Table of Contents and Schedule of Presentations

NTC	Ted Carnevale
WWL	Bill Lytton
RAM	Robert McDougal

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

Monday, 6/10 Morning session

Afternoon session

Tuesday, 6/11 Morning session

Time	Speaker	Title	Page
9:00 AM	Q & A		
9:15	NTC	Channel Builder	45
10:30	Coffee Break		
10:45	RAM	Working with morphometric data	53 *
12:00	Lunch		

Afternoon session

1:00	NTC	NMODL: the NEURON Model Description Language	61 *
2:00	RAM	ModelDB and Model View	69 *
3:00	Coffee Break and Free time		
3:15	RAM	Building a model cell	
4:45	Daily wrapup		
5:00	End of afternoon session		

Evening session

7:00	Hands-on exercises, personal projects, and special topics

Wednesday, 6/12 Morning session

Time	Speaker	Title	Page
9:00 AM	Q & A		
9:15	RAM	Reaction-diffusion	89 *
10:30	Coffee Break		
10:45	NTC	Inhomogeneous channel distributions	115 *
12:00	Lunch		

Afternoon session

1:00	NTC / RAM	Families of simulations in parallel	119 *
2:45	Coffee Break		
3:00	Matthew Johnson: Neurostimulation for treament of movement disorders and paralysis		
4:45	Daily wrapup		
5:00	End of afternoon session		

Thursday, 6/13 Morning session

Time	Speaker	Title	Page
9:00 AM	Q & A		
9:15	RAM	Numerical methods: accuracy, stability, speed	125
10:30	Coffee Break		
10:45	NTC	Networks: synapses, events, and artificial spiking cells	139
12:00	Lunch		

Afternoon session

1:00	NTC	Variable time steps and parameter discontinuities	151
2:00	WWL	Introduction to NetPyNE	165
3:00	Coffee Break		
3:15	WWL	Introduction to NetPyNE *continued*	
4:00	Hands-on exercises and personal projects		
4:45	Daily wrapup		
5:00	End of afternoon session		

Evening session

7:00	WWL	Survival in Computational Neuroscience
	Hands-on exercises, personal projects, and special topics	

Friday, 6/14 Morning session

Time	Speaker	Title	Page
9:00 AM	Q & A		
9:15	NTC	Initialization	187 *
10:30	Coffee Break		
10:45	RAM	The hoc programming language	197 *
12:00	Lunch		

Afternoon session

1:00	NTC	Threads	217 *
2:00	RAM	Building a ring network--interactive session	229 **
3:00	Coffee Break		
3:15	RAM	Building a ring network *continued*	
4:45	Daily wrapup		
5:00	End of afternoon session		

Evening session

7:00	Hands-on exercises, personal projects, and special topics

Saturday, 6/15 Morning session

Time	Speaker	Title	Page
9:00 AM	Q & A		
9:15	RAM	Parallel computation: distributed network models	231 *
10:30	Coffee Break		
10:45	NTC	High performance computing via the Neuroscience Gateway Portal	259 *
12:00	Lunch		

Afternoon session

1:00	RAM, NTC	Parallel examples
	WWL	NetPyNE *continued*
3:00	Coffee Break	
3:15	Hands-on exercises and personal projects	
4:30	Wrapup, review, and evaluation (see last page in this booklet)	
5:00	End of afternoon session	

Other material

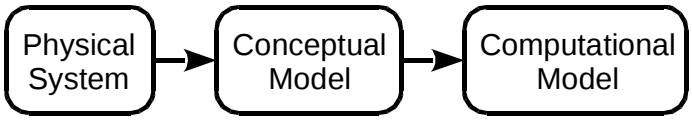
The NEURON Simulation Environment

An intensive hands-on course presented at
University of Minnesota, Minneapolis
June 10 - 16, 2019

N.T. Carnevale, R.A. McDougal, W.W. Lytton

Supported in part by NIH and NSF

The Workflow



The modeler's tasks:

- create the computational model
- investigate/explore/control/use that model

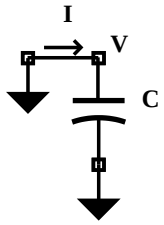
Lipid Bilayer

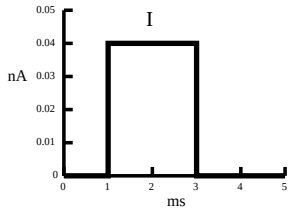
Physical System

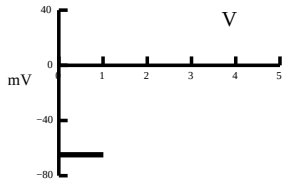
Membrane with no channels

Model

Capacitor







Simulation

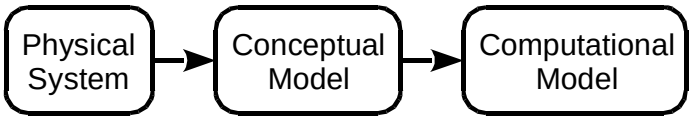
Representation

create soma

Topics

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

From Physical System to Computational Model



Conceptual model

a simplified representation of the physical system

Computational model

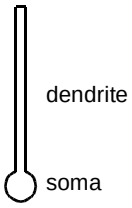
an accurate representation of the conceptual model

From Physical System to Computational Model

Physical system

Ca1 pyramidal cell

Conceptual model



ball and stick

Computational model

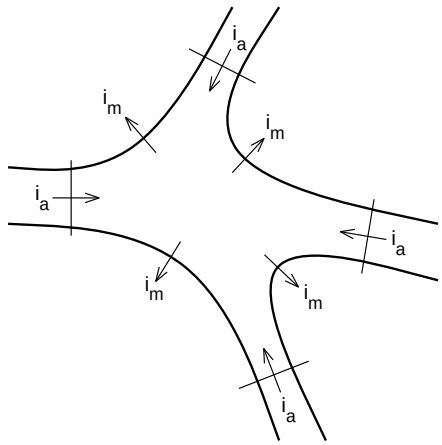
```
# python
soma = h.Section(name='soma')
dendrite = h.Section(name='dendrite')
dendrite.connect(soma(1))
```

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

Fundamental Concepts

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

Conservation of Charge



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

Example: Single Compartment

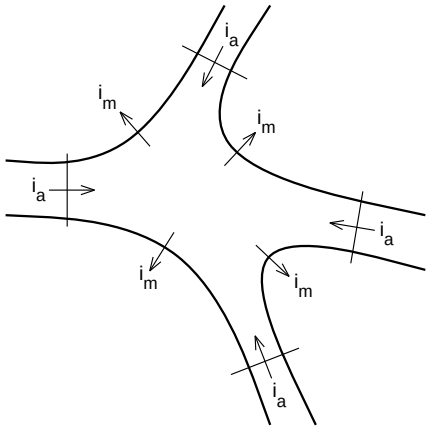
Lipid bilayer (no channels)

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

Conservation of Charge



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

The Model Equations

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

v_j membrane potential in compartment j

i_{ion_j} net transmembrane ionic current in compartment j

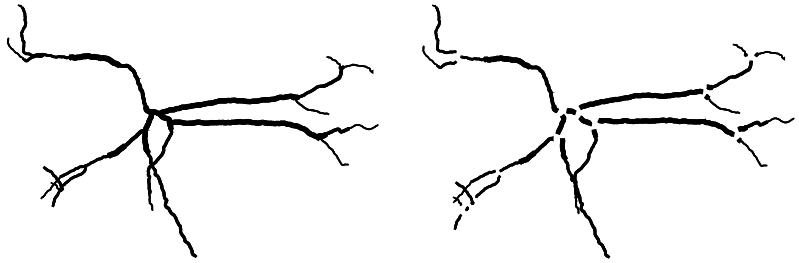
c_j membrane capacitance of compartment j

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

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

$$\frac{dm}{dt} = -\alpha_m m + \beta_m(1-m) \quad \alpha_m = \frac{0.1(V+40)}{1-e^{-0.1(V+40)}} \quad \beta_m = 4e^{-(V+65)/18}$$

$$\frac{dh}{dt} = -\alpha_h h + \beta_h(1-h) \quad \alpha_h = 0.07e^{-0.05(V+65)} \quad \beta_h = \frac{1}{1+e^{-0.1(V+35)}}$$

$$\frac{dn}{dt} = -\alpha_n n + \beta_n(1-n) \quad \alpha_n = \frac{0.01(V+55)}{1-e^{-0.1(V+55)}} \quad \beta_n = 0.125e^{-(V+65)/80}$$

Model

Hodgkin-Huxley cable equations

$$\frac{D}{4R_a}\frac{\partial^2 V}{\partial x^2} = C_m\frac{\partial V}{\partial t}$$

$$+ \bar{g}\,m^3 h\cdot(V-E_{na}) + \bar{g}_k n^4\cdot(V-E_k) + g_l\cdot(V-E_l)$$

$$\frac{dm}{dt} = -\alpha_m m + \beta_m(1-m) \quad \alpha_m = \frac{0.1(V+40)}{1-e^{-0.1(V+40)}} \quad \beta_m = 4e^{-(V+65)/18}$$

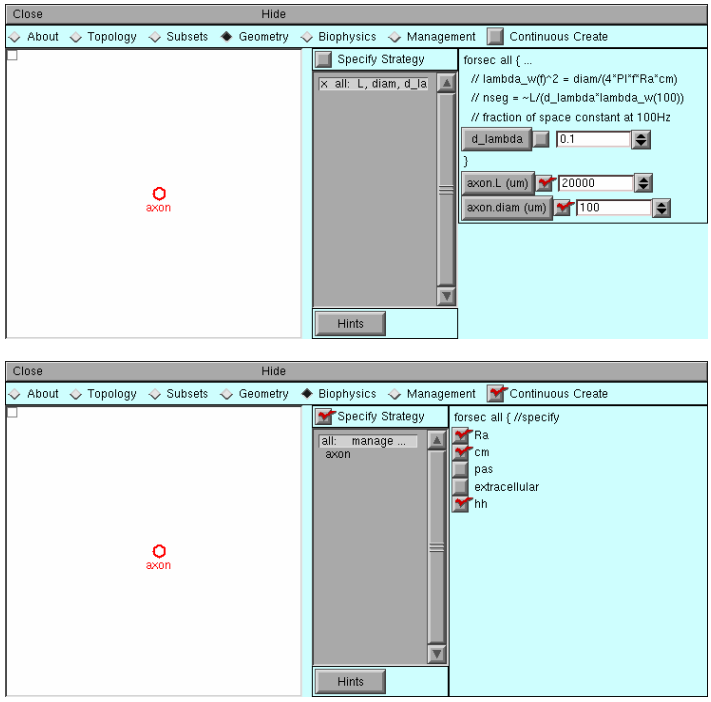
$$\frac{dh}{dt} = -\alpha_h h + \beta_h(1-h) \quad \alpha_h = 0.07e^{-0.05(V+65)} \quad \beta_h = \frac{1}{1+e^{-0.1(V+35)}}$$

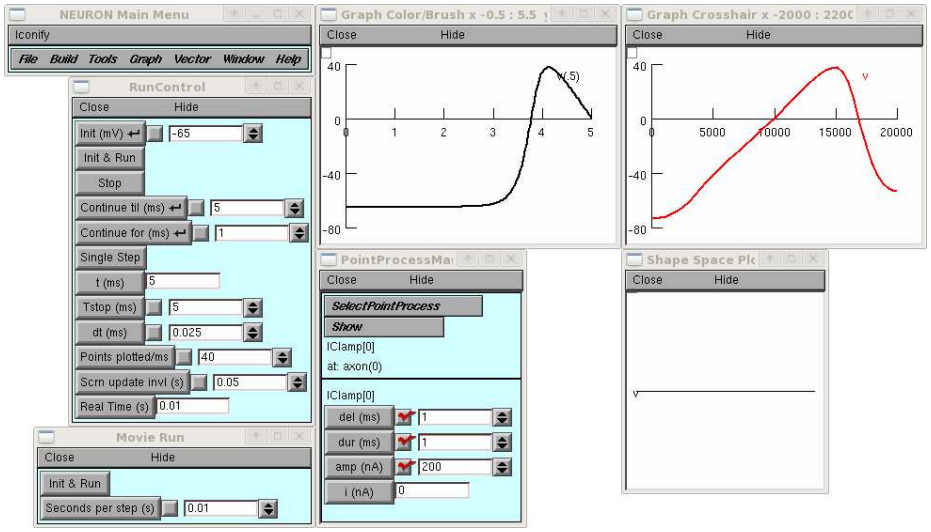
$$\frac{dn}{dt} = -\alpha_n n + \beta_n(1-n) \quad \alpha_n = \frac{0.01(V+55)}{1-e^{-0.1(V+55)}} \quad \beta_n = 0.125e^{-(V+65)/80}$$

Simulation

Representation

```
axon = h.Section(name = 'axon')
axon.L = 2e4
axon.diam = 100
axon.nseg = 43
axon.insert('hh')
```



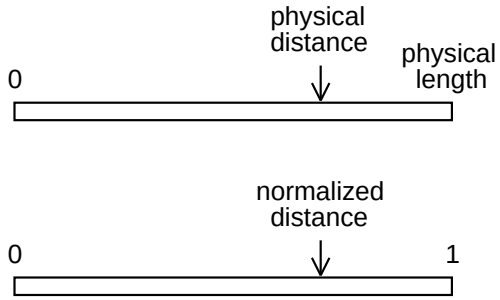
Range Variables

Name	Meaning	Units
`diam`	diameter	[µm]
`cm`	specific membrane capacitance	[µf/cm^2]
`g_pas` (hoc) `pas.g` (Python)	specific conductance of the `pas` mechanism	[siemens/cm^2]
`v`	membrane potential	[mV]

range

normalized position along the length of a section

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



Syntax:

secname(*range*).*rangevar*

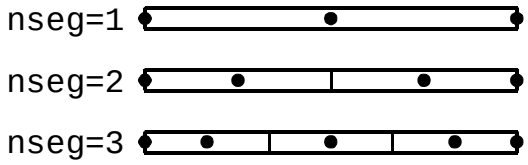
Translation: "in *secname*
at the location corresponding to *range*
access the value of *rangevar*"

Examples:

```
# v at middle of dend
dend(0.5).v # shortcut: dend.v

# at each point in dend
# where v is calculated
# print range, anat distance, and v
for seg in dend.allseg():
  print seg.x, seg.x*dend.L, dend(seg.x).v
```

`nseg`

the number of points in a section at which
the discretized cable equation is integrated



Example: `axon.nseg = 3`

To test spatial resolution

```
for sec in h.allsec():
  sec.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

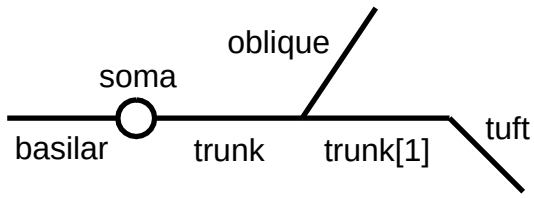
Physical system: anatomically complex cell

Conceptual model: "stick figure"

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

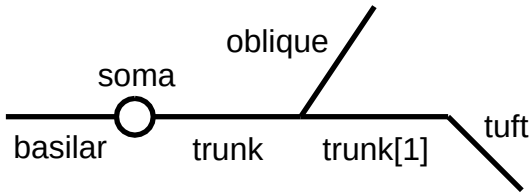
Project goals:

- 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
  }
}
```


Scripting NEURON

Robert A. McDougal

Yale School of Medicine

11 June 2019

What is a script?

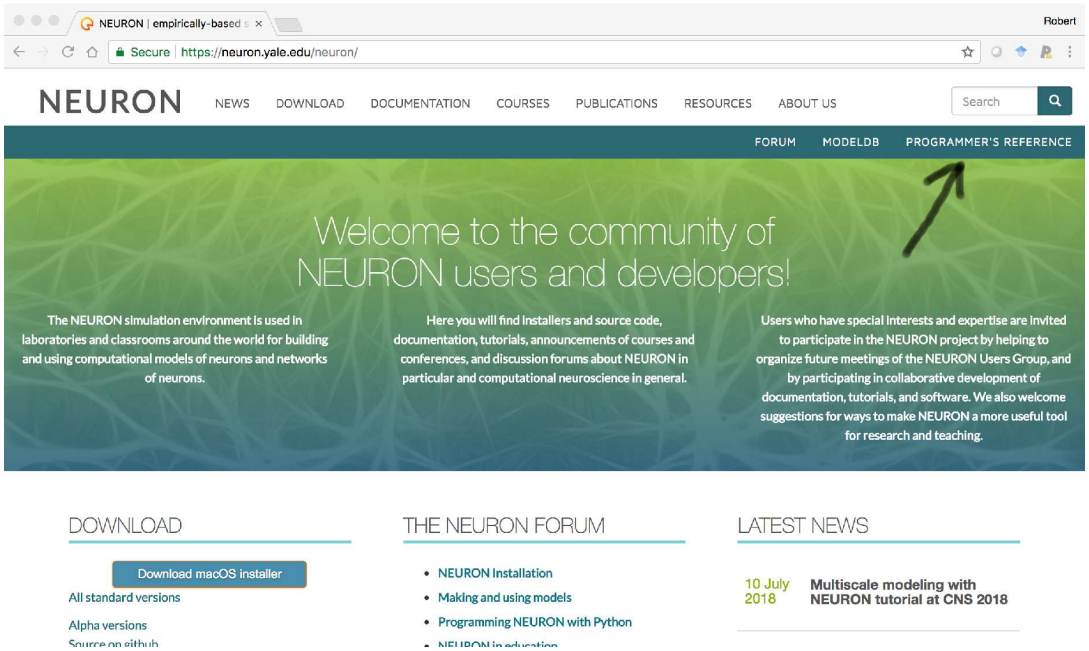
A **script** is a file with computer-readable instructions for performing a task.

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

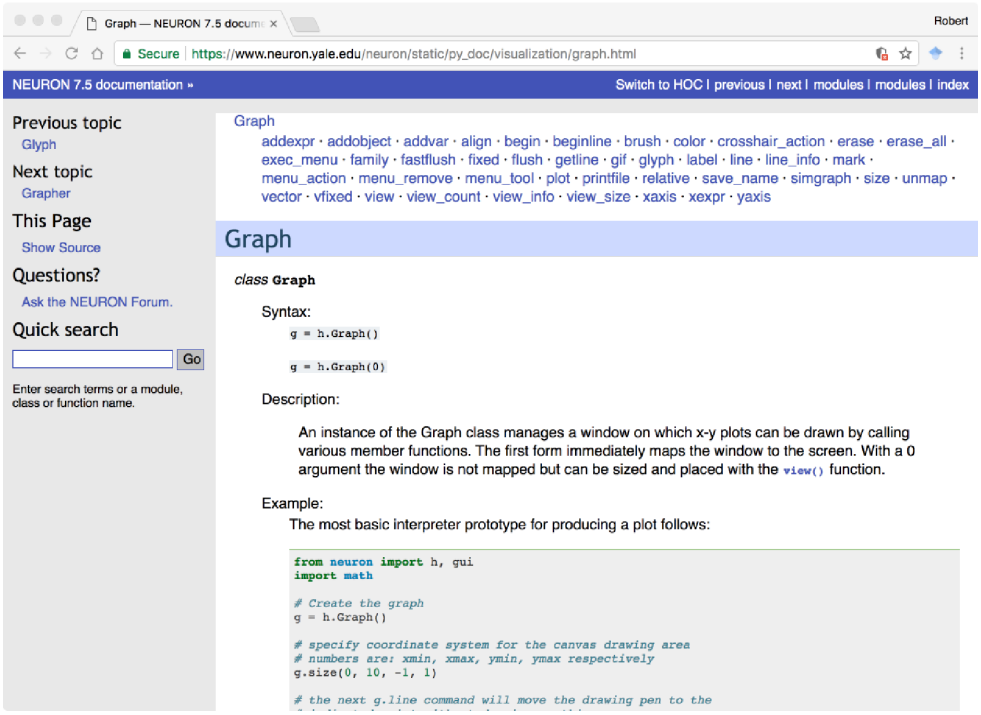
Why write scripts for NEURON?

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

Programmer's Reference



neuron.yale.edu



Use the "Switch to HOC" link in the upper-right corner of every page if you need documentation for HOC, NEURON's original programming language. HOC may be used in combination with Python: use `h.load_file` to load a HOC library; the functions and classes are then available with an `h.` prefix.

Introduction to Python

Displaying results

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

It can display fixed text:

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

```
Hello everyone.
```

or the results of a calculation:

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

```
25
```

Storing results

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

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

```
5.2
```

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

Don't repeat yourself

Lists and for loops

To do the same thing to several items, put the items in a list and use a `for` loop:

```
numbers = [1, 3, 5, 7, 9]
for number in numbers:
    print(number * number)
```

```
1 9 25 49 81
```

Items can be accessed directly using the [] notation; e.g. `n = number[2]`

To check if an item is in a list, use `in`:

```
print(4 in [3, 1, 4, 1, 5, 9])
print(7 in [3, 1, 4, 1, 5, 9])
```

```
True
False
```

Dictionaries

If there is no natural order, specify your own keys using a dictionary.

```
data = {'soma': 42, 'dend': 14, 'axon': 'blue'}
print(data['dend'])
```

```
14
```

Don't repeat yourself

Functions

If there is a particularly complicated calculation that is used once or a simple one used at least twice, give it a name via `def` and refer to it by the name. Return the result of the calculation with the `return` keyword.

```
def area_of_cylinder(diameter, length):
    return 3.14 / 4 * diameter ** 2 * length

area1 = area_of_cylinder(2, 100)
area2 = area_of_cylinder(10, 10)
```


Using libraries

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

```
import math
```

Use dot notation to access a function from the module:

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

0.5

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

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

Other modules

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

Getting help

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

```
from neuron import h
print(dir(h))
```

Displays:

```
['APCount', 'AlphaSynapse', 'BBSaveState', 'CVode', 'DEG', 'Deck',
 'E', 'Exp2Syn', 'ExpSyn', 'FARADAY', 'FInitializeHandler',
 'File', 'GAMMA', 'GUIMath', 'Glyph', 'Graph', 'HBox', 'IClamp',
 'Impedance', 'IntFire1', 'IntFire2', 'IntFire4', 'KSChan', ...]
```

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

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

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

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

Basic NEURON scripting

Creating and naming sections

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

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

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

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

```
print(a == apical)                                    True
```

Python's `str` function returns the name of a Section:

```
print(str(apical))                                    apical
```

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

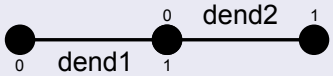
The last print is equivalent to `print(apical)` but `str` was shown to illustrate how to get a string representation.

Connecting sections

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

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

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



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

```
h.topology()
```

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

Example

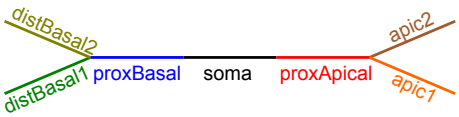
Python script:

```
from neuron import h

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

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

# list topology
h.topology()
```

Output:

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

Morphology:



Length, diameter, and position

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

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

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

To specify the $(x, y, z; d)$ coordinates that a section `sec` passes through, use e.g. `sec.pt3dadd(x, y, z, d)`. The section `sec` has `sec.n3d()` 3D points; their `i`th x-coordinate is `sec.x3d(i)`. The methods `.y3d`, `.z3d`, and `.diam3d` work similarly.

Warning: the default diameter is based on a squid giant axon and is not appropriate for modeling mammalian cells. Likewise, the temperature (`h.celsius`) is by default 6.3 degrees (appropriate for squid, but not for mammals).

Tip: Define a cell inside a class

Consider the code

```
class Pyramidal:
    def __init__(self):
        self.soma = h.Section(name='soma', cell=self)
```

The `__init__` method is run whenever a new `Pyramidal` cell is created, e.g. via

```
pyr1 = Pyramidal()
```

The soma can be accessed using dot notation:

```
print(pyr1.soma.L)
```

By defining a cell in a class, once we're happy with it, we can create multiple copies of the cell in a single line of code.

```
pyr2 = Pyramidal()
```

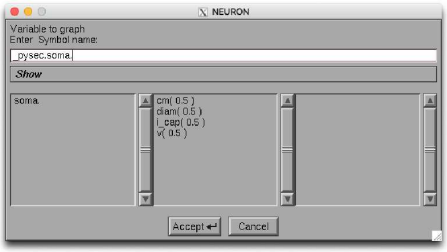
or even

```
pyrs = [Pyramidal() for i in range(1000)]
```


Tip: Sections that work well with GUI tools

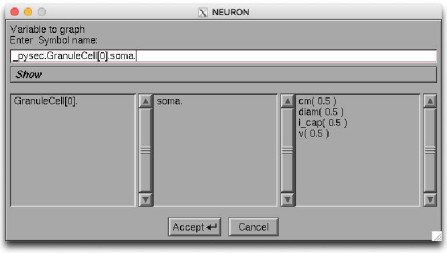
For meaningful Section names to appear in the GUI tools, the `name` attribute must be specified for top-level Sections:

```
soma = h.Section(name='soma')
```



For Sections in cells, specify the `name` of the Section and the `__str__` of the cell:

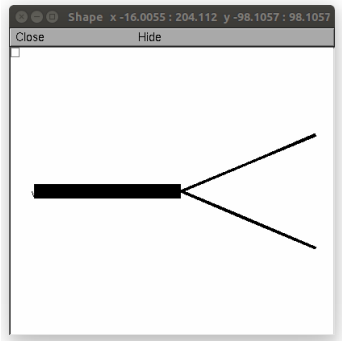
```
class GranuleCell:
    def __init__(self, gid):
        self._gid = gid
        self.soma = h.Section(name='soma', cell=self)
    def __str__(self):
        return 'GranuleCell[{}]'.format(self._gid)

g = GranuleCell(0)
```



To see the list of Sections or cells, select Show > Python Sections.

Viewing the morphology with h.PlotShape

```
from neuron import h, gui

class Cell:
  def __init__(self):
    main = h.Section(name='main', cell=self)
    dend1 = h.Section(name='dend1', cell=self)
    dend2 = h.Section(name='dend2', cell=self)

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

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

    # Important: store the sections
    self.main = main; self.dend1 = dend1
    self.dend2 = dend2

my_cell = Cell()

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



To save the PlotShape `ps` use `ps.printfile('filename.eps')`.

Use the PlotShape.plot method to plot on a matplotlib figure.

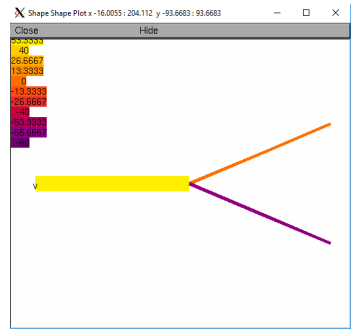
Viewing voltage, sodium, etc

Suppose we make the voltage (`'v'`) nonuniform, which we can do via:

```
my_cell.main.v = 50
my_cell.dend1.v = 0
my_cell.dend2.v = -65
```

We can create a PlotShape that color-codes the sections by voltage:

```
ps = h.PlotShape()
ps.variable('v')
ps.scale(-80, 80)
ps.exec_menu('Shape Plot')
ps.show(0)
```

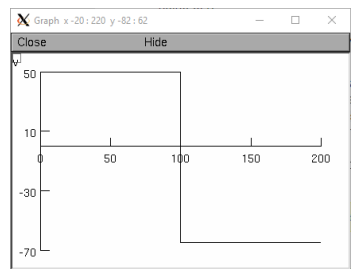


After increasing the spatial resolution:

```
for sec in h.allsec():  sec.nseg = 101
```

We can plot the voltage as a function of distance from `main(0)` to `dend2(1)`:

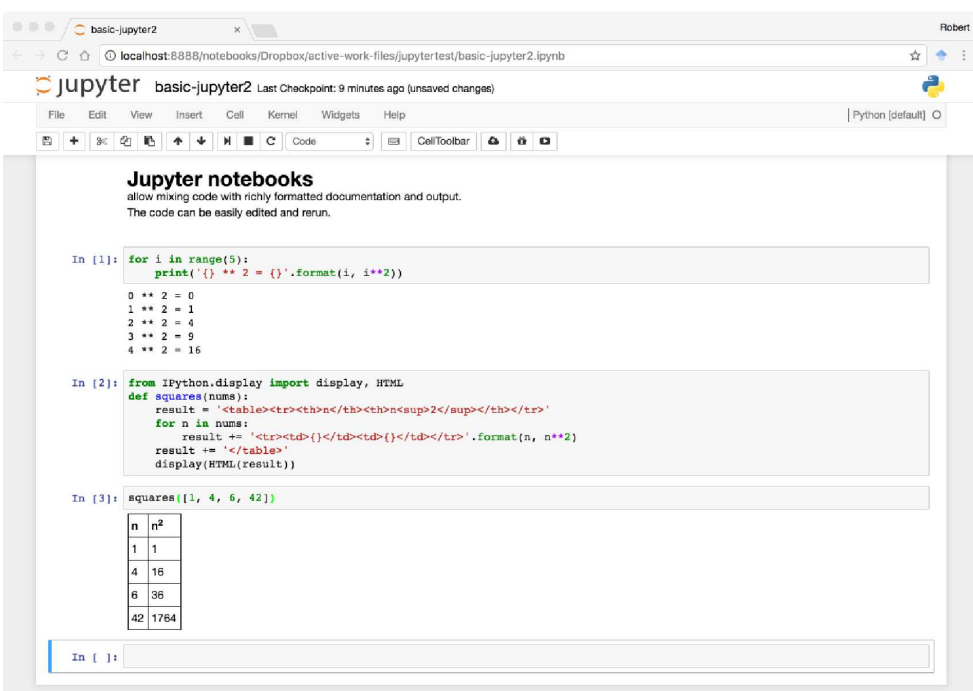
```
rvp = h.RangeVarPlot(
            'v', my_cell.main(0), my_cell.main(1))
g = h.Graph()
rvp.plot(g)
g.exec_menu('View = plot')
```



Sodium concentration could be plotted with `'nai'` instead of `'v'`, etc.

RangeVarPlot.plot can also be used to plot on a matplotlib axis or bokeh.

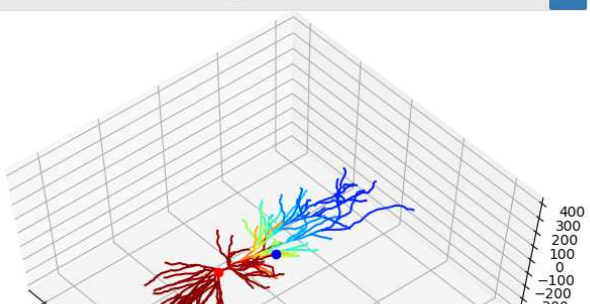
Aside: Jupyter



Aside: Jupyter

```
In [1]: %matplotlib notebook

In [2]: from neuron import h
        from matplotlib import pyplot, cm
        h.load_file('stdrun.hoc')
Out[2]: 1.0

In [3]: h.load_file('geo5038804.hoc')
        for sec in h.allsec():
            sec.insert('hh')

In [4]: ic = h.IClamp(h.soma[0](0.5))
        ic.delay = 0; ic.dur = 1; ic.amp = 5
        h.finitialize(-65)
        h.continuerun(2)
Out[4]: 0.0

In [5]: ps = h.PlotShape(False)
        ps.plot(pyplot, cmap=cm.jet).mark(h.soma[0](0.5)).mark(h.apical_dendrite[68](1), marker='ob')
```

Figure 1



Loading morphology from an swc file

To create `pyr`, a Pyramidal cell with morphology from the file `c91662.swc`:

```
from neuron import h, gui
h.load_file('import3d.hoc')

class Pyramidal:
    def __init__(self):
        self.load_morphology()
        # do discretization, ion channels, etc
    def load_morphology(self):
        cell = h.Import3d_SWC_read()
        cell.input('c91662.swc')
        i3d = h.Import3d_GUI(cell, 0)
        i3d.instantiate(self)

pyr = Pyramidal()
```

`pyr` has lists of Sections: `pyr.apic`, `.axon`, `.soma`, and `.all`. Each Section has the appropriate `.name()` and `.cell()`.

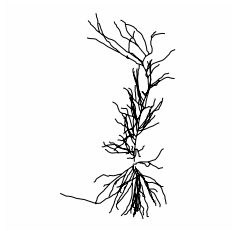
Only do this in code after you've already examined the cell with the Import3D GUI tool and fixed any issues in the SWC file.

Working with multiple cells

Suppose `Pyramidal` is defined as before and we create several copies:

```
mypyrs = [Pyramidal(i) for i in range(10)]
```

We then view these in a shape plot:



Where are the other 9 cells?

Working with multiple cells

To create a method to reposition a cell and call it from `__init__`:

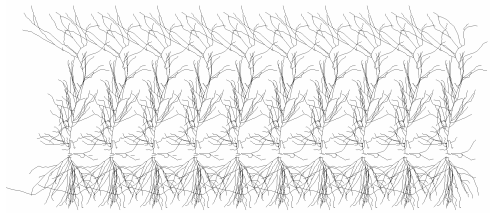
```
class Pyramidal:
  def _shift(self, x, y, z):
    soma = self.soma[0]
    n = soma.n3d()
    xs = [soma.x3d(i) for i in range(n)]
    ys = [soma.y3d(i) for i in range(n)]
    zs = [soma.z3d(i) for i in range(n)]
    ds = [soma.diam3d(i) for i in range(n)]
    for i, (a, b, c, d) in enumerate(zip(xs, ys, zs, ds)):
      soma.pt3dchange(i, a + x, b + y, c + z, d)
```

```
def __init__(self, gid, x, y, z):
  self._gid = gid
  self.load_morphology()
  self._shift(x, y, z)

def load_morphology(self):
  cell = h.Import3d_SWC_read()
  cell.input('c91662.swc')
  i3d = h.Import3d_GUI(cell, 0)
  i3d.instantiate(self)
```

Now if we create ten, while specifying offsets,

```
mypyrs = [Pyramidal(i, i * 100, 0, 0) for i in range(10)]
```

The PlotShape will show all the cells separately:



Does position matter?

Sometimes.

Position matters with:

- Connections based on proximity of axon to dendrite.
- Connections based on cell-to-cell proximity.
- Extracellular diffusion.
- Communicating about your model to other humans.

Distributed mechanisms

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

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

Point processes

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

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

To find the segment containing a point process pp, use

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

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

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

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

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


Setting and reading parameters

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

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

Setting variables works the same way:

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

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

```
apical.nseg = 11
```

To specify the temperature, use `h.celsius`:

```
h.celsius = 37
```

Setting and reading parameters

Often you will want to read or write values on all segments in a section. To do this, use a `for` loop over the Section:

```
for segment in apical:
    segment.hh.gkbar = 0.037
```

The above is equivalent to `apical.gkbar_hh = 0.037`, however the first version allows setting values nonuniformly.

A list comprehension can be used to create a Python list of all the values of a given property in a segment:

```
apical_gkbars = [segment.hh.gkbar for segment in apical]
```

Note: looping over a Section only returns true Segments. If you want to include the voltage-only nodes at 0 and 1, iterate over, e.g. `apical.allseg()` instead.

HOC's for (x,0) and for (x) are equivalent to looping over a section and looping over allseg, respectively.

Running simulations: the basics

To initialize a simulation to -65 mV:

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

To advance a single time step:

```
h.fadvance()
```

For higher-level controls, load the `stdrun.hoc` library:

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

With that library loaded, we can:

Run a simulation until $t = 50$ ms:

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

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

`stdrun.hoc` is loaded automatically during a `from neuron import gui`.

Running simulations: improving accuracy

Increase time resolution (by reducing time steps) via, e.g.

```
h.dt = 0.01
```

Enable variable step (allows error control):

```
h.CVode().active(True)
```

Set the absolute tolerance to e.g. 10^{-5}:

```
h.CVode().atol(1e-5)
```

Increase spatial resolution:

```
sec.nseg = 11
```

To increase nseg for all sections:

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

The default absolute tolerance is 10^{-2}, but with different variables assigned different tolerance scales using `cvode.atolscale` or Tools > VariableStepControl > Atol Scale Tool. Relative tolerance may also be set using `rtol`, but if using that set `atol` to 0 first, otherwise the allowed error will be greater than both; see the programmer's reference for details.

If using the NEURON GUI for plotting, use h.cvode_active(True) to activate CVode to ensure the graphs make the right assumptions about interpreting timesteps; this function is only available when the gui module is loaded.

Recording data

To see how a variable changes over time, create a `Vector` and pass in a pointer (prefix the end of the variable name with `_ref_`) to the record method; e.g. to record `soma(0.3).ina`, use

```
data = h.Vector().record(soma(0.3)._ref_ina)
```

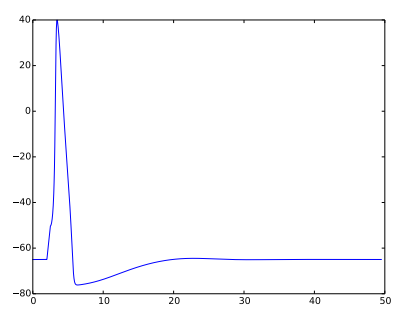
Tips

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

If `v` is a `Vector`, then `v.as_numpy()` provides the equivalent `numpy` array; that is, changing one changes the other.

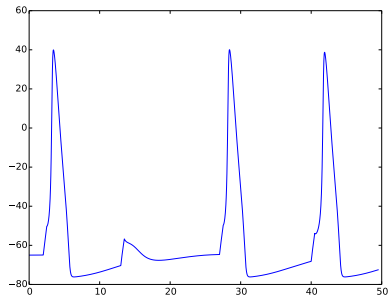
Example: Hodgkin-Huxley

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

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

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

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

# simulation
h.finitialize(-65)
h.continuerun(49.5)

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



Operational definition of a spike: Vm crossing a threshold (by default 10 mV) in a positive-going direction. We could analyze the time series to find this, but NEURON's `NetCon` objects can detect this directly. Changes from the previous example are highlighted.

```
from neuron import h, gui
from matplotlib import pyplot
soma = h.Section(name='soma')
soma.insert('hh')
# current clamps
stim_ts = [2, 13, 27, 40]
iclamps = [h.IClamp(soma(0.5)) for t in stim_t
for t, iclamp in zip(stim_ts, iclamps):
    iclamp.delay = t # ms
    iclamp.dur = 0.5 # ms
    iclamp.amp = 50
# recording
t = h.Vector().record(h._ref_t)
v = h.Vector().record(soma(0.5)._ref_v)
nc = h.NetCon(soma(0.5)._ref_v, None, sec=soma)
spike_times = h.Vector()
nc.record(spike_times)
# simulation
h.finitialize(-65)
h.continuerun(49.5)
print('spike times:')
print(list(spike_times))
# plotting
pyplot.plot(t, v)
pyplot.show()
```



The console displays:

```
spike times:
[3.225000000100012, 28.20000000009893,
41.70000000010092]
```

That is, the cell spiked at: 3.225 ms, 28.200 ms, and 41.700 ms.

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

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

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

The result:

```
[24.97499999999892, 13.50000000000199]
```

That is, the delays between spikes were 24.975 ms and 13.500 ms.

Vector's deriv method can also be used to calculate ISIs: `sis = list(spike_times.c().deriv(1, 1))`

Networks of neurons

Suppose we have the simple neuron model:

```
from neuron import h, gui

class Cell:
    def __init__(self):
        self.soma = h.Section(name='soma', cell=self)
        self.soma.insert('hh')
```

and two cells:

```
neuron1 = Cell()
neuron2 = Cell()
```

one of which is stimulated by a current clamp:

```
ic = h.IClamp(neuron1.soma(0.5))
ic.amp = 50
ic.delay = 2 # ms
ic.dur = 0.5 # ms
```

A synapse from that cell to the other may cause the second cell to fire when the first cell is stimulated. In NEURON, the post-synaptic side of the synapse is a point process; presynaptic threshold detection is done with an `h.NetCon`.

Networks of neurons

Setup the post-synaptic side:

```
postsyn = h.ExpSyn(neuron2.soma(0.5))
postsyn.e = 0  # reversal potential
```

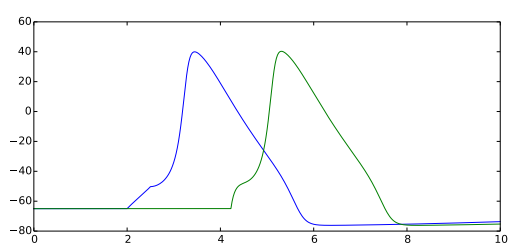
Setup the presynaptic side, transmission delay, and synaptic weight:

```
syn = h.NetCon(neuron1.soma(0.5)._ref_v, postsyn, sec=neuron1.soma)
syn.delay = 1
syn.weight[0] = 5
```

Then we can setup recording, run, and plot as usual:

```
t = h.Vector().record(h._ref_t)
v1 = h.Vector().record(neuron1.soma(0.5)._ref_v)
v2 = h.Vector().record(neuron2.soma(0.5)._ref_v)
```

```
h.finitialize(-65)
h.continuerun(10)

from matplotlib import pyplot
pyplot.plot(t, v1, t, v2)
pyplot.xlim((0, 10))
pyplot.show()
```



`h.ExpSyn` is one of several general synapse types distributed with NEURON; additional ones may be specified in NMODL or downloaded from ModelDB.

The use of `h.NetCon` must be modified slightly to support parallel simulation; this is discussed in a different presentation.

Storing data to CSV to share with other tools

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

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

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

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

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

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

For more information

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

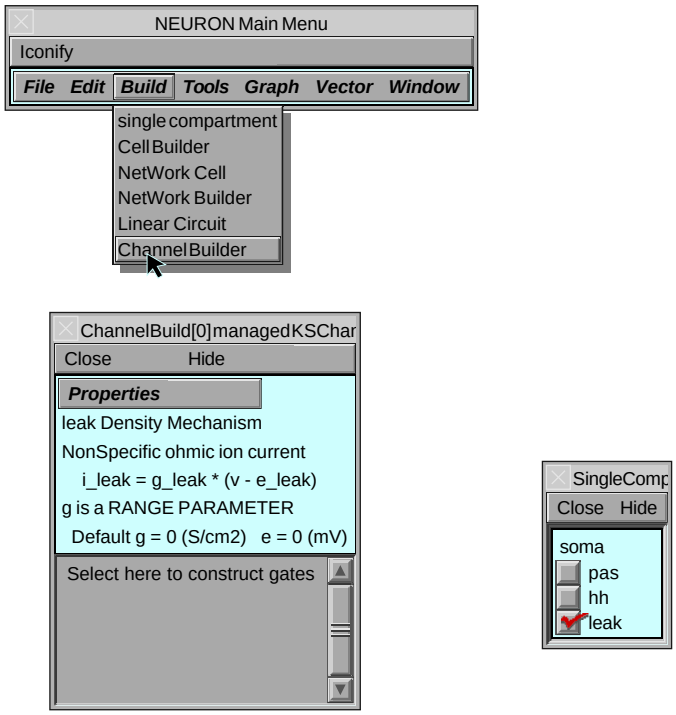
http://neuron.yale.edu/neuron/static/docs/neuronpython/index.html

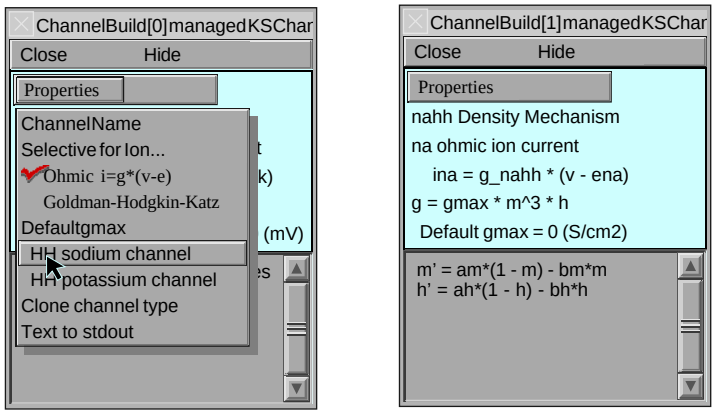
The NEURON Python programmer's reference is available at:

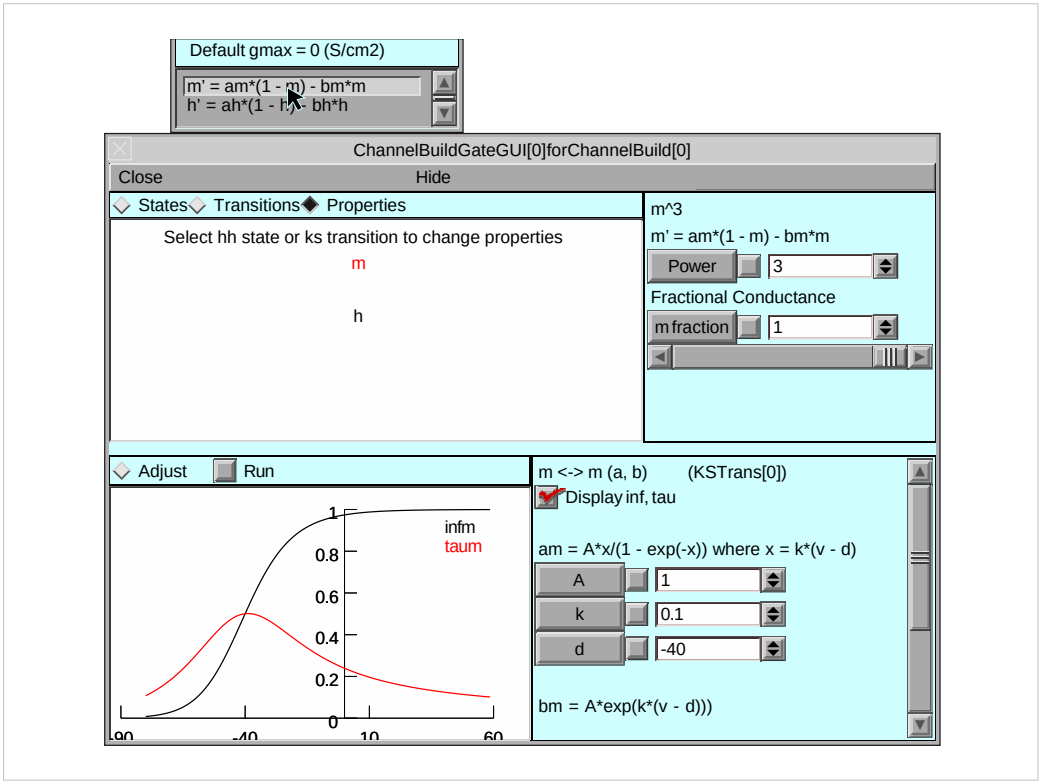
http://neuron.yale.edu/neuron/static/py_doc/index.html

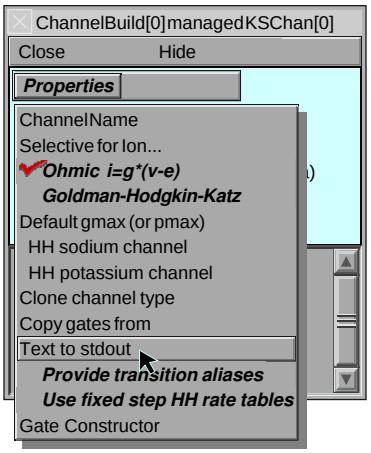
Ask questions on the NEURON forum:

http://neuron.yale.edu/phpbb

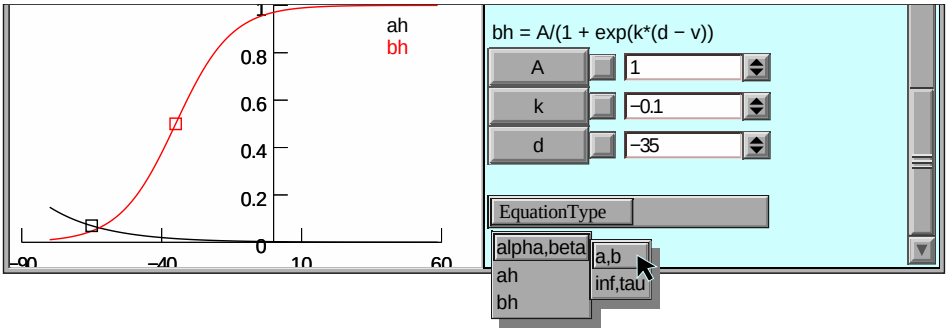


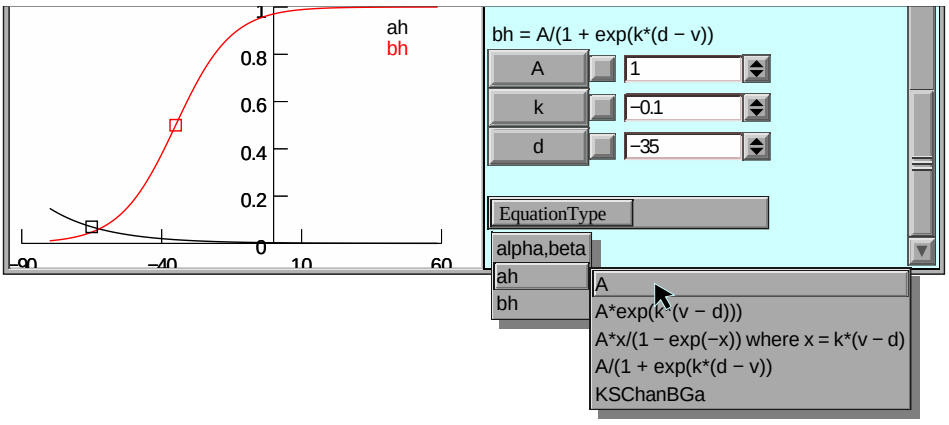


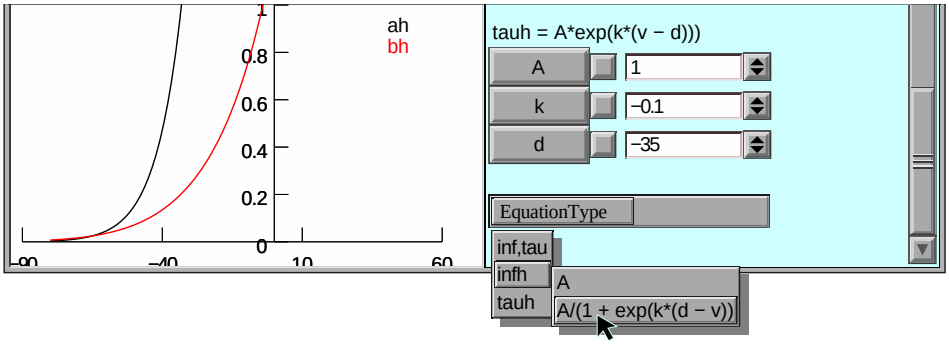


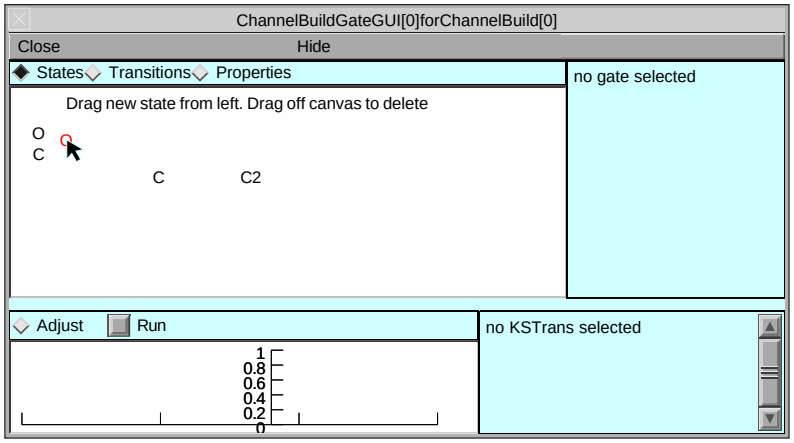


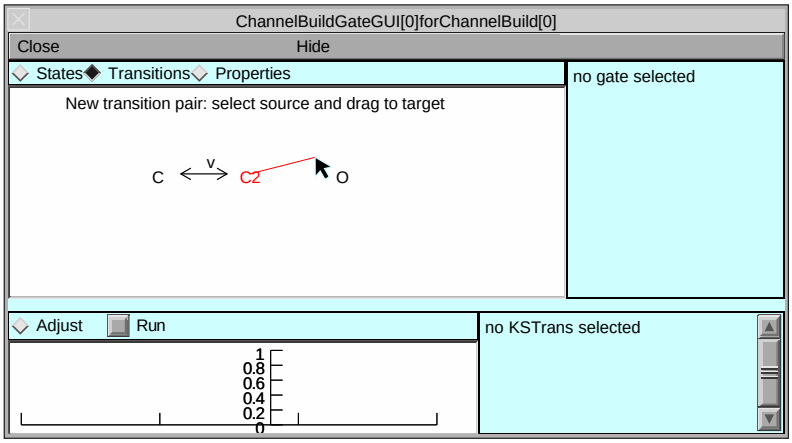

```
nahh Density Mechanism
  na ohmic ion current
    ina (mA/cm2) = g_nahh * (v - ena)
  g = gmax * m^3 * h
    Default gmax = 0 (S/cm2)
    m' = am*(1 - m) - bm*m                          (KSTrans[0])
      am = 1*x/(1 - exp(-x)) where x = 0.1*(v + 40) (Vector[7])
      bm = 4*exp(-0.05556*(v + 65)))                (Vector[8])
    h' = ah*(1 - h) - bh*h                          (KSTrans[1])
      ah = 0.07*exp(-0.05*(v + 65)))                (Vector[11])
      bh = 1/(1 + exp(0.1*(-35 - v)))               (Vector[12])
```

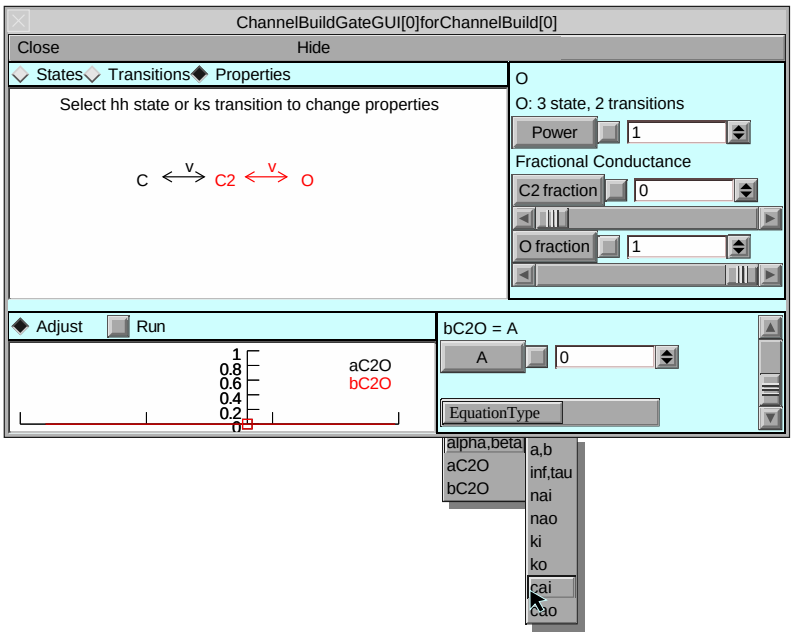



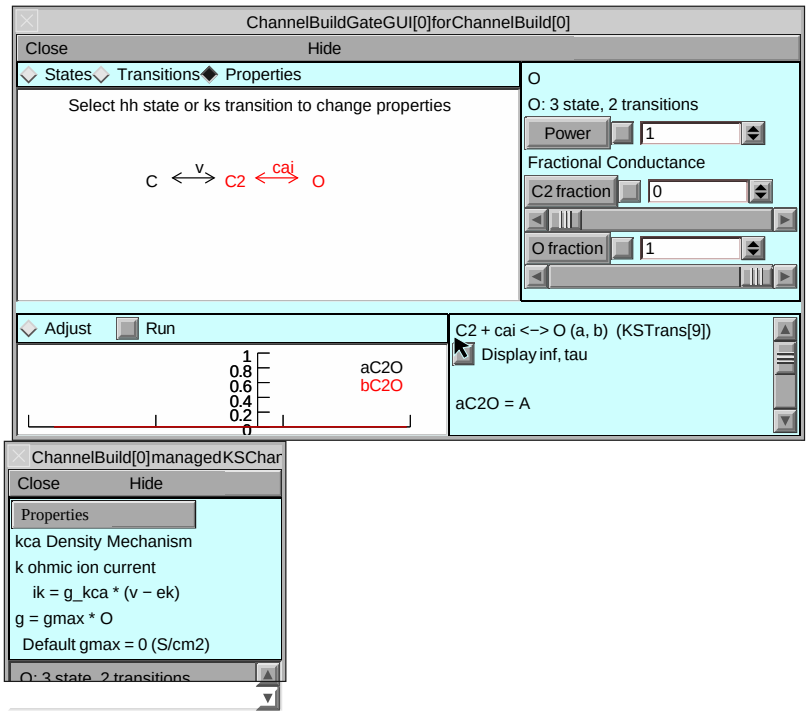


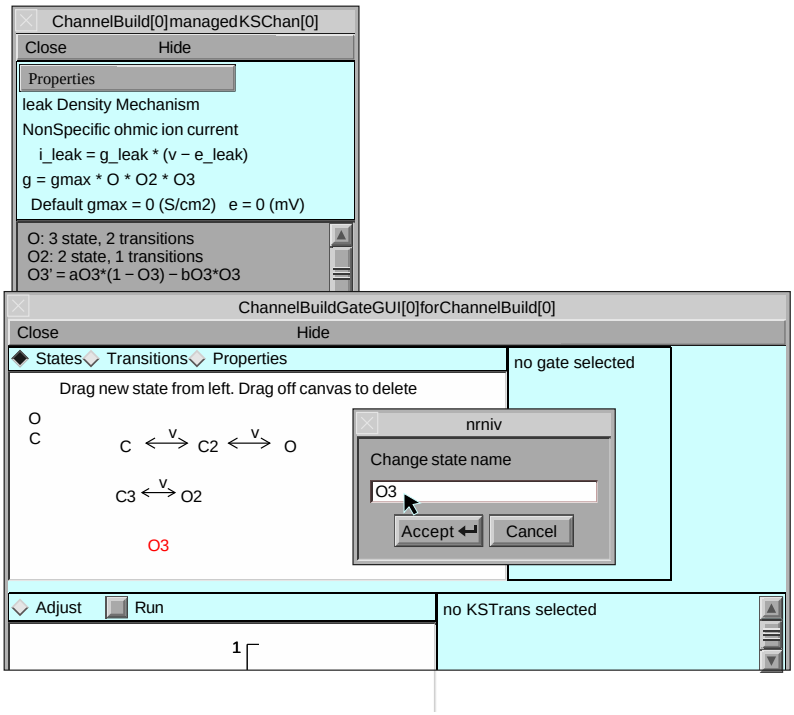


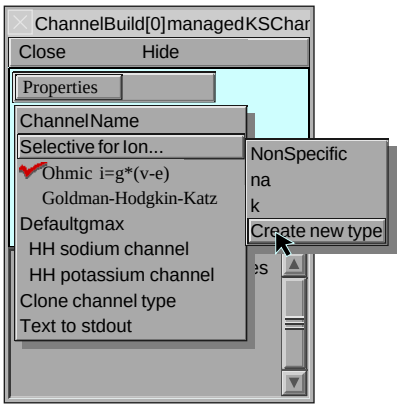


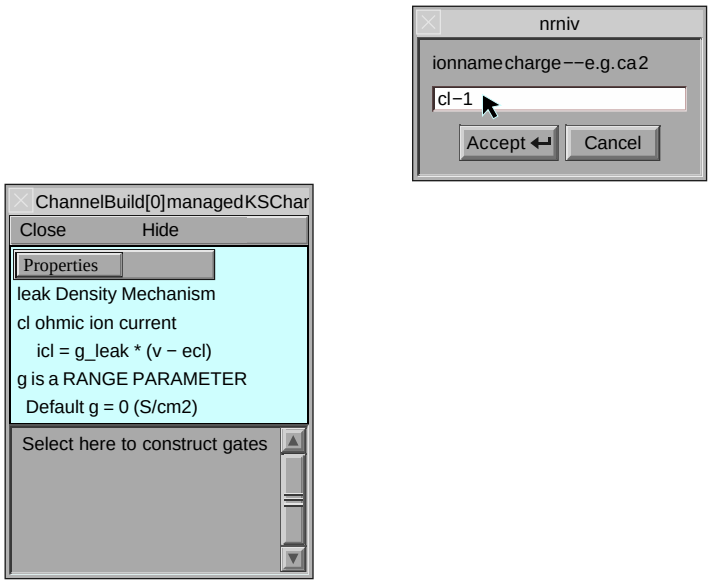












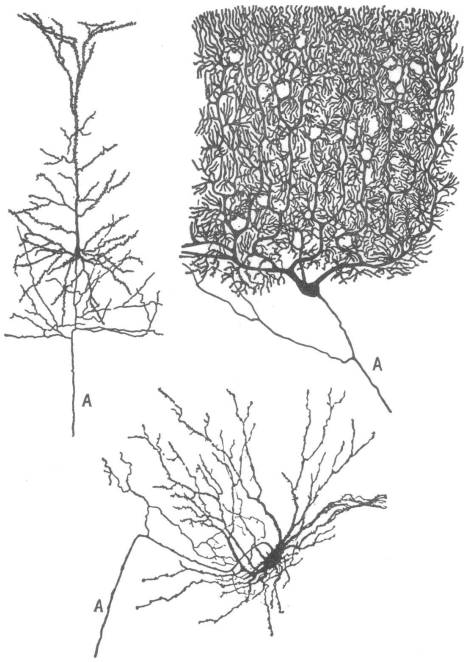
Working with morphometric data

Robert A. McDougal

Yale School of Medicine

7 August 2018

Neurons have complicated morphology



Cajal 1909, as reproduced in Rall 1962.

Neuron morphology data

Generally consists of a set of $(x, y, z; d)$ points and connectivity. Common formats: swc, asc.

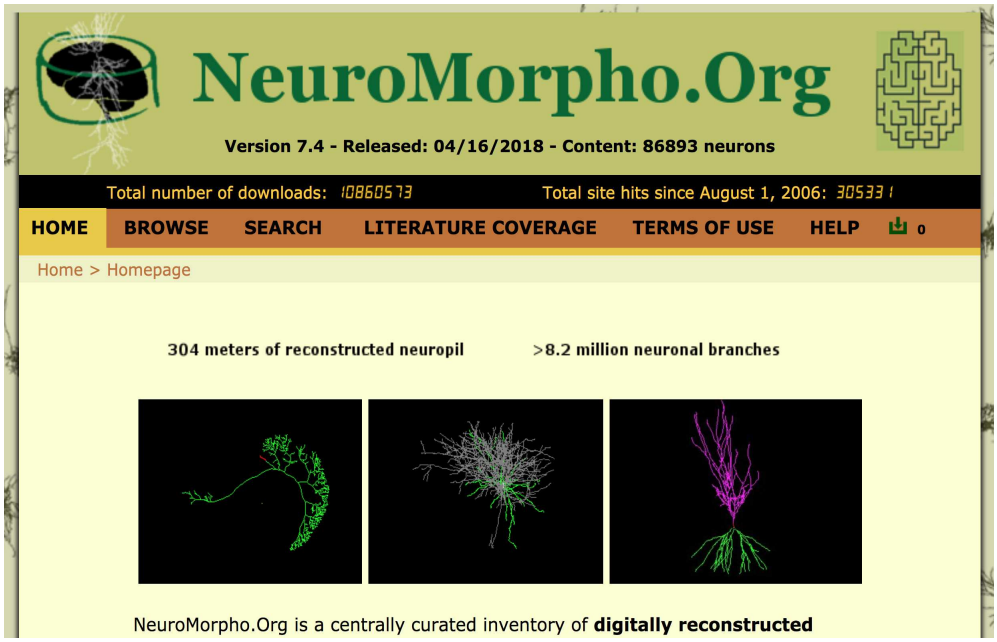
Where to get it

- Do it yourself.
- From the kindness of others.
- ModelDB (modeldb.yale.edu).
- NeuroMorpho.Org.

How to get it into NEURON

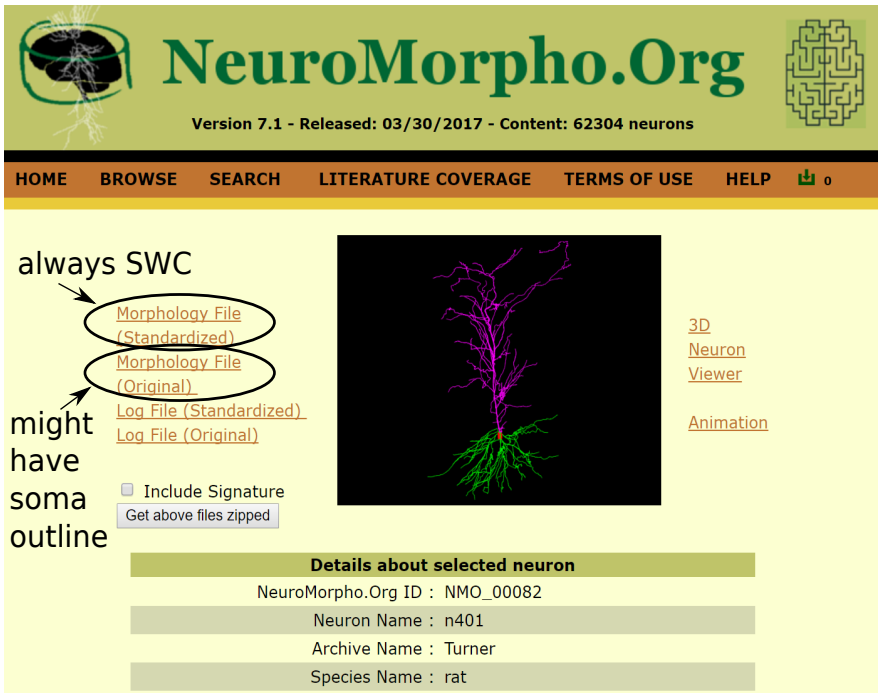
- Standalone conversion programs.
- Maybe it is already there (e.g. if from ModelDB).
- Import3D tool (GUI or programmatic).

NeuroMorpho.Org



86,893 reconstructions · 514 cell types · 278 brain regions

NeuroMorpho.Org: cell page



Examine metadata

NeuroMorpho.Org ID : NMO_00082
Neuron Name : n401
Archive Name : Turner
Species Name : rat
Strain : Fischer 344
Structural Domains : Dendrites, Soma, No Axon
Physical Integrity : Dendrites Complete
Morphological Attributes : Diameter, 3D, Angles
Min Age : 2.0 months
Max Age : 8.0 months
Gender : Male/Female
Min Weight : 200 grams
Max Weight : 350 grams
Development : young
Primary Brain Region : hippocampus
Secondary Brain Region : CA1
Tertiary Brain Region : Not reported
Primary Cell Class : principal cell
Secondary Cell Class : pyramidal
Tertiary Cell Class : Not reported
Original Format : CVAPP.swc
Experiment Protocol : in vivo
Experimental Condition : Control
Staining Method : biocytin
Slicing Direction : coronal
Slice Thickness : 80.00 µm
Tissue Shrinkage : Reported 25% in xy, 75% in z
Corrected 133% in xy, 400% in z
Objective Type : oil
Magnification : 100x
Reconstruction Method : Neurolucida
Date of Deposition : 2005-12-31
Date of Upload : 2006-08-01

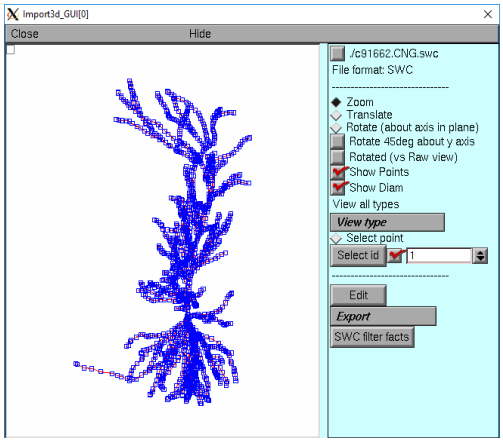
Soma Surface : 903.25 µm2
Number of Stems : 7
Number of Bifurcations : 113
Number of Branches : 233
Overall Width : 363.7 µm
Overall Height : 717.18 µm
Overall Depth : 364.21 µm
Average Diameter : 1.16 µm
Total Length : 22216.3 µm
Total Surface : 84796.1 µm2
Total Volume : 30674.3 µm3
Max Euclidean Distance : 668.56 µm
Max Path Distance : 1893.37 µm
Max Branch Order : 25
Average Contraction : 0.7
Total Fragmentation : 5460
Partition Asymmetry : 0.56
Average Rall's Ratio : 1.78
Average Bifurcation Angle Local : 89.59°
Average Bifurcation Angle Remote : 75.23°
Fractal Dimension : 1.07

THE JOURNAL OF COMPARATIVE NEUROLOGY 391:335–352 (1998)

Dendritic Properties of Hippocampal CA1 Pyramidal Neurons in the Rat: Intracellular Staining In Vivo and In Vitro

G.K. PYAPALI,[1,2] A. SIK,[3] M. PENTTONEN,[3] G. BUZSAKI,[3] AND D.A. TURNER[1,2,4*]
[1]Department of Neurosurgery, Duke University, Durham, North Carolina 27710
[2]Durham Veterans Affairs Medical Center, Durham, North Carolina 27710
[3]Center for Molecular and Behavioral Neuroscience, Rutgers, The State University of New Jersey, Newark, New Jersey 07102
[4]Department of Neurobiology, Duke University, Durham, North Carolina 27710

Import3D



Access via Tools - Miscellaneous - Import 3D.
Can instantiate directly into NEURON or transfer to CellBuilder.

For more details, see: neuron.yale.edu/neuron/docs/import3d

Loading morphologies via Python scripts is discussed in a different talk.

Potential issues

Warning: not every morphology was reconstructed with the intent of being in a simulation.

Potential factors affecting the quality of the data:

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

Before using a morphology found online, *always* read the associated paper(s) to make sure you understand any limitations of the reconstruction.

Trust but verify...

Qualitative tests.

Look for orphan sections and bottlenecks.

- Insert `pas`, set `Ra` and `g_pas = pas.g` low.
- Inject large depolarizing current at soma.
- Examine shape plot of v.

Look for z-axis drift and backlash.

- Rotate the cell on the shape plot and look for abrupt jumps.

. . . and verify some more

Quantitative tests.

Is a diameter too large or too small?

```
diam_min = 1e10
diam_max = 0
for sec in h.allsec():
    for i in range(sec.n3d()):
        diam_min = min(diam_min, sec.diam3d(i))
        diam_max = max(diam_max, sec.diam3d(i))

print('Min diam: %g' % diam_min)
print('Max diam: %g' % diam_max)
```

Can also test for systematic errors, e.g. by looking at a histogram of diameter measurements.

For more on issues with morphology

Kaspirzhny, A. V., Gogan, P., Horcholle-Bossavit, G., & Tyč-Dumont, S. (2002). Neuronal morphology data bases: morphological noise and assesment of data quality. Network: Computation in Neural Systems, 13(3), 357-380.

Scorcioni, R., Lazarewicz, M. T., & Ascoli, G. A. (2004). Quantitative morphometry of hippocampal pyramidal cells: differences between anatomical classes and reconstructing laboratories. Journal of Comparative Neurology, 473(2), 177-193.

MorphoUnit (SciUnit-based morphology testing):
https://github.com/appukuttan-shailesh/morphounit

Exercise

Download and examine the following three CA1 pyramidal cell morphologies (use the "standardized" version). What are your thoughts on the appropriateness of each for simulation?

- http://tinyurl.com/neuromorpho-n123

- http://tinyurl.com/neuromorpho-c91662

- http://tinyurl.com/neuromorpho-calsynteninKO

NMODL

The 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

Advantages

- 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 = 0.036 (S/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 integrated)

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

State equations

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

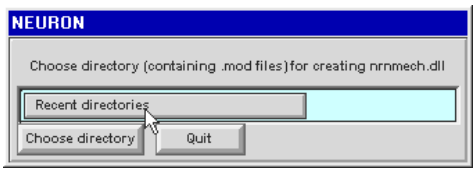
Functions and procedures

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

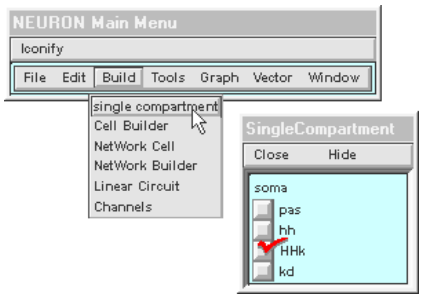

UNIX `nrnivmodl`

MSWin

mknrndll



Result: NEURON has a new mechanism



Density mechanism

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

PARAMETER {
    g = 0.001 (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 = (0.001)*(v - e)/r
}
```


Density mechanism

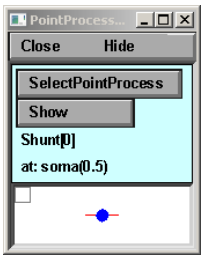
Point Process

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 = 0.0001
}
print soma.i_leak(0.5)

soma.insert('leak')
soma.g_leak = 0.0001
print soma(0.5).i_leak
```

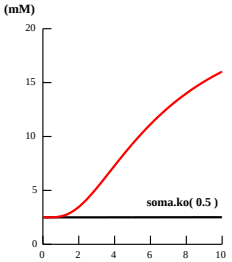
```
objref s
soma s = new Shunt(0.5)
s.r = 2

print s.i

s = h.Shunt(soma(0.5))
s.r = 2
```

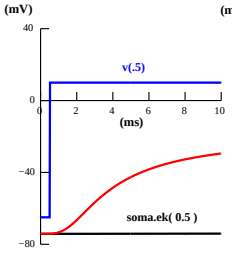
Ion Channel

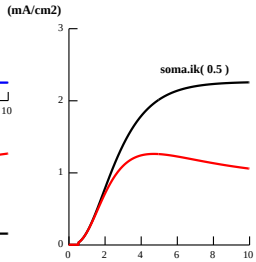
```
NEURON {
  USEION k READ ek WRITE ik
}
BREAKPOINT {
  SOLVE states METHOD cnexp
  ik = gbar*n*n*n*n*(v - ek)
}
DERIVATIVE states {
  rate(v*1(/mV))
  n' = (inf - n)/tau
}
```

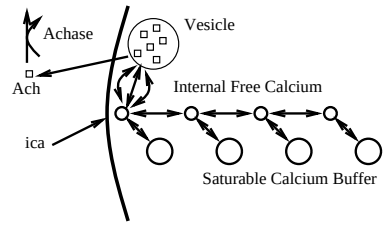
Ion Accumulation

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

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










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

UNITS Checking

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

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


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

The output from
modlunit shunt
is:

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

To fix the problem replace the line with:

```
       i = (0.001)*(v − e)/r
```

What conversion factor will make the following consistent?

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

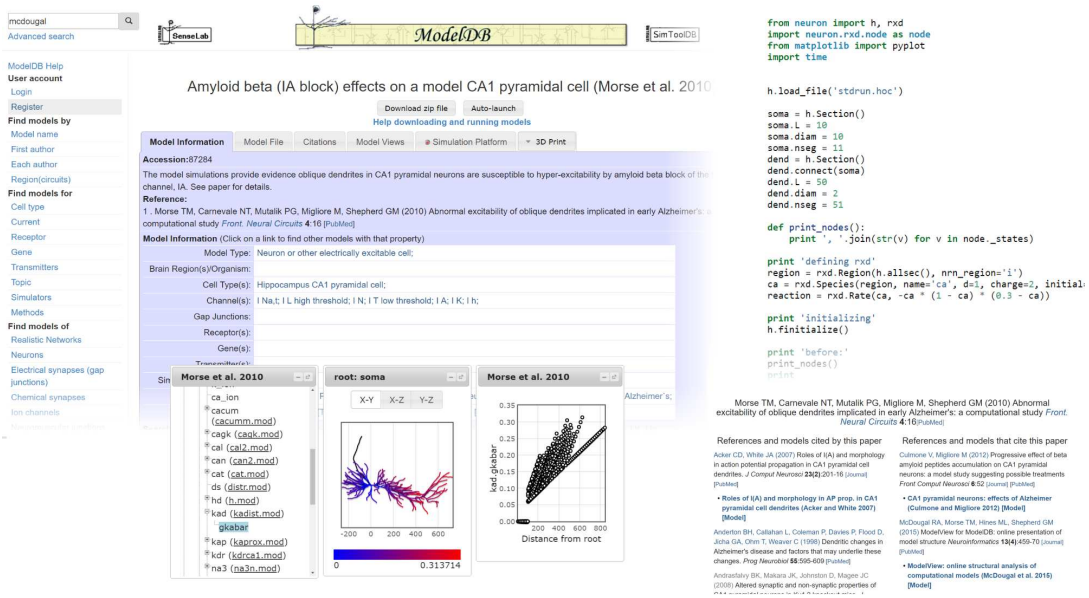
Don't reinvent the brain

Using ModelDB and other archives for your research

Robert A. McDougal

Yale School of Medicine

8 August 2018



modeldb.yale.edu

J Comput Neurosci
DOI 10.1007/s10827-016-0623-7

Twenty years of ModelDB and beyond: building essential modeling tools for the future of neuroscience

Robert A. McDougal[1] · Thomas M. Morse[1] · Ted Carnevale[1] · Luis Marenco[1,2,3] · Rixin Wang[3,4] · Michele Migliore[1,5] · Perry L. Miller[2,3,4] · Gordon M. Shepherd[1] · Michael L. Hines[1]

Received: 9 June 2016 / Revised: 17 August 2016 / Accepted: 30 August 2016

Abstract Neuron modeling may be said to have originated with the Hodgkin and Huxley action potential model in 1952 and Rall's models of integrative activity of dendrites in 1964.

groups (Allen Brain Institute, EU Human Brain Project, etc.) are emerging that collect data across multiple scales and integrate that data into many complex models, presenting new

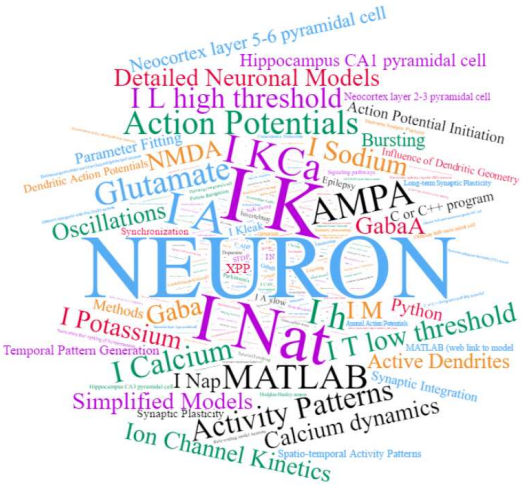
What is in ModelDB?

Models for:

- 178 cell types
- 16+ species
- 54 ion channels, pumps, etc
- 145 topics (Alzheimer's, STDP, etc)
- 24+ mammalian brain regions

1350 published models from 88 simulators

- 635 NEURON models
- 372 "realistic" networks
- 54 connectionist networks



Numbers are as of July 22, 2018

On reproducibility

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

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

What is needed for a model to be reproducible?

Model

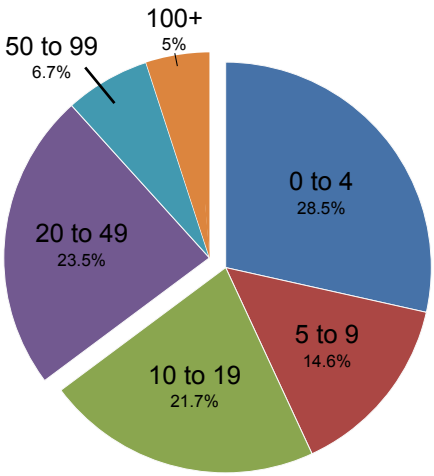
- an approximation of the system of interest
  e.g. a model organism or a complete statement of the properties of the model in mathematical or computable form

Experimental protocol

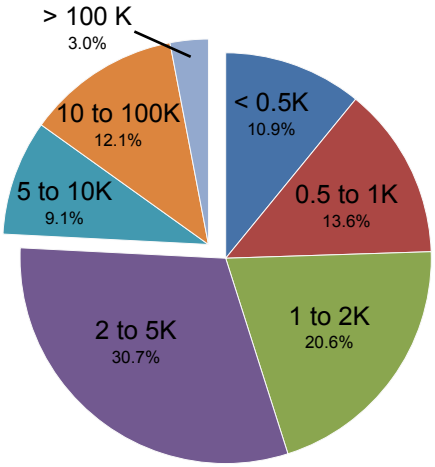
- what was done with the model to produce the data

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

Models are complicated



Files per Model



File Size

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

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

Model sharing helps, but only reuse what you understand

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

But beware:

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

To reduce the risk of problems:

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

Reproducibility in Computational Neuroscience Models and Simulations

Robert A. McDougal, Anna S. Bulanova, William W. Lytton

***Abstract*—Objective: Like all scientific research, computational neuroscience research must be reproducible. Big data science, including simulation research, cannot depend exclusively on journal articles as the method to provide the sharing and transparency required for reproducibility.**

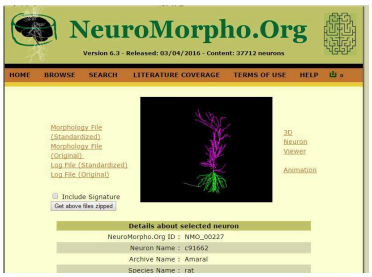
build novel theoretical frameworks. A century ago, work by Lapicque led to the development of integrate-and-fire models [4]. A half century later, Hodgkin and Huxley provided a detailed multiscale biophysical model of the squid axon [2],

- Simulators (NEURON, MCell, XPPAUT, NEST, etc)
- Multi-simulator interoperability (NeuroML, SWC, PyNN, NeuroConstruct, etc)
- Shared resources (Neuroscience Gateway, Simulation Platform)
- Sharing resources (ModelDB, OpenSourceBrain, NeuroMorpho.Org, etc)
- More: NSDF, NeuroLex, NIF, MIASE, licensing, etc

McDougal et al (2016) IEEE TBME 63(10):2021-2035; doi:10.1109/TBME.2016.2539602

Neurobiological context

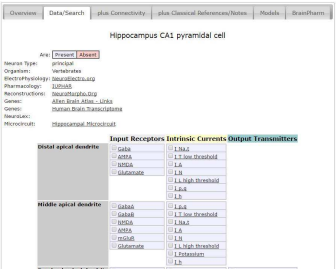
Morphology



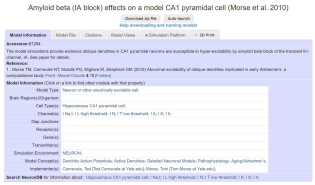
Metadata

cell types, channels, receptors, genes, transmitters, model topics, publication

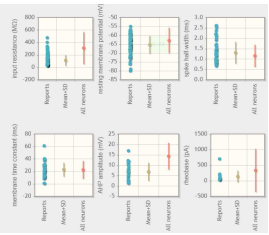
NeuronDB



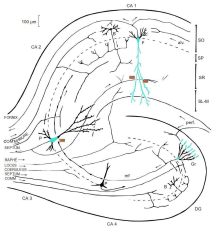
Model Entry



Electrophysiology



Microconnectome



Every model is a review of the literature.

ModelDB reveals what has been modeled in each cell type.

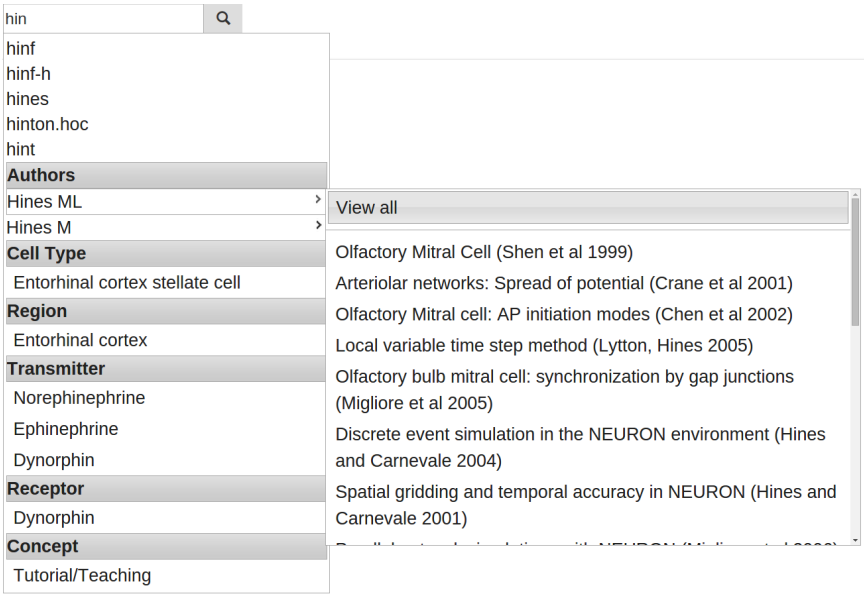
Comparing models shows what mechanisms are considered critical by the community.

Hippocampus CA1 Pyramidal Cells

I_A	47 models: 2796, 7386, 9769, 19696, 20212, 32992, 44050, 55035, . . .
$I_{K,Ca}$	11 models: 20212, 87284, 115356, 119266, 123927, 125152, . . .
I_M	16 models: 2937, 20212, 66268, 112546, 115356, 118986, 119266, . . .

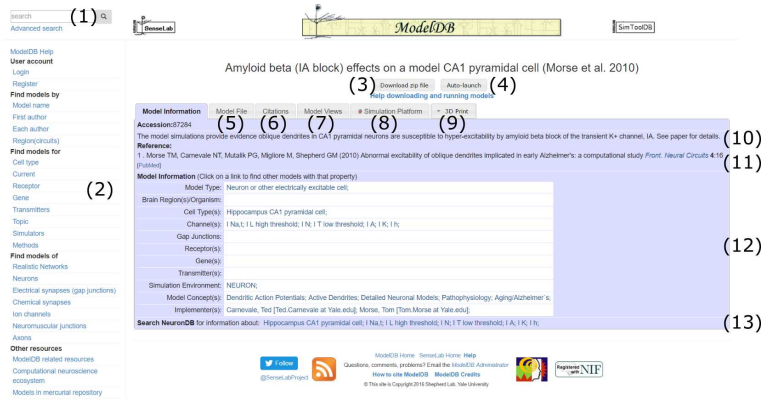
26 currents, 6 transmitters, 10 receptors

Finding models



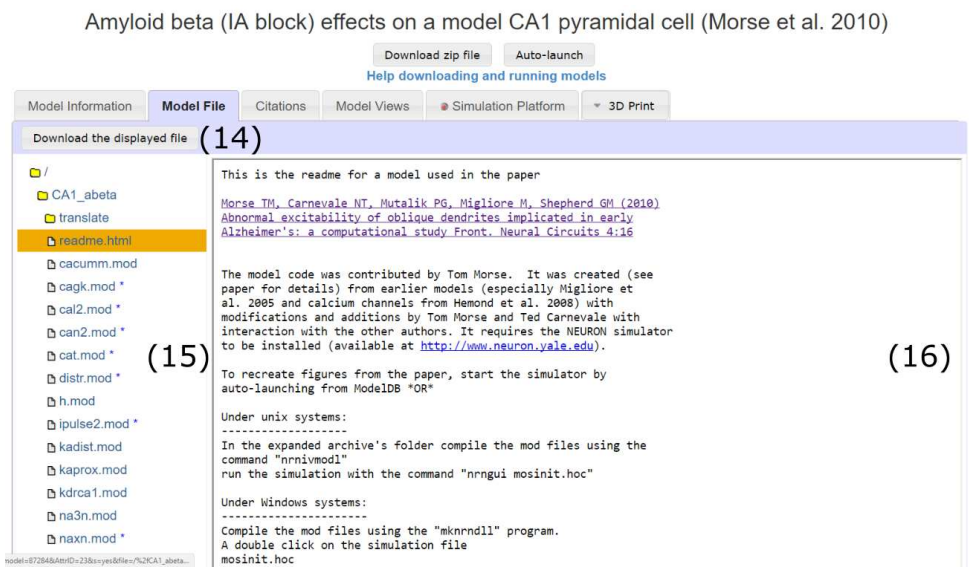
- **Search box** on the top-left of every page.
- Do **full text** or **attribute** searches.
- Word completions (based on ModelDB entries not English) and attribute results **updated as you type**.
- **Advanced search** and **browsing** are also available.

ShowModel features



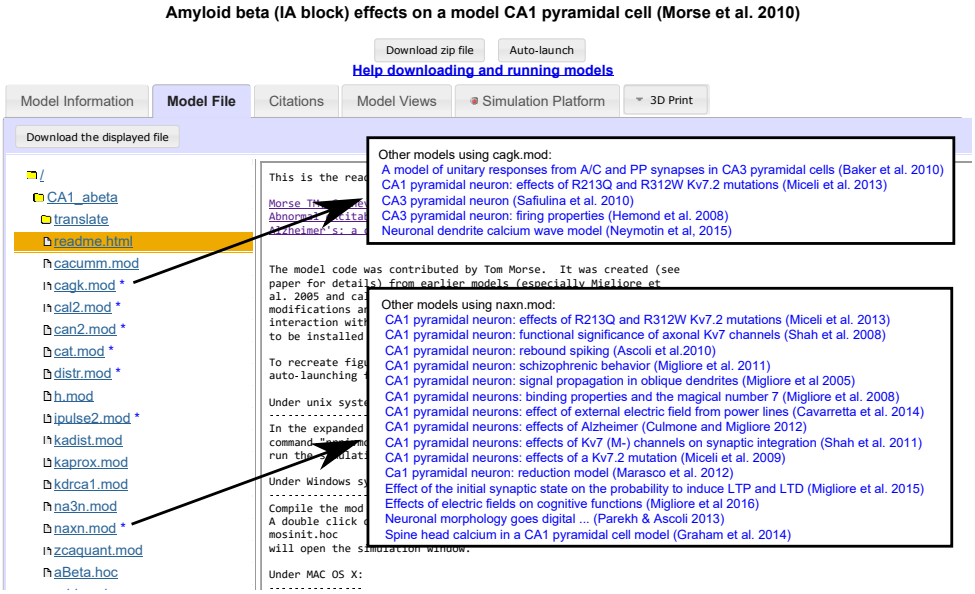
(1) Search models. **(2)** Browse models. **(3)** Link to download the entire model code. **(4)** Auto-launch a NEURON simulation (requires browser configuration). **(5)** View model files. **(6)** Find models and papers cited by this model's paper, or that cite this model. **(7)** ModelView: visualize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

ShowModel features



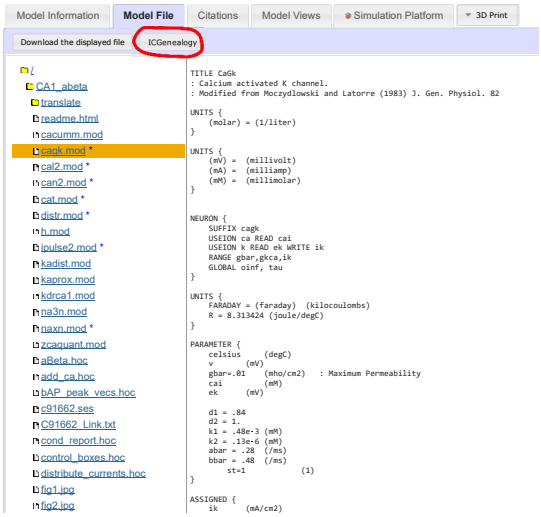
(14) Download the currently selected file. **(15)** Directory browser, showing model files. **(16)** View pane for the currently selected file.

Identifying existing reuse



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

ICGenealogy: ion channel metadata



General data

- **ICG id:** 2464
- **ModelDB id:** 87284
- **Reference:** *Morse TM, Carnevale NT, Mutalik PG, Migliore M, Shepherd GM (2010)*: Abnormal Excitability of Oblique Dendrites Implicated in Early Alzheimer's: A Computational Study.

Metadata classes

- **Animal Model:** rat
- **Brain Area:** hippocampus, CA1
- **Classes:** KCa
- **Ion Type:** K
- **Neuron Region:** unspecified
- **Neuron Type:** pyramidal cell
- **Runtime Q:** Q4 (slow)
- **Subtype:** not specified

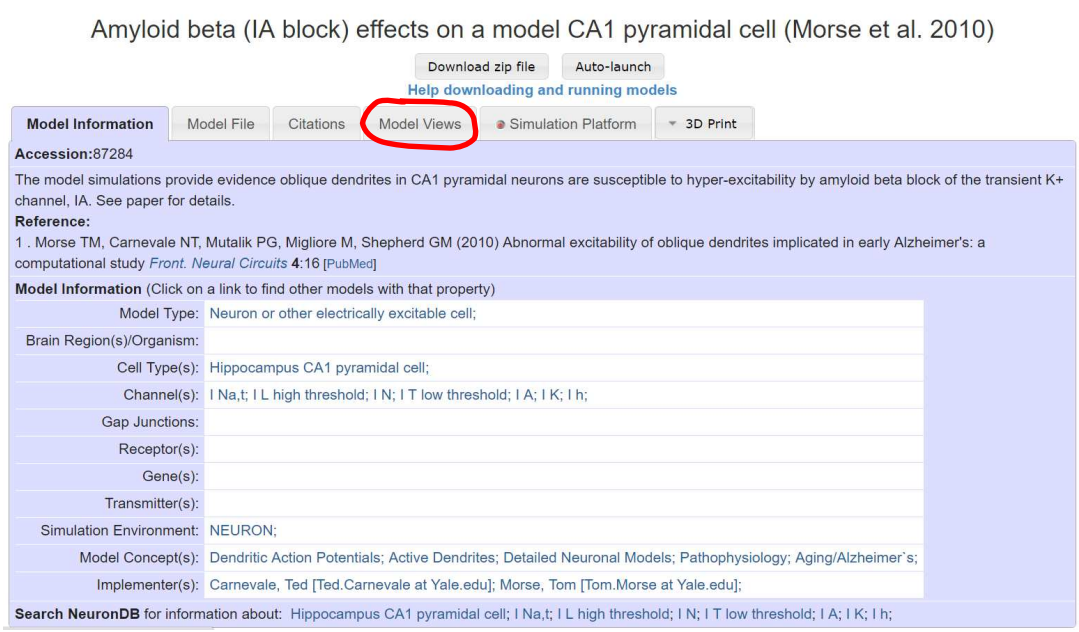
Metadata generic

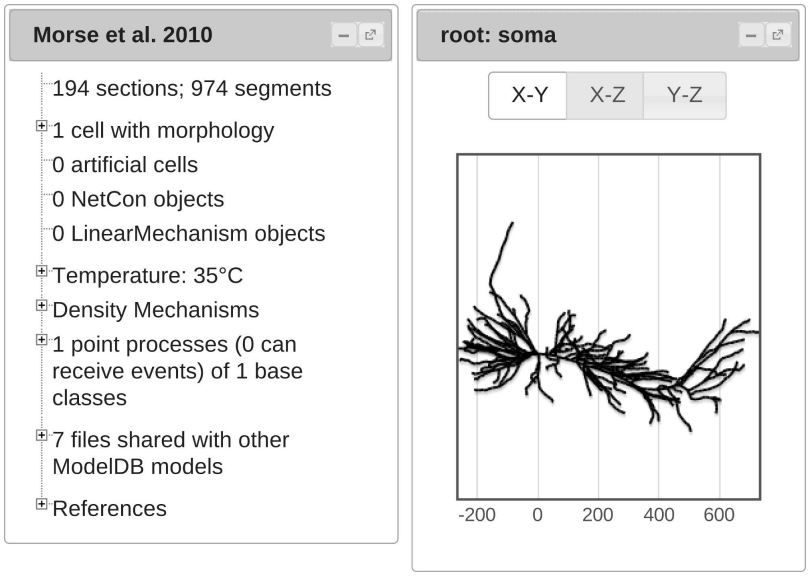
- **Age:** 7-14 weeks old.
- **Comments:** Calcium activated k channel, modified from moczydlowski and latorre (1983). From hemond et al. (2008), model no. 101629, with no changes (identical mod file). Animal model taken from chen (2005) which is used to constrain model. Channel kinetics from previous study on hippocampal pyramidal neuron (hemond et al. 2008)
- **Runtime:** 76.722

When viewing most `mod` files describing an ion channel, an ICGenealogy button appears. Clicking this button loads the corresponding page of the ICGenealogy database which shows curated information about the channel model (how it was derived, information about the underlying data, etc) and response curves.

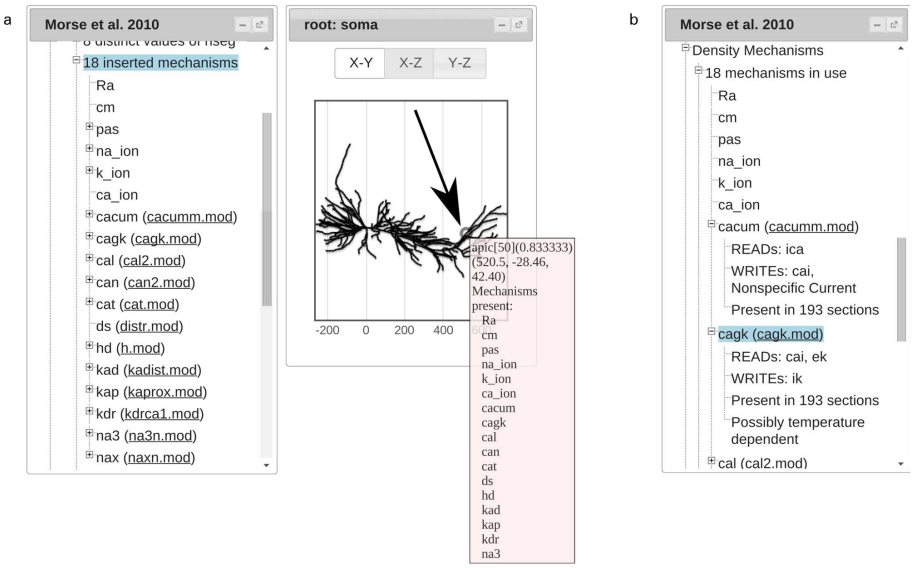
Podlaski et al., 2017. doi:10.7554/eLife.22152.001

ModelView

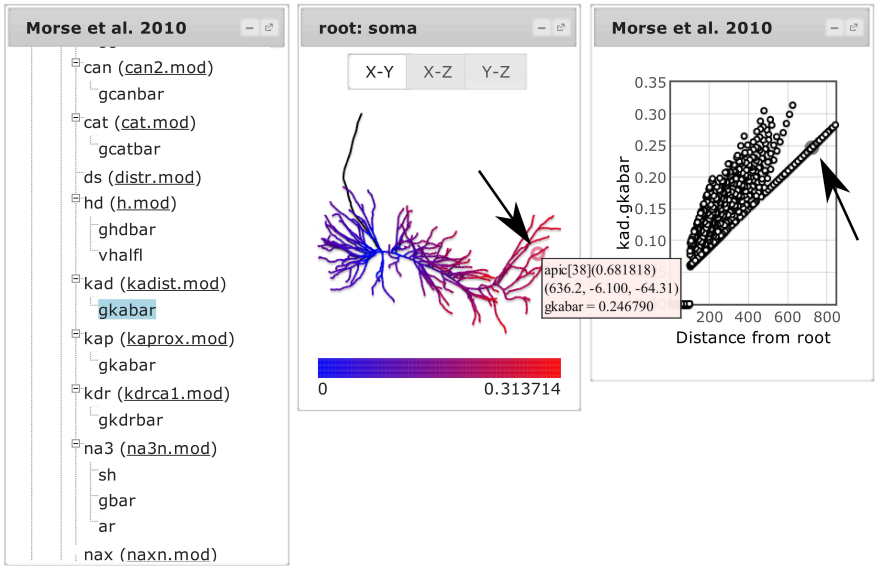




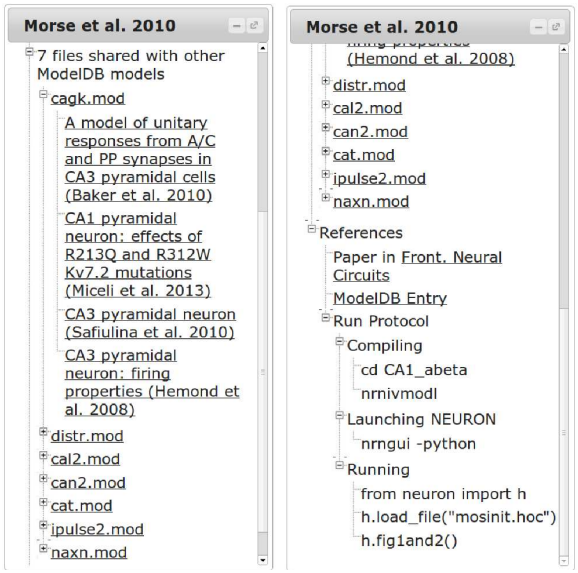
McDougal et al, *Neuroinformatics* 2015



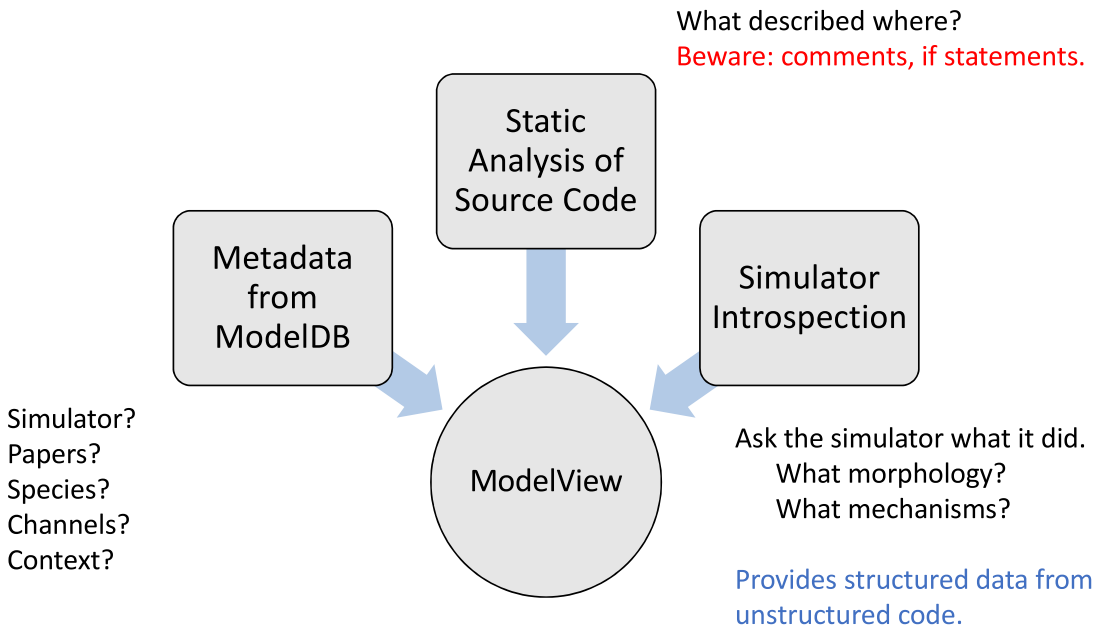
McDougal et al, *Neuroinformatics* 2015



McDougal et al, *Neuroinformatics* 2015



McDougal et al, *Neuroinformatics* 2015



How do people use ModelDB?

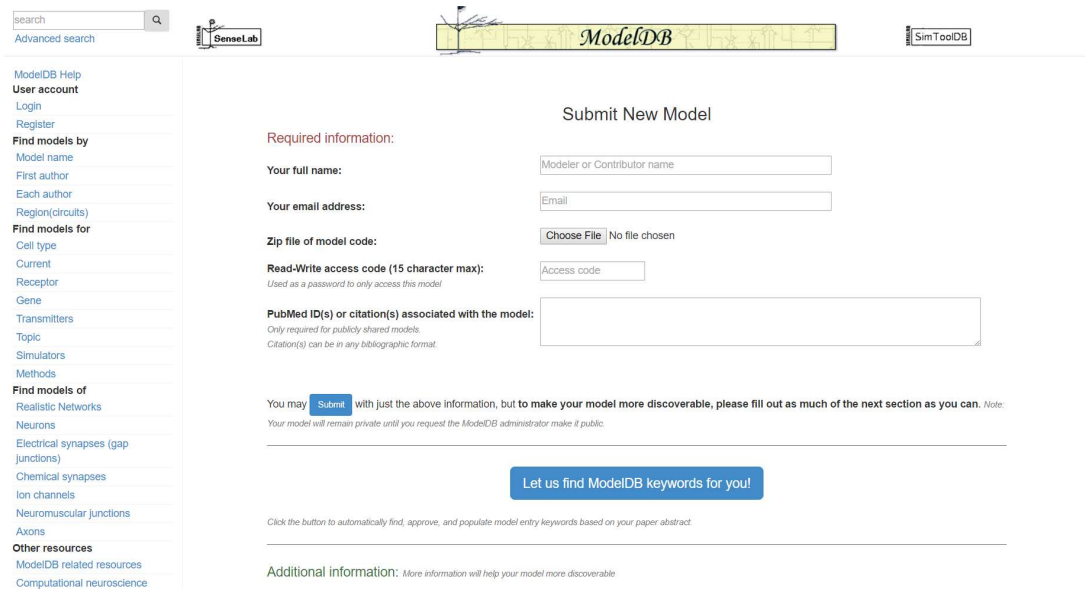
- Find a model described in a paper, download it, and experiment to understand the model's predictions.
- Find a model described in a paper. Use ModelView to understand the model's structure.
- Locate models and modeling papers on a given topic.
- Locate model components (e.g. L-type calcium channel)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

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

Sharing your models

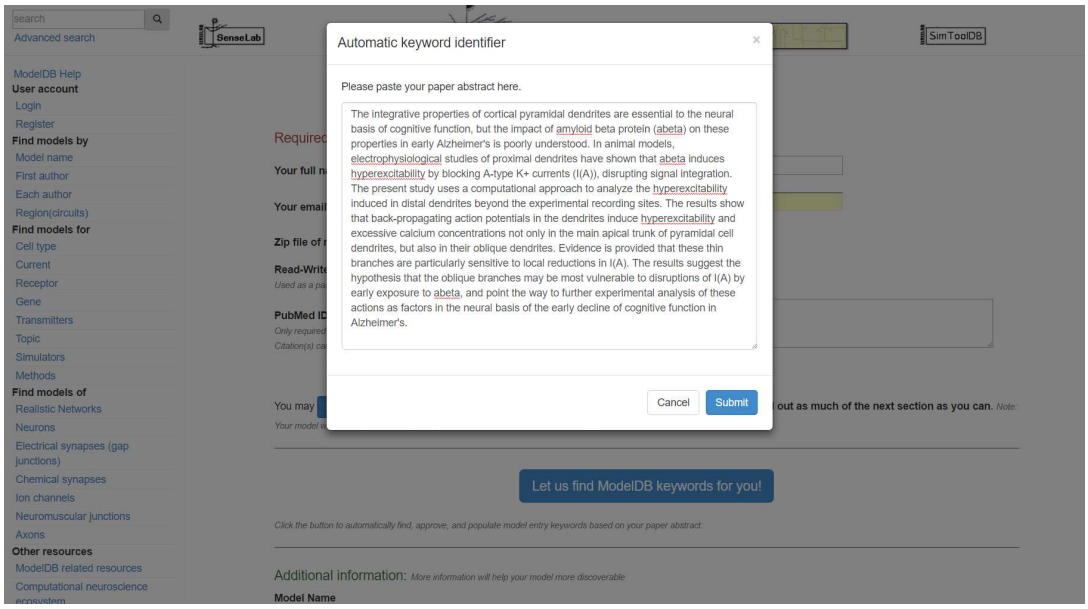
McDougal, Dalal, Morse, Shepherd submitted

Sharing your models



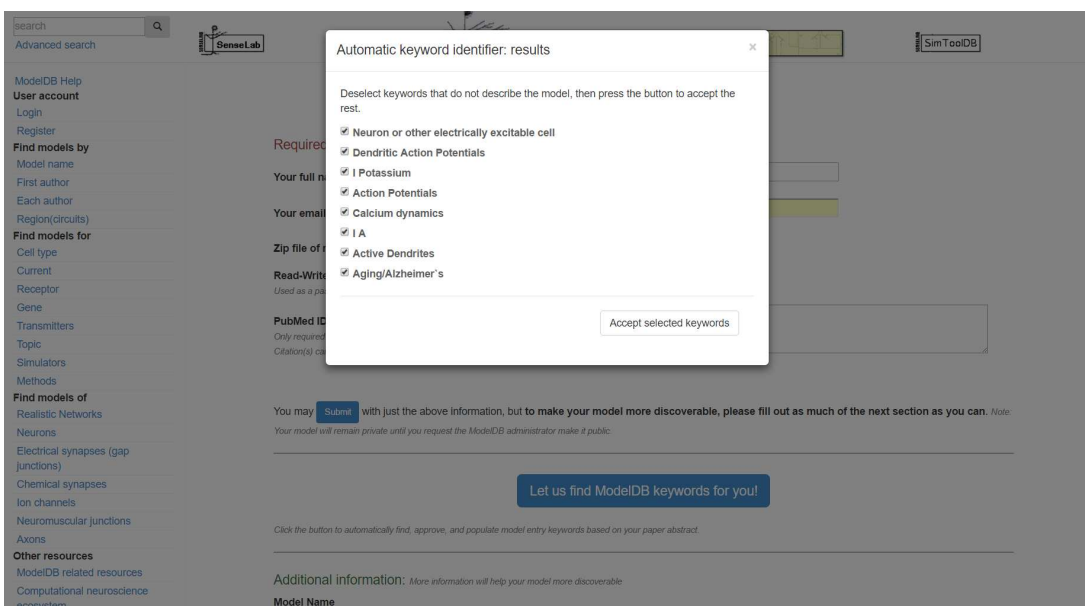
McDougal, Dalal, Morse, Shepherd submitted

Sharing your models



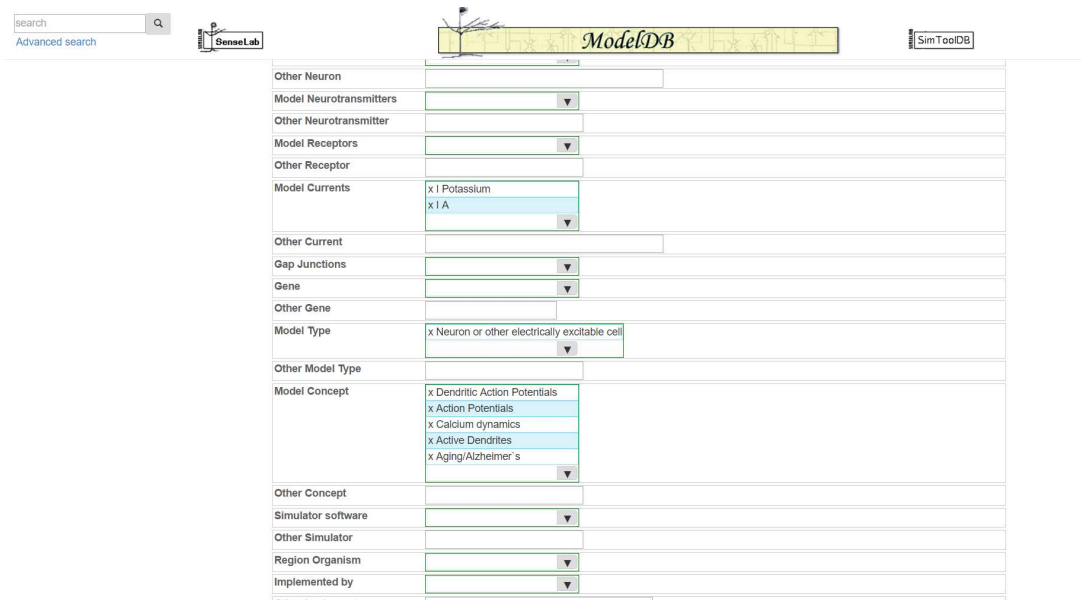
McDougal, Dalal, Morse, Shepherd submitted; abstract from Morse et al, 2010.

Sharing your models



McDougal, Dalal, Morse, Shepherd submitted

Sharing your models

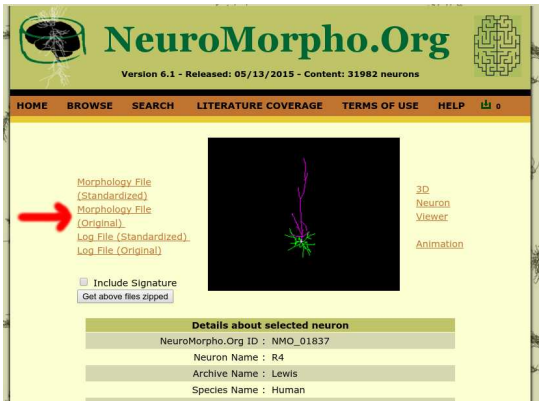


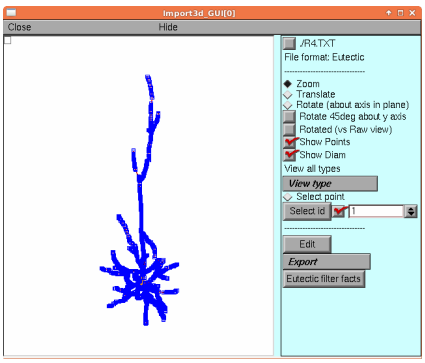
McDougal, Dalal, Morse, Shepherd submitted

@SenseLabProject: newly available models

Other resources

NeuroMorpho.Org

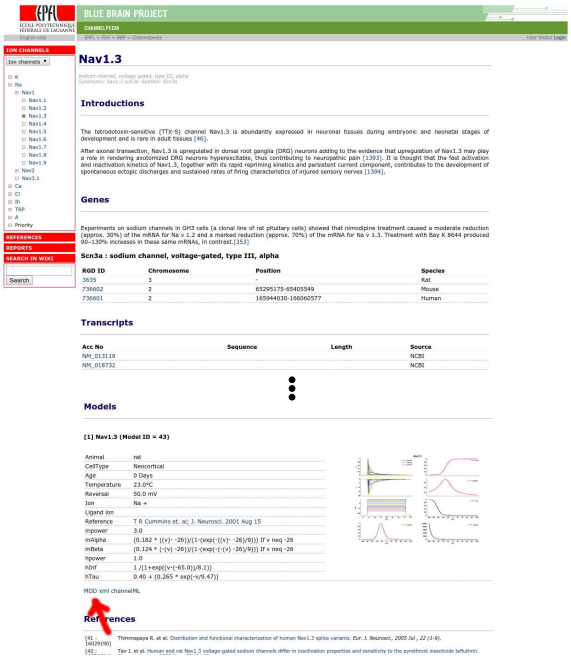




Tools ▶ Miscellaneous ▶ Import 3D

- NeuroMorpho.Org is home to 86,893 reconstructed neurons from 514 cell types and 53 species as of July 22, 2018.
- Warning: not every morphology was reconstructed with the intent of being in a simulation. Before using: rotate to check for z-axis errors, check to make sure the diameters are not all equal.
- Use the Import 3D tool to import morphologies into NEURON. For details, see: neuron.yale.edu/neuron/docs/import3d

Channelpedia (Channelpedia.epfl.ch)

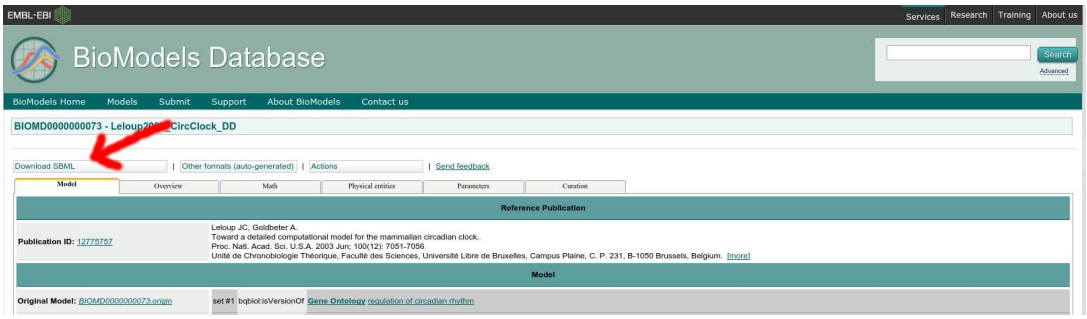


Home to information about ion channels.

Many channels have one or more associated models (e.g. different species or cell types); all are downloadable as MOD files.

Shows gating variable and channel response to voltage clamp for each model.

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



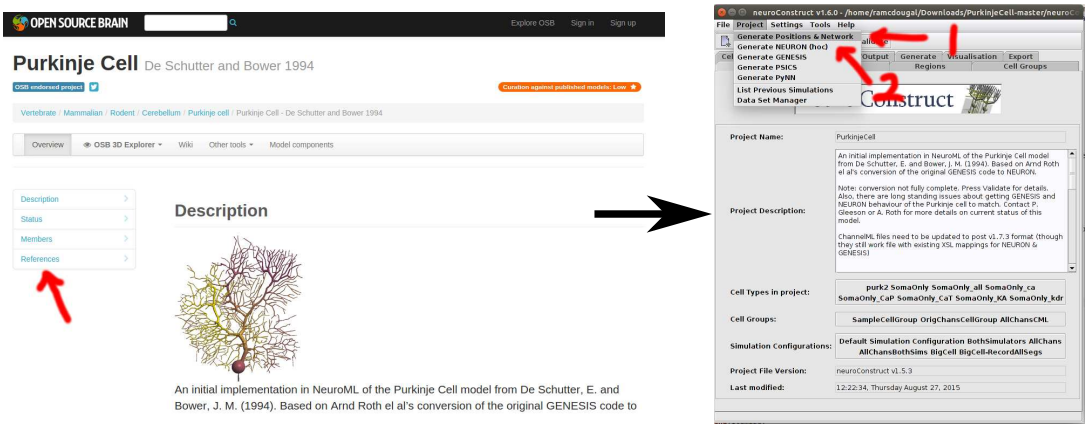

```
jnml BIOMD0000000073_LEMS.xml -neuron
```

Biomodels model (SBML) ⟶ LEMS model ⟶ MOD file

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

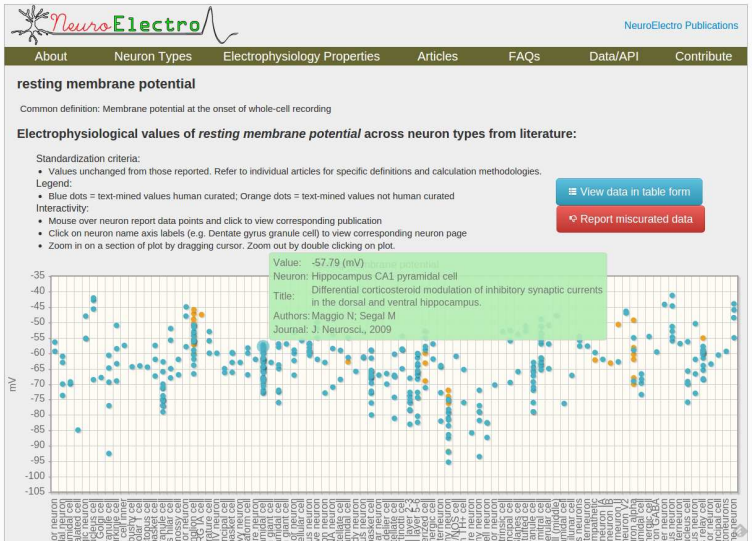
- Biomodels is a systems biology model repository.
- Models are in SBML but can be converted to MOD files via e.g. jNeuroML (github.com/NeuroML/jNeuroML). Test converted models before using in a larger model. Edits will likely be necessary to get them to interoperate with other mechanisms.
- A native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

Open Source Brain (OpenSourceBrain.org)



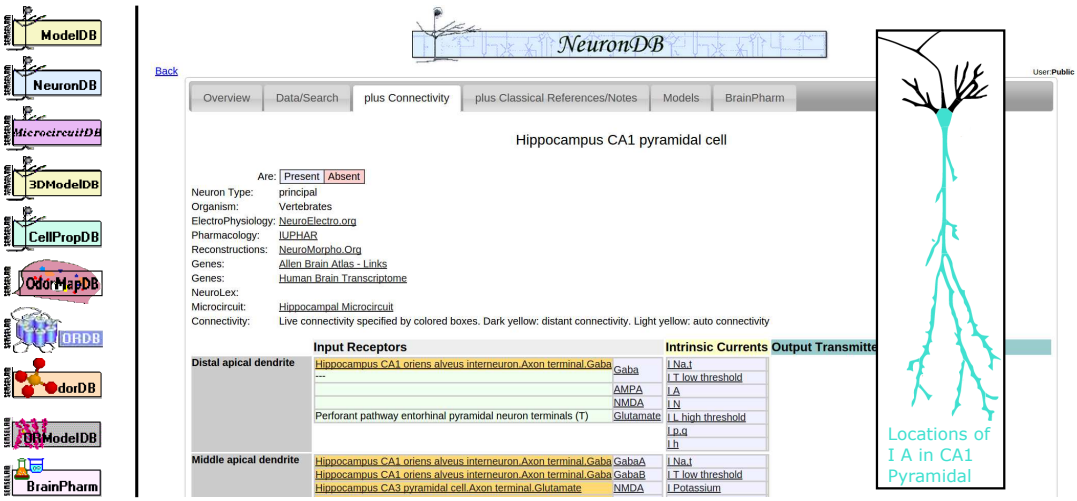
- Open Source Brain promotes collaborative model development via github.
- Models are typically in NeuroML or neuroConstruct format; neuroConstruct (neuroConstruct.org) converts both formats to NEURON.
- The conversion process places different ion channels in different MOD files, which allows extracting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

NeuroElectro (NeuroElectro.org)



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

SenseLab (senselab.med.yale.edu)



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

Stay up to date

Twitter

Many groups announce new developments on Twitter, including:

- SenseLab (including ModelDB): @SenseLabProject
- Open Source Brain: @OSBTeam
- NeuroMorpho.Org: @NeuroMorphoOrg
- ICGenealogy Project: @ICGenealogy
- Int. Neuroinformatics Coordinating Facility (INCF): @INCForg

Modeling intracellular neuronal dynamics

Robert A McDougal

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

https://en.wikipedia.org/wiki/Reaction%E2%80%93diffusion_system

Mass-Action kinetics

The model

- A reaction's product is formed at a rate proportional to the concentration of the reactants.

Example

- Consider the reaction

$$\text{Na} + \text{Cl} \xrightarrow{k} \text{NaCl}$$

- Then:

$$[\text{Na}]' = -k[\text{Na}][\text{Cl}]$$
$$[\text{Cl}]' = -k[\text{Na}][\text{Cl}]$$
$$[\text{NaCl}]' = k[\text{Na}][\text{Cl}]$$

Conservation of mass.

Matter is neither created nor destroyed by reactions.

In our equations, this means:

[Na] + [NaCl] = constant
[Cl] + [NaCl] = constant

Exercise

Use the law of mass-action to write a system of equations describing the formation of *calcium chloride*:

$$\text{Ca} + 2\,\text{Cl} \underset{k_b}{\overset{k_f}{\rightleftarrows}} \text{CaCl}_2$$

Answer:

$$[\text{Ca}]' = -k_f[\text{Ca}][\text{Cl}]^2 + k_b[\text{CaCl}_2]$$
$$[\text{Cl}]' = -2k_f[\text{Ca}][\text{Cl}]^2 + 2k_b[\text{CaCl}_2]$$
$$[\text{CaCl}_2]' = k_f[\text{Ca}][\text{Cl}]^2 - k_b[\text{CaCl}_2]$$

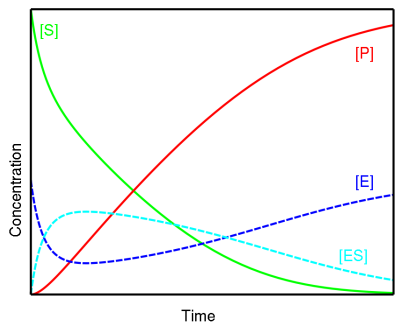
Enzyme kinetics

It is generally **not** the case that a substrate transforms directly into a product:

$$S \rightarrow P$$

Instead, an enzyme is often involved:

$$\text{E + S} \underset{k_b}{\overset{k_f}{\rightleftarrows}} \text{ES} \xrightarrow{k_{\text{cat}}} \text{E + P}$$



https://commons.wikimedia.org/wiki/File:Michaelis_Menten_S_P_E_ES.svg

Michaelis-Menten

If we can assume either:

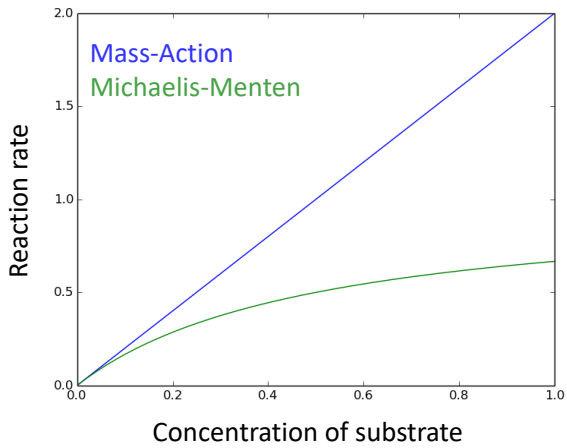
- the substrate (S) and the complex (ES) are in instantaneous equilibrium, or
- the concentration of the complex (ES) does not change on the time-scale of product formation

Then the rate of the enzymatic reaction reduces to:

$$\frac{V_{max}\,[S]}{K_M + [S]}$$

K_M is called the *Michaelis constant*. It is the concentration at which the reaction proceeds at half its maximum rate.

Michaelis-Menten vs Mass-Action



$$S \rightarrow P$$

Both curves on the left have the same rate of reaction when the substrate concentration is low, but the Michaelis-Menten rate levels off (due to limited enzyme availability) as concentrations increase.

$$y = 2x$$

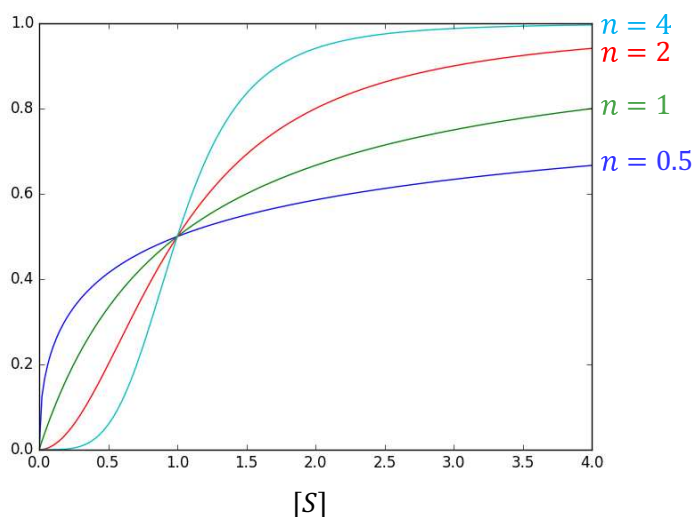
$$y = \frac{x}{x + 0.5}$$

Hill equation: cooperative binding

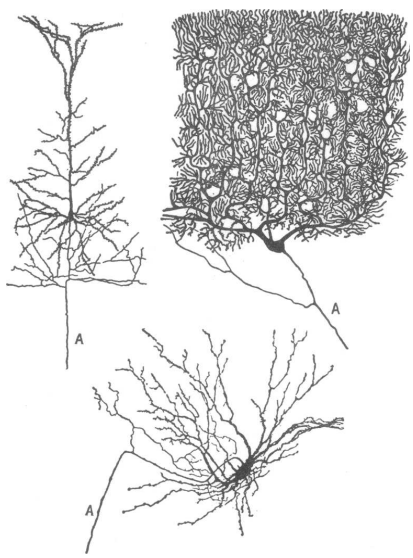
$$\frac{V_{max}\,[S]^n}{[k_A]^n + [S]^n}$$

If n > 1, positive cooperativity.
If n < 1, negative cooperativity.

$$\frac{[S]^n}{1^n + [S]^n}$$



Neurons have spatial extent



Cajal 1909 as reproduced in Rall 1962.

Effects of non-point-ness:

- Ion and protein concentrations vary with space.
- Cellular mechanisms (ER, ion channels, etc) vary with space.

Concentrations at different locations affect each other:

- Transport
- Diffusion

Fick's First Law and the diffusion equation

Fick's First Law:

- Diffusive flux is proportional to the concentration gradient.

$$J = -D\nabla\varphi$$

- Here D is called the *diffusion coefficient*.

Fick's Second Law (the diffusion equation):

$$\frac{\partial\varphi}{\partial t} = \nabla \cdot (D\nabla\varphi) = D\,\nabla^2\varphi$$

where the last equality only holds if D is constant.

Where does diffusion occur?

- Cytosol
  - But not full cross section because of organelles
- Organelles (e.g. ER)
- Extracellular space
  - Tortuosity
  - Anisotropy
  - Volume fraction

Practical limits of pure diffusion

The expected time $\mathrm{E}[t]$ for a molecule with diffusion constant D to diffuse a distance x is:

$$\mathrm{E}[t] = \frac{x^2}{2D}$$

So in particular, if

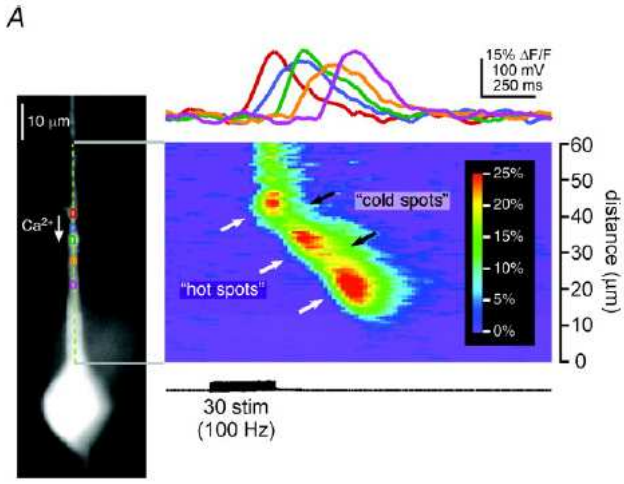
$D = 1$ μm²/ms and

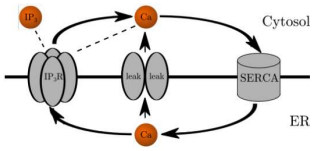
$x = 100$ μm,

Then

$E[t] = \frac{100^2}{2} = 5000$ ms.

Diffusion with regenerative dynamics can quickly spread signals





Adapted from Neymotin*, McDougal* et al 2015.

Fitzpatrick, J. S., Hagenston, A. M., Hertle, D. N., Gipson, K. E., Bertetto-D'Angelo, L., & Yeckel, M. F. (2009). Inositol-1, 4, 5-trisphosphate eceptor-mediated Ca2+waves in pyramidal neuron dendrites propagate through hot spots and cold spots. *The Journal of physiology*, *587*(7), 1439-1459.

Why use NEURON's `rxd` module?

Reduces typing

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

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

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

Allows arbitrary domains

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

Or use crxd for faster simulation.

`rxd` module overview

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

Interface design principle

Reaction-diffusion model specification is independent of:

- Deterministic vs stochastic.
- 1D or 3D.

Declare a region: rxd.Region

Basic Usage

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

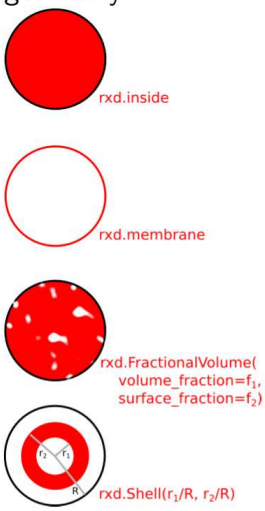
Identify with a standard region

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

Specify the cross-sectional shape

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

geometry:



Adapted from:
McDougal et al 2013.

rxd.Region tips

Specify nrn_region if concentrations interact with NMODL

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

To declare a region that exists on all sections

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

Use list comprehensions to select sections

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

Declare ions & proteins: rxd.Species

Basic usage

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

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

Initial conditions

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

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

Connecting with HOC

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

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

protein.initial can be read and set, to allow exploration of the role of initial conditions

Tip:

Variable step integration

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

Since NEURON measures concentration in mM and some cell biology concentrations (e.g. calcium) are in µM, this tolerance may be too high. Compensate by using an `atolscale` in the constructor, e.g.

```
ca = rxd.Species(cyt, atolscale=1e-6)
```

Example:

Handling non-uniform initialization

Initial value as a function of distance from a point:

```
def my_initial(node):
    # compute the distance
    distance = h.distance(soma(0.5), node.segment)
    # return a certain function of the distance
    return 2 * h.tanh(distance / 1000.)
cyt = rxd.Region(h.allsec(), name='cyt', nrn_region='i')

ip3 = rxd.Species(cyt, name='ip3', charge=2
                  initial=my_initial)
```


Example:

Handling non-uniform initialization

Initial value as a function of spatial position:

```
def my_initial(node):
    # return a certain function of the x-coordinate
    return 1 + h.tanh(node.x3d / 100.)
cyt = rxd.Region(h.allsec(), name='cyt', nrn_region='i')

ip3 = rxd.Species(cyt, name='ip3', charge=2
                  initial=my_initial)
```

Tip:

rxd.Parameter

- Used to represent things that vary spatially or across different simulations:
  - `α = rxd.Parameter(cyt, name='α', value=0.3)`
- Used to limit reactions to specific segments:
  - `soma_only = rxd.Parameter(cyt, name='paramA', value=lambda nd: 1 if nd.segment in soma else 0)`
- Used as constant terms in Reactions:
  - `k = rxd.Species([cyt, mem], name='k', d=1, charge=1, initial=54.4)`
  - `kecs = rxd.Parameter(ecs, name='k', charge=1, value=2.5)`
  - `ki, ko = k[cyt], kecs[ecs]`
  - `k_current = rxd.MultiCompartmentReaction(ki, ko, gk*(rxd.v - ek), mass_action=False, membrane=mem,membrane_flux=True)`

bit.ly/2wyG91y

Tip:

`rxd.Parameter`

- Use short-hand to avoid repeatedly writing `rxd.Parameter` boilerplate; e.g.

```
def declare_parameters(r, **kwargs):
    '''enables clean declaration of parameters in top namespace'''
    for key, value in kwargs.items():
        globals()[key] = rxd.Parameter(r, name=key, initial=value)
```

- Can then, e.g.:

```
from neuron.units import nM, hour
declare_parameters(
   vsP=1.1 * nM / hour,
   vmP=1.0 * nM / hour,
   KmP=0.2 * nM,
   KIP=1.0 * nM,
   ksP=0.9 / hour)
```

Specifying dynamics: `rxd.Reaction`

Mass-action kinetics

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

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

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

Repeated reactants

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

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

Arbitrary reaction formula, e.g. Hill dynamics

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

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

Hill dynamics are often used to model cooperative reactions.

rxd.Rate and rxd.MultiCompartmentReaction

rxd.Rate

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

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

rxd.MultiCompartmentReaction

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

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

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

Manipulating nodes

Getting a list of nodes

- nodelist = protein.nodes

Filtering a list of nodes

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

Other operations

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

Concentration pointers

To get a pointer to a concentration, use `node._ref_concentration`:

Recording traces

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

Plotting

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

Remember, you can use e.g. `dir(ca.nodes)` to find out what methods exist.

If there is only one node, you can omit the [0] before the `._ref_concentration`.

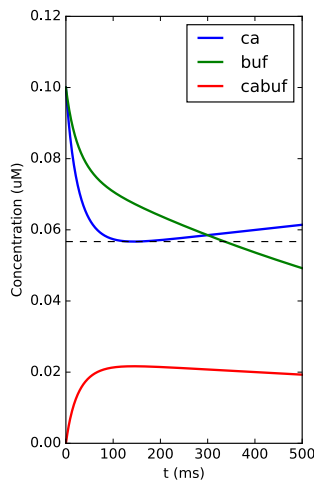
Example:

Calcium buffering*

Consider calcium buffering with a degradable buffer:

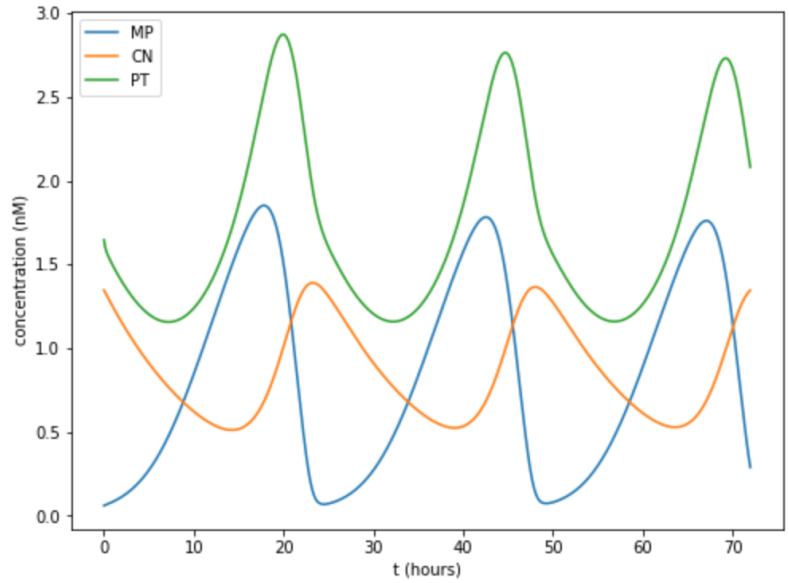
$$2\ \mathrm{Ca} + \mathrm{Buf} \leftrightarrow \mathrm{CaBuf}, \qquad \mathrm{Buf} \rightarrow (\text{degraded})$$

```
from neuron import h, rxd

# where
soma = h.Section(name='soma')
cyt = rxd.Region([soma], nrn_region='i')

# who
ca = rxd.Species(cyt, name='ca', charge=2, initial=1e-4)
buf = rxd.Species(cyt, name='buf', initial=1e-4)
cabuf = rxd.Species(cyt, name='cabuf', initial=0)

# what
buffering = rxd.Reaction(2 * ca + buf, cabuf, 1e6, 1e-2)
degradation = rxd.Rate(buf, -1e-3 * buf)
```



Example:

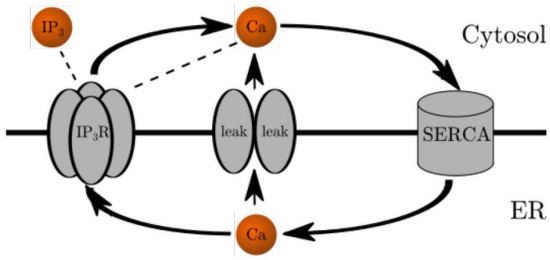
Leloup, Gonze, Goldbeter 1999



See: https://neuron.yale.edu/neuron/docs/example-circadian-rhythm

Example:

CICR



$$\frac{\partial \mathrm{Ca}^{2+}_{\mathrm{cyt}}}{\partial t} = d_{\mathrm{Ca}^{2+}_{\mathrm{cyt}}} \cdot \Delta \mathrm{Ca}^{2+}_{\mathrm{cyt}} + \frac{J_{\mathrm{IP3R}} - J_{\mathrm{SERCA}} + J_{\mathrm{leakER}}}{f_{cyt}} + c_{\mathrm{ionic}},$$

$$\frac{\partial \mathrm{Ca}^{2+}_{\mathrm{ER}}}{\partial t} = d_{\mathrm{Ca}^{2+}_{\mathrm{ER}}} \cdot \Delta \mathrm{Ca}^{2+}_{\mathrm{ER}} - \frac{J_{\mathrm{IP3R}} - J_{\mathrm{SERCA}} + J_{\mathrm{leakER}}}{f_{ER}},$$

$$\frac{\partial \mathrm{IP}_3}{\partial t} = d_{\mathrm{IP}_3} \cdot \Delta \mathrm{IP}_3,$$

$$J_{\mathrm{IP3R}} = \bar{p}_{\mathrm{IP}_3\mathrm{R}} \cdot m^3_{\mathrm{IP}_3\mathrm{R}} \cdot n^3_{\mathrm{IP}_3\mathrm{R}} \cdot h^3_{\mathrm{IP}_3\mathrm{R}} \cdot (\mathrm{Ca}^{2+}_{\mathrm{ER}} - \mathrm{Ca}^{2+}_{\mathrm{cyt}})/\Xi.$$

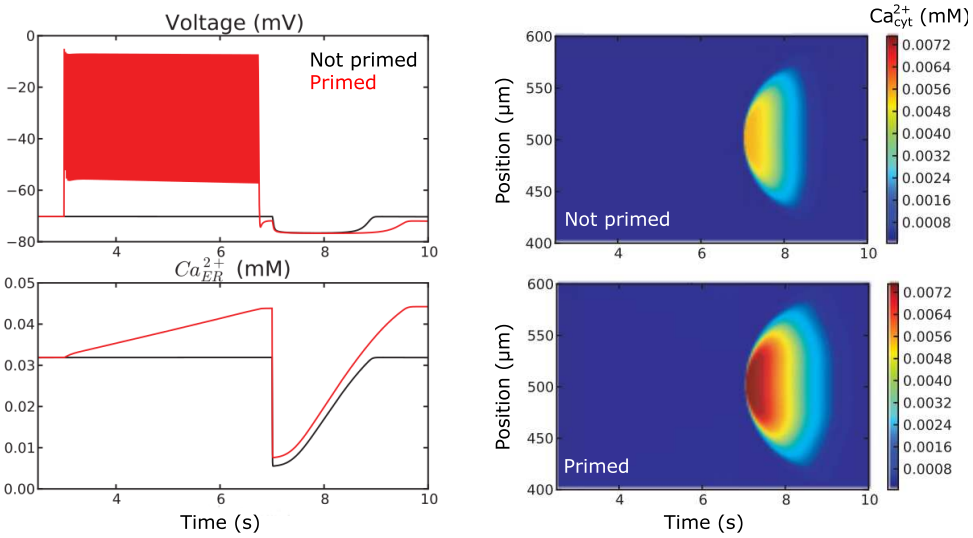
$$J_{\mathrm{SERCA}} = -\frac{\bar{p}_{serca} \cdot \mathrm{Ca}^{2+2}_{\mathrm{cyt}}}{(k^2_{serca} + \mathrm{Ca}^{2+2}_{\mathrm{cyt}}) \cdot \Xi},$$

$$J_{\mathrm{leakER}} = \bar{p}_{leakER} \cdot (\mathrm{Ca}^{2+}_{\mathrm{ER}} - \mathrm{Ca}^{2+}_{\mathrm{cyt}})/\Xi.$$

Neymotin*, McDougal*, et al. (2015)

Example:

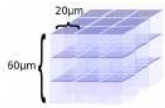
CICR



Neymotin*, McDougal*, et al. (2015). Figure 11.

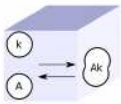
Extracellular diffusion

- Uses the same simple Python interface




```
ecs = rxd.Extracellular(xlo=-30, ylo=-30, zlo=-30,
                        xhi=30, yhi=30, zhi=30,
                        dx=20, tortuosity=1.6,
                        volume_fraction=0.2)
```

```
astrocytic_buffering = rxd.Reaction(A + k, AK, kf, kb)
```



- Rectangular cuboid grid

- Supports
  - anisotropy & heterogeneous tissue characteristics

```
k = rxd.Species(ecs, name='k', d=2.62, charge=1,
    initial=lambda nd: 40 if nd.x3d**2 + nd.y3d**2 + nd.z3d**2 < 100**2 else 3.5,
    ecs_boundary_conditions=3.5
```

neuron.yale.edu/neuron/docs/extracellular-diffusion

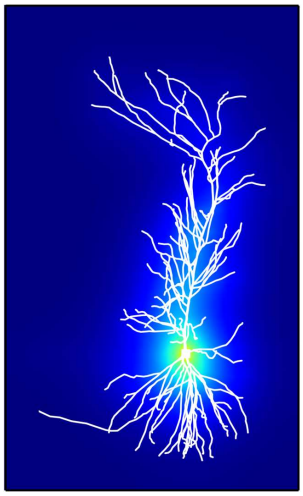
Extracellular diffusion

- Accessing and recording concentrations

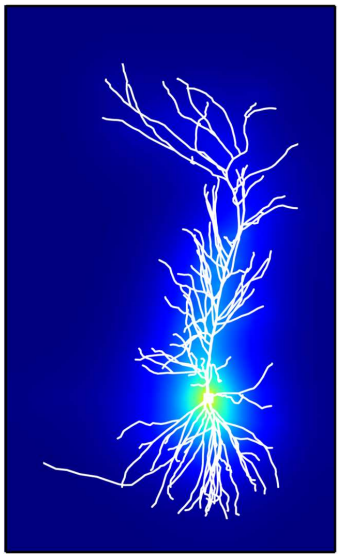
```
k[ecs].states3d    # numpy array of the extracellular states
k_vec = h.Vector().record(k[ecs].node_by_location(0,0,0)._ref_value)
```

- Inhomogeneous diffusion characteristics

```
Lx, Ly, Lz = 1000, 1000, 1000
alpha0, alpha1 = 0.07, 0.2
tort0, tort1 = 1.8, 1.6
r0 = 100

def alpha(x, y, z) :
  return alpha0 if x**2 + y**2 + z**2 < r0**2
  else min(alpha1, alpha0 +(alpha1-alpha0)
   *((x**2+y**2+z**2)**0.5-r0)/(Lx/2))

def tort(x, y, z) :
  return tort0 if x**2 + y**2 + z**2 < r0**2
  else max(tort1, tort0 - (tort0-tort1)
   *((x**2+y**2+z**2)**0.5-r0)/(Lx/2))

ecs = rxd.Extracellular(-Lx/2.0, -Ly/2.0,
  -Lz/2.0, Lx/2.0, Ly/2.0, Lz/2.0, dx=10,
  volume_fraction=alpha, tortuosity=tort)
```



Extracellular diffusion

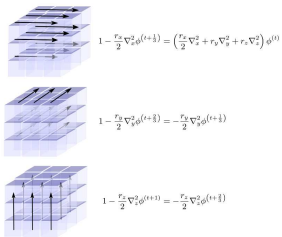


New region type:

```
ecs = rxd.Extracellular(xlo, ylo, zlo, xhi, yhi, zhi,
                        dx=dx, tortuosity=1, volume_fraction=1)
```

Setting/getting extracellular concentrations:

```
ca[ecs].states3d[5:15, 5:15, :] = 1
pyplot.imshow(ca[ecs].states3d[:, :, 0],
              interpolation='nearest', vmin=0, vmax=1,
              extent=ca[ecs].extent('xy'), origin='lower')
```



We use a finite-volume method, the Douglas-Gunn Alternating Direction Implicit algorithm is **unconditionally stable**.

Each time-step is divided into an x-, y- and z-direction and requires solving diagonally dominant tridiagonal systems of equations. This is solved with the **Thomas algorithm**, so the runtime scales linearly with the number of voxels.

We currently support zero-flux Neumann boundary conditions which conserves the total concentration.

Specifying 3D Simulations

Just add one line of code[2]:

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

Plotting

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

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

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

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

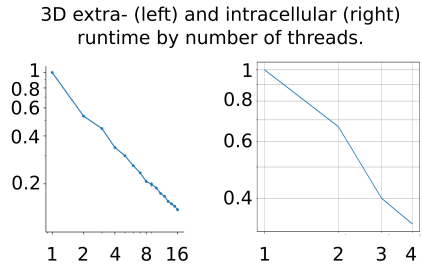
[2] `rxd.set_solve_type` can optionally take a list of sections as its first argument; in that case only the specified sections will be simulated in three dimensions.

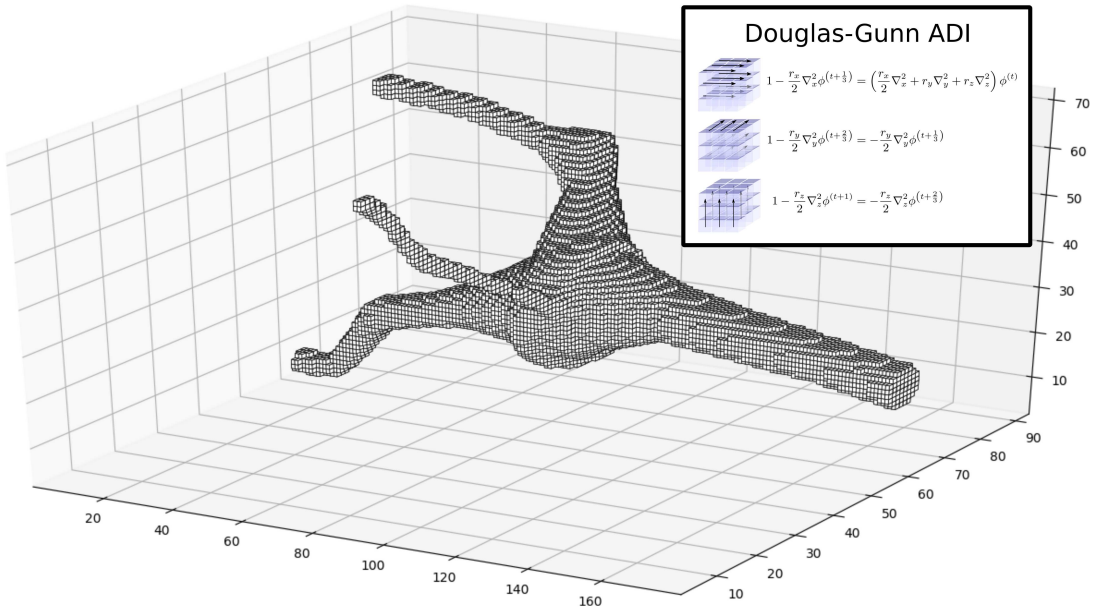
Threading

- Extracellular and 3D simulations may be threaded using, e.g.

```
rxd.nthread(4) # for four threads
```

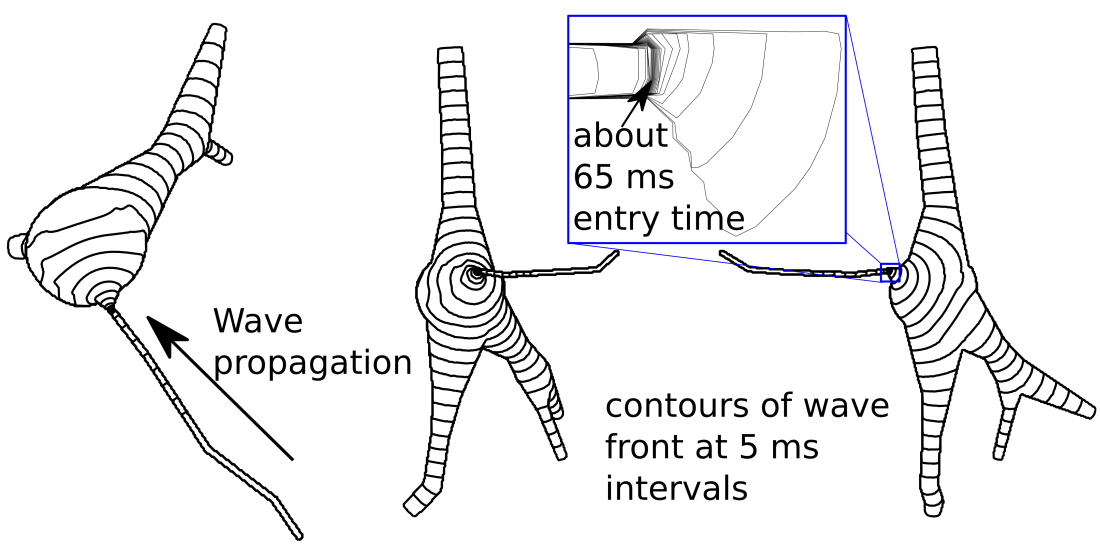
- Either electrophysiology or reaction-diffusion can be threaded, but not both.





NEURON 7.7 uses threaded DG-ADI; previous versions used bicgstab.

Wave curvature and delays at soma



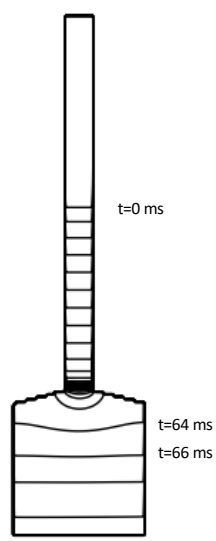
$$u_t = \mathrm{D}\,\nabla^2 u - \xi u(1-u)(\alpha - u)$$

For about 750k voxels, a pre-alpha branch of NEURON 7.7 simulated 300 ms of this (dt=0.025ms) with four threads in 258 s.

Full 3D Simulation

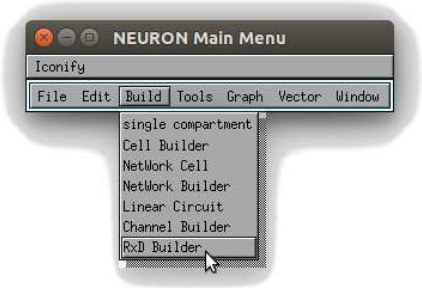
```
from neuron import h, rxd

h.load_file('stdrun.hoc')

#Create a soma
soma = h.Section(name='soma')
soma.L = 5
soma.diam = 5
soma.nseg = 5

#Create a dendrite
dend = h.Section(name='dend')
dend.L = 15
dend.diam = 1
dend.nseg = 15

dend.connect(soma(1))

#Tell NEURON we want both sections to be in 3D
rxd.set_solve_type(dimension=3)

#Where
r = rxd.Region(h.allsec(), dx=0.1)
#Who
ca = rxd.Species(r, d=0.3, name='ca', charge=2, initial=lambda node:1 if node.sec==dend and node.segment.x > 0.5 else 0)
#How
bistable_reaction = rxd.Rate(ca, -ca * (1 - ca) * (0.05 - ca))
h.finitialize(-65)

plt.figure(figsize=(15,6))
for i in range(1,50):
    h.continuerun(i*2)
    print(h.t)
    plot_contours(ca)
plt.show()
```

```
def plot_contours(species):
    g = species.nodes.value_to_grid()
    gprime = np.nan_to_num(g)
    plt.subplot(1, 1, 1)
    collapsed = np.max(gprime, axis=1)
    xs, ys = np.meshgrid(range(collapsed.shape[1]), range(collapsed.shape[0]))
    plt.contour(xs, ys, collapsed, 1, colors='k', linewidths=1)
    plt.axis('equal')
    plt.axis('off')
```

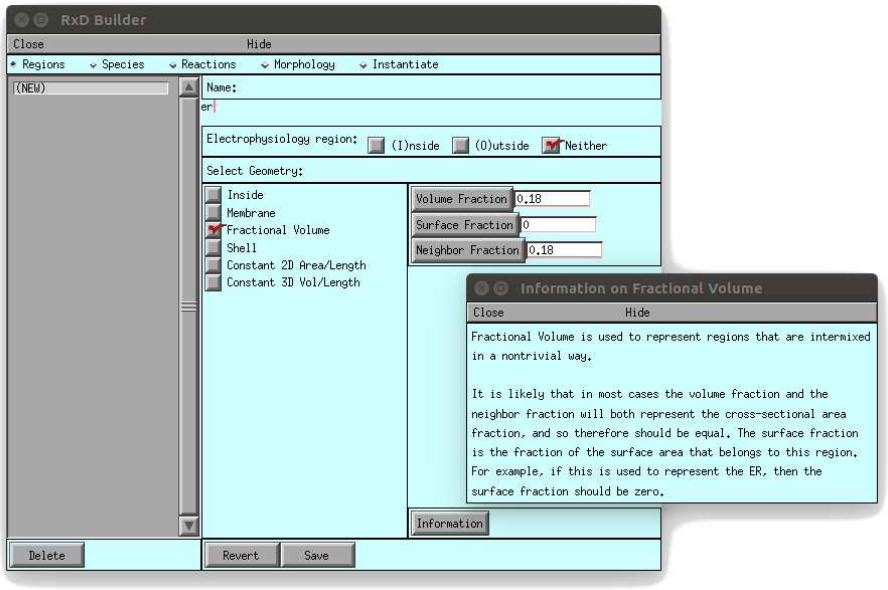


GUI-based specification

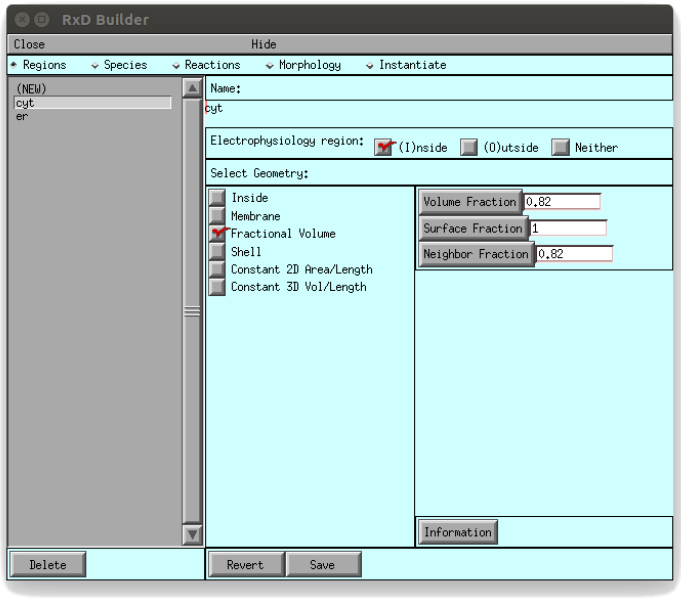
Reaction-diffusion dynamics can also be specified using the GUI. This option appears only when rxd is supported in your install (Python and scipy must be available).



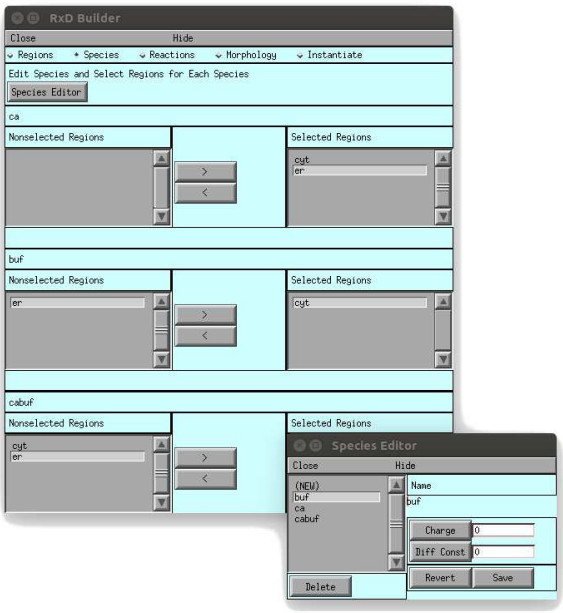
GUI-based specification



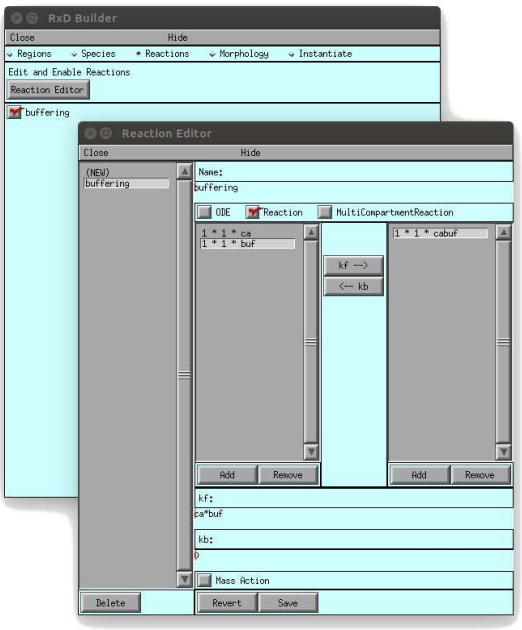
GUI-based specification



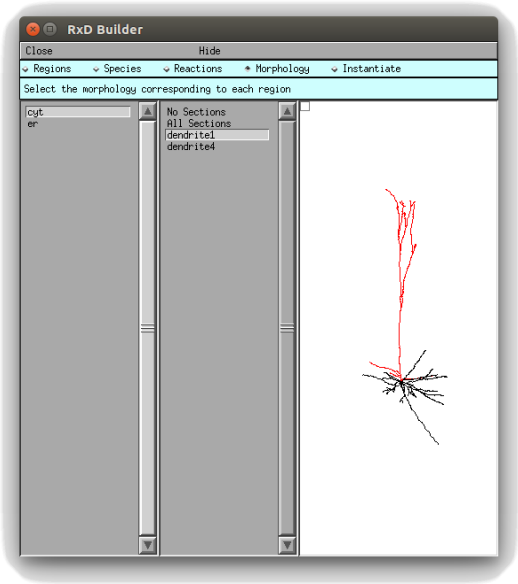
GUI-based specification



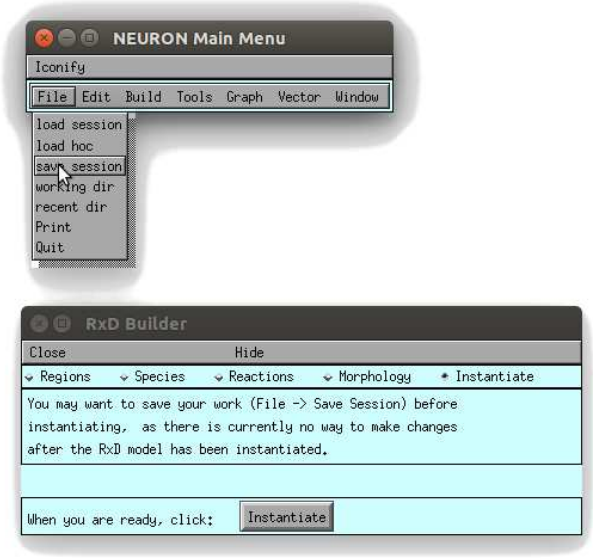
GUI-based specification



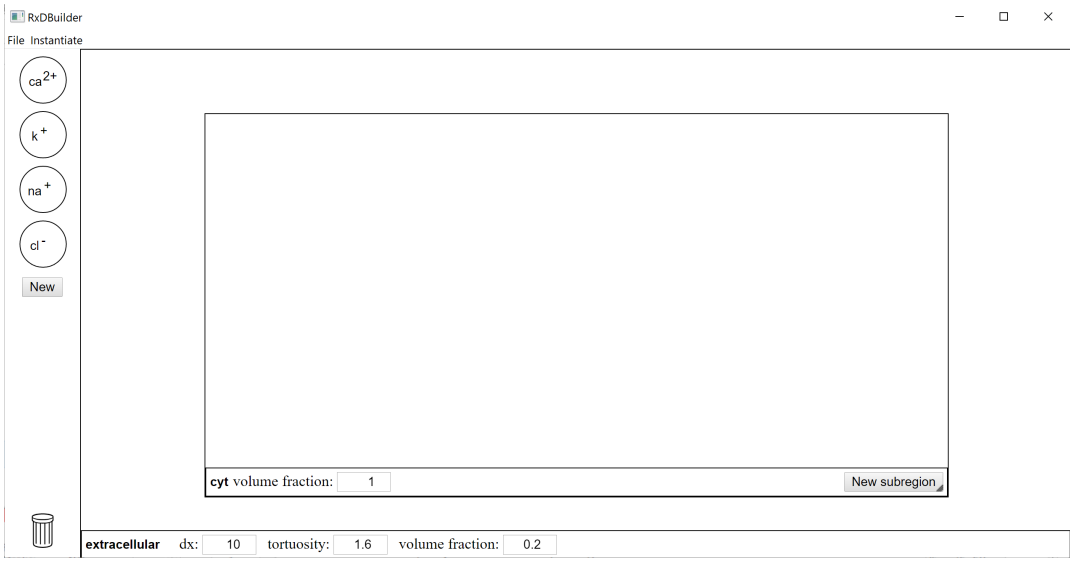
GUI-based specification



GUI-based specification

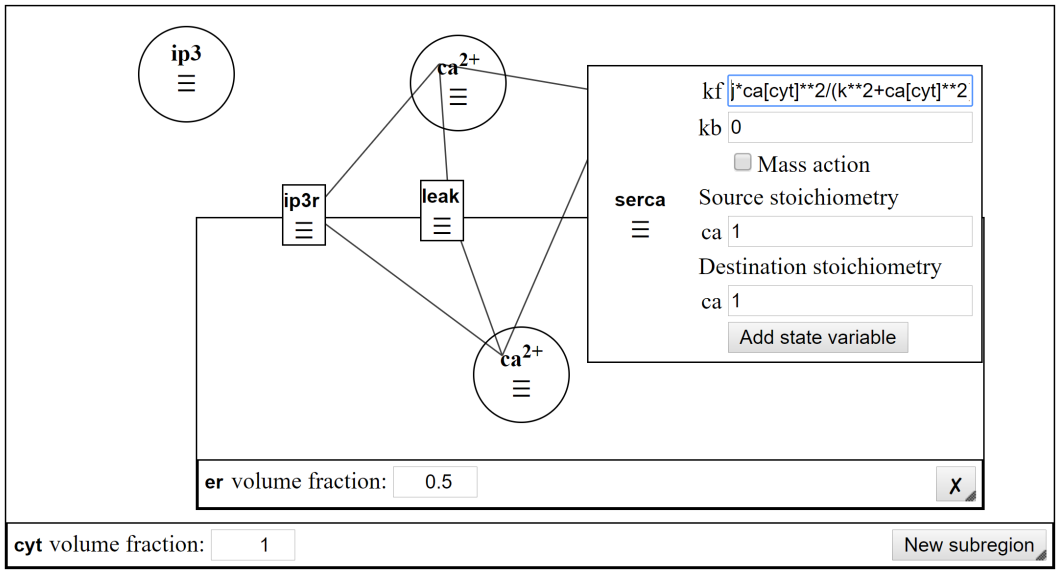


Experimental GUI specification



https://github.com/ramcdougal/rxdbuilder

Experimental GUI specification



https://github.com/ramcdougal/rxdbuilder

For more information

Journal articles on reaction-diffusion in NEURON

- McDougal RA, Hines ML, Lytton WW. (2013). Reaction-diffusion in the NEURON simulator. *Frontiers in Neuroinformatics*, 7.
- McDougal RA, Hines ML, Lytton WW. (2013). Water-tight membranes from neuronal morphology files. *Journal of Neuroscience Methods*, 220(2), 167-178.
- Newton AJH, McDougal RA, Hines ML, Lytton WW. (2018). Using NEURON for reaction-diffusion modeling of extracellular dynamics. *Frontiers in Neuroinformatics*, 12, 41.

Online resources

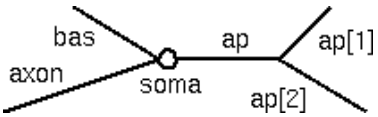
- NEURON forum
- Programmer's reference
- NEURON reaction-diffusion tutorials: https://neuron.yale.edu/neuron/docs/reaction-diffusion

Spatially inhomogeneous parameters

Rules

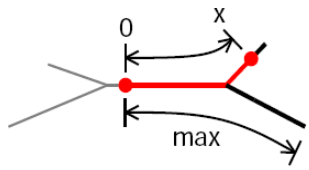
- None (arbitrary values)
- Constant over sets of sections
  use SectionLists (CellBuilder Subsets)
- A function of position

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:

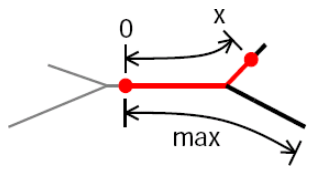
`gnabar_hh = 0.12 * (1 - p)` where `p =` L_{0x}/L_{max}

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

The general problem:

`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")



The general problem:

`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")

Equivalent idioms:

```
forsec subset for (x,0) \
  {rangevar_suffix(x)=f(p(x))} // hoc
for sec in subset:
  for seg in sec:
    sec(seg.x).rangevar = f(p(seg.x)) # Python
```


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 Python or GUI (CellBuilder)

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 in range(10):
    s += f(i)
```

Parallel

```
s = 0
for i in range(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

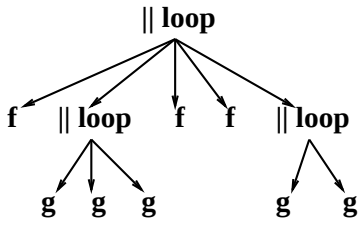
Workstation cluster − 1, 3, 15, 100 machines.

Wide variety of machine speeds.

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

Domain

Very coarse grain parallelization.



NEURON's style

A bulletin board
. . . on top of MPI.

Launching a parallel program

```
pc = h.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.py

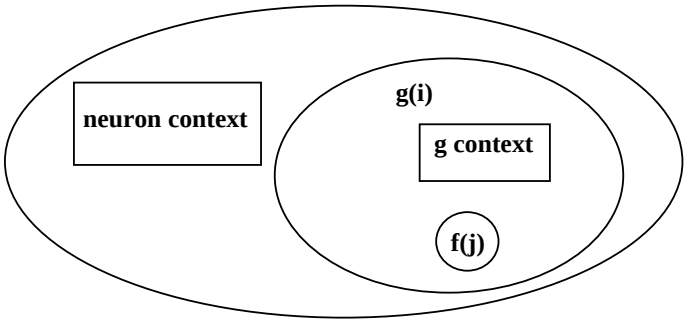
```
from neuron import h
pc = h.ParallelContext()

def f(j):
  s = 0
  for i in range(100000):
    s += j
  return s

pc.runworker()

runtime = h.startsw()
s = 0

for i in range(10):
  pc.submit(f, i)

while (pc.working()):
  s += pc.pyret()

print "sum = ", s
print "runtime ", h.startsw() - runtime
pc.done()
h.quit()
```



```
$ mpiexec -n 1 nrniv -mpi example.hoc
numprocs=1
NEURON -- VERSION 7.5 (1512:e0bd0137f04c) 2017-01-30
...
sum = 4500000
runtime 0.099999...

$ mpiexec -n 4 nrniv -mpi example.hoc
numprocs=4
NEURON -- VERSION 7.5 (1512:e0bd0137f04c) 2017-01-30
...
sum = 4500000
runtime 0.039999...

$
```

Context and Communication

NEURON



post → **Bulletin board**

take
look ← **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()
```


Numerical Methods

Accuracy, stability, speed

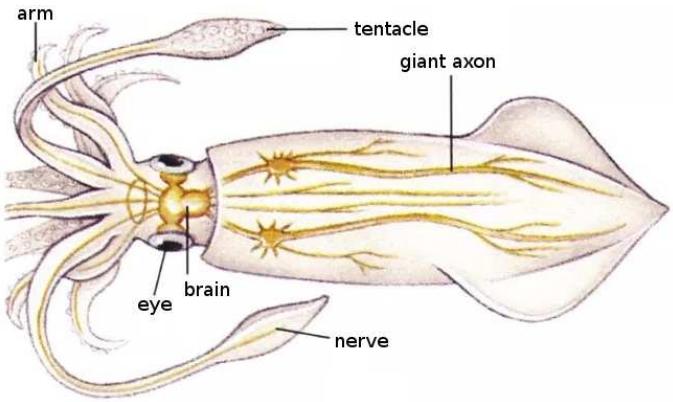
Robert A. McDougal

Yale School of Medicine

9 August 2018

Hodgkin and Huxley: squid giant axon experiments

Top: Alan Lloyd Hodgkin; Bottom: Andrew Fielding Huxley. Images from Wikipedia.



Adapted from Pearson Education 2009.

Hodgkin and Huxley equations

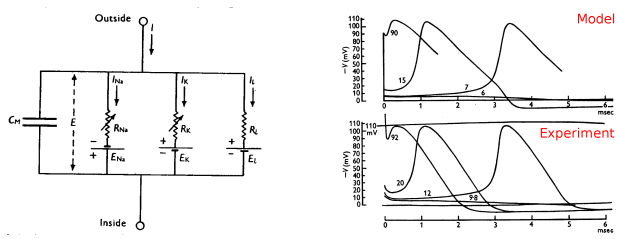
Top: Alan Lloyd Hodgkin; Bottom: Andrew Fielding Huxley. Images from Wikipedia.

$$C\frac{dV}{dt} = -\left(g_{Na}m^3h(V-E_{Na}) + g_K n^4(V-E_K) + g_\ell(V-E_\ell)\right)$$

$$\frac{dm}{dt} = \alpha_m(V)(1-m) - \beta_m(V)m$$

$$\frac{dh}{dt} = \alpha_h(V)(1-h) - \beta_h(V)h$$

$$\frac{dn}{dt} = \alpha_n(V)(1-n) - \beta_n(V)n$$



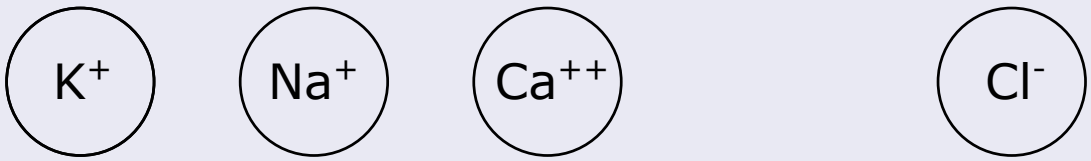
Adapted from Hodgkin and Huxley 1952.

What does it mean?

Electronics 101

Current

Current is the movement of charge. In electronics, current is carried by the movement of electrons. In neurons, current flows across the membrane by the movement of ions. These ions can be positively or negatively charged.



Resistors = Conductors

A **resistor** is a material that impedes current flow. This includes essentially all materials. For those materials obeying **Ohm's law**,

$$v = IR$$

where v is the voltage drop across the resistor, I is the current, and R is the **resistance** (this may be constant or a function of time).

This may alternatively be written as

$$I = gV$$

where $g = 1/R$ is the **conductance**.

Ion channels

Ion channels allow current to pass in the form of moving ions. They are therefore resistors. The resistance varies over time.

Capacitors

A **capacitor** accumulates charge according to

$$CV = Q$$

where Q is the charge, V is the potential, and C is the capacitance.

The capacitive current is the rate at which charge is being stored on the current, dQ/dt. Thus differentiating both sides of the above, we find

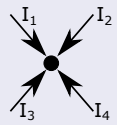
$$C\frac{dV}{dt} = \frac{dQ}{dt} = I.$$

Cell membrane

Charged ions accumulate along a neuron's membrane. It is therefore a capacitor.

Kirchhoff's Current Law

The algebraic sum of currents in a network of conductors meeting at a point is zero.

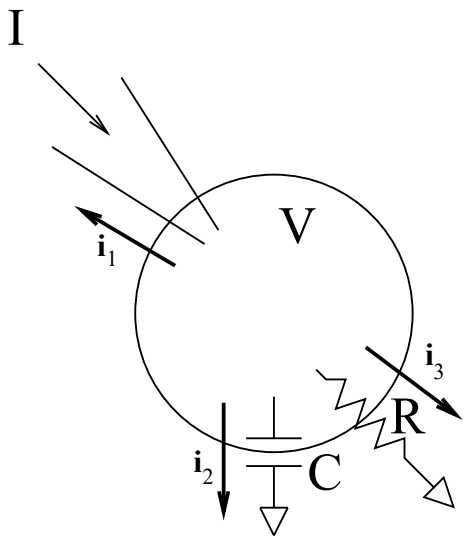


$$\sum_k I_k = 0.$$

Wording from https://en.wikipedia.org/wiki/Kirchhoff%27s_circuit_laws

Putting it together: the electronics of a neuron

Consider a simplified cell with three currents:



By Kirchoff,

$$0 = i_1 + i_2 + i_3$$
$$= -I + C\frac{dV}{dt} + gV$$

Rearranging terms, we conclude:

$$C\frac{dV}{dt} = -gV + I.$$

The Hodgkin-Huxley equations account for a pull on ions due to the balance of chemical and electrical gradients. This approximately acts as a battery with potential E associated with each resistor and leads to terms of the form g(V - E).

Solving a differential equation

Consider the differential equation

$$C\frac{dV}{dt} = -gV + I, V(0) = V_0$$

We can solve this for $V(t)$ by **separation of variables**:

$$\frac{dV}{I - gV} = \frac{dt}{C}$$
$$\int \frac{dV}{I - gV} = \int \frac{dt}{C}$$
$$\frac{-1}{g} \ln |I - gV| = \frac{t}{C} + c_1$$
$$I - gV = c_2 e^{-gt/C}$$

Therefore,

$$V = \frac{1}{g}\left(I - c_2 e^{-gt/C}\right).$$

We can then solve for c_2 by plugging in $V(0) = V_0$:

$$V_0 = \frac{1}{g}(I - c_2)$$

so

$$c_2 = I - gV_0$$

and thus

$$V = \frac{1}{g}\left(I - (I - gV_0)e^{-gt/C}\right).$$

Note: This is a lot of work and is only possible because the equation is simple. This type of equation appears in leaky integrate and fire and is the basis of the `cnexp` solver.

To solve general differential equations, we must use numerical techniques.

Here we're assuming g is a constant. This is not true for voltage gated ion channels.

In the **Explicit Euler** method, we approximate

$$\frac{dy}{dt} \approx \frac{\Delta y}{\Delta t}$$

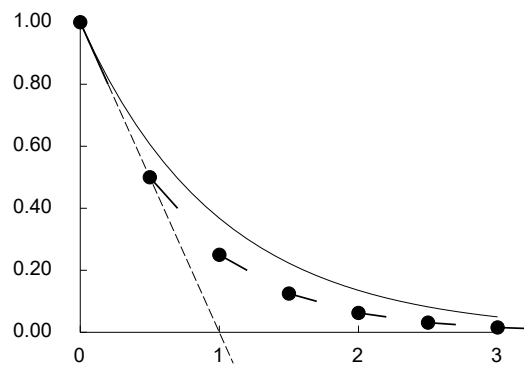
for some small time step Δt and estimate the function at a series of time points. Here $\Delta y_n = y_{n+1} - y_n$ and $\Delta t_n = t_{n+1} - t_n$.

Then starting from some initial point (t_0, y_0), we approximate $\frac{dy}{dt} = f(t, y)$ as $\frac{\Delta y_n}{\Delta t_n} = f(t_n, y_n)$ and thus

$$\Delta y_n = \Delta t_n f(t_n, y_n)$$

and therefore

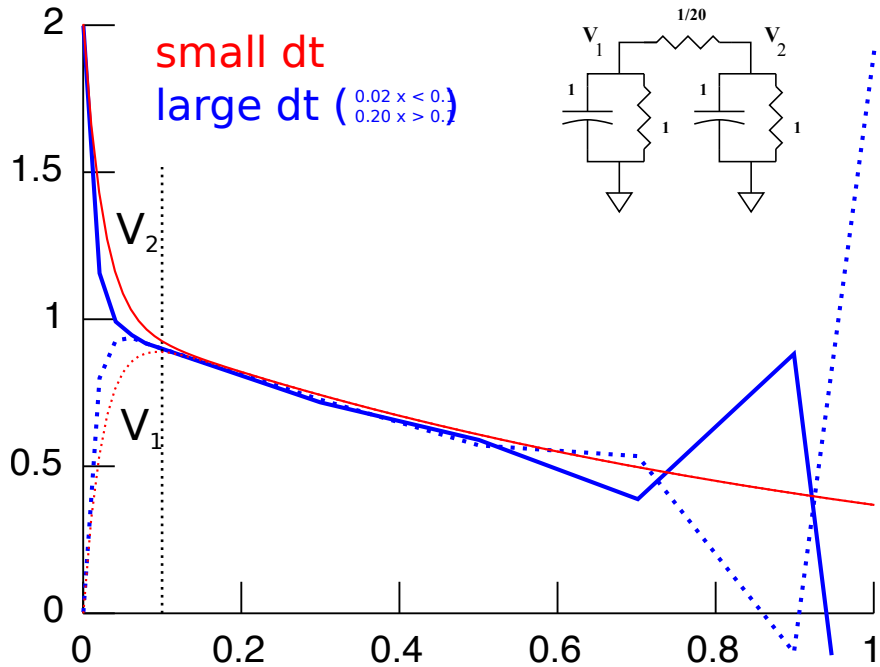
$$y_{n+1} = y_n + \Delta t_n f(t_n, y_n).$$



Explicit Euler starts at a point, moves in the direction of the tangent line (slope dy/dt) for a time Δt, then repeats.

Explicit Euler is numerically unstable

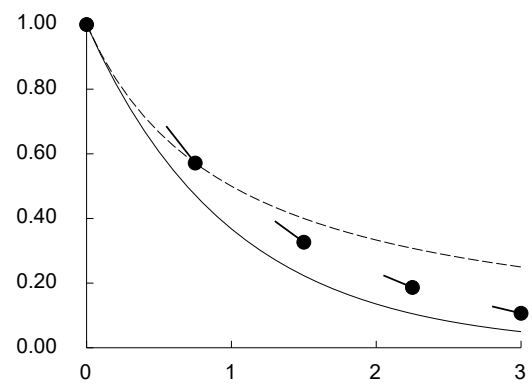
If the time step in Explicit Euler is too large, the solution will be unstable:



The **Implicit Euler** method is almost the same as the Explicit Euler method except instead of evaluating at $f(t_n, y_n)$, we evaluate at $f(t_{n+1}, y_{n+1})$. That is,

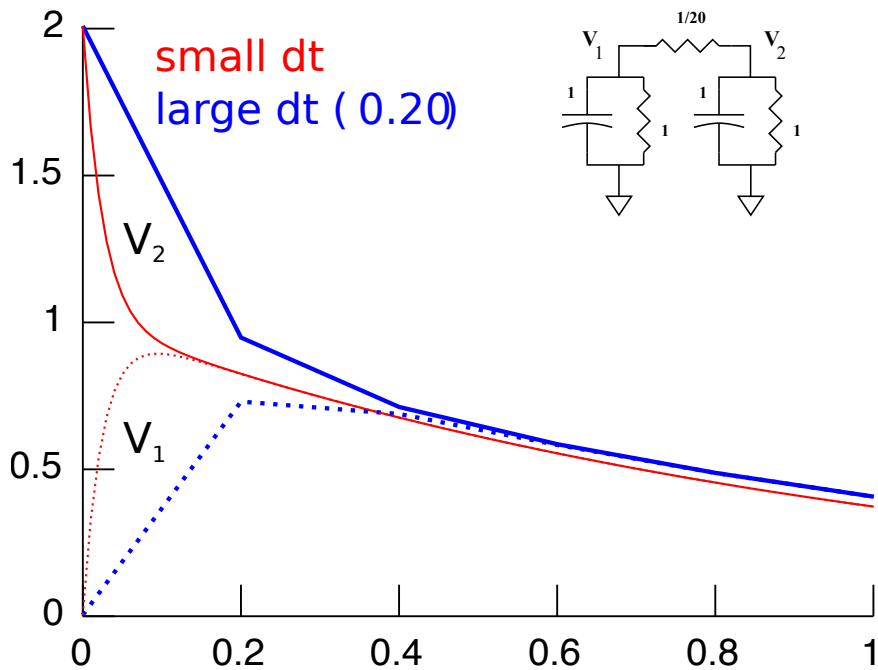
$$y_{n+1} = y_n + \Delta t_n f(t_{n+1}, y_{n+1}).$$

Note that y_{n+1} is on both sides, and thus we have an algebraic equation that must be solved to find y_{n+1}.



Implicit Euler finds a new point such that if we moved in the direction of the tangent line (slope dy/dt) backward in time by Δt, we would get where we started.

Implicit Euler is numerically stable



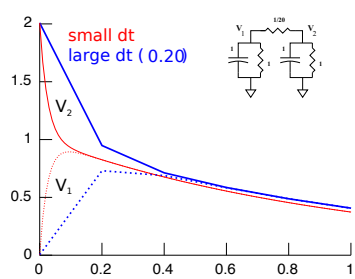
As Implicit Euler is numerically stable, it is NEURON's default integration method.

Accuracy of Implicit Euler

Note that the solutions found with a small *dt* and a large *dt* are different, even after the initial rapid change.

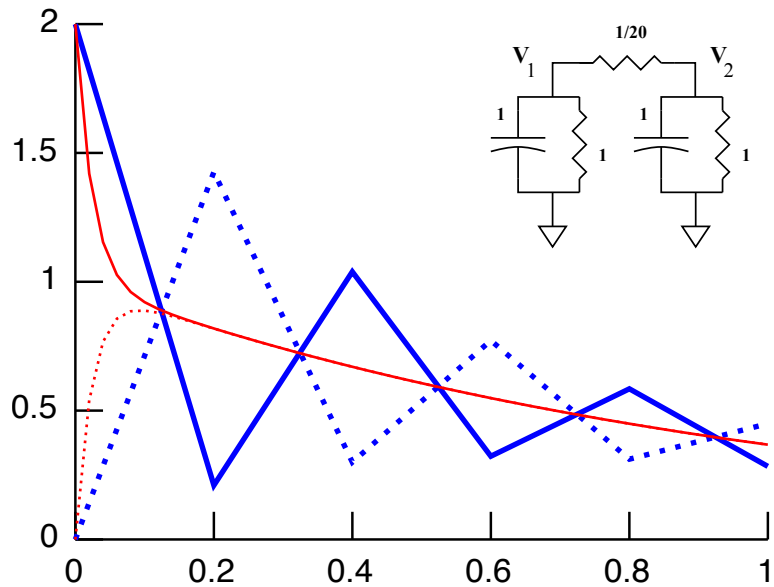
One can prove that halving *dt* will approximately halve the difference between the computed value and the true value.

Thus **Implicit Euler is a first order method**.



Error convergence estimates are true in the limit as $dt \to 0$.

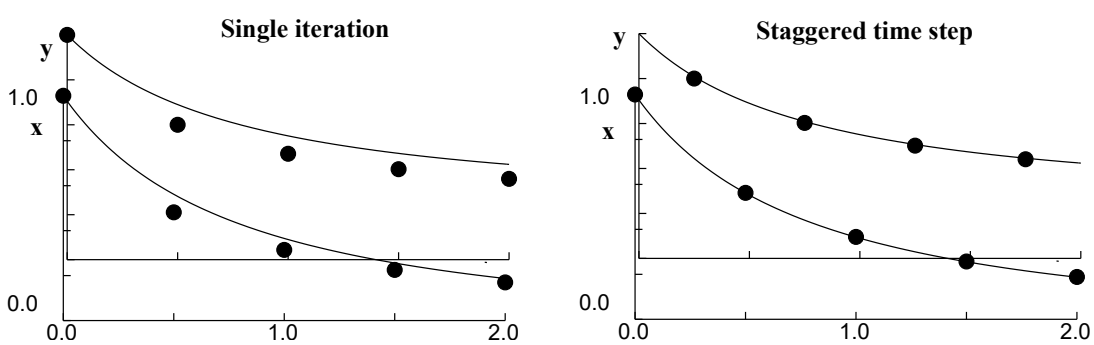
Crank-Nicolson is stable but can oscillate



NEURON also supports the second-order Crank-Nicolson method (`h.secondorder=2`). The solution is stable and converges faster than Implicit or Explicit Euler, but it can exhibit oscillations.

If `h.secondorder=2`, then membrane potentials are second order correct at time t, currents at $t - dt/2$, and channel conductances at $t + dt/2$. To plot these correctly in NEURON, use a voltage axis, current axis, or state axis, respectively.

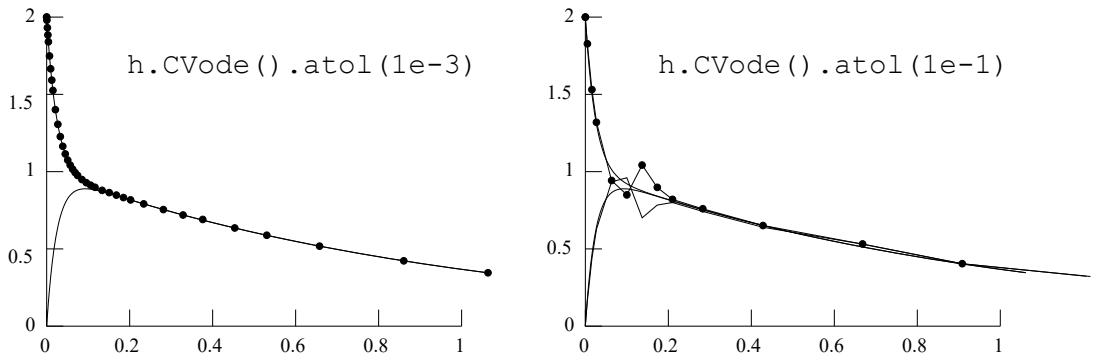
$$\dot{x} = -1.4xy, \quad \dot{y} = -xy$$



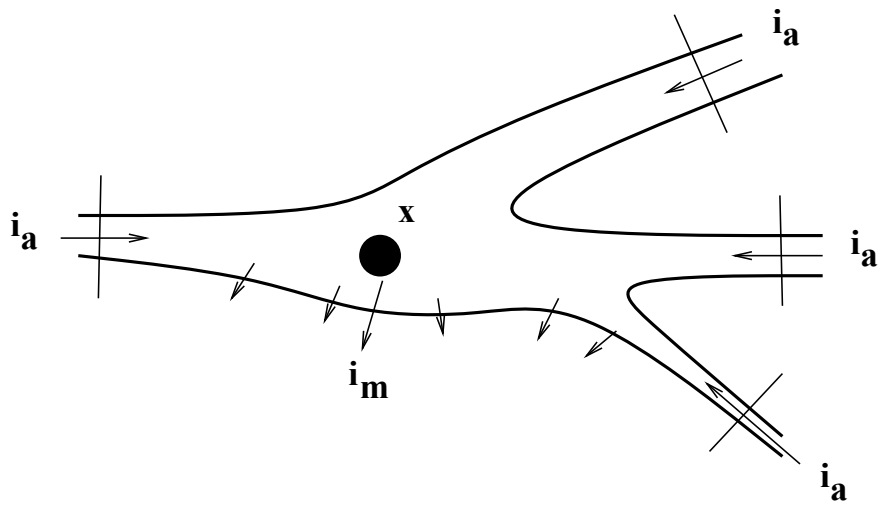
Variable time steps

So far, we have considered numerical error as a function of the time step *dt*. We can instead choose an error tolerance and use that to pick a new *dt* at each time step.

NEURON provides the `CVode` object for enabling variable step simulation.

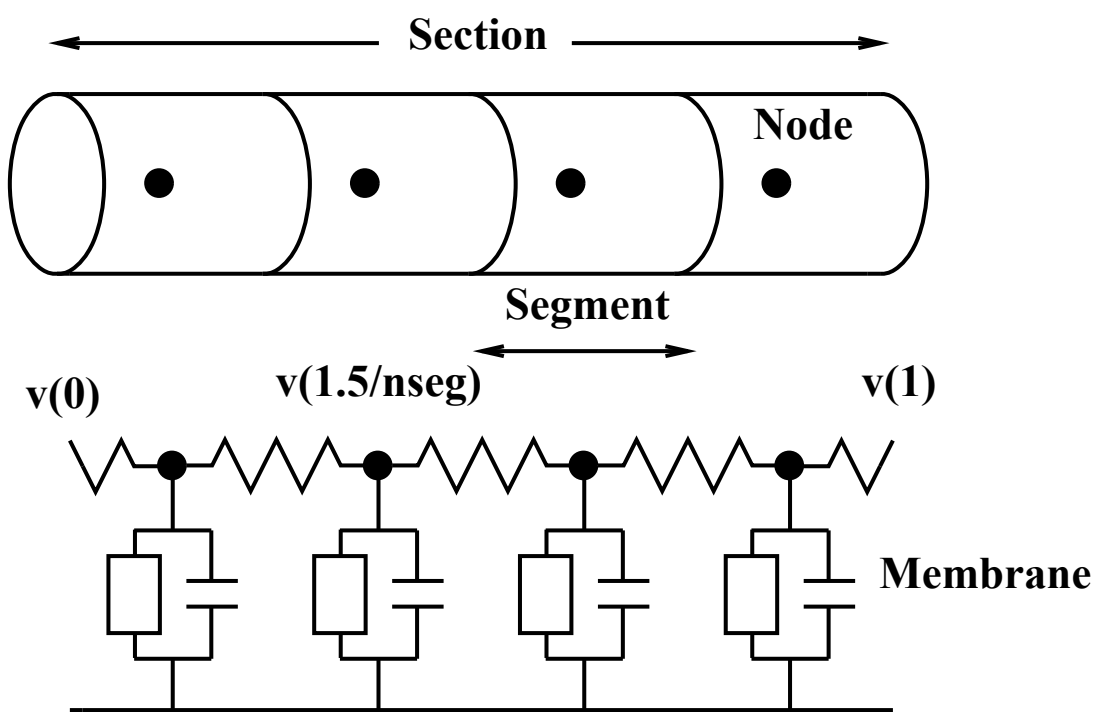


Incorporating space



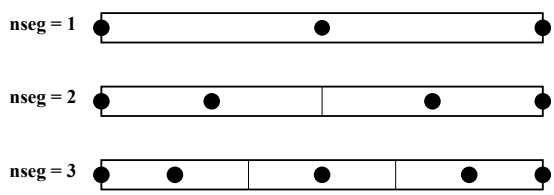
$$\int i_m = \sum i_a$$

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



Improving accuracy by increasing nseg

Improve accuracy by reducing the size of spatial compartments. In NEURON, do this by increasing `nseg`, the number of segments:



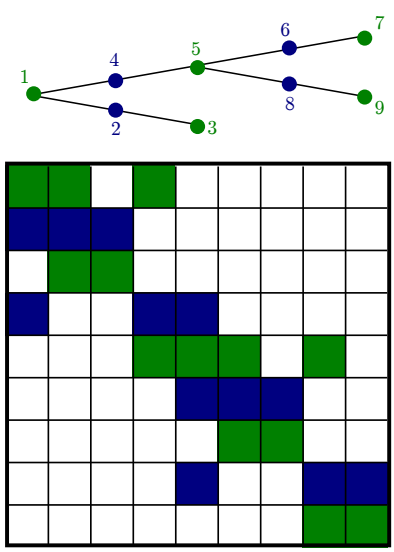

```
for sec in h.allsec():
    sec.nseg *= 3
```

Note that you must multiply `nseg` by an **odd** number to preserve the location of the computed values, which is essential to testing convergence.

Trees can be solved stably in $\mathcal{O}(n)$
Only unstable methods can solve arbitrary shapes in $\mathcal{O}(n)$

To solve $A\Delta y = b$ where y and b have n entries (e.g. if we want to solve for 4 variables at $n/4$ points) takes time proportional to:

- n^3 via Gaussian Elimination
- $n^{\log_2 7}$ via Strassen (1969)
- n if A corresponds to a "tree-matrix" (e.g. a neuron) discretized in a certain way (right).



McDougal et al 2013

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

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

Communication between cells

Gap junctions

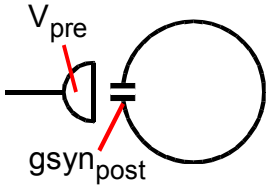
Synaptic transmission
- graded
- spike-triggered

Graded synaptic transmission

Physical system:

A presynaptic variable governs continuous transmitter release

Transmitter modulates a postsynaptic property



$gsyn_{post} = f(V_{pre})$

Problem: how does postsynaptic cell know V_{pre}?

Graded synaptic transmission *continued*

Answer: use POINTER to link postsynaptic variable to the presynaptic variable

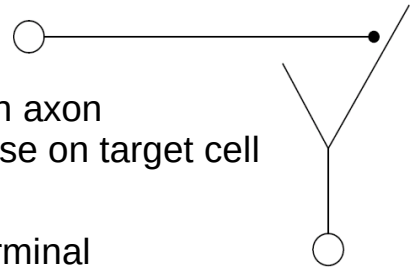
NMODL specification of synaptic mechanism:

```
NEURON {
    POINT_PROCESS Syn
    POINTER v_pre
}
```

hoc usage

```
objref syn
dend syn = new Syn(0.5)
setpointer syn.v_pre, precell.axon.v(1)
```


Spike-triggered synaptic transmission

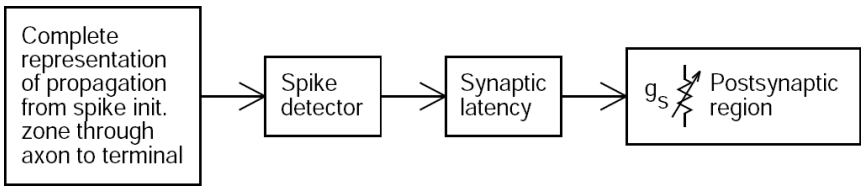


Physical system:
- Presynaptic neuron with axon that projects to synapse on target cell

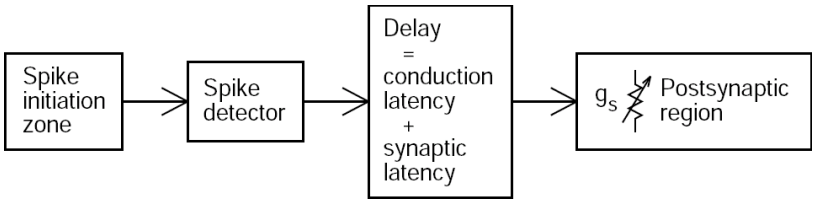
Conceptual model:
- Spike in presynaptic terminal triggers transmitter release; presynaptic details unimportant
- Postsynaptic effect described by DE or kinetic scheme that is perturbed by occurrence of a presynaptic spike

Spike-triggered transmission: computational implementation

Basic idea



More efficient: "virtual spike propagation"



The NetCon class

Python usage

```
nc = h.NetCon(source, target)
nc = h.NetCon(source_ref_v, target
     [, threshold, delay, weight,
     sec = section])
```

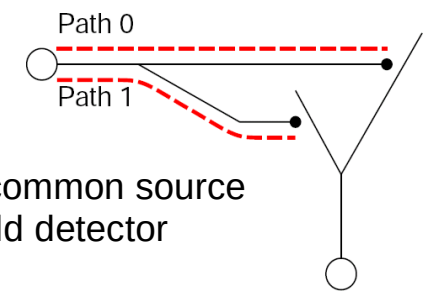
Defaults

```
nc.threshold = 10
nc.delay = 1 # must be >= 0
nc.weight[0] = 0 # weight is an array
```

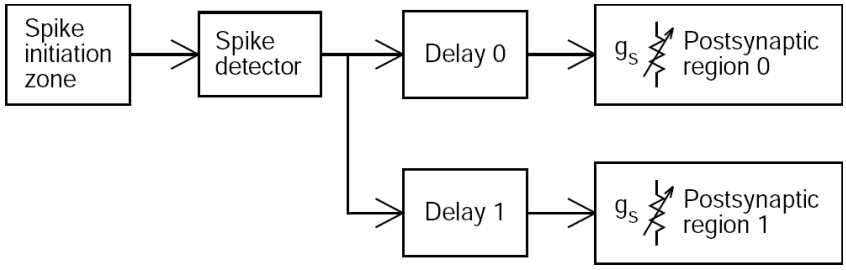
NMODL specification of synaptic mechanism

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

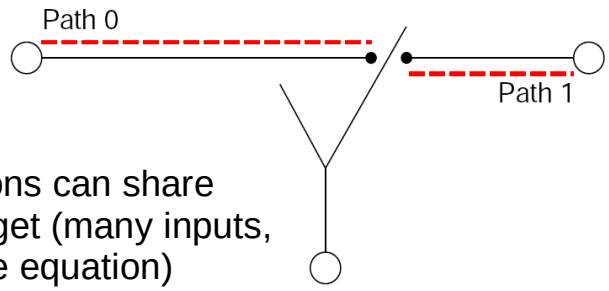
Efficient divergence



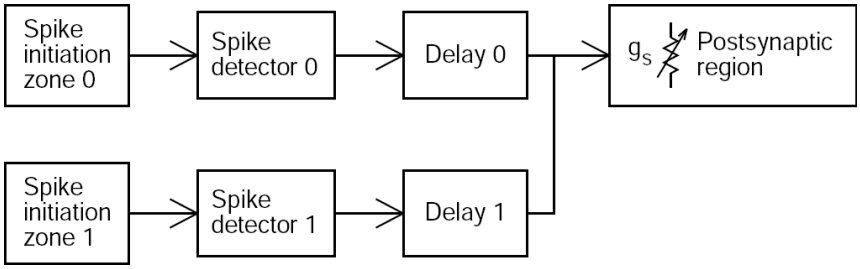
Multiple NetCons with a common source share a single threshold detector



Efficient convergence



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

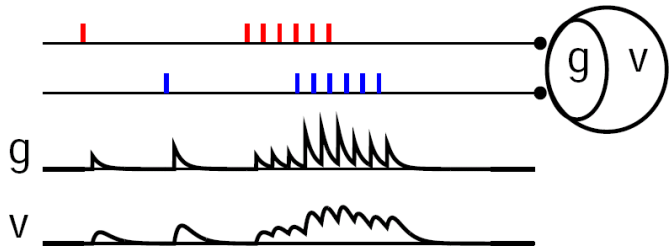


Example: g_s with fast rise and exponential decay

```
NEURON {
  POINT_PROCESS ExpSyn
  RANGE tau, e, i
  NONSPECIFIC_CURRENT i
}
  . . . declarations . . .
INITIAL { g = 0 }
BREAKPOINT {
  SOLVE state METHOD cnexp
  i = g*(v-e)
}
DERIVATIVE state { g' = -g/tau }
NET_RECEIVE(w (uS)) { g = g + w }
```

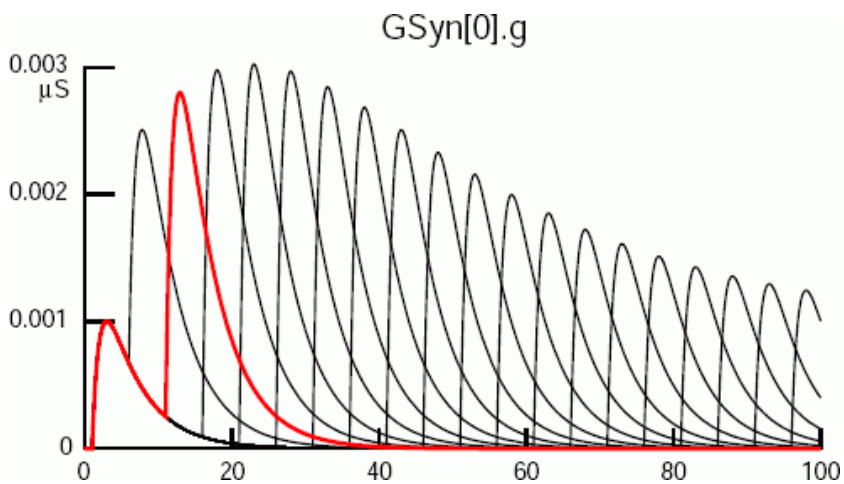

g_s with fast rise and exponential decay

continued



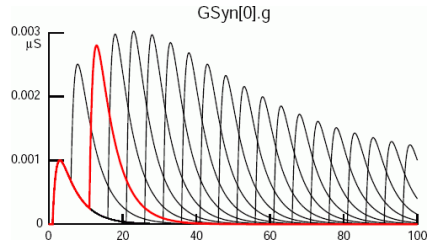

```
BREAKPOINT {
  SOLVE state METHOD cnexp
  i = g*(v-e)
}
DERIVATIVE state { g' = -g/tau }
NET_RECEIVE(w (uS)) { g = g + w }
```

Example: use-dependent synaptic plasticity



Use-dependent synaptic plasticity *continued*

```
BREAKPOINT {
  SOLVE state METHOD cnexp
  g = B - A
  i = g*(v-e)
}
DERIVATIVE state {
  A' = -A/tau1
  B' = -B/tau2
}
NET_RECEIVE(weight (uS), w, G1, G2, t0 (ms)) {
  INITIAL {w=0 G1=0 G2=0 t0=t}
  G1 = G1*exp(-(t-t0)/Gtau1)
  G2 = G2*exp(-(t-t0)/Gtau2)
  G1 = G1 + Ginc*Gfactor
  G2 = G2 + Ginc*Gfactor
  t0 = t
  w = weight*(1 + G2 - G1)
  g = g + w
  A = A + w*factor
  B = B + w*factor
}
```



Artificial spiking cells

"Integrate and fire" cells

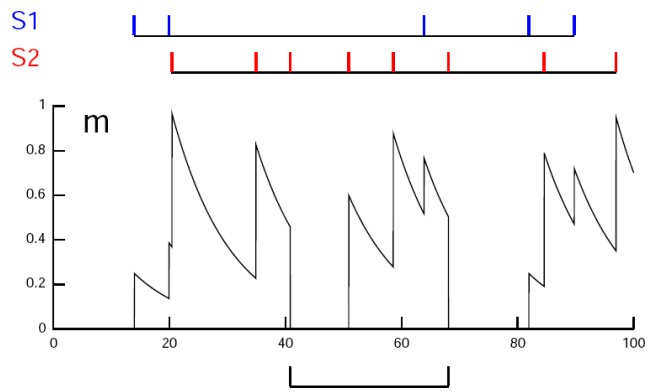
Prerequisite: all state variables must be analytically computable from a new initial condition

Orders of magnitude faster than numerical integration

Event-driven simulation run time is
- *proportional* to # of received events
- *independent* of # of cells, # of connections, and problem time

Hybrid networks

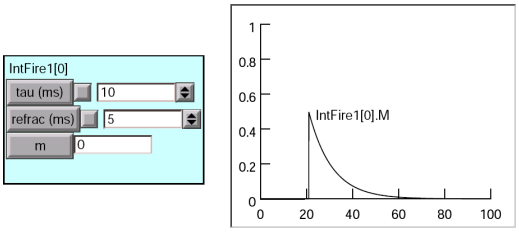
Example: leaky integrate and fire model



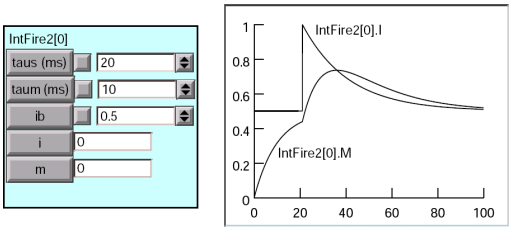
Leaky integrate and fire model *continued*

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

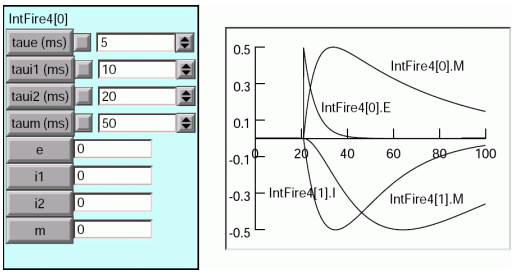

IntFire1



IntFire2



IntFire4



Defining the types of cells

Artificial spiking cells

- `ARTIFICIAL_CELL` with a `NET_RECEIVE` block that calls `net_event`
- `NetStim`, `IntFire1`, `IntFire2`, `IntFire4`

Biophysical model cells

- "Real" model cells
- Sections and density mechanisms
- Synapses are `POINT_PROCESS`es that affect membrane current and have a `NET_RECEIVE` block, e.g. `ExpSyn`, `Exp2Syn`

Defining types of biophysical model cells

Encapsulate in a class

Export `hoc` class definition from CellBuilder or Network Builder
or
write your own in Python.

```
class Cell:
  def __init__(self)
    # specify geom, topol, biophys
    soma = h.Section(name='soma')
    self.soma = soma
    ... etc. ...

cells[]
N = 1000
for i in range(N):
  cell = Cell() # h.Cell() if Cell is defined in hoc
  cells.append(cell)
```

Connecting cells

Which setup strategy is more efficient?

Iterate over sources

```
for each cell {
  connect this cell to its targets
}
```

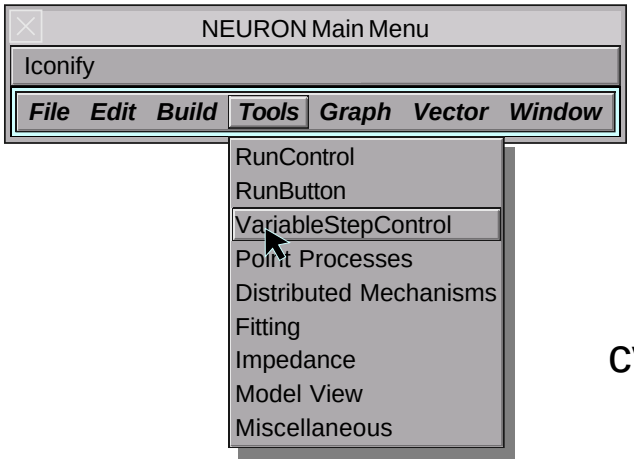
or iterate over targets?

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

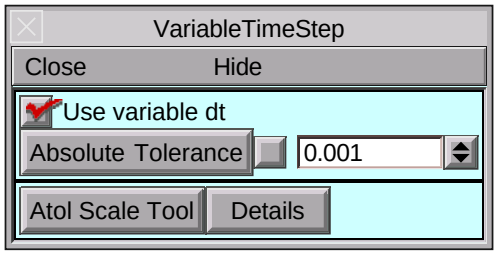

Connecting cells

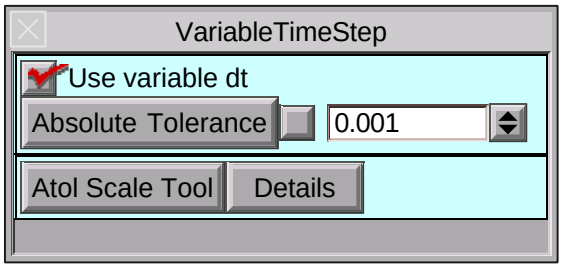
For a net distributed over multiple CPUs,
it is most efficient to iterate over targets first.

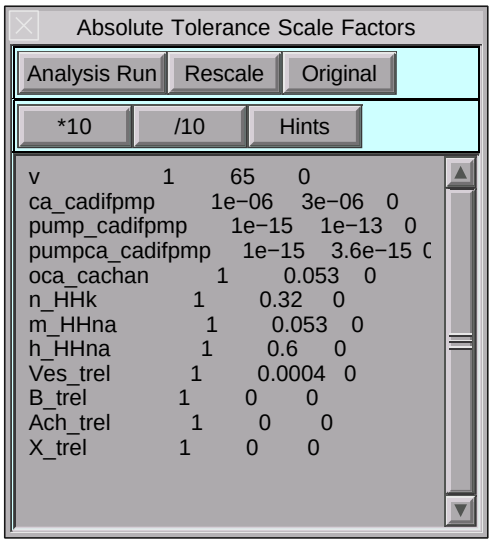
```
for each cell {
  connect sources to this cell
}
```

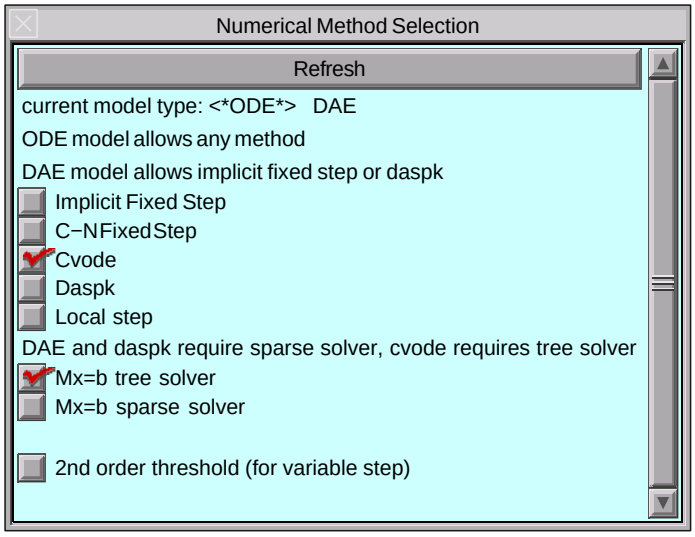



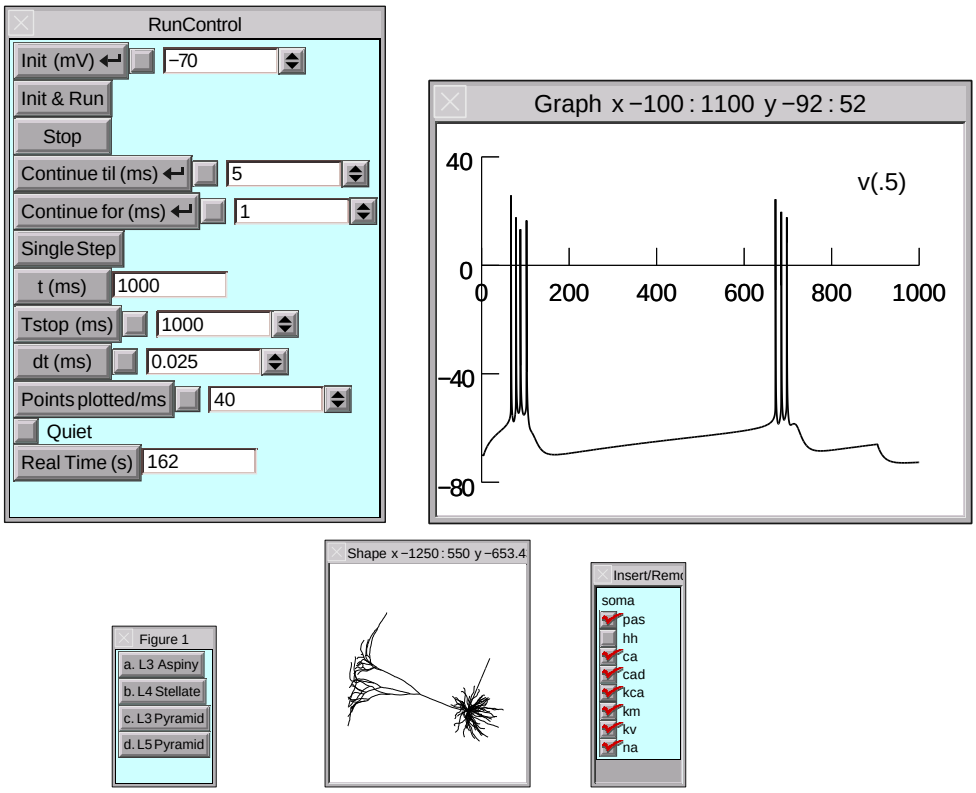
cvode_active(1)

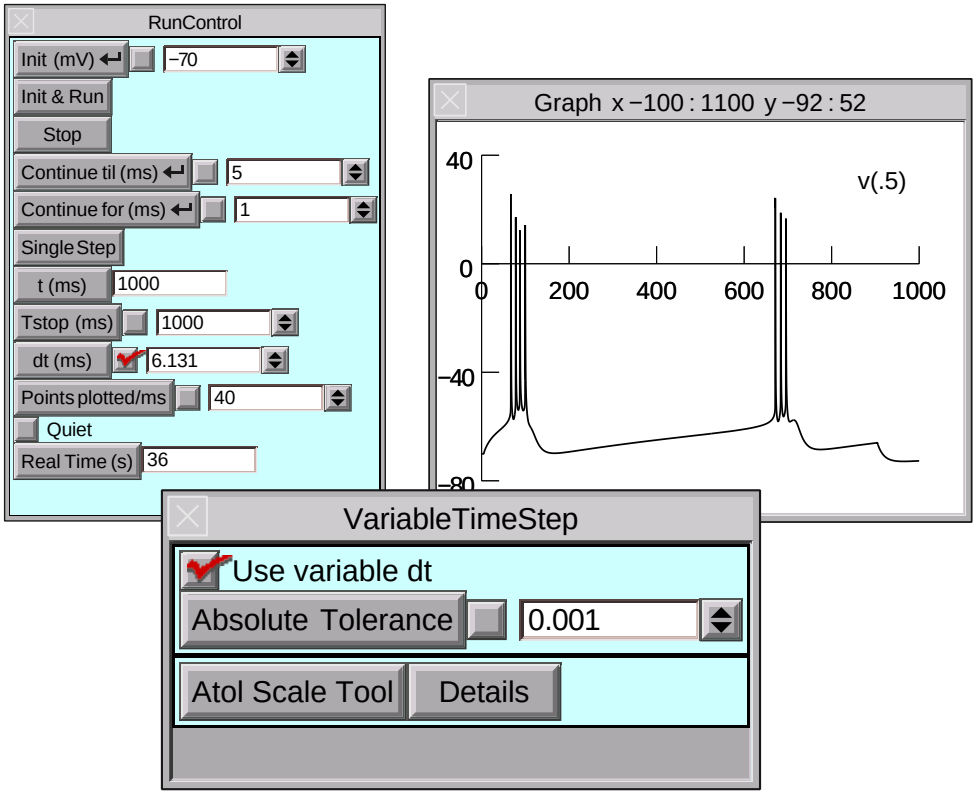


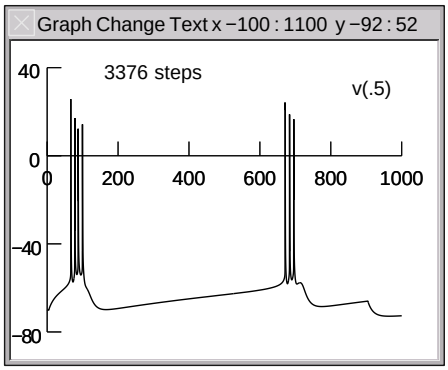


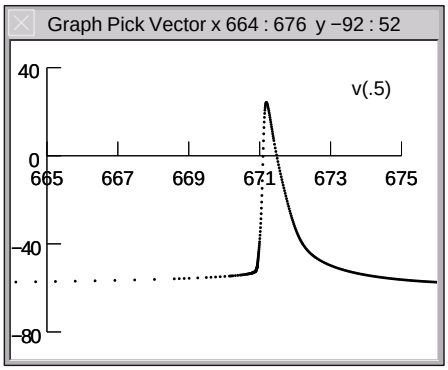


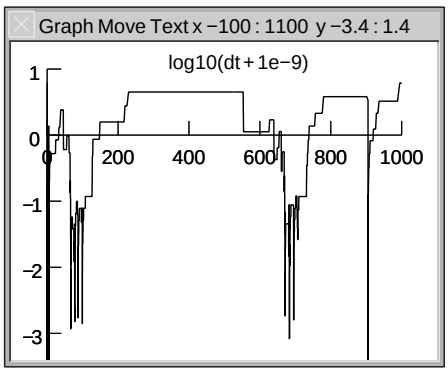


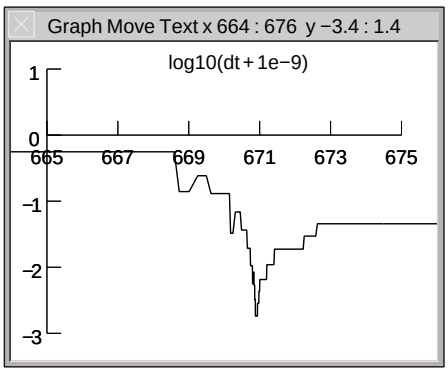


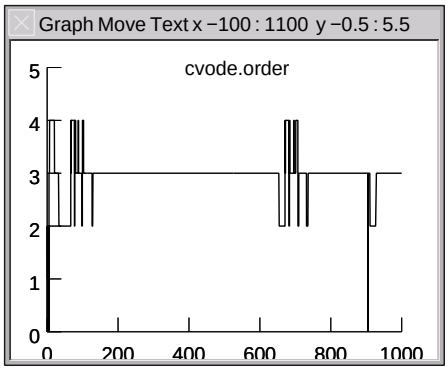


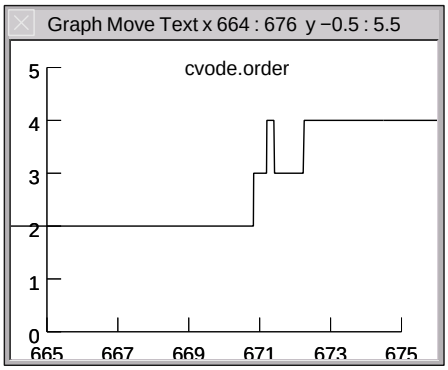












Spinal Motor Neuron: McIntyre et al 2002

Simulation of peripheral nervous system (PNS) mylelinated axon. This model is described in detail in: McIntyre CC, Richa
Grill WM.(2002)
Reference: McIntyre CC, Richardson AG, Grill WM (2002) Modeling the excitability of Mammalian nerve fibers: influenc
afterpotentials on the recovery cycle. *J Neurophysiol* **87**:995-1006 [PubMed]

Citations Citation Browser

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

Model Type: Axon;
Cell Type(s): Spinal motor neuron;
Channel(s): I Na,p; I Na,t; I K; I Sodium; I Potassium;
Receptor(s):
Transmitter(s):
Simulation Environment: Neuron;
Model Concept(s): Axonal Action Potentials; Action Potentials;
Implementer(s): MacIntyre, CC ;

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

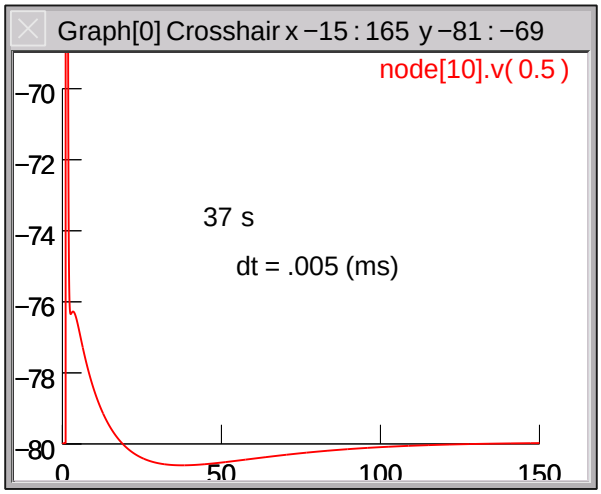
Model files Download zip file | Auto-launch | **Help downloading and running models**

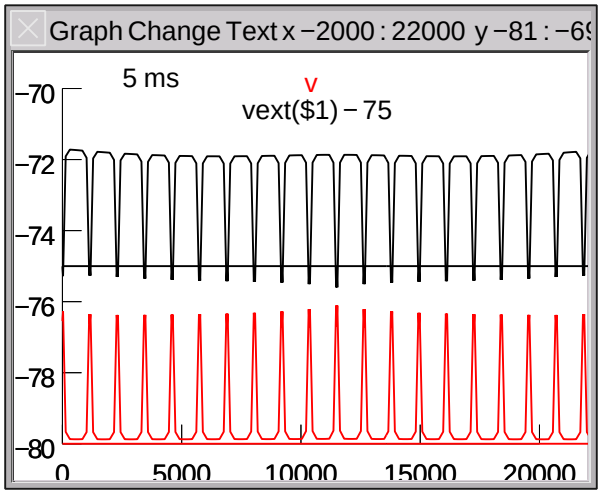
- \
  - MRGaxon
    - README
    - AXNODE.mod
    - MRGaxon.hoc
    - mosinit.hoc
    - MRGaxon.ses

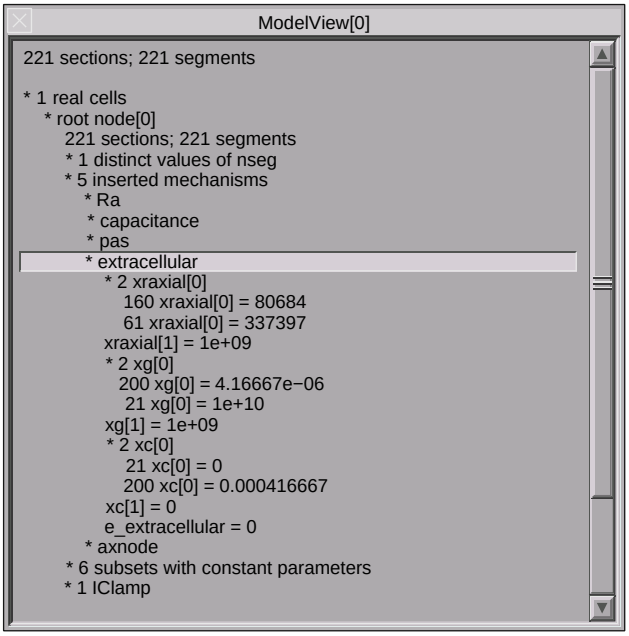
```
SIMULATION OF PNS MYELINATED AXON

This model is described in detail in:

McIntyre CC, Richardson AG, and Grill WM. Modeling the excitability
mammalian nerve fibers: influence of afterpotentials on the recover
cycle. Journal of Neurophysiology 87:995-1006, 2002.

The model is set up to reproduce part of Fig 2A from this paper.

This model can not be used with NEURON v5.1 as errors in the
extracellular mechanism of v5.1 exist related to xc. The original
stimulations were run on v4.3.1. NEURON v5.2 has corrected the
limitations in v5.1 and can be used to run this model.

Please contact mcintyre@bme.jhu.edu if you have any questions about
```

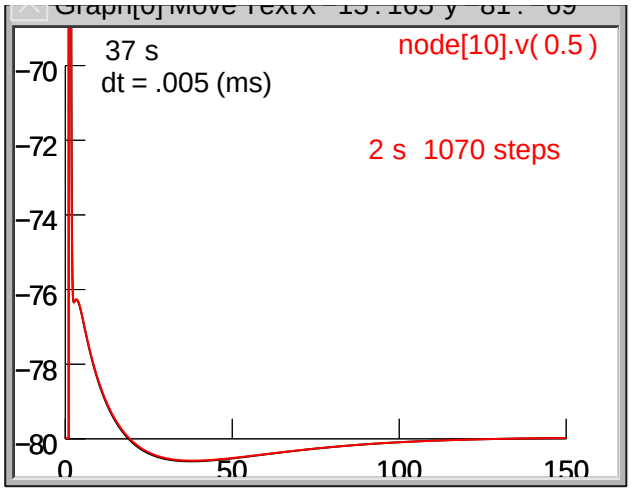
Total site hits since January 1, 2002: **346093**

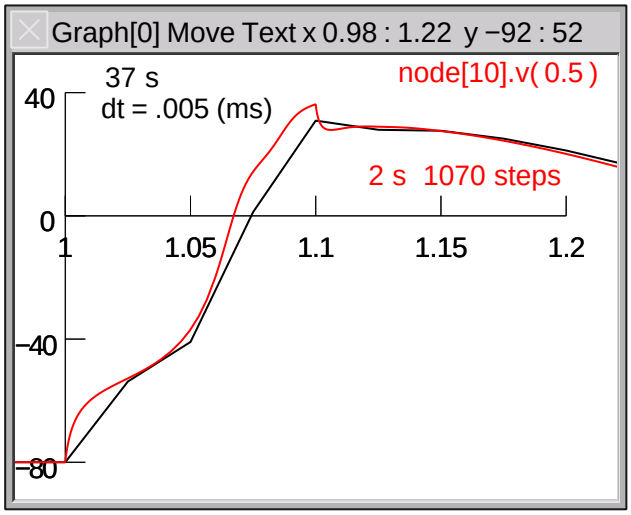
ModelDB Home SenseLab Home Help
Questions, comments, problems? Email the *ModelDB Administrator*
How to cite ModelDB

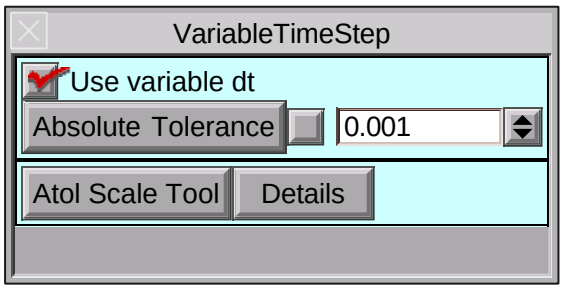


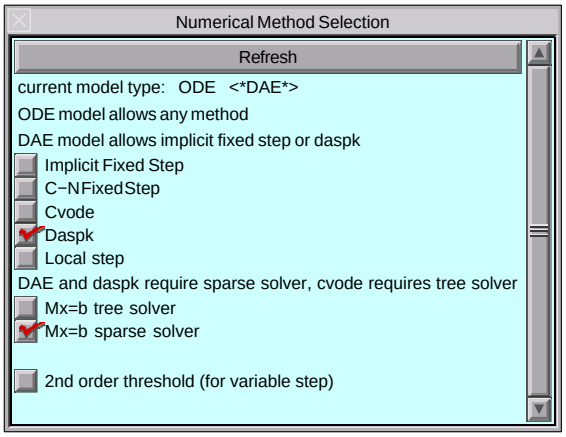


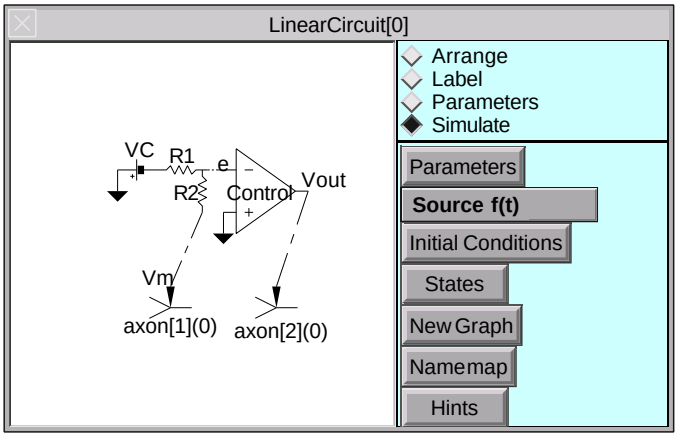


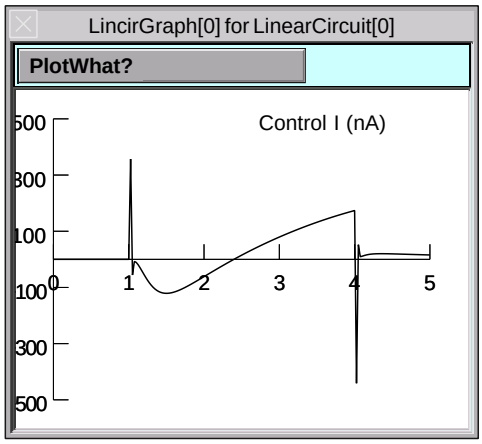


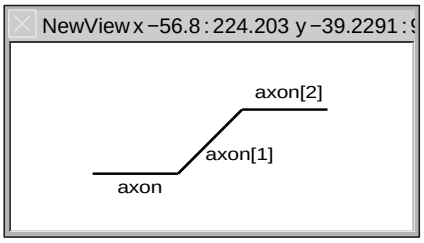


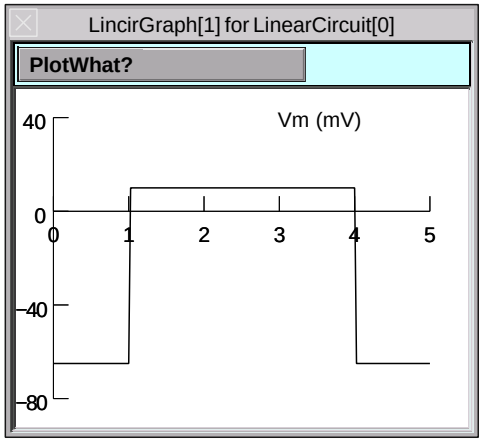


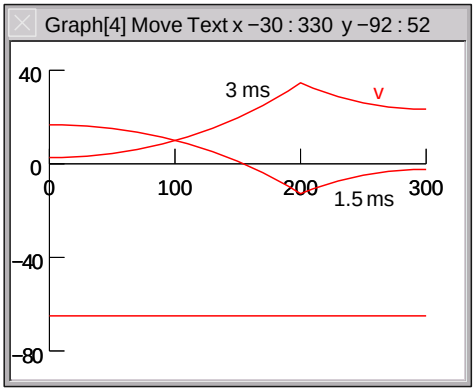


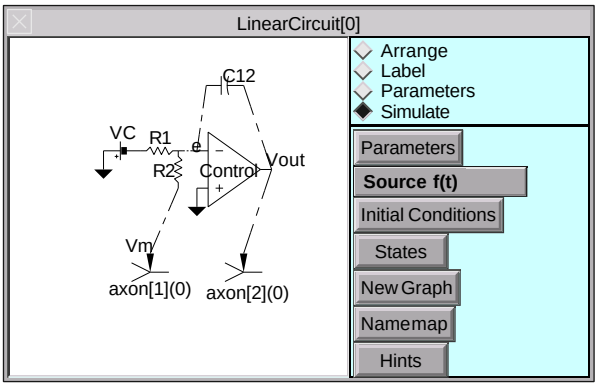


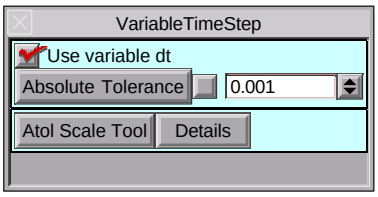


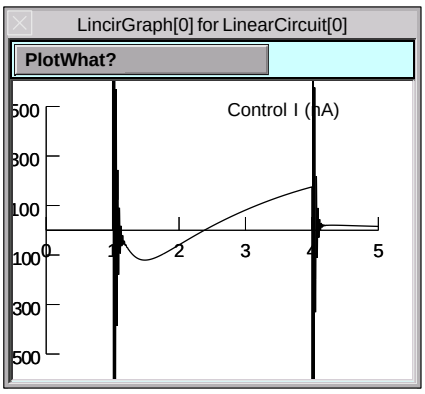


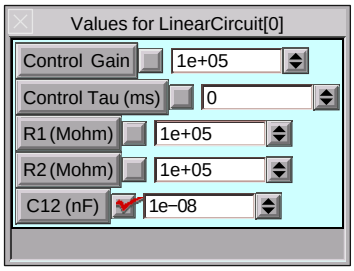


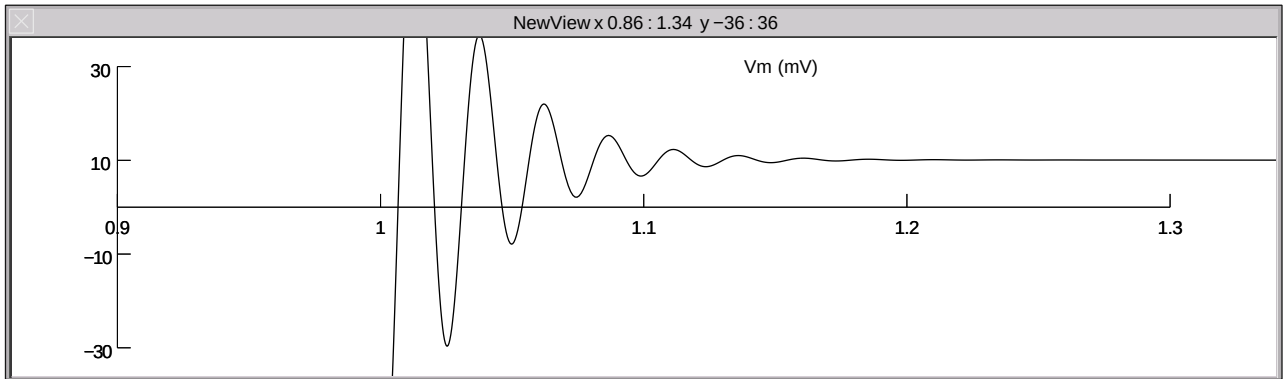












Fixed dt (analytical)

```
STATE { O }

BREAKPOINT {
  SOLVE state
  ik = gbar*o*(v-ek)
}

LOCAL fac

PROCEDURE state() {
  rate(v)
  o = o + fac*(oinf-o)
}

PROCEDURE rate(v(mV)) {
  LOCAL a
  a = alp(v)
  tau = 1/(a + bet(v))
  oinf = a*tau
  fac = (1 - exp(-dt/tau))
}
```

Dynamics specified by ODE

```
STATE { O }

BREAKPOINT {
  SOLVE state METHOD cnexp
  ik = gbar*o*(v-ek)
}

DERIVATIVE state {
  rate(v)
  o' = (oinf-o)/tau
}

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

Abrupt parameter change

```
BREAKPOINT {
  if (t >= del) {
    i = f(t-del)
  } else {
    i = 0
  }
}
```

at_time(del)

***** deprecated *****

Fixed dt only

```
BREAKPOINT {
  if (t>=del) {
    i = f(t-del)
  } else {
    i = 0
  }
}
```

Better: self-event!

```
INITIAL {
  on = 0
  net_send(del, 1)
}
BREAKPOINT {
  if (on == 1) {
    i = f(t-del)
  } else {
    i = 0
  }
}
NET_RECEIVE(w) {
  if (flag == 1) {
    on = 1
  }
}
```

Using hoc to control what happens in a simulation

At time t1 do X

Old: change std run system

```
proc advance() {
  fadvance()
  if (t == t1) { p() }
}
```

Better: use events

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

If Y happens do X

Old: change std run system

```
proc advance() {
  fadvance()
  if (soma.v(0.5) > 10) { p() }
}
```

Better: use events

```
soma {nc = new NetCon(&v(0.5), nil)
nc.threshold = 10
nc.record("p()")
```

```
proc p() {
  . . . statements . . .
  // if p changed ANY parameters or states
  // then be sure to
  // cvode.re_init()
}
```


Using Python to control what happens in a simulation

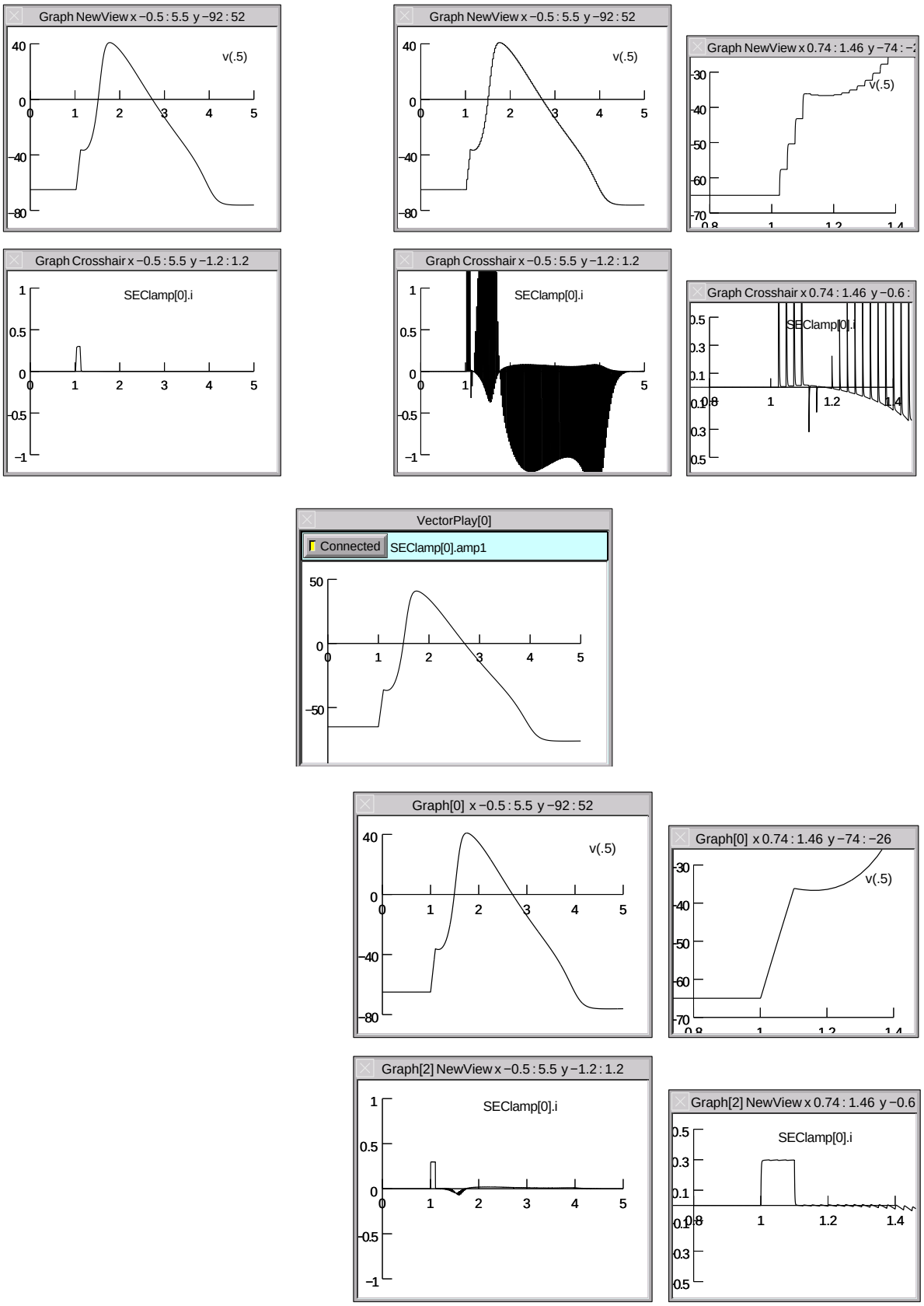
by means of events!

At time t1 do X

```
fih = h.FInitializeHandler(0,ev)
def ev():
  h.cvode.event(t1, p)
}
```

If Y happens do X

```
nc = h.NetCon(soma(0.5)._ref_v, None, sec=soma)
nc.threshold = 10
nc.record(p)

    def p:
      . . . statements . . .
      # if p changed ANY parameters or states
      # then be sure to
      # h.cvode.re_init()
    }
```

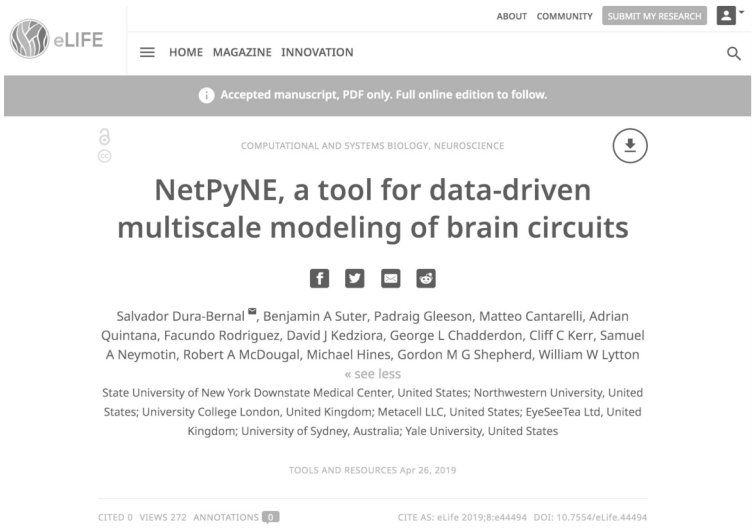



soma vvec.play(&SEClamp[0].amp1, tvec, 1)

NetPyNE

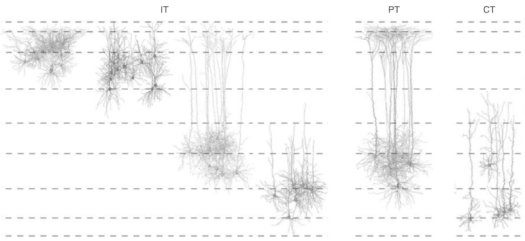
- Network Python for NEURON simulations
- Declarative high-level descriptors which are translated into NEURON
- See paper: elifesciences.org/articles/44494
- netpyne.org has doc, tutorials, fora

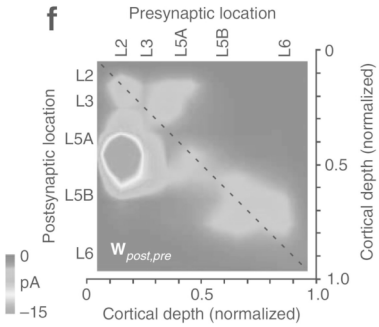
NetPyNE tool: Publication in eLife



Features

Facilitate incorporation of experimental data at multiple scales





Features

Separate model parameters from standardized implementation

Standardize format – easy to read, interpret, edit, share, reproduce, etc

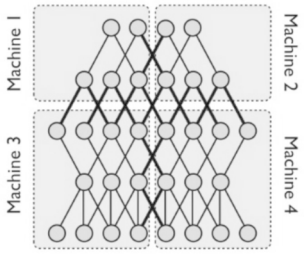
```
popParams['EXC_L2'] = {
 'cellType': 'PYR',
 'yRange':    [100, 400],
 'numCells':  50}
```

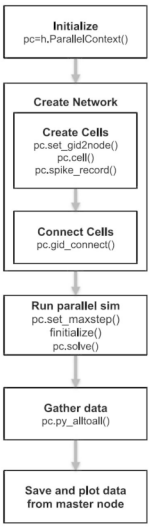
```
for gid in range(pop.numCells):
      cell = sim.Cell()
      cell.y = numpy.random(100,400)
      cell.type = 'PYR'
      pc.cell(gid, h.NetCon(v_soma, threshold))
```

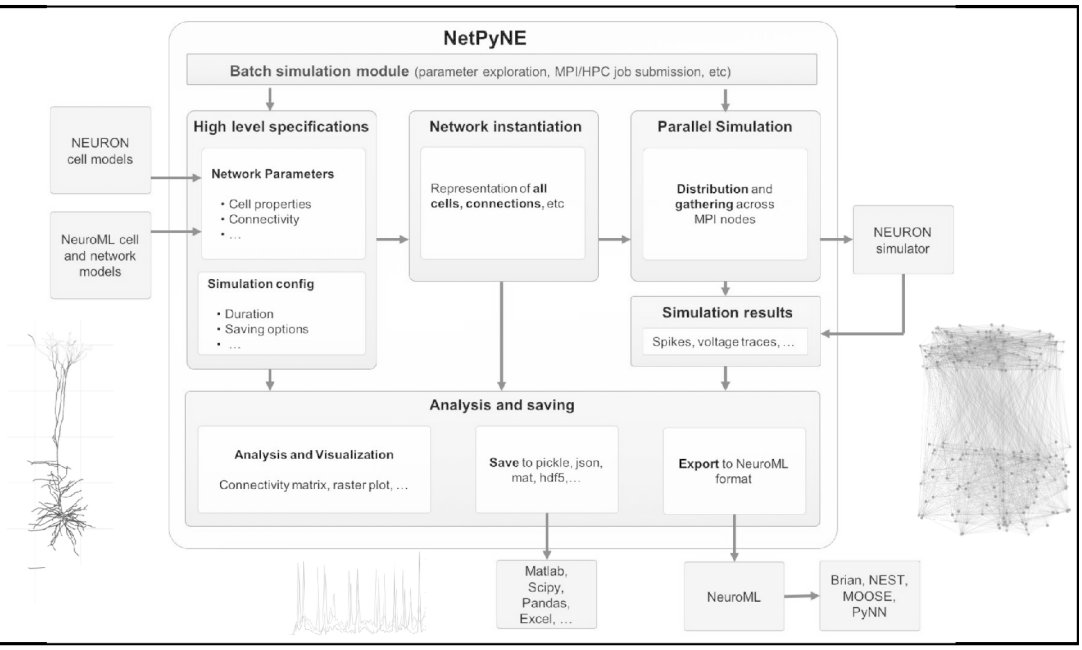
Models have very different implementations
(arbitrary functions, variables, file names etc.)

Features

- Facilitate model parallelization (reproducible)
- Batch parameter exploration/optimization





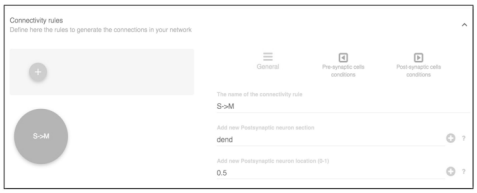


High level specifications

NetPyNE GUI

NetPyNE

NEURON




```
## Cell connectivity rules
netParams.connParams['S->M'] = {
        'preConds': {'pop': 'S'},
        'postConds': {'pop': 'M'},
        'probability': 0.5,
        'weight': 0.01,
        'delay': 5,
        'synMech': 'exc'}
```

```
# add exc connection
postSyn1 = h.ExpSyn(postCell.dend(0.5))
postSyn1.tau = 2
postSyn1.e = -90

pre1Con = h.NetCon(preCell1.soma(0.5)._ref_v,
                   postSyn1,
                   sec=preCell1.soma)
pre1Con.delay = 1
pre1Con.weight[0] = 0.001
pre1Con.threshold = 0
```

High level specifications

A **standardized, declarative** Python format (JSON-like, lists and dicts) to define:

Populations: cell type, number of neurons or density, spatial extent, ...

Cell properties: Morphology, biophysics, molecular processes …

Synaptic mechanisms: Time constants, reversal potential, …

Stimulation: Spike generators, current clamps, spatiotemporal patterns, …

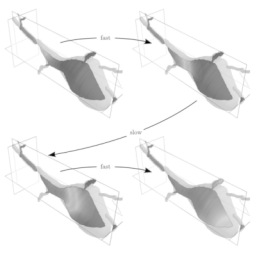
Connectivity rules: conditions of pre- and post-synaptic cells, functions, …

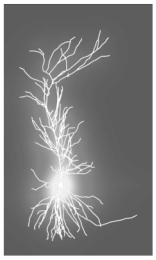
Simulation configuration: duration, saving and analysis, visualization, ...

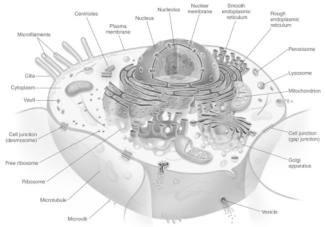
High level specifications

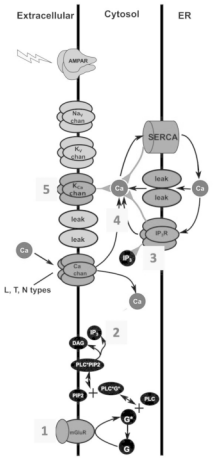
Molecular reaction-diffusion (RxD)

- Intra- and extracellular **diffusion** of ions, proteins (eg, calcium, potassium, IP3, ...)
- Cell internal structures/**organelles** (eg, endoplasmic reticulum, mitochondria,...)
- Molecular **processes** (eg, phosphorylation, buffering, 2nd messenger cascades,...)
- **Interaction** with cell and network scales (eg, firing, plasticity, …)





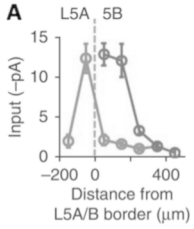




High level specifications

Connectivity

- **Flexible connectivity rules** based on pre- and post-synaptic cell properties (eg, type or location).
- Connectivity **functions** available: probabilistic, convergent, divergent, custom, ...
- Parameters (eg, probability, weight, delay) as a **function of pre/post-synaptic spatial properties**, eg, delays or probability that depend on distance between cells or cortical depth.
- Easily add synapses with **learning** mechanisms (STDP and RL) and **gap junctions.**

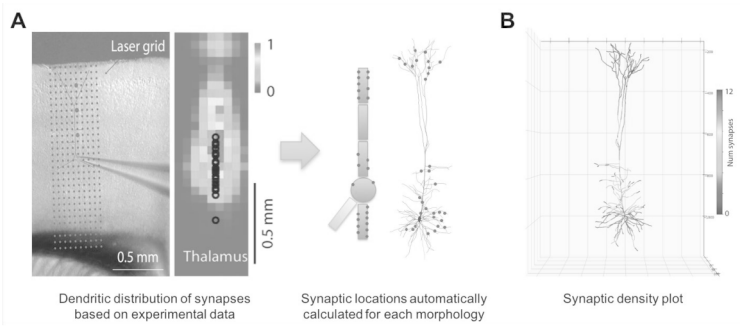


NetPyNE facilitates building models based on experimental data

High level specifications

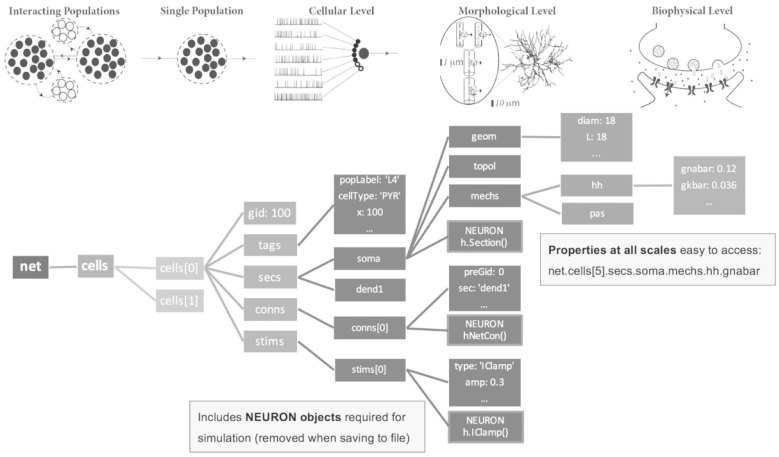
Connectivity

- Specify dendritic **distribution of synapses** as 1D or 2D density map, or based on distance from section
- Synaptic distribution automatically **adapted to morphology** of each cell model



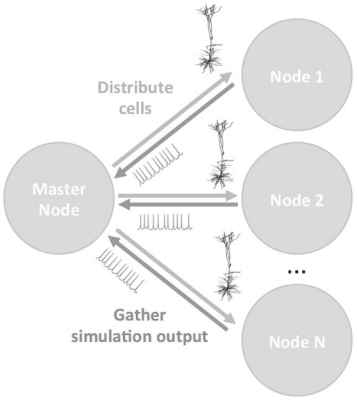
Network Instantiation

Network instance as **standardized hierarchical** Python structure (JSON-like, lists and dicts)

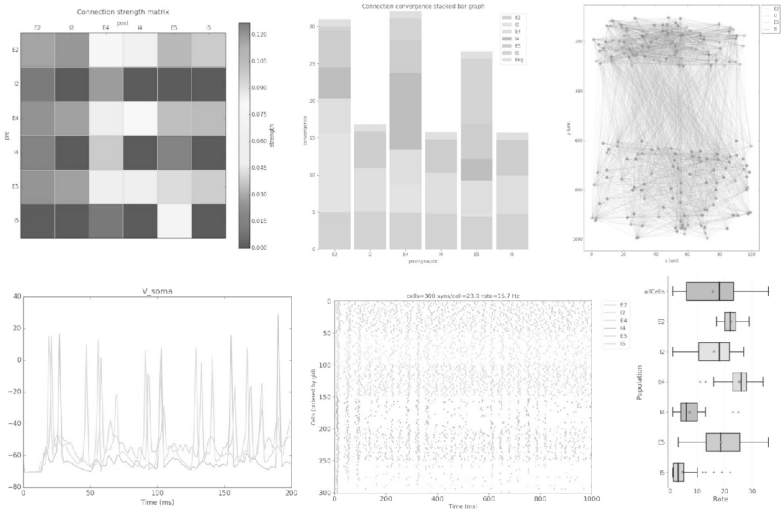


Parallel Simulation

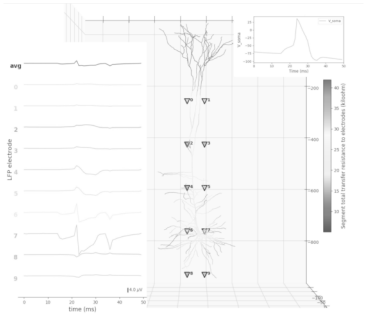
- Set up for MPI **parallel simulation** across multiple nodes (via NEURON simulator).
- Takes care of balanced **distribution** of cells and **gathering** of simulation output from nodes.



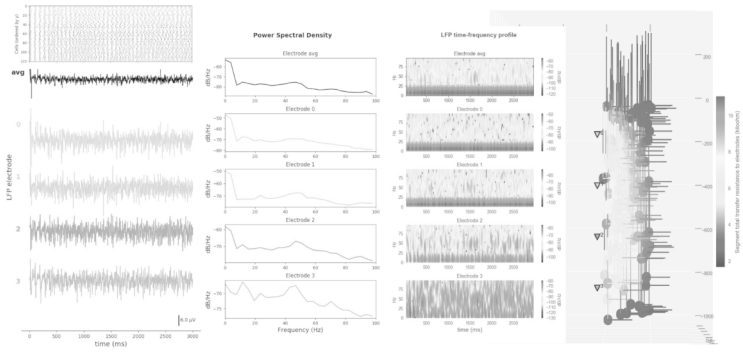
Analysis



Local Field Potentials



Single cell

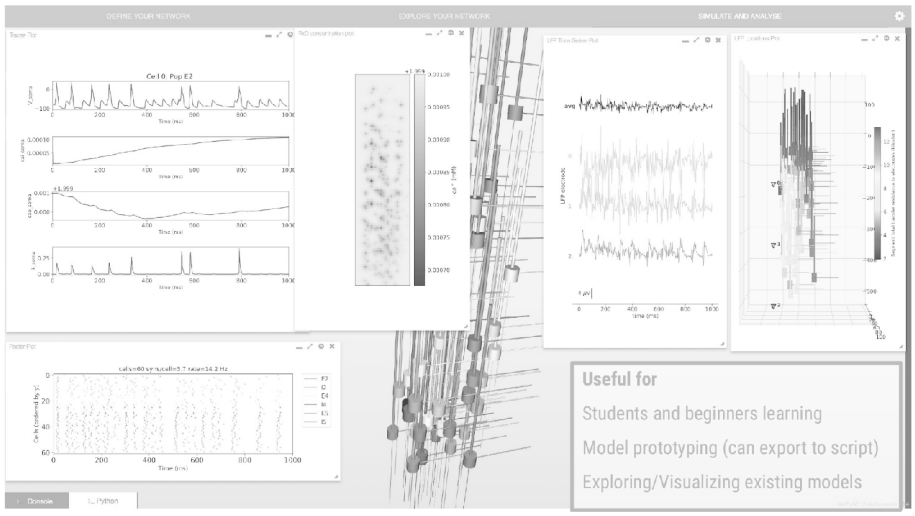


Network

Data Saving and Exporting

- **Save and load** high-level specifications, network instance, simulation config, simulation results.
- **Multiple formats** supported: pickle, JSON and Matlab (CSV and HDF5 in progress)
- **Export/import** network instance to/from **NeuroML** and **SONATA**

Development, Simulation, Analysis via GUI

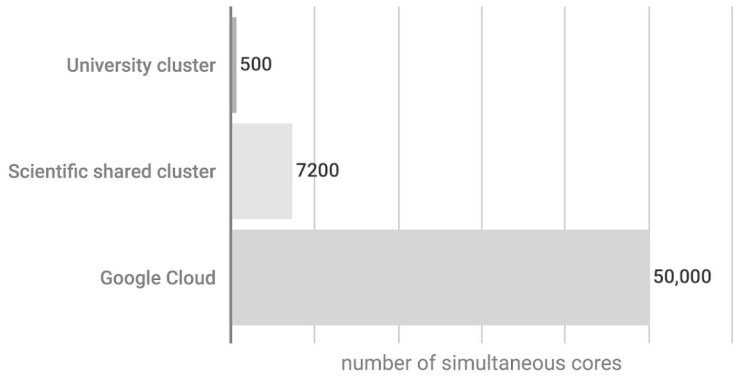


Batch Parallel Simulation

- **Easy specification** of parameters and range of values to explore in batch simulations (evolutionary + grid search)
- **Pre-defined, configurable** setups to automatically **submit jobs** in multicore machines (Bulletin board) or supercomputers (SLURM or PBS Torque)

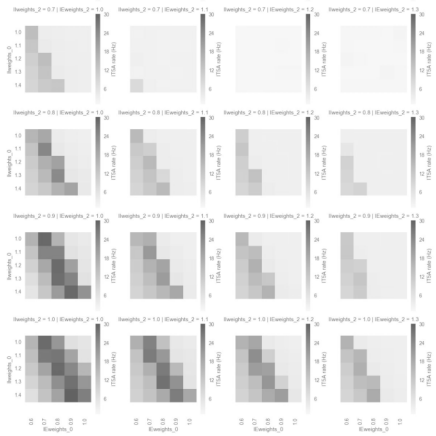
SDSC SAN DIEGO SUPERCOMPUTER CENTER

Batch Parallel Simulation

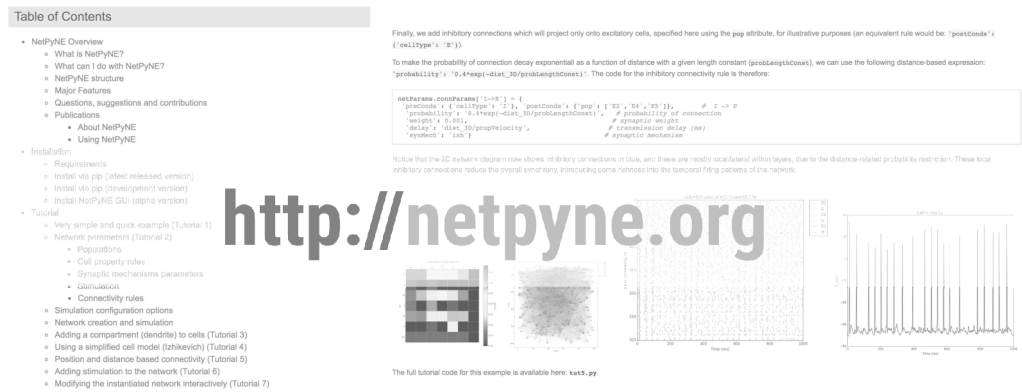


Batch Parallel Simulation

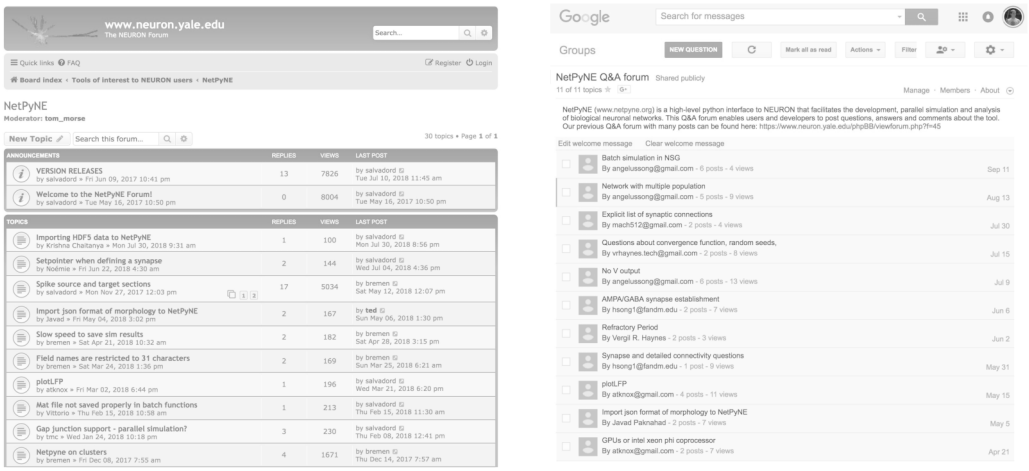
Analysis and **visualization** of multidimensional batch simulation results.



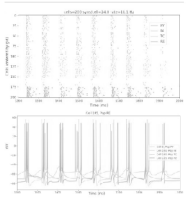
Documentation and Tutorials



Q&A and Forums



Existing NetPyNE Models



- Traub **thalamocortical** network (P. Gleeson, UCL / S. Crook, Arizona)
- Hippocampus **CA3** (B. Tessler, SUNY DMC)
- **Spinal cord** circuits (V. Caggiano, IBM Watson)
- **Striatal** microcircuits (Hanbing/Christina Weaver, Franklin and Marshall College)
- **V1** network (Vinicius/Antonio Roque, Sao Paulo University)
- **Cerebellum** (Sergio Solinas/Stefano Masoli, University of Pavia)
- **Dentate Gyrus** (F. Rodriguez, SUNY DMC)
- **Ischemia** in cortical network (Alex Seidenstein, SUNY DMC)
- **TMS/tDCS** network (Aman Aberra, Duke University)
- **LFP** oscillations (Christian Fink, Ohio Wesleyan)
- **Dendritic** computations (Birgit Kriener, Oslo)
- Thalamocortical **epilepsy** network (Andrew Knox, Cincinatti Hospital)

Full list of 53 models (many under development): www.netpyne.org/models

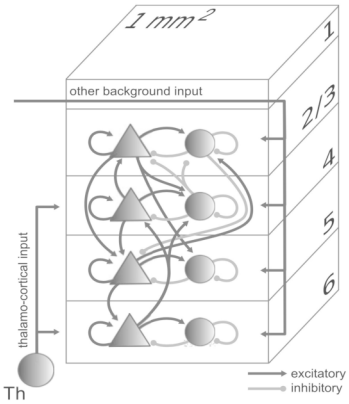
Potjan's and Diesmann model

~80k neurons (point model in NMODL)

~300M synapses

Converted to NetPyNE

Executed on Google Cloud



Potjan's and Diesmann model

NEST NetPyNE/NEURON NEST NetPyNE/NEURON

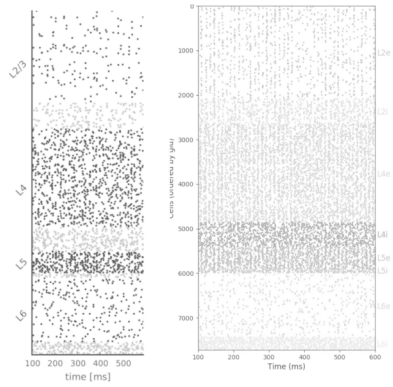


Figure 2: Raster plots network models scaled down to 100% of the original size (with around 80,000 neurons). NEST model on the left and NetPyNE model on the right.

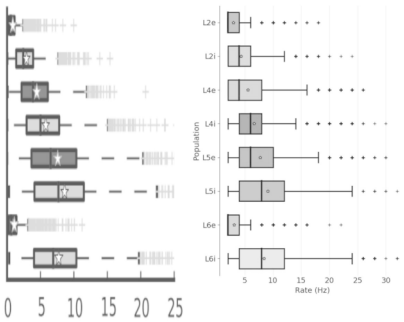
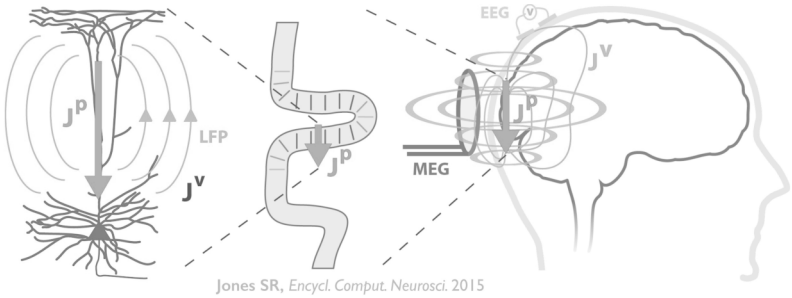


Figure 3: Statistics of spiking activity of all 8 neural populations for rescaling of the PD model to 100% of its original size (around 80,000 neurons). NEST model on the left and NetPyNe model on the right.

Human Neocortical Neurosolver

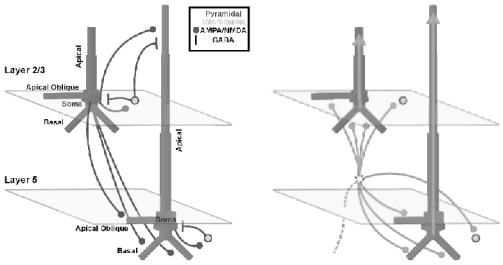
- Stephanie Jones (Brown), PI of NIH BRAIN R01
- Tool to reproduce/understand EEG/MEG signals using biophysical circuit model

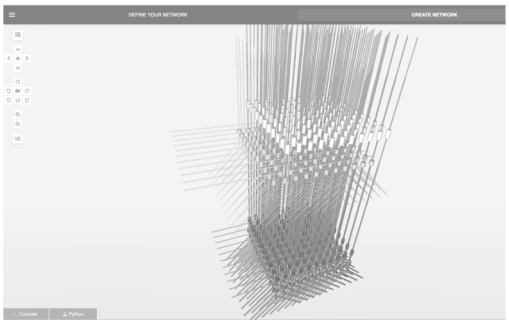


Jones SR, *Encycl. Comput. Neurosci.* 2015

Human Neocortical Neurosolver

- Converted circuit model to NetPyNE
- Facilitate scaling, extension and customization



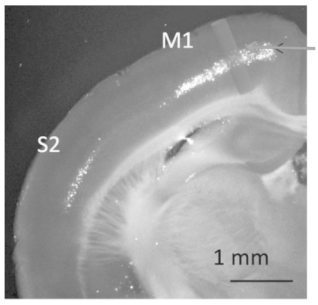


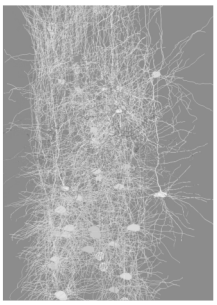
Mouse M1 microcircuits model

Full scale cylinder of **300 µm** (diameter) x **1350 µm** (cortical depth)

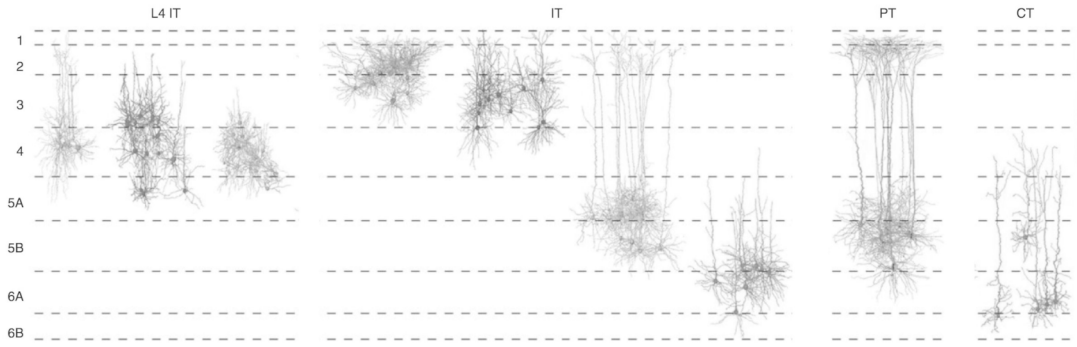
~10,000 neurons of 5 classes distributed in 15 populations

~**30M** synapses

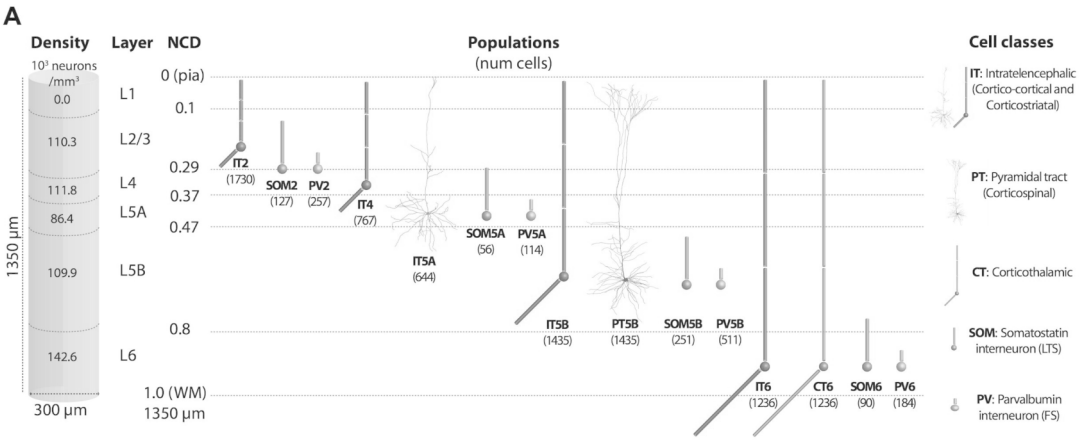




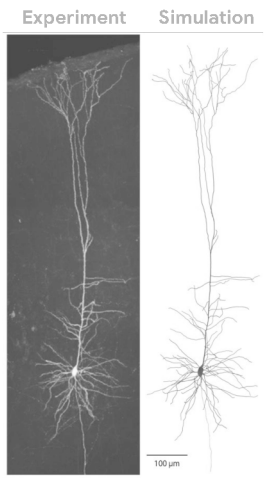
Cortical circuits: experimental data

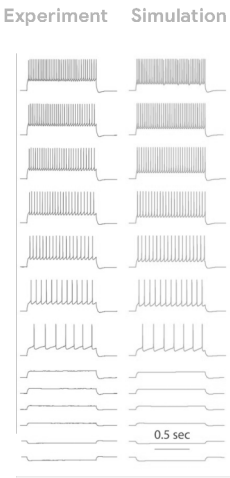


M1 model: cell types and populations



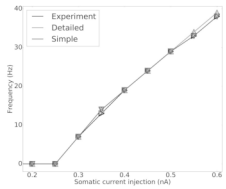
M1 model: single neuron models

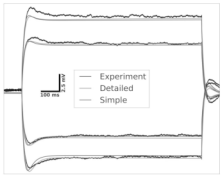




Reproduce morphology and physiology of real neurons

Ionic channel parameters optimized to match data (Na, Kdr, Ka, Kd, HCN, CaL, CaN, KCa)

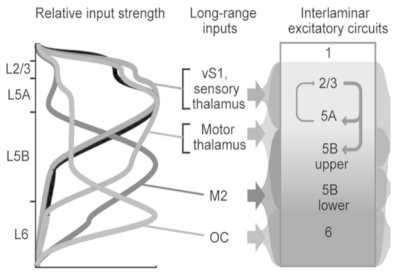




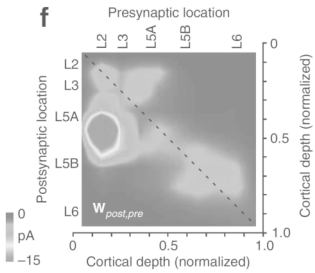
M1 model: connectivity

Incorporates connectivity data at multiple scales

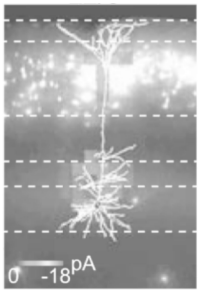
Long-range inputs



Local microcircuits



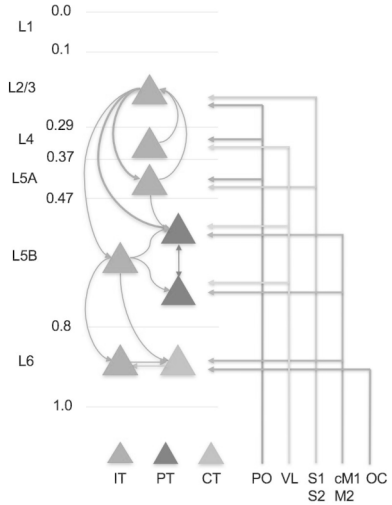
Dendritic inputs



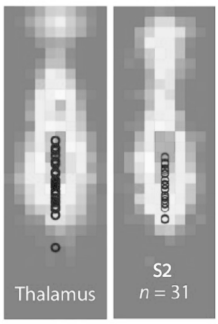
M1 model: connectivity

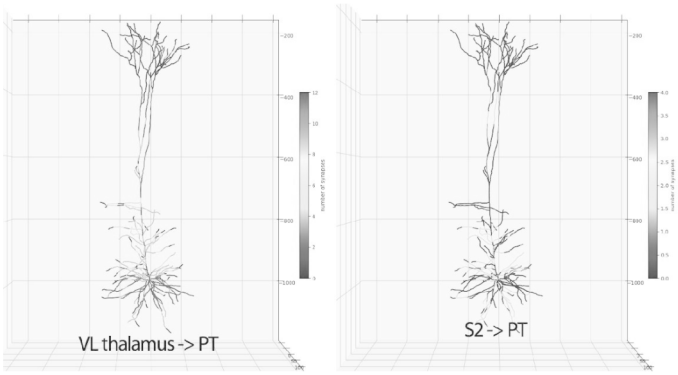
· Main long-range and local excitatory connections

· Depends on cell class and cortical depth (100 um resolution)



M1 model: connectivity

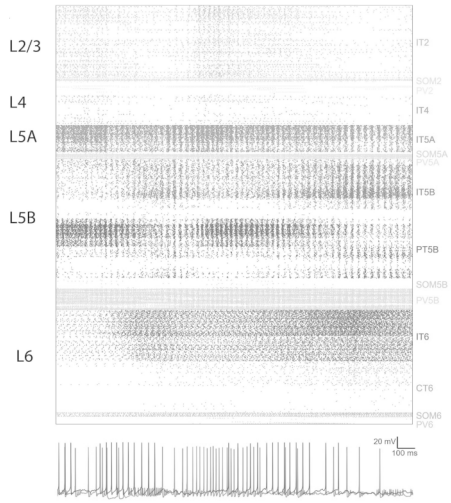




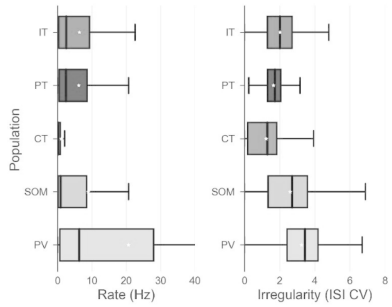
Experimental data

Synaptic locations/density across neuron (simulation)

M1 model: firing properties

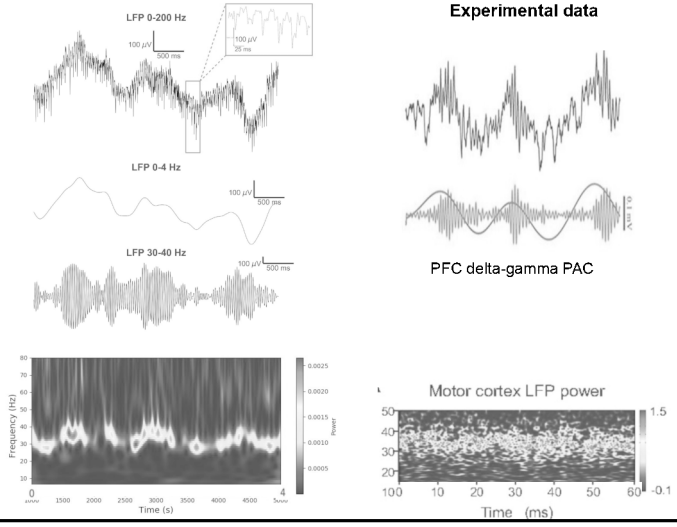


- Depend on cell class, layer and sublaminar location
- Match cortical data



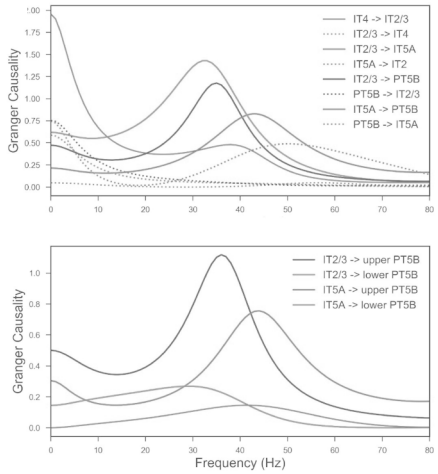
M1 model: LFP oscillations

- Delta and beta/gamma range
- Emerged without rhythmic inputs
- Phase-amplitude coupling
- Role in movement



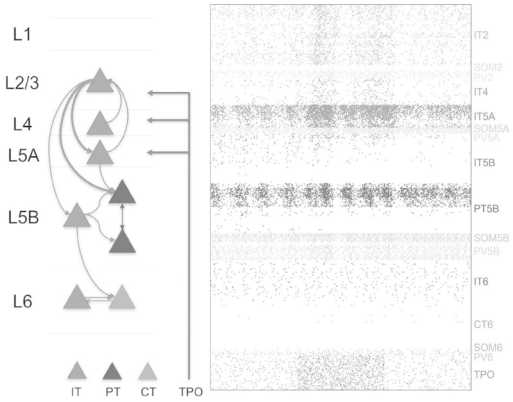
M1 model: information flow

- Granger Causality analysis
- IT → PT but not opposite direction
- Peak in beta/gamma

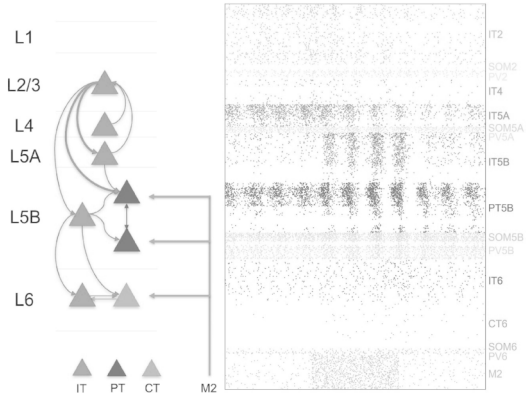


M1 model: pathway dynamics

Sensory-related inputs (thalamus)

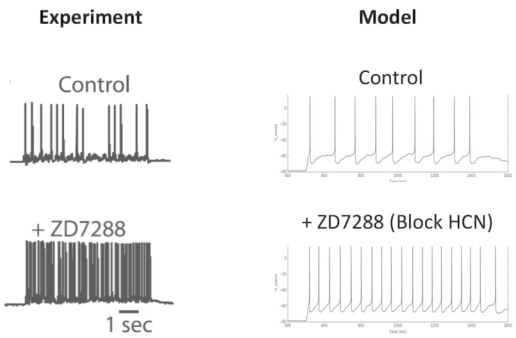


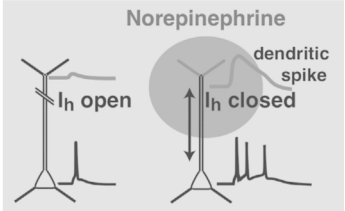
Motor-related inputs (M2)



M1 model: multiscale interactions

Evaluate molecular/pharmacological effects:
blocking HCN channels (Ih) reduces PT output





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() {
  cvode_simgraph()
  realtime=0  // "run time" in seconds
  setdt()
  init()
  initPlot()
}
```


`init()`

Most customizations are made here

```
proc init() {
  finitialize(v_init)
  // Extra initialization should normally go here.
  // If you change any states or parameters after
  // an finitialize, then you should complete
  // the initialization with
  /*
  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` or `Python`

```
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?

hoc: Use a custom `init()` procedure
loaded after the standard library
so it won't be overwritten.

Python: Use an FInitializeHandler (much cleaner).

class **FInitializeHandler**

Syntax:

```
fih = h.FInitializeHandler(py_callable)
fih = h.FInitializeHandler(type, py_callable)
```

Description:

Install an initialization handler statement to be called during a call to `finitialize()`. The default type is 1.

. . .

Type 1 handlers are called after the mechanism INITIAL blocks. This is the best place to change state values.

Initializing to steady state

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

```
def ssinit():
  # these params depend on your model
  T0 = -1e3 # how far back to jump
  DUR = 1e2 # time allowed to reach steady state
  DT = 0.025 # to restore h.dt if simulation uses var dt
  #
  h.t = T0 # jump back
  # if cvode is on, turn it off
  tmp = h.cvode.active()
  if (tmp!=0):
    h.cvode.active(0)
    h.dt = DT # prevent crazy large h.dt
  while (h.t < T0 + DUR): h.fadvance()
  h.t = 0 # return to the present
  # restore cvode if necessary
  if (tmp!=0): h.cvode.active(1)
  if (h.cvode.active()):
    h.cvode.re_init()
  else:
    h.fcurrent()
  h.frecord_init()
```


Initializing to a desired state

Especially useful for oscillating or chaotic models.

Run a "warmup simulation," then save the states

```
svstate = h.SaveState()
svstate.save()
```

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

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

To read from a file

```
svstate = h.SaveState()
f = h.File("states.dat")
svstate.fread(f)
```

Then use an FInitializeHandler to restore the saved states.

Initializing to a desired state continued

Using an FInitializeHandler to restore the saved states.

```
def restate():
  svstate.restore()
  h.t = 0 // t is one of the "states"
  if (h.cvode.active()):
    h.cvode.re_init()
  else:
    h.fcurrent()
  frecord_init()

fih = h.FInitializeHandler(restate)
```


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

```
h.finitialize(h.v_init) # set all v to v_init

def fixrp():
  etmp = (soma.ina+soma.ik+soma.gl_hh*h.v_init)/soma.gl_hh
  soma.el_hh = etmp
  if (h.cvode.active()):
    cvode.re_init()
  else:
    h.fcurrent()
  h.frecord_init()

fih = h.FInitializeHandler(rp1)
```

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 FInitializeHandler.

FInitializeHandler to use with constant current mechanism:

```
soma.insert('constant') # make sure constant exists
h.finitialize(h.v_init) # set all v to v_init

def rp2():
  soma.ic_constant = -(soma.ina + soma.ik + soma.il_hh)
  if (h.cvode.active()):
    h.cvode.re_init()
  else:
    h.fcurrent()
  h.frecord_init()

fih = h.FInitializeHandler(rp2)
```


HOC

for reading knowledge

Robert A. McDougal

Yale School of Medicine

HOC in History

- HOC was introduced in Kernighan and Pike (1984) to demonstrate using Yacc.
- HOC = Higher Order Calculator
- oc = object-oriented extension
- HOC was NEURON's original programming language.
- Hundreds of NEURON models in HOC from before (and after) Python support was added are available on ModelDB.

Objective: Be able to read HOC code, so that we can understand what it does and use it from Python.

Accessing a HOC interpreter

NEURON's HOC interpreter may be accessed by typing `nrniv` or double clicking the corresponding icon:

```
Roberts-MBP:~ ramcdougal$ nrniv
NEURON -- VERSION 7.6.1 master (a558837) 2018-08-01
Duke, Yale, and the BlueBrain Project -- Copyright 1984-2018
See http://neuron.yale.edu/neuron/credits

oc>
```

To exit nrniv, press ctrl-D at the prompt or type `quit()`

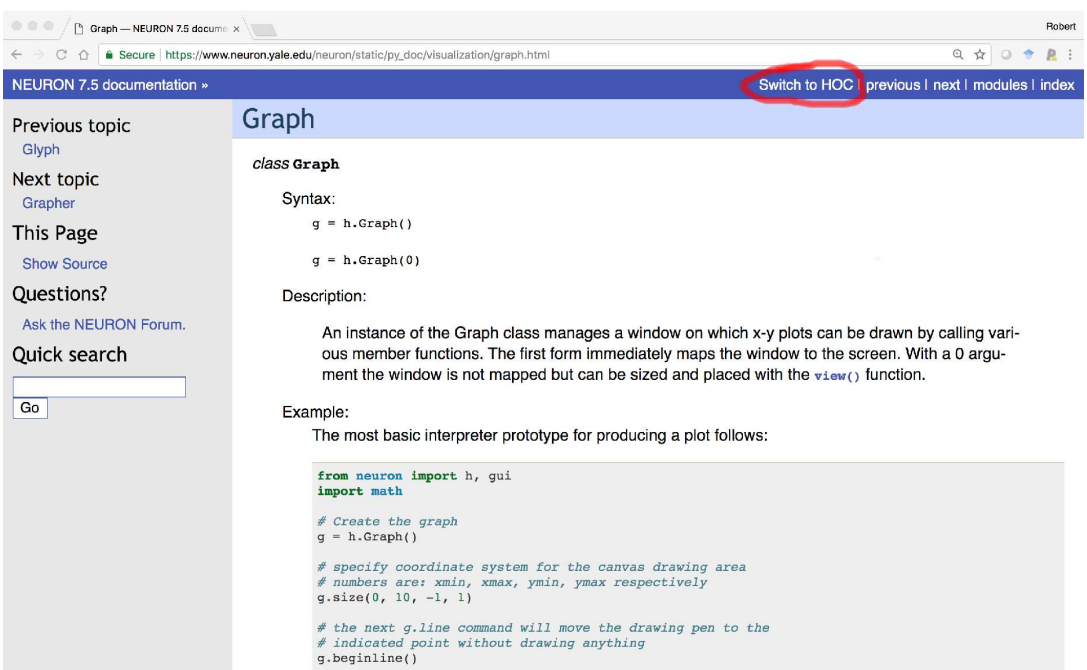
Note: launching `nrniv` does not load the compiled mechanisms automatically. To do that, launch `nrngui` instead.

`nrn_load_dll` can be used to load MOD file mechanisms from nrniv.

`nrniv` and `nrngui` can both take a filename parameter to run the file automatically, e.g. `nrniv my_file.hoc`

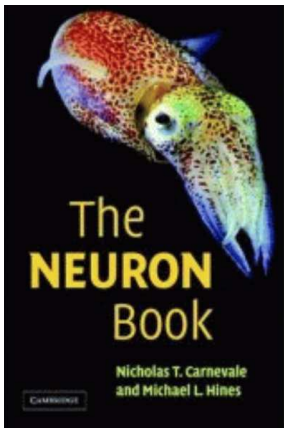
To run an MPI simulation with `nrniv`, use the `-mpi` flag, e.g. `mpiexec -n 4 nrniv -mpi my_file.hoc`

To learn more: Programmer's reference pages also in HOC



To learn more: The NEURON Book and ModelDB

The NEURON Book provides a HOC introduction and all examples are in HOC:



Search ModelDB for specific terms and restrict your searches to HOC files:

```
finitializehandler file:*.hoc
```

Basic HOC syntax

Flow control

Familiar flow control statements are available in HOC:

if

```
if (a == b) {
    print "same"
} else {
    print "different"
}
```

for

```
for i = 1, 5 {
    print i
} // note: both end points are included

for (i = 1; i < 1025, i *= 2) {
    print i
}
```

Flow control

while

```
i = 0
while (i < 7) {
    i = i + 2
    print i
} // prints 2, 4, 6, 8
```


Grouping statements

Unlike Python which uses indentation to indicate grouping, e.g.

```
for i in range(10):
    print(i)
```

HOC uses curly brackets like C++, JavaScript, etc:

```
for(i=0; i<10; i+=1) {
print i
}
```

It's good style to also indent HOC code, but not everyone did. Indentation may also be inconsistent.

In fact, HOC uses context to figure out when an instruction end, so you may run into multiple instructions on one line:

```
for(i=0; i<10; i+=1) {j = i * 2 print j}
```

Operators

Arithmetic operators are the same in HOC and Python:

`+ - * / %`

Comparison operators are the same in HOC and Python:

`< <= == >= >`

Logical operators are not the same:

HOC	Python
&&	and
||	or
!	not

Note that unlike Python, HOC has no explicit concepts of `True` or `False` and uses numbers for these purposes instead, with 0 for False and non-zero for True.

```
oc>print 4 < 2, 2 < 4
0 1
oc>print 4 < 2 || 2 < 4
1
oc>print !(4 < 2)
1
```

Python understands this notation as well, but provides explicits boolean variables.

HOC → Python gotchas: fuzzy comparisons

HOC allows fuzzy comparisons.

The variable `float_epsilon` sets the tolerance for equality.

By default, it is 10^{-11}, which is several orders of magnitude larger than machine epsilon. So numbers that compare equal in HOC may not compare equal in Python.

Example:

```
oc>1 < 1.01
 1
oc>float_epsilon = 0.1
oc>1 < 1.01
 0
oc>1 == 1.01
 1
```

The good news: as of 8/10/18, only one ModelDB model sets `float_epsilon`.

The bad news: even when it is not explicitly set, comparison works differently in HOC and Python.

Data types

HOC uses rigid data types.

Once a variable name has been used to store a given data type, it cannot be used again for a different data type. Doubles (floating point numbers) may be used without explicit declaration:

```
x = 2
```

Strings must be declared before use:

```
strdef s
s = "hello world"  // only double quotes are allowed
```

Objects must also be declared:

```
objref pyobj
pyobj = new PythonObject()
```

HOC does not explicitly have a concept of integers or booleans.

Comments

HOC provides two forms of comments:

// denotes a comment that continues until end of line (same as Python's #):

```
a = 2
// increment a by one
a += 1
```

/* with a matching */ denotes arbitrarily long, arbitrarily located comments

```
a = /* please don't do this but it is valid HOC */ 2
```

There is no direct Python equivalent, but when used as multi-line comments, this is similar to using a multi-line string for commenting in Python:

```
proc solve_three_body_problem() {
    /*
        Analytically solves the three body problem

        Implementation left as an exercise for the reader.
    */
}
```

func and proc

HOC has two types of callables: `func` and `proc`. These correspond to Python `def` that respectively do or do not return a value.

```
proc say_fact() {
    print "The sin of PI / 6 is ", sin(PI / 6)
}

func return_one() {return 1}
```

These are called with parentheses as in Python:

```
oc>say_fact()
The sin of PI / 6 is 0.5
oc>result = return_one()
oc>print result
1
```

Note: HOC has no concept of namespaces. `func` and `proc` are either at the top level or class/template methods; compare `sin` above with Python's `math.sin`.

func and proc: arguments

Values passed to HOC functions and procedures are accessed by 1-indexed position and data type.

Numeric parameters are accessed via e.g. $1, $2, $3, ...

```
func add_things() {
    return $1 + $2
}
print add_things(4, 7) // prints 11
```

String parameters are accessed via e.g. $s1, $s2, $s3, ...

```
proc hello() {
    print "hello ", $s1
}
```

Object parameters are accessed via e.g. $o1, $o2, $o3, ...

Scalar pointers are accessed via e.g. $&1, $&2, $&3, ...

HOC → Python gotchas: variable scoping

In Python, setting a variable assigns to a local scope by default. HOC uses global scope by default instead:

```
oc>a = 2
oc>proc do_a_thing() {
> oc>a = 3
> oc>print a
> oc>}
oc>do_a_thing()
3
oc>print a
3
```


Local variables

Local variables in HOC are explicitly declared using `local` in the *first line* of a `proc` or `func`:

```
oc>print a
3
oc>proc do_another_thing() {local a
> oc>a = 4
> oc>print a
> oc>}
oc>do_another_thing()
4
oc>print a
3
```

HOC → Python gotchas: syntactic flexibility

HOC is relatively forgiving about syntax.

A method that takes no arguments may be called with or without using the parentheses:

```
oc>objref vec
oc>vec = new Vector(100)
oc>vec.size
100
oc>vec.size()
100
```

In Python, however, `vec.size` would be the method while `vec.size()` would be the value returned by the method; i.e. these are two different things.

Thus: when porting code, be careful to add parentheses after all method invocations.

The no-parentheses option does *not* apply to top-level `proc` or `func`, which require the parentheses.

HOC → Python gotchas: syntactic flexibility

In HOC a single = is valid in an `if` statement, but it does assignment. Like Python, == must be used for comparison:

```
oc>a = 1
oc>b = 2
oc>if (a = b) {
> oc>print "a equals b???"
> oc>}
a equals b???
oc>a
 2
```

This is occasionally useful but often indicates a bug.

HOC → Python gotchas: syntactic flexibility

In HOC an array of doubles may be declared as in:

```
double x[10]
```

Values may be read and set using `[]` like for Python lists or numpy arrays:

```
x[3] = 2
```

The 0th item may be accessed using `[0]` or by omitting the indexing entirely:

```
oc>x
 0
oc>x[0] = 4
oc>x
 4
```

This is true even for assignment; *once a variable has been declared an array it is always an array*:

```
oc>x=5
oc>x[0]
 5
```


Using HOC to control NEURON

Most NEURON functions and classes available by dropping the `h.`

```
objref vec, cvode
vec = new Vector(10)
cvode = new CVode()
cvode.active(1)
```

On very rare occasions, some names may be slightly different. The one you are most likely to see is an `IClamp` delay, which in Python is `.delay` but in HOC is `.del`:

```
objref ic
soma ic = new IClamp(0.5)
ic.del = 1
```

The difference here is because `del` is a reserved keyword in Python.

Special syntax for sections

Creating sections with HOC:

```
create soma
create dend[10]
```

Dot notation *may* be used to access section properties:

```
soma.diam = soma.L = 20
```

But typically the *currently accessed section* is used instead, specified either with the `access` statement; e.g.

```
access soma
diam = 20
L = 20
```

or by prefixing a statement of block of statements with the section name, e.g.

```
soma {
    diam = 20
    L = 20
}
```

The curly brace after the section name must occur on the same line as the section name.

Using the currently accessed section

Most of Python's `Section` methods (e.g. `n3d`, `pt3dadd`) appear to HOC as functions that depend on the currently accessed section (they cannot be accessed using dot notation):

```
soma my_n3d = n3d()  // in Python:  my_n3d = soma.n3d()
```

Where Python takes a segment, HOC typically takes a normalized x-value and finds that in the currently accessed segment. e.g.

```
objref rvp
rvp = new RangeVarPlot("v")
soma rvp.begin(0)    // in Python:  rvp.begin(soma(0))
```

There is no direct HOC equivalent of Python's `sec.psection()`. There is a `psection()` that uses the currently accessed section, but that prints some (less) data to the screen, while the Python version returns a data structure that can be examined by a script or by a human.

Connecting sections

`connect` is a keyword in HOC instead of a procedure or method. General form is `connect child, parent`.

```
create soma, dend1, dend2
access soma
connect dend1(0), soma(1)
connect dend2(0), 1      // soma is implicit since current sec
```

Range variables

In Python, range variables are accessed through segments. There is no equivalent of a Python segment object in HOC. Instead, the range variable comes first then the normalized position within the section, where the section is either specified through dot notation or taken as the currently accessed section. e.g.

```
print soma.v(0.5) // in Python:  soma(0.5).v

soma print v(0.5)
```

Range variables that are part of a mechanism are accessed using the variable name, an underscore, and then the mechanism name:

```
soma insert hh       // in Python: soma.insert('hh')
print soma.m_hh(0.5) // in Python:  soma(0.5).hh.m
```


Pointers

A single ampersand (&) before a variable name turns it into a pointer (this is roughly equivalent to the _ref_ prefix for NEURON variables in Python):

```
create soma
access soma
objref v_trace
v_trace = new Vector()
v_trace.record(&v(0.5))  // in Python:
                         // v_trace.record(soma(0.5)._ref_v)
```

Question: how do we know that we're recording the soma's membrane potential in the HOC code?

Iterators

Iterators are like generators in Python, where the HOC `iterator_statement` is equivalent to the Python `yield`.

```
iterator case() {local i
        for i = 2, numarg() {
                $&1 = $i
                iterator_statement
        }
}

x=0
for case (&x, 1,2,4,7,-25) {
        print x
}
```

Coroutines are a related concept.

Looping over sections

To loop over all sections (changing the currently accessed section), use `forall`, e.g.

```
forall {
    print secname()
}
```

To do the same for a `SectionList`, use `forsec`, e.g.

```
forsec my_section_list {
    print secname()
}
```

Regular expressions matching the names of desired sections may be specified instead. e.g. to find all sections whose name begins with apical, use

```
forsec "apical" {
    print secname()
}
```

Sections are not objects in HOC and so they cannot be stored in a List. A special SectionLast class is used instead.

Looping over segment locations

As HOC does not have a segment object, you cannot loop over segments, but you can loop over the normalized segment locations via, e.g.

```
for (x, 0) {print x}
```

If nseg is 5, the above would print 0.1, 0.3, 0.5, 0.7, 0.9 (on separate lines.)

Unfortunately in many HOC codes, where people meant to do the above they instead left out the ,0 and get all of the above values and the end points (0 and 1). In Python that would be equivalent to iterating over `sec.allseg()`, but that is generally not useful and risks setting the end segments twice.

Templates

Templates are like classes in Python and are used to make arbitrary many copies of a cell.

```
begintemplate RE32695
    public nmda, ampa, gabaa, gabab, x, y, z ...
    proc init () { local i,j
        x=$1 y=$2 z=$3 // locations ndend = 59
        create soma, dend[ndend] ...
        soma {
            gabaa = new Exp2Syn (0.5) ...
```

Every section defined inside of a template knows what cell it belongs to; there is no need to explicitly specify the cell in HOC.

Looping over all sections inside of a template method loops over all of that cell's sections.

Example template courtesy of Bill Lytton.

HOC and Python interoperability via NEURON

To load a HOC library from Python, use `h.load_file`:

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

NEURON makes HOC variables, available to Python using the `h.` prefix as if they were NEURON built-ins:

```
from neuron import h
h.finitialize(-65)          # NEURON function; always works
# h.continuerun(10)         # defined in a HOC library;
                            # would give an error here
h.load_file('stdrun.hoc')
h.continuerun(10)           # ok here
```

HOC libraries for NEURON may thus be reused from Python without changes.

Pass in a string to the `h` object to execute it as HOC:

```
>>> from neuron import h
>>> h('''
...     proc hello() {
...         print "hello ", $s1
...     }
... ''')
1
>>> h.hello('world')
hello world
0.0
>>>
```

In particular, strings, numbers, and objects may be passed between Python and HOC.

HOC is not NEURON: data types

Despite the fact that both NEURON and HOC entities may be accessed through the `h` object, when it comes to numeric types, NEURON may return `int`, `bool`, or float; HOC always returns floats, *even if it's just reporting what NEURON did*:

```
>>> h('''
...     func get_vec_size() {return $o1.size()}
...     func identity() {return $1}
... ''')
1
>>> v = h.Vector([1, 2, 12])
>>> type(v.size())
<class 'int'>
>>> h.get_vec_size(v)
3.0
>>> type(v.contains(3))
<class 'bool'>
>>> h.identity(False)
0.0
```

Accessing Python from HOC

Python statements may be run from HOC using `nrnpython`, e.g.

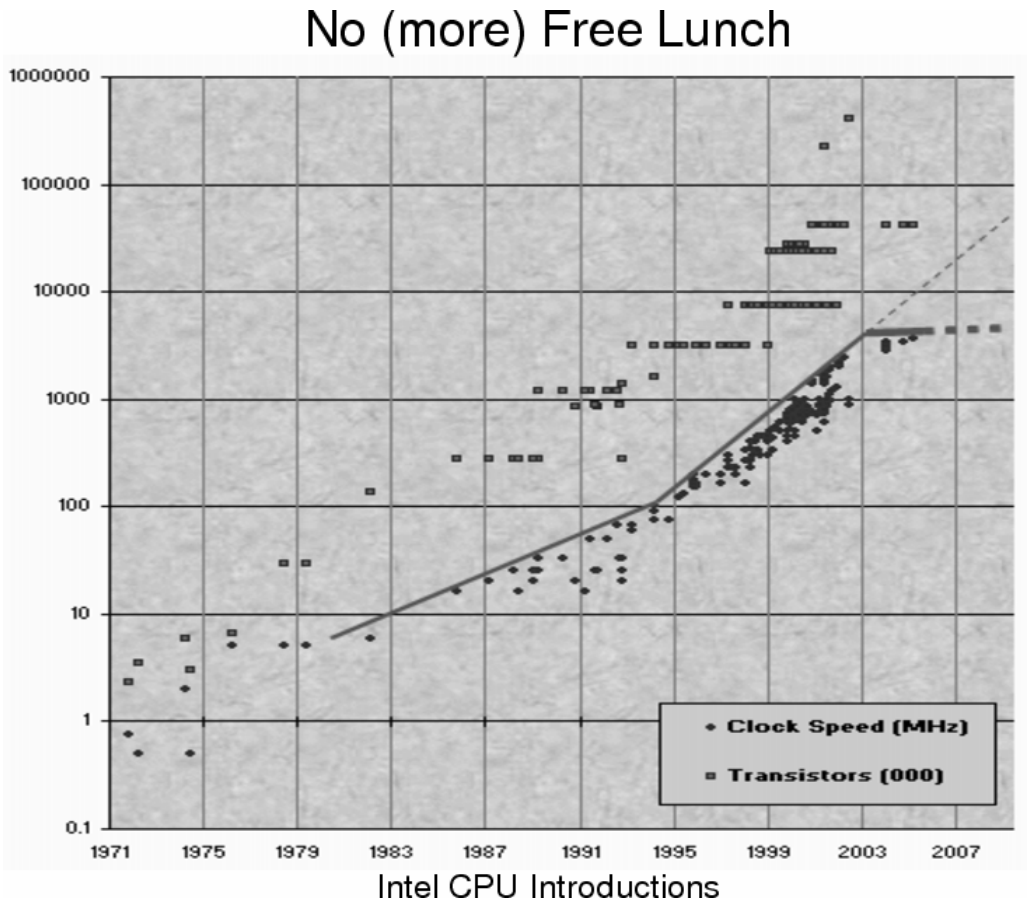
```
nrnpython("import math")
```

Python functions may be called from HOC using a `PythonObject`, e.g.

```
objref pyobj
pyobj = new PythonObject()
print "result is ", pyobj.math.acosh(2)
// prints: result is 1.3169579
```


NEURON + Threads

Simulations on multicore desktops.



Thread style in NEURON

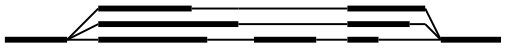
Join

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

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

We never use.

Condition Wait

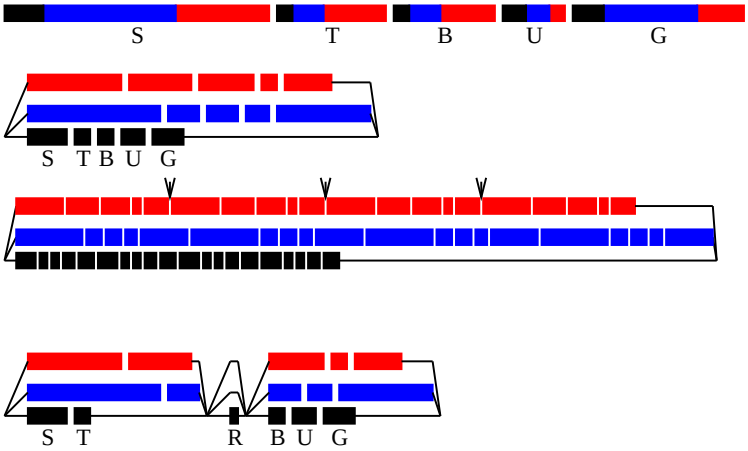
```
multithread_job(run)

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



Reminiscent of MPI

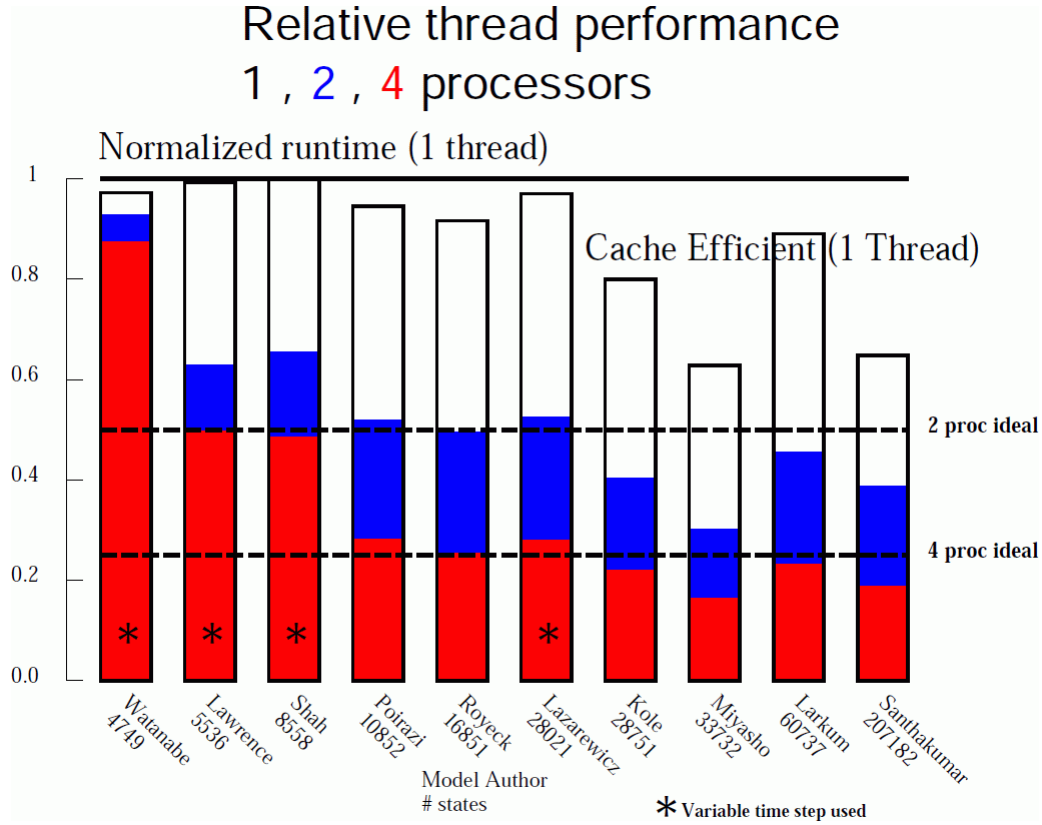
Fixed step: t -> t + dt

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

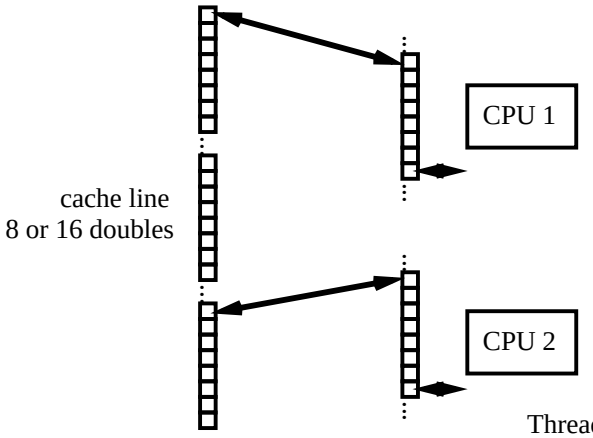


Global var dt y' = f() dy'/dy

27 ||Vector operations



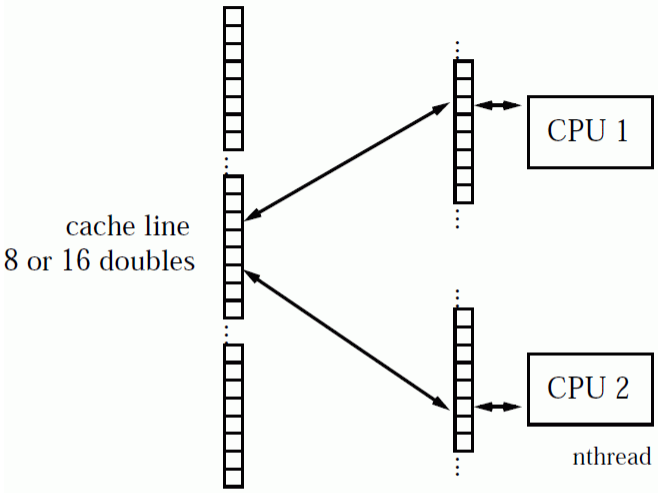
Ideal cache efficiency



10000 passive compartments
4 core 3GHz x86_64

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

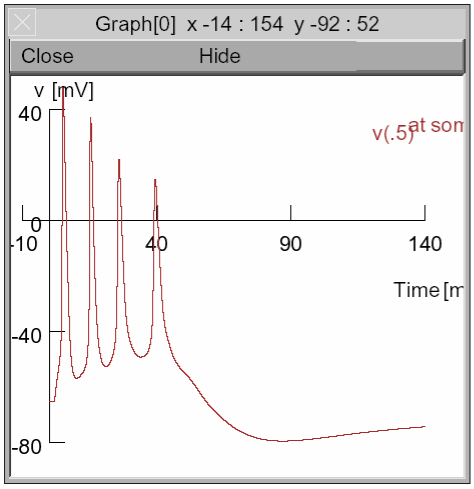
False cache line sharing

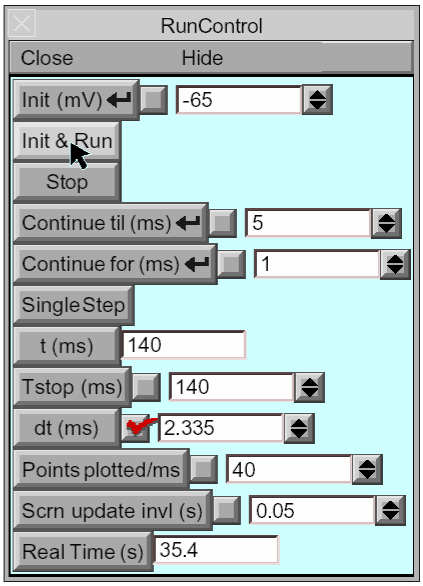


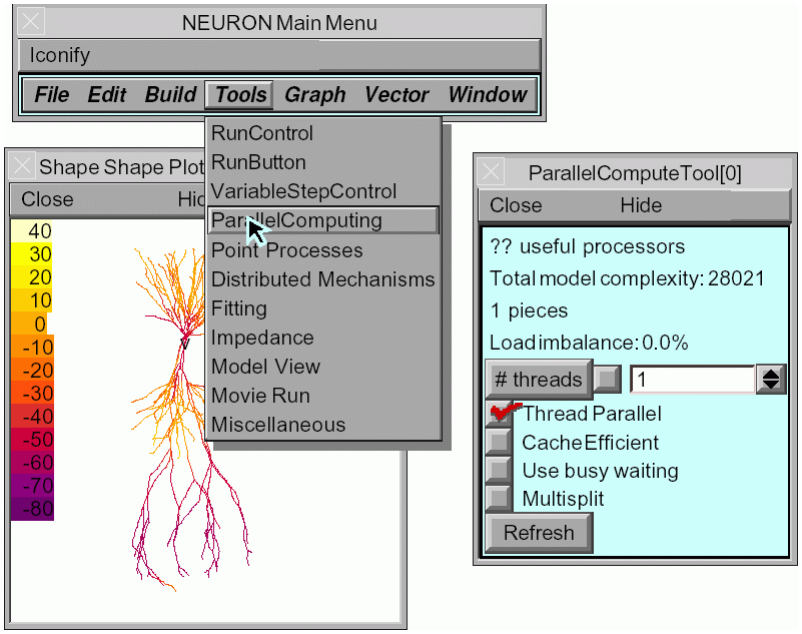

```
for ithread = 1, nthread {
  for i=1, 100million { a[ithread] += i }
}
```

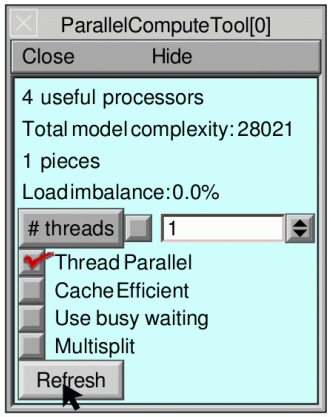
nthread	separation (bytes)	runtime (s)
1	x	0.4
4	4	4.5
4	32	5.0
4	64	0.4
8	4	15.0
8	64	0.7

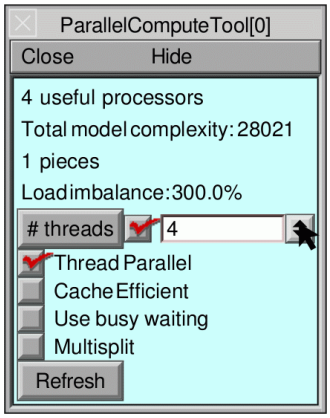
Lazarewicz 2002, CA3 Pyramidal Neuron



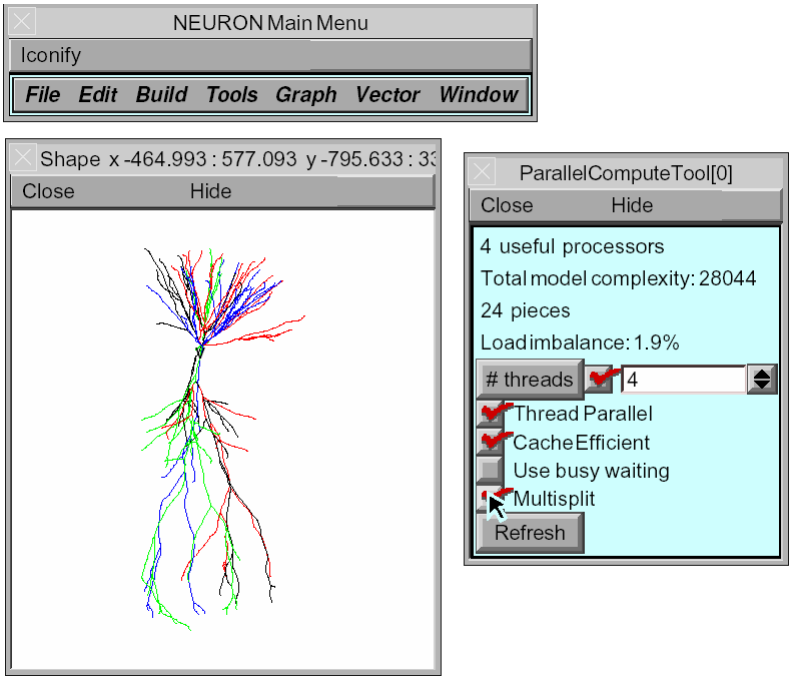


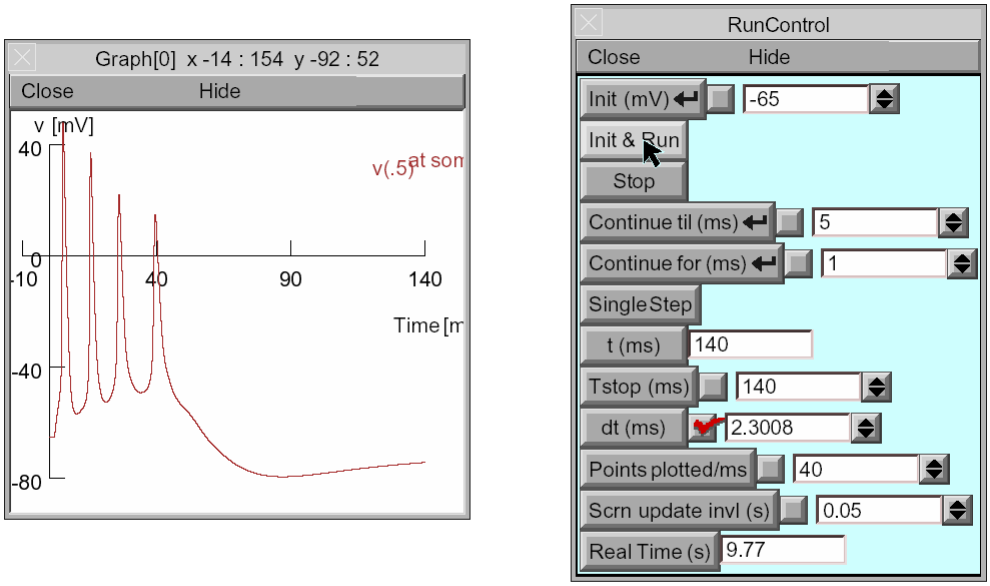







```
oc>nthread walltime (count to 1e8 on each thread)
  1     0.0500002
  2     0.0599999
  4     0.0599999
  8     0.14
```



instead of 35.4s


```
$ mkthreadsafe
NEURON {
        SUFFIX CAlM95
        USEION ca READ cai,cao WRITE ica
        RANGE  gbar,ica
        GLOBAL minf,tau
}
Translating CAlM95.mod into CAlM95.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 CAlM95
                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

load_file("prcellstate.hoc")

```
// serial model
finitialize(−70)
prcellall(0) // constructs cs0.0.1

//switch to 4 threads
finitialize(−70)
prcellall(1) // constructs cs1.0.1
```

```
diff cs*|more
```

notice differences in ik and ica
and in particular

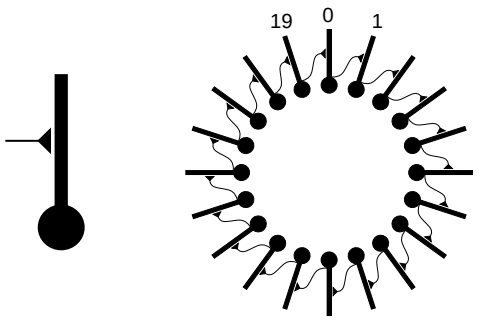
```
595,605c595,605
< 0 594 0.29053584721744774 gk_km(0.0454545)
< 0 595 0.29053584721744774 gk_km(0.136364)
−−−
> 0 594 0 gk_km(0.0454545)
> 0 595 0 gk_km(0.136364)

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)
```


Case study: building a ring network

Physical system: neocortex

Conceptual model: postulated "reverberating loop"

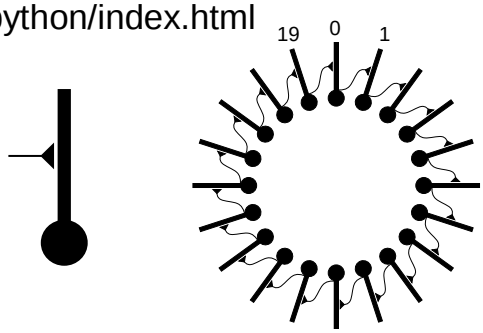
Computational model: ball and stick model cells
connected by spike-triggered
excitatory synaptic transmission



Ring network

hoc users: see Hines & Carnevale, J Neurosci Methods 169:425-55, 2008, PMID 17997162, https://modeldb.yale.edu/96444

Python users: work through NEURON + Python tutorial https://neuron.yale.edu/neuron/static/docs /neuronpython/index.html



Building, Running, and Visualizing Parallel NEURON Models

Robert A. McDougal

Yale School of Medicine

Why use parallel computation?

Four reasons:

- Get the results for a simulation in less real time.
- Run a larger simulation in the same amount of time.
- Run more simulations (e.g. parameter sweeps).
- Run models needing more memory than is available on one machine.

What are the downsides?

Parallel models introduce:

- Greater programming complexity.
- New kinds of bugs.

You have to decide if the time spent parallelizing your model can be recovered.

Other considerations

The 16384 core EPFL IBM BlueGene/P can theoretically do as many calculations in 1 hour at 850 MHz as a 3 GHz desktop computer can do in 6 months.

Building a parallelizable model typically requires little extra effort from building a serial model; converting a serial model to a parallel model is often more difficult.

Three main classes of parallel problems

Parameter sweeps

Running the same (typically fast) simulation 1000s of times with different parameters is an example of an *embarrassingly parallel* problem. NEURON supports this natively with bulletin boards; Calin-Jageman and Katz (2006) developed a screen saver solution.

Distributing networks across processors

Cells can communicate by

- logical spike events with significant axonal, synaptic delay.
- postsynaptic conductance depending continuously on presynaptic voltage.
- gap junctions.

Distributing single cells across processors

The *multisplit* method distributes portions of the tree cable equation across different machines.

A parallel model can fall in 1, 2, or 3 of these classes.

Some parallel philosophy

- A network of neurons is composed of many individual neurons of potentially many cell types. As much as possible, design and debug each cell type separately before building the network.
- A simulation should give the same results regardless of the number of processors used to run it.
- When possible, parameterize your network so you can run a small test first.

For parallel networks: cell classes should have a gid

In addition, it will be convenient to specify morphology in a dedicated method, and add a `__repr__` method to identify the object.

```
from neuron import h, gui
h.load_file('import3d.hoc')

class Pyramidal:
    def __init__(self, gid):
        self._gid = gid
        self._setup_morphology()
    def _setup_morphology(self):
        cell = h.Import3d_SWC_read()
        cell.input('c91662.swc')
        i3d = h.Import3d_GUI(cell, 0)
        i3d.instantiate(self)
    def __repr__(self):
        return 'Pyramidal[%d]' % self._gid
```

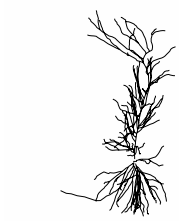
Here, the `gid` should be a globally unique identifying integer. We do not use class variables to generate the integer automatically because: (1) the numbers should not repeat between different processors, and (2) we may wish to recreate a single specific cell instead of the entire network.

Working with multiple cells

Suppose `Pyramidal` is defined as before and we create several copies:

```
mypyrs = [Pyramidal(i) for i in range(10)]
```

We then view these in a shape plot:



Where are the other 9 cells?

Working with multiple cells

To can create a method to reposition a cell and call it from `__init__`:

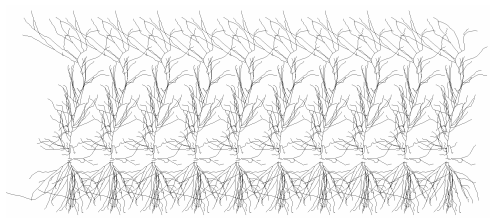
```
class Pyramidal:
  def _shift(self, x, y, z):
    for sec in self.all:
      n = sec.n3d()
      xs = [sec.x3d(i) for i in range(n)]
      ys = [sec.y3d(i) for i in range(n)]
      zs = [sec.z3d(i) for i in range(n)]
      ds = [sec.diam3d(i) for i in range(n)]
      for i, (a, b, c, d) in enumerate(zip(xs, ys, zs, ds)):
        sec.pt3dchange(i, a + x, b + y, c + z, d)
```

```
def __init__(self, gid, x, y, z):
  self._gid = gid
  self._setup_morphology()
  self._shift(x, y, z)

def _setup_morphology(self):
  cell = h.Import3d_SWC_read()
  cell.input('c91662.swc')
  i3d = h.Import3d_GUI(cell, 0)
  i3d.instantiate(self)
```

Now if we create ten, while specifying offsets,

```
mypyrs = [Pyramidal(i, i * 100, 0, 0) for i in range(10)]
```

The PlotShape will show all the cells separately:



Does position matter?

Sometimes.

Position matters with:

- Connections based on proximity of axon to dendrite.
- Connections based on cell-to-cell proximity.
- Extracellular diffusion.
- Communicating about your model to other humans.

Discretize, declare channels, set parameters

```
class Pyramidal:
    def __init__(self, gid):
        self._gid = gid
        self._setup_morphology()
        self._discretize()
        self._add_channels()
    def _setup_morphology(self):
        cell = h.Import3d_SWC_read()
        cell.input('c91662.swc')
        i3d = h.Import3d_GUI(cell, 0)
        i3d.instantiate(self)
    def __repr__(self):
        return 'p[%d]' % self._gid
    def _discretize(self, max_seg_length=20):
        for sec in self.all:
            sec.nseg = 1 + 2 * int(sec.L / max_seg_length)
    def _add_channels(self):
        for sec in self.soma:
            sec.insert('hh')
        for sec in self.all:
            sec.insert('pas')
            for seg in sec:
                seg.pas.g = 0.001
```

Remember: you typically want to have an odd number of segments so there is a node at the middle.

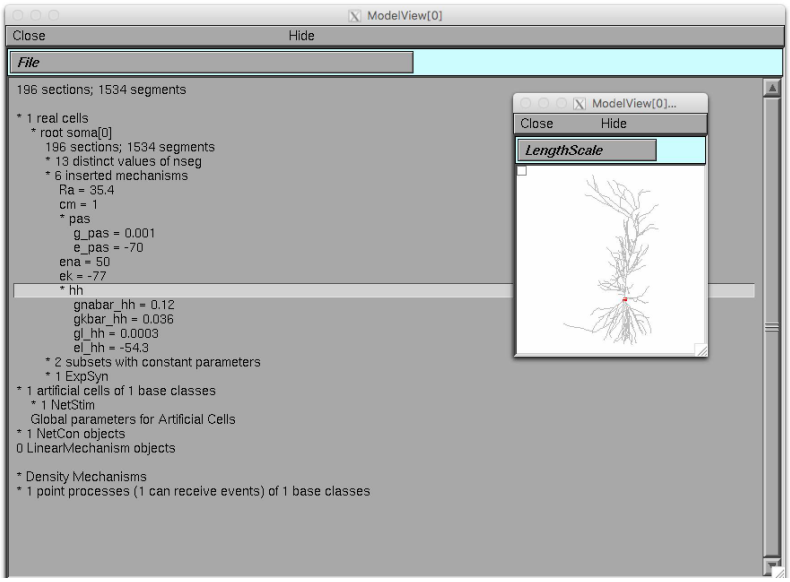
When refining a mesh, multiply by an odd number to preserve old nodes.

```
for sec in self.all:
    sec.nseg *= 3
```

An alternative discretization strategy is to use the d_lambda rule:

```
def _discretize(self):
    h.load_file('stdlib.hoc')
    for sec in self.all:
        sec.nseg = int((sec.L/(0.1*h.lambda_f(100)) + .9)/2.)*2 + 1
```

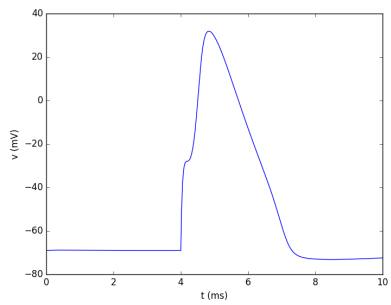
Examine for errors: Tools → ModelView



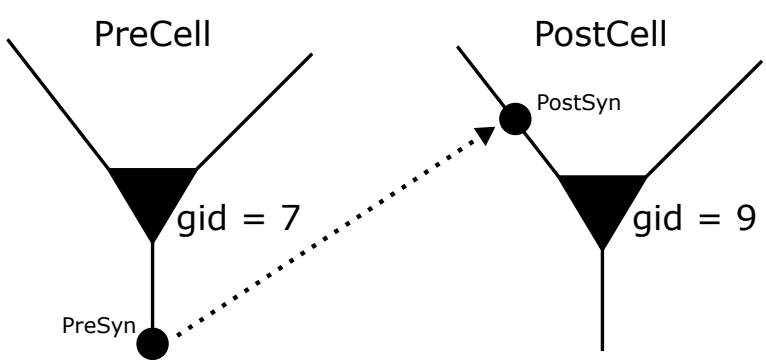
New way to run via h.ParallelContext()

```
from neuron import h
from PyNeuronToolbox import morphology
from matplotlib import pyplot

h.load_file('stdrun.hoc')

# class Pyramidal defined as before

myPyramidal = Pyramidal(0)

postsyn = h.ExpSyn(myPyramidal.dend[0](0.5))
postsyn.e = 0 # reversal potential

stim = h.NetStim()
stim.number = 1
stim.start = 3
ncstim = h.NetCon(stim, postsyn)
ncstim.delay = 1
ncstim.weight[0] = 1

t = h.Vector().record(h._ref_t)
v = h.Vector().record(myPyramidal.soma[0](0.5)._ref_v)
```

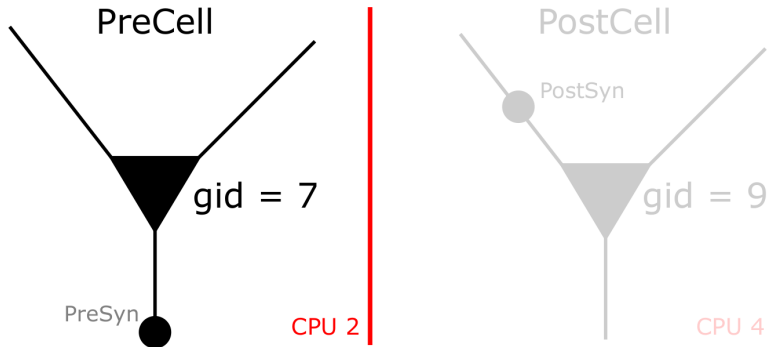
```
pc = h.ParallelContext()
pc.set_maxstep(10)
h.v_init = -69
h.stdinit()
pc.psolve(10)

pyplot.plot(t, v)
pyplot.xlabel('t (ms)')
pyplot.ylabel('v (mV)')
pyplot.show()
```



Building synapses



Configuring the presynaptic connection site



Create cell only where the gid exists:

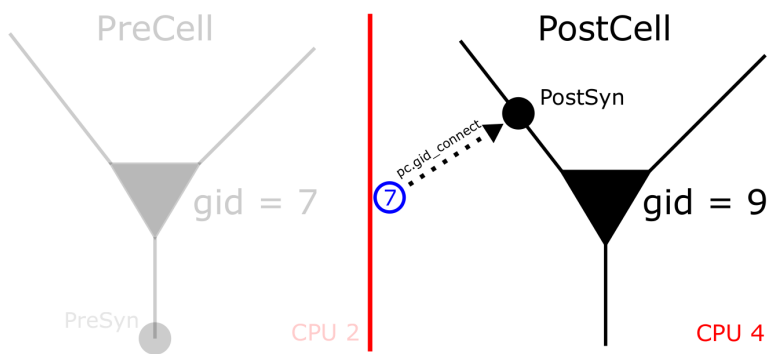
```
if pc.gid_exists(7):
    PreCell = Cell()
```

Associate gid with spike source:

```
nc = h.NetCon(PreSyn, None, sec=presec)
pc.cell(7, nc)
```

PreSyn here is a **pointer**, e.g. `PreCell.soma(0.5)._ref_v`

Configuring the postsynaptic connection site

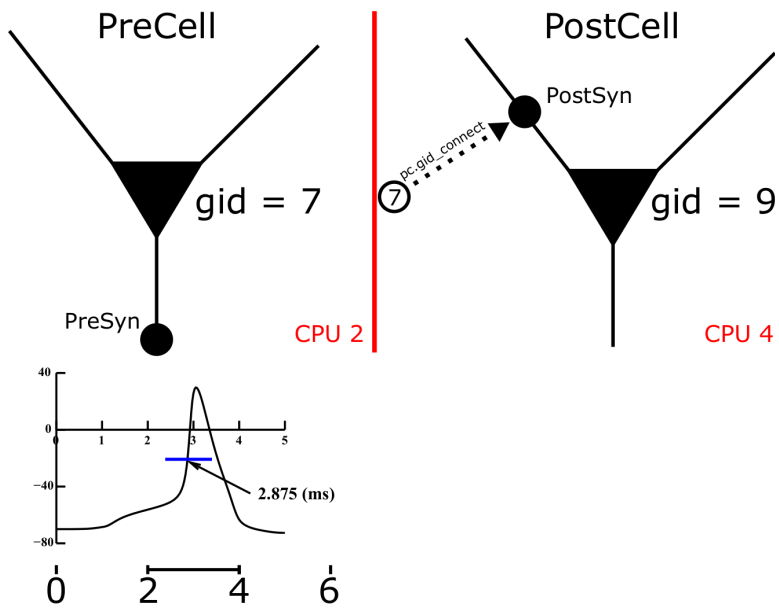


Create NetCon on node where target exists:

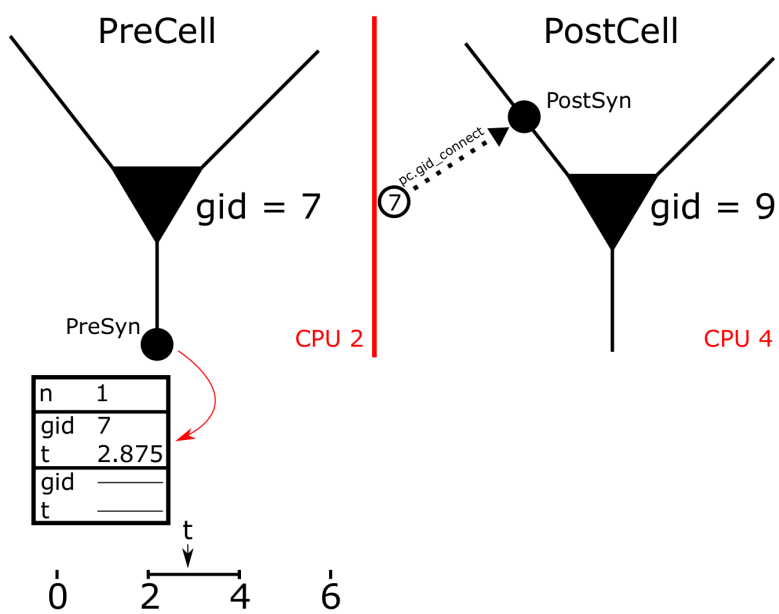
```
nc = pc.gid_connect(7, PostSyn)
```

PostSyn here is a Point Process, e.g. an ExpSyn.

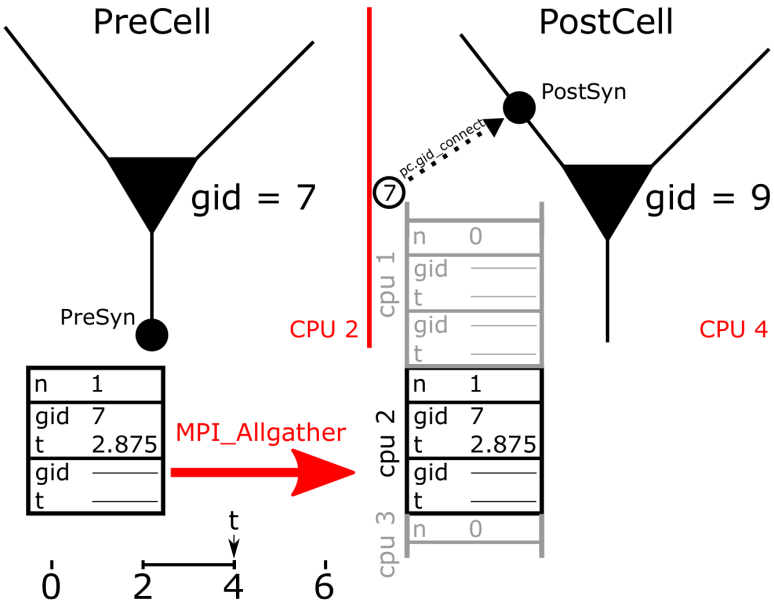
Spike exchange method



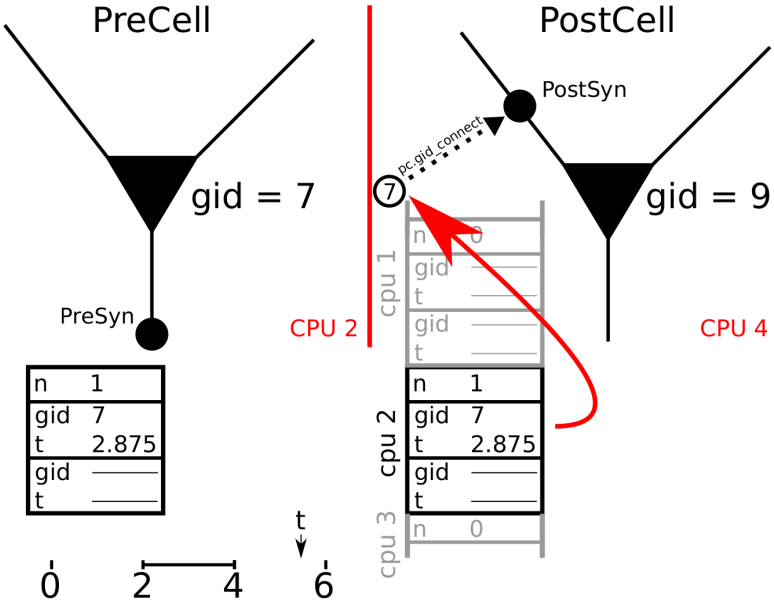
Spike exchange method



Spike exchange method



Spike exchange method

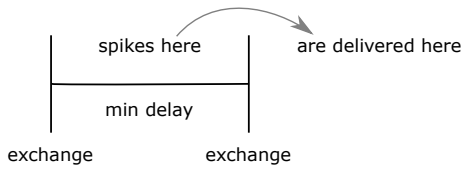


Exploit transmission delays: using pc.set_maxstep

Run using the idiom:

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

NEURON will pick an event exchange interval equal to the smaller of all the NetCon delays and of the argument to pc.set_maxstep. In general, larger intervals are better because they reduce communication overhead.



pc.set_maxstep must be called on each node; it uses MPI_Allreduce to determine the minimum delay.

Adding a presynaptic site

```
class Pyramidal:
    def __init__(self, gid):
        self._gid = gid
        self._setup_morphology()
        self._discretize()
        self._add_channels()
        self._register_netcon()
    def _register_netcon(self):
        self.nc = h.NetCon(self.soma[0](0.5)._ref_v, None, sec=self.soma[0])
        pc = h.ParallelContext()
        pc.set_gid2node(self._gid, pc.id())
        pc.cell(self._gid, self.nc)
    # the rest of the class stays unchanged
```

For most models, the delay due to axon propagation can be incorporated into a synaptic delay and thus it suffices to only make one connection point at the soma or axon hillock.

pc.set_gid2node must be called before pc.cell.

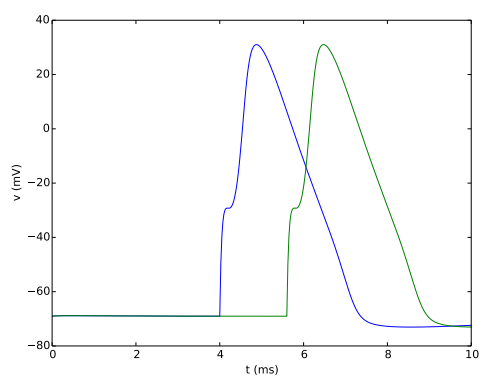
Building a two cell network

```
from neuron.units import ms, mV

class Network:
    def __init__(self):
        self.cells = [Pyramidal(i) for i in range(2)]
        # setup an exciteable ExpSyn on each cell's dendrites
        self.syns = [h.ExpSyn(cell.dend[0](0.5)) for cell in self.cells]
        for syn in self.syns:
            syn.e = 0 * mV
        # connect cell 0 to cell 1
        pc = h.ParallelContext()
        pre = 0
        post = 1
        self.nc = pc.gid_connect(pre, self.syns[post])
        self.nc.delay = 1 * ms
        self.nc.weight[0] = 1

n = Network()
```

Note: we use `for` loops and list comprehensions even when there is only two cells to avoid repeating ourselves (the DRY-principle) and to allow future generalization.

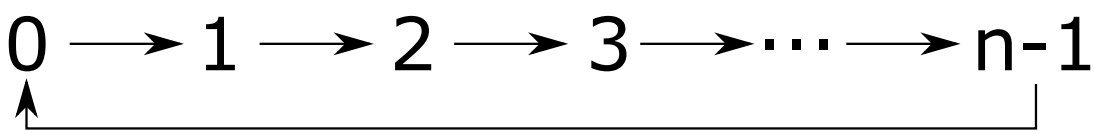
Running the two cell network

```
# drive the 0th cell
stim = h.NetStim()
stim.number = 1
stim.start = 3 * ms
ncstim = h.NetCon(stim, n.syns[0])
ncstim.delay = 1 * ms
ncstim.weight[0] = 1

t = h.Vector().record(h._ref_t)
v = [h.Vector().record(cell.soma[0](0.5)._ref_v)
     for cell in n.cells]

pc = h.ParallelContext()
pc.set_maxstep(10 * ms)
h.v_init = -69 * mV
h.stdinit()
pc.psolve(10 * ms)

for myv in v:
    pyplot.plot(t / ms, myv / mV)
pyplot.xlabel('t (ms)')
pyplot.ylabel('v (mV)')
pyplot.show()
```



Exercise: Generalizing to *n* cells in a ring network

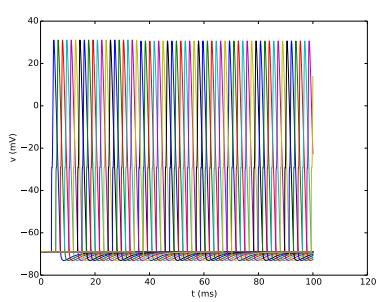
How can we generalize to a ring network with *n* cells?



Hint: As i increases, $i \% n$ counts: $0, 1, 2, \ldots, n-1, 0, 1, \ldots$

Solution: Generalizing to *n* cells in a ring network (100ms)

```
class Network:
    def __init__(self, num):
        self.cells = [Pyramidal(i) for i in range(num)]
        # setup an exciteable ExpSyn on each cell's dendrites
        self.syns = [h.ExpSyn(cell.dend[0](0.5)) for cell in self.cells]
        for syn in self.syns:
            syn.e = 0 * mV
        # connect cell i to cell (i + 1) % num
        pc = h.ParallelContext()
        self.ncs = []
        for i in range(num):
            nc = pc.gid_connect(i, self.syns[(i + 1) % num])
            nc.delay = 1 * ms
            nc.weight[0] = 1
            self.ncs.append(nc)

n = Network(20)
```



Storing spike times

With 20 cells, it is hard to distinguish the cells when simultaneously plotting the membrane potentials. Let's just store the spike times.

We begin by modifying `Pyramidal._register_netcon`:

```
def _register_netcon(self):
    self.nc = h.NetCon(self.soma[0](0.5)._ref_v, None, sec=self.soma[0])
    pc = h.ParallelContext()
    pc.set_gid2node(self._gid, pc.id())
    pc.cell(self._gid, self.nc)
    self.spike_times = h.Vector()
    self.nc.record(self.spike_times)
```

When the simulation is over, we can print out the spike times:

```
for i, cell in enumerate(n.cells):
    print('%d: %r' % (i, list(cell.spike_times)))
```

Beginning of output:

```
0: [4.600000000100032, 36.62500000009977, 69.12500000010715]
1: [6.200000000100054, 38.25000000010014, 70.75000000010752]
2: [7.800000000100077, 39.875000000100506, 72.37500000010789]
3: [9.4000000001, 41.500000000100876, 74.00000000010826]
```

Storing spike times: JSON

To store spike times in JSON, we can use the following code:

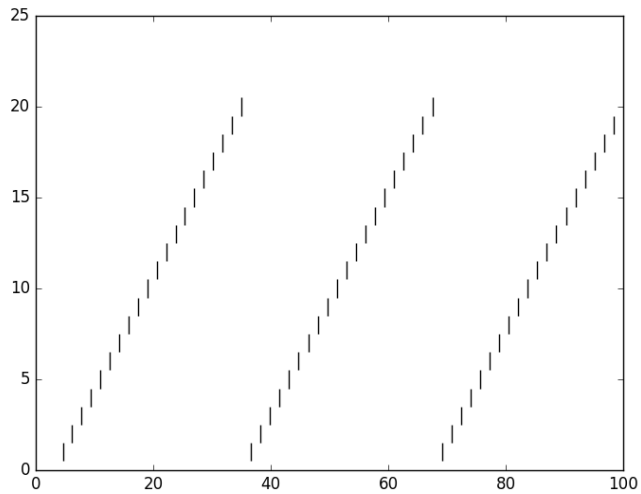
```
import json
with open('output.json', 'w') as f:
    f.write(json.dumps({i: list(cell.spike_times) for i, cell in enumerate(n.cells)},
                       indent=4))
```

This creates a file `output.json` which begins:

```
"0": [
    4.600000000100032,
    36.62500000009977,
    69.12500000010715
],
"1": [
    6.200000000100054,
    38.25000000010014,
    70.75000000010752
],
"2": [
    7.800000000100077,
    39.875000000100506,
    72.37500000010789
],
```

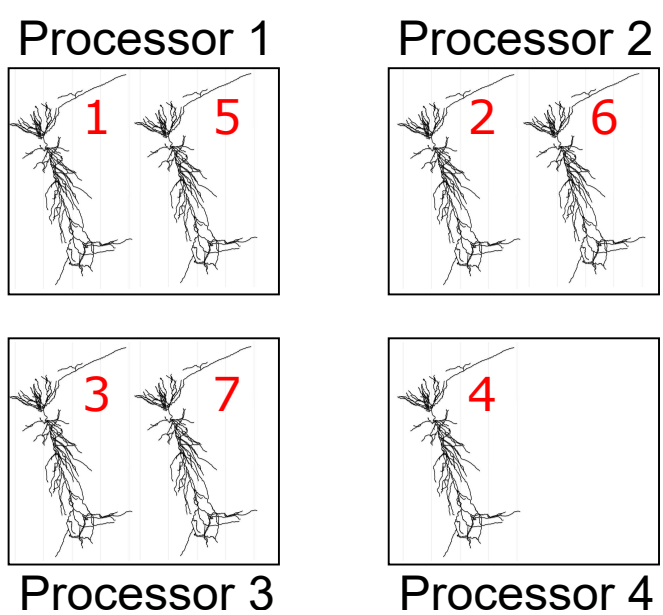
JSON is a standard format for data interchange. Libraries are available for most programming languages.

Raster plots




```
for i, cell in enumerate(n.cells):
    pyplot.vlines(cell.spike_times, i + 0.5, i + 1.5)
pyplot.show()
```

Simple load-balancing strategy: round-robin.



Simple load-balancing strategy: round-robin.

CPU 0			CPU 3		CPU 4	
pc.id	0	•••	pc.id	3	pc.id	4
pc.nhost	5		pc.nhost	5	pc.nhost	5
ncell	14		ncell	14	ncell	14
gid			gid		gid	
0			3		4	
5			8		9	
10			13			

An efficient way to distribute, especially if all cells similar:

```
for gid in range(pc.id(), ncell, pc.nhost()):
    pc.set_gid2node(gid, pc.id())
    ...
```

(Note: the body is executed at most $\lceil$`ncell/nhost`$\rceil$ times, not `ncell`.)

Advanced load-balancing: balance work not number of cells

Strategy:

- Distribute cells round-robin to all processors, instantiate them.
- Compute an estimate of the computational complexity:

```
def complexity(self):
    h.load_file('loadbal.hoc')
    lb = h.LoadBalance()
    return lb.cell_complexity(sec=self.all[0])
```

- Destroy the cells, send the gid-complexity data to node 0.
- (On node 0): distribute gids such that the next gid goes to the node with the least amount of complexity.
- Send the gids to the nodes; instantiate the cells.

For a more accurate (but computationally more intensive) estimate of complexity, use lb.ExperimentalMechComplex and lb.read_complex.

Parallelizing our ring network with round-robin

Very few changes are necessary.

MPI must be initialized before we can use it:

```
h.nrnmpi_init()
```

The `Network` class only instantiates gids on the current processor.

```
class Network:
    def __init__(self, num):
        pc = h.ParallelContext()
        mygids = list(range(pc.id(), num, pc.nhost()))
        self.cells = [Pyramidal(i) for i in mygids]
        # setup an exciteable ExpSyn on each cell's dendrites
        self.syns = [h.ExpSyn(cell.dend[0](0.5)) for cell in self.cells]
        for syn in self.syns:
            syn.e = 0 * mV
        # connect cell (i - 1) % num to cell i
        self.ncs = []
        for i, syn in zip(mygids, self.syns):
            nc = pc.gid_connect((i - 1) % num, syn)
            nc.delay = 1 * ms
            nc.weight[0] = 1
            self.ncs.append(nc)
```

Parallelizing our ring network

We must modify the initial netstim to ensure it only attaches to gid 0 not to the 0th cell in each process.

```
# drive the 0th cell
if pc.gid_exists(0):
    stim = h.NetStim()
    stim.number = 1
    stim.start = 3
    ncstim = h.NetCon(stim, n.syns[0])
    ncstim.delay = 1
    ncstim.weight[0] = 1
```

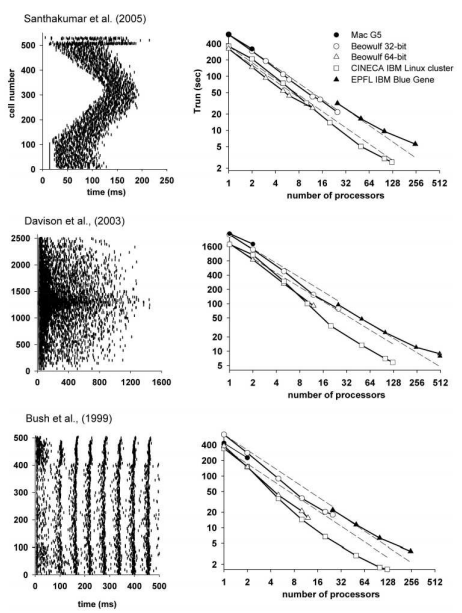
Finally, we modify the write to do it on a per-processor basis:

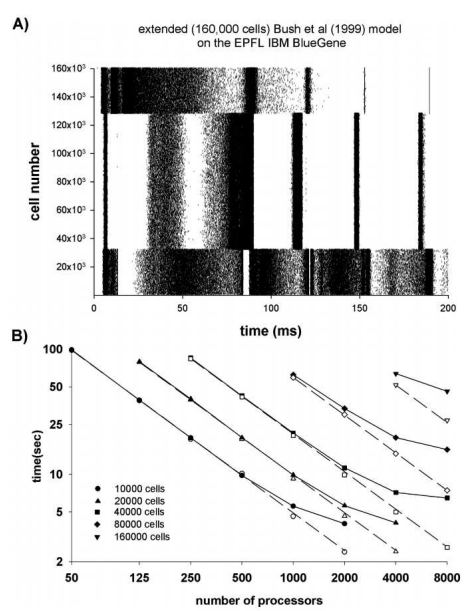
```
 with open('output%d.json' % pc.id(), 'w') as f:
    f.write(json.dumps({cell._gid: list(cell.spike_times) for cell in n.cells},
        indent=4))
```


Optional: use pc.py_alltoall to send all spikes to node 0

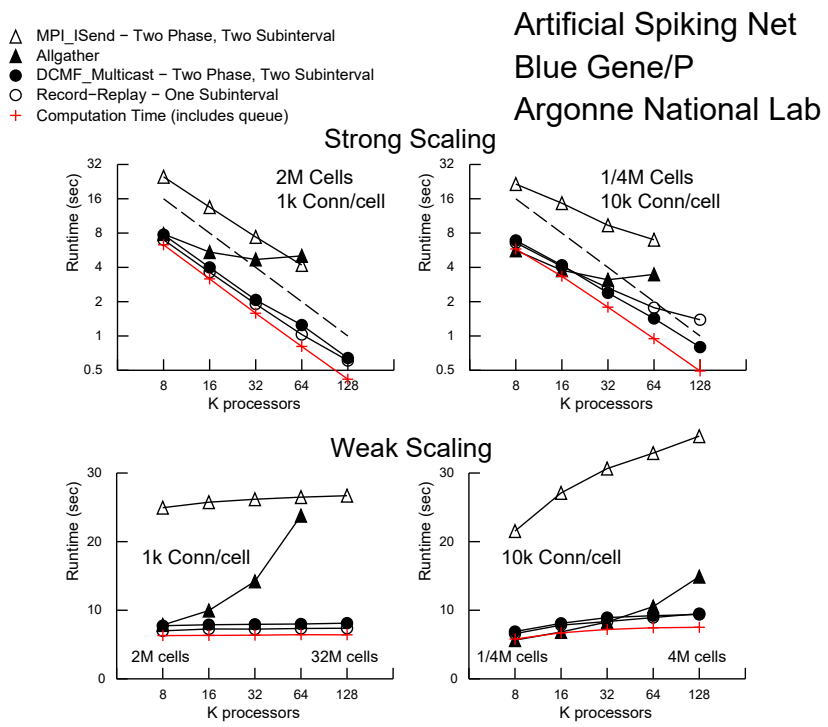
```
local_data = {cell._gid: list(cell.spike_times) for cell in n.cells}
all_data = pc.py_alltoall([local_data] + [None] * (pc.nhost() - 1))

if pc.id() == 0:
    # only do output from node 0
    import json
    combined_data = {}
    for node_data in all_data:
        combined_data.update(node_data)
    with open('output.json', 'w') as f:
        f.write(json.dumps(combined_data, indent=4))
```

Performance: MPI scaling





Performance: Spike exchange strategies



Performance Tip

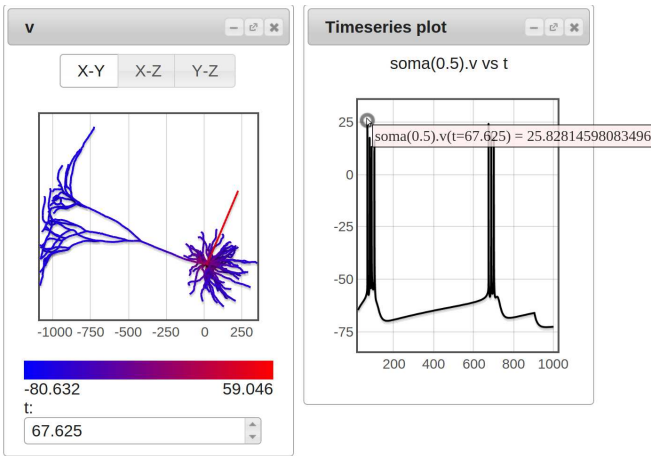
Tip: For network models, use a fixed step solver and not a variable step solver.

Question

Suppose we now realize we want to know the time series of the *m* variable in the center of the soma of cell 5. We only stored spike times. Do we have to modify our code to store that variable and rerun the entire simulation?

Tip: Store synaptic events; recreate single cells as needed

initial conditions
+
synaptic events → neuron dynamics



Using spike data to recreate a variable of interest

We will need `vecevent.mod`. If you have NEURON, this file should be on your computer somewhere. Alternatively, you can download it from:

https://github.com/neuronsimulator/nrn/blob/master/share/examples/nrniv/netcon/vecevent.mod

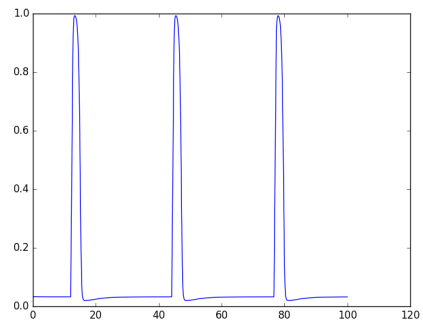
Using spike data to recreate a variable of interest

```
import json
from neuron import h
from neuron.units import ms, mV
from PyNeuronToolbox import morphology
from matplotlib import pyplot
h.load_file('stdrun.hoc')
num_cells = 20

# class Pyramidal as before

# read spike times
with open('output.json') as f:
    spike_times_by_cell = json.load(f)
```

(continued)

Using spike data to recreate a variable of interest

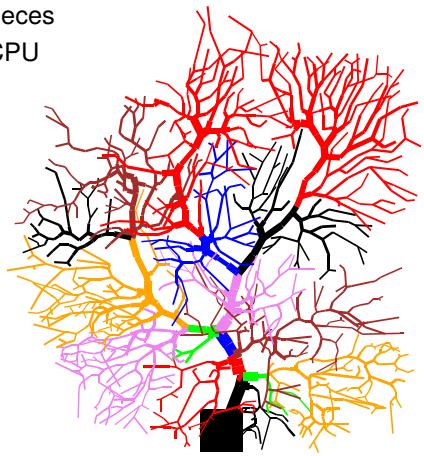
```
def get_m(gid):
    p = Pyramidal(gid)
    # recreate synaptic inputs (here, only one; you may have multiple)
    precell = (gid - 1) % num_cells
    vs = h.VecStim()
    spike_vec = h.Vector(spike_times_by_cell[str(precell)])
    vs.play(spike_vec)
    syn = h.ExpSyn(p.dend[0](0.5))
    nc = h.NetCon(vs, syn)
    nc.delay = 1 * ms
    nc.weight[0] = 1
    # setup recording
    t = h.Vector().record(h._ref_t)
    m = h.Vector().record(p.soma[0](0.5)._ref_m_hh)
    # do run
    pc = h.ParallelContext()
    pc.set_maxstep(10 * ms)
    h.v_init = -69 * mV
    h.stdinit()
    pc.psolve(100 * ms)
    return t, m

t, m = get_m(5)
pyplot.plot(t, m)
pyplot.show()
```

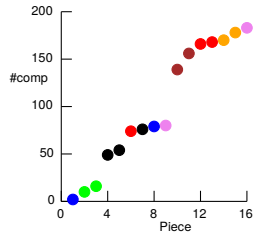


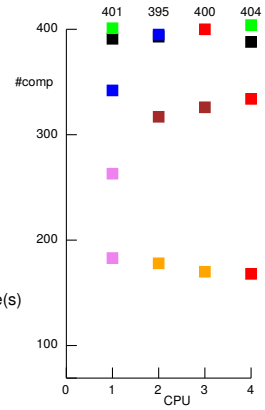
Multisplit

Improve load balancing with multisplit

16 Pieces
4 CPU





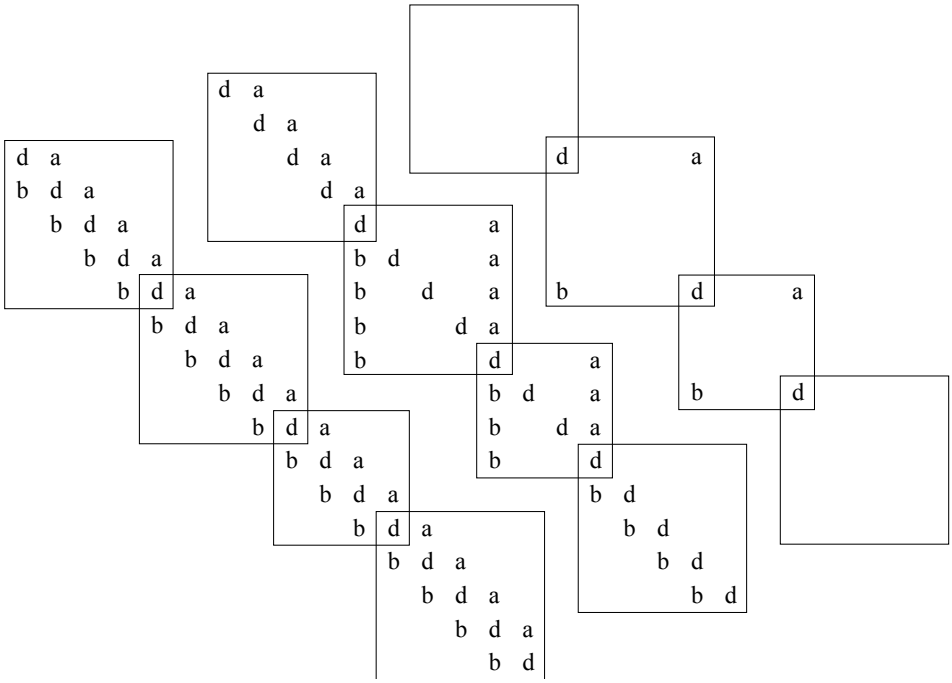


	Time (s)	
CPU	Computation	Exchange
0	13.82	0.56
1	13.35	1.03
2	13.47	0.90
3	13.56	0.82

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

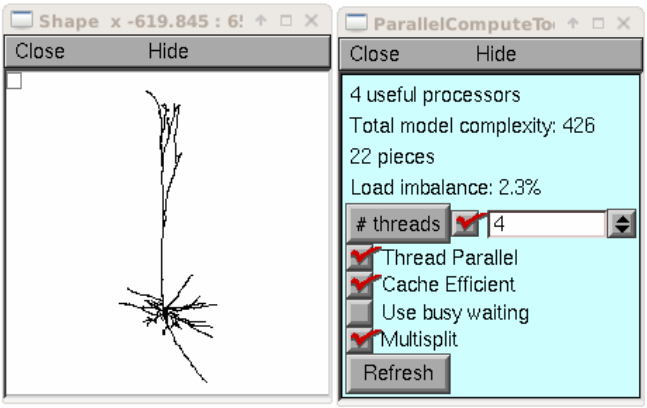
Multisplit algorithm described in Hines et al 2008. DOI: 10.1007/s10827-008-0087-5

Multisplit: methods



Using multisplit (threads)

When not using MPI, enabling thread-based multisplit is as easy as clicking a checkbox:



Using multisplit (MPI)

For process-based multisplit (with MPI), use `pc.multisplit` to declare split nodes:

```
pc.multisplit(x, subtreeid, sec=sec)
```

After all split nodes are declared, **every** process must execute:

```
pc.multisplit()
```

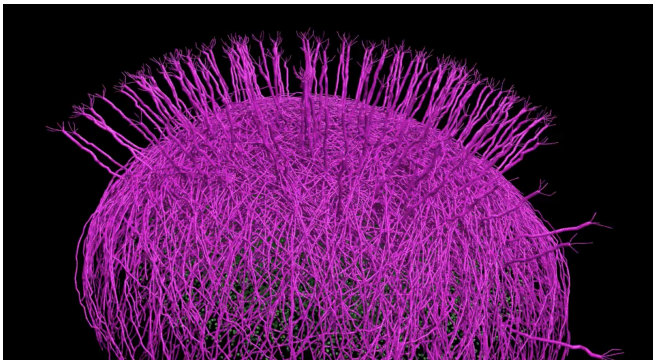
If created, destroy any parts of the cell that do not belong on the processor.

Rules:

- Each subtree can have at most two split nodes.
- Does not support variable step, linear mechanisms, extracellular, or reaction-diffusion.
- `h.distance` cannot compute path distances that cross a split node.

Tip: For load balancing, it is sometimes convenient to split cells into more pieces than processes.

Example: Migliore et al 2014



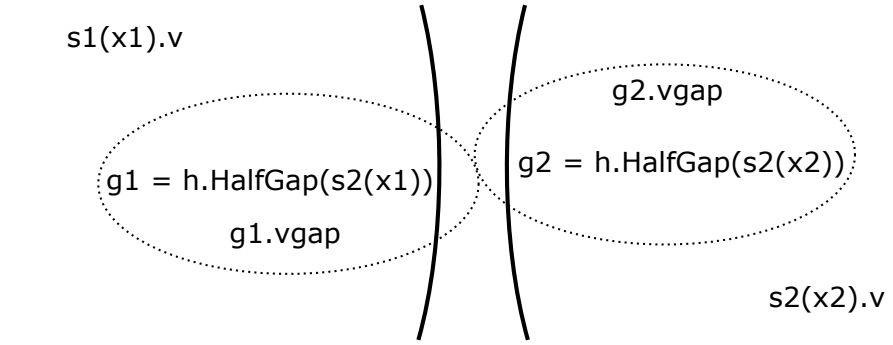
Migliore et al 2014 used multisplit to improve load balancing on a model of the olfactory bulb.

http://modeldb.yale.edu/151681

See, in particular, the file `multisplit_distrib.py`.

Gap Junctions

Continuous voltage exchange

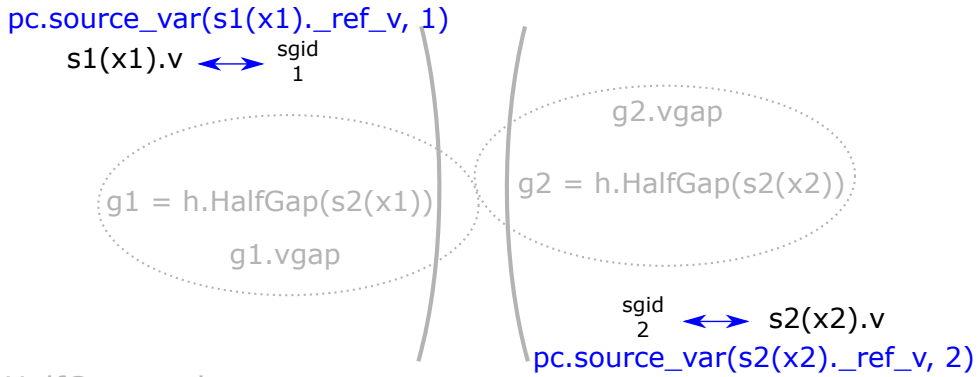


HalfGap.mod

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

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

pc.source_var to declare source sgid

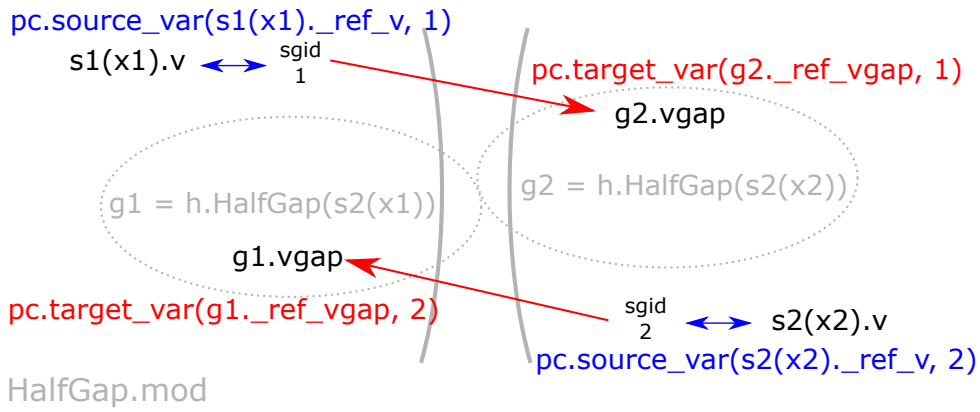


HalfGap.mod

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

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

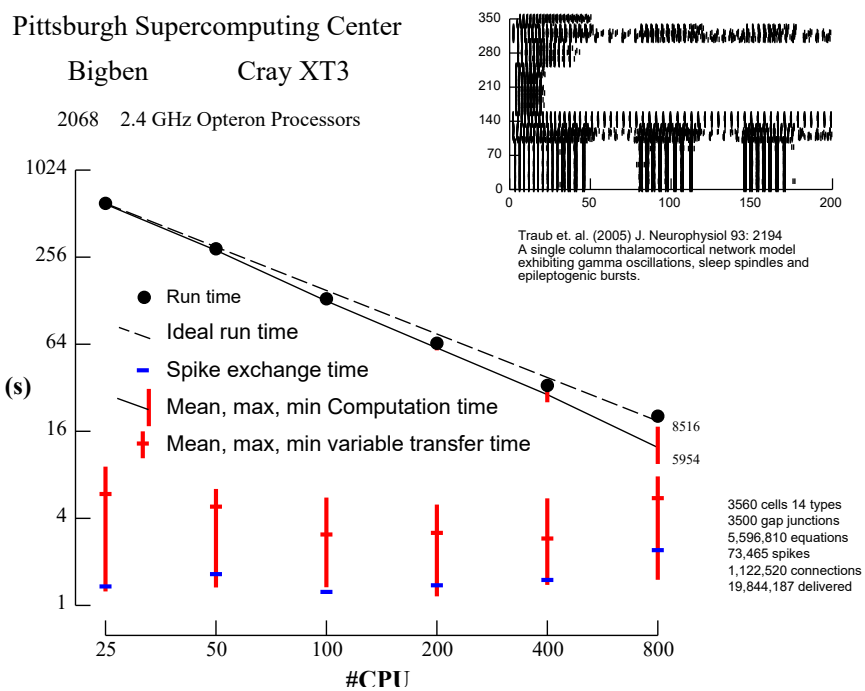

pc.target_var to declare target connection



HalfGap.mod

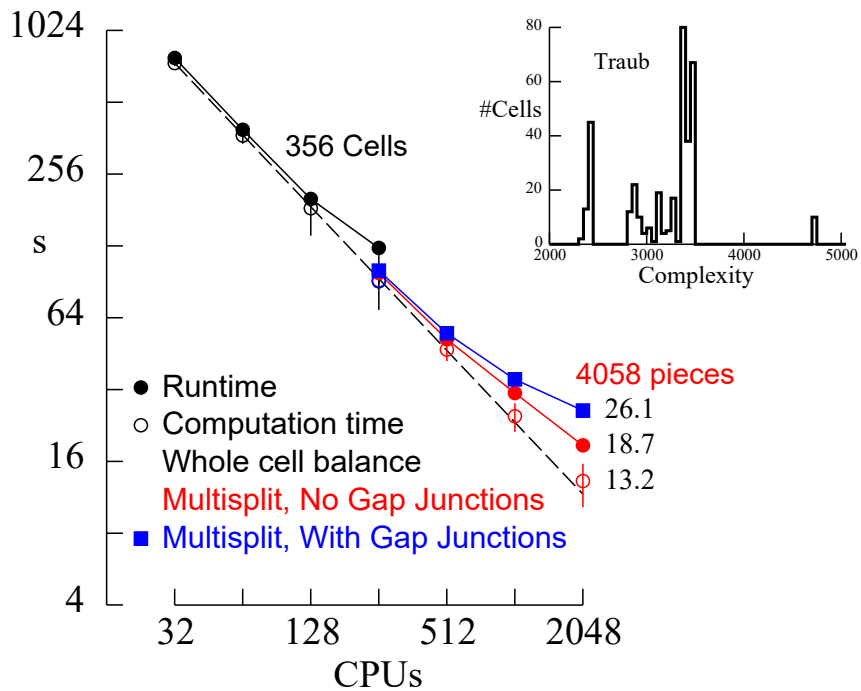
```
NEURON  {                          ASSIGNED {
  POINT_PROCESS HalfGap              v (millivolt)
  ELECTRODE_CURRENT i                vgap (millivolt)
  RANGE r, i, vgap                   i (nanoamp)
}                                  }
PARAMETER { r = 1e9 (megohm) }     CURRENT { i = (vgap - v) / r }
```

Performance: Traub model



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

Performance: Traub model with multisplit



Finally: Subworlds

Use `pc.subworlds` to combine parallel simulation with parallel bulletin-board based parameter search.

```
from neuron import h
h.nrnmpi_init()
pc = h.ParallelContext()
pc.subworlds(2)

from model import runmodel

pc.runworker()

for ncell in range(5, 10):
    pc.submit(runmodel, ncell, 1, 100)

while (pc.working()):
    print(pc.pyret())

pc.done()
h.quit()
```

Note: Unless memory on a single node is a limiting factor, you will likely want either 1 subworld (everything) or `pc.nhost()` subworlds. In the first case, there is no need to use subworlds since simulations are run one at a time; in the other extreme, there is also no need since each simulation runs on a single processor.

Neuroscience Gateway: Enabling Supercomputing for Neuroscience Research and Education

Amitava Majumdar[1], Subhashini Sivagnanam[1],
Ted Carnevale[2], Kenneth Yoshimoto[1]
[1]UCSD, San Diego, CA; [2]Yale University, New Haven, CT

Neuroscience's Growing Need for High Performance Computing (HPC)

- Increased size and complexity of computational models
- Wider use of optimization and parameter space exploration
- Projects that require running many simulations, e.g. to examine roles of noise or stochasticity, determine parameter sensitivity, evaluate learning rules
- Expanding use of experimental methods that generate massive amounts of data requiring computationally intensive analysis

Barriers to using HPC

- Writing peer-reviewed proposals for computer time.
- Understanding HPC machines, policies, complex OS/software.
- Installing and benchmarking complex tools on HPC resources.
- Understanding and managing multiple remote authentication systems.
- Dealing with data transfer, management, and storage issues.

Few neuroscientists could access HPC before the Neuroscience Gateway was developed. Projects may have started small by design, but entry barriers forced many to stay small.

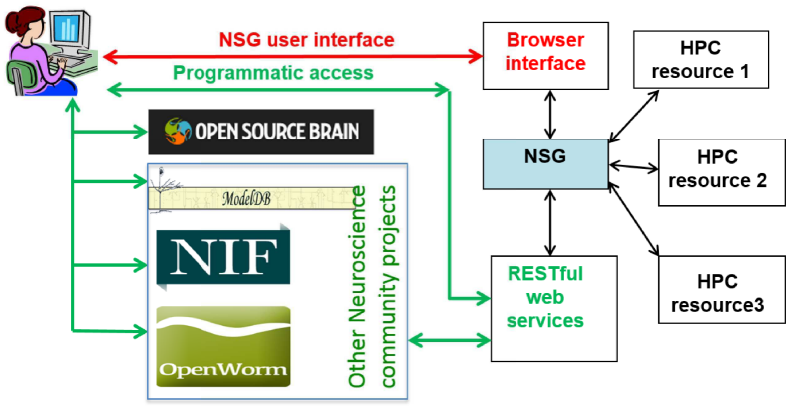
The Neuroscience Gateway (NSG)

NSG https://www.nsgportal.org provides free, simple, secure access to XSEDE's HPC resources. NSG's portal and programmatic service make it easy to use neuroscience-related software and tools.

Partial list of tools currently available at NSG:

Brian	NetPyNE	EEGLAB	Python
CARLsim	NEURON	FREESURFER	MATLAB
DynaSim	PyNN	FSL	Octave
GENESIS	BluePyOpt	TensorFlow	R
NEST		Virtual Brain Empirical Pipeline	

New tools are added on request.



GUI Access

NSG Portal's simple, easy to use web interface provides

- access to XSEDE HPC resources, HPC software stack.
- access to architectures such as GPUs, KNL.
- support for "bundling" of jobs, i.e. multiple single core executions in parallel (e.g. for embarrassingly parallel tasks, such as parameter sweep studies).
- support for custom workflows, e.g. Virtual Brain pipeline.

Programmatic Access

A RESTful API (NSG-R) offers most portal functionality

- submit, cancel, and delete jobs
- list and check status of submitted jobs
- list and download results
- list working directory

Example:

```
curl –u username:$PASSWORD -H cipres-
appkey:$KEY $URL/job/username -F
tool=NEURON74_TG -F
input.infile_=@./JonesEtAl2009_r31.zip
-F vparam.number_nodes_=2
```

Getting an Account

An account is needed to use NSG directly through its portal or RESTful interface.

- Apply at https://www.nsgportal.org/gest/reg.php
- Contact and brief technical information required for user verification.
- Accounts are usually set up within 24 hours.
- Users are added to the NSG email list, which gets occasional news posts.

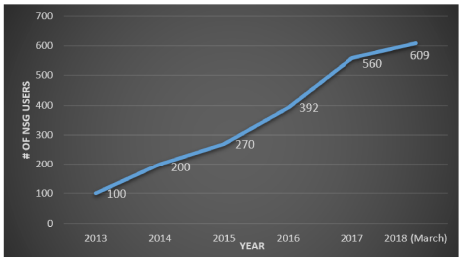
Do you even need an account?

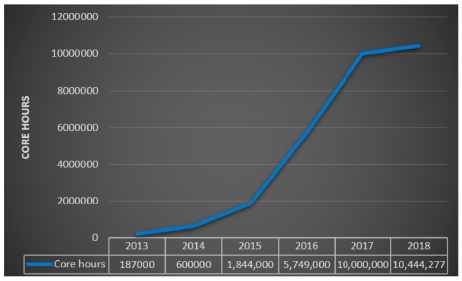
OpenSourceBrain, BluePyOpt, and some other neuroscience community projects have their own "umbrella accounts" that cover their users.

Their users can run jobs on NSG without having to register with NSG.

Users of downloadable software packages that have integrated NSG access, such as SimTracker, don't need individual accounts.

Growth of NSG Usage





Evolution of NSG

Initially implemented to provide streamlined access to HPC resources for neuroscientists dealing with large scale modeling projects.

Subsequently expanded to meet other HPC needs of the broader neuroscience community, especially cognitive and experimental neuroscientists faced with computationally challenging tasks.

Evolution of NSG *continued*

Neuroscience software tools/application development and dissemination

- Integration with EEGLAB (Scott Makeig, UCSD), Human Neocortical Neurosolver (HNN) (Stephanie Jones, Brown University), HBP's BluePyOpt (Michele Migliore, CNR, Italy)
- CARLsim--GPU-accelerated SNN simulator (Jeffrey Krichmar, UCI), LSNM--Large Scale Neural Simulator (Antonio Ulloa, Neural Bytes LLC)

Education and training

- NEURON summer course, NIH- and NSF-supported computational neuroscience and cyberinfrastructure training (U. Missouri, UCSD), workshops at SFN and OCNS meetings

Collaborative environment

- for application development and testing, sharing code and data

Summary

NSG catalyzes and democratizes computational neuroscience research for everyone, regardless of local or institutional resources.

- If you use NSG, please cite **S. Sivagnanam, A. Majumdar, K. Yoshimoto, V. Astakhov, A. Bandrowski, M.E. Martone, and N.T. Carnevale. Introducing the Neuroscience Gateway, IWSG, vol. 993 of CEUR Workshop Proceedings, CEUR-WS.org, 2013.**
- Also please notify us nsghelp@sdsc.edu of your presentations and publications so we can include them in reports.

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`

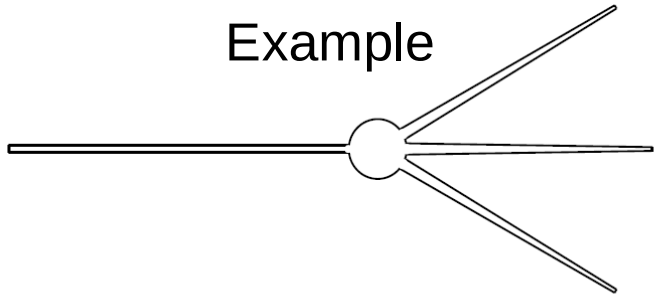
Specify the default section

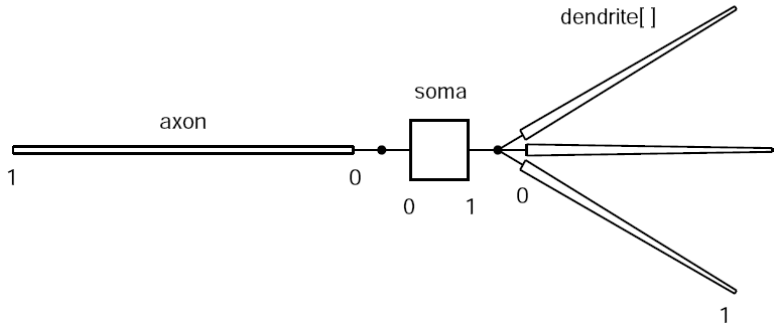
`access`

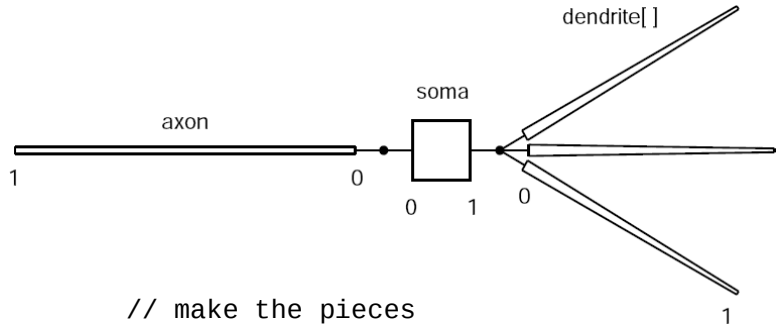
Assemble the pieces

`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"

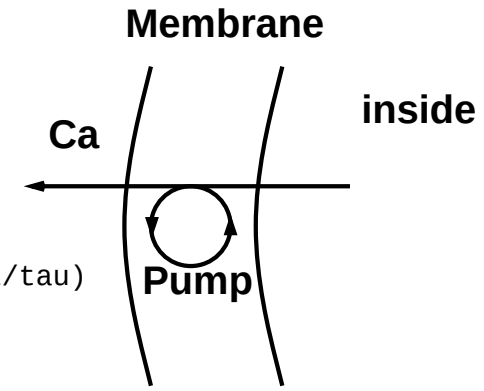
```
load_file("nrngui.hoc")
load_file("modelspec.hoc")
load_file("rig.ses")
```

Workflow

Develop/debug "virtual organism"
- hoc, Python, NMODL, GUI (CellBuilder, Channnel 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

Rate limited active transport




```
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 has dimensions of (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)
}
```


Declarations for capump.mod

```
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)
}

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

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

Equations for capump.mod

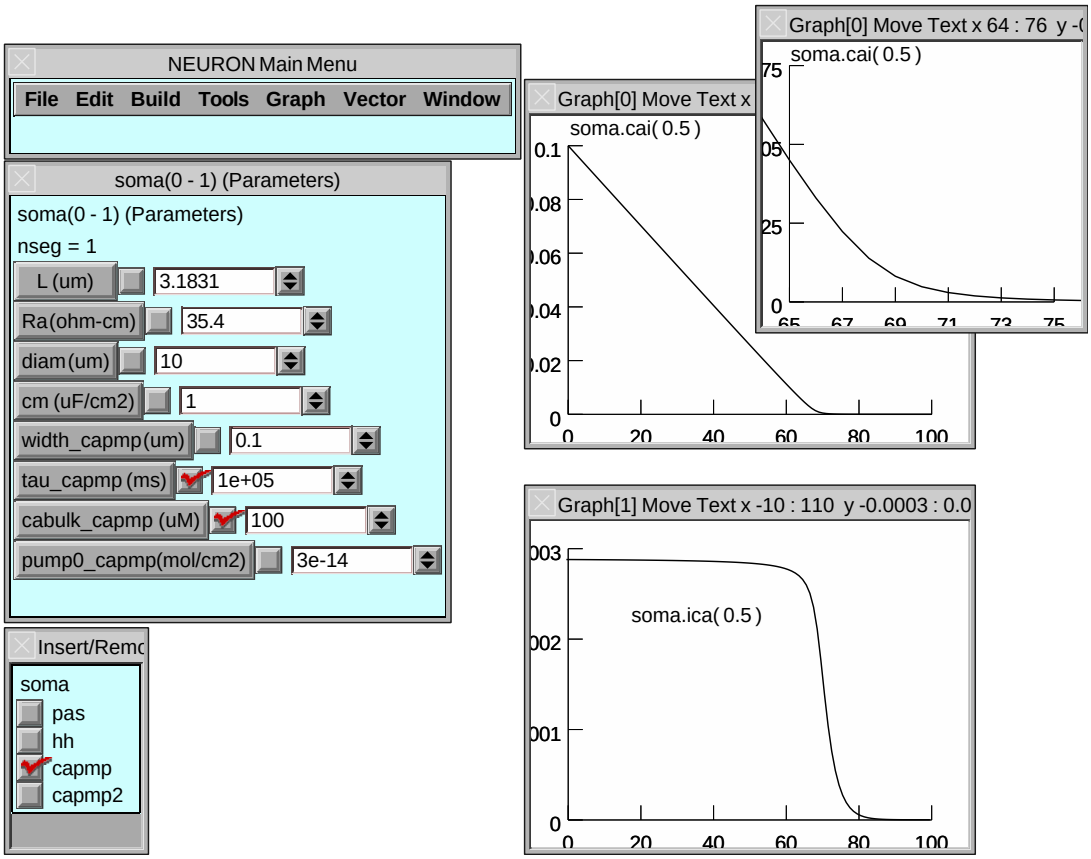
```
INITIAL {
  ica = 0  ica_pmp = 0
  ica_pmp_last = 0
  SOLVE pmp STEADYSTATE sparse
}
```

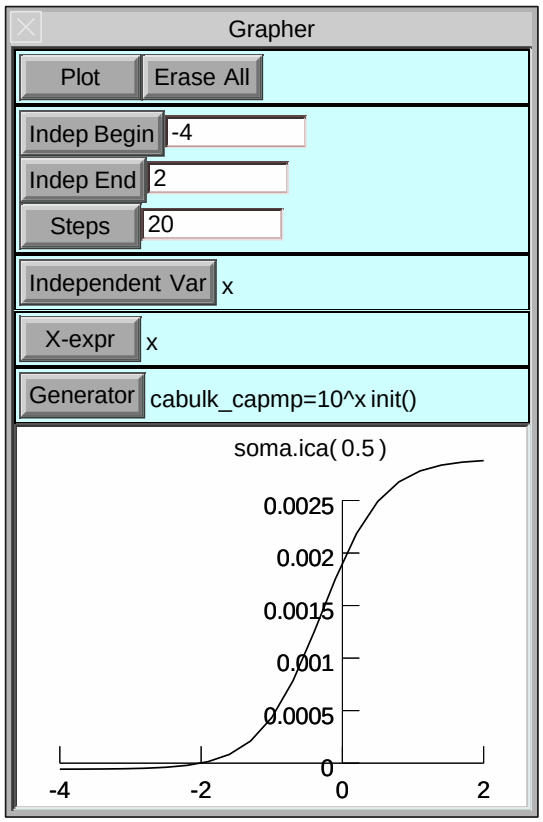
```
BREAKPOINT {
  SOLVE pmp METHOD sparse
  ica_pmp_last = ica_pmp
  ica = ica_pmp
}
```

```
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
  ~ 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("nrngui.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()
  if (cvode.active()) { cvode.re_init() }
}
```





The Linear Circuit Builder

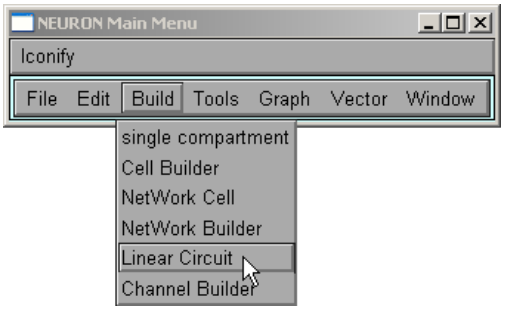
For building models that have linear circuit elements and may also involve neurons

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

Potential applications include

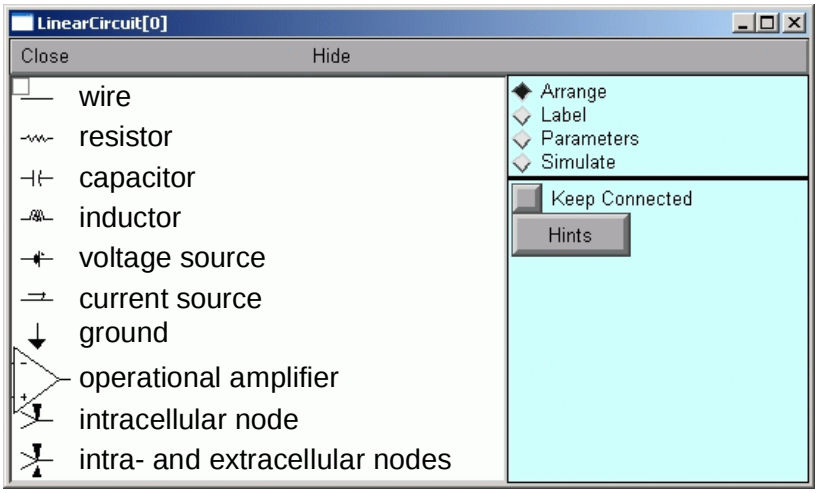
- effects and compensation of electrode R & C
- two-electrode voltage clamp
- ohmic and nonlinear gap junctions

1. Bring up a Linear Circuit Builder



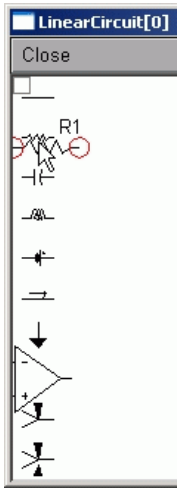
NEURON Main Menu / Build / Linear Circuit

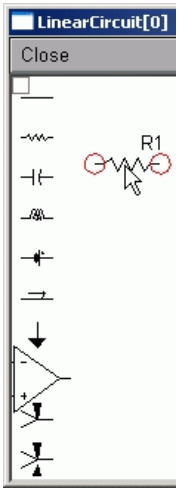
The Linear Circuit Builder



Arrange: spawn components

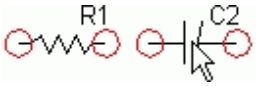
Click on palette and drag onto canvas



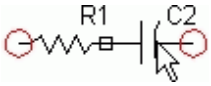


Arrange: connect components

Click and drag to overlap red circles



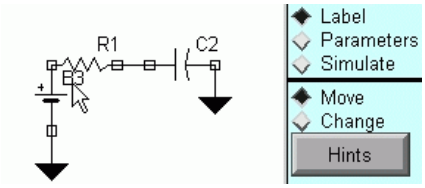
Black square is "solder joint"

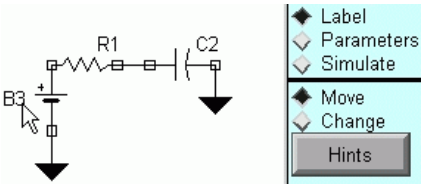


Pull apart to break connection

Label: move labels

Click and drag to new location

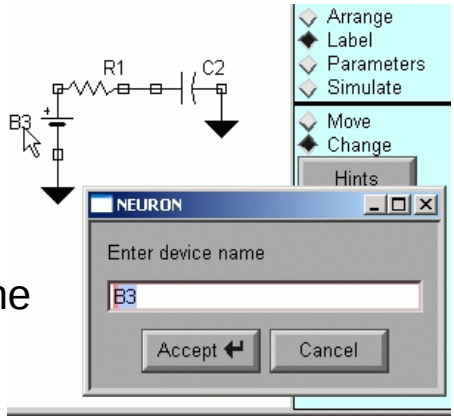




Label: change labels 1

Click on a label . . .

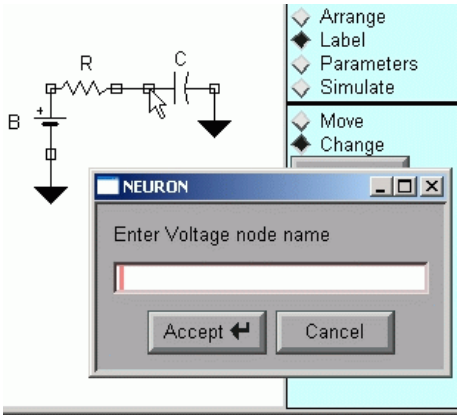
. . . to change its name



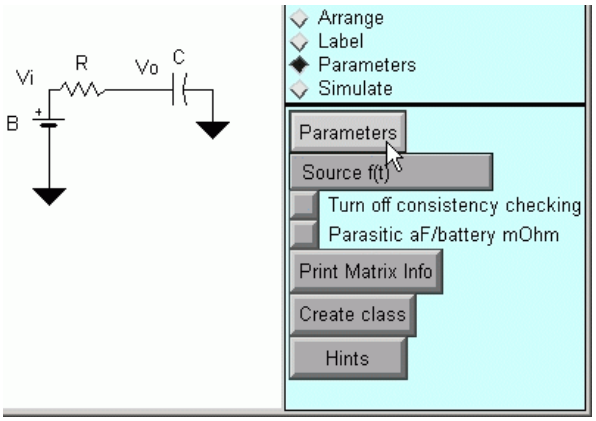
Label: change labels 2

Click on a node . . .

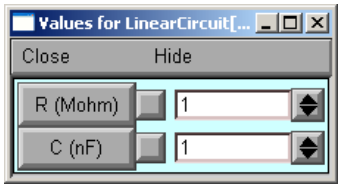
. . . to label a voltage



Parameters: non-source elements

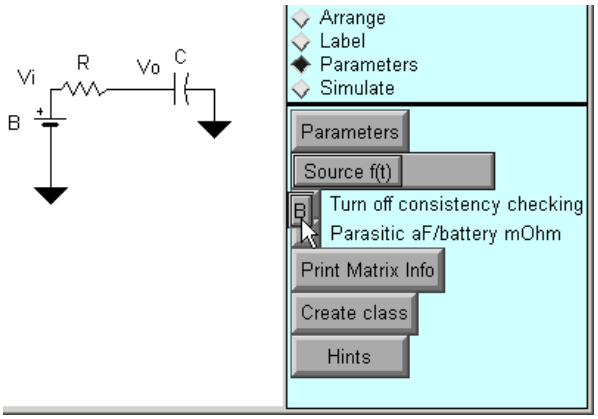


Click on "Parameters"

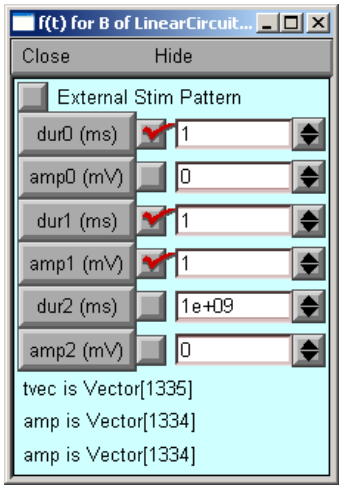


Parameters: signal sources

Source f(t) / B



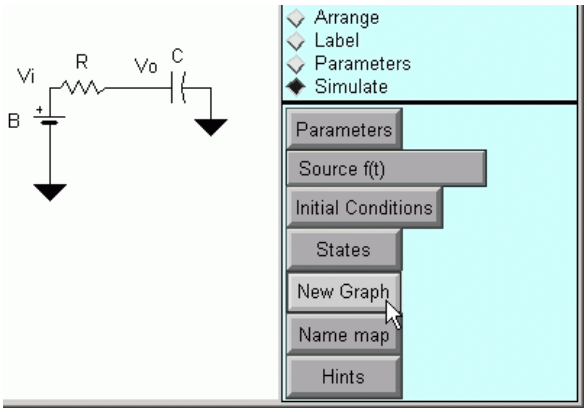
Parameters: signal sources *continued*



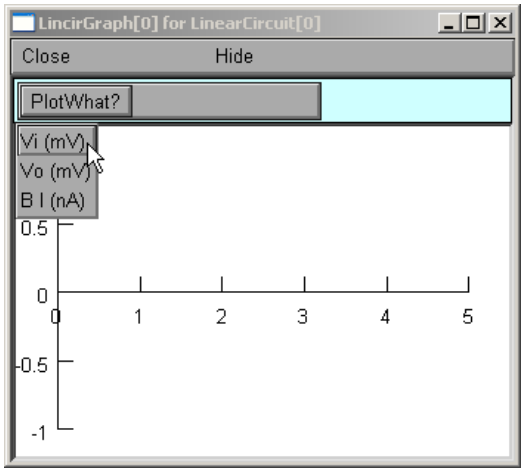
Configured

Simulate: creating a graph

New Graph

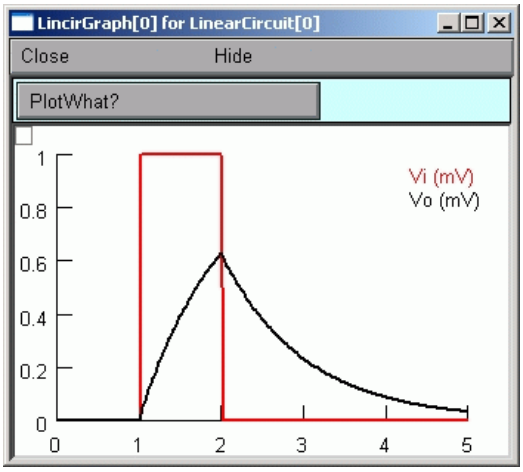


Simulate: specifying what to plot



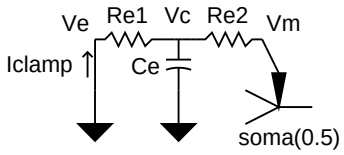
PlotWhat? / *variable_label*

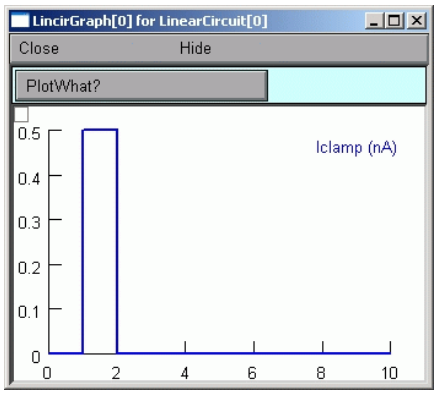
Simulate: simulation results

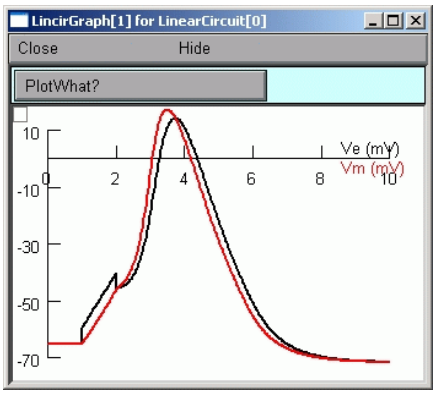


After minor cosmetic changes

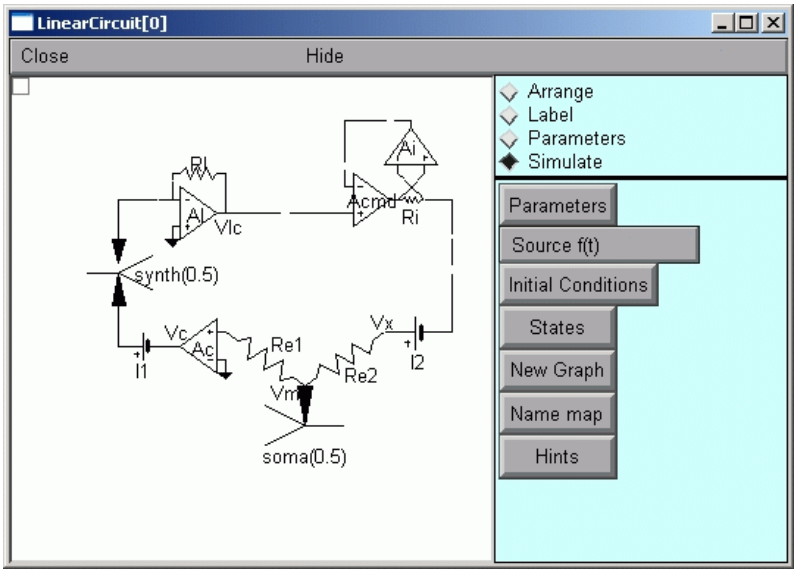
Patch clamp with electrode R and C







NEURON demo: dynamic clamp



NEURON's tools for Analysis of Electrical Signaling

- Input and transfer impedances
- Voltage transfer ratio

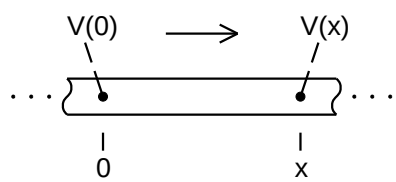
  $V_{downstream}/V_{upstream}$
- Electrotonic transformation

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

. . . all as functions of frequency and space

Classical Cable Theory

Infinite cylinder
in the steady state:



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

x = physical distance

λ = length constant

Classical "electrotonic distance"

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

so attenuation $A^V(x) = V(0)/V(x) = e^X$

Intuitively simple

Problems

Neurons are not infinite cylinders.

Attempted fix: reduce dendritic tree to finite length equivalent cylinder

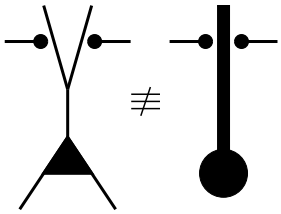
$A^V(x) = \cosh L_{classical} / \cosh (L_{classical} - X)$

$L_{classical}$ = physical length / λ

$X = x / \lambda$

The bad news about the equivalent cylinder approximation

- Neither intuitive nor simple.
- Destroys spatial relationships among synaptic inputs.
- Classical electrotonic distance $X = x / \lambda$ fosters conceptual error by obscuring the direction-dependence of attenuation in finite structures.



The good news about the equivalent cylinder approximation

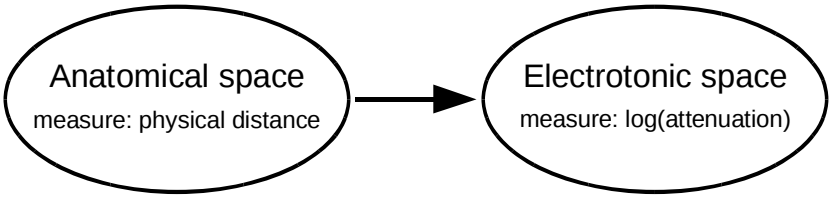
It isn't valid

Property	Assumption	Truth
Dendritic terminations	electrically equidistant from soma	varies widely
Diameters	cylindrical	irregular
Branch points	3/2 power rule $d_p^{3/2} = \Sigma\, d_d^{3/2}$	no

The Electrotonic Transformation

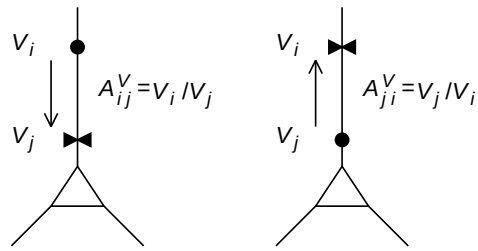
A transformation from anatomical to electrotonic space that

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



Foundation: two-port analysis of electrotonus

How well do signals propagate?



Signal transfer is direction-dependent: $A_{ij}^V \neq A_{ji}^V$

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

The Electrotonic Transformation

Functional definition of electrotonic distance

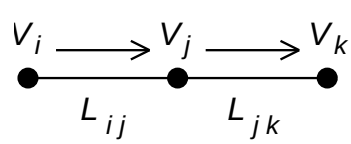
L = log(attenuation)

✔ simple, direct relationship to attenuation

✔ direction-dependent: $L_{ij}^V = \log(A_{ij}^V)$, $L_{ji}^V = \log(A_{ji}^V)$, and in general $L_{ij}^V \neq L_{ji}^V$

✔ in an infinite cylinder, is identical to classical electrotonic distance

✔ additive over a path with constant direction of propagation



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

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

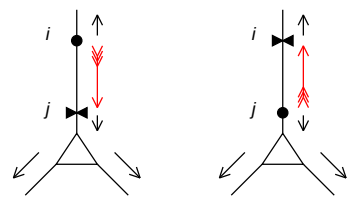
Using the Electrotonic Transformation

At a frequency of interest

1. compute log(attenuation) between a reference point and all other points of interest
2. display results graphically (optional)

A convenient reference point: the soma

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



The attenuation identities give us the transform identities

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

Synaptic location and synaptic efficacy

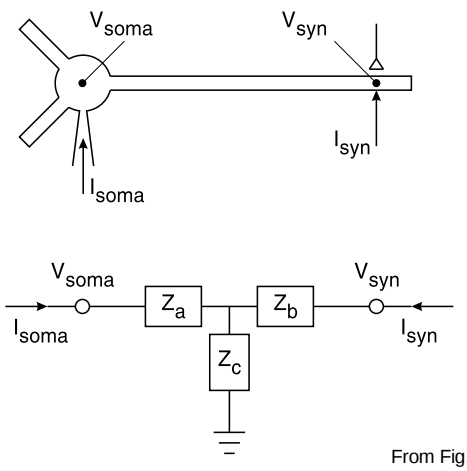
Q: What predicts how PSP amplitude at the soma varies with synaptic location?

A: If synapses act like voltage sources,

A_{in}^{V} (voltage attenuation from synapse to soma)

or

$k_{syn\text{->}soma}$ (synapse to soma voltage transfer ratio)



From Fig. 1 in Jaffe & Carnevale 1999

If synapses act like voltage sources,

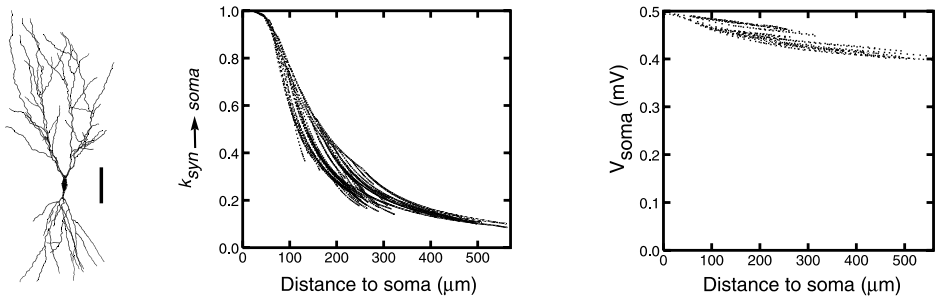
$V_{syn}(t)$ is independent of synaptic location

and

synapse to soma voltage transfer ratio

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

predicts variation of somatic PSP amplitude with synaptic location.



Somatic PSP predicted by voltage transfer ratio $k_{syn\text{->}soma}$

Somatic PSP generated by conductance-change synapse

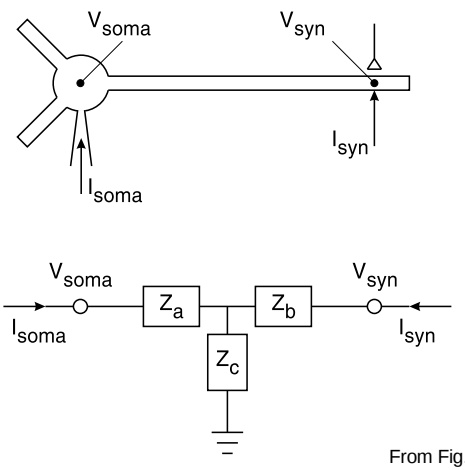
From Fig. 5 in Jaffe & Carnevale 1999

Results:

1. $k_{syn\text{->}soma}$ fails to predict the relationship between somatic PSP amplitude and synaptic location.
2. Synapses do not act like voltage sources.

Q1: What do synapses act like?

Q2: What would be a better predictor of the relationship between somatic PSP and synaptic location?



From Fig. 1 in Jaffe & Carnevale 1999

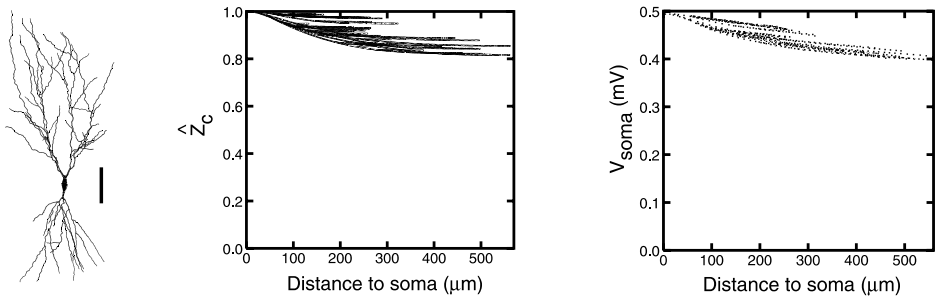
If synapses act like current sources,

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

and

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

Let's try it . . .



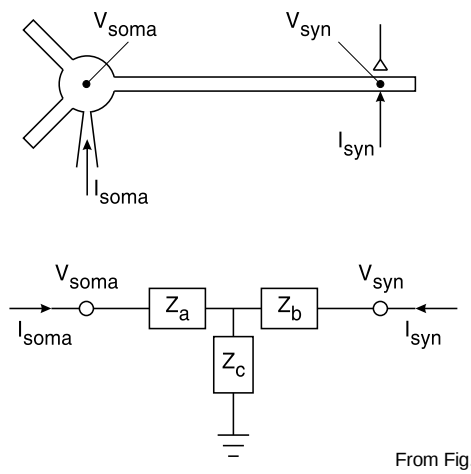
Somatic PSP predicted by transfer impedance $\hat{Z}_c$

Somatic PSP generated by conductance-change synapse

From Fig. 5 in Jaffe & Carnevale 1999

Results:

1. Normalized transfer impedance $\hat{Z}_c$ predicts the relationship between somatic PSP amplitude and synaptic location.
2. Synapses act like current sources.



From Fig. 1 in Jaffe & Carnevale 1999

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

Proof: $k_{soma\text{->}syn} = Z_c / (Z_a + Z_c) = Z_c / Z_N^{soma}$

but Z_N^{soma} is the maximum transfer Z between any location and the soma.

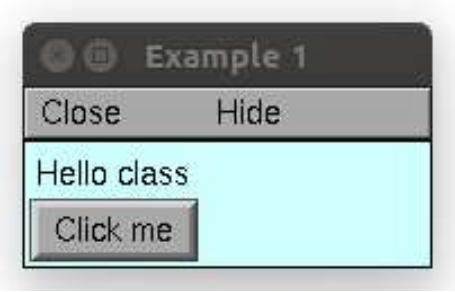
Therefore $k_{soma\text{->}syn} = \hat{Z}_c$

GUI development

Making your own graphical interface

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

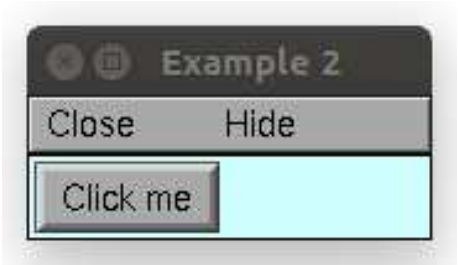
```
from neuron import h, gui

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



Button actions

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

```
from neuron import h, gui

def say_hello():
    print('hello!')

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



Pressing the button displays:

```
hello!
```

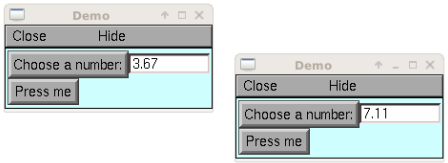
Pressing the button twice:

```
hello!
hello!
```

Number fields and classes

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

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

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



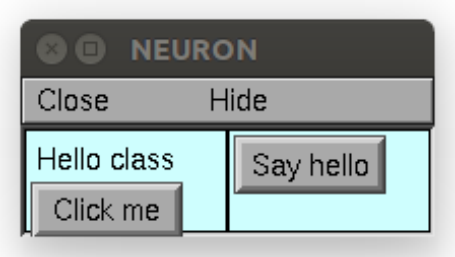
Clicking "Press me" on the left window and then on the right window displays:

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


Layout: HBox and VBox

Combine windows horizontally with HBox and vertically with VBox.

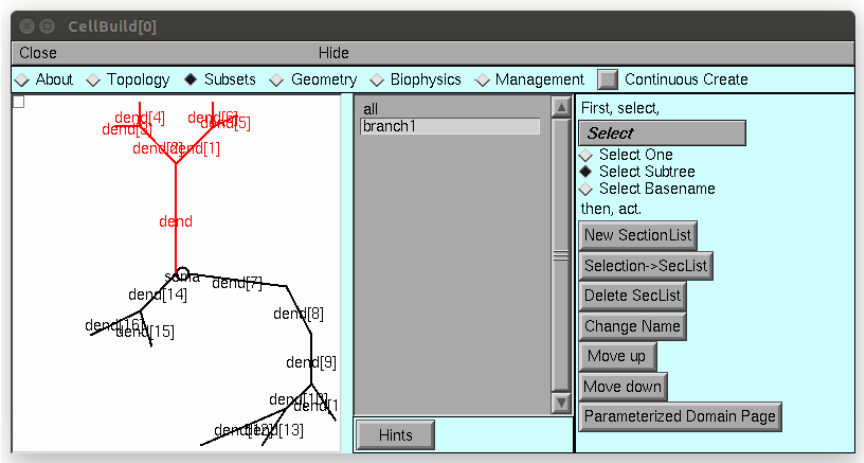
```
from neuron import h, gui
hbox = h.HBox()
hbox.intercept(1)
h.xpanel('Example 1')
h.xlabel('Hello class')
h.xbutton('Click me')
h.xpanel()
h.xpanel('Example 3')
h.xbutton('Say hello')
h.xpanel()
h.xpanel()
hbox.intercept(0)
hbox.map()
```



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

Layout: HBox and VBox

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



Version control with Git

Robert A. McDougal

Yale School of Medicine

Why use version control?

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

`git` is one of the most widely used version control tools today. You can download it from:

https://git-scm.com/

Many people choose to share their git repositories (privately* or publicly) on GitHub.com or BitBucket.org.

Fees may apply for private repositories, but both of these websites provide free exceptions in certain cases, and your university may provide a free alternative

Version control: git basics

Setup

```
git init
```

Stage new/modified files for next commit:

```
git add FILENAME
```

See what has changed

```
git diff
```

See the status of the repo (what files are missing, etc)

```
git status
```

Commit a version (so can return to it later); you will be prompted to enter a commit message:

```
git commit
```

Return to the version of FILENAME from 2 commits ago

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

Version control: git branches

Develop features in branches and then merge back.

Create a new branch:

```
git checkout -b branchname
```

Switch back to an existing branch:

```
git checkout existingbranchname
```

Merge from another branch:

```
git merge otherbranchname
```

Delete a branch:

```
git branch -d branchtoremove
```

Other options for merging branches are available, including `git rebase` and `git merge --squash`.

Version control: git

View list of changes

```
git log
```

Remove a file from tracking

```
git rm FILENAME
```

Rename a tracked file

```
git mv OLDNAME NEWNAME
```

Version control: git and remote servers

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

Clone (download) from a server

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

Clone a specific branch

```
git clone http://URL -b branchname
```

Get changes from server and merge with local changes

```
git pull
```

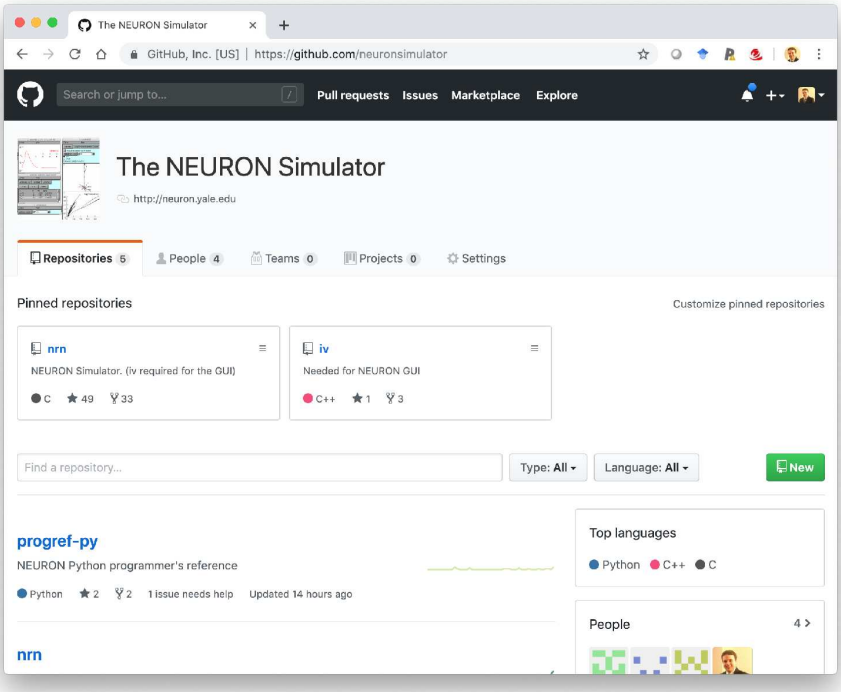
Sync local, committed changes to the server

```
git push
```

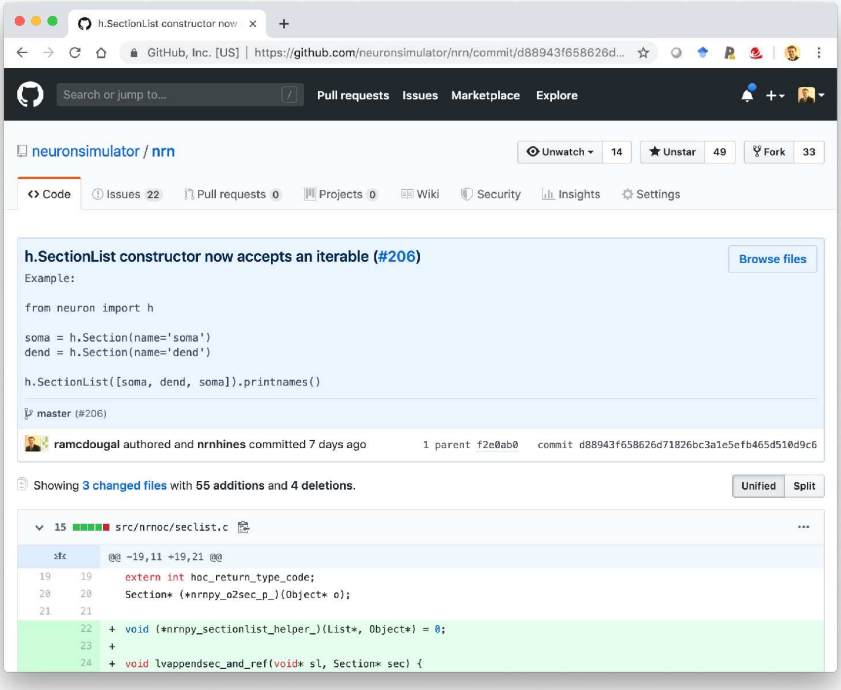
Sync changes on local `master` to a new branch on server

```
git push origin master:remote-branch-name
```


GitHub



GitHub



Version control: syncing data with code

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

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

Then, for example, save matplotlib graphics with:

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


Receipt

Received:

From:

For: NEURON 2019 Summer Course
http://www.neuron.yale.edu/neuron/static/courses/summer2019/summer2019.html

Date:

By: N.T. Carnevale
Director, NEURON 2019 Summer Course
203-494-7381
ted.carnevale@yale.edu

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

Survey

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

Please score these 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	_____		

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++, Python etc.).

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

Which software? _____