Chapter 5

Representing neurons with a digital computer

Information processing in the nervous system involves the spread and interaction of electrical and chemical signals within and between neurons and glia. From the perspective of the experimentalist working at the level of cells and networks, these signals are continuous variables. They are described by the diffusion equation and the closely-related cable equation [Crank, 1979 #377][Rall, 1977, #108], in which potential (voltage, concentration) and flux (current, movement of solute) are smooth functions of time and space. But everything in a digital computer is inherently discontinuous: memory addresses, data, and instructions are all specified in terms of finite sequences of 0s and 1s, and there are finite limits on the precision with which numbers can be represented. Thus there is no direct parallel between the continuous world of biology and what exists in digital computers, so special effort is required to implement digital computer models of biological neural systems. The aim of this chapter is to show how the NEURON simulation environment makes it easier to bridge this gap.

Discretization

To simulate the operation of biological neurons, NEURON uses the tactic of discretizing time and space, which means approximating these partial differential equations by a set of algebraic difference equations that can be solved numerically (numerical integration; see **Chapter 4: Essentials of numerical methods for neural modeling**). Indeed, spatial discretization, in one form or another, lies at the core of all simulators used to model biological neurons.

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

Choosing an appropriate discretization is a recurring practical problem in neural modeling. The accuracy required of a discrete approximation to a continuous system, and the effort needed to compute it, depend on the anatomical and biophysical complexity of the original system and the question that is being asked. Thus finding the resting membrane potential of an isopotential model with passive membrane may require only a few large time steps at one point in space, but determining the time course of V_m throughout a highly branched model with active membrane as it fires a burst of spikes demands much finer spatiotemporal resolution; furthermore, selecting Δx and Δt for complex models can be especially difficult.

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

Over the years, a certain amount of folklore and numerous unreliable rules of thumb have emerged concerning the topic of "compartment size." Among the topics we cover in this chapter are a practical method for quickly testing spatial accuracy, and a rational basis for specifying the spatial grid that makes use of the AC length constant at high frequencies [Hines, 2001 #568].

No less important is the practical question of how to manage all the parameters that exist throughout a model. Returning briefly to the metaphor of "compartments," let us consider membrane capacitance, a parameter that has a different value in each compartment. Rather than specify the *capacitance* of each compartment individually, it is better to deal in terms of a single *specific membrane capacitance* that is constant over the entire cell, and have the program compute the values of the individual capacitances from the areas of the compartments. Other parameters such as diameter or channel density may vary widely over short distances, so the granularity of their representation may have little to do with numerically adequate discretization.

How NEURON separates anatomy and biophysics from purely numerical issues

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

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

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

Sections and section variables

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

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

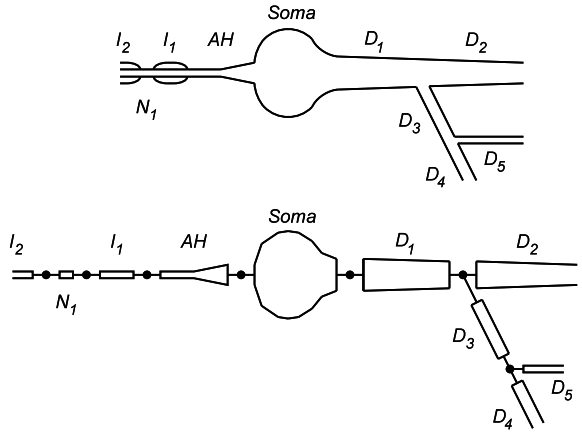

Figure 5.1. Top: Cartoon of a neuron indicating biologically significant structures. Bottom: How these structures are represented by sections in a NEURON model. Reproduced from [Hines, 1997 #208].

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

Table 5.1. Section variables

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

Range and range variables

Many variables in real neurons are continuous functions of position throughout the cell. In NEURON these are called *range variables* (see Table 5.2 for examples). While each section is ultimately discretized into compartments, range variables are specified in terms of a continuous parameter: normalized distance along the centroid of each section. This normalized distance, which is called *range* or *arc length*, varies linearly from 0 at one end of the section to 1 at the other. Figure 5.2 depicts the correspondence between the physical distance of a point along the length of a section and its location in units of normalized distance.

Table 5.2. Some examples of range variables

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

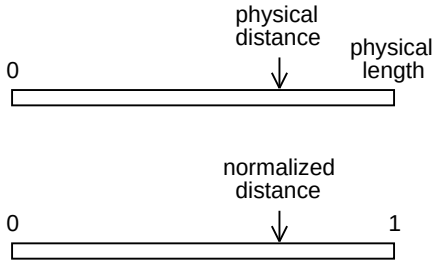

Figure 5.2. Top: The arrow indicates the location of a point at a particular physical distance from one end of a section. Bottom: In NEURON, this location is expressed in terms of normalized distance ("range") along the length of the section.

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

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

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

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

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

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

Segments

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

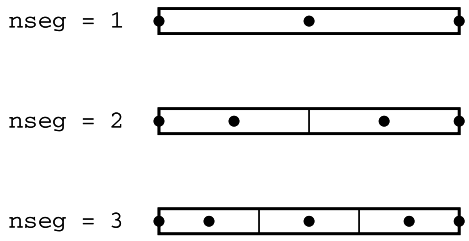

Figure 5.3. Each section has a discretization parameter `nseg` that governs the number of internal nodes (black dots inside the section) at which membrane potential is computed. The thin lines mark conceptual boundaries between adjacent segments.

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

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

```
axon.nseg = 3
```

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

Implications and applications of this strategy

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

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

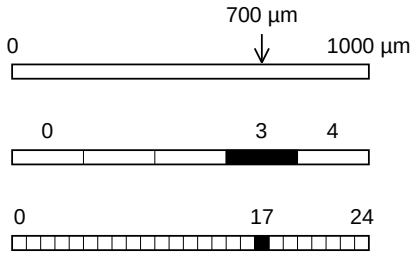

Figure 5.4. Boxed in by compartments. Top: Conceptual model of an unbranched axon 1000 μm long. We are interested in membrane potential at a point 700 μm from its left end. Middle and bottom: The index of the compartment that corresponds to the location of interest depends on how many compartments there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

Spatial accuracy

As we mentioned in **Chapter 4**, the spatial discretization method employed by NEURON produces solutions that are second order correct in space, i.e. spatial error

within a section is proportional to the square of its segment length. It is crucial to realize that the location of the second order correct voltage is not at the edge of a segment but rather at its *center*, i.e. at its node (Fig. 5.3; also see **Spatial discretization** in **Chapter 4**). This has several important consequences.

- To allow branching and injection of current at the precise ends of a section while maintaining second order correctness, extra voltage nodes that represent compartments with 0 area are defined at the section ends. It is possible to achieve second order accuracy with sections whose end nodes have nonzero area compartments, but the areas of these terminal compartments would have to be exactly half that of the internal compartments, and extra complexity would be imposed on administration of channel density at branch points.
- To preserve second order accuracy, localized current sources (e.g. synapses, current clamps, voltage clamps--see **Point processes** below) must be placed at nodes. For the same reason, all sections should be connected at nodes.
- If `nseg` is even, `dend.v(0.5)` and `dend.v` will return a value that actually comes from "the nearest internal node" which is not at the middle of `dend` but instead depends on roundoff error. Using odd values for `nseg` avoids such capricious outcomes by ensuring that there will be a node at the midpoint of each section.
- Second order spatial accuracy means that the results of a NEURON simulation are a piecewise linear approximation to the continuous system. Therefore second order accurate estimates of continuous variables at intermediate locations in space can be found by linear interpolation between nodes.

A practical test of spatial accuracy

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

```
forall nseg *= 3
```

which increases spatial resolution by a factor of 3 throughout the model and reduces spatial error terms by a factor of 9. Now run a "test" simulation and see if a significant qualitative or quantitative change has occurred. The absence of a significant change is evidence that the control simulation was sufficiently accurate in space.

Why triple `nseg` instead of just doubling it? Because NEURON uses a piecewise linear approximation to emulate the continuous variation of membrane current and voltage in space. The breakpoints in this piecewise linear approximation are located at the internal nodes of each section. Multiplying `nseg` by an even number will shift these breakpoints to new locations, making it hard to compare the results of the control and test simulations. For instance, with `nseg` = 1, voltage is computed at arc length = 0.5, but with `nseg` = 2 it is computed at arc length = 0.25 and 0.75 (see Fig. 5.3). If simulations with `nseg` = 1 and `nseg` = 2 did produce different results, it could be difficult to know whether this reflects improved spatial accuracy or is just due to the fact that the two simulations computed solutions at different points in space. Tripling `nseg` adds new

breakpoints (at arc length = 1/6 and 5/6 in Fig. 5.3) without changing the locations of any that were already there (at 0.5 in this case). Any odd multiple could be used, but 3 is a practical value since it reduces spatial error by almost an order of magnitude, which is probably enough to detect inadequate spatial accuracy.

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

How to specify model properties

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

Which section do we mean?

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

NEURON offers three ways to specify the currently accessed section, each being compact in some contexts and cumbersome in others: dot notation, section stack, and default section. We consider them in order of precedence, starting with the method that has highest priority.

1. **Dot notation**

   Syntax *`sectionname.variablename`*

   Examples

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


Comments

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

2. Section stack

Syntax *`sectionname { stmt }`*

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

Comments

- This is the most useful method for programming, since the user has explicit control over the scope of the section and can set several range variables.
- Nesting is allowed to any level, i.e.

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

- Avoid the error

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

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

- Control flow should reach the end of *`stmt`* in order to automatically pop the section stack. Therefore *`stmt`* should not include the `continue`, `break`, or `return` statement.
- A section cannot be used as a variable for assignment or passing as an argument to a function or procedure. However, the same effect can be obtained with the `SectionRef` class, which allows sections to be referenced by normal object variables. The use of `push_section()` for this purpose should be avoided except as a last resort.
- Looping over sets of sections is most often done with the `forall` and `forsec` commands.

3. Default section

Syntax `access` *`sectionname`*

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

```
access soma
```


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

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

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

Comments

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

  ```
  sectionname { stmt }
  ```

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

How to set up model topology

In the NEURON simulation environment, the branched topology of a model cell is constructed by creating sections and attaching them to each other in the form of a tree. Sections are created with `hoc` statements of the form

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

```
create sectionname
```

They can be attached to each other with the syntax

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

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

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

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

Loops of sections

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

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

A section may have only one parent

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

The root section

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

Attach sections at 0 or 1 for accuracy

Section attachments must be located at nodes to preserve second order spatial accuracy. It is generally best for `x` to be either 0 or 1, rather than an intermediate value. Attempting to connect a section to a non-node location will result in the section actually being connected to the nearest internal node of the parent, which depends on the value of `nseg` and may be quite far from the intended position. Even if a section is connected to an internal node, if `nseg` is then changed, e.g. to test for spatial accuracy, the attachment could be repositioned to a different site (another reason to increase `nseg` by an odd factor). This would affect the electrotonic architecture of the model, causing spurious changes in simulation results. Therefore the best policy is to connect child sections only to the 0 or the 1 end of the parent, and not to intermediate locations. Because of their small size, dendritic spines are a possible exception to this rule.

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

Checking the tree structure with `topology()`

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

For example the statements

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


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

```
topology()
```

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

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

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

Viewing topology with a Shape plot

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

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

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

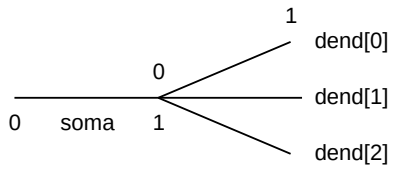

Figure 5.5. A Shape plot display of the topology of a model in which the 0 ends of three child sections are attached to the 1 end of the root section.

How to specify geometry

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

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


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

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

Stylized specification

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

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

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

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

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

3-D specification

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

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

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

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

To prevent confusion, when using the 3-D method one should generally attach only the 0 end of a child section to a parent. This will ensure that `diam(x)` (segment diameter) as `x` ranges from 0 to 1 has the same sense as `diam3d(i)` (the actual morphometric diameters) as `i` ranges from 0 to `n3d()`-1 (`n3d()` is the number of (x, y, z, diam) points used to specify the geometry of the section). It can also prevent unexpected distortions of the model appearance in a Shape plot (see **The case of the disappearing section** in **Chapter 6**).

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

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

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

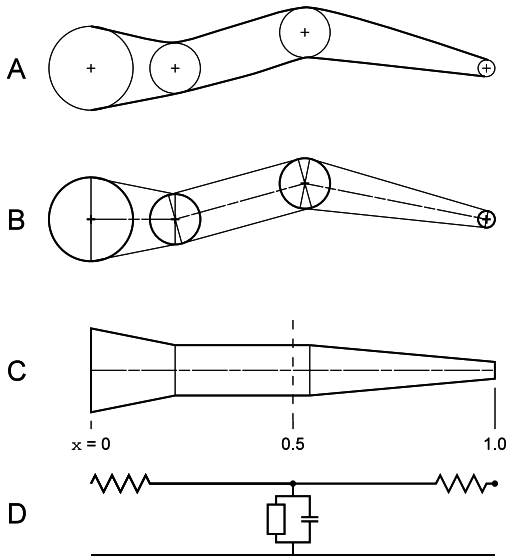

Figure 5.6. A: cartoon of an unbranched neurite (thick lines). Quantitative morphometry has generated successive diameter measurements (circles) centered at x, y, z coordinates (crosses). B: Each adjacent pair of diameter measurements is treated as parallel faces of a truncated cone or frustum. The central axis of the chain of solids is indicated by a thin centerline. C: After straightening the centerline so the faces of adjacent frusta are flush with each other. The scale beneath the figure shows the distance along the midline of the section in terms of arc length, symbolized here by the variable x. The vertical dashed line at x = 0.5 divides the section into two halves of equal length. D: Equivalent circuit of the section when nseg = 1. The open rectangle includes all mechanisms for ionic (non-capacitive) transmembrane currents. Reproduced from [Hines, 1997 #208].

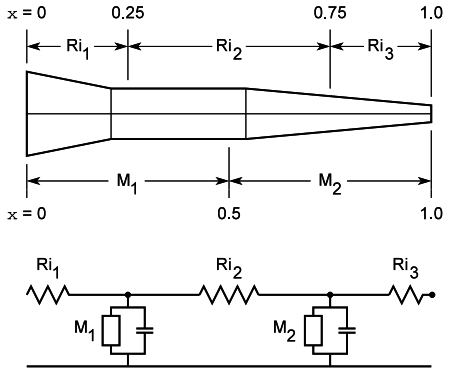

Figure 5.7. Representation of the neurite of Fig. 5.6 when nseg = 2. The equivalent circuit now has two nodes. See text for details. Reproduced from [Hines, 1997 #208].

Avoiding artifacts

Beware of zero diameter

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

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

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

will produce an error message like this

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

While NEURON prevents the diameter from becoming 0, 10^{-6} µm is so narrow that axial resistance in the affected region is, for modeling intents and purposes, infinite. Models constructed with the stylized specification can be checked for narrow diameters by executing

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

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

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

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

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

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

Stylized specification may be reinterpreted as 3-D specification

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

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

For example, this code

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

defines a section with three segments, with `diam` = 10 μm in the segments centered at 0.16666667 and 0.5, and 20 μm in the segment centered at 0.83333333. Since the stylized method was used to create this section, there will be no 3-D points. We can verify this by typing `n3d()` and noting that the returned value is 0. We can also check `diam` and the computed values of `area` and `ri` with the statement

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

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

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

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

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

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

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

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

However, the values of `diam`, `area`, and `ri` have been altered in the segments adjacent to the diameter change (Fig. 5.8). This effect is smaller when `nseg` is larger. It is caused by the fact that the 3-D points define a series of truncated cones rather than right circular cylinders. The reported `diam(x)` is the average diameter over the corresponding length of the 3-D model, and `area(x)` is the integral of the 3-D surface; this is not necessarily equal to the stylized area `PI*diam(x)*L/nseg`, which ignores end area associated with abrupt diameter changes. This latter difference may be small, as in this case where `area(x)` for the second and third segments is 1185 and 1974 μm^2 respectively, compared to 1178 and 1963 μm^2 for the stylized area (all values rounded to the nearest μm^2), but actual results depend on model geometry and whether these have a significant effect on simulation results can only be judged on a case by case basis. What

is clear for all cases, however, is that abrupt diameter changes should only take place at the boundaries of sections if we wish to view shape and also use the smallest possible number of segments.

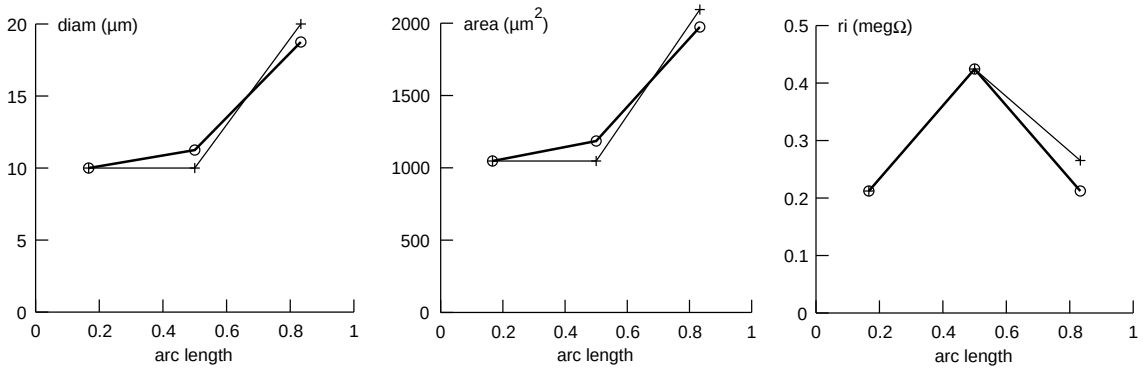

Figure 5.8. `diam`, `area`, and `ri` at the internal nodes of a 100 µm long section with `nseg` = 3 and `Ra` = 100 Ω cm. Thin lines with + show the values immediately after `geometry` was specified, when no 3-D points existed. Thick lines with circles show the values after `define_shape()` was executed,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

How to specify biophysical properties

As we mentioned in **How to specify geometry**, the only biophysical attributes of a new section are cytoplasmic resistivity `Ra` and specific membrane capacitance `cm`, whose default values are 35.4 Ω cm and 1 µf/cm^2, respectively. A new section has no membrane conductances, pumps, or buffers. It is assumed to lie in an extracellular medium with zero resistance or capacitance, and there are no synapses, gap junctions, or voltage or current clamps. Anything other than the bare bones framework of `Ra` and `cm` must be added.

Distributed mechanisms

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

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

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


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

Point processes

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

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

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

An object syntax

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

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

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

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

When a point process is no longer referenced by any object reference, it is removed from the section and destroyed. Consequently, redeclaring `stim` with the statement

`objref stim` would destroy this `IClamp`, since no other object reference would reference it.

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

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

would actually place `stim1` at 0.1 and `stim2` at 0.7. The error introduced by this "shift" can be avoided by explicitly placing point processes at internal nodes, and restricting changes of `nseg` to odd multiples. However, this may not be possible in models that are based closely on real anatomy, because actual synaptic locations are unlikely to be situated precisely at segment centers. To completely avoid `nseg`-dependent shifts of point process locations, one can choose sections with lengths such that the point processes are located at 0 or 1 ends.

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

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

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

User-defined mechanisms

User-defined distributed mechanisms and point processes can be added to NEURON with the model description language NMODL. This lets the user focus on specifying the equations for a channel or ionic process without regard to its interactions with other mechanisms. The NMODL translator constructs C code which properly and efficiently computes the total current of each ionic species used, as well as the effect of that current on ionic concentration, reversal potential, and membrane potential. This code is compiled and linked into NEURON. NMODL is discussed extensively in **Chapter 9** and **10**, but it is useful to review some of its advantages here.

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

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

2. Consistency of units is ensured.

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

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

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

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

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

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

Working with range variables

Iterating over nodes

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

```
for (var) stmt
```

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

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

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

Linear taper

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

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

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

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

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

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

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

How changing `nseg` affects range variables

If `nseg` is increased after range variables have been specified, all old segments are relocated to their nearest new locations (no instance variables are modified and no pointers to data in those segments become invalid), and new segments are allocated and given mechanisms and values that are identical to the old segment in which the center of the new segment is located. If range variables are not constant, then the `hoc` expressions

used to set them should be re-executed. To see why, let us return to our axon with a linearly tapering gnabar_hh, specified by executing

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

after which we check by executing

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

which returns

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

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

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

and see

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

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

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

and when we now test the gradient we find

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

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

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

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

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

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

which clearly differs from the result of executing

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

(see Table 5.4).

Choosing a spatial grid

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

A consideration of intent and judgment

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

Consider a model of a small spherical cell with passive membrane that is subjected to a depolarizing current pulse (Fig. 5.9). The spatial grid for this isopotential cell only needs a single node, i.e. this is a situation in which the sole consideration to be weighed is the discretization of time.

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

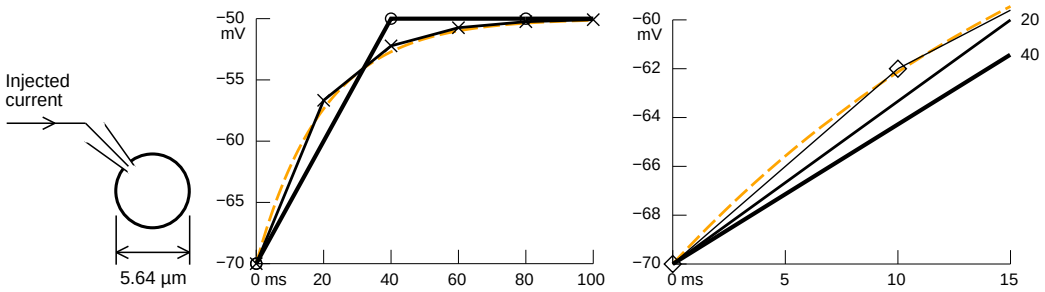

Figure 5.9. A spherical cell (left) with a surface area of 100 µm^2 (diameter = 5.64 µm) is subjected to a 1 pA depolarizing current that starts at t = 0 ms. Resting potential is -70 mV, specific membrane capacitance and resistance are C_m = 1 µf / cm^2 and R_m = 20,000 Ω cm^2, respectively (τ_m = 20 ms). The dashed orange trace in the middle and right graphs is the analytic solution for V_m. The solid black traces are the numeric solutions computed with time steps Δt = 40 ms (thick trace, open circles), 20 ms (medium trace, ×), and 10 ms (thin trace, diamond, right figure only). Modified from [Hines, 2001 #568].

Spatial discretization becomes important in models that are extensive enough for the propagation of electrical or chemical signals to involve significant delay. We illustrate this with a model of fast excitatory synaptic input onto a dendritic branch. The synapse in this model is attached to the middle of an unbranched cylinder (Fig. 5.10). To prevent possible confounding effects of active current kinetics and complex geometry, we assume that the cylinder has passive membrane and is five DC length constants long. The biophysical properties are within the range reported for mammalian central neurons [Spruston, 1992 #78]. The time course of the synaptic conductance follows an alpha function with time constant τ_s and reversal potential E_s chosen to emulate an AMPA synapse [Kleppe, 1999 #375], and g_{max} selected to produce a peak depolarization of approximately 10 mV. We will compare the analytic solution for V_m in this model with the numeric solution computed for a very coarse spatial grid (Δx = 1 λ). The numeric

solution uses a time step Δt = 1 μs, which is more than two orders of magnitude smaller than necessary to follow the EPSP waveform, so that differences from the analytic solution are almost entirely attributable to the spatial grid.

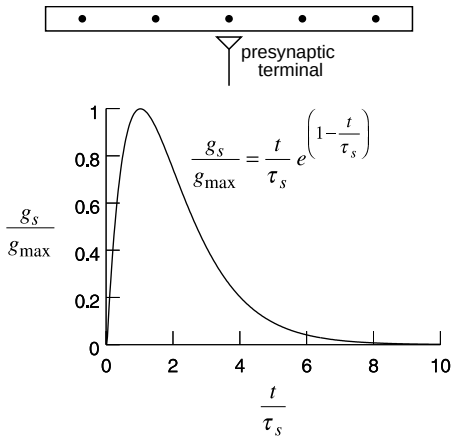

Figure 5.10. Model of synaptic input onto a dendrite The dendrite is represented by an unbranched cylinder (top) with diameter = 1 μm, length = 2500 μm, R_a = 180 Ω cm, C_m = 1 μf / cm^2, and R_m = 16,000 Ω cm^2 with a resting potential of -70 mV. The DC length constant λ_{DC} of the cylinder is 500 μm, so its sealed end terminations have little effect on the EPSP produced by a synapse located at its midpoint. The dots are the locations at which the numeric solution would be computed using a grid with intervals of 1 λ_{DC}, i.e. 250, 750, 1250, 1750, and 2250 μm. The synaptic conductance g_s is governed by an alpha function (bottom) with τ_s = 1ms, g_{max} = 10^{-9} siemens, and reversal potential E_s = 0 mV. Modified from [Hines, 2001 #568].

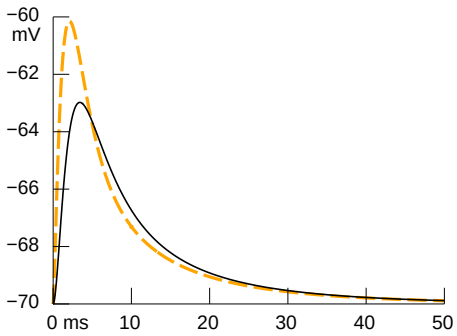

Figure 5.11. Time course of V_m at the synaptic location. The dashed orange line is the analytic solution, and the solid black line is the numeric solution computed with Δt = 1 μs. Modified from [Hines, 2001 #568].

Compared to the analytic solution for V_m at the site of synaptic input (dashed orange trace in Fig. 5.11), the numeric solution (solid black trace) rises and falls more slowly, and has a peak depolarization that is substantially delayed and smaller. These differences reflect the fact that solutions based on the coarse grid lack sufficient amplitude in the

high frequency terms that are needed to reproduce rapidly changing signals. Such errors could lead to serious misinterpretations if our intent were to examine how synaptic input might affect depolarization-activated currents with fast kinetics like I_A, spike sodium current, and transient I_{Ca}.

The graphs in Fig. 5.12 present the spatial profile of V_m along the dendrite at two times selected from the rising and falling phases of the EPSP. These curves, which are representative of the early and late response to synaptic input, show that the error of the numeric solution is most pronounced in the part of the cell where V_m changes most rapidly, i.e. in the near vicinity of the synapse. However, at greater distances the analytic solution itself changes much more slowly because of low pass filtering produced by cytoplasmic resistance and membrane capacitance. At these distances the error of the numeric solution is surprisingly small, even though it was computed with a very crude spatial grid. Furthermore, error decreases progressively as time advances and high frequency terms become less important. This suggests that the coarse grid may be quite sufficient if our real interests are in slow processes that take place at some distance from the site of synaptic input.

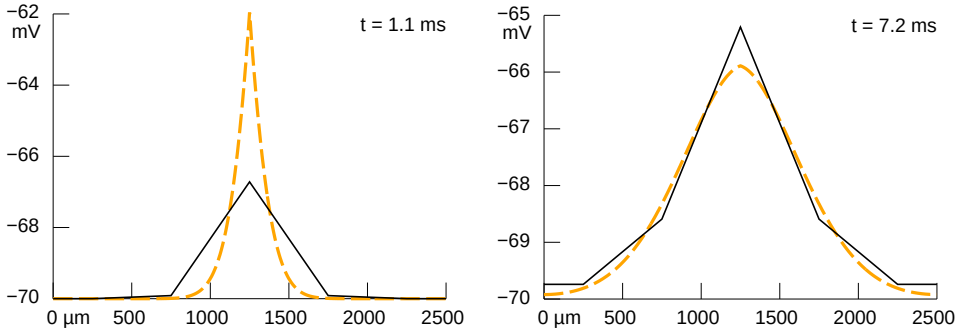

Figure 5.12. V_m vs. distance along the dendrite computed during the rising (left) and falling (right) phases of the EPSP. The analytic and numeric solutions are shown with dashed orange and solid black lines, respectively. The error of the numeric solution is greatest in the region where V_m changes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

Discretization guidelines

Various strategies have appeared in the literature as aids to the use of judgment in choosing a spatial grid. One common practice is to keep the distance between adjacent grid points smaller than some fraction (e.g. 5 - 10%) of the DC length constant λ_{DC} of an infinite cylinder with identical anatomical and biophysical properties [Mainen, 1998 #304][Segev, 1998 #379]. This plausible approach has two chief limitations. First, large changes in membrane resistance and λ_{DC} can be produced by activation of voltage-dependent channels (e.g. I_h [Magee, 1998 #315][Stuart, 1998 #301]), Ca^{2+}-gated channels [Wessel, 1999 #509], or synaptic inputs [Bernander, 1991 #162][Destexhe,

1999 #352][Häusser, 1997 #381][Pare, 1998 #349]. The second but more fundamental problem is that the spatial decay of transient signals is unrelated to λ_{DC}. Cytoplasmic resistivity R_a and specific membrane capacitance C_m constitute a spatially distributed low pass filter, so transient signals suffer greater distortion and attenuation with distance than do slowly changing signals or DC. In other words, by virtue of their high frequency components in time, transient signals also have high frequency components in space. Just as high temporal frequencies demand a short time step, high spatial frequencies demand a fine grid.

The d_lambda rule

As a more rational approach, we have suggested what we call the "d_lambda rule" [Hines, 2001 #568], which predicates the spatial grid on the AC length constant λ_f computed at a frequency *f* that is high enough for transmembrane current to be primarily capacitive, yet still within the range of frequencies relevant to neuronal function. Ionic and capacitive transmembrane currents are equal at the frequency $f_m = 1 / 2 \pi \tau_m$, so specific membrane resistance R_m has little effect on the propagation of signals $\geq 5 f_m$. For instance, a membrane time constant of 30 ms corresponds to $f_m \sim 5$ Hz, which implies that R_m would be irrelevant to signal spread at frequencies ≥ 25 Hz. Most cells of current interest have $\tau_m \geq 8$ ms ($f_m \sim 20$ Hz), so we suggest that the distance between adjacent nodes should be no larger than a user-specified fraction "d_lambda" of λ_{100}, the length constant at 100 Hz. This frequency is high enough for signal propagation to be insensitive to shunting by ionic conductances, but it is not unreasonably high because the rise time τ_r of fast EPSPs and spikes is ~ 1 ms, which corresponds to a bandpass of $1/\tau_r \sqrt{2\pi}$ ~400 Hz.

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

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

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

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

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

In NEURON the d_lambda rule is implemented in the CellBuilder, which allows the maximum anatomical distance between grid points to be specified as a fraction of λ_{100}

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

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

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

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

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

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

which is included in the file

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

or

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

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

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

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

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

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

To see how the d_lambda rule works in practice, consider the model in Fig. 5.13, which represents a granule cell from the dentate gyrus of the rat hippocampus. This

model is based on quantitative morphometric data provided by Dennis Turner (available from http://www.cns.soton.ac.uk/~jchad/cellArchive/cellArchive.html or http://www.compneuro.org/CDROM/nmorph/cellArchive.html), and the biophysical parameters are from Spruston and Johnston [Spruston, 1992 #78]: R_m = 40 k Ω cm^2, C_m = 1 μf / cm^2, and R_a = 200 Ω cm. An excitatory synapse attached to the soma is an excitatory synapse whose conductance is governed by an alpha function with τ_s = 1 ms, g_{max} = 2 · 10^{-9} S, and reversal potential E_s = 0 mV.

The right side of Fig. 5.13 shows the simulated time course of V_m at the soma for three different methods of specifying the spatial grid: one or three nodes in each branch, and `d_lambda` = 0.3. On the scale of this figure, solutions with `d_lambda` ≤ 0.3 are indistinguishable from each other, so `d_lambda` = 0.3 serves as the standard for accuracy. Plots generated with constant `nseg` per branch converged toward this trace as `nseg` increased. Even the crudest spatial grid (`nseg` = 1) would suffice if the purpose of the model were to evaluate effects of synaptic input on V_{soma} well after the peak of the EPSP (t > 7 ms). However a finer grid is clearly necessary if the maximum somatic depolarization produced by the EPSP is of concern.

Additional refinements to the grid are needed if we want to know how the EPSP spreads into other parts of the cell, e.g. along the path marked by orange in Fig. 5.14 left. To compute the maximum depolarization produced by a somatic EPSP along this path, a grid that has only 3 nodes per branch is quite sufficient (Fig. 5.14 center). If the timing of this peak is important, e.g. for coincidence detection or activation of voltage-gated currents, a finer grid must be used (Fig. 5.14 right).

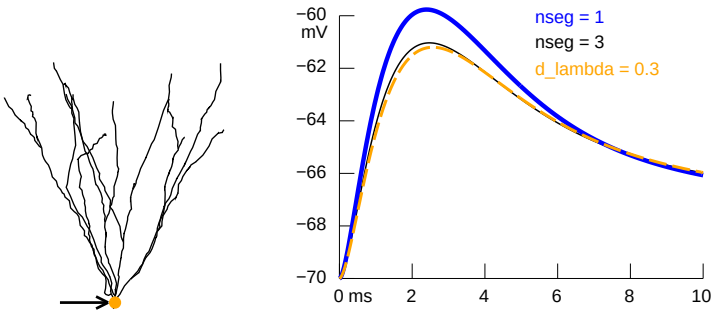

Figure 5.13. Left: Anatomically complex model of a granule cell from the dentate gyrus of rat hippocampus. A fast excitatory synapse is attached to the soma (location indicated by arrow and orange dot). See text for details. Right: Time course of V_{soma} computed using spatial grids with one or three nodes per branch (thick blue and thin black traces for `nseg` = 1 and 3, respectively) or specified with `d_lambda` = 0.3 (dashed orange trace). Modified from [Hines, 2001 #568].

The computational cost of these simulations is approximately proportional to the number of nodes. Least burdensome, but also least accurate, were the simulations generated with one node per branch, which involved a total of 28 nodes in the model.

Increasing the number of nodes per branch to 3 (total nodes in model = 84) improved accuracy considerably, but obvious errors remained (Fig. 5.14 right) that disappeared only after an additional tripling of the number of nodes per branch (total nodes = 252; results not shown). The greatest accuracy with least sacrifice of efficiency was achieved with the grid specified by `d_lambda` = 0.3, which contained only 110 nodes.

As these figures suggest, the relative advantage of the d_lambda rule will be most apparent when signal propagation throughout the entire model must be simulated to a similar level of accuracy. If the focus is on a limited region, then a grid with fewer nodes and a simpler representation of electrically remote regions may be acceptable. Special features of the model may also allow a simpler grid to be used. In principal neurons of mammalian cortex, for example, proximal dendritic branches tend to have larger diameters [Hillman, 1979 #128][Rall, 1959 #383] and shorter lengths [Cannon, 1999 #382] than do distal branches. Therefore models based on quantitative morphometry of such neurons will have fewer nodes in proximal dendrites than in more distal dendrites if the grid is specified by the d_lambda or d_X rule. Indeed, many proximal branches may have only one or three nodes, regardless of which rule is applied, and in such a case the differences between gridding strategies will be manifest only in the thinner and longer distal branches. Such differences will have little effect on accuracy if signals in the vicinity of the soma are the only concern.

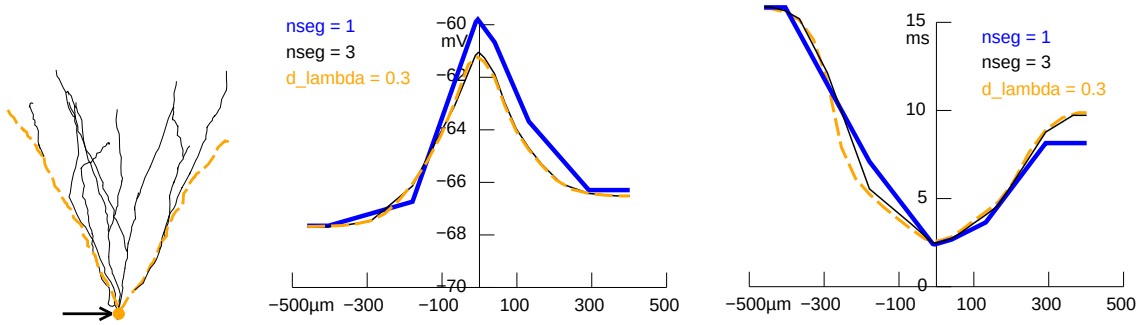

Figure 5.14. The EPSP evoked by activation of a synapse at the soma (arrow in left panel) spread into the dendrites, producing a transient depolarization which grew smaller and occurred later as distance from the soma increased. The center and right panels show the magnitude and timing of this depolarization along the path marked by the dashed orange line. Peak amplitude was quite accurate with `nseg` = 3 (thin black trace, center panel), but noticeable error persisted in the time of peak depolarization for distances between -300 and -150 µm (right panel, especially between). The dashed orange trace in the center and right panels was obtained with `d_lambda` = 0.3. Time step was 25 µs.

Chapter 5 Index

