

## Table of Contents and Schedule of Presentations

NTC	Ted Carnevale
MLH	Michael Hines
WWL	Bill Lytton
TJS	Terry Sejnowski

Hands-on exercises are indicated by an asterisk \* in the Page column.  
Times shown are approximate, except for lunch.

### Saturday, 6/20 Morning session

Time	Speaker	Title	Page
9:00 AM	MLH	Welcome to the NEURON summer course	
9:15	NTC	Introduction to modeling with NEURON	5
9:45	NTC	Example: single compartment	9 *
10:45	Coffee Break		
11:00	NTC	Fundamental concepts	11
11:30	MLH	Hodgkin-Huxley axon	17 *
12:15	End of morning session		
12:30	Lunch		

### Afternoon session

1:30	TJS	Lost in parameter space: the art of modeling	
2:45	Coffee Break		
3:00	NTC	Elementary project management	19
3:30	NTC	Ball-stick model	21 *
5:00	End of afternoon session		

**Sunday, 6/21 Morning session**

<b>Time</b>	<b>Speaker</b>	<b>Title</b>	<b>Page</b>
9:00 AM	Q & A		
9:15	MLH	Numerical methods: accuracy, stability, speed	23
10:15	NTC	An outline for coding NEURON models	41
10:30	Coffee Break		
10:45	WWL	The hoc programming language	47 *
12:15	End of morning session		
12:30	Lunch		

**Afternoon session**

1:30	NTC	Model control: arbitrary forcing functions	61 *
2:30	NTC	Model control: simulation families	63 *
3:30	Coffee Break		
3:45	NTC	The Linear Circuit Builder	65 *
5:00	End of afternoon session		

**Monday, 6/22 Morning session**

<b>Time</b>	<b>Speaker</b>	<b>Title</b>	<b>Page</b>
9:00 AM	Q & A		
9:15	MLH	NMODL: the NEURON Model Description Language	73 *
10:30	Coffee Break		
10:45	NTC	ModelDB: a resource for reproducibility in computational neuroscience	83
12:15	End of morning session		

12:30 Lunch

### **Afternoon session**

1:30	NTC	Optimizing models with NEURON	89 *
3:15	Coffee Break		
3:30	MLH	NEURON + threads	91 *
5:00	End of afternoon session		

### **Tuesday, 6/23 Morning session**

<b>Time</b>	<b>Speaker</b>	<b>Title</b>	<b>Page</b>
9:00 AM	Q & A		
9:15	NTC	Networks: synapses, events, and artificial spiking cells	101 *
10:30	Coffee Break		
10:45	WWL	Networks: inhibitory synchronizing network	113 *
12:15	End of morning session		
12:30	Lunch		

### **Afternoon session**

1:30	MLH	Variable time steps and parameter discontinuities	135
3:00	Coffee Break		
3:15	MLH	Parallel computation: distributed network models	151
5:00	End of afternoon session		

**Wednesday, 6/24 Morning session**

<b>Time</b>	<b>Speaker</b>	<b>Title</b>	<b>Page</b>
9:00 AM	Q & A		
9:15	MLH	The standard run system	169
10:30	Coffee Break		
10:45	NTC	Initialization	171 *
12:15	End of morning session		
12:30	Lunch		

**Afternoon session**

1:30	NTC	Working with morphometric data	179 *
2:00	NTC	NEURON's tools for analyzing electrotonus	181 *
3:15	Coffee Break		
3:30	Review discussion		
4:45	Evaluation form (see last page in this booklet)		
5:00	End of afternoon session		
Optional	MLH	The Channel Builder	187
	MLH	Python + NEURON	195
Appendix	Translating network models to parallel hardware in NEURON		before survey
	NEURON and Python		

**Survey** **last page**



## The What and the Why of Neural Modeling

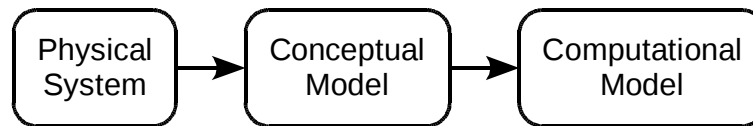
The moment-to-moment processing of information in the nervous system involves the propagation and interaction of electrical and chemical signals that are distributed in space and time.

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

## Topics

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

## From Physical System to Computational Model



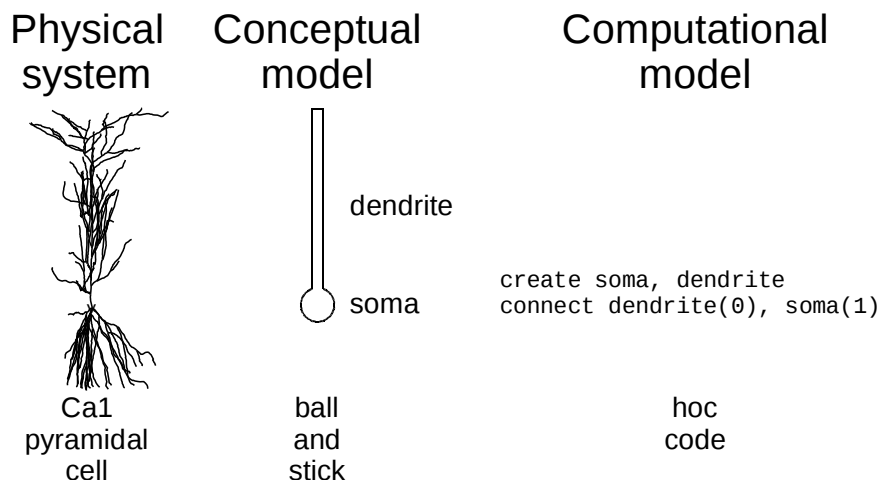
### Conceptual model

a simplified representation of the physical system

### Computational model

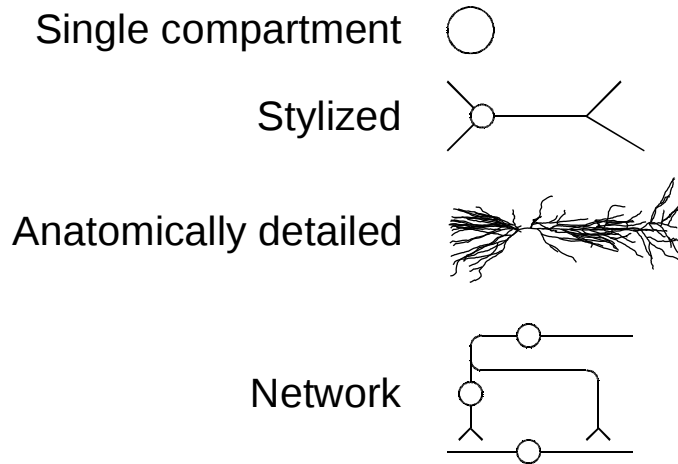
an accurate representation of the conceptual model

## From Physical System to Computational Model



## Hierarchies of Complexity

### Structure



## Hierarchies of Complexity

### Mechanism

Passive and Active currents

HH-style  
kinetic scheme

Synaptic transmission

continuous  
spike-triggered

Gap junctions

Extracellular fields, Linear circuits

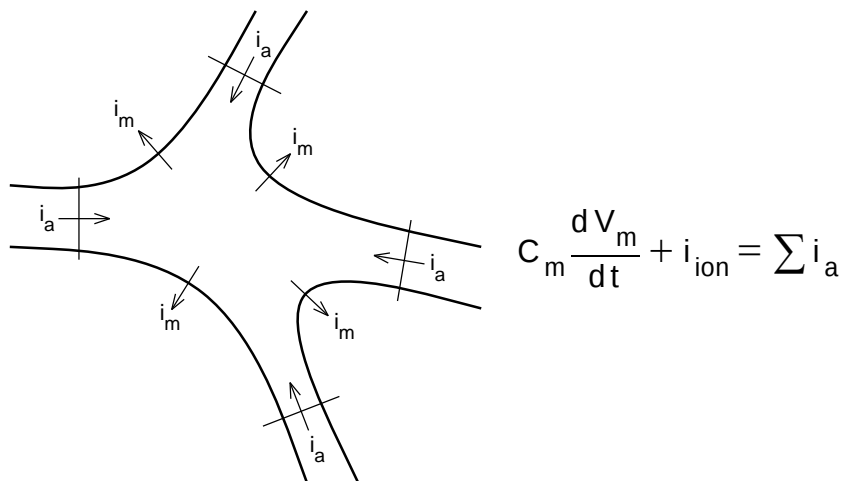
Diffusion, buffers, transport & exchange

Artificial spiking cells ("integrate & fire")

## Fundamental Concepts in NEURON

Signals	What moves	Driving force	What is conserved
Electrical	charge carriers	voltage gradient	charge
Chemical	solute	concentration gradient	mass

## Conservation of Charge



## Example: Single Compartment

Lipid bilayer (no channels)

Membrane with linear ion channels (passive leak)

Project goals:

- Run simulation
- Change stimulus intensity and duration
- Adjust graphical displays of simulation results
- Adjust dt and Points Plotted / ms



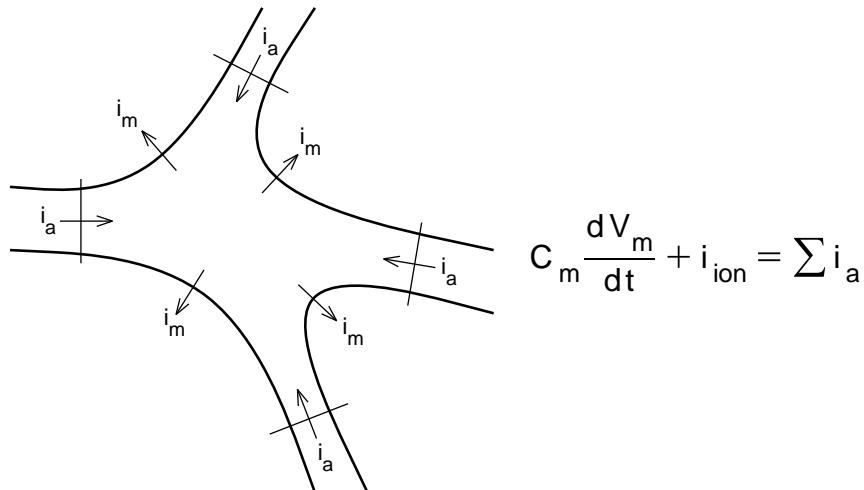
# Fundamental Concepts in NEURON

What equations are being solved?

How to separate the biology  
from computational details?

It's all about conceptual control . . .

## Conservation of Charge



## The Model Equations

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

$v_j$  membrane potential in compartment  $j$

$i_{ion_j}$  net transmembrane ionic current in compartment  $j$

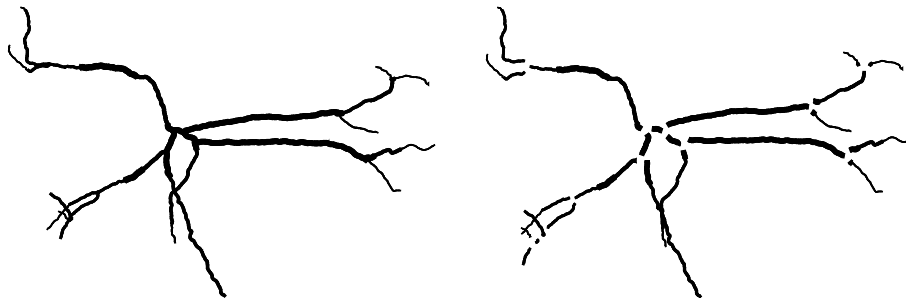
$c_j$  membrane capacitance of compartment  $j$

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

## Separating Anatomy and Biophysics from Purely Numerical Issues

section

a continuous length of unbranched cable



Anatomical data from A.I. Gulyás



**Syntax:** `create sectionname`

**Example:** `create soma, dend[3]`  
 creates one section called `soma`  
 and three sections called `dend[0]`, `dend[1]`, and `dend[2]`

**Assigning anatomical and biophysical attributes:**

```
soma {
    L = 50      // [um] length
    diam = 50   // [um] diameter
    insert hh   // Hodgkin-Huxley mechanism
}
for i=0,2 dend[i] {
    L = 200
    diam = 2
    insert pas  // passive channels
}
```

## Range Variables

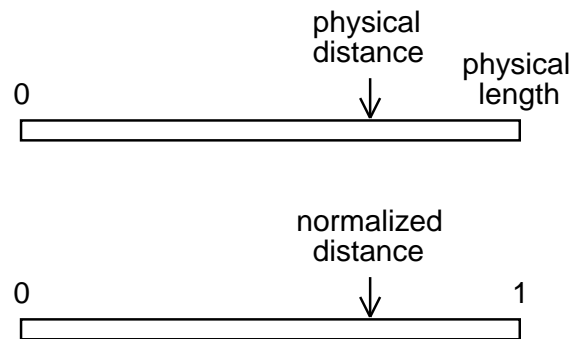
Name	Meaning	Units
diam	diameter	[ $\mu\text{m}$ ]
cm	specific membrane capacitance	[ $\mu\text{f}/\text{cm}^2$ ]
g_pas	specific conductance of the pas mechanism	[siemens/ $\text{cm}^2$ ]
v	membrane potential	[mV]

**range**

normalized position along the length of a section

$$0 \leq \text{range} \leq 1$$

any variable name can be used for range, e.g.  $x$

**Syntax:**

```
sectionname.rangevar(range)
```

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


**Examples:**

```
dend.v(0.5)
```


returns membrane potential at the middle of dend  
Shortcut: `dend.v`  
`dend for (x) print x*L, v(x)`  
prints physical distance and  $v$   
at each point in dend where  $v$  was calculated

**nseg**

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

nseg=1 

nseg=2 

nseg=3 

Example: axon nseg = 3

To test spatial resolution

forall nseg = nseg\*3

and repeat the simulation

## Units

Category	Variable	Units
Time	t	[ms]
Voltage	v	[mV]
Current	i	[mA/cm <sup>2</sup> ] (distributed) [nA] (point process)
Concentration	na i etc.	[mM]
Specific capacitance	cm	[ $\mu$ f/cm <sup>2</sup> ]
Length	diam, L	[ $\mu$ m]
Conductance	g	[S/cm <sup>2</sup> ] (distributed) [ $\mu$ S] (point process)
Cytoplasmic resistivity	Ra	[ $\Omega$ cm]
Resistance	ri	[10 <sup>6</sup> $\Omega$ ]



# Physical System



From <http://www.mbl.edu>

## Model

### Hodgkin-Huxley cable equations

$$\frac{D}{4R_a} \cdot \frac{\partial^2 V}{\partial x^2} = C_m \frac{\partial V}{\partial t} + \bar{g}_{na} m^3 h \cdot (V - E_{na}) + \bar{g}_k n^4 \cdot (V - E_k) + g_l \cdot (V - E_l)$$

$$\begin{aligned} \frac{dm}{dt} &= -\alpha_m m + \beta_m \cdot (1 - m) & \alpha_m &= \frac{.1(V+40)}{1 + e^{-.1(V+40)}} & \beta_m &= 4e^{-(V+65)/18} \\ \frac{dh}{dt} &= -\alpha_h h + \beta_h \cdot (1 - h) & \alpha_h &= .07e^{-.05(V+65)} & \beta_h &= \frac{1}{1 + e^{-.1(V+35)}} \\ \frac{dn}{dt} &= -\alpha_n n + \beta_n \cdot (1 - n) & \alpha_n &= \frac{.01(V+55)}{1 + e^{-1(V+55)}} & \beta_n &= .125e^{-(V+65)/80} \end{aligned}$$

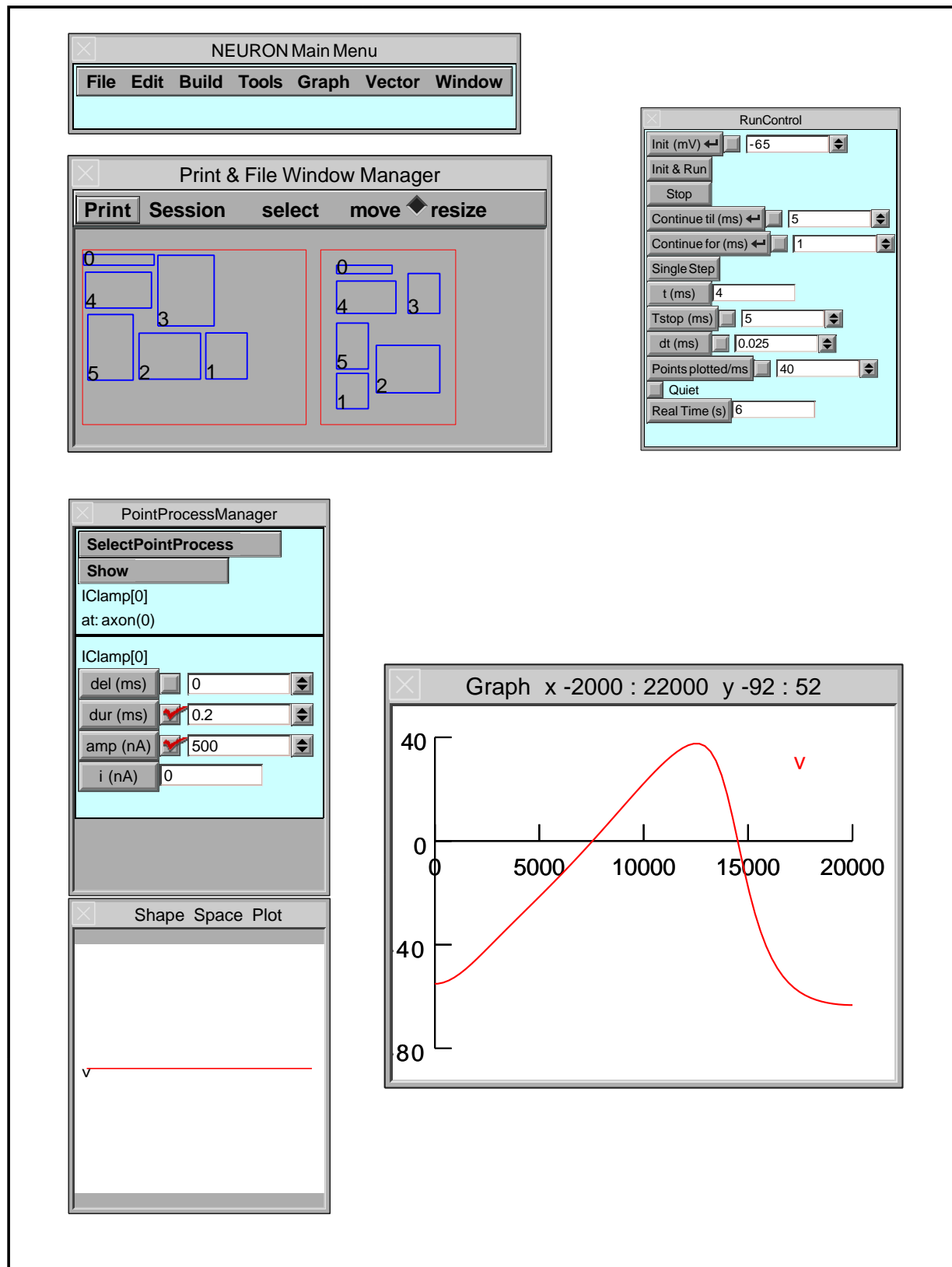
## Simulation

### Representation

```
create axon
axon {
    nseg = 50
    diam = 100
    L = 20000
    insert hh
}
```

### Run NEURON with above spec.

Exercises



## Elementary Project Management

Keep

specification of the model

separate from

specification of the interface

in order to

maximize clarity and reduce effort.

`modelspec.hoc`

- "virtual organism"
- specifies intrinsic properties of the model: topology, anatomy, biophysics

`rig.ses`

- "virtual experimental rig"
- initially empty or an innocuous statement, e.g.  
`print "ready"`
- eventually contains the custom interface (synapses, electrodes, graphs, run control, etc.)

`init.hoc`

- "administrative wrapper"

```
load_file("nrngui.hoc")  
load_file("modelspec.hoc")  
load_file("rig.ses")
```

## Usage

Execute `init.hoc`

UNIX: `nrngui init.hoc`

OS X: drag & drop `init.hoc` onto `nrngui`

MSWin: double click on `init.hoc`

Use **NEURONMainMenu** to customize interface

Attach synapses & electrodes

Set up graphs & run control

Save interface to `rig.ses`

Now executing `init.hoc` recreates the model  
and custom interface



## Example: Ball & Stick Model

Physical system: anatomically complex cell

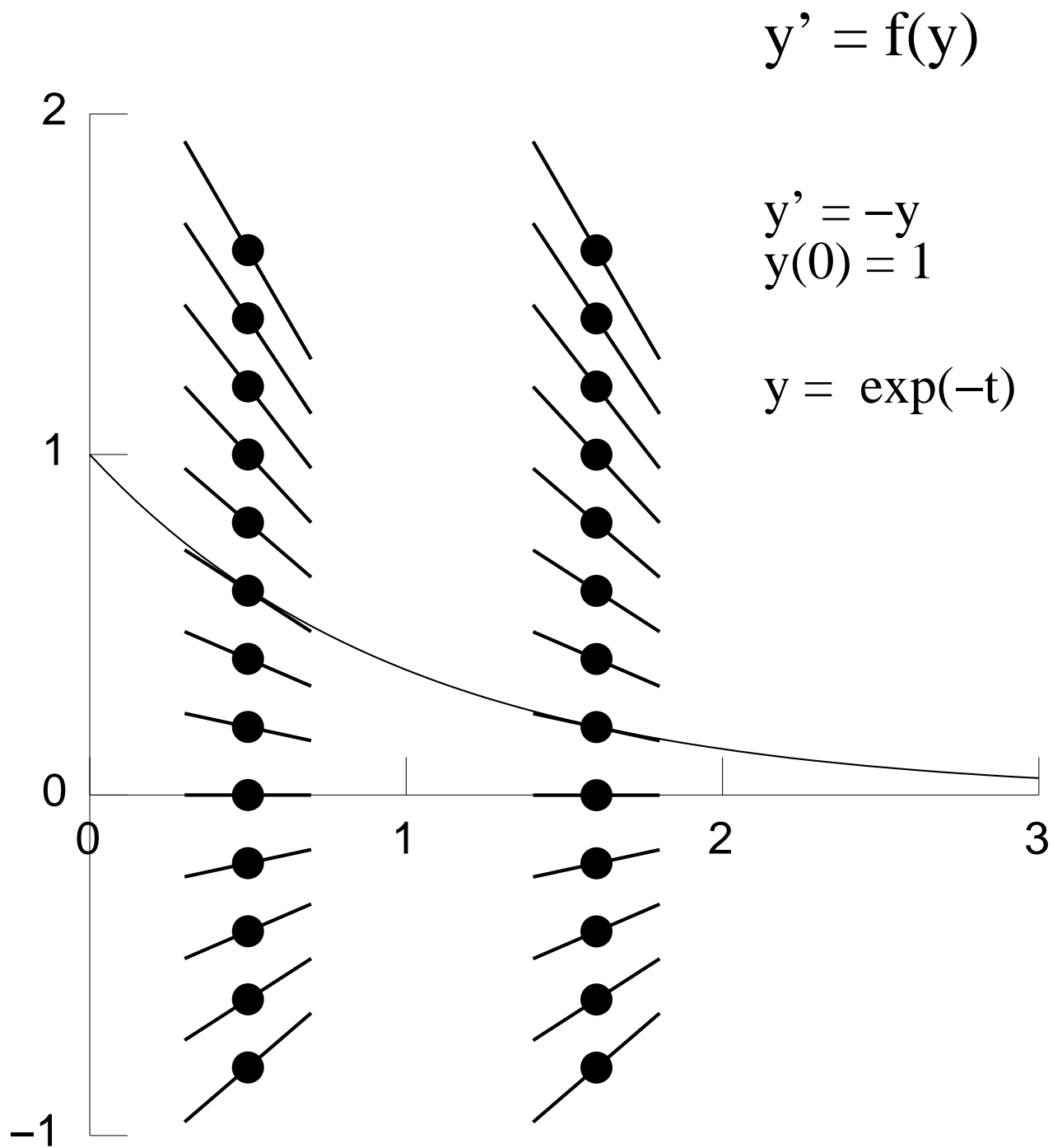
Conceptual model: "ball and stick" model

Computational model: soma + dendritic cylinder

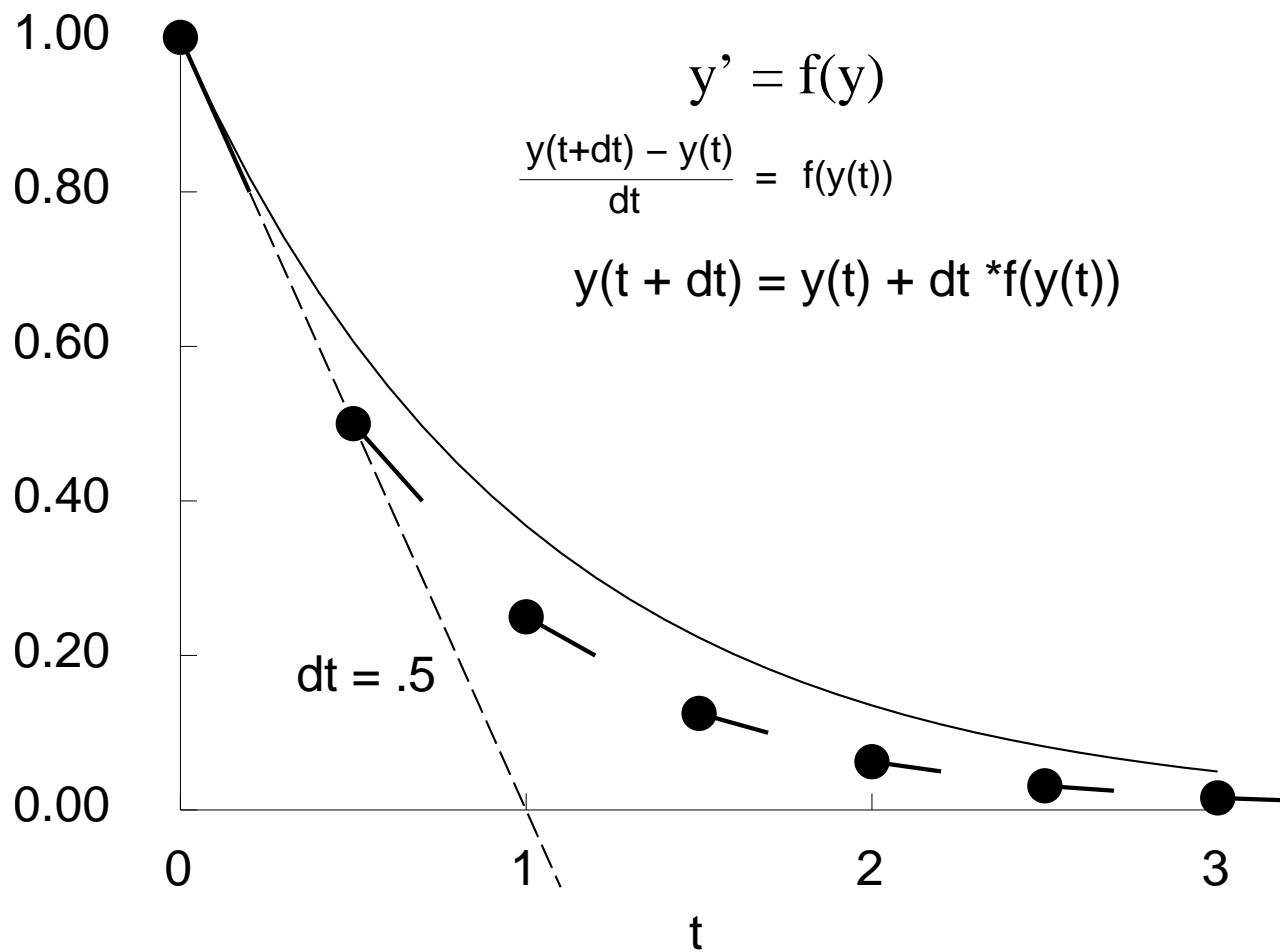
Project goals:

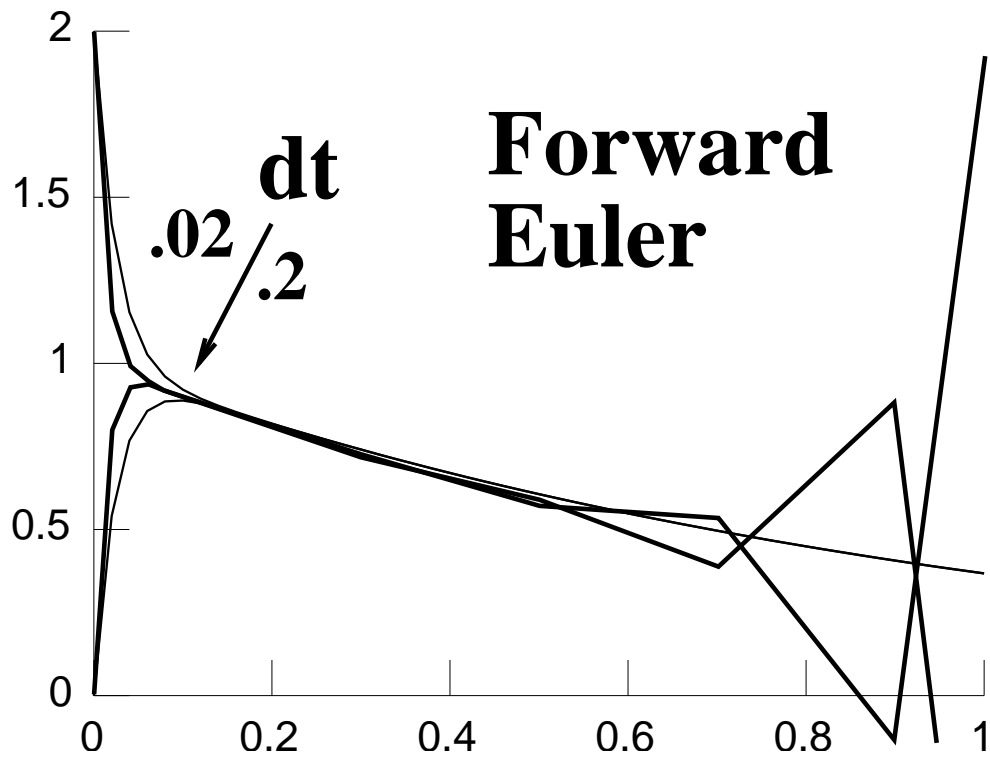
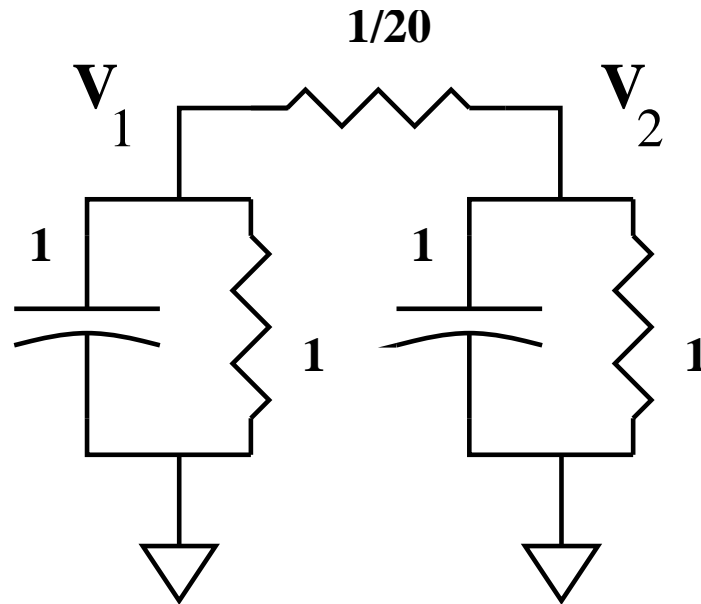
- Create model and custom GUI from scratch
- Learn how to use CellBuilder
- Use session files to save and retrieve user interface (elementary project management)
- Test model and simulation:  
structural integrity  
discretization of space and time



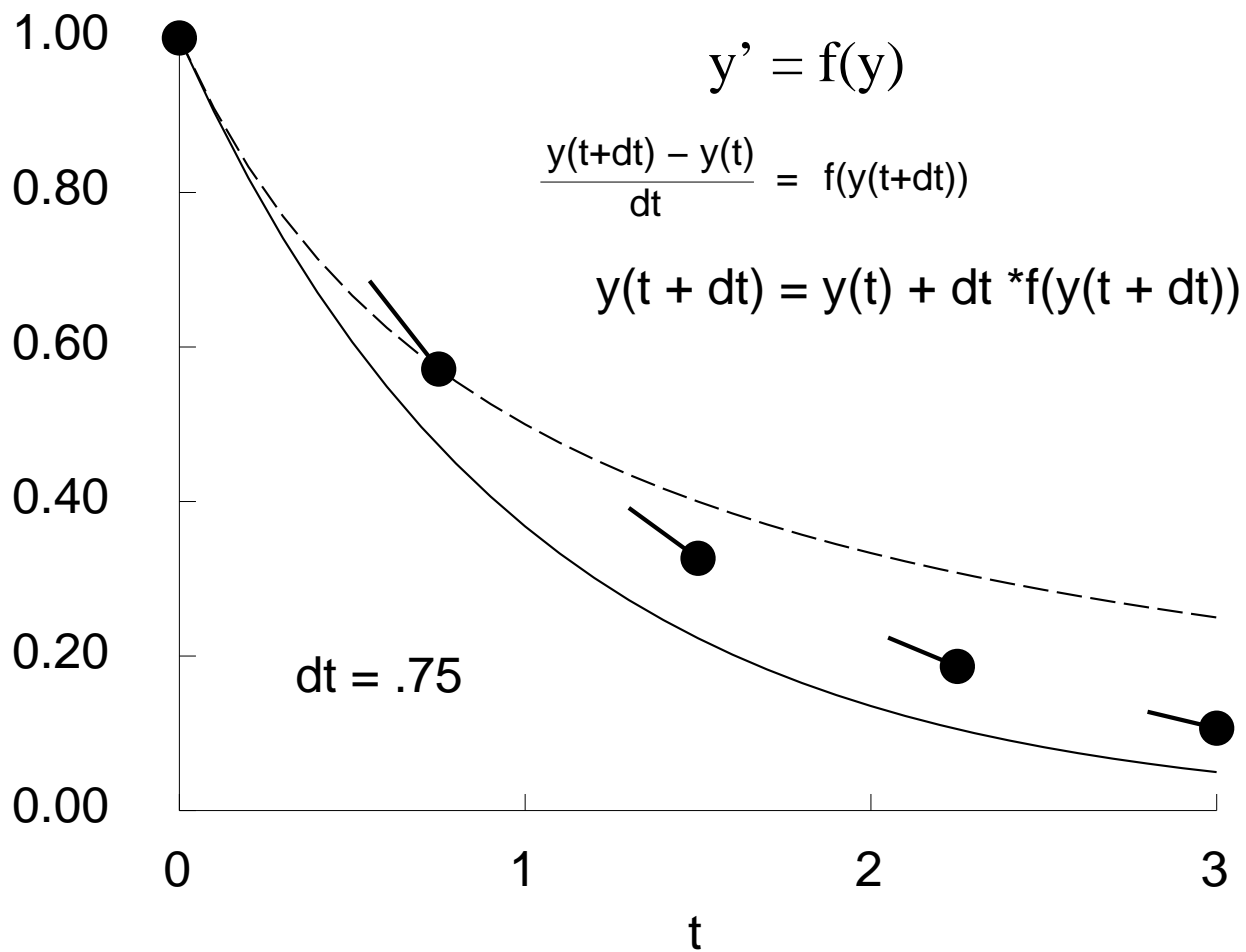


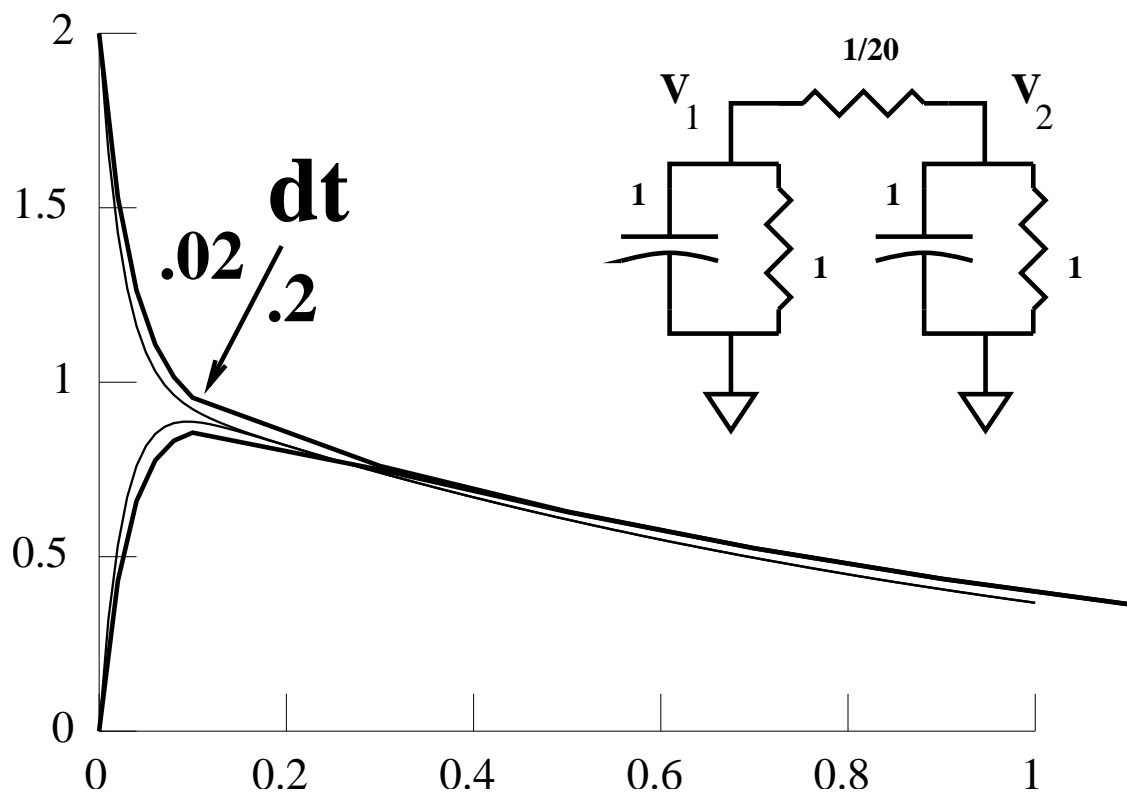
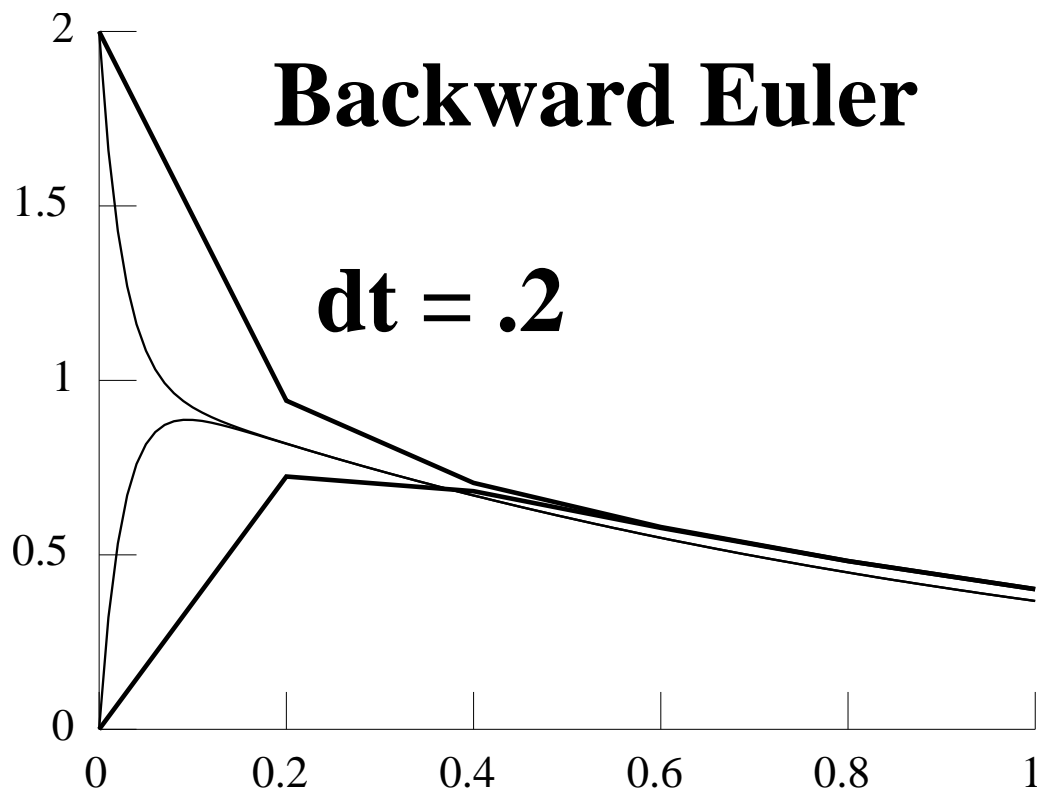
# Forward Euler



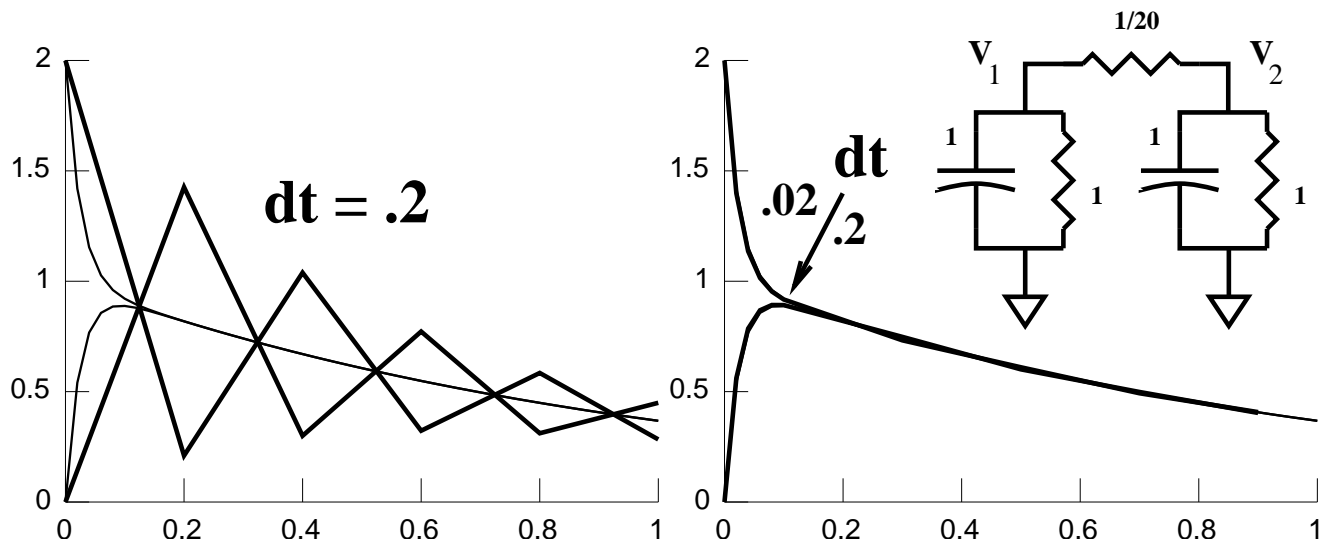
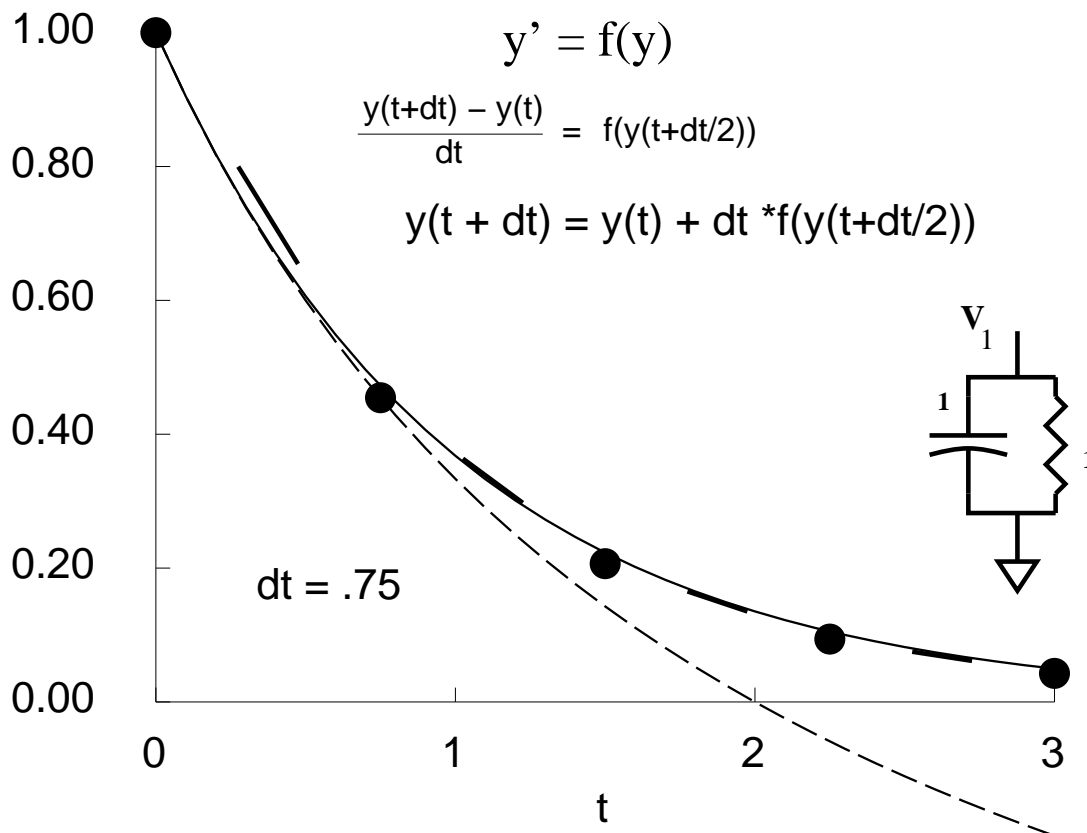


# Backward Euler



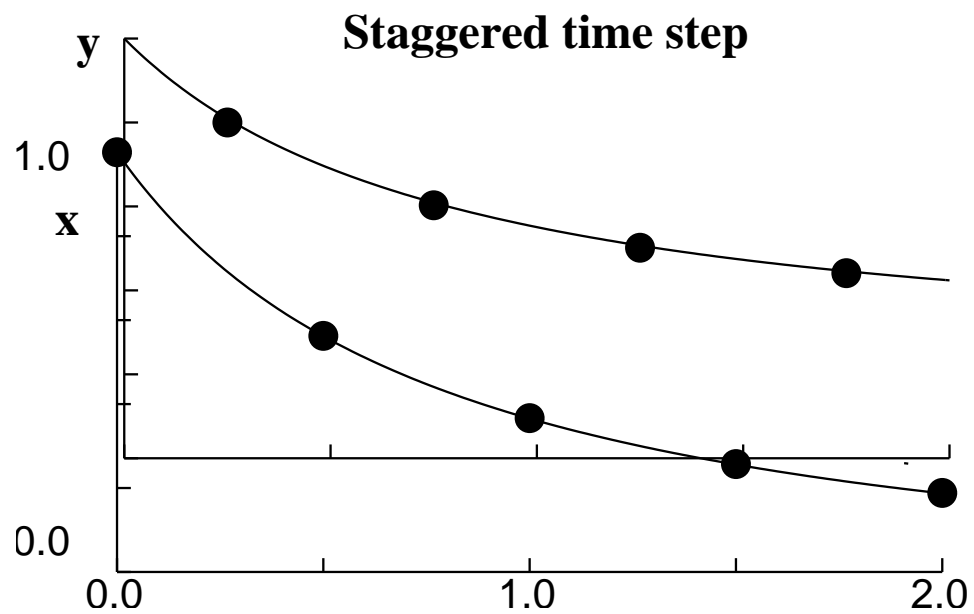
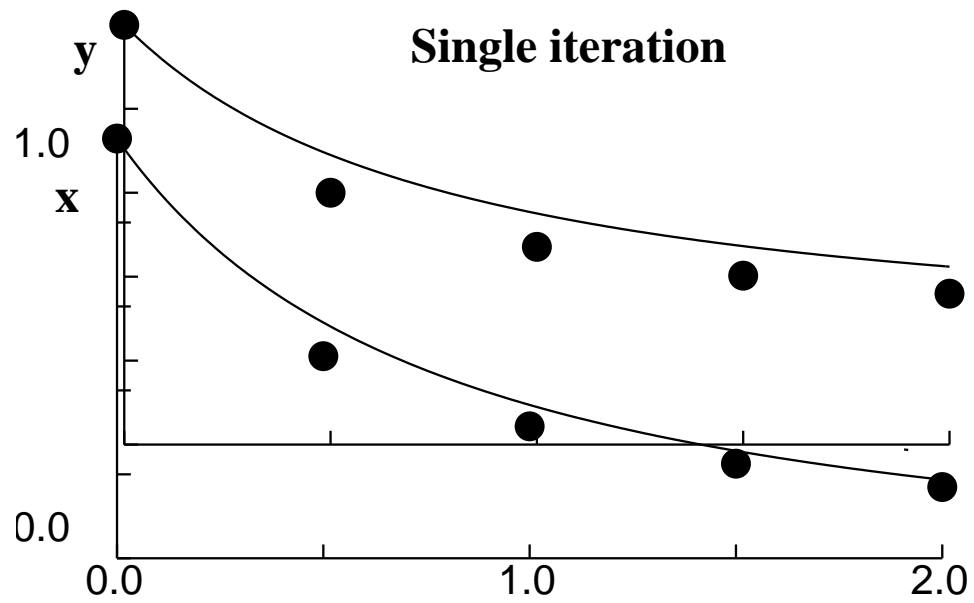


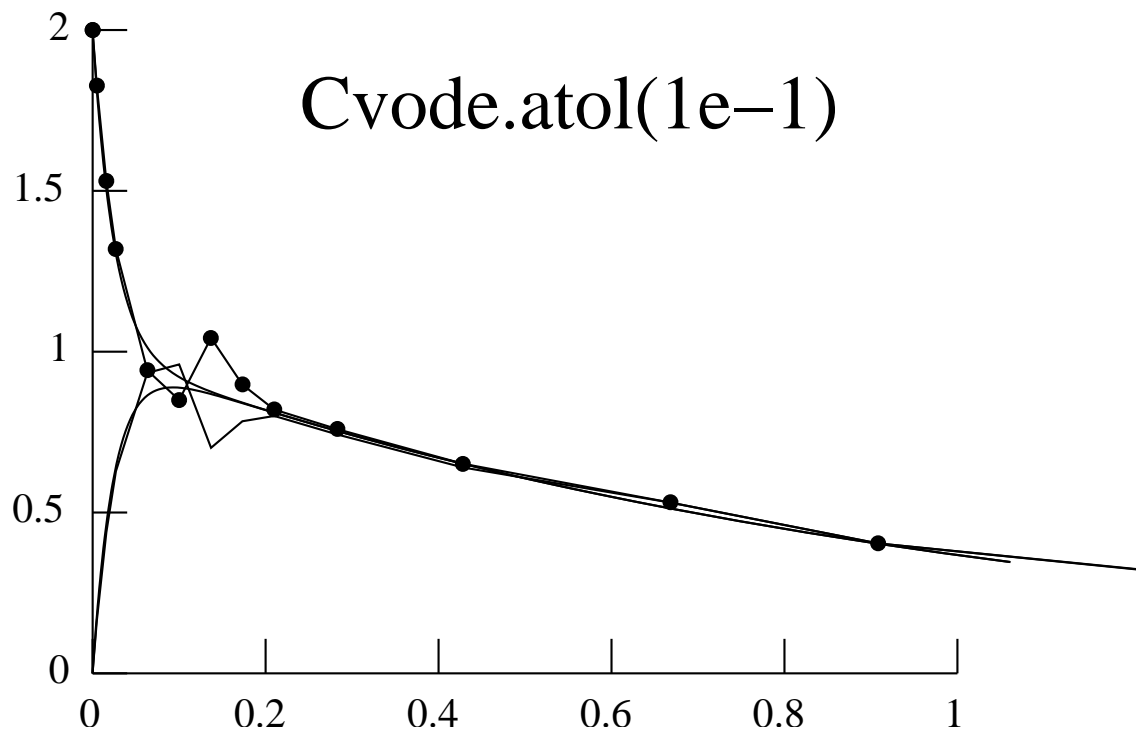
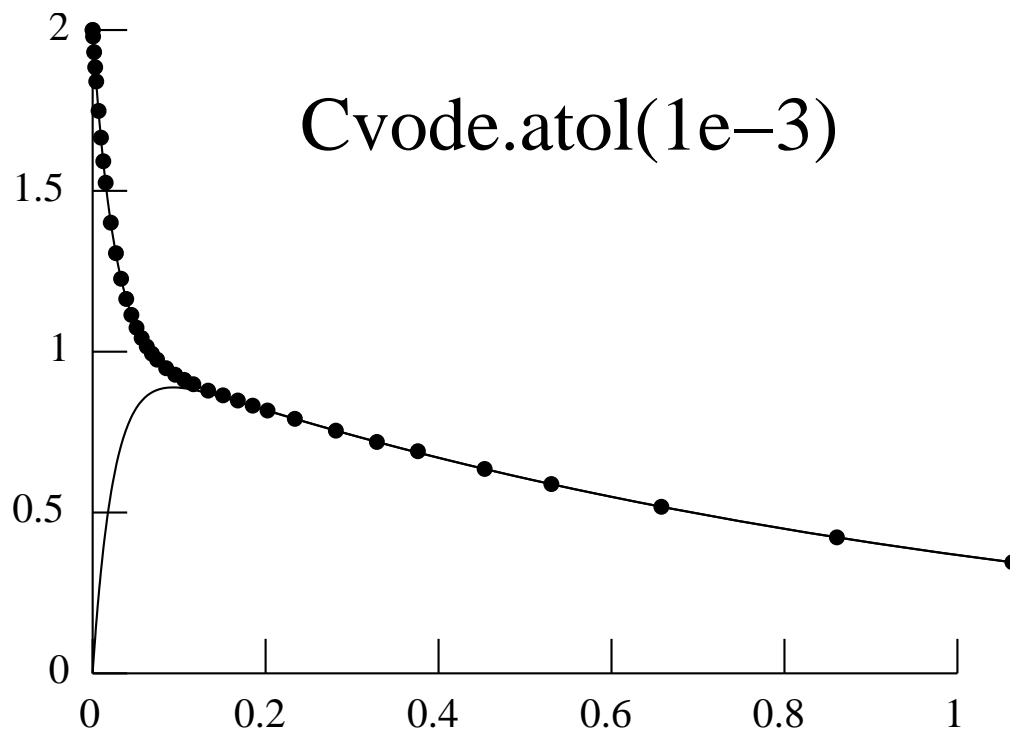
# Crank–Nicholson

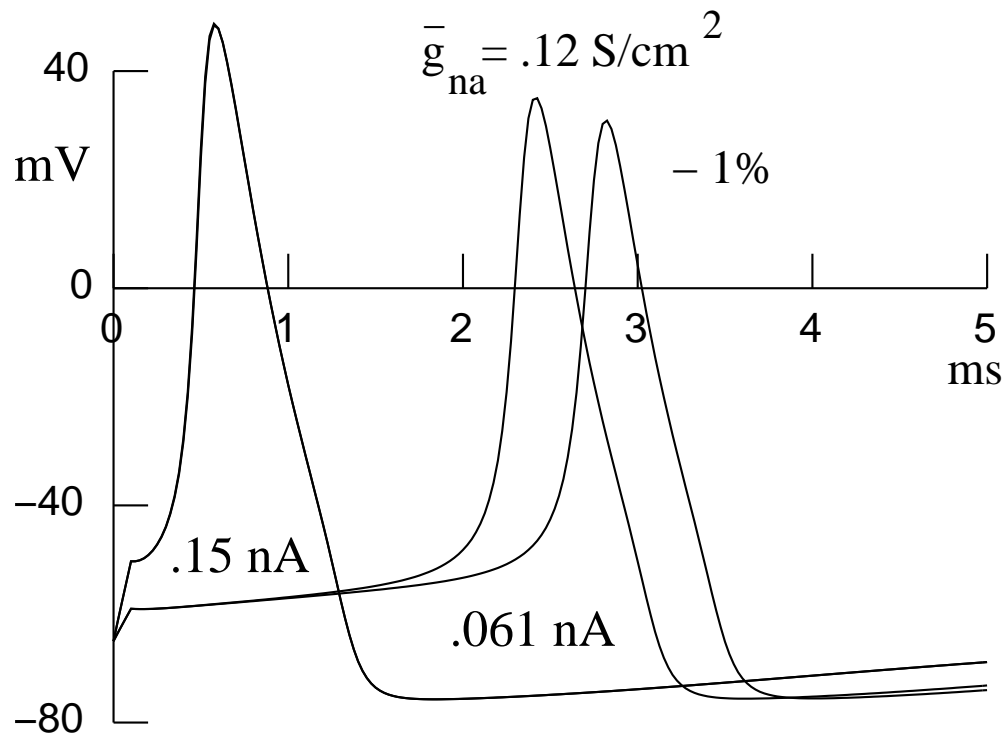
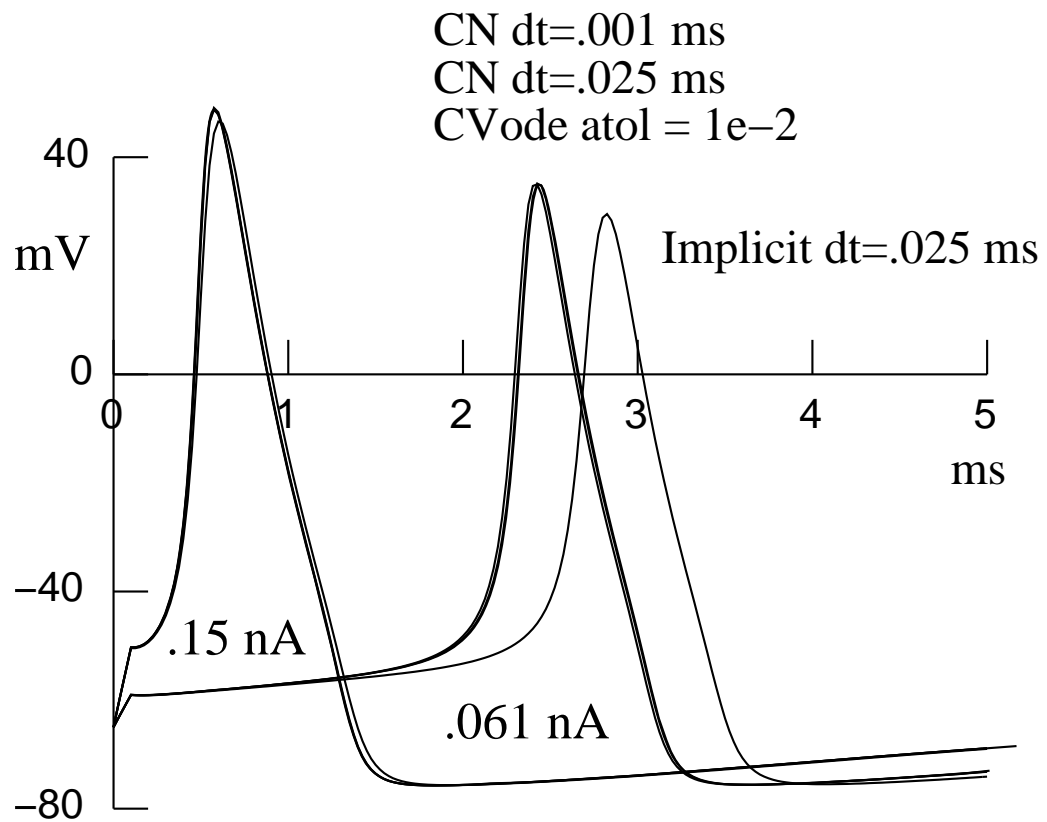


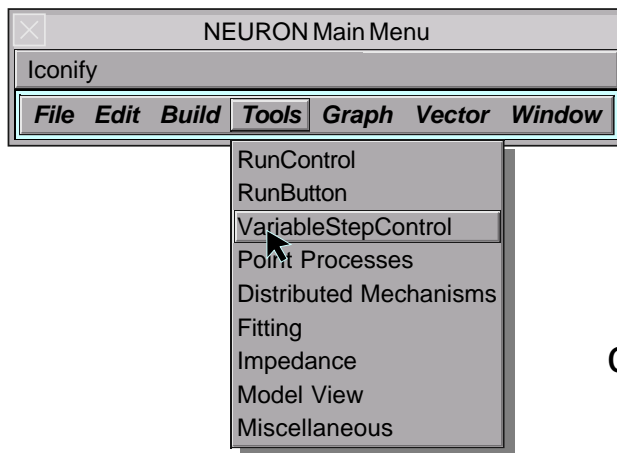


$$\begin{aligned}x' &= -1.4xy \\ y' &= -xy\end{aligned}$$

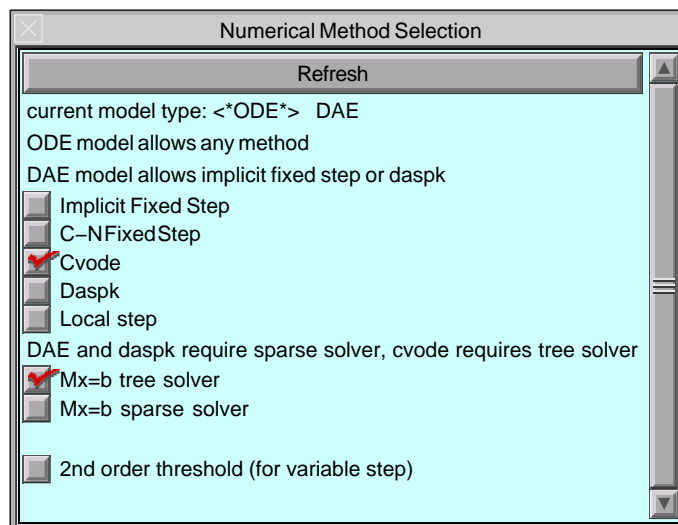
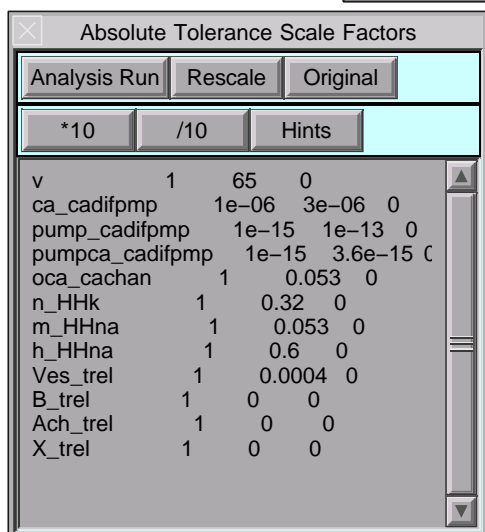
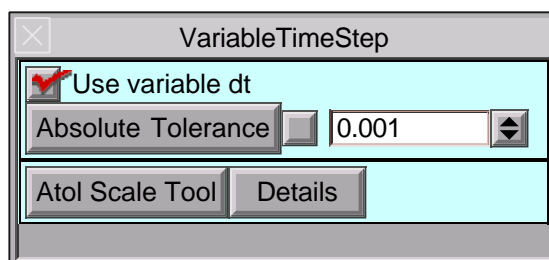
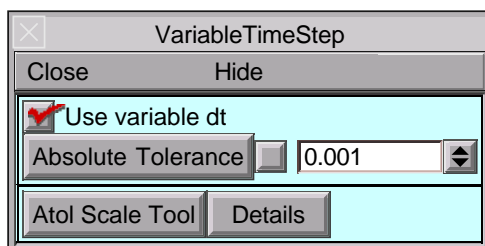


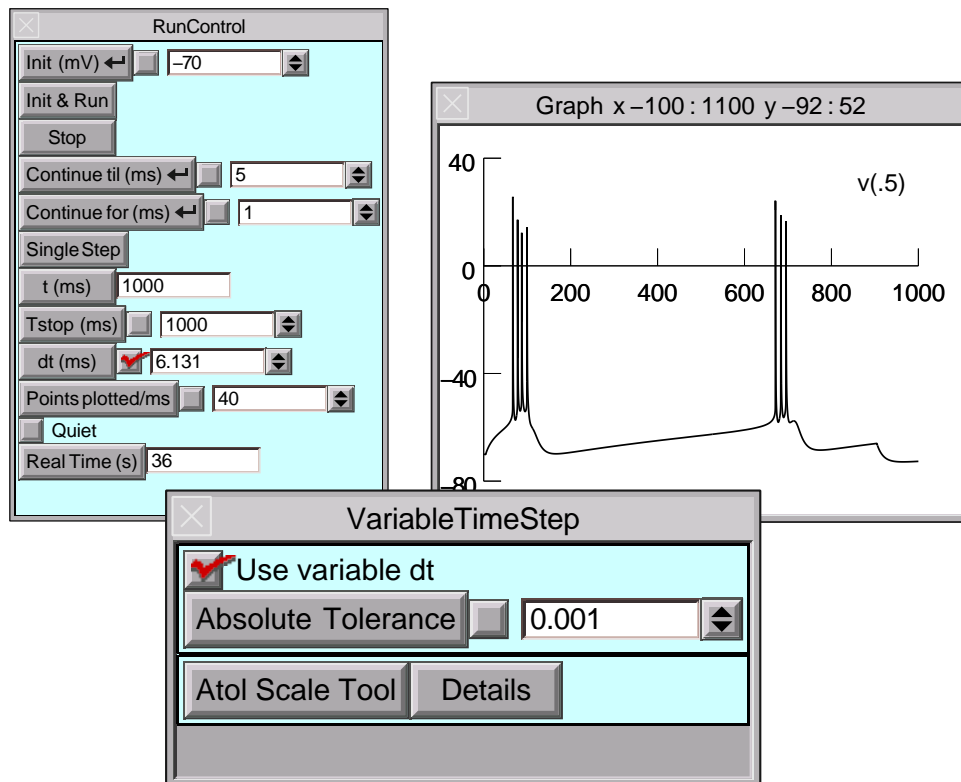
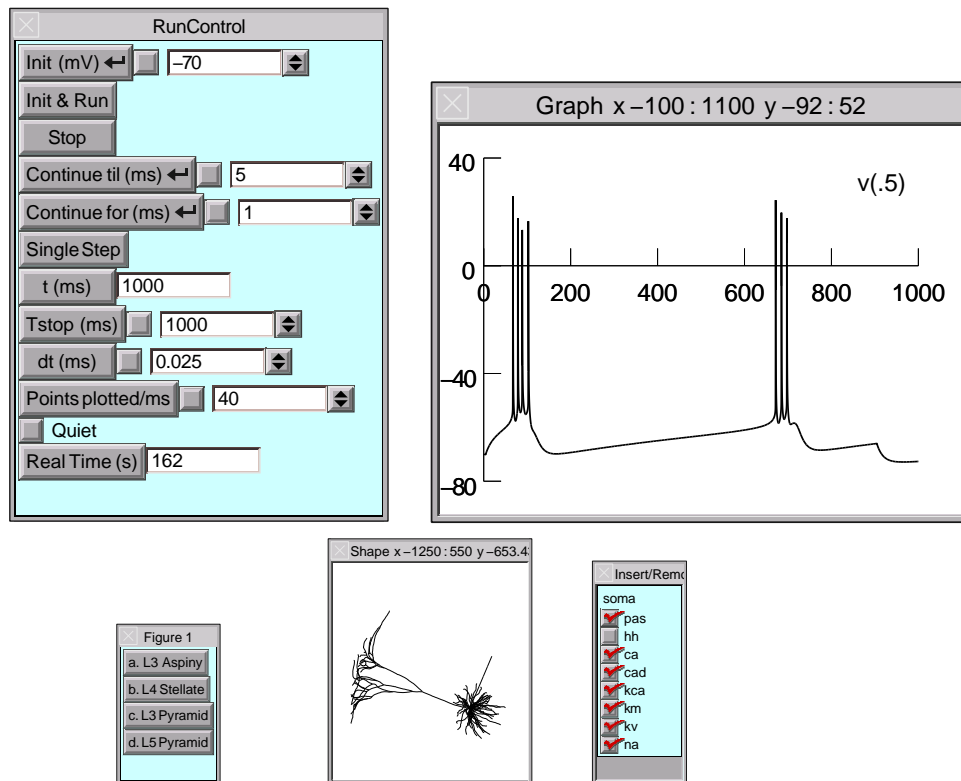


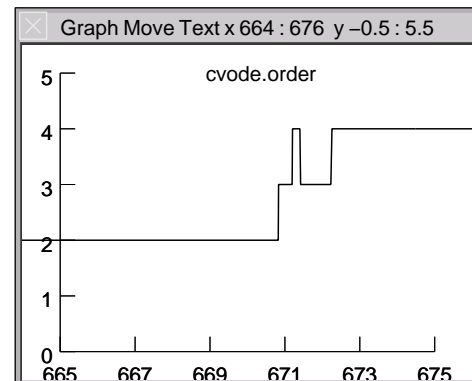
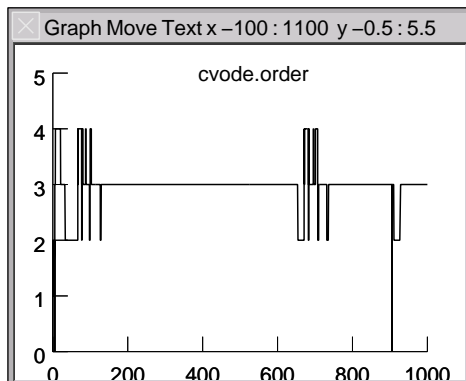
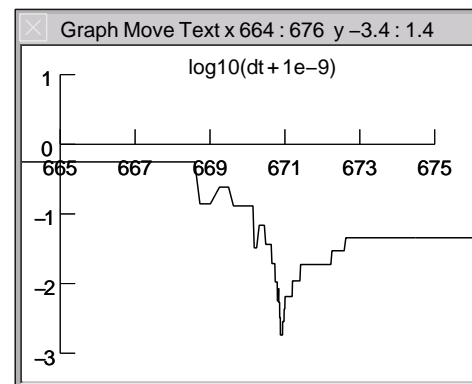
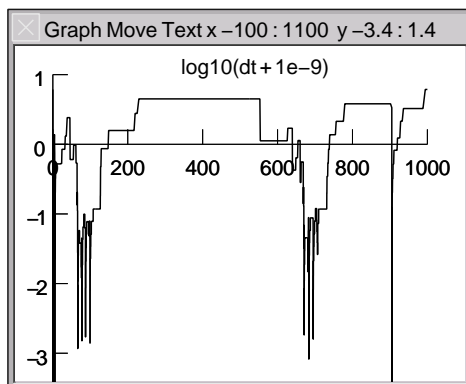
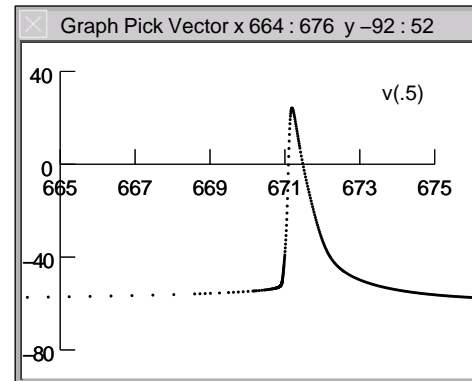
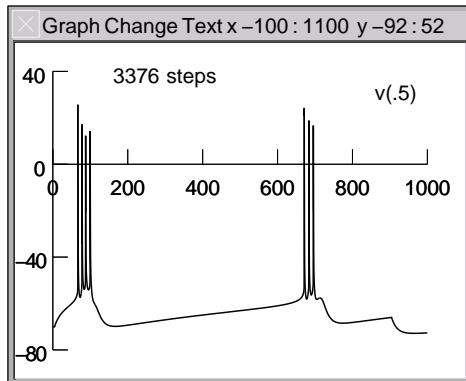


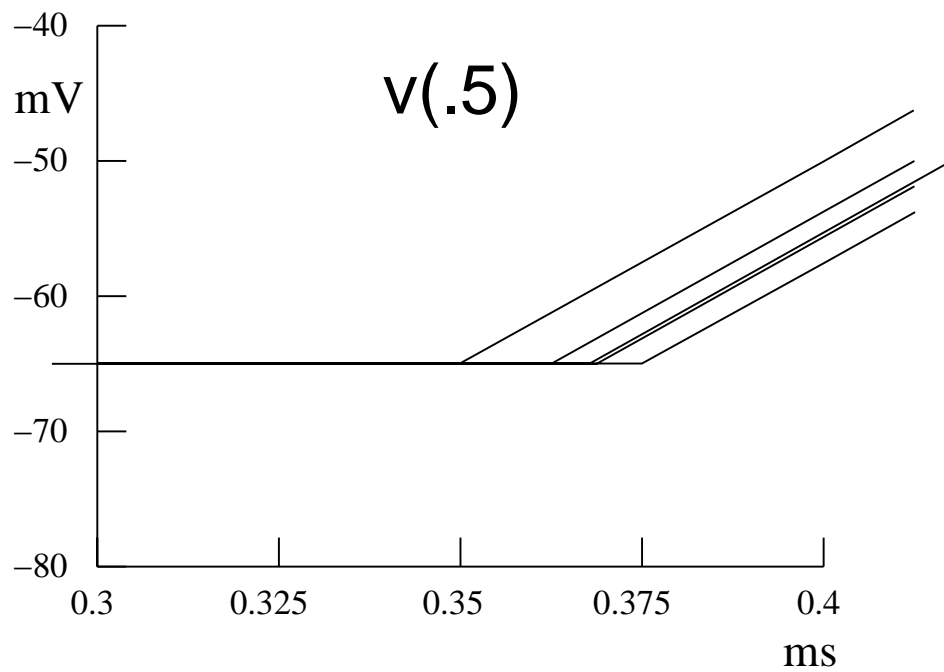
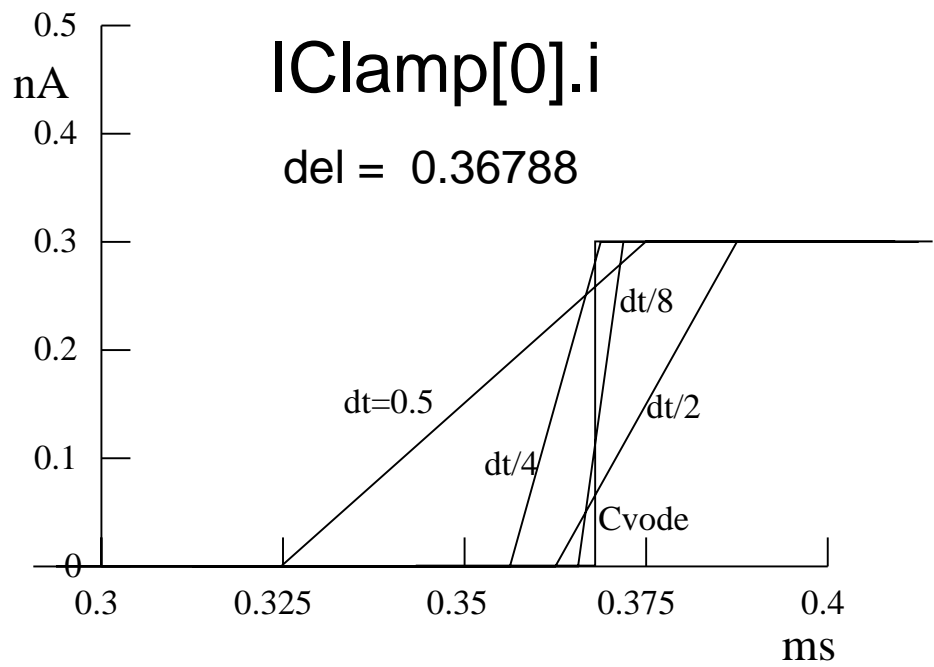


`cvar_active(1)`









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](#);

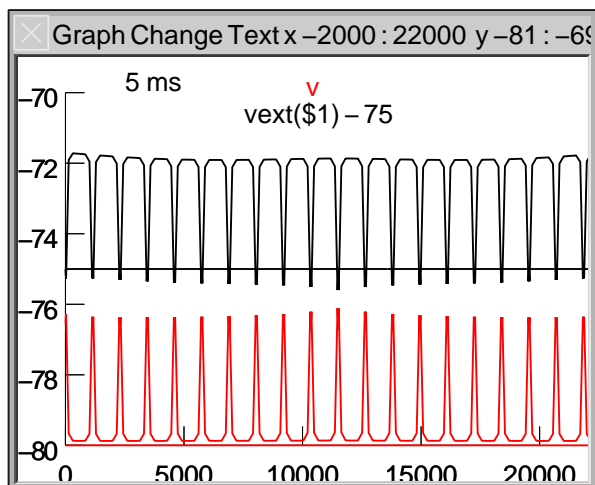
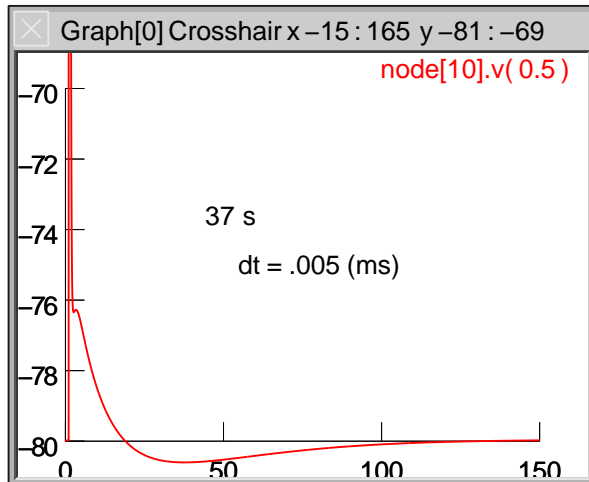
**Search NeuronDB** for information about: [Spinal motor neuron](#); [I Na,p](#); [I Na,t](#); [I K](#); [I Sodium](#); [I Potassium](#);

Model files	Download zip file	Auto-launch	<a href="#">Help downloading and running models</a>
<ul style="list-style-type: none"> <li><a href="#">MRGaxon</a></li> <li><a href="#">README</a></li> <li><a href="#">AXNODE.mod</a></li> <li><a href="#">MRGaxon.hoc</a></li> <li><a href="#">mosinit.hoc</a></li> <li><a href="#">MRGaxon.ses</a></li> </ul>	<p>SIMULATION OF PNS MYELINATED AXON</p> <p>This model is described in detail in:</p> <p>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.</p> <p>The model is set up to reproduce part of Fig 2A from this paper.</p> <p>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.</p> <p>Please contact <a href="mailto:mcintyre@bme.jhu.edu">mcintyre@bme.jhu.edu</a> if you have any questions about</p>		

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

[ModelDB Home](#) [SenseLab Home](#) [Help](#)  
 Questions, comments, problems? Email the [ModelDB Administrator](#)  
[How to cite ModelDB](#)

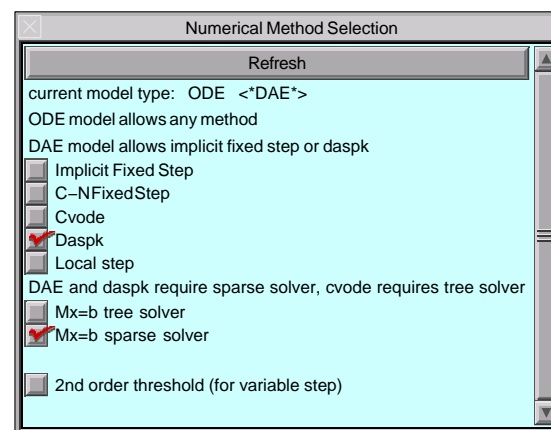
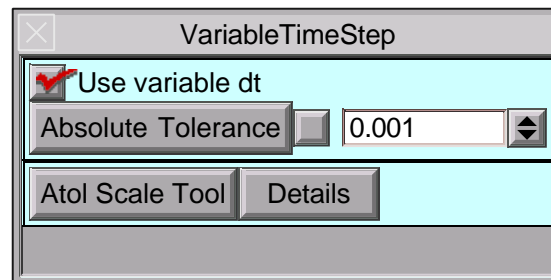
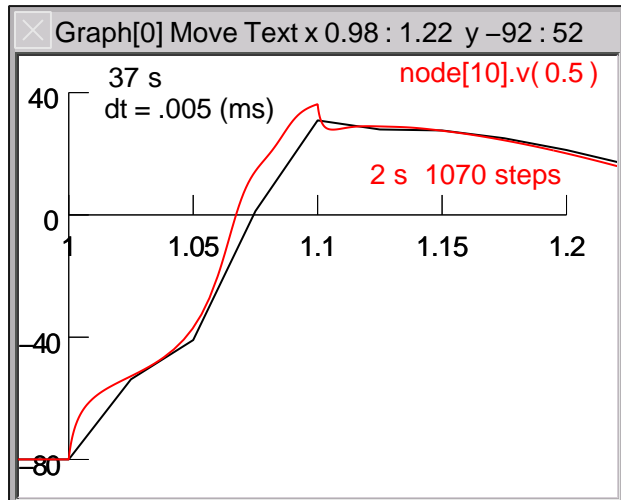
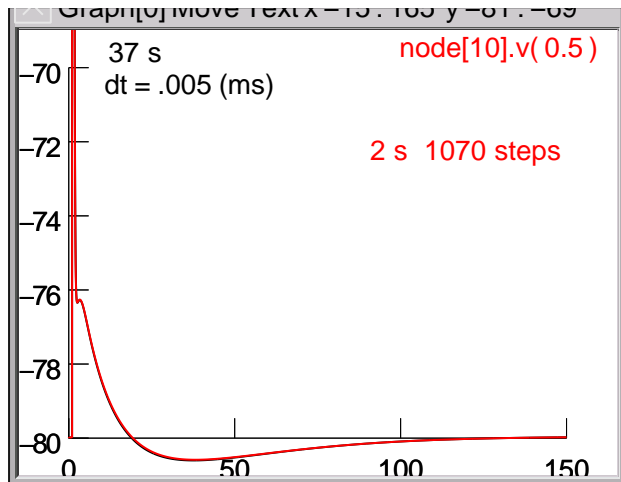


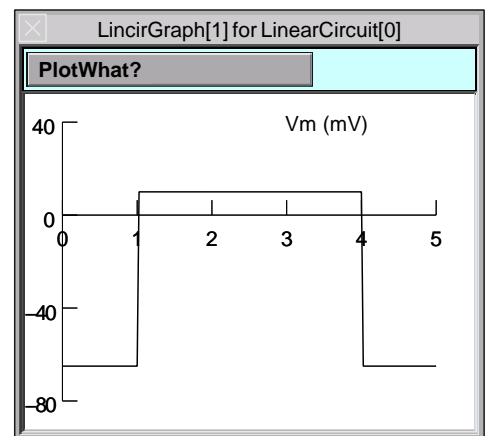
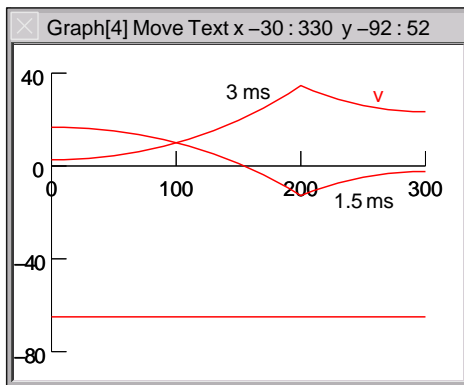
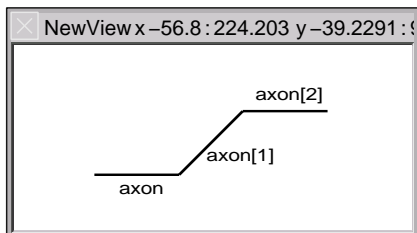
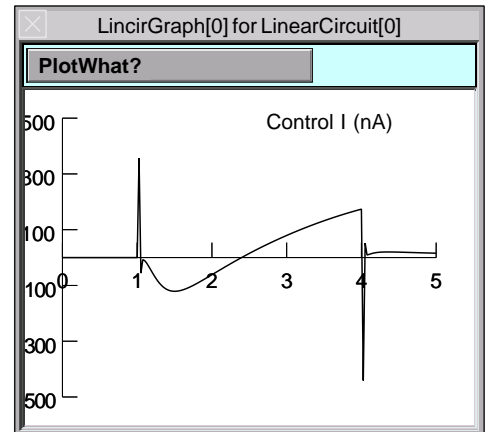
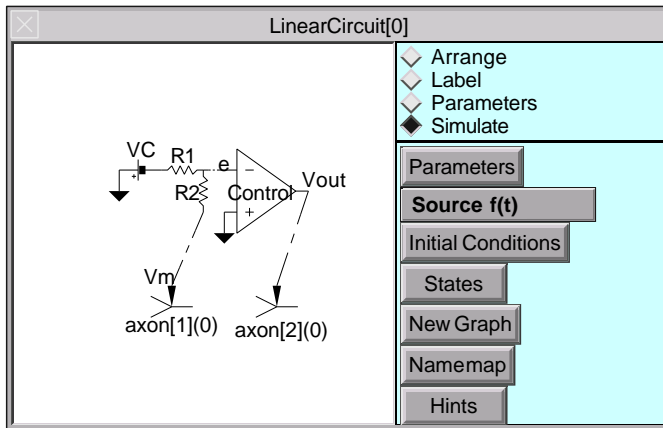


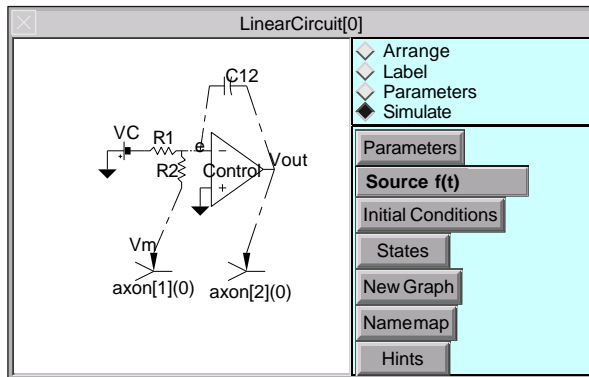
ModelView[0]

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







Values for LinearCircuit[0]

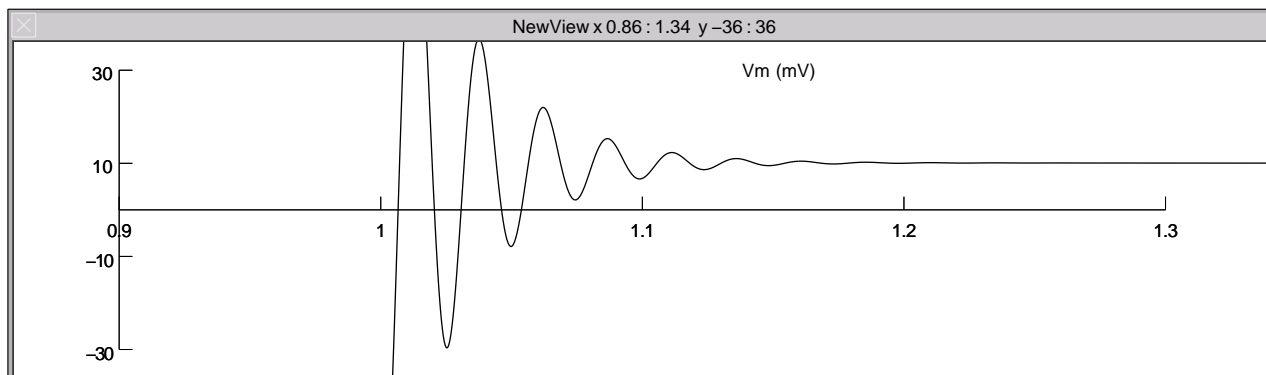
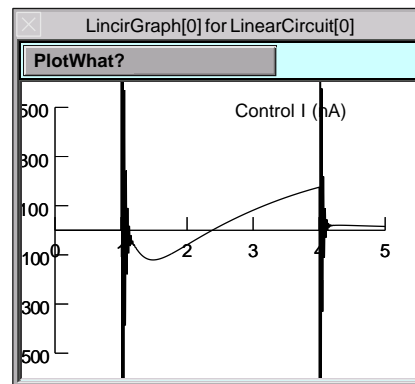
Control Gain	1e+05
Control Tau (ms)	0
R1 (Mohm)	1e+05
R2 (Mohm)	1e+05
C12 (nF)	1e-08

VariableTimeStep

☒ Use variable dt

Absolute Tolerance 0.001

Atol Scale Tool Details



## Specifying and using models with hoc

- Specify properties
  - by writing hoc code
  - and using the graphical interface

- Build user interface
  - session files

- Run tests
  - structural integrity
  - spatial grid
  - time steps

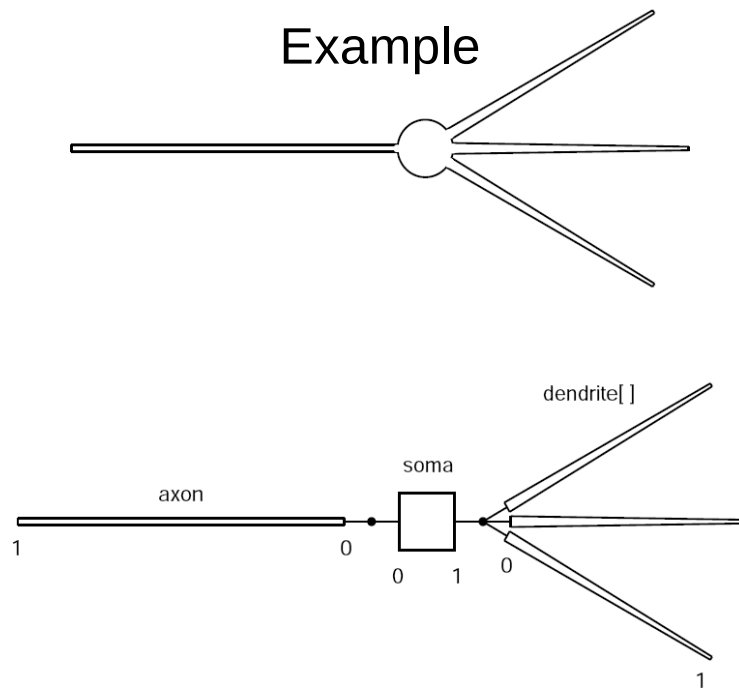
## Specifying a model with hoc

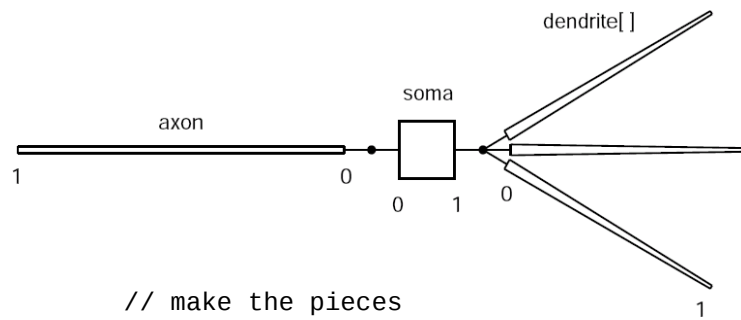
1. Establish model topology
2. Assign properties
3. Attach synapses and electrodes
4. Control simulation time course

## 1. Establish model topology

- Make the pieces (sections)  
create
- Assemble the pieces  
connect
- Specify the default section  
access

### Example





```
// make the pieces
create soma, axon, dendrite[3]

// specify default section
access soma

// assemble them
connect axon(0), soma(0)
for i=0,2 {
    connect dendrite[i](0), soma(1)
}
```

## 2. Assign properties

- Compartmentalization  
nseg
- Anatomical properties  
L, diam
- Biophysical properties  
insert (density mechanisms)

```
soma {  
    nseg = 1  
    L = 50      // [um] length  
    diam = 50   // [um] diameter  
    insert hh   // Hodgkin-Huxley currents  
}  
  
axon {  
    nseg = 21   // odd so there is a node at 0.5  
    L = 1000  
    diam = 1  
    insert hh  
}  
  
for i=0,2 dendrite[i] {  
    nseg = 5  
    L = 200  
    diam(0:1) = 10:3 // taper  
    insert pas       // passive membrane  
}  
  
forall Ra = 60 // [ohm cm]
```

### Range variables

Vary continuously in space along the length of a section

Examples: v, cm, diam

### Section variables

Pertain to an entire section

Examples: Ra (cytoplasmic resistivity), L, nseg

### Global variables

Same across all sections

Examples: celsius, t and dt (fixed time step integration)



### 3. Attach synapses and electrodes

```
objref stim
// attach to middle of soma
soma stim = new IClamp(0.5)

stim.del = 1    // [ms] delay
stim.dur = 0.1  // [ms] duration
stim.amp = 60   // [nA] amplitude
```

### 4. Control simulation time course

```
finitialize(-65) // initialize v, state variables, time

// a minimalistic approach for fixed dt simulations

dt = 0.025 // [ms] integration time step
tstop = 5   // [ms]

proc simulate() {
  // show start time & initial somatic v
  print t, v(0.5) // soma is default section

  while (t < tstop) {
    fadvance() // advance solution by dt
    // function calls to save or plot results, e.g.
    print t, v(0.5)
    // statements to change model parameters
  }
}
```

## Distributed mechanisms

Examples:

- voltage-gated ion channels
- ion accumulation & buffering

```
soma insert hh
```

## Point processes

Examples:

- synapses
- electrodes, clamps, spike detectors

Object syntax

```
objref stim
// attach to middle of soma
soma stim = new IClamp(0.5)

stim.del = 1    // [ms] delay
stim.dur = 0.1  // [ms] duration
stim.amp = 60   // [nA] amplitude
```

# NEURON: the HOC programming language

Bill Lytton

SUNY - Downstate  
Brooklyn, NY

NEURON: the HOC programming language – p.1/21

## TOC

- 2. HOC is the interactive language for NEURON
- 3. Numbers
- 4. Functions & operators: pluses and minuses
- 5. NB: x=5 vs x==5
- 6. Assignments
- 7. Block of code
- 8. Conditionals and controls
- 9. Procedures and functions (proc and func)
- 10. Number arguments to procedures:
- 11. Strings
- 12. Objects
- 13. Simulation commands
- 14. Sim - stim
- 15. Sim - running
- 16. Vectors
- 17. What have we recorded?
- 18. Can analyze signals using vectors
- 19. Quick & dirty graphics
- 20. Graphing a vector
- 21. Find spikes
- 22. Check results graphically
- 23. Now can calculate means etc.
- 24. Other useful vector functions
- 25. Putting up buttons
- 26. Reading and writing files

NEURON: the HOC programming language – p.27/21

## Talk to the simulator

- Similar to **C** or **Perl** but *DON'T* use semicolons
- **HOC**=Higher Order Calculator (Kernighan)
- **oc** is an object-oriented augmentation

NEURON: the HOC programming language – p.2/21

## Numbers

- Integers are handled internally with full precision: 5 same as 5.0
- Can declare an array of numbers: double x[10]
- but vectors are usually better
- Scientific notation uses 'e' or 'E'
- `oc>5e3`  
5000  
`oc>5E3`  
5000

NEURON: the HOC programming language – p.3/21

## Functions & operators: pluses and minuse

- Functions: sin, cos, tan, sqrt, log, log10, exp
- Arithmetic operators: + - / %  
oc>5+3 // put comment after double slash
- 8
- Logical operators: && || !
- Comparison operators: == != < >  
oc>5==5
- 1
- NB: x=5 vs x==5

NEURON: the HOC programming language – p.4/21

## NB: x=5 vs x==5

- oc>x = 5 + 7 /\* another way to comment \*/
- oc>x==12
- 1
- oc>x==(5+8)
- 0
- oc>x
- 12

NEURON: the HOC programming language – p.5/21

## Assignments

- `x = x+1`
- `x += 1`
- `x *= 2`
- NO: `x++` (**C** but not in **HOC**)

NEURON: the HOC programming language – p.6/21

## Block of code

- A section of code that gets executed together
- Can be used in a conditional or a procedure
- Statements surrounded by curly brackets – no separator
- Confusing: `{ x = 7 print x x = 12 print x }`  
7  
12
- Better on individual lines:  
`{ x = 7  
print x  
x = 12  
print x }`

NEURON: the HOC programming language – p.7/21

## Conditionals and controls

- Decides whether or how often to execute a block
- `if (5==5) { print "yes" } else { print "no" }`
- did I mention?: 'if (x=5)' – you mean 'if (x==5)'
- `while (x<=7) { print x x+=1 }`
- `for x=1,7 print x`
- `for (x=1;x<=7;x+=2) print x`

NEURON: the HOC programming language – p.8/27

## proc and func

- `proc hello () { print "hello" }`
- `oc>hello()`  
hello
- functions can only return a number
- `func hello () { print "hello" return 1 }`
- `oc>hello()`  
hello  
1

NEURON: the HOC programming language – p.9/27

## Number arguments to procedures:

```
● proc add () { print $1 + $2 }  
● oc>add(5,3)  
8  
● func add () { return $1 + $2 }  
● print 7*add(5,3)  
56
```

NEURON: the HOC programming language – p.10/21

## Strings

```
● Unlike numbers, string variables must be explicitly  
declared  
● oc>strdef str  
oc>str=5  
nrniv: parse error  
str=5  
oc>str= "hello"  
oc>print str  
hello
```

NEURON: the HOC programming language – p.11/21



## Objects

- objref or objectvar declares an object pointer:  
objref g,vec[5],list
- the command *new* creates a new instance of an object
- Graphs, vectors, lists, files are all handled as objects  
g = new Graph()  
for ii=0,4 vec[ii] = new Vector()  
list= new List()
- “dot” notation accesses object components or procedures  
g.erase() // only makes sense if g is a graph  
vec.x[3] // will access a location in vector vec

NEURON: the HOC programming language – p.12/21

## Simulation commands

- GUI buttons are connected to hoc level commands
- Can create and run simulations from the command line
- oc> create soma
- oc> access soma
- oc> insert hh
- oc> ismembrane("hh")  
1

NEURON: the HOC programming language – p.13/21

## Sim - stim

- oc> objref stim
- oc> stim = new IClamp(0.5) // current clamp obj
- oc> stim.amp=20 // need big stim (big L, diam)
- oc> stim.dur=1e10 // duration

NEURON: the HOC programming language – p.14/21

## Sim - running

- oc> tstop = 2 // stop at the peak of the spike
- oc> run()
- oc> print v, v(0.5), soma.v(0.5) // all equivalent
- 38.764279

NEURON: the HOC programming language – p.15/21

## Vectors

- Can record to vectors and then analyze the contents
- objref vec  
oc> vec=new Vector()  
oc> vec.record(&soma.v(0.5))  
oc> tstop = 100  
oc> run()  
resize\_chunk 2046  
resize\_chunk 4094  
resize\_chunk 8190  
resize\_chunk 16382

NEURON: the HOC programming language – p.16/21

## What have we recorded?

- print vec.size(),dt,vec.size\*dt,tstop
- print vec.min,vec.max  
-74.774437 40.444033
- print  
vec.min\_ind,vec.max\_ind,vec.min\_ind\*dt,vec.max\_ind\*dt  
470 190 4.7 1.9
- print vec.x[470],vec.x[190]  
-74.774437 40.444033

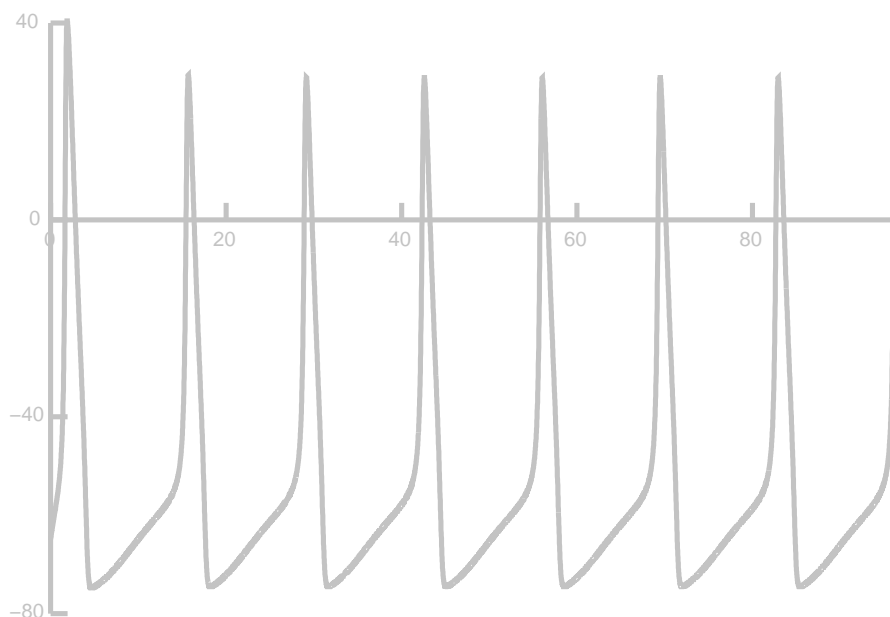
NEURON: the HOC programming language – p.17/21

## Can analyze signals using vectors

- Find the steepest action potential
- `vec[1].deriv(vec,dt)`
- `print vec[1].max_ind,vec[1].max_ind*dt`  
168 1.68

NEURON: the HOC programming language – p.18/21

## Quick & dirty graphics



NEURON: the HOC programming language – p.19/21

## Graphing a vector

- Can put up a graph from the main menu or by hand  
`g = new Graph()`
- Draw the vector on the graph  
`vec.line(g,dt)`
- Need a time vector if using var dt
- Erase and redraw  
`g.erase`

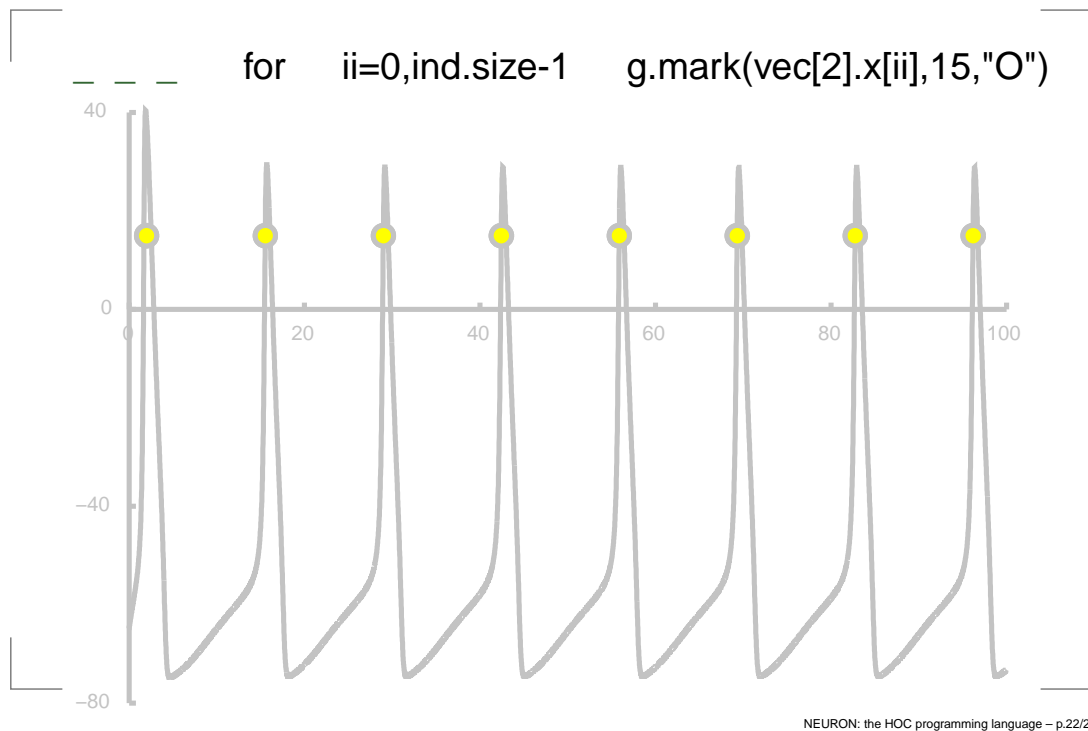
NEURON: the HOC programming language – p.20/21

## Find spikes

- `vec[1].indvwhere(vec,">",15)` // indices above a threshold
- `vec[1].mul(dt)` // times
- `spktime=0`
- for `ii=0,vec[1].size-1` if `(vec[1].x[ii]<spktime+2)`  
`vec[1].x[ii]=-1` else `spktime=vec[1].x[ii]`
- `vec[2].where(vec[1],">",0)`

NEURON: the HOC programming language – p.21/21

## Check results graphically



## Now can calculate means etc.

- calculate differences: `vec[3].sub(othervec)`
- take inverses: `vec[3].resize()`, `vec[3].fill(1)`, `vec[3].div(othervec)`
- print `vec[3].mean()`, `vec[3].stdev()`

## Other useful vector functions

- `vec.setrand(rdm)` // where `rdm=new Random()`
- `vec.fft()` // fast fourier transform
- `vec.sort()`
- `vec.histogram()`
- `vec.apply("user_func")`

NEURON: the HOC programming language – p.24/27

## Putting up buttons



NEURON: the HOC programming language – p.25/27

## Reading and writing files

- `file=new File()`
- `file.wopen("tmp")`
- `vec.printf(file) // or vec.vwrite(file) for binary`
- `file.close()`

NEURON: the HOC programming language – p.26/27



## The Vector class

Efficient methods for collecting  
and manipulating arrays of numbers

## Vector functions

### Initialization & storage

buffer_size	size	resize	
fill	indgen	setrand	sin
append	c	cl	copy
rebin	resample	reverse	rotate
insrt	remove		

### Database

sort	sortindex		
contains	label		
index	max_ind	min_ind	ind
where	indwhere	indvwhere	

### I/O

fread	fwrite	
vread	vwrite	
scanf	printf	scantil

### Graph

plot	line	ploterr	mark
------	------	---------	------

## Vector functions *continued*

### Algebra

add	div	sub	mul	dot
log	log10	tanh	pow	sqrt
abs	max	min		
sum	sumsq	mag		
integral	deriv			

### Signal processing

fft	spctrm	convlv	correl	filter
medfltr	addrand	apply		
psth	spikebin	trigavg		

### Statistics

mean	median		
stderr	stdev	var	meansqerr
hist	histogram	smhist	sumgauss

### Simulation

record	play	play_remove
--------	------	-------------

## Vector record

### Syntax

```
vdest.record(&var)
vdest.record(&var, Dt)
vdest.record(&var, tvec)
```

### Description

Saves the stream of values of `var` during a simulation into `vdest`, sampling at solution times, regular intervals `i*Dt`, or times specified by `tvec`.

## Vector play

### Syntax

```
vsrc.play(&var, Dt [, continuous])
vsrc.play(&var, tvec [, continuous])
vsrc.play(&var, tvec [, continuous])
```

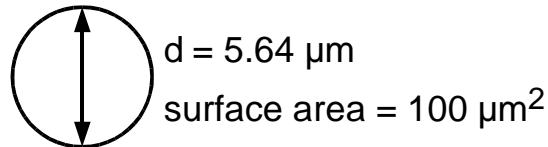
### Description

The `vsrc` values are assigned to `var` during a simulation. If optional third argument is nonzero, linear interpolation defines values of `var` when `t` falls between `i*Dt` or `tvec` points.

## Model Control: arbitrary forcing functions

Physical system: patch of squid axon

Model: compact compartment with HH currents



Project goals:

- Set up and test a voltage clamp (SEClamp)

- Generate arbitrary waveform, e.g. voltage ramp (Grapher)

- Capture and transfer Vector data ("clipboard")

- Use recorded waveform as SEClamp's "command" (Vector Play tool)

## Model Control: simulation families

Physical system: pyramidal neuron

Model: ball and stick model

conductance-change synapse

Project goals:

- How does peak synaptic depolarization vary with synaptic location?

- Program control of PointProcess location

- Exploratory data collection and analysis

- Use Vector class to automate data collection and analysis across multiple runs



# The Linear Circuit Builder

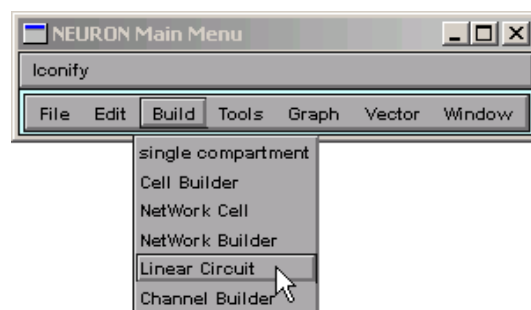
For building models that have linear circuit elements  
and may also involve neurons

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

Potential applications include

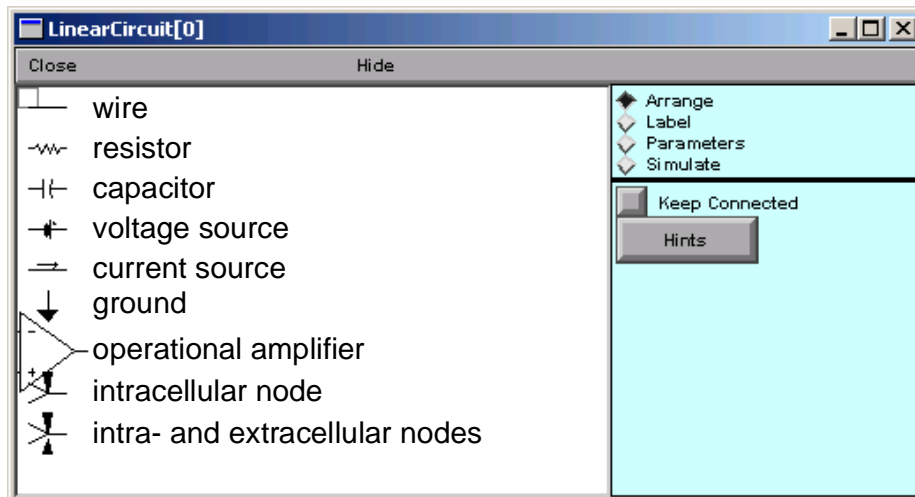
- effects and compensation of electrode R & C
- two-electrode voltage clamp
- ohmic and nonlinear gap junctions

## 1. Bring up a Linear Circuit Builder



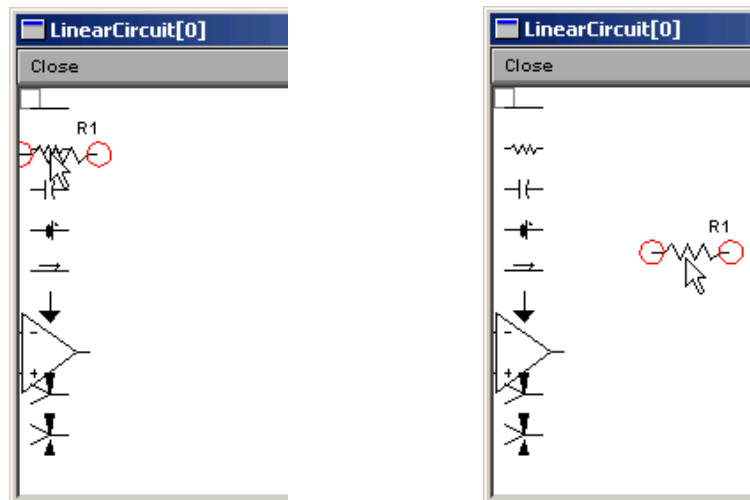
NEURON Main Menu / Build / Linear Circuit

## The Linear Circuit Builder



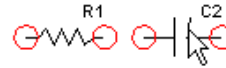
### Arrange: spawn components

Click on palette and drag onto canvas

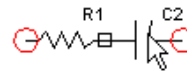


## Arrange: connect components

Click and drag to  
overlap red circles



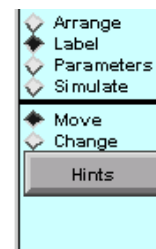
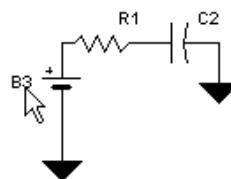
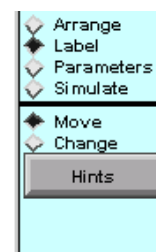
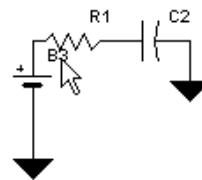
Black square is  
"solder joint"



Pull apart to break connection

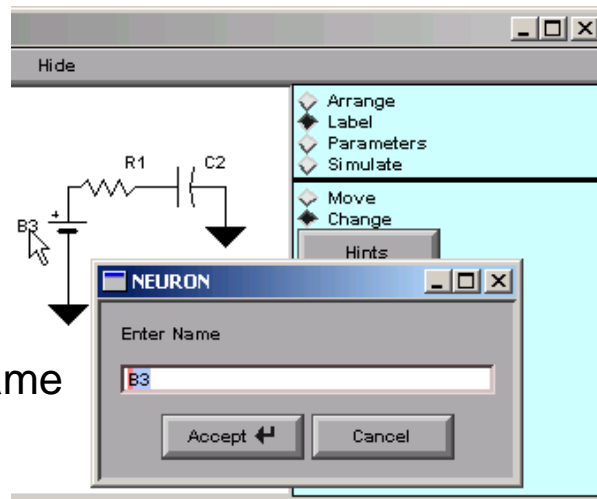
## Label: move labels

Click and drag  
to new location



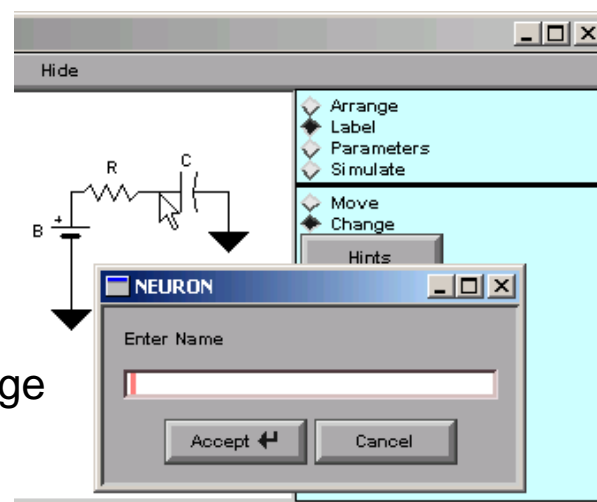
## Label: change labels 1

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



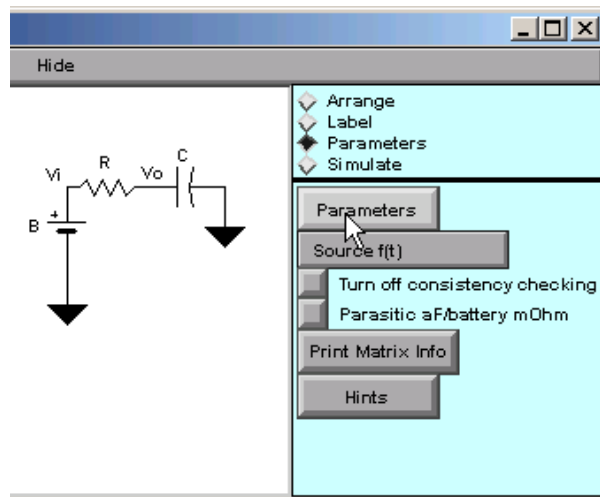
## Label: change labels 2

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

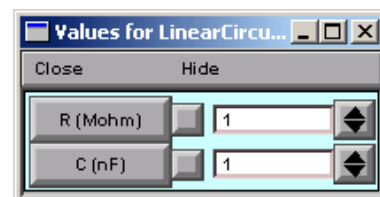




## Parameters: non-source elements

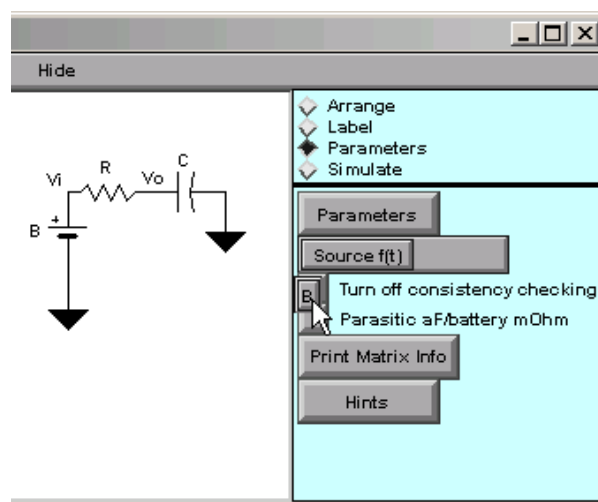


Click on  
"Parameters"

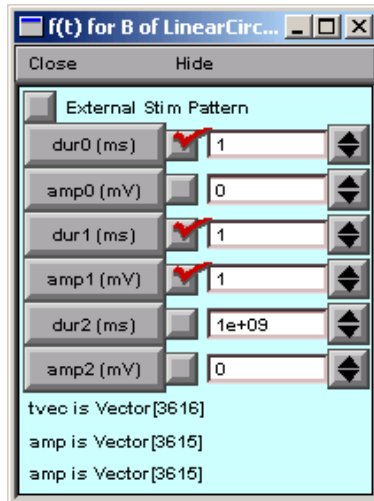


## Parameters: signal sources

Source f(t) / B

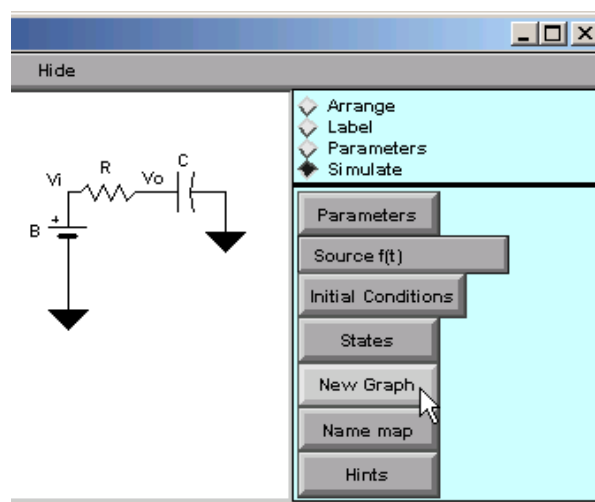


## Parameters: signal sources *continued*



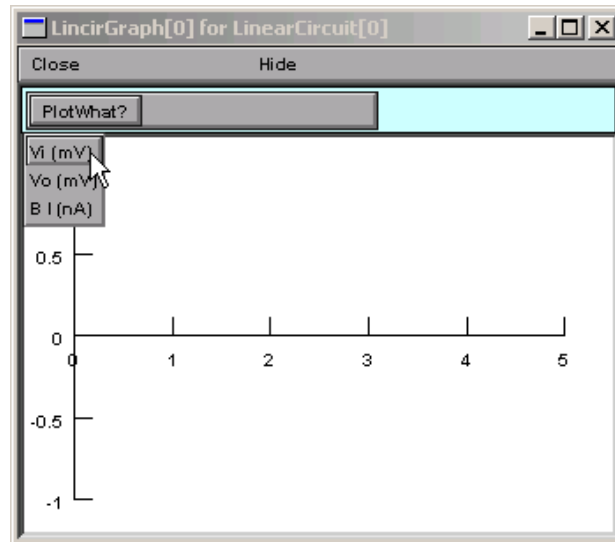
Configured

## Simulate: creating a graph



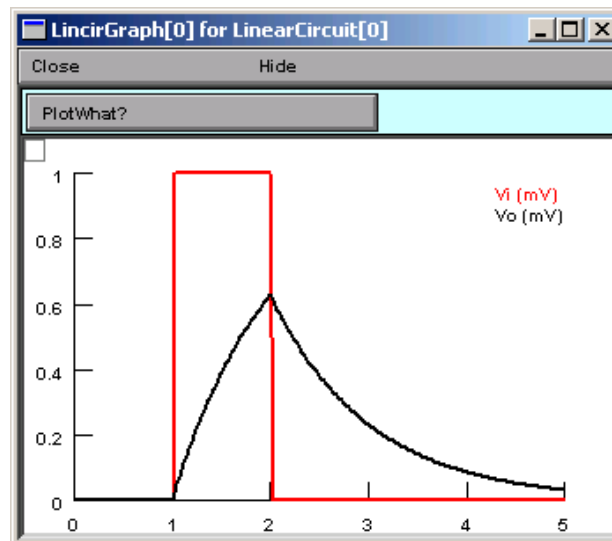
New Graph

## Simulate: specifying what to plot



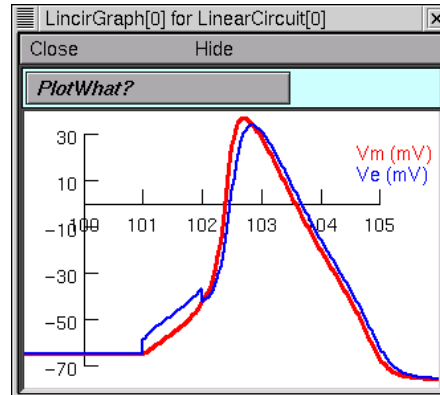
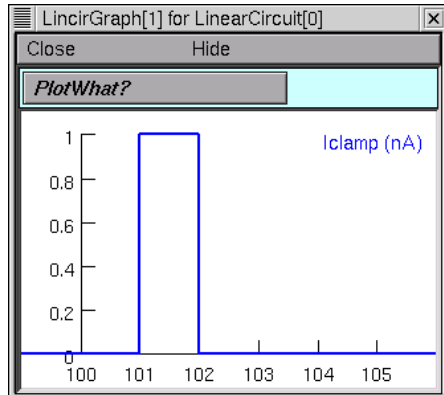
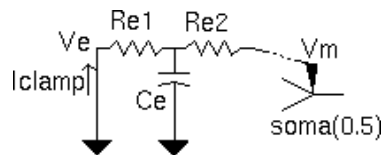
PlotWhat? / *variable\_label*

## Simulate: simulation results

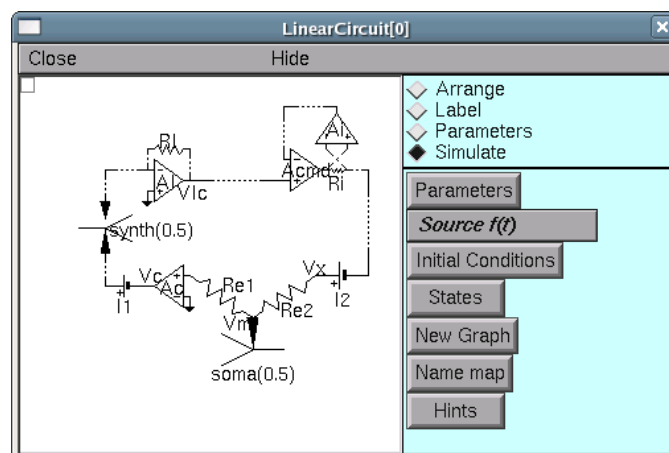


After minor cosmetic changes

## Patch clamp with electrode R and C



## NEURON demo: dynamic clamp



# NMODL

NEURON Model Description Language

## Add new membrane mechanisms to NEURON

### Density mechanisms

- Distributed Channels
- Ion accumulation

### Point Processes

- Electrodes
- Synapses

## Described by

- Differential equations
- Kinetic schemes
- Algebraic equations

## Benefits

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

# NMODL general block structure

## What the model looks like from outside

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

## What names are manipulated by this model

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

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

STATE { n ... }

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

## Initial default values for states

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

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

(and specify how states are to be integrated)

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

## State equations

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

## Functions and procedures

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

## Density mechanism

## Point Process

### NMODL

```

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

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

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

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

```

```

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

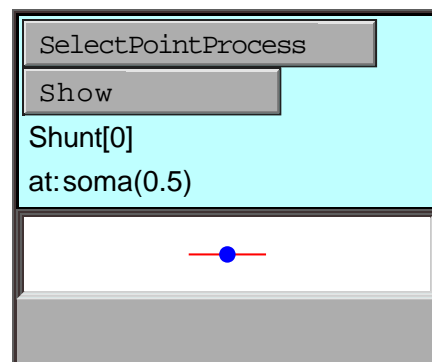
PARAMETER {
    r = 1 (gigaohm) <1e-9,1e9>
    e = 0 (millivolt)
}

ASSIGNED {
    i (nanoamp)
    v (millivolt)
}

BREAKPOINT {
    i = (.001)*(v - e)/r
}

```

### GUI



### Interpreter

```

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

```

```

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

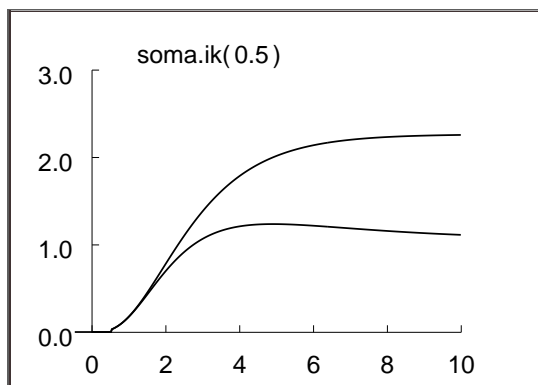
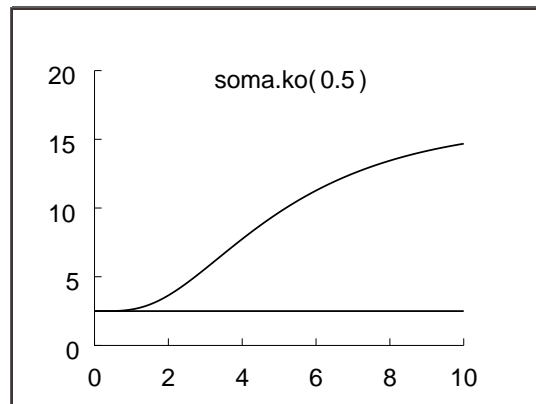
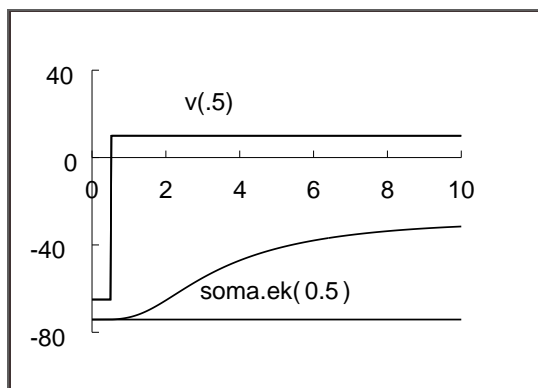
```

## Ion Channel

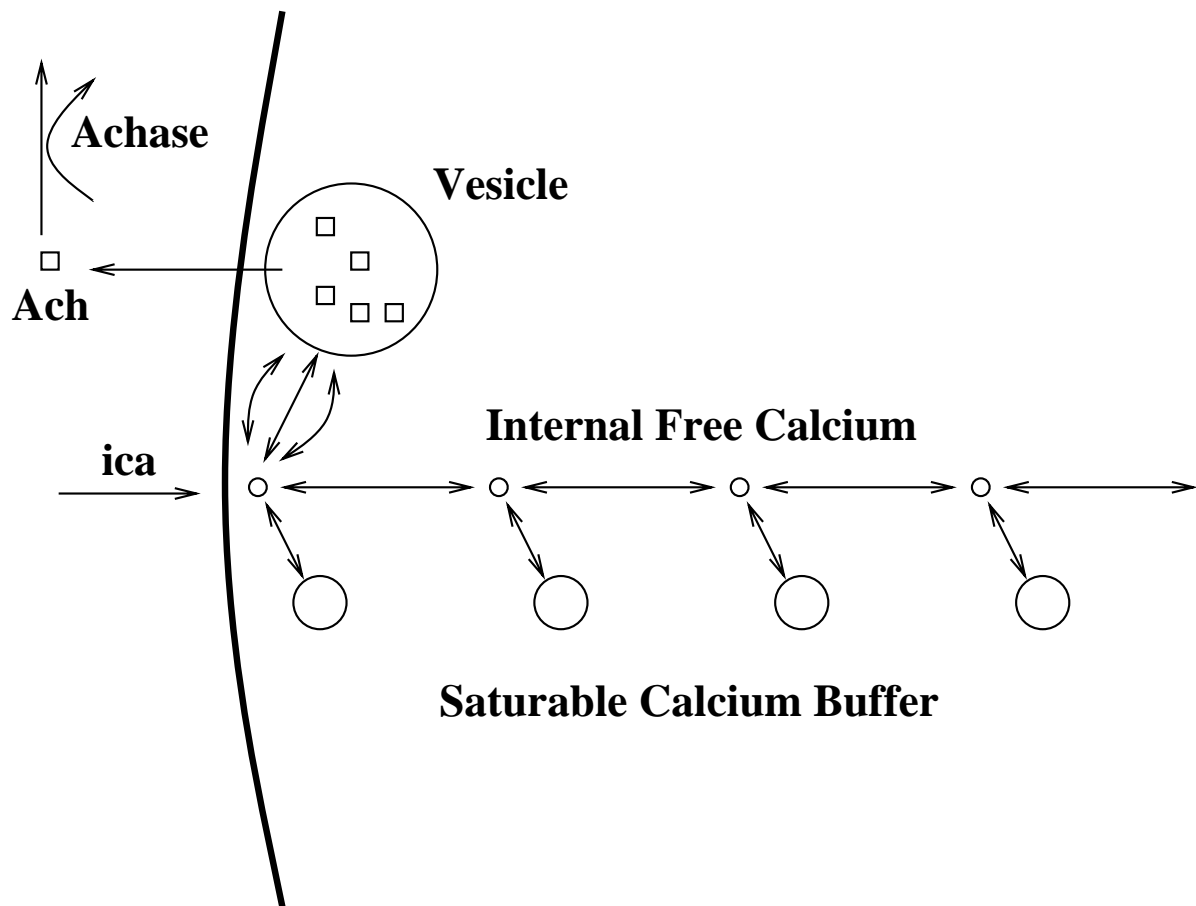
```
NEURON {
  USEION k READ ek WRITE ik
}
BREAKPOINT {
  SOLVE states METHOD cnexp
  ik = gbar*n*n*n*n*(v - ek)
}
DERIVATIVE states {
  rate(v*1(/mV))
  n' = (inf - n)/tau
}
```

## Ion Accumulation

```
NEURON {
  USEION k READ ik WRITE ko
}
BREAKPOINT {
  SOLVE state METHOD cnexp
}
DERIVATIVE state {
  ko' = ik/fhspace/F*(1e8)
      + k*(kbath - ko)
}
```







```

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

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

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

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

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

```

# UNITS Checking

```

NEURON { POINT_PROCESS Shunt ... }

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

ASSIGNED {
    i (nanoamp)
    v (millivolt)
}

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

```

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

```
modlunit shunt
```

is:

```

Checking units of shunt.mod
The previous primary expression with units: 1-12 coul/sec
is missing a conversion factor and should read:
    (0.001)*()
at line 14 in file shunt.mod
    i = (v - e)/r<>

```

To fix the problem replace the line with:

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

---

## What conversion factor will make the following consistent?

$$\begin{array}{lcl} \text{nai}' & = & \text{ina} \quad / \quad \text{FARADAY} \quad * \quad (\text{c/radius}) \\ (\text{uM/ms}) & & (\text{mA/cm}^2) \quad / \quad (\text{coulomb/mole}) \quad / \quad (\text{um}) \end{array}$$

## UNIX

In the directory containing the desired mod files:

```
nrnivmodl  
nrngui
```

Select NEURONMainMenu/Build/singlecompartment.

## MSWIN

Launch mknrndll from the icon in the NEURON program group.

Navigate to the directory containing the desired mod files.  
Select "Make nrnmech.dll".

Launch nrngui from the icon in the NEURON program group.

Select NEURONMainMenu/File/RecentDir to change the working dir and load  
nrnmech.dll.  
Select NEURONMainMenu/Build/singlecompartment.

---

### single.hoc

```
load_file("stdgui.hoc")  
create soma  
access soma  
// area 100 um2 means mA/cm2 identical to nA  
{diam=10 L=10/PI}
```

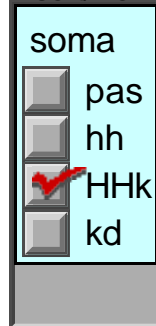
: Hodgkin - Huxley k channel

```

NEURON {
  SUFFIX HHk
  USEION k READ ek WRITE ik
  RANGE gkbar, ik, g
  GLOBAL inf, tau
}
UNITS {
  (mA) = (milliamp)
  (mV) = (millivolt)
}
PARAMETER {
  gkbar= 0.036 (mho/cm2) <0,1e9>
}
STATE {
  n
}
ASSIGNED {
  v (mV)
  ek (mV)
  celsius (degC)
  ik (mA/cm2)
  inf
  tau (ms)
  g (mho/cm2)
}
INITIAL {
  rate(v)
  n = inf
}
BREAKPOINT {
  SOLVE states METHOD cnexp
  g = gkbar*n*n*n*n
  ik = g*(v - ek)
}
DERIVATIVE states {
  rate(v)
  n' =(inf - n)/tau
}
FUNCTION alp(v(mV))/(ms) { LOCAL q10
  v = -v - 65
  q10 = 3^((celsius - 6.3)/10 (degC))
  alp = q10 * 0.01(/ms-mV)*expM1(v + 10, 10(mV))
}
FUNCTION bet(v(mV))/(ms) { LOCAL q10
  v = -v - 65
  q10 = 3^((celsius - 6.3)/10 (degC))
  bet = q10 * 0.125(/ms)*exp(v/80(mV))
}
FUNCTION expM1(x (mV),y (mV)) (mV) {
  if (fabs(x/y) < 1e-6) {
    expM1 = y*(1 - x/y/2)
  }else{
    expM1 = x/(exp(x/y) - 1)
  }
}
PROCEDURE rate(v (mV)) {LOCAL a, b
  TABLE inf, tau DEPEND celsius FROM -100 TO 100 WITH 200
  a = alp(v)  b=bet(v)
  tau = 1/(a + b)
  inf = a/(a + b)
}

```

### Insert/Remove Mechanisms



### soma(0 - 1) (Parameters)

soma(0 - 1) (Parameters)

nseg = 1

L (um) 3.1831

Ra (ohm-cm) 35.4

diam (um) 10

cm (uF/cm2) 1

gkbar\_HHk(mho/cm2) 0.036

ek (mV) -77

### soma(0.5) (States)

soma(0.5) (States)

v -65

n\_HHk 0

### HHk (Globals)

inf\_HHk 0.9725

tau\_HHk(ms) 0.92617

usetable\_HHk 1

### soma(0.5) (Assigned)

soma(0.5) (Assigned)

v -65

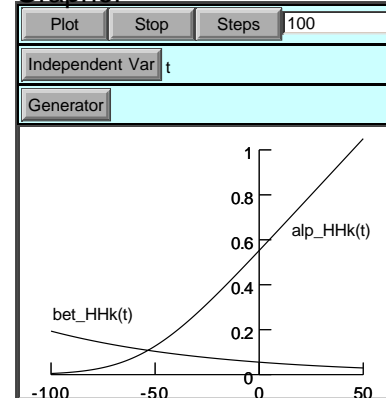
i\_cap 0

ik\_HHk(mA/cm2) 0

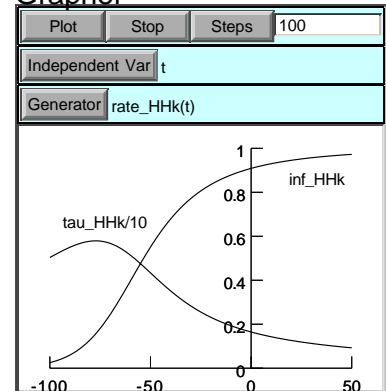
g\_HHk(mho/cm2) 0

ik (mA/cm2) 0

### Grapher



### Grapher







# Computational Modeling and Neuroscience

Does computational modeling have a role  
in neuroscience research?

1

## Best Practices

Know the literature  
Collaborate with experimentalists  
Use Occam's razor  
Adhere to scientific method

2

## Scientific Method

Observation

Hypothesis

Prediction

Verification

Evaluation

3

## Reproducibility

### **The ideal:**

"Reproducibility is the cornerstone of scientific method."

"Experiments should be fully described so that anyone can reproduce them."

### **The harsh reality: Velilind's Laws of Experimentation**

"If reproducibility may be a problem, conduct the test only once.

If a straight line is required, obtain only two data points."

7



<http://modeldb.yale.edu>

**ModelDB**

ModelDB provides an accessible location for storing and efficiently retrieving computational neuroscience models. ModelDB is tightly coupled with [NeuronDB](#). Models can be coded in any language for any environment. Model code can be viewed before downloading and browsers can be set to auto-launch the models. About [model sharing in general](#) and [ModelDB in particular](#).

[Submit a new model entry](#) [Model entry tutorial](#) [Help](#)

**Find models by**

- [model name](#)
- [first author](#)
- [each author](#)
- [Region\(circuits\)](#)

**Find models for**

- [Cell type](#)
- [Current](#)
- [Receptor](#)
- [Transmitters](#)
- [Topic](#)
- [Simulators](#)
- [Methods](#)

**Find models of**

- [Networks](#)
- [Neurons](#)
- [Synapses](#)
- [Electrical synapses \(gap junctions\)](#)
- [Chemical synapses](#)
- [Ion channels](#)
- [Neuromuscular junctions](#)
- [Axons](#)

Search for models by author name or accession number

Search for SenseLab models using Google   [Hints](#)

[Search for publications in ModelDB](#) or [in PubMed](#)

[Register](#) for an account  
[Login](#) to access your models  
 Related [Resources](#)  
 Some models versions are available in a [mercurial repository](#)

8

## Accommodates models from a wide range of simulation environments

488 entries as of 6/16/2009

BioPAX	1	KInNeSS	2	PyNN	1
Brian	2	MATLAB	86	Python	5
C / C++	42	MCell	1	Q/Quick/Turbo Basic	2
CSIM	4	MOOSE / PyMOOSE	1	QuB	1
CalC	7	MVASpike	1	SABER	1
Catcomb	1	MadSim	1	SNNAP	21
CellExcite	1	NCS	1	SciLab	2
CellML	1	NEST	2	Simulink	6
Chemesis	2	NEURONPM	2	Sspice	1
Emergent/PDP++	3	Network	1	Topographica	1
FORTTRAN	5	Neuron	244	Virtual Cell	2
GNU/Next/Openstep	1	Octave	1	XML	3
Genesis	19	PCSIM	1	XPP	49
IGOR Pro	3	PSpice	2	neuroConstruct	1

9

## Search results

### Models by moore

1. [Nerve terminal currents at lizard neuromuscular junction, Lindgren and Moore 1989](#)  
Lindgren CA, Moore JW (1989) Identification of ionic currents at presynaptic nerve endings of the lizard. J Physiol 414:201—22 [[PubMed](#)]
2. [Presynaptic calcium dynamics at neuromuscular junction, Stockbridge and Moore 1984](#)  
Stockbridge N, Moore JW (1984) Dynamics of intracellular calcium and its possible relationship to phasic transmitter release and facilitation at the frog neuromuscular junction. J Neurosci 4:803—11 [[PubMed](#)]
3. [Site of impulse initiation in a neuron by Moore et al 1983](#)  
Moore JW, Stockbridge N, Westerfield M (1983) On the site of impulse initiation in a neurone. J Physiol 336:301—11 [[PubMed](#)]
4. [Current flow during PAP in squid axon at diameter change: Joyner et al 1980](#)  
Joyner RW, Westerfield M, Moore JW (1980) Effects of cellular geometry on current flow during a propagated action potential. Biophys J 31:183—94 [[PubMed](#)]
5. [Conduction in uniform myelinated axons: Moore et al 1978](#)  
Moore JW, Joyner RW, Brill MH, Waxman SD, Najar-Joa M (1978) Simulations of conduction in uniform myelinated fibers. Relative sensitivity to changes in nodal and internodal parameters. Biophys J 21:147—60 [[PubMed](#)]
6. [Temperature-Sensitive conduction at axon branch points by Westerfield et al 1978](#)  
Westerfield M, Joyner RW, Moore JW (1978) Temperature-sensitive conduction failure at axon branch points. J Neurophysiol 41:1—8 [[PubMed](#)]
7. [Myelinated axon conduction velocity by Brill et al 1977](#)  
Brill MH, Waxman SG, Moore JW, Joyner RW (1977) Conduction velocity and spike configuration in myelinated fibres: computed dependence on internode distance. J Neurol Neurosurg Psychiatry 40:769—74 [[PubMed](#)]

10

## Site of impulse initiation in a neuron by Moore et al. 1983

### Site of impulse initiation in a neuron by Moore et al 1983

Examines the effect of temperature, the taper of the axon hillock, and HH channel density on antidromic spike invasion into the soma and spike initiation under dendritic stimulation.

Reference: Moore JW, Stockbridge N, Westerfield M (1983) On the site of impulse initiation in a neurone. J Physiol 336:301—11 [[PubMed](#)]

Citations [Citation Browser](#)

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

Model Type:	<a href="#">Neuron</a>
Cell Type(s):	<a href="#">Spinal motor neuron</a>
Channel(s):	<a href="#">I<sub>Na,t</sub></a> ; <a href="#">I<sub>K</sub></a>
Receptor(s):	
Transmitter(s):	
Simulation Environment:	<a href="#">Neuron</a>
Model Concept(s):	<a href="#">Action Potential Initiation</a> ; <a href="#">Simplified Models</a>

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

Model files [Download zip file](#) [Auto-launch](#) [Help downloading and running models](#)

<ul style="list-style-type: none"> <li>moore83</li> <li><b>README</b></li> <li>mosinit.hoc</li> <li>init.hoc</li> <li>startses</li> </ul>	<p>Moore, Stockbridge, and Westerfield. (1983) On the site of impulse initiation in a neurone. J. Physiol. 336: 301-311.</p> <p>This model qualitatively reproduces figures 1-5. Note that orthodromic stimulus amplitude is considerably different from that noted in the paper. IClamp[0].amp was chosen to give qualitative similarity. We attribute minor quantitative differences to the following:</p> <ol style="list-style-type: none"> <li>1) The precise site of axon v vs t curve is not specified. We plot axon.v(0.25).</li> <li>2) The antidromic stimulus was unspecified.</li> </ol> <p>The NEURON implementation of this model was prepared by Michael Hines. Questions about details of this implementation should be addressed to him at michael.hines@yale.edu.</p>
---	---

## How to proceed

Read abstract / paper

Download, extract zip, compile mod files, run mosinit.hoc

Analyze model

    ModelView

    topology, Shape plot

    forall psection()

    Read code . . .

Reusable components?

13



# Optimization

Given:

data from an experiment  
and  
a model with parameters,  
some measured, some estimated

The task:

adjust the parameters  
so that  
the model's activity approximates the data

1

Objective function  $f$

A measure of how well the model meets some objective

Typically the sum of squared errors between  
experimentally recorded membrane potential  
and

the model's output generated by a simulation

Parameters that affect the value of  $f$

Algorithm for adjusting the parameters to reduce error

2

## Multiple-objective optimization

Multiple objective functions  $f_1, f_2 \dots$

Examples:

- Fitting a channel model to a family of current recordings obtained under voltage clamp

- Fitting a model of a cell to membrane potential recorded simultaneously with multiple patch electrodes

Strategy: minimize the weighted sum  $f = \sum w_i f_i$

3

## The Multiple Run Fitter

Multiple objective functions  $f_1, f_2 \dots$

Examples:

- Fitting a channel model to a family of current recordings obtained under voltage clamp

- Fitting a model of a cell to membrane potential recorded simultaneously with multiple patch electrodes

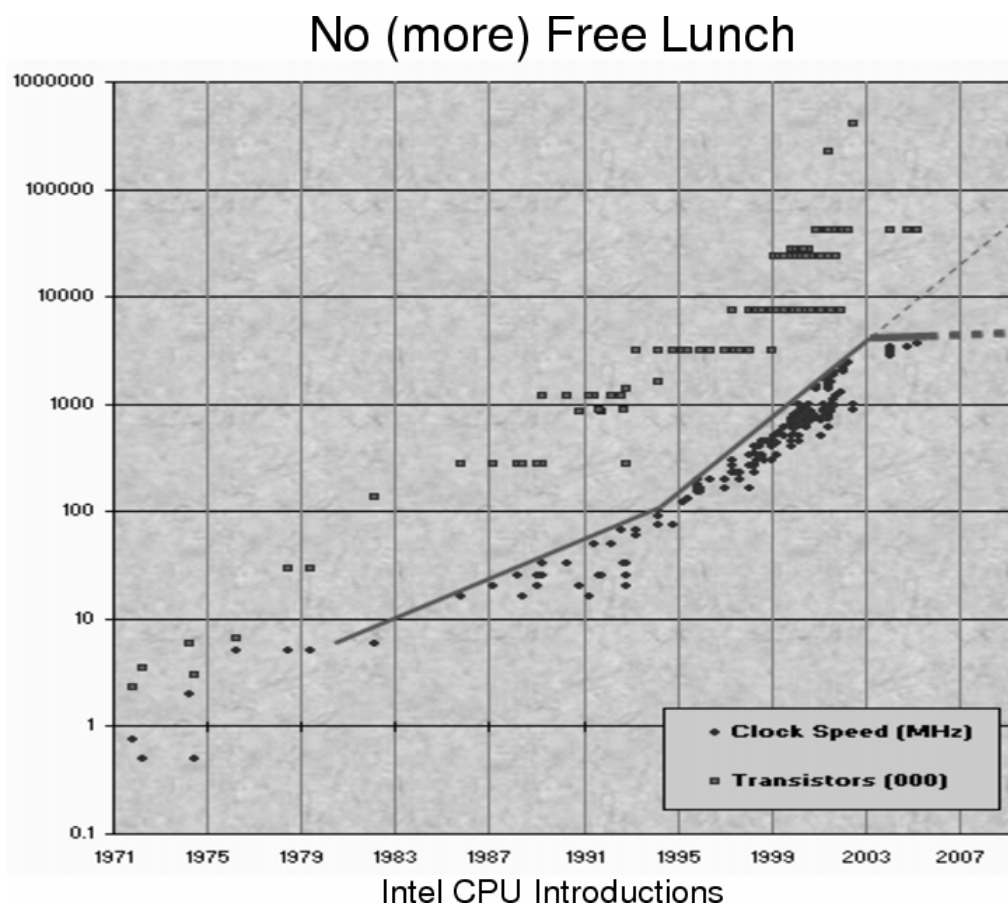
Strategy: minimize the weighted sum  $f = \sum w_i f_i$

Optimization algorithm: PRAXIS, but you can add others

4

# NEURON + Threads

## Simulations on multicore desktops.



## Thread style in NEURON

## Join

```
run() {
  while (t < tstop) {
    multithread_job(step)
    plot()
  }
}
```

```
void* step(NrnThread* nt) {
  ... nt->id ...
}
```

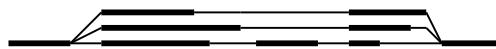


We never use.

## Condition Wait

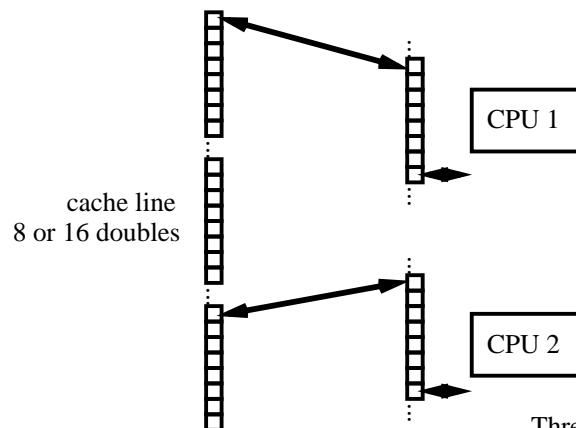
```
multithread_job(run)
```

```
run(NrnThread* nt) {
  while(t < tstop) {
    step(nt)
    barrier()
    if (nt->id == 0) { plot() }
    barrier()
  }
}
```



Reminiscent of MPI

## Ideal cache efficiency

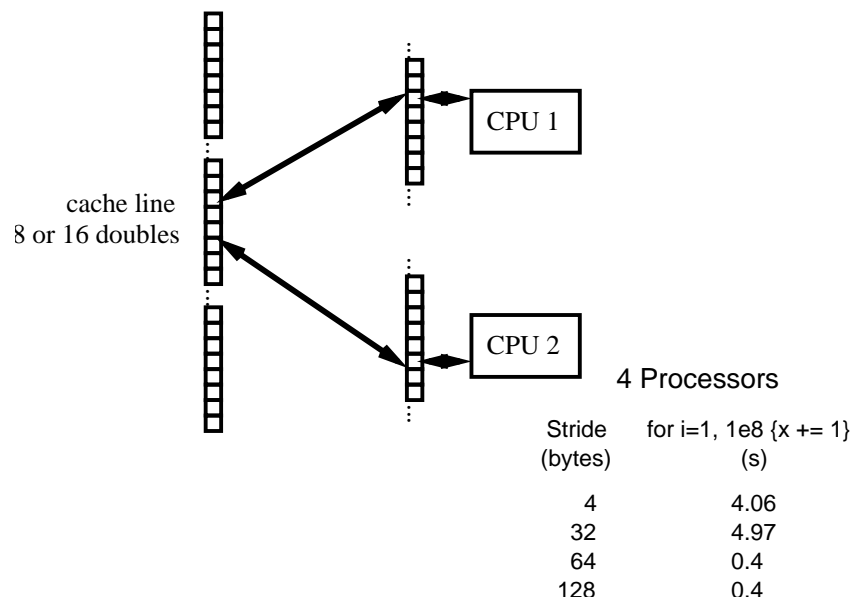
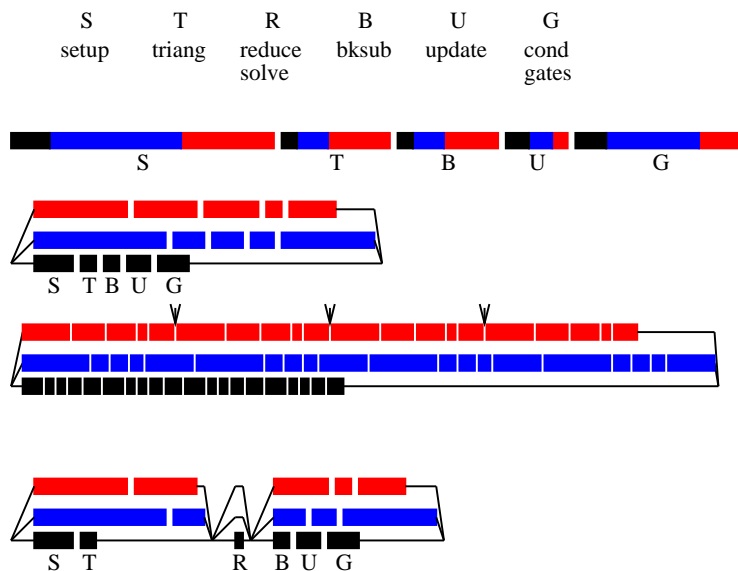


10000 passive compartments  
4 core 3GHz x86\_64

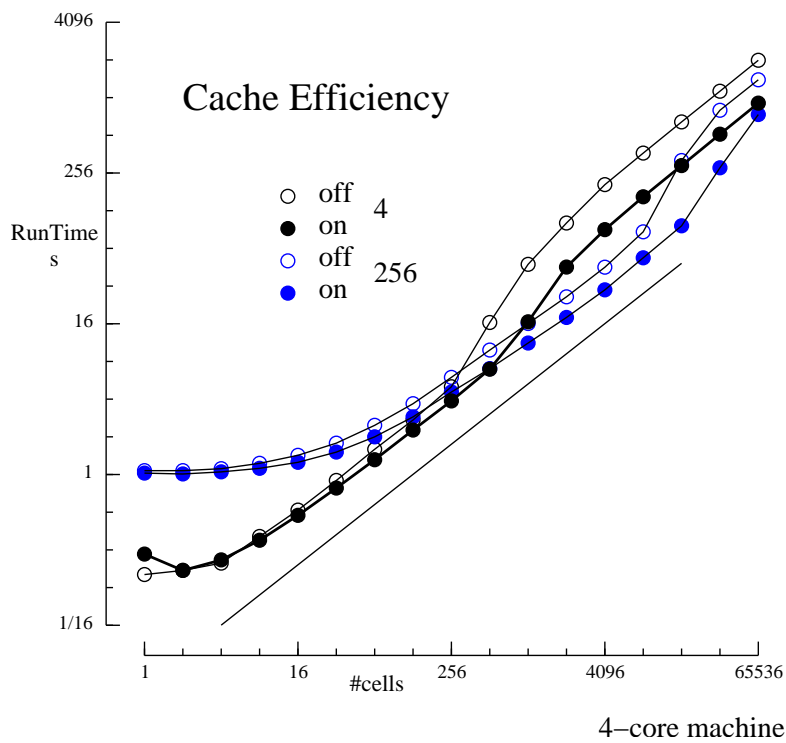
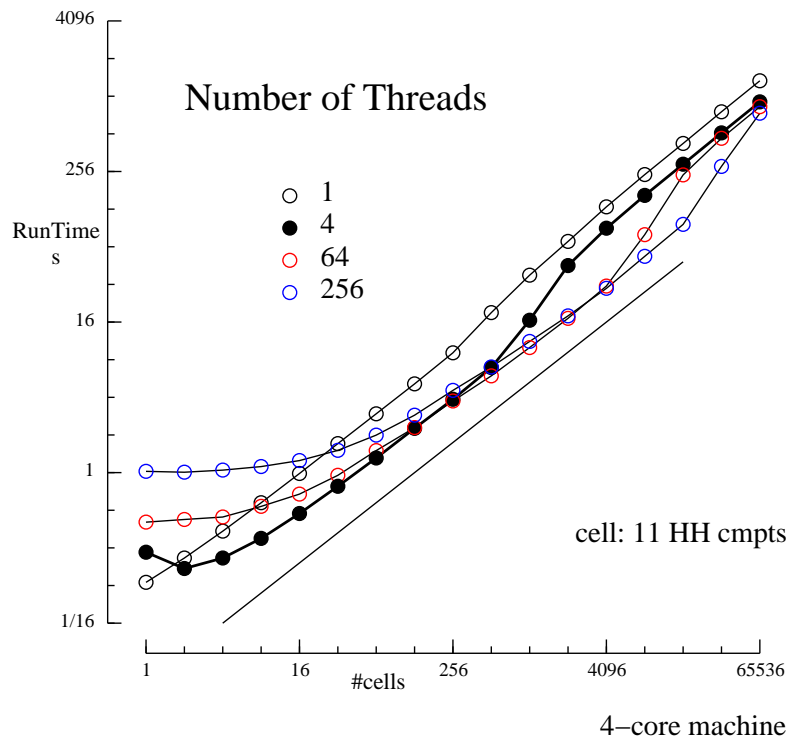
Threads	Runtime (s)	
	Cache Off	Cache On
1	4.92	0.45
2	1.14	0.23
4	0.29	0.12
8	0.23	0.09

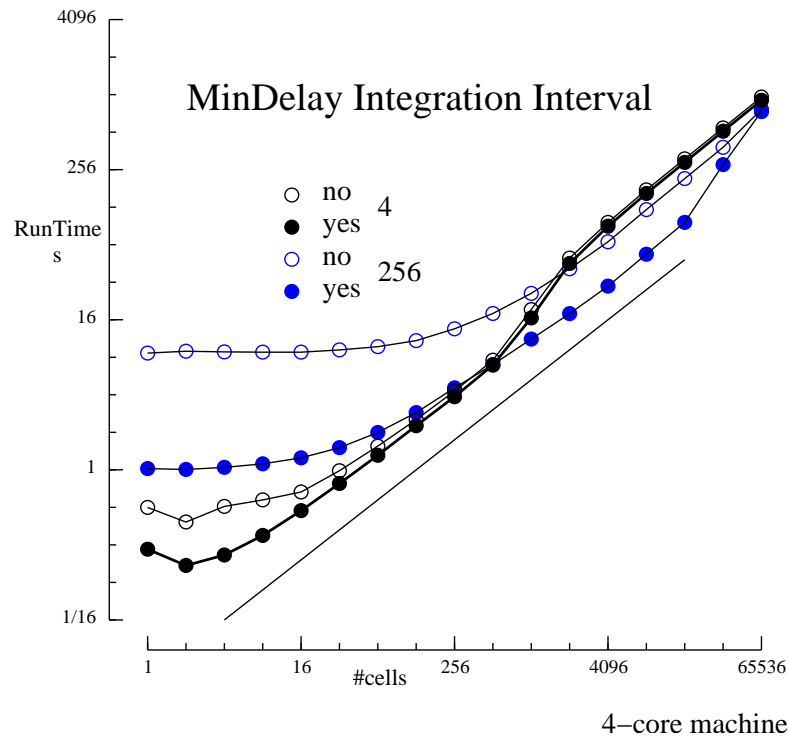


## False cache line sharing

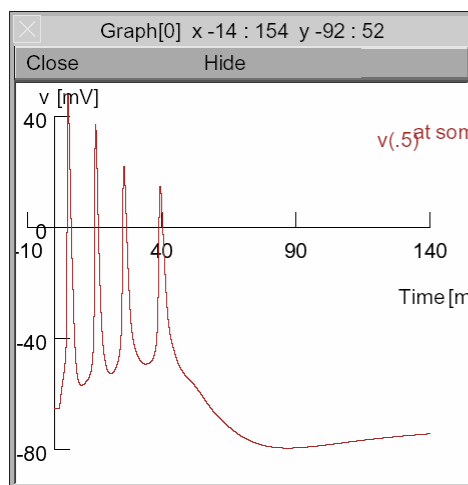
Fixed step:  $t \rightarrow t + dt$ 

Global var dt     $y' = f()$      $dy'/dy$   
 27 ||Vector operations





## Lazarewicz 2002, CA3 Pyramidal Neuron



RunControl

Close Hide

Init (mV) -65

Init & Run

Stop

Continue til (ms) 5

Continue for (ms) 1

SingleStep

t (ms) 140

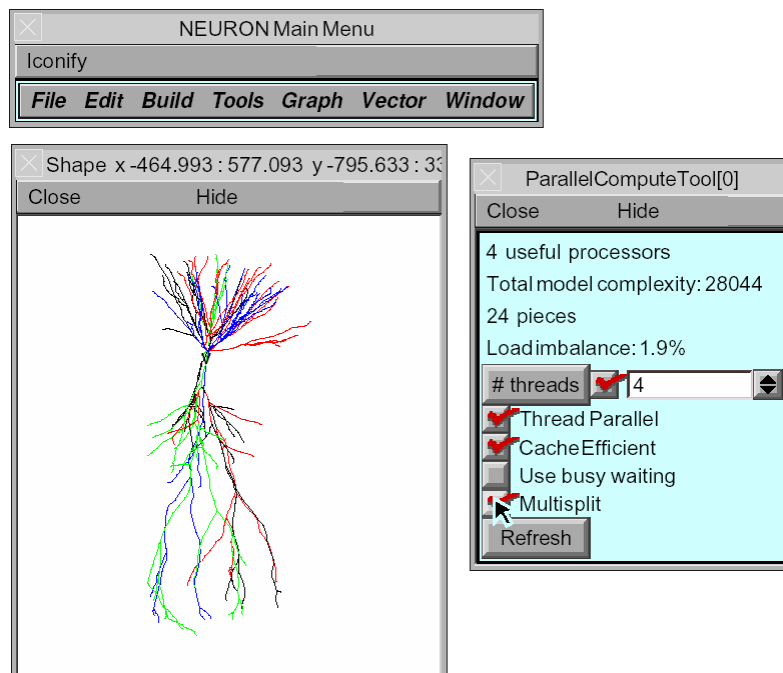
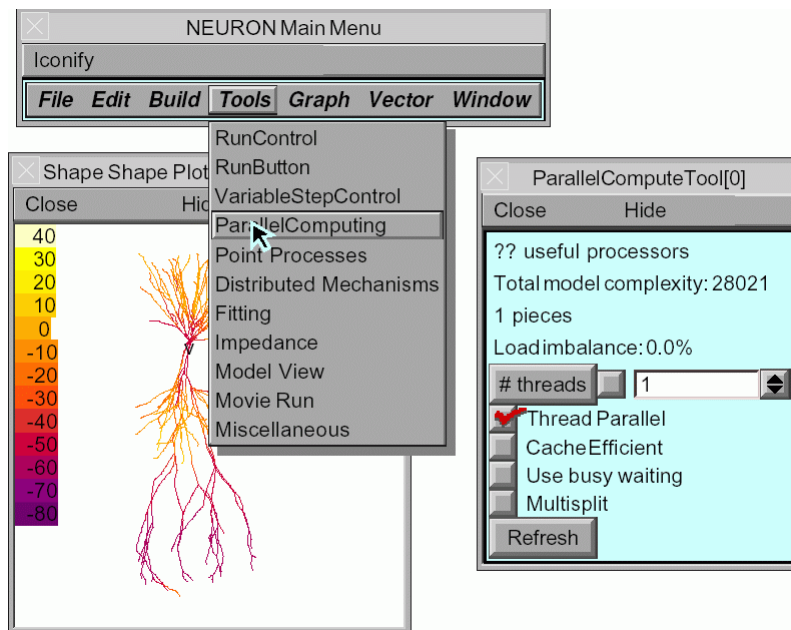
Tstop (ms) 140

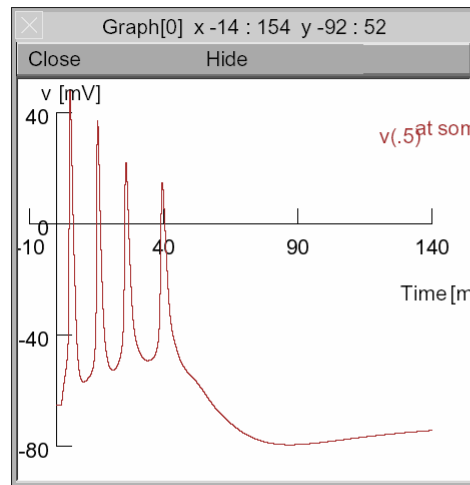
dt (ms) ☒ 2.335

Points plotted/ms 40

Scrn update invl (s) 0.05

Real Time (s) 35.4





instead of 35.4s

### \$ mkthreadsafe

```
NEURON {
    SUFFIX CAIM95
    USEION ca READ cai,cao WRITE ica
    RANGE gbar,ica
    GLOBAL minf,tau
}
```

Translating CAIM95.mod into CAIM95.c

Notice: Assignment to the GLOBAL variable, "minf", is not thread safe

Notice: Assignment to the GLOBAL variable, "tau", is not thread safe

Force THREADSAFE? [y][n]: n

```

DERIVATIVE state {
    rate(v)
    m' = (minf - m)/tau
}

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

```

Force THREADSAFE? [y][n]: n  
**y**

```

NEURON {
    THREADSAFE
    SUFFIX CAIM95
    USEION ca READ cai,cao WRITE ica
    RANGE gbar,ica
    GLOBAL minf,tau
}

```

```

$ mkthreadsafe
NEURON {
    POINT_PROCESS GABAa
    POINTER pre
    ...
}
VERBATIM
return 0;
ENDVERBATIM

```

Translating gabaa.mod into gabaa.c

Notice: Use of POINTER is not thread safe.

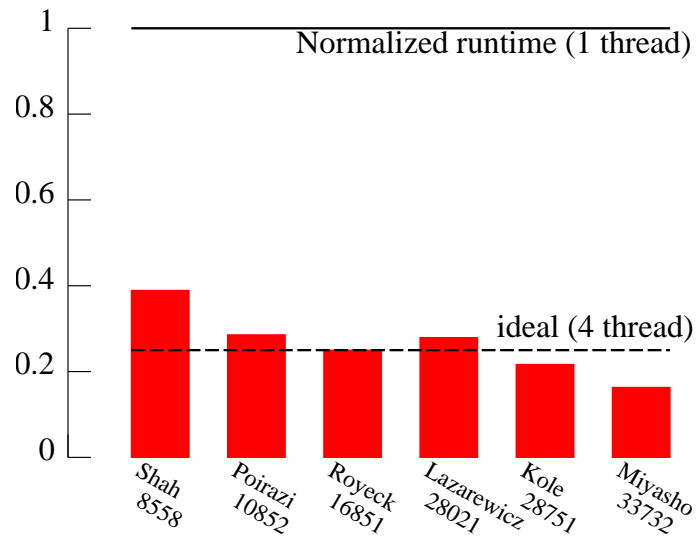
Notice: VERBATIM blocks are not thread safe

Notice: Assignment to the GLOBAL variable, "Rtau", is not thread safe

Notice: Assignment to the GLOBAL variable, "Rinf", is not thread safe

Force THREADSAFE? [y][n]: n

## Relative thread performance 1 vs 4 processors







## 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

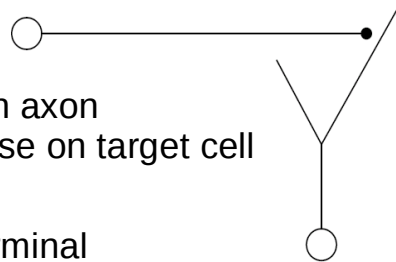
Physical system:

Presynaptic neuron with axon  
that projects to synapse on target cell

Conceptual model:

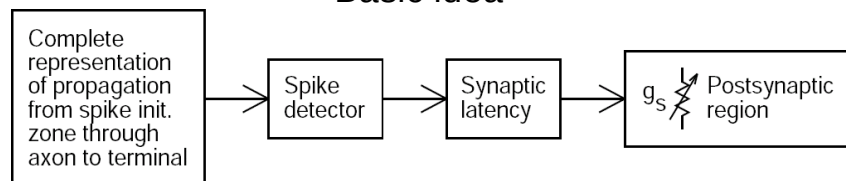
Spike in presynaptic terminal  
triggers transmitter release;  
presynaptic details unimportant

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

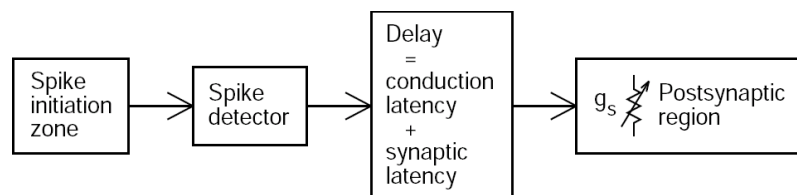


## Spike-triggered transmission: computational implementation

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

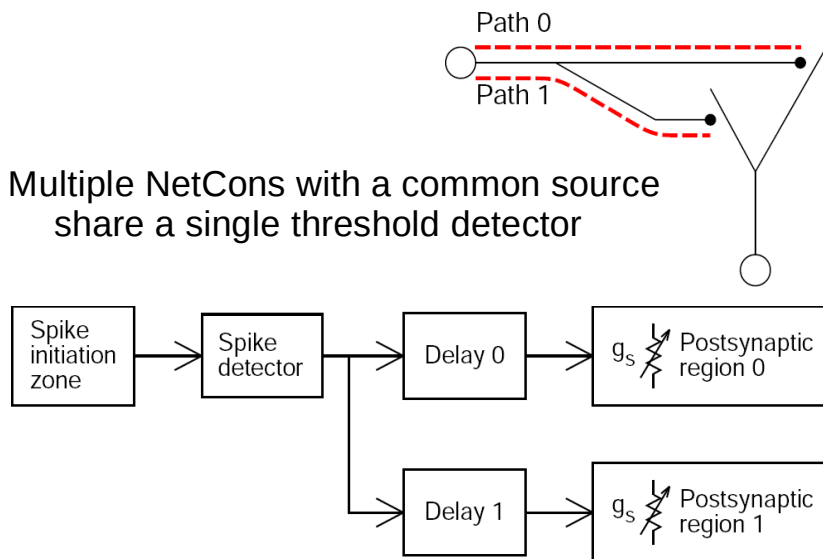
```
threshold = 10
delay = 1 // must be >= 0
weight = 0
```

NMODL specification of synaptic mechanism

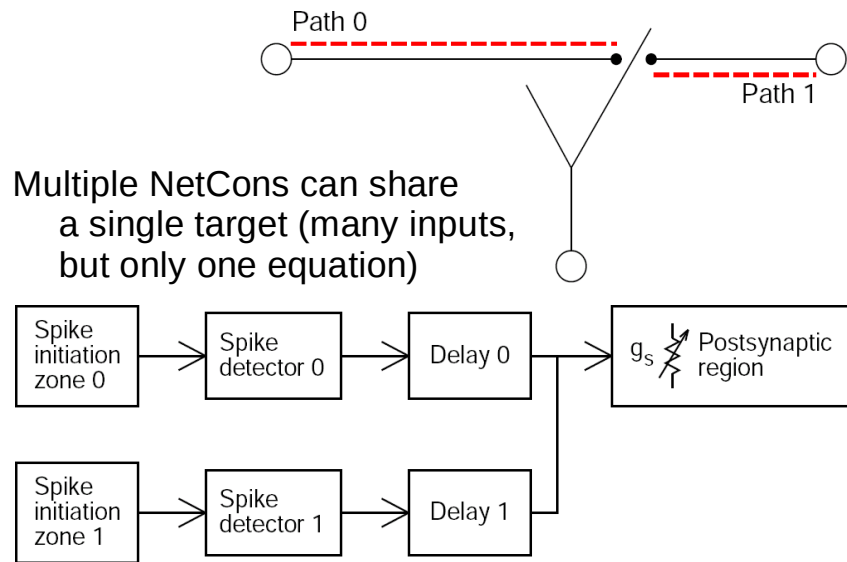
```
NET_RECEIVE(weight(microsiemens)) {
    . . .
}
```

## Efficient divergence

Multiple NetCons with a common source  
share a single threshold detector



## Efficient convergence

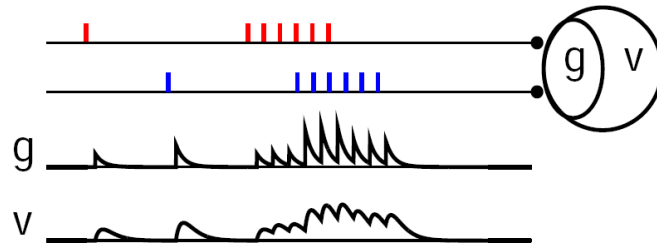


## Example: $g_s$ with fast rise and exponential decay

```

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*

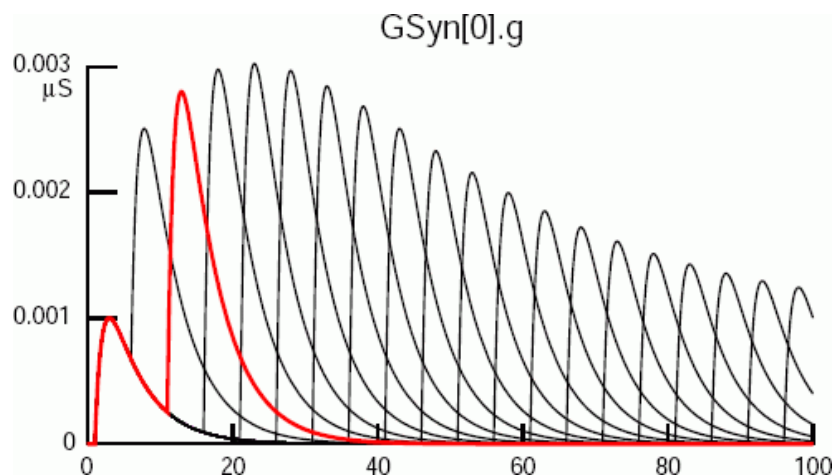


```

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

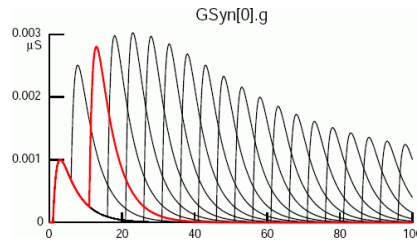


## Use-dependent synaptic plasticity *continued*

```

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

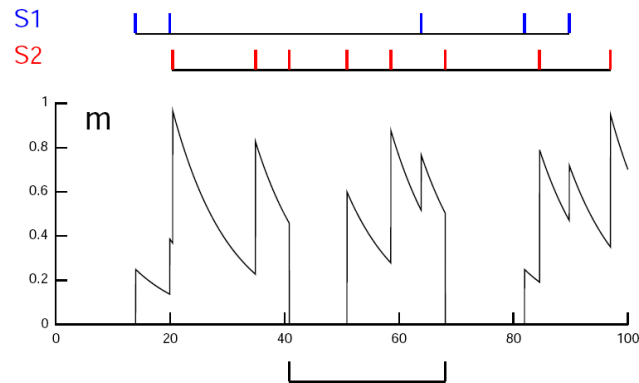
Event-driven simulation run time is

*proportional* to # of received events

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

Hybrid networks

## Example: leaky integrate and fire model



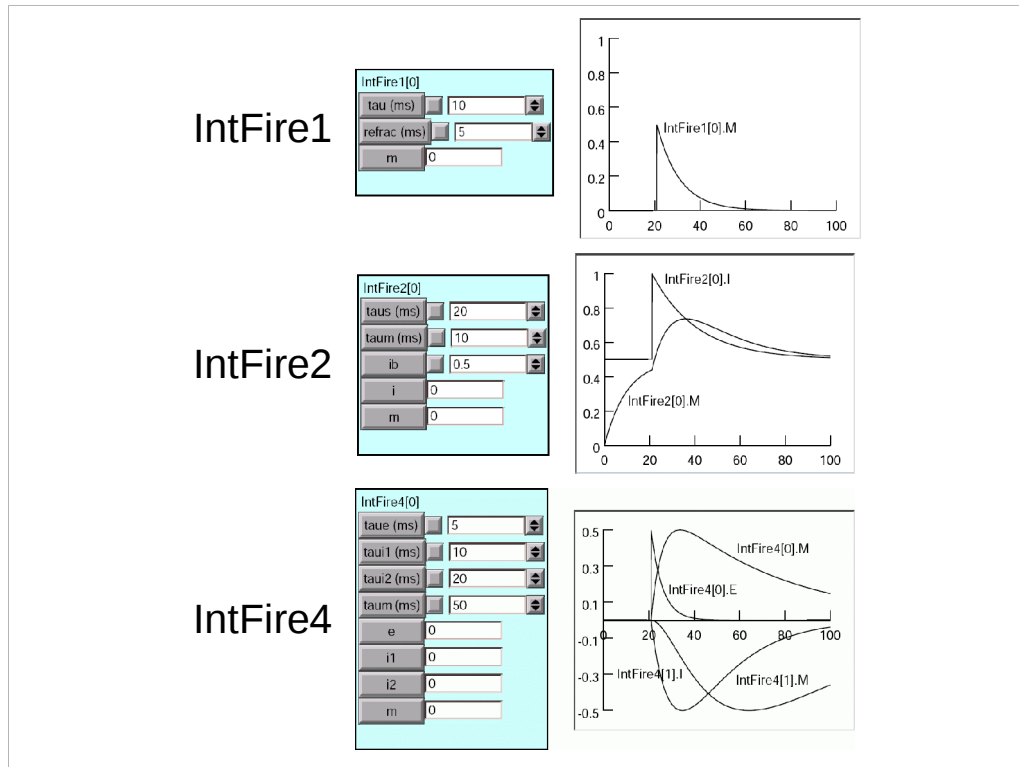
## Leaky integrate and fire model *continued*

```

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***



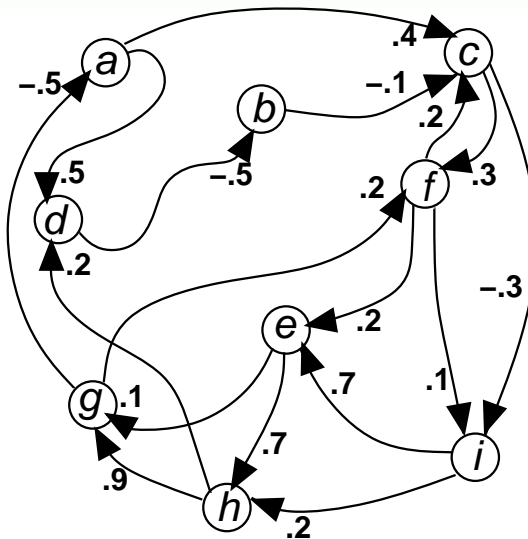
- 2. Simple network      3. Connections table      4. Connectivity matrix
- 5. Define connectivity    6. Hopfield-Brody synchronization model    7. Unconnected cells
- 8. Negative (inhibitory) connectivity      9. Q&D synchronization measure
- 10. Check sync with increasing inhib    11. Graph results    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. `cnode.netconlist()`    23. In addition to `cnode.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.7/4

## 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

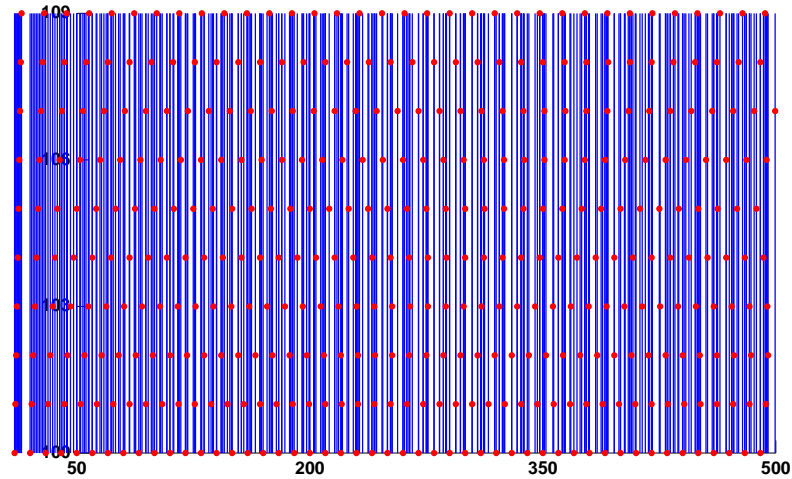
Wiring networks in Neuron – p.8/4



## Unconnected cells

$wt = -1e-5$

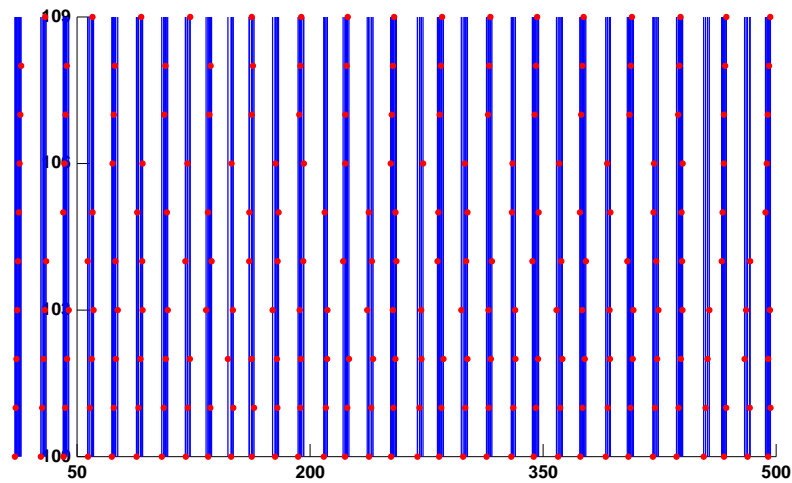
cell fire at own rates and go out of phase



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.11/42

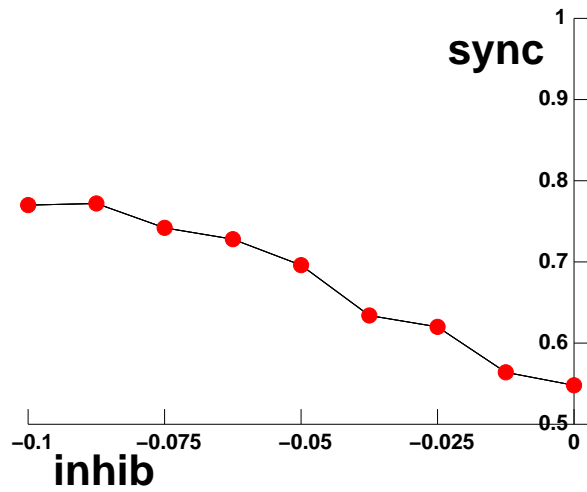
## Check sync with increasing inhib

```
// 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())
}
```

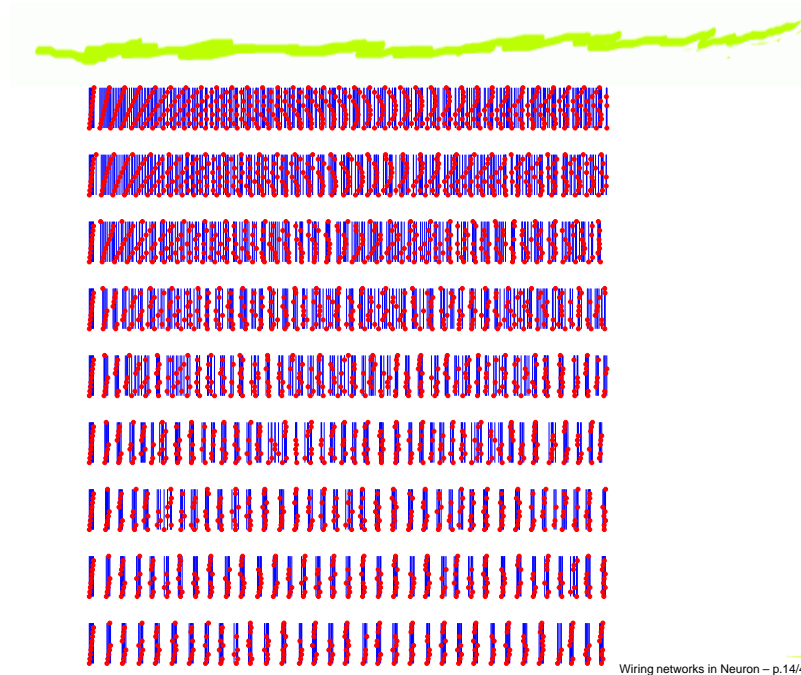
Wiring networks in Neuron – p.12/42

**Graph results**

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

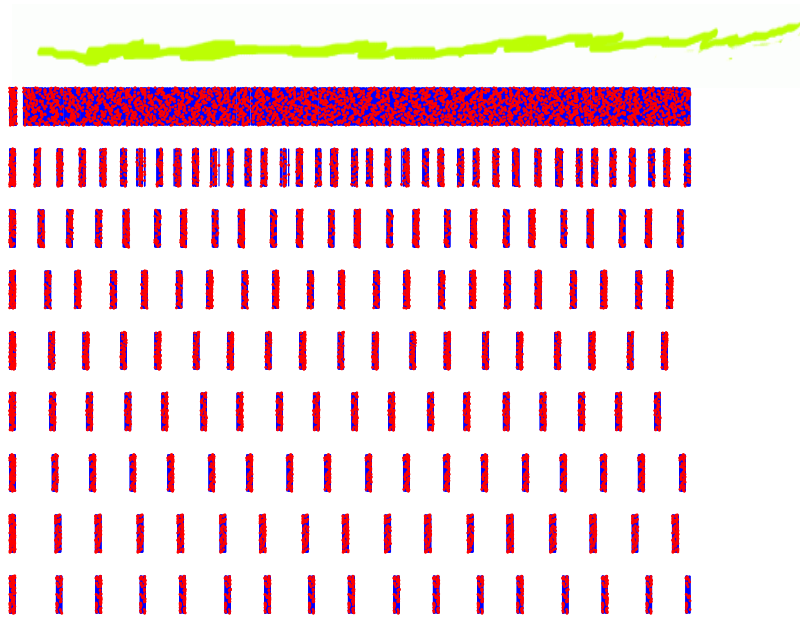


Wiring networks in Neuron – p.13/4;

**Confirm with raw data**

Wiring networks in Neuron – p.14/4;

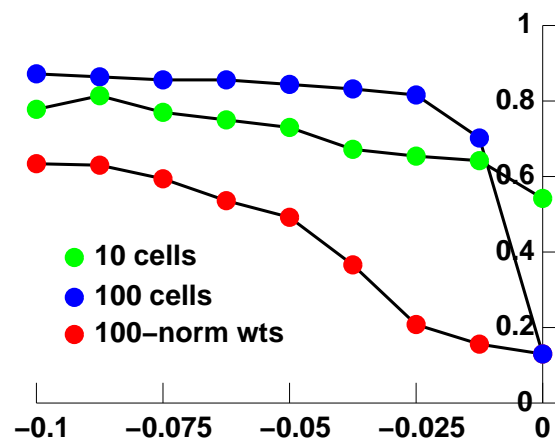
## Scale up to 100 cells



Wiring networks in Neuron – p.15/4;

## Need to normalize weights

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



Wiring networks in Neuron – p.16/4;

## 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.17/4;

## 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.18/4;

**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.19/42

**Rewiring for different densities**

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

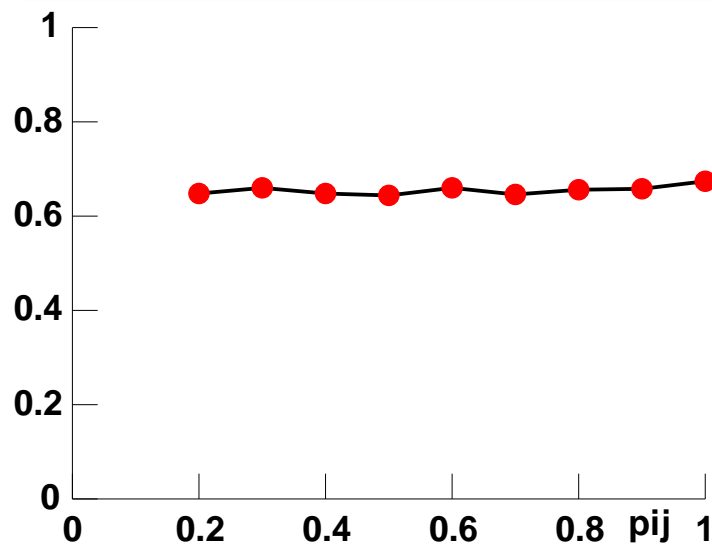
```
// don't create if syn_num is not on list
// note that num of nclist.object(num)
//   no longer meaningful
// here array better: 'objref nc[9900]'
for i=0,ncell-1 for j=0,ncell-1 {
  if (i!=j && !vec.contains(i*100+j)) {
    ... new NetCon ...
```

Wiring networks in Neuron – p.20/42

**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.25/4;

## ***fconn() – find connections***

```
// fconn(PREVEC,POSTVEC) places values of
// pre- and post-syn cells in parallel vector
// only lists pairs with non-zero connections
// getcnum() returns index of cell obj
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.26/4;

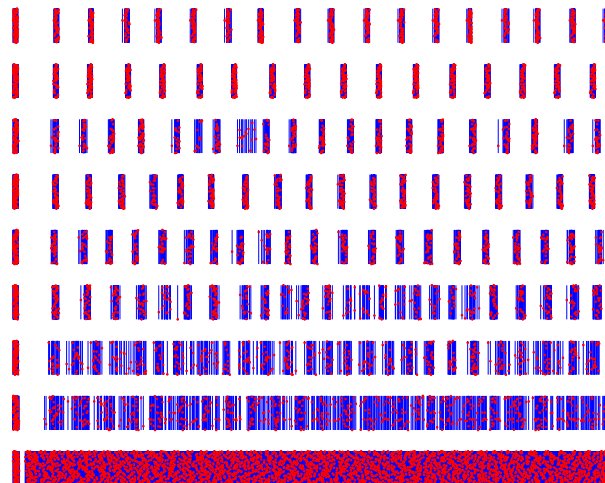
## Now can check non-zero connectivity

- 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

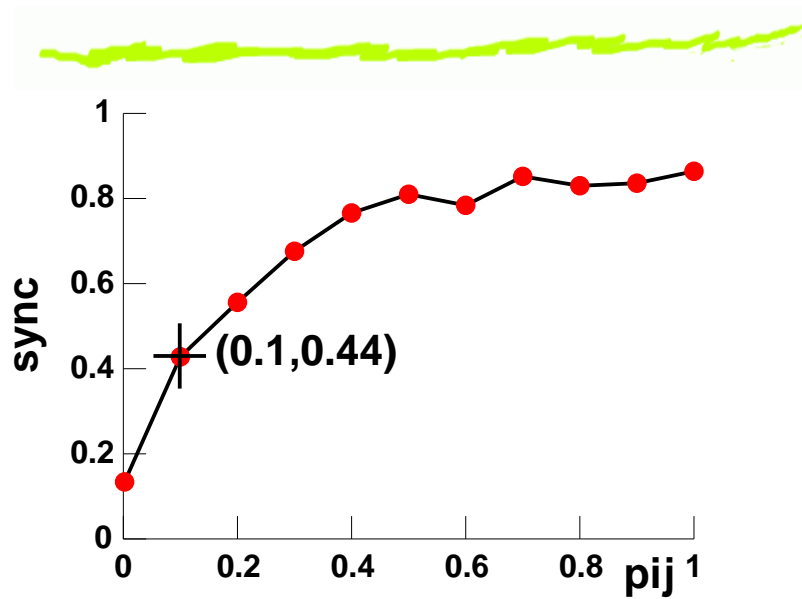
Wiring networks in Neuron – p.27/4;

## Rewrite randomizer

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



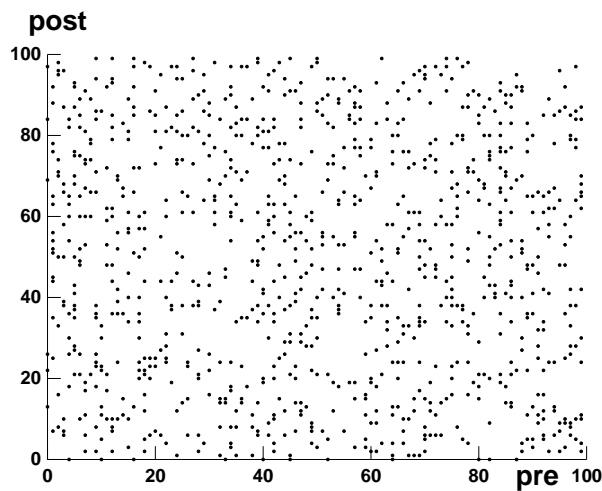
Wiring networks in Neuron – p.28/4;

**Synchronization measure**

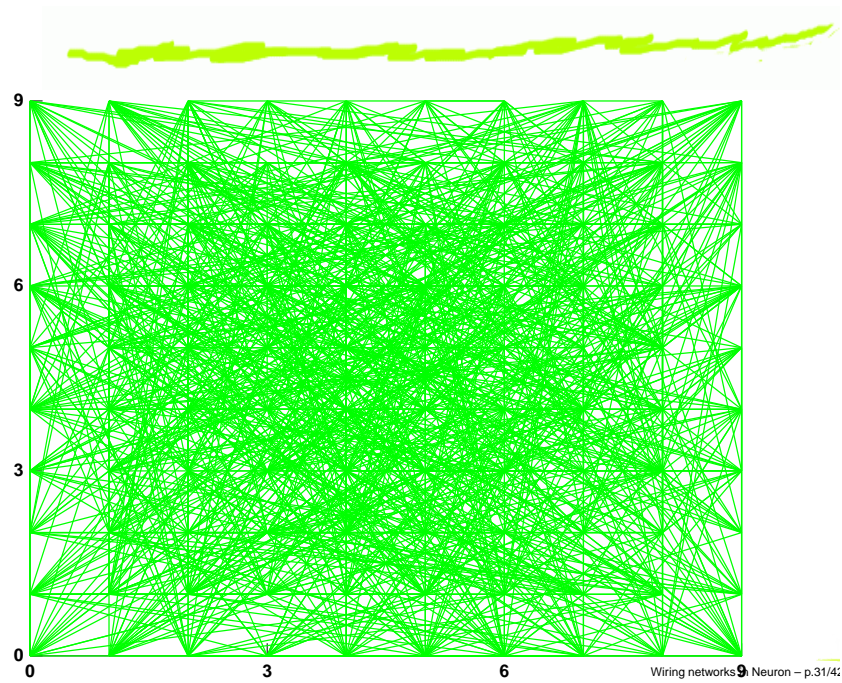
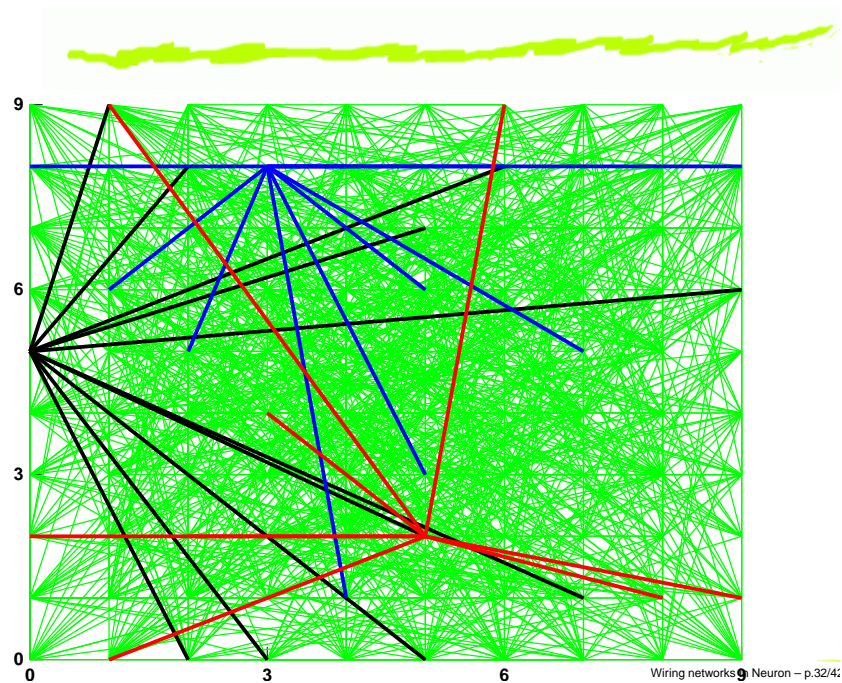
Wiring networks in Neuron – p.29/4;

**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;

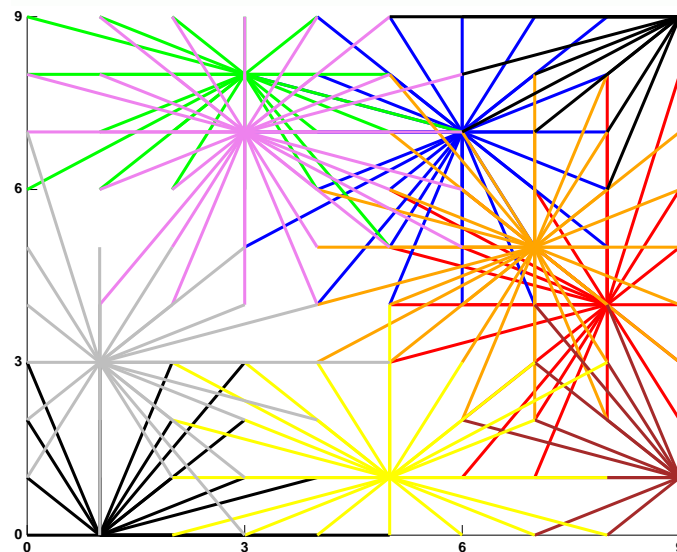
## Random wiring with distance fall-off

- 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

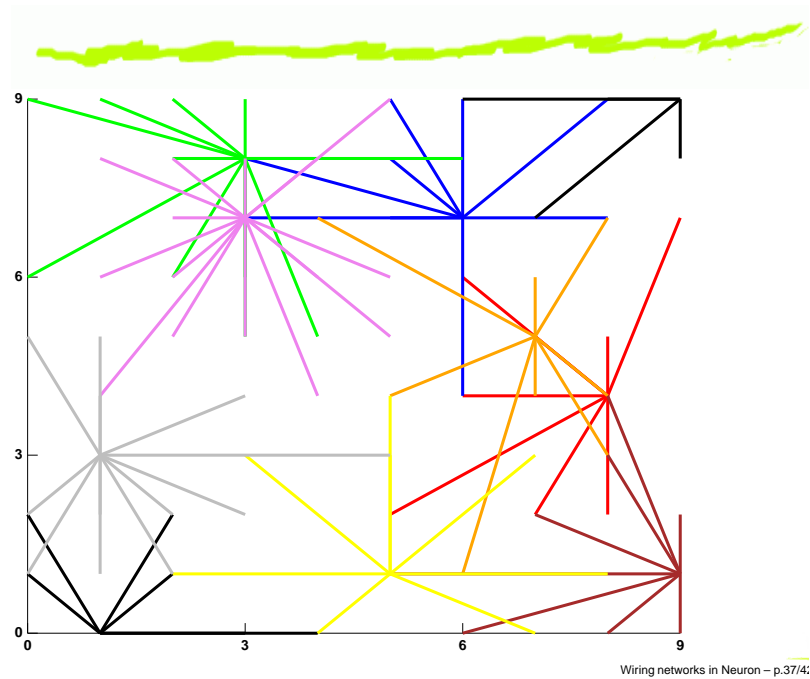
Wiring networks in Neuron – p.35/42

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

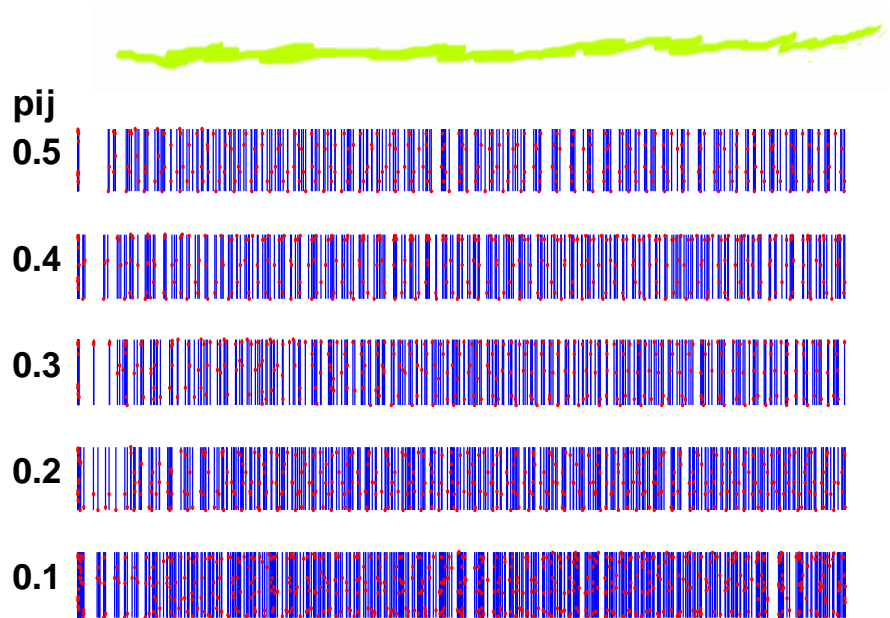


Wiring networks in Neuron – p.36/42

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

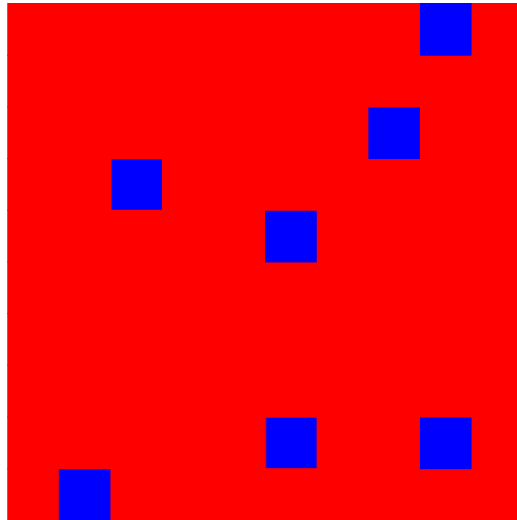


**Doesn't sync very well**



## *Is there localized sync'ing?*

Look at animation.



Wiring networks in Neuron – p.39/40

## *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.40/41



## ***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

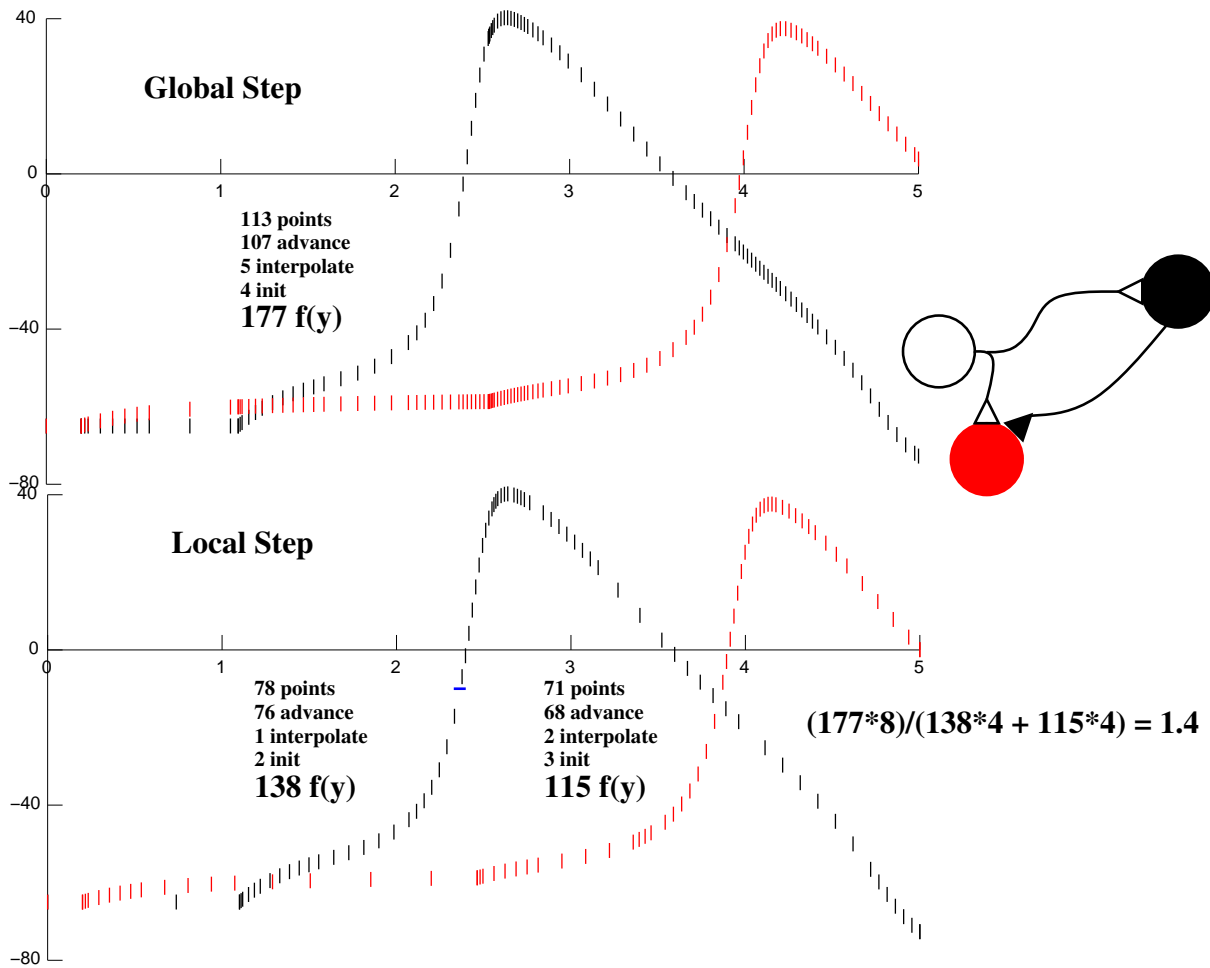
Wiring networks in Neuron – p.41/4;

## ***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;





**One integrator instance per cell**

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

**ta**      **t**      **tb**

○      ●      ●

○

●

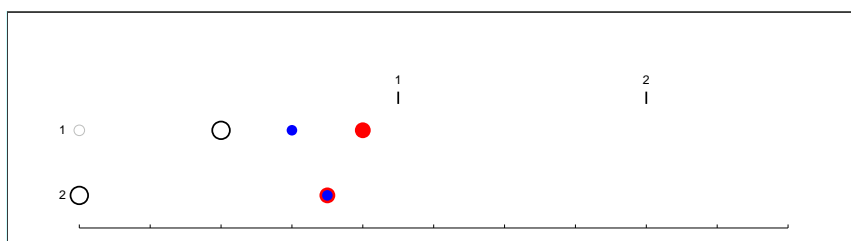
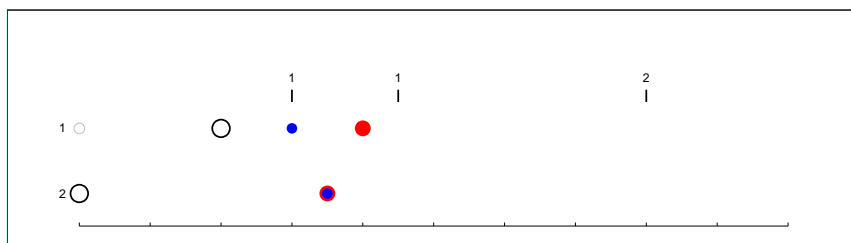
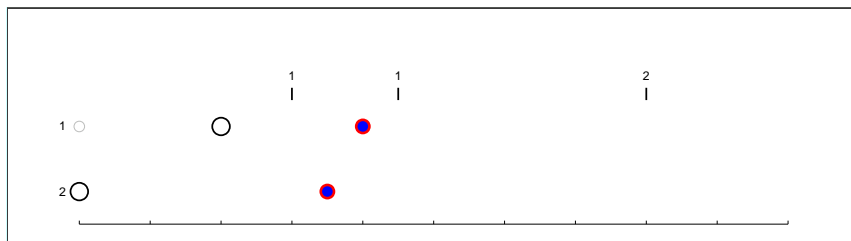
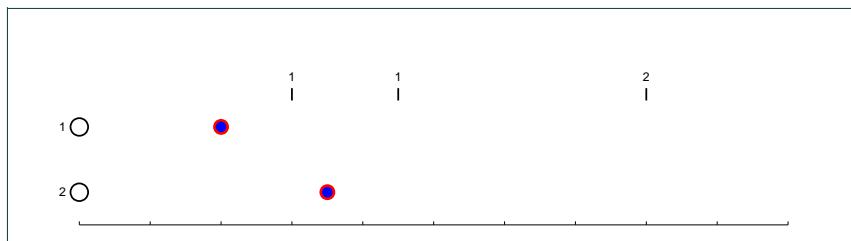
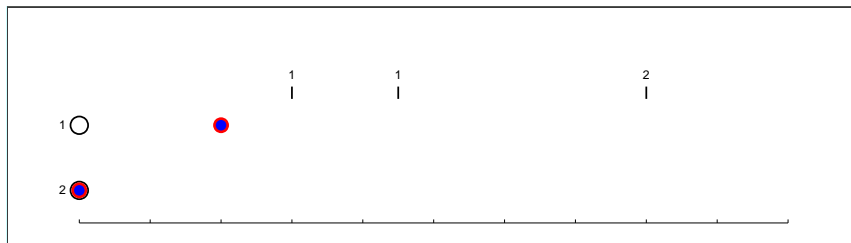
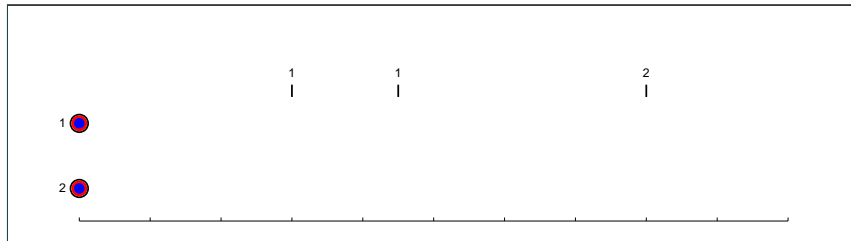
**advance**

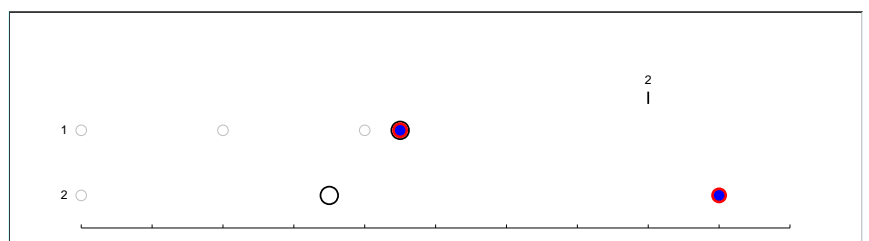
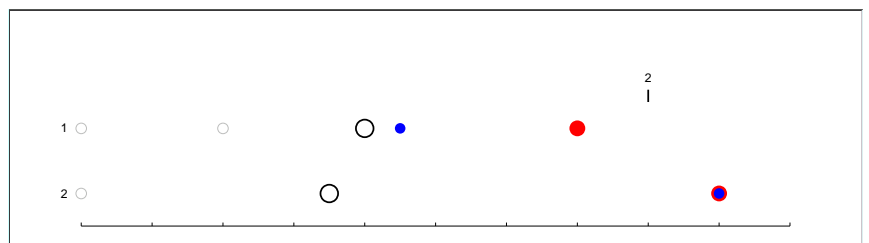
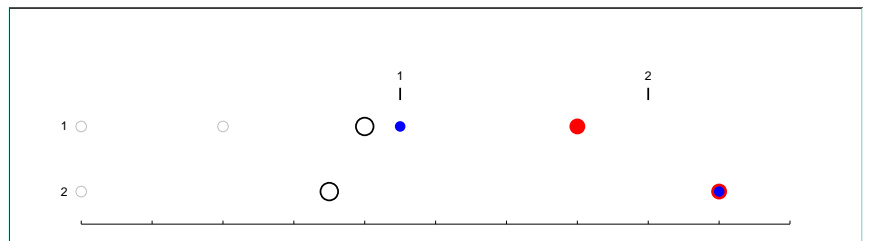
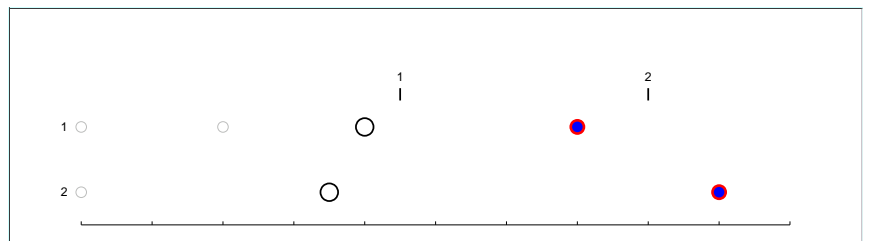
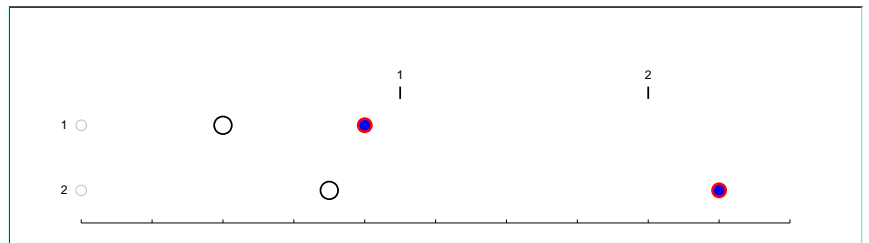
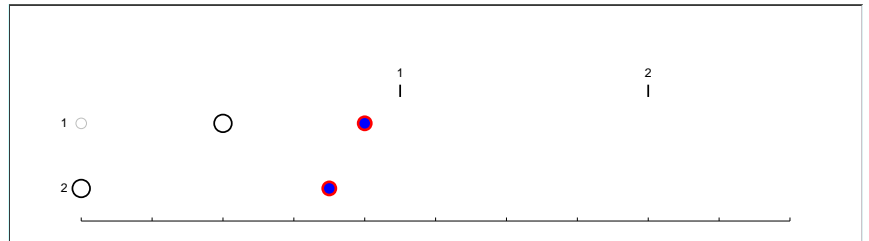
○      ●      ●

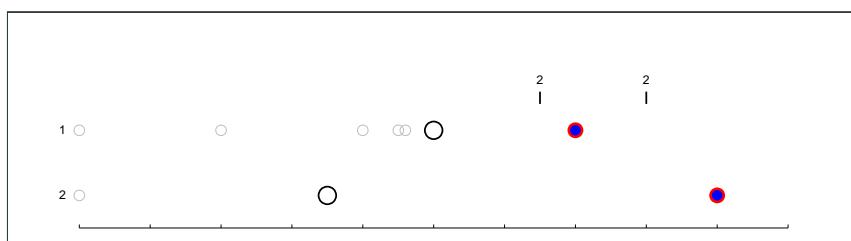
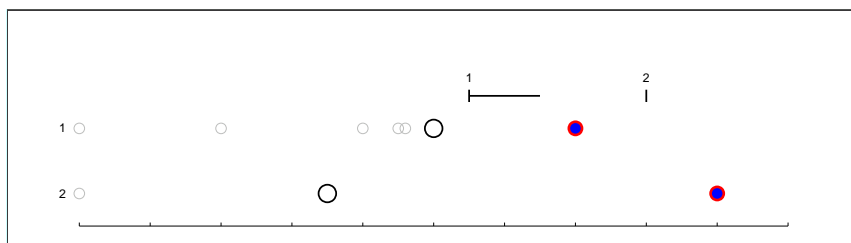
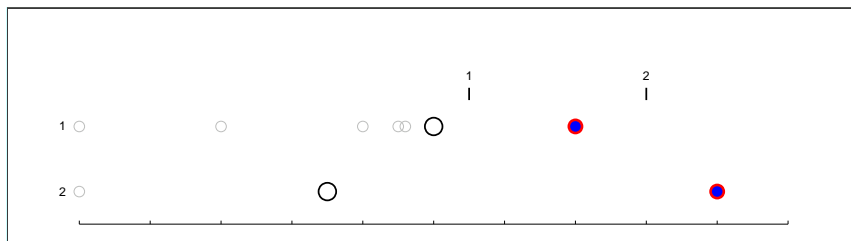
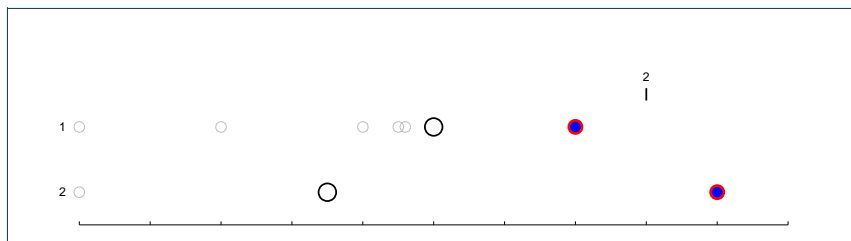
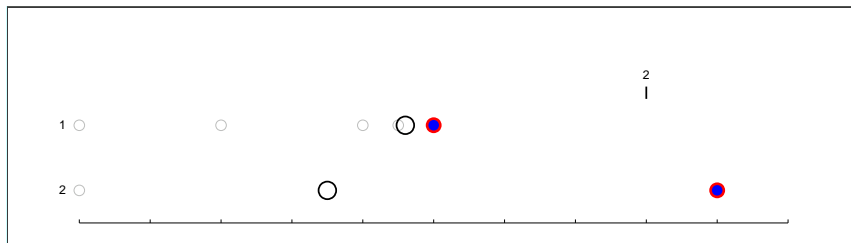
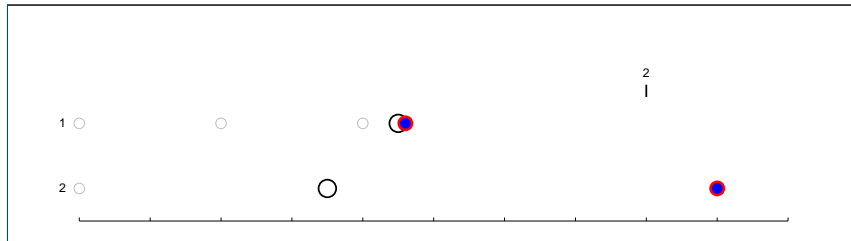
**interpolate**

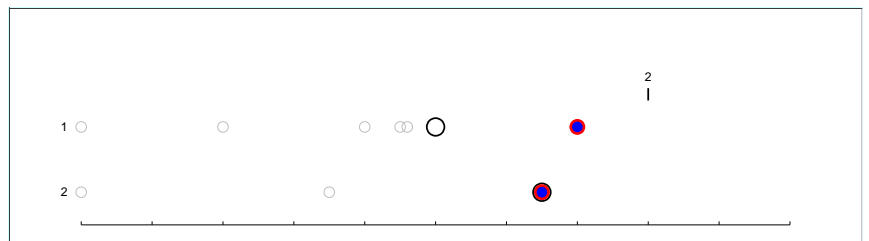
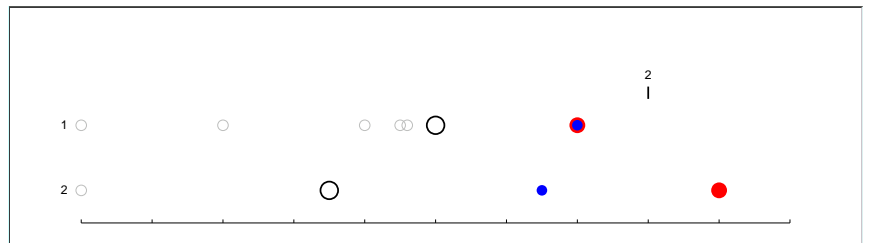
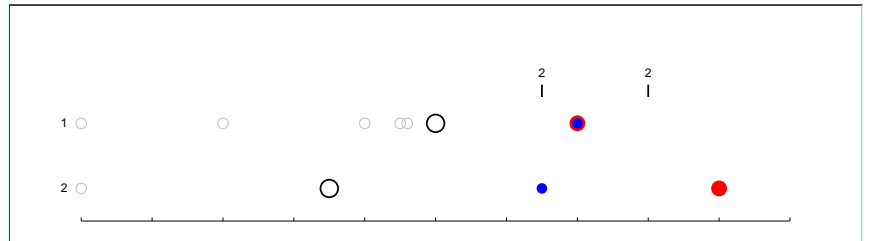
●

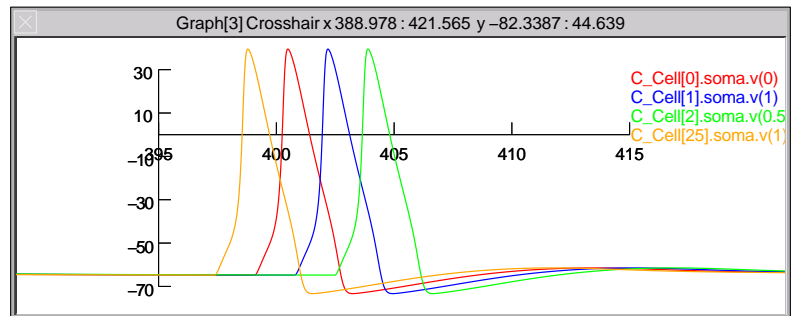
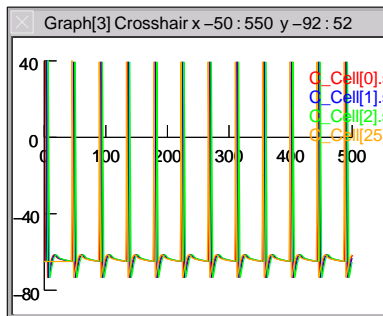
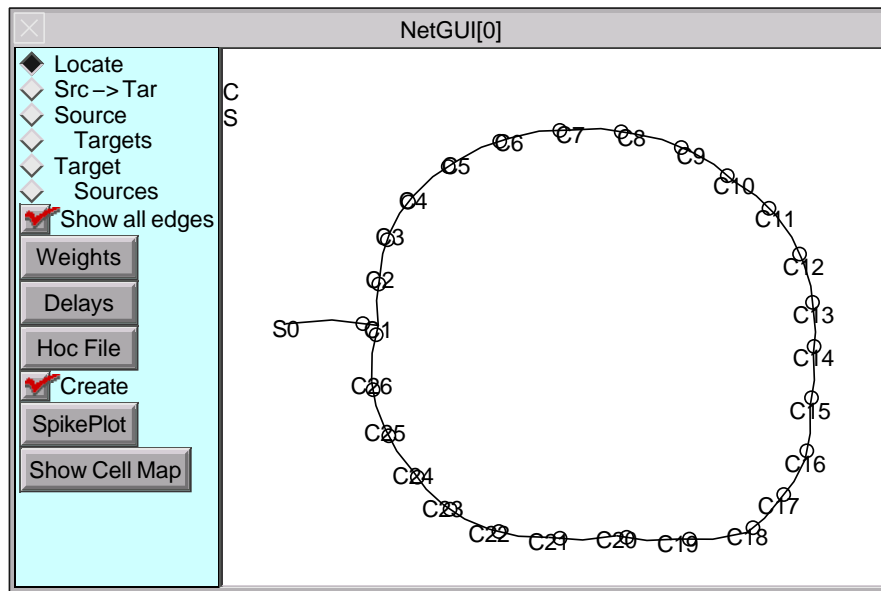
**init**



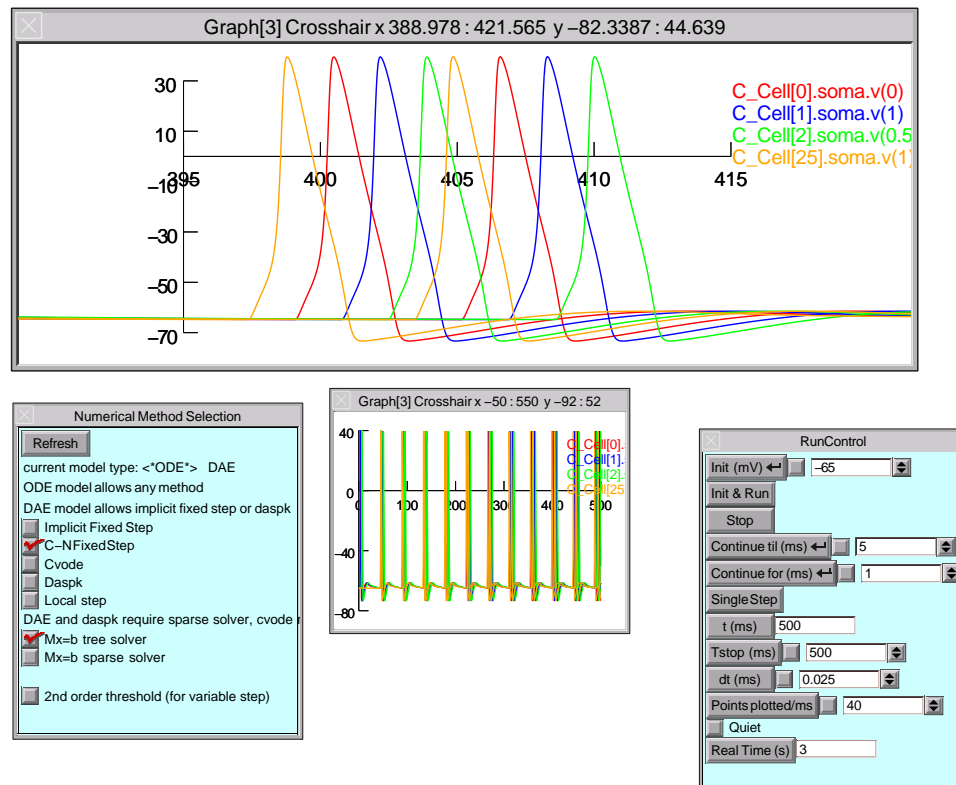
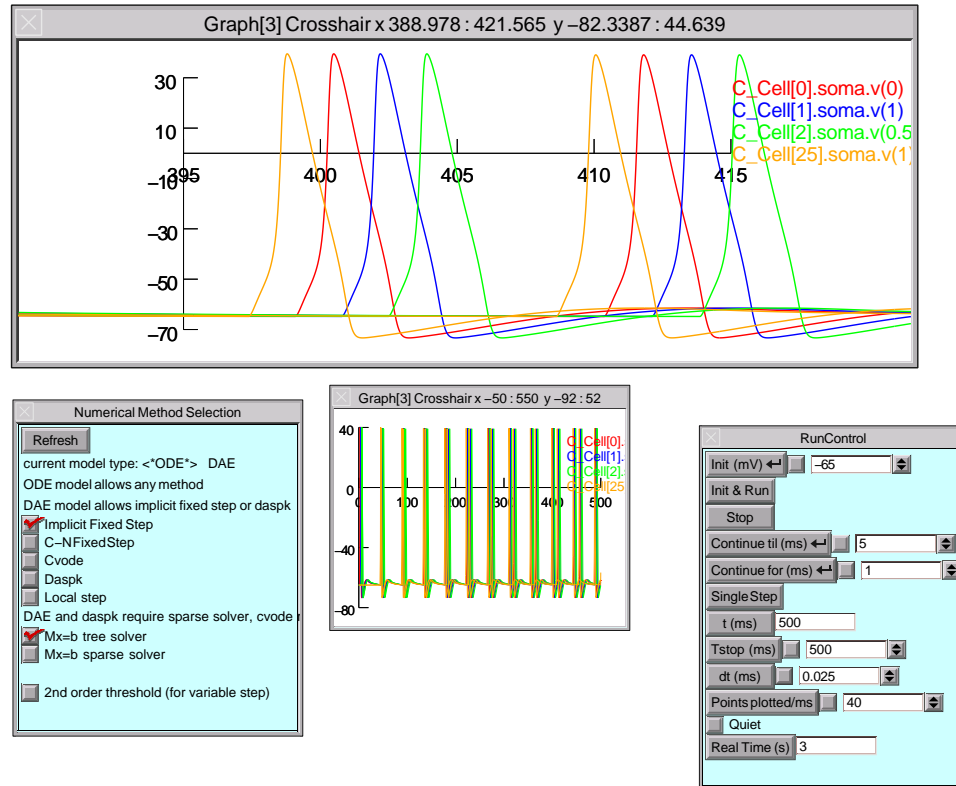


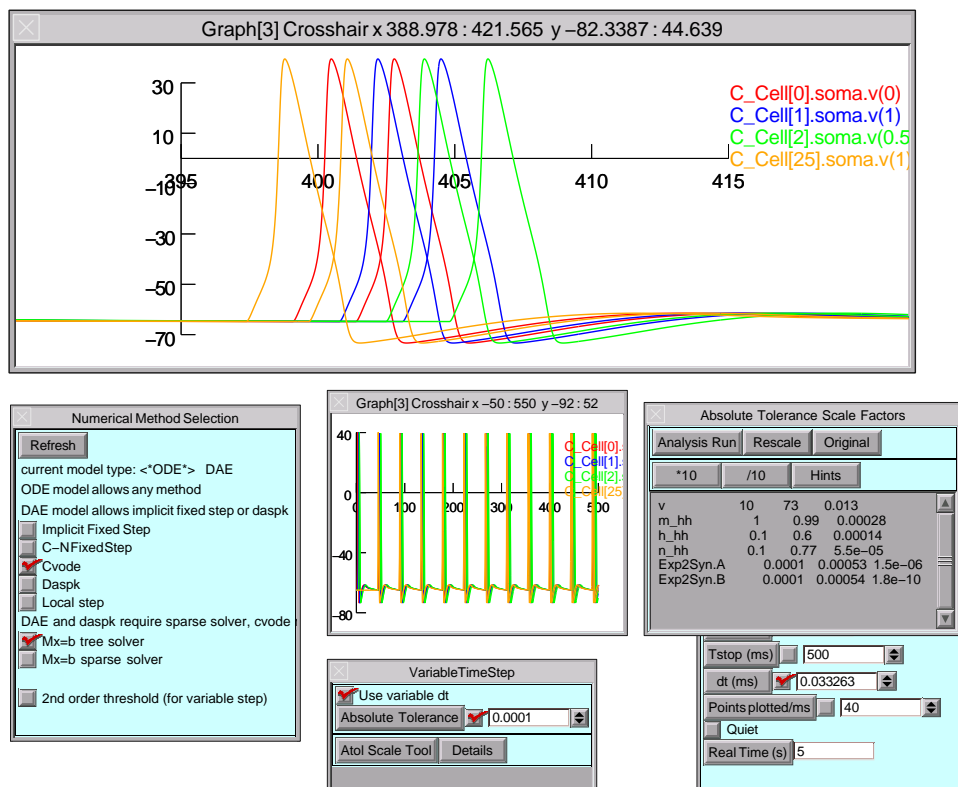
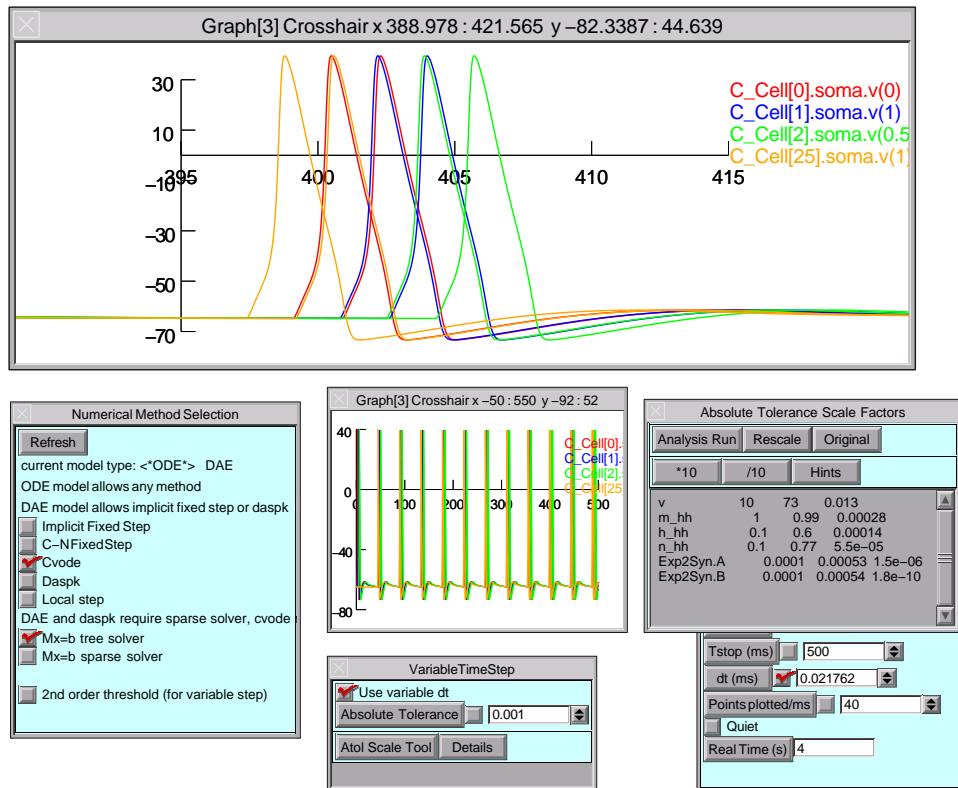


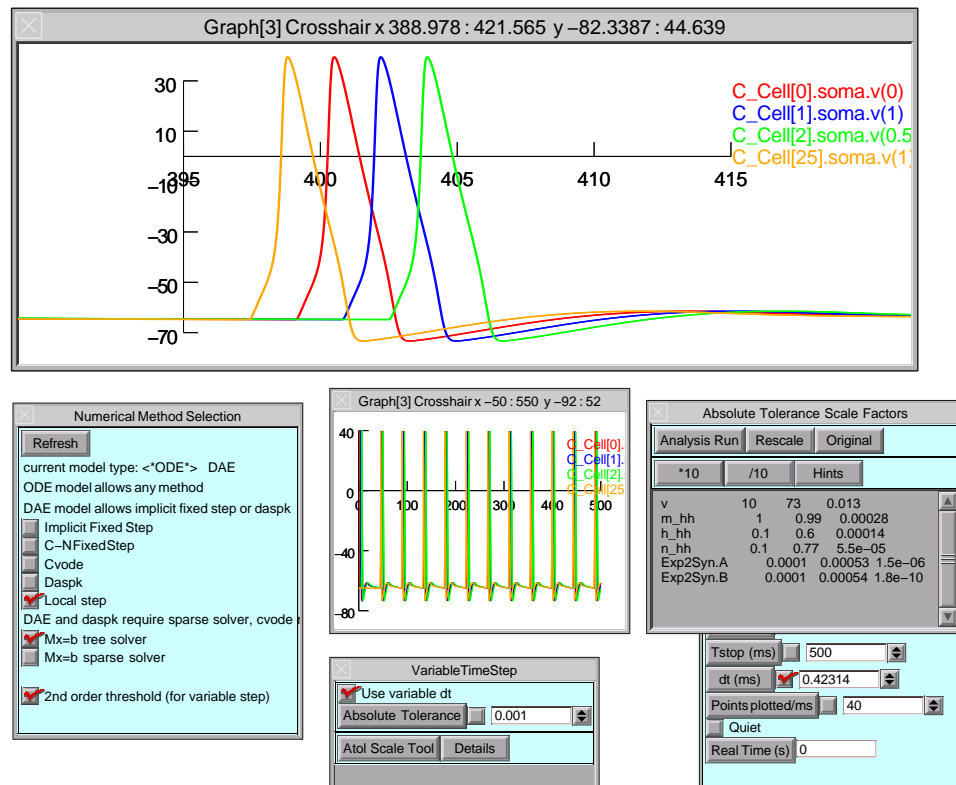
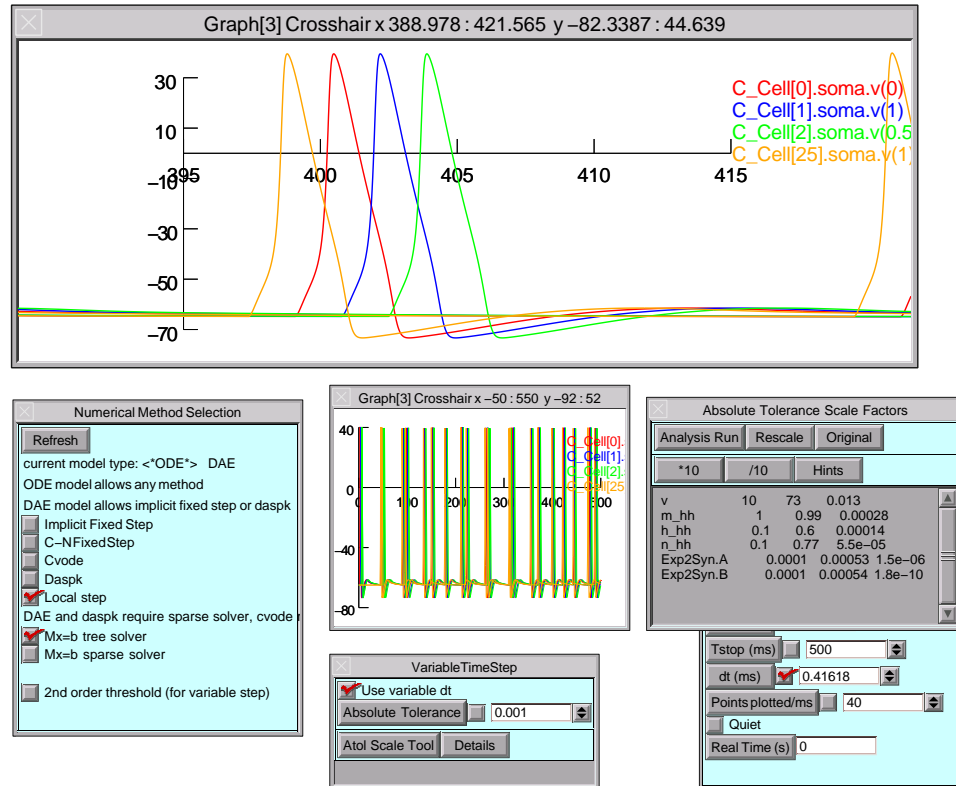












```
STATE { o }
```

```
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).

```

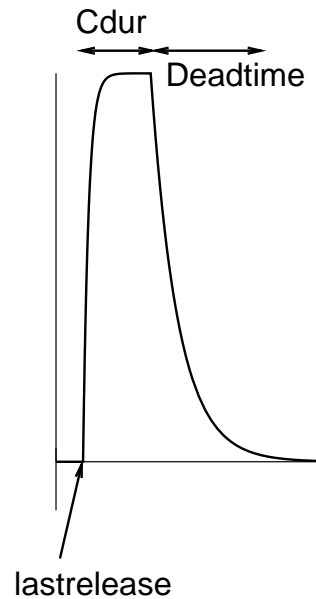
PROCEDURE release() { LOCAL q
:will crash if user hasn't set pre with the connect statement

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
  }
}
}

```

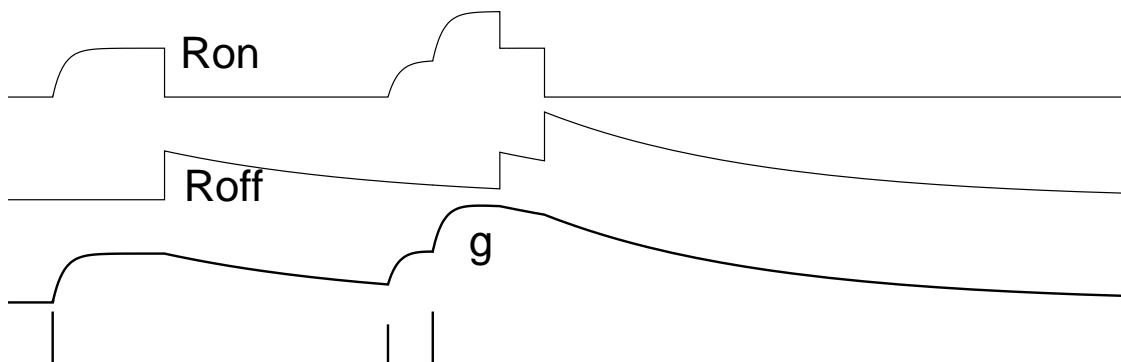
```

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() }
}
```



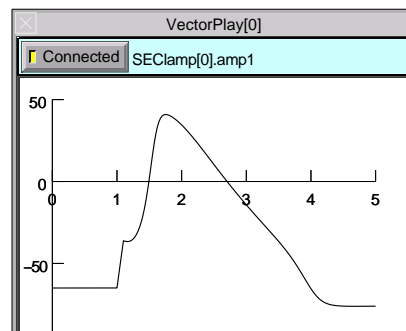
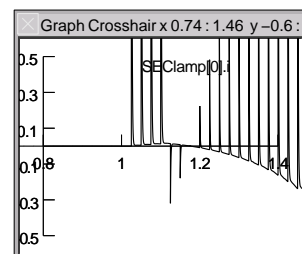
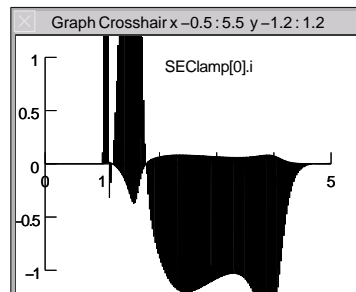
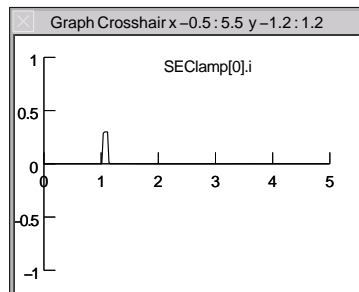
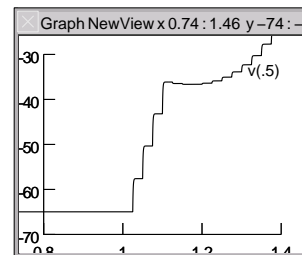
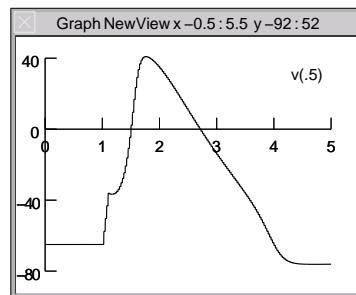
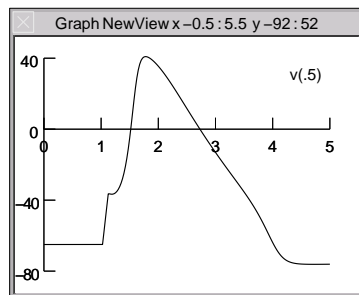
```
fh = new FInitializeHandler("ev()")
proc ev() {
  ccode.event(t1, "p()")
}
```

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

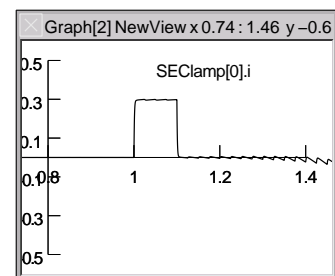
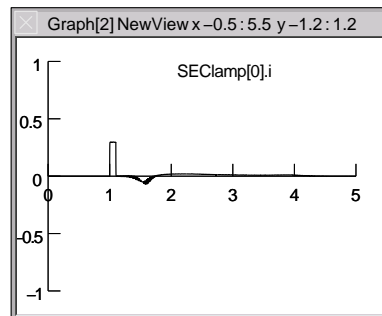
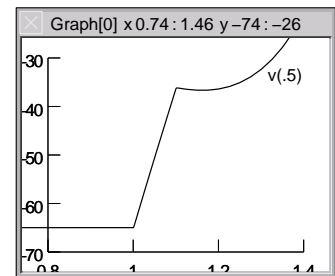
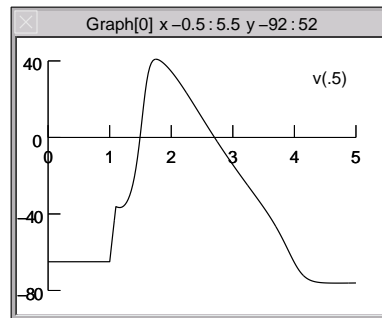


```
soma { nc = new NetCon(&v(.5), nil)}
nc.threshold = 10
nc.record("p()")
```

```
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

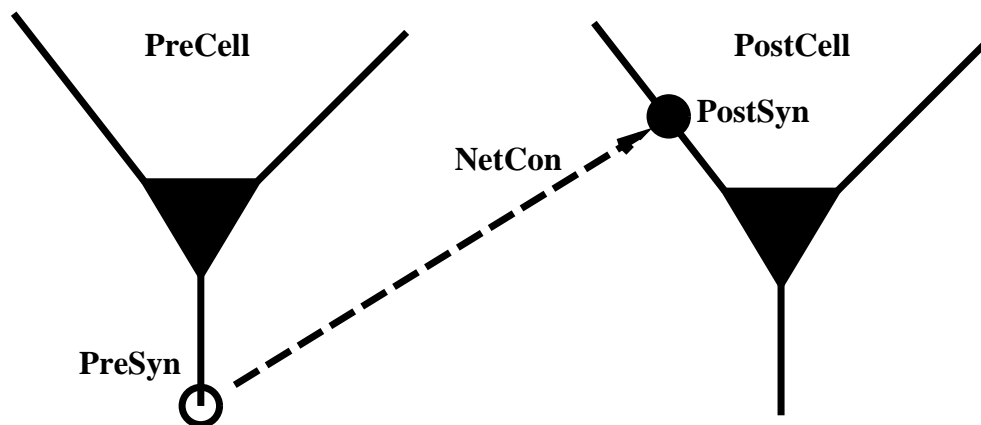
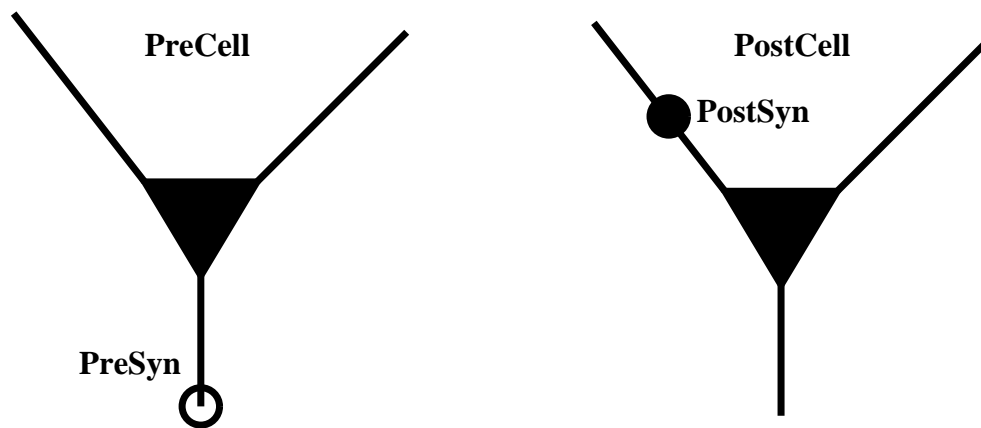
## Parallel Computation

A simulation run takes hours.

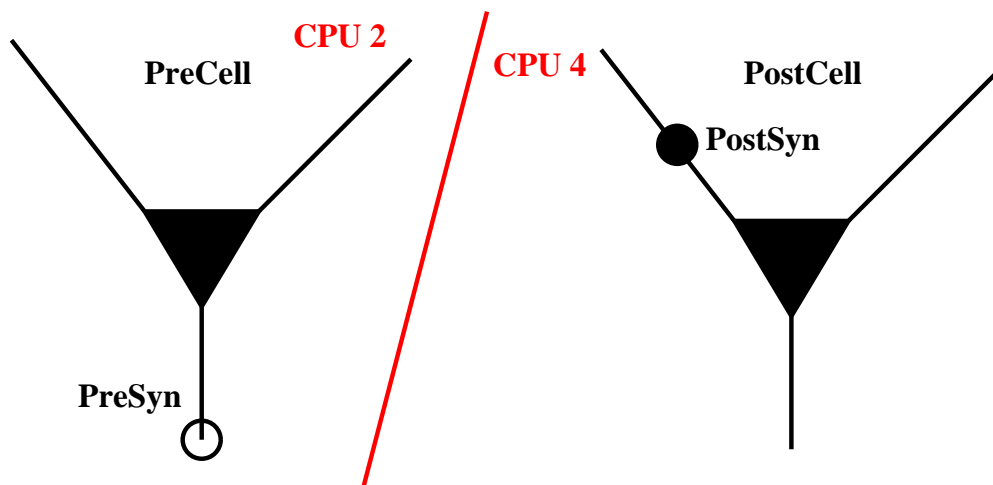
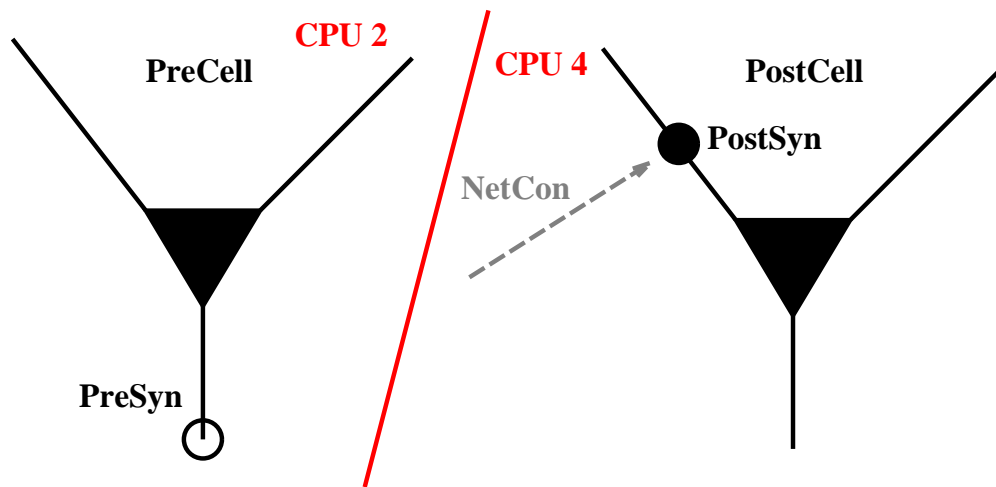
want to spread the problem over several machines.

Single cells

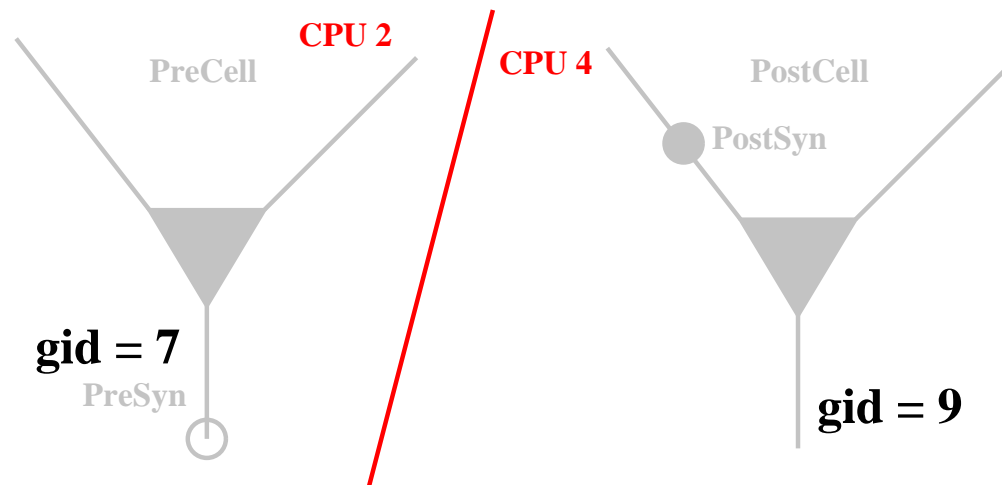
portions of the tree cable equation on  
different machines.



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



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



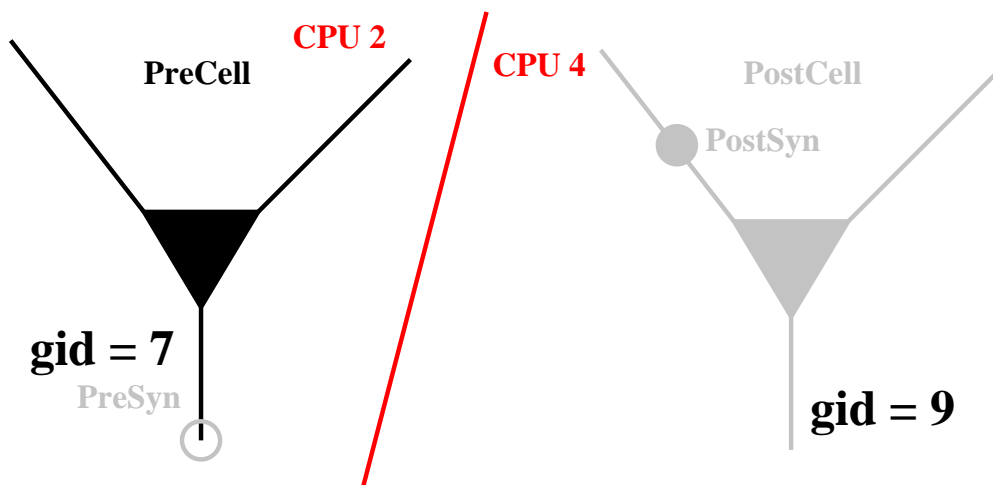
**Every spike source (cell) must have a global id number.**

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:**

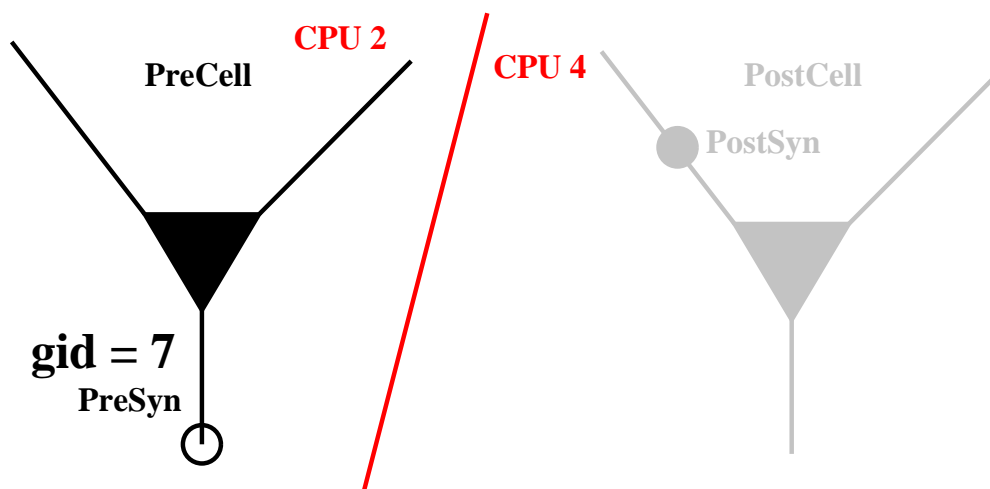
```
for (gid = pc.id; gid < ncell; gid += pc.nhost)
    pc.set_gid2node(gid, pc.id)
    ...
}
```

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



Create cell only where the gid exists.

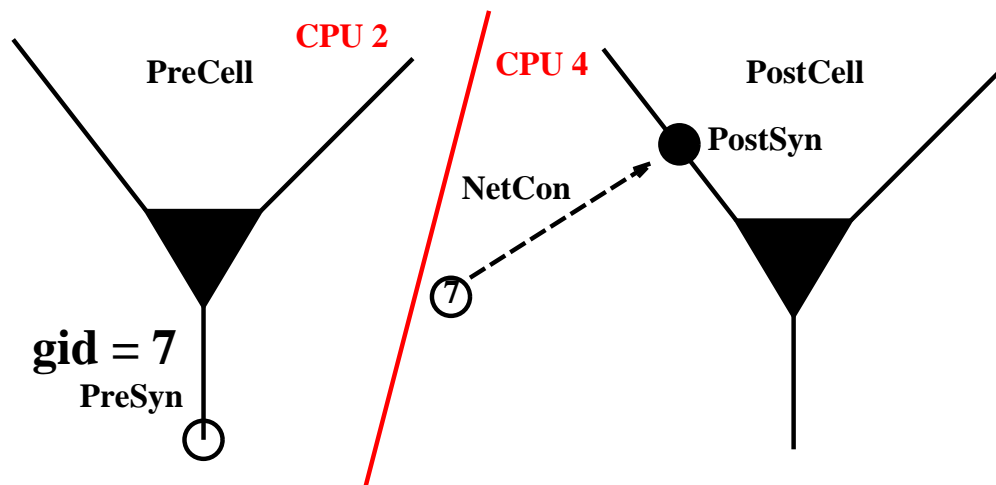
```
if (pc.gid_exists(7)) {
    PreCell = new Cell()
}
```



Associate gid with spike source.

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





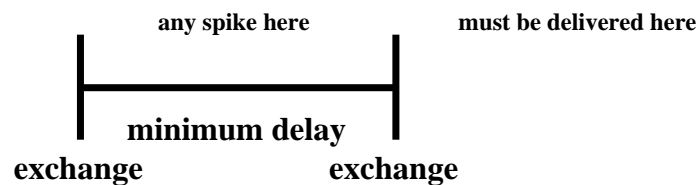
**Create NetCon on CPU where target exists.**

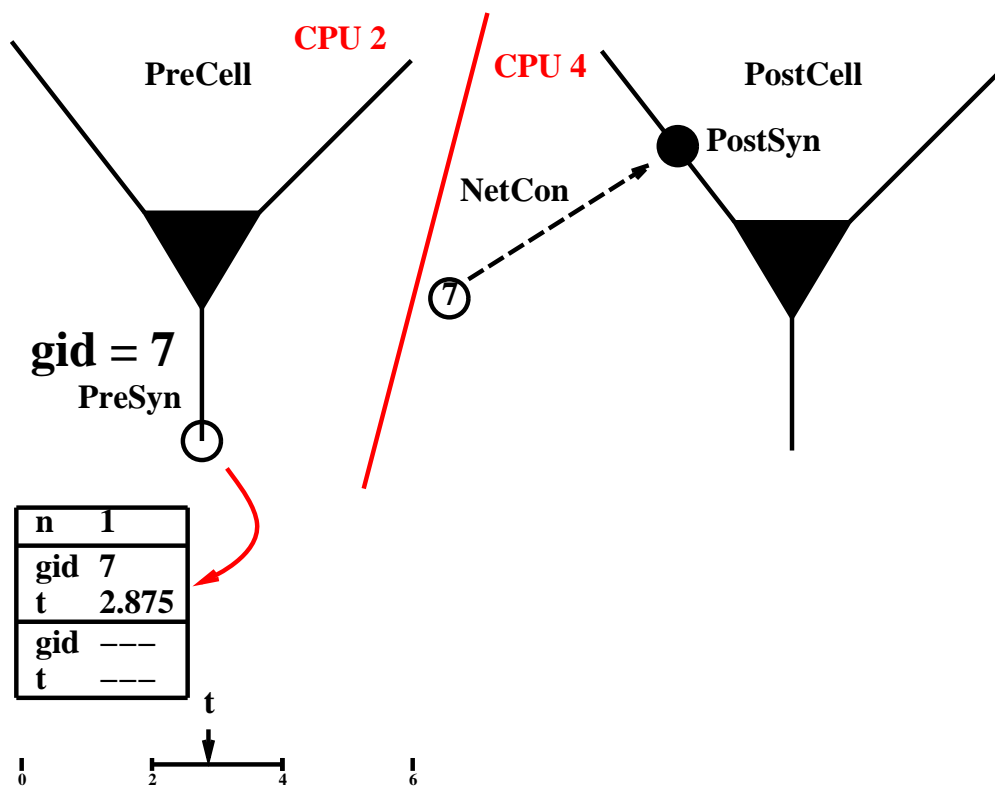
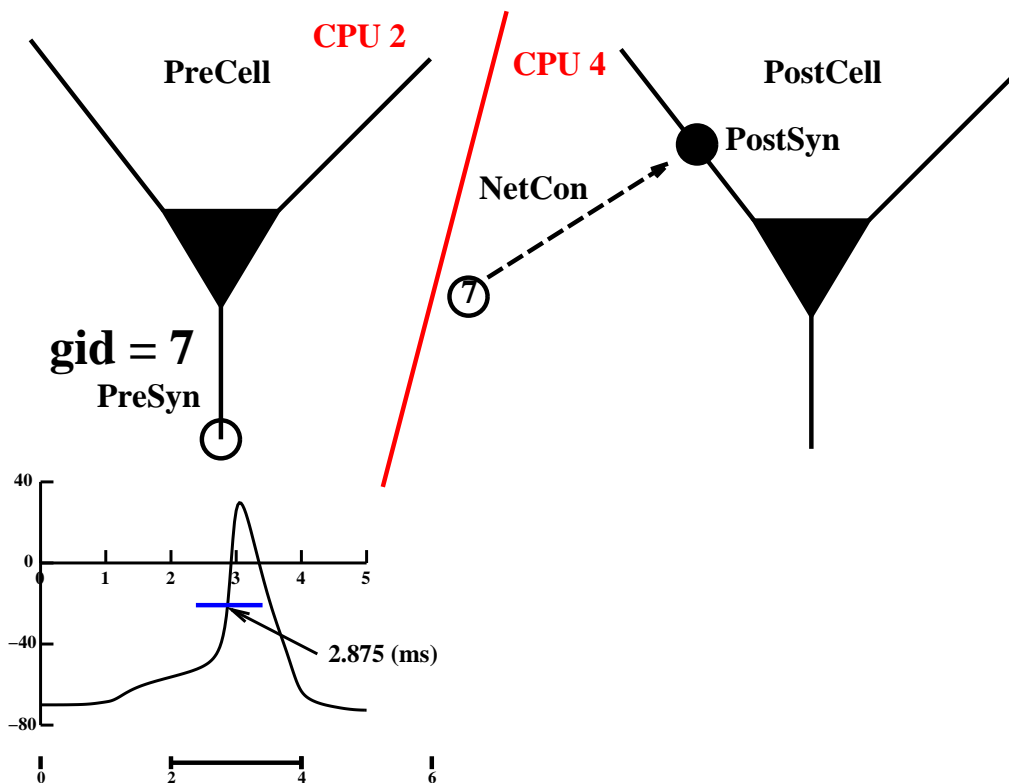
```
nc = pc.gid_connect(7, PostSyn
```

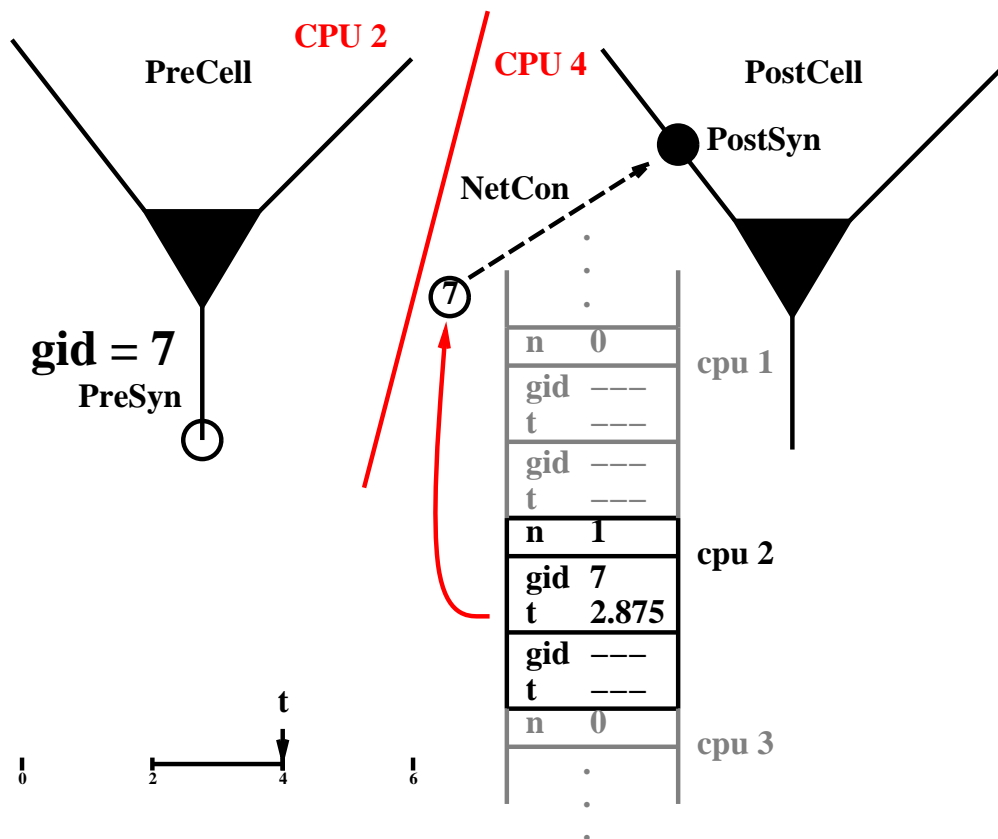
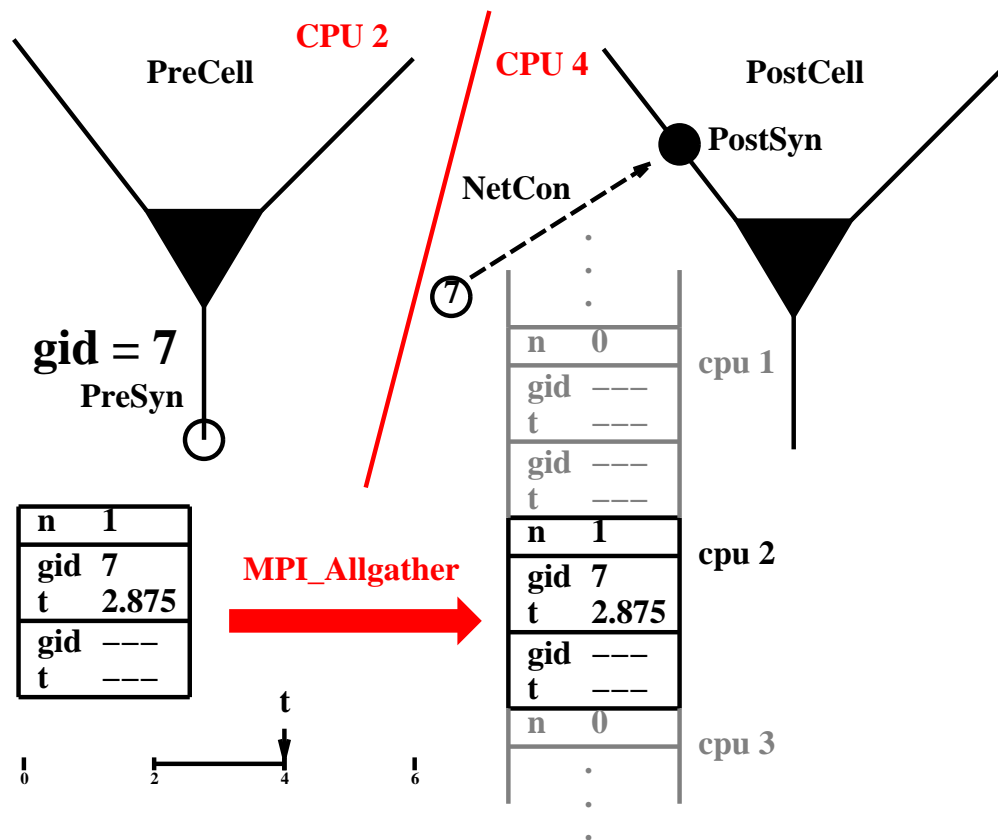
## Run using the idiom

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

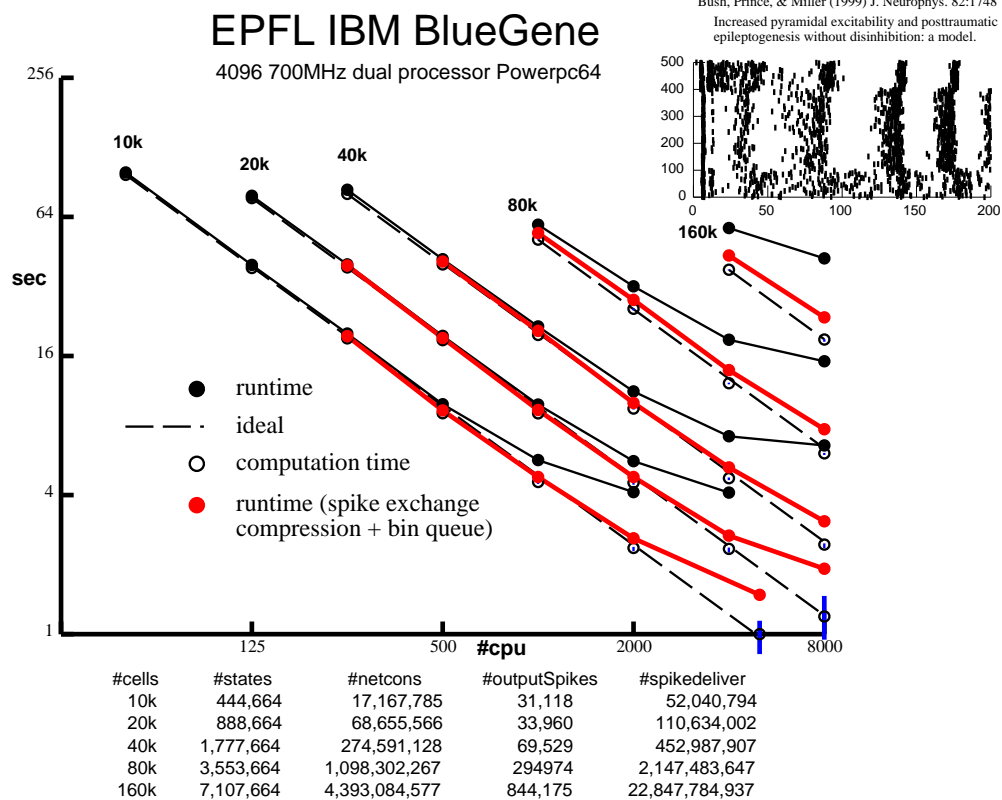
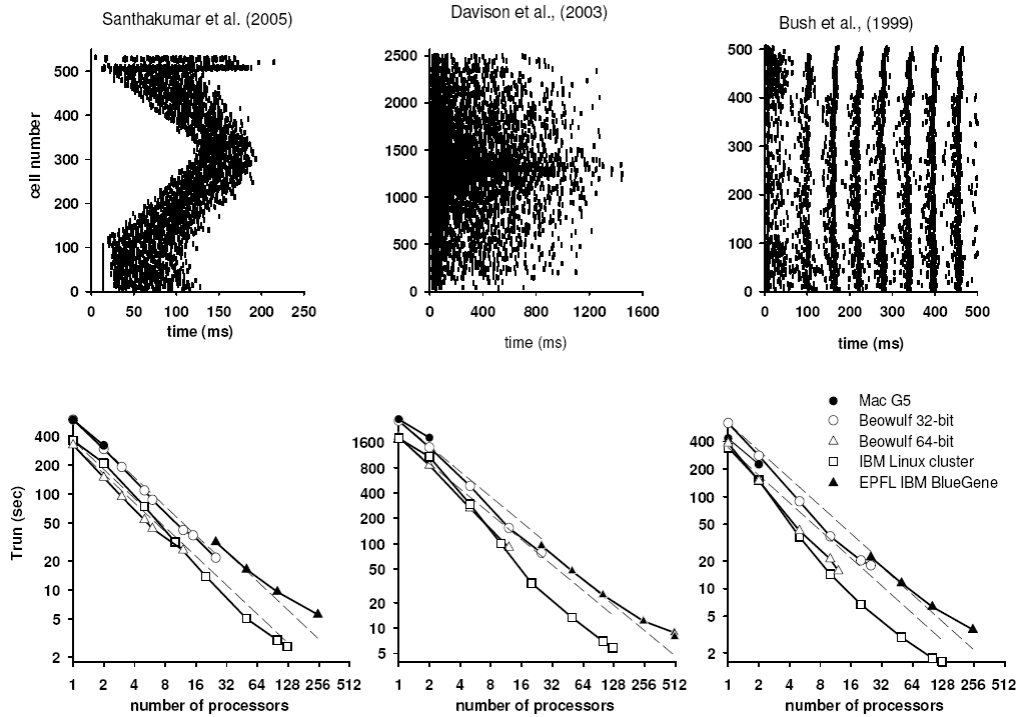
**pc.set\_maxstep()** uses  
**MPI\_Allreduce**  
to determine minimum delay.



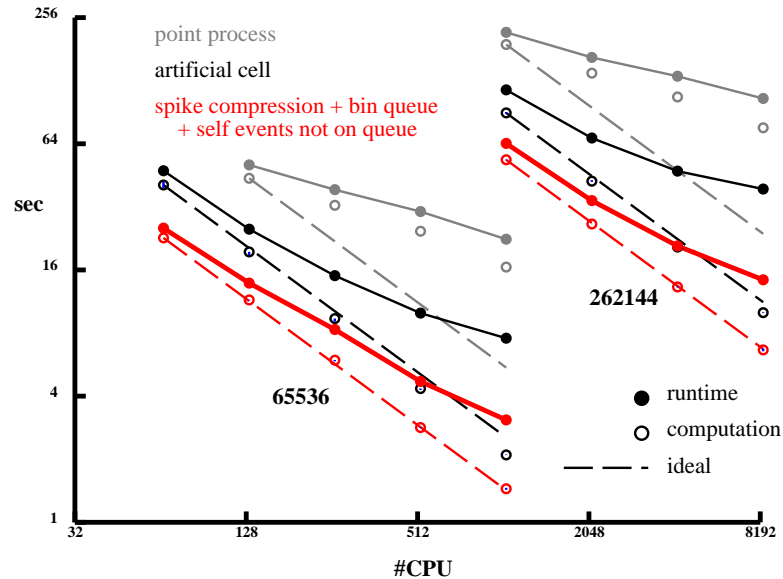




Migliore et al (2006) J. Comput. Neurosci. 21(2):119



## Artificial Spiking Net Performance

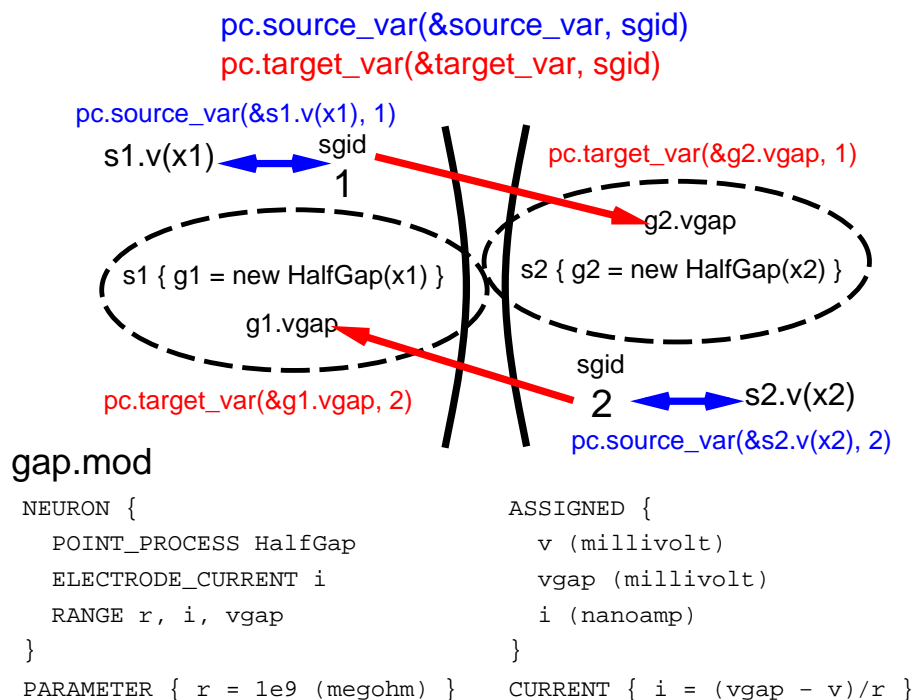


Each cell fires randomly every 10 to 20 ms.  
65K cells, 1000 random connections per cell  
256K cells, 10,000 random connections per cell

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

## Gap Junction Specification

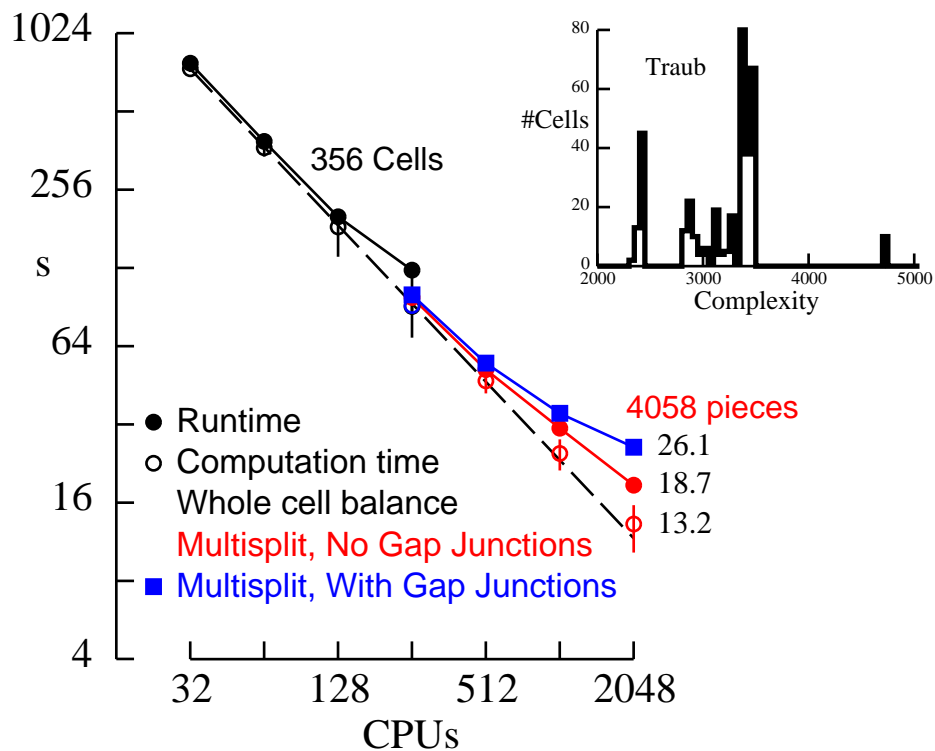
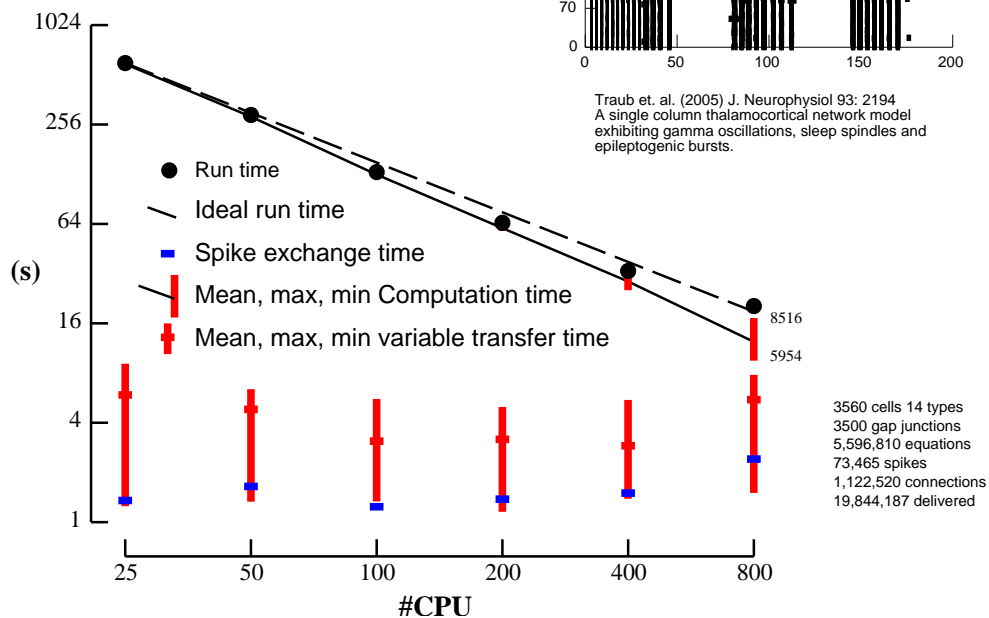
### Continuous Voltage Exchange



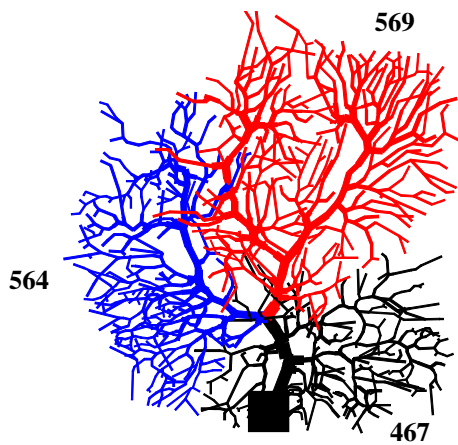
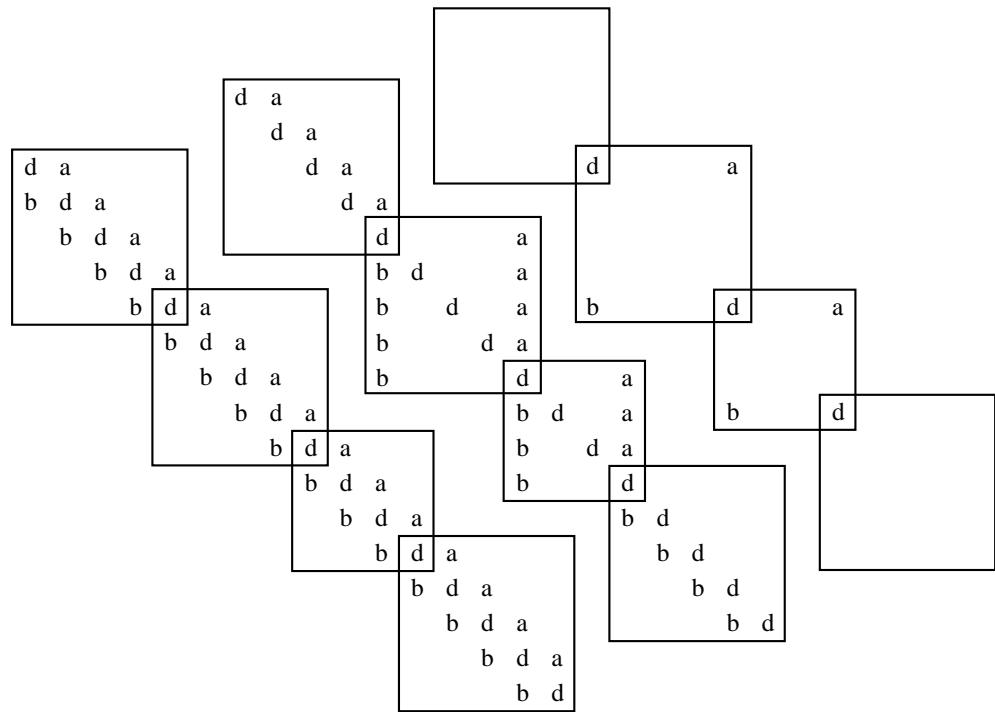
Pittsburgh Supercomputing Center

Bigben Cray XT3

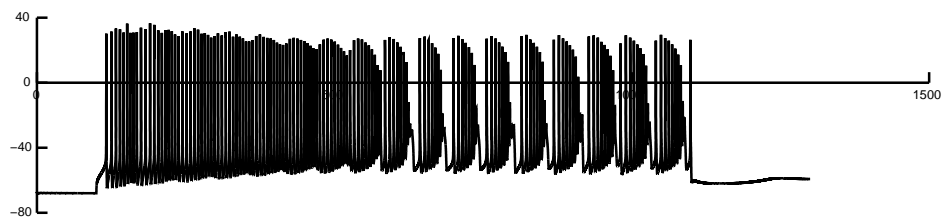
2068 2.4 GHz Opteron Processors



Multisplit Gaussian Elimination

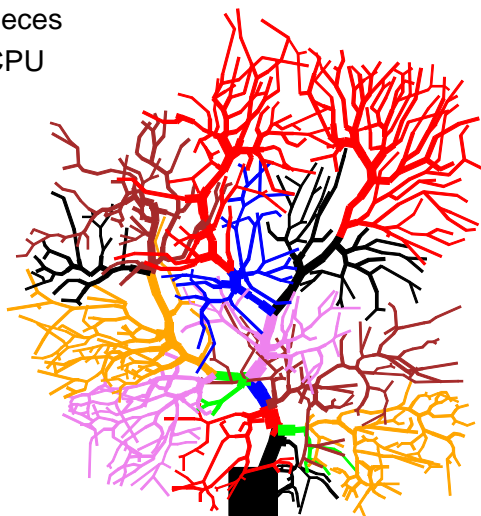


#CPU	Runtime (s)		
1	54.8		
3	22.2	19.5 expected	18.3 ideal

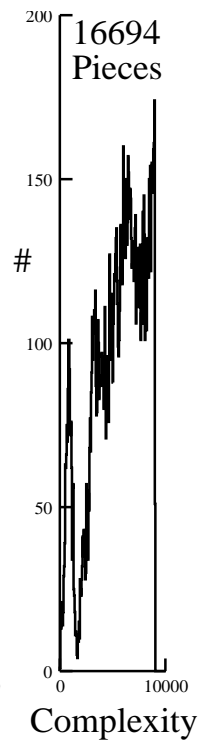
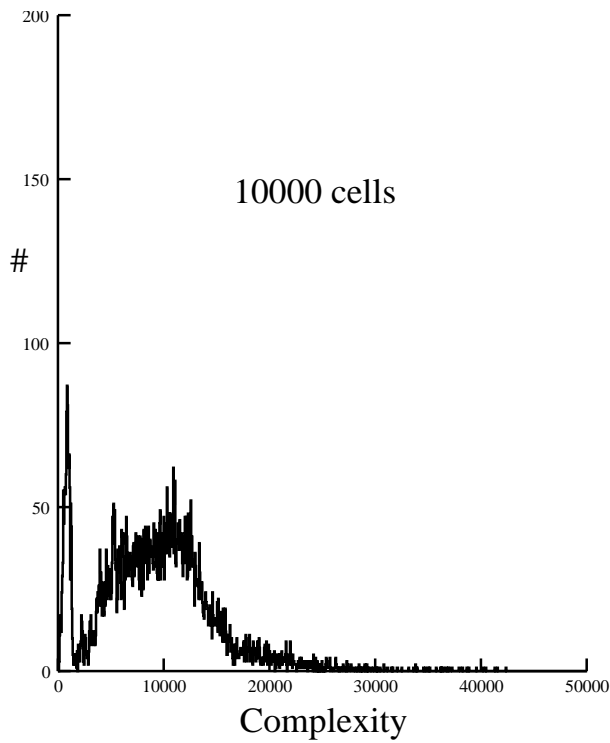
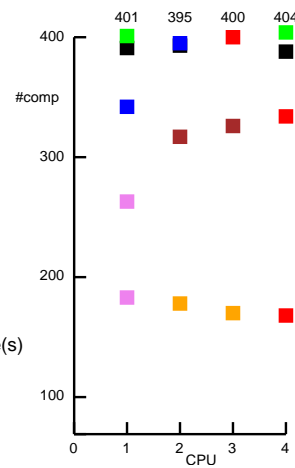
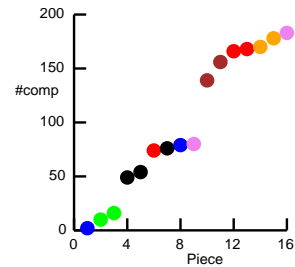


De Schutter & Bower (1994) J. Neurophysiol 71:375  
Ported to NEURON by Jenny Davie, Volker Steuber, and Arnd Roth.

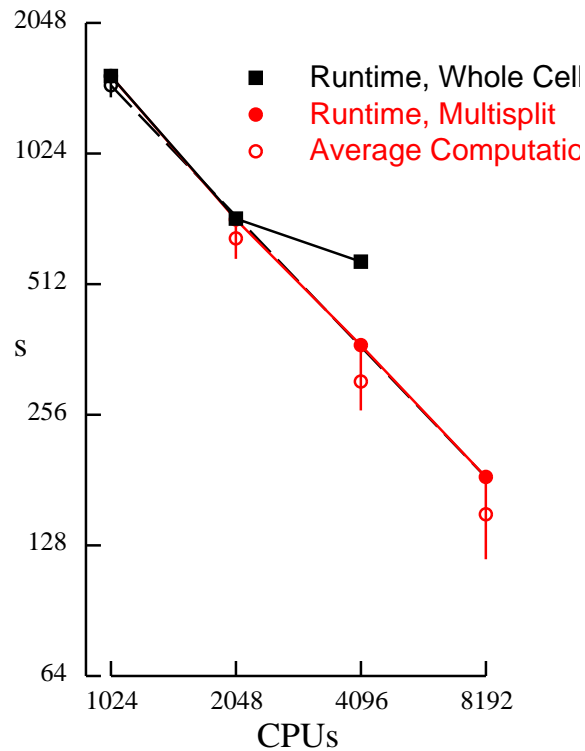
16 Pieces  
4 CPU



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		







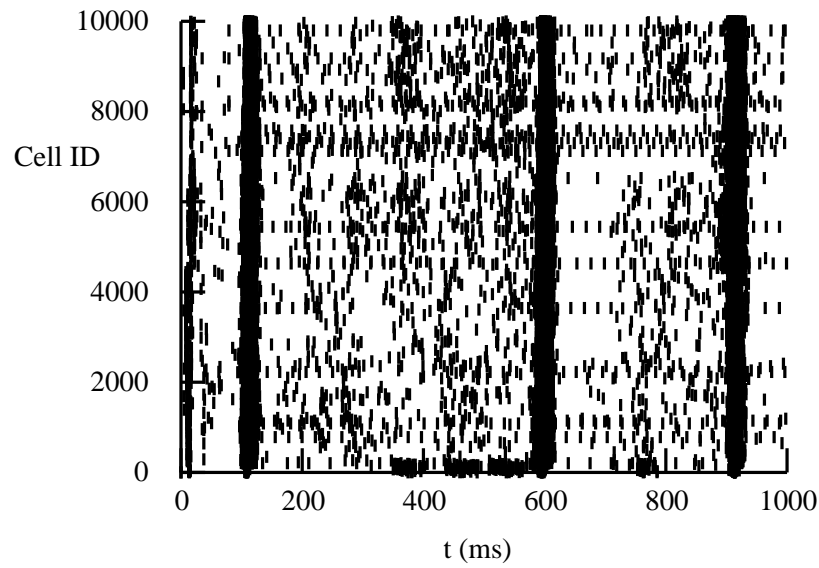
Results must be independent of  
 Number of processors  
 Distribution of cells

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



out.spk (58,858 lines)

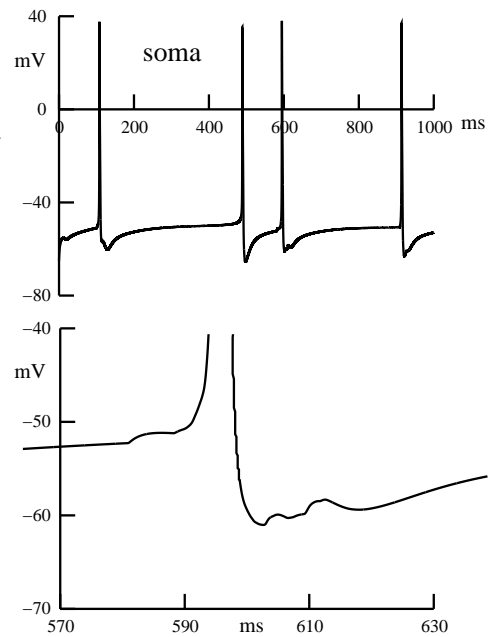
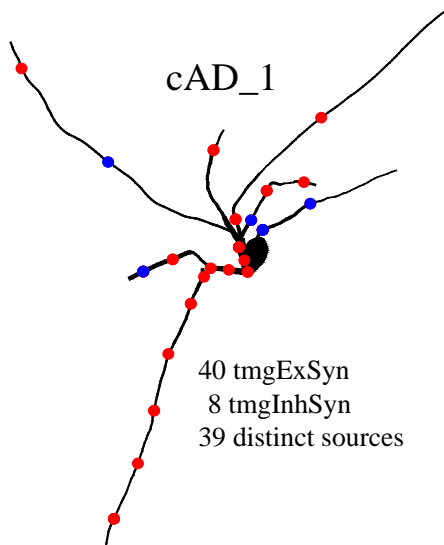
3.975 8050

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 standard run system

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

```
proc init() {  
    finitialize(v_init)  
}
```

```
proc advance() {  
    fadvance()  
}
```

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



## 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 equilb 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 METHODsteadystate
}
e.g.
NEURON {
    USEION k READ ek WRITE ik
}
STATE { c1 c2 o }
INITIAL {
    SOLVE scheme METHODsteadystate
}
BREAKPOINT {
    SOLVE scheme METHODsparse
    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

The NEURON Simulation Environment

Figure 1

## **Anatomy and Empirically-based Models**

### Quality of data

- histology

- staining, amputation, shrinkage

- human error

- diameter

- spines

### Data formats

Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

Figure 2

## **Tests for Quality of Data**

### Quick and dirty "litmus test"

- insert pas

- set Ra and g\_pas low

- inject large depolarizing current at soma

- examine shape plot of v

### Quantitative: look for pt3d data

- with suspicious diameters,

- e.g. too large or too small, == 0, etc.

- A "one liner"

```
forall for i=0, n3d()-1
  if (diam3d(i) == 0)
    print secname(), i, diam3d(i)
```

### More quantitative: look for systematic

- errors, e.g. with a histogram of diameters

Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved





The NEURON Simulation Environment

Figure 1

## NEURON's tools for Electrotonic Analysis

Input and transfer impedances

Voltage transfer ratio

$$V_{\text{downstream}} / V_{\text{upstream}}$$

Electrotonic transformation

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

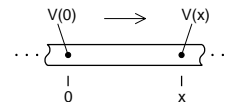
... all as functions of frequency and space

Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

Figure 2

## CLASSICAL CABLE THEORY



Infinite cylinder in the steady state

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

$x \equiv$  physical distance

$\lambda \equiv$  length constant

Classical Definition of Electrotonic Distance:

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

$$\therefore \text{attenuation } A^V(x) = V(0) / V(x) = e^{x/\lambda}$$

Intuitively simple

Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

Figure 3

BUT neurons  $\neq$  infinite cylinders

Attempted fix: reduce dendritic tree to an equivalent cylinder of finite length

Finite cylinder in the steady state

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

$$L_{\text{classical}} \equiv \text{physical length of cylinder} / \lambda$$

$$X \equiv x / \lambda$$

Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

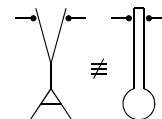
The NEURON Simulation Environment

Figure 4

## The good and the bad news about the equivalent cylinder approximation

### The bad news:

- ◆ Neither intuitive nor simple
- ◆ Destroys the spatial relationships among synaptic inputs



- ◆ Classical electrotonic distance  $X \equiv x / \lambda$  fosters conceptual error because it obscures the direction-dependence of attenuation in finite structures

Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

Figure 5

**The good news:** it's not valid either

Property	Assumption	Truth
Dendritic terminations	electrically equidistant from soma	varies widely
Diameters	cylindrical	irregular
Branch points	3/2 power criterion	no

$$d_p^{3/2} = \sum d_d^{3/2}$$

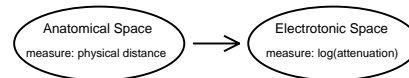
Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

Figure 6

Alternative: a transformation from anatomical to electrotonic space that

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



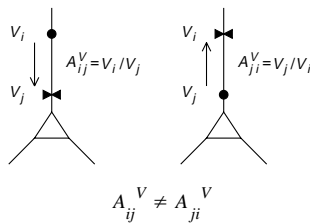
Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

Figure 7

Foundation of this approach:  
two-port analysis of electrotonus

How well do signals propagate?



Signal transfer is direction-dependent

Attenuation identities

$$A_{ij}^V = A_{ji}^I \quad A_{ij}^I = A_{ji}^Q$$

Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

Figure 8

## The Electrotonic Transformation

Functional definition of electrotonic distance

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

- ✓ simple, direct relationship to attenuation

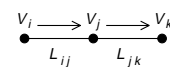
- ✓ direction-dependent:  $L_{ij}^V = \ln(A_{ij}^V)$ ,

$$L_{ji}^V = \ln(A_{ji}^V), \text{ and in general } L_{ij}^V \neq L_{ji}^V$$

Each physical segment  $ij$  of a cell has **two** representations in electrotonic space—one for each direction of propagation!

- ✓ identical to classical electrotonic distance for an infinite cylinder

- ✓ additive over a path with a consistent direction of propagation



$$A_{ik} = \frac{V_i}{V_k} = \frac{V_i}{V_j} \frac{V_j}{V_k} = A_{ij} A_{jk}$$

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

Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

Figure 9

### Using the Electrotonic Transformation

At each frequency of interest:

Step 1: Transform from anatomical to electrotonic space

- Compute attenuations between points of interest
- Map into electrotonic space (log)

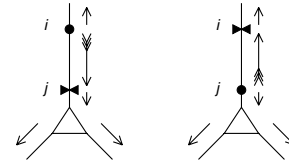
Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

Figure 10

Step 2: Render graphically with respect to a reference point  
(because attenuation is direction-dependent)

- A convenient reference: the soma
- Changing the reference point location alters only the direction of signal flow on the direct path between the old and new locations.



Therefore somatocentric renderings of the transform can be rearranged to generate the renderings for any other reference location.

- The attenuation identities give us the transform identities

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

Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

Figure 11

**Q:** How does somatic peak PSP amplitude depend on synaptic location?

**A?:**  $A_{in}^V$  (voltage attenuation) or  $k_{syn \rightarrow soma}$  (synapse to soma voltage transfer ratio)?

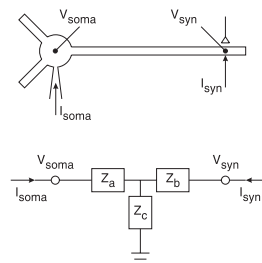
Time-honored and **wrong!**

- ◆ assumes synapses act like voltage sources.
- ◆ real synapses act more like current sources (Jaffe & Carnevale 1999).

Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

Figure 12



Modified from Fig. 1 in Jaffe & Carnevale 1999.

If synapse  $\approx$  voltage source, then

- $V_{syn}(t) \approx$  independent of synaptic location

and

- Synapse to soma voltage transfer ratio

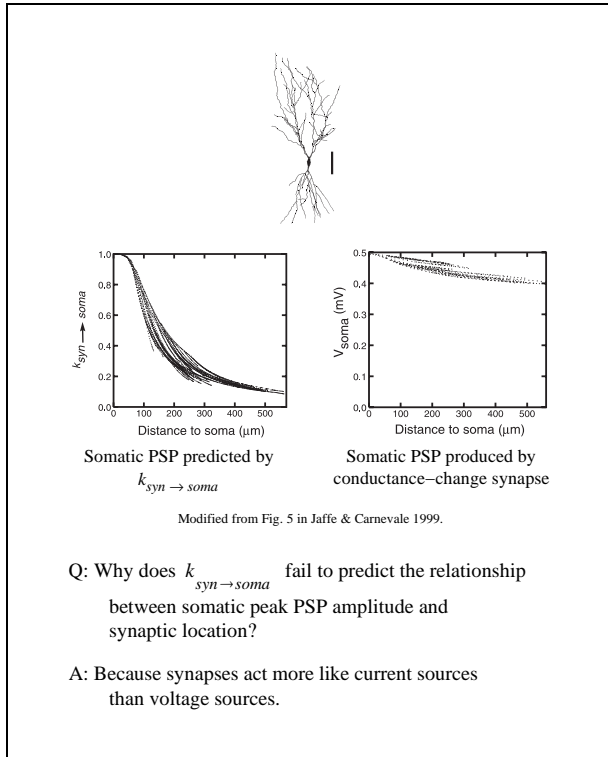
$$k_{syn \rightarrow soma} = 1/A_{in}^V = Z_c / (Z_b + Z_c)$$

predicts the variation of somatic PSP amplitude with synaptic location

Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

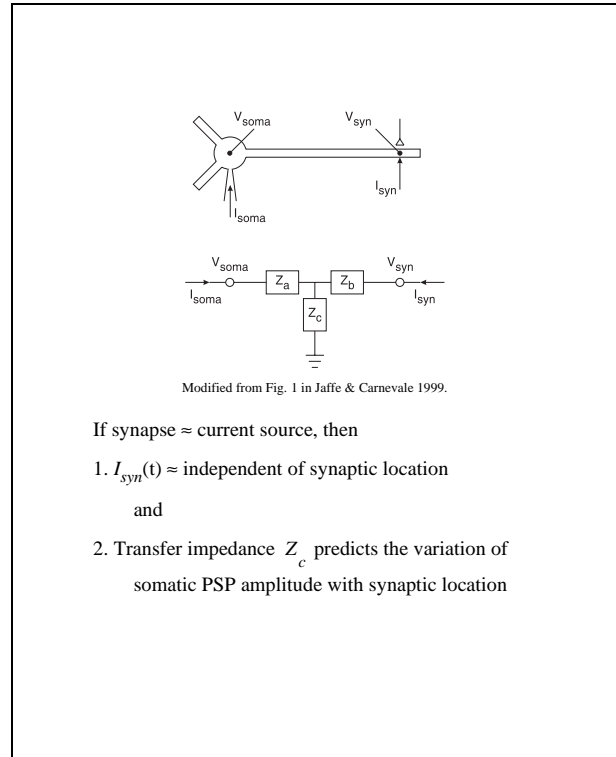
Figure 13



Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

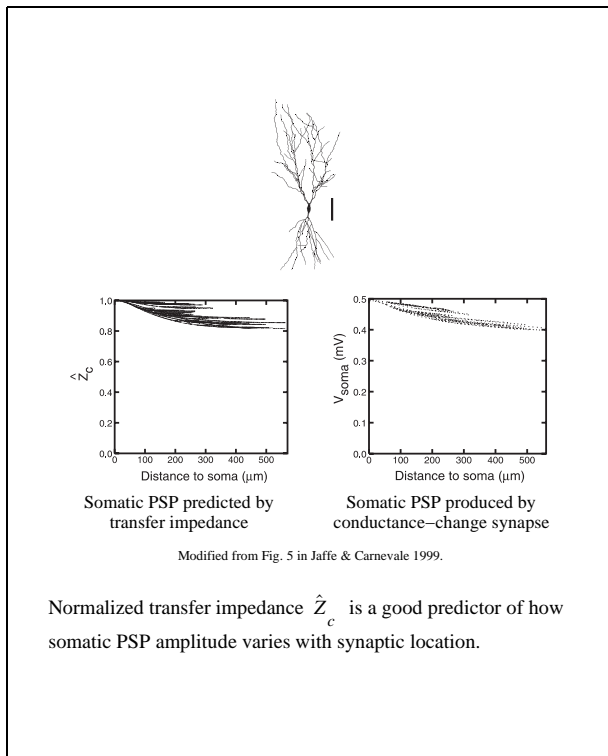
Figure 14



Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

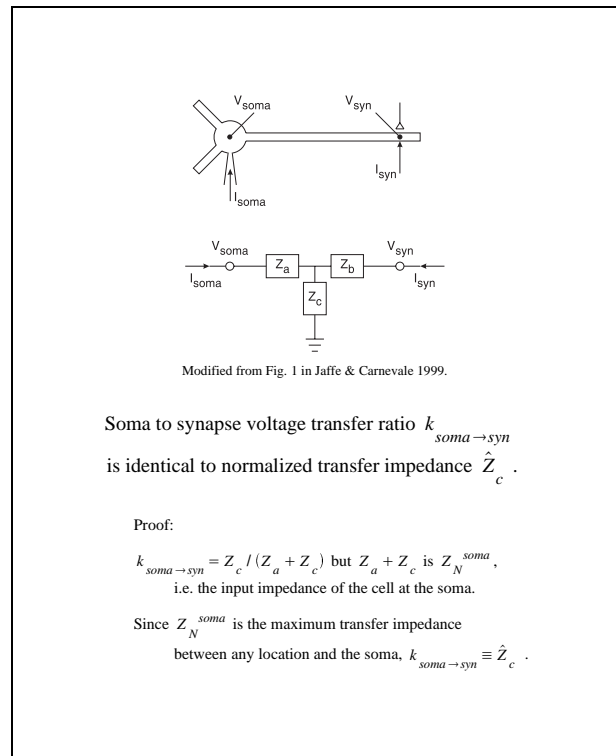
Figure 15



Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved

The NEURON Simulation Environment

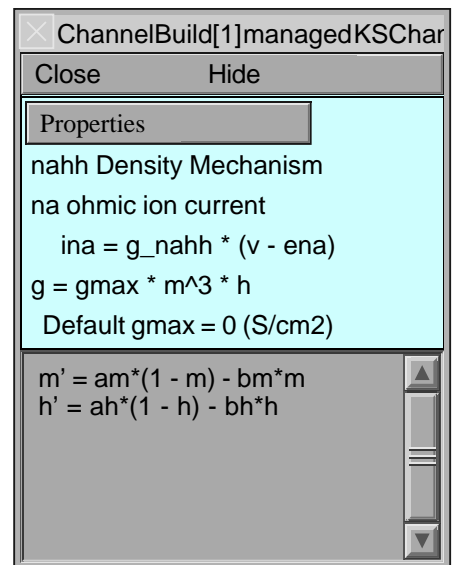
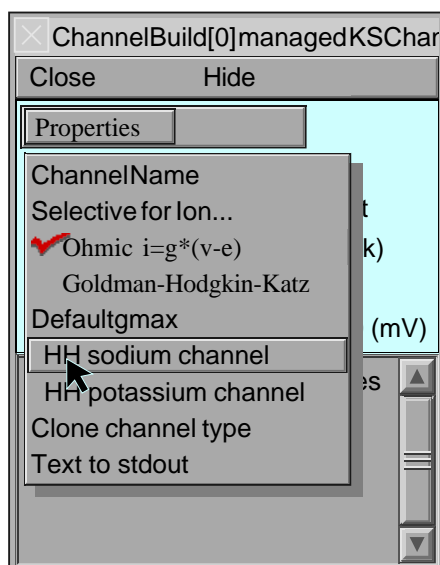
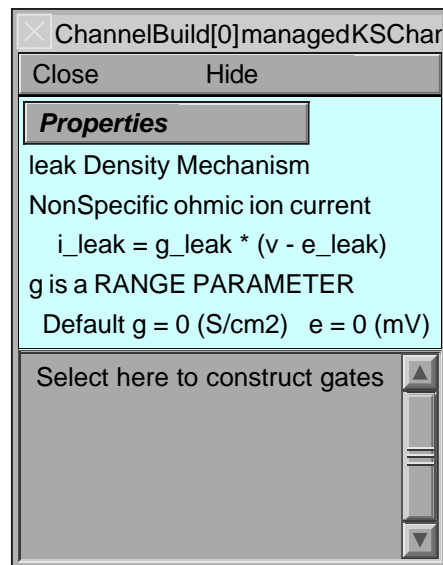
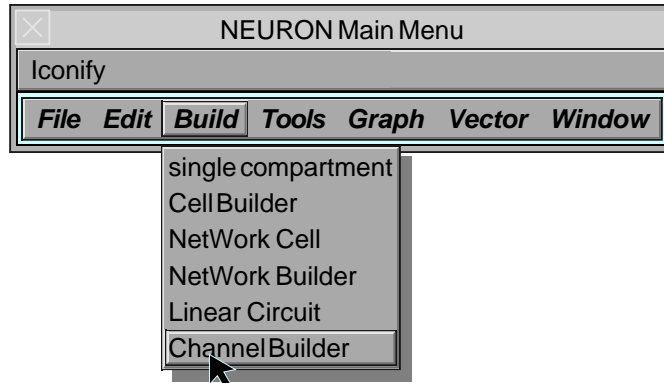
Figure 16



Copyright © 1998–2001 N.T. Carnevale and M.L. Hines, all rights reserved







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 <-> 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

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

infm  
taum

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

nahh Density Mechanism

na ohmic ion current

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

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

Default gmax = 0 (S/cm2)

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

$$am = 1*x/(1 - \exp(-x)) \text{ where } x = 0.1*(v + 40) \quad (\text{KSTrans[0]})$$

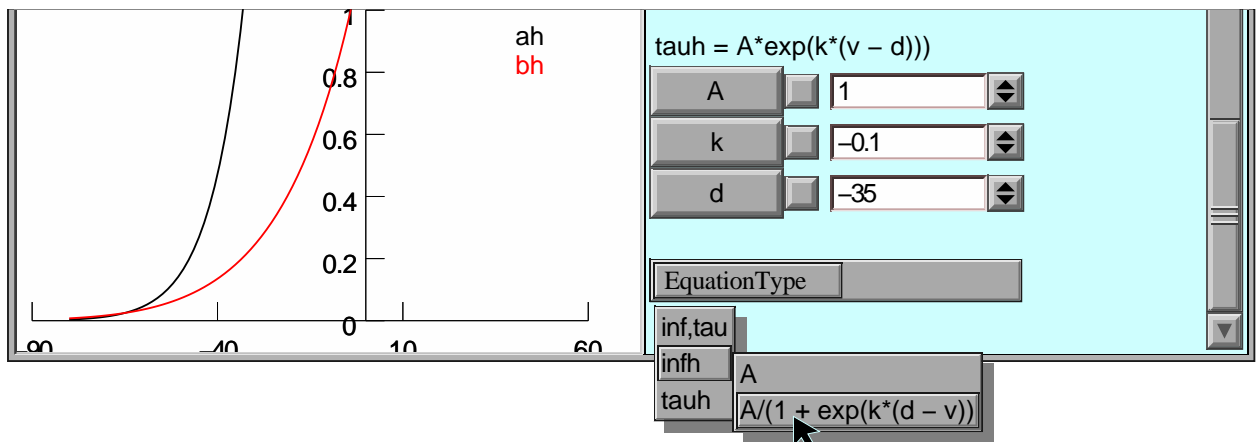
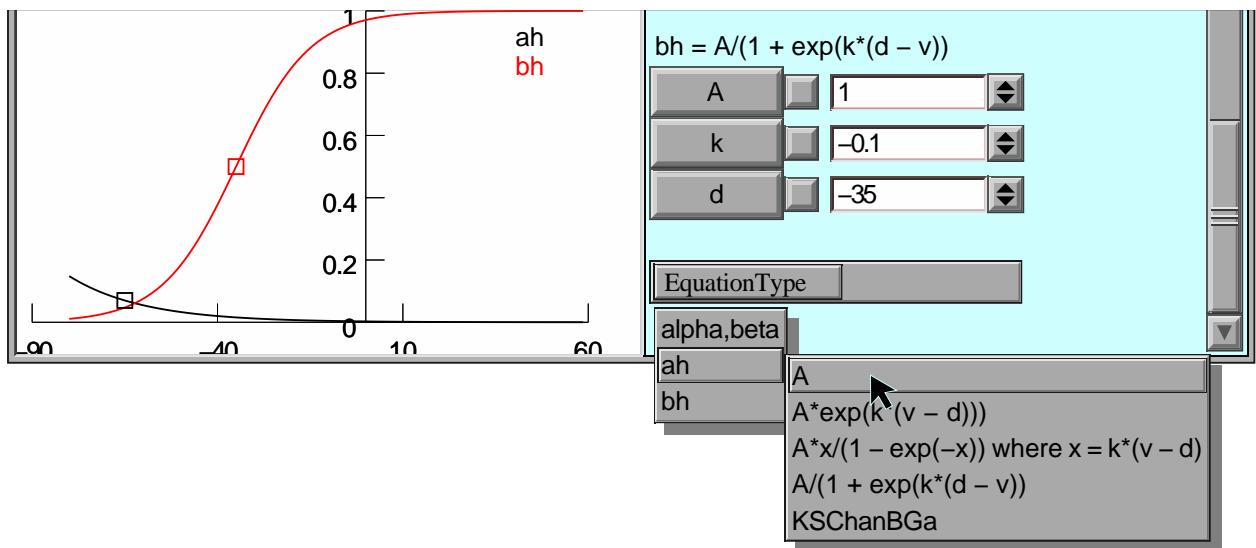
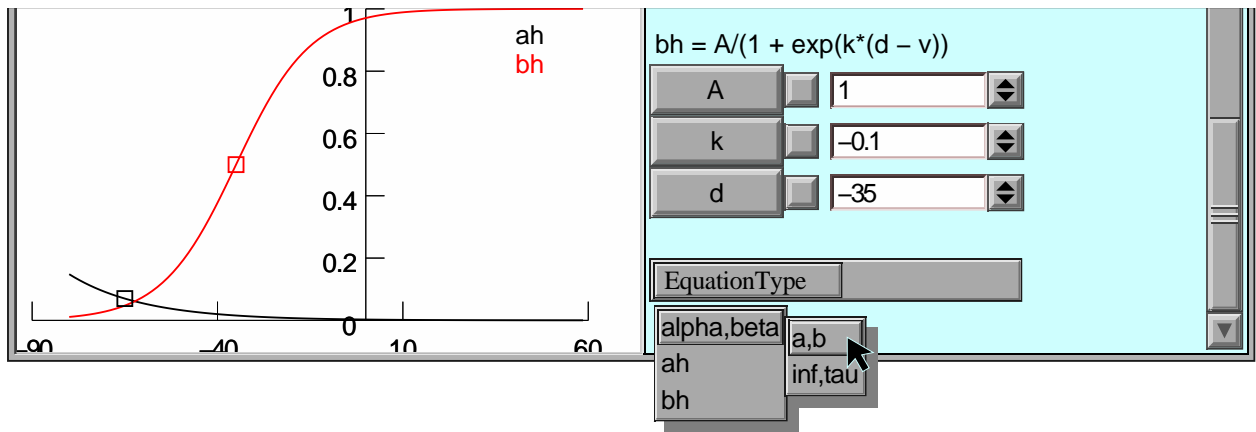
$$bm = 4*\exp(-0.05556*(v + 65))) \quad (\text{Vector[7]})$$

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

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

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





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 simulation environment. The top window, titled "ChannelBuild[0]managedKSChan[0]", shows the "Properties" tab with 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  $g_{\text{max}} = 0 \text{ (S/cm}^2\text{)}$   $e = 0 \text{ (mV)}$
- 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 contains a diagram of state transitions:

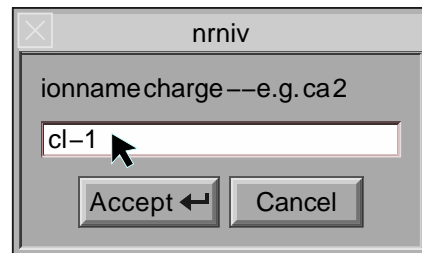
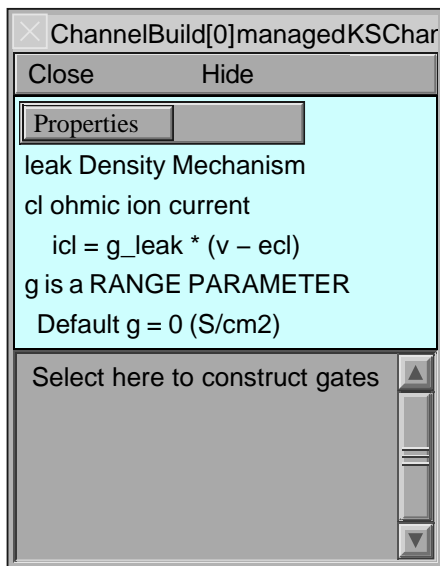
```

O
C
C <-->[v] C2 <-->[v] O
C3 <-->[v] O2
O3

```

A dialog box titled "nrniv" is open, titled "Change state name", with a text field containing "O3" and "Accept" and "Cancel" buttons. The bottom of the window has "Adjust" and "Run" buttons, and a status bar showing "no KSTrans selected".

The screenshot shows the "ChannelBuild[0]managedKSChar" window with the "Properties" tab. A list of channel types is displayed, with "Ohmic  $i = g * (v - e)$ " selected. Other options include "Goldman-Hodgkin-Katz", "HH sodium channel", "HH potassium channel", "Clone channel type", and "Text to stdout". A "Create new type" button is visible on the right.

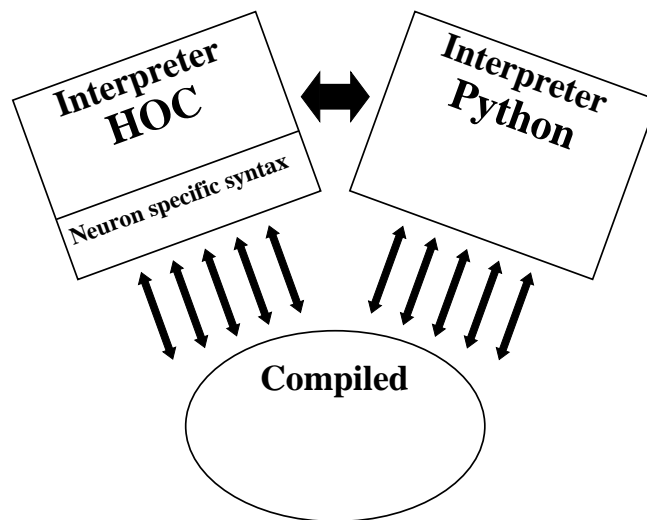


# Python + NEURON

All legacy models must work.

Superior representation of  
underlying concepts.

No extra installation difficulty.



## Installation

```
>>> import neuron
```

Linux	i686		
	x86_64		2.3
Mac	OS X 10.4		2.4
	10.5	Python	2.5
			2.6
MSWin	Cygwin		3.0
	MinGW		

Launch	NEURON	
	Python	NumPy

```
$ nrniv -python
NEURON -- VERSION 7.1 ...

>>> from neuron import h
>>> print h
TopLevelHocInterpreter
>>> h('''strdef s
... s = "hello"
... func square() { return $1*$1
... ''')
1
>>> print h.s, h.square(4)
hello 16.0

>>> v = h.Vector(4).indgen().add(10)
>>> print v, len(v), v.size(), v.x[2], v[
Vector[1] 4 4.0 12.0 12.0
>>> v.printf()
10      11      12      13
4.0
>>> for x in v: print x
...
10.0
11.0
12.0
13.0
>>>
```



```

>>> import numpy
>>> na = numpy.arange(0, 10, 0.00001) # 0.01
>>> v = h.Vector(na) # 0.01
>>> v.size()
1000000.0
>>> nb = numpy.array(v) # 0.01
>>> nb[999999]
9.999990000000000004
>>> b = list(v) # 0.07
>>> for i in xrange(0, len(nb)):
...     v.x[i] = na[i]
... # 3.74

>>> def callback(a = 1, b = 2):
...     print "callback: a=%d b=%d" % (a, b)
...
>>> fih = h.FInitializeHandler(callback)
>>> h.finititalize()
callback: a=1 b=2
1.0
>>> fih = h.FInitializeHandler((callback,
... (4, 5)))
>>> h.finititalize()
callback: a=4 b=5
1.0
>>>

```

```

# assume hh soma model

vvec = h.Vector()
vvec.record(soma(.5)._ref_v, sec=soma)

tvec = h.Vector()
tvec.record(h._ref_t, sec=soma)

h.run()

g = h.Graph()
g.size(0, 5, -80, 40)
vvec.line(g, tvec)

>>> from neuron import h
>>> soma = h.Section()
>>> axon = h.Section()
>>> axon.connect(soma, 1)
>>> axon.nseg = 5
>>> h.topology()

|-|          PySec_2b371cd171d0(0-1)
  |--|       PySec_2b371cd17190(0-1)

1.0

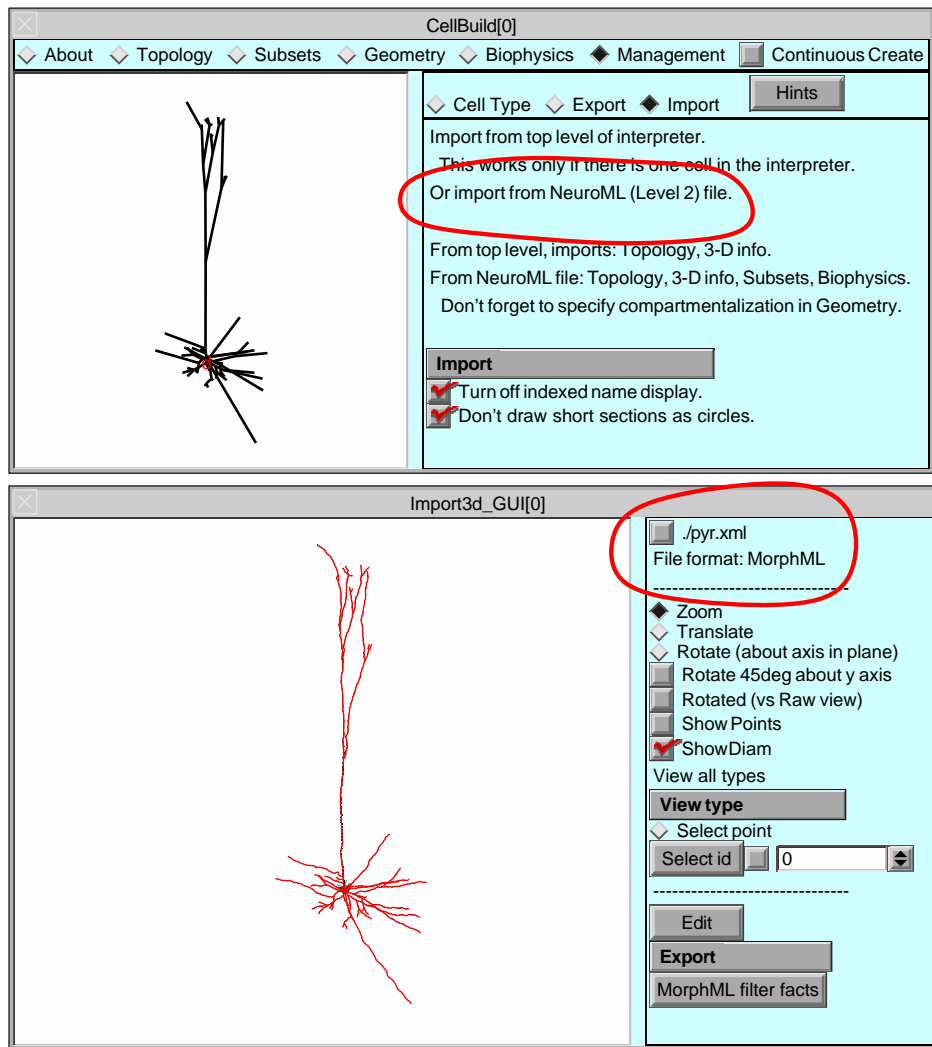
>>> axon.L = 1000
>>> axon.diam = 1

>>> for sec in h.allsec()
...     sec.cm = 1
...     sec.Ra = 100
...     sec.insert('hh')
...

```

```
>>> axon.gnabar_hh = .1
>>> axon(.5).hh.gnabar = .09
>>> for seg in axon:
...     print seg.x, seg.hh.gnabar
...
0.1 0.1
0.3 0.1
0.5 0.09
0.7 0.1
0.9 0.1
```

```
>>> stim = h.IClamp(.5, sec=soma)
>>> stim.delay = .5
>>> stim.dur = .1
>>> stim.amp = .4
```



### **nrn/lib/hoc/import3d/read\_morphml**

```

begintemplate Import3d_MorphML
public input, parsed
...
proc init() {
  nrnpython("import rdxml")
  p = new PythonObject()
}
proc input() {
  p.rdxml.rdxml($s1, this)
}
proc parsed() { ...

```

**nrn/lib/python/rdxml.py**

```

import xml
def rdxml(fname, ho) :
    xml.sax.parse(fname, MyContentHandler(ho))

class Point Cable CableGroup BioParm BioMech
class MyContentHandler(xml.sax.ContentHandler):
    def __init__(self, ho):
        self.i3d = ho
    ...
    def endDocument(self):
        self.i3d.parsed(self)

```

**nrn/lib/hoc/import3d/read\_morphml**

```

proc parsed() { ...
    for i=0, po.len($o1.cables) - 1 {
        cab = $o1.cables._[i]
        sec = new Import3d_Section(cab.first, cab.pcnt)
        sections.append(sec)
        if (cab.parent_cable_id >= 0) {
            ip = $o1.cableid2index[cab.parent_cable_id]
            sec.parentsec = sections.object(ip)
        }
    }
}

```

