

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

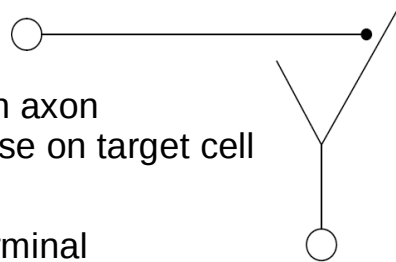
Physical system:

Presynaptic neuron with axon
that projects to synapse on target cell

Conceptual model:

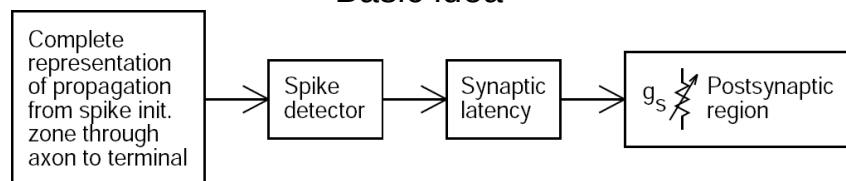
Spike in presynaptic terminal
triggers transmitter release;
presynaptic details unimportant

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

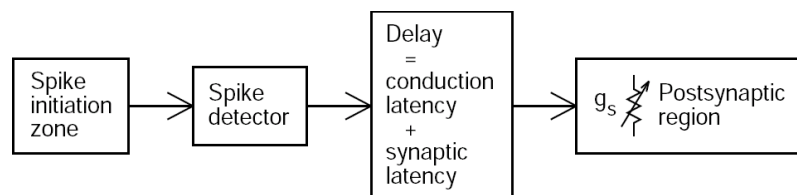


Spike-triggered transmission: computational implementation

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

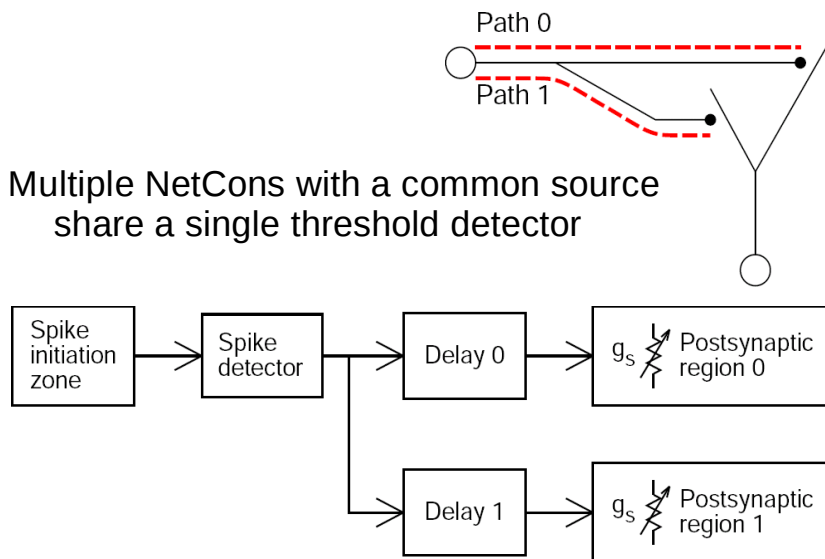
```
threshold = 10
delay = 1 // must be >= 0
weight = 0
```

NMODL specification of synaptic mechanism

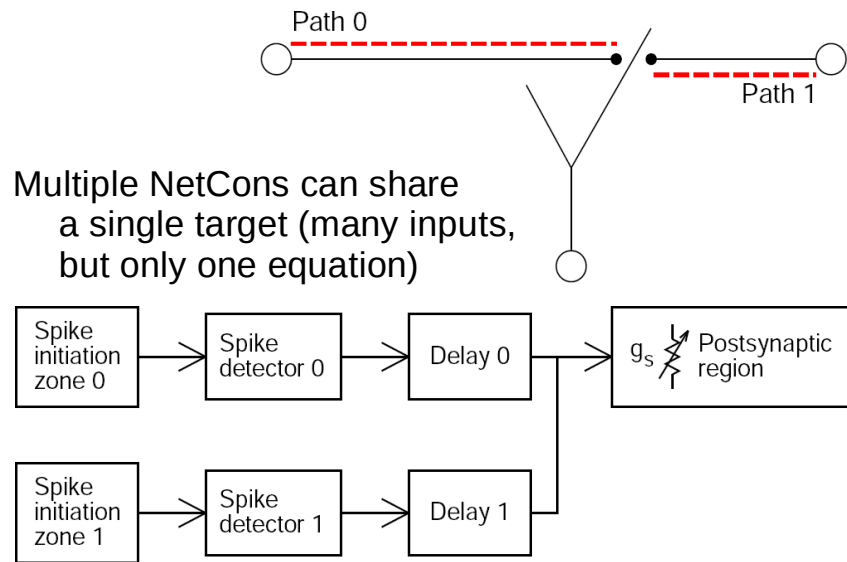
```
NET_RECEIVE(weight(microsiemens)) {
    . . .
}
```

Efficient divergence

Multiple NetCons with a common source
share a single threshold detector



Efficient convergence



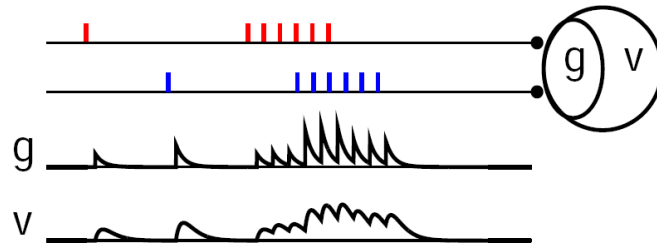
Example: g_s with fast rise and exponential decay

```

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*

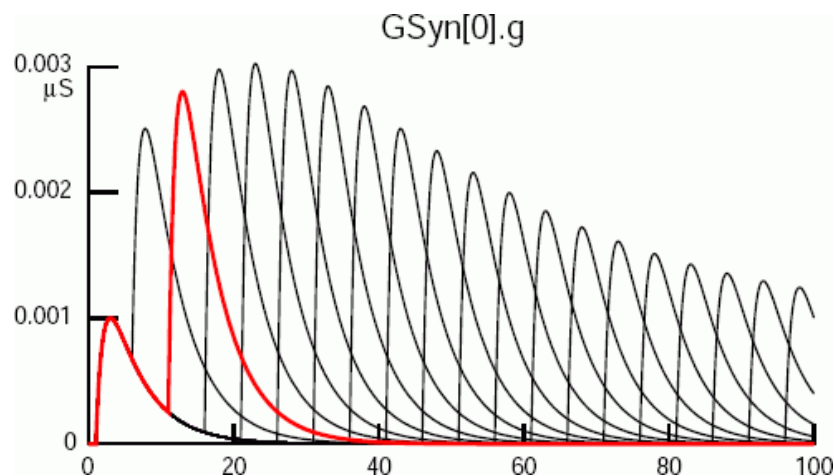


```

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

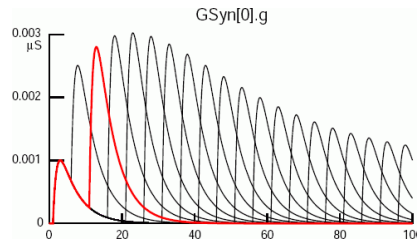


Use-dependent synaptic plasticity *continued*

```

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

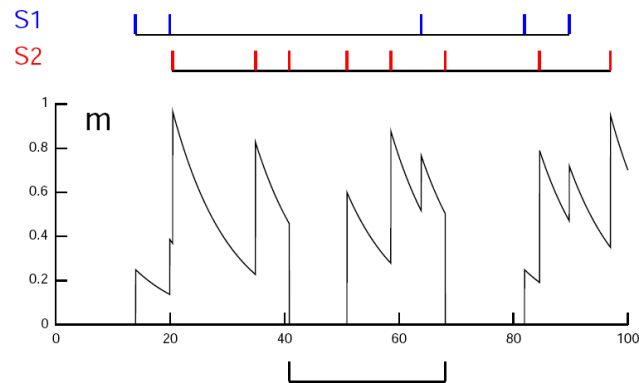
Event-driven simulation run time is

proportional to # of received events

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

Hybrid networks

Example: leaky integrate and fire model

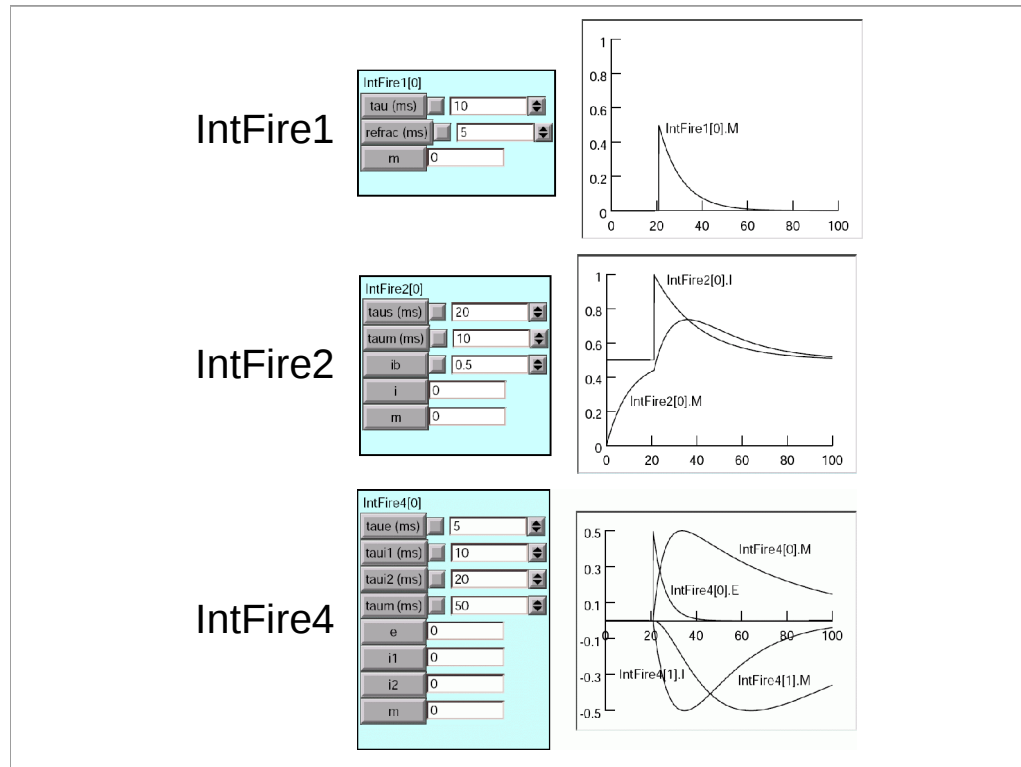


Leaky integrate and fire model *continued*

```

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  
}
```


Synchronizing network parameters

Close Hide

Number of all to all cells 10

All to all connection weight ☒ 0

Delay (ms) ☒ 0

Cell time constant (ms) 10

Lowest natural interval 10

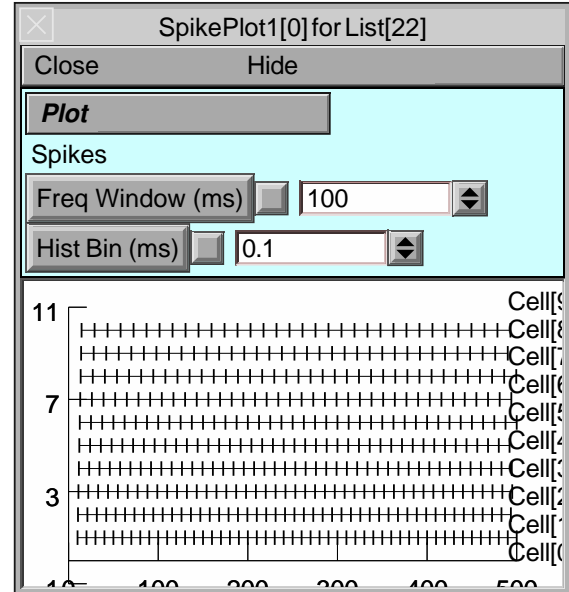
Highest natural interval 15

NEURONDemonstrations

Close Hide

Synchronizing net (artificial cells)

- ☐ Patch: HH
- ☐ Stylized
- ☐ Pyramidal
- ☐ Release
- ☒ Synchronizing net (artificial cells)
- ☒ LinearCircuit:DynamicClamp
- ☐ Stochastic Single Channels: HH
- ☐ No model



Synchronizing network parameters

Close Hide

Number of all to all cells 10

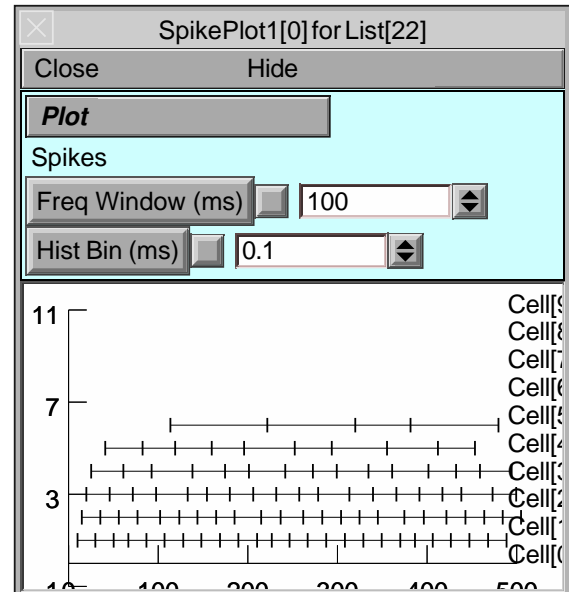
All to all connection weight -0.3

Delay (ms) ☒ 0

Cell time constant (ms) 10

Lowest natural interval 10

Highest natural interval 15



Synchronizing network parameters

Close Hide

Number of all to all cells

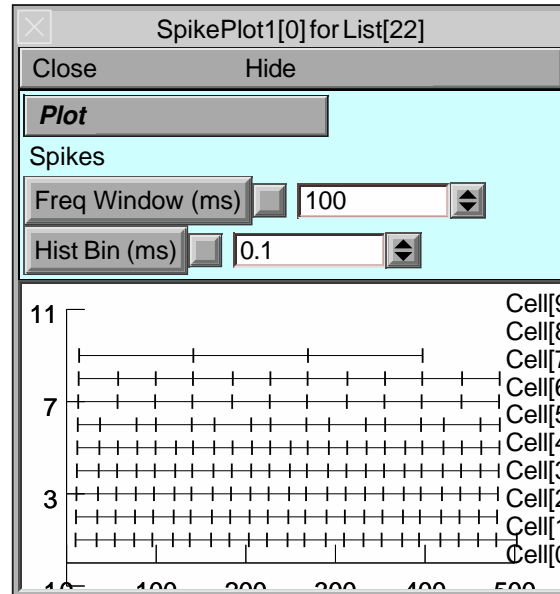
All to all connection weight

Delay (ms)

Cell time constant (ms)

Lowest natural interval

Highest natural interval



Synchronizing network parameters

Close Hide

Number of all to all cells

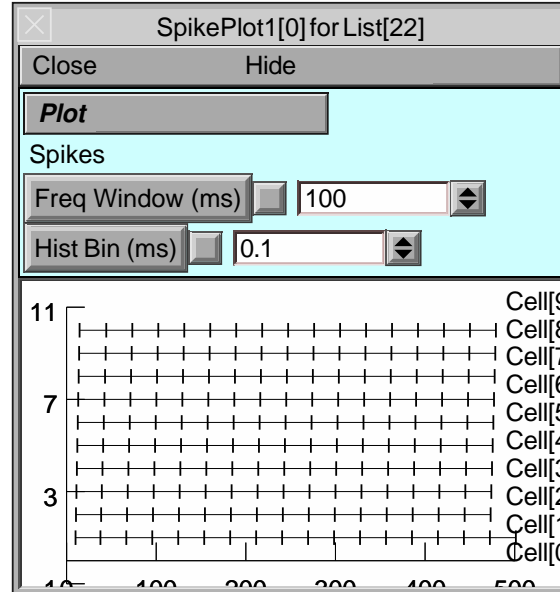
All to all connection weight

Delay (ms) ☒

Cell time constant (ms)

Lowest natural interval

Highest natural interval



```

NEURON {
  ARTIFICIAL_CELL IntervalFire
  RANGE tau, m, invl
  : m plays the role of voltage
}

: dm/dt = (minf - m)/tau
: input event adds w to m
: when m = 1, or event makes m >= 1 cell fires
: minf is calculated so that the natural
: interval between spikes is invl

INITIAL {
  minf = 1/(1 - exp(-invl/tau))
  m = 0
  t0 = t
  net_send(firetime(), 1)
}

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

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

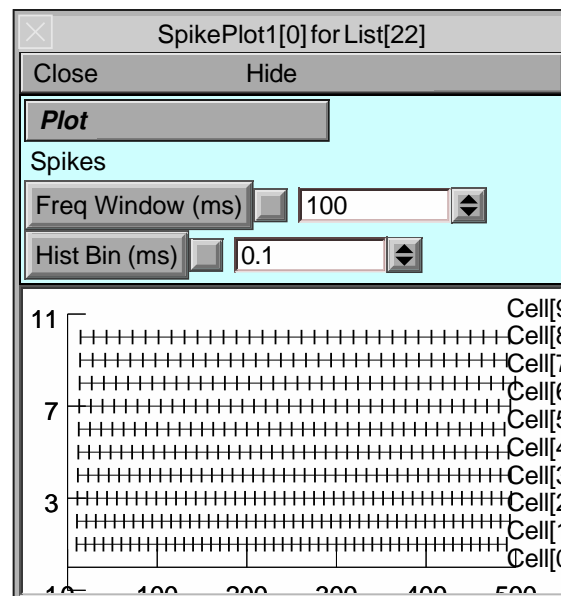
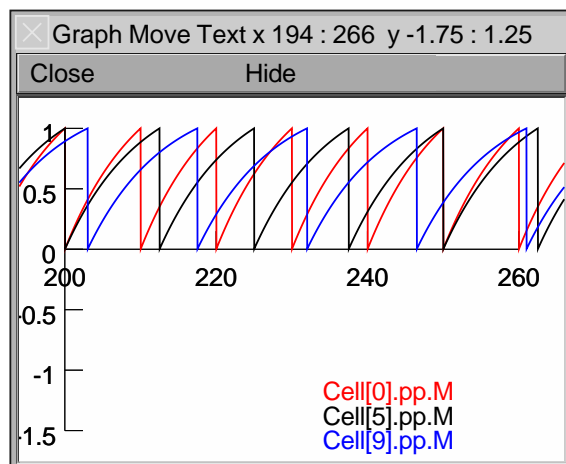
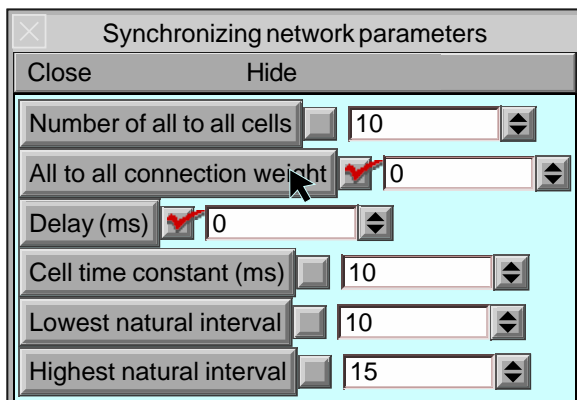
```

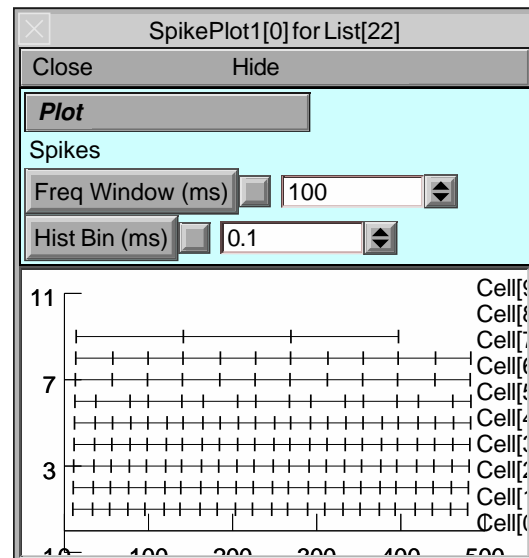
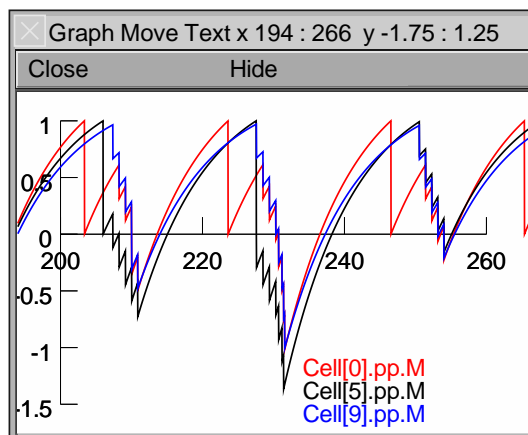
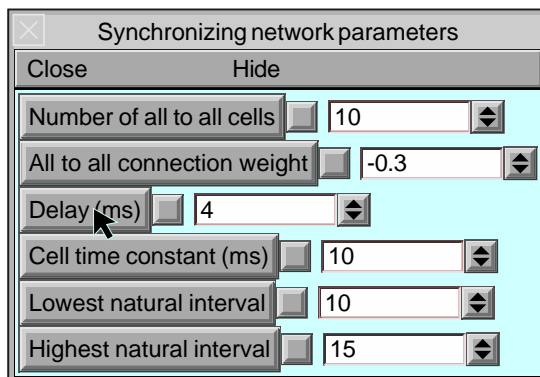
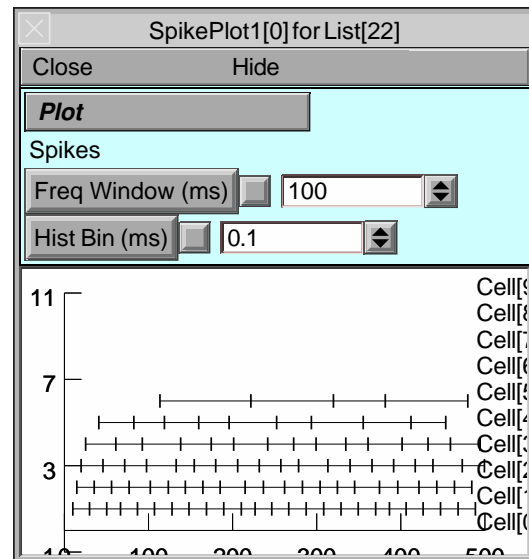
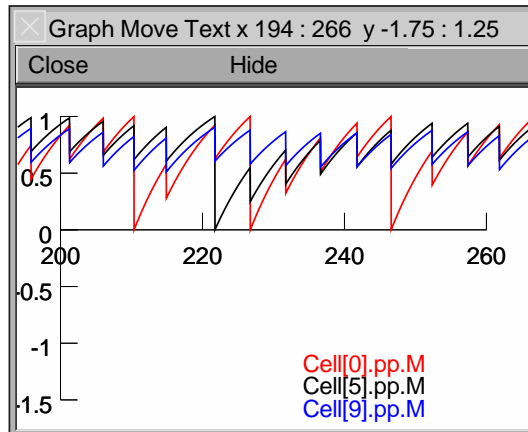
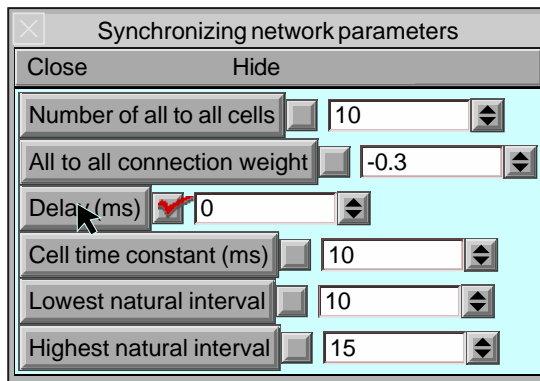
```

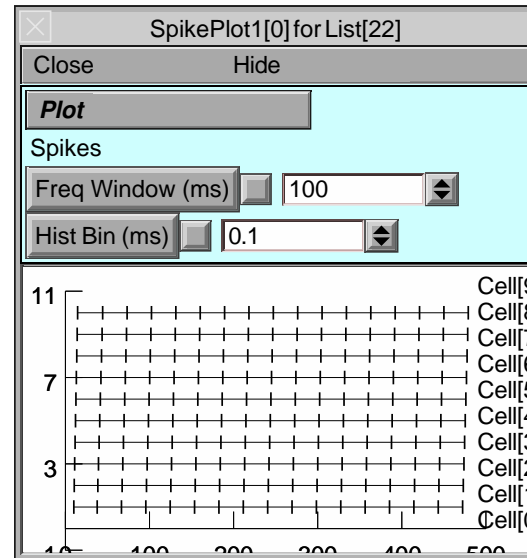
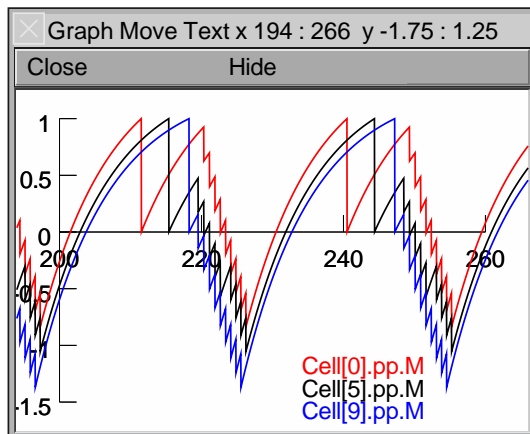
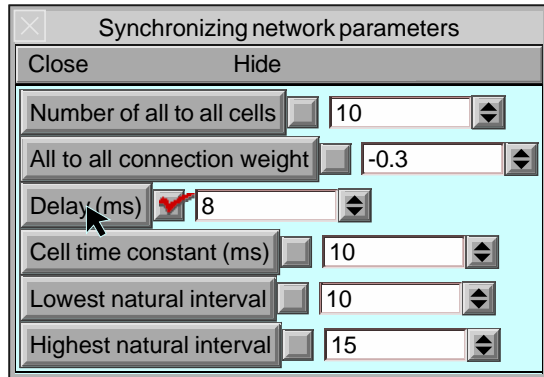
objref cells, nclist
{cells = new List() nclist = new List()}

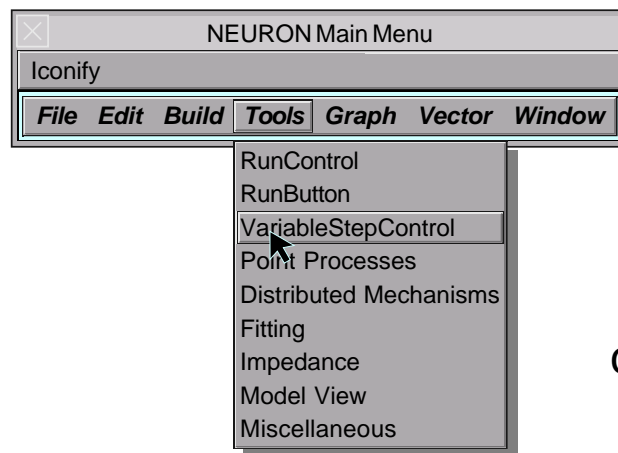
proc createnet() {local i, j
  ncell = $1
  nclist.remove_all()
  cells.remove_all()
  for i=0, ncell-1 {
    cell_append(new Cell(), i, 0, 0)
  }
  for i=0, ncell-1 for j=0, ncell-1 if (i != j) {
    nc_append(i, j, -1, 0, 1)
  }
}

```

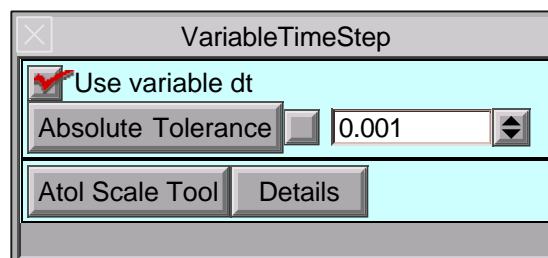
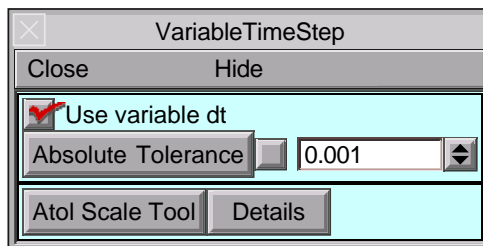








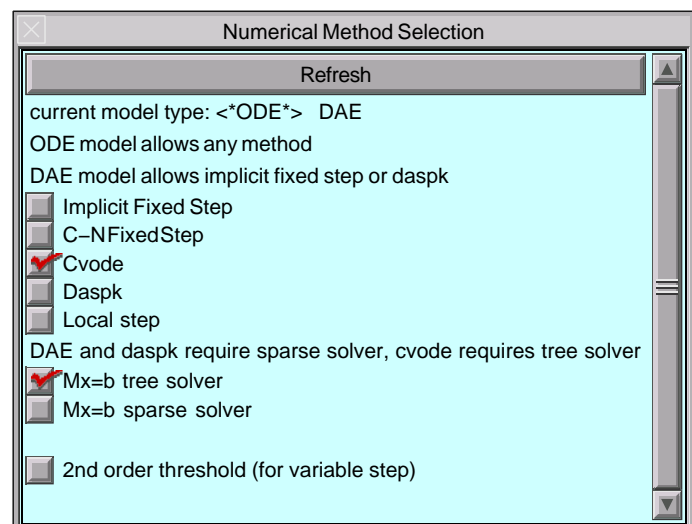
`cvode_active(1)`

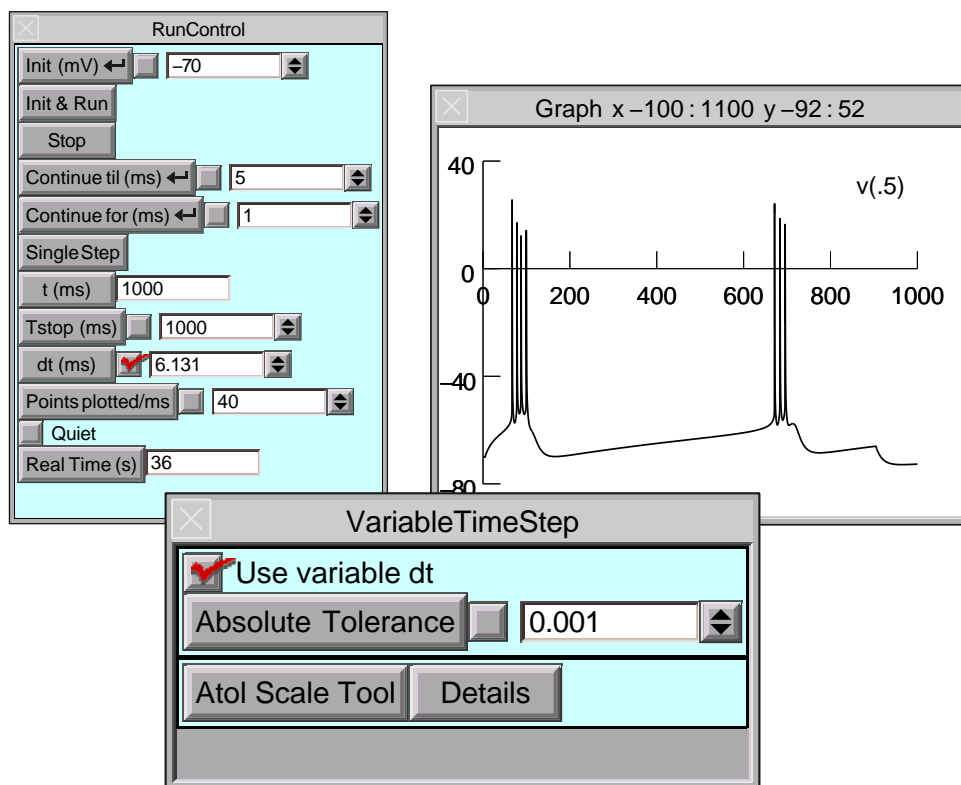
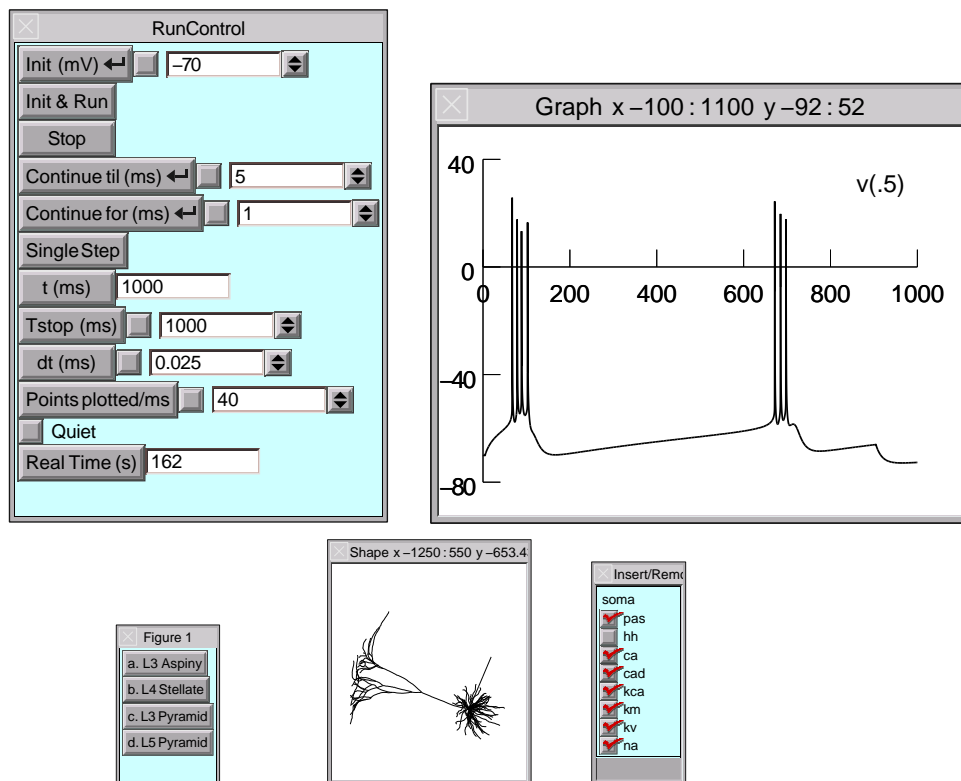


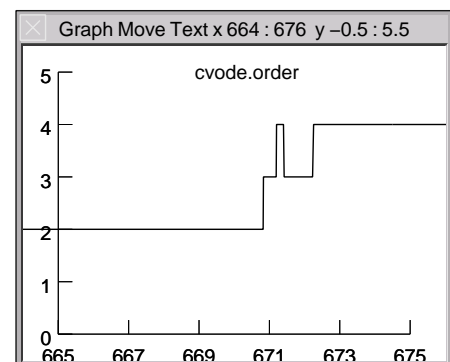
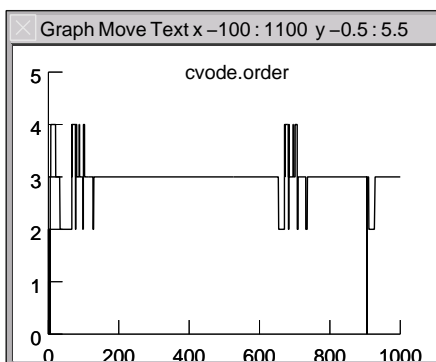
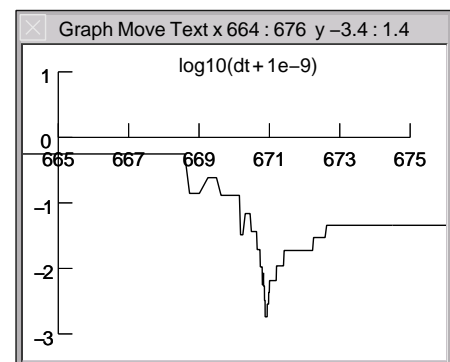
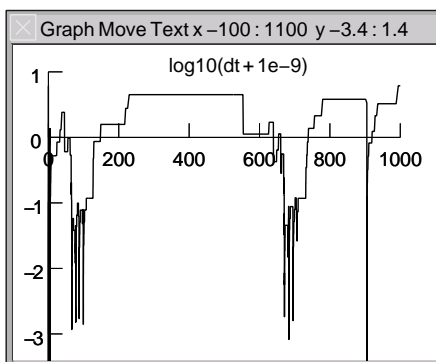
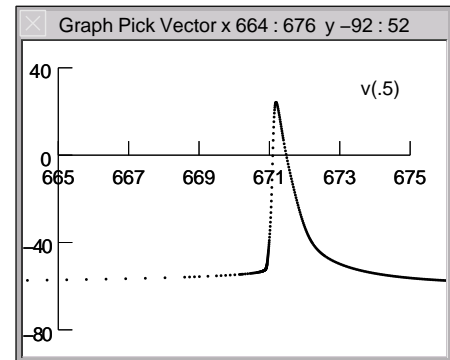
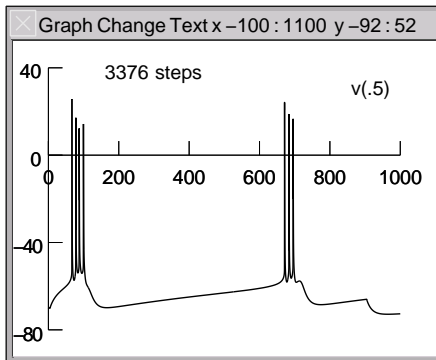
Absolute Tolerance Scale Factors

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 | C |
| 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_{Na,p}](#); [I_{Na,t}](#); [I_K](#); [I_{Sodium}](#); [I_{Potassium}](#);

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_{Na,p}](#); [I_{Na,t}](#); [I_K](#); [I_{Sodium}](#); [I_{Potassium}](#);

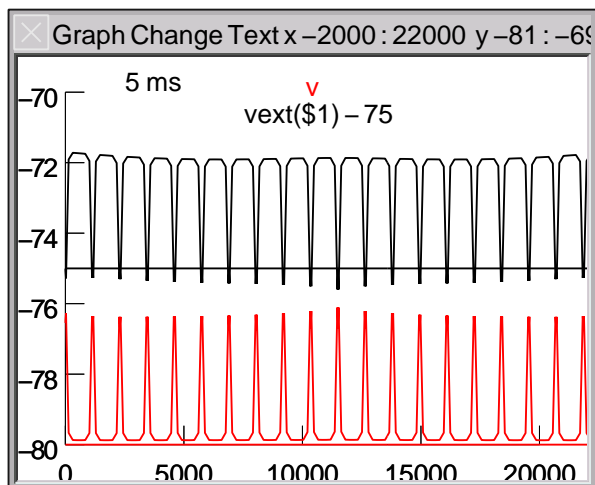
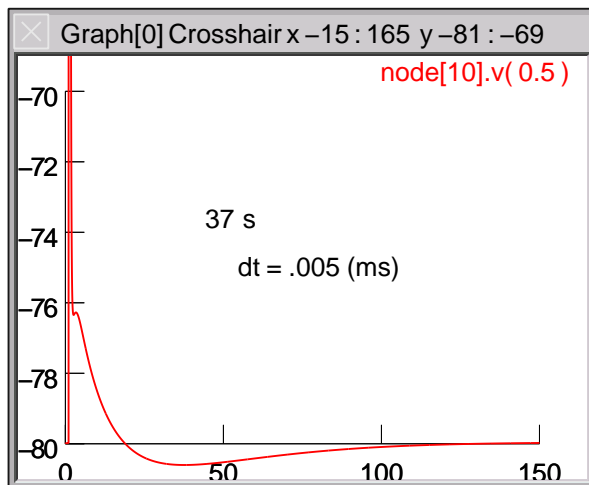
| Model files | Download zip file | Auto-launch | Help downloading and running models |
|---|--|-------------|---|
| \ MRGaxon README AXNODE.mod MRGaxon.hoc mosinit.hoc MRGaxon.ses | SIMULATION OF PNS MYELINATED AXON This model is described in detail in: McIntyre CC, Richardson AG, and Grill WM. Modeling 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](#)

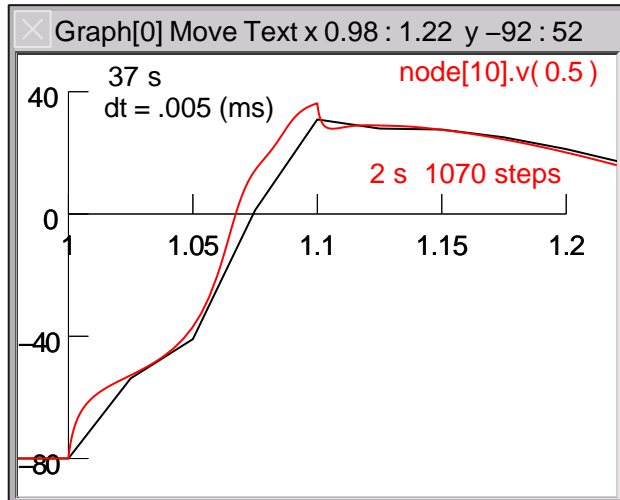
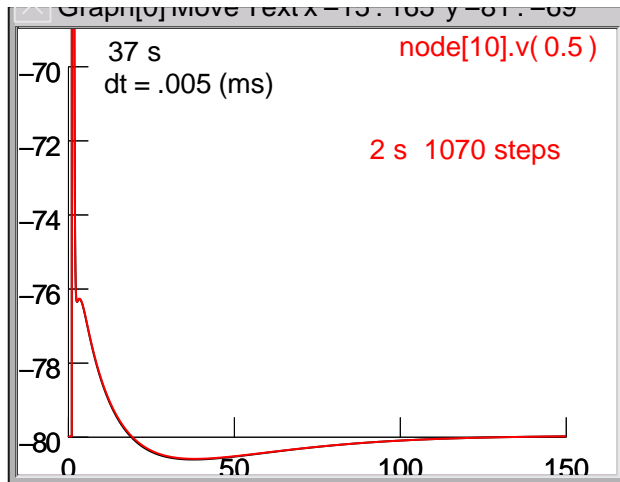
[How to cite ModelDB](#)



ModelView[0]

221 sections; 221 segments

- * 1 real cells
 - * root node[0]
 - 221 sections; 221 segments
 - * 1 distinct values of nseg
 - * 5 inserted mechanisms
 - * Ra
 - * capacitance
 - * pas
- * extracellular
 - * 2 xrxial[0]
 - 160 xrxial[0] = 80684
 - 61 xrxial[0] = 337397
 - xrxial[1] = 1e+09
 - * 2 xg[0]
 - 200 xg[0] = 4.16667e-06
 - 21 xg[0] = 1e+10
 - xg[1] = 1e+09
 - * 2 xc[0]
 - 21 xc[0] = 0
 - 200 xc[0] = 0.000416667
 - xc[1] = 0
 - e_extracellular = 0
- * axnode
- * 6 subsets with constant parameters
- * 1 IClamp



VariableTimeStep

☒ Use variable dt

Absolute Tolerance

Atol Scale Tool Details

Numerical Method Selection

Refresh

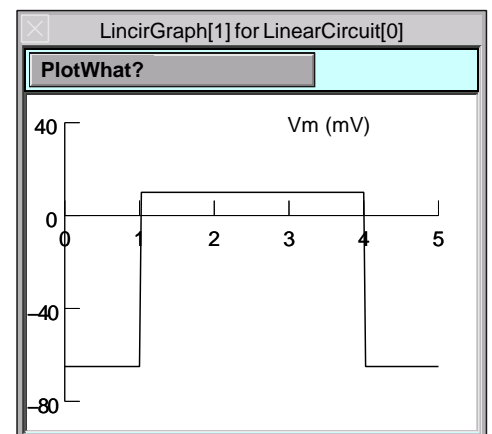
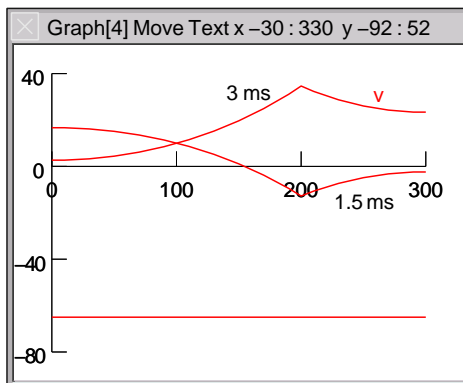
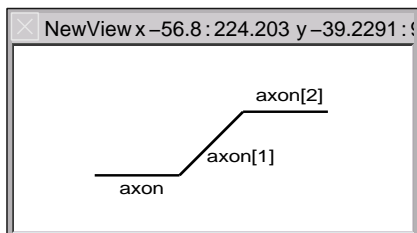
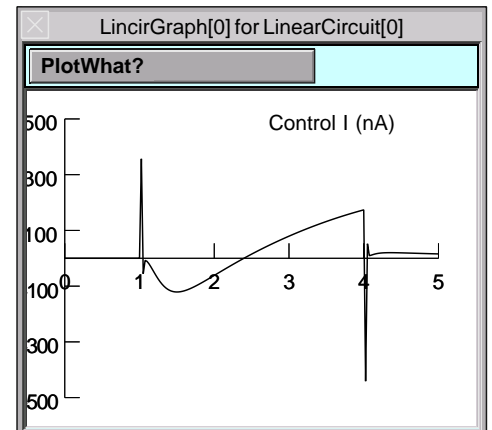
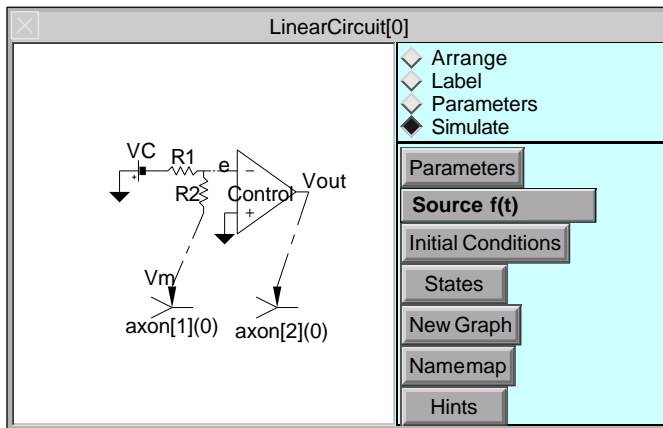
current model type: ODE <*DAE*>
ODE model allows any method
DAE model allows implicit fixed step or daspk

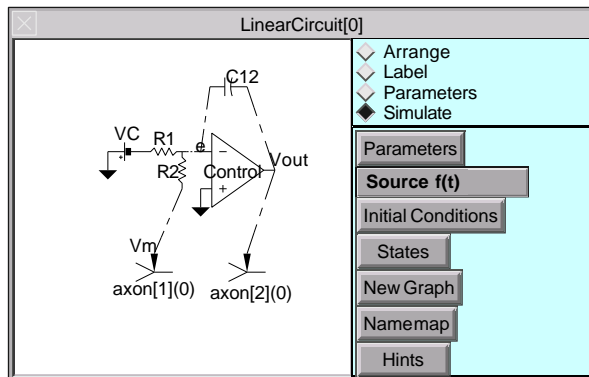
☐ Implicit Fixed Step
☐ C-NFixedStep
☐ Cvode
☒ Daspk
☐ Local step

DAE and daspk require sparse solver, cvode requires tree solver

☐ Mx=b tree solver
☒ Mx=b sparse solver

☐ 2nd order threshold (for variable step)





Values for LinearCircuit[0]

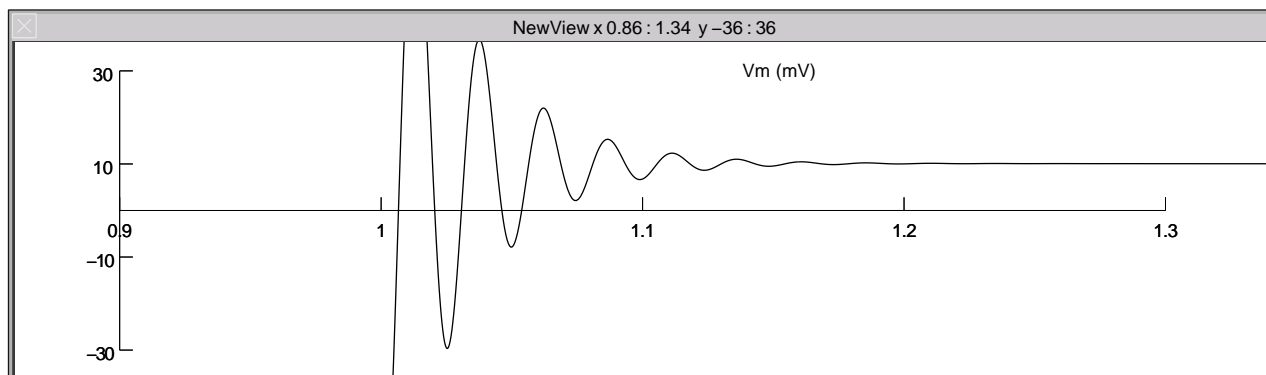
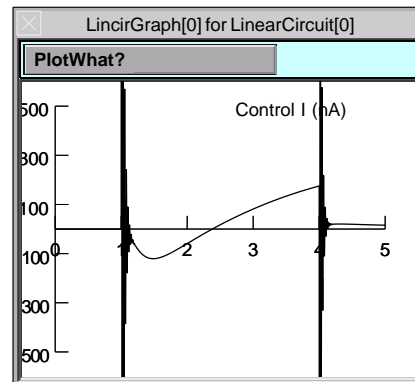
| | |
|------------------|-------|
| Control Gain | 1e+05 |
| Control Tau (ms) | 0 |
| R1 (Mohm) | 1e+05 |
| R2 (Mohm) | 1e+05 |
| C12 (nF) | 1e-08 |

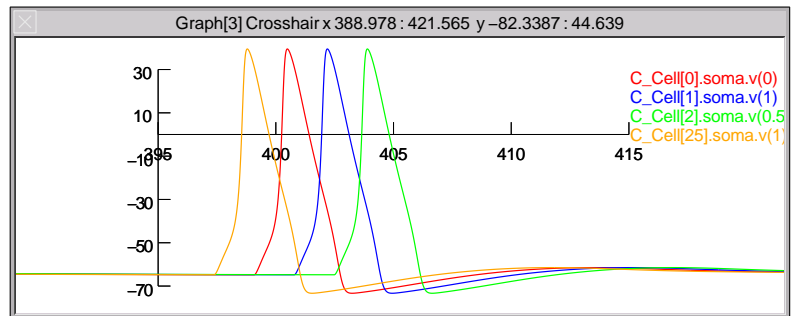
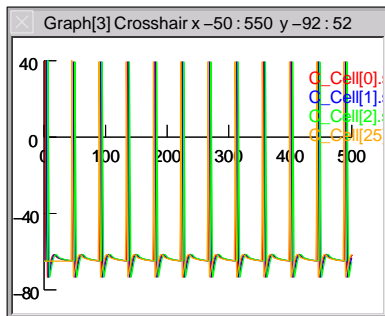
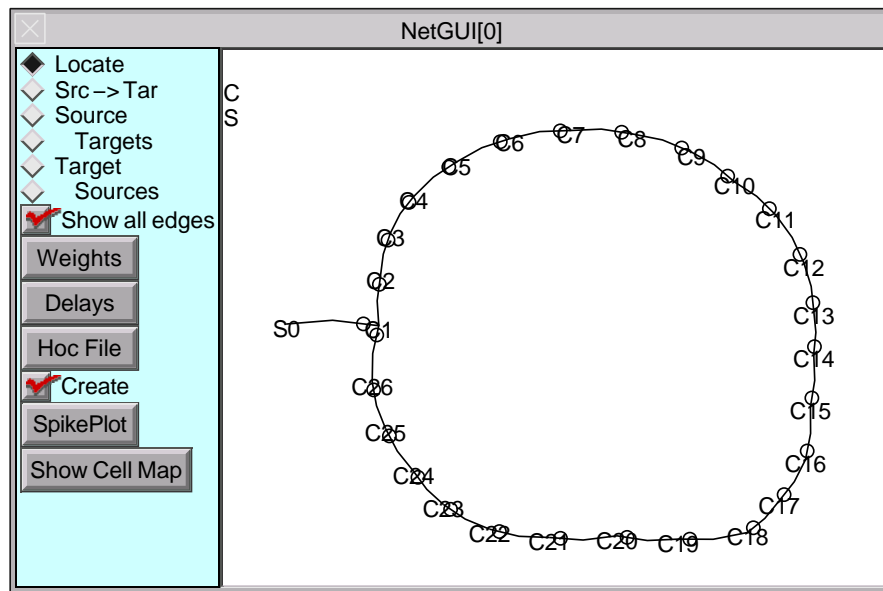
VariableTimeStep

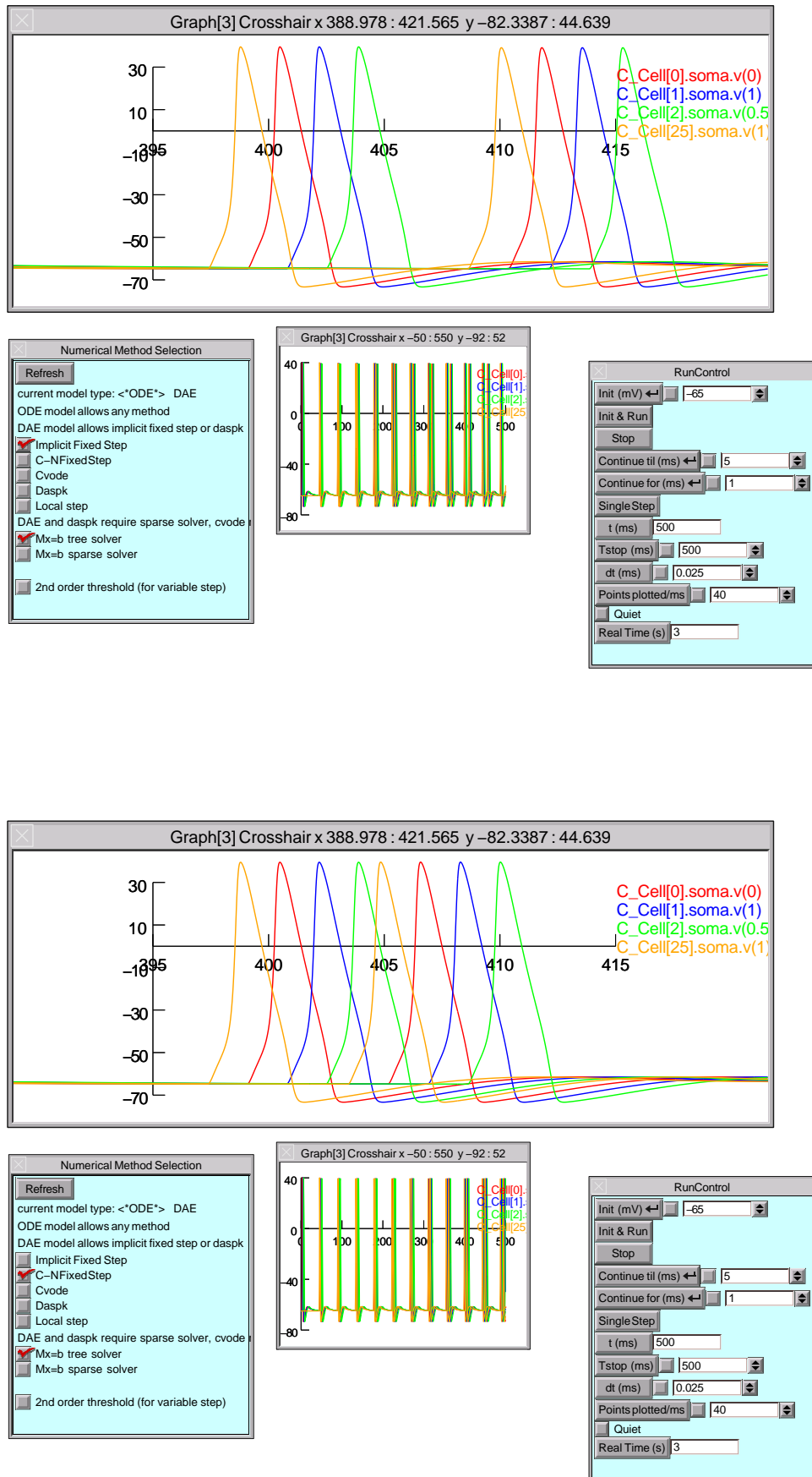
☒ Use variable dt

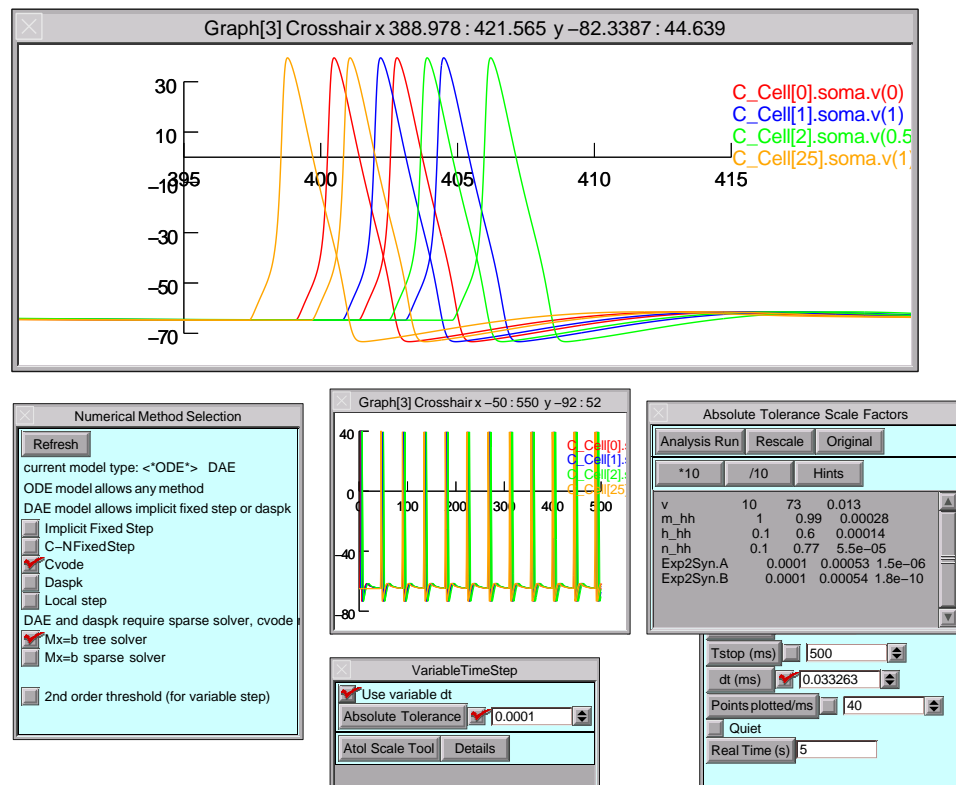
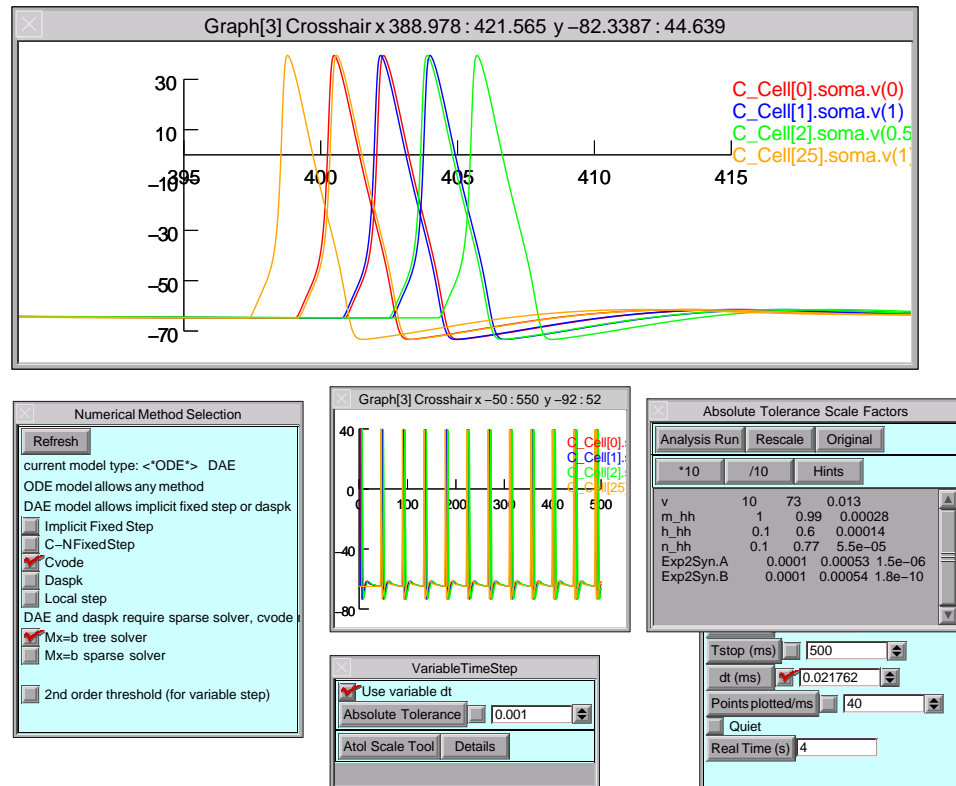
Absolute Tolerance 0.001

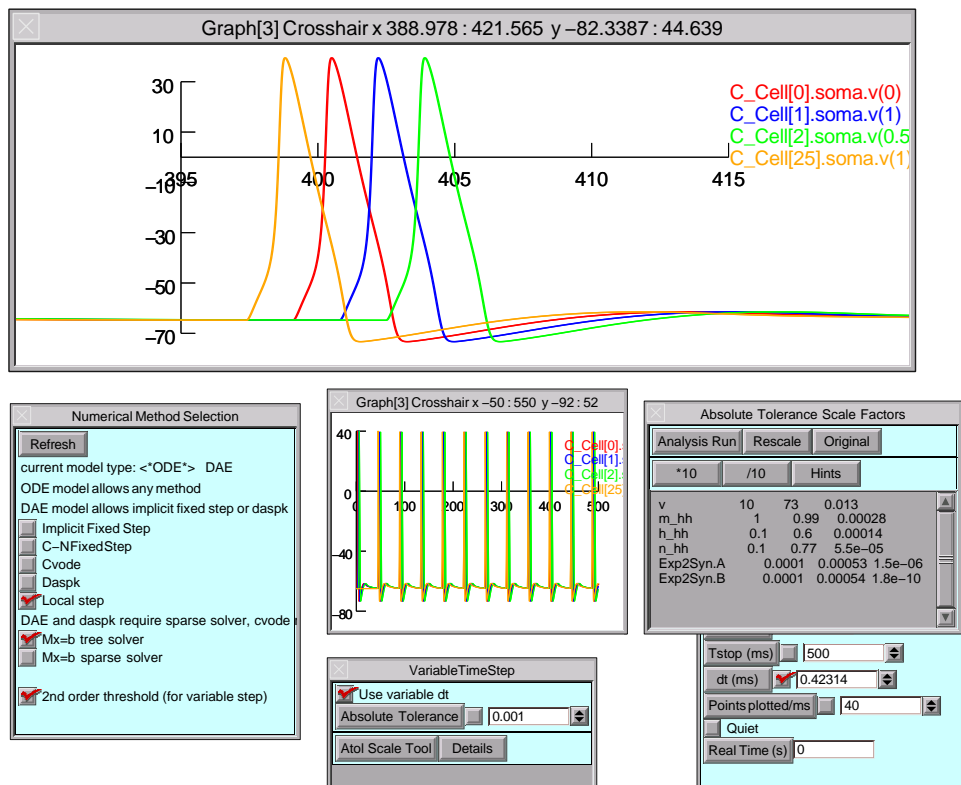
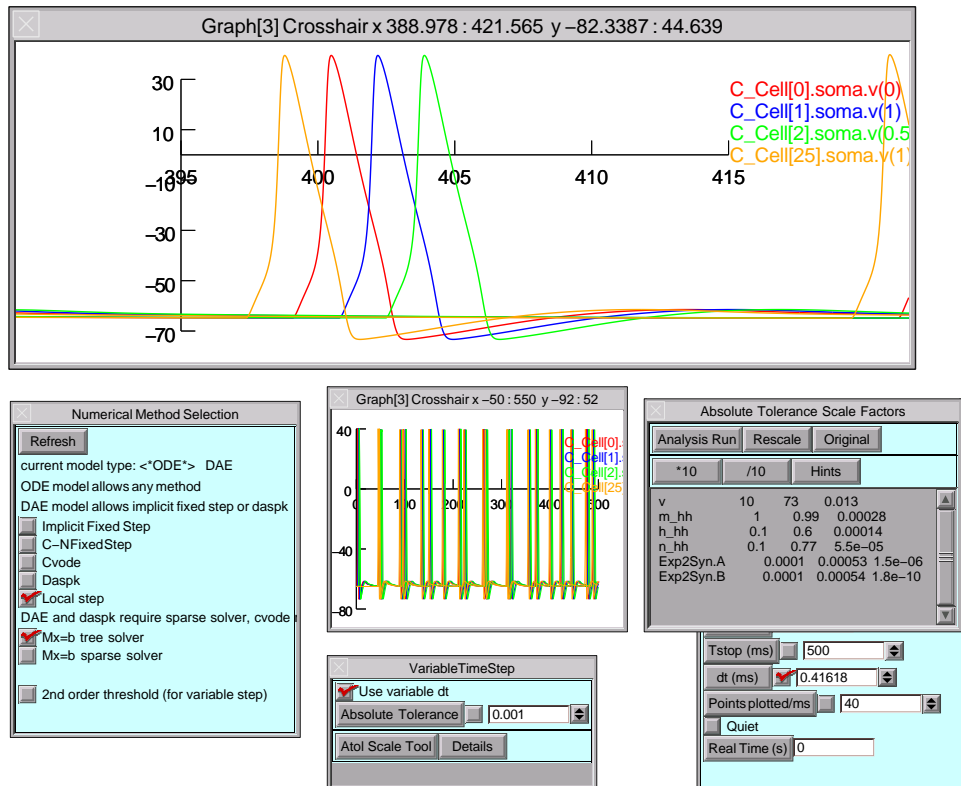
Atol Scale Tool Details

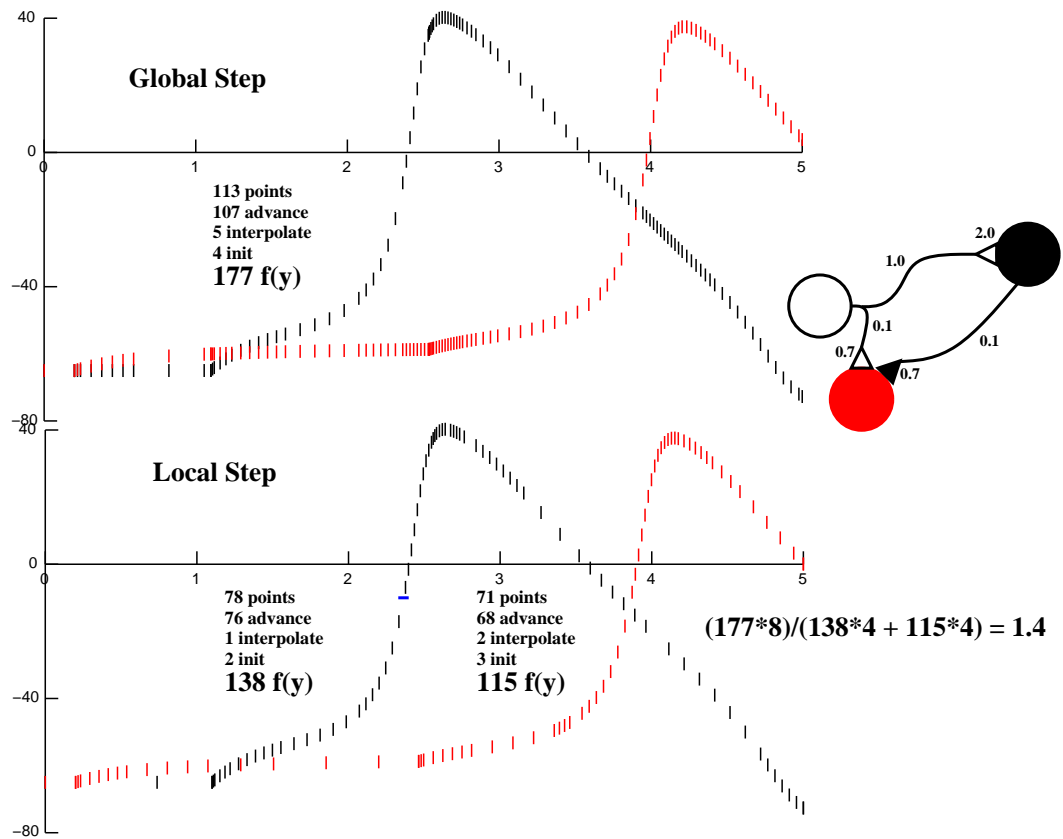






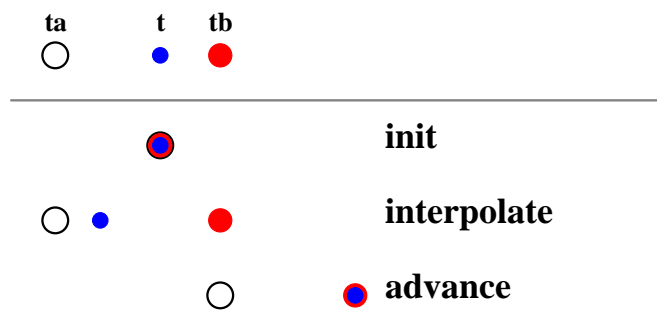


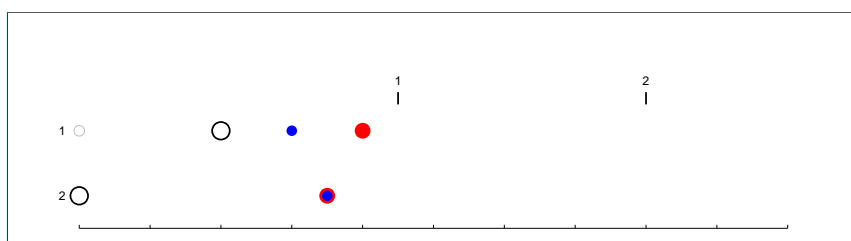
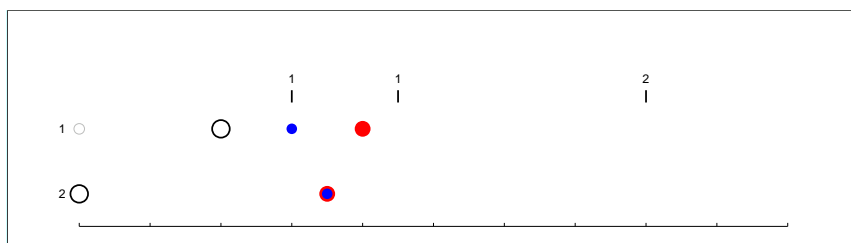
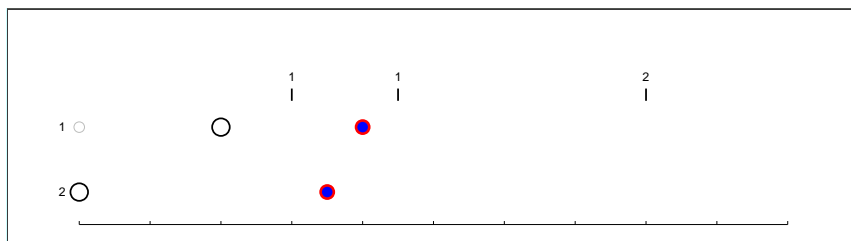
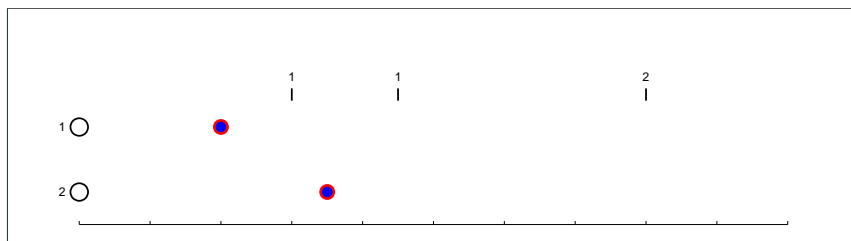
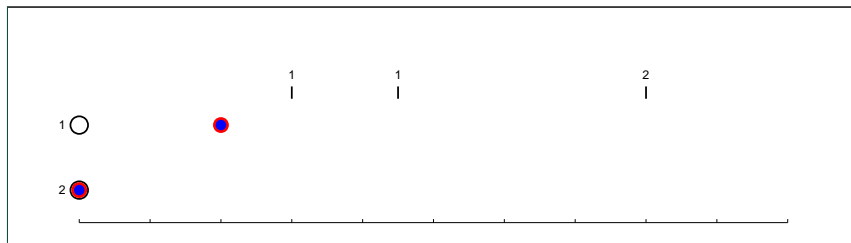
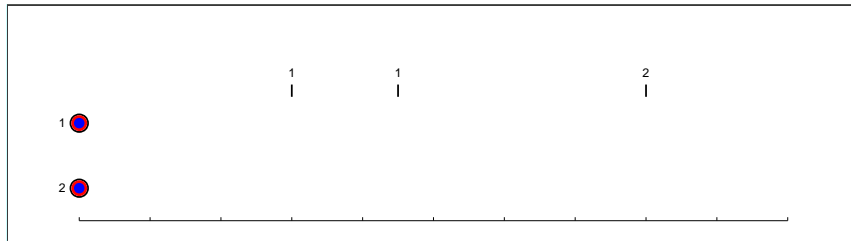


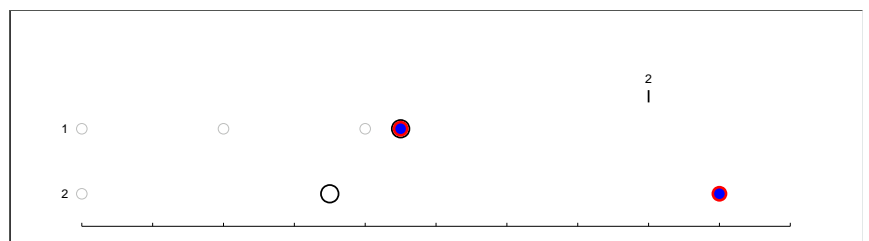
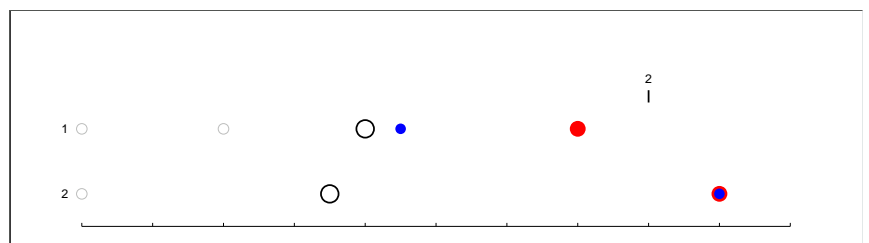
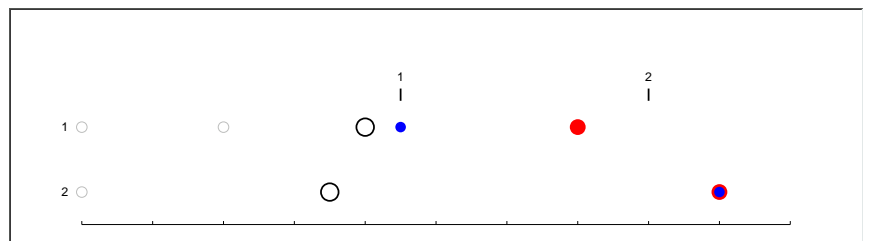
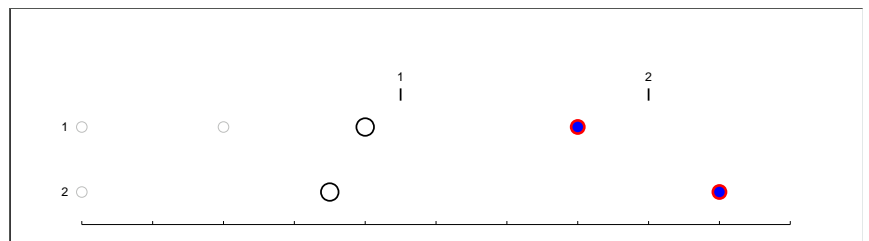
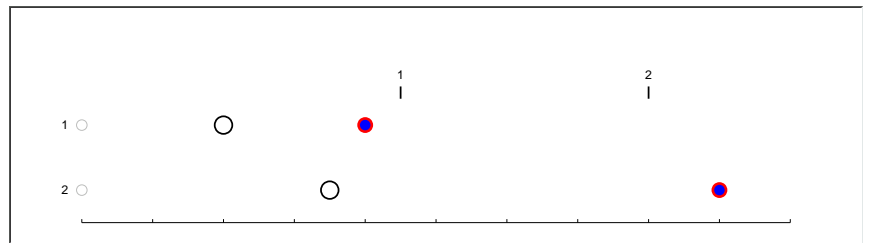
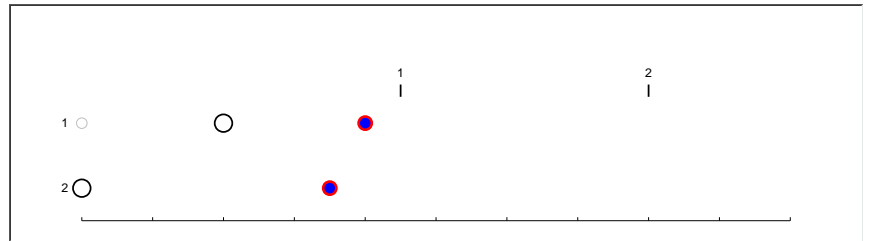


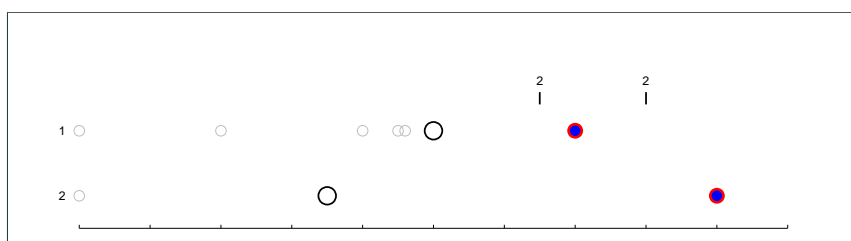
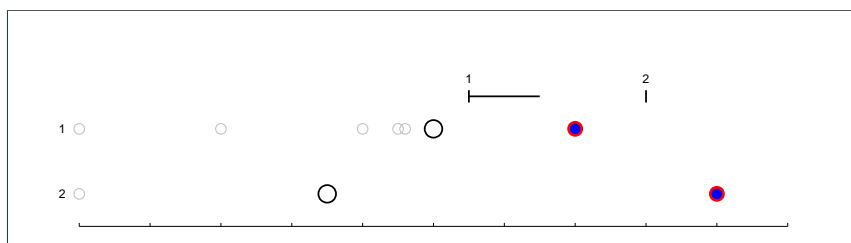
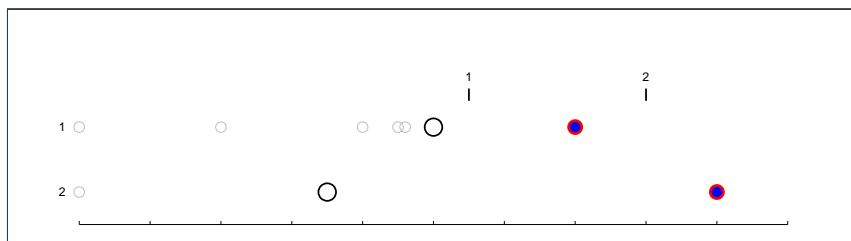
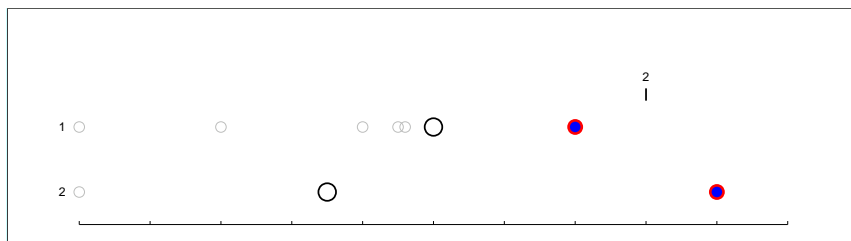
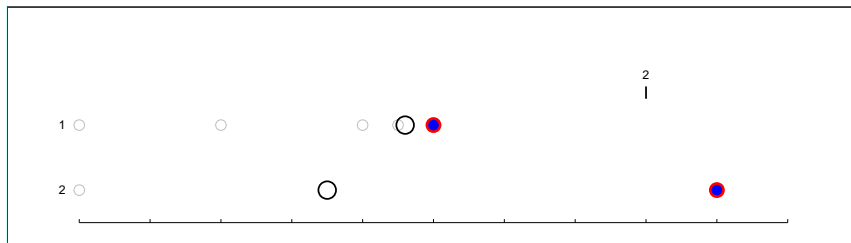
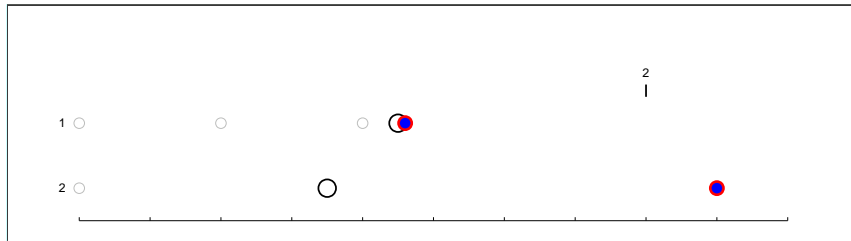
One integrator instance per cell

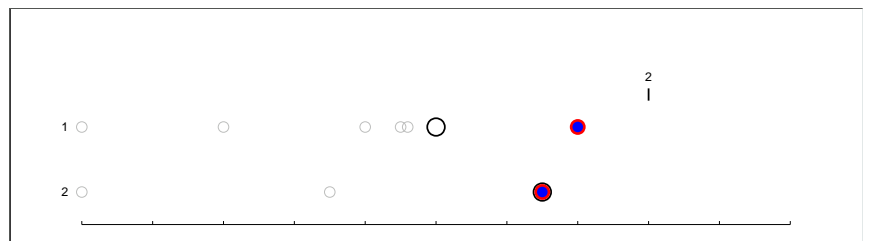
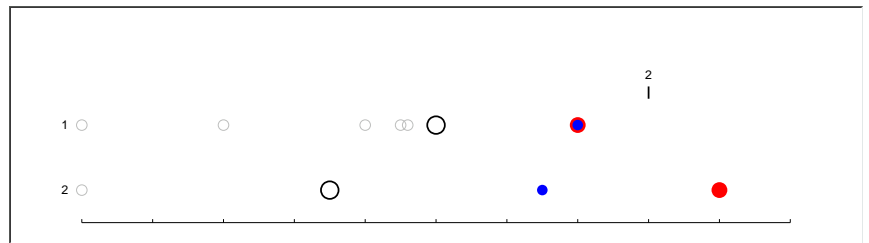
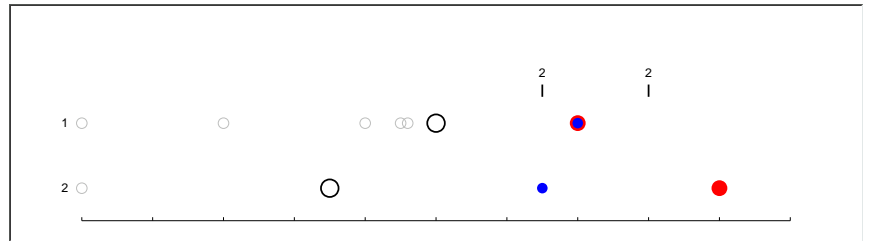
$$\forall i, j: ta_i \leq tb_j$$











```
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_A receptors

COMMENT

Minimal kinetic model for GABA_A 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:

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

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

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).

```

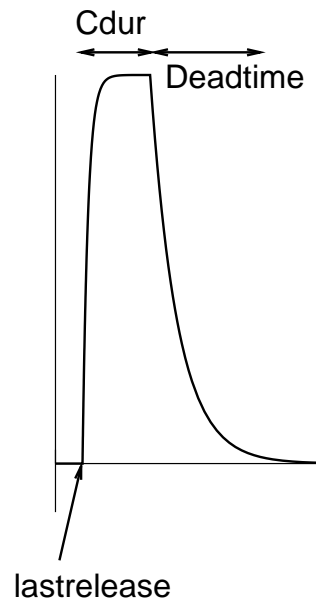
PROCEDURE release() { LOCAL q
:will crash if user hasn't set pre with the connect statement

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}

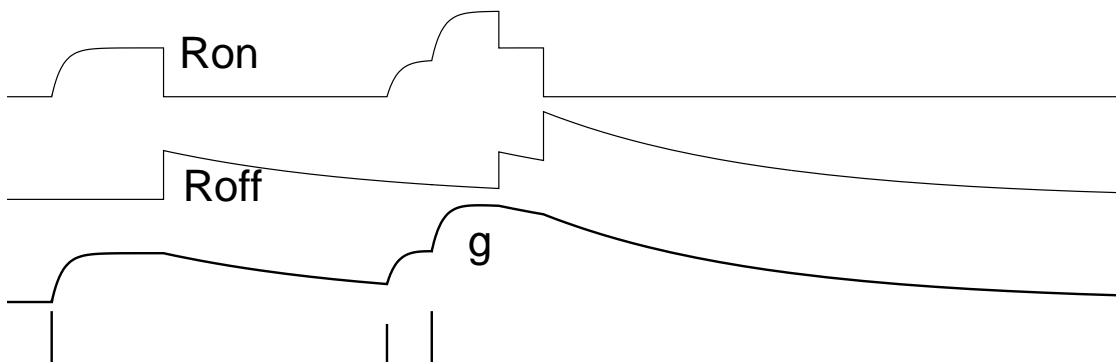
INITIAL {
  Ron = 0  Roff = 0
  Rinf = Alpha / (Alpha + Beta)
  Rtau = 1 / (Alpha + Beta)
  Rdelta = Rinf*(1 - exp(-Cdur/Rtau))
  synon = 0
}

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() }
}
```



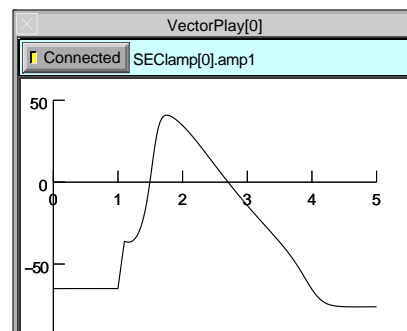
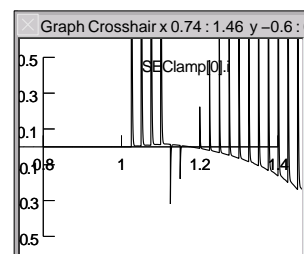
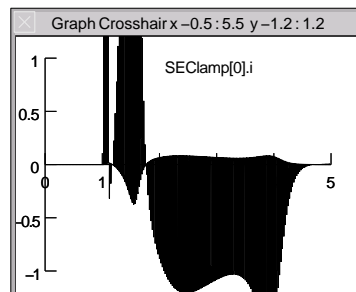
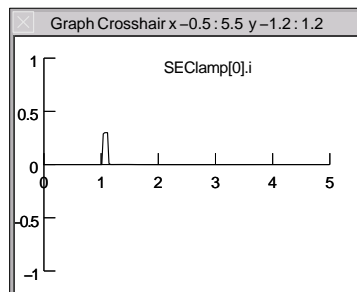
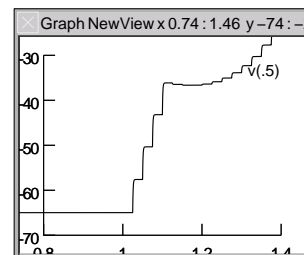
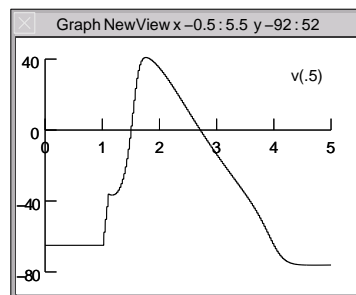
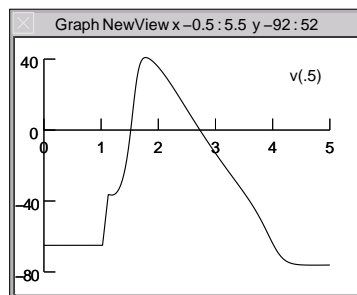
```
fh = new FInitializeHandler("ev()")
proc ev() {
  ccode.event(t1, "p()")
}
```

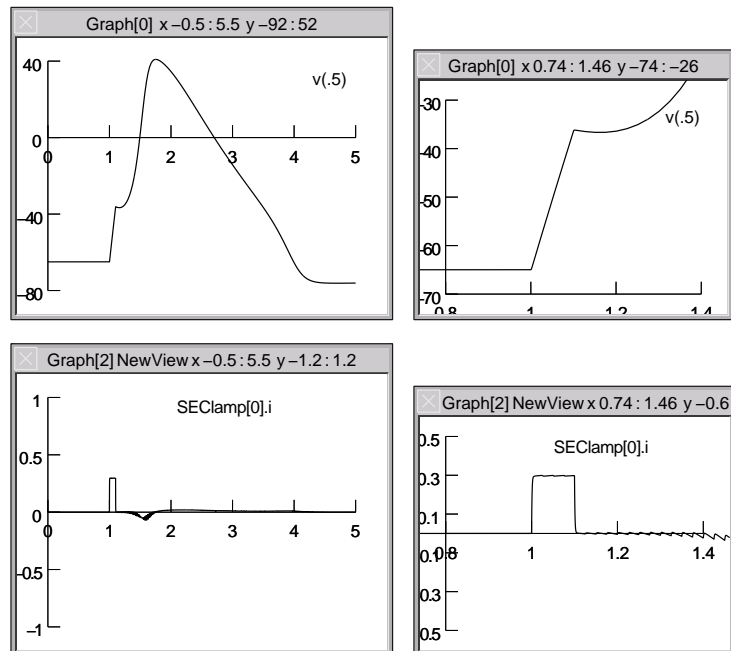
```
proc advance() {
  fadvance()
  if (soma.v(.5) > 10) { p() }
}
```



```
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)
```


Initialization, broadly speaking:

We want to get the same result every time we click on
Init & Run, no matter what we did before

Note: this presentation explicitly omits details of initialization
of ionic concentrations and equilibrium potentials

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

Slide 2

Initialization should assign values at $t = 0$ for

- membrane potential
- gating states
- ionic concentrations
- chemical kinetic states
- voltage across capacitors in linear circuits
- internal states of op amps
- random number generators

and properly configure

- event queues
- vector record and play
- counters

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

NEURON's `finitialize()`

- sets `t = 0`
- clears event queue
- sets up internal data structures that depend on topology and geometry
- initializes `Vector.play` controller
- delivers events whose delivery time is 0
- if `finitialize` was called with `v_init` argument, sets `v` in all compartments to `v_init`
- calls `INITIAL` block of every inserted mechanism in every segment
- if `extracellular` is used, sets `vext` to 0
- initializes ions; calculates equilibrium potentials if necessary
- initializes mechanisms that `WRITE` ion concentrations; recalcs equilib potentials as needed
- calls all other `INITIAL` blocks
- initializes `LinearMechanism` states
- calls `INITIAL` blocks inside `NET_RECEIVE` blocks; if this spawns network events, delivers any whose delay is 0 to their target `NET_RECEIVE` blocks
- if fixed time step integrator is used, calls all `BREAKPOINT` blocks
- initializes adaptive integrator (if being used)
- initializes any `ccode.record` and `vector.record` recordings

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

Default initialization: the standard run library

`nrn/share/nrn/lib/hoc/stdrun.hoc`
 (MSWin: `c:\nrn\lib\hoc\stdrun.hoc`)

`stdinit()`

Called when you

click on `Init` or `Init & Run` in the `RunControl`

or

enter a new value for `v_init` in the `Init` button's field editor

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

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

init()

Most customizations are made here

```

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

```

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

INITIAL blocks in NMODL**HH-like mechanisms**

```

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

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

```

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

Kinetic schemes

```

INITIAL {
    SOLVE scheme METHOD steadystate
}
e.g.
NEURON {
    USEION k READ ek WRITE ik
}
STATE { c1 c2 o }
INITIAL {
    SOLVE scheme METHOD steadystate
}
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)
}

```

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

Default initialization of STATES

Use state0, e.g.

```

PARAMETER {
    state0 = 1
}

```

or alternative syntax

```

STATE {
    state START 1
}

```

It's best to be explicit

```

INITIAL {
    m = m0
    h = h0
}

```

To make them visible from hoc

```

NEURON {
    GLOBAL m0
    RANGE h0
}

```

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

Typical custom initializations

Steady state

unperturbed system

system under constant voltage or current clamp

Defined starting point on a trajectory
of an oscillating or chaotic system

Adjust parameters to meet some condition

How?

Use a custom `init()` procedure.

Load after the standard library, so it won't be overwritten.

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

Initializing to steady state

"Travel into the past," take large steps with implicit Euler, then return to the present.

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

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

Initializing to a desired state

Especially useful for oscillating or chaotic models.

Run a "warmup simulation," then save all states

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

If desired, write state info to a file for future use

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

To read from a file

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

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

Initializing to a desired state continued

A custom init() that restores saved states

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

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

Initializing to a particular resting potential

One approach: adjust the leakage equilibrium potential
so that leakage current balances the other ionic currents
when the cell is at the desired resting potential

Example: for a single compartment model with hh,

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

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

Alternative strategy: add a mechanism that injects a constant current
to balance the other currents.

Example:

```
NEURON {
  SUFFIX constant
  NONSPECIFIC_CURRENT i
  RANGE i, ic
}

UNITS {
  (mA) = (milliamp)
}

PARAMETER {
  ic = 0 (mA/cm2)
}

ASSIGNED {
  i (mA/cm2)
}

BREAKPOINT {
  i = ic
}
```

This needs a different custom `init()`

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

Custom `init()` to use with constant current mechanism:

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

Copyright © 1998–2003 N.T. Carnevale and M.L. Hines, all rights reserved

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

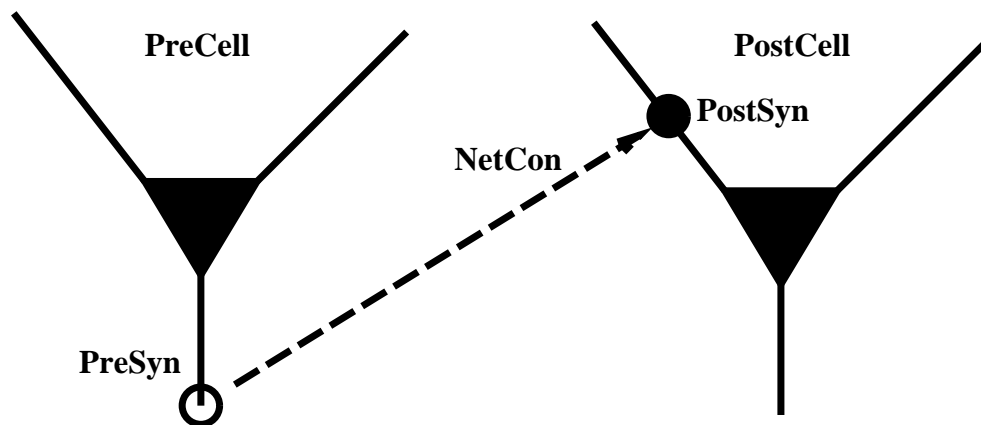
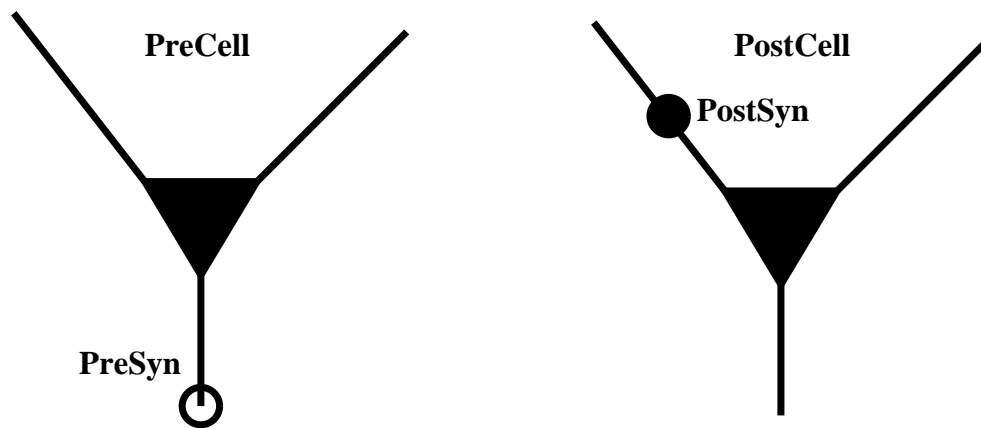
Parallel Computation

A simulation run takes hours.

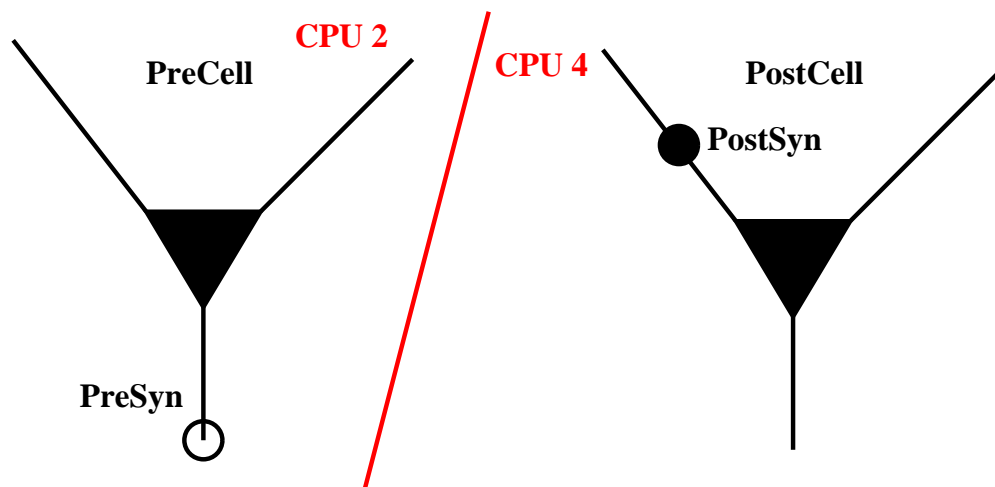
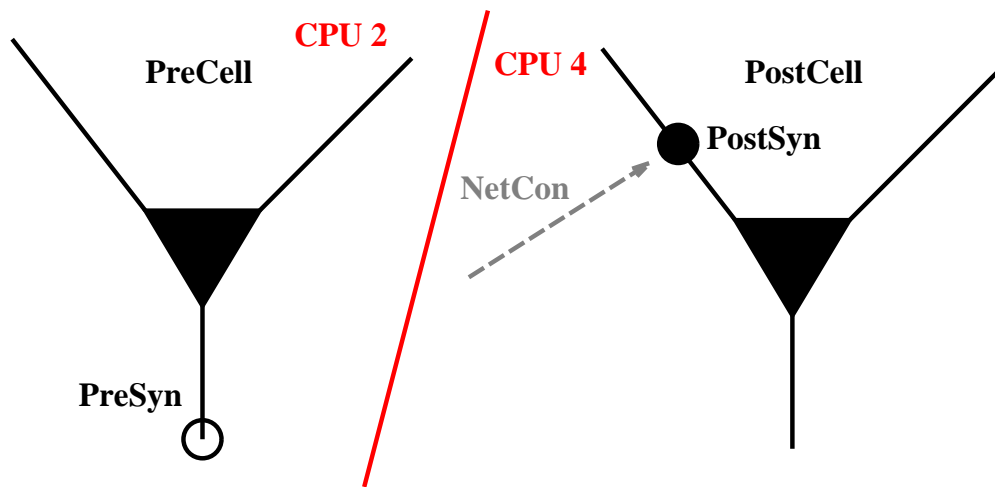
want to spread the problem over several machines.

Single cells

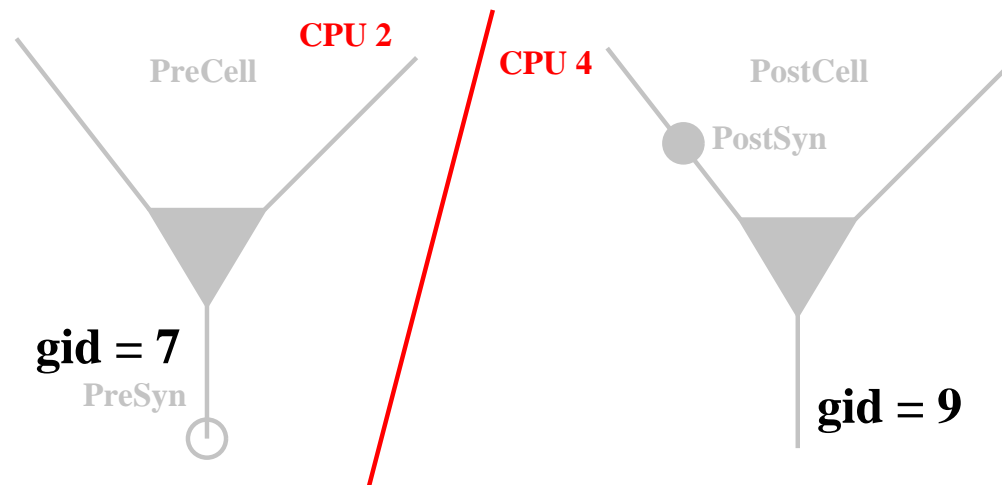
portions of the tree cable equation on
different machines.



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



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



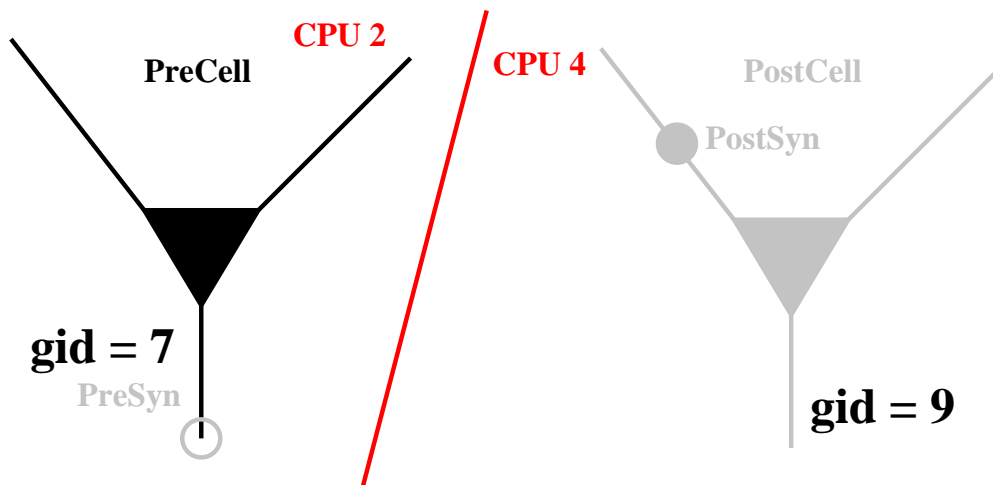
Every spike source (cell) must have a global id number.

| 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:

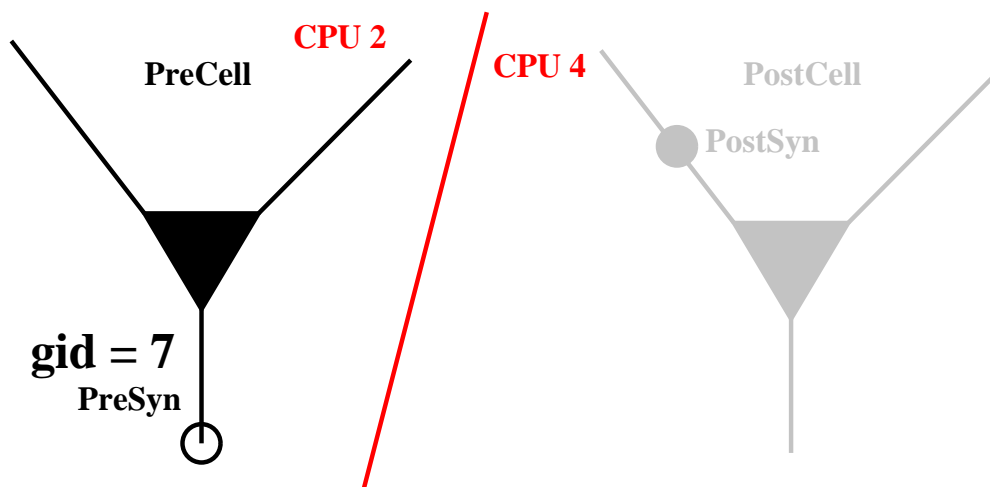
```
for (gid = pc.id; gid < ncell; gid += pc.nhost)
    pc.set_gid2node(gid, pc.id)
    ...
}
```

body executed only ncell/nhost times, not ncell.



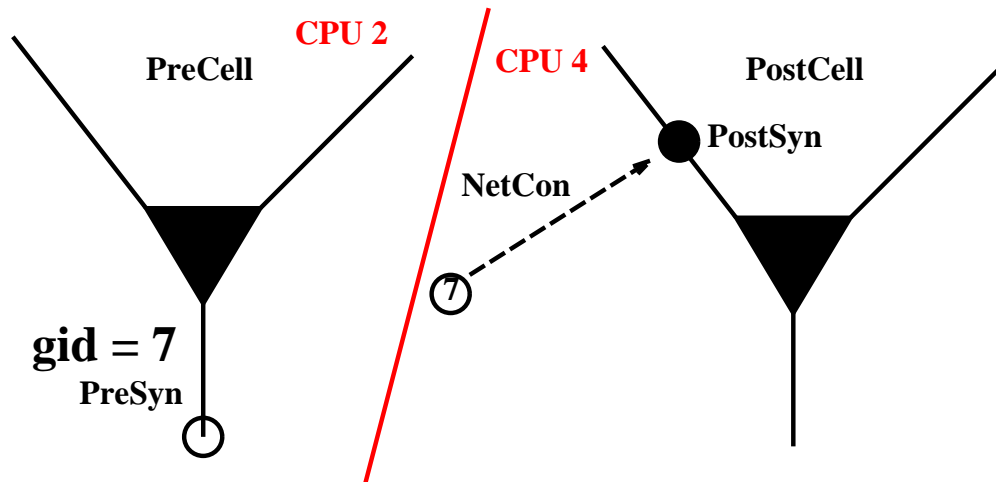
Create cell only where the gid exists.

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



Associate gid with spike source.

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

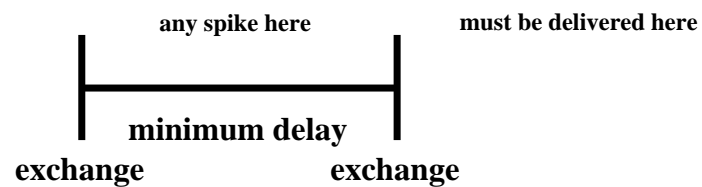
Create NetCon on CPU where target exists.

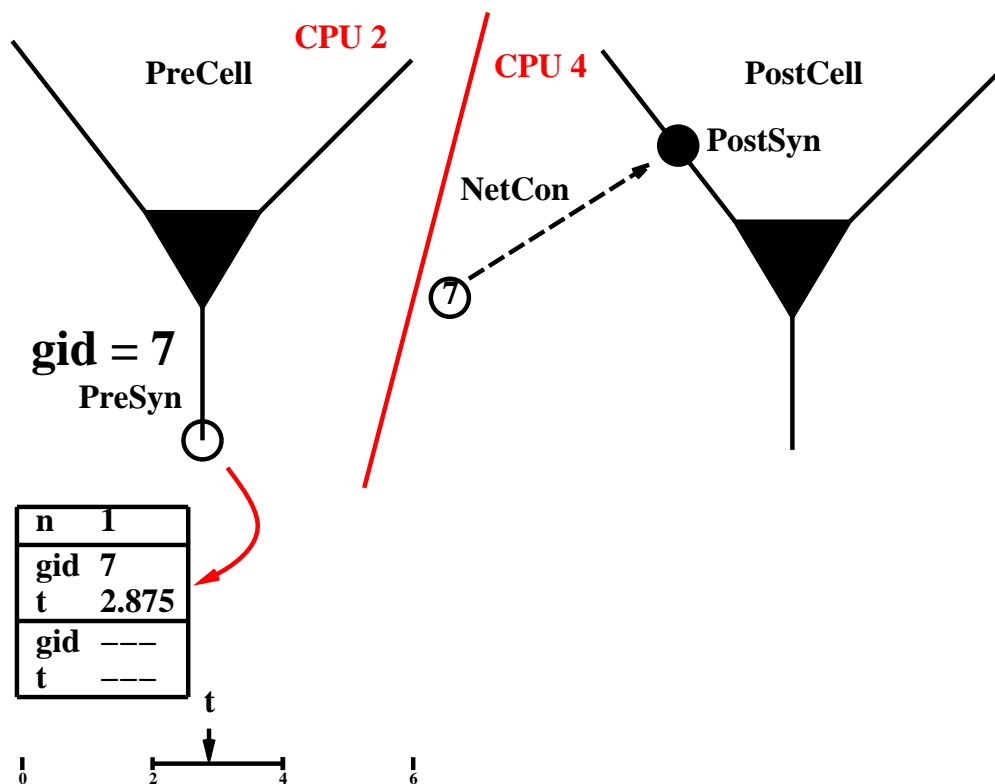
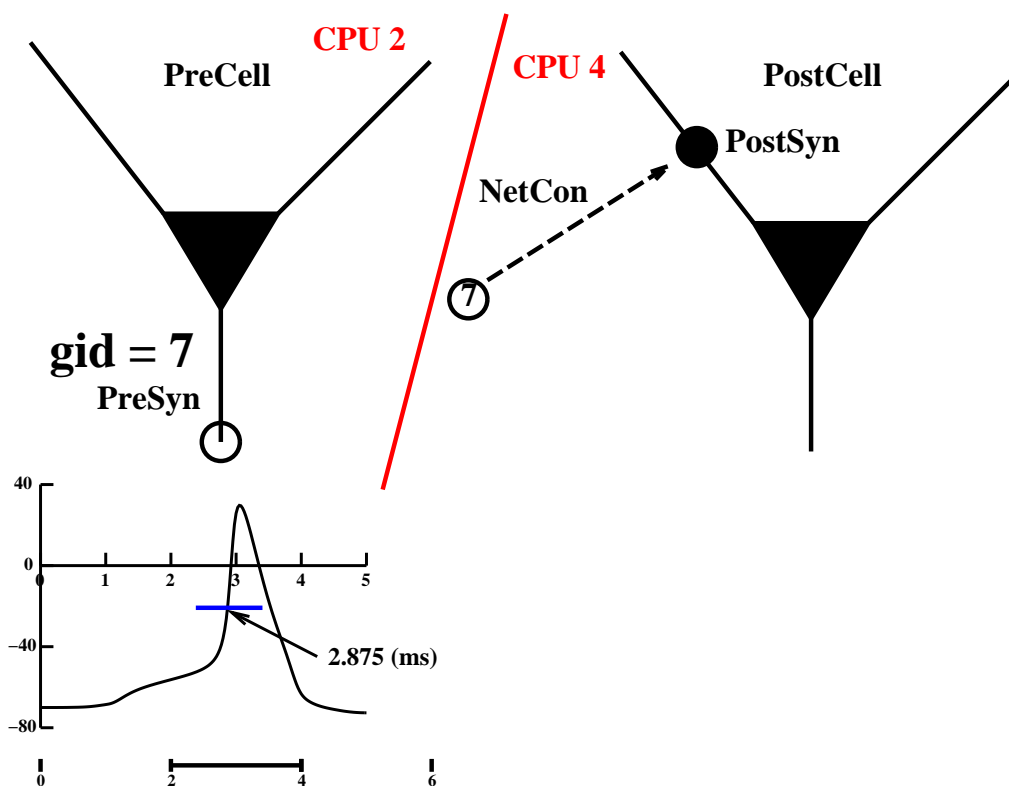
```
nc = pc.gid_connect(7, PostSyn
```

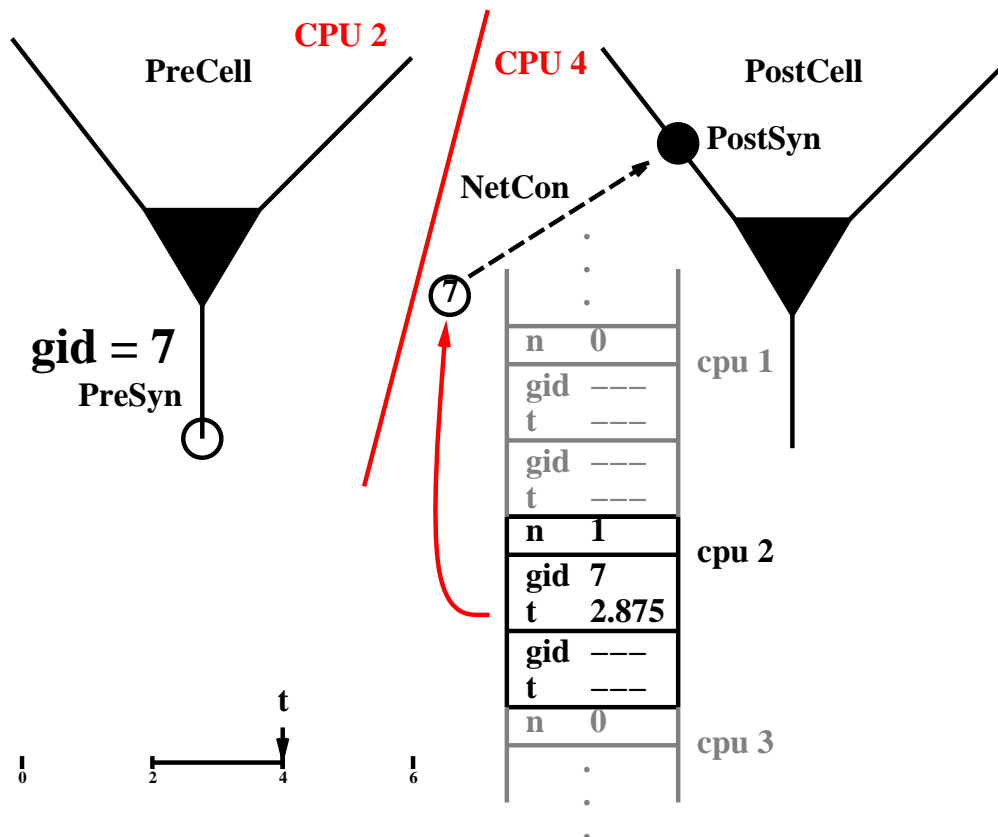
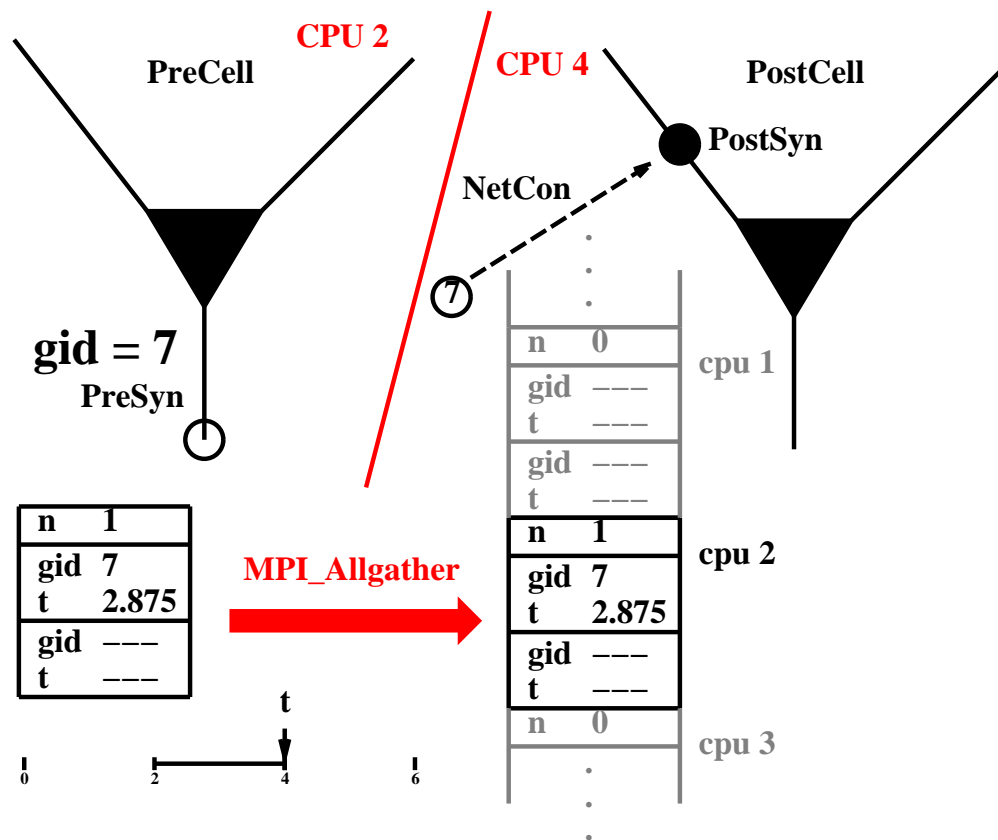
Run using the idiom

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

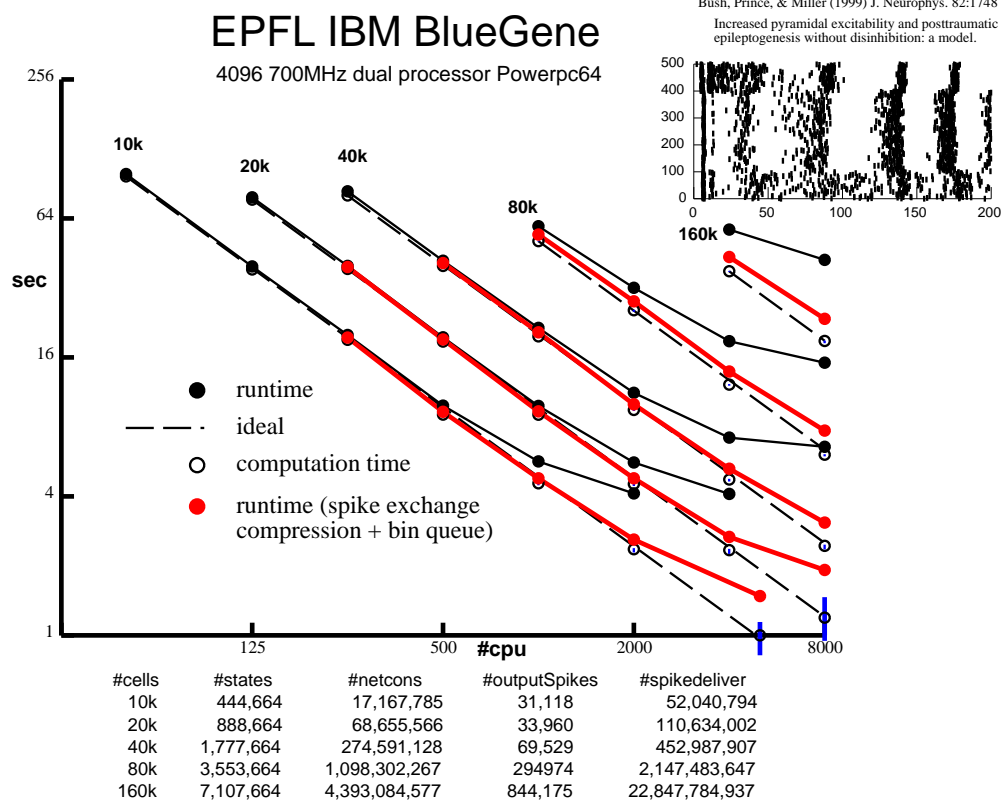
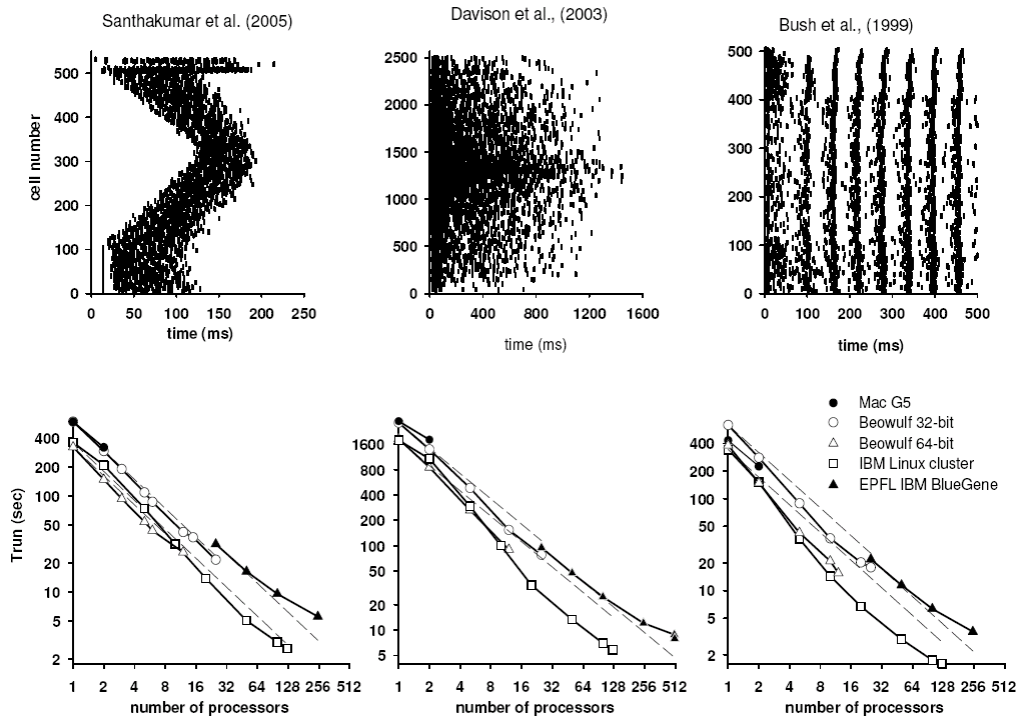
pc.set_maxstep() uses
MPI_Allreduce
to determine minimum delay.



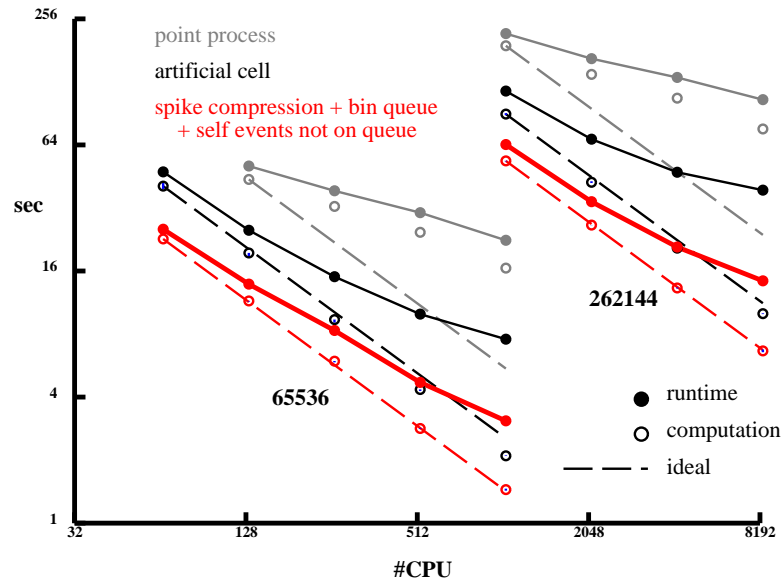




Migliore et al (2006) J. Comput. Neurosci. 21(2):119



Artificial Spiking Net Performance

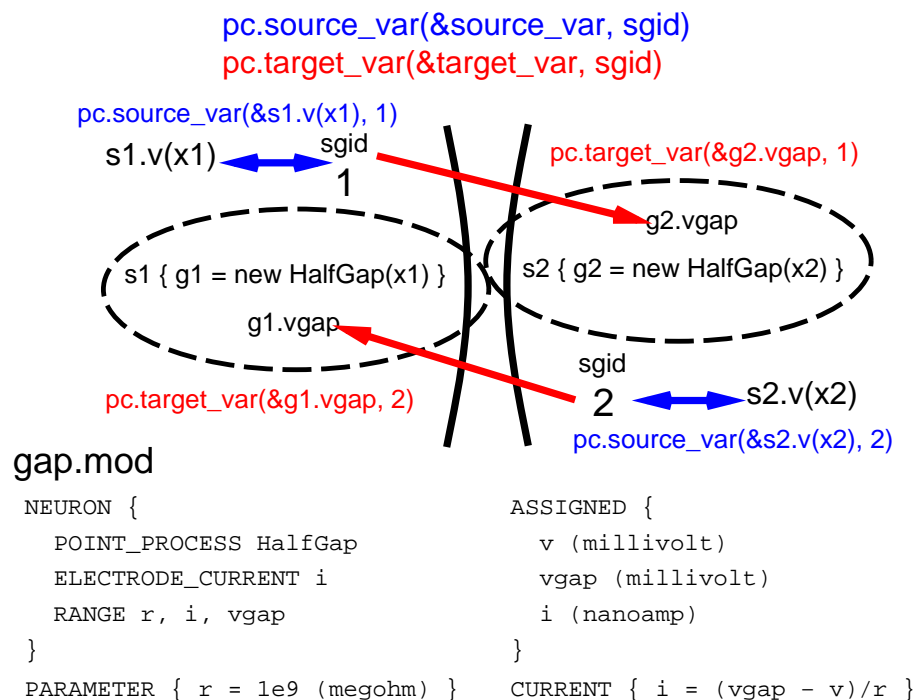


Each cell fires randomly every 10 to 20 ms.
65K cells, 1000 random connections per cell
256K cells, 10,000 random connections per cell

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

Gap Junction Specification

Continuous Voltage Exchange

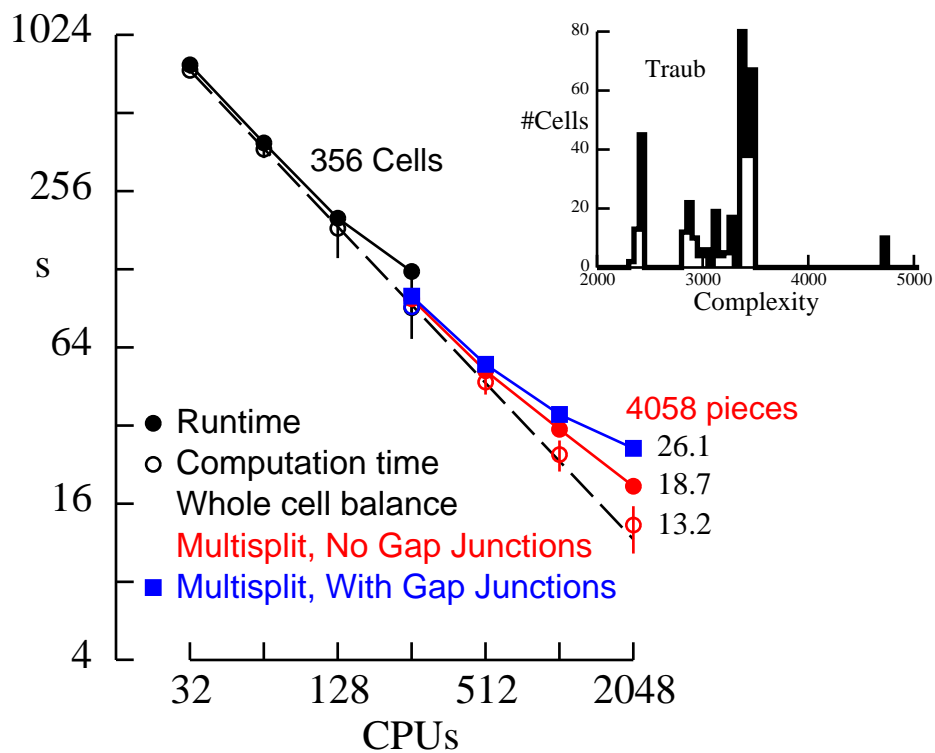
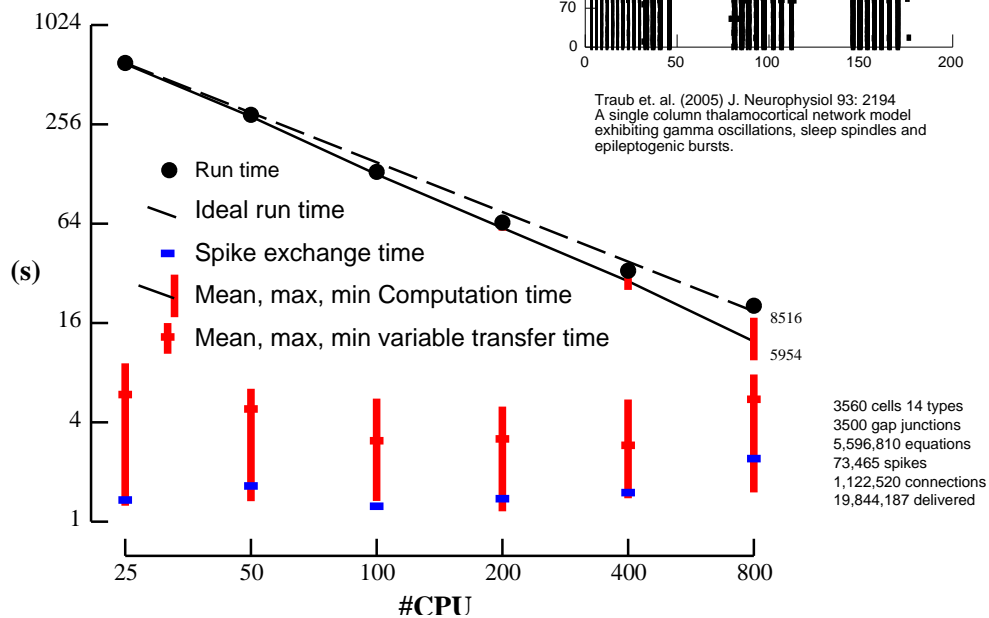


Pittsburgh Supercomputing Center

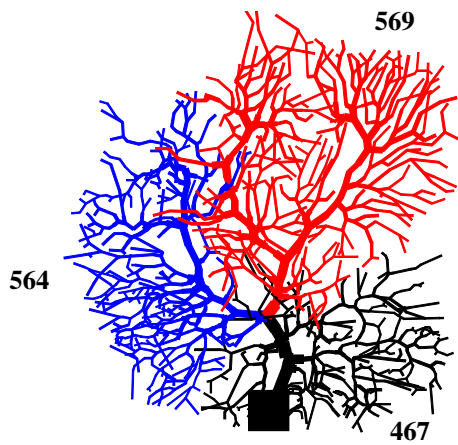
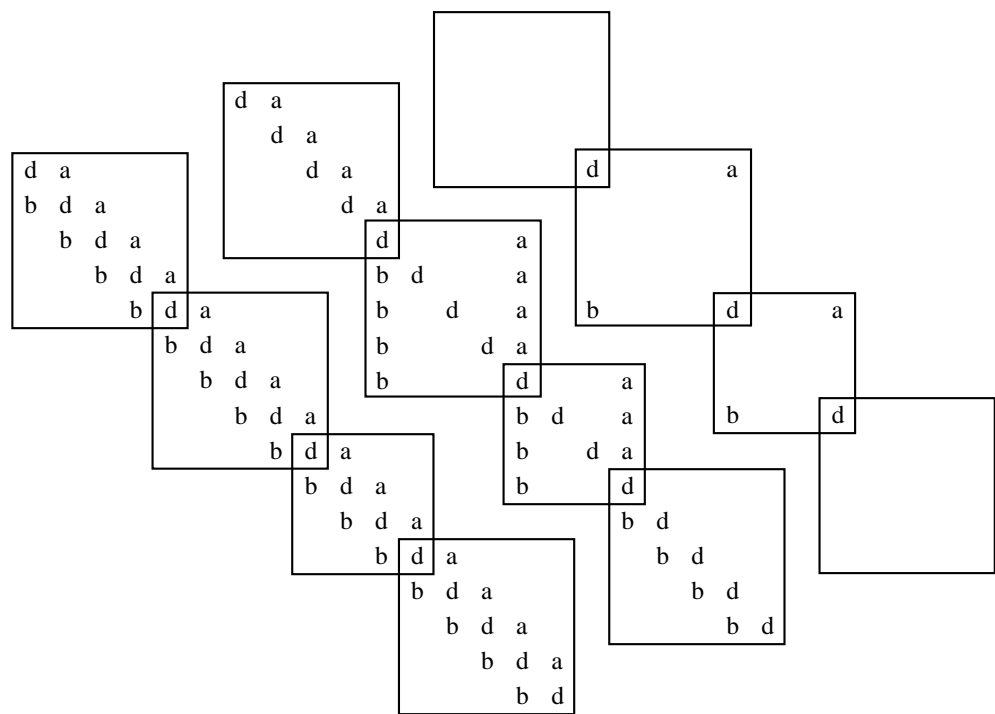
Bigben

Cray XT3

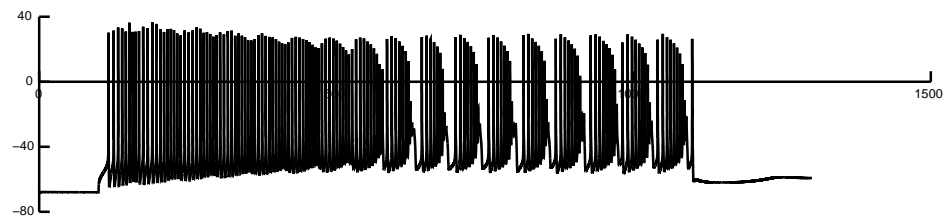
2068 2.4 GHz Opteron Processors



Multisplit Gaussian Elimination

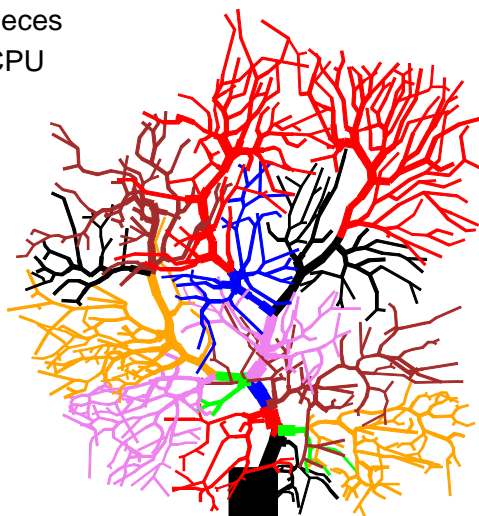


| #CPU | Runtime (s) | | |
|------|-------------|----------|-------|
| 1 | 54.8 | | |
| 3 | 22.2 | 19.5 | 18.3 |
| | | expected | ideal |

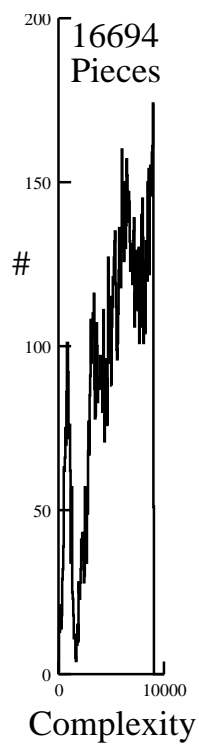
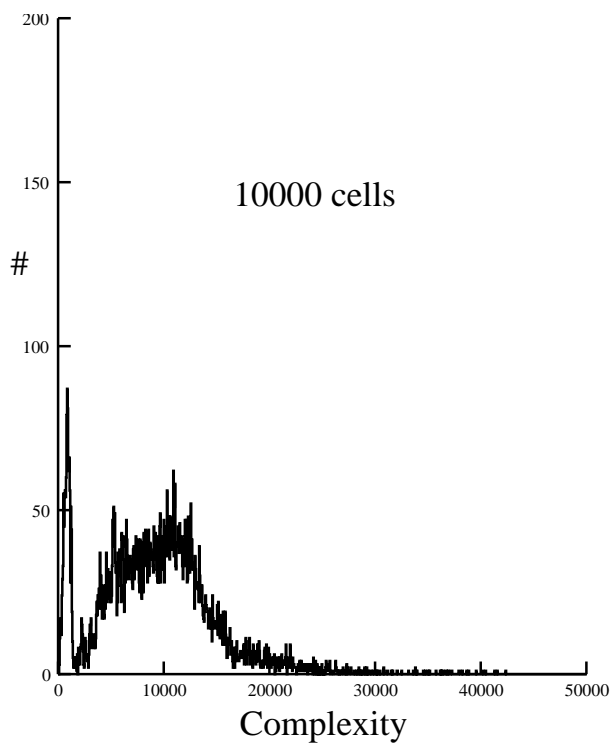
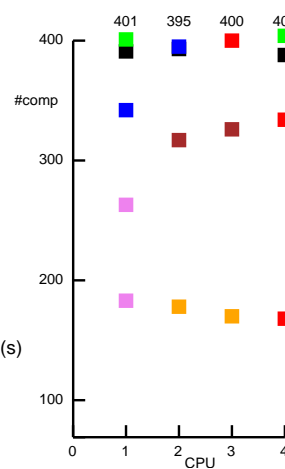
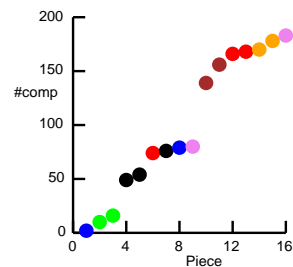


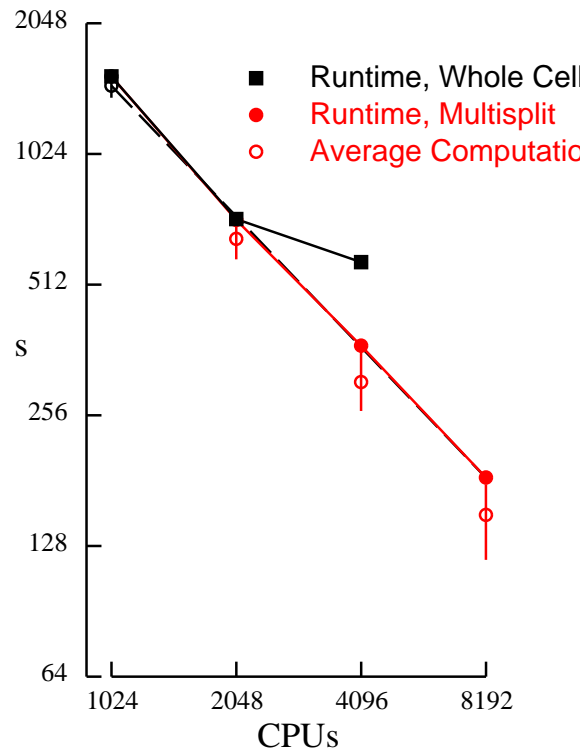
De Schutter & Bower (1994) J. Neurophysiol 71:375
Ported to NEURON by Jenny Davie, Volker Steuber, and Arnd Roth.

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 | | |





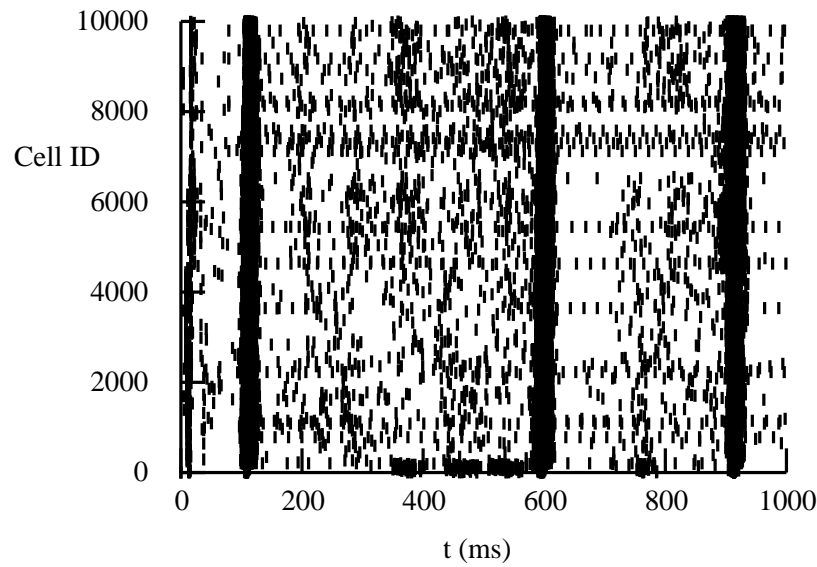
Results must be independent of
 Number of processors
 Distribution of cells

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



out.spk (58,858 lines)

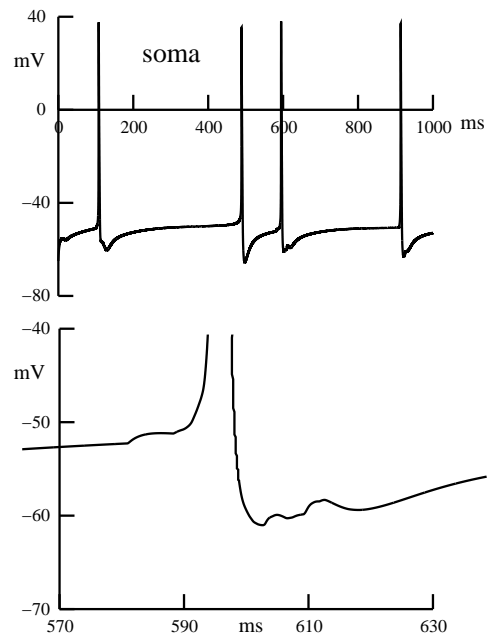
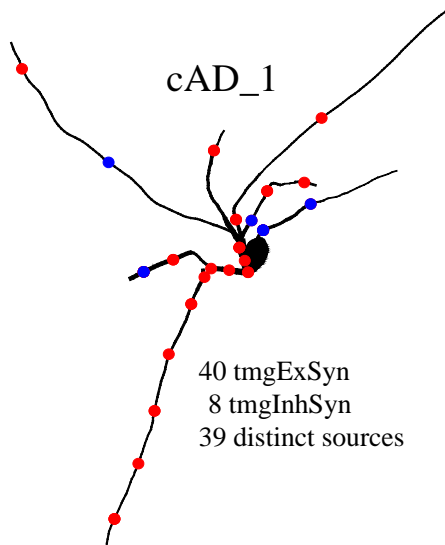
3.975 8050

3.975 8621

...

999.125 4632

| Line# | spike(ms) | gid |
|--------|-----------|-----|
| 8548: | 107.25 | 1 |
| 18501: | 488.625 | 1 |
| 27276: | 594.25 | 1 |
| 51470: | 913.475 | 1 |



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

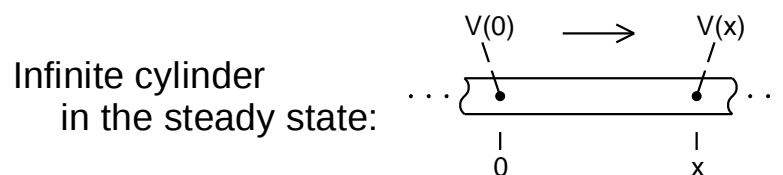
$$V_{downstream}/V_{upstream}$$

- 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

λ = 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

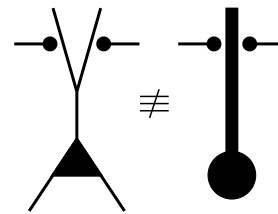
$$A^V(x) = \cosh L_{\text{classical}} / \cosh (L_{\text{classical}} - X)$$

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

$$X = x / \lambda$$

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

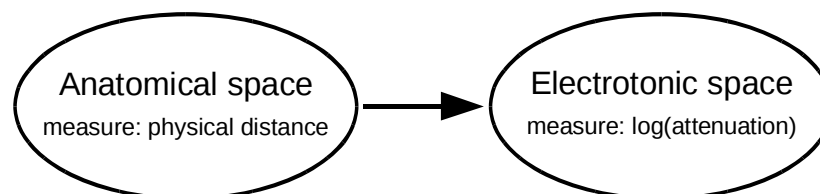
It isn't valid

| 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

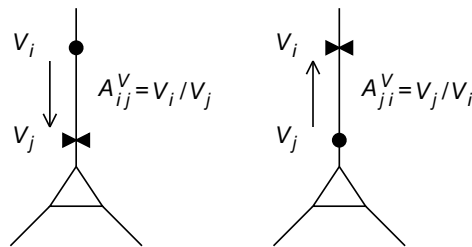
A transformation from anatomical to electrotonic space that

- 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

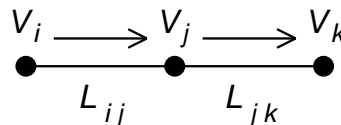
Functional definition of electrotonic distance

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

- ✓ 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

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

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



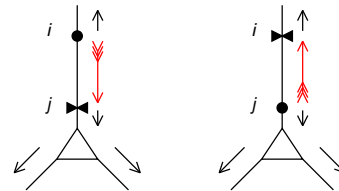
Using the Electrotonic Transformation

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

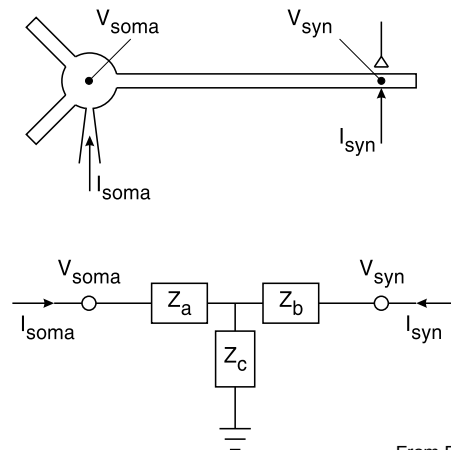
Q: What predicts how PSP amplitude at the soma varies with synaptic location?

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,

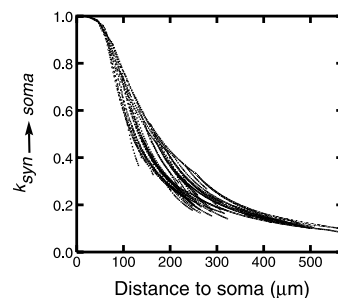
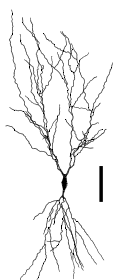
$V_{syn}(t)$ is independent of synaptic location

and

synapse to soma voltage transfer ratio

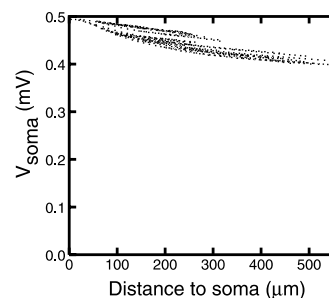
$$k_{syn \rightarrow soma} = 1 / A_{in}^V = Z_c / (Z_b + Z_c)$$

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

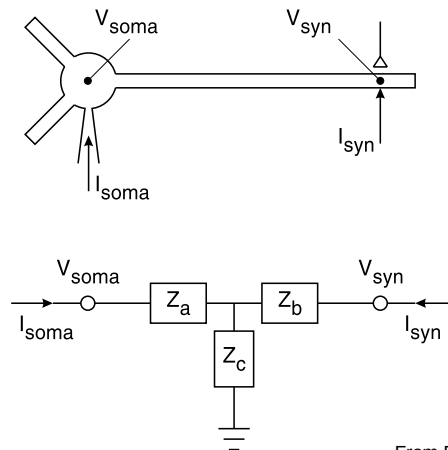
From Fig. 5 in Jaffe & Carnevale 1999

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

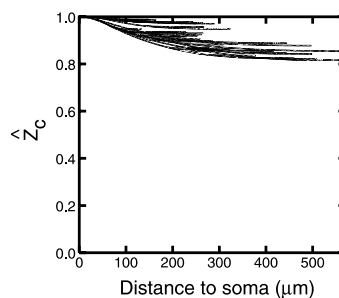
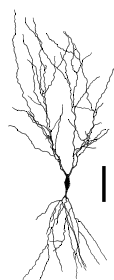
If synapses act like current sources,

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

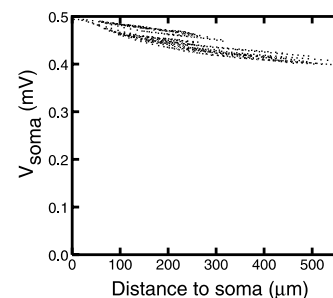
and

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

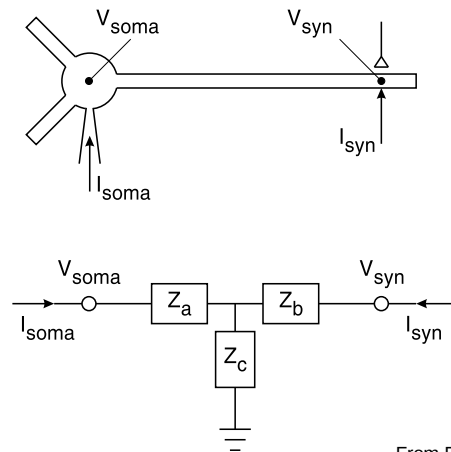


Somatic PSP generated by
conductance-change synapse

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

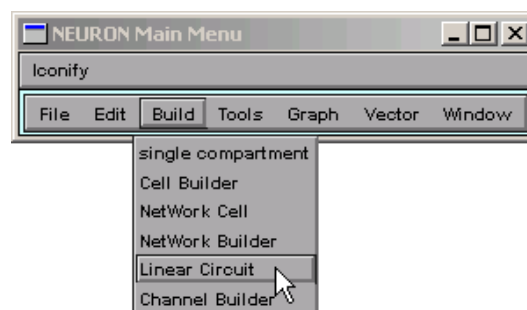
For building models that have linear circuit elements
and may also involve neurons

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

Potential applications include

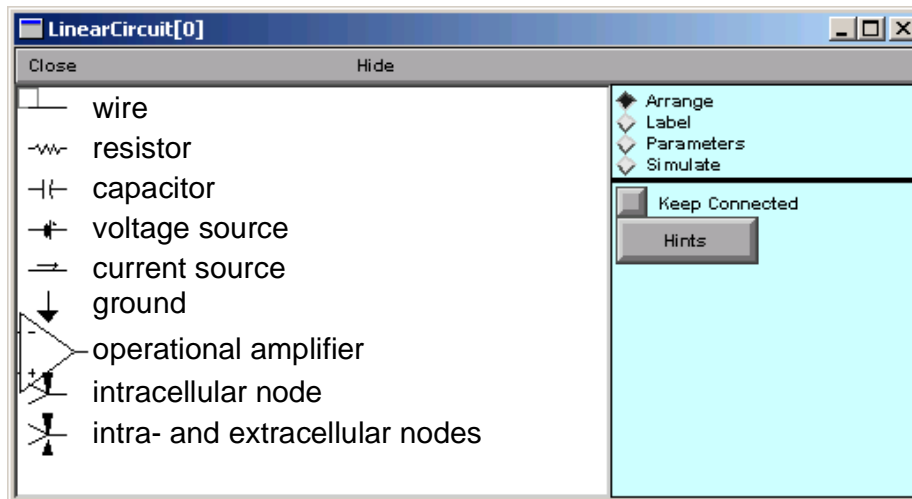
- effects and compensation of electrode R & C
- two-electrode voltage clamp
- ohmic and nonlinear gap junctions

1. Bring up a Linear Circuit Builder



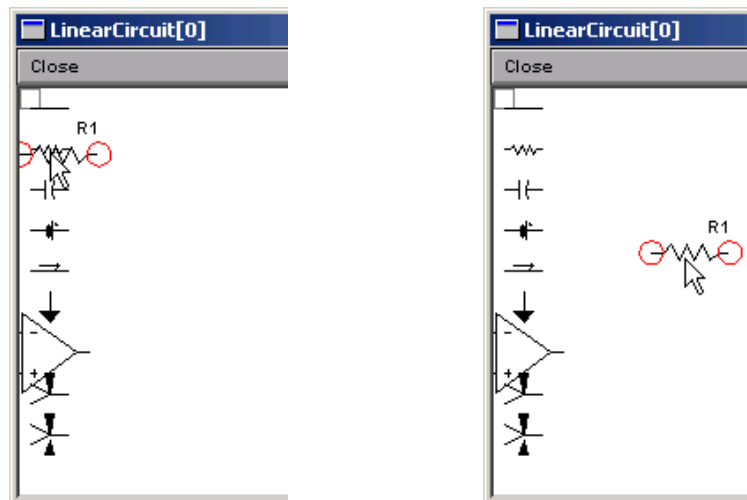
NEURON Main Menu / Build / Linear Circuit

The Linear Circuit Builder



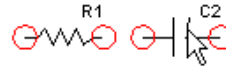
Arrange: spawn components

Click on palette and drag onto canvas

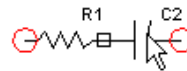


Arrange: connect components

Click and drag to
overlap red circles



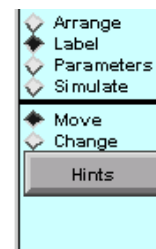
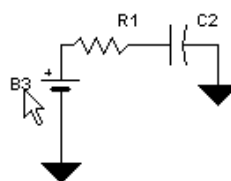
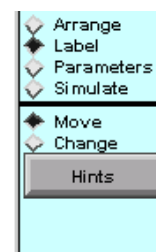
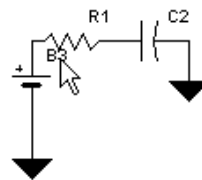
Black square is
"solder joint"



Pull apart to break connection

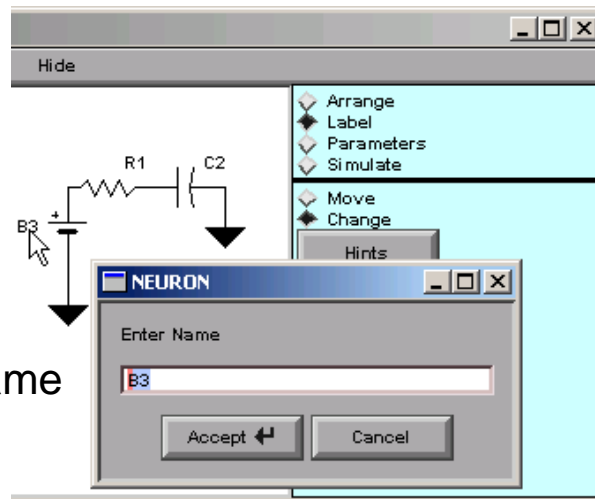
Label: move labels

Click and drag
to new location



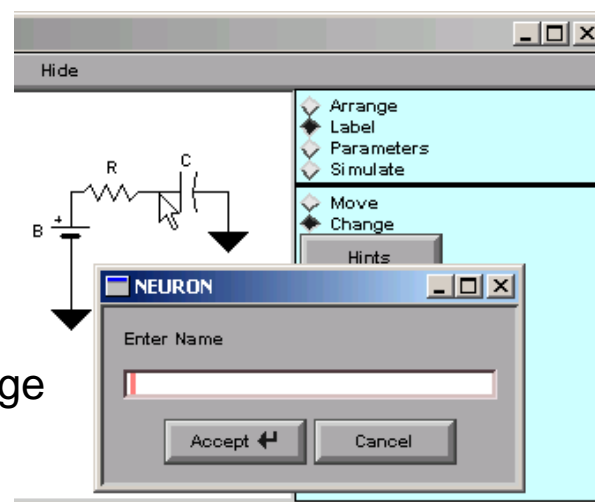
Label: change labels 1

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

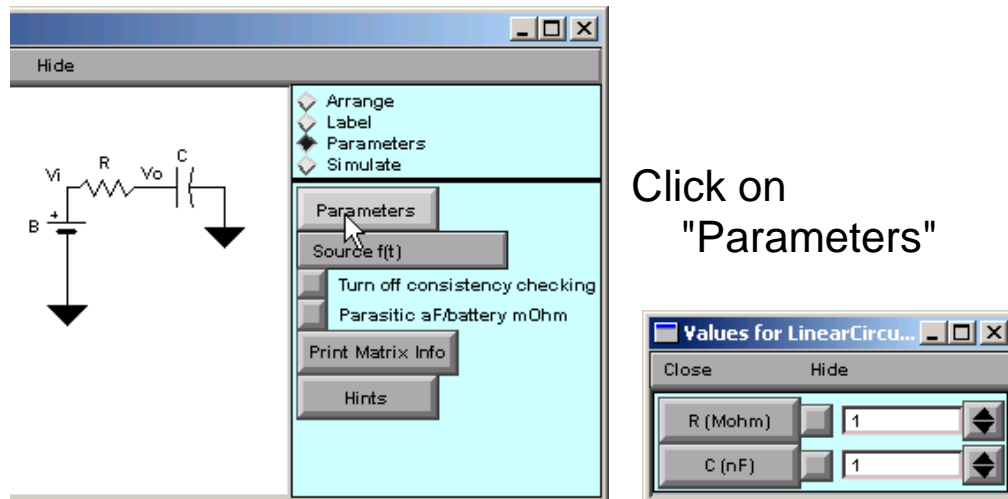


Label: change labels 2

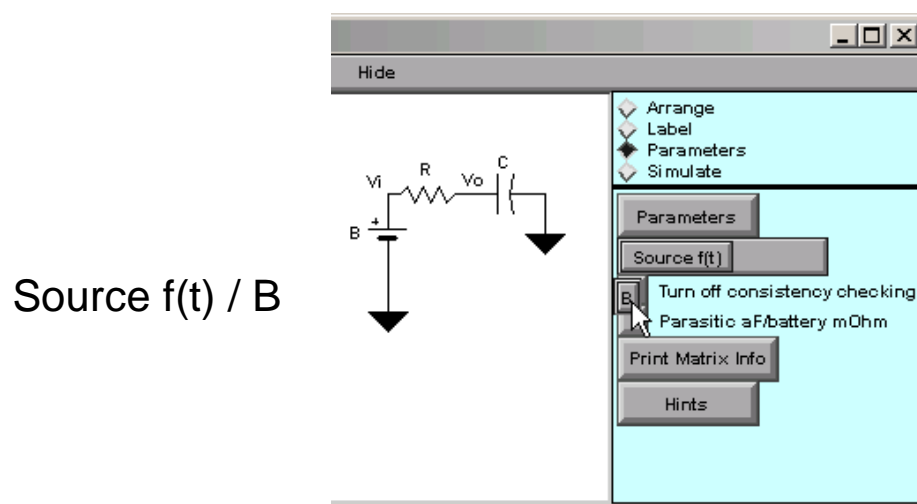
Click on a node . . .
 . . . to label a voltage



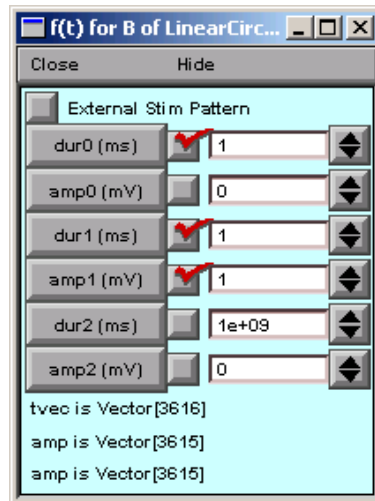
Parameters: non-source elements



Parameters: signal sources

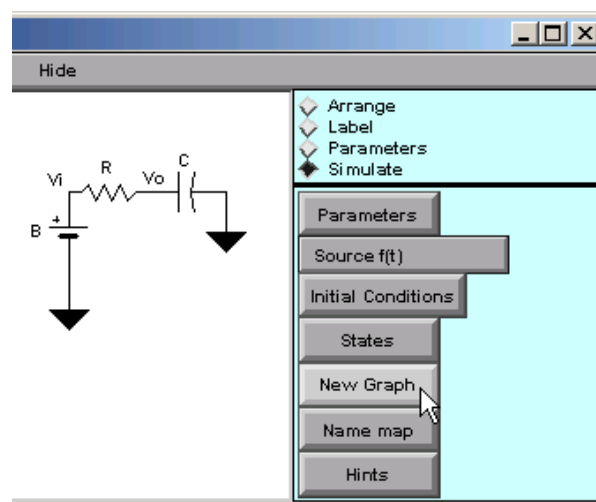


Parameters: signal sources *continued*

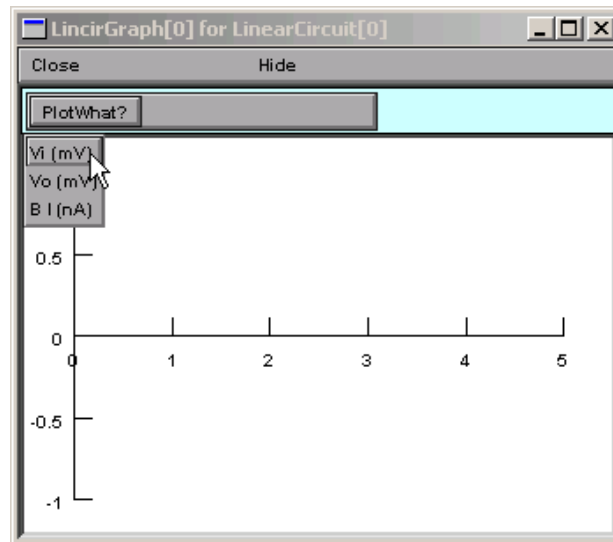


Configured

Simulate: creating a graph

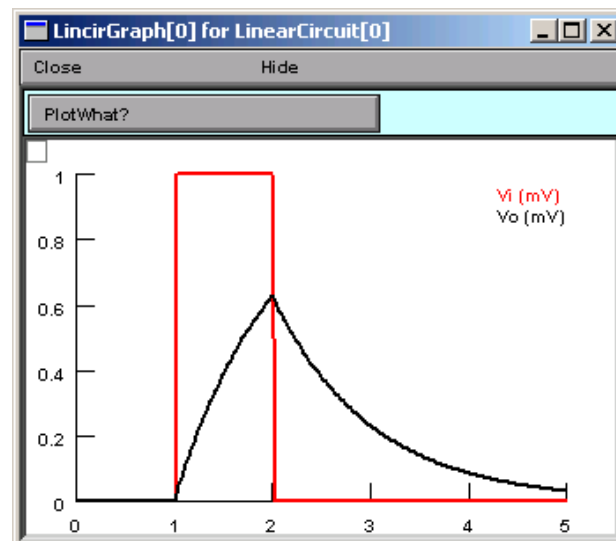


Simulate: specifying what to plot



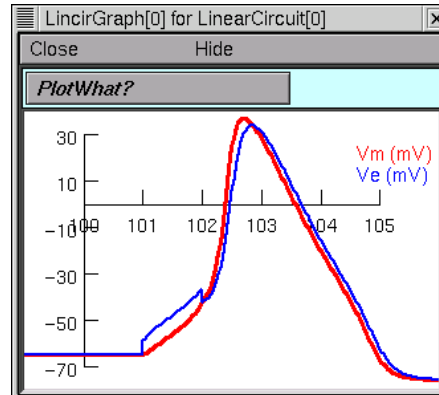
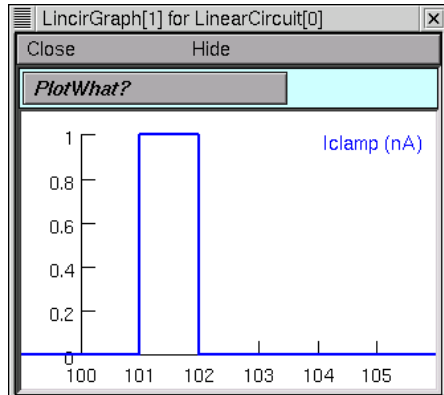
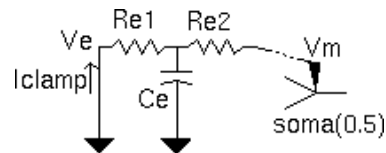
PlotWhat? / *variable_label*

Simulate: simulation results



After minor cosmetic changes

Patch clamp with electrode R and C



NEURON demo: dynamic clamp

