HOC

for reading knowledge

Robert A. McDougal

Yale School of Medicine

HOC in History

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

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

Accessing a HOC interpreter

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

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

oc>
```

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

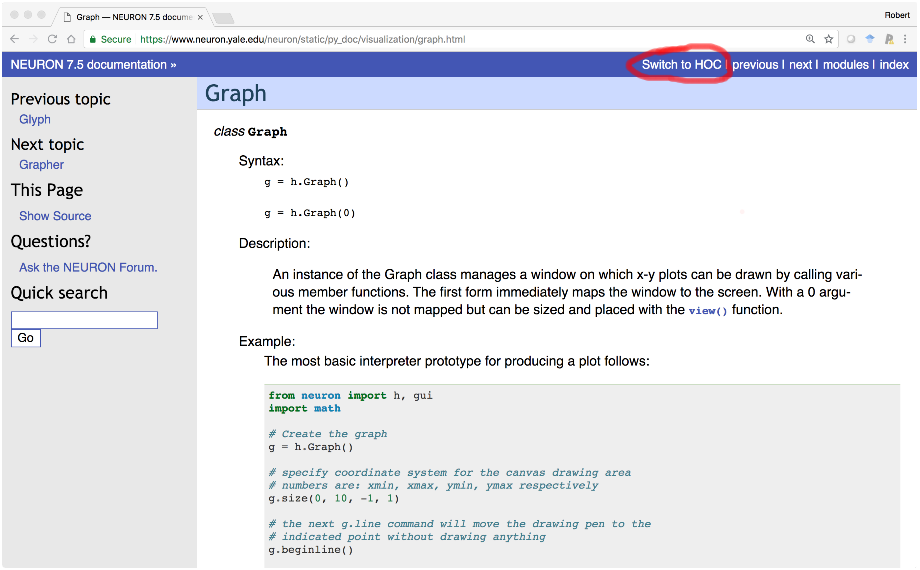
Note: launching `nrniv` does not load the compiled mechanisms automatically. To do that, launch `nrngui` instead.

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

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

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

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


Graph — NEURON 7.5 docume
Robert
Secure
https://www.neuron.yale.edu/neuron/static/py_doc/visualization/graph.html
NEURON 7.5 documentation »
Switch to HOC | previous | next | modules | index
Previous topic
Glyph
Next topic
Grapher
This Page
Show Source
Questions?
Ask the NEURON Forum.
Quick search
Go

Graph

class **Graph**

Syntax:

```
g = h.Graph()

g = h.Graph(0)
```

Description:

An instance of the Graph class manages a window on which x-y plots can be drawn by calling various member functions. The first form immediately maps the window to the screen. With a 0 argument the window is not mapped but can be sized and placed with the `view()` function.

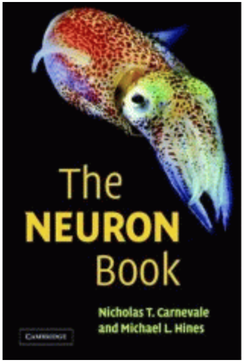
Example:

The most basic interpreter prototype for producing a plot follows:

```
from neuron import h, gui
import math

# Create the graph
g = h.Graph()

# specify coordinate system for the canvas drawing area
# numbers are: xmin, xmax, ymin, ymax respectively
g.size(0, 10, -1, 1)

# the next g.line command will move the drawing pen to the
# indicated point without drawing anything
g.beginline()
```

To learn more: The NEURON Book and ModelDB

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



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

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

Basic HOC syntax

Flow control

Familiar flow control statements are available in HOC:

if

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

for

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

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

Flow control

while

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

Grouping statements

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

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

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

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

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

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

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

Operators

Arithmetic operators are the same in HOC and Python:

`+   -   *   /   %`

Comparison operators are the same in HOC and Python:

`<   <=   ==   >=   >`

Logical operators are not the same:

HOC	Python
`&&`	`and`
`||`	`or`
`!`	`not`

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

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

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

HOC → Python gotchas: fuzzy comparisons

HOC allows fuzzy comparisons.

The variable `float_epsilon` sets the tolerance for equality.

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

Example:

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

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

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

Data types

HOC uses rigid data types.

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

```
x = 2
```

Strings must be declared before use:

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

Objects must also be declared:

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

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

Comments

HOC provides two forms of comments:

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

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

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

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

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

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

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

func and proc

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

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

func return_one() {return 1}
```

These are called with parentheses as in Python:

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

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

func and proc: arguments

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

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

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

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

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

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

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

HOC → Python gotchas: variable scoping

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

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

Local variables

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

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

HOC → Python gotchas: syntactic flexibility

HOC is relatively forgiving about syntax.

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

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

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

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

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

HOC → Python gotchas: syntactic flexibility

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

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

This is occasionally useful but often indicates a bug.

HOC → Python gotchas: syntactic flexibility

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

```
double x[10]
```

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

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

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

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

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

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

Using HOC to control NEURON

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

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

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

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

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

Special syntax for sections

Creating sections with HOC:

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

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

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

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

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

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

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

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

Using the currently accessed section

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

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

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

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

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

Connecting sections

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

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

Range variables

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

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

soma print v(0.5)
```

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

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

Pointers

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

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

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

Iterators

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

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

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

Coroutines are a related concept.

Looping over sections

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

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

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

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

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

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

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

Looping over segment locations

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

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

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

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

Templates

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

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

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

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

Example template courtesy of Bill Lytton.

HOC and Python interoperability via NEURON

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

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

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

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

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

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

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

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

HOC is not NEURON: data types

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

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

Accessing Python from HOC

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

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

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

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