Building, Running, and Visualizing Parallel NEURON Models

Robert A. McDougal

Yale School of Medicine

5-6 December 2016

Overview

What is the course?

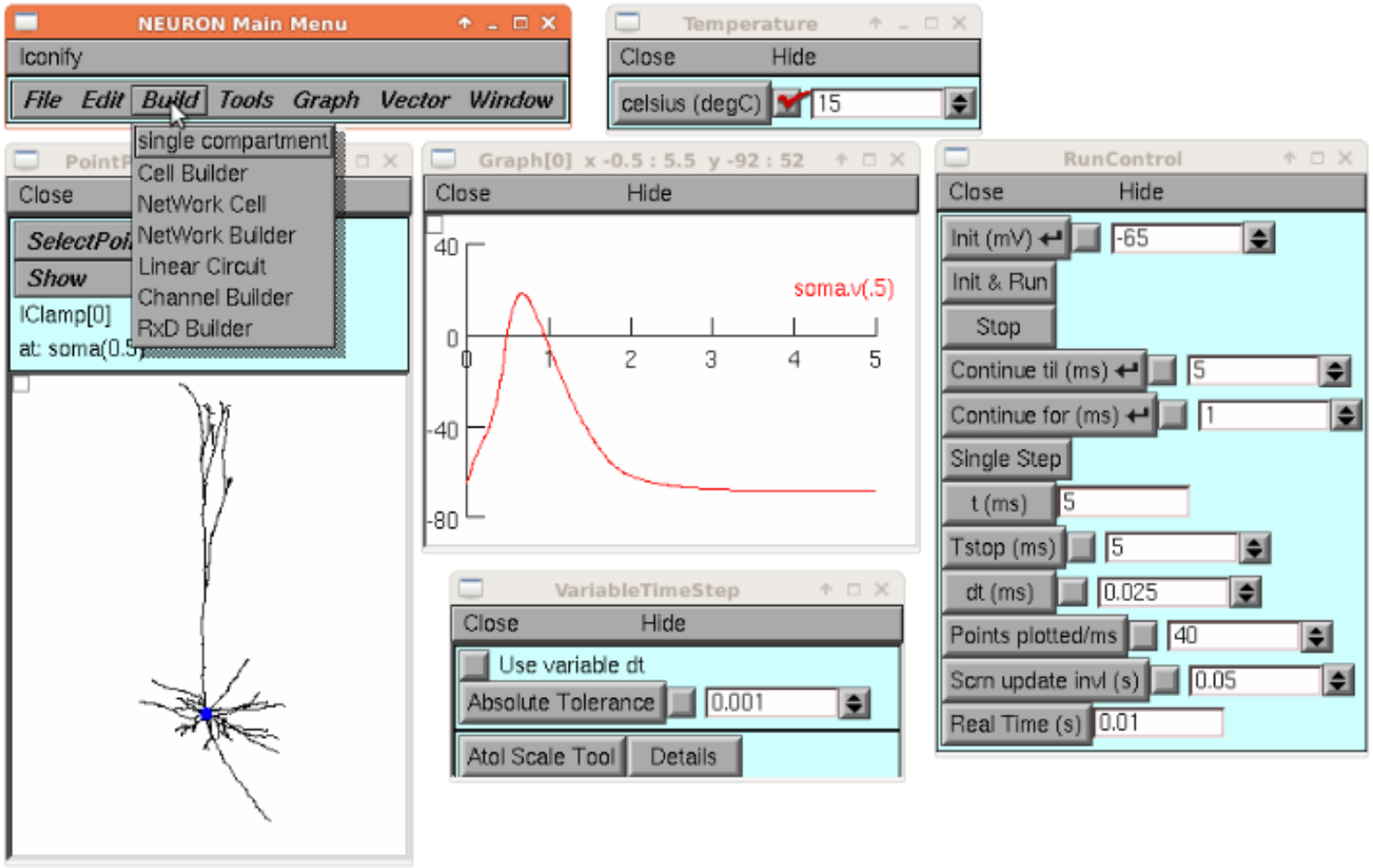
This course will give experienced NEURON modelers the best-practices background needed to design, run, and analyze parallel network models with morphologically detailed neurons.

The guiding philosophy is that a network consists of many instances of cell objects. All cell models should stand alone (for development and analysis purposes), but should be written in a way that they can be combined into parallel network models.

What isn't this course?

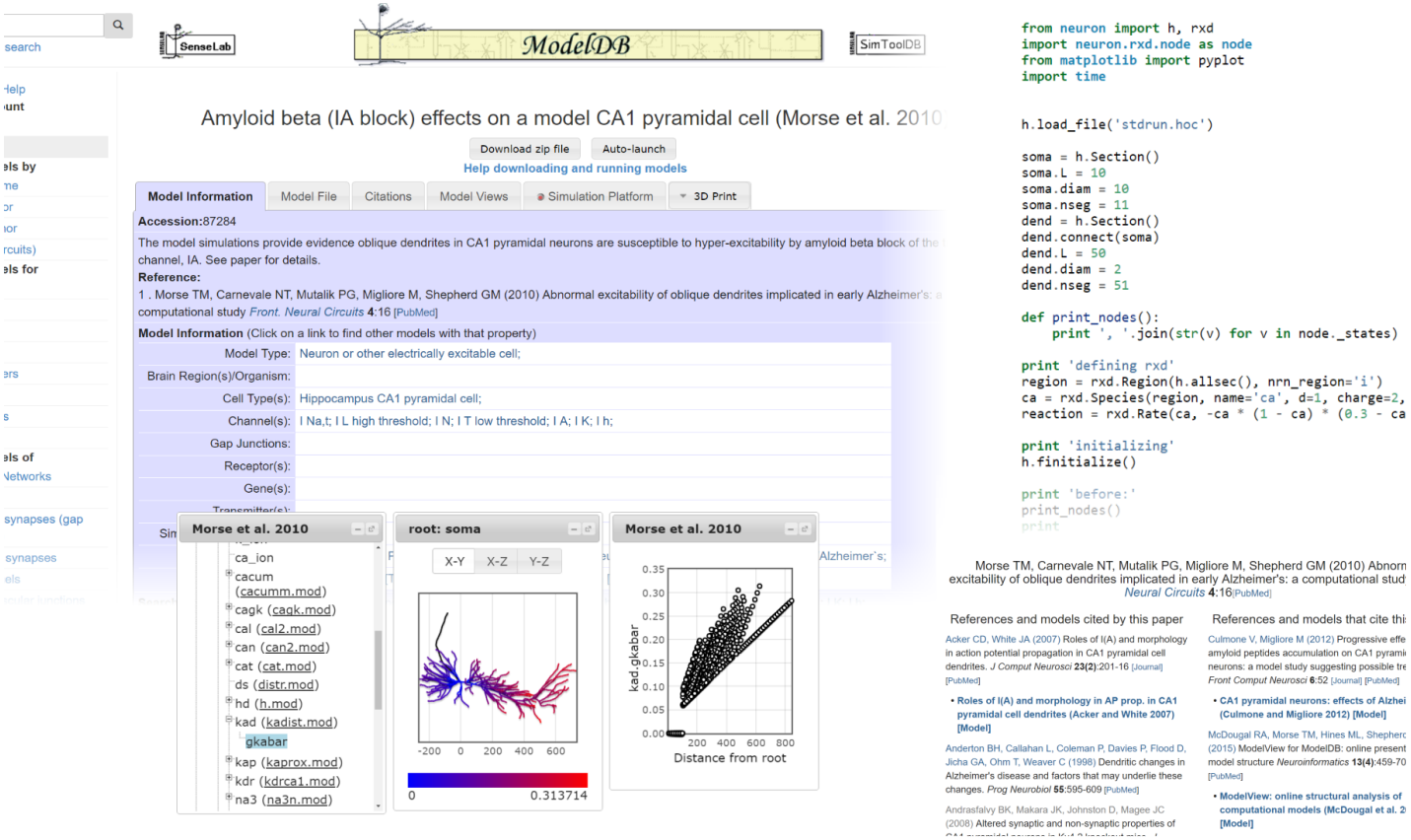
This course is **not**:

- GUI driven.
- A basic introduction to NEURON or parallel programming.
- An exhaustive presentation of every known parallel NEURON strategy.



neuron.yale.edu

Forum: neuron.yale.edu/phpBB
Scripting tutorial: neuron.yale.edu/neuron/static/docs/neuronpython/firststeps.html
List of publications (over 1600) using NEURON: neuron.yale.edu/neuron/static/bib/usednrn.html



ModelDB.yale.edu

ModelDB is a resource for discovery, sharing, and analysis. ModelDB provides source code for approximately **1150 published computational neuroscience models** on **139 topics** with at least **48 types of ion channels**/pumps/etc; code for at least 76 different simulation environments is available.

Why use parallel computation?

Two reasons:

- "Faster" run-time simulations.
- Support for large models that do not fit 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.

Getting started

Connecting to MPI and to NEURON

Connect to MPI:

```
from mpi4py import MPI
```

Connect to NEURON:

```
from neuron import h
```

Connect to NEURON when running locally:

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

Test parallel NEURON

Test script (test.py):

```
from mpi4py import MPI
from neuron import h
pc = h.ParallelContext()
id = int(pc.id())
nhost = int(pc.nhost())
print id, "of", nhost
```

Run with:

```
mpiexec -n 4 python test.py
```

Output (if successful; order may vary**):**

```
0 of 4
1 of 4
3 of 4
2 of 4
```

Note: If instead, you see 4 NEURON headers and 4 "0 of 1" messages, then NEURON has not been compiled with parallel support. Recompile with the `--with-paranrn` flag.

Test parallel NEURON (with Slurm)

Create job file with Slurm options

```
#!/bin/bash
#SBATCH -J test                # Job name
#SBATCH -o job.%j.out          # Name of stdout output file
#SBATCH -n 50                  # Total number of cores
#SBATCH -t 00:01:00            # Run time (hh:mm:ss)
mpiexec python testmpi.py
```

Submit the job

```
sbatch job.mpi
```

Here `job.mpi` is the name of the job file created above.

View output when done

```
cat mysim.460.out
```

Here 460 was the job number returned from the `sbatch` command.

The Slurm Workload Manager is available from https://slurm.schedmd.com/

Building a model (part 1)

Neuronal building block: the Section

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

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

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

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

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

```
True
```

It is **strongly recommended** to **name Sections** and to **identify what cell** they belong to:

```
soma = h.Section(name='soma', cell=myCell)
```

Here `myCell` can be any Python object, but in practice it is best if this is an instance of a cell class.
To access the name or cell, use `.name()` or `.cell()`:

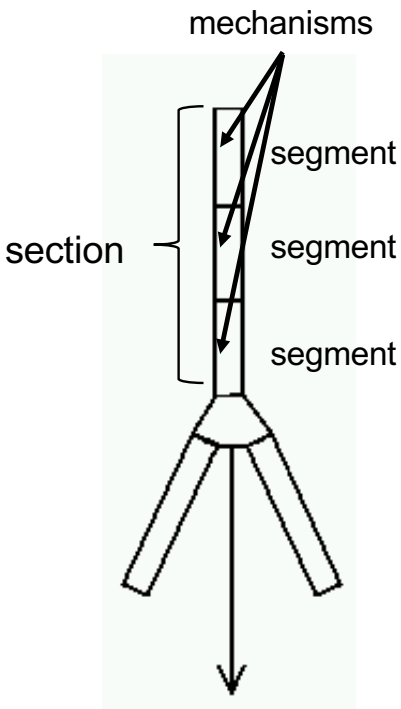
```
print (soma.name())
```

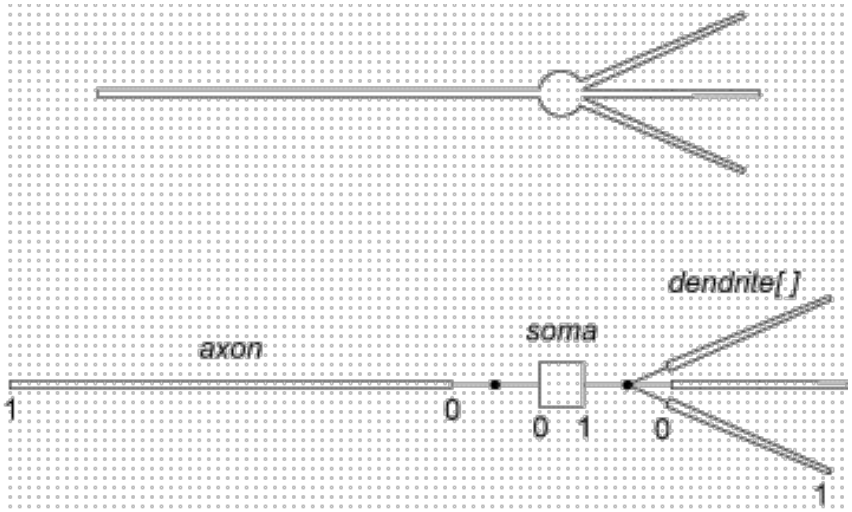
```
soma
```

In recent versions of NEURON, named Sections will print with their name; e.g. it suffices to say `print (soma)`.

Sections and Segments

- Sections are unbranched lengths of continuous cable connected together to form a neuron.
- Do not confuse sections with segments!
- Sections are divided into segments of equal length for numerical simulation purposes (see nseg).





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: Define a cell inside a class

It will be convenient to assign an identifier (gid), specify morphology in a dedicated method, and add a `__repr__` method to identify the object.

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

Length and diameter

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

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

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

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

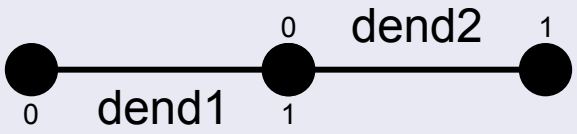
Warning: the default diameter is based on a squid giant axon and is not appropriate for modeling mammalian cells. Likewise, the temperature (`h.celsius`) is by default 6.3 degrees (appropriate for squid, but not for mammals).

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

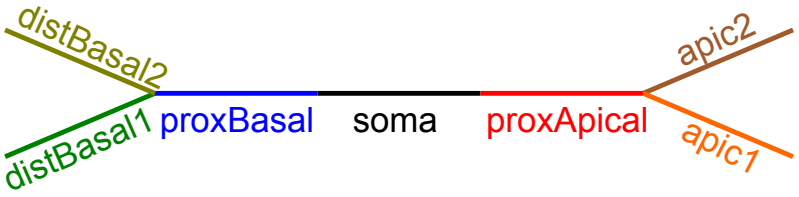
Python script:

```
from neuron import h, gui

class Pyramidal:
    def __init__(self, gid):
        self._gid = gid
        self._setup_morphology()
    def _setup_morphology(self):
        self.soma = self._section('soma')
        self.papic = self._section('papic')
        self.apic1, self.apic2, self.pb, self.db1, self.db2 = [
            self._section(name) for name in
                ['apic1', 'apic2', 'pb', 'db1', 'db2']]
        self.papic.connect(self.soma)
        self.pb.connect(self.soma(0))
        self.apic1.connect(self.papic)
        self.apic2.connect(self.papic)
        self.db1.connect(self.pb)
        self.db2.connect(self.pb)
    def _section(self, name):
        return h.Section(name=name, cell=self)
    def __repr__(self):
        return 'p[%d]' % self._gid

myPyramidal = Pyramidal(0)
h.topology()

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

Output:

```
|-|       p[0].soma(0-1)
    `|      p[0].proxApical(0-1)
      `|      p[0].apic1(0-1)
      `|      p[0].apic2(0-1)
  `|      p[0].proxBasal(0-1)
    `|      p[0].distBasal1(0-1)
    `|      p[0].distBasal2(0-1)
```

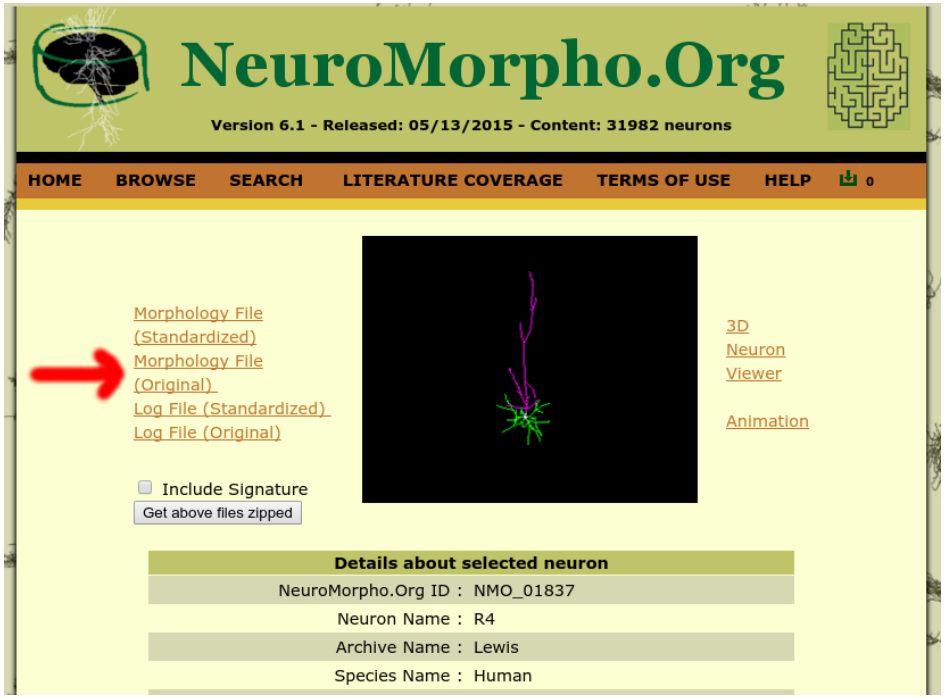
Morphology:

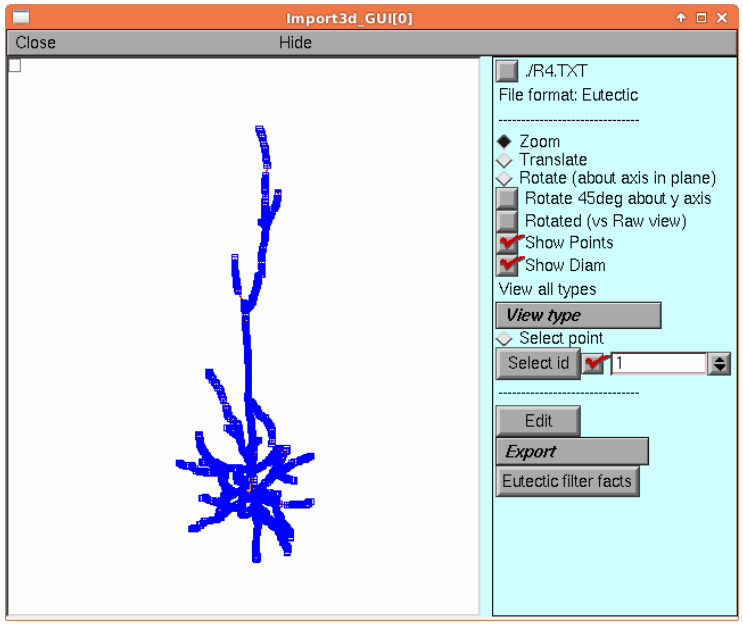


Note: `PlotShape` can also be used to see the distribution of a parameter or calculated variable. To save the image in plot shape ps use `ps.printfile('filename.eps')`

The PlotShape window can be opened from the GUI by selecting Graph - Shape Plot.

NeuroMorpho.Org for realistic morphologies





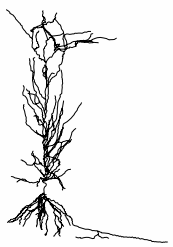
Tools ▶ Miscellaneous ▶ Import 3D

- NeuroMorpho.Org is home to 50,356 reconstructed neurons from 212 cell types and 37 species as of October 24, 2016.
- **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

Exercise

Download and examine the following three CA1 pyramidal cell morphologies (use the "standardized" version). Which is most appropriate for simulation?

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



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

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

DO. 10.5281/zenodo.12576 launch binder

PyNeuron-Toolbox

To see a live demo in a Jupyter Notebook: CLICK HERE

The NEURON simulation environment is one of the most popular options for simulating multi-compartment neuron models. Hines et al. (2009) developed a module that allowed users to execute simulations from python. This option appears to be very popular with users.

However, much of the data analysis capabilities of NEURON (e.g. shape plots) are still limited to the traditional InterViews plotting environment. This toolbox provides some functions to do data analysis and visualization in matplotlib. One of the advantages of this approach is that plots and animations can be easily shared with other researchers in iPython notebooks.

Disclaimer: This code is only a side project at the moment. Use with caution and let me know if you find any unexpected behaviors. Feature requests are also welcome.

https://github.com/ahwillia/PyNeuron-Toolbox

```
git clone https://github.com/ahwillia/PyNeuron-Toolbox.git
```

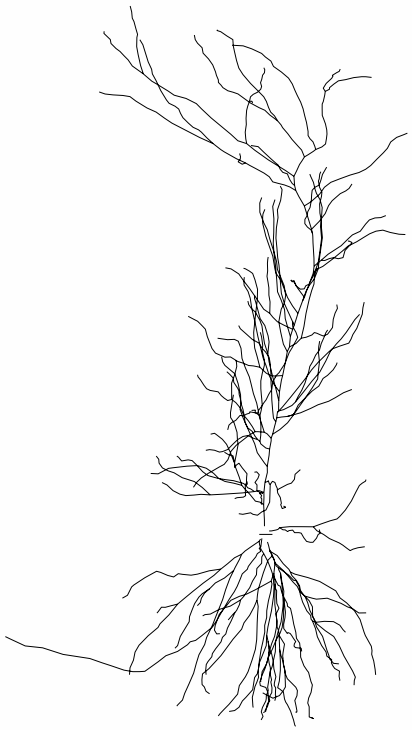
Either install or copy the PyNeuron-Toolbox folder into the project directory.

Loading a morphology with PyNeuron-Toolbox

Python script:

```
from neuron import h, gui
from PyNeuronToolbox import morphology

class Pyramidal:
    def __init__(self, gid):
        self._gid = gid
        self._setup_morphology()
    def _setup_morphology(self):
        self.soma, self.axon = [], []
        self.dend, self.apic = [], []
        morphology.load('c91662.swc', fileformat='swc',
            cell=self)
    def __repr__(self):
        return 'p[%d]' % self._gid

myPyramidal = Pyramidal(0)

ps = h.PlotShape()
ps.show(1)
```

Output:



Built-in support for instantiating morphologies into Python class is available in NEURON 7.5, the current development version.

Aside: version control

Version control: git

Why use version control?

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

Version control: git basics

Setup

```
git init
```

Declare files to be tracked

```
git add FILENAME
```

Commit a version (so can return to it later)

```
git commit -a
```

Return to the version of FILENAME from 2 commits ago

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

Version control: git

View list of changes

```
git log
```

Remove a file from tracking

```
git rm FILENAME
```

Rename a tracked file

```
git mv OLDNAME NEWNAME
```

Version control: git and remote servers

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

Download from a server

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

Get changes from server and merge with local changes

```
git pull
```

Sync local, committed changes to the server

```
git push
```

Version control: syncing data with code

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

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

Then, for example, save matplotlib graphics with:

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

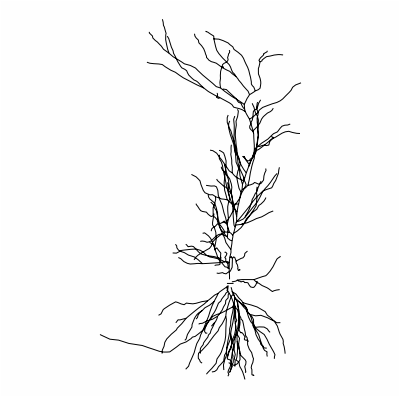
Building a model (part 2)

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__`:

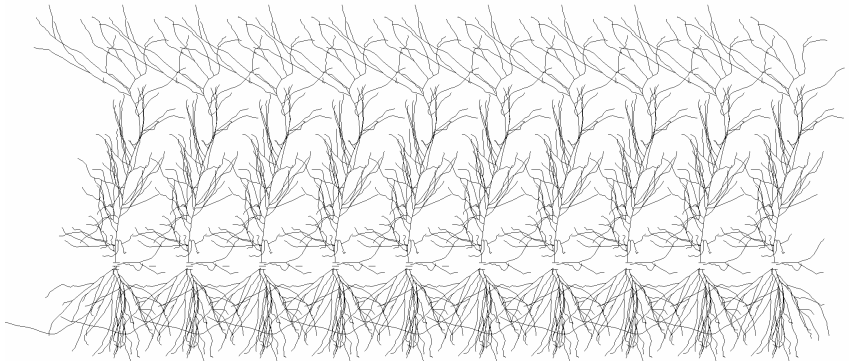
```
class Pyramidal:
  def _shift(self, x, y, z):
    for sec in self.all:
      n = int(h.n3d(sec=sec))
      xs = [h.x3d(i, sec=sec) for i in range(n)]
      ys = [h.y3d(i, sec=sec) for i in range(n)]
      zs = [h.z3d(i, sec=sec) for i in range(n)]
      ds = [h.diam3d(i, sec=sec) for i in range(n)]
      i = 0
      for a, b, c, d in zip(xs, ys, zs, ds):
        h.pt3dchange(i, a + x, b + y, c + z, d, sec=sec)
        i += 1
```

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

def _setup_morphology(self):
  self.soma, self.axon = [], []
  self.dend, self.apic = [], []
  morphology.load('c91662.swc',
    fileformat='swc',
    cell=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.

Ion channel specification: NMODL

- Used for low-level mechanisms (e.g. synapses, integrate-and-fire cells) that need to be fast
- Translated directly into C code, then compiled
- Good for:
  - Being really fast
  - Handling nuisances like units
  - Being *really* fast
- Bad because:
  - Steep learning curve
  - `VERBATIM` blocks can be scary

Ion channel specification: NMODL

- Example: voltage-gated K^+ channel

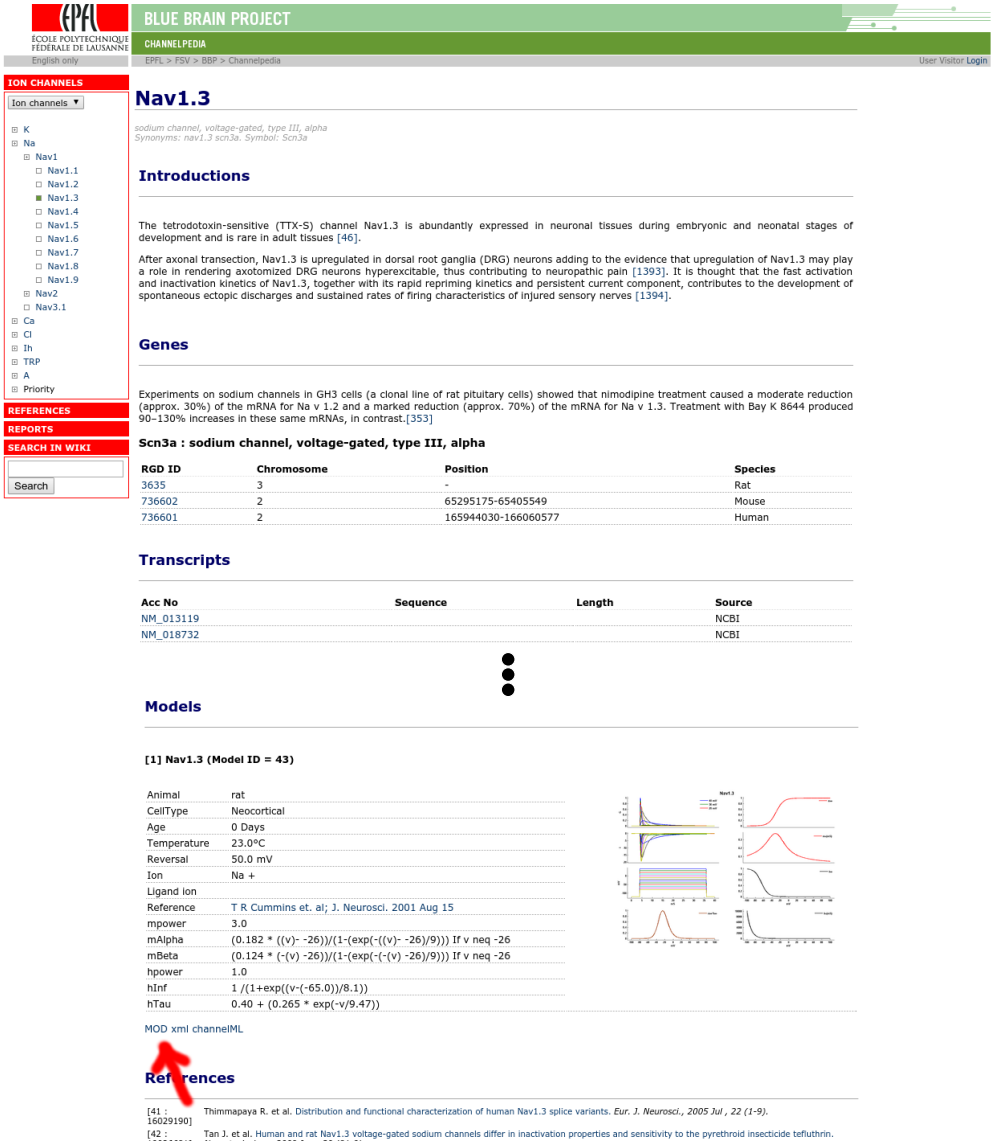
```
NEURON {
   SUFFIX kd
   USEION k READ ek WRITE ik
   RANGE gbar, g, i
}

UNITS {
   (S) = (siemens)
   (mV) = (millivolt)
   (mA) = (milliamp)
}

PARAMETER { gbar = 0.036 (S/cm2) }

ASSIGNED {
   v (mV)
   ek (mV)
   ik (mA/cm2)
   i (mA/cm2)
   g (S/cm2)
}

BREAKPOINT {
   SOLVE states METHOD cnexp
   g = gbar * n^4
   i = g * (v - ek)
   ik = i
}

INITIAL {
   n = alpha(v)/(alpha(v) + beta(v))
}

DERIVATIVE states {
   n' = (1-n)*alpha(v) - n*beta(v)
}

FUNCTION alpha(Vm (mV)) (/ms) {
   LOCAL x
   UNITSOFF
   x = (Vm+55)/10
   if (fabs(x) > 1e-6) {
      alpha = 0.1*x/(1 - exp(-x))
   } else {
      alpha = 0.1/(1 - 0.5*x)
   } UNITSON
}

FUNCTION beta(Vm (mV)) (/ms) {
   UNITSOFF
   beta = 0.125*exp(-(Vm+65)/80)
   UNITSON
}
```

Ion channel specification

In addition to NMODL, two other options allow compiled-speed ion channels:

- Channels may be defined with NEURON's ChannelBuilder tool.
  - This instantiates `KSChan` objects, which define the channel.
- Channels may be defined using LEMS (NeuroML) and converted to NMODL via jNeuroML.

Channelpedia (Channelpedia.epfl.ch)



BLUE BRAIN PROJECT

CHANNELPEDIA

English only | EPFL > FSV > BBP > Channelpedia | User Visitor Login

ION CHANNELS

Ion channels

- K
- Na
  - Nav1
    - Nav1.1
    - Nav1.2
    - Nav1.3
    - Nav1.4
    - Nav1.5
    - Nav1.6
    - Nav1.7
    - Nav1.8
    - Nav1.9
  - Nav2
  - Nav3.1
- Ca
- Cl
- Ih
- TRP
- A
- Priority

REFERENCES

REPORTS

SEARCH IN WIKI

Search

Nav1.3

sodium channel, voltage-gated, type III, alpha
Synonyms: nav1.3 scn3a. Symbol: Scn3a

Introductions

The tetrodotoxin-sensitive (TTX-S) channel Nav1.3 is abundantly expressed in neuronal tissues during embryonic and neonatal stages of development and is rare in adult tissues [46].

After axonal transection, Nav1.3 is upregulated in dorsal root ganglia (DRG) neurons adding to the evidence that upregulation of Nav1.3 may play a role in rendering axotomized DRG neurons hyperexcitable, thus contributing to neuropathic pain [1393]. It is thought that the fast activation and inactivation kinetics of Nav1.3, together with its rapid repriming kinetics and persistent current component, contributes to the development of spontaneous ectopic discharges and sustained rates of firing characteristics of injured sensory nerves [1394].

Genes

Experiments on sodium channels in GH3 cells (a clonal line of rat pituitary cells) showed that nimodipine treatment caused a moderate reduction (approx. 30%) of the mRNA for Na v 1.2 and a marked reduction (approx. 70%) of the mRNA for Na v 1.3. Treatment with Bay K 8644 produced 90–130% increases in these same mRNAs, in contrast.[353]

Scn3a : sodium channel, voltage-gated, type III, alpha

RGD ID	Chromosome	Position	Species
3635	3	-	Rat
736602	2	65295175-65405549	Mouse
736601	2	165944030-166060577	Human

Transcripts

Acc No	Sequence	Length	Source
NM_013119			NCBI
NM_018732			NCBI

Models

[1] Nav1.3 (Model ID = 43)

Animal	rat
CellType	Neocortical
Age	0 Days
Temperature	23.0°C
Reversal	50.0 mV
Ion	Na +
Ligand ion	
Reference	T R Cummins et. al; J. Neurosci. 2001 Aug 15
mpower	3.0
mAlpha	(0.182 * ((v)- -26))/(1-(exp(-((v)- -26)/9))) If v neq -26
mBeta	(0.124 * (-(v) -26))/(1-(exp(-(-(v) -26)/9))) If v neq -26
hpower	1.0
hInf	1 /(1+exp((v-(-65.0))/8.1))
hTau	0.40 + (0.265 * exp(-v/9.47))

MOD xml channelML

References

[41 : 16029190] Thimmapaya R. et al. Distribution and functional characterization of human Nav1.3 splice variants. *Eur. J. Neurosci., 2005 Jul , 22 (1-9).*

[42 : Tan J. et al. Human and rat Nav1.3 voltage-gated sodium channels differ in inactivation properties and sensitivity to the pyrethroid insecticide tefluthrin.

- 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.

ModelDB (modeldb.yale.edu)

ATP-senstive potassium current	
Ca pump	
Channelrhodopsin (ChR)	
I ANO2	
I Calcium	I Ca,p
	I L high threshold
	I N
	I p,q
	I Q
	I R
	I T low threshold
I Chloride	I Cl, leak
	I Cl,Ca
I MI	
I Mixed	I CAN
	I CNG
	I h
	I K,Ca
I Potassium	I A
	I A, slow
	I K
	I K,leak
	I Krp
	I M
	I_AHP
	I_K,Na
	I_KD
	I_KHT
	I_KLT
	I_Ks
	KCNQ1
	Kir
	Kir2 leak
I Sodium	I Na, leak
	I Na,p
	I Na,t
	I_Na,Ca
	Late Na
I trp	
I TRPM8	
I_HCO3	
I_HERG	
I_SERCA	
KCC2	
Na/Ca exchanger	
Na/K pump	
NKCC1	
Osmosis-driven water flux	

Calcium response prediction in the striatal spines depending on input timing (Nakano et al. 2013)

Download zip file | Auto-launch

Help downloading and running models

Model Information | Model File | Citations | Model Views | Versions

Accession:151458

We construct an electric compartment model of the striatal medium spiny neuron with a realistic morphology and predict the calcium responses in the synaptic spines with variable timings of the glutamatergic and dopaminergic inputs and the postsynaptic action potentials. The model was validated by reproducing the responses to current inputs and could predict the electric and calcium responses to glutamatergic inputs and back-propagating action potential in the proximal and distal synaptic spines during up and down states.

Reference:

1 . Nakano T, Yoshimoto J, Doya K (2013) A model-based prediction of the calcium responses in the striatal synaptic spines depending on the timing of cortical and dopaminergic inputs and post-synaptic spikes. *Front Comput Neurosci* **7**:119 [PubMed]

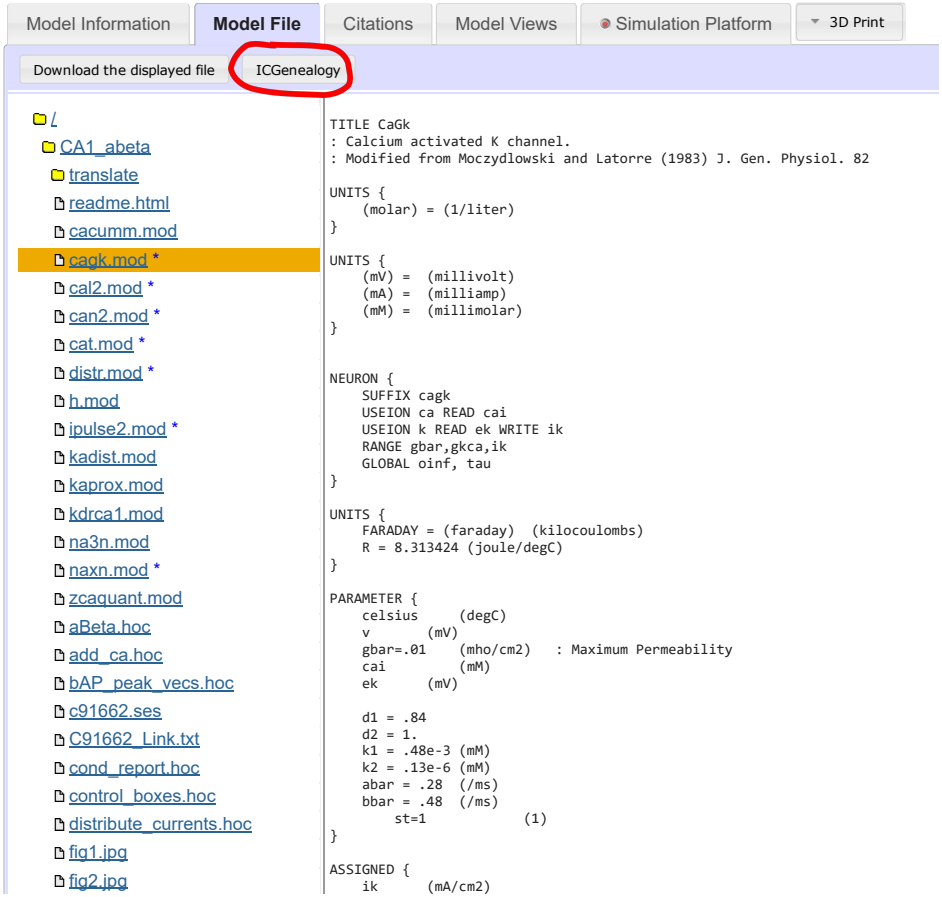
Model Information (Click on a link to find other models with that property)

Model Type:	Neuron or other electrically excitable cell; Synapse;
Brain Region(s)/Organism:	
Cell Type(s):	Neostriatum spiny direct pathway neuron;
Channel(s):	I Na,p; I Na,t; I L high threshold; I A; I K; I K,leak; I K,Ca; I CAN; I Sodium; I Calcium; I Potassium; I A, slow; I Krp; I R; I Q; I Na, leak; I Ca,p; Ca pump;
Gap Junctions:	
Receptor(s):	D1; AMPA; NMDA; Glutamate; Dopaminergic Receptor; IP3;
Gene(s):	
Transmitter(s):	
Simulation Environment:	NEURON;
Model Concept(s):	Reinforcement Learning; STDP; Calcium dynamics; Reward-modulated STDP;
Implementer(s):	Nakano, Takashi [nakano.takashi at gmail.com];

Search NeuronDB for information about: Neostriatum spiny direct pathway neuron; D1; AMPA; NMDA; Glutamate; Dopaminergic Receptor; IP3; I Na,p; I Na,t; I L high threshold; I A; I K; I K,leak; I K,Ca; I CAN; I Sodium; I Calcium; I Potassium; I A, slow; I Krp; I R; I Q; I Na, leak; I Ca,p; Ca pump;

ModelDB offers links to models with many channel types, but they are classified by **publication** not **channel**, so you will have to locate the specific file you need.

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

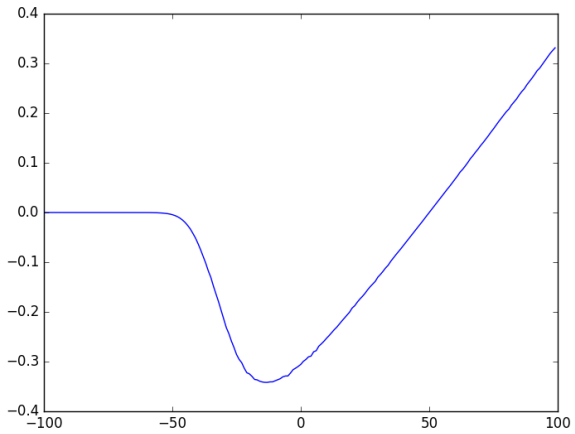
When viewing most `mod` files describing an ion channel, an ICGenealogy button appears. Clicking this button loads the corresponding page of the ICGenealogy database which shows curated information about the channel model (how it was derived, information about the underlying data, etc) and response curves.

Always review dynamics you borrow

Suppose you found a mechanism on ModelDB with `SUFFIX na3`.

After compiling it (`nrnivmodl`), you can use PyNeuron-Toolbox to examine its I-V curve:

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

h.CVode().active(1)
ina, v = channel_analysis.ivcurve('na3', 'ina')
pyplot.plot(v, ina)
pyplot.show()
```



https://senselab.med.yale.edu/ModelDB/showmodel.cshtml?model=87284&file=/CA1_abeta/na3n.mod

To see additional options for IV-curve analysis, do `help(channel_analysis.ivcurve)`

Inserting distributed mechanisms

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

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

Inserting point processes

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

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

To find the segment containing a point process pp, use

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

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

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

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

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

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.

The HOC instruction `for(x)` includes the 0 and 1 voltage-only nodes when iterating and is equivalent to using `section.allseg()`.

Example: 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):
        self.soma, self.axon = [], []
        self.dend, self.apic = [], []
        morphology.load('c91662.swc', fileformat='swc', cell=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.

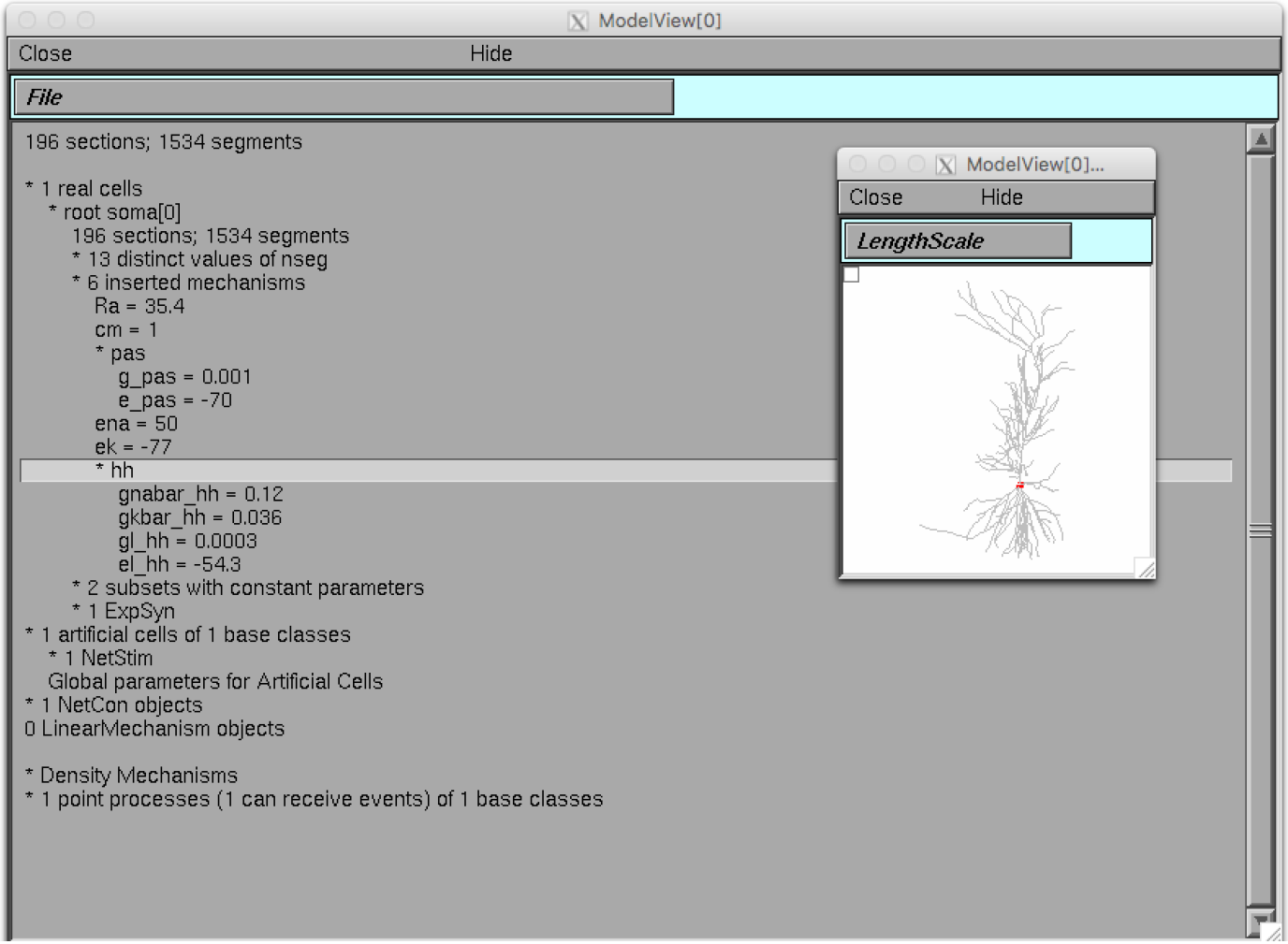
When refining a mesh, multiply by an odd number to preserve old nodes.

```
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
```

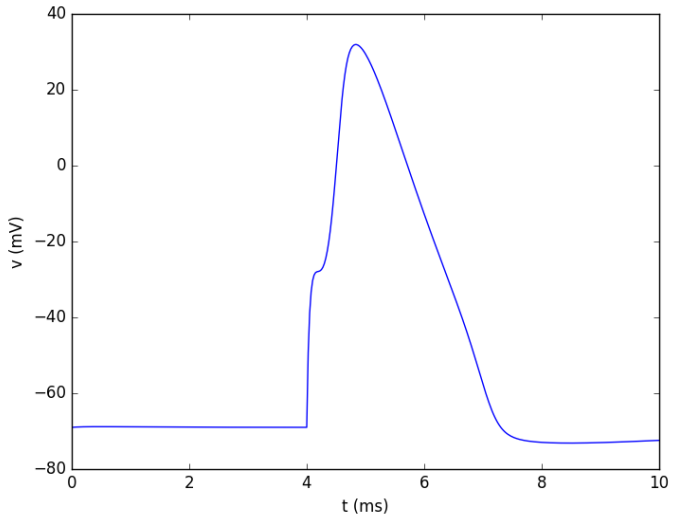
Tools → ModelView



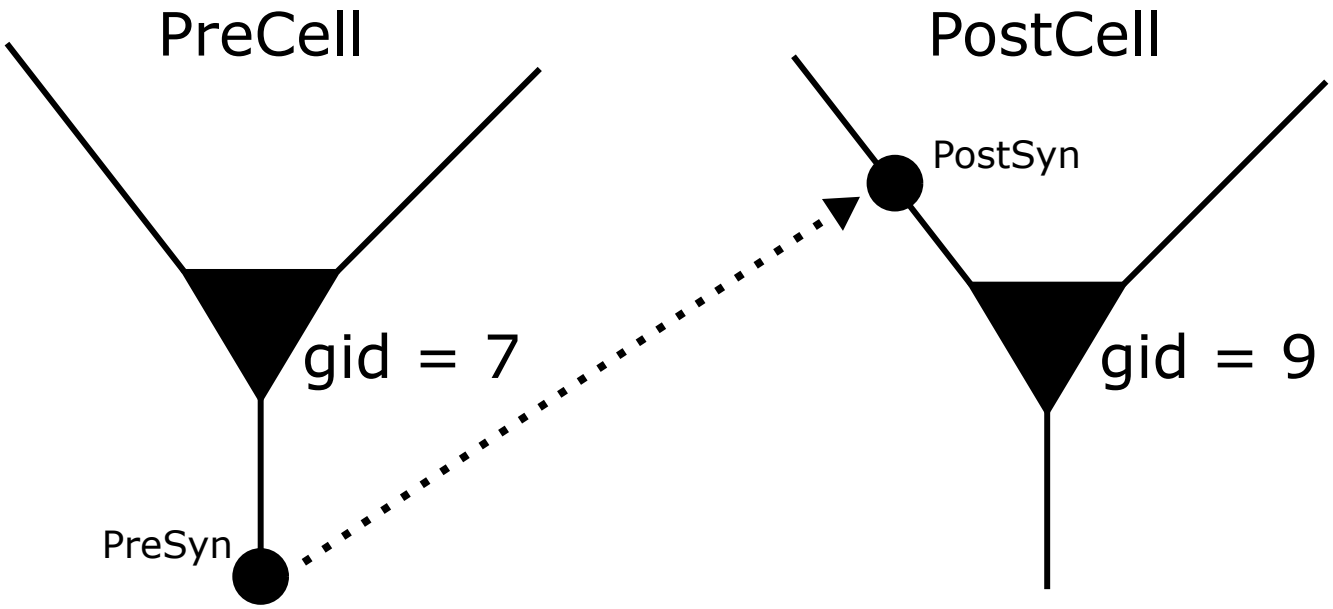
Example: adding a synapse, giving it artificial stimulation, recording data, running simulation

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

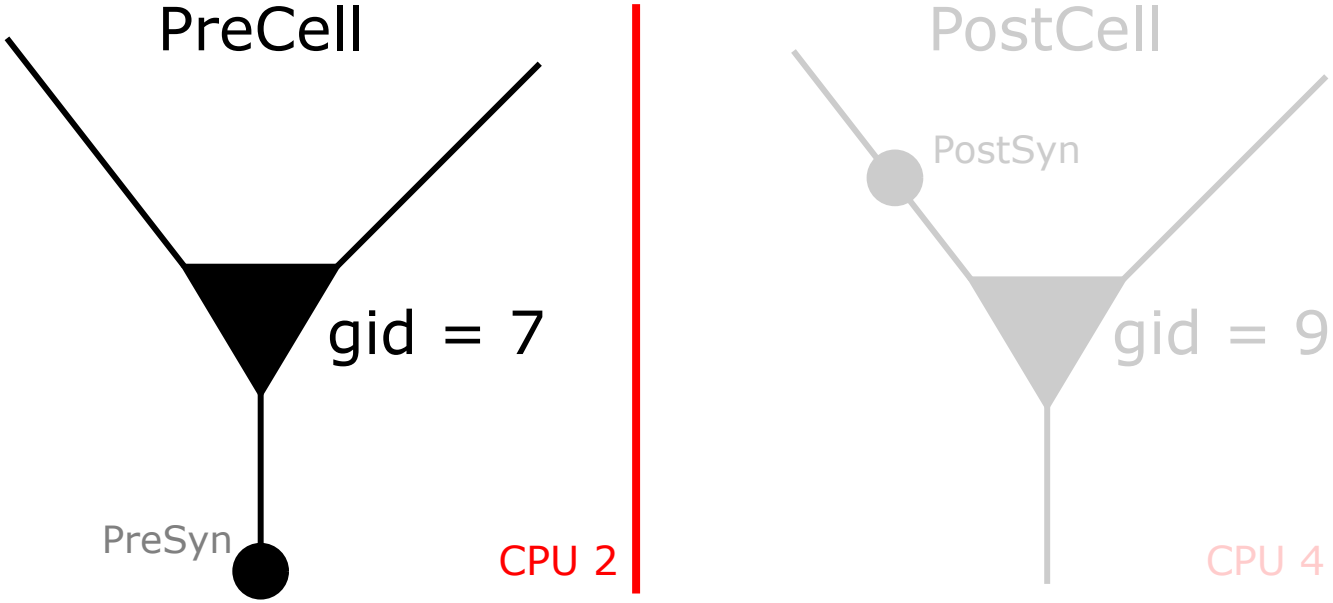
```
pc = h.ParallelContext()
pc.set_maxstep(10)
h.v_init = -69
h.stdinit()
pc.psolve(10)

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:

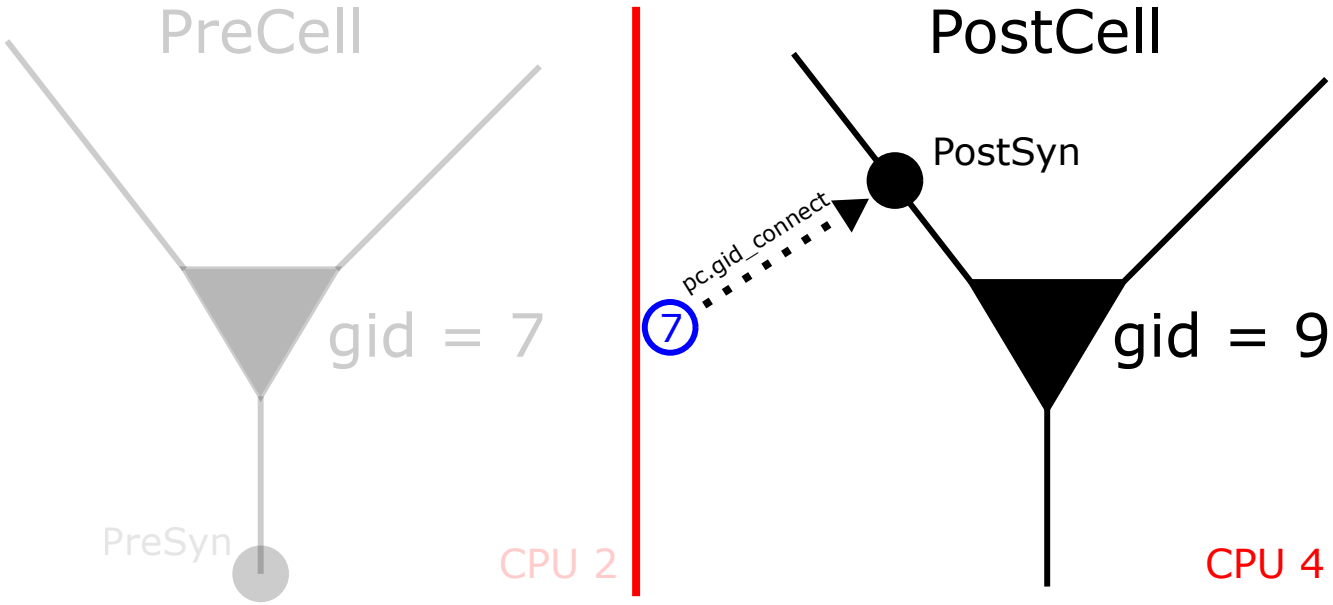
```
if pc.gid_exists(7):
    PreCell = Cell()
```

Associate gid with spike source:

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

PreSyn here is a **pointer**.

Configuring the postsynaptic connection site

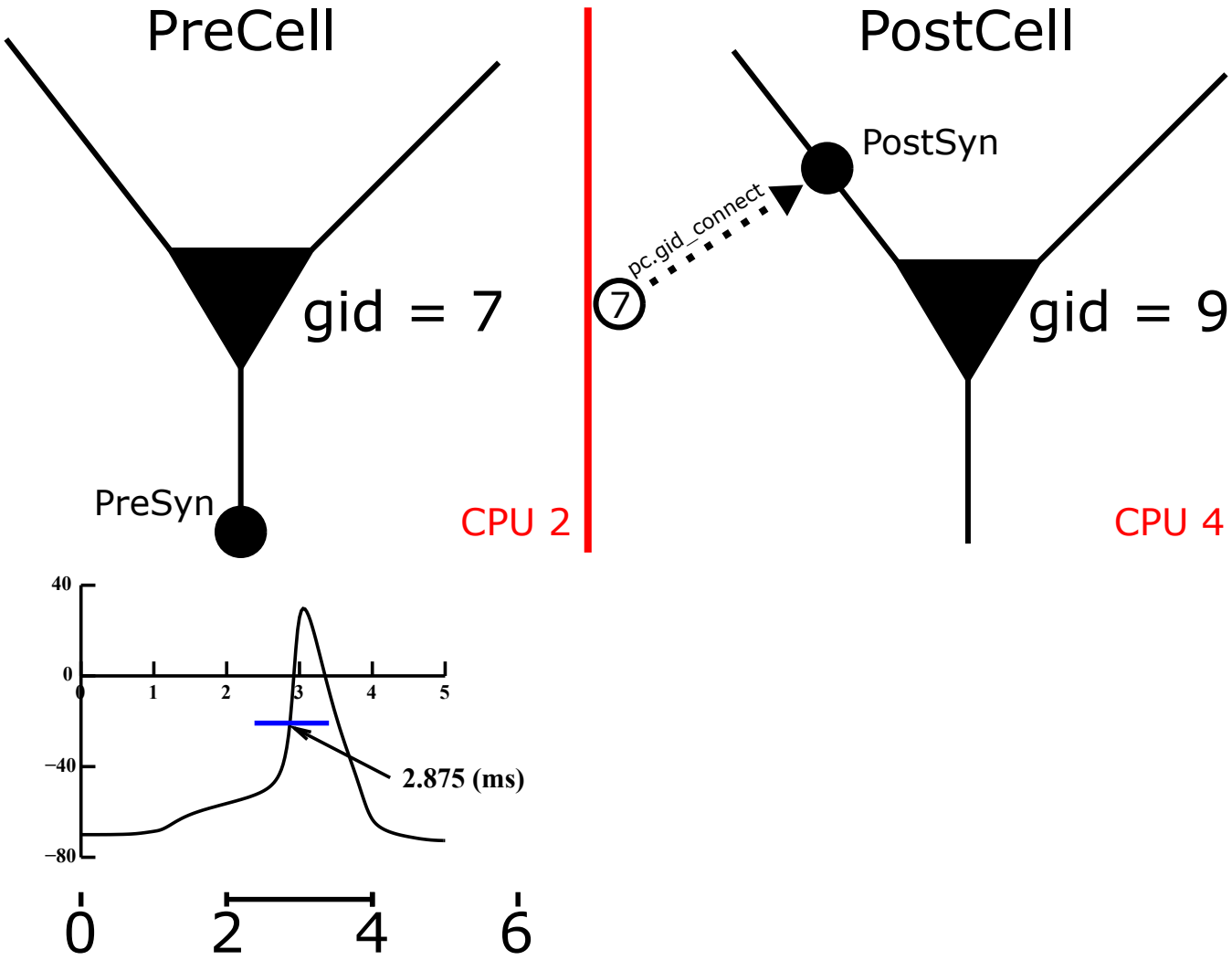


Create NetCon on node where target exists:

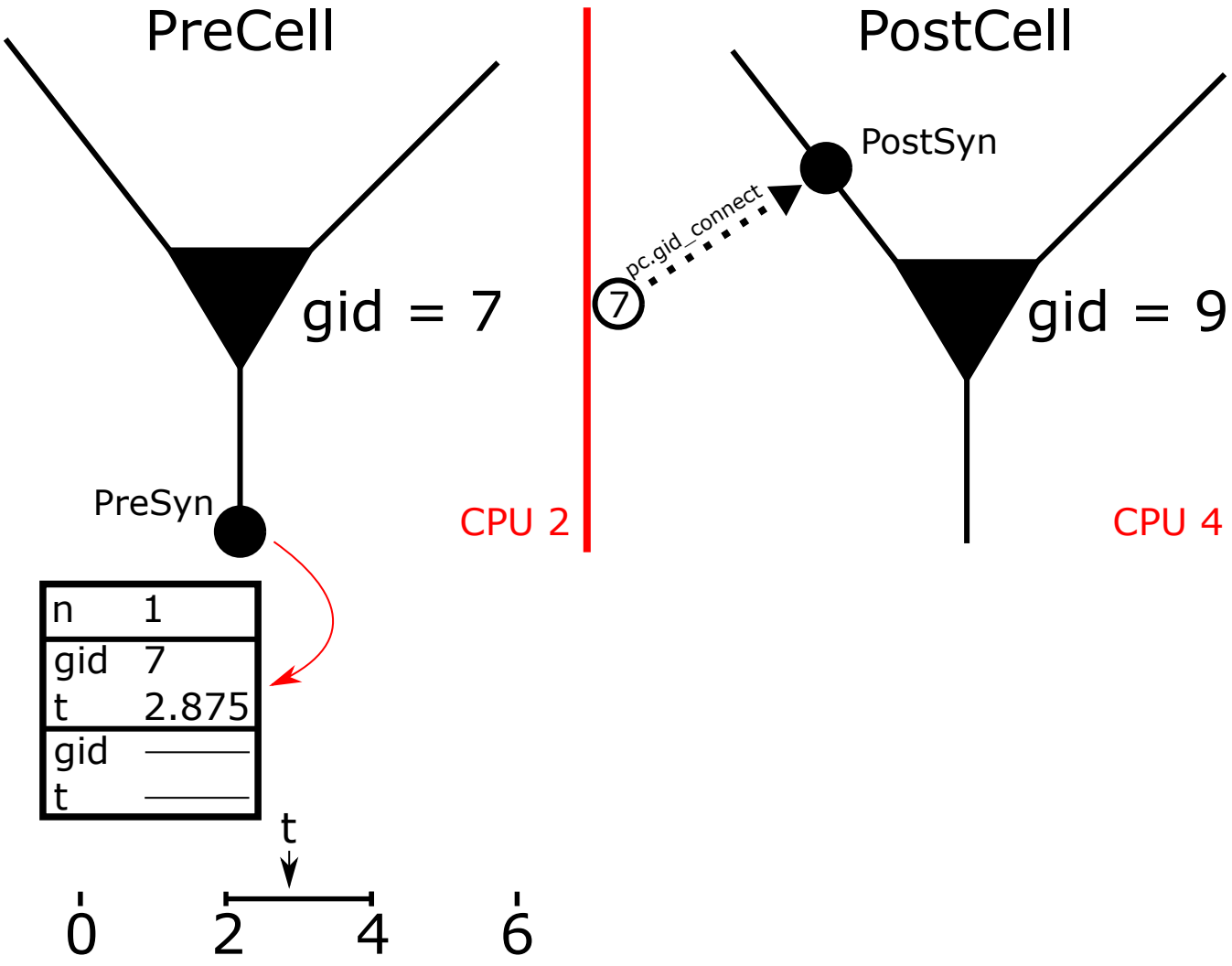
```
nc = pc.gid_connect(7, PostSyn)
```

PostSyn here is a Point Process, e.g. an ExpSyn.

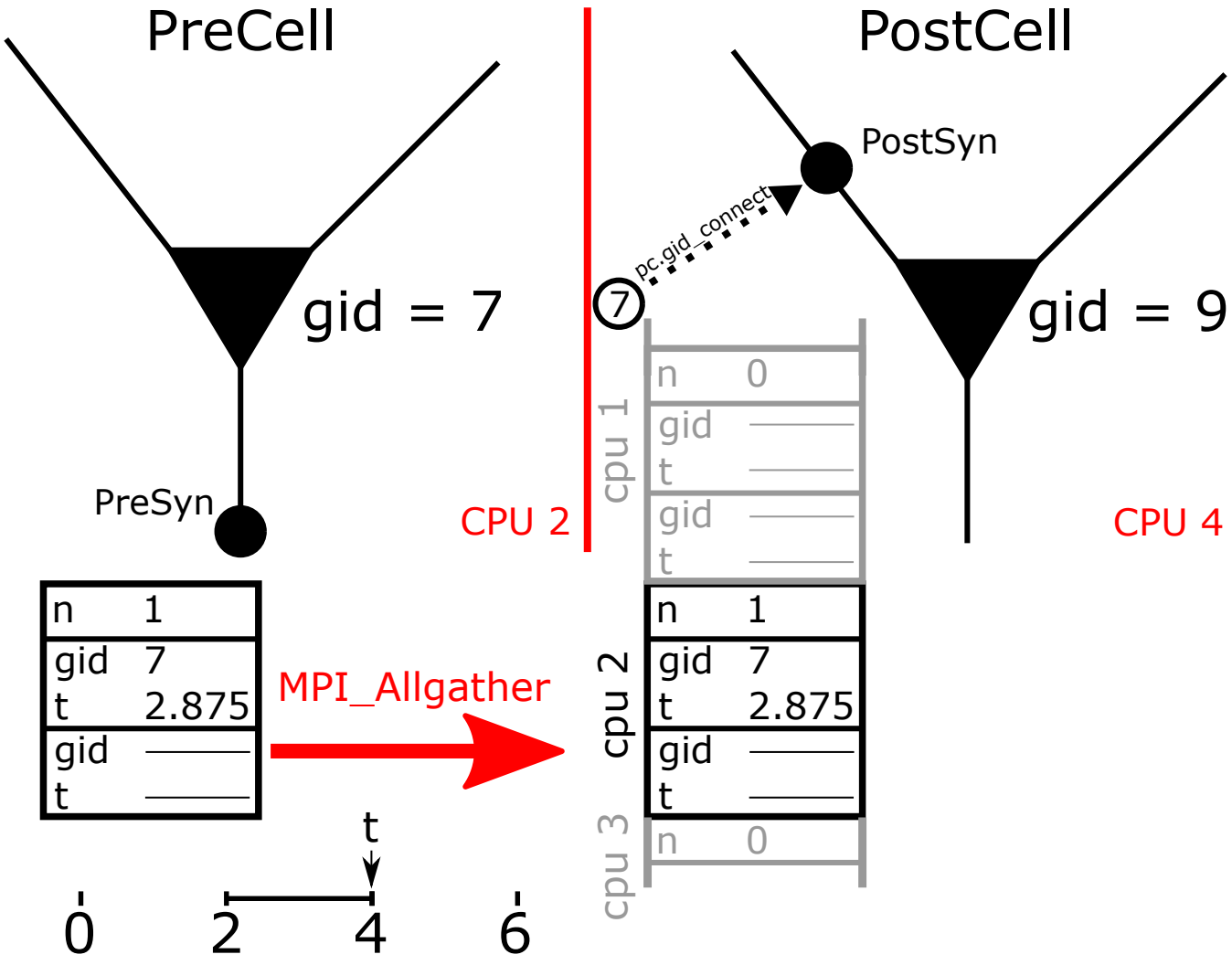
Spike exchange method



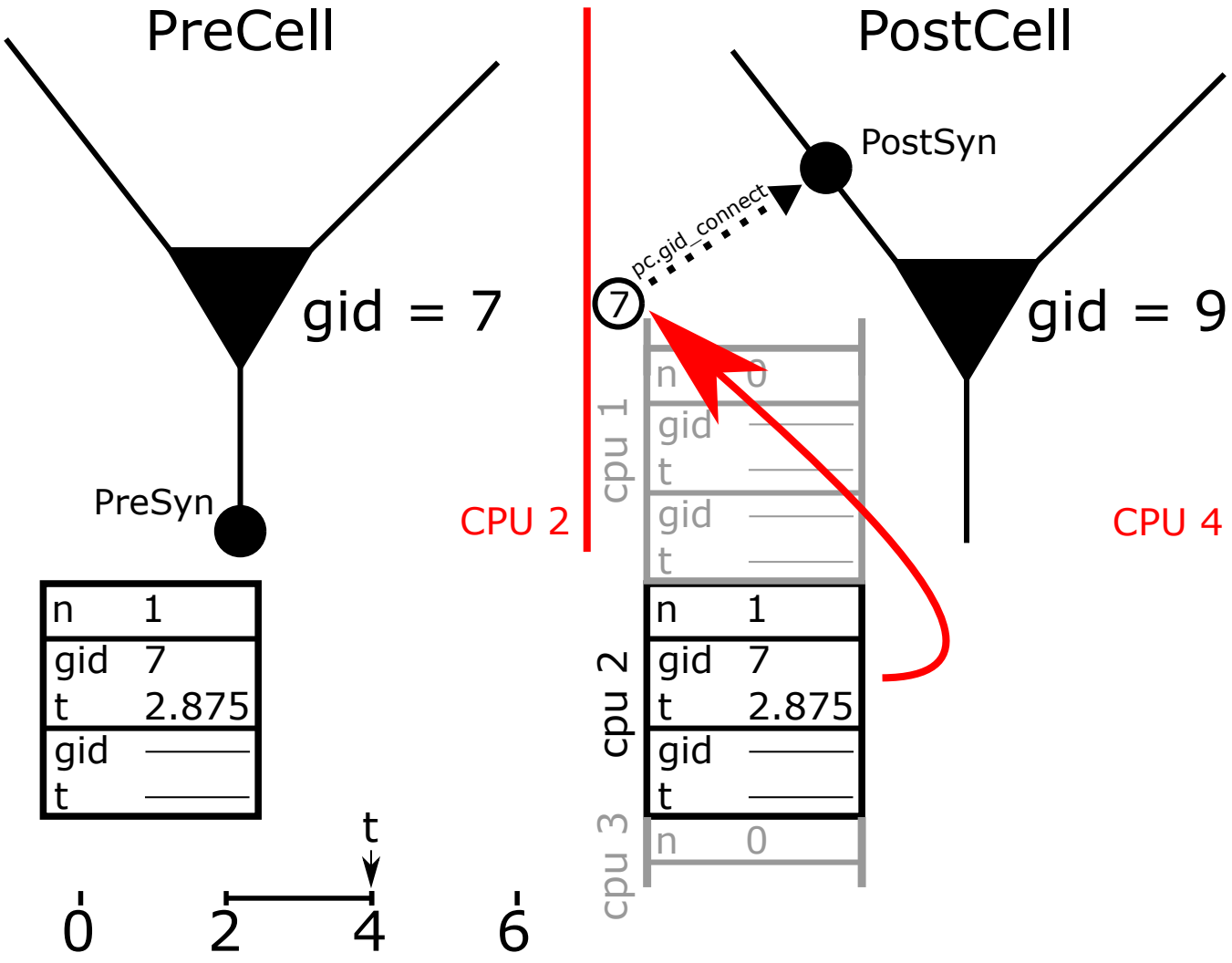
Spike exchange method



Spike exchange method



Spike exchange method

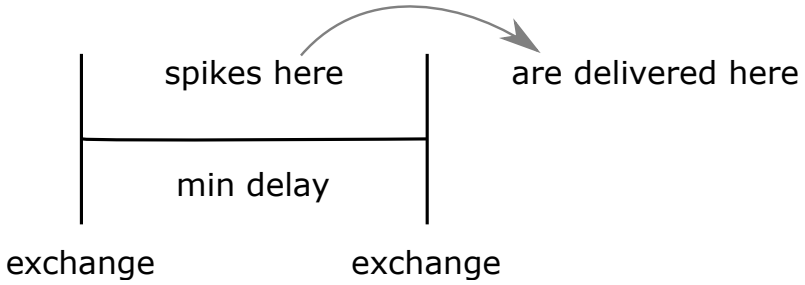


Exploit transmission delays: using pc.set_maxstep

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, int(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`.

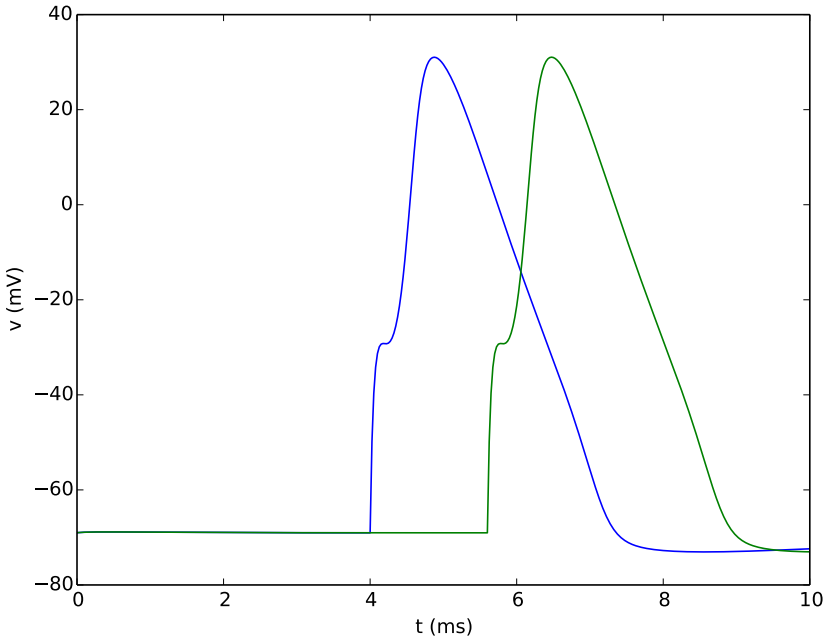
Building a two cell network

```
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
        # 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
        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.

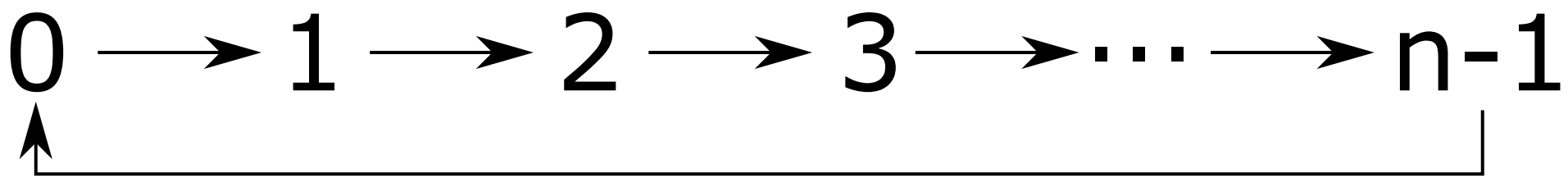
Running the two cell network

```
# drive the 0th cell
stim = h.NetStim()
stim.number = 1
stim.start = 3
ncstim = h.NetCon(stim, n.syns[0])
ncstim.delay = 1
ncstim.weight[0] = 1

t = h.Vector()
t.record(h._ref_t)
v = [h.Vector() for cell in n.cells]
for myv, cell in zip(v, n.cells):
    myv.record(cell.soma[0](0.5)._ref_v)

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

for myv in v:
    pyplot.plot(t, myv)
pyplot.xlabel('t (ms)')
pyplot.ylabel('v (mV)')
pyplot.show()
```



Exercise: Generalizing to n cells in a ring network

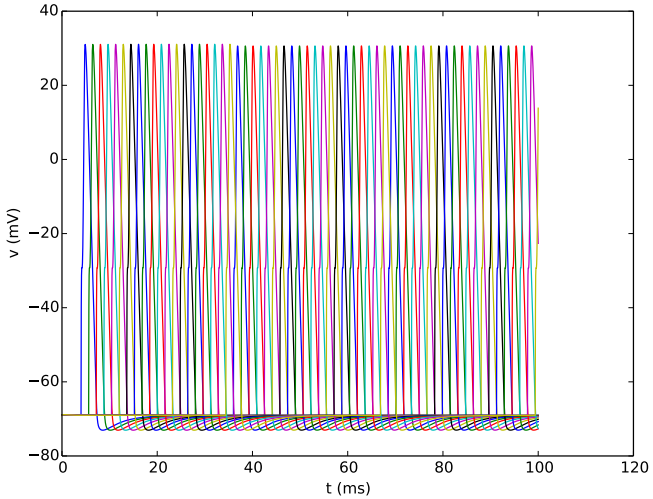
How can we generalize to a ring network with n cells?



Hint: As i increases, i % n counts: 0, 1, 2, ..., $n - 1$, 0, 1, ...

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
        # 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
            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, int(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:

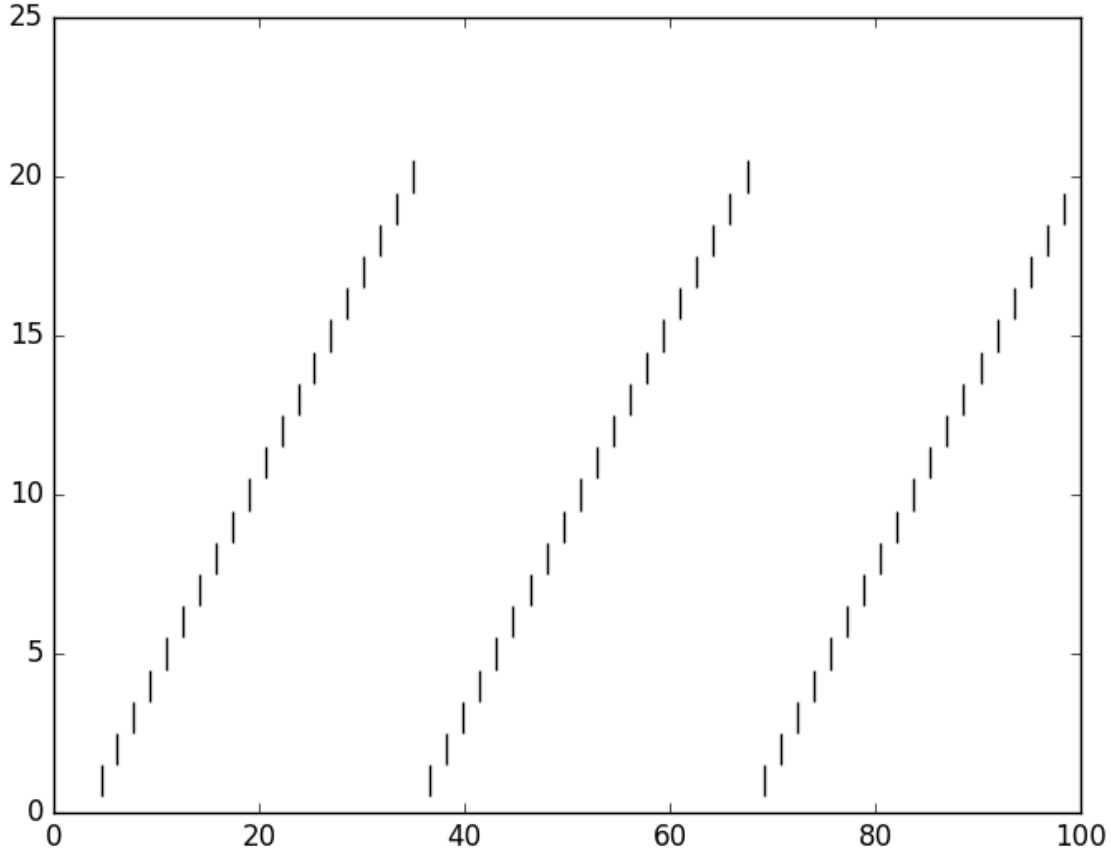
```
import json
with open('output.json', 'w') as f:
    f.write(json.dumps({
        i: list(cell.spike_times)
        for i, cell in enumerate(n.cells)},
        indent=4))
```

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
    ],
```

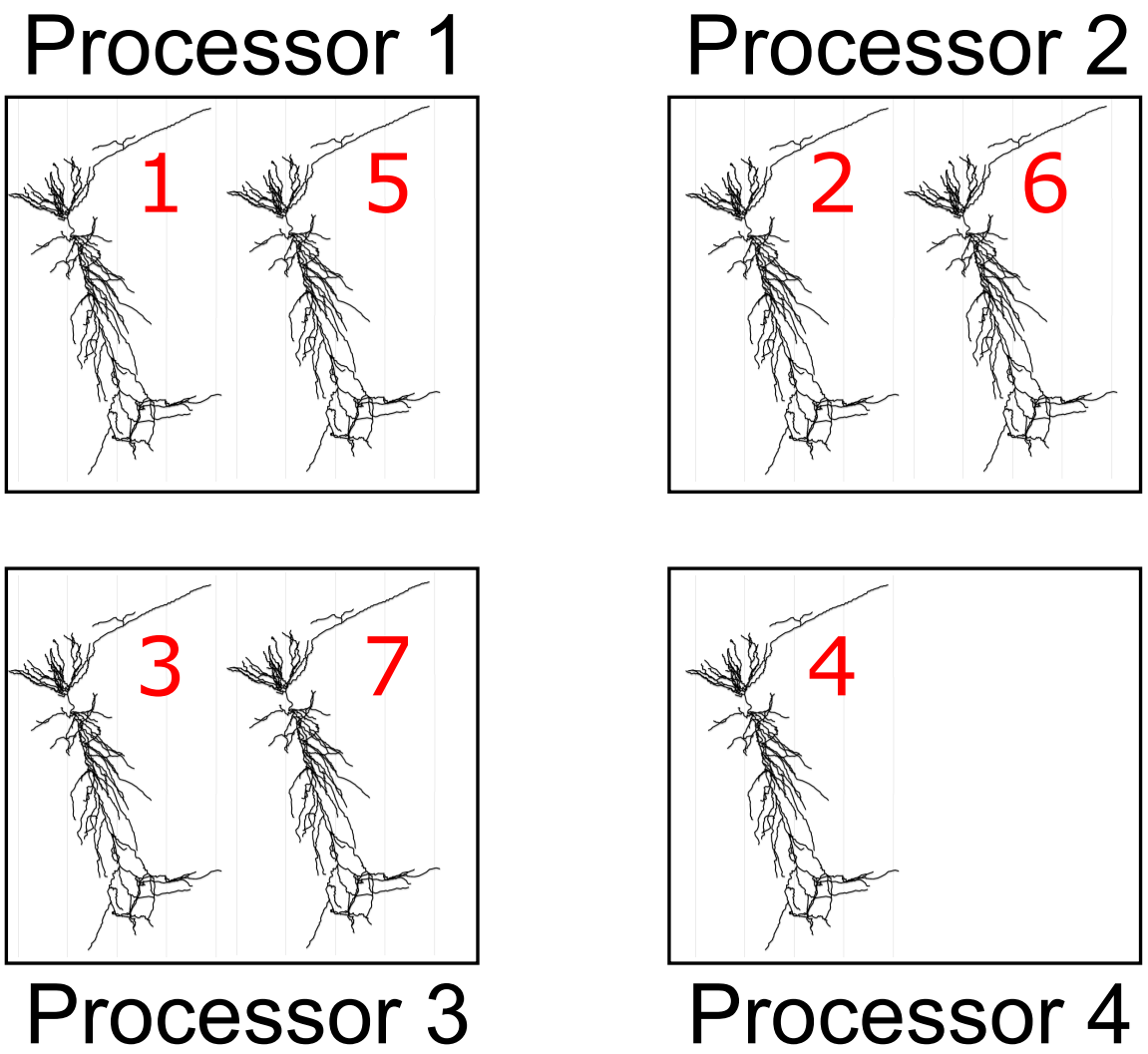
JSON is a standard format for data interchange. Libraries are available for most programming languages.

Raster plots




```
for i, cell in enumerate(n.cells):
    pyplot.vlines(cell.spike_times, i + 0.5, i + 1.5)
pyplot.show()
```

Simple parallelization strategy: round-robin.



Simple parallelization 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(int(pc.id()), ncell, int(pc.nhost())):
    pc.set_gid2node(gid, int(pc.id()))
    ...
```

(Note: the body is executed at most $\lceil$`ncell/nhost`$\rceil$ times, not `ncell`.)

Parallelizing our ring network

Very few changes are necessary.

An extra import at the very beginning:

```
from mpi4py import MPI
```

The `Network` class only instantiates gids on the current processor.

```
class Network:
    def __init__(self, num):
        pc = h.ParallelContext()
        mygids = list(range(int(pc.id()), num, int(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
        # 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
            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
```

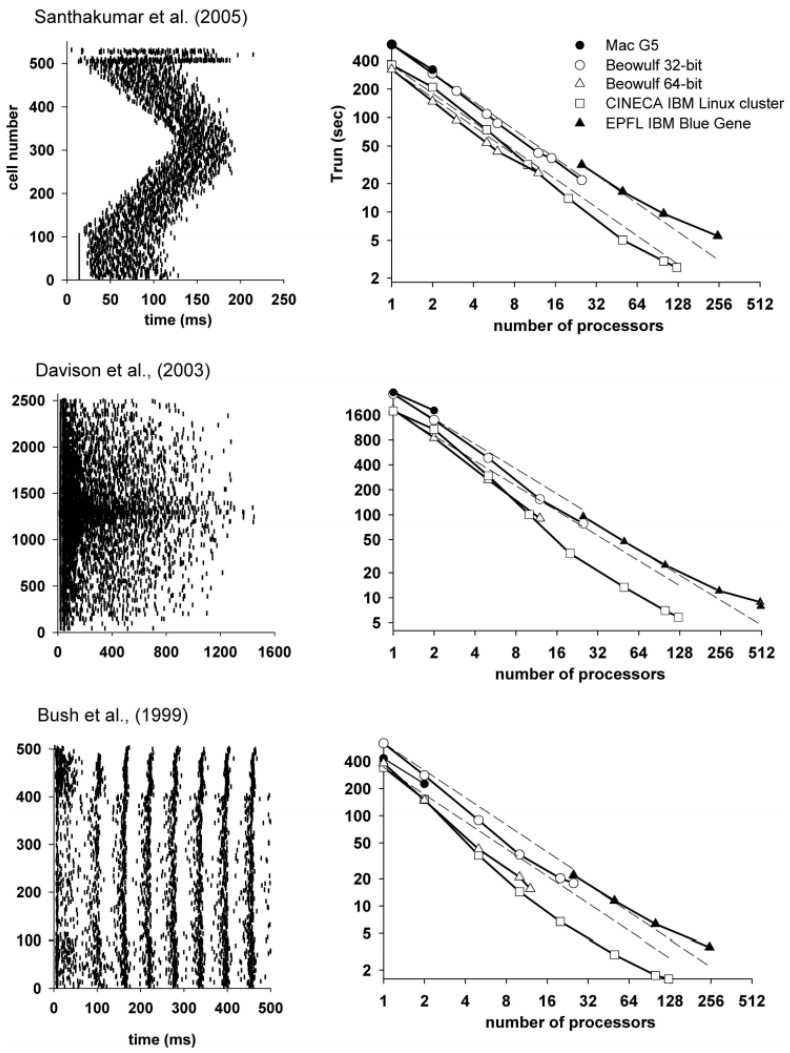
Finally, we modify the write to do it on a per-processor basis:

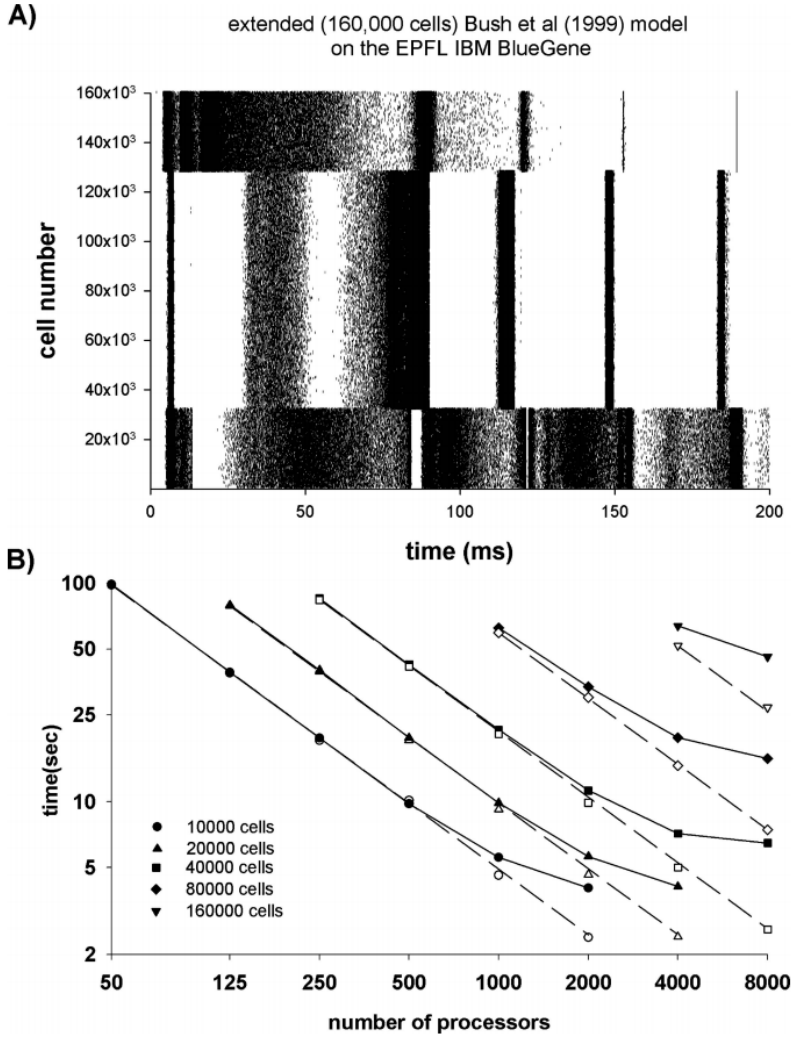
```
with open('output%d.json' % int(pc.id()), 'w') as f:
    f.write(json.dumps({cell._gid: list(cell.spike_times) for cell in n.cells},
        indent=4))
```

Optional: use pc.py_alltoall to send all spikes to node 0

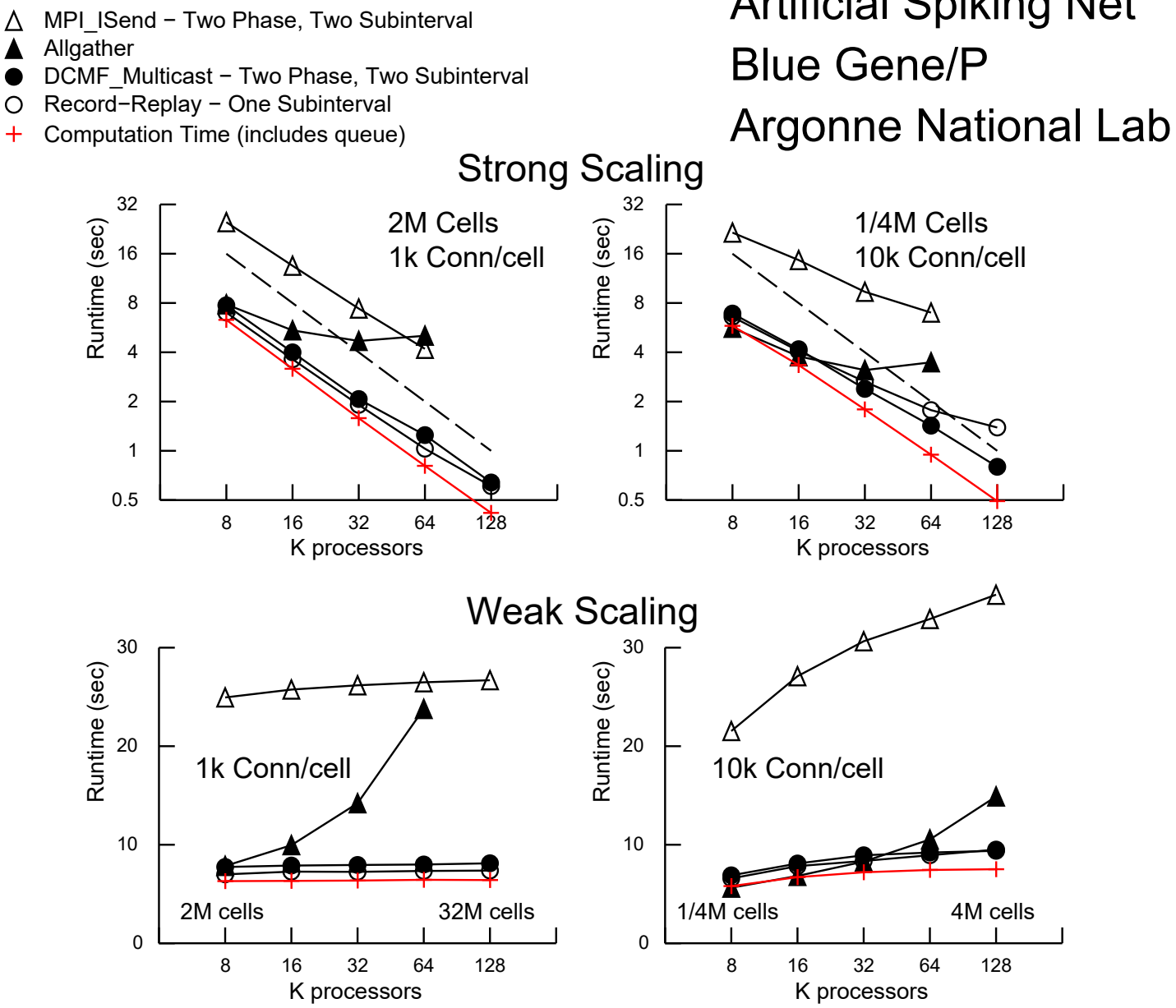
```
local_data = {cell._gid: list(cell.spike_times) for cell in n.cells}
all_data = pc.py_alltoall([local_data] + [None] * (int(pc.nhost()) - 1))

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

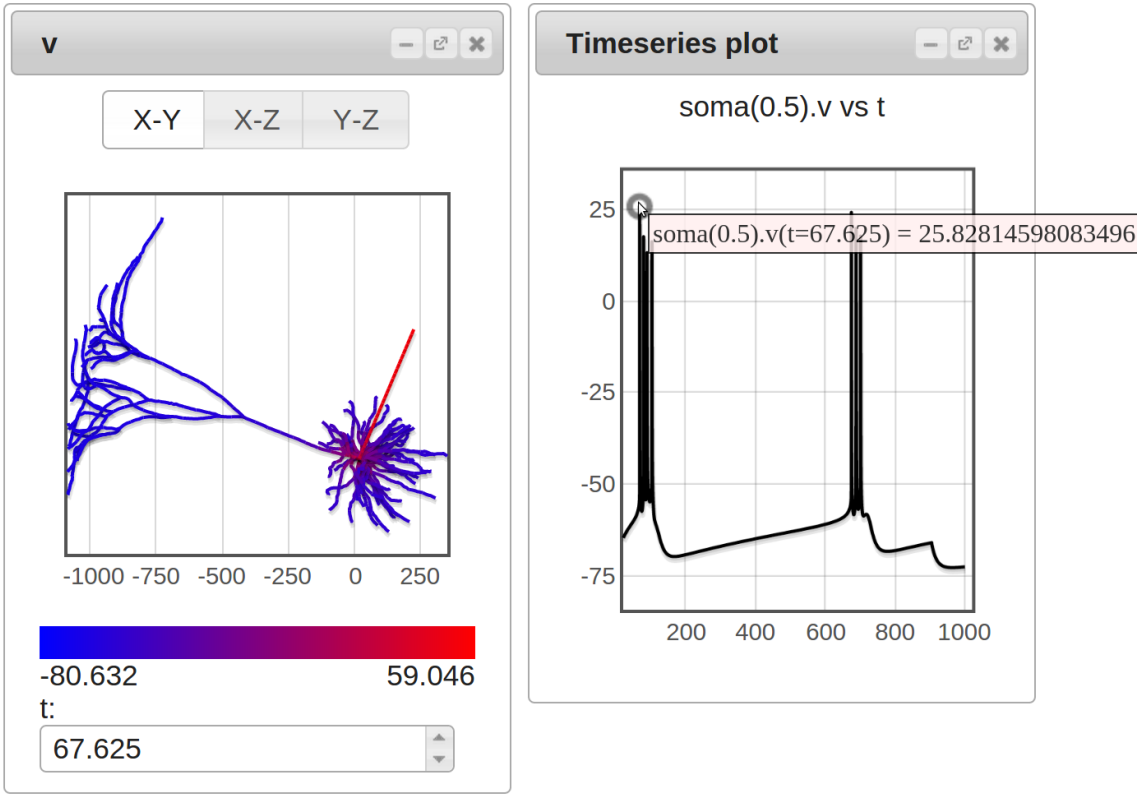
Tip: For network models, use a fixed step solver and not a variable step solver.

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:

http://www.neuron.yale.edu/hg/neuron/nrn/raw-file/tip/share/examples/nrniv/netcon/vecevent.mod

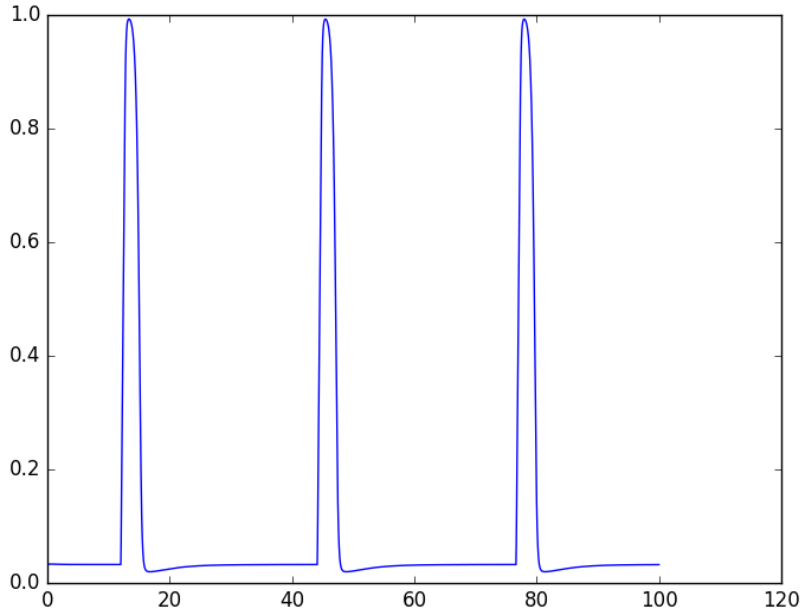
Using spike data to recreate a variable of interest

```
import json
from neuron import h
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
    nc.weight[0] = 1
    # setup recording
    t, m = h.Vector(), h.Vector()
    t.record(h._ref_t)
    m.record(p.soma[0](0.5)._ref_m_hh)
    # do run
    pc = h.ParallelContext()
    pc.set_maxstep(10)
    h.v_init = -69
    h.stdinit()
    pc.psolve(100)
    return t, m

t, m = get_m(5)
pyplot.plot(t, m)
pyplot.show()
```



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

We are in the process of translating the NEURON help documentation from HOC to Python. The partly translated documentation is available online at:

http://neurosimlab.org/ramcd/pyhelp/