Chapter 12

hoc, NEURON's interpreter

Much of the flexibility of NEURON is due to its use of a built-in interpreter, called hoc (pronounced "hoak"), for defining the anatomical and biophysical properties of models of neurons and neuronal networks, controlling simulations, and creating a graphical user interface. In this chapter we present a survey of hoc and how it is used in NEURON. Readers who seek the most up-to-date list of hoc keywords and documentation of syntax are referred to the online Programmer's Reference (see link at http://www.neuron.yale.edu/neuron/docs/docs.html). This can also be downloaded as a pkzip archive for convenient offline viewing with any WWW browser. The standard distribution for MSWindows includes a copy of the Programmer's Reference which is current as of the date of the NEURON executable that it accompanies (see the "Documentation" item in the NEURON program group).

NEURON's hoc is based on the floating point calculator by the same name that was developed by Kernighan and Pike (1984). The original hoc has a C-like syntax and is very similar to the bc calculator. The latest implementation of hoc in NEURON contains many enhancements and extensions beyond its original incarnation, both in added functions and additions to the syntax. Despite these enhancements, for the most part programs written for versions as far back as 2.x will work correctly with the most recent release of NEURON.

One important addition to hoc is an object-oriented syntax, which first appeared in version 3 of NEURON. Although it lacks inheritance, hoc can be used to implement abstract data types and encapsulation of data (see **Chapter 13**). Other extensions include functions that are specific to the domain of neural simulations, and functions that implement a graphical user interface. Also, the user can build customized hoc interpreters that incorporate special functions and variables which can be called and accessed interactively. As a result of these extensions, hoc in NEURON has become a powerful language for implementing and exercising models.

NEURON simulations are not subject to the performance penalty often associated with interpreted (as opposed to compiled) languages because computationally intensive tasks are carried out by highly efficient, precompiled code. Some of these tasks are related to integration of the cable equation, and others are involved in the emulation of biological mechanisms that generate and regulate chemical and electrical signals.

In this context, several important facts bear mention. First, a large part of what constitutes the NEURON simulation environment is actually written in hoc. This includes the standard run system (an extensive library of functions for initializing and controlling simulations--see **Chapters 7** and **8**), and almost the entire suite of GUI tools (the sole exception being the Print & File Window Manager, which is implemented in C).

Second, the GUI tools for building models of cells and networks (which are, of course, all written in hoc) actually work by constructing hoc programs.

Finally, and perhaps most important, all of the hoc code that defines the standard run system and GUI tools is provided in plain text files ("hoc libraries") that accompany the standard distribution of NEURON. Under UNIX/Linux these are located in nrn-x.x/share/nrn/lib/hoc/, and in MSWindows they are in c:\nrnxx\lib\hoc\. Users can readily review the implementational details of the functions and procedures that are defined in the standard libraries, and, if necessary, modify and replace them. Since hoc is an interpreter, it is easy to make such changes without having to alter the actual files that contain the standard libraries themselves. Instead, just write hoc code that defines functions or procedures with the same names as the ones that are to be replaced, and put this in a new file. The only caveat is to be sure to load the alternatives *after* the standard library. For example, to replace the init() procedure (see **Examples of custom initializations** in **Chapter 8**), the text of the new procedure should occur sometime after the statement load_file("nrngui.hoc"). If the new definition is read prior to loading the library version, the library version will overwrite the user version instead of the other way around.

The interpreter

The hoc interpreter has served as the general input/output module in many kinds of applications, and as such is directly executed under many different names, but we will confine our attention to its use in NEURON. The simplest interface between hoc and domain-specific problems consists of a set of functions and variables that are callable from hoc. This was the level of implementation of the the original CABLE program (NEURON version 1). NEURON version 2 broke from this style by introducing neuron-specific syntax into the interpreter itself. This allowed users to specify cellular properties at a level of discourse more appropriate to neurons, and helped relieve the confusion and reduce the mental energy required to constantly shift between high level neural concepts and their low level representation in the computer. NEURON version 3 added object syntax to allow much better structuring of the conceptual pieces that must be assembled in order to build and use a model.

Installing NEURON under UNIX or Linux results in the construction of several programs, but the principal one that we are concerned with in this book is nrniv, which is located in nrn/i686/bin. This is the main executable, which contains the hoc interpreter with all of its extensions. Since the bulk of the code needed by NEURON is in shared libraries, nrniv and the various "special" executables created by nrnivmodl (see **Adding new mechanisms to the interpreter** below) are very small.

Under Linux, nrniv can add new mechanisms and functions to hoc by dynamically loading shared objects that have been compiled from model description files (see **Adding new mechanisms to the interpreter**; also see **Chapter 9**). For example, the demonstration program that comes with NEURON is started by executing neurondemo. This is actually a shell script that starts nrniv with a command line that makes it load a shared object that contains additional biophysical mechanisms. Under non-Linux UNIX there is great

variation in how, or even if it is possible, to dynamically load shared objects. Therefore in those environments `neurondemo` is a complete duplicate of `nrniv` plus the extra mechanisms needed by the demonstration.

Under MSWindows, the program that corresponds to `nrniv` is `nrniv.exe`. There is also a program called `neuron.exe`, which is a short stub that starts a Cygwin terminal window (see `http://cygwin.com/`) and then runs `nrniv.exe` in that window. It is `neuron.exe` that is the target of icons and shortcuts used to start NEURON. As with the Linux version, `nrniv.exe` can load new mechanisms dynamically (see next section).

Adding new mechanisms to the interpreter

To add new mechanisms, you first write a specification of the mechanism properties in the NMODL language (see **Chapter 9**), and then you compile it. To compile under UNIX and Linux, you execute the shell script `nrnivmodl`, which is located in `nrn/i686/bin`. Most often, `nrnivmodl` is called with no file name arguments, which results in compilation of all "`mod` files" in the current working directory (i.e. files that have the suffix `.mod`).

It can also be called with one or more file name arguments, e.g.

```
nrnivmodl file1 file2 . . .
```

compiles the model descriptions defined in `file1.mod`, `file2.mod`, etc.. Regardless of how `nrnivmodl` is invoked, the first step in the process is translation of the model descriptions from NMODL into C by the `nocmodl` translator.

Under Linux, the end result is a shared object located in a subdirectory `.i686/.libs` of the current working directory, as well as a shell script called `special` in the `.i686` subdirectory that starts `nrniv` and makes it load the shared object. Under non-Linux UNIX, the result is a complete executable called `special`.

The MSWindows version of `nrnivmodl` is called `mknrndll`. It compiles and links the models into a dynamically loadable library called `nrnmech.dll`. `neuron.exe` automatically looks in the current working directory for a `nrnmech.dll` file, and if one exists, loads it into memory and makes the mechanisms available to the interpreter. More than one `dll` file can be loaded by listing them after the `-dll` argument to `neuron.exe` when it is run.

The stand-alone interpreter

The rest of this chapter describes general aspects of the interpreter that are common to all applications that contain it. Although for concreteness we use `nrniv` or `neuron.exe`, all the examples and fragments can be typed to any program that contains the interpreter, such as `oc`.

Starting and exiting the interpreter

Under UNIX and Linux, hoc is started by typing the program name in a terminal window

```
nrniv [filenames] [-]
```

where the brackets indicate optional elements. When there are no file name arguments, hoc takes its commands from standard input and prints its results to standard output. With file name arguments, the files are read in turn and the commands executed. After the last file is executed, hoc exits. The - signals that commands are to be taken from standard input until an EOT character (^D) is encountered. One can also exit by executing quit().

When starting hoc with arguments it is easy to forget the final - and be surprised when the program quickly exits. Generally the - is omitted only when running the interpreter in batch mode under control of a shell script.

With the MSWindows version (neuron.exe), omitting the trailing - does not cause the program to exit. This makes it more convenient to attach neuron.exe to hoc files so that one can start the program and read a hoc file by merely clicking on the file's name in a file manager such as Windows Explorer. Also, neuron.exe starts a Cygwin terminal window into which one can type hoc commands. Exiting can be done by typing ^D or quit() at the interpreter's oc> prompt. If the NEURON Main Menu is present, one can also exit by selecting File / Quit; this works under all operating systems.

On startup, NEURON prints a banner that reports the current version and last change date.

```
NEURON --  Version 5.6 2004-5-19 23:5:24 Main (81)
by John W. Moore, Michael Hines, and Ted Carnevale
Duke and Yale University -- Copyright 2001

oc>
```

The oc> prompt at the beginning of a line means the interpreter is waiting for a command. This is sometimes called "immediate mode" to signify that commands are evaluated and executed (if valid) immediately, as shown in the following listing (user entries are **bold** while the interpreter's output is plain).

```
oc>2
        2
oc>1+2
        3
oc>x=2
first instance of x
oc>x
        2
oc>x*x
        4
oc>
```

The interpreter has a "history function" that allows scrolling through previous commands by pressing the keyboard's up and down arrow keys. This facilitates repeating prior commands, with or without modification. For example, in


```
oc>proc foo() { print x^3 }
oc>foo()
8
oc>proc foo() { print x^4 }
oc>foo()
16
oc>
```

line 1 defines a new procedure that prints the value of the cube of `x`, line 2 calls this procedure, and line 3 shows the numeric result. The fourth line was created by pressing the up arrow key twice, to recall the first line. Then the left arrow key was pressed twice to move the editing cursor (blinking vertical line on the monitor) just to the right of the `3`. At this point, pressing the backspace key deleted the `3`, and pressing the numeral `4` on the keyboard changed the exponent to a `4`. Finally the return ("enter") key was pressed, and the interpreter responded with an `oc>` prompt. Now typing the command `foo()` produced a new numeric result.

In immediate mode, each statement must be contained in a single line. Very long statements can be assembled by using the continuation character \ (backslash) to terminate all but the last line. Thus in

```
oc>proc foo() { print x^4 \
oc>, x }
oc>foo()
16 2
oc>
```

the interpreter merges the first and second lines into the single line

```
proc foo() { print x^4 , x }
```

Quoted strings that are constructed with continuation characters have a limit of 256 characters, and each continuation character becomes an embedded newline (line break).

Error handling

This is one of many areas where `hoc` falls short. Debugging large programs is difficult, so it is best to practice modular programming, breaking code into short procedures and functions.

`hoc` is implemented as a stack machine. This means that commands are first parsed into a more efficient stack machine representation, and subsequently the stack machine is interpreted.

Errors found during parsing are called parse errors. These range from invalid syntax

```
oc>1++1
nrniv: parse error near line 3
1++1
   ^
oc>
```

to the use of undefined names


```
oc>print x[5], "hello"
nrniv: x not an array variable near line 9
print x[5], "hello"
           ^
```

Such errors are usually easy to fix since they stop the parser immediately, and the error message, which always refers to a symptom, generally points to the cause. Error messages specify the current line number of the file being interpreted and print the line along with a carat pointing to the location where the parser failed. This is usually an important clue, but failure may not occur until several tokens after the actual mistake. One common parse error in apparently well-formed statements results from using a name of the wrong type, e.g. specifying a string where a scalar variable is required.

Errors during interpretation of the stack machine are called run-time errors:

```
oc>sqrt(-1)
sqrt: DOMAIN error
nrniv: sqrt argument out of domain near line 5
sqrt(-1)
       ^
```

Generally, run-time error messages are more pertinent to that actual problem than are syntax error messages, although logic errors can be very difficult to diagnose. These errors usually occur within a function, and the error message prints the call chain

```
oc>proc p() {execute("sqrt(-1)")}
oc>p()
sqrt: DOMAIN error
nrniv: sqrt argument out of domain near line 8
{sqrt(-1)}
          ^
        execute("sqrt(-1)")
      p()
nrniv: execute error: sqrt(-1) near line 8
^
oc>
```

Unfortunately there is no trace facility to help debug run-time errors, and the line number is of no help at all because it refers to the last line that was parsed, instead of the location of the offending statement.

Interpretation of a `hoc` program may be interrupted by typing one or two `^C` at the terminal. For example if the interpreter is in an infinite loop, as in

```
oc>while(1) {}
```

a *single* `^C` will stop it

```
^Cnrniv: interrupted near line 2
while(1) {}

oc>
```

Generally one `^C` is preferred, because this allows the interpreter to reach a safe place before it halts execution. Two `^C` will interrupt the interpreter immediately, even if it is in the middle of updating an internal data structure. There are two situations in which the second `^C` may be necessary:

1. if the program is waiting inside a system call, e.g. waiting for console input.
2. if the program is executing a compiled function that is taking so long that program control doesn't reach a known safe place in a reasonable time.

Syntax

Names

A name is a string that starts with an alpha character and contains fewer than 100 alphanumeric characters or the underscore _. A user-created name can be associated with any one of the following:

global scalar (available to all procedures/functions)
local scalar (created/destroyed on procedure entry/exit)
array
string
function or procedure
template (class or type)
object reference

User-created names must not conflict with keywords or built-in functions. Names have global scope except if a `local` declaration is used to create a local scalar within a procedure or function, or when the name is declared within a template (i.e. class definition, although one then speaks of visibility instead of scope, and the distinction is between public and private).

Keywords

The `hoc` interpreter in the current version of NEURON has many keywords that have been added over the years. It is helpful to have a general idea of what these are useful for, and specific knowledge of where they are declared. This first table presents the most basic keywords, built-in constants, and functions of the `hoc` interpreter with object extensions and elementary functionality for neuronal modeling; the authoritative list is in `nrn-x.x/src/oc/hoc_init.c`.

General declaration

```
proc            func            local
double          strdef          iterator
eqn             depvar
```

Flow control

```
return          break           stop      continue
if              else            for       while
iterator
```

Built-in constants

```
PI              E               GAMMA             DEG
PHI             FARADAY         R
```


Built-in variables

```
float_epsilon        hoc_ac_
```

Built-in functions

```
sin                  cos              atan
log                  log10            exp                sqrt
int                  abs              erf                erfc
use_mcell_ran4       mcell_ran4       mcell_ran4_init
variable_domain      units
prmat                solve            eqinit
sred                 xred
chdir                getcwd           neuronhome
ropen                wopen            xopen
load_proc            load_func        load_template
load_file            load_java
getstr               strcmp
printf               fprint           fscan
ivoc_style
save_session         print_session
xpanel               xcheckbox
xbutton              xstatebutton     xradiobutton
xmenu                xlabel           xvarlabel          xslider
xvalue               xpvalue          xfixedvalue
doEvents             doNotify
numarg               symbols
object_id            object_push      object_pop
allobjectvars        allobjexts       name_declared
boolean_dialog       continue_dialog  string_dialog
pwman_place          startsw          stopsw
execute              execute1
machine_name         saveaudit        retrieveaudit
show_errmess_always  coredump_on_error
checkpoint           system           quit
```

Miscellaneous

```
print                read             delete
em                   debug
```

Object-oriented

```
begintemplate        endtemplate
public               external
objectvar            objref           new
```

Neuron-specific

```
create               connect
access               setpointer
```

```
insert              uninsert
forall              ifsec           forsec    secname
```

The following functions and variables are specific to modeling neurons; the authoritative list of these is in `nrn-x.x/src/nrnoc/neuron.h`.

Variables

```
t                   dt              secondorder       stoprun
celsius             diam_changed
```

Functions

```
pt3dclear           pt3dadd         p3dconst
x3d                 y3d             z3d               diam3d
n3d                 arc3d
define_shape
spine3d             setSpineArea    getSpineArea
initnrn             distance        area
topology            ri
issection           ismembrane      sectionname       psection
disconnect          delete_section
pop_section         push_section
this_section        this_node
parent_section      parent_node     parent_connection
section_orientation
ion_style           nernst          ghk
finitialize         fadvance
batch_run           batch_save
fit_praxis          attr_praxis
stop_praxis         pval_praxis
```

Mechanism types and variables are defined in `nrn-x.x/src/nrnoc` by `capac.c`, `extcelln.c`, `hh.mod`, and `pas.mod`. This directory also contains several `mod` files that define neuron-specific point process classes such as `IClamp`, `SEClamp`, and `AlphaSynapse`.

There are also several other built-in object classes, including neuron-specific examples like `SectionList`, `SectionRef`, and `Shape`, and more generic classes such as `List`, `Graph`, `HBox`, `File`, `Random`, and `Vector`. The Programmer's Reference (see link at `http://www.neuron.yale.edu/neuron/docs/docs.html`)

Variables

Double precision variables are defined when a name is assigned a value in an assignment expression, e.g.

```
var = 2
```

Such scalars are available to all interpreted procedures and functions, i.e. they have global scope.

There are several built-in variables that should be treated as constants:

`FARADAY`	coulombs/mole
`R`	molar gas constant, joules/mole/deg-K
`DEG`	`180/PI`, i.e. degrees per radian
`E`	base of natural logarithms
`GAMMA`	Euler constant
`PHI`	golden ratio
`PI`	circular transcendental number
`float_epsilon`	resolution for logical comparisons and `int()`

Arbitrarily dimensioned arrays are declared with the `double` keyword, as in

```
double vector[10], array[5][6], cube[first][second][third]
```

Array elements are initialized to 0. Array indices are truncated to integers and run from 0 to the declared value minus 1. When an array name is used without an index, the index is assumed to be 0. Arrays can be dynamically re-dimensioned within procedures.

String variables are declared with the `strdef` keyword, e.g.

```
strdef st1, st2
```

Assignments are made to string variables, as in

```
st1 = "this is a string"
```

String variables may be used in any context that requires a string, but no operations, such as addition of strings, are available (but see `sprint()` below).

After a name has been defined as a scalar, string, or array, it cannot be changed to another type. The `double` and `strdef` keywords can appear within a compound statement and are useful for throwing away previous data and reallocating space. However the names must originally have been declared outside any `func` or `proc` before they can be redeclared (as the same type) in a procedure. These restrictions also apply to object references (`objref`s--see **Declaring an object reference** in **Chapter 13**).

Expressions

The arithmetic result of an expression is immediately typed on standard output unless the expression is embedded in a statement or is an assignment expression. Thus

```
2*5
```

typed at the keyboard prints

```
10
```

and

```
sqrt(4)
```

yields

```
2
```


The operators used in expressions are, in order of precedence from high to low,

`()`	function call
`^`	exponentiation (right to left precedence)
`- !`	unaryminus, logical negation ("not")
`* / %`	multiplication, division, remainder
`+ -`	addition, subtraction
`> >= < <= != ==`	logical comparison
`&&`	logical AND
`||`	logical OR
`=`	assignment (right to left precedence)

Logical expressions are valued 1.0 (TRUE) or 0.0 (FALSE), and a nonzero value is treated as TRUE. The remainder `a%b` is in the range $0 \le$ `a%b` $<$ `b` and can be thought of as the value that results from repeatedly subtracting *or* adding `b` until the result is in the range [0, `b`). This differs from the C syntax in which `(-1)%5` is -1. For us, `(-1)%5` is 4.

Logical comparisons of real values are inherently ambiguous due to roundoff error. Roundoff can also be a problem when computing integers from reals and indices for vectors. For this reason the built-in global variable `float_epsilon` is used for logical comparisons and computing vector indices. The constant ϵ in this table stands for `float_epsilon`, which has a default value of 10^{-11} but can be assigned a different value by the user.

`hoc`	math or C equivalent
`x == y`	$-\epsilon \le x - y \le \epsilon$
`x < y`	$x < y - \epsilon$
`x <= y`	$x \le y + \epsilon$
`x != y`	$x < y - \epsilon$ or $x > y + \epsilon$
`x > y`	$x > y + \epsilon$
`x >= y`	$x \ge y - \epsilon$
`int(x)`	$(int)(x + \epsilon)$
`a[x]`	$a[(int)(x + \epsilon)]$

Statements

A statement terminated with a newline is executed immediately. A group of statements separated by newlines or whitespace and enclosed in curly brackets `{}` form a compound statement, which is not executed until the closing `}` is typed. Statements typed interactively do not produce a value. An assignment is parsed by default as a statement

rather than an expression, so assignments typed interactively do not print their values. Note, though, the expression

```
(a = 4)
```

would print the value

```
4
```

An expression is treated as a statement when it is within a compound statement.

Comments

Text between /* and */ is treated as a comment.

```
/* a single line comment */
/* this comment
   spans
   several lines */
```

Comments to the end of the line (single line comments) may be started by the double slash, as in

```
print PI    // this comment is limited to one line
```

Flow control

In the syntax below, `stmt` stands for either a simple or compound statement.

```
if (expr) stmt
if (expr) stmt1 else stmt2
while (expr) stmt
for (expr1; expr2; expr3) stmt
for var = expr1, expr2, expr3 stmt
for iterator_name( . . . ) stmt
```

In the `if` statement, `stmt` is executed only if `expr` evaluates to a non-zero value. The `else` form of the `if` statement executes `stmt1` when `expr` evaluates to a non-zero (TRUE) value; otherwise, it executes `stmt2`.

The `while` statement is a looping construct that repeatedly executes `stmt` as long as `expr` is TRUE. The `expr` is evaluated prior to each execution of `stmt`, so if `expr` is 0 on the first pass, `stmt` will not be executed even once.

The general form of the `for` statement is executed as follows: The first `expr` is evaluated. As long as the second `expr` is true the `stmt` is executed. After each execution of the `stmt`, the third `expr` is evaluated.

The short form of the `for` statement is similar to the `DO` loop of FORTRAN but is often more convenient to type. However, it is very restrictive in that the increment can only be unity. If `expr2` is less than `expr1` the `stmt` will not be executed even once. Also the expressions are evaluated once at the beginning of the `for` loop and not reevaluated.

The iterator form of the `for` statement is an object-oriented construct that separates the idea of iteration over a set of items from the idea of what work is to be performed on each item. As such, it is most useful for dealing with objects that are collections of other

objects. It is also useful whenever iteration over a set of items has a nontrivial mapping to a sequence of numbers. As a concrete example of this, let us define an iterator called `case`. To do this, we use the `hoc` keywords `iterator` and `iterator_statement`, e.g. like this

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

It is easy to use this iterator to loop over small sets of unrelated integers, as in

```
for case(&x, 1, -1, 3, 25, -3) print x
```

Of course, this requires that `x` has already been used as a scalar variable (otherwise the expression `&x` will be invalid) An alternative would be the relatively tedious

```
double num[5]
num[0] = 1
num[1] = -1
num[2] = 3
num[3] = 25
num[4] = -3
for i = 0, 4 {
   x = num[i]
   print x
}
```

We should point out that `iterator case()` is already included in `stdlib.hoc` (in `nrn-x.x/share/lib/hoc/` (UNIX/Linux) or `c:\nrnxx\lib\hoc\` (MSWindows)). This is automatically available after `nrngui.hoc` has been loaded.

These statements are used to modify the normal flow of control:

`break`	Exit from the enclosing `while` or `for` loop.
`continue`	Jump to end of the enclosing `while` or `for`.
`return`	Exit from the enclosing procedure.
`return expr`	Exit from the enclosing function.
`stop`	Exit to the top level of the interpreter.
`quit()`	Exit from the interpreter.

Functions and procedures

The definition syntax is

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

Functions must return a value via a

```
return expr
```

statement. Procedures do not return a value. As a trivial example of a function definition, consider


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

This defines the function `three()` which returns a fixed numeric value. Typing the name of this function at the `oc>` prompt will cause its returned value to be printed.

```
oc>three()
   3
oc>
```

Notice the recommended placement of `{}`. The opening `{` must appear on the same line as the statement to which it is a part. This also applies to conditional statements. The closing `}` is free form, but clarity is best served if it is placed directly under the beginning of the statement it closes and interior statements are indented.

Arguments

Scalars, strings, and objects can be passed as arguments to functions and procedures. Arguments are retrieved positionally, e.g.

```
func quotient() {
  return $1/$2
}
```

defines the function `quotient()` which expects two scalar arguments. The `$1` and `$2` inside the function refer to the first and second arguments, respectively.

Formally, an argument starts with the letter `$` followed by an optional `&` (the "pointer operator") to refer to a scalar pointer, followed by an optional `s` or `o` that signifies a string or object reference, followed by an integer. Thus a string argument in the first position would be known as `$s1`, while an object argument in the third position would be `$o3`.

For example,

```
proc printerr(){
  print "Error ", $1, "-- ", $s2
}
```

defines a procedure that expects a scalar for its first argument and a string for its second argument. If we invoke this procedure with the statement `printerr(29, "too many channels")`, it will print the message `Error 29 : too many channels`.

There is also a "symbolic positional syntax" which uses the variable `i` in place of the positional constant to denote which argument is to be retrieved, e.g. if `i` equals 2, then `$i` and `$2` refer to the same argument. The value of `i` must be in the range [1, `numarg()`], where `numarg()` is a built-in function that returns the number of arguments to a user-written function. This usage literally requires the symbol `$i`; `$` plus any other letter (e.g. `$j` or `$x`) will not work. Furthermore, `i` must be declared `local` to the function or procedure.

The function `numarg()` can be called inside a user-written function or procedure to obtain the number of arguments. Thus if we declare


```
proc countargs(){
  print "Number of arguments is ", numarg()
}
```

and then execute `countargs(x, sin(0.1), 9)`, where `x` is a scalar that we have defined previously, NEURON's interpreter will print `Number of arguments is 3`. Generally `numarg()` is used in procedures and functions that employ symbolic positional syntax, as in

```
proc printargs() { local i
  for i = 1, numarg() print $i
}
```

If we execute `printargs(PI, -4, sqrt(5))`, the interpreter will respond by printing

```
3.1415927
-4
2.236068
```

Similarly, we could define a function

```
proc printstrs() { local i
  for i = 1, numarg() print $si
}
```

and then execute `printstrs("foo", "faugh", "fap")` to get the printed output

```
foo
faugh
fap
```

Call by reference vs. call by value

Scalar arguments use call by value so the variable in the calling statement cannot be changed. If the calling statement has a `&` prepended to the variable, that variable is passed by reference and must be retrieved with the syntax `$&1`, `$&2`, etc..

If the variable passed by reference is a one-dimensional array (i.e. a double), then `$&1` refers to its first (0th) element and the `j`th element is denoted `$&1[j-1]`. Be warned that there is ***no*** array bounds checking, and the array is treated as being one-dimensional. A scalar or array reference may be passed to another procedure with `&$&1`. To save a scalar reference, use the `Pointer` class.

Arguments of type `strdef` and `objref` use call by reference, so the calling value may be changed by the called `func` or `proc`. Objects are discussed in **Chapter 13**.

Local variables

Local variables maintained on a stack can be defined with the `local` statement. The `local` statement must be the first statement in the function and on the same line as the `proc` statement. For example, in

```
proc squares() { local i, j, k /* print squares up to arg */
   for (i=1; i <= $1; i=i+1) print i*i
}
```

declaring `i`, `j`, and `k` to be `local` insures that this procedure does not affect any previously defined global variables with these names.

Recursive functions

User defined functions can be used in any expression, so functions can be called recursively. For example, the factorial function can be defined as

```
func fac() {
  if ($1 == 0) {
    return 1
  } else {
    return fac($1-1)*$1
  }
}
```

and the call

```
fac(3)
```

would produce

```
6
```

It would be a user error to call this function with a negative or non-integer argument. Besides the fact that the algorithm is numerical nonsense for those values, in theory the function would never return since the recursive argument would never be 0. Actually, after some time the stack frame list would overflow and an error message would be printed, as in

```
oc>fac(-1)
nrnoc: fac call nested too deeply near line 10
fac(-1)
        ^
        fac(-99)
      fac(-98)
    fac(-97)
  fac(-96)
and others
oc>
```

Input and output

The following describes simple text-based input and output. User interaction is better performed with the graphical interface, and dealing with multiple files requires use of the `File` class.

Standard `hoc` supplied `read()` and `print`, which use standard input and output, respectively. Their use is illustrated by this example

```
while (read(x)) {
  print "value is ", x
}
```

The return value of `read()` is 1 if a value was read, and 0 if there was an error or end of file (EOF). The `print` statement takes a comma-separated list of arguments that may be strings or variables. A newline is printed at the end.

For greater flexibility, the following built-in functions are also available.

`printf("format string", arg1, arg2, . . . )`

`printf()` is compatible with the standard C library function of the same name. It allows `f`, `g`, `d`, `o`, and `x` formats for scalar arguments, and the `s` format for strings. All the `%` specifications for field width apply.

`fprint("format string", arg1, arg2, . . . )`

`fprint()` is similar to `printf()`, but its output goes to the file that was opened by `wopen("filename")`. Such files are closed by `wopen()` with no arguments, or by the alternative `wopen("")`. When no write file is open, `fprint()` defaults to standard output. `wopen()` returns 0 on failure of the attempted open.

`sprint(strdef, "format string", arg1, . . . )`

This function is very useful for building file names, and even command strings, out of other variables. For example, if data files are to be named `drat.1`, `drat.2`, etc., the names can be generated with variables in the following manner.

```
strdef filename, prefix
prefix = "rat"
num = 1
sprint(filename, "d%s.%d", prefix, num)
```

After execution of these statements the, string variable `filename` contains `drat.1`.

`fscan()`

`fscan()` returns the value read sequentially from the file that was opened by `ropen("filename")`. The file is closed by calling `ropen()` with no argument or with a different file name argument. `ropen()` returns 0 if the file could not be opened. If no read file is open, `fscan()` takes its input from standard input.

Read files must consist of whitespace- or newline-separated numbers in any meaningful format. An EOF will interrupt the program with an error message. The user can avoid this with a sentinel value as the last number in the file or by knowing how many times to call `fscan()`.

`getstr(strvar)`

`getstr()` reads the next line from the file that was opened by `ropen()`, and assigns it to the string variable argument. The trailing newline is part of the string.

`xred("prompt", default, min, max)`

`xred()` places a prompt on the standard error device along with the `default` value, and waits for input on standard input. If a newline is typed, `xred` returns the default value. If a number is typed, it is checked to see if it is in the range defined by `min` and `max`. If so, the input value is returned. If the typed number is not in the range, the user is prompted again for a number within the proper range.

`xopen("filename")`

The file called `filename` is read in and executed by `hoc`. This is useful for loading previously written procedures and functions that were left out of the command line during `hoc` invocation.

Editing

The `em` command invokes a public domain editor that is similar, if not identical, to MicroEMACS. Readers who wish to try this editor will find a description of it in **Appendix A2**. However, most users are already familiar with some other editor, and it is quite easy to transfer text files into `hoc` with `xopen()` or `load_file()`.

References

Kernighan, B.W. and Pike, R. Appendix 2: Hoc manual. In: *The UNIX Programming Environment*. Englewood Cliffs, NJ: Prentice-Hall, 1984, p. 329-333.

Chapter 12 Index

