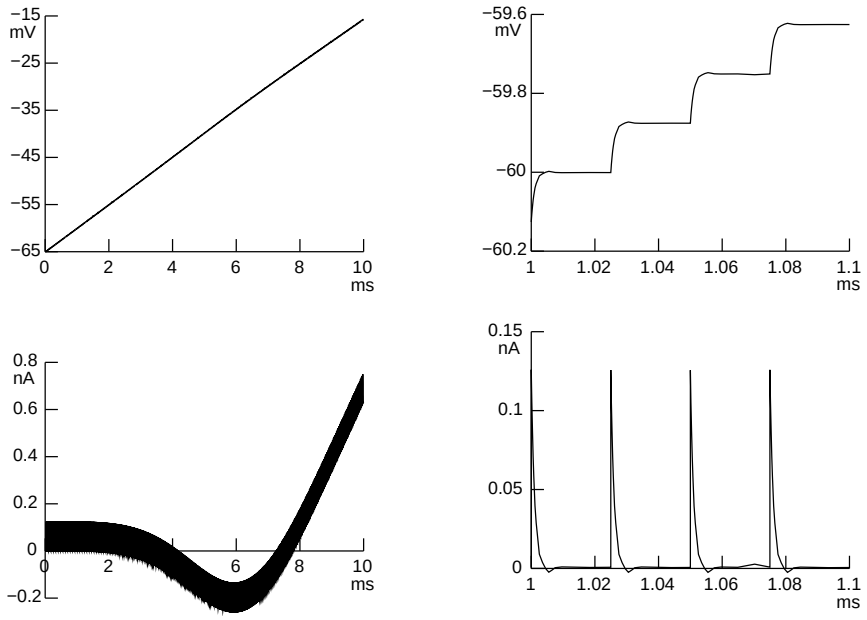


Figure 4.15. Using `Vector.play()` with CVODE produces large capacitive transients in clamp current (bottom traces) because the value sequence in the `Vector` that drives command potential is treated as a first order step function. The local absolute error tolerance parameter `atol` is 0.001 in this simulation and in Fig. 4.16.

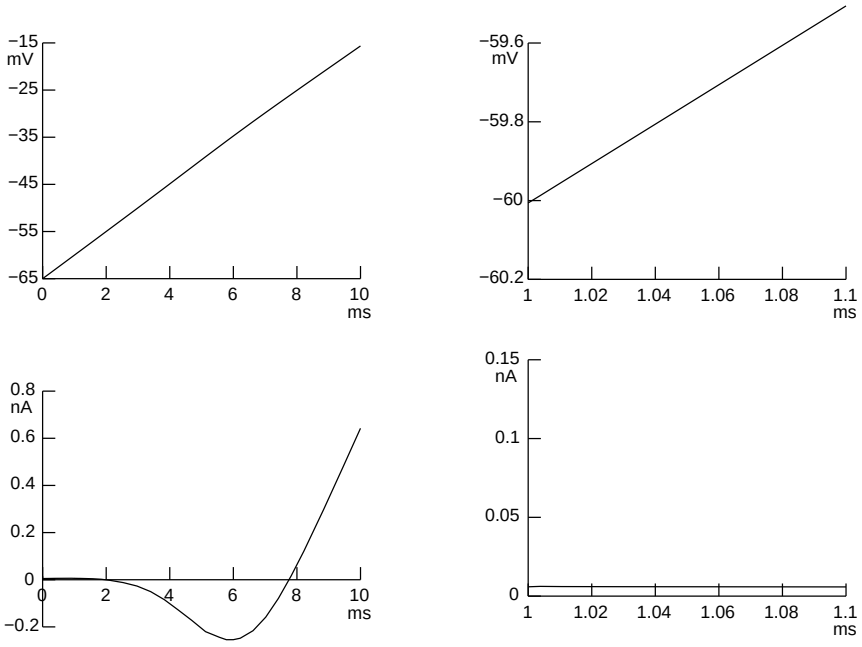


Figure 4.16. `Vector.play()` with interpolation works well with CVODE because the `Vector` that drives command potential is treated as a piecewise linear function. See text for details.

However, under CVODE the stream must be considered a first order equivalent step function. Driving the voltage clamp with this step function makes membrane potential jump from one level to another and produces substantial capacitance current transients at each step discontinuity (Fig. 4.15).

This problem has been addressed in NEURON 5.4 by adding a linear interpolation option to the `Vector` class's `play()` method. This option, which works both with fixed Δt and CVODE, treats our two vectors as if they defined a piecewise linear function. This means we can represent the ramp command used in this example by a pair of vectors whose elements

In the future, `play()` will be extended to cubic spline and will allow "continuous" play of a smooth function defined by a `Vector`.

are -65, -15 and 0, 10, respectively. Simulation results using `Vector.play()` with linear interpolation under CVODE are shown in Fig. 4.16.

Error control

An important issue in adaptive integration is selection of appropriate values for local error control. Variable time steps elevate the issue of "physiological accuracy" (see **Error** below) to a level of high concern. Experience so far suggests that control of local absolute error is much more important than control of local relative error. One *can* specify an error criterion based on local relative error, but in neural modeling there is hardly ever a reason to require increasing absolute accuracy around the 0 value of most states, especially voltage.

The scale of states is often a crucial consideration, in that the maximum absolute error must be consistent with the desired resolution of each state. An extreme example is a calcium pump model with pump density measured in [moles/cm^2]. Here an appropriate value is 10^{-14} [mole/cm^2], and an allowable error of 0.01 is clearly nonsense. For this reason, it is essential that each state that is badly scaled, e.g. $[Ca^{2+}]_i$ measured in [mM], be given its own explicit maximum absolute error. NEURON accommodates this need by allowing the user to set specific error criteria for individual states that take precedence over any global criterion.

NEURON's default error setting for CVODE is 10 μV for membrane potential and 0.1 nM for internal free calcium concentration, so that a simulation of the classical

Hodgkin-Huxley action potential at 6.3° C has accuracy comparable to a second order correct simulation with fixed Δt = 25 μs.

Local variable time step method

NEURON provides a network connection (`NetCon`) class for network simulations in which cell to cell communication can be abstractly represented by the (possibly delayed) delivery of logical events, as opposed to graded interaction via gap junctions or electrical synapses (see **Chapter 10**). The notion of a cell driven by discrete input events naturally suggests an expansion of the simulation domain wherein variable time step methods provide substantial performance gains.

It may happen that only a few cells in a network are active at any one time, but with a global time step these active cells govern the time step for all (Fig. 4.17). NEURON's local variable time step method merely uses a separate CVODE solver instance for each cell, thus integrating that cell's states with time steps governed only by those state dynamics and the discrete input events.

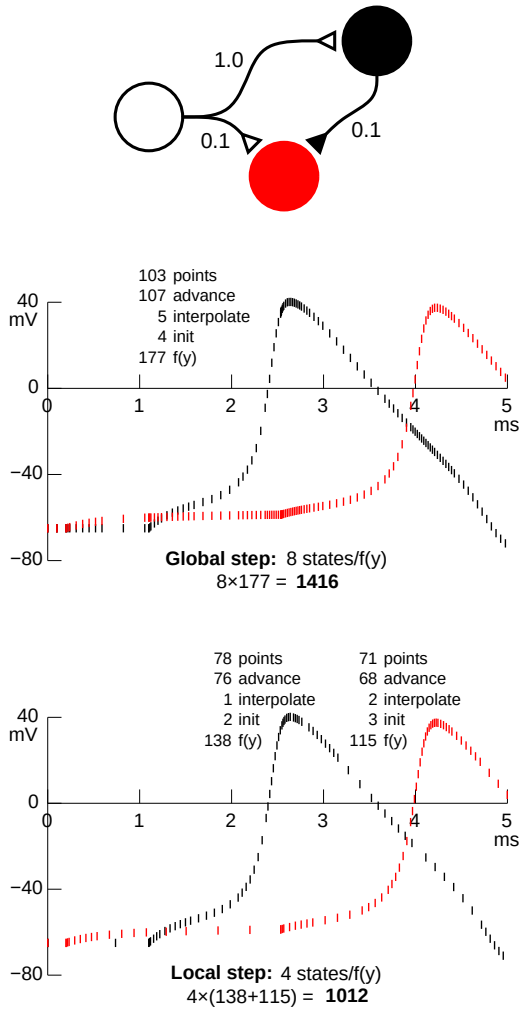


Figure 4.17. Integration with local variable time steps can significantly improve computational efficiency. The top figure shows a simple feedforward network implemented with a `NetStim` artificial spiking cell (white) and a pair of single compartment biophysical model neurons with Hodgkin-Huxley membrane (black and red). All synapses are excitatory, with latencies between presynaptic spike and postsynaptic conductance change shown in ms. The white cell produces a single spike at t = 0 ms. This triggers a spike in the black cell, but the red cell requires inputs from both synapses to make it fire. The short vertical lines in the middle and bottom figures mark the times at which solutions are computed using the global (middle) and local (bottom) variable time step methods. Note that, if rapid changes occur in any cell (e.g. onset of an

epsp, or the upstroke and peak of a spike), the global method forces extra computations in all cells, even those in which nothing much is happening. This does not occur with the local method. The total computational cost of a simulation depends chiefly on the total number of times that new STATEs are calculated. The global method evaluated $\boldsymbol{f}(\mathbf{y})$ (see Eq. 4.29a) 177 times, calculating 8 STATEs each time (4 STATEs per cell), for a total of 1416; the local method required 253 evaluations of $\boldsymbol{f}(\mathbf{y})$, but these were in individual cells so only 4 STATEs were calculated each time, and the local method's total was 1012. Therefore the global method was ~1.4 times more costly than the local method.

All cells are always on a list ordered by their current time and all outstanding events are on a list ordered by their delivery time. These lists are implemented as splay trees to minimize insertion and removal times (proportional to the log of the size of the list), and the least time element can be accessed in constant time. The last fact that prepares our arena for action is that a CVODE instance can, without integrating equations, retreat from its current time to any time back to the beginning of its previous time step.

The network simulation advances in time by checking the cell and event lists to find the least time cell or event, whichever is first. If a cell is first, that cell is integrated according to its current time step, and moved to a location on the cell list appropriate to its new time. If an event is first, it is delivered to the proper cell. That cell retreats to the delivery time and becomes the least time cell, and the event is removed from the list and discarded.

It is easy to devise networks in which the speed improvement of the local time step approach is arbitrarily great. e.g. chains of neurons. However, this method yields no benefit in periods of synchronous activity. If events are extremely numerous, neither the local nor the global variable time step method improves simulation speed. When multiple

events per reasonable Δt arrive regularly, fixed time step integration nicely aggregates all events in a step without regard to their temporal microstructure, whereas variable step methods' scrupulous handling of each event is out of all proportion to the conceptual approximation of the network.

The choice of methods is thus dependent on the problem and the user's intent. To encourage the exploration that is necessary to determine which method may be best suited for a particular application, NEURON allows any of its fixed or variable time step methods to be used with no changes to the user-level specification of the problem.

The local variable time step method considerably increases the complexity of the underlying communication between interpreter and solver with respect to recording results. With a global time step, whether fixed or variable, the `fadvance()` function (see **Chapter 7**) has a clear and precise meaning, i.e. the exit time differs from the entry time by the interval Δt. The problem is that, with the local variable time step, each cell has its own time stream, so each recorded variable must be mapped to the appropriate time stream. This problem is solved by the `CVode` class's `record()`, which records both a variable and its associated times into a pair of `Vectors`.

Discrete event simulations

One limiting case of the variable step simulation style is the "event-driven" or discrete event simulation, in which cells jump from event to event. Here a single compartment is used merely as a stage in which the voltage never changes (the natural time step is infinite), and the "cells" are represented by point processes that receive

events from, and produce events to, the `NetCon` instances. A large variety of useful artificial spiking cells (e.g. integrate and fire, firing frequency dependent on input), as well as mechanisms of use-dependent synaptic plasticity, are susceptible to discrete event simulation because their equations can be solved analytically, so that "cell" state needs only to be computed at the event. This topic is discussed more thoroughly in **Chapter 10**.

Error

The total or global error in a simulation is a combination of errors from two sources. The local error emerges from the extrapolation process within a time step. For the backward Euler method this is easily analyzed with Taylor's series truncated at the term proportional to Δt.

$$V(t+\Delta t) = V(t) + V'(t+\Delta t)\,\Delta t - V''(t^*)\,\frac{\Delta t^2}{2} \qquad \text{Eq. 4.31}$$

where $t \le t^* \le t+\Delta t$.

The forward and backward Euler methods both ignore second and higher order terms, so the error at each step is proportional to Δt^2. Integrating over a time interval T requires $T/\Delta t$ steps, so the error that accumulates in this interval is on the order of $\Delta t^2 T/\Delta t$, i.e. the local error for the Euler methods is proportional to Δt. Applying a similar analysis to the Crank-Nicholson method finds that its local error is proportional to Δt^2. Therefore we can

always decrease the local error of these fixed step methods as much as we like by reducing Δt.

The second contribution to total error comes from the cumulative effect of past errors, which have moved the computed solution away from the trajectory of the analytic solution. Thus, if our computer solution has a nonzero total error at time t_1, then even if we could solve the equations *exactly* from that time forward using the state values at t_1 as our initial condition, the future solution will be inaccurate because we are on a different trajectory. This means that the second component of total error depends on the dynamics of the system itself.

The total error of a simulation is therefore not easy to analyze. For the one and two compartment models we have examined in this chapter, all trajectories end up at the same steady state, so total error tends to decrease with time, but not all systems behave like this. Particularly treacherous are systems with chaotic behavior, in which, once the computed solution diverges even slightly from the proper trajectory, it subsequently moves rapidly away from the original and the time evolution becomes totally different.

Chaos is not the only circumstance that may produce high sensitivity to numerical error. Consider the Hodgkin-Huxley membrane action potentials elicited by two current stimuli, one near threshold and the other twice as strong. The left panel of Fig. 4.18 shows action potentials computed with the backward Euler method using time steps of 25 and 5 μs, the Crank-Nicholson method using Δt = 25 μs, and CVODE using local absolute error tolerance = 0.01. For the strong stimulus, all three integration methods

produced nearly identical results. However, the backward Euler method displayed a noticeable error when the 25 μs time step was used to compute the response to the weak stimulus (dashed line). The weak stimulus allowed membrane potential to hover near spike threshold, so that a small error due to the time step could grow into a large error in the time of occurrence of the action potential. The error was much smaller in the simulation computed with Δt = 5 μs.

However, behavior near threshold is highly sensitive to almost any factor, be it a parameter of the numerical integration method (e.g. Δt or Δx) or a parameter of the model itself. This is seen in the right panel of Fig. 4.18, where all solutions were computed with CVODE (local absolute error tolerance = 0.01) and the sodium channel density $\bar{g}_{Na}$ was varied by only 1%. This small variation of $\bar{g}_{Na}$ did almost nothing to the response to the strong stimulus, but its effect on the latency of the spike elicited by the weak stimulus was comparable to the integration error of the backward Euler method with Δt = 25 μs. This demonstrates that it is important to know the sensitivity of results to every model parameter, and Δt is just one more parameter that is added as a condition of being able to run simulations on a digital computer.

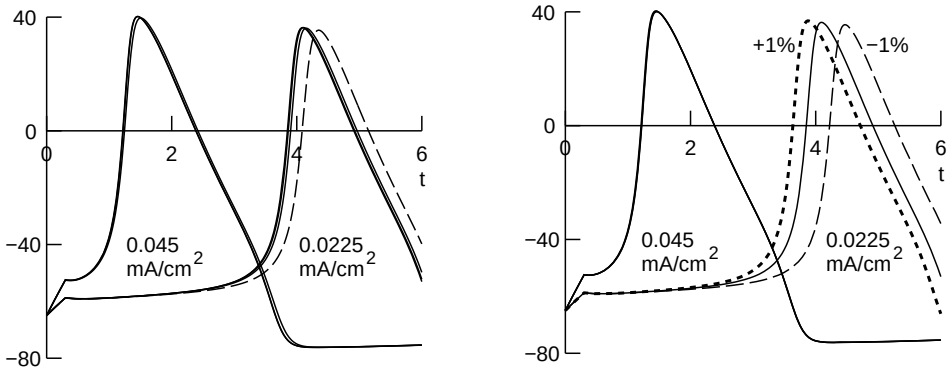


Figure 4.18. Simulations of Hodgkin-Huxley membrane action potentials elicited by 0.3 ms current stimuli with amplitude of 0.0225 or 0.045 mA/cm^2. Left: Sensitivity to integration time step. For each stimulus amplitude, responses were computed using CVODE (local absolute error tolerance = 0.01), Crank-Nicholson (Δt = 25 μs), and backward Euler (Δt = 25 and 5 μs). The backward Euler solution with 25 μs time step showed a noticeable error. Right: Sensitivity to variation in $\bar{g}_{Na}$. All traces were computed with CVODE (local absolute error tolerance = 0.01). Peak sodium conductance was 0.12 S/cm^2 (solid lines) ± 1% (dotted and dashed lines). The three traces elicited with the large stimulus are indistinguishable in this graph.

Using extremely small Δt might seem to be the best way to reduce error. However, computers represent real numbers as floating point numbers with a fixed number of digits, so if you keep adding 10^{-20} to 1 you may always get a value of 1, even after repeating the process 10^{20} times. Operations that involve the difference of similar numbers, as when differences are substituted for derivatives, are especially prone to such roundoff error. Consequently there is a limit to the accuracy improvement that can be achieved by decreasing Δt.

Generally speaking, it would be desirable to use what might be called "physiological" values of Δt, i.e. time steps that give a good representation of the state trajectories without having a numerical accuracy that is orders of magnitude better than the accuracy of our physiological measurements (which is generally not as good as 5%, and seldom better). The question is not so much how large the error of a simulation is relative to the analytic solution, but whether the simulation error leads us to trajectories that are significantly different from the set of trajectories defined by the error in our parameters. Insofar as removal of any source of error has value, there is a temptation to treat the model equations as sacred runes which must be solved to an arbitrarily high precision. Nevertheless, determining the meaning of a simulation run requires judgment. A misplaced emphasis on numerical accuracy should not obscure the fact that qualitative results may be quite sufficient. We agree with John Moore, our mentor and colleague, who is fond of quoting R. Hamming: "The purpose of computing is insight, not numbers" (Hamming 1987).

Summary of NEURON's integration methods

NEURON offers the user a choice of several different integration methods. For any particular problem, the best way to determine which is the method of choice is to run comparison simulations with several values of Δt or local error tolerance to see which executes most quickly while achieving the desired accuracy. In performing such trials, one must remember that the stability properties of a simulation depend on the entire system that is being modeled. Because of interactions between "biological" components

and any "nonbiological" elements, such as stimulators or voltage clamps, the time constants of the *entire* system may be different from those of the biological components alone. A current source (perfect current clamp) does not affect stability because it does not change the time constants. Any other signal source imposes a load on the compartment to which it is attached, changing time constants and potentially introducing troublesome stiffness. The more closely a signal source approximates a voltage source (perfect voltage clamp), the greater this effect will be.

Fixed time step integrators

Implicit integrators are used as NEURON's fixed time step methods. This is in part because of their superior stability compared to explicit integrators (Dahlquist and Bjorck 1974).

Default: backward Euler

NEURON's default integration method is backward Euler, a fixed step first order implicit scheme that produces good qualitative results with large time steps when extremely stiff ODEs and even algebraic equations are present in the system, e.g. models that involve voltage clamps. Because of its robust stability, it can be used with extremely large time steps to find the steady state solution for a linear ("passive") system.

Crank-Nicholson

When the global parameter `secondorder` is set to 2, NEURON uses a variant of the Crank-Nicholson method. This has local error proportional to Δt^2 and is therefore particularly accurate for small time steps.

In implicit integration methods, all current balance equations must be solved simultaneously. The backward Euler algorithm does not resort to iteration to deal with nonlinearities, since its numerical error is proportional to Δt anyway. The special feature of the Crank-Nicholson variant is its use of a staggered time step algorithm to avoid iteration of nonlinear equations (see **Efficiency** in the section **Crank-Nicholson: stable and more accurate** above). This converts the current balance part of the problem to one that requires only the solution of simultaneous *linear* equations, making the computational cost per time step almost identical to the backward Euler method.

The second order fixed time step method works with HH-type Ohm's law channels, but its accuracy is really only first order when the instantaneous current-voltage relation of the channels is nonlinear or when channel gating models are expressed with kinetic schemes (the `SOLVE scheme METHOD sparse` statement in NMODL solves kinetic schemes using the fully implicit method). Accuracy is also formally first order for models involving changing ion concentration, though that is a negligible issue when `dt` is small enough to accurately follow voltage changes.

Although the Crank-Nicholson method is formally stable, models with stiff equations require small Δt to avoid numerical oscillations (Fig. 4.8). It is unusable in the presence

of voltage clamps, extracellular mechanisms, or linear circuits, since the solution of algebraic equations gives results with large numerical oscillations.

Adaptive integrators

NEURON's adaptive integrators free the user from having to choose an integration step size. Instead, they automatically adjust integration order and Δt so that the solution satisfies a user-specified error criterion. While this may be the most salient feature of these methods, there are several reasons why they may be preferable to fixed step integrators:

- Adaptive integrators usually require less time for a given degree of accuracy.
- They avoid the problem of "empty temporal resolution" (many solution points when nothing is happening) that occurs with fixed time step integration.
- Currents, voltages, and conductances are all known to the same accuracy at the same time, unlike the staggered Crank-Nicholson method.
- Events occur at their actual times instead of being constrained to multiples of Δt. For example, with fixed time steps, current step discontinuities are only first order correct unless they are defined to lie on time step boundaries. Precise timing may be particularly important in network simulations.

Switching between fixed and variable time step methods is as easy as a button press (**NEURON Main Menu / Tools / VariableStepControl / Use variable dt**) and does not affect any GUI tools. Plots of expressions vs. time still look the same, and `Vector` recording of temporal streams still works. There is no need to change model descriptions,

or at least to change the statements that define the equations. Ease of switching is crucial since relative performance between high overhead variable step and low overhead fixed step methods ranges widely. For example, simulation of the demonstration models by Mainen and Sejnowski (1996) slowed down by a factor of 2 or sped up by a factor of 7, depending on number of spikes in a simulation run and whether there were long intervals in which *no* state changed rapidly.

CVODE

CVODE handles any kind of model description involving `DERIVATIVE` or `KINETIC` representations of gating states, ion accumulation/diffusion, or nonlinear current-voltage relations. It does not work with models that involve extracellular mechanisms, linear circuits, perfect voltage clamps, or capacitors between nodes. Each cell in a network simulation may have its own local time step, but time steps must be global if there are gap junctions between different cells. Cell mechanisms that have analytical solutions (e.g. integrate and fire artificial spiking cells) can be implemented in a way that allows discrete event simulations.

DASPK

The DASPK method is suitable for models that involve extracellular mechanisms, linear circuits, perfect voltage clamps, or capacitors between nodes. However, there is no local variable step variant of DASPK.

References

Brown, P.N., Hindmarsh, A.C., and Petzold, L.R. Using Krylov methods in the solution of large-scale differential-algebraic systems. *SIAM Journal of Scientific Computing* 15:1467-1488, 1994.

Cohen, S.D. and Hindmarsh, A.C. CVODE User Guide. Livermore, CA: Lawrence Livermore National Laboratory, 1994.

Cohen, S.D. and Hindmarsh, A.C. CVODE, a stiff/nonstiff ODE solver in C. *Computers in Physics* 10:138-143, 1996.

Crank, J. *The Mathematics of Diffusion*. 2 ed. London: Oxford University Press, 1979.

Crank, J. and Nicholson, P. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. *Proceedings of the Cambridge Philosophical Society* 43:50-67, 1947.

Dahlquist, G. and Bjorck, A. *Numerical Methods*. Englewood Cliffs, New Jersey: Prentice-Hall, 1974.

Hamming, R.W. *Numerical Methods for Scientists and Engineers*. 2 ed: Dover Publications, 1987.

Hindmarsh, A.C. and Serban, R. User documentation for CVODES, an ODE solver with sensitivity analysis capabilities: Lawrence Livermore National Laboratory, 2002.

Hindmarsh, A.C. and Taylor, A.G. User documentation for IDA, a differential-algebraic equation solver for sequential and parallel computers: Lawrence Livermore National Laboratory, 1999.

Hines, M. Efficient computation of branched nerve equations. *Int. J. Bio-Med. Comput.* 15:69-76, 1984.

Hines, M.L. and Carnevale, N.T. The NEURON simulation environment. *Neural Computation* 9:1179-1209, 1997.

Hines, M.L. and Carnevale, N.T. NEURON: a tool for neuroscientists. *The Neuroscientist* 7:123-135, 2001.

Kundert, K. Sparse matrix techniques. In: *Circuit Analysis, Simulation and Design*, edited by A. Ruehli: North-Holland, 1986.

Mainen, Z.F. and Sejnowski, T.J. Influence of dendritic structure on firing pattern in model neocortical neurons. *Nature* 382:363-366, 1996.

Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P. *Numerical Recipes in C*. 2 ed. Cambridge: Cambridge University Press, 1992.

Stewart, D. and Leyk, Z. *Meschach: Matrix Computations in C*. Proceedings of the Centre for Mathematics and its Applications. Vol. 32. Canberra, Australia: School of Mathematical Sciences, Australian National University, 1994.

Strang, G. *Introduction to Applied Mathematics*. Wellesley, MA: Wellesley-Cambridge Press, 1986.

Chapter 4 Index

V

Chapter 5

Representing neurons with a digital computer

Information processing in the nervous system involves the spread and interaction of electrical and chemical signals within and between neurons and glia. From the perspective of the experimentalist working at the level of cells and networks, these signals are continuous variables. They are described by the diffusion equation and the closely-related cable equation (Crank 1979; Rall 1977), in which potential (voltage, concentration) and flux (current, movement of solute) are smooth functions of time and space. But everything in a digital computer is inherently discontinuous: memory addresses, data, and instructions are all specified in terms of finite sequences of 0s and 1s, and there are finite limits on the precision with which numbers can be represented. Thus there is no direct parallel between the continuous world of biology and what exists in digital computers, so special effort is required to implement digital computer models of biological neural systems. The aim of this chapter is to show how the NEURON simulation environment makes it easier to bridge this gap.

Discretization

To simulate the operation of biological neurons, NEURON uses the tactic of discretizing time and space, which means approximating these partial differential

equations by a set of algebraic difference equations that can be solved numerically (numerical integration; see **Chapter 4: Essentials of numerical methods for neural modeling**). Indeed, spatial discretization, in one form or another, lies at the core of all simulators used to model biological neurons.

Discretization is often couched in terms of "compartmentalization," i.e. approximating the cable equation by a series of compartments connected by resistors (see **Chapter 4** and **Cables** in **Chapter 3**). However, it is more insightful to regard discretization as an approximation of the original continuous system by another system that is discontinuous in time and space. Viewed in this way, simulating a discretized model amounts to computing the values of spatiotemporally continuous variables over a set of discrete points in space (a "grid" of "nodes") for a finite number of instants in time. The size of the time step and the fineness of the spatial grid jointly determine the accuracy of the solution, and may also affect its stability. How faithfully a computed solution emulates the behavior of the continuous system depends on the spatial intervals between adjacent nodes, and the temporal intervals between solution times. These should be small enough that the discontinuous variables in the discretized model can approximate the curvature in space and time of the continuous variables in the original physical system.

Choosing an appropriate discretization is a recurring practical problem in neural modeling. The accuracy required of a discrete approximation to a continuous system, and the effort needed to compute it, depend on the anatomical and biophysical complexity of the original system and the question that is being asked. Thus finding the resting membrane potential of an isopotential model with passive membrane may require only a

few large time steps at one point in space, but determining the time course of V_m throughout a highly branched model with active membrane as it fires a burst of spikes demands much finer spatiotemporal resolution; furthermore, selecting Δx and Δt for complex models can be especially difficult.

Although the time scale of biophysical processes may suggest a natural Δt, it is usually not clear at the outset how fine the spatial grid should be. Both the accuracy of the approximation and the computation time increase as the number of nodes used to represent a cable increases. A single node is usually adequate to represent a short cable in its entirety, but a large number of closely spaced nodes may be necessary for long cables or highly branched structures. Also, as we intimated above, the choice of a spatial grid is closely related to the choice of the integration time step, especially with NEURON's Crank-Nicholson (second order) integrator, which can produce spurious oscillations if the time step is too long for the spatial grid (see **Chapter 4**).

Over the years, a certain amount of folklore and numerous unreliable rules of thumb have emerged concerning the topic of "compartment size." Among the topics we cover in this chapter are a practical method for quickly testing spatial accuracy, and a rational basis for specifying the spatial grid that makes use of the AC length constant at high frequencies (Hines and Carnevale 2001).

No less important is the practical question of how to manage all the parameters that exist throughout a model. Returning briefly to the metaphor of "compartments," let us consider membrane capacitance, a parameter that has a different value in each compartment. Rather than specify the *capacitance* of each compartment individually, it is

better to deal in terms of a single *specific membrane capacitance* that is constant over the entire cell, and have the program compute the values of the individual capacitances from the areas of the compartments. Other parameters such as diameter or channel density may vary widely over short distances, so the granularity of their representation may have little to do with numerically adequate discretization.

How NEURON separates anatomy and biophysics from purely numerical issues

Thinking in terms of compartments leads to model implementations that require users to keep track of the correspondence between compartments and anatomical locations. If we change the size or number of compartments, e.g. to see whether spatial discretization is adequate for numerical accuracy, we must also abandon the old mapping between compartments and locations in favor of a completely new one.

So even though NEURON is a compartmental modeling program, it has been designed to separate the specification of biological properties (neuron shape and physiology) from computational issues such as the number and size of compartments. This makes it easy to trade off between accuracy and speed, and enables convenient verification of the numerical correctness of simulations. It also shields users from numerical details, so they can focus on matters that are biologically relevant.

NEURON accomplishes this by employing four related concepts: sections, range, range variables, and segments. These concepts are defined in the following paragraphs, and discussed later in this chapter under the heading **How to specify model properties**.

Sections and section variables

A *section* is a continuous length of unbranched cable with its own anatomical and biophysical properties. Each section in a model can be the direct counterpart of a neurite in the original cell. This reduces the difficulty of managing anatomically detailed models, because neuroscientists naturally tend to think in terms of axonal and dendritic branches rather than compartments.

Figure 5.1 illustrates how a cell might be mapped into sections. The cartoon at the top shows how an anatomist might regard this cell: the soma gives rise to a branched dendritic tree and an axon hillock which is connected to a myelinated axon. The bottom of Fig. 5.1 shows how to break this cell into sections in order to build a NEURON model. Notice that each biologically significant anatomical structure corresponds to one or more sections of the model: the cell body (*Soma*), axon hillock (*AH*), myelinated internodes (I_i), nodes of Ranvier (N_i), and dendrites (D_i). Sections allow this kind of functional/anatomical parcellation of a cell to remain foremost in the mind of the person who constructs and uses a NEURON model.

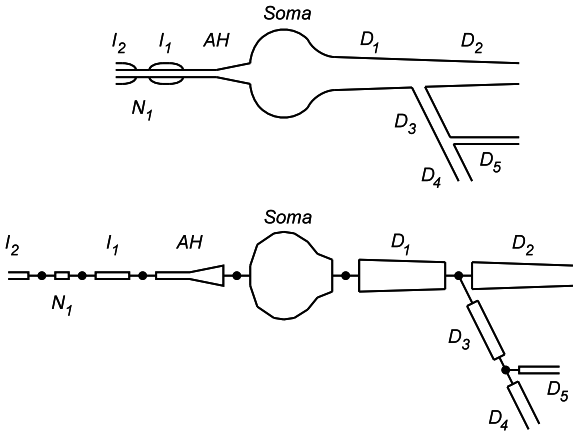


Figure 5.1. Top: Cartoon of a neuron indicating biologically significant structures. Bottom: How these structures are represented by sections in a NEURON model. Reproduced from (Hines and Carnevale 1997).

Certain properties apply to a section as a whole. These properties, which are sometimes called *section variables*, are length `L`, cytoplasmic resistivity `Ra`, and the discretization parameter `nseg` (see Table 5.1 and following section).

Table 5.1. Section variables

Name	Meaning	Units
`L`	section length	[μm]
`Ra`	cytoplasmic resistivity	[Ω cm]
`nseg`	discretization parameter	[1], i.e. dimensionless

Range and range variables

Many variables in real neurons are continuous functions of position throughout the cell. In NEURON these are called *range variables* (see Table 5.2 for examples). While each section is ultimately discretized into compartments, range variables are specified in terms of a continuous parameter: normalized distance along the centroid of each section.

This normalized distance, which is called *range* or *arc length*, varies linearly from 0 at one end of the section to 1 at the other. Figure 5.2 depicts the correspondence between the physical distance of a point along the length of a section and its location in units of normalized distance.

Table 5.2. Some examples of range variables

Name	Meaning	Units
`diam`	diameter	[μm]
`cm`	specific membrane capacitance	[μf/cm^2]
`v`	membrane potential	[mV]
`ina`	sodium current	[mA/cm^2]
`nai`	internal sodium concentration	[mM]
`n_hh`	Hodgkin-Huxley potassium conductance gating variable	[1], i.e. dimensionless

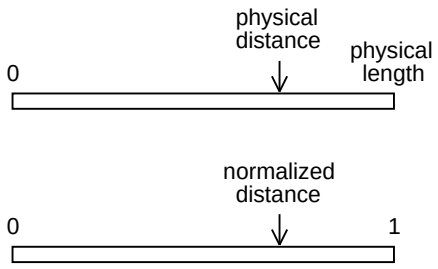


Figure 5.2. Top: The arrow indicates the location of a point at a particular physical distance from one end of a section. Bottom: In NEURON, this location is expressed in terms of normalized distance ("range") along the length of the section.

One way to access the values of range variables and other section properties is by dot notation, which specifies the name of the section, the name of the variable, and the location of interest. Thus

```
soma.diam(0) = 10
```


sets the diameter closest to the 0 end of the `soma` section to 10 μm, and

```
dend.v(0.5)
```

returns the membrane potential at the middle of the `dend` section. Note that the value returned by `sectionname.rangevar(x)` is the value at the center of the segment (see below) that contains `x`, *not* the linear interpolation of the values associated with the centers of adjacent segments. If parentheses are omitted, the position defaults to 0.5 (middle of the section), i.e. `dend.v(0.5)` and `dend.v` both refer to membrane potential at the midpoint of `dend`.

Range variables and related topics are covered more thorougly below in **How to specify model properties**.

Segments

As already mentioned, NEURON computes membrane current and potential at one or more discrete positions ("nodes") that are equally spaced along the *interior* of a section. In addition to these internal nodes, there are terminal nodes at the 0 and 1 ends. However, no membrane properties are associated with terminal nodes so the voltage at the 0 and 1 locations is defined by a simple algebraic equation (the weighted average of the potential at adjacent internal nodes) rather than an ordinary differential equation. Each section has a parameter `nseg` that controls the number of internal nodes. These nodes are located at arc length = (2 `i` - 1) / 2 `nseg` where `i` is an integer in the range [1, `nseg`] (Fig. 5.3).

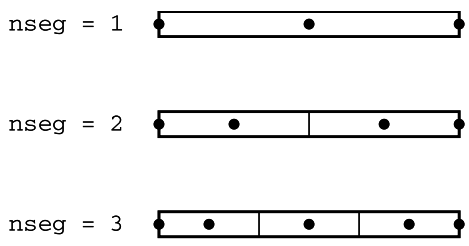


Figure 5.3. Each section has a discretization parameter nseg that governs the number of internal nodes (black dots inside the section) at which membrane potential is computed. The thin lines mark conceptual boundaries between adjacent segments.

You can think of a section as being broken into nseg segments of equal length, which are conceptually demarcated by evenly spaced boundaries at intervals of 1 / nseg, so that each segment has one node at its midpoint. This internal node is the point at which the voltage of the segment is defined. The transmembrane currents over the entire surface area of a segment are associated with its node. Nodes of adjacent segments are connected by resistors that represent the resistance of the intervening cytoplasm (Fig. 5.7).

Each section in a model can have a different value for nseg. One way to specify this value is with dot notation, e.g.

```
axon.nseg = 3
```

ensures that membrane current and potential will be computed at three points along the length of the section called axon. The value to choose for nseg depends on the degree of spatial accuracy and resolution that is desired: larger values of nseg mean more nodes spaced at shorter intervals, so that the piecewise linear approximation in space becomes more accurate and smoother. Strategies for selecting appropriate values of nseg are discussed later in this chapter under **Discretization guidelines**.

Implications and applications of this strategy

Range, range variables, and `nseg` free the user from having to keep track of the correspondence between segment number and position along each branch of a model. This avoids the tendency of compartmental modeling approaches to confound representation of the physical properties of neurons, which are biologically relevant, with implementational details such as compartment size, which are mere artifacts of having to use a digital computer to emulate the behavior of a distributed physical system that is continuous in time and space.

For a concrete example of the complications that can arise in a compartment-oriented simulation environment, suppose the axon shown in Fig. 5.4 is 1000 μm long and we are particularly interested in the membrane potential at a point 700 μm from its left end. If our model has 5 compartments numbered 0 to 4, then we want to know the membrane potential in compartment 3, but if there are 25 compartments, it is compartment 17 that deserves our attention. It is easy to see that dealing with highly branched models can be quite confusing. But in NEURON, the membrane potential of interest is simply called `axon.v(0.7)`, regardless of the value of `axon`'s discretization parameter `nseg`.

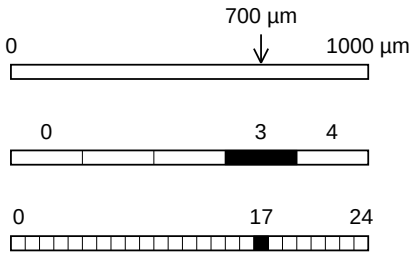


Figure 5.4. Boxed in by compartments. Top: Conceptual model of an unbranched axon 1000 µm long. We are interested in membrane potential at a point 700 µm from its left end. Middle and bottom: The index of the compartment that corresponds to the location of interest depends on how many compartments there are.

Spatial accuracy

As we mentioned in **Chapter 4**, the spatial discretization method employed by NEURON produces solutions that are second order correct in space, i.e. spatial error within a section is proportional to the square of its segment length. It is crucial to realize that the location of the second order correct voltage is not at the edge of a segment but rather at its *center*, i.e. at its node (Fig. 5.3; also see **Spatial discretization** in **Chapter 4**). This has several important consequences.

- To allow branching and injection of current at the precise ends of a section while maintaining second order correctness, extra voltage nodes that represent compartments with 0 area are defined at the section ends. It is possible to achieve second order accuracy with sections whose end nodes have nonzero area compartments, but the areas of these terminal compartments would have to be exactly half that of the internal

compartments, and extra complexity would be imposed on administration of channel density at branch points.

- To preserve second order accuracy, localized current sources (e.g. synapses, current clamps, voltage clamps--see **Point processes** below) must be placed at nodes. For the same reason, all sections should be connected at nodes.
- If `nseg` is even, `dend.v(0.5)` and `dend.v` will return a value that actually comes from "the nearest internal node" which is not at the middle of `dend` but instead depends on roundoff error. Using odd values for `nseg` avoids such capricious outcomes by ensuring that there will be a node at the midpoint of each section.
- Second order spatial accuracy means that the results of a NEURON simulation are a piecewise linear approximation to the continuous system. Therefore second order accurate estimates of continuous variables at intermediate locations in space can be found by linear interpolation between nodes.

A practical test of spatial accuracy

A convenient way to test the spatial accuracy of a model is to start by running a "control" simulation with the current resolution that will serve as a basis for comparison. Then execute the command

```
forall nseg *= 3
```

which increases spatial resolution by a factor of 3 throughout the model and reduces spatial error terms by a factor of 9. Now run a "test" simulation and see if a significant

qualitative or quantitative change has occurred. The absence of a significant change is evidence that the control simulation was sufficiently accurate in space.

Why triple nseg instead of just doubling it? Because NEURON uses a piecewise linear approximation to emulate the continuous variation of membrane current and voltage in space. The breakpoints in this piecewise linear approximation are located at the internal nodes of each section. Multiplying nseg by an even number will shift these breakpoints to new locations, making it hard to compare the results of the control and test simulations. For instance, with nseg = 1, voltage is computed at arc length = 0.5, but with nseg = 2 it is computed at arc length = 0.25 and 0.75 (see Fig. 5.3). If simulations with nseg = 1 and nseg = 2 did produce different results, it could be difficult to know whether this reflects improved spatial accuracy or is just due to the fact that the two simulations computed solutions at different points in space. Tripling nseg adds new breakpoints (at arc length = 1/6 and 5/6 in Fig. 5.3) without changing the locations of any that were already there (at 0.5 in this case). Any odd multiple could be used, but 3 is a practical value since it reduces spatial error by almost an order of magnitude, which is probably enough to detect inadequate spatial accuracy.

While repeatedly tripling nseg throughout an entire model is certainly a convenient and effective method for testing the spatial grid, this is generally not a good way to achieve computational efficiency, especially if geometry is complex and biophysical properties are nonuniform. Models based on quantitative morphometric data often have several branches that need nseg ≥ 9, while many other branches require only 1 or 3 nodes. By the time the spatial grid is just adequate in the former, it will be much finer

than necessary in the latter, increasing storage and prolonging run time. We return to this problem at the end of this chapter in the section **Choosing a spatial grid**.

How to specify model properties

In **Chapter 1** we used the CellBuilder to implement a computational model of a particular conceptual model. First we specified the topology (branched architecture) of the computational model, then its geometry (physical dimensions), and finally its biophysical properties. This is also a good sequence to follow when implementing a computational model by writing `hoc` code, and we will examine each of these steps in turn. However, at some points it will be necessary to address syntactic details. The first syntactic detail has to do with "the currently accessed section," an idea so fundamental that we must consider it before proceeding to topology.

Which section do we mean?

Most of our attention in the following paragraphs will be devoted to sections. We will see how to create sections, assemble them into a model with the desired topology, and specify their geometric and biophysical attributes. Because sections share property names (e.g. length `L`, diameter `diam`), it is always necessary to specify which section is being discussed. This is called the currently accessed section.

NEURON offers three ways to specify the currently accessed section, each being compact in some contexts and cumbersome in others: dot notation, section stack, and

default section. We consider them in order of precedence, starting with the method that has highest priority.

1. **Dot notation**

   Syntax *sectionname.variablename*

   Examples

   ```
   dendrite[2].L = dendrite[1].L + dendrite[0].L
   axon.v = soma.v
   print soma.gnabar
   axon.nseg = 3*axon.nseg
   ```

   Comments

   - This takes precedence over the other methods
   - Dot notation is necessary in order to refer to more than one section within a single statement

2. **Section stack**

   Syntax *sectionname* { *stmt* }

   where *stmt* is one or more statements. *sectionname* becomes the currently selected section during execution of *stmt*. Afterwards, the currently selected section reverts to whatever it was before *sectionname* was seen.

   Comments

   - This is the most useful method for programming, since the user has explicit control over the scope of the section and can set several range variables.

- Nesting is allowed to any level, i.e.

```
sectionname1 {
   stmt1
   sectionname2 {
      stmt2
      sectionname3 {
         etc.
      }
   }
}
```

- Avoid the error

```
soma L=10 diam=10
```

  (i.e. missing curly brackets), which sets `soma.L`, then pops the section stack and sets `diam` for whatever section is then on the stack.

- Control flow should reach the end of *`stmt`* in order to automatically pop the section stack. Therefore *`stmt`* should not include the `continue`, `break`, or `return` statement.

- A section cannot be used as a variable for assignment or passing as an argument to a function or procedure. However, the same effect can be obtained with the `SectionRef` class, which allows sections to be referenced by normal object variables. The use of `push_section()` for this purpose should be avoided except as a last resort.

- Looping over sets of sections is most often done with the `forall` and `forsec` commands.

3. **Default section**

Syntax `access` *`sectionname`*

defines a default section that will be the currently selected section when the first two methods are not in effect. If a model has a conceptually privileged section that gets most of the use, it is best to declare it as the default section, e.g.

```
access soma
```

Having done this, one can determine the values of voltage and other variables by a minimum of typing at the interpreter's `oc>` prompt. Thus after `soma` is declared to be the default section,

```
print v, ina, gk_hh
```

will print out the membrane potential, sodium current, and Hodgkin-Huxley potassium conductance at `soma(0.5)`.

Comments

- Dot notation and stack of sections both take precedence over this method.
- The `access` statement should only be used once in a program. The

  `sectionname` `{` *`stmt`* `}`

  form is almost always the right way to specify the current section.

How to set up model topology

In the NEURON simulation environment, the branched topology of a model cell is constructed by

A tree has the property that any two points on it are connected by a unique path.

creating sections and attaching them to each other in the form of a tree. Sections are created with hoc statements of the form

```
create sectionname
```

They can be attached to each other with the syntax

```
connect child(0 or 1), parent(x)
```

which connects the 0 or 1 end of *child* to location x on *parent*. The alternative syntax

```
connect child(0 or 1), x
```

attaches *child* to location x on the currently accessed section.

Loops of sections

A model of a cell cannot contain a loop of sections. If a sequence of connect statements produces a loop of sections, an error is generated when the internal data structures are created, and NEURON's interpreter will require that the loop be broken by disconnecting one of the sections in the loop. Tight electrical loops can be implemented with the LinearMechanism class.

Loops that involve sections *are* allowed if at least one element in the loop is a membrane mechanism, e.g. a gap junction. For the sake of stability it may be preferable to use the the LinearMechanism class to set up this kind of nonlocal coupling between system equations. Gap junctions can also be implemented with mechanisms that use POINTER variables, but this may cause spurious oscillations if coupling is tight (see **Example 10.2: a gap junction** in **Chapter 10**).

A section may have only one parent

If an attempt is made to attach a child to more than one parent, a notice is printed on the standard error device saying that the section has been reconnected. To avoid the notice, disconnect the section first with the procedure `disconnect()`.

The root section

Each section in a tree has a parent section, except for the root section. The root section is determined implicitly by the fact that we never "connect" it to anything. Any section can be used as a root section, and the identity of the root section has no effect on computational efficiency. The root section and the default section (i.e. the section specified by the `access` statement) are different things and shouldn't be confused with each other. Every model has a root section, and most often this turns out to be something called `soma`, but there is no absolute requirement that a model have a default section. Usually it is most convenient to construct a model in such a way that the root and default sections are the same, but this isn't mandatory.

Attach sections at 0 or 1 for accuracy

Section attachments must be located at nodes to preserve second order spatial accuracy. It is generally best for `x` to be either 0 or 1, rather than an intermediate value. Attempting to connect a section to a non-node location will result in the section actually being connected to the nearest internal node of the parent, which depends on the value of `nseg` and may be quite far from the intended position. Even if a section is connected to an internal node, if `nseg` is then changed, e.g. to test for spatial accuracy, the attachment

could be repositioned to a different site (another reason to increase `nseg` by an odd factor). This would affect the electrotonic architecture of the model, causing spurious changes in simulation results. Therefore the best policy is to connect child sections only to the 0 or the 1 end of the parent, and not to intermediate locations. Because of their small size, dendritic spines are a possible exception to this rule.

Note that sections attached to internal locations will not be repositioned if `nseg` is increased by an odd factor. Nonetheless, the best policy is to attach only to the 0 or 1 end of the parent section.

Checking the tree structure with `topology()`

The `topology()` function prints the tree structure using a kind of "typewriter art." Each section appears on a separate line, starting with the root section. The root section is shown with its 0 and 1 ends at the left and right, respectively, and marked by a `|` (vertical bar). The remaining sections are printed with a `` ` `` (grave) at the end that is attached to the parent, and a `|` at the other end. Each segment in every section is marked by a `-` (hyphen).

For example the statements

```
create soma, dend[3]
soma for i=0,2 {
   connect dend[i](0), 1
}
```

create a section named `soma` and an array of three sections named `dend[0]`, `dend[1]`, and `dend[2]`, and then attaches the 0 end of each `dend` to the 1 end of `soma`. If we now type


```
topology()
```

at the oc> prompt, NEURON's interpreter will print

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

This confirms that soma is the root section of this tree, that the three dend[] sections are attached to its 1 end, and that all sections have one segment.

Viewing topology with a Shape plot

For a graphical display of the topology of our model, we can execute the statements

```
objref s
s = new Shape()
```

to create a Shape plot (Fig. 5.5). The labels in this figure have been added to identify the sections and their orientation. The root section is soma, and the three child branches are dend[0] - dend[3]. Each of the child sections are connected to the 1 end of soma, and all sections are drawn from left (0 end) to right (1 end). If a section were attached to the 0 end of the root section, it would be drawn right to left. The rules that govern the appearance of a model in a Shape plot are further discussed under **3-D specification** below and under **Strange shapes?** in **Chapter 6**.

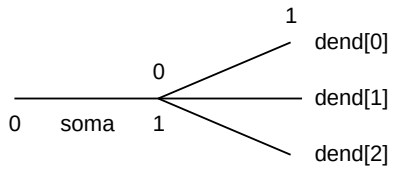


Figure 5.5. A Shape plot display of the topology of a model in which the 0 ends of three child sections are attached to the 1 end of the root section.

How to specify geometry

A newly created section has certain default properties, as we can see by executing

```
oc>create axon
oc>forall psection()
axon { nseg=1  L=100  Ra=35.4
   /*location 0 attached to cell 0*/
   /* First segment only */
   insert morphology { diam=500}
   insert capacitance { cm=1}
}
```

where the units are [μm] for length `L` and diameter `diam`, [Ω cm] for cytoplasmic resistivity `Ra`, and [μf/cm^2] for specific membrane capacitance `cm`. Users will generally want to change these values, except for `cm` and perhaps `Ra`.

Below we discuss the two ways to specify the physical dimensions of a section: the "stylized method" and the "3-D method." Regardless of which method is used, NEURON calculates the values of internal model parameters, such as average diameter, axial resistance, and compartment area, that are assigned to each segment. This calculation takes any nonuniformity of anatomical or biophysical properties into account.

Stylized specification

With the "stylized method" one assigns values directly to section length and diameter with statements like

```
axon { L=1000  diam=1 }
```

This is appropriate if the notions of cable length and diameter are authoritative and three dimensional shape is irrelevant.

Segment surface area `area` and axial resistance `ri` are computed as if the section were a sequence of right circular cylinders of length `L` / `nseg`, whose diameters are given by the `diam` range variable at the center of each segment. Cylinder ends do not contribute to surface area, and segment surface area is very close to the surface area of a truncated cone as long as diameter does not change too much. Abrupt diameter changes should be restricted to section boundaries, for reasons that are explained below (see **Avoiding artifacts**). For plotting purposes, `L` and `diam` are used to automatically generate 3-D information for a stylized straight cylinder.

One fact that is often useful when working with stylized models is that the surface area of a cylinder with length equal to diameter is identical to that of a sphere of the same diameter. Another fact to remember is that, when the surface area of a single compartment model is 100 μm^2, total transmembrane current over the entire surface of the model in [nA] will be numerically equal to the membrane current density in [mA/cm^2]. This implies that the current delivered by a current clamp in [nA] will also be numerically equal to the membrane current density in [mA/cm^2].

3-D specification

The alternative to the stylized method is the 3-D method, in which one specifies a list of (x, y, z) coordinates and corresponding diameters, e.g.

```
dend {
   pt3dadd(10,0,0,5)  // x, y, z, diam
   pt3dadd(16,10,0,3)
   pt3dadd(25,14,-3,2)
}
```

NEURON then computes section length and diameter from these values. The 3-D method is preferable if the model is based on quantitative morphometry, or if visualization is important.

The anatomical data are kept in an internal list of (x, y, z, diam) "points," in which the first point is associated with the end of the section that is connected to the parent--this is not necessarily the 0 end--and the last point is associated with the opposite end. There must be at least two points per section, and they should be ordered in terms of monotonically increasing arc length. This 3-D information, or "pt3d list," is the authoritative definition of the shape of the section and automatically determines section length `L`, segment diameter `diam`, `area`, and `ri`. Properly used, the 3-D method allows substantial control over the appearance of a model in a Shape plot (see **Strange Shapes?** in **Chapter 6**). However, side-effects can occur if geometry was originally specified with the stylized method (see **Avoiding artifacts** below).

To prevent confusion, when using the 3-D method one should generally attach only the 0 end of a child section to a parent. This will ensure that `diam(x)` (segment diameter) as `x` ranges from 0 to 1 has the same sense as `diam3d(i)` (the actual

morphometric diameters) as `i` ranges from 0 to `n3d()`-1 (`n3d()` is the number of (x, y, z, diam) points used to specify the geometry of the section). It can also prevent unexpected distortions of the model appearance in a Shape plot (see **The case of the disappearing section** in **Chapter 6**).

When 3-D specification is used, a section is treated as a sequence of frusta (truncated cones), as in the example shown in Fig. 5.6. The morphometric data for this particular neurite consist of four (x, y, z, diam) measurements (Fig. 5.6 A). These 3-D points define the locations and diameters of the ends of the frusta (Fig. 5.6 B). The length `L` of the section is the sum of the distances from one 3-D point to the next. The effective `diam`, `area`, and axial resistance `ri` of each segment are computed from this sequence of points by trapezoidal integration along the centroid of the segment. This takes into account the extra area introduced by diameter changes; even degenerate cones of 0 length can be specified (i.e. two points with identical coordinates but different diameters), which add surface area but not length to the section. No attempt is made to deal with the effects of centroid curvature on surface area.

The number of 3-D points used to describe the shape of the section has nothing to do with `nseg` and does not affect simulation speed. Thus if we represent the neurite of with a section using `nseg` = 1, the entire section will have only one node, and that node will be located midway along its length (`x` = 0.5 in Fig. 5.6 C). The membrane properties associated with this node are computed by integrating over the entire surface area of the section ($0 \le x \le 1$). The values of the axial resistors to either side of the node are determined by integrating the cytoplasmic resistivity along the paths from the 0 and 1

ends of the section to its midpoint (dashed line in Fig. 5.6 C). Thus the left and right hand axial resistances of Fig. 5.6 D are evaluated over the x intervals [0, 0.5] and [0.5, 1], respectively.

Figure 5.7 shows what happens when nseg = 2. Now the section is broken into two segments of equal length that correspond to x intervals [0, 0.5] and [0.5, 1]. The membrane properties over these intervals are attached to the nodes at 0.25 and 0.75, respectively. The three axial resistors Ri_1, Ri_2, and Ri_3 are determined by integrating the path resistance over the x intervals [0, 0.25], [0.25, 0.75], and [0.75, 1].

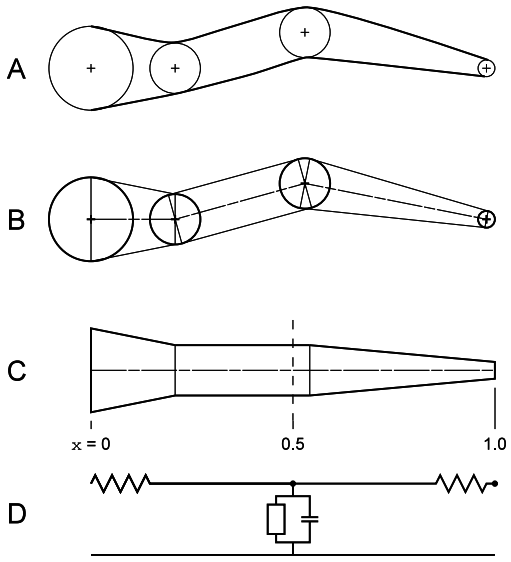


Figure 5.6. A: cartoon of an unbranched neurite (thick lines). Quantitative morphometry has generated successive diameter measurements (circles) centered at x, y, z coordinates (crosses). B: Each adjacent pair of diameter measurements is treated as parallel faces of a truncated cone or frustum. The central axis of the chain of solids is indicated by a thin centerline. C: After straightening the centerline so the faces of adjacent frusta are flush with each other. The scale beneath the figure shows the distance along the midline of the section in terms of arc length, symbolized here by the variable x. The vertical dashed line at x = 0.5 divides the section into two halves of equal length. D: Equivalent circuit of the section when nseg = 1. The open rectangle includes all mechanisms for ionic (non-capacitive) transmembrane currents. Reproduced from (Hines and Carnevale 1997).

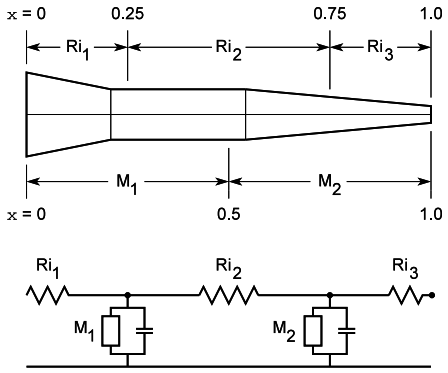


Figure 5.7. Representation of the neurite of Fig. 5.6 when `nseg` = 2. The equivalent circuit now has two nodes. See text for details. Reproduced from (Hines and Carnevale 1997).

Avoiding artifacts

Beware of zero diameter

If diameter equals 0, axial resistance becomes essentially infinite, decoupling adjacent segments. The diameter at the 0 and 1 ends of a section generally should equal the diameter of the end of the connecting section.

A blatant attempt to set diameter to 0 using the stylized method, e.g. with a statement such as

```
dend.diam(0.3) = 0
```

will produce an error message like this

```
nrniv: dend diameter diam = 0. Setting to 1e-6 in dd.hoc near line 16
```

While NEURON prevents the diameter from becoming 0, 10^{-6} μm is so narrow that axial resistance in the affected region is, for modeling intents and purposes, infinite. Models

constructed with the stylized specification can be checked for narrow diameters by executing

```
forall for (x) if (diam(x)<1) { print secname(), " ", x, " ", diam(x) }
```

which reports all locations at which `diam` falls below 1 μm. The numeric criterion in the `if` statement can be changed from 1 μm to whatever value is appropriate for the data in question. However, this will not produce definitive results if the geometry has been reinterpreted as 3-D data, in which case the 3-D data points need to be tested (see below).

The 3-D specification is more often a source of diameter problems. Morphometric data files sometimes contain measurements with diameters that are extremely small or even 0. This may occur because of operator error, or because the soma (or some other structure) was treated as a sphere with initial and terminal diameters equal to 0. Such problems can be difficult to track down because morphometric data files generally contain hundreds, if not thousands, of measurements. Furthermore, the `hoc` interpreter does not issue an error message when it encounters a `pt3dadd()` with a diameter argument of 0.

When 3-D data points exist, the value returned by `diam(x)` is the diameter of a right cylinder that would have the same length and area as the segment that contains `x`. This means that `diam(x)` may seem reasonable even though the 3-D data contain one or more points with zero (or very small) diameter so that axial resistance blows up. Therefore it is little use to check `diam(x)` when 3-D data exist. Instead, we must test the 3-D diameters by executing

```
forall for i=0,n3d()-1 if (diam3d(i)==0) { print secname(), " ", i }
```


This uses `forall` to iterate over all sections, testing each 3-D data point, and printing the name of the section and the index of each point at which diameter is found to be 0.

Stylized specification may be reinterpreted as 3-D specification

When a model is created using the stylized specification of geometry, the 3-D data list is initially empty. If the `define_shape()` procedure is then called, a number of 3-D points is created equal to the number of segments plus the end areas. This happens automatically if a `Shape` object is created, either with `hoc` statements or by using the GUI to bring up a Shape plot or any of the GUI tools that show the shape of the model, e.g. a PointProcessManager. As we mentioned above, when 3-D points exist, they determine the calculation of `L`, `diam`, `area` and `ri`. Therefore `diam`, `area`, and `ri` can change slightly merely due to `Shape` creation.

After this happens, when `L` and `diam`, are changed, there is first a change to the 3-D points, and then `L` and `diam` are updated to reflect the values of these 3-D points. In general, specifying a varying `diam` will not give exactly the same diameter values as in the case where no 3-D information exists.

For example, this code

```
create a
access a
L=100
Ra=100
nseg = 3
diam=10
diam(0.66:1)=20:20
```

defines a section with three segments, with `diam` = 10 μm in the segments centered at 0.16666667 and 0.5, and 20 μm in the segment centered at 0.83333333. Since the stylized

method was used to create this section, there will be no 3-D points. We can verify this by typing `n3d()` and noting that the returned value is 0. We can also check `diam` and the computed values of `area` and `ri` with the statement

```
for (x) print x*L, diam(x), area(x), ri(x)
```

If we now create a `Shape`, e.g. by executing

```
objref s
s = new Shape()
```

we will find that `n3d()` returns 5, i.e. there are now five 3-D points. The statement

```
for i=0, n3d()-1 print arc3d(i), diam3d(i)
```

(`arc3d(i)` is the anatomical distance of the `i`th 3-D point from the 0 end of the section) produces the output

```
0 10
16.666666 10
50 10
83.333336 20
100 20
```

which shows that the 3-D diameters have taken on the values that we had assigned using the stylized method.

However, the values of `diam`, `area`, and `ri` have been altered in the segments adjacent to the diameter change (Fig. 5.8). This effect is smaller when `nseg` is larger. It is caused by the fact that the 3-D points define a series of truncated cones rather than right circular cylinders. The reported `diam(x)` is the average diameter over the corresponding length of the 3-D model, and `area(x)` is the integral of the 3-D surface; this is not necessarily equal to the stylized area `PI*diam(x)*L/nseg`, which ignores end area associated with abrupt diameter changes. This latter difference may be small, as in this

case where `area(x)` for the second and third segments is 1185 and 1974 μm^2 respectively, compared to 1178 and 1963 μm^2 for the stylized area (all values rounded to the nearest μm^2), but actual results depend on model geometry and whether these have a significant effect on simulation results can only be judged on a case by case basis. What is clear for all cases, however, is that abrupt diameter changes should only take place at the boundaries of sections if we wish to view shape and also use the smallest possible number of segments.

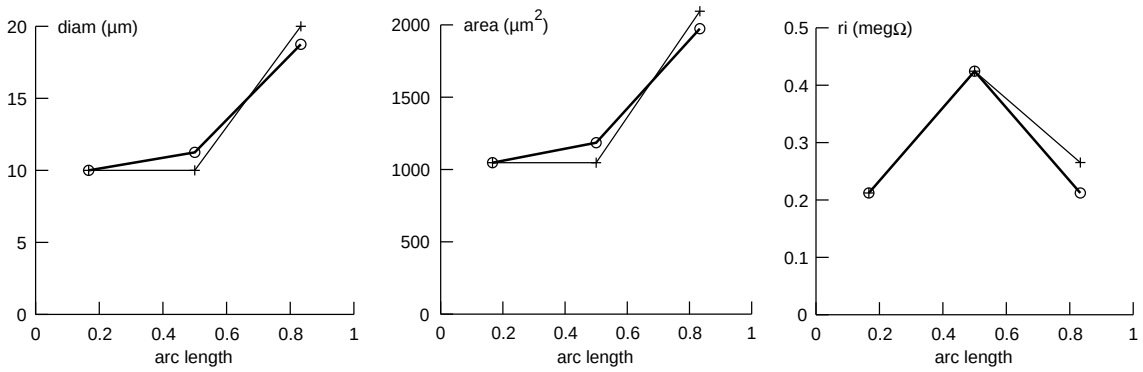


Figure 5.8. `diam`, `area`, and `ri` at the internal nodes of a 100 μm long section with `nseg` = 3 and `Ra` = 100 Ω cm. Thin lines with + show the values immediately after `geometry` was specified, when no 3-D points existed. Thick lines with circles show the values after `define_shape()` was executed, creating a set of 3-D points and forcing recalculation of `diam`, `area`, and `ri`.

How to specify biophysical properties

As we mentioned in **How to specify geometry**, the only biophysical attributes of a new section are cytoplasmic resistivity `Ra` and specific membrane capacitance `cm`, whose

default values are 35.4 Ω cm and 1 μf/cm^2, respectively. A new section has no membrane conductances, pumps, or buffers. It is assumed to lie in an extracellular medium with zero resistance or capacitance, and there are no synapses, gap junctions, or voltage or current clamps. Anything other than the bare bones framework of `Ra` and `cm` must be added.

Distributed mechanisms

Many biophysical mechanisms that generate or modulate electrical and chemical signals are distributed over the membrane or throughout the cytoplasm of a cell. In the NEURON simulation environment, these are called *distributed mechanisms*. Examples of distributed mechanisms include voltage-gated ion channels like those that generate the Hodgkin-Huxley currents, active transport mechanisms like the sodium pump, ion accumulation in a restricted space, and calcium buffers. Distributed mechanisms associated with cell membrane are often called "density mechanisms" because they are specified with *density* units, e.g. current per unit area, conductance per unit area, or pump capacity per unit area (see Table 5.3).

Distributed mechanisms are assigned to a section with an `insert` statement, as in

```
soma insert hh
dend insert pas
```

These particular statements would add the `hh` (Hodgkin-Huxley) mechanism to `soma` and the `pas` (passive) mechanism to `dend`.

Point processes

Distributed mechanisms are not the most appropriate representation of all signal sources. Localized membrane shunts (e.g. a hole in the membrane), synapses, and electrodes are called *point processes*. They are best specified using *absolute* units, i.e. microsiemens and nanoamperes, rather than the density units that are appropriate for distributed mechanisms (see Table 5.3).

Table 5.3. Examples of units associated with distributed mechanisms and point processes

Name	Meaning	Units
`gna_hh`	conductance density of open Hodgkin-Huxley sodium channels	[S/cm^2]
`ina`	net sodium current density (i.e. produced by *all* mechanisms in a section that generate sodium current)	[mA/cm^2]
`rs`	series resistance of an `SEClamp`	[10^6 Ω]
`gmax`	peak conductance of an `AlphaSynapse`	[μS]
`i`	total current delivered by an `SEClamp` or an `AlphaSynapse`	[nA]

An object syntax

```
objref varname
secname varname = new Classname(x)
varname.attribute = value
```

is used to manage the creation, insertion, attributes, and destruction of point processes. Object oriented programming in NEURON is discussed thoroughly in **Chapters 13** and **14**; to illustrate the pertinent essentials for dealing with point processes, let us consider the following code, which implements a current clamp attached to the middle of a section called `soma`.


```
objref stim
soma stim = new IClamp(0.5)
stim.amp = 0.1
stim.del = 1
stim.dur = 0.1
```

The first line declares that `stim` is a special kind of variable called an `objref` (object reference), which we will use to refer to the current clamp object. The second line creates a new instance of the `IClamp` object class, located at the middle of `soma`, and assigns this to `stim`. The next three lines specify that `stim` will deliver a 0.1 nA current pulse that begins at t = 1 ms and lasts for 0.1 ms.

When a point process is no longer referenced by any object reference, it is removed from the section and destroyed. Consequently, redeclaring `stim` with the statement `objref stim` would destroy this `IClamp`, since no other object reference would reference it.

The `x` position specified for a point process can have any value in the range [0,1]. If `x` is specified to be 0 or 1, the point process will be located at the corresponding end of the section. For specified locations $0 < x < 1$, the actual position used by NEURON will be the center of the segment that contains `x`. Thus, if `dend` has `nseg` = 5, the segment centers (internal nodes) are located at `x` = 0.1, 0.3, 0.5, 0.7 and 0.9, so

```
objref stim1, stim1
dend stim1 = new IClamp(0.04)
dend stim2 = new IClamp(0.61)
```

would actually place `stim1` at 0.1 and `stim2` at 0.7. The error introduced by this "shift" can be avoided by explicitly placing point processes at internal nodes, and restricting changes of `nseg` to odd multiples. However, this may not be possible in models that are based closely on real anatomy, because actual synaptic locations are unlikely to be

situated precisely at segment centers. To completely avoid `nseg`-dependent shifts of point process locations, one can choose sections with lengths such that the point processes are located at 0 or 1 ends.

The location of a point process can be changed without affecting its other attributes. Thus `dend stim2.loc(0)` would move `stim2` to the 0 end of `dend`.

If a section's `nseg` is changed, the point processes on that section are relocated to the centers of the new segments that contain the centers of the old segments to which the point processes had been assigned. When a segment is destroyed, as by re-creating the section, all of its point processes lose their attributes, including `x` location and which section they belong to.

Many distributed mechanisms and point processes can be simultaneously present in each segment. One important difference between distributed mechanisms and point processes is that any number of the same kind of point process can exist at the same location, whereas a distributed mechanism is either present or not present in a section. For example, several `AlphaSynapses` might be attached to the `soma`, but the `hh` mechanism would either be present or absent.

User-defined mechanisms

User-defined distributed mechanisms and point processes can be added to NEURON with the model description language NMODL. This lets the user focus on specifying the equations for a channel or ionic process without regard to its interactions with other mechanisms. The NMODL translator constructs C code which properly and efficiently computes the total current of each ionic species used, as well as the effect of that current

on ionic concentration, reversal potential, and membrane potential. This code is compiled and linked into NEURON. NMODL is discussed extensively in **Chapter 9** and **10**, but it is useful to review some of its advantages here.

1. Details of interfacing new mechanisms to NEURON are handled automatically--and there are a great many such details. For instance,

- NEURON needs to know that model states are range variables, and which model parameters can be assigned values and evaluated from the interpreter.
- Point processes need to be accessible via the interpreter's object syntax, and density mechanisms need to be added to a section when the `insert` statement is executed.
- If two or more channels use the same ion at the same place, the individual current contributions must be added together to calculate a total ionic current.

2. Consistency of units is ensured.

3. Mechanisms described by kinetic schemes are written with a syntax in which the reactions are clearly apparent. The translator provides tremendous leverage by generating a large block of C code that calculates the analytic Jacobian and the state fluxes.

4. There is often a great increase in clarity since statements are at the model level instead of the C programming level and are independent of the numerical method. For instance, sets of differential and nonlinear simultaneous equations are written using an expression syntax such as

```
x' = f(x, y, t)
~ g(x, y) = h(x, y)
```


where the prime refers to the derivative with respect to time (multiple primes such as `x''` refer to higher derivatives) and the tilde introduces an algebraic equation. The algebraic portion of such systems of equations is solved by Newton's method, and a variety of methods are available for solving the differential equations (see **Chapter 9**).

5. Function tables can be generated automatically for efficient computation of complicated expressions.

6. Default initialization behavior of a channel can be specified.

Working with range variables

Iterating over nodes

As we mentioned above in **How NEURON separates anatomy and biophysics from purely numerical issues**, many anatomical and biophysical properties can vary along the length of a section, and these are represented in NEURON by range variables. The syntax

```
for (var) stmt
```

is a convenient idiom for working with range variables. This statement assigns the location of each node (in arc length, starting at 0 and ending at 1) to *`var`* and then executes *`stmt`*. For example,

```
axon for (x) print x*L, v(x)
```

will print the membrane potential as a function of physical distance along `axon`.

Linear taper

If a range variable is a linear, or nearly linear, function of distance along a section, it can be specified with the syntax

```
rangevar(xmin:xmax) = e1:e2
```

where the four italicized symbols are expressions. The position expressions must satisfy the constraint $0 \le xmin \le xmax \le 1$. The values of the property at *xmin* and *xmax* are `e1` and `e2`, respectively, and linear interpolation is used to assign the values of the property at the nodes that lie in the position range [*xmin*, *xmax*]. If the range variable is `diam`, neither `e1` nor `e2` should be 0, or the corresponding axial resistance will be infinite. As an example, suppose `axon` contained the Hodgkin-Huxley spike channels, and we wanted the density of sodium channels to start at its normal level of 0.12 siemens/cm^2 at the 0 end and fall linearly with distance until it becomes 0 at the other end. This could be done with the statement

```
axon.gnabar_hh(0:1) = 0.12:0
```

The actual conductance densities in the individual segments will depend on the value of `nseg`, as shown in Table 5.4. This assignment must be executed *after* the desired value of `nseg` has been specified, for reasons that are explained in the next few paragraphs.

Table 5.4. Effect of `nseg` on linear variation of sodium channel density `gnabar_hh` with distance.

nseg	Segment centers (in units of arc length)	Channel density [siemens/cm^2]
1	0.5	0.06
2	0.25	0.09
	0.75	0.03
3	0.1667	0.1
	0.5	0.06
	0.8333	0.02
5	0.1	0.108
	0.3	0.084
	0.5	0.06
	0.7	0.036
	0.9	0.012

How changing `nseg` affects range variables

If `nseg` is increased after range variables have been specified, all old segments are relocated to their nearest new locations (no instance variables are modified and no pointers to data in those segments become invalid), and new segments are allocated and given mechanisms and values that are identical to the old segment in which the center of the new segment is located. If range variables are not constant, then the `hoc` expressions used to set them should be re-executed. To see why, let us return to our axon with a linearly tapering `gnabar_hh`, specified by executing

```
nseg = 3
axon.gnabar_hh(0:1) = 0.12:0
```

after which we check by executing

```
axon for (x) print x, gnabar_hh(x)
```


which returns

```
0 0.1
0.16666667 0.1
0.5 0.06
0.83333333 0.02
1 0.02
```

as we expect from Table 5.4 (the values at the 0 and 1 ends are merely copied from the nearest nodes, and don't really matter since the areas associated with the 0 and 1 ends are 0). Now we triple the number of nodes and check `gnabar_hh` by executing

```
nseg *= 3
axon for (x) print x, gnabar_hh(x)
```

and see

```
0 0.1
0.055555556 0.1
0.16666667 0.1
0.27777778 0.1
0.38888889 0.1
0.5 0.06
0.61111111 0.06
0.72222222 0.06
0.83333333 0.02
0.94444444 0.02
1 0.02
```

Even though we have nine internal nodes, the spatial gradient for `gnabar_hh` is just as crude as before, with only three transitions along the length of our section. To fix this, we must reassert

```
axon.gnabar_hh(0:1) = 0.12:0
```

and when we now test the gradient we find


```
0 0.11333333
0.055555556 0.11333333
0.16666667 0.1
0.27777778 0.086666667
0.38888889 0.073333333
0.5 0.06
0.61111111 0.046666667
0.72222222 0.033333333
0.83333333 0.02
0.94444444 0.0066666667
1 0.0066666667
```

i.e. `gnabar_hh` is progressively smaller at each internal node of `axon`, which is what we wanted all along.

What if we *decrease* `nseg`? All the new segments will in fact be the old segments that are nearest to the new segments. Another way to think about this is to see what old segments contain the new nodes, and those are the segments that will be preserved. This is what makes it so useful to increase and decrease `nseg` by the same odd factor, e.g. 3. So going from `nseg` = 9 back to `nseg` = 3 restores our original model with its original parameter values, even if we don't bother to execute

```
axon.gnabar_hh(0:1) = 0.12:0
```

again. If instead we reduced `nseg` from 9 to 5, the spatial profile of `gnabar_hh` would be

```
0 0.11333333
0.1 0.11333333
0.3 0.086666667
0.5 0.06
0.7 0.033333333
0.9 0.0066666667
1 0.0066666667
```


which clearly differs from the result of executing

```
nseg = 5
axon.gnabar_hh(0:1) = 0.12:0
```

(see Table 5.4).

Choosing a spatial grid

Designing the spatial grid for a computational model involves a tradeoff between improving accuracy, on the one hand, and increasing storage requirements and runtime on the other. The goal is to achieve sufficient accuracy while keeping the computational burden as small as possible.

A consideration of intent and judgment

The question of how to achieve sufficient accuracy depends in part on what one means by "sufficient." The answer depends both on the anatomical and biophysical attributes of the conceptual model *and* the modeler's intent. Most treatments of discretization tend to ignore intent, and judgment, its close cousin. Intent and judgment are inherently tied closely to the particular interests of the individual investigator, so it is difficult to make general pronouncements about them. However, they can be dominant factors in the discretization of time and space, as the following two examples demonstrate.

Consider a model of a small spherical cell with passive membrane that is subjected to a depolarizing current pulse (Fig. 5.9). The spatial grid for this isopotential cell only

needs a single node, i.e. this is a situation in which the sole consideration to be weighed is the discretization of time.

The middle and right panels in Fig. 5.9 show the analytic solution for membrane potential V_m (dashed orange trace) along with numeric solutions that were computed using several different values of Δt (solid black trace). Clearly it is the numeric solution computed with the smallest Δt that best reflects the curvature of V_m in time. Solutions computed with large Δt lack the high frequency terms needed to follow the initial rapid change of V_m (see **Analytic solutions: continuous in time and space** in **Chapter 4**). However, with the advance of time, even the least accurate numeric solution soon becomes indistinguishable from the analytic solution. Which of these solutions "best" suits our needs depends on our intent. If it is essential to us that the solution faithfully captures the smooth curve of the analytic solution, we would prefer to use the smallest Δt, perhaps even smaller than 10 ms. But if we are only interested in the final steady state value of V_m, then Δt = 40 ms is probably good enough.

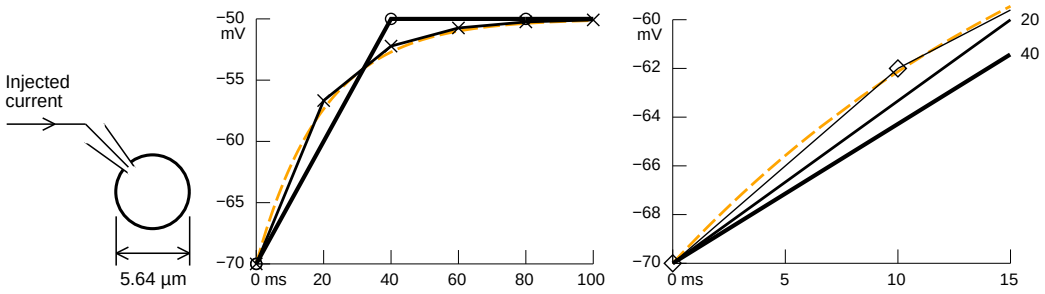


Figure 5.9. A spherical cell (left) with a surface area of 100 μm^2 (diameter = 5.64 µm) is subjected to a 1 pA depolarizing current that starts at t = 0 ms. Resting potential is -70 mV, specific membrane capacitance and resistance are C_m = 1 µf / cm^2 and R_m = 20,000 Ω cm^2, respectively (τ_m = 20 ms). The dashed orange trace in the middle and right graphs is the analytic solution for V_m. The solid black traces are the numeric solutions computed with time steps Δt = 40 ms (thick trace, open circles), 20 ms (medium trace, ×), and 10 ms (thin trace, diamond, right figure only). Modified from (Hines and Carnevale 2001).

Spatial discretization becomes important in models that are extensive enough for the propagation of electrical or chemical signals to involve significant delay. We illustrate this with a model of fast excitatory synaptic input onto a dendritic branch. The synapse in this model is attached to the middle of an unbranched cylinder (Fig. 5.10). To prevent possible confounding effects of active current kinetics and complex geometry, we assume that the cylinder has passive membrane and is five DC length constants long. The biophysical properties are within the range reported for mammalian central neurons (Spruston and Johnston 1992). The time course of the synaptic conductance follows an alpha function with time constant τ_s and reversal potential E_s chosen to emulate an

AMPA synapse (Kleppe and Robinson 1999), and g_{max} selected to produce a peak depolarization of approximately 10 mV. We will compare the analytic solution for V_m in this model with the numeric solution computed for a very coarse spatial grid ($\Delta x = 1\ \lambda$). The numeric solution uses a time step $\Delta t = 1$ μs, which is more than two orders of magnitude smaller than necessary to follow the EPSP waveform, so that differences from the analytic solution are almost entirely attributable to the spatial grid.

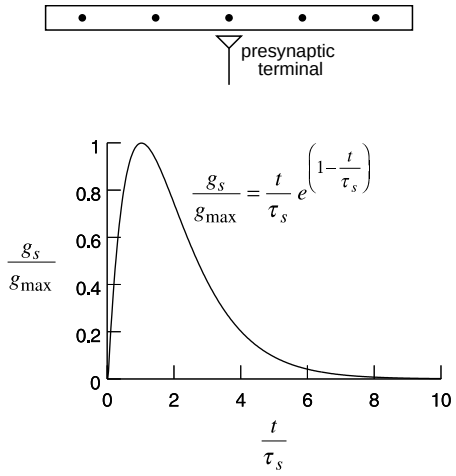


Figure 5.10. Model of synaptic input onto a dendrite The dendrite is represented by an unbranched cylinder (top) with diameter = 1 μm, length = 2500 μm, R_a = 180 Ω cm, C_m = 1 μf / cm^2, and R_m = 16,000 Ω cm^2 with a resting potential of -70 mV. The DC length constant λ_{DC} of the cylinder is 500 μm, so its sealed end terminations have little effect on the EPSP produced by a synapse located at its midpoint. The dots are the locations at which the numeric solution would be computed using a grid with intervals of 1 λ_{DC}, i.e. 250, 750, 1250, 1750, and 2250 μm. The synaptic conductance g_s is governed by an alpha function (bottom) with τ_s = 1ms, $g_{max} = 10^{-9}$ siemens, and reversal potential E_s = 0 mV. Modified from (Hines and Carnevale 2001).

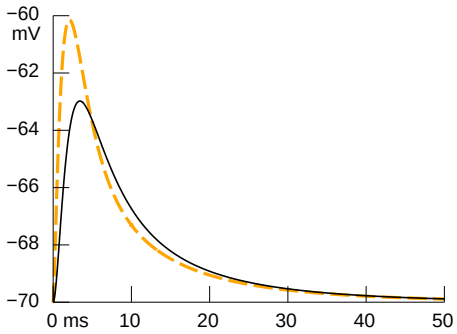


Figure 5.11. Time course of V_m at the synaptic location. The dashed orange line is the analytic solution, and the solid black line is the numeric solution computed with Δt = 1 μs. Modified from (Hines and Carnevale 2001).

Compared to the analytic solution for V_m at the site of synaptic input (dashed orange trace in Fig. 5.11), the numeric solution (solid black trace) rises and falls more slowly, and has a peak depolarization that is substantially delayed and smaller. These differences reflect the fact that solutions based on the coarse grid lack sufficient amplitude in the high frequency terms that are needed to reproduce rapidly changing signals. Such errors could lead to serious misinterpretations if our intent were to examine how synaptic input might affect depolarization-activated currents with fast kinetics like I_A, spike sodium current, and transient I_{Ca}.

The graphs in Fig. 5.12 present the spatial profile of V_m along the dendrite at two times selected from the rising and falling phases of the EPSP. These curves, which are representative of the early and late response to synaptic input, show that the error of the numeric solution is most pronounced in the part of the cell where V_m changes most rapidly, i.e. in the near vicinity of the synapse. However, at greater distances the analytic

solution itself changes much more slowly because of low pass filtering produced by cytoplasmic resistance and membrane capacitance. At these distances the error of the numeric solution is surprisingly small, even though it was computed with a very crude spatial grid. Furthermore, error decreases progressively as time advances and high frequency terms become less important. This suggests that the coarse grid may be quite sufficient if our real interests are in slow processes that take place at some distance from the site of synaptic input.

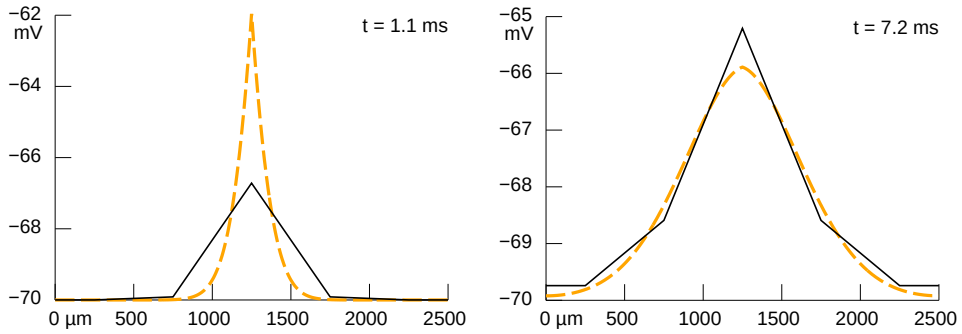


Figure 5.12. V_m vs. distance along the dendrite computed during the rising (left) and falling (right) phases of the EPSP. The analytic and numeric solutions are shown with dashed orange and solid black lines, respectively. The error of the numeric solution is greatest in the region where V_m changes most rapidly, i.e. in the neighborhood of the synapse.

Discretization guidelines

Various strategies have appeared in the literature as aids to the use of judgment in choosing a spatial grid. One common practice is to keep the distance between adjacent grid points smaller than some fraction (e.g. 5 - 10%) of the DC length constant λ_{DC} of an

infinite cylinder with identical anatomical and biophysical properties (Mainen and Sejnowski 1998; Segev and Burke 1998). This plausible approach has two chief limitations. First, large changes in membrane resistance and λ_{DC} can be produced by activation of voltage-dependent channels (e.g. I_h (Magee 1998; Stuart and Spruston 1998)), Ca^{2+}-gated channels (Wessel et al. 1999), or synaptic inputs (Bernander et al. 1991; Destexhe and Pare 1999; Häusser and Clark 1997; Pare et al. 1998). The second but more fundamental problem is that the spatial decay of transient signals is unrelated to λ_{DC}. Cytoplasmic resistivity R_a and specific membrane capacitance C_m constitute a spatially distributed low pass filter, so transient signals suffer greater distortion and attenuation with distance than do slowly changing signals or DC. In other words, by virtue of their high frequency components in time, transient signals also have high frequency components in space. Just as high temporal frequencies demand a short time step, high spatial frequencies demand a fine grid.

The d_lambda rule

As a more rational approach, we have suggested what we call the "d_lambda rule" (Hines and Carnevale 2001), which predicates the spatial grid on the AC length constant λ_f computed at a frequency *f* that is high enough for transmembrane current to be primarily capacitive, yet still within the range of frequencies relevant to neuronal function. Ionic and capacitive transmembrane currents are equal at the frequency $f_m = 1 / 2 \pi \tau_m$, so specific membrane resistance R_m has little effect on the propagation of

signals ≥ 5 f_m. For instance, a membrane time constant of 30 ms corresponds to f_m ~ 5 Hz, which implies that R_m would be irrelevant to signal spread at frequencies ≥ 25 Hz. Most cells of current interest have $\tau_m \geq 8$ ms (f_m ~ 20 Hz), so we suggest that the distance between adjacent nodes should be no larger than a user-specified fraction "d_lambda" of λ_{100}, the length constant at 100 Hz. This frequency is high enough for signal propagation to be insensitive to shunting by ionic conductances, but it is not unreasonably high because the rise time τ_r of fast EPSPs and spikes is ~ 1 ms, which corresponds to a bandpass of $1/\tau_r\sqrt{2\pi}$ ~400 Hz.

At frequencies where R_m can be ignored, the attenuation of signal amplitude is described by

$$\log\left|\frac{V_0}{V_x}\right| \approx 2x\sqrt{\frac{\pi f R_a C_m}{d}} \qquad \text{Eq. 5.1}$$

so the distance over which an *e*-fold attenuation occurs is

$$\lambda_f \approx \frac{1}{2}\sqrt{\frac{d}{\pi f R_a C_m}} \qquad \text{Eq. 5.2}$$

where *f* is in Hz. For example, a dendrite with diameter = 1 μm, R_a = 180 Ω cm, C_m = 1 μf / cm^2, and R_m = 16,000 Ω cm^2 has λ_{DC} = 500 μm, but λ_{100} is only ~225 μm.

In NEURON the d_lambda rule is implemented in the CellBuilder, which allows the maximum anatomical distance between grid points to be specified as a fraction of λ_{100} using an adjustable parameter called `d_lambda`. The default value of `d_lambda` is 0.1, which is more than adequate for most purposes, but a smaller value can be used if τ_m is shorter than 8 ms. For increased flexibility, the CellBuilder also provides two alternative strategies for establishing the spatial grid: specifying `nseg`, the actual number of grid points; specifying `d_X`, the maximum anatomical distance between grid points in μm. The d_lambda and the d_X rules both deliberately set `nseg` to an odd number, which guarantees that every branch will have a node at its midpoint. These strategies can be applied to any section or set of sections, each having its own rule and compartmentalization parameter. Barring special circumstances e.g. localized high membrane conductance, it is usually sufficient to use the d_lambda rule for the entire model. However, regardless of which strategy is selected, it is always advisable to try a few exploratory runs with a finer grid to be sure that spatial error is acceptable.

Eq. 5.2 shows that the attenuation of fast signals (e.g. fast PSPs, rapid steps under voltage clamp, passively conducted spikes) is governed by cytoplasmic resistivity, specific membrane capacitance, and neurite diameter. Specific membrane resistance and membrane time constant are irrelevant. Therefore channel blockers will not improve the fidelity of recordings of fast signals, or the ability to clamp fast active currents.

Of course the d_lambda rule can also be applied without having to use the GUI. The following procedure


```
proc geom_nseg() {
  soma area(0.5) // make sure diam reflects 3d points
  forall {nseg = int((L/(0.1*lambda_f(100))+0.9)/2)*2 + 1}
}
```

iterates over all sections to ensure that each section has an odd `nseg` that is large enough to satisfy the d_lambda rule. This makes use of the function

```
func lambda_f() { // currently accessed section, $1 == frequency
        return 1e5*sqrt(diam/(4*PI*$1*Ra*cm))
}
```

which is included in the file

`nrn-x.x/share/lib/hoc/stdlib.hoc` (UNIX/Linux)

or

`c:\nrnxx\lib\hoc\stdlib.hoc` (MSWindows)

(`x.x` and `xx` are used here to refer to the version number of NEURON). This file is automatically loaded when

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

is executed or the `nrngui` script or icon is launched. Alternatively, `stdlib.hoc` can be loaded alone with the command

```
load_file("stdlib.hoc")
```

or else `func lambda_f()` can be recreated by itself with `hoc`.

To see how the d_lambda rule works in practice, consider the model in Fig. 5.13, which represents a granule cell from the dentate gyrus of the rat hippocampus. This model is based on quantitative morphometric data provided by Dennis Turner (available from `http://www.cns.soton.ac.uk/~jchad/cellArchive/cellArchive.html` or

`http://www.compneuro.org/CDROM/nmorph/cellArchive.html`), and the biophysical parameters are from Spruston and Johnston (Spruston and Johnston 1992): R_m = 40 k Ω cm^2, C_m = 1 μf / cm^2, and R_a = 200 Ω cm. An excitatory synapse attached to the soma is an excitatory synapse whose conductance is governed by an alpha function with τ_s = 1 ms, $g_{max} = 2 \cdot 10^{-9}$ S, and reversal potential E_s = 0 mV.

The right side of Fig. 5.13 shows the simulated time course of V_m at the soma for three different methods of specifying the spatial grid: one or three nodes in each branch, and `d_lambda` = 0.3. On the scale of this figure, solutions with `d_lambda` ≤ 0.3 are indistinguishable from each other, so `d_lambda` = 0.3 serves as the standard for accuracy. Plots generated with constant `nseg` per branch converged toward this trace as `nseg` increased. Even the crudest spatial grid (`nseg` = 1) would suffice if the purpose of the model were to evaluate effects of synaptic input on V_{soma} well after the peak of the EPSP (t > 7 ms). However a finer grid is clearly necessary if the maximum somatic depolarization produced by the EPSP is of concern.

Additional refinements to the grid are needed if we want to know how the EPSP spreads into other parts of the cell, e.g. along the path marked by orange in Fig. 5.14 left. To compute the maximum depolarization produced by a somatic EPSP along this path, a grid that has only 3 nodes per branch is quite sufficient (Fig. 5.14 center). If the timing of this peak is important, e.g. for coincidence detection or activation of voltage-gated currents, a finer grid must be used (Fig. 5.14 right).

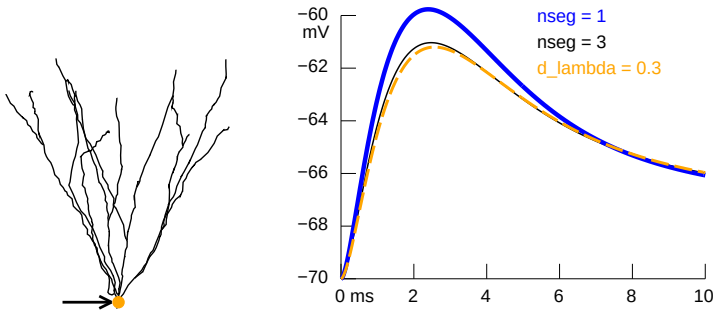


Figure 5.13. Left: Anatomically complex model of a granule cell from the dentate gyrus of rat hippocampus. A fast excitatory synapse is attached to the soma (location indicated by arrow and orange dot). See text for details. Right: Time course of V_{soma} computed using spatial grids with one or three nodes per branch (thick blue and thin black traces for `nseg` = 1 and 3, respectively) or specified with `d_lambda` = 0.3 (dashed orange trace). Modified from (Hines and Carnevale 2001).

The computational cost of these simulations is approximately proportional to the number of nodes. Least burdensome, but also least accurate, were the simulations generated with one node per branch, which involved a total of 28 nodes in the model. Increasing the number of nodes per branch to 3 (total nodes in model = 84) improved accuracy considerably, but obvious errors remained (Fig. 5.14 right) that disappeared only after an additional tripling of the number of nodes per branch (total nodes = 252; results not shown). The greatest accuracy with least sacrifice of efficiency was achieved with the grid specified by `d_lambda` = 0.3, which contained only 110 nodes.

As these figures suggest, the relative advantage of the d_lambda rule will be most apparent when signal propagation throughout the entire model must be simulated to a similar level of accuracy. If the focus is on a limited region, then a grid with fewer nodes

and a simpler representation of electrically remote regions may be acceptable. Special features of the model may also allow a simpler grid to be used. In principal neurons of mammalian cortex, for example, proximal dendritic branches tend to have larger diameters (Rall 1959; Hillman 1979) and shorter lengths (Cannon et al. 1999) than do distal branches. Therefore models based on quantitative morphometry of such neurons will have fewer nodes in proximal dendrites than in more distal dendrites if the grid is specified by the d_lambda or d_X rule. Indeed, many proximal branches may have only one or three nodes, regardless of which rule is applied, and in such a case the differences between gridding strategies will be manifest only in the thinner and longer distal branches. Such differences will have little effect on accuracy if signals in the vicinity of the soma are the only concern.

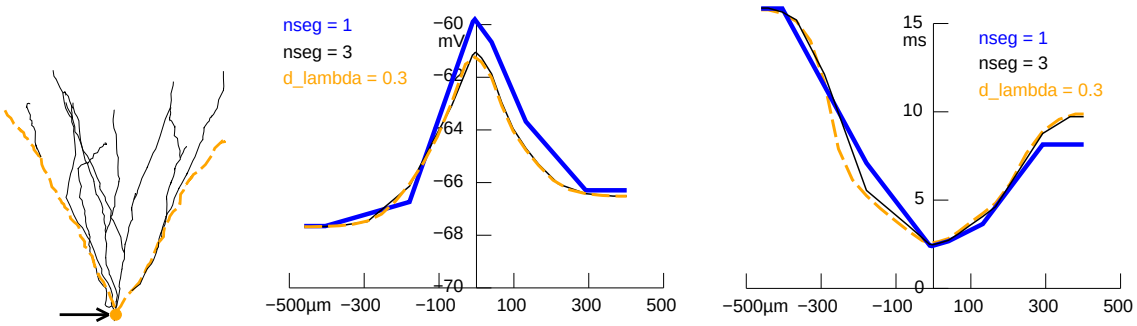


Figure 5.14. The EPSP evoked by activation of a synapse at the soma (arrow in left panel) spread into the dendrites, producing a transient depolarization which grew smaller and occurred later as distance from the soma increased. The center and right panels show the magnitude and timing of this depolarization along the path marked by the dashed orange line. Peak amplitude was quite accurate with `nseg` = 3 (thin black trace, center panel), but noticeable error persisted in the time of peak depolarization for distances between -300 and -150 µm (right panel, especially between). The dashed orange trace in the center and right panels was obtained with `d_lambda` = 0.3. Time step was 25 µs.

References

Bernander, Ö., Douglas, R.J., Martin, K.A.C., and Koch, C. Synaptic background activity influences spatiotemporal integration in single pyramidal cells. *Proc. Nat. Acad. Sci.* 88:11569-11573, 1991.

Cannon, R.C., Wheal, H.V., and Turner, D.A. Dendrites of classes of hippocampal neurons differ in structural complexity and branching patterns. *J. Comp. Neurol.* 413:619-633, 1999.

Crank, J. *The Mathematics of Diffusion*. 2 ed. London: Oxford University Press, 1979.

Destexhe, A. and Pare, D. Impact of network activity on the integrative properties of neocortical pyramidal neurons in vivo. *J. Neurophysiol.* 81:1531-1547, 1999.

Häusser, M. and Clark, B.A. Tonic synaptic inhibition modulates neuronal output pattern and spatiotemporal synaptic integration. *Neuron* 19:665-678, 1997.

Hillman, D.E. Neuronal shape parameters and substructures as a basis of neuronal form. In: *The Neurosciences: Fourth Study Program*, edited by F.O. Schmitt and F.G. Worden. Cambridge, MA: MIT Press, 1979, p. 477-498.

Hines, M.L. and Carnevale, N.T. The NEURON simulation environment. *Neural Computation* 9:1179-1209, 1997.

Hines, M.L. and Carnevale, N.T. NEURON: a tool for neuroscientists. *The Neuroscientist* 7:123-135, 2001.

Kleppe, I.C. and Robinson, H.P.C. Determining the activation time course of synaptic AMPA receptors from openings of colocalized NMDA receptors. *Biophys. J.* 77:1418-1427, 1999.

Magee, J.C. Dendritic hyperpolarization-activated currents modify the integrative properties of hippocampal CA1 pyramidal neurons. *J. Neurosci.* 18:7613-7624, 1998.

Mainen, Z.F. and Sejnowski, T.J. Modeling active dendritic processes in pyramidal neurons. In: Methods in Neuronal Modeling, edited by C. Koch and I. Segev. Cambridge, MA: MIT Press, 1998, p. 171-209.

Pare, D., Shink, E., Gaudreau, H., Destexhe, A., and Lang, E.J. Impact of spontaneous synaptic activity on the resting properties of cat neocortical pyramidal neurons in vivo. *J. Neurophysiol.* 79:1450-1460, 1998.

Rall, W. Branching dendritic trees and motoneuron membrane resistivity. *Experimental Neurology* 1:491-527, 1959.

Rall, W. Core conductor theory and cable properties of neurons. In: *Handbook of Physiology, vol. 1, part 1: The Nervous System*, edited by E.R. Kandel. Bethesda, MD: American Physiological Society, 1977, p. 39-98.

Spruston, N. and Johnston, D. Perforated patch-clamp analysis of the passive membrane properties of three classes of hippocampal neurons. *J. Neurophysiol.* 67:508-529, 1992.

Stuart, G. and Spruston, N. Determinants of voltage attenuation in neocortical pyramidal neuron dendrites. *J. Neurosci.* 18:3501-3510, 1998.

Wessel, R., Kristan, W.B., and Kleinfeld, D. Dendritic Ca2+-activated K+ conductances regulate electrical signal propagation in an invertebrate neuron. *J. Neurosci.* 19:8319-8326, 1999.

Chapter 5 Index

C

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

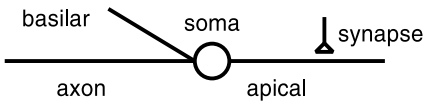


Fig. 6.1. The model neuron. The conductance change synapse can be located anywhere on the cell.

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.

```
//////////////////////////
/* 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
}

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 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}
}
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.

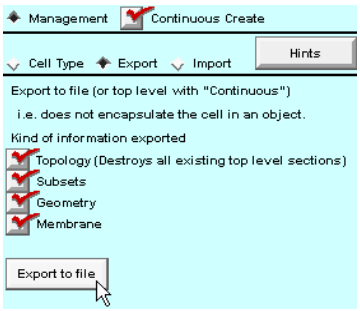


Figure 6.2. The Management page of the CellBuilder. We have clicked on the Export radio button, and are about to export the model's topology, subsets, geometry, and membrane information to a hoc file that can be executed to recreate the model cell.

```
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()

  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 `proc`s (the temptingly simple `topol()` and the slightly puzzling `basic_shape()`) before encountering a declaration of three `objref`s (see **Chapter 13: Object oriented programming**)

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

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.

```
//////////////////////
/*   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 (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 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?

`nrngui` also loads the standard run library

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.

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**).

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?

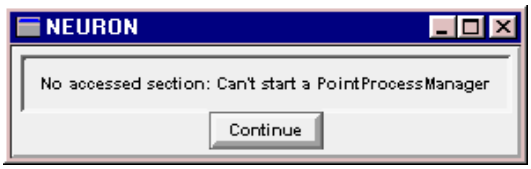


Fig. 6.2. A useful error message.

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

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

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 }

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.

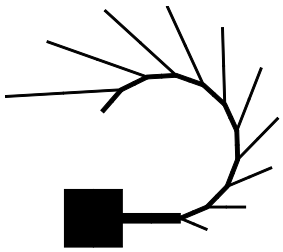


Fig. 6.3. Shape plot rendering of the model produced by the code in Listing 6.5. To help indicate the location of the soma section, Shape Style: Show Diam was enabled.

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)
}
```



```
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.

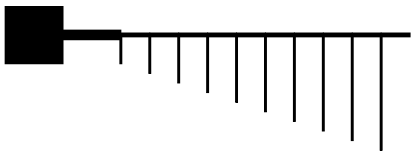


Fig. 6.4. Shape plot rendering of the model when the geometry is specified using the 3-D method shown in Listing 6.6.

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)
}

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)
}

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.

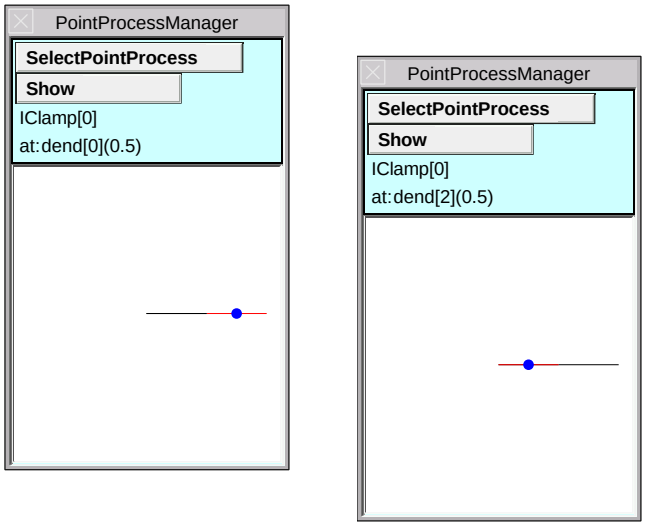


Fig. 6.5. The code in Listing 6.8 produces a model that seems not to have a `dend[1]`--or at least, we can't find `dend[1]` when we try to use a PointProcessManager to attach an `IClamp` to it.

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.

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.

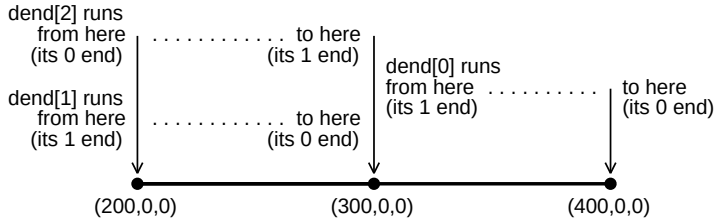


Fig. 6.6. Deciphering the `pt3dadd()` statements in Listing 6.8 leads us to realize that we only see two sections in the Shape plot because two of them (`dend[1]` and `dend[2]`) are drawn in the same place. This figure shows the (x,y,z) coordinates of the sections and indicates their 0 and 1 ends.

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 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.

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 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.

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.

Chapter 6 Index

Chapter 7

How to control simulations

Simulation control with the GUI

The RunControl panel (Fig. 7.1 right) has several buttons and field editors (boxes that contain numbers) that provide a basic set of controls for initializing, starting, and stopping simulations. The actions listed in Table 7.1 are "defaults," i.e. the standard behavior of the tool. These actions are all customizable, because the RunControl works by calling procedures that are defined in hoc (see below) so you can always create a new procedure with the same name that substitutes for the default code.

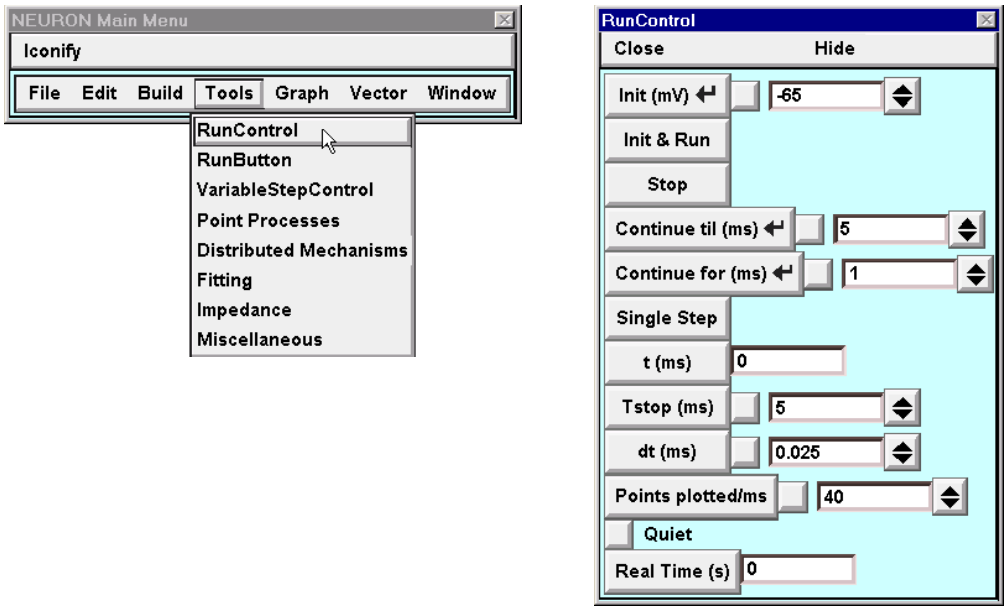


Fig. 7.1. Left: NEURON Main Menu / Tools / RunControl brings up a panel with controls for running simulations. Right: The RunControl panel allows a great deal of control over the execution of simulations. See text for details.

In learning to use the RunControl panel it may help to keep in mind that adjacent controls have related functions. The three buttons at the top (Init, Init & Run, and Stop) perform the most common operations: initializing, starting, and stopping simulations. The next three (Continue til (ms), Continue for (ms), and Single Step) are particularly helpful for exploratory dissection of the time sequence of events in dynamically complex simulations.

Graphs created from the NEURON Main Menu respond appropriately to all of these controls. Init erases unsaved traces from graphs whose x axis shows time, and makes all other graphs (e.g. variables vs. anatomical location, phase plane plots) show initial values, whereas Init & Run, Continue til, Continue for, and Single Step cause graphs to be updated at intervals governed by Points plotted/ms and Quiet.

Table 7.1. Functions of the RunControl panel

Button	Action
Init (mV)	Sets time to 0, changes V_m throughout the model to the value displayed in the adjacent field editor, initializes ionic concentrations, and sets biophysical mechanisms (e.g. ionic conductances, pumps) to their corresponding steady state values.
Init & Run	Same as the Init button, but then launches a simulation that runs until t equals Tstop (see below). Graphs constructed from the NEURON Main Menu are updated at a rate specified by Points plotted/ms and Quiet (see below).
Stop	Stops a simulation at the end of a step.
Continue til (ms)	Continues a simulation until t ≥ the value displayed in the adjacent field editor. Graphs are updated according to Points plotted/ms (see below).
Continue for (ms)	Continues a simulation for the amount of time displayed in the adjacent field editor. Graphs are updated according to Points plotted/ms (see below).
Single Step	Continues a simulation for one step and plots. A step is 1 / (Points plotted/ms) milliseconds and consists of 1 / (dt · Points plotted/ms) calls to `fadvance()`.
t (ms)	No action. The adjacent numeric field shows model time during the course of a simulation.
Tstop (ms)	No action. Adjacent field is used to specify stop time for Init & Run.
dt (ms)	No action. Adjacent field shows the fundamental integration time step used by `fadvance()`. Values entered into this field editor are automatically rounded down so that an integral multiple of `fadvance`s make up a Single Step.
Points plotted/ms	No action. Adjacent field is used to specify the number of times per millisecond at which graphs are updated. Notice that reducing dt does not by itself increase the number of points plotted. If 1 / (Points plotted/ms) is not an integral multiple of dt, then dt is rounded down to the nearest integral fraction of 1 / (Points plotted/ms).
Quiet	When checked, turns off graph updates during a simulation. This can speed things up considerably, e.g. when using the Multiple Run Fitter in the presence of a shape movie plot under MSWindows.

Real Time (s)	No action. Adjacent field shows a running display of computation time ("run time"), with a resolution of 1 second.

The standard run system

The Init & Run button of the RunControl panel is probably the user's first contact with the standard run system. The standard run system is implemented in the file

`nrn-x.x/share/lib/hoc/stdrun.hoc` (UNIX/Linux)

or

`c:\nrnxx\lib\hoc\stdrun.hoc` (MSWindows)

which is interpreted with a number of other files when

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

is executed or the `nrngui` script or icon is launched. This system is a considerable elaboration over the minimal "oscilloscope level" simulation

```
proc run() {
   finitialize(-65)
   fcurrent()
   while (t < 5) {
      fadvance()
   }
}
```

which integrates a cell specification from `t` = 0 to `t` = 5 ms. The elaborations consist of various parameters and hooks for starting and stopping the simulation and obtaining information during the simulation run. Tools that involve the analysis of simulation results, e.g. optimization tools such as the Multiple Run Fitter, assume the existence of a

`run()` procedure to carry out their evaluation of the difference between simulation result and data.

Understanding a few aspects of the standard run system is necessary in order to be able to write functions or objects that can work in the presence of this framework, or at least do not vitiate it. It is generally much easier to work with and reuse components of this system than attempt to recreate a great deal of existing functionality. Most users have come to count on existing features that allow plotting of any variable during a run, or easy switching between integration methods.

NEURON's standard run system was designed with the realization that research requirements are quite varied, so no generic implementation will suffice in all cases. Therefore an attempt was made to divide the run process into as many elements as seemed reasonable in order to make it easy for the user to replace any one of them. In most cases a replacement procedure requires only one or two specific code statements directed toward maintaining its standard function. The standard run system has proven to be usable without changes in a wide variety of situations, with the exception of the `init()` procedure for initialization (this is discussed extensively in **Chapter 8**). Nevertheless, certain problems can only be overcome by writing `hoc` code, or even low level C code, so it is helpful to have a tour of the sequence of events that leads to an actual time step advance. Some details of the following discussion may change because methods are constantly being revised to improve performance, but the broad outline of program organization and execution will remain the same--especially in areas that are most likely to require customization.

Be sure to load replacements of standard functions *after* the standard library. Otherwise the library version will overwrite your version instead of the other way around.

An outline of the standard run system

The chain of execution follows the outline

```
run()
   stdinit()
      init()
         finitialize()
   continuerun() or steprun()
      step()
         advance()
            fadvance()
```

Each of these routines is very compact except for `continuerun()`, which employs rarely used graphical interface functions to optimize both simulation speed and graph line drawing so that the lines seem to be drawn in real time as the simulation progresses. Let's start with `fadvance()` and work up from there.

`fadvance()`

For now it suffices that `fadvance()` integrates all equations of the system from `t` to `t+dt` and then replaces the value of `t` by `t+dt`; we will examine the details of this later. The value of `dt` is either set by the user when the default fixed step integration method is used, or chosen by the integrator if the variable step method is used.

`advance()`

The `advance()` routine

```
proc advance() {
   fadvance()
}
```


provides the hook for doing any desired calculations before and/or after each time step. With the default fixed step method, anything is allowed. That is, we may change any state or any parameter, including `dt`. Each advance takes place as though it starts from a new initial condition without any previous history. Things are not so easy with the variable time step methods. Although it is safe to evaluate any variable and save it in an array or write it to a file, changing a parameter or state is not allowed unless we execute `cvode.re_init()` after the change. This is because CVODE saves state and derivative information from previous steps and assumes that all coefficients and states are differentiable up to its current order of accuracy. Changing a parameter or state constitutes a new set of equations, which constitutes a new problem. The only ways that time-varying parameters may be simulated with variable step methods is in the context of a model description or by using the interpolated form of `Vector.play()` (see **Time-dependent PARAMETER changes** in **Chapter 9**).

`step()`

`advance()` is called by the `step()` procedure, which is implemented as

```
proc step() {local i
   if (using_cvode_) {
      advance()
   } else for i=1,nstep_steprun {
      advance()
   }
   Plot()
}
```

The idea behind this function is that numerical accuracy may require a smaller time step than needed for plotting. That is, the interval between plots (call it `Dt`) is an integral multiple of the underlying `fadvance()` time step `dt`. This integral multiple is calculated

in a `setdt()` function which reduces `dt` if necessary to ensure that the `Dt` steps lie on a `dt` boundary. The RunControl panel has a field editor labeled Points plotted/ms which displays the value of the variable `steps_per_ms`. This value, along with `dt`, is used to calculate `nstep_steprun` and perhaps modify `dt` whenever either changes by calling `setdt()`. One can see that when CVODE is active, a step is just a single advance. At the end of a step, the `Plot()` procedure iterates over all the `Graph`s in the various plot lists that need to be updated during a simulation run. The purpose of these lists is detailed later in this chapter (see **Incorporating `Graph`s and new objects into the plotting system**).

Adding an object that can carry out certain specific methods to one of the `graphLists` can be an effective way to perform special tasks during a simulation. One advantage over replacing `proc step()` is that objects can automatically add themselves to, and remove themselves from, these lists.

`steprun()` and `continuerun()`

The `step()` procedure is called by the `continuerun()` and `steprun()` procedures. `steprun()` is

```
proc steprun() {
   step()
   flushPlot()
}
```

which implements the action for the Single Step button of the RunControl. It ensures that all the plot lists are flushed so that any deferred graph updates are performed.

`continuerun()` is called directly as an action by the Continue til and Continue for buttons in the RunControl. The actions are `continuerun(runStopAt)` and

`continuerun(t+runStopIn)` respectively. `continuerun()` is quite complex, and it is doubtful that anyone will want to replace it with something more complicated. It takes a single argument which is the time at which the integration should stop.

Before every `step()`, `continuerun()` checks to see if the `stoprun` variable is nonzero; if so it immediately breaks out of its loop. `continuerun()` sets `stoprun` to 0 on entry; `stoprun` is set nonzero if the user presses the Stop button on the RunControl. `stoprun` is a global variable in C so it can be checked by any C or C++ class that can carry out multiple runs and needs to properly clean up and return, e.g. optimization routines such as the praxis optimizer. In designing any class that manages a family of runs, one must decide what to do when the user presses Stop. If `stoprun` becomes nonzero but the class ignores it, the current simulation run will end and the next run in the family will start.

`continuerun()` uses the stopwatch to count the seconds in a variable called `realtime` while it is executing, and this value is displayed in the Real Time field editor. The resolution of the stopwatch is one second, and after each second the plots are flushed with a special method that avoids redrawing the portions of lines that are already plotted, all field editors are updated if the values they are watching have changed, and any outstanding events are handled (otherwise pressing the Stop button would have no effect). Actually, to give more rapid response to events, the `doEvents()` function is called at every step for the first two seconds and less often after that to avoid overhead if steps are very fast.

When `continuerun()` has reached its stopping time, a full flush of all the plots is done. Plots are flushed at intermediate times only if the variable `stdrun_quiet` is 0; this variable is toggled by the Quiet checkbox in the RunControl. Drawing plots on the screen is expensive and considerable speedup can often be seen if plotting is deferred to the end of a run. However, it often seems worth the penalty to view the progress of a simulation.

> "On one side hung a very large oil-painting so thoroughly besmoked, and every way defaced, that in the unequal cross-lights by which you viewed it, it was only by diligent study and a series of systematic visits to it, and careful inquiry of the neighbors, that you could any way arrive at an understanding of its purpose. Such unaccountable masses of shades and shadows, that at first you almost thought some ambitious young artist, in the time of the New England hags, had endeavored to delineate chaos bewitched. But by dint of much and earnest contemplation, and oft repeated ponderings, and especially by throwing open the little window towards the back of the entry, you at last come to the conclusion that such an idea, however wild, might not be altogether unwarranted."

`run()`

The `run()` procedure

```
proc run() {
   stdinit()
   continuerun(tstop)
}
```

is invoked as an action by the Init & Run button to initialize the system and integrate up to `tstop`, i.e. the value shown in the Tstop field editor of the RunControl. The initialization process is discussed at length in **Chapter 8**, but we should note that `stdinit()`

```
proc stdinit() {
   realtime=0
   startsw()
   setdt()
   init()
   initPlot()
}
```

calls `init()`

```
proc init() {
   finitialize(v_init)
   fcurrent()
}
```

which is generally the only function in the system that needs to be replaced in order to implement complex initialization strategies.

Details of `fadvance()`

The `fadvance()` function is implemented in `nrn.../src/nrnoc/fadvance.c`. In one form or another, `fadvance()` has always been the workhorse of the NEURON simulator, dating back to before NEURON's progenitor CABLE and even prior to the `hoc` interpreter, when all PDP8 FOCAL (FOrmula CALculator) functions had to begin with the letter `f`. One could easily do without an `finitialize()` function, since the interpreter overhead for computing steady states is small compared

> "From the chocks it hangs in a slight festoon over the bows, and is then passed inside the boat again; and some ten or twenty fathoms (called box-line) being coiled upon the box in the bows, it continues its way to the gunwale still a little further aft, and is then attached to the short-warp --the rope which is immediately connected with the harpoon; but previous to that connexion, the short-warp goes through sundry mystifications too tedious to detail."

to the computational effort of taking `tstop`/`dt` steps to do a simulation. But fast integration is most naturally carried out in compiled code, which is on the order of a hundred times faster than the interpreter.

Extending NEURON's numerical methods and simulation domain has been an incremental process carried out over several years. It may help to understand the current structure of `fadvance()` if we first consider how it evolved. The order of additions was CVODE (variable order, variable time step integrator), `NetCon` (event delivery system), `LinearMechanism` (overlay of algebraic equations onto the Jacobian), and DASPK (differential algebraic solver). Each major increase in functionality reused as much of the existing functions and program structure as possible, but a few functions needed small changes so they could support both the old and new methods. These increases in functionality also had to be usable with the least amount of effort on the part of the user. For example, turning variable time step integration on or off can be done by clicking on a checkbox in the NEURON Main Menu / Tools / VariableStepControl panel.

Our dissection of `fadvance()` follows its evolution by

- reviewing the details of what happens during classical fixed time step integration, i.e. the fully implicit (backward Euler) and Crank-Nicholson methods. Topics examined include the strategies that account for NEURON's reputation for speed:
    1. exploiting the tree topology of the branched nerve equations. Tree topologies require exactly the same number of add/multiply/divide operations as a single unbranched cable.

2. using a staggered time step to avoid Newton iterations of HH-like nonlinear channels. This gives the second order Crank-Nicholson method the same performance per time step as the first order implicit method.
3. using rate tables involving the value of `dt`. This optimizes the analytic integration of channel states by trivial assignment statements like `m=m+mexp*(minf-m)`.

- discussing the variable time step, variable order ordinary differential equation solvers.
- walking through the operation of the local variable time step method to learn how it works and how it handles discrete events.

Many of these items are closely related to each other, so we must occasionally mention later additions to complete the discussion of earlier ones.

The fixed step methods: backward Euler and Crank-Nicholson

It is easiest to understand the reasons for the particular sequence of actions if we focus on the second order correct Crank-Nicholson method (CVODE is inactive and the global variable `secondorder` has the value of 2). Assume that, on entry to `fadvance()`, the value of `t` is `tentry`, the voltages are second order correct at `tentry`, and the gating states are second order correct at `tentry` + `dt`/2. The last assumption may seem odd, but we will see how it helps accelerate integration.

When the Crank-Nicholson method is chosen, the purpose of `fadvance()` is to integrate the voltages and states such that, on exit from `fadvance()`,

`t` = `tentry` + `dt` (call this `texit`)

`v` and concentrations are second order correct at `texit`

gating states are second order correct at `texit` + `dt`/2

and as a side effect

ionic currents are second order correct at `texit` - `dt`/2

Notice that these exit conditions satisfy the entry conditions for a subsequent call to `fadvance()`.

One might object that the entry assertions are not satisfied at `t` = 0 since the gating states are second order correct at time 0, not time `dt`/2. We'll discuss this in detail, however second order correctness refers to the integrated error over a specific time interval `Dt` as more and more `dt` steps are used. The local error over a single `dt` step for second order correctness is proportional to $\texttt{dt}^3$ and for first order correctness it is $\texttt{dt}^2$. So as long as d*state*/d*t* = 0 at *t* = 0, as it must be in the steady state, the error associated with using *state*(*t* = 0) as the value of *state*(*t* = `dt`/2) is itself proportional to $\texttt{dt}^2$ and is a once-only error which does not accumulate for each `dt` time step. If non-steady state initializations are performed, then the gating states should be adjusted to their values according to *state* = *state* + d*state*/d*t* · `dt`/2.

For the default backward Euler and Crank-Nicholson methods, the sequence of operations carried out by `fadvance()` is

1. Check to see if any voltages or other variables that are sources for `NetCon` objects have reached threshold. Deliver any discrete events whose delivery time is earlier

than `tentry+dt/2`. With fixed step methods, events necessarily lie on time step boundaries, so this certainly delivers all events outstanding at time `tentry`.

The function that carries this out (`NetCvode::deliver_net_events()` in `nrn.../src/nrncvode/netcvode.cpp`) first appends the value of `state` at `tentry` to the corresponding `Vector` according to the list defined by `cvode.record(&state, vec, tvec)` statements. This list is most useful with the local variable step method; indeed, this is the only meaningful way at this time to retrieve results from a simulation that uses local variable time steps, since `t` values on return from a sequence of `fadvance()` calls are not monotonic and only a small fraction of states (the states in only one cell) is integrated on a single `fadvance()` (see **Local time step integration with discrete events** below). Of course, `cvode.record()` also works with the fixed step methods.

As of version 5.4, `Vectors` that are played or recorded at specific times are handled as a sequence of discrete events.

2. When `Vector.play()` is treated as an interpolated (continuous) function, values are interpolated at time = `tentry+dt`/2. The syntax `Vector.play(&var)`, which has no specific time `Vector` or declared play interval, cannot be used by variable step methods and is therefore deprecated. However, in case you find it in old code, we mention that `Vector.play(&var)` makes `var` receive its value from the *next* `Vector` element; thus the first `fadvance()` after `finitialize()` will assign `Vector.x[1]` to `var`.

3. The matrix equation for voltage is set up with the global variable `t` = `tentry`+`dt`/2. This is done by calling the function `setup_tree_matrix()` in `nrn.../src/nrnoc/treeset.c`. Prior to version 5, NEURON was limited, as the names of this function and file imply, to coupled voltage equations with the topology of a tree, i.e. each voltage node had at most one parent node. This is not only well-matched to neuronal structure, but also has the attractive property that solution of linear equations with this structure by Gaussian elimination takes exactly the same number of arithmetic operations as if the equations had the topology of an unbranched cable with the same number of nodes. It is the tree structure which makes the simulation time proportional to the number of voltage nodes. Speed suffers when the topology is not equivalent to a tree, e.g. when gap junctions, linear circuits, or the extracellular mechanism is present. Completely general graph structures have a worst case Gaussian elimination time which is proportional to the cube of the number of voltage nodes.

   The purpose of the `setup_tree_matrix()` function is to create the algebraic equation for each node. In abstract terms we are setting the problem up as a matrix equation in the form

$$\mathbf{M}\ \mathbf{v}(t_{entry} + \Delta t) = \text{r.h.s.} \qquad \text{Eq. 7.1a}$$

   ("r.h.s." = right hand side) for the backward Euler method, or

$$\mathbf{M}\ \mathbf{v}(t_{entry} + \frac{\Delta t}{2}) = \text{r.h.s.} \qquad \text{Eq. 7.1b}$$

for the Crank-Nicholson method. Tree structures are very similar to tridiagonal cable equations. For unbranched cables the most straightforward description of the spatially discrete cable equation has a row structure

$$b_i v_{i-1} + d_i v_i + a_i v_{i+1} = \text{r.h.s.}_i \qquad \text{Eq. 7.2}$$

and each coefficient and variable in the row is kept in a node structure (*b*, *d*, and *a* are the subdiagonal ("*b*elow"), *d*iagonal, and supradiagonal ("*a*bove") elements of **M**). Generalization to a tree preserves the association of *b*, *d*, *v*, and r.h.s. in the node equation. The only change is that `Node.a` (see next paragraph) refers to the matrix element in the parent node equation.

Setup of the matrix equations begins by first checking a flag to see if any diameters or section lengths have changed, and if so, recalculating the two connection coefficients between a node and its parent. These connection coefficients are both stored in the node. `Node[i].b` is the resistance between node `i` and its parent divided by the area of the node. `Node[i].a` is the same thing but divided by the area of the parent node. Next, the *d* and r.h.s. elements of all nodes are set to zero in preparation for incrementally adding conductance and current contributions to them. The *a* and *b* elements of the matrix generally do not change during a simulation. Fortunately, they are not destroyed during Gaussian elimination and so only need to be computed when the morphology changes.

At this point the membrane current and conductance contributions to the node equations are added to r.h.s. and *d* respectively. This is done by calling the `nrn_cur` functions of every mechanism in every node (pointers to these functions are kept in

the `memb_func[type].current` structure). These functions are the model description translation of the `BREAKPOINT` block. The most common usage of the `BREAKPOINT` block in a model description is to calculate channel currents from the values of `STATE` variables and membrane potential `v` (see **Chapter 9**). In the translation of a `BREAKPOINT` block, the `SOLVE` statement information, which tells how to integrate the `STATE` variables, is segregated into a `nrn_state` function (see step 6 below), and the remaining statements are used to construct a `nrn_current` function which takes voltage as an argument. The `nrn_current` function is called twice by the `nrn_cur` function, once with an argument of `v` + 0.001 and then with an argument of `v`, in order to calculate the numerical derivative *di/dv* as well as the current. The `nrn_cur` function then adds the *di/dv* value to the diagonal element `Node.d` (i.e. the diagonal element of the Jacobian) and the value of `-i` to the right hand side element `Node.rhs`. The form of this expression follows from the current conservation equation evaluated at $t + \Delta t$

$$C\frac{\Delta v_i}{\Delta t} + \frac{di_i}{dv_i}\Delta v_i - \sum_j \frac{\Delta v_j - \Delta v_i}{area_i\ r_{ij}} = -i_i(v_i(t)) + \sum_j \frac{v_j - v_i}{area_i\ r_{ij}} \qquad \text{Eq. 7.3}$$

where

$$i_i(v_i(t+\Delta t)) = i_i(v_i(t)) + \Delta v_i \frac{di_i}{dv_i} \qquad \text{Eq. 7.4}$$

All terms that are proportional to Δv go into the matrix (left) side of Eq. 7.1, and all constant terms or product terms of $v(t)$ go into the right hand side. If Δv_j refers to the

parent of node *i*, the coefficient $1/area_i\, r_{ij}$ is the *i*th node's *b* element (see Eq. 7.2); if Δv_j refers to a child, the coefficient is the child node's *a* element.

4. The `nrn_solve()` function in `nrn.../src/nrnoc/solve.c` is called to solve the voltage node equations. Normally these equations are tree-structured, which allows use of triangularization and back substitution functions that are specifically crafted to minimize pointer arithmetic overhead by taking advantage of the details of our `Node` structure in `nrn.../src/nrnoc/section.h`. This step executes approximately twice as fast as the more general sparse matrix Gaussian elimination package necessary for non-tree structures. However this has less significance than it appears since Gaussian elimination of tree structures takes much less than half the time required to set up equations containing channel currents. On exit from `nrn_solve()` the r.h.s. field of the `Node` structures contains the values of Δv.

   If `secondorder` is 2 then the currents are updated with a call to `second_order_current`, which uses `di_ion/dv` along with Δv to compute the second order correct ionic currents at `tentry`+`dt`/2. Therefore when `fadvance()` returns and `t` is `tentry + dt`, the ionic currents are second order correct at `t - dt`/2. Note that individual currents associated with particular channel mechanisms and available to the interpreter as `ASSIGNED` variables are not updated to be second order correct. That is, individual model description current variables are approximated by `g(texit - dt/2)*(v`

> Nowadays, voltage clamp models are best implemented as linear mechanisms. Voltage and current states in such a model are computed simultaneously with the membrane potential, so the issues associated with staggered time steps do not arise.

`(texit-dt) - erev)`. Without special attention to this problem, model descriptions of voltage clamp currents that are appropriate for the internal use made of them during `fadvance()` would be complete nonsense when plotted, since they do not take into account the large change between `v(texit-dt)` and `v(texit)`. For this reason, particularly stiff models, such as voltage clamps, are careful to recalculate the current variable within the block called by the `BREAKPOINT`'s `SOLVE` statement (see step 6 below), which occurs when the voltage values are at `texit`.

For fixed step methods, one should always compare plots of individual model current and conductance variables with their time courses computed with smaller `dt`. In some cases it may be useful for plotting to introduce a `FUNCTION` into the channel model which uses the present values of `t`, `v`, and `STATE`s to return the consistent first order values of those currents. Equivalently, one could call `fcurrent()` on return from `fadvance()` (`fcurrent()` carries out step 3) to reevaluate the currents and conductances at the present values of `t`, `v`, and `STATE`s.

With the variable step methods (see below), all variables have their appropriate values at `texit`. One of the most significant benefits of the variable time step methods is the ease of plotting current and conductance variables at the accuracy of the underlying computation.

5. The voltages are updated using the equation `v` = `v` + r.h.s. for the backward Euler method and `v` = `v` + 2 r.h.s. for the Crank-Nicholson method. The global variable `t` is set to `tentry` + `dt`.

6. `nonvint()` is called, which integrates all states EXCEPT the voltages. This is done by executing the `nrn_state` function for every mechanism in every segment of every section (pointers to these functions are kept in the `memb_func[type].state` structure). These functions are the model description translation of the `SOLVE` statement in the `BREAKPOINT` block. Since `v` is now at `tentry` + `dt`, or the midpoint of the integration interval from `tentry` + `dt` + `dt`/2, second order correct integration schemes that treat `v` as a constant in the integration interval remain second order correct. Specifically, the analytic integration of Hodgkin-Huxley-like channel gating states, e.g.

$$m(t + \frac{\Delta t}{2}) = m(t - \frac{\Delta t}{2}) + (1 - e^{-\Delta t \,/\, tau(v(t))}) \, (m_\infty(v(t)) - m(t - \frac{\Delta t}{2})) \qquad \text{Eq. 7.5}$$

where $v(t)$ is assumed constant, is second order correct for smooth functions of v. It should be remembered, however, that the calculation of m is only first order correct with the fixed step method (i.e. backward Euler) since the value of v itself is only first order correct.

When fixed step methods were used exclusively, it was common practice to factor the integration statement into the form

```
m = m + mexp(v)*(minf(v)-m)
```

where `mexp` and `minf` were calculated with fast interpolated table lookup. However, since the `mexp` table is dependent on the value of `dt`, this no longer works with variable step methods. Of course, `minf` and `mtau` could still be stored in tables, but

the speedup is marginal, and in these days of fast floating point processors, `minf` and `mtau` have to be quite complicated to justify the use of tables.

7. All variables that are recorded due to `Vector.record(&variable)` statements (i.e. without an associated sampling interval or `Vector` of recording times) are stored in the `Vector` elements associated with time `tentry` + `dt`. Starting with version 5.4, sampling times specified by a sampling interval or `Vector` of recording times are handled by the discrete event system.

Adaptive integrators

Our chief aim here is to see how adaptive integration operates in the context of a simulation, and in particular how it fits in with the event delivery system. Mathematical aspects of adaptive integration are discussed more thoroughly in **Chapter 4**.

Adaptive integrators adjust the time step and order of integration so that the local error for each state is less than a user-specified tolerance. For a given `dt` they are three times slower than the fixed step methods, because calculating the local error involves a lot of overhead and it is no longer is possible to use `dt`-dependent rate tables or avoid Newton iterations. However, the time step can be so large during interspike intervals that total run time is often almost an order of magnitude faster than with fixed step methods yielding the same accuracy. From the user's perspective, a potentially more important advantage of adaptive integration is that it eliminates the need for trial and error adjustments of `dt` in order to achieve satisfactory accuracy; instead, one merely specifies the local step accuracy and the integrator does the rest.

In models that involve asynchronous events, adaptive integration can improve simulation accuracy by guaranteeing that all events occur at their specified times (see below), rather than being forced to a `dt` step boundary as they do with fixed time step methods. Furthermore, all variables are computed at the same model time, so there is no need to wonder whether to plot a variable at `t`, `t+dt/2`, or `t-dt/2` (see step 4 under **The fixed step methods: backward Euler and Crank-Nicholson** above).

Adaptive integration was first added to NEURON starting with CVODE (Cohen and Hindmarsh 1994; 1996) for global time steps in version 4.0, and this was extended to local time steps in version 4.1. The original CVODE required modifications in order to work with models that involved `at_time()` events, which were used to implement abrupt changes of a parameter or a state. A strategy for dealing with an event that occurs at t_{event} is to stop integration at t_{event}, change the parameters or states that are modified by the event, calculate a new initial condition at t_{event}, and then resume integration. However, the CVODE integrator had no provision for stopping at a specified time, so it needed custom revisions. DASPK (Brown et al. 1994), which was subsequently added to deal with models in which some states are determined by algebraic equations (e.g. extracellular fields or linear circuit elements), had a specifiable stop time beyond which the integrator would not proceed, so it had a very different way of handling `at_time()`. It would have been nice if DASPK could simply have replaced CVODE, but DASPK did not directly support the interpolation operation needed by the local step method, and it has even more overhead per step than CVODE. Therefore a significant amount of code was required to provide the logical machinery that would make all these different pieces

of the NEURON simulation environment work properly with each other, while at the same time allowing users to easily switch between the various integrators. The later addition of an delivery system to NEURON greatly increased the complexity of the code that ties all these pieces together.

This complexity has been much reduced in the most recent releases of NEURON by replacing CVODE and DASPK with CVODES and IDA of the SUNDIALS package (available from `http://www.llnl.gov/CASC/sundials/`). CVODES (Hindmarsh and Serban 2002) is similar to CVODE but accepts a `tstop` beyond which the solution will not proceed, and IDA (Hindmarsh and Taylor 1999) is a new Initial value Differential Algebraic solver version of DASPK which now does support the interpolation operation. However, for historical reasons the class that is used to manage adaptive integration in NEURON is called `CVode`, and in this book we often use the term "CVODE" as a generic reference to any of NEURON's adaptive integrators.

The normal CVODE integration step consists of a prediction followed by a correction. Generating the prediction involves an evaluation of $\boldsymbol{f}(\mathbf{y}, t)$ (see Eq. 4.28a and 4.29a) which consumes most of the computational effort in an integration step. When CVODE returns, all `STATE`s have the correct values at the new time, but the `ASSIGNED` variables (which include currents) still have their "predicted" values. Correcting the `ASSIGNED` variables requires another evaluation of $\boldsymbol{f}(\mathbf{y}, t)$, but this nearly doubles the total computational overhead per integration step. For many purposes the uncorrected values are sufficiently accurate, and tightening the error tolerance takes care of most cases when it is not. Future releases of NEURON will apply the correction by default but

may offer users the option of disabling the `ASSIGNED` variable correction with the extra call to $\boldsymbol{f}(\mathbf{y}, t)$ after a CVODE step.

Now we are ready to see how the solution proceeds when adaptive integration is used. We start with local variable step integration, and then briefly consider global step integration.

Local time step integration with discrete events

In local time step integration, an independent CVODE method is created for each cell, and the solution for each cell moves forward at its own pace. As with fixed time step integration, at the `hoc` level one repeatedly calls `fadvance()` to make the simulation progress in time. However, at any point in the simulation the cells are all at different times, managed by their individual CVODE instances, so `fadvance()` is not very useful as a means for governing the plotting or recording of data; instead, special CVODE-specific procedures are employed.

It is also not very useful to think about the process of integration in terms of `fadvance()` calls. For a much better understanding of what is going on, we will focus on the sequence of elementary actions, or "microsteps," that are applied to individual cells. There are three kinds of microsteps, and they are called initialize, advance, and interpolate because of how they affect each cell's time--but more about this shortly.

When local time steps are used, there is a queue of event times and a queue of cell times. The event times are the times at which events are to be delivered, and the cell times are the current times of each cell in the model. Executing a simulation consists of repeatedly checking these queues and dealing with whichever is earliest: the earliest

event or the earliest cell. If there is a tie, the event is handled first. Handling an event removes the event from the event queue, but when a cell is handled the cell is just put back into the cell queue with a new time.

Each cell has three variables, called `t0`, `t_`, and `tn`, that are related to the progress of the simulation in time. `t_` is the current time of the cell and determines its position in the cell queue; the significance of `t0` and `tn` will become clear in the next few paragraphs. Handling a cell involves carrying out a microstep, which leaves these variables in one of the configurations shown in Figure 7.2. For the purpose of illustration, we assume that before the microstep is taken, the cell starts with `t0`, `t_`, and `tn` as depicted in the top row of this figure.

1. **Initialize**: reset the integrator at time t and then return. Before an initialization, the user may assign any values whatever to the states and parameters. Those values, along with the equations, define a new initial value problem. After initialization `t0`, `t_`, and `tn` are all equal to `t`.

2. **Advance**: perform a normal integration step to some new time `t` and then return. This involves computing values for the `STATE`s and `ASSIGNED` variables at some new time `t`, updating `t0` to the old `tn`, and making `t_` and `tn` equal to the new `t`.

3. **Interpolate**: return just before the time t_{event} of the next event. On exit from `fadvance()`, `t_` lies between `t0` and `tn` with a value equal to t_{event}. `STATE` values at `t_` are calculated from their values at `tn`, `t0`, and prior solution points according to CVODE's interpolation formulas (this is much less costly than a numeric integration step). If an integration step carries `tn` past the time of an event, or if a new event

arrives with $t_{event} < t\_$, interpolation will be applied so that t_ retreats to t_{event}. However, a cell can't retreat to a time earlier than its t0. If there are multiple cells, the largest t0 is the "least event time," i.e. the time before which *no* cell can retreat.

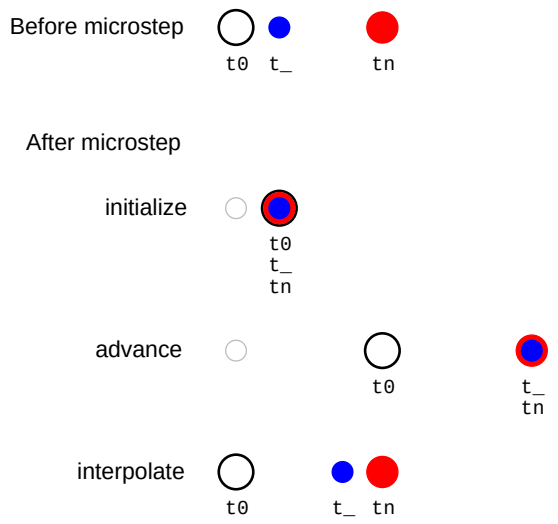


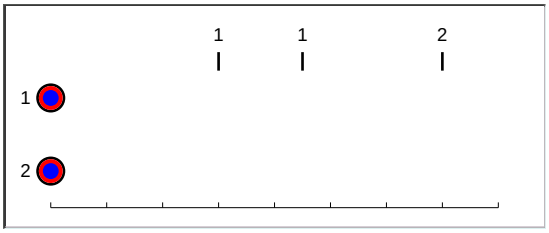
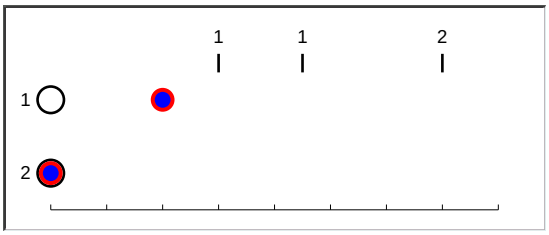
Fig. 7.2. After a microstep, the relative positions of t0 (black open circle), t_ (blue dot), and tn (red filled circle) in time depends on whether the microstep performed an initialization, a normal integration step, or an interpolation to just before the next event. The small grey circle after initialize and advance marks the former location of t0. Time increases toward the right in each row.

Note that the STATE and ASSIGNED values at t_ and tn are "tentative" because an event may arrive in the [t0, tn] interval that requires a new initialization and forces the solution into a new trajectory. The values at t0 are "real" in the sense that a cell cannot retreat to a time earlier than its own t0.

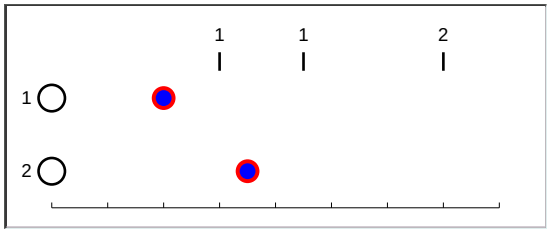
If multiple events occur at the same time, they are all handled. If more than one of these requires an initialization, the initialization is deferred until after all simultaneous events are handled. Thus if there are 4 events at the same time and 3 of them require

initialization, each event will be handled but there will be only one initialization, which is performed after all four have been handled.

To make this more concrete, let's walk through a hypothetical simulation of a small network model using the local variable time step method. This model has two neurons called 1 and 2. A `NetCon` delivers events to an excitatory synapse on cell 1, and cell 1 projects via another `NetCon` to a synapse on cell 2. In the following discussion the "step" number refers to how many microsteps have been taken, the "action" is what kind of microstep it was, and the "outcome" is a diagram that shows the relative positions in time of events and each cell's `t0`, `t_`, and `tn`.

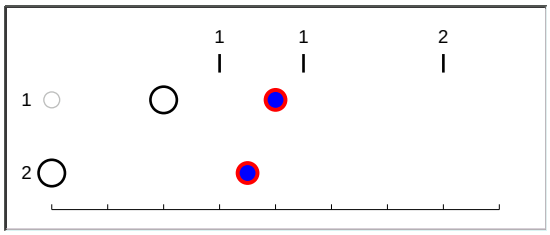
Step, action, and outcome	**Comments**
0. Initialize the model 	This is done by `finitialize()`. Notice that `t0` = `t_` = `tn` = `t` = 0 ms. Also three events are placed in the event queue, two for cell 1 and one for cell 2, at the indicated times. There are no events at `t` = 0 ms . . .
1. Advance cell 1 	. . . so the first microstep *advances* one of the cells. For the sake of illustration, we'll say it advances cell 1. This makes 2 the earliest cell.

2. Advance cell 2



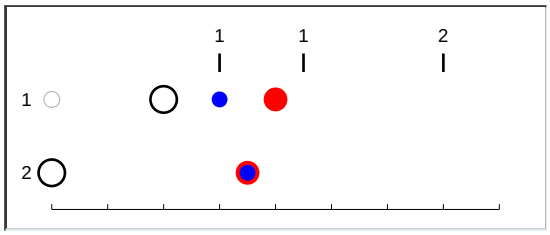
Cell 2's `t_` and `tn` move past the earliest event, but that's OK because the event isn't for cell 2. Cell 1 is now earliest.

3. Advance cell 1



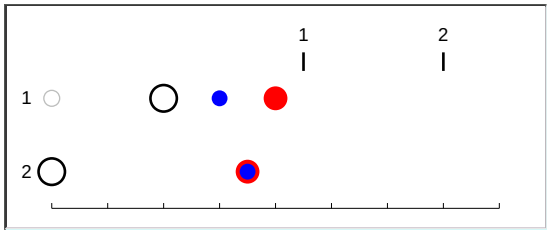
Cell 1's `t_` and `tn` move to a new time. Notice how `t0` follows behind `tn`, jumping from its original location (marked by the small "ghost" circle) to the prior location of `tn`. But also notice that `t_` has moved past an event for cell 1.

4. Interpolate cell 1



Cell 1's `t_` retreats to the event time, and its `STATE`s at `t_` are calculated by interpolation. We are ready to handle the event.

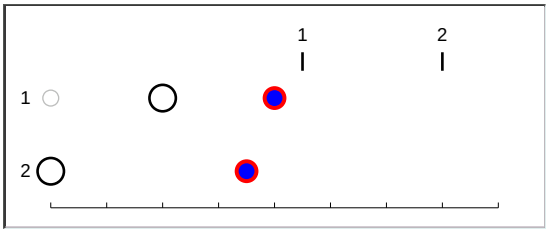
Handle the event



Handling the event removes it from the event queue.

Cell 1 is still earliest. Let's say the event we just handled didn't do anything to cell 1 that forces initialization . . .

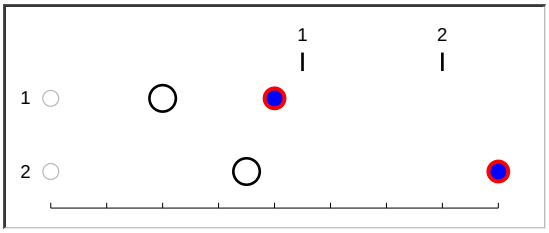
5. Interpolate cell 1



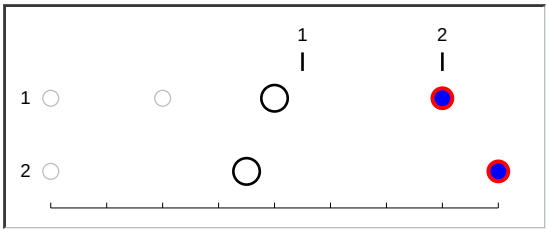
. . . so cell 1's trajectory isn't affected. There are no events between its current time `t_` and `tn`, so `t_` can be moved right up to `tn`, as shown here. Technically speaking this is an "interpolation" even though no real calculations are involved.

The earliest cell is now cell 2.

6. Advance cell 2

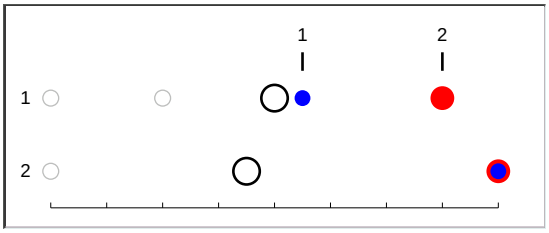


7. Advance cell 1



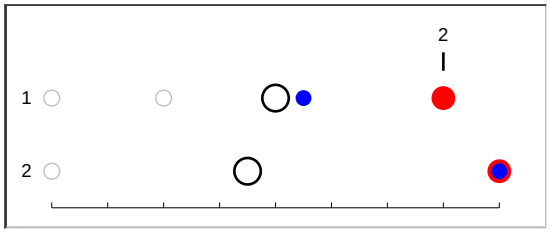
We have seen this before.

8. Interpolate cell 1



It is now time to deal with the event . . .

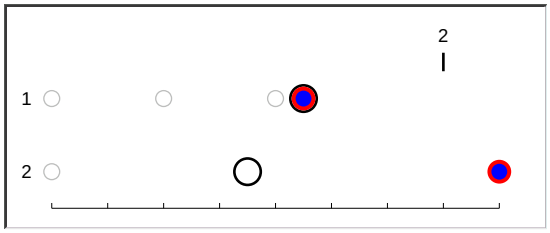
Handle the event



. . . which removes it from the queue.

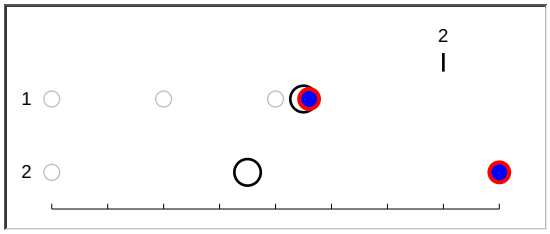
And it's also time to introduce a little excitement. Unlike the first event, which didn't affect cell 1's trajectory, we'll stipulate that this one was delivered to the excitatory synaptic mechanism on cell 1 by a `NetCon` with a strong positive weight, causing an abrupt change in one of the that mechanism's parameters. This means the next microstep has to *initialize* cell 1.

9. Initialize cell 1



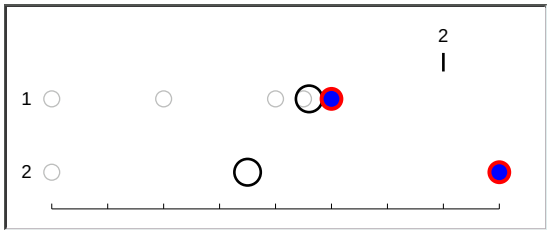
Notice that cell 1's `t0`, `t_` and `tn` are exactly at the handled event time.

10. Advance cell 1

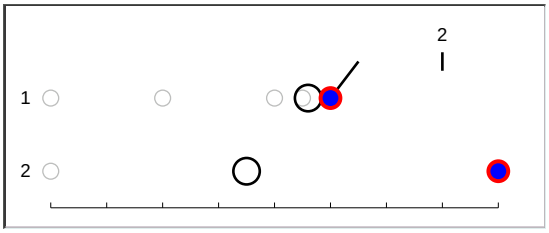


The strong synaptic input drives cell 1 toward firing threshold. Since its membrane potential is changing rapidly, satisfying the error criterion requires short advances.

11. Advance cell 1

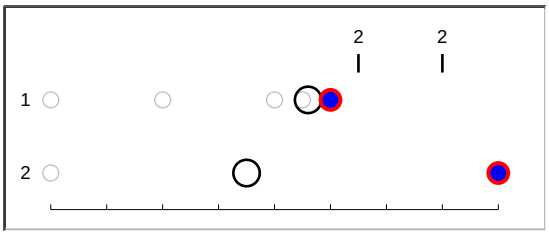


Cell 1 generates a spike event . . .



That last advance took cell 1 over the threshold of the `NetCon` that monitors its membrane potential.

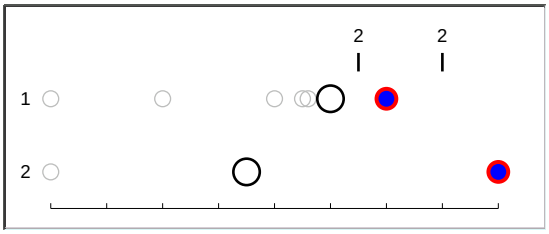
. . . which is inserted into the event queue



The spike event will be delivered to the synapse on cell 2 at the new time indicated in this figure.

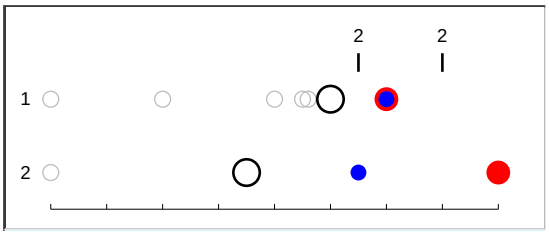
Cell 1 is the earliest cell now . . .

12. Advance cell 1



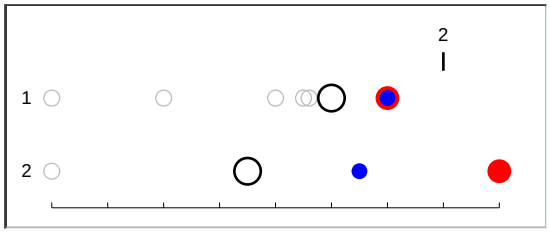
. . . and again. But it has moved past the spike event for cell 2, so that becomes the next thing to deal with.

13. Interpolate cell 2



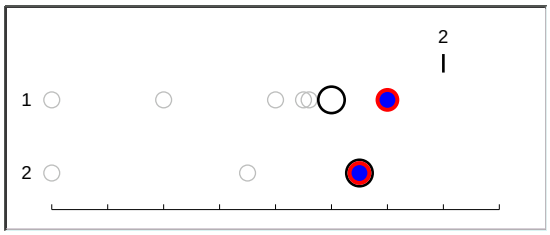
Cell 2 retreats to the time of its event.

Handle the event



The event disappears from the event queue.

14. Initialize cell 2



The event caused an abrupt change in a variable in cell 2's synapse, requiring initialization.

From the user's standpoint, this is all easier done than said, thanks to the behind-the-scenes coordination of adaptive integration and discrete events in NEURON.

Since the values calculated at `t_` and `tn` are only tentative, the solution trajectory for any cell is defined by its sequence of `t0`s paired with the variables that were computed at those times. The `CVode` class's `record()` method captures the stream of `t0`s into one vector and the values of a user-specified range variable into another vector. Currently, plotting of trajectories is controlled at the `hoc` level in the `run()` procedure on return from `fadvance()`. To allow normal plotting of variables with local variable time steps, in the next version of NEURON each variable that is plotted will be associated with a specific cell so that it can be plotted when `t0` for that cell advances.

Global time step integration with discrete events

Global time step integration uses only one CVODE method for the entire model, so in a sense it is just a degenerate case of what happens with local time steps. More particularly, in the local variable step method a call to `fadvance()` produces a microstep, but in the global step method calling `fadvance()` results in one or more microsteps arranged so that time increases monotonically. In fact, the global step method is analogous to fixed step integration in that `fadvance()` returns before an initializing event, after an initialization, and after a regular integration. Furthermore, on return from `fadvance()` there is never an outstanding event earlier than time `t`, and `t_` is always identical to `t`. Since time increases monotonically, and all cells are at the same time, recording and plotting variables with the global step method is much more straightforward than with local time steps.

Incorporating `Graph`s and new objects into the plotting system

Objects that need to be notified at every step of a simulation are appended to one of six lists. The first four lists are referenced by `graphList[n_graph_lists]` and their normal contents are `Graph` objects that plot variables requested by each `Graph`'s `addexpr` or `addvar` statement. Variables are plotted as line drawings in which the abscissa is related to `t` and the ordinate is the magnitude of the variable. `Graph`s are

added to these four lists when one of the buttons of the NEURON Main Menu / Graph menu titled Voltage axis, Current axis, State axis, and Phase Plane is pressed.

For	each variable is plotted vs.
`graphList[0]`	`t`
`graphList[1]`	`t-0.5*dt`
`graphList[2]`	`t+0.5*dt`
`graphList[3]`	an arbitrary function of `t` called an x-expression

The most useful of these lists is `graphList[0]`, which is recommended for all line drawings. `graphList[1]` and `[2]` are useful only to provide second order correct plots of ionic currents and state variables, respectively, when the Crank-Nicholson method has been selected through the variable `secondorder=2`. The offset is meaningless when the default first order method is used (`secondorder=1`) because first order accuracy holds at all instants in the interval `[t-0.5*dt, t+0.5*dt]`. When the variable time step methods are chosen, all variables are computed at the same `t` so the offset is 0 and the `[1]` and `[2]` `graphList` lists are identical to `graphList[0]`.

The remaining two lists whose object elements are notified at every step are called `flush_list` and `fast_flush_list`. The first is for `Graph`s that plot `Vector`s that may change every time step. These do the `Vector` movies and Space Plots requested from a Shape plot. The `fast_flush_list` is for Shape Plots or Hinton plots in which it is not necessary to redraw an entire cell or pattern because only a few rectangles change color during each step.

Plots are initialized by a call from `stdinit()` to `initPlot()`. The `initPlot()` procedure first removes any objects in the graph or flush lists for which there is no view on the screen by checking the return value of the `view_count()` method of the objects,

and then calls the `begin()` method for all objects in the `graphLists`. Finally, it calls the `Plot()` and `flushPlot()` procedures to plot things properly at `t=0`.

The `Plot()` procedure is called at the end of each step. `Plot()` calls `plot(t)` for the `graphList` objects (actually the previously discussed offsets may be used for `graphList[1]` and `[2]`). If `stdrun_quiet` is 0, `Plot()` also calls `begin()` and `flush()` methods for items in the `flush_list` so that any `Vector` plots are updated. Lastly it calls the `fast_flush()` method for each item in the `fast_flush_list` so that any color changes are seen on the screen.

During `continuerun()`, the `fast_flushPlot()` procedure is called once at every second of simulation time and the `flushPlot()` procedure is called at the end. `fast_flushPlot()` calls the `fast_flush()` method for each item in the four `graphLists`. This special call is very efficient for time plots because it erases and redraws only the portion of the lines that accumulated since the last `fast_flush`. Otherwise, damaging a small part of a line entails damaging the entire bounding box of the line, which implies damaging all the lines that intersect the bounding box, which ends up damaging the entire canvas and consequently requires erasing and redrawing everything on the canvas. `flushPlot()` calls the `flush()` method for each item in all six lists, which ends up redrawing everything in every canvas. While this is expensive, the screen accurately reflects exactly the internal data structures of the lines and shapes.

A `Graph` object constructed by the user with

```
objref g
g = new Graph()
```

can be added to the standard run system with


```
graphList[0].append(g)
```

or perhaps even better with

```
addplot(g, 0)
```

since the latter will also set the abscissa to range from 0 to `tstop` (and the vertical axis from -1 to 1). Also, since the methods called on a `graphList` are `begin()`, `plot(t)`, `view_count()`, `fast_flush()`, `flush()`, and `size(x0, x1, y0, y1)`, any object that implements these functions, even as stubs, can be appended to `graphList[0]` in order to carry out calculations during a run. The SpikePlot of the NetGUI tool is implemented in just this way. This is an example of how the `hoc` interpreter provides a poor man's version of polymorphism; more information about object-oriented programming in `hoc` is presented in **Chapter 13**.

References

Brown, P.N., Hindmarsh, A.C., and Petzold, L.R. Using Krylov methods in the solution of large-scale differential-algebraic systems. *SIAM Journal of Scientific Computing* 15:1467-1488, 1994.

Cohen, S.D. and Hindmarsh, A.C. CVODE User Guide. Livermore, CA: Lawrence Livermore National Laboratory, 1994.

Cohen, S.D. and Hindmarsh, A.C. CVODE, a stiff/nonstiff ODE solver in C. *Computers in Physics* 10:138-143, 1996.

Hindmarsh, A.C. and Serban, R. User documentation for CVODES, an ODE solver with sensitivity analysis capabilities: Lawrence Livermore National Laboratory, 2002.

Hindmarsh, A.C. and Taylor, A.G. User documentation for IDA, a differential-algebraic equation solver for sequential and parallel computers: Lawrence Livermore National Laboratory, 1999.

Chapter 7 Index

Chapter 8

How to initialize simulations

In most cases, initialization basically means the assignment of values at time $t = 0$ for membrane potential, gating states, and ionic concentrations at every spatial position in the model. A model is properly initialized when clicking on the Init & Run button produces exactly the same results, regardless of previous simulation history. Of course we assume that model parameters have not changed between runs, and that any random number generator has been re-initialized with the same seed so that it produces the same sequence of "random" numbers. Models described by kinetic schemes require that each of the reactant states be initialized to some concentration. If linear circuits are involved, initial values must be assigned to voltages across capacitors and the internal states of operational amplifiers. For networks and other models that use the event delivery system, initialization also includes specifying which events are in transit to their destinations at time 0 (i.e. events generated, at least conceptually, at $t \leq 0$ for delivery at $t \geq 0$). Complex models often have complex recording and analysis methods, perhaps involving counters and vectors, and these may also need to be initialized.

State variables and STATE variables

In rough mathematical terms, if a system consists of *n* first order differential equations, then initialization consists in specifying the starting values of *n* variables. For the Hodgkin-Huxley membrane patch (only one compartment), these equations have the form

$$\frac{dv}{dt} = f_1(m, h, n, v) \qquad \text{Eq. 8.1a-d}$$

$$\frac{dm}{dt} = f_2(m, v)$$

$$\frac{dh}{dt} = f_3(h, v)$$

$$\frac{dn}{dt} = f_4(n, v)$$

so that, knowing the value of each variable at time *t*, we can specify the slope of each variable at time *t*. We have already seen (**Chapter 7**) that integration of these equations is an iterative process in which the purpose of an individual integration step (`fadvance()`) is to carry the system from time *t* to time $t + \Delta t$ using more or less sophisticated equations of the form

$$v(t+\Delta t) = v(t) + \Delta t \, \frac{dv(t^*)}{dt} \qquad \text{Eq. 8.2}$$

$$m(t+\Delta t) = m(t) + \Delta t \, \frac{dm(t^*)}{dt}$$

. . .

where the sophistication is in the choice of a value of t^* somewhere between t and $t + \Delta t$. However, regardless of the integration method, the iterative process cannot begin without choosing starting values for v, m, h, and n. This choice is arbitrary over the domain of the variables ($-\infty < v < \infty$, $0 \leq m \leq 1$, . . .), but once the initial v, m, h, and n are chosen, all auxiliary variables (e.g. conductances, currents, *d/dt* terms) at that instant of time are determined, and the equations determine the trajectories of each variable forever after. The actual evaluation of these auxiliary variables is normally done with assignment statements, such as

```
gna = gnabar*m*m*m*h
ina = gna*(v - ena)
```

This is why the model description language NMODL designates `gna` and `ina` as `ASSIGNED` variables, as opposed to the gating variables `m`, `h`, and `n`, which are the dependent variables in differential equations and are therefore termed `STATE` variables.

Unfortunately, over time an abuse of notation has evolved so that `STATE` refers to any variable that is an unknown quantity in a set of equations, and `ASSIGNED` refers to any variable that is not a `STATE` or a `PARAMETER` (`PARAMETER`s can be meaningfully set by the user as constants throughout the simulation, e.g. `gnabar`). Currently, within a single model description, `STATE` just specifies which variables are the dependent variables of `KINETIC` schemes, algebraic equations in `LINEAR` and `NONLINEAR` blocks, or differential equations in `DERIVATIVE` blocks. Generally the number of `STATE`s in a model description is equal to the number of equations. Thus, locally in a model description, the membrane potential `v` is never a dependent variable (the model description contains no equation that solves for its value) and it cannot be regarded as a

user-specified value. Instead, it is declared in model descriptions as an ASSIGNED variable, even though it is obviously a *state variable* at the level of the entire simulation. This abuse of terminology also occurs in linear circuits, where the potential at every node is an unknown to be solved and therefore a STATE. However, a resistive network does not add any differential equation to the system (although it adds algebraic equations), so those additional dependent variables do not strictly need to be initialized.

While STATE variables may be assigned any values whatever during initialization, in practice only a few general categories of custom initialization are used. Some of these are analogous to experimental methods for preparing a system for stimulation, e.g. letting the system rest without experimental perturbation, or using a voltage clamp or constant injected current to hold the system at a defined membrane potential--the idea is that the system should reach an unchanging steady state independent of previous history. It is from this steady state that the simulation begins at time t = 0. When there is no steady state, as for oscillating or chaotic systems, whatever initialization is ultimately chosen will need to be saved in order to be able to reproduce the simulation. More complicated initializations involve finding parameters that meet certain conditions, such as what value of some parameter or set of parameters yields a steady state with a desired potential. Some initial conditions may not be physically realizable by any possible manipulations of membrane potential. For example, with the hh model the h gating state has a steady state of 1 at large hyperpolarized potentials and the n gating state has a steady state of 1 at large depolarized potentials. It would therefore be impossible to reach a condition of h = 1 and n = 1 by controlling only voltage.

Basic initialization in NEURON: `finitialize()`

Basic initialization in NEURON is accomplished with the `finitialize()` function, which is defined in `nrn-x.x/src/nrnoc/fadvance.c` (UNIX/Linux). This carries out several actions.

1. `t` is set to 0 and the event queue is cleared (undelivered events from the previous run are thrown away).
2. Variables that receive a random stream (the list defined by `Random.play()` statements) are set to values picked from the appropriate random distributions.
3. All internal structures that depend on topology and geometry are updated, and chosen solvers are made ready.
4. The controller for `Vector.play()` variables is initialized. The controller makes use of the event delivery system for `Vector.play()` specifications that define transfer times for a step function in terms of `dt` or a time `Vector`.

   Events at time `t` = 0 (e.g. appropriate `Vector.play()` events) are delivered.
5. If `finitialize()` was called with an argument `v_init`, the membrane potential `v` in every compartment is set to the value `v_init` with a statement equivalent to

   ```
   forall for (x) v(x) = v_init
   ```
6. The `INITIAL` block of every inserted mechanism in every segment of every section is called. This includes point processes as well as distributed mechanisms (see **`INITIAL` blocks in NMODL** later in this chapter). The order in which mechanisms are

initialized depends on whether any mechanism has a `USEION` statement or `WRITE`s an ion concentration.

Ion initialization is performed first, including calculation of equilibrium potentials. Then mechanisms that `WRITE` an ion concentration are initialized; this necessitates recalculation of the equilibrium potentials for any affected ions. Finally, all other mechanism `INITIAL` blocks are called.

Apart from these constraints, the call order of user-defined mechanisms is currently defined by the alphabetic list of `mod` file names or the order of the `mod` file arguments to `nrnivmodl` (or `mknrndll`). However one should avoid sequence-dependent `INITIAL` blocks. Thus if the `INITIAL` block of one mechanism needs the values of variables another mechanism, the latter should be assigned before `finitialize()` is executed.

If extracellular mechanisms exist, their `vext` states are initialized to 0 before any other mechanism is initialized. Therefore, for every mechanism that computes an `ELECTRODE_CURRENT`, `v_init` refers to both the internal potential and the membrane potential.

`INITIAL` blocks are discussed in further detail below.

7. `LinearMechanism` states, if any, are initialized.
8. Network connections are initialized. This means that the `INITIAL` block inside any `NET_RECEIVE` block that is a target of a `NetCon` object is called to initialize the states of the `NetCon` object.

9. The `INITIAL` blocks may have initiated `net_send` events whose delay is 0. These events are delivered to the corresponding `NET_RECEIVE` blocks.

10. If fixed step integration is being used, all mechanism BREAKPOINT blocks are called (essentially equivalent to a call to `fcurrent()`) in order to initialize all assigned variables (conductances and currents) based on the initial STATE and membrane voltage.

    If any variable time step method is active, then those integrators are initialized. In this case, if you desire to change any state variable (here "state variable" means variables associated with differential equations, such as gating states, membrane potential, chemical kinetic states, or ion concentrations in accumulation models) after `finitialize()` is called, you must then call `cvode.re_init()` to notify the variable step methods that their copy of the initial states needs to be updated. Note that initialization of the differential algebraic solver IDA consists of two very short (`dt` = 10^{-6} ms) backward Euler time steps in order to ensure the validity of $\boldsymbol{f}(\mathbf{y}', \mathbf{y}) = \mathbf{0}$.

11. `Vector` recording of variables using the list defined by `cvode.record(&state, vector)` statements is initialized. As discussed in **Chapter 7** under **The fixed step methods: backward Euler and Crank-Nicholson**, `cvode.record()` is the only good way of keeping the proper association between local step state value and local `t`.

12. `Vectors` that record a variable, and are in the list defined by `Vector.record()` statements, record the value in `Vector.x[0]`, if `t` = 0 is a requested time for recording.

Default initialization in the standard run system:

`stdinit()` and `init()`

The standard run system's default initialization takes effect when you enter a new value for `v_init` into the field editor next to the RunControl panel's Init button, or when you press either RunControl panel's Init or Init & Run button. These buttons do not call the `init()` procedure directly but instead execute a procedure called `stdinit()` which has the implementation

```
proc stdinit() {
  realtime=0  // "run time" in seconds
  startsw()   // initialize run time stopwatch
  setdt()
  init()
  initPlot()
}
```

`setdt()` ensures (by reducing `dt`, if necessary) that the points plotted fall on time step boundaries, i.e. that `1/(steps_per_ms*dt)` is an integer. The `initPlot()` procedure begins each plotted line at `t` = 0 with the proper *y* value.

The default `init()` procedure itself is

```
proc init() {
  finitialize(v_init)
  // User-specified customizations go here.
  // If this invalidates the initialization of
  // variable time step integration and vector recording,
  // uncomment the following code.
  /*
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
  */
}
```


Custom initialization is generally accomplished by inserting additional statements after the call to `finitialize()`. These statements often have the effect of changing one or more state variables, i.e. variables associated with differential equations, such as gating states, membrane potential, chemical kinetic states, or ion concentrations in accumulation models. This invalidates the initialization of the variable time step integrator, making it necessary to call `cvode.re_init()` to notify the variable step integrator that its copy of the initial states needs to be updated. If instead fixed step integration is being used, `fcurrent()` should be called to make the values of conductances and currents consistent with the new states. Changing state variables after calling `finitialize()` can also cause incorrect values to be stored as the first element of recorded vectors. Adding `frecord_init()` to the end of `init()` prevents this.

`INITIAL` blocks in NMODL

`INITIAL` blocks for channel models generally set the gating states to their steady state values with respect to the present value of `v`. Hodgkin-Huxley style models do this easily and explicitly by calculating the voltage sensitive alpha and beta rates for each gating state and using the two state formula for the steady state, e.g.

```
PROCEDURE rates(v(mv)) {
  minf = alpha(v)/(alpha(v) + beta(v))
  . . .
}
```

and then

```
INITIAL {
  rates(v)
  m = minf
  . . .
}
```


When channel models are described by kinetic schemes, it is common to calculate the steady states with the idiom

```
INITIAL {
  SOLVE scheme STEADYSTATE sparse
}
```

where `scheme` is the name of a `KINETIC` block. To place this in an almost complete setting, consider this implementation of a three state potassium channel with two closed states and an open state:

```
NEURON {
  USEION k READ ek WRITE ik
}

STATE { c1 c2 o }

INITIAL {
  SOLVE scheme STEADYSTATE sparse
}

BREAKPOINT {
  SOLVE scheme METHOD sparse
  ik = gbar*o*(v - ek)
}

KINETIC scheme {
  rates(v) : calculate the 4 k rates.
  ~ c1 <-> c2 (k12, k21)
  ~ c2 <-> o ( k2o, ko2)
}
```

(the `rates()` procedure and some minor variable declarations are omitted for clarity). As mentioned earlier in **Default initialization in the standard run system: `stdinit()` and `init()`**, when initialization has been customized so that states are changed after `finitialize()` has been called, it is generally useful to call the `fcurrent()` function to make the values of all conductances and currents consistent with the newly initialized states. In particular this will call the `BREAKPOINT` block (twice, in order to compute the Jacobian (*di/dv*) elements for the voltage matrix equation) for all mechanisms in all

segments, and on return the ionic currents such as `ina`, `ik`, and `ica` will equal the corresponding net ionic currents through each segment.

Default vs. explicit initialization of STATEs

In model descriptions, a default initialization of the `STATE`s of the model occurs just prior to the execution of the `INITIAL` block. However, this default initialization is rarely useful, and one should always explicitly implement an `INITIAL` block. If the name of a `STATE` variable is `state`, then there is also an implicitly declared parameter called `state0`. The default value of `state0` is specified either in the `PARAMETER` block

```
PARAMETER {
  state0 = 1
}
```

or implicitly in the `STATE` declaration with the syntax

```
STATE {
  state START 1
}
```

If a specific value for `state0` is not declared by the user, `state0` will be assigned a default value of 0. `state0` is not accessible from the interpreter unless it is explicitly mentioned in the `GLOBAL` or `RANGE` list of the `NEURON` block. For example,

```
NEURON {
  GLOBAL m0
  RANGE h0
}
```

specifies that every `m` will be set to the single global `m0` value during initialization, while `h` will be set to the possibly spatially-varying `h0` values. Clarity will be served if, in using the `state0` idiom, you explicitly use an `INITIAL` block of the form


```
INITIAL {
  m = m0
  h = h0
  n = n0
}
```

Ion concentrations and equilibrium potentials

Each ion type is managed by its own separate ion mechanism, which keeps track of the total membrane current carried by the ion, its internal and external concentrations, and its equilibrium potential. The name of this mechanism is formed by appending the suffix `_ion` to the name of the ion specified in the `USEION` statement. Thus if `cai` and `cao` are integrated by a model that declares

```
USEION ca READ ica WRITE cai, cao
```

there would also be an automatically created mechanism called `ca_ion`, with associated variables `ica`, `cai`, `cao`, and `eca`. The initial values of `cai` and `cao` are set globally to the values of `cai0_ca_ion` and `cao0_ca_ion`, respectively (see ***Initializing concentrations in*** **hoc** below).

> Since calcium currents, concentrations, and equilibrium potentials are managed by the `ca_ion` mechanism, one might reasonably ask why we can refer to the short names `ica`, `cai`, `cao` and `eca`, rather than the longer forms that include the suffix `_ion`, i.e. `ica_ca_ion` etc.. The answer is that there is unlikely to be any mistake about the meaning of `ica`, `cai`, . . . so we might as well take advantage of the convenience of using these short names.

Prior to version 4.1, model descriptions could not initialize concentrations, or at least it was very cumbersome to do so. Instead, the automatically created ion mechanism would initialize the ionic concentration adjacent to the membrane according to global variables. The reason that model mechanisms were not allowed to specify ion variables

(or other potentially shared variables such as `celsius`) was that confusion could result if more that one mechanism at the same location tried to assign different values to the same variable. The unintended consequence of this policy is confusion of a different kind, which happens when a model declares an ion variable, such as `ena`, to be a `PARAMETER` and attempts to assign a value to it. The attempted assignment has no effect, other than to generate a warning message. Consider the mechanism

```
NEURON {
  SUFFIX test
  USEION na READ ena
}

PARAMETER {
  ena = 25 (mV)
}
```

When this model is translated by `nrnivmodl` (or `mknrndll`) we see

```
$ nrnivmodl test.mod
Translating test.mod into test.c
Warning: Default 25 of PARAMETER ena will be ignored and set by NEURON.
```

and use of the model in NEURON shows that the value of `ena` is that defined by the `na_ion` mechanism itself, instead of what was asserted in the `test` model.

```
$ nrngui
 . . .
Additional mechanisms from files
 test.mod
 . . .
oc>create soma
oc>access soma
oc>insert test
oc>ena
        50
```


If we add the initialization

```
INITIAL {
  printf("ena was %g\n", ena)
  ena = 30
  printf("we think we changed it to %g\n", ena)
}
```

to `test.mod`, we quickly discover that `ena` remains unchanged.

```
oc>finitialize(-65)
ena was 50
we think we changed it to 30
         1
oc>ena
         50
```

It is perhaps not a good idea to invite diners into the kitchen, but the reason for this can be seen from the careful hiding of the ion variables by making local copies of them in the C code generated by the `nocmodl` translator. Translation of the `INITIAL` block into a model-specific `initmodel` function is an almost verbatim copy, except for some trivial boiler plate. However, `finitialize()` calls this indirectly via the model-generic `nrn_init` function, which can be seen in all its gory detail in the C file output from `nocmodl test.mod`:

```
/**************************/
static nrn_init(_count, _nodes, _data, _pdata, _type_ignore)
        int _count, _type_ignore; Node** _nodes; double** _data; Datum** _pdata;
{ int _ix; double _v;
 _p = _data; _ppvar = _pdata;

#if _CRAY
#pragma _CRI ivdep
#endif
        for (_ix = 0; _ix < _count; ++_ix) {
 _v = _nodes[_ix]->_v;
                v = _v;
  ena = _ion_ena;
                initmodel(_ix);
        }
}
/**************************/
```

It suffices merely to call attention to the statement


```
ena = _ion_ena;
```

which shows the difference between the local copy of `ena` and the pointer to the ion variable itself. The model description can touch only the local copy and is unable to change the value referenced by `_ion_ena`. Some old model descriptions circumvented this hiding by using the actual reference to the ion mechanism variables in the `INITIAL` block (from a knowledge of the translation implementation), but that was always considered an absolutely last resort.

This hands-off policy for ion variables has recently been relaxed for the case of models that `WRITE` ion concentrations, but only if the concentration is declared to be a `STATE` *and* the concentration is initialized explicitly in an `INITIAL` block. It is meaningless for more than one model at the same location to specify the same concentrations, and an error is generated if multiple models `WRITE` the same concentration variable at the same location.

If we try this mechanism

```
NEURON {
  SUFFIX test2
  USEION na WRITE nai
  RANGE nai0
}

PARAMETER {
  nai0 = 20 (milli/liter)
}

STATE {
  nai (milli/liter)
}

INITIAL {
  nai = nai0
}
```

we get this result


```
oc>create soma
oc>access soma
oc>insert test2
oc>nai
        10
oc>finitialize(-65)
        1
oc>nai
        20
oc>nai0_test2 = 30
oc>finitialize(-65)
        1
oc>nai
        30
```

If the `INITIAL` block is not present, the `nai0_test2` starting value will have no effect.

Initializing concentrations in hoc

The best way to initialize concentrations depends on the design and intended use of the model. One must ask whether the concentration is supposed to start at the same value in all sections where the mechanism has been inserted, or should it be nonuniform from the outset?

Take the case of a mechanism that `WRITE`s an ion concentration. Such a mechanism has an associated global variable that can be used to initialize the concentration to the same value in each section where the mechanism exists. These global variables have default values for [Na], [K] and [Ca] that are broadly "reasonable" but probably incorrect for any particular case. The default concentrations for ion names created by the user are 1 mM; these should be assigned correct values in `hoc`. A subsequent call to `finitialize()` will use this to initialize ionic concentrations.

The name of the global variable is formed from the name of the ion that the mechanism uses and the concentration that it `WRITE`s. For example, suppose we have a mechanism `kext` that implements extracellular potassium accumulation as described by

Frankenhaeuser and Hodgkin (Frankenhaeuser and Hodgkin 1956). The `kext` mechanism `WRITE`s `ko`, so the corresponding global variable is `ko0_k_ion`. The sequence of instructions

```
ko0_k_ion = 10      // seawater, 4 x default value (2.5)
ki0_k_ion = 4*54.4  // 4 x default value, preserves ek
finitialize(v_init) // v_init is the starting Vm
```

will set `ko` to 10 mM and `ki` to 217.6 mM in every segment that has the `kext` mechanism.

What if one or more sections of the model are supposed to have different initial concentrations? For these particular sections we can use the `ion_style()` function to assert that the global variable is not to be used to initialize the concentration for this particular ion. A complete discussion of `ion_style()`, its arguments, and its actions is contained in NEURON's help system, but we will consider one specific example here. Let's say we have inserted `kext` into section `dend`. Then the numeric arguments in the statement

```
dend ion_style("k_ion",3,2,1,1,0)
```

would have the following effects on the `kext` mechanism in the `dend` section (in sequence): treat `ko` as a `STATE` variable; treat `ek` as an `ASSIGNED` variable; on call to `finitialize()` use the Nernst equation to compute `ek` from the concentrations; compute `ek` from the concentrations on every call to `fadvance()`; do *not* use `ko0_k_ion` or `ki0_k_ion` to set the initial values of `ko` and `ki`. The proper initialization is to set `ko` and `ki` explicitly for this section, e.g.


```
ko0_k_ion = 10  // all sections start with ko = 10 mM
dend {ko = 5  ki = 2*54.4}  //  . . . except dend
finitialize(v_init)
```

Examples of custom initializations

Initializing to a particular resting potential

Perhaps the most trivial custom initialization is to force the initialized voltage to be the resting potential. Returning our consideration to initialization of the HH membrane compartment,

```
finitialize(-65)
```

will indeed set the voltage to -65 mV, and `m`, `h`, and `n` will be in steady state relative to that voltage. However, this must be considered analogous to a voltage clamp initialization since the sum of all the currents may not be 0 at this potential, i.e. -65 mV may not be the resting potential. For this reason it is common to adjust the equilibrium potential of the leak current so that the resting potential is precisely -65 mV.

This can be done with a user-defined `init()` procedure based on the idea that total membrane current at steady state must be 0. For our single compartment HH model, this means that

```
0 = ina + ik + gl_hh*(v - el_hh)
```

so our custom `init()` is

Remember to load user-defined versions of functions or procedures that are part of the standard system, such as `init()`, *after* loading stdrun.hoc. Otherwise, the user-defined version will be overwritten.


```
proc init() {
  finitialize(-65)
  el_hh = (ina + ik + gl_hh*v)/gl_hh
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

The `cvode.re_init()` call is not essential here since states have not been changed, but it is still good practice since it will update the calculation of all the *dstate/dt* (note that now *dv/dt* should be 0 as a consequence of the change in `el_hh`) as well as internally make a call to `fcurrent()` (necessary because changing `el_hh` requires recalculation of `il_hh`).

Calculating the value of leak equilibrium potential in order to realize a specific resting potential is not fail-safe in the sense that the resultant value of `el_hh` may be very large and out of its physiological range--after all, `gl_hh` may be a very small quantity. It may sometimes be better to introduce a constant current mechanism and set its value so that

```
0 = ina + ik + ica + i_constant
```

holds at the desired resting potential. An example of such a mechanism is

```
: constant current for custom initialization

NEURON {
  SUFFIX constant
  NONSPECIFIC_CURRENT i
  RANGE i, ic
}

UNITS {
  (mA) = (milliamp)
}
```



```
PARAMETER {
  ic = 0 (mA/cm2)
}

ASSIGNED {
  i (mA/cm2)
}

BREAKPOINT {
  i = ic
}
```

and the corresponding custom `init()` would be

```
proc init() {
  finitialize(-65)
  ic_constant = -(ina + ik + il_hh)
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

Before moving on to the next example, we should mention that testing is required to verify that the system is stable at the desired `v_init`, i.e. that the system returns to `v_init` after small perturbations.

Initializing to steady state

In **Chapter 4** we mentioned that NEURON's default integrator uses the backward Euler method, which can find the steady state of a linear system in a single step if the integration step size is large compared to the longest system time constant. Backward Euler can also find the steady state of many nonlinear systems, but it may be necessary to perform several iterations with large `dt`. An `init()` that takes advantage of this fact is


```
proc init() { local dtsav, temp
  finitialize(v_init)
  t = -1e10
  dtsav = dt
  dt = 1e9
  // if cvode is on, turn it off to do large fixed step
  temp = cvode.active()
  if (temp!=0) { cvode.active(0) }
  while (t<-1e9) {
    fadvance()
  }
  // restore cvode if necessary
  if (temp!=0) { cvode.active(1) }
  dt = dtsav
  t = 0
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

This first performs a preliminary "voltage clamp" initialization to `v_init`. Then it sets time to a very large negative value (to prevent triggering point processes and other events) and integrates over several steps with a large fixed `dt` so that the system can reach steady state. The procedure wraps up by returning `dt` to its original value, setting `t` back to 0, and, if necessary, reactivating the variable step integrator. The last few statements are the familiar re-initialization of cvode or invocation of `fcurrent()`, followed by initialization of vector recording.

This initialization strategy generally works well, but there are circumstances in which it may fail. Active transport mechanisms can be troublesome with fixed time step integration if `dt` is large, because even a small pump rate may produce a negative concentration. To see a more mundane example of instability with large `dt`, construct a single compartment model that has the `hh` mechanism. With the default `hh` parameters, and in the absence of any injected current, this is quite stable even for huge values of `dt`

(e.g. 10^5 ms). Now reduce `gnabar_hh` to 0, increase `dt` to 100 ms, and watch what happens over the course of 5000 ms. The result is an oscillation whose peak-to-peak amplitude gradually increases to ~ 10 mV. It would be all to easy to miss such oscillations when using steady state initialization with large steps. This underscores the need for careful testing of any initialization strategy, since in a sense all of them work "behind the scenes."

Initializing to a desired state

Suppose the end of some run is to serve as the initial condition for subsequent runs; this is a particularly useful strategy for dealing with models that oscillate or otherwise lack a "resting" state. We can save all the states with

```
objref svstate, f
svstate = new SaveState()
svstate.save()
```

The binary state information can be saved for use in later neuron sessions with

```
f = new File("states.dat")
svstate.fwrite(f)
```

and future sessions can read the file into the `SaveState` object with

```
objref svstate, f
svstate = new SaveState()
f = new File("states.dat")
svstate.fread(f)
```

Whether or not the state information comes from a `svstate.save()` in this session or was read from a file, we only have to make a minor change to `init()` in order to use that information to initialize the system.


```
proc init() {
  finitialize(v_init)
  svstate.restore()
  t = 0 // t is one of the "states"
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

This might be called a "groundhog day initialization," after the movie in which the protagonist awakened to the same day over and over again.

Now every simulation will start from the state that we saved earlier.

Initializing by changing model parameters

Occasionally the aim is to bring a model to an initial condition that it would never reach on its own. This can be a particular challenge if the model involves several interacting nonlinear processes, making it difficult or impossible to know in advance what values the states should have. Such problems can sometimes be circumvented by changing the parameters of the model so that initialization reaches the desired state, and then restoring the original parameters of the model.

As a specific example, consider a conceptual model of the regulation of the calcium concentration in a thin intracellular compartment ("shell") adjacent to the cell membrane (Fig. 8.1). Calcium (Ca^{+2}) can enter or leave the shell in one of three ways: by diffusion between the shell and the core of the cell, by active transport via a membrane-bound pump, or as a result of non-pump calcium current I_{Ca} (i.e. transmembrane calcium flux not produced by the pump). For the sake of simplicity, we will assume that Ca_{core} and Ca_o ($[Ca^{+2}]$ in the core and extracellular solution) are constant. However, the problems

that we encounter, and the manner in which we solve them, would be the same even if Ca_{core} and Ca_o were allowed to vary.

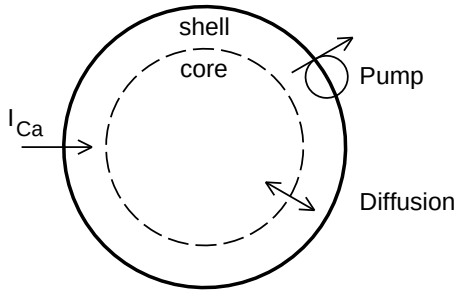


Fig. 8.1. Schematic diagram of a model of regulation of $[Ca^{+2}]$ in a thin shell just inside the cell membrane.

Our goals are to:

1. initialize the internal calcium concentration next to the membrane $[Ca^{+2}]_{shell}$ (hereafter called Ca_{shell}) to a desired value and then plot Ca_{shell} and the pump current $I_{Ca_{pump}}$ as functions of time
2. plot the starting value of $I_{Ca_{pump}}$ as a function of the initial Ca_{shell}

To achieve these goals, we must be able to set the initial value of Ca_{shell} to whatever level we want and ensure that the pump reaches its corresponding steady state.

Details of the mechanism

The kinetic scheme that describes this mechanism of calcium regulation is

diffusion

$$Ca_{core} \underset{1/\tau}{\overset{1/\tau}{\rightleftarrows}} Ca_{shell} \qquad \text{Eq. 8.3a}$$

active transport

$$Ca_{shell} + Pump \underset{k_2}{\overset{k_1}{\rightleftarrows}} CaPump \qquad \text{Eq. 8.3b and c}$$

$$CaPump \underset{k_4}{\overset{k_3}{\rightleftarrows}} Ca_o + Pump$$

calcium current

$$Ca_{shell} \overset{1/2\,F\,vol}{\leftarrow} -I_{Ca} \qquad \text{Eq. 8.3d}$$

where τ is the time constant for equilibration of Ca^{+2} between the shell and the core, F is Faraday's constant, and vol is the volume of the shell.

The NMODL code that implements this mechanism is

```
NEURON {
  SUFFIX capmp
  USEION ca READ cao, ica, cai WRITE cai, ica
  RANGE tau, width, cacore, ica, pump0
}

UNITS {
  (um)     =  (micron)
  (molar)  =  (1/liter)
  (mM)     =  (millimolar)
  (uM)     =  (micromolar)
  (mA)     =  (milliamp)
  (mol)    =  (1)
  FARADAY  =  (faraday)  (coulomb)
}
```



```
PARAMETER {
  width = 0.1    (um)
  tau = 1        (ms) : corresponds to D = 2e-7 cm2/s
  : D for Ca in water is 6e-6 cm2/s, i.e. 30x faster
  k1 = 5e8       (/mM-s)
  k2 = 0.25e6    (/s)
  k3 = 0.5e3     (/s)
  k4 = 5e0       (/mM-s)
  cacore = 0.1   (uM)
  pump0 = 3e-14  (mol/cm2)
}

ASSIGNED {
  cao     (mM) : on the order of 10 mM
  cai     (mM) : on the order of 0.001 mM
  ica     (mA/cm2)
  ica_pmp (mA/cm2)
  ica_pmp_last (mA/cm2)
}

STATE {
  cashell  (uM)       <1e-6>
  pump     (mol/cm2)  <1e-16>
  capump   (mol/cm2)  <1e-16>
}

INITIAL {
  ica = 0
  ica_pmp = 0
  ica_pmp_last = 0
  SOLVE pmp STEADYSTATE sparse
}

BREAKPOINT {
  SOLVE pmp METHOD sparse
  ica_pmp_last = ica_pmp
  ica = ica_pmp
}

KINETIC pmp {
  : volume/unit surface area has dimensions of um
  : area/unit surface area is dimensionless
  COMPARTMENT width {cashell}
  COMPARTMENT (1e13) {pump capump}
  COMPARTMENT 1(um) {cacore}
  COMPARTMENT (1e3)*1(um) {cao}
  CONSERVE pump + capump = (1e13)*pump0
  ~ cacore <-> cashell (width/tau, width/tau)
  ~ cashell + pump <-> capump ((1e7)*k1, (1e10)*k2)
  ~ capump <-> cao + pump ((1e10)*k3, (1e10)*k4)
  ica_pmp = (1e-7)*2*FARADAY*(f_flux - b_flux)

  : ica_pmp is the "new" value, but cashell must be
  : computed using the "old" value, i.e. ica_pmp_last
  ~ cashell << (-(ica - ica_pmp_last)/(2*FARADAY)*(1e7))

  cai = (0.001)*cashell
}
```


Initializing the mechanism

For the sake of convenience we will assume that our model cell has only one section called `soma`, and that `soma` is the default section. Also suppose that we have already assigned the desired value of Ca_{shell} to a parameter we will call `ca_init`, e.g. with a statement of the form `ca_init = somevalue`. Our problem is how to ensure that initialization makes `cashell_capmp` take on the value of `ca_init`.

As a naive first stab at this problem, we might try changing the `init()` procedure like this

```
proc init() {
  cashell_capmp = ca_init
  finitialize(v_init)
}
```

i.e. inserting a line that sets the desired value of Ca_{shell} before calling `finitialize()`. To see whether this has the desired effect, we need only to run a simulation and examine the time course of Ca_{shell} and the pump current $I_{Ca_{pump}}$. This quickly shows that, no matter what value we first assign to `cashell_capmp`, `finitialize()` drives Ca_{shell} and $I_{Ca_{pump}}$ to the same steady state levels (Fig. 8.2). We might have anticipated this result, because it is what steady state initialization is supposed to do. If Ca_{shell} is too high, the excess calcium will diffuse into the core or be pumped out of the cell until Ca_{shell} returns to the steady state value. On the other hand, if Ca_{shell} is too low, calcium will diffuse into the shell from the core, and the pump will slow or may even reverse, until

Ca_{shell} comes back to the steady state value. Thus, regardless of how we perturb Ca_{shell}, steady state initialization always brings the model back to the same condition.

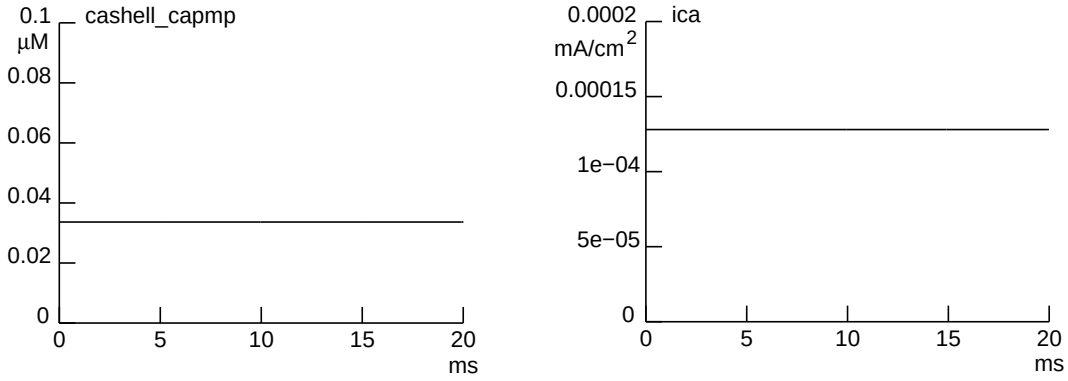


Fig. 8.2. Default initialization after setting `cashell_capmp` to 0.1 µM leaves Ca_{shell} (left) and $I_{Ca_{pump}}$ (right) at their steady state levels of ~ 0.034 µM and ~ 1.3×10^{-4} mA/cm^2, respectively.

For our second attempt we try calling `finitialize()` first, and then setting the desired value of Ca_{shell}.

```
proc init() {
  finitialize(v_init)
  cashell_capmp = ca_init
  // we've changed a state, so the following are needed
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

This is partly successful, in that it does affect Ca_{shell} and $I_{Ca_{pump}}$, but plots of these variables seem to start from the wrong initial conditions. For example, if we try `ca_init` = 0.1 µM, the plot of `cashell_capmp` appears to start with a value of

~ 0.044 μM instead. Using the Graph menu's Color/Brush to change the color and thickness of the plots of `cashell_capmp` and `ica`, we discover the presence of early, fast transients that overlie the y axis (Fig. 8.3 top). Thus `cashell_capmp` really does start at the right initial value, but in less than 5 microseconds it drops by ~ 56%. So we have solved one mystery only to uncover another: what causes these fast transients?

Some reflection brings the realization that, although we changed the concentration in the shell, we did not properly initialize the pump. Consequently, as soon as we launch a simulation, Ca^{+2} starts binding to the pump, and this is responsible for the precipitous drop of Ca_{shell}. At the same time, the rate of active transport begins to rise, which is reflected in the increase of $I_{Ca_{pump}}$. These changes produce the "pump transients" in Ca_{shell} and $I_{Ca_{pump}}$, which can be quite large as Fig. 8.3 shows.

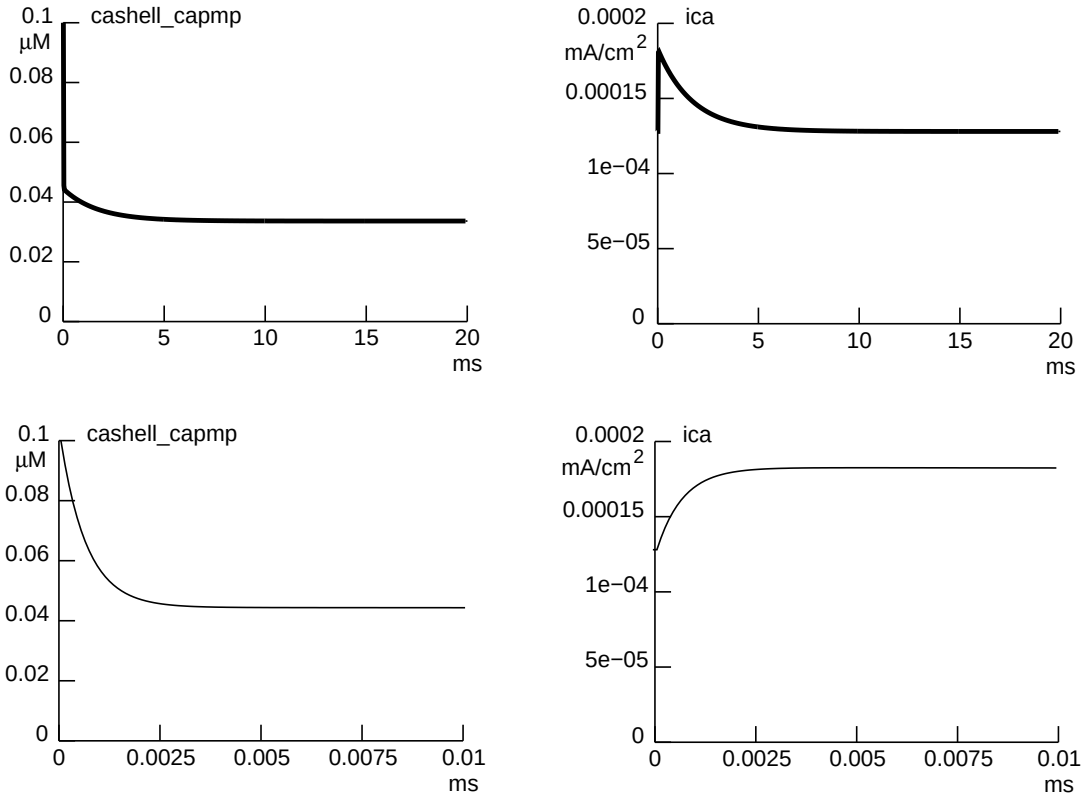


Fig. 8.3. Time course of Ca_{shell} (left) and $I_{Ca_{pump}}$ (right) following an initialization that increased Ca_{shell} abruptly after calling `init()`. The traces in the top figures were thickened to make the early fast transients easier to see. The time scale of the bottom figures has been expanded to reveal the details of these fast transients. The final steady state levels of Ca_{shell} and $I_{Ca_{pump}}$ are the same as in Fig. 8.2.

A strategy that does what we want is to change the value of `cacore_capmp` to `ca_init` and make τ very fast before calling `finitialize()`, then wrap up by restoring the values of `cacore_capmp` and τ. This amounts to changing the model in order to achieve the desired initialization. One example of such a custom `init()` is


```
proc init() { local savcore, savtau
  // make cacore equal to ca_init
  savcore = cacore_capmp
  cacore_capmp = ca_init
  // initialize cashell to cacore
  savtau = tau_capmp
  tau_capmp = 1e-6  // so cashell tracks cacore closely
  finitialize(v_init)
  // restore cacore and tau
  cacore_capmp = savcore
  tau_capmp = savtau
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

This code ensures that the difference between Ca_{shell} and Ca_{core} becomes vanishingly small, and at the same time allows the pump to initialize properly (Fig. 8.4).

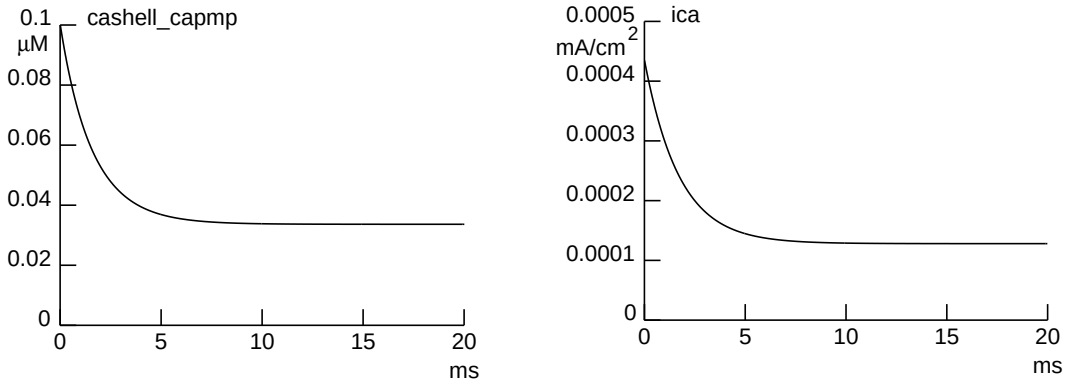


Fig. 8.4. Following proper initialization, plots of Ca_{shell} (left) and $I_{Ca_{pump}}$ (right) begin at the correct values and do not display the early fast transient that appeared in Fig. 8.3.

Now we can plot the starting value of $I_{Ca_{pump}}$ as a function of the initial Ca_{shell}. Figure 8.5 shows a Grapher configured to do this. To make this a semilog plot, we used an independent variable `x` to sweep `ca_init` from 10^{-4} to 10^{2} μM in 30 logarithmically

equally spaced intervals. For each value of `x` the Grapher calculated the corresponding value of `ca_init` as 10^x, called our custom `init()`, and plotted the resulting `ica_capmp` vs. `log10(cashell_capmp)`, i.e. $\log_{10}(Ca_{shell})$. Note that `log10(cashell_capmp)` ranges from -4 to 2, which means that Ca_{shell} ranges from 10^{-4} to 10^2 μM, i.e. exactly the same range of concentrations as `ca_init`. This confirms the ability of our custom `init()` to set `cashell_capmp` to the desired values.

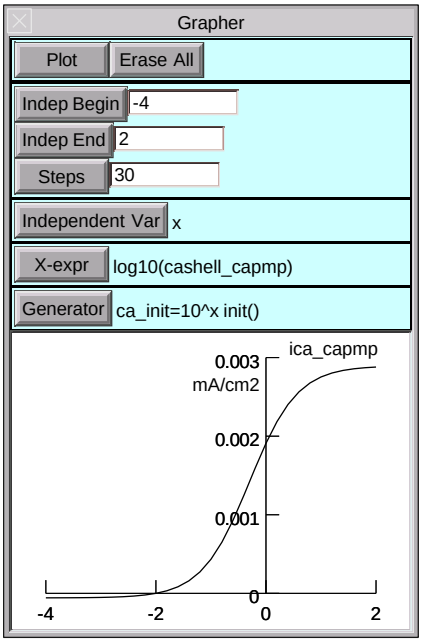


Fig. 8.5. A Grapher used to plot of $I_{Ca_{pump}}$ vs. initial Ca_{shell}. The Graph menu's Change Text was used to add the `mA/cm2` label.

References

Frankenhaeuser, B. and Hodgkin, A.L. The after-effects of impulses in the giant nerve fibers of *Loligo*. *J. Physiol.* 131:341-376, 1956.

Chapter 8 Index

J

Chapter 9

How to expand NEURON's library of mechanisms

Neuronal function involves the interaction of electrical and chemical signals that are distributed in time and space. The mechanisms that generate these signals and regulate their interactions are marked by a wide diversity of properties, differing across neuronal cell class, developmental stage, and species (e.g. chapter 7 in (Johnston and Wu 1995); also see (McCormick 1998)). To be useful in research, a simulation environment must provide a flexible and powerful means for incorporating new biophysical mechanisms in models. It must also help the user remain focused on the model instead of programming.

Such a means is provided to NEURON by NMODL, a high level language that was originally implemented for NEURON by Michael Hines and later extended by him and Upinder Bhalla to generate code suitable for linking with GENESIS (Wilson and Bower 1989). This chapter shows how to use NMODL to represent biophysical mechanisms by presenting a sequence of increasingly complex examples.

Overview of NMODL

A brief overview of how NMODL is used will clarify its rationale. First one writes a text file (a "`mod` file") that describes a mechanism as a set of nonlinear algebraic

equations, differential equations, or kinetic reaction schemes. The description employs a syntax that closely resembles familiar mathematical and chemical notation. This text is passed to a translator that converts each statement into many statements in C, automatically generating code that handles details such as mass balance for each ionic species and producing code suitable for each of NEURON's integration methods. The output of the translator is then compiled for computational efficiency. This achieves tremendous conceptual leverage and savings of effort, not only because the high level mechanism specification is much easier to understand and far more compact than the equivalent C code, but also because it spares the user from having to bother with low level programming issues like how to "interface" the code with other mechanisms and with NEURON itself.

NMODL is a descendant of the MOdel Description Language (MODL (Kohn et al. 1994)), which was developed at Duke University by the National Biomedical Simulation Resource project for the purpose of building models that would be exercised by the Simulation Control Program (SCoP (Kootsey et al. 1986)). NMODL has the same basic syntax and style of organizing model source code into named blocks as MODL. Variable declaration blocks, such as `PARAMETER`, `STATE`, and `ASSIGNED`, specify names and attributes of variables that are used in the model. Other blocks are directly involved in setting initial conditions or generating solutions at each time step (the equation definition blocks, e.g. `INITIAL`, `BREAKPOINT`, `DERIVATIVE`, `KINETIC`, `FUNCTION`, `PROCEDURE`). Furthermore, C code can be inserted inside the model source code to accomplish implementation-specific goals.

NMODL recognizes all the keywords of MODL, but we will address only those that are relevant to NEURON simulations. We will also examine the changes and extensions that were necessary to endow NMODL with NEURON-specific features. To give these ideas real meaning, we will consider them in the context of models of the following kinds of mechanisms:

- a passive "leak" current and a localized transmembrane shunt (distributed mechanisms vs. point processes)
- an electrode stimulus (discontinuous parameter changes with variable time step methods)
- voltage-gated channels (differential equations vs. kinetic schemes)
- ion accumulation in a restricted space (extracellular K^+)
- buffering, diffusion, and active transport (Ca^{2+} pump)

Features of NMODL that are used in models of synaptic transmission and networks are examined in **Chapter 10**.

Example 9.1: a passive "leak" current

A passive "leak" current is one of the simplest biophysical mechanisms. Because it is distributed over the surface of a cell, it is described in terms of conductance per unit area and current per unit area, and therefore belongs to the class of "density" or "distributed mechanisms" (see **Distributed mechanisms** in **Chapter 5**).

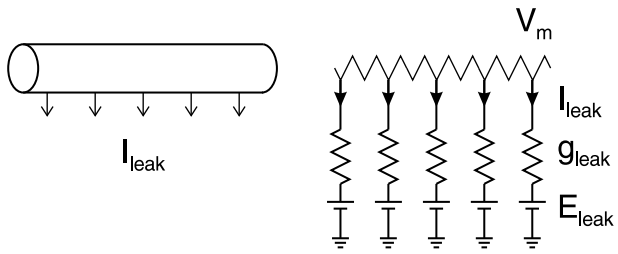


Figure 9.1

Figure 9.1 illustrates a branch of a neuron with a distributed leak current (left) and the equivalent circuit of a model of this passive current mechanism (right): a distributed, voltage-independent conductance g_{leak} in series with a voltage source E_{leak} that represents the equilibrium potential for the ionic current. The leak current density is given by $i_{leak} = g_{leak} (V_m - E_{leak})$, where V_m is membrane potential. Since this is a model of a physical system that is distributed in space, the variables i_{leak} and V_m and the parameters g_{leak} and E_{leak} are all functions of position.

Listing 9.1 presents an implementation of this mechanism with NMODL. Single line comments start with a : (colon) and terminate at the end of the line. NMODL also allows multiple line comments, which are demarcated by the keywords `COMMENT` and `ENDCOMMENT`.

```
COMMENT
This is a
multiple line
comment
ENDCOMMENT
```

A similar syntax can be used to embed C code in a `mod` file, e.g.


```
VERBATIM
   /* C statements */
ENDVERBATIM
```

The statements between `VERBATIM` and `ENDVERBATIM` will appear without change in the output file written by the NMODL translator. Although this should be done only with great care, `VERBATIM` can be a convenient and effective way to add new features or even to employ NEURON as a "poor man's C compiler."

```
: A passive leak current

NEURON {
  SUFFIX leak
  NONSPECIFIC_CURRENT i
  RANGE i, e, g
}

PARAMETER {
  g = 0.001  (siemens/cm2)  < 0, 1e9 >
  e = -65    (millivolt)
}

ASSIGNED {
  i  (milliamp/cm2)
  v  (millivolt)
}

BREAKPOINT { i = g*(v - e) }
```

Listing 9.1. `leak.mod`

Named blocks have the general form *`KEYWORD`* `{` *`statements`* `}`, where *`KEYWORD`* is all upper case. User-defined variable names in NMODL can be up to 20 characters long. Each variable must be defined before it is used. The variable names chosen for this example were `i`, `g`, and `e` for the leak current, its specific conductance, and its equilibrium potential, respectively. Some variables are not "owned" by any mechanism but are available to all mechanisms; these include `v`, `celsius`, `t`, `diam`, and `area`.

Another variable that is available to all mechanisms is `dt`. However, using `dt` in NMODL is neither necessary nor good practice. Before variable time step methods were added to NEURON, analytic expressions involving `dt` were often used for efficient modeling of voltage sensitive channel states. This idiom is now built-in and employed automatically when such models are described in their underlying derivative form.

The `NEURON` block

The principal extension that differentiates NMODL from its MODL origins is that there are separate instances of mechanism data, with different values of states and parameters, in each segment (compartment) of a model cell. The `NEURON` block was introduced to make this possible by defining what the model of the mechanism looks like from the "outside" when many instances of it are sprinkled at different locations on the cell. The specifications entered in this block are independent of any particular simulator, but the detailed "interface code" requirements of a particular simulator determine whether the output C file is suitable for NEURON (NMODL) or GENESIS (GMODL). For this paper, we assume the translator is NMODL and that it produces code accepted by NEURON.

The actual name of the current NMODL translator is `nocmodl` (`nocmodl.exe` under MSWindows). This translator is consistent with the object oriented extensions that were introduced with version 3 of NEURON. However, the older translator, which predated these extensions, was called `nmodl`, and we will use the generic name NMODL to refer to NEURON compatible translators.

The SUFFIX keyword has two consequences. First, it identifies this to be a distributed mechanism, which can be incorporated into a NEURON cable section by an `insert` statement (see **Usage** below). Second, it tells the NEURON interpreter that the names for variables and parameters that belong to this mechanism will include the suffix `_leak`, so there will be no conflict with similar names in other mechanisms.

The stipulation that `i` is a NONSPECIFIC_CURRENT also has two consequences. First, the value of `i` will be reckoned in charge balance equations. Second, this current will make no direct contribution to mass balance equations (it will have no direct effect on ionic concentrations). In later examples, we will see how to implement mechanisms with specific ionic currents that can change concentrations.

The RANGE statement asserts that `i`, `e`, and `g` are range variables, and can be accessed by the hoc interpreter using range variable syntax (see **Range and range variables** in **Chapter 5**). That is, each of these variables is a function of position, and can have a different value in each of the segments that make up a section. Each variable mentioned in a RANGE statement should also be declared in a PARAMETER or ASSIGNED block. The alternative to RANGE is GLOBAL, which is discussed below in **The PARAMETER block**.

Membrane potential `v` is not mentioned in the NEURON block because it is one of the variables that are available to all mechanisms, and because it is a RANGE variable by default. However, for model completeness in non-NEURON contexts, and to enable units checking, `v` should be declared in the ASSIGNED block (see below).

Variable declaration blocks

As noted above, each user-defined variable must be declared before it is used. Even if it is named in the `NEURON` block, it still has to appear in a variable declaration block.

Mechanisms frequently involve expressions that mix constants and variables whose units belong to different scales of investigation and which may themselves be defined in terms of other, more "fundamental" units. This can easily produce arithmetic errors that are difficult to isolate and rectify. Therefore NMODL has special provisions for establishing and maintaining consistency of units. To facilitate units checking, each variable declaration includes a specification of its units in parentheses. The names used for these specifications are defined in a file called `nrnunits.lib`, which is based on the UNIX units database (`/usr/share/units.dat` in Linux). `nrnunits.lib` is located in `nrn-x.x/share/lib/` under UNIX/Linux, and `c:\nrnxx\lib\` under MSWindows). A variable whose units are not specified is taken to be dimensionless.

The user may specify whatever units are appropriate except for variables that are defined by NEURON itself. These include `v` (millivolts), `t` (milliseconds), `celsius` (°C), `diam` (μm), and `area` (μm^2). Currents, concentrations, and equilibrium potentials created by the `USEION` statement also have their own particular units (see **The `NEURON` block** in **Example 9.6: extracellular potassium accumulation** below). In this particular distributed mechanism, `i` and `g` are given units of current per unit area (milliamperes/cm^2) and conductance per unit area (siemens/cm^2), respectively.

The PARAMETER block

Variables whose values are normally specified by the user are parameters, and are declared in a PARAMETER block. PARAMETERs generally remain constant during a simulation, but they can be changed in mid-run if necessary to emulate some external influence on the characteristic properties of a model (see **Models with discontinuities** and **Time-dependent PARAMETER changes** near the end of this chapter)

The PARAMETER block in this example assigns default values of 0.001 siemens/cm^2 and -65 mV to g and e, respectively. The pair of numbers in angle brackets specifies the minimum and maximum limits for g that can be entered into the field editor of the GUI. In this case, we are trying to keep conductance g from assuming a negative value. This protection, however, only holds for field editors and does not prevent a hoc statement from giving g a negative value.

Because g and e are PARAMETERs, their values are visible at the hoc level and can be overridden by hoc commands or altered through the GUI. If a PARAMETER does not appear in a NEURON block's RANGE statement, it will have GLOBAL scope, which means that changing its value will affect every instance of that mechanism throughout an entire model. However, the RANGE statement in the NEURON block of this particular mechanism asserts that g and e are range variables, so they can be given different values in every segment that has this leak current.

The ASSIGNED block

The ASSIGNED block is used to declare two kinds of variables: those that are given values outside the mod file, and those that appear on the left hand side of assignment statements within the mod file. The first group includes variables that are potentially available to every mechanism, such as v, celsius, t, and ionic variables (ionic variables are discussed in connection with **The NEURON block** in **Example 9.6: extracellular potassium accumulation** below). The second group specifically omits variables that are unknowns in a set of simultaneous linear or nonlinear algebraic equations, or that are dependent variables in differential equations or kinetic reaction schemes, which are handled differently (see **Example 9.4: a voltage-gated current** below for a discussion of the STATE block).

By default, a mechanism-specific ASSIGNED variable is a range variable, in that it can have a different value for each instance of the mechanism. However, it will not be visible at the hoc level unless it is declared in a RANGE or GLOBAL statement in the NEURON block. This contrasts with ASSIGNED variables that are not "owned" by any mechanism (v, celsius, t, dt, diam, and area) which *are* visible at the hoc level but are *not* mentioned in the NEURON block.

The current i is not a state variable because the model of the leak current mechanism does not define it in terms of a differential equation or kinetic reaction scheme; that is to say, i has no dynamics of its own. Furthermore it is not an unknown in a set of equations, but is merely calculated by direct assignment. Therefore it is declared in the ASSIGNED block.

For similar reasons membrane potential `v` is also declared in the `ASSIGNED` block. Although membrane potential is unquestionably a state variable in a model of a cell, to the leak current mechanism it is a driving force rather than a state variable (or even a `STATE` variable).

Equation definition blocks

One equation suffices to describe this simple leak current model. This equation is defined in the `BREAKPOINT` block. As we shall see later, more complicated models may require invoking NMODL's built-in routines to solve families of simultaneous algebraic equations or perform numeric integration.

The `BREAKPOINT` block

The `BREAKPOINT` block is the main computation block in NMODL. Its name derives from SCoP, which executes simulations by incrementing an independent variable over a sequence of steps or "breakpoints" at which the dependent variables of the model are computed and displayed (Kohn et al. 1994). At exit from the `BREAKPOINT` block, all variables should be consistent with the independent variable. The independent variable in NEURON is always time `t`, and neither `t` nor the time step `dt` should be changed in NMODL.

Usage

The following `hoc` code illustrates how this mechanism might be used. Note the use of `RANGE` syntax to examine the value of `i_leak` near one end of `cable`.


```
cable {
  nseg = 5
  insert leak
  // override defaults
  g_leak = 0.002  // S/cm2
  e_leak = -70    // mV
}

// show leak current density near 0 end of cable
print cable.i_leak(0.1)
```

The `leak` mechanism automatically appears with the other distributed mechanisms in GUI tools such as the Distributed Mechanism Inserter (Fig. 9.2). This is a consequence of interface code that is generated by the NMODL compiler when it parses the definitions in the `NEURON` block.

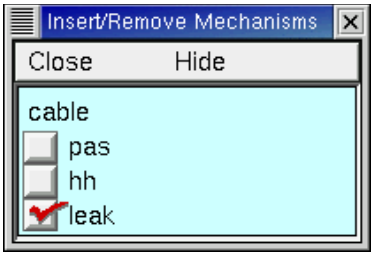


Figure 9.2. Compiling the leak mechanism automatically makes it available through NEURON's graphical user interface, as in this Distributed Mechanism Inserter (brought up by NEURON Main Menu / Tools / Distributed Mechanisms / Managers / Inserter). The check mark signifies that the `leak` mechanism has been inserted into the section named `cable`.

Example 9.2: a localized shunt

At the opposite end of the spatial scale from a distributed passive current is a localized shunt induced by microelectrode impalement (Durand 1984; Staley et al.

1992). A shunt is restricted to a small enough region that it can be described in terms of a net conductance (or resistance) and total current, i.e. it is a point process (see **Point processes** in **Chapter 5**). Most synapses are also best represented by point processes.

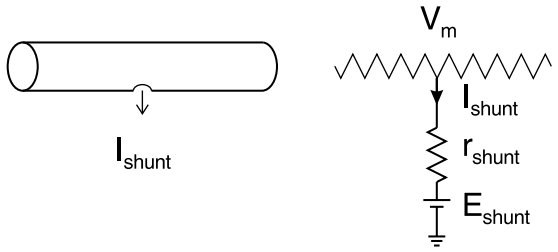


Figure 9.3

The localized nature of the shunt is emphasized in a cartoon of the neurite (Fig. 9.3 left). The equivalent circuit of the shunt (right) is similar to the equivalent circuit of the distributed leak current (Fig. 9.1 right), but here the resistance and current are understood to be concentrated in a single, circumscribed part of the cell. We will focus on how the NMODL code for this localized shunt (Listing 9.2) differs from the leak distributed mechanism of **Example 9.1**.

The `NEURON` block

The `POINT_PROCESS` statement in the `NEURON` block identifies this mechanism as a point process, so it will be managed in `hoc` using an object oriented syntax (see **Usage** below). Declaring `i`, `e`, and `r` to be `RANGE` means that each instance of this point process can have separate values for these variables. If a variable is declared in a `GLOBAL` statement, then its value is shared among all instances of the mechanism (however, see

Equation definition blocks: The DERIVATIVE block in **Example 9.5: a calcium-activated, voltage-dependent current**).

Variable declaration blocks

These are nearly identical to the `PARAMETER` and `ASSIGNED` blocks of the `leak` mechanism. However, `Shunt` is a point process so all of its current flows at one site instead of being distributed over an area. Therefore its `i` and `r` are in units of nanoamperes (total current) and gigaohms (0.001 / total conductance in microsiemens), respectively.

This code specifies default values for the `PARAMETER`s `r` and `e`. Allowing a minimum value of 10^{-9} for `r` prevents an inadvertent divide by 0 error (infinite conductance) by ensuring that a user cannot set `r` to 0 in its GUI field editor. However, as we noted in the `leak` model, the *<minval, maxval>* syntax does not prevent a `hoc` statement from assigning `r` a value outside of the desired range.

```
: A shunt current

NEURON {
  POINT_PROCESS Shunt
  NONSPECIFIC_CURRENT i
  RANGE i, e, r
}

PARAMETER {
  r = 1 (gigaohm)  < 1e-9, 1e9 >
  e = 0 (millivolt)
}

ASSIGNED {
  i  (nanoamp)
  v  (millivolt)
}
```



```
BREAKPOINT { i = (0.001)*(v - e)/r }
```

Listing 9.2. `shunt.mod`

Equation definition blocks

Like the leak current mechanism, the shunt mechanism is extremely simple and involves no state variables. Its single equation is defined in the `BREAKPOINT` block.

The `BREAKPOINT` block

The sole "complication" is that computation of `i` includes a factor of 0.001 to reconcile the units on the left and right hand sides of this assignment (nanoamperes vs. millivolts divided by gigaohms). The parentheses surrounding this conversion factor are a convention for units checking: they disambiguate it from mere multiplication by a number. When the NMODL code in Listing 9.2 is checked with NEURON's `modlunit` utility, no inconsistencies will be found.

```
[ted@fantom dshunt]$ modlunit shunt.mod
model   1.1.1.1   1994/10/12 17:22:51
Checking units of shunt.mod
[ted@fantom dshunt]$
```

However if the conversion factor were not enclosed by parentheses, there would be an error message that reports inconsistent units.

```
[ted@fantom dshunt]$ modlunit shunt.mod
model   1.1.1.1   1994/10/12 17:22:51
Checking units of shunt.mod
The previous primary expression with units: 1-12 coul/sec
is missing a conversion factor and should read:
  (0.001)*()
 at line 20 in file shunt.mod
        i = 0.001*(v - e)/r<<ERROR>>
[ted@fantom dshunt]$
```


An error message would also result if parentheses surrounded a number which the user intended to be a quantity, since the units would be inconsistent.

The simple convention of enclosing single numbers in parentheses to signify units conversion factors minimizes the possibility of mistakes, either by the user or by the software. It is important to note that expressions that involve more than one number, such as "(1 + 1)", will *not* be interpreted as units conversion factors.

Usage

This hoc code illustrates how the shunt mechanism might be applied to a section called cable; note the object syntax for specifying the shunt resistance and current (see **Point processes** in **Chapter 5**).

```
objref s
cable s = new Shunt(0.1) // put near 0 end of cable
s.r = 0.2 // pretty good for a sharp electrode
print s.i // show shunt current
```

The definitions in the NEURON block of this particular model enable NEURON's graphical tools to include the Shunt object in the menus of its PointProcessManager and Viewer windows (Fig. 9.4). The check mark on the button adjacent to the numeric field for r indicates that the shunt resistance has been changed from its default value (0.2 gigaohm when the shunt was created by the hoc code immediately above) to 0.1 gigaohm.

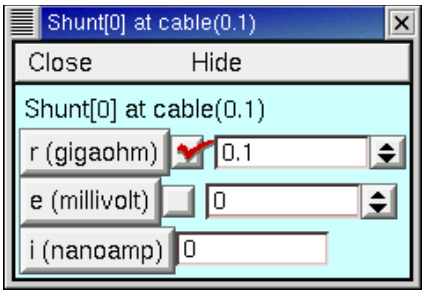


Figure 9.4. The properties of a specific instance of the `Shunt` mechanism are displayed in this Point Process Viewer (brought up by NEURON Main Menu / Tools / Point Processes / Viewers / PointProcesses / Shunt and then selecting `Shunt[0]` from the displayed list).

Example 9.3: an intracellular stimulating electrode

An intracellular stimulating electrode is similar to a shunt in the sense that both are localized sources of current that are modeled as point processes. However, the current from a stimulating electrode is not generated by an opening in the cell membrane, but instead is injected directly into the cell. This particular model of a stimulating electrode (Listing 9.3) has the additional difference that the current changes discontinuously, i.e. it is a pulse with distinct start and stop times.

The `NEURON` block

This mechanism is identical to NEURON's built-in `IClamp`. Calling it `IClamp1` allows the reader to test and modify it without conflicting with the existing `IClamp` point process.

This model of a current clamp generates a rectangular current pulse whose amplitude `amp` in nanoamperes, start time `del` in milliseconds, and duration `dur` in milliseconds are all adjustable by the user. Furthermore, these parameters need to be individually adjustable for each separate instance of this mechanism, so they appear in a `RANGE` statement.

The current `i` delivered by `IClamp1` is declared in the `NEURON` block to make it available for examination. The `ELECTRODE_CURRENT` statement has two important consequences: positive values of `i` will depolarize the cell (in contrast to the hyperpolarizing effect of positive transmembrane currents), and when the `extracellular` mechanism is present there will be a change in the extracellular potential `vext`.

Equation definition blocks

The `BREAKPOINT` block

The logic for deciding whether `i` = 0 or `i` = `amp` is straightforward, but the `at_time()` calls need explanation. From the start we wish to emphasize that `at_time()` has become a "deprecated" function, i.e. it still works but it should not be used in future model development. We bring it up here because you may encounter it in legacy code. However, NEURON's event delivery system (see **Chapter 10**) provides a far better way to implement discontinuities.

To work properly with variable time step methods, e.g. CVODE, models that change parameters discontinuously during a simulation must notify NEURON when such events

take place. With fixed time step methods, users implicitly assume that events occur on time step boundaries (integer multiples of `dt`), and they would never consider defining a pulse duration narrower than `dt`. Neither eventuality can be left to chance with variable time step methods.

When this mechanism is used in a variable time step simulation, the first `at_time()` call guarantees there will be a time step boundary just before `del`, and that integration will restart from a new initial condition just after `del` (see **Models with discontinuities** near the end of this chapter for more details).

```
: Current clamp

NEURON {
  POINT_PROCESS IClamp1
  RANGE del, dur, amp, i
  ELECTRODE_CURRENT i
}

UNITS { (nA) = (nanoamp) }

PARAMETER {
  del (ms)
  dur (ms)  < 0, 1e9 >
  amp (nA)
}

ASSIGNED { i (nA) }

INITIAL { i = 0 }

BREAKPOINT {
  at_time(del)
  at_time(del+dur)
  if (t < del + dur && t > del) {
    i = amp
  } else {
    i = 0
  }
}
```

Listing 9.3. `iclamp1.mod`

The `INITIAL` block

The code in the `INITIAL` block is executed when the standard run system's `finitialize()` is called. The initialization here consists of making sure that `IClamp1.i` is 0 when `t` = 0. Initialization of more complex mechanisms is discussed below in **Example 9.4: a voltage-gated current** and **Example 9.6: extracellular potassium accumulation**, and **Chapter 8** considers the topic of initialization from a broader perspective.

Usage

Regardless of whether a fixed or variable time step integrator is chosen, `IClamp1` looks the same to the user. In either case, a current stimulus of 0.01 nA amplitude that starts at `t` = 1 ms and lasts for 2 ms would be created by this `hoc` code

```
objref ccl
// put at middle of soma
soma ccl = new IClamp1(0.5)
ccl.del = 1
ccl.dur = 2
ccl.amp = 0.01
```

or through the PointProcessManager GUI tool (Fig. 9.5).

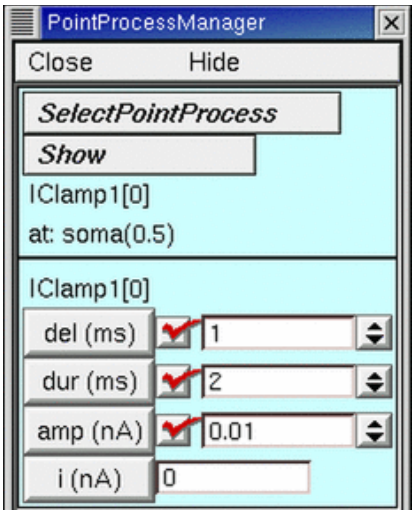


Figure 9.5. A PointProcessManager configured as an `Iclamp1` object.

Example 9.4: a voltage-gated current

One of the particular strengths of NMODL is its flexibility in dealing with ion channels whose conductances are not constant but instead are regulated by factors such as membrane potential and/or ligand concentrations on one or both sides of the membrane. Here we use the well known Hodgkin-Huxley (HH) delayed rectifier to show how a voltage-gated current can be implemented; in this example, membrane potential is in absolute millivolts, i.e. reversed in polarity from the original Hodgkin-Huxley convention and shifted to reflect a resting potential of -65 mV. In **Example 9.5** we will examine a potassium current model that depends on both voltage and intracellular calcium concentration.

The delayed rectifier and all other ion channels that are distributed over the cell surface are distributed mechanisms. Therefore their NMODL representations and hoc

usage be similar to those of **Example 9.1: a passive "leak" current**. The following discussion focuses on the significant differences between the implementations of the delayed rectifier and passive leak current models.

```
: HH voltage-gated potassium current

NEURON {
  SUFFIX kd
  USEION k READ ek WRITE ik
  RANGE gbar, g, i
}

UNITS {
  (S)  =  (siemens)
  (mV) =  (millivolt)
  (mA) =  (milliamp)
}

PARAMETER { gbar = 0.036 (S/cm2) }

ASSIGNED {
  v  (mV)
  ek (mV) : typically ~ -77.5
  ik (mA/cm2)
  i  (mA/cm2)
  g  (S/cm2)
}

STATE { n }

BREAKPOINT {
  SOLVE states METHOD cnexp
  g = gbar * n^4
  i = g * (v - ek)
  ik = i
}

INITIAL {
  : Assume v has been constant for a long time
  n = alpha(v)/(alpha(v) + beta(v))
}

DERIVATIVE states {
  : Computes state variable n at present v & t
  n' = (1-n)*alpha(v) - n*beta(v)
}
```



```
FUNCTION alpha(Vm (mV)) (/ms) {
  LOCAL x
  UNITSOFF
  x = (Vm+55)/10
  if (fabs(x) > 1e-6) {
    alpha = 0.1*x/(1 - exp(-x))
  } else {
    alpha = 0.1/(1 - 0.5*x)
  }
  UNITSON
}

FUNCTION beta(Vm (mV)) (/ms) {
  UNITSOFF
  beta = 0.125*exp(-(Vm+65)/80)
  UNITSON
}
```

Listing 9.4. kd.mod

The NEURON block

As with the passive leak model, SUFFIX marks this as a distributed mechanism, whose variables and parameters are identified in hoc by a particular suffix. Three RANGE variables are declared in this block: the peak conductance density gbar (the product of channel density and "open" conductance per channel), the macroscopic conductance g (the product of gbar and the fraction of channels that are open at any moment), and the current i that passes through g. At the level of hoc, these will be available as gbar_kd, g_kd, and i_kd.

This model also has a fourth range variable: the gating variable n, which is declared in the STATE block (see **The STATE block** below). STATE variables are automatically RANGE variables and do not need to be declared in the NEURON block.

A mechanism needs a separate USEION statement for each of the ions that it affects or that affect it. This example has one USEION statement, which includes READ ek because

the potential gradient that drives `i_kd` depends on the equilibrium potential for K^+ (potassium). Since the resulting ionic flux may change local $[K^+]$, this example also includes `WRITE ik`. The `WRITE i`*x* syntax enables NEURON to keep track of the total outward current that is carried by an ionic species, its internal and external concentrations, and its equilibrium potential. We will return to this point in the context of a model with extracellular K^+ accumulation.

The `UNITS` block

The statements in the `UNITS` block define new names for units in terms of existing names in the UNIX units database. This can increase legibility and convenience, and is helpful both as a reminder to the user and as a means for automating the process of checking for consistency of units.

Variable declaration blocks

The `ASSIGNED` block

This is analogous to the `ASSIGNED` block of the `leak` mechanism. For the sake of clarity, variables whose values are computed outside this `mod` file are listed first. Note that `ek` is listed as an `ASSIGNED` variable, unlike the leak mechanism's `e` which was a `PARAMETER`. The reason for this difference is that mechanisms that produce fluxes of specific ions, such as K^+, may cause the ionic equilibrium potential to change in the course of a simulation. However, the `NONSPECIFIC_CURRENT` generated by the leak

mechanism was not linked to any particular ionic species, so `e_leak` remains fixed unless explicitly altered by `hoc` statements or the GUI.

The STATE block

If a model involves differential equations, families of algebraic equations, or kinetic reaction schemes, their dependent variables or unknowns are to be listed in the `STATE` block. Therefore gating variables such as the delayed rectifier's `n` are declared here.

In NMODL, variables that are declared in the `STATE` block are called `STATE` variables, or simply `STATE`s. This NMODL-specific terminology should not be confused with the physics or engineering concept of a "state variable" as a variable that describes the state of a system. While membrane potential is a "state variable" in the engineering sense, it would never be a `STATE` because its value is calculated only by NEURON and never by NMODL code. Likewise, the unknowns in a set of simultaneous equations (e.g. specified in a `LINEAR` or `NONLINEAR` block) would not be state variables in an engineering sense, yet they would all be `STATE`s (see **State variables and STATE variables** in **Chapter 8**).

All `STATE`s are automatically `RANGE` variables. This is appropriate, since channel gating can vary with position along a neurite.

Equation definition blocks

In addition to the `BREAKPOINT` block, this model also has `INITIAL`, `DERIVATIVE`, and `FUNCTION` blocks.

The BREAKPOINT block

This is the main computation block of the mechanism. By the end of the BREAKPOINT block, all variables are consistent with the new time. If a mechanism has STATEs, this block must contain one SOLVE statement that tell how the values of the STATEs will be computed over each time step. The SOLVE statement specifies a block of code that defines the simultaneous equations that govern the STATEs. Currents are set with assignment statements at the end of the BREAKPOINT block.

There are two major reasons why variables that depend on the number of executions, such as counts or flags or random numbers, should generally not be calculated in a BREAKPOINT block. First, the assignment statements in a BREAKPOINT block are usually called twice per time step. Second, with variable time step methods the value of t may not even be monotonically increasing. The proper way to think about this is to remember that the BREAKPOINT block is responsible for making all variables consistent at time t. Thus assignment statements in this block are responsible for trivially specifying the values of variables that depend *only* on the values of STATEs, t, and v, while the SOLVE statements perform the magic required to make the STATEs consistent at time t. It is not belaboring the point to reiterate that the assignment statements should produce the same result regardless of how many times BREAKPOINT is called with the same STATEs, t, and v. All too often, errors have resulted from an attempt to explicitly compute what is conceptually a STATE in a BREAKPOINT block. Computations that must be performed only once per time step should be placed in a PROCEDURE, which in turn would be invoked by a SOLVE statement in a BREAKPOINT block.

We must also emphasize that the SOLVE statement is not a function call, and that the body of the DERIVATIVE block (or any other block specified in a SOLVE statement) will be executed asynchronously with respect to BREAKPOINT assignment statements. Therefore it is incorrect to invoke rate functions from the BREAKPOINT block; instead these must be called from the block that is specified by the SOLVE statement (in this example, from within the DERIVATIVE block).

Models of active currents such as i_kd are generally formulated in terms of ionic conductances governed by gating variables that depend on voltage and time. The SOLVE statements at the beginning of the BREAKPOINT block specify the differential equations or kinetic schemes that govern the kinetics of the gating variables. The algebraic equations that compute the ionic conductances and currents follow the SOLVE statements.

The INITIAL block

Though often overlooked, proper initialization of *all* STATEs is as important as correctly computing their temporal evolution. This is accomplished for the common case by the standard run system's finitialize(), which executes the initialization strategy defined in the INITIAL block of each mechanism (see also **INITIAL blocks in NMODL** in **Chapter 8**). The INITIAL block may contain any instructions that should be executed when the hoc function finitialize() is called.

Prior to executing the INITIAL block, STATE variables are set to the values asserted in the STATE block (or to 0 if no specific value was given in the STATE block). A NET_RECEIVE block, if present, may also have its own INITIAL block for nonzero initialization of NetCon "states" (the NET_RECEIVE block and its initialization are

discussion further in **Chapter 10** and under **Basic initialization in NEURON: `finitialize()`** in **Chapter 8**).

The `INITIAL` block should be used to initialize `STATE`s with respect to the initial values of membrane potential and ionic concentrations. There are several other ways to prepare `STATE`s for a simulation run, the most direct of which is simply to assign values explicitly with `hoc` statements such as `axon.n_kd(0.3) = 0.9`. However, this particular strategy can create arbitrary initial conditions that would be quite unnatural. A more "physiological" approach, which may be appropriate for models of oscillating or chaotic systems or whose mechanisms show other complex interactions, is to perform an initialization run during which the model converges toward its limit cycle or attractor. One practical alternative for systems that settle to a stable equilibrium point when left undisturbed is to assign `t` a large negative value and then advance the simulation over several large time steps (keeping `t` < 0 prevents the initialization steps from triggering scheduled events such as stimulus currents or synaptic inputs); this takes advantage of the strong stability properties of NEURON's implicit integration methods (see **Chapter 4**). For a more extensive discussion of initialization, see **Chapter 8**, especially **Examples of custom initializations**).

This delayed rectifier mechanism sets `n` to its steady state value for the local membrane potential wherever the mechanism has been inserted. This potential itself can be "left over" from a previous simulation run, or it can be specified by the user, e.g. uniformly over the entire cell with a statement like `finitialize(-55)`, or on a compartment by compartment basis by asserting statements such as `dend.v(0.2) =`

`-48` before calling `finitialize()` (see **Default initialization in the standard run system: `stdinit()` and `init()`** in **Chapter 8**).

The `DERIVATIVE` block

This is used to assign values to the derivatives of those `STATE`s that are described by differential equations. The statements in this block are of the form *`y' = expr`*, where a series of apostrophes can be used to signify higher order derivatives.

In fixed time step simulations, these equations are integrated using the numerical method specified by the `SOLVE` statement in the `BREAKPOINT` block. The `SOLVE` statement should explicitly invoke one of the integration methods that is appropriate for systems in which state variables can vary widely during a time step (stiff systems). The `cnexp` method, which combines second order accuracy with computational efficiency, is a good choice for this example. It is appropriate when the right hand side of $y' = f(v,y)$ is linear in y and involves no other states, so it is well suited to models with HH-style ionic currents. This method calculates the `STATE`s analytically under the assumption that all other variables are constant throughout the time step. If the variables change but are second order correct at the midpoint of the time step, then the calculation of `STATE`s is also second order correct.

If $f(v,y)$ is not linear in y, then the `SOLVE` statement in the `BREAKPOINT` block should specify the implicit integration method

Other integrators, such as `runge` and `euler`, are defined but are not useful in the NEURON context. Neither is guaranteed to be numerically stable, and `runge`'s high order accuracy is wasted since voltage does not have an equivalent order of accuracy.

`derivimplicit`. This provides first order accuracy and is usable with general ODEs regardless of stiffness or nonlinearity.

With variable time step methods, *no* variable is assumed to be constant. These methods not only change the time step, but adaptively choose a numerical integration formula with accuracy that ranges from first order up to $O(\Delta t^6)$. The present implementation of NMODL creates a diagonal Jacobian approximation for the block of `STATE`s. This is done analytically if $y_i' = f_i(v,y)$ is polynomial in y_i; otherwise, the Jacobian is approximated by numerical differencing. In the rare case where this is inadequate, the user may supply an explicit Jacobian. Future versions of NMODL may attempt to deal with Jacobian evaluation in a more sophisticated manner. This illustrates a particularly important benefit of the NMODL approach: improvements in methods do not affect the high level description of the membrane mechanism.

The `FUNCTION` block

The functions defined by `FUNCTION` blocks are available at the `hoc` level and in other mechanisms by adding the suffix of the mechanism in which they are defined, e.g. `alpha_kd()` and `beta_kd()`. Functions or procedures can be simply called from `hoc` if they do not reference range variables (references to `GLOBAL` variables are allowed). If a function or procedure does reference a range variable, then prior to calling the function from `hoc` it is necessary to specify the proper instance of the mechanism (its location on the cell). This is done by a `setdata_` function that has the syntax

```
section_name setdata_suffix(x)
```


where *section_name* is the name of the section that contains the mechanism in question, *suffix* is the mechanism suffix, and x is the normalized distance along the section where the particular instance of the mechanism exists. The functions in our kd example do not use range variables, so a specific instance is not needed.

The differential equation that describes the kinetics of n involves two voltage-dependent rate constants whose values are computed by the functions alpha() and beta(). The original algebraic form of the equations that define these rates is

$$\alpha = \frac{0.1\left(\frac{v+55}{10}\right)}{1-\mathrm{e}^{-\left(\frac{v+55}{10}\right)}} \quad \text{and} \quad \beta = 0.125\ \mathrm{e}^{-\left(\frac{v+65}{80}\right)} \qquad \text{Eq. 9.1}$$

The denominator for α goes to 0 when v = -55 mV, which could cause numeric overflow. The code used in alpha() avoids this by switching, when v is very close to -55, to an alternative expression that is based on the first three terms of the infinite series expansion of e^x.

As noted elsewhere in this paper, NMODL has features that facilitate establishing and maintaining consistency of units. Therefore the rate functions alpha() and beta() are introduced with the syntax

```
FUNCTION f_name(arg1 (units1), arg2 (units2), . . . )(returned_units)
```

to declare that their arguments are in units of millivolts and that their returned values are in units of inverse milliseconds ("/ms"). This allows automatic units checking on entry to and return from these functions. For the sake of legibility the UNITSOFF . . . UNITSON directives disable units checking just within the body of these functions. This is

acceptable because the terms in the affected statements are mutually consistent. Otherwise the statements would have to be rewritten in a way that makes unit consistency explicit at the cost of legibility, e.g.

```
x = (Vm + 55 (millivolt))/(10 (millivolt))
```

Certain variables exist solely for the sake of computational convenience. These typically serve as scale factors, flags, or temporary storage for intermediate results, and are not of primary importance to the mechanism. Such variables are often declared as `LOCAL` variables declared within an equation block, e.g. `x` in this mechanism. `LOCAL` variables that are declared in an equation block are not "visible" outside the block and they do not retain their values between invocations of the block. `LOCAL` variables that are declared outside an equation block have very different properties and are discussed under **Variable declaration blocks** in **Example 9.8: calcium diffusion with buffering**.

Usage

The `hoc` code and graphical interface for using this distributed mechanism are similar to those for the `leak` mechanism (Fig. 9.2). However, the `kd` mechanism involves more range variables, and this is reflected in the choices available in the range variable menu of variable browsers, such as the Plot what? tool (brought up from the primary menu of a Graph). Since `kd` uses potassium, the variables `ek` and `ik` (total K^+ current) appear in this list along with the variables that are explicitly declared in `RANGE` statements or the `STATE` block of `kd.mod` (see Fig. 9.6). The total K^+ current `ik` will differ from `i_kd` only if another mechanism that `WRITE`s `ik` is present in this section.

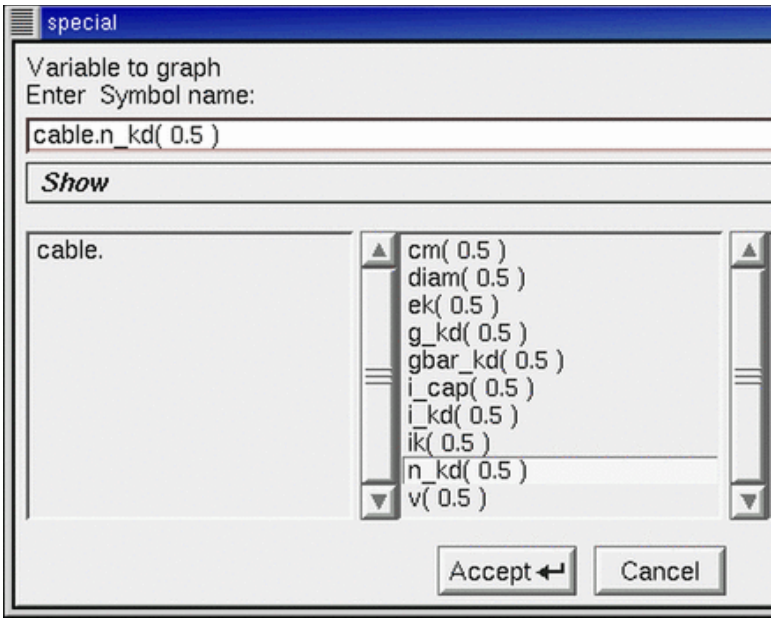


Figure 9.6. A Plot what? tool from a Graph created after the `kd` mechanism was inserted into a section called `cable`. Note the `hoc` names of variables associated with the `kd` mechanism.

Example 9.5: a calcium-activated, voltage-gated current

This model of a potassium current that depends on both voltage and intracellular calcium concentration $[Ca^{2+}]_i$. is based on the work of Moczydlowski and Latorre (Moczydlowski and Latorre 1983). It is basically an elaboration of the HH mechanism in which the forward and backward rates depend jointly on membrane potential and $[Ca^{2+}]_i$.

Here we point out the salient implementational differences between this and the previous model.

```
: Calcium-activated K channel

NEURON {
  SUFFIX cagk
  USEION ca READ cai
  USEION k READ ek WRITE ik
  RANGE gkbar
  GLOBAL oinf, tau
}

UNITS {
  (mV)    = (millivolt)
  (mA)    = (milliamp)
  (S)     = (siemens)
  (molar) = (1/liter)
  (mM)    = (millimolar)
  FARADAY = (faraday) (kilocoulombs)
  R       = (k-mole) (joule/degC)
}

PARAMETER {
  gkbar = 0.01  (S/cm2)
  d1    = 0.84
  d2    = 1.0
  k1    = 0.18  (mM)
  k2    = 0.011 (mM)
  bbar  = 0.28  (/ms)
  abar  = 0.48  (/ms)
}

ASSIGNED {
  cai      (mM)    : typically 0.001
  celsius  (degC)  : typically 20
  v        (mV)
  ek       (mV)
  ik       (mA/cm2)
  oinf
  tau      (ms)
}

STATE { o }    : fraction of channels that are open

BREAKPOINT {
  SOLVE state METHOD cnexp
  ik = gkbar*o*(v - ek)
}

DERIVATIVE state {
  rate(v, cai)
  o' = (oinf - o)/tau
}
```



```
INITIAL {
  rate(v, cai)
  o = oinf
}

: the following are all callable from hoc

FUNCTION alp(v (mV), ca (mM)) (/ms) {
  alp = abar/(1 + exp1(k1,d1,v)/ca)
}

FUNCTION bet(v (mV), ca (mM)) (/ms) {
  bet = bbar/(1 + ca/exp1(k2,d2,v))
}

FUNCTION exp1(k (mM), d, v (mV)) (mM) {
  : numeric constants in an addition or subtraction
  : expression automatically take on the unit values
  : of the other term
  exp1 = k*exp(-2*d*FARADAY*v/R/(273.15 + celsius))
}

PROCEDURE rate(v (mV), ca (mM)) {
  LOCAL a
  : LOCAL variable takes on units of right hand side
  a = alp(v,ca)
  tau = 1/(a + bet(v, ca))
  oinf = a*tau
}
```

Listing 9.5. `cagk.mod`

The `NEURON` block

This potassium conductance depends on $[Ca^{2+}]_i$, so two `USEION` statements are required. Since this potassium channel depends on intracellular calcium concentration, it must `READ cai`. The `RANGE` statement declares the peak conductance density `gkbar`. However, there is no `g`, so this mechanism's ionic conductance will not be visible from `hoc` (in fact, this model doesn't even calculate the activated ionic conductance density). Likewise, there is no `i_cagk` to report this particular current component separately, even though it will be added to the total K^+ current `ik` because of `WRITE ik`.

The variables `oinf` and `tau`, which govern the gating variable `o`, should be accessible in `hoc` for the purpose of seeing how they vary with membrane potential and $[Ca^{2+}]_i$. At the same time, the storage and syntax overhead required for a `RANGE` variable does not seem warranted because it appears unlikely to be necessary or useful to plot either `oinf` or `tau` as a function of space. Therefore they have been declared in a `GLOBAL` statement. On first examination, this might seem to pose a problem. The gating of this K^+ current depends on membrane potential and $[Ca^{2+}]_i$, both of which may vary with location, so how can it be correct for `oinf` and `tau` to be `GLOBAL`s? And if some reason did arise to examine the values of these variables at a particular location, how could this be done? The answers to these questions lie in the `DERIVATIVE` and `PROCEDURE` blocks, as we shall see below.

The `UNITS` block

The last two statements in this block require some explanation. The first parenthesized item on the right hand side of the equal sign is the numeric value of a standard entry in `nrnunits.lib`, which may be expressed on a scale appropriate for physics rather than membrane biophysics. The second parenthesized item rescales this to the biophysically appropriate units chosen for this model. Thus `(faraday)` appears in the units database in terms of coulombs/mole and has a numeric value of 96,485.309, but for this particular mechanism we prefer to use a constant whose units are kilocoulombs/mole. The statement

```
FARADAY = (faraday) (kilocoulombs)
```


results in FARADAY having units of kilocoulombs and a numeric value of 96.485309. The item (k-mole) in the statement

```
R       = (k-mole) (joule/degC)
```

is not kilomoles but instead is a specific entry in the units database equal to the product of Boltzmann's constant and Avogadro's number. The end result of this statement is that R has units of joules/°C and a numeric value of 8.313424. These special definitions of FARADAY and R pertain to this mechanism only; a different mechanism could assign different units and numeric values to these labels.

Another possible source of confusion is the interpretation of the symbol e. Inside a UNITS block this is always the electronic charge (~ $1.6 \cdot 10^{-19}$ coulombs), but elsewhere a *single* number in parentheses is treated as a units conversion factor, e.g. the expression (2e4) is a conversion factor of $2 \cdot 10^4$. Errors involving e in a units expression are easy to make, but they are always caught by modlunit.

Variable declaration blocks

The ASSIGNED block

Comments in this block can be helpful to the user as reminders of "typical" values or usual conditions under which a mechanism operates. For example, the cagk mechanism is intended for use in the context of $[Ca^{2+}]_i$ on the order of 0.001 mM. Similarly, the temperature sensitivity of this mechanism is accommodated by including the global variable celsius, which is used to calculate the rate constants (see **The FUNCTION and**

PROCEDURE blocks below). NEURON's default value for `celsius` is 6.3ºC, but as the comment in this `mod` file points out, the parameter values for this particular mechanism were intended for an "operating temperature" of 20ºC. Therefore the user may need to change `celsius` through `hoc` or the GUI.

The variables `oinf` and `tau`, which were made accessible to NEURON by the `GLOBAL` statement in the `NEURON` block, are given values by the procedure `rate` and are declared as `ASSIGNED`.

The `STATE` block

This mechanism needs a `STATE` block because `o`, the fraction of channels that are open, is described by a differential equation.

Equation definition blocks

The `BREAKPOINT` block

This mechanism does not make its ionic conductance available to `hoc`, so the `BREAKPOINT` block just calculates the ionic current passing through these channels and doesn't bother with separate computation of a conductance.

The `DERIVATIVE` block

The gating variable `o` is governed by a first order differential equation. The procedure `rate()` assigns values to the voltage sensitive parameters of this equation: the steady state value `oinf`, and the time constant `tau`. This answers the first question that was raised above in the discussion of the `NEURON` block. The procedure `rate()` will be

executed individually for each segment in the model that has the `cagk` mechanism. Each time `rate()` is called, its arguments will equal the membrane potential and $[Ca^{2+}]_i$ of the segment that is being processed, since `v` and `cai` are both `RANGE` variables. Therefore `oinf` and `tau` can be `GLOBAL` without destroying the spatial variation of the gating variable `o`.

The `FUNCTION` and `PROCEDURE` blocks

The functions `alp()`, `bet()`, `exp1()`, and the procedure `rate()` implement the mathematical expressions that describe `oinf` and `tau`. To facilitate units checking, their arguments are tagged with the units that they use. For efficiency, `rate()` calls `alp()` once and uses the returned value twice; calculating `oinf` and `tau` separately would have required two calls to `alp()`.

Now we can answer the second question that was raised in the discussion of the `NEURON` block: how to examine the variation of `oinf` and `tau` over space. This is easily done in `hoc` with nested loops, e.g.

```
forall { // iterate over all sections
  for (x) { // iterate over each segment
    rate_cagk(v(x), cai(x))
    // here put statements to plot
    //   or save oinf and tau
  }
}
```

Usage

This mechanism involves both K^+ and Ca^{2+}, so the list of `RANGE` variables displayed by Plot what? has more entries than it did for the `kd` mechanism (Fig. 9.7; compare this

with Fig. 9.6). However, `cai`, `cao`, and `eca` will remain constant unless the section in which this mechanism has been inserted also includes something that can affect calcium concentration (e.g. a pump or buffer).

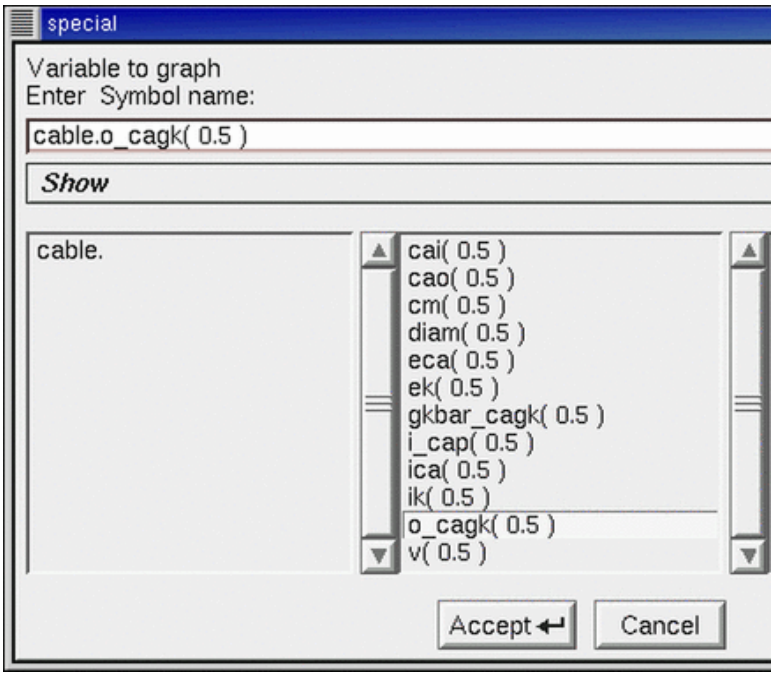


Figure 9.7. A Plot what? tool from a Graph created after the `cagk` mechanism was inserted into a section called `cable`. Note the `hoc` names of variables associated with the `cagk` mechanism.

Example 9.6: extracellular potassium accumulation

Because mechanisms can generate transmembrane fluxes that are attributed to specific ionic species by the `USEION` *`x`* `WRITE` *`ix`* syntax, modeling the effects of restricted diffusion is straightforward. The `kext` mechanism described here emulates potassium accumulation in the extracellular space adjacent to squid axon (Fig. 9.8). The

experiments of Frankenhaeuser and Hodgkin (Frankenhaeuser and Hodgkin 1956) indicated that satellite cells and other extracellular structures act as a diffusion barrier that prevents free communication between this space and the bath. When there is a large efflux of K^+ ions from the axon, e.g. during the repolarizing phase of an action potential or in response to injected depolarizing current, K^+ builds up in this "Frankenhaeuser-Hodgkin space" (F-H space). This elevation of $[K^+]_o$ shifts E_K in a depolarized direction, which has two important consequences. First, it reduces the driving force for K^+ efflux and causes a decline of the outward I_K. Second, when the action potential terminates or the injected depolarizing current is stopped, the residual elevation of $[K^+]_o$ causes an inward current that decays gradually as $[K^+]_o$ equilibrates with $[K^+]_{bath}$.

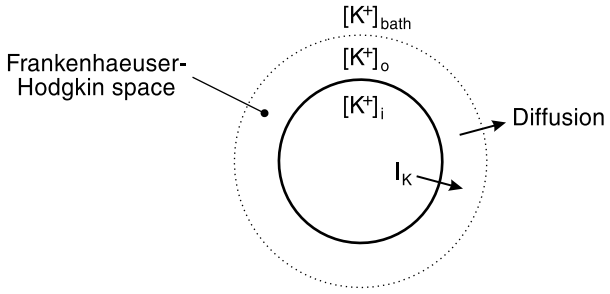


Figure 9.8. Restricted diffusion may cause extracellular potassium accumulation adjacent to the cell membrane. From Fig. 1 in (Hines and Carnevale 2000).


```
: Extracellular potassium ion accumulation

NEURON {
  SUFFIX kext
  USEION k READ ik WRITE ko
  GLOBAL kbath
  RANGE fhspace, txfer
}

UNITS {
  (mV)    = (millivolt)
  (mA)    = (milliamp)
  FARADAY = (faraday) (coulombs)
  (molar) = (1/liter)
  (mM)    = (millimolar)
}

PARAMETER {
  kbath   =  10 (mM)        : seawater (squid axon!)
  fhspace = 300 (angstrom)  : effective thickness of F-H space
  txfer   =  50 (ms)  : tau for F-H space <-> bath exchange = 30-100
}

ASSIGNED { ik  (mA/cm2) }

STATE { ko  (mM) }

BREAKPOINT { SOLVE state METHOD cnexp }

DERIVATIVE state {
  ko' = (1e8)*ik/(fhspace*FARADAY) + (kbath - ko)/txfer
}
```

Listing 9.6. `kext.mod`

The `NEURON` block

A compartment may contain several mechanisms that have direct interactions with ionic concentrations (e.g. diffusion, buffers, pumps). Therefore NEURON must be able to compute the total currents and concentrations consistently. The `USEION` statement sets up the necessary "bookkeeping" by automatically creating a separate mechanism that keeps track of four essential variables: the total outward current carried by an ion, the internal and external concentrations of the ion, and its equilibrium potential (also see **Ion concentrations and equilibrium potential** in **Chapter 8**). In this case the name of the

ion is "k" and the automatically created mechanism is called "k_ion" in the hoc interpreter. The k_ion mechanism has variables ik, ki, ko, and ek, which represent I_K, $[K^+]_i$, $[K^+]_o$, and E_K, respectively. These do not have suffixes; furthermore, they are RANGE variables so they can have different values in every segment of each section in which they exist. In other words, the K^+ current through Hodgkin-Huxley potassium channels near one end of the section cable is cable.ik_hh(0.1), but the total K^+ current generated by all sources, including other ionic conductances and pumps, would be cable.ik(0.1).

This mechanism computes $[K^+]_o$ from the outward potassium current, so it READs ik and WRITEs ko. When a mechanism WRITEs a particular ionic concentration, it sets the value for that concentration at all locations in every section into which it has been inserted. This has an important consequence: in any given section, no ionic concentration should be "written" by more than one mechanism.

The bath is assumed to be a large, well stirred compartment that envelops the entire "experimental preparation." Therefore kbath is a GLOBAL variable so that all sections that contain the kext mechanism will have the same numeric value for $[K^+]_{bath}$. Since this would be one of the controlled variables in an experiment, the value of kbath is specified by the user and remains constant during a simulation. The thickness of the F-H space is fhspace, the time constant for equilibration with the bath is txfer, and both are RANGE variables so they can vary along the length of each section.

Variable declaration blocks

The PARAMETER block

The default value of kbath is set to 10 mM, consistent with the composition of seawater (Frankenhaeuser and Hodgkin 1956). Since kbath is GLOBAL, a single hoc statement can change this to a new value that will affect all occurrences of the kext mechanism, e.g. kbath_kext = 8 would change it to 8 mM everywhere.

The STATE block

Ionic concentration is a STATE of a mechanism only if that mechanism calculates the concentration in a DERIVATIVE or KINETIC block. This model computes ko, the potassium concentration in the F-H space, according to the dynamics specified by an ordinary differential equation.

Equation definition blocks

The BREAKPOINT block

This mechanism involves a single differential equation that tells the rate of change of ko, the K^+ concentration in the F-H space. The choice of integration method in NMODL is based on the fact that the equation is linear in ko. The total K^+ current ik might also vary during a time step (see the DERIVATIVE block) if membrane potential, some K^+ conductance, or ko itself is changing rapidly. In a simulation where such rapid changes

are likely to occur, proper modeling practice would lead one either to use NEURON with CVODE, or to use a fixed time step that is short compared to the rate of change of `ik`.

The `INITIAL` block

How to provide for initialization of variables is a recurring question in model implementation, and here it comes again. The answer is important because it bears directly on how the model will be used. The only `STATE` in this mechanism is the ionic concentration `ko`, which we could initialize in several different ways. The simplest might be with the `INITIAL` block

```
INITIAL {
  ko = kbath
}
```

but this seems too limiting. One alternative is to declare a new `RANGE` variable `ko0` in the `NEURON` block, specify its value in the `PARAMETER` block

```
PARAMETER {
   . . .
  ko0     =  10 (mM)  =  10 (mM)
}
```

and use this `INITIAL` block

```
INITIAL {
  ko = ko0
}
```

This would be a very flexible implementation, allowing `ko0` to vary with location wherever `kext` has been `inserted`. But some care is needed in its use, because ion concentration assignment in an `INITIAL` block can result in an inconsistent initialization

on return from `finitialize()` (see **Ion concentrations and equilibrium potentials** in **Chapter 8**).

So for this example we have decided to let the initial value of `ko` be controlled by the built-in `hoc` variable `ko0_k_ion` (see ***Initializing concentrations in* hoc** in **Chapter 8**). To make our mechanism rely on `ko0_k_ion` for the initial value of `ko`, we merely omit any `ko = . . .` assignment statement from the `INITIAL` block. Since `ko` is `kext`'s only `STATE`, we don't need an `INITIAL` block at all. This might seem a less flexible approach than using our own `ko0 RANGE` variable, because `ko0_k_ion` is a global variable (has the same value wherever `ko` is defined), but ***Initializing concentrations in* hoc** in **Chapter 8** shows how to work around this apparent limitation.

The `DERIVATIVE` block

At the core of this mechanism is a single differential equation that relates $d[K^+]_o/dt$ to the sum of two terms. The first term describes the contribution of `ik` to $[K^+]_o$, subject to the assumption that the thickness F-H space is much smaller than the diameter of the section. The units conversion factor of 10^8 is required because `fhspace` is given in Ångstroms. The second term describes the exchange of K^+ between the bath and the F-H space.

Usage

If this mechanism is present in a section, the following `RANGE` variables will be accessible through `hoc`: $[K^+]$ inside the cell and within the F-H space (`ki` and `ko`);

equilibrium potential and total current for K (`ek` and `ik`); thickness of the F-H space and the rate of equilibration between it and the bath (`fhspace_kext` and `txfer_kext`). The bath [K^+] will also be available as the global variable `kbath_kext`.

General comments about kinetic schemes

Kinetic schemes provide a high level framework that is perfectly suited for compact and intuitively clear specification of models that involve discrete states in which material is conserved. The basic notion is that flow out of one state equals flow into another (also see **Chemical reactions** in **Chapter 3**). Almost all models of membrane channels, chemical reactions, macroscopic Markov processes, and diffusion can be elegantly expressed through kinetic schemes. It will be helpful to review some fundamentals before proceeding to specific examples of mechanisms implemented with kinetic schemes.

The unknowns in a kinetic scheme, which are usually concentrations of individual reactants, are declared in the `STATE` block. The user expresses the kinetic scheme with a notation that is very similar to a list of simultaneous chemical reactions. The NMODL translator converts the kinetic scheme into a family of ODEs whose unknowns are the `STATE`s. Hence

```
STATE { mc   m }
KINETIC scheme1 {
  ~ mc <-> m (a(v), b(v))
}
```

is equivalent to


```
DERIVATIVE scheme1 {
  mc' = -a(v)*mc + b(v)*m
  m'  =  a(v)*mc - b(v)*m
}
```

The first character of a reaction statement is the tilde "~", which is used to immediately distinguish this kind of statement from other sequences of tokens that could be interpreted as an expression. The expressions on the left and right of the three character reaction indicator "<->" specify the reactants. The two expressions in parentheses are the forward and reverse reaction rates (here the rate functions `a(v)` and `b(v)`). Immediately after each reaction, the variables `f_flux` and `b_flux` are assigned the values of the forward and reverse fluxes respectively. These can be used in assignment statements such as

```
~ cai + pump <-> capump (k1,k2)
~ capump <-> pump + cao (k3,k4)
ica = (f_flux - b_flux)*2*Faraday/area
```

In this case, the forward flux is `k3*capump`, the reverse flux is `k4*pump*cao`, and the "positive outward" current convention is consistent with the sign of the expression for `ica` (in the second reaction, forward flux means positive ions move from the inside to the outside).

More complicated reaction sequences such as the wholly imaginary

```
KINETIC scheme2 {
  ~ 2A + B <-> C (k1,k2)
  ~  C + D <-> A + 2B (k3,k4)
}
```

begin to show the clarity of expression and suggest the comparative ease of modification of the kinetic representation over the equivalent but stoichiometrically confusing


```
DERIVATIVE scheme2 {
  A' = -2*k1*A^2*B + 2*k2*C   + k3*C*D   - k4*A*B^2
  B' =   -k1*A^2*B   + k2*C + 2*k3*C*D - 2*k4*A*B^2
  C' =    k1*A^2*B   - k2*C   - k3*C*D   + k4*A*B^2
  D' =                        - k3*C*D   + k4*A*B^2
}
```

Clearly a statement such as

```
~ calmodulin + 3Ca <-> active (k1, k2)
```

would be easier to modify (e.g. so it requires combination with 4 calcium ions) than the relevant term in the three differential equations for the STATEs that this reaction affects. The kinetic representation is easy to debug because it closely resembles familiar notations and is much closer to the conceptualization of what is happening than the differential equations would be.

Another benefit of kinetic schemes is the simple polynomial nature of the flux terms, which allows the translator to easily perform a great deal of preprocessing that makes implicit numerical integration more efficient. Specifically, the nonzero $\partial y'_i / \partial y_j$ elements (partial derivatives of dy_i / dt with respect to y_j) of the sparse matrix are calculated analytically in NMODL and collected into a C function that is called by solvers to calculate the Jacobian. Furthermore, the form of the reaction statements determines if the scheme is linear, obviating an iterative computation of the solution. Voltage-sensitive rates are allowed, but to guarantee numerical stability, reaction rates should not be functions of STATEs. Thus writing the calmodulin example as

```
~ calmodulin <-> active (k3*Ca^3, k2)
```


will work but is potentially unstable if `Ca` is a `STATE` in other simultaneous reactions in the same `mod` file. Variable time step methods such as CVODE will compensate by reducing `dt`, but this will make the simulation run more slowly.

Kinetic scheme representations provide a great deal of leverage because a single compact expression is equivalent to a large amount of C code. One special advantage from the programmer's point of view is the fact that these expressions are independent of the solution method. Different solution methods require different code, but the NMODL translator generates this code automatically. This saves the user's time and effort and ensures that all code expresses the same mechanism. Another advantage is that the NMODL translator handles the task of interfacing the mechanism to the remainder of the program. This is a tedious exercise that would require the user to have special knowledge that is not relevant to neurophysiology and which may change from version to version.

Special issues are raised by mechanisms that involve fluxes between compartments of different size, or whose reactants have different units. The first of the following examples has none of these complications, which are addressed later in models of diffusion and active transport.

Example 9.7: kinetic scheme for a voltage-gated current

This illustration of NMODL's facility for handling kinetic schemes implements a simple three state model for the conductance state transitions of a voltage gated potassium current

$$\mathrm{C}_1 \underset{kb_1}{\overset{kf_1}{\rightleftarrows}} \mathrm{C}_2 \underset{kb_2}{\overset{kf_2}{\rightleftarrows}} \mathrm{O} \qquad \text{Eq. 9.2}$$

The closed states are C_1 and C_2, the open state is O, and the rates of the forward and backward state transitions are calculated in terms of the equilibrium constants and time constants of the isolated reactions through the familiar expressions $K_i(v) = kf_i / kb_i$ and $\tau_i(v) = 1/(kf_i + kb_i)$. The equilibrium constants $K_i(v)$ are the Boltzmann factors $K_1 = \mathrm{e}^{[k_2(d_2 - v) - k_1(d_1 - v)]}$ and $K_2 = \mathrm{e}^{-k_2(d_2 - v)}$, where the energies of states C_1, C_2, and O are 0, $k_1(d_1 - v)$, and $k_2(d_2 - v)$ respectively.

The typical sequence of analysis is to determine the constants k_1, d_1, k_2, and d_2 by fitting the steady state voltage clamp data, and then to find the voltage sensitive transition time constants $\tau_1(v)$ and $\tau_2(v)$ from the temporal properties of the clamp current at each voltage pulse level. In this example the steady state information has been

incorporated in the NMODL code, and the time constants are conveyed by tables (arrays) that are created within the interpreter.

```
: Three state kinetic scheme for HH-like potassium channel
: Steady state v-dependent state transitions have been fit
: Needs v-dependent time constants
:   from tables created under hoc

NEURON {
  SUFFIX k3st
  USEION k READ ek WRITE ik
  RANGE g, gbar
}

UNITS { (mV) = (millivolt) }

PARAMETER {
  gbar = 33     (millimho/cm2)
  d1   = -38    (mV)
  k1   = 0.151  (/mV)
  d2   = -25    (mV)
  k2   = 0.044  (/mV)
}

ASSIGNED {
  v    (mV)
  ek   (mV)
  g    (millimho/cm2)
  ik   (milliamp/cm2)
  kf1  (/ms)
  kb1  (/ms)
  kf2  (/ms)
  kb2  (/ms)
}

STATE { c1 c2 o }

BREAKPOINT {
  SOLVE kin METHOD sparse
  g = gbar*o
  ik = g*(v - ek)*(1e-3)
}

INITIAL { SOLVE kin STEADYSTATE sparse }

KINETIC kin {
  rates(v)
  ~ c1 <-> c2   (kf1, kb1)
  ~ c2 <-> o    (kf2, kb2)
  CONSERVE c1 + c2 + o = 1
}

FUNCTION_TABLE tau1(v(mV)) (ms)
FUNCTION_TABLE tau2(v(mV)) (ms)
```



```
PROCEDURE rates(v(millivolt)) {
  LOCAL K1, K2
  K1 = exp(k2*(d2 - v) - k1*(d1 - v))
  kf1 = K1/(tau1(v)*(1+K1))
  kb1 = 1/(tau1(v)*(1+K1))
  K2 = exp(-k2*(d2 - v))
  kf2 = K2/(tau2(v)*(1+K2))
  kb2 = 1/(tau2(v)*(1+K2))
}
```

Listing 9.7. `k3st.mod`

The `NEURON` block

With one exception, the `NEURON` block of this model is essentially the same as for the delayed rectifier presented in **Example 9.4: a voltage-gated current**. The difference is that, even though this model contributes to the total K^+ current `ik`, its own current is not available separately (i.e. there will be no `ik_k3st` at the `hoc` level) because `ik` is not declared as a `RANGE` variable.

Variable declaration blocks

The `STATE` block

The `STATE`s in this mechanism are the fractions of channels that are in closed states 1 or 2 or in the open state. Since the total number of channels in all states is conserved, the sum of the `STATE`s must be unity, i.e. `c1` + `c2` + `o` = 1. This conservation rule means that the `k3st` mechanism really has only two independent `STATE` variables, a fact that underscores the difference between a `STATE` in NMODL and the general concept of a state variable. It also affects how NMODL sets up the equations that are to be solved, as we will see in the discussion of the `KINETIC` block below.

Not all reactants have to be STATEs. If the reactant is an ASSIGNED or PARAMETER variable, then a differential equation is not generated for it, and it is treated as constant for the purposes of calculating the declared STATEs. Statements such as

```
PARAMETER {kbath (mM)}
STATE {ko (mM)}
KINETIC scheme3 {
  ~ ko <-> kbath (r, r)
}
```

are translated to the single ODE equivalent

```
ko' = r*(kbath - ko)
```

i.e. ko tends exponentially to the steady state value of kbath.

Equation definition blocks

The BREAKPOINT block

The recommended idiom for integrating a kinetic scheme is

```
BREAKPOINT {
  SOLVE scheme METHOD sparse
  . . .
}
```

which integrates the STATEs in the scheme one dt step per call to fadvance(). The sparse method is generally faster than computing the full Jacobian matrix, though both use Newton iterations to advance the STATEs with a fully implicit method (first order correct). Additionally, the sparse method separates the Jacobian evaluation from the calculation of the STATE derivatives, thus allowing adaptive integration methods, such as CVODE, to efficiently compute only what is needed to advance the STATEs. Nonimplicit methods, such as Runge-Kutta or forward Euler, should be avoided since kinetic schemes

commonly have very wide ranging rate constants that make these methods numerically unstable with reasonable `dt` steps. In fact, it is not unusual to specify equilibrium reactions such as

```
~ A <-> B (1e6*sqrt(K), 1e6/sqrt(K))
```

which can only be solved by implicit methods.

The `INITIAL` block

Initialization of a kinetic scheme to its steady state is accomplished with

```
INITIAL {
   SOLVE scheme STEADYSTATE sparse
}
```

Appropriate `CONSERVE` statements should be part of the scheme (see the following discussion of the `KINETIC` block) so that the equivalent system of ODEs is linearly independent. It should be kept in mind that source fluxes (constant for infinite time) have a strong effect on the steady state. Finally, it is crucial to test the scheme in NEURON under conditions in which the correct behavior is known.

The `KINETIC` block

The voltage-dependent rate constants are computed in procedure `rates()`. That procedure computes the equilibrium constants `K1` and `K2` from the constants `k1`, `d1`, `k2`, and `d2`, whose empirically determined default values are given in the `PARAMETER` block, and membrane potential `v`. The time constants `tau1` and `tau2`, however, are found from tables created under `hoc` (see **The `FUNCTION_TABLE`s** below).

The other noteworthy item in this block is the CONSERVE statement. As mentioned above in **General comments about kinetic schemes**, the basic idea is to systematically account for conservation of material. If there is neither a source nor a sink reaction for a STATE, the differential equations are not linearly independent when steady states are calculated (dt approaches infinity). For example, in scheme1 above the steady state condition m' = mc' = 0 yields two identical equations. Steady states can be approximated by integrating for several steps from any initial condition with large dt, but roundoff error can be a problem if the Jacobian matrix is nearly singular. To help solve the equations while maintaining strict numerical conservation throughout the simulation (no accumulation of roundoff error), the user is allowed to explicitly specify conservation equations with the CONSERVE statement. The conservation law for scheme1 is specified in NMODL by

```
CONSERVE m + mc = 1
```

The CONSERVE statement does not add to the information content of a kinetic scheme and should be considered only as a hint to the translator. The NMODL translator uses this algebraic equation to replace the ODE for the last STATE on the left side of the equal sign. If one of the STATE names is an array, the conservation equation will contain an implicit sum over the array. If the last STATE is an array, then the ODE for the last STATE array element will be replaced by the algebraic equation. The choice of which STATE ODE is replaced by the algebraic equation depends on the implementation, and does not affect the solution (to within roundoff error). If a CONSERVEd STATE is relative to a compartment size, then compartment size is implicitly taken into account for the STATEs on the left hand side of the CONSERVE equation (see **Example 9.8: calcium**

diffusion with buffering for discussion of the `COMPARTMENT` statement). The right hand side is merely an expression, in which any necessary compartment sizes must be included explicitly.

Thus in a calcium pump model

$$\mathrm{Ca}_{\mathrm{int}} + \mathrm{Pump} \underset{k_2}{\overset{k_1}{\rightleftarrows}} \mathrm{Ca}\cdot\mathrm{Pump} \underset{k_4}{\overset{k_3}{\rightleftarrows}} \mathrm{Ca}_{\mathrm{ext}} + \mathrm{Pump} \qquad \text{Eq. 9.3}$$

the pump is conserved and one could write

```
CONSERVE pump + pumpca = total_pump * pumparea
```

The `FUNCTION_TABLE`s

As noted above, the steady state clamp data define the voltage dependence of K_1 and K_2, but a complete description of the K^+ current requires analysis of the temporal properties of the clamp current to determine the rate factors at each of the command potentials. The result would be a list or table of membrane potentials with associated time constants. One way to handle these numeric values would be to fit them with a pair of approximating functions, but the tactic used in this example is to leave them in tabular form for NMODL's `FUNCTION_TABLE` to deal with.

This is done by placing the numeric values in three `Vector`s, say `v_vec`, `tau1_vec`, and `tau2_vec`, where the first is the list of voltages and the other two are the corresponding time constants. These `Vector`s would be attached to the `FUNCTION_TABLE`s of this model with the `hoc` commands


```
table_tau1_k3st(tau1_vec, v_vec)
table_tau2_k3st(tau2_vec, v_vec)
```

Then whenever `tau1(x)` is called in the NMODL file, or `tau1_k3st(x)` is called from `hoc`, the returned value is interpolated from the array.

A useful feature of `FUNCTION_TABLE`s is that, prior to developing the `Vector` database, they can be attached to a scalar value, as in

```
table_tau1_k3st(100)
```

effectively becoming constant functions. Also `FUNCTION_TABLE`s can be declared with two arguments and attached to doubly dimensioned `hoc` arrays. In this case the table is linearly interpolated in both dimensions. This is useful with rates that depend on both voltage and calcium.

Usage

Inserting this mechanism into a section makes the `STATE`s `c1_k3st`, `c2_k3st`, and `o_k3st` available at the `hoc` level, as well as the conductances `gbar_k3st` and `g_k3st`.

Example 9.8: calcium diffusion with buffering

This mechanism illustrates how to use kinetic schemes to model intracellular Ca^{2+} diffusion and buffering. It differs from the prior example in several important aspects: Ca^{2+} is not conserved but instead enters as a consequence of the transmembrane Ca^{2+} current; diffusion involves the exchange of Ca^{2+} between compartments of unequal size; Ca^{2+} is buffered.

Only free Ca^{2+} is assumed to be mobile, whereas bound Ca^{2+} and free buffer are stationary. Buffer concentration and rate constants are based on the bullfrog sympathetic ganglion cell model described by Yamada et al. (Yamada et al. 1998). For a thorough treatment of numeric solution of the diffusion equations the reader is referred to Oran and Boris (Oran and Boris 1987).

Modeling diffusion with kinetic schemes

Diffusion is modeled as the exchange of Ca^{2+} between adjacent compartments. We begin by examining radial diffusion, and defer consideration of longidudinal diffusion to **Equation definition blocks: The `KINETIC` block** later in this example.

For radial diffusion, the compartments are a series of concentric shells around a cylindrical core, as shown in Fig. 9.9 for `Nannuli` = 4. The index of the outermost shell is 0 and the index of the core is `Nannuli` - 1. The outermost shell is half as thick as the others so that $[Ca^{2+}]$ will be second order correct with respect to space at the surface of the segment. Concentration is also second order correct midway through the thickness of the other shells and at the center of the core. These depths are indicated by "x" in Fig. 9.9. The radius of the cylindrical core equals the thickness of the outermost shell, and the intervening `Nannuli` - 2 shells each have thickness Δr = `diam` / 2 (`Nannuli` - 1), where `diam` is the diameter of the segment.

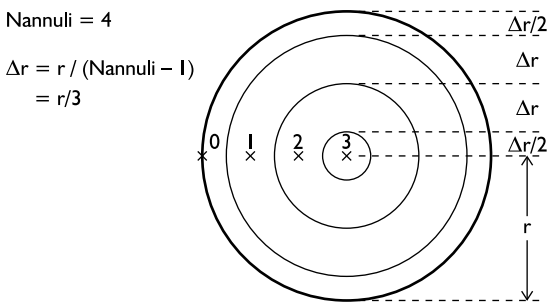


Figure 9.9. Diagram of the concentric shells used to model radial diffusion. The × mark the radial distances at which concentration will be second order correct in space.

Because segment diameter and the number of shells affect the dimensions of the shells, they also affect the time course of diffusion. The flux between adjacent shells is $\Delta[Ca^{2+}]\ D_{Ca}\ A / \Delta r$, where $\Delta[Ca^{2+}]$ is the concentration difference between the shell centers, D_{Ca} is the diffusion coefficient for Ca^{2+}, A is the area of the boundary between shells, and Δr is the distance between their centers. This suggests that diffusion can be described by the basic kinetic scheme

```
FROM i = 0 TO Nannuli-2 {
  ~ ca[i] <-> ca[i+1] (f[i+1], f[i+1])
}
```

where `Nannuli` is the number of shells, `ca[i]` is the concentration midway through the thickness of shell `i` (except for `ca[0]` which is the concentration at the outer surface of shell 0), and the rate constants `f[i+1]` equal $D_{Ca}\ A_{i+1} / \Delta r$. For each adjacent pair of shells, both A_{i+1} and Δr are directly proportional to segment diameter. Therefore the ratios $A_{i+1} / \Delta r$ depend only on shell index, i.e. once they have been computed for one

segment, they can be used for all segments that have the same number of radial compartments regardless of segment diameter.

As it stands, this kinetic scheme is dimensionally incorrect. Dimensional consistency requires that the product of STATEs and rates be in units of STATE per time (also see **Compartment size** in the section on **Chemical reactions** in **Chapter 3**). In the present example the STATEs `ca[]` are intensive variables (concentration, or mass/volume), so the product of `f[]` and `ca[]` must be in units of concentration/time. However, the rates `f[]` have units of volume/time, so this product is actually in units of mass/time, i.e. it is a flux that signifies the rate at which Ca^{2+} is entering or leaving a compartment. This is the time derivative of an extensive variable (i.e. of a variable that describes amount of material).

This disparity is corrected by specifying STATE volumes with the `COMPARTMENT` statement, as in

```
COMPARTMENT volume {state1 state2 . . . }
```

where the STATEs named in the braces have the same compartment volume given by the *volume* expression after the `COMPARTMENT` keyword. The volume merely multiplies the *d*STATE*/dt* left hand side of the equivalent differential equations, converting it to an extensive quantity and making it consistent with flux terms in units of absolute quantity per time.

The volume of each cylindrical shell depends on its index and the total number of shells, and is proportional to the square of segment diameter. Consequently the volumes can be computed once for a segment with unit diameter and then scaled by `diam^2` for use in each segment that has the same `Nannuli`. The equations that describe the radial

movement of Ca^{2+} are independent of segment length. Therefore it is convenient to express shell volumes and surface areas in units of μm^2 (volume/length) and μm (area/length), respectively.

```
: Calcium ion accumulation with radial and longitudinal diffusion

NEURON {
  SUFFIX cadifus
  USEION ca READ cai, ica WRITE cai
  GLOBAL vrat, TotalBuffer  : vrat must be GLOBAL--see INITIAL block
                            : however TotalBuffer may be RANGE
}

DEFINE Nannuli 4  : must be >=2 (i.e. at least shell and core)

UNITS {
  (molar) = (1/liter)
  (mM)    = (millimolar)
  (um)    = (micron)
  (mA)    = (milliamp)
  FARADAY = (faraday)  (10000 coulomb)
  PI      = (pi)       (1)
}

PARAMETER {
  DCa   = 0.6 (um2/ms)
  k1buf = 100 (/mM-ms) : Yamada et al. 1989
  k2buf = 0.1 (/ms)
  TotalBuffer = 0.003  (mM)
}

ASSIGNED {
  diam      (um)
  ica       (mA/cm2)
  cai       (mM)
  vrat[Nannuli]  (1) : dimensionless
             : vrat[i] is volume  of annulus i of a 1um diameter cylinder
             : multiply by diam^2 to get volume per um length
  Kd        (/mM)
  B0        (mM)
}

STATE {
  : ca[0] is equivalent to cai
  : ca[] are very small, so specify absolute tolerance
  ca[Nannuli]       (mM) <1e-10>
  CaBuffer[Nannuli] (mM)
  Buffer[Nannuli]   (mM)
}

BREAKPOINT { SOLVE state METHOD sparse }
```



```
LOCAL factors_done

INITIAL {
   if (factors_done == 0) {  : flag becomes 1 in the first segment
      factors_done = 1       :   all subsequent segments will have
      factors()              :   vrat = 0 unless vrat is GLOBAL
   }

  Kd = k1buf/k2buf
  B0 = TotalBuffer/(1 + Kd*cai)

  FROM i=0 TO Nannuli-1 {
    ca[i] = cai
    Buffer[i] = B0
    CaBuffer[i] = TotalBuffer - B0
  }
}

LOCAL frat[Nannuli]  : scales the rate constants for model geometry

PROCEDURE factors() {
  LOCAL r, dr2
  r = 1/2                : starts at edge (half diam)
  dr2 = r/(Nannuli-1)/2  : full thickness of outermost annulus,
                         : half thickness of all other annuli
  vrat[0] = 0
  frat[0] = 2*r
  FROM i=0 TO Nannuli-2 {
    vrat[i] = vrat[i] + PI*(r-dr2/2)*2*dr2  : interior half
    r = r - dr2
    frat[i+1] = 2*PI*r/(2*dr2)  : outer radius of annulus
                                : div by distance between centers
    r = r - dr2
    vrat[i+1] = PI*(r+dr2/2)*2*dr2  : outer half of annulus
  }
}

LOCAL dsq, dsqvol  : can't define local variable in KINETIC block
                   :   or use in COMPARTMENT statement

KINETIC state {
  COMPARTMENT i, diam*diam*vrat[i] {ca CaBuffer Buffer}
  LONGITUDINAL_DIFFUSION i, DCa*diam*diam*vrat[i] {ca}
  ~ ca[0] << (-ica*PI*diam/(2*FARADAY))  : ica is Ca efflux
  FROM i=0 TO Nannuli-2 {
    ~ ca[i] <-> ca[i+1]  (DCa*frat[i+1], DCa*frat[i+1])
  }
  dsq = diam*diam
  FROM i=0 TO Nannuli-1 {
    dsqvol = dsq*vrat[i]
    ~ ca[i] + Buffer[i] <-> CaBuffer[i]  (k1buf*dsqvol, k2buf*dsqvol)
  }
  cai = ca[0]
}
```

Listing 9.8. `cadif.mod`

The `NEURON` block

This model `READ`s `cai` to initialize both intracellular [Ca^{2+}] and the buffer (see **The `INITIAL` block** below), and it `WRITE`s `cai` because it computes [Ca^{2+}] in the outermost shell during a simulation run. It also `READ`s `ica`, which is the Ca^{2+} influx into the outermost shell.

There are two `GLOBAL`s. One is the total buffer concentration `TotalBuffer`, which is assumed to be uniform throughout the cell. The other is `vrat`, an array whose elements will be the numeric values of the (volume/length) of the shells for a segment with unit diameter. These values are computed by `PROCEDURE factors()` near the end of Listing 9.8. As noted above, a segment with diameter `diam` has shells with volume/length equal to `diam^2 * vrat[i]`. Because each instance of this mechanism has the same number of shells, the same `vrat[i]` can be used to find the shell volumes at each location in the model cell where the mechanism exists.

The `DEFINE` statement sets the number of shells to 4. Many of the variables in this model are arrays, and NMODL arrays are not dynamic. Instead, their size must be specified when the NMODL code is translated to C.

The `UNITS` block

Faraday's constant is rescaled here so we won't have to include a separate units conversion factor in the statement in the `KINETIC` block where transmembrane current `ica` is reckoned as the efflux of Ca^{2+} from the outermost shell. Each statement in a

`UNITS` block must explicitly assert the units that are involved, so the statement that assigns the value 3.141 . . . to `PI` includes a `(1)` to mark it as a dimensionless constant.

Variable declaration blocks

The `ASSIGNED` block

The variable `vrat` is declared to be an array with `Nannuli` elements. As with C, array indices run from 0 to `Nannuli` - 1. The variables `Kd` and `B0` are the dissociation constant for the buffer and the initial value of free buffer, which are computed in the `INITIAL` block (see below). Both the total buffer and the initial concentration of Ca^{2+} are assumed to be uniform throughout all shells, so a scalar is used for `B0`.

The `STATE` block

In addition to diffusion, this mechanism involves Ca^{2+} buffering

$$\mathrm{Ca} + \mathrm{Buffer} \underset{k_2^{buf}}{\overset{k_1^{buf}}{\rightleftarrows}} \mathrm{Ca \cdot Buffer} \qquad \text{Eq. 9.4}$$

This happens in each of the shells, so `ca`, `Buffer` and `CaBuffer` are all arrays.

The declaration of `ca[]` uses the syntax *`state (units) <absolute_tolerance>`* to specify the local absolute error tolerance that will be employed by CVODE. The solver tries to use a step size for which the local error ϵ_i of each $state_i$ satisfies at least one of these two inequalities:

$$\epsilon_i < relative\_tolerance \cdot |state_i|$$

or

$$\epsilon_i < absolute\_tolerance$$

The default values for these tolerances are 0 and 10^{-3}, respectively, so only a `STATE` that is extremely small (such as intracellular [Ca^{2+}]) needs to have its absolute tolerance specified. As an alternative to specifying a smaller absolute tolerance, `ca[]` could have been defined in terms of units such as micromolar or nanomolar, which would have increased the numeric value of these variables. This would necessitate different units conversion factors in many of the statements that involve `ca[]`. For example, the assignment for `cai` (which is required to be in mM) would be `cai = (1e-6)*ca[0]`.

`LOCAL` variables declared outside of equation definition blocks

A `LOCAL` variable that is declared outside of an equation definition block is equivalent to a static variable in C. That is, it is visible throughout the mechanism (but not at the `hoc` level), it retains its value, and it is shared between all instances of a given mechanism. The initial value of such a variable is 0.

This particular mechanism employs four variables of this type: `factors_done`, `frat[]`, `dsq`, and `dsqvol`. The meaning of each of these is discussed below.

Equation definition blocks

The `INITIAL` block

Initialization of this mechanism is a two step process. The first step is to use `PROCEDURE factors()` (see below) to set up the geometry of the model by computing `vrat[]` and `frat[]`, the arrays of units conversion factors that are applied to the shell volumes and rate constants. This only has to be done once because the same conversion factors are used for all segments that have the same number of shells, as noted above in **Modeling diffusion with kinetic schemes**. The variable `factors_done` is a flag that indicates whether `vrat[]` and `frat[]` have been computed. The NMODL keyword `LOCAL` means that the value of `factors_done` will be the same in all instances of this mechanism, but that it will not be visible at the `hoc` level. Therefore `factors()` will be executed only once, regardless of how many segments contain the `cadifus` mechanism.

The second step is to initialize the mechanism's `STATE`s. This mechanism assumes that the total buffer concentration and the initial free calcium concentration are uniform in all shells, and that buffering has reached its steady state. Therefore the initial concentration of free buffer is computed from the initial [Ca^{2+}] and the buffer's dissociation constant. It should be noted that the value of `cai` will be set to `cai0_ca_ion` just prior to executing the code in the `INITIAL` block (see **Ion concentrations and equilibrium potentials** in **Chapter 8**).

It may be instructive to compare this initialization strategy with the approach that was used for the voltage-gated current of Listing 9.7 (`k3st.mod`). That previous example

initialized the STATE through numeric solution of a kinetic scheme, so its KINETIC block required a CONSERVE statement to ensure that the equivalent system of ODEs would be linearly independent. Here, however, the STATEs are initialized by explicit algebraic assignment, so no CONSERVE statement is necessary.

PROCEDURE factors()

The arrays vrat[] and frat[], which are used to scale the shell volumes and rate constants to ensure consistency of units, are computed here. Their values depend only on the number of shells, so they do not have to be recomputed if diam or DFree is changed.

The elements of vrat[] are the volumes of a set of concentric cylindrical shells, whose total volume equals the volume of a cylinder with diameter and length of 1 μm. These values are computed in two stages by the FROM i=0 TO Nannuli-2 { } loop. The first stage finds the volume of the outer half and the second finds the volume of the inner half of the shell.

The frat array is declared to be LOCAL because it applies to all segments that have the cadifus mechanism, but it is unlikely to be of interest to the user and therefore does not need to be visible at the hoc level. This contrasts with vrat, which is declared as GLOBAL within the NEURON block so that the user can see its values. The values frat [i+1] equal A_{i+1} / Δr, where A_{i+1} is the surface area between shells i and i+1 for 0 ≤ i < Nannuli, and Δr is the distance between shell centers (radius / (Nannuli - 1)).

The KINETIC block

The first statement in this block specifies the shell volumes for the STATEs ca, CaBuffer, and Buffer. As noted above in **Modeling diffusion with kinetic schemes**, these volumes equal the elements of vrat[] multiplied by the square of the segment diameter. This mechanism involves many compartments whose relative volumes are specified by the elements of an array, so we can deal with all compartments with a single statement of the form

```
COMPARTMENT index, volume[index] { state1 state2 . . . }
```

where the diffusing STATEs are listed inside the braces.

Next in this block is a LONGITUDINAL_DIFFUSION statement, which specifies that this mechanism includes "nonlocal" diffusion, i.e. longitudinal diffusion along a section and into connecting sections. The syntax for scalar STATEs is

```
LONGITUDINAL_DIFFUSION flux_expr { state1 state2 . . . }
```

where *flux_expr* is the product of the diffusion constant and the area of the cross section between adjacent compartments. Units of the *flux_expr* must be (micron^4/ms), i.e. the diffusion constant has units of (micron^2/ms) and the cross sectional area has units of (micron^2). For cylindrical shell compartments, the cross sectional area is just the volume per unit length. If the states are arrays then all elements are assumed to diffuse between corresponding volumes in adjacent segments and the iteration variable must be specified as in

```
LONGITUDINAL_DIFFUSION index, flux_expr(index) { state1 state2 . . . }
```


A `COMPARTMENT` statement is also required for the diffusing `STATE`s and the units must be (micron^2), i.e. (micron^3/micron).

The compactness of `LONGITUDINAL_DIFFUSION` specification contrasts nicely with the great deal of trouble imposed on the computational methods used to solve the equations. The standard backward Euler method, historically the default method used by NEURON (see **Chapter 4**), can no longer find steady states with extremely large (e.g. 10^9 ms) steps since not every Jacobian element for both flux and current with respect to voltage and concentration is presently accurately computed. CVODE works well for these problems since it does not allow `dt` to grow beyond the point of numerical instability. Despite these occasional limitations on numerical efficiency, it is satisfying that, as methods evolve to handle these problems more robustly, the specification of the models does not change.

The third statement in this block is equivalent to a differential equation that describes the contribution of transmembrane calcium current to Ca^{2+} in the outermost shell. The << signifies an explicit flux. Because of the `COMPARTMENT` statement, the left hand side of the differential equation is not $d[Ca^{2+}]_0/dt$ but $d(total\ Ca^{2+}$ in the outermost shell$)/dt$. This is consistent with the right hand side of the equation, which is in units of mass per time.

Next is the kinetic scheme for radial diffusion. The rate constants in this scheme equal the product of `DCa` and the factor `frat[]` for reasons that were explained earlier in **Modeling diffusion with kinetic schemes**.

It may not be immediately clear why the rate constants in the kinetic scheme for Ca^{2+} buffering are scaled by the compartment volume `dsqvol`; however, the reason will become obvious when one recalls that the `COMPARTMENT` statement at the beginning of the `KINETIC` block has converted the units of the *d*`STATE`*/dt* on the left hand side of the equivalent differential equations from concentration per time to mass per time. If the reaction rate constants were left unchanged, the right hand side of the differential equations for buffering would have units of concentration per time, which is inconsistent. Multiplying the rate constants by compartment volume removes this inconsistency by changing the units of the right hand side to mass per time.

The last statement in the `KINETIC` block updates the value of `cai` from `ca[0]`. This is necessary because intracellular [Ca^{2+}] is known elsewhere in NEURON as `cai`, e.g. to other mechanisms and to NEURON's internal routine that computes E_{Ca}.

When developing a new mechanism or making substantive changes to an existing mechanism, it is generally advisable to check for consistency of units with `modlunit`. Given the dimensional complexity of this model, such testing is absolutely indispensable.

Usage

If this mechanism is inserted in a section, the concentrations of Ca^{2+} and the free and bound buffer in all compartments will be available through `hoc` as `ca_cadifus[]`, `Buffer_cadifus[]`, and `CaBuffer_cadifus[]`. These `STATE`s will also be available for plotting and analysis through the GUI.

The PARAMETERs DCa, k1buf, k2buf, and TotalBuffer will also be available for inspection and modification through both the graphical interface and hoc statements (with the _cadifus suffix). All PARAMETERs are GLOBALs by default, i.e. they will have the same values in each location where the cadifus mechanism has been inserted. Therefore in a sense it is gratuitous to declare in the NEURON block that TotalBuffer is GLOBAL. However, this declaration serves to remind the user of the nature of this important variable, which is likely to be changed during exploratory simulations or optimization.

In some cases it might be useful for one or more of the PARAMETERs to be RANGE variables. For example, TotalBuffer and even DCa or the buffer rate constants might not be uniform throughout the cell. To make TotalBuffer and DCa RANGE variables only requires replacing the line

```
GLOBAL vrat, TotalBuffer
```

in the NEURON block with

```
GLOBAL vrat
RANGE TotalBuffer, DCa
```

The GLOBAL volume factors vrat[] are available through hoc for inspection, but it is inadvisable to change their values because they would likely be inconsistent with the frat[] values and thereby cause errors in the simulation.

All occurrences of this mechanism will have the same number of shells, regardless of the physical diameter of the segments in which the mechanism has been inserted. With Nannuli = 4, the thickness of the outermost shell will be ≤ 1 μm in segments with diam

≤ 6 μm. If this spatial resolution is inadequate, or if the model has segments with larger diameters, then `Nannuli` may have to be increased. NMODL does not have dynamic arrays, so in order to change the number of shells one must recompile the mechanism after assigning a new value to `Nannuli` by editing the NMODL source code.

Example 9.9: a calcium pump

This mechanism involves a calcium pump based on the reaction scheme outlined in the description of the `KINETIC` block of **Example 9.7: kinetic scheme for a voltage-gated current**. It is a direct extension of the model of calcium diffusion with buffering in **Example 9.8: calcium diffusion with buffering**, the principal difference being that a calcium pump is present in the cell membrane. The following discussion focuses on the requisite changes in Listing 9.8, and the operation and use of this new mechanism. For all other details the reader should refer to **Example 9.8**.

The `NEURON` block

Changes in the `NEURON` block are marked in **bold**. The first nontrivial difference from the prior example is that this mechanism `READ`s the value of `cao`, which is used in the pump reaction scheme.

```
NEURON {
   SUFFIX cdp
   USEION ca READ cao, cai, ica WRITE cai, ica
   RANGE ica_pmp
   GLOBAL vrat, TotalBuffer, TotalPump
}
```


The mechanism also `WRITE`s a pump current that is attributed to `ica` so that its transmembrane Ca^{2+} flux will be factored into NEURON's calculations of $[Ca^{2+}]_i$. This current, which is a `RANGE` variable known as `ica_pmp_cdp` to the `hoc` interpreter, constitutes a net movement of positive charge across the cell membrane, and it follows the usual sign convention (outward current is "positive"). The pump current has a direct effect on membrane potential, which, because of the rapid activation of the pump, is manifest by a distinct delay of the spike peak and a slight increase of the postspike hyperpolarization. This mechanism could be made electrically "silent" by having it `WRITE` an equal but opposite `NONSPECIFIC_CURRENT` or perhaps a current that involves some other ionic species, e.g. Na^+, K^+, or Cl^-.

The variable `TotalPump` is the total density of pump sites on the cell membrane, whether free or occupied by Ca^{2+}. Making it `GLOBAL` means that it is user adjustable, and that the pump is assumed to have uniform density wherever the mechanism has been inserted. If local variation is required, this should be a `RANGE` variable.

The `UNITS` block

This mechanism includes the statement `(mol) = (1)` because the density of pump sites will be specified in units of `(mol/cm2)`. The term `mole` cannot be used here because it is already defined in NEURON's units database as $6.022169 \cdot 10^{23}$ (Avogadro's number).

Variable declaration blocks

The PARAMETER block

Five new statements have been added because this mechanism uses the rate constants of the pump reactions and the density of pump sites on the cell membrane.

```
k1 = 1      (/mM-ms)
k2 = 0.005  (/ms)
k3 = 1      (/ms)
k4 = 0.005  (/mM-ms)
: to eliminate pump, set TotalPump to 0 in hoc
TotalPump = 1e-11  (mol/cm2)
```

These particular rate constant values were chosen to satisfy two criteria: the pump influx and efflux should be equal at $[Ca^{2+}]$ = 50 nM, and the rate of transport should be slow enough to allow a slight delay in accelerated transport following an action potential that included a voltage-gated Ca^{2+} current. The density `TotalPump` is sufficient for the pump to have a marked damping effect on $[Ca^{2+}]_i$ transients; lower values reduce the ability of the pump to regulate $[Ca^{2+}]_i$.

The ASSIGNED block

These three additions have been made.

```
cao      (mM)
ica_pmp  (mA/cm2)
parea    (um)
```

This mechanism treats $[Ca^{2+}]_o$ as a constant. The pump current and the surface area over which the pump is distributed are also clearly necessary.

The CONSTANT block

Consistency of units requires explicit mention of an extracellular volume in the kinetic scheme for the pump.

```
CONSTANT { volo = 1e10  (um2) }
```

The value used here is equivalent to 1 liter of extracellular space per micron length of the cell, but the actual value is irrelevant to this mechanism because `cao` is treated as a constant. Since the value of `volo` is not important for this mechanism, there is no need for it to be accessible through `hoc` commands or the GUI, so it is not a `PARAMETER`. On the other hand, there is a sense in which it is an integral part of the pump mechanism, so it would not be appropriate to make `volo` be a `LOCAL` variable since `LOCAL`s are intended for temporary storage of "throwaway" values. Finally, the value of `volo` would never be changed in the course of a simulation. Therefore `volo` is declared in a `CONSTANT` block.

The STATE block

The densities of pump sites that are free or have bound Ca^{2+}, respectively, are represented by the two new `STATE`s

```
pump    (mol/cm2)
pumpca  (mol/cm2)
```

Equation definition blocks

The BREAKPOINT block

This block has one additional statement


```
BREAKPOINT {
  SOLVE state METHOD sparse
  ica = ica_pmp
}
```

The assignment `ica = ica_pmp` is needed to ensure that the pump current is reckoned in NEURON's calculation of $[Ca^{2+}]_i$.

The `INITIAL` block

The statement

```
parea = PI*diam
```

must be included to specify the area per unit length over which the pump is distributed.

If it is correct to assume that $[Ca^{2+}]_i$ has been equal to `cai0_ca_ion` (default = 50 nM) for a long time, the initial levels of `pump` and `pumpca` can be set by using the steady state formula

```
pump = TotalPump/(1 + (cai*k1/k2))
pumpca = TotalPump - pump
```

An alternative initialization strategy is to place

```
ica = 0
SOLVE state STEADYSTATE sparse
```

at the end of the `INITIAL` block, where the `ica = 0` statement is needed because the kinetic scheme treats transmembrane Ca^{2+} currents as a source of Ca^{2+} flux. This idiom makes NEURON compute the initial values of `STATE`s, which can be particularly convenient for mechanisms whose steady state solutions are difficult or impossible to express in analytical form. This would require adding a `CONSERVE` statement to the

`KINETIC` block to insure that the equations that describe the free and bound buffer are independent (see also **The `INITIAL` block** in **Example 9.7: kinetic scheme for a voltage-gated current**).

Both of these initializations explicitly assume that the net Ca^{2+} current generated by other sources equals 0, so the net pump current following initialization is also 0. If this assumption is incorrect, as is almost certainly the case if one or more voltage-gated Ca^{2+} currents are included in the model, then $[Ca^{2+}]_i$ will start to change immediately when a simulation is started. Most often this is not the desired outcome. The proper initialization of a model that contains mechanisms with complex interactions may involve performing an "initialization run" and using `SaveState` objects (see **Examples of custom initializations** in **Chapter 8**).

The `KINETIC` block

Changes in this block are marked in **bold**. For dimensional consistency, the pump scheme requires the new `COMPARTMENT` statements and units conversion factor `(1e10)`.

```
KINETIC state {
  COMPARTMENT i, diam*diam*vrat[i] {ca CaBuffer Buffer}
  COMPARTMENT (1e10)*parea {pump pumpca}
  COMPARTMENT volo {cao}
  LONGITUDINAL_DIFFUSION i, DCa*diam*diam*vrat[i] {ca}

  :pump
  ~ ca[0] + pump <-> pumpca  (k1*parea*(1e10), k2*parea*(1e10))
  ~ pumpca <-> pump + cao    (k3*parea*(1e10), k4*parea*(1e10))
  CONSERVE pump + pumpca = TotalPump * parea * (1e10)
  ica_pmp = 2*FARADAY*(f_flux - b_flux)/parea

  : all currents except pump
  ~ ca[0] << (-(ica - ica_pmp)*PI*diam/(2*FARADAY))
  FROM i=0 TO Nannuli-2 {
    ~ ca[i] <-> ca[i+1] (DCa*frat[i+1], DCa*frat[i+1])
  }
```



```
  dsq = diam*diam
  FROM i=0 TO Nannuli-1 {
    dsqvol = dsq*vrat[i]
    ~ ca[i] + Buffer[i] <-> CaBuffer[i] (k1buf*dsqvol, k2buf*dsqvol)
  }

  cai = ca[0]
}
```

The pump reaction statements implement the scheme outlined in the description of the `KINETIC` block of **Example 9.7: kinetic scheme for a voltage-gated current**. Also as described in that section, the `CONSERVE` statement ensures strict numerical conservation, which is helpful for convergence and accuracy.

In the steady state, the net forward flux in the first and second reactions must be equal. Even during physiologically relevant transients, these fluxes track each other effectively instantaneously. Therefore the transmembrane Ca^{2+} flux generated by the pump is taken to be the net forward flux in the second reaction.

This mechanism `WRITE`s `ica` in order to affect $[Ca^{2+}]_i$. The total transmembrane Ca^{2+} flux is the sum of the pump flux and the flux from all other sources. Thus to make sure that `ica_pmp` is not counted twice, it is subtracted from total Ca^{2+} current `ica` in the expression that relates Ca^{2+} current to Ca^{2+} flux.

Usage

The `STATE`s and `PARAMETER`s that are available through `hoc` and the GUI are directly analogous to those of the `cadifus` mechanism, but they will have the suffix `_cdp` rather than `_cadifus`. The additional pump variables `pump_cdp`, `pumpca_cdp`, `ica_pmp_cdp`, and `TotalPump_cdp` will also be available and are subject to similar

concerns and constraints as their counterparts in the diffusion reactions (see **Usage** in **Example 9.8: calcium diffusion with buffering**).

Models with discontinuities

The incorporation of variable time step integration methods in NEURON made it necessary to provide a way to ensure proper handling of abrupt changes in `PARAMETER`s, `ASSIGNED` variables, and `STATE`s. At first this was accomplished by adding `at_time()` and `state_discontinuity()` to NMODL, but the advent of NEURON's event delivery system has obviated the need for these functions and we strongly advise against using them in any new model development. Even so, they have been employed in several mechanisms of recent vintage, e.g. models of pulse generators and synaptic transmission, so the following discussion contains explanations of why they were used and what they do, as well as current recommendations for preferred ways to implement models that involve discontinuities.

Discontinuities in `PARAMETER`s and `ASSIGNED` variables

Before CVODE was added to NEURON, sudden changes in `PARAMETER`s and `ASSIGNED` variables, such as the sudden change in current injection during a current pulse, had been implicitly assumed to take place on a time step boundary. This is inadequate with variable time step methods because it is unlikely that a time step boundary will correspond to the onset or offset of the pulse. Worse, the time step may be longer than the pulse itself, which may thus be entirely ignored.

The `at_time()` function was added so that a model description could explicitly notify NEURON of the times at which any discontinuities occur. This function has no effect on fixed time step integration, but during variable time step integration, the statement `at_time(tevent)` guarantees that the integrator reduces the step size so that it completes at time $\texttt{tevent}_-$, which is on the order of roundoff error *before* `tevent`. The integrator then reinitializes at $\texttt{tevent}_+$, which is on the order of roundoff error *after* `tevent`, and the solution continues from there. This is how the built-in current clamp model `IClamp` notifies NEURON of the time of onset of the pulse and its offset (see the `BREAKPOINT` block of **Example 9.3: an intracellular stimulating electrode**). As noted above, however, now the preferred way to implement abrupt changes in `PARAMETER`s and `ASSIGNED` variables is to take advantage of NEURON's event delivery system (specifically, self-events) because of improved computational efficiency and greater conceptual clarity (see **Chapter 10**).

In the course of a variable time step simulation, a missing `at_time()` call may cause one of two symptoms. If a `PARAMETER` changes but returns to its original value within the same interval, the pulse may be entirely missed. More often, a single discontinuity will take place within a time step interval, causing the integrator to start what seems like a binary search for the location of the discontinuity in order to satisfy the error tolerance on the step; of course, this is very inefficient.

Discontinuities in STATEs

Some kinds of synaptic models involve a discontinuity in one or more STATE variables. For example, a synapse whose conductance follows the time course of an alpha function (for more detail about the alpha function itself see Rall (Rall 1977) and Jack et al. (Jack et al. 1983)) can be implemented as a kinetic scheme in the two state model

```
KINETIC state {
  ~ a <-> g (k, 0)
  ~ g -> (k)
}
```

("->" indicates a sink reaction), where a discrete synaptic event results in an abrupt increase of STATE a. This formulation has the attractive property that it can handle multiple streams of events with different weights, so that g will be the sum of the individual alpha functions with their appropriate onsets.

Abrupt changes in STATEs require particularly careful treatment because of the special nature of states in variable time step ODE solvers. Before the advent of an event delivery system in NEURON, this required not only an `at_time()` call to notify NEURON about the time of the discontinuity, but also a `state_discontinuity()` statement to specify how the affected STATE would change. Furthermore, `state_discontinuity()` could only be used in an `if (at_time()){}` block. Thus the BREAKPOINT block for a synaptic event that starts at `onset` and reaches a maximum conductance `gmax` would look like this


```
BREAKPOINT {
  if (at_time(onset)) {
    : scale factor exp(1) = 2.718... ensures
    : that peak conductance will be gmax
    state_discontinuity(a, a + gmax*exp(1))
  }
  SOLVE state METHOD sparse
  i = g*(v - e)
}
```

The first argument to `state_discontinuity()` is interpreted as a reference to the `STATE`, and the second argument is an expression for its new value. The first argument will be assigned the value of its second argument just *once* for any time step. This is important because, for several integration methods, `BREAKPOINT` assignment statements are often executed twice to calculate the *di/dv* terms of the Jacobian matrix.

This synaptic model works well with deterministic stimulus trains, but it is difficult for the user to supply the administrative `hoc` code for managing the `onset` and `gmax` variables to take advantage of the promise of "multiple streams of input events with different weights." The most important problem is how to save events that have significant delay between their generation and their handling at time `onset`. As this model stands, an event can be passed to it by assigning values to `onset` and `gmax` only after the previous onset event has been handled.

These complexities have been eliminated by the event delivery system. Instead of handling the state discontinuity in the `BREAKPOINT` block, the synaptic model should now be written in the form


```
BREAKPOINT {
  SOLVE state METHOD sparse
  i = g*(v - e)
}

NET_RECEIVE(weight (microsiemens)) {
  a = a + weight*exp(1)
}
```

in which event distribution is handled internally from a specification of network connectivity (see next section). Note that there is no need to use either `at_time()` or `state_discontinuity()`. Also, the `BREAKPOINT` block should not have any `if` statements. All discontinuities should be handled in a `NET_RECEIVE` block. For further details of how to deal with streams of synaptic events with arbitrary delays and weights, see **Chapter 10**.

> We should mention that early implementations of the event delivery system did require `state_discontinuity()`. Thus you may encounter a legacy synaptic model whose `NET_RECEIVE` block contains a statement such as `state_discontinuity(a, a+w*exp(1))`. This requirement no longer exists, and we discourage the use of this syntax.

Event handlers

With recent versions of NEURON, the most powerful and general strategy for dealing with discontinuities in `ASSIGNED` variables, `PARAMETER`s, and `STATE`s is to use the `NetCon` class's `event()` method, which exploits NEURON's event delivery system (see **Chapter 10**). The *`handler`*`()` procedure in *`netcon`*`.event(`*`te`*`, "`*`handler`*`()")` can contain statements that change anything discontinuously, as long as the last statement in *`handler`*`()` is `cvode.re_init()` (see **Chapter 8**).

Time-dependent PARAMETER changes

One way to translate the concept of a "smoothly varying" parameter into a computational implementation is by explicit specification in a model description, as in

```
BREAKPOINT { i = imax*sin(w*t) }
```

This works with both fixed and variable time step integration. Time-dependent changes can also be specified at the `hoc` interpreter level, but care is needed to ensure they are properly computed in the context of variable time steps. For instance, it might seem convenient to change `PARAMETER`s prior to `fadvance()` calls, e.g.

```
proc advance() {
  IClamp[0].amp = imax*sin(w*t)
  fadvance()
}
```

This does work with fixed `dt` but is discouraged because it produces inaccurate results with variable `dt` methods.

An alternative that works well with fixed and variable time step integration is to use the `Vector` class's `play()` method with linear interpolation, which became available in NEURON 5.4. This is invoked with

```
vec.play(&rangevar, tvec, 1)
```

in which *`vec`* and *`tvec`* are a pair of `Vector`s that define a piecewise linear function of time $y = f(t)$, i.e. *`tvec`* contains a monotonically increasing sequence of times, and *`vec`* holds the corresponding y values. The *`rangevar`* is the variable that is to be driven by $f()$. In the future, `Vector.play` will be extended to cubic spline interpolation and will allow "continuous" play of a smooth function defined by a `Vector`.

References

Durand, D. The somatic shunt cable model for neurons. *Biophys. J.* 46:645-653, 1984.

Frankenhaeuser, B. and Hodgkin, A.L. The after-effects of impulses in the giant nerve fibers of *Loligo*. *J. Physiol.* 131:341-376, 1956.

Hines, M.L. and Carnevale, N.T. Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Computation* 12:995-1007, 2000.

Jack, J.J.B., Noble, D., and Tsien, R.W. *Electric Current Flow in Excitable Cells*. London: Oxford University Press, 1983.

Johnston, D. and Wu, S.M.-S. *Foundations of Cellular Neurophysiology*. Cambridge, MA: MIT Press, 1995.

Kohn, M.C., Hines, M.L., Kootsey, J.M., and Feezor, M.D. A block organized model builder. *Mathematical and Computer Modelling* 19:75-97, 1994.

Kootsey, J.M., Kohn, M.C., Feezor, M.D., Mitchell, G.R., and Fletcher, P.R. SCoP: an interactive simulation control program for micro- and minicomputers. *Bulletin of Mathematical Biology* 48:427-441, 1986.

McCormick, D.A. Membrane properties and neurotransmitter actions. In: *The Synaptic Organization of the Brain*, edited by G.M. Shepherd. NY: Oxford University Press, 1998, p. 37-75.

Moczydlowski, E. and Latorre, R. Gating kinetics of Ca^{2+}-activated K^{+} channels from rat muscle incorporated into planar lipid bilayers. *Journal of General Physiology* 82:511-542, 1983.

Oran, E.S. and Boris, J.P. *Numerical Simulation of Reactive Flow*. New York: Elsevier, 1987.

Rall, W. Core conductor theory and cable properties of neurons. In: *Handbook of Physiology, vol. 1, part 1: The Nervous System*, edited by E.R. Kandel. Bethesda, MD: American Physiological Society, 1977, p. 39-98.

Staley, K.J., Otis, T.S., and Mody, I. Membrane properties of dentate gyrus granule cells: comparison of sharp microelectrode and whole-cell recordings. *J. Neurophysiol.* 67:1346-1358, 1992.

Wilson, M.A. and Bower, J.M. The simulation of large scale neural networks. In: *Methods in Neuronal Modeling*, edited by C. Koch and I. Segev. Cambridge, MA: MIT Press, 1989, p. 291-333.

Yamada, W.M., Koch, C., and Adams, P.R. Multiple channels and calcium dynamics. In: *Methods in Neuronal Modeling*, edited by C. Koch and I. Segev. Cambridge, MA: MIT Press, 1998, p. 137-170.

Chapter 9 Index

N

Chapter 10

Synaptic transmission and artificial spiking cells

In NEURON, a cell model is a set of differential equations. Network models consist of cell models and the connections between them. Some forms of communication between cells, e.g. graded synapses, gap junctions, and ephaptic interactions, require more or less complete representations of the underlying biophysical mechanisms. In these cases, coupling between cells is achieved by adding terms that refer to one cell's variables into equations that belong to a different cell. The first part of this chapter describes the `POINTER` syntax that makes this possible in NEURON.

The same approach can be used for detailed mechanistic models of spike-triggered transmission, which entails spike initiation and propagation to the presynaptic terminal, transmitter release, ligand-receptor interactions on the postsynaptic cell, and somatodendritic integration. However, it is far more efficient to use the widespread practice of treating spike propagation from the trigger zone to the synapse as a delayed logical event. The second part of this chapter tells how the `NetCon` (network connection) class supports this event-based style of communication.

In the last part of this chapter, we use event-based communication to simplify representation of the neurons themselves, creating highly efficient implementations of artificial spiking cells, e.g. integrate and fire "neurons." Artificial spiking cells are very

convenient sources of spike trains for driving synaptic mechanisms attached to biophysical neuron models. Networks that consist entirely of artificial spiking cells run hundreds of times faster than their biophysical counterparts, so they are particularly suitable for prototyping network models. They are also excellent tools in their own right for studying the functional consequences of network architectures and synaptic plasticity rules. In **Chapter 11** we demonstrate network models that involve various combinations of biophysical and artificial neuron models.

Modeling communication between cells

Experiments have demonstrated many kinds of interactions between neurons, but for most cells the principal avenues of communication are gap junctions and synapses. Gap junctions and synapses generate localized ionic currents, so in NEURON they are represented by point processes (see **Point processes** in **Chapter 5**, and **Example 9.2: a localized shunt** and **Example 9.3: an intracellular stimulating electrode** in **Chapter 9**).

The point processes used to represent gap junctions and synapses must produce a change at one location in the model that depends on information (membrane potential, calcium concentration, the occurrence of a spike) from some other location. This is in sharp contrast to the examples we discussed in **Chapter 9**, all of which are "local" in the sense that an instance of a mechanism at a particular

Models with `LONGITUDINAL_DIFFUSION` might also be considered "nonlocal," but their dependence on concentration in adjacent segments is handled automatically by the NMODL translator.

location on the cell depends only on the STATEs and PARAMETERs of that model *at that location*. They may also depend on voltage and ionic variables, but these also are *at that location* and automatically available to the model. To see how to do this, we will examine models of graded synaptic transmission, gap junctions, and spike-triggered synaptic transmission.

Example 10.1: graded synaptic transmission

A minimal conceptual model of graded synaptic transmission is that neurotransmitter is released continuously at a rate that depends on something in the presynaptic terminal, and that this causes some change in the postsynaptic cell. For the sake of discussion, let's say this something is $[Ca^{2+}]_{pre}$, the concentration of free calcium in the presynaptic terminal. We will also assume that the transmitter changes an ionic conductance in the postsynaptic cell.

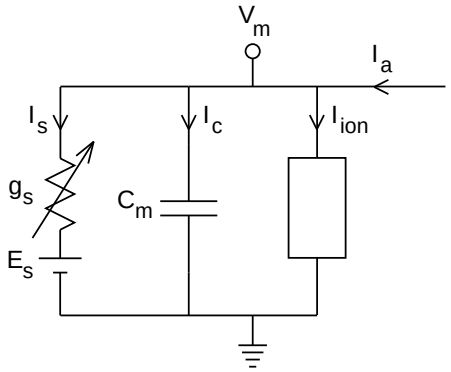


Figure 10.1. Membrane potential in the immediate neighborhood of a postsynaptic conductance depends on the synaptic current (I_s), the currents through the local membrane capacitance and ionic conductances (I_c and I_{ion}), and the axial current arriving from adjacent regions of the cell (I_a).

From the standpoint of the postsynaptic cell, a conductance-change synapse might look like Fig. 10.1, where g_s, E_s, and I_s are the synaptic conductance, equilibrium potential, and current, respectively. The effect of graded synaptic transmission on the postsynaptic cell is expressed in Equation 10.1.

$$C_m \frac{dV_m}{dt} + I_{ion} = I_a - (V_m - E_s) \cdot g_s([Ca^{2+}]_{pre}) \qquad \text{Eq. 10.1}$$

This is the charge balance equation for the electrical vicinity of the postsynaptic region. The terms on the left hand side are the usual local capacitive and ionic transmembrane currents. The first term on the right hand side is the current that enters the postsynaptic region from adjacent parts of the cell, which NEURON takes care of automatically. The second term on the right hand side expresses the effect of the ligand-gated channels. The current through these channels is the product of two factors. The first factor is merely the

local electrochemical gradient for ion flow. The second factor is a conductance term that depends on the calcium concentration at some other location.

We already know that a localized conductance is implemented in NEURON with a point process, and that such a mechanism is automatically able to access all the local variables that it needs (in this case, the local membrane potential and the synapse's equilibrium potential). But the calcium concentration in the presynaptic terminal is nonlocal, and that poses a problem; furthermore, it is likely to change with every `fadvance()`.

We could try inserting a `hoc` statement like this into the main computational loop

```
somedendrite.syn.capre = precell.bouton.cai(1)
```

At each time step, this would update the variable `capre` in the synaptic mechanism `syn` attached to the postsynaptic section `somedendrite`, making it equal to the free calcium concentration `cai` at the 1 end of the `bouton` section in the presynaptic cell `precell`. However, this statement would have to be reinterpreted at each `fadvance()`, which might slow down the simulation considerably.

If what happens to the postsynaptic cell depends on the moment-to-moment details of what is going on in the presynaptic terminal, it is far more efficient to use a `POINTER` variable (see Listing 10.1). In NMODL, a `POINTER` variable holds a reference to another variable. The specific reference is defined by a `hoc` statement, as we shall see below.

`POINTER` variables are not limited to point processes. Distributed mechanisms can also use `POINTER`s, although possibly for very different purposes.


```
: Graded synaptic transmission

NEURON {
  POINT_PROCESS GradSyn
  POINTER capre
  RANGE e, k, g, i
  NONSPECIFIC_CURRENT i
}

UNITS {
  (nA) = (nanoamp)
  (mV) = (millivolt)
  (uS) = (microsiemens)
  (molar) = (1/liter)
  (mM) = (millimolar)
}

PARAMETER {
  e = 0  (mV) : reversal potential
  k = 0.02  (uS/mM3)
}

ASSIGNED {
  v  (mV)
  capre (mM)  : presynaptic [Ca]
  g  (uS)
  i  (nA)
}

BREAKPOINT {
  g = k*capre^3
  i = g*(v - e)
}
```

Listing 10.1. `gradsyn.mod`

The `NEURON` block

The `POINTER` statement in the NEURON block declares that `capre` refers to some other variable that may belong to a noncontiguous segment, possibly even in a different section; below we show how to attach this to the free calcium concentration in a presynaptic terminal. The synaptic strength is not specified by a peak conductance, but in terms of a "transfer function scale factor" `k`, which has units of ($\mu S/mM^3$).

The BREAKPOINT block

The synaptic conductance g is proportional to the cube of capre and does not saturate. This is similar to the calcium dependence of synaptic conductance in a model described by De Schutter et al. (1993).

Usage

After creating a new instance of the GradSyn point process, we link its POINTER variable to the variable at some other location we want it to follow with hoc statements, e.g.

```
objref syn
somedendrite syn = new GradSyn(0.8)
setpointer syn.cp, precell.bouton.cai(0.5)
```

The second statement attaches an instance of the GradSyn mechanism, called syn, to somedendrite. The third statement uses setpointer to assert that the synaptic conductance of syn will be governed by cai in the middle of a section called bouton that is part of cell precell. Of course this assumes that the presynaptic section precell.bouton contains a calcium accumulation mechanism.

Figure 10.2 shows simulation results from a model of graded synaptic transmission. In this model, the presynaptic terminal precell is a 1 μm diameter hemisphere with voltage-gated calcium current cachan (cachan.mod in c:nrnxx\examples\nrniv\nmodl under MSWindows or nrn-x.x/share/examples/nrniv/nmodl under UNIX) and a calcium accumulation mechanism that includes diffusion, buffering, and a pump (cdp, discussed in **Example 9.9: a calcium pump**). The postsynaptic cell is a passive single

compartment with surface area 100 μm^2, C_m = 1 $\mu f/cm^2$, and τ_m = 30 ms. A `GradSyn` synapse with transfer function scale factor `k` = 0.2 $\mu S/mM^3$ is attached to the postsynaptic cell, and presynaptic membrane potential is driven between -70 and -30 mV by a sinusoid with a period of 400 ms. The time course of presynaptic $[Ca]_i$ and synaptic conductance show clipping of the negative phases of the sine wave; the postsynaptic membrane potential shows less clipping because of filtering by membrane capacitance.

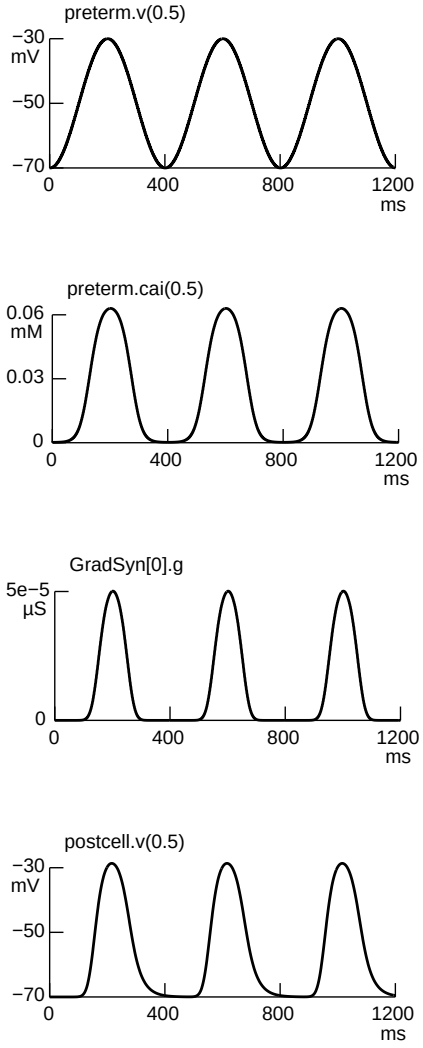


Figure 10.2. Graded synaptic transmission. Top two graphs: Presynaptic membrane potential `preterm.v` was "clamped" to -70-20cos(2πt/400) mV, producing a periodic increase of $[Ca]_i$ (`preterm.cai` is the concentration just inside the cell membrane) with clipping of the negative peaks. Bottom two graphs: The synaptic conductance GradSyn[0].g shows even more clipping of the negative phases of the sinusoid, but membrane capacitance smoothes the time course of postsynaptic membrane potential.

Example 10.2: a gap junction

The current that passes through a gap junction depends on the moment-to-moment fluctuations of voltage on both sides of the junction. This can be handled by a pair of point processes on the two sides that use `POINTER`s to monitor each other's voltage, as in

```
section1 gap1 = new Gap(x1)
section2 gap2 = new Gap(x2)
setpointer gap1.vpre, section2.v(x2)
setpointer gap2.vpre, section1.v(x1)
```

Conservation of charge requires the use of two point processes: one drains current from one side of the gap junction, and the other delivers an equal current to the other side.

Listing 10.2 presents the NMODL specification of a point process that can be used to implement ohmic gap junctions.

```
NEURON {
  POINT_PROCESS Gap
  POINTER vgap
  RANGE r, i
  NONSPECIFIC_CURRENT i
}

PARAMETER { r = 1e10 (megohm) }

ASSIGNED {
  v (millivolt)
  vgap (millivolt)
  i (nanoamp)
}

BREAKPOINT { i = (v - vgap)/r }
```

Listing 10.2. `gap.mod`

This implementation can cause spurious oscillations if the coupling between the two voltages is too tight (i.e. if the resistance `r` is too low) because it degrades the Jacobian matrix of the system equations. While it does introduce off-diagonal terms to couple the nodes on either side of the gap junction, it fails to add the conductance of the gap junction

to the terms on the main diagonal. The result is an *approximate* Jacobian, which makes numeric integration effectively a modified Euler method, instead of the fully implicit or Crank-Nicholson methods which are numerically more robust. Consequently, results are satisfactory only if coupling is loose (i.e. if `r` is large compared to the total conductance of the other ohmic paths connected to the affected nodes). If oscillations do occur, their amplitude can be reduced by decreasing `dt`, and they can be eliminated by using CVODE. In such cases, it may be preferable to implement gap junctions is with the `LinearMechanism` class (e.g. by using the LinearCircuitBuilder), which sets up the diagonal and off-diagonal terms of the Jacobian properly so that simulations are completely stable.

Usage

The following `hoc` code use this mechanism to set up a model of a gap junction between two cells. The `Gap` mechanisms allow current to flow between the internal node at the 1 end of `a` and the internal node at the 0 end of `b`.

```
create a,b
access a

forall {nseg=10 L=1000 diam=10 insert hh}

objref g[2]
for i=0,1 {
  g[i] = new Gap()
  g[i].r = 3
}

a g[0].loc(0.9999)  // just inside "distal" end of a
b g[1].loc(0.0001)  // just inside "proximal" end of b
setpointer g[0].vgap, b.v(0.0001)
setpointer g[1].vgap, a.v(0.9999)
```


Modeling spike-triggered synaptic transmission: an event-based strategy

Prior to NEURON 4.1, model descriptions of synaptic transmission could only use `POINTER` variables to obtain their presynaptic information. This required a detailed piecing together of individual components that was acceptable for models with only a few synapses. Models of larger networks required users to exert considerable administrative effort to create mechanisms that handle synaptic delay, exploit potentially great simulation efficiencies offered by simplified models of synapses, and maintain information about network connectivity.

The experience of NEURON users in creating special strategies for managing network simulations (e.g. (Destexhe et al. 1994a; Lytton 1996)) stimulated the development of NEURON's network connection (`NetCon`) class and event delivery system. Instances of the `NetCon` class manage the delivery of presynaptic "spike" events to synaptic point processes via the event delivery system. This works for all of NEURON's integrators, including the local variable time step method in which each cell is integrated with a time step appropriate to its own state changes. Model descriptions of synapses never need to queue events, and there is no need for heroic efforts to make them work properly with adaptive integration. These features offer enormous convenience to users who are interested in models that involve synaptic transmission at any level of complexity from single cell to large networks.

Conceptual model

In its most basic form, the physical system that we want to represent consists of a presynaptic neuron with a spike initiation zone that gives rise to an axon, which leads to a terminal that makes a synaptic connection onto a postsynaptic cell (Fig. 10.3). Our conceptual model of spike-triggered transmission is that arrival of a spike at the presynaptic terminal has some effect (e.g. a conductance change) in the postsynaptic cell that is described by a differential equation or kinetic scheme. Details of what goes on at the spike initiation zone are assumed to be unimportant--all that matters is whether a spike has, or has not, reached the presynaptic terminal. This conceptual model lets us take advantage of special features of NEURON that allow extremely efficient computation.

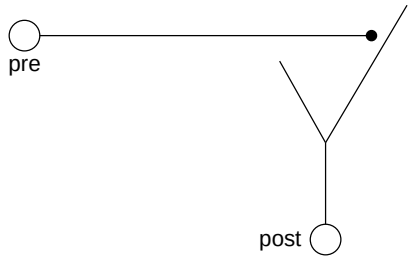


Figure 10.3. Cartoon of a synaptic connection (filled circle) between a presynaptic cell pre and a postsynaptic cell post.

A first approach to implementing a computational representation of our conceptual model might be something like the top of Fig. 10.4. We would monitor membrane potential at the presynaptic terminal for spikes (watch for threshold crossing). When a spike is detected, we wait for an appropriate delay (latency of transmitter release plus diffusion time) and then notify the synaptic mechanism that it's time to go into action. For this simple example, we have assumed that synaptic transmission simply causes a

conductance change in the postsynaptic cell. It is also possible to implement more complex mechanisms that include representations of processes in the presynaptic terminal (e.g. processes involved in use-dependent plasticity).

We can speed things up a lot by leaving out the axon and presynaptic terminal entirely, i.e. instead of computing the propagation of the action potential along the axon, just monitor the spike initiation zone. Once a spike occurs, we wait for a total delay equal to the sum of the conduction latency and the synaptic latency, and then activate the postsynaptic conductance change (Fig. 10.4 bottom).

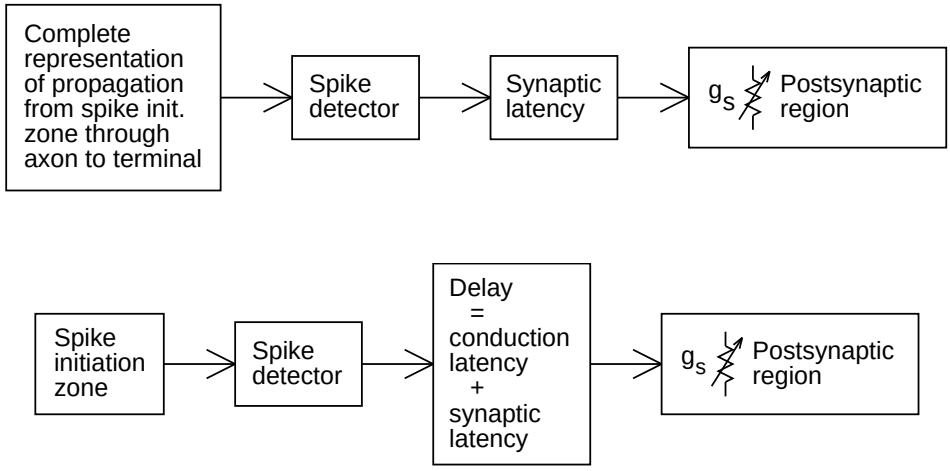


Figure 10.4. Computational implementation of a model of spike-triggered synaptic transmission. Top: The basic idea is that a presynaptic spike causes some change in the postsynaptic cell. Bottom: A more efficient version doesn't bother computing conduction in the presynaptic axon.

The `NetCon` class

Let's step back from this problem for a moment and think about the bottom diagram in Fig. 10.4. The "spike detector" and "delay" in the middle of this diagram are the seed

of an idea for a general strategy for dealing with synaptic connections. In fact, the `NetCon` object class is used to apply this strategy in defining the synaptic connection between a source and a target.

A `NetCon` object connects a presynaptic variable, such as voltage, to a target point process (here a synapse) with arbitrary delay and weight. If the presynaptic variable crosses `threshold` in a positive direction at time `t`, then at time `t+delay` a special `NET_RECEIVE` procedure in the target point process is called and receives the `weight` information. Each `NetCon` can have its own `threshold`, `delay`, and `weight`, i.e. these parameters are stream-specific. The only constraint on `delay` is that it be nonnegative. There is no limit on the number of events that can be "in the pipeline," and there is no loss of events under any circumstances. Events always arrive at the target at the interval `delay` after the time they were generated.

When you create a `NetCon` object, at a minimum you must specify the source variable and the target. The source variable is generally the membrane potential of the currently accessed section, as shown here. The target is a point process that contains a `NET_RECEIVE` block (see Listing 10.3 below).

```
section netcon = new NetCon(&v(x), target, thresh, del, wt)
```

Threshold, delay, and weight are optional; their defaults are shown here, and they can be specified after the `NetCon` object has been constructed.

```
netcon.threshold = 10 // mV
netcon.delay = 1      // ms
netcon.weight = 0     // uS
```


The weight associated with a `NetCon` object is actually the first element of a weight vector. The number of elements in the weight vector depends on the number of arguments in the `NET_RECEIVE` statement of the NMODL source code that defines the point process. We will return to this in **Example 10.5: use-dependent synaptic plasticity** and **Example 10.6: saturating synapses**.

NEURON's event-based approach to implementing communication between cells reduces the computational burden of network simulations tremendously, because it supports efficient, unlimited divergence and convergence (fan-out and fan-in). To understand why, first consider divergence. What if a presynaptic cell projects to multiple postsynaptic targets (Fig. 10.5 top)? Easy enough--just add a `NetCon` object for each target (Fig. 10.5 bottom). This is computationally efficient because threshold detection is done on a "per source" basis, rather than a "per `NetCon`" basis. That is, if multiple `NetCon`s have the same source with the same `threshold`, they all share a single threshold detector. The source variable is checked only once per time step and, when it crosses `threshold` in the positive direction, events are generated for each connecting `NetCon` object. Each of these `NetCon`s can have its own weight and delay, and the target mechanisms can belong to different classes.

Now consider convergence. Suppose a neuron receives multiple inputs that are anatomically close to each other and of the same type (Fig. 10.6 top). In other words, we're assuming that each synapse has its postsynaptic action through the same kind of mechanism (i.e. it has identical kinetics, and (in the case of conductance-change synapses) the same equilibrium potential). We can represent this by connecting multiple `NetCon` objects to the same postsynaptic point process (Fig. 10.6 bottom). This yields

large efficiency improvements because a single set of synaptic equations can be shared by many input streams (one input stream per connecting `NetCon` instance). Of course, these synapses can have different strengths and latencies, because each `NetCon` object has its own weight and delay.

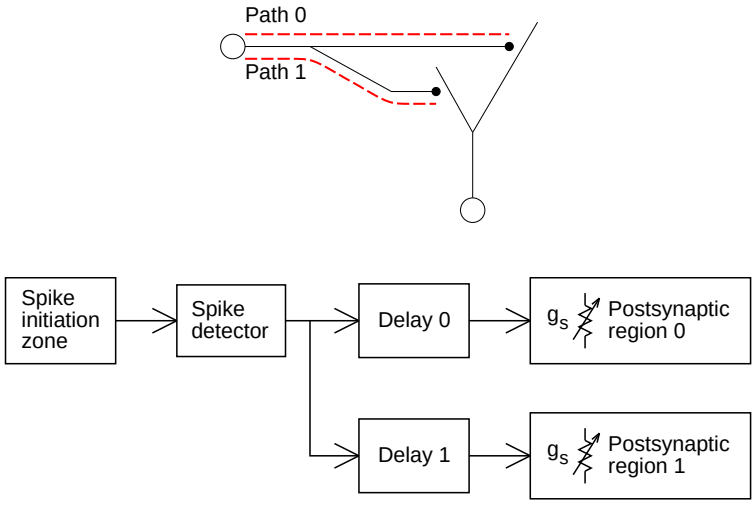


Figure 10.5. Efficient divergence. Top: A single presynaptic neuron projects to two different target synapses. Bottom: Computational model of this circuit uses multiple `NetCon`s with a single threshold detector that monitors a common source.

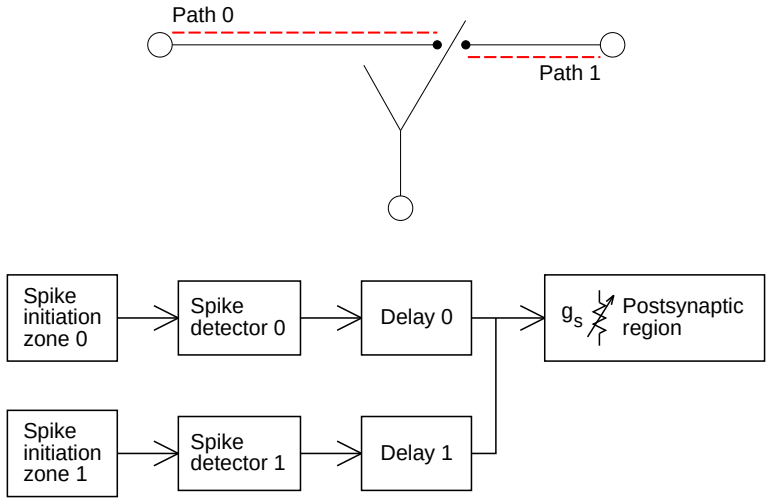


Figure 10.6. Efficient convergence. Top: Two different presynaptic cells make synaptic connections of the same class that are electrically close to each other. Bottom: Computational model of this circuit uses multiple `NetCon`s that share a single postsynaptic mechanism (single equation handles multiple input streams).

Having seen the rationale for using events to implement models of synaptic transmission, we are ready to examine some point processes that include a `NET_RECEIVE` block and can be used as synaptic mechanisms in network models.

Example 10.3: synapse with exponential decay

Many kinds of synapses produce a synaptic conductance that increases rapidly and then declines gradually with first order kinetics, e.g. AMPAergic excitatory synapses. This can be modeled by an abrupt change of conductance, which is triggered by arrival of an event, and then decays with a single time constant.

The NMODL code that implements such a mechanism is shown in Listing 10.3. This mechanism is similar to NEURON's built in `ExpSyn`. Calling it `ExpSyn1` allows us to test and modify it without conflicting with NEURON's built-in `ExpSyn`.

The synaptic conductance of this mechanism summates not only when events arrive from a single presynaptic source, but also when they arrive from different places (multiple input streams). This mechanism handles both situations by defining a single conductance state *g* which is governed by a differential equation whose solution is

$g(t) = g(t_0)\, e^{(t-t_0)/\tau}$, where $g(t_0)$ is the conductance at the time of the most recent event.

```
: expsyn1.mod

NEURON {
  POINT_PROCESS ExpSyn1
  RANGE tau, e, i
  NONSPECIFIC_CURRENT i
}

PARAMETER {
  tau = 0.1  (ms)
  e   = 0    (millivolt)
}

ASSIGNED {
  v  (millivolt)
  i  (nanoamp)
}

STATE { g (microsiemens) }

INITIAL { g = 0 }

BREAKPOINT {
  SOLVE state METHOD cnexp
  i = g*(v - e)
}
```



```
DERIVATIVE state { g' = -g/tau }

NET_RECEIVE(weight (microsiemens)) {
  g = g + weight
}
```

Listing 10.3. `expsyn1.mod`

The `BREAKPOINT` block

The `BREAKPOINT` block of this mechanism is its main computational block. This contains the `SOLVE` statement that tells how states will be integrated. The `cnexp` method is used because the kinetics of `ExpSyn1` are described by a differential equation of the form $y' = f(y)$, where $f(y)$ is linear in y (see also **The `DERIVATIVE` block** in **Example 9.4: a voltage-gated current** in **Chapter 9**). The `BREAKPOINT` block ends with an assignment statement that sets the value of the synaptic current.

The `DERIVATIVE` block

The `DERIVATIVE` block contains the differential equation that describes the time course of the synaptic conductance `g`: a first order decay with time constant `tau`.

The `NET_RECEIVE` block

The `NET_RECEIVE` block contains the code that specifies what happens in response to presynaptic activation. This is called by the `NetCon` event delivery system when an event arrives at this point process.

So suppose we have a model with an `ExpSyn1` point process that is the target of a `NetCon`. Imagine that the `NetCon` detects a presynaptic spike at time `t`. What happens next?

ExpSyn1's conductance g continues to follow a smooth exponential decay with time constant tau until time t+delay, where delay is the delay associated with the NetCon object. At this point, an event is delivered to the ExpSyn1. Just before entry to the NET_RECEIVE block, NEURON makes all STATEs, v, and values assigned in the BREAKPOINT block consistent at t+delay. Then the code in the NET_RECEIVE block is executed, making the synaptic conductance suddenly jump up by the amount specified by the NetCon's weight.

> As we mentioned in **Chapter 9**, earlier versions of NEURON had to change g with a state_discontinuity() statement. This is no longer necessary.

Usage

Suppose we wanted to set up a synaptic connection between two cells using an ExpSyn1 mechanism, as in Fig. 10.7.

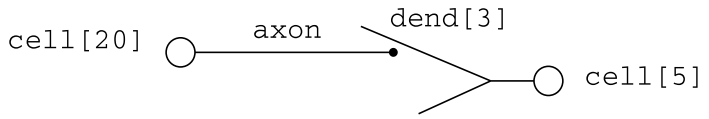


Figure 10.7. Schematic of a synaptic connection between two cells.

This could be done with the following hoc code, which also illustrates the use of a List of NetCon objects as a means for keeping track of the synaptic connections in a network.

```
// keep connectivity in a list of NetCon objects
objref ncl
ncl = new List()

// attach an ExpSyn1 point process called syn
//   to the 0.3 location on dend[3] of cell[5]
objref syn
cell[5].dend[3] syn = new ExpSyn1(0.3)
```



```
// presynaptic v is cell[20].axon.v(1)
// connect this to syn via a new NetCon object
//   and add the NetCon to the list ncl
cell[20].axon ncl.append(new NetCon(&v(1), \
         syn, threshold, delay, weight)
```

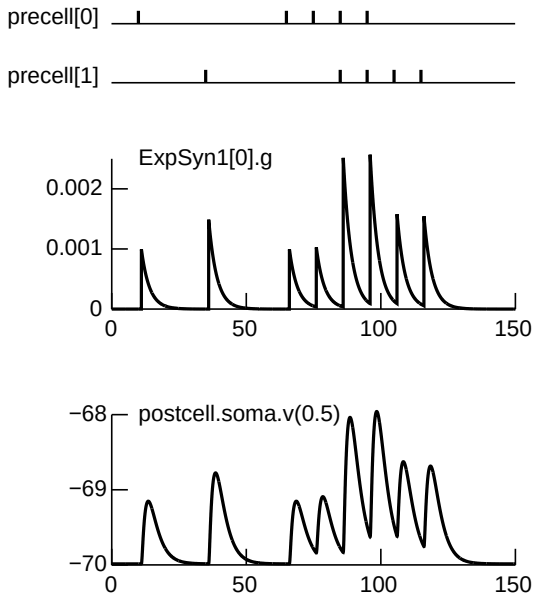


Figure 10.8. Simulation results from the model shown in Fig. 10.6. Note stream-specific synaptic weights and temporal summation of synaptic conductance and membrane potential.

Figure 10.8 shows results of a simulation of two input streams that converge onto a single `ExpSyn1` attached to a postsynaptic cell, as in the diagram at the top of Fig. 10.6. The presynaptic firing times are indicated by the rasters labeled `precell[0]` and `precell[1]`. The synaptic conductance and postsynaptic membrane potential (middle and bottom graphs) display stream-specific synaptic weights, and also show temporal summation of inputs within an individual stream and between inputs on multiple streams.

Example 10.4: alpha function synapse

With a few small changes, we can extend `ExpSyn1` to implement an alpha function synapse. We only need to replace the differential equation with the two state kinetic scheme

```
STATE { a (microsiemens) g (microsiemens) }
KINETIC state {
  ~ a <-> g (1/tau, 0)
  ~ g -> (1/tau)
}
```

and change the `NET_RECEIVE` block to

```
NET_RECEIVE(weight (microsiemens)) {
  a = a + weight*exp(1)
}
```

The factor `exp(1)` = *e* is included so that an isolated event produces a peak conductance of magnitude `weight`, which occurs at time `tau` after the event. Since this mechanism involves a `KINETIC` block instead of a `DERIVATIVE` block, we must also change the integration method specified by the `SOLVE` statement from `cnexp` to `sparse`.

The extra computational complexity of using a kinetic scheme is offset by the fact that, no matter how many `NetCon` streams connect to this model, the computation time required to integrate `STATE g` remains constant. Some increase of efficiency can be gained by recasting the kinetic scheme as two linear differential equations

```
DERIVATIVE state {
..a' = -a/tau1
..b' = -b/tau
..g = b - a
}
```

which are solved by the `cnexp` method (this is what NEURON's built in `Exp2Syn` mechanism does). As `tau1` approaches `tau`, `g` approaches an alpha function (although

the factor by which `weight` must be multiplied approaches infinity; see `factor` in the next example). Also, there are now two state discontinuities in the `NET_RECEIVE` block

```
NET_RECEIVE(weight (microsiemens)) {
  a = a + weight*factor
  b = b + weight*factor
}
```

Example 10.5: use-dependent synaptic plasticity

Here the alpha function synapse is extended to implement a form of use-dependent synaptic plasticity. Each presynaptic event initiates two distinct processes: direct activation of ligand-gated channels, which causes a transient conductance change, and activation of a mechanism that in turn modulates the conductance change produced by successive synaptic activations. In this example we presume that modulation depends on the postsynaptic increase of a second messenger, which we will call "G protein" for illustrative purposes. We must point out that this example is entirely hypothetical, and that it is quite different from models described by others (Destexhe and Sejnowski 1995) in which the G protein itself gates the ionic channels.

For this mechanism it is essential to distinguish each stream into the generalized synapse, since each stream has to maintain its own [G] (concentration of activated G protein). That is, streams are independent of each other in terms of the effect on [G], but their effects on synaptic conductance show linear superposition.

```
: gsyn.mod

NEURON {
  POINT_PROCESS GSyn
  RANGE tau1, tau2, e, i
  RANGE Gtau1, Gtau2, Ginc
  NONSPECIFIC_CURRENT i
  RANGE g
}
```



```
UNITS {
  (nA)   = (nanoamp)
  (mV)   = (millivolt)
  (umho) = (micromho)
}

PARAMETER {
  tau1   = 1  (ms)
  tau2   = 1.05    (ms)
  Gtau1  = 20   (ms)
  Gtau2  = 21   (ms)
  Ginc   = 1
  e      = 0    (mV)
}

ASSIGNED {
  v  (mV)
  i  (nA)
  g  (umho)
  factor
  Gfactor
}

STATE {
  A  (umho)
  B  (umho)
}

INITIAL {
  LOCAL tp
  A = 0
  B = 0
  tp = (tau1*tau2)/(tau2 - tau1) * log(tau2/tau1)
  factor = -exp(-tp/tau1) + exp(-tp/tau2)
  factor = 1/factor
  tp = (Gtau1*Gtau2)/(Gtau2 - Gtau1) * log(Gtau2/Gtau1)
  Gfactor = -exp(-tp/Gtau1) + exp(-tp/Gtau2)
  Gfactor = 1/Gfactor
}

BREAKPOINT {
  SOLVE state METHOD cnexp
  g = B - A
  i = g*(v - e)
}

DERIVATIVE state {
  A' = -A/tau1
  B' = -B/tau2
}
```



```
NET_RECEIVE(weight (umho), w, G1, G2, t0 (ms)) {
  G1 = G1*exp(-(t-t0)/Gtau1)
  G2 = G2*exp(-(t-t0)/Gtau2)
  G1 = G1 + Ginc*Gfactor
  G2 = G2 + Ginc*Gfactor
  t0 = t

  w = weight*(1 + G2 - G1)
  A = A + w*factor
  B = B + w*factor
}
```

Listing 10.4. `gsyn.mod`

The `NET_RECEIVE` block

The conductance of the ligand-gated ion channel uses the differential equation approximation for an alpha function synapse. The peak synaptic conductance depends on the value of [G] at the moment of synaptic activation. A similar, albeit much slower, alpha function approximation describes the time course of [G]. These processes peak approximately `tau1` and `Gtau1` after delivery of an event, respectively.

The peak synaptic conductance elicited by an individual event is specified in the `NET_RECEIVE` block, where `w = weight*(1+G2-G1)` describes how the effective weight of the synapse is modified by [G]. Even though conductance is integrated, [G] is needed only at discrete event times so it can be computed analytically from the elapsed time since the prior synaptic activation. The `INITIAL` block sets up the factors that are needed to make the peak changes equal to the values of `w` and `Ginc`.

Note that `G1` and `G2` are not `STATE`s in this mechanism. They are not even variables in this mechanism, but instead are "owned" by the particular `NetCon` instance that delivered the event. Each `NetCon` object instance keeps an array (the weight vector) whose size equals the number of arguments to `NET_RECEIVE`, and the arguments to

NET_RECEIVE are really references to the elements of this array. Unlike the arguments to a PROCEDURE or FUNCTION block, which are "call by value," the arguments to a NET_RECEIVE block are "call by reference." Therefore assignment statements in gsyn.mod's NET_RECEIVE block can change the values of variables that belong to the NetCon object, and this means that the NetCon's weight vector can be used to hold stream-specific state information. In the context of this particular example, each connection has its own [G], so gsyn uses "stream-specific plasticity" to represent "synapse-specific plasticity."

> On initialization, all elements of the weight vector other than the first one are automatically set to 0. However, a NET_RECEIVE block may have its own INITIAL block, and this can contain statements that assign nonzero values to NetCon "states." Such an INITIAL block is executed when finitialize() is called.

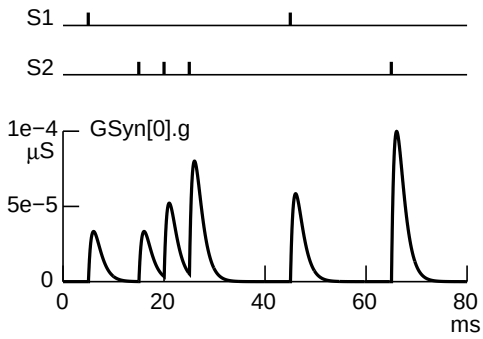


Figure 10.9. Simulation results from the model shown in Fig. 10.6 when the synaptic mechanism is GSyn. Note stream-specific use-dependent plasticity.

To illustrate the operation of this mechanism, imagine the network of Fig. 10.6 with a single GSyn driven by the two spike trains shown in Fig. 10.9. This emulates two synapses that are electrotonically close to each other, but with separate pools of [G]. The train with spikes at 5 and 45 ms (S1) shows some potentiation of the second conductance

transient, but the train that starts at 15 ms with a 200 Hz burst of three spikes displays a large initial potentiation that is even larger when tested after a 40 ms interval.

Example 10.6: saturating synapses

Several authors (e.g. (Destexhe et al. 1994a; Lytton 1996)) have used synaptic transmission mechanisms based on a simple conceptual model of transmitter-receptor interaction:

$$C + T \underset{\beta}{\overset{\alpha}{\rightleftarrows}} O \qquad \text{Eq. 10.2}$$

where transmitter *T* binds to a closed receptor channel *C* to produce an open channel *O*. In this conceptual model, spike-triggered transmitter release produces a transmitter concentration in the synaptic cleft that is approximated by a rectangular pulse with a fixed duration and magnitude (Fig. 10.10). A "large excess of transmitter" is assumed, so that while transmitter is present (the "onset" state, "ligand binding to channel") the postsynaptic conductance increases toward a maximum value with a single time constant $1/(\alpha T + \beta)$. After the end of the transmitter pulse (the "offset" state, "ligand·channel complex dissociating"), the conductance decays with time constant $1/\beta$. Further details of saturating mechanisms are covered by (Destexhe et al. 1994a and b) and (Lytton 1996).

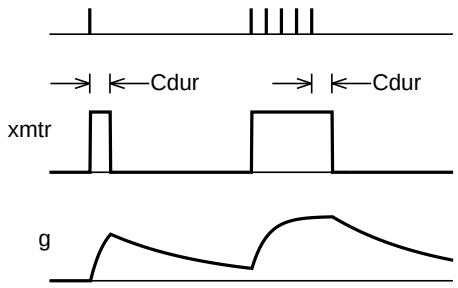


Figure 10.10. A saturating synapse model. A single presynaptic spike (top trace) causes a pulse of transmitter in the synaptic cleft with fixed duration (Cdur) and concentration (middle trace). This elicits a rapid increase of postsynaptic conductance followed by a slower decay (bottom trace). A high frequency burst of spikes produces a sustained elevation of transmitter that persists until Cdur after the last spike and causes saturation of the postsynaptic conductance.

There is an ambiguity when one or more spikes arrive on a single stream during the onset state triggered by an earlier spike: should the mechanism ignore the "extra" spikes, concatenate onset states to make the transmitter pulse longer without increasing its concentration, or increase (summate) the transmitter concentration? Summation of transmitter requires the onset time constant to vary with transmitter concentration. This places transmitter summation outside the scope of the Destexhe/Lytton model, which assumes a fixed time constant for the onset state. We resolve this ambiguity by choosing concatenation, so that repetitive impulses on one stream produce a saturating conductance change (Fig. 10.10). However, conductance changes elicited by separate streams will summate.

A model of the form used in Examples 10.4 and 10.5 can capture the idea of saturation, but the separate onset/offset formulation requires keeping track of how much

"material" is in the onset or offset state. The mechanism in Listing 10.5 implements an effective strategy for doing this. A noteworthy feature of this model is that the event delivery system serves as more than a conduit for receiving inputs from other cells: discrete events are used to govern the duration of synaptic activation, and are thus an integral part of the mechanism itself.

```
: ampa.mod

NEURON {
  POINT_PROCESS AMPA_S
  RANGE g
  NONSPECIFIC_CURRENT i
  GLOBAL Cdur, Alpha, Beta, Erev, Rinf, Rtau
}

UNITS {
  (nA)   = (nanoamp)
  (mV)   = (millivolt)
  (umho) = (micromho)
}

PARAMETER {
  Cdur  = 1.0   (ms)  : transmitter duration (rising phase)
  Alpha = 1.1   (/ms) : forward (binding) rate
  Beta  = 0.19  (/ms) : backward (dissociation) rate
  Erev  = 0     (mV)  : equilibrium potential
}

ASSIGNED {
  v     (mV)   : postsynaptic voltage
  i     (nA)   : current = g*(v - Erev)
  g     (umho) : conductance
  Rtau  (ms)   : time constant of channel binding
  Rinf  : fraction of open channels if xmtr is present "forever"
  synon : sum of weights of all synapses in the "onset" state
}

STATE { Ron Roff }  : initialized to 0 by default
: Ron and Roff are the total conductances of all synapses
:   that are in the "onset" (transmitter pulse ON)
:   and "offset" (transmitter pulse OFF) states, respectively

INITIAL {
  synon = 0
  Rtau = 1 / (Alpha + Beta)
  Rinf = Alpha / (Alpha + Beta)
}
```



```
BREAKPOINT {
  SOLVE release METHOD cnexp
  g = (Ron + Roff)*1(umho)
  i = g*(v - Erev)
}

DERIVATIVE release {
  Ron' = (synon*Rinf - Ron)/Rtau
  Roff' = -Beta*Roff
}

NET_RECEIVE(weight, on, r0, t0 (ms)) {
  : flag is an implicit argument of NET_RECEIVE, normally 0
  if (flag == 0) {
    : a spike arrived, start onset state if not already on
    if (!on) {
      : this synapse joins the set of synapses in onset state
      synon = synon + weight
      r0 = r0*exp(-Beta*(t - t0)) : r0 at start of onset state
      Ron = Ron + r0
      Roff = Roff - r0
      t0 = t
      on = 1
      : come again in Cdur with flag = 1
      net_send(Cdur, 1)
    } else {
      : already in onset state, so move offset time
      net_move(t + Cdur)
    }
  }
  if (flag == 1) {
    : "turn off transmitter"
    : i.e. this synapse enters the offset state
    synon = synon - weight
    : r0 at start of offset state
    r0 = weight*Rinf + (r0 - weight*Rinf)*exp(-(t - t0)/Rtau)
    Ron = Ron - r0
    Roff = Roff + r0
    t0 = t
    on = 0
  }
}
```

Listing 10.5. `ampa.mod`

The PARAMETER block

The actual value of the transmitter concentration in the synaptic cleft during the onset state is unimportant to this model, as long as it remains constant. To simplify the

mechanism, we assume transmitter concentration to be dimensionless, with a numeric value of 1. This allows us to specify the forward rate constant `Alpha` in units of 1/ms.

The `STATE` block

This mechanism has two `STATE`s. `Ron` is the total conductance of all synapses that are in the onset state, and `Roff` is the total conductance of all synapses that are in the offset state. These are declared without units, so a units factor will have to be applied elsewhere (in this example, this is done in the `BREAKPOINT` block).

The `INITIAL` block

At the start of a simulation, we assume that all channels are closed and no transmitter is present at any synapse. The initial values of `Ron`, `Roff`, and `synon` must therefore be 0. This initialization happens automatically for `STATE`s and does not require explicit specification in the `INITIAL` block, but `synon` needs an assignment statement.

The `INITIAL` block also calculates `Rtau` and `Rinf`. `Rtau` is the time constant for equilibration of the closed (free) and open (ligand-bound) forms of the postsynaptic receptors when transmitter is present in the synaptic cleft. `Rinf` is the open channel fraction if transmitter is present forever.

The `BREAKPOINT` and `DERIVATIVE` blocks

The total conductance is numerically equal to `Ron+Roff`. The `*1(umho)` factor is included for dimensional consistency.

The DERIVATIVE block specifies the first order differential equations that govern these STATEs. The meaning of each term in

```
Roff' = -Beta*Roff
```

is obvious, and in

```
Ron' = (synon*Rinf - Ron)/Rtau
```

the product synon*Rinf is the value that Ron approaches with increasing time.

The NET_RECEIVE block

The NET_RECEIVE block performs the task of switching each synapse between its onset and offset states. In broad outline, if an external event (an event generated by the NetCon's source passing threshold) arrives at time t to start an onset, the NET_RECEIVE block generates an event that it sends to itself. This self-event will be delivered at time t+Cdur, where Cdur is the duration of the transmitter pulse. Arrival of the self-event is the signal to switch the synapse back to the offset state. If another external event arrives from the same NetCon before the self-event does, the self-event is moved to a new time that is Cdur in the future. Thus resetting to the offset state can happen only if an interval of Cdur passes without new external events arriving.

"External event" and "input event" are synonyms. We will use the former term as clarity dictates when contrasting them with self-events.

To accomplish this strategy, the NET_RECEIVE block must distinguish an external event from a self-event. It does this by exploiting the fact that every event has an implicit argument called flag, the value of which is automatically 0 for an external event.

Handling of external events

The event flag is "call by value," unlike the explicit arguments that are declared inside the parentheses of the `NET_RECEIVE()` statement, which are "call by reference."

Arrival of an external event causes execution of the statements inside the `if (flag==0){}` clause. These begin with `if (!on)`, which tests whether this synapse should switch to the onset state.

Switching to the onset state involves keeping track of how much "material" is in the onset and offset states. This requires moving the synapse's channels into the pool of channels that are exposed to transmitter, which simply means adding the synapse's `weight` to `synon`. Also, the conductance of this synapse, which had been decaying with rate constant `1/Beta`, must now start to grow with rate constant `Rtau`. This is done by computing `r0`, the synaptic conductance at the present time `t`, and then adding `r0` to `Ron` and subtracting it from `Roff`. Next the value of `t0` is updated for future use, and `on` is set to 1 to signify that the synapse is in the onset state. The last statement inside `if (!on){}` is `net_send(Cdur,nspike)`, which generates a self-event with delay given by the first argument and flag value given by the second argument. All the explicit arguments of this self-event will have the values of this particular `NetCon`, so when this self-event returns we will know how much "material" to switch from the onset to the offset state.

The `else {}` clause takes care of what happens if another external event arrives while the synapse is still in the onset state. The `net_move(t+Cdur)` statement moves the self-event to a new time that is `Cdur` in the future (relative to the arrival time of the new external event). In other words, this prolongs synaptic activation until `Cdur` after the most recent external event.

Handling of self-events

When the self-event is finally delivered, it triggers an offset. We know it is a self-event because its `flag` is 1. Once again we keep track of how much "material" is in the onset and offset states, but now we subtract the synapse's `weight` from `synon` to remove the synapse's channels from the pool of channels that are exposed to transmitter. Likewise, the conductance of this synapse, which was growing with rate constant `Rtau`, must now begin to decay with rate constant `1/Beta`. Finally, the value of `t0` is updated and `on` is reset to 0.

Artificial spiking cells

NEURON's event delivery system was created with the primary aim of making it easier to represent synaptic connections between biophysical model neurons. However, the event delivery system turns out to be quite useful for implementing a wide range of mechanisms that require actions to be taken after a delay. The saturating synapse model presented above is just one example of this.

The previous section also showed how spike-triggered synaptic transmission makes extensive use of the network connection class to define connections between cells. The typical `NetCon` object watches a source cell for the occurrence of a spike, and then, after some delay, delivers a weighted event to a target synaptic mechanism, i.e. it is a metaphor for axonal spike propagation. More generally, a `NetCon` object can be regarded as a channel on which a stream of events generated at a source is transmitted to a target. The target can be a point process, a distributed mechanism, or an artificial neuron (e.g. an

integrate and fire model). The effect of events on a target is specified in NMODL by statements in a `NET_RECEIVE` block, which is called only when an event has been delivered.

The event delivery system also opens up a large domain of simulations in which certain types of artificial spiking cells, and networks of them, can be simulated hundreds of times faster than with numerical integration methods. Discrete event simulation is possible when all the state variables of a model cell can be computed analytically from a new set of initial conditions. That is, if an event occurs at time t_1, all state variables must be computable from the state values and time t_0 of the previous event. Since computations are performed only when an event occurs, total computation time is proportional to the number of events delivered and independent of the number of cells, number of connections, or problem time. Thus handling 100,000 spikes in one hour for 100 cells takes the same time as handling 100,000 spikes in 1 second for 1 cell.

Artificial spiking cells are implemented in NEURON as point processes, but unlike ordinary point processes, they can serve as targets and sources for `NetCon` objects. They can be targets because they have a `NET_RECEIVE` block, which specifies how incoming events from one or more `NetCon` objects are handled, and details the calculations necessary to generate outgoing events. They can also be sources because the same `NET_RECEIVE` block generates discrete output events which are delivered through one or more `NetCon` objects to targets.

The following examples analyze the three broad classes of integrate and fire cells that are built into NEURON. In order to emphasize how the event delivery system is used to

implement the dynamics of these mechanisms, we have omitted many details from the NMODL listings. Ellipses indicate elisions, and listings include *italicized pseudocode* where necessary for clarity. Complete source code for all three of these cell classes is provided with NEURON.

Example 10.7: `IntFire1`, a basic integrate and fire model

The simplest integrate and fire mechanism built into NEURON is `IntFire1`, which has a membrane state variable *m* (analogous to membrane potential) which decays toward 0 with time constant τ.

$$\tau \frac{dm}{dt} + m = 0 \qquad \text{Eq. 10.3}$$

An input event of weight *w* adds instantaneously to *m*, and if *m* reaches or exceeds the threshold value of 1, the cell "fires," producing an output event and returning *m* to 0. Negative weights are inhibitory while positive weights are excitatory. This is analogous to a cell with a membrane time constant τ that is very long compared to the time course of individual synaptic conductance changes. Every synaptic input to such a cell shifts membrane potential to a new level in a time that is much shorter than τ, and each cell firing erases all traces of prior inputs. Listing 10.6 presents an initial implementation of `IntFire1`.

```
NEURON {
  ARTIFICIAL_CELL IntFire1
  RANGE tau, m
}

PARAMETER { tau = 10 (ms) }
```



```
ASSIGNED {
  m
  t0 (ms)
}

INITIAL {
  m = 0
  t0 = 0
}

NET_RECEIVE (w) {
  m = m*exp(-(t - t0)/tau)
  m = m + w
  t0 = t
  if (m > 1) {
    net_event(t)
    m = 0
  }
}
```

Listing 10.6. A basic implementation of `IntFire1`.

The `NEURON` block

As the introduction to this section mentions, artificial spiking cells are implemented in NEURON as point processes. The keyword `ARTIFICIAL_CELL` is in fact a synonym for `POINT_PROCESS`, but we use it as a deliberate reminder to ourselves that this model has a `NET_RECEIVE` block, lacks a `BREAKPOINT` block, and does not have to be associated with a section location or numerical integrator. Unlike other point processes, an artificial cell is isolated from the usual things that link mechanisms to each other: it does not refer to membrane potential `v` or any ions, and it does not use `POINTER` variables. Instead, the "outside" can affect it only by sending it discrete events, and it can only affect the "outside" by sending discrete events.

The NET_RECEIVE block

The mechanisms we have seen so far use BREAKPOINT and KINETIC or DERIVATIVE blocks to specify the calculations that are performed during a time step dt, but an artificial cell model does not have these blocks. Instead, calculations only take place when a new event arrives, and these are performed in the NET_RECEIVE block.

When a NetCon delivers a new event to an IntFire1 cell, the present value of m is computed analytically and then m is incremented by the weight w of the event. According to the NET_RECEIVE block, the present value of m is found by applying an exponential decay to the value it had immediately after the previous event; therefore the code contains variable t0 which keeps track of the last event time.

If an input event drives m to or above threshold, the net_event(t) statement notifies all NetCons, for which this point process is the source, that it fired a spike at time t (the argument to net_event() can be any time at or later than the current time t). Then the cell resets m to 0. The code in Listing 10.6 imposes no limit on firing frequency--if a NetCon with delay of 0 and a weight of 1.1 has such an artificial cell as both its source and target, the system will behave "properly," in the sense that events will be generated and delivered without time ever advancing. It is easy to prevent the occurrence of such a runaway stream of events (see ***Adding a refractory period*** below).

There is no threshold test overhead at every dt because IntFire1 has no variable for NetCons to watch. That is, this artificial spiking cell does not need the usual test for local membrane potential v to cross NetCon.threshold, which is essential at every time step for event generation with biophysical neuron models. Furthermore the event

delivery system only places the earliest event to be delivered on the event queue. When that time finally arrives, all targets whose `NetCon`s have the same source and delay get the event delivery, and longer delay streams are put back on the event queue to await their specific delivery time.

Enhancements to the basic mechanism

Visualizing the membrane state variable

The membrane state variable *m* is difficult to plot in an understandable manner, since it is represented in the computer by a variable `m` that remains unchanged over the interval between input events regardless of how many numerical integration steps were performed in that interval. Consequently `m` always has the value that was calculated after the last event was received, and plots of it look like a staircase (Fig. 10.11 left), with no apparent decay or indication of what the value of *m* was just before the event.

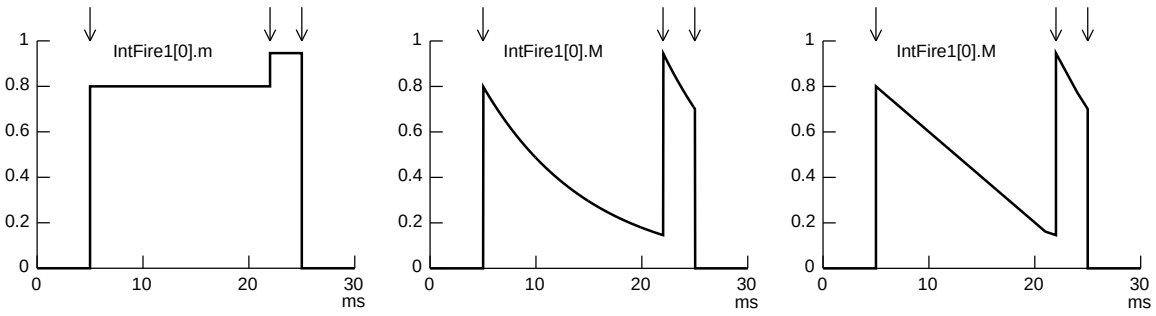


Figure 10.11. Response of an `IntFire1` cell with τ = 10 ms to input events with weight = 0.8 arriving at *t* = 5, 22, and 25 ms (arrows). The third input initiates a "spike." Left: The variable `m` is evaluated only when a new event arrives, so its plot looks like a staircase. A function can be included in `IntFire1`'s mod file (see text) to better indicate the time course of the membrane state variable *m*. Center: Plotting this function during a simulation with fixed `dt` (0.025 ms here) demonstrates the decay of *m* between events. Right: In a variable time step simulation, *m* appears to follow a sequence of linear ramps. This artifact is a consequence of the efficiency of adaptive integration, which computed analytical solutions at only a few instants, so the Graph tool could only draw lines from instant to instant.

This can be partially repaired by adding a function

```
FUNCTION M() {
  M = m*exp(-(t - t0)/tau)
}
```

that returns the present value of the membrane state variable *m*. This gives nice trajectories when fixed time step integration is used (Fig. 10.11 center). However, the natural step with the variable step method is the interspike interval itself, unless intervening events occur in other cells (e.g. 1 ms before the second input event in Fig. 10.11 right). At least the integration step function `fadvance()` returns 10^{-9} ms before and after the event to properly indicate the discontinuity in `M`.

Adding a refractory period

It is easy to add a relative refractory period by initializing `m` to a negative value after the cell fires (alternatively, a depolarizing afterpotential can be emulated by initializing `m` to a value in the range (0,1)). However, incorporating an absolute refractory period requires self-events.

Suppose we want to limit the maximum firing rate to 200 spikes per second, which corresponds to an absolute refractory period of 5 ms. To specify the duration of the refractory period, we use a variable named `refrac`, which is declared and assigned a value of 5 ms in the `PARAMETER` block. Adding the statement `RANGE refrac` to the `NEURON` block allows us to adjust this parameter from the interpreter and graphical interface. We also use a variable to keep track of whether the point process is in the refractory period or not. The name we choose for this variable is the eponymous `refractory`, and it is declared in the `ASSIGNED` block and initialized to a value of 0 ("false") in the `INITIAL` block.

The `NET_RECEIVE` implementation is then

```
NET_RECEIVE (w) {
  if (refractory == 0) {
    m = m*exp(-(t - t0)/tau)
    m = m + w
    t0 = t
    if (m > 1) {
      net_event(t)
      refractory = 1
      net_send(refrac, refractory)
    }
  } else if (flag == 1) {
    : self-event arrived, so terminate refractory period
    refractory = 0
    m = 0
    t0 = t
  } : else ignore the external event
}
```


If `refractory` equals 0, the cell accepts external events (i.e. events delivered by a `NetCon`) and calculates the state variable `m` and whether to fire the cell. When the cell fires a spike, `refractory` is set to 1 and further external events are ignored until the end of the refractory period (Fig. 10.12).

Recall from the saturating synapse example that the `flag` variable that accompanies an external event is 0. If this mechanism receives an event with a nonzero `flag`, it must be a self-event, i.e. an event generated by acall to `net_send()` when the cell fired. The `net_send(interval, flag)` statement places an event into the delivery system as an "echo" of the current event, i.e. it will come back to the sender after the specified `interval` with the specified `flag`. In this case we aren't interested in the weight but only the `flag`. Arrival of this self-event means that the refractory period is over.

The top of Fig. 10.12 shows the response of this model to a train of input stimuli. Temporal summation triggers a spike on the fourth input. The fifth input arrives during the refractory interval and has no effect.

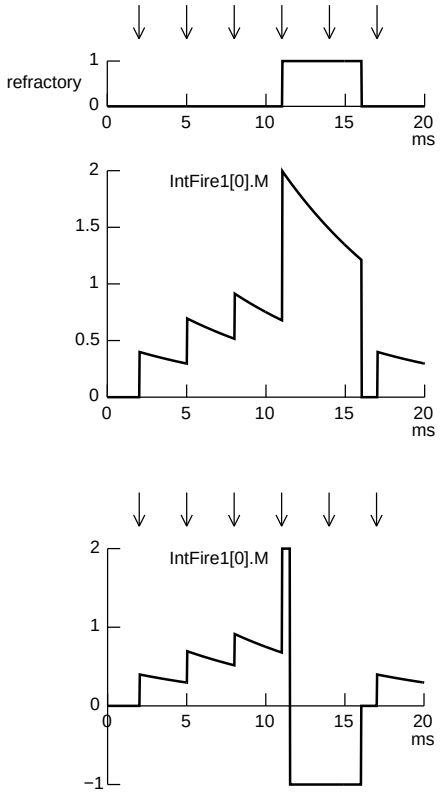


Figure 10.12. Response of an `IntFire1` cell with a 5 ms refractory interval to a run of inputs at 3 ms intervals (arrows), each with weight = 0.4. Top: The cell accepts inputs when `refractory` == 0. The fourth input (at 11 ms) drives the cell above threshold. This triggers an output event, increases `refractory` to 1 (top trace), and function `M`, which reflects the membrane state variable *m*, jumps to 2. During the 5 ms refractory period, `M` decays gradually, but the cell is unresponsive to further inputs (note that the input at 14 ms produces no change in the membrane state variable). At 16 ms `refractory` falls to 0, making the cell once again responsive to inputs, and `M` also returns to 0 until the next external event arrives. Bottom: After modifying the function `M` to generate rectangular pulses that emulate a spike followed by postspike hyperpolarization.

Improved presentation of the membrane state variable

The performance in the top of Fig. 10.12 is satisfactory, but the model could be further improved by one relatively minor change. As it stands the `M` function shows an exponential decay during the refractory period, which is at best distracting and irrelevant to the operation of the model, and potentially misleading at worst. It would be better for `M` to follow a stereotyped time course, e.g. a brief positive pulse followed by a longer negative pulse. This would not be confused with the subthreshold operation of the model, and it might be more suggestive of an action potential.

The most direct way to do this is to make `M` take different actions depending on whether or not the model is "spiking." One possibility is

```
FUNCTION M() {
  if (refractory == 0) {
    M = m*exp(-(t - t0)/tau)
  } else if (refractory == 1) {
    if (t - t0 < 0.5) {
      M = 2
    } else {
      M = -1
    }
  }
}
```

which is exactly what the built-in `IntFire1` model does. The bottom of Fig. 10.12 shows the time course of this revised function.

This demonstrates how visualization of cell operation can be enhanced by simple calculations of patterns for the spiking and refractory trajectories, with no overhead for cells that are not plotted. We must emphasize that the simulation calculations are analytic and performed only at event arrival, regardless of the refinements we introduced for the purpose of esthetics.

Sending an event to oneself to trigger deferred computation involves very little overhead, yet it allows elaborate calculations to be performed much more efficiently than if they were executed on a per `dt` basis. Self-events are heavily exploited in the implementation of `IntFire2` and `IntFire4`, which both offer greater kinetic complexity than `IntFire1`.

Example 10.8: `IntFire2`, firing rate proportional to input

The `IntFire2` model, like `IntFire1`, has a membrane state variable *m* that follows first order kinetics with time constant τ_m. However, an input event to `IntFire2` does not affect *m* directly. Instead it produces a discontinuous change in a synaptic current state variable *i*. Between events, *i* decays with its own time constant τ_s toward a steady "bias" value specified by the parameter i_b. That is,

$$\tau_s \frac{di}{dt} + i = i_b \qquad \text{Eq. 10.4}$$

where an input event causes *i* to change abruptly by *w* (Fig. 10.13 top). This current *i* drives *m*, i.e.

$$\tau_m \frac{dm}{dt} + m = i \qquad \text{Eq. 10.5}$$

where $\tau_m < \tau_s$. Thus an input event produces a gradual change in *m* that is described by two time constants and approximates an alpha function if $\tau_m \approx \tau_s$. When *m* crosses a threshold of 1 in a positive direction, the cell fires, *m* is reset to 0, and integration

resumes immediately, as shown in the bottom of Fig. 10.13. Note that *i* is not reset to 0, i.e. unlike `IntFire1`, firing of an `IntFire2` cell does not obliterate all traces of prior synaptic activation.

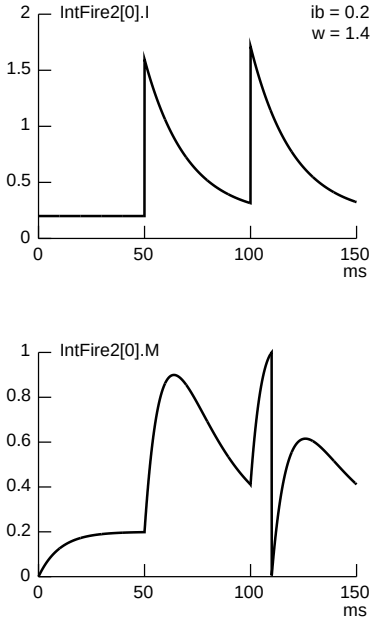


Figure 10.13. Top: Time course of synaptic current *i* in an `IntFire2` cell with τ_s = 20 ms and τ_m = 10 ms. This cell has bias current i_b = 0.2 and receives inputs with weight *w* = 1.4 at *t* = 50 and 100 ms. Bottom: The membrane state variable *m* of this cell is initially 0 and approaches the value of i_b (0.2 in this example) with time constant τ_m. The first synaptic input produces a subthreshold response, but temporal summation drives *m* above threshold at *t* = 109.94 ms. This resets *m* to 0 and integration resumes.

Depending on its parameters, `IntFire2` can emulate a wide range of relationships between input pattern and firing rate. Its firing rate is ~ i / τ_m if i is >> 1 and changes slowly compared to τ_m.

The parameter i_b is analogous to the combined effect of a baseline level of synaptic drive plus a bias current injected through an electrode. The requirement that $\tau_m < \tau_s$ is equivalent to asserting that the membrane time constant is faster than the decay of the current produced by an individual synaptic activation. This is plausible for slow inhibitory inputs, but where fast excitatory inputs are concerned an alternative interpretation can be applied: each input event signals an abrupt increase (followed by an exponential decline) in the mean firing rate of one or more afferents that produce brief but temporally overlapping postsynaptic currents. The resulting change of i is the moving average of these currents.

The `IntFire2` mechanism is amenable to discrete event simulation because Eqns. 10.4 and 10.5 have analytic solutions. If the last input event was at time t_0 and the values of i and m immediately after that event were $i(t_0)$ and $m(t_0)$, then their subsequent time course is given by

$$i(t) = i_b + [i(t_0) - i_b]\, \mathrm{e}^{-(t-t_0)/\tau_s} \qquad \text{Eq. 10.6}$$

and

$$m(t) = i_b + \left[i(t_0) - i_b \right] \frac{\tau_s}{\tau_s - \tau_m} \, e^{-(t-t_0)/\tau_s} + \left\{ m(t_0) - i_b - \left[i(t_0) - i_b \right] \frac{\tau_s}{\tau_s - \tau_m} \right\} e^{-(t-t_0)/\tau_m} \qquad \text{Eq. 10.7}$$

Implementation in NMODL

The core of the NMODL implementation of `IntFire2` is the function `firetime()`, which is discussed below. This function projects when *m* will equal 1 based on the present values of i_b, *i*, and *m*, assuming that *no new input events arrive*. The value returned by `firetime()` is 10^9 if the cell will never fire with no additional input. Note that if $i_b > 1$ the cell fires spontaneously even if no input events occur.

```
INITIAL {
   . . .
  net_send(firetime(args), 1)
}

NET_RECEIVE (w) {
   . . .
  if (flag == 1) { : time to fire
    net_event(t)
    m = 0
     . . .
    net_send(firetime(args), 1)
  } else {
     . . .
    update m
    if (m >= 1) {
      net_move(t) : the time to fire is now
    } else {
       . . .
      net_move(firetime(args) + t)
    }
  }
  update t0 and i
}
```

Listing 10.7. Key excerpts from `intfire2.mod`

The `INITIAL` block in `IntFire2` calls `firetime()` and uses the returned value to put a self-event into the delivery system. The strategy, which is spelled out in the `NET_RECEIVE` block, is to respond to external events by moving the delivery time of the self-event back and forth with the `net_move()` function. When the self-event is finally delivered (potentially never), `net_event()` is called to signal that this cell is firing. Notice that external events always have an effect on the value of *i*, and are never ignored--and shouldn't be, even if we introduced a refractory period in which we refused to integrate *m*.

The function `firetime()` returns the first $t \geq 0$ for which

$$a + b\, \mathrm{e}^{-t/\tau_s} + (c - a - b)\, \mathrm{e}^{-t/\tau_m} = 1 \qquad \text{Eq. 10.8}$$

where the parameters *a*, *b* and *c* are defined by the coefficients in Eq. 10.7. If there is no such *t* the function returns 10^9. This represents the time of the next cell firing, relative to the time t_0 of the most recent synaptic event.

Since `firetime()` must be executed on every input event, it is important to minimize the number of Newton iterations needed to calculate the next firing time. For this we use a strategy that depends on the behavior of the function

$$f_1(x) = a + b\, x^r + (c - a - b)\, x \qquad \text{Eq. 10.9a}$$

$$\text{where } \begin{matrix} x = \mathrm{e}^{-t/\tau_m} \\ r = \tau_m / \tau_s \end{matrix} \qquad \text{Eq. 10.9b}$$

over the domain $0 < x \leq 1$. Note that $c < 1$ is the value of f_1 at $x = 0$ (i.e. at $t = \infty$). The function f_1 is either linear in x (if $b = 0$) or convex up ($b > 0$) or down ($b < 0$) with no inflection points. Since $r < 1$, f_1 is tangent to the y axis for any nonzero b (i.e. $f_1'(0)$ is infinite).

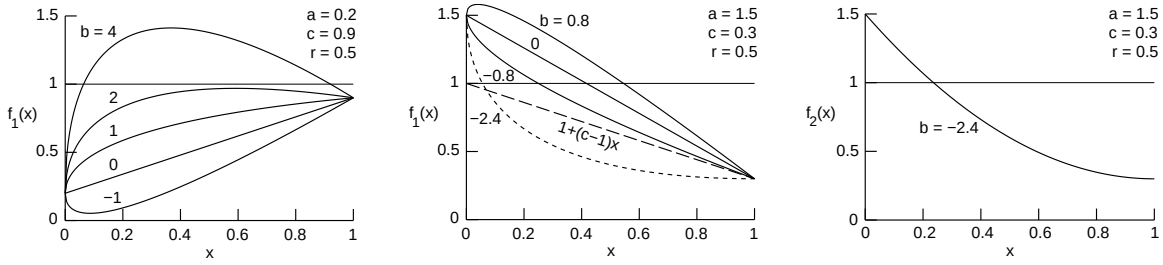


Figure 10.14. Plots of f_1 and f_2 computed for $r = 0.5$. See text for details.

The left panel of Fig. 10.14 illustrates the qualitative behavior of f_1 for $a \leq 1$. It is easy to analytically compute the maximum in order to determine if there is a solution to $f_1(x) = 1$. If a solution exists, f_1 will be concave downward so Newton iterations starting at $x = 1$ will underestimate the firing time.

For $a > 1$, a solution is guaranteed (Fig. 10.14 middle). However, starting Newton iterations at $x = 1$ is inappropriate if the slope there is more negative than $c - 1$ (straight dashed line in Fig. 10.14 middle). In that case, the transformation $x = \mathrm{e}^{-t/\tau_s}$ is used, giving the function

$$f_2(x) = a + b\,x + (c - a - b)\,x^{1/r} \qquad \text{Eq. 10.9c}$$

and the Newton iterations begin at $x = 0$ (Fig. 10.14 right).

These computations are performed over regions in which f_1 and f_2 are relatively linear, so the `firetime()` function usually requires only two or three Newton iterations to converge to the next firing time. The only exception is when f_1 has a maximum that is just slightly larger than 1, in which case it may be a good idea to stop after a couple of iterations and issue a self-event. The advantage of this would be the deferral of a costly series of iterations, allowing an interval in which another external event might arrive that would force computation of a new projected firing time. Such an event, whether excitatory or inhibitory, would likely make it easier to compute the next firing time.

Example 10.9: `IntFire4`, different synaptic time constants

`IntFire2` can emulate an input-output relationship with more complex dynamics than `IntFire1` does, but it is somewhat restricted because its response to every external event, whether excitatory or inhibitory, has the same kinetics. As we pointed out in the discussion of `IntFire2`, it is possible to interpret excitatory events in a way that partially sidesteps this issue. However, experimentally observed synaptic excitation tends to be faster than inhibition (e.g. (Destexhe et al. 1998)) so a more flexible integrate and fire mechanism is needed.

The `IntFire4` mechanism addresses this need. Its dynamics are specified by four time constants: τ_e for a fast excitatory current, τ_{i_1} and τ_{i_2} for a slower inhibitory current, and τ_m for the even slower leaky "membrane" which integrates these currents. When the

membrane state variable *m* reaches 1, the cell "fires," producing an output event and returning *m* to 0. This does not affect the other states of the model.

The differential equations that govern `IntFire4` are

$$\frac{de}{dt} = -k_e e \qquad \text{Eq. 10.10}$$

$$\frac{di_1}{dt} = -k_{i_1} i_1 \qquad \text{Eq. 10.11}$$

$$\frac{di_2}{dt} = -k_{i_2} i_2 + a_{i_1} i_1 \qquad \text{Eq. 10.12}$$

$$\frac{dm}{dt} = -k_m m + a_e e + a_{i_2} i_2 \qquad \text{Eq. 10.13}$$

where each *k* is a rate constant that equals the reciprocal of the corresponding time constant, and it is assumed that $k_e > k_{i_1} > k_{i_2} > k_m$ (i.e. $\tau_e < \tau_{i_1} < \tau_{i_2} < \tau_m$). An input event with weight $w > 0$ (i.e. an excitatory event) adds instantaneously to the excitatory current *e*. Equations 10.11 and 12, which define the inhibitory current i_2, are based on the reaction scheme

$$i_1 \xrightarrow{k_{i_1}} i_2 \xrightarrow{k_{i_2}} bath \qquad \text{Eq. 10.14}$$

in which an input event with weight $w < 0$ (i.e. an inhibitory event) adds instantaneously to i_1. The constants a_e, a_{i_1}, and a_{i_2} are chosen to normalize the response of the states *e*, i_1, i_2, and *m* to input events (Fig. 10.15). Therefore an input with weight $w_e > 0$ (an

"excitatory" input) produces a peak e of w_e and a maximum "membrane potential" m of w_e. Likewise, an input with weight $w_i < 0$ (an "inhibitory" input) produces an inhibitory current i_2 with a minimum of w_i and drives m to a minimum of w_i. Details of the analytic solution to these equations are presented in **Appendix A1: Mathematical analysis of IntFire4**.

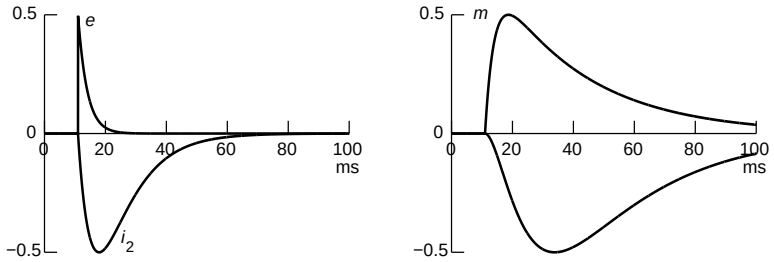


Figure 10.15. Left: Current generated by a single input event with weight 0.5 (e) or -0.5 (i_2). Right: The corresponding response of m. Parameters were $\tau_e = 3$, $\tau_{i_1} = 5$, $\tau_{i_2} = 10$, and $\tau_m = 30$ ms.

`IntFire4`, like `IntFire2`, finds the next firing time through successive approximation. However, `IntFire2` generally iterates to convergence every time an input event is received, whereas `IntFire4`'s algorithm implement a series of deferred Newton iterations by exploiting the downward convexity of the membrane potential trajectory and using NEURON's event delivery system. The result is an alternating sequence of self-events and single Newton iterations that converges to the correct firing time, yet remains computationally efficient in the face of heavy input event traffic.

This is illustrated in Fig. 10.16. If an event arrives at time t_0, values of $e(t_0)$, $i_1(t_0)$, $i_2(t_0)$, and $m(t_0)$ are calculated analytically. Should $m(t_0)$ be subthreshold, the self-event is moved to a new approximate firing time t_f that is based on the slope approximation to m

$$t_f = t_0 + (1 - m(t_0)) / m'(t_0) \text{ if } m'(t_0) > 0 \qquad \text{Eq. 10.15}$$

or

$$\infty \text{ if } m'(t_0) \leq 0$$

(Fig. 10.16 left and middle). If instead $m(t_0)$ reaches threshold, the cell "fires" so that `net_event()` is called (producing an output event that is picked up by all `NetCon`s for which this cell is a source) and m is reset to 0. The self-event is then moved to an approximate firing time that is computed from Eq. 10.15 using the values assigned to m and m' immediately after the "spike" (Fig. 10.16 right).

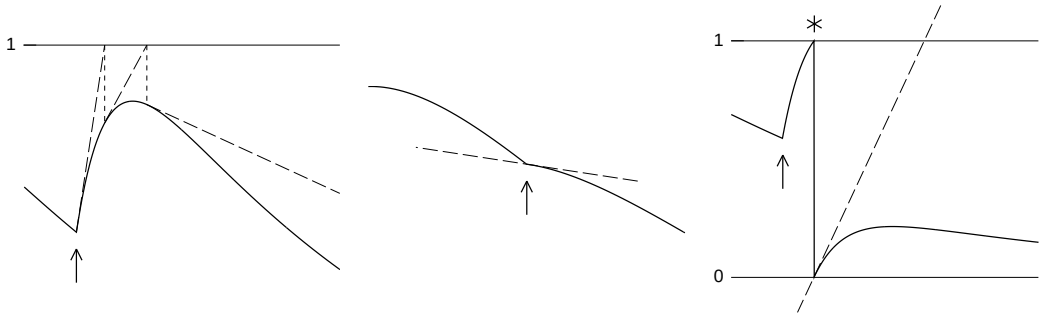


Figure 10.16. Excerpts from simulations of `IntFire4` cells showing time course of *m*. Arrival of an event (arrow = external event, vertical dotted line = self-event) triggers a Newton iteration. Slanted dashed lines are slope approximations to *m* immediately after an event. Left: Although Eq. 10.15 yields a finite t_f, this input is too weak for the cell to fire. Middle: Here $m' < 0$ immediately after an input event, so both t_f and the true firing time are infinite. Right: The slope approximation following the excitatory input is not shown, but it obviously crosses threshold before the actual firing time (asterisk). Following the "spike" *m* is reset to 0 but bounces back up because of persistent excitatory current. This dies away without eliciting a second spike, even though t_f is finite (dashed line).

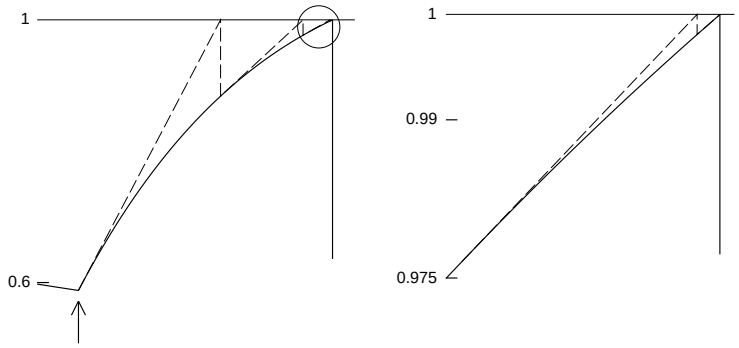


Figure 10.17. These magnified views of the trajectory from the right panel of Fig. 10.16 indicate how rapidly the event-driven Newton iterations converge to the next firing time. In this simulation, spike threshold was reached in four iterations after the excitatory input (arrow). The first two iterations are evident in the left panel, and additional magnification of the circled region reveals the last two iterations (right panel).

The justification for this approach stems from several considerations. The first of these is that t_f is never later than the true firing time. This assertion, which we prove in **Appendix A1**, is of central importance because the simulation would otherwise be in error.

Another consideration is that successive approximations must converge rapidly to the true firing time, in order to avoid the overhead of a large number of self-events. Using the slope approximation to m is equivalent to the Newton method for solving $m(t) = 1$, so convergence is slow only when the maximum value of m is close to 1. The code in `IntFire4` guards against missing "real" firings when m is asymptotic to 1, because it actually tests for `m > 1 - eps`, where the default value of `eps` is 10^{-6}. This

convergence tolerance `eps` is a user-settable `GLOBAL` parameter, so one can easily augment or override this protection.

Finally, the use of a series of self-events is superior to carrying out a complete Newton method solution because it is most likely that external events will arrive in the interval between firing times. Each external event would invalidate the previous computation of firing time and force a recalculation. This might be acceptable for the `IntFire2` mechanism with its efficient convergence, but the complicated dynamics of `IntFire4` suggest that the cost would be too high. How many iterations should be carried out per self-event is an experimental question, since the self-event overhead depends partly on the number of outstanding events in the event queue.

Other comments regarding artificial spiking cells

NEURON's event delivery system has been used to create many more kinds of artificial spiking neurons than the three classes that we have just examined. Specific examples include pacemakers, bursting cells, models with various forms of use-dependent synaptic plasticity, continuous or quantal stochastic variation of synaptic weight, and an "IntFire3" with a bias current and time constants $\tau_m > \tau_i > \tau_e$.

References

De Schutter, E., Angstadt, J.D., and Calabrese, R.L. A model of graded synaptic transmission for use in dynamic network simulations. *J. Neurophysiol.* 69:1225-1235, 1993.

Destexhe, A., Mainen, Z.F., and Sejnowski, T.J. An efficient method for computing synaptic conductances based on a kinetic model of receptor binding. *Neural Computation* 6:14-18, 1994a.

Destexhe, A., Mainen, Z.F., and Sejnowski, T.J. Synthesis of models for excitable membranes, synaptic transmission, and neuromodulation using a common kinetic formalism. *J. Comput. Neurosci.* 1:195-231, 1994b.

Destexhe, A., Mainen, Z.F., and Sejnowski, T.J. Kinetic models of synaptic transmission. In: *Methods in Neuronal Modeling*, edited by C. Koch and I. Segev. Cambridge, MA: MIT Press, 1998, p. 1-25.

Destexhe, A. and Sejnowski, T.J. G-protein activation kinetics and spillover of γ-aminobutyric acid may account for differences between inhibitory responses in the hippocampus and thalamus. *Proc. Nat. Acad. Sci.* 92:9515-9519, 1995.

Lytton, W.W. Optimizing synaptic conductance calculation for network simulations. *Neural Computation* 8:501-509, 1996.

Chapter 10 Index

Chapter 11

Modeling networks

NEURON was initially developed to handle models of individual cells or parts of cells, in which complex membrane properties and extended geometry play important roles (Hines 1989; 1993; 1995). However, as the research interests of experimental and theoretical neuroscientists evolved, NEURON has been revised to meet their changing needs. Since the early 1990s it has been used to model networks of biological neurons (e.g. (Destexhe et al. 1993; Lytton et al. 1997; Sohal et al. 2000)). This work stimulated the development of powerful strategies that increase the convenience and efficiency of creating, managing, and exercising such models (Destexhe et al. 1994; Lytton 1996; Hines and Carnevale 2000). Increasing research activity on networks of spiking neurons (e.g. (Riecke et al. 1997; Maass and Bishop 1999)) prompted further enhancements to NEURON, such as inclusion of an event delivery system and development of the `NetCon` (network connection) class (see **Chapter 10**).

Consequently, since the latter 1990s, NEURON has been capable of efficient simulations of networks that may include biophysical neuron models and/or artificial spiking neurons. biophysical neuron models are built around representations of the biophysical mechanisms that are involved in neuronal function, so they have sections, density

What could be more oxymoronic than "real model neuron"?

mechanisms, and synapses (see **Chapter 5**). A synapse onto a biophysical neuron model is a point process with a `NET_RECEIVE` block that affects membrane current (e.g. `ExpSyn`) or a second messenger (see **Chapter 10**). The membrane potential of a biophysical neuron model is governed by complex, interacting nonlinear mechanisms, and spatial nonuniformities may also be present, so numerical integration is required to advance the solution in time.

As we discussed in **Chapter 10**, artificial spiking neurons are really point processes with a `NET_RECEIVE` block that calls `net_event()` (e.g. `IntFire1`). The "membrane state variable" of an artificial neuron has very simple dynamics, and space is not a factor, so the time course of the membrane state is known analytically and it is relatively easy to compute when the next spike will occur. Since artificial neurons do not need numerical integration, they can be used in discrete event simulations that run several orders of magnitude faster than simulations involving biophysical neuron models. Their simplicity also makes it very easy to work with them. Consequently, artificial spiking neurons are particularly useful for prototyping network models.

In this chapter we present an example of how to build network models by combining the strengths of the GUI and `hoc` programming. The GUI tools for creating and managing network models are most appropriate for exploratory simulations of small nets. Once you have set up and tested a small network with the GUI, a click of a button creates a `hoc` file that contains reusable cell class definitions and procedures. This eliminates the laborious, error-prone task of writing "boilerplate" code. Instead, you can just combine NEURON's automatically generated code with your own `hoc` programming to quickly construct large scale nets with complex architectures. Of course, network models can be

constructed entirely by writing hoc code, and NEURON's WWW site contains links to a tutorial for doing just that (Gillies and Sterratt, 2004). However, by taking advantage of GUI shortcuts, you'll save valuable time that can be used to do more research with your models.

Building a simple network with the GUI

Regardless of whether you use the GUI or write hoc code, creating and using a network model involves these basic steps:

1. Define the types of cells.
2. Create each cell in the network.
3. Connect the cells.
4. Set up instrumentation for adjusting model parameters and recording and/or displaying simulation results.
5. Set up controls for running simulations.

We will demonstrate this process by constructing a network model that can be used to examine the contributions of synaptic, cellular, and network properties to the emergence of synchronous and/or correlated firing patterns.

Conceptual model

The conceptual model is a fully connected network, i.e. each cell projects to all other cells, but not to itself (Fig. 11.1 left). All conduction delays and synaptic latencies are identical.

The cells are spontaneously active integrate and fire neurons, similar to those that we discussed in **Chapter 10**. All cells have the same time constant and firing threshold, but in isolation each has its own natural interspike interval (ISI), and the ISIs of the population are distributed uniformly over a fixed range (Fig. 11.1 right).

Figure 11.2 illustrates the dynamics of these cells. Each spike is followed by a "post-spike" hyperpolarization of the membrane state variable *m*, which then decays monoexponentially toward a suprathreshold level. When *m* reaches threshold (1), it triggers another spike and the cycle repeats. A synaptic input hyperpolarizes the cell and prolongs the ISI in which it occurred, shifting subsequent spikes to later times. Each input produces the same hyperpolarization of *m*, regardless of where in the ISI it falls. Even so, the shift of the spike train depends on the timing of the input. If it arrives shortly after a spike, the additional hyperpolarization decays quickly and the spike train shifts by only a small amount (Fig. 11.2 left). An input that arrives late in the ISI can cause a much larger shift in the subsequent spike train (Fig. 11.2 right).

Our task is to create a model that will allow us to examine how synaptic weight, membrane time constant and natural firing frequency, number of cells and conduction latency interact to produce synchronized or correlated spiking in this network.

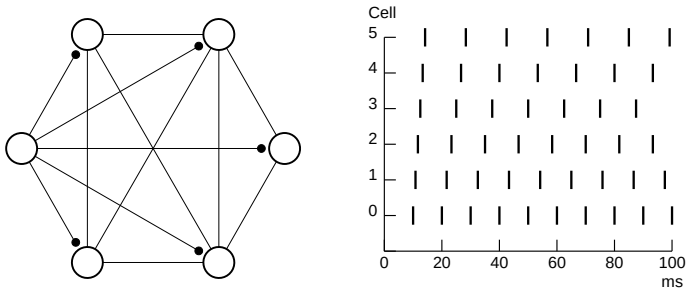


Figure 11.1. Left: An example of a fully connected net. Thin lines indicate reciprocal connections between each pair of cells, and thick lines mark projections from one cell to its targets. Right: When disconnected from each other, every cell has its own natural firing frequency.

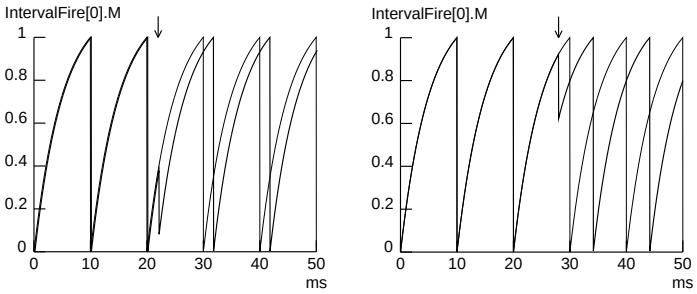


Figure 11.2. Time course of the membrane state variable *m* in the absence (thin traces) and presence (thick traces) of an inhibitory input. Notice that *m* follows a monoexponential "depolarizing" time course which carries it toward a suprathreshold level. When *m* reaches 1, a spike is triggered and *m* is reset to 0 ("post-spike hyperpolarization"). An inhibitory synaptic event causes the same hyperpolarizing shift of *m* no matter where in the ISI it arrives, but its effect on later spike times depends on its relative position in the ISI. Left: Inhibitory events that occur early in the ISI decay quickly, so following spikes are shifted to slightly later times. Right: An inhibitory event that occurs late in the ISI has a longer lasting effect and causes a greater delay of the subsequent spike train.

Adding a new artificial spiking cell to NEURON

Before we start to build this network, we need to add a new kind of artificial spiking cell to NEURON. Our model will use cells whose membrane state variable *m* is governed by the equation

$$\tau \frac{dm}{dt} + m = m_\infty \qquad \text{Eq. 11.3}$$

where $m_\infty > 1$ and is set to a value that produces spontaneous firing with the desired ISI. An input event with weight *w* adds instantaneously to *m*, and if *m* reaches or exceeds the threshold value of 1, the cell "fires," producing an output event and returning *m* to 0. We will call this the `IntervalFire` model, and the NMODL code for it is shown in Listing 11.1. `IntervalFire` has essentially the same dynamics as `IntFire1`, but because its membrane state relaxes toward a suprathreshold value, it uses a `firetime()` function to compute the time of the next spike (see discussions of `IntFire1` and `IntFire2` in **Chapter 10**).

```
NEURON {
  ARTIFICIAL_CELL IntervalFire
  RANGE tau, m, invl
}

PARAMETER {
  tau = 5 (ms)   <1e-9,1e9>
  invl = 10 (ms) <1e-9,1e9>
}

ASSIGNED {
  m
  minf
  t0(ms)
}
```



```
INITIAL {
  minf = 1/(1 - exp(-invl/tau)) : so natural spike interval is invl
  m = 0
  t0 = t
  net_send(firetime(), 1)
}

NET_RECEIVE (w) {
  m = M()
  t0 = t
  if (flag == 0) {
    m = m + w
    if (m > 1) {
      m = 0
      net_event(t)
    }
    net_move(t+firetime())
  } else {
    net_event(t)
    m = 0
    net_send(firetime(), 1)
  }
}

FUNCTION firetime()(ms) { : m < 1 and minf > 1
  firetime = tau*log((minf-m)/(minf - 1))
}

FUNCTION M() {
  M = minf + (m - minf)*exp(-(t - t0)/tau)
}
```

Listing 11.1. NMODL implementation of `IntervalFire`. Figures 11.1 (right) and 11.3 illustrate its operation.

Creating a prototype net with the GUI

After we compile the code in Listing 11.1 (see **Chapter 9**), when we launch `nrngui` these lines should appear at the end of NEURON's startup message

```
Additional mechanisms from files
 invlfire.mod
```

to reassure us that what was defined in `invlfire.mod`--i.e. the `IntervalFire` cell class--is now available. We are ready to use the GUI to build and test a prototype net.

1. Define the types of cells

This involves using the existing cell classes to create the types of cells that we will employ in our network. Our network contains artificial spiking cells, so we need an ArtCellGUI tool, which we get by clicking on Build / NetWork Cell / Artificial Cell in the NEURON Main Menu toolbar (Fig. 11.3).

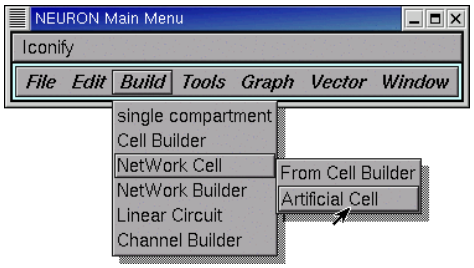


Figure 11.3. Using the NEURON Main Menu to bring up an ArtCellGUI tool.

The gray area in the lower left corner of the ArtCellGUI tool displays a list of the types of artificial spiking cells that will be available to the NetWork Builder. It starts out empty because we haven't done anything yet (Fig. 11.4). To remedy this, click on New and scroll down to select IntervalFire (Fig. 11.5 left), and then release the mouse button. The Artificial Cell types list now contains a new item called IntervalFire, and the right panel of the ArtCellGUI tool shows the user-settable parameters for this cell type (Fig. 11.5 right). These default values are fine for our initial exploratory simulations, so we'll leave them as is.

However, there is one small change that will make it easier to use the NetWork Builder: IntervalFire is a big word, and the NetWork Builder's canvas is relatively small. To avoid clutter, let's give our cell type a short, unique name, like IF (see Figs. 11.6 and 11.7).

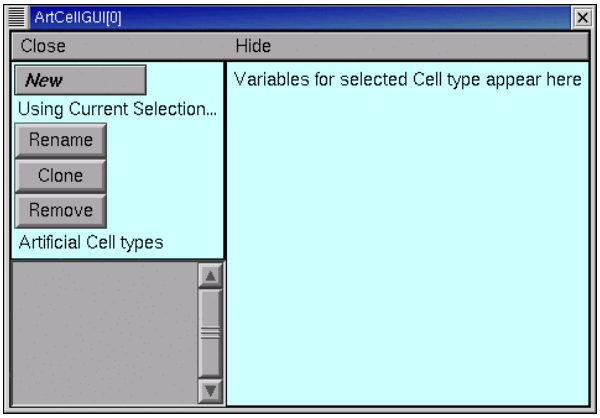


Figure 11.4. The ArtCellGUI tool starts with an empty Artificial Cell types list.

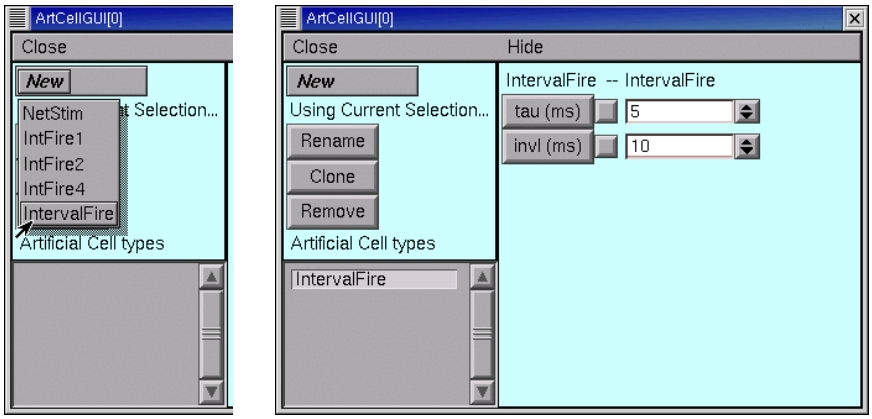
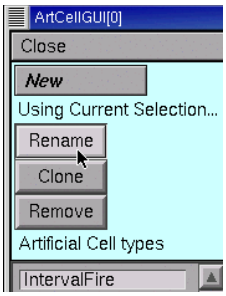
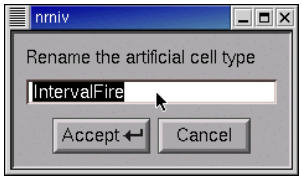


Figure 11.5. Click on New / IntervalFire to add it to the Artificial Cell types list.

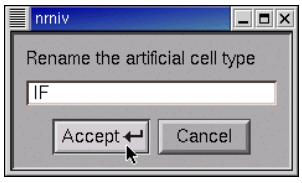
Figure 11.6. Changing the name of one of the Artificial Cell types.



To change the name of one of the Artificial Cell types, select it (if it isn't already selected) and then click on the Rename button.



This pops up a window with a string editor field. Click in the field . . .



. . . change the name to IF, and then click the Accept button.

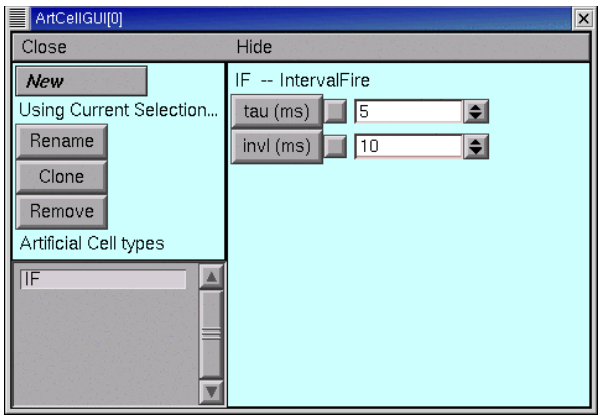


Figure 11.7. The ArtCellGUI tool after renaming the cell type. The right panel shows that IF is based on the IntervalFire class.

Now that we have configured the ArtCellGUI tool, it would be a good idea to save everything to a session file with NEURON Main Menu / File / save session (also see Fig. 1.23 and **Save the model cell** in **Chapter 1**). If you like, you may hide the

ArtCellGUI tool by clicking on Hide just above the drag bar, but don't close it--the NetWork Builder will need it to exist.

2. Create each cell in the network

Having specified the cell types that will be used in the network, we are ready to use the NetWork Builder to create each cell in the network and connect them to each other. In truth, we'll just be creating the specification of each cell in the net; no cells are really created and there is no network until the Create button in the NetWork Builder is ON.

To get a NetWork Builder, click on NEURON Main Menu / Build / NetWork Builder (Fig. 11.8).

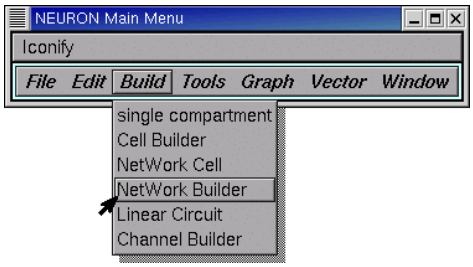


Figure 11.8. Bringing up a NetWork Builder.

The NetWork Builder's drag bar reveals that this tool is an instance of the `NetGUI` class (see Fig. 11.9).

The right panel of a NetWork Builder is a canvas for laying out the network. The "palette" for this canvas is a menu of the cell types that were created with the ArtCellGUI tool. These names appear along the upper left edge of the canvas (for this example, a limited palette indeed: IF is the only cell type). Context-dependent hints are displayed at the top of the canvas.

The left panel of a NetWork Builder contains a set of buttons that control its operation. When a NetWork Builder is first created, its Locate radio button is automatically ON. This means that the NetWork Builder is ready for us to create new cells. We do this by merely following the hint (Fig. 11.10). Notice that the cell names are generated by concatenating the base name (name of the cell type) with a number that starts at 0 and increases by 1 for each new cell. We'll say more about cell names in **A word about cell names** under **7. Caveats and other comments** below.

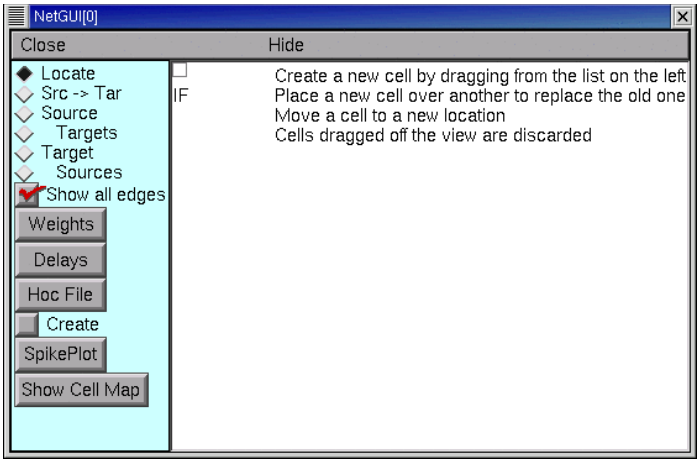
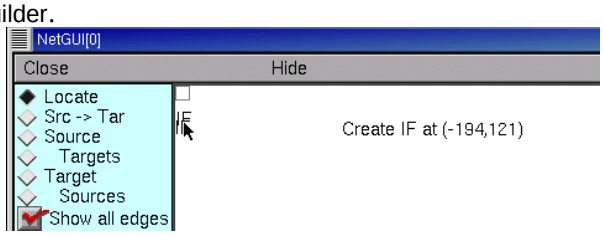


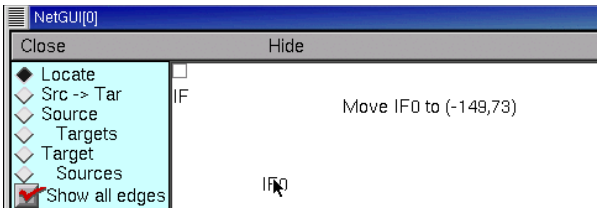
Figure 11.9. A new NetWork Builder.

Figure 11.10. Creating new cells in the NetWork Builder.

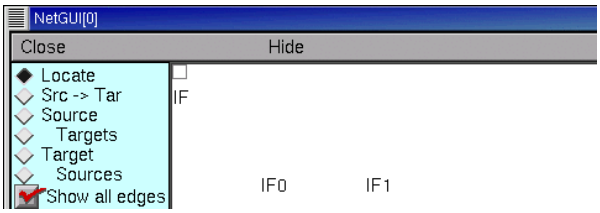
To create a new cell, click on one of the items in the palette (in this example, the only item is IF) and hold the mouse button down . . .



while dragging the new cell to a convenient location on the canvas. Release the mouse button, and you will see a new cell labeled IF0.



After you create a second IF cell, the NetWork Builder should look like this.



If the mouse button is released while the cursor is close to one of the palette items, the new cell will be hard to select since palette item selection takes precedence over selection of a cell. If this happens, just select Translate in the canvas's secondary menu (the canvas is just a modified graph!) and then left click on the canvas and drag it to the right (if you have a three button mouse, or a mouse with a scroll wheel, don't bother with the canvas's menu--just click on the middle button or scroll wheel and drag the canvas). This will pull the cell out from under the palette items, which never move from their position along the left edge of the canvas. Finally, click on one of the radio buttons (Locate, Src -> Tar, etc.) and continue working with the NetWork Builder.

3. Connect the cells

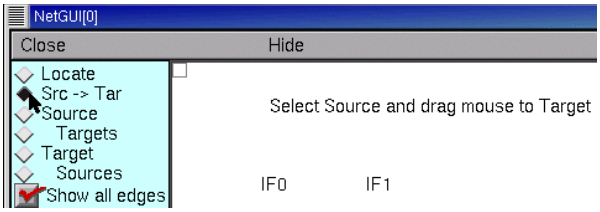
Connecting the cells entails two closely related tasks: setting up the network's architecture, and specifying the delays and weights of these connections.

Setting up network architecture

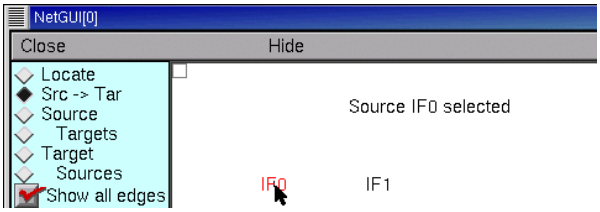
To set up the architecture, we click on the Src -> Tar radio button, read the new hint in the canvas, and do what it says (Fig. 11.11).

Figure 11.11. Setting up network architecture.

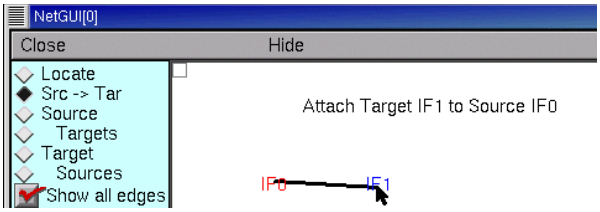
Clicking on the Src -> Tar button brings out a new hint.



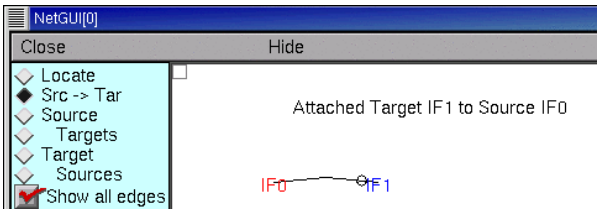
So we click on IF0 and hold the mouse button down while dragging the cursor toward IF1. A thin "rubber band" line will stretch from IF0 to the cursor.



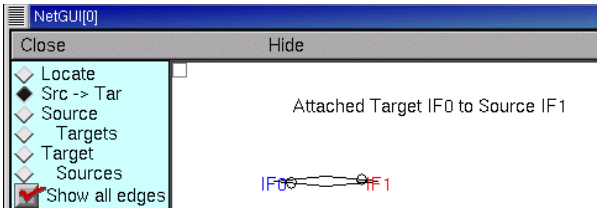
When the cursor is on top of IF1, the rubber band becomes a thick black line, and the hint changes to the message shown here.



To complete the attachment, we just release the mouse button. The projection ("edge") from IF0 to IF1 will appear as a thin line with a slight bend near its midpoint. The O marks the target end of this connection.



Making the reciprocal connection requires only that we click on IF1, drag to IF0, and release the mouse button.



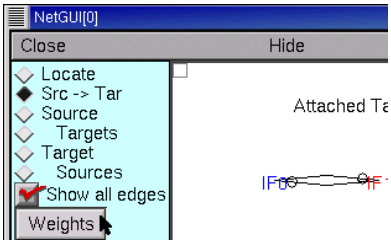
This is a good time to save everything to a session file.

Specifying delays and weights

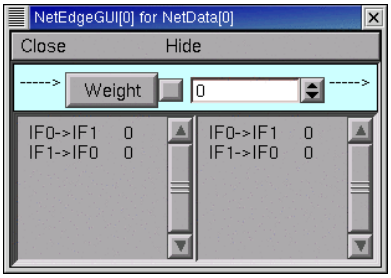
The default initial value of all synaptic weights is 0, i.e. a presynaptic cell will have no effect on its postsynaptic targets. The NetWork Builder has a special tool that we can use to change the weights to what we want (Fig. 11.12).

Figure 11.12. Setting the synaptic weights.

Clicking on the Weights button in the NetWork Builder . . .

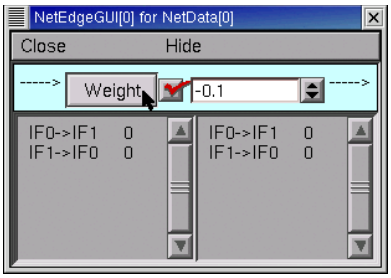


. . . brings up a tool for specifying synaptic weights. The top of this tool has a numeric field with its associated spinner and button (labeled Weight). The value in the numeric field can be set in the usual ways (direct entry, using the spinner, etc.), but note the arrows, which suggest other possibilities.



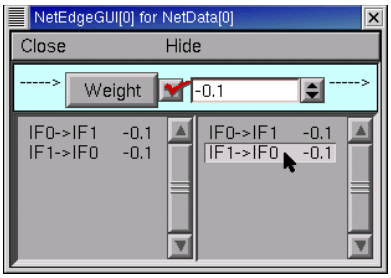
The bottom of the weights tool contains two panels that list the weights of all synaptic connections (aka "edges" in graph theory). Clicking on a connection in the left list copies from the connection to the numeric field, and clicking on a connection in the right list copies from the numeric field to the connection.

Let's give both synapses a weight of -0.1 (mild inhibition). First we change Weight to -0.1 . . .



. . . and then we click on IF0->IF1 and IF1->IF0 in the right panel. We're finished when the weights tool looks like this.

Now we can close this window. If we need it again, clicking on the NetWork Builder's Weights button will bring it back.



All delays are 1 ms by default, which is fine for our purposes. If we wanted to change this to something else, we would click on the NetWork Builder's Delays button (see Fig. 11.9) to bring up a tool for setting delays. The delay tool works just like the weight tool.

At this point, the ArtCellGUI tool plus the NetWork Builder together constitute a complete specification of our network model. We should definitely save another session file before doing anything else!

Now we have a decision to make. We could use the NetWork Builder to create a hoc file that, when executed, would create an instance of our network model. A better choice is to use the GUI to test our model. If there are any problems with what we have done so far, this is a good time to find out and make the necessary corrections.

However, before we can run tests, there must first be something to test. We have a network specification, but no network. As we pointed out earlier in **2. Create each cell in the network**, the network doesn't really exist yet. Clicking on the Create button in the NetWork Builder fixes that (Fig. 11.13).

4. Set up instrumentation

We want to see what our network does, and to explore how its behavior is affected by model parameters. Clicking on the SpikePlot button in the NetWork Builder brings up a tool that will show the input and output spike trains (Fig. 11.14).

We already know how to adjust model parameters. With the NetWork Builder we can change synaptic weights and delays, and the IF cells' properties can be changed with the ArtCellGUI tool. Suddenly, we realize that both IF cells will have the same time constant and firing rate. No problem--our goal is to combine the strengths of the GUI and hoc. We will take care of this later, by combining the hoc code that the NetWork Builder generates with our own hoc code. Using a few lines of hoc, we can easily assign unique firing rates across the entire population of IF cells. And if we insisted on sticking with GUI tools to the bitter end, we could just bring up a PointProcessGroupManager (NEURON Main Menu / Tools / Point Processes / Managers / Point Group), which would allow us to control the attributes of each cell in our network individually.

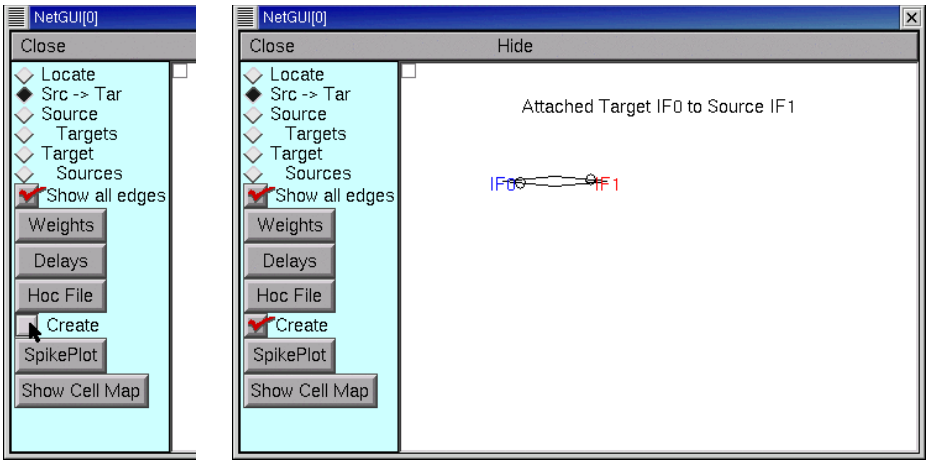


Figure 11.13. Left: Toggling the Create button ON causes the network specification to be executed. Right: Once Create is ON, the representation of the network is available for NEURON's computational engine to use in a simulation.

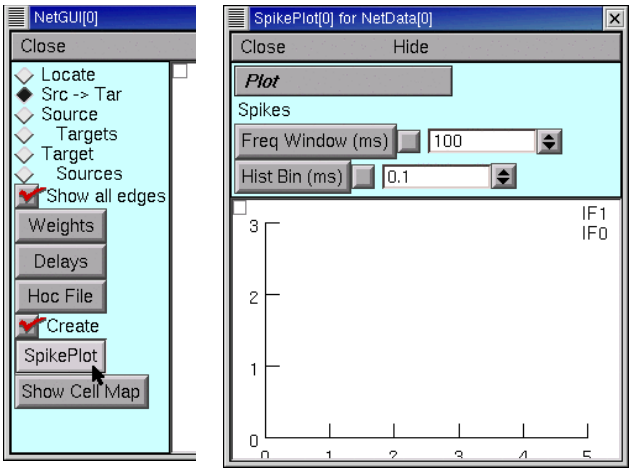


Figure 11.14. The NetWork Builder's SpikePlot button (left) brings up a tool for displaying and analyzing spike trains (right).

5. Set up controls for running simulations

At a minimum, we need a RunControl panel (NEURON Main Menu / Tools / RunControl, as shown in **5. Set up controls for running the simulation** in **Chapter 1**). Also, since our network contains only artificial spiking neurons, we can use adaptive integration to achieve extremely fast, discrete event simulations. We'll need a VariableTimeStep panel (NEURON Main Menu / Tools / VariableStepControl (Fig. 11.15)), which makes it easy to choose between fixed time step or adaptive integration (Fig. 11.16).

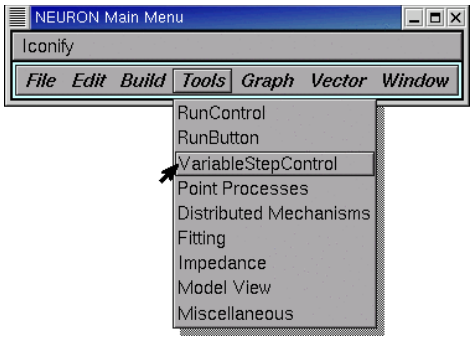


Figure 11.15. Bringing up a VariableTimeStep panel.

Figure 11.16. Toggling adaptive integration ON and OFF.

The VariableTimeStep panel's Use variable dt checkbox is empty, which means that adaptive integration is off.

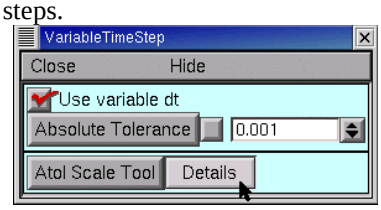
To turn adaptive integration ON, we click on the Use variable dt checkbox.

The check mark in the Use variable dt checkbox tells us that adaptive integration is ON. Clicking on this checkbox again will turn it back OFF so that fixed time steps are used.

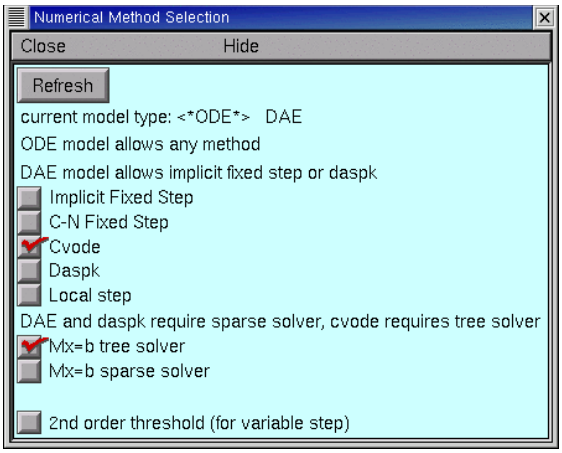
Adaptive integration can use either global or local time steps, each of which has its own particular strengths and weaknesses (see **Adaptive integrators** in **Chapter 7**). The VariableTimeStep panel's default setting is to use global time steps, which is best for models of single cells or perfectly synchronous networks. Our toy network has two identical cells connected by identical synapses, so we would expect them to fire synchronously. However, when we build our net with hoc code, the cells will all have different natural firing frequencies, and who can tell in advance that they will achieve perfect synchrony? Besides, this is a tutorial, so let's use local time steps (Fig. 11.17).

Figure 11.17. Toggling between global and local time steps.

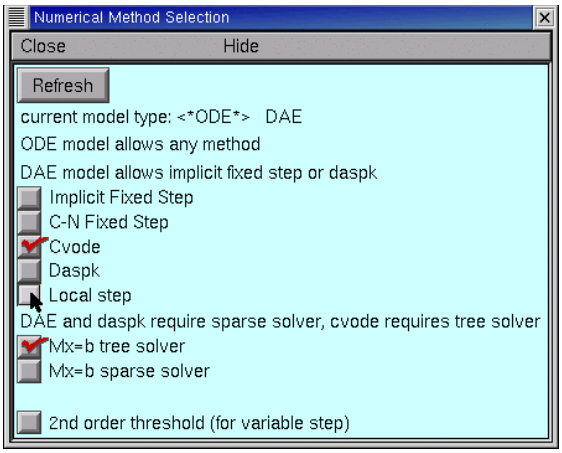
To specify whether to use global or local time steps, we first click on the VariableTimeStep panel's Details button.



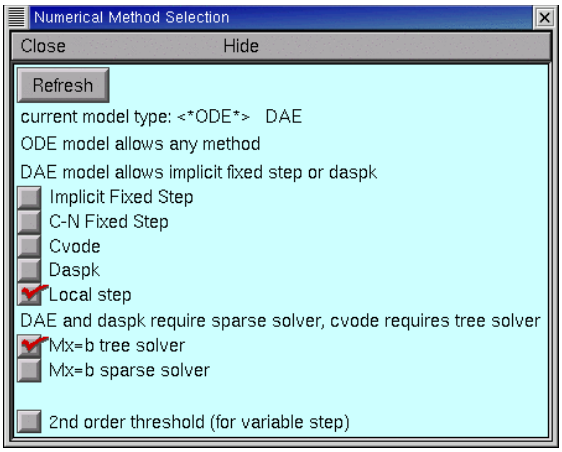
We are concerned with the Local step checkbox, which is empty. To activate the use of local variable time steps . . .



. . . we just click on the Local step checkbox . . .



. . . and now each cell in our network will advance with its own time step. If we want to restore global time steps, we can just click on the Cvode button. Now we can close this panel; should we need it again, we only have to click on the VariableTimeStep panel's Details button.



After rearrangement, the various windows we have created should look something like Fig. 11.18. The tools we used to specify the network are on the left, simulation controls are in the middle, and the display of simulation results is on the right. Quick, save it to a session file!

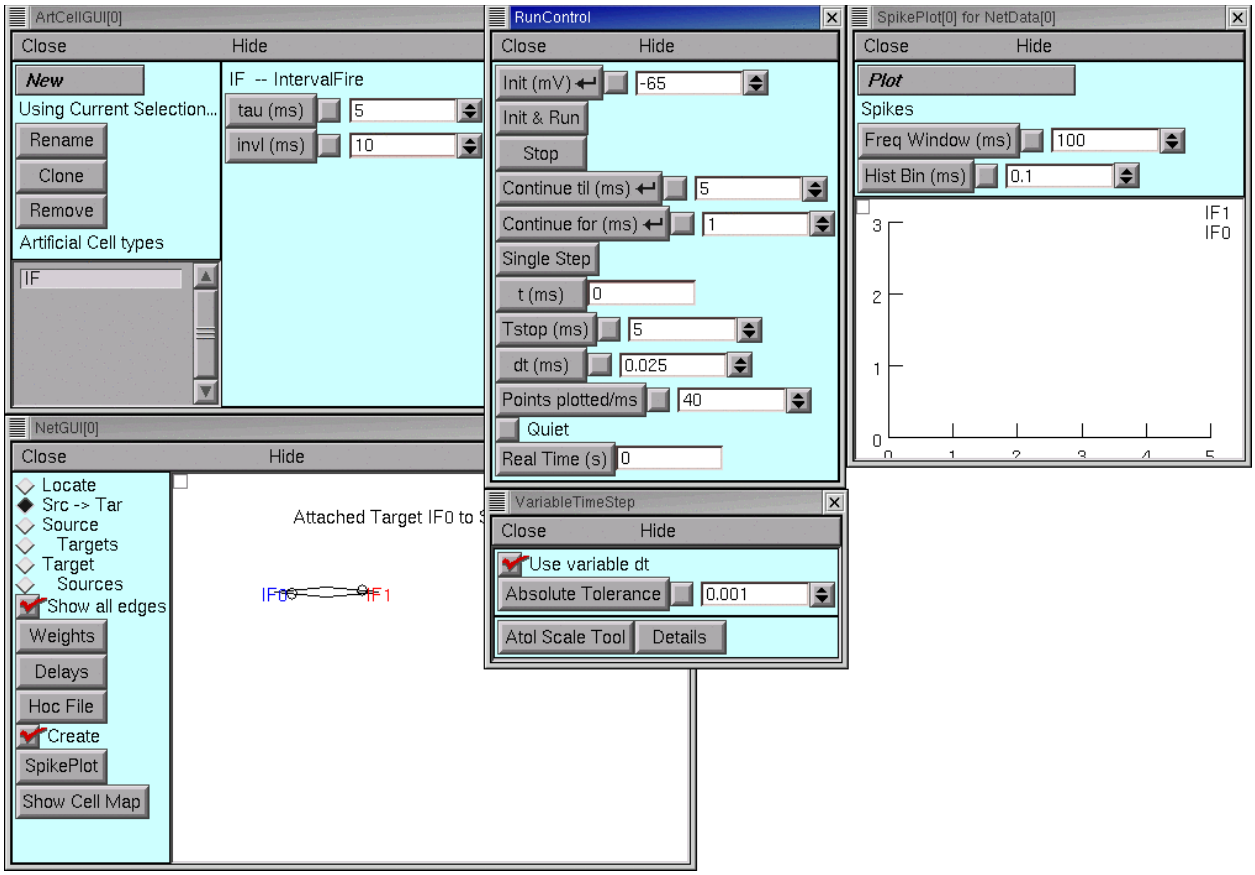


Figure 11.18. The completed model with controls for running simulations and displaying results.

6. Run a simulation

This is almost too easy. Clicking on Init & Run in the RunControl panel, we see--nothing! Well, almost nothing. The t field in the RunControl panel shows us that time advanced from 0 to 5 ms, but there were no spikes. A glance at the ArtCellGUI tool tells us why: invl is 5 ms, which means that our cells won't fire their first spikes for another 5 ms. Let's change Tstop to 200 ms so we'll get a lot of spikes, and try again. This time we're successful (Fig. 11.19).

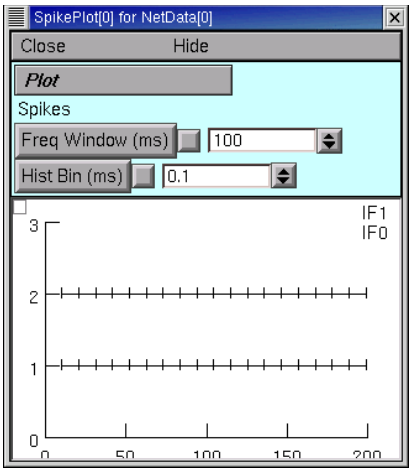


Figure 11.19. The SpikePlot shows the spike trains generated by the cells in our network model. Note that rasters correspond to cell names from top to bottom, and that the raster for cell i is plotted along the line $y = i + 1$.

7. Caveats and other comments

Changing the properties of an existing network

As we have seen, the ArtCellGUI tool is used to specify what artificial spiking cell types are available to a NetWork Builder. The same ArtCellGUI tool can be used to adjust the parameters of those cells, and such changes take effect immediately, even if the network already exists (i.e. even if the NetWork Builder's Create button is ON).

The NetReadyCellGUI tool (NEURON Main Menu / Build / NetWork Cell / From Cell Builder) is used to configure biophysical neuron model types for use with a NetWork Builder. In fact, we would use a separate NetReadyCellGUI instance for each different type of biophysical neuron model we wanted to use in the net. The NetReadyCellGUI tool has its own CellBuilder for specifying topology, geometry, and biophysical properties,

plus a SynapseTypes tool for adding synaptic mechanisms to the cell (see the tutorial at `http://www.neuron.yale.edu/neuron/docs/netbuild/main.html`). However, changes made with a NetReadyCellGUI tool do *not* affect an existing network; instead, it is necessary to save a session file, exit NEURON, restart and reload the session file.

What about changes to the network itself? Any changes whatsoever can be made in the NetWork Builder, as long as its Create button is OFF. Once it is ON, some changes are possible (e.g. adding new cells and synaptic connections to an existing network), but additional actions may be required (a pre-existing SpikePlot will not show spike trains from new cells), and there is a risk of introducing a mismatch between one's conceptual model and what is actually in the computer. The best policy is to toggle Create OFF (see Fig. 11.20), make whatever changes are needed, save everything to a session file, exit NEURON, and then restart and load the new session file.

Figure 11.20. Trying to turn Create OFF brings up this window, which offers the opportunity to change one's mind. Select Turn off if it is necessary to make substantial changes to an existing network in the NetWork Builder.

A word about cell names

As we mentioned above in **2. Create each cell in the network**, the cell names that appear in the NetWork Builder are generated automatically by concatenating the name of the cell type with a sequence of numbers that starts at 0 and increases by 1 for each

additional cell. But that's only part of the story. These are really only short "nicknames," a stratagem for preventing the NetWork Builder and its associated tools from being cluttered with long character strings.

This is fine as long as the NetWork Builder does everything we want. But suppose we need to use one of NEURON's other GUI tools, or we have to write some hoc code that refers to one of our model's cells? For example, we might have a network that includes a biophysical neuron model, and we want to see the time course of somatic membrane potential. In that case, it is absolutely necessary to know the actual cell names.

That's where the NetWork Builder's Cell Map comes in. Clicking on Show Cell Map brings up a small window that often needs to be widened by clicking and dragging on its left or right margin (Fig. 11.21). Now we realize that, when we used the ArtCellGUI tool to create an IF cell "type," we were actually specifying a new cell class whose name is a concatenation of our "type" (IF), an underscore character, and the name of the root class (the name of the class that we based IF on, which was IntervalFire).

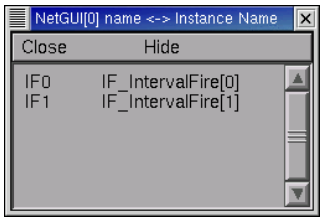


Figure 11.21. The Cell Map for our toy network. See text for details.

Combining the GUI and programming

Creating a hoc file from the NetWork Builder

Having tested our prototype model, we are now ready to write a hoc file that can be mined for reusable code. Clicking on the Hoc File button in the NetWork Builder brings up a tool that looks much like what we used to specify file name and location when saving a session file. Once we're satisfied with our choices, clicking on this tool's "Open" button writes the hoc file (yes, the button should say Close). This file, which we will call `prototype.hoc`, is presented in Listing 11.2, and executing it would recreate the toy network that we just built with the NetWork Builder.

```
// NetGUI default section. Artificial cells, if any, are located here.
  create acell_home_
  access acell_home_

//Network cell templates
//Artificial cells
//   IF_IntervalFire


begintemplate IF_IntervalFire
public pp, connect2target, x, y, z, position, is_art
external acell_home_
objref pp
proc init() {
  acell_home_ pp = new IntervalFire(.5)
}
func is_art() { return 1 }
proc connect2target() { $o2 = new NetCon(pp, $o1) }
proc position(){x=$1  y=$2  z=$3}
endtemplate IF_IntervalFire

//Network specification interface

objref cells, nclist, netcon
{cells = new List()  nclist = new List()}

func cell_append() {cells.append($o1)  $o1.position($2,$3,$4)
   return cells.count - 1
}
```



```
func nc_append() {//srcindex, tarcelindex, synindex
  if ($3 >= 0) {
    cells.object($1).connect2target(cells.object($2).synlist.object($3), \
                                    netcon)
    netcon.weight = $4   netcon.delay = $5
  }else{
    cells.object($1).connect2target(cells.object($2).pp,netcon)
    netcon.weight = $4   netcon.delay = $5
  }
  nclist.append(netcon)
  return nclist.count - 1
}

//Network instantiation

  /* IF0 */  cell_append(new IF_IntervalFire(),    -149,   73, 0)
  /* IF1 */  cell_append(new IF_IntervalFire(),    -67,    73, 0)
  /* IF1 -> IF0    */  nc_append(1, 0, -1,  -0.1,1)
  /* IF0 -> IF1    */  nc_append(0, 1, -1,  -0.1,1)
```

Listing 11.2. Clicking on the Hoc File button in the NetWork Builder produces a file which we have called `prototype.hoc`.

A quick glance over the entire listing reveals that `prototype.hoc` is organized into several parts, which are introduced by one or more lines of descriptive comments. Let us consider each of these in turn, to see how it works and think about what we might reuse to make a network of any size we like.

NetGUI default section

The first part of the file creates `acell_home_` and make this the default section. What is a *section* doing in a model that contains artificial spiking cells? Remember that artificial spiking cells are basically point processes (see **Artificial spiking cells** in **Chapter 10**), and just like other point processes, they must be attached to a section. Suddenly the meaning of the comment `Artificial cells, if any, are located here` becomes clear: `acell_home_` is merely a "host" for artificial spiking

cells. It has no biophysical mechanisms of its own, so it introduces negligible computational overhead.

Network cell templates

The NetWork Builder and its associated tools make extensive use of object-oriented programming. Each cell in the network is an instance of a cell class, and this is where the templates that declare these classes are located (templates and other aspects of object-oriented programming in NEURON are discussed in **Chapter 13**).

The comments that precede the templates contain a list of the cell class names. Our toy network uses only one cell class, so `prototype.hoc` contains only one template, which defines the `IF_IntervalFire` class. When biophysical neuron models are present, they are declared first. Thus, if we had a NetWork Builder whose palette contained a biophysical neuron model type called pyr, and an artificial spiking cell type S that was derived from the `NetStim` class, the corresponding cell classes would be called `pyr_Cell` and `S_NetStim`, and the header in the exported hoc file would read

```
//Network cell templates
//   pyr_Cell
//Artificial cells
//   S_NetStim
```

Functions and procedures with the same names as those contained in the `IF_IntervalFire` template will be found in every cell class used by a NetWork Builder (although some of their internal details may differ). The first of these is `init()`, which is executed automatically whenever a new instance of the `IF_IntervalFire` class is created. This in turn creates a new instance of the `IntervalFire` class that will be

associated with the `acell_home_` section. As an aside, we should mention that this is an example of how the functionality of a basic object class can be enhanced by wrapping it inside a template in order to define a new class with additional features, i.e. an example of emulating inheritance in `hoc` (see **Polymorphism and inheritance** in **Chapter 13**).

The remaining `func`s and `proc`s are public so they can be called from outside the template. If we ever need to determine which elements in a network are artificial spiking cells and which are biophysical neuron models, `is_art()` is clearly the way to do it. The next is `connect2target()`, which looks useful for setting up network connections, but it turns out that the `hoc` code we write ourselves won't call this directly (see **Network specification interface below**). The last is `position()` which can be used to specify unique xyz coordinates for each instance of this cell. The coordinates themselves are public (accessible from outside the template--see **Chapter 13** for more about accessing variables, `func`s and `proc`s declared in a template). Position may seem an arcane attribute for an artificial spiking neuron, but it is helpful for algorithmically creating networks in which connectivity or synaptic weight are functions of location or distance between cells.

Network specification interface

These are the variables and functions that we will actually call from our own hoc code. These are intended to offer us a uniform, compact and convenient syntax for setting up our own network. That is, they serve as a "programming interface" between the code we write and the lower level code that accomplishes our ultimate aims.

The purpose of the first two lines in this part of `prototype.hoc` is evident if we keep in mind that the NetWork Builder implements a network model with objects, some of which represent cells while others represent the connections between them. The `List` class is the programmer's workhorse for managing collections of objects, so it is reasonable that the cells and connections of our network model will be packaged into two `List`s called `cells` and `nclist`, respectively.

The functions that add new elements to these `List`s are `cell_append()` and `nc_append()`, respectively. The first argument to `cell_append()` is an `objref` that points to a new cell that is to be added to the list, and the remaining arguments are the xyz coordinates that are to be assigned to that cell. The `nc_append()` function uses an `if . . . else` to deal properly with either biophysical neuron models or artificial spiking cells. In either case, its first two arguments are integers that indicate which elements in `cells` are the `objref`s that correspond to the pre- and postsynaptic cells, and the last two arguments are the synaptic weight and delay. If the postsynaptic cell is a biophysical neuron model, one or more synaptic mechanisms will be attached to it (see the tutorial at `http://www.neuron.yale.edu/neuron/docs/netbuild/main.html`). In this case, the third argument to `nc_append()` is a nonnegative integer that specifies which synaptic mechanism is to be the target of the new `NetCon`. If instead the postsynaptic cell is an artificial spiking cell, the argument is just -1.

Network instantiation

So far everything has been quite generic, in the sense that we can use it to create cells and assemble them into whatever network architecture we desire. In other words, the

code up to this point is exactly the reusable code that we needed. The statements in the "network instantiation" group are just a concrete demonstation of how to use it to spawn a particular number of cells and link them with a specific network of connections. Let's make a copy of `prototype.hoc`, call it `netdefs.hoc`, and then insert `//` at the beginning of each of last four lines of `netdefs.hoc` so they persist as a reminder of how to call `cell_append()` and `nc_append()` but won't be executed. We are now ready to use `netdefs.hoc` to help us build our own networks.

Exploiting the reusable code

Where should we begin? A good way to start is by imagining the overall organization of the entire program at the "big picture" level. We'll need the GUI library, the class definitions and other code in `netdefs.hoc`, code to specify the network model itself, and code that sets up controls for adjusting model parameters, running simulations, and displaying simulation results. Following our recommended practices of modular programming and separating model specification from user interface (see **Elementary project management** in **Chapter 6**), we turn this informal outline into an `init.hoc` file that pulls all these pieces together (Listing 11.3).

```
load_file("nrngui.hoc")
load_file("netdefs.hoc")  // code from NetWork Builder-generated hoc file
load_file("makenet.hoc")  // specifies network
load_file("rig.hoc")  // for adjusting model params and running simulations
```

Listing 11.3. The `init.hoc` for our own network program.

For now, we can comment out the last two lines with `//` so we can test `netdefs.hoc` by using NEURON to execute `init.hoc`. and then typing a few

commands at the `oc>` prompt (user entries are **`Courier bold`** while the interpreter's output is plain `Courier`).

```
Additional mechanisms from files
 invlfire.mod
   1
   1
oc>objref foo
oc>foo = new IF_IntervalFire()
oc>foo
   IF_IntervalFire[0]
oc>
```

So far so good. We are ready to apply the strategy of iterative program development (see **Iterative program development** in **Chapter 6**) to fill in the details.

The first detail is how to create a network of a specific size. If we call the number of cells `ncell`, then this loop

```
for i=0, ncell-1 {
  cell_append(new IF_IntervalFire(), i, 0, 0)
}
```

will make them for us, and this nested loop

```
for i=0, ncell-1 for j=0, ncell-1 if (i != j) {
  nc_append(i, j, -1, 0, 1)
}
```

will attach them to each other. An initial stab at embedding both of these in a procedure which takes a single argument that specifies the size of the net is

```
proc createnet() { local i, j
  ncell = $1
  for i=0, $1-1 {
    cell_append(new IF_IntervalFire(), i, 0, 0)
  }
  for i=0, $1-1 for j=0, $1-1 if (i != j) {
    nc_append(i, j, -1, 0, 1)
  }
}
```

and that's what we put in the first version of `makenet.hoc`.

We can test this by uncommenting the `load_file("makenet.hoc")` line in `init.hoc`, using NEURON to execute `init.hoc`, and then typing a few commands at the `oc>` prompt.

```
oc>createnet(2)
oc>ncell
   2
oc>print cells, nclist
   List[8] List[9]
oc>print cells.count, nclist.count
   2 2
oc>for i=0,1 print cells.object(i), nclist.object(i)
IF_IntervalFire[0] NetCon[0]
IF_IntervalFire[1] NetCon[1]
oc>
```

So it works. But almost immediately a wish list of improvements comes to mind. In order to try networks of different sizes, we'll be calling `createnet()` more than once during a single session. As it stands, repeated calls to `createnet()` just tack more and more new cells and connections onto the ends of the `cells` and `nclist` lists. Also, `createnet()` should be protected from nonsense arguments (a network should have at least two cells).

We can add these fixes by changing `ncell = $1` to

```
if ($1<2) { $1 = 2 }
ncell = $1
nclist.remove_all()
cells.remove_all()
```

The first line ensures our net will have two or more cells. The last two lines use the `List` class's `remove_all()` to purge `cells` and `nclist`. Of course we check this

```
oc>createnet(1)
oc>ncell
   2
oc>createnet(2)
oc>ncell
   2
```



```
oc>createnet(3)
oc>ncell
   3
oc>
```

which is exactly what should happen.

What else should go into `makenet.hoc`? How about procedures that make it easy to change the properties of the cells and connections? As a case in point, this

```
proc delay() { local i
  del = $1
  for i=0, nclist.count-1 {
    nclist.object(i).delay = $1
  }
}
```

lets us set all synaptic delays to the same value by calling `delay()` with an appropriate argument. Similar `proc`s can take care of weights and cellular time constants. Setting ISIs seems more complicated at first, but after a few false starts we come up with

```
proc interval() { local i, x, dx
  low = $1
  high = $2
  x = low
  dx = (high - low)/(cells.count-1)
  for i=0, cells.count-1 {
    cells.object(i).pp.invl = x
    x += dx
  }
}
```

This assigns the `low` ISI to the first cell in `cells`, the `high` ISI to the last cell in `cells`, and evenly spaced intermediate values to the other cells.

Does that mean the first cell is the fastest spiker, and the last is the slowest? Only if we are careful about the argument sequence when we call `interval()`. For that matter, what prevents us from calling `interval()` with one or both arguments < 0? Come to

think of it, some of our other `proc`s might also benefit by being protected from nonsense arguments. We might protect against negative delays by changing

```
del = $1
```

in `proc delay()` to

```
if ($1<0) $1=0
del = $1
```

and we could insert similar argument-trapping code into other `proc`s as necessary.

However, it makes more sense to try to identify a common task that can be split out into a separate function that can be called by any `proc` that needs it. It may help to tabulate the vulnerable variables and the constraints we want to enforce.

Variable	**Constraint**
`ncell`	≥ 2
`tau`	> 0
`low` ISI	> 0
`high` ISI	$\geq$ `low` ISI
`del`	≥ 0

Most of these constraints are "greater than or equal to," the two holdouts being `tau` and `low` ISI. After a moment we realize that there are practical lower limits to these variables--say 0.1 ms for `tau` and 1 ms for `low` ISI--so "greater than or equal to" constraints can be applied to all.

The final version of `makenet.hoc` (Listing 11.4) contains all of these refinements. The statements at the very end create a network by calling our revised `proc`s.


```
/*
returns value >= $2
for bulletproofing procs against nonsense arguments
*/

func ge() {
  if ($1<$2) {
    $1=$2
  }
  return $1
}

////////// create a network //////////

// argument is desired number of cells

proc createnet() { local i, j
  $1 = ge($1,2) // force net to have at least two cells
  ncell = $1
  // so we can make a new net without having to exit and restart
  nclist.remove_all()
  cells.remove_all()
  for i=0, $1-1 {
    cell_append(new IF_IntervalFire(), i, 0, 0)
  }
  for i=0, $1-1 for j=0, $1-1 if (i != j) {
    // let weight be 0; we'll give it a nonzero value elsewhere
    nc_append(i, j, -1, 0, 1)
  }
  objref netcon  // leave no loose ends (see nc_append())
}

////////// specify parameters //////////

// call this settau() to avoid conflict with scalar tau

proc settau() { local i
  $1 = ge($1,0.1)  // min tau is 0.1 ms
  tau = $1
  for i=0, cells.count-1 {
    cells.object(i).pp.tau = $1
  }
}

// args are low and high

proc interval() { local i, x, dx
  $1 = ge($1,1)  // min low ISI is 1 ms
  $2 = ge($2,$1)
  low = $1
  high = $2
  x = low
  dx = (high - low)/(cells.count-1)
  for i=0, cells.count-1 {
    cells.object(i).pp.invl = x
    x += dx
  }
}
```



```
proc weight() { local i
  w = $1
  for i=0, nclist.count-1 {
    nclist.object(i).weight = $1
  }
}

proc delay() { local i
  $1 = ge($1,0)  // min del is 0 ms
  del = $1
  for i=0, nclist.count-1 {
    nclist.object(i).delay = $1
  }
}

////////// actually make net and set parameters //////////

createnet(2)
settau(10)
interval(10, 11)
weight(0)
delay(1)
```

Listing 11.4. Final implementation of `makenet.hoc`.

Time for more tests!

```
oc>del
0
oc>{delay(-1) print del}
0
oc>{delay(3) print del}
3
oc>createnet(4)
oc>ncell
   4
oc>del
3
oc>
```

Of course we can and should test the other `proc`s, especially `interval()`. As certain mathematics texts say, "this is left as an exercise to the reader."

Our attention now shifts to creating the user interface for adjusting model parameters, controlling simulations, and displaying results. To evoke the metaphor of an experimental rig, this is placed in a file called `rig.hoc`.

An initial implementation of `rig.hoc` might look like this

```
load_file("runctl.ses")  // RunControl and VariableTimeStep

xpanel("Model parameters")
xvalue("Weight","w", 1,"weight(w)", 0, 0 )
xvalue("Delay (ms)","del", 1,"delay(del)", 0, 0 )
xvalue("Cell time constant (ms)","tau", 1,"settau(tau)", 0, 0 )
xvalue("Shortest natural ISI","low", 1,"interval(low, high)", 0, 0 )
xvalue("Longest natural ISI","high", 1,"interval(low, high)", 0, 0 )
xpanel(500,400)
```

In the spirit of taking advantage of every shortcut the GUI offers, the first statement loads a session file that recreates a RunControl and a VariableTimeStep panel configured for the desired simulation duration (Tstop = 500 ms) and integration method (adaptive integration with local time steps). The other statements set up a panel with numeric fields and controls for displaying and adjusting model parameters. This implementation of `rig.hoc` lacks two important features: a graph that displays spike trains, and the ability to change the number of cells in the network.

What about plots of spike trains? There is a way to create a graph that provides all the functionality of the NetWork Builder's own SpikePlot, but analyzing the necessary code would lead us into details that really belong in a chapter on advanced GUI programming. For didactic purposes it is better if we make our own raster plot, if only because this will draw our attention to topics that are likely to be more widely useful.

To prepare to record and plot spike trains, we can insert the following code right after the `load_file()` statement:


```
objref netcon, vec, spikes, nil, graster

proc preprasterplot() {
  spikes = new List()
  for i=0,cells.count()-1 {
    vec = new Vector()
    netcon = new NetCon(cells.object(i).pp, nil)
    netcon.record(vec)
    spikes.append(vec)
  }
  objref netcon, vec

  graster = new Graph(0)
  graster.view(0, 0, tstop, cells.count(), 300, 105, 300.48, 200.32)
}

preprasterplot()
```

For each cell in the net, this creates a new `Vector`, uses the `NetCon` class's `record()` method to record the time of that cell's spikes into the `Vector`, and appends the `Vector` to a `List`. After the end of the `for` loop that iterates over the cells, the `netcon` and `vec` `objref`s point to the last `NetCon` and `Vector` that were created, exposing them to possible interference if we ever do anything that reuses these `objref` names. The `objref netcon, vec` statement breaks the link between them and the objects, thereby preventing such undesirable effects.

The last two statements in `preprasterplot()` create a `Graph` and place it at a desired location on the screen. How can we tell what the numeric values should be for the arguments in the `graster.view()` statement? By creating a graph (NEURON Main Menu / Graph / Voltage axis will do), dragging it to the desired location, saving it to a session file all by itself, and then stealing the argument list from that session file's `save_window_.view()` statement--being careful to change the third and fourth arguments so that the x and y axes span the correct range of values. No cut and try guesswork for us! While we're at it, we might as well use the same strategy to fix the

location for our model parameter panel, but now we only need the fifth and sixth arguments to `view()`, which are the screen coordinates where the `Graph` is positioned. For my monitor, this means the second `xpanel` statement becomes `xpanel(300,370)`.

Running a new test, we find that our user interface looks like Fig. 11.22. Everything is in the right place, and time advances when we click on Init & Run, but no rasters are plotted.

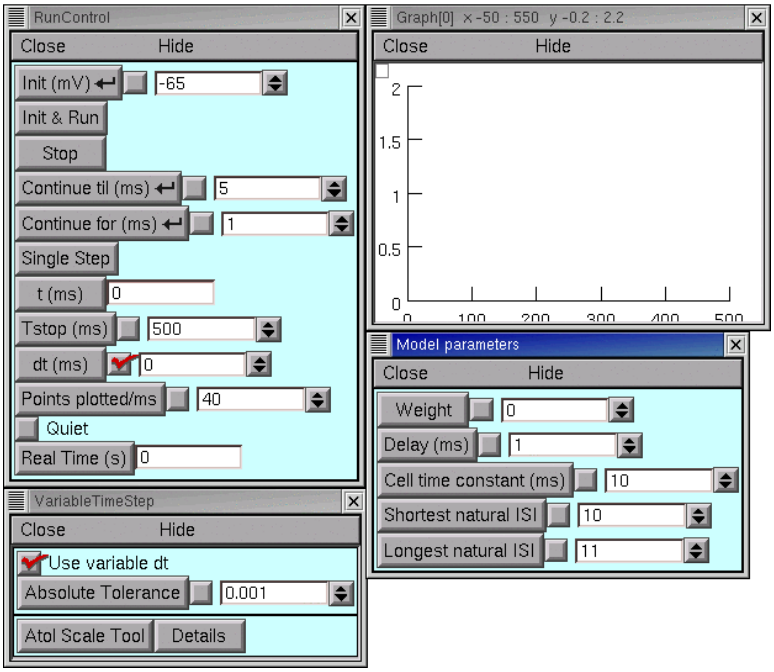


Figure 11.22. The user interface after the first revision to `rig.hoc`, in which we added `preprasterplot()`.

For each cell we need to draw a sequence of short vertical lines on `graster` whose x coordinates are the times at which that cell fired. To help us tell one cell's spikes from another's, the vertical placement of their rasters should correspond to their ordinal position in `cells`. We can do this by inserting the following code into `rig.hoc`, right

after the call to `preprasterplot()`. The first thing that `proc showraster()` does is to clear any previous rasters off the `Graph`. Then, for each cell in turn, it uses three `Vector` class methods in succession: `c()` to create a `Vector` that has as many elements as the number of spikes that the cell fired, `fill()` to fill those elements with an integer that is one more than the ordinal position of that cell in `cells`, and `mark()` to mark the firing times.

```
objref spikey

proc showraster() {
  graster.erase_all()
  for i = 0,cells.count()-1 {
    spikey = spikes.object(i).c
    spikey.fill(i+1)
    spikey.mark(graster, spikes.object(i), "|", 6)
  }
  objref spikey
}
```

Testing once again, we run a simulation and then type `showraster()` at the `oc>` prompt, and sure enough, there are the spikes. We change the longest natural ISI to 20 ms, run another simulation, and type `showraster()` once more, and it works again.

All this typing is tedious. Why not customize the `run()` procedure so that it automatically calls `showraster()` after each simulation? Adding this

```
proc run() {
  stdinit()
  continuerun(tstop)
  showraster()
}
```

to the end of `rig.hoc` does the job (see **An outline of the standard run system** in **Chapter 7: How to control simulations**).

Another test and we are overcome with satisfaction--it works. Then we change Tstop to 200 ms, run a simulation, and are disappointed that the raster plot's x axis does not rescale to match the new Tstop. One simple fix for this is to write a custom `init()` procedure that uses the Graph class's `size()` method to adjust the size of the raster plot during initialization (see **Default initialization in the standard run system: stdinit() and init()** in **Chapter 8**). So we insert this

```
proc init() {
  finitialize(v_init)
  graster.erase_all()
  graster.size(0,tstop,0,cells.count())
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

right after our custom `run()`. Notice that this also rescales the y axis, which will be helpful when we finally add the ability to change the number of cells in the network.

Success upon success! It works!

We can finally get around to changing the number of cells. Let's think this out carefully before doing anything. We'll need a new control in the `xpanel`, to show how many cells there are and let us specify a new number. That's easy--just put this line

```
xvalue("Number of cells","ncell", 1,"recreate(ncell)", 0, 0 )
```

right after `xpanel("Model parameters")` so that when we change the value of `ncell`, we automatically call a new procedure called `recreate()` that will throw away the old cells and their connections, and create a new set of each.

But what goes in `recreate()`? We'll want the new cells and connections to have the same properties as the old ones. And we'll have to replace the old raster plot with a new one, complete with all the `NetCons` and `Vectors` that it uses to record spikes. So `recreate()` should be

```
proc recreate() {
  createnet($1)
  settau(tau)
  interval(low, high)
  weight(w)
  delay(del)
  preprasterplot()
}
```

A good place for this is right before the `xpanel`'s code.

So now we have completed `rig.hoc` (see Listing 11.5). The parameter panel has all the right buttons (Fig. 11.23) so it is easy to explore the effects of parameter changes (Fig. 11.24). How to develop an understanding of what accounts for these effects is beyond the scope of this chapter, but we can offer one hint: run some simulations of a net containing only 2 or 3 cells, using fixed time steps, and plot their membrane state variables (well, their `M` functions).

```
////////// user interface //////////
load_file("runctl.ses")  // RunControl and VariableTimeStep
// prepare to record and display spike trains
objref netcon, vec, spikes, nil, graster
```



```
proc preprasterplot() {
  spikes = new List()
  for i=0,cells.count()-1 {
    vec = new Vector()
    netcon = new NetCon(cells.object(i).pp, nil)
    netcon.record(vec)
    spikes.append(vec)
  }
  objref netcon, vec

  graster = new Graph(0)
  graster.view(0, 0, tstop, cells.count(), 300, 105, 300.48, 200.32)
}

preprasterplot()

objref spikey

proc showraster() {
  graster.erase_all()
  for i = 0,cells.count()-1 {
    spikey = spikes.object(i).c
    spikey.fill(i+1)
    spikey.mark(graster, spikes.object(i), "|", 6)
  }
  objref spikey
}

// destroys existing net and makes a new one
// also spawns a new spike train raster plot
// called only if we need a different number of cells

proc recreate() {
  createnet($1)
  settau(tau)
  interval(low, high)
  weight(w)
  delay(del)
  preprasterplot()
}

xpanel("Model parameters")
xvalue("Number of cells","ncell", 1,"recreate(ncell)", 0, 0 )
xvalue("Weight","w", 1,"weight(w)", 0, 0 )
xvalue("Delay (ms)","del", 1,"delay(del)", 0, 0 )
xvalue("Cell time constant (ms)","tau", 1,"settau(tau)", 0, 0 )
xvalue("Shortest natural ISI","low", 1,"interval(low, high)", 0, 0 )
xvalue("Longest natural ISI","high", 1,"interval(low, high)", 0, 0 )
xpanel(300,370)

////////// custom run() and init() //////////

proc run() {
  stdinit()
  continuerun(tstop)
  showraster()  // show results at the end of each simulation
}
```



```
proc init() {
  finitialize(v_init)
  graster.erase_all()
  graster.size(0,tstop,0,cells.count())  // rescale x and y axes
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

Listing 11.5. Complete implementation of `rig.hoc`.

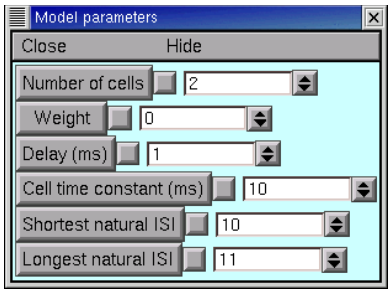
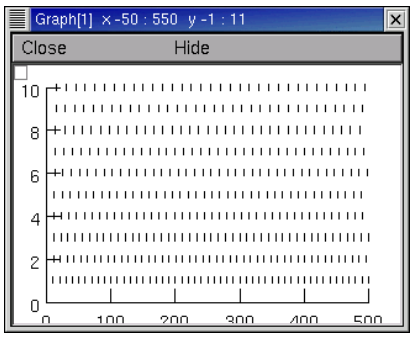


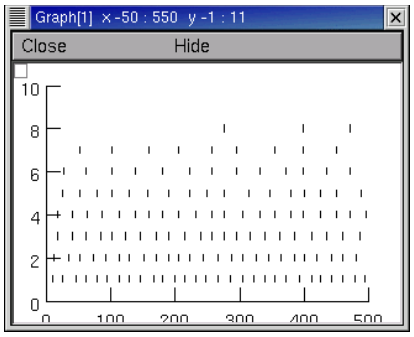
Figure 11.23. The parameter panel after addition of a control for changing the number of cells.

Figure 11.24. Simulations of a fully connected network with 10 cells whose natural ISIs are spaced uniformly over the range 10-15 ms. The rasters are arranged with ISIs in descending order from top to bottom.

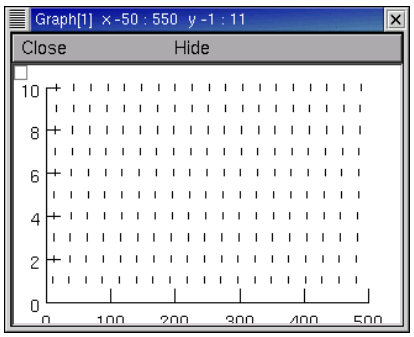


A: With all synaptic weights 0, cell firing is asynchronous and uncorrelated.

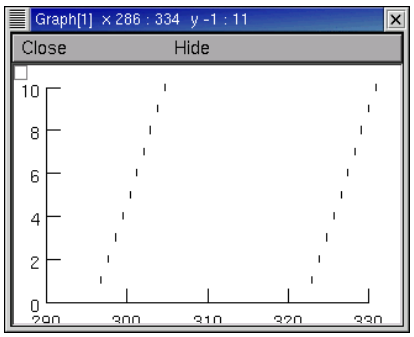
B: Mild inhibitory coupling (weight -0.2) with a delay of 1 ms silences the slowest cells and reduces the firing rates of the others. There is a suggestion of spike clustering, but no obvious synchrony or strong correlation.



C: Increasing synaptic delay to 8 ms allows the slowest cells to escape from inhibition and results in strong correlation.



D: Close examination reveals that spikes are not synchronous, but lag progressively across the population with increasing natural ISI.



References

Destexhe, A., Mainen, Z.F., and Sejnowski, T.J. An efficient method for computing synaptic conductances based on a kinetic model of receptor binding. *Neural Computation* 6:14-18, 1994.

Destexhe, A., McCormick, D.A., and Sejnowski, T.J. A model for 8-10 Hz spindling in interconnected thalamic relay and reticularis neurons. *Biophys. J.* 65:2474-2478, 1993.

Hines, M. A program for simulation of nerve equations with branching geometries. *Int. J. Bio-Med. Comput.* 24:55-68, 1989.

Hines, M. NEURON--a program for simulation of nerve equations. In: *Neural Systems: Analysis and Modeling*, edited by F. Eeckman. Norwell, MA: Kluwer, 1993, p. 127-136.

Hines, M. and Carnevale, N.T. Computer modeling methods for neurons. In: *The Handbook of Brain Theory and Neural Networks*, edited by M.A. Arbib. Cambridge, MA: MIT Press, 1995, p. 226-230.

Hines, M.L. and Carnevale, N.T. Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Computation* 12:995-1007, 2000.

Lytton, W.W. Optimizing synaptic conductance calculation for network simulations. *Neural Computation* 8:501-509, 1996.

Lytton, W.W., Contreras, D., Destexhe, A., and Steriade, M. Dynamic interactions determine partial thalamic quiescence in a computer network model of spike-and-wave seizures. *J. Neurophysiol.* 77:1679-1696, 1997.

Maass, W. and Bishop, C.M., eds. *Pulsed Neural Networks*. Cambridge, MA: MIT Press, 1999.

Riecke, F., Warland, D., de Ruyter van Steveninck, R., and Bialek, W. *Spikes: Exploring the Neural Code*. Cambridge, MA: MIT Press, 1997.

Sohal, V.S., Huntsman, M.M., and Huguenard, J.R. Reciprocal inhibitory connections regulate the spatiotemporal properties of intrathalamic oscillations. *J. Neurosci.* 20:1735-1745, 2000.

Web site retrieved 11/8/2004. NEURON Tutorial by Andrew Gillies and David Sterratt. http://www.anc.ed.ac.uk/school/neuron/

Chapter 11 Index

Chapter 12

hoc, NEURON's interpreter

Much of the flexibility of NEURON is due to its use of a built-in interpreter, called `hoc` (pronounced "hoak"), for defining the anatomical and biophysical properties of models of neurons and neuronal networks, controlling simulations, and creating a graphical user interface. In this chapter we present a survey of `hoc` and how it is used in NEURON. Readers who seek the most up-to-date list of `hoc` keywords and documentation of syntax are referred to the online Programmer's Reference (see link at `http://www.neuron.yale.edu/neuron/docs/docs.html`). This can also be downloaded as a pkzip archive for convenient offline viewing with any WWW browser. The standard distribution for MSWindows includes a copy of the Programmer's Reference which is current as of the date of the NEURON executable that it accompanies (see the "Documentation" item in the NEURON program group).

NEURON's `hoc` is based on the floating point calculator by the same name that was developed by Kernighan and Pike (1984). The original `hoc` has a C-like syntax and is very similar to the `bc` calculator. The latest implementation of `hoc` in NEURON contains many enhancements and extensions beyond its original incarnation, both in added functions and additions to the syntax. Despite these enhancements, for the most part

programs written for versions as far back as 2.x will work correctly with the most recent release of NEURON.

One important addition to hoc is an object-oriented syntax, which first appeared in version 3 of NEURON. Although it lacks inheritance, hoc can be used to implement abstract data types and encapsulation of data (see **Chapter 13**). Other extensions include functions that are specific to the domain of neural simulations, and functions that implement a graphical user interface. Also, the user can build customized hoc interpreters that incorporate special functions and variables which can be called and accessed interactively. As a result of these extensions, hoc in NEURON has become a powerful language for implementing and exercising models.

NEURON simulations are not subject to the performance penalty often associated with interpreted (as opposed to compiled) languages because computationally intensive tasks are carried out by highly efficient, precompiled code. Some of these tasks are related to integration of the cable equation, and others are involved in the emulation of biological mechanisms that generate and regulate chemical and electrical signals.

In this context, several important facts bear mention. First, a large part of what constitutes the NEURON simulation environment is actually written in hoc. This includes the standard run system (an extensive library of functions for initializing and controlling simulations--see **Chapters 7** and **8**), and almost the entire suite of GUI tools (the sole exception being the Print & File Window Manager, which is implemented in C). Second, the GUI tools for building models of cells and networks (which are, of course, all written in hoc) actually work by constructing hoc programs.

Finally, and perhaps most important, all of the hoc code that defines the standard run system and GUI tools is provided in plain text files ("hoc libraries") that accompany the standard distribution of NEURON. Under UNIX/Linux these are located in `nrn-x.x/share/nrn/lib/hoc/`, and in MSWindows they are in `c:\nrnxx\lib\hoc\`. Users can readily review the implementational details of the functions and procedures that are defined in the standard libraries, and, if necessary, modify and replace them. Since hoc is an interpreter, it is easy to make such changes without having to alter the actual files that contain the standard libraries themselves. Instead, just write hoc code that defines functions or procedures with the same names as the ones that are to be replaced, and put this in a new file. The only caveat is to be sure to load the alternatives *after* the standard library. For example, to replace the `init()` procedure (see **Examples of custom initializations** in **Chapter 8**), the text of the new procedure should occur sometime after the statement `load_file("nrngui.hoc")`. If the new definition is read prior to loading the library version, the library version will overwrite the user version instead of the other way around.

The interpreter

The hoc interpreter has served as the general input/output module in many kinds of applications, and as such is directly executed under many different names, but we will confine our attention to its use in NEURON. The simplest interface between hoc and domain-specific problems consists of a set of functions and variables that are callable from hoc. This was the level of implementation of the the original CABLE program

(NEURON version 1). NEURON version 2 broke from this style by introducing neuron-specific syntax into the interpreter itself. This allowed users to specify cellular properties at a level of discourse more appropriate to neurons, and helped relieve the confusion and reduce the mental energy required to constantly shift between high level neural concepts and their low level representation in the computer. NEURON version 3 added object syntax to allow much better structuring of the conceptual pieces that must be assembled in order to build and use a model.

Installing NEURON under UNIX or Linux results in the construction of several programs, but the principal one that we are concerned with in this book is `nrniv`, which is located in `nrn/i686/bin`. This is the main executable, which contains the `hoc` interpreter with all of its extensions. Since the bulk of the code needed by NEURON is in shared libraries, `nrniv` and the various "special" executables created by `nrnivmodl` (see **Adding new mechanisms to the interpreter** below) are very small.

Under Linux, `nrniv` can add new mechanisms and functions to `hoc` by dynamically loading shared objects that have been compiled from model description files (see **Adding new mechanisms to the interpreter**; also see **Chapter 9**). For example, the demonstration program that comes with NEURON is started by executing `neurondemo`. This is actually a shell script that starts `nrniv` with a command line that makes it load a shared object that contains additional biophysical mechanisms. Under non-Linux UNIX there is great variation in how, or even if it is possible, to dynamically load shared objects. Therefore in those environments `neurondemo` is a complete duplicate of `nrniv` plus the extra mechanisms needed by the demonstration.

Under MSWindows, the program that corresponds to `nrniv` is `nrniv.exe`. There is also a program called `neuron.exe`, which is a short stub that starts a Cygwin terminal window (see `http://cygwin.com/`) and then runs `nrniv.exe` in that window. It is `neuron.exe` that is the target of icons and shortcuts used to start NEURON. As with the Linux version, `nrniv.exe` can load new mechanisms dynamically (see next section).

Adding new mechanisms to the interpreter

To add new mechanisms, you first write a specification of the mechanism properties in the NMODL language (see **Chapter 9**), and then you compile it. To compile under UNIX and Linux, you execute the shell script `nrnivmodl`, which is located in `nrn/i686/bin`. Most often, `nrnivmodl` is called with no file name arguments, which results in compilation of all "`mod` files" in the current working directory (i.e. files that have the suffix `.mod`).

It can also be called with one or more file name arguments, e.g.

```
nrnivmodl file1 file2 . . .
```

compiles the model descriptions defined in `file1.mod`, `file2.mod`, etc.. Regardless of how `nrnivmodl` is invoked, the first step in the process is translation of the model descriptions from NMODL into C by the `nocmodl` translator.

Under Linux, the end result is a shared object located in a subdirectory `.i686/.libs` of the current working directory, as well as a shell script called `special`

in the `.i686` subdirectory that starts `nrniv` and makes it load the shared object. Under non-Linux UNIX, the result is a complete executable called `special`.

The MSWindows version of `nrnivmodl` is called `mknrndll`. It compiles and links the models into a dynamically loadable library called `nrnmech.dll`. `neuron.exe` automatically looks in the current working directory for a `nrnmech.dll` file, and if one exists, loads it into memory and makes the mechanisms available to the interpreter. More than one `dll` file can be loaded by listing them after the `-dll` argument to `neuron.exe` when it is run.

The stand-alone interpreter

The rest of this chapter describes general aspects of the interpreter that are common to all applications that contain it. Although for concreteness we use `nrniv` or `neuron.exe`, all the examples and fragments can be typed to any program that contains the interpreter, such as `oc`.

Starting and exiting the interpreter

Under UNIX and Linux, `hoc` is started by typing the program name in a terminal window

```
nrniv [filenames] [-]
```

where the brackets indicate optional elements. When there are no file name arguments, `hoc` takes its commands from standard input and prints its results to standard output. With file name arguments, the files are read in turn and the commands executed. After

the last file is executed, hoc exits. The - signals that commands are to be taken from standard input until an EOT character (^D) is encountered. One can also exit by executing quit().

When starting hoc with arguments it is easy to forget the final - and be surprised when the program quickly exits. Generally the - is omitted only when running the interpreter in batch mode under control of a shell script.

With the MSWindows version (neuron.exe), omitting the trailing - does not cause the program to exit. This makes it more convenient to attach neuron.exe to hoc files so that one can start the program and read a hoc file by merely clicking on the file's name in a file manager such as Windows Explorer. Also, neuron.exe starts a Cygwin terminal window into which one can type hoc commands. Exiting can be done by typing ^D or quit() at the interpreter's oc> prompt. If the NEURON Main Menu is present, one can also exit by selecting File / Quit; this works under all operating systems.

On startup, NEURON prints a banner that reports the current version and last change date.

```
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 oc> prompt at the beginning of a line means the interpreter is waiting for a command. This is sometimes called "immediate mode" to signify that commands are evaluated and executed (if valid) immediately, as shown in the following listing (user entries are **bold** while the interpreter's output is plain).


```
oc>2
        2
oc>1+2
        3
oc>x=2
first instance of x
oc>x
        2
oc>x*x
        4
oc>
```

The interpreter has a "history function" that allows scrolling through previous commands by pressing the keyboard's up and down arrow keys. This facilitates repeating prior commands, with or without modification. For example, in

```
oc>proc foo() { print x^3 }
oc>foo()
8
oc>proc foo() { print x^4 }
oc>foo()
16
oc>
```

line 1 defines a new procedure that prints the value of the cube of `x`, line 2 calls this procedure, and line 3 shows the numeric result. The fourth line was created by pressing the up arrow key twice, to recall the first line. Then the left arrow key was pressed twice to move the editing cursor (blinking vertical line on the monitor) just to the right of the `3`. At this point, pressing the backspace key deleted the `3`, and pressing the numeral `4` on the keyboard changed the exponent to a `4`. Finally the return ("enter") key was pressed, and the interpreter responded with an `oc>` prompt. Now typing the command `foo()` produced a new numeric result.

In immediate mode, each statement must be contained in a single line. Very long statements can be assembled by using the continuation character \ (backslash) to terminate all but the last line. Thus in


```
oc>proc foo() { print x^4 \
oc>, x }
oc>foo()
16 2
oc>
```

the interpreter merges the first and second lines into the single line

```
proc foo() { print x^4 , x }
```

Quoted strings that are constructed with continuation characters have a limit of 256 characters, and each continuation character becomes an embedded newline (line break).

Error handling

This is one of many areas where hoc falls short. Debugging large programs is difficult, so it is best to practice modular programming, breaking code into short procedures and functions.

hoc is implemented as a stack machine. This means that commands are first parsed into a more efficient stack machine representation, and subsequently the stack machine is interpreted.

Errors found during parsing are called parse errors. These range from invalid syntax

```
oc>1++1
nrniv: parse error near line 3
1++1
   ^
oc>
```

to the use of undefined names

```
oc>print x[5], "hello"
nrniv: x not an array variable near line 9
print x[5], "hello"
           ^
```


Such errors are usually easy to fix since they stop the parser immediately, and the error message, which always refers to a symptom, generally points to the cause. Error messages specify the current line number of the file being interpreted and print the line along with a carat pointing to the location where the parser failed. This is usually an important clue, but failure may not occur until several tokens after the actual mistake. One common parse error in apparently well-formed statements results from using a name of the wrong type, e.g. specifying a string where a scalar variable is required.

Errors during interpretation of the stack machine are called run-time errors:

```
oc>sqrt(-1)
sqrt: DOMAIN error
nrniv: sqrt argument out of domain near line 5
sqrt(-1)
        ^
```

Generally, run-time error messages are more pertinent to that actual problem than are syntax error messages, although logic errors can be very difficult to diagnose. These errors usually occur within a function, and the error message prints the call chain

```
oc>proc p() {execute("sqrt(-1)")}
oc>p()
sqrt: DOMAIN error
nrniv: sqrt argument out of domain near line 8
{sqrt(-1)}
          ^
       execute("sqrt(-1)")
     p()
nrniv: execute error: sqrt(-1) near line 8
^
oc>
```

Unfortunately there is no trace facility to help debug run-time errors, and the line number is of no help at all because it refers to the last line that was parsed, instead of the location of the offending statement.

Interpretation of a hoc program may be interrupted by typing one or two ^C at the terminal. For example if the interpreter is in an infinite loop, as in

```
oc>while(1) {}
```

a *single* ^C will stop it

```
^Cnrniv: interrupted near line 2
while(1) {}

oc>
```

Generally one ^C is preferred, because this allows the interpreter to reach a safe place before it halts execution. Two ^C will interrupt the interpreter immediately, even if it is in the middle of updating an internal data structure. There are two situations in which the second ^C may be necessary:

1. if the program is waiting inside a system call, e.g. waiting for console input.
2. if the program is executing a compiled function that is taking so long that program control doesn't reach a known safe place in a reasonable time.

Syntax

Names

A name is a string that starts with an alpha character and contains fewer than 100 alphanumeric characters or the underscore _. A user-created name can be associated with any one of the following:

global scalar (available to all procedures/functions)

local scalar (created/destroyed on procedure entry/exit)

array

string

function or procedure

template (class or type)

object reference

User-created names must not conflict with keywords or built-in functions. Names have global scope except if a `local` declaration is used to create a local scalar within a procedure or function, or when the name is declared within a template (i.e. class definition, although one then speaks of visibility instead of scope, and the distinction is between public and private).

Keywords

The `hoc` interpreter in the current version of NEURON has many keywords that have been added over the years. It is helpful to have a general idea of what these are useful for, and specific knowledge of where they are declared. This first table presents the most basic keywords, built-in constants, and functions of the `hoc` interpreter with object extensions and elementary functionality for neuronal modeling; the authoritative list is in `nrn-x.x/src/oc/hoc_init.c`.

General declaration

```
proc            func            local
double          strdef          iterator
eqn             depvar
```

Flow control

```
return          break           stop    continue
if              else            for     while
iterator
```


Built-in constants

PI	E	GAMMA	DEG
PHI	FARADAY	R	

Built-in variables

float_epsilon	hoc_ac_

Built-in functions

sin	cos	atan	
log	log10	exp	sqrt
int	abs	erf	erfc
use_mcell_ran4	mcell_ran4	mcell_ran4_init	
variable_domain	units		
prmat	solve	eqinit	
sred	xred		
chdir	getcwd	neuronhome	
ropen	wopen	xopen	
load_proc	load_func	load_template	
load_file	load_java		
getstr	strcmp		
printf	fprint	fscan	
ivoc_style			
save_session	print_session		
xpanel	xcheckbox		
xbutton	xstatebutton	xradiobutton	
xmenu	xlabel	xvarlabel	xslider
xvalue	xpvalue	xfixedvalue	
doEvents	doNotify		
numarg	symbols		
object_id	object_push	object_pop	
allobjectvars	allobjexts	name_declared	
boolean_dialog	continue_dialog	string_dialog	
pwman_place	startsw	stopsw	
execute	execute1		
machine_name	saveaudit	retrieveaudit	
show_errmess_always	coredump_on_error		
checkpoint	system	quit	

Miscellaneous

print	read	delete
em	debug	

Object-oriented

```
begintemplate       endtemplate
public              external
objectvar           objref          new
```

Neuron-specific

```
create              connect
access              setpointer
insert              uninsert
forall              ifsec           forsec    secname
```

The following functions and variables are specific to modeling neurons; the authoritative list of these is in `nrn-x.x/src/nrnoc/neuron.h`.

Variables

```
t                   dt              secondorder       stoprun
celsius             diam_changed
```

Functions

```
pt3dclear           pt3dadd         p3dconst
x3d                 y3d             z3d               diam3d
n3d                 arc3d
define_shape
spine3d             setSpineArea    getSpineArea
initnrn             distance        area
topology            ri
issection           ismembrane      sectionname       psection
disconnect          delete_section
pop_section         push_section
this_section        this_node
parent_section      parent_node     parent_connection
section_orientation
ion_style           nernst          ghk
finitialize         fadvance
batch_run           batch_save
fit_praxis          attr_praxis
stop_praxis         pval_praxis
```

Mechanism types and variables are defined in `nrn-x.x/src/nrnoc` by `capac.c`, `extcelln.c`, `hh.mod`, and `pas.mod`. This directory also contains several `mod` files that

define neuron-specific point process classes such as `IClamp`, `SEClamp`, and `AlphaSynapse`.

There are also several other built-in object classes, including neuron-specific examples like `SectionList`, `SectionRef`, and `Shape`, and more generic classes such as `List`, `Graph`, `HBox`, `File`, `Random`, and `Vector`. The Programmer's Reference (see link at `http://www.neuron.yale.edu/neuron/docs/docs.html`)

Variables

Double precision variables are defined when a name is assigned a value in an assignment expression, e.g.

```
var = 2
```

Such scalars are available to all interpreted procedures and functions, i.e. they have global scope.

There are several built-in variables that should be treated as constants:

`FARADAY`	coulombs/mole
`R`	molar gas constant, joules/mole/deg-K
`DEG`	`180/PI`, i.e. degrees per radian
`E`	base of natural logarithms
`GAMMA`	Euler constant
`PHI`	golden ratio
`PI`	circular transcendental number
`float_epsilon`	resolution for logical comparisons and `int()`

Arbitrarily dimensioned arrays are declared with the `double` keyword, as in

```
double vector[10], array[5][6], cube[first][second][third]
```

Array elements are initialized to 0. Array indices are truncated to integers and run from 0 to the declared value minus 1. When an array name is used without an index, the index is assumed to be 0. Arrays can be dynamically re-dimensioned within procedures.

String variables are declared with the `strdef` keyword, e.g.

```
strdef st1, st2
```

Assignments are made to string variables, as in

```
st1 = "this is a string"
```

String variables may be used in any context that requires a string, but no operations, such as addition of strings, are available (but see `sprint()` below).

After a name has been defined as a scalar, string, or array, it cannot be changed to another type. The `double` and `strdef` keywords can appear within a compound statement and are useful for throwing away previous data and reallocating space. However the names must originally have been declared outside any `func` or `proc` before they can be redeclared (as the same type) in a procedure. These restrictions also apply to object references (`objref`s--see **Declaring an object reference** in **Chapter 13**).

Expressions

The arithmetic result of an expression is immediately typed on standard output unless the expression is embedded in a statement or is an assignment expression. Thus

```
2*5
```


typed at the keyboard prints

```
10
```

and

```
sqrt(4)
```

yields

```
2
```

The operators used in expressions are, in order of precedence from high to low,

`()`	function call
`^`	exponentiation (right to left precedence)
`- !`	unaryminus, logical negation ("not")
`* / %`	multiplication, division, remainder
`+ -`	addition, subtraction
`> >= < <= != ==`	logical comparison
`&&`	logical AND
`||`	logical OR
`=`	assignment (right to left precedence)

Logical expressions are valued 1.0 (TRUE) or 0.0 (FALSE), and a nonzero value is treated as TRUE. The remainder `a%b` is in the range $0 \le$ `a%b` $<$ `b` and can be thought of as the value that results from repeatedly subtracting *or* adding `b` until the result is in the range [0, `b`). This differs from the C syntax in which `(-1)%5` is -1. For us, `(-1)%5` is 4.

Logical comparisons of real values are inherently ambiguous due to roundoff error. Roundoff can also be a problem when computing integers from reals and indices for vectors. For this reason the built-in global variable `float_epsilon` is used for logical comparisons and computing vector indices. The constant ϵ in this table stands for `float_epsilon`, which has a default value of 10^{-11} but can be assigned a different value by the user.

hoc	math or C equivalent
`x == y`	$-\epsilon \le x - y \le \epsilon$
`x < y`	$x < y - \epsilon$
`x <= y`	$x \le y + \epsilon$
`x != y`	$x < y - \epsilon$ or $x > y + \epsilon$
`x > y`	$x > y + \epsilon$
`x >= y`	$x \ge y - \epsilon$
`int(x)`	$(int)(x + \epsilon)$
`a[x]`	$a[(int)(x + \epsilon)]$

Statements

A statement terminated with a newline is executed immediately. A group of statements separated by newlines or whitespace and enclosed in curly brackets `{}` form a compound statement, which is not executed until the closing `}` is typed. Statements typed interactively do not produce a value. An assignment is parsed by default as a statement rather than an expression, so assignments typed interactively do not print their values. Note, though, the expression

```
(a = 4)
```


would print the value

```
4
```

An expression is treated as a statement when it is within a compound statement.

Comments

Text between /* and */ is treated as a comment.

```
/* a single line comment */
/* this comment
   spans
   several lines */
```

Comments to the end of the line (single line comments) may be started by the double slash, as in

```
print PI    // this comment is limited to one line
```

Flow control

In the syntax below, `stmt` stands for either a simple or compound statement.

```
if (expr) stmt
if (expr) stmt1 else stmt2
while (expr) stmt
for (expr1; expr2; expr3) stmt
for var = expr1, expr2, expr3 stmt
for iterator_name( . . . ) stmt
```

In the `if` statement, `stmt` is executed only if `expr` evaluates to a non-zero value. The `else` form of the `if` statement executes `stmt1` when `expr` evaluates to a non-zero (TRUE) value; otherwise, it executes `stmt2`.

The `while` statement is a looping construct that repeatedly executes `stmt` as long as `expr` is TRUE. The `expr` is evaluated prior to each execution of `stmt`, so if `expr` is 0 on the first pass, `stmt` will not be executed even once.

The general form of the `for` statement is executed as follows: The first `expr` is evaluated. As long as the second `expr` is true the `stmt` is executed. After each execution of the `stmt`, the third `expr` is evaluated.

The short form of the `for` statement is similar to the `DO` loop of FORTRAN but is often more convenient to type. However, it is very restrictive in that the increment can only be unity. If `expr2` is less than `expr1` the `stmt` will not be executed even once. Also the expressions are evaluated once at the beginning of the `for` loop and not reevaluated.

The iterator form of the `for` statement is an object-oriented construct that separates the idea of iteration over a set of items from the idea of what work is to be performed on each item. As such, it is most useful for dealing with objects that are collections of other objects. It is also useful whenever iteration over a set of items has a nontrivial mapping to a sequence of numbers. As a concrete example of this, let us define an iterator called `case`. To do this, we use the `hoc` keywords `iterator` and `iterator_statement`, e.g. like this

```
iterator case() {local i
  for i = 2, numarg() {
    $&1 = $i
    iterator_statement
  }
}
```

It is easy to use this iterator to loop over small sets of unrelated integers, as in


```
for case(&x, 1, -1, 3, 25, -3) print x
```

Of course, this requires that `x` has already been used as a scalar variable (otherwise the expression `&x` will be invalid) An alternative would be the relatively tedious

```
double num[5]
num[0] = 1
num[1] = -1
num[2] = 3
num[3] = 25
num[4] = -3
for i = 0, 4 {
   x = num[i]
   print x
}
```

We should point out that `iterator case()` is already included in `stdlib.hoc` (in `nrn-x.x/share/lib/hoc/` (UNIX/Linux) or `c:\nrnxx\lib\hoc\` (MSWindows)). This is automatically available after `nrngui.hoc` has been loaded.

These statements are used to modify the normal flow of control:

`break`	Exit from the enclosing `while` or `for` loop.
`continue`	Jump to end of the enclosing `while` or `for`.
`return`	Exit from the enclosing procedure.
`return expr`	Exit from the enclosing function.
`stop`	Exit to the top level of the interpreter.
`quit()`	Exit from the interpreter.

Functions and procedures

The definition syntax is

```
func name() {stmt}
proc name() {stmt}
```


Functions must return a value via a

```
return expr
```

statement. Procedures do not return a value. As a trivial example of a function definition, consider

```
func three() {
  return 3
}
```

This defines the function `three()` which returns a fixed numeric value. Typing the name of this function at the `oc>` prompt will cause its returned value to be printed.

```
oc>three()
   3
oc>
```

Notice the recommended placement of `{}`. The opening `{` must appear on the same line as the statement to which it is a part. This also applies to conditional statements. The closing `}` is free form, but clarity is best served if it is placed directly under the beginning of the statement it closes and interior statements are indented.

Arguments

Scalars, strings, and objects can be passed as arguments to functions and procedures. Arguments are retrieved positionally, e.g.

```
func quotient() {
  return $1/$2
}
```

defines the function `quotient()` which expects two scalar arguments. The `$1` and `$2` inside the function refer to the first and second arguments, respectively.

Formally, an argument starts with the letter `$` followed by an optional `&` (the "pointer operator") to refer to a scalar pointer, followed by an optional `s` or `o` that signifies a string or object reference, followed by an integer. Thus a string argument in the first position would be known as `$s1`, while an object argument in the third position would be `$o3`.

For example,

```
proc printerr(){
  print "Error ", $1, "-- ", $s2
}
```

defines a procedure that expects a scalar for its first argument and a string for its second argument. If we invoke this procedure with the statement `printerr(29, "too many channels")`, it will print the message `Error 29 : too many channels`.

There is also a "symbolic positional syntax" which uses the variable `i` in place of the positional constant to denote which argument is to be retrieved, e.g. if `i` equals 2, then `$i` and `$2` refer to the same argument. The value of `i` must be in the range [1, `numarg()`], where `numarg()` is a built-in function that returns the number of arguments to a user-written function. This usage literally requires the symbol `$i`; `$` plus any other letter (e.g. `$j` or `$x`) will not work. Furthermore, `i` must be declared `local` to the function or procedure.

The function `numarg()` can be called inside a user-written function or procedure to obtain the number of arguments. Thus if we declare

```
proc countargs(){
  print "Number of arguments is ", numarg()
}
```


and then execute `countargs(x, sin(0.1), 9)`, where `x` is a scalar that we have defined previously, NEURON's interpreter will print `Number of arguments is 3`. Generally `numarg()` is used in procedures and functions that employ symbolic positional syntax, as in

```
proc printargs() { local i
  for i = 1, numarg() print $i
}
```

If we execute `printargs(PI, -4, sqrt(5))`, the interpreter will respond by printing

```
3.1415927
-4
2.236068
```

Similarly, we could define a function

```
proc printstrs() { local i
  for i = 1, numarg() print $si
}
```

and then execute `printstrs("foo", "faugh", "fap")` to get the printed output

```
foo
faugh
fap
```

Call by reference vs. call by value

Scalar arguments use call by value so the variable in the calling statement cannot be changed. If the calling statement has a & prepended to the variable, that variable is passed by reference and must be retrieved with the syntax `$&1`, `$&2`, etc..

If the variable passed by reference is a one-dimensional array (i.e. a double), then `$&1` refers to its first (0th) element and the `j`th element is denoted `$&1[j-1]`. Be warned that there is ***no*** array bounds checking, and the array is treated as being one-dimensional.

A scalar or array reference may be passed to another procedure with `&$&1`. To save a scalar reference, use the `Pointer` class.

Arguments of type `strdef` and `objref` use call by reference, so the calling value may be changed by the called `func` or `proc`. Objects are discussed in **Chapter 13**.

Local variables

Local variables maintained on a stack can be defined with the `local` statement. The `local` statement must be the first statement in the function and on the same line as the `proc` statement. For example, in

```
proc squares() { local i, j, k /* print squares up to arg */
   for (i=1; i <= $1; i=i+1) print i*i
}
```

declaring `i`, `j`, and `k` to be `local` insures that this procedure does not affect any previously defined global variables with these names.

Recursive functions

User defined functions can be used in any expression, so functions can be called recursively. For example, the factorial function can be defined as

```
func fac() {
  if ($1 == 0) {
    return 1
  } else {
    return fac($1-1)*$1
  }
}
```

and the call

```
fac(3)
```


would produce

```
6
```

It would be a user error to call this function with a negative or non-integer argument. Besides the fact that the algorithm is numerical nonsense for those values, in theory the function would never return since the recursive argument would never be 0. Actually, after some time the stack frame list would overflow and an error message would be printed, as in

```
oc>fac(-1)
nrnoc: fac call nested too deeply near line 10
fac(-1)
        ^
        fac(-99)
      fac(-98)
    fac(-97)
  fac(-96)
and others
oc>
```

Input and output

The following describes simple text-based input and output. User interaction is better performed with the graphical interface, and dealing with multiple files requires use of the `File` class.

Standard `hoc` supplied `read()` and `print`, which use standard input and output, respectively. Their use is illustrated by this example

```
while (read(x)) {
  print "value is ", x
}
```


The return value of `read()` is 1 if a value was read, and 0 if there was an error or end of file (EOF). The `print` statement takes a comma-separated list of arguments that may be strings or variables. A newline is printed at the end.

For greater flexibility, the following built-in functions are also available.

```
printf("format string", arg1, arg2, . . . )
```

`printf()` is compatible with the standard C library function of the same name. It allows `f`, `g`, `d`, `o`, and `x` formats for scalar arguments, and the `s` format for strings. All the `%` specifications for field width apply.

```
fprint("format string", arg1, arg2, . . . )
```

`fprint()` is similar to `printf()`, but its output goes to the file that was opened by `wopen("filename")`. Such files are closed by `wopen()` with no arguments, or by the alternative `wopen("")`. When no write file is open, `fprint()` defaults to standard output. `wopen()` returns 0 on failure of the attempted open.

```
sprint(strdef, "format string", arg1, . . . )
```

This function is very useful for building file names, and even command strings, out of other variables. For example, if data files are to be named `drat.1`, `drat.2`, etc., the names can be generated with variables in the following manner.

```
strdef filename, prefix
prefix = "rat"
num = 1
sprint(filename, "d%s.%d", prefix, num)
```

After execution of these statements the, string variable `filename` contains `drat.1`.

`fscan()`

`fscan()` returns the value read sequentially from the file that was opened by `ropen("filename")`. The file is closed by calling `ropen()` with no argument or with a different file name argument. `ropen()` returns 0 if the file could not be opened. If no read file is open, `fscan()` takes its input from standard input.

Read files must consist of whitespace- or newline-separated numbers in any meaningful format. An EOF will interrupt the program with an error message. The user can avoid this with a sentinel value as the last number in the file or by knowing how many times to call `fscan()`.

`getstr(strvar)`

`getstr()` reads the next line from the file that was opened by `ropen()`, and assigns it to the string variable argument. The trailing newline is part of the string.

`xred("prompt", default, min, max)`

`xred()` places a prompt on the standard error device along with the `default` value, and waits for input on standard input. If a newline is typed, `xred` returns the default value. If a number is typed, it is checked to see if it is in the range defined by `min` and `max`. If so, the input value is returned. If the typed number is not in the range, the user is prompted again for a number within the proper range.

`xopen("filename")`

The file called `filename` is read in and executed by `hoc`. This is useful for loading previously written procedures and functions that were left out of the command line during `hoc` invocation.

Editing

The `em` command invokes a public domain editor that is similar, if not identical, to MicroEMACS. Readers who wish to try this editor will find a description of it in **Appendix A2**. However, most users are already familiar with some other editor, and it is quite easy to transfer text files into `hoc` with `xopen()` or `load_file()`.

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.

Chapter 12 Index

Chapter 13

Object-oriented programming

Object orientation is in many ways a natural style of programming whose techniques are reinvented constantly by every programmer (Coplien 1992). Object notation consolidates these techniques so that much of the tedious programming necessary to use them is automatically handled by the interpreter. An object can be thought of as an abstract data type that is very useful in separating the idea of *what a thing does* from the details of *the way it goes about doing it*. Support for objects in hoc came late to NEURON, after the notion of cable sections, and as a consequence there are several types of variables (e.g. sections, mechanisms, range variables) that are clearly treated as objects from a conceptual point of view but grew up without a uniform syntax.

In hoc, an object is a collection of functions, procedures, and data, where the data defines the state of the object. There is just enough extra syntax in hoc to support a subset of the object-oriented programming paradigm: specifically, it supports information hiding and polymorphism, but not inheritance. Yet this subset is sufficient to greatly increase the user's ability to maintain conceptual control of complex programs. This immediately provides all the power of data structures of languages such as C or Pascal, and most of the power of modules.

Object vs. class

First let's clarify the distinction between *object* and *class*. You're close to the mark if you think of a class as a cookie cutter that cuts out objects called cookies. A class is a general type, whereas an object of the class is a specific instance of the type. The idea of a class as a template motivated the keyword that signals the definition of classes in `hoc`: one surrounds a collection of functions, procedures, and variables with the keywords `begintemplate` and `endtemplate`.

From the user's point of view it is necessary to discuss how to create and destroy objects; what is an object reference; how to call an object's methods or access its data; and how to pass objects to functions. From the programmer's point of view it is necessary to discuss how to define a class. Before we plunge into these details, a general overview of objects in `hoc` will be useful.

The object model in hoc

The object model used by `hoc` manipulates references to objects, never the objects themselves. An object reference is equivalent to a pointer, and can be regarded as a label or alias for the actual object. Thus the assignment

```
ob1 = ob2
```

means that `ob1` refers to the same object referred to by `ob2`, NOT that a new object is cloned from `ob2` and pointed to by `ob1`. Thus if `ob2`'s object contains a variable called `data` and that value is changed by the statement


```
ob2.data = 5
```

then

```
ob1.data
```

will print the value

```
5
```

It quickly becomes tedious to always talk about "the object referred to by `xxx`" so we often shorten the phrase to "`xxx`", always recalling that `xxx` is only a label for that object--in fact, `xxx` is only one of possibly many labels for the object that it points to. In the next few paragraphs we'll strictly maintain the distinction between object reference and object, but be aware that we don't always exert such discipline.

Objects and object references

Declaring an object reference

Just as it is often convenient to deal with variables that can take on different numeric values (algebra is more powerful than arithmetic), it is often convenient to deal with object references that can refer to different objects at different times. Object references are declared with

```
objref name1, name2, name3, . . .
```

The deprecated keyword `objectvar` is a synonym for `objref` that may be found in older programs. The preferred `objref` emphasizes the pointer nature of object references and is easier to type.

After an object reference has been declared, it refers to the `NULLobject` until it is associated with some other object (see below).

Once a variable has been declared to be an object reference, it cannot be redefined as a scalar, double, or string. The `objref` keyword can appear within a compound statement, but the names must originally have been declared outside any `func` or `proc` before they can be redeclared (as `objref`s) in a procedure.

Creating and destroying an object

You create an object with the `new` keyword. Thus

```
objref g
g = new Graph()
```

uses the `Graph` template to create one `Graph` object that we can refer to as `g`. We'll talk about where the templates come from later. Executing these two statements will create one graph window on the screen.

Several object references can refer to the same object. Continuing with the present example,

```
objref h
h = g
```

does not create a second graph but merely associates `h` with the same `Graph` object as `g`. The "reference count" of an object is the number of object references that point to it. We would say that this `Graph` object has a reference count of 2.

If an object is no longer referenced, i.e. when its reference count is 0, it is destroyed and the memory that held its data becomes available for any other purpose. In this example, we can break the association between `g` and the `Graph` object by redeclaring `g`

```
objref g
```

so that `g` once again points to the `NULLobject`. However, the graph will persist on our screen because it is still referenced by `h`. To get rid of the graph we have to break this final reference, e.g. with the statement

```
h = g
```

Using an object reference

The object reference `g` should be thought of as pointing to an actual object located in the computer. This object has "members" which consist of variables that describe its state, plus "methods" (functions and procedures) that do things to itself and to the outside world. Some of these members are hidden from the outside world (i.e. "private"), but others are visible ("public") and can be accessed from outside the object. The syntax for using the public members of an object employs a "dot" notation that is reminiscent of how one accesses an element of a structure in C. For example, the `Graph` class has a method called `erase()` that erases graph lines, so if `g` is an `objref` that points to a `Graph` object, the statement

Of course, in C the object reference really is a pointer, so one would use the arrow notation `a->b`. In C++, the object reference has the same syntax as a reference variable.

```
g.erase()
```

will erase the lines in the `Graph`.

Passing `objref`s (and objects) to functions

As mentioned in **Chapter 12** (see ***Arguments*** under **Functions and procedures**), `objref` arguments are passed using call by reference. This has two consequences: the called `func` or `proc` can change which object the `objref` argument points to, and also that it can change the object itself. As a rather artificial example of the first consequence, let us define a `proc` that swaps the objects that two `objref`s point to.

```
objref otmp  // so it can be used as an objref in a proc
proc oswap() {
  otmp = $o1
  $o1 = $o2
  $o2 = otmp
  objref otmp  // destroy link between otmp and $o2
}
```

Suppose `a` and `b` are `objref`s that point to a `Graph` and a `Vector`, respectively, so that

```
print "a is ", a, ", b is ", b
```

returns

```
a is Graph[0] , b is Vector[3]
```

If we call

```
oswap(a, b)
```

and then repeat

```
print "a is ", a, ", b is ", b
```

we now see

```
a is Vector[3] , b is Graph[0]
```

In other words, `oswap()` made these `objref`s point to different objects.

For an even more artificial example of the second consequence, consider

```
proc foo() {  // expects a Vector argument with size >= 2
  $o1.x[1] = PI
}
```

Suppose we declare

```
objref data
data = new Vector(3)
data.indgen()
```

which makes `data` point to a `Vector` with three elements whose values are 0,1, and 2, so that `data.printf()` returns

```
0  1  2
```

Calling `foo(data)` and then trying `data.printf()` once more gives us

```
0  3.14159 2
```

i.e. `foo()` changed the object itself. In passing we note that call by reference also applies to the rare situations in which it might be useful to pass an actual object name (as distinct from an `objref`--see **Object references vs. object names** below) to a `proc` or `func`.

Defining an object class

A new object class can be defined by writing `hoc` code that specifies the properties of the class. This code is called a template, and once the `hoc` interpreter has parsed the code in a template, the class that it defines is fixed for that session. This means that any changes to a template require exiting NEURON and restarting.

The syntax for writing a template is

```
begintemplate classname
public name1, name2, name3, . . .
external variable1, string2, function3, template4, . . .
   . . . hoc code . . .
endtemplate classname
```

where `classname` is the name of the class that the template defines. The `hoc` code can be almost anything you like, but generally it consists of declarations of variables and definitions of procedures and functions. As noted above, a function or procedure that is defined in a class is also called a *method*.

By default, every variable, `proc`, and `func` that belongs to an object will be hidden from the outside. To make something visible from the outside, you must declare that it is `public`. Inside the template you cannot refer to any *user-defined* global variables or functions except those that appear in an `external` statement. However, you can execute built-in functions such as `printf()` and `exp()`, and you can also create objects from any externally-defined template.

Direct commands

Direct commands within a template, e.g.

```
begintemplate Foo
   public a
   a = 5      // this is a direct command
endtemplate Foo
```

are executed once when the template is interpreted. This means that declarations such as `double`, `strdef`, `func`, `xopen(file)`, etc., that need to be executed only once *and not for each object* are useful as direct commands. However, direct commands such as `a = 5` are less useful, since the value of `a` is lost when an actual object is created,

because the assignment statement is not executed at that time. Thus if we create a new object of class `Foo` named `footest`

```
oc>objref footest
oc>footest = new Foo()
oc>footest.a
        0
oc>
```

we see that the value of `footest.a` is 0, not 5.

Initializing variables in an object

All variables that are declared in a template start off with a value of 0 by default. To initialize variables to something other than 0, the template must contain an `init()` procedure. This procedure will be executed automatically every time a new object is created. If `init()` appears in the `public` list, you can execute it explicitly as well. For example, if we define a new class `Foo2` as

```
begintemplate Foo2
   public init, a
   proc init() {
      a = 5
   }
endtemplate Foo2
```

and then create a new object of this class

```
oc>objref foo2test
oc>foo2test = new Foo2()
```

now we find that `foo2test.a` has the nonzero value that we wanted

```
oc>foo2test.a
        5
oc>
```

Furthermore, if we assign a different value to `footest.a`


```
oc>foo2test.a = 6
oc>foo2test.a
        6
oc>
```

we can restore the original value by invoking `foo2test.init()`

```
oc>foo2test.init()
        0
oc>foo2test.a
        5
oc>
```

Keyword names

One restriction on templates is that `hoc` keywords cannot be redefined. This is an artifact of the order in which symbol tables are searched. For an example of how this affects programming, suppose we wanted to add a method to our `Stack` class that would print the name of every object in the stack. It might seem reasonable do this by inserting

```
proc print() {local cnt, i
   cnt = list.count()
   if (cnt == 0) {
      print "stack is empty"
   } else {
      for i=0,cnt-1 print list.object(i)
   }
}
```

into the body of the template and adding `print` to the `public` statement. This would allow us to call our new method with the highly mnemonic statement `stack.print()`. But when the interpreter tried to translate this to intermediate code, it would issue the error message

```
nrniv: parse error in stack3.hoc near line 2
   public push, pop, print
                        ^
```


and we would have to change the name of the method to something else, e.g. `printnames`.

Object references vs. object names

Up to this point we have been using *object references* to refer to objects, emphasizing the difference between an object itself and what we call it. Actually, each object does have a unique name that can be used anywhere a reference to the object is used. However, these unique names are primarily intended for use by the library routines that construct NEURON's graphical interface. While it may occasionally be useful to employ these unique names in user-written code (e.g. for diagnostic or didactic purposes), this should never be done in ordinary programming. Object names are not guaranteed to be the same between different sessions of NEURON unless the sequence of creation and destruction of objects of the same type is identical. This is because the object name is defined as `classname[index]`, where the "index" is automatically incremented every time a new instance of that class is created. Index numbers are not reused after objects are deleted except when there are no existing objects of that type; then the index starts over again at 0.

The reason why unique object names are allowed at all is because some objects, such as the `PointProcessManager`, should be destroyed when their window is dismissed. This could not happen if the interpreter had an `objref` to that object, since an object is destroyed only when its reference count goes to 0. Thus the idiom is to cause the `VBox` window itself to increment the reference count for the object (and decrement it when the window is dismissed, using the `VBox`'s `ref()` or `dismiss_action()` method). Now

the hoc `objref` that holds the reference can safely discard it, and the object will not be immediately destroyed. But the consequence is that there is now no way to get to the object (or the objects it created) from the interpreter except to use the object name, e.g. there is no other way to graph one of the point process variables in the `PointProcessManager`.

An example of the didactic use of object names

The name of an object can be used in any context in which a string is expected, e.g. a `print objref` statement. For example, if we execute the statements

```
objref g, h
g = new Graph()
h = g
```

then we see a graph on the computer screen, and

```
print g, h
```

returns

```
Graph[0] Graph[0]
```

because both `g` and `h` refer to the same `Graph` object. At this point if we type the command `print Graph[0]` we also get `Graph[0]`.

After redeclaring `g`

```
objref g
```

we find that `print g, h` gives us

```
NULLobject Graph[0]
```


Since one object reference (h) still points to `Graph[0]`, the graph is still visible, and `print Graph[0]` still produces `Graph[0]`.

Now asserting

```
h = g
```

discards the last reference to `Graph[0]`, destroying this object. Consequently the graph disappears from the screen, and `print g, h` produces

```
NULLobject NULLobject
```

Any lingering doubts concerning the fate of `Graph[0]` are dispelled when we find that `print Graph[0]` generates the message

```
nrniv: Object ID doesn't exist: Graph[0]
 near line 11
print Graph[0]
              ^
```

Using objects to solve programming problems

Dealing with collections or sets

Most, if not all, nontrivial programming problems seem to involve the notion of a set or collection of objects. `hoc` can represent the concept of "more than one" in several ways, but the workhorses are the array of objects and the list of objects. The array is the most efficient but requires a prior knowledge of the number of objects to be stored. The list can store any number of objects at any time; this fact makes `List` the most often used class.

Array of objects

Storage for an array of objects is declared with

```
objref array[size]
```

Only rarely is the size known when the program is written, so it is common practice to separate the declaration from the size definition, specifying the latter just after the point in execution when the size is finally known, as in

```
objref array[1] // size must be declared even if wrong
proc set_size() {
   objref array[$1]
}
```

After the size is set, it can no longer be changed without redeclaring the entire array, which discards the references to any objects referenced by its previous incarnation. When an array is declared or redeclared, all of its elements reference the `NULLobject`.

An array is a random access object because its individual elements can easily be retrieved in any sequence, just specifying the corresponding index. For example an array of five graphs can be created with

```
objref graphs[5]
for i=0, 4 { graphs[i] = new Graph() }
```

The internal name of each item in the array can be printed in reverse order with

```
for (i=4; i >= 0; i -= 1) { print graphs[i] }
```

Suppose we wanted to destroy the third (index = 2) graph. We can't simply say

```
objref graphs[2]
```


because this would discard the entire array, throwing away all of our graphs and creating a new array whose elements all point to the `NULLobject`. Instead, the way to make the reference count for the third graph become 0 is

```
objref nil       // nil points to NULLobject
graphs[2] = nil  // and now so does graphs[2]
```

Example: emulating an "array of strings"

Even very simple templates have their uses. There is no such thing in `hoc` as an array of strings, but consider

```
begintemplate String
   public s
   strdef s
endtemplate String
```

Now an array of objects can be used to get the functionality of an array of strings.

```
objref s[3]
for i=0,2 s[i] = new String()  // they all start out empty
s[0].s = "hello"
s[2].s = "goodbye"
```

It is important to realize that there is no conflict between the use of `s` as the name of a `strdef` inside the template and the use of `s` as the name of an object reference outside the template.

We must mention that NEURON comes with a very similar implementation of the `String` class (see `stdlib.hoc` in `nrn-x.x/share/lib/hoc/` (UNIX/Linux) or `c:\nrnxx\lib\hoc\` (MSWindows)). This is automatically available after `nrngui.hoc` has been loaded.

List of objects

A list of objects uses the `List` class

```
objref list
list = new List()
```

Objects are added to the list with the `append()` method, as in

```
for i=0, 4 { list.append(new Graph()) }
```

Notice that we do not have to know how many items will be added to the list before we start adding them. One can print the names of the objects in a list with the statement,

```
for i=0, list.count - 1 { print list.object(i) }
```

The `List` class's `count()` method always returns the number of objects in the list, and the `object()` method returns the item.

Iteration over a list is one of the most commonly used programming idioms. This allows processing of each item in the list, as in

```
objref tobj
for i=0, list.count - 1 {
   tobj = list.object(i)
   // do something to the object referenced by tobj
}
objref tobj // only the list holds a reference to the last object
```

Notice how a temporary `objref` is employed to refer to each object in turn.

Example: a stack of objects

This template defines a class that can be used to create stacks of objects.

```
begintemplate Stack
   public push, pop
   objref list
```



```
    proc init() {
       list = new List()
    }

    proc push() {
       list.append($o1)
    }

    proc pop() {local cnt
       cnt = list.count()
       if (cnt == 0) {
          print "stack underflow"
          stop
       }
       $o1 = list.object(cnt-1)
       list.remove(cnt-1)
    }
endtemplate Stack
```

After `hoc` parses this template, the statements

```
objref stack
stack = new Stack()
```

create an object that functions as a stack. At the time this new object is created, its `init()` procedure is executed, which creates an empty list for use by the `push()` and `pop()` procedures. Notice that `push()` and `pop()` are public, but the internal list is private.

Suppose we already have three `Graph` objects `g[0]`, `g[1]`, and `g[2]` (see **Creating an object** under **Objects and object references** above). Then `stack.push(g[1])` adds a reference to the second `Graph` at the end of the `Stack` object's internal list. `stack.pop(g[2])` would cause `g[2]` to reference the same object as `g[1]` and remove it from the stack.

In this example, we have exploited an existing object class (`List`) to create a new object class (`Stack`) that can be used to hold a stack of objects of any class we like--not just objects of any of NEURON's built-in classes, but also objects of any other classes

that we might dream up in the future! Note the use of the `List` class's `count()` and `remove()` methods to find the object at the end of the list and to remove this reference from the list.

Encapsulating code

Suppose you have a `hoc` file that works perfectly all by itself (when nothing else is loaded) and does something meaningful when you type `run()` at the `oc>` prompt. Also suppose the file has no direct commands except declarations (if it does have direct commands, just collect them into an `init()` procedure). Then, if you put the these lines at the beginning of the file

```
begintemplate F1
public run
```

and this line at the end of the file

```
endtemplate F1
```

you have an object template. You can use this template to create an object and run it, like this

```
objref f1
f1 = new(F1)
f1.run()
```

and you will get identical behavior as before. What's been gained? Well, you can do this to a bunch of files and load them all together and never worry about variable or function name clashes between files because nothing (except the object templates and specific object names) is global.

Don't forget that the default initialization of variables declared in a template is 0. It is a good idea to include an `init()` procedure that uses explicit assignment statements to make sure that variables will start off with the proper values. It is possible to declare a variable with an assignment statement in procedure `P1`, and then use it in a `public` procedure `P2`, but be mindful of the possibility that someone may execute `P2` before executing `P1`. If this happens, the variable will have a value of 0.

Polymorphism and inheritance

A language supports polymorphism when it automatically does the right thing whether a function is called on the base class or on an object of a subclass. Since an object reference can refer to any type of object, `hoc`'s object model is polymorphic. Thus, if `A` and `B` are different classes but happen to have a method with the same name, e.g. `foo()`, then if `oref` refers to an instance of either `A` or `B`, we can say `oref.foo()` and the method of the particular object type will be called.

For a concrete example, suppose we have defined several different classes of objects that generate specialized graphs called `BodePlot`, `PowerSpect`, and `CrossCorr`, and that each of these classes has its own `plot()` and `erase()` method. We can easily automate plotting and erasing if we declare

```
proc plotall() { local i
  for i = 0, glist.count()-1 glist.object(i).plot()
}

proc eraseall() { local i
  for i = 0, glist.count()-1 glist.object(i).erase()
}
```


and, every time we spawn a new instance of one of these classes, we append it to a `List` object, e.g.

```
objref mygraflist
glist = new List()
  . . .
objref bp, ps, cc
bp = new BodePlot()
glist.append(bp)
ps = new PowerSpect()
glist.append(ps)
cc = new CrossCorr()
glist.append(cc)
```

Now we can take care of all of these graphs at once by invoking `plotall()` or `eraseall()`.

Inheritance allows us to define many kinds of subclasses starting from a more abstract base class. It is useful in capturing the "IS A" relationship, and is most effective when the subtype "IS A" kind of base type, i.e. whenever a program uses an object of the base type then it would also make sense if it used an object of the subtype. People often (ab)use inheritance when the IS A relationship does not hold, in order to conveniently reuse a portion of the base class. When one class is "ALMOST LIKE" another, and that other is ready and waiting to be used, it is tempting to inherit the whole behavior and replace only the parts that are different. It's best to avoid this practice and instead factor out the behavior common to both classes, placing that in a base class which can be inherited by both classes.

In `hoc`, inheritance can only be emulated by having the "subclass" instance create its "superclass" instance during initialization and supply stub methods for calling the public methods of the superclass. For example, consider the trivial `Base` class


```
begintemplate Base
public a, b
objref this

proc a() {
   printf ("inside %s.a()\n", this)
}

proc b() {
   printf("inside %s.b()\n", this)
}
endtemplate Base
```

Then the following will look like a subclass of `Base`, where we provide our own implementation of `a` and "inherit" the method `b`:

```
begintemplate Sub
public a, b
objref this, base

proc init() {
   base = new Base()
}

proc a() {
   printf("inside %s.a\n", this)
}

proc b() {
   base.b()
}

endtemplate Sub
```

References

Coplien, J.O. *Advanced C++ Programming Styles and Idioms*. Reading, MA: Addison-Wesley, 1992.

Chapter 13 Index

T

Chapter 14

How to modify NEURON itself

NEURON's extensive library of functions and graphical tools has been developed with an eye to providing those features that are most widely applicable in empirically-based neural modeling. Since they are necessarily rather generic, it is sometimes desirable to change these features or add new ones in order to meet the special needs of individual projects. This is particularly important where the graphical user interface is concerned, given the well-established role of the GUI in enhancing software utility. Here we show how to create new GUI tools and add new functions to NEURON.

A word about graphics terminology

Computer graphics literature is full of technical jargon, of which we will use only the most minute part. Think of a *scene* as being like a sheet of paper that has its own coordinate system (*scene coordinates*), and a *view* as a rectangular area on that sheet. *Model* (in the graphical sense) is just a synonym for "scene," and "model coordinates" are the same as "scene coordinates." The *screen* is simply the computer's display (CRT, flat panel, etc.), and *screen coordinates* represent actual physical locations on the display. Finally, when something is "mapped to the screen," the visible consequence is that a view of it appears on the display.

Graphical interface programming

In **Chapter 6** we noted that an iterative process of incremental revision and testing can be an effective strategy for software development. This is especially true when the task is to create a tool that interacts with the user through a graphical interface. It is rarely clear at the outset what manipulations of objects on the screen are feasible with reasonable programming effort, have obvious or easily learned meanings on first approach, and allow users to straightforwardly and rapidly reach their goals. The approach outlined in this section has proven useful in creating every one of the tools in the NEURONMainMenu suite.

As a concrete example, suppose we want to make a tool for graphically specifying the parameters of the Boltzmann function

$$y(x) = \frac{1}{e^{4k(d-x)}} \qquad \text{Eq. 14.1}$$

The half maximum of this curve is located at (d, 0.5), and the slope at this point is k.

The graphical metaphor for setting the value of d seems straightforward: just use the cursor to drag the (d, 0.5) point on the curve horizontally. By putting a small open square mark centered at (d, 0.5) on the curve, we can suggest that this is a user-selectable "control point."

How to set the value of k is less obvious. It seems quite natural to use the cursor's x coordinate for midpoint control, but simultaneously using its y coordinate to control the slope turns out to be neither intuitive nor convenient. One alternative could be to use a second control point at the 80% y value that can be dragged back and forth horizontally;

this limits the minimum slope, but it might be adequate. Another possibility is to place the slope control point at some fixed distance from the midpoint. In this case the function is always drawn so that the slope control point is on the line between the current cursor location and the midpoint. Finally, perhaps it would be better to allow the user to click on any point of the curve and drag, and have the GUI tool redraw the curve so that it follows the cursor.

But we're getting ahead of ourselves. We can always experiment with the various styles of setting the slope parameter later--the burning question now is how to even get started.

Clearly we need a canvas on which to plot the curve, and a way to get mouse button events and cursor coordinates when one clicks in the canvas and drags the cursor to a new location. We can put a `Graph` on the screen with

```
objref g
g = new Graph()
```

Now we can do some exploratory tests of how to deal with mouse events. Let's specify a procedure to handle a mouse event

```
g.menu_tool("A new mouse handler", "handle_mouse")
```

This creates a new radio style menu item labeled "A new mouse handler" that appears at the end of the primary graph menu. If this menu item is selected, then future mouse button presses in the `Graph`'s canvas will call the procedure named `handle_mouse()` with four arguments. These arguments specify the state of the left mouse button, cursor coordinates, and whether the Shift, Control, and/or Alt key is being pressed. The `handle_mouse()` procedure does not have to be defined when `menu_tool()` is

executed, but it certainly must be defined by the time it is supposed to be called. As a quick test, we can use this procedure

```
proc handle_mouse() {
   print $1, $3, $3, $4
}
```

for a concrete demonstration of mouse event handling. This is an example of how simple tests can provide a far better understanding of the meaning of a method, than we could ever gain from just reading the reference manual.

General issues

We can go quite far by trying things out one at a time and verifying that the hoc code is working the way we want. However, experience has shown that sooner or later it is becomes necessary to grapple with the following issues.

- How to encapsulate a tool so that its variables do not conflict with other tools. This tool defines the variables *k* and *d*, and it will probably also demand that we invent a lot of other names for functions and variables. Keeping all of these names localized to a particular context so that it doesn't matter if they are used in other contexts is one of the benefits of object-oriented programming.
- How to allow many instances of the same type of tool. If this tool is ever used as a part of some larger tool, e.g. a channel builder that describes steady state as a function of voltage, there will be a need for separate pairs of Boltzmann parameters for every channel state transition. This is another benefit that one gets almost for free with object-oriented programming.

- How to save a tool so the user doesn't lose data that was specified with it. The values of *k* and *d* for one of our objects may represent a great deal of thought and effort. Even the small matter of repeatedly dragging a window to a desired location and resizing it quickly becomes tedious. The user should be able to save the tool, with its present state, in a session file so that it can be re-instantiated by retrieving the session file.
- How to destroy the tool when its window is closed. We want our tool to exist only by sufferance of the user. It should be destroyed if and only if the user presses the Close button. Objects in `hoc` are reference counted, which normally means that an object stays in existence only as long as there is some object reference (`objref`) that points to it (see **Creating and destroying an object** in **Chapter 13**). Think of a reference as a label on the object: when the last label is removed the object is destroyed. If the reference issue is not properly handled, two opposite problems result. First, closing the window would not reclaim the memory used by the tool. This isn't so bad in our situation, but it could be very confusing for tools that manage, for example, a point process or even an entire cell! The second problem is that a tool instance and its window could be inadvertently destroyed even if the user hadn't pressed the window's Close button. This can happen if creating a new instance of the tool reuses the only reference to an already existing instance of the tool.

Although it may seem tedious at first, the effort of starting a tool development project with a pattern that addresses all these issues is quickly repaid. We create such a pattern in the following pages, starting by defining a new class that will be used to encapsulate the tool data and code. For the sake of concreteness, we will assume that the `hoc` code for our tool will be saved to a file named `bp.hoc`.

A pattern for defining a GUI tool template

The name we have chosen for this class is `BoltzmannParameters` (Listing 14.1). The overall style of the pattern is to first declare the names of things used in the template, and then to declare procedures and functions. The `public g` statement declares that `g` is accessible to the outside world, and `objref g` declares that its type is "reference to an object" (see **Chapter 13**). When a new instance of the `BoltzmannParameters` class is created, the `init()` procedure in this template will bring up a new `Graph`; this is merely so that creating a new `BoltzmannParameters` object will produce a visible result (Fig. 14.1).

The two statements that follow the template definition create an instance of the class. This facilitates the edit-run-diagnose cycle by ensuring that, when NEURON executes this code, we don't need to type anything into the interpreter in order to see the result of our file modifications.

As an aside, we must note that mistyping and other trivial errors are common. This is yet another reason for developing a program in stages, making only a few additions at a time. Building a tool requires many iterations of incrementally editing and adding to the `hoc` code that defines it, and then launching NEURON to test it. In a windowed desktop environment, this can be facilitated by using two windows: one for keeping the file open in an editor, and the other for running NEURON after an editor save (e.g. by double clicking on the name of the `hoc` file).


```
begintemplate BoltzmannParameters
   public g
   objref g

   proc init() {
      g = new Graph()
   }
endtemplate BoltzmannParameters

// the following lines facilitate debugging
objref b
b = new BoltzmannParameters()
```

Listing 14.1. Initial version of `bp.hoc` creates an object and produces a visible result (Fig. 14.1).

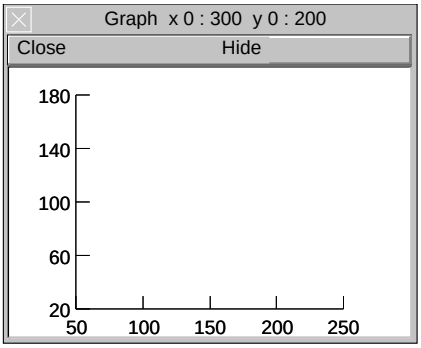


Fig. 14.1. The graph window generated by the code in Listing 14.1.

After executing the code in Listing 14.1, we can perform additional tests by entering a few simple commands into the interpreter at the `oc>` prompt:

```
oc>b
   BoltzmannParameters[0]
oc>b.g
   Graph[0]
oc>b.init()
init not a public member of BoltzmannParameters
```

The `init()` procedure is automatically called whenever a new instance of the template is created (see **Initializing variables in an object** in **Chapter 13**). In most cases it doesn't make sense to call `init()` again, so it is usually not declared public.

Enclosing the GUI tool in a single window

We quickly discover that the relationship between the window on the screen and the object `BoltzmannParameters[0]` is inappropriate. Pressing the Close button destroys the window but doesn't get rid of the `BoltzmannParameters[0]` object, because the latter is still referenced by `b`. Furthermore, the `BoltzmannParameters[0]` object is not protected from outside interference--redeclaring `b` or making it reference another object will drop `BoltzmannParameters[0]`'s reference count to 0 and destroy it. For a tool, this should only happen when one closes the window.

The current situation is that `b` references our tool instance, `b.g` references the `Graph` in our tool, but the window doesn't reference anything. We need a way for the window to reference the tool. To make this happen, we augment our pattern by enclosing the `Graph` in a `VBox` which itself references the tool, and making sure that this `VBox` is the only reference to the tool. The pattern now looks like Listing 14.2.

```
begintemplate BoltzmannParameters
   public g, box
   objref g, box, this

   proc init() {
      box = new VBox()
      box.ref(this)
      box.intercept(1)
      g = new Graph()
      box.intercept(0)
      box.map()
   }
endtemplate BoltzmannParameters

// the following lines facilitate debugging
objref tmpobj
proc makeBoltzmannParameters() {
   tmpobj = new BoltzmannParameters()
   objref tmpobj
}
makeBoltzmannParameters()
```

Listing 14.2. `bp.hoc` revised for proper management of reference count.

Between the `box.intercept(1)` and `box.intercept(0)` statements, anything that would create a window becomes arranged vertically in a `VBox` (horizontally if it was an `HBox`). When `this` is declared as an object reference, it always refers to the object that declared it. The `box.ref(this)` statement has the effect that, when the last window created by the box is closed, the reference count of the tool is decremented. If this makes the reference count equal to 0, the tool is destroyed. When boxes are nested, only the outermost (the one with an actual window) requires this idiom. The code at the end of this revised `bp.hoc` creates an instance of our tool, and then immediately redeclares `tmpobj` in order to guarantee that the only remaining reference to the tool is the one associated with the `Vbox`.

The pattern is now in a form that can be integrated into the standard NEURONMainMenu / Tools / Miscellaneous menu. We can do this with

```
NEURONMainMenu[0].miscellaneous_add("BoltzmannParameters", \
                    "makeBoltzmannParameters()")
```

Those who recall our injunction against writing code that invokes unique object names (see **Object references vs. object names** in **Chapter 13**) may be somewhat uncomfortable at the appearance of the object name `NEURONMainMenu[0]` in this statement. Such excellent retention of the lore of `hoc` is to be commended. Rest assured, however, that there is never more than one instance of the `NEURONMainMenu` class, and when such an object exists, it is *always* called `NEURONMainMenu[0]`. Hence this object name is *truly* unique, making this a situation in which it is safe to inscribe the proscribed.

Our `init()` procedure is collecting a lot of odds and ends; readability would be better served by separating the different idioms into procedures, as shown in Listing 14.3.

Also note the changes to `map()`, which add the useful feature of giving the window a title that reflects the name of the tool instance. These changes produce the window shown in Fig. 14.2.

```
strdef tstr

proc init() {
   build()
   map()
}

proc build() {
   box = new VBox()
   box.ref(this)
   box.intercept(1)
   g = new Graph()
   box.intercept(0)
}

proc map() {
   sprint(tstr, "%s", this)
   box.map(tstr)
}
```

Listing 14.3. Revision of `init()` to enhance clarity.

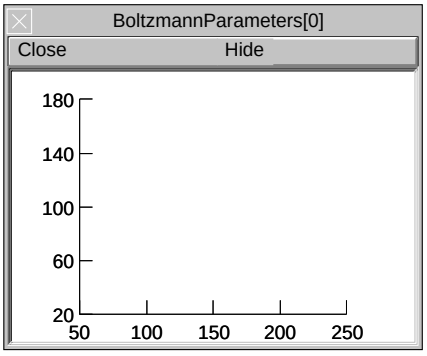


Fig. 14.2. Changes in the `map()` procedure (Listing 14.3) turn the window title into the name of the tool instance (compare with Fig. 14.1).

Saving the window to a session

The last enhancement to our basic tool pattern provides a way to save the tool in a session file so it can be recreated properly when the session file is loaded. This enhancement involves several changes to `bp.hoc` (see Listing 14.4). Before examining these, we must point out that everything in this code is independent of the future implementation of the particulars of what we want our tool to do. It just gives us a starting framework to which we can add specific functionality.

The principal change to `bp.hoc` is the addition of a `save()` procedure that can print executable `hoc` code to the session file. Most of the statements in `save()` are of the form `box.save("string")`. Since our tool has a `Graph`, it is a good idea to save its size. This can be done explicitly by building an appropriate statement, but a simple idiom is to call its `save_name()` method (see `g.save_name("ocbox_.g", 1)`) since this will also save any new views of the graph that were generated by the user. This must be done at the top level, so the `Graph` reference has to be public.

> Here "top level" refers to the interpreter dealing with global variables rather than variables that are only visible inside templates. Put another way, one can say the interpreter is executing at the top level unless it is executing code that is declared in a template or class. When the interpreter is not at the top level, it is often useful to temporarily get back to the top level with `execute("statement")`.

A related change to `bp.hoc` is the addition of `box.save("save()")` to the `build()` procedure. This designates `save()` as the procedure that will be called when the tool is saved to a session file.

Changes to `init()` and `map()` are necessitated by the fact that we want the tool to be properly mapped to the screen regardless of whether we have created it for the first time, or instead are reconstituting it from a session file. If we are creating a tool *de novo*, it should appear in a default location with default values for its various user-specifiable parameters. However, if we are recreating it from a session file, it should appear in the same location and with the same parameter values as when we saved it.

```
begintemplate BoltzmannParameters
   public g, box, map
   objref g, box, this
   strdef tstr

   proc init() {
     build()
     if (numarg() == 0) {
       map()
     }
   }

   proc build() {
     box = new VBox()
     box.save("save()")
     box.ref(this)
     box.intercept(1)
     g = new Graph()
     box.intercept(0)
   }

   proc map() {
     sprint(tstr, "%s", this)
     if (numarg() == 0) {
       box.map(tstr)
     } else {
       box.map(tstr, $2, $3, $4, $5)
     }
   }

   proc save() {
     box.save(\"load_file(\"bp.hoc\", \"BoltzmannParameters\")\n}\n{")
     box.save("ocbox_ = new BoltzmannParameters(1)")
     box.save("}\n{object_push(ocbox_)}")
     // insert tool-dependent statements here
     box.save("{object_pop()}\n{")
     g.save_name("ocbox_.g", 1)
   }
endtemplate BoltzmannParameters
```



```
// the following lines facilitate debugging
objref tmpobj
proc makeBoltzmannParameters() {
   tmpobj = new BoltzmannParameters()
   objref tmpobj
}
makeBoltzmannParameters()
```

Listing 14.4. This refinement of `bp.hoc` contains a tool that can be saved and retrieved from a session file. Despite its specific name, this "`BoltzmannParameters`" template is a *completely generic starting point* that can be customized to implement specific GUI tools.

Making this happen involves three related changes to `bp.hoc`. First, the `init()` procedure has been modified so that calling it without an argument will cause the window to be mapped immediately with its defaults. Otherwise, the mapping is deferred, so that code in the session file can specify its location etc.. The second change is to the `map()` procedure: when called with no arguments, it makes the tool appear in its default position. Alternatively `map()` can be called from the session file with arguments that specify where the tool will be drawn on the screen. This brings us to the third change, which is to add `map` to the `public` statement at the top of the template (otherwise `map()` couldn't be called from the session file).

```
load_file("nrngui.hoc")
objectvar save_window_, rvp_
objectvar scene_vector_[3]
objectvar ocbox_, ocbox_list_, scene_, scene_list_
{ocbox_list_ = new List()  scene_list_ = new List()}
{pwman_place(0,0,0)}

//Begin BoltzmannParameters[0]
{
load_file("bp.hoc", "BoltzmannParameters")
}
{
ocbox_ = new BoltzmannParameters(1)
}
{object_push(ocbox_)}
{object_pop()}
```



```
{
{
save_window_=ocbox_.g
save_window_.size(0,300,0,200)
scene_vector_[2] = save_window_
ocbox_.g = save_window_
save_window_.save_name("ocbox_.g")
}
ocbox_.map("BoltzmannParameters[0]", 211, 211, 315, 251.1)
}
objref ocbox_
//End BoltzmannParameters[0]

objectvar scene_vector_[1]
{doNotify()}
```

Listing 14.5. The contents of the session file created by executing Listing 14.4 and then saving the tool to a session.

Saving this tool to a session file by itself produces the text shown in Listing 14.5. NEURON's internal code for saving session files first prints several statements that, among other things, declare object reference variables used by various kinds of windows. The only statement that concerns us here is the declaration of `ocbox_`, which is used by boxes to map themselves to the screen as windows.

> Although `ses` files use the deprecated `objectvar`, the preferred keyword for declaring an object reference is `objref` (see **Declaring an object reference** in **Chapter 13**)

Next it prints a comment that contains the title of the window, and then it calls our `save()` procedure, which prints the lines in the `ses` file up to `save_window_`. The first of these strings is

```
load_file("bp.hoc", "BoltzmannParameters")
```

which makes sure that the code for our template is loaded, and the next is

```
ocbox_ = new BoltzmannParameters(1)
```


which uses the `ocbox_` object reference to create an instance of our tool. Note the use of an argument to prevent the `init()` procedure from mapping the tool to the screen. The `object_push()` causes succeeding statements to be executed in the context of the object. This allows access to any of the variables or functions in the object, *even if they are not public*--which is necessary in order to save the state of our object. The graph reference must be public because `object_pop()` returns to the previous context, which is normally the top level of the interpreter.

After the `save()` procedure returns, the `map()` method is called for `ocbox_` with the first argument being the title of the window, followed by four more size arguments which specify the screen location of the window. Since we commandeered `ocbox_` as the reference for our tool, our own `map()` is called instead, and we replace whatever title was used by the current authoritative instance name. Our `map()` also uses the four placement arguments if they exist (they don't if `map()` is called from `init()`).

Tool-specific development

Plotting

This tool is supposed to display a Boltzmann function, so we clearly need to define the function, declare the parameters we will be managing, and plot the function. Any names we invent will be nicely segregated so they can't conflict with names at the top level of the interpreter or names in any other tool. Thus we can safely use the natural names that first come to mind.

The Boltzmann function can go anywhere in the template.

```
func b() {
   return 1/(1 + exp(-4*k*(d - $1))
}
```

The first argument is referred to as $1 in a function body. The parameters are used here, and this is sufficient to declare them, but their default values would be 0. To give them better defaults we add

```
k = 1
d = 0
```

to the body of `init()`.

Plotting the function raises the issue of the proper domain and the number of points. We could get sophisticated here, plotting points at *x* values that nicely follow the curve, but instead we will just plot 100 line segments over a domain defined by the current graph scene size.

```
proc pl() {local i, x, x1, x2
   g.erase_all
   x1 = g.size(1)
   x2 = g.size(2)
   g.begin_line
   for i=0, 100 {
      x = x1 + i*(x2 - x1)
      g.line(x, b(x))
   }
   g.flush
}
```

This procedure starts with `erase_all` because it will eventually be called whenever `k` or `d` is changed. For now we get a drawing on the graph by calling `pl()` at the end of the `map()` procedure.

Making these changes and running `bp.hoc` produces the error message

```
nrniv: syntax error in bp.hoc near line 51
    return 1/(1 + exp(-4*k*(d - $1))
                                    ^
```

This is a parse error, which means that something is wrong with the syntax of the statement (see **Error handling** in **Chapter 12: hoc, NEURON's interpreter**). The parser often fails a few tokens after the actual error, but in this case the carat is right on target: this statement is missing a closing parenthesis.

Adding the parenthesis and trying again, we find that `beginline` was misspelled. This error occurred at run time, so the call stack was printed, which gives very helpful information about the location of the error.

```
begin_line not a public member of Graph
nrniv: Graph begin_line in bp.hoc near line 77
    makeBoltzmannParameters()
                             ^
BoltzmannParameters[0].pl()
BoltzmannParameters[0].map()
BoltzmannParameters[0].init()
makeBoltzmannParameters()
```

Another try gives

```
nrniv: exp result out of range in bp.hoc near line 77
makeBoltzmannParameters()
                         ^
```

Testing `exp(-10000)` shows there is no underflow problem with this function, and `exp(700)` is below the overflow limit. Taking this into account requires a slight elaboration of `b()`

```
func b() { local x
   x = -4*k*(d - $1)
   if (x > 700) { return 0 }
   return 1/(1 + exp(x))
}
```


Now we no longer get an "out of range" message, but the `Graph` still appears to empty, looking identical to Fig. 14.2. Invoking the item View = plot from the `Graph`'s secondary menu turns this into a featureless grey rectangle (Fig. 14.3).

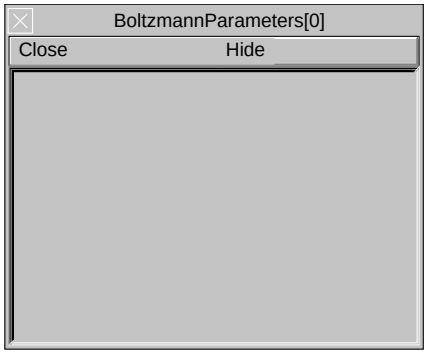


Fig. 14.3. After the numeric overflow problem is eliminated, the `Graph` seems to be empty. However, View = plot turns the window contents grey, indicating that something else is wrong.

This is puzzling at first, but after a moment's reflection we try adding

```
print $1, 1/(1 + exp(x))
```

just before the `return` statement in `func b()`, and run the code again. This diagnostic test results in NEURON printing just one pair of values

```
25 3.720076e-44
```

which is a clue to what should have been obvious: instead of one mistake, there are two.

The first error was in the way the desired *x* value was calculated in `proc pl()`. This should have read

```
x = x1 + i*(x2 - x1)/100
```


The second error was to accept the default dimensions of the `Graph`, which have *x* running from 25 to 275 so that `b()` is too small to be of interest. This can be fixed by adding

```
g.size(-5, 5, 0, 1)
```

after creating the `Graph` in the `build()` procedure.

Now the `Graph` produced by this template looks pretty good (Fig. 14.4), but the plotted curve has a slope of -1 at x = 0. This is just carelessness; maybe we fell into thinking that exponentials always have negative arguments, as if everything is a time constant. Let us strike a mutual pact never to make another mistake.

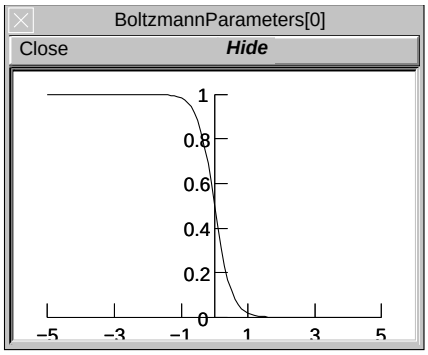


Fig. 14.4. The `Graph` after fixing the range of *x* values and specifying appropriate dimensions.

Handling events

The following lines, placed in the `build()` procedure after the `Graph` has been created, specifies how mouse events are handled.

```
g.menu_tool("Adjust", "adjust")
g.exec_menu("Adjust")
```


The `exec_menu()` selects the menu item `Adjust` as the current tool for mouse event handling. That is, press, drag, or release of the left mouse button while the mouse cursor is over the `Graph` will cause a procedure named `adjust()` to be called with arguments that specify the type of event (press, drag, or release), the scene coordinates of the cursor, and the state of the Shift, Control, and Alt keys.

```
proc adjust() {
   if ($1 == 2) { // left mouse button pressed
      adjust_ = 0
      // $2 and $3 are scene coords of mouse cursor
      if (ptdist($2, $3, d, 0.5) < 100) {
         adjust_ = 1
      } else if (ptdist($2, $3, $2, b($2)) < 100) {
         adjust_ = 2
      }
   }
   if (adjust_ == 1) {
      d = $2
      pl()
   }
   if (adjust_ == 2) {
      // keep y value within function domain
      if ($3 > 0.99) $3 = 0.99
      if ($3 < 0.01) $3 = 0.01
      // avoid singularity at x == d
      if ($2 > d || $2 < d) {
         // change k so that curve passes through
         //   cursor location
         k = log(1/$3 - 1)/(4 * (d - $2))
         pl()
      }
   }
}

func ptdist() {
   return 1
}
```

Listing 14.6. The handler for mouse events is `proc adjust()`.

The `adjust()` procedure needs to decide whether *d* or *k* is the parameter of interest. If the mouse cursor is within 10 pixels of the current (*d*, 0.5), then on every event we'll set the value of *d* to the cursor's *x* coordinate. Otherwise, if the cursor is within 10 pixels of any other point on the curve, we'll set the value of *k* so that the curve passes through

the cursor's location. Just to get things working, for now we defer the calculation of pixel distance between two points, using instead a `ptdist()` function that always returns 1 so that we always end up setting *d*. We compare to 100, instead of 10, because we won't bother taking the square root when we calculate the distance.

Running `bp.hoc`, placing the mouse cursor over the `Graph` canvas, and dragging it back and forth with the mouse button pressed shows satisfying tracking of the curve with the cursor (Fig. 14.5 A), so there is no need to worry yet about performance.

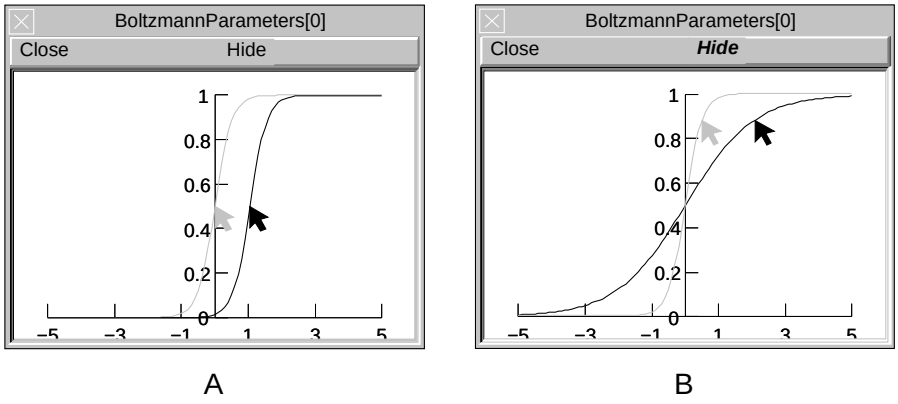


Fig. 14.5. A. The code in Listing 14.6 enables click and drag to shift the curve from side to side. B. Forcing `adjust_` to 2 allows us to test the use of this tool to set *k*.

Testing the ability of `adjust()` to change *k* is easily done by temporarily forcing `adjust_` to have the value 2 on the press event. In considering the inverse of the Boltzmann function, there are limits on the range $0 < y < 1$ and a singularity at $x == d$. This first attempt at doing something reasonable is likely to be sufficient but we won't know until we try it. In fact, it works much better than adding another control point for *k* (Fig. 14.5 B).

Some kind of mark, e.g. a little open square, is needed to indicate the control point at (*d*, 0.5). To do this we put

```
g.mark(d, 0.5, "s", 8)
```

just before `g.flush` in the `pl()` procedure. A size of 8 points is perhaps a bit large for a control point (Fig. 14.6). Later we will add a visual readout of the values of *d* and *k*.

So far, it has been most natural to specify plotting and event handling in terms of scene coordinates. However, calculating proximity between something plotted on a `Graph` and the location of the mouse cursor is best done in pixel or screen coordinates, since that avoids the problems caused by disparities between scene x and y scales. This requires revisions to `ptdist()`, where the information needed to transform between scene and screen coordinates will be obtained by a sequence of calls to `view_info()`. To recall the details of `view_info()`'s various arguments, we'll have to refer to the documentation of the `Graph` class and its methods in the Programmer's Reference.

Since each `Graph` can be displayed in more than one view simultaneously, the transformation between scene coordinates and screen coordinates depends on which view we are interested in. Of course, the view we want is the one that contains the mouse cursor, and this is the purpose of the first call to `view_info()`.

```
func ptdist() {local i, x1, y1, x2, y2
   i = g.view_info()  // i is the view in which
                      // the mouse cursor is located
   // $1..$4 are scene (x,y) of mouse cursor
   // and corresponding point on curve, respectively
   x1 = g.view_info(i, 13, $1)
   y1 = g.view_info(i, 14, $2)
   x2 = g.view_info(i, 13, $3)
   y2 = g.view_info(i, 14, $4)
   return (x1 - x2)^2 + (y1 - y2)^2
}
```


A little testing shows us that it is difficult to get the cursor near enough to the curve when the slope is large, because even a slight offset in the x coordinate causes a large jump of `b(x)` away from the cursor's y coordinate. We can fix this by adding a clause to `adjust()` that detects whether the horizontal distance between the cursor and the curve is small. Thus

```
} else if (ptdist($2, $3, $2, b($2)) < 100) {
```

becomes

```
} else if (ptdist($2, $3, $2, b($2)) < 100 \
           || abs($2 - b_inv($3)) < 10 {
```

(the backslash at the end of a line is a statement continuation character). Implementation of `b_inv()`, the inverse to the Boltzmann function, is more than 90% argument and parameter testing.

```
func b_inv() {local x
   if ($1 >= 1) {
      x = 700
   } else if ($1 <= 0){
      x = -700
   } else {
      x = log(1/$1 - 1)
   }
   if (k == 0) {
      return 1e9
   }
   return d - x/(4*k)
}
```

Finishing up

As mentioned earlier, it would be nice to have a direct indication of the values of the parameters. To do this, we add a horizontal panel to the bottom of the `VBox` and make the field editor buttons call the plot function `pl()` whenever the user changes the value.

Also, when a `Graph` menu tool is created within the scope of an open `xpanel`, the tool selector appears as a radio button in the panel. The code fragment to do this, in context in the `build()` procedure, is

```
g = new Graph()
g.size(-5,5,0,1)
xpanel("", 1)
   xpvalue("k", &k, 1, "pl()")
   xpvalue("d", &d, 1, "pl()")
   g.menu_tool("Adjust", "adjust")
xpanel()
g.exec_menu("Adjust")
```

We use `xpvalue()` and pointers, instead of `xvalue()` and variable names, because field editors assume variable names are at the top level of the interpreter. This is in contrast to action statements, such as `pl()` in this instance, which are executed in the context of the object. Figure 14.6 shows what the tool now looks like.

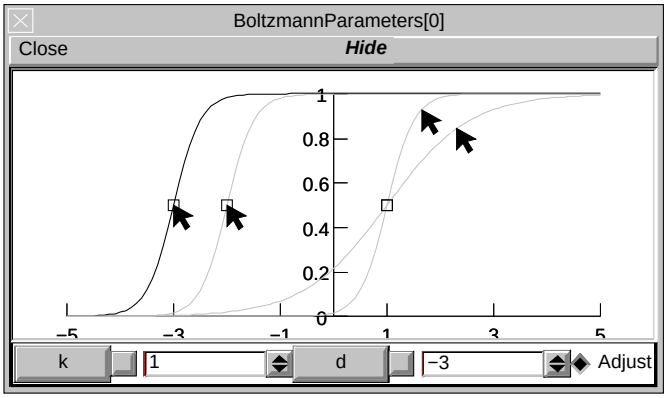


Fig. 14.6. Final appearance of the "Boltzmann Parameters" tool. The parameter values displayed are for the leftmost curve. Other curves with arrows suggest the process of selection and dragging the mouse.

The last touch is to change the `save()` procedure so that it saves the tool-specific state. By default, scene coordinates are saved instead of view coordinates, so it is a good

idea in this case to make them equivalent. This is done by adding these tool-dependent statements to `save()`

```
// insert tool-dependent statements here
sprint(tstr, "{k=%g d=%g}", k, d)
box.save(tstr)
g.exec_menu("Scene=View")
```

The first two statements, which incidentally take advantage of the `tstr` string variable that already exists, write the assignments of *k* and *d* to the session file, while the third one takes care of making scene and view coordinates equivalent.

The entire code for the final implementation of our tool is shown in Listing 14.7.

```
begintemplate BoltzmannParameters
  public g, box, map
  objref g, box, this
  strdef tstr

  proc init() {
    k = 1
    d = 0
    build()
    if (numarg() == 0) {
      map()
    }
  }

  proc build() {
    box = new VBox()
    box.save("save()")
    box.ref(this)
    box.intercept(1)
    g = new Graph()
    g.size(-5,5,0,1)
    xpanel("", 1)
      xpvalue("k", &k, 1, "pl()")
      xpvalue("d", &d, 1, "pl()")
      g.menu_tool("Adjust", "adjust")
    xpanel()
    g.exec_menu("Adjust")
    box.intercept(0)
  }
```



```
proc map() {
  sprint(tstr, "%s", this)
  if (numarg() == 0) {
    box.map(tstr)
  } else {
    box.map(tstr, $2, $3, $4, $5)
  }
  pl()
}

proc save() {
  box.save("load_file(\"bp.hoc\", \"BoltzmannParameters\")\n}\n{")
  box.save("ocbox_ = new BoltzmannParameters(1)")
  box.save("}\n{object_push(ocbox_)}")
  // insert tool-dependent statements here
  sprint(tstr, "{k=%g d=%g}", k, d)
  box.save(tstr)
  g.exec_menu("Scene=View")
  // end of tool-dependent statements
  box.save("{object_pop()}\n{")
  g.save_name("ocbox_.g", 1)
}

func b() { local x
  x = 4*k*(d - $1)
  if (x > 700) { return 0 }
  return 1/(1 + exp(x))
}

proc pl() { local i, x, x1, x2
  g.erase_all
  x1 = g.size(1)
  x2 = g.size(2)
  g.beginline
  for i=0, 100 {
    x = x1 + i*(x2 - x1)/100
    g.line(x, b(x))
  }
  g.mark(d, 0.5, "s", 8)
  g.flush
}
```



```
  proc adjust() {
    if ($1 == 2) { // left mouse button pressed
      adjust_ = 0
      // $2 and $3 are scene coords of mouse cursor
      if (ptdist($2, $3, d, 0.5) < 100) {
        adjust_ = 1
      } else if (ptdist($2, $3, $2, b($2)) < 100 \
                 || abs($2 - b_inv($3)) < 10) {
        adjust_ = 2
      }
    }
    if (adjust_ == 1) {
      d = $2
      pl()
    }
    if (adjust_ == 2) {
      if ($3 > 0.99) $3 = 0.99
      if ($3 < 0.01) $3 = 0.01
      if ($2 > d || $2 < d) {
        k = log(1/$3 - 1)/(4 * (d - $2))
        pl()
      }
    }
  }

  func ptdist() {local i, x1, y1, x2, y2
    i = g.view_info() // i is the view in which
                      // the mouse cursor is located
    // $1..$4 are scene (x,y) of mouse cursor
    // and corresponding point on the curve, respectively
    x1 = g.view_info(i, 13, $1)
    y1 = g.view_info(i, 14, $2)
    x2 = g.view_info(i, 13, $3)
    y2 = g.view_info(i, 14, $4)
    return (x1 - x2)^2 + (y1 - y2)^2
  }

  func b_inv() {local x
    if ($1 >= 1) {
      x = 700
    } else if ($1 <= 0){
      x = -700
    } else {
      x = log(1/$1 - 1)
    }
    if (k == 0) {
      return 1e9
    }
    return d - x/(4*k)
  }
endtemplate BoltzmannParameters
```

Listing 14.7. Complete source code for the BoltzmannParameters tool.

Chapter 14 Index

X

Appendix A1

Mathematical analysis of IntFire4

The `IntFire4` mechanism is an artificial spiking cell with a fast, monoexponentially decaying excitatory current *e* and a slower biexponential (similar to alpha function) inhibitory current i_2 that are summed by an even slower leaky integrator. It fires when the membrane state *m* reaches 1; after firing, only the membrane state returns to 0. The dynamics of `IntFire4` are specified by four time constants--τ_e for the excitatory current, τ_{i_1} and τ_{i_2} for the inhibitory current, and τ_m for the leaky integrator--and it is assumed that $\tau_e < \tau_{i_1} < \tau_{i_2} < \tau_m$. However, the differential equations that govern `IntFire4` are more conveniently written in terms of rate constants, i.e.

$$\frac{de}{dt} = -k_e e \qquad \text{Eq. A1.1}$$

$$\frac{di_1}{dt} = -k_{i_1} i_1 \qquad \text{Eq. A1.2}$$

$$\frac{di_2}{dt} = -k_{i_2} i_2 + a_{i_1} i_1 \qquad \text{Eq. A1.3}$$

$$\frac{dm}{dt} = -k_m m + a_e e + a_{i_2} i_2 \qquad \text{Eq. A1.4}$$

where each rate constant k is the reciprocal of the corresponding time constant, and $k_e > k_{i_1} > k_{i_2} > k_m$. An input event adds its weight w instantaneously to e or i_1, depending on whether w is > 0 (excitatory) or < 0 (inhibitory), respectively. The states e, i_1, i_2, and m are normalized by the constants a_e, a_{i_1}, and a_{i_2}, so that an excitatory weight w_e drives e and m to a maximum of w_e, and an inhibitory weight w_i drives i_1, i_2, and m to a minimum of w_i (see Fig. 10.15).

This system of equations can be solved by repeatedly making use of the fact that the solution to

$$\frac{dy}{dt} = -k_1 y + a\,\mathrm{e}^{-k_2 t} \qquad \text{Eq. A1.5}$$

is

$$y = y_0\,\mathrm{e}^{-k_1 t} + b\left(\mathrm{e}^{-k_1 t} - \mathrm{e}^{-k_2 t}\right) \qquad \text{Eq. A1.6}$$

where y_0 is the value of y at $t = 0$, and $b = a / (k_2 - k_1)$. Note that when $a > 0$ and $k_2 > k_1$, we may conclude that $b > 0$.

The solution to Eqns. A1.1-A1.4 is

$$e(t) = e_0\,\mathrm{e}^{-k_e (t - t_0)} \qquad \text{Eq. A1.7}$$

$$i_1(t) = i_{1_0}\,\mathrm{e}^{-k_{i_1} (t - t_0)} \qquad \text{Eq. A1.8}$$

$$i_2(t) = i_{2_0} e^{-k_{i_2}(t-t_0)} + b_{i_1} i_1 (e^{-k_{i_2}(t-t_0)} - e^{-k_{i_1}(t-t_0)}) \qquad \text{Eq. A1.9}$$

$$\begin{aligned} m(t) = {} & m_0 e^{-k_m(t-t_0)} \\ & + b_e (e^{-k_m(t-t_0)} - e^{-k_e(t-t_0)}) e_0 \\ & + b_{i_2} (e^{-k_m(t-t_0)} - e^{-k_{i_2}(t-t_0)}) i_{2_0} \\ & + b_{i_2} b_{i_1} (e^{-k_m(t-t_0)} - e^{-k_{i_2}(t-t_0)}) i_{1_0} \\ & - b_{i_2} b_{i_1} \frac{k_{i_2} - k_m}{k_{i_1} - k_m} (e^{-k_m(t-t_0)} - e^{-k_{i_1}(t-t_0)}) i_{1_0} \end{aligned} \qquad \text{Eq. A1.10}$$

where

t_0 is the time of the most recent input event

e_0, i_{1_0}, i_{2_0}, and m_0 are the values of e , i_1, i_2, and m immediately after that event was handled

IntFire4 uses self-events to successively approximate the firing time. At initialization, a self-event is issued that will return at $t = 10^9$ ms (i.e. never). Arrival of a new event at time t_{event} causes the following sequence of actions:

- The current values of the states e , i_1, i_2, and m are calculated analytically from Eqns. A1.7-A1.10.
- The values of e_0, i_{1_0}, i_{2_0}, and m_0 are updated to the current values of e , i_1, i_2, and m, and the value of t_0 is updated to t_{event}.

- If $m > 1 - \epsilon$, the cell fires and m is reset to 0.
- If the event was a self-event, the next firing time is estimated and a new self-event is issued that will return at that time.
- If the event was an input event, then depending on whether it was excitatory or inhibitory (i.e. weight $w < 0$ or > 0), w is instantaneously added to e or i_1, respectively. That done, the next firing time is estimated, and the yet-outstanding self-event is moved to that time.

The next firing time is approximated from the values of m and its derivative immediately after the event is handled. If $m(t_0)' \leq 0$, then the estimated firing time is set to 10^9, i.e. never. If $m(t_0)' > 0$, the estimated firing time is $(1-m(t_0))/m(t_0)'$. In the following sections we prove that this strategy produces an estimate that is never later than the true firing time; otherwise, the simulation would be in error.

From a practical perspective, it is also important that successive approximations converge rapidly to the true firing time, to avoid the overhead of a large number of self events. Since the slope approximation is equivalent to Newton's method for finding the t at which $m = 1$, we only expect slow convergence when the maximum value of m is close to 1. Using a sequence of self-events is superior to carrying out a complete Newton method solution for the firing time, because it is most likely that external (input) events will arrive in the interval between firing times, invalidating the computation of the next firing time. The number of iterations that should be carried out per self-event remains an experimental question, because self-event overhead depends partly on the number of

outstanding events in the event queue. A single Newton iteration generally takes longer than the overhead associated with self-events.

Proof that the estimate is never later than the true firing time

For notational clarity, we will use m_0 and m_0' to refer to the values of m and m' immediately after the event is handled. The proof consists of two major parts. First we show that if $m_0' \leq 0$, then $m(t)$ remains < 1. Then we show that if $m_0' > 0$, then $(1-m_0)/m_0'$ underestimates the firing time. This latter part is divided into the cases $m_0 \leq 0$, and $m_0 > 0$. First, however, we present a useful lemma.

Lemma:

If

$$k_1 > k_2 > k \qquad \text{Eq. A1.11}$$

$$f_1(t) = \mathrm{e}^{-k\,t} - \mathrm{e}^{-k_1\,t} \qquad \text{Eq. A1.12}$$

$$f_2(t) = \mathrm{e}^{-k\,t} - \mathrm{e}^{-k_2\,t} \qquad \text{Eq. A1.13}$$

then

$$\frac{f_1(t)}{k_1 - k} \le \frac{f_2(t)}{k_2 - k} \qquad \text{Eq. A1.14}$$

for all $t \ge 0$.

Proof:

First note that $f_1(0) = f_2(0) = 0$ so the lemma holds at $t = 0$. Also note that $f_1'(0) = k_1 - k$ and $f_2'(0) = k_2 - k$ so both sides of the inequality we are trying to prove have slope 1 at $t = 0$.

Next consider $t > 0$. $e^{-k\,t} > e^{-k_1\,t} > e^{-k_2\,t}$ so it is safe to divide by $e^{-k\,t} - e^{-k_2\,t}$, and we can write

$$\frac{f_1(t)}{k_1 - k} - \frac{f_2(t)}{k_2 - k} = \frac{k_2 - k}{f_2(t)} \left(\frac{f_1(t)}{f_2(t)} \frac{k_2 - k}{k_1 - k} - 1 \right) \qquad \text{Eq. A1.15}$$

Analyzing the right hand side of this equation, we see that the ratio $(k_2 - k) / f_2(t)$ is positive. Also, $(k_2 - k)/(k_1 - k) < 1$. Furthermore, f_1 and f_2 are positive, and since $e^{-k\,t} - e^{-k_2\,t}$ then $f_1 / f_2 < 1$. Thus the expression inside the parentheses is negative, and the entire right hand side of Eq. A1.16 is < 0. This completes the proof of the lemma.

Note that Eq. A1.15 can be expressed as

$$f_1(t) \frac{k_2 - k}{k_1 - k} \le f_2(t) \qquad \text{Eq. A1.16}$$

Also, in the limit as k_2 approaches k, we have

$$\frac{f_1(t)}{k_1 - k} \le t\, e^{-k t} \qquad \text{Eq. A1.17}$$

Part 1: if $m_0' \le 0$, then $m(t)$ remains < 1

We now prove that if $m_0' \le 0$, then m(t) remains < 1 (i.e. the firing time is infinity) regardless of e, i_1, or i_2. Since we are trying to predict the trajectory of m based on the values of m and m' immediately following the most recent event, it will be advantageous to think in terms of the time that has elapsed since that event, i.e. relative time, rather than absolute time. Therefore we substitute t for $t - t_0$, and rewrite Eq. A1.10 as

$$\begin{aligned} m(t) = {} & m_0 e^{-k_m t} \\ & + b_e \left(e^{-k_m t} - e^{-k_e t}\right) e_0 \\ & + b_{i_2} \left(e^{-k_m t} - e^{-k_{i_2} t}\right) i_{2_0} \\ & + b_{i_2} b_{i_1} \left(e^{-k_m t} - e^{-k_{i_2} t}\right) i_{1_0} \\ & - b_{i_2} b_{i_1} \frac{k_{i_2} - k_m}{k_{i_1} - k_m} \left(e^{-k_m t} - e^{-k_{i_1} t}\right) i_{1_0} \end{aligned} \qquad \text{Eq. A1.18}$$

From the lemma we see that the sum of the last two major terms on the right hand side is ≤ 0. Factoring out the common multiplier $b_{i_2} b_{i_1} i_{1_0}$ from these terms leaves the expression

$$(e^{-k_m t} - e^{-k_{i_2} t}) - \frac{k_{i_2} - k_m}{k_{i_1} - k_m} (e^{-k_m t} - e^{-k_{i_1} t})$$

which is positive because $k_{i_1} > k_{i_2} > k_m$. However, the multiplier $b_{i_2} b_{i_1} i_{1_0}$ itself is ≤ 0 because i_{1_0} is ≤ 0 while b_{i_1} and b_{i_2} are both > 0.

Thus

$$\begin{aligned} m(t) \leq\ & m_0 e^{-k_m t} \\ & + b_e (e^{-k_m t} - e^{-k_e t}) e_0 \\ & + b_{i_2} (e^{-k_m t} - e^{-k_{i_2} t}) i_{2_0} \end{aligned} \qquad \text{Eq. A1.19}$$

The last term here is negative (except at $t = t_0$, where it is 0), and we can use our lemma again to replace it with something that is not so negative, i.e.

$$\begin{aligned} m(t) \leq\ & m_0 e^{-k_m t} \\ & + b_e (e^{-k_m t} - e^{-k_e t}) e_0 \\ & + b_{i_2} \frac{k_{i_2} - k_m}{k_e - k_m} (e^{-k_m t} - e^{-k_e t}) i_{2_0} \end{aligned} \qquad \text{Eq. A1.20}$$

Rewriting this as

$$\begin{aligned} m(t) \leq\ & m_0 e^{-k_m t} \\ & + \frac{1}{k_e - k_m} (e^{-k_m t} - e^{-k_e t}) (a_e e_0 + a_{i_2} i_{2_0}) \end{aligned} \qquad \text{Eq. A1.21}$$

we note that $a_e e_0 + a_{i_2} i_{2_0}$ is $m_0' + k_m m_0$, so

$$m(t) \le m_0 e^{-k_m t} + \frac{k_m}{k_e - k_m}\left(e^{-k_m t} - e^{-k_e t}\right) m_0 + \frac{1}{k_e - k_m}\left(e^{-k_m t} - e^{-k_e t}\right) m_0' \qquad \text{Eq. A1.22}$$

We have stipulated that $m_0' \le 0$, so the last term is ≤ 0 and we can remove it and write

$$m(t) \le m_0 \left[e^{-k_m t} + \frac{k_m}{k_e - k_m}\left(e^{-k_m t} - e^{-k_e t}\right) \right] \qquad \text{Eq. A1.23}$$

Since $m_0 < 1$, we only have to prove that the bracketed expression is ≤ 1. Clearly it is 1 when $t = 0$. Factoring this expression gives

$$\frac{k_e}{k_e - k_m} e^{-k_m t} - \frac{k_m}{k_e - k_m} e^{-k_e t}$$

whose derivative is

$$-\frac{k_e k_m}{k_e - k_m} e^{-k_m t} + \frac{k_e k_m}{k_e - k_m} e^{-k_e t}$$

or

$$-\frac{k_e k_m}{k_e - k_m}\left(e^{-k_m t} - e^{-k_e t}\right)$$

which is 0 at $t = 0$ and negative for $t > 0$. A function that is 1 at $t = 0$ and has a negative derivative for $t > 0$ must be ≤ 1 for $t > 0$.

This completes Part 1 of the proof. Next we prove that, if $m_0' > 0$, the first Newton iteration estimate $(1 - m_0) / m_0'$ is never later than the true firing time.

Part 2: if $m' > 0$, $1 - m / m'$ underestimates the firing time

The last thing to do is to prove that, if $m_0' > 0$, the Newton iteration $(1 - m_0) / m_0'$ is never later than the firing time. We start from Eq. A1.22, but since we now stipulate that $m_0' > 0$, the last term is positive. According to our lemma, we can replace it with the larger term $t\, e^{-k_m t}\, m_0'$ to get

$$\begin{aligned} m(t) \le\ & m_0 e^{-k_m t} \\ & + \frac{k_m}{k_e - k_m} (e^{-k_m t} - e^{-k_e t})\, m_0 \\ & + t\, e^{-k_m t}\, m_0' \end{aligned} \qquad \text{Eq. A1.24}$$

Consider the case where $m_0 > 0$. The sum of the first two terms is $\le m_0$ and the third term is $\le t\, m_0'$, so

$$m(t) \le m_0 + m_0'\, t \qquad \text{Eq. A1.25}$$

and the Newton iteration underestimates the firing time.

Now consider case where $m_0 < 0$. The second term of Eq. A1.24 is ≤ 0 so we can throw it out and write

$$m(t) \le (m_0 + m_0'\, t) e^{-k_m t} \qquad \text{Eq. A1.26}$$

We complete our proof by applying a geometric interpretation to this inequality. The value of t at which the line $y(t) = m_0 + m_0' t$ intersects $y = 1$ is the estimated firing time found by a Newton iteration. Equation A1.24 shows that the trajectory of the membrane state variable runs at or below that line. Consequently, the Newton iteration underestimates the true firing time.

Appendix A2

NEURON's built-in editor

Many people are already comfortable with their own text editor and will find it most convenient to load text files into `hoc` with the `xopen()` or `load_file()` command. However, NEURON has a built-in editor that is closely related, if not identical, to MicroEMACS (`http://uemacs.tripod.com/`), and which we will simply call "emacs." In this era of richly menued, GUI-based editing software, NEURON's tiny emacs editor is definitely showing its age, and noone would ever confuse it with the much more powerful EMACS that has Lisp-like syntax (Cameron and Rosenblatt 1991). Nonetheless, it is quite capable and has the advantage of having same "look and feel" on all platforms.

Like EMACS, emacs is a command-driven editor with modes and multiple buffers. What does this mean? Being "command-driven" means that there are no menus or buttons to click on with a mouse. Instead, special keystrokes, like the "keyboard shortcuts" in other editors, are used as commands to the editor. Having "modes" implies that some of these commands change how emacs responds to what you type, like the way many editors can be switched between "insert" and "replace" mode. The notion of "buffers" may seem strange, but if you have ever used any kind of editor or word processing software, then you have almost certainly worked with buffers even if you

didn't realize it. When you open a file to edit it, a copy of the file's contents are placed in a buffer in the computer's memory, and you edit the contents of the buffer. When you save your work, the buffer is written back to the file.

In the following pages we describe the commands that are available in emacs. The keystrokes for these commands are represented with this shorthand notation:

notation	means
^A	press and hold the "control" (Ctrl) key while typing a or A
^A B	first type ^A, then type b or B
Esc	press and release the "escape" (Esc) key
Esc-A	press and release the "escape" (Esc) key, then type a or A
Esc n A	press and release the "escape" (Esc) key, type a number, then type a or A
space	press and release the space bar
Tab	press and release the tab key

Starting and stopping

Switching from hoc to emacs

Type the command em at the oc> prompt.

Returning from emacs to hoc

Type ^C. The current edit buffer will be read into hoc and executed. If the hoc interpreter encounters a syntax error, NEURON will return to emacs with the cursor at the line where the parser failed. Note that ^C only executes a buffer. It does not save any unsaved buffer. On exit from hoc you will be prompted for whether or not to save each unsaved buffer.

Killing the current command

^G stops the current command. Sorry, emacs has no "undo."

Moving the cursor

^F	Forward 1 character.*
^B	Backward 1 character.*
Esc-F	Forward 1 word.
Esc-B	Backward 1 word.
^E	To end of line.
^A	To start of line.
^N	Next line.
^P	Previous line.
Esc-N	To end of next paragraph.
Esc-P	To start of previous paragraph.
^V	Scroll down one screen.*
Esc-V	Scroll up one screen.*
Esc->	To end of buffer.*
Esc-<	To start of buffer.*

*--Under MSWindows the arrow keys, Page Down, Page Up, End, and Home keys may work.

Modes

There are five modes that can be used individually, in any combination, or not at all.

^X M	Add mode. At the prompt type the name of the mode to be added.
^X ^M	Delete mode. At the prompt type the name of the mode to be removed.
Mode name	**Description**
over	New typing replaces (overwrites) instead of inserting characters.

`cmode`	Automatic indenting of C programs.
`wrap`	Automatic word wrap (inserts return at right margin).
`exact`	Search using exact case matching.
`view`	View buffer without changing it (ignores commands to change buffer).

Deleting and inserting

`^H`	Delete previous character.
`^D`	Delete next character.
`Esc-^H`	Delete previous word.
`Esc-D`	Delete next word.
`^K`	Delete to end of line.
`^O`	Insert line.

Under MSWindows, the Insert key inserts a space, and the Delete key acts like `^H`.

Blocks of text: marking, cutting, and pasting

`Esc-space`	Set mark at cursor position. After a mark has been set, the "region" is the text between the mark and the current position of the cursor. The "region" changes dynamically as the cursor is moved around.
`^X ^X`	Exchange mark and cursor.
`^W`	Delete (cut) region between mark and cursor. The deleted text is kept in the "kill buffer," replacing whatever the kill buffer may have already contained.
`Esc-W`	Copy region between mark and cursor to the kill buffer.
`^Y`	Copy (paste) text from kill buffer to cursor location.

Searching and replacing

`^S`	Search forward. At the prompt enter the search string followed by Esc. Case sensitive if buffer is in "exact" mode.
`^R`	Search backward.
`Esc-R`	Start from present location and replace every instance of the first string (the "search string") with the second string.

Esc-Q	Like Esc-R but asks for user's approval. User may reply with y or press the space bar to replace and continue ! to replace the rest n to reject this replacement and continue to the next occurrence of the search string ^G to stop . to stop and return to the starting point ? for a list of options.

Text formatting and other tricks

Esc-C	Capitalize word.
Esc-U	Change word to upper case.
^X ^U	Change region to upper case.
Esc-L	Change word to lower case.
^X ^L	Change region to lower case.
^T	Transpose character at cursor with prior character.
^Q	"Quote" next character (allows entry of control codes into text).
^X ^U	Change region to upper case.
Esc n ^X F	Set right margin to column n.
Esc-J	Justify the paragraph so it fits between the margins.
Esc n Tab	Set right margin to column n.
^X =	Show line number, character count, and size of buffer.
^X !	Send a command to the OS.

Buffers and file I/O

When emacs first starts, it comes up with only one buffer, and that buffer is empty until you enter some text or type a command that loads a file. You can also have more than one buffer, each containing its own text, and you can copy text between them. To help keep things from getting mixed up, each buffer has a name. The first buffer is automatically called main.

It is a good idea to break large programs into many files and edit them in separate buffers. Buffers larger than 100 lines may take a noticeable time to interpret.

`^X B`	Switch to another buffer. You will be prompted for the name of a buffer. If the name you type doesn't exist, a new buffer will be created and given that name.
`^X K`	Delete a non-displayed buffer.
`^X ^B`	Show a directory of the buffers in a window.
`^X X`	Switch to the next buffer in the directory.
`Esc-^X 1`	Clear the buffer directory window.
`^X ^F`	Create a new buffer and load a file into it. You will be prompted for the name of the file. If the file doesn't already exist, the buffer will still be created with that name but it will be empty.
`^X ^R`	Read a file into the current buffer. Pre-existing contents of the current buffer are overwritten.
`^X ^I`	Insert a file into the current buffer at the location of the cursor.
`^X ^V`	Read a file into the current buffer and set the buffer to view mode.
`^X ^S`	Write current buffer to disk, using the name of the buffer as the name of the file. Overwrites pre-existing file with same name.
`^X ^W`	Write current buffer to disk. You will be prompted for a file name. This name will also become the name of the current buffer.
`^C`	Exit from emacs to `hoc`. The current buffer is passed to the `hoc` interpreter. For more details see **Returning from emacs to hoc** under **Starting and stopping** above.
`^X ^C`	Exit from both emacs and NEURON. You will be asked if you want to save buffers.

Windows

The emacs editor appears inside the terminal window from which NEURON was started. At first this window contains only one area for editing text, but you can split this into several different areas. These areas are called "windows" in standard EMACS and MicroEMACS terminology (Cameron and Rosenblatt 1991). Each window may show a different part of the same buffer, or part of a different one.

^X 2	Split the current window in two.
^X 1	Remove all but the current window.
^X N	Put the cursor in the next window.
^X P	Put the cursor in the previous window.
^X ^	Enlarge the current window.
Esc-^V	Scroll the other window down.
Esc-^Z	Scroll the other window up.

This use of the word "windows" was well-established years before the advent of microcomputer operating systems with windowed GUIs, in an era when the thought of copyrighting this word would have seemed ludicrous.

Macros and repeating commands

Esc n -	Repeats the next command n times.
^U n	Repeats the next command n times.
^U	Repeats the next command 4 times. Typing ^U several times in a row repeats the next command by a power of 4, e.g. ^U ^U ^U ^N moves the cursor down 64 lines.
^X (	Begin macro.
^X)	End macro.
^X E	Execute macro. All keystrokes entered from the beginning to the end of the macro are interpreted as emacs commands.

References

Cameron, D. and Rosenblatt, B. Learning GNU Emacs. O'Reilly & Associates, Inc., Sebastopol, CA. 1991.

Web site retrieved Oct. 27, 2004. MicroEMACS: Binaries, executables, documentation, source code. http://uemacs.tripod.com/.

Appendix A2 Index

