**EXERCISE 1: (i)** HOC – stands for ‘Higher Order Calculator’. It is an interpretive language, i.e., runs on its own and does not need to be compiled. Read through the material below to learn about the HOC commands, which are similar to the ones used in a high school calculator.

Note: You don’t have to do all, but enough so you are familiar with the HOC language.

(ii) Go through the Gilles & Stearatt NEURON tutorial and run all the programs described there.

**From the site:** **http://www.neuron.yale.edu/neuron/static/courses/2008/course/interpreter/hoc.html**

**HOC exercises**

**//** Executable lines below are shown with the hoc prompt **//** Typing these, although trivial, can be a valuable way to get familiar with the language

oc> **//** A comment

oc> /\* ditto \*/

**Data types: numbers strings and objects**

* **anything not explicitly declared is assumed to be a number**

oc> x=5300 **//** no previous declaration as to what 'x' is

* + **numbers are all doubles (high precision numbers)**

there is no integer type in Hoc

* + **Scientific notation use e or E**

oc> print 5.3e3,5.3E3 **//** e preferred (see next)

* + **there are some useful built-in values**

oc> print PI, E, FARADAY, R

* **Do you have anything to declare?: objects and strings**
  + **Must declare an object reference (=object variable) before making an object**
  + **Objref: manipulate references to objects, not the objects themselves**

often names are chosen that make it easy to remember what an object reference is to be used for (eg g for a Graph or vec for a Vector) but it's important to remember that these are just for convenience and that any object reference can be used to point to any kind of object

* + **Objects include vectors, graphs, lists, ...**

oc> objref XO,YO **//** capital 'oh' not zero

oc> print XO,YO **//** these are object references

oc> XO = new List() **//** 'new' creates a new instance of the List class

oc> print XO,YO **//** XO now points to something, YO does not

oc> objref XO **//** redeclaring an objref breaks the link; if this is the only reference to that object the object is destroyed

oc> XO = new List() **//** a new *new* List

oc> print XO **//** notice the List[#] -- this is a different List, the old one is gone

* + **After creating object reference, can use it to point a new or old object**

oc> objref vec,foo **//** two object refs

oc> vec = new Vector() **//** use 'new' to create something

oc> foo = vec **//** foo is now just another reference to the same thing

oc> print vec, foo **//** same thing

oc> vec=XO

oc> print vec, foo **//** vec no longer points to a vector

oc> objectvar vec **//** objref and objectvar are the same; redeclaring an objref breaks the link between it and the object it had pointed to

oc> print vec, foo **//** vec had no special status, foo still points equally well

* + **Can create an array of objrefs**

oc> objref objarr[10]

oc> objarr[0]=XO

oc> print objarr, objarr[0] **//** two ways of saying same thing

oc> objarr[1]=foo

oc> objarr[2]=objarr[0] **//** piling up more references to the same thing

oc> print objarr[0],objarr[1],objarr[2]

* + **Exercises: Lists are useful for maintaining pointers to objects so that they are maintained when explicit object references are removed**

1. Make vec point to a new vector. Print out and record its identity (*print vec*). Now print using the object name (ie *print Vector[#]* with the right #). This confirms that the object exists. Destroy the object by reinitializing the vec reference. Now try to print using the object name. What does it say.

2. As in Exercise 1: make vec point to a new vector and use print to find the vector name. Make XO a reference to a new list. Append the vector to the list: {XO.append(vec). Now dereference vec as in Exercise 1. Print out the object by name and confirm that it still exists. Even though the original objref is gone, it is still point to by the list.

3. Identify the vector on the list: (*print XO.object(0)*). Remove the vector from the list (*print XO.remove(0)*). Confirm that this vector no longer exists.

* + **Strings**
  + **Must declare a string before assigning it**

oc> mystr = "hello" **//** **ERROR**: needed to be declared

oc> strdef mystr **//** declaration

oc> mystr = "hello" **//** can't declare and set together

oc> print mystr

oc> printf("-%s-", mystr) **//** tab-string-newline; printf=print formatted; see documentation

* + **There are no string arrays; get around this using arrays of String objects**
  + **Can also declare number arrays, but vectors are often more useful**

oc> x=5

oc> double x[10]

oc> print x **//** overwrote prior value

oc> x[0]=7

oc> print x, x[0] **//** these are the same

**Operators and numerical functions**

oc> x=8 **//** assignment

oc> print x+7, x\*7, x/7, x%7, x-7, x^7 **//** doesn't change x

oc> x==8 **//** comparison

oc> x==8 && 5==3 **//** logical AND, 0 is False; 1 is True

oc> x==8 \\ 5==3 **//** logical OR

oc> !(x==8) **//** logical NOT, need parens here

oc> print 18%5, 18/5, 5^3, 3\*7, sin(3.1), cos(3.1), log(10), log10(10), exp(1)

oc> print x, x+=5, x\*=2, x-=1, x/=5, x **//** each changes value of x; **no** x++

**Blocks of code {}**

oc> { x=7

print x

x = 12

print x

}

**Conditionals**

oc> x=8

oc> if (x==8) print "T" else print "F" **//** brackets optional for single statements

oc> if (x==8) {print "T"} else {print "F"} **//** usually better for clarity

oc> {x=1 while (x<=7) {print x x+=1}} **//** nested blocks, statements separate by space

oc> {x=1 while (x<=7) {print x, x+=1}} **//** notice difference: comma makes 2 args of print

oc> for x=1, 7 print x **//** simplest for loop

oc> for (x=1;x<=7;x+=2) print x **//** (init;until;change)

**Procedures and functions**

oc> proc hello () { print "hello" }

oc> hello()

oc> func hello () { print "hello" return 1.7 } **//** functions return a number

oc> hello()

* **Numerical arguments to procedures and functions**

oc> proc add () { print $1 + $2 } **//** first and second argument, then $3, $4...

oc> add(5, 3)

oc> func add () { return $1 + $2 }

oc> print 7\*add(5, 3) **//** can use the returned value

oc> print add(add(2, 4), add(5, 3)) **//** nest as much as you want

* **String ($s1, $s2, ...) and object arguments ($o1, $o2, ...)**

oc> proc prstuff () { print $1, "::", $s2, "::", $o3 }

oc> prstuff(5.3, "hello", vec)

* **Exercises**

\*\*\* Use printf in a procedure to print out a formatted table of powers of 2

\*\*\* Write a function that returns the average of 4 numbers

\*\*\* Write a procedure that creates a section called soma and sets diam and L to 2 args

**Built-in object types: graphs, vectors, lists, files**

* **Graph**

oc> objref g[10]

oc> g = new Graph()

oc> g.size(5, 10, 2, 30) **//** set x and y axes

oc> g.beginline("line", 2, 3) **//** start a red (2), thick (3) line

oc> {g.line(6, 3) g.line(9, 25)} **//** draw a line (x, y) to (x, y)

oc> g.flush() **//** show the line

* **Exercises**
  + **write proc that draws a colored line ($1) from (0, 0) to given coordinate ($2, $3) assume g is a graph object**
  + **write a proc that puts up two new graphs**
  + **bring up a graph using GUI, on graph use right-button right pull-down to "Object Name"; set 'g' objectvar to point to this graph and use g.size() to resize it**
* **Vector**

oc> objref vec[10]

oc> for ii=0, 9 vec[ii]=new Vector()

oc> vec.append(3, 12, 8, 7) **//** put 4 values in the vector

oc> vec.append(4) **//** put on one more

oc> vec.printf **//** look at them

oc> vec.size **//** how many are there?

oc> print vec.sum/vec.size, vec.mean **//** check average two ways

oc> {vec.add(7) vec.mul(3) vec.div(4) vec.sub(2) vec.printf}

oc> vec.resize(vec.size-1) **//** get rid of last value

oc> for ii=0, vec.size-1 print vec.x[ii] **//** print values

oc> vec[1].copy(vec[0]) **//** copy vec into vec[1]

oc> vec[1].add(3)

oc> vec.mul(vec[1]) **//** element by element; must be same size

* **Exercises**
  + **write a proc to make $o1 vec elements the product of $o2\*$o3 elements**

(use resize to get $o1 to right size; generate error if sizes wrong eg

if ($o2.size!=$o3.size) { print "ERROR: wrong sizes" return }

* + **graph vector values: vec.line(g, 1) or vec.mark(g, 1)**

play with colors and mark shapes (see doc for details)

* + **graph one vec against another: vec.line(g, vec[1]); vec.mark(g, vec[1])**
  + **write a proc to multiply the elements of a vector by sequential values from 1 to size-1**

hint: use vec.resize, vec.indgen, vec.mul

**File**

oc> objref file

oc> mystr = "AA.dat" **//** use as file name

oc> file = new File()

oc> file.wopen(mystr) **//** 'w' means write, arg is file name

oc> vec.vwrite(file) **//** binary format

oc> file.close()

oc> vec[1].fill(0) **//** set all elements to 0

oc> file.ropen(mystr) **//** 'r' means read

oc> vec[1].vread(file)

oc> if (vec.eq(vec[1])) print "SAME" **//** should be the same

* **Exercises**
  + **proc to write a vector ($o1) to file with name $s1**
  + **proc to read a vector ($o1) from file with name $s1**
  + **proc to append a number to end of a file: tmpfile.aopen(), tmpfile.printf**

**List**

oc> objref list

oc> list = new List()

oc> list.append(vec) **//** put an object on the list

oc> list.append(g) **//** can put different kind of object on

oc> list.append(list) **//** pointless

oc> print list.count() **//** how many things on the list

oc> print list.object(2) **//** count from zero as with arrays

oc> list.remove(2) **//** remove this object

oc> for ii=0, list.count-1 print list.object(ii) // **remember** list.count, vec.size

* **Excercise**
  + **write proc that takes a list $o1 with a graph (.object(0)) followed by a vector (.object(1)) and shows the vector on the graph**
  + **modify this proc to read the vector out of file given in $s2**

**Simulation**

oc> create soma

oc> access soma

oc> insert hh

oc> ismembrane("hh") **//** make sure it's set

oc> print v, v(0.5), soma.v, soma.v(0.5) **//** only have 1 seg in section

oc> tstop=50

oc> run()

oc> print t, v

oc> print gnabar\_hh

oc> gnabar\_hh \*= 10

oc> run()

oc> print t, v **//** what happened?

oc> gnabar\_hh /= 10 **//** put it back

**Recording the simulation**

oc> cvode\_active(0) **//** this turns off variable time step

oc> dt = 0.025

oc> vec.record(&soma.v(0.5)) **//** '&' gives a pointer to the voltage

oc> objref stim

oc> soma stim = new IClamp(0.5) **//** current clamp at location 0.5 in soma

oc> stim.amp = 20 **//** need high amp since cell is big

oc> stim.dur = 1e10 **//** forever

oc> run()

oc> print vec.size()\*dt, tstop **//** make sure stored the right amount of data

**Graphing and analyzing data**

oc> g=new Graph()

oc> vec.line(g, dt, 2, 2)

oc> g.size(0, tstop, -80, 50)

oc> print vec.min, vec.max, vec.min\_ind\*dt, vec.max\_ind\*dt

oc> vec[1].deriv(vec, dt)

oc> print vec[1].max, vec[1].max\_ind\*dt **//** steepest AP

* **Exercises**
  + **change params (stim.amp, gnabar\_hh, gkbar\_hh), regraph and reanalyze**
  + **bring up the GUI and demonstrate that the GUI and command line control same parameters**
  + **write proc to count spikes and determine spike frequency (use vec.where)**

**Roll your own GUI**

oc> proc sety () { y=x print x }

oc> xpanel("test panel")

oc> xvalue("Set x", "x")

oc> xvalue("Set y", "y")

oc> xbutton("Set y to x", "sety()")

oc> xpanel()

* **Exercises**
  + **put up panel to run sim and display (in an xvalue) the average frequency**

*Last updated: Jun 16, 2003 (11:09)*