Table of Contents and Schedule of Presentations

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

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

Saturday, 6/21
Morning session

Afternoon session

3:00	NTC	Elementary project management	19
3:30	NTC	Ball−stick model	19 *
5:00	End of afternoon session		

Sunday, 6/22
Morning session

Time	**Speaker**	**Title**	**Page**
9:00 AM	Q & A		
9:15	MLH	Numerical methods: accuracy, stability, speed	21
10:00	NTC	An outline for coding NEURON models	33
10:30	Coffee Break		
10:45	WWL	The hoc programming language	37 *
12:15	End of morning session		
12:30	Lunch		

Afternoon session

1:30	NTC	Model control: arbitrary forcing functions	51 *
2:30	NTC	Model control: simulation families	53 *
3:30	Coffee Break		
3:45	MLH	The Extracellular and Linear mechanisms	55 *
5:00	End of afternoon session		

Monday, 6/23
Morning session

Time	Speaker	Title	Page
9:00 AM	Q & A		
9:15	MLH	NMODL: the NEURON Model Description Language	57 *
10:30	Coffee Break		
10:45	MLH	Kinetic scheme: potassium channel	65 *
12:15	End of morning session		
12:30	Lunch		

Afternoon session

Time	Speaker	Title	Page
1:30	MLH	Optimizing a mechanism	75 *
2:30	NTC	ModelDB: a resource for reproducibility in computational neuroscience	83 *
3:15	Coffee Break		
3:30	MLH	Kinetic scheme: calcium pump	89 *
5:00	End of afternoon session		

Tuesday, 6/24
Morning session

Time	Speaker	Title	Page
9:00 AM	Q & A		
9:15	MLH	Variable time steps and parameter discontinuities: pulse stimulus	93 *

10:30	Coffee Break		
10:45	NTC	Networks 1: synapses	101 *
12:15	End of morning session		
12:30	Lunch		

Afternoon session

1:30	MLH	Networks 2: network construction	113 *
3:00	Coffee Break		
3:15	WWL	Networks 3: inhibitory synchronizing network	121 *
5:00	End of afternoon session		

Tuesday evening: “graduation dinner”

6:30	**TBA**

Wednesday, 6/25
Morning session

Time	Speaker	Title	Page
9:00 AM	Q & A		
9:15	MLH	The standard run system	143
10:30	Coffee Break		
10:45	NTC	Initialization	145 *
12:15	End of morning session		
12:30	Lunch		

Afternoon session

1:30	NTC	Working with morphometric data	153 *
2:00	NTC	NEURON's tools for analyzing electrotonus	155 *
3:15	Coffee Break		
3:30	Review discussion		
4:45	Evaluation form		
5:00	End of afternoon session		

Survey **last page**

Bound separately:

NEURON: Selected Preprints
Chapters from the latest draft of The NEURON Book

The NEURON Simulation Environment

M.L. Hines
N.T. Carnevale
W.W. Lytton
T.J. Sejnowski

Supported by NINDS

The What and the Why of Neural Modeling

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

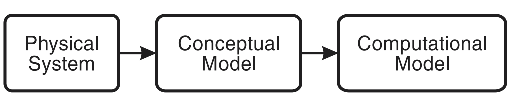
Empirically-based modeling is needed to test hypotheses about the mechanisms that govern these signals and how nervous system function emerges from the operation of these mechanisms.

Topics

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

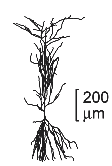
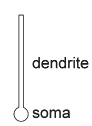
From Physical System to Simulation

Simulation for Insight



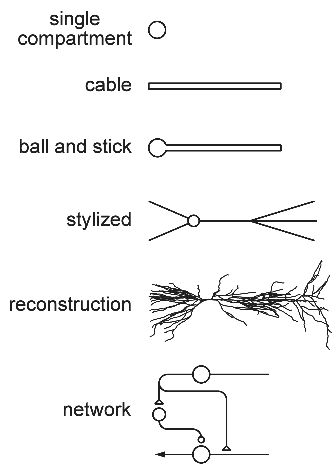
Conceptual Model
  a simplified representation
  of a complex system

Computational Model
  an accurate representation
  of this approximation

Physical System	Model	NEURON Simulation
		`create soma, dendrite` `connect dendrite(0), soma(1)`
CA1 pyramidal cell	ball and stick	`hoc` code

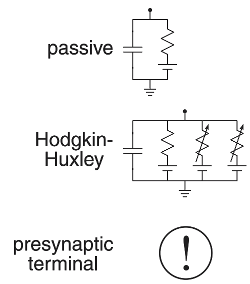
Hierarchies of Complexity

Structure



Hierarchies of Complexity

Mechanism

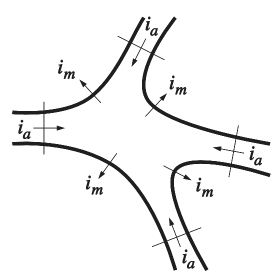


Fundamental Concepts in NEURON

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

Conservation of Charge

total current leaving = total current entering



$$Cm \frac{dV_m}{dt} + i_{ion} \quad \sum i_a$$

Example: Single Compartment

Lipid bilayer (no ionic channels)

Membrane with linear ion channels
(passive leak conductance).

```
oc> insert pas ↵
oc> e_pas = 0 ↵
```

Virtual molecular biology!

Project goals:

- Run simulation, changing stimulus intensity and duration
- Insert a membrane mechanism
- Adjust graphical displays of simulation results
- Adjust dt and Points plotted / ms

Fundamental Concepts

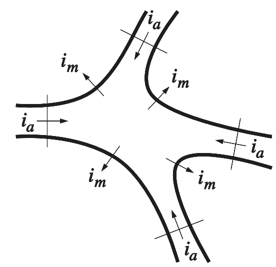
What equations are being solved?

How to separate the biology
from computational details?

"It's all about conceptual control . . . "

Conservation of Charge

total current leaving = total current entering



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

The Simulation 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*

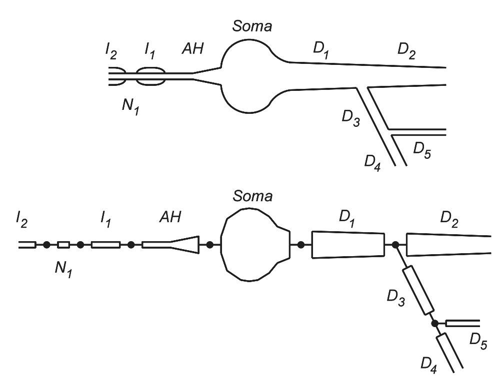
c_j membrane capacitance of compartment *j*

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

Separating Anatomy and Biophysics from Purely Numerical Issues

Section

a continuous length of unbranched cable



Syntax: `create sectionname`

Example: `create soma, dend[3]`

creates one section named `soma`
and an array of three sections named
`dend[0]`, `dend[1]`, and `dend[2]`

Assigning anatomical and biophysical attributes:

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

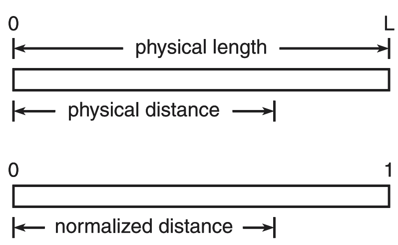

Range Variables

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

Range

The normalized position along the length of a section ("normalized arc length").

0 ≤ `range` ≤ 1 (any variable name can be used to represent range, e.g. `x`)



Syntax:

`sectionname.rangevar(range)`

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

Examples:

`dend.v(0.5)`

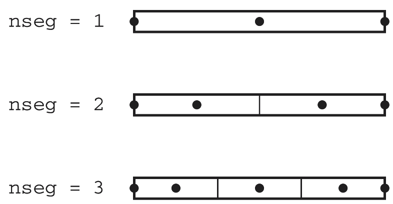
returns membrane potential at the middle of a section. Shortcut: `dend.v`

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

prints the physical distance and `v` at each point in `dend` where `v` was calculated. The results appear in the NEURON interpreter window.

nseg

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



Example: `axon {nseg = 3}`
or `axon.nseg = 3`

To test spatial resolution

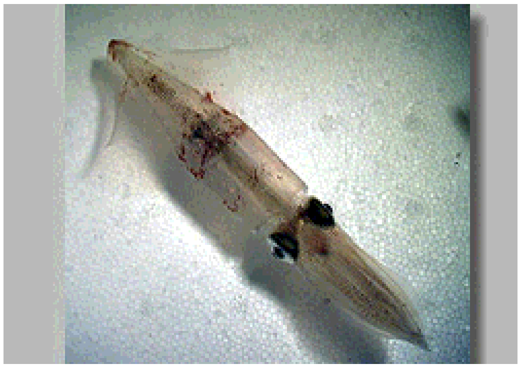
```
forall {nseg = nseg * 3}
```

and repeat the simulation.

Units

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

Physical System


From http://www.mbl.edu

Model

Hodgkin-Huxley cable equations

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

$$\frac{dm}{dt} = -\alpha_m m + \beta_m \cdot (1 - m) \quad \alpha_m = \frac{.1(V+40)}{1-e^{-.1(V+40)}} \quad \beta_m = 4e^{-(V+65)/18}$$

$$\frac{dh}{dt} = -\alpha_h h + \beta_h \cdot (1 - h) \quad \alpha_h = .07e^{-.05(V+65)} \quad \beta_h = \frac{1}{1+e^{-.1(V+35)}}$$

$$\frac{dn}{dt} = -\alpha_n n + \beta_n \cdot (1 - n) \quad \alpha_n = \frac{.01(V+55)}{1-e^{-1(V+55)}} \quad \beta_n = .125e^{-(V+65)/80}$$

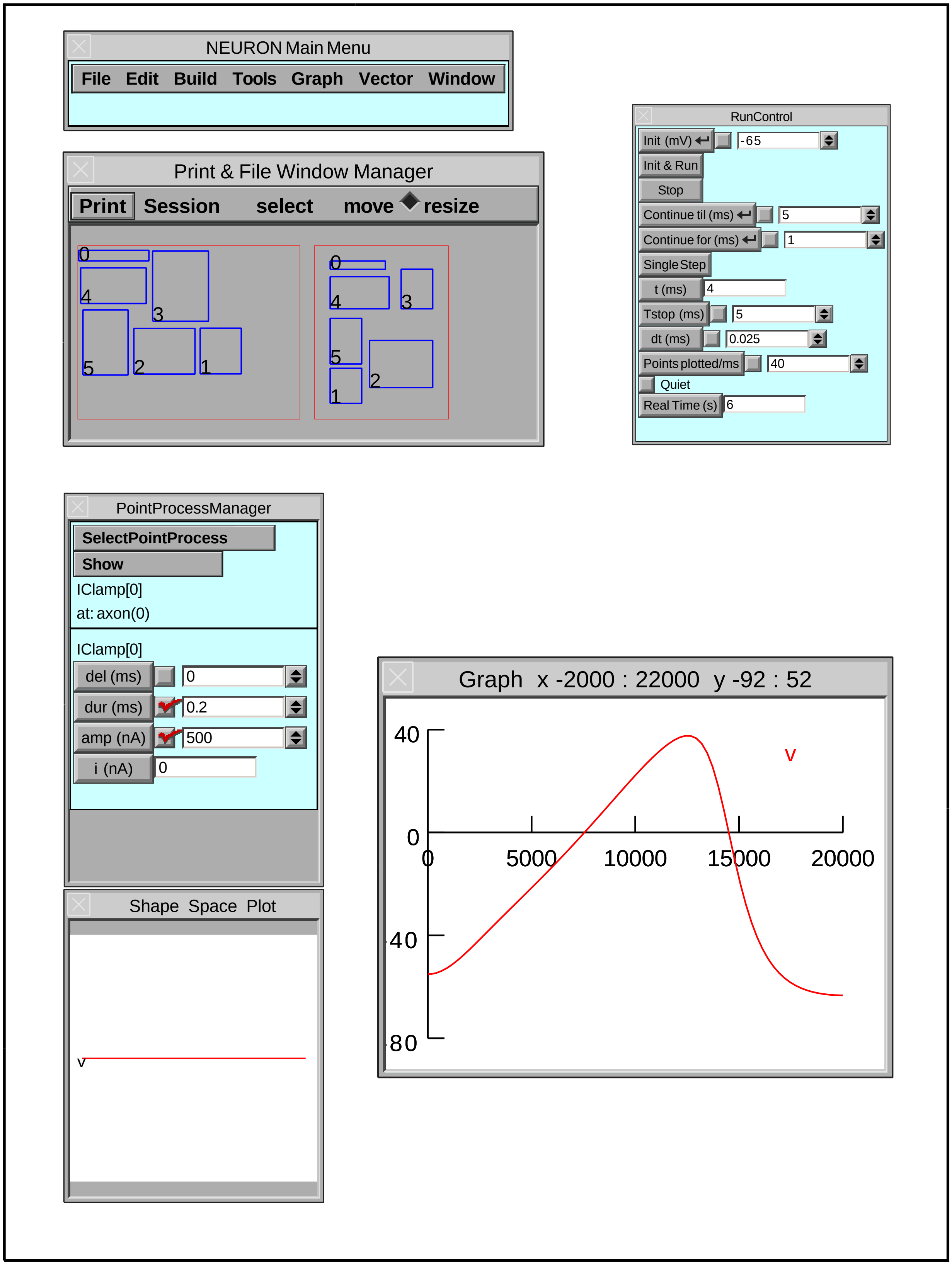
Simulation

Representation

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

Run NEURON with above spec.

Exercises


NEURON Main Menu
File Edit Build Tools Graph Vector Window
Print & File Window Manager
Print Session select move resize
RunControl
Init (mV) -65
Init & Run
Stop
Continue til (ms) 5
Continue for (ms) 1
Single Step
t (ms) 4
Tstop (ms) 5
dt (ms) 0.025
Points plotted/ms 40
Quiet
Real Time (s) 6
PointProcessManager
SelectPointProcess
Show
IClamp[0]
at: axon(0)
IClamp[0]
del (ms) 0
dur (ms) 0.2
amp (nA) 500
i (nA) 0
Graph x -2000 : 22000 y -92 : 52
v
Shape Space Plot
v

Elementary Project Management

Keep

specification of the model

separate from

specification of the interface

in order to

maximize clarity and reduce effort

Elementary Project Management

continued

`modelfile.hoc`

"The organism"

Intrinsic properties of the model:
topology, anatomy, biophysics.

`start.ses`

Eventually will contain custom interface
(synapses, electrodes, graphs,
run control, etc.)

Initially empty or an innocuous statement
`print "ready"`

`init.hoc`

The adminstrative wrapper

```
load_file("nrngui.hoc")
load_file("modelfile.hoc")
load_file("start.ses")
```


Elementary Project Management

continued

Usage

1. Execute init.hoc
   UNIX: `nrngui init.hoc`
   MSWin: double click on `init.hoc`

   This brings up NEURONMainMenu

2. Customize interface
   Attach synapses and electrodes
   Set up graphs and run control

3. Save interface to `start.ses`

The next time you run `init.hoc`,
the interface you saved in `start.ses`
is automatically retrieved.

Example: Ball - Stick model

Physical system: pyramidal neuron
enormously complicated dendritic tree

Model: isopotential soma with dendritic cylinder

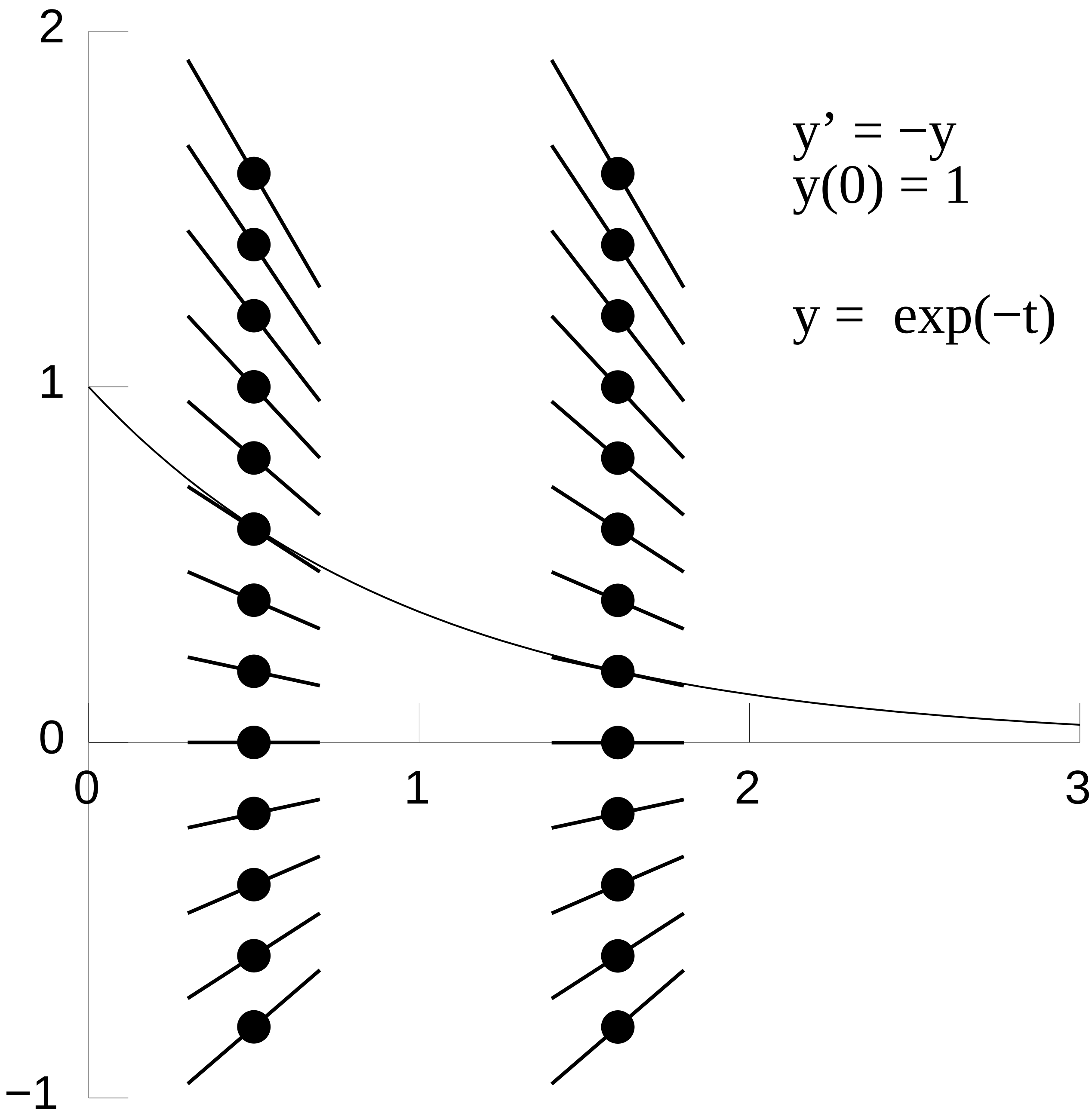
Project goals:

- Create the model and custom GUI from scratch.
- Learn how to use the CellBuilder.
- Use session files to save and retrieve the user interface (elementary project management).
- Test the simulation:
  structural integrity
  spatial grid
  time steps

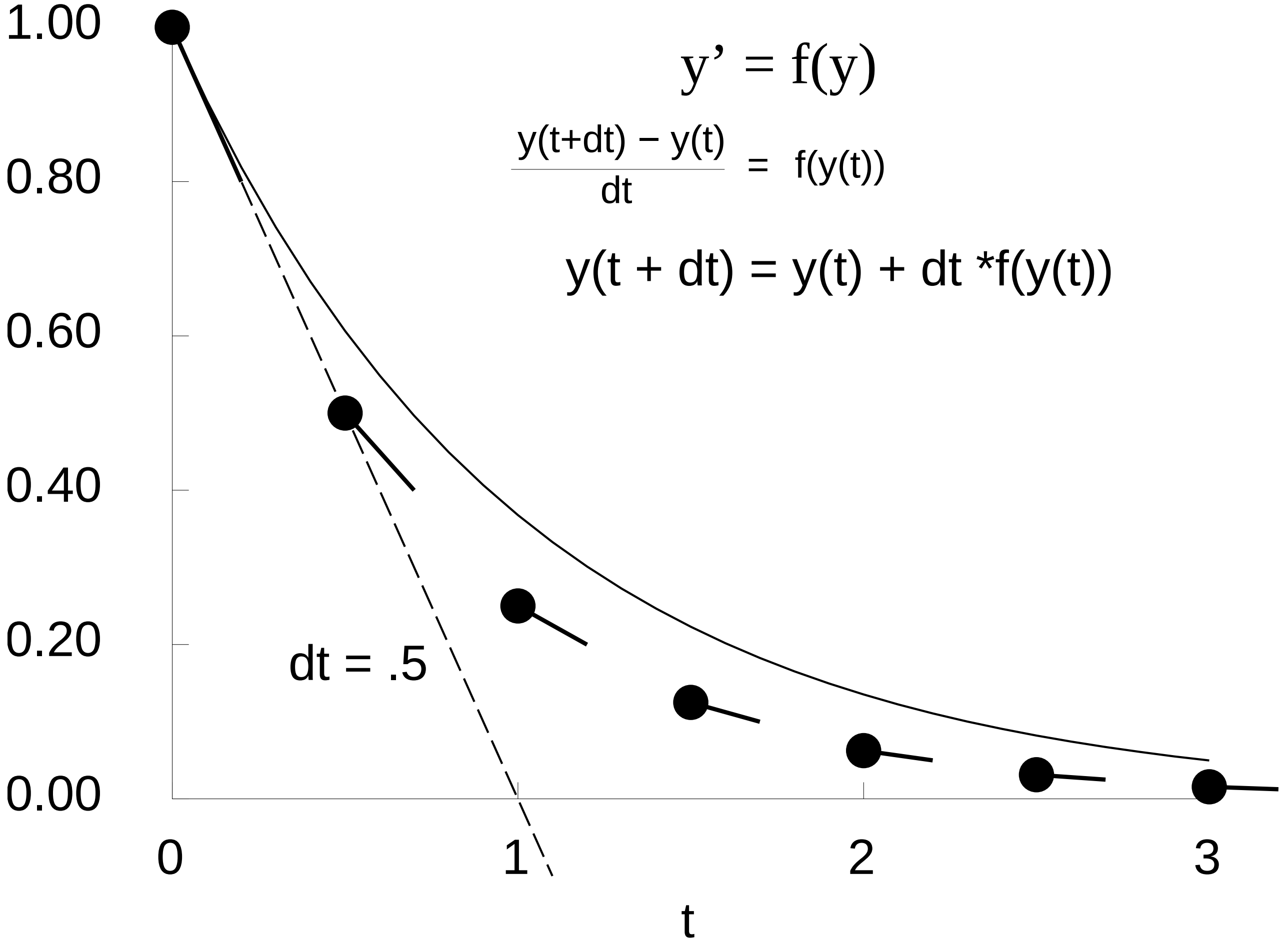
$$y' = f(y)$$

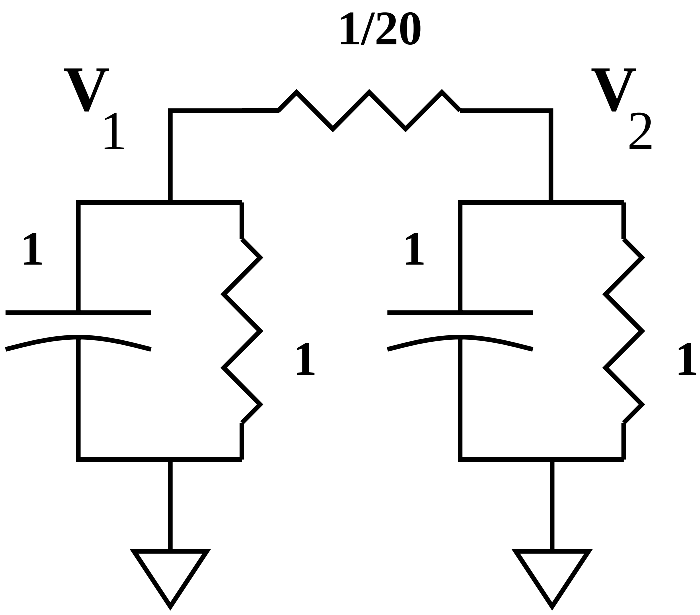
$$y' = -y$$
$$y(0) = 1$$

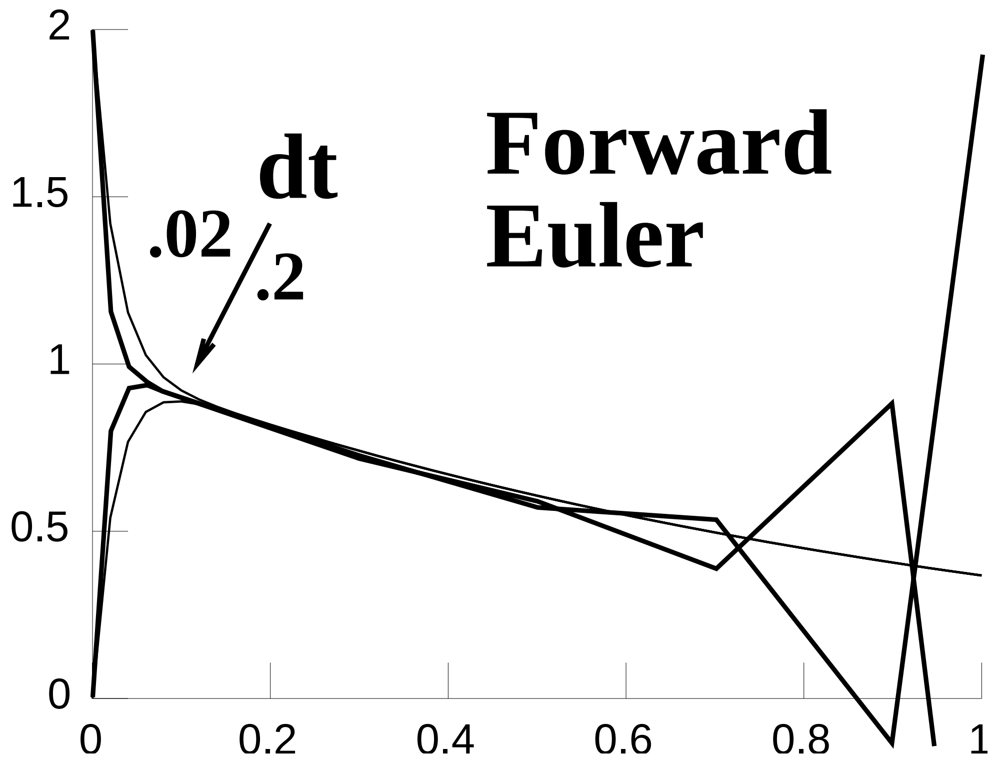
$$y = \exp(-t)$$



Forward Euler




1/20
V1
V2
1
1
1
1

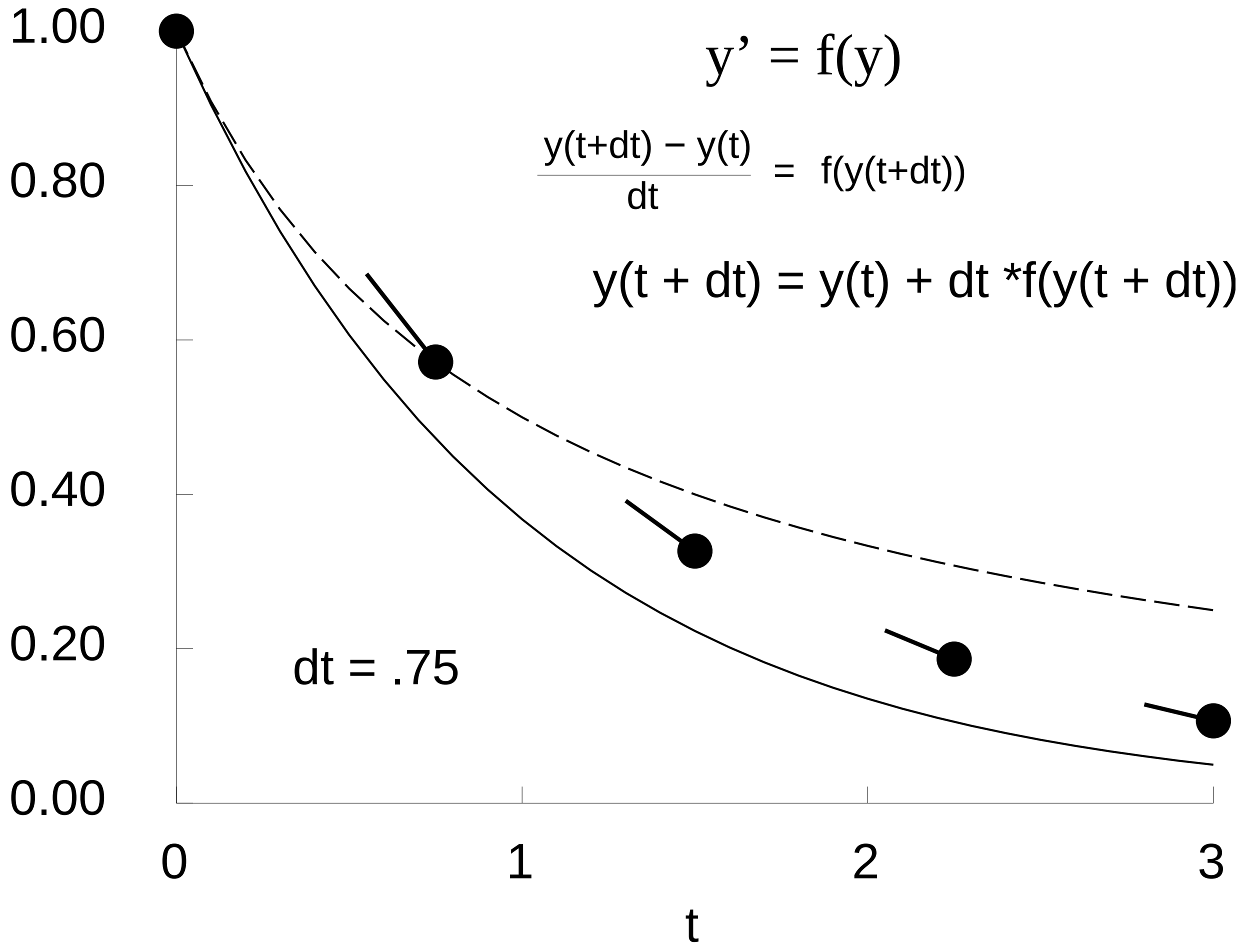

2
1.5
1
0.5
0
0
0.2
0.4
0.6
0.8
1
dt
.02
.2
Forward
Euler

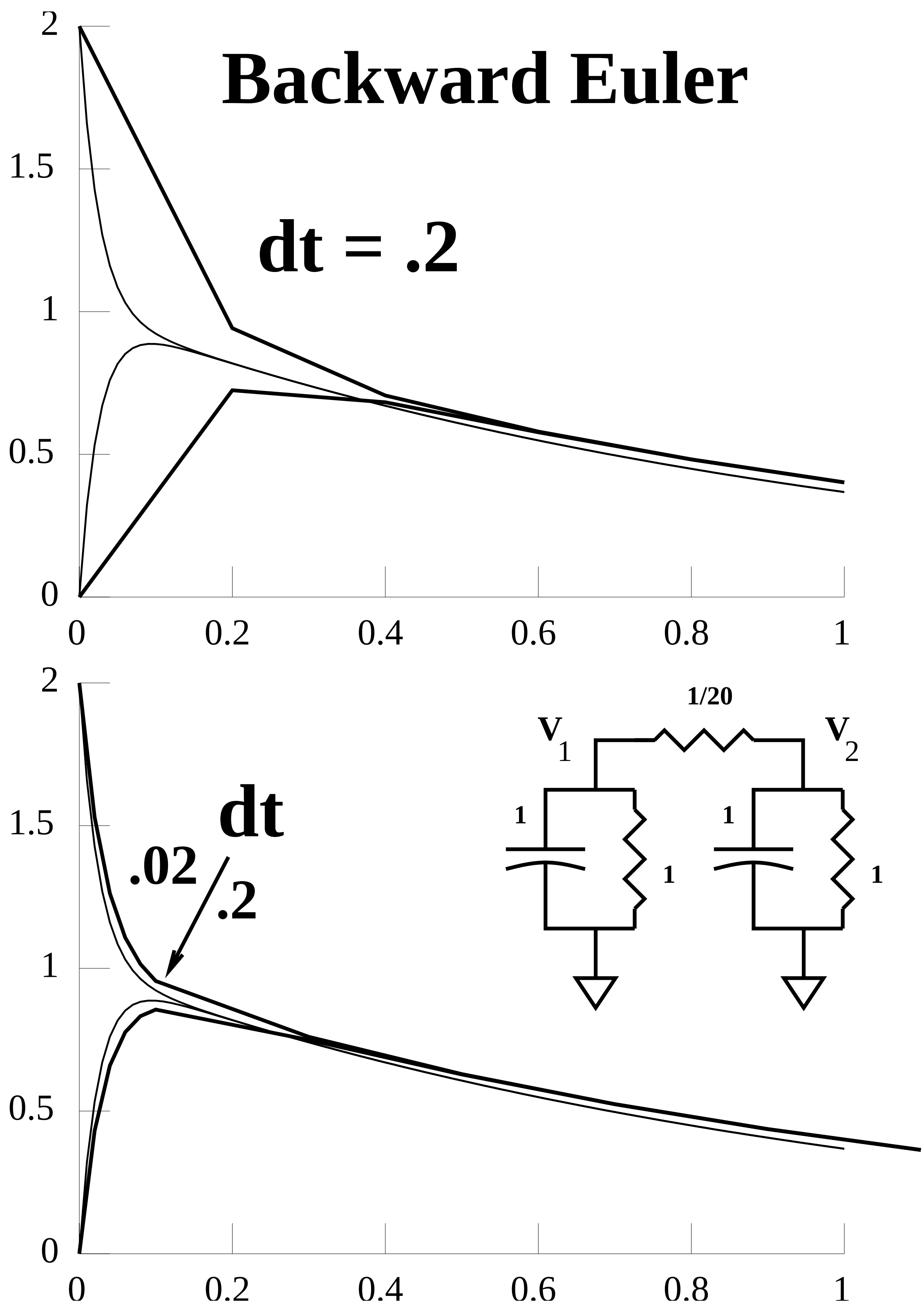
Backward Euler

$$y' = f(y)$$

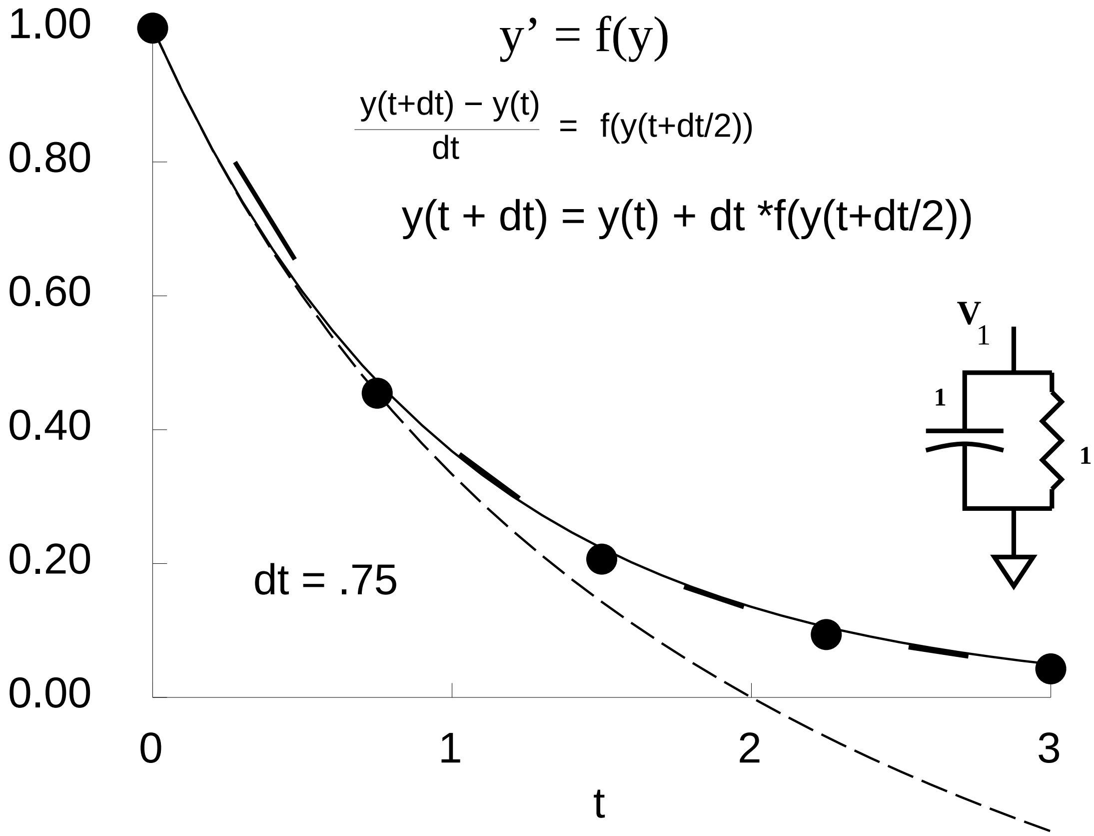
$$\frac{y(t+dt) - y(t)}{dt} = f(y(t+dt))$$

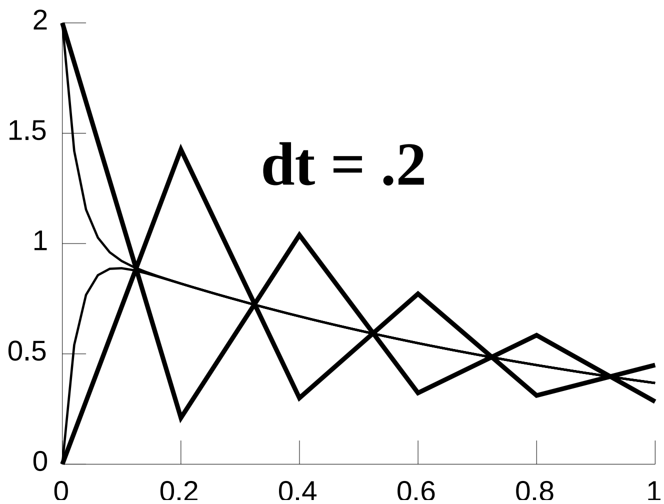
$$y(t + dt) = y(t) + dt\, {*}f(y(t + dt))$$

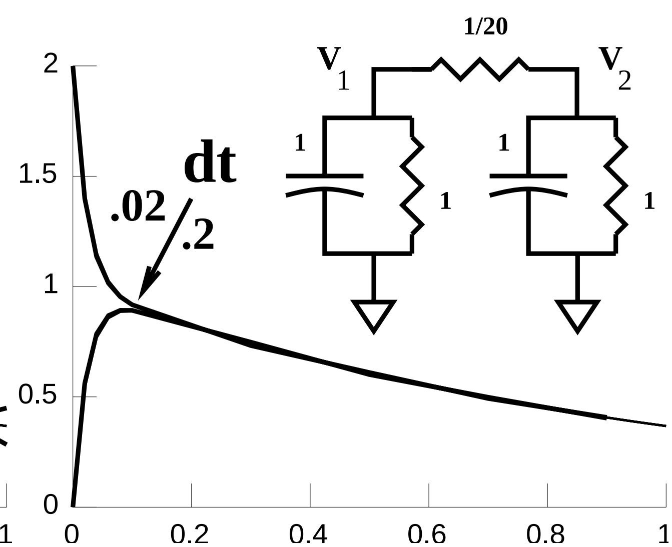



Backward Euler
dt = .2
2
1.5
1
0.5
0
0
0.2
0.4
0.6
0.8
1
dt
.02
.2
1/20
V1
V2
1
1
1
1

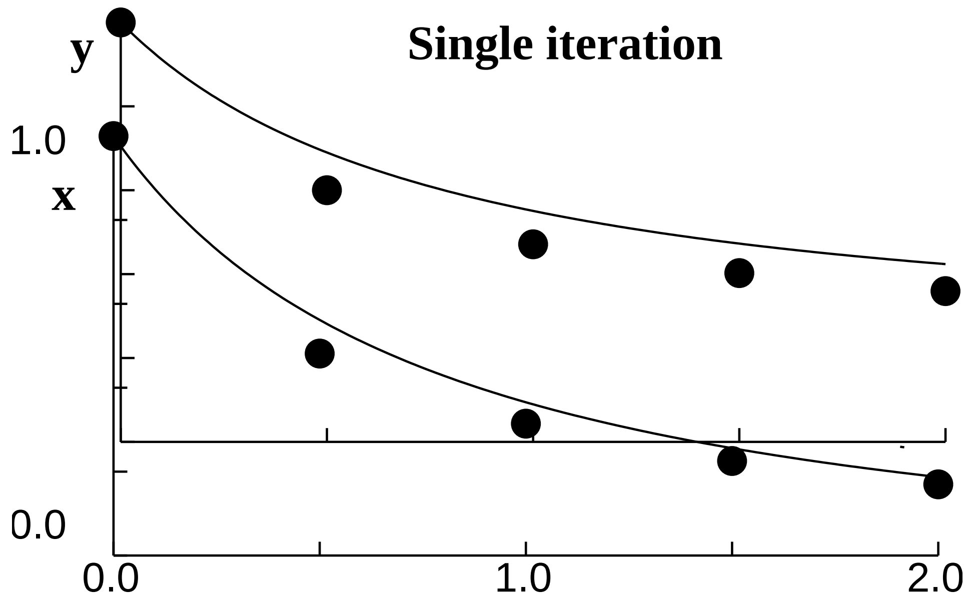
Crank−Nicholson

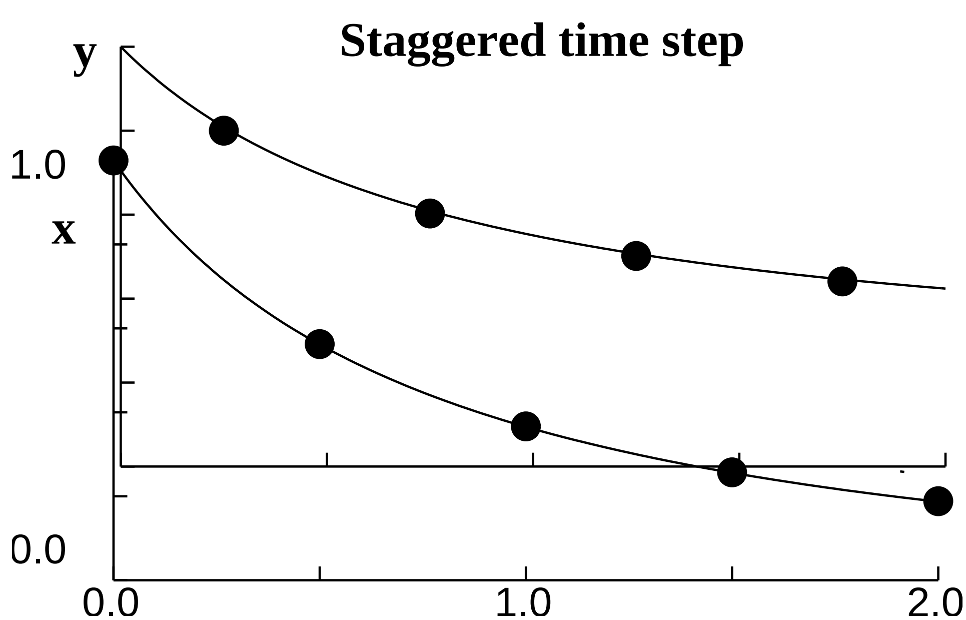


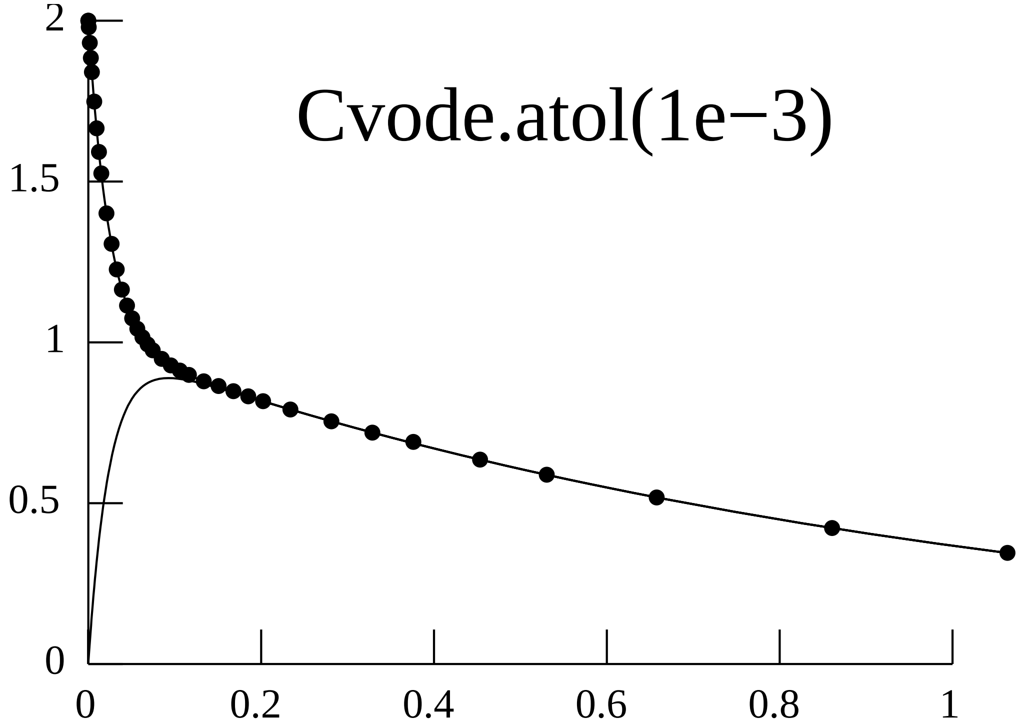


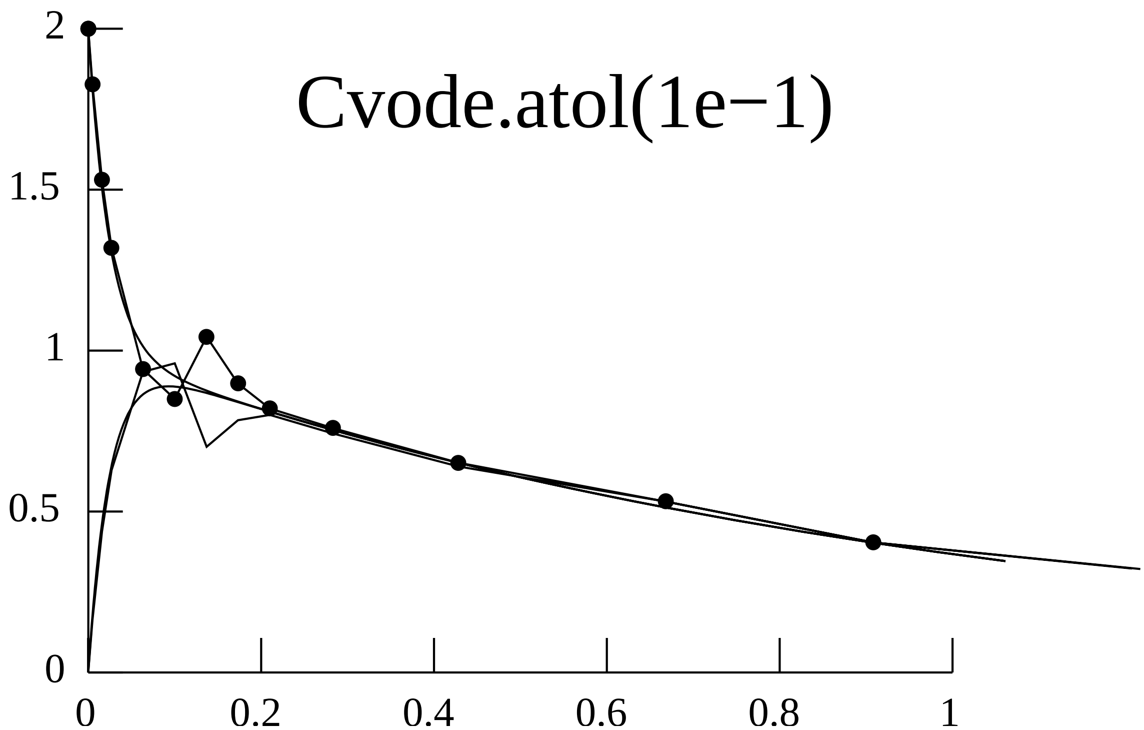


$$x' = -1.4xy$$
$$y' = -xy$$

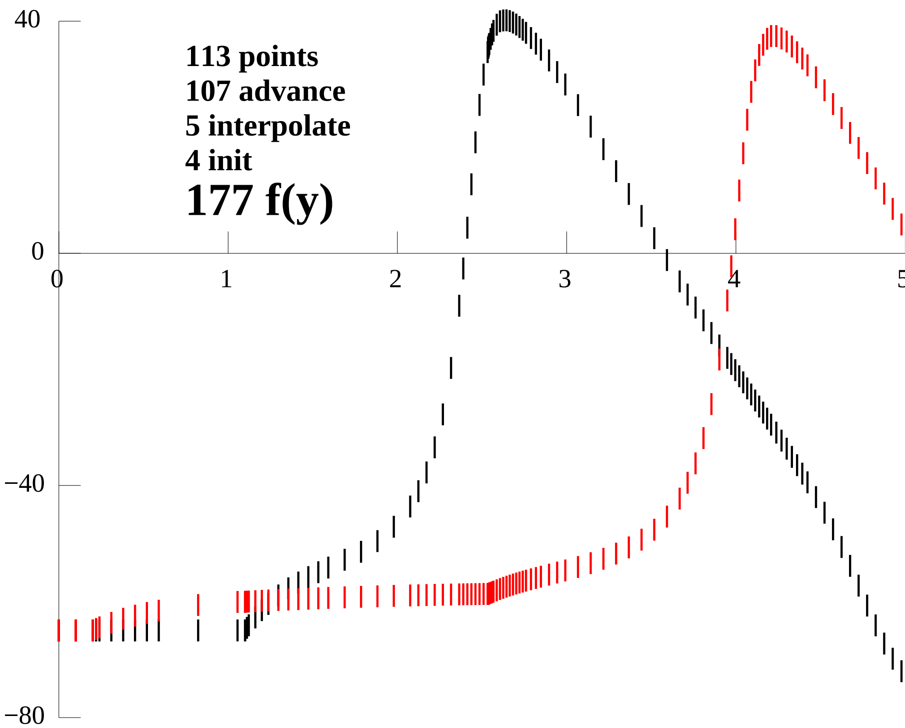

Single iteration
y
1.0
x
0.0
0.0
1.0
2.0

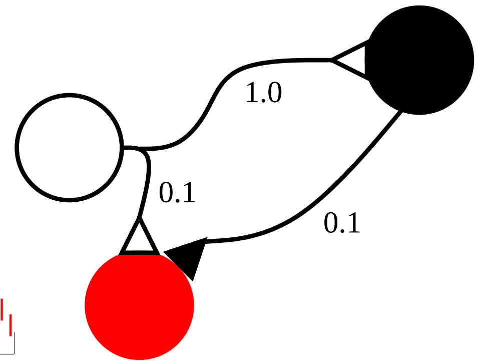

Staggered time step
y
1.0
x
0.0
0.0
1.0
2.0


Cvode.atol(1e−3)
2
1.5
1
0.5
0
0 0.2 0.4 0.6 0.8 1


Cvode.atol(1e−1)
2
1.5
1
0.5
0
0 0.2 0.4 0.6 0.8 1

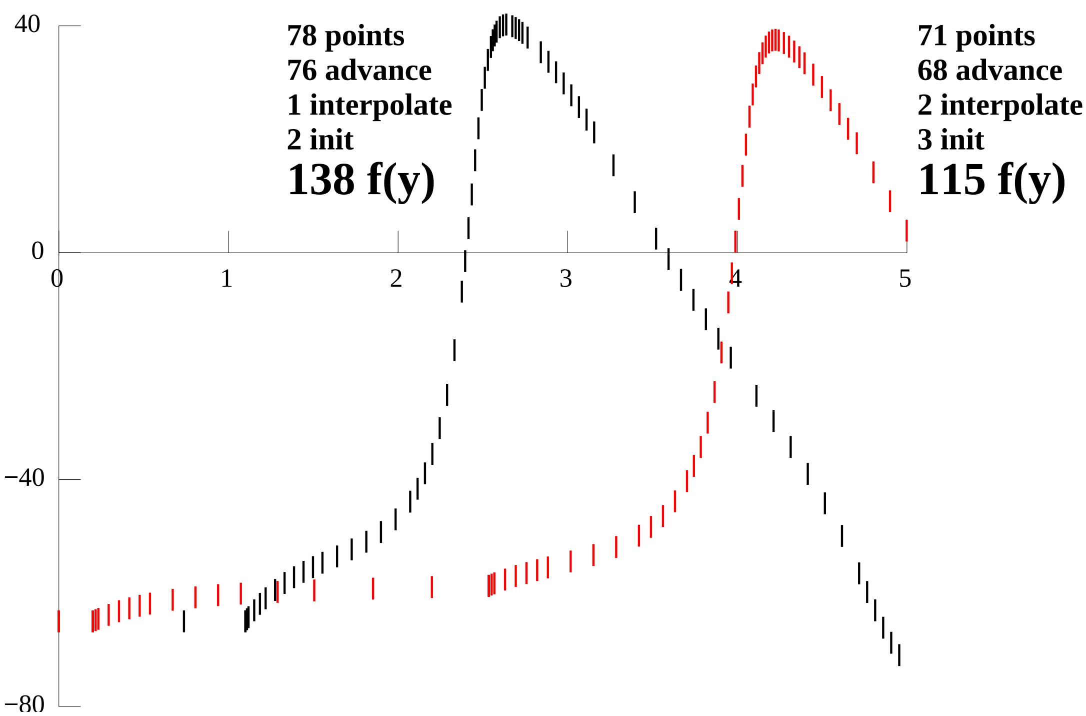
Global Step



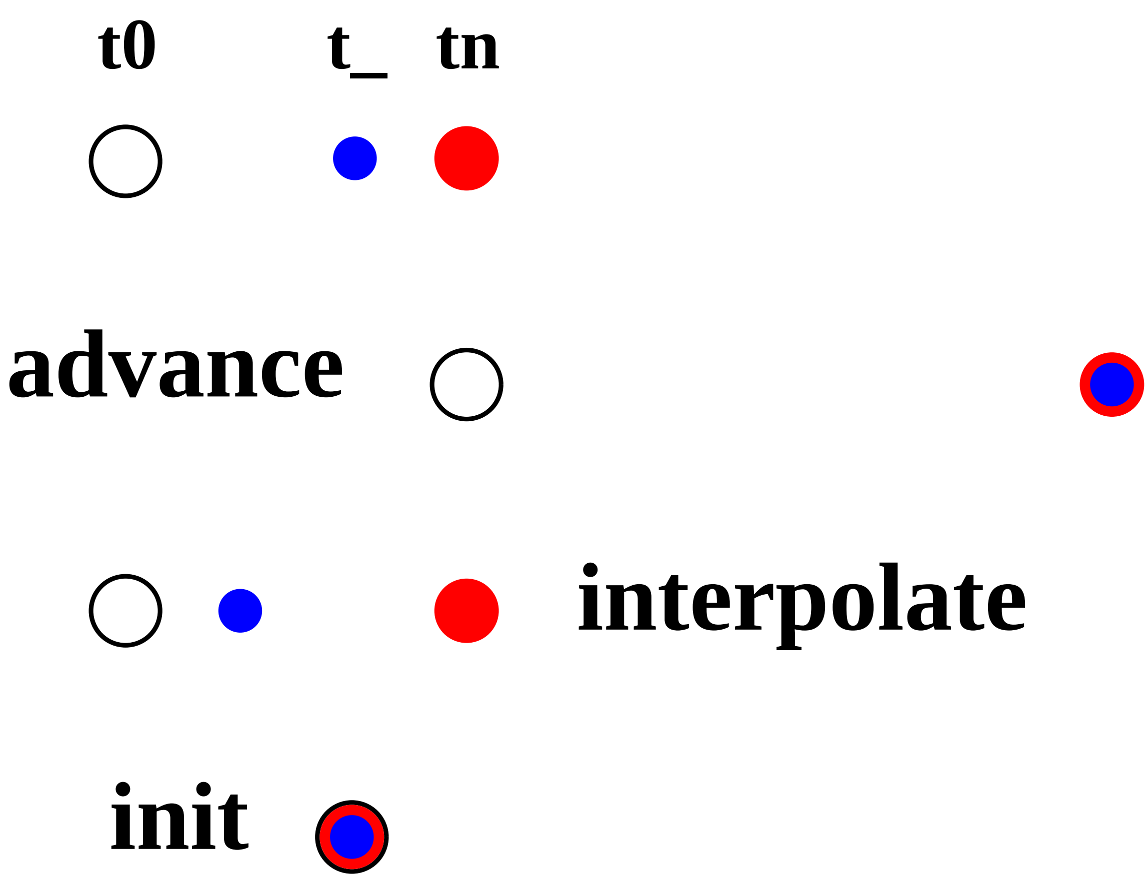


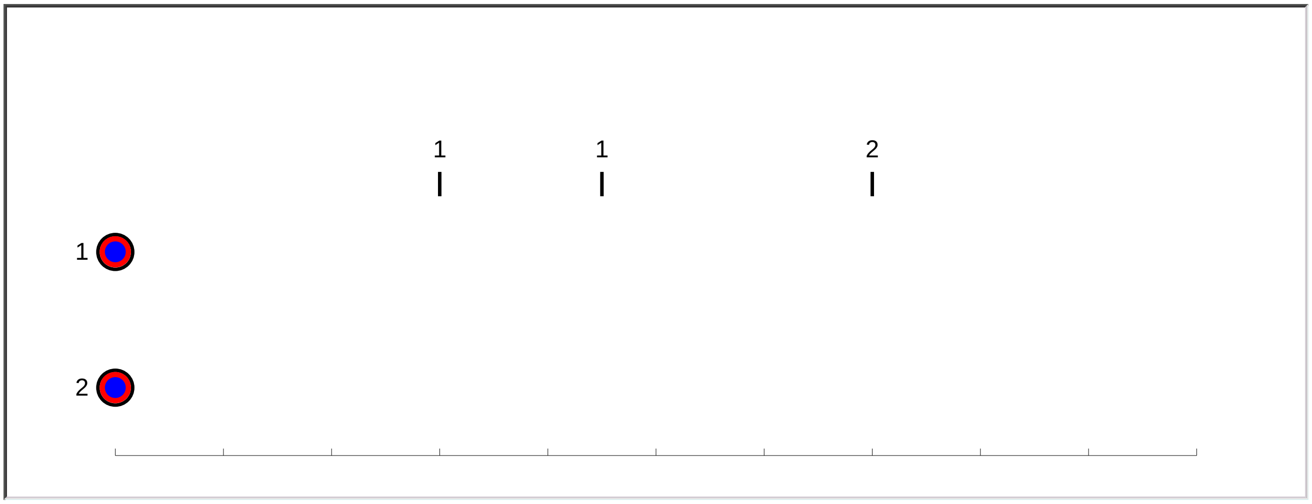
(177*8)/(138*4 + 115*4) = 1.4

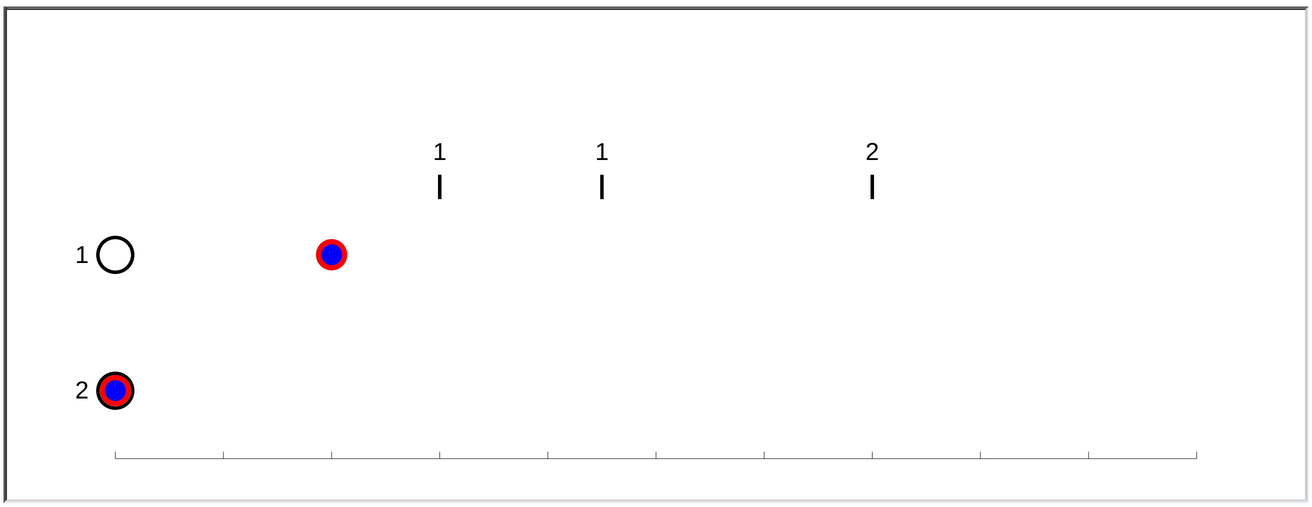
Local Step

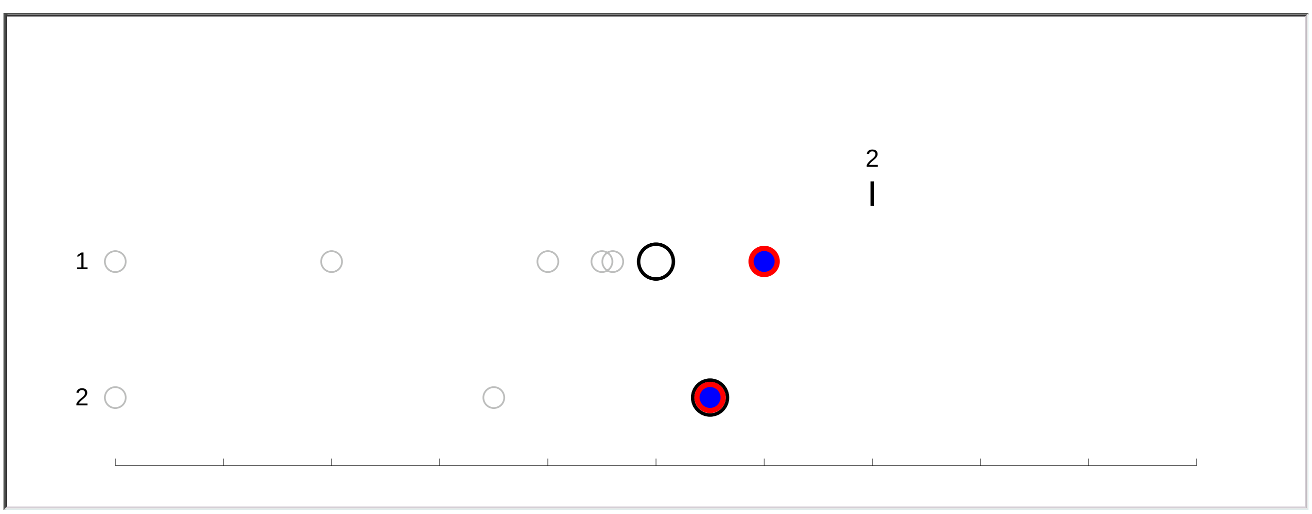


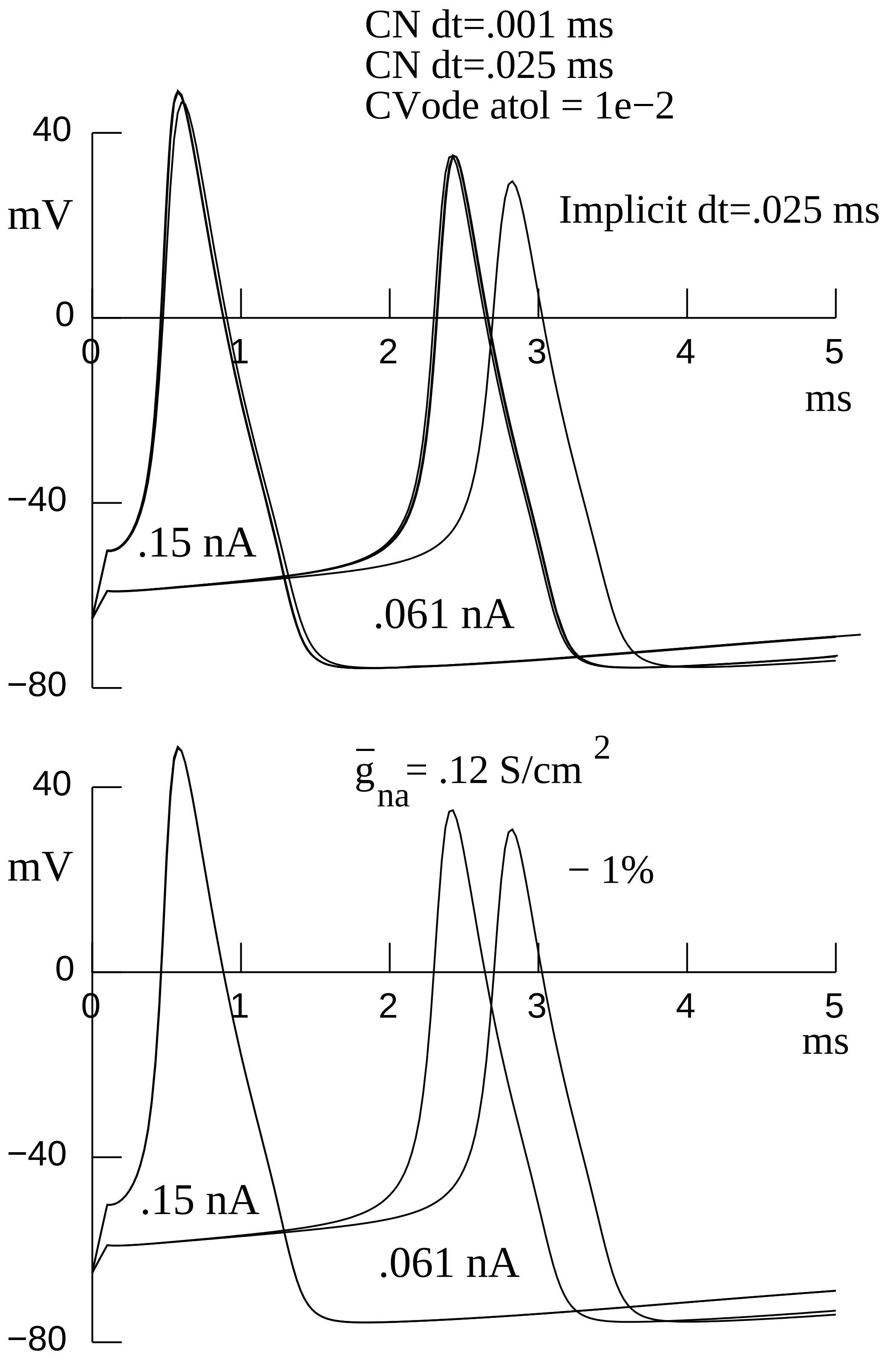
One integrator instance per cell




1
1
2
1
2


1
1
2
1
2


2
1
2


CN dt=.001 ms
CN dt=.025 ms
CVode atol = 1e−2
Implicit dt=.025 ms
mV
40
0
−40
−80
0 1 2 3 4 5
ms
.15 nA
.061 nA
$\bar{g}_{na}$ = .12 S/cm 2
− 1%
mV
40
0
−40
−80
0 1 2 3 4 5
ms
.15 nA
.061 nA

An Outline for Coding NEURON Models

Creating a model
- by writing `hoc` code
- and using the graphical interface

User interface
- session files

Tests
- structural integrity
- spatial grid
- time steps

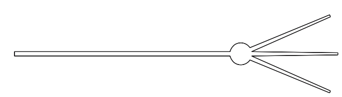
Creating a Model with `hoc`

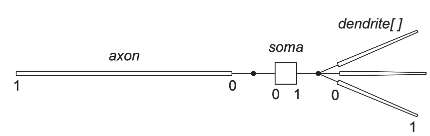
1. Establish model topology
2. Assign properties
3. Attach synapses and electrodes
4. Control simulation time course

1. Establish model topology

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

Example






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

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

// specify default section
access soma
```


2. Assign properties

- Compartmentalization
  `nseg`
- Anatomical properties
  `L, diam`
- Biophysical properties
  `insert`


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

axon {
   nseg = 20
   L = 1000
   diam = 1
   insert hh
}

for i=0,2 dendrite[i] {
   nseg = 5
   L = 200
   diam(0:1) = 10:3  // taper
   insert pas        // passive membrane
}

forall Ra = 60  // [ohm cm]
```


Range variables

Vary continuously in space
along the length of a section

Examples: v, cm, diam

Section variables

Pertain to an entire section

Examples: Ra (cytoplasmic resistivity),
L, nseg

Global variables

Same across all sections

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

3. Attach synapses and electrodes

```
objref stim     // create it
// put it in middle of soma
soma stim = new IClamp(0.5)
stim.del = 1    // [ms] delay
stim.dur = 0.1  // [ms] duration
stim.amp = 60   // [nA] amplitude
```

4. Control simulation time course

```
dt = 0.05  // [ms] integration time step
tstop = 5  // [ms]
finitialize(-65)  // initialize Vm,
                  //   state variables, and time


proc integrate() {
  // show starting time & initial somatic Vm
  print t, v(0.5)  // soma is default section

  while (t < tstop) {
    fadvance()  // advance solution by dt

    // function calls to save or plot results
    //   might go here, e.g.
    // print t, v(0.5)
  }
}
```


Custom initialization

```
proc init() {
  // set Vm to -65 mV in all sections
  forall v = -65

  // set Vm to -70 in "basal"
  basal v = -70

  // set t to 0
  //   & initialize all mechanisms
  //   using the assigned v values
  finitialize()

  // make all assigned variables consistent
  //   (currents, conductances,
  //    & equilibrium potentials)
  fcurrent()
}
```


Distributed Mechanisms

Examples:

- ion buffers
- voltage-gated ion channels

```
soma insert hh
```


Point Processes

Examples:

- synapses
- clamps, spike detectors

Object syntax

```
objref stim
soma stim = new IClamp(0.5)
stim.del = 1     // [ms]
stim.dur = 0.1   // [ms]
stim.amp = 60    // [nA]
```


NEURON: the HOC programming language

Bill Lytton

SUNY - Downstate

Brooklyn, NY

TOC

Talk to the simulator

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

Numbers

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

Functions & operators: pluses and minuse

_ _ _

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

NEURON: the HOC programming language – p.4/27

NB: x=5 vs x==5

_ _ _

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

NEURON: the HOC programming language – p.5/27

Assignments

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

Block of code

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


Conditionals and controls

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

proc and func

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

Number arguments to procedures:

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

Strings

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

Objects

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

Simulation commands

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

Sim - stim

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

Sim - running

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

Vectors

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

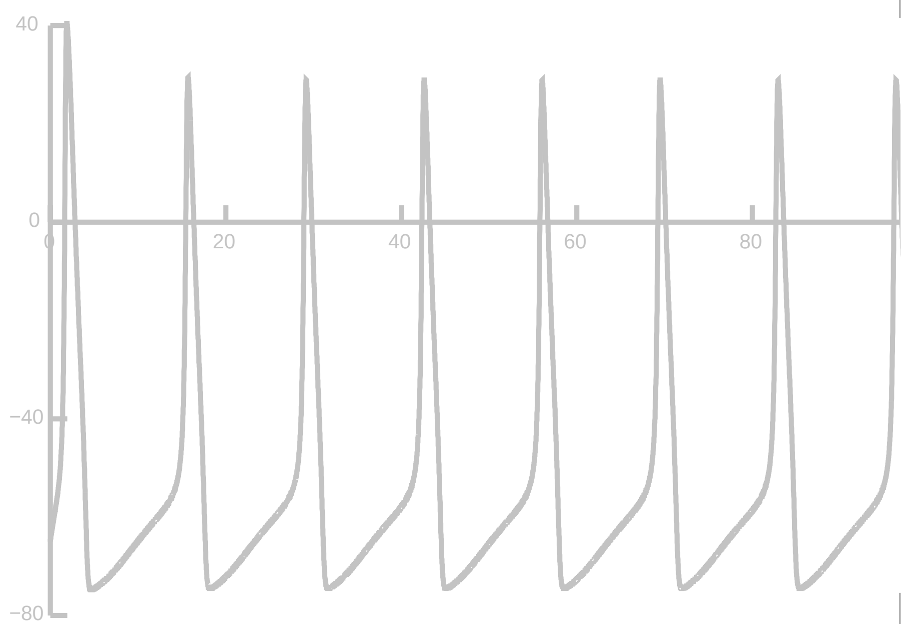
What have we recorded?

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

Can analyze signals using vectors

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

Quick & dirty graphics



Graphing a vector

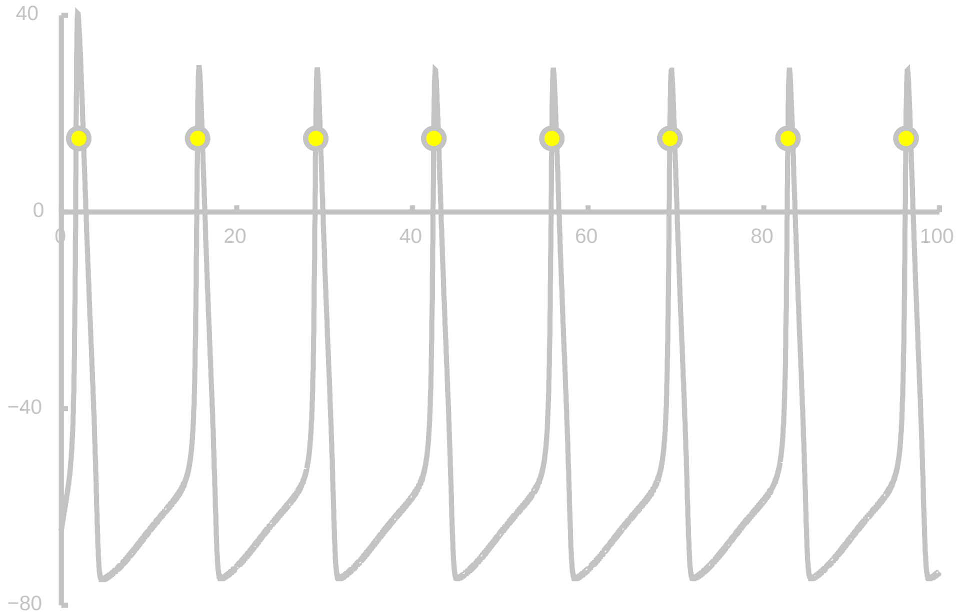
- Can put up a graph from the main menu or by hand
  g = new Graph()
- Draw the vector on the graph
  vec.line(g,dt)
- Need a time vector if using var dt
- Erase and redraw
  g.erase

Find spikes

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

Check results graphically

for ii=0,ind.size-1 g.mark(vec[2].x[ii],15,"O")



Now can calculate means etc.

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

Other useful vector functions

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

Putting up buttons

Reading and writing files

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

The Vector class

Efficient methods for collecting
and manipulating arrays of numbers

Vector functions page 1 of 2

Initialization & storage

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

Database

sort	sortindex		
contains	label		
index	max_ind	min_ind	ind
where	indwhere	indvwhere	

I/O

fread	fwrite	
vread	vwrite	
scanf	printf	scantil

Graph

plot	line	ploterr	mark

Vector functions page 2 of 2

Algebra

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

Signal processing

fft	spctrm	convlv	correl	filter
medfltr	addrand	apply		
psth	spikebin	trigavg		
reduce				

Statistics

mean	median		
stderr	stdev	var	meansqerr
fit			
hist	histogram	smhist	sumgauss

Simulation

record	play	play_remove
interpolate		

Vector record

SYNTAX

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

DESCRIPTION

Save the stream of values of "`var`" during a simulation into the `vdest` vector.

Vector play

SYNTAX

```
vsrc.play(&var)
vsrc.play(&var, Dt)
vsrc.play(&var, tvec)
vsrc.play("stmt involving $1",
  optional Dt or tvec arg)
vsrc.play(index)
```

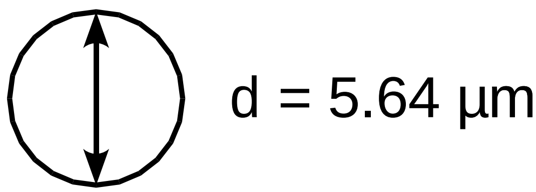
DESCRIPTION

The `vsrc` vector values are assigned to the "`var`" variable during a simulation.

Example: Model control
Arbitrary forcing functions

Physical system: patch of squid axon

Model: isopotential compartment with HH currents



surface area = 100 µm^2

Project goals:

- Set up and test a voltage clamp (SEClamp)
- Grapher tool: generate an arbitrary waveform (voltage ramp)
- Clipboard: capture and transfer Vector data
- Vector Play tool: use the recorded waveform as the voltage clamp's "command"

The NEURON Simulation Environment Figure 1

Example: Simulation Families

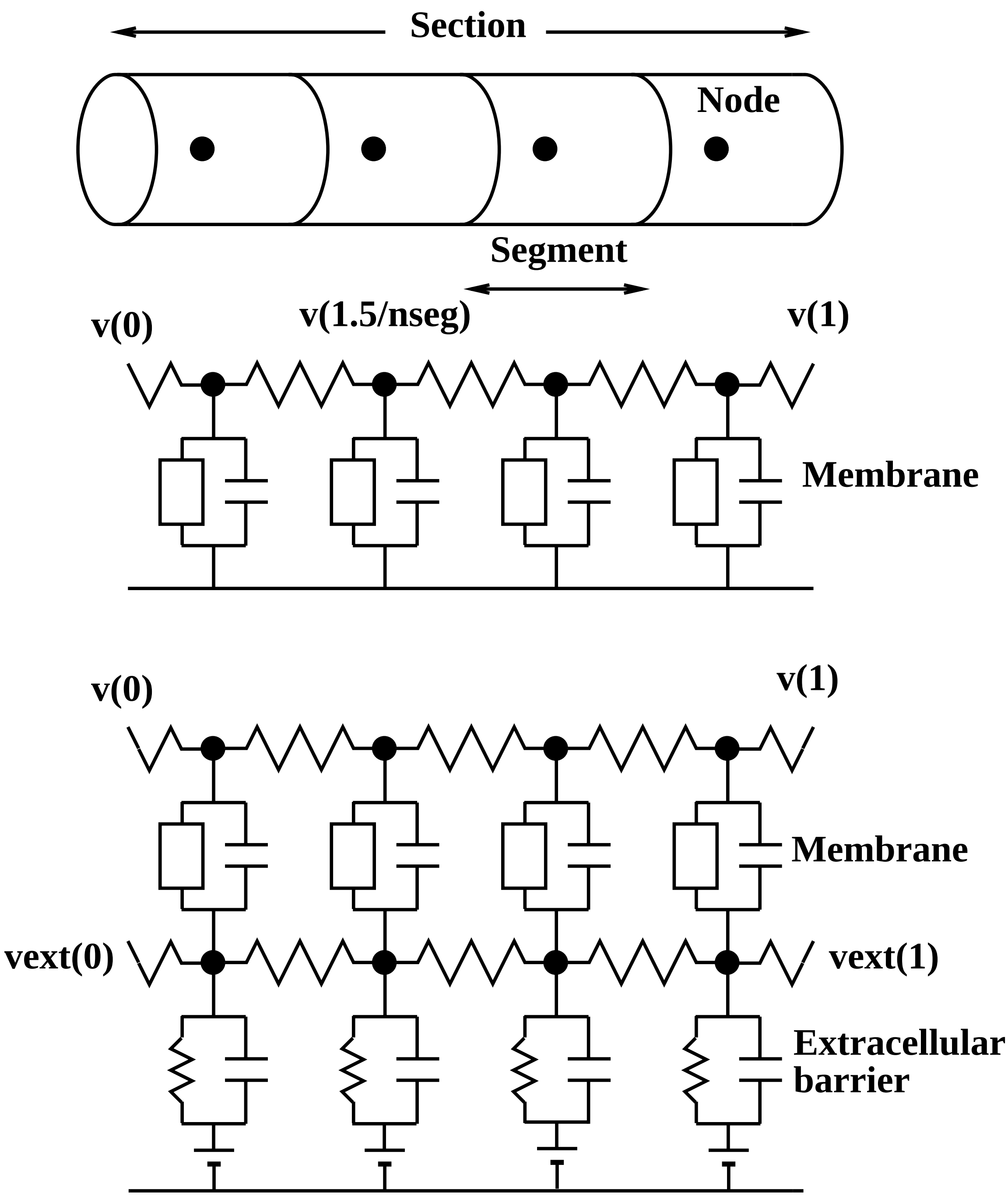
Physical System: pyramidal neuron

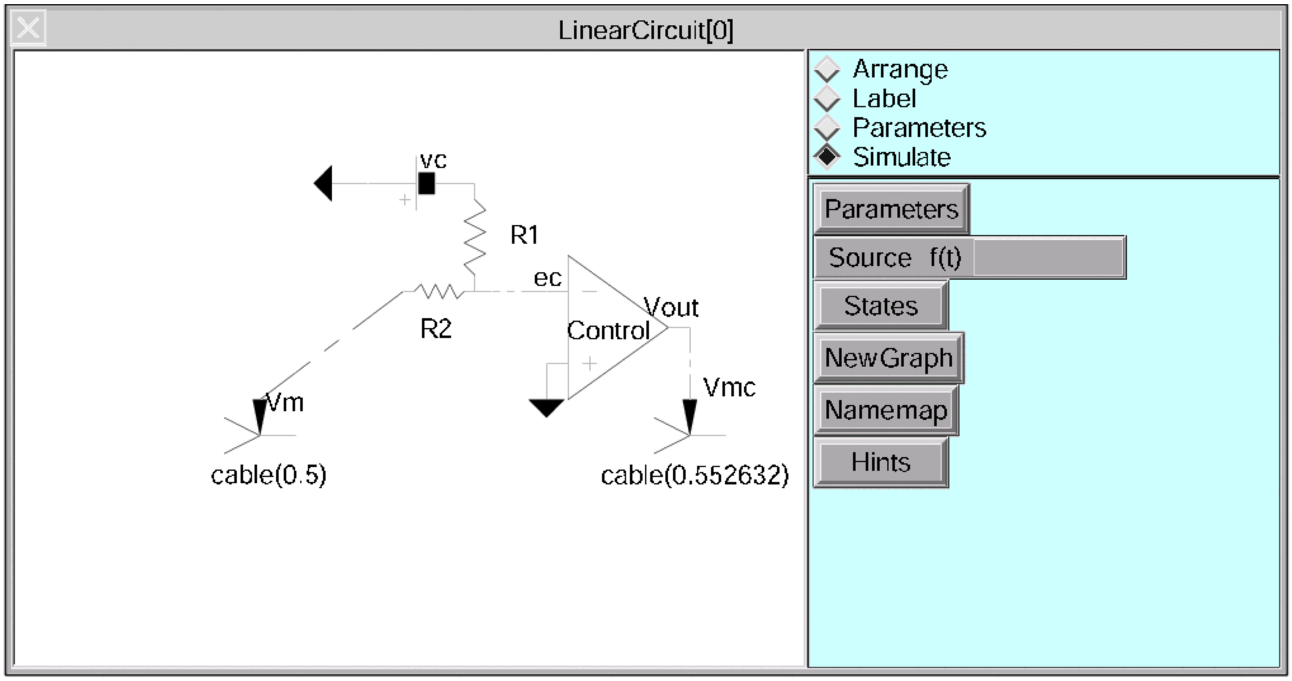
Model: isopotential soma with dendritic cylinder

conductance−change synapse

Project goals:

- How does peak synaptic depolarization vary with synaptic location?
- Program control of PointProcess location
- Exploratory data collection and analysis
- Using the Vector class to automate data collection and analysis across multiple simulation runs.


Section
Node
Segment
v(0)
v(1.5/nseg)
v(1)
Membrane
v(0)
v(1)
Membrane
vext(0)
vext(1)
Extracellular
barrier



Cable

```
d a              v1
b d a            vm
  b d a          v3
    b d a        vmc
      b d        v5
```

C y' + G y = b

```
x                vm
  x     x        vmc
x   x x          ec
    x x   x      vc
    x            IC
      x          Ivc
```

```
d a                    v1
b d a                  vm
  b d a                v3
    b d a     x        vmc
      b d              v5
  x     x x            ec
        x x   x        vc
        x              IC
          x            Ivc
```


NMODL

NEURON Model Description Language

Add new membrane mechanisms to NEURON

Density mechanisms

- Distributed Channels
- Ion accumulation

Point Processes

- Electrodes
- Synapses

Described by

- Differential equations
- Kinetic schemes
- Algebraic equations

Benefits

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

NMODL general block structure

What the model looks like from outside

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

What names are manipulated by this model

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

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

STATE { n ... }

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

Initial default values for states

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

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

(and specify how states are to be integrated)

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

State equations

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

Functions and procedures

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


Density mechanism

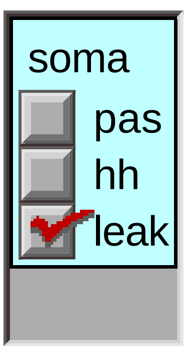
Point Process

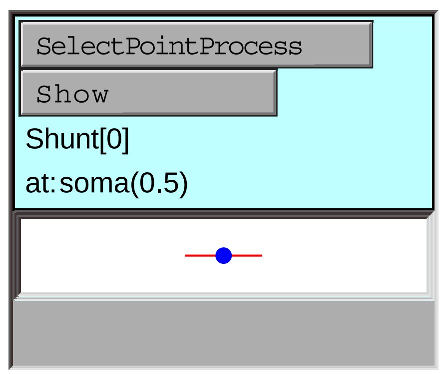
NMODL

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

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

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

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

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

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

ASSIGNED {
    i (nanoamp)
    v (millivolt)
}

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

GUI





Interpreter

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

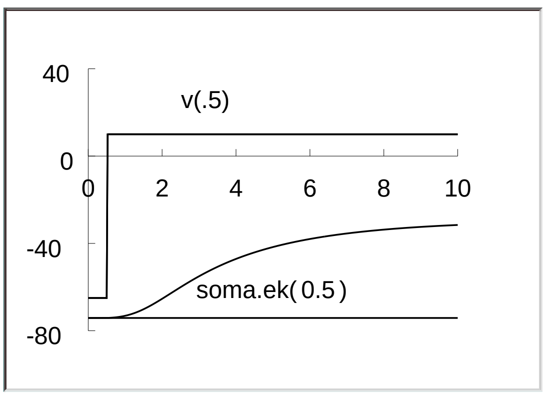
```
objref s
soma s = new Shunt(.5)
s.r = 2
```

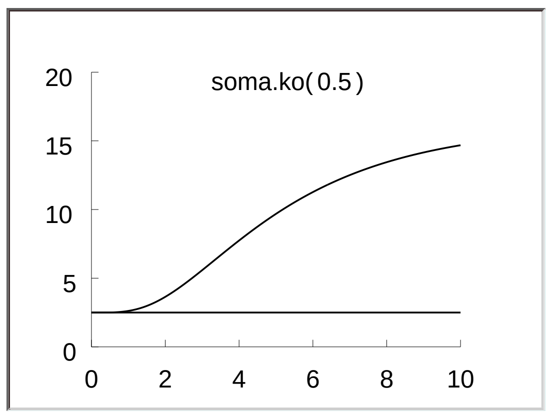

Ion Channel

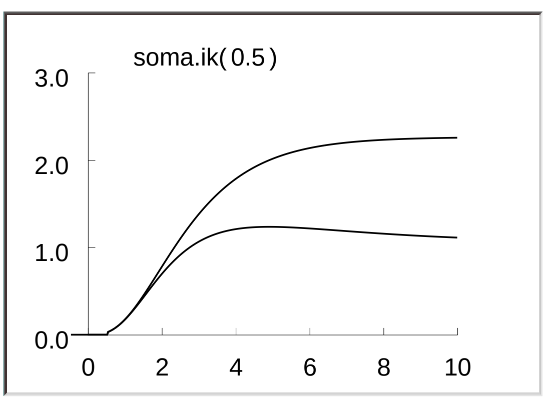
```
NEURON {
  USEION k READ ek WRITE ik
}
BREAKPOINT {
  SOLVE states METHOD cnexp
  ik = gbar*n*n*n*n*(v - ek)
}
DERIVATIVE states {
  rate(v*1(/mV))
  n' = (inf - n)/tau
}
```

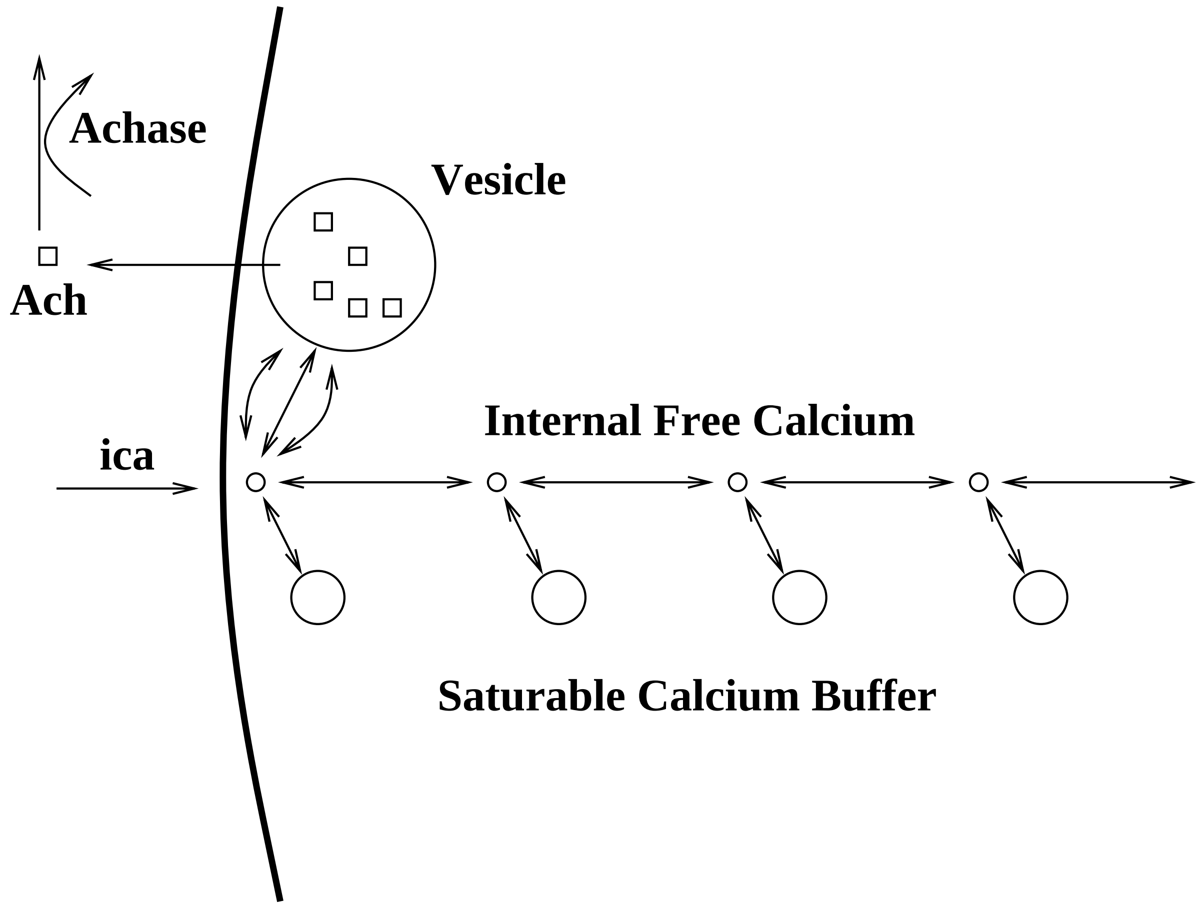
Ion Accumulation

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

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










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

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

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

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

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

UNITS Checking

```
NEURON { POINT_PROCESS Shunt ... }

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

ASSIGNED {
        i (nanoamp)
        v (millivolt)
}

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

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

```
modlunit shunt
```

is:

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

To fix the problem replace the line with:

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

What conversion factor will make the following consistent?

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


UNIX

In the directory containing the desired mod files:

```
nrnivmodl
nrngui
```

Select NEURONMainMenu/Build/singlecompartment.

MSWIN

Launch mknrndll from the icon in the NEURON program group.

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

Launch nrngui from the icon in the NEURON program group.

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

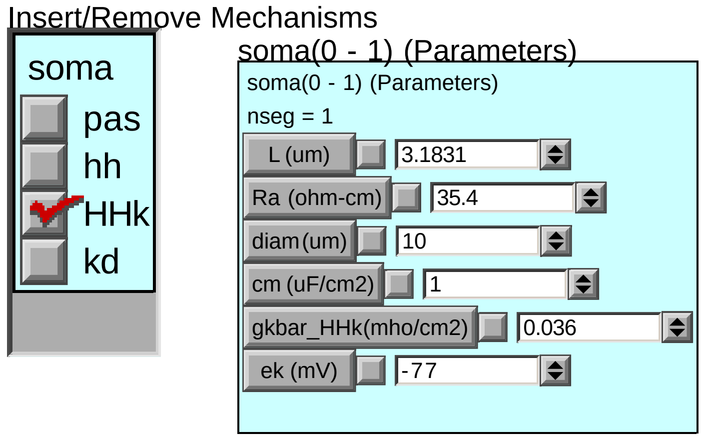
---

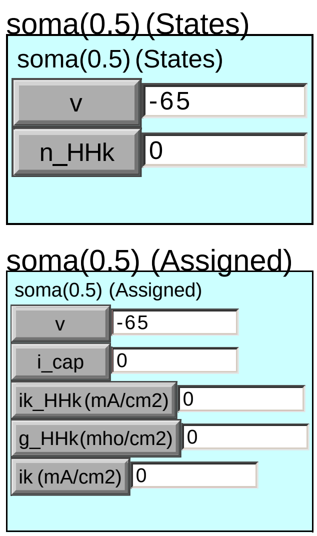
single.hoc

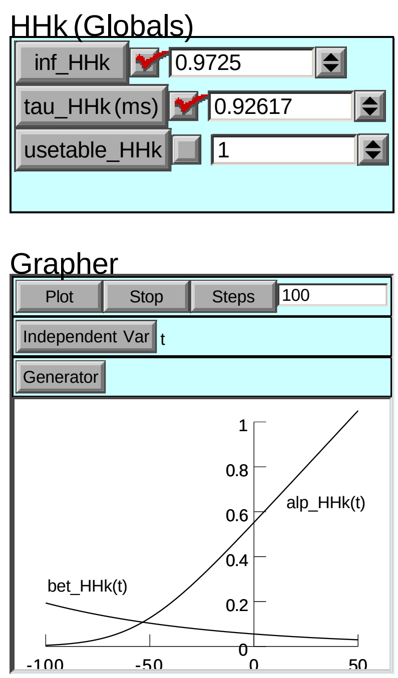
```
load_file("stdgui.hoc")
create soma
access soma
// area 100 um2 means mA/cm2 identical to nA
{diam=10  L=10/PI}
```

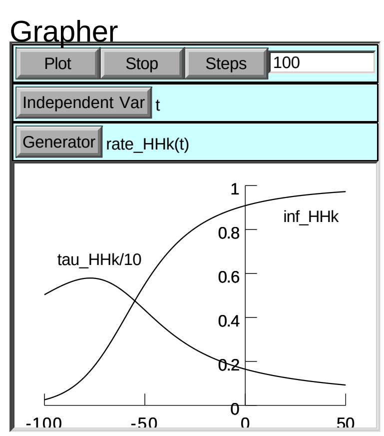


```
: Hodgkin - Huxley k channel

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









HH potassium channel conductance data

508 *A. L. HODGKIN AND A. F. HUXLEY*

From eqn. (6) this may be transformed into a form suitable for comparison with the experimental results, i.e.

$$g_K = \{(g_{K\infty})^{\frac{1}{4}} - [(g_{K\infty})^{\frac{1}{4}} - (g_{K0})^{\frac{1}{4}}] \exp(-t/\tau_n)\}^4, \quad (11)$$

where $g_{K\infty}$ is the value which the conductance finally attains and g_{K0} is the conductance at $t=0$. The smooth curves in Fig. 3 were calculated from

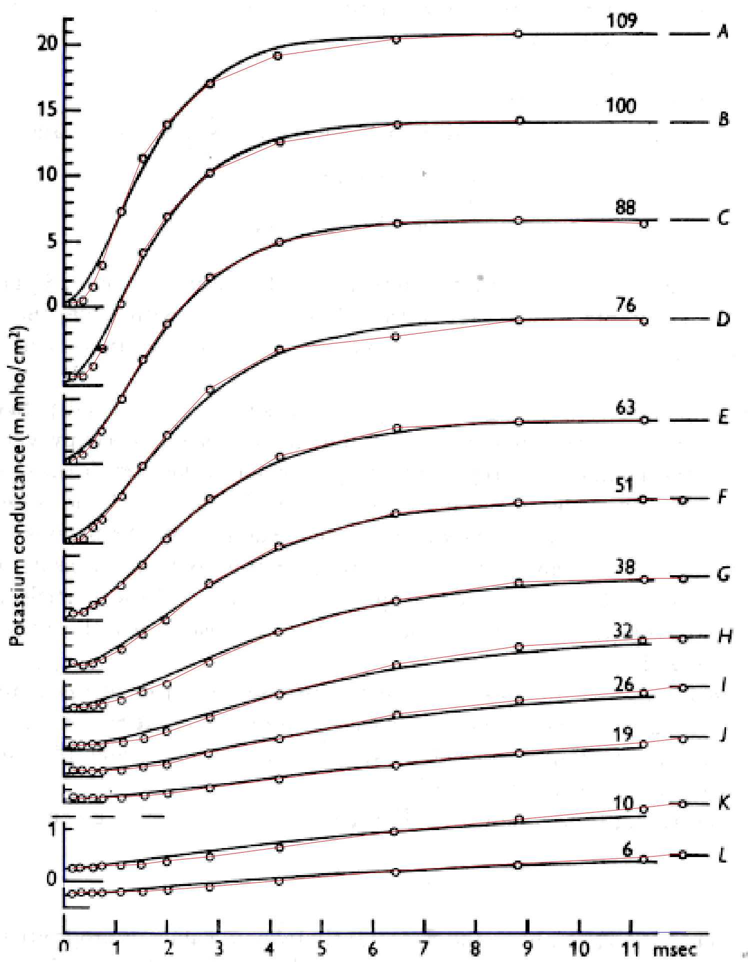
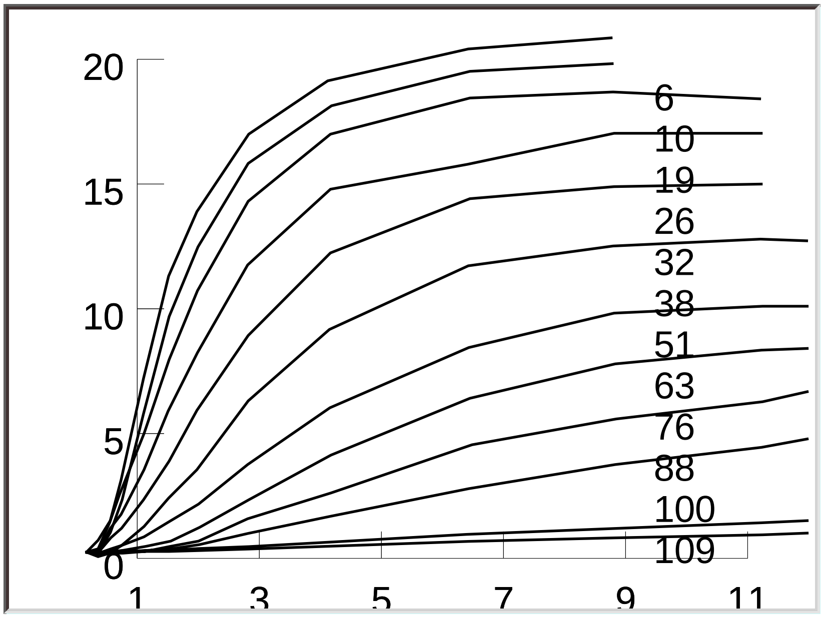


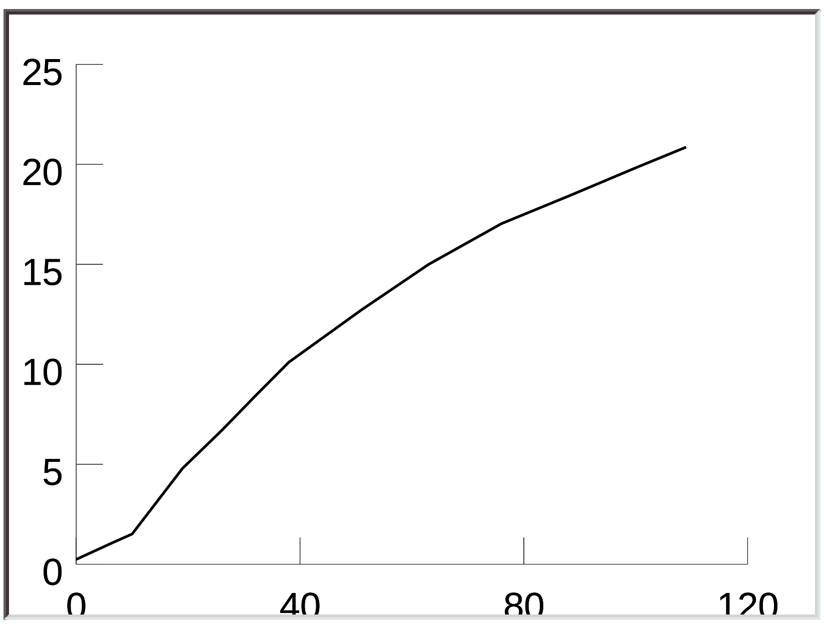
Fig. 3. Rise of potassium conductance associated with different depolarizations. The circles are experimental points obtained on axon 17, temperature 6–7° C, using observations in sea water and choline sea water (see Hodgkin & Huxley, 1952*a*). The smooth curves were drawn from eqn. (11) with $g_{K0} = 0{\cdot}24$ m.mho/cm² and other parameters as shown in Table 1. The time scale applies to all records. The ordinate scale is the same in the upper ten curves (*A* to *J*) and is increased fourfold in the lower two curves (*K* and *L*). The number on each curve gives the depolarization in mV.

Reading the data -- hh508.hoc

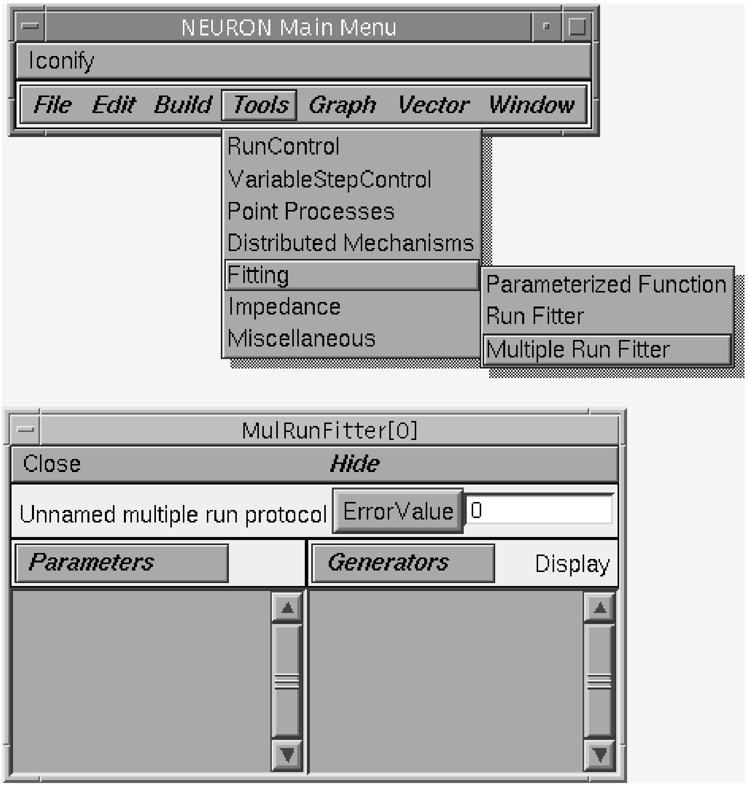
```
objref tobj
tobj = new StringFunctions()
if (tobj.is_name("nrnmainmenu") == 0) {
        xopen("$(NEURONHOME)/lib/hoc/noload.hoc")
}
objref tobj

// read hh508.dat file and display fig 3 of HH paper (page 508)
// also display steady state  conductance as function of clamp voltage

objref vc, gss, g[12], time[12]
strdef tstr

proc readdata() {local i, n, j
//      ropen("hh508.dat")
        vc = new Vector(13)
        gss = new Vector(13)
        for i=0, 11 {
                vc.x[i+1] = fscan()
                n = fscan()
                g[i] = new Vector(n)
                time[i] = new Vector(n)
                sprint(tstr, "%g", vc.x[i+1])
                g[i].label(tstr)
                for j=0, n-1 {
                        time[i].x[j] = fscan()
                        g[i].x[j] = fscan()
                }
                g[i].add(.24 - g[i].x[0])
                gss.x[i+1] = g[i].x[n-1]
        }
        vc.x[0] = 0  gss.x[0] = .24
        ropen()
}

objref g1, g2
g1 = new Graph()
g1.size(0,11,0,20)
g2 = new Graph()
g2.size(0,120,0,25)
{
readdata()
for i=0, 11 g[i].plot(g1, time[i])
gss.line(g2, vc)
}

// following is the hh.dat file
6
13
0.148767 0.280702
0.323787 0.298246
...
109
11
0.148767 0.310345
...
8.786 20.931
```


Conductance data in NEURON





NEURONMainMenu/Tools/ Fitting/MultipleRunFitter

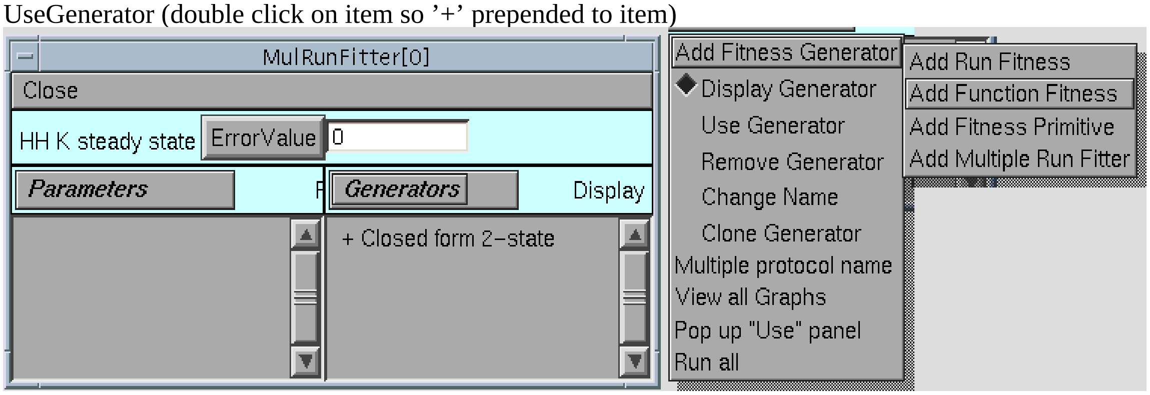


Define a FunctionFitness generator for a 2-state steady state Boltzmann distribution

Multiple protocol name --- HH K steady state

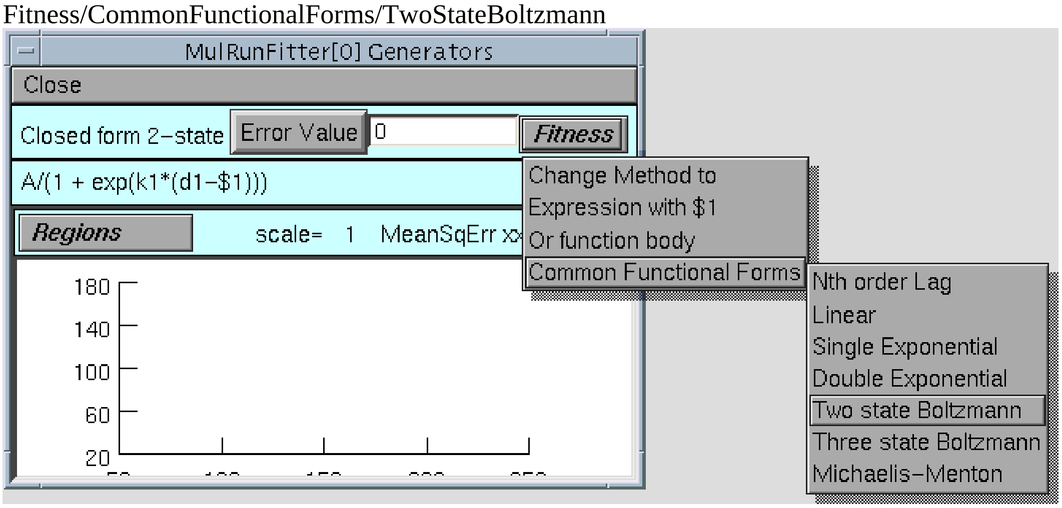
Declare the Generator type
AddFitnessGenerator/AddFunctionFitness
Change Name (double click on item)-- Closed form 2-state

UseGenerator (double click on item so '+' prepended to item)

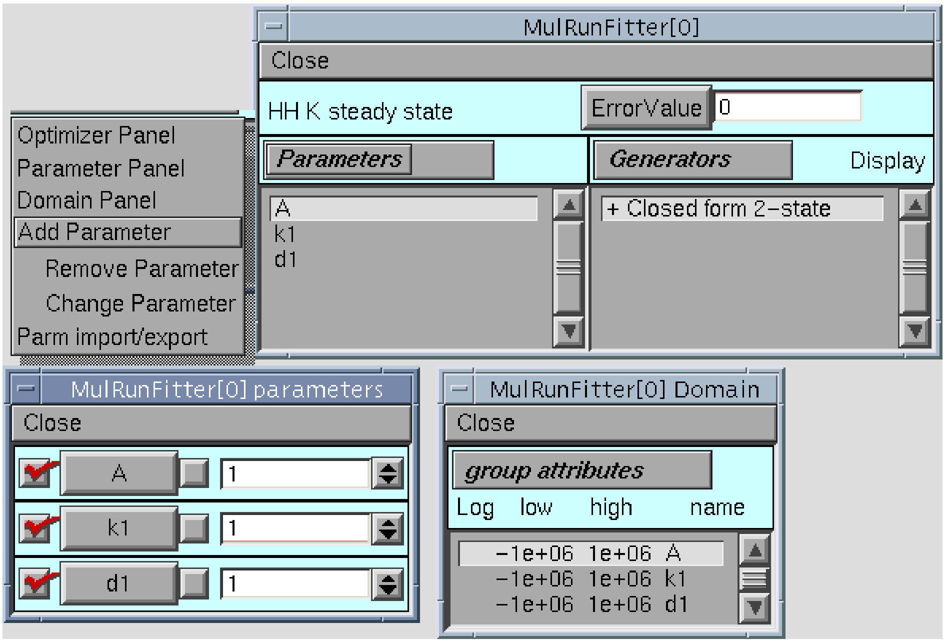


Display Generator (double click on item)

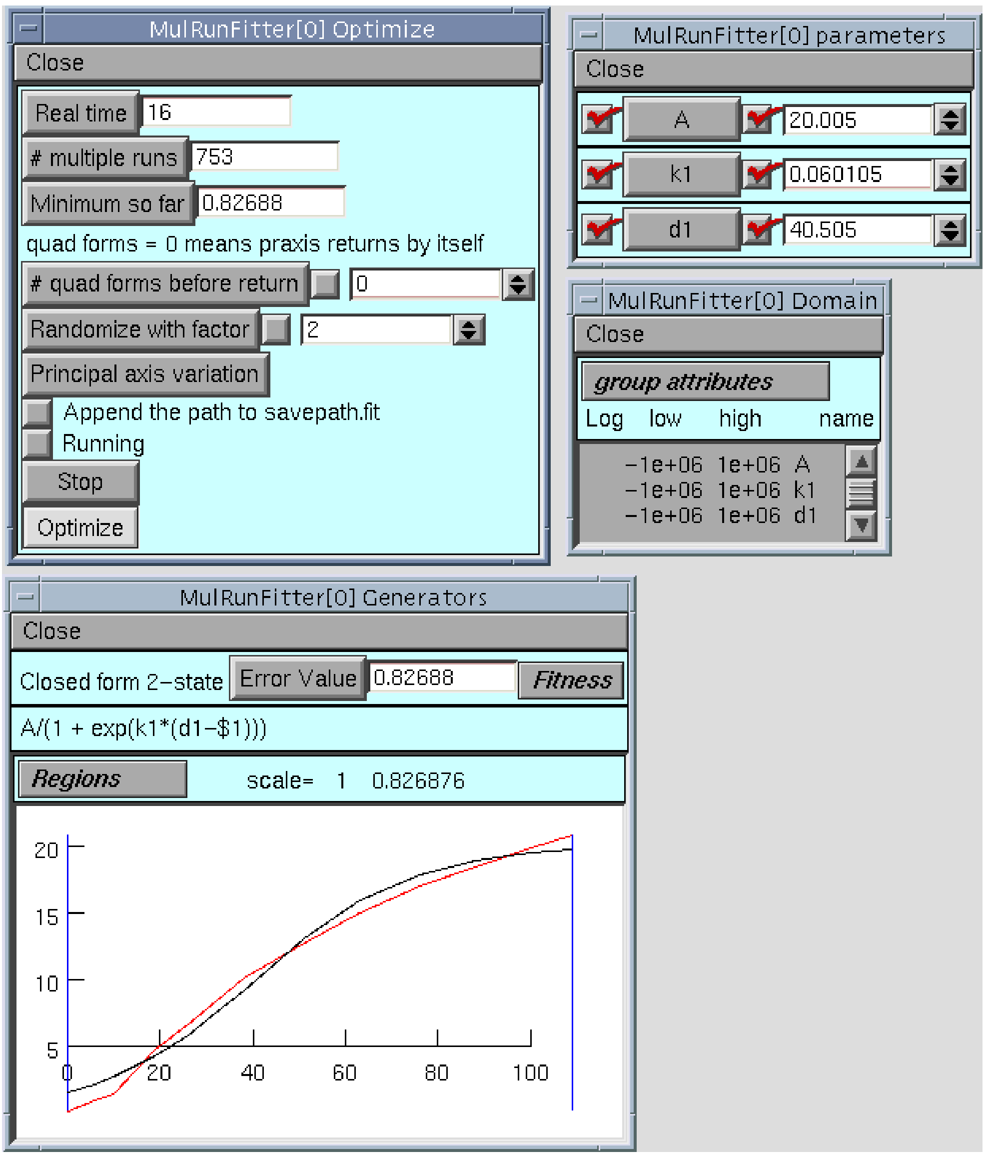
Specify the expression for the FunctionFitness generator
Fitness/CommonFunctionalForms/TwoStateBoltzmann



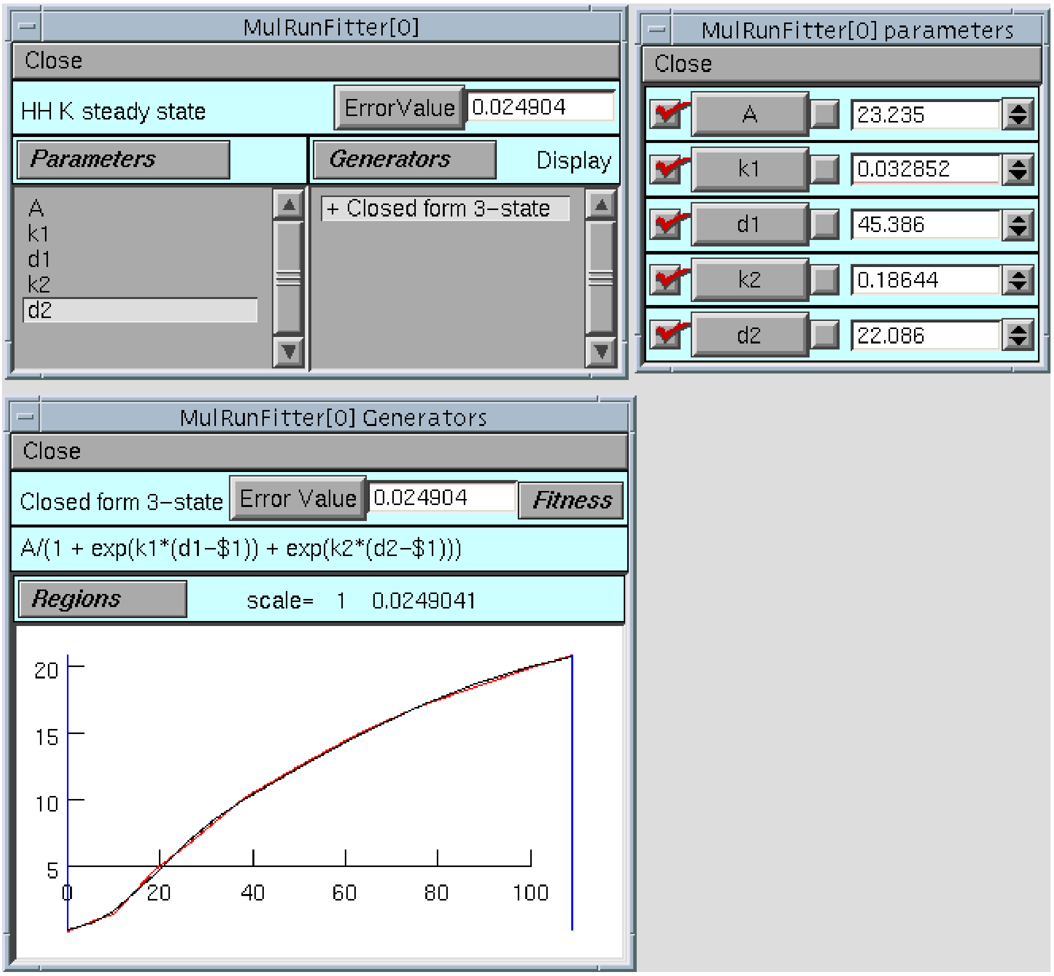
Choose parameters to optimize



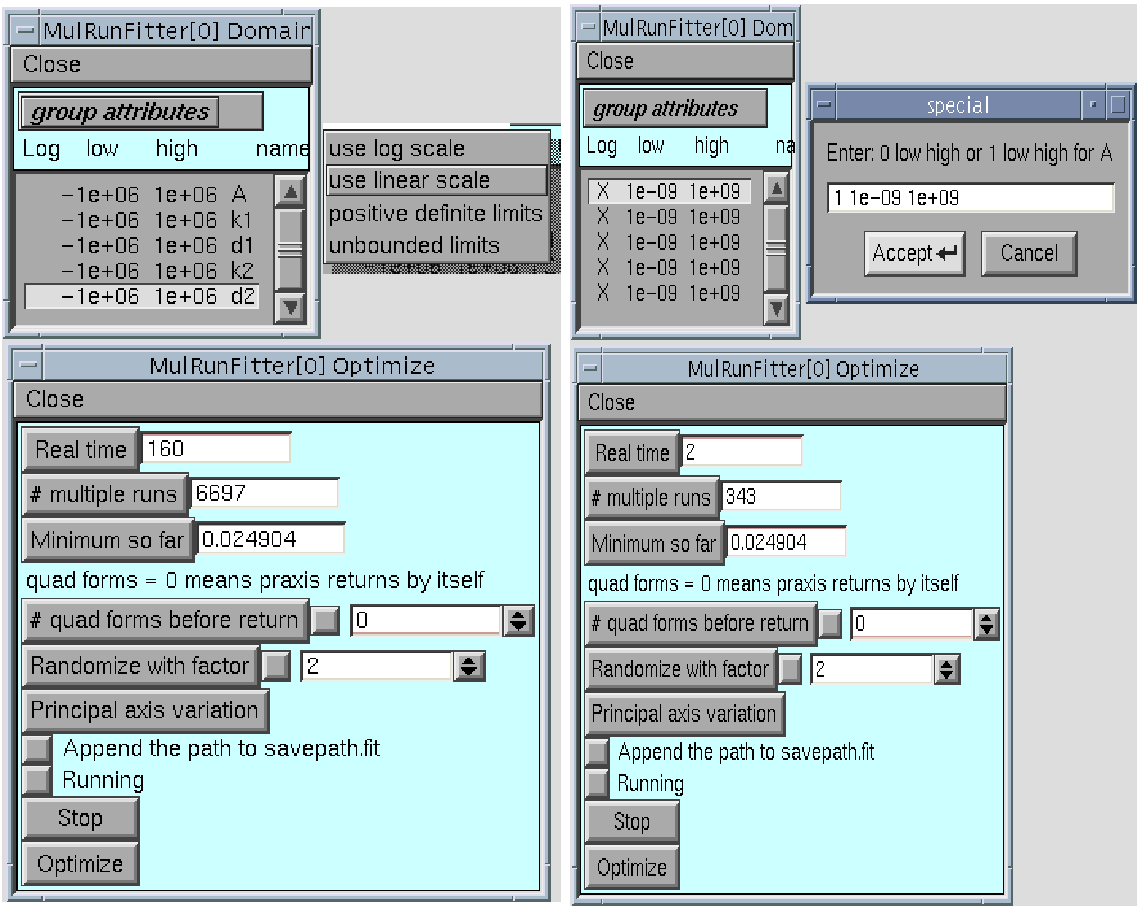
Steady state (two state Boltzmann model)



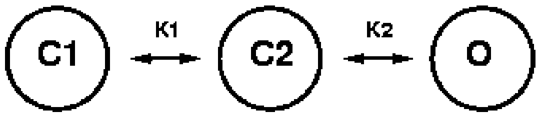
Steady state (three state Boltzmann model)



Log scaling sometimes has higher performance

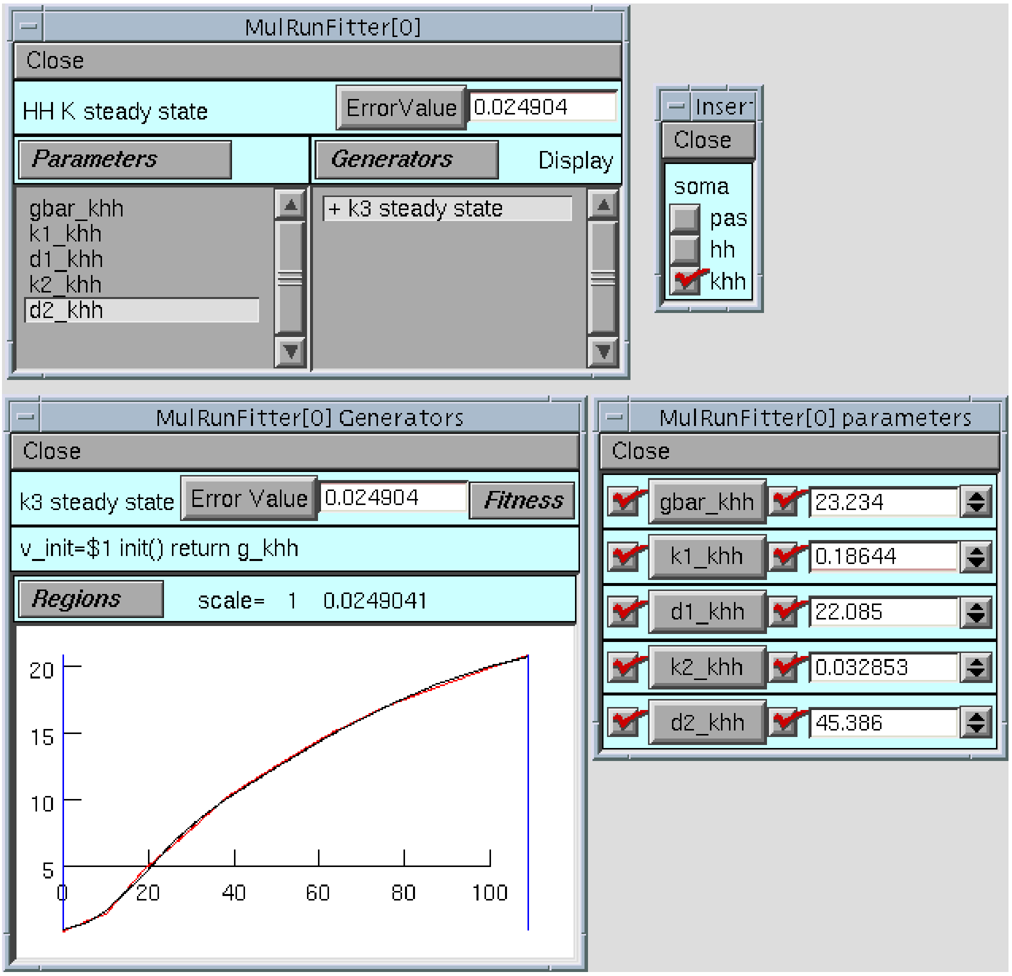


k3.mod -- Three state kinetic scheme

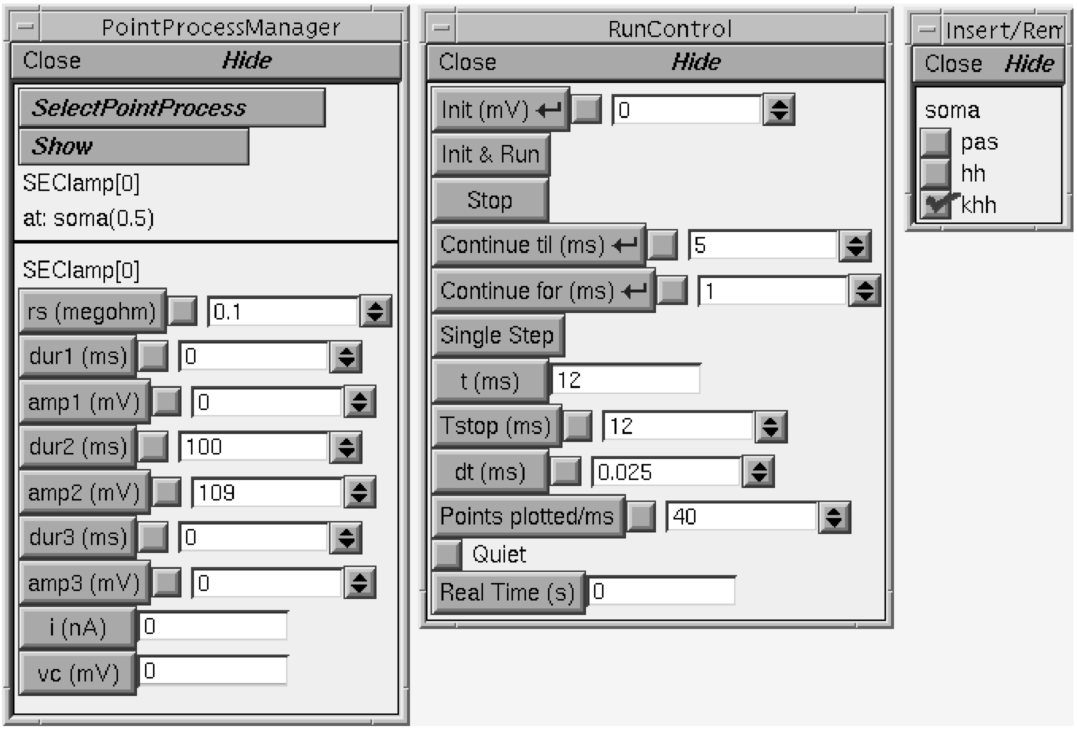



```
NEURON {
  SUFFIX khh
  USEION k READ ek WRITE ik
  RANGE g, gbar
  GLOBAL a1, b1, a2, b2, K1, K2, tau1, tau2
}

PARAMETER { ... }  ASSIGNED { ... }

STATE {c1 c2  o}

INITIAL { SOLVE kin STEADYSTATE sparse }

BREAKPOINT {
  SOLVE kin METHOD sparse
  g = gbar*o
  ik = g*(v - ek)*(1e-3)
}

KINETIC kin {
  rates(v) :
  ~ c1 <-> c2  (a1, b1)
  ~ c2 <-> o  (a2, b2)
  CONSERVE c1 + c2 + o = 1
}

PROCEDURE rates(v(millivolt)) {
  LOCAL vr
  vr = v - vrest : v = vrest means rates at 0

  K2 = exp(-(k2*(d2 - vr)))
  K1 = exp((k2*(d2 - vr)) - (k1*(d1 - vr)))
  tau1 = ta1*exp(tk1*vr)  tau2 = ta2*exp(tk2*vr)

  a1 = K1/(tau1*(K1+1))  b1 = 1/(tau1*(K1+1))
  a2 = K2/(tau2*(K2+1))  b2 = 1/(tau2*(K2+1))
}
```

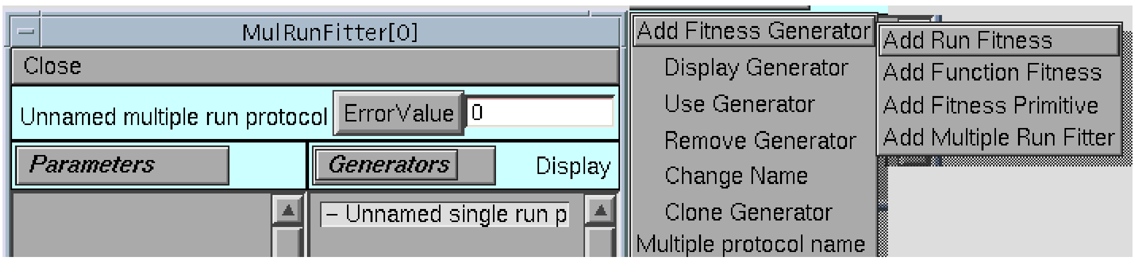

Steady state fit of k3.mod



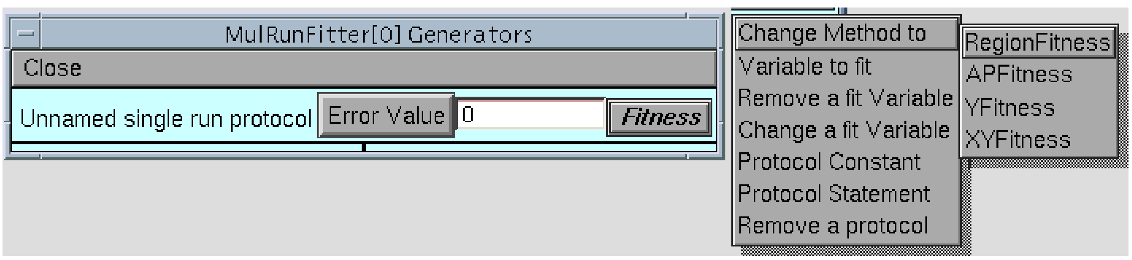
Voltage clamp setup



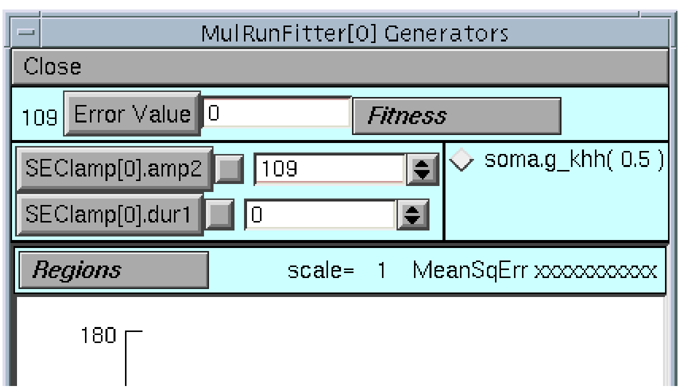
Setting up a RunFitness generator



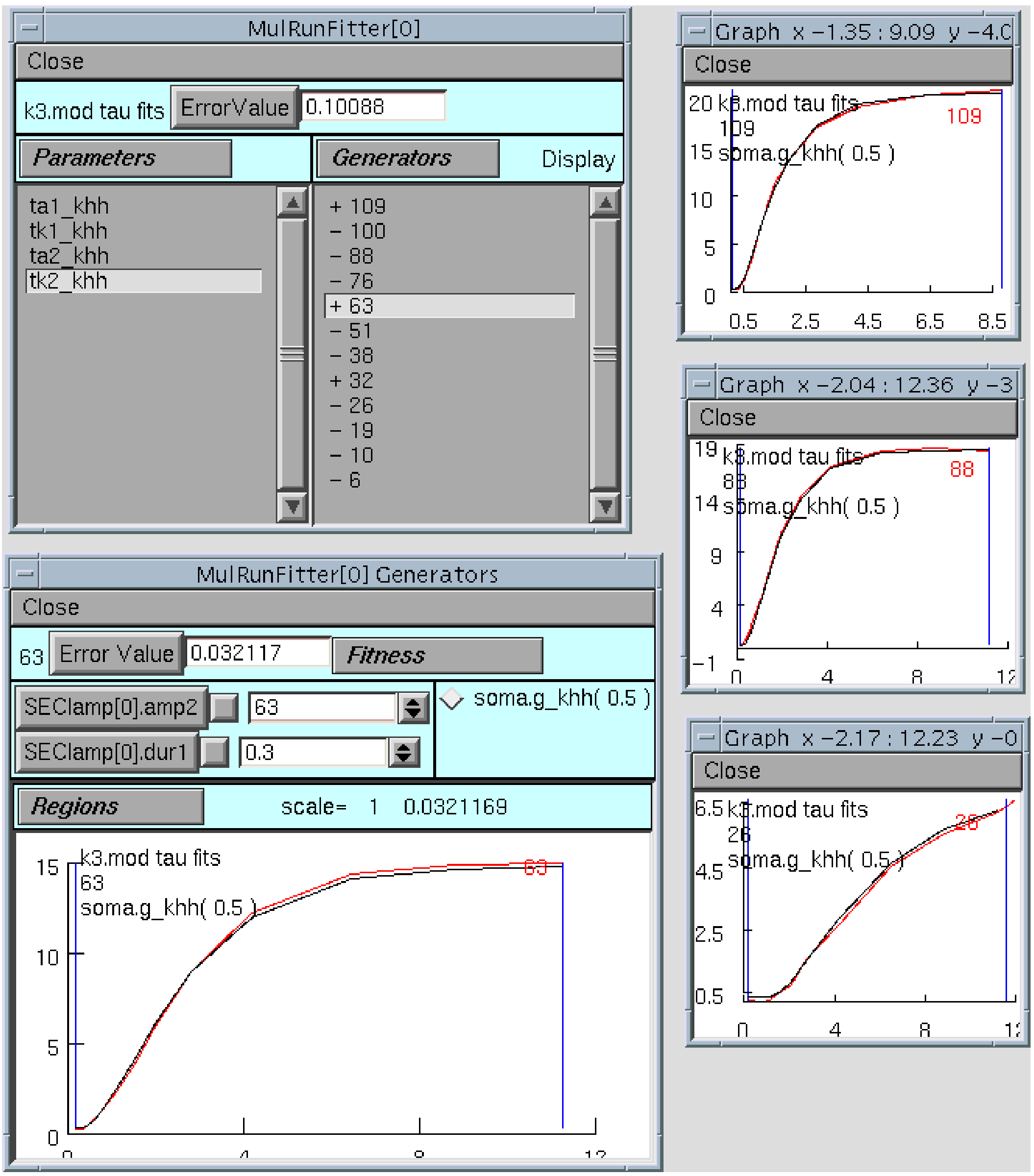
A RunFitness generator does an "Init&Run" to compute the difference between model and data.



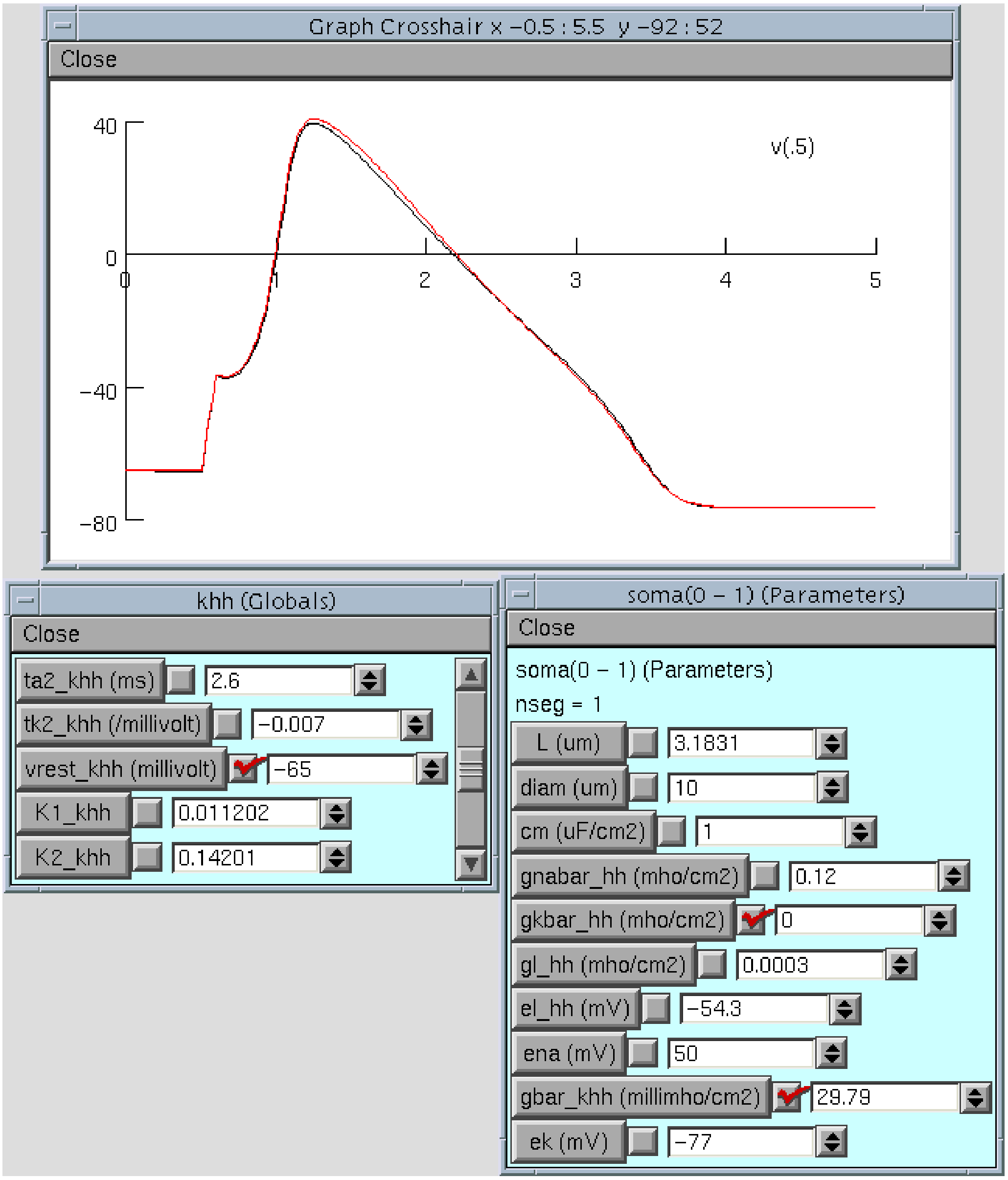
Protocol constants distinguish one run from another.
Variable to fit is some variable computed during a run.



Temporal fit of voltage clamp family



Comparing the action potential



Computational Modeling and Neuroscience

Does computational modeling have a role in neuroscience research?

Best Practices

Know the literature

Collaborate with experimentalists

Use Occam’s razor

Adhere to scientific method

Scientific Method

Observation

Hypothesis

Prediction

Verification

Evaluation

The ideal: Reproducibility

"Reproducibility is the cornerstone
of scientific method."

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

Harsh reality: Velilind's Laws of Experimentation

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

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

http://senselab.med.yale.edu/senselab/modeldb/

SenseLab ModelDB NeuronDB

ModelDB provides an accessible location for storing and efficiently retrieving compartmental neuron models. ModelDB is tightly coupled with NeuronDB. Models can be coded in any language for any environment, though ModelDB has been initially constructed for use with NEURON and GENESIS. Model code can be viewed before downloading and browsers can be set to auto-launch the models. Help

- Search for models by author name [ ] Search
- List models sorted by first author, by each author or by model name
- Find models of a particular Neuron type
- Find models containing a particular Property: Currents, Receptors, or Transmitters
- Find models that relate to a Concept, e.g. synaptic plasticity, pattern recognition, etc.
- Find models that run in a particular Simulation environment
- List models of: Networks, Neurons, Synapses (and ligand-gated ion channels), Neuromuscular Junctions, Axons, voltage-gated Ion Channels
- Find models containing the following words [ ] Search ☐ Case Sensitive
- Search for publications

ModelDB can accommodate models from a wide range of simulation environments

SenseLab ModelDB

Find Models by Simulation Environment

Click on a link to show a list of models implemented in that simulation environment or programming language.

Simulation Environment	Homepage	Number of models
BLISS/SYNOD		0
C or C++ program		0
CellML		0
Genesis		1
Genesis (web link to model)		2
L-Neuron		0
MCell		0
Neosim		0
Neuron		93
Neuron (web link to model)		3
Surf-Hippo		0

Search results

Models by moore

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

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

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

Examines the effect of temperature, the taper of the axon hillock, and HH channel density on antidromic spike invasion into the soma and spike initiation under dendritic stimulation.
Reference: Moore JW, Stockbridge N, Westerfield M (1983) On the site of impulse initiation in a neurone. J Physiol 336:301—11 [PubMed]

Citations Citation Browser

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

Model Type:	Neuron;
Cell Type(s):	Spinal motor neuron;
Channel(s):	I Na,t; I K;
Receptor(s):	
Transmitter(s):	
Simulation Environment:	Neuron;
Model Concept(s):	Action Potential Initiation; Simplified Models;

Search NeuronDB for information about: Spinal motor neuron; I Na,t; I K;

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

- \
  - moore83
    - README
    - mosinit.hoc
    - init.hoc
    - start.ses

```
Moore, Stockbridge, and Westerfield. (1983) On the site of impulse initiation
in a neurone.
J. Physiol. 336: 301-311.

This model qualitatively reproduces figures 1-5.
Note that orthodromic stimulus amplitude is considerably different from
that noted in the paper. IClamp[0].amp was chosen to give qualitative similarity.
We attribute minor quantitative differences to the following:
1) The precise site of axon v vs t curve is not specified. We plot
axon.v(0.25).
2) The antidromic stimulus was unspecified.

The NEURON implementation of this model was prepared by Michael Hines.
Questions about details of this implementation should be addressed to him
at michael.hines@yale.edu.
```

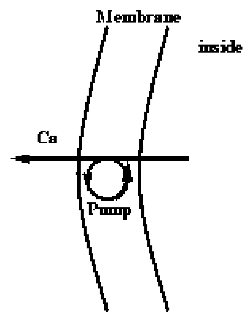
How to proceed

- Read model notes, abstract, paper
- Download and extract the zip file
- Compile the mod files
- Run mosinit.hoc and see what it does
- Figure out what's there and how it works
  - topology(), Shape plot
  - forall psection()
  - analyze hoc code
- Look for reusable components

model specification	topology and geometry
	mechanisms
interface specification	parameter control
	instrumentation
	run control

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

Rate limited active transport of calcium.




```
KINETIC pmp {
        ~ cabulk <-> cam                (1/tau, 1/tau)
        ~ cam + pump <-> capump         (k1, k2)
        ~ capump <-> cao + pump         (k3, k4)
        ica_pmp = 2*FARADAY*(f_flux - b_flux)

        ~ cam << -(ica) : there is a problem here

        COMPARTMENT width {cam} : volume in (um)
        COMPARTMENT 1 {pump capump} : area is dimensionless
        COMPARTMENT 1(m) {cao cabulk}
}
```

Declarations for capump.mod

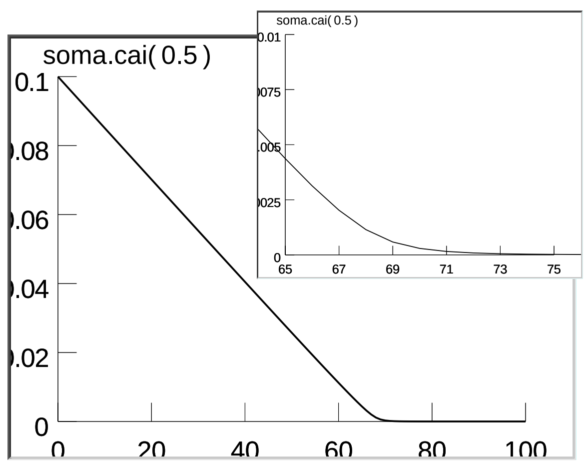
```
NEURON {
        SUFFIX capmp
        USEION ca READ cao, ica, cai WRITE cai, ica
        RANGE tau, width, cabulk, ica, pump0
}

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

PARAMETER {
        width = 0.1 (um)
        tau = 1 (ms)
        k1 = 5e8        (/mM-s)
        k2 = 0.25e6     (/s)
        k3 = 0.5e3      (/s)
        k4 = 5e0        (/mM-s)
        cabulk = 0.1 (uM)
        pump0 = 3e-14 (mol/cm2)
}

ASSIGNED {
        cao (mM) : 10
        cai (mM) : 1e-3
        ica (mA/cm2)
        ica_pmp (mA/cm2)
        ica_pmp_last (mA/cm2)
}

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

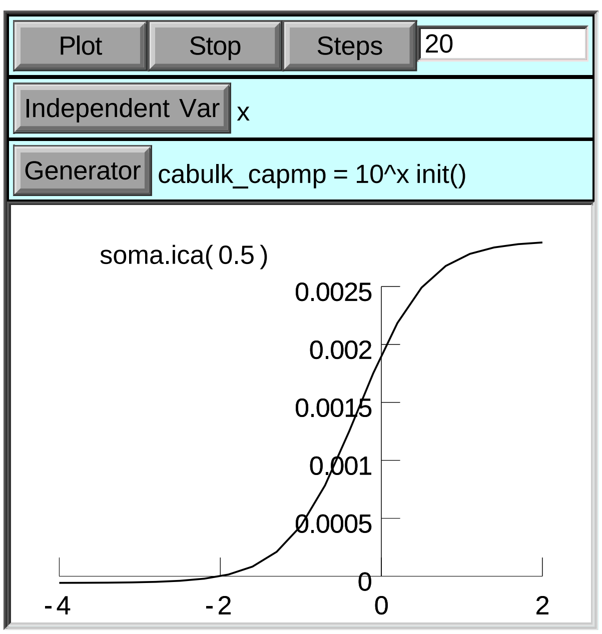

Equations for capump.mod

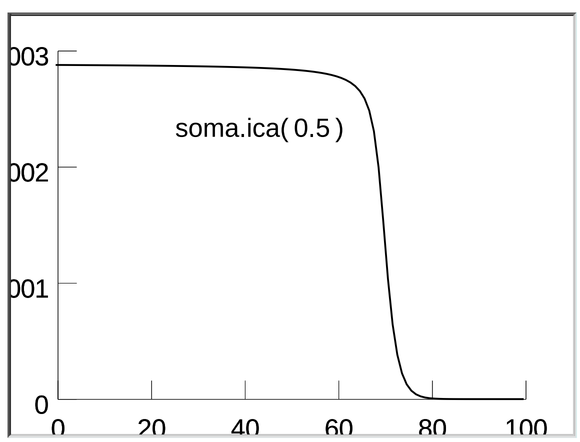
```
INITIAL {
    ica = 0
    ica_pmp = 0
    ica_pmp_last = 0
    SOLVE pmp STEADYSTATE sparse
}

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

KINETIC pmp {
    ~ cabulk <-> cam (width/tau, width/tau)
    ~ cam + pump <-> capump ((1e7)*k1, (1e10)*k2)
    ~ capump <-> cao + pump ((1e10)*k3, (1e10)*k4)
    ica_pmp = (1e-7)*2*FARADAY*(f_flux - b_flux)

    : ica_pmp_last vs ica_pmp needed because
    : of STEADYSTATE calculation
    ~ cam << (-(ica - ica_pmp_last)/(2*FARADAY)*(1e7))

    CONSERVE pump + capump = (1e13)*pump0
    COMPARTMENT width {cam}     : volume has dimensions of um
    COMPARTMENT (1e13) {pump capump} : area is dimensionless
    COMPARTMENT 1(um) {cabulk}
    COMPARTMENT (1e3)*1(um) {cao}

    cai = (0.001)*cam
}
```


Testing capump.mod

```
load_file("single.hoc")

// define a replacement for the stdrun.hoc version of
// proc init() {
//      finitialize(v_init)
//      fcurrent()
// }
// that lets you escape from the tyranny of the
// steady state initialization of cai.

proc init() { local savtau
        // will initialize cai to cabulk
        savtau = tau_capmp
        tau_capmp = 1e-6
        finitialize(v_init)
        tau_capmp = savtau
        fcurrent()
}
```

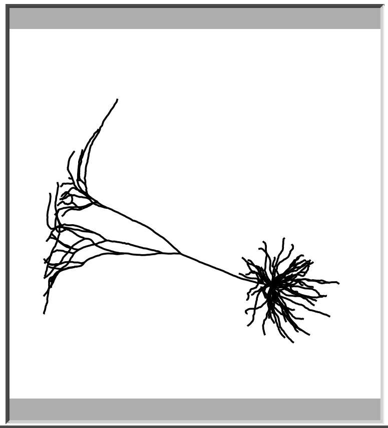


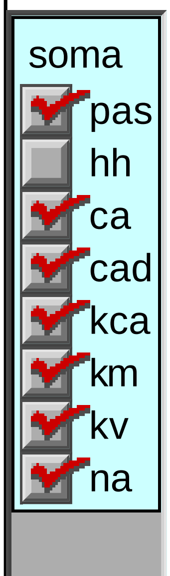


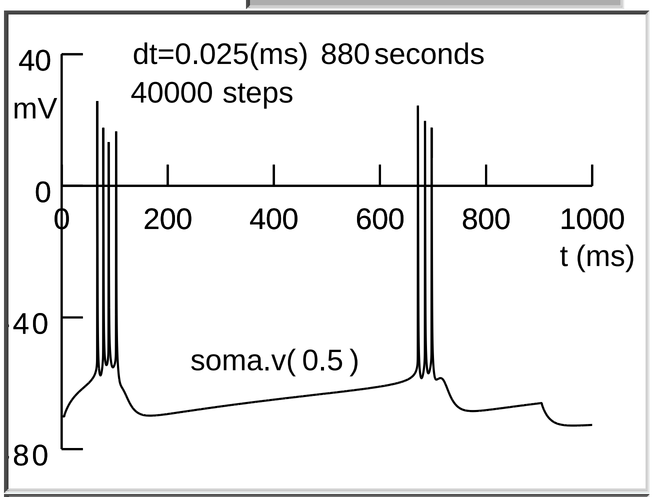


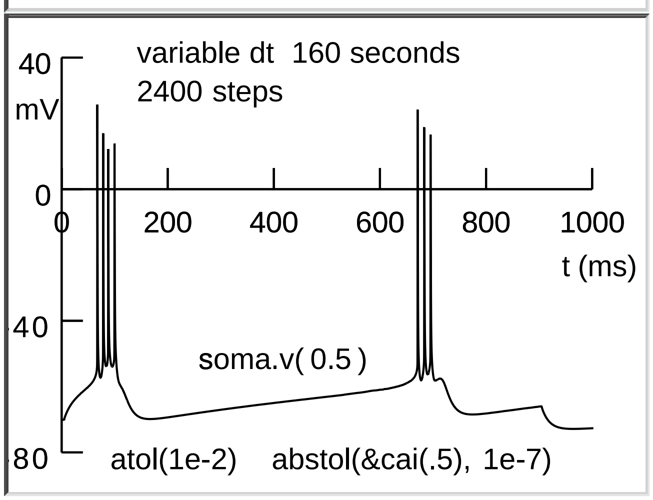
L5 Pyramid demo

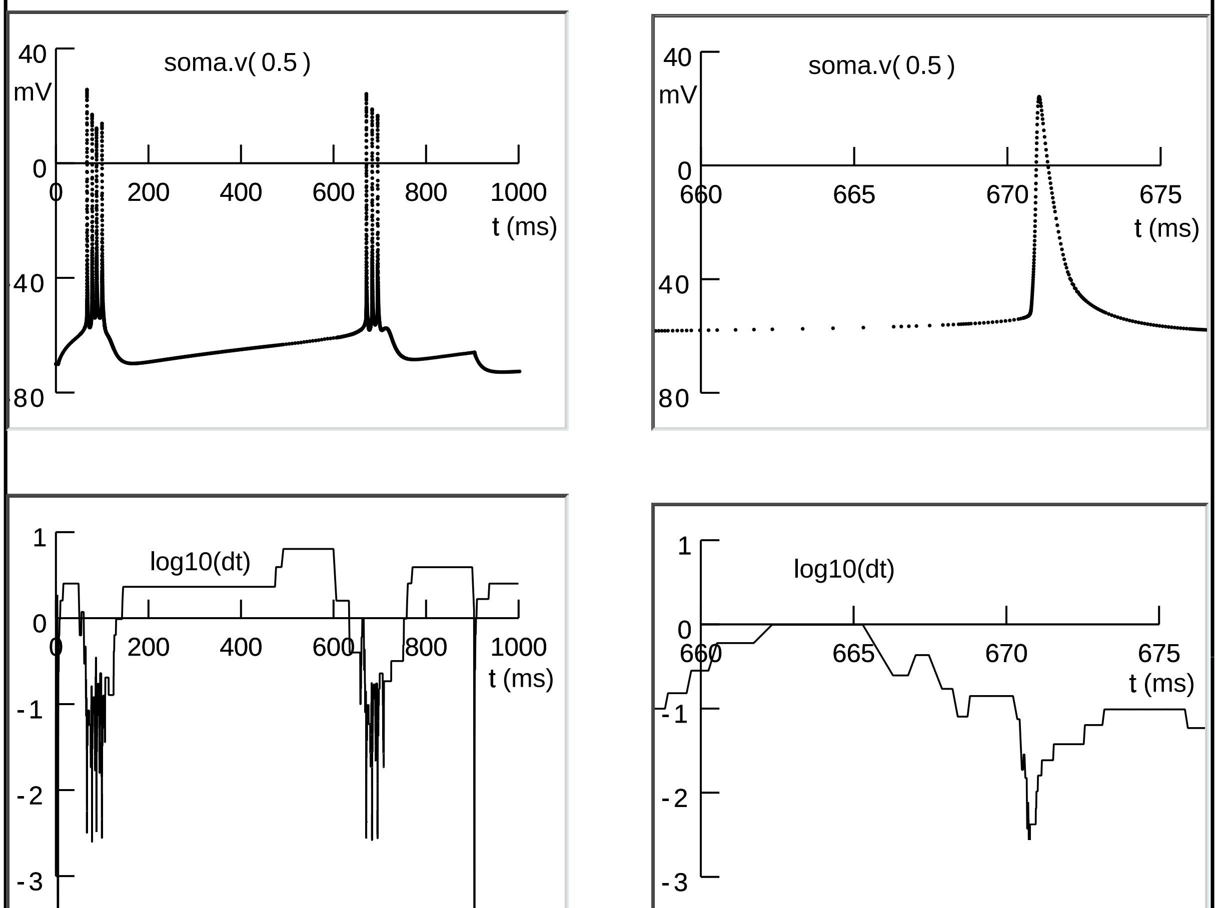
From:
Z. F. Mainen and T. J. Sejnowski (1996)
Influence of dendritic structure on
firing pattern in model neocortical
neurons. Nature 382: 363-366.



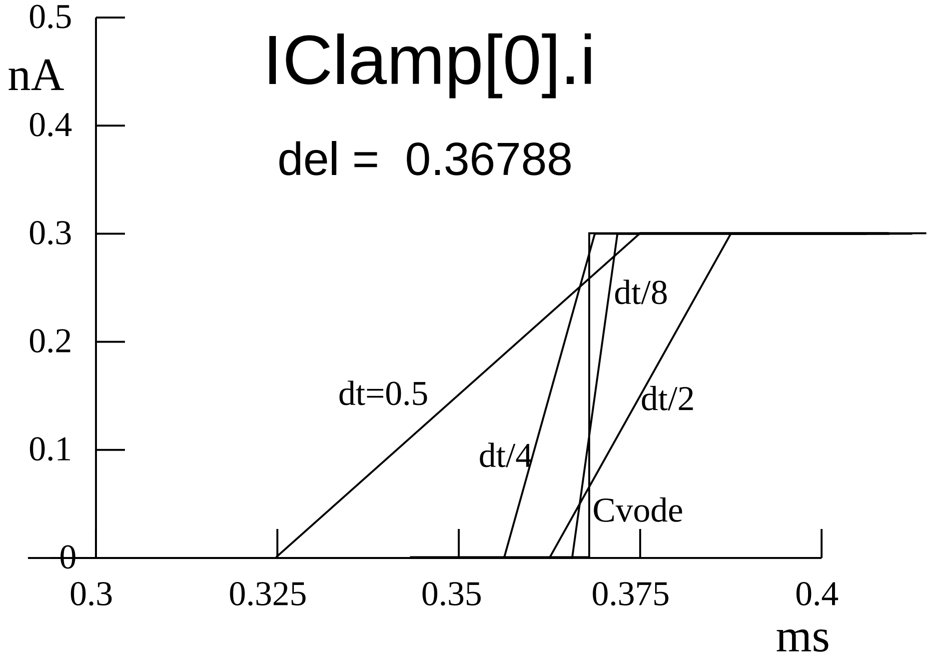


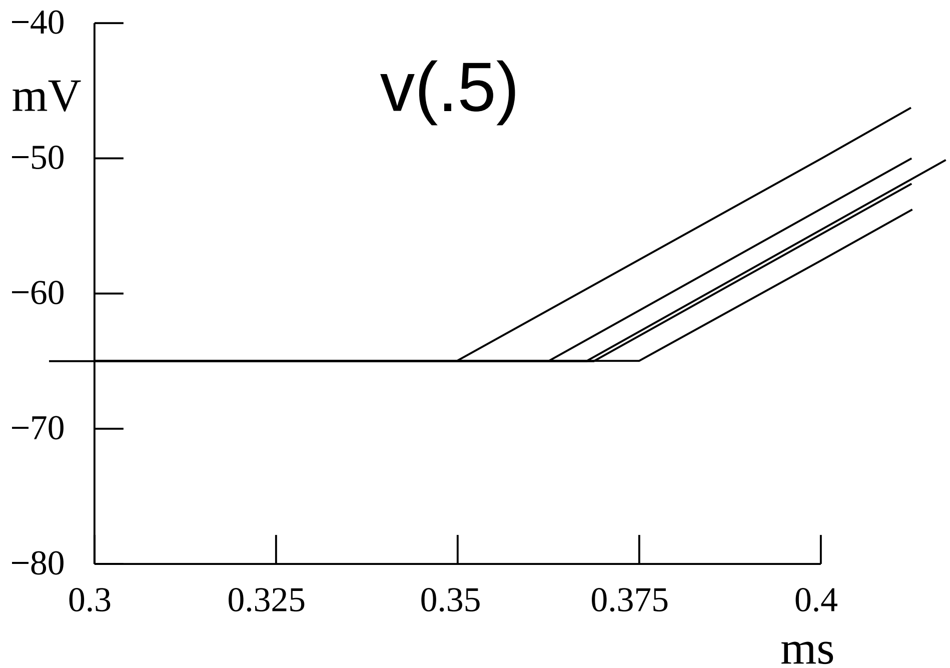






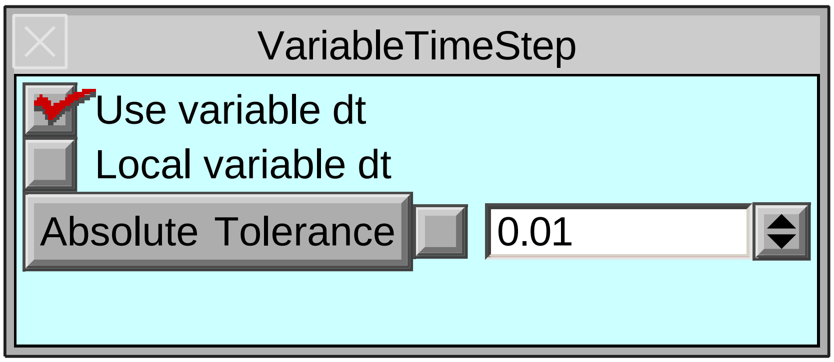
soma.v(0.5)
40
mV
0
0
200
400
600
800
1000
t (ms)
40
80
soma.v(0.5)
40
mV
0
660
665
670
675
t (ms)
40
80
log10(dt)
1
0
0
200
400
600
800
1000
t (ms)
-1
-2
-3
log10(dt)
1
0
660
665
670
675
t (ms)
-1
-2
-3


IClamp[0].i
del = 0.36788
nA
0.5
0.4
0.3
0.2
0.1
0
dt=0.5
dt/4
dt/8
dt/2
Cvode
0.3
0.325
0.35
0.375
0.4
ms


v(.5)
mV
−40
−50
−60
−70
−80
0.3
0.325
0.35
0.375
0.4
ms

Control for Variable time step method

GUI control



HOC control

`cvode = new CVode()` instance managed by stdrun.hoc

```
cvode_active(0 or 1)
cvode_local(0 or 1)
```

Error Control

$$e_i \; < \; RTOL \cdot |y_i| + ATOL_i$$

```
cvode.rtol(0)

cvode.atol(1e-2)

cvode.atolscale(&cai(.5), 1e-8)
```

```
STATE {
    cai      (mM)        <1e-8>
    pump     (mol/cm2)   <1e-17>
    pumpca   (mol/cm2)   <1e-17>
}
```


Use DERIVATIVE block for hh-like channels

```
BREAKPOINT {
        SOLVE states METHOD cnexp
        ina = gnabar*m^3*h * (v - ena)
}

DERIVATIVE states {
        rates(v)
        m' = (minf-m)/mtau
        h' = (hinf-h)/htau
}
```

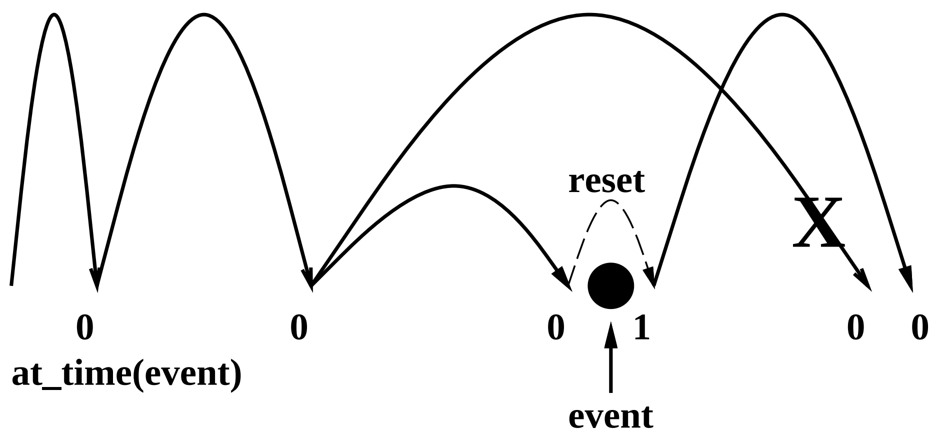
```
PROCEDURE states() {
        rates(v)
        m = m + m_exp * (m_inf - m)
        h = h + h_exp * (h_inf - h
}

        m_exp = 1 - Exp(-dt/tau_m)
```

Events with CVODE

Generated internally by the model

```
INITIAL {
    i = 0
}

BREAKPOINT {
    if (at_time(del)) {
        i = amp
    }
    if (at_time(del+dur)) {
        i = 0
    }
}
```

```
INITIAL {
    i = 0
}

BREAKPOINT {
    at_time(del)  at_time(del+dur)
    if (t >= del && t < del + dur) {
        i = amp
    }else{
        i = 0
    }
}
```



Externally generated event

```
NET_RECEIVE(value) {
   i = value
}
```

```
NET_RECEIVE(value) {
   if (flag == 0) {
      i = i + value
      net_send(dur, 1)
   }else{
      i = i - value
   }
}
```

Abrupt change in state

```
STATE { A (uS) G (uS) }

BREAKPOINT {
   if (gmax && at_time(onset)) {
      state_discontinuity(A, A + E*gmax)
   }
   SOLVE state METHOD sparse
   i = G*(v - e)
}

KINETIC state {
   ~ A <-> G       (k, 0)
   ~ G ->               (k)
}

NET_RECEIVE(weight) {
   state_discontinuity(A, A + E*weight)
}
```


Communication between cells

Gap junctions

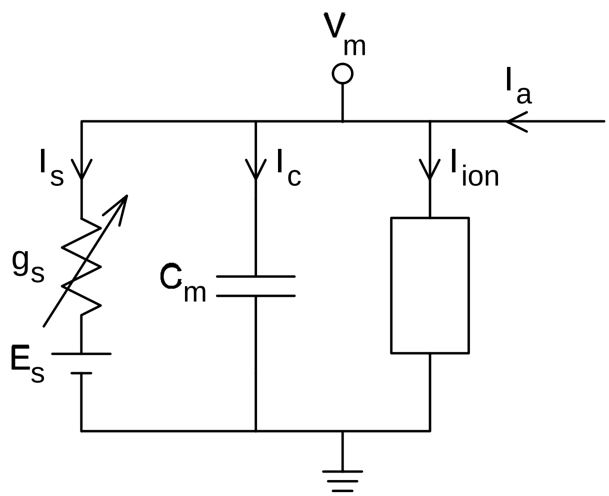
Synapses

Graded synaptic transmission

Physical system:

A presynaptic variable governs the continuous release of transmitter, which in turn modulates some property of the postsynaptic cell.

Conceptual model:



where g_s is a function of V_{pre}

$$C_m \frac{dV_m}{dt} + I_{ion} = I_a - (V_m - E_s) \cdot g_s(V_{pre})$$

The NEURON Simulation Environment Figure 3

Graded synaptic transmission
continued

Computational implementation of model:

1. Inefficient hack (don't ever do this!)

At each time step, use a hoc statement to update the synaptic mechanism's V_{pre}

```
post_cell.syn.v_pre = pre_cell.axon.v(1)
```

2. More efficient: POINTER variable has same effect as

```
section1.mechanism1.variable1(x1) =
    section2.mechanism2.variable2(x2)
```

but is much faster

NMODL specification of synaptic mechanism

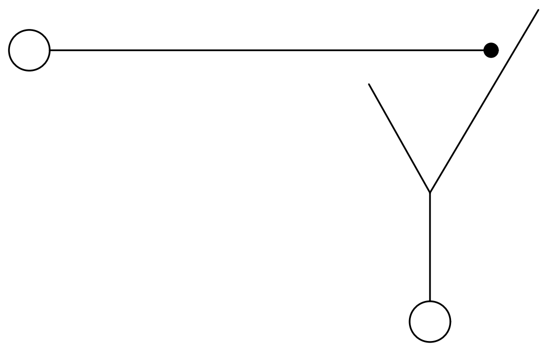
```
NEURON {
    POINT PROCESS Syn
    POINTER v_pre
}
```

hoc usage

```
objref syn
somedendrite syn = new Syn(0.8)
setpointer syn.v_pre, pre_cell.axon.v(1)
```

Spike-triggered synaptic transmission

Physical system:



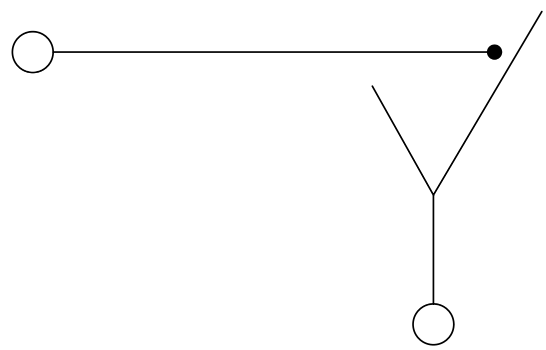
Presynaptic neuron with spike trigger zone and an axon leading to a terminal that makes a synaptic connection onto a postsynaptic cell.

Conceptual model:

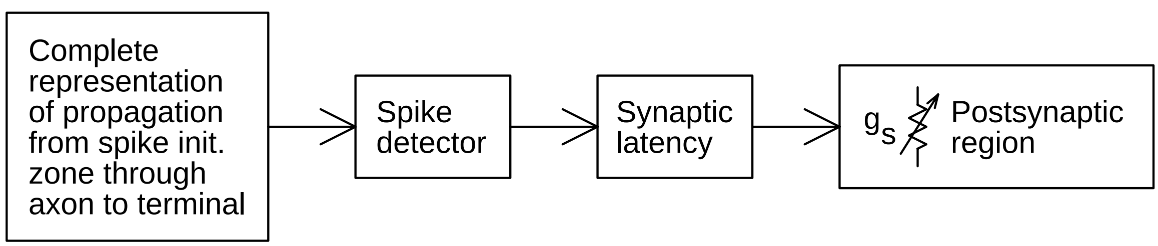
Spike in presynaptic terminal triggers transmitter release; presynaptic details are unimportant

Postsynaptic effect is described by a DE or kinetic scheme that is perturbed by presynaptic spikes

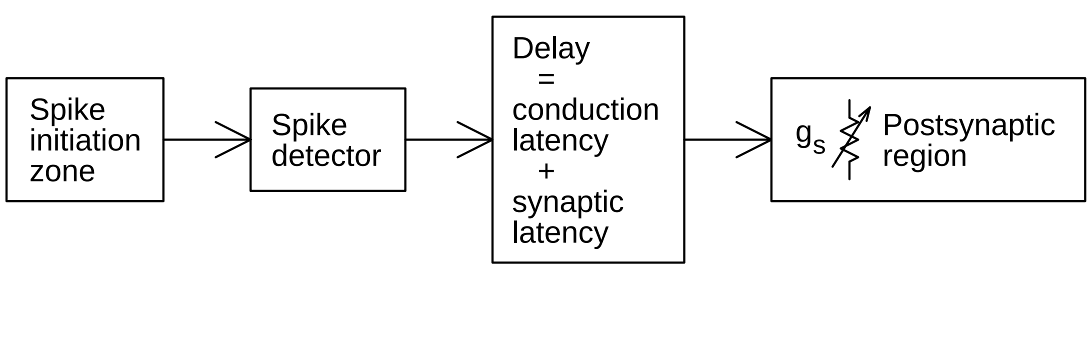
Computational implementation of model of spike-triggered synaptic transmission:



The basic idea



More efficient: "virtual spike propagation"



The NetCon class

hoc usage

```
section netcon = new Netcon(&v(x), target,
    threshold, delay, weight)

netcon = new Netcon(source, target,
    threshold, delay, weight)

section netcon = new Netcon(&v(x), target)

netcon = new Netcon(source, target)
```

Defaults

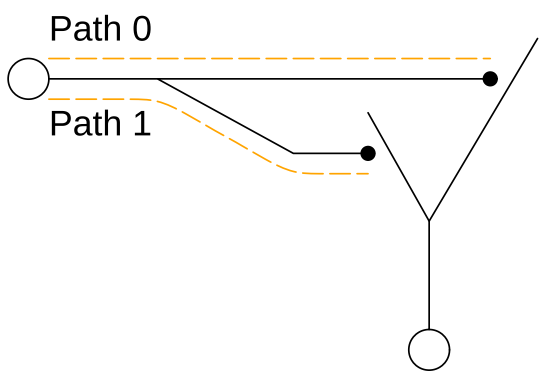
```
threshold = 10

delay = 1  // must be >= 0

weight = 0
```

NMODL specification of synaptic mechanism

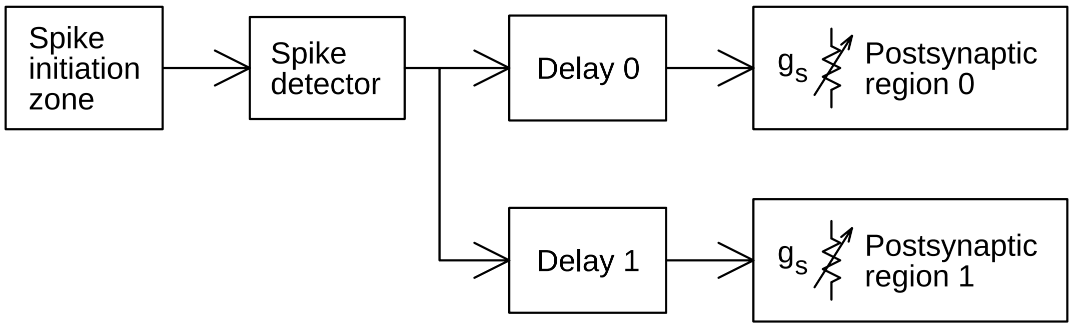
```
NET_RECEIVE(weight(microsiemens)) {
    . . .
}
```


The NEURON Simulation Environment

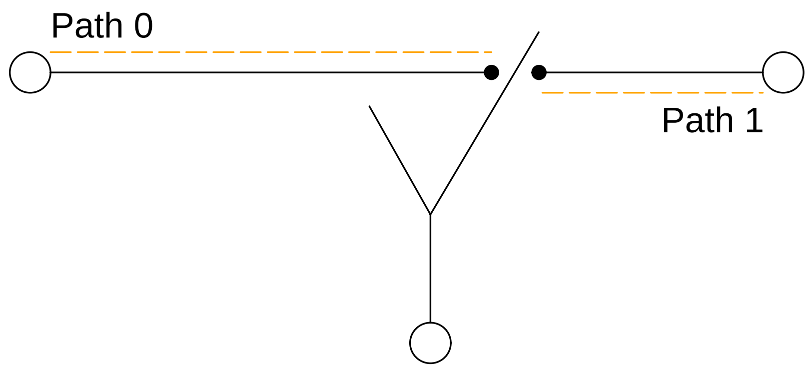
Efficient divergence



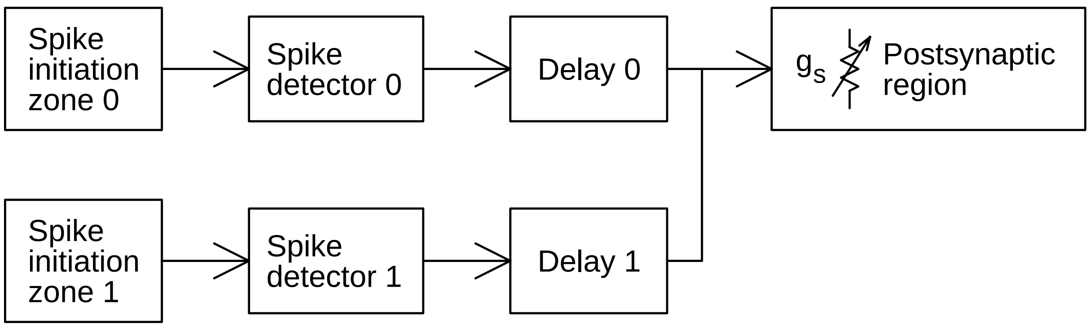
Multiple NetCons with a common source
share a single threshold detector



Efficient convergence



NetCons can share a postsynaptic mechanism
(single equation handles multiple inputs)



The NEURON Simulation Environment Figure 9

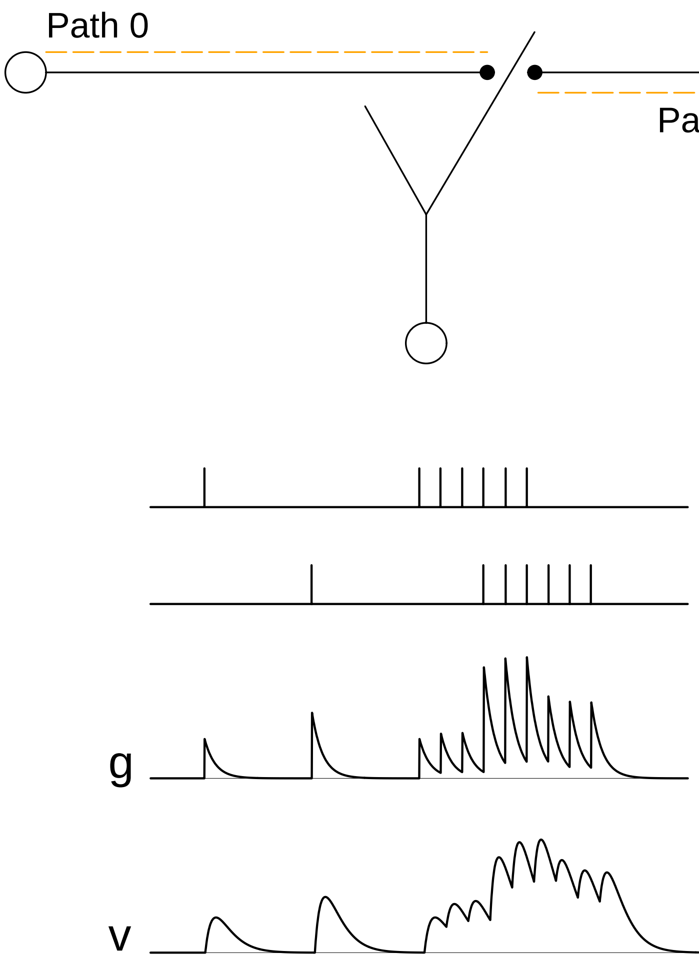
Example: g_s with fast rise and exponential decay

Each synaptic activation produces an abrupt increase of g_s, which then decays with a single time constant

```
NEURON {
     POINT_PROCESS ExpSyn
     RANGE tau, e, i
     NONSPECIFIC_CURRENT i
}

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

ASSIGNED {
     v (millivolt)
     i (nanoamp)
}

STATE { g (micromho) }

INITIAL { g=0 }

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

DERIVATIVE state { g' = -g/tau }

NET_RECEIVE(weight (micromho)) {
     state_discontinuity(g, g + weight)
}
```

The NEURON Simulation Environment

Test of **ExpSyn**



g summates linearly

v shows nonlinear summation

Spike-triggered synaptic transmission

Separate specification of what is connected

from

specification of the postsynaptic mechanism

Network Construction

1) Define the types of cells

2) Create each cell in the network

3) Connect the cells

Define the types of cells

"Real" cell types

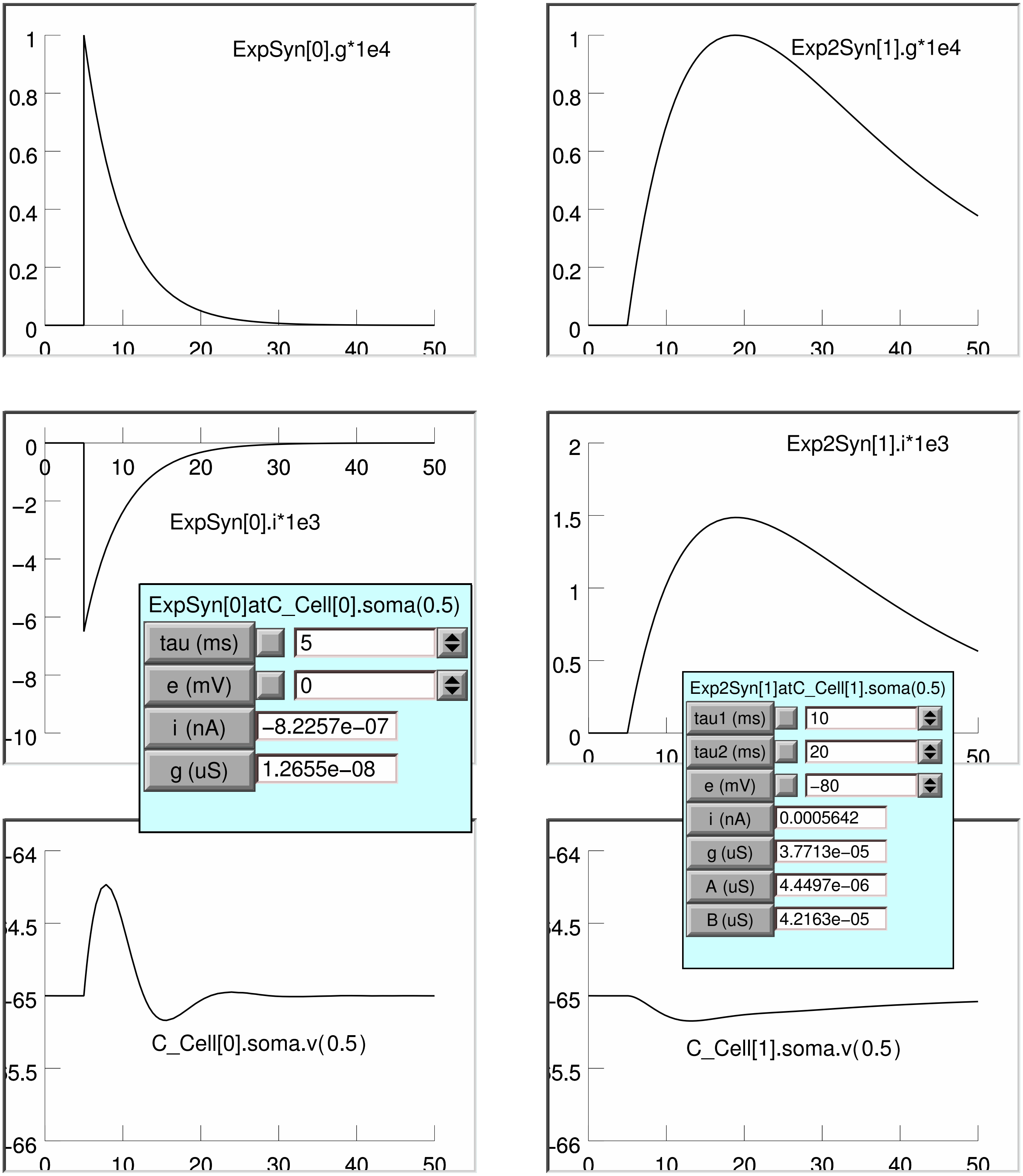
```
Sections + density mechanisms + synapses.

The latter are PointProcesses that
have a NET_RECEIVE block that affects
membrane current.

    ExpSyn
    Exp2Syn
```

Encapsulate in a class

```
begintemplate Cell
    public soma, E, I
    create soma
    objref E, I
    proc init() {
        soma insert hh
        soma { E = new ExpSyn(.5)  I = new ExpSyn(.5) }
        I.e = -80
    }
endtemplate Cell
```

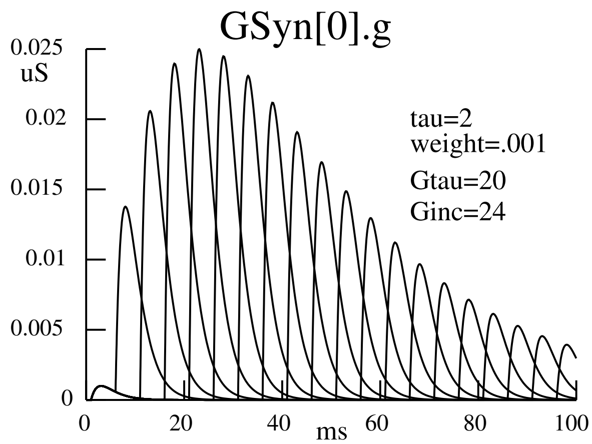
Artificial cell types

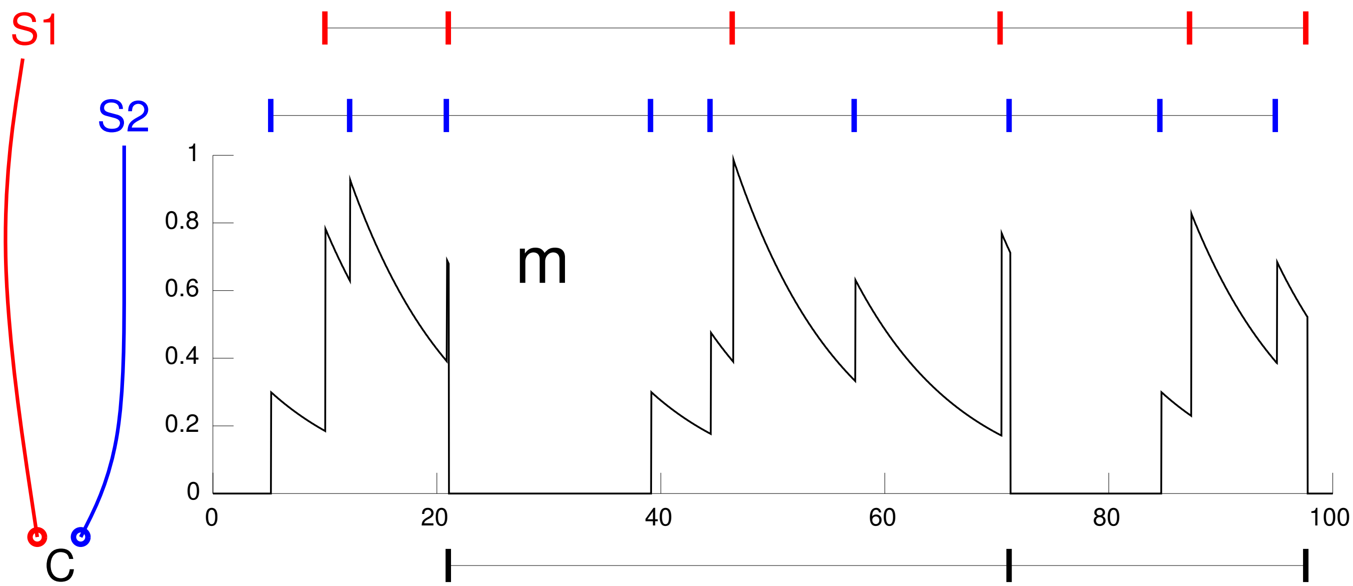
```
PointProcesses that have a NET_RECEIVE
block that calls net_event

    NetStim
    IntFire1
    IntFire2
    IntFire4
```



ExpSyn[0].g*1e4
Exp2Syn[1].g*1e4
ExpSyn[0].i*1e3
Exp2Syn[1].i*1e3
ExpSyn[0]atC_Cell[0].soma(0.5)
tau (ms) 5
e (mV) 0
i (nA) –8.2257e–07
g (uS) 1.2655e–08
Exp2Syn[1]atC_Cell[1].soma(0.5)
tau1 (ms) 10
tau2 (ms) 20
e (mV) –80
i (nA) 0.0005642
g (uS) 3.7713e–05
A (uS) 4.4497e–06
B (uS) 4.2163e–05
C_Cell[0].soma.v(0.5)
C_Cell[1].soma.v(0.5)

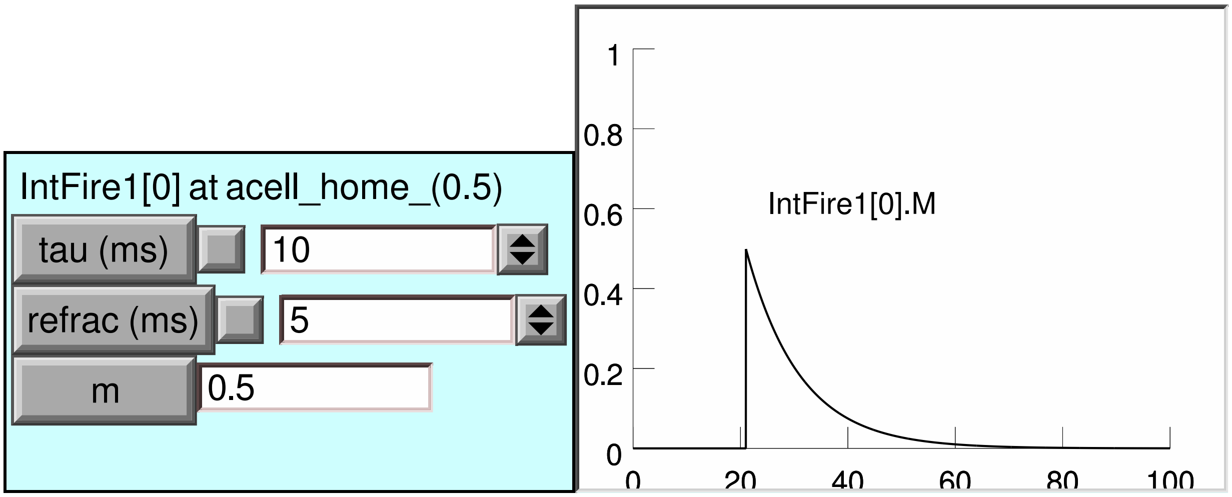
G-Protein synapse -- gsyn.mod

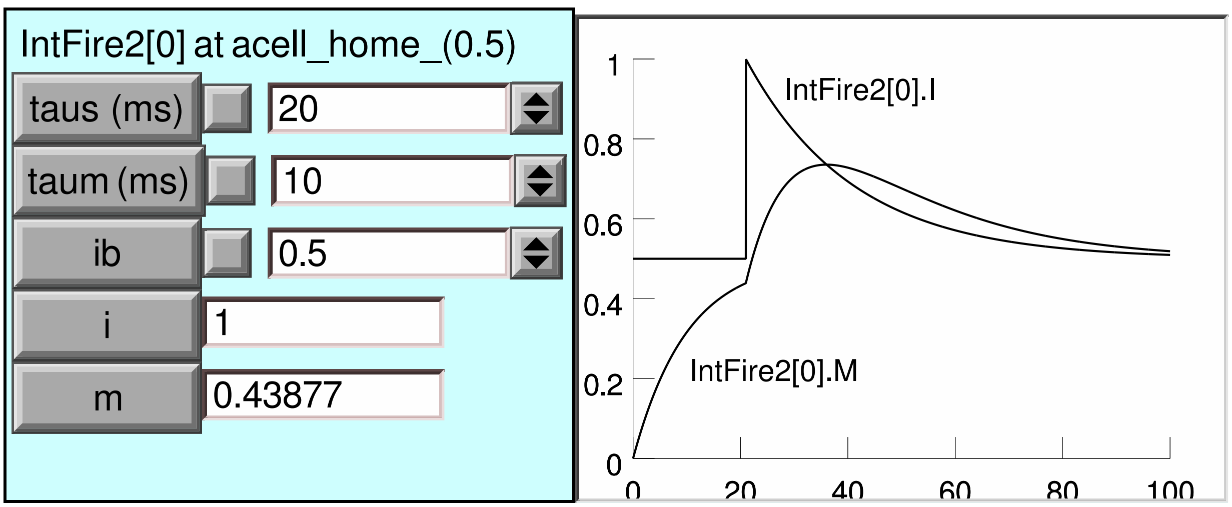
```
NEURON {
        POINT_PROCESS GSyn
        RANGE tau1, tau2, e, i
        RANGE Gtau1, Gtau2, Ginc
        NONSPECIFIC_CURRENT i
        RANGE g
}

PARAMETER {
        tau1=0.1 (ms)
        tau2 = 1 (ms)
        Gtau1 = 20 (ms)
        Gtau2 = 21 (ms)
        Ginc = 1
        e=0     (mV)
}

STATE {
        A (umho)
        B (umho)
}

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

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

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

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

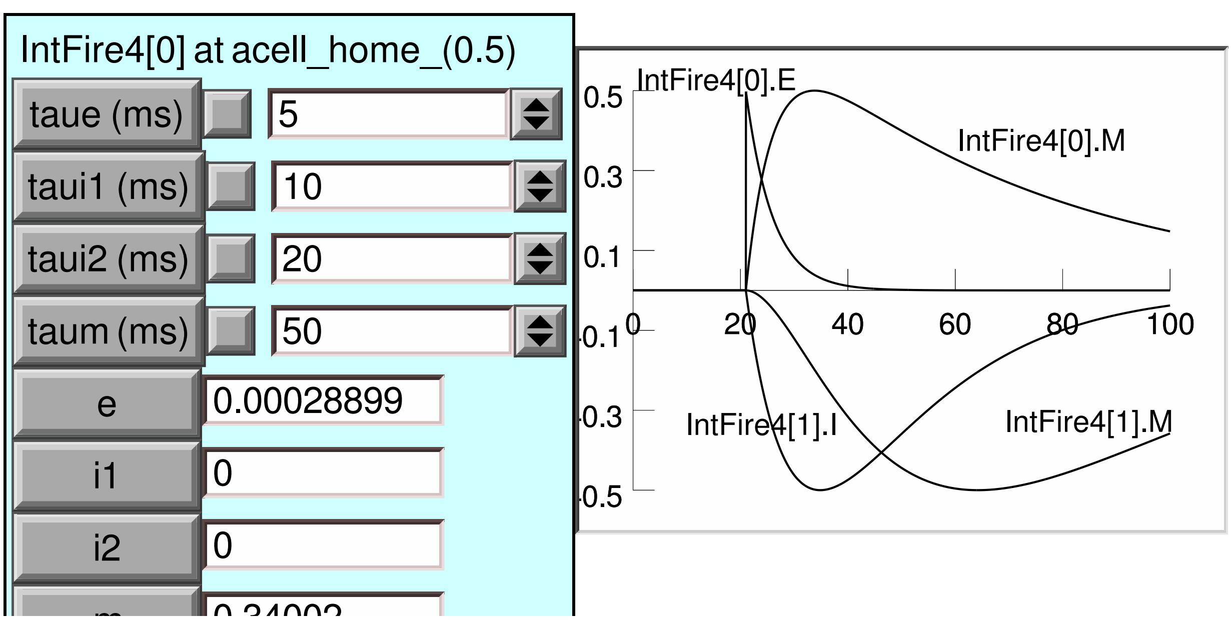


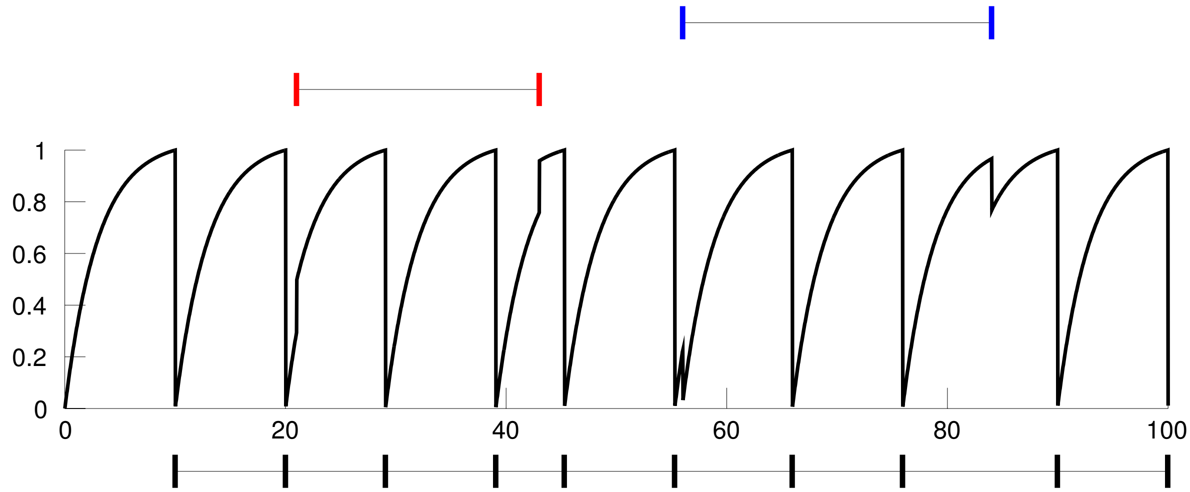



```
NEURON {
  ARTIFICIAL_CELL IntFire
  RANGE tau, m
}

...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[0] at acell_home_(0.5)
tau (ms)
10
refrac (ms)
5
m
0.5
IntFire1[0].M
1
0.8
0.6
0.4
0.2
0
0
20
40
60
80
100


IntFire2[0] at acell_home_(0.5)
taus (ms)
20
taum (ms)
10
ib
0.5
i
1
m
0.43877
IntFire2[0].I
IntFire2[0].M
1
0.8
0.6
0.4
0.2
0
0
20
40
60
80
100


IntFire4[0] at acell_home_(0.5)
taue (ms)
5
taui1 (ms)
10
taui2 (ms)
20
taum (ms)
50
e
0.00028899
i1
0
i2
0
IntFire4[0].E
IntFire4[0].M
IntFire4[1].I
IntFire4[1].M
0.5
0.3
0.1
-0.1
-0.3
-0.5
0
20
40
60
80
100




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

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

NET_RECEIVE (w) {
   m = minf + (m - minf)*exp(-(t - t0)/tau)
   t0 = t
   if (flag == 0) {
      m = m + w
      if (m > 1) {
         m = 0
         net_event(t)
      }
      net_move(t+firetime())
   }else{
      net_event(t)
      m = 0
      net_send(firetime(), 1)
   }
}

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

NetCon and NET_RECEIVE

```
NetCon(source, target, threshold, delay, weight)

   Event delivery with axonal delay

   Watch the source for threshold crossing in positive
   direction.
      &soma.v(.5)
      PresynapticObject.x
      Or PresynapticObject has a NET_RECEIVE block and calls
        net_event(t1) (discrete event simulation)

      All NetCon objects with same source use same threshold
      detector.

      If threshold occurs at time, t1, insert
      NetCon objects with that source into delivery queue
      for delivery at time t2 = t1 + NetCon.delay

   Balanced binary tree queue implementation.
   No loss of events. Works for delay=0.

   Event delivered to target NET_RECEIVE block at time t2.
      Same synaptic target equations used by many NetCon s.

      All declared NET_RECEIVE arguments are call by
      reference with separate storage in each NetCon.
         Calculations for different streams can be done once
         per event instead of once per dt.

      A target can send itself an event with
      net_send(delay, flag) and move it to a new time with
      net_move(t)

   With variable step methods, and 1 or more events at time
   t, fadvance() returns at time t before the events are
   delivered and at time t after the events are delivered.
```


Wiring networks in Neuron

Bill Lytton

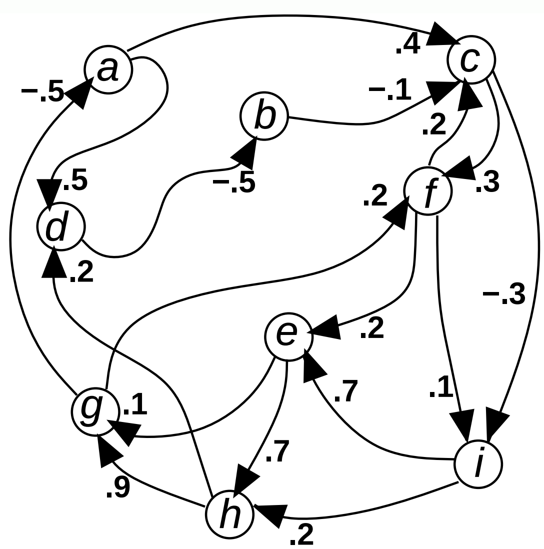
SUNY - Downstate

Brooklyn, NY

TOC

TOC2

Simple network



Connections table

FROM ⇒ / TO ⇓	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
a	■						-.5		
b		■		-.5					
c	.4	-.1	■			.2			
d	.5			■				.2	
e					■	.2			.7
f			.3			■	.2		
g					.1		■	.9	
h					.7			■	.2
i			-.3			.1			■

Connectivity matrix

$$\begin{pmatrix} \mathbf{0} & 0 & 0 & 0 & 0 & 0 & -.5 & 0 & 0 \\ 0 & \mathbf{0} & 0 & -.5 & 0 & 0 & 0 & 0 & 0 \\ .4 & -.1 & \mathbf{0} & 0 & 0 & .2 & 0 & 0 & 0 \\ .5 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & .2 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{0} & .2 & 0 & 0 & .7 \\ 0 & 0 & .3 & 0 & 0 & \mathbf{0} & .2 & 0 & 0 \\ 0 & 0 & 0 & 0 & .1 & 0 & \mathbf{0} & .9 & 0 \\ 0 & 0 & 0 & 0 & .7 & 0 & 0 & \mathbf{0} & .2 \\ 0 & 0 & -.3 & 0 & 0 & .1 & 0 & 0 & \mathbf{0} \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \end{pmatrix}$$

Define connectivity

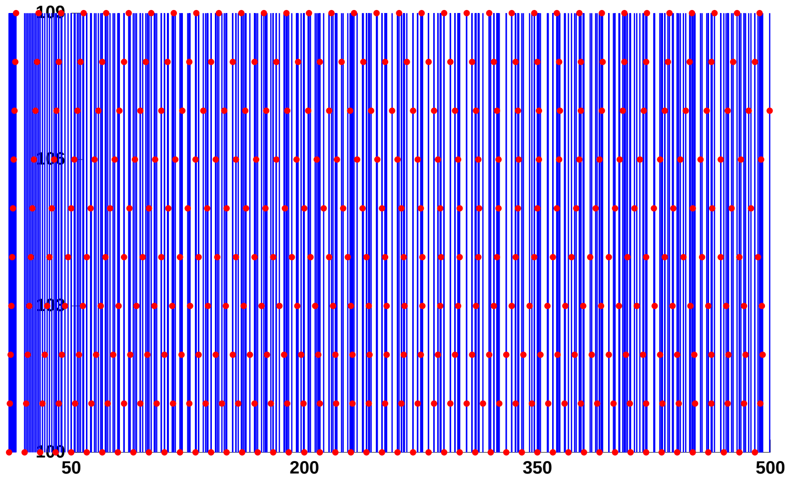
- S=number of syns; D=divergence; C=convergence
- $S = C \cdot Post$; $S = D \cdot Pre$
- connectivion density
  $p_{ij} = C/Pre = D/Post = S/(Pre \cdot Post)$
- Below: 1 kind of cell ⇒ Pre and Post are the same

Hopfield-Brody synchronization model

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

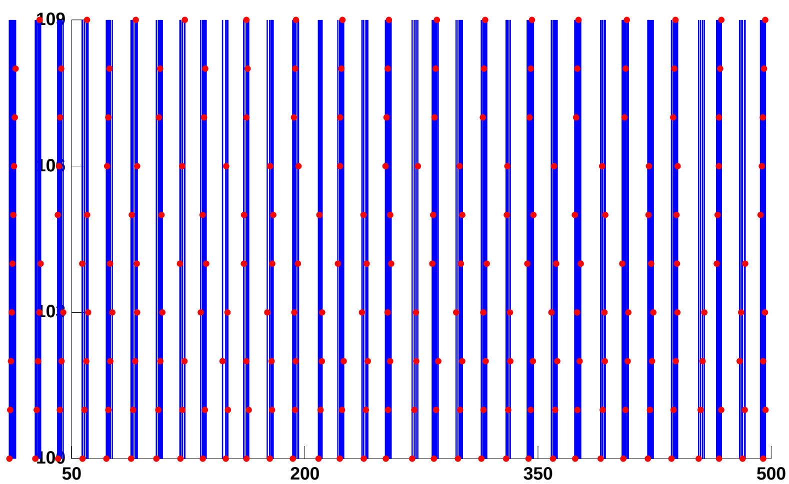
Unconnected cells

wt=-1e-5
cell fire at own rates and go out of phase



Negative (inhibitory) connectivity

wt=-3e-1



Q&D synchronization measure

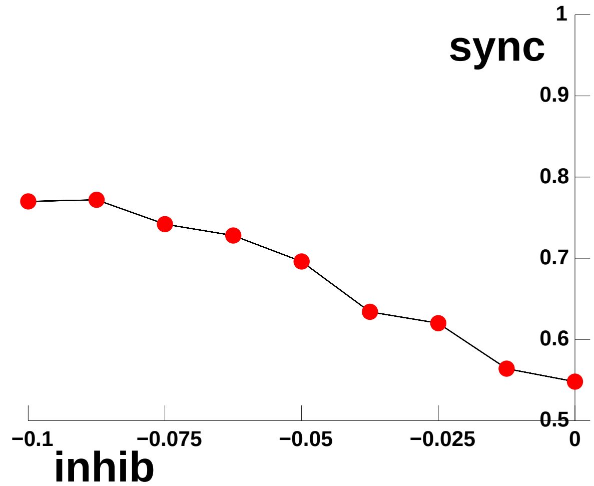
```
// syncer() :: returns sync measure 0 to <1
// measures how well spikes "fill up" the ti
// assumes spike times in tvec, tstop
// param: width
func syncer () { local t0,tt,cnt,width
  t0=-1 width=1 cnt=0
  for ii=0,tvec.size-1 {
    tt=tvec.x[ii]
    if (tt>=t0+width) {t0=tt cnt+=1}
  }
  return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


Check sync with increasing inhib

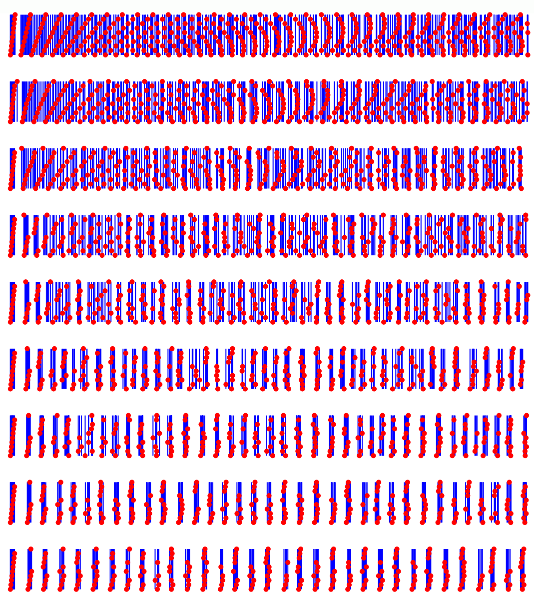
```
// loop increasing neg. synaptic weight
// save measures in vec[1],vec[0]
max= -0.1
for (w=0;w>=max;w+=(max/8)) {
  w+=1e-6 // avoid using zero weight
  setparams()  run()
  // g=new Graph() showspks()
  vec[1].append(w) vec.append(syncer())
}
```


Graph results

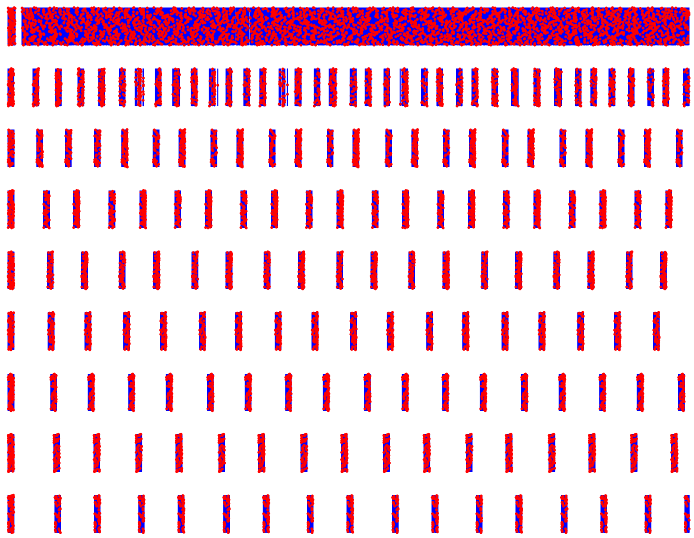
```
{vec.line(g,vec[1]) vec.mark(g,vec[1])}
```



Confirm with raw data

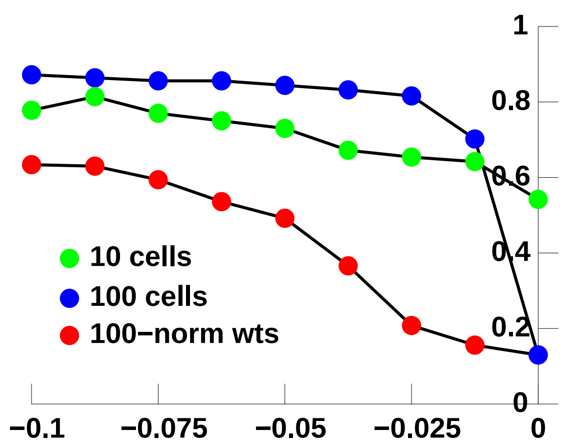


Scale up to 100 cells



Need to normalize weights

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



NEURON's list *object*

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

Wiring the network

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


100 x 100 matrix

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

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


Rewiring for different densities

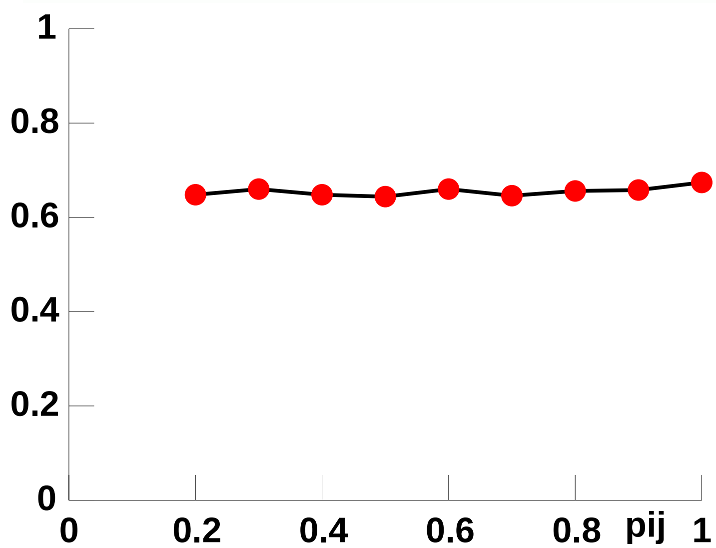
- Can either rewire (if sparse) or just set weights to 0 (if not)
- rewrite wire():

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


Or rewrite weight()

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


Density doesn't make much difference!



Checking connectivity

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

Wiring networks in Neuron – p.23/42

cvode.netconlist()

access NEURON's internal NetCon list

- Unwieldy, but important to make sure that WYWIWYG
- To check divergence:
```
for ii=0,ncell-1 print\
cvode.netconlist(cells.object(ii).pp,"","'
.count
```
- To check convergence:
```
for ii=0,ncell-1 print\
cvode.netconlist("","",cells.object(ii).pp
.count
```
- Could count non-zero synapses by iterating through `cvode.netconlist("","","")`

Wiring networks in Neuron – p.24/42

In addition to cvode.netconlist()

Develop a parallel database

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

fconn() – find connections

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

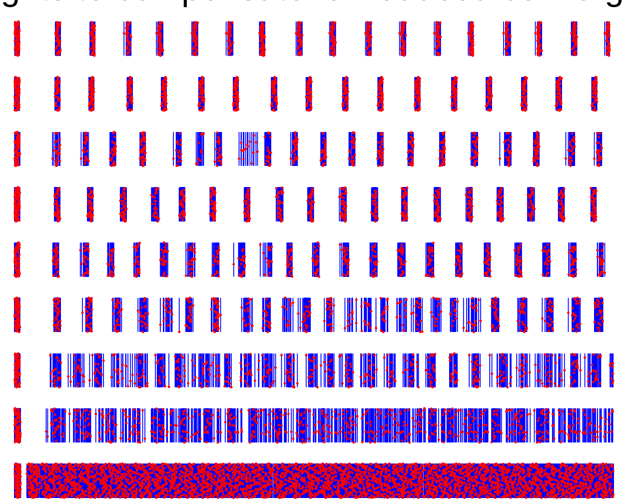

Now can check non-zero connectivity

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

Wiring networks in Neuron – p.27/4

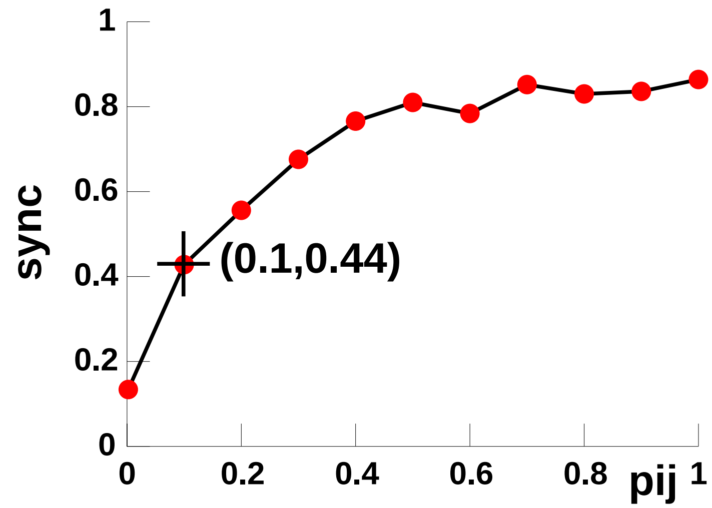
Rewrite randomizer

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



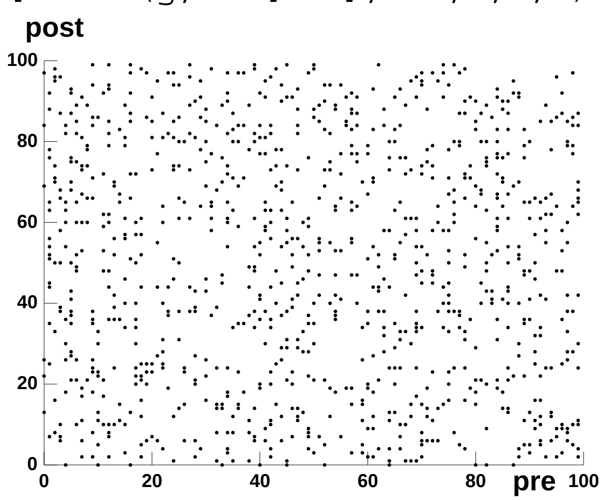
Wiring networks in Neuron – p.28/4

Synchronization measure

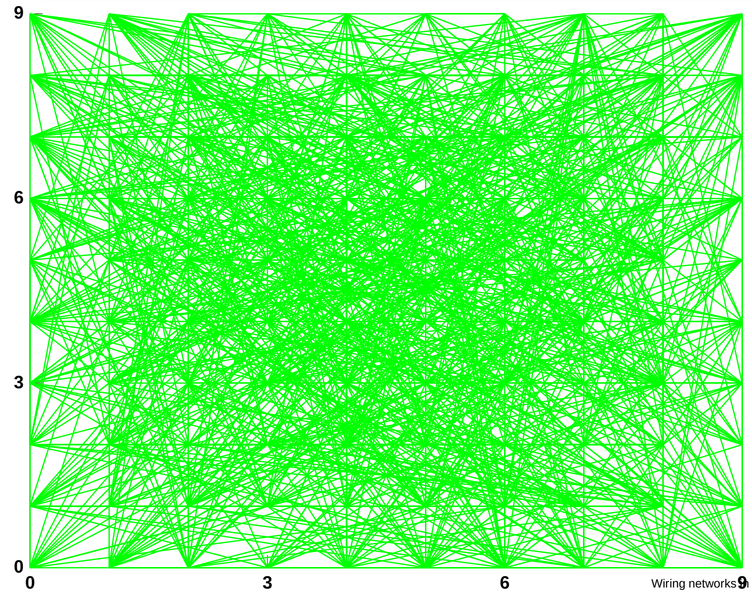


Look at 10% connectivity

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

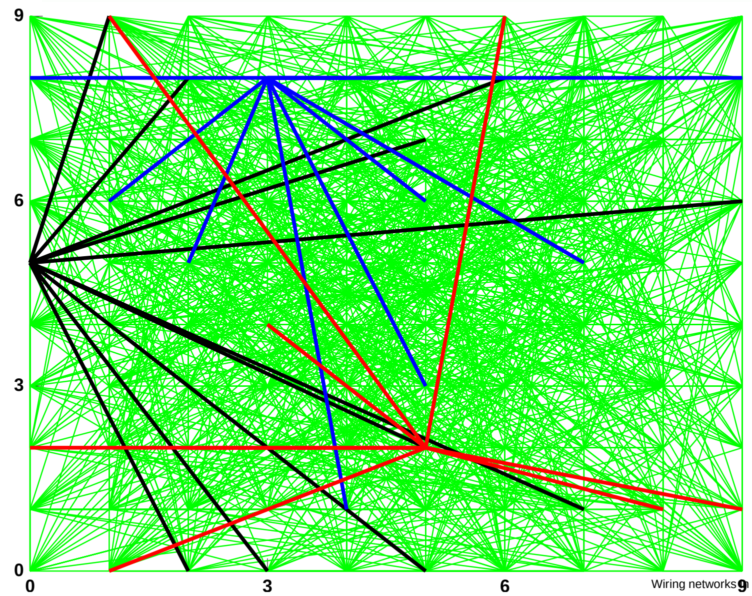


Show all connections



Wiring networks in Neuron – p.31/42

Show selected connections



Wiring networks in Neuron – p.32/42

Balancing convergence and divergence

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

Geographic connectivity

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

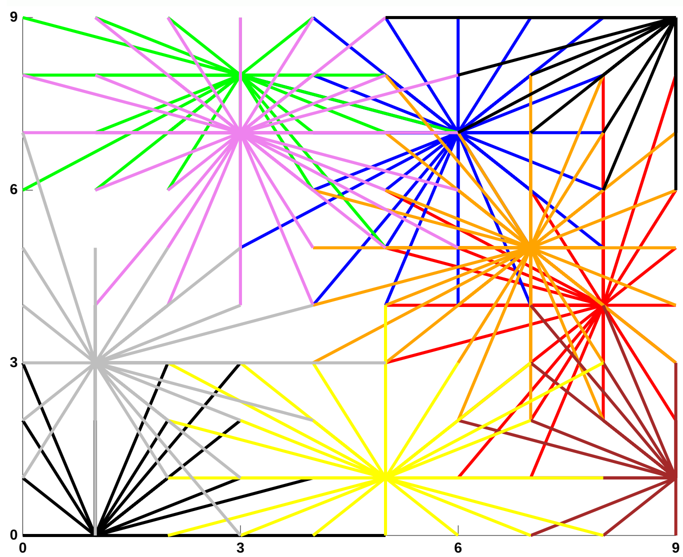
Random wiring with distance fall-off

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

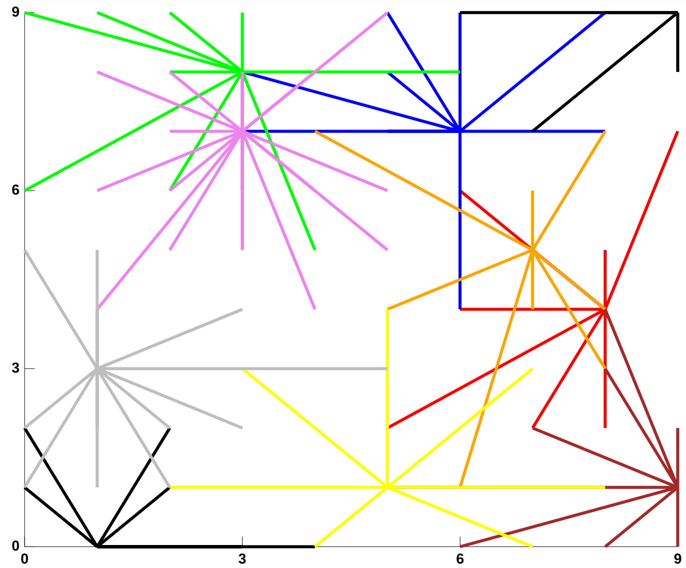
```
   rdm[1].uniform(0,1) // for flipping
coin
   prob = 1-distn()/maxdist
   if (rdm[1].repick<prob) { ...
```

- 3. Count syns till reach desired density
- Better if used a hexagonal array

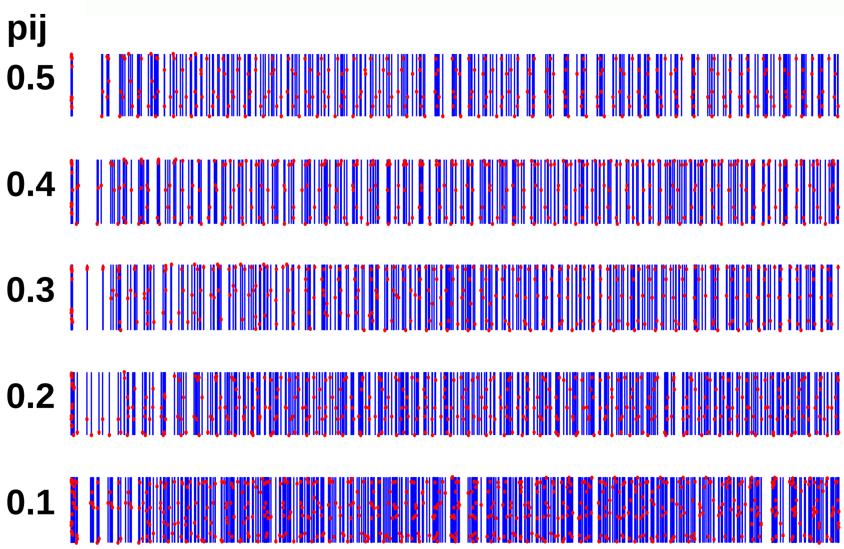
$p_{ij} = .5$: convergence for 10 cells



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

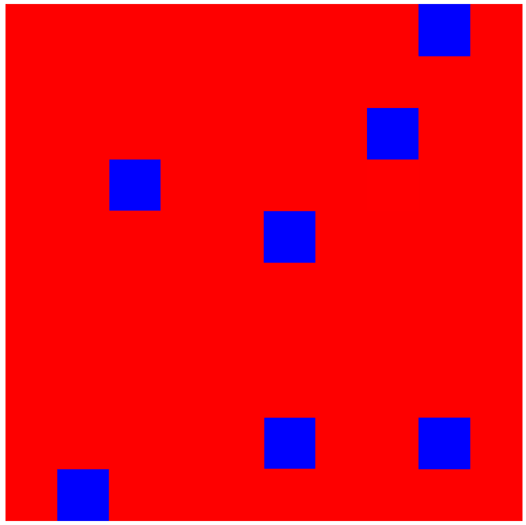


Doesn't sync very well



Is there localized sync'ing?

Look at animation.



How to animate

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


Other explorations

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

Advantages of NEURON *for networks*

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

NEURON's standard run system

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

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

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

Initialization, broadly speaking:

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

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

Initialization should assign values at t = 0 for

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

and properly configure

- event queues
- vector record and play
- counters

NEURON's `finitialize()`

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

Default initialization: the standard run library

`nrn/share/nrn/lib/hoc/stdrun.hoc`

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

`stdinit()`

Called when you

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

or

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

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


init()

Most customizations are made here

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


INITIAL blocks in NMODL

HH−like mechanisms

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

Kinetic schemes

```
INITIAL {
  SOLVE scheme METHOD steadystate
}
```

e.g.

```
NEURON {
  USEION k READ ek WRITE ik
}
STATE { c1 c2 o }
INITIAL {
  SOLVE scheme METHOD steadystate
}
BREAKPOINT {
  SOLVE scheme METHOD sparse
  ik = gbar*o*(v - ek)
}
KINETIC scheme {
  rates(v) : calculate the 4 k rates.
  ~ c1 <-> c2 (k12, k21)
  ~ c2 <-> o ( k2o, ko2)
}
```


Default initialization of `STATE`s

Use `state0`, e.g.

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

or alternative syntax

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

It's best to be explicit

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

To make them visible from `hoc`

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


Typical custom initializations

Steady state

unperturbed system

system under constant voltage or current clamp

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

Adjust parameters to meet some condition

How?

Use a custom `init()` procedure.

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

Initializing to steady state

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

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

Initializing to a desired state

Especially useful for oscillating or chaotic models.

Run a "warmup simulation," then save all states

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

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

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

To read from a file

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


Initializing to a desired state continued

A custom init() that restores saved states

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


Initializing to a particular resting potential

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

Example: for a single compartment model with hh,

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


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

Example:

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

UNITS {
  (mA) = (milliamp)
}

PARAMETER {
  ic = 0 (mA/cm2)
}

ASSIGNED {
  i (mA/cm2)
}

BREAKPOINT {
  i = ic
}
```

This needs a different custom `init()`

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

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


Anatomy and Empirically−based Models

Quality of data

- histology
  - staining, amputation, shrinkage
- human error
- diameter
- spines

Data formats

Tests for Quality of Data

Quick and dirty "litmus test"

- insert pas
- set Ra and g_pas low
- inject large depolarizing current at soma
- examine shape plot of v

Quantitative: look for pt3d data
with suspicious diameters,
e.g. too large or too small, = = 0, etc.

A "one liner"

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

More quantitative: look for systematic
errors, e.g. with a histogram of diameters

NEURON's tools for Electrotonic Analysis

Input and transfer impedances

Voltage transfer ratio

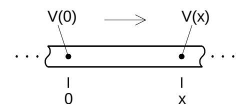
$V_{downstream} / V_{upstream}$

Electrotonic transformation

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

. . . all as functions of
frequency and space

Classical Cable Theory



Infinite cylinder in the steady state

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

$x \equiv$ physical distance

$\lambda \equiv$ length constant

Classical Definition of Electrotonic Distance:

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

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

Intuitively simple

BUT neurons ≠ infinite cylinders

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

Finite cylinder in the steady state

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

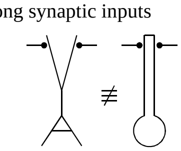
$L_{classical} \equiv$ physical length of cylinder / λ

$X \equiv x / \lambda$

The good and the bad news about the equivalent cylinder approximation

The bad news:

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



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

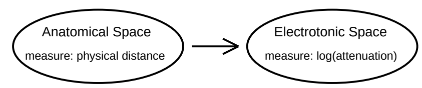
The good news: it's not valid either

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

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

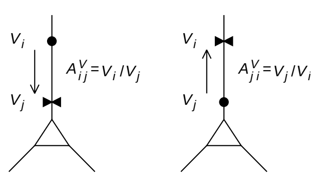
Alternative: a transformation from anatomical to electrotonic space that

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



Foundation of this approach:
two−port analysis of electrotonus

How well do signals propagate?



$$A_{ij}^{V} \neq A_{ji}^{V}$$

Signal transfer is direction−dependent

Attenuation identities

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

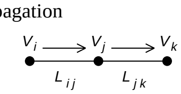
The Electrotonic Transformation

Functional definition of electrotonic distance

***L* = log(attenuation)**

- ✔ simple, direct relationship to attenuation
- ✔ direction−dependent: $L_{ij}^{V} = \ln(A_{ij}^{V})$,
  $L_{ji}^{V} = \ln(A_{ji}^{V})$, and in general $L_{ij}^{V} \neq L_{ji}^{V}$

  Each physical segment *ij* of a cell has **two** representations in electrotonic space−−one for each direction of propagation!
- ✔ identical to classical electrotonic distance for an infinite cylinder
- ✔ additive over a path with a consistent direction of propagation



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

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

Using the Electrotonic Transformation

At each frequency of interest:

Step 1: Transform from anatomical to electrotonic space

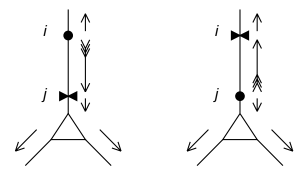
a. Compute attenuations between points of interest

b. Map into electrotonic space (log)

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

a. A convenient reference: the soma

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



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

c. The attenuation identities give us the transform identities

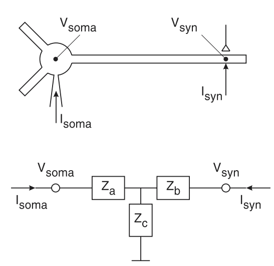
$V_{in} = I_{out} = Q_{out}$ and $V_{out} = I_{in} = Q_{in}$

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

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

Time−honored and ***wrong!***

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



Modified from Fig. 1 in Jaffe & Carnevale 1999.

If synapse ≈ voltage source, then

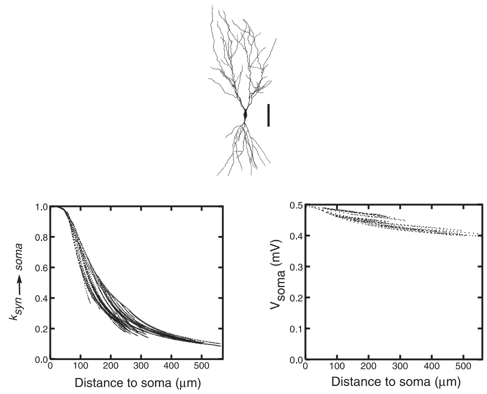
1. V_{syn}(t) ≈ independent of synaptic location

and

2. Synapse to soma voltage transfer ratio

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

predicts the variation of somatic PSP amplitude with synaptic location



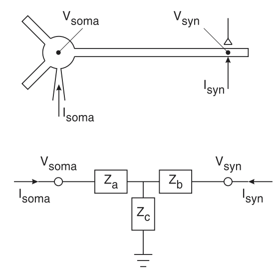
Somatic PSP predicted by $k_{syn \to soma}$

Somatic PSP produced by conductance−change synapse

Modified from Fig. 5 in Jaffe & Carnevale 1999.

Q: Why does $k_{syn \to soma}$ fail to predict the relationship between somatic peak PSP amplitude and synaptic location?

A: Because synapses act more like current sources than voltage sources.



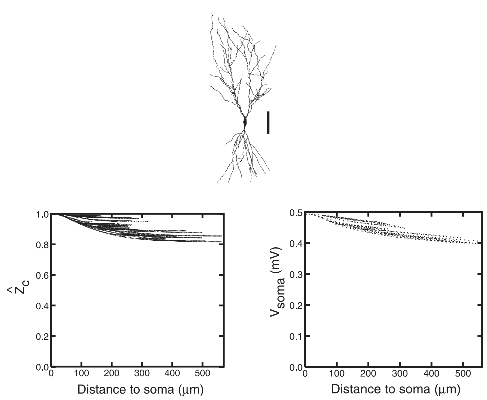
Modified from Fig. 1 in Jaffe & Carnevale 1999.

If synapse ≈ current source, then

1. $I_{syn}(t)$ ≈ independent of synaptic location

and

2. Transfer impedance Z_c predicts the variation of somatic PSP amplitude with synaptic location

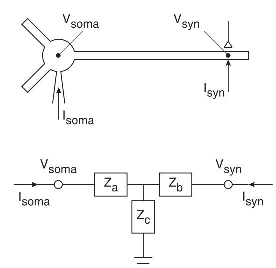


Somatic PSP predicted by transfer impedance

Somatic PSP produced by conductance−change synapse

Modified from Fig. 5 in Jaffe & Carnevale 1999.

Normalized transfer impedance $\hat{Z}_c$ is a good predictor of how somatic PSP amplitude varies with synaptic location.



Modified from Fig. 1 in Jaffe & Carnevale 1999.

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

Proof:

$k_{soma \to syn} = Z_c \, / \, (Z_a + Z_c)$ but $Z_a + Z_c$ is Z_N^{soma}, i.e. the input impedance of the cell at the soma.

Since Z_N^{soma} is the maximum transfer impedance between any location and the soma, $k_{soma \to syn} \equiv \hat{Z}_c$.

Survey

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

Please score these items	. . . . .	**according to this scale**	
Overall impression	______	no opinion	0
Relevance to my research	______	poor, not helpful	1
Didactic presentations	______	fair	2
Hands−on exercises	______	good	3
Written handouts	______	excellent, very helpful	4
Overhead transparencies	______		
Computer projection	______		
Computer classroom	______		
Food	______		
Housing	______		

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.

My area of primary research interest is ____

circle one

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

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

Which software? ____