Modeling in Python-NEURON

Michael Hines

Yale School of Medicine

School of Brain Cells and Circuits; Erice 2015

Special thanks to Robert McDougal
Program development supported by NINDS
These slides available at:
`http://neuron.yale.edu/ftp/neuron/neuron-python-erice2015.pdf`

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.

Why Python?

Why Python?...

Very large community.

- Huge number of modules useful in CNS.
- Graphics.
- Vector, Matrix computation.
- Databases.
- Connections to the rest of the world.

...Why Python?

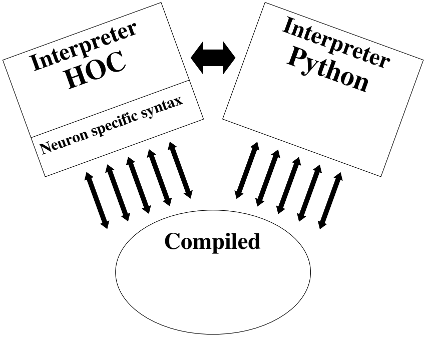
Far greater elegance and expressiveness than Hoc.

- Complete modern object oriented language.
- Conceptual control even when codes get large.
- Debugging
- list, dict, callable, iteration, modules

Everything in NEURON still works

and is visible in all details from Python.

- Including ALL legacy models and graphics.
- Python can execute any HOC statement.
- HOC can execute any Python statement.



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.

Ordered list of objects

Objects can be appended to a list.

```
    a = []
    for i in range(100):
      a.append(2*i)
```

The length of the list is

```
    print (len(a))                                              100
```

If you know the index, you can find the object very quickly.

```
    print (a[40])                                                80
```

List comprehensions simplify the building of lists.

```
    a = [2*i for i in range(100)]
```

You can iterate over a list by index, object, or both

```
    for i in range(len(a)):
      if a[i] == 2*80:  print (i)                                80
    for x in a:
      if x == 2*80:  print (x)                                  160
    for i,x in enumerate(a):
      if x == 2*80:  print (i)                                   80
```

Key,Value store

We can invert the association between index and value using a dict.

```
    d = {}
    for i, x in enumerate(a):
      d[x] = i
```

or, more elegantly,

```
    d = dict((x, i) for i, x in enumerate(a))
```

If you know the key, you can find the value very quickly.

```
    print (d[160])                                        80
```

You can iterate over keys, values, or both.

```
    for k in d:
      if d[k] == 80:  print (k)                          160
    for v in d.values():
      if v == 80:  print (a[v])                          160
    for k, v in d.items():
      if d[k] == 80:  print ((k, v))               (160, 80)
```

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), . . .

Finding help within Python

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

```
import numpy
print (dir(numpy))
```

Displays:

```
['__doc__', '__name__', '__package__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', ...
```

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

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

Online resources

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

Importing NEURON

```
from neuron import h
```

Makes everything in NEURON available to the Python interpreter.

```
print h.hname()                    TopLevelHocInterpreter

help(h)
```

```
NEURON Python Online Help System
================================

neuron.h
========

neuron.h is the top-level HocObject, allowing
    interaction between python and Hoc.
...
```

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:

```
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
```

To name a section, declare a `name` attribute:

```
dend2 = h.Section(name='apical')
```

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

```
print (dend2.name())                                      apical
```

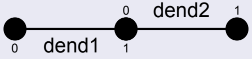
Also available: a `cell` attribute for grouping sections by cell.

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

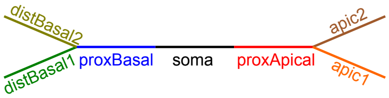
Python script:

```
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:



Reduce work by writing functions

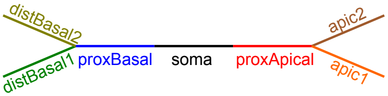
Python script:

```
from neuron import h

# helper functions
def sections(*names):
    secs = [h.Section(name=n)
            for n in names]
    return tuple(secs)

def connect(connections):
    for parent in connections:
        for child in connections[parent]:
            child.connect(parent)

# define, connect, print
soma, papic, apic1, apic2, pb, db1, db2 = \
    sections('soma', 'proxApical', 'apic1',
             'apic2', 'proxBasal',
             'distBasal1', 'distBasal2')

connect({soma: [papic],
         papic: [apic1, apic2],
         pb: [db1, db2]})
pb.connect(soma(0))

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 and diameter

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

```
    sec.L = 20

    sec.diam = 2
```

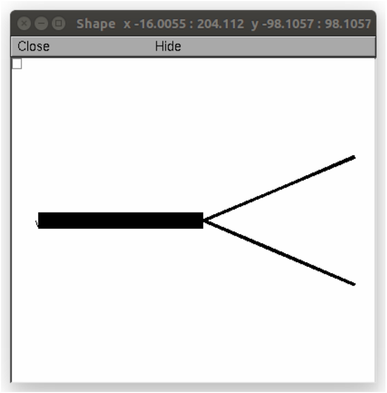
Note: Diameter need not be constant; it can be set per segment.

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 modelling mammalian cells.

Viewing the morphology with h.PlotShape

```
from neuron import h, gui

main = h.Section()
dend1 = h.Section()
dend2 = h.Section()

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

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

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 calculated variable. To save the image in plot shape ps use `ps.printfile('filename.eps')`

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
```

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())
```

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(1)`
Increase the discretization resolution: `sec.nseg = 11`

To increase nseg for all sections:

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

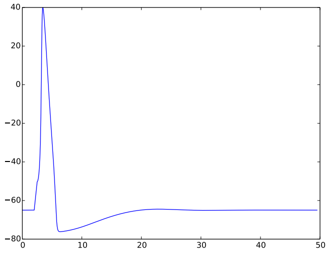
Tips

- 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 neuron import h, gui
from matplotlib import pyplot

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

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

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

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

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



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.

HOC Vector and 1-D numpy array operate well together

```
import numpy
```

Creating an array of 1 million elements takes

```
na = numpy.arange(0., 10., .00001)        0.00663 seconds
```

Copying that numpy array to a new HOC Vector takes

```
hv = h.Vector(na)                         0.00499 seconds
```

... much faster than

```
hv = h.Vector()
for x in na:
  hv.append(x)                               1.95 seconds
```

Copying the other way is also fast.

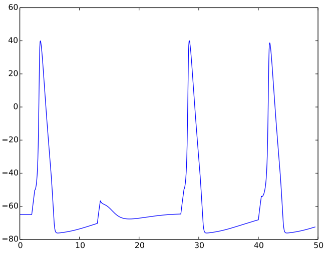
```
na = numpy.array(hv)                       0.0025 seconds
```

Sharing the data is fastest of all.

```
na = hv.as_numpy()                       0.000107 seconds
na[45] = 3.14
print (na[45] == hv.x[45])                           True
```

A spike occurs whenever V_m crosses some threshold (e.g. 0 mV). Python can easily find all spike times. Only changes from the previous example are highlighted.

```
from neuron import h, gui
from matplotlib import pyplot
soma = h.Section()
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()
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.

HOC was NEURON's original programming language. There are many valuable HOC functions in ModelDB and elsewhere. Python scripts can easily use these functions via a two step process:

Load the HOC library, here `libraryname.hoc`:

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

Invoke the HOC function, here `test` by proceeding its name with an `h.` and passing the appropriate arguments:

```
h.test(13, 172.2)
```

SES files created by saving the session are written in HOC and may be loaded the same as with any other HOC file.

Example

HOC code: `myneuron.hoc`

```
// define a cell
create soma, apic, basal
soma {
    connect apic(0), 1
    connect basal(0), 0
    L = 20
    diam = 20
}
```

Python script:

```
from neuron import h
h.load_file('myneuron.hoc')
h.topology()
```

Running the Python script shows:

```
|-|       soma(0-1)
  `|        apic(0-1)
 `|       basal(0-1)
```

Section-dependent functions

Some NEURON functions depend on the section; specify that with a `sec=` argument.

Example: calculating path distance

For example, `h.distance` is used to calculate the path distance in μm between two points along the neuron. To set a reference point at the center (0.5) of the soma, use:

```
h.distance(0, 0.5, sec=soma)
```

The distance from the reference point to the 1 end of apic is

```
h.distance(1, sec=apic)
```

Advanced topics

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.check_output([
        'git', 'log', '-1', '--pretty=format:%h']),
        suffix)
```

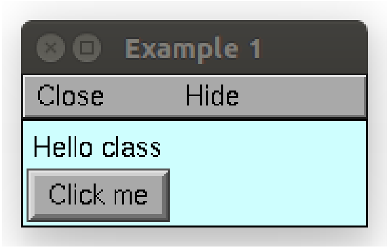
Then, for example, save matplotlib graphics with:

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

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`.

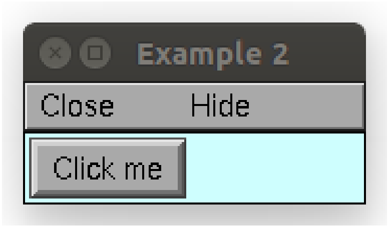
```
from neuron import h, gui

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!
```

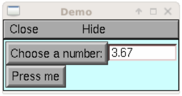
Pressing the button twice:

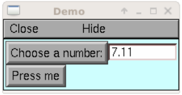
```
hello!
hello!
```

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()
```





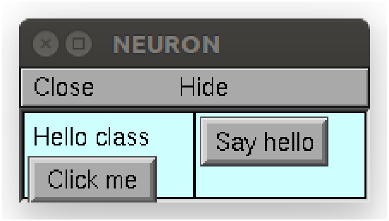
Clicking “Press me” on the left window and then on the right window displays:

```
You chose:
3.67
You chose:
7.11
```

Layout: HBox and VBox

Combine windows horizontally with HBox and vertically with VBox.

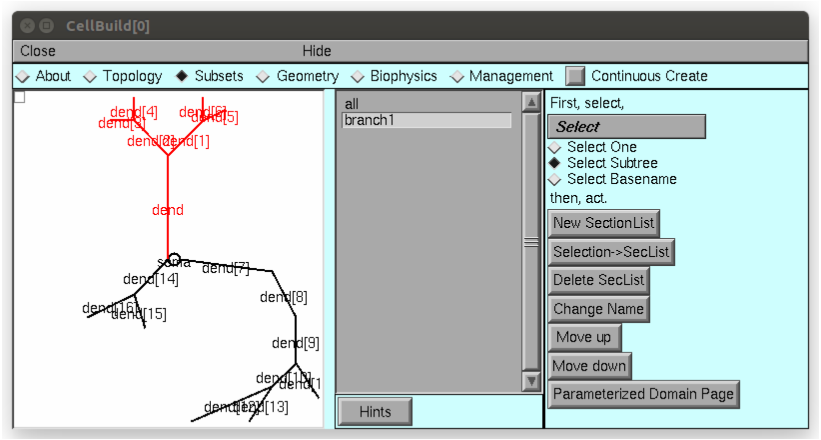
```
from neuron import h, gui
hbox = h.HBox()
hbox.intercept(1)
h.xpanel('Example 1')
h.xlabel('Hello class')
h.xbutton('Click me')
h.xpanel()
h.xpanel('Example 3')
h.xbutton('Say hello')
h.xpanel()
h.xpanel()
hbox.intercept(0)
hbox.map()
```



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:



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