Electrophysiology simulation with NEURON

Robert A. McDougal

Yale School of Medicine

13 July 2018

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
```

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

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

```
apical
```

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

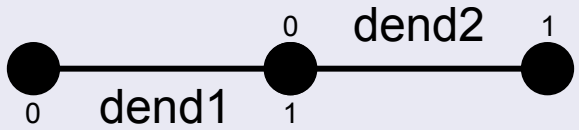
In recent versions of NEURON, named Sections will print with their name; e.g. it suffices to say `print (apical)`.

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

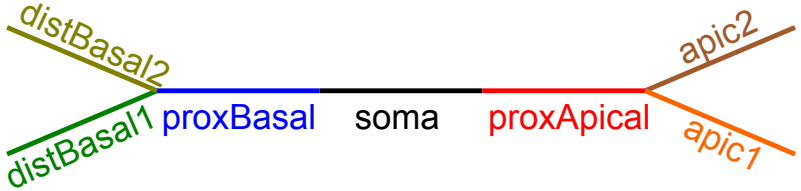
Python script:

```
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 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.

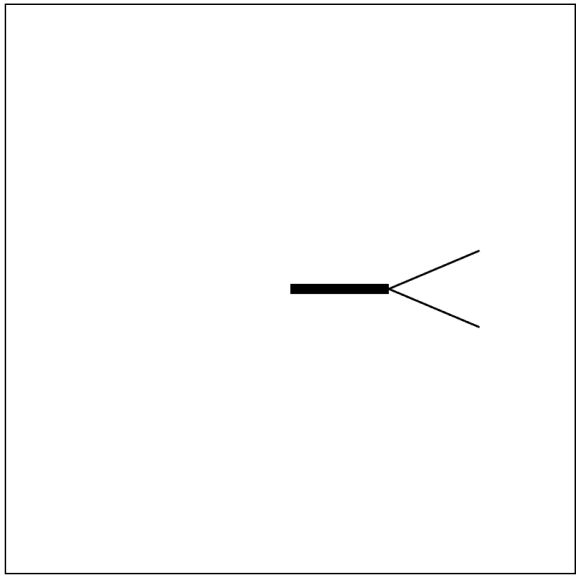
```
pyr2 = Pyramidal()
```

or even

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

Viewing the morphology with gui2.PlotShape

```
from neuron import h, gui2
gui2.set_backend('jupyter')

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 = gui2.PlotShape()
# use 1 instead of 0 to hide diams
ps.show(0)
```



Note: `PlotShape` can also be used to see the distribution of a parameter or variable.

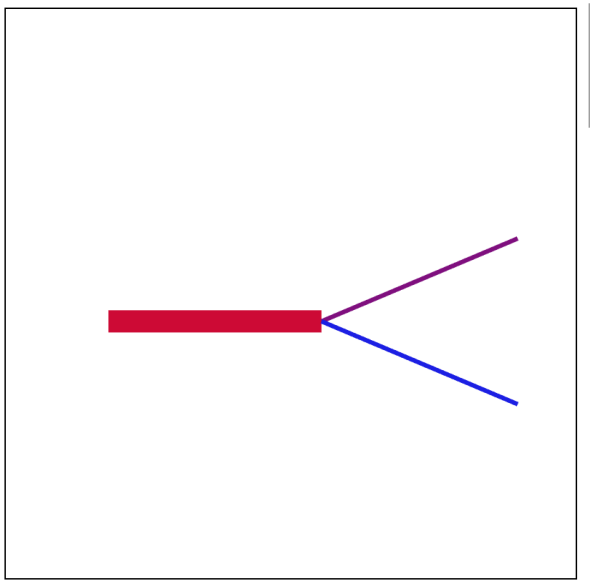
Viewing voltage, sodium, etc

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 modify our PlotShape to color-code the sections by voltage:

```
ps.scale(-80, 80)
ps.variable('v')
```

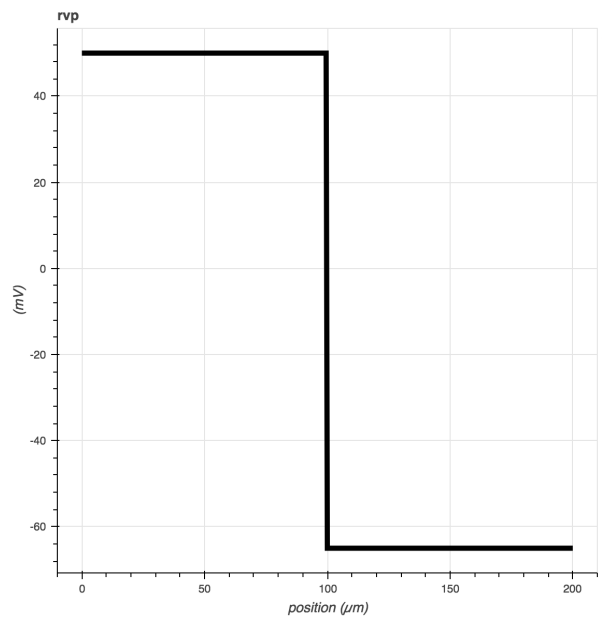


After increasing the spatial resolution:

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

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

```
# bokeh already loaded, output_notebook()
rvp = h.RangeVarPlot('v')
rvp.begin(0, sec=my_cell.main)
rvp.end(1, sec=my_cell.dend2)
x, y = h.Vector(), h.Vector()
rvp.to_vector(y, x)
p = figure(title='rvp')
p.line(x, y, line_width=5, line_color='black')
show(p)
```

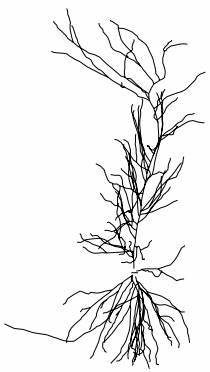


Sodium concentration could be plotted with ’nai’ instead of ’v’, etc.

Loading morphology from an swc file

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

```
from neuron import h, gui2
gui2.set_backend('jupyter')
h.load_file('stdgui.hoc')
h.load_file('import3d.hoc')

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

pyr = Pyramidal()

ps = gui2.PlotShape()
```



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

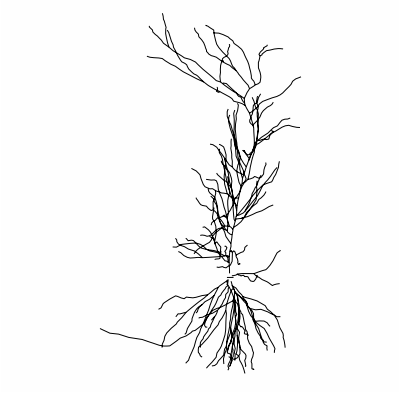
Only do this in code after you've already examined the cell with the Import3D GUI tool and fixed any issues in the SWC file.

Working with multiple cells

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

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

We then view these in a PlotShape:



Where are the other 9 cells?

Working with multiple cells

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

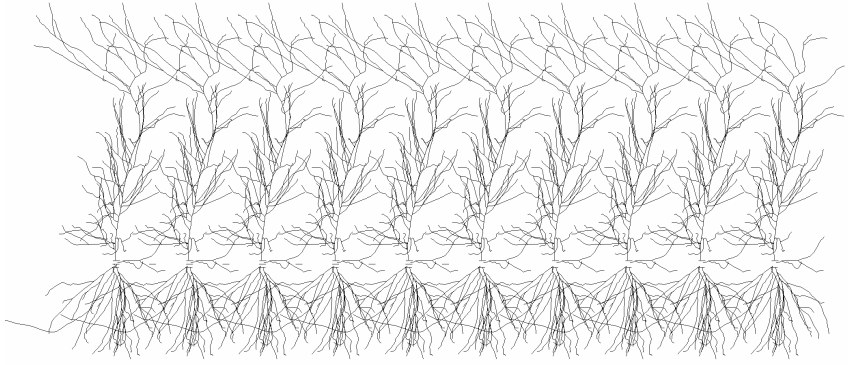
```
class Pyramidal:
  def _shift(self, x, y, z):
    soma = self.soma[0]
    n = soma.n3d()
    xs = [soma.x3d(i) for i in range(n)]
    ys = [soma.y3d(i) for i in range(n)]
    zs = [soma.z3d(i) for i in range(n)]
    ds = [soma.diam3d(i) for i in range(n)]
    for i, (a, b, c, d) in enumerate(zip(xs,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

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

Running simulations

Basics

To initialize a simulation to -65 mV:

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

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

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

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

Ways to improve accuracy

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

To increase nseg for all sections:

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

Recording data

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

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

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

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

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

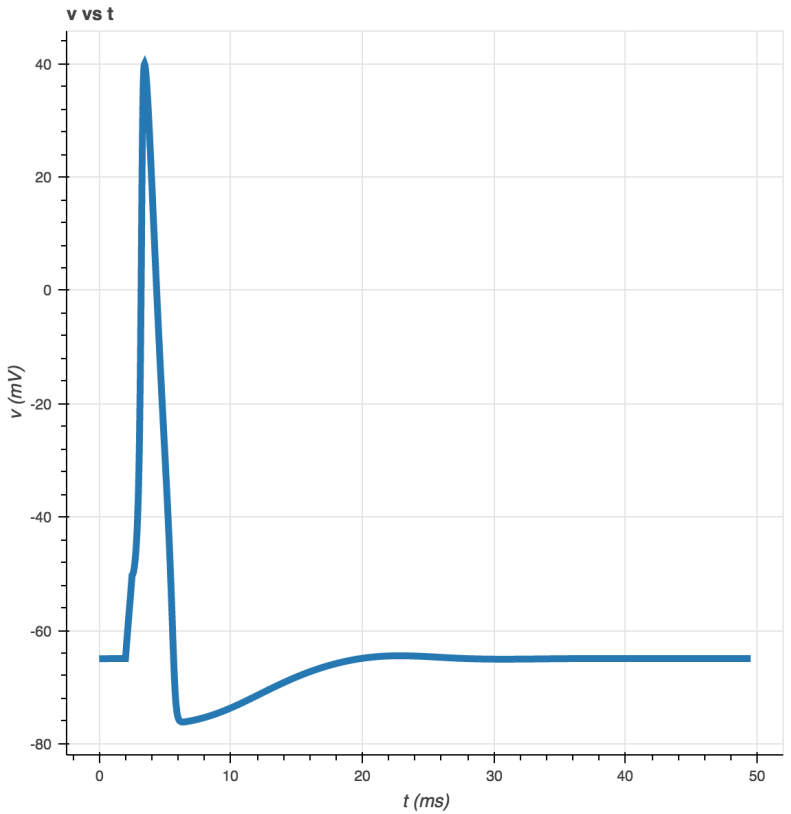
Tips

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

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 bokeh.io import output_notebook
from bokeh.plotting import figure, show
from neuron import h
output_notebook()
h.load_file('stdrun.hoc')

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

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

p = figure(title='v vs t', x_axis_label='t (ms)', y_axis_label='v (mV)')
p.line(t, v, line_width=5)
show(p)
```



Storing data to CSV to share with other tools

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

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

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

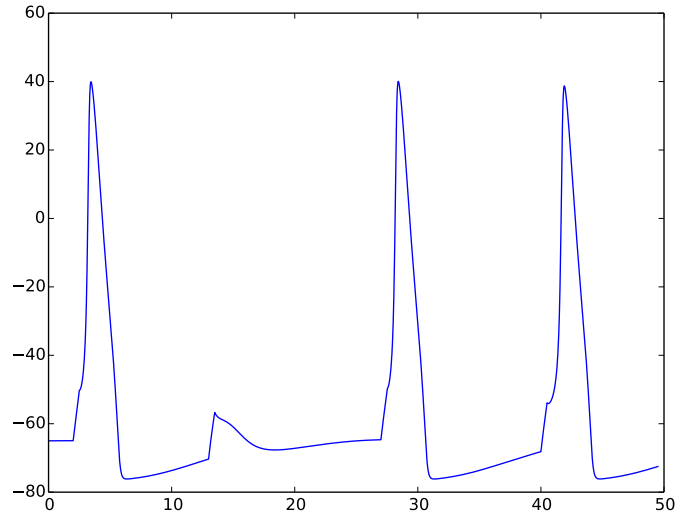
```
import csv
with open('data.csv', 'wb') as f:
    csv.writer(f).writerows(zip(t, v))
```

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

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

A spike occurs whenever V_m crosses some threshold (e.g. 0 mV).
Python can easily find all spike times.

```
from neuron import h, gui
from matplotlib import pyplot
soma = h.Section(name='soma')
soma.insert('hh')
# current clamps
iclamps = []
for t in [2, 13, 27, 40]:
    i = h.IClamp(soma(0.5))
    i.delay = t # ms
    i.dur = 0.5 # ms
    i.amp = 50
    iclamps.append(i)
# recording
t = h.Vector()
v = h.Vector()
t.record(h._ref_t)
v.record(soma(0.5)._ref_v)
# simulation
h.finitialize(-65)
h.continuerun(49.5)
# compute spike times
st = [t[j] for j in range(len(v) - 1)
      if v[j] <= 0 and v[j + 1] > 0]
print ('spike times:')
print (st)
# plotting
pyplot.plot(t, v)
pyplot.show()
```



The console displays:

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

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

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

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

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

The result:

```
[24.974999999998925, 13.475000000001966]
```

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

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:

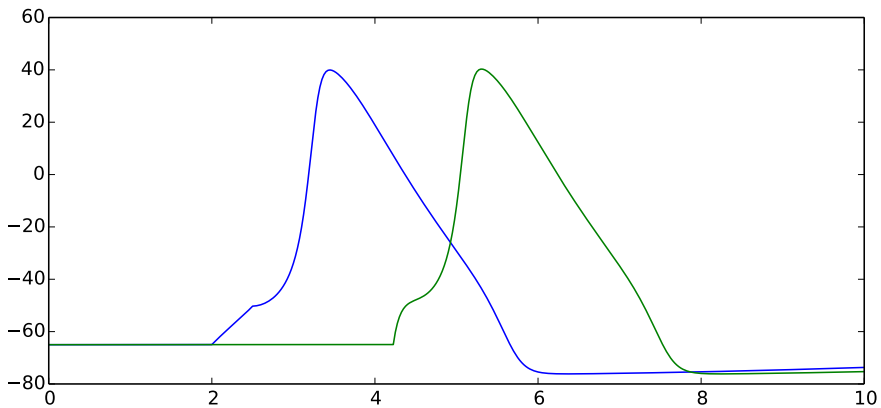
```
postsyn = h.ExpSyn(neuron2.soma(0.5))
postsyn.e = 0  # reversal potential
```

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

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