Scripting NEURON

Robert A. McDougal

Yale School of Medicine

18 October 2019

What is a script?

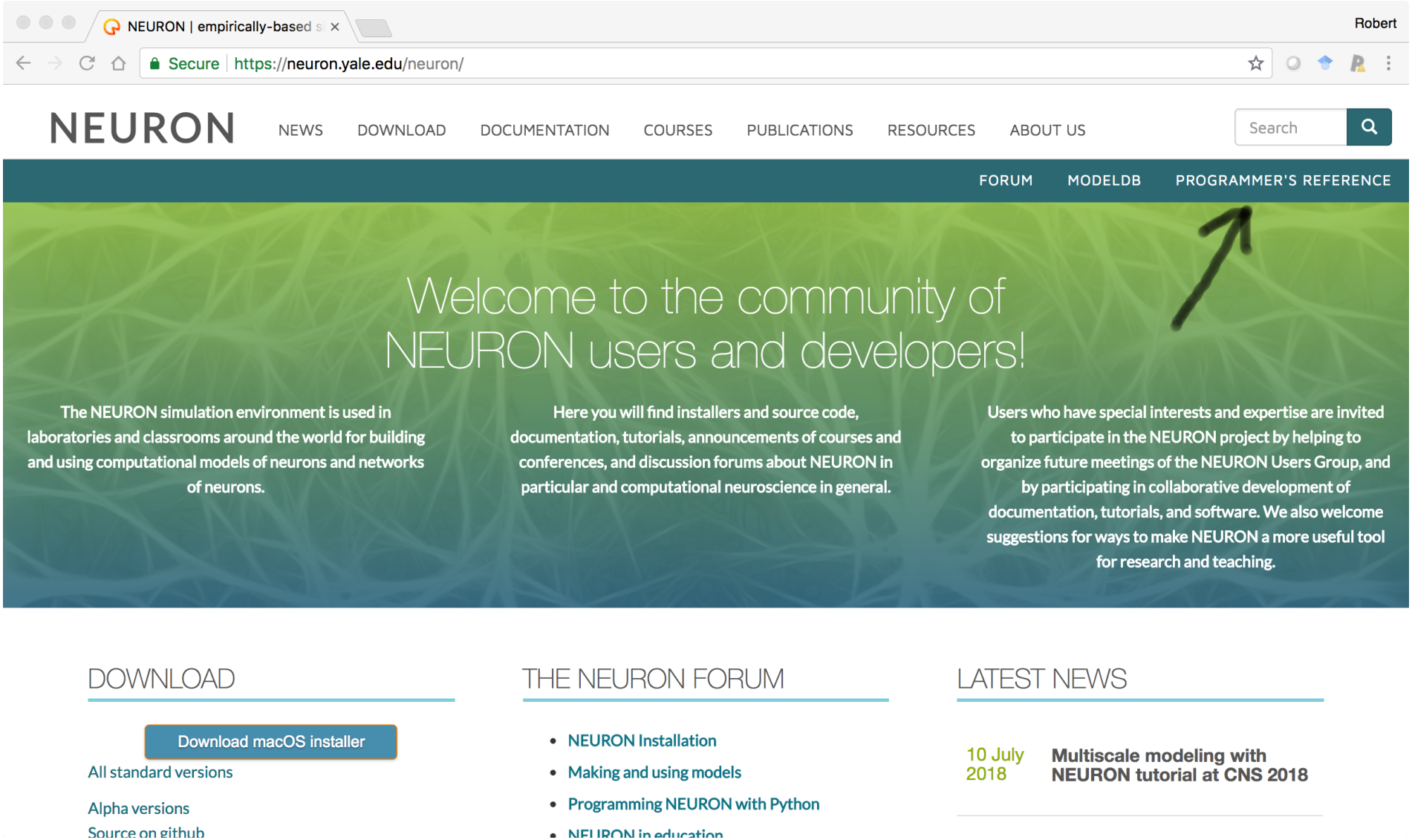
A **script** is a file with computer-readable instructions for performing a task.

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

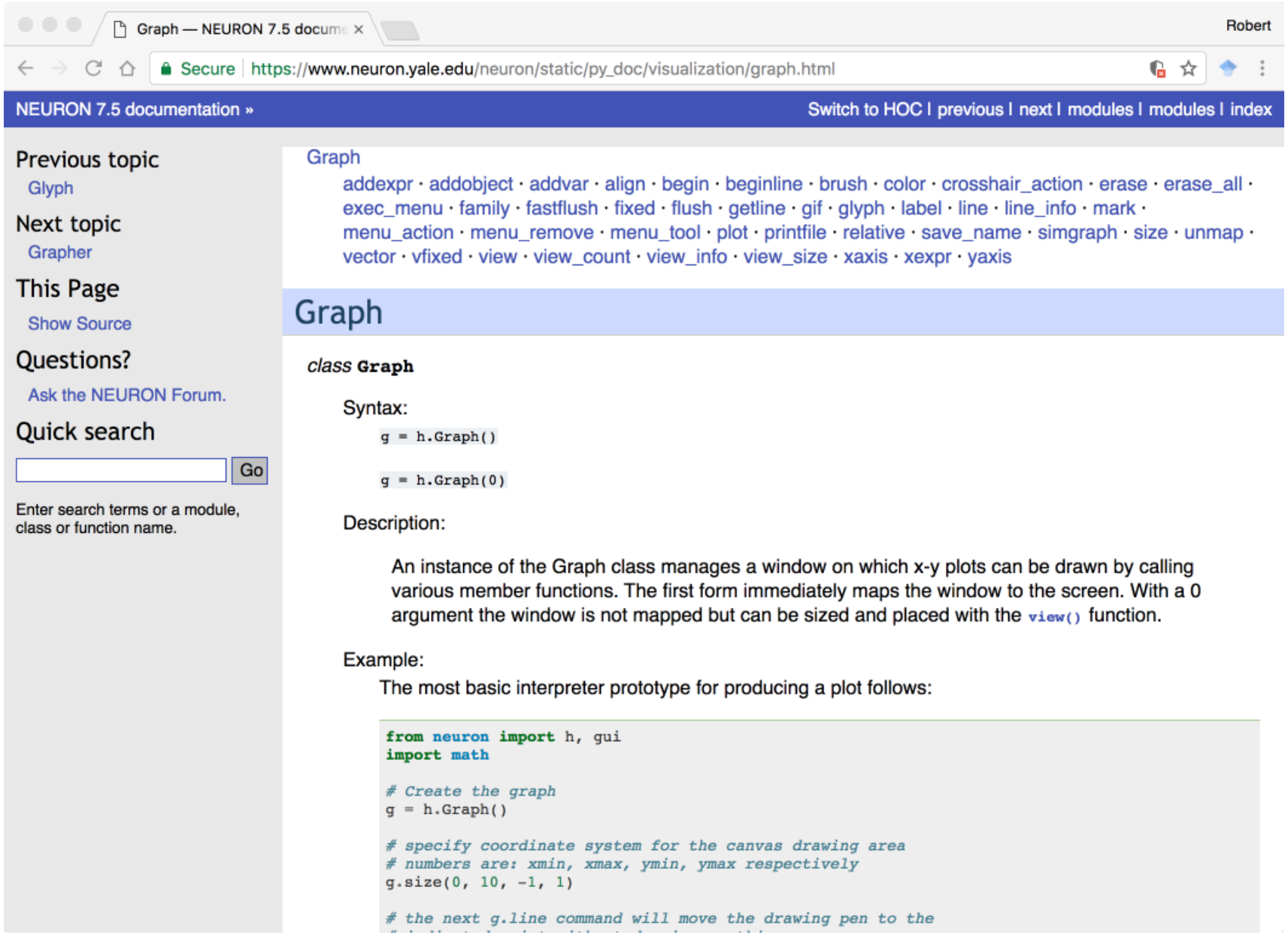
Why write scripts for NEURON?

- 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.

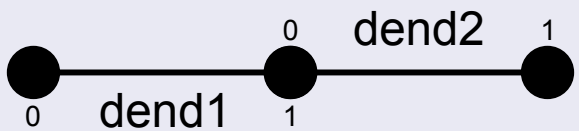
The last print is equivalent to `print (apical)` but `str` was shown to illustrate how to get a string representation.

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

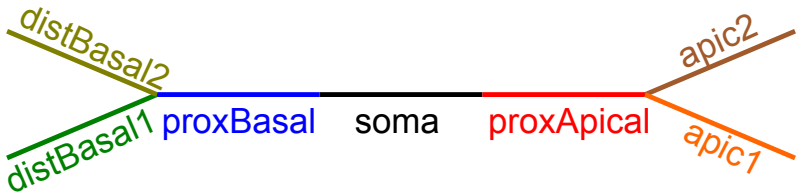
Python script:

```
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()
```

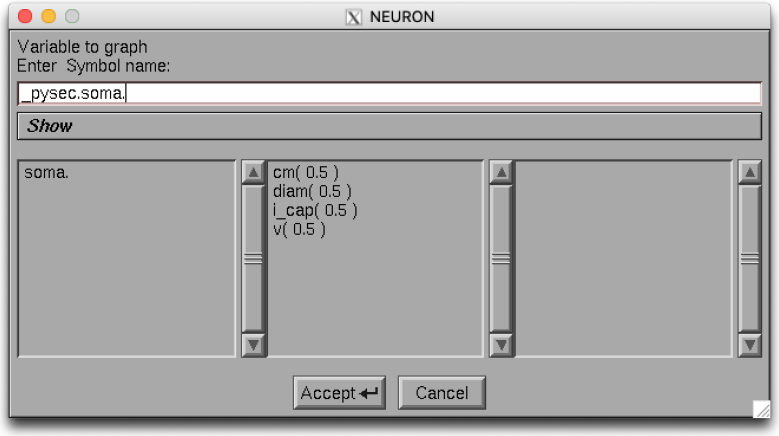
or even

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

Tip: Sections that work well with GUI tools

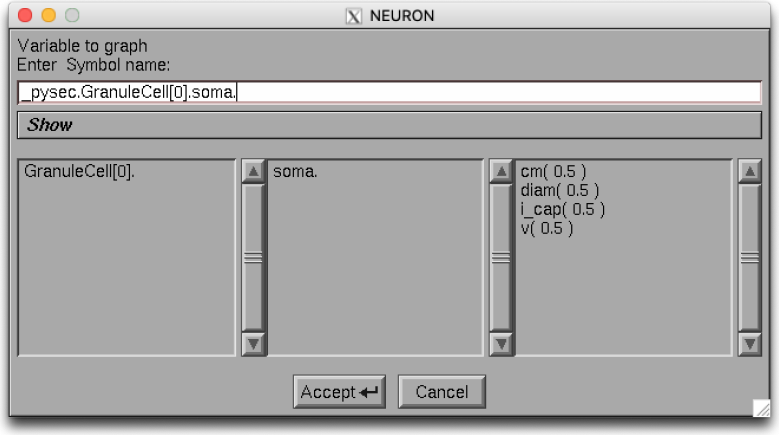
For meaningful Section names to appear in the GUI tools, the `name` attribute must be specified for top-level Sections:

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



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

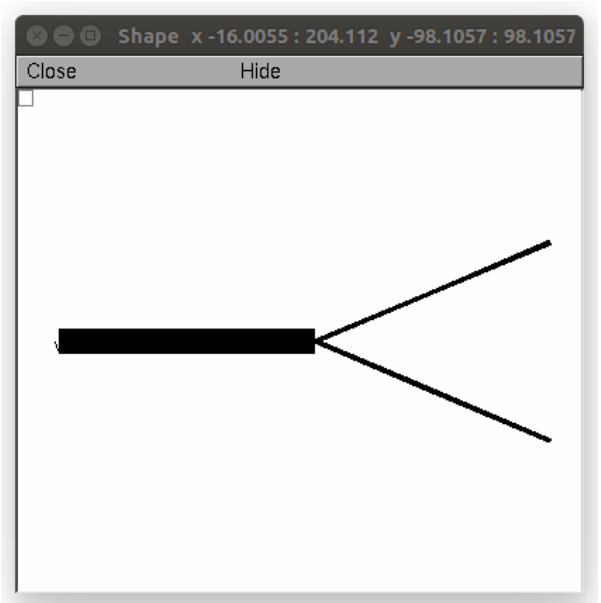
```
class GranuleCell:
    def __init__(self, gid):
        self._gid = gid
        self.soma = h.Section(name='soma', cell=self)
    def __str__(self):
        return 'GranuleCell[{}]'.format(self._gid)

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

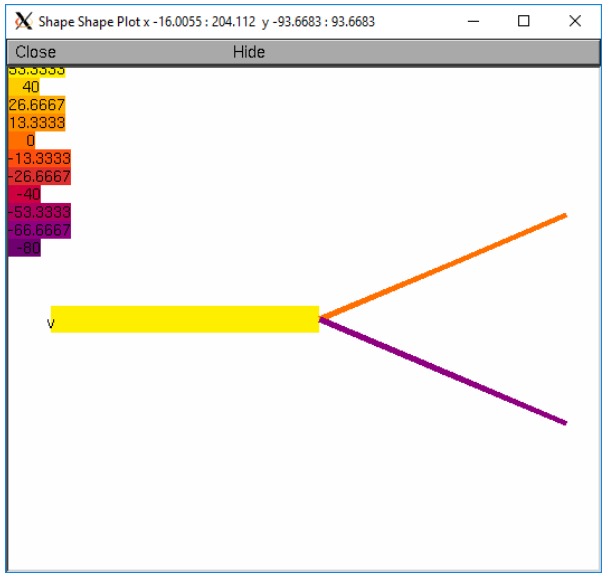
Viewing voltage, sodium, etc

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)
```

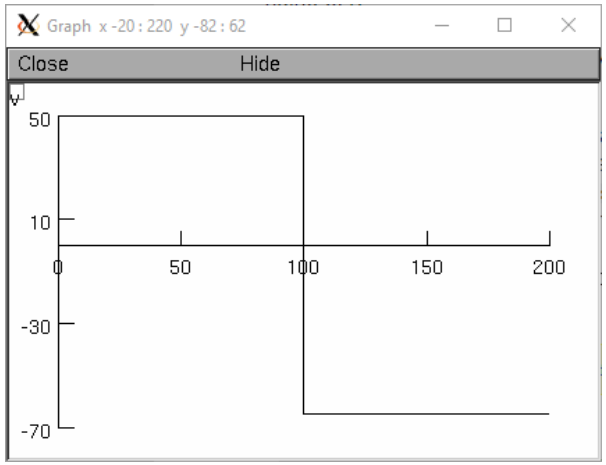


After increasing the spatial resolution:

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

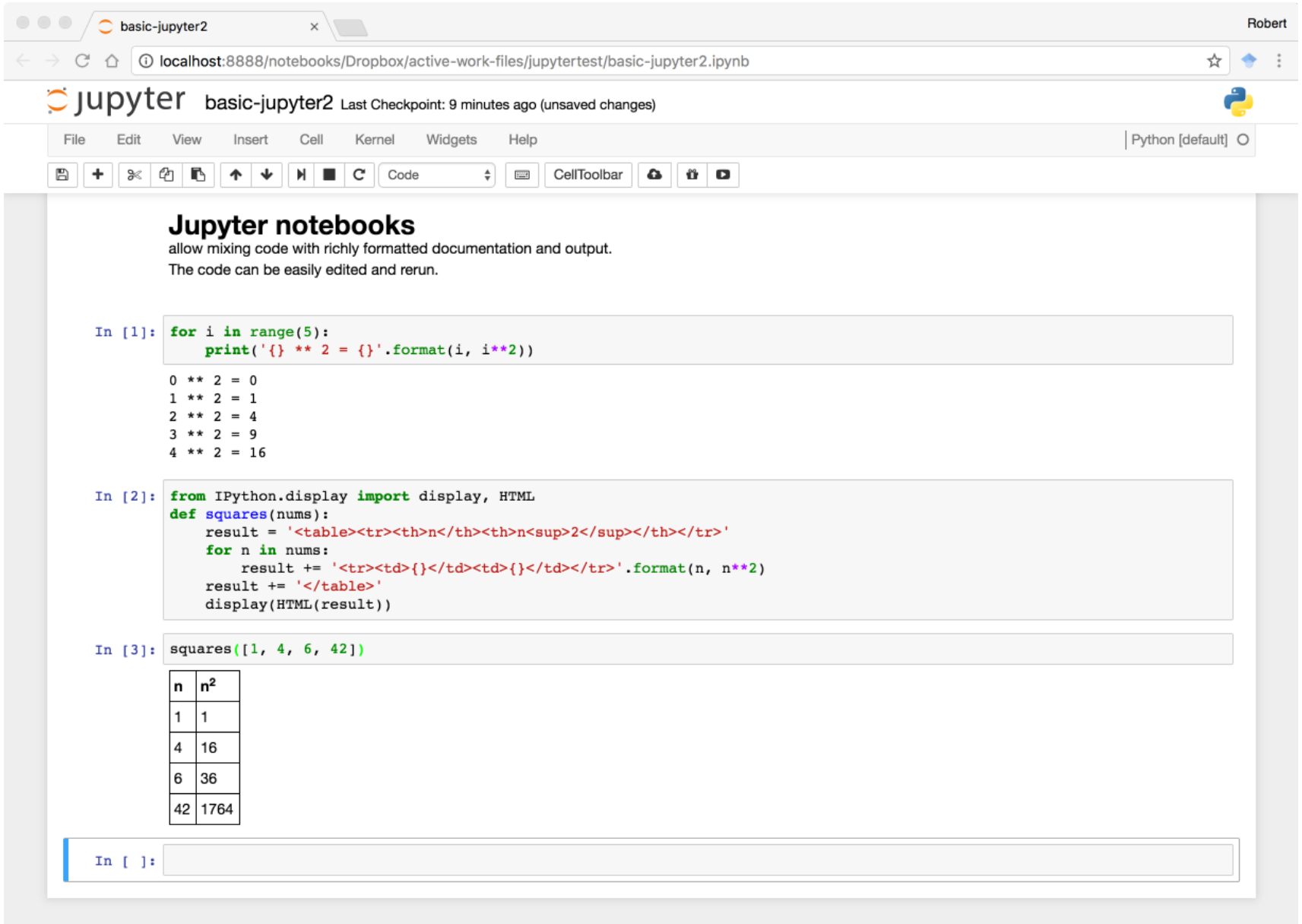
We can plot the voltage as a function of distance from `main(0)` to `dend2(1)`:

```
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.

Aside: Jupyter



Aside: Jupyter

In [1]:
```
%matplotlib notebook
```

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

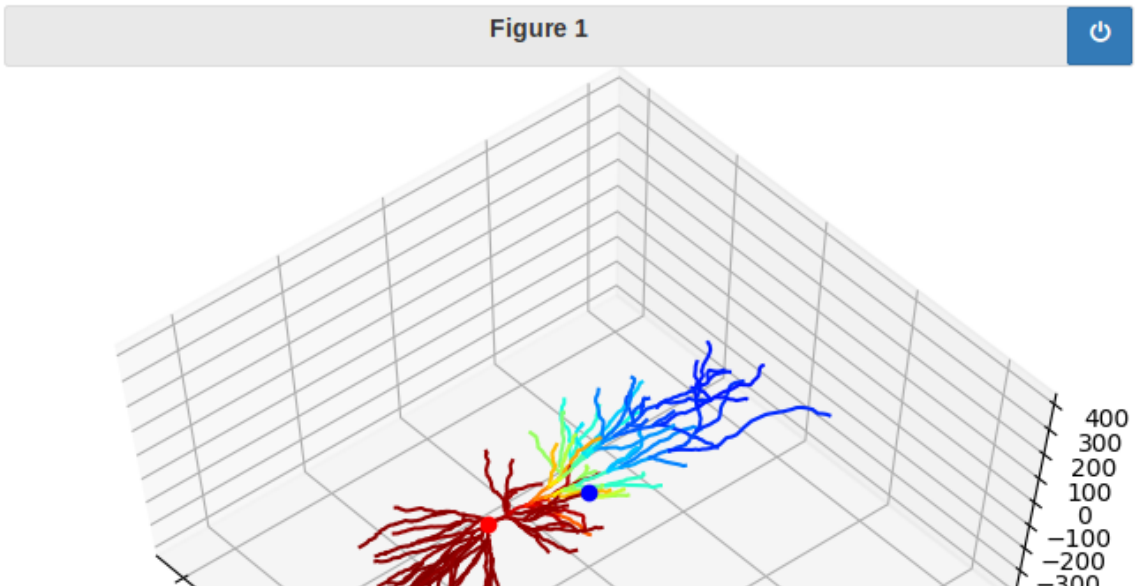
Out[2]: 1.0

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

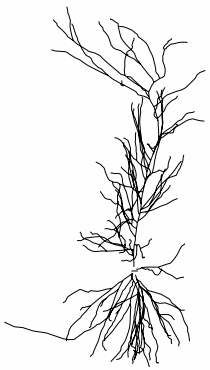
In [5]:
```
ps = h.PlotShape(False)
ps.plot(pyplot, cmap=cm.jet).mark(h.soma[0](0.5)).mark(h.apical_dendrite[68](1), marker='ob')
```



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()`.

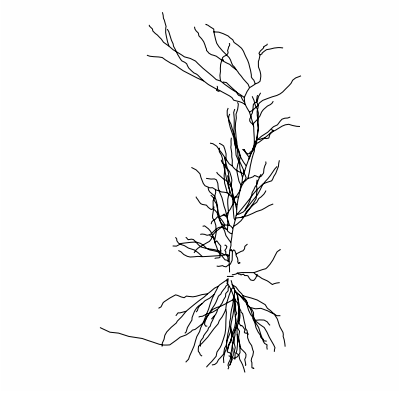
Only do this in code after you've already examined the cell with the Import3D GUI tool and fixed any issues in the SWC file.

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__`:

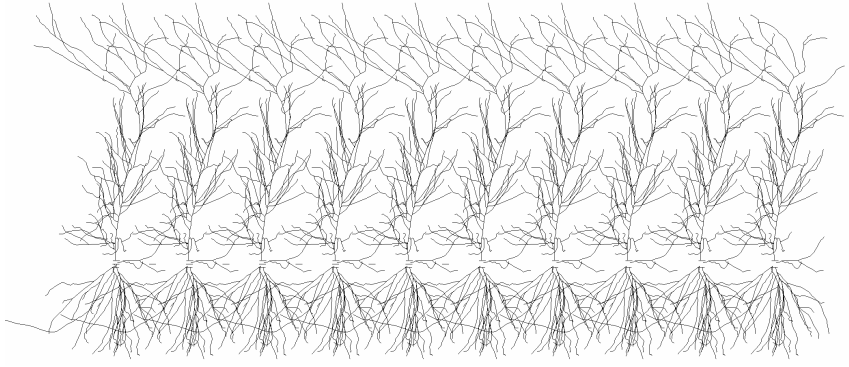
```
class Pyramidal:
  def _shift(self, x, y, z):
    soma = self.soma[0]
    n = soma.n3d()
    xs = [soma.x3d(i) for i in range(n)]
    ys = [soma.y3d(i) for i in range(n)]
    zs = [soma.z3d(i) for i in range(n)]
    ds = [soma.diam3d(i) for i in range(n)]
    for i, (a, b, c, d) in enumerate(zip(xs,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

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)
```

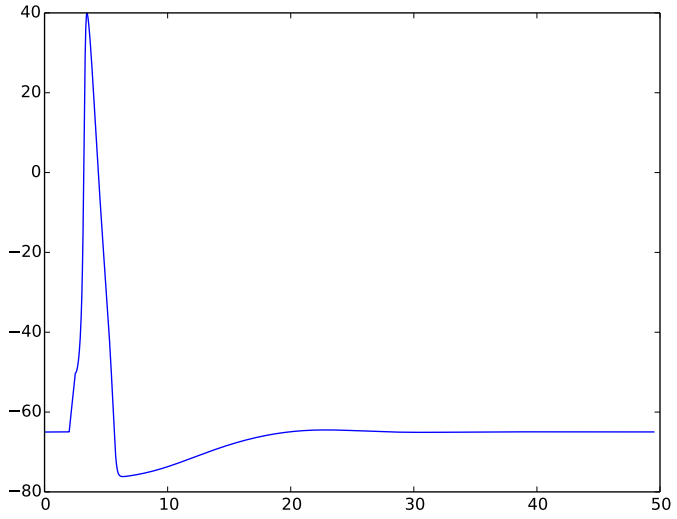
Tips

- 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.

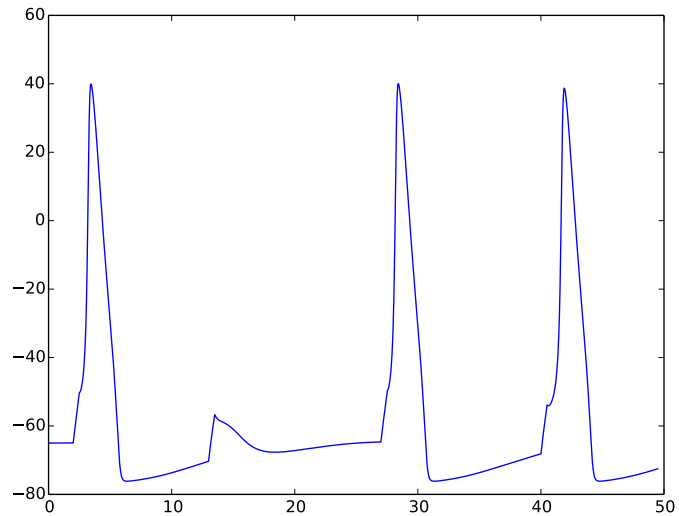
Example: Hodgkin-Huxley

```
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 (e.g. 0 mV) in a positive-going direction. NEURON's `NetCon` objects can detect this directly, but Python can easily find all spike times from a voltage time series. Only 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
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().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:

```
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
from neuron.units import ms


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:

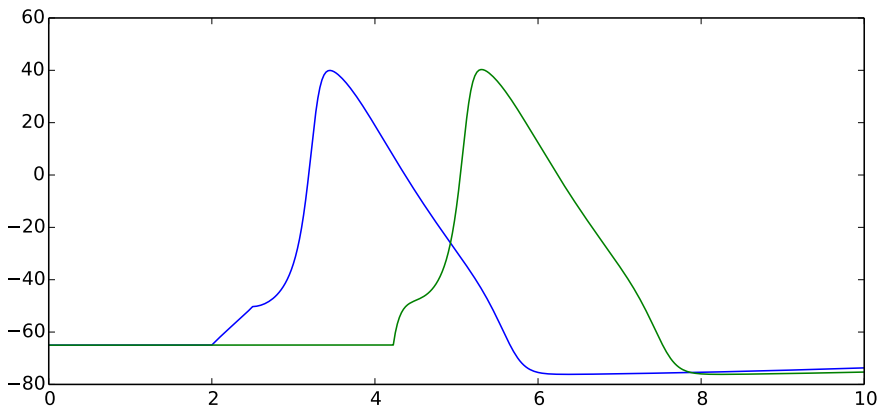
```
syn = h.NetCon(neuron1.soma(0.5)._ref_v, postsyn, sec=neuron1.soma)
syn.delay = 1
syn.weight[0] = 5
```

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