Chapter 10

Synaptic transmission and artificial spiking cells

In NEURON, a cell model is a set of differential equations. Network models consist of cell models and the connections between them. Some forms of communication between cells, e.g. graded synapses and gap junctions, require more or less complete representations of the underlying biophysical mechanisms. In these cases, coupling between the cells is achieved by adding terms that refer to one cell's variables into equations that belong to a different cell. The first part of this chapter describes the `POINTER` syntax that makes this possible in NEURON.

The same approach can be used for detailed mechanistic models of spike triggered transmission, which entails spike initiation and propagation to the presynaptic terminal, transmitter release, ligand−receptor interactions on the postsynaptic cell, and somatodendritic integration. However, it is far more efficient to use the widespread practice of treating spike propagation from the trigger zone to the synapse as a delayed logical event. The second part of this chapter tells how NEURON's network connection class (`NetCon`) supports this event−based style of communication.

In the last part of this chapter, we use event−based communication to simplify representation of the neurons themselves, creating highly efficient models of artificial spiking neurons, e.g. integrate and fire cells. Artificial spiking cells are very convenient spike train sources for driving synaptic mechanisms attached to biophysical model cells. Networks that consist entirely of artificial spiking cells run hundreds of times faster than their biophysical counterparts. Thus they are particularly suitable for prototyping network models. They are also excellent tools in their own right for studying the functional consequences of network architectures and synaptic plasticity rules. In **Chapter 11** we demonstrate network models that involve various combinations of biophysical and artificial neural elements.

Modeling communication between cells

Experiments have demonstrated many kinds of interactions between neurons, but for most cells the principal avenues of communication are gap junctions and synapses. Gap junctions and synapses generate localized ionic currents, so in NEURON they are represented by point processes (see **Point processes** in **Chapter 5**, and examples **9.2: a localized shunt** and **9.3: an intracellular stimulating electrode** in **Chapter 9**).

The point processes used to represent gap junctions and synapses must produce a change at one location in the model that depends on information (membrane potential, calcium concentration, the occurrence of a spike) from some other location. This is in

sharp contrast to the examples we discussed in **Chapter 9**, all of which are "local" in the sense that an instance of a mechanism at a particular location on the cell depends only on the STATEs and PARAMETERs of that model *at that location*. They may also depend on voltage and ionic variables, but these also are *at that location* and automatically available to the model. To see how to do this, we will examine models of graded synaptic transmission, gap junctions, and spike−triggered synaptic transmission.

> Models with `LONGITUDINAL_DIFFUSION` might also be considered "nonlocal," but their dependence on concentration in adjacent segments is handled automatically by the NMODL translator.

Example 10.1: graded synaptic transmission

At synapses that operate by graded transmission, neurotransmitter is released continuously, and the rate of release depends on something in the presynaptic terminal. For the sake of discussion, let's say this something is $[Ca^{2+}]_{pre}$, the concentration of free calcium in the presynaptic terminal. We will also assume that the transmitter changes an ionic conductance in the postsynaptic cell.

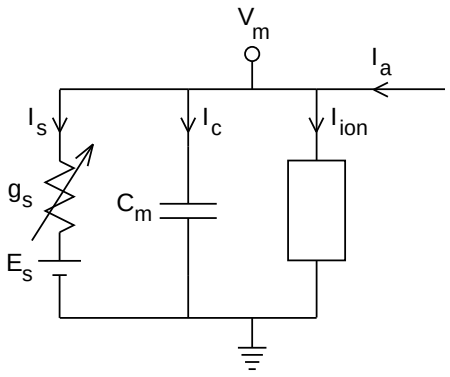

Figure 10.1. Membrane potential in the immediate neighborhood of a postsynaptic conductance depends on the synaptic current (I_s), the currents through the local membrane capacitance and ionic conductances (I_c and I_{ion}), and the axial current arriving from adjacent regions of the cell (I_a).

From the standpoint of the postsynaptic cell, a simplified abstraction of a conductance−change synapse might look like Fig. 10.1, where g_s, E_s, and I_s are the synaptic conductance, equilibrium potential, and current, respectively. The effect of graded synaptic transmission on the postsynaptic cell is expressed in Equation 10.1.

$$C_m \frac{dV_m}{dt} + I_{ion} = I_a - (V_m - E_s) \cdot g_s([Ca^{2+}]_{pre}) \qquad \text{Eq. 10.1}$$

This is the familiar mathematical statement of charge balance applied to the electrical vicinity of the postsynaptic region. The terms on the left hand side are the usual local capacitive and ionic transmembrane currents. The first term on the right hand side is the current that enters the postsynaptic region from adjacent parts of the cell, which

NEURON takes care of automatically. The second term on the right hand side expresses the effect of the ligand−gated channels. The current through these channels is the product of two factors. The first factor is merely the local electrochemical gradient for ion flow. The second factor is a conductance term that depends on the calcium concentration at some other location. This could well be in a completely different section, and its value might change with every `fadvance()`.

We could insert a `hoc` statement like this into the main computational loop

```
somedendrite.syn.capre = precell.bouton.cai(0.5)
```

At each time step, this would update the variable `capre` in the synaptic mechanism `syn` attached to the postsynaptic section `somedendrite`, making it equal to the free calcium concentration `cai` in the section called `bouton` in the presynaptic cell `precell`. However, this instruction would have to be reinterpreted at each `fadvance()`, which might slow down the simulation considerably.

If the moment−by−moment details of what is going on in the presynaptic terminal are important to what happens to the postsynaptic cell, it is far more efficient to use a `POINTER`. A `POINTER` in NMODL holds a reference to another variable. The specific reference is defined by a `hoc` statement, as we shall see as we examine a simple model of graded synaptic transmission (Listing 10.1).

`POINTER` variables are not limited to point processes. Distributed mechanisms can also use `POINTER`s, although perhaps for very different purposes.

```
: Graded synaptic transmission

NEURON {
  POINT_PROCESS GradSyn
  POINTER capre
  RANGE e, k, g, i
  NONSPECIFIC_CURRENT i
}

UNITS {
  (nA) = (nanoamp)
  (mV) = (millivolt)
  (uS) = (microsiemens)
  (molar) = (1/liter)
  (mM) = (millimolar)
}

PARAMETER {
  e = 0  (mV) : reversal potential
  k = 0.02  (uS/mM3)
}

ASSIGNED {
  v  (mV)
  capre (mM)  : presynaptic [Ca]
  g  (uS)
  i  (nA)
}
```



```
BREAKPOINT {
  g = k*capre^3
  i = g*(v - e)
}
```

Listing 10.1. gradsyn.mod

The NEURON block

The POINTER statement in the NEURON block declares that capre refers to some other variable that may belong to a different section; below we show how to attach this to the free calcium concentration in a presynaptic terminal. Instead of a peak conductance, the synaptic strength is governed by k, a "transfer function slope", which has units of ($\mu S/mM^3$).

The BREAKPOINT block

The synaptic conductance g is proportional to the cube of capre and does not saturate. This is similar to the calcium dependence of synaptic conductance in a model described by De Schutter et al. [, 1993 #717].

Usage

After creating a new instance of the GradSyn point process, we link its POINTER variable to the variable at some other location we want it to follow with hoc statements, e.g.

```
objref syn
somedendrite syn = new GradSyn(0.8)
setpointer syn.cp, precell.bouton.cai(0.5)
```

The second statement attaches an instance of the GradSyn mechanism, called syn, to somedendrite. The third statement asserts that the synaptic conductance of syn will be governed by cai in the middle of a section called bouton that is part of cell precell. Of course this assumes that the presynaptic section precell.bouton contains a calcium accumulation mechanism.

Figure 10.2 shows simulation results from a model of graded synaptic transmission. In this model, the presynaptic terminal precell is a 1 μm diameter hemisphere with voltage−gated calcium current cachan (cachan.mod in nrn54/examples/nrniv/nmodl under MSWindows or nrn-5.4/share/examples/nrniv/nmodl under UNIX) and a calcium accumulation mechanism that includes diffusion, buffering, and a pump (cdp, discussed in **Example 9.9: a calcium pump**). The postsynaptic cell is a passive single compartment with surface area 100 μm^2, C_m = 1 $\mu f/cm^2$, and τ_m = 30 ms. A GradSyn synapse with transfer function slope k = 0.2 $\mu S/mM^3$ is attached to the postsynaptic cell, and presynaptic membrane potential is driven between −70 and −30 mV by a sinusoid with a period of 400 ms. The time course of presynaptic $[Ca]_i$ and synaptic conductance show clipping of the negative phases of the sine wave; the postsynaptic membrane potential shows less clipping because of filtering by membrane capacitance.

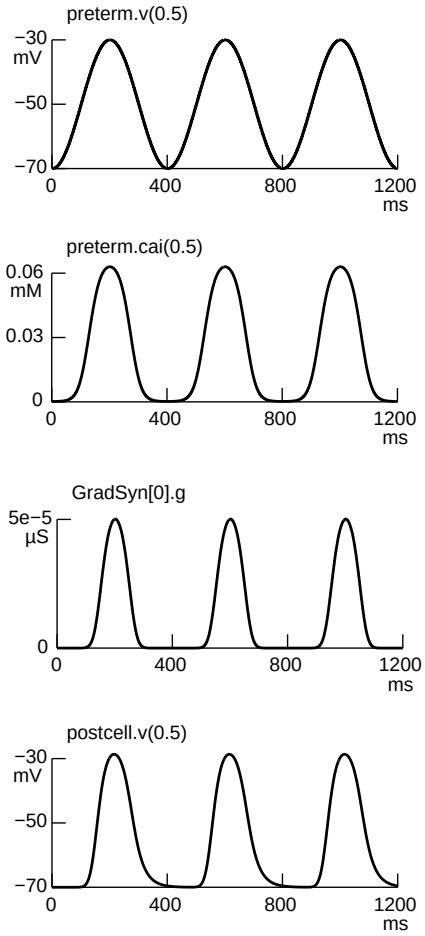

Figure 10.2. Graded synaptic transmission. Top two graphs: Presynaptic membrane potential `preterm.v` was "clamped" to $-70-20\cos(2\pi t/400)$ mV, producing a periodic increase of $[Ca]_i$ (`preterm.cai` is the concentration just inside the cell membrane) with clipping of the negative peaks. Bottom two graphs: The synaptic conductance GradSyn[0].g shows more even more clipping of the negative phases of the sinusoid, but membrane capacitance smoothes the time course of postsynaptic membrane potential.

Example 10.2: a gap junction

Gap junctions and ephaptic synapses can be handled by a pair of point processes on the two sides of the junction that point to each other's voltage, as in

```
section1 gap1 = new Gap(x1)
section2 gap2 = new Gap(x2)
setpointer gap1.vpre, section2.v(x2)
setpointer gap2.vpre, section1.v(x1)
```

Linear and nonlinear gap junctions can also be implemented with the linear circuit builder.

Listing 10.2 presents the NMODL specification of a point process that can be used in just this way to implement ohmic gap junctions.


```
NEURON {
  POINT_PROCESS Gap
  POINTER vgap
  RANGE r, i
  NONSPECIFIC_CURRENT i
}

PARAMETER { r = 1e10 (megohm) }

ASSIGNED {
  v (millivolt)
  vgap (millivolt)
  i (nanoamp)
}

BREAKPOINT { i = (v - vgap)/r }
```

Listing 10.2. `gap.mod`

This implementation can introduce spurious oscillations if the coupling between the two voltages is too tight (i.e. if the resistance `r` is too low) because it degrades the Jacobian matrix of the system equations. While it does introduce off−diagonal terms to couple the nodes on either side of the gap junction, it fails to add the conductance of the gap junction to the terms on the main diagonal. The result is an *approximate* Jacobian, so that the integration method is effectively modified Euler. This produces satisfactory results only if coupling is loose (i.e. if `r` is large compared to the total conductance of the other ohmic paths attached to the affected nodes). If oscillations do occur, their amplitude can be reduced by reducing `dt`, and they can be eliminated by using CVODE. We should mention that an alternative way to implement gap junctions is with the linear circuit method, which properly sets diagonal and off−diagonal terms of the Jacobian so that simulations are completely stable.

Usage

The following `hoc` code use this mechanism to set up a model of a gap junction between two cells. The `Gap` mechanisms allow current to flow between the internal node at the 1 end of `a` and the internal node at the 0 end of `b`.

```
create a,b
access a

forall {nseg=10 L=1000 diam=10 insert hh}

objref g[2]
for i=0,1 {
  g[i] = new Gap()
  g[i].r = 3
}

a g[0].loc(0.9999)  // "distal" end of a
b g[1].loc(0.0001)  // "proximal" end of b
setpointer g[0].vgap, b.v(0.0001)
setpointer g[1].vgap, a.v(0.9999)
```


Modeling spike−triggered synaptic transmission: an event−based strategy

Prior to NEURON 4.1, model descriptions of synaptic transmission could only use `POINTER` variables to obtain their presynaptic information. This required a detailed piecing together of individual components that was acceptable for models with only a few synapses. Models of larger networks required users to exert considerable administrative effort to create mechanisms that handle synaptic delay, exploit very great simulation efficiencies offered by simplified models of synapses, and maintain information about network connectivity.

The experience of NEURON users in creating special strategies for managing network simulations (e.g. [Destexhe, 1994 #267][Lytton, 1996 #206]) stimulated the development of a network connection (`NetCon`) class and an event delivery system. Instances of the `NetCon` class manage the delivery of presynaptic "spike" events to synaptic point processes via NEURON's event delivery system. This works for all of NEURON's integrators, including the local variable time step method in which each cell is integrated with a time step appropriate to its own state changes. Model descriptions of synapses never need to queue events, and heroic efforts are not needed to make them work properly with variable time step methods. These features offer enormous convenience to users who are interested in models that involve synaptic transmission at any level of complexity from the single cell to large networks.

Conceptual model

In its most basic form, the physical system that we want to represent consists of a presynaptic neuron with a spike trigger zone that gives rise to an axon, which leads to a terminal that makes a synaptic connection onto a postsynaptic cell (Fig. 10.3). Our conceptual model of spike−triggered transmission is that arrival of a spike triggers some effect (e.g. a conductance change) on the postsynaptic cell that is described by a differential equation or kinetic scheme. Details of what goes on at the spike trigger zone are assumed to have no effect on what happens at the synapse. All that matters is whether a spike has, or has not, reached the presynaptic terminal. This conceptual model lets us take advantage of special features of NEURON that allow extremely efficient computation.

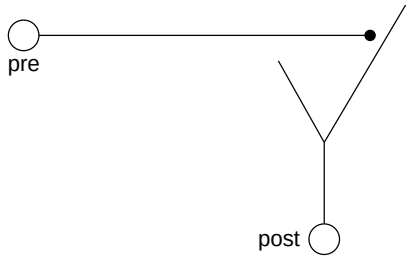

Figure 10.3. Cartoon of a synaptic connection (filled circle) between a presynaptic cell pre and a postsynaptic cell post.

A first approach to implementing a computational representation of our conceptual model might be something like the top of Fig. 10.4. We would monitor membrane potential at the presynaptic terminal for spikes (watch for threshold crossing). When a spike is detected, we wait for an appropriate delay (latency of transmitter release plus diffusion time) and then notify the synaptic mechanism that it's time to go into action. For this simple example, we have assumed that synaptic transmission simply causes a conductance change in the postsynaptic cell. It is also possible to implement more complex mechanisms that include representations of processes in the presynaptic terminal (e.g. processes involved in use−dependent plasticity).

We could speed things up a lot by leaving out the axon and presynaptic terminal entirely, i.e. instead of computing the propagation of the action potential along the axon, just monitor the spike trigger zone. Once a spike occurs, we wait for a total delay equal to the sum of the conduction latency and the synaptic latency, and then activate the postsynaptic conductance change (Fig. 10.4 bottom).

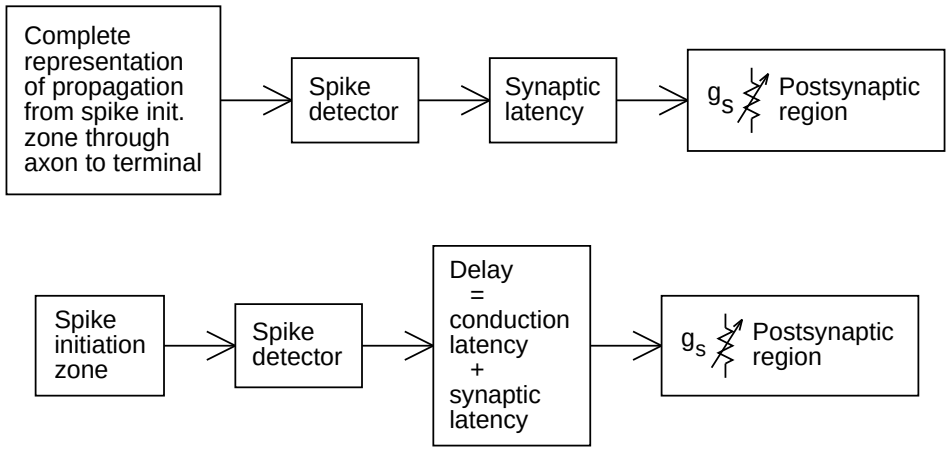

Figure 10.4. Computational implementation of a model of spike−triggered synaptic transmission. Top: The basic idea is that a presynaptic spike causes some change in the postsynaptic cell. Bottom: A more efficient version doesn't bother computing conduction in the presynaptic axon.

The `NetCon` class

Let's step back from this problem for a moment and think about the bottom diagram in Fig. 10.4. The "spike detector" and "delay" in the middle of this diagram are the seed of an idea for a general strategy for dealing with synaptic connections. In fact, the `NetCon` object class is used to apply this strategy in defining the synaptic connection between a source and a target.

A `NetCon` connects a presynaptic variable, such as voltage, to a target point process (here a synapse) with arbitrary delay and weight (individually specified on a per `NetCon` instance). If the presynaptic variable crosses threshold in a positive direction at time `t`, then at time `t+delay` a special `NET_RECEIVE` procedure in the target point process is called and receives the weight information. The only constraint on `delay` is that it be nonnegative. There is no limit on the number of events that can be "in the pipeline," and

there is no loss of events under any circumstances. Events always arrive at the target at the interval `delay` after the time they were generated.

When you create a `NetCon` object, at a minimum you must specify the source variable and the target. The source variable is generally the membrane potential of the currently accessed section, as shown here. The target is a point process that contains a `NET_RECEIVE` block (see Listing 10.3 below).

```
section netcon = new NetCon(&v(x), target, thresh, del, wt)
```

Threshold, delay, and weight are optional; their defaults are shown here, and they can be specified after the `NetCon` object has been constructed.

```
netcon.threshold = 10 // mV
netcon.delay = 1      // ms
netcon.weight = 0     // uS
```

The weight in these `hoc` statements is actually the first element of a weight vector. The number of elements in the weight vector depends on the number of arguments in the `NET_RECEIVE` statement of the NMODL source code that defines the point process. We will return to this later.

The `NetCon` class reduces the computational burden of network simulations, because the event delivery system supports efficient, unlimited fan−out and fan−in (divergence and convergence). To see what this implies, first consider fan−out. What if a presynaptic cell projects to multiple postsynaptic targets (Fig. 10.5 top)? Easy enough−−just add a `NetCon` object for each target (Fig. 10.5 bottom). This is computationally efficient because threshold detection is done on a "per source" basis, rather than a "per `NetCon` basis." That is, if multiple `NetCon`s have the same source, they all share a single threshold detector. The presynaptic variable is checked only once per time step and, when it crosses threshold in the positive direction, events are generated for each connecting `NetCon` object. Each `NetCon` can have its own weight and latency, and the target synaptic mechanisms do not have to be of the same class.

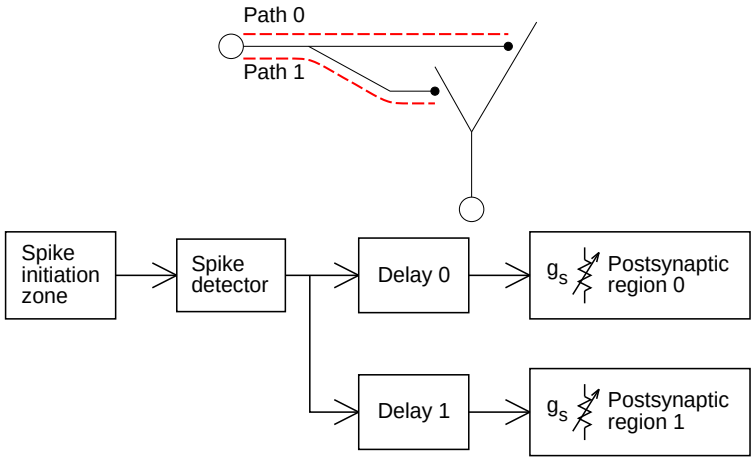

Figure 10.5. Efficient divergence. Top: A single presynaptic neuron projects to two different target synapses. Bottom: Computational model of this circuit uses multiple `NetCon`s with a single threshold detector that monitors a common source.

Now consider fan−in. Suppose a neuron receives multiple inputs that are anatomically close to each other and of the same type (Fig. 10.6 top). In other words, we're assuming that each synapse has its postsynaptic action through the same kind of mechanism (i.e. it has identical kinetics, and (in the case of conductance−change synapses) the same equilibrium potential). We can represent this by connecting multiple `NetCon` objects to the same postsynaptic point process (Fig. 10.6 bottom). This yields large efficiency improvements because a single set of synaptic equations can be shared by many input streams (one input stream per connecting `NetCon` instance). Of course, these synapses can have different strengths and latencies, because each `NetCon` object has its own weight and delay.

Having seen the rationale for using events with models of synaptic transmission, we are ready to examine some point processes that include a `NET_RECEIVE` procedure and can be used as synaptic mechanisms.

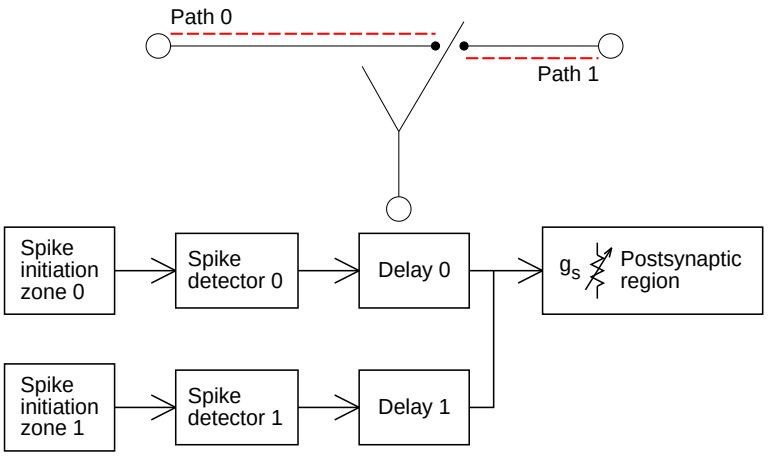

Figure 10.6. Efficient convergence. Top: Two different presynaptic cells make synaptic connections of the same class that are electrically close to each other. Bottom: Computational model of this circuit uses multiple `NetCon`s that share a single postsynaptic mechanism (single equation handles multiple input streams).

Example 10.3: synapse with exponential decay

Many kinds of synapses produce a synaptic conductance that increases rapidly and then declines gradually with first order kinetics, e.g. AMPAergic excitatory synapses. This can be modeled by an abrupt change in conductance, which is triggered by arrival of an event, and then decays with a single time constant.

The NMODL code that implements such a mechanism is shown in Listing 10.3. The synaptic conductance of this `ExpSyn` mechanism summates not only when events arrive from a single presynaptic source, but also when they arrive from different places (multiple input streams). This mechanism handles both these situations by defining a single conductance state g which is governed by a differential equation with the solution

$g(t) = g(t_0)\, e^{(t-t_0)/\tau}$, where $g(t_0)$ is the conductance at the time of the most recent event.

```
: expsyn.mod

NEURON {
  POINT_PROCESS ExpSyn
  RANGE tau, e, i
  NONSPECIFIC_CURRENT i
}

PARAMETER {
  tau = 0.1  (ms)
  e   = 0    (millivolt)
}

ASSIGNED {
  v  (millivolt)
  i  (nanoamp)
}

STATE { g (microsiemens) }

INITIAL { g = 0 }

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

DERIVATIVE state { g' = -g/tau }

NET_RECEIVE(weight (microsiemens)) {
  g = g + weight
}
```

Listing 10.3. `expsyn.mod`

The `BREAKPOINT` block

The main computational block of this mechanism is the `BREAKPOINT` block. This contains the `SOLVE` statement that tells how states will be integrated. The kinetics of `ExpSyn` are described by a differential equation of the form $y' = f(y)$ where $f(y)$ is linear in y, so it uses the `cnexp` method (see also **The `DERIVATIVE` block** in **Example 9.4: a voltage−gated current** in **Chapter 9**). The `BREAKPOINT` block ends with an assignment statement that sets the value of the synaptic current.

The `DERIVATIVE` block

The DERIVATIVE block contains the differential equation that describes the time course of the synaptic conductance `g`: a first order decay with time constant `tau`.

The NET_RECEIVE block

The NET_RECEIVE block contains the code that specifies what happens in response to presynaptic activation. This is called by the NetCon event delivery system when an event arrives at this point process.

So suppose we have a model with an ExpSyn point process that is the target of a NetCon. Imagine that the NetCon detects a presynaptic spike at time t. What happens next?

The ExpSyn's conductance g continues to follow a smooth exponential decay with time constant tau until time t+delay, where delay is the delay associated with the NetCon object. At this point, an event is delivered to the ExpSyn. Just before entry to the NET_RECEIVE block, NEURON makes all STATEs, v, and values assigned in the BREAKPOINT block consistent at t+delay. Then the code in the NET_RECEIVE block is executed, making the synaptic conductance jump up suddenly by an amount equal to the NetCon's weight.

> As we mentioned in **Chapter 9**, earlier versions of NEURON had to change g with a state_discontinuity() statement. This is no longer necessary.

Usage

Suppose we wanted to set up an ExpSyn synaptic connection between the cells shown in Fig. 10.7. This could be done with the following hoc code, which also illustrates the use of a List of NetCon objects as a means for keeping track of the synaptic connections in a network.

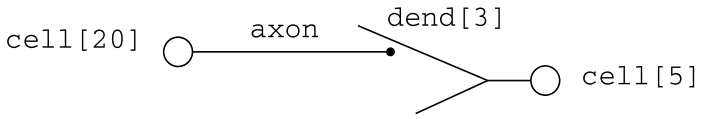

Figure 10.7

```
// keep connectivity in a list of NetCon objects
objref ncl
ncl = new List()

// attach an ExpSyn point process called syn
//   to the 0.3 location on dend[3] of cell[5]
objref syn
cell[5].dend[3] syn = new ExpSyn(0.3)

// presynaptic v is cell[20].axon.v(1)
// connect this to syn via a new NetCon object
//   and add the NetCon to the list ncl
cell[20].axon ncl.append(new NetCon(&v(1), \
         syn, threshold, delay, weight)
```


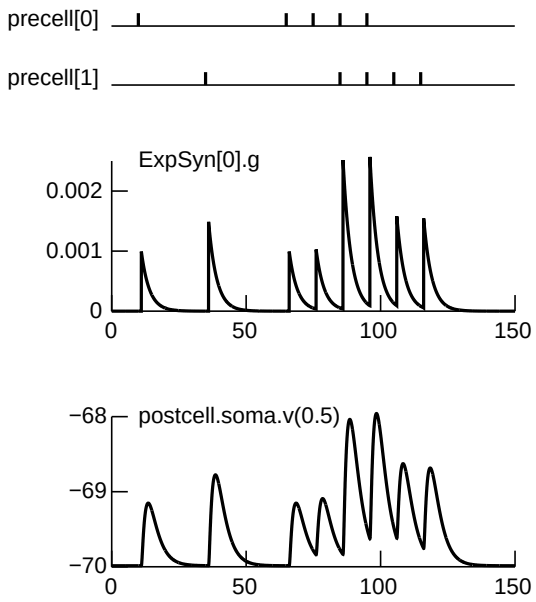

Figure 10.8. Simulation results from the model shown in Fig. 10.6. Note stream−specific synaptic weights and temporal summation of synaptic conductance and membrane potential.

Figure 10.8 shows results of a simulation of two input streams that converge onto a single `ExpSyn` attached to a postsynaptic cell, as in the diagram at the top of Fig. 10.6. The presynaptic firing times are indicated by the rasters labeled `precell[0]` and `precell[1]`. The synaptic conductance and postsynaptic membrane potential (middle and bottom graphs) display stream−specific synaptic weights, and also show temporal summation of inputs within an individual stream and between inputs on multiple streams.

Example 10.4: alpha function synapse

With a few small changes, we can extend `ExpSyn` to implement an alpha function synapse. We only need to replace the differential equation with the two state kinetic scheme

```
STATE { a (microsiemens) g (microsiemens) }
KINETIC state {
  ~ a <-> g (1/tau, 0)
  ~ g -> (1/tau)
}
```

and change the `NET_RECEIVE` block to

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

The factor `exp(1)` = *e* is included so that an isolated event produces a peak conductance of magnitude `weight`, which occurs at time `tau` after the event. Since this mechanism involves a `KINETIC` block instead of a `DERIVATIVE` block, the integration method specified by the `SOLVE` statement must be changed from `cnexp` to `sparse`.

The extra computational complexity of using a kinetic scheme is offset by the fact that, no matter how many `NetCon` streams connect to this model, the computation time required to integrate `STATE g` remains constant. Some increase of efficiency can be gained by recasting the kinetic scheme as two linear differential equations

```
DERIVATIVE state {
..a′ = −a/tau1
..b′ = −b/tau
..g = b − a
}
```

which are solved efficiently by the `cnexp` method. As `tau1` approaches `tau`, `g` approaches an alpha function (although the factor by which `weight` must be multiplied approaches infinity; see `factor` in the next example). Also, there are now two state discontinuities in the `NET_RECEIVE` block

```
NET_RECEIVE(weight (microsiemens)) {
  a = a + weight*factor
  b = b + weight*factor
}
```

Example 10.5: Use−dependent synaptic plasticity

Here the alpha function synapse is extended to implement a form of use−dependent synaptic plasticity. Each presynaptic event initiates two distinct processes: direct activation of ligand−gated channels, which causes a transient conductance change, and activation of a mechanism that in turn modulates the conductance change produced by successive synaptic activations. In this example we presume that modulation depends on the postsynaptic increase of a second messenger, which we will call "G protein" for illustrative purposes. We must point out that this example is entirely hypothetical, and that it is quite different from models described by others [Destexhe, 1995 #168] in which the G protein itself gates the ionic channels.

For this mechanism it is essential to distinguish each stream into the generalized synapse, since each stream has to maintain its own [G] (concentration of activated G protein). That is, streams are independent of each other in terms of the effect on [G], but their effects on synaptic conductance show linear superposition.

```
: gsyn.mod

NEURON {
  POINT_PROCESS GSyn
  RANGE tau1, tau2, e, i
  RANGE Gtau1, Gtau2, Ginc
  NONSPECIFIC_CURRENT i
  RANGE g
}

UNITS {
  (nA)   = (nanoamp)
  (mV)   = (millivolt)
  (umho) = (micromho)
}
```



```
PARAMETER {
  tau1   = 1  (ms)
  tau2   = 1.05    (ms)
  Gtau1  = 20   (ms)
  Gtau2  = 21   (ms)
  Ginc   = 1
  e      = 0    (mV)
}

ASSIGNED {
  v  (mV)
  i  (nA)
  g  (umho)
  factor
  Gfactor
}

STATE {
  A  (umho)
  B  (umho)
}

INITIAL {
  LOCAL tp
  A = 0
  B = 0
  tp = (tau1*tau2)/(tau2 - tau1) * log(tau2/tau1)
  factor = -exp(-tp/tau1) + exp(-tp/tau2)
  factor = 1/factor
  tp = (Gtau1*Gtau2)/(Gtau2 - Gtau1) * log(Gtau2/Gtau1)
  Gfactor = -exp(-tp/Gtau1) + exp(-tp/Gtau2)
  Gfactor = 1/Gfactor
}

BREAKPOINT {
  SOLVE state METHOD cnexp
  g = B - A
  i = g*(v - e)
}

DERIVATIVE state {
  A' = -A/tau1
  B' = -B/tau2
}

NET_RECEIVE(weight (umho), w, G1, G2, t0 (ms)) {
  G1 = G1*exp(-(t-t0)/Gtau1)
  G2 = G2*exp(-(t-t0)/Gtau2)
  G1 = G1 + Ginc*Gfactor
  G2 = G2 + Ginc*Gfactor
  t0 = t

  w = weight*(1 + G2 - G1)
  A = A + w*factor
  B = B + w*factor
}
```

Listing 10.4. `gsyn.mod`

The NET_RECEIVE block

The conductance of the ligand−gated ion channel uses the differential equation approximation for an alpha function synapse. The peak synaptic conductance depends on the value of [G] at the moment of synaptic activation. A similar, albeit much slower, alpha function approximation describes the time course of [G]. These processes peak approximately `tau1` and `Gtau1` after delivery of an event, respectively.

The peak synaptic conductance elicited by an active `NetCon` is specified in the `NET_RECEIVE` block, where `w = weight*(1+G2−G1)` describes how the effective weight of the synapse is modified by [G]. Even though conductance is integrated, [G] is needed only at discrete event times so it can be computed analytically from the elapsed time since the prior synaptic activation. The `INITIAL` block sets up the factors that are needed to make the peak changes equal to the values of `w` and `Ginc`.

Note that `G1` and `G2` are not `STATE`s in this mechanism. They are not even variables in this mechanism, but instead are "owned" by the particular `NetCon` instance that delivered the event. A `NetCon` object instance keeps an array of size equal to the number of arguments to `NET_RECEIVE`, and the arguments to `NET_RECEIVE` are really references to the elements of this array. Unlike the arguments to a `PROCEDURE` or `FUNCTION` block, which are "call by value," the arguments to a `NET_RECEIVE` block are "call by reference." This means that assignment statements in `gsyn.mod`'s `NET_RECEIVE` block can change the values of variables that belong to the `NetCon` object. The individual `NetCon` objects all contribute linearly to the total synaptic conductance, but there is a separate array for each `NetCon` object that connects to this model, so each connection has its own [G]. Thus this mechanism uses "stream−specific plasticity" to emulate "synapse−specific plasticity."

On initialization, all arguments of `NET_RECEIVE` after the first one are automatically set to 0.

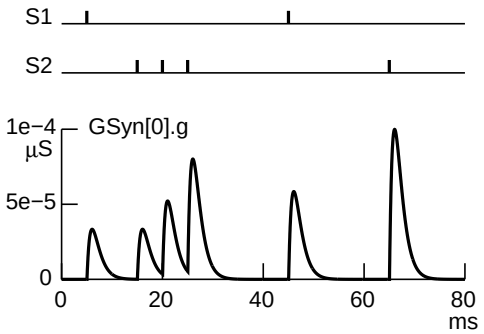

Figure 10.9. Simulation results from the model shown in Fig. 10.6 when the synaptic mechanism is `GSyn`. Note stream−specific use−dependent plasticity.

To illustrate the operation of this mechanism, imagine the network of Fig. 10.6 with a single `GSyn` driven by the two spike trains shown in Fig. 10.9. This emulates two synapses that are electrotonically close to each other, but with separate pools of [G]. The train with spikes at 5 and 45 ms (S1) shows some potentiation of the second conductance

transient, but the train that starts at 15 ms with a 200 Hz burst of three spikes displays a large initial potentiation that is even larger when tested after a 40 ms interval.

Example 10.6: saturating synapses

Several authors (e.g. [Destexhe, 1994 #267], [Lytton, 1996 #206]) have used synaptic transmission mechanisms based on a simple conceptual model of transmitter−receptor interaction:

$$C + T \underset{\beta}{\overset{\alpha}{\rightleftarrows}} O \qquad \text{Eq. 10.2}$$

where transmitter *T* binds to a closed receptor channel *C* to produce an open channel *O*. In this conceptual model, spike−triggered transmitter release produces a transmitter concentration in the synaptic cleft that is approximated by a rectangular pulse with a fixed duration and magnitude (Fig. 10.10). A "large excess of transmitter" is assumed, so that while transmitter is present (the "onset" state, "ligand binding to channel") the postsynaptic conductance increases toward a maximum value with a single time constant $1/(\alpha T + \beta)$. After the end of the transmitter pulse (the "offset" state, "ligand·channel complex dissociating"), the conductance decays with time constant $1/\beta$. Further details of saturating mechanisms are covered by [Destexhe, 1994 #267][Destexhe, 1994 #266] and [Lytton, 1996 #206].

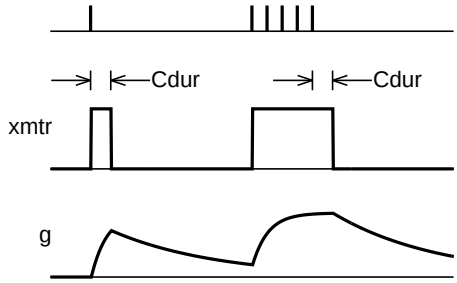

Figure 10.10. A saturating synapse model. A single presynaptic spike (top trace) causes a pulse of transmitter in the synaptic cleft with fixed duration (Cdur) and concentration (middle trace). This elicits a rapid increase of postsynaptic conductance followed by a slower decay (bottom trace). A high frequency burst of spikes produces a sustained elevation of transmitter that persists until Cdur after the last spike and causes saturation of the postsynaptic conductance.

There is an ambiguity when one or more spikes arrive on a single stream during the onset state triggered by an earlier spike: should the mechanism ignore the "extra" spikes, concatenate onset states to make the transmitter pulse longer without increasing its concentration, or increase (summate) the transmitter concentration? Summation of transmitter requires the onset time constant to vary with transmitter concentration. This places transmitter summation outside the scope of the Destexhe/Lytton model, which assumes a fixed time constant for the onset state. We resolve this ambiguity by choosing

concatenation, so that repetitive impulses on one stream produce a saturating conductance change (Fig. 10.10). However, conductance changes elicited by separate streams will summate.

A model of the form used in Examples 10.4 and 10.5 can capture the idea of saturation, but the separate onset/offset formulation requires keeping track of how much "material" is in the onset or offset state. The mechanism in Listing 10.5 implements an effective strategy for doing this. A noteworthy feature of this model is that the event delivery system serves as more than a conduit for receiving inputs from other cells: discrete events are used to govern the duration of synaptic activation, and are thus an integral part of the mechanism itself.

```
: ampa.mod

NEURON {
  POINT_PROCESS AMPA_S
  RANGE g
  NONSPECIFIC_CURRENT i
  GLOBAL Cdur, Alpha, Beta, Erev, Rinf, Rtau
}

UNITS {
  (nA)   = (nanoamp)
  (mV)   = (millivolt)
  (umho) = (micromho)
}

PARAMETER {
  Cdur  = 1.0   (ms)  : transmitter duration (rising phase)
  Alpha = 1.1   (/ms) : forward (binding) rate
  Beta  = 0.19  (/ms) : backward (dissociation) rate
  Erev  = 0     (mV)  : equilibrium potential
}

ASSIGNED {
  v     (mV)   : postsynaptic voltage
  i     (nA)   : current = g*(v - Erev)
  g     (umho) : conductance
  Rtau  (ms)   : time constant of channel binding
  Rinf  : fraction of open channels if xmtr is present "forever"
  synon : sum of weights of all synapses in the "onset" state
}

STATE { Ron Roff }  : initialized to 0 by default
: Ron and Roff are the total conductances of all synapses
:   that are in the "onset" (transmitter pulse ON)
:   and "offset" (transmitter pulse OFF) states, respectively

INITIAL {
  synon = 0
  Rtau = 1 / (Alpha + Beta)
  Rinf = Alpha / (Alpha + Beta)
}
```



```
BREAKPOINT {
  SOLVE release METHOD cnexp
  g = (Ron + Roff)*1(umho)
  i = g*(v − Erev)
}

DERIVATIVE release {
  Ron' = (synon*Rinf − Ron)/Rtau
  Roff' = −Beta*Roff
}

NET_RECEIVE(weight, on, r0, t0 (ms)) {
  : flag is an implicit argument of NET_RECEIVE, normally 0
  if (flag == 0) {
    : a spike arrived, start onset state if not already on
    if (!on) {
      : this synapse joins the set of synapses in onset state
      synon = synon + weight
      r0 = r0*exp(−Beta*(t − t0)) : r0 at start of onset state
      Ron = Ron + r0
      Roff = Roff − r0
      t0 = t
      on = 1
      : come again in Cdur with flag = 1
      net_send(Cdur, 1)
    } else {
      : already in onset state, so move offset time
      net_move(t + Cdur)
    }
  }
  if (flag == 1) {
    : "turn off transmitter"
    : i.e. this synapse enters the offset state
    synon = synon − weight
    : r0 at start of offset state
    r0 = weight*Rinf + (r0 − weight*Rinf)*exp(−(t − t0)/Rtau)
    Ron = Ron − r0
    Roff = Roff + r0
    t0 = t
    on = 0
  }
}
```

Listing 10.5. `ampa.mod`

The PARAMETER block

The actual value of the transmitter concentration in the synaptic cleft during the onset state is unimportant to this model, as long as it remains constant. To simplify the mechanism, we assume transmitter concentration to be dimensionless and have a numeric value of 1. This allows us to specify the forward rate constant `Alpha` with units of 1/ms.

The STATE block

This mechanism has two STATEs. `Ron` is the total conductance of all synapses that are in the onset state, and `Roff` is the total conductance of all synapses that are in the

offset state. These are declared without units, so a units factor will have to be applied elsewhere (in this example, this is done in the BREAKPOINT block).

The INITIAL block

At the start of a simulation, we assume that all channels are closed and no transmitter is present at any synapse. The initial values of Ron, Roff, and synon must therefore be 0. This initialization happens automatically for STATEs and does not require explicit specification in the INITIAL block, but synon needs an assignment statement.

The INITIAL block also calculates Rtau and Rinf. Rtau is the time constant for equilibration of the closed (free) and open (ligand−bound) forms of the postsynaptic receptors when transmitter is present in the synaptic cleft. Rinf is the open channel fraction if transmitter is present forever.

The BREAKPOINT and DERIVATIVE blocks

The total conductance is numerically equal to Ron+Roff. The *1(umho) factor is included for dimensional consistency.

The DERIVATIVE block specifies the first order differential equations that govern these STATEs. The meaning of each term in

```
Roff' = −Beta*Roff
```

is obvious, and in

```
Ron' = (synon*Rinf − Ron)/Rtau
```

the product synon*Rinf is the value that Ron approaches with increasing time.

The NET_RECEIVE block

The NET_RECEIVE block performs the task of switching each synapse between its onset and offset states. In broad outline, if an external event (an event generated by the NetCon's source passing threshold) arrives at time t to start an onset, the NET_RECEIVE block generates an event that it sends to itself. This self−event will be delivered at time t+Cdur, where Cdur is the duration of the transmitter pulse. Arrival of the self−event is the signal to switch the synapse back to the offset state. If a spike (external event) arrives from the same NetCon before the self−event does, the self−event is moved to a new time that is Cdur in the future. Thus resetting to the offset state can happen only if an interval of Cdur passes without new spikes arriving.

To accomplish this strategy, the NET_RECEIVE block must distinguish an external event from a self−event. It does this by exploiting the facts that every event has an implicit argument called flag, and the value of flag is 0 for an external event.

The event flag is "call by value," unlike the explicit arguments that are declared inside the parentheses of the NET_RECEIVE() statement, which are "call by reference."

Handling of external (spike) events

Arrival of an external event causes execution of the statements inside the `if (flag==0){}` clause. These begin with `if (!on)`, which tests whether this synapse should switch to the onset state.

Switching to the onset state involves keeping track of how much "material" is in the onset and offset states. This requires moving the synapse's channels into the pool of channels that are exposed to transmitter, which simply means adding the synapse's `weight` to `synon`. Also, the conductance of this synapse, which had been decaying with rate constant `1/Beta`, must now start to grow with rate constant `Rtau`. This is done by computing `r0`, the synaptic conductance at the present time `t`, and then adding `r0` to `Ron` and subtracting it from `Roff`. Next the value of `t0` is updated for future use, and `on` is set to 1 to signify that the synapse is in the onset state. The last statement inside `if (!on){}` is `net_send(Cdur,nspike)`, which statement generates an event with delay given by the first argument and flag value given by the second argument. All the explicit arguments of this self−event will have the values of this particular `NetCon`, so when this self−event returns we will know how much "material" to switch from the onset to the offset state.

The `else {}` clause takes care of what happens if another external event arrives while the synapse is still in the onset state. The `net_move(t+Cdur)` statement moves the self−event to a new time that is `Cdur` in the future (relative to the arrival time of the new external event). In other words, this prolongs synaptic activation until `Cdur` after the most recent input event.

Handling of self−events

When the self−event is finally delivered, it triggers an offset. We know it is a self−event because its `flag` is 1. Once again we keep track of how much "material" is in the onset and offset states, but now we subtract the synapse's `weight` from `synon` to remove the synapse's channels from the pool of channels that are exposed to transmitter. Likewise, the conductance of this synapse, which was growing with rate constant `Rtau`, must now begin to decay with rate constant `1/Beta`. Finally, the value of `t0` is updated and `on` is set to 0.

Artificial spiking cells

NEURON's event delivery system was created with the primary aim of making it easier to represent synaptic connections between biophysical model neurons. However, the event delivery system turns out to be quite useful for implementing a wide range of mechanisms with algorithms that require actions to be taken after a delay. The saturating synapse model presented above is just one example of this.

The previous section also showed how spike−triggered synaptic transmission makes extensive use of the network connection class to define connections between cells. The typical `NetCon` object watches a source cell for the occurrence of a spike, and then, after some delay, delivers a weighted input event to a target synaptic mechanism, i.e. it

represents axonal spike propagation. More generally, a `NetCon` object can be regarded as a channel on which a stream of events generated at a source is transmitted to a target. The target can be a point process, a distributed mechanism, or an artificial neuron (e.g. an integrate and fire model). The effect of events on a target is specified in NMODL by statements in a `NET_RECEIVE` block, which is called only when an event has been delivered.

The event delivery system also opens up a large domain of simulations in which certain types of artificial spiking cells, and networks of them, can be simulated hundreds of times faster than with numerical integration methods. Discrete event simulations are possible when all the state variables of a model cell can be computed analytically from a new set of initial conditions. That is, if an event occurs at time t_1, all state variables must be computable from the state values and time t_0 of the previous event. Since computations are performed only when an event occurs, total computation time is proportional to the number of events delivered and independent of the number of cells, number of connections, or problem time. Thus handling 100,000 spikes in one hour for 100 cells takes the same time as handling 100,000 spikes in 1 second for 1 cell.

The following examples analyze the three broad classes of integrate and fire cells that are built into NEURON. These artificial cells are point processes that serve as both targets and sources for `NetCon` objects. They are targets because they have a `NET_RECEIVE` block, which specifies how incoming events from one or more `NetCon` objects are handled, and details the calculations necessary to generate outgoing events. They are also sources because the same `NET_RECEIVE` block generates discrete output events which are delivered through one or more `NetCon` objects to targets. In order to preserve an emphasis on how NEURON's event delivery system was used to implement the dynamics of these mechanisms, we have omitted many details from the NMODL listings. Ellipses indicate elisions, and listings include *italicized pseudocode* where necessary for clarity. Complete source code for all three built−in artificial cell models is included with NEURON.

Example 10.7: `IntFire1`, a basic integrate and fire model

The simplest integrate and fire mechanism built into NEURON is `IntFire1`, which has a "membrane potential" state m which decays toward 0 with time constant τ.

$$\tau \frac{dm}{dt} + m = 0 \qquad \text{Eq. 10.3}$$

An input event of weight w adds instantaneously to m, and when $m > 1$ the cell "fires," producing an output event and returning m to 0. Negative weights are inhibitory while positive weights are excitatory. This is analogous to a cell with a membrane time constant τ that is very long compared to the time course of individual synaptic conductance changes. Every synaptic input to such a cell shifts membrane potential to a new level in a time that is much shorter than τ, and each cell firing erases all traces of prior inputs. An initial implementation of `IntFire1` is shown in Listing 10.6.


```
NEURON {
  ARTIFICIAL_CELL IntFire1
  RANGE tau, m
}

PARAMETER { tau = 10 (ms) }

ASSIGNED {
  m
  t0 (ms)
}

INITIAL {
  m = 0
  t0 = 0
}

NET_RECEIVE (w) {
  m = m*exp(-(t - t0)/tau)
  m = m + w
  t0 = t
  if (m > 1) {
    net_event(t)
    m = 0
  }
}
```

Listing 10.6. A basic implementation of `IntFire1`.

The `NEURON` block

As the introduction to this section mentions, artificial spiking cell models are point processes. However, the `NEURON` block of this and the other artificial cell models will instead use the keyword `ARTIFICIAL_CELL`. This synonym for `POINT_PROCESS` is intended as a reminder that this model has a `NET_RECEIVE` block, lacks a `BREAKPOINT` block, and does not have to be associated with a section location or numerical integrator. Consequently it is isolated from the usual things that link mechanisms to each other: it does not refer to membrane potential `v` or any ions, and it also does not have a `POINTER`. Instead, the "outside" can affect it only by sending it discrete events, and it can only affect the "outside" by sending discrete events.

The `NET_RECEIVE` block

The mechanisms we have seen so far use `BREAKPOINT` and `SOLVE` blocks to specify the calculations that are performed during a time step `dt`, but an artificial cell model does not have these blocks. Instead, calculations only take place when a `NetCon` delivers a new event, and these are performed in the `NET_RECEIVE` block. When a new event arrives, the present value of `m` is computed analytically and then incremented by the weight `w` of the event. In this model, the value of `m` is just the exponential decay from its value at the previous event; therefore the code contains variable `t0` which keeps track of the last event time.

If an input event drives `m` to or above threshold, the `net_event(t)` statement notifies all `NetCon`s that have this point process as their source that it fired a spike at time `t` (the argument to `net_event()` can be any time at or later than the current time `t`). Then the cell resets `m` to 0. The code in Listing 10.6 imposes no limit on firing frequency, but adding a refractory period is not difficult (see below). However, if a `NetCon` with `delay` of 0 and a `weight` of 1.1 has such an artificial cell as both its source and target, the system will behave "properly": events will be generated and delivered without time ever advancing.

There is no threshold test overhead at every `dt` because this point process has no variable for `NetCon`s to watch. That is, this artificial cell does not need the usual test for local membrane potential `v` to cross `NetCon.threshold`, which is essential at every time step for event generation with "real" cells. Furthermore the event delivery system only places the earliest event to be delivered on the event queue. When that time finally arrives, all targets whose `NetCon`s have the same source and delay get the event delivery, and longer delay streams are put back on the event queue to await their specific delivery time.

Enhancements to the basic mechanism

Visualizing the integration state

The integration state *m* is difficult to plot in an understandable manner, since the value of `m` remains unchanged in the interval between input events regardless of how many numerical integration steps were performed in that interval. Consequently `m` always has the value that was calculated after the last event was received, and a plot of `m` looks like a staircase (Fig. 10.11 left), with no apparent decay or indication of what the value of *m* was just before the event.

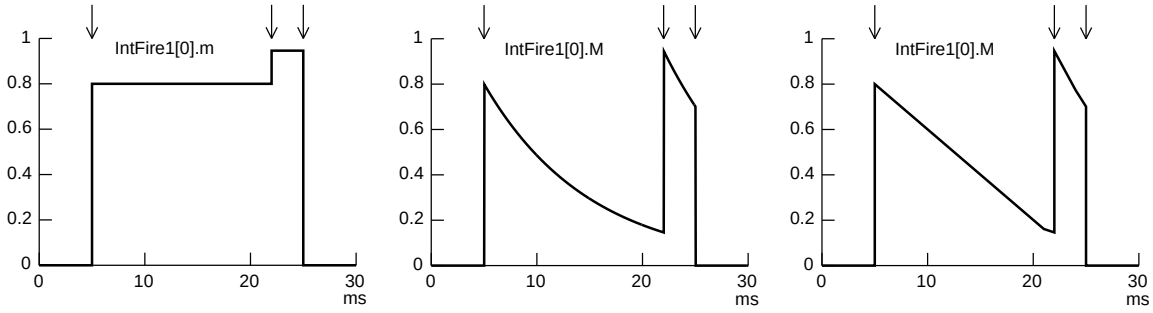

Figure 10.11. Response of an `IntFire1` cell with τ = 10 ms to input events with weight = 0.8 arriving at *t* = 5, 22, and 25 ms (arrows). The third input triggers a "spike." Left: The variable `m` is evaluated only when a new input event arrives, so its plot looks like a staircase. A function can be included in the IntFire1 mod file (see text) to better indicate the time course of the integration state *m*. Center: Plotting this function during a simulation with fixed `dt` (0.025 ms here) demonstrates the decay of *m* between input events. Right: In a variable time step simulation, *m* appears to follow a sequence of linear ramps. This is only an artifact of the Graph tool drawing lines between points that were computed analytically at a small number of times chosen by the integrator.

This can be partially repaired by adding a function

```
FUNCTION M() {
  M = m*exp(−(t − t0)/tau)
}
```

that returns the present value of the integration state *m*. This gives nice trajectories when fixed time step integration is used (Fig. 10.11 center). However, the natural step with the variable step method is the interspike interval itself, unless intervening events occur in other cells (e.g. 1 ms before the second input event in Fig. 10.11 right). At least the integration step function `fadvance()` returns 10^{-9} ms before and after the event to properly indicate the discontinuity in `M`.

Adding a refractory period

It is easy to add a relative refractory period by initializing `m` to a negative value after the cell fires (alternatively, a depolarizing afterpotential can be emulated by initializing `m` to values in the range (0,1)). However, incorporating an absolute refractory period requires self−events.

Suppose we want to limit the maximum firing rate to 200 spikes per second, which corresponds to an absolute refractory period of 5 ms. To specify the duration of the refractory period, we use a variable named `refrac`, which is declared and assigned a value of 5 ms in the `PARAMETER` block. Adding the statement `RANGE refrac` to the `NEURON` block allows us to adjust this parameter from the interpreter and graphical interface. We also use a variable to keep track of whether the point process is in the refractory period or not. The name we choose for this variable is the eponymous `refractory`, and it is declared in the `ASSIGNED` block and initialized to a value of 0 ("false") in the `INITIAL` block.

The `NET_RECEIVE` implementation is then

```
NET_RECEIVE (w) {
  if (refractory == 0) {
    m = m*exp(−(t − t0)/tau)
    m = m + w
    t0 = t
    if (m > 1) {
      net_event(t)
      refractory = 1
      net_send(refrac, refractory)
    }
  } else if (flag == 1) {
    : self−event arrived, so terminate refractory period
    refractory = 0
    m = 0
    t0 = t
  } : else ignore the external event
}
```

If `refractory` equals 0, the cell accepts external events (i.e. events delivered by a `NetCon`) and calculates the state variable `m` and whether to fire the cell. When the cell fires a spike, `refractory` is set to 1 and further external events are ignored until the end of the refractory period (Fig. 10.12).

Recall from the saturating synapse example that the `flag` variable that accompanies an external event is 0. If its value is non−zero, it must have been set by a call to `net_send()` when the cell fired. The `net_send(interval, flag)` statement places an event into the delivery system as an "echo" of the current event, i.e. it will come back to the sender after the specified `interval` with the specified `flag`. In this case we aren't interested in the weight but only the `flag`. Arrival of this self−event means that the refractory period is over.

The top of Fig. 10.12 shows the response of this model to a train of input stimuli. Temporal summation triggers a spike on the fourth input. The fifth input arrives during the refractory interval and has no effect.

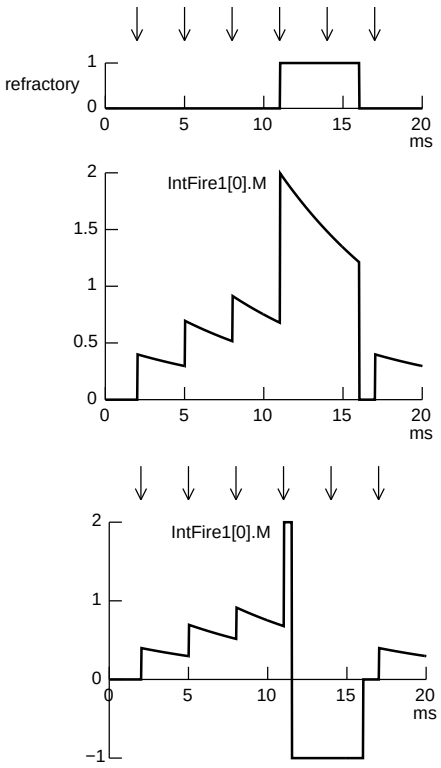

Figure 10.12. Response of an `IntFire1` cell with a 5 ms refractory interval to a run of inputs at 3 ms intervals (arrows), each with weight = 0.4. Top: The cell accepts inputs when `refractory` == 0. The fourth input (at 11 ms) drives the cell above threshold. This triggers an output event, increases `refractory` to 1 (top trace), and the integration state function `M` jumps to 2. During the 5 ms refractory period, `M` decays gradually, but the cell is unresponsive to further inputs (note that the input at 14 ms produces no change in the integration state). At 16 ms `refractory` falls to 0, making the cell once again responsive to inputs, and `M` also returns to 0 until the next input event arrives. Bottom: After modifying the function `M` to generate rectangular pulses that emulate a spike followed by postspike hyperpolarization.

Improved presentation of the integration state

The performance in the top of Fig. 10.12 is satisfactory, but the model could be further improved by one relatively minor change. As it stands the `M` function shows an

exponential decay during the refractory period, which is distracting and irrelevant to the operation of the model at best, and potentially misleading at worst. It would be better for `M` to follow a stereotyped time course, e.g. a brief positive pulse followed by a longer negative pulse. This would not be confused with the subthreshold operation of the model, and it might be more suggestive of an action potential.

The most direct way to do this is to make `M` take different actions depending on whether or not the model is "spiking." One possibility is

```
FUNCTION M() {
  if (refractory == 0) {
    M = m*exp(−(t − t0)/tau)
  } else if (refractory == 1) {
    if (t − t0 < 0.5) {
      M = 2
    } else {
      M = −1
    }
  }
}
```

which is exactly what the built−in `IntFire1` model does. The bottom of Fig. 10.12 shows the time course of this revised function.

This demonstrates how visualization of cell operation can be enhanced by simple calculations of patterns for the spiking and refractory trajectories, without overhead for non−plotted cells. We must emphasize that the simulation calculations are analytic and performed only at event arrival, regardless of esthetic graphical refinements introduced by such plotting routines.

Sending an event to oneself involves very little overhead, yet it allows elaborate calculations to be performed much more efficiently than if they were executed on a per `dt` basis. This is exploited in the implementation of two other built−in integrate and fire mechanisms that offer greater kinetic complexity than `IntFire1`.

Example 10.8: `IntFire2`, firing rate proportional to input

The `IntFire2` model, like `IntFire1`, has a "membrane potential" state *m* that follows first order kinetics with time constant τ_m. However, an input event to `IntFire2` does not affect *m* directly. Instead it produces a discontinuous change in the net synaptic input current *i*. Between events, *i* decays with time constant τ_m toward a steady input current of magnitude i_b. That is,

$$\tau_s \frac{di}{dt} + i = i_b \qquad \text{Eq. 10.4}$$

where an input event causes *i* to change abruptly by *w* (Fig. 10.13 top). This piecewise continuous current *i* drives *m*, i.e.

$$\tau_m \frac{dm}{dt} + m = i \qquad \text{Eq. 10.5}$$

where $\tau_m < \tau_s$. Thus an input event produces a gradual change in *m* that is described by two time constants and approximates an alpha function if $\tau_m \approx \tau_s$. When *m* crosses a threshold of 1 in a positive direction, the cell fires, *m* is reset to 0, and integration resumes immediately, as shown in the bottom of Fig. 10.13. Note that *i* is not reset to 0, i.e. cell firing does not obliterate all traces of prior synaptic activation, as happened with `IntFire1`.

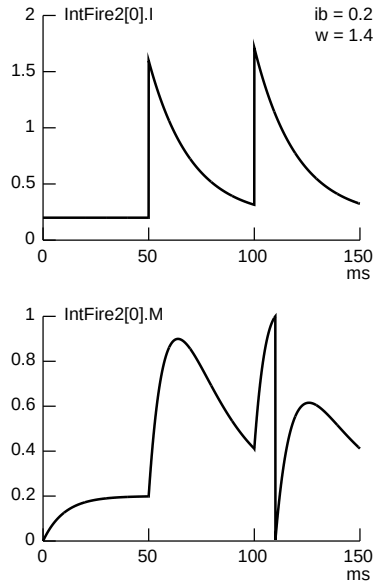

Figure 10.13. Top: Time course of synaptic current *i* in an `IntFire2` cell with τ_s = 20 ms and τ_m = 10 ms. This cell has i_b = 0.2 and receives inputs with weight *w* = 1.4 at *t* = 50 and 100 ms. Bottom: The "membrane potential" *m* of this cell is initially 0 and approaches 0.2 (the numeric value of i_b) with time constant τ_m. The first synaptic input produces a subthreshold response, but temporal summation drives *m* above threshold at t = 109.94 ms. This resets *m* to 0 and integration resumes.

Depending on its parameters, `IntFire2` can emulate a wide range of relationships between input pattern and firing rate. The firing rate is ~ i / τ_m if *i* is >> 1 and changes slowly compared to τ_m.

The i_b current is analogous to the combined effect of a baseline level of synaptic drive plus a bias current injected through an electrode. The requirement that $\tau_m < \tau_s$ is equivalent to asserting that the membrane time constant is faster than the decay of the current produced by an individual synaptic activation. This is plausible for slow inhibitory inputs, but where fast excitatory inputs are concerned an alternative interpretation can be applied: each input event signals an abrupt increase (followed by an exponential decline) in the mean firing rate of one or more afferents that produce brief but temporally overlapping postsynaptic currents. The resulting change of *i* is the moving average of these currents.

The `IntFire2` mechanism is amenable to discrete event simulations because Eqns. 10.4 and 10.5 have analytic solutions. If the last input event was at time t_0 and the values of i and m immediately after that event were $i(t_0)$ and $m(t_0)$, then their subsequent time course is given by

$$i(t) = i_b + [i(t_0) - i_b]\, e^{-(t-t_0)/\tau_s} \qquad \text{Eq. 10.6}$$

and

$$m(t) = i_b + \left[i(t_0) - i_b\right] \frac{\tau_s}{\tau_s - \tau_m} e^{-(t-t_0)/\tau_s} + \left\{ m(t_0) - i_b - \left[i(t_0) - i_b\right] \frac{\tau_s}{\tau_s - \tau_m} \right\} e^{-(t-t_0)/\tau_m} \qquad \text{Eq. 10.7}$$

Implementation in NMODL

The core of the NMODL implementation of `IntFire2` is the function `firetime()`, which is discussed below. This function projects when m will equal 1 based on the present values of i_b, i, and m, assuming that *no new input events arrive*. The value returned by `firetime()` is 10^9 if the cell will never fire with no additional input. Note that if $i_b > 1$ the cell fires spontaneously even if no input events occur.

```
INITIAL {
   . . .
  net_send(firetime(args), 1)
}

NET_RECEIVE (w) {
   . . .
  if (flag == 1) { : time to fire
    net_event(t)
    m = 0
     . . .
    net_send(firetime(args), 1)
  } else {
     . . .
    update m
    if (m >= 1) {
      net_move(t) : the time to fire is now
    } else {
       . . .
      net_move(firetime(args) + t)
    }
  }
  update t0 and i
}
```

Listing 10.6. Key excerpts from `intfire2.mod`

The `INITIAL` block in `IntFire2` calls `firetime()` and uses the returned value to put a self−event into the delivery system. The strategy, which is spelled out in the `NET_RECEIVE` block, is to respond to external events by moving the delivery time of the self−event back and forth with the `net_move()` function. When the self−event is finally delivered (potentially never), `net_event()` is called to signal that this cell is firing. Notice that external events are never ignored−−and shouldn't be even if we introduced a refractory period where we refused to integrate m−−but always have an effect on the value of i.

The function `firetime()` returns the first $t \geq 0$ for which

$$a + b\, e^{-t/\tau_s} + (c - a - b)\, e^{-t/\tau_m} = 1 \qquad \text{Eq. 10.8}$$

where the parameters a, b and c are defined by the coefficients in Eq. 10.7. If there is no such t the function returns 10^9. This represents the time of the next cell firing, relative to the time t_0 of the most recent synaptic event.

Since `firetime()` must be executed on every input event, it is important to minimize the number of Newton iterations needed to calculate the next firing time. For this we use a strategy that depends on the behavior of the function

$$f_1(x) = a + b\, x^r + (c - a - b)\, x \qquad \text{Eq. 10.9a}$$

$$\text{where } \begin{matrix} x = e^{-t/\tau_m} \\ r = \tau_m / \tau_s \end{matrix} \qquad \text{Eq. 10.9b}$$

over the domain $0 < x \leq 1$. Note that $c < 1$ is the value of f_1 at $x = 0$ (i.e. at $t = \infty$). The function f_1 is either linear in x (if $b = 0$) or convex up ($b > 0$) or down ($b < 0$) with no inflection points. Since $r < 1$, f_1 is tangent to the y axis for any nonzero b (i.e. $f_1'(0)$ is infinite).

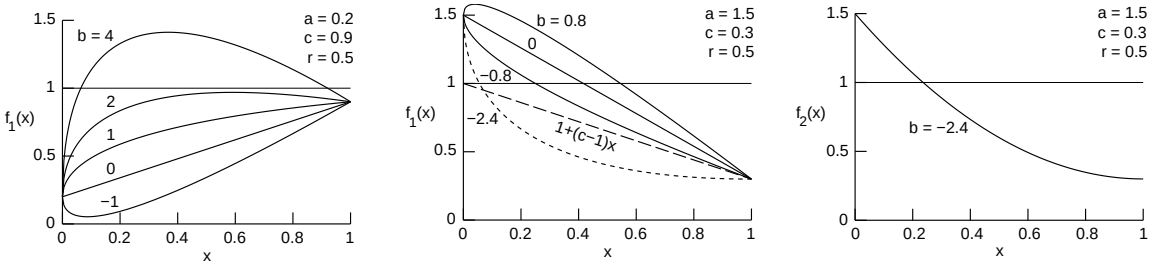

Figure 10.14. Plots of f_1 and f_2 computed for $r = 0.5$. See text for details.

The left panel of Fig. 10.14 illustrates the qualitative behavior of f_1 for $a \leq 1$. It is easy to analytically compute the maximum in order to determine if there is a solution to $f_1(x) = 1$. If a solution exists, f_1 will be concave downward so Newton iterations starting at $x = 1$ will underestimate the firing time.

For $a > 1$, a solution is guaranteed (Fig. 10.14 middle). However, Newton iterations starting at that $x = 1$ are inappropriate if the slope at $x = 1$ is more negative than $c - 1$ (straight dashed line in Fig. 10.14 middle). In that case, the transformation $x = e^{-t/\tau_s}$ is used, giving the function

$$f_2(x) = a + b\ x + (c - a - b)\ x^{1/r} \qquad \text{Eq. 10.9c}$$

and the Newton iterations begin at $x = 0$ (Fig. 10.14 right).

Since iterations are performed over regions in which f_1 and f_2 are relatively linear, the `firetime()` function usually requires only two or three Newton iterations to converge to the next firing time. The only exception is when f_1 has a maximum that is just slightly larger than 1, in which case it may be a good idea to stop after a couple of iterations and issue a self−event. The advantage of this strategy is that it would defer a costly series of iterations, allowing an interval in which another external event might arrive that would force computation of a new projected firing time. Such an event, regardless of whether excitatory or inhibitory, would probably make it easier to compute the next firing time.

Example 10.9: `IntFire4`, different synaptic time constants

`IntFire2` can emulate an input−output relationship with more complex dynamics than `IntFire1` does, but it is somewhat restricted in that the response to any external event, whether excitatory or inhibitory, has the same kinetics. As we pointed out in the discussion of `IntFire2`, it is possible to interpret excitatory events in a way that partially sidesteps this issue. However, experimentally observed synaptic excitation tends to be faster than inhibition (e.g. [Destexhe, 1998 #278]) so a more flexible integrate and fire mechanism is needed.

The `IntFire4` mechanism addresses this need. Its dynamics are specified by four time constants: τ_e for a fast excitatory current, τ_{i1} and τ_{i2} for a slower inhibitory current, and τ_m for the even slower leaky "membrane" which integrates these currents. When the membrane state m reaches 1, the cell "fires," producing an output event and returning m to 0. This does not affect the other states of the model.

The differential equations that govern `IntFire4` are

$$\frac{de}{dt} = -k_e e \qquad \text{Eq. 10.10}$$

$$\frac{di_1}{dt} = -k_{i1} i_1 \qquad \text{Eq. 10.11}$$

$$\frac{di_2}{dt} = -k_{i2} i_2 + a_{i1} i_1 \qquad \text{Eq. 10.12}$$

$$\frac{dm}{dt} = -k_m m + a_e e + a_{i2} i_2 \qquad \text{Eq. 10.13}$$

where each k is a rate constant that equals the reciprocal of the corresponding time constant, and it is assumed that $k_e > k_{i1} > k_{i2} > k_m$. An input event with weight $w > 0$ (i.e. an excitatory event) adds instantaneously to the excitatory current e. Equations 10.11 and 12, which define the inhibitory current i_2, are based on the reaction scheme

$$i_1 \xrightarrow{k_{i1}} i_2 \xrightarrow{k_{i2}} bath \qquad \text{Eq. 10.14}$$

in which an input event with weight $w < 0$ (i.e. an inhibitory event) adds instantaneously to i_1. The constants a_e, a_{i1}, and a_{i2} are chosen to normalize the response of the states e, i_1, i_2, and m to input events (Fig. 10.15). Therefore an input with weight $w_e > 0$ (an "excitatory" input) produces a peak e of w_e and a maximum "membrane potential" m of w_e. Likewise, an input with weight $w_i < 0$ (an "inhibitory" input) produces an inhibitory current i_2 with a minimum of w_i and drives m to a minimum of w_i. Details of the analytic solution to these equations are presented in **Appendix Y**.

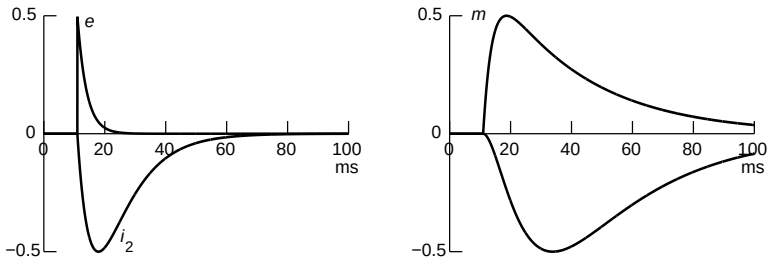

Figure 10.15. Left: Current generated by a single input event with weight 0.5 (e) or −0.5 (i_2). Right: The corresponding response of m. Parameters were $\tau_e = 3$, $\tau_{i1} = 5$, $\tau_{i2} = 10$, and $\tau_m = 30$ ms.

`IntFire4`, like `IntFire2`, finds the next firing time through successive approximation. However, `IntFire2` generally iterates to convergence every time an input event is received, whereas the algorithm used by `IntFire4` exploits the convexity of the membrane potential trajectory so that single Newton iterations alternating with self−events converge to the correct firing time. Specifically, if an event arrives at time t_0, values of $e(t_0)$, $i_1(t_0)$, $i_2(t_0)$, and $m(t_0)$ are calculated analytically. Should $m(t_0)$ be subthreshold, the self−event is moved to a new approximate firing time t_f that is based on the slope approximation to m

$$t_f = t_0 + (1 - m(t_0)) / m'(t_0) \text{ if } m'(t_0) > 0 \qquad \text{Eq. 10.15}$$

or

$$\infty \text{ if } m'(t_0) \le 0$$

(Fig. 10.16 left and middle). If instead $m(t_0)$ reaches threshold, the cell "fires," generating a `net_event` and setting m to 0. The self−event is then moved to an approximate firing time that is computed from Eq. 10.15 using the values assumed by m and m' immediately after the "spike" (Fig. 10.16 right).

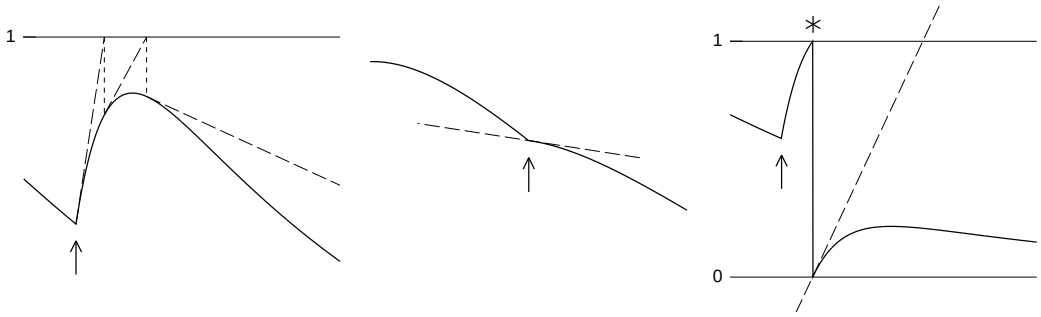

Figure 10.16. Excerpts from simulations of `IntFire4` cells showing time course of m. Arrival of an event (arrow = external event, vertical dotted line = self−event) triggers a Newton iteration. Slanted dashed lines are slope approximations to m immediately after an event. Left: Although Eq. 10.15 yields a finite t_f, this input is too weak for the cell to fire. Middle: Here $m' < 0$ immediately after an input event, so both t_f and the true firing time are infinite. Right: The slope approximation following the excitatory input is not shown, but it obviously crosses threshold before the actual firing time (asterisk). Following the "spike" m is reset to 0 but bounces back up because of persistent excitatory current. This dies away without eliciting a second spike, even though t_f is finite (dashed line).

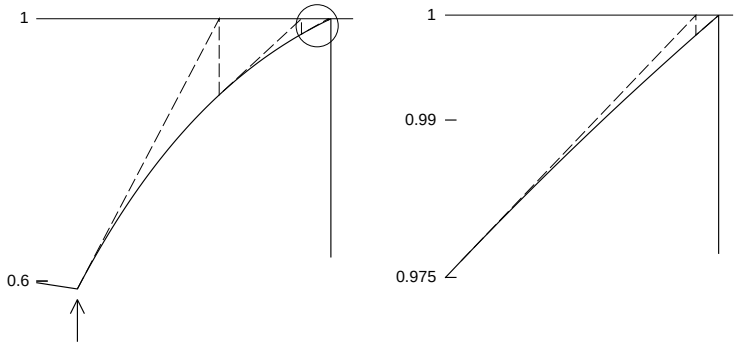

Figure 10.17. These magnified views of the trajectory from the right panel of Fig. 10.16 indicate how rapidly the event−driven Newton iterations converge to the next firing time. In this simulation, spike threshold was reached in four iterations after the excitatory input (arrow). The first two iterations are evident in the left panel, and additional magnification of the circled region reveals the last two iterations (right panel).

The justification for this approach stems from several considerations. The first of these is that t_f is never later than the true firing time. This stipulation, which we prove in **Appendix Y**, is of central importance because the simulation would otherwise be in error.

Another consideration is that successive approximations must converge rapidly to the true firing time, in order to avoid the overhead of a large number of self−events. Using the slope approximation to *m* is equivalent to the Newton method for solving $m(t) = 1$, so convergence is slow only when the maximum value of *m* is close to 1. The code in `IntFire4` guards against missing "real" firings when *m* is asymptotic to 1, because it actually tests for `m > 1 - eps`, where the default value of `eps` is 10^{-6}. Since `eps` is a user−settable `GLOBAL` parameter, one can easily augment or override this protection.

Finally, the use of a series of self−events is superior to carrying out a complete Newton method solution because it is most likely that external input events will arrive in the interval between firing times. Each external event would invalidate the previous computation of firing time and force a recalculation. This might be acceptable for the `IntFire2` mechanism with its efficient convergence, but the complicated dynamics of `IntFire4` suggest that the cost would be too high. How many iterations should be carried out per self−event is an experimental question, since the self−event overhead depends partly on the number of outstanding events in the event queue.

Other comments regarding artificial cells

NEURON's event delivery system has been used to create many more kinds of artificial spiking neurons than the three classes that we have just examined. Specific examples include pacemakers, bursting cells, models with various forms of use−dependent synaptic plasticity, continuous or quantal stochastic variation of synaptic weight, and an "IntFire3" with a bias current and time constants $\tau_m > \tau_i > \tau_e$.

