Schedule of Presentations

NTC	Ted Carnevale
MLH	Michael Hines
RM	Robert McDougal

Morning session

Time	Speaker	Title	Page
9:00 AM	MLH	Welcome	3
9:05	NTC	NEURON: a brief tour	5
	NTC	The basics	9
	NTC	Construction and use of models	19
	NTC	Using the CellBuilder to make a stylized model	20
	NTC	Creating and using an interface for running simulations	32
10:15	NTC	The Linear Circuit Builder	43
10:30	Coffee Break		
10:45	MLH	Using NMODL to add new biophysical mechanisms	51
11:15	MLH	Numerical methods: accuracy, stability, speed	59
11:30 AM	NTC	Networks: spike-triggered synaptic transmission, events, and artificial spiking cells	65
12:15 PM	Lunch		

Afternoon session

1:15 PM	MLH	Numerical methods: adaptive integration and events	75
1:30	MLH	Parallelizing network simulations	79
2:00	RM	NEURON with Python	95

3:15	Coffee Break		
3:30	RM	ModelDB and other resources for computational neuroscience	115
4:00	RM	Reactive Diffusion	129
4:45	MLH	Future directions	
5:00		End of afternoon session	

Receipt and Survey **last two pages**

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

Satellite Symposium, Society for Neuroscience

Using NEURON to Model Cells and Networks

Chicago, IL

Friday, October 16, 2015

Ted Carnevale
Michael Hines
Robert McDougal

Supported by NINDS

NEURON http://neuron.yale.edu/

NEURON: a brief tour

A tool for empirically-based models of neurons and neural circuits

Open source project directed by Michael Hines

Active development and user support

- Documentation, tutorials, and forum at http://www.neuron.yale.edu/
- Courses
- SFN meetings
- summer course at UCSD and elsewhere
- other courses

The NEURON user community

Used by experimentalists, theoreticians, and educators for neuroscience research and teaching

As of September 2015

- more than 1600 publications
- more than 1700 subscribers to mailing list and forum http://www.neuron.yale.edu/phpBB/
- source code for >500 published models at ModelDB http://modeldb.yale.edu/

Specifying and using models with NEURON

- Model specifications written in hoc and/or Python
  - and/or
- created with GUI tools (work via hoc)
  - CellBuilder, Channel Builder,
  - Network Builder, Linear Circuit Builder
- Add new functionality with NMODL (compiled)
  - ion channels, synaptic mechanisms
  - signal sources
  - accumulation, diffusion, transport, reactions
    - described by ODEs, kinetic schemes,
    - algebraic equations
  - events, state machines, artificial spiking cells
- Add reactive diffusion (uses Python)

Not model specification, but necessary

- Instrumentation
  - stimulators, current or voltage clamps
  - plotting and recording variables
- Simulation control
  - default and custom initializations
  - integration methods
    - fixed time step
    - adaptive integration
  - event system useful for implementing
    - "experimental protocols"
- User interface

Other features

- Parallel simulation
  - multithreaded execution
  - embarrassingly parallel problems
  - distributed models
- Optimization tools
- Model analysis
  - Impedance tools
  - ModelView
- Import3D for detailed morphometric data

Where to learn more

- The NEURON Book
- NEURON's home page `neuron.yale.edu`
  - Documentation
    - hints and tutorials
    - FAQ list
    - key papers about NEURON
  - Programmer's Reference
  - Courses
- The NEURON Forum `neuron.yale.edu/phpBB`
  - Getting started
  - Hot tips

The What and the Why of Neural Modeling

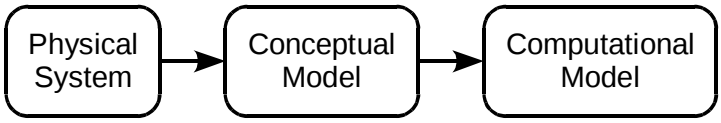
The moment-to-moment processing of information in the nervous system involves the propagation and interaction of electrical and chemical signals that are distributed in space and time.

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

Topics

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

From Physical System to Computational Model



Conceptual model

a simplified representation of the physical system

Computational model

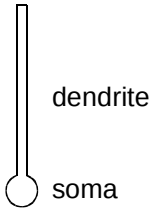
an accurate representation of the conceptual model

From Physical System to Computational Model

Physical system

Ca1 pyramidal cell

Conceptual model



ball and stick

Computational model

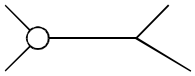
```
create soma, dendrite
connect dendrite(0), soma(1)
```

hoc code

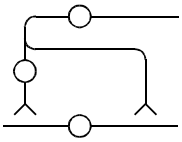
Hierarchies of Complexity

Structure

Single compartment 

Stylized 

Anatomically detailed 

Network 

Hierarchies of Complexity

Mechanism

Passive and Active currents
- HH-style
- kinetic scheme

Synaptic transmission
- continuous
- spike-triggered

Gap junctions

Extracellular fields, Linear circuits

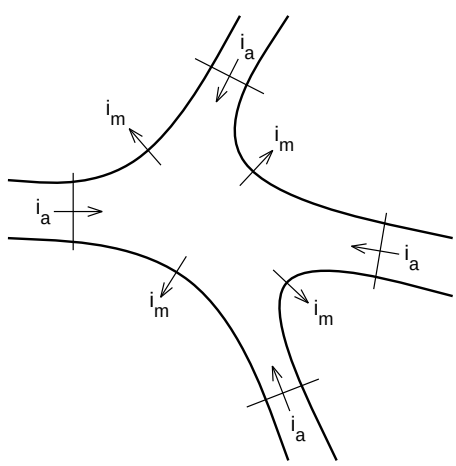
Diffusion, buffers, transport & exchange

Artificial spiking cells ("integrate & fire")

Fundamental Concepts in NEURON

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

Conservation of Charge



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

The Model Equations

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

v_j membrane potential in compartment j

i_{ion_j} net transmembrane ionic current in compartment j

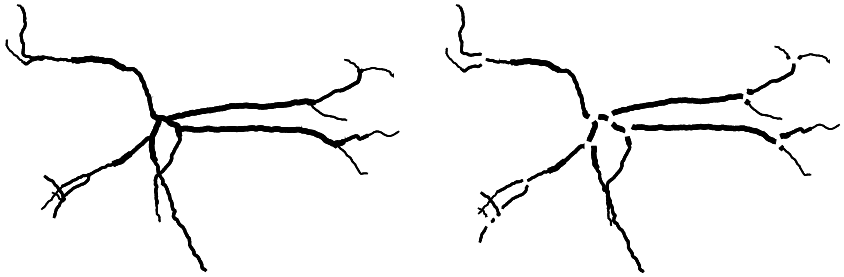
c_j membrane capacitance of compartment j

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

Separating Anatomy and Biophysics from Purely Numerical Issues

section

a continuous length of unbranched cable



Anatomical data from A.I. Gulyás

Mathematical description of a section

What we want:

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

What a new section gives us:

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

i.e. membrane capacitance and axial resistance,
but no ionic current.

How can we put ion channels in the membrane?

Adding mechanisms to sections

Density mechanisms
- distributed channels
- ion accumulation

Point processes
- electrodes, synapses

Described by
- differential equations
- kinetic schemes
- algebraic equations

Constructed with
- NMODL
- Channel Builder

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

soma {
  L = 50  // [um] length
  diam = 50  // [um] diameter
  insert hh  // Hodgkin-Huxley mechanism
  nseg = 1
}

dend {
  L = 200
  diam = 2
  insert pas // passive channels
  nseg = 3
}
```

Range Variables

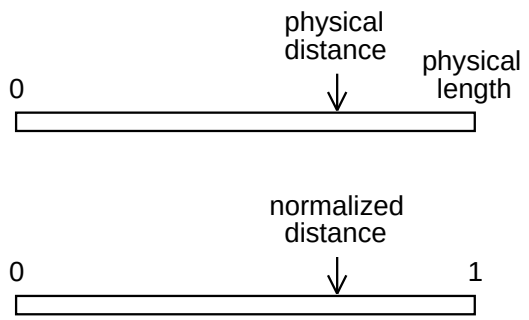
Name	Meaning	Units
diam	diameter	[µm]
cm	specific membrane capacitance	[µf/cm^2]
g_pas	specific conductance of the pas mechanism	[siemens/cm^2]
v	membrane potential	[mV]

range

normalized position along the length of a section

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

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



Syntax:

`sectionname.rangevar(range)`

returns or sets value of rangevar
at location that corresponds to range

Examples:

`dend.v(0.5)`

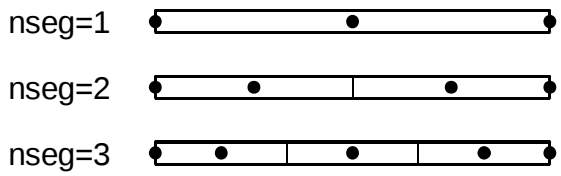
returns v at middle of `dend`
Shortcut: `dend.v`

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

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

nseg

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



Example: `axon nseg = 3`

To test spatial resolution

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

and repeat the simulation

Category	**Variable**	**Units**
Time	`t`	[ms]
Distance	`diam, L`	[µm]
Voltage	`v`	[mV]
Current		
specific	`i`	[mA/cm^2] (density)
absolute		[nA] (point process)
Capacitance		
specific	`cm`	[µf/cm^2]
absolute		[nf] (point process)
Conductance		
specific	`g`	[S/cm^2] (density)
absolute		[µS] (point process)
Cytoplasmic resistivity	`Ra`	[Ω cm]
Resistance	`SEClamp.rs`	[10^6 Ω]
Concentration	`cai`, `nao`, etc.	[mM]

Model specification summary

Topology: create and connect sections

Geometry: stylized (L & diam) or 3D (x,y,z,diam)

Biophysics: insert density mechanisms,
attach "biological" point processes (synapses)

Network models
- Create cells
- Connect cells

Construction and Use of Models

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

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

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

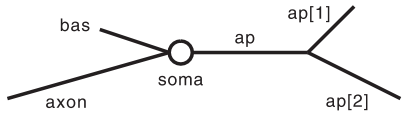
Step 0: Conceptualize the task

Shape
- stick figure / detailed

Channel distribution
- uniform / nonuniform
- whole cell / region / individual neurite

Creation
- single cell / use in a network

Step 1: using the CellBuilder to make a stylized model



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

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

§ - e_pas = - 65 mV

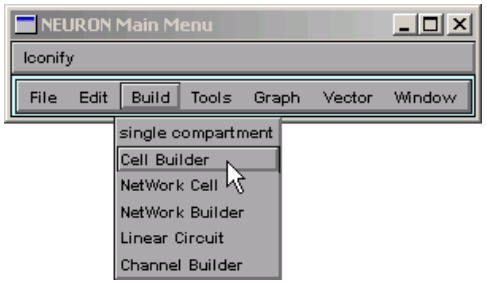
Throughout the cell Ra = 160 Ω cm, cm = 1 µf / cm^2

Launch NEURON with its library of graphical tools

UNIX/Linux `nrngui`

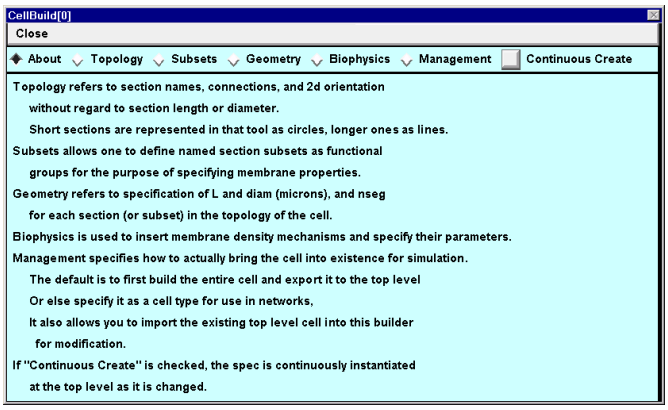
MSWin or OS X

Bring up a CellBuilder



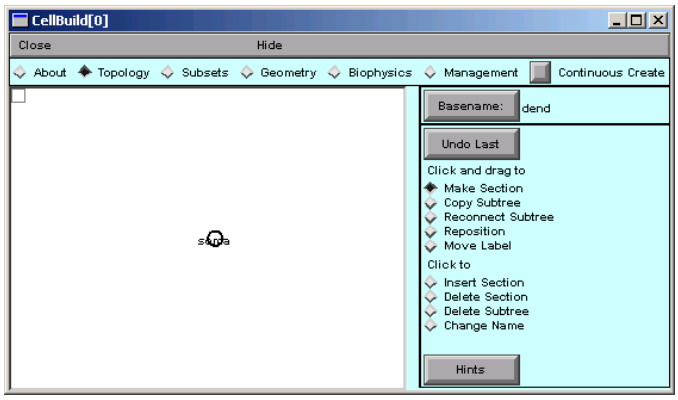
NEURON Main Menu / Build / Cell Builder

The CellBuilder



Use buttons from left to right.

Topology



CB starts with a "soma" section.
We want to create new sections.

Specifying the "Basename"

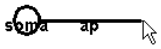
Making a new section

Place cursor near end of existing section

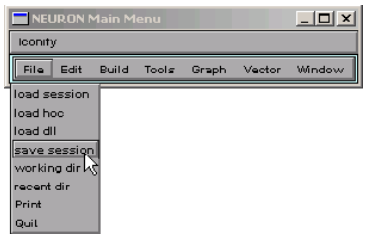
Click to start new section

Drag to desired length

Release mouse button

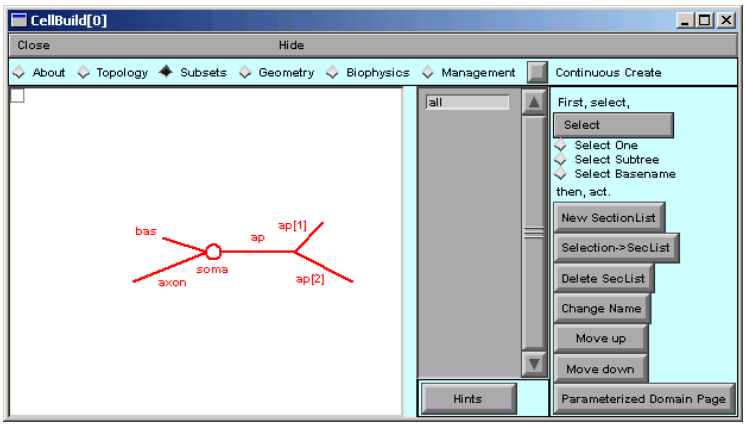


Save your work as you make progress!



NEURON Main Menu / File / save session

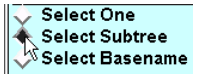
Subsets



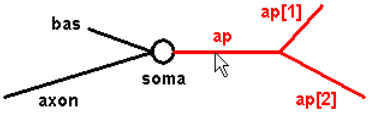
Group sections that have shared properties.
We want to make an "apicals" subset.

Making a new subset

Click "Select Subtree"

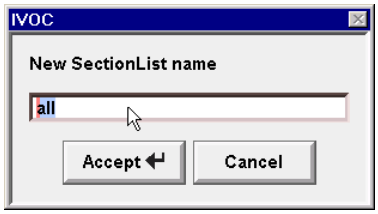


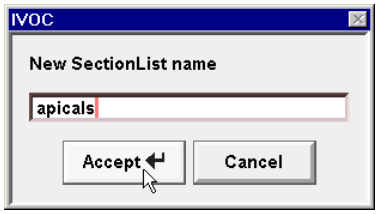
Click root of apical tree . . .



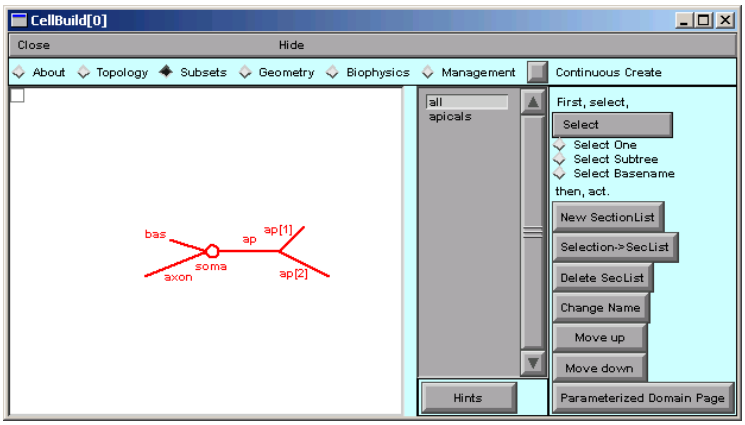
. . . then "New SectionList"

Making a new subset *continued*





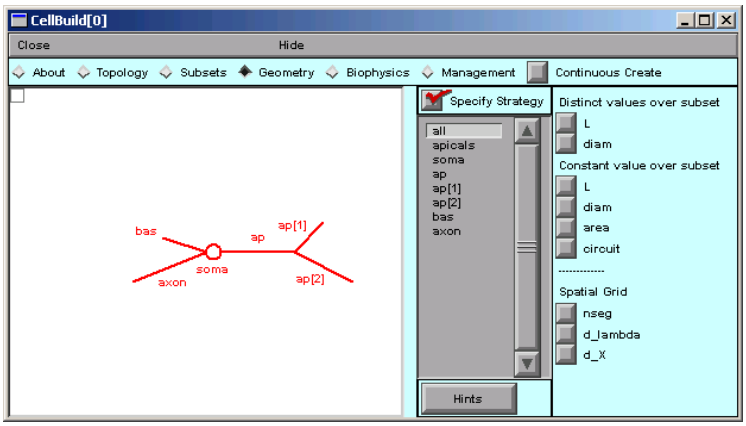
Subsets finished



Note "apicals".

Time to save a new session file.

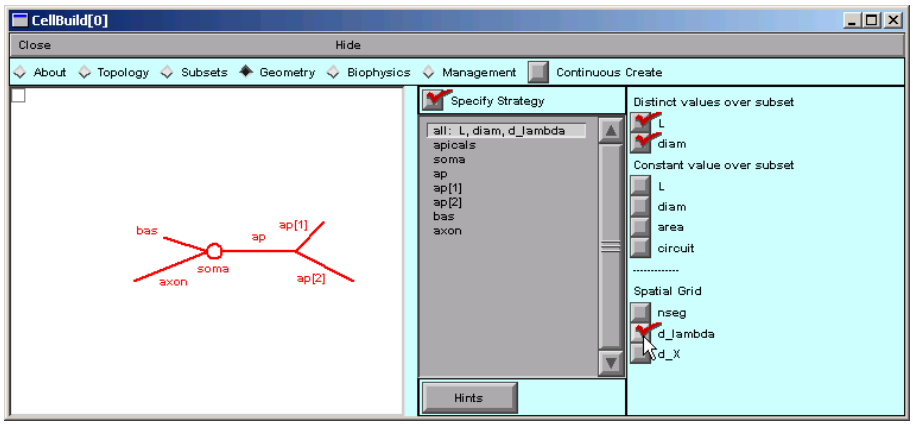
Geometry



"Specify Strategy" is ON.

A good strategy is a concise strategy.

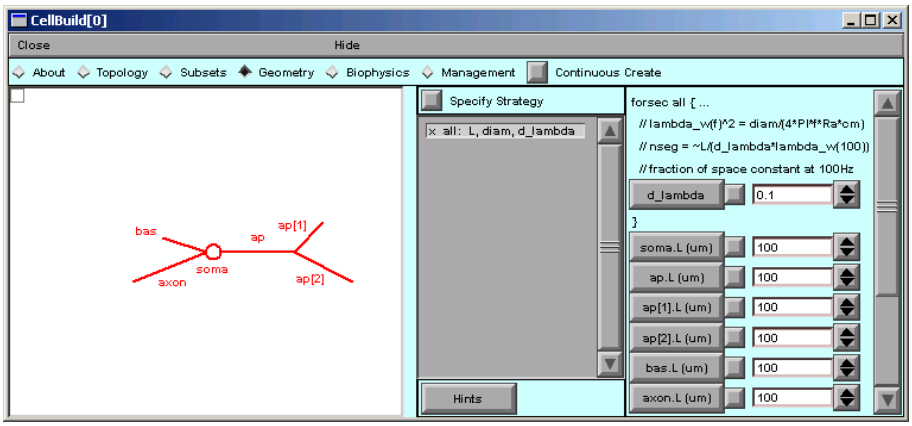
Geometry strategy



Each section has a different L and diam.

Compartmentalize according to $\lambda_{100\ Hz}$ (d_lambda rule).

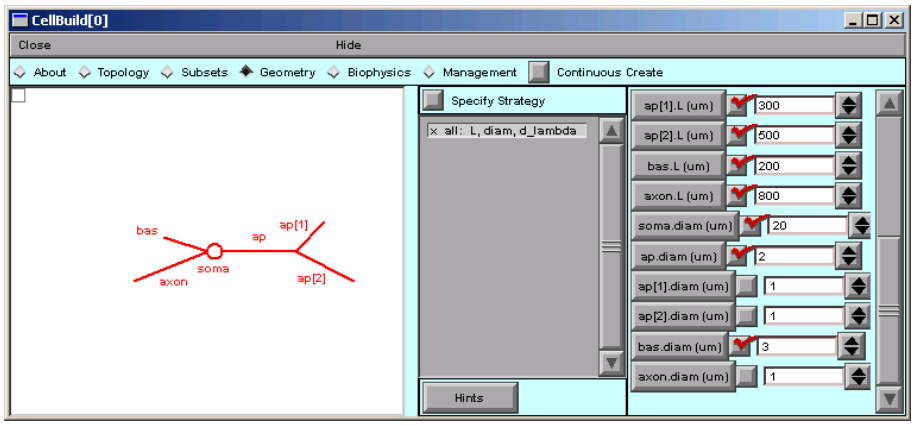
Implementing geometry strategy



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

d_lambda = 0.1 at 100 Hz usually gives good spatial accuracy.

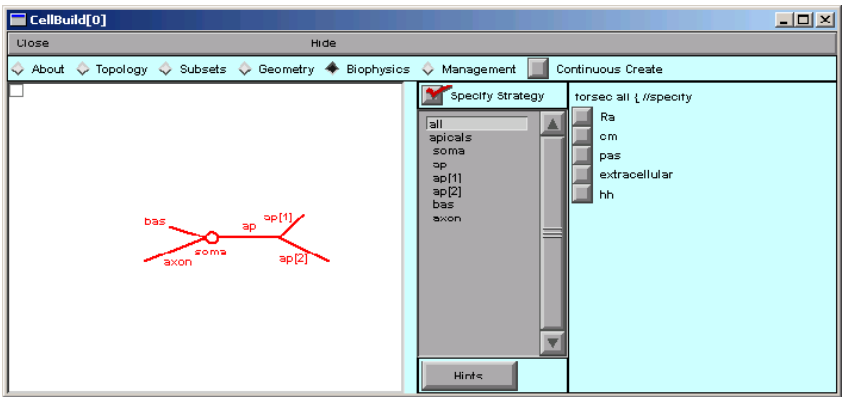
Implementing geometry *continued*



Set L and diam for all sections.

Time to save to a session file!

Biophysics

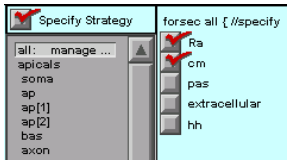


"Specify Strategy" is ON.

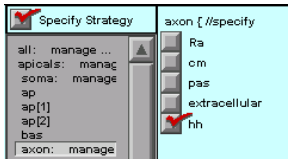
Base the plan on shared properties.

Biophysics strategy

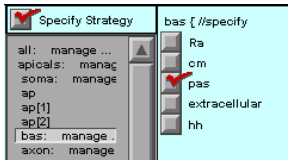
Ra and cm are homogeneous



apicals, soma and axon have hh

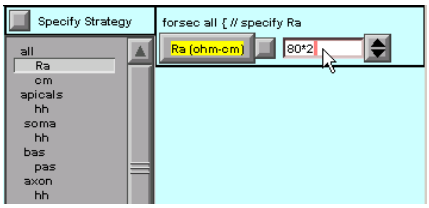


bas has pas

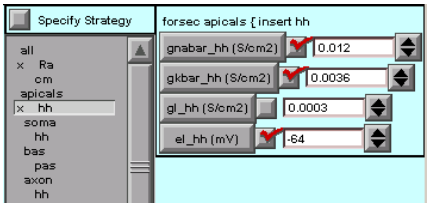


Implementing biophysics strategy

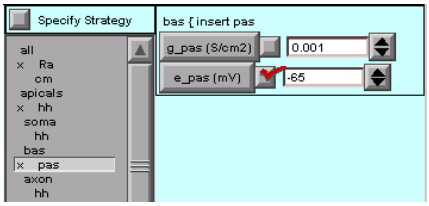
Double Ra



Fix apicals hh params



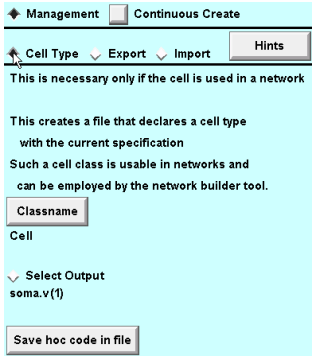
Shift e_pas in bas



Save another session file!!

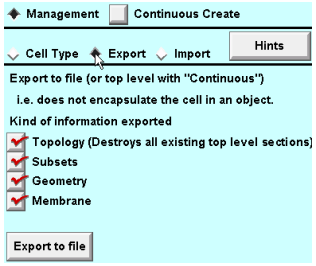
Management

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



Management *continued*

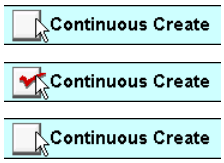
Option 2: save as hoc file



Management *continued*

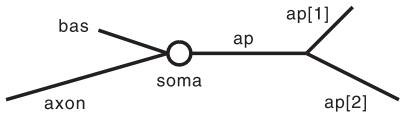
Option 3: export to interpreter

Toggle Continuous Create ON and OFF



or just leave it ON all the time.

Step 2: creating and using an interface for running simulations



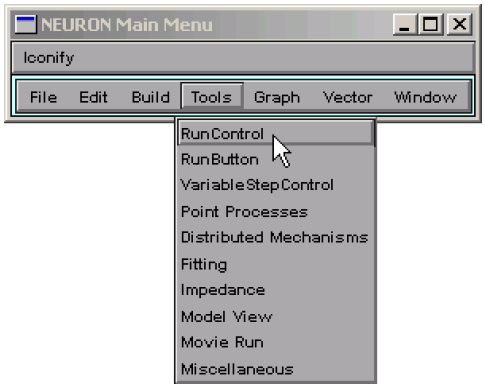
We want to

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

We need

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

Get a "Run" button



NEURON Main Menu / Tools / RunControl

RunControl panel

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

Init & Run does an Init, then starts a simulation

Stop interrupts the simulation

Continue til runs until displayed time

Continue for runs for displayed interval

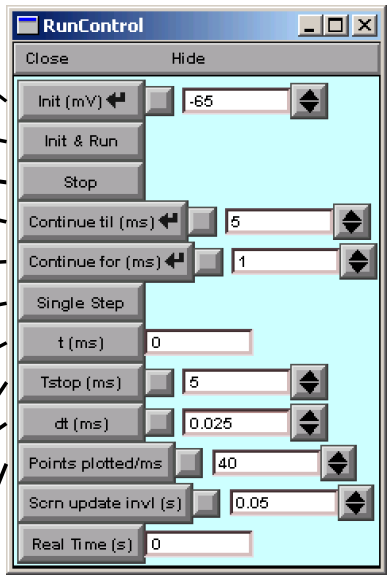
Single step advances by 1/(**Points plotted/ms**)

t numeric field shows model time

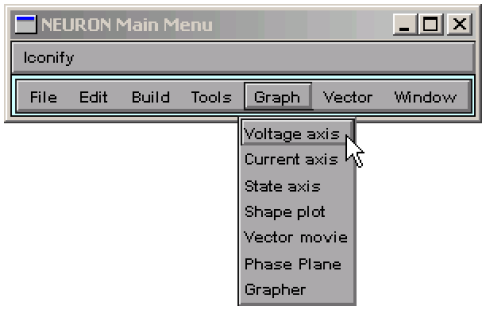
Tstop specifies when simulation ends

dt is integration time step; must be integer fraction of 1/(**Points plotted/ms**)

Points plotted/ms is plotting interval

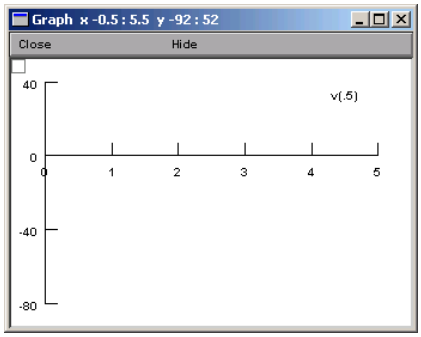


We need to plot Vm(t) at soma



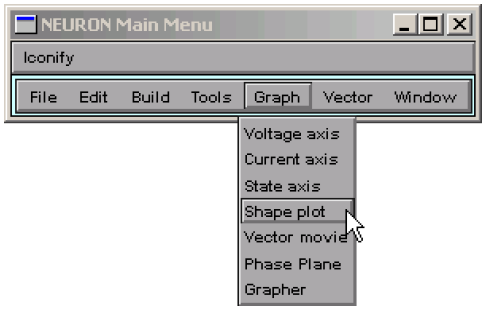
NEURON Main Menu / Graph / Voltage axis

Graph window



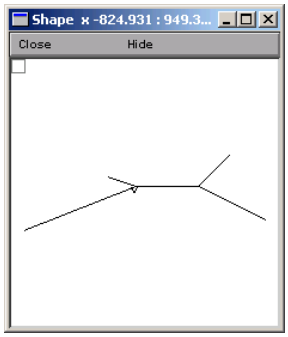
v(.5) is Vm at middle of default section (soma in this example)

We need to plot Vm along a path



NEURON Main Menu / Graph / Shape plot

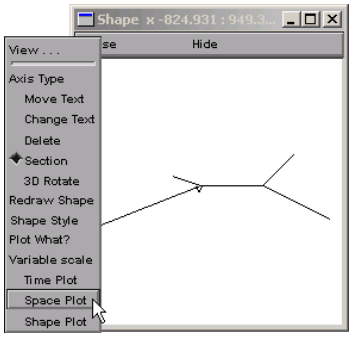
Bringing up a space plot



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

Click on its "menu box" . . .

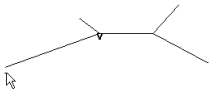
Bringing up a space plot *continued*



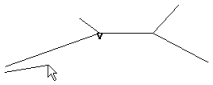
. . . and scroll down to "Space Plot".

Bringing up a space plot *continued*

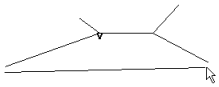
Click just left of the shape



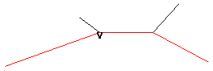
Hold button down while dragging from left . . .



. . . to right . . .

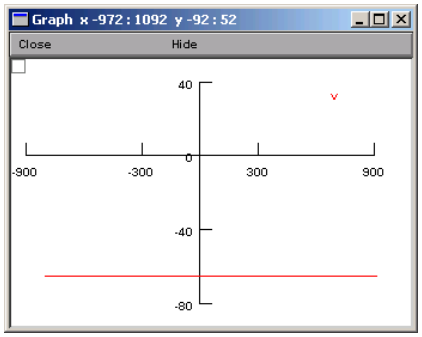


. . . then release button.



This pops up a . . .

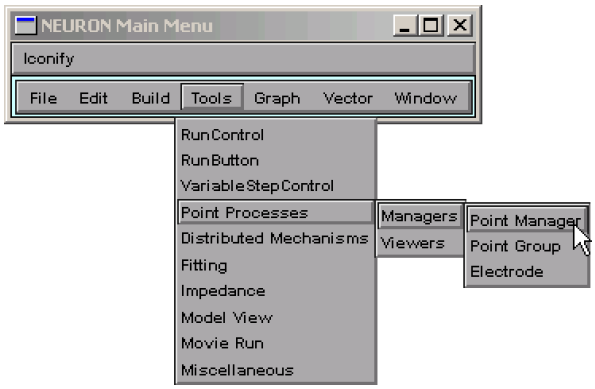
Space plot



A plot of Vm vs. distance along a path.

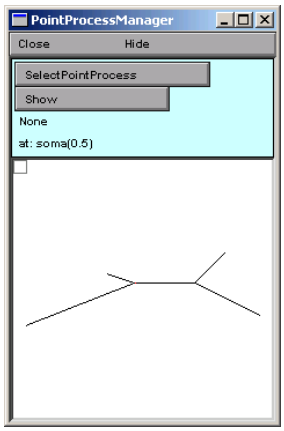
Better save a session file.

We need a stimulator



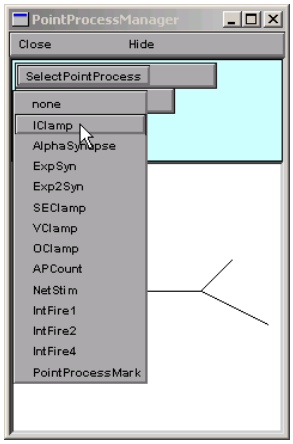
NEURON Main Menu / Tools / Point Processes / Managers / Point Manager

PointProcessManager window



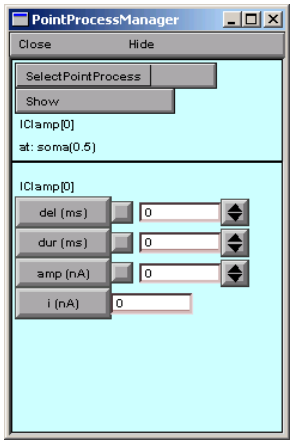
To make this an IClamp . . .

Creating an IClamp



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

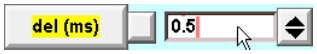
IClamp parameter panel



Next: set parameter values.

Entering values into numeric fields

Direct entry

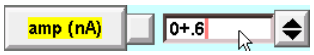


Note yellow highlight on button

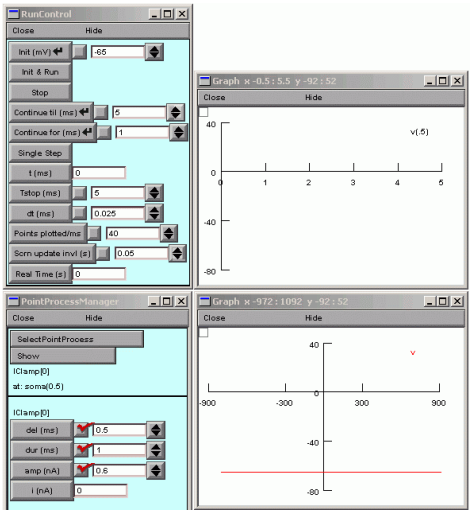
Spinner

Red check means value has been changed from default

Mathematical expression

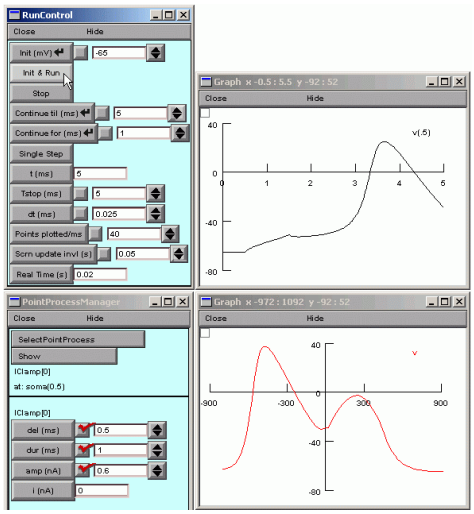


Our user interface

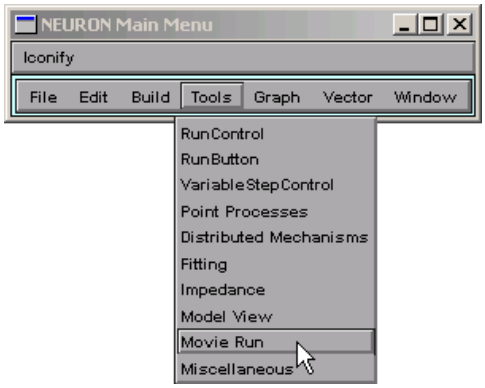


Time to save to a new session file!

It works!



How to get nice space plot "movies"



NEURON Main Menu / Tools / Movie Run

Space plot "movies" *continued*

Movie Run / Init & Run

What if channel density in the apical tree varies systematically with position, e.g. distance from the soma?

See "Specifying parameterized variation of biophysical properties" in the CellBuilder tutorial at http://neuron.yale.edu/neuron/docs

The Linear Circuit Builder

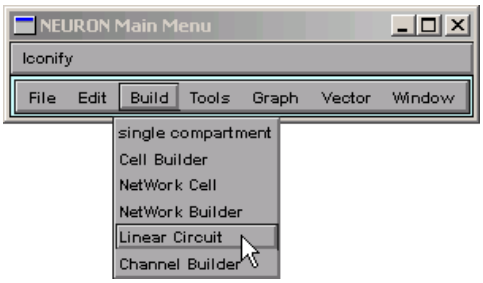
For building models that have linear circuit elements and may also involve neurons

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

Potential applications include

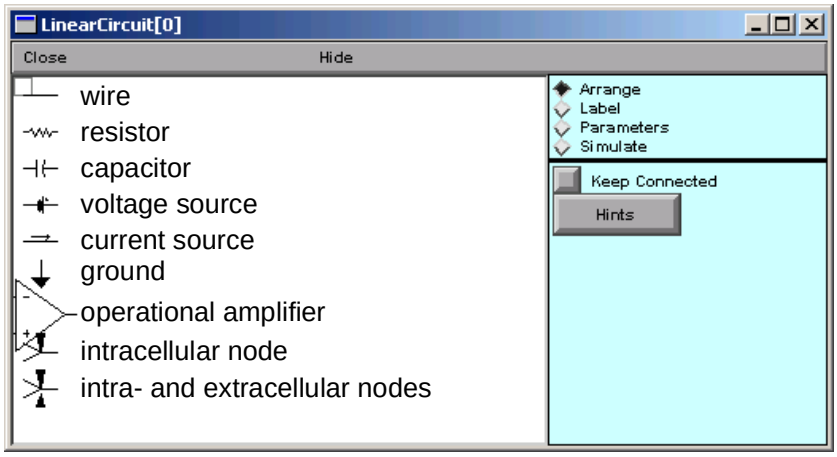
- effects and compensation of electrode R & C
- two-electrode voltage clamp
- ohmic and nonlinear gap junctions

1. Bring up a Linear Circuit Builder



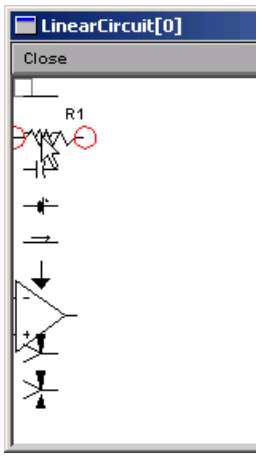
NEURON Main Menu / Build / Linear Circuit

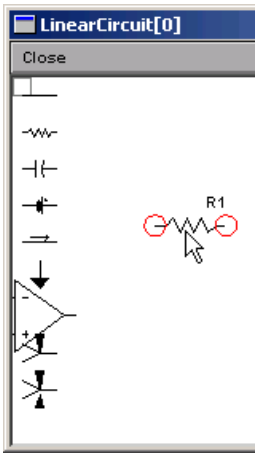
The Linear Circuit Builder



Arrange: spawn components

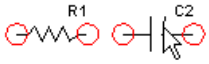
Click on palette and drag onto canvas



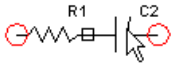


Arrange: connect components

Click and drag to overlap red circles



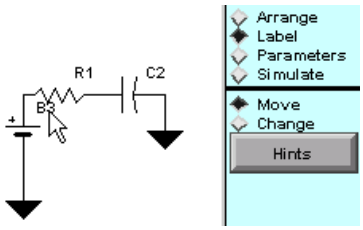
Black square is "solder joint"

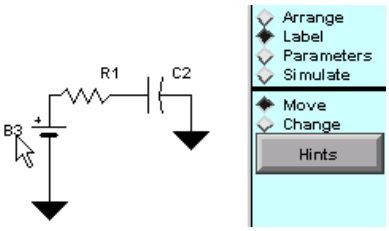


Pull apart to break connection

Label: move labels

Click and drag to new location

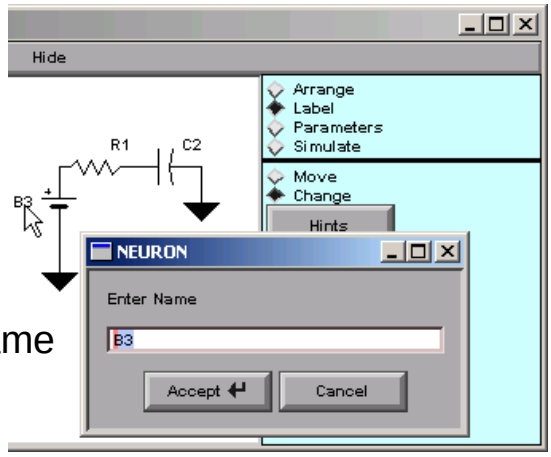




Label: change labels 1

Click on a label . . .

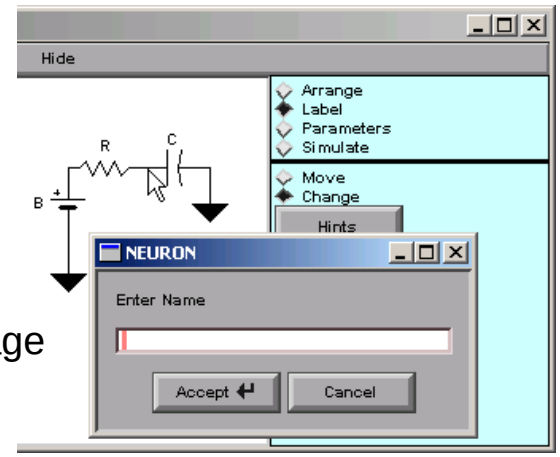
. . . to change its name



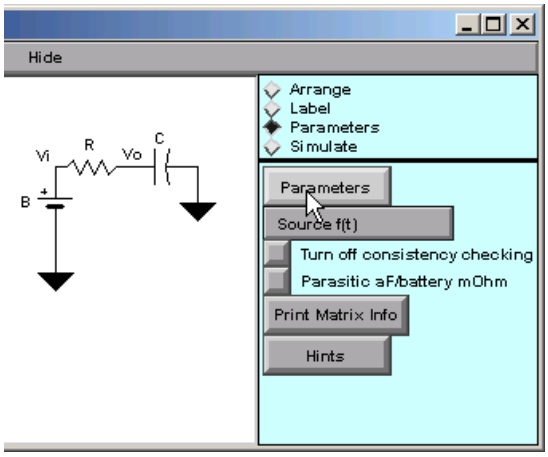
Label: change labels 2

Click on a node . . .

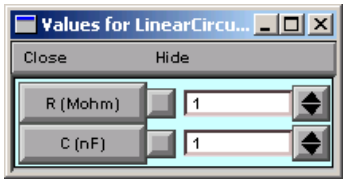
. . . to label a voltage



Parameters: non-source elements

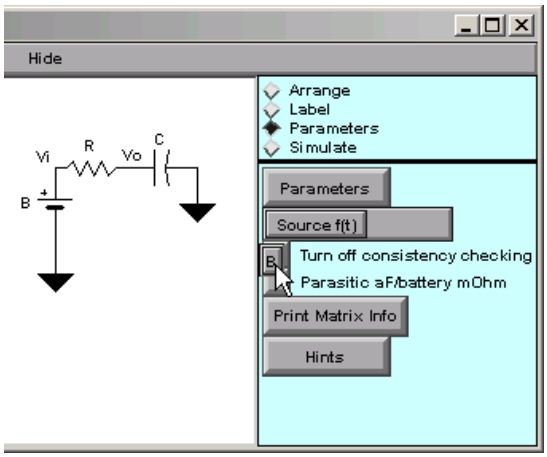


Click on "Parameters"

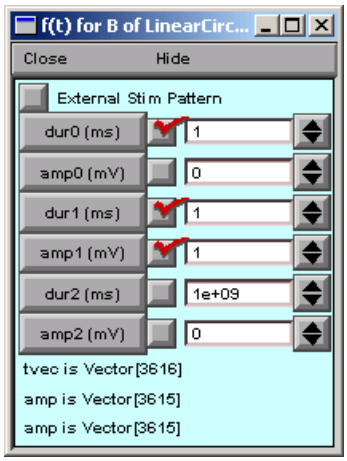


Parameters: signal sources

Source f(t) / B



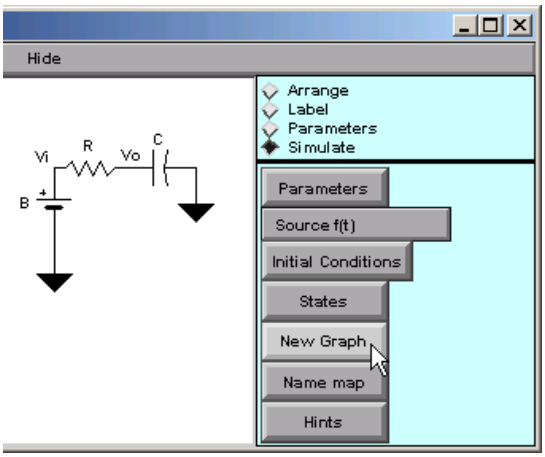
Parameters: signal sources *continued*



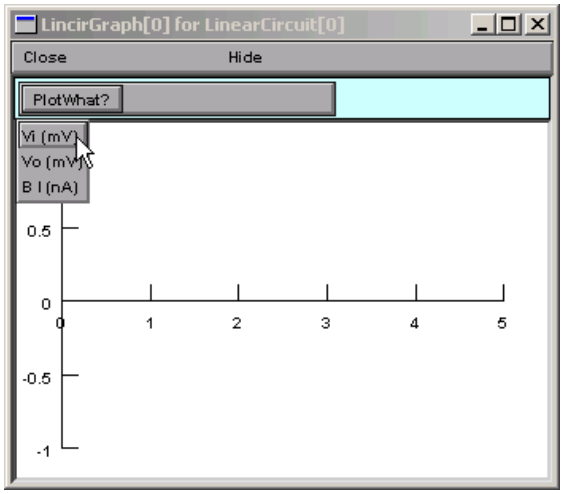
Configured

Simulate: creating a graph

New Graph

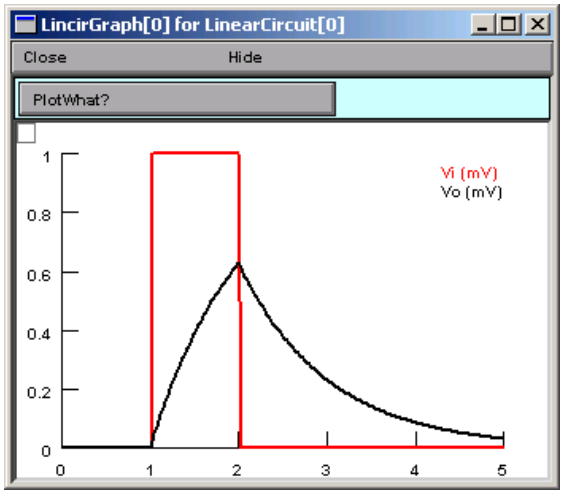


Simulate: specifying what to plot



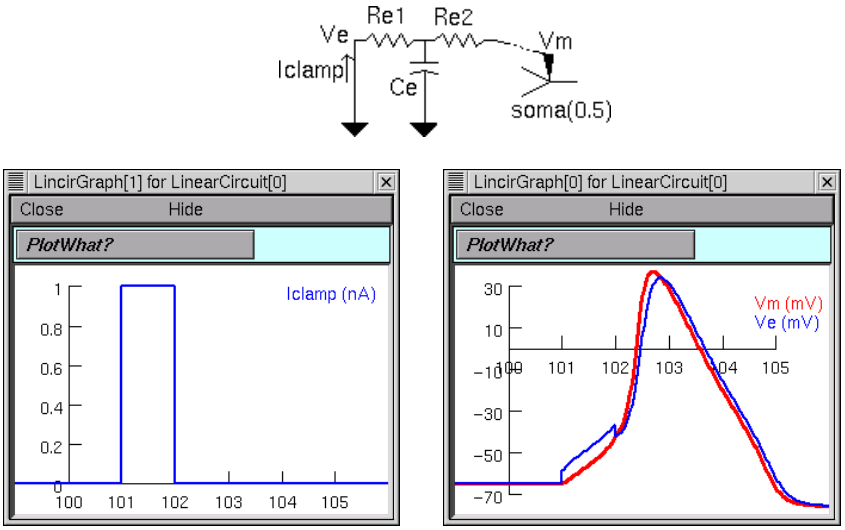
PlotWhat? / *variable_label*

Simulate: simulation results

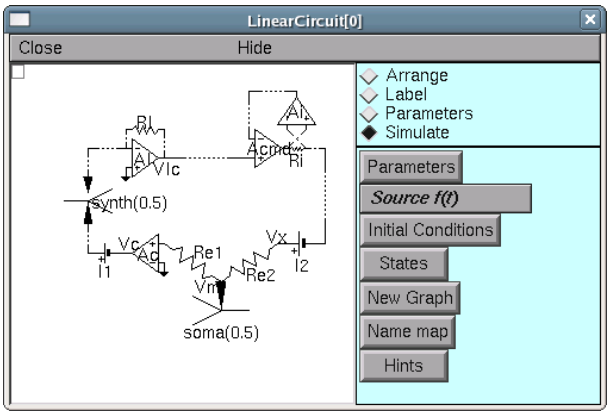


After minor cosmetic changes

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

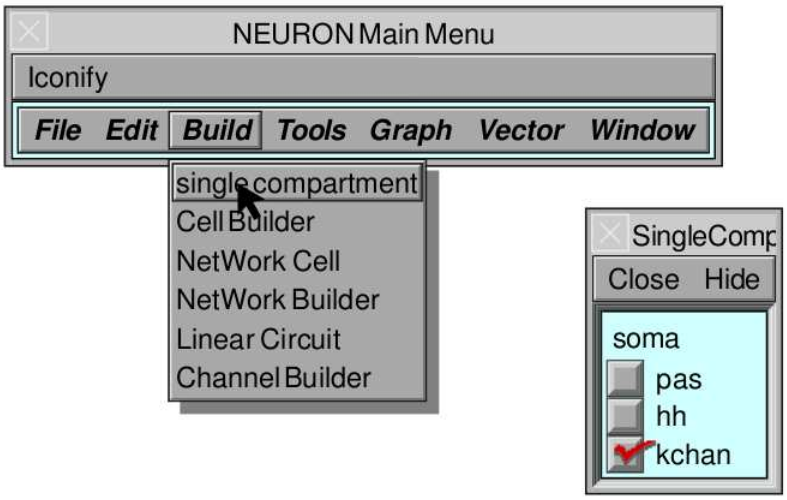
```
PROCEDURE rates(v(mV)) {
    ...
}
```


UNIX

```
nrnivmodl
nrngui
```

MSWIN

Select NEURON Main Menu / Build / single compartment



Density mechanism

```
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)
}
```

Point Process

```
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
}
```

Density mechanism

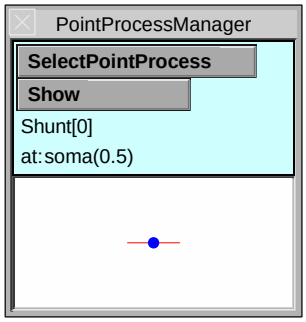
Point Process

NMODL

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

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

GUI



Interpreter

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

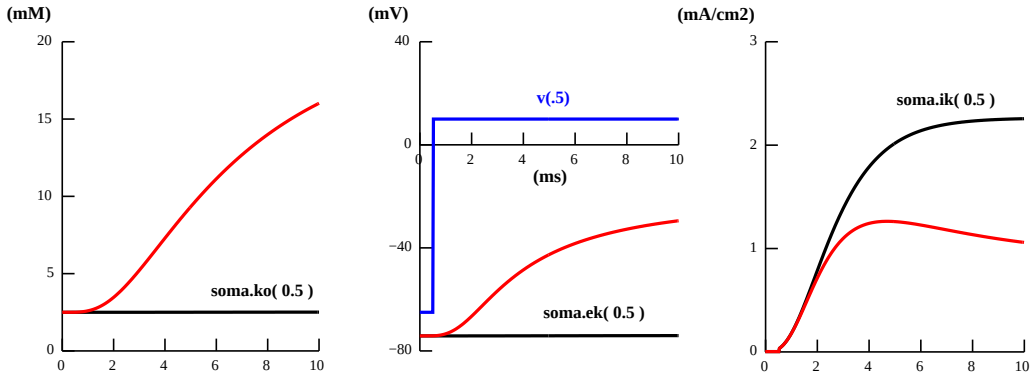
```
objref s
soma s = new Shunt(.5)
s.r = 2
```

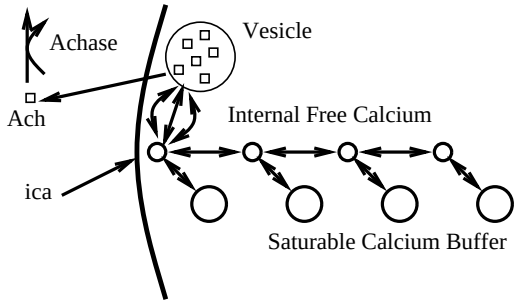
Ion Channel

```
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.

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

The output from
modlunit shunt
is:

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

To fix the problem replace the line with:

```
i = (.001)*(v − e)/r
```

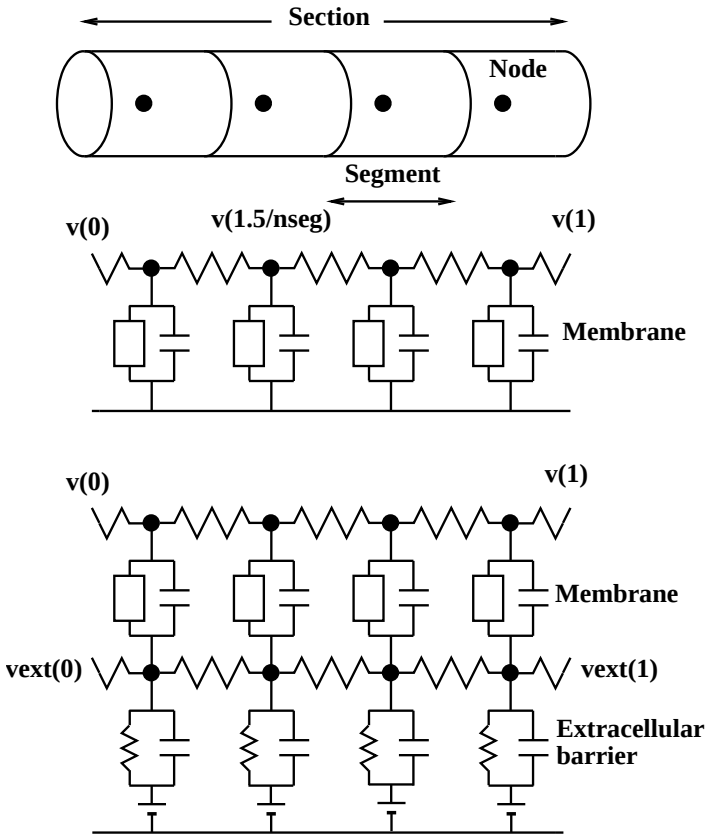
What conversion factor will make the following consistent?

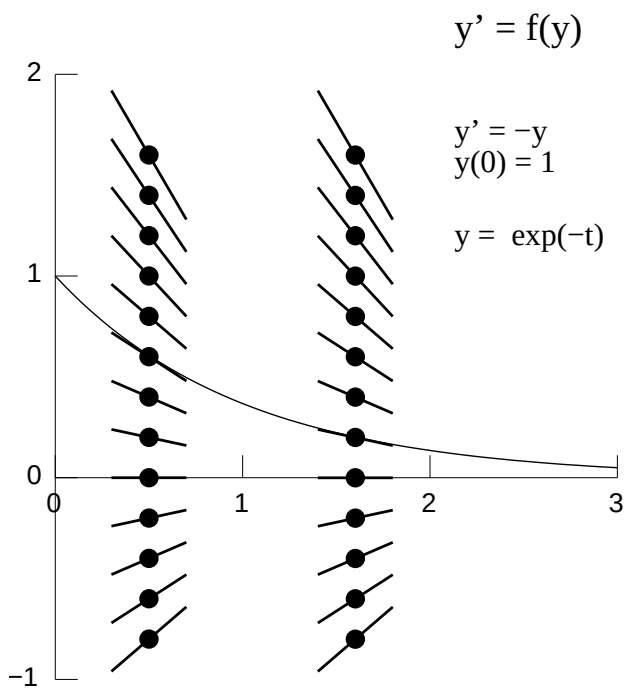
nai' = ina / FARADAY * (c/radius)
(uM/ms) (mA/cm2) / (coulomb/mole) / (um)

Compartmental Modeling

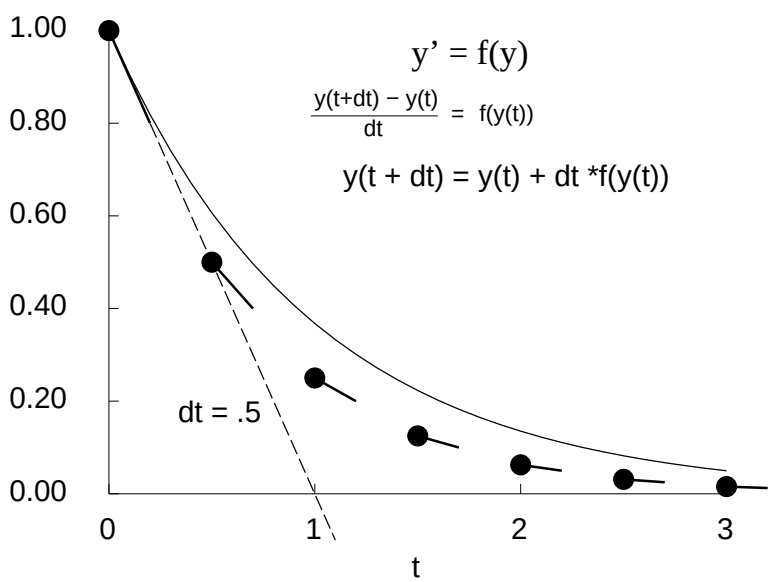
Not much mathematics required.

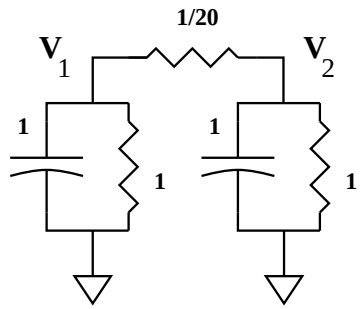
Good judgment essential!

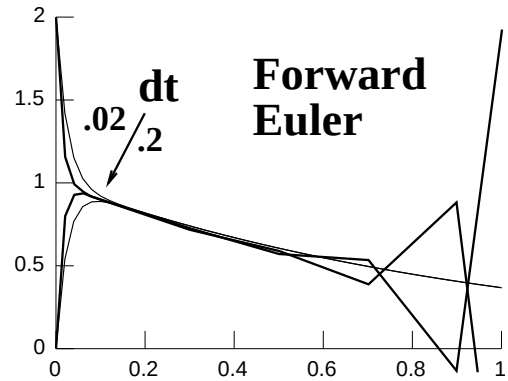




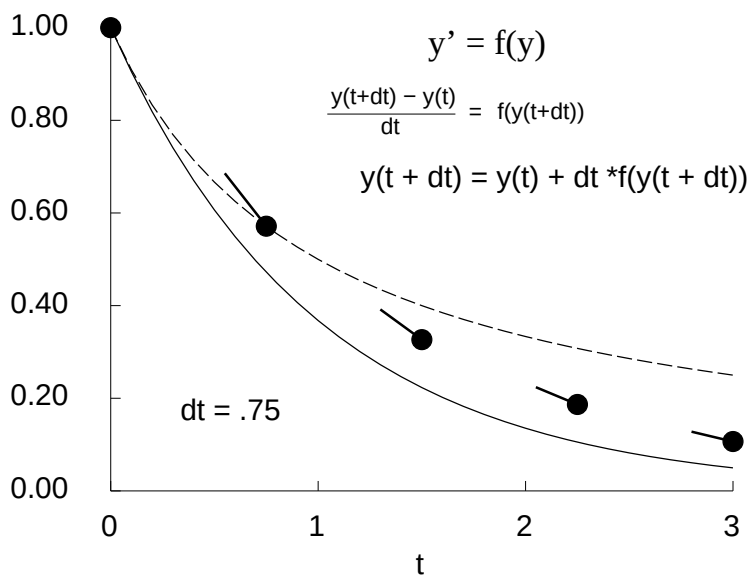
Forward Euler

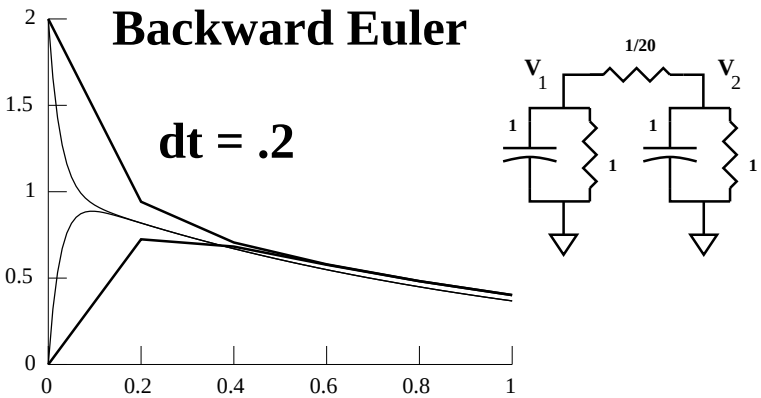


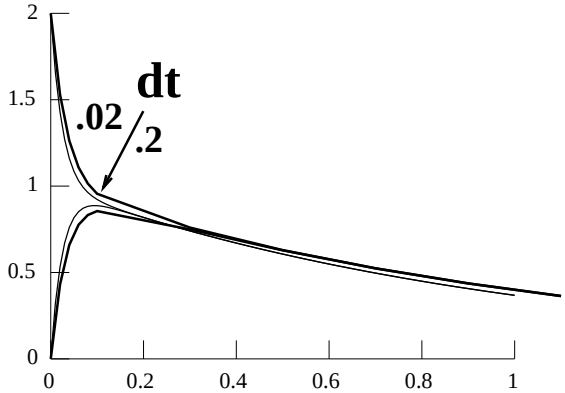


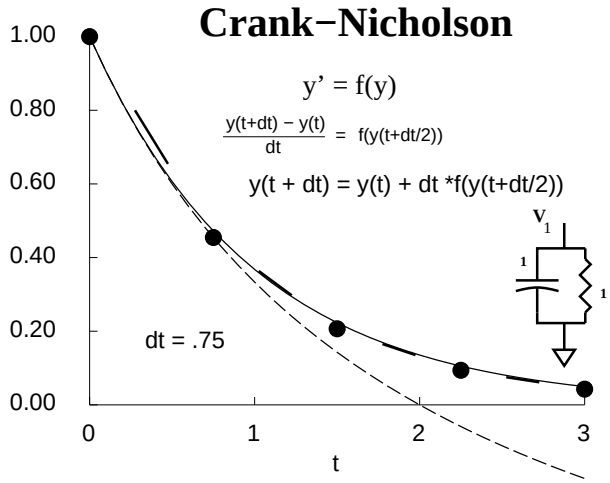


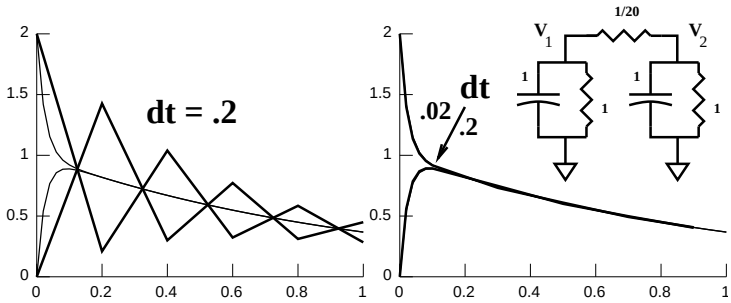
Backward Euler

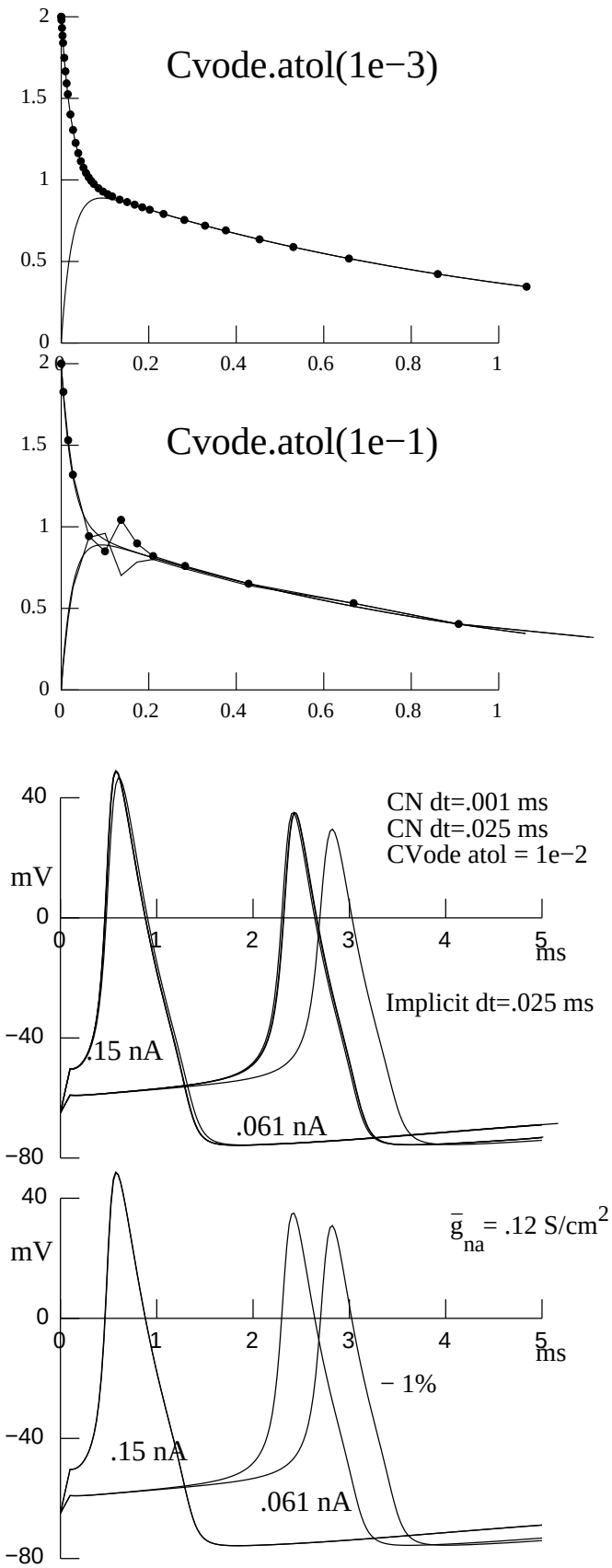












Networks: spike-triggered synaptic transmission, events, and artificial spiking cells

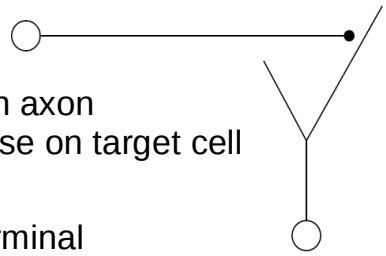
1. Define the types of cells
2. Create each cell in the network
3. Connect the cells

Communication between cells

Gap junctions

Synaptic transmission
- graded
- spike-triggered

Spike-triggered synaptic transmission

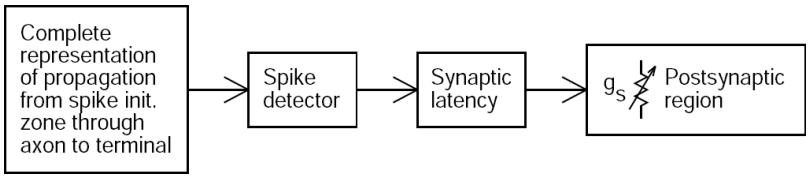


Physical system:
- Presynaptic neuron with axon that projects to synapse on target cell

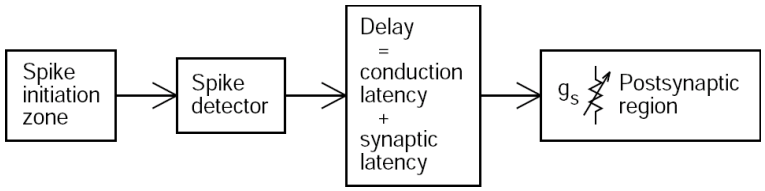
Conceptual model:
- Spike in presynaptic terminal triggers transmitter release; presynaptic details unimportant
- Postsynaptic effect described by DE or kinetic scheme that is perturbed by occurrence of a presynaptic spike

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)
```

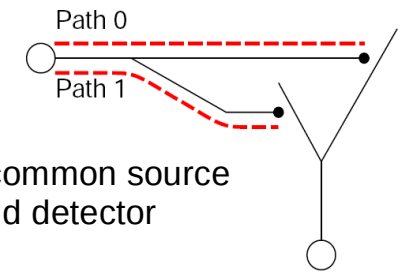
Defaults

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

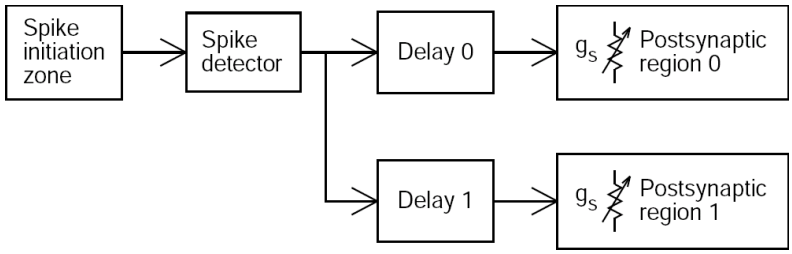
NMODL specification of synaptic mechanism

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

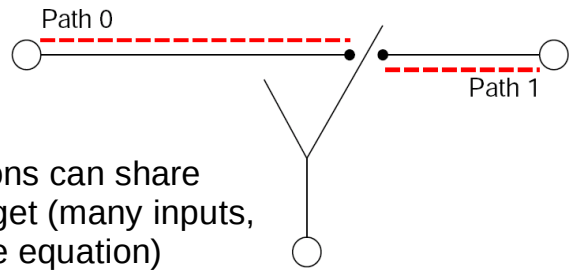
Efficient divergence



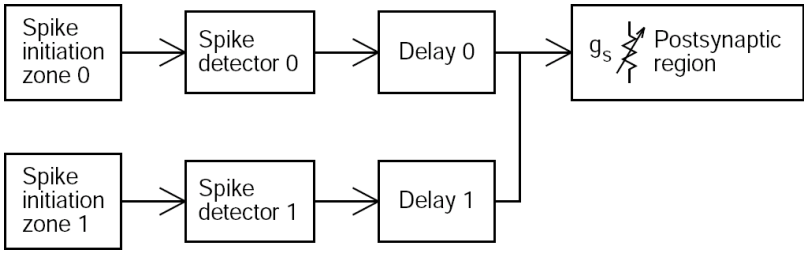
Multiple NetCons with a common source share a single threshold detector



Efficient convergence



Multiple NetCons can share a single target (many inputs, but only one equation)

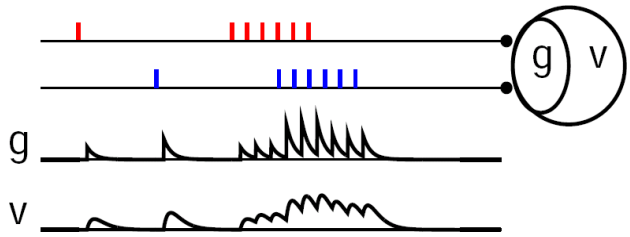


Example: g_s with fast rise and exponential decay

```
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 }
```

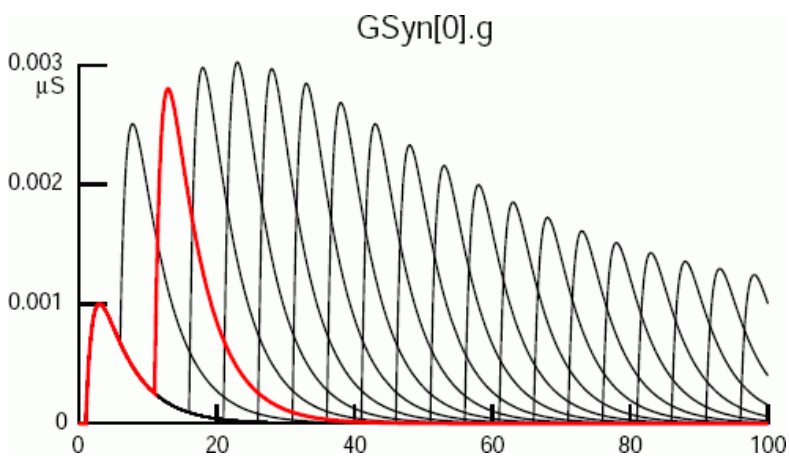
g_s with fast rise and exponential decay

continued



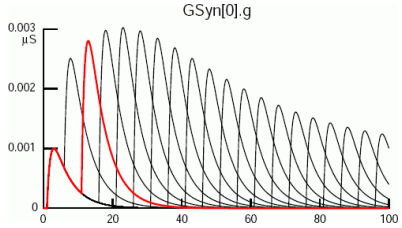

```
BREAKPOINT {
  SOLVE state METHOD cnexp
  i = g*(v-e)
}
DERIVATIVE state { g' = -g/tau }
NET_RECEIVE(w (uS)) { g = g + w }
```

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

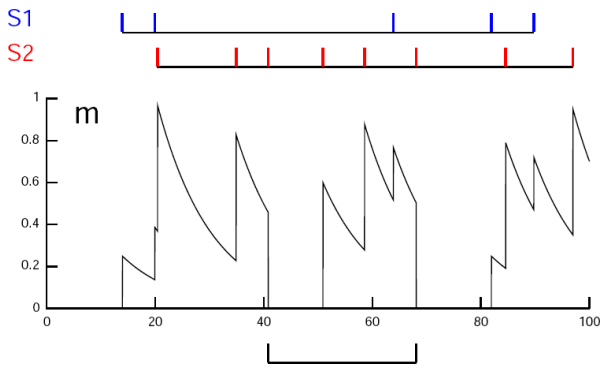
Prerequisite: all state variables must be analytically computable from a new initial condition

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

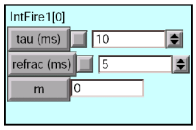
Example: leaky integrate and fire model

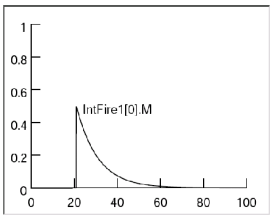


Leaky integrate and fire model *continued*

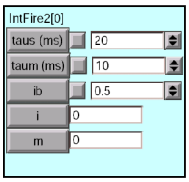
```
NEURON {
  ARTIFICIAL_CELL IntFire
  RANGE tau, m
}
  . . . declarations . . .
INITIAL { m = 0   t0 = t }
NET_RECEIVE (w) {
  m = m*exp(-(t-t0)/tau)
  t0 = t
  m = m + w
  if (m > 1) {
    net_event(t)
    m = 0
  }
}
```

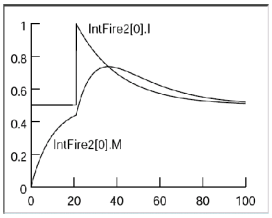

IntFire1



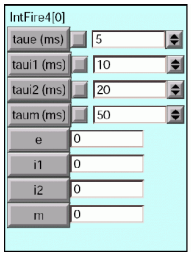


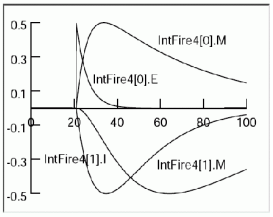
IntFire2





IntFire4





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
}
```

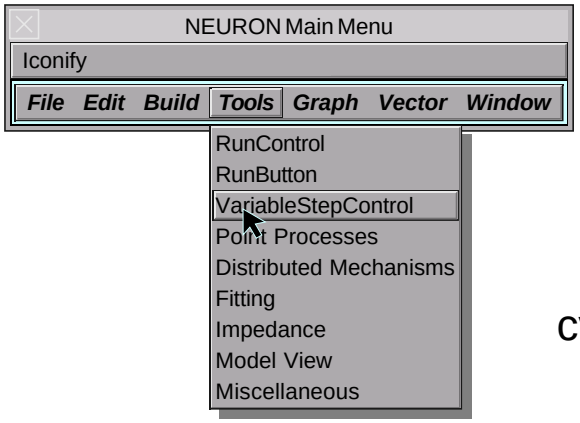
or iterate over targets?

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

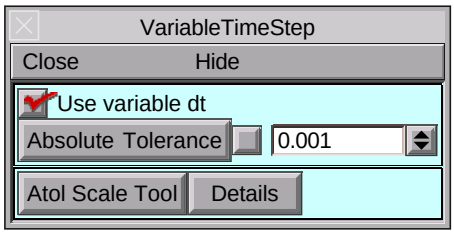
Connecting cells

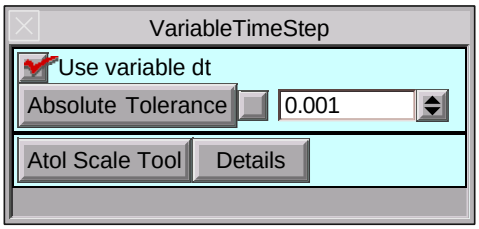
For a net distributed over multiple CPUs,
it is most efficient to iterate over targets first.

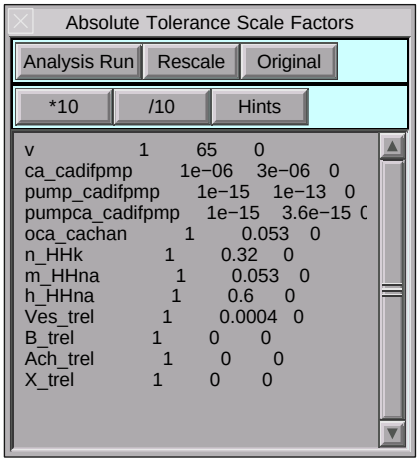
```
for each cell {
  connect sources to this cell
}
```

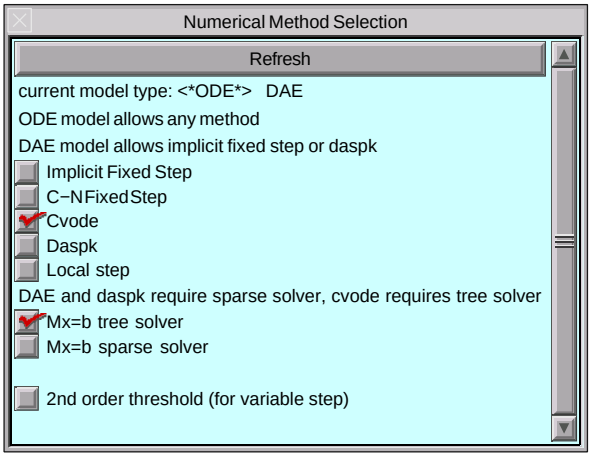


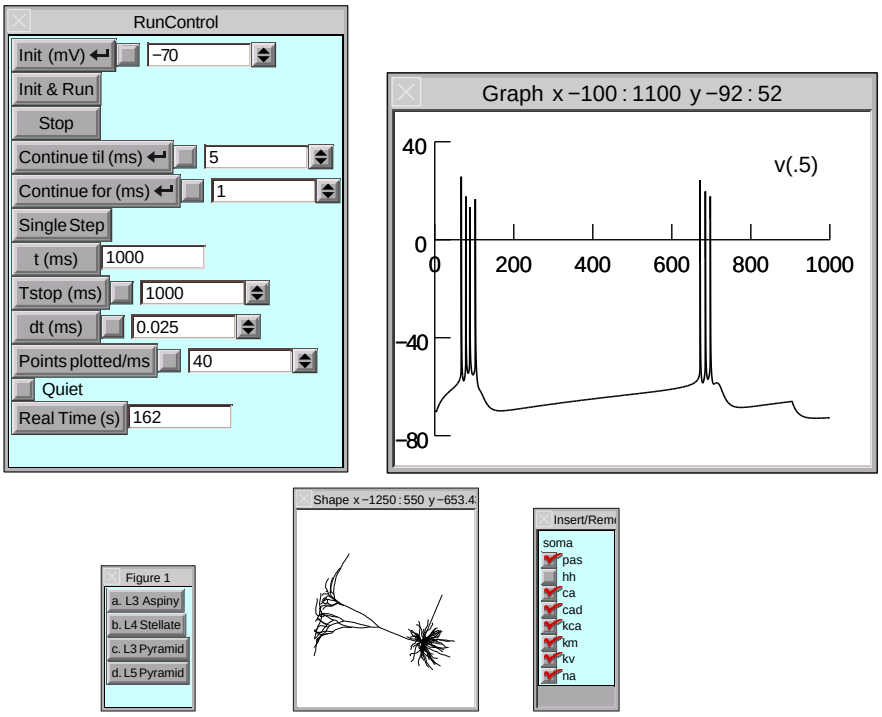
cvode_active(1)

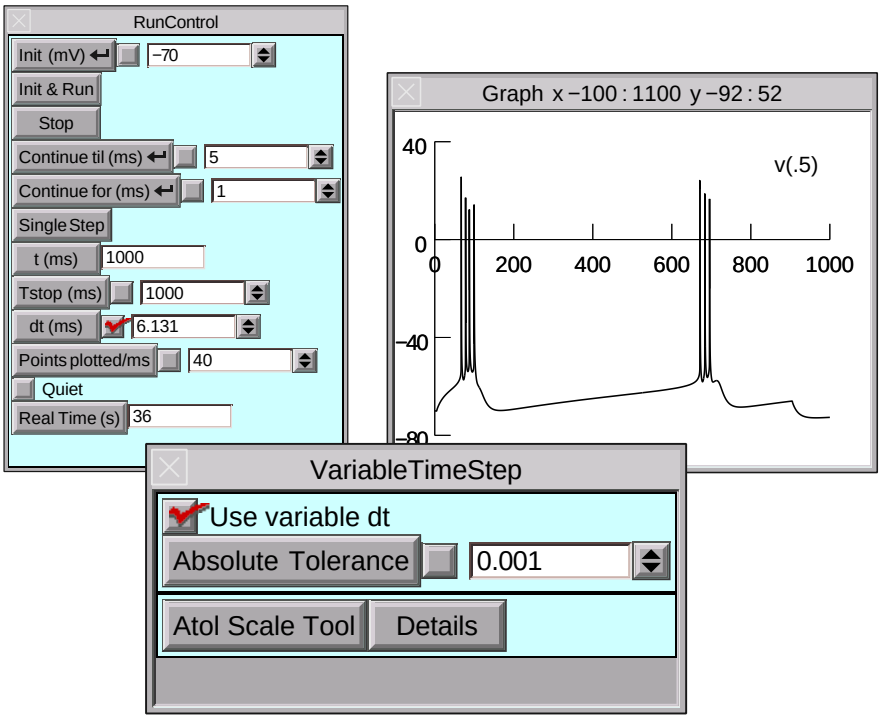


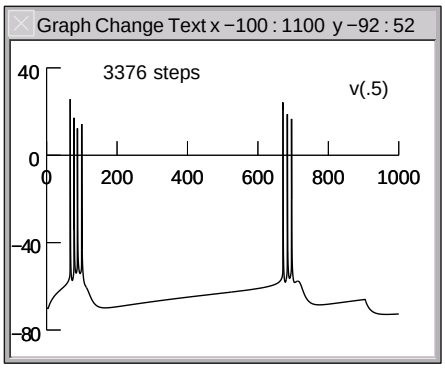


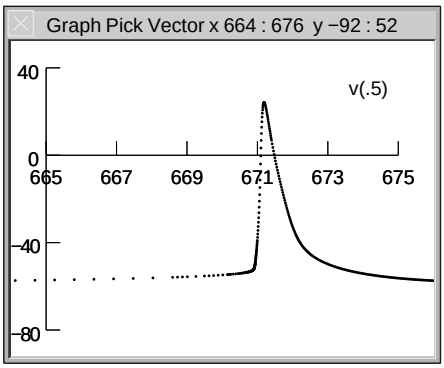


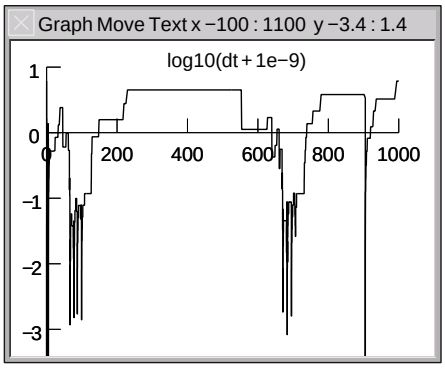


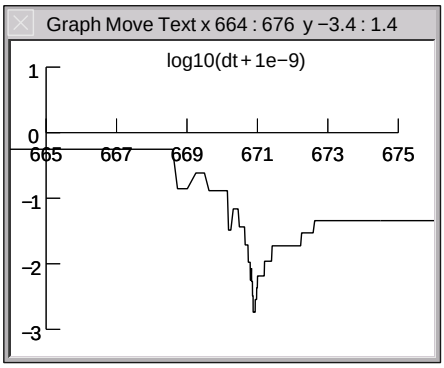


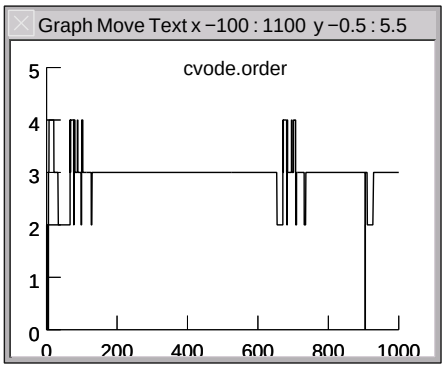


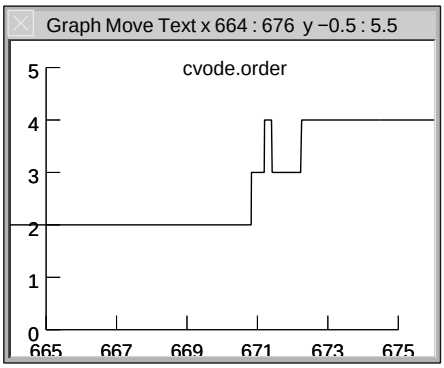


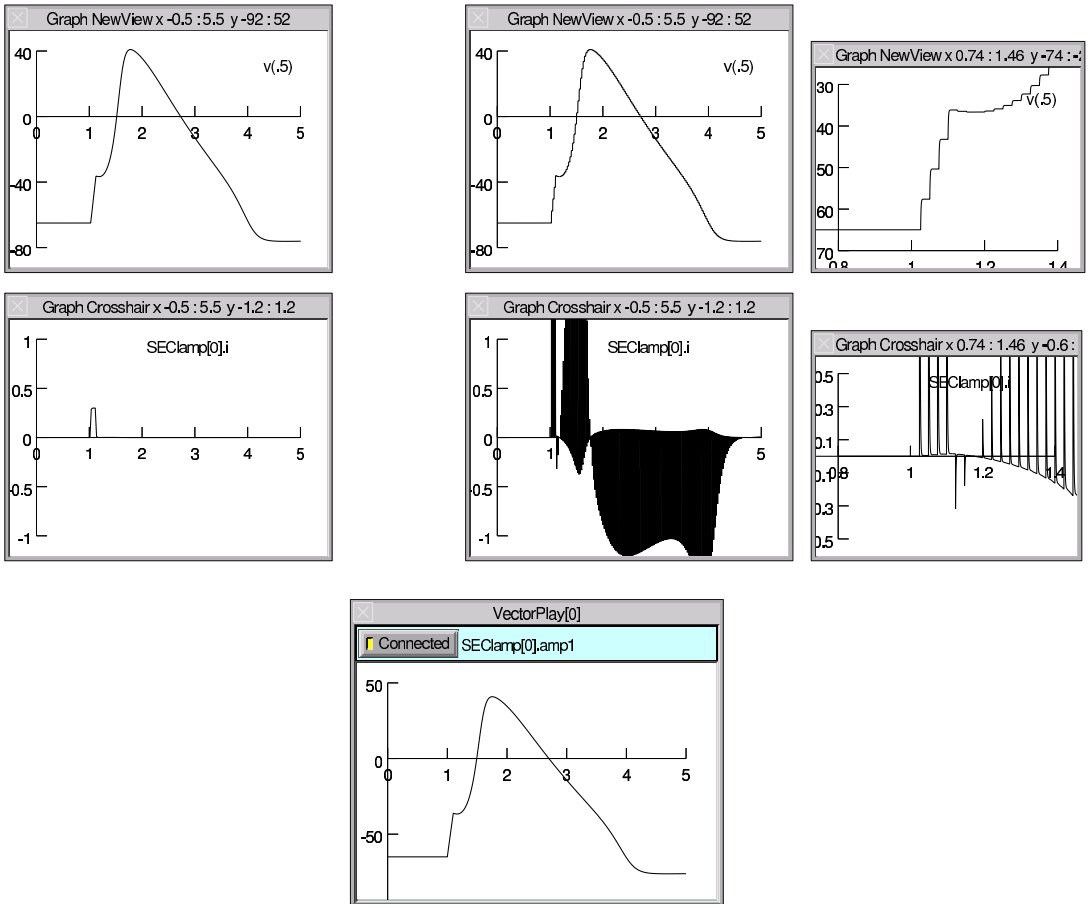


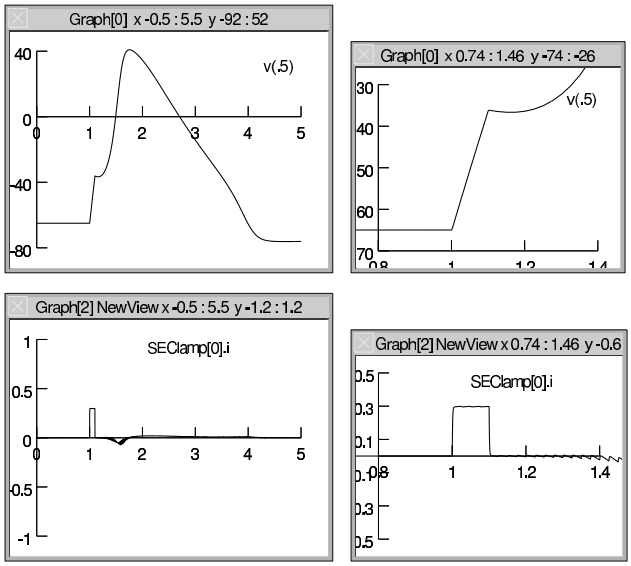













```
soma vvec.play(&SEClamp[0].amp1, tvec, 1)
```


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?

16384 core EPFL IBM BlueGene/P

1 hour at 850MHz

6 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.

- Screensaver Calin−Jageman and Katz, 2006
- Bulletin−board (Linda)

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

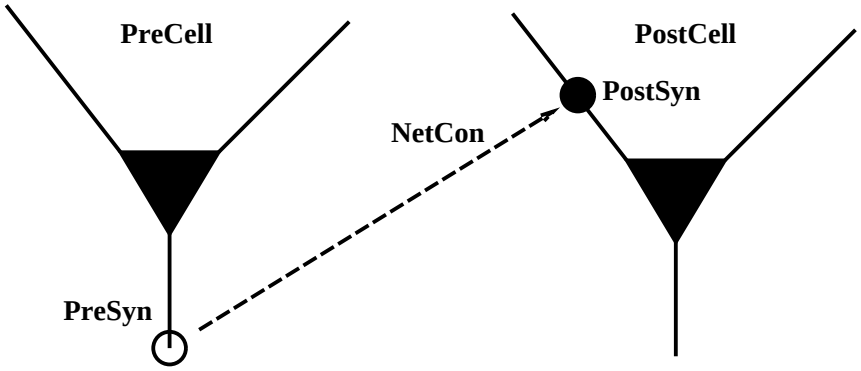
Parallel Computation

A simulation run takes hours.

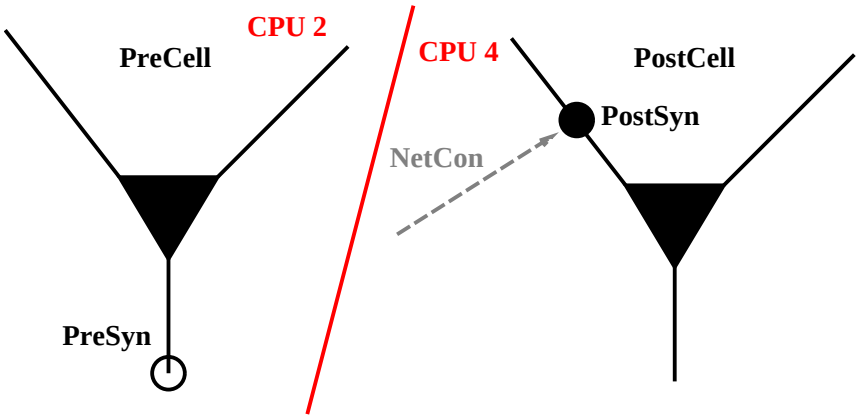
want to spread the problem over several machines.

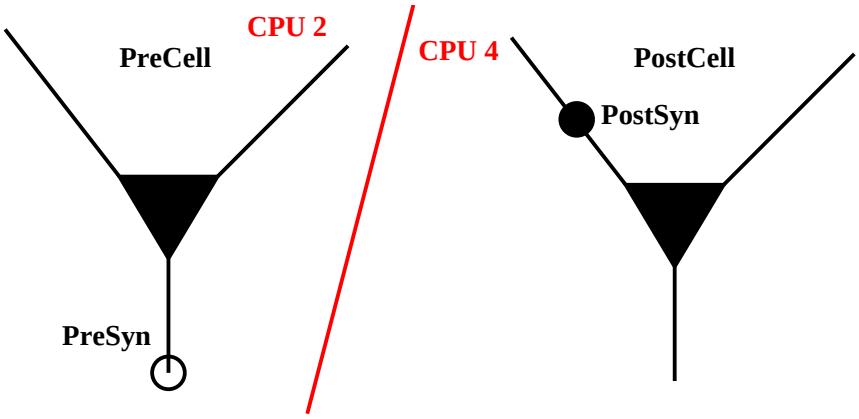
Single cells

portions of the tree cable equation on different machines.

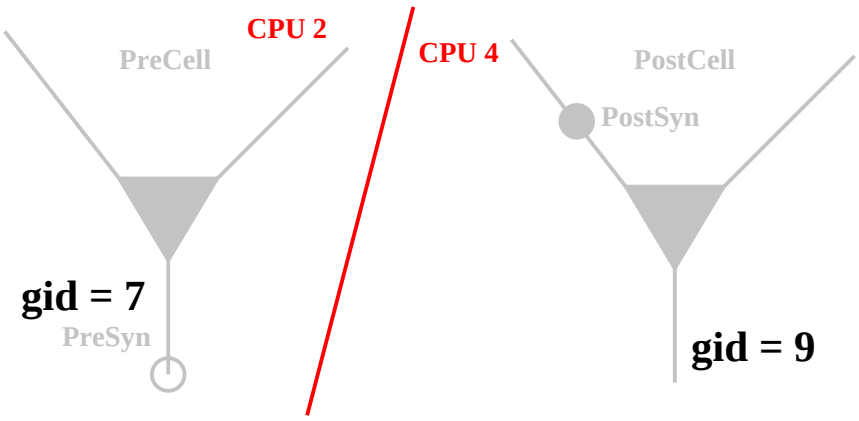



```
nc = new NetCon(PreSyn, PostSyn)
```






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



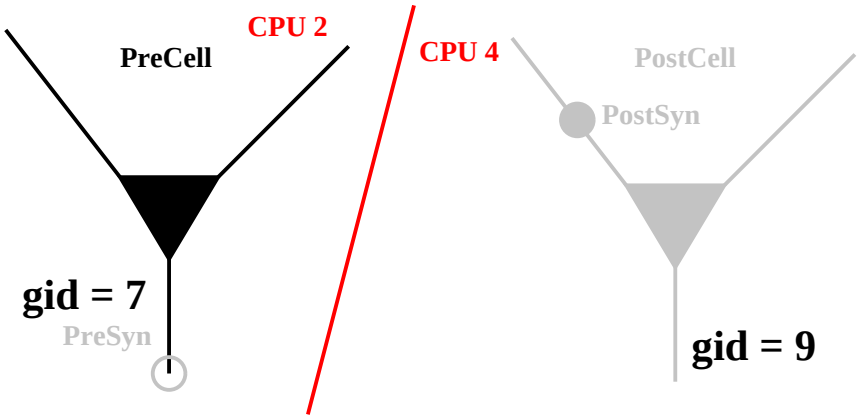
Every spike source (cell) must have a global id number.

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:

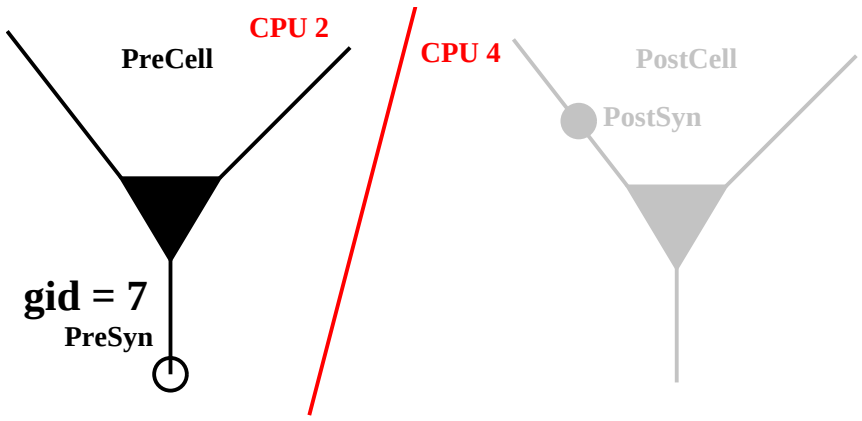
```
for (gid = pc.id; gid < ncell; gid += pc.nhost)
    pc.set_gid2node(gid, pc.id)
    ...
}
```

body executed only ncell/nhost times, not ncell.



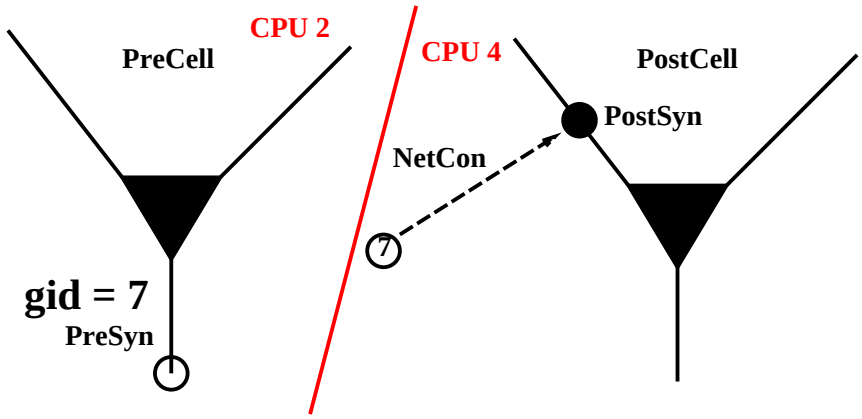
Create cell only where the gid exists.

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

Associate gid with spike source.

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



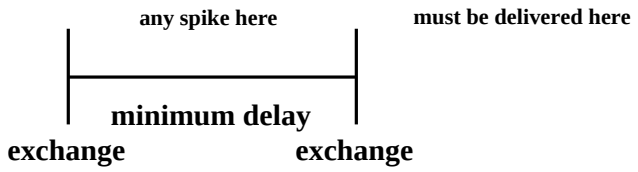
Create NetCon on CPU where target exists.

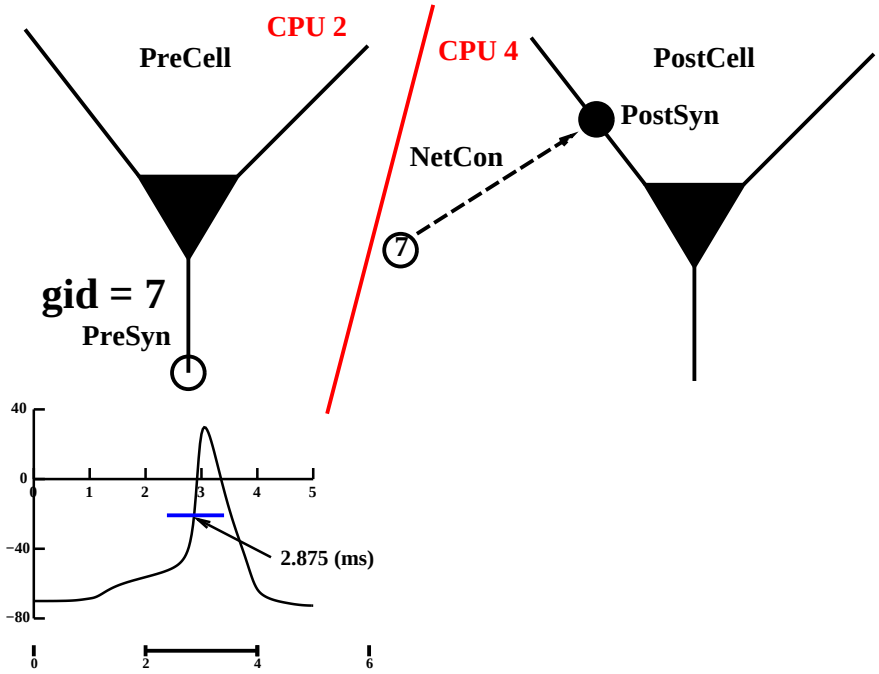
```
nc = pc.gid_connect(7, PostSyn)
```

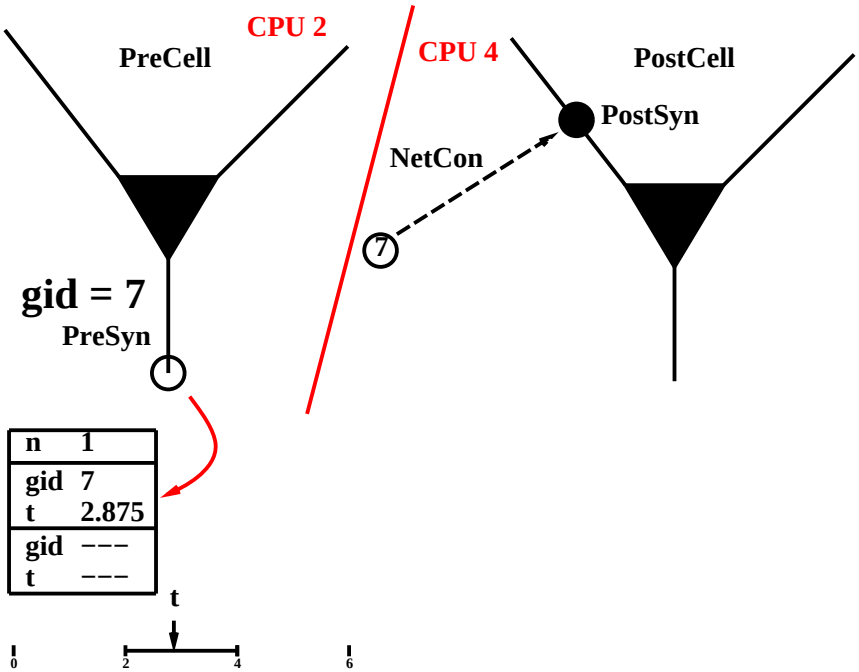

Run using the idiom

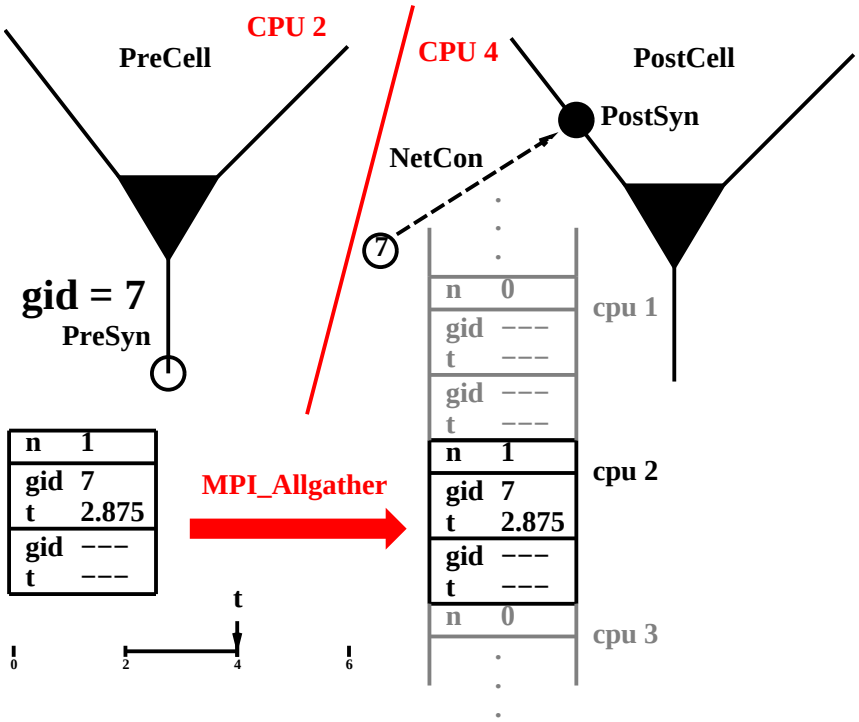
```
pc.set_maxstep(10)
stdinit()
pc.psolve(tstop)
```

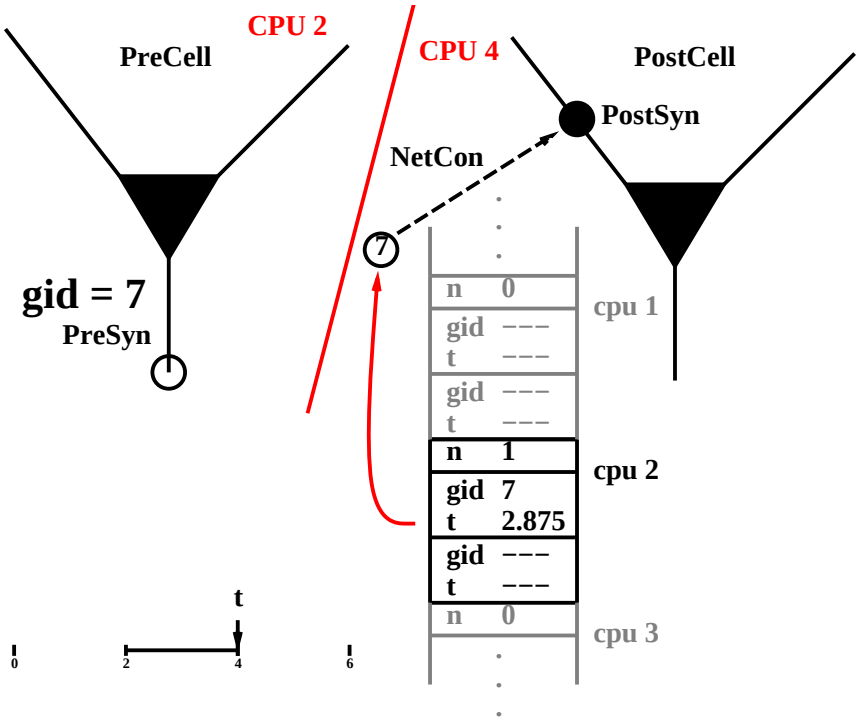
pc.set_maxstep() uses MPI_Allreduce to determine minimum delay.



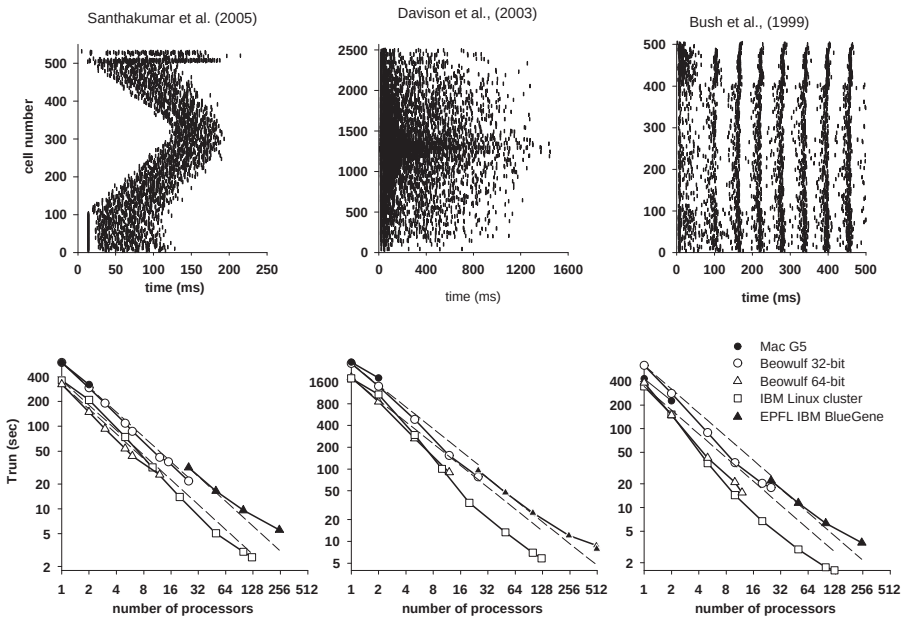


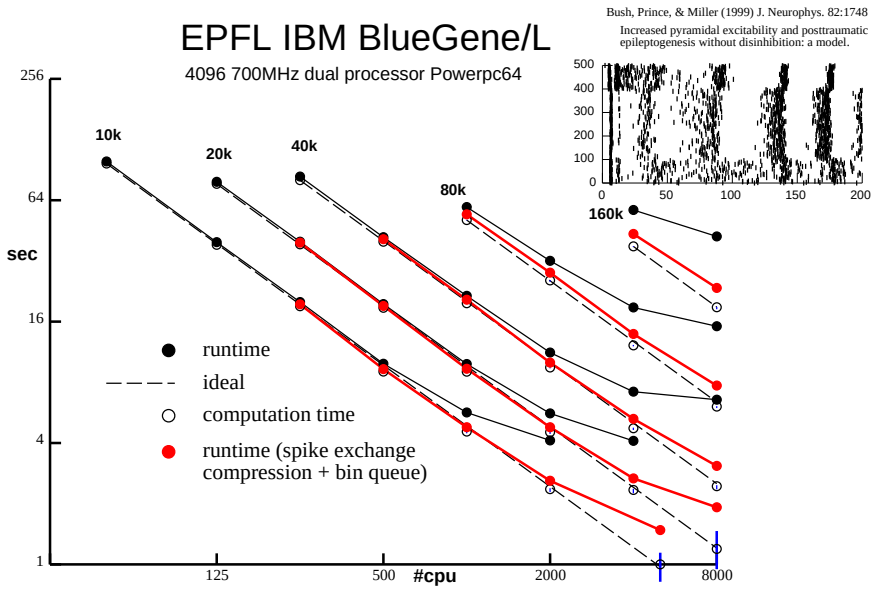




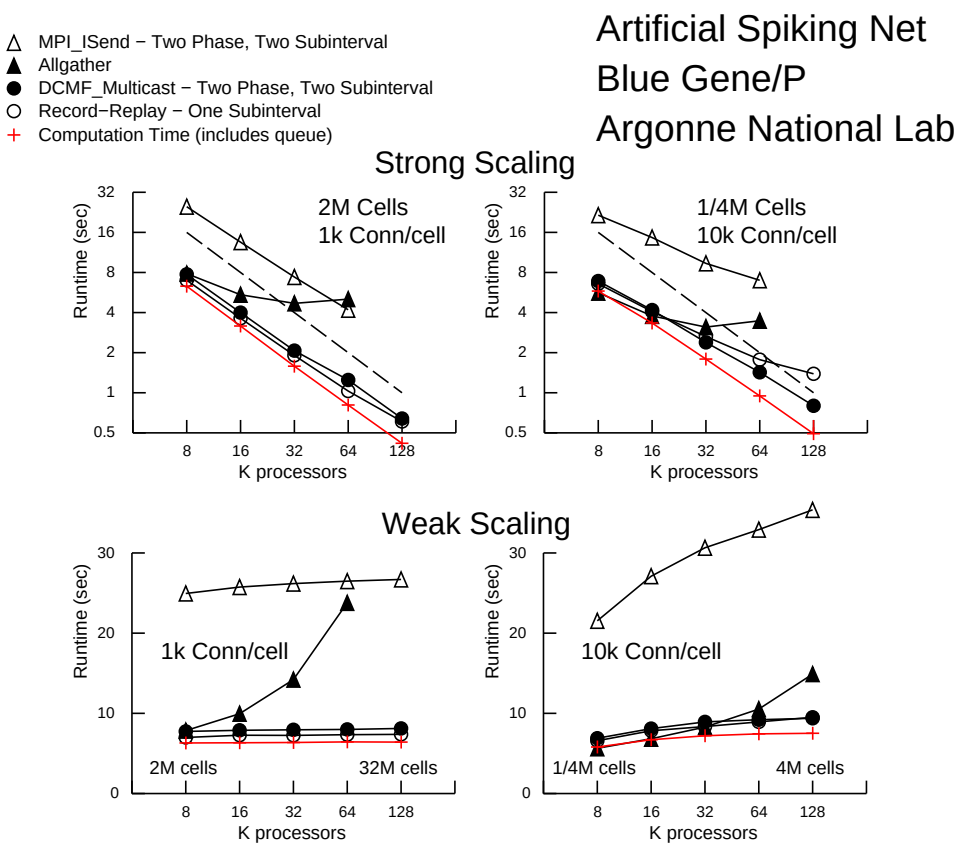


Migliore et al (2006) J. Comput. Neurosci. 21(2):119

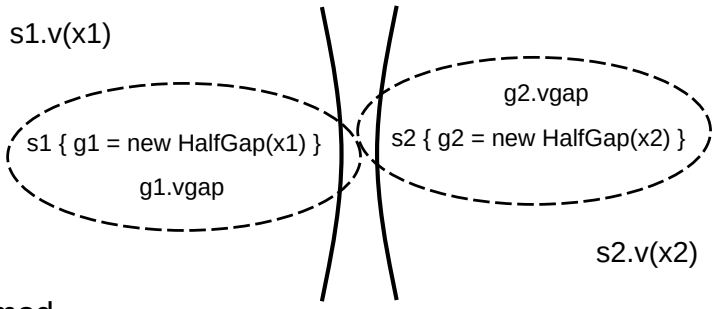




#cells	#states	#netcons	#outputSpikes	#spikedeliver
10k	444,664	17,167,785	31,118	52,040,794
20k	888,664	68,655,566	33,960	110,634,002
40k	1,777,664	274,591,128	69,529	452,987,907
80k	3,553,664	1,098,302,267	294974	2,147,483,647 *
160k	7,107,664	4,393,084,577	844,175	22,847,784,937



Continuous Voltage Exchange



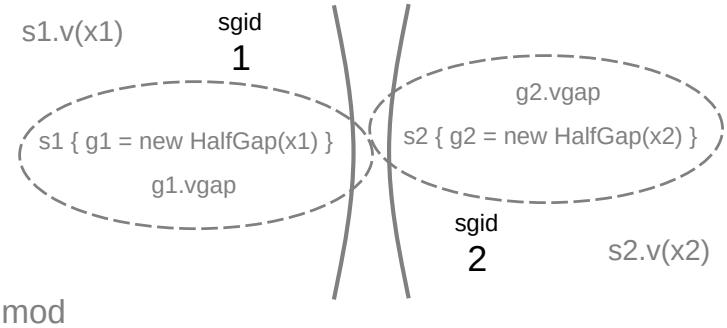
gap.mod

```
NEURON {
  POINT_PROCESS HalfGap
  ELECTRODE_CURRENT i
  RANGE r, i, vgap
}
PARAMETER { r = 1e9 (megohm) }
```

```
ASSIGNED {
  v (millivolt)
  vgap (millivolt)
  i (nanoamp)
}
CURRENT { i = (vgap − v)/r }
```

Continuous Voltage Exchange

pc.source_var(&source_var, sgid)



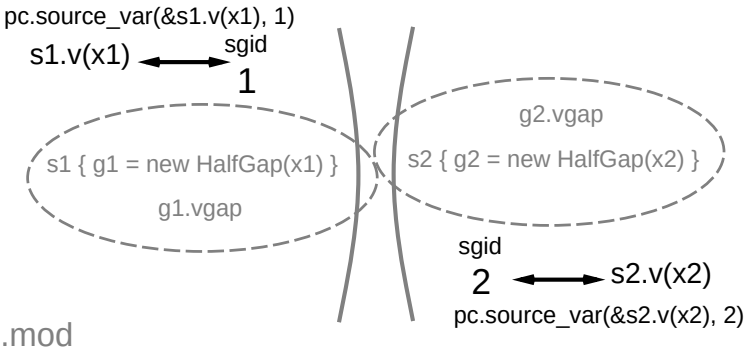
gap.mod

```
NEURON {
  POINT_PROCESS HalfGap
  ELECTRODE_CURRENT i
  RANGE r, i, vgap
}
PARAMETER { r = 1e9 (megohm) }
```

```
ASSIGNED {
  v (millivolt)
  vgap (millivolt)
  i (nanoamp)
}
CURRENT { i = (vgap − v)/r }
```


Continuous Voltage Exchange

pc.source_var(&source_var, sgid)



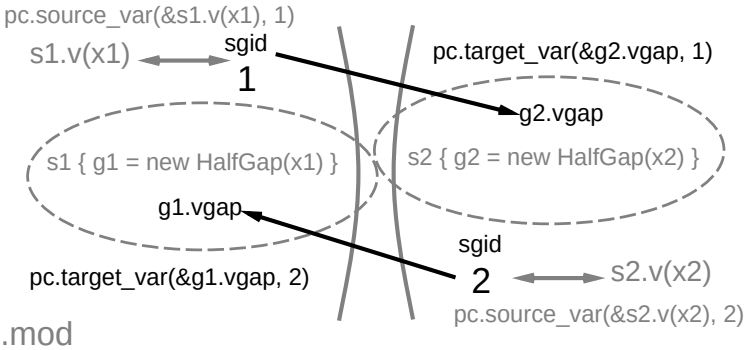
gap.mod

```
NEURON {
  POINT_PROCESS HalfGap
  ELECTRODE_CURRENT i
  RANGE r, i, vgap
}
PARAMETER { r = 1e9 (megohm) }
```

```
ASSIGNED {
  v (millivolt)
  vgap (millivolt)
  i (nanoamp)
}
CURRENT { i = (vgap − v)/r }
```

Continuous Voltage Exchange

pc.source_var(&source_var, sgid)
pc.target_var(&target_var, sgid)



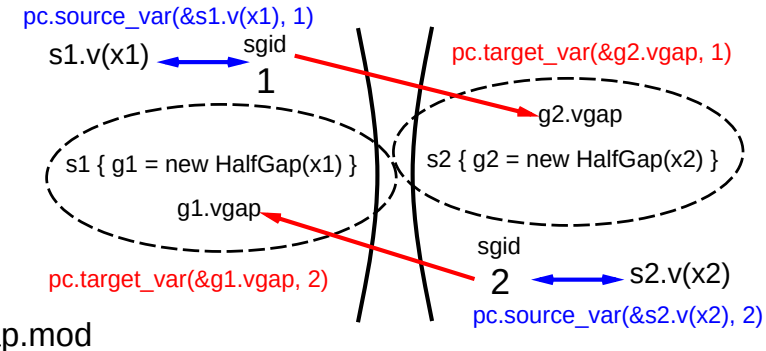
gap.mod

```
NEURON {
  POINT_PROCESS HalfGap
  ELECTRODE_CURRENT i
  RANGE r, i, vgap
}
PARAMETER { r = 1e9 (megohm) }
```

```
ASSIGNED {
  v (millivolt)
  vgap (millivolt)
  i (nanoamp)
}
CURRENT { i = (vgap − v)/r }
```


Continuous Voltage Exchange

pc.source_var(&source_var, sgid)
pc.target_var(&target_var, sgid)



gap.mod

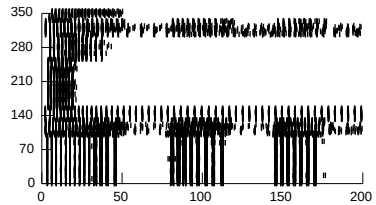
```
NEURON {
  POINT_PROCESS HalfGap
  ELECTRODE_CURRENT i
  RANGE r, i, vgap
}
PARAMETER { r = 1e9 (megohm) }
```

```
ASSIGNED {
  v (millivolt)
  vgap (millivolt)
  i (nanoamp)
}
CURRENT { i = (vgap - v)/r }
```

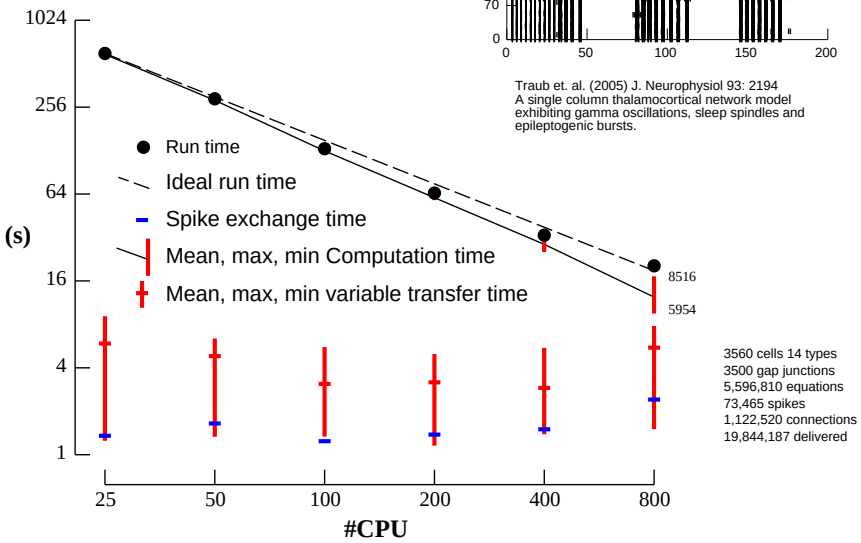
Pittsburgh Supercomputing Center

Bigben Cray XT3

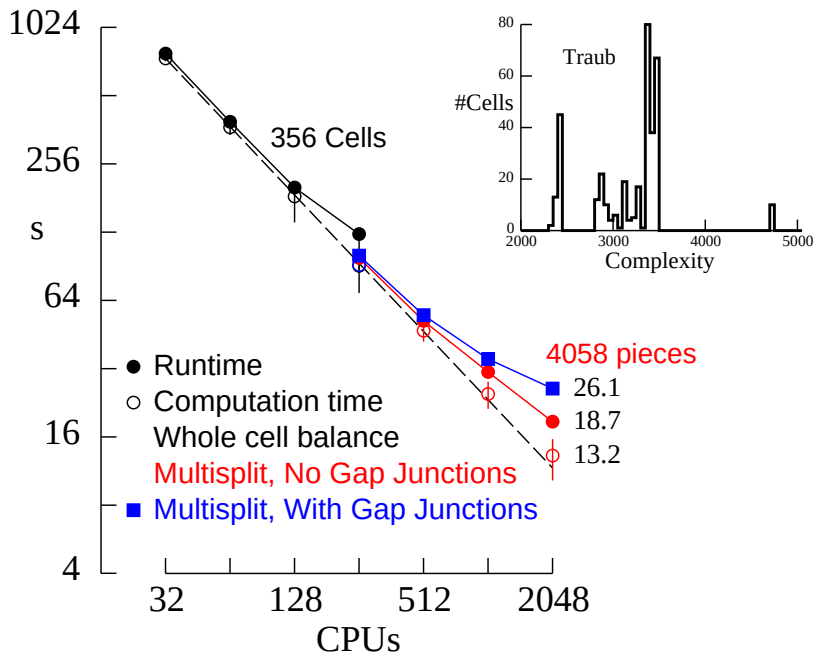
2068 2.4 GHz Opteron Processors



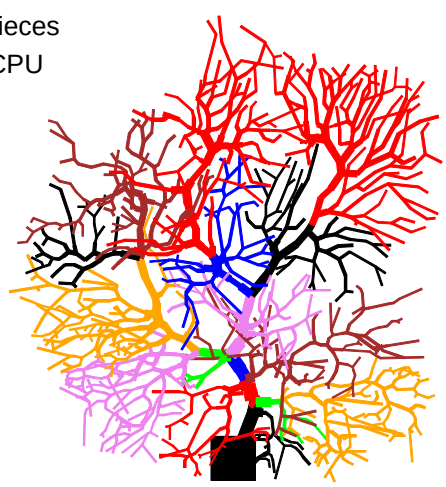
Traub et. al. (2005) J. Neurophysiol 93: 2194
A single column thalamocortical network model exhibiting gamma oscillations, sleep spindles and epileptogenic bursts.

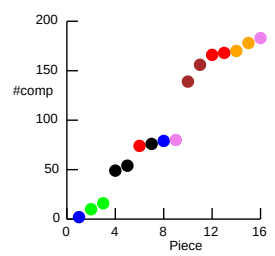


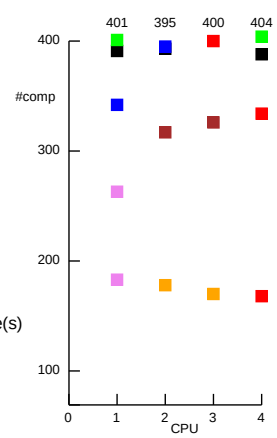
3560 cells 14 types
3500 gap junctions
5,596,810 equations
73,465 spikes
1,122,520 connections
19,844,187 delivered



16 Pieces
4 CPU



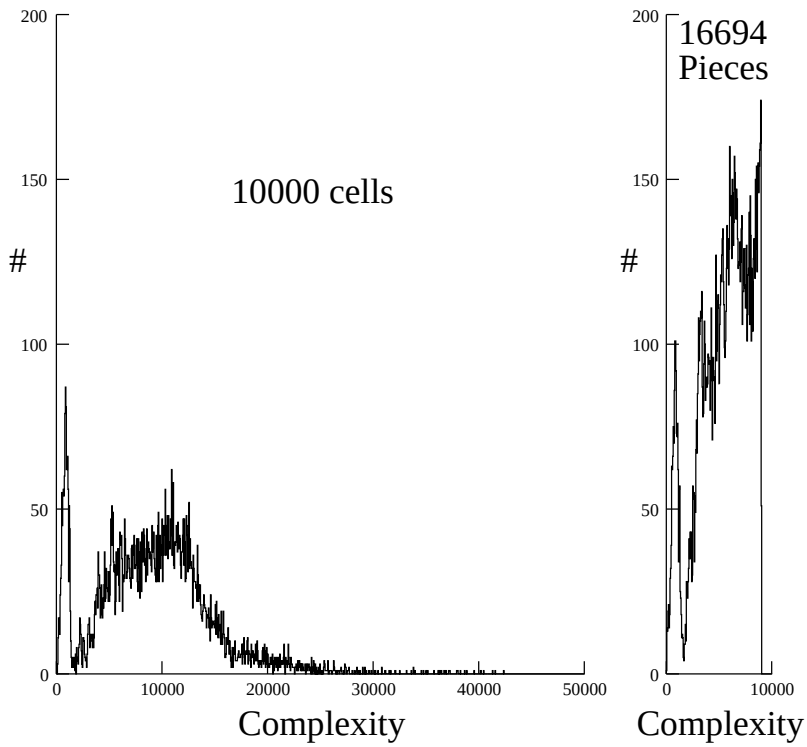


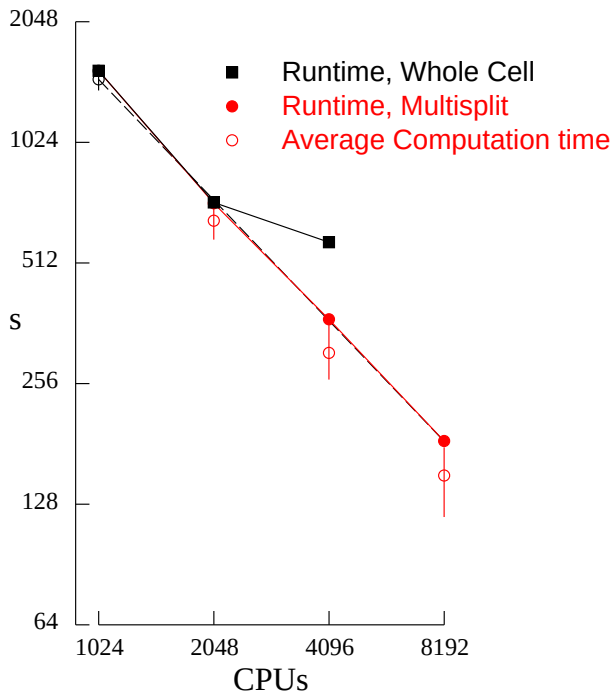


Time (s)

CPU	Computation	Exchange
0	13.82	0.56
1	13.35	1.03
2	13.47	0.90
3	13.56	0.82

	Runtime(s)
16 pieces, 1 cpu	55.0
wholecell, 1 cpu	56.2
16 pieces, 4 cpu	14.4





Scripting NEURON with Python

Robert A. McDougal

Yale School of Medicine

16 October 2015

What is a script?

A **script** is a file with computer-readable instructions for performing a task.

In NEURON, scripts can: set-up a model, define and perform an experimental protocol, record data, . . .

Why write scripts for NEURON?

- Automation ensures consistency and reduces manual effort.
- Facilitates comparing the suitability of different models.
- Facilitates repeated experiments on the same model with different parameters (e.g. drug dosages).
- Facilitates recollecting data after change in experimental protocol.
- Provides a complete, reproducible version of the experimental protocol.

Introduction to Python

Displaying results

The `print` command is used to display non-graphical results.

It can display fixed text:

```
print ('Hello everyone.')
```

```
Hello everyone.
```

or the results of a calculation:

```
print (5 * (3 + 2))
```

```
25
```

Storing results

Give values a name to be able to use them later.

```
a = max([1.2, 5.2, 1.7, 3.6])
print (a)
```

```
5.2
```

In Python 2.x, `print` is a keyword and the parentheses are unnecessary. Using the parentheses allows your code to work with both Python 2.x and 3.x.

Using libraries

Libraries ("modules" in Python) provide features scripts can use.
To load a module, use `import`:

```
import math
```

Use dot notation to access a function from the module:

```
print (math.cos(math.pi / 3))
```

```
0.5
```

One can also load specific items from a module.
For NEURON, we often want:

```
from neuron import h, gui
```

Other modules

Python ships with a large number of modules, and you can install more (like NEURON). Useful ones for neuroscience include: `math` (basic math functions), `numpy` (advanced math), `matplotlib` (2D graphics), `mayavi` (3D graphics), `pandas` (analysis and databasing), ...

Finding help within Python

To get a list of functions, etc in a module (or class) use `dir`:

```
import numpy
print (dir(numpy))
```

Displays:

```
['__doc__', '__name__', '__package__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', ...
```

To see help information for a specific function, use `help`:

```
help(math.cosh)
```


Online resources

Python is widely used, and there are many online resources available, including:

- docs.python.org – the official documentation
- Stack Overflow – a general-purpose programming forum
- the NEURON programmer's reference – NEURON documentation
- the NEURON forum – for NEURON-related programming questions

Basic NEURON scripting

Creating and naming sections

A `section` in NEURON is an unbranched stretch of e.g. dendrite.

To create a section, use `h.Section` and assign it to a variable:

```
dend1 = h.Section()
```

A section can have multiple references to it. If you set `a = dend1`, there is still only one section. Use `==` to see if two variables refer to the same section:

```
print (a == dend1)                    True
```

To name a section, declare a `name` attribute:

```
dend2 = h.Section(name='apical')
```

To access the name, use `.name()`:

```
print (dend2.name())                  apical
```

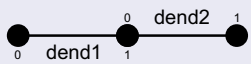
Also available: a `cell` attribute for grouping sections by cell.

Connecting sections

To reconstruct a neuron's full branching structure, individual sections must be connected using `.connect`:

```
dend2.connect(dend1(1))
```

Each section is oriented and has a 0- and a 1-end. In NEURON, traditionally the 0-end of a section is attached to the 1-end of a section closer to the soma. In the example above, dend2's 0-end is attached to dend1's 1-end.



To print the topology of cells in the model, use `h.topology()`. The results will be clearer if the sections were assigned names.

```
h.topology()
```

If no position is specified, then the 0-end will be connected to the 1-end as in the example.

Example

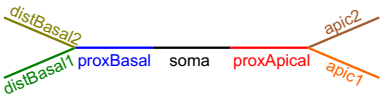
Python script:

```
from neuron import h

# define sections
soma = h.Section(name='soma')
papic = h.Section(name='proxApical')
apic1 = h.Section(name='apic1')
apic2 = h.Section(name='apic2')
pb = h.Section(name='proxBasal')
db1 = h.Section(name='distBasal1')
db2 = h.Section(name='distBasal2')

# connect them
papic.connect(soma)
pb.connect(soma(0))
apic1.connect(papic)
apic2.connect(papic)
db1.connect(pb)
db2.connect(pb)

# list topology
h.topology()
```

Output:

```
|-|       soma(0-1)
  `|        proxApical(0-1)
    `|        apic1(0-1)
    `|        apic2(0-1)
 `|       proxBasal(0-1)
  `|        distBasal1(0-1)
  `|        distBasal2(0-1)
```

Morphology:



Reduce work by writing functions

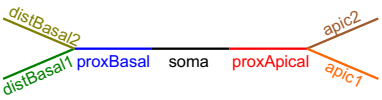
Python script:

```
from neuron import h

# helper functions
def sections(*names):
    secs = [h.Section(name=n)
            for n in names]
    return tuple(secs)

def connect(connections):
    for parent in connections:
        for child in connections[parent]:
            child.connect(parent)

# define, connect, print
soma, papic, apic1, apic2, pb, db1, db2 = \
    sections('soma', 'proxApical', 'apic1',
             'apic2', 'proxBasal',
             'distBasal1', 'distBasal2')

connect({soma: [papic],
         papic: [apic1, apic2],
         pb: [db1, db2]})
pb.connect(soma(0))

h.topology()
```

Output:

```
|-|       soma(0-1)
  `|        proxApical(0-1)
    `|        apic1(0-1)
    `|        apic2(0-1)
 `|       proxBasal(0-1)
  `|        distBasal1(0-1)
  `|        distBasal2(0-1)
```

Morphology:



Length and diameter

Set a section's length (in μm) with `.L` and diameter (in μm) with `.diam`:

```
sec.L = 20
sec.diam = 2
```

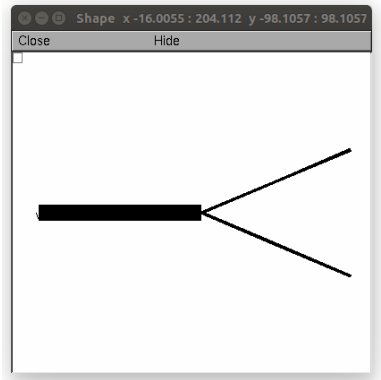
Note: Diameter need not be constant; it can be set per segment.

To specify the $(x, y, z; d)$ coordinates that a section passes through, use `h.pt3dadd`.

Warning: the default diameter is based on a squid giant axon and is not appropriate for modelling mammalian cells.

Viewing the morphology with h.PlotShape

```
from neuron import h, gui

main = h.Section()
dend1 = h.Section()
dend2 = h.Section()

dend1.connect(main)
dend2.connect(main)

main.diam = 10
dend1.diam = 2
dend2.diam = 2

ps = h.PlotShape()
# use 1 instead of 0 to hide diams
ps.show(0)
```



Note: `PlotShape` can also be used to see the distribution of a parameter or calculated variable. To save the image in plot shape ps use `ps.printfile('filename.eps')`

Setting and reading parameters

In NEURON's coordinate system, each section has normalized positions from 0 to 1.
To read the value of a parameter defined by a range variable at a given normalized position use: section(x).MECHANISM.VARNAME
e.g.

```
gkbar = apical(0.2).hh.gkbar
```

Setting variables works the same way:

```
apical(0.2).hh.gkbar = 0.037
```

To specify how many evenly-sized pieces (segments) a section should be broken into (each potentially with their own value for range variables), use `section.nseg`:

```
apical.nseg = 11
```


Distributed mechanisms

Use `.insert` to insert a distributed mechanism into a section. e.g.

```
axon.insert('hh')
```

Point processes

To insert a point process, specify the segment when creating it, and save the return value. e.g.

```
pp = h.IClamp(soma(0.5))
```

To find the segment containing a point process pp, use

```
seg = pp.get_segment()
```

The section is then `seg.sec` and the normalized position is `seg.x`.

The point process is removed when no variables refer to it.

Use `List` to find out how many point processes of a given type have been defined:

```
all_iclamp = h.List('IClamp')
print ('Number of IClamps:')
print (all_iclamp.count())
```


Running simulations

Basics

To initialize a simulation to -65 mV:

```
h.finitialize(-65)
```

To run a simulation until $t = 50$ ms:

```
h.continuerun(50)
```

Additional `h.continuerun` calls will continue from the last time.

Ways to improve accuracy

Reduce time steps via, e.g. `h.dt = 0.01`
Enable variable step (allows error control): `h.CVode().active(1)`
Increase the discretization resolution: `sec.nseg = 11`

To increase nseg for all sections:

```
for sec in h.allsec():  sec.nseg = sec.nseg * 3
```


Recording data

To see how a variable changes over time, create a `Vector` to store the time course:

```
data = h.Vector()
```

and do a `.record` with the last part of the name prefixed by `_ref_`.

e.g. to record `soma(0.3).ina`, use

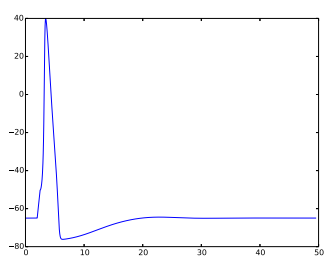
```
data.record(soma(0.3)._ref_ina)
```

Tips

- Be sure to also record `h._ref_t` to know the corresponding times.
- `.record` must be called before `h.finitialize()`.

Example: Hodgkin-Huxley

```
from neuron import h, gui
from matplotlib import pyplot

# morphology and dynamics
soma = h.Section()
soma.insert('hh')

# current clamp
i = h.IClamp(soma(0.5))
i.delay = 2 # ms
i.dur = 0.5 # ms
i.amp = 50

# recording
t = h.Vector()
v = h.Vector()
t.record(h._ref_t)
v.record(soma(0.5)._ref_v)

# simulation
h.finitialize()
h.continuerun(49.5)

# plotting
pyplot.plot(t.as_numpy(), v.as_numpy())
pyplot.show()
```



Storing data to CSV to share with other tools

The CSV format is widely supported by mathematics, statistics, and spreadsheet programs and offers an easy way to pass data back-and-forth between them and NEURON.

In Python, we can use the `csv` module to read and write csv files.

Adding the following code after the `continuerun` in the example will create a file `data.csv` containing the course data.

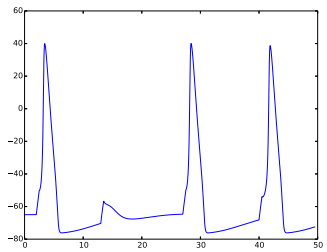
```
import csv
with open('data.csv', 'wb') as f:
    csv.writer(f).writerows(zip(*(t, v)))
```

Each row in the file corresponds to one time point. The first column contains t values; the second contains v values. Additional columns can be stored by adding them after the `t, v`.

For more complicated data storage needs, consider the `pandas` or `h5py` modules. Unlike `csv`, these must be installed separately.

A spike occurs whenever V_m crosses some threshold (e.g. 0 mV). Python can easily find all spike times. Only changes from the previous example are highlighted.

```
from neuron import h, gui
from matplotlib import pyplot
soma = h.Section()
soma.insert('hh')
# current clamps
iclamps = []
for t in [2, 13, 27, 40]:
    i = h.IClamp(soma(0.5))
    i.delay = t # ms
    i.dur = 0.5 # ms
    i.amp = 50
    iclamps.append(i)
# recording
t = h.Vector(); v = h.Vector()
t.record(h._ref_t)
v.record(soma(0.5)._ref_v)
# simulation
h.finitialize()
h.continuerun(49.5)
# compute spike times
st = [t[j] for j in range(len(v) - 1)
      if v[j] <= 0 and v[j + 1] > 0]
print ('spike times:'); print (st)
# plotting
pyplot.plot(t.as_numpy(), v.as_numpy())
pyplot.show()
```



The console displays:

```
spike times:
[3.1750000000000114, 28.149999999998936,
41.6250000000009]
```

That is, the cell spiked at: 3.175 ms, 28.150 ms, and 41.625 ms.

Interspike intervals (ISIs) are the delays between spikes; that is, they are the differences between consecutive spike times.

To display ISIs for the previous example, we add the lines:

```
isis = [next - last for next, last in zip(st[1:], st[:-1])]
print ('ISIs:'); print (isis)
```

The result:

```
[24.974999999998925, 13.475000000001966]
```

That is, the delays between spikes were 24.975 ms and 13.475 ms.

HOC was NEURON's original programming language. There are many valuable HOC functions in ModelDB and elsewhere. Python scripts can easily use these functions via a two step process:

Load the HOC library, here `libraryname.hoc`:

```
h.load_file('libraryname.hoc')
```

Invoke the HOC function, here `test` by proceeding its name with an `h.` and passing the appropriate arguments:

```
h.test(13, 172.2)
```

SES files created by saving the session are written in HOC and may be loaded the same as with any other HOC file.

Example

HOC code: `myneuron.hoc`

```
// define a cell
create soma, apic, basal
soma {
    connect apic(0), 1
    connect basal(0), 0
    L = 20
    diam = 20
}
```

Python script:

```
from neuron import h
h.load_file('myneuron.hoc')
h.topology()
```

Running the Python script shows:

```
|-|       soma(0-1)
  `|        apic(0-1)
 `|       basal(0-1)
```


Section-dependent functions

Some NEURON functions depend on the section; specify that with a `sec=` argument.

Example: calculating path distance

For example, `h.distance` is used to calculate the path distance in μm between two points along the neuron. To set a reference point at the center (0.5) of the soma, use:

```
h.distance(0, 0.5, sec=soma)
```

The distance from the reference point to the 1 end of apic is

```
h.distance(1, sec=apic)
```


Advanced topics

Version control: git

Why use version control?

- **Protects against losing working code**: if something used to work but no longer does, you can test previous versions to identify what change caused the error.
- **Provides a record of script history**: authorship, changes, . . .
- **Promotes collaboration**: provides tools to combine changes made independently on different copies of the code.

Version control: git basics

Setup

```
git init
```

Declare files to be tracked

```
git add FILENAME
```

Commit a version (so can return to it later)

```
git commit -a
```

Return to the version of FILENAME from 2 commits ago

```
git checkout HEAD~2 FILENAME
```


Version control: git

View list of changes

```
git log
```

Remove a file from tracking

```
git rm FILENAME
```

Rename a tracked file

```
git mv OLDNAME NEWNAME
```


Version control: git and remote servers

`git` (and mercurial) is a distributed version control system, designed to allow you to collaborate with others. You can use your own server or a public one like github or bitbucket.

Download from a server

```
git clone http://URL.git
```

Get changes from server and merge with local changes

```
git pull
```

Sync local, committed changes to the server

```
git push
```


Version control: syncing data with code

One simple way to ensure you always know what version of the code generated your data is to include the git hash in the filename. The following function can help:

```
def git_hash():
    import subprocess
    suffix = ''
    if subprocess.check_output(['git', 'diff']):
        suffix = '+'
    return '%s%s' % (subprocess.check_output([
        'git', 'log', '-1', '--pretty=format:%h']),
        suffix)
```

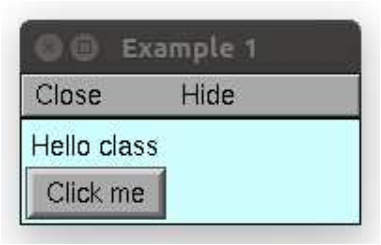
Then, for example, save matplotlib graphics with:

```
    pyplot.savefig('filename_' + git_hash() + '.pdf')
```


Making your own graphical interface

- To ensure your GUI responds to user input, be sure to: `from neuron import gui`
- Place basic widgets (text, buttons, checkboxes, . . .) in an `h.xpanel`.

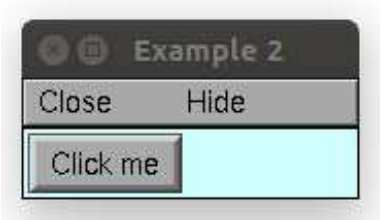
```
from neuron import h, gui

h.xpanel('Example 1')
h.xlabel('Hello class')
h.xbutton('Click me')
h.xpanel()
```



Button actions

To perform an action when a button is pressed, write it as a function, and then pass the function to `h.xbutton`.

```
from neuron import h, gui

def say_hello():
    print 'hello!'

h.xpanel('Example 2')
h.xbutton('Click me',
          say_hello)
h.xpanel()
```



Pressing the button displays:

```
hello!
```

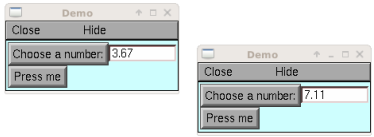
Pressing the button twice:

```
hello!
hello!
```


Number fields and classes

Place your GUI commands in a `class` to allow independent reuse.

```
from neuron import h, gui
class Demo:
    def __init__(self):
        self.value = 7.18
        h.xpanel('Demo')
        h.xvalue('Choose a number:',
            (self, 'value'))
        h.xbutton('Press me',
            self.print_value)
        h.xpanel()
    def print_value(self):
        print ('You chose:')
        print (self.value)

# make two demos
d1 = Demo()
d2 = Demo()
```



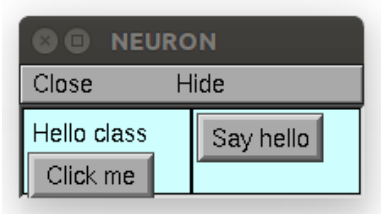
Click "Press me" on the left window and then on the right window displays:

```
You chose:
3.67
You chose:
7.11
```


Layout: HBox and VBox

Combine windows horizontally with HBox and vertically with VBox.

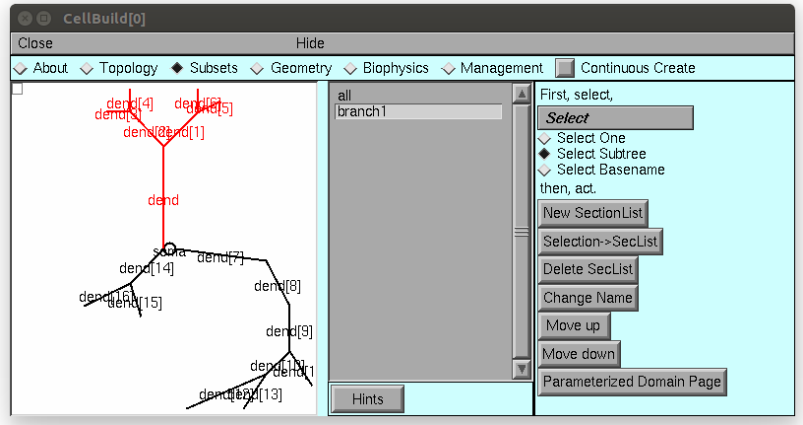
```
from neuron import h, gui
hbox = h.HBox()
hbox.intercept(1)
h.xpanel('Example 1')
h.xlabel('Hello class')
h.xbutton('Click me')
h.xpanel()
h.xpanel('Example 3')
h.xbutton('Say hello')
h.xpanel()
h.xpanel()
hbox.intercept(0)
hbox.map()
```



Note: HBox and VBox can contain: H/VBox, Deck, xpanel, Graph, . . .

Layout: HBox and VBox

Complicated layouts can be constructed using nested VBox and HBox objects:



For more information

For more background and a step-by-step guide to creating a network model, see the NEURON + Python tutorial at:

http://neuron.yale.edu/neuron/static/docs/neuronpython/index.html

Don't reinvent the brain

Using ModelDB and other archives for your research

Robert A. McDougal

Yale School of Medicine

16 October 2015

General issues

On reproducibility

"Non-reproducible single occurrences are of no significance to science."

– Karl Popper in *The logic of scientific discovery*, 1959.

What is needed for a model to be reproducible?

Model

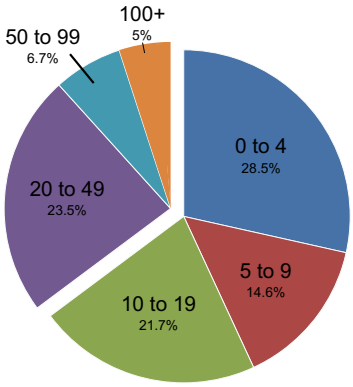
- an approximation of the system of interest
  e.g. a model organism or a complete statement of the properties of the model in mathematical or computable form

Experimental protocol

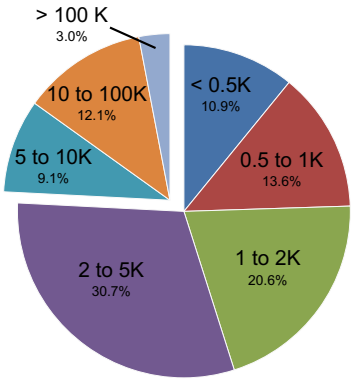
- what was done with the model to produce the data

Science builds upon previous work; in order to do that, the previous work needs to be reproducible.

Models are complicated



Files per Model



File Size

- **38.5%** of ModelDB models have **over 20 files**; **24.2%** of files are **over 5K**.
- It is often hard to fully describe this complexity in a paper.
- Any bugs, typos, errors, or omissions might completely change the dynamics.

Distributions from ModelDB, Fall 2013. A model was counted as having 0 files if it was not hosted on ModelDB.

Model sharing helps, but only reuse what you understand

The easiest way to replicate someone else's results – a first step toward building on them – is to get their model code from a repository such as ModelDB.

But beware:

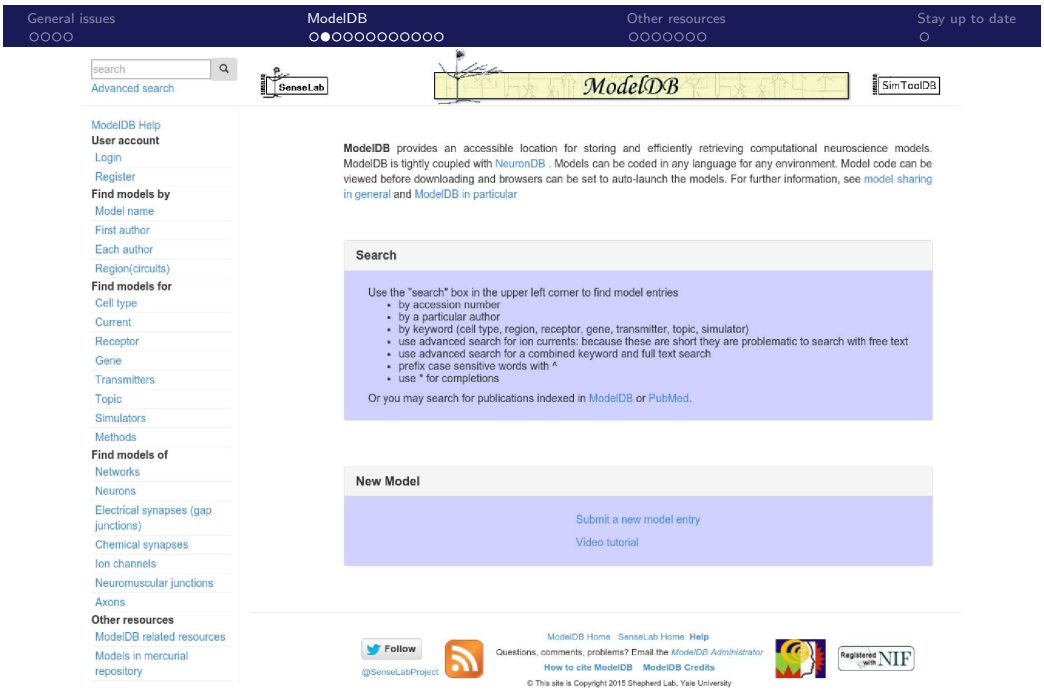
- They may be solving a different problem than you (with respect to species, temperature, age, etc).
- Their code may have bugs.

To reduce the risk of problems:

- **Read** the associated paper.
- **Compare** the model and results to other similar models.
- **Examine** the model with ModelView and/or psection.
- **Test** ion channels individually.
- **Collaborate** with an experimentalist.

ModelDB

Part of the SenseLab Project



modeldb.yale.edu

What is in ModelDB?

Models for:

- 176 cell types
- 19+ species
- 52 ion channels, pumps, etc
- 129 topics (Alzheimer's, STDP, etc)

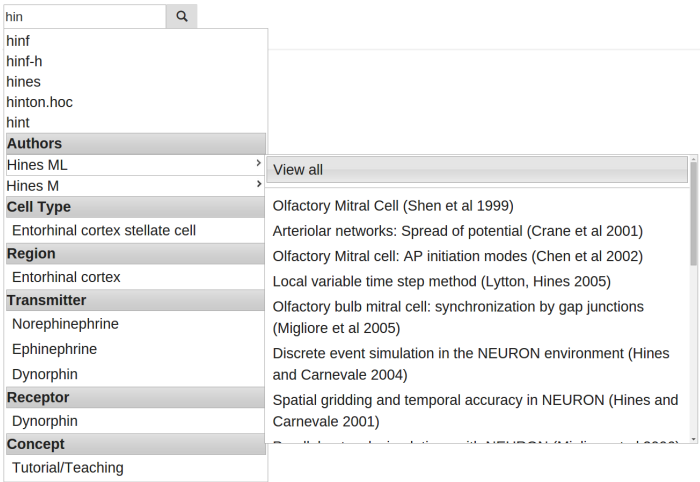
1052 published models from 70+ simulators

- 509 NEURON models

Numbers are as of September 25, 2015

General issues | ModelDB | Other resources | Stay up to date

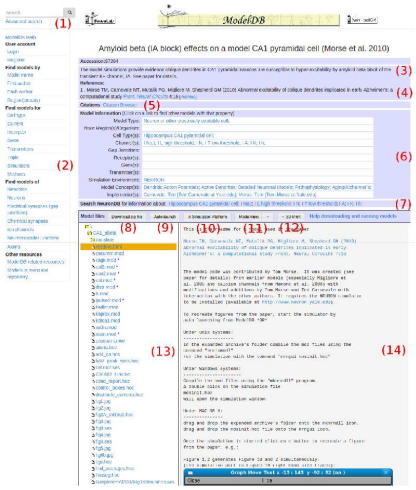
Finding models



- **Search box** on the top-left of every page.
- Do **full text** or **attribute** searches.
- Word completions (based on ModelDB entries not English) and attribute results **updated as you type**.
- **Advanced search** and **browsing** are also available.

General issues | ModelDB | Other resources | Stay up to date

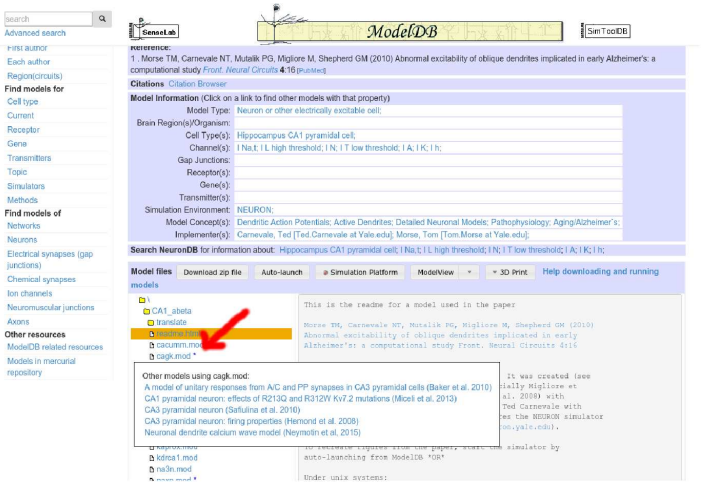
Anatomy of a ShowModel page



(1) Search models.
(2) Browse models.
(3) Description of model.
(4) Paper(s) describing or using model.
(5) Find models and papers cited by this model's paper, or that cite this model.
(6) Searchable metadata.
(7) Links to NeuronDB (channel distributions etc within cell types).
(8) Link to download the entire simulation.
(9) Auto-launch a NEURON simulation (requires browser configuration).
(10) Simulation platform (5 minutes of remote desktop access to experiment with the model).
(11) ModelView: visualize model structure.
(12) 3D printable versions of cells from the model (in 3DModelDB).
(13) Directory browser, showing model files.
(14) View pane for the currently selected file.

General issues | ModelDB | Other resources | Stay up to date

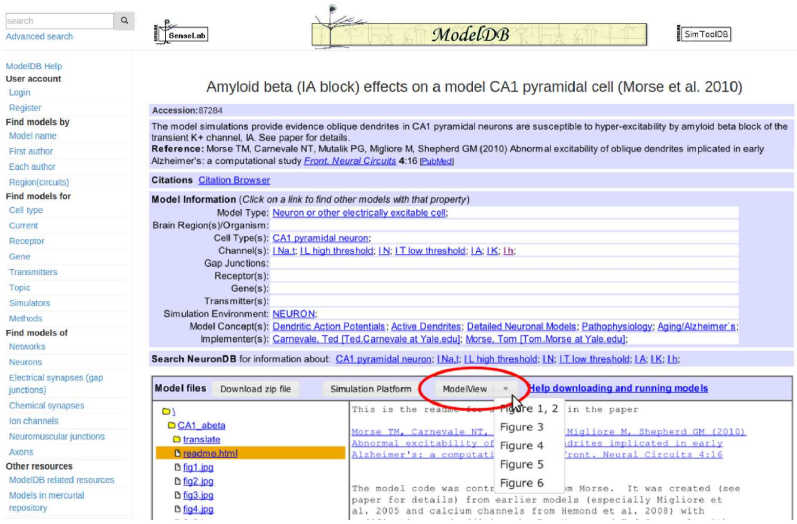
Identifying existing reuse



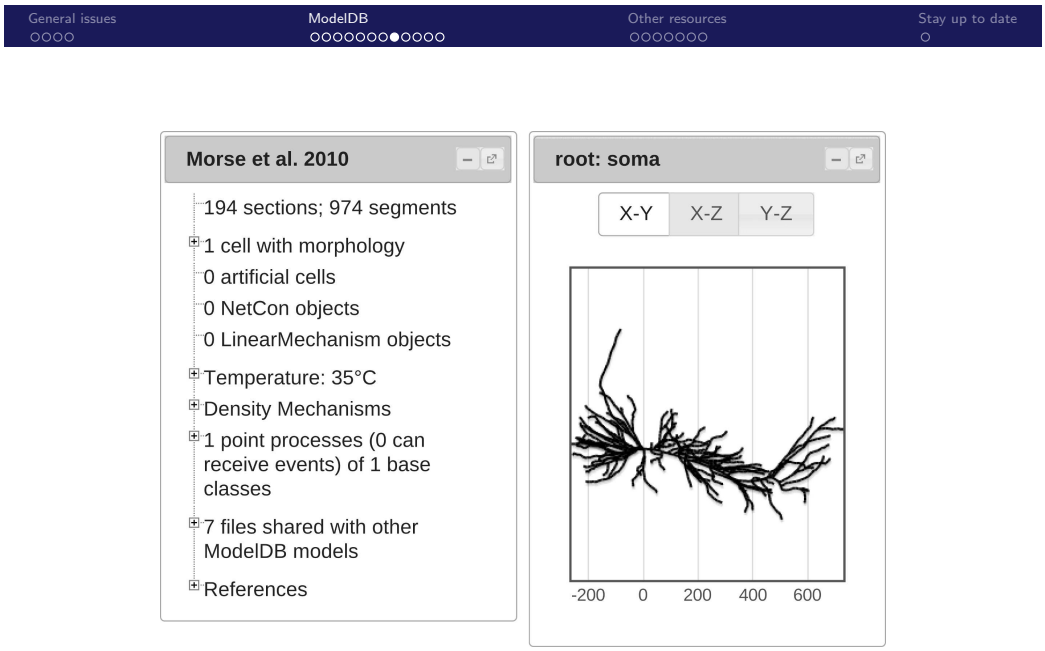
Asterisks in the file browser indicate that the file is reused in other models; click the asterisk to see a list of the other models.

General issues | ModelDB | Other resources | Stay up to date

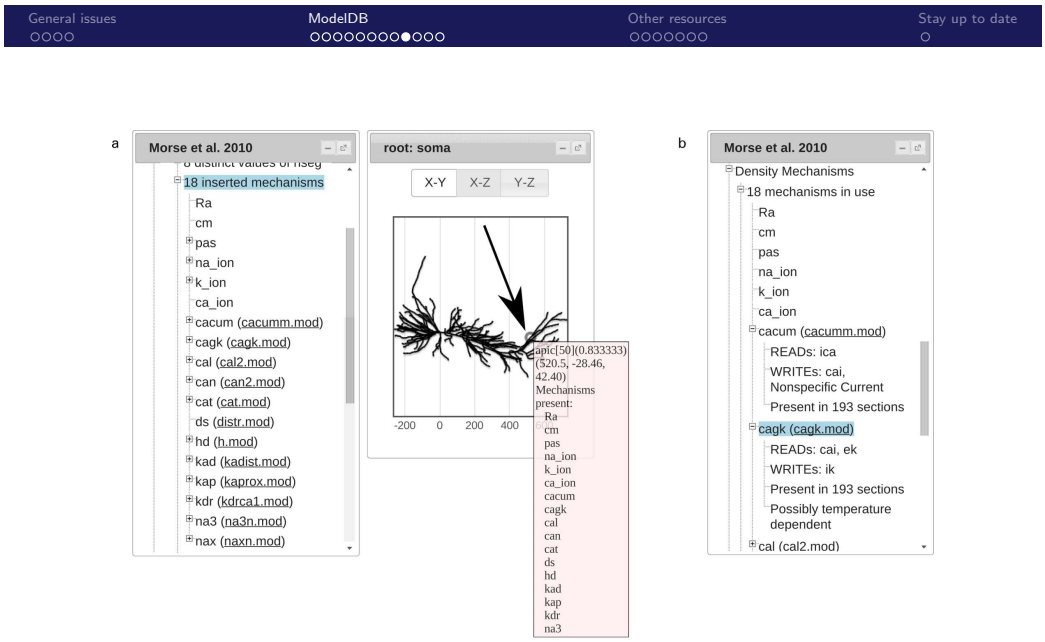
ModelView



Adapted from McDougal et al, *Neuroinformatics* 2015 (online ahead of print)

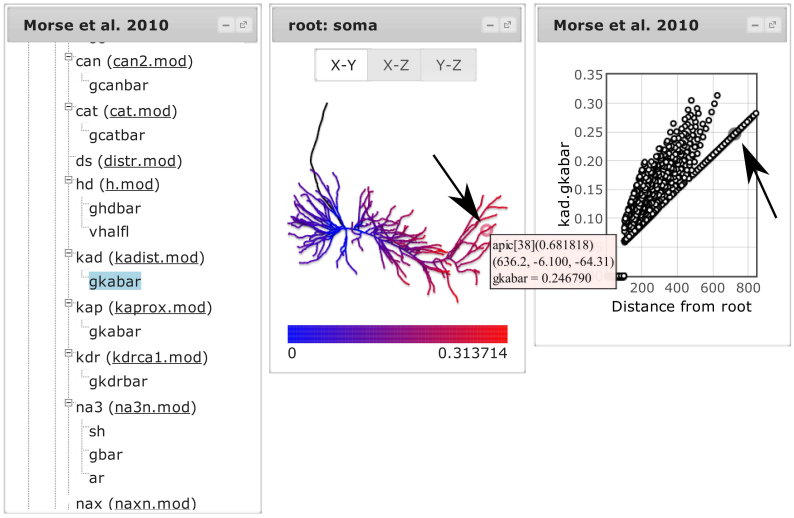


McDougal et al, *Neuroinformatics* 2015 (online ahead of print)



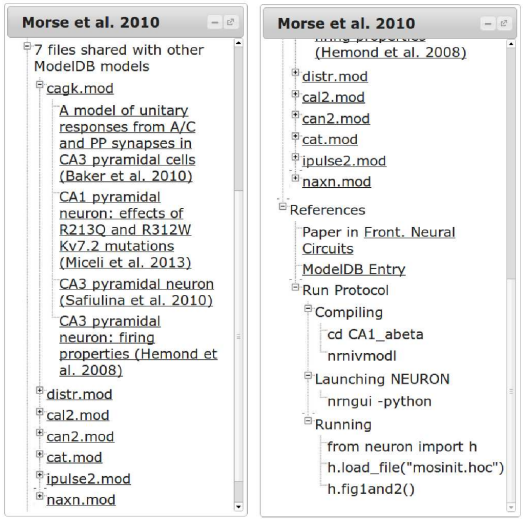
McDougal et al, *Neuroinformatics* 2015 (online ahead of print)

General issues ○○○○ ModelDB ○○○○○○○○○●○○ Other resources ○○○○○○○ Stay up to date ○



McDougal et al, *Neuroinformatics* 2015 (online ahead of print)

General issues ○○○○ ModelDB ○○○○○○○○○○●○ Other resources ○○○○○○○ Stay up to date ○



McDougal et al, *Neuroinformatics* 2015 (online ahead of print)

How do people use ModelDB?

- Find a model described in a paper, download it, and experiment to understand the model's predictions.
- Find a model described in a paper. Use ModelView to understand the model's structure.
- Locate models and modeling papers on a given topic.
- Locate model components (e.g. L-type calcium channel) for potential reuse.
- Search for simulator keywords (e.g. FInitializeHandler) to find examples of how to use them.

You can help by sharing your model code on ModelDB after publication.

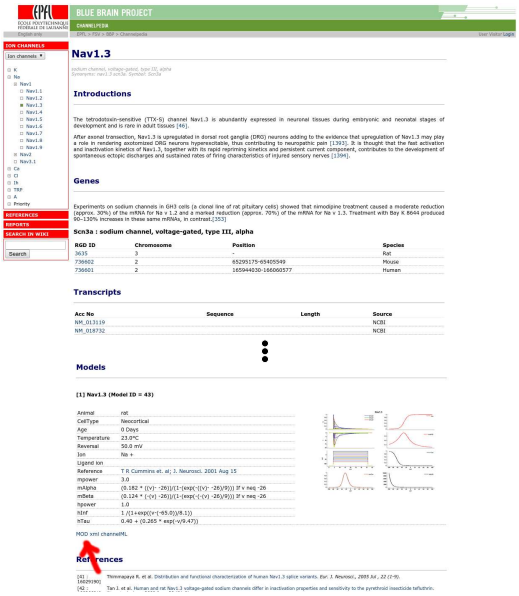
Other resources

NeuroMorpho.Org

Tools ▶ Miscellaneous ▶ Import 3D

- NeuroMorpho.Org is home to 31,982 reconstructed neurons from 140 cell types and 24 species as of September 24, 2015.
- Warning: not every morphology was reconstructed with the intent of being in a simulation. Before using: rotate to check for z-axis errors, check to make sure the diameters are not all equal.
- Use the Import 3D tool to import morphologies into NEURON. For details, see: neuron.yale.edu/neuron/docs/import3d

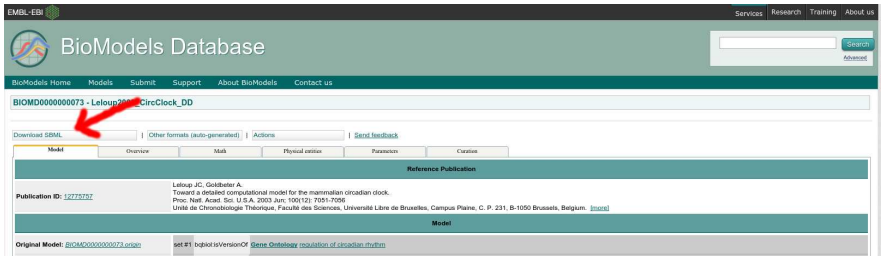
Channelpedia (Channelpedia.epfl.ch)



- Home to information about ion channels.
- Many channels have one or more associated models (e.g. different species or cell types); all are downloadable as MOD files.
- Shows gating variable and channel response to voltage clamp for each model.

General issues ○○○○ ModelDB ○○○○○○○○○○○○ Other resources ○○○●○○○ Stay up to date ○

Biomodels (www.ebi.ac.uk/biomodels-main)




```
jnml BIOMD0000000073_LEMS.xml -neuron
```

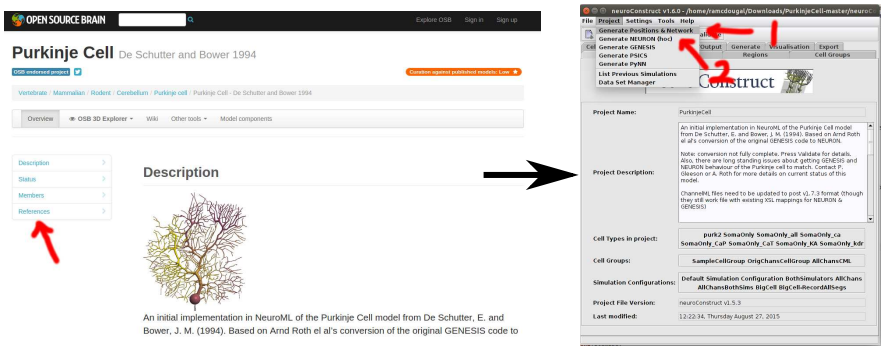
Biomodels model (SBML) → LEMS model → MOD file

```
jnml -sbml-import BIOMD0000000073.xml 1000 5
```

- Biomodels is a systems biology model repository.
- Models are in SBML but can be converted to MOD files via e.g. jNeuroML (github.com/NeuroML/jNeuroML). Test converted models before using in a larger model. Edits will likely be necessary to get them to interoperate with other mechanisms.
- A native SBML importer for NEURON's rxd module is under development.

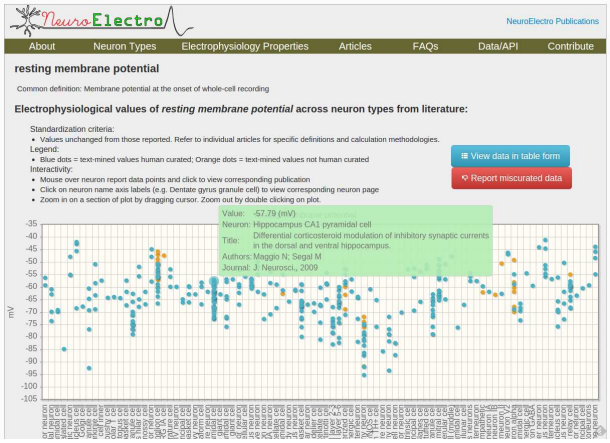
General issues ○○○○ ModelDB ○○○○○○○○○○○○ Other resources ○○○○●○○ Stay up to date ○

Open Source Brain (OpenSourceBrain.org)



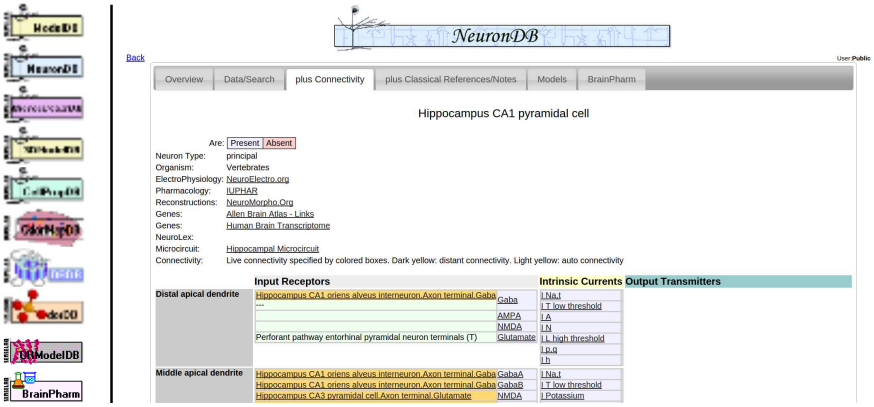
- Open Source Brain promotes collaborative model development via github.
- Models are typically in NeuroML or neuroConstruct format; neuroConstruct (neuroConstruct.org) converts both formats to NEURON.
- The conversion process places different ion channels in different MOD files, which allows extracting model components.

NeuroElectro (NeuroElectro.org)



- NeuroElectro archives experimentally measured electrophysiology values for different cell types; it shows the spread and allows comparing values across different cell types.
- Read the paper associated with a value to understand: species, experimental conditions, etc.

SenseLab (senselab.med.yale.edu)



- SenseLab is a suite of 10 interconnected databases (listed at left).
- ModelDB and NeuronDB (at right) are the most useful for modeling.
- NeuronDB shows what channels are present and the inputs and outputs *by cell region* (e.g. distal apical dendrite vs proximal apical dendrite).

Stay up to date

Twitter

Many repositories announce new developments on Twitter, including:

- SenseLab (including ModelDB): @SenseLabProject
- Open Source Brain: @OSBTeam
- NeuroMorpho.Org: @NeuroMorphoOrg

Reaction-Diffusion in the NEURON Simulator

Robert A. McDougal

Yale School of Medicine

16 October 2014

Getting Started

What is a reaction-diffusion system?

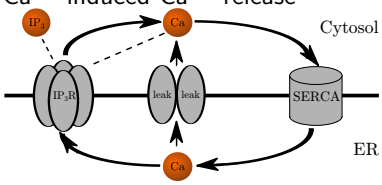
"Reaction–diffusion systems are mathematical models which explain how the **concentration** of one or more substances distributed in space changes under the influence of two processes: **local chemical reactions** in which the substances are transformed into each other, and **diffusion** which causes the substances to spread out over a surface in space."[1]

[1] http://en.wikipedia.org/wiki/Reaction%E2%80%93diffusion_system

Examples

- Pure Diffusion
- Protein Degradation
- Buffering

- Circadian Oscillator
- Ca^{2+}-induced Ca^{2+} release



What does the rxd module do?

Reduces typing

- **In 2 lines:** declare a domain, then declare a molecule, allowing it to diffuse and respond to flux from ion channels.

```
all = rxd.Region(h.allsec(), nrn_region='i')
ca = rxd.Species(all, name='ca', d=1, charge=2)
```

- **Reduces** the risk for **errors** from typos or misunderstandings.

Allows arbitrary domains

NEURON traditionally only identified concentrations just inside and just outside the plasma membrane. The rxd module allows you to **declare your own regions** of interest (e.g. ER, mitochondria, etc).

The three questions

- **Where** do the dynamics occur?
  - Cytosol
  - Endoplasmic Reticulum
  - Mitochondria
  - Extracellular Space
- **Who** are the actors?
  - Ions
  - Proteins
- **What** are the reactions?
  - Buffering
  - Degradation
  - Phosphorylation

Declare a region with rxd.Region

Basic Usage

cyt = rxd.Region(seclist)
seclist may be any iterable of sections; e.g. a SectionList or a Python list.

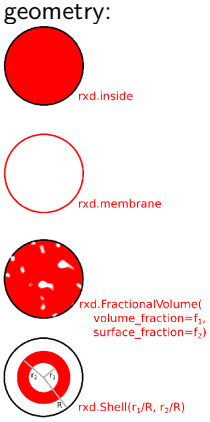
Identify with a standard region

cyt = rxd.Region(seclist, nrn_region='i')
nrn_region may be i or o, corresponding to the locations of e.g. nai vs nao.

Specify the cross-sectional shape

cyt = rxd.Region(seclist,
geometry=rxd.Shell(0.5, 1))
The default geometry is rxd.inside.
The geometry and nrn_region arguments may both be specified.

geometry:



Adapted from:
McDougal et al 2013.

rxd.Region tips

Specify nrn_region if concentrations interact with NMODL

If NMODL mechanisms (ion channels, point processes, etc) depend on or affect the concentration of a species living in a given region, that region must declare a nrn_region (typically 'i').

To declare a region that exists on all sections

r = rxd.Region(h.allsec())

Use list comprehensions to select sections

r = rxd.Region([sec for sec in h.allsec() if 'apical' in sec.name()])

Declare proteins and ions with rxd.Species

Basic usage

protein = rxd.Species(region, d=16)

d is the diffusion constant in μm^2/ms. region is an rxd.Region or an iterable of rxd.Region objects.

Initial conditions

protein = rxd.Species(region, initial=value)

value is in mM. It may be a constant or a function of the node.

Connecting with HOC

ca = rxd.Species(region, name='ca', charge=2)

If the nrn_region of region is "i", the concentrations of this species will be stored in cai, and its concentrations will be affected by ica.

Specifying dynamics: rxd.Reaction

Mass-action kinetics

$\mathrm{ca} + \mathrm{buffer} \underset{kb}{\overset{kf}{\longleftrightarrow}} \mathrm{cabuffer}$

buffering = rxd.Reaction(ca + buffer, cabuffer, kf, kb)

kf is the forward reaction rate, kb is the backward reaction rate. kb may be omitted if the reaction is unidirectional.
In a mass-action reaction, the reaction rate is proportional to the product of the concentrations of the reactants.

Repeated reactants

$2\mathrm{H} + \mathrm{O} \underset{kb}{\overset{kf}{\longleftrightarrow}} \mathrm{H2O}$

water_reaction = rxd.Reaction(2 * H + O, H2O, kf, kb)

Arbitrary reaction formula, e.g. Hill dynamics

$\mathrm{a} + \mathrm{b} \longrightarrow \mathrm{c}$

hill_reaction = rxd.Reaction(a + b, c, a ^ 2 / (a ^ 2 + k ^ 2),
mass_action=False)

Hill dynamics are often used to model cooperative reactions.

rxd.Rate and rxd.MultiCompartmentReaction

rxd.Rate

Use rxd.Rate to specify an explicit contribution to the rate of change of some concentration or state variable.

```
ip3degradation = rxd.Rate(ip3, -k * ip3)
```

rxd.MultiCompartmentReaction

Use rxd.MultiCompartmentReaction when the dynamics span multiple regions; e.g. a pump or channel.

```
ip3r = rxd.MultiCompartmentReaction(ca[er], ca[cyt], kf, kb,
                                    membrane=cyt_er_membrane)
```

The rate of these dynamics is proportional to the membrane area.

Manipulating nodes

Getting a list of nodes

- nodelist = protein.nodes

Filtering a list of nodes

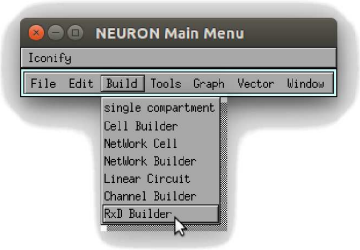
- nodelist2 = nodelist(region)
- nodelist2 = nodelist(0.5)
- nodelist2 = nodelist(section)(region)(0.5)

Other operations

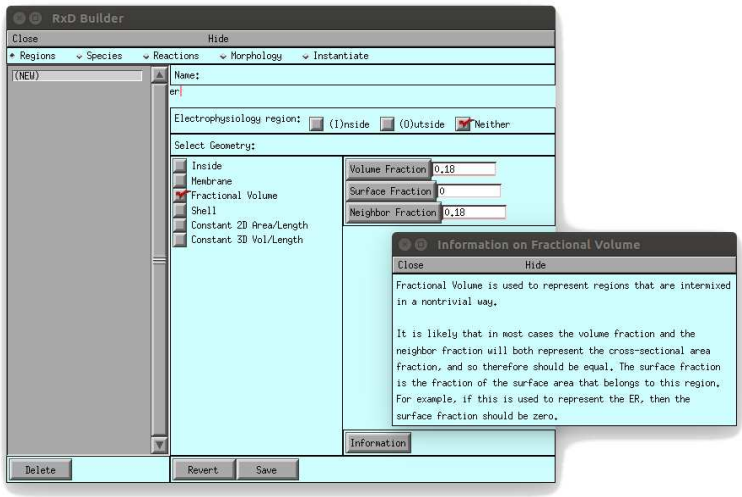
- nodelist.concentration = value
- values = nodelist.concentration
- surface_areas = nodelist.surface_area
- volumes = nodelist.volume
- node = nodelist[0]

GUI

Reaction-diffusion dynamics can also be specified via the GUI. This option appears only when rxd is supported in your install (Python and scipy must be available).



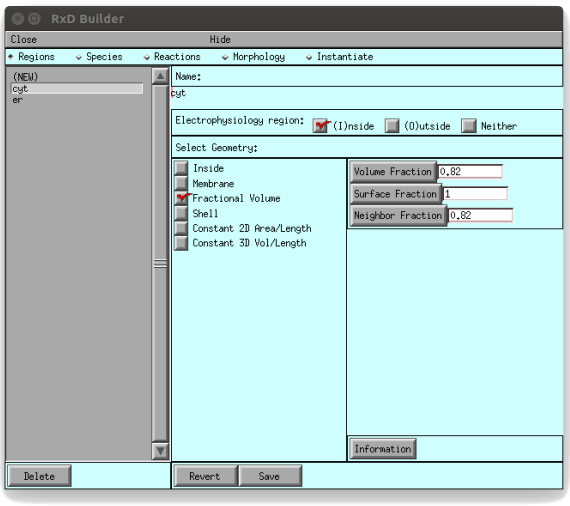
GUI



Getting Started Examples Notes 3D Simulations References

How do I use the rxd module?

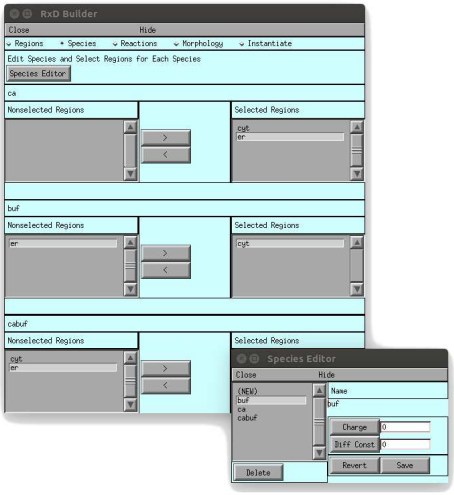
GUI



Getting Started Examples Notes 3D Simulations References

How do I use the rxd module?

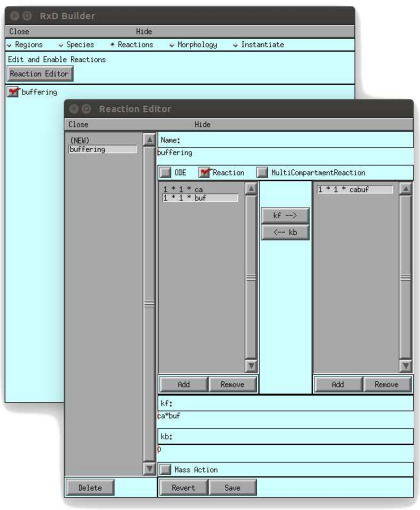
GUI



Getting Started Examples Notes 3D Simulations References

How do I use the rxd module?

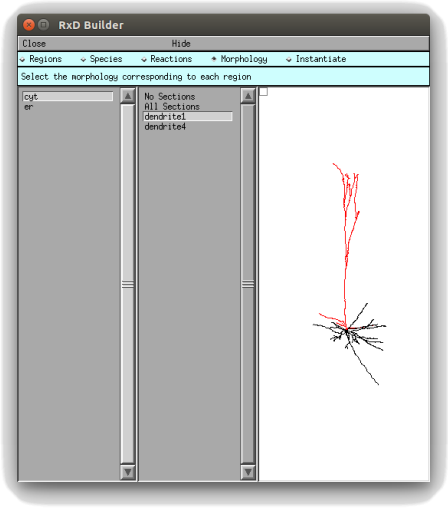
GUI



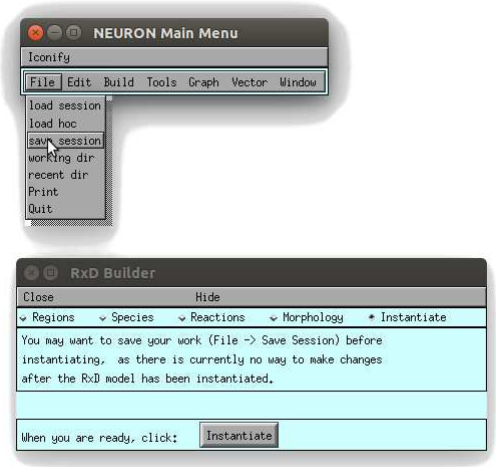
Getting Started Examples Notes 3D Simulations References

How do I use the rxd module?

GUI



GUI

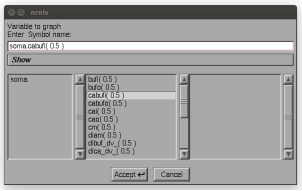


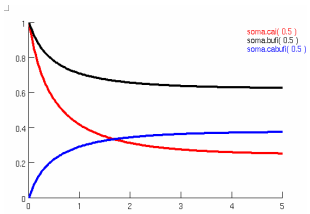
Example: Calcium buffering

```
from neuron import h, rxd, gui

h('create soma')
soma_region = rxd.Region([h.soma], nrn_region='i')

ca = rxd.Species(soma_region, initial=1,
                 name='ca', charge=2)
buf = rxd.Species(soma_region, initial=1,
                  name='buf')
cabuf = rxd.Species(soma_region, initial=0,
                    name='cabuf')

buffering = rxd.Reaction(2 * ca + buf, cabuf, 1, 0.1)
```

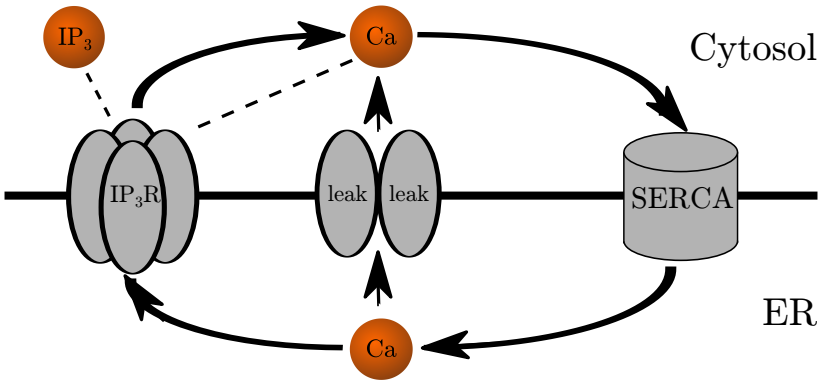
In this example, we suppose each buffer molecule binds two molecules of calcium. Other buffers have different properties.

Use the GUI to create a graph and run the simulation.





Example: Calcium wave dynamics



Wagner et al. 2004. *Cell Calcium*. DOI: 10.1016/j.ceca.2003.10.009
Neymotin et al. 2015. *Neural Computation*. DOI: 10.1162/NECO_a_00712
ModelDB: 168874

There are three regions:

Cytosol:

```
cyt_geom = rxd.FractionalVolume(0.83, surface_fraction=1)
cyt = rxd.Region(h.allsec(),
                 nrn_region='i',
                 geometry=cyt_geom)
```

Endoplasmic Reticulum (ER):

```
er = rxd.Region(h.allsec(),
                geometry=rxd.FractionalVolume(0.17))
```

The ER membrane:

```
er_membrane = rxd.Region(h.allsec(),
                         geometry=rxd.ScalableBorder(1))
```


There are two species:

Calcium:

```
def ca_init(node):
    return ca_cyt0 if node.region == cyt else ca_er0

ca = rxd.Species([cyt, er],
                 d=caDiff,
                 name='ca',
                 charge=2,
                 initial=ca_init)
```

Inositol trisphosphate (IP3):

```
ip3 = rxd.Species(cyt,
                  d=ip3Diff,
                  initial=ip3_init)
```


Each pump and channel corresponds to its own "reaction":

Leak:

```
leak = rxd.MultiCompartmentReaction(
    ca[er] <> ca[cyt],
    gleak,
    gleak,
    membrane=er_membrane)
```

SERCA:

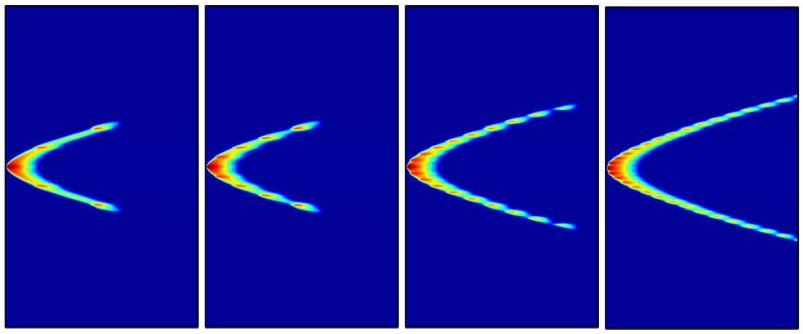
```
serca = rxd.MultiCompartmentReaction(
    ca[cyt] > ca[er],
    gserca * (ca[cyt])**2 / (Kserca**2 + (ca[cyt])**2),
    membrane=er_membrane,
    mass_action=False)
```


The IP_3R is more complicated because it is essentially a channel that opens and closes slowly as a function of calcium concentration.

IP_3R:

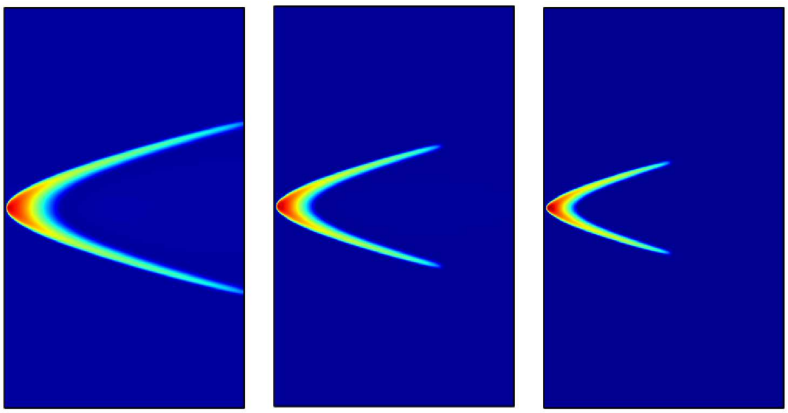
```
# slow gating due to calcium
ip3r_gate_state = rxd.State(er_membrane, initial=0.8)
h_gate = ip3r_gate_state[er_membrane]
ip3rg = rxd.Rate(h_gate,
                 (1. / (1 + ca[cyt] / scale) - h_gate) / tau)

# fast gating due to calcium and IP3
minf = ip3[cyt] * ca[cyt] / (ip3[cyt] + Kip3) / (ca[cyt] + Kact)

# same permeability for either direction of flow
k = gip3r[cyt] * (minf * h_gate) ** 3

# the actual channel
ip3r = rxd.MultiCompartmentReaction(
    ca[er] <> ca[cyt], k, k, membrane=er_membrane)
```


Reducing IP_3R spacing facilitates wave propagation



More closely spaced IP_3R $\longrightarrow$

Increasing SERCA activity weakens wave propagation



Increasing SERCA activity ⟶

Interacting with the rest of NEURON

node._ref_concentration or node._ref_value returns a pointer.

Recording traces

```
v = h.Vector()
v.record(ca.nodes[0]._ref_concentration)
```

Plotting

```
g = h.Graph()
g.addvar('ca[er][dend](0.5)',
                    ca.nodes(er)(dend)(0.5)[0]._ref_concentration)
h.graphList[0].append(g)
```


Tips

dir(·)

To find out what methods and properties are available, use dir:

```
dir(ca.nodes)
```

CVode and atol

NEURON's variable step solver has a default absolute tolerance of 0.001.

Since NEURON measures concentration in mM and many cell biology concentrations are in μM, this tolerance may be too high. Try lowering it:

```
h.CVode().atol(1e-8)
```


3D Simulations

The Third Dimension

Specifying 3D Simulations

Just add one line of code[2]:

```
rxd.set_solve_type(dimension=3)
all = rxd.Region(h.allsec())
ca = rxd.Species(all, d=1)
ca.initial = lambda node: 1 if node.x3d < 50 else 0
```

Plotting

Get the concentration values expressed on a regular 3D grid via nodelist.value_to_grid()

```
values = ca.nodes.value_to_grid()
```

Pass the result to a 3d volume plotter, such as Mayavi's VolumeSlicer:

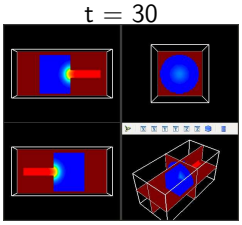
```
graph = VolumeSlicer(data=ca.nodes.value_to_grid())
graph.configure_traits()
```

[2] rxd.set_solve_type can optionally take a list of sections as its first argument; in that case only the specified sections will be simulated in three dimensions.

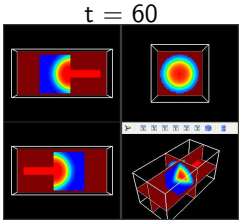
Example: wave curvature

```
from neuron import h, gui, rxd
import volume_slicer

sec1, sec2 = h.Section(), h.Section()
h.pt3dadd(2, 0, 0, 2, sec=sec1)
h.pt3dadd(9.9, 0, 0, 2, sec=sec1)
h.pt3dadd(10, 0, 0, 2, sec=sec1)
h.pt3dadd(10, 0, 0, 10, sec=sec2)
h.pt3dadd(18, 0, 0, 10, sec=sec2)

def do_init(node):
    return 1 if node.x3d < 8 else 0

all3d = rxd.Region(h.allsec(), dimension=3)
ca = rxd.Species(all3d, initial=do_init, d=0.05)
r = rxd.Rate(ca, -ca * (1 - ca) * (0.1 - ca))

def plot_it():
    graph = volume_slicer.VolumeSlicer(
        data=ca.nodes.value_to_grid(),
        vmin=0, vmax=1)
    graph.configure_traits()

h.finitialize()
for t in [30, 60]:
    h.continuerun(t)
    plot_it()
```

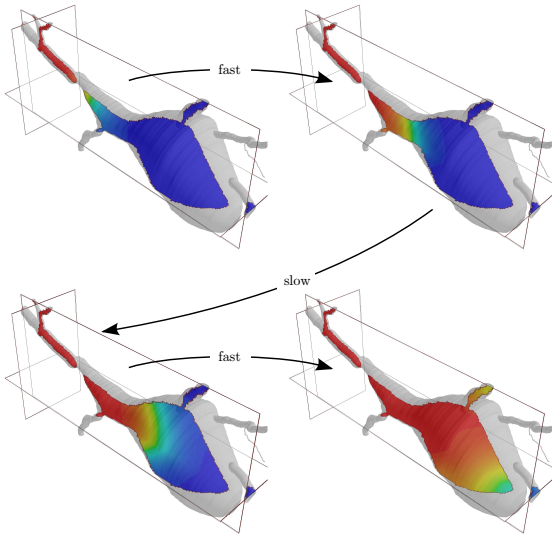
t = 30



t = 60



Example: wave curvature at soma entry



References

For more information, see:

Journal Articles on Reaction-Diffusion in NEURON

- McDougal, R. A., Hines, M. L., Lytton, W. W. (2013). Reaction-diffusion in the NEURON simulator. *Frontiers in Neuroinformatics*, 7.
- McDougal, R. A., Hines, M. L., Lytton, W. W. (2013). Water-tight membranes from neuronal morphology files. *Journal of Neuroscience Methods*, 220(2), 167-178.

Online Resources

- NEURON Forum
- Programmer's Reference
- NEURON Reaction-Diffusion Tutorials

Receipt

Received: $150

From:

For: Using the NEURON Simulation Environment
Held Oct. 16, 2015 in Chicago, IL
http://www.neuron.yale.edu/neuron/static/courses/chi2015/chi2015.html

By: N.T. Carnevale
Director, Using the NEURON Simulation Environment
203-494-7381
ted.carnevale@yale.edu

For deposit in: Yale University account "NNC--Fees"

Survey

We'd appreciate your frank opinions and suggestions to help us refine this course and design future offerings on related subjects.

Please score these	**. . . . .**	**according to this scale**	
Overall impression	_____	no opinion	0
Relevance to my research	_____	poor, not helpful	1
Didactic presentations	_____	fair	2
Written handouts	_____	good	3
Overhead transparencies	_____	excellent, very helpful	4
Computer projection	_____		
Classroom	_____		
Food	_____		

Best feature _____

Weakest feature _____

_____

Additional topics that should be covered, topics that should receive more or less coverage, or other suggestions for improvement.

_____

_____

_____

Circle one

Y N I would recommend this course to others who are interested in neural modeling.

Y N I have developed my own modeling software using a high-level language (FORTRAN, C/C++, Python etc.).

Y N I have created my own models using modeling software.

Which software? _____

My primary area of research interest is _____

To help us better meet the needs of NEURON users, please circle all platforms that you plan to use for modeling.

Hardware Mac PC Other _____

OS MacOS X Win Vista | 7 | 8 | 9 UNIX | Linux | OS X | BSD

If Linux, which distribution? _____

