Chapter 9

How to expand NEURON's library of mechanisms

Neuronal function involves the interaction of electrical and chemical signals that are distributed in time and space. The mechanisms that generate these signals and regulate their interactions are marked by a wide diversity of properties, differing across neuronal cell class, developmental stage, and species (e.g. chapter 7 in [Johnston, 1995 #218]; also see [McCormick, 1998 #288]). To be useful in research, a simulation environment must provide a flexible and powerful means for incorporating new biophysical mechanisms in models. It must also help the user remain focused on the model instead of programming.

Such a means is provided to NEURON by NMODL, a high level language that was originally implemented for NEURON by Michael Hines and later extended by him and Upinder Bhalla to generate code suitable for linking with GENESIS [Wilson, 1989 #289]. This chapter shows how to use NMODL to represent biophysical mechanisms by presenting a sequence of increasingly complex examples.

Overview of NMODL

A brief overview of how NMODL is used will clarify its rationale. First one writes a text file (a "`mod` file") that describes a mechanism as a set of nonlinear algebraic equations, differential equations, or kinetic reaction schemes. The description employs a syntax that closely resembles familiar mathematical and chemical notation. This text is passed to a translator that converts each statement into many statements in C, automatically generating code that handles details such as mass balance for each ionic species and producing code suitable for each of NEURON's integration methods. The output of the translator is then compiled for computational efficiency. This achieves tremendous conceptual leverage and savings of effort, not only because the high level mechanism specification is much easier to understand and far more compact than the equivalent C code, but also because it spares the user from having to bother with low level programming issues like how to "interface" the code with other mechanisms and with NEURON itself.

NMODL is a descendant of the MOdel Description Language (MODL [Kohn, 1994 #290]), which was developed at Duke University by the National Biomedical Simulation Resource project for the purpose of building models that would be exercised by the Simulation Control Program (SCoP [Kootsey, 1986 #286]). NMODL has the same basic syntax and style of organizing model source code into named blocks as MODL. Variable declaration blocks, such as `PARAMETER`, `STATE`, and `ASSIGNED`, specify names and attributes of variables that are used in the model. Other blocks are directly involved in setting initial conditions or generating solutions at each time step (the equation definition

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

blocks, e.g. `INITIAL`, `BREAKPOINT`, `DERIVATIVE`, `KINETIC`, `FUNCTION`, `PROCEDURE`). Furthermore, C code can be inserted inside the model source code to accomplish implementation-specific goals.

NMODL recognizes all the keywords of MODL, but we will address only those that are relevant to NEURON simulations. We will also examine the changes and extensions that were necessary to endow NMODL with NEURON-specific features. To give these ideas real meaning, we will consider them in the context of models of the following kinds of mechanisms:

- a passive "leak" current and a localized transmembrane shunt (distributed mechanisms vs. point processes)
- an electrode stimulus (discontinuous parameter changes with variable time step methods)
- voltage-gated channels (differential equations vs. kinetic schemes)
- ion accumulation in a restricted space (extracellular K^+)
- buffering, diffusion, and active transport (Ca^{2+} pump)

Features of NMODL that are used in models of synaptic transmission and networks are examined in **Chapter 10**.

Example 9.1: a passive "leak" current

A passive "leak" current is one of the simplest biophysical mechanisms. Because it is distributed over the surface of a cell, it is described in terms of conductance per unit area and current per unit area, and therefore belongs to the class of "density" or "distributed mechanisms" (see **Distributed mechanisms** in **Chapter 5**).

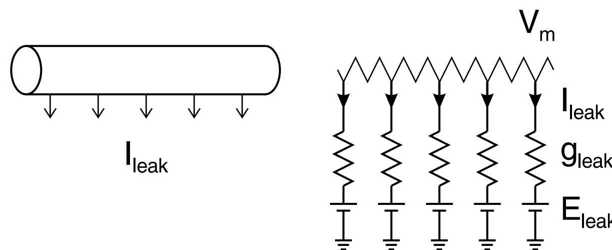


Figure 9.1

Figure 9.1 illustrates a branch of a neuron with a distributed leak current (left) and the equivalent circuit of a model of this passive current mechanism (right): a distributed, voltage-independent conductance g_{leak} in series with a voltage source E_{leak} that represents the equilibrium potential for the ionic current. The leak current density is given by $i_{leak} = g_{leak} (V_m - E_{leak})$, where V_m is membrane potential. Since this is a model of a physical system that is distributed in space, the variables i_{leak} and V_m and the parameters g_{leak} and E_{leak} are all functions of position.

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

Listing 9.1 presents an implementation of this mechanism with NMODL. Single line comments start with a : (colon) and terminate at the end of the line. NMODL also allows multiple line comments, which are demarcated by the keywords `COMMENT` and `ENDCOMMENT`.

```
COMMENT
This is a
multiple line
comment
ENDCOMMENT
```

A similar syntax can be used to embed C code in a `mod` file, e.g.

```
VERBATIM
   /* C statements */
ENDVERBATIM
```

The statements between `VERBATIM` and `ENDVERBATIM` will appear without change in the output file written by the NMODL translator. Although this should be done only with great care, `VERBATIM` can be a convenient and effective way to add new features or even to employ NEURON as a "poor man's C compiler."

```
: A passive leak current

NEURON {
  SUFFIX leak
  NONSPECIFIC_CURRENT i
  RANGE i, e, g
}

PARAMETER {
  g = 0.001  (siemens/cm2)  < 0, 1e9 >
  e = -65    (millivolt)
}

ASSIGNED {
  i  (milliamp/cm2)
  v  (millivolt)
}

BREAKPOINT { i = g*(v - e) }
```

Listing 9.1. `leak.mod`

Named blocks have the general form `KEYWORD { statements }`, where `KEYWORD` is all upper case. User-defined variable names in NMODL can be up to 20 characters long. Each variable must be defined before it is used. The variable names chosen for this example were `i`, `g`, and `e` for the leak current, its specific conductance, and its equilibrium potential, respectively. Some variables are not "owned" by any mechanism but are available to all mechanisms; these include `v`, `celsius`, `t`, `diam`, and `area`.

Another variable that is available to all mechanisms is `dt`. However, using `dt` in NMODL is neither necessary nor good practice. Before variable time step methods were added to NEURON, analytic expressions involving `dt` were often used for efficient modeling of voltage sensitive channel states. This idiom is now built-in and employed automatically when such models are described in their underlying derivative form.

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

The NEURON block

The principal extension that differentiates NMODL from its MODL origins is that there are separate instances of mechanism data, with different values of states and parameters, in each segment (compartment) of a model cell. The NEURON block was introduced to make this possible by defining what the model of the mechanism looks like from the "outside" when many instances of it are sprinkled at different locations on the cell. The specifications entered in this block are independent of any particular simulator, but the detailed "interface code" requirements of a particular simulator determine whether the output C file is suitable for NEURON (NMODL) or GENESIS (GMODL). For this paper, we assume the translator is NMODL and that it produces code accepted by NEURON.

The actual name of the current NMODL translator is `nocmodl` (`nocmodl.exe` under MSWindows). This translator is consistent with the object oriented extensions that were introduced with version 3 of NEURON. However, the older translator, which predated these extensions, was called `nmodl`, and we will use the generic name NMODL to refer to NEURON compatible translators.

The `SUFFIX` keyword has two consequences. First, it identifies this to be a distributed mechanism, which can be incorporated into a NEURON cable section by an `insert` statement (see **Usage** below). Second, it tells the NEURON interpreter that the names for variables and parameters that belong to this mechanism will include the suffix `_leak`, so there will be no conflict with similar names in other mechanisms.

The stipulation that `i` is a `NONSPECIFIC_CURRENT` also has two consequences. First, the value of `i` will be reckoned in charge balance equations. Second, this current will make no direct contribution to mass balance equations (it will have no direct effect on ionic concentrations). In later examples, we will see how to implement mechanisms with specific ionic currents that can change concentrations.

The `RANGE` statement asserts that `i`, `e`, and `g` are range variables, and can be accessed by the `hoc` interpreter using range variable syntax (see **Range and range variables** in **Chapter 5**). That is, each of these variables is a function of position, and can have a different value in each of the segments that make up a section. Each variable mentioned in a `RANGE` statement should also be declared in a `PARAMETER` or `ASSIGNED` block. The alternative to `RANGE` is `GLOBAL`, which is discussed below in **The PARAMETER block**.

Membrane potential `v` is not mentioned in the `NEURON` block because it is one of the variables that are available to all mechanisms, and because it is a `RANGE` variable by default. However, for model completeness in non-NEURON contexts, and to enable units checking, `v` should be declared in the `ASSIGNED` block (see below).

Variable declaration blocks

As noted above, each user-defined variable must be declared before it is used. Even if it is named in the `NEURON` block, it still has to appear in a variable declaration block.

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

Mechanisms frequently involve expressions that mix constants and variables whose units belong to different scales of investigation and which may themselves be defined in terms of other, more "fundamental" units. This can easily produce arithmetic errors that are difficult to isolate and rectify. Therefore NMODL has special provisions for establishing and maintaining consistency of units. To facilitate units checking, each variable declaration includes a specification of its units in parentheses. The names used for these specifications are defined in a file called `nrnunits.lib`, which is based on the UNIX units database (`/usr/share/units.dat` in Linux). `nrnunits.lib` is located in `nrn-x.x/share/lib/` under UNIX/Linux, and `c:\nrnxx\lib\` under MSWindows). A variable whose units are not specified is taken to be dimensionless.

The user may specify whatever units are appropriate except for variables that are defined by NEURON itself. These include `v` (millivolts), `t` (milliseconds), `celsius` (ºC), `diam` (µm), and `area` (µm^2). Currents, concentrations, and equilibrium potentials created by the `USEION` statement also have their own particular units (see **The `NEURON` block** in **Example 9.6: extracellular potassium accumulation** below). In this particular distributed mechanism, `i` and `g` are given units of current per unit area (milliamperes/cm^2) and conductance per unit area (siemens/cm^2), respectively.

The `PARAMETER` block

Variables whose values are normally specified by the user are parameters, and are declared in a `PARAMETER` block. `PARAMETER`s generally remain constant during a simulation, but they can be changed in mid-run if necessary to emulate some external influence on the characteristic properties of a model (see **Models with discontinuities** and **Time-dependent `PARAMETER` changes** near the end of this chapter)

The `PARAMETER` block in this example assigns default values of 0.001 siemens/cm^2 and -65 mV to `g` and `e`, respectively. The pair of numbers in angle brackets specifies the minimum and maximum limits for `g` that can be entered into the field editor of the GUI. In this case, we are trying to keep conductance `g` from assuming a negative value. This protection, however, only holds for field editors and does not prevent a `hoc` statement from giving `g` a negative value.

Because `g` and `e` are `PARAMETER`s, their values are visible at the `hoc` level and can be overridden by `hoc` commands or altered through the GUI. If a `PARAMETER` does not appear in a `NEURON` block's `RANGE` statement, it will have `GLOBAL` scope, which means that changing its value will affect every instance of that mechanism throughout an entire model. However, the `RANGE` statement in the `NEURON` block of this particular mechanism asserts that `g` and `e` are range variables, so they can be given different values in every segment that has this leak current.

The `ASSIGNED` block

The `ASSIGNED` block is used to declare two kinds of variables: those that are given values outside the `mod` file, and those that appear on the left hand side of assignment statements within the `mod` file. The first group includes variables that are potentially available to every mechanism, such as `v`, `celsius`, `t`, and ionic variables (ionic

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

variables are discussed in connection with **The NEURON block** in **Example 9.6: extracellular potassium accumulation** below). The second group specifically omits variables that are unknowns in a set of simultaneous linear or nonlinear algebraic equations, or that are dependent variables in differential equations or kinetic reaction schemes, which are handled differently (see **Example 9.4: a voltage-gated current** below for a discussion of the `STATE` block).

By default, a mechanism-specific `ASSIGNED` variable is a range variable, in that it can have a different value for each instance of the mechanism. However, it will not be visible at the `hoc` level unless it is declared in a `RANGE` or `GLOBAL` statement in the `NEURON` block. This contrasts with `ASSIGNED` variables that are not "owned" by any mechanism (`v`, `celsius`, `t`, `dt`, `diam`, and `area`) which *are* visible at the `hoc` level but are *not* mentioned in the `NEURON` block.

The current `i` is not a state variable because the model of the leak current mechanism does not define it in terms of a differential equation or kinetic reaction scheme; that is to say, `i` has no dynamics of its own. Furthermore it is not an unknown in a set of equations, but is merely calculated by direct assignment. Therefore it is declared in the `ASSIGNED` block.

For similar reasons membrane potential `v` is also declared in the `ASSIGNED` block. Although membrane potential is unquestionably a state variable in a model of a cell, to the leak current mechanism it is a driving force rather than a state variable (or even a `STATE` variable).

Equation definition blocks

One equation suffices to describe this simple leak current model. This equation is defined in the `BREAKPOINT` block. As we shall see later, more complicated models may require invoking NMODL's built-in routines to solve families of simultaneous algebraic equations or perform numeric integration.

The BREAKPOINT block

The `BREAKPOINT` block is the main computation block in NMODL. Its name derives from SCoP, which executes simulations by incrementing an independent variable over a sequence of steps or "breakpoints" at which the dependent variables of the model are computed and displayed [Kohn, 1994 #290]. At exit from the `BREAKPOINT` block, all variables should be consistent with the independent variable. The independent variable in NEURON is always time `t`, and neither `t` nor the time step `dt` should be changed in NMODL.

Usage

The following `hoc` code illustrates how this mechanism might be used. Note the use of `RANGE` syntax to examine the value of `i_leak` near one end of `cable`.

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

```
cable {
  nseg = 5
  insert leak
  // override defaults
  g_leak = 0.002  // S/cm2
  e_leak = -70    // mV
}

// show leak current density near 0 end of cable
print cable.i_leak(0.1)
```

The `leak` mechanism automatically appears with the other distributed mechanisms in GUI tools such as the Distributed Mechanism Inserter (Fig. 9.2). This is a consequence of interface code that is generated by the NMODL compiler when it parses the definitions in the `NEURON` block.

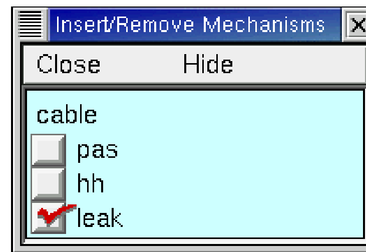


Figure 9.2. Compiling the leak mechanism automatically makes it available through NEURON's graphical user interface, as in this Distributed Mechanism Inserter (brought up by NEURON Main Menu / Tools / Distributed Mechanisms / Managers / Inserter). The check mark signifies that the `leak` mechanism has been inserted into the section named `cable`.

Example 9.2: a localized shunt

At the opposite end of the spatial scale from a distributed passive current is a localized shunt induced by microelectrode impalement [Durand, 1984 #291][Staley, 1992 #151]. A shunt is restricted to a small enough region that it can be described in terms of a net conductance (or resistance) and total current, i.e. it is a point process (see **Point processes** in **Chapter 5**). Most synapses are also best represented by point processes.

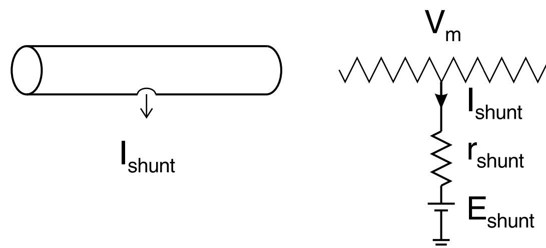


Figure 9.3

The localized nature of the shunt is emphasized in a cartoon of the neurite (Fig. 9.3 left). The equivalent circuit of the shunt (right) is similar to the equivalent circuit of the

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

distributed leak current (Fig. 9.1 right), but here the resistance and current are understood to be concentrated in a single, circumscribed part of the cell. We will focus on how the NMODL code for this localized shunt (Listing 9.2) differs from the leak distributed mechanism of **Example 9.1**.

The NEURON block

The POINT_PROCESS statement in the NEURON block identifies this mechanism as a point process, so it will be managed in hoc using an object oriented syntax (see **Usage** below). Declaring i, e, and r to be RANGE means that each instance of this point process can have separate values for these variables. If a variable is declared in a GLOBAL statement, then its value is shared among all instances of the mechanism (however, see **Equation definition blocks: The DERIVATIVE block** in **Example 9.5: a calcium-activated, voltage-dependent current**).

Variable declaration blocks

These are nearly identical to the PARAMETER and ASSIGNED blocks of the leak mechanism. However, Shunt is a point process so all of its current flows at one site instead of being distributed over an area. Therefore its i and r are in units of nanoamperes (total current) and gigaohms (0.001 / total conductance in microsiemens), respectively.

This code specifies default values for the PARAMETERs r and e. Allowing a minimum value of 10^{-9} for r prevents an inadvertent divide by 0 error (infinite conductance) by ensuring that a user cannot set r to 0 in its GUI field editor. However, as we noted in the leak model, the <*minval, maxval*> syntax does not prevent a hoc statement from assigning r a value outside of the desired range.

```
: A shunt current

NEURON {
  POINT_PROCESS Shunt
  NONSPECIFIC_CURRENT i
  RANGE i, e, r
}

PARAMETER {
  r = 1 (gigaohm)  < 1e-9, 1e9 >
  e = 0 (millivolt)
}

ASSIGNED {
  i  (nanoamp)
  v  (millivolt)
}

BREAKPOINT { i = (0.001)*(v - e)/r }
```

Listing 9.2. shunt.mod

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

Equation definition blocks

Like the leak current mechanism, the shunt mechanism is extremely simple and involves no state variables. Its single equation is defined in the `BREAKPOINT` block.

The `BREAKPOINT` block

The sole "complication" is that computation of `i` includes a factor of 0.001 to reconcile the units on the left and right hand sides of this assignment (nanoamperes vs. millivolts divided by gigaohms). The parentheses surrounding this conversion factor are a convention for units checking: they disambiguate it from mere multiplication by a number. When the NMODL code in Listing 9.2 is checked with NEURON's `modlunit` utility, no inconsistencies will be found.

```
[ted@fantom dshunt]$ modlunit shunt.mod
model   1.1.1.1   1994/10/12 17:22:51
Checking units of shunt.mod
[ted@fantom dshunt]$
```

However if the conversion factor were not enclosed by parentheses, there would be an error message that reports inconsistent units.

```
[ted@fantom dshunt]$ modlunit shunt.mod
model   1.1.1.1   1994/10/12 17:22:51
Checking units of shunt.mod
The previous primary expression with units: 1-12 coul/sec
is missing a conversion factor and should read:
  (0.001)*()
 at line 20 in file shunt.mod
        i = 0.001*(v - e)/r<<ERROR>>
[ted@fantom dshunt]$
```

An error message would also result if parentheses surrounded a number which the user intended to be a quantity, since the units would be inconsistent.

The simple convention of enclosing single numbers in parentheses to signify units conversion factors minimizes the possibility of mistakes, either by the user or by the software. It is important to note that expressions that involve more than one number, such as "(1 + 1)", will *not* be interpreted as units conversion factors.

Usage

This `hoc` code illustrates how the shunt mechanism might be applied to a section called `cable`; note the object syntax for specifying the shunt resistance and current (see **Point processes** in **Chapter 5**).

```
objref s
cable s = new Shunt(0.1) // put near 0 end of cable
s.r = 0.2 // pretty good for a sharp electrode
print s.i // show shunt current
```

The definitions in the `NEURON` block of this particular model enable NEURON's graphical tools to include the `Shunt` object in the menus of its PointProcessManager and Viewer windows (Fig. 9.4). The check mark on the button adjacent to the numeric field

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

for r indicates that the shunt resistance has been changed from its default value (0.2 gigaohm when the shunt was created by the hoc code immediately above) to 0.1 gigaohm.

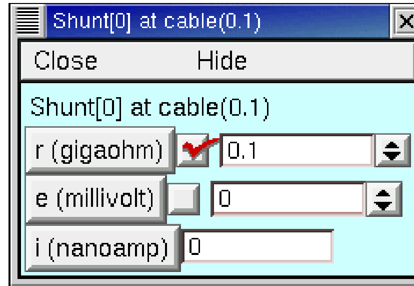


Figure 9.4. The properties of a specific instance of the Shunt mechanism are displayed in this Point Process Viewer (brought up by NEURON Main Menu / Tools / Point Processes / Viewers / PointProcesses / Shunt and then selecting Shunt[0] from the displayed list).

Example 9.3: an intracellular stimulating electrode

An intracellular stimulating electrode is similar to a shunt in the sense that both are localized sources of current that are modeled as point processes. However, the current from a stimulating electrode is not generated by an opening in the cell membrane, but instead is injected directly into the cell. This particular model of a stimulating electrode (Listing 9.3) has the additional difference that the current changes discontinuously, i.e. it is a pulse with distinct start and stop times.

The NEURON block

This mechanism is identical to NEURON's built-in IClamp. Calling it IClamp1 allows the reader to test and modify it without conflicting with the existing IClamp point process.

This model of a current clamp generates a rectangular current pulse whose amplitude amp in nanoamperes, start time del in milliseconds, and duration dur in milliseconds are all adjustable by the user. Furthermore, these parameters need to be individually adjustable for each separate instance of this mechanism, so they appear in a RANGE statement.

The current i delivered by IClamp1 is declared in the NEURON block to make it available for examination. The ELECTRODE_CURRENT statement has two important consequences: positive values of i will depolarize the cell (in contrast to the hyperpolarizing effect of positive transmembrane currents), and when the extracellular mechanism is present there will be a change in the extracellular potential vext.

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

Equation definition blocks

The BREAKPOINT block

The logic for deciding whether `i = 0` or `i = amp` is straightforward, but the `at_time()` calls need explanation. From the start we wish to emphasize that `at_time()` has become a "deprecated" function, i.e. it still works but it should not be used in future model development. We bring it up here because you may encounter it in legacy code. However, NEURON's event delivery system (see **Chapter 10**) provides a far better way to implement discontinuities.

To work properly with variable time step methods, e.g. CVODE, models that change parameters discontinuously during a simulation must notify NEURON when such events take place. With fixed time step methods, users implicitly assume that events occur on time step boundaries (integer multiples of `dt`), and they would never consider defining a pulse duration narrower than `dt`. Neither eventuality can be left to chance with variable time step methods.

When this mechanism is used in a variable time step simulation, the first `at_time()` call guarantees there will be a time step boundary just before `del`, and that integration will restart from a new initial condition just after `del` (see **Models with discontinuities** near the end of this chapter for more details).

```
: Current clamp

NEURON {
  POINT_PROCESS IClamp1
  RANGE del, dur, amp, i
  ELECTRODE_CURRENT i
}

UNITS { (nA) = (nanoamp) }

PARAMETER {
  del (ms)
  dur (ms)  < 0, 1e9 >
  amp (nA)
}

ASSIGNED { i (nA) }

INITIAL { i = 0 }

BREAKPOINT {
  at_time(del)
  at_time(del+dur)
  if (t < del + dur && t > del) {
    i = amp
  } else {
    i = 0
  }
}
```

Listing 9.3. `iclamp1.mod`

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

The `INITIAL` block

The code in the `INITIAL` block is executed when the standard run system's `finitialize()` is called. The initialization here consists of making sure that `IClamp1.i` is 0 when `t` = 0. Initialization of more complex mechanisms is discussed below in **Example 9.4: a voltage-gated current** and **Example 9.6: extracellular potassium accumulation**, and **Chapter 8** considers the topic of initialization from a broader perspective.

Usage

Regardless of whether a fixed or variable time step integrator is chosen, `IClamp1` looks the same to the user. In either case, a current stimulus of 0.01 nA amplitude that starts at `t` = 1 ms and lasts for 2 ms would be created by this `hoc` code

```
objref ccl
// put at middle of soma
soma ccl = new IClamp1(0.5)
ccl.del = 1
ccl.dur = 2
ccl.amp = 0.01
```

or through the PointProcessManager GUI tool (Fig. 9.5).

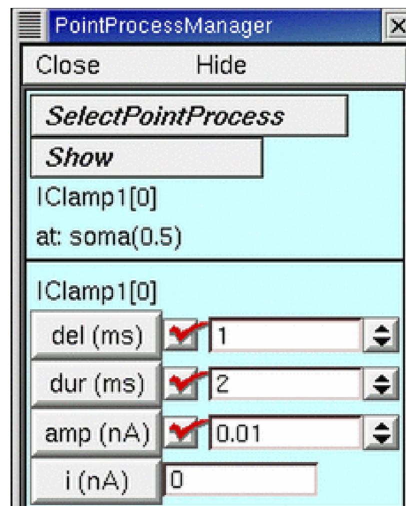


Figure 9.5. A PointProcessManager configured as an `Iclamp1` object.

Example 9.4: a voltage-gated current

One of the particular strengths of NMODL is its flexibility in dealing with ion channels whose conductances are not constant but instead are regulated by factors such as membrane potential and/or ligand concentrations on one or both sides of the membrane. Here we use the well known Hodgkin-Huxley (HH) delayed rectifier to show how a voltage-gated current can be implemented; in this example, membrane potential is in

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

absolute millivolts, i.e. reversed in polarity from the original Hodgkin-Huxley convention and shifted to reflect a resting potential of -65 mV. In **Example 9.5** we will examine a potassium current model that depends on both voltage and intracellular calcium concentration.

The delayed rectifier and all other ion channels that are distributed over the cell surface are distributed mechanisms. Therefore their NMODL representations and hoc usage be similar to those of **Example 9.1: a passive "leak" current**. The following discussion focuses on the significant differences between the implementations of the delayed rectifier and passive leak current models.

```
: HH voltage-gated potassium current

NEURON {
  SUFFIX kd
  USEION k READ ek WRITE ik
  RANGE gbar, g, i
}

UNITS {
  (S)  =  (siemens)
  (mV) =  (millivolt)
  (mA) =  (milliamp)
}

PARAMETER { gbar = 0.036 (S/cm2) }

ASSIGNED {
  v  (mV)
  ek (mV) : typically ~ -77.5
  ik (mA/cm2)
  i  (mA/cm2)
  g  (S/cm2)
}

STATE { n }

BREAKPOINT {
  SOLVE states METHOD cnexp
  g = gbar * n^4
  i = g * (v - ek)
  ik = i
}

INITIAL {
  : Assume v has been constant for a long time
  n = alpha(v)/(alpha(v) + beta(v))
}

DERIVATIVE states {
  : Computes state variable n at present v & t
  n' = (1-n)*alpha(v) - n*beta(v)
}
```

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

```
FUNCTION alpha(Vm (mV)) (/ms) {
  LOCAL x
  UNITSOFF
  x = (Vm+55)/10
  if (fabs(x) > 1e-6) {
    alpha = 0.1*x/(1 - exp(-x))
  } else {
    alpha = 0.1/(1 - 0.5*x)
  }
  UNITSON
}

FUNCTION beta(Vm (mV)) (/ms) {
  UNITSOFF
  beta = 0.125*exp(-(Vm+65)/80)
  UNITSON
}
```

Listing 9.4. `kd.mod`

The `NEURON` block

As with the passive leak model, `SUFFIX` marks this as a distributed mechanism, whose variables and parameters are identified in `hoc` by a particular suffix. Three `RANGE` variables are declared in this block: the peak conductance density `gbar` (the product of channel density and "open" conductance per channel), the macroscopic conductance `g` (the product of `gbar` and the fraction of channels that are open at any moment), and the current `i` that passes through `g`. At the level of `hoc`, these will be available as `gbar_kd`, `g_kd`, and `i_kd`.

This model also has a fourth range variable: the gating variable `n`, which is declared in the `STATE` block (see **The `STATE` block** below). `STATE` variables are automatically `RANGE` variables and do not need to be declared in the `NEURON` block.

A mechanism needs a separate `USEION` statement for each of the ions that it affects or that affect it. This example has one `USEION` statement, which includes `READ ek` because the potential gradient that drives `i_kd` depends on the equilibrium potential for K^+ (potassium). Since the resulting ionic flux may change local [K^+], this example also includes `WRITE ik`. The `WRITE ix` syntax enables NEURON to keep track of the total outward current that is carried by an ionic species, its internal and external concentrations, and its equilibrium potential. We will return to this point in the context of a model with extracellular K^+ accumulation.

The `UNITS` block

The statements in the `UNITS` block define new names for units in terms of existing names in the UNIX units database. This can increase legibility and convenience, and is helpful both as a reminder to the user and as a means for automating the process of checking for consistency of units.

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

Variable declaration blocks

The ASSIGNED block

This is analogous to the ASSIGNED block of the leak mechanism. For the sake of clarity, variables whose values are computed outside this mod file are listed first. Note that ek is listed as an ASSIGNED variable, unlike the leak mechanism's e which was a PARAMETER. The reason for this difference is that mechanisms that produce fluxes of specific ions, such as K^+, may cause the ionic equilibrium potential to change in the course of a simulation. However, the NONSPECIFIC_CURRENT generated by the leak mechanism was not linked to any particular ionic species, so e_leak remains fixed unless explicitly altered by hoc statements or the GUI.

The STATE block

If a model involves differential equations, families of algebraic equations, or kinetic reaction schemes, their dependent variables or unknowns are to be listed in the STATE block. Therefore gating variables such as the delayed rectifier's n are declared here.

In NMODL, variables that are declared in the STATE block are called STATE variables, or simply STATEs. This NMODL-specific terminology should not be confused with the physics or engineering concept of a "state variable" as a variable that describes the state of a system. While membrane potential is a "state variable" in the engineering sense, it would never be a STATE because its value is calculated only by NEURON and never by NMODL code. Likewise, the unknowns in a set of simultaneous equations (e.g. specified in a LINEAR or NONLINEAR block) would not be state variables in an engineering sense, yet they would all be STATEs (see **State variables and STATE variables** in **Chapter 8**).

All STATEs are automatically RANGE variables. This is appropriate, since channel gating can vary with position along a neurite.

Equation definition blocks

In addition to the BREAKPOINT block, this model also has INITIAL, DERIVATIVE, and FUNCTION blocks.

The BREAKPOINT block

This is the main computation block of the mechanism. By the end of the BREAKPOINT block, all variables are consistent with the new time. If a mechanism has STATEs, this block must contain one SOLVE statement that tell how the values of the STATEs will be computed over each time step. The SOLVE statement specifies a block of code that defines the simultaneous equations that govern the STATEs. Currents are set with assignment statements at the end of the BREAKPOINT block.

There are two major reasons why variables that depend on the number of executions, such as counts or flags or random numbers, should generally not be calculated in a

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

BREAKPOINT block. First, the assignment statements in a BREAKPOINT block are usually called twice per time step. Second, with variable time step methods the value of t may not even be monotonically increasing. The proper way to think about this is to remember that the BREAKPOINT block is responsible for making all variables consistent at time t. Thus assignment statements in this block are responsible for trivially specifying the values of variables that depend *only* on the values of STATEs, t, and v, while the SOLVE statements perform the magic required to make the STATEs consistent at time t. It is not belaboring the point to reiterate that the assignment statements should produce the same result regardless of how many times BREAKPOINT is called with the same STATEs, t, and v. All too often, errors have resulted from an attempt to explicitly compute what is conceptually a STATE in a BREAKPOINT block. Computations that must be performed only once per time step should be placed in a PROCEDURE, which in turn would be invoked by a SOLVE statement in a BREAKPOINT block.

We must also emphasize that the SOLVE statement is not a function call, and that the body of the DERIVATIVE block (or any other block specified in a SOLVE statement) will be executed asynchronously with respect to BREAKPOINT assignment statements. Therefore it is incorrect to invoke rate functions from the BREAKPOINT block; instead these must be called from the block that is specified by the SOLVE statement (in this example, from within the DERIVATIVE block).

Models of active currents such as i_kd are generally formulated in terms of ionic conductances governed by gating variables that depend on voltage and time. The SOLVE statements at the beginning of the BREAKPOINT block specify the differential equations or kinetic schemes that govern the kinetics of the gating variables. The algebraic equations that compute the ionic conductances and currents follow the SOLVE statements.

The INITIAL block

Though often overlooked, proper initialization of *all* STATEs is as important as correctly computing their temporal evolution. This is accomplished for the common case by the standard run system's finitialize(), which executes the initialization strategy defined in the INITIAL block of each mechanism (see also **INITIAL blocks in NMODL** in **Chapter 8**). The INITIAL block may contain any instructions that should be executed when the hoc function finitialize() is called.

Prior to executing the INITIAL block, STATE variables are set to the values asserted in the STATE block (or to 0 if no specific value was given in the STATE block). A NET_RECEIVE block, if present, may also have its own INITIAL block for nonzero initialization of NetCon "states" (the NET_RECEIVE block and its initialization are discussion further in **Chapter 10** and under **Basic initialization in NEURON: finitialize()** in **Chapter 8**).

The INITIAL block should be used to initialize STATEs with respect to the initial values of membrane potential and ionic concentrations. There are several other ways to prepare STATEs for a simulation run, the most direct of which is simply to assign values explicitly with hoc statements such as axon.n_kd(0.3) = 0.9. However, this particular strategy can create arbitrary initial conditions that would be quite unnatural. A

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

more "physiological" approach, which may be appropriate for models of oscillating or chaotic systems or whose mechanisms show other complex interactions, is to perform an initialization run during which the model converges toward its limit cycle or attractor. One practical alternative for systems that settle to a stable equilibrium point when left undisturbed is to assign `t` a large negative value and then advance the simulation over several large time steps (keeping `t` < 0 prevents the initialization steps from triggering scheduled events such as stimulus currents or synaptic inputs); this takes advantage of the strong stability properties of NEURON's implicit integration methods (see **Chapter 4**). For a more extensive discussion of initialization, see **Chapter 8**, especially **Examples of custom initializations**).

This delayed rectifier mechanism sets `n` to its steady state value for the local membrane potential wherever the mechanism has been inserted. This potential itself can be "left over" from a previous simulation run, or it can be specified by the user, e.g. uniformly over the entire cell with a statement like `finitialize(-55)`, or on a compartment by compartment basis by asserting statements such as `dend.v(0.2) = -48` before calling `finitialize()` (see **Default initialization in the standard run system: `stdinit()` and `init()`** in **Chapter 8**).

The `DERIVATIVE` block

This is used to assign values to the derivatives of those `STATE`s that are described by differential equations. The statements in this block are of the form `y' = expr`, where a series of apostrophes can be used to signify higher order derivatives.

In fixed time step simulations, these equations are integrated using the numerical method specified by the `SOLVE` statement in the `BREAKPOINT` block. The `SOLVE` statement should explicitly invoke one of the integration methods that is appropriate for systems in which state variables can vary widely during a time step (stiff systems). The `cnexp` method, which combines second order accuracy with computational efficiency, is a good choice for this example. It is appropriate when the right hand side of $y' = f(v,y)$ is linear in y and involves no other states, so it is well suited to models with HH-style ionic currents. This method calculates the `STATE`s analytically under the assumption that all other variables are constant throughout the time step. If the variables change but are second order correct at the midpoint of the time step, then the calculation of `STATE`s is also second order correct.

If $f(v,y)$ is not linear in y, then the `SOLVE` statement in the `BREAKPOINT` block should specify the implicit integration method `derivimplicit`. This provides first order accuracy and is usable with general ODEs regardless of stiffness or nonlinearity.

Other integrators, such as `runge` and `euler`, are defined but are not useful in the NEURON context. Neither is guaranteed to be numerically stable, and `runge`'s high order accuracy is wasted since voltage does not have an equivalent order of accuracy.

With variable time step methods, *no* variable is assumed to be constant. These methods not only change the time step, but adaptively choose a numerical integration formula with accuracy that ranges from first order up to $O(\Delta t^6)$. The present implementation of NMODL creates a diagonal Jacobian approximation for the block of

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

STATEs. This is done analytically if $y_i' = f_i(v,y)$ is polynomial in y_i; otherwise, the Jacobian is approximated by numerical differencing. In the rare case where this is inadequate, the user may supply an explicit Jacobian. Future versions of NMODL may attempt to deal with Jacobian evaluation in a more sophisticated manner. This illustrates a particularly important benefit of the NMODL approach: improvements in methods do not affect the high level description of the membrane mechanism.

The FUNCTION block

The functions defined by FUNCTION blocks are available at the hoc level and in other mechanisms by adding the suffix of the mechanism in which they are defined, e.g. alpha_kd() and beta_kd(). Functions or procedures can be simply called from hoc if they do not reference range variables (references to GLOBAL variables are allowed). If a function or procedure does reference a range variable, then prior to calling the function from hoc it is necessary to specify the proper instance of the mechanism (its location on the cell). This is done by a setdata_ function that has the syntax

```
section_name setdata_suffix(x)
```

where *section_name* is the name of the section that contains the mechanism in question, *suffix* is the mechanism suffix, and x is the normalized distance along the section where the particular instance of the mechanism exists. The functions in our kd example do not use range variables, so a specific instance is not needed.

The differential equation that describes the kinetics of n involves two voltage-dependent rate constants whose values are computed by the functions alpha() and beta(). The original algebraic form of the equations that define these rates is

$$\alpha = \frac{0.1\left(\frac{v+55}{10}\right)}{1 - e^{-\left(\frac{v+55}{10}\right)}} \quad \text{and} \quad \beta = 0.125\ e^{-\left(\frac{v+65}{80}\right)} \qquad \text{Eq. 9.1}$$

The denominator for α goes to 0 when v = -55 mV, which could cause numeric overflow. The code used in alpha() avoids this by switching, when v is very close to -55, to an alternative expression that is based on the first three terms of the infinite series expansion of e^x.

As noted elsewhere in this paper, NMODL has features that facilitate establishing and maintaining consistency of units. Therefore the rate functions alpha() and beta() are introduced with the syntax

```
FUNCTION f_name(arg1 (units1), arg2 (units2), . . . )(returned_units)
```

to declare that their arguments are in units of millivolts and that their returned values are in units of inverse milliseconds ("/ms"). This allows automatic units checking on entry to and return from these functions. For the sake of legibility the UNITSOFF . . . UNITSON directives disable units checking just within the body of these functions. This is acceptable because the terms in the affected statements are mutually consistent.

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

Otherwise the statements would have to be rewritten in a way that makes unit consistency explicit at the cost of legibility, e.g.

```
x = (Vm + 55 (millivolt))/(10 (millivolt))
```

Certain variables exist solely for the sake of computational convenience. These typically serve as scale factors, flags, or temporary storage for intermediate results, and are not of primary importance to the mechanism. Such variables are often declared as `LOCAL` variables declared within an equation block, e.g. `x` in this mechanism. `LOCAL` variables that are declared in an equation block are not "visible" outside the block and they do not retain their values between invocations of the block. `LOCAL` variables that are declared outside an equation block have very different properties and are discussed under **Variable declaration blocks** in **Example 9.8: calcium diffusion with buffering**.

Usage

The `hoc` code and graphical interface for using this distributed mechanism are similar to those for the `leak` mechanism (Fig. 9.2). However, the `kd` mechanism involves more range variables, and this is reflected in the choices available in the range variable menu of variable browsers, such as the Plot what? tool (brought up from the primary menu of a Graph). Since `kd` uses potassium, the variables `ek` and `ik` (total K^+ current) appear in this list along with the variables that are explicitly declared in `RANGE` statements or the `STATE` block of `kd.mod` (see Fig. 9.6). The total K^+ current `ik` will differ from `i_kd` only if another mechanism that `WRITE`s `ik` is present in this section.

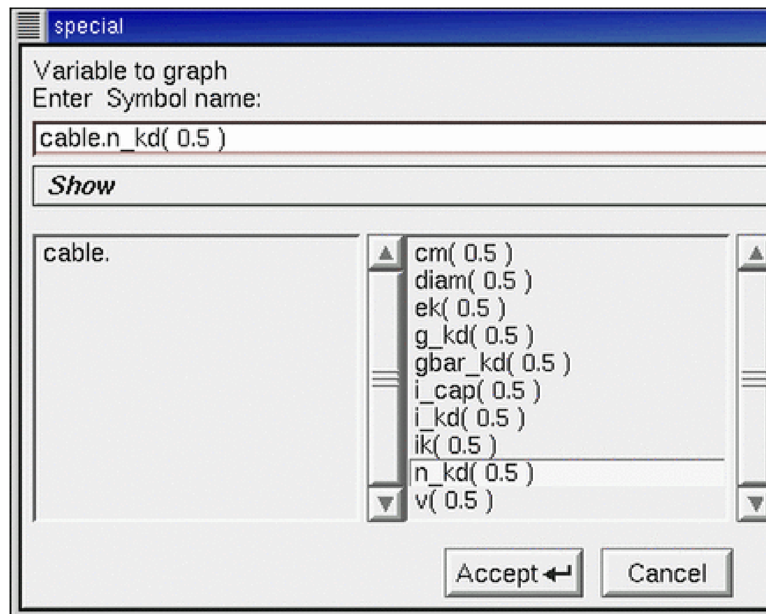


Figure 9.6. A Plot what? tool from a Graph created after the `kd` mechanism was inserted into a section called `cable`. Note the `hoc` names of variables associated with the `kd` mechanism.

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

Example 9.5: a calcium-activated, voltage-gated current

This model of a potassium current that depends on both voltage and intracellular calcium concentration $[Ca^{2+}]_i$. is based on the work of Moczydlowski and Latorre [Moczydlowski, 1983 #306]. It is basically an elaboration of the HH mechanism in which the forward and backward rates depend jointly on membrane potential and $[Ca^{2+}]_i$. Here we point out the salient implementational differences between this and the previous model.

```
: Calcium-activated K channel

NEURON {
  SUFFIX cagk
  USEION ca READ cai
  USEION k READ ek WRITE ik
  RANGE gkbar
  GLOBAL oinf, tau
}

UNITS {
  (mV)    = (millivolt)
  (mA)    = (milliamp)
  (S)     = (siemens)
  (molar) = (1/liter)
  (mM)    = (millimolar)
  FARADAY = (faraday) (kilocoulombs)
  R       = (k-mole) (joule/degC)
}

PARAMETER {
  gkbar = 0.01  (S/cm2)
  d1    = 0.84
  d2    = 1.0
  k1    = 0.18  (mM)
  k2    = 0.011 (mM)
  bbar  = 0.28  (/ms)
  abar  = 0.48  (/ms)
}

ASSIGNED {
  cai      (mM)    : typically 0.001
  celsius  (degC)  : typically 20
  v        (mV)
  ek       (mV)
  ik       (mA/cm2)
  oinf
  tau      (ms)
}
```

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

```
STATE { o }    : fraction of channels that are open

BREAKPOINT {
  SOLVE state METHOD cnexp
  ik = gkbar*o*(v - ek)
}

DERIVATIVE state {
  rate(v, cai)
  o' = (oinf - o)/tau
}

INITIAL {
  rate(v, cai)
  o = oinf
}

: the following are all callable from hoc

FUNCTION alp(v (mV), ca (mM)) (/ms) {
  alp = abar/(1 + exp1(k1,d1,v)/ca)
}

FUNCTION bet(v (mV), ca (mM)) (/ms) {
  bet = bbar/(1 + ca/exp1(k2,d2,v))
}

FUNCTION exp1(k (mM), d, v (mV)) (mM) {
  : numeric constants in an addition or subtraction
  : expression automatically take on the unit values
  : of the other term
  exp1 = k*exp(-2*d*FARADAY*v/R/(273.15 + celsius))
}

PROCEDURE rate(v (mV), ca (mM)) {
  LOCAL a
  : LOCAL variable takes on units of right hand side
  a = alp(v,ca)
  tau = 1/(a + bet(v, ca))
  oinf = a*tau
}
```

Listing 9.5. `cagk.mod`

The `NEURON` block

This potassium conductance depends on $[Ca^{2+}]_i$, so two `USEION` statements are required. Since this potassium channel depends on intracellular calcium concentration, it must `READ cai`. The `RANGE` statement declares the peak conductance density `gkbar`. However, there is no `g`, so this mechanism's ionic conductance will not be visible from `hoc` (in fact, this model doesn't even calculate the activated ionic conductance density). Likewise, there is no `i_cagk` to report this particular current component separately, even though it will be added to the total K^+ current `ik` because of `WRITE ik`.

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

The variables `oinf` and `tau`, which govern the gating variable `o`, should be accessible in `hoc` for the purpose of seeing how they vary with membrane potential and $[Ca^{2+}]_i$. At the same time, the storage and syntax overhead required for a `RANGE` variable does not seem warranted because it appears unlikely to be necessary or useful to plot either `oinf` or `tau` as a function of space. Therefore they have been declared in a `GLOBAL` statement. On first examination, this might seem to pose a problem. The gating of this K^+ current depends on membrane potential and $[Ca^{2+}]_i$, both of which may vary with location, so how can it be correct for `oinf` and `tau` to be `GLOBAL`s? And if some reason did arise to examine the values of these variables at a particular location, how could this be done? The answers to these questions lie in the `DERIVATIVE` and `PROCEDURE` blocks, as we shall see below.

The `UNITS` block

The last two statements in this block require some explanation. The first parenthesized item on the right hand side of the equal sign is the numeric value of a standard entry in `nrnunits.lib`, which may be expressed on a scale appropriate for physics rather than membrane biophysics. The second parenthesized item rescales this to the biophysically appropriate units chosen for this model. Thus `(faraday)` appears in the units database in terms of coulombs/mole and has a numeric value of 96,485.309, but for this particular mechanism we prefer to use a constant whose units are kilocoulombs/mole. The statement

```
FARADAY = (faraday) (kilocoulombs)
```

results in `FARADAY` having units of kilocoulombs and a numeric value of 96.485309. The item `(k-mole)` in the statement

```
R       = (k-mole) (joule/degC)
```

is not kilomoles but instead is a specific entry in the units database equal to the product of Boltzmann's constant and Avogadro's number. The end result of this statement is that `R` has units of joules/°C and a numeric value of 8.313424. These special definitions of `FARADAY` and `R` pertain to this mechanism only; a different mechanism could assign different units and numeric values to these labels.

Another possible source of confusion is the interpretation of the symbol `e`. Inside a UNITS block this is always the electronic charge ($\sim 1.6 \cdot 10^{-19}$ coulombs), but elsewhere a *single* number in parentheses is treated as a units conversion factor, e.g. the expression `(2e4)` is a conversion factor of $2 \cdot 10^4$. Errors involving `e` in a units expression are easy to make, but they are always caught by `modlunit`.

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

Variable declaration blocks

The ASSIGNED block

Comments in this block can be helpful to the user as reminders of "typical" values or usual conditions under which a mechanism operates. For example, the cagk mechanism is intended for use in the context of $[Ca^{2+}]_i$ on the order of 0.001 mM. Similarly, the temperature sensitivity of this mechanism is accommodated by including the global variable celsius, which is used to calculate the rate constants (see **The FUNCTION and PROCEDURE blocks** below). NEURON's default value for celsius is 6.3ºC, but as the comment in this mod file points out, the parameter values for this particular mechanism were intended for an "operating temperature" of 20ºC. Therefore the user may need to change celsius through hoc or the GUI.

The variables oinf and tau, which were made accessible to NEURON by the GLOBAL statement in the NEURON block, are given values by the procedure rate and are declared as ASSIGNED.

The STATE block

This mechanism needs a STATE block because o, the fraction of channels that are open, is described by a differential equation.

Equation definition blocks

The BREAKPOINT block

This mechanism does not make its ionic conductance available to hoc, so the BREAKPOINT block just calculates the ionic current passing through these channels and doesn't bother with separate computation of a conductance.

The DERIVATIVE block

The gating variable o is governed by a first order differential equation. The procedure rate() assigns values to the voltage sensitive parameters of this equation: the steady state value oinf, and the time constant tau. This answers the first question that was raised above in the discussion of the NEURON block. The procedure rate() will be executed individually for each segment in the model that has the cagk mechanism. Each time rate() is called, its arguments will equal the membrane potential and $[Ca^{2+}]_i$ of the segment that is being processed, since v and cai are both RANGE variables. Therefore oinf and tau can be GLOBAL without destroying the spatial variation of the gating variable o.

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

The `FUNCTION` and `PROCEDURE` blocks

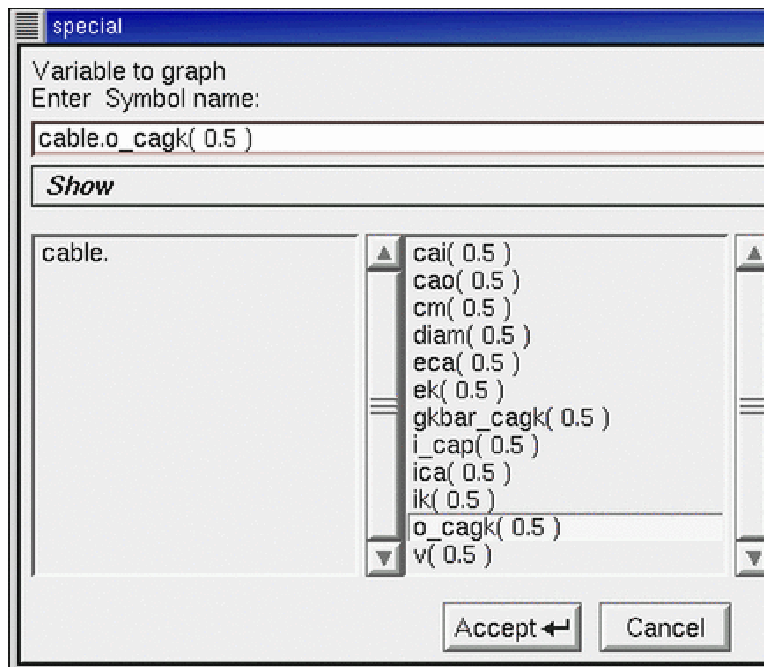
The functions `alp()`, `bet()`, `exp1()`, and the procedure `rate()` implement the mathematical expressions that describe `oinf` and `tau`. To facilitate units checking, their arguments are tagged with the units that they use. For efficiency, `rate()` calls `alp()` once and uses the returned value twice; calculating `oinf` and `tau` separately would have required two calls to `alp()`.

Now we can answer the second question that was raised in the discussion of the `NEURON` block: how to examine the variation of `oinf` and `tau` over space. This is easily done in `hoc` with nested loops, e.g.

```
forall { // iterate over all sections
  for (x) { // iterate over each segment
    rate_cagk(v(x), cai(x))
    // here put statements to plot
    //   or save oinf and tau
  }
}
```

Usage

This mechanism involves both K^+ and Ca^{2+}, so the list of `RANGE` variables displayed by Plot what? has more entries than it did for the `kd` mechanism (Fig. 9.7; compare this with Fig. 9.6). However, `cai`, `cao`, and `eca` will remain constant unless the section in which this mechanism has been inserted also includes something that can affect calcium concentration (e.g. a pump or buffer).



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

Figure 9.7. A Plot what? tool from a Graph created after the `cagk` mechanism was inserted into a section called `cable`. Note the `hoc` names of variables associated with the `cagk` mechanism.

Example 9.6: extracellular potassium accumulation

Because mechanisms can generate transmembrane fluxes that are attributed to specific ionic species by the `USEION` *x* `WRITE` *ix* syntax, modeling the effects of restricted diffusion is straightforward. The `kext` mechanism described here emulates potassium accumulation in the extracellular space adjacent to squid axon (Fig. 9.8). The experiments of Frankenhaeuser and Hodgkin [Frankenhaeuser, 1956 #307] indicated that satellite cells and other extracellular structures act as a diffusion barrier that prevents free communication between this space and the bath. When there is a large efflux of K^+ ions from the axon, e.g. during the repolarizing phase of an action potential or in response to injected depolarizing current, K^+ builds up in this "Frankenhaeuser-Hodgkin space" (F-H space). This elevation of $[K^+]_o$ shifts E_K in a depolarized direction, which has two important consequences. First, it reduces the driving force for K^+ efflux and causes a decline of the outward I_K. Second, when the action potential terminates or the injected depolarizing current is stopped, the residual elevation of $[K^+]_o$ causes an inward current that decays gradually as $[K^+]_o$ equilibrates with $[K^+]_{bath}$.

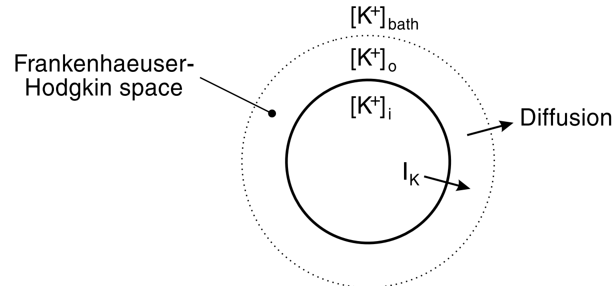


Figure 9.8. Restricted diffusion may cause extracellular potassium accumulation adjacent to the cell membrane. From Fig. 1 in [Hines, 2000 #323].

```
: Extracellular potassium ion accumulation

NEURON {
  SUFFIX kext
  USEION k READ ik WRITE ko
  GLOBAL kbath
  RANGE fhspace, txfer
}
```

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

```
UNITS {
  (mV)    = (millivolt)
  (mA)    = (milliamp)
  FARADAY = (faraday) (coulombs)
  (molar) = (1/liter)
  (mM)    = (millimolar)
}

PARAMETER {
  kbath   =  10 (mM)        : seawater (squid axon!)
  fhspace = 300 (angstrom)  : effective thickness of F-H space
  txfer   =  50 (ms)  : tau for F-H space <-> bath exchange = 30-100
}

ASSIGNED { ik  (mA/cm2) }

STATE { ko  (mM) }

BREAKPOINT { SOLVE state METHOD cnexp }

DERIVATIVE state {
  ko' = (1e8)*ik/(fhspace*FARADAY) + (kbath - ko)/txfer
}
```

Listing 9.6. `kext.mod`

The `NEURON` block

A compartment may contain several mechanisms that have direct interactions with ionic concentrations (e.g. diffusion, buffers, pumps). Therefore NEURON must be able to compute the total currents and concentrations consistently. The `USEION` statement sets up the necessary "bookkeeping" by automatically creating a separate mechanism that keeps track of four essential variables: the total outward current carried by an ion, the internal and external concentrations of the ion, and its equilibrium potential (also see **Ion concentrations and equilibrium potential** in **Chapter 8**). In this case the name of the ion is "`k`" and the automatically created mechanism is called "`k_ion`" in the `hoc` interpreter. The `k_ion` mechanism has variables `ik`, `ki`, `ko`, and `ek`, which represent I_K, $[K^+]_i$, $[K^+]_o$, and E_K, respectively. These do not have suffixes; furthermore, they are `RANGE` variables so they can have different values in every segment of each section in which they exist. In other words, the K^+ current through Hodgkin-Huxley potassium channels near one end of the section `cable` is `cable.ik_hh(0.1)`, but the total K^+ current generated by all sources, including other ionic conductances and pumps, would be `cable.ik(0.1)`.

This mechanism computes $[K^+]_o$ from the outward potassium current, so it `READ`s `ik` and `WRITE`s `ko`. When a mechanism `WRITE`s a particular ionic concentration, it sets the value for that concentration at all locations in every section into which it has been inserted. This has an important consequence: in any given section, no ionic concentration should be "written" by more than one mechanism.

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

The bath is assumed to be a large, well stirred compartment that envelops the entire "experimental preparation." Therefore kbath is a GLOBAL variable so that all sections that contain the kext mechanism will have the same numeric value for $[K^+]_{bath}$. Since this would be one of the controlled variables in an experiment, the value of kbath is specified by the user and remains constant during a simulation. The thickness of the F-H space is fhspace, the time constant for equilibration with the bath is txfer, and both are RANGE variables so they can vary along the length of each section.

Variable declaration blocks

The PARAMETER block

The default value of kbath is set to 10 mM, consistent with the composition of seawater [Frankenhaeuser, 1956 #307]. Since kbath is GLOBAL, a single hoc statement can change this to a new value that will affect all occurrences of the kext mechanism, e.g. kbath_kext = 8 would change it to 8 mM everywhere.

The STATE block

Ionic concentration is a STATE of a mechanism only if that mechanism calculates the concentration in a DERIVATIVE or KINETIC block. This model computes ko, the potassium concentration in the F-H space, according to the dynamics specified by an ordinary differential equation.

Equation definition blocks

The BREAKPOINT block

This mechanism involves a single differential equation that tells the rate of change of ko, the K^+ concentration in the F-H space. The choice of integration method in NMODL is based on the fact that the equation is linear in ko. The total K^+ current ik might also vary during a time step (see the DERIVATIVE block) if membrane potential, some K^+ conductance, or ko itself is changing rapidly. In a simulation where such rapid changes are likely to occur, proper modeling practice would lead one either to use NEURON with CVODE, or to use a fixed time step that is short compared to the rate of change of ik.

The INITIAL block

How to provide for initialization of variables is a recurring question in model implementation, and here it comes again. The answer is important because it bears directly on how the model will be used. The only STATE in this mechanism is the ionic concentration ko, which we could initialize in several different ways. The simplest might be with the INITIAL block

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

```
INITIAL {
  ko = kbath
}
```

but this seems too limiting. One alternative is to declare a new RANGE variable ko0 in the NEURON block, specify its value in the PARAMETER block

```
PARAMETER {
   . . .
  ko0     =  10 (mM)   =   10 (mM)
}
```

and use this INITIAL block

```
INITIAL {
  ko = ko0
}
```

This would be a very flexible implementation, allowing ko0 to vary with location wherever kext has been inserted. But some care is needed in its use, because ion concentration assignment in an INITIAL block can result in an inconsistent initialization on return from finitialize() (see **Ion concentrations and equilibrium potentials** in **Chapter 8**).

So for this example we have decided to let the initial value of ko be controlled by the built-in hoc variable ko0_k_ion (see ***Initializing concentrations in* hoc** in **Chapter 8**). To make our mechanism rely on ko0_k_ion for the initial value of ko, we merely omit any ko = . . . assignment statement from the INITIAL block. Since ko is kext's only STATE, we don't need an INITIAL block at all. This might seem a less flexible approach than using our own ko0 RANGE variable, because ko0_k_ion is a global variable (has the same value wherever ko is defined), but ***Initializing concentrations in* hoc** in **Chapter 8** shows how to work around this apparent limitation.

The DERIVATIVE block

At the core of this mechanism is a single differential equation that relates $d[K^+]_o/dt$ to the sum of two terms. The first term describes the contribution of ik to $[K^+]_o$, subject to the assumption that the thickness F-H space is much smaller than the diameter of the section. The units conversion factor of 10^8 is required because fhspace is given in Ångstroms. The second term describes the exchange of K^+ between the bath and the F-H space.

Usage

If this mechanism is present in a section, the following RANGE variables will be accessible through hoc: $[K^+]$ inside the cell and within the F-H space (ki and ko); equilibrium potential and total current for K (ek and ik); thickness of the F-H space and the rate of equilibration between it and the bath (fhspace_kext and txfer_kext). The bath $[K^+]$ will also be available as the global variable kbath_kext.

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

General comments about kinetic schemes

Kinetic schemes provide a high level framework that is perfectly suited for compact and intuitively clear specification of models that involve discrete states in which material is conserved. The basic notion is that flow out of one state equals flow into another (also see **Chemical reactions** in **Chapter 3**). Almost all models of membrane channels, chemical reactions, macroscopic Markov processes, and diffusion can be elegantly expressed through kinetic schemes. It will be helpful to review some fundamentals before proceeding to specific examples of mechanisms implemented with kinetic schemes.

The unknowns in a kinetic scheme, which are usually concentrations of individual reactants, are declared in the `STATE` block. The user expresses the kinetic scheme with a notation that is very similar to a list of simultaneous chemical reactions. The NMODL translator converts the kinetic scheme into a family of ODEs whose unknowns are the `STATE`s. Hence

```
STATE { mc   m }
KINETIC scheme1 {
  ~ mc <-> m (a(v), b(v))
}
```

is equivalent to

```
DERIVATIVE scheme1 {
  mc' = -a(v)*mc + b(v)*m
  m'  =  a(v)*mc - b(v)*m
}
```

The first character of a reaction statement is the tilde "~", which is used to immediately distinguish this kind of statement from other sequences of tokens that could be interpreted as an expression. The expressions on the left and right of the three character reaction indicator "`<->`" specify the reactants. The two expressions in parentheses are the forward and reverse reaction rates (here the rate functions `a(v)` and `b(v)`). Immediately after each reaction, the variables `f_flux` and `b_flux` are assigned the values of the forward and reverse fluxes respectively. These can be used in assignment statements such as

```
~ cai + pump <-> capump (k1,k2)
~ capump <-> pump + cao (k3,k4)
ica = (f_flux - b_flux)*2*Faraday/area
```

In this case, the forward flux is `k3*capump`, the reverse flux is `k4*pump*cao`, and the "positive outward" current convention is consistent with the sign of the expression for `ica` (in the second reaction, forward flux means positive ions move from the inside to the outside).

More complicated reaction sequences such as the wholly imaginary

```
KINETIC scheme2 {
  ~ 2A + B <-> C (k1,k2)
  ~  C + D <-> A + 2B (k3,k4)
}
```

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

begin to show the clarity of expression and suggest the comparative ease of modification of the kinetic representation over the equivalent but stoichiometrically confusing

```
DERIVATIVE scheme2 {
  A' = -2*k1*A^2*B + 2*k2*C   + k3*C*D   - k4*A*B^2
  B' =   -k1*A^2*B   + k2*C + 2*k3*C*D - 2*k4*A*B^2
  C' =    k1*A^2*B   - k2*C   - k3*C*D   + k4*A*B^2
  D' =                        - k3*C*D   + k4*A*B^2
}
```

Clearly a statement such as

```
~ calmodulin + 3Ca <-> active (k1, k2)
```

would be easier to modify (e.g. so it requires combination with 4 calcium ions) than the relevant term in the three differential equations for the STATEs that this reaction affects. The kinetic representation is easy to debug because it closely resembles familiar notations and is much closer to the conceptualization of what is happening than the differential equations would be.

Another benefit of kinetic schemes is the simple polynomial nature of the flux terms, which allows the translator to easily perform a great deal of preprocessing that makes implicit numerical integration more efficient. Specifically, the nonzero $\partial y'_i / \partial y_j$ elements (partial derivatives of dy_i / dt with respect to y_j) of the sparse matrix are calculated analytically in NMODL and collected into a C function that is called by solvers to calculate the Jacobian. Furthermore, the form of the reaction statements determines if the scheme is linear, obviating an iterative computation of the solution. Voltage-sensitive rates are allowed, but to guarantee numerical stability, reaction rates should not be functions of STATEs. Thus writing the calmodulin example as

```
~ calmodulin <-> active (k3*Ca^3, k2)
```

will work but is potentially unstable if Ca is a STATE in other simultaneous reactions in the same mod file. Variable time step methods such as CVODE will compensate by reducing dt, but this will make the simulation run more slowly.

Kinetic scheme representations provide a great deal of leverage because a single compact expression is equivalent to a large amount of C code. One special advantage from the programmer's point of view is the fact that these expressions are independent of the solution method. Different solution methods require different code, but the NMODL translator generates this code automatically. This saves the user's time and effort and ensures that all code expresses the same mechanism. Another advantage is that the NMODL translator handles the task of interfacing the mechanism to the remainder of the program. This is a tedious exercise that would require the user to have special knowledge that is not relevant to neurophysiology and which may change from version to version.

Special issues are raised by mechanisms that involve fluxes between compartments of different size, or whose reactants have different units. The first of the following examples has none of these complications, which are addressed later in models of diffusion and active transport.

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

Example 9.7: kinetic scheme for a voltage-gated current

This illustration of NMODL's facility for handling kinetic schemes implements a simple three state model for the conductance state transitions of a voltage gated potassium current

$$C_1 \underset{kb_1}{\overset{kf_1}{\rightleftarrows}} C_2 \underset{kb_2}{\overset{kf_2}{\rightleftarrows}} O \qquad \text{Eq. 9.2}$$

The closed states are C_1 and C_2, the open state is O, and the rates of the forward and backward state transitions are calculated in terms of the equilibrium constants and time constants of the isolated reactions through the familiar expressions $K_i(v) = kf_i / kb_i$ and $\tau_i(v) = 1/(kf_i + kb_i)$. The equilibrium constants $K_i(v)$ are the Boltzmann factors $K_1 = e^{[k_2(d_2 - v) - k_1(d_1 - v)]}$ and $K_2 = e^{-k_2(d_2 - v)}$, where the energies of states C_1, C_2, and O are 0, $k_1(d_1 - v)$, and $k_2(d_2 - v)$ respectively.

The typical sequence of analysis is to determine the constants k_1, d_1, k_2, and d_2 by fitting the steady state voltage clamp data, and then to find the voltage sensitive transition time constants $\tau_1(v)$ and $\tau_2(v)$ from the temporal properties of the clamp current at each voltage pulse level. In this example the steady state information has been incorporated in the NMODL code, and the time constants are conveyed by tables (arrays) that are created within the interpreter.

```
: Three state kinetic scheme for HH-like potassium channel
: Steady state v-dependent state transitions have been fit
: Needs v-dependent time constants
:   from tables created under hoc

NEURON {
  SUFFIX k3st
  USEION k READ ek WRITE ik
  RANGE g, gbar
}

UNITS { (mV) = (millivolt) }

PARAMETER {
  gbar = 33     (millimho/cm2)
  d1   = -38    (mV)
  k1   = 0.151  (/mV)
  d2   = -25    (mV)
  k2   = 0.044  (/mV)
}
```

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

```
ASSIGNED {
  v     (mV)
  ek    (mV)
  g     (millimho/cm2)
  ik    (milliamp/cm2)
  kf1   (/ms)
  kb1   (/ms)
  kf2   (/ms)
  kb2   (/ms)
}

STATE { c1 c2 o }

BREAKPOINT {
  SOLVE kin METHOD sparse
  g = gbar*o
  ik = g*(v - ek)*(1e-3)
}

INITIAL { SOLVE kin STEADYSTATE sparse }

KINETIC kin {
  rates(v)
  ~ c1 <-> c2   (kf1, kb1)
  ~ c2 <-> o    (kf2, kb2)
  CONSERVE c1 + c2 + o = 1
}

FUNCTION_TABLE tau1(v(mV)) (ms)
FUNCTION_TABLE tau2(v(mV)) (ms)

PROCEDURE rates(v(millivolt)) {
  LOCAL K1, K2
  K1 = exp(k2*(d2 - v) - k1*(d1 - v))
  kf1 = K1/(tau1(v)*(1+K1))
  kb1 = 1/(tau1(v)*(1+K1))
  K2 = exp(-k2*(d2 - v))
  kf2 = K2/(tau2(v)*(1+K2))
  kb2 = 1/(tau2(v)*(1+K2))
}
```

Listing 9.7. `k3st.mod`

The `NEURON` block

With one exception, the `NEURON` block of this model is essentially the same as for the delayed rectifier presented in **Example 9.4: a voltage-gated current**. The difference is that, even though this model contributes to the total K^+ current `ik`, its own current is not available separately (i.e. there will be no `ik_k3st` at the `hoc` level) because `ik` is not declared as a `RANGE` variable.

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

Variable declaration blocks

The STATE block

The STATEs in this mechanism are the fractions of channels that are in closed states 1 or 2 or in the open state. Since the total number of channels in all states is conserved, the sum of the STATEs must be unity, i.e. c1 + c2 + o = 1. This conservation rule means that the k3st mechanism really has only two independent STATE variables, a fact that underscores the difference between a STATE in NMODL and the general concept of a state variable. It also affects how NMODL sets up the equations that are to be solved, as we will see in the discussion of the KINETIC block below.

Not all reactants have to be STATEs. If the reactant is an ASSIGNED or PARAMETER variable, then a differential equation is not generated for it, and it is treated as constant for the purposes of calculating the declared STATEs. Statements such as

```
PARAMETER {kbath (mM)}
STATE {ko (mM)}
KINETIC scheme3 {
  ~ ko <-> kbath (r, r)
}
```

are translated to the single ODE equivalent

```
ko' = r*(kbath - ko)
```

i.e. ko tends exponentially to the steady state value of kbath.

Equation definition blocks

The BREAKPOINT block

The recommended idiom for integrating a kinetic scheme is

```
BREAKPOINT {
  SOLVE scheme METHOD sparse
  . . .
}
```

which integrates the STATEs in the scheme one dt step per call to fadvance(). The sparse method is generally faster than computing the full Jacobian matrix, though both use Newton iterations to advance the STATEs with a fully implicit method (first order correct). Additionally, the sparse method separates the Jacobian evaluation from the calculation of the STATE derivatives, thus allowing adaptive integration methods, such as CVODE, to efficiently compute only what is needed to advance the STATEs. Nonimplicit methods, such as Runge-Kutta or forward Euler, should be avoided since kinetic schemes commonly have very wide ranging rate constants that make these methods numerically unstable with reasonable dt steps. In fact, it is not unusual to specify equilibrium reactions such as

```
~ A <-> B (1e6*sqrt(K), 1e6/sqrt(K))
```

which can only be solved by implicit methods.

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

The `INITIAL` block

Initialization of a kinetic scheme to its steady state is accomplished with

```
INITIAL {
   SOLVE scheme STEADYSTATE sparse
}
```

Appropriate `CONSERVE` statements should be part of the scheme (see the following discussion of the `KINETIC` block) so that the equivalent system of ODEs is linearly independent. It should be kept in mind that source fluxes (constant for infinite time) have a strong effect on the steady state. Finally, it is crucial to test the scheme in NEURON under conditions in which the correct behavior is known.

The `KINETIC` block

The voltage-dependent rate constants are computed in procedure `rates()`. That procedure computes the equilibrium constants `K1` and `K2` from the constants `k1`, `d1`, `k2`, and `d2`, whose empirically determined default values are given in the `PARAMETER` block, and membrane potential `v`. The time constants `tau1` and `tau2`, however, are found from tables created under `hoc` (see **The `FUNCTION_TABLES`** below).

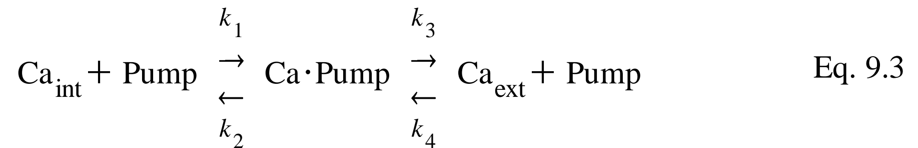
The other noteworthy item in this block is the `CONSERVE` statement. As mentioned above in **General comments about kinetic schemes**, the basic idea is to systematically account for conservation of material. If there is neither a source nor a sink reaction for a `STATE`, the differential equations are not linearly independent when steady states are calculated (`dt` approaches infinity). For example, in `scheme1` above the steady state condition `m'` = `mc'` = 0 yields two identical equations. Steady states can be approximated by integrating for several steps from any initial condition with large `dt`, but roundoff error can be a problem if the Jacobian matrix is nearly singular. To help solve the equations while maintaining strict numerical conservation throughout the simulation (no accumulation of roundoff error), the user is allowed to explicitly specify conservation equations with the `CONSERVE` statement. The conservation law for `scheme1` is specified in NMODL by

```
CONSERVE m + mc = 1
```

The `CONSERVE` statement does not add to the information content of a kinetic scheme and should be considered only as a hint to the translator. The NMODL translator uses this algebraic equation to replace the ODE for the last `STATE` on the left side of the equal sign. If one of the `STATE` names is an array, the conservation equation will contain an implicit sum over the array. If the last `STATE` is an array, then the ODE for the last `STATE` array element will be replaced by the algebraic equation. The choice of which `STATE` ODE is replaced by the algebraic equation depends on the implementation, and does not affect the solution (to within roundoff error). If a `CONSERVE`d `STATE` is relative to a compartment size, then compartment size is implicitly taken into account for the `STATE`s on the left hand side of the `CONSERVE` equation (see **Example 9.8: calcium diffusion with buffering** for discussion of the `COMPARTMENT` statement). The right hand side is merely an expression, in which any necessary compartment sizes must be included explicitly.

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

Thus in a calcium pump model



$$\mathrm{Ca_{int}} + \mathrm{Pump} \underset{k_2}{\overset{k_1}{\rightleftarrows}} \mathrm{Ca \cdot Pump} \underset{k_4}{\overset{k_3}{\rightleftarrows}} \mathrm{Ca_{ext}} + \mathrm{Pump} \quad \text{Eq. 9.3}$$

the pump is conserved and one could write

```
CONSERVE pump + pumpca = total_pump * pumparea
```

The FUNCTION_TABLES

As noted above, the steady state clamp data define the voltage dependence of K_1 and K_2, but a complete description of the K^+ current requires analysis of the temporal properties of the clamp current to determine the rate factors at each of the command potentials. The result would be a list or table of membrane potentials with associated time constants. One way to handle these numeric values would be to fit them with a pair of approximating functions, but the tactic used in this example is to leave them in tabular form for NMODL's FUNCTION_TABLE to deal with.

This is done by placing the numeric values in three Vectors, say v_vec, tau1_vec, and tau2_vec, where the first is the list of voltages and the other two are the corresponding time constants. These Vectors would be attached to the FUNCTION_TABLEs of this model with the hoc commands

```
table_tau1_k3st(tau1_vec, v_vec)
table_tau2_k3st(tau2_vec, v_vec)
```

Then whenever tau1(x) is called in the NMODL file, or tau1_k3st(x) is called from hoc, the returned value is interpolated from the array.

A useful feature of FUNCTION_TABLEs is that, prior to developing the Vector database, they can be attached to a scalar value, as in

```
table_tau1_k3st(100)
```

effectively becoming constant functions. Also FUNCTION_TABLEs can be declared with two arguments and attached to doubly dimensioned hoc arrays. In this case the table is linearly interpolated in both dimensions. This is useful with rates that depend on both voltage and calcium.

Usage

Inserting this mechanism into a section makes the STATEs c1_k3st, c2_k3st, and o_k3st available at the hoc level, as well as the conductances gbar_k3st and g_k3st.

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

Example 9.8: calcium diffusion with buffering

This mechanism illustrates how to use kinetic schemes to model intracellular Ca^{2+} diffusion and buffering. It differs from the prior example in several important aspects: Ca^{2+} is not conserved but instead enters as a consequence of the transmembrane Ca^{2+} current; diffusion involves the exchange of Ca^{2+} between compartments of unequal size; Ca^{2+} is buffered.

Only free Ca^{2+} is assumed to be mobile, whereas bound Ca^{2+} and free buffer are stationary. Buffer concentration and rate constants are based on the bullfrog sympathetic ganglion cell model described by Yamada et al. [Yamada, 1998 #580]. For a thorough treatment of numeric solution of the diffusion equations the reader is referred to Oran and Boris [Oran, 1987 #93].

Modeling diffusion with kinetic schemes

Diffusion is modeled as the exchange of Ca^{2+} between adjacent compartments. We begin by examining radial diffusion, and defer consideration of longidudinal diffusion to **Equation definition blocks: The `KINETIC` block** later in this example.

For radial diffusion, the compartments are a series of concentric shells around a cylindrical core, as shown in Fig. 9.9 for `Nannuli` = 4. The index of the outermost shell is 0 and the index of the core is `Nannuli` - 1. The outermost shell is half as thick as the others so that $[Ca^{2+}]$ will be second order correct with respect to space at the surface of the segment. Concentration is also second order correct midway through the thickness of the other shells and at the center of the core. These depths are indicated by "x" in Fig. 9.9. The radius of the cylindrical core equals the thickness of the outermost shell, and the intervening `Nannuli` - 2 shells each have thickness Δr = `diam` / 2 (`Nannuli` - 1), where `diam` is the diameter of the segment.

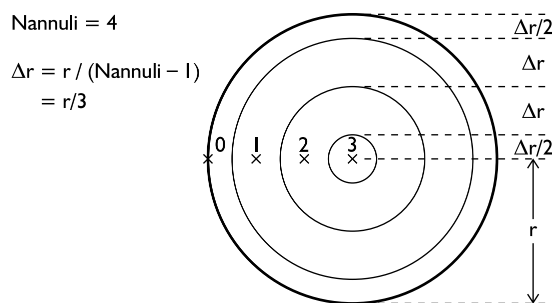


Figure 9.9. Diagram of the concentric shells used to model radial diffusion. The × mark the radial distances at which concentration will be second order correct in space.

Because segment diameter and the number of shells affect the dimensions of the shells, they also affect the time course of diffusion. The flux between adjacent shells is $\Delta[Ca^{2+}]\ D_{Ca}\ A / \Delta r$, where $\Delta[Ca^{2+}]$ is the concentration difference between the shell

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

centers, D_{Ca} is the diffusion coefficient for Ca^{2+}, A is the area of the boundary between shells, and Δr is the distance between their centers. This suggests that diffusion can be described by the basic kinetic scheme

```
FROM i = 0 TO Nannuli-2 {
  ~ ca[i] <-> ca[i+1] (f[i+1], f[i+1])
}
```

where `Nannuli` is the number of shells, `ca[i]` is the concentration midway through the thickness of shell `i` (except for `ca[0]` which is the concentration at the outer surface of shell 0), and the rate constants `f[i+1]` equal $D_{Ca} A_{i+1} / \Delta r$. For each adjacent pair of shells, both A_{i+1} and Δr are directly proportional to segment diameter. Therefore the ratios $A_{i+1} / \Delta r$ depend only on shell index, i.e. once they have been computed for one segment, they can be used for all segments that have the same number of radial compartments regardless of segment diameter.

As it stands, this kinetic scheme is dimensionally incorrect. Dimensional consistency requires that the product of `STATE`s and rates be in units of `STATE` per time (also see **Compartment size** in the section on **Chemical reactions** in **Chapter 3**). In the present example the `STATE`s `ca[]` are intensive variables (concentration, or mass/volume), so the product of `f[]` and `ca[]` must be in units of concentration/time. However, the rates `f[]` have units of volume/time, so this product is actually in units of mass/time, i.e. it is a flux that signifies the rate at which Ca^{2+} is entering or leaving a compartment. This is the time derivative of an extensive variable (i.e. of a variable that describes amount of material).

This disparity is corrected by specifying `STATE` volumes with the `COMPARTMENT` statement, as in

```
COMPARTMENT volume {state1 state2 . . . }
```

where the `STATE`s named in the braces have the same compartment volume given by the *`volume`* expression after the `COMPARTMENT` keyword. The volume merely multiplies the *d*`STATE`*/dt* left hand side of the equivalent differential equations, converting it to an extensive quantity and making it consistent with flux terms in units of absolute quantity per time.

The volume of each cylindrical shell depends on its index and the total number of shells, and is proportional to the square of segment diameter. Consequently the volumes can be computed once for a segment with unit diameter and then scaled by `diam^2` for use in each segment that has the same `Nannuli`. The equations that describe the radial movement of Ca^{2+} are independent of segment length. Therefore it is convenient to express shell volumes and surface areas in units of μm^2 (volume/length) and μm (area/length), respectively.

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

```
: Calcium ion accumulation with radial and longitudinal diffusion

NEURON {
  SUFFIX cadifus
  USEION ca READ cai, ica WRITE cai
  GLOBAL vrat, TotalBuffer  : vrat must be GLOBAL--see INITIAL block
                            : however TotalBuffer may be RANGE
}

DEFINE Nannuli 4  : must be >=2 (i.e. at least shell and core)

UNITS {
  (molar) = (1/liter)
  (mM)    = (millimolar)
  (um)    = (micron)
  (mA)    = (milliamp)
  FARADAY = (faraday)  (10000 coulomb)
  PI      = (pi)       (1)
}

PARAMETER {
  DCa   = 0.6 (um2/ms)
  k1buf = 100 (/mM-ms) : Yamada et al. 1989
  k2buf = 0.1 (/ms)
  TotalBuffer = 0.003  (mM)
}

ASSIGNED {
  diam      (um)
  ica       (mA/cm2)
  cai       (mM)
  vrat[Nannuli]  (1) : dimensionless
              : vrat[i] is volume of annulus i of a 1um diameter cylinder
              : multiply by diam^2 to get volume per um length
  Kd        (/mM)
  B0        (mM)
}

STATE {
  : ca[0] is equivalent to cai
  : ca[] are very small, so specify absolute tolerance
  ca[Nannuli]       (mM) <1e-10>
  CaBuffer[Nannuli] (mM)
  Buffer[Nannuli]   (mM)
}

BREAKPOINT { SOLVE state METHOD sparse }
```

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

```
LOCAL factors_done

INITIAL {
   if (factors_done == 0) {  : flag becomes 1 in the first segment
      factors_done = 1       :   all subsequent segments will have
      factors()              :   vrat = 0 unless vrat is GLOBAL
   }

  Kd = k1buf/k2buf
  B0 = TotalBuffer/(1 + Kd*cai)

  FROM i=0 TO Nannuli-1 {
    ca[i] = cai
    Buffer[i] = B0
    CaBuffer[i] = TotalBuffer - B0
  }
}

LOCAL frat[Nannuli]  : scales the rate constants for model geometry

PROCEDURE factors() {
  LOCAL r, dr2
  r = 1/2                 : starts at edge (half diam)
  dr2 = r/(Nannuli-1)/2   : full thickness of outermost annulus,
                          : half thickness of all other annuli
  vrat[0] = 0
  frat[0] = 2*r
  FROM i=0 TO Nannuli-2 {
    vrat[i] = vrat[i] + PI*(r-dr2/2)*2*dr2  : interior half
    r = r - dr2
    frat[i+1] = 2*PI*r/(2*dr2)  : outer radius of annulus
                                : div by distance between centers
    r = r - dr2
    vrat[i+1] = PI*(r+dr2/2)*2*dr2  : outer half of annulus
  }
}

LOCAL dsq, dsqvol  : can't define local variable in KINETIC block
                   :   or use in COMPARTMENT statement

KINETIC state {
  COMPARTMENT i, diam*diam*vrat[i] {ca CaBuffer Buffer}
  LONGITUDINAL_DIFFUSION i, DCa*diam*diam*vrat[i] {ca}
  ~ ca[0] << (-ica*PI*diam/(2*FARADAY))  : ica is Ca efflux
  FROM i=0 TO Nannuli-2 {
    ~ ca[i] <-> ca[i+1]  (DCa*frat[i+1], DCa*frat[i+1])
  }
  dsq = diam*diam
  FROM i=0 TO Nannuli-1 {
    dsqvol = dsq*vrat[i]
    ~ ca[i] + Buffer[i] <-> CaBuffer[i]  (k1buf*dsqvol, k2buf*dsqvol)
  }
  cai = ca[0]
}
```

Listing 9.8. `cadif.mod`

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

The `NEURON` block

This model `READ`s `cai` to initialize both intracellular [Ca^{2+}] and the buffer (see **The `INITIAL` block** below), and it `WRITE`s `cai` because it computes [Ca^{2+}] in the outermost shell during a simulation run. It also `READ`s `ica`, which is the Ca^{2+} influx into the outermost shell.

There are two `GLOBAL`s. One is the total buffer concentration `TotalBuffer`, which is assumed to be uniform throughout the cell. The other is `vrat`, an array whose elements will be the numeric values of the (volume/length) of the shells for a segment with unit diameter. These values are computed by `PROCEDURE factors()` near the end of Listing 9.8. As noted above, a segment with diameter `diam` has shells with volume/length equal to `diam^2 * vrat[i]`. Because each instance of this mechanism has the same number of shells, the same `vrat[i]` can be used to find the shell volumes at each location in the model cell where the mechanism exists.

The `DEFINE` statement sets the number of shells to 4. Many of the variables in this model are arrays, and NMODL arrays are not dynamic. Instead, their size must be specified when the NMODL code is translated to C.

The `UNITS` block

Faraday's constant is rescaled here so we won't have to include a separate units conversion factor in the statement in the `KINETIC` block where transmembrane current `ica` is reckoned as the efflux of Ca^{2+} from the outermost shell. Each statement in a `UNITS` block must explicitly assert the units that are involved, so the statement that assigns the value 3.141 . . . to `PI` includes a `(1)` to mark it as a dimensionless constant.

Variable declaration blocks

The `ASSIGNED` block

The variable `vrat` is declared to be an array with `Nannuli` elements. As with C, array indices run from 0 to `Nannuli` - 1. The variables `Kd` and `B0` are the dissociation constant for the buffer and the initial value of free buffer, which are computed in the `INITIAL` block (see below). Both the total buffer and the initial concentration of Ca^{2+} are assumed to be uniform throughout all shells, so a scalar is used for `B0`.

The `STATE` block

In addition to diffusion, this mechanism involves Ca^{2+} buffering

$$\mathrm{Ca} + \mathrm{Buffer} \underset{k_2^{buf}}{\overset{k_1^{buf}}{\rightleftarrows}} \mathrm{Ca \cdot Buffer} \qquad \text{Eq. 9.4}$$

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

This happens in each of the shells, so `ca`, `Buffer` and `CaBuffer` are all arrays.

The declaration of `ca[]` uses the syntax *`state (units) <absolute_tolerance>`* to specify the local absolute error tolerance that will be employed by CVODE. The solver tries to use a step size for which the local error ϵ_i of each $state_i$ satisfies at least one of these two inequalities:

$$\epsilon_i < relative_tolerance \cdot |state_i|$$

or

$$\epsilon_i < absolute_tolerance$$

The default values for these tolerances are 0 and 10^{-3}, respectively, so only a `STATE` that is extremely small (such as intracellular [Ca^{2+}]) needs to have its absolute tolerance specified. As an alternative to specifying a smaller absolute tolerance, `ca[]` could have been defined in terms of units such as micromolar or nanomolar, which would have increased the numeric value of these variables. This would necessitate different units conversion factors in many of the statements that involve `ca[]`. For example, the assignment for `cai` (which is required to be in mM) would be `cai = (1e-6)*ca[0]`.

`LOCAL` variables declared outside of equation definition blocks

A `LOCAL` variable that is declared outside of an equation definition block is equivalent to a static variable in C. That is, it is visible throughout the mechanism (but not at the `hoc` level), it retains its value, and it is shared between all instances of a given mechanism. The initial value of such a variable is 0.

This particular mechanism employs four variables of this type: `factors_done`, `frat[]`, `dsq`, and `dsqvol`. The meaning of each of these is discussed below.

Equation definition blocks

The `INITIAL` block

Initialization of this mechanism is a two step process. The first step is to use `PROCEDURE factors()` (see below) to set up the geometry of the model by computing `vrat[]` and `frat[]`, the arrays of units conversion factors that are applied to the shell volumes and rate constants. This only has to be done once because the same conversion factors are used for all segments that have the same number of shells, as noted above in **Modeling diffusion with kinetic schemes**. The variable `factors_done` is a flag that indicates whether `vrat[]` and `frat[]` have been computed. The NMODL keyword `LOCAL` means that the value of `factors_done` will be the same in all instances of this mechanism, but that it will not be visible at the `hoc` level. Therefore `factors()` will be executed only once, regardless of how many segments contain the `cadifus` mechanism.

The second step is to initialize the mechanism's `STATE`s. This mechanism assumes that the total buffer concentration and the initial free calcium concentration are uniform

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

in all shells, and that buffering has reached its steady state. Therefore the initial concentration of free buffer is computed from the initial [Ca^{2+}] and the buffer's dissociation constant. It should be noted that the value of `cai` will be set to `cai0_ca_ion` just prior to executing the code in the `INITIAL` block (see **Ion concentrations and equilibrium potentials** in **Chapter 8**).

It may be instructive to compare this initialization strategy with the approach that was used for the voltage-gated current of Listing 9.7 (`k3st.mod`). That previous example initialized the `STATE` through numeric solution of a kinetic scheme, so its `KINETIC` block required a `CONSERVE` statement to ensure that the equivalent system of ODEs would be linearly independent. Here, however, the `STATE`s are initialized by explicit algebraic assignment, so no `CONSERVE` statement is necessary.

`PROCEDURE factors()`

The arrays `vrat[]` and `frat[]`, which are used to scale the shell volumes and rate constants to ensure consistency of units, are computed here. Their values depend only on the number of shells, so they do not have to be recomputed if `diam` or `DFree` is changed.

The elements of `vrat[]` are the volumes of a set of concentric cylindrical shells, whose total volume equals the volume of a cylinder with diameter and length of 1 µm. These values are computed in two stages by the `FROM i=0 TO Nannuli-2 { }` loop. The first stage finds the volume of the outer half and the second finds the volume of the inner half of the shell.

The `frat` array is declared to be `LOCAL` because it applies to all segments that have the `cadifus` mechanism, but it is unlikely to be of interest to the user and therefore does not need to be visible at the `hoc` level. This contrasts with `vrat`, which is declared as `GLOBAL` within the `NEURON` block so that the user can see its values. The values `frat[i+1]` equal $A_{i+1} / \Delta r$, where A_{i+1} is the surface area between shells `i` and `i+1` for 0 ≤ `i` < `Nannuli`, and Δr is the distance between shell centers (radius / (`Nannuli` - 1)).

The `KINETIC` block

The first statement in this block specifies the shell volumes for the `STATE`s `ca`, `CaBuffer`, and `Buffer`. As noted above in **Modeling diffusion with kinetic schemes**, these volumes equal the elements of `vrat[]` multiplied by the square of the segment diameter. This mechanism involves many compartments whose relative volumes are specified by the elements of an array, so we can deal with all compartments with a single statement of the form

```
COMPARTMENT index, volume[index] { state1 state2 . . . }
```

where the diffusing `STATE`s are listed inside the braces.

Next in this block is a `LONGITUDINAL_DIFFUSION` statement, which specifies that this mechanism includes "nonlocal" diffusion, i.e. longitudinal diffusion along a section and into connecting sections. The syntax for scalar `STATE`s is

```
LONGITUDINAL_DIFFUSION flux_expr { state1 state2 . . . }
```

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

where *flux_expr* is the product of the diffusion constant and the area of the cross section between adjacent compartments. Units of the *flux_expr* must be (micron^4/ms), i.e. the diffusion constant has units of (micron^2/ms) and the cross sectional area has units of (micron^2). For cylindrical shell compartments, the cross sectional area is just the volume per unit length. If the states are arrays then all elements are assumed to diffuse between corresponding volumes in adjacent segments and the iteration variable must be specified as in

```
LONGITUDINAL_DIFFUSION index, flux_expr(index) { state1 state2 . . . }
```

A COMPARTMENT statement is also required for the diffusing STATEs and the units must be (micron^2), i.e. (micron^3/micron).

The compactness of LONGITUDINAL_DIFFUSION specification contrasts nicely with the great deal of trouble imposed on the computational methods used to solve the equations. The standard backward Euler method, historically the default method used by NEURON (see **Chapter 4**), can no longer find steady states with extremely large (e.g. 10^9 ms) steps since not every Jacobian element for both flux and current with respect to voltage and concentration is presently accurately computed. CVODE works well for these problems since it does not allow dt to grow beyond the point of numerical instability. Despite these occasional limitations on numerical efficiency, it is satisfying that, as methods evolve to handle these problems more robustly, the specification of the models does not change.

The third statement in this block is equivalent to a differential equation that describes the contribution of transmembrane calcium current to Ca^{2+} in the outermost shell. The << signifies an explicit flux. Because of the COMPARTMENT statement, the left hand side of the differential equation is not $d[Ca^{2+}]_0/dt$ but d(*total* Ca^{2+} in the outermost shell)$/dt$. This is consistent with the right hand side of the equation, which is in units of mass per time.

Next is the kinetic scheme for radial diffusion. The rate constants in this scheme equal the product of DCa and the factor frat[] for reasons that were explained earlier in **Modeling diffusion with kinetic schemes**.

It may not be immediately clear why the rate constants in the kinetic scheme for Ca^{2+} buffering are scaled by the compartment volume dsqvol; however, the reason will become obvious when one recalls that the COMPARTMENT statement at the beginning of the KINETIC block has converted the units of the *d*STATE*/dt* on the left hand side of the equivalent differential equations from concentration per time to mass per time. If the reaction rate constants were left unchanged, the right hand side of the differential equations for buffering would have units of concentration per time, which is inconsistent. Multiplying the rate constants by compartment volume removes this inconsistency by changing the units of the right hand side to mass per time.

The last statement in the KINETIC block updates the value of cai from ca[0]. This is necessary because intracellular $[Ca^{2+}]$ is known elsewhere in NEURON as cai, e.g. to other mechanisms and to NEURON's internal routine that computes E_{Ca}.

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

When developing a new mechanism or making substantive changes to an existing mechanism, it is generally advisable to check for consistency of units with `modlunit`. Given the dimensional complexity of this model, such testing is absolutely indispensable.

Usage

If this mechanism is inserted in a section, the concentrations of Ca^{2+} and the free and bound buffer in all compartments will be available through `hoc` as `ca_cadifus[]`, `Buffer_cadifus[]`, and `CaBuffer_cadifus[]`. These `STATE`s will also be available for plotting and analysis through the GUI.

The `PARAMETER`s `DCa`, `k1buf`, `k2buf`, and `TotalBuffer` will also be available for inspection and modification through both the graphical interface and `hoc` statements (with the `_cadifus` suffix). All `PARAMETER`s are `GLOBAL`s by default, i.e. they will have the same values in each location where the `cadifus` mechanism has been inserted. Therefore in a sense it is gratuitous to declare in the `NEURON` block that `TotalBuffer` is `GLOBAL`. However, this declaration serves to remind the user of the nature of this important variable, which is likely to be changed during exploratory simulations or optimization.

In some cases it might be useful for one or more of the `PARAMETER`s to be `RANGE` variables. For example, `TotalBuffer` and even `DCa` or the buffer rate constants might not be uniform throughout the cell. To make `TotalBuffer` and `DCa` `RANGE` variables only requires replacing the line

```
GLOBAL vrat, TotalBuffer
```

in the `NEURON` block with

```
GLOBAL vrat
RANGE TotalBuffer, DCa
```

The `GLOBAL` volume factors `vrat[]` are available through `hoc` for inspection, but it is inadvisable to change their values because they would likely be inconsistent with the `frat[]` values and thereby cause errors in the simulation.

All occurrences of this mechanism will have the same number of shells, regardless of the physical diameter of the segments in which the mechanism has been inserted. With `Nannuli` = 4, the thickness of the outermost shell will be ≤ 1 µm in segments with `diam` ≤ 6 µm. If this spatial resolution is inadequate, or if the model has segments with larger diameters, then `Nannuli` may have to be increased. NMODL does not have dynamic arrays, so in order to change the number of shells one must recompile the mechanism after assigning a new value to `Nannuli` by editing the NMODL source code.

Example 9.9: a calcium pump

This mechanism involves a calcium pump based on the reaction scheme outlined in the description of the `KINETIC` block of **Example 9.7: kinetic scheme for a voltage-gated current**. It is a direct extension of the model of calcium diffusion with buffering in

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

Example 9.8: calcium diffusion with buffering, the principal difference being that a calcium pump is present in the cell membrane. The following discussion focuses on the requisite changes in Listing 9.8, and the operation and use of this new mechanism. For all other details the reader should refer to **Example 9.8**.

The `NEURON` block

Changes in the `NEURON` block are marked in **bold**. The first nontrivial difference from the prior example is that this mechanism `READ`s the value of `cao`, which is used in the pump reaction scheme.

```
NEURON {
   SUFFIX cdp
   USEION ca READ cao, cai, ica WRITE cai, ica
   RANGE ica_pmp
   GLOBAL vrat, TotalBuffer, TotalPump
}
```

The mechanism also `WRITE`s a pump current that is attributed to `ica` so that its transmembrane Ca^{2+} flux will be factored into NEURON's calculations of $[Ca^{2+}]_i$. This current, which is a `RANGE` variable known as `ica_pmp_cdp` to the `hoc` interpreter, constitutes a net movement of positive charge across the cell membrane, and it follows the usual sign convention (outward current is "positive"). The pump current has a direct effect on membrane potential, which, because of the rapid activation of the pump, is manifest by a distinct delay of the spike peak and a slight increase of the postspike hyperpolarization. This mechanism could be made electrically "silent" by having it `WRITE` an equal but opposite `NONSPECIFIC_CURRENT` or perhaps a current that involves some other ionic species, e.g. Na^+, K^+, or Cl^-.

The variable `TotalPump` is the total density of pump sites on the cell membrane, whether free or occupied by Ca^{2+}. Making it `GLOBAL` means that it is user adjustable, and that the pump is assumed to have uniform density wherever the mechanism has been inserted. If local variation is required, this should be a `RANGE` variable.

The `UNITS` block

This mechanism includes the statement `(mol) = (1)` because the density of pump sites will be specified in units of `(mol/cm2)`. The term `mole` cannot be used here because it is already defined in NEURON's units database as $6.022169 \cdot 10^{23}$ (Avogadro's number).

Variable declaration blocks

The `PARAMETER` block

Five new statements have been added because this mechanism uses the rate constants of the pump reactions and the density of pump sites on the cell membrane.

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

```
k1 = 1       (/mM-ms)
k2 = 0.005   (/ms)
k3 = 1       (/ms)
k4 = 0.005   (/mM-ms)
: to eliminate pump, set TotalPump to 0 in hoc
TotalPump = 1e-11  (mol/cm2)
```

These particular rate constant values were chosen to satisfy two criteria: the pump influx and efflux should be equal at $[Ca^{2+}]$ = 50 nM, and the rate of transport should be slow enough to allow a slight delay in accelerated transport following an action potential that included a voltage-gated Ca^{2+} current. The density `TotalPump` is sufficient for the pump to have a marked damping effect on $[Ca^{2+}]_i$ transients; lower values reduce the ability of the pump to regulate $[Ca^{2+}]_i$.

The `ASSIGNED` block

These three additions have been made.

```
cao      (mM)
ica_pmp  (mA/cm2)
parea    (um)
```

This mechanism treats $[Ca^{2+}]_o$ as a constant. The pump current and the surface area over which the pump is distributed are also clearly necessary.

The `CONSTANT` block

Consistency of units requires explicit mention of an extracellular volume in the kinetic scheme for the pump.

```
CONSTANT { volo = 1e10  (um2) }
```

The value used here is equivalent to 1 liter of extracellular space per micron length of the cell, but the actual value is irrelevant to this mechanism because `cao` is treated as a constant. Since the value of `volo` is not important for this mechanism, there is no need for it to be accessible through `hoc` commands or the GUI, so it is not a `PARAMETER`. On the other hand, there is a sense in which it is an integral part of the pump mechanism, so it would not be appropriate to make `volo` be a `LOCAL` variable since `LOCAL`s are intended for temporary storage of "throwaway" values. Finally, the value of `volo` would never be changed in the course of a simulation. Therefore `volo` is declared in a `CONSTANT` block.

The `STATE` block

The densities of pump sites that are free or have bound Ca^{2+}, respectively, are represented by the two new `STATE`s

```
pump    (mol/cm2)
pumpca  (mol/cm2)
```

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

Equation definition blocks

The `BREAKPOINT` block

This block has one additional statement

```
BREAKPOINT {
  SOLVE state METHOD sparse
  ica = ica_pmp
}
```

The assignment `ica = ica_pmp` is needed to ensure that the pump current is reckoned in NEURON's calculation of $[Ca^{2+}]_i$.

The `INITIAL` block

The statement

```
parea = PI*diam
```

must be included to specify the area per unit length over which the pump is distributed.

If it is correct to assume that $[Ca^{2+}]_i$ has been equal to `cai0_ca_ion` (default = 50 nM) for a long time, the initial levels of `pump` and `pumpca` can be set by using the steady state formula

```
pump = TotalPump/(1 + (cai*k1/k2))
pumpca = TotalPump - pump
```

An alternative initialization strategy is to place

```
ica = 0
SOLVE state STEADYSTATE sparse
```

at the end of the `INITIAL` block, where the `ica = 0` statement is needed because the kinetic scheme treats transmembrane Ca^{2+} currents as a source of Ca^{2+} flux. This idiom makes NEURON compute the initial values of `STATE`s, which can be particularly convenient for mechanisms whose steady state solutions are difficult or impossible to express in analytical form. This would require adding a `CONSERVE` statement to the `KINETIC` block to insure that the equations that describe the free and bound buffer are independent (see also **The `INITIAL` block** in **Example 9.7: kinetic scheme for a voltage-gated current**).

Both of these initializations explicitly assume that the net Ca^{2+} current generated by other sources equals 0, so the net pump current following initialization is also 0. If this assumption is incorrect, as is almost certainly the case if one or more voltage-gated Ca^{2+} currents are included in the model, then $[Ca^{2+}]_i$ will start to change immediately when a simulation is started. Most often this is not the desired outcome. The proper initialization of a model that contains mechanisms with complex interactions may involve performing an "initialization run" and using `SaveState` objects (see **Examples of custom initializations** in **Chapter 8**).

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

The `KINETIC` block

Changes in this block are marked in **bold**. For dimensional consistency, the pump scheme requires the new `COMPARTMENT` statements and units conversion factor `(1e10)`.

```
KINETIC state {
  COMPARTMENT i, diam*diam*vrat[i] {ca CaBuffer Buffer}
  COMPARTMENT (1e10)*parea {pump pumpca}
  COMPARTMENT volo {cao}
  LONGITUDINAL_DIFFUSION i, DCa*diam*diam*vrat[i] {ca}

  :pump
  ~ ca[0] + pump <-> pumpca  (k1*parea*(1e10), k2*parea*(1e10))
  ~ pumpca <-> pump + cao    (k3*parea*(1e10), k4*parea*(1e10))
  CONSERVE pump + pumpca = TotalPump * parea * (1e10)
  ica_pmp = 2*FARADAY*(f_flux - b_flux)/parea

  : all currents except pump
  ~ ca[0] << (-(ica - ica_pmp)*PI*diam/(2*FARADAY))
  FROM i=0 TO Nannuli-2 {
    ~ ca[i] <-> ca[i+1] (DCa*frat[i+1], DCa*frat[i+1])
  }
  dsq = diam*diam
  FROM i=0 TO Nannuli-1 {
    dsqvol = dsq*vrat[i]
    ~ ca[i] + Buffer[i] <-> CaBuffer[i] (k1buf*dsqvol, k2buf*dsqvol)
  }

  cai = ca[0]
}
```

The pump reaction statements implement the scheme outlined in the description of the `KINETIC` block of **Example 9.7: kinetic scheme for a voltage-gated current**. Also as described in that section, the `CONSERVE` statement ensures strict numerical conservation, which is helpful for convergence and accuracy.

In the steady state, the net forward flux in the first and second reactions must be equal. Even during physiologically relevant transients, these fluxes track each other effectively instantaneously. Therefore the transmembrane Ca^{2+} flux generated by the pump is taken to be the net forward flux in the second reaction.

This mechanism `WRITE`s `ica` in order to affect $[Ca^{2+}]_i$. The total transmembrane Ca^{2+} flux is the sum of the pump flux and the flux from all other sources. Thus to make sure that `ica_pmp` is not counted twice, it is subtracted from total Ca^{2+} current `ica` in the expression that relates Ca^{2+} current to Ca^{2+} flux.

Usage

The `STATE`s and `PARAMETER`s that are available through `hoc` and the GUI are directly analogous to those of the `cadifus` mechanism, but they will have the suffix `_cdp` rather than `_cadifus`. The additional pump variables `pump_cdp`, `pumpca_cdp`, `ica_pmp_cdp`, and `TotalPump_cdp` will also be available and are subject to similar

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

concerns and constraints as their counterparts in the diffusion reactions (see **Usage** in **Example 9.8: calcium diffusion with buffering**).

Models with discontinuities

The incorporation of variable time step integration methods in NEURON made it necessary to provide a way to ensure proper handling of abrupt changes in `PARAMETER`s, `ASSIGNED` variables, and `STATE`s. At first this was accomplished by adding `at_time()` and `state_discontinuity()` to NMODL, but the advent of NEURON's event delivery system has obviated the need for these functions and we strongly advise against using them in any new model development. Even so, they have been employed in several mechanisms of recent vintage, e.g. models of pulse generators and synaptic transmission, so the following discussion contains explanations of why they were used and what they do, as well as current recommendations for preferred ways to implement models that involve discontinuities.

Discontinuities in `PARAMETER`s and `ASSIGNED` variables

Before CVODE was added to NEURON, sudden changes in `PARAMETER`s and `ASSIGNED` variables, such as the sudden change in current injection during a current pulse, had been implicitly assumed to take place on a time step boundary. This is inadequate with variable time step methods because it is unlikely that a time step boundary will correspond to the onset or offset of the pulse. Worse, the time step may be longer than the pulse itself, which may thus be entirely ignored.

The `at_time()` function was added so that a model description could explicitly notify NEURON of the times at which any discontinuities occur. This function has no effect on fixed time step integration, but during variable time step integration, the statement `at_time(tevent)` guarantees that the integrator reduces the step size so that it completes at time $\mathtt{tevent}_-$, which is on the order of roundoff error *before* `tevent`. The integrator then reinitializes at $\mathtt{tevent}_+$, which is on the order of roundoff error *after* `tevent`, and the solution continues from there. This is how the built-in current clamp model `IClamp` notifies NEURON of the time of onset of the pulse and its offset (see the `BREAKPOINT` block of **Example 9.3: an intracellular stimulating electrode**). As noted above, however, now the preferred way to implement abrupt changes in `PARAMETER`s and `ASSIGNED` variables is to take advantage of NEURON's event delivery system (specifically, self-events) because of improved computational efficiency and greater conceptual clarity (see **Chapter 10**).

In the course of a variable time step simulation, a missing `at_time()` call may cause one of two symptoms. If a `PARAMETER` changes but returns to its original value within the same interval, the pulse may be entirely missed. More often, a single discontinuity will take place within a time step interval, causing the integrator to start what seems like a binary search for the location of the discontinuity in order to satisfy the error tolerance on the step; of course, this is very inefficient.

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

Discontinuities in STATEs

Some kinds of synaptic models involve a discontinuity in one or more STATE variables. For example, a synapse whose conductance follows the time course of an alpha function (for more detail about the alpha function itself see Rall [Rall, 1977 #108] and Jack et al. [Jack, 1983 #90]) can be implemented as a kinetic scheme in the two state model

```
KINETIC state {
  ~ a <-> g (k, 0)
  ~ g -> (k)
}
```

("->" indicates a sink reaction), where a discrete synaptic event results in an abrupt increase of STATE a. This formulation has the attractive property that it can handle multiple streams of events with different weights, so that g will be the sum of the individual alpha functions with their appropriate onsets.

Abrupt changes in STATEs require particularly careful treatment because of the special nature of states in variable time step ODE solvers. Before the advent of an event delivery system in NEURON, this required not only an at_time() call to notify NEURON about the time of the discontinuity, but also a state_discontinuity() statement to specify how the affected STATE would change. Furthermore, state_discontinuity() could only be used in an if (at_time()){} block. Thus the BREAKPOINT block for a synaptic event that starts at onset and reaches a maximum conductance gmax would look like this

```
BREAKPOINT {
  if (at_time(onset)) {
    : scale factor exp(1) = 2.718... ensures
    : that peak conductance will be gmax
    state_discontinuity(a, a + gmax*exp(1))
  }
  SOLVE state METHOD sparse
  i = g*(v - e)
}
```

The first argument to state_discontinuity() is interpreted as a reference to the STATE, and the second argument is an expression for its new value. The first argument will be assigned the value of its second argument just *once* for any time step. This is important because, for several integration methods, BREAKPOINT assignment statements are often executed twice to calculate the *di/dv* terms of the Jacobian matrix.

This synaptic model works well with deterministic stimulus trains, but it is difficult for the user to supply the administrative hoc code for managing the onset and gmax variables to take advantage of the promise of "multiple streams of input events with different weights." The most important problem is how to save events that have significant delay between their generation and their handling at time onset. As this model stands, an event can be passed to it by assigning values to onset and gmax only after the previous onset event has been handled.

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

These complexities have been eliminated by the event delivery system. Instead of handling the state discontinuity in the BREAKPOINT block, the synaptic model should now be written in the form

```
BREAKPOINT {
  SOLVE state METHOD sparse
  i = g*(v - e)
}

NET_RECEIVE(weight (microsiemens)) {
  a = a + weight*exp(1)
}
```

in which event distribution is handled internally from a specification of network connectivity (see next section). Note that there is no need to use either `at_time()` or `state_discontinuity()`. Also, the BREAKPOINT block should not have any `if` statements. All discontinuities should be handled in a NET_RECEIVE block. For further details of how to deal with streams of synaptic events with arbitrary delays and weights, see **Chapter 10**.

> We should mention that early implementations of the event delivery system did require `state_discontinuity()`. Thus you may encounter a legacy synaptic model whose NET_RECEIVE block contains a statement such as `state_discontinuity(a, a+w*exp(1))`. This requirement no longer exists, and we discourage the use of this syntax.

Event handlers

With recent versions of NEURON, the most powerful and general strategy for dealing with discontinuities in ASSIGNED variables, PARAMETERs, and STATEs is to use the NetCon class's `event()` method, which exploits NEURON's event delivery system (see **Chapter 10**). The *handler*`()` procedure in *netcon*`.event(`*te*`, "`*handler*`()")` can contain statements that change anything discontinuously, as long as the last statement in *handler*`()` is `cvode.re_init()` (see **Chapter 8**).

Time-dependent PARAMETER changes

One way to translate the concept of a "smoothly varying" parameter into a computational implementation is by explicit specification in a model description, as in

```
BREAKPOINT { i = imax*sin(w*t) }
```

This works with both fixed and variable time step integration. Time-dependent changes can also be specified at the hoc interpreter level, but care is needed to ensure they are properly computed in the context of variable time steps. For instance, it might seem convenient to change PARAMETERs prior to `fadvance()` calls, e.g.

```
proc advance() {
  IClamp[0].amp = imax*sin(w*t)
  fadvance()
}
```

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

This does work with fixed `dt` but is discouraged because it produces inaccurate results with variable `dt` methods.

An alternative that works well with fixed and variable time step integration is to use the `Vector` class's `play()` method with linear interpolation, which became available in NEURON 5.4. This is invoked with

```
vec.play(&rangevar, tvec, 1)
```

in which *vec* and *tvec* are a pair of `Vector`s that define a piecewise linear function of time $y = f(t)$, i.e. *tvec* contains a monotonically increasing sequence of times, and *vec* holds the corresponding y values. The *rangevar* is the variable that is to be driven by $f()$. In the future, `Vector.play` will be extended to cubic spline interpolation and will allow "continuous" play of a smooth function defined by a `Vector`.

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

Chapter 9 Index

A

absolute error
    local
        tolerance 41
active transport
    electrically silent 45
    pump current 45-47
        countering with a NONSPECIFIC_CURRENT 45
        initialization 47
ASSIGNED block 5, 15, 23, 40
ASSIGNED variable 5
    GLOBAL
        spatial variation 23
        vs. RANGE 8, 22
    is a range variable by default 6
    v, celsius, t, dt, diam, and area 6
    visibility at the hoc level 6, 10
    when to use for an equilbrium potential 15

B

Backward Euler method
    and LONGITUDINAL_DIFFUSION 43
BREAKPOINT block 6, 15
    and computations that must be performed only once per time step 16
    and counts, flags, and random numbers 15
    and PROCEDUREs 16
    and rate functions 16
    and variables that depend on the number of executions 15
    currents assigned at end of 15
    SOLVE 15, 17
        cnexp 17
        derivimplicit 17

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

is not a function call 16
sparse 33
C
celsius 5, 23
channel
gating model
HH type 17
nonlinear 17
computational efficiency 1, 3, 17, 30, 33, 43, 49
conceptual clarity 1, 30, 49
CONSTANT
vs. PARAMETER or LOCAL variable 46
CONSTANT block 46
CVODE
and LONGITUDINAL_DIFFUSION 43
and model descriptions
at_time() 11, 49-51
state_discontinuity() 49-51
CVode class
re_init() 51
D
DERIVATIVE block 17, 23, 28
DERIVATIVE block
' (apostrophe) 17
diffusion
kinetic scheme 29
longitudinal 42
radial 36, 43
distributed mechanism 2, 13
Distributed Mechanism GUI
Manager
Inserter 7

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

dt
use in NMODL 3, 6
E
e
electronic charge vs. units conversion factor 22
equation
conservation 4
current balance 4
event
self-event 49
times
with adaptive integration 11, 51
Example 9.1: a passive "leak" current 2
Example 9.2: a localized shunt 7
Example 9.3: an intracellular stimulating electrode 10
Example 9.4: a voltage-gated current 12
Example 9.5: a calcium-activated, voltage-gated current 20
Example 9.6: extracellular potassium accumulation 25
Example 9.7: kinetic scheme for a voltage-gated current 31
Example 9.8: calcium diffusion with buffering 36
Example 9.9: a calcium pump 44
F
forward Euler method
stability 33
FUNCTION block 18, 24
G
GENESIS 1, 4
GMODL 4
H
hoc
calling an NMODL FUNCTION or PROCEDURE 18
specifying proper instance with setdata_ 18

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

I

INITIAL block 16

SOLVE

STEADYSTATE sparse 34, 47

initialization

categories 16

finitialize() 12, 16

strategies 16

steady state initialization of complex kinetic schemes 47

ion accumulation

initialization

of model geometry 41

ion mechanism

automatically created 26

initialization 27

J

Jacobian 30, 33

approximate 17, 43

computing di/dv elements 50

nearly singular 34

user-supplied 18

K

KINETIC block 34, 42, 48

-> (sink reaction indicator) 50

~ (tilde) 29

<-> (reaction indicator) 29

<< (explicit flux) 43

b_flux 29

COMPARTMENT 37, 42, 43

CONSERVE 34

when is it required for initialization? 34, 42, 47

f_flux 29

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

LONGITUDINAL_DIFFUSION 42
reactants 29
ASSIGNED or PARAMETER variables as 33
reaction rates 29
STATE-dependent, and instability 30
voltage-sensitive 30
reaction statement 29
L
LINEAR block 15
LOCAL variable
declared outside an equation block
initial value 41
scope and persistence 41
declared within an equation block
scope and persistence 19
M
Markov process
kinetic scheme 29
material
conservation 29, 33-35, 48
mod file 1
MODL 1
vs. NMODL 1, 4
modlunit 9, 22
N
National Biomedical Simulation Resource project 1
NET_RECEIVE block
handling abrupt changes and discontinuities 51
INITIAL block 16
state_discontinuity() 51
NetCon class
event() 51

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

NEURON block 4
- ELECTRODE_CURRENT 10
  - effect on extracellular mechanism 10
- GLOBAL 4
- NONSPECIFIC_CURRENT 4
  - equilibrium potential 15
- POINT_PROCESS 8
- RANGE 4, 10
- SUFFIX 4
- USEION 14, 21
  - READ ex (reading an equilibrium potential) 14
  - READ ix (reading an ionic current) 26, 40
  - READ xi (reading an intracellular concentration) 21, 40
  - READ xo (reading an extracellular concentration) 45
  - WRITE ix (writing an ionic current) 14, 21, 25, 45, 48
  - WRITE xi (writing an intracellular concentration) 40
  - WRITE xo (writing an extracellular concentration) 26

NMODL
- arrays
  - are not dynamic 40, 44
  - index starts at 0 40
- comments 3
- declaring variables 4
  - specifying units 5
- DEFINE 40
- FROM . . . TO . . . (loop statement) 42
- FUNCTION_TABLE 35
- named blocks 1
  - equation definition 1
  - general form 3
  - variable declaration 1
- translator 1, 3, 30, 34

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

translator
nmodl 4
nocmodl 4
nocmodl.exe 4
units conversion factor 9, 22, 28, 40, 41, 48
units conversion factor
parentheses 9
UNITSOFF . . . UNITSON 18
user-defined variable 3
VERBATIM . . . ENDVERBATIM 3
NONLINEAR block 15
nrnunits.lib 5, 22
numeric integration
adaptive 16, 17, 27, 33, 49, 51
explicit 33
fixed time step 17, 27, 51
fixed time step
event aggregation to time step boundaries 11, 49
implicit 33
order of accuracy
first 17, 33
second 17, 36
variable 17
P
PARAMETER block 5
assigning default PARAMETER values 5
specifying minimum and maximum limits 5
PARAMETER variable 5
GLOBAL vs. RANGE 5, 27, 45
is GLOBAL by default 44
RANGE 10
time-dependent 51

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

visibility at the hoc level 5
when to use for an equilbrium potential 15
Plot what? GUI 19, 24
point process 7, 10
Point Process Viewer GUI 10
PROCEDURE block 24
R
Runge-Kutta method
stability 33
S
SCoP 1, 6
standard run system
fadvance() 33, 51
STATE block 15
specifying local absolute error tolerance 41
state variable
of a mechanism vs. state variable of a model 6
STATE variable 15
array in NMODL 34
initialization 16
state0 28
ion concentration as 27
is automatically RANGE 15
vs. state variable 33
T
t
the independent variable in NEURON 6
use in NMODL 6
U
units
checking 5, 9, 44
consistency 37, 43

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

database 5
dimensionless
(1) 40
by default 5
e 22
faraday 22
k-mole 22
mole 45
specifying 18
UNITS block 14
defining new names 14
units scaling 22, 40
V
v
is a RANGE variable 4
variable
abrupt change of 49-51
extensive 37
intensive 37
Vector class
play()
with interpolation 52
X
x 45

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