The NEURON Simulation Environment

M.L. Hines[1, 3] and N.T. Carnevale[2, 3]

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

and [3]Neuroengineering and Neuroscience Center

Yale University

michael.hines@yale.edu

ted.carnevale@yale.edu

an extended preprint of

Hines, M.L. and Carnevale, N.T.:

The NEURON Simulation Environment

Neural Computation 9:1179-1209, 1997.

CONTENTS

1. INTRODUCTION

NEURON (Hines 1984; 1989; 1993; 1994) provides a powerful and flexible environment for implementing biologically realistic models of electrical and chemical signaling in neurons and networks of neurons. This article describes the concepts and strategies that have guided the design and implementation of this simulator, with emphasis on those features that are particularly relevant to its most efficient use.

1.1 The problem domain

Information processing in the brain results from the spread and interaction of electrical and chemical signals within and among neurons. This involves nonlinear mechanisms that span a wide range of spatial and temporal scales (Carnevale and Rosenthal 1992) and are constrained to operate within the intricate anatomy of neurons and their interconnections. Consequently the equations that describe brain mechanisms generally do not have analytical solutions, and intuition is not a reliable guide to understanding the working of the cells and circuits of the brain. Furthermore, these nonlinearities and spatiotemporal complexities are quite unlike those that are encountered in most nonbiological systems, so the utility of many quantitative and qualitative modeling tools that were developed without taking these features into consideration is severely limited.

NEURON is designed to address these problems by enabling both the convenient creation of biologically realistic quantitative models of brain mechanisms and the efficient simulation of the operation of these mechanisms. In this context the term "biological realism" does not mean "infinitely detailed." Instead it means that the choice of which details to include in the model and which to omit are at the discretion of the investigator who constructs the model, and not forced by the simulation program.

To the experimentalist NEURON offers a tool for cross-validating data, estimating experimentally inaccessible parameters, and deciding whether known facts account for experimental observations. To the theoretician it is a means for testing hypotheses and determining the smallest subset of anatomical and biophysical properties that is necessary and sufficient to account for particular phenomena. To the student in a laboratory course it provides a vehicle for illustrating and exploring the operation of brain mechanisms in a simplified form that is more robust than the typical "wet lab" experiment. For experimentalist, theoretician, and student alike, a powerful simulation tool such as NEURON can be an indispensable aid to developing the insight and intuition that are needed if one is to discover the order hidden within the intricacy of biological phenomena, the order that transcends the complexity of accident and evolution.

1.2 Experimental advances and quantitative modeling

Experimental advances drive and support quantitative modeling. Over the past two decades the field of neuroscience has seen striking developments in experimental techniques that include

- high-quality electrical recording from neurons *in vitro* and *in vivo* using patch clamp
- multiple impalements of visually identified cells
- simultaneous intracellular recording from paired pre- and postsynaptic neurons
- simultaneous measurement of electrical and chemical signals
- multisite electrical and optical recording
- quantitative analysis of anatomical and biophysical properties from the same neuron
- photolesioning of cells
- photorelease of caged compounds for spatially precise chemical stimulation
- new drugs such as channel blockers and receptor agonists and antagonists
- genetic engineering of ion channels and receptors
- analysis of mRNA and biophysical properties from the same neuron
- "knockout" mutations

These and other advances are responsible for impressive progress in the definition of the molecular biology and biophysics of receptors and channels, the construction of libraries of identified neurons and neuronal classes that have been characterized anatomically, pharmacologically, and biophysically, and the analysis of neuronal circuits involved in perception, learning, and sensorimotor integration.

The result is a data avalanche that catalyzes the formulation of new hypotheses of brain function, while at the same time serving as the empirical basis for the biologically realistic quantitative models that must be used to test these hypotheses. Some examples from the large list of topics that have been investigated through the use of such models include

- the cellular mechanisms that generate and regulate chemical and electrical signals (Destexhe et al. 1996; Jaffe et al. 1994)
- drug effects on neuronal function (Lytton and Sejnowski 1992)
- presynaptic (Lindgren and Moore 1989) and postsynaptic (Destexhe and Sejnowski 1995; Traynelis et al. 1993) mechanisms underlying communication between neurons
- integration of synaptic inputs (Bernander et al. 1991; Cauller and Connors 1992)
- action potential initiation and conduction (Häusser et al. 1995; Hines and Shrager 1991; Mainen et al. 1995)
- cellular mechanisms of learning (Brown et al. 1992; Tsai et al. 1994a)
- cellular oscillations (Destexhe et al. 1993a; Lytton et al. 1996)
- thalamic networks (Destexhe et al. 1993b; Destexhe et al. 1994)
- neural information encoding (Hsu et al. 1993; Mainen and Sejnowski 1995; Softky 1994)

2. OVERVIEW OF NEURON

NEURON is intended to be a flexible framework for handling problems in which membrane properties are spatially inhomogeneous and where membrane currents are complex. Since it was designed specifically to simulate the equations that describe nerve cells, NEURON has three important advantages over general purpose simulation programs. First, the user is not required to translate the problem into another domain, but instead is able to deal directly with concepts that are familiar at the neuroscience level. Second, NEURON contains functions that are tailored specifically for controlling the simulation and graphing the results of real neurophysiological problems. Third, its computational engine is particularly efficient because of the use of special methods that take advantage of the structure of nerve equations (Hines 1984; Mascagni 1989).

However, the general domain of nerve simulation is still too large for any single program to deal optimally with every problem. In practice, each program has its origin in a focused attempt to solve a restricted class of problems. Both speed of simulation and the ability of the user to maintain conceptual control degrade when any program is applied to problems outside the class for which it is best suited.

NEURON is computationally most efficient for problems that range from parts of single cells to small numbers of cells in which cable properties play a crucial role. In terms of conceptual control, it is best suited to tree-shaped structures in which the membrane channel parameters are approximated by piecewise linear functions of position. Two classes of problems for which it is particularly useful are those in which it is important to calculate ionic concentrations, and those where one needs to compute the extracellular potential just next to the nerve membrane. It is especially capable for investigating new kinds of membrane channels since they are described in a high level language (NMODL (Moore and Hines 1996)) which allows the expression of models in terms of kinetic schemes or sets of simultaneous differential and algebraic equations. To maintain efficiency, user defined mechanisms in NMODL are automatically translated into C, compiled, and linked into the rest of NEURON.

The flexibility of NEURON comes from a built-in object oriented interpreter which is used to define the morphology and membrane properties of neurons, control the simulation, and establish the appearance of a graphical interface. The default graphical interface is suitable for exploratory simulations involving the setting of parameters, control of voltage and current stimuli, and graphing variables as a function of time and position.

Simulation speed is excellent since membrane voltage is computed by an implicit integration method optimized for branched structures (Hines 1984). The performance of NEURON degrades very slowly with increased complexity of morphology and membrane mechanisms, and it has been applied to very large network models (10^4 cells with 6 compartments each, total of 10^6 synapses in the net [T. Sejnowski, personal communication]).

3. MATHEMATICAL BASIS

Strategies for numerical solution of the equations that describe chemical and electrical signaling in neurons have been discussed in many places. Elsewhere we have briefly presented an intuitive rationale for the most commonly used methods (Hines and Carnevale 1995). Here we start from this base and proceed to address those aspects which are most pertinent to the design and application of NEURON.

3.1 The cable equation

The application of cable theory to the study of electrical signaling in neurons has a long history, which is briefly summarized elsewhere (Rall 1989). The basic computational task is to numerically solve the cable equation

$$\frac{\partial V}{\partial t} + I(V,t) = \frac{\partial^2 V}{\partial x^2} \qquad (1)$$

which describes the relationship between current and voltage in a one-dimensional cable. The branched architecture typical of most neurons is incorporated by combining equations of this form with appropriate boundary conditions.

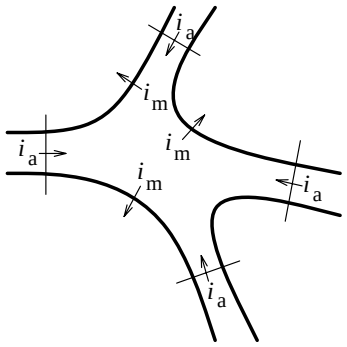

Figure 3.1. The net current entering a region must equal zero.

Spatial discretization of this partial differential equation is equivalent to reducing the spatially distributed neuron to a set of connected compartments. The earliest example of a multicompartmental approach to the analysis of dendritic electrotonus was provided by Rall (1964).

Spatial discretization produces a family of ordinary differential equations of the form

$$c_j \frac{dv_j}{dt} + i_{ion_j} = \sum_k \frac{v_k - v_j}{r_{jk}} \qquad (2)$$

Equation 2 is a statement of Kirchhoff's current law, which asserts that net transmembrane current leaving the *j*th compartment must equal the sum of axial currents entering this compartment from all sources (Fig. 3.1). The left hand side of this equation is the total membrane current, which is the sum of capacitive and ionic components. The capacitive component is $c_j\, dv_j/dt$, where c_j is the membrane capacitance of the compartment. The ionic component i_{ion_j} includes all currents through ionic channel conductances. The right hand side of Eq. 2 is the sum of axial currents that enter this compartment from its adjacent neighbors. Currents injected through a microelectrode would be added to the right hand side. The sign conventions for current are: outward transmembrane current is positive; axial current flow into a region is positive; positive injected current drives v_j in a positive direction.

Equation 2 involves two approximations. First, axial current is specified in terms of the voltage drop 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 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.

If the compartments are of equal size, it is easy to use Taylor's series to show that both of these approximations have errors proportional to the square of compartment length. Thus replacing the second partial derivative by its central difference approximation introduces errors proportional to Δx^2, and doubling the number of compartments reduces the error by a factor of four.

It is often not convenient for the size of all compartments to be equal. Unequal compartment size might be expected to yield simulations that are only first order accurate. However, comparison of simulations in which unequal compartments are halved or quartered in size generally reveals a second-order reduction of error. A rough rule of thumb is that simulation error is proportional to the square of the size of the largest compartment.

The first of two special cases of Eq. 2 that we wish to discuss allows us to recover the usual parabolic differential 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 the natural indices *j–1* and *j+1*, i.e.

$$c_j \frac{dv_j}{dt} + i_{ion_j} = \frac{v_{j-1} - v_j}{r_{j-1,j}} + \frac{v_{j+1} - v_j}{r_{j,j+1}}$$

If the compartments have the same length Δx and diameter d, then the capacitance of a compartment is $C_m \pi d \Delta x$ and the axial resistance is $R_a \Delta x / \pi (d/2)^2$. C_m is called the specific capacitance of the membrane, which is generally taken to be 1 µf / cm^2. R_a is the axial resistivity, which has different reported values for different cell classes (e.g. 35.4 Ω cm for squid axon). Eq. 2 then becomes

$$C_m \frac{dv_j}{dt} + i_j = \frac{d}{4R_a} \frac{v_{j+1} - 2v_j + v_{j-1}}{\Delta x^2}$$

where we have replaced the total ionic current i_{ion_j} with the current density i_j. The right hand term, as $\Delta x \to 0$, is just $\partial^2 V / \partial x^2$ at the location of the now infinitesimal compartment j.

The second special case of Eq. 2 allows us to recover the boundary condition. This is an important exercise since naive discretizations at the ends of the cable have destroyed the second order accuracy of many simulations. Nerve boundary conditions are that no axial current flows at the end of the cable, i.e. the end is sealed. This is implicit in Eq. 2, where the right hand side consists 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.

3.2 Spatial discretization in a biological context: sections and segments

Every nerve simulation program solves for the longitudinal spread of voltage and current by approximating the cable equation as a series of compartments connected by resistors (Fig. 3.4 and Eq. 2). The sum of all the compartment areas is the total membrane area of the whole nerve. Unfortunately, it is usually not clear at the outset how many compartments should be used. Both the accuracy of the approximation and the computation time increase as the number of compartments used to represent the cable increases. When the cable is "short," a single compartment can be made to adequately represent the entire cable. For long cables or highly branched structures, it may be necessary to use a large number of compartments.

This raises the question of how best to manage all the parameters that exist within these compartments. Consider membrane capacitance, which 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 which 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 graininess of their representation may have little to do with numerically adequate compartmentalization.

Although NEURON is a compartmental modeling program, the specification of biological properties (neuron shape and physiology) has been separated from the numerical issue of compartment size. What makes this possible is the notion of a **section**, which is a continuous length of unbranched cable. Although each section is ultimately discretized into compartments, values that can vary with position along the length of a section are specified in terms of a continuous parameter that ranges from 0 to 1 (normalized distance). In this way, section properties are discussed without regard to the number of segments used to represent it. This makes it easy to trade off between accuracy and speed, and enables convenient verification of the numerical correctness of simulations.

Sections are connected together to form any kind of branched tree structure. Fig. 3.2 illustrates how sections are used to represent biologically significant anatomical features. The top of this figure is a cartoon of a neuron with a soma that gives rise to a branched dendritic tree and an axon hillock connected to a myelinated axon. Each biologically significant component of this cell has its counterpart in one of the sections of the NEURON model, as shown in the bottom of Fig. 3.2: 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 the cell to remain foremost in the mind of the person who constructs and uses a NEURON model.

To accommodate requirements for numerical accuracy, NEURON represents each section by one or more **segments** of equal length (Figs. 3.3 and 3.4). The number of segments is specified by the parameter **nseg**, which can have a different value for each section.

At the center of each segment is a **node**, the location where the internal voltage of the segment is defined. The transmembrane currents over the entire surface area of a segment are associated with its node.

The nodes of adjacent segments are connected by resistors.

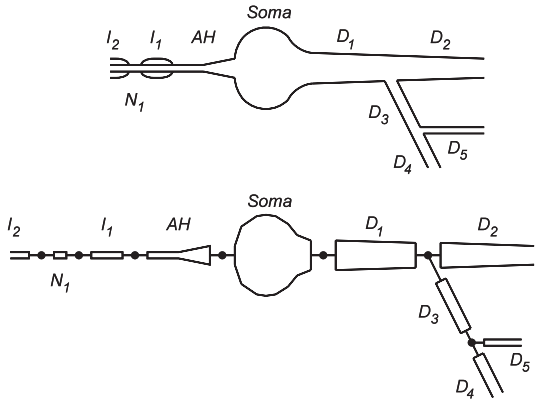

Figure 3.2. Top: cartoon of a neuron indicating the approximate boundaries between biologically significant structures. The left hand side of the cell body (*Soma*) is attached to an axon hillock (*AH*) that drives a myelinated axon (myelinated internodes I_i alternating with nodes of Ranvier N_i). From the right hand side of the cell body originates a branched dendritic tree (D_i). Bottom: how sections would be employed in a NEURON model to represent these structures.

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. This is the discretization method employed by NEURON. 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. However, 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.

Based on the position of the nodes, NEURON calculates the values of internal model parameters such as the average diameter, axial resistance, and compartment area that are assigned to each segment. Figs. 3.3 and 3.4 show how an unbranched portion of a neuron, called a neurite (Fig. 3.3A), is represented by a section with one or more segments. Morphometric analysis generates a series of diameter measurements whose centers lie on the midline of the neurite (thin axial line in Fig. 3.3B). These measurements and the path lengths between their centers are the dimensions of the section, which can be regarded as a chain of truncated cones or frusta (Fig. 3.3C).

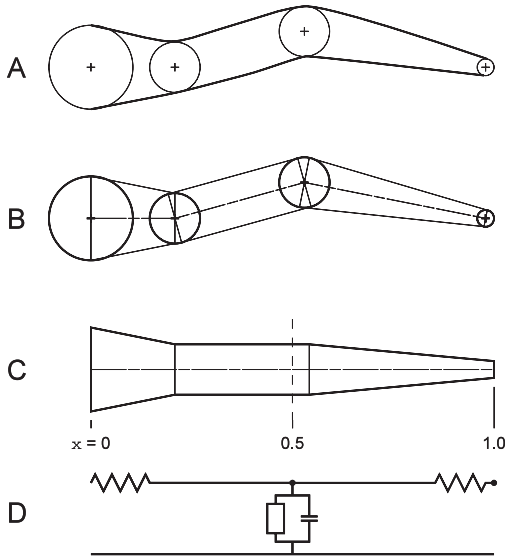

Figure 3.3. A: cartoon of an unbranched neurite (thick lines) that is to be represented by a section in a NEURON model. Computer-assisted morphometry generates a file that stores successive diameter measurements (circles) centered at x, y, z coordinates (crosses). B: each adjacent pair of diameter measurements becomes the parallel faces of a truncated cone or frustum. A thin centerline passes through the central axis of the chain of solids. C: the centerline has been straightened so the faces of adjacent frusta are flush with each other. The scale underneath the figure shows the distance along the midline of the section in terms of the normalized position parameter **x**. The vertical dashed line at x = 0.5 divides the section into two halves of equal length. D: Electrical equivalent circuit of the section as represented by a single segment (**nseg** = 1). The open rectangle includes all mechanisms for ionic (non-capacitive) transmembrane currents.

Distance along the length of a section is discussed in terms of the normalized position parameter **x**. That is, one end of the section corresponds to x = 0 and the other end to x = 1. In Fig. 3.3C these locations are depicted as being on the left and right hand ends of the section. The locations of the nodes and the boundaries between segments are conveniently specified in terms of this normalized position parameter. In general, a section has `nseg` segments that are demarcated by evenly spaced boundaries at intervals of 1 / `nseg`. The nodes at the centers of these segments are located at

x = (2 *i* – 1) / 2 nseg where *i* is an integer in the range [1, nseg]. As we shall see later, x is also used in specifying model parameters or retrieving state variables that are a function of position along a section (see **4.4 Range variables**).

The special importance of x and nseg lies in the fact that they free the user from having to keep track of the correspondence between segment number and position on the nerve. In early versions of NEURON, all nerve properties were stored in vector variables where the vector index was the segment number. Changing the number of segments was an error prone and laborious process that demanded a remapping of the relationship between the user's mental image of the biologically important features of the model, on the one hand, and the implementation of this model in a digital computer, on the other. The use of x and nseg insulates the user from the most inconvenient aspects of such low-level details.

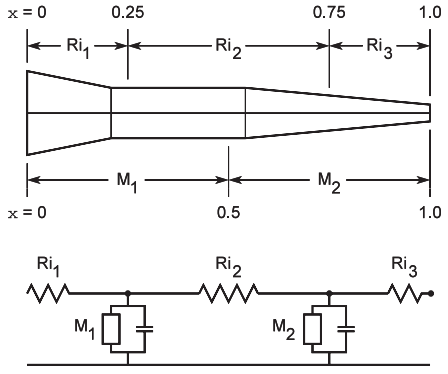

Figure 3.4. How the neurite of Fig. 3.3 would be represented by a section with two segments (nseg = 2). Now the electrical equivalent circuit (bottom) has two nodes. The membrane properties attached to the first and second nodes are based on neurite dimensions and biophysical parameters over the x intervals [0, 0.5] and [0.5, 1], respectively. The three axial resistances are computed from the cytoplasmic resistivity and neurite dimensions over the x intervals [0, 0.25], [0.25, 0.75], and [0.75, 1].

When nseg = 1 the entire section is lumped into a single compartment. This compartment has only one node, which is located midway along its length, i.e. at x = 0.5 (Fig. 3.3C and D). The integral of the surface area over the entire length of the section ($0 \le x \le 1$) is used to calculate the membrane properties associated with this node. The values of the axial resistors are determined by integrating the cytoplasmic resistivity along the paths from the ends of the section to its midpoint (dashed line in Fig. 3.3C). The left and right hand axial resistances of Fig. 3.3D are evaluated over the x intervals [0, 0.5] and [0.5, 1], respectively.

Fig. 3.4 shows what happens when nseg = 2. Now NEURON breaks the section 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].

3.3 Integration methods

Spatial discretization reduced the cable equation, a partial differential equation with derivatives in space and time, to a set of ordinary differential equations with first order derivatives in time. Selection of a method for numerical integration of these equations is guided by concerns of stability, accuracy, and efficiency (Hines and Carnevale 1995). To illustrate these important concepts and explain the rationale for the integrators used in NEURON, we turn first to the simplest approach for solving such equations numerically: explicit or forward Euler (which is NOT used in NEURON).

3.3.1 The forward Euler method: simple, unstable, inaccurate

Imagine a model of a neuron that has passive membrane, i.e. membrane conductance is constant and linear. The techniques that we use to understand and control error are immediately generalizable to the nonlinear case.

Suppose the model has only one compartment, so there is no axial current and the right hand side of Eq. 2 is zero. This equation can then be written as

$$\frac{dV}{dt} + kV = 0 \qquad (3)$$

where the constant k is the inverse of the membrane time constant. The analytic solution of Eq. 3 is

$$V(t) = V(0)e^{-kt} \qquad (4)$$

Let us compare this to the results of our computer methods.

The forward Euler method is based on a simple approximation. We know the initial value of the

dependent variable ($V(0)$, given by the initial conditions) and the initial slope of the solution ($-kV(0)$, given by Eq. 3). The approximation is to assume that the slope 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.

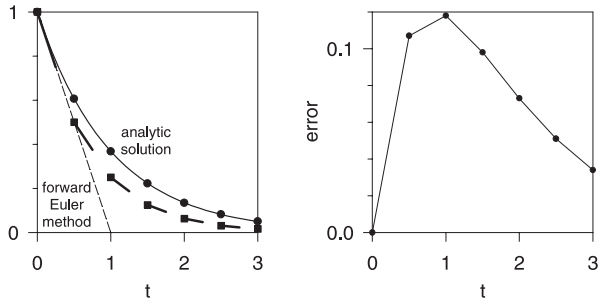

Figure 3.5. Left: comparison of analytic solution to Eq. 3 (solid line) with results of forward Euler method (filled squares) for $V(0) = 1$, $k = 1$, and $\Delta t = 0.5$. Right: absolute error of forward Euler method.

This is illustrated in the left panel of Fig. 3.5, where the initial condition is $V(0) = 1$, the rate parameter is $k = 1$, and the time interval over which we extrapolate is $\Delta t = 0.5$. To simulate the behavior of Eq. 3 we march forward by intervals of width Δt, assuming the 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 right panel of Fig. 3.5 shows the absolute difference between the analytic solution and the results of the forward Euler method. The error increases for the first few time steps, then decreases as the analytic and simulation solutions approach the same steady state ($V = 0$).

3.3.2 Numerical stability

What would happen if Eq. 3 were subjected to the forward Euler method with a very large time step, e.g. $\Delta t = 3$? 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. 3.6 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, the potentials in the two compartments rapidly converge toward each other (time constant 1/41), and then slowly decay toward 0 (time constant 1).

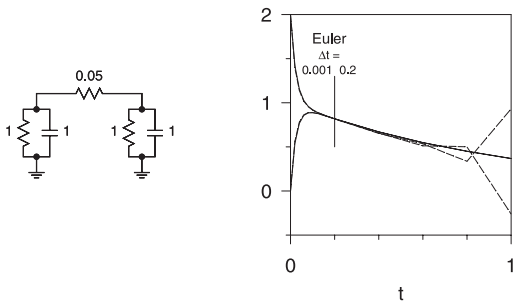

Figure 3.6. Left: The two compartments of this model are connected by a small axial resistance, so the membrane potentials are normally in quasi-equilibrium and at the same time are decaying fairly slowly. Right: The forward Euler method (dashed lines) is numerically unstable whenever Δt is greater than twice the smallest time constant. The analytic solution (thin lines) is the sum of two exponentials with time constants 1 and 1/41. The solution step size is 0.001 second for the first 0.2 second, after which it is increased to 0.2 second.

If we use the forward Euler method with $\Delta t = 0.5$, we realize that there will be a great deal of trouble during the time where the voltages are changing rapidly. We might therefore think that all we need to do is choose 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. 3.6. After 0.2 seconds with $\Delta t = 0.001$, the two voltages have nearly come into equilibrium. Then we changed to $\Delta t = 0.2$, which is small enough to follow the slow decay closely. Unfortunately what happens is that, no matter how small the difference between the voltages (even if it consists only of roundoff error), the difference grows geometrically at each time step. 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 amplitude of each of these patterns, or eigenvectors, is given by $e^{t\lambda_i}$, where λ_i is called the eigenvalue of the *i*th eigenvector. Each eigenvalue is the reciprocal of one of the time constants of the solutions to the differential equations that describe the system. The *i*th pattern decays exponentially to 0 if the real part of λ_i is negative; if the real part is positive, the amplitude grows catastrophically. If λ_i has an imaginary component, the pattern oscillates with frequency $\omega_i = \mathrm{Im}(\lambda_i)$.

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}.

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 as in the single compartment of Eq. 3. When the equations that describe such a system are solved numerically, the time step Δt must be small enough that the solution of each equation is stable. This is the reason 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.

A driving force may alter the time constants that describe a system, thereby changing its stability properties. 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 the time constants and their corresponding eigenvectors. The more closely it approximates a voltage source (perfect voltage clamp), the greater this effect will be.

3.3.3 The backward Euler method: 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.

$$V(t+\Delta t) = V(t) + \Delta t f\left(V(t+\Delta t), t+\Delta t\right) \qquad (5)$$

Equation 5 can be derived from Taylor's series truncated at the Δt term but with $t + \Delta t$ in place of t. Therefore this approach is called the implicit or backward Euler method.

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

$$V(t+\Delta t) = \frac{V(t)}{1+k\Delta t} \qquad (6)$$

Several iterations of Eq. 6 are shown in Fig. 3.7. Each step moves to a new point $(t_{i+i}, V(t_{i+1}))$ such that the slope there points back to the previous point $(t_i, V(t_i))$. If Δt is very large, the solution converges exponentially toward the steady state instead of oscillating with geometrically increasing amplitude.

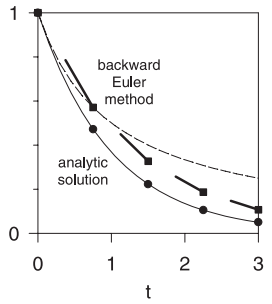

Figure 3.7. Comparison of analytic solution to Eq. 3 with results from backward Euler method (Eq. 6) for $V(0) = 1$, $k = 1$, and $\Delta t = 0.75$. At the end of each step the slope at the new value points back to the beginning of the step. The dashed line shows the voltage after the first time step as a function of Δt.

Applying the implicit method to the two compartment model demonstrates its attractive stability properties (Fig. 3.8). Notice that a large Δt gives a reasonable qualitative understanding of the behavior, even if it does not allow us to follow the initial rapid voltage changes. Furthermore the step size can be changed according to how quickly the states are changing, yet the solution remains stable.

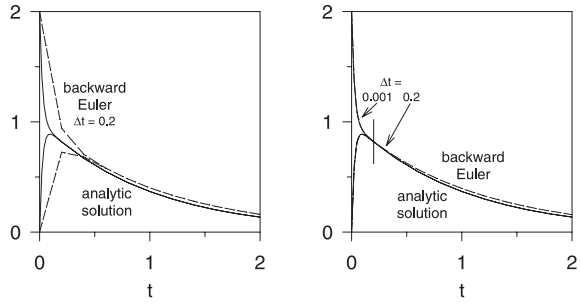

Figure 3.8. Two compartments as in Fig. 3.6 simulated with backward Euler method. Left: Δt = 0.2 second, much larger than the fast time constant. Right: for the first 0.2 second, Δt is small enough to accurately follow the fast time constant. Thereafter, Δt is increased to 0.2 second, yet the simulation remains numerically stable.

The backward Euler method requires the 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. The first order implicit method is practical for initial exploratory simulations because reasonable values of Δt produce fast simulations that are almost always qualitatively correct, and tightly coupled compartments do not generate large error oscillations but instead come quickly into equilibrium because of its robust stability properties.

3.3.4 Error

The total or global error 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 theorem truncated at the term proportional to Δt.

$$V(t+\Delta t) = V(t) + \Delta t V'(t+\Delta t) - \frac{\Delta t^2}{2} V''(t^*) \qquad (7)$$

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

Both the forward and backward Euler methods ignore second and higher order terms, so the error at each step is proportional to Δt^2. Integrating over a fixed time interval T requires $T/\Delta t$ steps, so the error that accumulates in this interval is on the order of $\Delta t^2 \cdot T/\Delta t$, i.e. the net error is proportional to Δt. Therefore we can always decrease the local error as much as we like by reducing Δt.

The second contribution to the total error comes from with 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 were to thereafter solve the equations exactly using the state values at t_1 as our initial condition, the future solution will be inaccurate because we are on a different trajectory.

The total error of the simulation is therefore not easy to analyze. In the example of Fig. 3.5, all trajectories end up at the same steady state so total error tends to decrease, but not all systems behave in this manner. Particularly treacherous are systems that behave chaotically so that, 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.

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 different from the set of trajectories defined by the error in our parameters. There may be some benefit in treating the model equations as sacred runes which must be solved to an arbitrarily high precision, insofar as removal of any source of error has value. Nevertheless, judgment is required in order to determine the meaning of a simulation run.

For example, consider the Hodgkin-Huxley membrane action potentials elicited by two current stimuli, one brief and strong and the other much weaker. The left panel of Fig. 3.9 compares the results of computing these action potentials by the backward Euler method using time steps of 25 and 5 μs. As noted above, shortening the time step decreases the simulation error. The effect is most noticeable for simulations involving the weaker stimulus: while the voltage hovers near threshold, a small error due to our time step grows into a large error in the time of occurrence of the spike.

However the behavior near threshold is highly sensitive to almost any parameter. This is demonstrated in the right of Fig. 3.9, where the sodium channel density is varied by only 1%. Clearly it is crucial to know the sensitivity of our results to every parameter of the model, and the time step is just one more parameter which is added as a condition of being able to simulate a model on the computer.

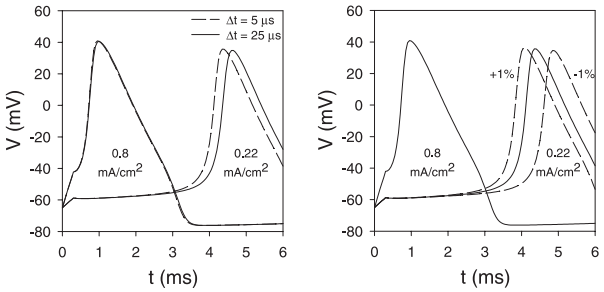

Figure 3.9. Backward Euler method simulations of Hodgkin Huxley membrane action potentials elicited by a current stimulus of duration 0.3 ms and amplitude 0.8 mA/cm^2 or 0.22 mA/cm^2. Left: Sensitivity to integration time step. The solid and dashed traces were computed with Δt = 25 and 5 μs, respectively. All action potentials were calculated with peak sodium conductance ($\bar{g}_{Na}$) 0.12 siemens/cm^2. Right: Sensitivity to $\bar{g}_{Na}$. All traces were computed with Δt = 5 μs. Peak sodium conductance was 0.12 siemens/cm^2 (solid lines) ± 1% (dashed lines). The three simulations that involved the large stimulus are indistinguishable in this plot.

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 nice to be able 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 many orders of magnitude better than the accuracy of our physiological measurements.

3.3.5 Crank-Nicholson Method: stable and more accurate

This motivates us to look into an integration strategy that combines the backward and forward Euler methods. The central difference or Crank-Nicholson method [Crank and Nicholson 1947], which is equivalent to advancing by one half step using backward Euler and then advancing by one half step using forward Euler, has global error proportional to the square of the step size. Fig. 3.10 illustrates the idea. The value at the end of each step is along a line determined by the estimated slope at the midpoint of the step.

Generally, for a given Δt we can expect a large accuracy increase with the Crank-Nicholson method. In fact, the simulation using the 0.75 second time step in Fig. 3.10 is much more accurate than the 0.5 second time step simulation with the forward Euler method (Fig. 3.5).

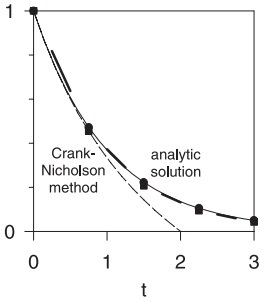

Figure 3.10. In the Crank-Nicholson method the slope at the midpoint of the step is used to determine the new value. The analytic and Crank-Nicholson solutions are almost indistinguishable in this figure. The dashed line shows the voltage after the first time step as a function of Δt.

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 $V(t + \Delta t/2)$ we just have

$$V(t+\Delta t) = 2V\left(t+\frac{\Delta t}{2}\right) - V(t) \qquad (8)$$

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

One might well ask what effect the forward Euler half step has on numerical stability. Fig. 3.11 shows the solution for the two compartment model of Fig. 5 computed using this central difference method, in which Δt was 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

$$V(t+\Delta t) = V(t)\frac{1-\frac{k\Delta t}{2}}{1+\frac{k\Delta t}{2}} \qquad (9)$$

As Δt gets very large, the step multiplier approaches –1 from above so the solution oscillates with decreasing amplitude.

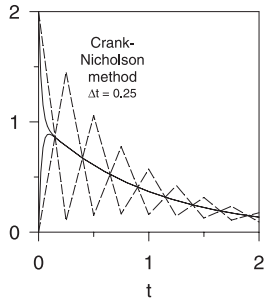

Figure 3.11. The Crank-Nicholson method can have significant error oscillations when there is a large amplitude component in the simulation that has a time constant much smaller than Δt. However, the oscillation amplitude decreases at each step, so the simulation is numerically stable.

Technically speaking the Crank-Nicholson method is stable because the error oscillations do decay with time. However, this example shows that it can produce artifactual large amplitude oscillations if the time step is too large. This can affect simulations of models that involve voltage clamps or in which adjacent segments are coupled by very small resistances.

3.3.6 The integration methods used in NEURON

The preceding discussion shows why NEURON offers the user a choice of two stable implicit integration methods: backward Euler, and a variant of Crank-Nicholson. Because of its robust numerical stability properties, backward Euler produces good qualitative results even with large time steps, and it works even if some or all of the equations are strictly algebraic relations among states. It can be used with extremely large time steps to find the steady-state solution for a linear ("passive") system. Backward Euler is therefore the default integrator used by NEURON.

When the global parameter `secondorder` is set to 2, a variant of the Crank-Nicholson method is used, which has numerical error proportional to Δt^2 and is therefore more 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 **3.3.7 Efficiency** below). This converts the current balance part of the problem to one that requires only the solution of simultaneous *linear* equations.

Although the Crank-Nicholson method is formally stable, it is sometimes plagued by spurious large amplitude oscillations (Fig. 3.11). This occurs when Δt is too large, as may occur in models that involve fast voltage clamps or that have compartments which are coupled by very small resistances. However, Crank-Nicholson is safe in most situations, and it can be much more efficient than backward Euler for a given accuracy.

These two methods are almost identical in terms of computational cost per time step (see **3.3.7 Efficiency** below). Since the current balance equations have the structure of a tree (there are no current loops), direct gaussian elimination is optimal for their solution (Hines 1984). This takes exactly the same number of computer operations as would be required for an unbranched cable with the same number of compartments.

For any particular problem, the best way to determine which is the method of choice is to compare both methods with several values of Δt to see which allows the largest Δt consistent with 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 the time constants and potentially requiring use of a smaller time step to avoid numerical oscillations in the Crank-Nicholson method. The more closely a signal source approximates a voltage source (perfect voltage clamp), the greater this effect will be.

3.3.7 Efficiency

Nonlinear equations generally need to be solved iteratively to maintain second order correctness. 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, still second order correct, that can be solved without iterations. A direct solution of the voltage equations at each time step $t \rightarrow t + \Delta t$ using the linearized membrane current $I(V,t) = G \cdot (V - E)$ 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 which can be computed using a separate time step that is 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.

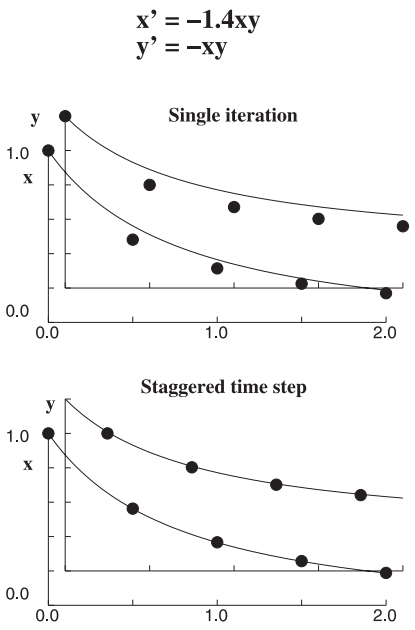

Figure 3.12. The equations shown here are computed using the Crank-Nicholson method. Top: $x(t + \Delta t)$ and $y(t + \Delta t)$ are determined using their values at time t. Bottom: staggered time steps yield decoupled linear equations. $y(t + \Delta t/2)$ is determined using $x(t)$, after which $x(t + \Delta t)$ is determined using $y(t + \Delta t/2)$.

Figure 3.12 contrasts this approach with the common technique of replacing nonlinear coefficients by their values at the beginning of a time step. For HH equations in a single compartment, the staggered time grid approach converts four simultaneous nonlinear equations at each time step to four independent linear equations that have the same order of accuracy at each time step. Since the voltage dependent rates use the voltage at the midpoint of the integration step, integration of channel states can be done analytically in just a single addition and multiplication operation and two table lookup operations. While this efficient scheme achieves second order accuracy, the tradeoff is that the tables depend on the value of the time step and must be recomputed whenever the time step changes.

Neuronal architecture can also be exploited to increase computational efficiency. Since neurons generally have a branched tree structure with no loops, the number of arithmetic operations required to solve the cable equation by Gaussian elimination is exactly the same as for an unbranched cable with the same number of compartments. That is, we need only O(N) arithmetic operations for the equations that describe N compartments connected in the form of a tree, even though standard Gaussian elimination generally takes $O(N^3)$ operations to solve N equations in N unknowns.

The tremendous efficiency increase results from the fact that, in a tree, one can always find a leaf compartment i that is connected to only one other compartment j, so that

$$a_{ii} V_i + a_{ij} V_j = b_i \tag{10a}$$

$$a_{ji} V_i + a_{jj} V_j + \left[\text{terms from other compartments}\right] = b_j \tag{10b}$$

In other words, the equation for compartment i (Eq. 10a) involves only the voltages in compartments i and j, and the voltage in compartment i appears only in the equations for compartments i and j (Eq. 10a and b). Using Eq. 10a to eliminate the V_i term from Eq. 10b, which requires O(1) (instead of N) operations, gives Eq. 11 and leaves N–1 equations in N–1 unknowns.

$$a'_{jj} V_j + \left[\text{terms from other compartments}\right] = b'_j \tag{11}$$

where $a'_{jj} = a_{jj} - \left(a_{ij} a_{ji} / a_{ii}\right)$

and $b'_j = b_j - \left(b_i a_{ji} / a_{ii}\right)$

This strategy can be applied until there is only one equation in one unknown.

Assume that we know the solution to these N–1 equations, and in particular that we know V_j. Then we can find V_i from Eq. 10a with O(1) step. Therefore the

effort to solve these N equations is O(1) plus the effort needed to solve N–1 equations. The number of operations required is independent of the branching structure, so a tree of N compartments uses exactly the same number of arithmetic operations as a one-dimensional cable of N compartments.

Efficient Gaussian elimination requires an ordering that can be found by a simple algorithm: choose the equation with the current minimum number of terms as the equation to use in the elimination step. This minimum degree ordering algorithm is commonly employed in standard sparse matrix solver packages. For example, NEURON's "Matrix" class uses the matrix library written by Stewart and Leyk (1994). This and many other sparse matrix packages are freely available at `http://www.netlib.org`.

4. THE NEURON SIMULATION ENVIRONMENT

No matter how powerful and robust its computational engine may be, the real utility of any software tool depends largely on its ease of use. Therefore a great deal of effort has been invested in the design of the simulation environment provided by NEURON. In this section we first briefly consider general aspects of the high-level language used for writing NEURON programs. Then we turn to an example of a model of a nerve cell to introduce specific aspects of the user environment, after which we cover these features more thoroughly.

4.1 The hoc interpreter

NEURON incorporates a programming language based on `hoc`, a floating point calculator with C-like syntax described by Kernighan and Pike (1984). This interpreter has been extended by the addition of object-oriented syntax (not including polymorphism or inheritance) that can be used to implement abstract data types and data encapsulation. Other extensions include functions that are specific to the domain of neural simulations, and functions that implement a graphical user interface (see below).

With `hoc` one can quickly write short programs that meet most problem-specific needs. The interpreter is used to execute simulations, customize the user interface, optimize parameters, analyze experimental data, calculate new variables such as impulse propagation velocity, etc..

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.

NEURON provides a built-in implementation of the microemacs text editor. Since the choice of a programming editor is highly personal, NEURON will also accept `hoc` code in the form of straight ASCII files created with any other editor.

4.2 A specific example

In the following example we show how NEURON might be used to model the cell in the top of Fig. 4.1. Comments in the `hoc` code are preceded by double slashes (`//`), and code blocks are enclosed in curly brackets (`{}`). Because the model is described in a piecewise fashion and many of the code specimens given below are meant to illustrate other features of NEURON, for the sake of clarity a fully commented and complete listing of the `hoc` code for this model is contained in **Appendix 1**.

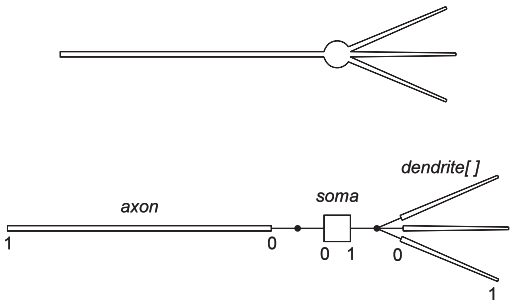

Figure 4.1. Top: cartoon of a neuron with a soma, three dendrites, and an unmyelinated axon (not to scale). The soma diameter is 50 µm. Each dendrite is 200 µm long and tapers uniformly along its length from 10 µm diameter at the soma to 3 µm at its distal end. The unmyelinated cylindrical axon is 1000 µm long and has a diameter of 1 µm. An electrode (not shown) is inserted into the soma for intracellular injection of a stimulating current. Bottom: topology of a NEURON model that represents this cell.

4.2.1 First step: establish model topology

One very important feature of NEURON is that it allows the user to think about models in terms that are familiar to the neurophysiologist, keeping numerical issues (e.g. number of spatial segments) entirely separate from the specification of morphology and biophysical properties. As noted in a previous section (**3.2 Spatial discretization . . .**), this separation is achieved through the use of one-dimensional cable "sections" as the basic building block from which model cells are constructed. These sections can be connected together to form any kind of branched cable and endowed with properties which may vary with position along their length.

The idealized neuron in Fig. 4.1 has several anatomical features whose existence and spatial relationships we want the model to include: a cell body (soma), three dendrites, and an unmyelinated axon. The following `hoc` code sets up the basic topology of the model:

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

The program starts by creating named sections that correspond to the important anatomical features of the cell. These sections are attached to each other using `connect` statements. As noted previously, each section has a normalized position parameter `x` which ranges from 0 at one end to 1 at the other. Because the axon and dendrites arise from opposite sides of the cell body, they are connected to the 0 and 1 ends of the `soma` section (see bottom of Fig. 4.1). A child section can be attached to any location on the parent, but attachment at locations other than 0 or 1 is generally employed only in special cases such as spines on dendrites.

4.2.2 Second step: assign anatomical and biophysical properties

Next we set the anatomical and biophysical properties of each section. Each section has its own segmentation, length, and diameter parameters, so it is necessary to indicate which section is being referenced. There are several ways to declare which is the currently accessed section, but here the most convenient is to precede blocks of statements with the appropriate section name.

```
soma {
   nseg = 1
   L = 50           // [µm] length
   diam = 50        // [µm] diameter
   insert hh        // HH currents
   gnabar_hh = 0.5*0.120   // [S/cm²]
}
axon {
   nseg = 20
   L = 1000
   diam = 1
   insert hh
}
for i=0,2 dendrite[i] {
   nseg = 5
   L = 200
   diam(0:1) = 10:3        // tapers
   insert pas// passive current
   e_pas = -65      // [mv] eq potential
   g_pas = 0.001    // [S/cm²]
}
```

The fineness of the spatial grid is determined by the compartmentalization parameter `nseg` (see **3.2 Spatial discretization . . .**). Here the soma is lumped into a single compartment (`nseg = 1`), while the axon and each of the dendrites are broken into several subcompartments (`nseg = 20` and `5`, respectively).

In this example, we specify the geometry of each section by assigning values directly to section length and diameter. This creates a "stylized model." Alternatively, one can use the "3-D method," in which NEURON computes section length and diameter from a list of (x, y, z, diam) measurements (see **4.5 Specifying geometry: stylized vs. 3-D**).

Since the axon is a cylinder, the corresponding section has a fixed diameter along its entire length. The spherical soma is represented by a cylinder with the same surface area as the sphere. The dimensions and electrical properties of the soma are such that its membrane will be nearly isopotential, so the cylinder approximation is not a significant source of error. If chemical signals such as intracellular ion concentrations were important in this model, it would be necessary to approximate not only the surface area but also the volume of the soma.

Unlike the axon, the dendrites become progressively narrower with distance from the soma. Furthermore, unlike the soma, they are too long to be lumped into a single compartment with constant diameter. The taper of the dendrites is accommodated by assigning a sequence of decreasing diameters to their segments. This is done through the use of "range variables," which are discussed below (**4.4 Range variables**).

In this model the soma and axon contain Hodgkin-Huxley (HH) sodium, potassium, and leak channels (Hodgkin and Huxley 1952), while the dendrites have constant, linear ("passive") ionic conductances. The `insert` statement assigns the biophysical mechanisms that govern electrical signals in each section. Particular values are set for the density of sodium channels on the soma (`gnabar_hh`) and for the ionic conductance and equilibrium potential of the passive current in the dendrites (`g_pas` and `e_pas`). More information about membrane mechanisms is presented in a later section (**4.6 Density mechanisms and point processes**).

4.2.3 Third step: attach stimulating electrodes

This code emulates the use of an electrode to inject a stimulating current into the soma by placing a current pulse stimulus in the middle of the soma section. The stimulus starts at `t` = 1 ms, lasts for 0.1 ms, and has an amplitude of 60 nA.

```
objref stim
// put stim in middle of soma
soma stim = new Iclamp(0.5)
stim.del = 1        // [ms] delay
stim.dur = 0.1      // [ms] duration
stim.amp = 60// [nA] amplitude
```

The stimulating electrode is an example of a point process. Point processes are discussed in more detail below (**4.6 Density mechanisms and point processes**).

4.2.4 Fourth step: control simulation time course

At this point all model parameters have been specified. All that remains is to define the simulation parameters, which govern the time course of the simulation, and write some code that executes the simulation.

This is generally done in two procedures. The first procedure initializes the membrane potential and the states of the inserted mechanisms (channel states, ionic concentrations, extracellular potential next to the membrane). The second procedure repeatedly calls the built-in single step integration function `fadvance()` and saves, plots, or computes functions of the desired output variables at each step. In this procedure it is possible to change the values of model parameters during a run.

The built-in function `finitialize()` initializes time `t` to 0, membrane potential `v` to -65 mv throughout the model, and the HH state variables m, n and h to their steady state values at `v` = –65 mv. Initialization can also be performed with a user-written routine if there are special requirements that `finitialize()` cannot accommodate, such as nonuniform membrane potential.

```
dt = 0.05     // [ms] time step
tstop = 5     // [ms]

// initialize membrane potential,
//   state variables, and time
finitialize(-65)

proc integrate() {
       // show somatic Vm at t=0
       print t, soma.v(0.5)
       while (t < tstop) {
              // advance solution by dt
              fadvance()
              // function calls to save
              //   or plot results
              //   would go here
              // show time and soma Vm
              print t, soma.v(0.5)
              // statements that change
              //   model parameters
              //   would go here
       }
}
```

Both the integration time step `dt` and the solution time `t` are global variables. For this example `dt` = 50 µs. The `while(){}` statement repeatedly calls `fadvance()`, which integrates the model equations over the interval `dt` and increments `t` by `dt` on each call. For this example, the time and somatic membrane potential are displayed at each step. This loop exits when `t` ≥ `tstop`.

The entire listing of the `hoc` code for the model is printed in **Appendix 1**. When this program is first processed by the NEURON interpreter, the model is set up and initiated but the `integrate()` procedure is not executed. When the user enters an `integrate()` statement in the NEURON interpreter window, the simulation advances for 5 ms using 50 µs time steps.

4.3 Section variables

Three parameters apply to the section as a whole: cytoplasmic resistivity `Ra` (Ω cm), the section length `L`, and the compartmentalization parameter `nseg`. The first two are "ordinary" in the sense that they do not

affect the structure of the equations that describe the model. Note that the `hoc` code specifies values for `L` but not for `Ra`. This is because each section in a model is likely to have a different length, whereas the cytoplasm (and therefore `Ra`) is usually assumed to be uniform throughout the cell. The default value of `Ra` is 35.4 Ω cm, which is appropriate for invertebrate neurons. Like `L` it can be assigned a new value in any or all sections (e.g. ~200 Ω cm for mammalian neurons).

The user can change the compartmentalization parameter `nseg` without having to modify any of the statements that set anatomical or biophysical properties. However, if parameters vary with position in a section, care must be taken to ensure that the model incorporates the spatial detail inherent in the parameter description.

4.4 Range variables

Like dendritic diameter in our example, most cellular properties are functions of the position parameter `x`. NEURON has special provisions for dealing with these properties, which are called "range variables." Other examples of range variables include the membrane potential `v`, and ionic conductance parameters such as the maximum HH sodium conductance `gnabar_hh` (siemens / cm^2).

Range variables enable the user to separate property specification from segment number. A range variable is assigned a value in one of two ways. The simplest and most common is as a constant. For example, the statement `axon.diam = 10` asserts that the diameter of the axon is uniform over its entire length.

Properties that change along the length of a section are specified with the syntax `rangevar(`*`xmin`*`:`*`xmax`*`) = `*`e1`*`:`*`e2`*. The four italicized symbols are expressions with *`e1`* and *`e2`* being the values of the property at *`xmin`* and *`xmax`*, respectively. The position expressions must meet the constraint $0 \le xmin \le xmax \le 1$. Linear interpolation is used to assign the values of the property at the segment centers that lie in the position range [*`xmin`*, *`xmax`*]. In this manner a continuously varying property can be approximated by a piecewise linear function. If the range variable is diameter, neither *`e1`* nor *`e2`* should be 0, or the corresponding axial resistance will be infinite.

In our model neuron, the simple dendritic taper is specified by `diam(0:1) = 10:3` and `nseg = 5`. This results in five segments that have centers at `x` = 0.1, 0.3, 0.5, 0.7 and 0.9 and diameters of 9.3, 7.9, 6.5, 5.1 and 3.7, respectively.

To underscore the relationship between parameter ranges, segment centers, and the values that are assigned to range variables, it may be helpful to consider a pair of examples. The second column of Table 4.1 shows the values of `x` at which segment centers would be located for `nseg` = 1, 2, 3 and 5. The third column shows the corresponding diameters of these segments for a dendrite with a diameter of 10 µm over the first 60% of its length and 14 µm over the remaining 40% of its length. The diameter that is assigned to each segment depends entirely on whether the segment center lies in the `x` interval that corresponds to 10 µm or 14 µm.

`nseg`	segment centers `x`	 see Note 1	 see Note 2
1	0.5	10	13
2	0.25	10	10.5
	0.75	14	14
3	0.1667	10	10
	0.5	10	13
	0.8333	14	14
5	0.1	10	10
	0.3	10	11
	0.5	10	13
	0.7	14	14
	0.9	14	14

Note 1
```
diam(0:0.6)=10:10
diam(0.6:1)=14:14
```

Note 2
```
diam(0:0.2)=10:10
diam(0.6:1)=14:14
diam(0.2:0.6)=10:14
```

Table 4.1. Diameter as a range variable: effects of `nseg` on segment diameters

The fourth column illustrates interpolation of segment diameters. Here the dendrite starts with a diameter of 10 µm over the first 20% of its length, has a linear flare from 10 to 14 µm over the next 40%, and ends with a diameter of 14 µm over the last 40% of its length. The segments that have centers in the interval [0.2, 0.6] are assigned diameters that are interpolated.

The value of a range variable at the center of a segment can appear in any expression using the syntax `rangevar(x)` in which $0 \le x \le 1$. The value returned is the value at the center of the segment containing `x`, NOT the linear interpolation of the values stored at the

centers of adjacent segments. If the parentheses are omitted, the position defaults to a value of 0.5 (middle of the section).

A special form of the `for` statement is available: `for (var) stmt`. For each value of the normalized position parameter `x` that defines the center of each segment in the selected section (along with positions 0 and 1), this statement assigns `var` that value and executes the `stmt`. This `hoc` code would print the membrane potential as a function of physical position (in μm) along the axon:

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

4.5 Specifying geometry: stylized vs. 3-D

As noted above (**4.2.2 Second step . . .**), there are two ways to specify section geometry. Our example uses the stylized method, which simply assigns values to section length and diameter. This is most appropriate when cable length and diameter are authoritative and 3-D shape is irrelevant.

It is best to use the 3-D method if the model is based on anatomical reconstruction data (quantitative morphometry), or if 3-D visualization is paramount. This approach keeps the anatomical data in a list of (x, y, z, diam) "points." 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 **pt3d list**, which is the authoritative definition of the shape of the section, automatically determines the length and diameter of the section.

When the pt3d list is non-empty, the shape model used for a section is a sequence of frusta. The pt3d points define the locations and diameters of the ends of these frusta. The effective area, diameter, and resistance of each segment are computed from this sequence of points by trapezoidal integration along the segment length. 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 same coordinates but different diameters), which add 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 a shape has nothing to do with `nseg` and does not affect simulation speed.

When diameter varies along the length of a section, the stylized and 3-D approaches to defining section structure can lead to very different model representations, even when the specifications might seem to be identical. Imagine a cell whose diameter varies with position `x` as shown in Fig. 4.2

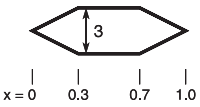

Figure 4.2. A hypothetical cell whose structure is to be approximated using the stylized and 3-D approaches.

This program contrasts the results of using the stylized and 3-D approaches to emulate the structure of this cell. It creates sections `stylized_model` and `three_d_model`, whose geometries are specified using the stylized and 3-D methods, respectively. For each segment of these two sections, it prints out the `x` location of the segment center, the segment diameter and surface area, and the axial resistance `ri` (in megohms) between the segment center and the center of the parent segment (i.e. the segment centered at the next smaller `x`; see the Ri in Fig. 3.3).

```
/* diampt3d.hoc */

create stylized_model, three_d_model

// set nseg and L for both models
forall {
   nseg = 5
   L = 1
}

stylized_model {
   diam(0:0.3) = 0:3
   diam(0.3:0.7) = 3:3
   diam(0.7:1) = 3:0
}

three_d_model {
   pt3dadd(0,0,0,0)
   pt3dadd(0.3,0,0,3)
   pt3dadd(0.7,0,0,3)
   pt3dadd(1,0,0,0)
}

forall {
   print secname()
   for (x) print x,diam(x),area(x),ri(x)
}
```

The output of this program is presented in Table 4.2. The stylized approach creates a section

composed of a series of cylindrical segments whose diameters are interpolated from the range variable specification of diameter (left panel of Fig. 4.3). The surface areas and axial resistances associated with these cylinders are based entirely on their cylindrical dimensions.

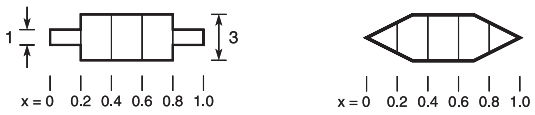

Figure 4.3. Left: The stylized representation of the hypothetical cell shown in Fig. 4.2. Right: In the 3-D approach, the diameter, surface area, and axial resistance of each segment are based on the integrals of these quantities over the corresponding interval in the original anatomy.

The 3-D approach, however, produces a model that is quite different. The reported diameter of each segment, `diam(x)`, is the average diameter over the corresponding length of the original anatomy, and the segment area, `area(x)`, is the integral of the surface area. Therefore `area(x)` is not necessarily equal to `π diam(x) L / nseg`. The axial resistances are computed by integrating the resistance of the cytoplasm along the path between the centers of adjacent segments. Both `ri(0)` and `ri(1)` are effectively infinite because the diameter of the volume elements along the integration path tapers to 0 at both ends of the 3-D data set.

Close examination of Table 4.2 reveals two items that require additional comment. The first item is that `ri(0)` = 10^{30} for both models. This is because, regardless of whether the stylized or the 3-D approach is used, the left hand side of the section is a "sealed end" or open circuit.

The second noteworthy item, which at first seems unexpected, is that even though the diameter is specified to be 0 at `x` = 0 and 1, the model generated by the 3-D approach reports nonzero values for diam(0) and diam(1). This reflects the fact that, except for membrane potential `v`, accessing the value of a range variable at `x` = 0 or 1 returns the value at the center of the first and last segment, respectively (i.e. at `x` = 0.5/`nseg` and 1 – 0.5/`nseg`). The technical reason for this behavior is that diameter is part of a morphology mechanism, and mechanisms exist only in the interior of a section. As noted in **4.4 Range variables**, `range_variable(x)` returns the value of the range variable at the center of the segment that contains `x`. The only range variable that exists at the 0 and 1 locations is `v`; other range variables are undefined at these points (`area()` and `ri()` are functions, not range variables). Point processes, which are discussed next, are not accessed as functions of position and can be placed anywhere in the interval $0 \le x \le 1$.

4.6 Density mechanisms and point processes

The `insert` statement assigns biophysical mechanisms, which govern electrical and (if present) chemical signals, to a section. Many sources of electrical and chemical signals are distributed over the membrane of the cell. These **density mechanisms** are described in terms of current per unit area and conductance per unit area; examples include voltage-gated ion channels such as the HH currents.

However, density mechanisms are not the most appropriate representation of all signal sources. Synapses and electrodes are best described in terms of localized absolute current in nanoamperes and conductance in microsiemens. These are called **point processes**.

An object syntax is used to manage the creation, insertion, attributes, and destruction of point processes. For example, a current clamp (electrode for injecting a current) is created by declaring an object variable and

	Stylized model			3-D model		
x	diam(x)	area(x)	ri(x)	diam(x)	area(x)	ri(x)
0	1	0	1e+30	1	0	1e+30
0.1	1	0.628318	0.0450727	1	3.20381	4.50727e+14
0.3	3	1.88495	0.0500808	2.75	4.94723	0.0300485
0.5	3	1.88495	0.0100162	3	1.88495	0.0100162
0.7	3	1.88495	0.0100162	2.75	4.94723	0.0100162
0.9	1	0.628318	0.0500808	1	3.20381	0.0300485
1.0	1	0	0.0450727	1	0	4.50727e+14

Table 4.2. Diameter as a range variable: differences between models created by stylized and 3-D methods for specifying section geometry.

assigning it a new instance of the IClamp object class (see **4.2.3 Third step: attach stimulating electrodes**). When a point process is no longer referenced by any object variable, the point process is removed from the section and destroyed. In our example, redeclaring `stim` with the statement `objref stim` would destroy the pulse stimulus, since no other object variable is referencing it.

The `x` position specified for a point process can have any value in the range [0,1]. If `x` is specified to be at 0 or 1, it 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 whichever segment contains `x`. For example, in a section with `nseg` = 5 the segment centers (internal nodes) are located at `x` = 0.1, 0.3, 0.5, 0.7 and 0.9. Point processes whose specified locations are 0.04 and 0.41 would be assigned by NEURON to the nodes at 0.1 and 0.5, respectively. 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 the centers of segments. To completely avoid `nseg`-dependent shifts of the `x` locations of point processes, one can choose sections with lengths such that the point processes are located at the 0 or 1 ends of sections.

The location of a point process can be changed with no effect on its other attributes. In our example the statement `dendrite[2] stim.loc(1)` would move the current stimulus to the distal end of the third dendrite.

If a section's `nseg` is changed, that section's point processes 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 user-defined density mechanisms and point processes can be simultaneously present in each compartment of a neuron. One important difference between density mechanisms and point processes is that any number of the same kind of point process can exist at the same location.

User-defined density mechanisms and point processes can be linked into NEURON using 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 then constructs the appropriate C program which is compiled and becomes available for use in NEURON. This program 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. An extensive discussion of NMODL is beyond the scope of this article, but its major advantages can be listed succinctly.

1. Interface details to NEURON are handled automatically — and there are a great many such details.
   - 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 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 need to 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 example, 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, such as Runge-Kutta or backward Euler.
5. Function tables can be generated automatically for efficient computation of complicated expressions.

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

4.7 Graphical interface

The user is not limited to operating within the traditional "code-based command-mode environment." Among its many extensions to hoc, NEURON includes functions for implementing a fully graphical, windowed interface. Through this interface, and without having to write any code at all, the user can effortlessly create and arrange displays of menus, parameter value editors, graphs of parameters and state variables, and views of the model neuron. Anatomical views, called "space plots," can be explored, revealing what mechanisms and point processes are present and where they are located.

The purpose of NEURON's graphical interface is to promote a match between what the user thinks is inside the computer, and what is actually there. These visualization enhancements are a major aid to maintaining conceptual control over the simulation because they provide immediate answers to questions about what is being represented in the computer.

The interface has no provision for constructing neuronal topology, a conscious design choice based on the strong likelihood that a graphical toolbox for building neuronal topologies would find little use. Small models with simple topology are so easily created in hoc that a graphical topology editor is unnecessary. More complex models are too cumbersome to deal with using a graphical editor. It is best to express the topological specifications of complex stereotyped models through algorithms, written in hoc, that generate the topology automatically. Biologically realistic models often involve hundreds or thousands of sections, whose dimensions and interconnections are contained in large data tables generated by hours of painstaking quantitative morphometry. These tables are commonly read by hoc procedures that in turn create and connect the required sections without operator intervention.

The basic features of the graphical interface and how to use it to monitor and control simulations are discussed elsewhere (Moore and Hines 1996). However, several sophisticated analyis and simulation tools that have special utility for nerve simulation are worthy of mention.

- The "Function Fitter" optimizes a parameterized mathematical expression to minimize the least squared difference between the expression and data.
- The "Run Fitter" allows one to optimize several parameters of a complete neuron model to experimental data. This is most useful in the context of voltage clamp data which is contaminated by incomplete space clamp or models that cannot be expressed in closed form, such as kinetic schemes for channel conductance.
- The "Electrotonic Workbench" plots small signal input and transfer impedance and voltage attenuation as functions of space and frequency (Carnevale et al. 1996). These plots include the neuromorphic (Carnevale et al. 1995) and L vs. x (O'Boyle et al. 1996) renderings of the electrotonic transformation (Brown et al. 1992; Tsai et al. 1994b; Zador et al. 1995). By revealing the effectiveness of signal transfer, the Workbench quickly provides insight into the "functional shape" of a neuron.

All interaction with these and other tools takes place in the graphical interface and no interpreter programming is needed to use them. However, they are constructed entirely within the interpreter and can be modified when special needs require.

4.8 Object-oriented syntax

4.8.1 Neurons

It is often convenient to deal with groups of sections that are related. Therefore NEURON provides a data class called a **SectionList** that can be used to identify subsets of sections. Section lists fit nicely with the "regular expression" method of selecting sections, used in earlier implementations of NEURON, in that

1. the section list is easily constructed by using regular expressions to add and delete sections
2. after the list is constructed it is available for reuse
3. it is much more efficient to loop over the sections in a section list than to pick out the sections accepted by a combination of regular expressions

This code

```
objref alldend
alldend = new SectionList()
forsec "dend" alldend.append()
forsec alldend print secname()
```

forms a list of all the sections whose names contain the string "dend" and then iterates over the list, printing the name of each section in it. For the example program

presented in this report, this would generate the following output in the NEURON interpreter window

```
dendrite[0]
dendrite[1]
dendrite[2]
```

although in this very simple example it would clearly have been easy enough to loop over the array of dendrites directly, e.g.

```
for i = 0,2 {
        dendrite[i] print secname()
}
```

4.8.2 Networks

To help the user manage very large simulations, the interpreter syntax has been extended to facilitate the construction of hierarchical objects. This is illustrated by the following code fragment, which specifies a pattern for a simple stylized neuron consisting of three dendrites connected to one end of a soma and an axon connected to the other end.

```
begintemplate Cellx
  public soma, dendrite, axon
  create soma, dendrite[3], axon
  proc init() {
    for i=0,2 {
      connect dendrite[i](0), soma(0)
    }
    connect axon(0), soma(1)
    axon insert hh
  }
endtemplate Cellx
```

Whenever a new instance of this pattern is created, the `init()` procedure automatically connects the `soma`, `dendrite`, and `axon` sections together. A complete pattern would also specify default membrane properties as well as the number of segments for each section.

Names that can be referenced outside the pattern are listed in the `public` statement. In this case, since `init` is not in the list, the user could not re-initialize by calling the `init()` procedure. Public names are referenced through a dot notation.

The particular benefit of using templates (“classes” in standard object oriented terminology) is the fact that they can be employed to create any number of instances of a pattern. For example,

```
objref cell[10][10]
for i=0,9 {
   for j=0,9 cell[i][j]=new Cellx()
}
```

creates an array of 100 objects of type `Cellx` that can be referenced individually via the object variable `cell`. In this example,

```
cell[4][5].axon.gnabar_hh(0.5)
```

is the value of the maximum HH sodium conductance in the middle of the axon of `cell[4][5]`.

As this example implies, templates offer a natural syntax for the creation of networks. However it is entirely up to the user to logically organize the templates in such a way that they appropriately reflect the structure of the problem. Generally, any given structural organization can be viewed as a hierarchy of container classes, such as cells, microcircuits, layers, or networks. The important issue is how much effort is required for the concrete network representation to support a range of logical views of the same abstract network. A logical view that organizes the cells differently may not be easy to compute if the network is built as an elaborate hierarchy. This kind of pressure tends to encourage relatively flat organizations that make it easier to implement functions that search for specific information. The bottom line is that network simulation design remains an ad hoc process that requires careful programming judgement.

One very important class of logical views that are not generally organizable as a hierarchy are those of synaptic organization. In connecting cells with synapses, one is often driven to deal with general graphs, which is to say, no structure at all.

In addition to the notions of classes and objects (a synapse is an object with a pre- and a postsynaptic logical connection) the interpreter offers one other fundamental language feature that can be useful in dealing with objects that are collections of other objects. This is the notion of “iterators,” taken from the Sather programming language (Murer et al. 1996). This is a separation of the process of iteration from that of “what is to be done for each item.” If a programmer implements one or more iterators in a collection class, the user of the class does not need to know how the class indexes its items. Instead the class will return each item in turn for execution in the context of the loop body. This allows the user to write

```
for layer2.synapses(syn, type) {
 // Statements that manipulate the
 // object reference named "syn".
 // The iterator causes "syn" to refer,
 // in turn, to each synapse of a
 // certain type in the layer2 object.
}
```


without being aware of the possibly complicated process of picking out these synapses from the layer (that is the responsibility of the author of the class of which `layer2` is an instance).

It is to be sadly emphasized that these kinds of language features, though very useful, do not impose any policy with regard to the design decisions users must make in building their networks. Different programmers express very different designs on the same language base, with the consequence that it is more often than not infeasible to reconcile slightly different representations of even very similar concepts.

An example of a useful way to deal uniformly with the issue of synaptic connectivity is the policy implemented in NEURON by Lytton (1996). This implementation uses the normal NMODL methodology to define a synaptic conductance model and enclose it within a framework that manages network connectivity.

5. SUMMARY

The recent striking expansion in the use of simulation tools in the field of neuroscience has been encouraged by the rapid growth of quantitative observations that both stimulate and constrain the formulation of new hypotheses of neuronal function, and enabled by the availability of ever-increasing computational power at low cost. These factors have motivated the design and implementation of NEURON, the goal of which is to provide a powerful and flexible environment for simulations of individual neurons and networks of neurons. NEURON has special features that accommodate the complex geometry and nonlinearities of biologically realistic models, without interfering with its ability to handle more speculative models that involve a high degree of abstraction.

As we note in this paper, one particularly advantageous feature is that the user can specify the physical properties of a cell without regard for the strictly computational concern of how many compartments are employed to represent each of the cable sections. In a future publication we will examine how the NMODL translator is used to define new membrane channels and calculate ionic concentration changes. Another will describe the Vector class. In addition to providing very efficient implementations of frequently needed operations on lists of numbers, the vector class offers a great deal of programming leverage, especially in the management of network models.

NEURON source code, executables, and documents are available at

`http://neuron.duke.edu`

and

`http://www.neuron.yale.edu`

and by ftp from

`ftp.neuron.yale.edu`.

ACKNOWLEDGMENTS

We wish to thank John Moore, Zach Mainen, Bill Lytton, David Jaffe, and the many other users of NEURON for their encouragement, helpful suggestions, and other contributions. This work was supported by NIH grant NS 11613 (“Computer Methods for Physiological Problems”) to MLH and by the Yale Neuroengineering and Neuroscience Center (NNC).

REFERENCES

Bernander, O., 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.

Brown, T.H., Zador, A.M., Mainen, Z.F., and Claiborne, B.J. Hebbian computations in hippocampal dendrites and spines. In: *Single Neuron Computation*, edited by T. McKenna, J. Davis, and S.F. Zornetzer. San Diego: Academic Press, 1992, p. 81-116.

Carnevale, N.T. and Rosenthal, S. Kinetics of diffusion in a spherical cell: I. No solute buffering. *J. Neurosci. Meth.* 41:205-216, 1992.

Carnevale, N.T., Tsai, K.Y., Claiborne, B.J., and Brown, T.H. The electrotonic transformation: a tool for relating neuronal form to function. In: *Advances in Neural Information Processing Systems*, vol. 7, edited by G. Tesauro, D.S. Touretzky, and T.K. Leen. Cambridge, MA: MIT Press, 1995, p. 69-76.

Carnevale, N.T., Tsai, K.Y., and Hines, M.L. The Electrotonic Workbench. *Society for Neuroscience Abstracts* 22:1741, 1996.

Cauller, L.J. and Connors, B.W. Functions of very distal dendrites: experimental and computational studies of layer I synapses on neocortical pyramidal cells. In: *Single Neuron Computation*, edited by T. McKenna, J. Davis, and S.F. Zornetzer. San Diego: Academic Press, 1992, p. 199-229.

Destexhe, A., Babloyantz, A., and Sejnowski, T.J. Ionic mechanisms for intrinsic slow oscillations in thalamic relay neurons. *Biophys. J.* 65:1538-1552, 1993a.

Destexhe, A., Contreras, D., Sejnowski, T.J., and Steriade, M. A model of spindle rhythmicity in the isolated thalamic reticular nucleus. *J. Neurophysiol.* 72:803-818, 1994.

Destexhe, A., Contreras, D., Steriade, M., Sejnowski, T.J., and Huguenard, J.R. In vivo, in vitro and computational analysis of dendritic calcium currents in thalamic reticular neurons. *J. Neurosci.* 16:169-185, 1996.

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, 1993b.

Destexhe, A. and Sejnowski, T.J. G-protein activation kinetics and spill-over of GABA may account for differences between inhibitory responses in the hippocampus and thalamus. *Proc. Nat. Acad. Sci.* 92:9515-9519, 1995.

Häusser, M., Stuart, G., Racca, C., and Sakmann, B. Axonal initiation and active dendritic propagation of action potentials in substantia nigra neurons. *Neuron* 15:637-647, 1995.

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

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. The NEURON simulation program. In: *Neural Network Simulation Environments*, edited by J. Skrzypek. Norwell, MA: Kluwer, 1994, p. 147-163.

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. and Shrager, P. A computational test of the requirements for conduction in demyelinated axons. *J. Restor. Neurol. Neurosci.* 3:81-93, 1991.

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.

Hsu, H., Huang, E., Yang, X.-C., Karschin, A., Labarca, C., Figl, A., Ho, B., Davidson, N., and Lester, H.A. Slow and incomplete inactivations of voltage-gated channels dominate encoding in synthetic neurons. *Biophys. J.* 65:1196-1206, 1993.

Jaffe, D.B., Ross, W.N., Lisman, J.E., Miyakawa, H., Lasser-Ross, N., and Johnston, D. A model of dendritic Ca2+ accumulation in hippocampal pyramidal neurons based on fluorescence imaging experiments. *J. Neurophysiol.* 71:1065-1077, 1994.

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

Lindgren, C.A. and Moore, J.W. Identification of ionic currents at presynaptic nerve endings of the lizard. *J. Physiol.* 414:210-222, 1989.

Lytton, W.W. Optimizing synaptic conductance calculation for network simulations. *Neural Computation* 8:501-509, 1996.

Lytton, W.W., Destexhe, A., and Sejnowski, T.J. Control of slow oscillations in the thalamocortical neuron: a computer model. *Neurosci.* 70:673-684, 1996.

Lytton, W.W. and Sejnowski, T.J. Computer model of ethosuximide's effect on a thalamic neuron. *Ann. Neurol.* 32:131-139, 1992.

Mainen, Z.F., Joerges, J., Huguenard, J., and Sejnowski, T.J. A model of spike initiation in neocortical pyramidal neurons. *Neuron* 15:1427-1439, 1995.

Mainen, Z.F. and Sejnowski, T.J. Reliability of spike timing in neocortical neurons. *Science* 268:1503-1506, 1995.

Mascagni, M.V. Numerical methods for neuronal modeling. In: *Methods in Neuronal Modeling*, edited by C. Koch and I. Segev. Cambridge, MA: MIT Press, 1989, p. 439-484.

Moore, J.W. and Hines, M. Simulations with NEURON 3.1, on-line documentation in html format, available at http://neuron.duke.edu, 1996.

Murer, S., Omohundro, S.M., Stoutamire, D., and Szyerski, C. Iteration abstraction in Sather. *ACM Transactions on Programming Languages and Systems* 18:1-15, 1996.

O'Boyle, M.P., Carnevale, N.T., Claiborne, B.J., and Brown, T.H. A new graphical approach for visualilzing the relationship between anatomical and electrotonic structure. In: *Computational Neuroscience: Trends in Research 1995*, edited by J.M. Bower. San Diego: Academic Press, 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. Cable theory for dendritic neurons. In: *Methods in Neuronal Modeling*, edited by C. Koch and I. Segev. Cambridge, MA: MIT Press, 1989, p. 8-62.

Softky, W. Sub-millisecond coincidence detection in active dendritic trees. *Neurosci.* 58:13-41, 1994.

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.

Traynelis, S.F., Silver, R.A., and Cull-Candy, S.G. Estimated conductance of glutamate receptor channels activated during epscs at the cerebellar mossy fiber-granule cell synapse. *Neuron* 11:279-289, 1993.

Tsai, K.Y., Carnevale, N.T., and Brown, T.H. Hebbian learning is jointly controlled by electrotonic and input structure. *Network* 5:1-19, 1994a.

Tsai, K.Y., Carnevale, N.T., Claiborne, B.J., and Brown, T.H. Efficient mapping from neuroanatomical to electrotonic space. *Network* 5:21-46, 1994b.

Zador, A.M., Agmon-Snir, H., and Segev, I. The morphoelectrotonic transform: a graphical approach to dendritic function. *J. Neurosci.* 15:1669-1682, 1995.

APPENDIX 1. LISTING OF hoc CODE FOR THE MODEL IN SECTION 4.2

```
// model topology
// declare sections
create soma, axon, dendrite[3]
// attach sections to each other
connect axon(0), soma(0)                        // 0 end of axon to 0 end of soma
for i=0,2 {                                     // 0 end of dendrite to 1 end of soma
      connect dendrite[i](0), soma(1)
}

// specify anatomical and biophysical properties
soma {
      nseg = 1                  // compartmentalization parameter
      L = 50                    // [µm] length
      diam = 50                 // [µm] diameter
      insert hh                 // standard Hodgkin-Huxley currents
      gnabar_hh = 0.5*0.120     // [S/cm^2]
                                // max gNa in soma will be half of axonal value
}
axon {
      nseg = 20
      L = 1000
      diam = 5
      insert hh
}
for i=0,2 dendrite[i] {
      nseg = 5
      L = 200
      diam(0:1) = 10:3          // dendritic diameter tapers along its length
      insert pas                // standard passive current
      e_pas = -65               // [mv] equilibrium potential for passive current
      g_pas = 0.001       // [S/cm2] conductance for passive current
}

// stimulating current
objref stim
soma stim = new IClamp(0.5)     // put it in middle of soma
stim.del = 1                    // [ms] delay
stim.dur = 0.1                  // [ms] duration
stim.amp = 60             // [nA] amplitude

// simulation time course
// set initial conditions
dt = 0.05                 // [ms] integration time step
tstop = 5                 // [ms]
finitialize(-65)          // init membrane potential, state variables, and time

// argument specifies how long to integrate
// does NOT reset t to 0
proc integrate() {
      print t, soma.v(0.5)      // show starting time
                                //    and initial somatic membrane potential
      while (t < tstop) {
            fadvance()                // advance solution by dt
            // function calls to save or plot results would go here
            print t, soma.v(0.5)      // show present time
                                      //    and somatic membrane potential
            // statements that change model parameters would go here
      }
}
```

