Schedule of Presentations

NTC	Ted Carnevale
MLH	Michael Hines
GMS	Gordon Shepherd

Morning session

Time	Speaker	Title	Page
9:00 AM	MLH	Welcome	3
9:05	NTC	The basics: creating and using models	5
10:30	Coffee Break		
10:45	NTC	The Channel Builder	49
11:15	MLH	Using NMODL to add new biophysical mechanisms	59
11:45	NTC	The Linear Circuit Builder	67
NOON	Lunch		

Afternoon session

1:00 PM	MLH	Numerical methods: accuracy, stability, speed	75
1:15	NTC	Networks: spike-triggered synaptic transmission, events, and artificial spiking neurons	81
2:30	MLH	Numerical methods: adaptive integration, and events	91
2:45	Coffee Break		
3:00	MLH	Parallelizing network simulations	97
3:30	GMS	Databases for computational neuroscience	107
4:00	MLH	Optimizing models with the Multiple Run Fitter	111
4:45	MLH	Future directions	119
5:00		End of afternoon session	

Appendices start after p. 120.
These are drafts of selected chapters of the NEURON book, as submitted to the publisher in late 2004; as such, they do not include revisions that were made during subsequent proofreading of the book.

We value your opinions and suggestions for improving this course. Please take a moment to fill out and hand in the survey.

Satellite Symposium, Society for Neuroscience

Using the NEURON Simulation Environment

Michael Hines
Ted Carnevale
Gordon Shepherd

Supported by NINDS

The What and the Why of Neural Modeling

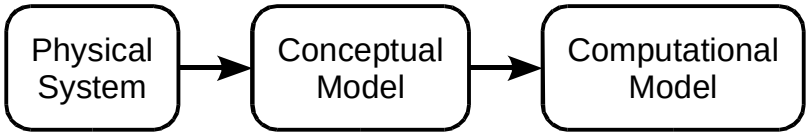
The moment-to-moment processing of information in the nervous system involves the propagation and interaction of electrical and chemical signals that are distributed in space and time.

Empirically-based modeling is needed to test hypotheses about the mechanisms that govern these signals and how nervous system function emerges from the operation of these mechanisms.

Topics

1. How to create and use models of neurons and networks of neurons
   - How to specify anatomical and biophysical properties
   - How to control, display, and analyze models and simulation results
2. How NEURON works
3. How to add user-defined biophysical mechanisms

From Physical System to Computational Model



Conceptual model

a simplified representation of the physical system

Computational model

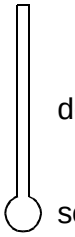
an accurate representation of the conceptual model

From Physical System to Computational Model

Physical system

Ca1 pyramidal cell

Conceptual model



ball and stick

Computational model

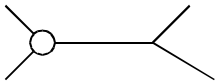
```
create soma, dendrite
connect dendrite(0), soma(1)
```

hoc code

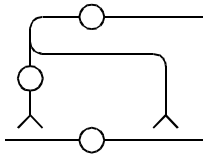
Hierarchies of Complexity

Structure

Single compartment 

Stylized 

Anatomically detailed 

Network 

Hierarchies of Complexity

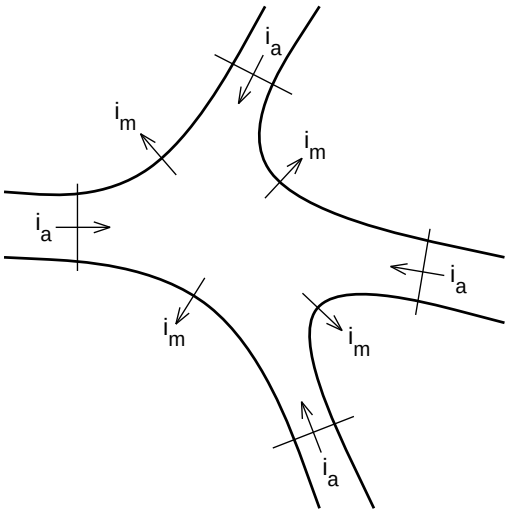
Mechanism

- Passive and Active currents
  - HH-style
  - kinetic scheme
- Synaptic transmission
  - continuous
  - spike-triggered
- Gap junctions
- Extracellular fields, Linear circuits
- Diffusion, buffers, transport & exchange
- Artificial spiking cells ("integrate & fire")

Fundamental Concepts in NEURON

Signals	Flux	Driving force	What is conserved
Electrical	current	voltage gradient	charge
Chemical	solute	concentration gradient	mass

Conservation of Charge



$$C_m \frac{dV_m}{dt} + i_{ion} = \sum i_a$$

The Model Equations

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

v_j membrane potential in compartment j

i_{ion_j} net transmembrane ionic current in compartment j

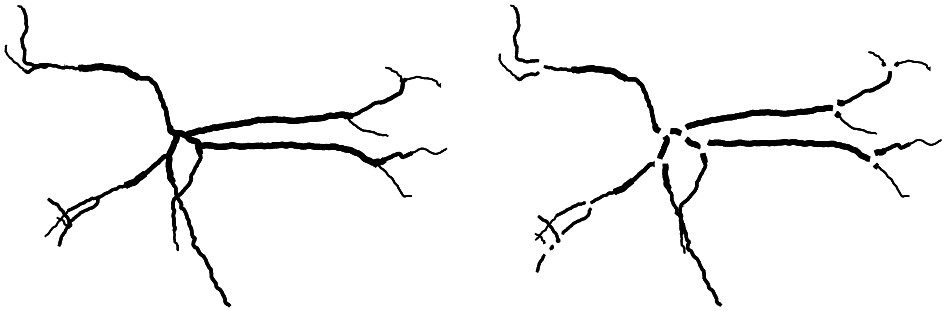
c_j membrane capacitance of compartment j

r_{jk} axial resistance between the centers of
compartment j
and
adjacent compartment k

Separating Anatomy and Biophysics from Purely Numerical Issues

section

a continuous length of unbranched cable



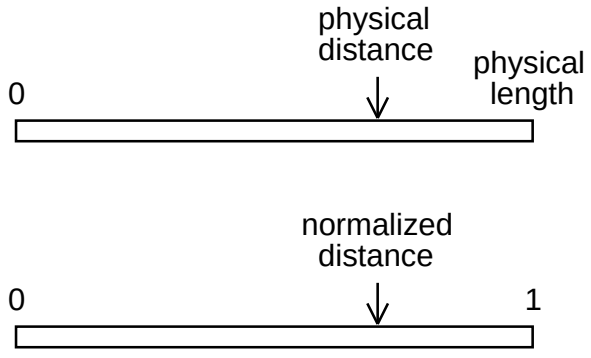
Anatomical data from A.I. Gulyás

Range Variables

Name	Meaning	Units
`diam`	diameter	[µm]
`cm`	specific membrane capacitance	[µf/cm^2]
`g_pas`	specific conductance of the `pas` mechanism	[siemens/cm^2]
`v`	membrane potential	[mV]

range

- normalized position along the length of a section
- $0 \leq \text{range} \leq 1$
- any variable name can be used for range, e.g. `x`



Syntax:

`sectionname.rangevar(range)`

returns or sets the value of rangevar at the location corresponding to range

Examples:

`dend.v(0.5)`

returns membrane potential at the middle of `dend`

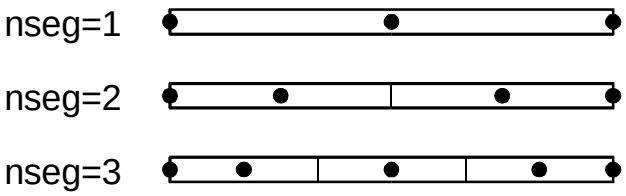
Shortcut: `dend.v`

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

prints physical distance and `v` at each point in `dend` where `v` was calculated

nseg

the number of points in a section section where membrane current and potential are computed



Example: `axon nseg = 3`

To test spatial resolution

```
forall nseg = nseg*3
```

and repeat the simulation

Units

Category	Variable	Units
Time	`t`	[ms]
Voltage	`v`	[mV]
Current	`i`	[mA/cm^2] (distributed) [nA] (point process)
Concentration	`nai` etc.	[mM]
Specific capacitance	`cm`	[µf/cm^2] (distributed) [µS] (point process)
Length	`diam, L`	[µm]
Conductance	`g`	[S/cm^2] (distributed) [µS] (point process)
Cytoplasmic resistivity	`Ra`	[Ω cm]
Resistance	`ri`	[10^6 Ω]

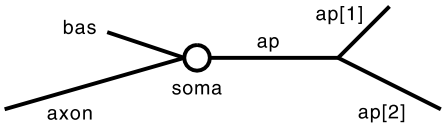
Construction and Use of Models

1. Specify the model ("virtual organism").
2. Specify the user interface ("virtual lab rig").
3. Tests
   - structural integrity
   - spatial grid
   - time steps

Example: using the GUI to build and exercise a stylized model

1. How to use the CellBuilder to create and manage a model cell.
2. How to use NEURON's graphical tools to make an interface for running simulations.

Step 1: using the CellBuilder to make a stylized model



Section	L	diam	Biophysics
soma	20 µm	20 µm	hh
ap[0]	400	2	reduced hh *
ap[1]	300	1	reduced hh *
ap[2]	500	1	reduced hh *
bas	200	3	pas §
axon	800	1	hh

* - gnabar_hh and gkbar_hh reduced to 10%, el_hh = - 64 mV

§ - e_pas = - 65 mV

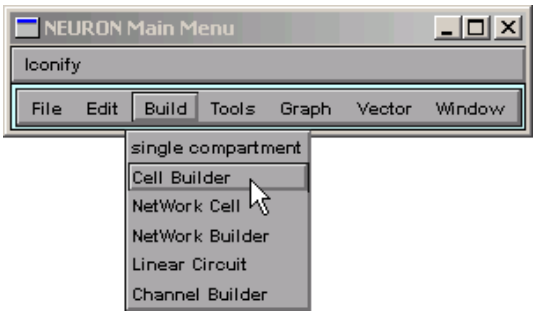
Throughout the cell Ra = 160 Ω cm, cm = 1 µf / cm^2

Launch NEURON with its library of graphical tools

UNIX/Linux `nrngui`

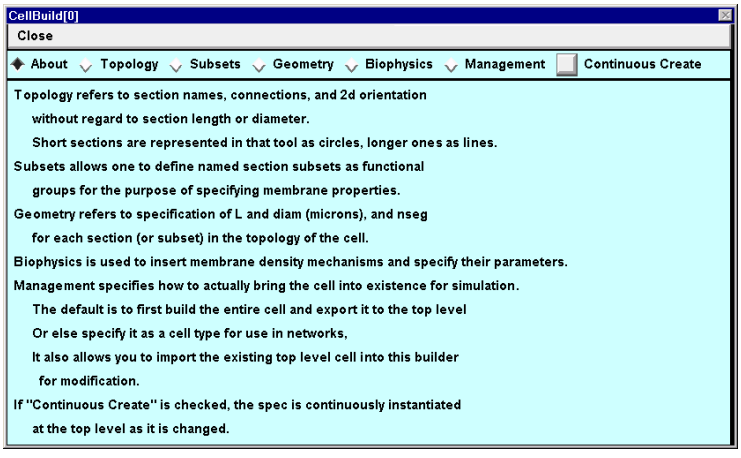
MSWin or OS X 

Bring up a CellBuilder



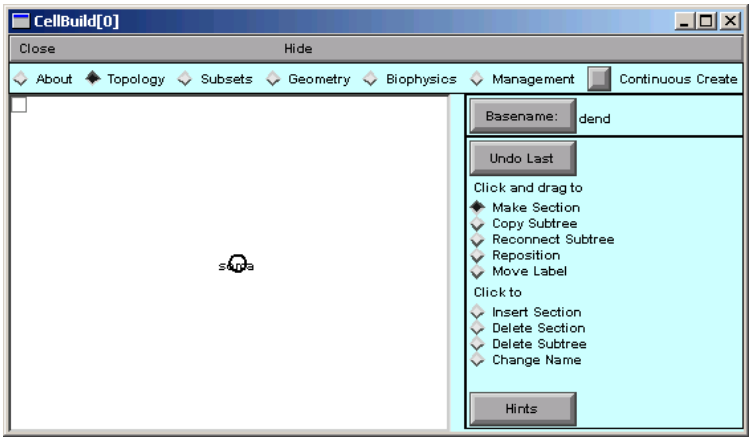
NEURON Main Menu / Build / Cell Builder

The CellBuilder



Use buttons from left to right.

Topology



CB starts with a "soma" section.

We want to create new sections.

Specifying the "Basename"

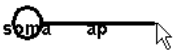
Making a new section

Place cursor near end of existing section

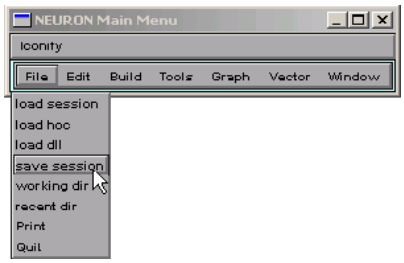
Click to start new section

Drag to desired length

Release mouse button

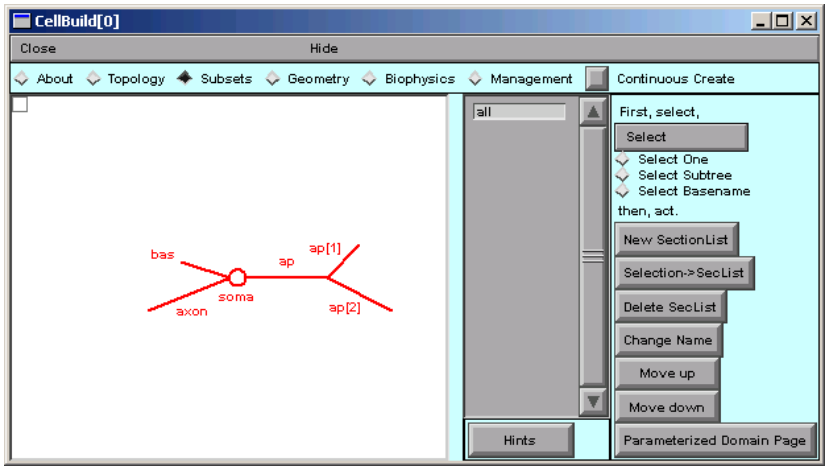


Save your work as you make progress!



NEURON Main Menu / File / save session

Subsets

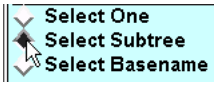


"all" includes all sections.

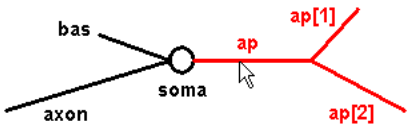
We want to make an "apicals" subset.

Making a new subset

Click "Select Subtree"

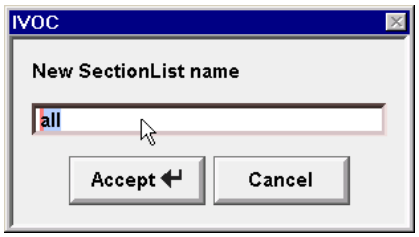


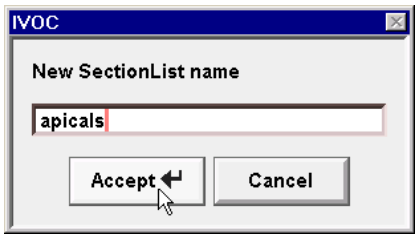
Click root of apical tree . . .



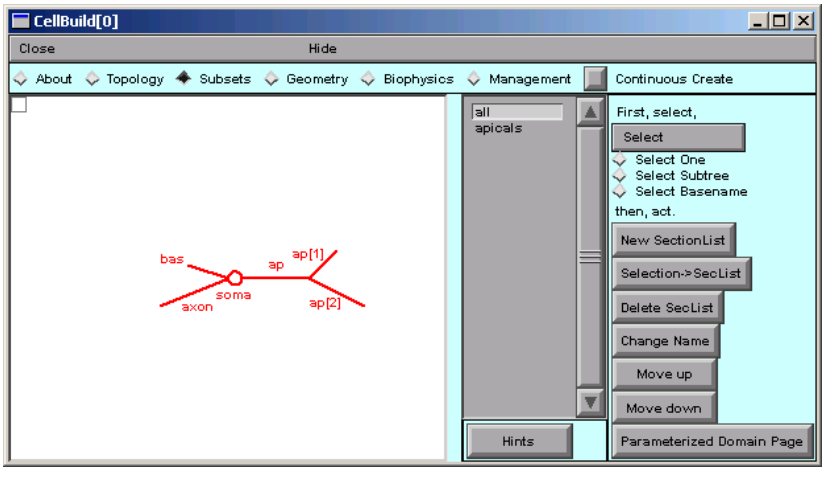
. . . then "New SectionList"

Making a new subset *continued*





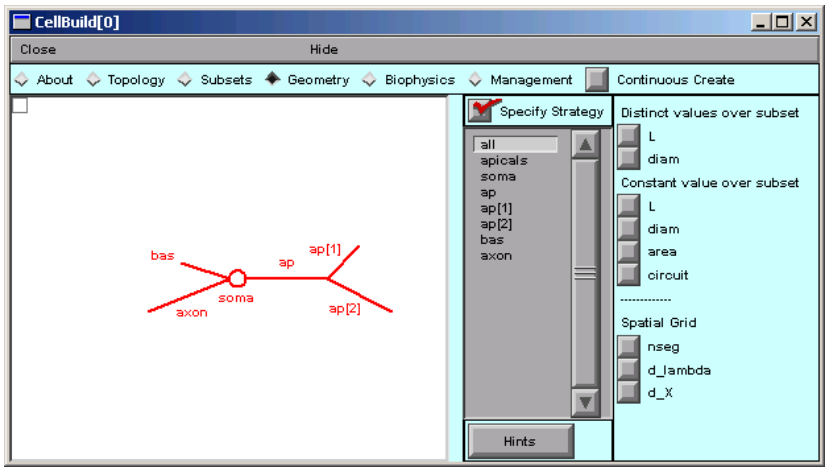
Subsets finished



Note "apicals".

Time to save a new session file.

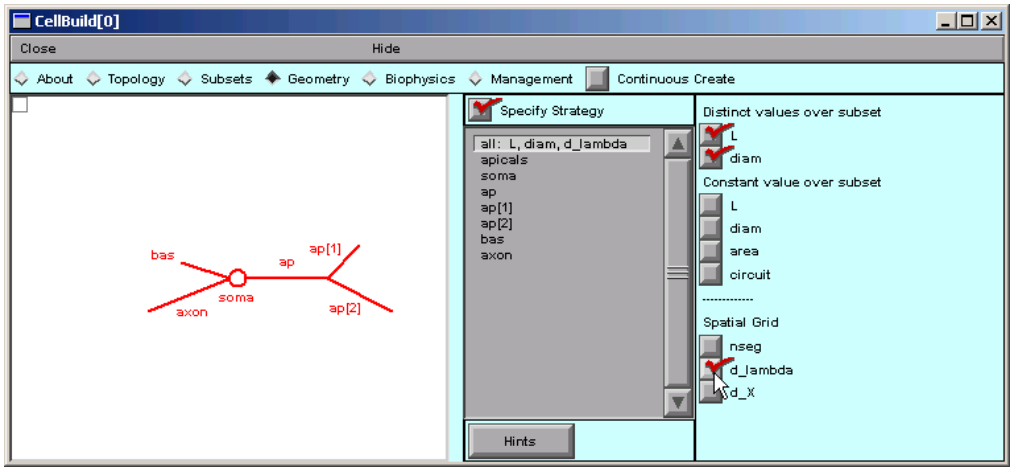
Geometry



"Specify Strategy" is ON.

Make an efficient plan.

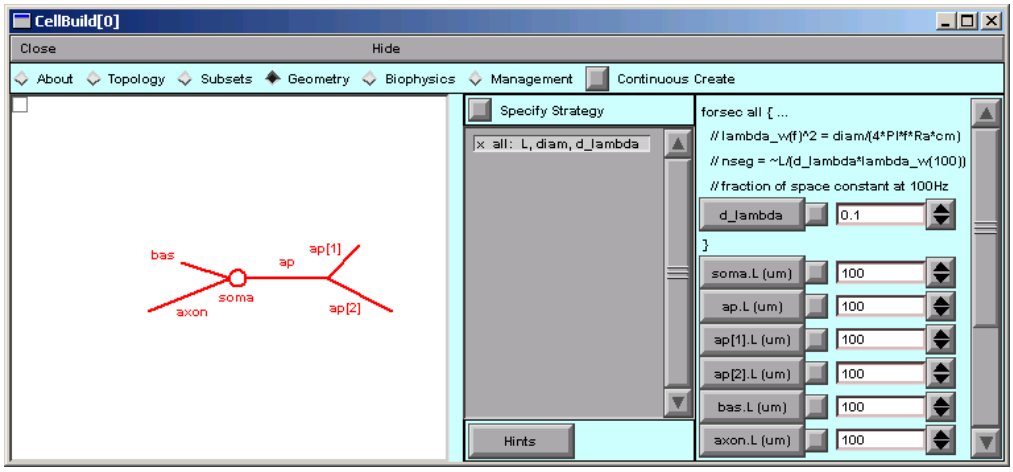
Geometry strategy



Each section has a different L and diam.

Use d_lambda rule to automatically adjust nseg.

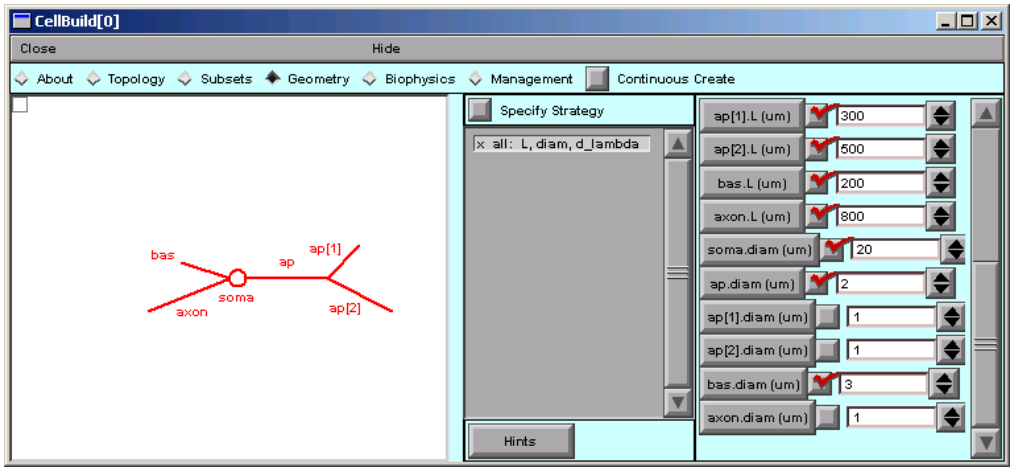
Implementing geometry strategy



When strategy is complete, turn "Specify Strategy" OFF and start assigning values to parameters.

0.1 at 100 Hz is generally OK for d_lambda.

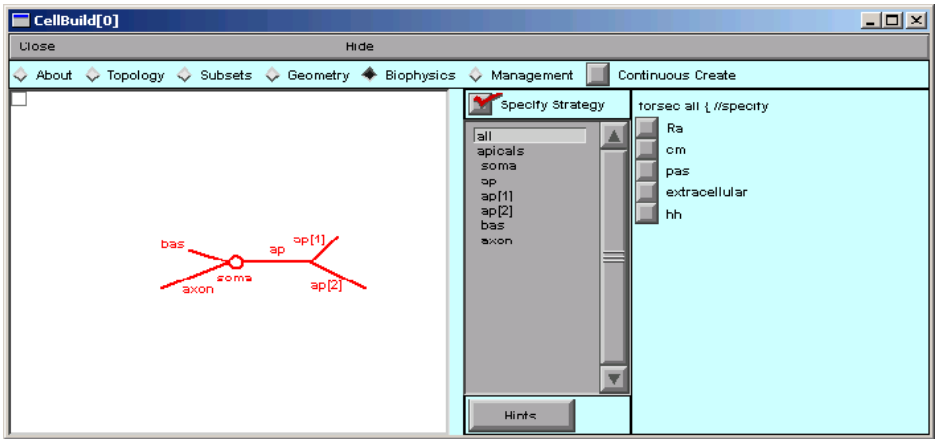
Implementing geometry *continued*



Set L and diam for all sections.

Time to save to a session file!

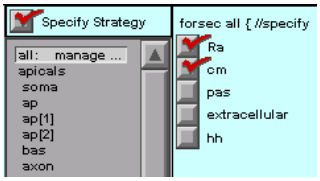
Biophysics



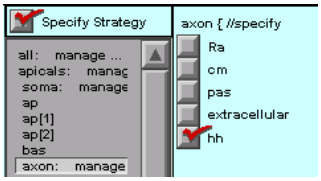
"Specify Strategy" is ON.
Make another efficient plan.

Biophysics strategy

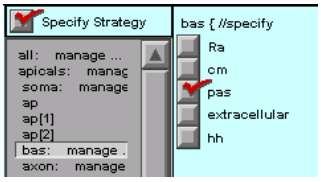
Ra and cm are homogeneous



apicals, soma and axon have hh

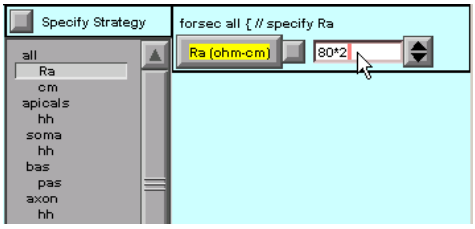


bas has pas

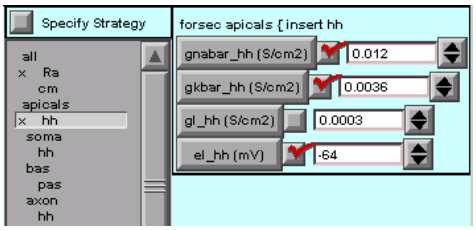


Implementing biophysics strategy

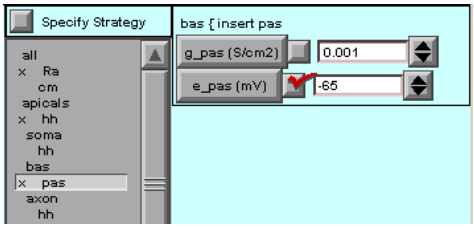
Double Ra



Fix apicals hh params



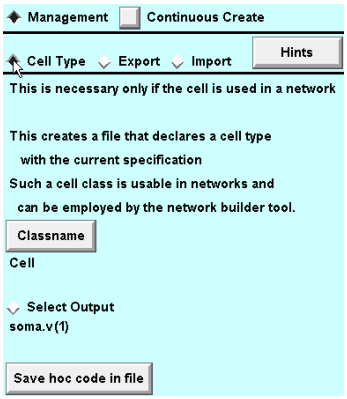
Shift e_pas in bas



Save another session file!!

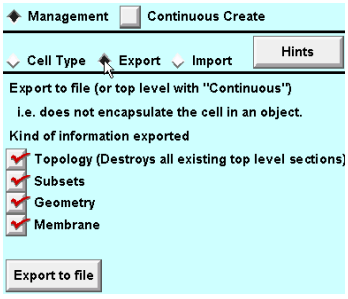
Management

Option 1: save as a Cell Type
for use in a network



Management *continued*

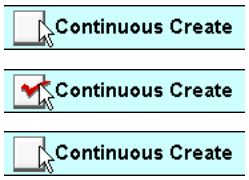
Option 2: save as hoc file



Management *continued*

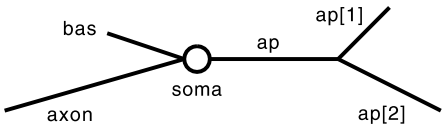
Option 3: export to interpreter

Toggle Continuous Create ON and OFF



or just leave it ON all the time.

Step 2: creating and using an interface for running simulations



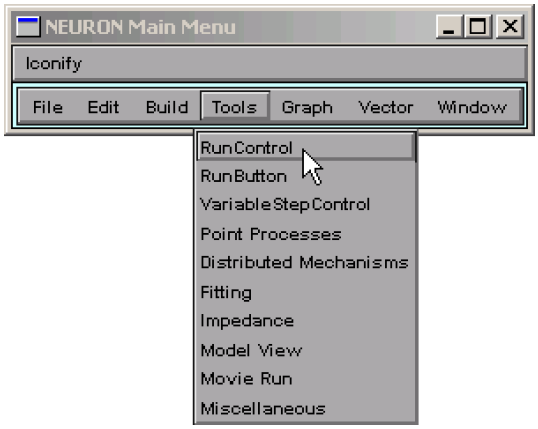
We want to

- attach a stimulating electrode
- evoke an action potential
- show time course of Vm at soma
- show Vm along a path from one end of the cell to the other

We need

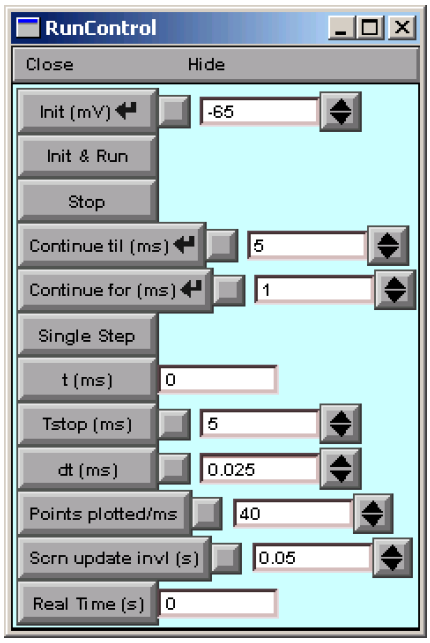
- a "Run" button
- graphs to plot results
- a stimulator

Get a "Run" button



NEURON Main Menu / Tools / RunControl

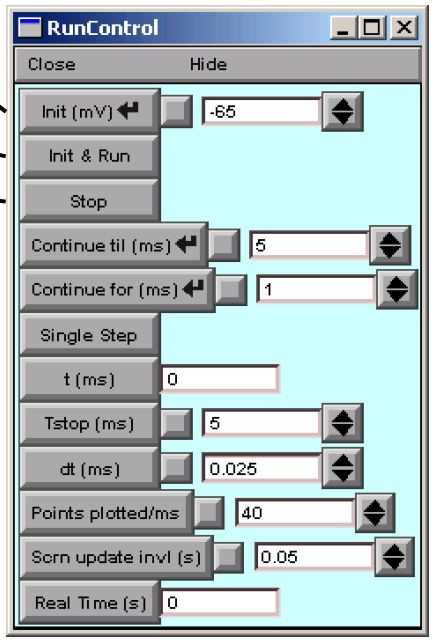
RunControl panel



Init sets time to 0,
Vm to displayed value, and
conductances to steady-state

Init & Run does an Init,
then starts a simulation

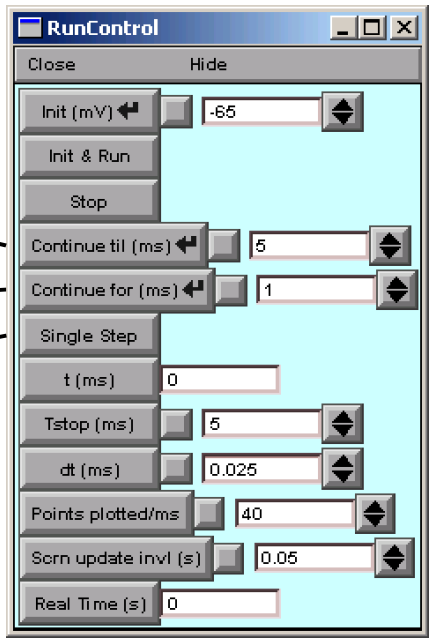
Stop interrupts the simulation



Continue til runs until displayed time

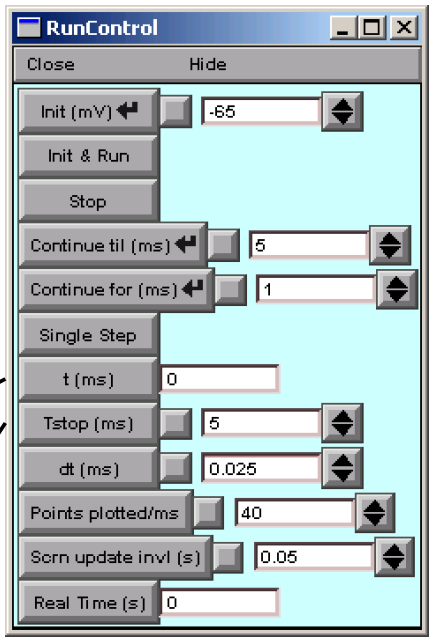
Continue for runs for displayed interval

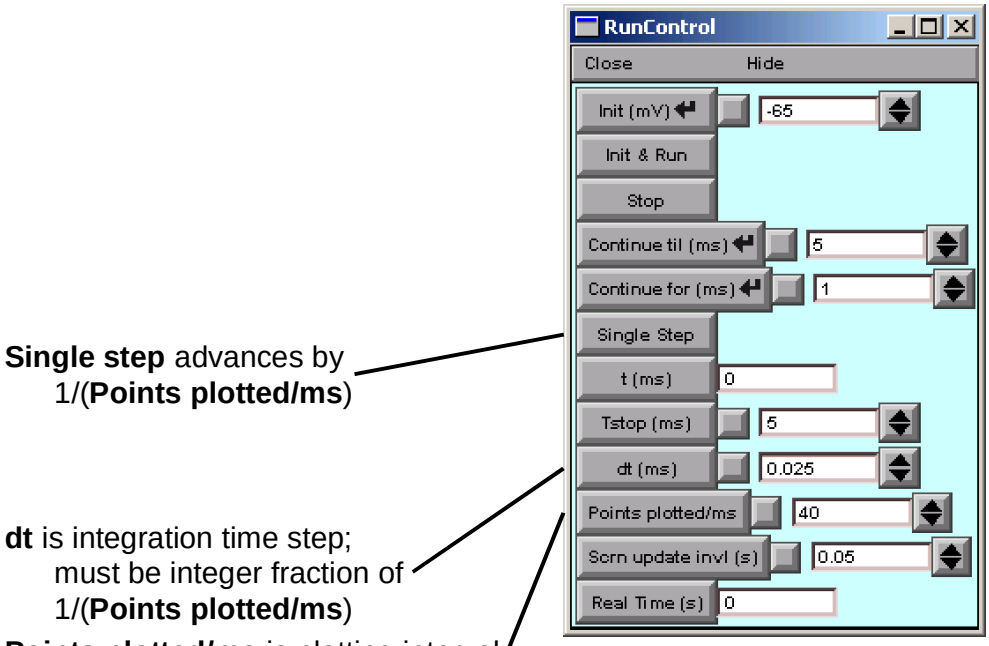
Single step advances by 1/(**Points plotted/ms**)



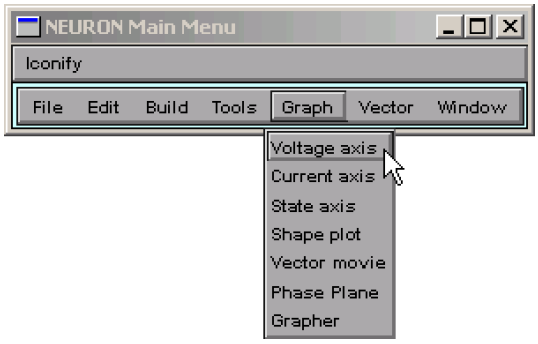
t numeric field shows model time

Tstop specifies when simulation ends



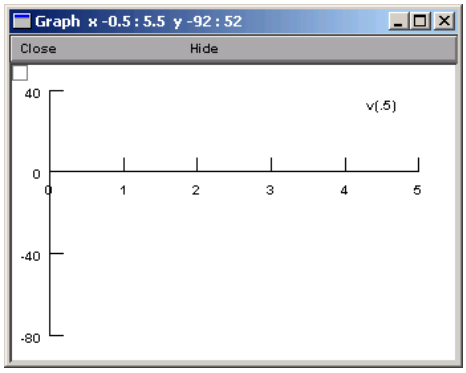


We need to plot Vm(t) at soma



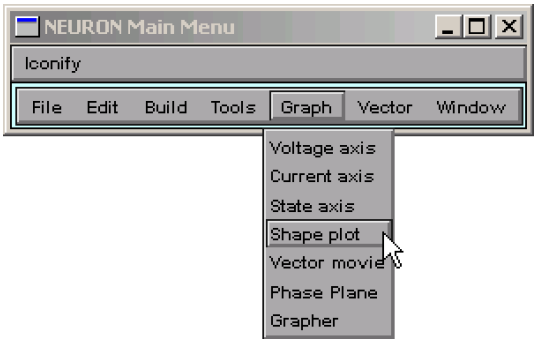
NEURON Main Menu / Graph / Voltage axis

Graph window



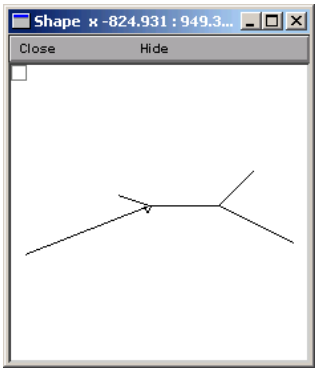
v(.5) is Vm at middle of default section (soma in this example)

We need to plot Vm along a path



NEURON Main Menu / Graph / Shape plot

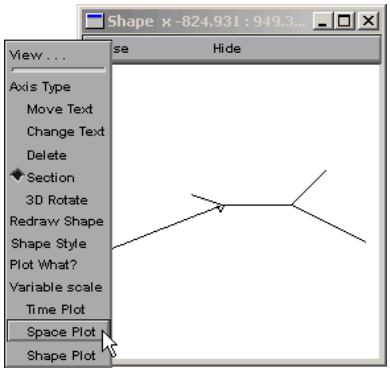
Bringing up a space plot



Use this "shape plot" to create a "space plot".

Click on its "menu box" . . .

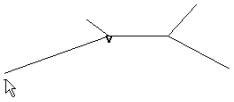
Bringing up a space plot *continued*



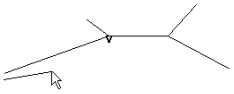
. . . and scroll down to "Space Plot".

Bringing up a space plot *continued*

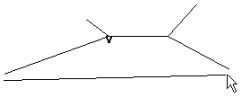
Click just left of the shape



Hold button down while dragging from left . . .



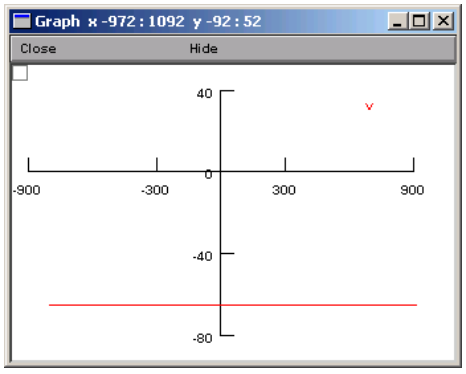
. . . to right . . .



. . . then release button.

This pops up a . . .

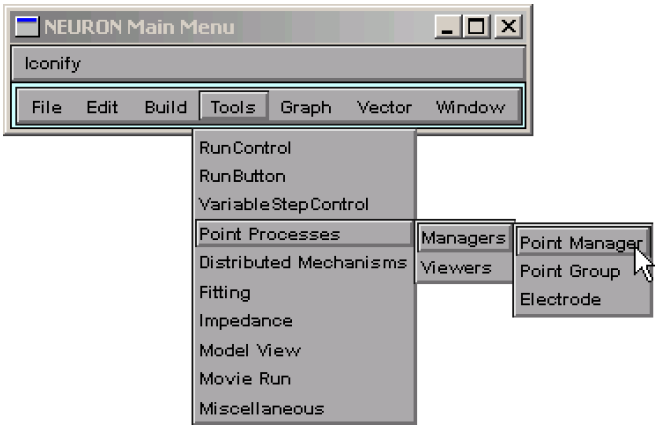
Space plot



A plot of Vm vs. distance along a path.

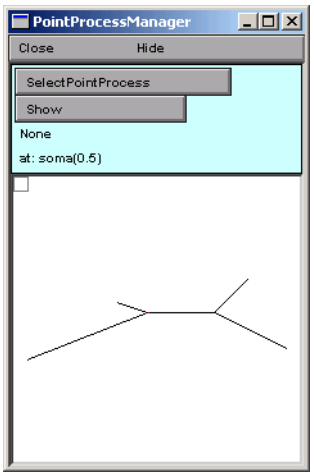
Better save a session file.

We need a stimulator



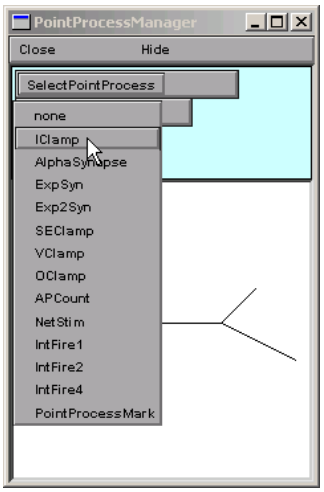
NEURON Main Menu / Tools / Point Processes / Managers / Point Manager

PointProcessManager window



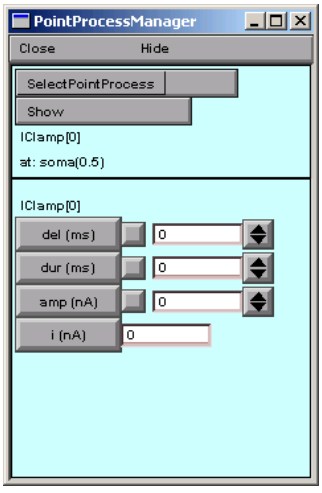
To make this an IClamp . . .

Creating an IClamp



. . . click on SelectPointProcess and scroll down to IClamp.

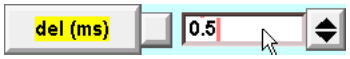
IClamp parameter panel



Next: set parameter values.

Entering values into numeric fields

Direct entry

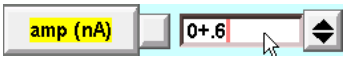


Note yellow highlight on button

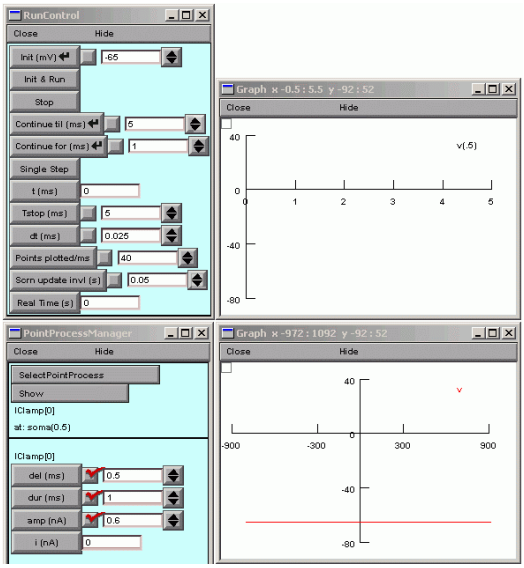
Spinner

Red check means value has been changed from default

Mathematical expression

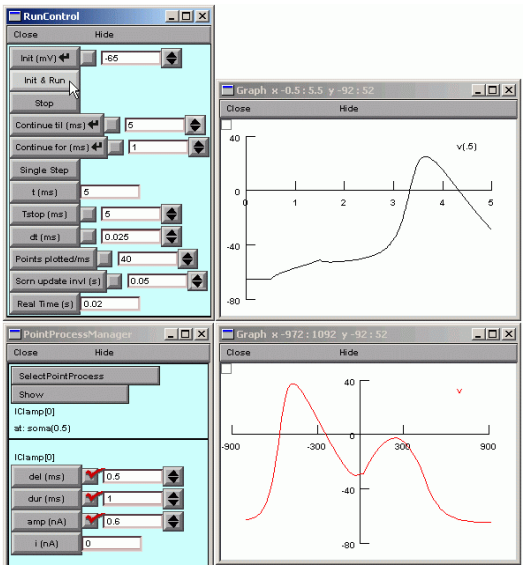


Our user interface

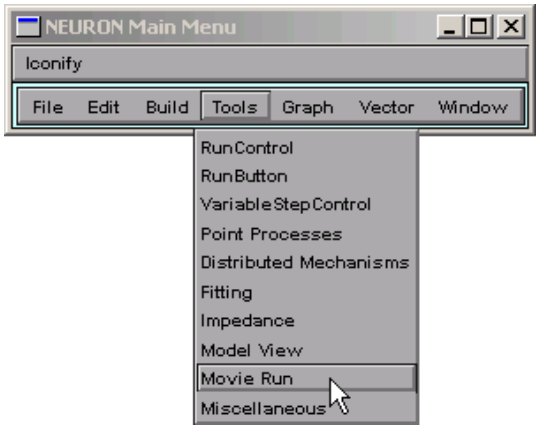


Time to save to a new session file!

It works!

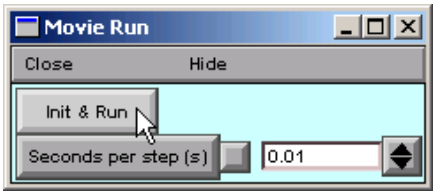


How to get nice space plot "movies"



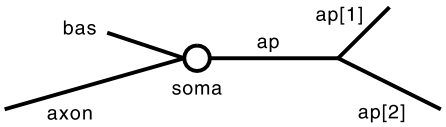
NEURON Main Menu / Tools / Movie Run

Space plot "movies" *continued*



Movie Run / Init & Run

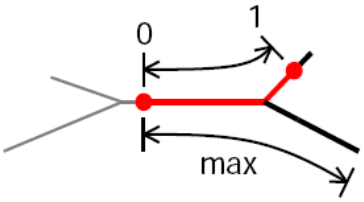
What if hh is nonuniform over the apicals?



Suppose gnabar_hh, gkbar_hh, and el_hh all derease linearly with distance from the origin of the apical tree?

Details:

1. All have full density at origin of apical tree.
2. Density falls to 0% at most the most distant termination.
3. For uniform -65 mV resting potential, e_pas = -54.3 mV.



This example:

gnabar_hh = 0.12 * (1 - p) where p = L_{01}/L_{max}
(normalized path distance from origin of apical tree)

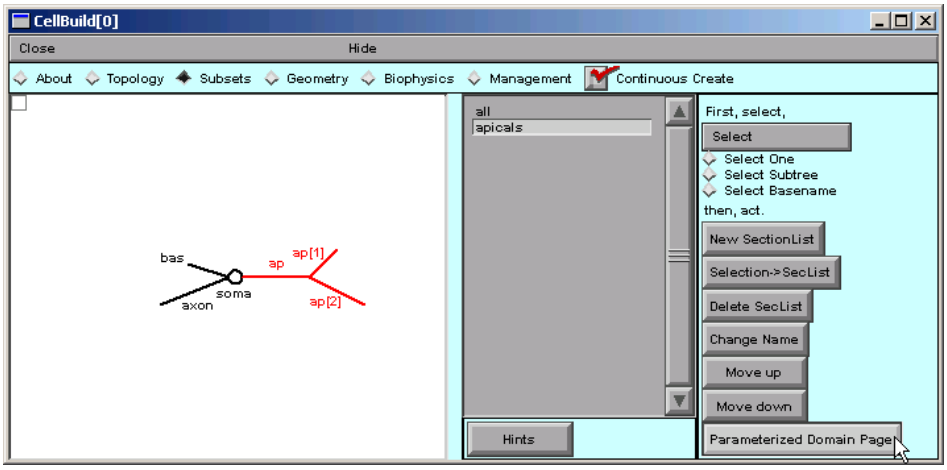
The general task: param = f(p), where p is one of the following:

- path length from a reference point
- radial distance from a reference point
- distance from a plane ("3D projection onto a line")

An equivalent hoc idiom:

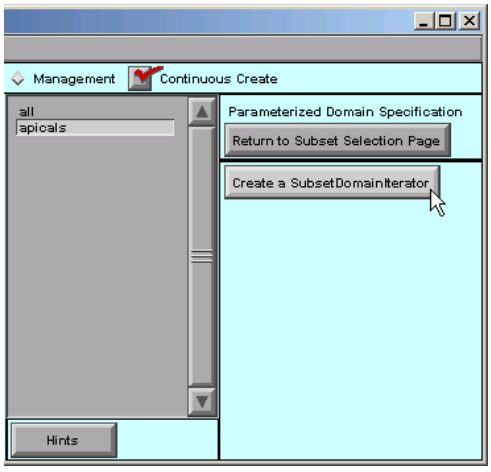
forsec subset for (x,0) { rangevar_suffix(x) = f(p(x)) }

Setting up a SubsetDomainIterator



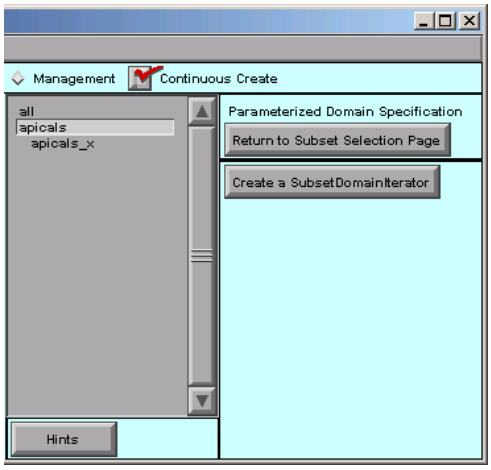
Select a subset, then click on "Parameterized Domain Page"

SubsetDomainIterator *continued*



Click on "Create a SubsetDomainIterator"

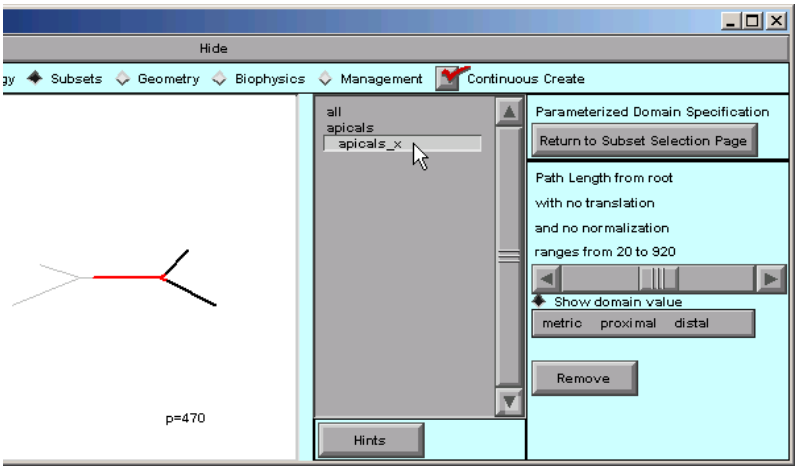
SubsetDomainIterator *continued*



Note "apicals_x" in middle panel.

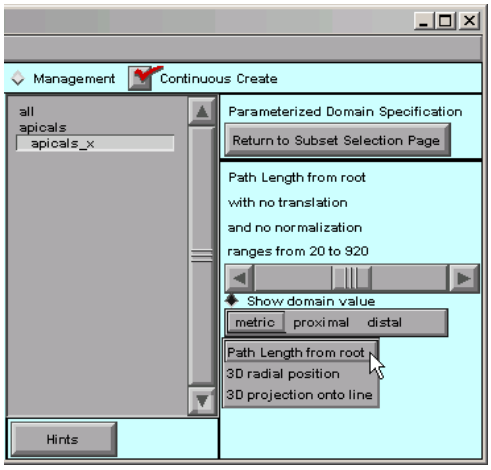
Click on it . . .

SubsetDomainIterator *continued*



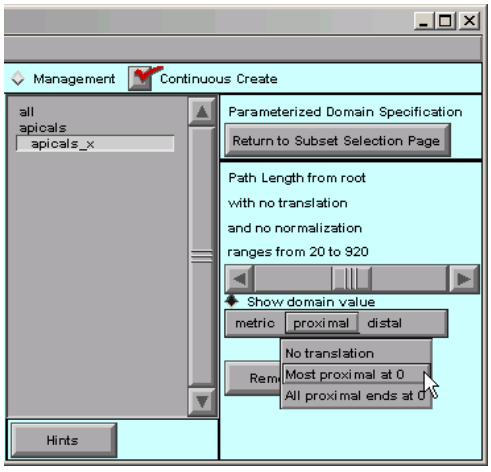
. . . to see controls for specifying the distance metric.

SubsetDomainIterator *continued*



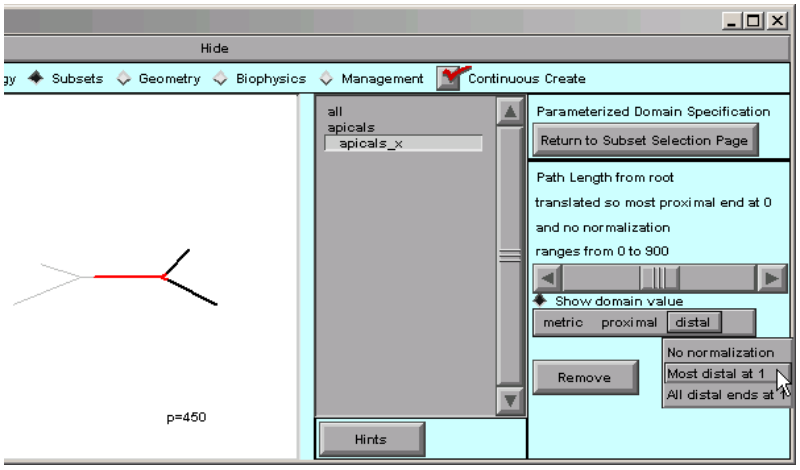
"metric" offers the three basic choices

SubsetDomainIterator *continued*



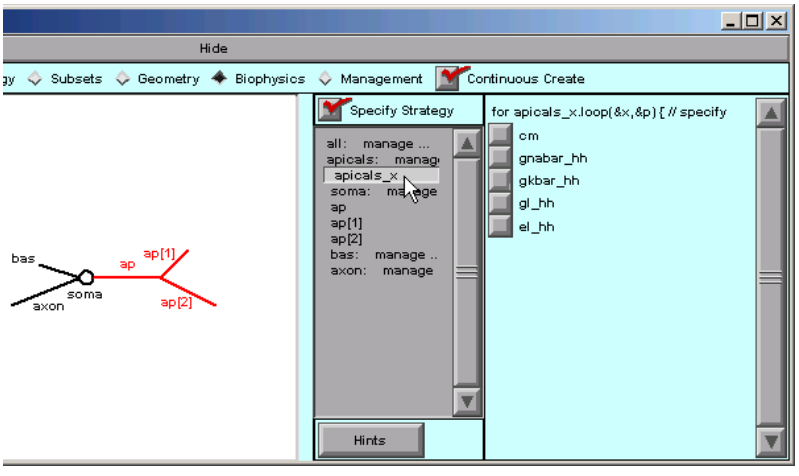
proximal / Most proximal at 0
makes distance start at root of apical tree

SubsetDomainIterator *continued*



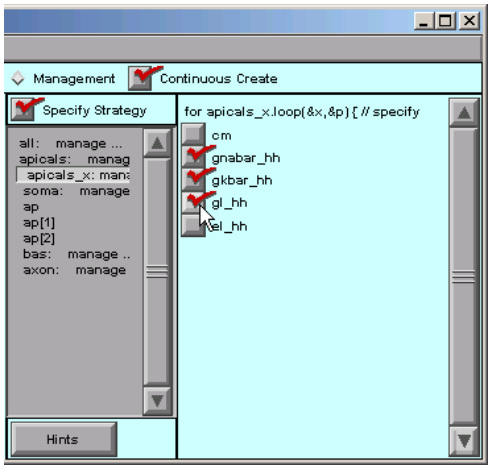
distal / Most distal at 1
finishes "normalization" of distance

Back to Biophysics Strategy



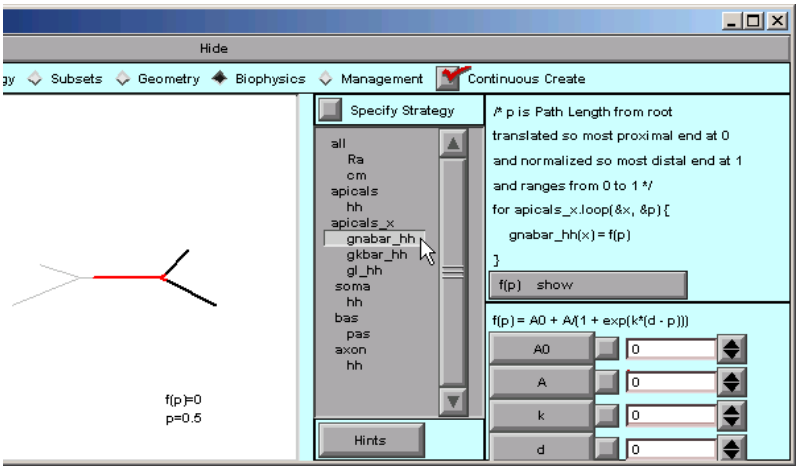
Click on apicals_x,
then select the parameters it will control.

Biophysics Strategy *continued*



We want gnabar_hh, gkbar_hh, and gl_hh to be inhomogeneous.

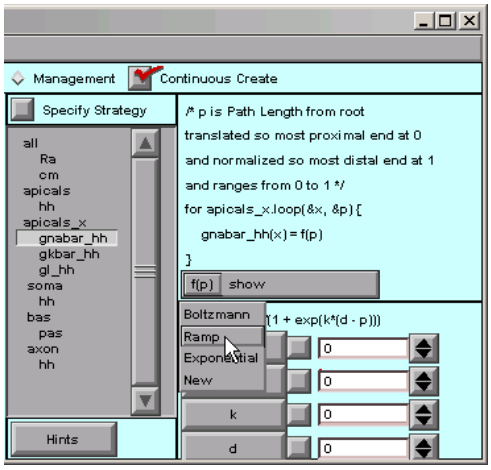
Implement the strategy



Click on one of the inhomogeneous parameters.

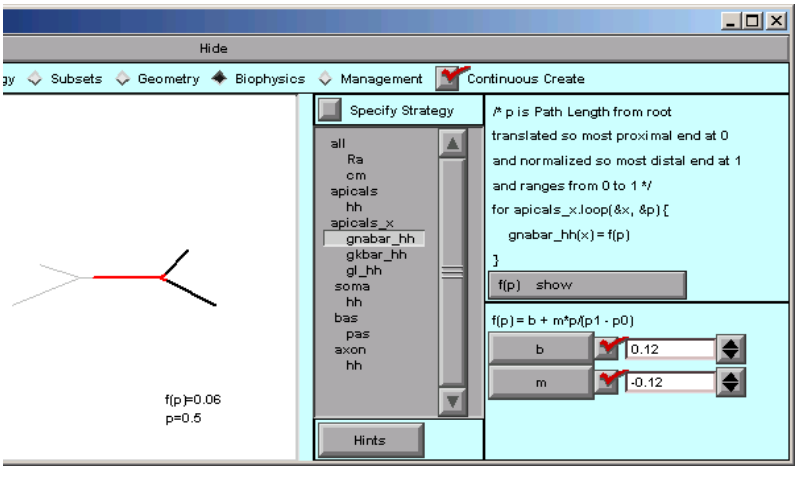
Note that default f( ) is Boltzmann.

Implement the strategy *continued*



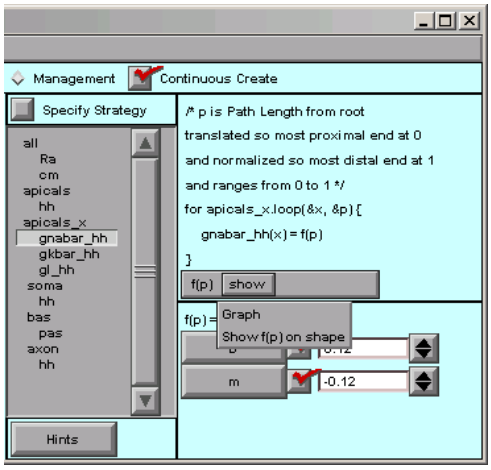
f(p) / Ramp
selects linear function

Implement the strategy *continued*



After setting intercept b and slope m for gnabar_hh

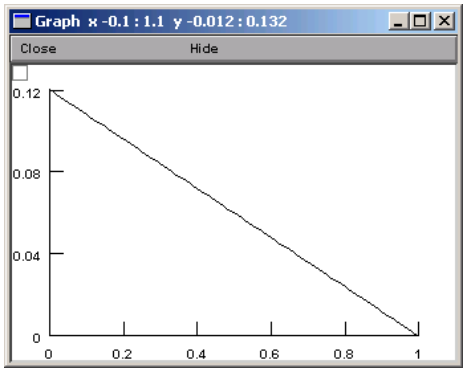
Verify the implementation



show / graph

show / Show f(p) on shape

Verify the implementation *continued*



"show / graph" results:

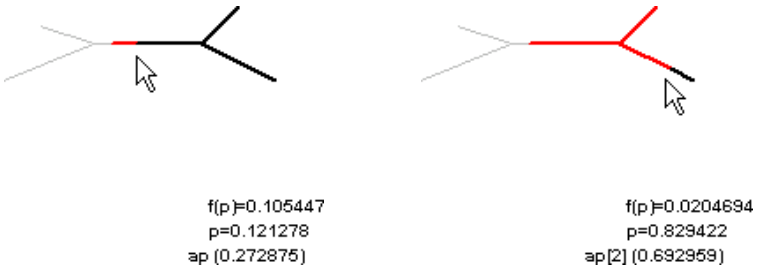
x axis: normalized distance from origin of apicals

y axis: gnabar_hh

Verify the implementation *continued*

1. show / Show f(p) on shape
2. Click next to shape and drag . . .

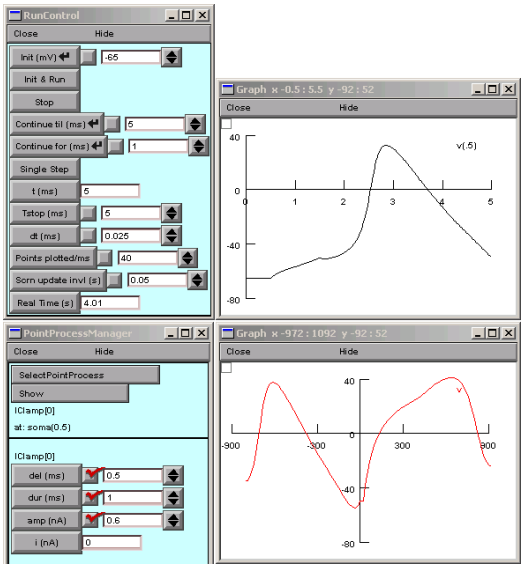
. . . from left . . . . . . to right . . .



. . . while watching the values of p and f(p)

Save another session file!!

A simulation with the revised model



The Channel Builder

Voltage- and ligand-gated channels

Kinetic schemes, HH-style differential equations

Optional stochastic gating mode for point processes

Faster than equivalent NMODL mechanisms

Much easier to use than writing NMODL code

Limited to channels

- NMODL needed for pumps, buffers, diffusion, event-driven synaptic mechanisms, artificial spiking cells

Tutorial: see Documentation at NEURON's home page
http://www.neuron.yale.edu/

Implementing the HH i_{Na} with the Channel Builder

$i_{Na} = g_{Na} (V - E_{Na})$ where

$g_{Na} = gbar_{Na}\ m^3h$

$gbar_{Na} = 0.12\ S/cm^2$

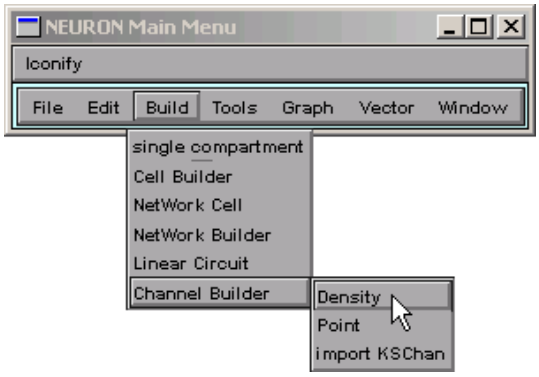
m and h are described by DEs of the form

$dx/dt = a_x (1 - x) - b_x x$

General strategy

1. Bring up a Channel Builder
2. Specify channel's basic properties
3. Specify channel gating
   - states
   - transitions (if a kinetic scheme)
   - effects of voltage and ligands

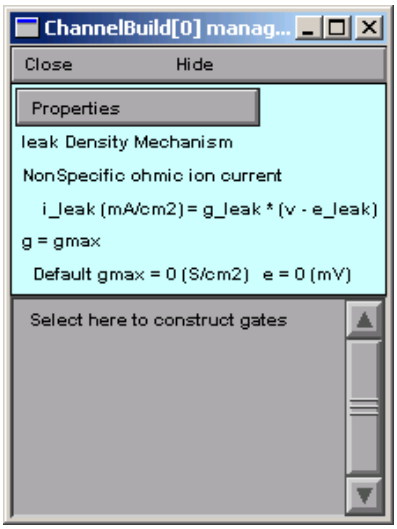
1. Bring up a Channel Builder



NEURON Main Menu / Build
/ Channel Builder / Density

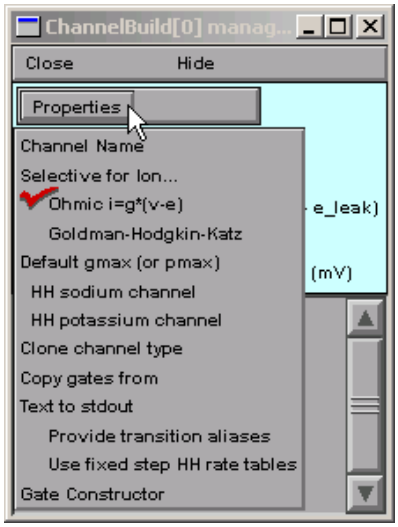
The Channel Builder

We need to change its name,
ion,
default conductance,
and equilibrium potential



2. Specify channel's basic properties

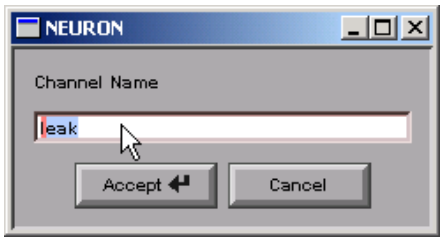
Click on Properties,
then select item to change

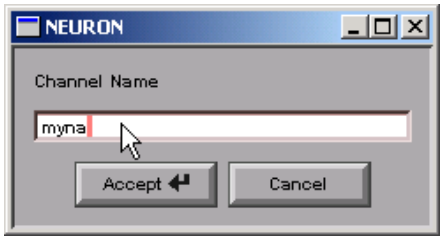


Name

Properties / Channel Name

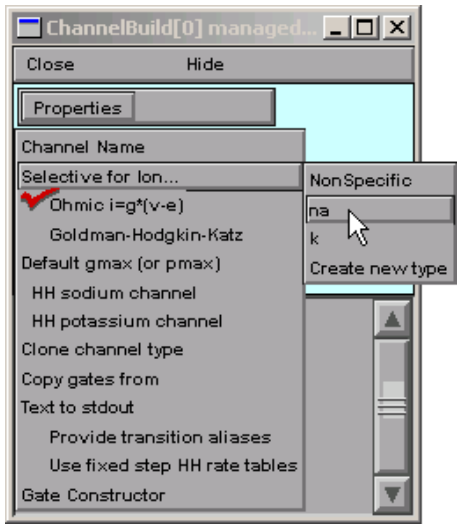
Then change leak to myna





Ion

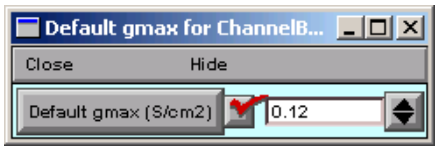
Properties
/ Selective for Ion... / na



Default conductance and equilibrium potential

Properties / Default gmax

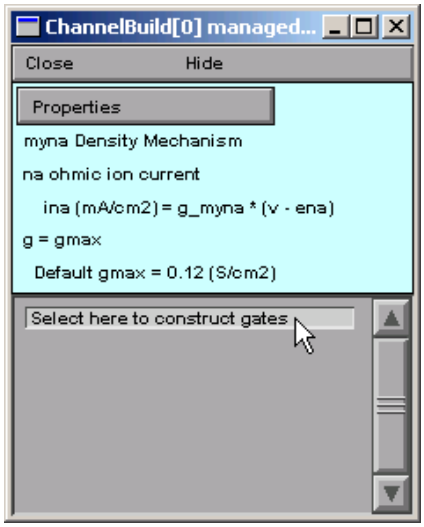
Specify 0.12 S/cm2



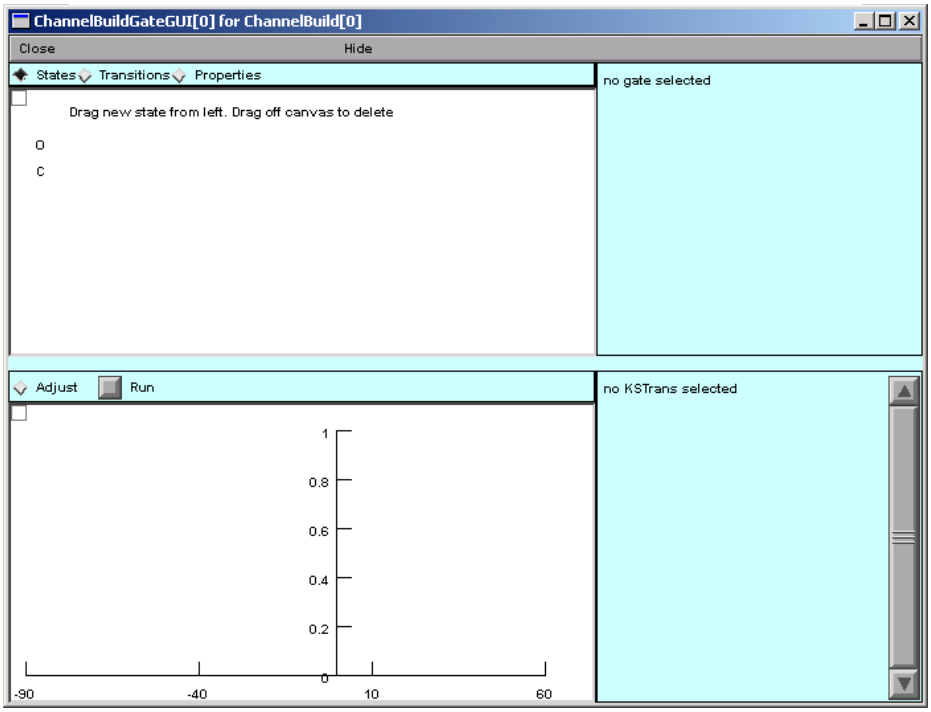
Equilibrium potential:
na has its own ena,
so nothing to do!

3. Specify channel gating

"Select here to construct gates"

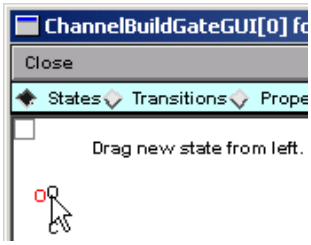


"GateGUI": States page



Spawn states

Click and drag O ("open") from palette . . .



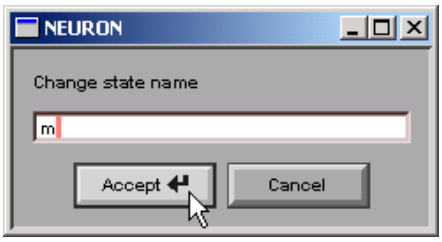
. . . to canvas.

Repeat for C ("closed")

Rename states

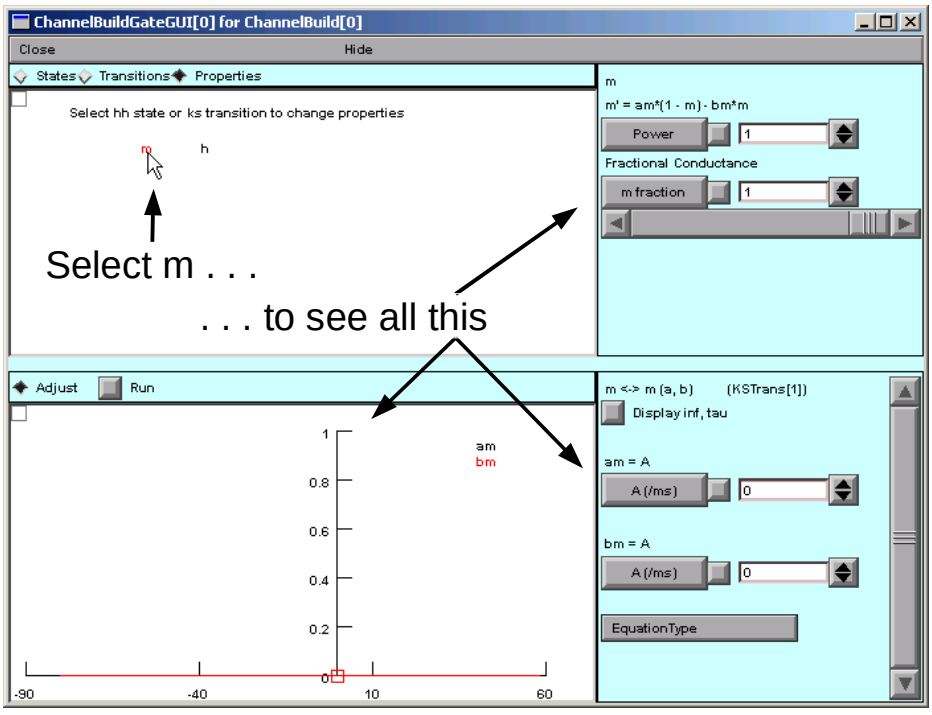
Click O without dragging

Change to m



Change C to h

"GateGUI": Properties page

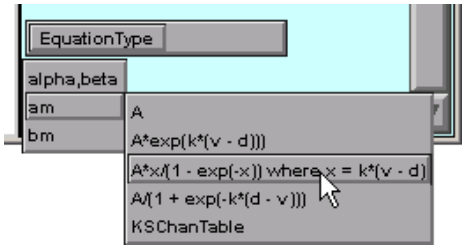


Set m exponent

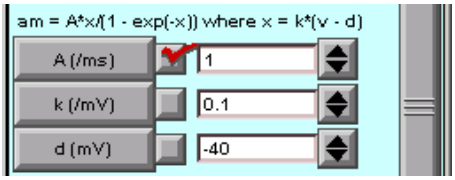
Change Power to 3

Specify voltage dependence of am and bm

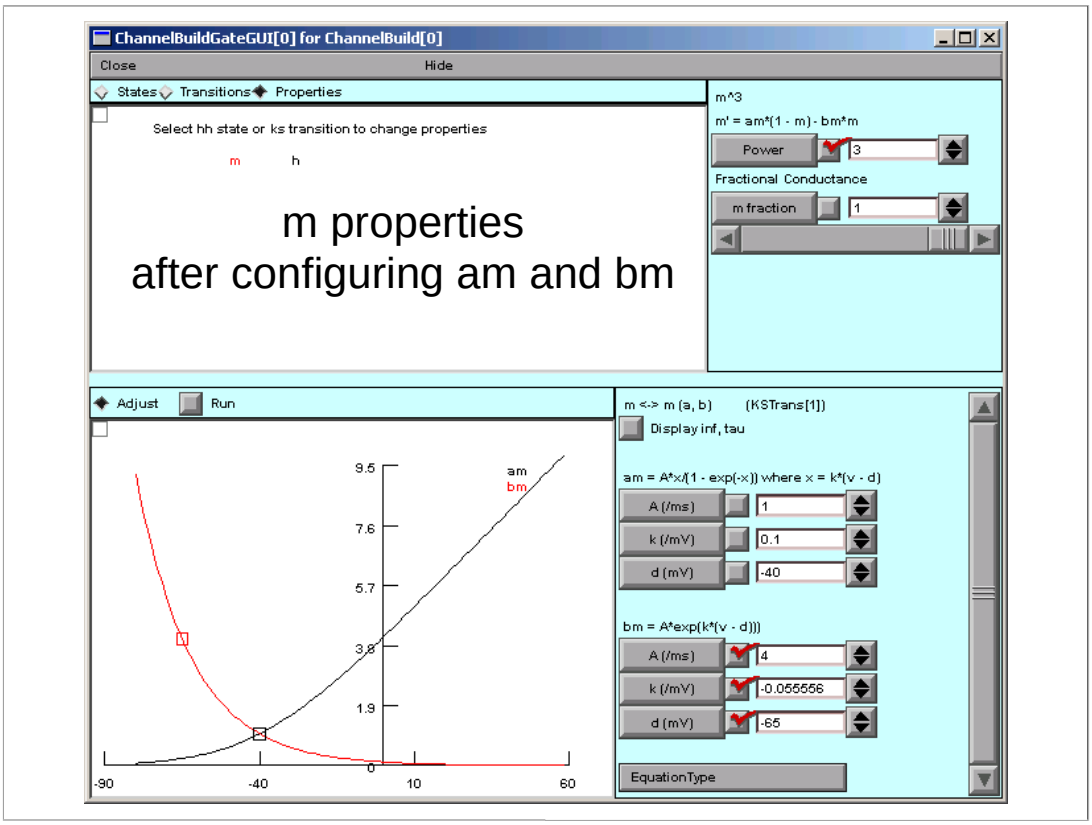
Choose functional form for am

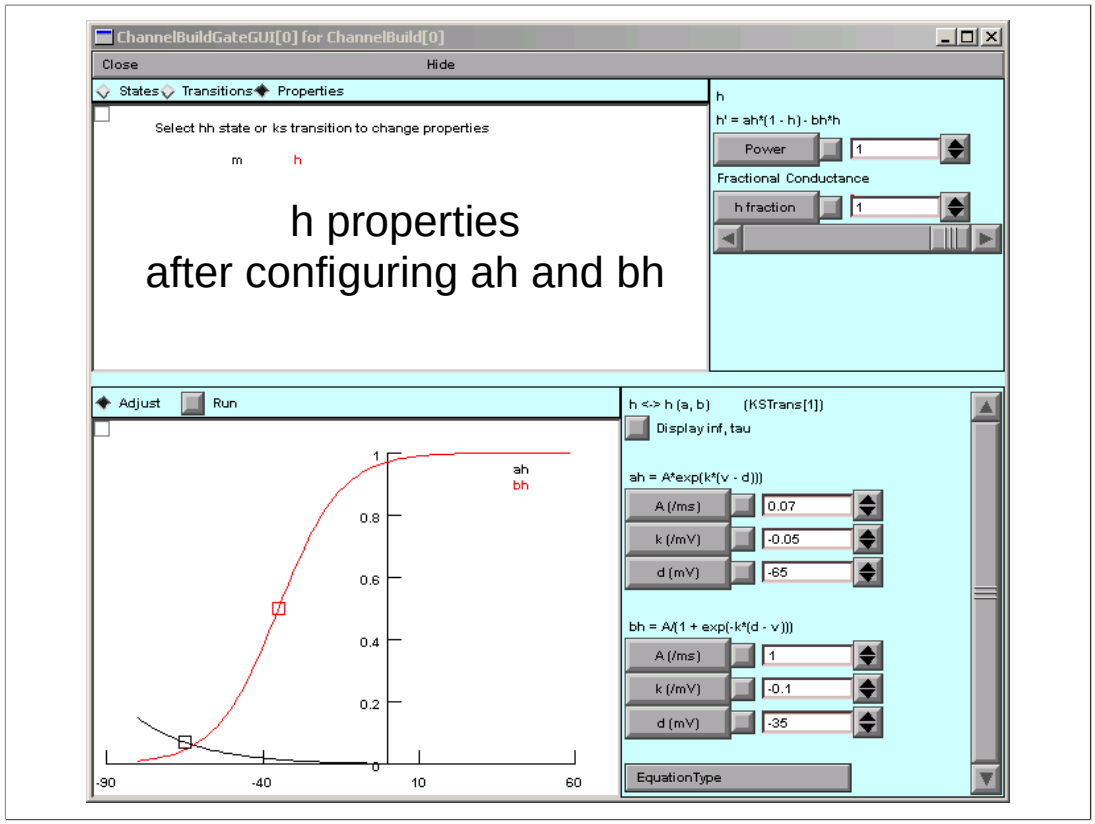


Set parameter values

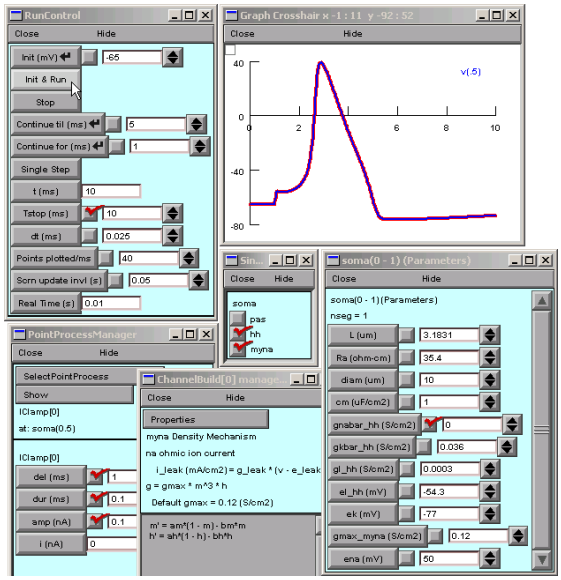


Do same for bm





Testing



NMODL

NEURON Model Description Language

Add new membrane mechanisms to NEURON

Density mechanisms

- Distributed Channels
- Ion accumulation

Point Processes

- Electrodes
- Synapses

Described by

- Differential equations
- Kinetic schemes
- Algebraic equations

Benefits

- Specification only -- independent of solution method.
- Efficient -- translated into C.
- Compact
  - One NMODL statement -> many C statements.
  - Interface code automatically generated.
- Consistent ion current/concentration interactions.
- Consistent Units

NMODL general block structure

What the model looks like from outside

```
NEURON {
        SUFFIX kchan
        USEION k  READ ek WRITE ik
        RANGE gbar, ...
}
```

What names are manipulated by this model

```
UNITS { (mV) = (millivolt) ... }

PARAMETER { gbar = .036 (mho/cm2) <0, 1e9>... }

STATE { n ... }

ASSIGNED { ik (mA/cm2) ... }
```

Initial default values for states

```
INITIAL {
        rates(v)
        n = ninf
}
```

Calculate currents (if any) as function of v, t, states

(and specify how states are to be integrated)

```
BREAKPOINT {
        SOLVE deriv METHOD cnexp
        ik = gbar * n^4 * (v - ek)
}
```

State equations

```
DERIVATIVE deriv {
        rates(v)
        n' = (ninf - n)/ntau
}
```

Functions and procedures

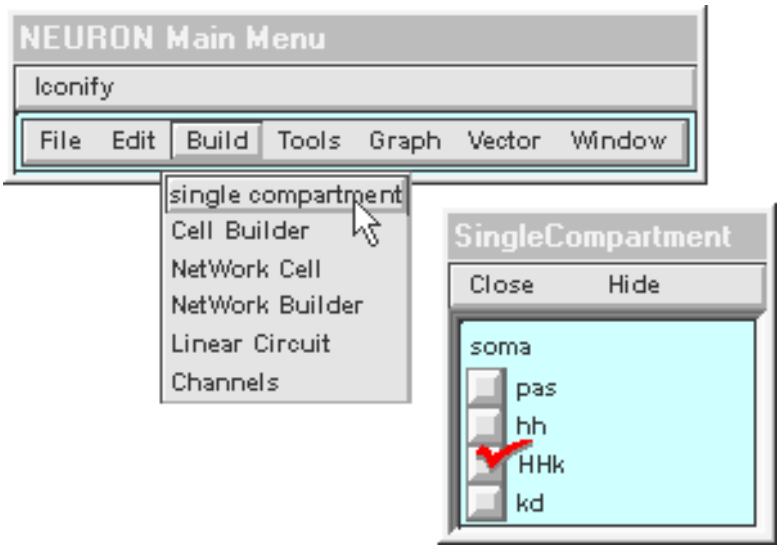
```
PROCEDURE rates(v(mV)) {
        ...
}
```

UNIX

```
nrnivmodl
nrngui
```

MSWIN

Select NEURON Main Menu / Build / single compartment



Density mechanism

NMODL

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

PARAMETER {
    g = .001 (mho/cm2) <0, 1e9>
    e = -65  (millivolt)
}

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

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

GUI

Interpreter

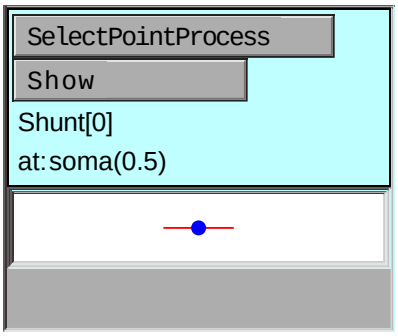
```
soma {
    insert leak
    g_leak = .0001
}
print soma.i_leak(.5)
```

Point Process

NMODL

```
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 = (.001)*(v - e)/r
}
```

GUI



Interpreter

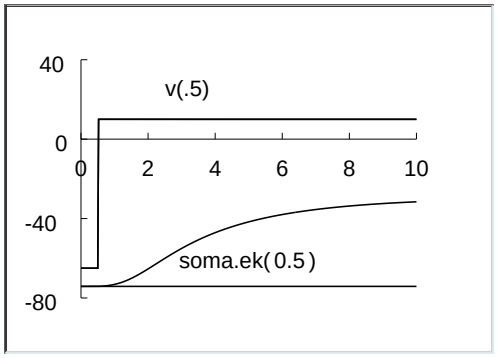
```
objref s
soma s = new Shunt(.5)
s.r = 2
```

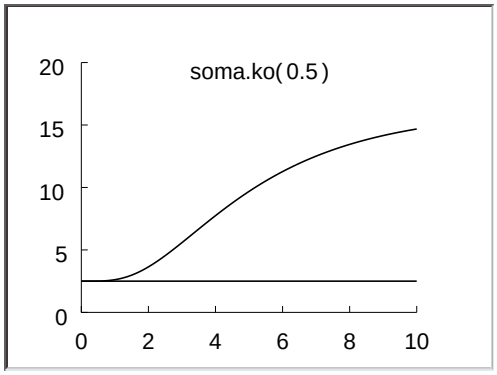
Ion Channel

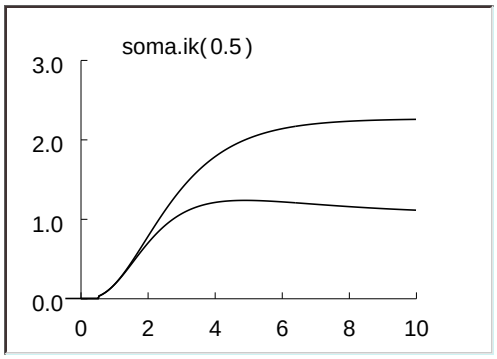
```
NEURON {
  USEION k READ ek WRITE ik
}
BREAKPOINT {
  SOLVE states METHOD cnexp
  ik = gbar*n*n*n*n*(v - ek)
}
DERIVATIVE states {
  rate(v*1(/mV))
  n' = (inf - n)/tau
}
```

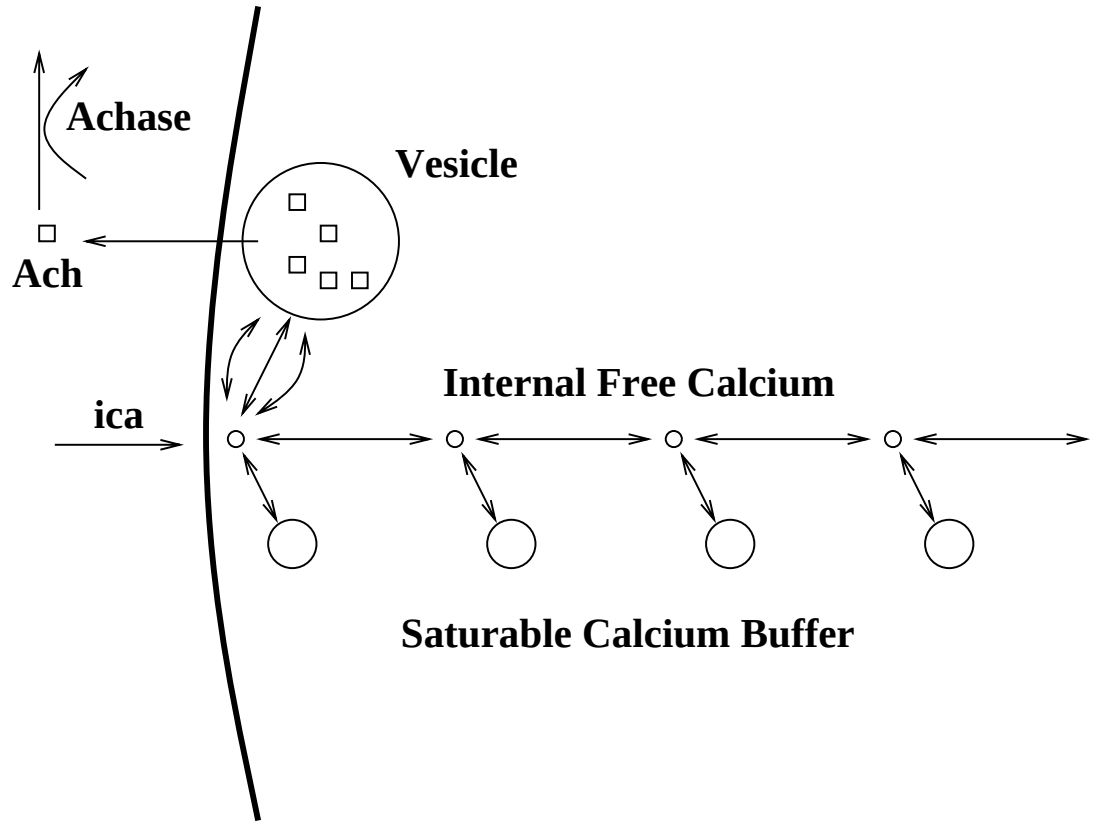
Ion Accumulation

```
NEURON {
  USEION k READ ik WRITE ko
}
BREAKPOINT {
  SOLVE state METHOD cnexp

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










```
STATE {
Vesicle Ach Achase Ach2ase X Buffer[N] CaBuffer[N] Ca[N]
}

KINETIC calcium_evoked_release {
    : release
 ~  Vesicle + 3Ca[0] <-> Ach   (Agen, Arev)
 ~  Ach + Achase <-> Ach2ase   (Aase2, 0)  :idiom for enzyme reaction
 ~  Ach2ase <-> X + Achase     (Aase2, 0)  : requires two reactions

   : Buffering
   FROM i = 0 TO N-1 {
    ~  Ca[i] + Buffer[i] <-> CaBuffer[i]   (kCaBuffer, kmCaBuffer)
   }

   :Diffusion
   FROM i = 1 TO N-1 {
    ~  Ca[i-1] <-> Ca[i]      (Dca*a[i-1], Dca*b[i])
   }

    : inward flux
 ~  Ca[0] <<     (ica)
}
```

UNITS Checking

```
NEURON { POINT_PROCESS Shunt ... }

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

ASSIGNED {
        i (nanoamp)
        v (millivolt)
}

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

Units are incorrect in the "i = ..." current assignment.
The output from

```
modlunit shunt
```

is:

```
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 14 in file shunt.mod
        i = (v - e)/r<>
```

To fix the problem replace the line with:

```
i = (.001)*(v - e)/r
```

What conversion factor will make the following consistent?

```
  nai'  =   ina    /    FARADAY  * (c/radius)
(uM/ms)   (mA/cm2) / (coulomb/mole) / (um)
```


The Linear Circuit Builder

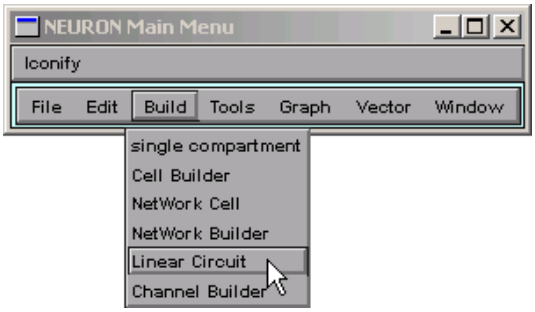
For building models that have linear circuit elements and may also involve neurons

Circuit elements include ground, current & voltage source, R, C, op amp

Potential applications include

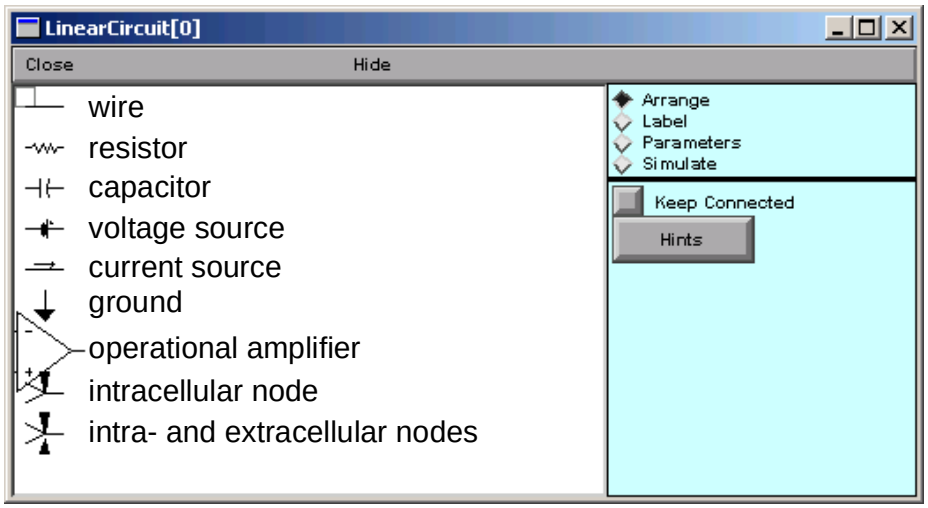
- effects and compensation of electrode R & C
- two-electrode voltage clamp
- ohmic and nonlinear gap junctions

1. Bring up a Linear Circuit Builder



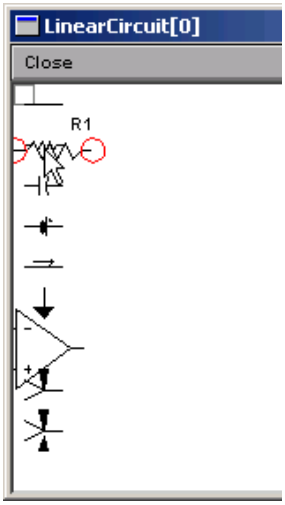
NEURON Main Menu / Build / Linear Circuit

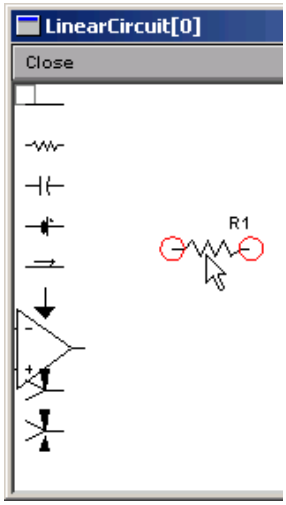
The Linear Circuit Builder



Arrange: spawn components

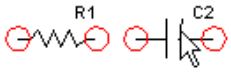
Click on palette and drag onto canvas



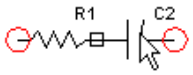


Arrange: connect components

Click and drag to overlap red circles



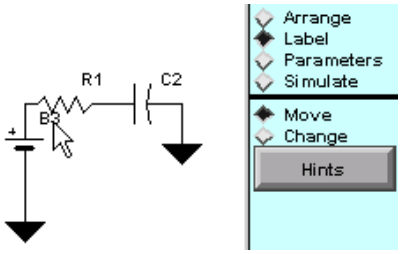
Black square is "solder joint"

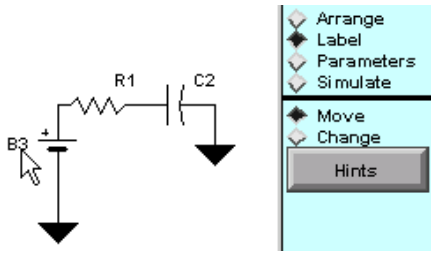


Pull apart to break connection

Label: move labels

Click and drag to new location

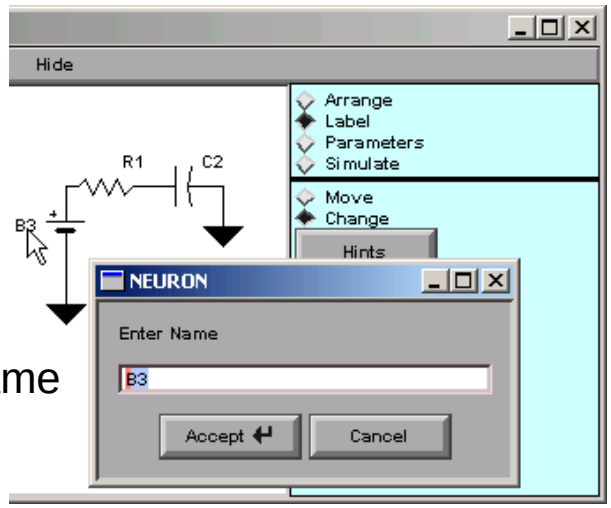




Label: change labels 1

Click on a label . . .

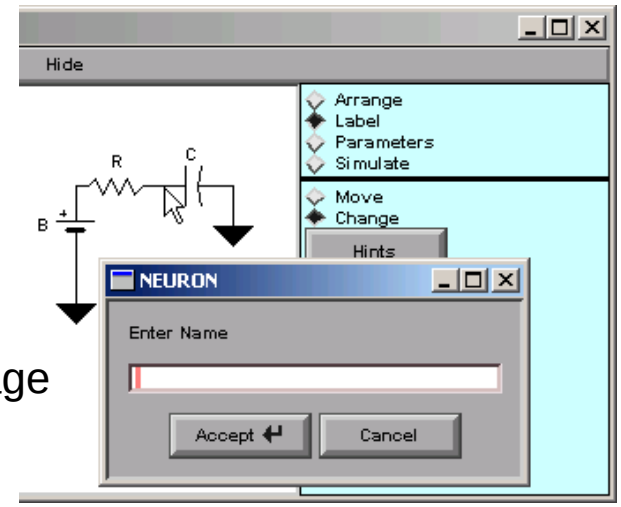
. . . to change its name



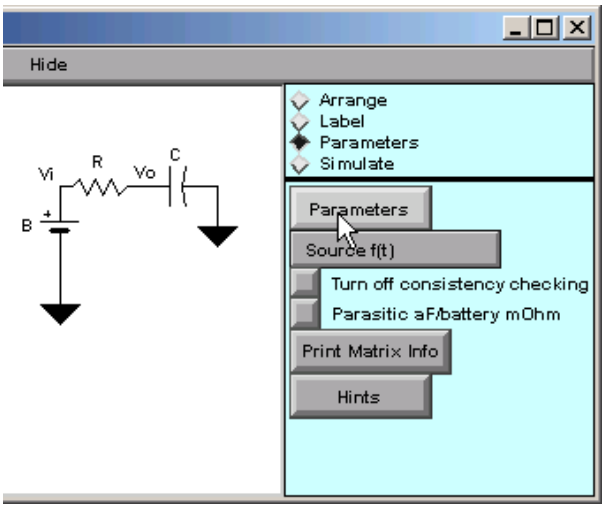
Label: change labels 2

Click on a node . . .

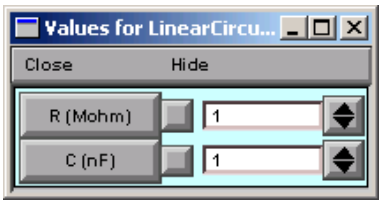
. . . to label a voltage



Parameters: non-source elements

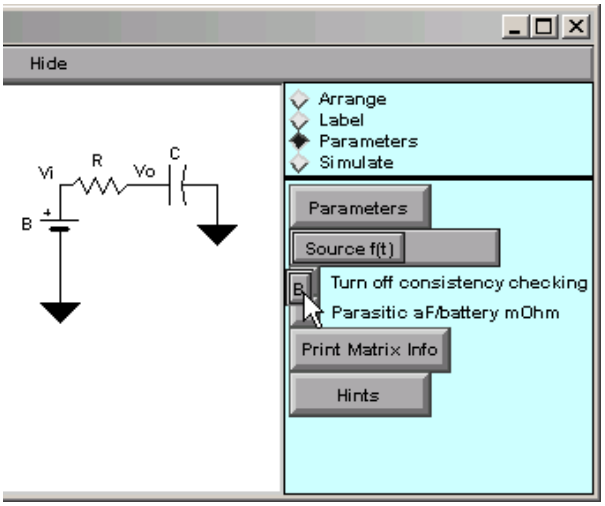


Click on "Parameters"

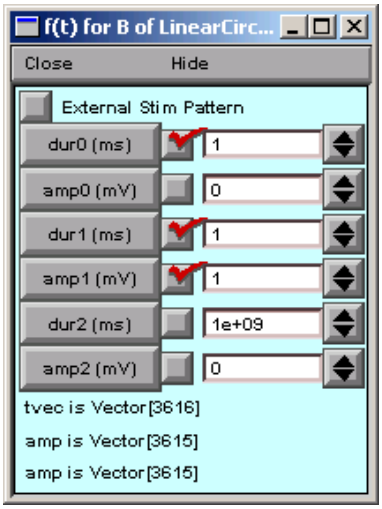


Parameters: signal sources

Source f(t) / B



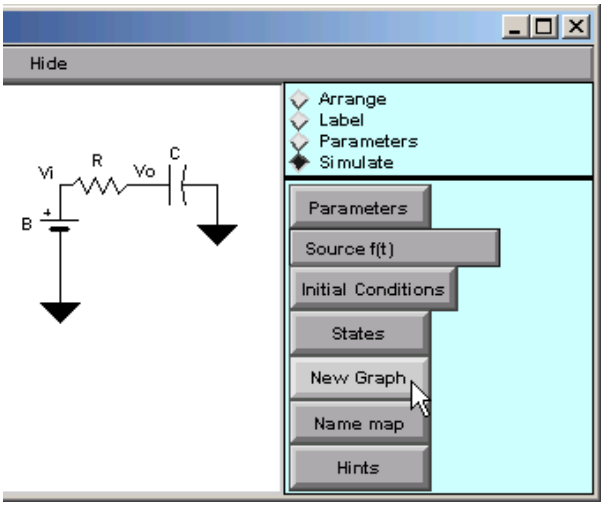
Parameters: signal sources *continued*



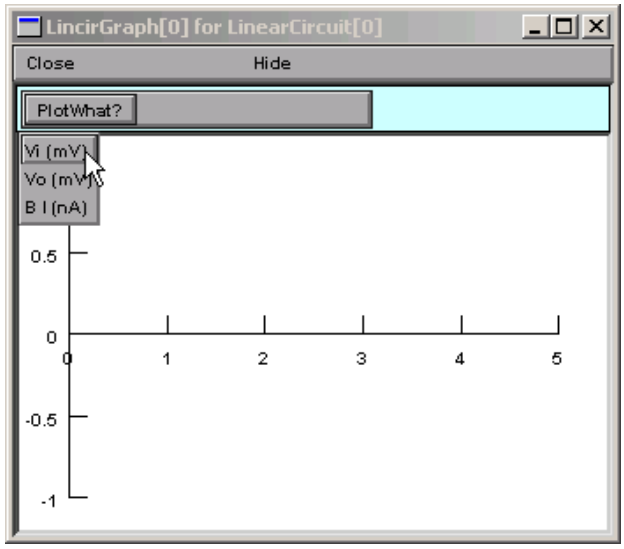
Configured

Simulate: creating a graph

New Graph

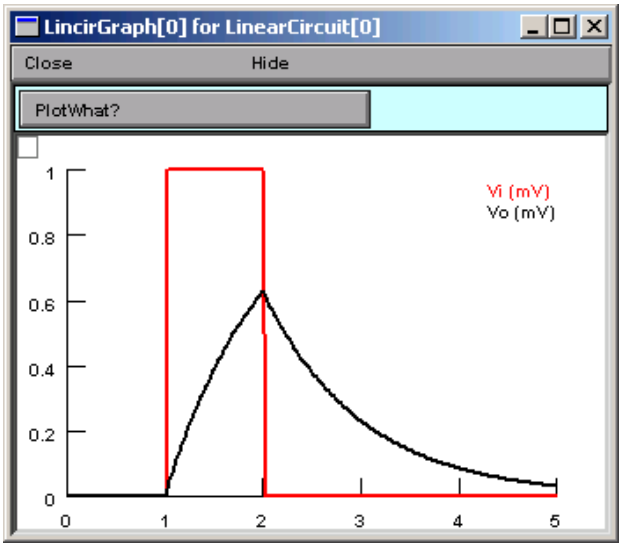


Simulate: specifying what to plot



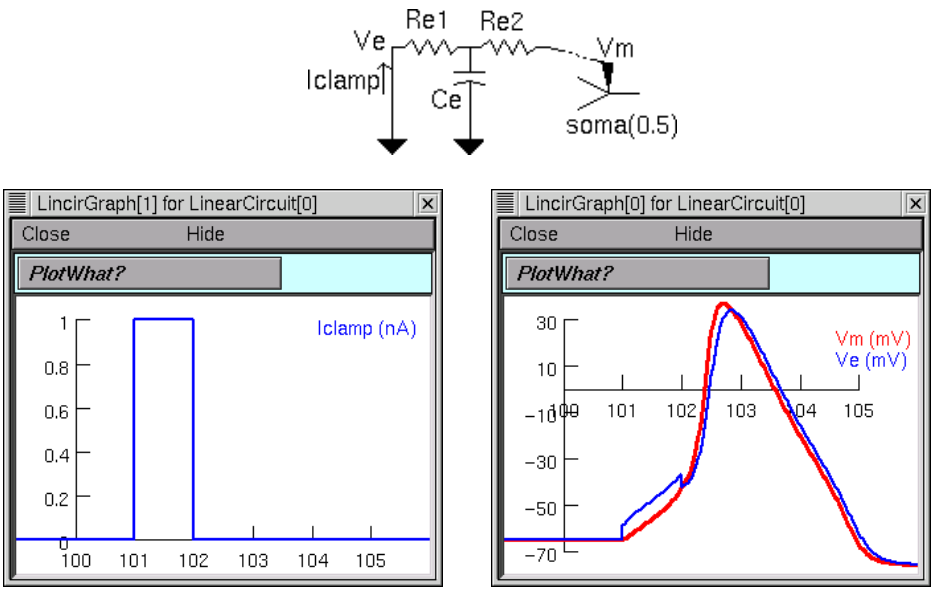
PlotWhat? / *variable_label*

Simulate: simulation results

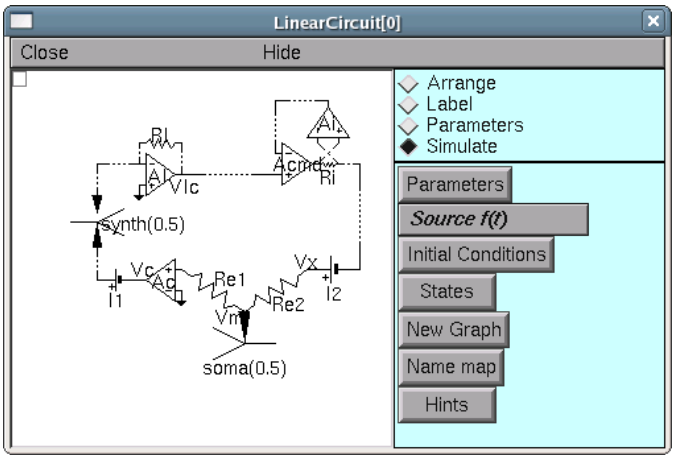


After minor cosmetic changes

Patch clamp with electrode R and C



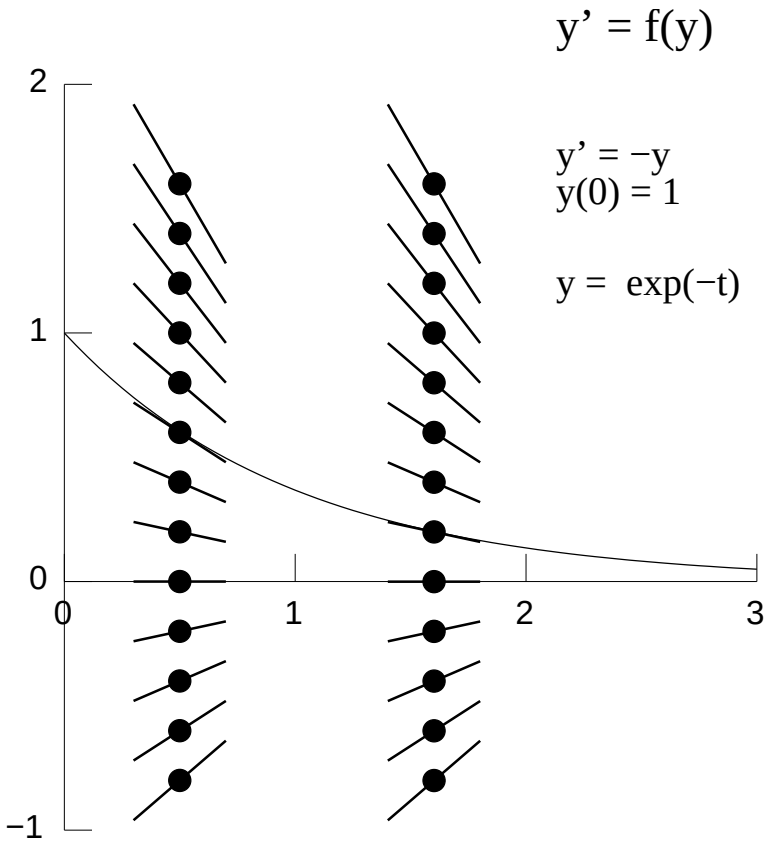
NEURON demo: dynamic clamp



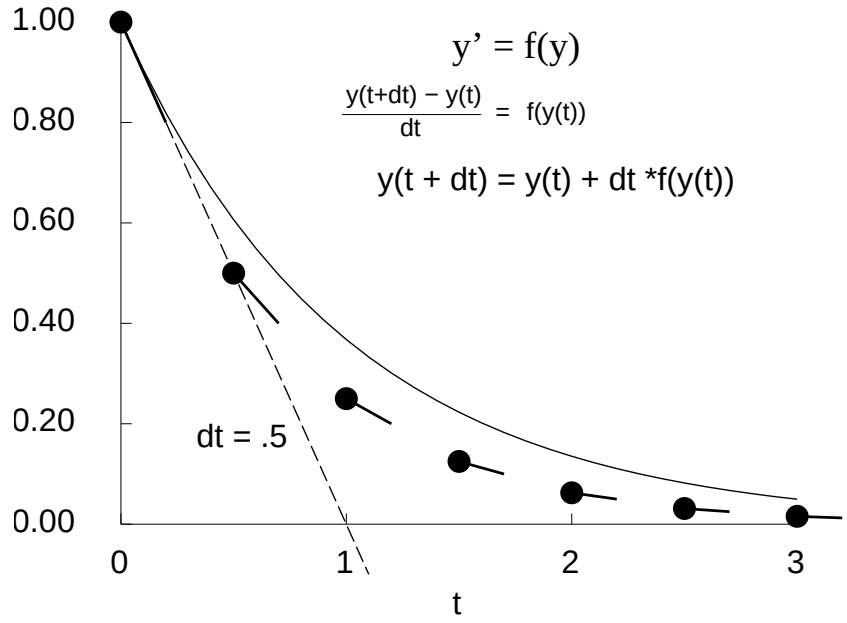
Compartmental Modeling

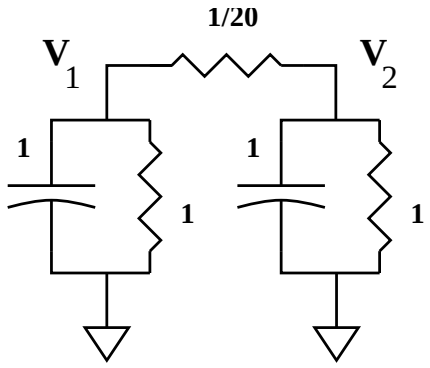
Not much mathematics required.

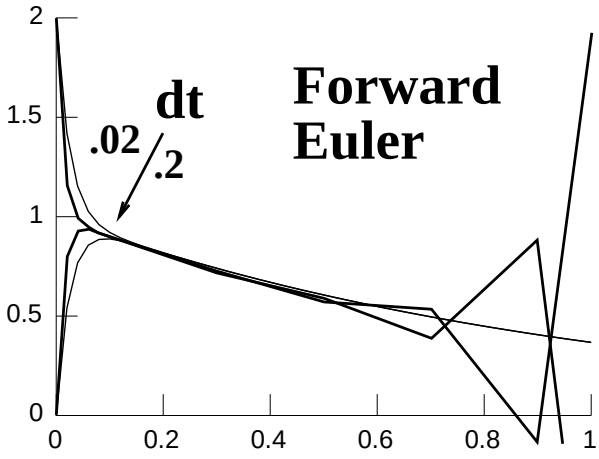
Good judgment essential!



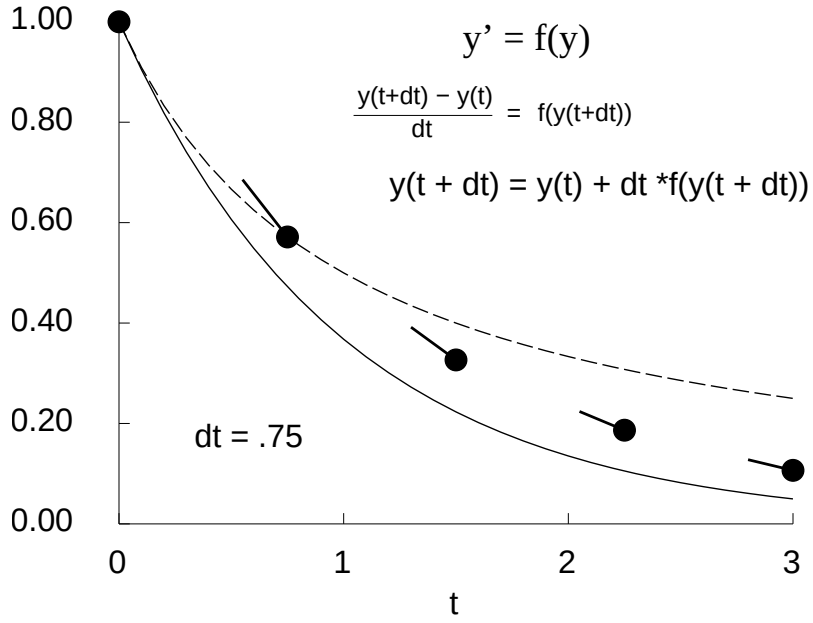
Forward Euler

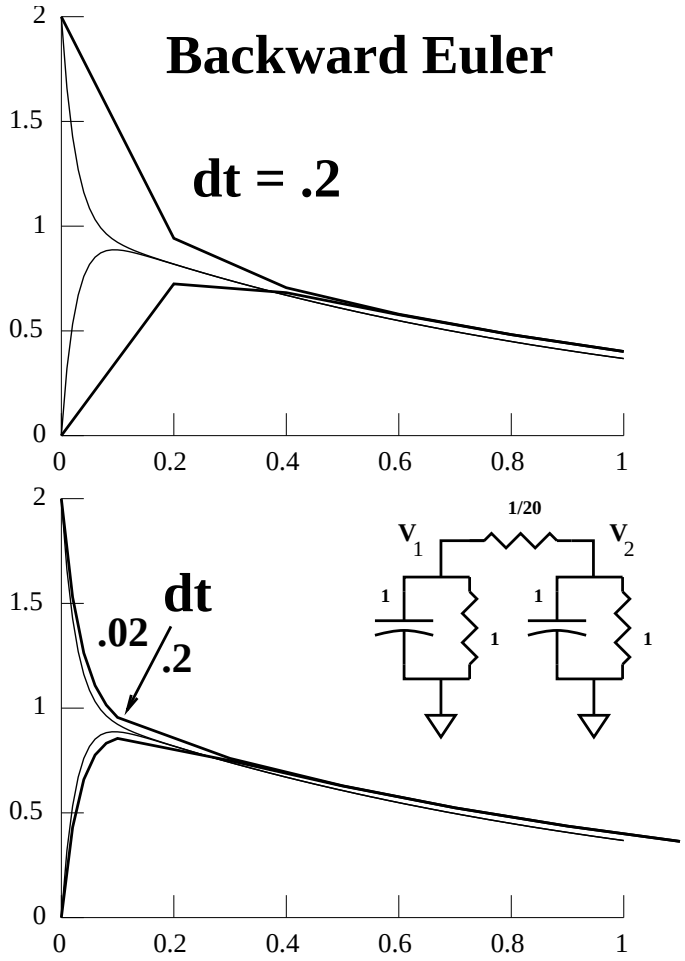




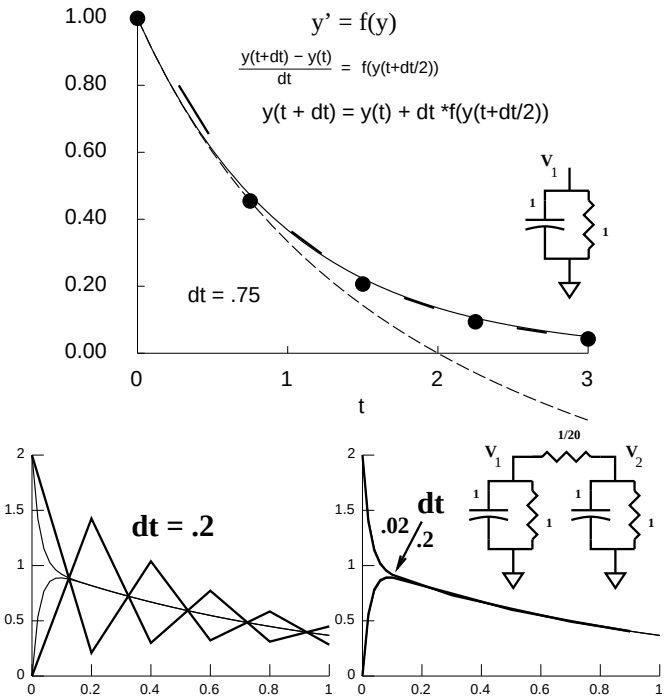


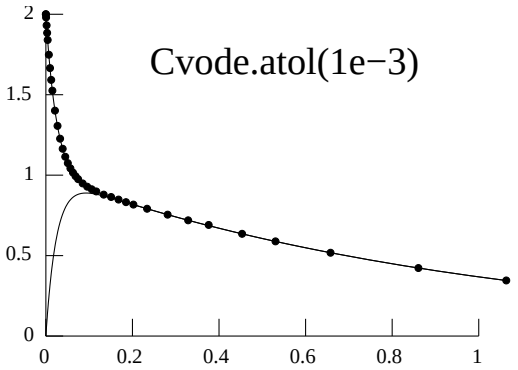
Backward Euler

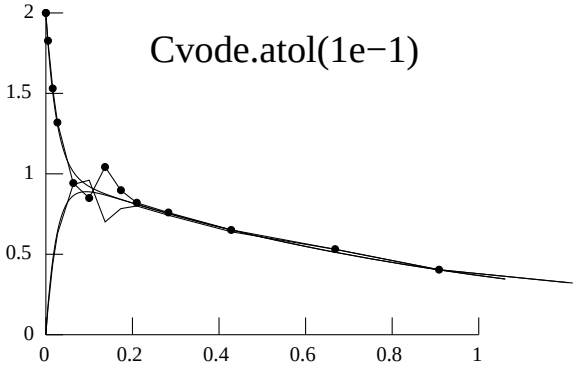


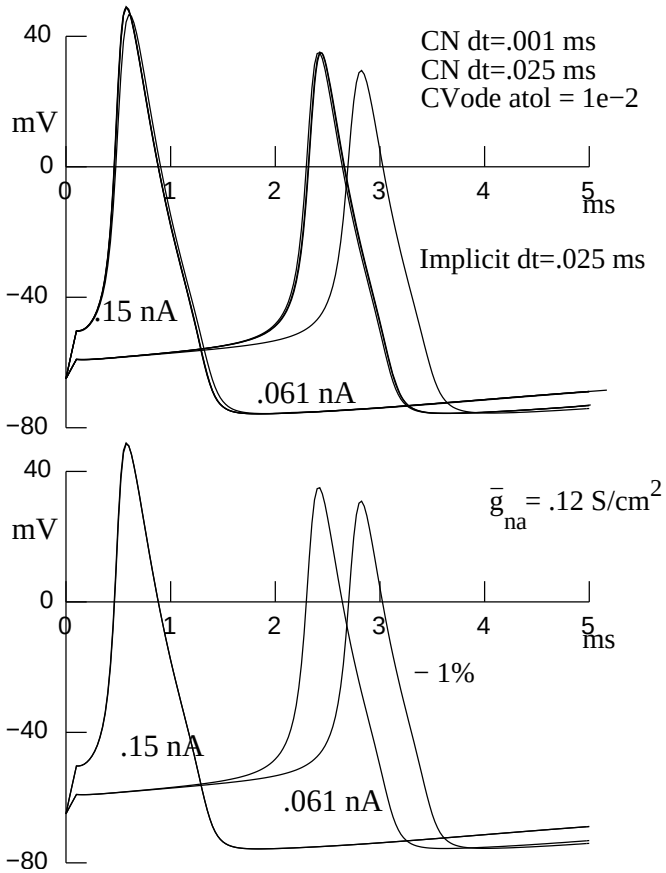


Crank−Nicholson









Networks: spike-triggered synaptic transmission, events, and artificial spiking cells

1. Define the types of cells
2. Create each cell in the network
3. Connect the cells

Communication between cells

Gap junctions

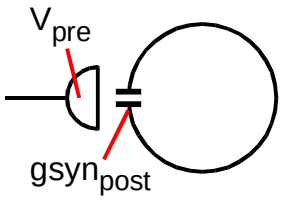
Synaptic transmission
- graded
- spike-triggered

Graded synaptic transmission

Physical system:

A presynaptic variable governs continuous transmitter release

Transmitter modulates a postsynaptic property



$gsyn_{post} = f(V_{pre})$

Problem: how does postsynaptic cell know V_{pre}?

Graded synaptic transmission *continued*

Answer: use POINTER to link postsynaptic variable to the presynaptic variable

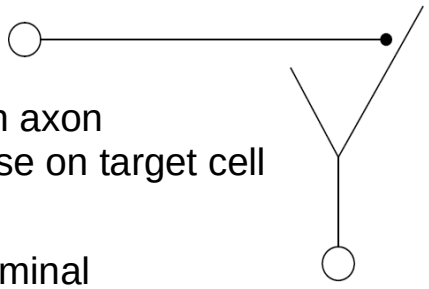
NMODL specification of synaptic mechanism:

```
NEURON {
   POINT_PROCESS Syn
   POINTER v_pre
}
```

hoc usage

```
objref syn
dend syn = new Syn(0.5)
setpointer syn.v_pre, precell.axon.v(1)
```

Spike-triggered synaptic transmission



Physical system:

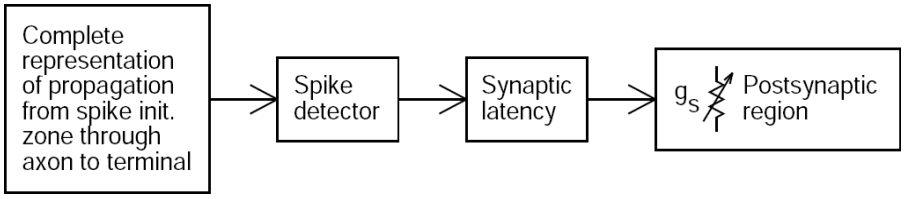
- Presynaptic neuron with axon that projects to synapse on target cell

Conceptual model:

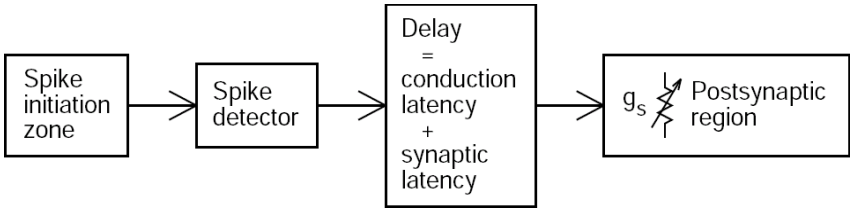
- Spike in presynaptic terminal triggers transmitter release; presynaptic details unimportant
- Postsynaptic effect described by DE or kinetic scheme that is perturbed by occurrence of a presynaptic spike

Spike-triggered transmission: computational implementation

Basic idea



More efficient: "virtual spike propagation"



The NetCon class

hoc usage

```
netcon = new NetCon(source, target)
presection netcon = new NetCon(&v(x), \
    target, threshold, delay, weight)
```

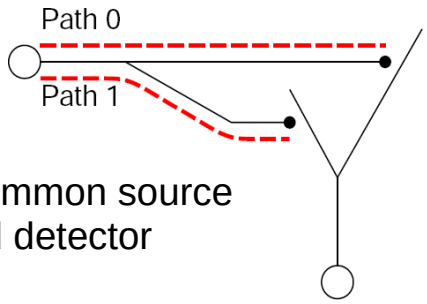
Defaults

```
threshold = 10
delay = 1 // must be > 0
weight = 0
```

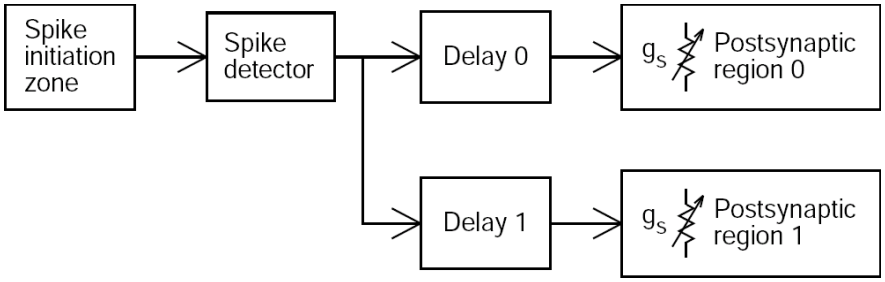
NMODL specification of synaptic mechanism

```
NET_RECEIVE(weight(microsiemens)) {
    . . .
}
```

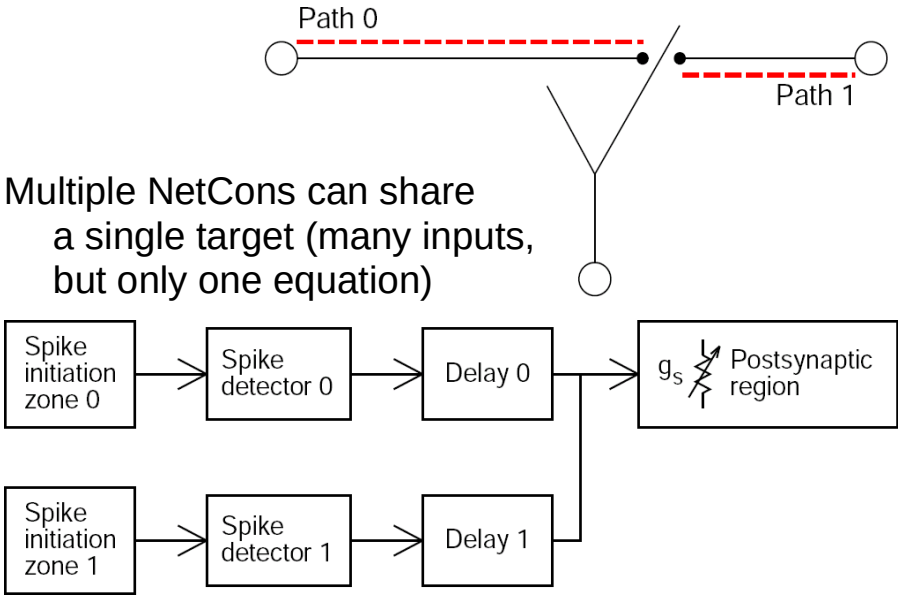
Efficient divergence



Multiple NetCons with a common source share a single threshold detector



Efficient convergence

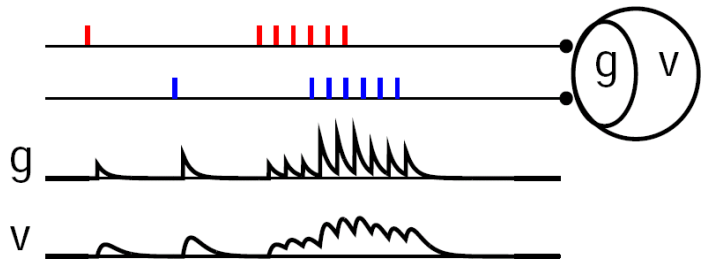


Example: g_s with fast rise and exponential decay

```
NEURON {
  POINT_PROCESS ExpSyn
  RANGE tau, e, i
  NONSPECIFIC_CURRENT i
}
  . . . declarations . . .
INITIAL { g = 0 }
CHANNEL {
  SOLVE state METHOD cnexp
  i = g*(v-e)
}
DERIVATIVE state { g' = -g/tau }
NET_RECEIVE(w) { g = g + w }
```

g_s with fast rise and exponential decay

continued

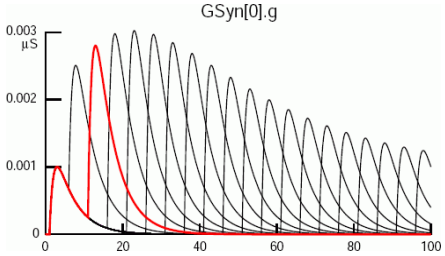



```
CHANNEL {
  SOLVE state METHOD cnexp
  i = g*(v-e)
}
DERIVATIVE state { g' = -g/tau }
NET_RECEIVE(w) { g = g + w }
```

Example: use-dependent synaptic plasticity

```
CHANNEL {
  SOLVE state METHOD cnexp
  g = B - A
  i = g*(v-e)
}
DERIVATIVE state {
  A' = -A/tau1
  B' = -B/tau2
}
NET_RECEIVE(weight, w, G1, G2, t0) {
  INITIAL {w=0 G1=0 G2=0 t0=t}
  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)
  g = g + w
  A = A + w*factor
  B = B + w*factor
}
```

Use-dependent synaptic plasticity *continued*




```
NET_RECEIVE(weight, w, G1, G2, t0) {
  INITIAL {w=0 G1=0 G2=0 t0=t}
  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)
  g = g + w
  A = A + w*factor
  B = B + w*factor
}
```

Artificial spiking cells

"Integrate and fire" cells

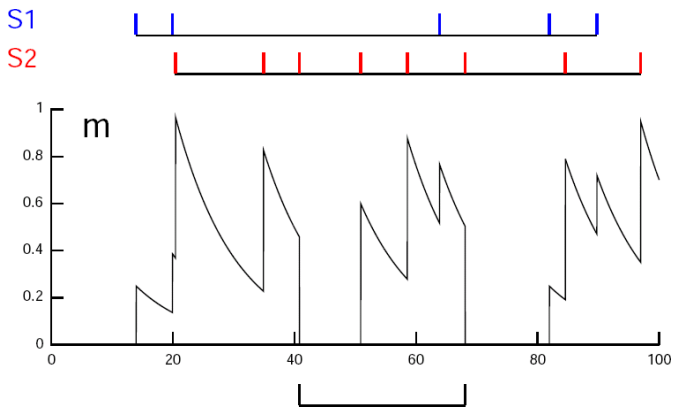
Prerequisite: all state variables must be analytically computable from a new initial condition

Orders of magnitude faster than numerical integration

Event-driven simulation run time is
- *proportional* to # of received events
- *independent* of # of cells, # of connections, and problem time

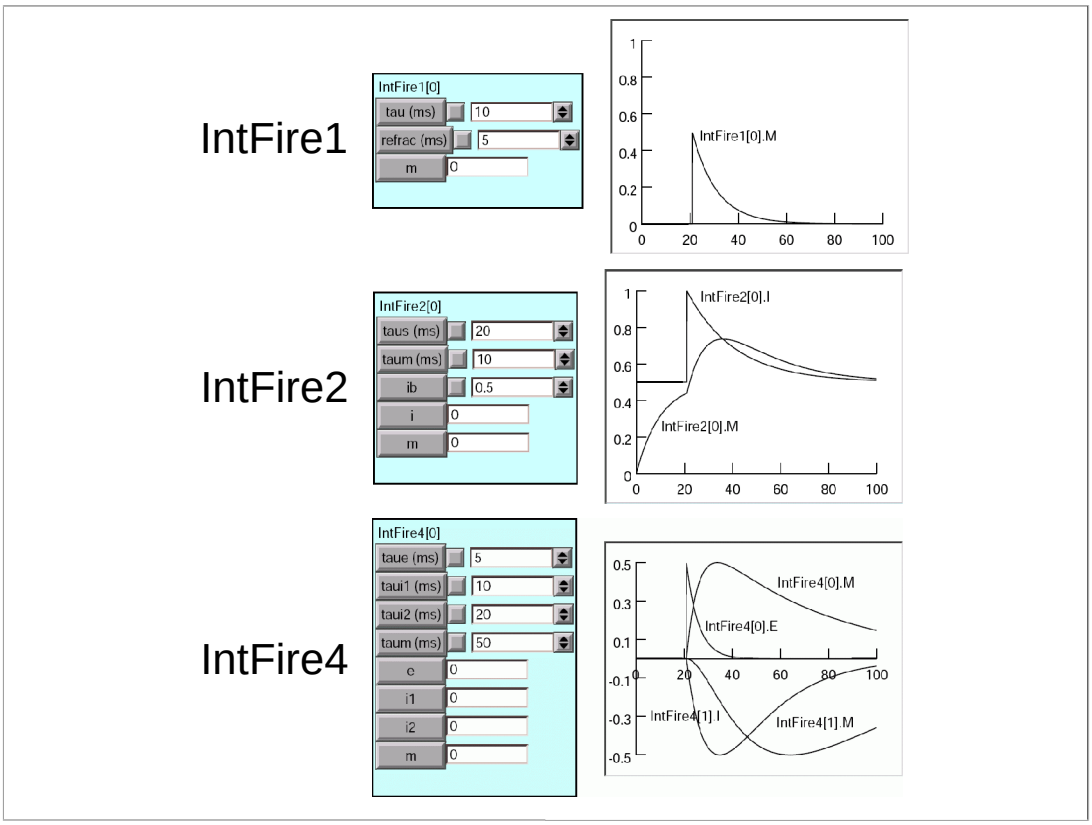
Hybrid networks

Example: leaky integrate and fire model



Leaky integrate and fire model *continued*

```
NEURON {
  ARTIFICIAL_CELL IntFire
  RANGE tau, m
}
  . . . declarations . . .
INITIAL { m = 0   t0 = t }
NET_RECEIVE (w) {
  m = m*exp(-(t-t0)/tau)
  t0 = t
  m = m + w
  if (m > 1) {
    net_event(t)
    m = 0
  }
}
```



Defining the types of cells

Artificial spiking cells

- Point Processes with a NET_RECEIVE block that calls net_event
- NetStim, IntFire1, IntFire2, IntFire4

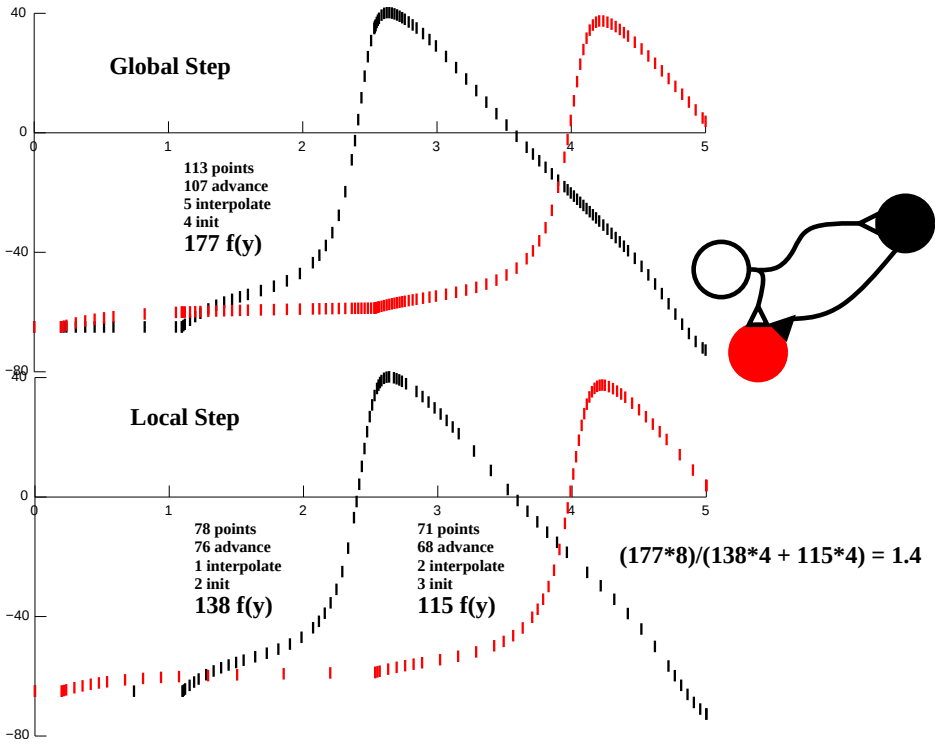
Biophysical model cells

- "Real" model cells
- Sections and density mechanisms
- Synapses are Point Proceses that affect membrane current and have a NET_RECEIVE block, e.g. ExpSyn, Exp2Syn

Defining types of biophysical model cells

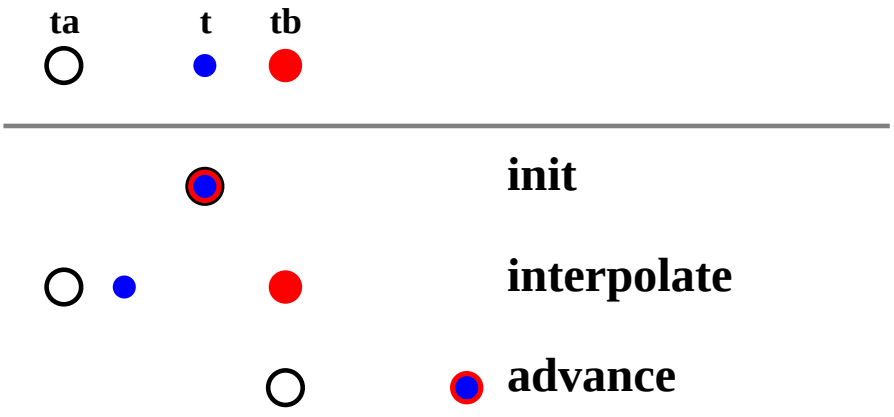
Encapsulate in a class

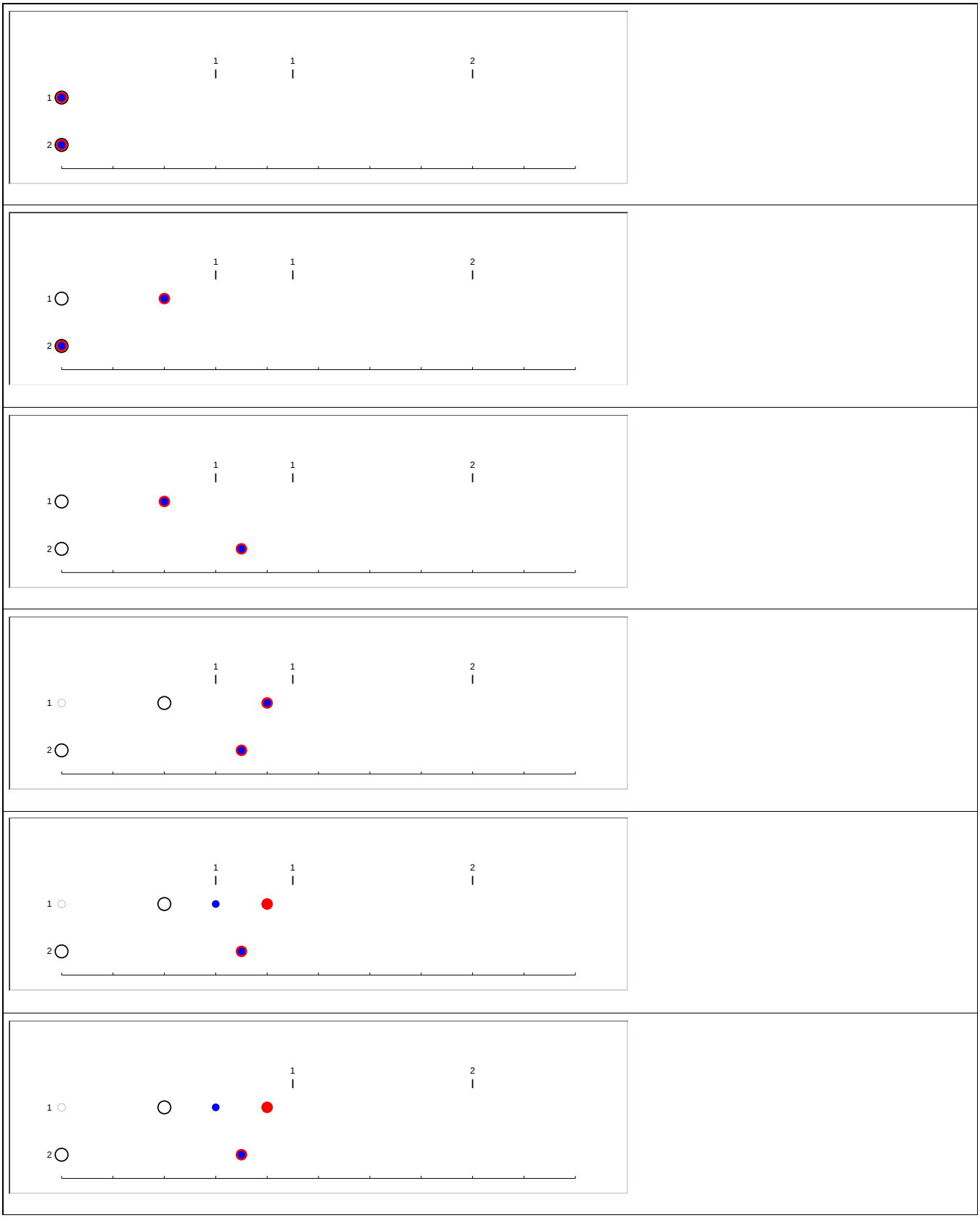
```
begintemplate Cell
  public soma, E, I
  create soma
  objref E, I
  proc init() {
    soma {
      insert hh
      E = new ExpSyn(0.5)
      I = new Exp2Syn(0.5)
      I.e = -80
    }
  }
endtemplate Cell

objref bag_of_cells
bag_of_cells = new List()
for i = 1,1000 bag_of_cells.append(new Cell())
```

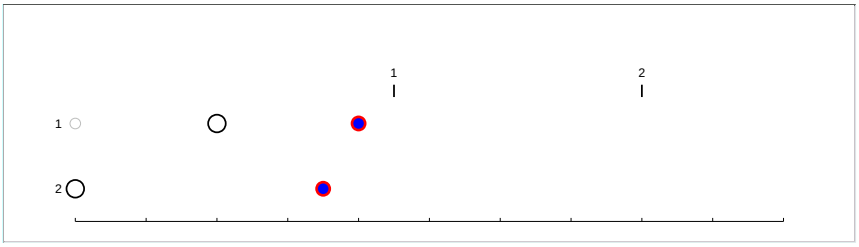


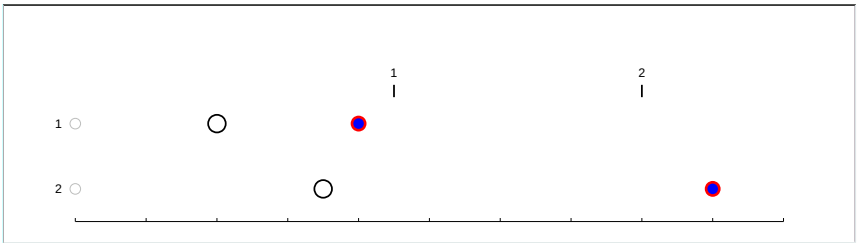
One integrator instance per cell

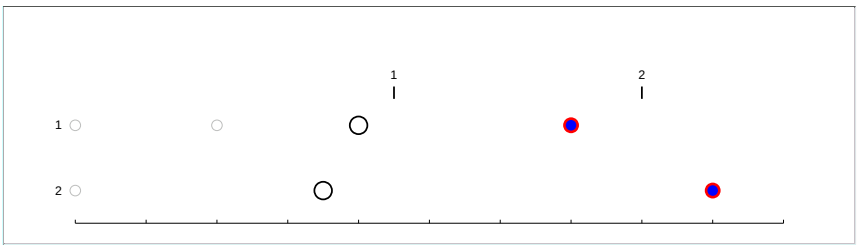
$$\forall i, j: \quad ta_i \leq tb_j$$

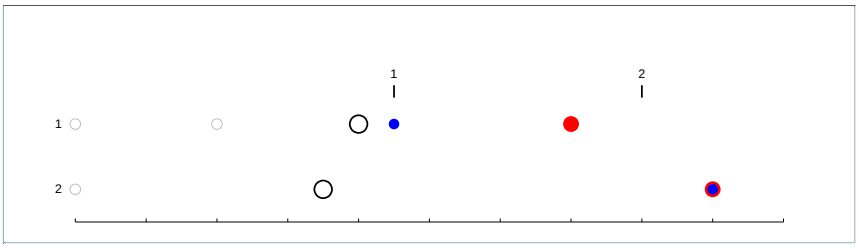


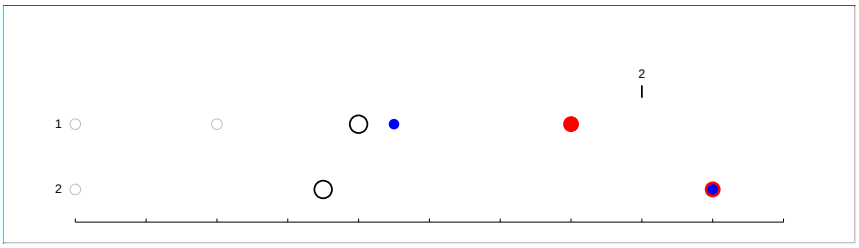


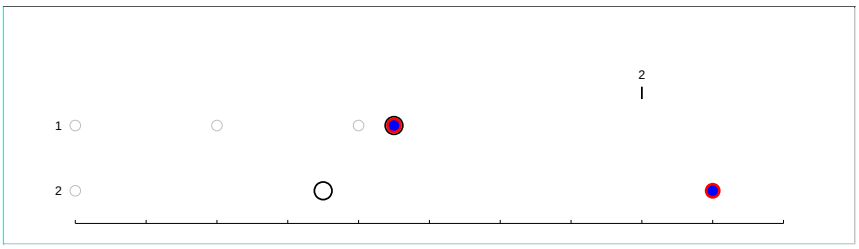


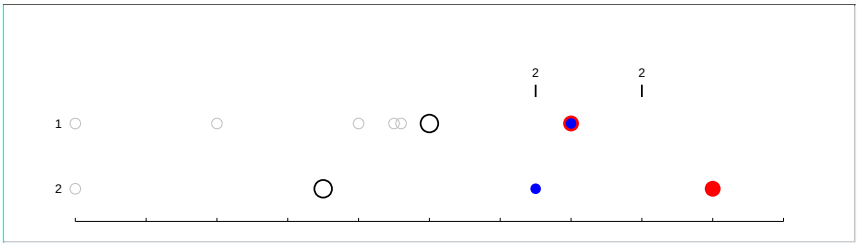


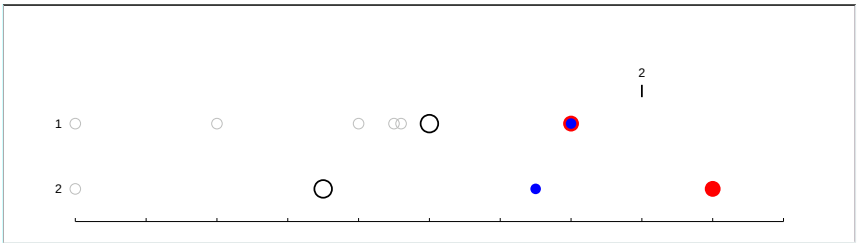


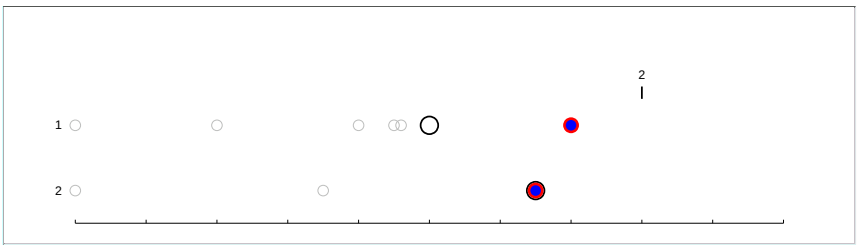


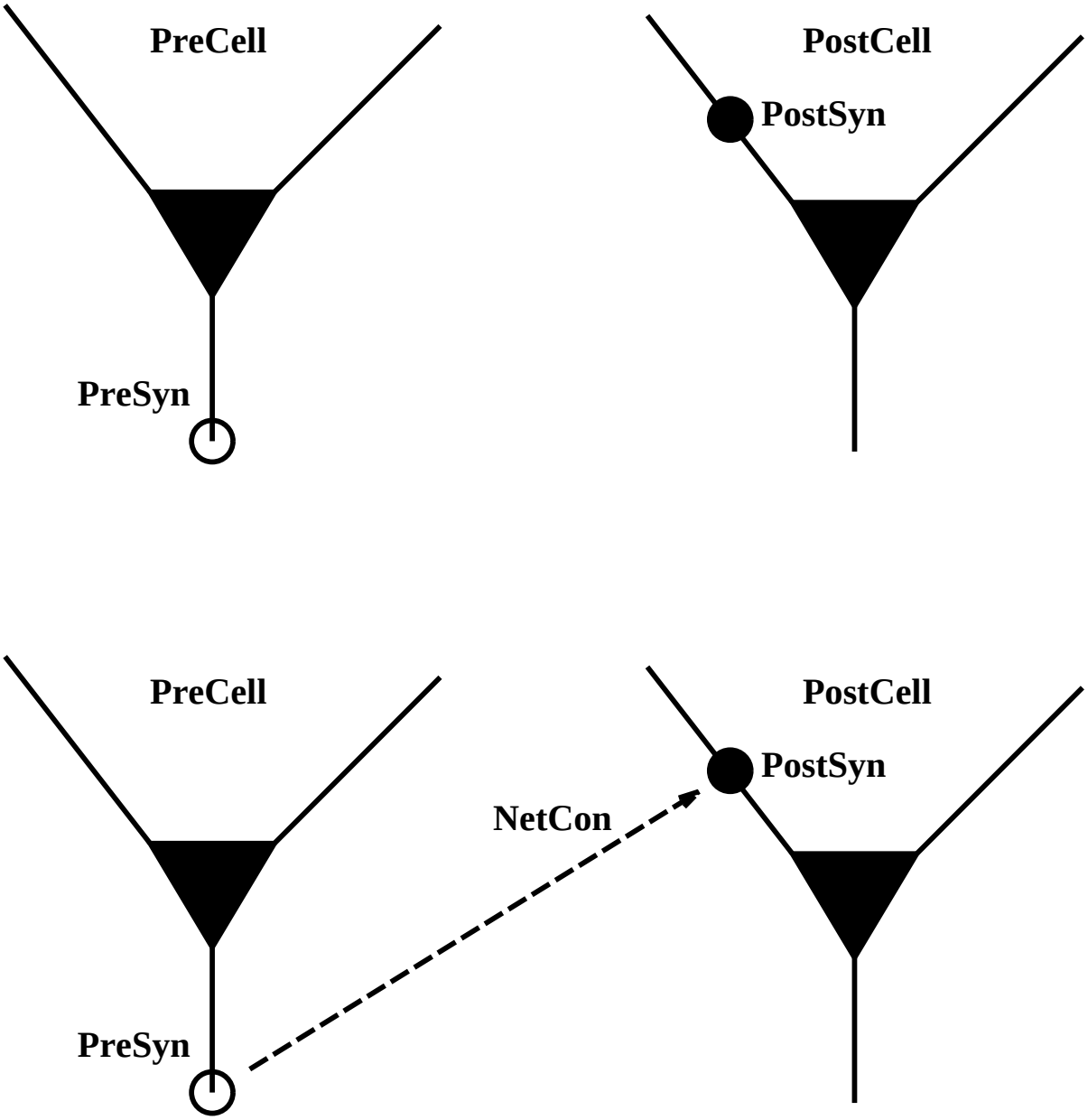




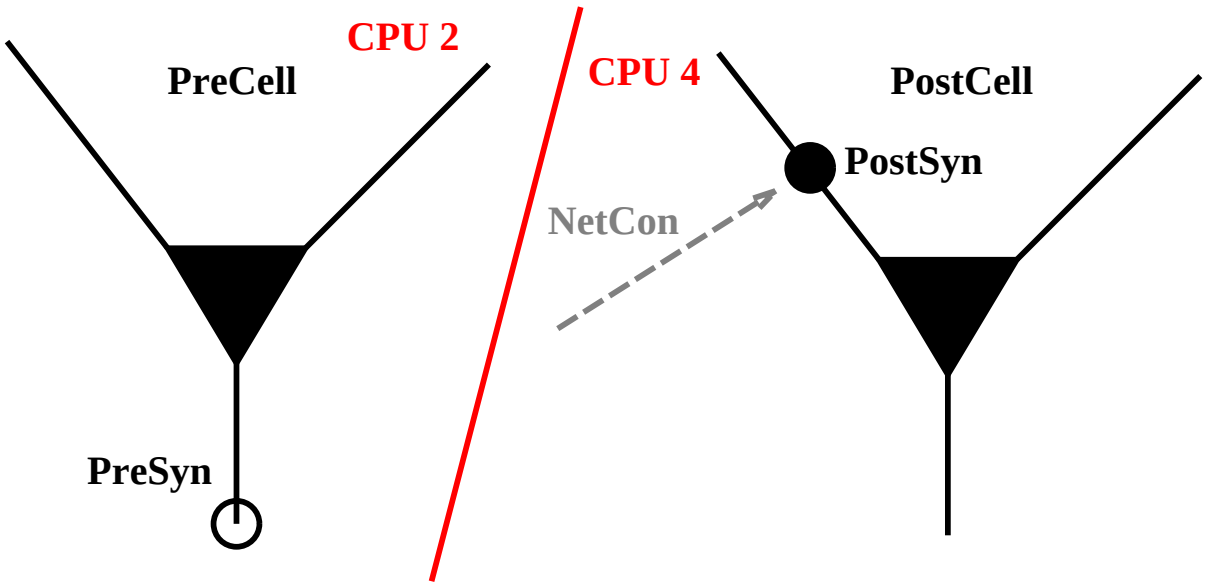


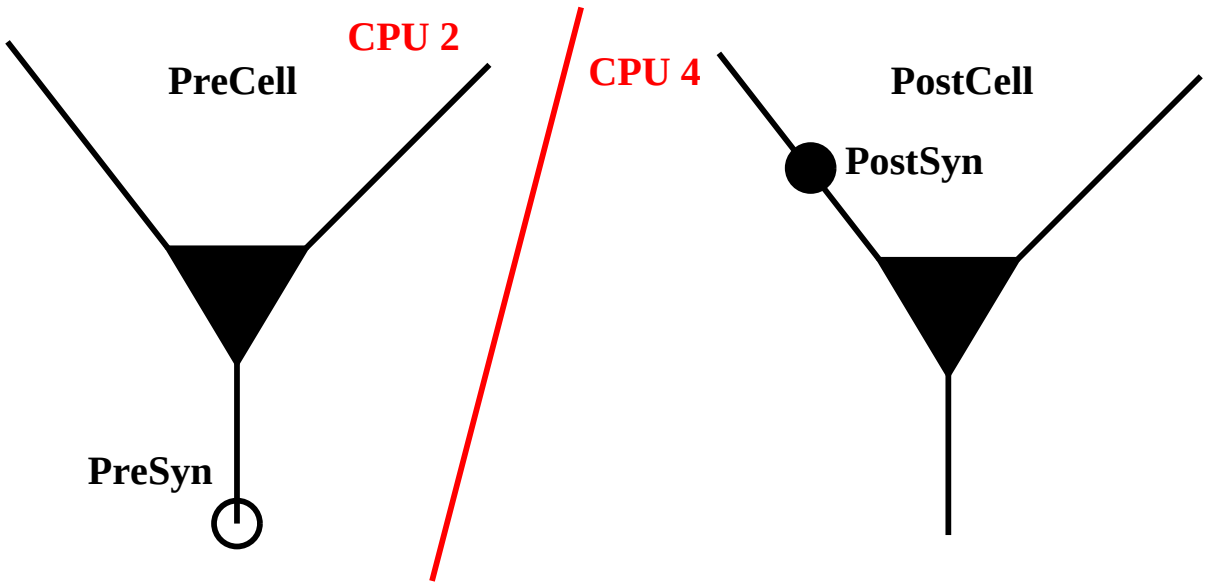




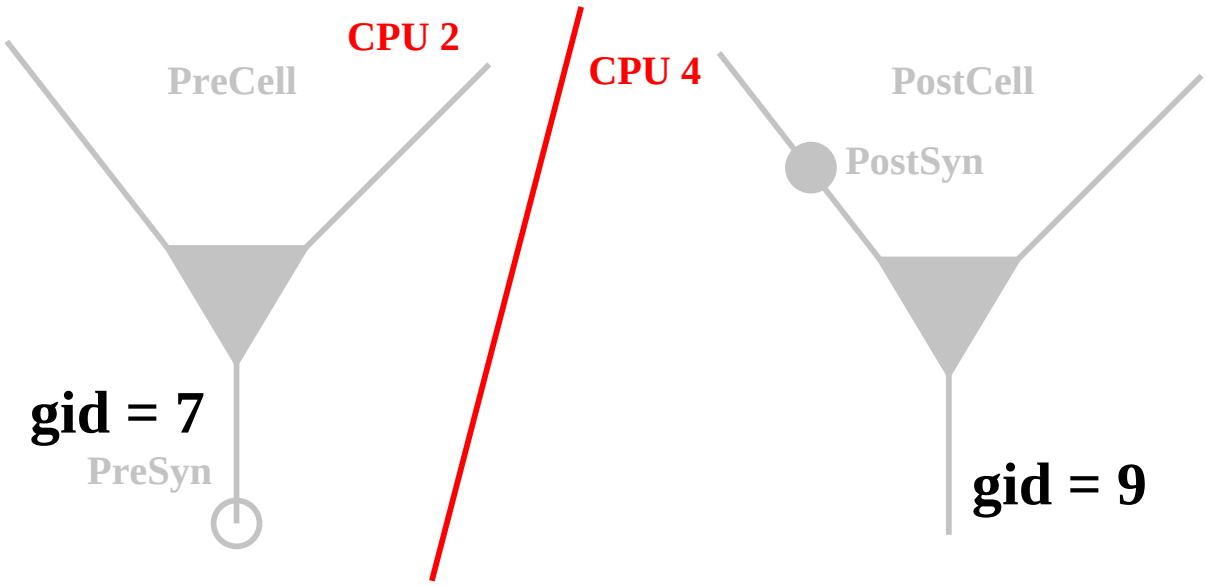



```
nc = new NetCon(PreSyn, PostSyn)
```




```
pc = new ParallelContext()
```



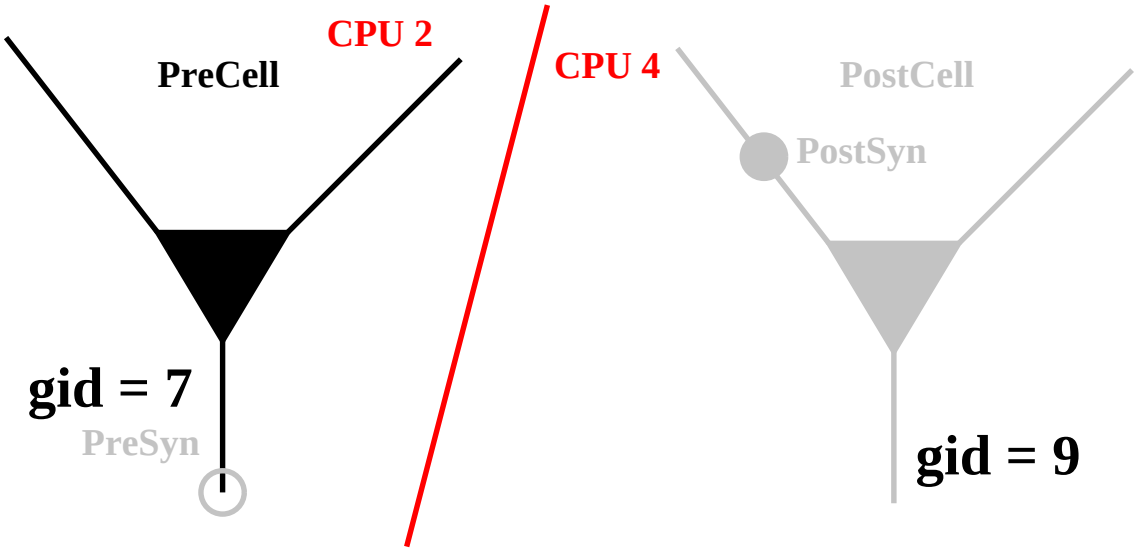
Every spike source (cell) must have a global id number.

CPU 0		CPU 3		CPU 4	
pc.id	0	pc.id	3	pc.id	4
pc.nhost	5	pc.nhost	5	pc.nhost	5
ncell	14	ncell	14	ncell	14
gid		gid		gid	
0		3		4	
5		8		9	
10		13			

An efficient way to distribute:

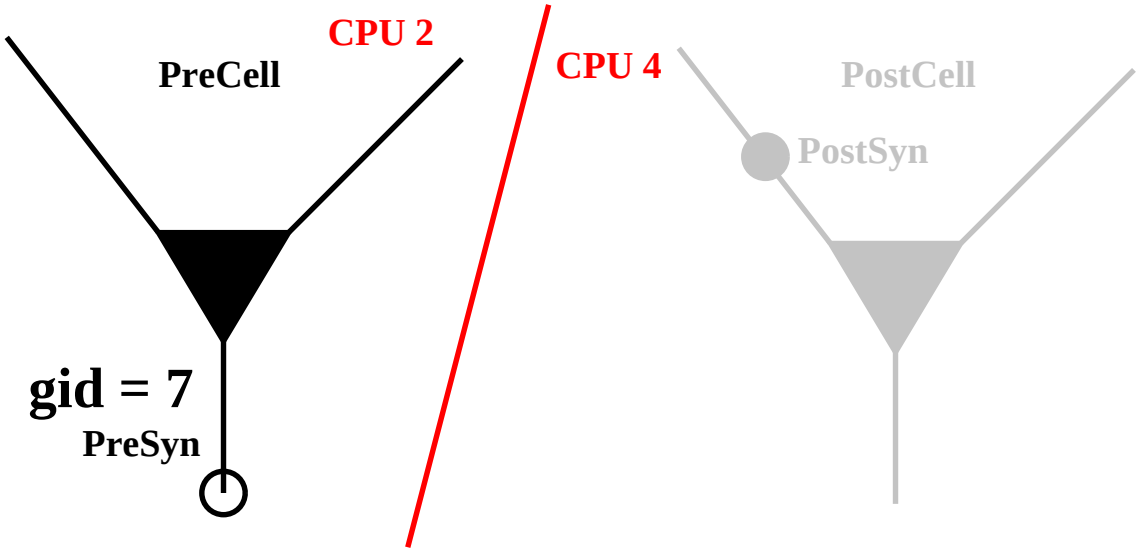
```
for (gid = pc.id; gid < ncell; gid += pc.nhost)
    pc.set_gid2node(gid, pc.id)
    ...
}
```

body executed only ncell/nhost times, not ncell.



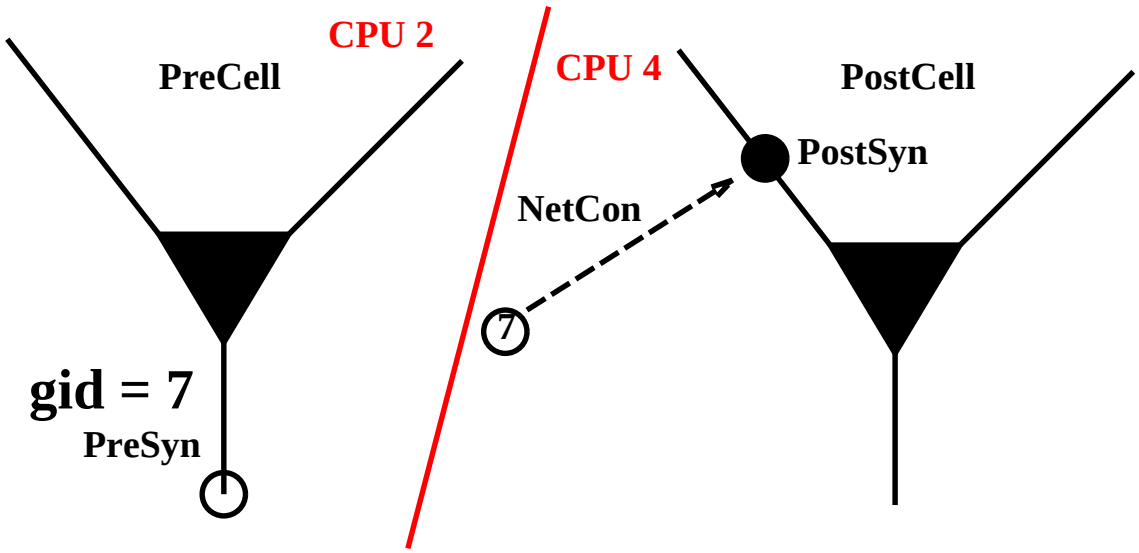
Create cell only where the gid exists.

```
if (pc.gid_exists(7)) {
    PreCell = new Cell()
}
```



Associate gid with spike source.

```
nc = new NetCon(PreSyn, nil)
pc.cell(7, nc)
```



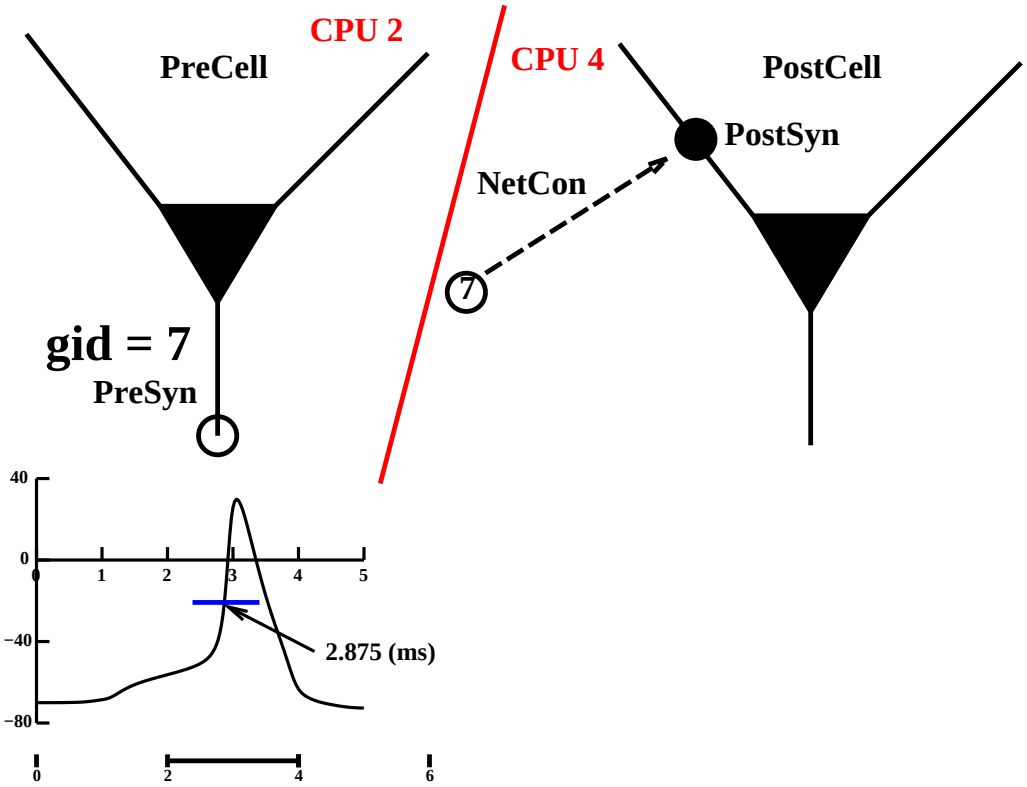
Create NetCon on CPU where target exists.

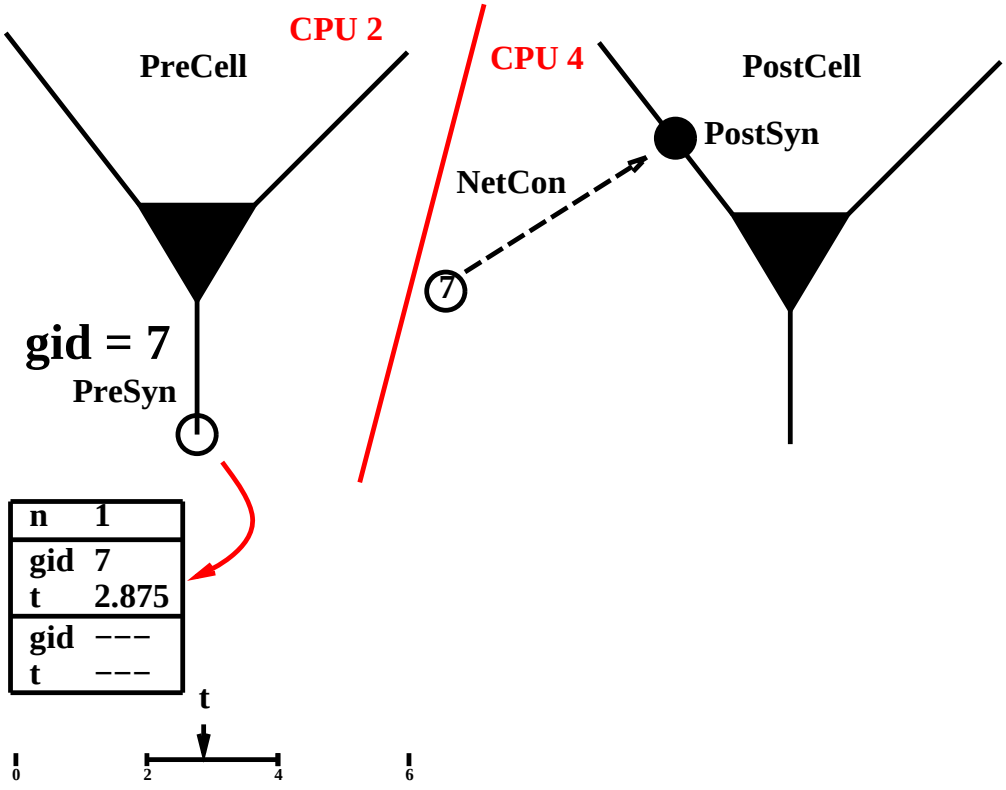
```
nc = pc.gid_connect(7, PostSyn
```

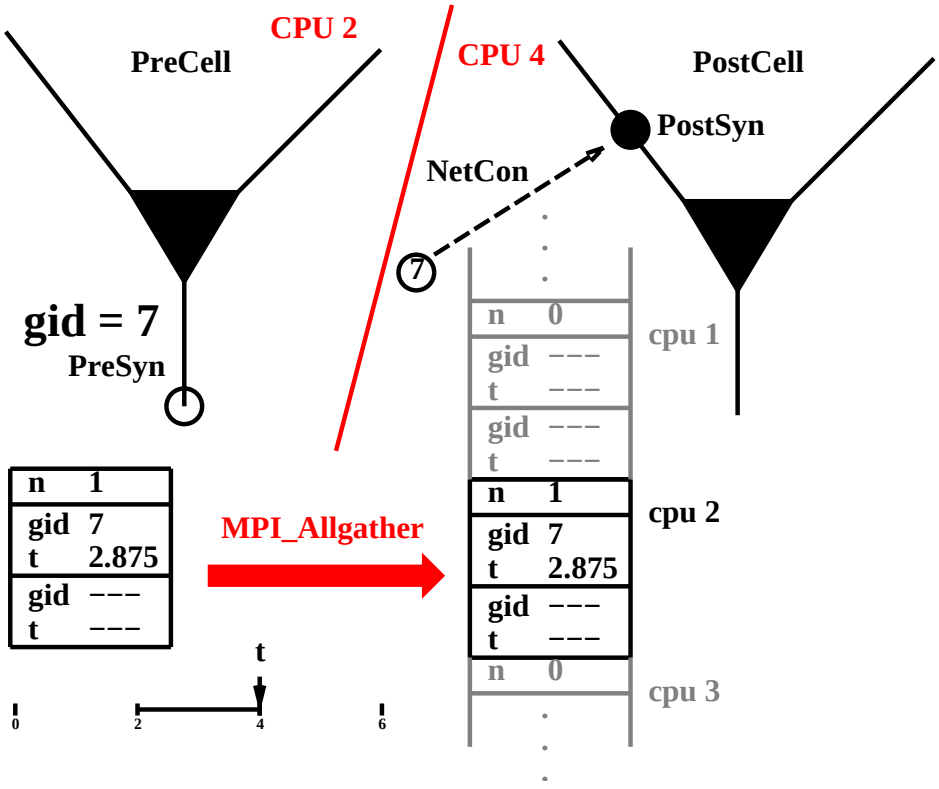
Run using the idiom

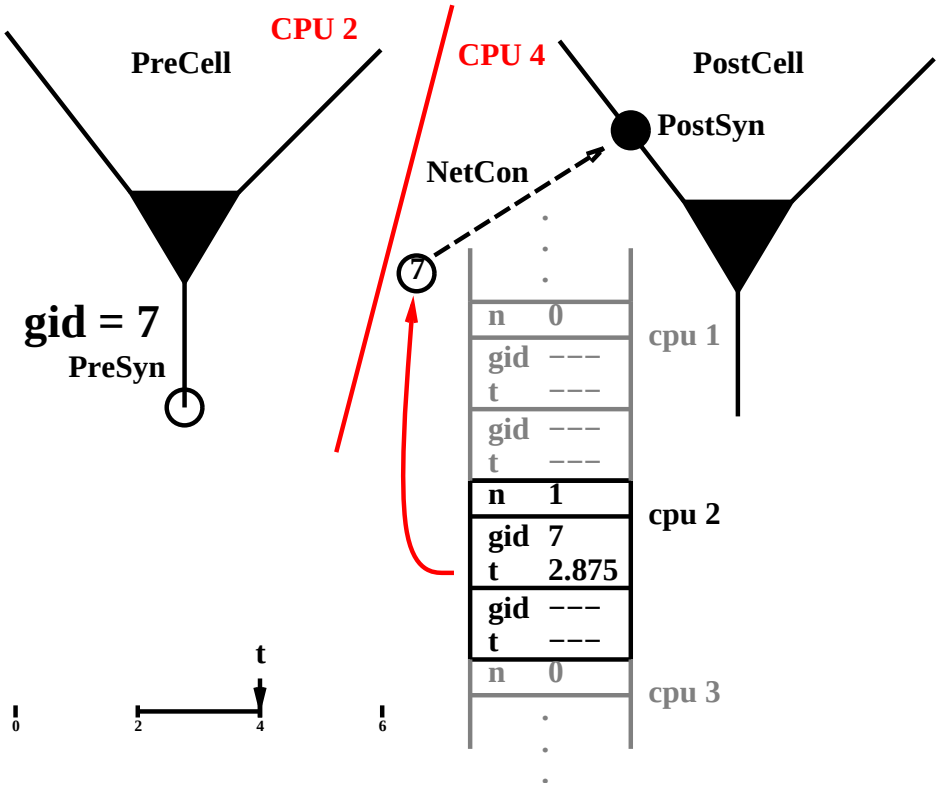
```
pc.set_maxstep(10)
stdinit()
pc.psolve(tstop)
```

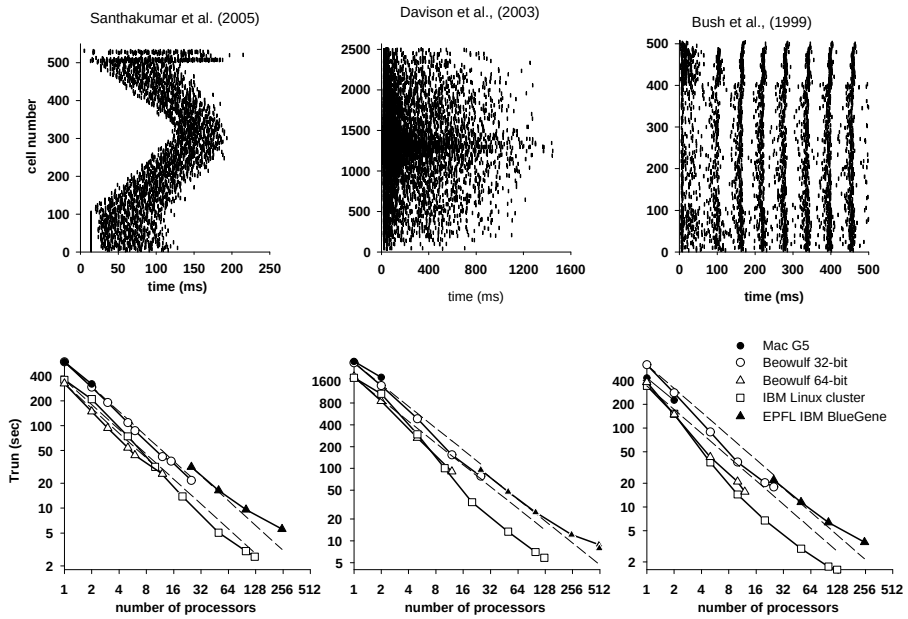
pc.set_maxstep() uses MPI_Allreduce to determine minimum delay.

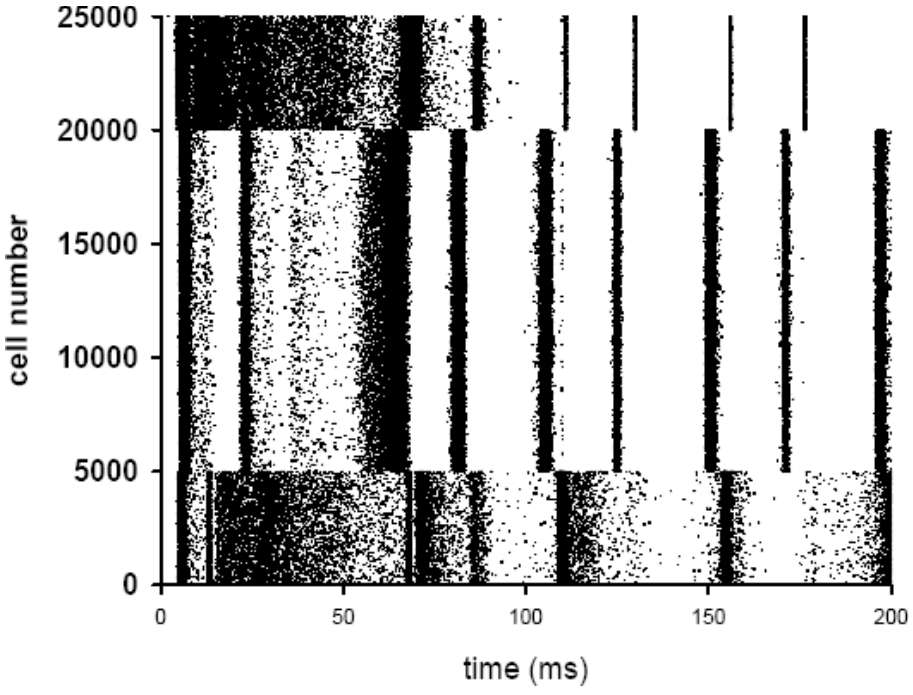


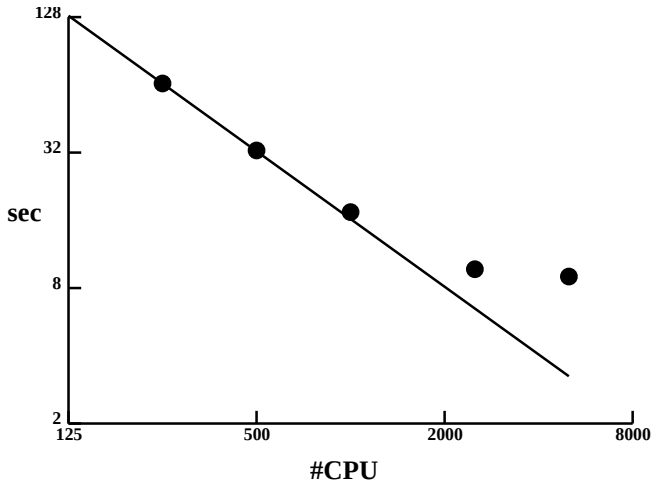




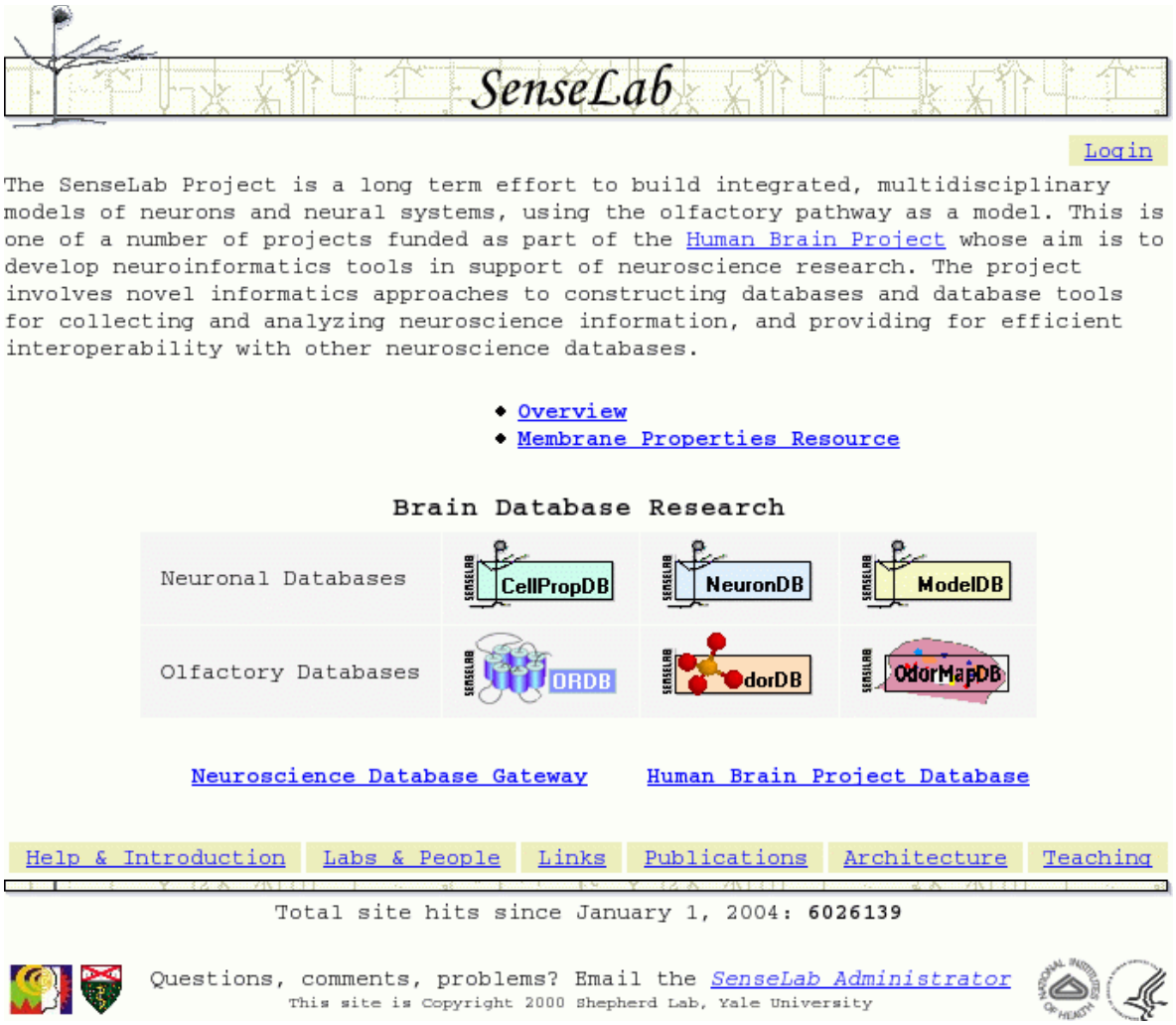








#CPU	Computation (s)	Communication (s)
1000	16.4	1.98
2500	6.3	3.42
5000	3.4	5.64



NeuronDB

Back to SenseLab

NeuronDB provides a dynamically searchable database of three types of neuronal properties: voltage gated conductances, neurotransmitter receptors, and neurotransmitter substances. It contains tools that provide for integration of these properties in a given type of neuron and compartment, and for comparison of properties across different types of neurons and compartments. Read the tutorial for searching for Neuron Properties in NeuronDB

This resource is intended to:

- Support the genomics and proteomics of neuron types
- Support research on neuron properties
- Facilitate the creation of computational neuronal models
- Identify receptors across neuron types to aid in drug development
- Serve as a teaching aid

Search the Database By:

- Neuron List Alphabetically
- Neuron List By Brain Regions
- Membrane Properties Comprehensive Inventory: Channels, Receptors, Neurotransmitters/Neuromodulators
- Membrane Properties for NeuronDB: Currents, Receptors, Neurotransmitters/Neuromodulators
- Canonical forms of neurons (see explanation)
- Bibliographic citations

- Signup to receive the NeuronDB email newsletter
- Give us your feedback
- Deposit to the Database
- FAQ and related Links
- NeuronDB Login

This database is being developed by Luis N. Marenco[2], Chiquito Crasto[2], Prakash M. Nadkarni[2], Perry L. Miller[2] and Gordon M. Shepherd[1], *[1]Section of Neurobiology, [2]Center for Medical Informatics, Yale University School of Medicine, New Haven, CT 06510.*

Some of the data, together with the graphics on the NeuronDB banner, are taken from The Synaptic Organization of the Brain, edited by G.M. Shepherd, New York: Oxford University Press (Second to Fourth Editions: 1979, 1990, 1998).

Supported by NIDCD, NASA, & NIMH (Human Brain Project) and the National Library of Medicine's IAIMS Program.

Total site hits since January 1, 2004: **130412**

Questions, comments, problems? Email the *NeuronDB Administrator*

Last Modified: August 02, 2004

NeuronDB

Back

Olfactory bulb mitral cell

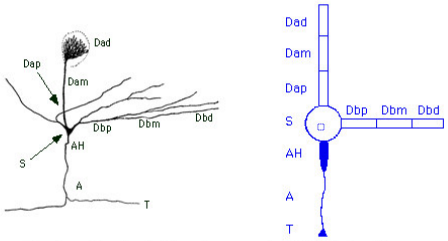
Mode: **Overview** Data/Search plus Connectivity plus References/Notes Models

Region: Dad Dam Dap Dbd Dbm Dbp Soma AH A T All Compartments

Properties: Receptors Channels Transmitters **All Properties**

Interoperation: Gene and Chromosome Experimental Data (neurodatabase.org) Microscopy Data (CCDB)

Neuron type: principal
Organism: Vertebrates



Key: Region: D, dendrite; S, soma (cell body); AH, axon hillock-initial segment of the axon; A, axon; T, axon terminal. Type of dendrite: e, equivalent cylinder (for single dendrites and multipolar trees); a, apical; b, basal; o, oblique. Level of dendrite: (p) proximal, (m) middle, and (d) distal with respect to the cell body. For further explanations, see canonical representations.

Graphic from: GM Shepherd, Synaptic Organization of the Brain, New York: Oxford University Press 1979.

Total site hits since January 1, 2004: **130414**

Questions, comments, problems? Email the *NeuronDB Administrator*

Last Modified: November 23, 2004

Back

Olfactory bulb mitral cell

Mode: Overview Data/Search plus Connectivity plus References/Notes Models

Region: Dad Dam Dap Dbd Dbm Dbp Soma AH A T All Compartments

Properties: Receptors Channels Transmitters All Properties

Interoperation: Gene and Chromosome Experimental Data (neurodatabase.org) Microscopy Data (CCDB)

Are: Present Absent

Neuron type: principal

Organism: Vertebrates

Dad

Input Receptors

Dopamine from periglomerular cell dendrite	Dopaminergic Receptor
DA receptors in mitral cell dendrites implied by DA localization in PG dendrites presynaptic to mitral dendrites (Halasz and et al, 1976[33]).	
Glutamate from olfactory receptor neuron axon terminals	AMPA
Intracellular recordings: CNQX blocks early component of EPSP elicited by olfactory nerve volley nerve volley (Berkowicz DA and Trombley PQ and Shepherd GM, 1994 [turtle][40]). Electrophysiology data: DNQX attenuates early and late excitatory components in peristimulus time histograms of mitral cell unit responses to olfactory nerve volleys (Ennis M and Zimmer LA and Shipley MT, 1996 [rat][20]). Intracellular recordings: CNQX blocks early component of EPSP response to olfactory nerve volley (Chen WR and Shepherd GM, 1997 [rat][16]). Paired whole-cell recording revealed reciprocal excitatory connections between mitral cells. Pharmacological analysis suggested that it could be mediated by both AMPA and NMDA receptors (Urban NN and Sakmann B, 2002[462]).	
Glutamate from olfactory receptor neuron axon terminals	NMDA
Intracellular recordings: AP5 blocks late component of EPSP elicited by olfactory nerve volley (Berkowicz DA and Trombley PQ and Shepherd GM, 1994 [turtle][40]). Electrophysiology data: AP5 attenuates delayed excitatory components in peristimulus time histograms of mitral cell unit responses to olfactory nerve volleys (Ennis M and Zimmer LA and Shipley MT, 1996 [rat][20]). Intracellular recordings: AP5 blocks late component of EPSP response to olfactory nerve. volley (Chen WR and Shepherd GM, 1997 [rat][16]). Paired whole-cell recording revealed reciprocal excitatory connections between mitral cells. Pharmacological analysis suggested that it could be mediated by both AMPA and NMDA receptors (Urban NN and Sakmann B, 2002[462]).	

Intrinsic Currents

I Na,t
Implied by recording of fast prepotential. Dual patch recordings provide evidence for both backpropagating and forward-propagating impulses in the primary dendrite (Mori K et al, 1982 [turtle][19] ; Chen et al 1997). Dendritic patch recordings showed an even density of Na channels (120pS um-2) up to 350 um from the soma along the primary dendrite to the origin of the glomerular tuft (Bischofberger J and Jonas P, 1997[397]). By combining intracellular recordings and two-photon microscopy imaging of [Ca]i in rat it was shown that APs backpropagate at full amplitude up to the tuft (Debarbieux F and Audinat E and Charpak S, 2003[462]).
I N
I A
(Bischofberger J and Jonas P, 1997[397]).

Output Transmitters

Glutamate	Mitral cell glomerular tuft and PG cell dendrites in the glomerulus (presumably)
Implied by Glu released by other compartments of the mitral cell (Dale's law). Target (destination) is presumably PG cell dendrites in the glomerulus (van den Pol AN, 1995 [rat][2]).	

Back to SenseLab

NeuronDB (in this context)

Models which contain: *Olfactory bulb mitral cell*

Models	Description
Olfactory Bulb Network (Davison et al 2003)	A biologically-detailed model of the mammalian olfactory bulb, incorporating the mitral and granule cells and the dendrodendritic synapses between them. The results of simulation experiments with electrical stimulation agree closely in most details with published experimental data. The model predicts that the time course of dendrodendritic inhibition is dependent on the network connectivity as well as on the intrinsic parameters of the synapses. In response to simulated odor stimulation, strongly activated mitral cells tend to suppress neighboring cells, the mitral cells readily synchronize their firing, and increasing the stimulus intensity increases the degree of synchronization. For more details, see the reference below.
Olfactory Mitral Cell (Bhalla, Bower 1993)	This is a conversion to NEURON of the mitral cell model described in Bhalla and Bower (1993). The original model was written in GENESIS and is available by joining
Olfactory Mitral Cell (Davison et al 2000)	A four-compartment model of a mammalian olfactory bulb mitral cell, reduced from the complex 286-compartment model described by Bhalla and Bower (1993). The compartments are soma/axon, secondary dendrites, primary dendrite shaft and primary dendrite tuft. The reduced model runs 75 or more times faster than the full model, making its use in large, realistic network models of the olfactory bulb practical.
Olfactory Mitral Cell (Shen et al 1999)	Mitral cell model with standard parameters for the paper: Shen, G.Y., Chen, W. R., Midtgaard, J., Shepherd, G.M., and Hines, M.L. (1999) Computational Analysis of Action Potential Initiation in Mitral Cell Soma and Dendrites Based on Dual Patch Recordings. Journal of Neurophysiology 82:3006. Contact Michael.Hines@yale.edu if you have any questions about the implementation of the model.
Olfactory Mitral Cell: I-A and I-K currents (Wang et al 1996)	NEURON mod files for the I-A and I-K currents from the paper: X.Y. Wang, J.S. McKenzie and R.E. Kemm, Whole-cell K+ currents in identified olfactory bulb output neurones of rats. J Physiol. 1996 490.1:63-77. Please see the readme.txt included in the model file for more information.
Olfactory Mitral cell: AP initiation modes (Chen et al 2002)	The mitral cell primary dendrite plays an important role in transmitting distal olfactory nerve input from olfactory glomerulus to the soma-axon initial segment. To understand how dendritic active properties are involved in this transmission, we have combined dual soma and dendritic patch recordings with computational modeling to analyze action-potential initiation and propagation in the primary dendrite.
Olfactory bulb granule cell: effects of odor deprivation (Saghatelyan et al 2005)	The model supports the experimental findings on the effects of postnatal odor deprivation, and shows that a ~10mV shift in the Na activation or a reduction in the dendritic length of newborn GC could independently explain the observed increase in excitability.
Olfactory bulb mitral cell: synchronization by gap junctions (Migliore et al 2005)	In a realistic model of two electrically connected mitral cells, the paper shows that the somatically-measured experimental properties of Gap Junctions (GJs) may correspond to a variety of different local coupling strengths and dendritic distributions of GJs in the tuft. The model suggests that the propagation of the GJ-induced local tuft depolarization is a major mechanim for intraglomerular synchronization of mitral cells.

Total site hits since January 1, 2004: **552289**

SenseLab ModelDB

Olfactory Mitral Cell (Shen et al 1999)

Mitral cell model with standard parameters for the paper: Shen, G.Y., Chen, W. R., Midtgaard, J., Shepherd, G.M., and Hines, M.L. (1999) Computational Analysis of Action Potential Initiation in Mitral Cell Soma and Dendrites Based on Dual Patch Recordings. Journal of Neurophysiology 82:3006. Contact Michael.Hines@yale.edu if you have any questions about the implementation of the model.
Reference: Shen GY, Chen WR, Midtgaard J, Shepherd GM, Hines ML (1999) Computational analysis of action potential initiation in mitral cell soma and dendrites based on dual patch recordings. *J Neurophysiol* **82**:3006-20 [PubMed]

Citations Citation Browser

Model Information (*Click on a link to find other models with that property*)

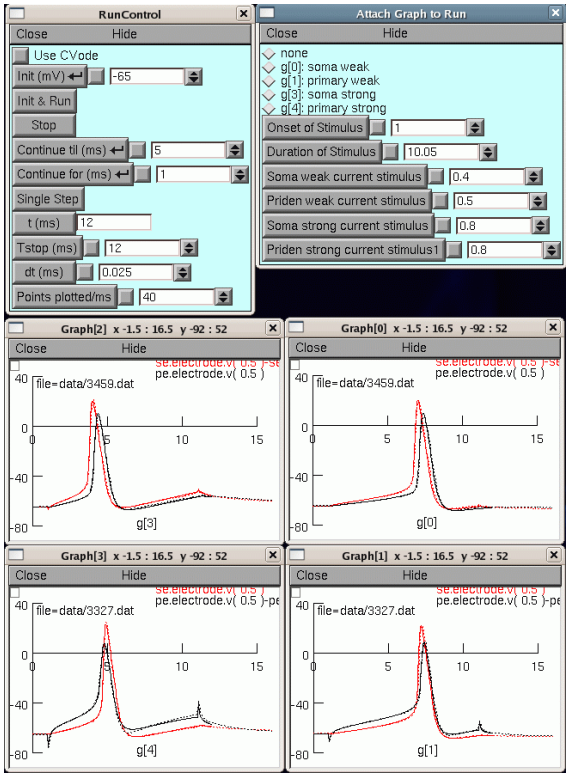
Model Type:	Neuron;
Cell Type(s):	Olfactory bulb mitral cell;
Channel(s):	I Na,t; I K; I Sodium; I Potassium;
Receptor(s):	
Transmitter(s):	
Simulation Environment:	Neuron;
Model Concept(s):	Action Potential Initiation; Dendritic Action Potentials; Parameter Fitting; Active Dendrites;
Implementer(s):	Hines, Michael ;

Search NeuronDB for information about: Olfactory bulb mitral cell; I Na,t; I K; I Sodium; I Potassium;

Model files Download zip file | Auto-launch | Help downloading and running models

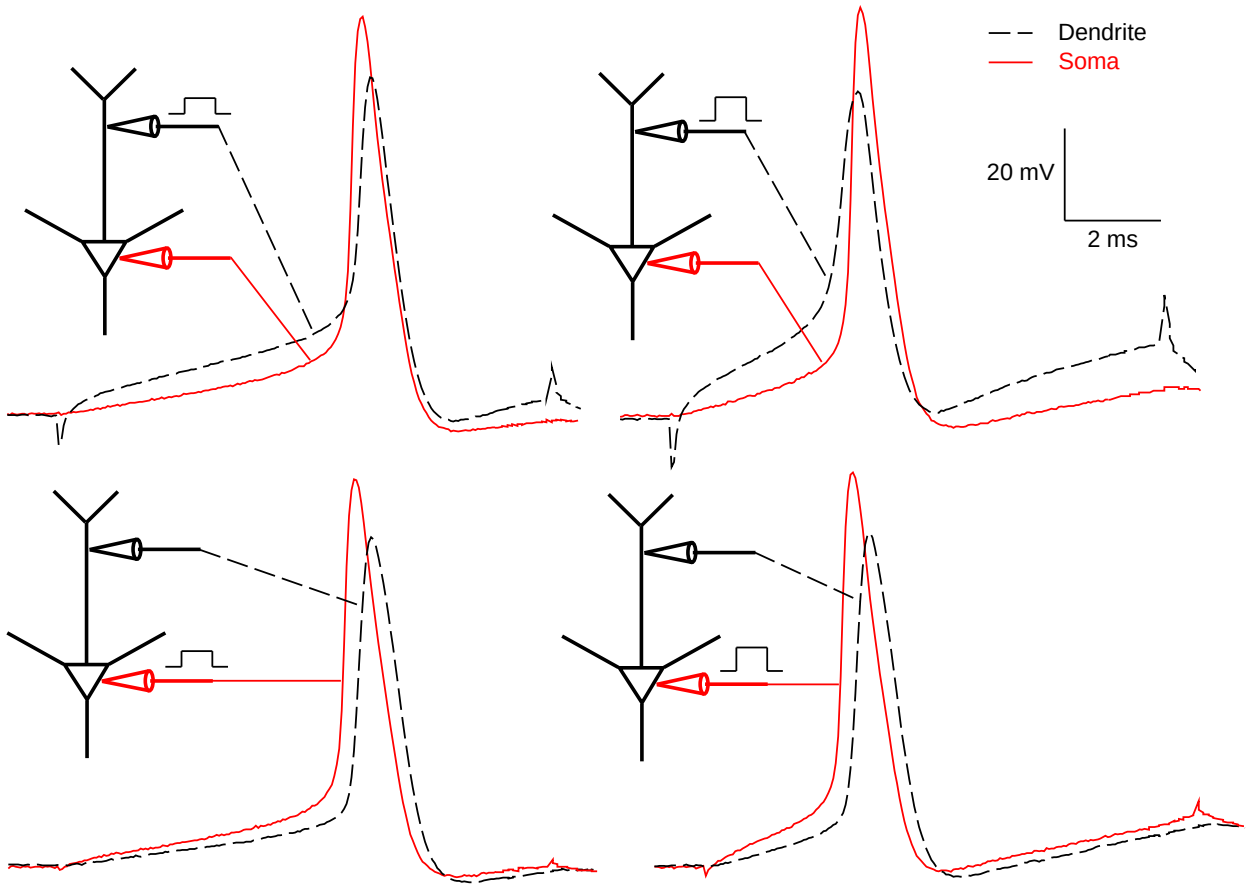
- \
  - mitral
    - cell2
    - data
    - XtraStuf.mac
    - README
    - kd.mod
    - na.mod
    - elec1.hoc
    - memb.hoc
    - mitral.hoc
    - mosinit.hoc
    - init.hoc
    - data_in.hoc

```
Mitral cell model with standard parameters for the paper:

Shen, G.Y., Chen, W. R., Midtgaard, J., Shepherd, G.M., and Hines, M.L.
(1999)
Computational Analysis of Action Potential Initiation in Mitral
Cell Soma and Dendrites Based on Dual Patch Recordings.
Journal of Neurophysiology  82: 3006

Questions about how to use this simulation should be
directed to
        michael.hines@yale.edu

Running the model (execution of the mosinit.hoc file) will display
the data and simulation results as in  Fig 3 and 5.

The cell2 subdirectory contains cell data and simulation which shows
only a limited decrease in the action potential interval
```



Multiple Run Fitter

Goal: Adjust model parameters to fit data from multiple experiments.



See: Shen et. al. (1999), J. Neurophysiol. 82:3006
Computational analysis of action potential initiation in mitral cell soma and dendrites based on dual patch recordings.

Optimizing over multiple experimental protocols

For each protocol:

Which model variables are to be compared to the data?

What is the data?

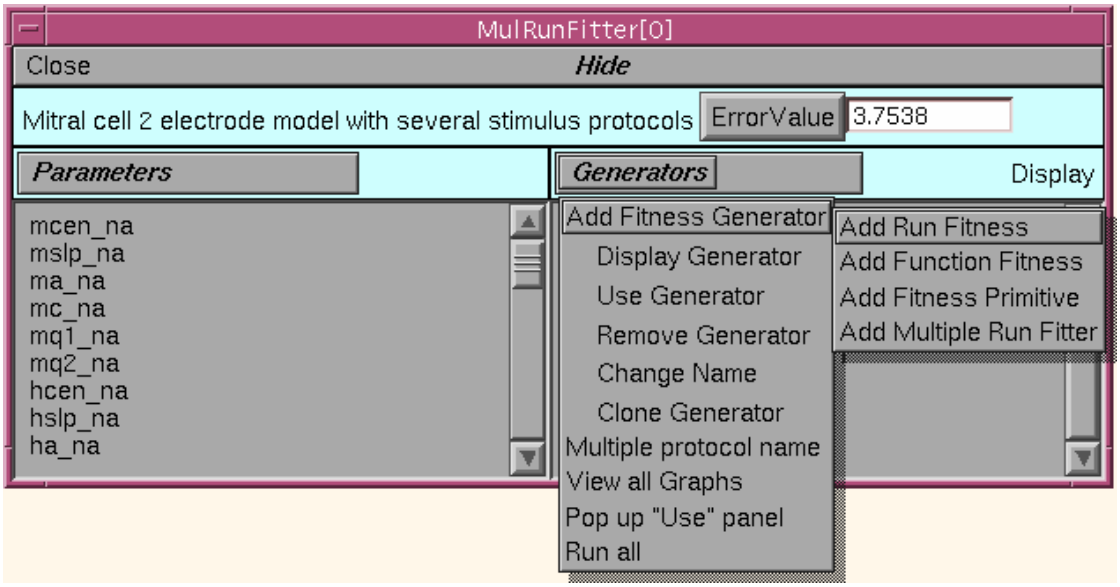
What is the error function?

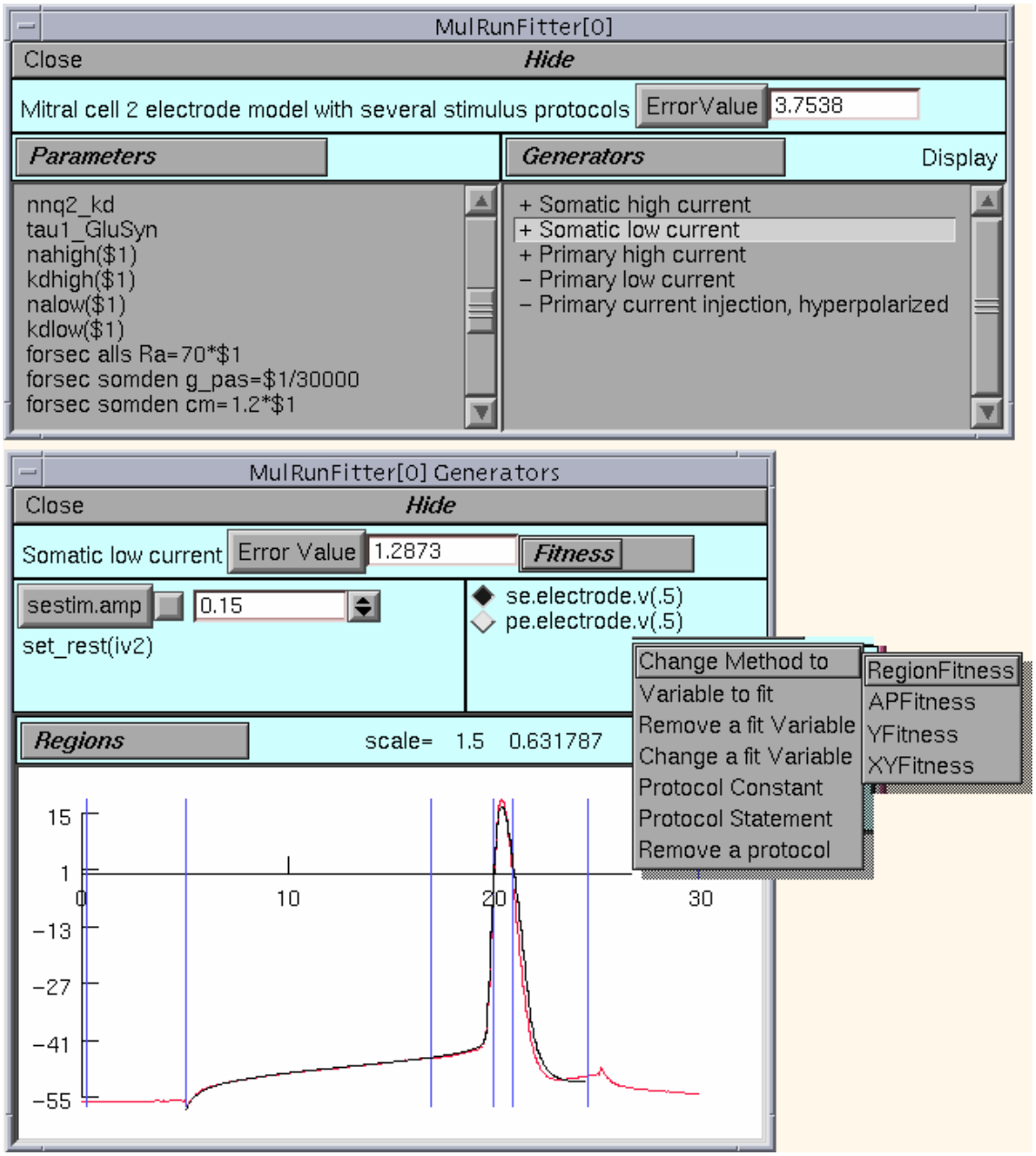
What is the stimulus protocol?

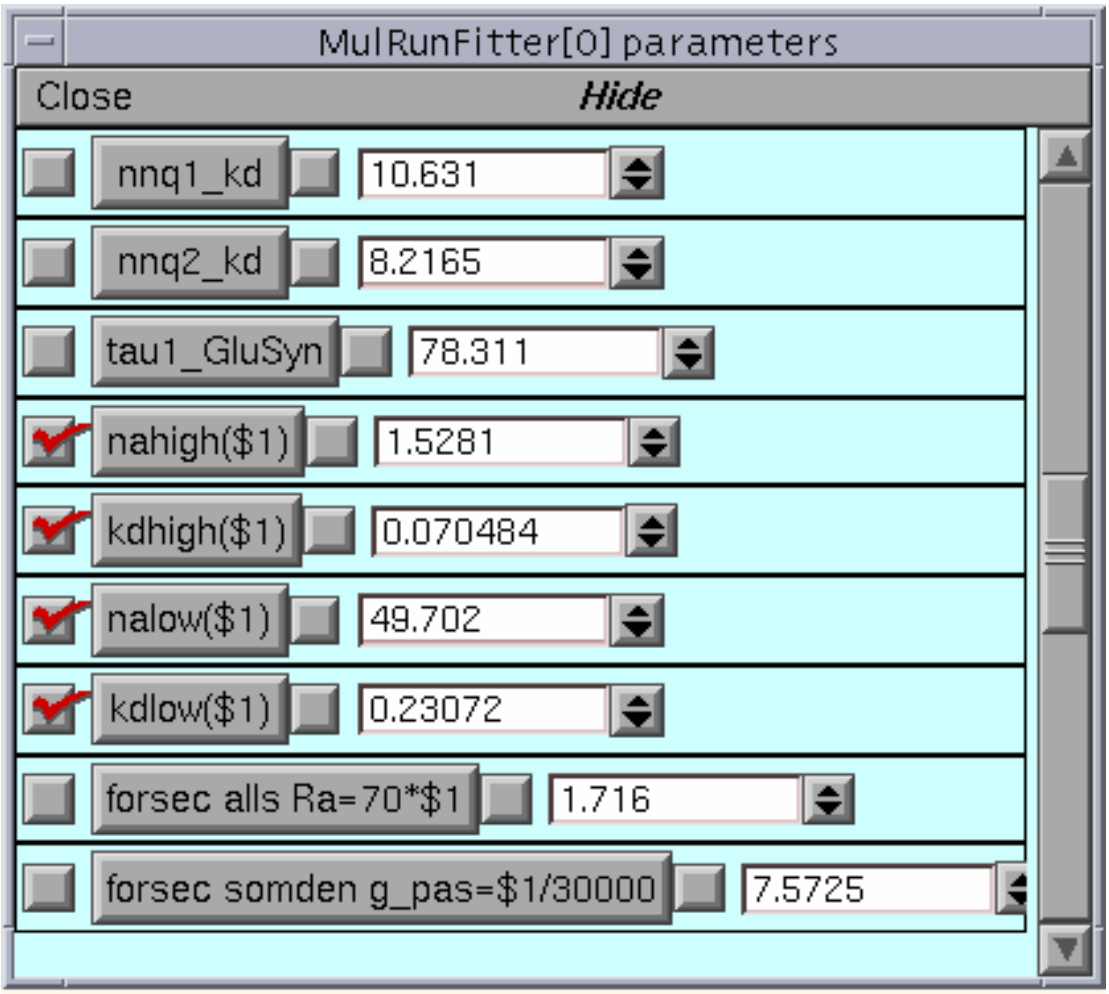
Parameters:

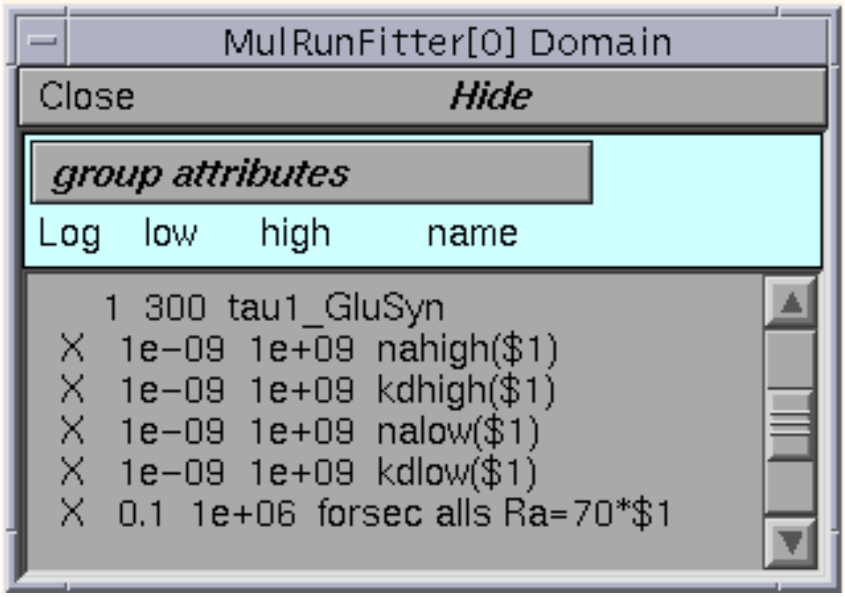
Which model parameters are you allowed to vary to obtain a fit.

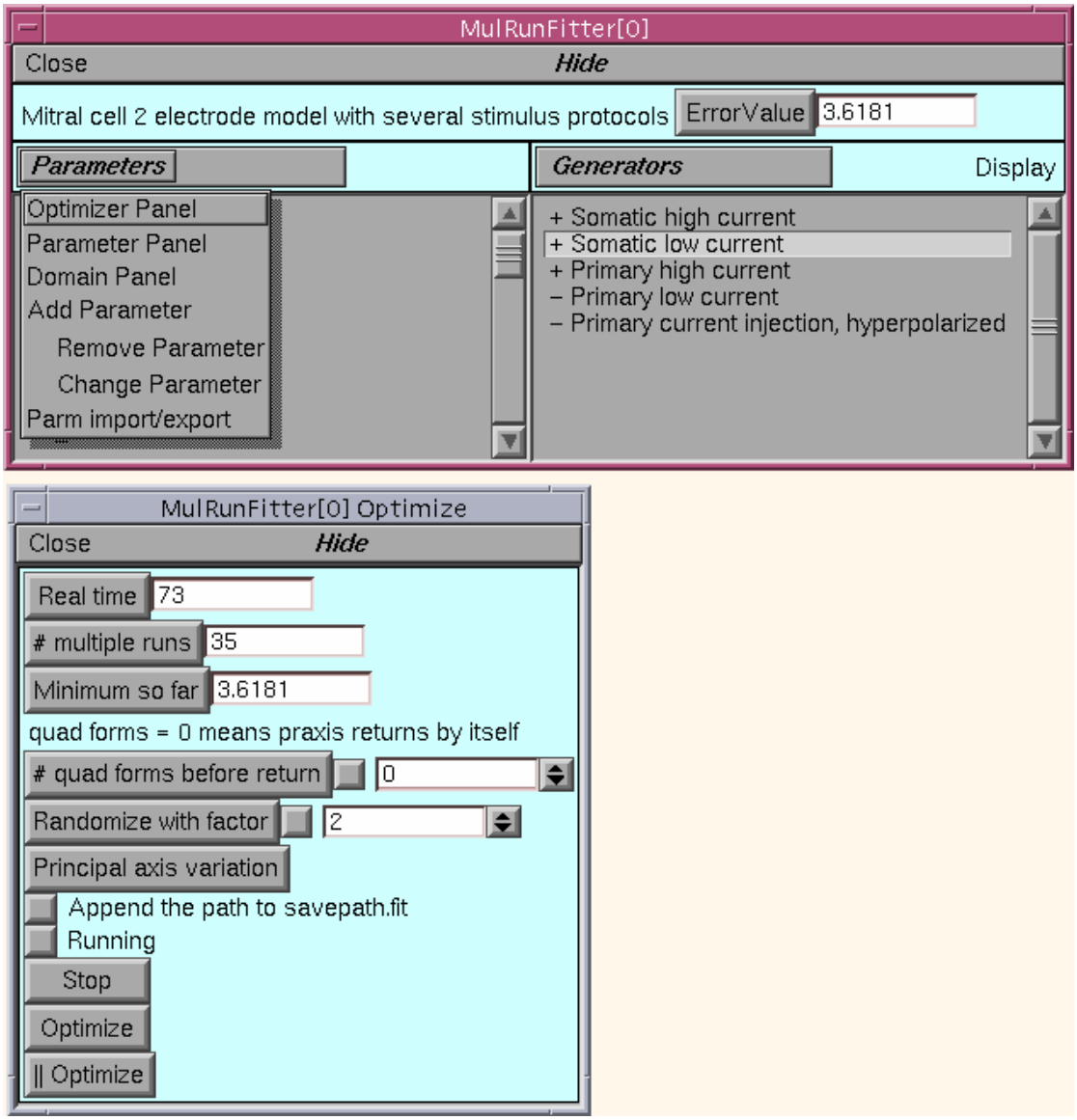
```
example.ses.ft1
---------------
ParmFitness:   Mitral cell 2 electrode model
    FitnessGenerator: Somatic high current
         RunConstant:    sestim.amp 0.3 0
         RunStatement:   1, set_rest(iv1)
         RegionFitness:  se.electrode.v(.5)
         RegionFitness:  pe.electrode.v(.5)

    FitnessGenerator: Primary high current
        RunConstant:    pestim.amp 0.4 0
        RunStatement:   1, set_rest(iv3)
        RegionFitness:  se.electrode.v(.5)
        RegionFitness:  pe.electrode.v(.5)

    FitnessGenerator: Primary current, hyperpolarize soma
        RunConstant:    pestim.amp 0 0
                        sestim.amp 0 0
                        sestim.del 0 1
                        sestim.dur 0 10
        APFitness:      se.electrode.v(.5)
        APFitness:      pe.electrode.v(.5)

    Parameters:
        "mcen_na",             -31.2258, -100,    20, 0, 0
        "nahigh($1)",           1.52812, 1e-9, 1e+09, 1, 1
        "kdhigh($1)",         0.0704842, 1e-9, 1e+09, 1, 1
        "nalow($1)",            49.7024, 1e-9, 1e+09, 1, 1
        "kdlow($1)",           0.230715, 1e-9, 1e+09, 1, 1
        "forsec alls Ra=70*$1",  1.716,  0.1, 1e+06, 0, 1
        "forsec axon g_pas=$1/1000", 1, 0.01,   100, 0, 0
        "iv1",                 -57.9528,  -67,   -50, 0, 0
        "iv4",                 -58.3375,  -67,   -50, 0, 0
        "pe.electrode.Ra=$1",   11.3141,  0.5,   100, 0, 0
        "pe.electrode.cm=$1",   10.5573, 0.01,    50, 0, 0
End ParmFitness
```









In the future...

- **Real−time Linux Dynamic Clamp with Gwendal LeMasson.**
- **Single neuron parallel with Maciej Lazarewicz.**
- **Re−vectorization.**
- **GUI tool for ionic accumulation models.**
- **Read/Write NeuroML.**

In the future...

- **WWW site: code from NEURON book, new tutorials, documentation wiki**
- **Courses: hands−on at UCSD, UMN**
- **NEURON Users' Group meeting TBA**

Chapter 6

How to build and use models of individual cells

In **Chapter 2** we remarked that a conceptual model is an absolute prerequisite for the scientific application of computational modeling. But if a computational model is to be a fair test of our conceptual model, we must take special care to establish a direct correspondence between concept and implementation. To this end, the research use of NEURON involves all of these steps:

1. Implement a computational model of the biological system
2. Instrument the model
3. Set up controls for running simulations
4. Save the model with instrumentation and run controls
5. Run simulation experiments
6. Analyze results

These steps are often applied iteratively. We first encountered them in **Chapter 1**, and we will return to each of them repeatedly in the remainder of this book.

GUI vs. hoc code: which to use, and when?

At the core of NEURON is an interpreter which is based on the hoc programming language (Kernighan and Pike 1984). In NEURON, hoc has been extended by the addition of many new features, some of which improve its general utility as a programming language, while others are specific to the construction and use of models of neurons and neural circuits in particular. One of these features is a graphical user interface (GUI) which provides graphical tools for performing most common tasks. We have already seen that many of these tools are especially useful for model development and exploratory simulations (**Chapter 1**).

Prior to the advent of the GUI, the only way to use NEURON was by writing programs in hoc. For many users, convenience is probably reason enough to use the GUI. We should also mention that several of the GUI tools are quite powerful in their own right, with functionality that would require significant effort for users to recreate by writing their own hoc code. This is particularly true of the tools for optimization and electrotonic analysis.

But sooner or later, even the most inveterate GUI user may encounter situations that call for augmenting or replacing the default implementations provided by the GUI. Traditional programming allows maximum control over model specification, simulation

control, and display and analysis of results. It is also appropriate for noninteractive simulations, such as "production" runs that generate large amounts of data for later analysis.

So the answer to our question is: use the GUI *and* write hoc code, in whatever combination gets the job done with the greatest conceptual clarity and the least human effort. Each has its own advantages, and the most productive strategy for working with NEURON is to combine them in a way that exploits their respective strengths. One purpose of this book is to help you learn what these strengths are.

Hidden secrets of the GUI

There aren't any, really. All but one of the GUI tools are implemented in hoc, and all of the hoc code is provided with NEURON (see `nrn-x.x/share/nrn/lib/hoc/` under UNIX/Linux, `c:\nrnxx\lib\hoc\` in MSWindows). Thus the CellBuilder, the Network Builder, and the Linear Circuit Builder are all implemented in hoc, and each of them works by executing hoc statements in a way that amounts to creating hoc programs "on the fly." It can be instructive to examine the source code for these and NEURON's other GUI tools. A recurring theme in many of them is a sequence of hoc statements that construct a string, followed by a hoc statement that executes this string (if it is a valid hoc statement) or uses it as an argument to some other hoc function or procedure. We will return to this idea in **Chapter 14: How to modify NEURON itself**, which shows how to create new GUI tools and add new functions to NEURON.

> The only GUI tool that is not implemented in hoc is the Print & File Window Manager, which is written in C. The source code for it is included with the UNIX distribution of NEURON.

Anything that can be done with a GUI tool can be done directly with hoc. To underscore this point, we will now use hoc statements to replicate the example that we built with the GUI in **Chapter 1**. Our code follows the same broad outline as before, specifying the model first, then instrumenting it, and finally setting up controls for running simulations. For clarity of presentation, we will consider this code in the same sequence: model implementation, instrumentation, and simulation control.

> If you want to work along with this example, it would be a good idea to create an empty directory in which to save the file or files that you will make. These will be plain text files, which are also sometimes known as ASCII files. Begin by using a text editor to create a file called `example.hoc` that will contain the code.

Implementing a model with hoc

The properties of our conceptual model neuron are summarized in Fig. 6.1 and Tables 6.1 and 6.2. For the most part, the steps required to implement a computational model of this cell with hoc statements parallel what we did to build the model with NEURON's GUI; differences will be noted and discussed as they arise. In the following program listings, single line comments begin with a pair of forward slashes `//` and multiple line

comments begin with `/*` and are terminated by `*/`. For a discussion of `hoc` syntax, see **Chapter 12**.

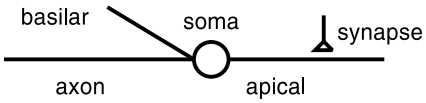


Fig. 6.1. The model neuron. The conductance change synapse can be located anywhere on the cell.

Table 6.1. Model cell parameters

	Length µm	Diameter µm	Biophysics
soma	30	30	HH g_{Na}, g_K, and g_{leak}
apical dendrite	600	1	passive with Rm = 5,000 Ω cm^2, E_{pas} = -65 mV
basilar dendrite	200	2	same as apical dendrite
axon	1000	1	same as soma

Cm = 1 µf / cm^2

cytoplasmic resistivity = 100 Ω cm

Temperature = 6.3 ^oC

Table 6.2. Synaptic mechanism parameters

g_{max}	0.05 µS
τ_s	0.1 ms
E_s	0 mV

Topology

Our first task is to map the branched architecture of this conceptual model onto the topology of the computational model. We want each unbranched neurite in the conceptual model to be represented by a corresponding section in the computational model, and this is done with a `create` statement (top of Listing 6.1). The `connect` statements attach these sections to each other so that the conceptual and computational models have the same shape. As we noted in **Chapter 5**, each section has a normalized position parameter which ranges from 0 at one end to 1 at the other. The `basilar` and `axon` sections arise from one end of the cell body while the `apical` section arises from the other, so they are attached by `connect` statements to the 0 and 1 ends of the `soma`, respectively.

This model is simple enough that its geometry and biophysical properties can be specified directly in `hoc` without having to resort to sophisticated strategies. Therefore we will not bother with subsets of sections, but proceed immediately to geometry.


```
//////////////////////
/* model specification */
//////////////////////

/////// topology ////////

create soma, apical, basilar, axon
connect apical(0), soma(1)
connect basilar(0), soma(0)
connect axon(0), soma(0)

/////// geometry ////////

soma {
  L = 30
  diam = 30
  nseg = 1
}

apical {
  L = 600
  diam = 1
  nseg = 23
}

basilar {
  L = 200
  diam = 2
  nseg = 5
}

axon {
  L = 1000
  diam = 1
  nseg = 37
}

/////// biophysics //////

forall {
  Ra = 100
  cm = 1
}

soma {
  insert hh
}

apical {
  insert pas
  g_pas = 0.0002
  e_pas = -65
}
```

```
basilar {
  insert pas
  g_pas = 0.0002
  e_pas = -65
}

axon {
  insert hh
}
```

Listing 6.1. The first part of `example.hoc` specifies the anatomical and biophysical attributes of our model.

Geometry

Each section of the model has its own length `L`, diameter `diam`, and discretization parameter `nseg`. The statements inside the block `soma { }` pertain to the `soma` section, etc. (the "stack of sections" syntax--see **Which section do we mean?** in **Chapter 5**). Since the emphasis here is on elementary aspects of model specification with `hoc`, we have assigned specific numeric values to `nseg` according to what we learned from prior use of the CellBuilder (see **Chapter 1**). A more general approach would be to wait until `L`, `diam`, and biophysical properties (`Ra` and `cm`) have been assigned, and then compute values for `nseg` based on a fraction of the AC length constant at 100 Hz (see **The d_lambda rule** in **Chapter 5**).

Biophysics

The biophysical properties of each section must be set up individually because we have not defined subsets of sections. Cytoplasmic resistivity `Ra` and specific membrane capacitance `cm` are supposed to be uniform throughout the model, so we use a `forall` statement to assign these values to each section.

The Hodgkin-Huxley mechanism `hh` and the passive mechanism `pas` are distributed mechanisms and are specified with `insert` statements (see **Distributed mechanisms** in **Chapter 5**). No further qualification is necessary for `hh` because our model cell uses its default ionic equilibrium potentials and conductance densities. However, the parameters of the `pas` mechanism in the `basilar` and `apical` sections differ from their default values, and so require explicit assignment statements.

Testing the model implementation

Testing is always important, especially when project development involves writing code. If you are working along with this example, this would be an excellent time to save what you have written to `example.hoc` and use NEURON to test it. Then, if you're using a Mac, just drag and drop `example.hoc` onto `nrngui`. Under MSWindows use Windows Explorer (the file manager, not Internet Explorer) to go to the directory where you saved `example.hoc` and double click on the name of the file. Under UNIX or Linux, type the command `nrniv example.hoc -` at the system prompt (we're

deliberately *not* typing `nrngui example.hoc`, to avoid having NEURON load its GUI library).

This will launch NEURON, and NEURON's interpreter will then process the contents of `example.hoc` and generate a message that looks something like this:

```
NEURON -- Version 5.6 2004-5-19 23:5:24 Main (81)
by John W. Moore, Michael Hines, and Ted Carnevale
Duke and Yale University -- Copyright 2001

oc>
```

The NEURON Main Menu toolbar will not appear under MSWindows, UNIX, or Linux. This happens because NEURON did not load its GUI library, which contains the code that implements the NEURON Main Menu. We're roughing it, remember? We trust that Mac users will pretend they don't see the toolbar, because dropping a `hoc` file on the `nrngui` icon automatically loads the GUI library.

Since we aren't using the CellBuilder, there isn't see a nice graphical summary of the model's properties. However a couple of `hoc` commands will quickly help you verify that the model has been properly specified.

We can check the branched architecture of our model by typing `topology()` at the `oc>` prompt (see **Checking the tree structure with topology()** in **Chapter 5**). This confirms that `soma` is the root section (i.e. the section that has no parent; note that this is *not* the same as the default section). It also shows that `apical` is attached to the 1 end of `soma`, and `basilar` and `axon` are connected to its 0 end.

```
oc>topology()

|-|       soma(0-1)
   `----------------------|       apical(0-1)
    `----|       basilar(0-1)
    `-----------------------------------|       axon(0-1)

      1
oc>
```

The command `forall psection()` generates a printout of the geometry and biophysical properties of each section. The printout is in the form of `hoc` statements that, if executed, will recreate the model.

```
oc>forall psection()
soma { nseg=1  L=30  Ra=100
   /*location 0 attached to cell 0*/
   /* First segment only */
   insert morphology { diam=30}
   insert capacitance { cm=1}
   insert hh { gnabar_hh=0.12 gkbar_hh=0.036 gl_hh=0.0003 el_hh=-54.3}
   insert na_ion { ena=50}
   insert k_ion { ek=-77}
}
```



```
apical { nseg=23  L=600  Ra=100
   soma connect apical (0), 1
   /* First segment only */
   insert capacitance { cm=1}
   insert morphology { diam=1}
   insert pas { g_pas=0.0002 e_pas=-65}
}
basilar { nseg=5  L=200  Ra=100
   soma connect basilar (0), 0
   /* First segment only */
   insert capacitance { cm=1}
   insert morphology { diam=2}
   insert pas { g_pas=0.0002 e_pas=-65}
}
axon { nseg=37  L=1000  Ra=100
   soma connect axon (0), 0
   /* First segment only */
   insert capacitance { cm=1}
   insert morphology { diam=1}
   insert hh { gnabar_hh=0.12 gkbar_hh=0.036 gl_hh=0.0003 el_hh=-54.3}
   insert na_ion { ena=50}
   insert k_ion { ek=-77}
}
oc>
```

After verifying that the model specification is correct, exit NEURON by typing `quit()` in the interpreter window.

An aside: how does our model implementation in hoc compare with the output of the CellBuilder?

The hoc code we have just written is supposed to set up a model that has the same anatomical and biophysical properties as the model that we created in **Chapter 1** with the CellBuilder. We can confirm that this is indeed the case by starting a fresh instance of NEURON, using it to load the session file that we saved in **Chapter 1**, and then typing `topology()` and `forall psection()`. But the CellBuilder can also create a file containing hoc statements that, when executed, recreate the model cell. How do the statements in this computer-generated file compare with the hoc code that we wrote for the purpose of specifying this model?

To find out, let us retrieve the session file from **Chapter 1**, and then select the Management page of the CellBuilder. Next we click on the Export button (Fig. 6.2), and save all the topology, subsets, geometry, and membrane information to a file called `cell.hoc`. Executing the hoc statements in this file will recreate the model cell that we specified with the CellBuilder.

It is instructive to briefly review the contents of `cell.hoc`, which are presented in Listing 6.2. At first glance this looks quite complicated, and its organization may seem a bit strange--after all, `cell.hoc` is a computer-generated file, and this might account for its peculiarities. But let him who has never written an idiosyncratic line of code cast the first stone! Actually, `cell.hoc` is fairly easy to understand if, instead of attempting a line-by-line analysis from top to bottom, we focus on the flow of program execution.

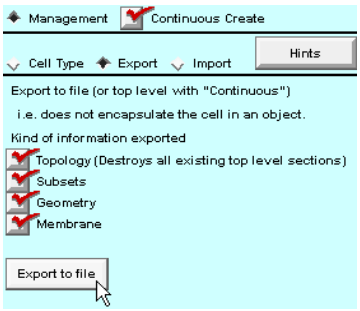


Figure 6.2. The Management page of the CellBuilder. We have clicked on the Export radio button, and are about to export the model's topology, subsets, geometry, and membrane information to a hoc file that can be executed to recreate the model cell.

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}

create soma, apical, basilar, axon

proc topol() { local i
  connect apical(0), soma(1)
  connect basilar(0), soma(0)
  connect axon(0), soma(0)
  basic_shape()
}

proc basic_shape() {
  soma {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(15, 0, 0, 1)}
  apical {pt3dclear() pt3dadd(15, 0, 0, 1) pt3dadd(75, 0, 0, 1)}
  basilar {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(-29, 30, 0, 1)}
  axon {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(-74, 0, 0, 1)}
}

objref all, has_HH, no_HH

proc subsets() { local i
  objref all, has_HH, no_HH
  all = new SectionList()
    soma all.append()
    apical all.append()
    basilar all.append()
    axon all.append()

  has_HH = new SectionList()
    soma has_HH.append()
    axon has_HH.append()
```



```
  no_HH = new SectionList()
    apical no_HH.append()
    basilar no_HH.append()
}

proc geom() {
  forsec all {  }
  soma {  L = 30  diam = 30  }
  apical {  L = 600  diam = 1  }
  basilar {  L = 200  diam = 2  }
  axon {  L = 1000  diam = 1  }
}

proc geom_nseg() {
  soma area(.5) // make sure diam reflects 3d points
  forsec all { nseg = int((L/(0.1*lambda_f(100))+.9)/2)*2 + 1  }
}

proc biophys() {
  forsec all {
    Ra = 100
    cm = 1
  }
  forsec has_HH {
    insert hh
      gnabar_hh = 0.12
      gkbar_hh = 0.036
      gl_hh = 0.0003
      el_hh = -54.3
  }
  forsec no_HH {
    insert pas
      g_pas = 0.0002
      e_pas = -65
  }
}

access soma

celldef()
```

Listing 6.2. The contents of `cell.hoc`, a file generated by exporting data from the CellBuilder that was used in **Chapter 1** to implement the model specified in Table 6.1 and 2 and shown in Fig. 6.1.

So we skip over the definition of `proc celldef()` to find the first statement that is executed:

```
create soma, apical, basilar, axon
```

Nothing too obscure about this. Next we jump over the definitions of two more `proc`s (the temptingly simple `topol()` and the slightly puzzling `basic_shape()`) before encountering a declaration of three `objref`s (see **Chapter 13: Object oriented programming**)

```
objref all, has_HH, no_HH
```


that are clearly used by the immediately following `proc subsets()` (what does it do? patience, all will be revealed . . .).

Finally at the end of the file we find a declaration of the default section, and then the procedure `celldef()` is called.

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}
```

This is the master procedure of this file. It invokes other procedures whose names remind us of that familiar sequence "topology, subsets, geometry, biophysics" before it ends with the eponymic `geom_nseg()`. Using `celldef()` as our guide, we can skim through the rest of the procedures.

- `topol()` first connects the sections to form the branched architecture of our model, and then it calls `basic_shape()`. The latter uses `pt3dadd` statements that are based on the shape of the stick figure that we saw in the CellBuilder itself. This establishes the orientations (angles) of sections, but the lengths and diameters will be superseded by statements in `geom()`, which is executed later.
- `subsets()` uses `SectionLists` to implement the three subsets that we defined in the CellBuilder (`all`, `has_HH`, `no_HH`).
- `geom()` specifies the actual physical dimensions of each of the sections.
- `biophys()` establishes the biophysical properties of the sections.
- `geom_nseg()` applies the discretization strategy we specified, which in this case is to ensure that no segment is longer than 0.1 times the length constant at 100 Hz (see **The d_lambda rule** in **Chapter 5**). This procedure is last to be executed because it needs to have the geometry and biophysical properties of the sections.

Instrumenting a model with hoc

The next part of `example.hoc` contains statements that set up a synaptic input and create a graphical display of simulation results (Listing 6.3). The synapse and the graph are specific instances of the `AlphaSynapse` and `Graph` classes, and are managed with object syntax (see **Chapter 13**). The synapse is placed at the middle of the `soma` and is assigned the desired time constant, peak conductance, and reversal potential. The graph will be used to show the time course of `soma.v(0.5)`, the somatic membrane potential.

The strategy for dealing with synapses depends on the nature of the model. They are treated as part of the instrumentation in cellular and subcellular models, and there is indeed a sense in which they can be regarded as "physiological extensions" of the stimulating apparatus. However, synapses between cells in a network model are clearly intrinsic to the biological system. This difference is reflected in the GUI tools for constructing models of individual cells and networks.


```
//////////////////////
/*   instrumentation   */
//////////////////////

///// synaptic input ////

objref syn
soma syn = new AlphaSynapse(0.5)
syn.onset = 0.5
syn.tau = 0.1
syn.gmax = 0.05
syn.e = 0

/// graphical display ///

objref g
g = new Graph()
g.size(0,5,-80,40)
g.addvar("soma.v(0.5)", 1, 1, 0.6, 0.9, 2)
```

Listing 6.3. The second part of `example.hoc` specifies the instrumentation used to stimulate and monitor our model.

Setting up simulation control with hoc

The code in the last part of `example.hoc` controls the execution of simulations. This code must accomplish many tasks. It must define the size of the time step and the duration of a simulation. It also has to initialize the simulation, which means setting time to 0, making membrane potential assume its proper initial value(s) throughout the model, and ensuring that all gating variables and ionic currents are consistent with these conditions. Furthermore, it has to advance the solution from beginning to end and plot the simulation results on the graph. Finally, if interactive use is important, initializing and running simulations should be as easy as possible.

Setting up simulation control is a recurring task in developing computational models, and much effort can be wasted trying to reinvent the wheel. For didactic purposes, in this example we create our own simulation control code *de novo*. However, it is always far more efficient to use the powerful, customizable functions and procedures that are built into NEURON's standard run system (see **Chapter 7**).

The code in Listing 6.4 accomplishes these goals for our simple example. Simulation initialization and execution are generally performed by separate procedures, as shown here; the sole purpose of the final procedure is to provide the minor convenience that simulations can be initialized and executed by merely typing the command `go()` at the `oc>` prompt.

The first three statements in Listing 6.4 specify the default values for the time step, simulation duration, and initial membrane potential. However, initialization doesn't actually happen until you invoke the `initialize()` procedure, which contains statements that set time, membrane potential, gating variables and ionic currents to their proper initial values. The main computational loop that executes the simulation (`while`

`(t<tstop) { }`) is in the `integrate()` procedure, with additional statements that make the plot of somatic membrane potential appear in the graph.

```
//////////////////////////
/* simulation control  */
//////////////////////////

dt = 0.025
tstop = 5
v_init = -65

proc initialize() {
   t = 0
   finitialize(v_init)
   fcurrent()
}

proc integrate() {
   g.begin()
   while (t<tstop) {
      fadvance()
      g.plot(t)
   }
   g.flush()
}

proc go() {
   initialize()
   integrate()
}
```

Listing 6.4. The final part of `example.hoc` provides for initialization and execution of simulations.

Testing simulation control

Use NEURON to execute `example.hoc` (a graph should appear) and then type the command `go()` (this should launch a simulation, and a trace will appear in the graph). Change the value of `v_init` to -60mV and repeat the simulation (at the `oc>` prompt type `v_init=-60`, then type `go()`). When you are finished, type `quit()` in the interpreter window to exit NEURON.

Evaluating and using the model

Now that we have a working model, we are almost ready to put it to practical use. We have already checked that its sections are properly connected, and that we have correctly specified their biophysical properties. Although we based the number of segments on `nseg` generated by the CellBuilder using the d_lambda rule, we have not really tested discretization in space or time, so some exploratory simulations to evaluate the spatial and temporal grid are advisable (see **Chapter 4** and **Choosing a spatial grid** in **Chapter 5**). Once we are satisfied with its accuracy, we may be interested in improving simulation speed, saving graphical and numerical results, automating simulations and

data collection, curve fitting and model optimization. These are somewhat advanced topics that we will examine later in this book. The remainder of this chapter is concerned with practical strategies for working with models and fixing common problems.

Combining hoc and the GUI

The GUI tools are a relatively "recent" addition to NEURON (recent is a relative term in a fast-moving field--would you believe 1995?) so many published models have been implemented entirely in `hoc`. Also, many long-time NEURON users continue to work quite productively by developing their models, instrumentation, and simulation control exclusively with `hoc`. Often the resulting software is elegantly designed and implemented and serves its original purpose quite well, but applying it to new research questions can be quite difficult if significant revision is required.

Some of this difficulty can be avoided by generic good programming practices such as modular design, in particular striving to keep the specifications of the model, instrumentation, and simulation control separate from each other (see **Elementary project management** below). There is also a large class of problems that would require significant programming effort if one starts from scratch, but which can be solved with a few clicks of the mouse by taking advantage of existing GUI tools. But what if you don't see the NEURON Main Menu toolbar, or (as often happens when you first start to work with a "legacy" model) you do see it but many of the GUI tools don't seem to work?

No NEURON Main Menu toolbar?

This is actually the easiest problem to solve. At the `oc>` prompt, type the command `load_file("nrngui.hoc")` and the toolbar should quickly appear. If you add this statement to the very beginning of the `hoc` file, you'll never have to bother with it again.

> `nrngui` also loads the standard run library

The toolbar will always appear if you use `nrngui` to load a `hoc` file. On the Mac this is what happens when you drag and drop a `hoc` file onto the `nrngui` icon. Under MSWindows you would have to start NEURON by clicking on its desktop `nrngui` icon (or on the `nrngui` item in the Start menu's NEURON program group), and then use NEURON Main Menu / File / load hoc to open the the `hoc` file. UNIX/Linux users can just type `nrngui` *`filename`* at the system prompt.

However, even if you see the toolbar, many of the GUI tools will not work if the `hoc` code didn't define a default section.

Default section? We ain't got no default section!

No badges, either. But to make full use of the GUI tools, you do need a default section. To see what happens if there isn't one, let's add a second synapse to the instrumentation of our example as if we were modeling feedforward inhibition. We could do this by writing `hoc` statements that define another point process, but this time let's use the GUI (see **4. Instrument the model. Signal sources** in **Chapter 1**).

First, change `example.hoc` by adding the statement

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

at the very beginning of the file. Now when NEURON executes the commands in `example.hoc`, the first thing that happens is the GUI library is loaded and the NEURON Main Menu toolbar appears.

UNIX/Linux users can go back to typing `nrngui example.hoc`.

But NEURON Main Menu / Tools / Point Processes / Managers / Point Manager doesn't work. Instead of a PointProcessManager we get an error message that there is "no accessed section" (Fig. 6.2). What went wrong, and how do we fix it?

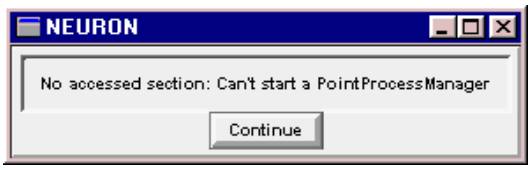


Fig. 6.2. A useful error message.

Many of the GUI tools, such as voltage graphs, shape plots, and point processes, must refer to a particular section at the moment they are spawned. This is because sections share property names, such as `L` and `v`. Remember the statement we used to create a point process in `example.hoc`:

```
soma syn = new AlphaSynapse(0.5)
```

This placed the newly created synapse at the 0.5 location on a particular section: the `soma`. But we're not writing `hoc` statements now; we're using a graphical tool (the NEURON Main Menu) to create another graphical tool that we will use to attach a point process to a section, and the NEURON Main Menu has no way to guess which section we're thinking about.

The way to fix this problem is to add the statement

```
access soma
```

to our model description, right after the `create` statement. The `access` statement defines the *default* section (see **Which section do we mean?** in **Chapter 5**). If we assign membrane properties or attach a point process to a model, the default section is affected unless we specify otherwise. And if we use the GUI to create a plot of voltage vs. time, `v` at the middle of the default section is automatically included in the list of things that are plotted.

If there are many sections, which one should be the default section? A good rule of thumb is to pick a conceptually privileged section that will get most of the use. The soma is generally a good choice.

So click on the "Continue" button to dismiss the error message, quit NEURON, add the `access soma` statement to `example.hoc`, and try again. This time it works. Configure the PointProcessManager to be an AlphaSynapse with onset = 0.5 ms, tau = 0.3 ms, gmax = 0.04 µS, and e = -70 mV and type

Scientific question: can you explain the effect of the inhibitory synapse's tau on cell firing?

go() to run a simulation. Run a couple more simulations with tau = 1 ms and 3 ms. Then exit NEURON.

Strange Shapes?

The barbed wire model

In **Chapter 1** we mentioned that the 3-D method for specifying geometry can be used to control the appearance of a model in a Shape plot. The benefits of the 3-D method for models based on detailed morphometric data are readily appreciated: the direct correspondence between the anatomy of the cell as seen under a microscope, and its representation in a Shape plot, can assist conceptual clarity when specifying model properties and understanding simulation results. Perhaps less obvious, but no less real, is the utility of the 3-D method for dealing with more abstract models, whose geometry is easy enough to specify in terms of L and diam. We hinted at this in the walkthrough of the hoc code exported by the CellBuilder, but a few examples will prove its value and at the same time help prevent misapplication and misunderstanding of this approach.

Suppose our conceptual model is a cell with an apical dendrite that gives rise to 10 oblique branches along its length. For the sake of visual variety, we will have the lengths of the obliques increase systematically with distance from the soma. Listing 6.5 presents an implementation of such a model using L and diam to specify geometry. The apical trunk is represented by the proximal section apical and the sequence of progressively more distal sections ap[0] - ap[NDEND-1]. With our mind's eye, aided perhaps by dim recollection of Ramon y Cajal's marvelous drawings, we can visualize the apical trunk stretching away from the soma in a more or less straight line, with the obliques coming off at an angle to one side.

```
/////// topology ////////

NDEND = 10

create soma, apical, dend[NDEND], oblique[NDEND]
access soma

connect apical(0), soma(1)
connect ap[0](0), apical(1)
connect oblique[0](0), apical(1)

for i=1,NDEND-1 {
  connect ap[i](0), ap[i-1](1)
  connect oblique[i](0), dend[i-1](1)
}

/////// geometry ////////

soma { L = 30 diam = 30 }

apical { L = 3 diam = 5 }
```



```
for i=0,NDEND-1 {
  ap[i] { L = 15 diam = 2 }
  oblique[i] { L = 15+5*i diam = 1 }
}
```

Listing 6.5. Implementation of an abstract model that has a moderate degree of dendritic branching using `L` and `diam` to specify geometry.

But executing this code and bringing up a Shape plot (e.g. by NEURON Main Menu / Graph / Shape plot) produces the results shown in Figure 6.3. So much for our mind's eye. Where did all the curvature of the apical trunk come from?

This violence to our imagination stems from the fact that stylized specification of model geometry says nothing about the orientation of sections. At every branch point, NEURON's internal routine for rendering shapes makes its own decision, and in doing so it follows a simple rule: make a fork with one child pointing to the left and the other to the right by the same amount relative to the orientation of the parent. Models with more complex branching patterns can look even stranger; if the detailed architecture of a real neuron is translated to simple `hoc` statements that assert nothing more than connectivity, length, and diameter, the resulting Shape may resemble a tangle of barbed wire.

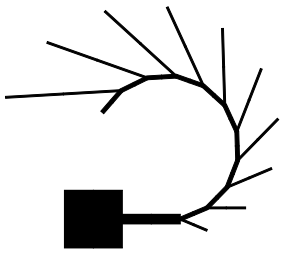


Fig. 6.3. Shape plot rendering of the model produced by the code in Listing 6.5. To help indicate the location of the soma section, Shape Style: Show Diam was enabled.

To gain control of the graphical appearance of our model, we must specify its geometry with the 3-D method. This is illustrated in Listing 6.6, where we have meticulously used absolute (x,y,z) coordinates, based on the actual location of each section, as arguments for the pt3dadd() statements. Now when we bring up a Shape plot, we get what we wanted: a nice, straight apical trunk with oblique branches coming off to one side (Fig. 6.4).

```
/////// geometry ////////

forall pt3dclear()

soma {
  pt3dadd(0, 0, 0, 30)
  pt3dadd(30, 0, 0, 30)
}

apical {
  pt3dadd(30, 0, 0, 5)
  pt3dadd(60, 0, 0, 5)
}
```



```
for i=0,NDEND-1 {
  ap[i] {
    pt3dadd(60+i*15, 0, 0, 2)
    pt3dadd(60+(i+1)*15, 0, 0, 2)
  }
  oblique[i] {
    pt3dadd(60+i*15, 0, 0, 1)
    pt3dadd(60+i*15, -15-5*i, 0, 1)
  }
}
```

Listing 6.6. Specification of model geometry using the 3-D method. This assumes the same model topology as shown in Listing 6.5.

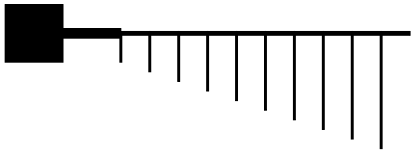


Fig. 6.4. Shape plot rendering of the model when the geometry is specified using the 3-D method shown in Listing 6.6.

Although we scrupulously used absolute (x,y,z) coordinates for each of the sections, we could have saved some effort by taking advantage of the fact that the root section is treated as the origin of the cell with respect to 3-D position. When any section's 3-D shape or length changes, the 3-D information of all child sections is translated to correspond to the new position. Thus, if the soma is the root section, we can move an entire cell to another location just by changing the location of the soma. Another useful implication of this feature allows us to simplify our model specification: the only `pt3dadd()` statements that must use absolute coordinates are those that belong to the root section. We can use relative coordinates for all child sections, instead of absolute (x,y,z) coordinates, as long as they result in proper length and orientation (see Listing 6.7).

```
/////// geometry ////////

forall pt3dclear()

soma {
  pt3dadd(0, 0, 0, 30)
  pt3dadd(30, 0, 0, 30)
}

apical {
  pt3dadd(0, 0, 0, 5)
  pt3dadd(30, 0, 0, 5)
}
```



```
for i=0,NDEND-1 {
  ap[i] {
    pt3dadd(0, 0, 0, 2)
    pt3dadd(15, 0, 0, 2)
  }
  oblique[i] {
    pt3dadd(0, 0, 0, 1)
    pt3dadd(0, -15-5*i, 0, 1)
  }
}
```

Listing 6.7. A simpler 3-D specification of model geometry that relies on the absolute coordinates of the root section and relative coordinates of all child sections. Compare the (x,y,z) coordinates in the `pt3dadd()` statements for `apical`, `ap`, and `oblique` with those in Listing 6.6.

The case of the disappearing section

In **Chapter 5** we mentioned that it is generally a good idea to attach the 0 end of a child section to its parent, in order to avoid confusion. For an example of one particularly vexing problem that can arise when this recommendation is ignored, consider Listing 6.8. The `access dend[0]` statement and the arguments to the `pt3dadd()` statements suggest that the programmer's conceptual model had the sections arranged in the left to right sequence `dend[0]` - `dend[1]` - `dend[2]`. Note that the 1 end of `dend[0]` is connected to the 0 end of `dend[1]`, and the 1 end of `dend[1]` is connected to the 0 end of `dend[2]`. This means that `dend[2]`, which is not connected to anything, is the root section. From a purely computational standpoint this is perfectly fine, and if we simulate the effect of a current step applied to the 0 end of `dend[0]`, there will be an orderly spread of charge and potential along each section from its 0 end to its 1 end, with the largest membrane potential shift in `dend[0]` and the smallest in `dend[2]`.

```
/////// topology ////////

NDEND = 3

create dend[NDEND]
access dend[0]

connect dend[0](1), dend[1](0)
connect dend[1](1), dend[2](0)

/////// geometry ////////

forall pt3dclear()

dend[0] {
  pt3dadd(0, 0, 0, 1)
  pt3dadd(100, 0, 0, 1)
}

dend[1] {
  pt3dadd(100, 0, 0, 1)
  pt3dadd(200, 0, 0, 1)
}
```



```
dend[2] {
  pt3dadd(200, 0, 0, 1)
  pt3dadd(300, 0, 0, 1)
}
```

Listing 6.8. The programmer's intent seems to be for `dend[0]`, `dend[1]`, and `dend[2]` to line up from left to right. However, the connect statements make `dend[2]` the root section, and thereby hangs a tale.

However, we're in for a surprise when we bring up a PointProcessManager (NEURON Main Menu / Tools / Point Processes / Managers / Point Manager) and try to place an `IClamp` at different locations in this model. No matter where we click, we can only put the `IClamp` on `dend[0]` or `dend[2]` (Fig. 6.5). Try as we might to find it, there just doesn't seem to be any `dend[1]`!

But `dend[1]` really does exist, and we can easily prove this by invoking the `topology()` function, which generates this diagram:

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

This not only confirms the existence of `dend[1]`, but also shows that `dend[2]` is the root section, with the 1 end of `dend[1]` connected to its to the 0 end, and the 1 end of `dend[0]` connected to the 0 end of `dend[1]`. Exactly as we expected, and just as specified by the code in Listing 6.8.

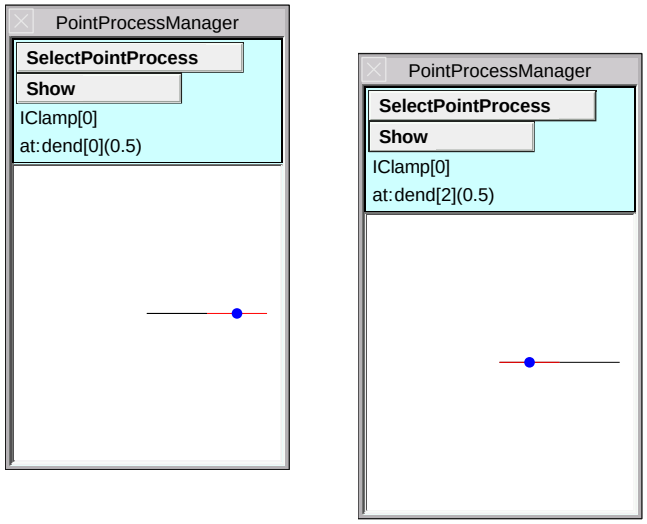


Fig. 6.5. The code in Listing 6.8 produces a model that seems not to have a `dend[1]`--or at least, we can't find `dend[1]` when we try to use a PointProcessManager to attach an `IClamp` to it.

But isn't something terribly wrong with the appearance of our model in the Shape plot? Not at all. Although we might not like it, the model looks exactly as it should, given the statements in Listing 6.8.

Here's why. As we mentioned above in **The barbed wire model**, the location of the root section determines the placement of all other sections. The root section is `dend[2]`, and the `pt3dadd()` statements in Listing 6.8 place its 0 end at (200, 0, 0) and its 1 end at (300, 0, 0) (Fig. 6.6).

Since `dend[1]` is attached to the 0 end of `dend[2]`, the first 3-D data point of `dend[1]` is mapped to (200, 0, 0) (see **3-D specification** in **Chapter 5**). According to the `pt3dadd()` statements for `dend[1]`, its last 3-D data point lies 100 μm to the right of its first 3-D point. This means that the 1 end of `dend[1]` is at (200, 0, 0) and its 0 end is at (300, 0, 0) (Fig. 6.6)--precisely the locations of the left and right ends of `dend[2]`! So `dend[1]` and `dend[2]` will appear as the same line in the Shape plot. When we try to select one of these sections by clicking on this line, the section we get will depend on the inner workings of NEURON's GUI library. It just happens that, for the particular `hoc` statements in Listing 6.8, we can only select points on `dend[2]`. This is as if `dend[1]` is hidden from view and shielded from our mouse cursor.

Finally we consider `dend[0]`, whose 1 end is connected to the 0 end of `dend[1]`. Thus its first 3-D data point is drawn at (300, 0, 0), and, following its `pt3dadd()` statements, its last 3-D data point lies 100 μm to the right, i.e. at (400, 0, 0). Thus `dend[0]` runs from (400, 0, 0) (its 0 end) to (300, 0, 0) (its 1 end), which is just to the right of `dend[2]` and the hidden `dend[1]` (Fig. 6.6).

So the mystery is solved. All three sections are present, but two are on top of each other.

The first lesson to take from this sad tale is the usefulness of `topology()` as a means for diagnosing problems with model architecture. The second lesson is the importance of following our recommendation to avoid confusion by connecting the 0 end of a child section to its parent. The strange appearance of the model in the Shape plot happened entirely because this advice was not followed. There are probably occasions in which it makes excellent sense to violate this simple rule; please be sure to let us know if you find one.

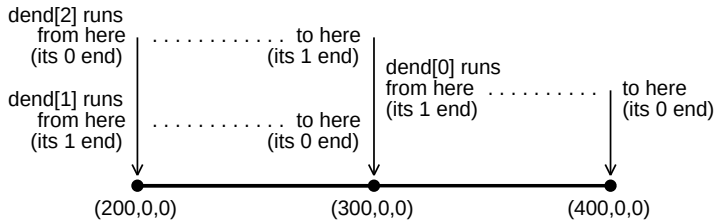


Fig. 6.6. Deciphering the `pt3dadd()` statements in Listing 6.8 leads us to realize that we only see two sections in the Shape plot because two of them (`dend[1]` and `dend[2]`) are drawn in the same place. This figure shows the (x,y,z) coordinates of the sections and indicates their 0 and 1 ends.

Graphs don't work?

If there is no default section, new graphs created with the GUI won't work properly. You've already seen how to declare the default section, so everything should be OK, right? Let's see for ourselves.

Make sure that `example.hoc` starts with `load_file("nrngui.hoc")` and contains an `access soma` statement, and then use NEURON to execute it. Then follow the steps shown in Fig. 1.27 (see **Signal monitors** in **Chapter 1**) to create a space plot that will show membrane potential along the length of the cell. Now type `go()`. What happens?

The graph of `soma.v(0.5)` shows an action potential, but the trace in the space plot remains a flat line. Is there something wrong with the space plot, or does the problem lie elsewhere?

To find out, use NEURON Main Menu / Tools / RunControl to bring up a RunControl window. Click on the RunControl's Init & Run button. Result: this time it's the space plot that works, and the graph of `soma.v(0.5)` that doesn't (Init & Run should have erased the trace in the latter and drawn a new one).

So there are actually two problems. The simulation control code in our `hoc` file can't update new graphs that we create with the GUI, and the GUI's own simulation control code can't update the "old" graph that is created by our `hoc` file. Of the many possible ways to deal with these problems, one is ridiculously easy and another requires a little effort (but only a very little).

The ridiculously easy solution is to use the GUI to make a new graph that shows the same variables, and ignore or throw away the old graph. In this example, resorting to NEURON Main Menu / Graph / Voltage axis gets us a new graph. Since the `soma` is the default section, the `v(.5)` that appears automatically in our new graph is really `soma.v(0.5)`.

What if a lot of programming went into one or more of the old graphs, so the GUI tools offer nothing equivalent? This calls for the solution that requires a little effort: specifically, we add a single line of `hoc` code for each old graph that needs to be fixed. In this example we would revise the code that defines the old graph by adding the line shown here in bold:

```
/// graphical display ///

objref g
g = new Graph()
addplot(g, 0)
g.size(0,5,-80,40)
g.addvar("soma.v(0.5)", 1, 1, 0.6, 0.9, 2)
```

Listing 6.9. Fixing an old graph so it works with NEURON's standard run system.

This takes advantage of NEURON's *standard run system*, a set of functions and procedures that orchestrate the execution of simulations (see **Chapter 7**). The statement `addplot(g, 0)` adds `g` to a list of graphs that the standard run system automatically

updates during the course of a simulation. Also, the x-axis of our graph will be adjusted automatically when we change `tstop` (Tstop in the RunControl panel). NEURON's GUI relies heavily on the standard run system, and every time we click on the RunControl's Init & Run button we are actually invoking routines that are built into the standard run system.

> The standard run system has many powerful features and can be used in any simulation, with or without the GUI. The statement `load_file("stdrun.hoc")` loads the `hoc` code that implements the standard run system, without loading the GUI.

Does this mean that we have to abandon the simulation control code in our `hoc` program, and does it matter if we do? The control code in `example.hoc` performs a "plain vanilla" initialization and simulation execution, so abandoning it in favor of the standard run system only makes things better by providing additional functionality. But what if we want a customized initialization or some unusual flow of simulation execution? As we shall see in **Chapter 7**, the standard run system was designed and implemented so that only minor changes are required to accommodate most special needs.

Conflicts between `hoc` code and GUI tools

Many of the GUI tools specify properties of the model or the interface, and this leads to the possibility of conflicts that cause a mismatch between what you *think* is in the computer, and what *actually* is in the computer. For example, suppose you use the CellBuilder to construct a model cell with a section called `dend` that has `diam` = 1 μm, `L` = 300 μm, and passive membrane, and you turn Continuous create ON. Then typing `dend psection()` at the `oc>` prompt will produce something like this

```
oc>dend psection()
dend { nseg=11  L=300  Ra=80
       . . .
    insert pas { g_pas=0.001 e_pas=-70}
}
```

(a few lines were omitted for clarity), which confirms the presence of the `pas` mechanism.

A bit later, you decide to make `dend` active and get rid of its `pas` mechanism. You could do this with the CellBuilder, but let's say you find it quicker just to type

```
oc>dend {uninsert pas insert hh}
```

and then confirm the result of your action with another `psection()`

```
oc>dend psection()
dend { nseg=11  L=300  Ra=80
       . . .
    insert hh { gnabar_hh=0.12 gkbar_hh=0.036 gl_hh=0.0003 el_hh=-54.3}
    insert na_ion { ena=50}
    insert k_ion { ek=-77}
}
```

So far, so good.

But check the Biophysics page of the CellBuilder, and you will see that the change you accomplished with hoc did not track back into the GUI tool, which still shows dend as having pas but not hh. This is particularly treacherous, because it is all too easy to become confused about what is the actual specification of the model. If these new biophysical properties lead to particularly interesting simulation results, you might save "everything" to a session file, thinking that you would be able to reproduce those results in the future--but the session file would only contain the state of the GUI tools. Completely absent would be any reflection of the fact that you had executed your own hoc statement to override the CellBuilder's model specification.

And still more surprises are in store. Using the CellBuilder, with Continuous create still ON, change dendritic diameter to 2 μm. Now use psection() to check the properties of dend

```
oc>dend psection()
dend { nseg=7  L=300  Ra=80
        . . .
    insert hh { gnabar_hh=0.12 gkbar_hh=0.036 gl_hh=0.0003 el_hh=-54.3}
    insert na_ion { ena=50}
    insert k_ion { ek=-77}
    insert pas { g_pas=0.001 e_pas=-70}
}
```

and you see that *both* pas and hh are present, despite the previous use of uninsert to get rid of the pas mechanism.

Similar conflicts can arise between hoc statements and other GUI tools (e.g. the PointProcessManager) All of these problems have a common source: changes you make at the hoc level are not propagated to the GUI tools, so if you then make any changes with the GUI tools, it is likely that all the changes you made with hoc statements will be lost. The lesson here is to exercise great caution when combining GUI tools and hoc statements, in order to avoid introducing potentially confusing conflicts.

Conflicts may also occur between the CellBuilder and older GUI tools for managing section properties.

Elementary project management

The example used in this chapter is simple so all of its code fits in a single, small file that can be quickly understood. Nonetheless, we were careful to organize example.hoc in a way that separates specification of the model *per se* from the specification of the interface, i.e. the instrumentation and control procedures for running simulations. This separation maximizes clarity and reduces effort, and it should begin while the model is still in the conceptual stage.

Designing a model starts by answering the questions: what anatomical features are necessary, and what biophysical properties should be included? The answers to these questions govern key decisions about what what kinds of stimuli to apply, what kinds of measurements to make, and how to display, record, and analyze these measurements. When it is finally time to implement the computational model, it is a good idea to try to

keep these questions separate. This is the way NEURON's graphical tools are organized, and this is the way models specified with hoc should be organized.

- First you create a model, specifying its topology, geometry, and biophysics, either with the CellBuilder or with hoc code. This is a representation of selected aspects of a biological system, and you might think of it as a virtual experimental preparation.
- Then you instrument that model. This is analogous to applying stimulating and recording electrodes and other apparatus to a real neuron or neural circuit in the laboratory.
- Finally, you set up controls for running simulations.

Instrumentation and simulation controls are the user interface for exercising the model. Metaphorically speaking, they amount to a virtual experimental rig. In a wet lab, noone would ever confuse a brain slice with the microscope or instrumentation rack. The physical and conceptual distinction between biological preparation and experimental rig them is an inescapable fact and has a strong bearing on the and execution of experiments. NEURON lets you carry this separation over into modeling. Why confound the code that defines the properties of a model cell with the code that generates a stimulus or governs the sequence of events in a simulation?

One way to help separate model specification from user interface is to put the code that defines them into separate files. One file, which we might call `cell.hoc`, would contain the statements that specify the properties of the model: its topology, geometry, and biophysics. The code that defines point processes, graphs, other instrumentation, and simulation controls would go into a second file that we might call `rig.hoc`. Finally, we would use a third file for purely administrative purposes, so that a single command will make NEURON execute the other files in proper sequence. This file, which we might call `init.hoc`, would contain only the statements shown in Listing 6.10. Executing `init.hoc` with NEURON will make NEURON load its GUI and standard run libraries, bring up a NEURON Main Menu toolbar, execute the statements in `cell.hoc` to reconstitute the model cell, and finally execute the statements in `rig.hoc` to reconstitute our user interface for exercising the model.

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

Listing 6.10. Contents of `init.hoc`.

For instance, we could recast `example.hoc` in this manner by putting its model specification component into `cell.hoc`, while the instrumentation and simulation control components would become `rig.hoc`. This would allow us to reuse the same model specification with different instrumentation configurations `rig1.hoc`, `rig2.hoc`, etc.. To make it easy to select which rig is used, we could create a corresponding series of `init` files (`init1.hoc`, `init2.hoc`, etc.) that differ only in the argument to the third `load_file()` statement. This strategy is not limited to hoc files, but can also be used to retrieve cells and/or interfaces that have been constructed with the GUI and saved to session (`ses`) files.

Iterative program development

A productive strategy for program development in NEURON is to revise and reinterpret `hoc` code and/or GUI tools repeatedly during the same session. Bugs afflict all nontrivial programs, and the process of making incremental changes, saving them to intermediate `hoc` or `ses` files, and testing at each step, reduces the difficulty of trying to diagnose and eliminate them. In this way it is possible begin with a program skeleton that consists of one or two `hoc` files with a handful of `load_file()` statements and function stubs, and quickly refine it until everything works properly. However, two caveats do apply.

First, a variable cannot be declared with a new type during the same session. In other words, "once a scalar, always a scalar" (or double, or string, or object reference). Attempting to redeclare a variable will produce an error message, e.g.

```
oc>x = 3
first instance of x
oc>objref x
/usr/local/nrn/i686/bin/nrniv: x  already declared near line 2
objref x
       ^
oc>
```

Trying to redefine a double, string, or object reference as something else will likewise fail. This is generally of little consequence, since it is rarely absolutely necessary to change the type assigned to a particular variable name. When this does happen, you just have to exit NEURON, make your changes to the `hoc` code, and restart.

The second caveat is that, once the `hoc` interpreter has parsed the code in a template (see **Chapter 13: Object-oriented programming**), the class that it defines is fixed for that session. This means that any changes to a template require exiting NEURON and restarting. The result is some inconvenience when developing and testing new classes, but this is still easier than having to recompile and link a program in C or C++.

References

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

Chapter 6 Index

Chapter 8

How to initialize simulations

In most cases, initialization basically means the assignment of values at time $t = 0$ for membrane potential, gating states, and ionic concentrations at every spatial position in the model. A model is properly initialized when clicking on the Init & Run button produces exactly the same results, regardless of previous simulation history. Of course we assume that model parameters have not changed between runs, and that any random number generator has been re-initialized with the same seed so that it produces the same sequence of "random" numbers. Models described by kinetic schemes require that each of the reactant states be initialized to some concentration. If linear circuits are involved, initial values must be assigned to voltages across capacitors and the internal states of operational amplifiers. For networks and other models that use the event delivery system, initialization also includes specifying which events are in transit to their destinations at time 0 (i.e. events generated, at least conceptually, at $t \leq 0$ for delivery at $t \geq 0$). Complex models often have complex recording and analysis methods, perhaps involving counters and vectors, and these may also need to be initialized.

State variables and STATE variables

In rough mathematical terms, if a system consists of n first order differential equations, then initialization consists in specifying the starting values of n variables. For the Hodgkin-Huxley membrane patch (only one compartment), these equations have the form

$$\frac{dv}{dt} = f_1(m, h, n, v) \qquad \text{Eq. 8.1a-d}$$

$$\frac{dm}{dt} = f_2(m, v)$$

$$\frac{dh}{dt} = f_3(h, v)$$

$$\frac{dn}{dt} = f_4(n, v)$$

so that, knowing the value of each variable at time t, we can specify the slope of each variable at time t. We have already seen (**Chapter 7**) that integration of these equations is an iterative process in which the purpose of an individual integration step (`fadvance()`) is to carry the system from time t to time $t + \Delta t$ using more or less sophisticated equations of the form

$$v(t+\Delta t) = v(t) + \Delta t \, \frac{dv(t^*)}{dt} \qquad \text{Eq. 8.2}$$

$$m(t+\Delta t) = m(t) + \Delta t \, \frac{dm(t^*)}{dt}$$

. . .

where the sophistication is in the choice of a value of t^* somewhere between t and $t + \Delta t$. However, regardless of the integration method, the iterative process cannot begin without choosing starting values for v, m, h, and n. This choice is arbitrary over the domain of the variables ($-\infty < v < \infty$, $0 \le m \le 1$, . . .), but once the initial v, m, h, and n are chosen, all auxiliary variables (e.g. conductances, currents, *d/dt* terms) at that instant of time are determined, and the equations determine the trajectories of each variable forever after. The actual evaluation of these auxiliary variables is normally done with assignment statements, such as

```
gna = gnabar*m*m*m*h
ina = gna*(v - ena)
```

This is why the model description language NMODL designates `gna` and `ina` as `ASSIGNED` variables, as opposed to the gating variables `m`, `h`, and `n`, which are the dependent variables in differential equations and are therefore termed `STATE` variables.

Unfortunately, over time an abuse of notation has evolved so that `STATE` refers to any variable that is an unknown quantity in a set of equations, and `ASSIGNED` refers to any variable that is not a `STATE` or a `PARAMETER` (`PARAMETER`s can be meaningfully set by the user as constants throughout the simulation, e.g. `gnabar`). Currently, within a single model description, `STATE` just specifies which variables are the dependent variables of `KINETIC` schemes, algebraic equations in `LINEAR` and `NONLINEAR` blocks, or differential equations in `DERIVATIVE` blocks. Generally the number of `STATE`s in a model description is equal to the number of equations. Thus, locally in a model description, the membrane potential `v` is never a dependent variable (the model description contains no equation that solves for its value) and it cannot be regarded as a user-specified value. Instead, it is declared in model descriptions as an `ASSIGNED` variable, even though it is obviously a *state variable* at the level of the entire simulation. This abuse of terminology also occurs in linear circuits, where the potential at every node is an unknown to be solved and therefore a `STATE`. However, a resistive network does not add any differential equation to the system (although it adds algebraic equations), so those additional dependent variables do not strictly need to be initialized.

While `STATE` variables may be assigned any values whatever during initialization, in practice only a few general categories of custom initialization are used. Some of these are analogous to experimental methods for preparing a system for stimulation, e.g. letting the system rest without experimental perturbation, or using a voltage clamp or constant injected current to hold the system at a defined membrane potential--the idea is that the system should reach an unchanging steady state independent of previous history. It is from this steady state that the simulation begins at time `t` = 0. When there is no steady state, as for oscillating or chaotic systems, whatever initialization is ultimately chosen

will need to be saved in order to be able to reproduce the simulation. More complicated initializations involve finding parameters that meet certain conditions, such as what value of some parameter or set of parameters yields a steady state with a desired potential. Some initial conditions may not be physically realizable by any possible manipulations of membrane potential. For example, with the `hh` model the `h` gating state has a steady state of 1 at large hyperpolarized potentials and the `n` gating state has a steady state of 1 at large depolarized potentials. It would therefore be impossible to reach a condition of `h` = 1 and `n` = 1 by controlling only voltage.

Basic initialization in NEURON: `finitialize()`

Basic initialization in NEURON is accomplished with the `finitialize()` function, which is defined in `nrn-x.x/src/nrnoc/fadvance.c` (UNIX/Linux). This carries out several actions.

1. `t` is set to 0 and the event queue is cleared (undelivered events from the previous run are thrown away).
2. Variables that receive a random stream (the list defined by `Random.play()` statements) are set to values picked from the appropriate random distributions.
3. All internal structures that depend on topology and geometry are updated, and chosen solvers are made ready.
4. The controller for `Vector.play()` variables is initialized. The controller makes use of the event delivery system for `Vector.play()` specifications that define transfer times for a step function in terms of `dt` or a time `Vector`.

   Events at time `t` = 0 (e.g. appropriate `Vector.play()` events) are delivered.
5. If `finitialize()` was called with an argument `v_init`, the membrane potential `v` in every compartment is set to the value `v_init` with a statement equivalent to

   ```
   forall for (x) v(x) = v_init
   ```
6. The `INITIAL` block of every inserted mechanism in every segment of every section is called. This includes point processes as well as distributed mechanisms (see **`INITIAL` blocks in NMODL** later in this chapter). The order in which mechanisms are initialized depends on whether any mechanism has a `USEION` statement or `WRITE`s an ion concentration.

   Ion initialization is performed first, including calculation of equilibrium potentials. Then mechanisms that `WRITE` an ion concentration are initialized; this necessitates recalculation of the equilibrium potentials for any affected ions. Finally, all other mechanism `INITIAL` blocks are called.

   Apart from these constraints, the call order of user-defined mechanisms is currently defined by the alphabetic list of `mod` file names or the order of the `mod` file arguments to `nrnivmodl` (or `mknrndll`). However one should avoid sequence-dependent `INITIAL` blocks. Thus if the `INITIAL` block of one mechanism needs the values of

variables another mechanism, the latter should be assigned before `finitialize()` is executed.

If extracellular mechanisms exist, their `vext` states are initialized to 0 before any other mechanism is initialized. Therefore, for every mechanism that computes an `ELECTRODE_CURRENT`, `v_init` refers to both the internal potential and the membrane potential.

`INITIAL` blocks are discussed in further detail below.

7. `LinearMechanism` states, if any, are initialized.
8. Network connections are initialized. This means that the `INITIAL` block inside any `NET_RECEIVE` block that is a target of a `NetCon` object is called to initialize the states of the `NetCon` object.
9. The `INITIAL` blocks may have initiated `net_send` events whose delay is 0. These events are delivered to the corresponding `NET_RECEIVE` blocks.
10. If fixed step integration is being used, all mechanism BREAKPOINT blocks are called (essentially equivalent to a call to `fcurrent()`) in order to initialize all assigned variables (conductances and currents) based on the initial STATE and membrane voltage.

    If any variable time step method is active, then those integrators are initialized. In this case, if you desire to change any state variable (here "state variable" means variables associated with differential equations, such as gating states, membrane potential, chemical kinetic states, or ion concentrations in accumulation models) after `finitialize()` is called, you must then call `cvode.re_init()` to notify the variable step methods that their copy of the initial states needs to be updated. Note that initialization of the differential algebraic solver IDA consists of two very short (`dt` = 10^{-6} ms) backward Euler time steps in order to ensure the validity of $\boldsymbol{f}(\mathbf{y}', \mathbf{y}) = \mathbf{0}$.
11. `Vector` recording of variables using the list defined by `cvode.record(&state, vector)` statements is initialized. As discussed in **Chapter 7** under **The fixed step methods: backward Euler and Crank-Nicholson**, `cvode.record()` is the only good way of keeping the proper association between local step state value and local `t`.
12. `Vectors` that record a variable, and are in the list defined by `Vector.record()` statements, record the value in `Vector.x[0]`, if `t` = 0 is a requested time for recording.

Default initialization in the standard run system: `stdinit()` and `init()`

The standard run system's default initialization takes effect when you enter a new value for `v_init` into the field editor next to the RunControl panel's Init button, or when you press either RunControl panel's Init or Init & Run button. These buttons do not call the

`init()` procedure directly but instead execute a procedure called `stdinit()` which has the implementation

```
proc stdinit() {
  realtime=0  // "run time" in seconds
  startsw()   // initialize run time stopwatch
  setdt()
  init()
  initPlot()
}
```

`setdt()` ensures (by reducing `dt`, if necessary) that the points plotted fall on time step boundaries, i.e. that `1/(steps_per_ms*dt)` is an integer. The `initPlot()` procedure begins each plotted line at `t` = 0 with the proper *y* value.

The default `init()` procedure itself is

```
proc init() {
  finitialize(v_init)
  // User-specified customizations go here.
  // If this invalidates the initialization of
  // variable time step integration and vector recording,
  // uncomment the following code.
  /*
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
  */
}
```

Custom initialization is generally accomplished by inserting additional statements after the call to `finitialize()`. These statements often have the effect of changing one or more state variables, i.e. variables associated with differential equations, such as gating states, membrane potential, chemical kinetic states, or ion concentrations in accumulation models. This invalidates the initialization of the variable time step integrator, making it necessary to call `cvode.re_init()` to notify the variable step integrator that its copy of the initial states needs to be updated. If instead fixed step integration is being used, `fcurrent()` should be called to make the values of conductances and currents consistent with the new states. Changing state variables after calling `finitialize()` can also cause incorrect values to be stored as the first element of recorded vectors. Adding `frecord_init()` to the end of `init()` prevents this.

`INITIAL` blocks in NMODL

`INITIAL` blocks for channel models generally set the gating states to their steady state values with respect to the present value of `v`. Hodgkin-Huxley style models do this easily and explicitly by calculating the voltage sensitive alpha and beta rates for each gating state and using the two state formula for the steady state, e.g.


```
PROCEDURE rates(v(mv)) {
  minf = alpha(v)/(alpha(v) + beta(v))
  . . .
}
```

and then

```
INITIAL {
  rates(v)
  m = minf
  . . .
}
```

When channel models are described by kinetic schemes, it is common to calculate the steady states with the idiom

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

where `scheme` is the name of a `KINETIC` block. To place this in an almost complete setting, consider this implementation of a three state potassium channel with two closed states and an open state:

```
NEURON {
  USEION k READ ek WRITE ik
}

STATE { c1 c2 o }

INITIAL {
  SOLVE scheme STEADYSTATE sparse
}

BREAKPOINT {
  SOLVE scheme METHOD sparse
  ik = gbar*o*(v - ek)
}

KINETIC scheme {
  rates(v) : calculate the 4 k rates.
  ~ c1 <-> c2 (k12, k21)
  ~ c2 <-> o ( k2o, ko2)
}
```

(the `rates()` procedure and some minor variable declarations are omitted for clarity). As mentioned earlier in **Default initialization in the standard run system: `stdinit()` and `init()`**, when initialization has been customized so that states are changed after `finitialize()` has been called, it is generally useful to call the `fcurrent()` function to make the values of all conductances and currents consistent with the newly initialized states. In particular this will call the `BREAKPOINT` block (twice, in order to compute the Jacobian (*di/dv*) elements for the voltage matrix equation) for all mechanisms in all segments, and on return the ionic currents such as `ina`, `ik`, and `ica` will equal the corresponding net ionic currents through each segment.

Default vs. explicit initialization of STATEs

In model descriptions, a default initialization of the STATEs of the model occurs just prior to the execution of the INITIAL block. However, this default initialization is rarely useful, and one should always explicitly implement an INITIAL block. If the name of a STATE variable is state, then there is also an implicitly declared parameter called state0. The default value of state0 is specified either in the PARAMETER block

```
PARAMETER {
  state0 = 1
}
```

or implicitly in the STATE declaration with the syntax

```
STATE {
  state START 1
}
```

If a specific value for state0 is not declared by the user, state0 will be assigned a default value of 0. state0 is not accessible from the interpreter unless it is explicitly mentioned in the GLOBAL or RANGE list of the NEURON block. For example,

```
NEURON {
  GLOBAL m0
  RANGE h0
}
```

specifies that every m will be set to the single global m0 value during initialization, while h will be set to the possibly spatially-varying h0 values. Clarity will be served if, in using the state0 idiom, you explicitly use an INITIAL block of the form

```
INITIAL {
  m = m0
  h = h0
  n = n0
}
```

Ion concentrations and equilibrium potentials

Each ion type is managed by its own separate ion mechanism, which keeps track of the total membrane current carried by the ion, its internal and external concentrations, and its equilibrium potential. The name of this mechanism is formed by appending the suffix _ion to the name of the ion specified in the USEION statement. Thus if cai and cao are integrated by a model that declares

```
USEION ca READ ica WRITE cai, cao
```

there would also be an automatically created mechanism called ca_ion, with associated variables ica, cai, cao, and eca. The initial values of cai and cao are set globally to the values of cai0_ca_ion and cao0_ca_ion, respectively (see ***Initializing concentrations in* hoc** below).

Since calcium currents, concentrations, and equilibrium potentials are managed by the `ca_ion` mechanism, one might reasonably ask why we can refer to the short names `ica`, `cai`, `cao` and `eca`, rather than the longer forms that include the suffix `_ion`, i.e. `ica_ca_ion` etc.. The answer is that there is unlikely to be any mistake about the meaning of `ica`, `cai`, . . . so we might as well take advantage of the convenience of using these short names.

Prior to version 4.1, model descriptions could not initialize concentrations, or at least it was very cumbersome to do so. Instead, the automatically created ion mechanism would initialize the ionic concentration adjacent to the membrane according to global variables. The reason that model mechanisms were not allowed to specify ion variables (or other potentially shared variables such as `celsius`) was that confusion could result if more that one mechanism at the same location tried to assign different values to the same variable. The unintended consequence of this policy is confusion of a different kind, which happens when a model declares an ion variable, such as `ena`, to be a `PARAMETER` and attempts to assign a value to it. The attempted assignment has no effect, other than to generate a warning message. Consider the mechanism

```
NEURON {
  SUFFIX test
  USEION na READ ena
}

PARAMETER {
  ena = 25 (mV)
}
```

When this model is translated by `nrnivmodl` (or `mknrndll`) we see

```
$ nrnivmodl test.mod
Translating test.mod into test.c
Warning: Default 25 of PARAMETER ena will be ignored and set by NEURON.
```

and use of the model in NEURON shows that the value of `ena` is that defined by the `na_ion` mechanism itself, instead of what was asserted in the `test` model.

```
$ nrngui
 . . .
Additional mechanisms from files
 test.mod
 . . .
oc>create soma
oc>access soma
oc>insert test
oc>ena
        50
```

If we add the initialization

```
INITIAL {
  printf("ena was %g\n", ena)
  ena = 30
  printf("we think we changed it to %g\n", ena)
}
```

to `test.mod`, we quickly discover that `ena` remains unchanged.


```
oc>finitialize(-65)
ena was 50
we think we changed it to 30
         1
oc>ena
         50
```

It is perhaps not a good idea to invite diners into the kitchen, but the reason for this can be seen from the careful hiding of the ion variables by making local copies of them in the C code generated by the `nocmodl` translator. Translation of the `INITIAL` block into a model-specific `initmodel` function is an almost verbatim copy, except for some trivial boiler plate. However, `finitialize()` calls this indirectly via the model-generic `nrn_init` function, which can be seen in all its gory detail in the C file output from `nocmodl test.mod`:

```
/**************************/
static nrn_init(_count, _nodes, _data, _pdata, _type_ignore)
        int _count, _type_ignore; Node** _nodes; double** _data; Datum** _pdata;
{ int _ix; double _v;
 _p = _data; _ppvar = _pdata;

#if _CRAY
#pragma _CRI ivdep
#endif
        for (_ix = 0; _ix < _count; ++_ix) {
 _v = _nodes[_ix]->_v;
                v = _v;
  ena = _ion_ena;
                initmodel(_ix);
        }
}
/**************************/
```

It suffices merely to call attention to the statement

```
ena = _ion_ena;
```

which shows the difference between the local copy of `ena` and the pointer to the ion variable itself. The model description can touch only the local copy and is unable to change the value referenced by `_ion_ena`. Some old model descriptions circumvented this hiding by using the actual reference to the ion mechanism variables in the `INITIAL` block (from a knowledge of the translation implementation), but that was always considered an absolutely last resort.

This hands-off policy for ion variables has recently been relaxed for the case of models that `WRITE` ion concentrations, but only if the concentration is declared to be a `STATE` *and* the concentration is initialized explicitly in an `INITIAL` block. It is meaningless for more than one model at the same location to specify the same concentrations, and an error is generated if multiple models `WRITE` the same concentration variable at the same location.

If we try this mechanism

```
NEURON {
  SUFFIX test2
  USEION na WRITE nai
  RANGE nai0
}
```



```
PARAMETER {
  nai0 = 20 (milli/liter)
}

STATE {
  nai (milli/liter)
}

INITIAL {
  nai = nai0
}
```

we get this result

```
oc>create soma
oc>access soma
oc>insert test2
oc>nai
        10
oc>finitialize(-65)
        1
oc>nai
        20
oc>nai0_test2 = 30
oc>finitialize(-65)
        1
oc>nai
        30
```

If the `INITIAL` block is not present, the `nai0_test2` starting value will have no effect.

Initializing concentrations in hoc

The best way to initialize concentrations depends on the design and intended use of the model. One must ask whether the concentration is supposed to start at the same value in all sections where the mechanism has been inserted, or should it be nonuniform from the outset?

Take the case of a mechanism that `WRITE`s an ion concentration. Such a mechanism has an associated global variable that can be used to initialize the concentration to the same value in each section where the mechanism exists. These global variables have default values for [Na], [K] and [Ca] that are broadly "reasonable" but probably incorrect for any particular case. The default concentrations for ion names created by the user are 1 mM; these should be assigned correct values in `hoc`. A subsequent call to `finitialize()` will use this to initialize ionic concentrations.

The name of the global variable is formed from the name of the ion that the mechanism uses and the concentration that it `WRITE`s. For example, suppose we have a mechanism `kext` that implements extracellular potassium accumulation as described by Frankenhaeuser and Hodgkin (Frankenhaeuser and Hodgkin 1956). The `kext` mechanism `WRITE`s `ko`, so the corresponding global variable is `ko0_k_ion`. The sequence of instructions

```
ko0_k_ion = 10      // seawater, 4 x default value (2.5)
ki0_k_ion = 4*54.4  // 4 x default value, preserves ek
finitialize(v_init) // v_init is the starting Vm
```


will set ko to 10 mM and ki to 217.6 mM in every segment that has the kext mechanism.

What if one or more sections of the model are supposed to have different initial concentrations? For these particular sections we can use the ion_style() function to assert that the global variable is not to be used to initialize the concentration for this particular ion. A complete discussion of ion_style(), its arguments, and its actions is contained in NEURON's help system, but we will consider one specific example here. Let's say we have inserted kext into section dend. Then the numeric arguments in the statement

```
dend ion_style("k_ion",3,2,1,1,0)
```

would have the following effects on the kext mechanism in the dend section (in sequence): treat ko as a STATE variable; treat ek as an ASSIGNED variable; on call to finitialize() use the Nernst equation to compute ek from the concentrations; compute ek from the concentrations on every call to fadvance(); do *not* use ko0_k_ion or ki0_k_ion to set the initial values of ko and ki. The proper initialization is to set ko and ki explicitly for this section, e.g.

```
ko0_k_ion = 10  // all sections start with ko = 10 mM
dend {ko = 5  ki = 2*54.4}  //  . . . except dend
finitialize(v_init)
```

Examples of custom initializations

Initializing to a particular resting potential

Perhaps the most trivial custom initialization is to force the initialized voltage to be the resting potential. Returning our consideration to initialization of the HH membrane compartment,

```
finitialize(-65)
```

will indeed set the voltage to -65 mV, and m, h, and n will be in steady state relative to that voltage. However, this must be considered analogous to a voltage clamp initialization since the sum of all the currents may not be 0 at this potential, i.e. -65 mV may not be the resting potential. For this reason it is common to adjust the equilibrium potential of the leak current so that the resting potential is precisely -65 mV.

This can be done with a user-defined init() procedure based on the idea that total membrane current at steady state must be 0. For our single compartment HH model, this means that

```
0 = ina + ik + gl_hh*(v - el_hh)
```

so our custom init() is

Remember to load user-defined versions of functions or procedures that are part of the standard system, such as init(), *after* loading stdrun.hoc. Otherwise, the user-defined version will be overwritten.


```
proc init() {
  finitialize(-65)
  el_hh = (ina + ik + gl_hh*v)/gl_hh
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

The `cvode.re_init()` call is not essential here since states have not been changed, but it is still good practice since it will update the calculation of all the *dstate/dt* (note that now *dv/dt* should be 0 as a consequence of the change in `el_hh`) as well as internally make a call to `fcurrent()` (necessary because changing `el_hh` requires recalculation of `il_hh`).

Calculating the value of leak equilibrium potential in order to realize a specific resting potential is not fail-safe in the sense that the resultant value of `el_hh` may be very large and out of its physiological range--after all, `gl_hh` may be a very small quantity. It may sometimes be better to introduce a constant current mechanism and set its value so that

```
0 = ina + ik + ica + i_constant
```

holds at the desired resting potential. An example of such a mechanism is

```
: constant current for custom initialization

NEURON {
  SUFFIX constant
  NONSPECIFIC_CURRENT i
  RANGE i, ic
}

UNITS {
  (mA) = (milliamp)
}

PARAMETER {
  ic = 0 (mA/cm2)
}

ASSIGNED {
  i (mA/cm2)
}

BREAKPOINT {
  i = ic
}
```

and the corresponding custom `init()` would be


```
proc init() {
  finitialize(-65)
  ic_constant = -(ina + ik + il_hh)
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

Before moving on to the next example, we should mention that testing is required to verify that the system is stable at the desired `v_init`, i.e. that the system returns to `v_init` after small perturbations.

Initializing to steady state

In **Chapter 4** we mentioned that NEURON's default integrator uses the backward Euler method, which can find the steady state of a linear system in a single step if the integration step size is large compared to the longest system time constant. Backward Euler can also find the steady state of many nonlinear systems, but it may be necessary to perform several iterations with large `dt`. An `init()` that takes advantage of this fact is

```
proc init() { local dtsav, temp
  finitialize(v_init)
  t = -1e10
  dtsav = dt
  dt = 1e9
  // if cvode is on, turn it off to do large fixed step
  temp = cvode.active()
  if (temp!=0) { cvode.active(0) }
  while (t<-1e9) {
    fadvance()
  }
  // restore cvode if necessary
  if (temp!=0) { cvode.active(1) }
  dt = dtsav
  t = 0
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

This first performs a preliminary "voltage clamp" initialization to `v_init`. Then it sets time to a very large negative value (to prevent triggering point processes and other events) and integrates over several steps with a large fixed `dt` so that the system can reach steady state. The procedure wraps up by returning `dt` to its original value, setting `t` back to 0, and, if necessary, reactivating the variable step integrator. The last few statements are the familiar re-initialization of cvode or invocation of `fcurrent()`, followed by initialization of vector recording.

This initialization strategy generally works well, but there are circumstances in which it may fail. Active transport mechanisms can be troublesome with fixed time step integration if `dt` is large, because even a small pump rate may produce a negative concentration. To see a more mundane example of instability with large `dt`, construct a single compartment model that has the `hh` mechanism. With the default `hh` parameters, and in the absence of any injected current, this is quite stable even for huge values of `dt` (e.g. 10^5 ms). Now reduce `gnabar_hh` to 0, increase `dt` to 100 ms, and watch what happens over the course of 5000 ms. The result is an oscillation whose peak-to-peak amplitude gradually increases to ~ 10 mV. It would be all to easy to miss such oscillations when using steady state initialization with large steps. This underscores the need for careful testing of any initialization strategy, since in a sense all of them work "behind the scenes."

Initializing to a desired state

Suppose the end of some run is to serve as the initial condition for subsequent runs; this is a particularly useful strategy for dealing with models that oscillate or otherwise lack a "resting" state. We can save all the states with

```
objref svstate, f
svstate = new SaveState()
svstate.save()
```

The binary state information can be saved for use in later neuron sessions with

```
f = new File("states.dat")
svstate.fwrite(f)
```

and future sessions can read the file into the `SaveState` object with

```
objref svstate, f
svstate = new SaveState()
f = new File("states.dat")
svstate.fread(f)
```

Whether or not the state information comes from a `svstate.save()` in this session or was read from a file, we only have to make a minor change to `init()` in order to use that information to initialize the system.

```
proc init() {
  finitialize(v_init)
  svstate.restore()
  t = 0 // t is one of the "states"
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

This might be called a "groundhog day initialization," after the movie in which the protagonist awakened to the same day over and over again.

Now every simulation will start from the state that we saved earlier.

Initializing by changing model parameters

Occasionally the aim is to bring a model to an initial condition that it would never reach on its own. This can be a particular challenge if the model involves several interacting nonlinear processes, making it difficult or impossible to know in advance what values the states should have. Such problems can sometimes be circumvented by changing the parameters of the model so that initialization reaches the desired state, and then restoring the original parameters of the model.

As a specific example, consider a conceptual model of the regulation of the calcium concentration in a thin intracellular compartment ("shell") adjacent to the cell membrane (Fig. 8.1). Calcium (Ca^{+2}) can enter or leave the shell in one of three ways: by diffusion between the shell and the core of the cell, by active transport via a membrane-bound pump, or as a result of non-pump calcium current I_{Ca} (i.e. transmembrane calcium flux not produced by the pump). For the sake of simplicity, we will assume that Ca_{core} and Ca_o ($[Ca^{+2}]$ in the core and extracellular solution) are constant. However, the problems that we encounter, and the manner in which we solve them, would be the same even if Ca_{core} and Ca_o were allowed to vary.

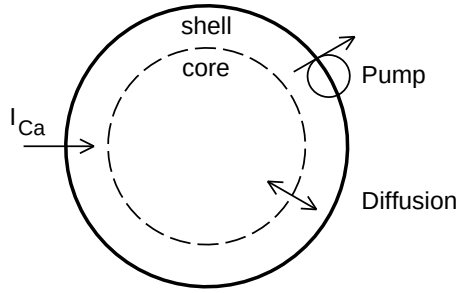


Fig. 8.1. Schematic diagram of a model of regulation of $[Ca^{+2}]$ in a thin shell just inside the cell membrane.

Our goals are to:

1. initialize the internal calcium concentration next to the membrane $[Ca^{+2}]_{shell}$ (hereafter called Ca_{shell}) to a desired value and then plot Ca_{shell} and the pump current $I_{Ca_{pump}}$ as functions of time
2. plot the starting value of $I_{Ca_{pump}}$ as a function of the initial Ca_{shell}

To achieve these goals, we must be able to set the initial value of Ca_{shell} to whatever level we want and ensure that the pump reaches its corresponding steady state.

Details of the mechanism

The kinetic scheme that describes this mechanism of calcium regulation is

diffusion $$\mathrm{Ca}_{\mathrm{core}} \underset{1/\tau}{\overset{1/\tau}{\rightleftarrows}} \mathrm{Ca}_{\mathrm{shell}}$$ Eq. 8.3a

active transport $$\mathrm{Ca}_{\mathrm{shell}} + \mathrm{Pump} \underset{k_2}{\overset{k_1}{\rightleftarrows}} \mathrm{CaPump}$$ Eq. 8.3b and c

$$\mathrm{CaPump} \underset{k_4}{\overset{k_3}{\rightleftarrows}} \mathrm{Ca}_o + \mathrm{Pump}$$

calcium current $$\mathrm{Ca}_{\mathrm{shell}} \overset{1/2\,F\,\mathrm{vol}}{\leftarrow} -I_{\mathrm{Ca}}$$ Eq. 8.3d

where τ is the time constant for equilibration of Ca^{+2} between the shell and the core, F is Faraday's constant, and vol is the volume of the shell.

The NMODL code that implements this mechanism is

```
NEURON {
  SUFFIX capmp
  USEION ca READ cao, ica, cai WRITE cai, ica
  RANGE tau, width, cacore, ica, pump0
}

UNITS {
  (um)     =  (micron)
  (molar)  =  (1/liter)
  (mM)     =  (millimolar)
  (uM)     =  (micromolar)
  (mA)     =  (milliamp)
  (mol)    =  (1)
  FARADAY  =  (faraday)  (coulomb)
}

PARAMETER {
  width = 0.1    (um)
  tau = 1        (ms) : corresponds to D = 2e-7 cm2/s
  : D for Ca in water is 6e-6 cm2/s, i.e. 30x faster
  k1 = 5e8       (/mM-s)
  k2 = 0.25e6    (/s)
  k3 = 0.5e3     (/s)
  k4 = 5e0       (/mM-s)
  cacore = 0.1   (uM)
  pump0 = 3e-14  (mol/cm2)
}
```



```
ASSIGNED {
  cao     (mM) : on the order of 10 mM
  cai     (mM) : on the order of 0.001 mM
  ica     (mA/cm2)
  ica_pmp (mA/cm2)
  ica_pmp_last (mA/cm2)
}

STATE {
  cashell  (uM)       <1e-6>
  pump     (mol/cm2)  <1e-16>
  capump   (mol/cm2)  <1e-16>
}

INITIAL {
  ica = 0
  ica_pmp = 0
  ica_pmp_last = 0
  SOLVE pmp STEADYSTATE sparse
}

BREAKPOINT {
  SOLVE pmp METHOD sparse
  ica_pmp_last = ica_pmp
  ica = ica_pmp
}

KINETIC pmp {
  : volume/unit surface area has dimensions of um
  : area/unit surface area is dimensionless
  COMPARTMENT width {cashell}
  COMPARTMENT (1e13) {pump capump}
  COMPARTMENT 1(um) {cacore}
  COMPARTMENT (1e3)*1(um) {cao}
  CONSERVE pump + capump = (1e13)*pump0
  ~ cacore <-> cashell (width/tau, width/tau)
  ~ cashell + pump <-> capump ((1e7)*k1, (1e10)*k2)
  ~ capump <-> cao + pump ((1e10)*k3, (1e10)*k4)
  ica_pmp = (1e-7)*2*FARADAY*(f_flux - b_flux)

  : ica_pmp is the "new" value, but cashell must be
  : computed using the "old" value, i.e. ica_pmp_last
  ~ cashell << (-(ica - ica_pmp_last)/(2*FARADAY)*(1e7))

  cai = (0.001)*cashell
}
```

Initializing the mechanism

For the sake of convenience we will assume that our model cell has only one section called `soma`, and that `soma` is the default section. Also suppose that we have already assigned the desired value of Ca_{shell} to a parameter we will call `ca_init`, e.g. with a statement of the form `ca_init = `*`somevalue`*. Our problem is how to ensure that initialization makes `cashell_capmp` take on the value of `ca_init`.

As a naive first stab at this problem, we might try changing the `init()` procedure like this

```
proc init() {
  cashell_capmp = ca_init
  finitialize(v_init)
}
```

i.e. inserting a line that sets the desired value of Ca_{shell} before calling `finitialize()`. To see whether this has the desired effect, we need only to run a simulation and examine the time course of Ca_{shell} and the pump current $I_{Ca_{pump}}$. This quickly shows that, no matter what value we first assign to `cashell_capmp`, `finitialize()` drives Ca_{shell} and $I_{Ca_{pump}}$ to the same steady state levels (Fig. 8.2). We might have anticipated this result, because it is what steady state initialization is supposed to do. If Ca_{shell} is too high, the excess calcium will diffuse into the core or be pumped out of the cell until Ca_{shell} returns to the steady state value. On the other hand, if Ca_{shell} is too low, calcium will diffuse into the shell from the core, and the pump will slow or may even reverse, until Ca_{shell} comes back to the steady state value. Thus, regardless of how we perturb Ca_{shell}, steady state initialization always brings the model back to the same condition.

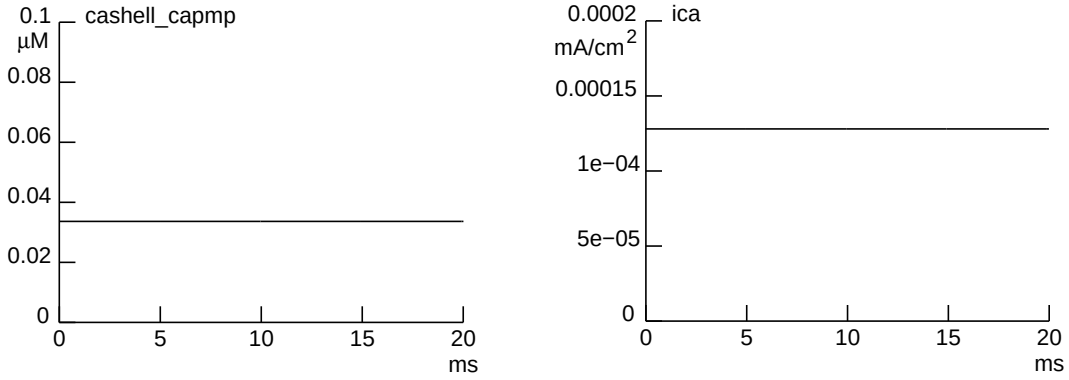


Fig. 8.2. Default initialization after setting `cashell_capmp` to 0.1 µM leaves Ca_{shell} (left) and $I_{Ca_{pump}}$ (right) at their steady state levels of ~ 0.034 µM and ~ 1.3×10^{-4} mA/cm^2, respectively.

For our second attempt we try calling `finitialize()` first, and then setting the desired value of Ca_{shell}.

```
proc init() {
  finitialize(v_init)
  cashell_capmp = ca_init
  // we've changed a state, so the following are needed
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

This is partly successful, in that it does affect Ca_{shell} and $I_{Ca_{pump}}$, but plots of these variables seem to start from the wrong initial conditions. For example, if we try `ca_init` = 0.1 μM, the plot of `cashell_capmp` appears to start with a value of ~ 0.044 μM instead. Using the Graph menu's Color/Brush to change the color and thickness of the plots of `cashell_capmp` and `ica`, we discover the presence of early, fast transients that overlie the y axis (Fig. 8.3 top). Thus `cashell_capmp` really does start at the right initial value, but in less than 5 microseconds it drops by ~ 56%. So we have solved one mystery only to uncover another: what causes these fast transients?

Some reflection brings the realization that, although we changed the concentration in the shell, we did not properly initialize the pump. Consequently, as soon as we launch a simulation, Ca^{+2} starts binding to the pump, and this is responsible for the precipitous drop of Ca_{shell}. At the same time, the rate of active transport begins to rise, which is reflected in the increase of $I_{Ca_{pump}}$. These changes produce the "pump transients" in Ca_{shell} and $I_{Ca_{pump}}$, which can be quite large as Fig. 8.3 shows.

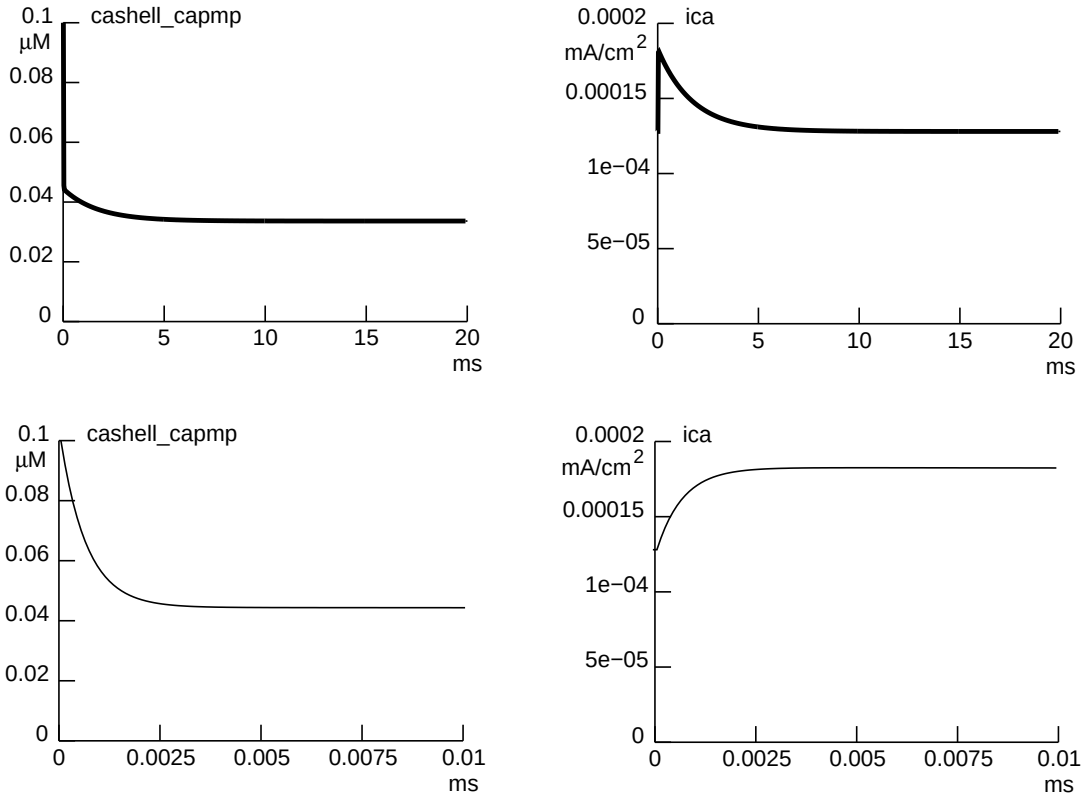


Fig. 8.3. Time course of Ca_{shell} (left) and $I_{Ca_{pump}}$ (right) following an initialization that increased Ca_{shell} abruptly after calling `init()`. The traces in the top figures were thickened to make the early fast transients easier to see. The time scale of the bottom figures has been expanded to reveal the details of these fast transients. The final steady state levels of Ca_{shell} and $I_{Ca_{pump}}$ are the same as in Fig. 8.2.

A strategy that does what we want is to change the value of `cacore_capmp` to `ca_init` and make τ very fast before calling `finitialize()`, then wrap up by restoring the values of `cacore_capmp` and τ. This amounts to changing the model in order to achieve the desired initialization. One example of such a custom `init()` is

```
proc init() { local savcore, savtau
  // make cacore equal to ca_init
  savcore = cacore_capmp
  cacore_capmp = ca_init
  // initialize cashell to cacore
  savtau = tau_capmp
  tau_capmp = 1e-6  // so cashell tracks cacore closely
  finitialize(v_init)
```



```
    // restore cacore and tau
    cacore_capmp = savcore
    tau_capmp = savtau
    if (cvode.active()) {
      cvode.re_init()
    } else {
      fcurrent()
    }
    frecord_init()
}
```

This code ensures that the difference between Ca_{shell} and Ca_{core} becomes vanishingly small, and at the same time allows the pump to initialize properly (Fig. 8.4).

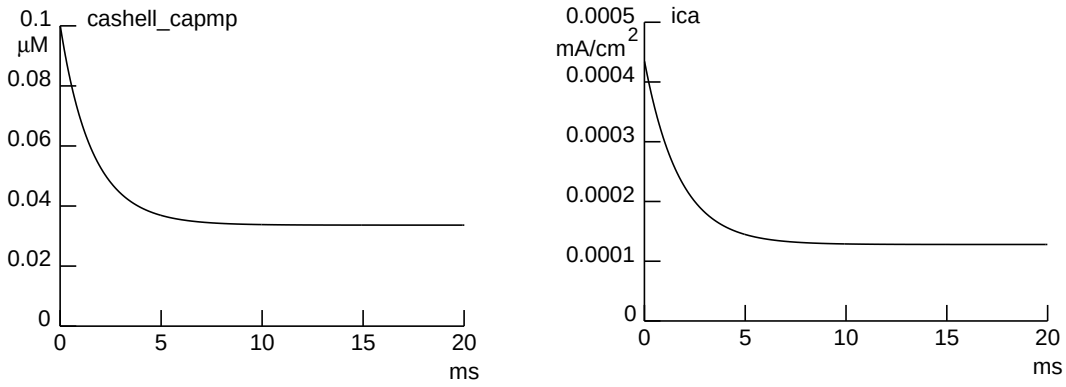


Fig. 8.4. Following proper initialization, plots of Ca_{shell} (left) and $I_{Ca_{pump}}$ (right) begin at the correct values and do not display the early fast transient that appeared in Fig. 8.3.

Now we can plot the starting value of $I_{Ca_{pump}}$ as a function of the initial Ca_{shell}. Figure 8.5 shows a Grapher configured to do this. To make this a semilog plot, we used an independent variable `x` to sweep `ca_init` from 10^{-4} to 10^{2} µM in 30 logarithmically equally spaced intervals. For each value of `x` the Grapher calculated the corresponding value of `ca_init` as 10^{x}, called our custom `init()`, and plotted the resulting `ica_capmp` vs. `log10(cashell_capmp)`, i.e. $\log_{10}(Ca_{shell})$. Note that `log10(cashell_capmp)` ranges from -4 to 2, which means that Ca_{shell} ranges from 10^{-4} to 10^{2} µM, i.e. exactly the same range of concentrations as `ca_init`. This confirms the ability of our custom `init()` to set `cashell_capmp` to the desired values.

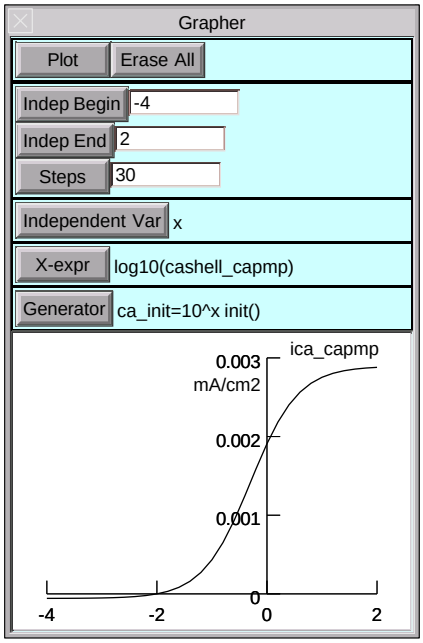


Fig. 8.5. A Grapher used to plot of $I_{Ca_{pump}}$ vs. initial Ca_{shell}. The Graph menu's Change Text was used to add the `mA/cm2` label.

References

Frankenhaeuser, B. and Hodgkin, A.L. The after-effects of impulses in the giant nerve fibers of *Loligo*. *J. Physiol.* 131:341-376, 1956.

Chapter 8 Index

Chapter 10

Synaptic transmission and artificial spiking cells

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

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

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

Modeling communication between cells

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

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

sharp contrast to the examples we discussed in **Chapter 9**, all of which are "local" in the sense that an instance of a mechanism at a particular 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

A minimal conceptual model of graded synaptic transmission is that neurotransmitter is released continuously at a rate that depends on something in the presynaptic terminal, and that this causes some change in the postsynaptic cell. For the sake of discussion, let's say this something is $[Ca^{2+}]_{pre}$, the concentration of free calcium in the presynaptic terminal. We will also assume that the transmitter changes an ionic conductance in the postsynaptic cell.

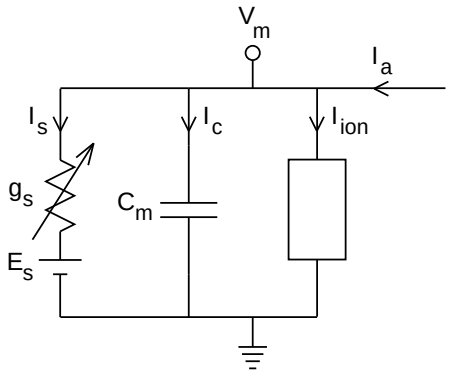


Figure 10.1. Membrane potential in the immediate neighborhood of a postsynaptic conductance depends on the synaptic current (I_s), the currents through the local membrane capacitance and ionic conductances (I_c and I_{ion}), and the axial current arriving from adjacent regions of the cell (I_a).

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

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

This is the charge balance equation for the electrical vicinity of the postsynaptic region. The terms on the left hand side are the usual local capacitive and ionic transmembrane currents. The first term on the right hand side is the current that enters the postsynaptic

region from adjacent parts of the cell, which NEURON takes care of automatically. The second term on the right hand side expresses the effect of the ligand-gated channels. The current through these channels is the product of two factors. The first factor is merely the local electrochemical gradient for ion flow. The second factor is a conductance term that depends on the calcium concentration at some other location.

We already know that a localized conductance is implemented in NEURON with a point process, and that such a mechanism is automatically able to access all the local variables that it needs (in this case, the local membrane potential and the synapse's equilibrium potential). But the calcium concentration in the presynaptic terminal is nonlocal, and that poses a problem; furthermore, it is likely to change with every `fadvance()`.

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

```
somedendrite.syn.capre = precell.bouton.cai(1)
```

At each time step, this would update the variable `capre` in the synaptic mechanism `syn` attached to the postsynaptic section `somedendrite`, making it equal to the free calcium concentration `cai` at the 1 end of the `bouton` section in the presynaptic cell `precell`. However, this statement would have to be reinterpreted at each `fadvance()`, which might slow down the simulation considerably.

If what happens to the postsynaptic cell depends on the moment-to-moment details of what is going on in the presynaptic terminal, it is far more efficient to use a `POINTER` variable (see Listing 10.1). In NMODL, a `POINTER` variable holds a reference to another variable. The specific reference is defined by a `hoc` statement, as we shall see below.

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

```
: Graded synaptic transmission

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

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

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



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

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

Listing 10.1. `gradsyn.mod`

The `NEURON` block

The `POINTER` statement in the NEURON block declares that `capre` refers to some other variable that may belong to a noncontiguous segment, possibly even in a different section; below we show how to attach this to the free calcium concentration in a presynaptic terminal. The synaptic strength is not specified by a peak conductance, but in terms of a "transfer function scale factor" `k`, which has units of ($\mu S/mM^3$).

The `BREAKPOINT` block

The synaptic conductance `g` is proportional to the cube of `capre` and does not saturate. This is similar to the calcium dependence of synaptic conductance in a model described by De Schutter et al. (1993).

Usage

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

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

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

Figure 10.2 shows simulation results from a model of graded synaptic transmission. In this model, the presynaptic terminal `precell` is a 1 μm diameter hemisphere with voltage-gated calcium current `cachan` (`cachan.mod` in `c:nrnxx\examples\nrniv\nmodl` under MSWindows or `nrn-x.x/share/examples/nrniv/nmodl` under UNIX) and a calcium accumulation mechanism that includes diffusion, buffering, and a pump (`cdp`, discussed in **Example 9.9: a calcium pump**). The postsynaptic cell is a passive single compartment with surface area 100 μm^2, C_m = 1 $\mu f/cm^2$, and τ_m = 30 ms. A `GradSyn`

synapse with transfer function scale factor k = 0.2 μS/mM^3 is attached to the postsynaptic cell, and presynaptic membrane potential is driven between -70 and -30 mV by a sinusoid with a period of 400 ms. The time course of presynaptic $[Ca]_i$ and synaptic conductance show clipping of the negative phases of the sine wave; the postsynaptic membrane potential shows less clipping because of filtering by membrane capacitance.

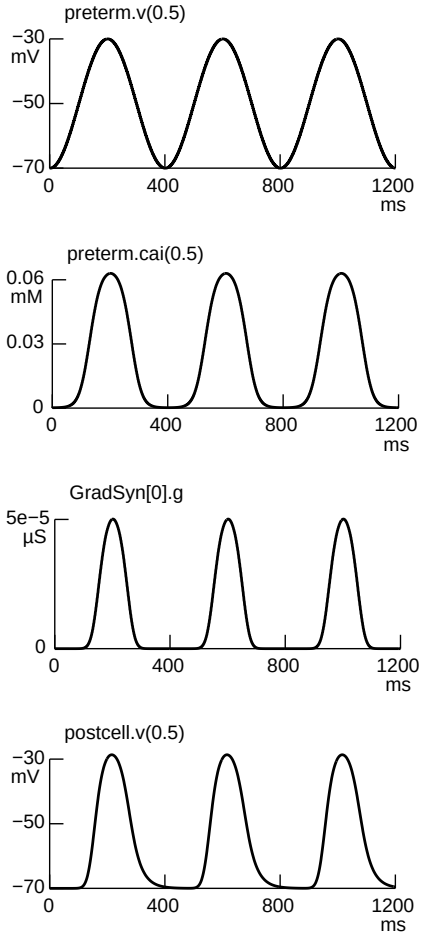


Figure 10.2. Graded synaptic transmission. Top two graphs: Presynaptic membrane potential `preterm.v` was "clamped" to -70-20cos(2πt/400) mV, producing a periodic increase of $[Ca]_i$ (`preterm.cai` is the concentration just inside the cell membrane) with clipping of the negative peaks. Bottom two graphs: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

Example 10.2: a gap junction

The current that passes through a gap junction depends on the moment-to-moment fluctuations of voltage on both sides of the junction. This can be handled by a pair of point processes on the two sides that use `POINTER`s to monitor each other's voltage, as in


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

Conservation of charge requires the use of two point processes: one drains current from one side of the gap junction, and the other delivers an equal current to the other side.

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

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

PARAMETER { r = 1e10 (megohm) }

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

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

Listing 10.2. `gap.mod`

This implementation can cause spurious oscillations if the coupling between the two voltages is too tight (i.e. if the resistance `r` is too low) because it degrades the Jacobian matrix of the system equations. While it does introduce off-diagonal terms to couple the nodes on either side of the gap junction, it fails to add the conductance of the gap junction to the terms on the main diagonal. The result is an *approximate* Jacobian, which makes numeric integration effectively a modified Euler method, instead of the fully implicit or Crank-Nicholson methods which are numerically more robust. Consequently, results are satisfactory only if coupling is loose (i.e. if `r` is large compared to the total conductance of the other ohmic paths connected to the affected nodes). If oscillations do occur, their amplitude can be reduced by decreasing `dt`, and they can be eliminated by using CVODE. In such cases, it may be preferable to implement gap junctions is with the `LinearMechanism` class (e.g. by using the LinearCircuitBuilder), which sets up the diagonal and off-diagonal terms of the Jacobian properly so that simulations are completely stable.

Usage

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

```
create a,b
access a

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



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

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

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

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

The experience of NEURON users in creating special strategies for managing network simulations (e.g. (Destexhe et al. 1994a; Lytton 1996)) stimulated the development of NEURON's network connection (`NetCon`) class and event delivery system. Instances of the `NetCon` class manage the delivery of presynaptic "spike" events to synaptic point processes via the event delivery system. This works for all of NEURON's integrators, including the local variable time step method in which each cell is integrated with a time step appropriate to its own state changes. Model descriptions of synapses never need to queue events, and there is no need for heroic efforts to make them work properly with adaptive integration. These features offer enormous convenience to users who are interested in models that involve synaptic transmission at any level of complexity from single cell to large networks.

Conceptual model

In its most basic form, the physical system that we want to represent consists of a presynaptic neuron with a spike initiation zone that gives rise to an axon, which leads to a terminal that makes a synaptic connection onto a postsynaptic cell (Fig. 10.3). Our conceptual model of spike-triggered transmission is that arrival of a spike at the presynaptic terminal has some effect (e.g. a conductance change) in the postsynaptic cell that is described by a differential equation or kinetic scheme. Details of what goes on at the spike initiation zone are assumed to be unimportant--all that matters is whether a spike has, or has not, reached the presynaptic terminal. This conceptual model lets us take advantage of special features of NEURON that allow extremely efficient computation.

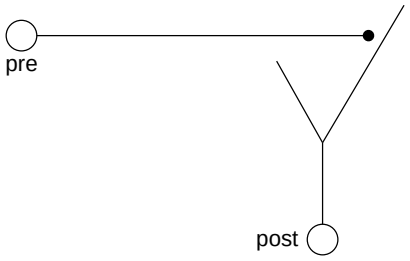


Figure 10.3. Cartoon of a synaptic connection (filled circle) between a presynaptic cell pre and a postsynaptic cell post.

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

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

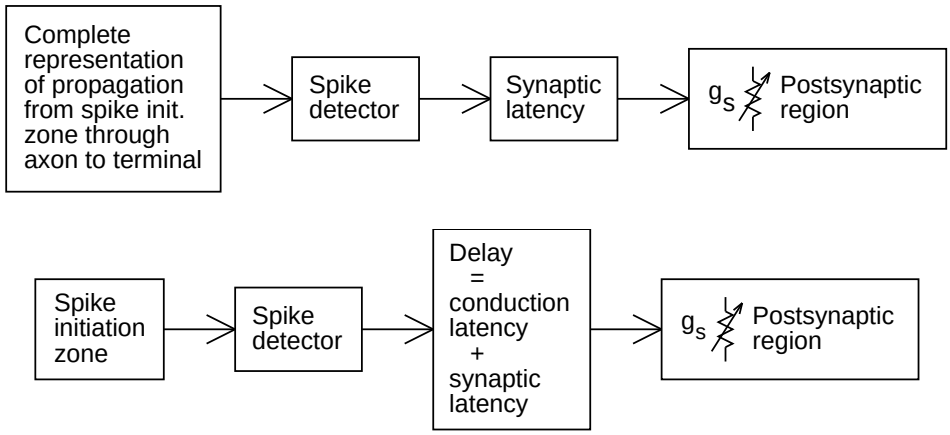


Figure 10.4. Computational implementation of a model of spike-triggered synaptic transmission. Top: The basic idea is that a presynaptic spike causes some change in the postsynaptic cell. Bottom: A more efficient version doesn't bother computing conduction in the presynaptic axon.

The NetCon class

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

A NetCon object connects a presynaptic variable, such as voltage, to a target point process (here a synapse) with arbitrary delay and weight. If the presynaptic variable crosses threshold in a positive direction at time t, then at time t+delay a special NET_RECEIVE procedure in the target point process is called and receives the weight information. Each NetCon can have its own threshold, delay, and weight, i.e. these parameters are stream-specific. The only constraint on delay is that it be nonnegative. There is no limit on the number of events that can be "in the pipeline," and there is no loss of events under any circumstances. Events always arrive at the target at the interval delay after the time they were generated.

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

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

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

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

The weight associated with a NetCon object is actually the first element of a weight vector. The number of elements in the weight vector depends on the number of arguments in the NET_RECEIVE statement of the NMODL source code that defines the point process. We will return to this in **Example 10.5: use-dependent synaptic plasticity** and **Example 10.6: saturating synapses**.

NEURON's event-based approach to implementing communication between cells reduces the computational burden of network simulations tremendously, because it supports efficient, unlimited divergence and convergence (fan-out and fan-in). To understand why, first consider divergence. What if a presynaptic cell projects to multiple postsynaptic targets (Fig. 10.5 top)? Easy enough--just add a NetCon object for each target (Fig. 10.5 bottom). This is computationally efficient because threshold detection is done on a "per source" basis, rather than a "per NetCon" basis. That is, if multiple NetCons have the same source with the same threshold, they all share a single threshold detector. The source variable is checked only once per time step and, when it crosses threshold in the positive direction, events are generated for each connecting NetCon object. Each of these NetCons can have its own weight and delay, and the target mechanisms can belong to different classes.

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

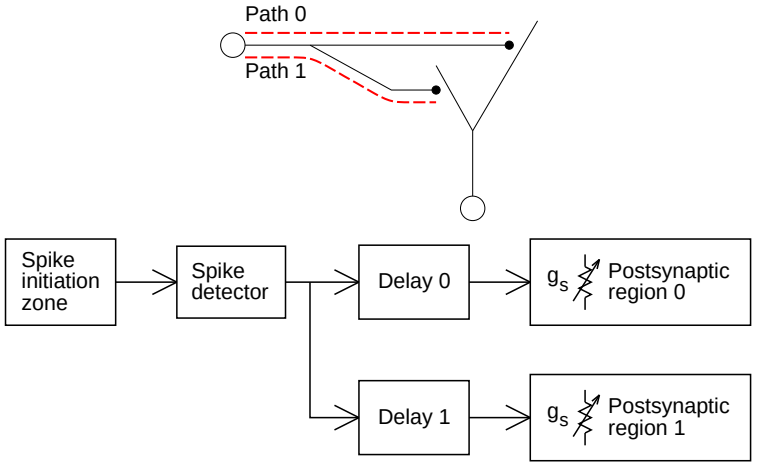


Figure 10.5. Efficient divergence. Top: A single presynaptic neuron projects to two different target synapses. Bottom: Computational model of this circuit uses multiple `NetCon`s with a single threshold detector that monitors a common source.

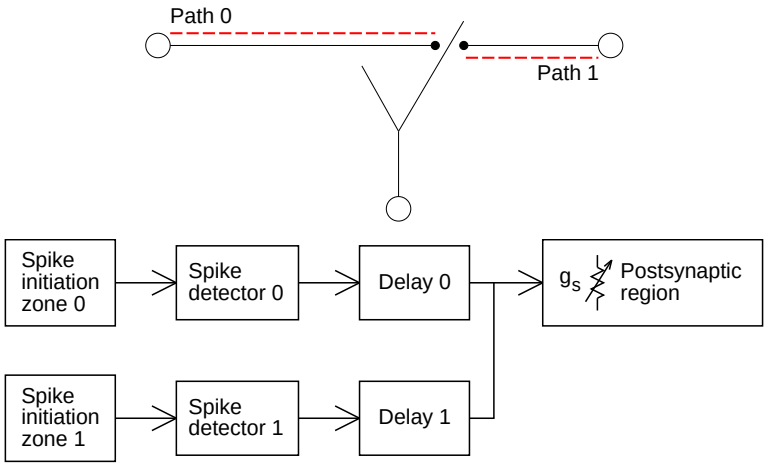


Figure 10.6. Efficient convergence. Top: Two different presynaptic cells make synaptic connections of the same class that are electrically close to each other. Bottom: Computational model of this circuit uses multiple `NetCon`s that share a single postsynaptic mechanism (single equation handles multiple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

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

Example 10.3: synapse with exponential decay

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

The NMODL code that implements such a mechanism is shown in Listing 10.3. This mechanism is similar to NEURON's built in `ExpSyn`. Calling it `ExpSyn1` allows us to test and modify it without conflicting with NEURON's built-in `ExpSyn`.

The synaptic conductance of this mechanism summates not only when events arrive from a single presynaptic source, but also when they arrive from different places (multiple input streams). This mechanism handles both situations by defining a single conductance state *g* which is governed by a differential equation whose solution is $g(t) = g(t_0)\, \mathrm{e}^{(t-t_0)/\tau}$, where $g(t_0)$ is the conductance at the time of the most recent event.

```
: expsyn1.mod

NEURON {
  POINT_PROCESS ExpSyn1
  RANGE tau, e, i
  NONSPECIFIC_CURRENT i
}

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

ASSIGNED {
  v  (millivolt)
  i  (nanoamp)
}

STATE { g (microsiemens) }

INITIAL { g = 0 }

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



```
DERIVATIVE state { g' = -g/tau }

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

Listing 10.3. `expsyn1.mod`

The `BREAKPOINT` block

The `BREAKPOINT` block of this mechanism is its main computational block. This contains the `SOLVE` statement that tells how states will be integrated. The `cnexp` method is used because the kinetics of `ExpSyn1` are described by a differential equation of the form $y' = f(y)$, where $f(y)$ is linear in y (see also **The `DERIVATIVE` block** in **Example 9.4: a voltage-gated current** in **Chapter 9**). The `BREAKPOINT` block ends with an assignment statement that sets the value of the synaptic current.

The `DERIVATIVE` block

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

The `NET_RECEIVE` block

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

So suppose we have a model with an `ExpSyn1` point process that is the target of a `NetCon`. Imagine that the `NetCon` detects a presynaptic spike at time `t`. What happens next?

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

`ExpSyn1`'s conductance `g` continues to follow a smooth exponential decay with time constant `tau` until time `t+delay`, where `delay` is the delay associated with the `NetCon` object. At this point, an event is delivered to the `ExpSyn1`. Just before entry to the `NET_RECEIVE` block, NEURON makes all `STATE`s, `v`, and values assigned in the `BREAKPOINT` block consistent at `t+delay`. Then the code in the `NET_RECEIVE` block is executed, making the synaptic conductance suddenly jump up by the amount specified by the `NetCon`'s weight.

Usage

Suppose we wanted to set up a synaptic connection between two cells using an `ExpSyn1` mechanism, as in Fig. 10.7.

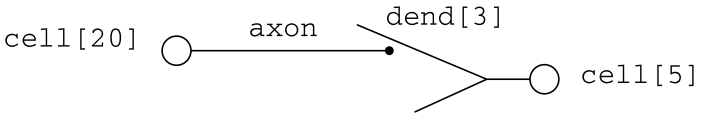


Figure 10.7. Schematic of a synaptic connection between two cells.

This could be done with the following hoc code, which also illustrates the use of a List of NetCon objects as a means for keeping track of the synaptic connections in a network.

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

// attach an ExpSyn1 point process called syn
//   to the 0.3 location on dend[3] of cell[5]
objref syn
cell[5].dend[3] syn = new ExpSyn1(0.3)

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

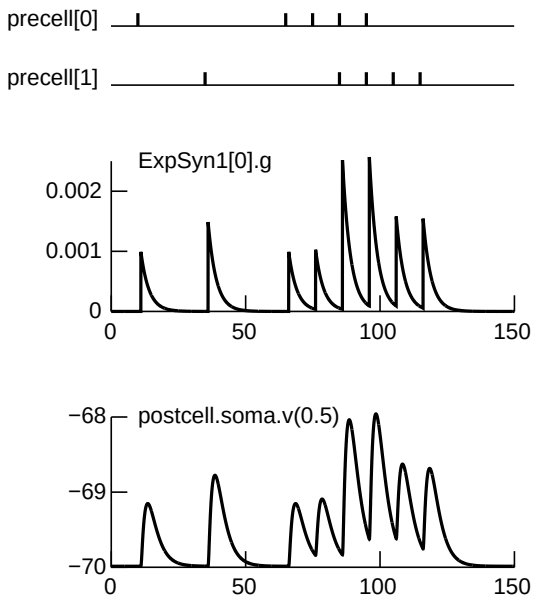


Figure 10.8. Simulation results from the model shown in Fig. 10.6. Note stream-specific synaptic weights and temporal summation of synaptic conductance and membrane potential.

Figure 10.8 shows results of a simulation of two input streams that converge onto a single ExpSyn1 attached to a postsynaptic cell, as in the diagram at the top of Fig. 10.6. The presynaptic firing times are indicated by the rasters labeled precell[0] and precell[1]. The synaptic conductance and postsynaptic membrane potential (middle and bottom graphs) display stream-specific synaptic weights, and also show temporal summation of inputs within an individual stream and between inputs on multiple streams.

Example 10.4: alpha function synapse

With a few small changes, we can extend `ExpSyn1` to implement an alpha function synapse. We only need to replace the differential equation with the two state kinetic scheme

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

and change the `NET_RECEIVE` block to

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

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

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

```
DERIVATIVE state {
..a' = -a/tau1
..b' = -b/tau
..g = b - a
}
```

which are solved by the `cnexp` method (this is what NEURON's built in `Exp2Syn` mechanism does). As `tau1` approaches `tau`, `g` approaches an alpha function (although the factor by which `weight` must be multiplied approaches infinity; see `factor` in the next example). Also, there are now two state discontinuities in the `NET_RECEIVE` block

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

Example 10.5: use-dependent synaptic plasticity

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

that it is quite different from models described by others (Destexhe and Sejnowski 1995) in which the G protein itself gates the ionic channels.

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

```
: gsyn.mod

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

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

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

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

STATE {
  A  (umho)
  B  (umho)
}

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



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

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

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

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

Listing 10.4. `gsyn.mod`

The `NET_RECEIVE` block

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

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

Note that `G1` and `G2` are not `STATE`s in this mechanism. They are not even variables in this mechanism, but instead are "owned" by the particular `NetCon` instance that delivered the event. Each `NetCon` object instance keeps an array (the weight vector) whose size equals the number of arguments to `NET_RECEIVE`, and the arguments to `NET_RECEIVE` are really references to the elements of this array. Unlike the arguments to a `PROCEDURE` or `FUNCTION` block, which are "call by value," the arguments to a `NET_RECEIVE` block are "call by reference." Therefore assignment statements in `gsyn.mod`'s `NET_RECEIVE` block can change the

> On initialization, all elements of the weight vector other than the first one are automatically set to 0. However, a `NET_RECEIVE` block may have its own `INITIAL` block, and this can contain statements that assign nonzero values to `NetCon` "states." Such an `INITIAL` block is executed when `finitialize()` is called.

values of variables that belong to the `NetCon` object, and this means that the `NetCon`'s weight vector can be used to hold stream-specific state information. In the context of this particular example, each connection has its own [G], so `gsyn` uses "stream-specific plasticity" to represent "synapse-specific plasticity."

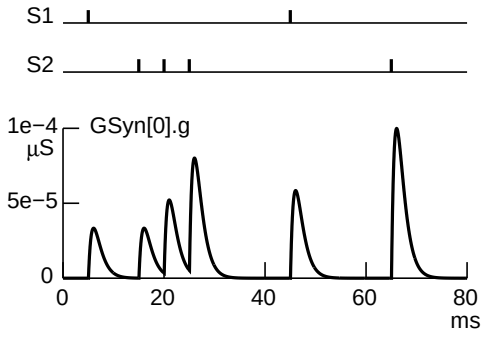


Figure 10.9. Simulation results from the model shown in Fig. 10.6 when the synaptic mechanism is `GSyn`. Note stream-specific use-dependent plasticity.

To illustrate the operation of this mechanism, imagine the network of Fig. 10.6 with a single `GSyn` driven by the two spike trains shown in Fig. 10.9. This emulates two synapses that are electrotonically close to each other, but with separate pools of [G]. The train with spikes at 5 and 45 ms (S1) shows some potentiation of the second conductance transient, but the train that starts at 15 ms with a 200 Hz burst of three spikes displays a large initial potentiation that is even larger when tested after a 40 ms interval.

Example 10.6: saturating synapses

Several authors (e.g. (Destexhe et al. 1994a; Lytton 1996)) have used synaptic transmission mechanisms based on a simple conceptual model of transmitter-receptor interaction:

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

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

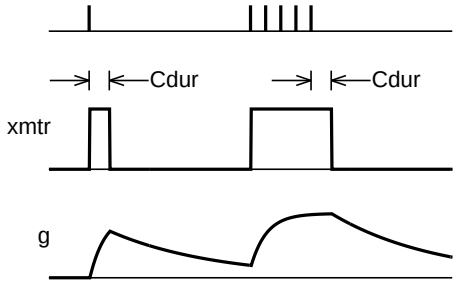


Figure 10.10. A saturating synapse model. A single presynaptic spike (top trace) causes a pulse of transmitter in the synaptic cleft with fixed duration (Cdur) and concentration (middle trace). This elicits a rapid increase of postsynaptic conductance followed by a slower decay (bottom trace). A high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

There is an ambiguity when one or more spikes arrive on a single stream during the onset state triggered by an earlier spike: should the mechanism ignore the "extra" spikes, concatenate onset states to make the transmitter pulse longer without increasing its concentration, or increase (summate) the transmitter concentration? Summation of transmitter requires the onset time constant to vary with transmitter concentration. This places transmitter summation outside the scope of the Destexhe/Lytton model, which assumes a fixed time constant for the onset state. We resolve this ambiguity by choosing concatenation, so that repetitive impulses on one stream produce a saturating conductance change (Fig. 10.10). However, conductance changes elicited by separate streams will summate.

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

```
: ampa.mod

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

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



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

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

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

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

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

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



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

Listing 10.5. `ampa.mod`

The `PARAMETER` block

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

The `STATE` block

This mechanism has two `STATE`s. `Ron` is the total conductance of all synapses that are in the onset state, and `Roff` is the total conductance of all synapses that are in the offset state. These are declared without units, so a units factor will have to be applied elsewhere (in this example, this is done in the `BREAKPOINT` block).

The `INITIAL` block

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

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

The BREAKPOINT and DERIVATIVE blocks

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

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

```
Roff' = -Beta*Roff
```

is obvious, and in

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

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

The NET_RECEIVE block

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

"External event" and "input event" are synonyms. We will use the former term as clarity dictates when contrasting them with self-events.

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

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 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 generates a self-event with delay given by the first argument and flag value given by the second argument. All the explicit arguments of this self-event will have the values of this particular `NetCon`, so when this self-event returns we will know how much "material" to switch from the onset to the offset state.

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

Handling of self-events

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

Artificial spiking cells

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

The previous section also showed how spike-triggered synaptic transmission makes extensive use of the network connection class to define connections between cells. The typical `NetCon` object watches a source cell for the occurrence of a spike, and then, after some delay, delivers a weighted event to a target synaptic mechanism, i.e. it is a metaphor for axonal spike propagation. More generally, a `NetCon` object can be regarded as a channel on which a stream of events generated at a source is transmitted to a target. The target can be a point process, a distributed mechanism, or an artificial neuron (e.g. an integrate and fire model). The effect of events on a target is specified in NMODL by statements in a `NET_RECEIVE` block, which is called only when an event has been delivered.

The event delivery system also opens up a large domain of simulations in which certain types of artificial spiking cells, and networks of them, can be simulated hundreds of times faster than with numerical integration methods. Discrete event simulation is possible when all the state variables of a model cell can be computed analytically from a

new set of initial conditions. That is, if an event occurs at time t_1, all state variables must be computable from the state values and time t_0 of the previous event. Since computations are performed only when an event occurs, total computation time is proportional to the number of events delivered and independent of the number of cells, number of connections, or problem time. Thus handling 100,000 spikes in one hour for 100 cells takes the same time as handling 100,000 spikes in 1 second for 1 cell.

Artificial spiking cells are implemented in NEURON as point processes, but unlike ordinary point processes, they can serve as targets and sources for `NetCon` objects. They can be targets because they have a `NET_RECEIVE` block, which specifies how incoming events from one or more `NetCon` objects are handled, and details the calculations necessary to generate outgoing events. They can also be sources because the same `NET_RECEIVE` block generates discrete output events which are delivered through one or more `NetCon` objects to targets.

The following examples analyze the three broad classes of integrate and fire cells that are built into NEURON. In order to emphasize how the event delivery system is used to implement the dynamics of these mechanisms, we have omitted many details from the NMODL listings. Ellipses indicate elisions, and listings include *italicized pseudocode* where necessary for clarity. Complete source code for all three of these cell classes is provided with NEURON.

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

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

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

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

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

PARAMETER { tau = 10 (ms) }
```



```
ASSIGNED {
  m
  t0 (ms)
}

INITIAL {
  m = 0
  t0 = 0
}

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

Listing 10.6. A basic implementation of `IntFire1`.

The `NEURON` block

As the introduction to this section mentions, artificial spiking cells are implemented in NEURON as point processes. The keyword `ARTIFICIAL_CELL` is in fact a synonym for `POINT_PROCESS`, but we use it as a deliberate reminder to ourselves that this model has a `NET_RECEIVE` block, lacks a `BREAKPOINT` block, and does not have to be associated with a section location or numerical integrator. Unlike other point processes, an artificial cell is isolated from the usual things that link mechanisms to each other: it does not refer to membrane potential `v` or any ions, and it does not use `POINTER` variables. Instead, the "outside" can affect it only by sending it discrete events, and it can only affect the "outside" by sending discrete events.

The `NET_RECEIVE` block

The mechanisms we have seen so far use `BREAKPOINT` and `KINETIC` or `DERIVATIVE` blocks to specify the calculations that are performed during a time step `dt`, but an artificial cell model does not have these blocks. Instead, calculations only take place when a new event arrives, and these are performed in the `NET_RECEIVE` block.

When a `NetCon` delivers a new event to an `IntFire1` cell, the present value of `m` is computed analytically and then `m` is incremented by the weight `w` of the event. According to the `NET_RECEIVE` block, the present value of `m` is found by applying an exponential decay to the value it had immediately after the previous event; therefore the code contains variable `t0` which keeps track of the last event time.

If an input event drives `m` to or above threshold, the `net_event(t)` statement notifies all `NetCon`s, for which this point process is the source, that it fired a spike at time `t` (the argument to `net_event()` can be any time at or later than the current time `t`). Then the cell resets `m` to 0. The code in Listing 10.6 imposes no limit on firing frequency--if a `NetCon` with `delay` of 0 and a `weight` of 1.1 has such an artificial cell

as both its source and target, the system will behave "properly," in the sense that events will be generated and delivered without time ever advancing. It is easy to prevent the occurrence of such a runaway stream of events (see ***Adding a refractory period*** below).

There is no threshold test overhead at every `dt` because `IntFire1` has no variable for `NetCon`s to watch. That is, this artificial spiking cell does not need the usual test for local membrane potential `v` to cross `NetCon.threshold`, which is essential at every time step for event generation with biophysical neuron models. Furthermore the event delivery system only places the earliest event to be delivered on the event queue. When that time finally arrives, all targets whose `NetCon`s have the same source and delay get the event delivery, and longer delay streams are put back on the event queue to await their specific delivery time.

Enhancements to the basic mechanism

Visualizing the membrane state variable

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

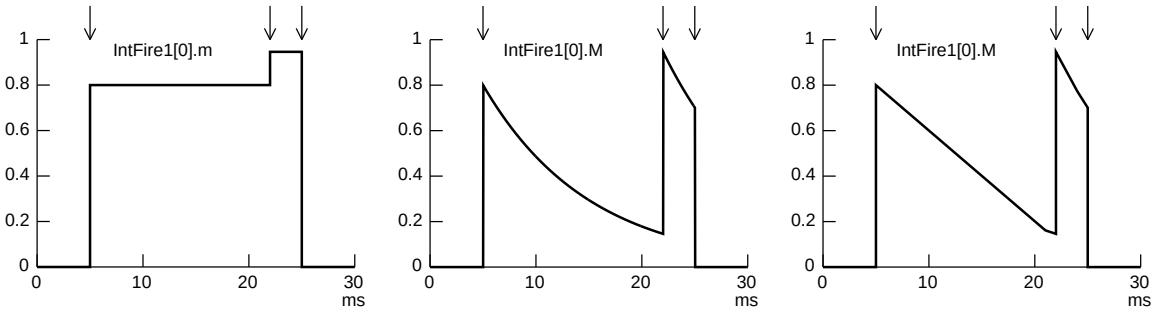


Figure 10.11. Response of an `IntFire1` cell with τ = 10 ms to input events with weight = 0.8 arriving at *t* = 5, 22, and 25 ms (arrows). The third input initiates a "spike." Left: The variable `m` is evaluated only when a new event arrives, so its plot looks like a staircase. A function can be included in `IntFire1`'s mod file (see text) to better indicate the time course of the membrane state variable *m*. Center: Plotting this function during a simulation with fixed `dt` (0.025 ms here) demonstrates the decay of *m* between events. Right: In a variable time step simulation, *m* appears to follow a sequence of linear ramps. This artifact is a consequence of the efficiency of adaptive integration, which computed analytical solutions at only a few instants, so the Graph tool could only draw lines from instant to instant.

This can be partially repaired by adding a function

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


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

Adding a refractory period

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

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

The `NET_RECEIVE` implementation is then

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

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

Recall from the saturating synapse example that the `flag` variable that accompanies an external event is 0. If this mechanism receives an event with a nonzero `flag`, it must be a self-event, i.e. an event generated by 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.

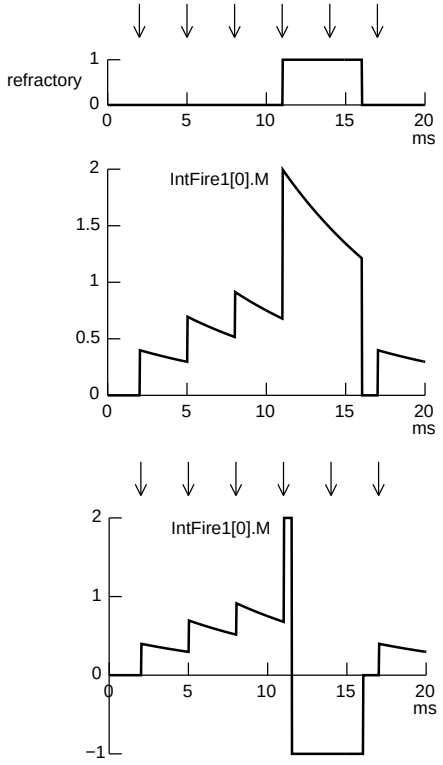


Figure 10.12. Response of an `IntFire1` cell with a 5 ms refractory interval to a run of inputs at 3 ms intervals (arrows), each with weight = 0.4. Top: The cell accepts inputs when `refractory` == 0. The fourth input (at 11 ms) drives the cell above threshold. This triggers an output event, increases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

Improved presentation of the membrane state variable

The performance in the top of Fig. 10.12 is satisfactory, but the model could be further improved by one relatively minor change. As it stands the `M` function shows an exponential decay during the refractory period, which is at best distracting and irrelevant to the operation of the model, and potentially misleading at worst. It would be better for `M` to follow a stereotyped time course, e.g. a brief positive pulse followed by a longer

negative pulse. This would not be confused with the subthreshold operation of the model, and it might be more suggestive of an action potential.

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

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

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

This demonstrates how visualization of cell operation can be enhanced by simple calculations of patterns for the spiking and refractory trajectories, with no overhead for cells that are not plotted. We must emphasize that the simulation calculations are analytic and performed only at event arrival, regardless of the refinements we introduced for the purpose of esthetics.

Sending an event to oneself to trigger deferred computation involves very little overhead, yet it allows elaborate calculations to be performed much more efficiently than if they were executed on a per `dt` basis. Self-events are heavily exploited in the implementation of `IntFire2` and `IntFire4`, which both offer greater kinetic complexity than `IntFire1`.

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

The `IntFire2` model, like `IntFire1`, has a membrane state variable *m* that follows first order kinetics with time constant τ_m. However, an input event to `IntFire2` does not affect *m* directly. Instead it produces a discontinuous change in a synaptic current state variable *i*. Between events, *i* decays with its own time constant τ_s toward a steady "bias" value specified by the parameter i_b. That is,

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

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

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

where $\tau_m < \tau_s$. Thus an input event produces a gradual change in *m* that is described by two time constants and approximates an alpha function if $\tau_m \approx \tau_s$. When *m* crosses a threshold of 1 in a positive direction, the cell fires, *m* is reset to 0, and integration resumes immediately, as shown in the bottom of Fig. 10.13. Note that *i* is not reset to 0, i.e. unlike `IntFire1`, firing of an `IntFire2` cell does not obliterate all traces of prior synaptic activation.

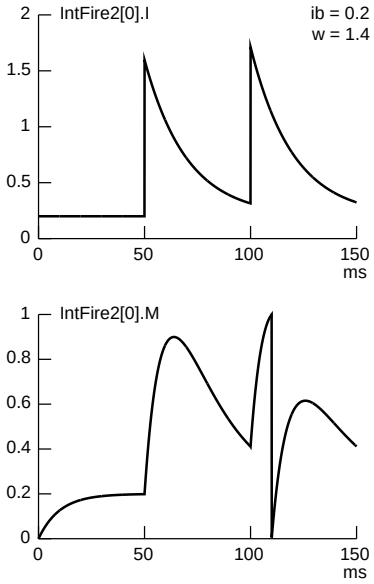


Figure 10.13. Top: Time course of synaptic current *i* in an `IntFire2` cell with τ_s = 20 ms and τ_m = 10 ms. This cell has bias current i_b = 0.2 and receives inputs with weight *w* = 1.4 at *t* = 50 and 100 ms. Bottom: The membrane state variable *m* of this cell is initially 0 and approaches the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

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

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

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

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

and

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

Implementation in NMODL

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

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

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

Listing 10.7. Key excerpts from `intfire2.mod`

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

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

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

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).

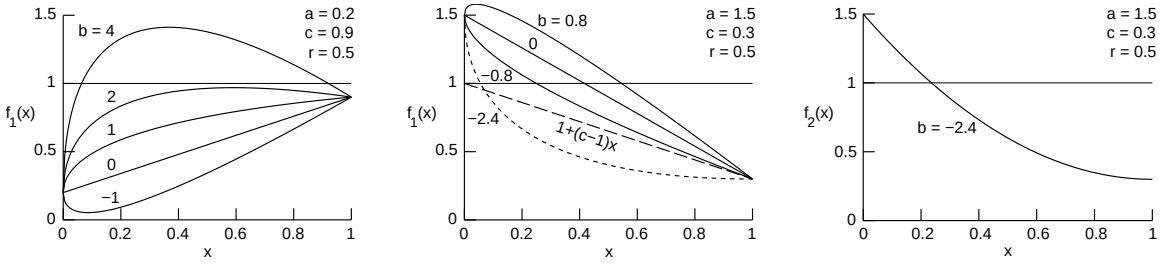


Figure 10.14. Plots of f_1 and f_2 computed for $r = 0.5$. See text for details.

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

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

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

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

Example 10.9: `IntFire4`, different synaptic time constants

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

The `IntFire4` mechanism addresses this need. Its dynamics are specified by four time constants: τ_e for a fast excitatory current, τ_{i_1} and τ_{i_2} for a slower inhibitory current, and τ_m for the even slower leaky "membrane" which integrates these currents. When the membrane state variable m reaches 1, the cell "fires," producing an output event and returning m to 0. This does not affect the other states of the model.

The differential equations that govern `IntFire4` are

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

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

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

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

where each k is a rate constant that equals the reciprocal of the corresponding time constant, and it is assumed that $k_e > k_{i_1} > k_{i_2} > k_m$ (i.e. $\tau_e < \tau_{i_1} < \tau_{i_2} < \tau_m$). An input event with weight $w > 0$ (i.e. an excitatory event) adds instantaneously to the excitatory current e. Equations 10.11 and 12, which define the inhibitory current i_2, are based on the reaction scheme

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

in which an input event with weight $w < 0$ (i.e. an inhibitory event) adds instantaneously to i_1. The constants a_e, a_{i_1}, and a_{i_2} are chosen to normalize the response of the states e, i_1, i_2, and m to input events (Fig. 10.15). Therefore an input with weight $w_e > 0$ (an "excitatory" input) produces a peak e of w_e and a maximum "membrane potential" m of w_e. Likewise, an input with weight $w_i < 0$ (an "inhibitory" input) produces an inhibitory current i_2 with a minimum of w_i and drives m to a minimum of w_i. Details of the analytic solution to these equations are presented in **Appendix A1: Mathematical analysis of IntFire4**.

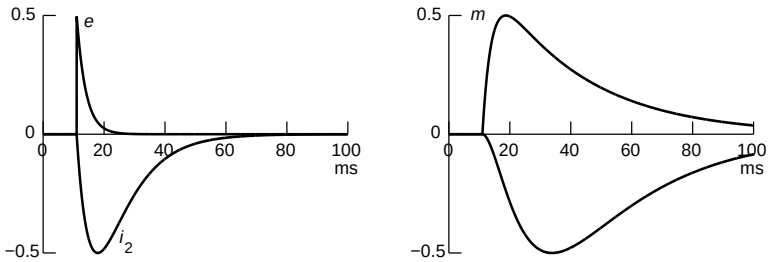


Figure 10.15. Left: Current generated by a single input event with weight 0.5 (e) or -0.5 (i_2). Right: The corresponding response of m. Parameters were $\tau_e = 3$, $\tau_{i_1} = 5$, $\tau_{i_2} = 10$, and $\tau_m = 30$ ms.

`IntFire4`, like `IntFire2`, finds the next firing time through successive approximation. However, `IntFire2` generally iterates to convergence every time an input event is received, whereas `IntFire4`'s algorithm implement a series of deferred Newton iterations by exploiting the downward convexity of the membrane potential trajectory and using NEURON's event delivery system. The result is an alternating sequence of self-events and single Newton iterations that converges to the correct firing time, yet remains computationally efficient in the face of heavy input event traffic.

This is illustrated in Fig. 10.16. If an event arrives at time t_0, values of $e(t_0)$, $i_1(t_0)$, $i_2(t_0)$, and $m(t_0)$ are calculated analytically. Should $m(t_0)$ be subthreshold, the self-event is moved to a new approximate firing time t_f that is based on the slope approximation to m

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

(Fig. 10.16 left and middle). If instead $m(t_0)$ reaches threshold, the cell "fires" so that `net_event()` is called (producing an output event that is picked up by all `NetCon`s for which this cell is a source) and m is reset to 0. The self-event is then moved to an approximate firing time that is computed from Eq. 10.15 using the values assigned to m and m' immediately after the "spike" (Fig. 10.16 right).

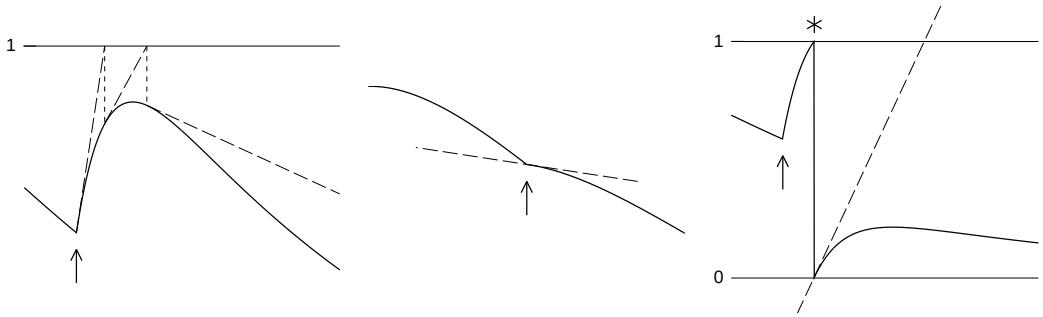


Figure 10.16. Excerpts from simulations of `IntFire4` cells showing time course of m. Arrival of an event (arrow = external event, vertical dotted line = self-event) triggers a Newton iteration. Slanted dashed lines are slope approximations to m immediately after an event. Left: Although Eq. 10.15 yields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

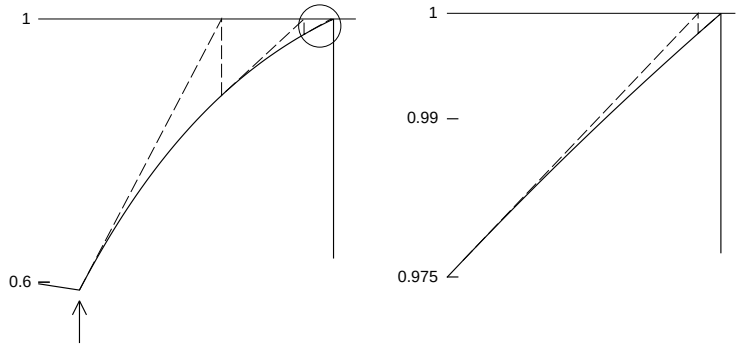


Figure 10.17. These magnified views of the trajectory from the right panel of Fig. 10.16 indicate how rapidly the event-driven Newton iterations converge to the next firing time. In this simulation, spike threshold was reached in four iterations after the excitatory input (arrow). The first two iterations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

The justification for this approach stems from several considerations. The first of these is that t_f is never later than the true firing time. This assertion, which we prove in **Appendix A1**, is of central importance because the simulation would otherwise be in error.

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

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

Other comments regarding artificial spiking cells

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

References

De Schutter, E., Angstadt, J.D., and Calabrese, R.L. A model of graded synaptic transmission for use in dynamic network simulations. *J. Neurophysiol.* 69:1225-1235, 1993.

Destexhe, A., Mainen, Z.F., and Sejnowski, T.J. An efficient method for computing synaptic conductances based on a kinetic model of receptor binding. *Neural Computation* 6:14-18, 1994a.

Destexhe, A., Mainen, Z.F., and Sejnowski, T.J. Synthesis of models for excitable membranes, synaptic transmission, and neuromodulation using a common kinetic formalism. *J. Comput. Neurosci.* 1:195-231, 1994b.

Destexhe, A., Mainen, Z.F., and Sejnowski, T.J. Kinetic models of synaptic transmission. In: *Methods in Neuronal Modeling*, edited by C. Koch and I. Segev. Cambridge, MA: MIT Press, 1998, p. 1-25.

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

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

Chapter 10 Index

local vs. nonlocal 1, 2

Chapter 11
Modeling networks

NEURON was initially developed to handle models of individual cells or parts of cells, in which complex membrane properties and extended geometry play important roles (Hines 1989; 1993; 1995). However, as the research interests of experimental and theoretical neuroscientists evolved, NEURON has been revised to meet their changing needs. Since the early 1990s it has been used to model networks of biological neurons (e.g. (Destexhe et al. 1993; Lytton et al. 1997; Sohal et al. 2000)). This work stimulated the development of powerful strategies that increase the convenience and efficiency of creating, managing, and exercising such models (Destexhe et al. 1994; Lytton 1996; Hines and Carnevale 2000). Increasing research activity on networks of spiking neurons (e.g. (Riecke et al. 1997; Maass and Bishop 1999)) prompted further enhancements to NEURON, such as inclusion of an event delivery system and development of the `NetCon` (network connection) class (see **Chapter 10**).

Consequently, since the latter 1990s, NEURON has been capable of efficient simulations of networks that may include biophysical neuron models and/or artificial spiking neurons. biophysical neuron models are built around representations of the biophysical mechanisms that are involved in neuronal function, so they have sections, density mechanisms, and synapses (see **Chapter 5**). A synapse onto a biophysical neuron model is a point process with a `NET_RECEIVE` block that affects membrane current (e.g. `ExpSyn`) or a second messenger (see **Chapter 10**). The membrane potential of a biophysical neuron model is governed by complex, interacting nonlinear mechanisms, and spatial nonuniformities may also be present, so numerical integration is required to advance the solution in time.

What could be more oxymoronic than "real model neuron"?

As we discussed in **Chapter 10**, artificial spiking neurons are really point processes with a `NET_RECEIVE` block that calls `net_event()` (e.g. `IntFire1`). The "membrane state variable" of an artificial neuron has very simple dynamics, and space is not a factor, so the time course of the membrane state is known analytically and it is relatively easy to compute when the next spike will occur. Since artificial neurons do not need numerical integration, they can be used in discrete event simulations that run several orders of magnitude faster than simulations involving biophysical neuron models. Their simplicity also makes it very easy to work with them. Consequently, artificial spiking neurons are particularly useful for prototyping network models.

In this chapter we present an example of how to build network models by combining the strengths of the GUI and `hoc` programming. The GUI tools for creating and managing network models are most appropriate for exploratory simulations of small nets. Once you have set up and tested a small network with the GUI, a click of a button creates

a hoc file that contains reusable cell class definitions and procedures. This eliminates the laborious, error-prone task of writing "boilerplate" code. Instead, you can just combine NEURON's automatically generated code with your own hoc programming to quickly construct large scale nets with complex architectures. Of course, network models can be constructed entirely by writing hoc code, and NEURON's WWW site contains links to a tutorial for doing just that (Gillies and Sterratt, 2004). However, by taking advantage of GUI shortcuts, you'll save valuable time that can be used to do more research with your models.

Building a simple network with the GUI

Regardless of whether you use the GUI or write hoc code, creating and using a network model involves these basic steps:

1. Define the types of cells.
2. Create each cell in the network.
3. Connect the cells.
4. Set up instrumentation for adjusting model parameters and recording and/or displaying simulation results.
5. Set up controls for running simulations.

We will demonstrate this process by constructing a network model that can be used to examine the contributions of synaptic, cellular, and network properties to the emergence of synchronous and/or correlated firing patterns.

Conceptual model

The conceptual model is a fully connected network, i.e. each cell projects to all other cells, but not to itself (Fig. 11.1 left). All conduction delays and synaptic latencies are identical.

The cells are spontaneously active integrate and fire neurons, similar to those that we discussed in **Chapter 10**. All cells have the same time constant and firing threshold, but in isolation each has its own natural interspike interval (ISI), and the ISIs of the population are distributed uniformly over a fixed range (Fig. 11.1 right).

Figure 11.2 illustrates the dynamics of these cells. Each spike is followed by a "post-spike" hyperpolarization of the membrane state variable *m*, which then decays monoexponentially toward a suprathreshold level. When *m* reaches threshold (1), it triggers another spike and the cycle repeats. A synaptic input hyperpolarizes the cell and prolongs the ISI in which it occurred, shifting subsequent spikes to later times. Each input produces the same hyperpolarization of *m*, regardless of where in the ISI it falls. Even so, the shift of the spike train depends on the timing of the input. If it arrives shortly after a spike, the additional hyperpolarization decays quickly and the spike train shifts by only a

small amount (Fig. 11.2 left). An input that arrives late in the ISI can cause a much larger shift in the subsequent spike train (Fig. 11.2 right).

Our task is to create a model that will allow us to examine how synaptic weight, membrane time constant and natural firing frequency, number of cells and conduction latency interact to produce synchronized or correlated spiking in this network.

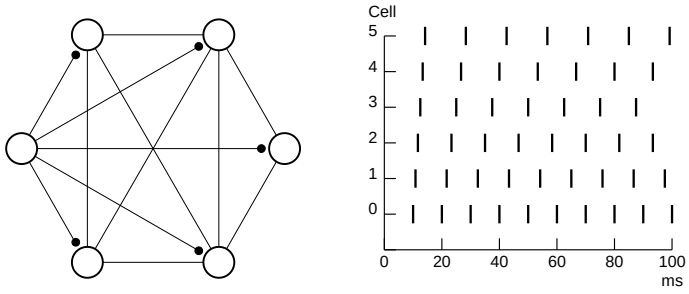


Figure 11.1. Left: An example of a fully connected net. Thin lines indicate reciprocal connections between each pair of cells, and thick lines mark projections from one cell to its targets. Right: When disconnected from each other, every cell has its own natural firing frequency.

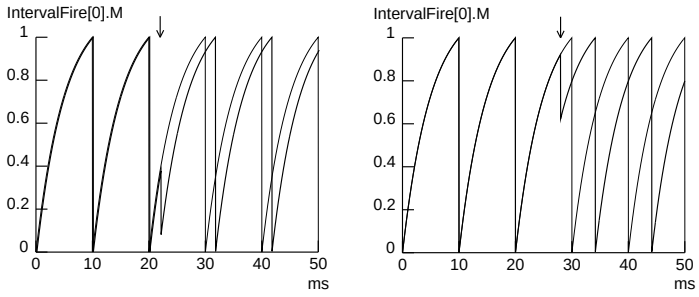


Figure 11.2. Time course of the membrane state variable *m* in the absence (thin traces) and presence (thick traces) of an inhibitory input. Notice that *m* follows a monoexponential "depolarizing" time course which carries it toward a suprathreshold level. When *m* reaches 1, a spike is triggered and *m* is reset to 0 ("post-spike hyperpolarization"). An inhibitory synaptic event causes the same hyperpolarizing shift of *m* no matter where in the ISI it arrives, but its effect on later spike times depends on its relative position in the ISI. Left: Inhibitory events that occur early in the ISI decay quickly, so following spikes are shifted to slightly later times. Right: An inhibitory event that occurs late in the ISI has a longer lasting effect and causes a greater delay of the subsequent spike train.

Adding a new artificial spiking cell to NEURON

Before we start to build this network, we need to add a new kind of artificial spiking cell to NEURON. Our model will use cells whose membrane state variable *m* is governed by the equation

$$\tau \frac{dm}{dt} + m = m_{\infty} \qquad \text{Eq. 11.3}$$

where $m_\infty > 1$ and is set to a value that produces spontaneous firing with the desired ISI.

An input event with weight *w* adds instantaneously to *m*, and if *m* reaches or exceeds the threshold value of 1, the cell "fires," producing an output event and returning *m* to 0. We will call this the `IntervalFire` model, and the NMODL code for it is shown in Listing 11.1. `IntervalFire` has essentially the same dynamics as `IntFire1`, but because its membrane state relaxes toward a suprathreshold value, it uses a `firetime()` function to compute the time of the next spike (see discussions of `IntFire1` and `IntFire2` in **Chapter 10**).

```
NEURON {
  ARTIFICIAL_CELL IntervalFire
  RANGE tau, m, invl
}

PARAMETER {
  tau = 5 (ms)   <1e-9,1e9>
  invl = 10 (ms) <1e-9,1e9>
}

ASSIGNED {
  m
  minf
  t0(ms)
}

INITIAL {
  minf = 1/(1 - exp(-invl/tau)) : so natural spike interval is invl
  m = 0
  t0 = t
  net_send(firetime(), 1)
}

NET_RECEIVE (w) {
  m = M()
  t0 = t
  if (flag == 0) {
    m = m + w
    if (m > 1) {
      m = 0
      net_event(t)
    }
    net_move(t+firetime())
  } else {
    net_event(t)
    m = 0
    net_send(firetime(), 1)
  }
}

FUNCTION firetime()(ms) { : m < 1 and minf > 1
  firetime = tau*log((minf-m)/(minf - 1))
}
```



```
FUNCTION M() {
  M = minf + (m - minf)*exp(-(t - t0)/tau)
}
```

Listing 11.1. NMODL implementation of `IntervalFire`. Figures 11.1 (right) and 11.3 illustrate its operation.

Creating a prototype net with the GUI

After we compile the code in Listing 11.1 (see **Chapter 9**), when we launch `nrngui` these lines should appear at the end of NEURON's startup message

```
Additional mechanisms from files
 invlfire.mod
```

to reassure us that what was defined in `invlfire.mod`--i.e. the `IntervalFire` cell class--is now available. We are ready to use the GUI to build and test a prototype net.

1. Define the types of cells

This involves using the existing cell classes to create the types of cells that we will employ in our network. Our network contains artificial spiking cells, so we need an ArtCellGUI tool, which we get by clicking on Build / NetWork Cell / Artificial Cell in the NEURON Main Menu toolbar (Fig. 11.3).

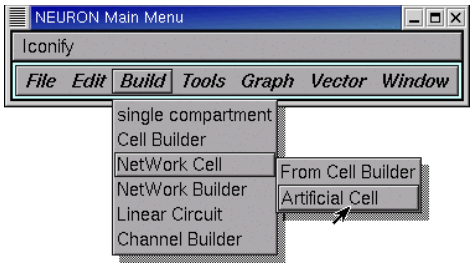


Figure 11.3. Using the NEURON Main Menu to bring up an ArtCellGUI tool.

The gray area in the lower left corner of the ArtCellGUI tool displays a list of the types of artificial spiking cells that will be available to the NetWork Builder. It starts out empty because we haven't done anything yet (Fig. 11.4). To remedy this, click on New and scroll down to select IntervalFire (Fig. 11.5 left), and then release the mouse button. The Artificial Cell types list now contains a new item called IntervalFire, and the right panel of the ArtCellGUI tool shows the user-settable parameters for this cell type (Fig. 11.5 right). These default values are fine for our initial exploratory simulations, so we'll leave them as is.

However, there is one small change that will make it easier to use the NetWork Builder: IntervalFire is a big word, and the NetWork Builder's canvas is relatively small. To avoid clutter, let's give our cell type a short, unique name, like IF (see Figs. 11.6 and 11.7).

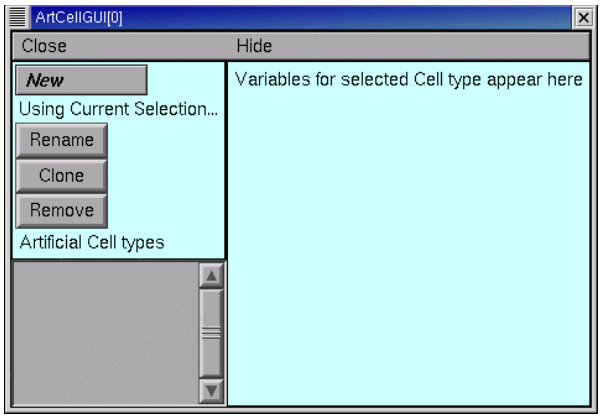


Figure 11.4. The ArtCellGUI tool starts with an empty Artificial Cell types list.

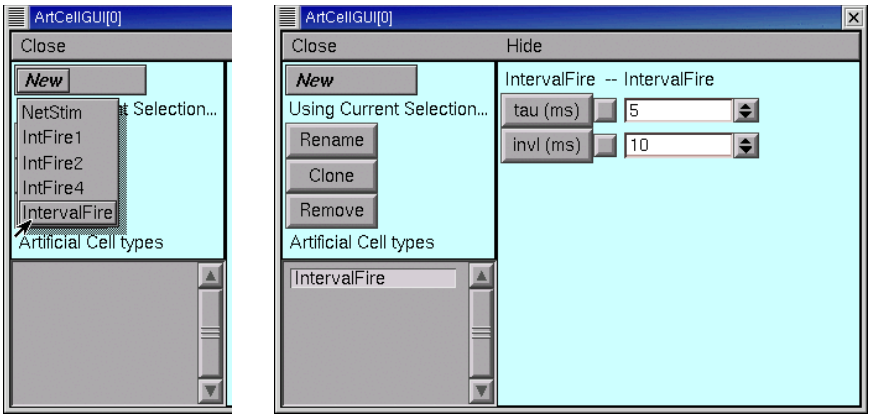
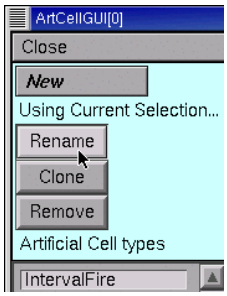


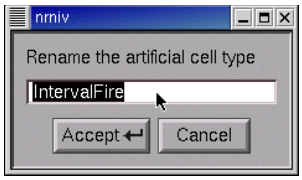
Figure 11.5. Click on New / IntervalFire to add it to the Artificial Cell types list.

Figure 11.6. Changing the name of one of the Artificial Cell types.

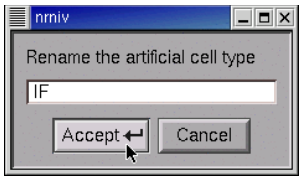
To change the name of one of the Artificial Cell types, select it (if it isn't already selected) and then click on the Rename button.



This pops up a window with a string editor field. Click in the field . . .



. . . change the name to IF, and then click the Accept button.



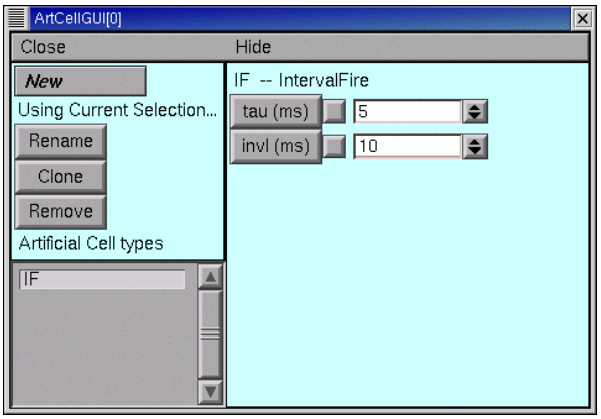


Figure 11.7. The ArtCellGUI tool after renaming the cell type. The right panel shows that IF is based on the IntervalFire class.

Now that we have configured the ArtCellGUI tool, it would be a good idea to save everything to a session file with NEURON Main Menu / File / save session (also see Fig. 1.23 and **Save the model cell** in **Chapter 1**). If you like, you may hide the ArtCellGUI tool by clicking on Hide just above the drag bar, but don't close it--the NetWork Builder will need it to exist.

2. Create each cell in the network

Having specified the cell types that will be used in the network, we are ready to use the NetWork Builder to create each cell in the network and connect them to each other. In truth, we'll just be creating the specification of each cell in the net; no cells are really created and there is no network until the Create button in the NetWork Builder is ON.

To get a NetWork Builder, click on NEURON Main Menu / Build / NetWork Builder (Fig. 11.8).

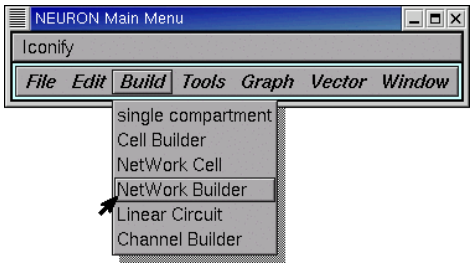


Figure 11.8. Bringing up a NetWork Builder.

The NetWork Builder's drag bar reveals that this tool is an instance of the `NetGUI` class (see Fig. 11.9).

The right panel of a NetWork Builder is a canvas for laying out the network. The "palette" for this canvas is a menu of the cell types that were created with the ArtCellGUI tool. These names appear along the upper left edge of the canvas (for this example, a limited palette indeed: IF is the only cell type). Context-dependent hints are displayed at the top of the canvas.

The left panel of a NetWork Builder contains a set of buttons that control its operation. When a NetWork Builder is first created, its Locate radio button is automatically ON. This means that the NetWork Builder is ready for us to create new cells. We do this by merely following the hint (Fig. 11.10). Notice that the cell names are generated by concatenating the base name (name of the cell type) with a number that starts at 0 and increases by 1 for each new cell. We'll say more about cell names in **A word about cell names** under **7. Caveats and other comments** below.

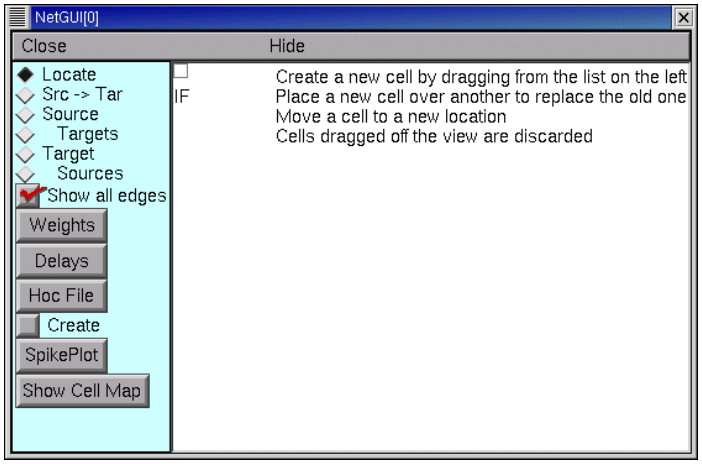
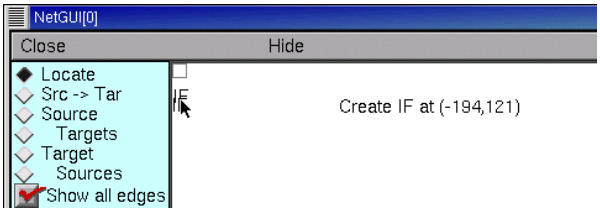


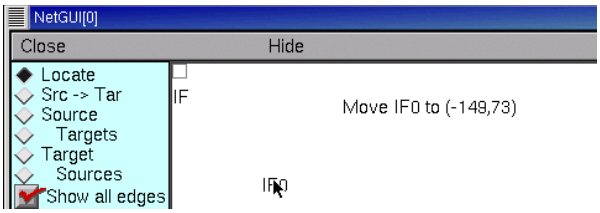
Figure 11.9. A new NetWork Builder.

Figure 11.10. Creating new cells in the NetWork Builder.

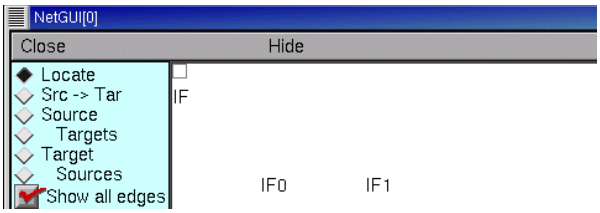
To create a new cell, click on one of the items in the palette (in this example, the only item is IF) and hold the mouse button down . . .



while dragging the new cell to a convenient location on the canvas. Release the mouse button, and you will see a new cell labeled IF0.



After you create a second IF cell, the NetWork Builder should look like this.



If the mouse button is released while the cursor is close to one of the palette items, the new cell will be hard to select since palette item selection takes precedence over selection of a cell. If this happens, just select Translate in the canvas's secondary menu (the canvas is just a modified graph!) and then left click on the canvas and drag it to the right (if you have a three button mouse, or a mouse with a scroll wheel, don't bother with the canvas's menu--just click on the middle button or scroll wheel and drag the canvas). This will pull the cell out from under the palette items, which never move from their position along the left edge of the canvas. Finally, click on one of the radio buttons (Locate, Src -> Tar, etc.) and continue working with the NetWork Builder.

3. Connect the cells

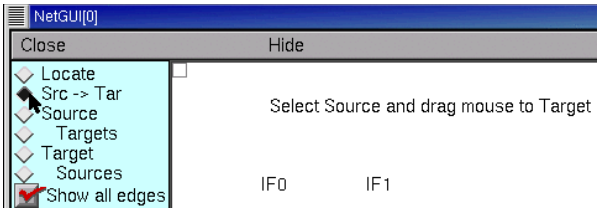
Connecting the cells entails two closely related tasks: setting up the network's architecture, and specifying the delays and weights of these connections.

Setting up network architecture

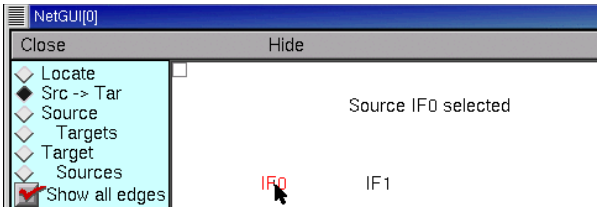
To set up the architecture, we click on the Src -> Tar radio button, read the new hint in the canvas, and do what it says (Fig. 11.11).

Figure 11.11. Setting up network architecture.

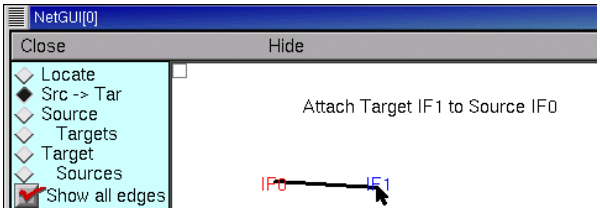
Clicking on the Src -> Tar button brings out a new hint.



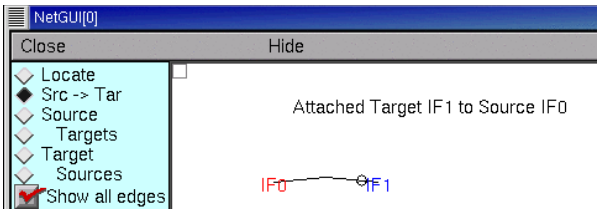
So we click on IF0 and hold the mouse button down while dragging the cursor toward IF1. A thin "rubber band" line will stretch from IF0 to the cursor.



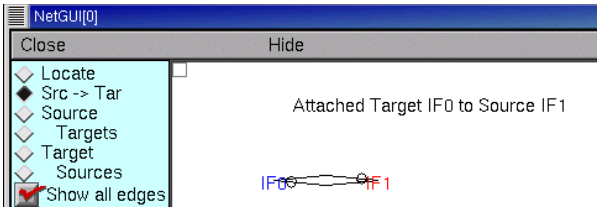
When the cursor is on top of IF1, the rubber band becomes a thick black line, and the hint changes to the message shown here.



To complete the attachment, we just release the mouse button. The projection ("edge") from IF0 to IF1 will appear as a thin line with a slight bend near its midpoint. The O marks the target end of this connection.



Making the reciprocal connection requires only that we click on IF1, drag to IF0, and release the mouse button.



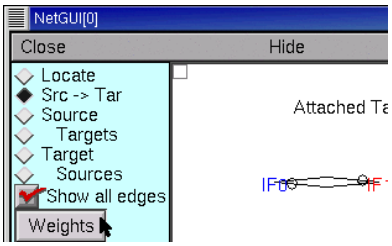
This is a good time to save everything to a session file.

Specifying delays and weights

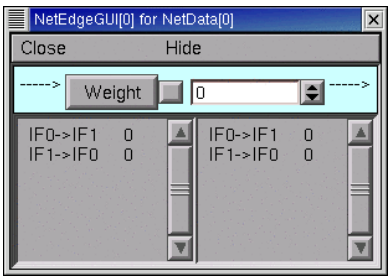
The default initial value of all synaptic weights is 0, i.e. a presynaptic cell will have no effect on its postsynaptic targets. The NetWork Builder has a special tool that we can use to change the weights to what we want (Fig. 11.12).

Figure 11.12. Setting the synaptic weights.

Clicking on the Weights button in the NetWork Builder . . .

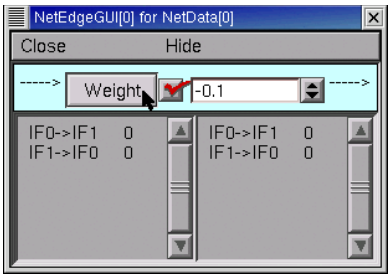


. . . brings up a tool for specifying synaptic weights. The top of this tool has a numeric field with its associated spinner and button (labeled Weight). The value in the numeric field can be set in the usual ways (direct entry, using the spinner, etc.), but note the arrows, which suggest other possibilities.



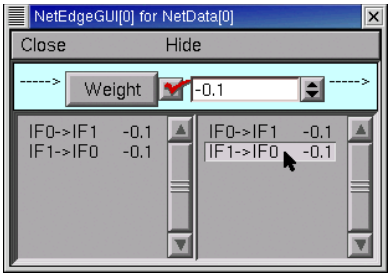
The bottom of the weights tool contains two panels that list the weights of all synaptic connections (aka "edges" in graph theory). Clicking on a connection in the left list copies from the connection to the numeric field, and clicking on a connection in the right list copies from the numeric field to the connection.

Let's give both synapses a weight of -0.1 (mild inhibition). First we change Weight to -0.1 . . .



. . . and then we click on IF0->IF1 and IF1->IF0 in the right panel. We're finished when the weights tool looks like this.

Now we can close this window. If we need it again, clicking on the NetWork Builder's Weights button will bring it back.



All delays are 1 ms by default, which is fine for our purposes. If we wanted to change this to something else, we would click on the NetWork Builder's Delays button (see Fig. 11.9) to bring up a tool for setting delays. The delay tool works just like the weight tool.

At this point, the ArtCellGUI tool plus the NetWork Builder together constitute a complete specification of our network model. We should definitely save another session file before doing anything else!

Now we have a decision to make. We could use the NetWork Builder to create a hoc file that, when executed, would create an instance of our network model. A better choice is to use the GUI to test our model. If there are any problems with what we have done so far, this is a good time to find out and make the necessary corrections.

However, before we can run tests, there must first be something to test. We have a network specification, but no network. As we pointed out earlier in **2. Create each cell in the network**, the network doesn't really exist yet. Clicking on the Create button in the NetWork Builder fixes that (Fig. 11.13).

4. Set up instrumentation

We want to see what our network does, and to explore how its behavior is affected by model parameters. Clicking on the SpikePlot button in the NetWork Builder brings up a tool that will show the input and output spike trains (Fig. 11.14).

We already know how to adjust model parameters. With the NetWork Builder we can change synaptic weights and delays, and the IF cells' properties can be changed with the ArtCellGUI tool. Suddenly, we realize that both IF cells will have the same time constant and firing rate. No problem--our goal is to combine the strengths of the GUI and hoc. We will take care of this later, by combining the hoc code that the NetWork Builder generates with our own hoc code. Using a few lines of hoc, we can easily assign unique firing

rates across the entire population of IF cells. And if we insisted on sticking with GUI tools to the bitter end, we could just bring up a PointProcessGroupManager (NEURON Main Menu / Tools / Point Processes / Managers / Point Group), which would allow us to control the attributes of each cell in our network individually.

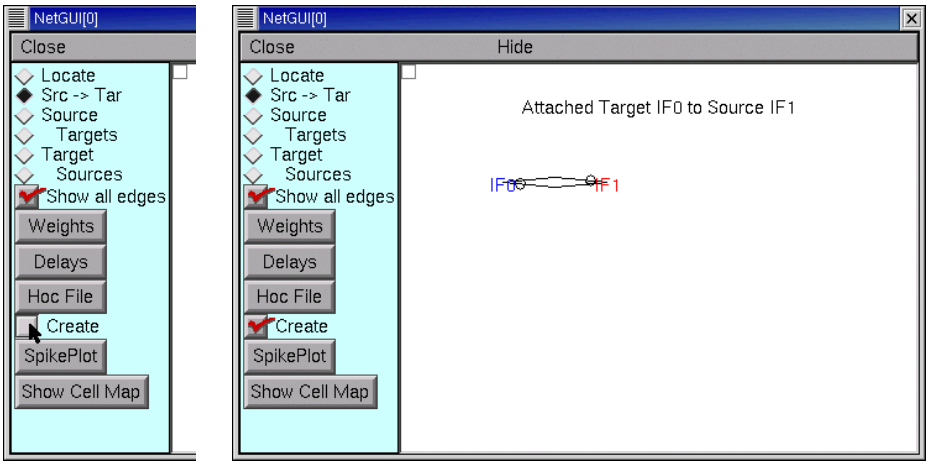


Figure 11.13. Left: Toggling the Create button ON causes the network specification to be executed. Right: Once Create is ON, the representation of the network is available for NEURON's computational engine to use in a simulation.

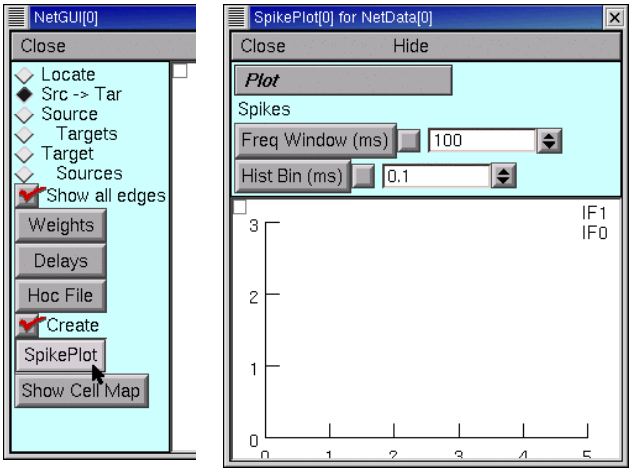


Figure 11.14. The NetWork Builder's SpikePlot button (left) brings up a tool for displaying and analyzing spike trains (right).

5. Set up controls for running simulations

At a minimum, we need a RunControl panel (NEURON Main Menu / Tools / RunControl, as shown in **5. Set up controls for running the simulation** in **Chapter 1**). Also, since our network contains only artificial spiking neurons, we can use adaptive

integration to achieve extremely fast, discrete event simulations. We'll need a VariableTimeStep panel (NEURON Main Menu / Tools / VariableStepControl (Fig. 11.15)), which makes it easy to choose between fixed time step or adaptive integration (Fig. 11.16).

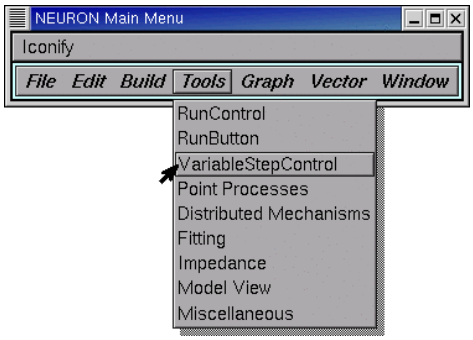


Figure 11.15. Bringing up a VariableTimeStep panel.

Figure 11.16. Toggling adaptive integration ON and OFF.

The VariableTimeStep panel's Use variable dt checkbox is empty, which means that adaptive integration is off.

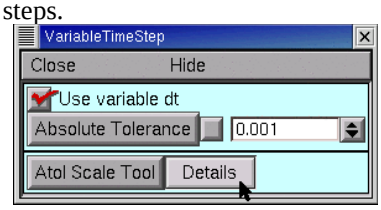
To turn adaptive integration ON, we click on the Use variable dt checkbox.

The check mark in the Use variable dt checkbox tells us that adaptive integration is ON. Clicking on this checkbox again will turn it back OFF so that fixed time steps are used.

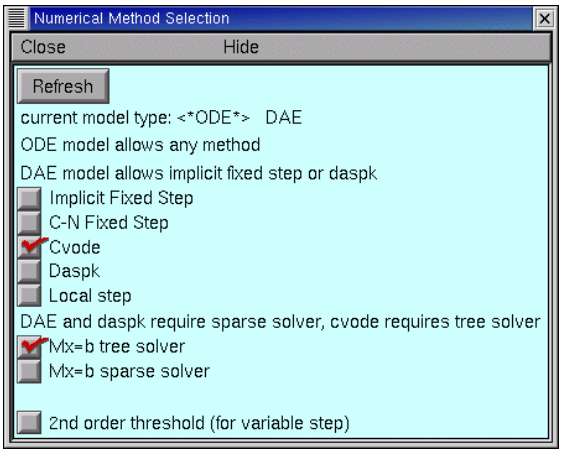
Adaptive integration can use either global or local time steps, each of which has its own particular strengths and weaknesses (see **Adaptive integrators** in **Chapter 7**). The VariableTimeStep panel's default setting is to use global time steps, which is best for models of single cells or perfectly synchronous networks. Our toy network has two identical cells connected by identical synapses, so we would expect them to fire synchronously. However, when we build our net with hoc code, the cells will all have different natural firing frequencies, and who can tell in advance that they will achieve perfect synchrony? Besides, this is a tutorial, so let's use local time steps (Fig. 11.17).

Figure 11.17. Toggling between global and local time steps.

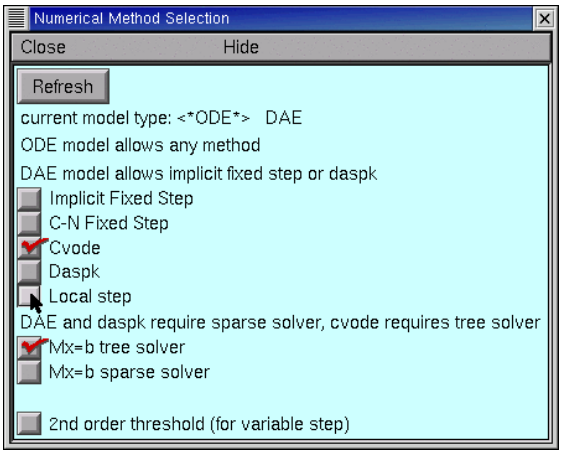
To specify whether to use global or local time steps, we first click on the VariableTimeStep panel's Details button.



We are concerned with the Local step checkbox, which is empty. To activate the use of local variable time steps . . .

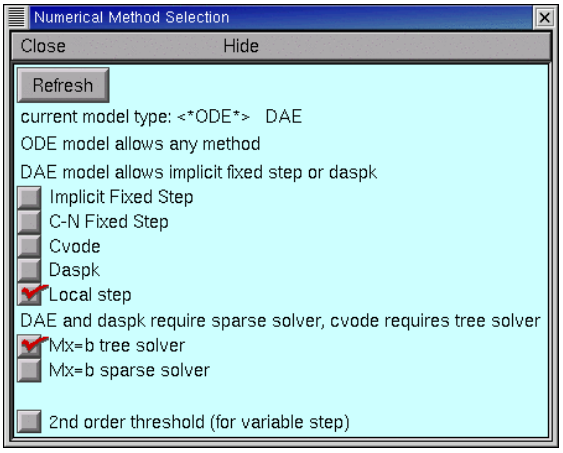


. . . we just click on the Local step checkbox . . .



. . . and now each cell in our network will advance with its own time step. If we want to restore global time steps, we can just click on the Cvode button.

Now we can close this panel; should we need it again, we only have to click on the VariableTimeStep panel's Details button.



After rearrangement, the various windows we have created should look something like Fig. 11.18. The tools we used to specify the network are on the left, simulation controls are in the middle, and the display of simulation results is on the right. Quick, save it to a session file!

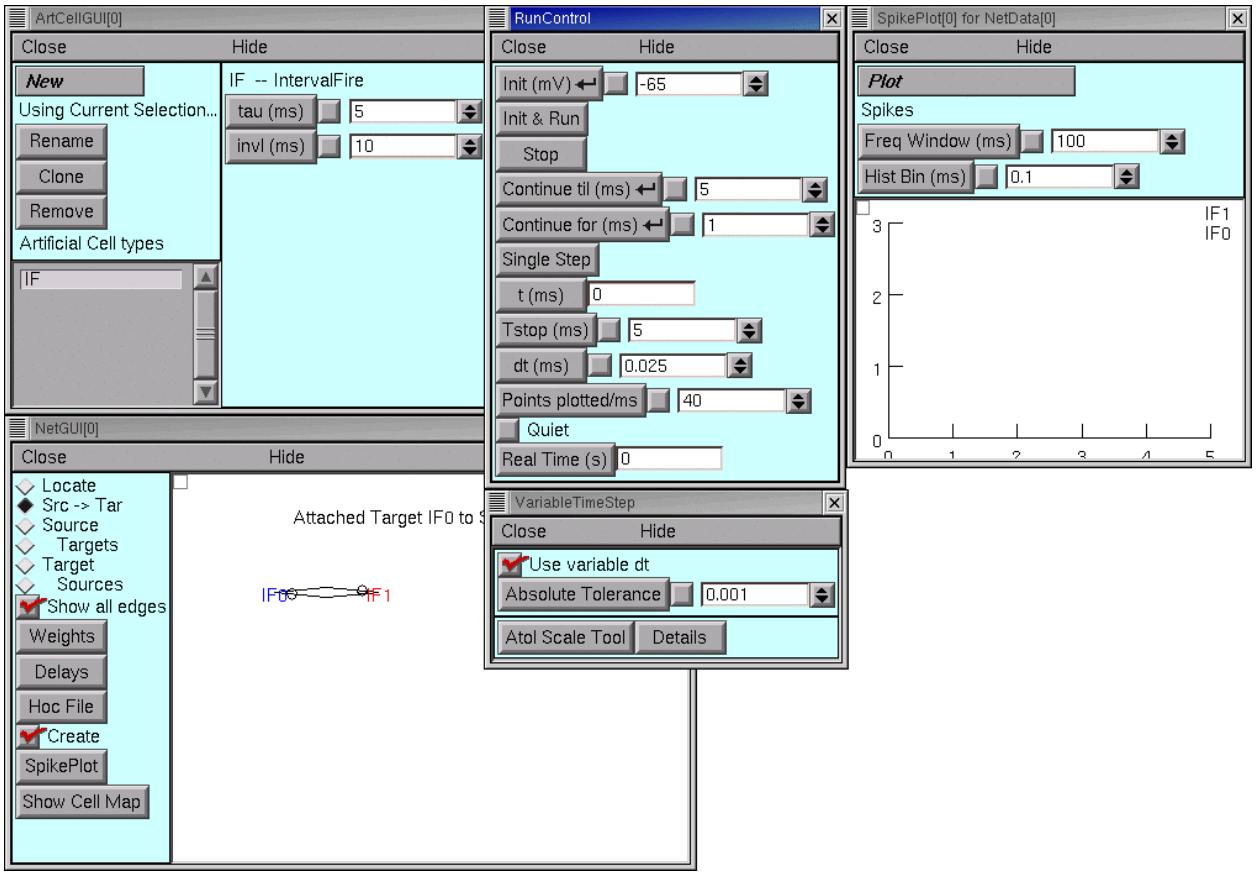


Figure 11.18. The completed model with controls for running simulations and displaying results.

6. Run a simulation

This is almost too easy. Clicking on Init & Run in the RunControl panel, we see--nothing! Well, almost nothing. The t field in the RunControl panel shows us that time advanced from 0 to 5 ms, but there were no spikes. A glance at the ArtCellGUI tool tells us why: invl is 5 ms, which means that our cells won't fire their first spikes for another 5 ms. Let's change Tstop to 200 ms so we'll get a lot of spikes, and try again. This time we're successful (Fig. 11.19).

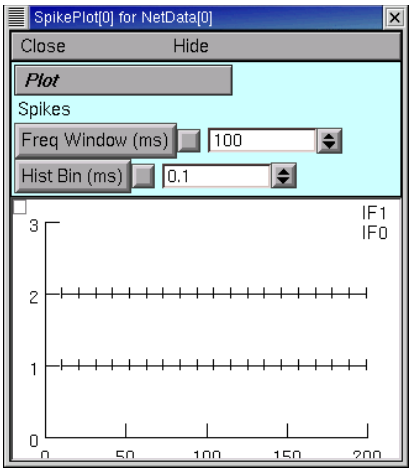


Figure 11.19. The SpikePlot shows the spike trains generated by the cells in our network model. Note that rasters correspond to cell names from top to bottom, and that the raster for cell *i* is plotted along the line $y = i + 1$.

7. Caveats and other comments

Changing the properties of an existing network

As we have seen, the ArtCellGUI tool is used to specify what artificial spiking cell types are available to a NetWork Builder. The same ArtCellGUI tool can be used to adjust the parameters of those cells, and such changes take effect immediately, even if the network already exists (i.e. even if the NetWork Builder's Create button is ON).

The NetReadyCellGUI tool (NEURON Main Menu / Build / NetWork Cell / From Cell Builder) is used to configure biophysical neuron model types for use with a NetWork Builder. In fact, we would use a separate NetReadyCellGUI instance for each different type of biophysical neuron model we wanted to use in the net. The NetReadyCellGUI tool has its own CellBuilder for specifying topology, geometry, and biophysical properties, plus a SynapseTypes tool for adding synaptic mechanisms to the cell (see the tutorial at `http://www.neuron.yale.edu/neuron/docs/netbuild/main.html`). However, changes made with a NetReadyCellGUI tool do *not* affect an existing network; instead, it is necessary to save a session file, exit NEURON, restart and reload the session file.

What about changes to the network itself? Any changes whatsoever can be made in the NetWork Builder, as long as its Create button is OFF. Once it is ON, some changes are possible (e.g. adding new cells and synaptic connections to an existing network), but additional actions may be required (a pre-existing SpikePlot will not show spike trains from new cells), and there is a risk of introducing a mismatch between one's conceptual model and what is actually in the computer. The best policy is to toggle Create OFF (see Fig. 11.20), make whatever changes are needed, save everything to a session file, exit NEURON, and then restart and load the new session file.

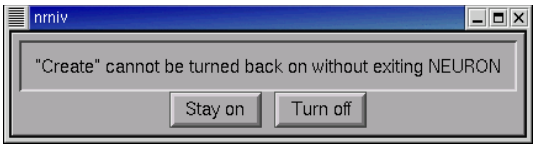


Figure 11.20. Trying to turn Create OFF brings up this window, which offers the opportunity to change one's mind. Select Turn off if it is necessary to make substantial changes to an existing network in the NetWork Builder.

A word about cell names

As we mentioned above in **2. Create each cell in the network**, the cell names that appear in the NetWork Builder are generated automatically by concatenating the name of the cell type with a sequence of numbers that starts at 0 and increases by 1 for each additional cell. But that's only part of the story. These are really only short "nicknames," a stratagem for preventing the NetWork Builder and its associated tools from being cluttered with long character strings.

This is fine as long as the NetWork Builder does everything we want. But suppose we need to use one of NEURON's other GUI tools, or we have to write some hoc code that refers to one of our model's cells? For example, we might have a network that includes a biophysical neuron model, and we want to see the time course of somatic membrane potential. In that case, it is absolutely necessary to know the actual cell names.

That's where the NetWork Builder's Cell Map comes in. Clicking on Show Cell Map brings up a small window that often needs to be widened by clicking and dragging on its left or right margin (Fig. 11.21). Now we realize that, when we used the ArtCellGUI tool to create an IF cell "type," we were actually specifying a new cell class whose name is a concatenation of our "type" (IF), an underscore character, and the name of the root class (the name of the class that we based IF on, which was IntervalFire).

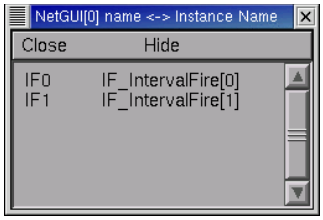


Figure 11.21. The Cell Map for our toy network. See text for details.

Combining the GUI and programming

Creating a hoc file from the NetWork Builder

Having tested our prototype model, we are now ready to write a hoc file that can be mined for reusable code. Clicking on the Hoc File button in the NetWork Builder brings up a tool that looks much like what we used to specify file name and location when

saving a session file. Once we're satisfied with our choices, clicking on this tool's "Open" button writes the hoc file (yes, the button should say Close). This file, which we will call prototype.hoc, is presented in Listing 11.2, and executing it would recreate the toy network that we just built with the NetWork Builder.

```
// NetGUI default section. Artificial cells, if any, are located here.
  create acell_home_
  access acell_home_

//Network cell templates
//Artificial cells
//   IF_IntervalFire


begintemplate IF_IntervalFire
public pp, connect2target, x, y, z, position, is_art
external acell_home_
objref pp
proc init() {
  acell_home_ pp = new IntervalFire(.5)
}
func is_art() { return 1 }
proc connect2target() { $o2 = new NetCon(pp, $o1) }
proc position(){x=$1  y=$2  z=$3}
endtemplate IF_IntervalFire

//Network specification interface

objref cells, nclist, netcon
{cells = new List()  nclist = new List()}

func cell_append() {cells.append($o1)  $o1.position($2,$3,$4)
   return cells.count - 1
}

func nc_append() {//srcindex, tarcelindex, synindex
  if ($3 >= 0) {
    cells.object($1).connect2target(cells.object($2).synlist.object($3), \
                                    netcon)
    netcon.weight = $4   netcon.delay = $5
  }else{
    cells.object($1).connect2target(cells.object($2).pp,netcon)
    netcon.weight = $4   netcon.delay = $5
  }
  nclist.append(netcon)
  return nclist.count - 1
}

//Network instantiation

  /* IF0 */  cell_append(new IF_IntervalFire(),    -149,   73, 0)
  /* IF1 */  cell_append(new IF_IntervalFire(),    -67,    73, 0)
  /* IF1 -> IF0    */  nc_append(1, 0, -1,  -0.1,1)
  /* IF0 -> IF1    */  nc_append(0, 1, -1,  -0.1,1)
```

Listing 11.2. Clicking on the Hoc File button in the NetWork Builder produces a file which we have called prototype.hoc.

A quick glance over the entire listing reveals that `prototype.hoc` is organized into several parts, which are introduced by one or more lines of descriptive comments. Let us consider each of these in turn, to see how it works and think about what we might reuse to make a network of any size we like.

NetGUI default section

The first part of the file creates `acell_home_` and make this the default section. What is a *section* doing in a model that contains artificial spiking cells? Remember that artificial spiking cells are basically point processes (see **Artificial spiking cells** in **Chapter 10**), and just like other point processes, they must be attached to a section. Suddenly the meaning of the comment `Artificial cells, if any, are located here` becomes clear: `acell_home_` is merely a "host" for artificial spiking cells. It has no biophysical mechanisms of its own, so it introduces negligible computational overhead.

Network cell templates

The NetWork Builder and its associated tools make extensive use of object-oriented programming. Each cell in the network is an instance of a cell class, and this is where the templates that declare these classes are located (templates and other aspects of object-oriented programming in NEURON are discussed in **Chapter 13**).

The comments that precede the templates contain a list of the cell class names. Our toy network uses only one cell class, so `prototype.hoc` contains only one template, which defines the `IF_IntervalFire` class. When biophysical neuron models are present, they are declared first. Thus, if we had a NetWork Builder whose palette contained a biophysical neuron model type called pyr, and an artificial spiking cell type S that was derived from the `NetStim` class, the corresponding cell classes would be called `pyr_Cell` and `S_NetStim`, and the header in the exported hoc file would read

```
//Network cell templates
//   pyr_Cell
//Artificial cells
//   S_NetStim
```

Functions and procedures with the same names as those contained in the `IF_IntervalFire` template will be found in every cell class used by a NetWork Builder (although some of their internal details may differ). The first of these is `init()`, which is executed automatically whenever a new instance of the `IF_IntervalFire` class is created. This in turn creates a new instance of the `IntervalFire` class that will be associated with the `acell_home_` section. As an aside, we should mention that this is an example of how the functionality of a basic object class can be enhanced by wrapping it inside a template in order to define a new class with additional features, i.e. an example of emulating inheritance in hoc (see **Polymorphism and inheritance** in **Chapter 13**).

The remaining `func`s and `proc`s are public so they can be called from outside the template. If we ever need to determine which elements in a network are artificial spiking cells and which are biophysical neuron models, `is_art()` is clearly the way to do it. The next is `connect2target()`, which looks useful for setting up network connections,

but it turns out that the hoc code we write ourselves won't call this directly (see **Network specification interface below**). The last is `position()` which can be used to specify unique xyz coordinates for each instance of this cell. The coordinates themselves are public (accessible from outside the template--see **Chapter 13** for more about accessing variables, `funcs` and `procs` declared in a template). Position may seem an arcane attribute for an artificial spiking neuron, but it is helpful for algorithmically creating networks in which connectivity or synaptic weight are functions of location or distance between cells.

Network specification interface

These are the variables and functions that we will actually call from our own hoc code. These are intended to offer us a uniform, compact and convenient syntax for setting up our own network. That is, they serve as a "programming interface" between the code we write and the lower level code that accomplishes our ultimate aims.

The purpose of the first two lines in this part of `prototype.hoc` is evident if we keep in mind that the NetWork Builder implements a network model with objects, some of which represent cells while others represent the connections between them. The `List` class is the programmer's workhorse for managing collections of objects, so it is reasonable that the cells and connections of our network model will be packaged into two `Lists` called `cells` and `nclist`, respectively.

The functions that add new elements to these `Lists` are `cell_append()` and `nc_append()`, respectively. The first argument to `cell_append()` is an `objref` that points to a new cell that is to be added to the list, and the remaining arguments are the xyz coordinates that are to be assigned to that cell. The `nc_append()` function uses an `if . . . else` to deal properly with either biophysical neuron models or artificial spiking cells. In either case, its first two arguments are integers that indicate which elements in `cells` are the `objrefs` that correspond to the pre- and postsynaptic cells, and the last two arguments are the synaptic weight and delay. If the postsynaptic cell is a biophysical neuron model, one or more synaptic mechanisms will be attached to it (see the tutorial at `http://www.neuron.yale.edu/neuron/docs/netbuild/main.html`). In this case, the third argument to `nc_append()` is a nonnegative integer that specifies which synaptic mechanism is to be the target of the new `NetCon`. If instead the postsynaptic cell is an artificial spiking cell, the argument is just -1.

Network instantiation

So far everything has been quite generic, in the sense that we can use it to create cells and assemble them into whatever network architecture we desire. In other words, the code up to this point is exactly the reusable code that we needed. The statements in the "network instantiation" group are just a concrete demonstation of how to use it to spawn a particular number of cells and link them with a specific network of connections. Let's make a copy of `prototype.hoc`, call it `netdefs.hoc`, and then insert `//` at the beginning of each of last four lines of `netdefs.hoc` so they persist as a reminder of how to call `cell_append()` and `nc_append()` but won't be executed. We are now ready to use `netdefs.hoc` to help us build our own networks.

Exploiting the reusable code

Where should we begin? A good way to start is by imagining the overall organization of the entire program at the "big picture" level. We'll need the GUI library, the class definitions and other code in `netdefs.hoc`, code to specify the network model itself, and code that sets up controls for adjusting model parameters, running simulations, and displaying simulation results. Following our recommended practices of modular programming and separating model specification from user interface (see **Elementary project management** in **Chapter 6**), we turn this informal outline into an `init.hoc` file that pulls all these pieces together (Listing 11.3).

```
load_file("nrngui.hoc")
load_file("netdefs.hoc")  // code from NetWork Builder-generated hoc file
load_file("makenet.hoc")  // specifies network
load_file("rig.hoc")  // for adjusting model params and running simulations
```

Listing 11.3. The `init.hoc` for our own network program.

For now, we can comment out the last two lines with `//` so we can test `netdefs.hoc` by using NEURON to execute `init.hoc`. and then typing a few commands at the `oc>` prompt (user entries are **`Courier bold`** while the interpreter's output is plain `Courier`).

```
Additional mechanisms from files
 invlfire.mod
   1
   1
oc>objref foo
oc>foo = new IF_IntervalFire()
oc>foo
   IF_IntervalFire[0]
oc>
```

So far so good. We are ready to apply the strategy of iterative program development (see **Iterative program development** in **Chapter 6**) to fill in the details.

The first detail is how to create a network of a specific size. If we call the number of cells `ncell`, then this loop

```
for i=0, ncell-1 {
  cell_append(new IF_IntervalFire(), i, 0, 0)
}
```

will make them for us, and this nested loop

```
for i=0, ncell-1 for j=0, ncell-1 if (i != j) {
  nc_append(i, j, -1, 0, 1)
}
```

will attach them to each other. An initial stab at embedding both of these in a procedure which takes a single argument that specifies the size of the net is


```
proc createnet() { local i, j
  ncell = $1
  for i=0, $1-1 {
    cell_append(new IF_IntervalFire(), i, 0, 0)
  }
  for i=0, $1-1 for j=0, $1-1 if (i != j) {
    nc_append(i, j, -1, 0, 1)
  }
}
```

and that's what we put in the first version of `makenet.hoc`.

We can test this by uncommenting the `load_file("makenet.hoc")` line in `init.hoc`, using NEURON to execute `init.hoc`, and then typing a few commands at the `oc>` prompt.

```
oc>createnet(2)
oc>ncell
   2
oc>print cells, nclist
   List[8] List[9]
oc>print cells.count, nclist.count
   2 2
oc>for i=0,1 print cells.object(i), nclist.object(i)
IF_IntervalFire[0] NetCon[0]
IF_IntervalFire[1] NetCon[1]
oc>
```

So it works. But almost immediately a wish list of improvements comes to mind. In order to try networks of different sizes, we'll be calling `createnet()` more than once during a single session. As it stands, repeated calls to `createnet()` just tack more and more new cells and connections onto the ends of the `cells` and `nclist` lists. Also, `createnet()` should be protected from nonsense arguments (a network should have at least two cells).

We can add these fixes by changing `ncell = $1` to

```
if ($1<2) { $1 = 2 }
ncell = $1
nclist.remove_all()
cells.remove_all()
```

The first line ensures our net will have two or more cells. The last two lines use the `List` class's `remove_all()` to purge `cells` and `nclist`. Of course we check this

```
oc>createnet(1)
oc>ncell
   2
oc>createnet(2)
oc>ncell
   2
oc>createnet(3)
oc>ncell
   3
oc>
```

which is exactly what should happen.

What else should go into `makenet.hoc`? How about procedures that make it easy to change the properties of the cells and connections? As a case in point, this

```
proc delay() { local i
  del = $1
  for i=0, nclist.count-1 {
    nclist.object(i).delay = $1
  }
}
```

lets us set all synaptic delays to the same value by calling `delay()` with an appropriate argument. Similar `proc`s can take care of weights and cellular time constants. Setting ISIs seems more complicated at first, but after a few false starts we come up with

```
proc interval() { local i, x, dx
  low = $1
  high = $2
  x = low
  dx = (high - low)/(cells.count-1)
  for i=0, cells.count-1 {
    cells.object(i).pp.invl = x
    x += dx
  }
}
```

This assigns the `low` ISI to the first cell in `cells`, the `high` ISI to the last cell in `cells`, and evenly spaced intermediate values to the other cells.

Does that mean the first cell is the fastest spiker, and the last is the slowest? Only if we are careful about the argument sequence when we call `interval()`. For that matter, what prevents us from calling `interval()` with one or both arguments < 0? Come to think of it, some of our other `proc`s might also benefit by being protected from nonsense arguments. We might protect against negative delays by changing

```
del = $1
```

in `proc delay()` to

```
if ($1<0) $1=0
del = $1
```

and we could insert similar argument-trapping code into other `proc`s as necessary.

However, it makes more sense to try to identify a common task that can be split out into a separate function that can be called by any `proc` that needs it. It may help to tabulate the vulnerable variables and the constraints we want to enforce.

Variable	**Constraint**
`ncell`	≥ 2
`tau`	> 0
`low` ISI	> 0
`high` ISI	≥ `low` ISI
`del`	≥ 0

Most of these constraints are "greater than or equal to," the two holdouts being `tau` and `low` ISI. After a moment we realize that there are practical lower limits to these

variables--say 0.1 ms for `tau` and 1 ms for `low` ISI--so "greater than or equal to" constraints can be applied to all.

The final version of `makenet.hoc` (Listing 11.4) contains all of these refinements. The statements at the very end create a network by calling our revised `proc`s.

```
/*
returns value >= $2
for bulletproofing procs against nonsense arguments
*/

func ge() {
  if ($1<$2) {
    $1=$2
  }
  return $1
}

////////// create a network //////////

// argument is desired number of cells

proc createnet() { local i, j
  $1 = ge($1,2) // force net to have at least two cells
  ncell = $1
  // so we can make a new net without having to exit and restart
  nclist.remove_all()
  cells.remove_all()
  for i=0, $1-1 {
    cell_append(new IF_IntervalFire(), i, 0, 0)
  }
  for i=0, $1-1 for j=0, $1-1 if (i != j) {
    // let weight be 0; we'll give it a nonzero value elsewhere
    nc_append(i, j, -1, 0, 1)
  }
  objref netcon  // leave no loose ends (see nc_append())
}

////////// specify parameters //////////

// call this settau() to avoid conflict with scalar tau

proc settau() { local i
  $1 = ge($1,0.1)  // min tau is 0.1 ms
  tau = $1
  for i=0, cells.count-1 {
    cells.object(i).pp.tau = $1
  }
}
```



```
// args are low and high

proc interval() { local i, x, dx
  $1 = ge($1,1)  // min low ISI is 1 ms
  $2 = ge($2,$1)
  low = $1
  high = $2
  x = low
  dx = (high - low)/(cells.count-1)
  for i=0, cells.count-1 {
    cells.object(i).pp.invl = x
    x += dx
  }
}

proc weight() { local i
  w = $1
  for i=0, nclist.count-1 {
    nclist.object(i).weight = $1
  }
}

proc delay() { local i
  $1 = ge($1,0)  // min del is 0 ms
  del = $1
  for i=0, nclist.count-1 {
    nclist.object(i).delay = $1
  }
}

////////// actually make net and set parameters //////////

createnet(2)
settau(10)
interval(10, 11)
weight(0)
delay(1)
```

Listing 11.4. Final implementation of `makenet.hoc`.

Time for more tests!

```
oc>del
0
oc>{delay(-1) print del}
0
oc>{delay(3) print del}
3
oc>createnet(4)
oc>ncell
   4
oc>del
3
oc>
```

Of course we can and should test the other `proc`s, especially `interval()`. As certain mathematics texts say, "this is left as an exercise to the reader."

Our attention now shifts to creating the user interface for adjusting model parameters, controlling simulations, and displaying results. To evoke the metaphor of an experimental rig, this is placed in a file called `rig.hoc`.

An initial implementation of `rig.hoc` might look like this

```
load_file("runctl.ses")  // RunControl and VariableTimeStep

xpanel("Model parameters")
xvalue("Weight","w", 1,"weight(w)", 0, 0 )
xvalue("Delay (ms)","del", 1,"delay(del)", 0, 0 )
xvalue("Cell time constant (ms)","tau", 1,"settau(tau)", 0, 0 )
xvalue("Shortest natural ISI","low", 1,"interval(low, high)", 0, 0 )
xvalue("Longest natural ISI","high", 1,"interval(low, high)", 0, 0 )
xpanel(500,400)
```

In the spirit of taking advantage of every shortcut the GUI offers, the first statement loads a session file that recreates a RunControl and a VariableTimeStep panel configured for the desired simulation duration (Tstop = 500 ms) and integration method (adaptive integration with local time steps). The other statements set up a panel with numeric fields and controls for displaying and adjusting model parameters. This implementation of `rig.hoc` lacks two important features: a graph that displays spike trains, and the ability to change the number of cells in the network.

What about plots of spike trains? There is a way to create a graph that provides all the functionality of the NetWork Builder's own SpikePlot, but analyzing the necessary code would lead us into details that really belong in a chapter on advanced GUI programming. For didactic purposes it is better if we make our own raster plot, if only because this will draw our attention to topics that are likely to be more widely useful.

To prepare to record and plot spike trains, we can insert the following code right after the `load_file()` statement:

```
objref netcon, vec, spikes, nil, graster

proc preprasterplot() {
  spikes = new List()
  for i=0,cells.count()-1 {
    vec = new Vector()
    netcon = new NetCon(cells.object(i).pp, nil)
    netcon.record(vec)
    spikes.append(vec)
  }
  objref netcon, vec

  graster = new Graph(0)
  graster.view(0, 0, tstop, cells.count(), 300, 105, 300.48, 200.32)
}

preprasterplot()
```

For each cell in the net, this creates a new `Vector`, uses the `NetCon` class's `record()` method to record the time of that cell's spikes into the `Vector`, and appends the `Vector` to a `List`. After the end of the `for` loop that iterates over the cells, the `netcon` and `vec` `objref`s point to the last `NetCon` and `Vector` that were created, exposing them to

possible interference if we ever do anything that reuses these `objref` names. The `objref netcon, vec` statement breaks the link between them and the objects, thereby preventing such undesirable effects.

The last two statements in `preprasterplot()` create a `Graph` and place it at a desired location on the screen. How can we tell what the numeric values should be for the arguments in the `graster.view()` statement? By creating a graph (NEURON Main Menu / Graph / Voltage axis will do), dragging it to the desired location, saving it to a session file all by itself, and then stealing the argument list from that session file's `save_window_.view()` statement--being careful to change the third and fourth arguments so that the x and y axes span the correct range of values. No cut and try guesswork for us! While we're at it, we might as well use the same strategy to fix the location for our model parameter panel, but now we only need the fifth and sixth arguments to `view()`, which are the screen coordinates where the `Graph` is positioned. For my monitor, this means the second `xpanel` statement becomes `xpanel(300,370)`.

Running a new test, we find that our user interface looks like Fig. 11.22. Everything is in the right place, and time advances when we click on Init & Run, but no rasters are plotted.

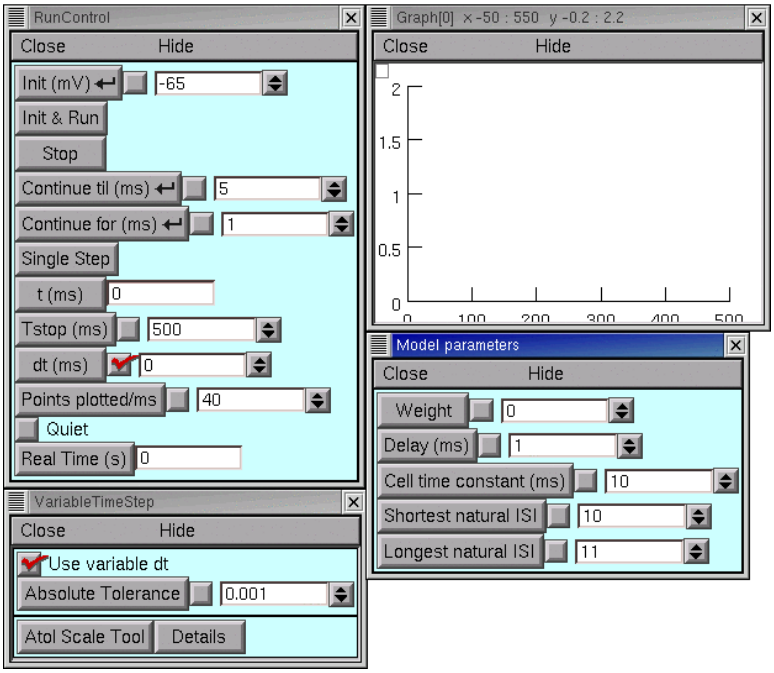


Figure 11.22. The user interface after the first revision to `rig.hoc`, in which we added `preprasterplot()`.

For each cell we need to draw a sequence of short vertical lines on `graster` whose x coordinates are the times at which that cell fired. To help us tell one cell's spikes from another's, the vertical placement of their rasters should correspond to their ordinal position in `cells`. We can do this by inserting the following code into `rig.hoc`, right after the call to `preprasterplot()`. The first thing that `proc showraster()` does is

to clear any previous rasters off the `Graph`. Then, for each cell in turn, it uses three `Vector` class methods in succession: `c()` to create a `Vector` that has as many elements as the number of spikes that the cell fired, `fill()` to fill those elements with an integer that is one more than the ordinal position of that cell in `cells`, and `mark()` to mark the firing times.

```
objref spikey

proc showraster() {
  graster.erase_all()
  for i = 0,cells.count()-1 {
    spikey = spikes.object(i).c
    spikey.fill(i+1)
    spikey.mark(graster, spikes.object(i), "|", 6)
  }
  objref spikey
}
```

Testing once again, we run a simulation and then type `showraster()` at the `oc>` prompt, and sure enough, there are the spikes. We change the longest natural ISI to 20 ms, run another simulation, and type `showraster()` once more, and it works again.

All this typing is tedious. Why not customize the `run()` procedure so that it automatically calls `showraster()` after each simulation? Adding this

```
proc run() {
  stdinit()
  continuerun(tstop)
  showraster()
}
```

to the end of `rig.hoc` does the job (see **An outline of the standard run system** in **Chapter 7: How to control simulations**).

Another test and we are overcome with satisfaction--it works. Then we change Tstop to 200 ms, run a simulation, and are disappointed that the raster plot's x axis does not rescale to match the new Tstop. One simple fix for this is to write a custom `init()` procedure that uses the Graph class's `size()` method to adjust the size of the raster plot during initialization (see **Default initialization in the standard run system: stdinit() and init()** in **Chapter 8**). So we insert this

```
proc init() {
  finitialize(v_init)
  graster.erase_all()
  graster.size(0,tstop,0,cells.count())
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

right after our custom `run()`. Notice that this also rescales the y axis, which will be helpful when we finally add the ability to change the number of cells in the network.

Success upon success! It works!

We can finally get around to changing the number of cells. Let's think this out carefully before doing anything. We'll need a new control in the `xpanel`, to show how many cells there are and let us specify a new number. That's easy--just put this line

```
xvalue("Number of cells","ncell", 1,"recreate(ncell)", 0, 0 )
```

right after `xpanel("Model parameters")` so that when we change the value of `ncell`, we automatically call a new procedure called `recreate()` that will throw away the old cells and their connections, and create a new set of each.

But what goes in `recreate()`? We'll want the new cells and connections to have the same properties as the old ones. And we'll have to replace the old raster plot with a new one, complete with all the `NetCons` and `Vectors` that it uses to record spikes. So `recreate()` should be

```
proc recreate() {
  createnet($1)
  settau(tau)
  interval(low, high)
  weight(w)
  delay(del)
  preprasterplot()
}
```

A good place for this is right before the `xpanel`'s code.

So now we have completed `rig.hoc` (see Listing 11.5). The parameter panel has all the right buttons (Fig. 11.23) so it is easy to explore the effects of parameter changes (Fig. 11.24). How to develop an understanding of what accounts for these effects is beyond the scope of this chapter, but we can offer one hint: run some simulations of a net containing only 2 or 3 cells, using fixed time steps, and plot their membrane state variables (well, their `M` functions).

```
////////// user interface //////////

load_file("runctl.ses")  // RunControl and VariableTimeStep

// prepare to record and display spike trains
objref netcon, vec, spikes, nil, graster

proc preprasterplot() {
  spikes = new List()
  for i=0,cells.count()-1 {
    vec = new Vector()
    netcon = new NetCon(cells.object(i).pp, nil)
    netcon.record(vec)
    spikes.append(vec)
  }
  objref netcon, vec

  graster = new Graph(0)
  graster.view(0, 0, tstop, cells.count(), 300, 105, 300.48, 200.32)
}

preprasterplot()
```



```
objref spikey

proc showraster() {
  graster.erase_all()
  for i = 0,cells.count()-1 {
    spikey = spikes.object(i).c
    spikey.fill(i+1)
    spikey.mark(graster, spikes.object(i), "|", 6)
  }
  objref spikey
}

// destroys existing net and makes a new one
// also spawns a new spike train raster plot
// called only if we need a different number of cells

proc recreate() {
  createnet($1)
  settau(tau)
  interval(low, high)
  weight(w)
  delay(del)
  preprasterplot()
}

xpanel("Model parameters")
xvalue("Number of cells","ncell", 1,"recreate(ncell)", 0, 0 )
xvalue("Weight","w", 1,"weight(w)", 0, 0 )
xvalue("Delay (ms)","del", 1,"delay(del)", 0, 0 )
xvalue("Cell time constant (ms)","tau", 1,"settau(tau)", 0, 0 )
xvalue("Shortest natural ISI","low", 1,"interval(low, high)", 0, 0 )
xvalue("Longest natural ISI","high", 1,"interval(low, high)", 0, 0 )
xpanel(300,370)

////////// custom run() and init() //////////

proc run() {
  stdinit()
  continuerun(tstop)
  showraster()  // show results at the end of each simulation
}

proc init() {
  finitialize(v_init)
  graster.erase_all()
  graster.size(0,tstop,0,cells.count())  // rescale x and y axes
  if (cvode.active()) {
    cvode.re_init()
  } else {
    fcurrent()
  }
  frecord_init()
}
```

Listing 11.5. Complete implementation of `rig.hoc`.

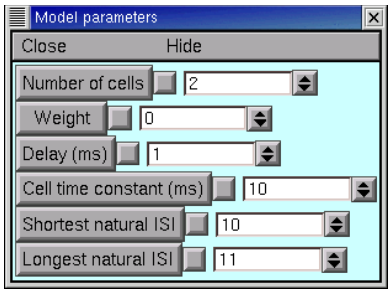
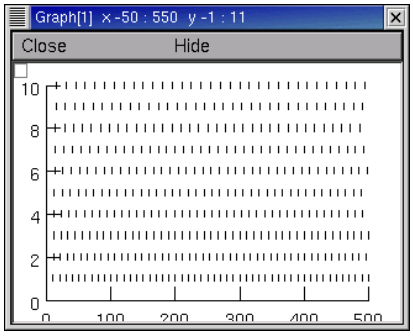


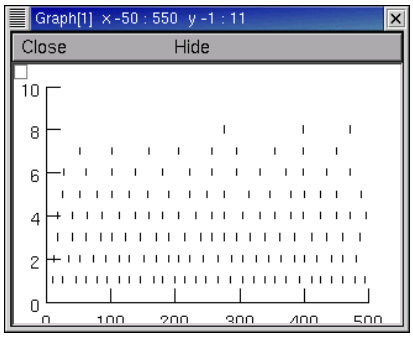
Figure 11.23. The parameter panel after addition of a control for changing the number of cells.

Figure 11.24. Simulations of a fully connected network with 10 cells whose natural ISIs are spaced uniformly over the range 10-15 ms. The rasters are arranged with ISIs in descending order from top to bottom.

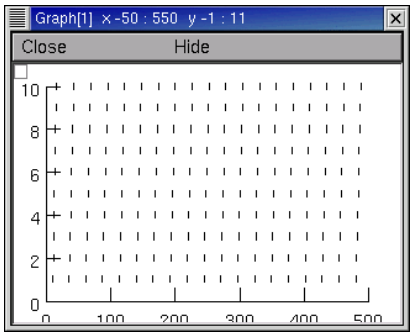
A: With all synaptic weights 0, cell firing is asynchronous and uncorrelated.



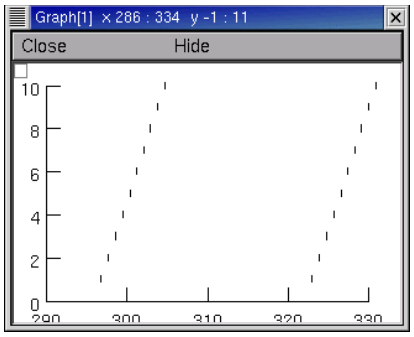
B: Mild inhibitory coupling (weight -0.2) with a delay of 1 ms silences the slowest cells and reduces the firing rates of the others. There is a suggestion of spike clustering, but no obvious synchrony or strong correlation.



C: Increasing synaptic delay to 8 ms allows the slowest cells to escape from inhibition and results in strong correlation.



D: Close examination reveals that spikes are not synchronous, but lag progressively across the population with increasing natural ISI.



References

Destexhe, A., Mainen, Z.F., and Sejnowski, T.J. An efficient method for computing synaptic conductances based on a kinetic model of receptor binding. *Neural Computation* 6:14-18, 1994.

Destexhe, A., McCormick, D.A., and Sejnowski, T.J. A model for 8-10 Hz spindling in interconnected thalamic relay and reticularis neurons. *Biophys. J.* 65:2474-2478, 1993.

Hines, M. A program for simulation of nerve equations with branching geometries. *Int. J. Bio-Med. Comput.* 24:55-68, 1989.

Hines, M. NEURON--a program for simulation of nerve equations. In: *Neural Systems: Analysis and Modeling*, edited by F. Eeckman. Norwell, MA: Kluwer, 1993, p. 127-136.

Hines, M. and Carnevale, N.T. Computer modeling methods for neurons. In: *The Handbook of Brain Theory and Neural Networks*, edited by M.A. Arbib. Cambridge, MA: MIT Press, 1995, p. 226-230.

Hines, M.L. and Carnevale, N.T. Expanding NEURON's repertoire of mechanisms with NMODL. *Neural Computation* 12:995-1007, 2000.

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

Lytton, W.W., Contreras, D., Destexhe, A., and Steriade, M. Dynamic interactions determine partial thalamic quiescence in a computer network model of spike-and-wave seizures. *J. Neurophysiol.* 77:1679-1696, 1997.

Maass, W. and Bishop, C.M., eds. *Pulsed Neural Networks*. Cambridge, MA: MIT Press, 1999.

Riecke, F., Warland, D., de Ruyter van Steveninck, R., and Bialek, W. *Spikes: Exploring the Neural Code*. Cambridge, MA: MIT Press, 1997.

Sohal, V.S., Huntsman, M.M., and Huguenard, J.R. Reciprocal inhibitory connections regulate the spatiotemporal properties of intrathalamic oscillations. *J. Neurosci.* 20:1735-1745, 2000.

Web site retrieved 11/8/2004. NEURON Tutorial by Andrew Gillies and David Sterratt. http://www.anc.ed.ac.uk/school/neuron/

Chapter 11 Index

Survey

We'd appreciate your frank opinions and suggestions to help us refine this course and design future offerings on related subjects.

Please score these items	· · · · ·	**according to this scale**	
Overall impression	_____	no opinion	0
Relevance to my research	_____	poor, not helpful	1
Didactic presentations	_____	fair	2
Written handouts	_____	good	3
Overhead transparencies	_____	excellent, very helpful	4
Computer projection	_____		
Classroom	_____		
Food	_____		

Best feature _____

Weakest feature _____

Additional topics that should be covered, topics that should receive more or less coverage, or other suggestions for improvement.

_____

_____

_____

Circle one

Y N I would recommend this course to others who are interested in neural modeling.

Y N I have developed my own modeling software using a high-level language (FORTRAN, C/C++ etc.).

Y N I have created my own models using modeling software.

Which software? _____

My area of primary research interest is _____

To help us better meet the needs of NEURON users, please circle all platforms that you plan to use for modeling.

Hardware Mac Intel Other _____

OS MacOS X Win NT | 2K | XP

UNIX | Linux | OS X | BSD

If Linux, which distribution? _____