

## 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



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

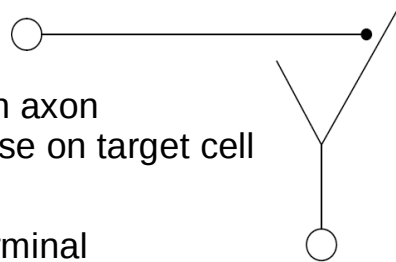
Physical system:

Presynaptic neuron with axon  
that projects to synapse on target cell

Conceptual model:

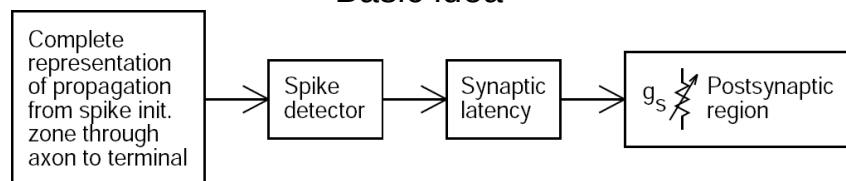
Spike in presynaptic terminal  
triggers transmitter release;  
presynaptic details unimportant

Postsynaptic effect described by  
DE or kinetic scheme that is perturbed by  
occurrence of a presynaptic spike

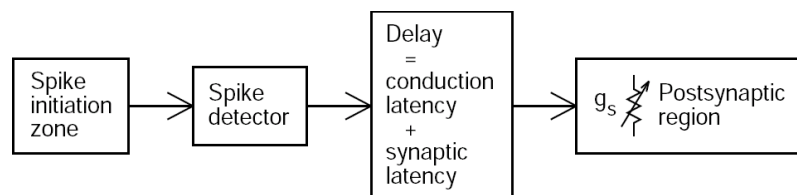


## Spike-triggered transmission: computational implementation

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

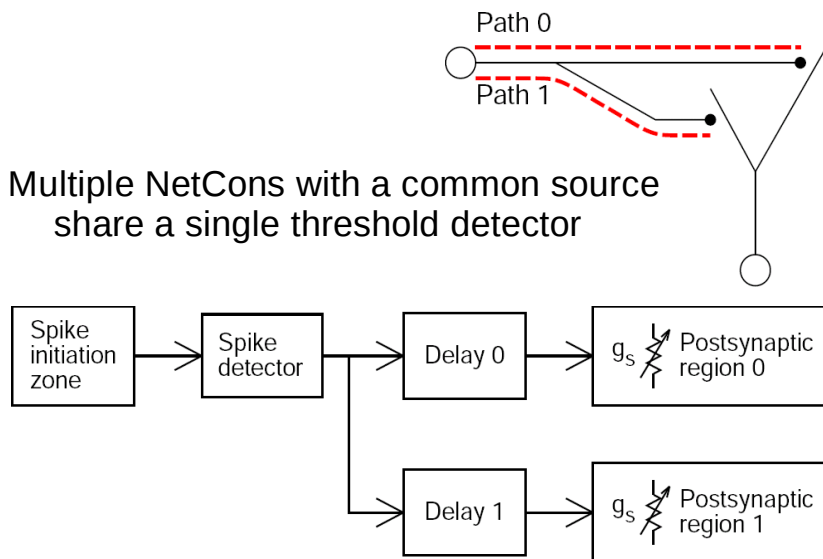
```
threshold = 10
delay = 1 // must be >= 0
weight = 0
```

NMODL specification of synaptic mechanism

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

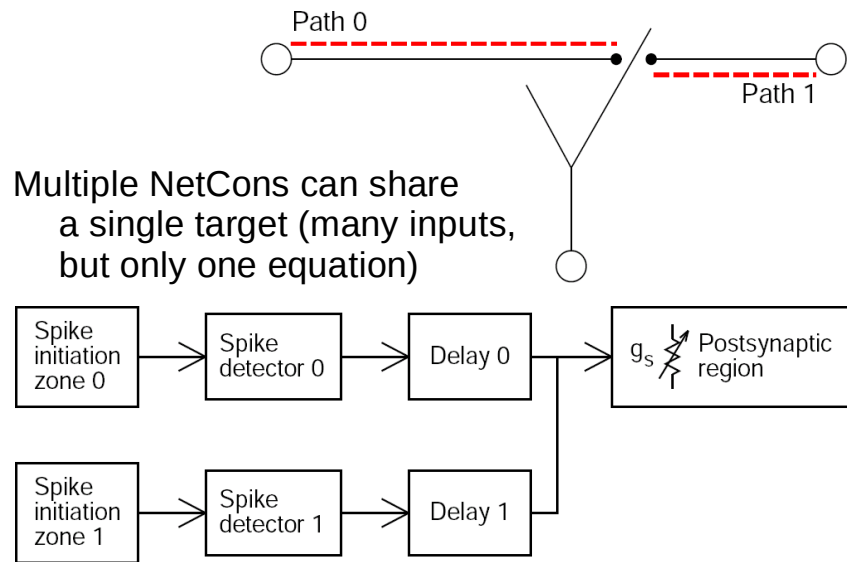
## Efficient divergence

Multiple NetCons with a common source  
share a single threshold detector





## Efficient convergence



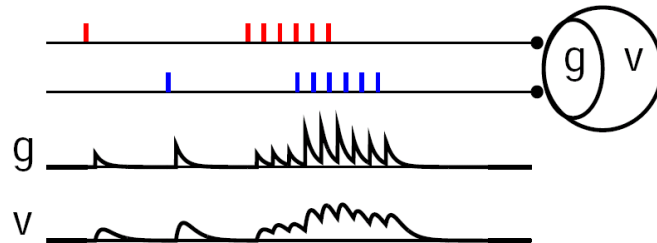
## Example: $g_s$ with fast rise and exponential decay

```

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

```

## $g_s$ with fast rise and exponential decay *continued*

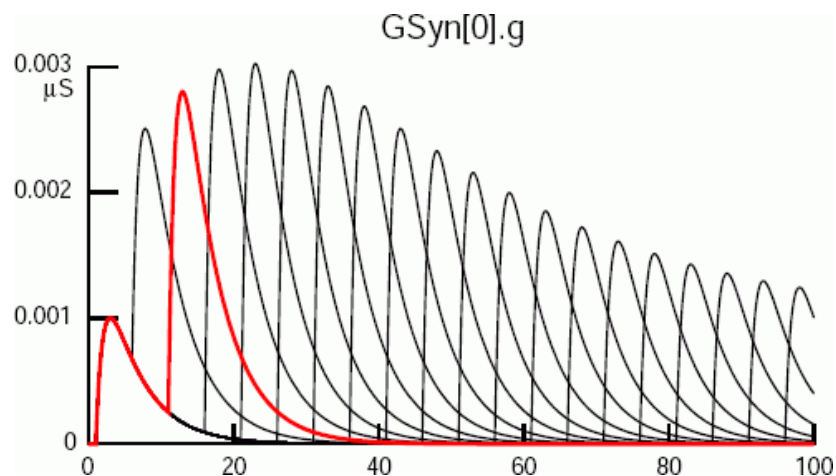


```

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

```

## Example: use-dependent synaptic plasticity

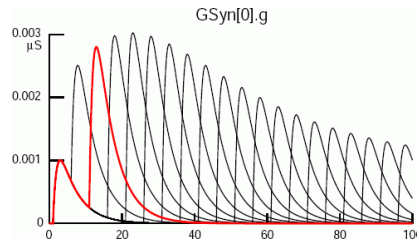


## Use-dependent synaptic plasticity *continued*

```

BREAKPOINT {
  SOLVE state METHOD cnexp
  g = B - A
  i = g*(v-e)
}
DERIVATIVE state {
  A' = -A/tau1
  B' = -B/tau2
}
NET_RECEIVE(weight (uS), w, G1, G2, t0 (ms)) {
  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

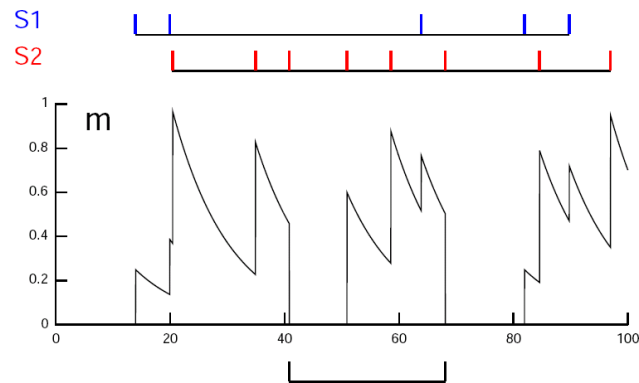
Event-driven simulation run time is

*proportional* to # of received events

*independent* of # of cells, # of connections,  
and problem time

Hybrid networks

## Example: leaky integrate and fire model

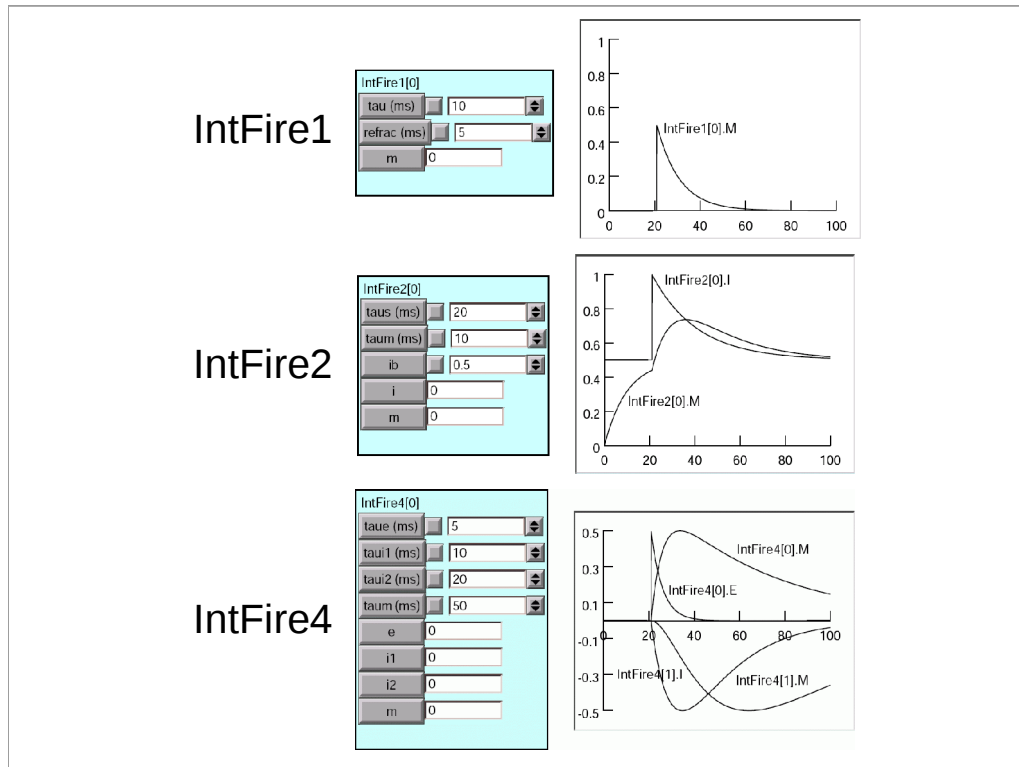


## Leaky integrate and fire model *continued*

```

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

ARTIFICIAL\_CELL 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\_PROCESSES  
 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())
```

## Connecting cells

Which setup strategy is more efficient?

Iterate over sources

```
for each cell {
  connect this cell to its targets
}
```

or iterate over targets?

```
for each cell {
  connect sources to this cell
}
```

## Connecting cells

For a net distributed over multiple CPUs,  
it is most efficient to iterate over targets first.

```
for each cell {  
    connect sources to this cell  
}
```





## ***Wiring networks in Neuron***

Bill Lytton

SUNY - Downstate  
Brooklyn, NY

Wiring networks in Neuron – p.1/4;

### ***TOC***

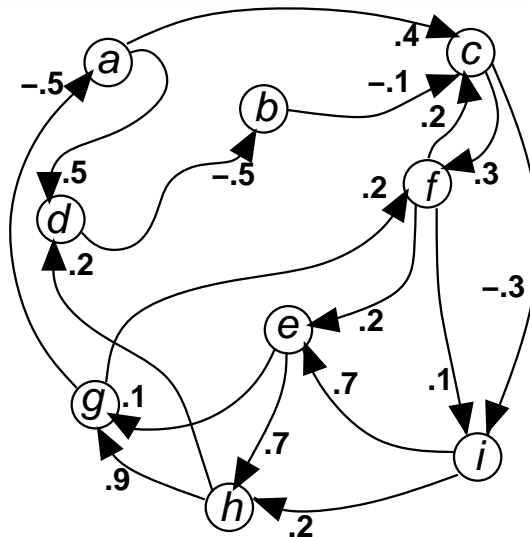
2. Simple network
3. Connections table
4. Connectivity matrix
5. Define connectivity
6. Hopfield-Brody synchronization model
7. Unconnected cells
8. Negative (inhibitory) connectivity
9. Q&D synchronization measure
10. Check sync with increasing inhib
11. Graph results
12. Confirm with raw data
13. Scale up to 100 cells
14. Need to normalize weights
15. NEURON's *list* object
16. Wiring the network
17. 100 x 100 matrix
18. Rewiring for different densities
19. Or rewrite weight()
20. Density doesn't make much difference!
21. Checking connectivity
22. `ccode.netconlist()`
23. In addition to `ccode.netconlist()`
24. `fconn()` – find connections
25. Now can check non-zero connectivity
26. Rewrite randomizer
27. Synchronization measure

Wiring networks in Neuron – p.2/4;

**TOC2**

28. Look at 10% connectivity    29. Show all connections    30. Show selected connections
31. Balancing convergence and divergence    32. Geographic connectivity
33. Random wiring with distance fall-off    34.  $p_{ij} = .5$ : convergence for 10 cells
35.  $p_{ij} = .1$ : lower density to be tested    36. Doesn't sync very well
37. Is there localized sync'ing?    38. How to animate    39. Other explorations
40. Advantages of NEURON for networks

Wiring networks in Neuron – p.3/4;

**Simple network**

Wiring networks in Neuron – p.4/4;

**Connections table**

FROM $\Rightarrow$	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>	<i>i</i>
TO $\Downarrow$ <i>a</i>	■						-0.5		
<i>b</i>		■		-0.5					
<i>c</i>	0.4	-0.1	■			0.2			
<i>d</i>	0.5			■				0.2	
<i>e</i>					■	0.2			0.7
<i>f</i>			0.3			■	0.2		
<i>g</i>					0.1		■	0.9	
<i>h</i>					0.7			■	0.2
<i>i</i>			-0.3			0.1			■

Wiring networks in Neuron – p.5/4;

**Connectivity matrix**

$$\begin{pmatrix}
 0 & 0 & 0 & 0 & 0 & 0 & -0.5 & 0 & 0 \\
 0 & 0 & 0 & -0.5 & 0 & 0 & 0 & 0 & 0 \\
 0.4 & -0.1 & 0 & 0 & 0 & 0.2 & 0 & 0 & 0 \\
 0.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0.2 & 0 & 0 & 0.7 \\
 0 & 0 & 0.3 & 0 & 0 & 0 & 0.2 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0.1 & 0 & 0 & 0.9 & 0 \\
 0 & 0 & 0 & 0 & 0.7 & 0 & 0 & 0 & 0.2 \\
 0 & 0 & -0.3 & 0 & 0 & 0.1 & 0 & 0 & 0
 \end{pmatrix}
 \begin{pmatrix}
 a \\
 b \\
 c \\
 d \\
 e \\
 f \\
 g \\
 h \\
 i
 \end{pmatrix}$$

Wiring networks in Neuron – p.6/4;

## Define connectivity

- ⑥ S=number of syns; D=divergence; C=convergence
- ⑥  $S = C \cdot Post; S = D \cdot Pre$
- ⑥ connectivity density  
 $p_{ij} = C/Pre = D/Post = S/(Pre \cdot Post)$
- ⑥ Below: 1 kind of cell  $\Rightarrow$  Pre and Post are the same

Wiring networks in Neuron – p.7/42

## Hopfield-Brody synchronization model

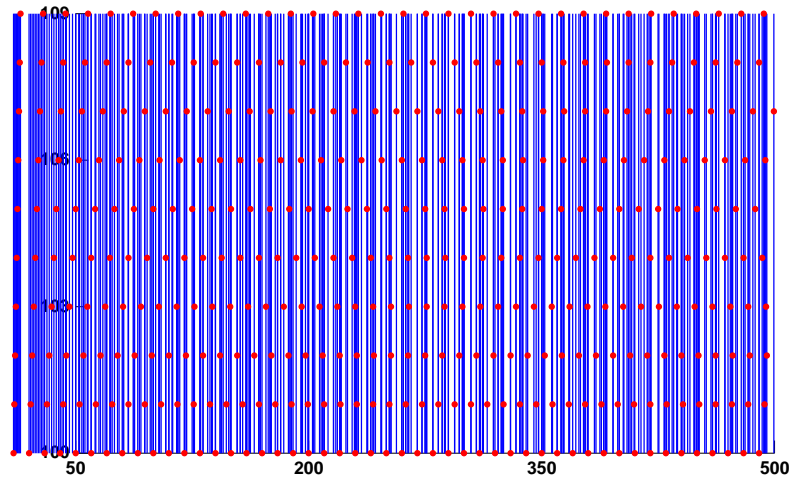
- ⑥ All to all connectivity
- ⑥ Firing cells synchronize due to mutual inhibition
- ⑥ Each cell has a natural period
- ⑥ Inhibition from other cells provides a reset, locking them together

Wiring networks in Neuron – p.8/42

## *Unconnected cells*

$wt = -1e-5$

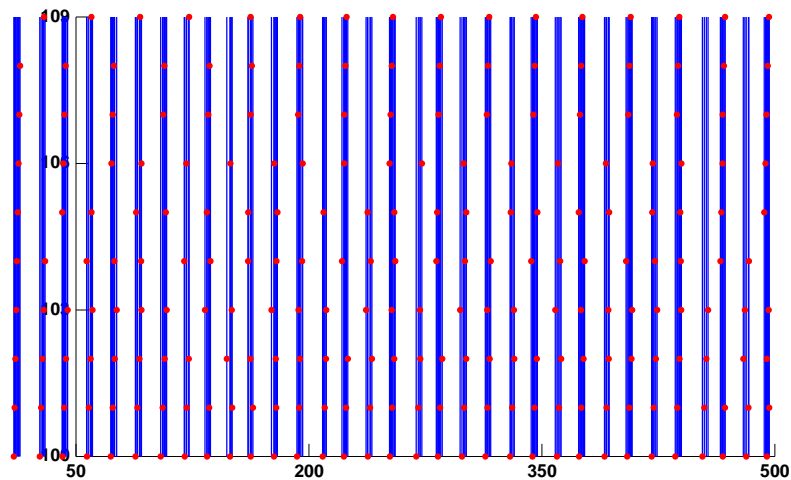
cell fire at own rates and go out of phase



Wiring networks in Neuron – p.9/4;

## *Negative (inhibitory) connectivity*

$wt = -3e-1$



Wiring networks in Neuron – p.10/4;

## Q&D synchronization measure

```
// syncer() :: returns sync measure 0 to <1
// measures how well spikes "fill up" the tir
// assumes spike times in tvec, tstop
// param: width
func syncer () { local t0,tt,cnt,width
  t0=-1 width=1 cnt=0
  for ii=0,tvec.size-1 {
    tt=tvec.x[ii]
    if (tt>=t0+width) {t0=tt cnt+=1}
  }
  return 1-cnt/(tstop/width)
}
```

Wiring networks in Neuron – p.11/4;

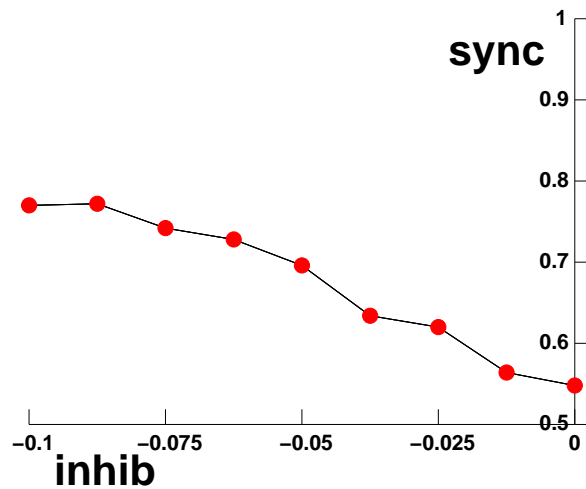
## Check sync with increasing inhib

```
// loop increasing neg. synaptic weight
// save measures in vec[1],vec[0]
max= -0.1
for (w=0;w>=max;w+=(max/8)) {
  w+=1e-6 // avoid using zero weight
  setparams() run()
  // g=new Graph() showspks()
  vec[1].append(w) vec.append(syncer())
}
```

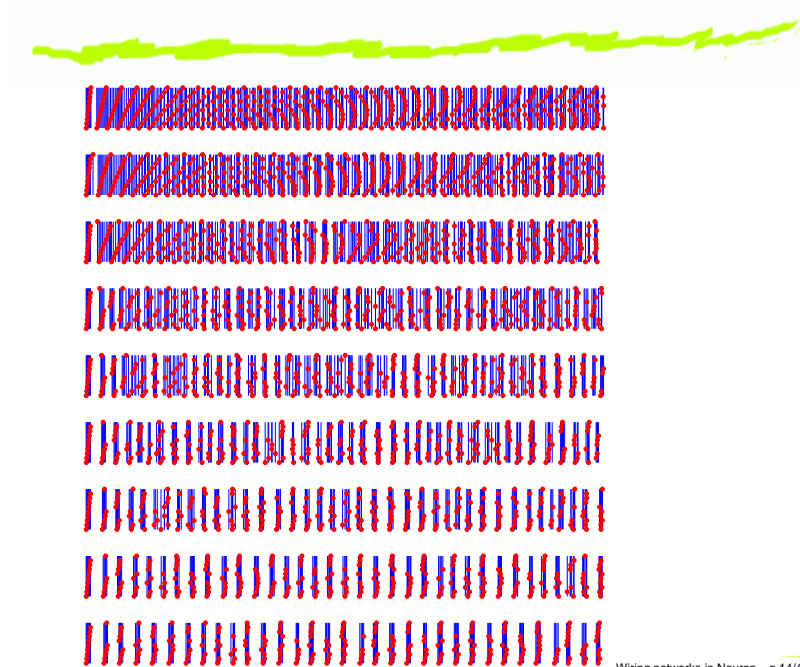
Wiring networks in Neuron – p.12/4;

**Graph results**

```
{vec.line(g,vec[1]) vec.mark(g,vec[1])}
```

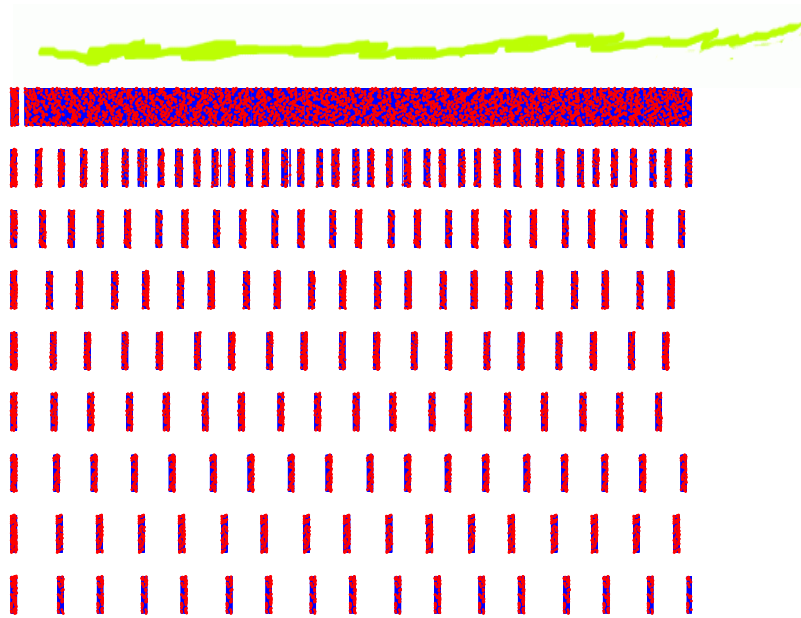


Wiring networks in Neuron – p.13/4;

**Confirm with raw data**

Wiring networks in Neuron – p.14/4;

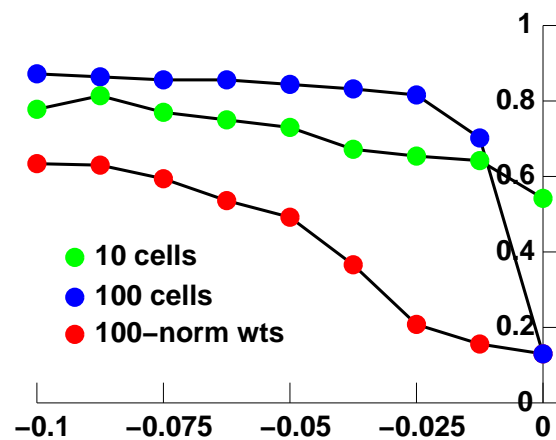
## Scale up to 100 cells



Wiring networks in Neuron – p.15/42

## Need to normalize weights

$$w = (ii + 1e-6) / (ncell / 10)$$



Wiring networks in Neuron – p.16/42



## NEURON's list object

- ⑥ An alternative to object array e.g., objref nc[9900]
- ⑥ Advantage: can change number of objects stored
- ⑥ `nclist = new List()`
- ⑥ add syn: `nclist.append(netcon)`
- ⑥ how many?: `nclist.count()`
- ⑥ retrieve syn #5: `netcon=nclist.object(5)`
- ⑥ clear: `nclist.remove_all`

Wiring networks in Neuron – p.17/42

## Wiring the network

```
// wire():: full non-self connectivity
// artificial cell template have obj.pp
// params: ncell
// creates nclist: list of NetCons
proc wire () {
  nclist.remove_all()
  for i=0,ncell-1 for j=0,ncell-1 if (i!=j)
    netcon = new NetCon(cells.object(i).pp,\
                        cells.object(j).pp)
    nclist.append(netcon)
  }
}
```

Wiring networks in Neuron – p.18/42

**100 x 100 matrix**

- ⑥ Index synapse from 0  $\rightarrow S = \text{ncells}^2 - \text{ncells}$
- ⑥ Either set  $p_{ij} \cdot S$  or delete (zero out)  $(1 - p_{ij}) \cdot S$  syns
- ⑥ e.g.,
 

```
rdm.discunif(0,S-1) // random indices
vec.resize((1-pij)*S)
vec.setrand(rdm)
```

Wiring networks in Neuron – p.19/4;

**Rewiring for different densities**

- ⑥ Can either rewire (if sparse) or just set weights to 0 (if not)
- ⑥ rewrite wire():
 

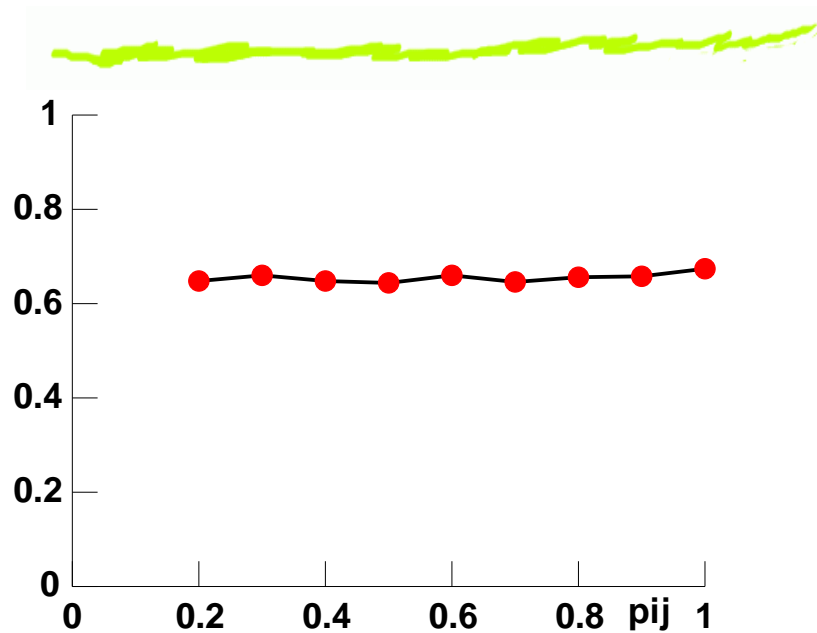
```
// don't create if syn_num is not on list
// note that num of nclist.object(num)
//   no longer meaningful
// here array better: 'objref nc[9900]'
for i=0,ncell-1 for j=0,ncell-1 {
  if (i!=j && !vec.contains(i*100+j)) {
    ... new NetCon ...
```

Wiring networks in Neuron – p.20/4;

**Or rewrite weight()**

```
// weight2(WT,EXCLUDE_VEC) :: set weight to 1
//      unless in EXCLUDE_VEC then set wt. to
proc weight2 () { local i,ww
  w = $1
  for i=0,nclist.count-1 {
    if ($o2.contains(i)) ww=0 else ww=w
    nclist.object(i).weight = ww
  }
}
```

Wiring networks in Neuron – p.21/4;

**Density doesn't make much difference!**

Wiring networks in Neuron – p.22/4;

## Checking connectivity

- ⑥ Surprising findings are often artifacts
- ⑥ Pseudo-random may be more pseudo than random
- ⑥ Best-written programs of mice & men
- ⑥ Check if network looks reasonable

Wiring networks in Neuron – p.23/4;

## ***cvode.netconlist()***

access NEURON's internal NetCon list

- ⑥ Unwieldy, but important to make sure that WYWIWYG
- ⑥ To check divergence:
 

```
for ii=0,ncell-1 print\
  cvode.netconlist(cells.object(ii).pp,"",""
  .count
```
- ⑥ To check convergence:
 

```
for ii=0,ncell-1 print\
  cvode.netconlist("","",cells.object(ii).p
  .count
```
- ⑥ Could count non-zero synapses by iterating through
 

```
cvode.netconlist("","","")
```

Wiring networks in Neuron – p.24/4;

## ***In addition to `cvscode.netconlist()`***

Develop a parallel database

- ⑥ I often use a sparse matrix for molding connectivity
- ⑥ In present case, we can work with `nclist`
- ⑥ Beware stray NetCons

Wiring networks in Neuron – p.25/42

## ***fconn() – find connections***

```
// fconn(PREVEC,POSTVEC) places values of
// pre- and post-syn cells in parallel vectors
// only lists pairs with non-zero connections
// getcnum() returns index of cell object
proc fconn () {
    $o1.resize(0) $o2.resize(0)
    for ii=0,nclist.count-1 {
        XO=nclist.object(ii)
        if (XO.weight!=0) {
            $o1.append(getcnum(XO.pre))
            $o2.append(getcnum(XO.syn))
        }
    }
}
```

Wiring networks in Neuron – p.26/42

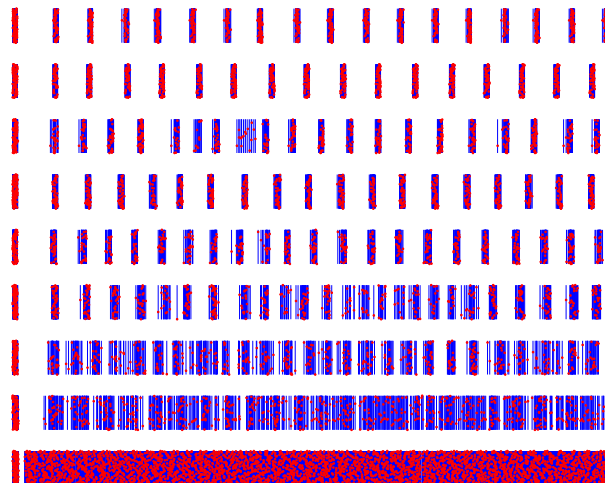
## Now can check non-zero connectivity

- ⑥  $p_{ij} = 0.2 \dots$
- ⑥ `fconn(vec[4],vec[5])`
- ⑥ `print vec[4].size`  
4479
- ⑥ Wrong answer!:  $p_{ij} \cdot S \sim 2000$
- ⑥ Zero-weighting vector had repeats

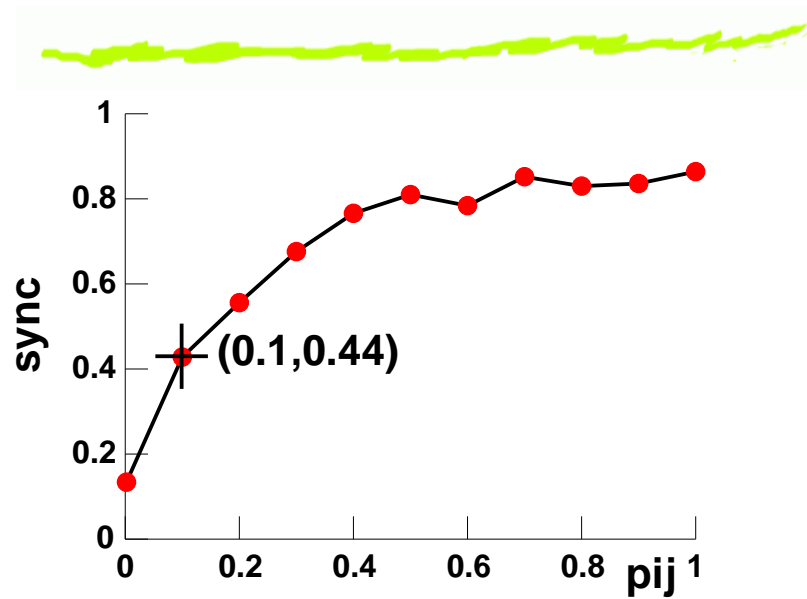
Wiring networks in Neuron – p.27/4;

## Rewrite randomizer

`rdmunq()` – augments vec by n unique vals from `rdm`  
scale weights to compensate for reduced convergence



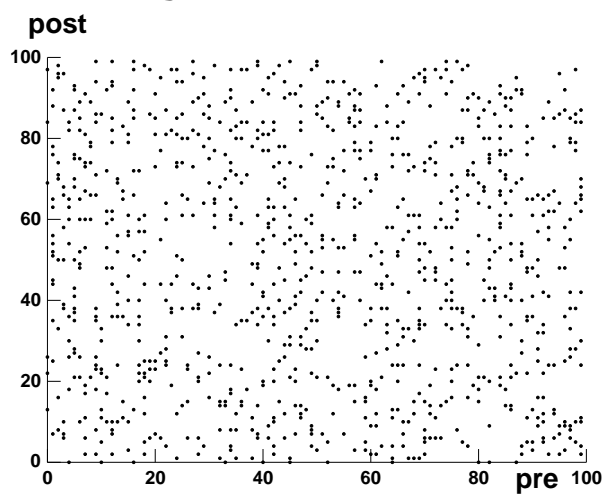
Wiring networks in Neuron – p.28/4;

**Synchronization measure**

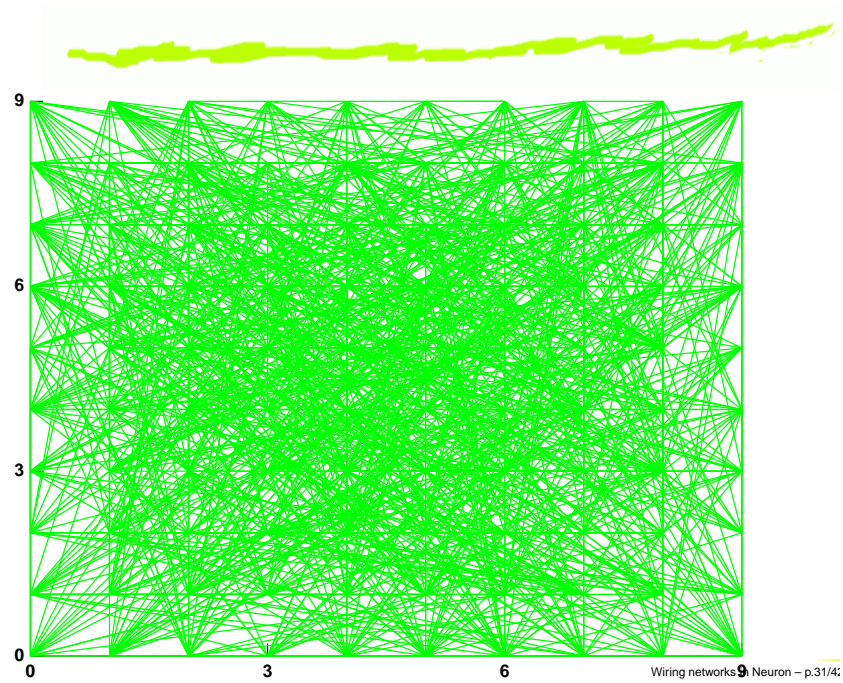
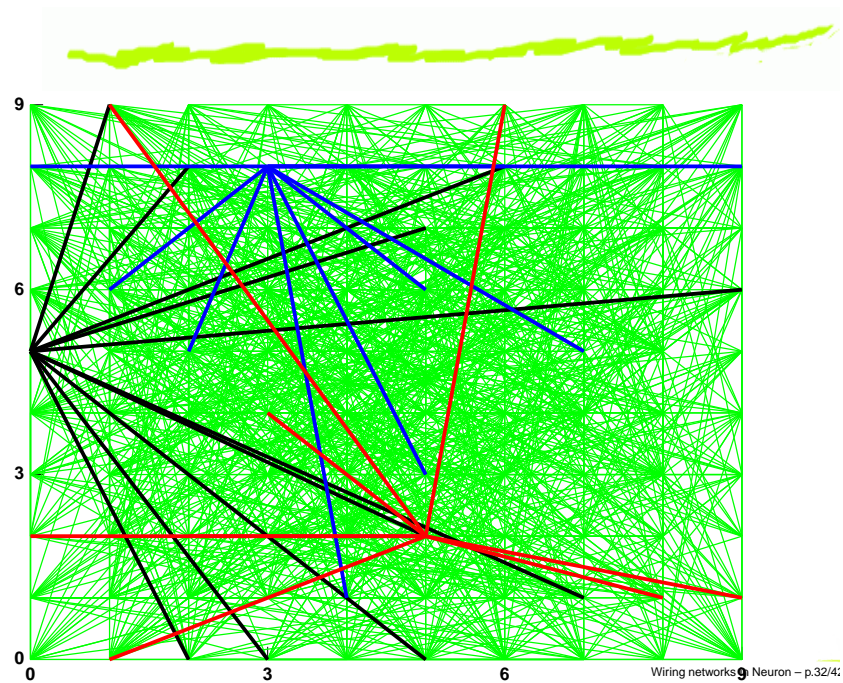
Wiring networks in Neuron – p.29/4;

**Look at 10% connectivity**

```
{PRE=4 POST=5 fconn(vec[PRE],vec[POST])}
vec[POST].mark(g,vec[PRE],"O",2,1,1)
```



Wiring networks in Neuron – p.30/4;

**Show all connections****Show selected connections**



## ***Balancing convergence and divergence***



- ⑥ Wide variation in connectivity:  $\langle C \rangle = 9.91 \pm 2.99$ ;  $\langle D \rangle = 9.91 \pm 3.09$
- ⑥  $C_{min}=3$ ;  $C_{max}=21$ ;  $D_{min}=4$ ;  $D_{max}=17$ ;
- ⑥ With realistic cells, must be careful to balance convergence or will blast some cells

Wiring networks in Neuron – p.33/4;

## ***Geographic connectivity***



- ⑥ Neuroanatomy furnishes non-random connectivity
- ⑥ Map onto model connectivity
- ⑥ e.g., fall-off with distance

Wiring networks in Neuron – p.34/4;

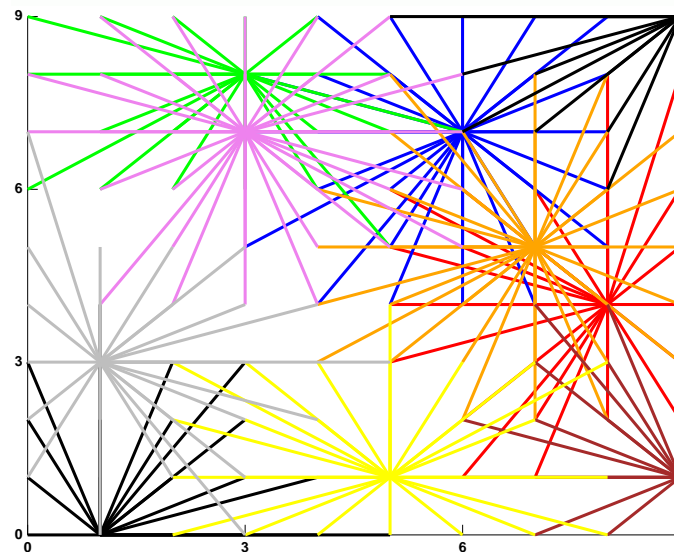
## Random wiring with distance fall-off

- ⑥ 1. Go through syns in random order
- ⑥ 2. Flip biased coin proportional to distance
 

```
rdm[1].uniform(0,1) // for flipping
coin
prob = 1-distn()/maxdist
if (rdm[1].repick<prob) { ...
```
- ⑥ 3. Count syns till reach desired density
- ⑥ Better if used a hexagonal array

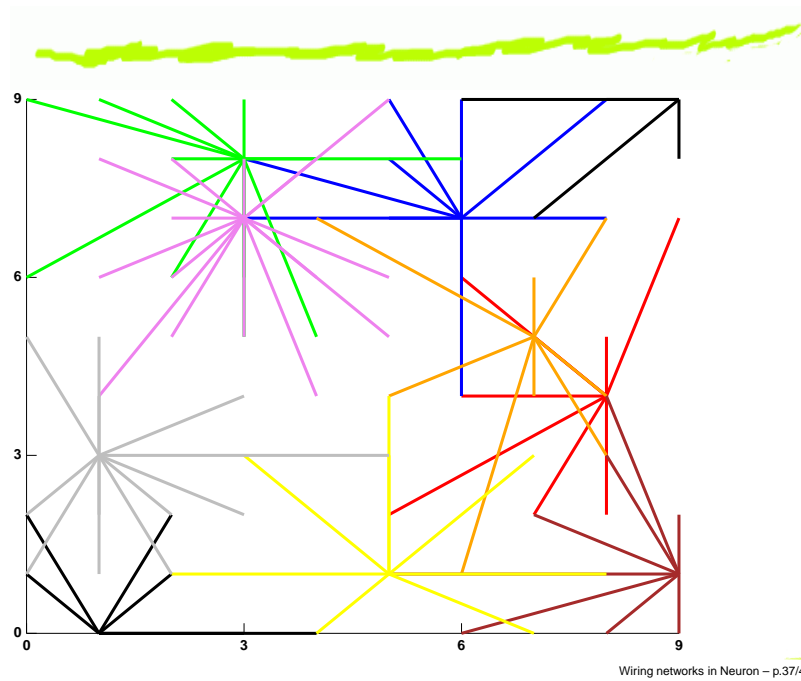
Wiring networks in Neuron – p.35/4;

$p_{ij} = .5$ : **convergence for 10 cells**

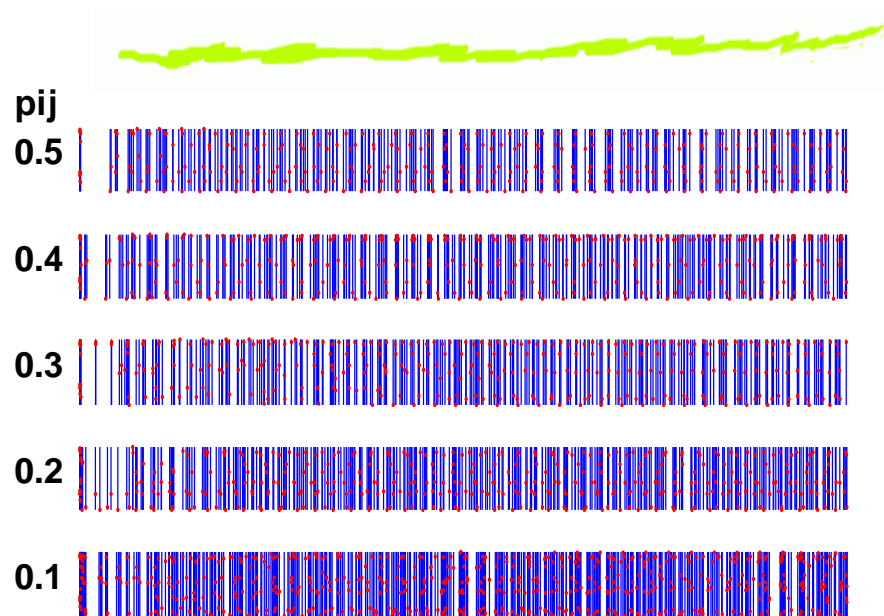


Wiring networks in Neuron – p.36/4;

$p_{ij} = .1$ : *lower density to be tested*

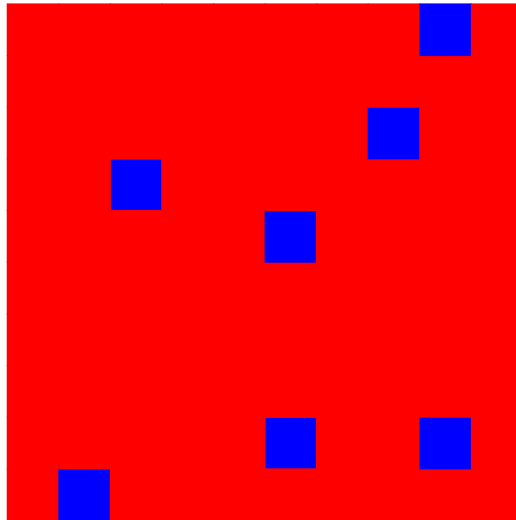


*Doesn't sync very well*



## *Is there localized sync'ing?*

Look at animation.



Wiring networks in Neuron – p.39/4;

## *How to animate*

```
for (tt=0;tt<=tstop;tt+=tstep) {
  for (;tt>tvec.x[ii];ii+=1){
    drv.x[animv.indwhere("==",ind.x[ii])]:
  for (;tt >scr.x[jj];jj+=1){
    drv.x[animv.indwhere("==",ind.x[jj])]:
```

• • •

Wiring networks in Neuron – p.40/4;

## ***Other explorations***

- ⑥ Try weight fall-off rather than restricted wiring
- ⑥ Mix excitatory and inhibitory connects ( $\pm$  Dale)
- ⑥ Connect two populations at various densities

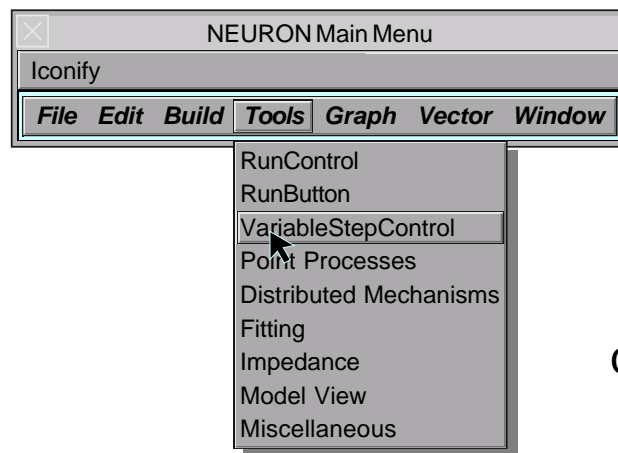
Wiring networks in Neuron – p.41/4;

## ***Advantages of NEURON for networks***

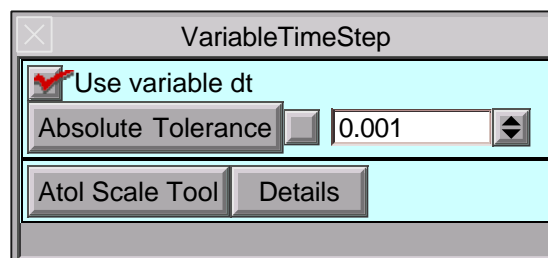
- ⑥ Local variable dt
- ⑥ Mix IF and realistic neurons
- ⑥ Flexibility (/learning curve)
- ⑥ Mike & Ted

Wiring networks in Neuron – p.42/4;

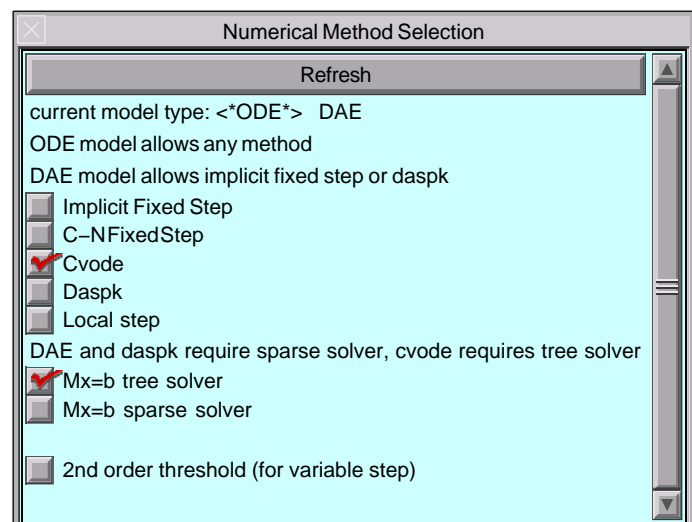


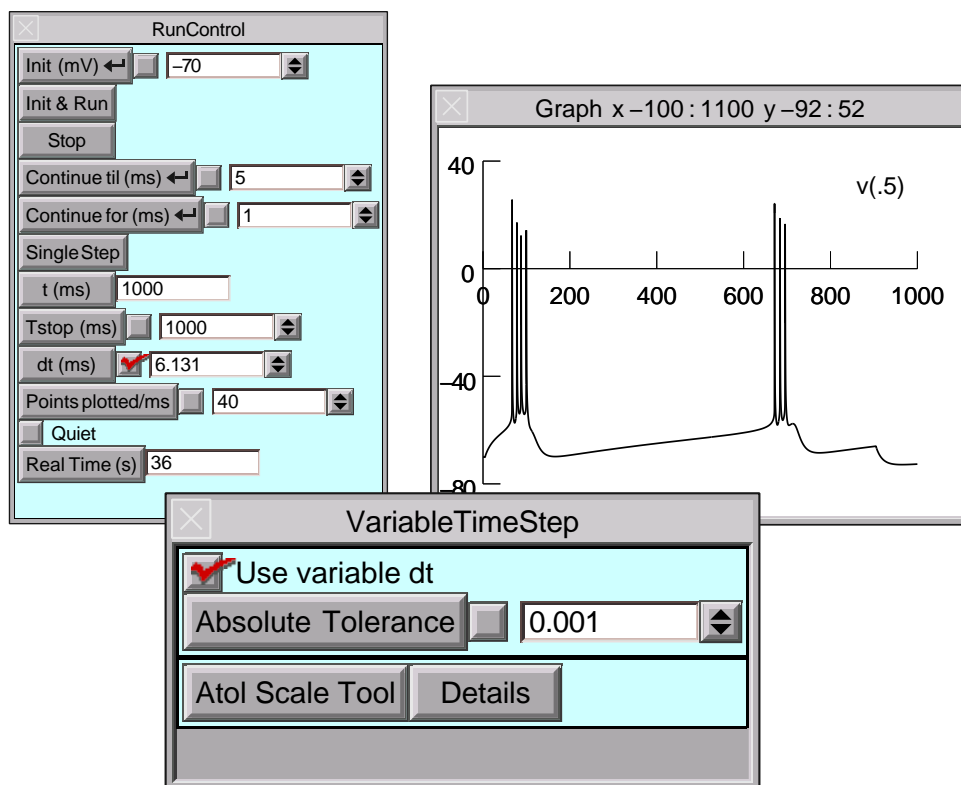
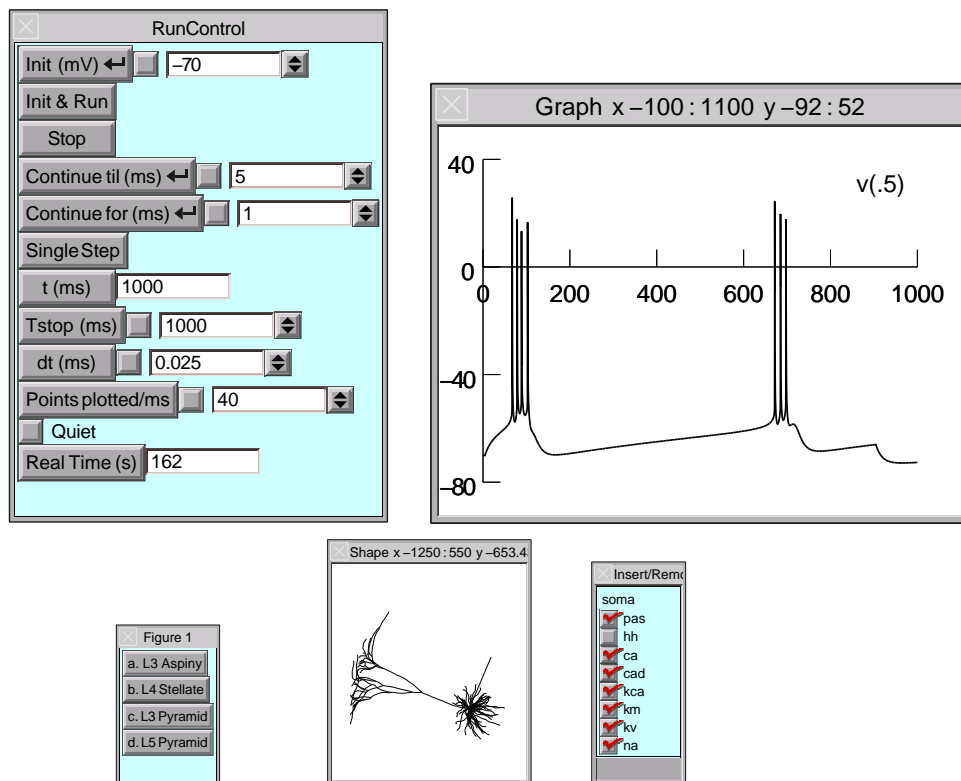


`cvode_active(1)`

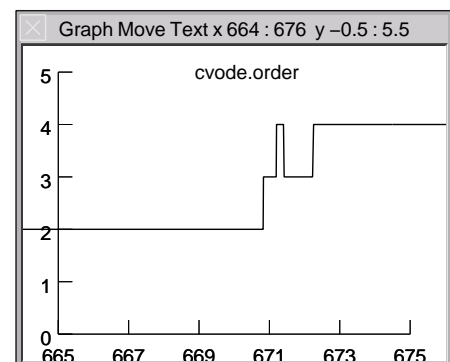
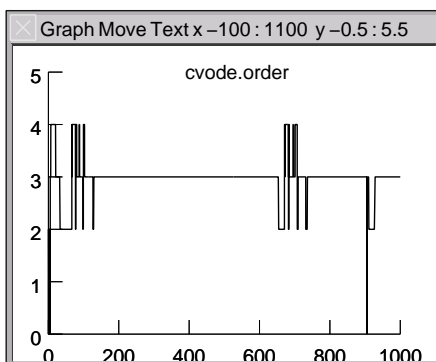
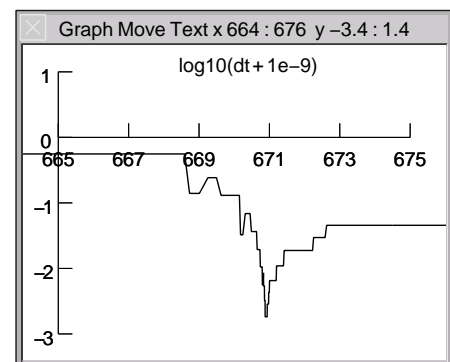
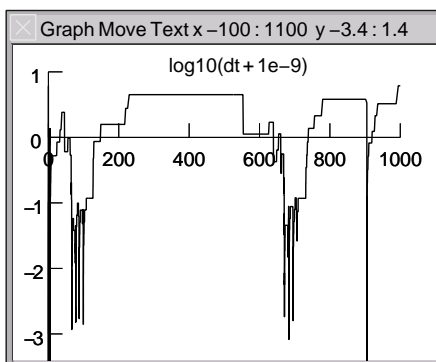
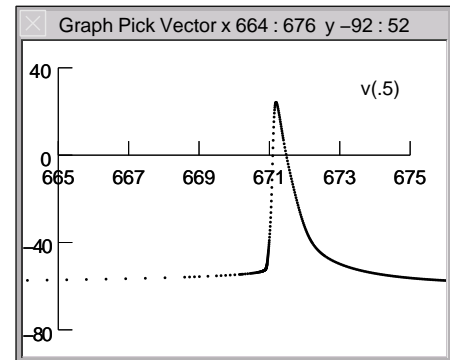
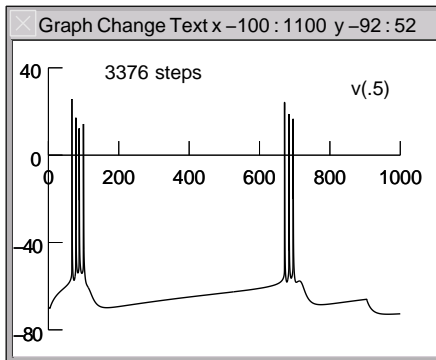


Absolute Tolerance Scale Factors				
Analysis Run Rescale Original				
*10 /10 Hints				
v	1	65	0	
ca_cadifmp	1e-06	3e-06	0	
pump_cadifmp	1e-15	1e-13	0	
pumpca_cadifmp	1e-15	3.6e-15	0	
oca_cachan	1	0.053	0	
n_HHk	1	0.32	0	
m_HHna	1	0.053	0	
h_HHna	1	0.6	0	
Ves_trel	1	0.0004	0	
B_trel	1	0	0	
Ach_trel	1	0	0	
X_trel	1	0	0	









ModelDB: Model Information

<http://senselab.med.yale.edu/senselab/ModelDB/ShowModel.asp?m...>**Spinal Motor Neuron: McIntyre et al 2002**

Simulation of peripheral nervous system (PNS) myelinated axon. This model is described in detail in: McIntyre CC, Richardson AG, Grill WM. (2002)

**Reference:** McIntyre CC, Richardson AG, Grill WM (2002) Modeling the excitability of Mammalian nerve fibers: influence of afterpotentials on the recovery cycle. *J Neurophysiol* **87**:995-1006 [[PubMed](#)]

**Citations** [Citation Browser](#)

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

Model Type: [Axon](#);

Cell Type(s): [Spinal motor neuron](#);

Channel(s): [I<sub>Na,p</sub>](#); [I<sub>Na,t</sub>](#); [I<sub>K</sub>](#); [I<sub>Sodium</sub>](#); [I<sub>Potassium</sub>](#);

Receptor(s):

Transmitter(s):

Simulation Environment: [Neuron](#);

Model Concept(s): [Axonal Action Potentials](#); [Action Potentials](#);

Implementer(s): [MacIntyre, CC](#);

**Search NeuronDB** for information about: [Spinal motor neuron](#); [I<sub>Na,p</sub>](#); [I<sub>Na,t</sub>](#); [I<sub>K</sub>](#); [I<sub>Sodium</sub>](#); [I<sub>Potassium</sub>](#);

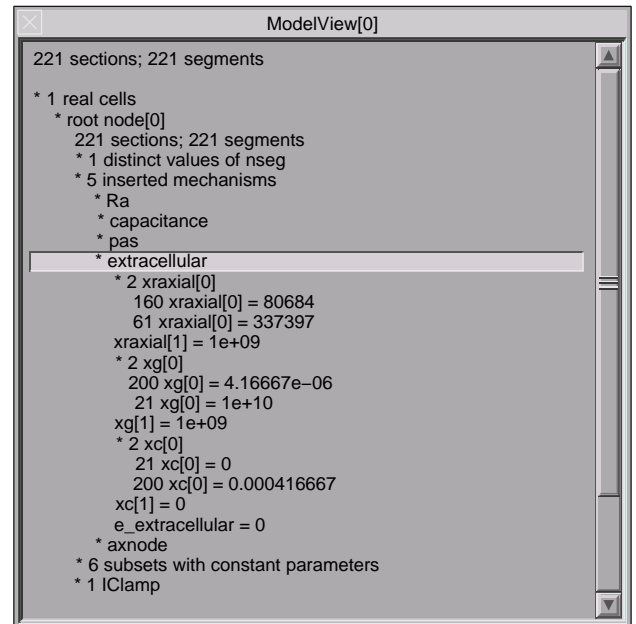
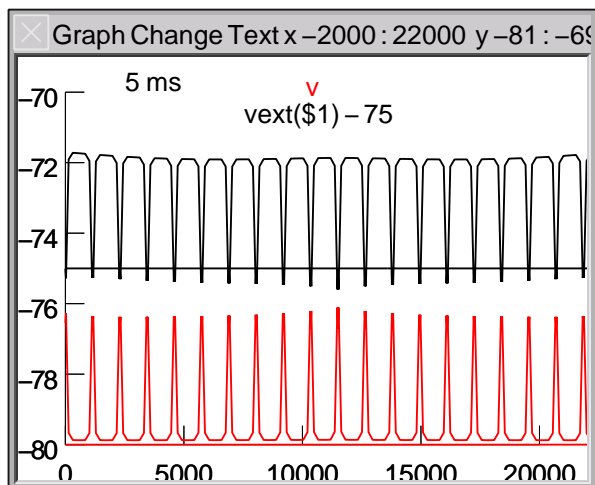
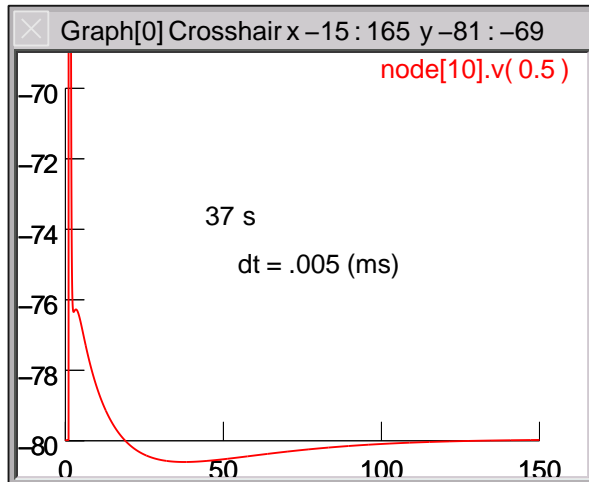
Model files	Download zip file	Auto-launch	<a href="#">Help downloading and running models</a>
\ <a href="#">MRGaxon</a> <a href="#">README</a> <a href="#">AXNODE.mod</a> <a href="#">MRGaxon.hoc</a> <a href="#">mosinit.hoc</a> <a href="#">MRGaxon.ses</a>	SIMULATION OF PNS MYELINATED AXON  This model is described in detail in:  McIntyre CC, Richardson AG, and Grill WM. Modeling the excitability of mammalian nerve fibers: influence of afterpotentials on the recovery cycle. <i>Journal of Neurophysiology</i> 87:995-1006, 2002.  The model is set up to reproduce part of Fig 2A from this paper.  This model can not be used with NEURON v5.1 as errors in the extracellular mechanism of v5.1 exist related to xc. The original stimulations were run on v4.3.1. NEURON v5.2 has corrected the limitations in v5.1 and can be used to run this model.  Please contact mcintyre@bme.jhu.edu if you have any questions about		

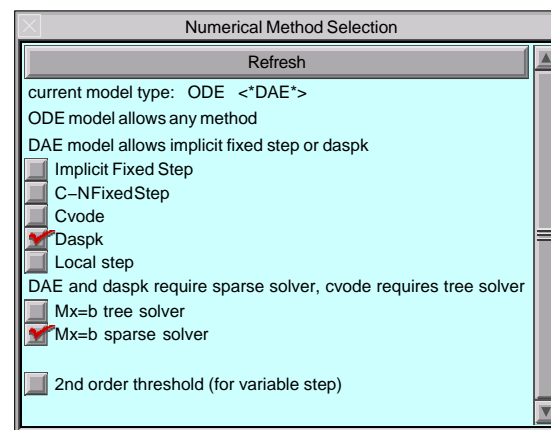
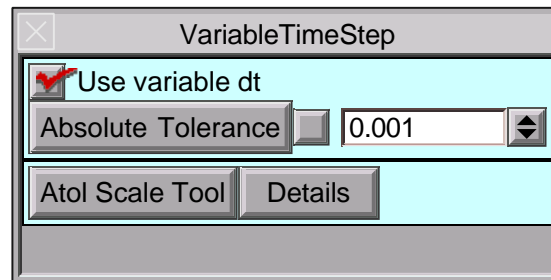
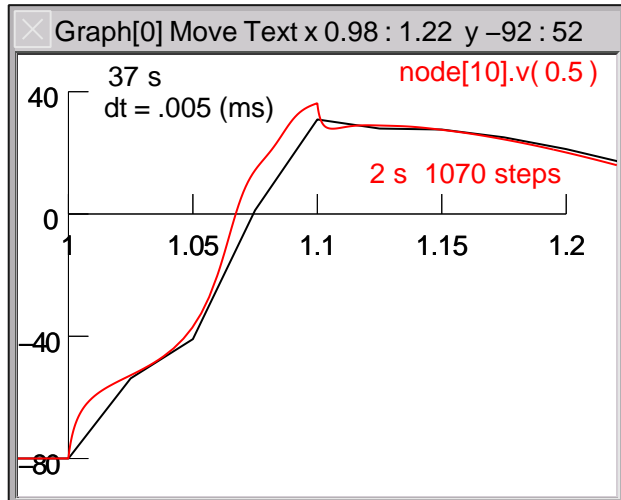
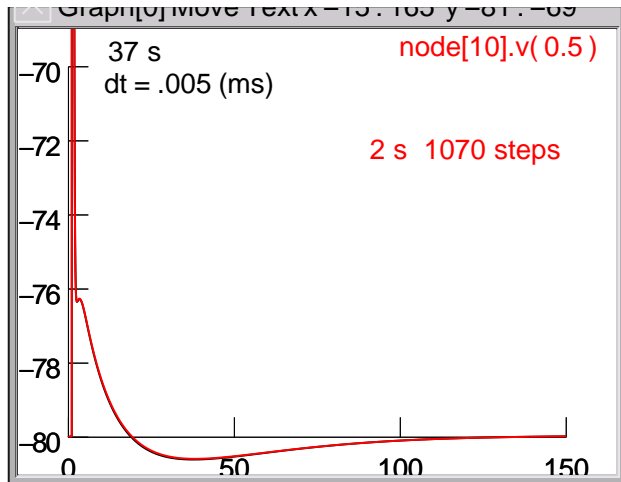
Total site hits since January 1, 2002: **346093**

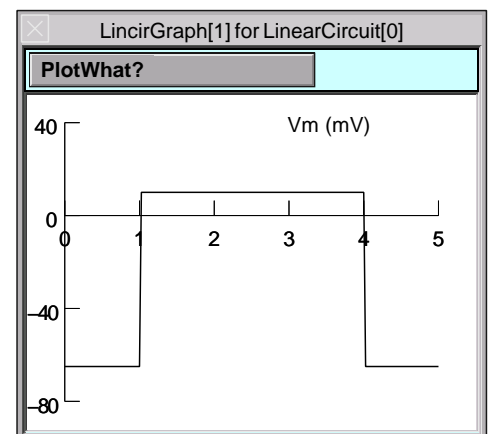
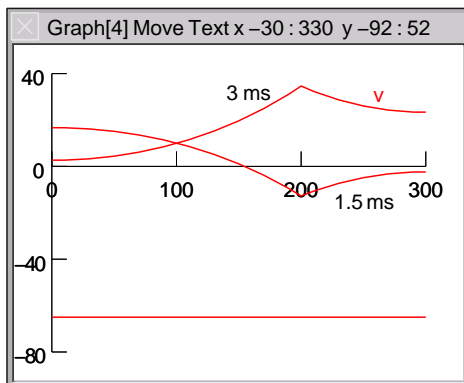
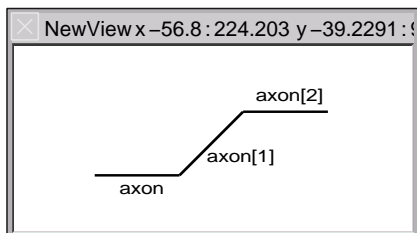
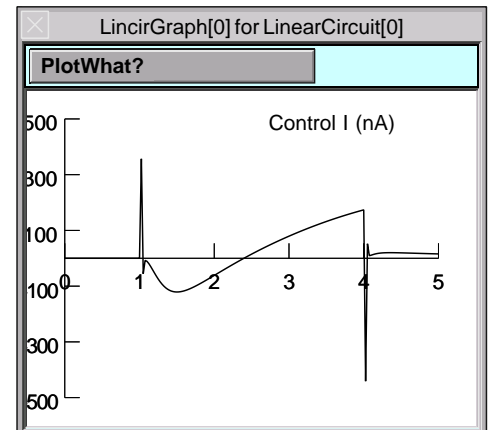
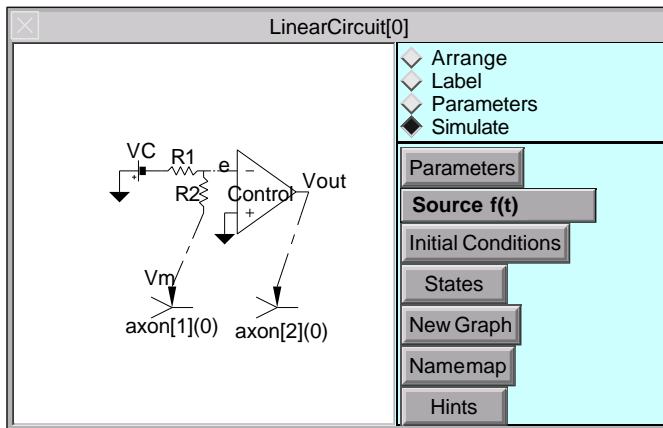
[ModelDB Home](#) [SenseLab Home](#) [Help](#)

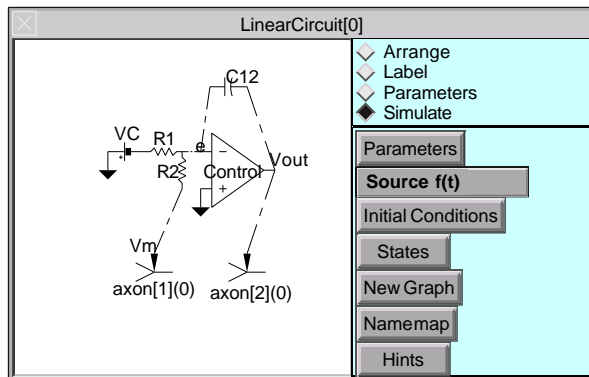
Questions, comments, problems? Email the [ModelDB Administrator](#)

[How to cite ModelDB](#)









Values for LinearCircuit[0]

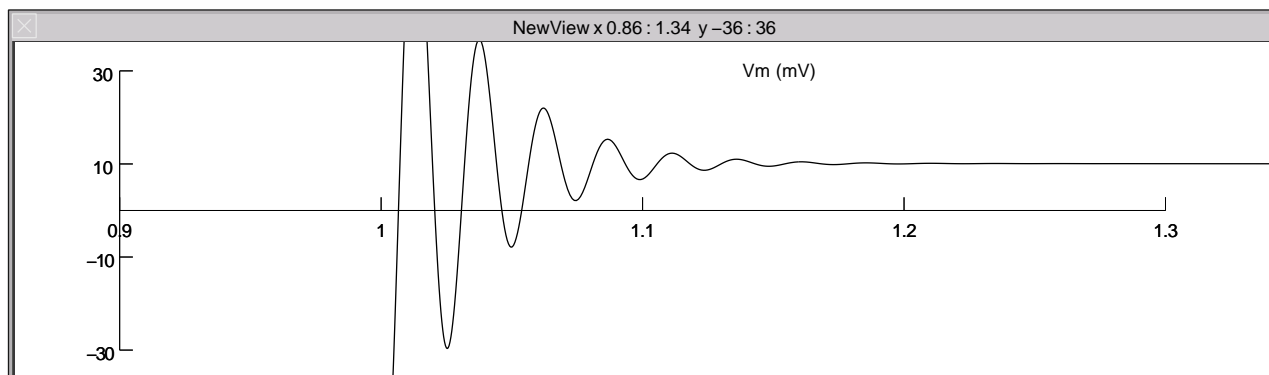
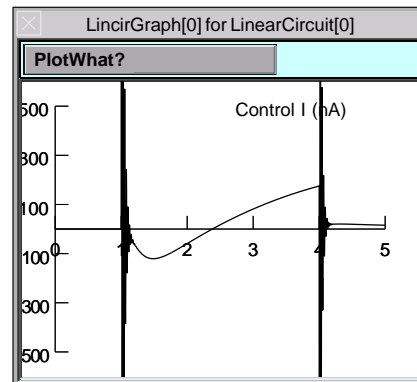
Control Gain	1e+05
Control Tau (ms)	0
R1 (Mohm)	1e+05
R2 (Mohm)	1e+05
C12 (nF)	1e-08

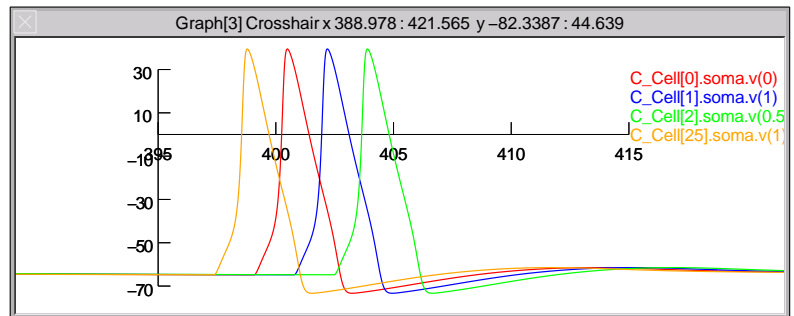
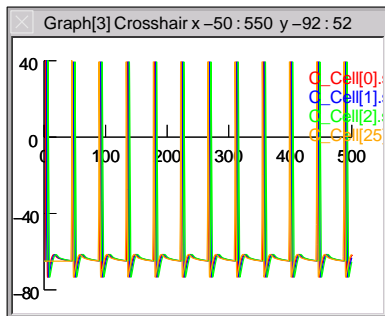
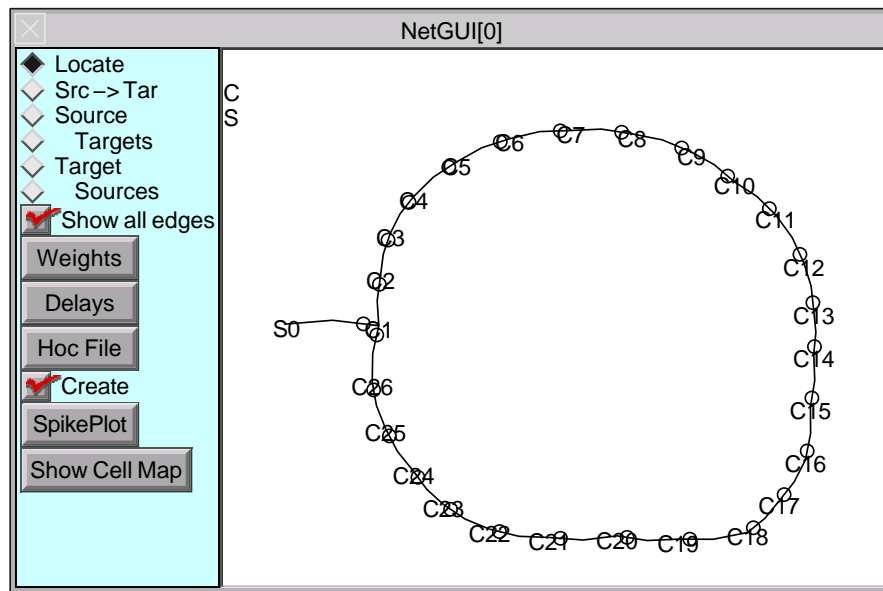
VariableTimeStep

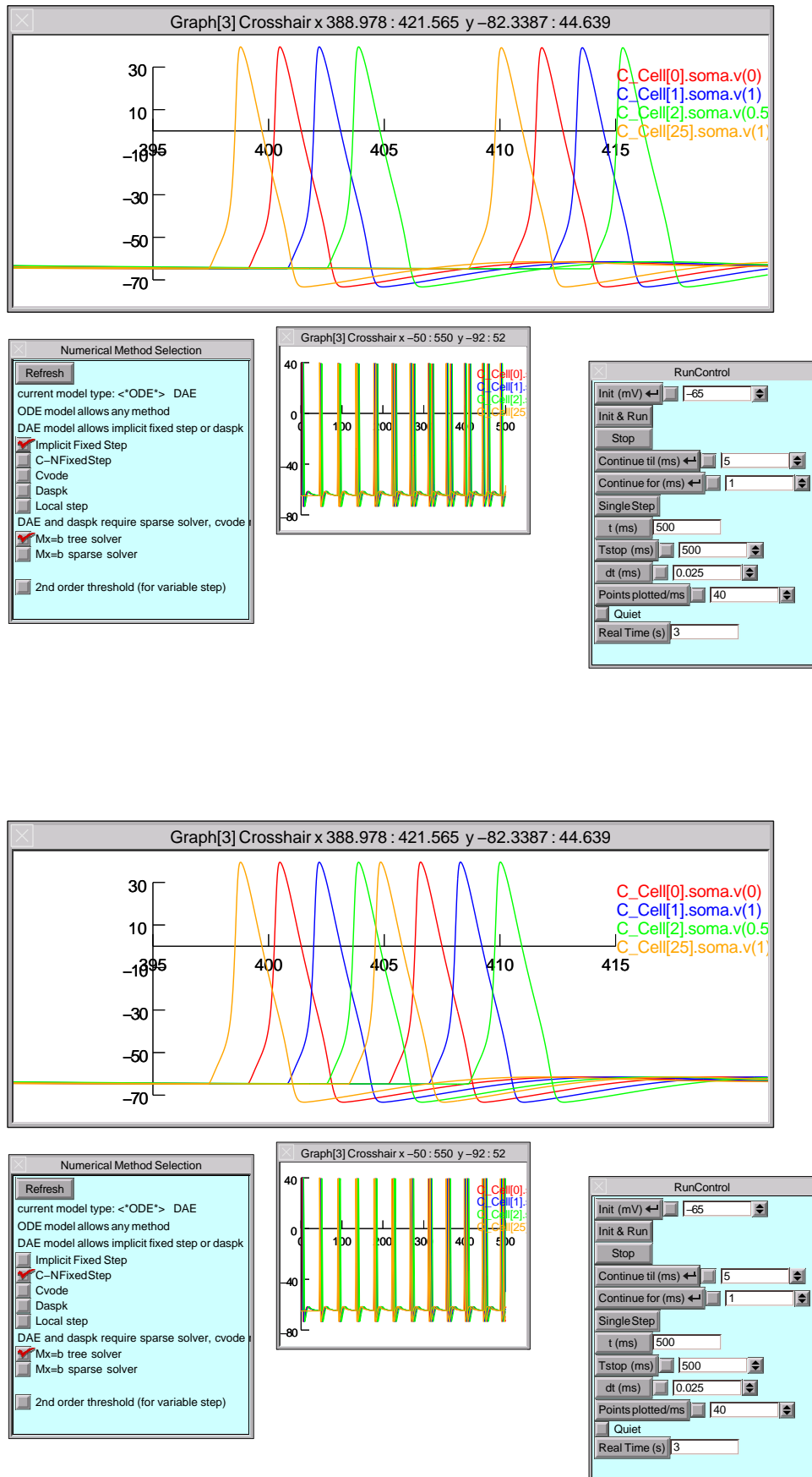
☒ Use variable dt

Absolute Tolerance 0.001

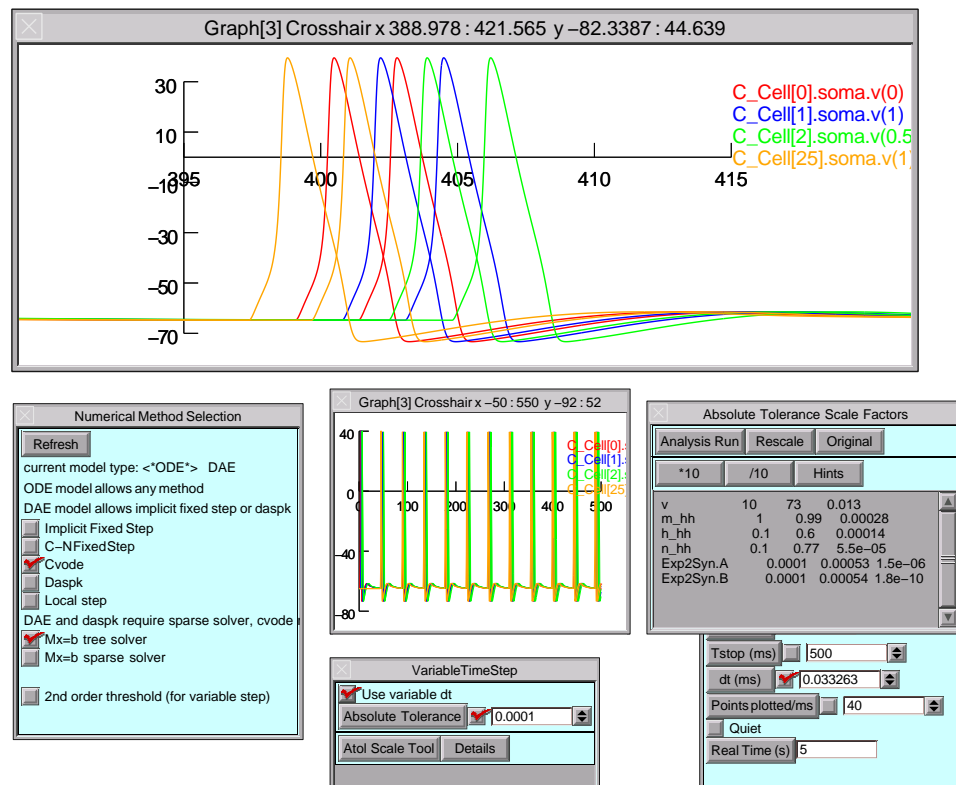
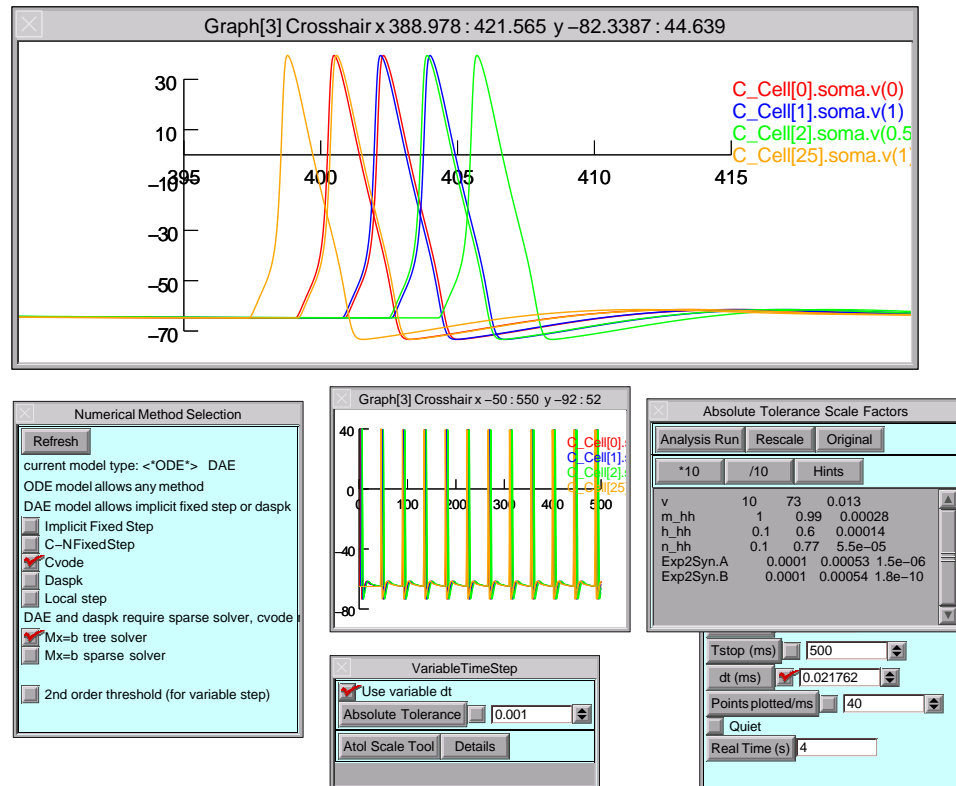
Atol Scale Tool Details

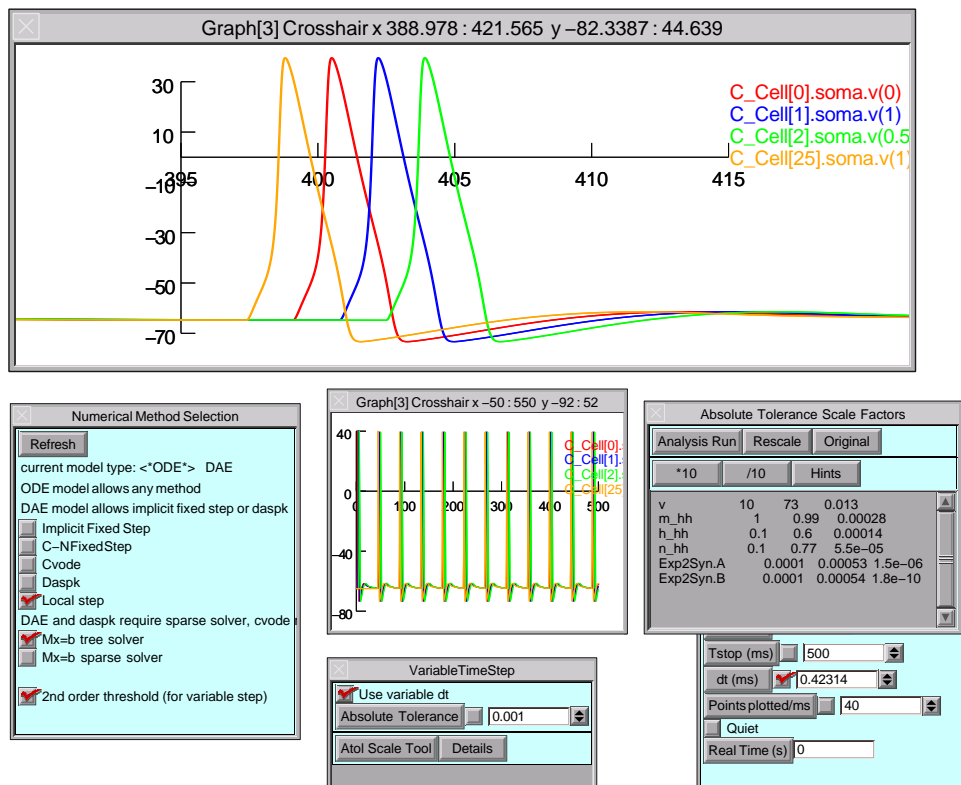
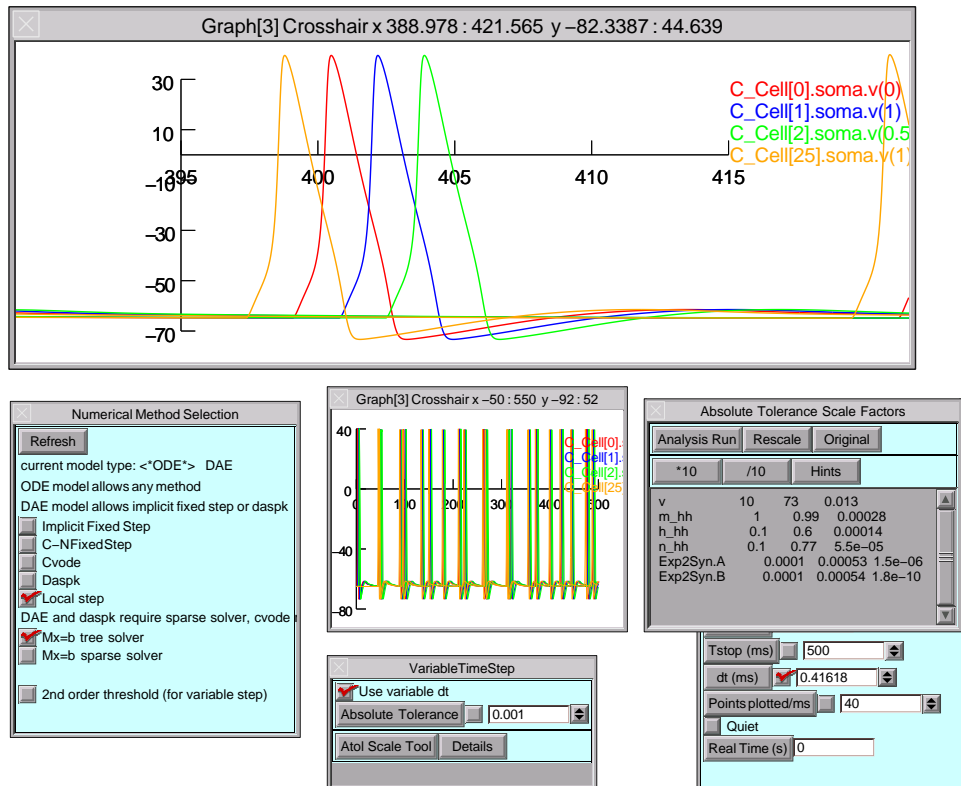


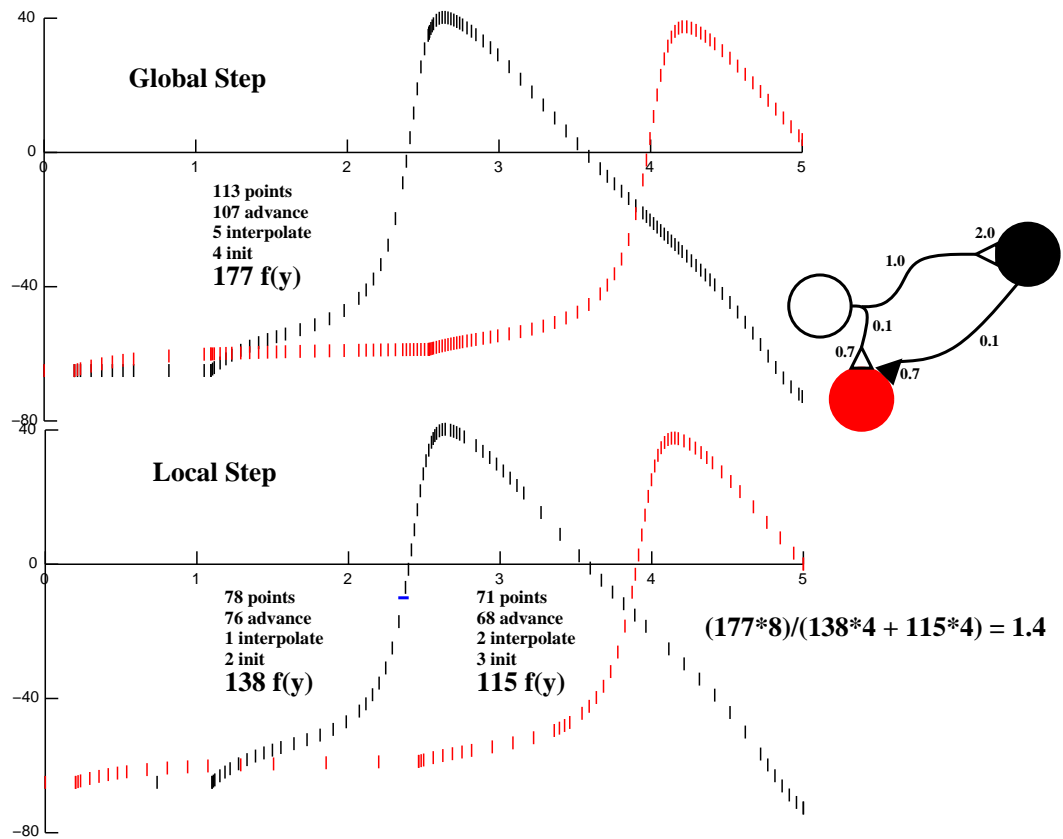






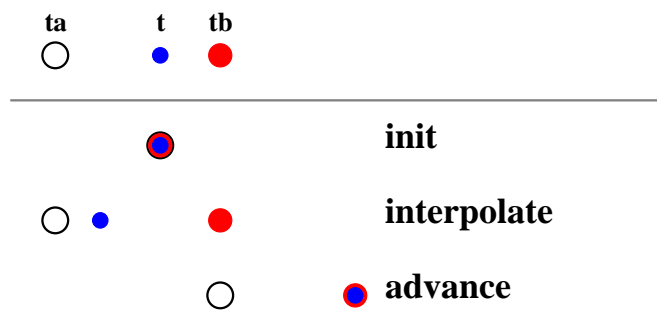


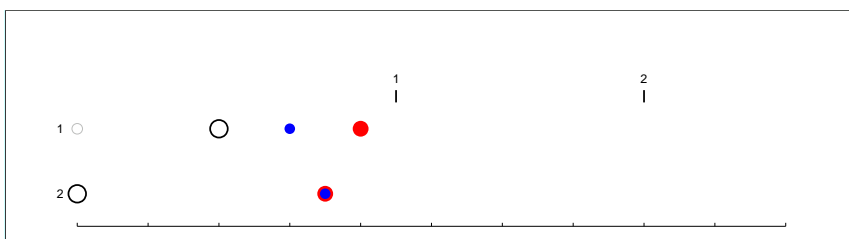
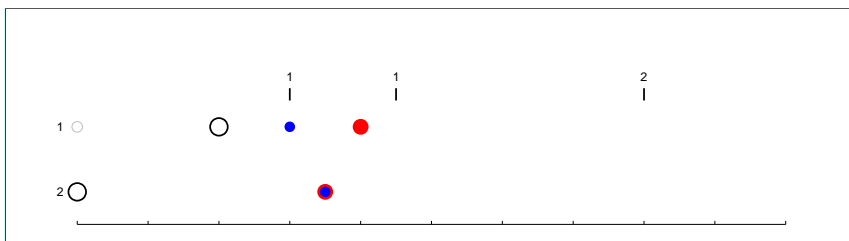
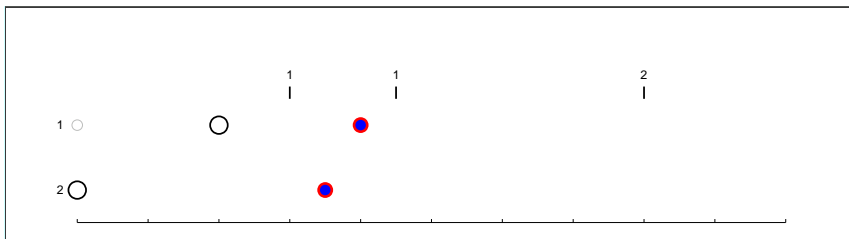
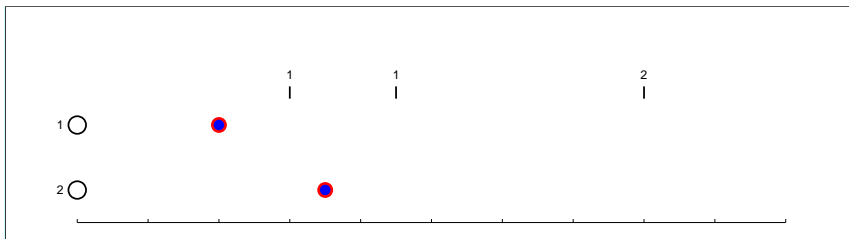
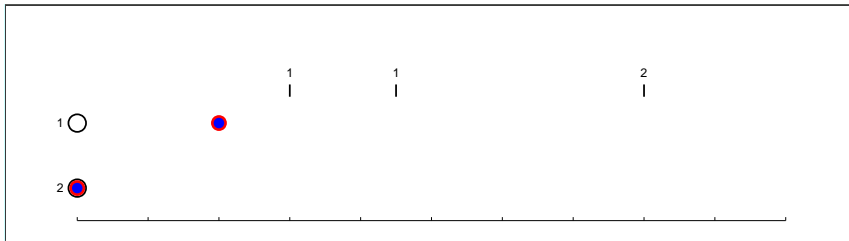
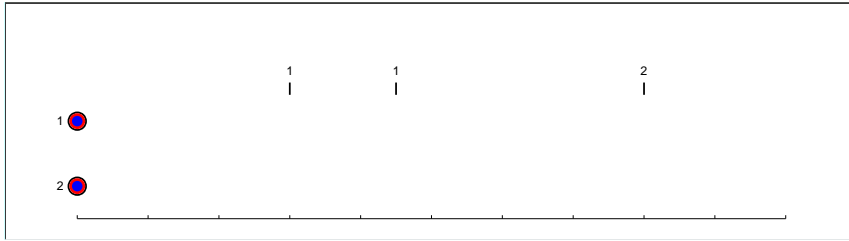


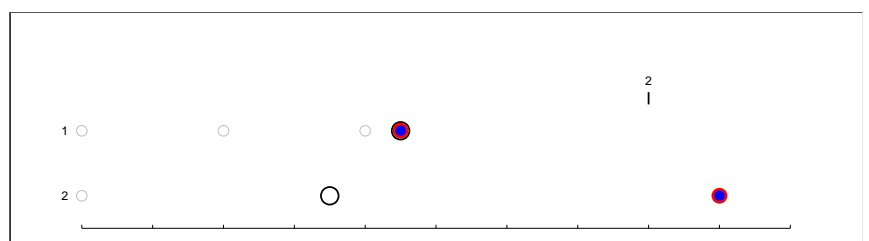
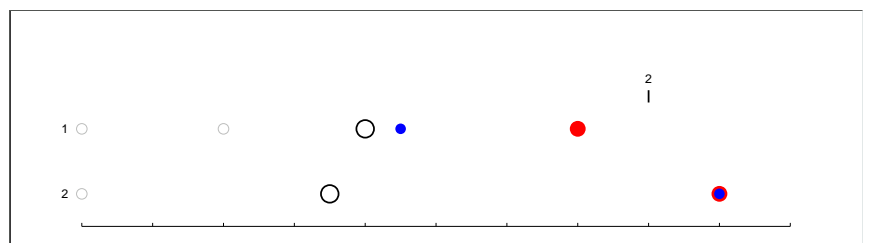
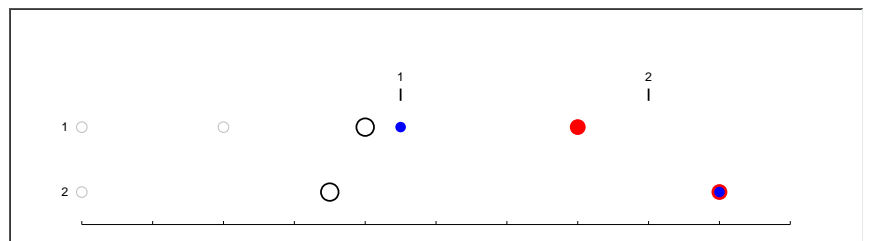
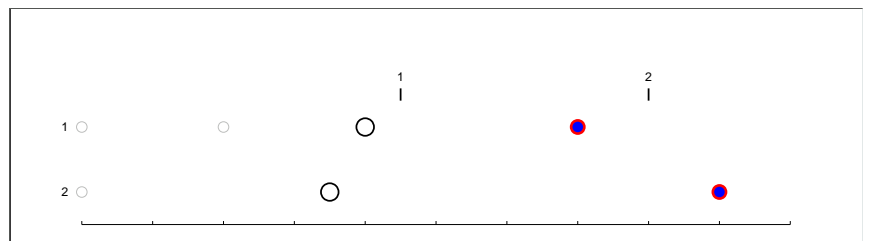
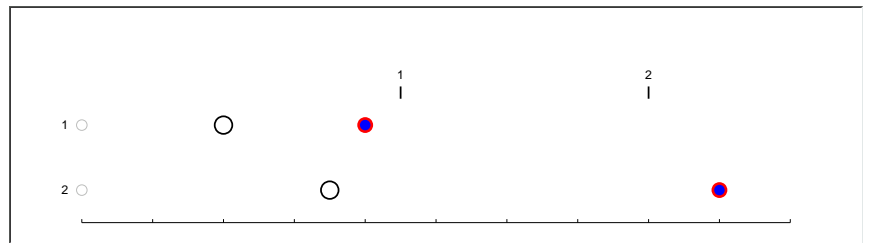
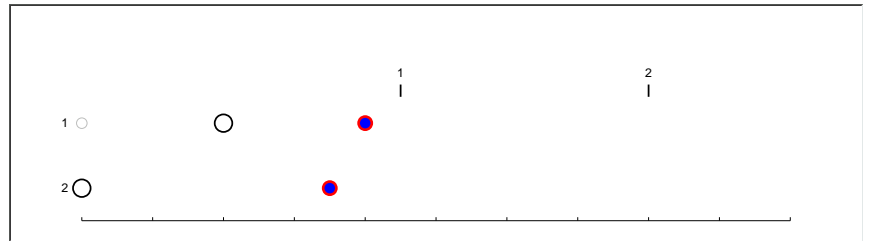


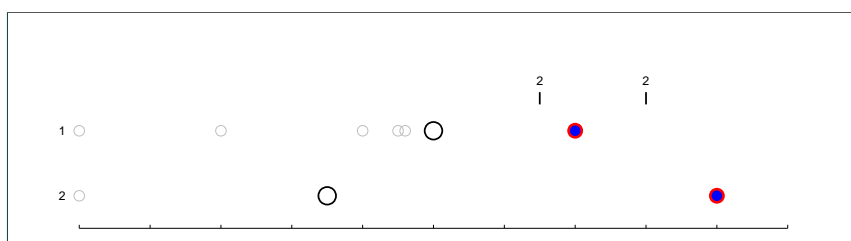
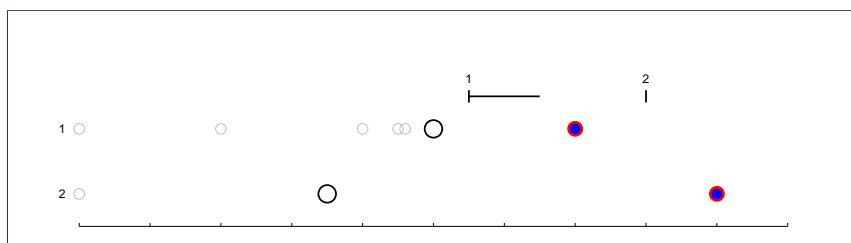
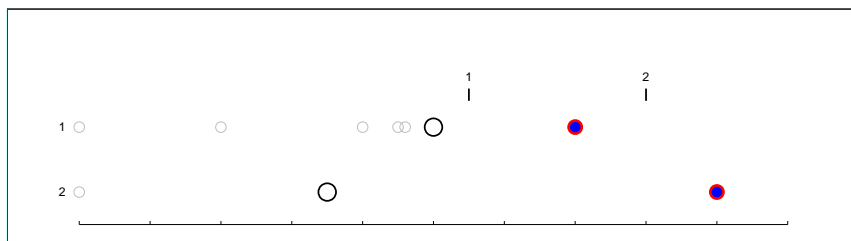
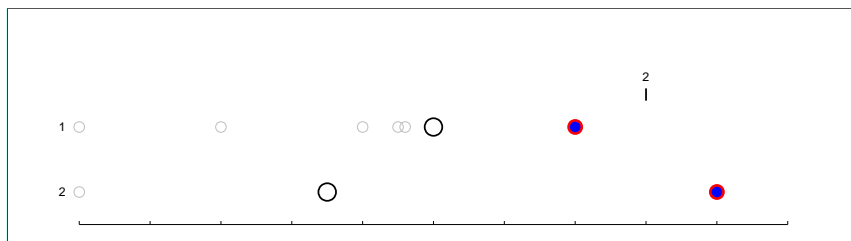
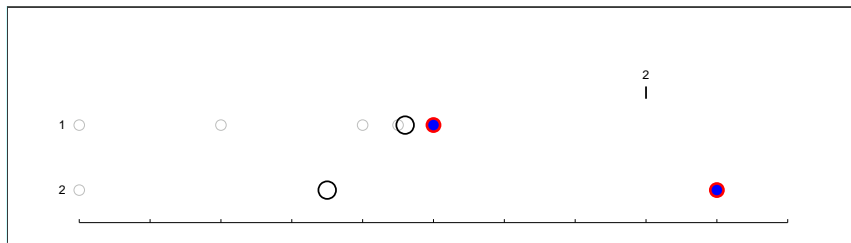
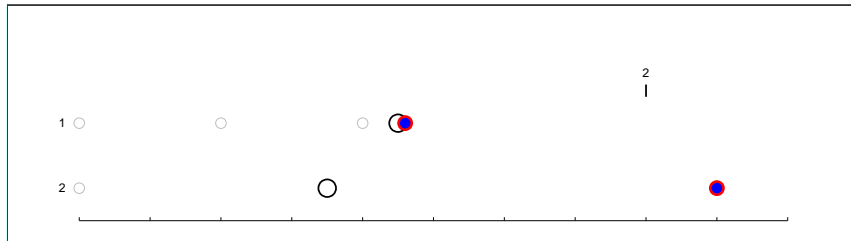
### One integrator instance per cell

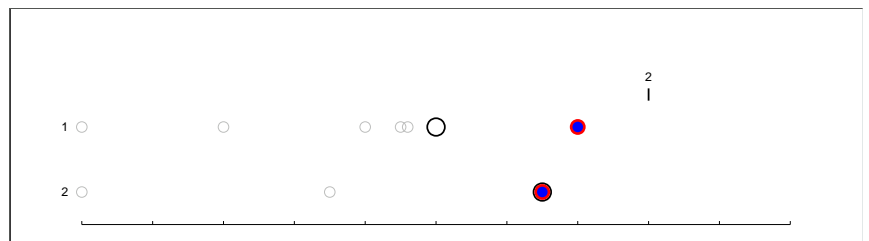
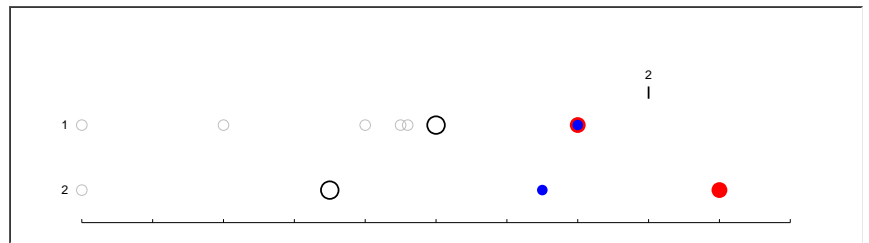
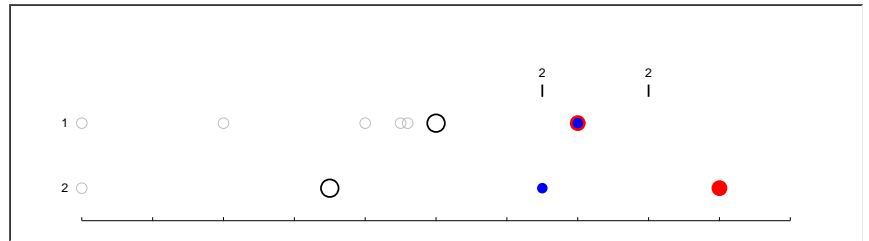
$$\forall i, j: \quad ta_i \leq tb_j$$











```
STATE { o }
```

```
BREAKPOINT {
  SOLVE state
  ik = gbar*o*(v - ek)
}
```

```
LOCAL fac
```

```
PROCEDURE state() {
  rate(v)
  o = o + fac*(oinf - o)
}
```

```
PROCEDURE rate(v (mV)) {
  LOCAL a
  a = alp(v)
  tau = 1/(a + bet(v))
  oinf = a*tau
  fac = (1 - exp(-dt/tau))
}
```

```
STATE { o }
```

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

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

```
PROCEDURE rate(v (mV)) {
  LOCAL a
  a = alp(v)
  tau = 1/(a + bet(v))
  oinf = a*tau
}
```

```
BREAKPOINT {
  if (t >= del) { ← at_time(del)
    i = f(t-del)
  }else{
    i = 0
  }
}
```

**(deprecated)**



```

INITIAL {
    on = 0
    net_send(del, 1)
}

BREAKPOINT {
    if (t >= del) {
        i = f(t-del)
    }else{
        i = 0
    }
}

BREAKPOINT {
    if (on == 1) {
        i = f(t-del)
    }else{
        i = 0
    }
}

NET_RECEIVE(w) {
    if (flag == 1) {
        on = 1
    }
}

```

TITLE minimal model of GABA<sub>A</sub> receptors

COMMENT

Minimal kinetic model for GABA<sub>A</sub> receptors

Model of Destexhe, Mainen & Sejnowski, 1994:

(closed) + T  $\leftrightarrow$  (open)

The simplest kinetics are considered for the binding of transmitter (T) to open postsynaptic receptors. The corresponding equations are in similar form as the Hodgkin–Huxley model:

$$dr/dt = \alpha * [T] * (1-r) - \beta * r$$

$$I = g_{\max} * [\text{open}] * (V - E_{\text{rev}})$$

where [T] is the transmitter concentration and r is the fraction of receptors in the open form.

If the time course of transmitter occurs as a pulse of fixed duration, then this first-order model can be solved analytically, leading to a very fast mechanism for simulating synaptic currents, since no differential equation must be solved (see Destexhe, Mainen & Sejnowski, 1994).

```

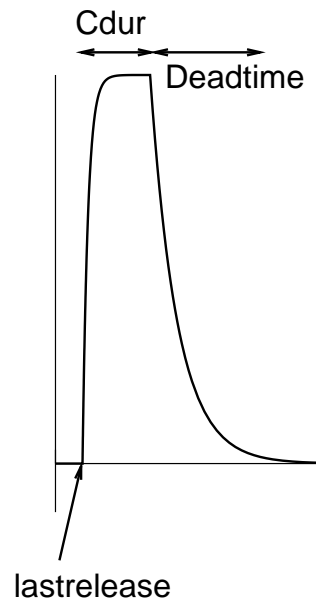
PROCEDURE release() { LOCAL q
:will crash if user hasn't set pre with the connect statement

q = ((t - lastrelease) - Cdur) : time since last release ended

                                : ready for another release?
if (q > Deadtime) {
  if (pre > Prethresh) {       : spike occurred?
    C = Cmax                   : start new release
    R0 = R
    lastrelease = t
  }
} else if (q < 0) {             : still releasing?
  : do nothing
} else if (C == Cmax) {        : in dead time after release
  R1 = R
  C = 0.
}

if (C > 0) {                   : transmitter being released?
  R = Rinf + (R0 - Rinf) * exp(-(t - lastrelease) / Rtau)
} else {                       : no release occurring
  R = R1 * exp(-Beta * (t - (lastrelease + Cdur)))
}
}

```



```

...
dr/dt = alpha * [T] * (1-r) - beta * r

```

where [T] is the transmitter concentration and r is the fraction of receptors in the open form.

```

...
INITIAL {
  t0 = 0
  r = 0
}

DERIVATIVE state {
  r' = (rinf - r)/rtau
}

NET_RECEIVE(w) {
  if (flag == 0) { : external spike, transmitter on
    rinf = alpha*T/(alpha*T + beta)
    rtau = 1/(alpha*T + beta)
    net_send(Cdur, 1)
  } else if (flag == 1) { :transmitter off
    rinf = 0
    rtau = 1/beta
  }
}

```

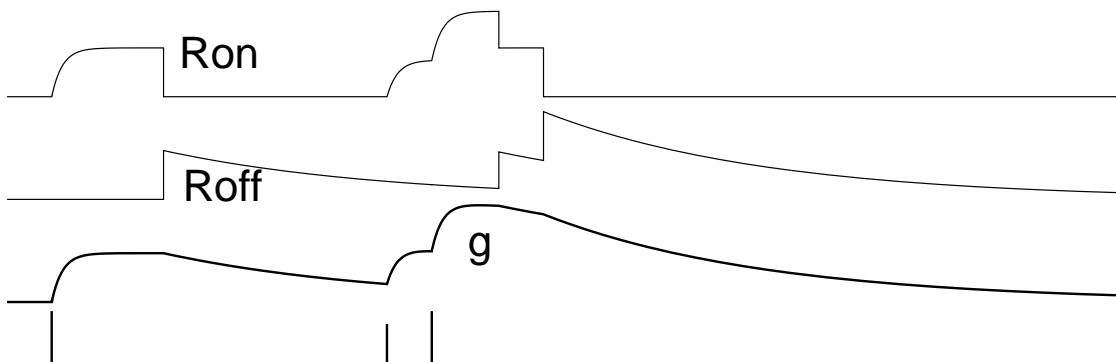
```

STATE {Ron Roff}
INITIAL {
  Ron = 0  Roff = 0
  Rinf = Alpha / (Alpha + Beta)
  Rtau = 1 / (Alpha + Beta)
  Rdelta = Rinf*(1 - exp(-Cdur/Rtau))
  synon = 0
}
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) {
  if (flag == 0) { : spike - T on
    synon = synon + weight
    net_send(Cdur, 1)
  }else{ : transmitter off
    synon = synon - weight
    Ron = Ron - weight*Rdelta
    Roff = Roff + weight*Rdelta
  }
}

```



```

setpointer gabaa[i], cell[j].axon.v(1)
gabaa[i].Prethresh = -10

```



```

cell[j].axon { nc = new NetCon(&v(1), gabaa[i]) }
nc.threshold = -10
nc.delay = 0

```

```
proc advance() {
  fadvance()
  if (t == t1) { p() }
}
```



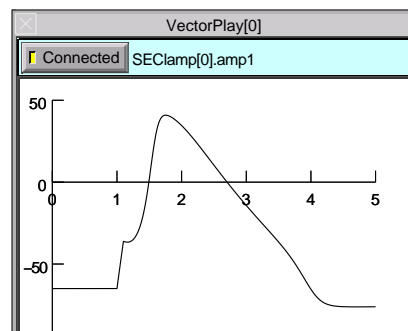
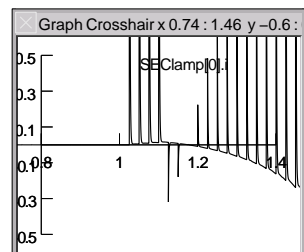
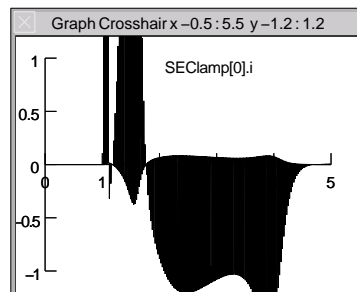
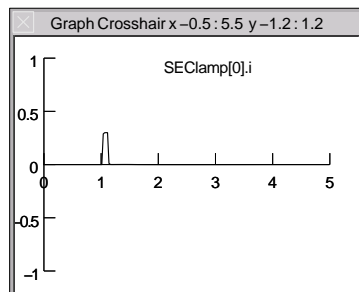
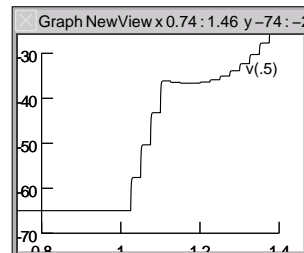
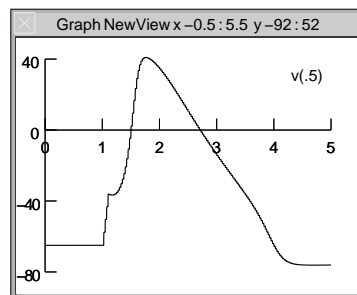
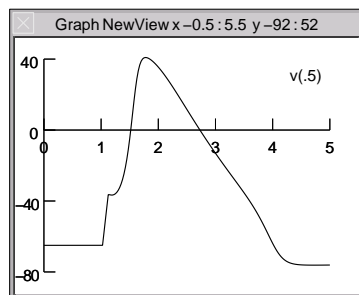
```
fh = new FInitializeHandler("ev()")
proc ev() {
  ccode.event(t1, "p()")
}
```

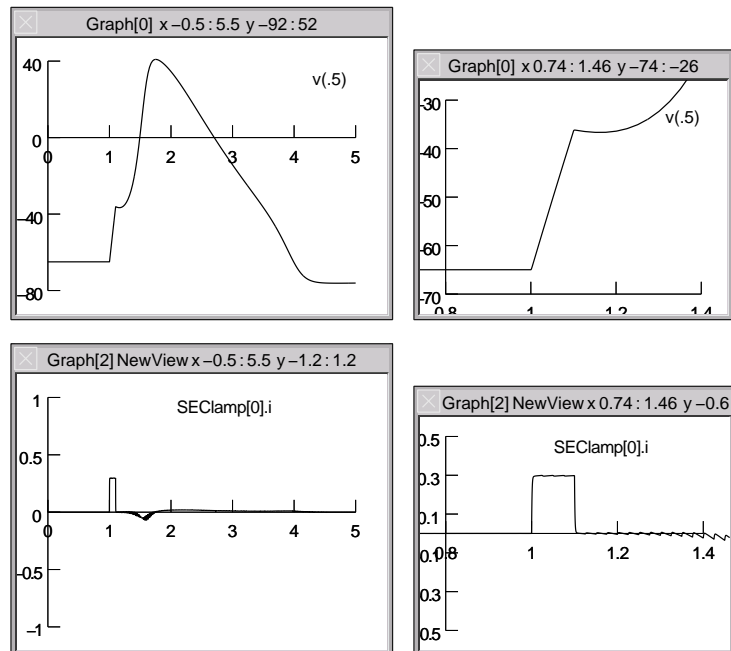
```
proc advance() {
  fadvance()
  if (soma.v(.5) > 10) { p() }
}
```



```
soma { nc = new NetCon(&v(.5), nil)}
nc.threshold = 10
nc.record("p()")
```

```
proc p() {
  // if ANY parameters or states
  // change then be sure to
  ccode.re_init()
}
```





```
soma vvec.play(&SEClamp[0].amp1, tvec, 1)
```



## Parallel Computation

"Faster" is the only reason

But...

- greater programming complexity
- new kinds of bugs
- ...and not much help for fixing them.

Can the day or week of user effort be recovered?

- 8192 processor EPFL IBM BlueGene
- 1 hour at 700MHz
- 3 months at 3GHz

## Parallel Computation

A simulation run takes about a second

- want to do 1000's of them,
- varying a dozen or so parameters.

A simulation run takes hours.

- want to spread the problem over several machines.

## Parallel Computation

A simulation run takes hours.

want to spread the problem over several machines.

Network

Subnets on different machines

Cells communicate by:

logical spike events with significant  
axonal, synaptic delay.

postsynaptic conductance depends  
continuously on presynaptic voltage.

gap junctions

## Parallel Computation

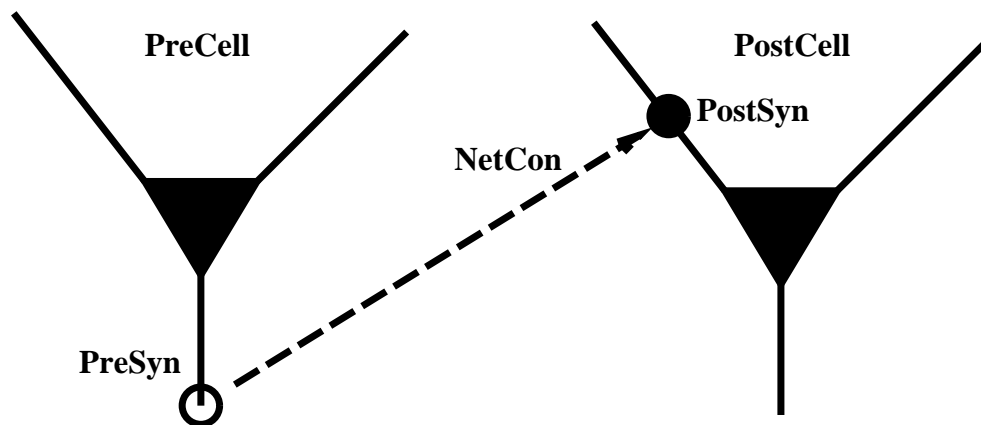
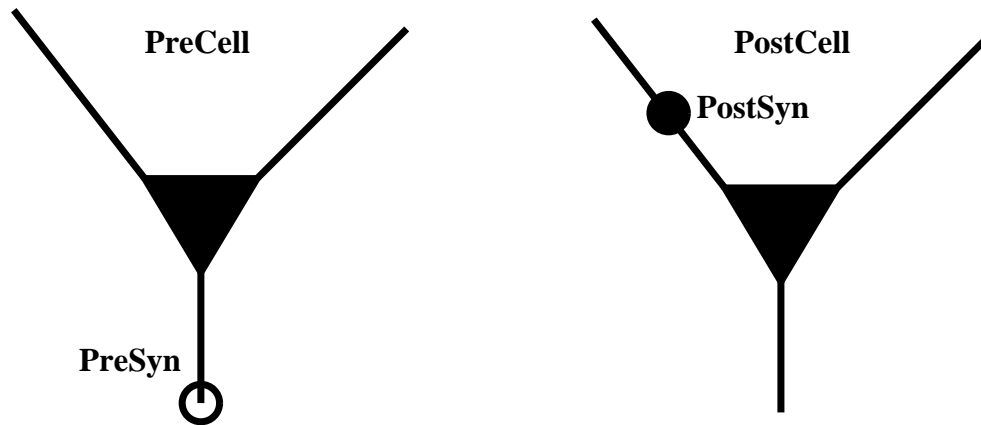
A simulation run takes hours.

want to spread the problem over several machines.

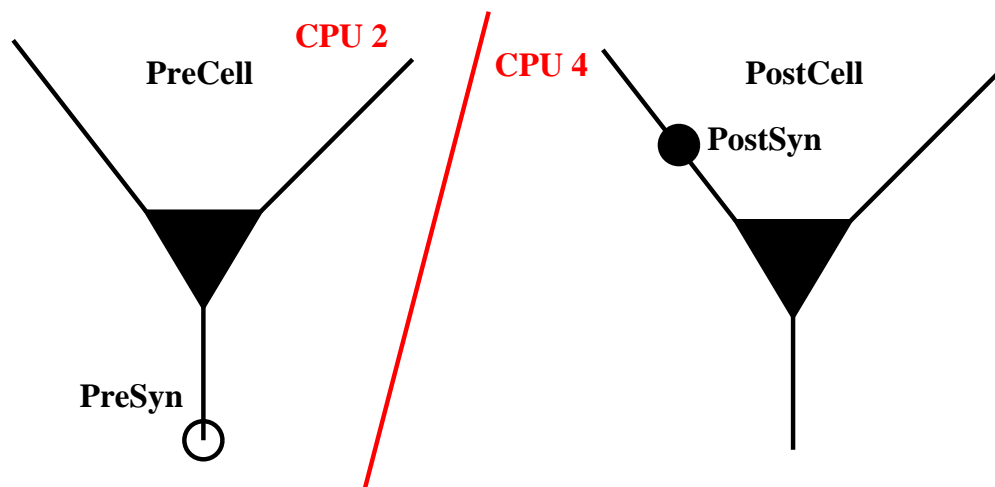
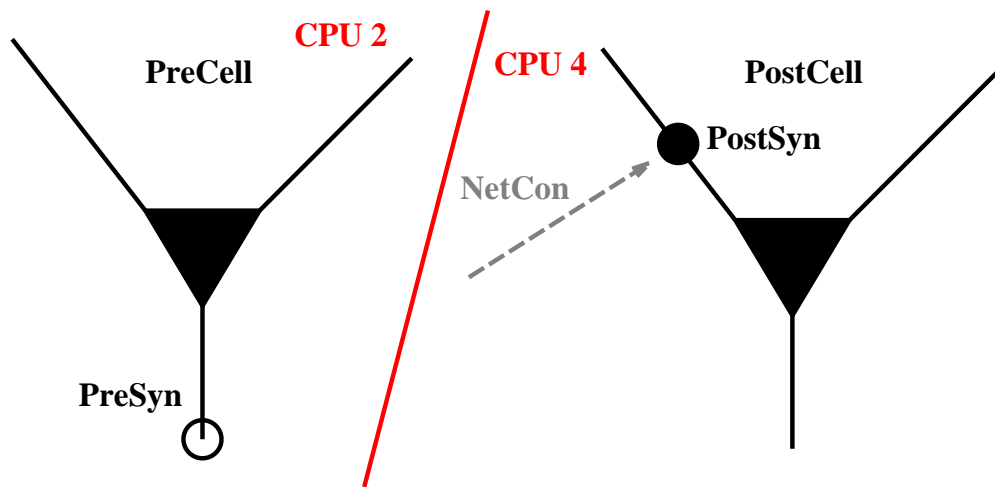
Single cells

portions of the tree cable equation on  
different machines.

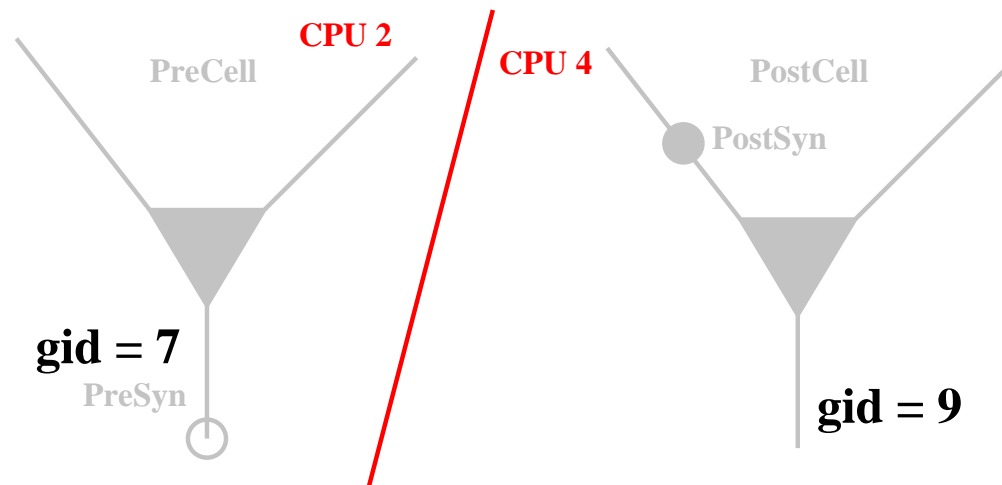




```
nc = new NetCon(PreSyn, PostSyn)
```



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



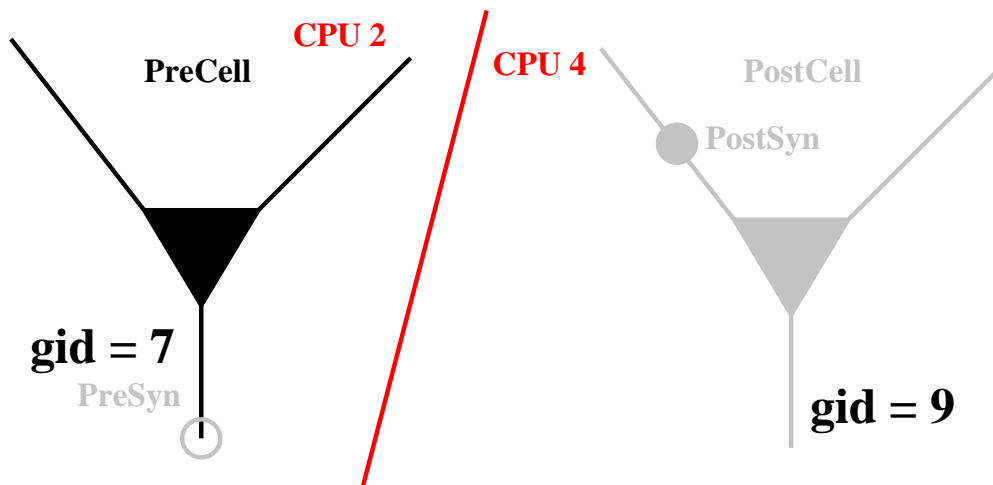
**Every spike source (cell) must have a global id number.**

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:**

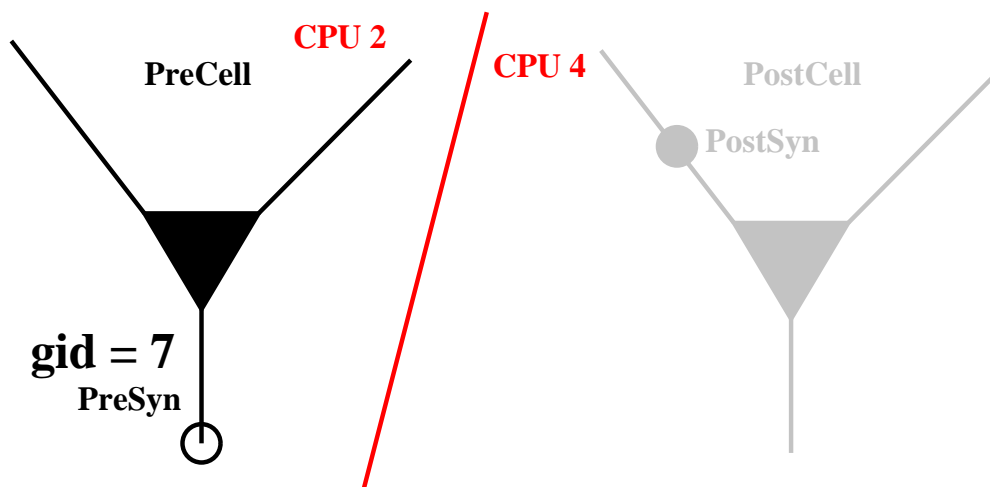
```
for (gid = pc.id; gid < ncell; gid += pc.nhost)
    pc.set_gid2node(gid, pc.id)
    ...
}
```

**body executed only ncell/nhost times, not ncell.**



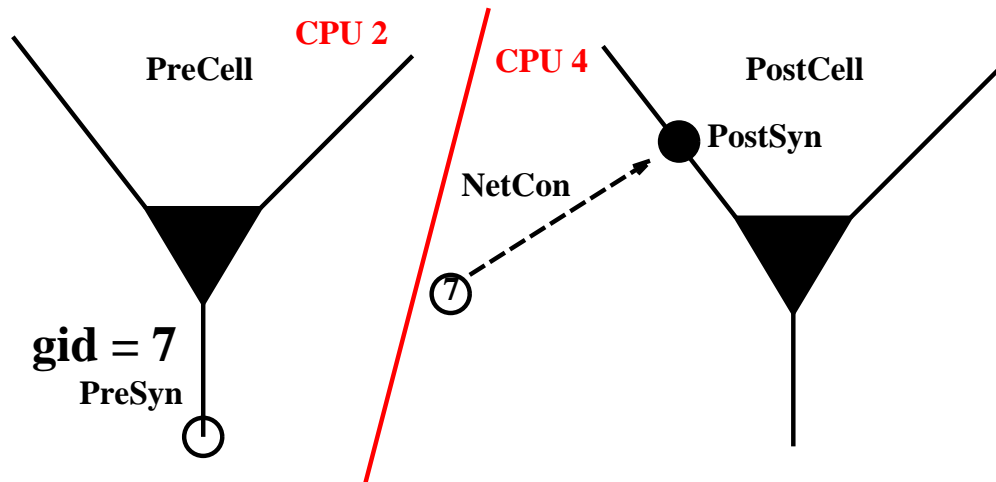
Create cell only where the gid exists.

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



Associate gid with spike source.

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



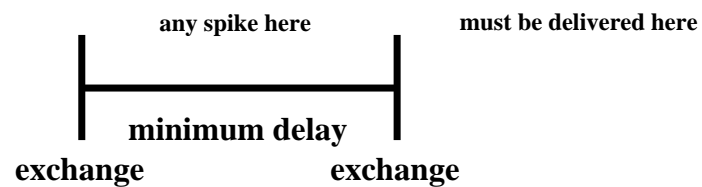
**Create NetCon on CPU where target exists.**

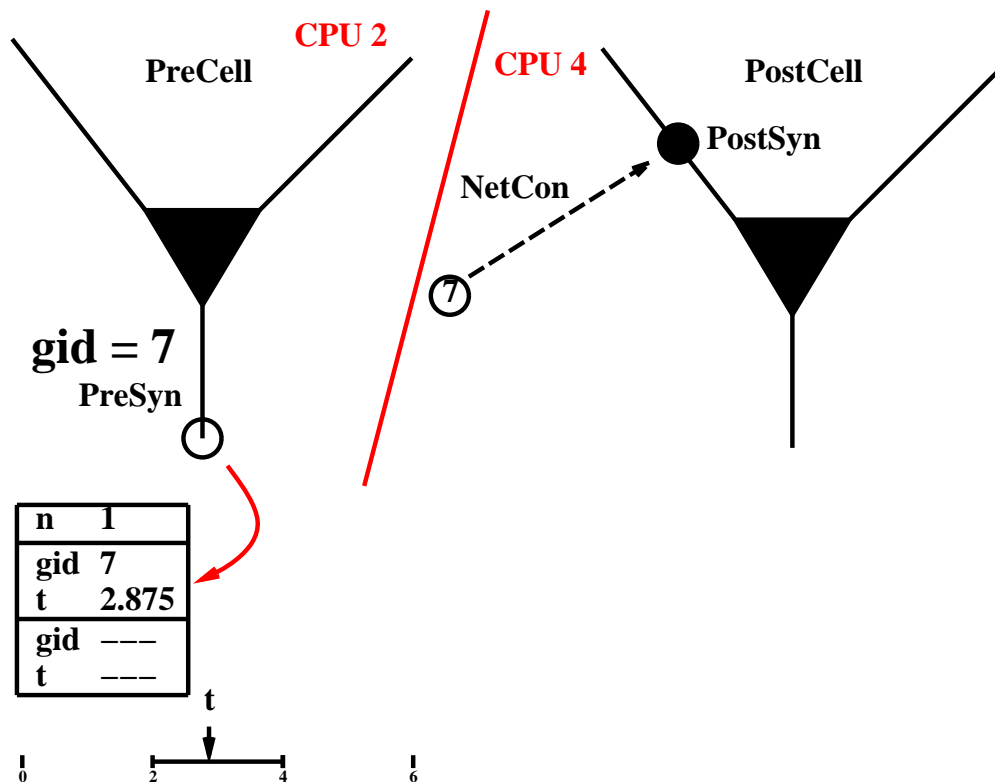
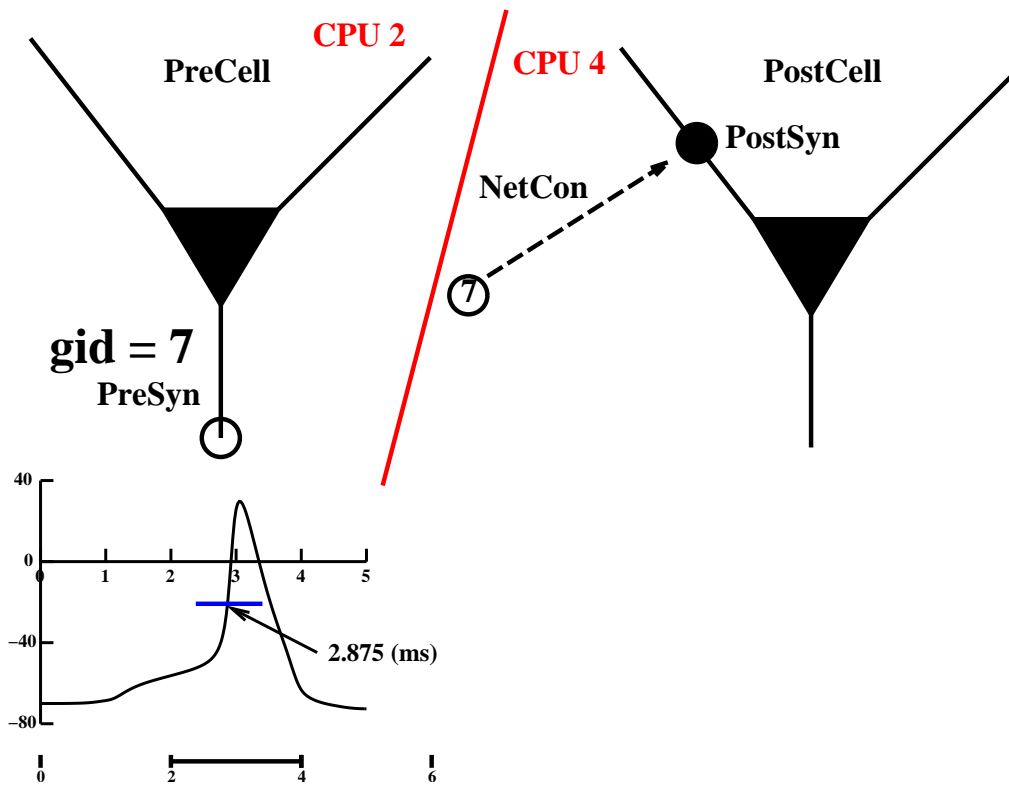
```
nc = pc.gid_connect(7, PostSyn
```

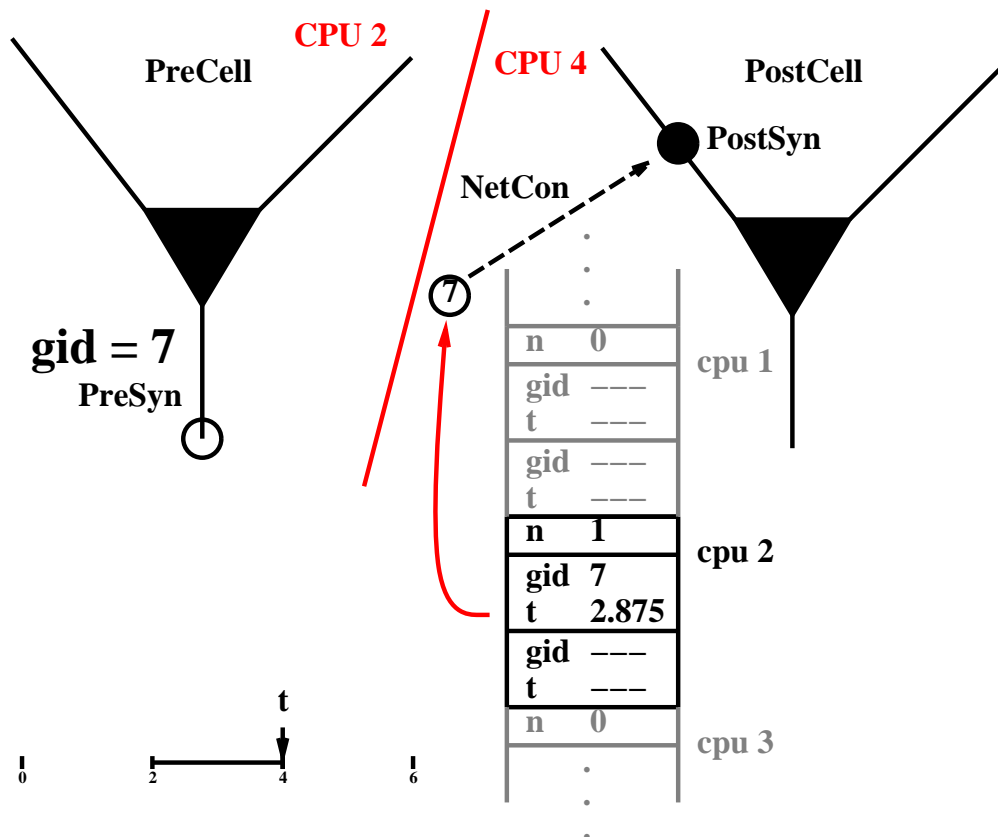
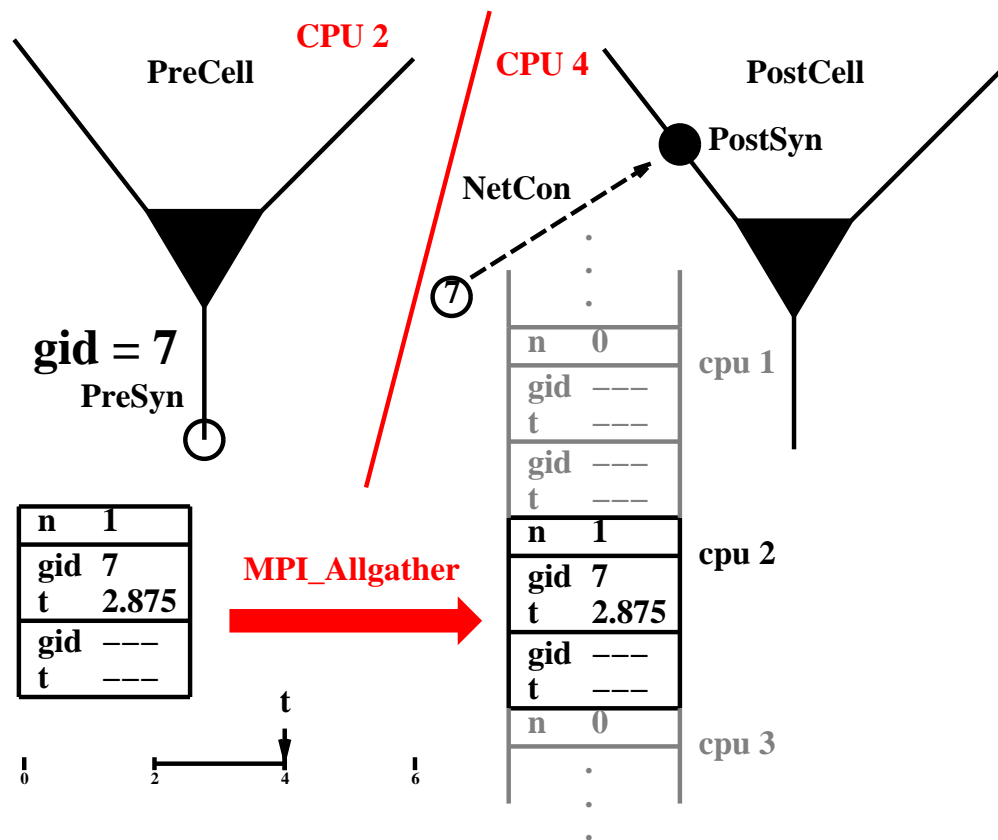
## Run using the idiom

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

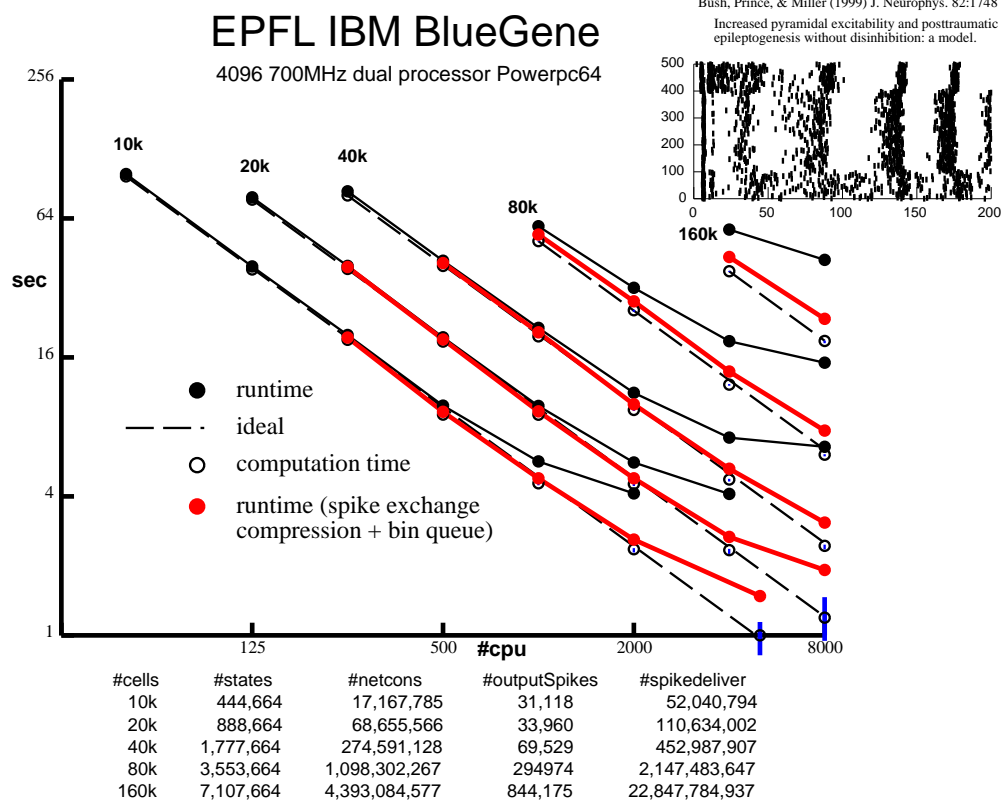
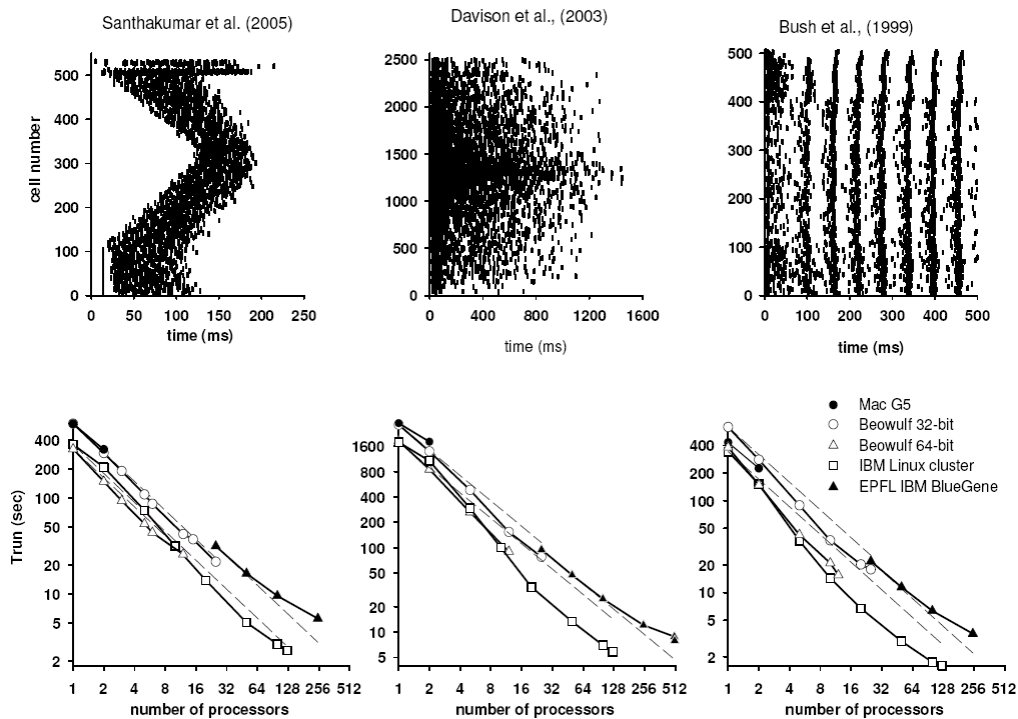
**pc.set\_maxstep()** uses  
**MPI\_Allreduce**  
to determine minimum delay.





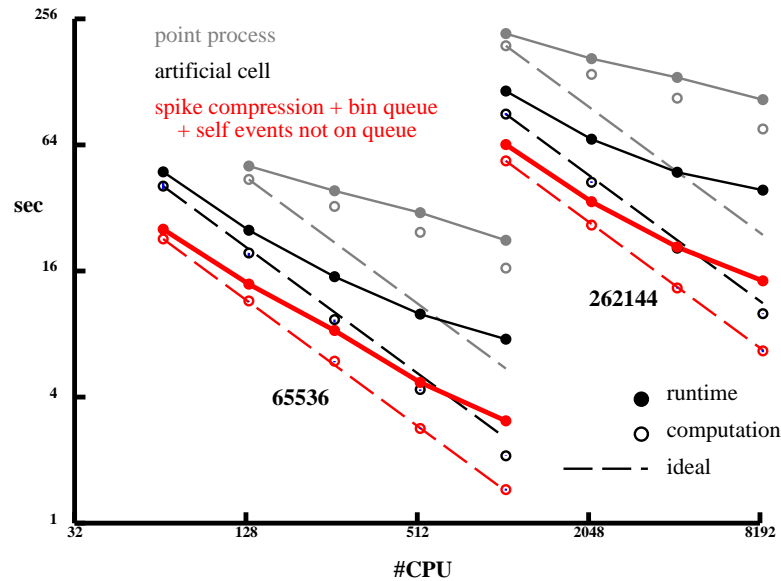


Migliore et al (2006) J. Comput. Neurosci. 21(2):119





## Artificial Spiking Net Performance

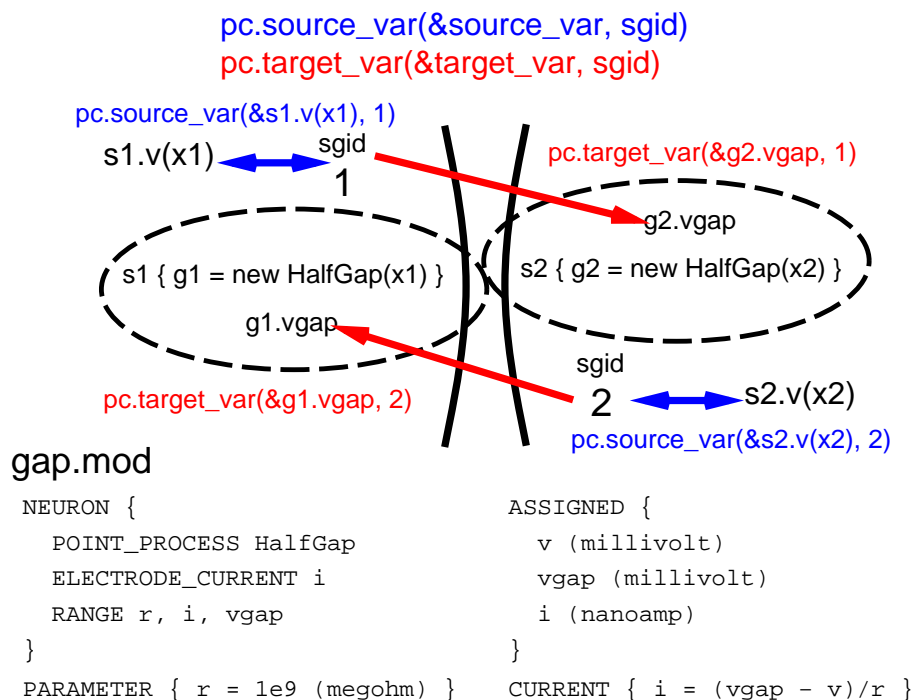


Each cell fires randomly every 10 to 20 ms.  
65K cells, 1000 random connections per cell  
256K cells, 10,000 random connections per cell

tstop = 200(ms)  
delay = 1(ms)  
weight = 0

## Gap Junction Specification

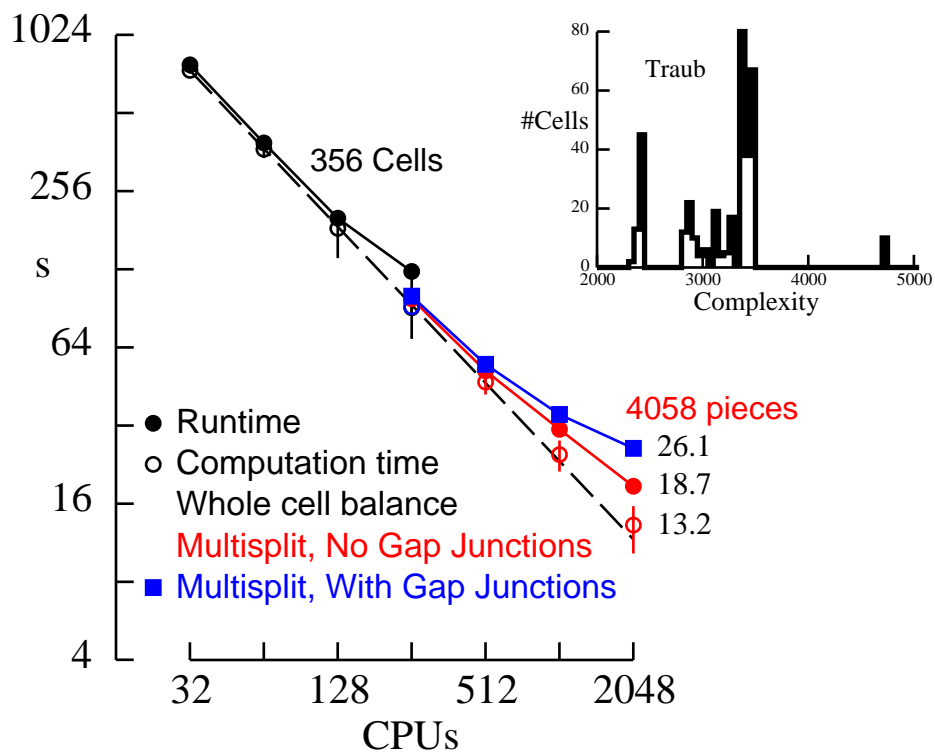
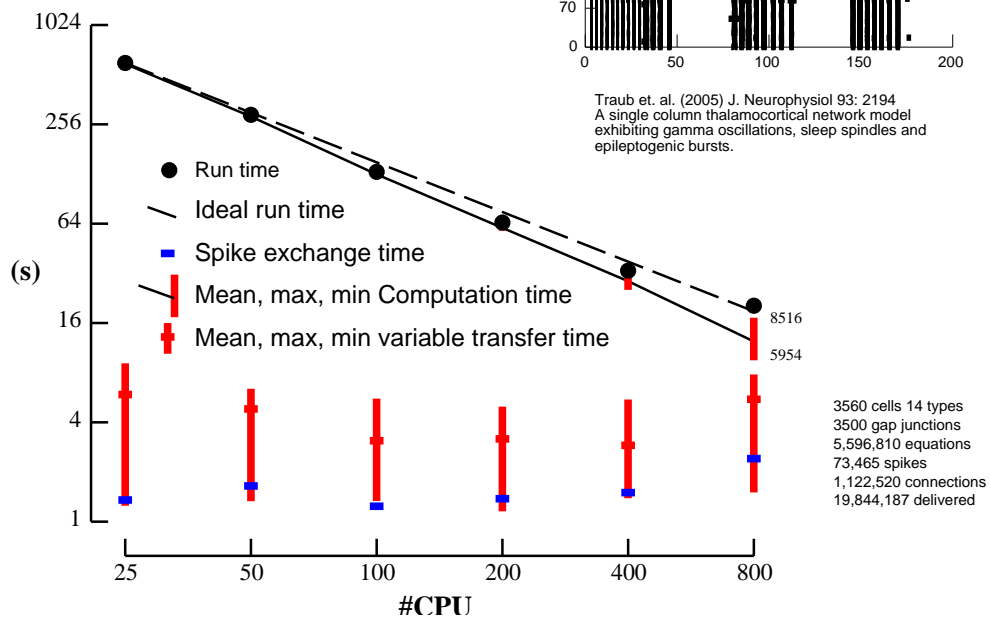
### Continuous Voltage Exchange



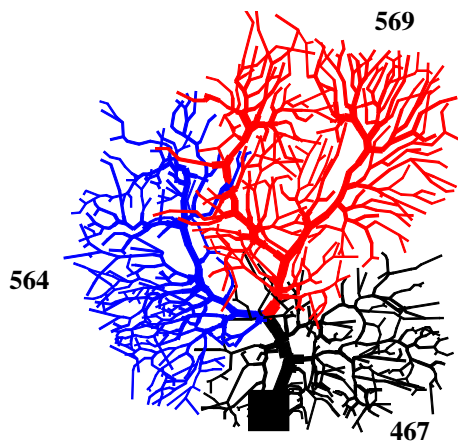
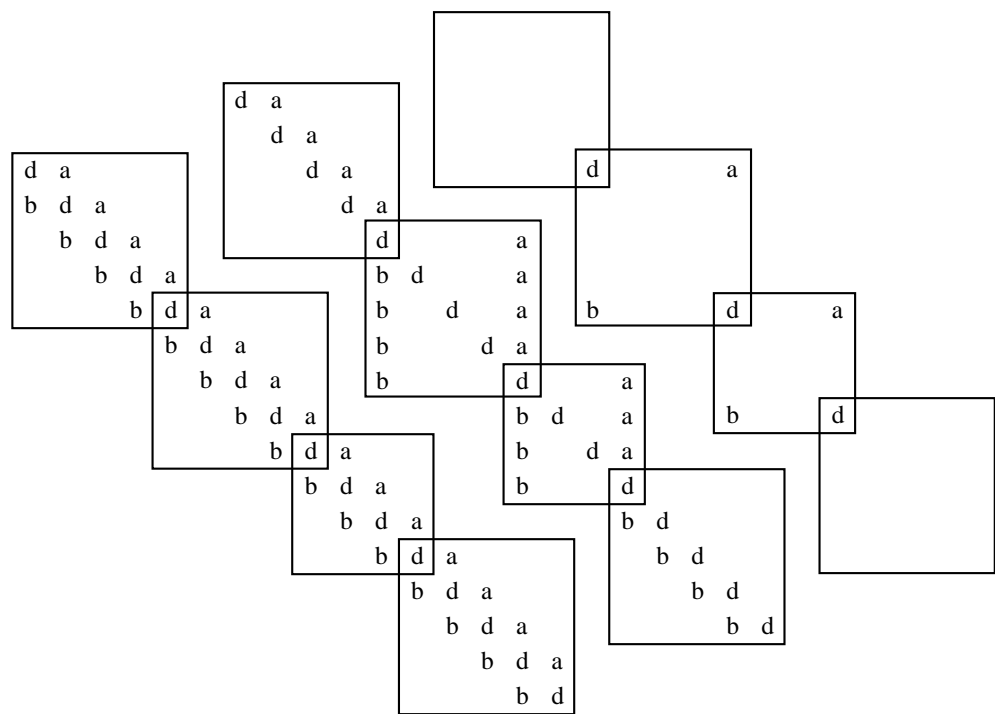
Pittsburgh Supercomputing Center

Bigben Cray XT3

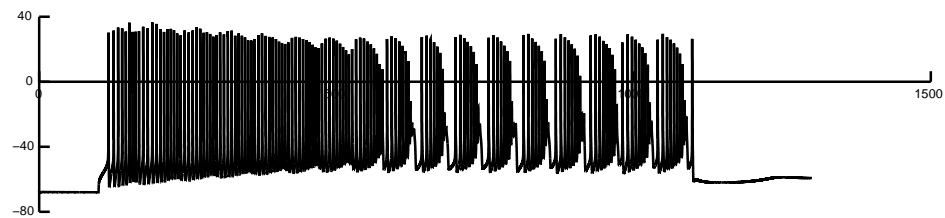
2068 2.4 GHz Opteron Processors



# Multisplit Gaussian Elimination

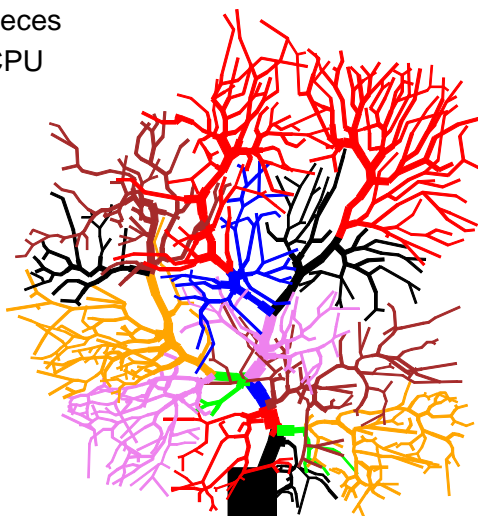


#CPU	Runtime (s)		
1	54.8		
3	22.2	19.5 expected	18.3 ideal

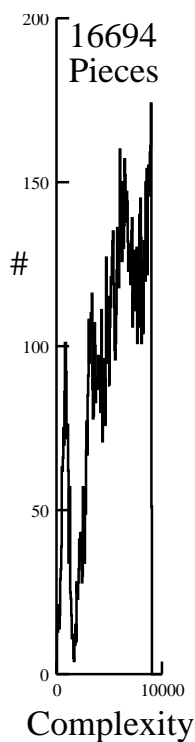
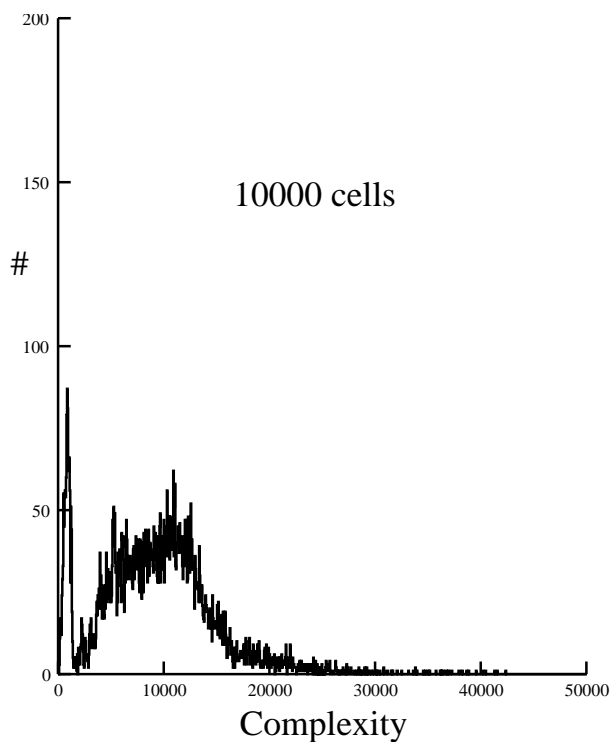
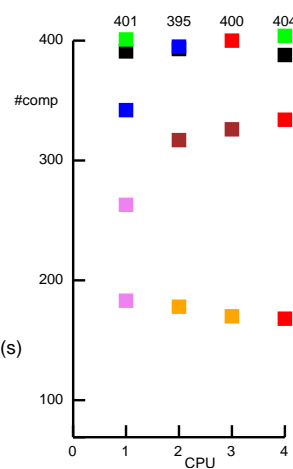
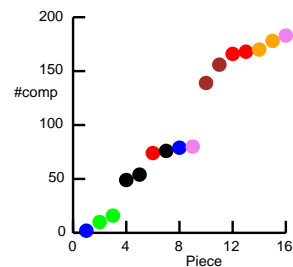


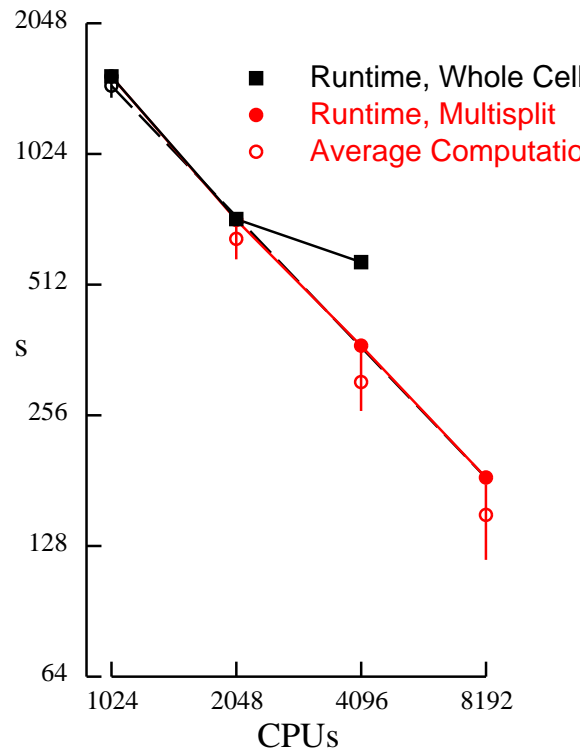
De Schutter & Bower (1994) J. Neurophysiol 71:375  
Ported to NEURON by Jenny Davie, Volker Steuber, and Arnd Roth.

16 Pieces  
4 CPU



CPU	Time (s)			Runtime(s)
	Computation	Exchange		
0	13.82	0.56	16 pieces, 1 cpu	55.0
1	13.35	1.03	wholecell, 1 cpu	56.2
2	13.47	0.90	16 pieces, 4 cpu	14.4
3	13.56	0.82		





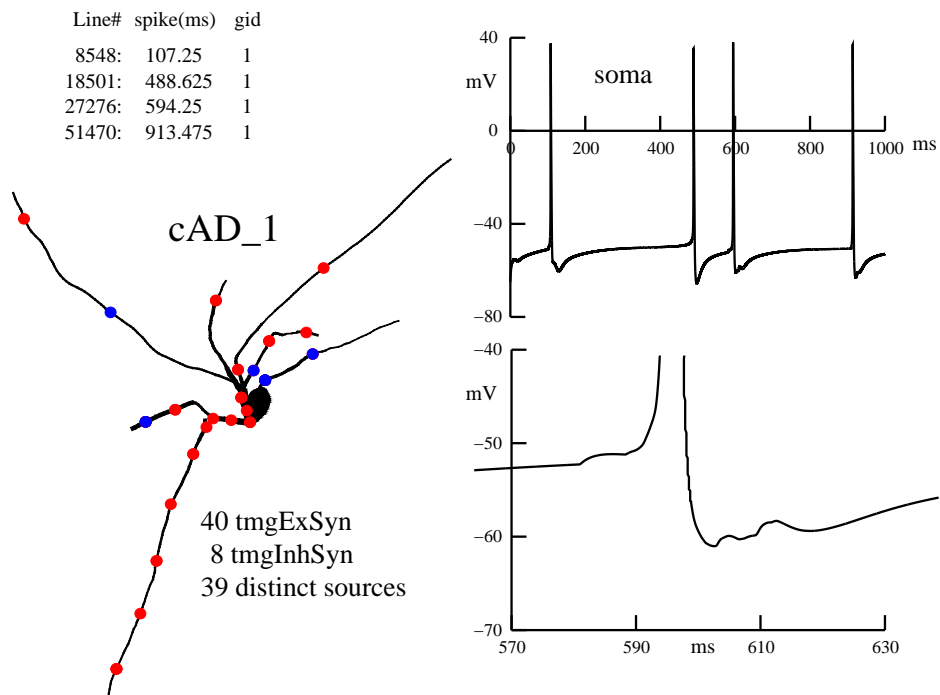
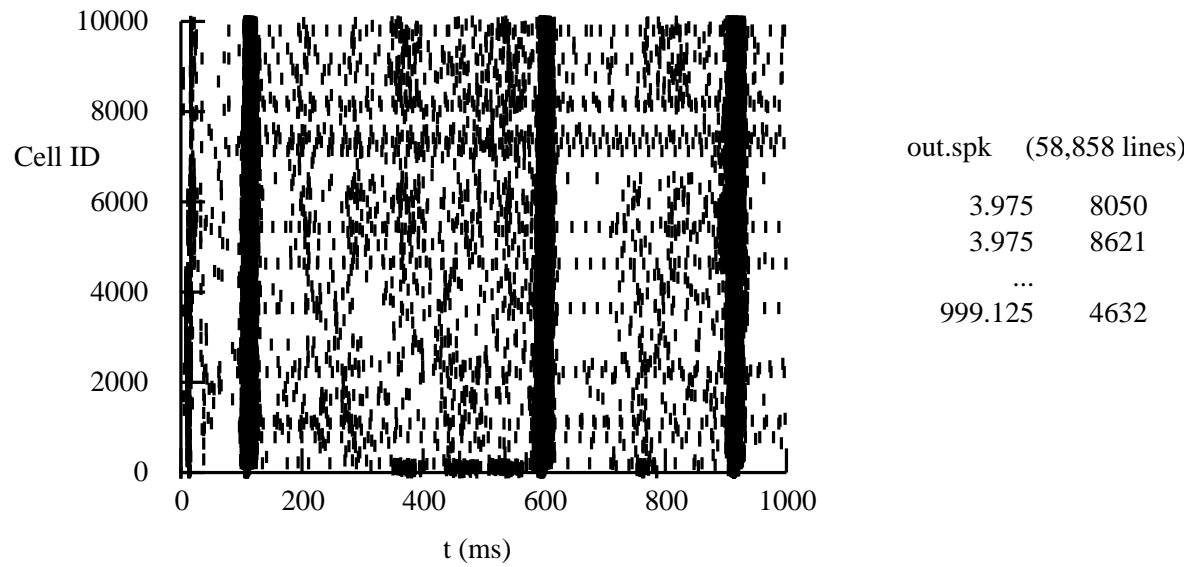
Results must be independent of  
 Number of processors  
 Distribution of cells

What about RANDOM?  
 Reproducible  
 Independent  
 Restartable

Associate a random stream with a cell.

Use cryptographic transformation of several integers.  
 run number  
 stream number (cell gid)  
 stream pick index

# PatternStim



# Debugging

- 1) GID and time of first spike difference.
- 2) All spikes delivered to synapses of that Cell?
- 3) When and what is the first state difference?





## NEURON's tools for Analysis of Electrical Signaling

- Input and transfer impedances
- Voltage transfer ratio

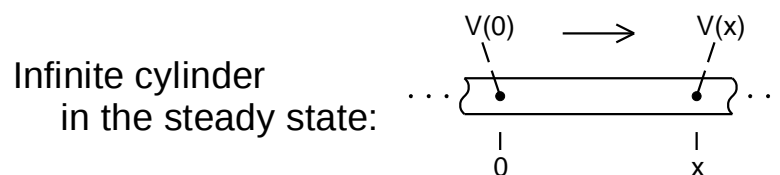
$$V_{downstream}/V_{upstream}$$

- Electrotonic transformation

$$\log(V_{downstream}/V_{upstream})$$

... all as functions of frequency and space

## Classical Cable Theory



$$V(x) = V(0) e^{-x/\lambda}$$

$x$  = physical distance

$\lambda$  = length constant

Classical "electrotonic distance"

$$X = \ln V(0)/V(x) = x/\lambda$$

so attenuation  $A^V(x) = V(0)/V(x) = e^X$

Intuitively simple

## Problems

Neurons are not infinite cylinders.

Attempted fix: reduce dendritic tree to  
finite length equivalent cylinder

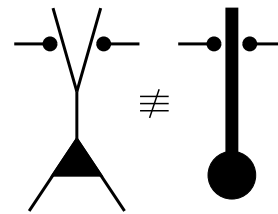
$$A^V(x) = \cosh L_{\text{classical}} / \cosh (L_{\text{classical}} - X)$$

$$L_{\text{classical}} = \text{physical length} / \lambda$$

$$X = x / \lambda$$

## The bad news about the equivalent cylinder approximation

- Neither intuitive nor simple.
- Destroys spatial relationships among synaptic inputs.
- Classical electrotonic distance  $X = x / \lambda$  fosters conceptual error by obscuring the direction-dependence of attenuation in finite structures.



## The good news about the equivalent cylinder approximation

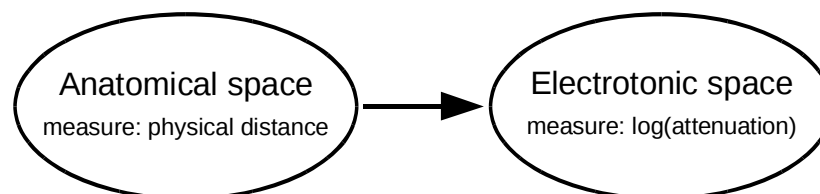
It isn't valid

Property	Assumption	Truth
Dendritic terminations	electrically equidistant from soma	varies widely
Diameters	cylindrical	irregular
Branch points	3/2 power rule $d_p^{3/2} = \sum d_d^{3/2}$	no

## The Electrotonic Transformation

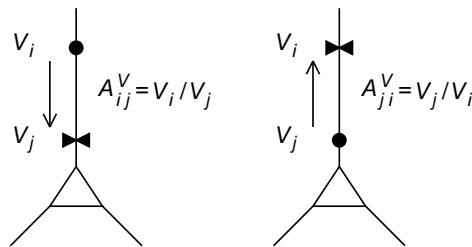
A transformation from anatomical to electrotonic space that

- is intuitive
- is empirically-based
- makes no restrictive assumptions about anatomy



## Foundation: two-port analysis of electrotonus

How well do signals propagate?



Signal transfer is direction-dependent:  $A_{ij}^V \neq A_{ji}^V$

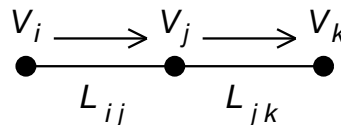
Attenuation identities:  $A_{ij}^V = A_{ji}^I$     $A_{ij}^I = A_{ij}^Q$

## The Electrotonic Transformation

Functional definition of electrotonic distance

**$L = \log(\text{attenuation})$**

- ✓ simple, direct relationship to attenuation
- ✓ direction-dependent:  $L_{ij}^V = \log(A_{ij}^V)$ ,  $L_{ji}^V = \log(A_{ji}^V)$ ,  
and in general  $L_{ij}^V \neq L_{ji}^V$
- ✓ in an infinite cylinder, is identical to classical electrotonic distance
- ✓ additive over a path with constant direction of propagation



$$A_{ik}^V = V_i/V_k = (V_i/V_j) \cdot (V_j/V_k) = A_{ij}^V A_{jk}^V$$

$$\therefore L_{ik} = L_{ij} + L_{jk}$$

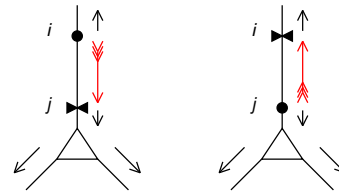
## Using the Electrotonic Transformation

At a frequency of interest

1. compute  $\log(\text{attenuation})$  between a reference point and all other points of interest
2. display results graphically (optional)

A convenient reference point: the soma

Changing the reference point affects only the direction of signal flow on the direct path between the old and new locations.



The attenuation identities give us the transform identities

$$V_{in} = I_{out} = Q_{out} \quad \text{and} \quad V_{out} = I_{in} = Q_{in}$$

## Synaptic location and synaptic efficacy

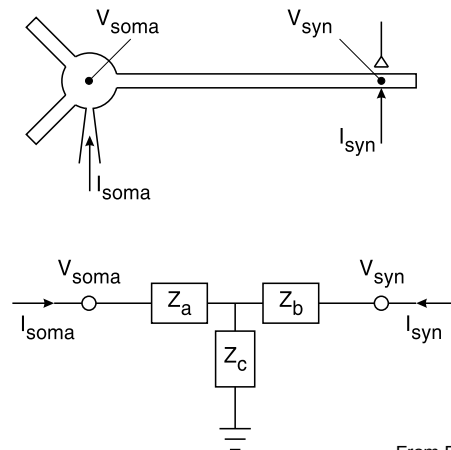
Q: What predicts how PSP amplitude at the soma varies with synaptic location?

A: If synapses act like voltage sources,

$A_{in}^V$  (voltage attenuation from synapse to soma)

or

$k_{syn \rightarrow soma}$  (synapse to soma voltage transfer ratio)



From Fig. 1 in Jaffe & Carnevale 1999

If synapses act like voltage sources,

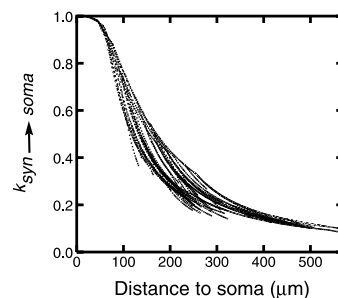
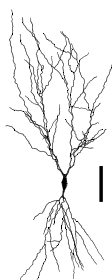
$V_{syn}(t)$  is independent of synaptic location

and

synapse to soma voltage transfer ratio

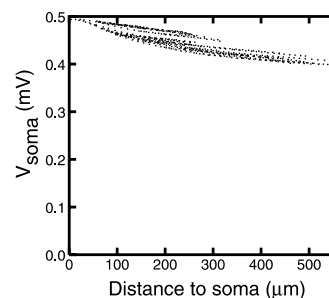
$$k_{syn \rightarrow soma} = 1 / A_{in}^V = Z_c / (Z_b + Z_c)$$

predicts variation of somatic PSP amplitude  
with synaptic location.



Somatic PSP predicted by  
voltage transfer ratio

$$k_{syn \rightarrow soma}$$



Somatic PSP generated by  
conductance-change synapse

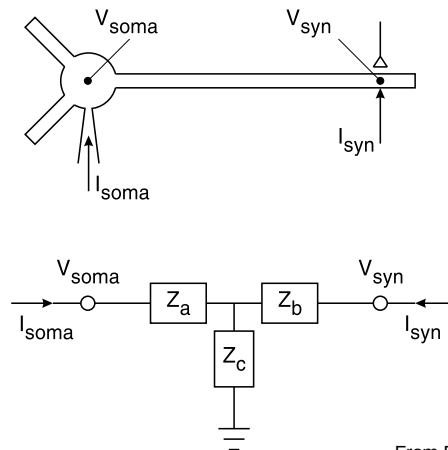
From Fig. 5 in Jaffe & Carnevale 1999

Results:

1.  $k_{syn \rightarrow soma}$  fails to predict the relationship between somatic PSP amplitude and synaptic location.
2. Synapses do not act like voltage sources.

Q1: What do synapses act like?

Q2: What would be a better predictor of the relationship between somatic PSP and synaptic location?



From Fig. 1 in Jaffe & Carnevale 1999

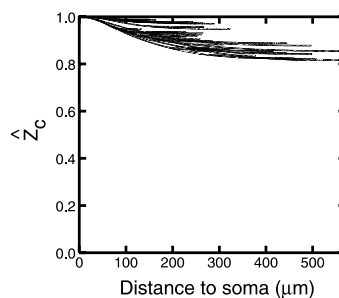
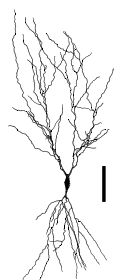
If synapses act like current sources,

$I_{syn}(t)$  is independent of synaptic location

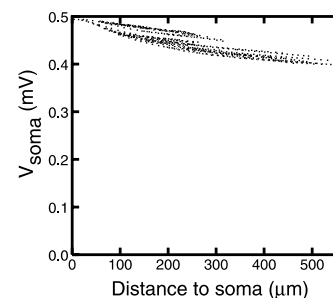
and

transfer impedance  $Z_c$  predicts the variation of  
somatic PSP amplitude with synaptic location.

Let's try it . . .



Somatic PSP predicted by  
transfer impedance  $\hat{Z}_c$

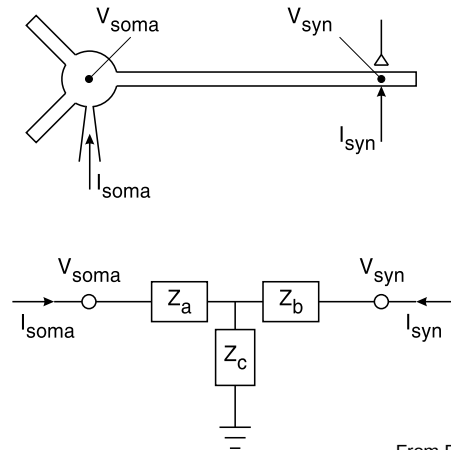


Somatic PSP generated by  
conductance-change synapse

From Fig. 5 in Jaffe & Carnevale 1999

Results:

1. Normalized transfer impedance  $\hat{Z}_c$  predicts the relationship between somatic PSP amplitude and synaptic location.
2. Synapses act like current sources.



From Fig. 1 in Jaffe & Carnevale 1999

Soma to synapse voltage transfer ratio  $k_{soma \rightarrow syn}$  is identical to normalized transfer impedance  $\hat{Z}_C$ .

Proof:  $k_{soma \rightarrow syn} = Z_C / (Z_a + Z_C) = Z_C / Z_N^{soma}$   
 but  $Z_N^{soma}$  is the maximum transfer  $Z$  between any location and the soma.

Therefore  $k_{soma \rightarrow syn} = \hat{Z}_C$



# The Linear Circuit Builder

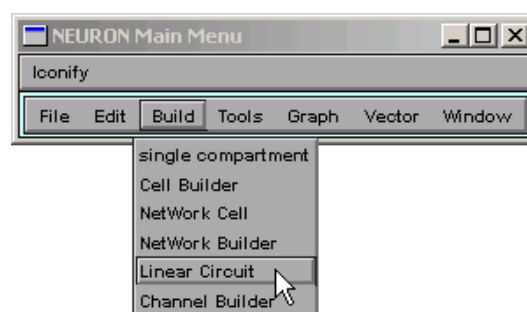
For building models that have linear circuit elements  
and may also involve neurons

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

Potential applications include

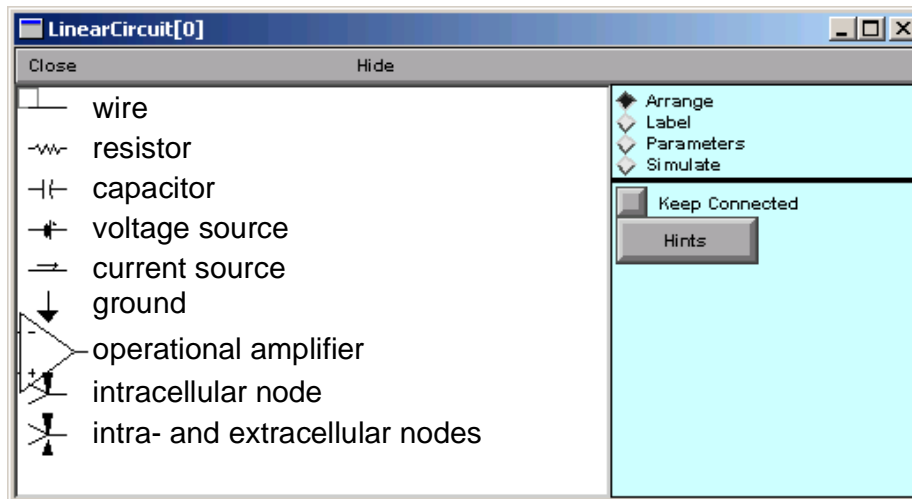
- effects and compensation of electrode R & C
- two-electrode voltage clamp
- ohmic and nonlinear gap junctions

## 1. Bring up a Linear Circuit Builder



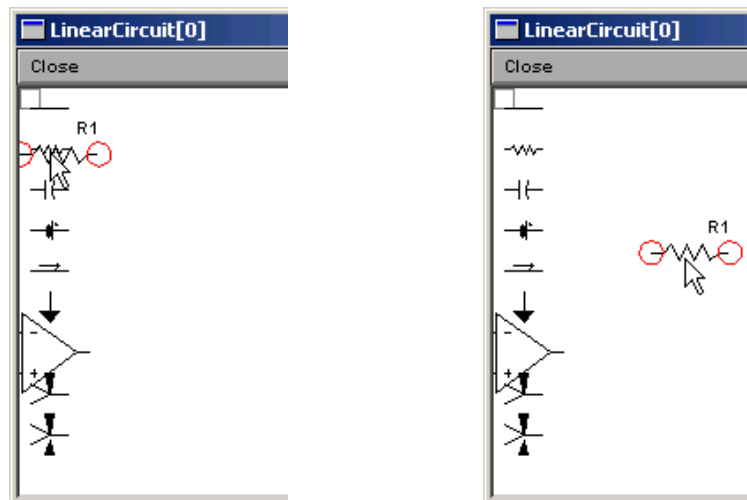
NEURON Main Menu / Build / Linear Circuit

# The Linear Circuit Builder



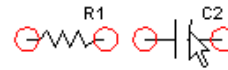
## Arrange: spawn components

Click on palette and drag onto canvas

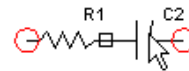


## Arrange: connect components

Click and drag to  
overlap red circles



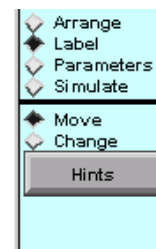
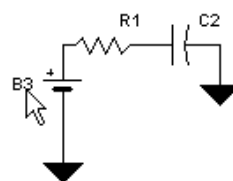
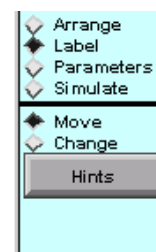
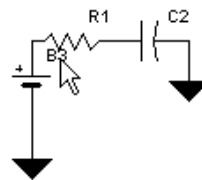
Black square is  
"solder joint"



Pull apart to break connection

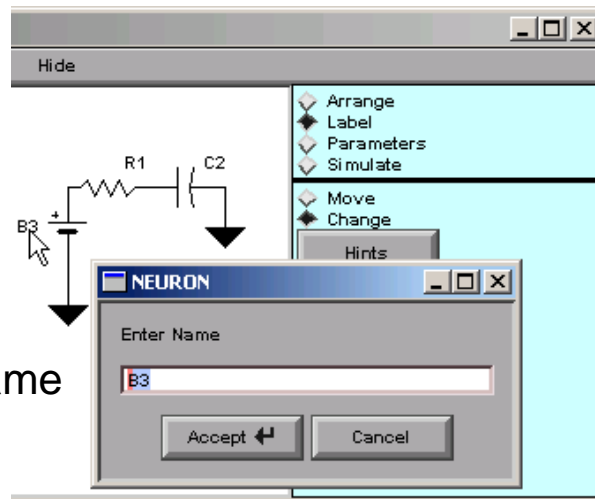
## Label: move labels

Click and drag  
to new location



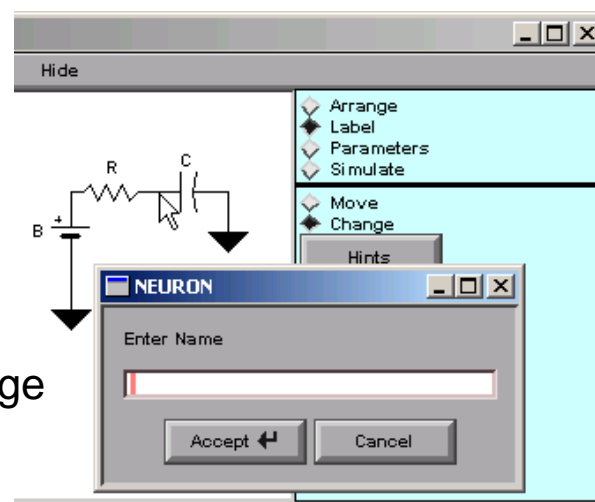
## Label: change labels 1

Click on a label . . .  
 . . . to change its name

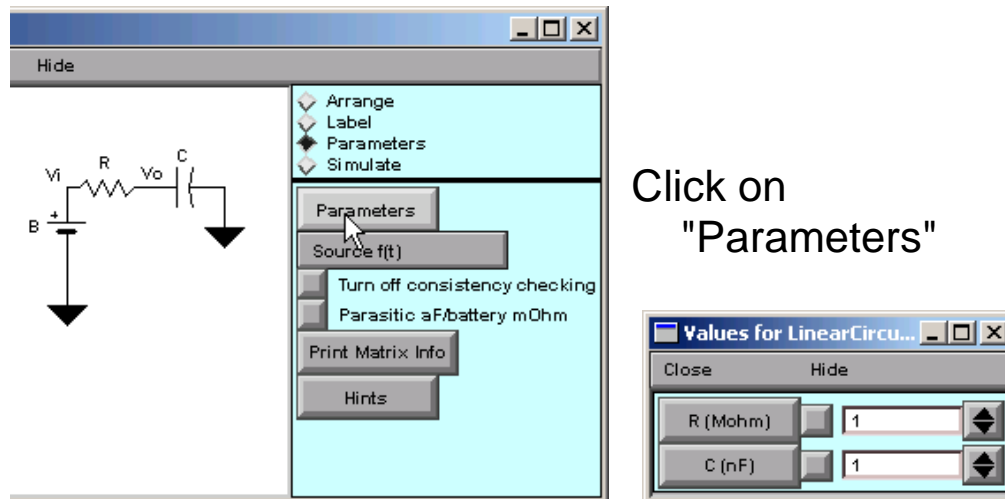


## Label: change labels 2

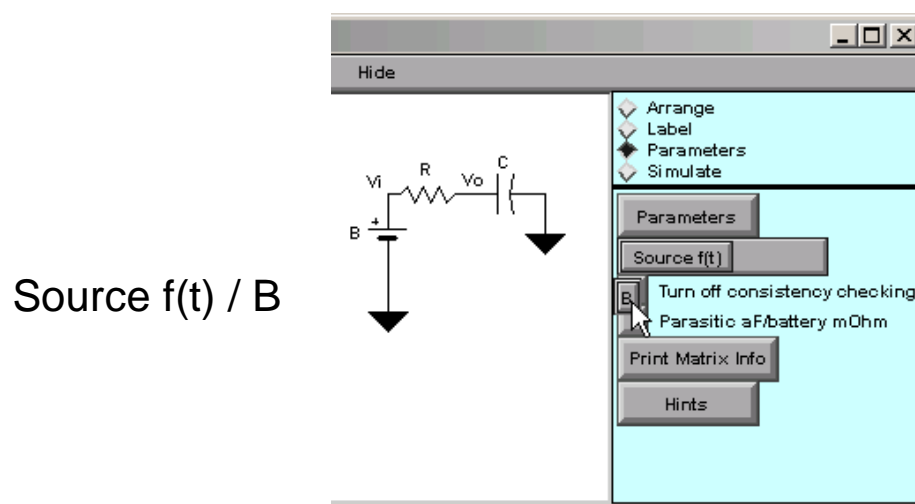
Click on a node . . .  
 . . . to label a voltage



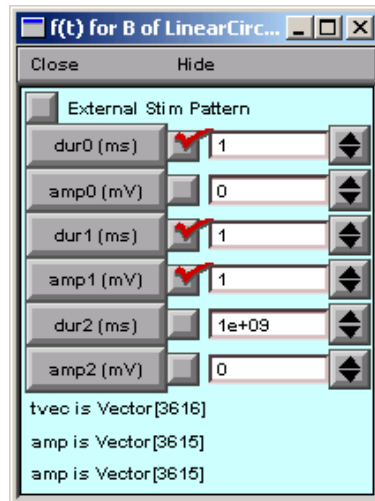
## Parameters: non-source elements



## Parameters: signal sources

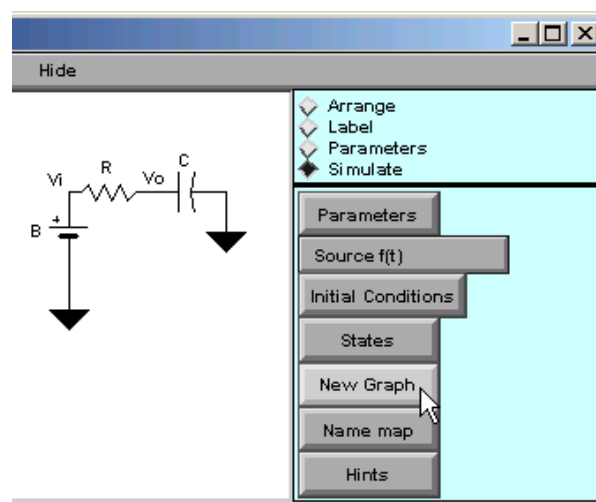


## Parameters: signal sources *continued*



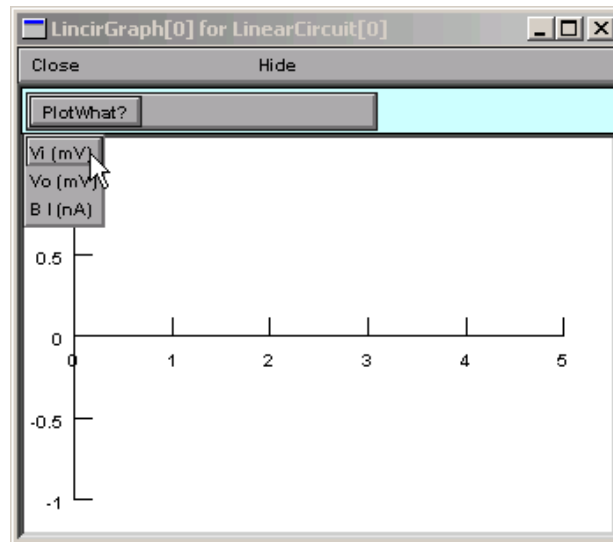
Configured

## Simulate: creating a graph



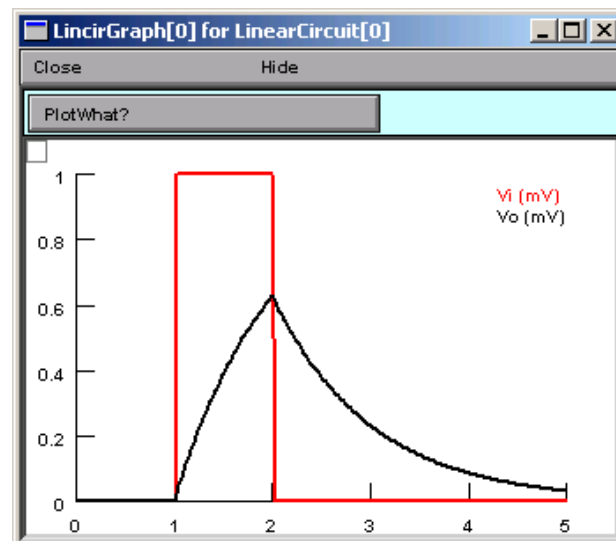
New Graph

## Simulate: specifying what to plot



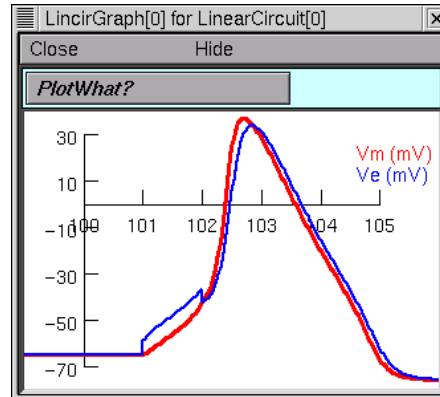
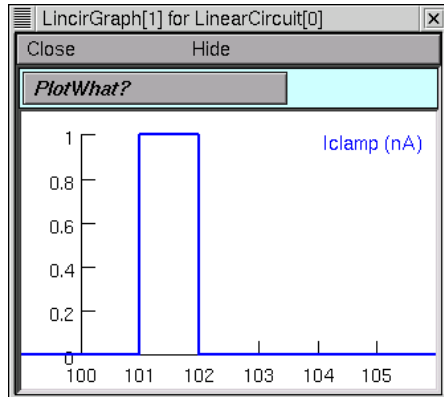
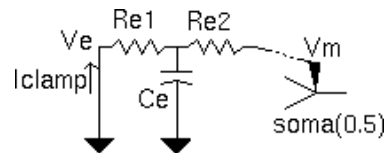
PlotWhat? / *variable\_label*

## Simulate: simulation results

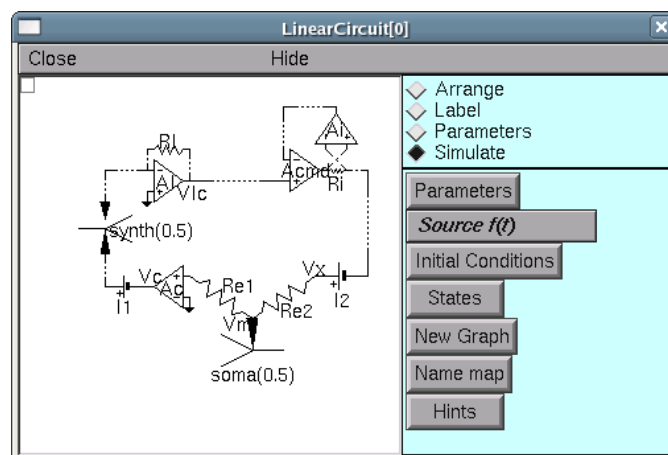


After minor cosmetic changes

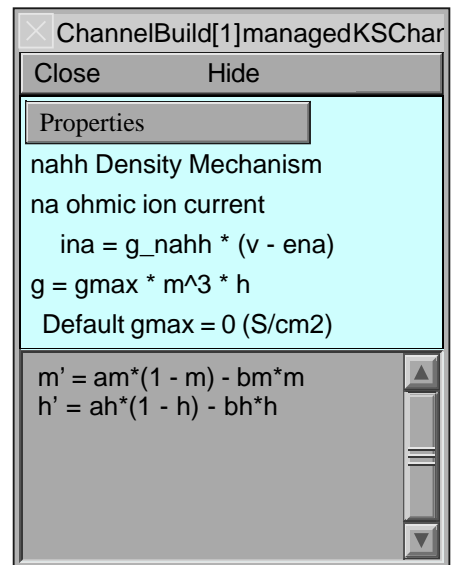
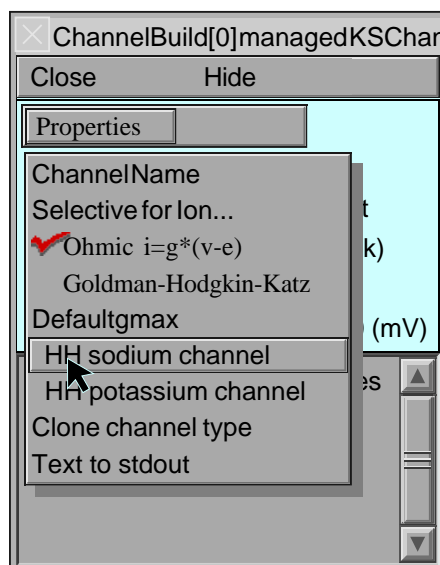
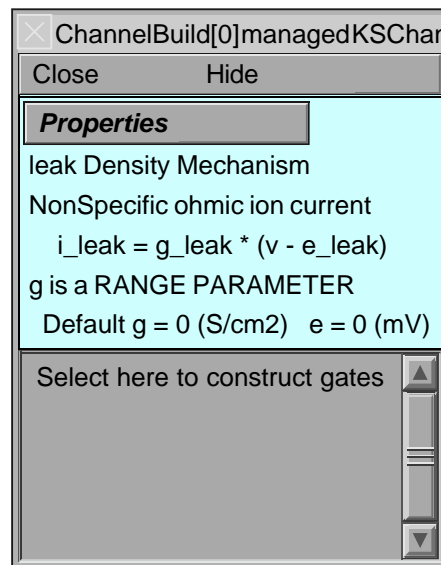
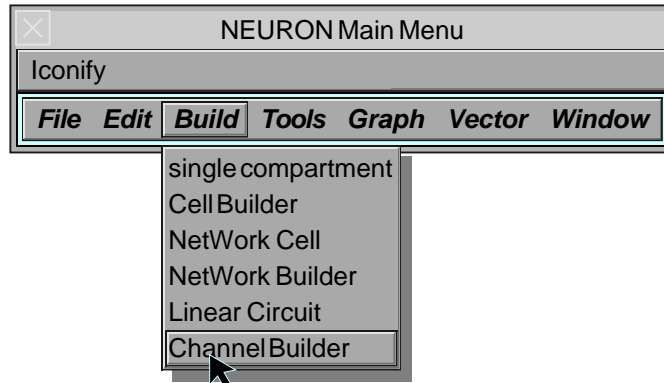
## Patch clamp with electrode R and C



## NEURON demo: dynamic clamp







Default gmax = 0 (S/cm2)

$m' = am*(1 - m) - bm*m$   
 $h' = ah*(1 - h) - bh*h$

ChannelBuildGateGUI[0]forChannelBuild[0]

Close Hide

States Transitions Properties

Select hh state or ks transition to change properties

$m$

$h$

$m^3$

$m' = am*(1 - m) - bm*m$

Power 3

Fractional Conductance

m fraction 1

Adjust Run

$m \leftrightarrow m(a, b)$  (KSTrans[0])

☒ Display inf, tau

$am = A*x/(1 - \exp(-x))$  where  $x = k*(v - d)$

A 1

k 0.1

d -40

$bm = A*\exp(k*(v - d))$

infrm

taum

nahh Density Mechanism

na ohmic ion current

$$ina = g\_nahh * (v - ena)$$

$$g = gmax * m^3 * h$$

Default gmax = 0 (S/cm2)

$$m' = am*(1 - m) - bm*m$$

$$am = 1*x/(1 - \exp(-x)) \text{ where } x = 0.1*(v + 40) \quad (\text{Vector}[7])$$

$$bm = 4*\exp(-0.05556*(v + 65)) \quad (\text{Vector}[8])$$

$$h' = ah*(1 - h) - bh*h \quad (\text{KSTrans}[1])$$

$$ah = 0.07*\exp(-0.05*(v + 65)) \quad (\text{Vector}[11])$$

$$bh = 1/(1 + \exp(-0.1*(-35 - v))) \quad (\text{Vector}[12])$$

ChannelBuild[0]managedKSChar

Close Hide

Properties

ChannelName

Selective for Ion...

☒ Ohmic  $i = g*(v - e)$

Goldman-Hodgkin-Katz

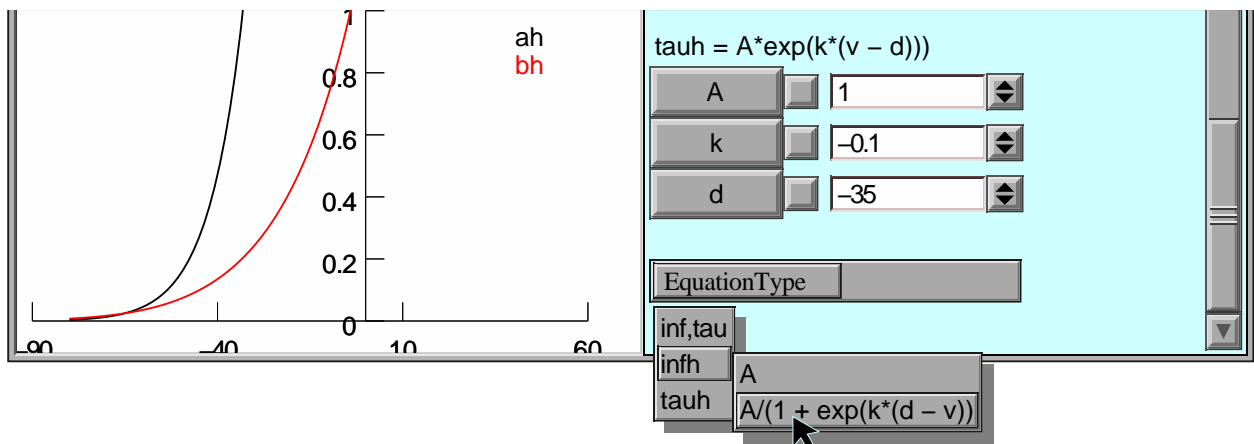
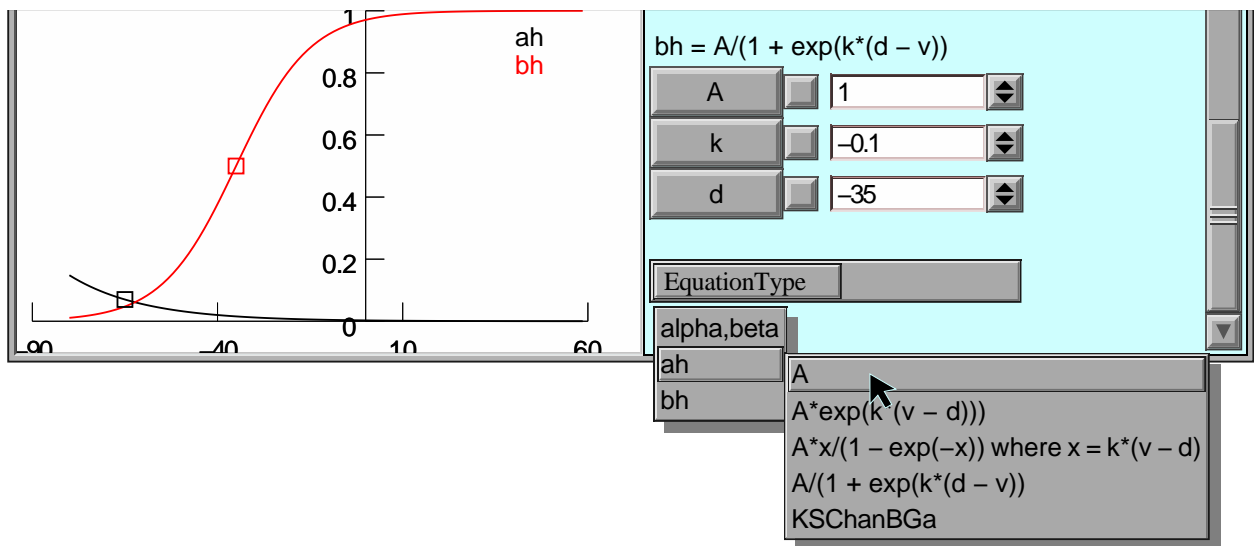
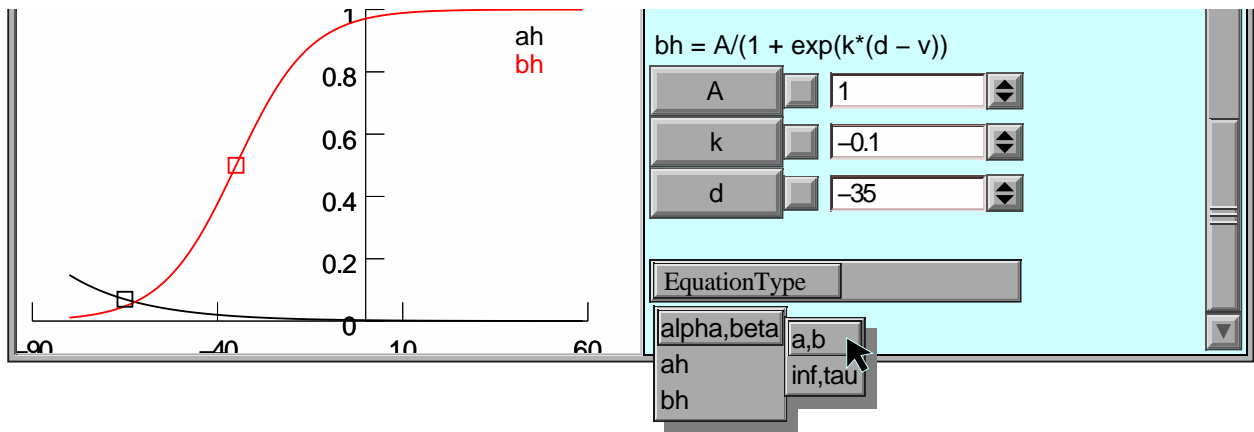
Defaultgmax

HH sodium channel

HH potassium channel

Clone channel type

Text to stdout



ChannelBuildGateGUI[0]forChannelBuild[0]

Close Hide

◆ States ◆ Transitions ◆ Properties

no gate selected

Drag new state from left. Drag off canvas to delete

O  
C

C C2

◆ Adjust ☐ Run

no KSTrans selected

1  
0.8  
0.6  
0.4  
0.2  
0

ChannelBuildGateGUI[0]forChannelBuild[0]

Close Hide

◆ States ◆ Transitions ◆ Properties

no gate selected

New transition pair: select source and drag to target

C  $\xleftrightarrow{v}$  C2

O

◆ Adjust ☐ Run

no KSTrans selected

1  
0.8  
0.6  
0.4  
0.2  
0

ChannelBuildGateGUI[0]forChannelBuild[0]

Close Hide

States Transitions **Properties**

Select hh state or ks transition to change properties

$$C \xrightleftharpoons{v} C2 \xrightleftharpoons{v} O$$

O

O: 3 state, 2 transitions

Power 1

Fractional Conductance

C2 fraction 0

O fraction 1

Adjust Run

1  
0.8  
0.6  
0.4  
0.2  
0

aC2O  
bC2O

bC2O = A

A 0

EquationType

alpha,beta  
aC2O  
bC2O

a,b  
inf,tau  
nai  
nao  
ki  
ko  
cai  
cao

Close

Hide

States

Transitions

Properties

Select hh state or ks transition to change properties

$$C \xrightleftharpoons{v} C2 \xrightleftharpoons{cai} O$$

O

O: 3 state, 2 transitions

Power

1

Fractional Conductance

C2 fraction

0

O fraction

1

Adjust

Run

1

0.8

0.6

0.4

0.2

0

aC2O

bC2O

C2 + cai <--> O (a, b) (KSTrans[9])

Display inf, tau

aC2O = A

ChannelBuild[0]managedKSChar

Close

Hide

Properties

kca Density Mechanism

k ohmic ion current

ik = g\_kca \* (v - ek)

g = gmax \* O

Default gmax = 0 (S/cm2)

O: 3 state, 2 transitions

The screenshot displays two windows from the NEURON ChannelBuild environment.

The top window, titled "ChannelBuild[0]managedKSChan[0]", shows the "Properties" tab. It contains the following text:

- leak Density Mechanism
- NonSpecific ohmic ion current
- $i_{\text{leak}} = g_{\text{leak}} * (v - e_{\text{leak}})$
- $g = g_{\text{max}} * O * O2 * O3$
- Default  $g_{\text{max}} = 0$  (S/cm2)  $e = 0$  (mV)
- O: 3 state, 2 transitions
- O2: 2 state, 1 transitions
- $O3' = aO3 * (1 - O3) - bO3 * O3$

The bottom window, titled "ChannelBuildGateGUI[0]forChannelBuild[0]", shows the "States" tab. It displays a state transition diagram with states O, C, C2, O2, and C3. The transitions are labeled with 'v'. The state O3 is highlighted in red. A dialog box titled "nrniv" is open, showing "Change state name" with "O3" in the input field and "Accept" and "Cancel" buttons.

At the bottom of the bottom window, there are "Adjust" and "Run" buttons, and a "no KSTrans selected" message.

The screenshot displays the "ChannelBuild[0]managedKSChar" window. It shows a list of channel types under the "Properties" tab. The list includes:

- ChannelName
- Selective for Ion...
- ☒ Ohmic  $i = g * (v - e)$
- Goldman-Hodgkin-Katz
- Default  $g_{\text{max}}$
- HH sodium channel
- HH potassium channel
- Clone channel type
- Text to stdout

A context menu is open over the list, showing options: "NonSpecific", "na", "k", and "Create new type".

