The hoc programming language

Based on Kernighan and Pike's "high-order calculator"

Used in

- most published NEURON models
- most NEURON models currently under development
- almost all NEURON models currently available from ModelDB
- NEURON's standard run system and GUI tools
  UNIX / Linux / OS X: **`nrnxx/share/nrn/lib/hoc`**
  MSWin: **`nrnxx\lib\hoc`**

Where to learn more:

- Programmer's Reference link at
  `http://www.neuron.yale.edu/neuron/`
- The NEURON Book (especially chapters 12 and 13),
  papers about NEURON
- standard run system and GUI tools
- session files, hoc code exported from CellBuilder, Network Builder
- your own programming experiments

Kernighan and Pike's hoc: scalars, arrays, strings, procedures, functions

NEURON adds:

- **Domain-specific features**
  -- specification of model properties (geometry, topology, biophysics)
  -- event delivery system for synaptic connections,
  artificial spiking cells, and simulation flow control
  -- selection of numerical integration method
  -- elementary initialization and simulation execution
  -- parallel simulation of cell and/or network models
- **Objects**
  -- built-in and user-specified classes
- **Graphics**
  -- *interactive* plots of variables vs. time, distance ("space plot")
  or another variable (phase plane)
  -- shape plots (2D renderings of 3D shapes,
  may show a variable in false color)
  -- customizable GUI for building, analyzing, and using models

And it's

- extendable via NMODL, Channel Builder
- Interoperable with Python

Starting

UNIX / Linux / OS X / MSWin via the rxvt terminal

`nrniv` at the system prompt

OS X / MSWin

double click on the `nrngui` icon

Result:

```
bash-4.1$ nrniv
NEURON -- VERSION 7.3 (725:87d07a86a67e) 2012-08-03
Duke, Yale, and the BlueBrain Project -- Copyright 1984-2012
See http://www.neuron.yale.edu/credits.html

oc>
```

Stopping and exiting

Stopping runaway code

`^C` halts execution "safely"

`^C^C` halts "immediately"

Exiting hoc

`^D` or `quit()` at the `oc>` prompt

If NEURONMainMenu toolbar ("NMM") exists
    NMM / File / Quit

Getting code into NEURON

Built-in microemacs

`oc>em` starts it

See `em` in the Programmer's Reference, and

`http://www.neuron.yale.edu/neuron/static/docs/help/emacs.txt`

Program files

Plain text (ASCII)

OS X: drag & drop hoc file onto nrngui icon

MSWin: double click on hoc file in Windows Explorer

If NEURONMainMenu toolbar ("NMM") exists

NMM / File / load hoc or NMM / File / load session

`xopen("`*`filename`*`")` reads *`filename`* on every call

`load_file("`*`filename`*`")` reads *`filename`* only once per session

Interactive code entry

Type commands at the `oc>` prompt

Interactive code entry

Particularly useful for

- revising and debugging small chunks of code
- using "toy examples" to understand hoc syntax

Keyboard arrow keys:

↑ goes back in command history ("recalls commands"), ↓ goes forward

←, → moves cursor by 1 character, ^←, ^→ by 1 word

EMACS-style cursor control:

^P goes back in command history, ^N forward

^A moves cursor to line start, ^E to end

^B moves back 1 character, ^F forward

esc B moves back 1 word, esc F forward

^K kills to end of line and copies to buffer, ^Y pastes from buffer

Basic hoc syntax

Similar to C but no semicolons

Numbers

Numeric values are double precision floating point.

Interpreter (user input **bold**):	Remarks:
oc>**1+0.2**	note immediate evaluation
1.2	
oc>**1e-3**	scientific notation
0.001	
oc>**1E-3**	
0.001	
oc>**PI**	a built-in variable
3.1415927	(treat it like a constant)

User-created names

May refer to

- a number ("ordinary" variable or "scalar")
- an array of numbers
- a string
- a function or procedure
- a class ("template")
- an object reference

Naming rules

- start with an alpha character `[A-Za-z]`
- contain alphanumeric characters or underscore _
  `[A-Za-z0-9_]`
- < 100 characters long
- must not conflict with keywords or built-in functions
  --see Programmer's Reference entries on
  keywords-general, functions-general,
  keywords-neuron, and functions-neuron
- scope is global except for
  --variables declared local in a procedure or function
  --variables declared in a template
  "visibility" (public / private)

Create a scalar by assigning a value to a new name.

```
oc>x
/usr/local/nrn/i686/bin/nrniv: undefined variable x
 near line 14
 x
  ^
oc>

oc>x=2
first instance of x
oc>x
        2
oc>
```

Anything that isn't a scalar must be defined before use.

```
oc>double y[3]
oc>for i=0,2 y[i]=sqrt(i)
oc>for i=0,2 print y[i]
0
1
1.4142136

oc>strdef hello
oc>hello="hi"
oc>hello
hi
```

Notes:

- indices start at 0
- if *name* is an array, *name* is a shortcut for *name*`[0]`

```
oc>print y
0
```

A defined name cannot be redefined as something else.

```
oc>strdef y
/usr/local/nrn/i686/bin/nrniv: y  already declared
 near line 31
 strdef y
        ^

oc>proc hello() { print "hi" }
/usr/local/nrn/i686/bin/nrniv: syntax error
 near line 32
 proc hello() { print "hi" }
          ^
```

However, the body of a proc or func can be changed.

```
oc>proc foo() { print x^3 }
oc>foo()
8
oc>proc foo() { print x, exp(-x) }
oc>foo()
2 0.13533528
```

Expression: a combination of numbers, variables, operators, and functions that, when "executed," produces ("returns") a value of some type (number, string, object class).

```
1
x+2
sin(PI*0.4)
```

Statement: contains one or more expressions

Simple statements

```
x=3 // assignment statement
xopen("cell.hoc") // function call
```

Compound statements

```
{ a=5  b=sqrt(a) } // works, hard to read
{ // one statement/line is better
  a=5
  b=sqrt(a)
}
if (x<1) print "tiny" else print "big"
```

Program: a sequence of one or more statements

Statements that span multiple lines

Interactive code entry:

```
oc>proc foo() {
>       oc>print "hello, \
oc>world"
>       oc>}
oc>foo()
hello,
world
```

In a program:

```
i = p*z^2*(V*F^2/(R*T))*(si-so*exp(-z*V*F/(R*T))) \
       /(1 - exp(-z*V*F/(R*T))
```

Comments

```
// a one-line comment

/* another way to comment */

/* a comment
that spans two
or more lines */
```

Indentation

Do whatever you like, but make it readable.

```
{
  x = 5.1
  y = sqrt(3)
  print x*y
}
```

is better than

```
{ x = 5.1 y = sqrt(3) print x*y }
```

Operator precedence

operator	example	comment
( )	2*(x+0.1)	grouping
^	x^2	exponentiation
- !	-3, !x	unaryminus, not
* / %	5%3	multiply, divide, remainder
+ -		add, subtract
> >= < <= != ==	x==2	logical comparison
&&	(x>1) && (x<2)	AND
||	(x<0) || (x>1e6)	OR
=	x=2	assignment

Assignment statements

`x = expression`

Two shortcuts:

`x += a` same as `x = x+a`

`x *= a` same as `x = x*a`

Note: no space between + and =, or * and =

Logical expressions and comparisons

```
oc>x=2
oc>x==2 // does x equal 2?
        1
oc>x==2 // does x equal 3?
        0
```

Problem: roundoff error.

```
oc>x=sqrt(2)
```

Does x^2 equal 2?

```
oc>x^2-2
          4.4408921e-16
```

How to deal with this?

`float_epsilon` sets the threshold for deciding equality/inequality

```
oc>float_epsilon
        1e-11

oc>y=1+0.9*float_epsilon
oc>y==1
        1

oc>y=1+float_epsilon
oc>y==1
        0
```

So even though roundoff error makes sqrt(2)^2 - 2 nonzero,

```
oc>sqrt(2)^2==2
        1
```

Flow control

```
if (expr) stmt
if (expr) stmt1 else stmt2
while (expr) stmt
for (stmt1; expr2; stmt3) stmt
for var = expr1,expr2 stmt
for iterator_name ( . . . ) stmt
```

Examples:

```
i=0
while (i<10) { i+=1  print i }
for (i=0; i<10; i+=1) print i
for i=0,9 print i
```

```
oc>x=3
oc>if (x==3) print "x is 3"
        x is 3
```

What if you typed = when you meant == ?

```
oc>x=4
oc>if (x=3) print "x is 3"
        x is 3
```

. . . and it really will be 3.

Why? Assignment inside ( ) returns a value

```
oc>(x=3)
        3
```

`if (`*`number`*`)` treats a nonzero *`number`* as "true"

Flow control: `iterator`

```
// this is included in stdlib.hoc
iterator case() { local i
  for i=2,numarg() {
    $&1 = $1
    iterator_statement
  }
}
// next line requires scalar x to exist
for case(&x, 1, -1, E, R) print x
```

produces this output:

```
1
-1
2.7182818
8.31441
oc>
```

Functions and procedures

`func` *`name`*`() { `*`stmt`*` }`

where *`stmt`* includes a "return statement"

```
   return expr
```

that returns a value. Example:

```
oc>func three() { return 3 }
oc>three()
        3
```

`proc` *`name`*`() { `*`stmt`*` }`

where *`stmt`* does not include a return statement

Arguments to funcs and procs

- scalars, strings or objects
- retrieved by position

```
oc>func quotient() { return $1/$2 }
oc>quotient(1,3)
        0.33333333
```

Variable type	Name of *n*th argument	Call by
scalar	$*n*	value
scalar pointer	$&*n*	reference
string	$s*n*	reference
object reference	$o*n*	reference

Example: scalar and string as arguments

```
oc>proc printerr() { print "Error ", $1, "-- ", $s2 }
oc>printerr(3, "file not open")
Error -- file not open
```

Example: updating and reporting a count. Note call by reference.

```
tally=0
proc count() {
  $&2+=1 // affects arg 2
  print $s1, $&2
}
for i=0,2 count("updated count--", &tally)
```

produces this result:

```
updated count--1
updated count--2
updated count--3
```

Local variables

- temporary--exist only while the proc or func is executed
- scope is local to the proc or func,
  no conflict with globals that have the same names

Declare as part of the `proc` *`name`* `{` line.

```
i=10
proc squares() { local i
  for i=1,$1 print i*i
}
squares(3)
print "i is ", i
```

produces this result:

```
1
4
9
i is 10
```

Classes, objects, and object references in hoc

Class: a type or category.

Object: a specific instance of a type.

Object reference: a label or alias for an object,
not the object itself. Similar to pointer.

In hoc there are no "free-standing" objects.

- If an object exists, there must also be an objref that refers to it.
- If an object's reference count drops to 0, the object is destroyed.

Creating and destroying an object

```
oc>objref cells         // make new objref
oc>cells
        NULLobject

oc>cells = new List() // make new List object
oc>                   //   and associate cells with it
oc>cells              // verify
        List[8]

oc>List[8]            // just making sure . . .
        List[8]

oc>objref b           // make another objref
oc>b
        NULLobject

oc>b = cells          // associate b with the same List object
oc>b                  // verify the association
        List[8]
```

```
oc>objref b             // break link between b and List[8]
oc>b                    // verify
        NULLobject

oc>cells                // make sure cells is still associated
        List[8]

oc>objref cells         // break link between cells and List[8]
oc>cells                // verify
        NULLobject

oc>// the List's reference count should now be 0

oc>List[8]              // does the object still exist?
/usr/local/nrn/i686/bin/nrniv: Object ID doesn't exist: List[8]

 near line 15
 List[8]
        ^
```

Using an objref as an argument

```
func totalarea() { local tmp
  tmp = 0 // clear any leftover value
  for $o1.all for (x,0) tmp += area(x)
  return tmp
}
print "total area of ", cell, "is ", totalarea(cell)
```

Using call by reference to modify an object

```
proc scalediam() {
  for $o1.all diam *= $2
}
scalediam(cell, 2) // doubles diam of cell's sections
```

obfunc: a function that returns an object

```
// creates customized Graph that plots user-specified variable vs. time
// $s1        name of variable to be plotted
// $2 and $3  y axis min and max
// $4 and $5  screen coordinates of left upper corner
// $6 and $7  graph width and height
// assumes standard run system, so tstop and graphList[0] exist

GWIDTH=300.48 // default width and height of entire graph
GHEIGHT=200.32

obfunc makegraph() { localobj gtmp
  gtmp = new Graph(0) // creates but does not display
  gtmp.size(0,tstop,$2,$3) // axis scaling
  gtmp.view(0, $2, tstop, $3-$2, $4, $5, $6, $7) // draws on the screen
    // with user-specified location and size
  graphList[0].append(gtmp) // so it updates at integer multiples of dt
  gtmp.addexpr($s1, 1, 1, 0.8, 0.9, 2)
  return gtmp
}

objref g
g = makegraph("soma.v(0.5)", -80, 40, 300, 150, GWIDTH, GHEIGHT)
```

Comment: note use of localobj

Analyzing a program

Understanding how existing programs work is key to

- debugging and maintenance
- modifying them to handle other tasks
- learning by example

Case study: a hoc file generated by the CellBuilder

Aims:

- discover how the program works
- identify the programming tactics and strategies that are used in it

cell.hoc *part 1 of 2*

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}

create soma, dend

proc topol() { local i
  connect dend(0), soma(1)
  basic_shape()
}
proc basic_shape() {
  soma {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(15, 0, 0, 1)}
  dend {pt3dclear() pt3dadd(15, 0, 0, 1) pt3dadd(90, 0, 0, 1)}
}

objref all
proc subsets() { local i
  objref all
  all = new SectionList()
    soma all.append()
    dend all.append()
}
```

cell.hoc *part 2 of 2*

```
proc geom() {
  forsec all {  }
  soma {  L = 10  diam = 10  }
  dend {  L = 1000  diam = 1  }
}
proc geom_nseg() {
  forsec all { nseg = int((L/(0.1*lambda_f(100))+.999)/2)*2 + 1  }
}
proc biophys() {
  forsec all {
    cm = 1
  }
  soma {
    insert hh
      gnabar_hh = 0.12
      gkbar_hh = 0.036
      gl_hh = 0.0003
      el_hh = -54.3
  }
  dend {
    insert pas
      g_pas = 0.0001
      e_pas = -65
  }
}
access soma

celldef()
```

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}
```

The first thing the hoc interpreter encounters when it reads `cell.hoc`: a procedure definition.

None of the statements in `celldef()` will do anything until some other statement is executed that calls it.

Will that ever happen? Remains to be seen.

```
create soma, dend

proc topol() { local i
  connect dend(0), soma(1)
  basic_shape()
}
proc basic_shape() {
  soma {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(15, 0, 0, 1)}
  dend {pt3dclear() pt3dadd(15, 0, 0, 1) pt3dadd(90, 0, 0, 1)}
}

objref all
proc subsets() { local i
  objref all
  all = new SectionList()
    soma all.append()
    dend all.append()
}
```

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}
```

`create soma, dend`

The first statement in `cell.hoc` that is actually executed.
Time to start building an outline.

```
proc topol() { local i
  connect dend(0), soma(1)
  basic_shape()
}
proc basic_shape() {
  soma {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(15, 0, 0, 1)}
  dend {pt3dclear() pt3dadd(15, 0, 0, 1) pt3dadd(90, 0, 0, 1)}
}

objref all
proc subsets() { local i
  objref all
  all = new SectionList()
    soma all.append()
    dend all.append()
}
```

We need a way to record our discoveries that is quick and easy to set up, and quick and easy to understand.

An outline that summarizes the sequence of program execution is sufficient for most cases.

```
create soma, dend
```

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}

create soma, dend

proc topol() { local i
  connect dend(0), soma(1)
  basic_shape()
}
proc basic_shape() {
  soma {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(15, 0, 0, 1)}
  dend {pt3dclear() pt3dadd(15, 0, 0, 1) pt3dadd(90, 0, 0, 1)}
}

objref all
proc subsets() { local i
  objref all
  all = new SectionList()
    soma all.append()
    dend all.append()
}
```

More procedure definitions.
Skip them for now--we're looking for the next statement that is executed.

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}

create soma, dend

proc topol() { local i
  connect dend(0), soma(1)
  basic_shape()
}
proc basic_shape() {
  soma {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(15, 0, 0, 1)}
  dend {pt3dclear() pt3dadd(15, 0, 0, 1) pt3dadd(90, 0, 0, 1)}
}
```

`objref all`

```
proc subsets() { local i
  objref all
  all = new SectionList()
    soma all.append()
    dend all.append()
}
```

Not very exciting--just a declaration of what kind of variable `all` is--but let's add it to the outline.

And skip the definition of proc subsets(); we're still looking for the next instruction that is executed.

The updated outline.

```
create soma, dend
objref all
```

```
proc geom() {
  forsec all {  }
  soma {  L = 10  diam = 10  }
  dend {  L = 1000  diam = 1  }
}
proc geom_nseg() {
  forsec all { nseg = int((L/(0.1*lambda_f(100))+.999)/2)*2 + 1  }
}
proc biophys() {
  forsec all {
    cm = 1
  }
  soma {
    insert hh
      gnabar_hh = 0.12
      gkbar_hh = 0.036
      gl_hh = 0.0003
      el_hh = -54.3
  }
  dend {
    insert pas
      g_pas = 0.0001
      e_pas = -65
  }
}
access soma

celldef()
```

Three more procedure definitions.
For now, jump over these . . .

to this bonanza--two executed statements in a row!
The pace quickens . . .

The latest version of the outline.

```
create soma, dend
objref all
access soma
celldef()
```

We have to examine `proc celldef()`
to find out what happens next.

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}
```

`celldef()` makes a lot of things happen.
Let's add these to our outline.

```
create soma, dend

proc topol() { local i
  connect dend(0), soma(1)
  basic_shape()
}
proc basic_shape() {
  soma {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(15, 0, 0, 1)}
  dend {pt3dclear() pt3dadd(15, 0, 0, 1) pt3dadd(90, 0, 0, 1)}
}

objref all
proc subsets() { local i
  objref all
  all = new SectionList()
    soma all.append()
    dend all.append()
}
```

`celldef()` really expands the outline--
look at all the procs it calls:

```
create soma, dend
objref all
access soma
celldef()
```

- **`topol()`**
- **`subsets()`**
- **`geom()`**
- **`biophys()`**
- **`geom_nseg()`**

These five are indented because they are called by `celldef()`

Next we examine each of these five procs to see what they do, and discover if they call any other procs or funcs.

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}
```

Working our way through proc celldef() . . .

First it calls topol()

```
create soma, dend

proc topol() { local i
  connect dend(0), soma(1)
  basic_shape()
}
proc basic_shape() {
  soma {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(15, 0, 0, 1)}
  dend {pt3dclear() pt3dadd(15, 0, 0, 1) pt3dadd(90, 0, 0, 1)}
}

objref all
proc subsets() { local i
  objref all
  all = new SectionList()
    soma all.append()
    dend all.append()
}
```

`topol()` sets up the topology of the model, then calls `basic_shape()`.

Time to revise the execution outline again.

An aside: why is there an unused local i?

`basic_shape()` specifies how the model would look in a CellBuilder.

Note the use of "section stack" syntax to specify the currently accessed section.

Also note the compound statements.

Will soma length and diameter really be 15 um and 1 um, respectively?

The revised outline.

```
create soma, dend
objref all
access soma
celldef()
  topol()
    basic_shape()
  subsets()
  geom()
  biophys()
  geom_nseg()
```

indented because called by topol()

Next we examine `subsets()`

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}

create soma, dend

proc topol() { local i
  connect dend(0), soma(1)
  basic_shape()
}
proc basic_shape() {
  soma {pt3dclear() pt3dadd(0, 0, 0, 1) pt3dadd(15, 0, 0, 1)}
  dend {pt3dclear() pt3dadd(15, 0, 0, 1) pt3dadd(90, 0, 0, 1)}
}

objref all
proc subsets() { local i
  objref all
  all = new SectionList()
    soma all.append()
    dend all.append()
}
```

So this is where the various subsets (`SectionLists`, actually) are assembled.

Why is it useful to execute `objref all` inside this proc?

Why not just do `objref all` at the top level of the interpreter (i.e. outside of any proc or func or object), and be done with it?

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}

 . . .

proc geom() {
  forsec all {  }
  soma {  L = 10  diam = 10  }
  dend {  L = 1000  diam = 1  }
}
proc geom_nseg() {
  forsec all { nseg = int((L/(0.1*lambda_f(100))+.999)/2)*2 + 1  }
}
proc biophys() {
  forsec all {
    cm = 1
  }
  soma {
    . . .
```

`geom()` specifies the lengths and diameters of the model's sections.

Also, more examples of section stack syntax to specify the currently accessed section.

Why is there a `forsec all { }` that doesn't do anything?

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}
```

Hmm. `geom_nseg()` is defined before `biophys()`, but `biophys()` is called before `geom_nseg()`. Why is that?

. . .

```
proc biophys() {
  forsec all {
    cm = 1
  }
  soma {
    insert hh
      gnabar_hh = 0.12
      gkbar_hh = 0.036
      gl_hh = 0.0003
      el_hh = -54.3
  }
  dend {
    insert pas
      g_pas = 0.0001
      e_pas = -65
  }
}
```

More section stack syntax.

Applying it to a bunch of statements grouped by { } saves a lot of typing.

. . .

```
proc celldef() {
  topol()
  subsets()
  geom()
  biophys()
  geom_nseg()
}
  . . .
```

Last but not least, spatial discretization. `geom_nseg()` calls `lambda_f()`, which is included in `stdlib.hoc`, but we'll count it as a call anyway.

```
proc geom_nseg() {
  forsec all { nseg = int((L/(0.1*lambda_f(100))+.999)/2)*2 + 1  }
}
  . . .
```

The complete outline.

```
create soma, dend
objref all
access soma
celldef()
  topol()
    basic_shape()
  subsets()
  geom()
  biophys()
  geom_nseg()
    lambda_f()
```

So what can you do now?

Change

- number and names of sections
- model topology
- section geometry
- section biophysics
- discretization strategy

Analyze

- other code generated by the CellBuilder or other GUI tools,
  e.g. cell classes exported from the CellBuilder,
  network models exported from the Network Builder
- ses files saved from the GUI, e.g. the RunControl panel
- code mined from the hoc library (not just stdlib.hoc and stdrun.hoc)

to discover how to create special-purpose GUI tools
and solve more complex problems.

Practical suggestions

Strategy: design programs to have a modular structure.
Code that consists of short, relatively simple procedures or functions is easier to develop, debug, and understand.

Tactics: before writing any code, write an outline that breaks the task into manageable steps.
Any step that requires more than a couple of statements is a candidate for implementing as a proc or func.

> In retrospect, `proc celldef()` was pretty close to being the outline of `cell.hoc`