

Table of Contents and Schedule of Presentations

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

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

Saturday, 6/23 Morning session

Time	Speaker	Title	Page
9:00 AM	MLH	Welcome to the NEURON summer course	
9:15	NTC	Introduction to modeling with NEURON	7
9:30	NTC	Example: building and using a simple model with the GUI	11 *
10:45	Coffee Break		
11:00	NTC	Fundamental concepts: neurites, cables, and sections	13
11:15	MLH	Interactive modeling: Hodgkin-Huxley axon	15 *
12:15	End of morning session		
12:30	Lunch		

Afternoon session

1:30	NTC	Fundamental concepts: range, range variables, nodes, and nseg	17
1:45	NTC	Example: constructing branched model cells with the CellBuilder	21 *

3:15	Coffee Break		
3:30	NTC	Working with morphometric data	23 *
5:00	End of afternoon session		

Sunday, 6/24 Morning session

Time	Speaker	Title	Page
9:00 AM	Q & A		
9:15	MLH	NMODL: the NEURON Model Description Language	27 *
10:30	Coffee Break		
10:45	NTC	Overview of creating and using NEURON models	35
11:00	WWL	The hoc programming language	41 *
12:15	End of morning session		
12:30	Lunch		

Afternoon session

1:30	MLH	Numerical methods: accuracy, stability, speed	57
2:30	NTC	ModelDB and Model View	65 *
3:30	Coffee Break		
3:45	NTC	Inhomogeneous channel distributions	71 *
5:00	End of afternoon session		

Monday, 6/25 Morning session

Time	Speaker	Title	Page
9:00 AM	Q & A		
9:15	MLH	Families of simulations in parallel	73 *
10:45	Coffee Break		
11:00	MLH	Using a supercomputer	79 *
12:15	End of morning session		
12:30	Lunch		

Afternoon session

1:30	MLH	NEURON + threads	85 *
3:15	Coffee Break		
3:30	NTC	Initialization	97 *
5:00	End of afternoon session		

Tuesday, 6/26 Morning session

Time	Speaker	Title	Page
9:00 AM	Q & A		
9:15	WWL	Python + NEURON	105
10:30	Coffee Break		
10:45	NTC	Networks: synapses, events, and artificial spiking cells	119 *
12:15	End of morning session		
12:30	Lunch		

Afternoon session

1:30	WWL	Networks: inhibitory synchronizing network	131 *
3:00	Coffee Break		
3:15	MLH	Variable time steps and parameter discontinuities	153 *
5:00	End of afternoon session		

Wednesday, 6/27 Morning session

Time	Speaker	Title	Page
9:00 AM	Q & A		
9:15	TJS	<i>TBA (special topic in computational neuroscience)</i>	
10:30	Coffee Break		
10:45	MLH	Parallel computation: distributed network models	177
12:15	End of morning session		
12:30	Lunch		

Afternoon session

1:30	NTC	NEURON's tools for analyzing electrotonus	195 *
3:00	Coffee Break		
3:15	NTC	The Linear Circuit Builder	203 *
4:15	Review discussion		
4:45	Evaluation form (see last page in this booklet)		
5:00	End of afternoon session		

Optional	MLH	The Channel Builder	211
Appendix	Translating network models to parallel hardware in NEURON (Hines & Carnevale 2008). Parallel tutorial from 2010 NEURON Users' Meeting. NEURON and Python (Hines et al. 2009).		before survey
Receipt			penultimate page
Survey			last page

The What and the Why of Neural Modeling

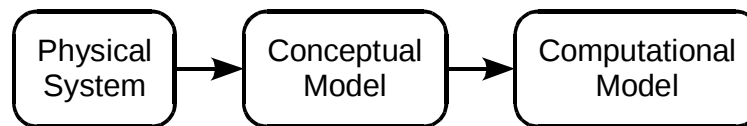
The moment-to-moment processing of information in the nervous system involves the propagation and interaction of electrical and chemical signals that are distributed in space and time.

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

Topics

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

From Physical System to Computational Model



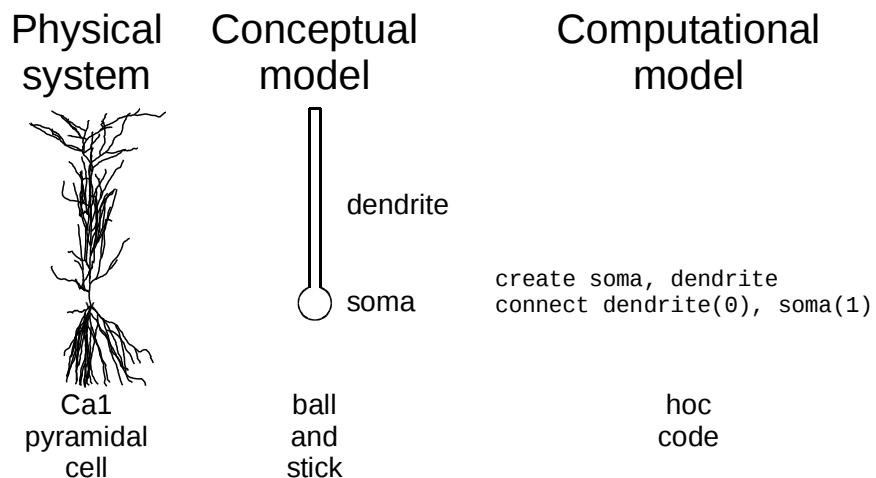
Conceptual model

a simplified representation of the physical system

Computational model

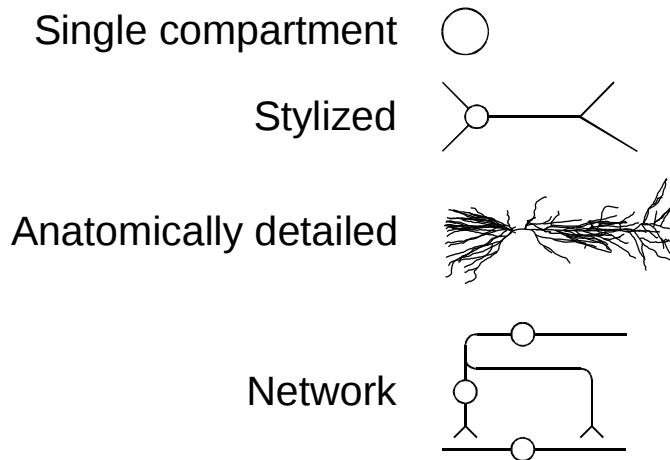
an accurate representation of the conceptual model

From Physical System to Computational Model



Hierarchies of Complexity

Structure



Hierarchies of Complexity

Mechanism

Passive and Active currents

HH-style
kinetic scheme

Synaptic transmission

continuous
spike-triggered

Gap junctions

Extracellular fields, Linear circuits

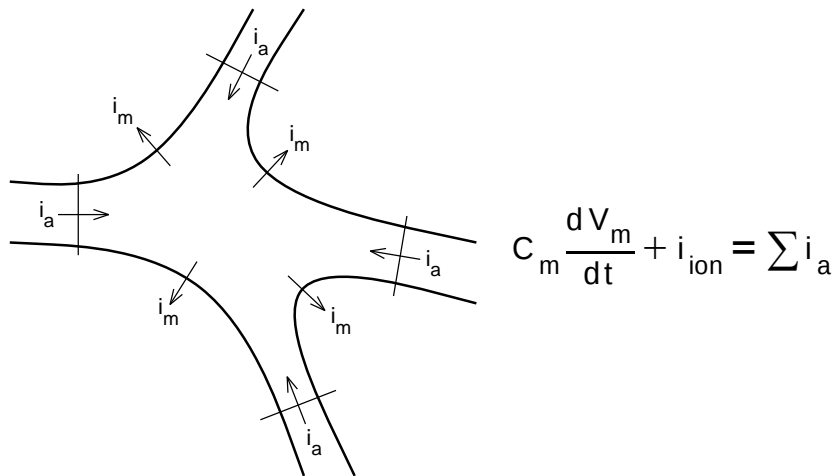
Diffusion, buffers, transport & exchange

Artificial spiking cells ("integrate & fire")

Fundamental Concepts in NEURON

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

Conservation of Charge



Example: Single Compartment

Lipid bilayer (no channels)

Membrane with linear ion channels (passive leak)

Project goals:

- Use the GUI to build the model and custom interface for using it
- Run simulations and analyze results
- Change stimulus intensity and duration
- Adjust graphical displays of simulation results
- Adjust dt and Points Plotted / ms

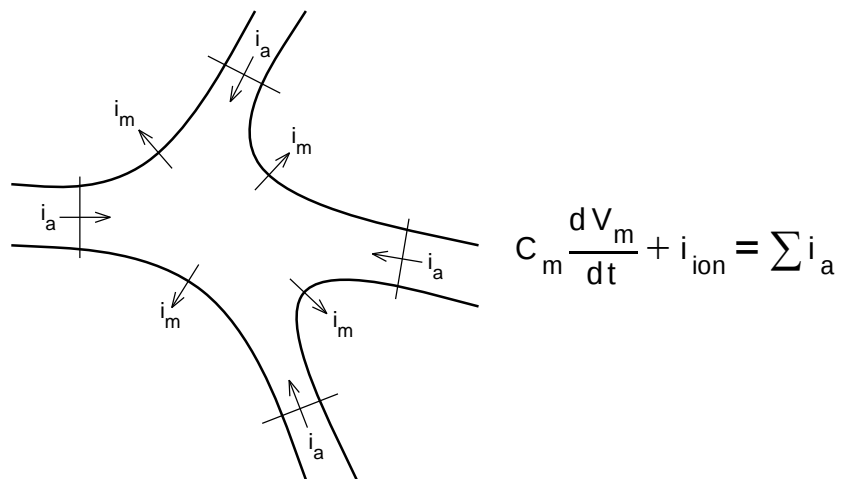
Fundamental Concepts in NEURON

What equations are being solved?

How to separate the biology
from computational details?

It's all about conceptual control . . .

Conservation of Charge



The Model Equations

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

v_j membrane potential in compartment j

i_{ion_j} net transmembrane ionic current in compartment j

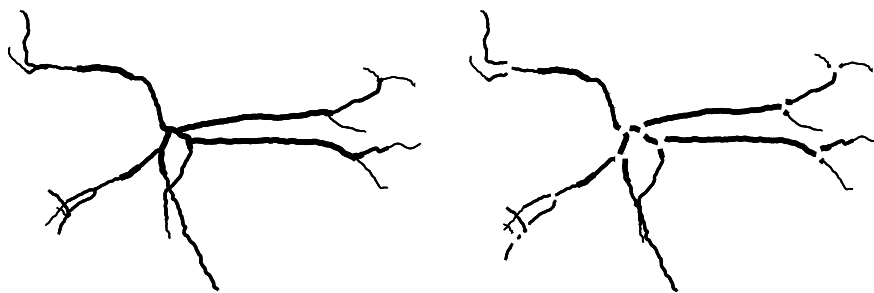
c_j membrane capacitance of compartment j

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

Separating Anatomy and Biophysics from Purely Numerical Issues

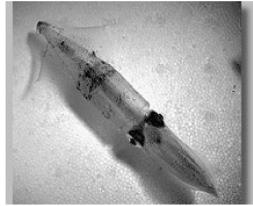
section

a continuous length of unbranched cable



Anatomical data from A.I. Gulyás

Physical System



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

Model

Hodgkin-Huxley cable equations

$$\frac{1}{4D} \frac{\partial}{\partial x} \left(\frac{D^2}{R_a} \frac{\partial V}{\partial x} \right) = C_m \frac{\partial V}{\partial t} + \bar{g} m^3 h \cdot (V - E_{na}) + \bar{g}_k n^4 \cdot (V - E_k) + g_l \cdot (V - E_l)$$

$$\begin{aligned} \frac{dm}{dt} &= -\alpha_m m + \beta_m (1-m) & \alpha_m &= \frac{0.1(V+40)}{1-e^{-0.1(V+40)}} & \beta_m &= 4e^{-(V+65)/18} \\ \frac{dh}{dt} &= -\alpha_h h + \beta_h (1-h) & \alpha_h &= 0.07e^{-0.05(V+65)} & \beta_h &= \frac{1}{1+e^{-0.1(V+35)}} \\ \frac{dn}{dt} &= -\alpha_n n + \beta_n (1-n) & \alpha_n &= \frac{0.01(V+55)}{1-e^{-0.1(V+55)}} & \beta_n &= 0.125e^{-(V+65)/80} \end{aligned}$$

Model

Hodgkin-Huxley cable equations

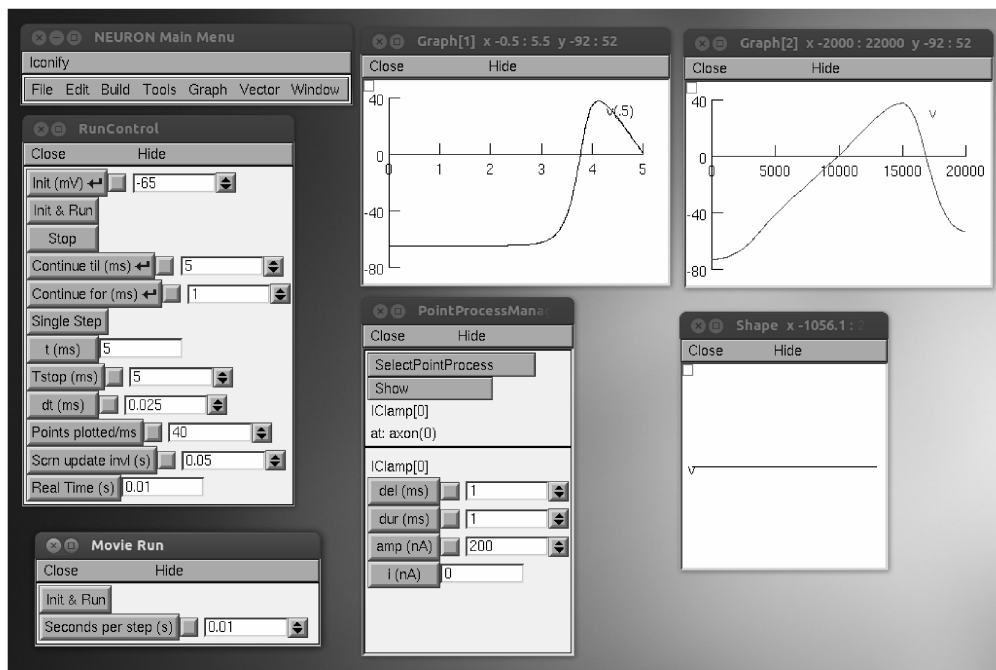
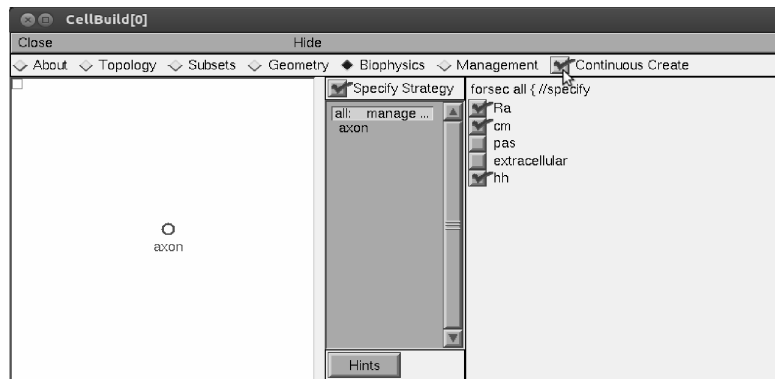
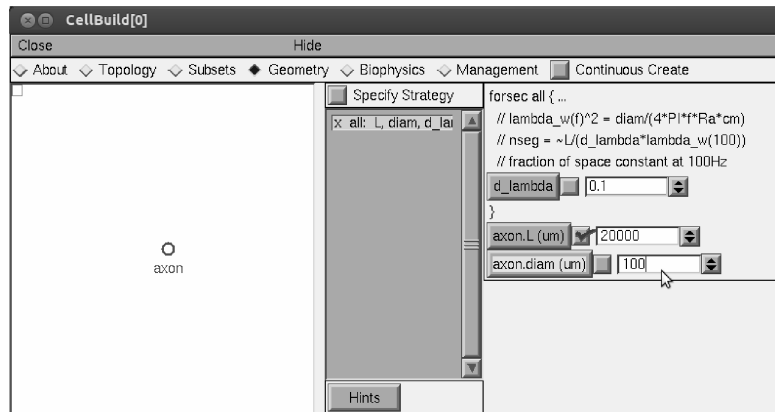
$$\frac{1}{4D} \frac{\partial}{\partial x} \left(\frac{D^2}{R_a} \frac{\partial V}{\partial x} \right) = C_m \frac{\partial V}{\partial t} + \bar{g} m^3 h \cdot (V - E_{na}) + \bar{g}_k n^4 \cdot (V - E_k) + g_l \cdot (V - E_l)$$

$$\begin{aligned} \frac{dm}{dt} &= -\alpha_m m + \beta_m (1-m) & \alpha_m &= \frac{0.1(V+40)}{1-e^{-0.1(V+40)}} & \beta_m &= 4e^{-(V+65)/18} \\ \frac{dh}{dt} &= -\alpha_h h + \beta_h (1-h) & \alpha_h &= 0.07e^{-0.05(V+65)} & \beta_h &= \frac{1}{1+e^{-0.1(V+35)}} \\ \frac{dn}{dt} &= -\alpha_n n + \beta_n (1-n) & \alpha_n &= \frac{0.01(V+55)}{1-e^{-0.1(V+55)}} & \beta_n &= 0.125e^{-(V+65)/80} \end{aligned}$$

Simulation

Representation

```
create axon
axon {
    nseg = 43
    diam = 100
    L = 20000
    insert hh
}
```



Syntax: create sectionname

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

Assigning anatomical and biophysical attributes:

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

Range Variables

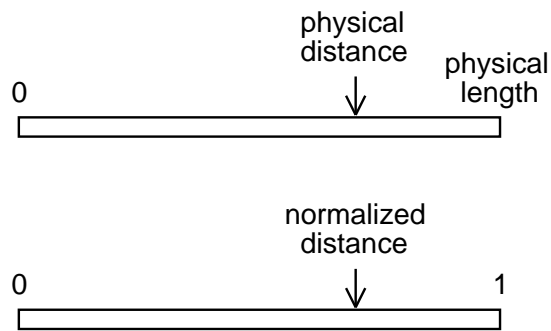
Name	Meaning	Units
diam	diameter	[μm]
cm	specific membrane capacitance	[$\mu\text{f}/\text{cm}^2$]
g_pas	specific conductance of the pas mechanism	[siemens/ cm^2]
v	membrane potential	[mV]

range

normalized position along the length of a section

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

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



Syntax:

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

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

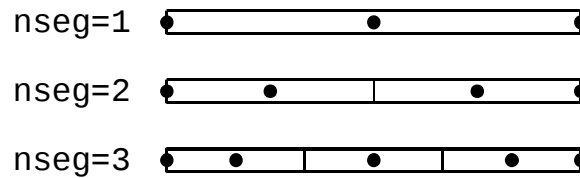
Examples:

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

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

nseg

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



Example: axon nseg = 3

To test spatial resolution
forall nseg = nseg*3
and repeat the simulation

Category	Variable	Units
Time	t	[ms]
Voltage	v	[mV]
Current		
specific	i	[mA/cm ²] (distributed)
absolute		[nA] (point process)
Capacitance		
specific	cm	[μf/cm ²]
absolute		[nf] (point process)
Length	diam, L	[μm]
Conductance		
specific	g	[S/cm ²] (distributed)
absolute		[μS] (point process)
Cytoplasmic resistivity	Ra	[Ω cm]
Resistance	ri()	[10 ⁶ Ω]
Concentration	nai etc.	[mM]

Example: Branched Model Cells

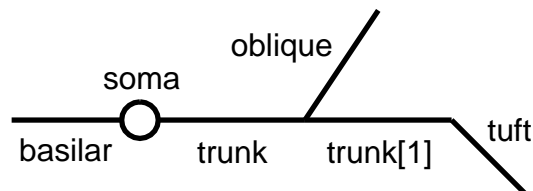
Physical system: anatomically complex cell

Conceptual model: "stick figure"

Computational model: soma + dendritic cylinder(s)
(and maybe an axon . . .)

Project goals:

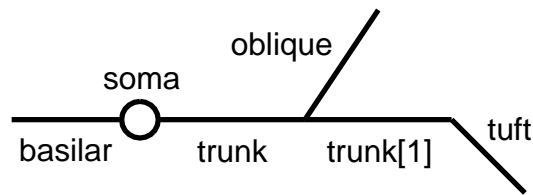
- Learn how to use CellBuilder
- Use session files to save and retrieve user interface (elementary project management)
- Test model and simulation:
structural integrity
discretization of space and time



From hoc file generated by CellBuilder:

```
create soma, trunk[2], oblique, tuft, basilar
```

```
proc topol() { local i
  connect trunk(0), soma(1)
  connect trunk[1](0), trunk(1)
  connect oblique(0), trunk(1)
  connect tuft(0), trunk[1](1)
  connect basilar(0), soma(0)
  . . .
```



From hoc file generated by CellBuilder:

```
proc geom() {
  forsec all { }
  soma { L = 30  diam = 30 }
  trunk { L = 400  diam = 3 }
  trunk[1] { L = 400  diam = 2 }
  oblique { L = 300  diam = 1.5 }
  tuft { L = 300  diam = 1 }
  basilar { L = 300  diam = 3 }
}
```

```
proc biophys() {
  forsec all {
    Ra = 160
    cm = 1
  }
  forsec dendrites {
    insert pas
    g_pas = 3e-05
    e_pas = -70
  }
  forsec apicals {
    insert hh
    gnabar_hh = 0.012
    gkbar_hh = 0.0036
    gl_hh = 0
    el_hh = -54.3
  }
  soma {
    insert hh
    gnabar_hh = 0.12
    gkbar_hh = 0.036
    gl_hh = 0.0003
    el_hh = -54.3
  }
}
```

Anatomy and Empirically-based Models

Quality of data

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

Data formats

Detailed Morphometric Data

Where to get it?

- DIY
- the kindness of strangers
- ModelDB
- NeuroMorpho.org

How to get it into NEURON?

- standalone conversion programs
- import into CellBuilder
- Import3D tool
- "already in hoc"

Trust but verify . . .

Qualitative tests

Orphan sections and bottlenecks?

insert pas, set Ra and g_pas low

inject large depol current at soma

examine shape plot of v

Z-axis drift and backlash?

examine side view of shape plot

for abrupt jumps

. . . and verify some more

Quantitative tests

Diameter

Too large?

```
_dmin=10
```

```
forall for i=0, n3d()-1 \
```

```
  if (diam3d(i)<_dmin) _dmin=diam3d(i)
```

```
print _dmin
```

Too small?

```
forall for i=0, n3d()-1 \
```

```
  if (diam3d(i)<0.1) \
```

```
    print secname(), " ", i, diam3d(i)
```


When it really matters . . .

Test for systematic errors

Favorite numbers?

 histogram of diameter measurements

Other tests

NMODL

NEURON Model Description Language

Add new membrane mechanisms to NEURON

Density mechanisms

- Distributed Channels
- Ion accumulation

Point Processes

- Electrodes
- Synapses

Described by

- Differential equations
- Kinetic schemes
- Algebraic equations

Benefits

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

NMODL general block structure

What the model looks like from outside

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

What names are manipulated by this model

```
UNITS { (mV) = (millivolt) ... }
PARAMETER { gbar = .036 (mho/cm2) <0, 1e9>... }
STATE { n ... }
ASSIGNED { ik (mA/cm2) ... }
```

Initial default values for states

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

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

(and specify how states are to be integrated)

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

State equations

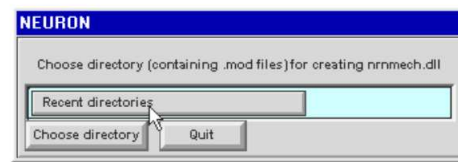
```
DERIVATIVE deriv {
    rates(v)
    n' = (ninf - n)/ntau
}
```

Functions and procedures

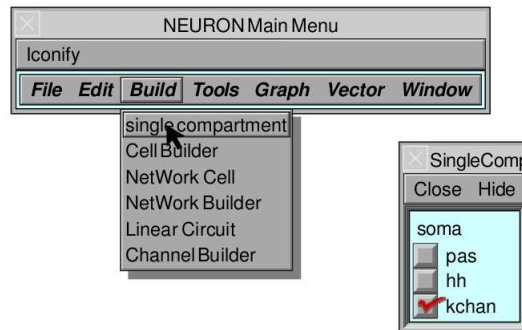
```
PROCEDURE rates(v(mV)) {
    ...
}
```

UNIX

nrnivmodl
nrngui

MSWIN

Select NEURON Main Menu / Build / single compartment



Density mechanism

```

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

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

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

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

```

Point Process

```

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

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

ASSIGNED {
    i (nanoamp)
    v (millivolt)
}

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

```

Density mechanism**Point Process****NMODL**

```

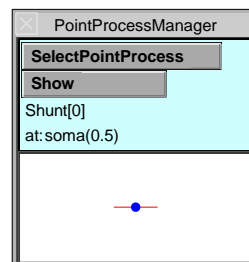
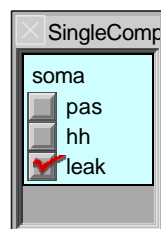
NEURON {
    SUFFIX leak
    NONSPECIFIC_CURRENT i
    RANGE i, e, g
}

```

```

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

```

GUI**Interpreter**

```

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

```

```

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

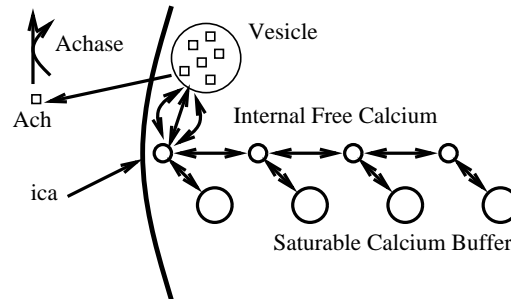
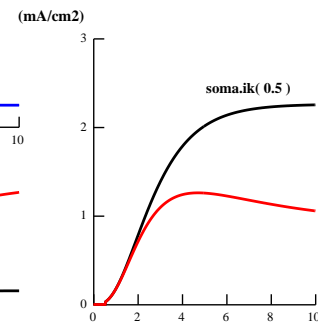
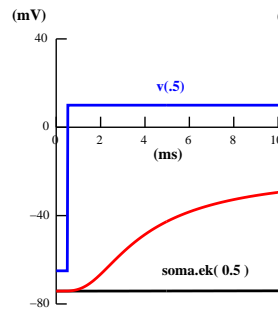
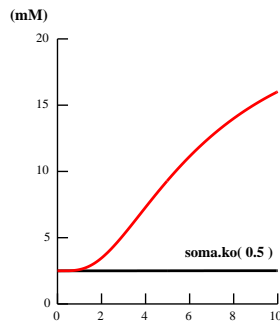
```

Ion Channel

```

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

```



```

STATE {
    Vesicle Ach Achase Ach2ase X Buffer[N] CaBuffer[N] Ca[N]
}
KINETIC calcium_evoked_release {
    : release
    ~ Vesicle + 3Ca[0] <-> Ach (Agen, Arev)
    ~ Ach + Achase <-> Ach2ase (Aase2, 0) : idiom for enzyme reaction
    ~ Ach2ase <-> X + Achase (Aase2, 0) : requires two reactions
    : Buffering
    FROM i = 0 TO N-1 {
        ~ Ca[i] + Buffer[i] <-> CaBuffer[i] (kCaBuffer, kmCaBuffer)
    }
    : Diffusion
    FROM i = 1 TO N-1 {
        ~ Ca[i-1] <-> Ca[i] (Dca*a[i-1], Dca*b[i])
    }
    : inward flux
    ~ Ca[0] << (ica)
}

```

UNITS Checking

```

NEURON { POINT_PROCESS Shunt ... }
PARAMETER {
    e = 0 (millivolt)
    r = 1 (gigaohm) <1e-9,1e9>
}
ASSIGNED {
    i (nanoamp)
    v (millivolt)
}
BREAKPOINT {
    i = (v - e)/r
}

```

Units are incorrect in the "i = ..." current assignment.

```

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

```

**The output from
modlunit shunt
is:**

```

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

```

To fix the problem replace the line with:

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

What conversion factor will make the following consistent?

$$\begin{array}{ccccccc}
 \text{na}^+ & = & i_{\text{na}} & / & \text{FARADAY} & * & (\text{c/radius}) \\
 (\mu\text{M/ms}) & & (\text{mA/cm}^2) & / & (\text{coulomb/mole}) & & / (\mu\text{m})
 \end{array}$$

Creating and using NEURON models

Use hoc, Python, and/or GUI to specify:

- Biological properties--anatomy, biophysics
- Instrumentation--signal sources and recording
- User interface--parameter panels, graphs
- Simulation control--dt, tstop, integration method

Hint: keep these separate from each other
for maximum clarity and to save effort

Verify:

- Close match to conceptual model?
- Numerical accuracy adequate?
(spatial grid, integration time step or error criterion)

Specifying biological properties

Topology (branching pattern)

Geometry (diameter, length)
and

Biophysics (membrane capacitance,
ion channels, pumps . . .)

Connections between cells
(synapses, gap junctions)

. . . *and anything else that makes sense* . . .

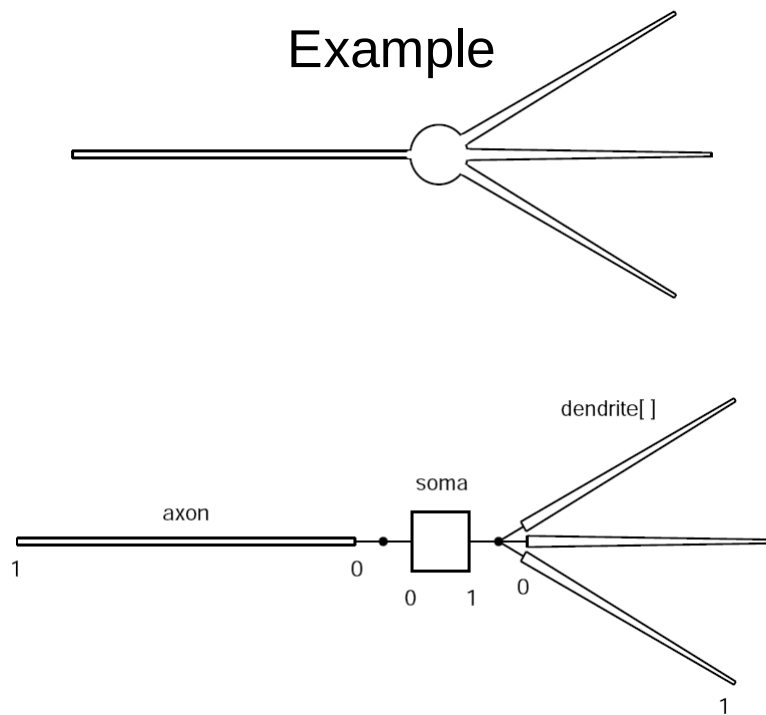
Biological properties: topology

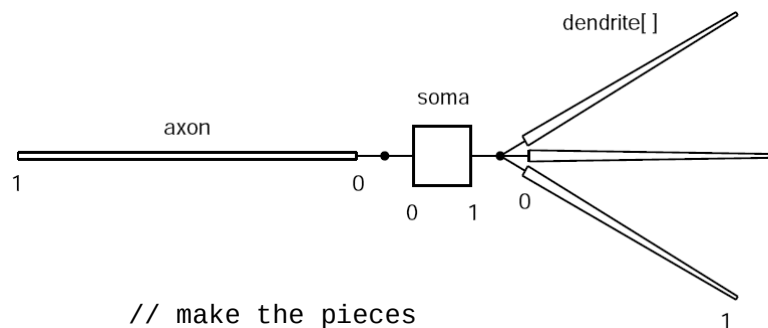
Make the pieces (sections)
create

Specify the default section
access

Assemble the pieces
connect

Example





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

// specify default section
access soma

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

Biological properties: geometry and biophysics

Compartmentalization

nseg

Geometry

L, diam

Biophysical properties

Density mechanisms: insert

Examples: ion channels distributed
over the cell surface, pumps,
ion accumulation, buffers

```

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

axon {
  nseg = 21   // odd so a node is at 0.5
  L = 1000
  diam = 1
  insert hh
}

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

forall Ra = 60 // [ohm cm]

```

Range variables

Vary continuously in space along the length of a section

Examples: `v`, `cm`, `diam`

Section variables

Pertain to an entire section

Examples: `Ra` (cytoplasmic resistivity), `L`, `nseg`

Global variables

Same across all sections

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

Instrumentation

This model needs an electrode at the soma to inject stimulating current.

Examples of "point processes":
current clamp, voltage clamp, synapse

Object syntax

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

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

Simulation control

Example 1: minimalist for fixed dt simulations

```
finitialize(-65) // initialize v, state variables, time
tstop = 5
dt = 0.025
proc simulate() {
  print t, v(0.5) // soma is default section
  while (t < tstop) {
    fadvance() // advance solution by dt
    // function calls to save or plot results, e.g.
    print t, v(0.5)
    // statements to change model parameters
  }
}
```

Example 2: using the standard run system

```
v_init = -65 // Vm at t==0
tstop = 5 // [ms]
steps_per_ms = 40 // points plotted/ms
dt = 0.025 // [ms] integration time step
setdt() // ensures that a whole number of dts
        // will fit into 1/steps_per_ms
run() // initialize, then run simulation
```

Program organization

```
modelspec.hoc
    "virtual organism"
    topology, geometry, biophysics
rig.ses
    "virtual experimental rig"
    clamps, graphs, run control
init.hoc
    "administrative wrapper"
    load_file("nrngui.hoc")
    load_file("modelspec.hoc")
    load_file("rig.ses")
```

Workflow

Develop/debug "virtual organism"
hoc, Python, NMODL, GUI (CellBuilder,
Channel Builder, Network Builder)
in whatever combination
Model View
Iterative cycle of incremental revision and testing

Use NEURONMainMenu to customize interface
attach synapses and electrodes
set up graphs and run control
specify integration method

NEURON: the HOC programming language

Bill Lytton

SUNY - Downstate
Brooklyn, NY

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

TOC

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

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

Talk to the simulator

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

NEURON: the HOC programming language – p.2/27

Numbers

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

NEURON: the HOC programming language – p.3/27

Functions & operators: pluses and minuse

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

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

NB: x=5 vs x==5

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

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

Assignments

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

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

Block of code

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

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

Conditionals and controls

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

NEURON: the HOC programming language – p.8/21

proc and func

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

NEURON: the HOC programming language – p.9/21

Number arguments to procedures:

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

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

Strings

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

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

Objects

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

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

Simulation commands

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

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

Sim - stim

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

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

Sim - running

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

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

Vectors

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

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

What have we recorded?

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

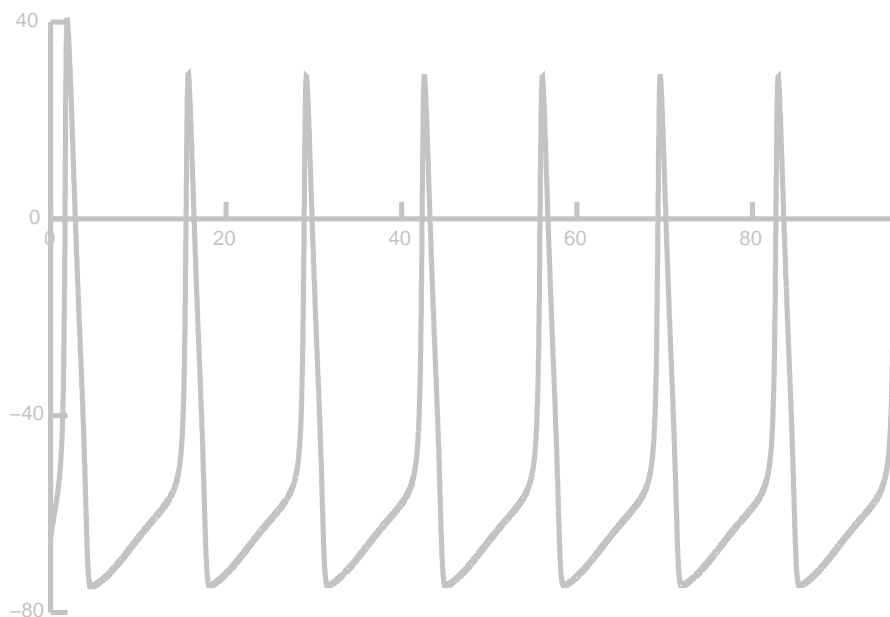
NEURON: the HOC programming language – p.17/21

Can analyze signals using vectors

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

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

Quick & dirty graphics



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

Graphing a vector

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

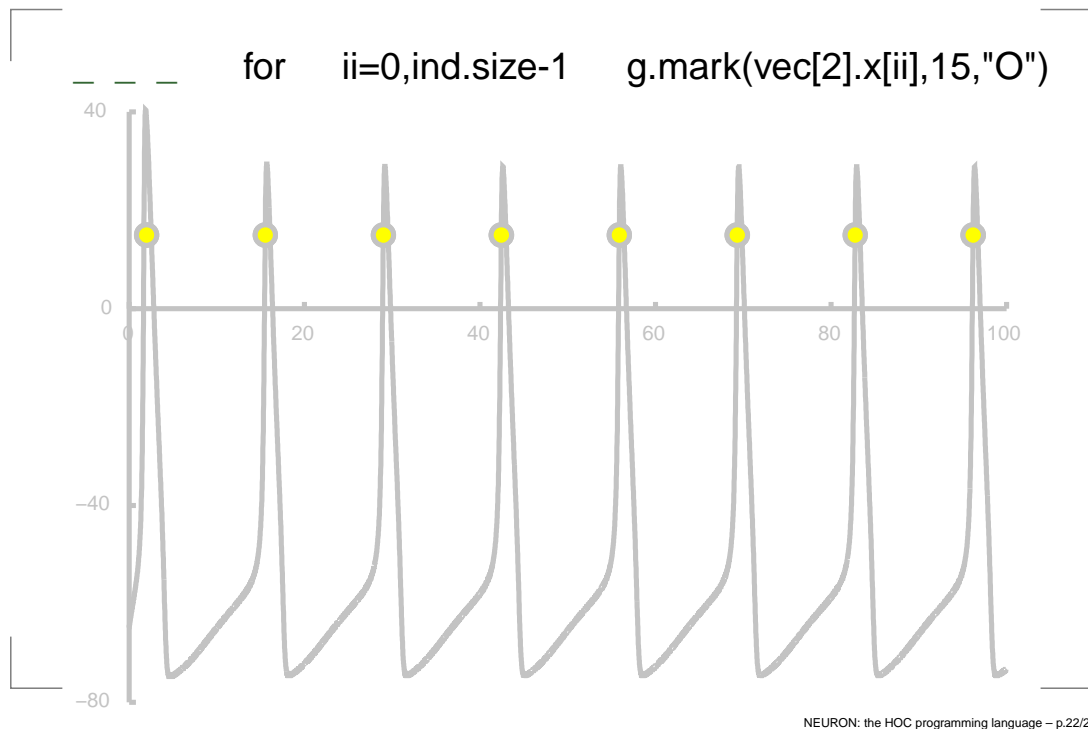
NEURON: the HOC programming language – p.20/21

Find spikes

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

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

Check results graphically



Now can calculate means etc.

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

Other useful vector functions

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

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

Putting up buttons



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

Reading and writing files

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

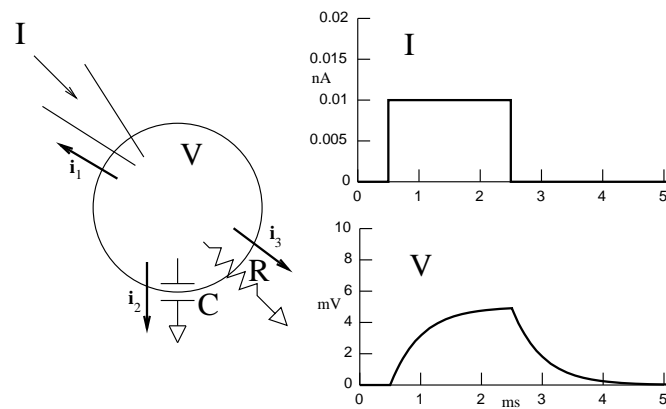
NEURON: the HOC programming language – p.26/27

Compartmental Modeling

Not much mathematics required.

Good judgment essential!

1



$$i_1 + i_2 + i_3 = 0$$

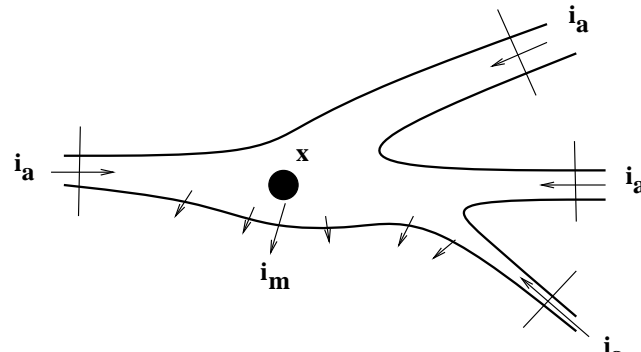
$$i_1 = -I$$

$$i_2 = C \, dV / dt$$

$$i_3 = V / R$$

$$C \, dV / dt + g \, V = I$$

2

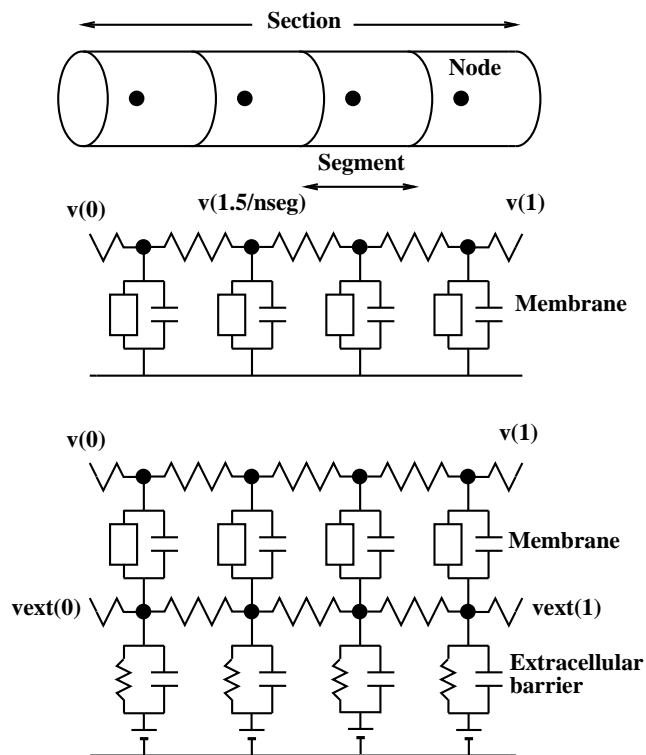


The diagram shows a neuron segment with a central node 'x'. Axial currents i_a flow into and out of the segment at its boundaries. Membrane currents i_m flow out of the segment across its surface. Below the diagram are two equations:

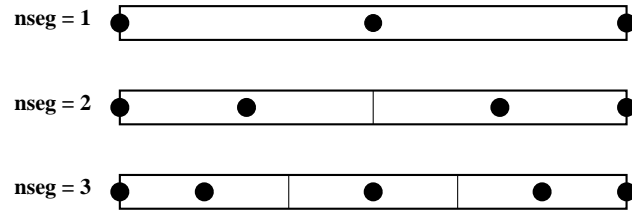
$$\int i_m = \sum i_a$$

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

3



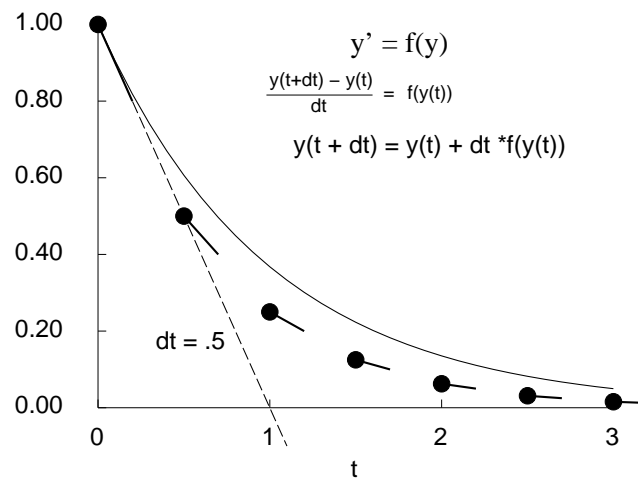
4



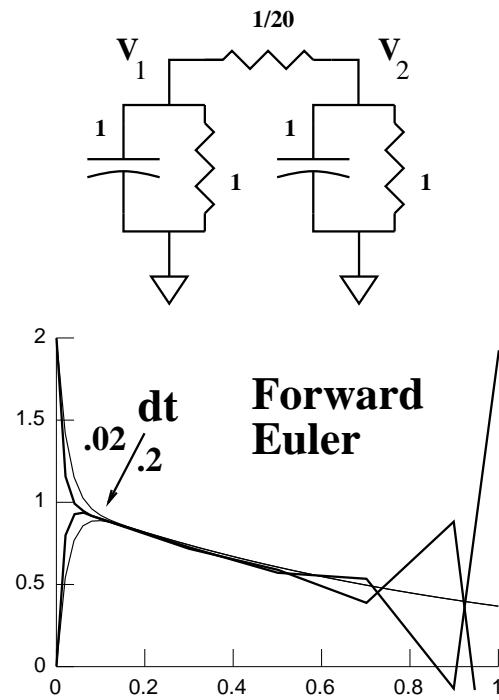
forall nseg *= 3

5

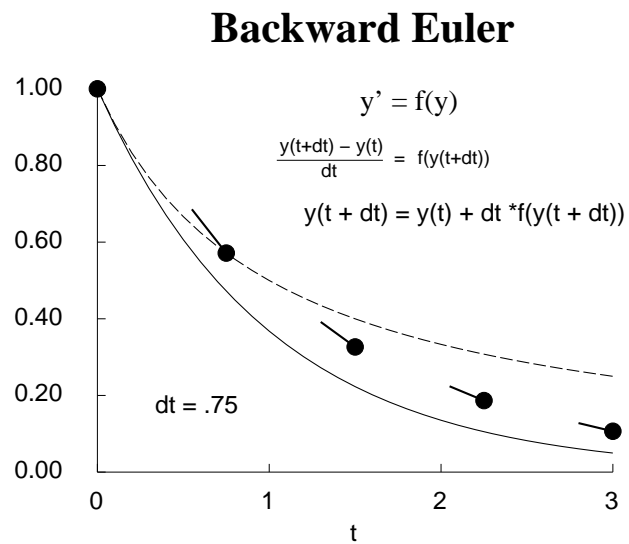
Forward Euler



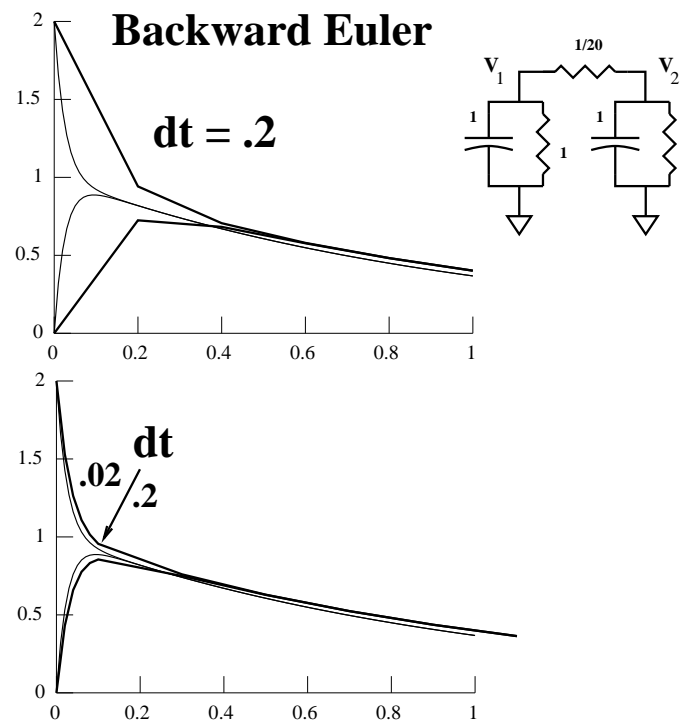
6



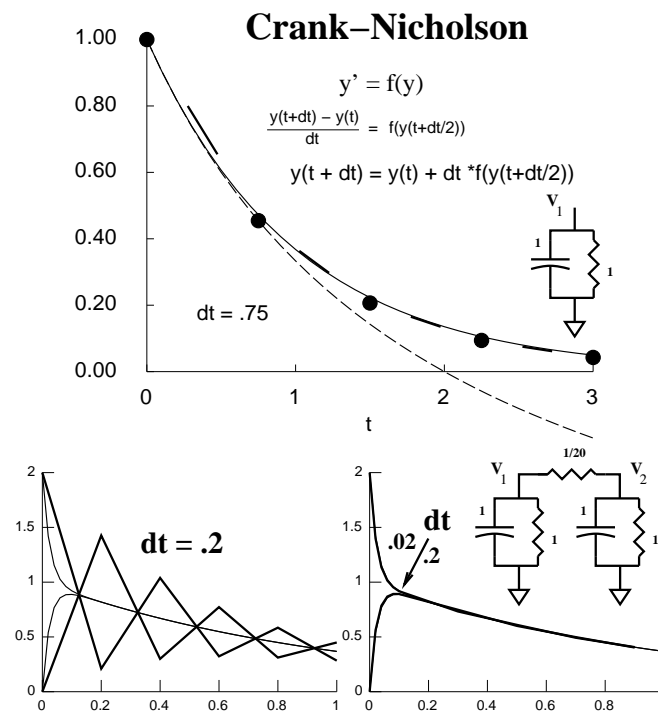
7



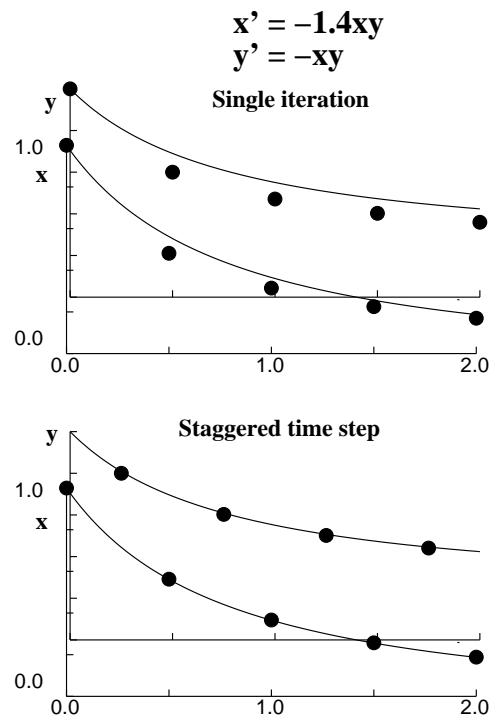
8



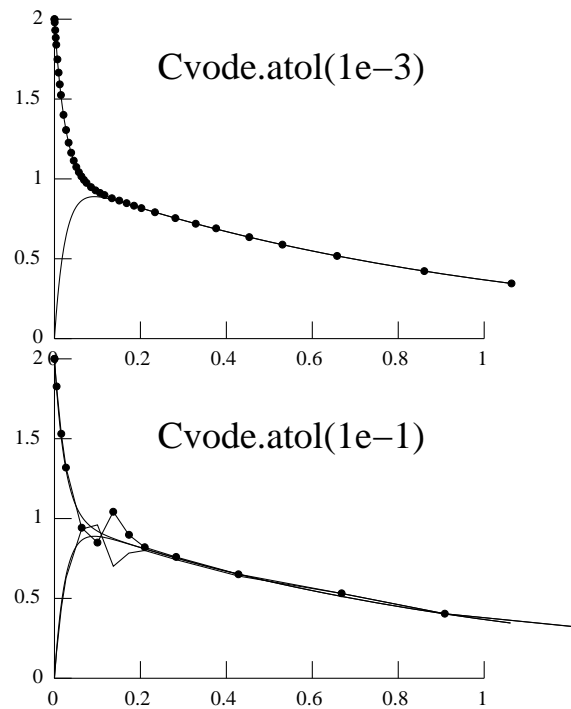
9



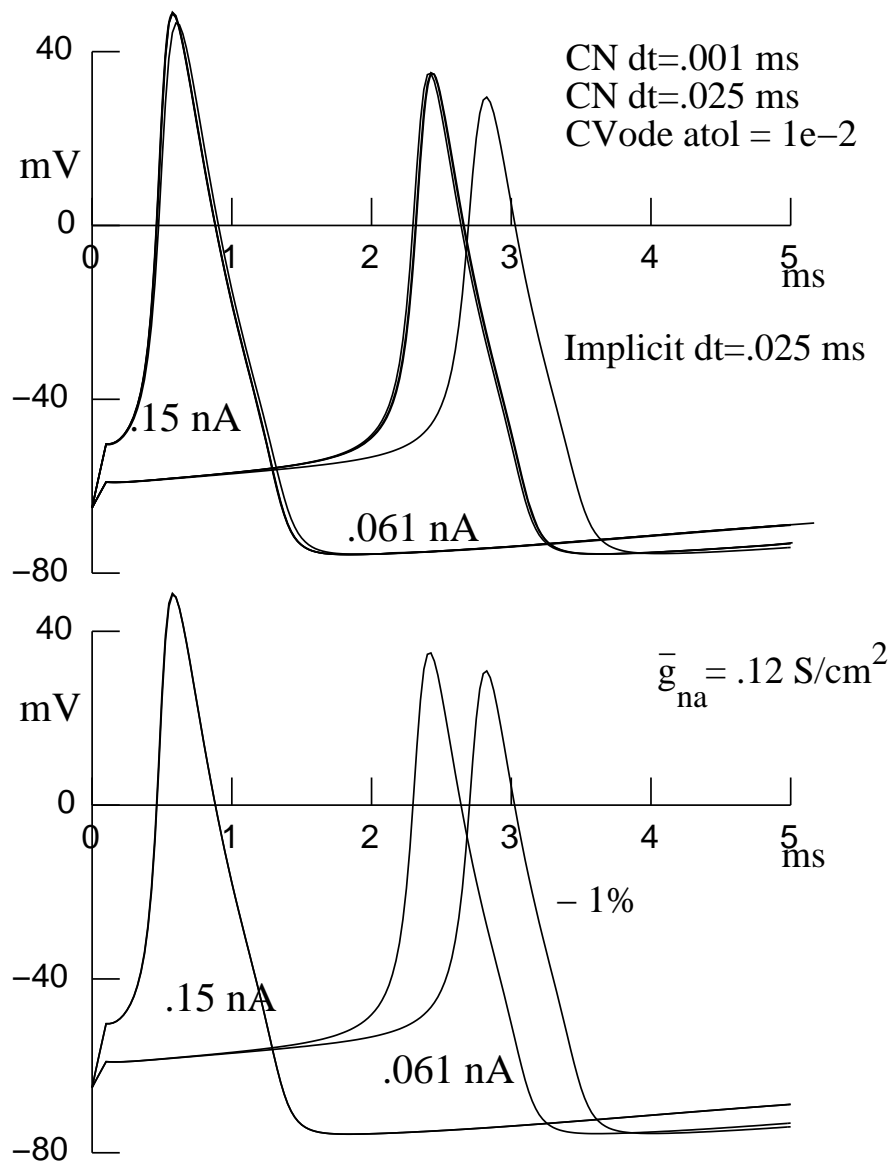
10



11



12



Computational Modeling and Neuroscience

Does computational modeling have a role
in neuroscience research?

1

Best Practices

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

2

Scientific Method

Observation

Hypothesis

Prediction

Verification

Evaluation

3

Reproducibility

The ideal:

"Reproducibility is the cornerstone
of scientific method."

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

The harsh reality:

Velilind's Laws of Experimentation

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

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

4

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



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

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

Find models by **Find models for** **Find models of**

- * [Model name](#)
- * [First author](#)
- * [Each author](#)
- * [Region\(circuits\)](#)
- * [Cell type](#)
- * [Current](#)
- * [Receptor](#)
- * [Gene](#)
- * [Transmitters](#)
- * [Topic](#)
- * [Simulators](#)
- * [Methods](#)
- * [Networks](#)
- * [Neurons](#)
- * [Electrical synapses \(gap junctions\)](#)
- * [Chemical synapses](#)
- * [Ion channels](#)
- * [Neuromuscular junctions](#)
- * [Axons](#)

Search for models by author name or accession number

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

Combine ModelDB curated keywords with a Google search: [ModelSearch](#)

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

[Register](#) for an account

[Login](#) to access your models

Related [Resources](#)

Some models versions are available in a [mercurial repository](#)

5

Accommodates models from a wide range of simulation environments

721 entries as of 6/17/2012

1 BioPAX	2 KinNeSS	1 PSICS
8 Brian	1 Lua	2 PSpice
79 C / C++	155 MATLAB	3 Pascal/Delphi
1 CNrun	1 MCell	2 PyNN
4 CSIM	1 MOOSE / PyMOOSE	13 Python
8 CalC	1 MVASpike	2 Q/Quick/Turbo Basic
1 Catacomb	1 MadSim	1 QuB
1 CellExcite	1 Mathematica	1 R
1 CellML	1 MySQL	1 SABER
2 Chemesis	1 NCS	1 SBML
1 Content	2 NEST	21 SNNAP
1 Dynamics Solver	2 NEURONPM	4 SciLab
1 ERNST	1 Nengo	9 Simulink
3 Emergent/PDP++	1 Network	1 Sspice
6 FORTRAN	1 NeuroRD	1 Topographica
1 GNU/Next/Openstep	345 Neuron	3 Virtual Cell
28 Genesis	1 NeuronExperiment	4 XML
1 IDL	2 Octave	66 XPP
3 IGOR Pro	1 PCSIM	5 neuroConstruct
7 Java	1 PGENESIS	2 parplex

6

Search results

Models by Moore

1. [Frog second-order vestibular neuron models \(Rössert et al. 2011\)](#)
Rössert C, Straka H, Moore LE, Glasauer S (2011) Cellular and network contributions to vestibular signal processing: impact of ion conductances, synaptic inhibition and noise. *J Neurosci* **31**:8359-8372
2. [Engaging distinct oscillatory neocortical circuits \(Vierling-Claassen et al. 2010\)](#)
Vierling-Claassen D, Cardin JA, Moore CI, Jones SR (2010) Computational modeling of distinct neocortical oscillations driven by cell-type selective failure at axon branch points. *J Neurophysiol* **41**:1-8 [PubMed]
11. [Myelinated axon conduction velocity \(Brill et al 1977\)](#)
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]

7

Site of impulse initiation in a neuron (Moore et al 1983)

Accession: 9852

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**:301-311 [PubMed]

Citations [Citation Browser](#)

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

Model Type: [Neuron or other electrically excitable cell](#);
Brain
Region(s)/Organism:
Cell Type(s): [Spinal motor neuron](#);
Channel(s): [I_{Na,t}](#); [I_K](#)
Gap Junctions:
Receptor(s):
Gene(s):
Transmitter(s):
Simulation
Environment: [Neuron](#);
Model Concept(s): [Action Potential Initiation](#); [Simplified Models](#);
Implementer(s): [Hines, Michael](#);

Search NeuronDB for information about: [Spinal motor neuron](#); [I_K](#); [I_{Na,t}](#)

Model files	Download zip file	Auto-launch	Help downloading and running models
<ul style="list-style-type: none"> moore83 README mosinit.hoc init.hoc start.ses 	<p>Moore, Stockbridge, and Westerfield. (1983) On the site of impulse initiation in a neurone. <i>J. Physiol.</i> 336: 301-311.</p> <p>This model qualitatively reproduces figures 1-5. Note that orthodromic stimulus amplitude is considerably different from that noted in the paper. IClamp[0].amp was chosen to give qualitative similarity. We attribute minor quantitative differences to the following:</p> <ol style="list-style-type: none"> 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. <p>The NEURON implementation of this model was prepared by Michael Hines. Questions about details of this implementation should be addressed to him at michael.hines@yale.edu.</p>		

8

How to proceed

Read abstract / paper

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

Analyze model

ModelView

topology(), Shape plot

forall psection()

Read code . . .

Reusable components?

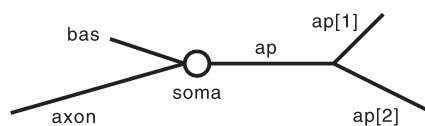
9

Spatially inhomogeneous parameters

Rules

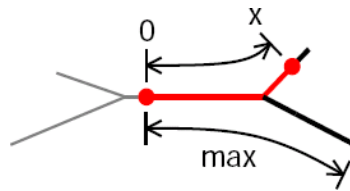
- None (arbitrary values)
- Constant over sets of sections
use SectionLists (CellBuilder Subsets)
- A function of position
hoc or ?

Example: model with hh in apical dendrites



Suppose `gnabar_hh` in the apical tree
decreases linearly with distance from the soma.

Details: 100% at tree origin, 0% at most distant termination.



This example:

$$\text{gnabar_hh} = 0.12 * (1 - p) \text{ where } p = L_{0x}/L_{\text{max}}$$

(normalized path distance from location x
to origin 0 of apical tree)

The general problem: $\text{param} = f(p)$, where f can be any function
and p is a "distance metric" such as:

- path length from a reference point
- radial distance from a reference point
- distance from a plane ("3D projection onto a line")

An equivalent hoc idiom:

forsec subset for (x,0) { rangevar_suffix(x) = $f(p(x))$ }

Conceptualize the task

1. Specify the

subset s

distance metric p

parameter that depends on distance

function f that governs the relationship
between the parameter and p

2. Verify the implementation

How? hoc or GUI (CellBuilder, Model View)

Two kinds of parallel problems

A simulation run takes about a second.

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

A simulation of a large network takes hours.

Want to spread the problem over several machines,
each machine handling a subset of the neurons in the network

Serial

```
s = 0
for i = 1, 10 {
  s += f(i)
}
```

Parallel

```
s = 0
for i = 1, 10 {
  pc.submit("f", i)
}
while (pc.working) {
  s += pc.retval
}
```

Goals

Keep all the machines as busy as possible.

If there is only one machine the parallel program
should run as fast as the serial program.

Things asked for earlier tend to get done earlier.

Assumptions

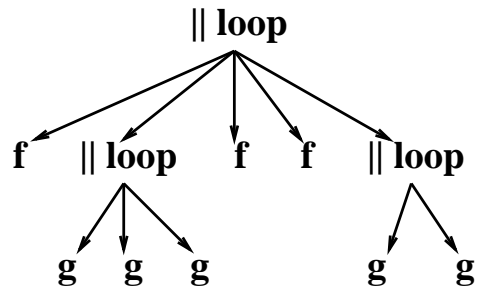
Workstation cluster – 1, 3, 15, 100 machines.

Wide variety of machine speeds.

Sending a byte is much slower than
executing a hoc statement.

Domain

Very coarse grain parallelization.



NEURON's style

A bulletin board
... on top of MPI.

Launching a parallel program

```

objref pc
pc = new ParallelContext()

    // setup which is exactly
    // the same on every machine
    // i.e. declaration of all
    // functions, procedures,
    // setup of neurons

pc.runworker

    // the master scatters tasks
    // onto the bulletin board
    // and gathers results

pc.done
  
```

Example.hoc

```
objref pc
pc = new ParallelContext()

func f() {local s, i
    s = 0
    for i=1,100000 {
        s += $1
    }
    return s
}

{pc.runworker()}

runtime = startsw()
s = 0
for i=1,10 {
    pc.submit("f", i)
}
while (pc.working()) {
    s += pc.retval
}
print "sum = ", s
print "runtime ", startsw() - runtime
{pc.done()}
quit()
```

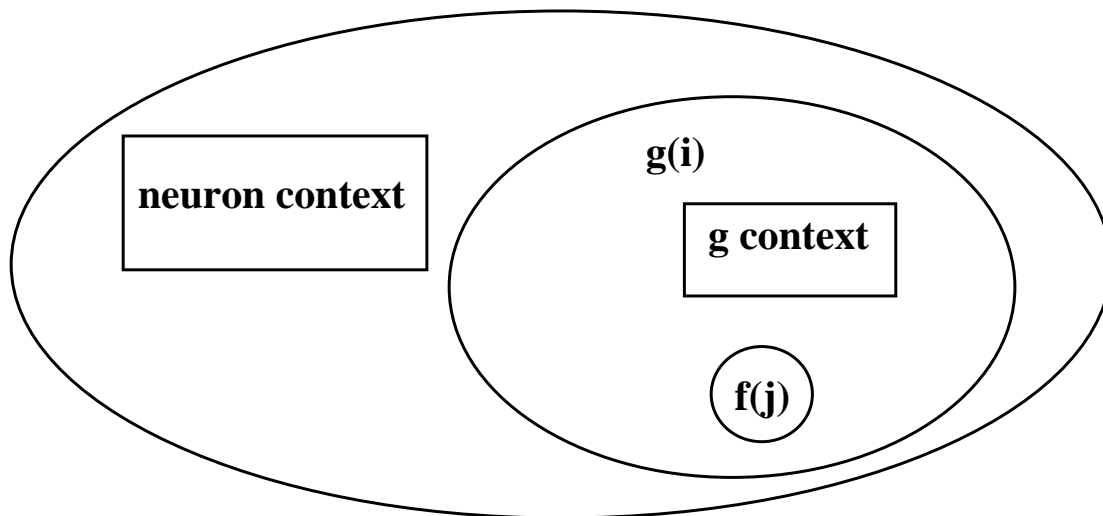
```
$ mpiexec -n 1 nrniv -mpi example.hoc
numprocs=1
NEURON -- VERSION 7.2 (428:986821b56b98) 2010-03-17
...
sum = 5500000
runtime 0.079999924
```

```
$ mpiexec -n 4 nrniv -mpi example.hoc
numprocs=4
NEURON -- VERSION 7.2 (428:986821b56b98) 2010-03-17
...
sum = 5500000
runtime 0.019999981
```

```
$
```

Context and Communication

NEURON



post \longrightarrow **Bulletin board**

take
look \longleftarrow **Bulletin board**
look_take

context("stmt") : stmt executed on every worker

Context and Communication

With Python

```
def f(arg1, arg2):  
    ...  
    return any_pickleable_object  
  
...  
pc.submit(f, (arg1, arg2))  
...  
while pc.working():  
    r = pc.pyret()
```


Supercomputer exercise

(using SDSC's Trestles machine)

The course has 200,000 cpu hours available

<http://www.sdsc.edu/us/resources/trestles/>



[SDSC](#) > [User Support](#) > [Compute and Data Resource Guides](#) > Trestles User Guide

Trestles User Guide: Technical Summary

Trestles is a dedicated [XSEDE](#) cluster designed by Appro and SDSC consisting of 324 compute nodes. Each compute node contains four sockets, each with a 8-core 2.4 GHz AMD Magny-Cours processor, for a total of 32 cores per node and 10,368 total cores for the system. Each node has 64 GB of DDR3 RAM, with a theoretical memory bandwidth of 171 GB/s. The compute nodes are connected via QDR InfiniBand interconnect, fat tree topology, with each link capable of 8 GB/s (bidirectional). Trestles has a theoretical peak performance of 100 TFlop/s.

System Component	Configuration
<i>AMD Magny-Cours Compute Nodes</i>	
Sockets	4
Cores	32
Clock speed	2.4 GHz
Flop speed	307 Gflop/s
Memory capacity	64 GB
Memory bandwidth	171 GB/s
STREAM Triad bandwidth	100 GB/s
Flash memory (SSD)	120 GB
<i>Full System</i>	
Total compute nodes	324
Total compute cores	10,368
Peak performance	100 Tflop/s
Total memory	20.7 TB
Total memory bandwidth	55.4 TB/s
Total flash memory	39 TB
<i>QDR InfiniBand Interconnect</i>	
Topology	Fat tree
Link bandwidth	8 GB/s (bidirectional)
Peak bisection bandwidth	5.2 TB/s (bidirectional)
MPI latency	1.3 μ s
<i>DISK I/O Subsystem</i>	
File Systems	NFS, Lustre
Storage capacity (usable)	150 TB: Dec 2010
	2 PB: June 2011
	4 PB: July 2012
I/O bandwidth	8 GB/s

```
alias trestles='ssh train31@trestles.sdsc.edu'
```

```
In .bashrc
```

```
module load pgi mvapich2
```

```
export PATH=/home/train31/neuron/mpi/x86_64/bin:$PATH
```

Or build NEURON yourself from sources and use
`export PATH=$HOME/neuron/mpi/x86_64/bin:$PATH`

NEURON sources come from

```
cd $HOME/neuron
```

```
hg clone http://www.neuron.yale.edu/hg/neuron/nrn
```

```
cd nrn
```

```
sh build.sh
```

If mercurial or autotools cannot be made available
then get sources as latest tar.gz file from
<http://www.neuron.yale.edu/ftp/neuron/versions/alpha/>
and
`tar xzf nrn*.tar.gz`
`mv nrn-7.2 nrn`

I encapsulate the configure;make;make install in a shell script as sometimes the configure can be very complex.

```
[train31@trestles ~]$ cat notes
#!/bin/sh
#building neuron. sources in $HOME/neuron/nrn
#build and install in $HOME/neuron/mpi

#assume previous 'module load pgi mvapich2'

cd $HOME/neuron/mpi

../nrn/configure --prefix='pwd' --without-x --without-memacs \
--with-paranrn --with-nrnpython \
CC=pgcc CXX=pgCC MPICC=mpicc MPICXX=mpicxx

make -j 4 install

time sh notes
...
161.878u 98.813s 1:53.76 229.1% 0+0k 0+0io 93pf+0w
```

Testing

```
[train31@trestles test]$ cat foo.mod
NEURON { SUFFIX nothing }

nrnivmodl

[train31@trestles test]$ cat test0.hoc
objref pc
pc = new ParallelContext()
{printf("I am %d of %d\n", pc.id, pc.nhost)}
{pc.runworker()}
{pc.done()}
quit()
```

```
[train31@trestles test]$ cat myjob
#!/bin/bash
# the queue to be used.
#PBS -q normal
# specify your project allocation
#PBS -A gue998
# number of nodes and number of processors per node requested
#PBS -l nodes=1:ppn=32
# requested Wall-clock time.
#PBS -l walltime=00:10:00
# name of the standard out file to be "output-file".
#PBS -o job_output
# name of the job
#PBS -N MPI_JOB
#PBS -V
cd $PBS_O_WORKDIR #change to the working directory
mpirun_rsh -np 32 -hostfile $PBS_NODEFILE nrniv -mpi test0.hoc
```

```
qsub myjob
```

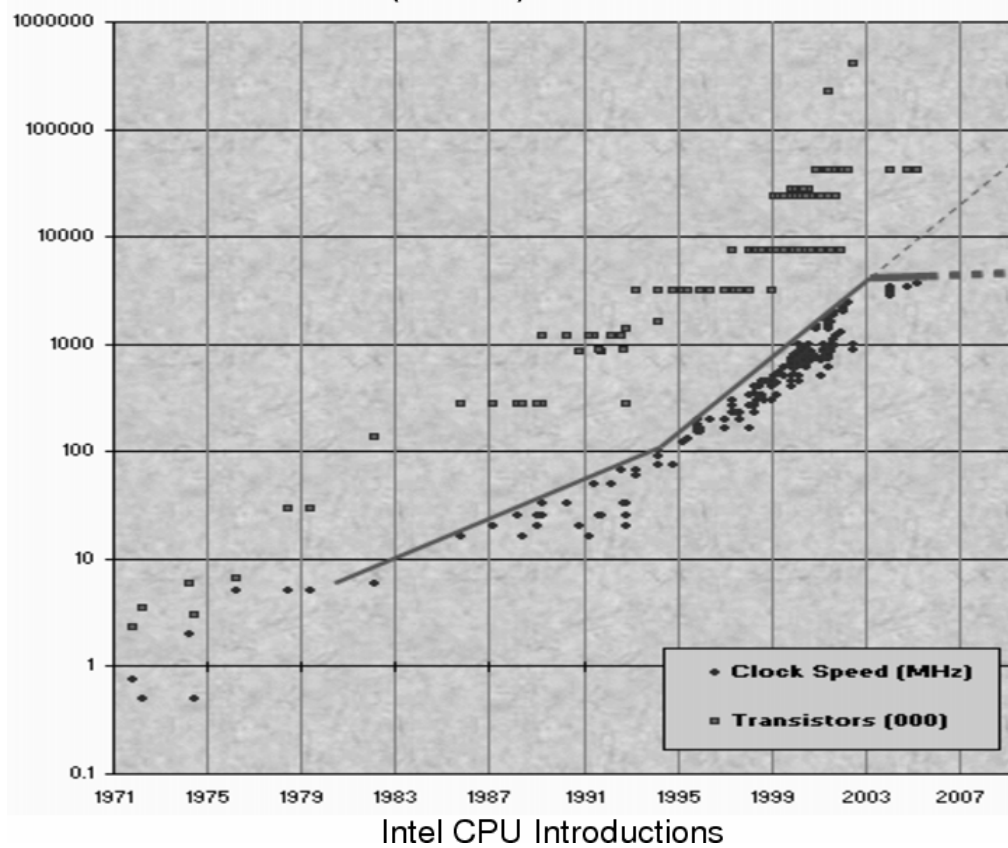
also useful is 'qstat | grep train31'

```
[train31@trestles test]$ cat job_output
numprocs=32
I am 2 of 32
I am 3 of 32
I am 1 of 32
...
I am 20 of 32
I am 28 of 32
I am 29 of 32
Nodes:      trestles-2-24
```


NEURON + Threads

Simulations on multicore desktops.

No (more) Free Lunch



Thread style in NEURON

Join

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

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

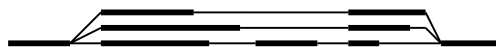


We never use.

Condition Wait

```
multithread_job(run)
```

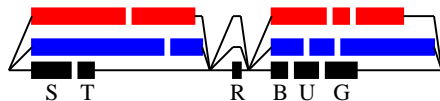
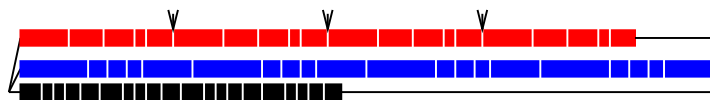
```
run(NrnThread* nt) {
  while(t < tstop) {
    step(nt)
    barrier()
    if (nt->id == 0) { plot() }
    barrier()
  }
}
```



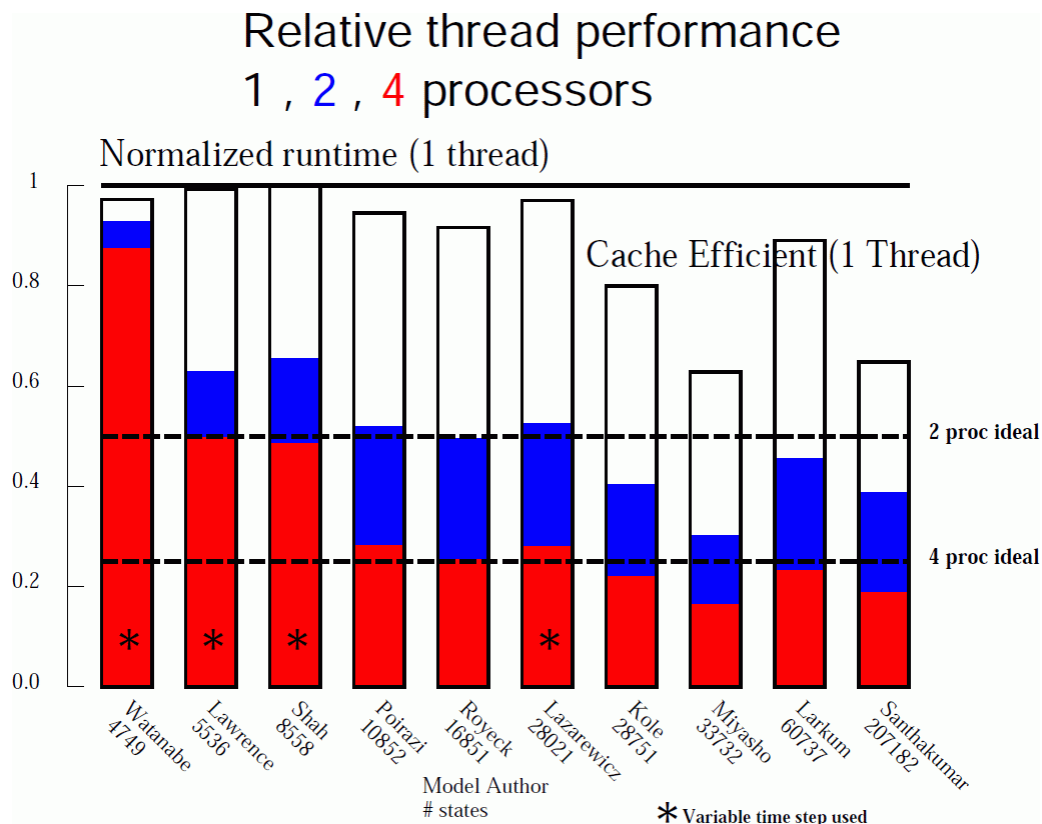
Reminiscent of MPI

Fixed step: $t \rightarrow t + dt$

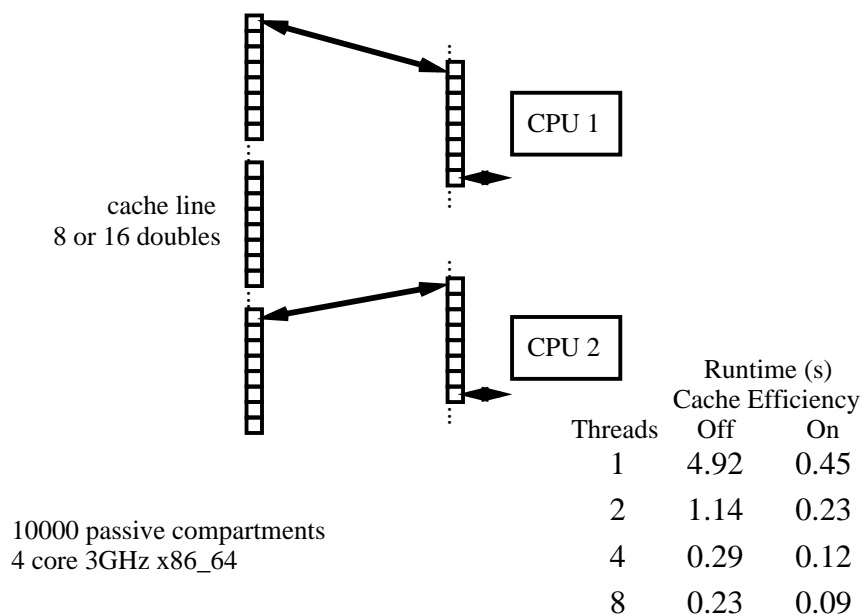
S	T	R	B	U	G
setup	triang	reduce solve	bksub	update	cond gates



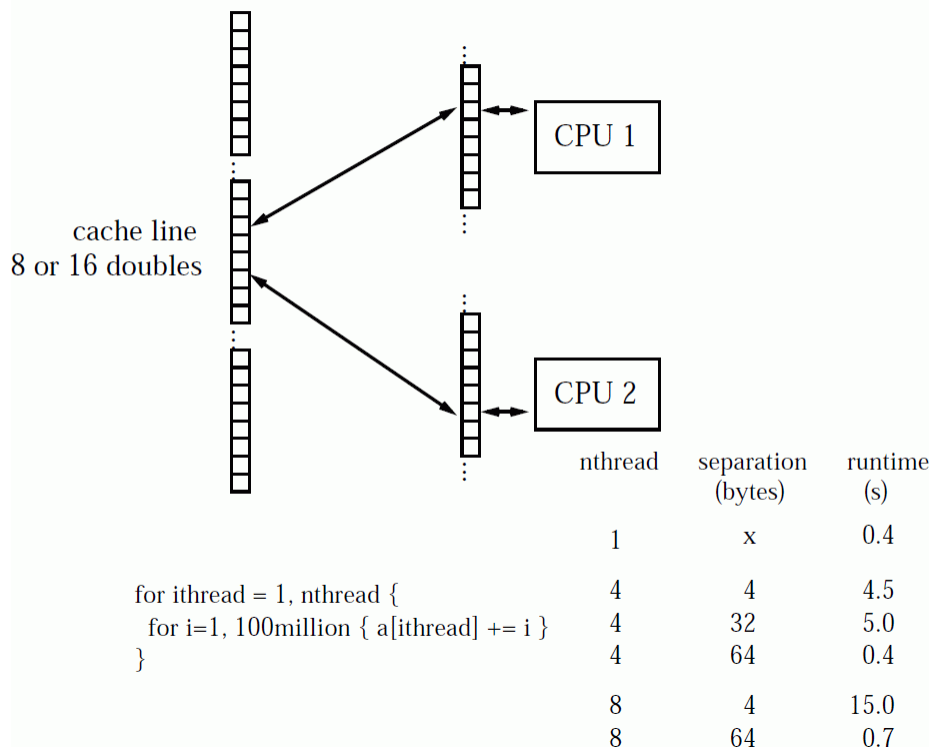
Global var dt $y' = f()$ dy'/dy
27 ||Vector operations



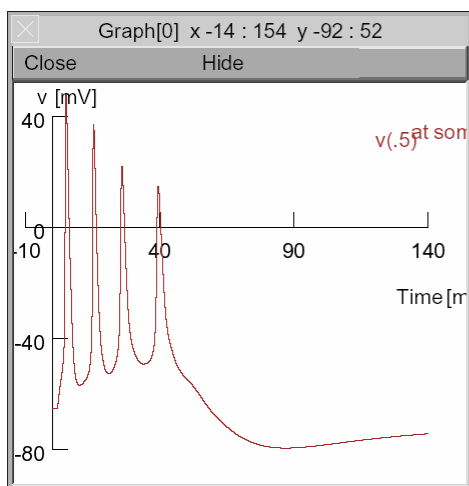
Ideal cache efficiency



False cache line sharing



Lazarewicz 2002, CA3 Pyramidal Neuron



RunControl

Close Hide

Init (mV) -65

Init & Run

Stop

Continue til (ms) 5

Continue for (ms) 1

SingleStep

t (ms) 140

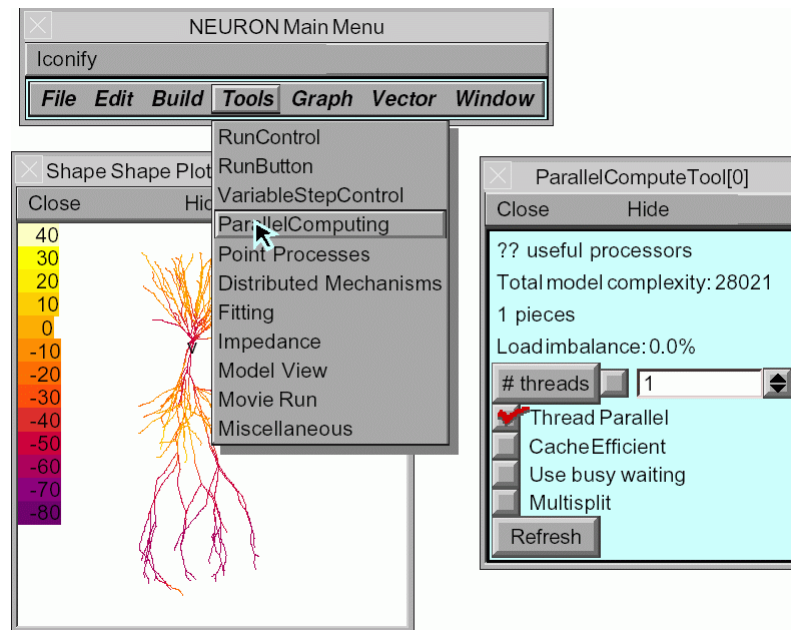
Tstop (ms) 140

dt (ms) ☒ 2.335

Points plotted/ms 40

Scrn update invl (s) 0.05

Real Time (s) 35.4



The first screenshot shows the 'ParallelComputeTool[0]' window with the following configuration:

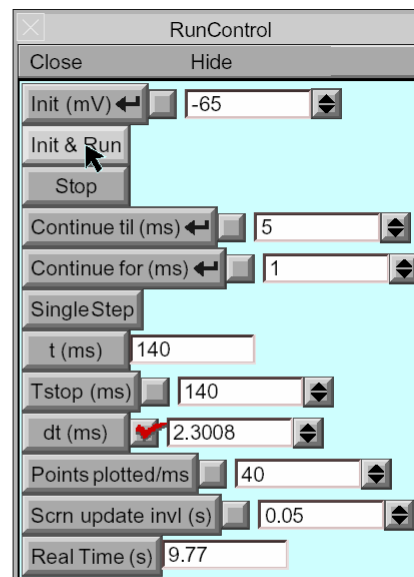
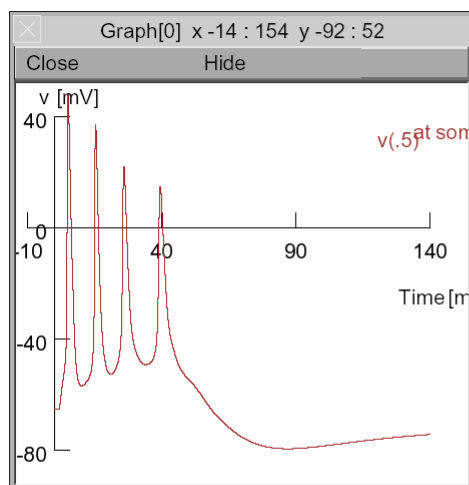
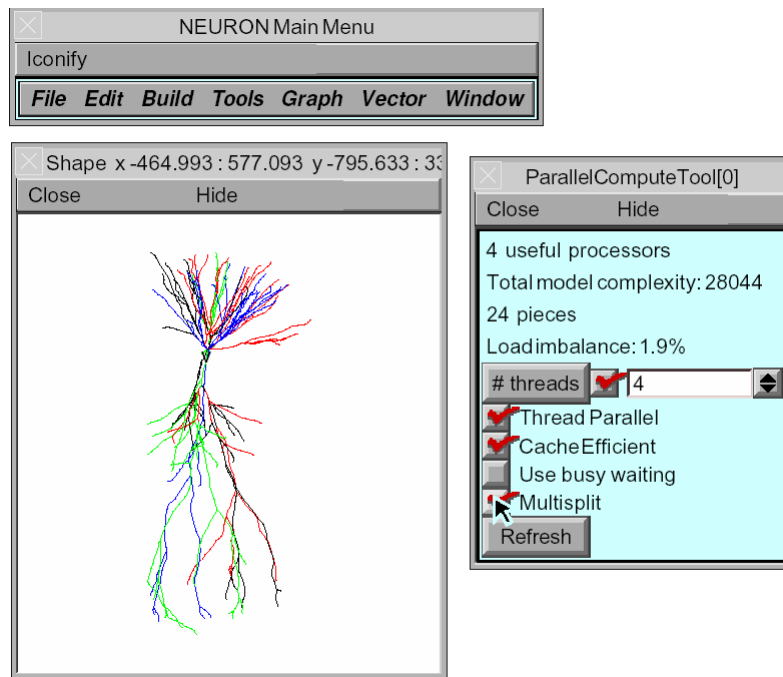
- Close Hide
- 4 useful processors
- Total model complexity: 28021
- 1 pieces
- Load imbalance: 0.0%
- # threads: 1
- ☒ Thread Parallel
- ☐ CacheEfficient
- ☐ Use busy waiting
- ☐ Multisplit
- Refresh

The second screenshot shows the same window with the following configuration:

- Close Hide
- 4 useful processors
- Total model complexity: 28021
- 1 pieces
- Load imbalance: 300.0%
- # threads: 4
- ☒ Thread Parallel
- ☐ CacheEfficient
- ☐ Use busy waiting
- ☐ Multisplit
- Refresh

oc>nthread walltime (count to 1e8 on each thread)

1	0.0500002
2	0.0599999
4	0.0599999
8	0.14



instead of 35.4s

\$ mkthreadsafe

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

Translating CAIM95.mod into CAIM95.c

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

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

Force THREADSAFE? [y][n]: n

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

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

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

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

\$ mkthreadsafe

```

NEURON {
    POINT_PROCESS GABAA
    POINTER pre
    ...
}
    VERBATIM
    return 0;
    ENDVERBATIM

```

Translating gabaa.mod into gabaa.c

Notice: Use of POINTER is not thread safe.

Notice: VERBATIM blocks are not thread safe

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

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

Force THREADSAFE? [y][n]: n

\$ mkthreadsafe

```

NEURON {
    SUFFIX Kv
    USEION k READ ek WRITE ik
    RANGE n, gk, gbar
    RANGE ninf, ntau
    GLOBAL Ra, Rb
    GLOBAL q10, temp, tadj, vmin, vmax
}

```

Translating kv.mod into kv.c

Notice: This mechanism cannot be used with CVODE

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

Force THREADSAFE? [y][n]: n

```

NEURON {
  GLOBAL q10, temp, tadj, vmin, vmax
INITIAL {
  trates(v)
  n = ninf
}
BREAKPOINT {
  SOLVE states
  gk = tadj*gbar*n
  ik = (1e-4) * gk * (v - ek)
}
PROCEDURE trates(v) {
  TABLE ninf, nexp
  tadj = q10^((celsius - temp)/10)

```

```

NEURON { THREADSAFE
  GLOBAL q10, temp, tadj, vmin, vmax
INITIAL {
  trates(v) tadj = q10^((celsius - temp)/10)
  n = ninf
}
BREAKPOINT {
  SOLVE states
  gk = tadj*gbar*n
  ik = (1e-4) * gk * (v - ek)
}
PROCEDURE trates(v) {
  TABLE ninf, nexp
  tadj = q10^((celsius - temp)/10)

```

... a case often seen in ca accumulation models

```

NEURON {
  GLOBAL vol, Buffer0

  ...
  INITIAL {

    if (coord_done == 0) {
      coord_done = 1
      coord()
    }

    ...
    vol[0] = 0
    FROM i=0 TO NANN-2 {
      vol[i] = vol[i] + PI*(r-dr2/2)*2*dr2
    }
    ...
    vol[i+1] = PI*(r+dr2/2)*2*dr2
  }

```

```

NEURON {
  GLOBAL vol, Buffer0
  THREADSAFE vol

  ...
  INITIAL {
    MUTEXLOCK
    if (coord_done == 0) {
      coord_done = 1
      coord()
    }
    MUTEXUNLOCK

    ...
    vol[0] = 0
    FROM i=0 TO NANN-2 {
      vol[i] = vol[i] + PI*(r-dr2/2)*2*dr2
    }
    ...
    vol[i+1] = PI*(r+dr2/2)*2*dr2
  }

```

**If thread results differ,
a good way to diagnose the
cause is to use prcellstate.hoc**

\$ nrngui mosinit.hoc ../prcellstate.hoc

```
// serial model
finitialize(-70)
prtop(0) // constructs cs0.0.1 (5MB)
```

```
//switch to 4 threads
finitialize(-70)
prtop(1) // constructs cs1.0.1
```

diff cs*|more
**notice differences in ik and ica
and in particular**

```
595,605c595,605
< 0 594 0.29053584721744774 gk_km(0.0454545)
< 0 595 0.29053584721744774 gk_km(0.136364)
---
> 0 594 0 gk_km(0.0454545)
> 0 595 0 gk_km(0.136364)
```

```
672,682c672,682
< 0 671 7.8321478840514193e-12 gca_sca(0.0454545)
< 0 672 7.8321478840514193e-12 gca_sca(0.136364)
---
> 0 671 0 gca_sca(0.0454545)
> 0 672 0 gca_sca(0.136364)
```


Initialization, broadly speaking:

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

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

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

The NEURON Simulation Environment

Slide 2

Initialization should assign values at $t = 0$ for

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

and properly configure

- event queues
- vector record and play
- counters

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

NEURON's `finitialize()`

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

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

Default initialization: the standard run library

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

`stdinit()`

Called when you

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

or

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

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

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

init()

Most customizations are made here

```

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

```

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

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

```

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

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

```

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

Kinetic schemes

```

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

```

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

Default initialization of STATES

Use state0, e.g.

```

PARAMETER {
    state0 = 1
}

```

or alternative syntax

```

STATE {
    state START 1
}

```

It's best to be explicit

```

INITIAL {
    m = m0
    h = h0
}

```

To make them visible from hoc

```

NEURON {
    GLOBAL m0
    RANGE h0
}

```

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

Typical custom initializations

Steady state

unperturbed system

system under constant voltage or current clamp

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

Adjust parameters to meet some condition

How?

Use a custom `init()` procedure.

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

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

Initializing to steady state

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

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

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

Initializing to a desired state

Especially useful for oscillating or chaotic models.

Run a "warmup simulation," then save all states

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

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

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

To read from a file

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

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

Initializing to a desired state continued

A custom init() that restores saved states

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

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

Initializing to a particular resting potential

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

Example: for a single compartment model with hh,

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

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

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

Example:

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

UNITS {
  (mA) = (milliamp)
}

PARAMETER {
  ic = 0 (mA/cm2)
}

ASSIGNED {
  i (mA/cm2)
}

BREAKPOINT {
  i = ic
}
```

This needs a different custom init()

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

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

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

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

Using Python to run NEURON

Python has many advantages



- A widely used/known language
- Many packages available (*cf* Matlab)
- Internally consistent o-o syntax (but 2.7 vs 3.1)

()

NPY

2 / 27

Py simulation from NEURON

- All legacy models work
- Better representation of concepts
 - Inheritance: PyrCell is a type of Cell
- Uniform approach to objects
- Data analysis and graphics:
numpy, scipy, pylab ...

()

NPY

3 / 27

Two computer interpreters coexist

- Ideally would stay in 1 language but ...
- Try to maintain name parallelism
- A new level of scoping issues: room for confusion *e.g.*,

```
def myrun ():
    tstop=200
    h.tstop = tstop
    # tstop here is a python local but h.tstop is
    # global since in hoc interpreter
```

()

NPY

4 / 27

Almost everything looks the same in Python and hoc

- Underlying simulation objects are same
- Underlying simulation calls are same
- Various useful and important ancillary things are available (vectors, lists)
 - map 1-to-1 on native Python objects
- Python more regular than hoc *e.g.*,


```
pyr.soma { print v(0.5) } // but 0.5 doesn't belong to v
→
print pyr.soma(0.5).v # it's a location on the soma
```

()

NPY

5 / 27

h: The Hoc object

- `from neuron import h`
- **h** is a function and an object
- **h()** executes arbitrary hoc
- **h.thing** then accesses a thing:
 - A built-in e.g., `h.Vector[0]`
 - Defined via NMODL e.g., `pyr.soma(0.5).g_pas`
 - **h.thing.hname()** tells you what it is

()

NPY

6 / 27

Python is strict about () and []

hoc is not

- `h.vec.size` # `hoc.HocObject...` pointer to a function
- `h.vec.size()` # 50.0 – the size you wanted
- `h.vec.x` # `<hoc.HocObject object at 0x91b93d8>`
- `h.vec.x[0]` # 7.0 – the number you wanted
- `h.vec.hname()` # 'Vector[0]' – the internal name

()

NPY

7 / 27

Introspection

- many kinds of help
- `help('keywords')`
- `dir(h)` # see everything that belongs to hoc
- `dir(soma)`
- `help(soma)`
- Others:
 - `pydoc.apropos()`
 - `help()` # in Python sans neuron

()

NPY

8 / 27

More introspection through inspect

- `import inspect`
- `inspect.isbuiltin(h.Section)` # true
- `inspect.getmembers(h.Section())`
- `inspect.getsourcefile(PyrAdr.set_morphology)`
'geom.py'

()

NPY

9 / 27

An example: define with **h()**

```
>>> h('''u = 5
... strdef s
... s = "hello"
... func square () { return $1*$1 }
... ''')
```

()

NPY

10 / 27

now access with **h.**

- `print h.u,h.s` # Python print
- `h('print u,s')` // hoc print
- `h.printf("%d %s\n",h.u,h.s)`
print here is in hoc but u and s not accessed directly
- `x=h.square(10)`
x is a Python variable
- can get more complicated: e.g., `p.func()` in a .hoc file loaded from Python

()

NPY

11 / 27

Making new things

- eschew the “new”
- *vec=new Vector()* → *vec=h.Vector()*
- *create soma* → *soma=h.Section()*
- *soma insert hh* → *soma.insert('hh')*

()

NPY

12 / 27

Accessing segments

- *soma for (x) gnabar_hh(x)=3e-3*x* →
for seg in soma:
*seg.gnabar_hh=3e-3*seg.x;*
- *Can still access segments relatively:*
soma print gnabar_hh(0.5) →
soma(0.5).hh.gnabar

()

NPY

13 / 27

Placement: membrane mechanisms and point processes

- `soma stim = new IClamp(0.5)`
- `stim = h.IClamp(soma(0.5))`
- *Set params similar to hoc usage:*
`stim.amp=0.1; stim.dur=10; stim.delay=100`

()

NPY

14 / 27

Template in hoc

This will become a class in Python

in hoc:

```
begintemplate Cell
proc init() {
    topology()
    subsets()
    ...
}
...
endtemplate Cell
```

()

NPY

15 / 27

Template (hoc) → class (python)

in Python:

```
class Cell:
    "General cell"
    def __init__(self,x,y,z,id):
        (self.x,self.y,self.z,self.id)= \
            (x,y,z,id)
        self.add_sec('soma',True)
        ...
```

()

NPY

16 / 27

Inheritance: a Pyr is a type of Cell

in Python:

```
class Pyr(Cell):
    def __init__(self):
        self.topology()
        self.subsets()
        ...
```

()

NPY

17 / 27

Setting up topology

```
def topology(self):
    self.add_sec('Bdend')
    self.add_sec('Adend1')
    ...
    self.Bdend.connect(self.soma,0,0)
```

()

NPY

18 / 27

Add compartment

```
def add_sec (self, name, rec=False):
    self.__dict__[name] = h.Section(name=name, cell=self)
    self.all_sec.append( self.__dict__[name])
    if rec:
        self.__dict__[name+"_volt"] = h.Vector(int(h.tstop/h.dt)+1)
        self.__dict__[name+"_volt"].record(\
            self.__dict__[name](0.5)._ref_v)
```

()

NPY

19 / 27

Porting useful neuron shortcuts

forall and forsec

```
#forall syntax - c gets executed, allsecs has Sections
def forall (c):
    """ NEURON forall syntax - iterates through all the sections
    available note that there's a dummy loop variable called s
    used in this function, so any command that needs to access a
    section should be via s. example: forall('print s.name()') ,
    will print all the section names. Also note that this function
    uses a global list, 'allsecs', which may need to get
    re-initialized when new sections are created, via the
    mkallsecs function."""

    global allsecs
    if (type(allsecs)==types.NoneType): mkallsecs()
    for s in allsecs: exec(c)
```

()

NPY

20 / 27

forsec

```
def forsec (secref="soma",command=""):
    """ NEURON forsec syntax - iterates over all sections which
    have a substring in their names matching seceref argument.
    command is executed if match found. This function also
    utilizes the allsecs global variable. """

    global allsecs
    if (type(allsecs)==types.NoneType): mkallsecs()
    if (type(secref)==types.StringTypes[0]):
        for s in allsecs:
            if s.name().count(secref) > 0:
                exec(command)
    else:
        for s in secref: exec(command)
```

()

NPY

21 / 27

mkallsecs()

```
allsecs=None # global list containing all NEURON sections

def mkallsecs ():
    """ mkallsecs - make the global allsecs variable, containing
        all the NEURON sections. Don't call till full sim loaded.
        Utilize via 'for sec in allsecs:'
    """
    global allsecs
    allsecs=h.SectionList() # no .clear() command
    roots=h.SectionList()
    roots.allroots()
    for s in roots:
        s.push()
        allsecs.wholetree()
    return allsecs
```

()

NPY

22 / 27

More background issues

- Compilations
- Python environments
- Levels of objects

()

NPY

23 / 27

Compiling `--with-nrnpython`

- many configuration options to choose from
- `--with-nrnpython`
- may need to set `PYLIBDIR`, `PYINC_DIR`, `PYTHONPATH`, `PYTHONHOME`
- post compile compilation: Python `setup.py`
 - `cd src/nrnpython`
 - `Python setup.py install \`
`--prefix=where_wanted`

()

NPY

24 / 27

Multiple python environments

- `nrniv -python`
- Python: `from neuron import h`
- Python vs iPython
- sage, eclipse, etc.

()

NPY

25 / 27

Potential for confusion

Try to maintain same names

- 2 languages → can make **different things** with **same name**
 - Or same things with different names
- `soma=h.Section(name='soma');`
- `print soma.L,h.soma.L` # different
- To avoid this: **`h('create soma');`**
`soma=h.soma`
- Important for shape plots etc.

()

NPY

26 / 27

Scoping issues

- Scoping important for exploration from the interpreter
- `execfile("geom.py")`
gives direct interpreter access
- `from geom import *`
requires defs in geom.py header
- “Global” in Python subroutine means
access a global

()

NPY

27 / 27

