7 The Scripting Language NSLS

In order to simulate a model created with NSLM it is necessary to specify the simulation interaction consisting of simulation control, model parameters, and visualization control. This can be done either by hard-coding the parameters within the model using NSLM, selecting the commands via the menu system, typing them in the Executive/Script window, or providing a script/batch file. While the NSLM language provides great expressive-ness, crucial for describing model architectures—and produces efficient code it requires the user to compile the models. To avoid re-compiling we provide the NSL script language known as NSLS which also provides a dynamic user control environment. With NSLS the user can interact very efficiently with a model during its simulation. Moreover, NSLS can be used to create a script/batch files to be executed over and over again. This is handy when the user is only interested in final results after a large number of iterations such as in the *Backpropagation* model (see chapter 3). NSLS contributes the following functionality:

- model parameter assignment
- input specification
- simulation control
- file control
- graphics control

Additionally, the NSL script interpreter interacts with the well-known scripting language TCL, the Tool Command Language (Ousterhout 1994) and Jacl, the Java Command Language extension to TCL (Scriptics 1999), thus providing NSL and TCL functionality. We refer to the combined language as NSLS and to individual commands as belonging to TCL or to NSL. An important characteristic of NSLS is that TCL and NSL commands may be combined as part of single more powerful commands, one of the advantages of having the two. In particular, NSLS commands may be applied to TCL primitive types and NSLM object types (NslFloat0, etc.) but unfortunately we cannot access the NSLM primitive types (float, etc.) from the Script Language. NSLS commands include assigning and retrieving NSLM object values as well as instantiating new NSLM objects from NSLS.

As introduced in chapter 2, when NSL is originally launched from the operating system, it brings up the NSL Executive window.[1] The top part of this window, the executive panel, contains menu buttons that let the user execute many of the NSLS control commands. The bottom part of the executive window, the script window, is where NSLS commands can be interactively typed.

Figure 7.1
The NSL Executive Window

In the rest of this chapter we describe how parameters, simulation control, and visualization control are interactively specified using the NSLS language.

7.1 Overview

We give an overview of general aspects to NSLS.

General Conventions

There are a number of general conventions we shall be using throughout this chapter:

- **Bold** letters indicate key words or commands and *italics* indicate variables or parameters to be provided by the user.
- Names for classes, modules and models start with upper case letters, e.g., **ModuleX**.
- Names for objects, module instances, model instances, and methods start with lower case letters, e.g., **moduleX**.
- Names of files storing NSLS simulation scripts should end with a ".nsl" extension.
- The script (interpreter) window provides a main prompt of "nsls%" and a secondary prompt of ">". The secondary prompt is used when a command is unfinished and continued on the next line.
- Either a carriage return or a semicolon marks the end of line in NSLS.
- A hash sign '#' precedes comments. If there is a comment at the end of a line, then a semi-colon must appear before the comment, i.e. ";#".
- To continue a command on more than one line a backslash "\" must be used at the end of every line except the last one. No characters may appear after the backslash, including empty spaces. The only exception is a statement involving an open bracket being closed on a different line.
- Spaces are very significant in NSLS (as in TCL). For example, *m*(1,1) is not the same as *m*(1, 1) (the latter has a space and generates two separate expressions instead of one).
- TCL commands do not have any particular prefix, they follow directly with the *command*,

```
nsls% command
```

- All NSL commands must have a **nsl** prefix followed by the actual *command*,

```
nsls% nsl command
```

- Variable names must begin with an alphabetic character either upper, lower case or an underscore.

- In terms of graphics, all frame and canvas names should start with a lower case letter.

Help

NSL provides extensive on-line information from the NSL web page at **http://www-hbp.usc.edu** or **http://cannes.rhon.itam.mx**. NSLS script related information is also available by typing one of the following commands in the Executive/Script window:

```
nsls% nsl help
```

or specific information on a command can be retrieved by typing:

```
nsls% nsl help command
```

Exit

To terminate the script interpreter and close all windows, type the following command in the Executive/Script window:

```
nsls% nsl exit
```

Note that the TCL **exit** command can also be used, but does not handle the termination as nicely as **nsl exit**. The **nsl exit** command calls **endModule** and **endSystem** as well as closing all files before exiting.

7.2 TCL Primitives Types

We briefly describe some of the more important commands on *primitive* types in TCL with relation to NSLS (for a complete language description please refer to Ousterhout (1994)). TCL primitive types are made primarily of numbers and characters and do not have any relation with primitive types defined in NSLM. In general, TCL is considered a *non-typed* language since variables do not involve an explicit type declaration but instead have their type implicitly specified according to initial value assignment. While there is some TCL support for *objects* types, such as with TK graphic objects, we will concentrate only in TCL basics, primarily variables, arrays, expressions, control statements and procedures.

Variables

TCL variables may be assigned values directly from TCL or indirectly from NSLM objects. TCL variables are assigned values as they are initialized using the *set* command as follows

```
set var value
```

For example, to set the value of *i* to 0 we would do

```
set i 0
```

TCL variable values can be obtained by preceding the variable name with a "$" sign, e.g. *$var*. For example, to set the value of a new variable *j* to the value of *i*:

```
set j $i
```

The values of TCL variables may then be used to set values to other TCL variables or NSLM objects attributes, the latter being of particular interest in NSLS.

Arrays

Arrays in TCL are somewhat different in both semantics and syntax from traditional programming language arrays. Arrays in TCL are actually *associative* arrays, in that elements are associated with a particular element name instead of index number. Arrays are only single sized but multiple dimensions can be simulated with names containing multiple commas. Arrays are defined using the **set** command together with parenthesis grouping elements. For example:

```
set vararray(one) 1
```

Multiple dimension arrays are simulated as follows,

```
set vararray(1,1) 30
set vararray(1,2) 99
set vararray(1,3) 7
```

where *vararray*(1,1) simulates the first row and column. Note that *vararray* is not really double sized in TCL but instead each vector element is associated with a "*x*,*y*" style name. Thus, you should be careful not to add additional letters to the element name such as spaces, since "*x*, *y*" (extra space character) would specify a different string and thus a different element name.

Expressions and Control Statements

TCL supports a number of expressions on numerical and character variables. These include numerical operators as well as commands and functions applied to both numeric and character sets. Brackets are quite particular in TCL in that they separate all kinds of expressions. In terms of control statements, TCL provides with the following (note the space between bracket sections):

- **for** control statement. For example

```
for {set x 0} {$x<10} {incr x} {
   puts "x is $x"
}
```

where curly brackets separate sections in the **for** loop, **incr** increments *x* while **puts**, which can also print to a file, prints to the screen.

- **while** control statement. For example

```
set x 0
while {$x<10} {
   puts "x is $x"
   incr x
}
```

which performs the same computation as the previous **for** loop.

- **if-else-then** control statement. For example

```
if { x > 0 } {
   set y 1
} elseif { x = 0 } {
set y 0 }
} else { set y -1 }
```

where there may be any number of **elseif** sections and both the **elseif** and **else** section are optional.

- **switch** control statement. For example

```
switch $x {
    abc {set b $y }
    hij {set b $z}
    default {set b $v}
}
```

where *x* is a string being compared to the different string options. If the string matches one of the options, then the corresponding execution statements within the curly brackets are executed.

Procedures

TCL procedures are helpful in reusing script code. For example, we define a *testHopfiledNet* procedure to run the *Hopfield* model with different input sets (the nsl set and run commands are described in section 7.3),

```
proc testHopfiledNet { input {distortion 0} } {
    puts "Testing with distortion $distorion"
    nsl set hopfieldModel.distortion $distortion
    nsl set hopfieldModel.in.pat $input
    nsl train
}
```

The procedure receives two parameters, *input* and *distortion*, the latter with default of 0. We can then generate a variable a set to simulate a double array of positive or negative ones. Note how array elements are specified in different (brackets) sections in variable *a*.

```
set a {
{ -1 -1 1 1 -1 -1 }
{ -1 1 -1 -1 1 -1 }
{ -1 1 1 1 1 -1 }
{ -1 1 1 1 1 -1 }
{ -1 1 -1 -1 1 -1 }
{ -1 1 -1 -1 1 -1 }
}
```

We call the procedure with *a* and 10 and obtain back the "puts" string.

```
testHopfieldNet $a 10
```

Testing with distortion equal 10, we call all call the procedure with only a single parameter, obtaining back the default value of 0 in this case.

```
testHopfieldNet $a
```

Testing with distortion equal 0.

Note that procedure parameters are always local to that procedure and are passed by value. If we want to pass variables by reference, we should use the TCL command **upvar**. If we want to use a non-local variable, we will have to let the procedure know the global variables by using the TCL **global** command.

System Commands

TCL offers a number of commands to interact with the external system environment. Of particular interest are the following commands,

```
cd
```

that changes the current relative directory to a new one. This is particularly important when the user wants to load script files from that directory and even more important when script files themsleves load additional files from that directory. Without doing a "cd" these latter files would not be found. Another useful command is,

```
pwd
```

checking the current working directory.

7.3 NSL Objects, Modules and Model Types

Besides TCL primitive type, NSLS can process object, module and model type variables as defined in NSLM. Sharing of variables between NSLM and NSLS is quite important since without it the user would not be able to have a good control over the model during its simulation. In general there are some limitations on how much can be accessed from NSLM in NSLS. For example, NSLM variables can be accessed from NSLS but the other way around is not possible. In the following sections we describe in more detail this sharing and how it is achieved.

Access

Accessing NSLM variables from NSLS is exclusively done by variable name and as long as the corresponding NSLM protection allows. Protections are set within NSLM via the **nslSetAccess** method where three options can be specified: "N" for no access, "R" for read access and "W" for write (and read) access, with the default being "W". (We hope to change the default access to "R" in a future version.)

```
public void initModule() {
    hv.nslSetAcccess('W');
}
```

Recall that **initModule** is the method where module variables are initialized. In chapter 2 we showed how to modify the offset, **hu**, from the scripting window. To do this, **hu**, must have **write access** which can be set with **nslSetAccess**. Since variables are not global in NSLM but local within some branch of a particular model hierarchy tree, to access a particular variable we must know its exact location within the tree, similar to referencing variables within NSLM.

Reference Tree for Model Variables

Variables are referenced using the "dot" notation similar to that in NSLM. The exception is that visibility for accessing NSLM objects is controlled *by name* and not by the variable's visibility modifier, i.e. *public*, *protected* or *private*. Referencing can be either *absolute* or *relative* as in NSLM referencing.

Absolute referencing starts always from **system** when accessing system variables or from a particular model name when accessing model variables. For example, the absolute reference to an object named *obj111* would be as follows

```
model.obj1.obj11.obj111
```

Relative referencing starts from a current location in the tree hierarchy using the path variable. For example, we could set the special variable called **varpath** that acts as a "bookmark" of where we are:

```
nsl set varpath model.obj1.obj11
```

Once this is set, we can use *relative* referencing from then on. The relative reference for *obj111* would simply be:

```
obj111
```

Expressions

NSLS provides two basic methods, **set** and **get**, to access NSLM attribute variables.

Set

Objects can be assigned data values using the **set** command, analogous to the *assignment* operator in NSLM.

To assign data,

```
nsl set object-name value
```

where *object-name* is the name of an existing object and *value* corresponds to a matching attribute type value.

For example, to set the value of "1" to a scalar object found in *model.obj1.obj11* would be use:

```
nsl set model.obj1.obj11 1
```

In general, the number of elements typed in *value* will correspond to the dimension defined for the object. The exception is to set all of the values in an *object* to a unique *value* corresponding to a single element. According to the object type and corresponding dimension NSL uses the following format:

- For **Nsl*Type*0** corresponding to single elements, for example, we set a scalar as

```
nsl set tu 1.0
```

assigning 1.0 to a zero dimension *tu* object.

- For **Nsl*Type*1** corresponding to a list or vector, for example, we set a 9-element vector as

```
nsl set s { 0 0 0 0 1 0 2 0 0 }
```

The expression assigns the nine integer values to a one dimension *s* object: 1 to *s*[4] and 2 to *s*[6], and the rest 0. (Remember that indices start with 0 and spaces must be left between brackets and other characters.) Objects can be assigned single element values by using parenthesis around element indices. For example:

```
nsl set s 0
nsl set s(4) 1
nsl set s(6) 2
```

is equivalent to the initial example.

- For **Nsl*Type*2** corresponding to a two-dimensional list or matrix, we set the values of a 2x9 matrix using:

```
nsl set s {{ 0 0 0 0 1 0 2 0 0 }
       { 3 0 0 0 0 0 0 0 0 }}
```

The expression assigns integer values to a two dimensional *s* object: 1 to *s*(0,4) 2 to *s*(0,6), 3 to *s*(0,1) and the rest 0. We can also type the above all on one line. (Notice that if an interactively specified command is incomplete then the ">" prompt will appear. When the command is complete the prompt will change back to "nsls%".)

Matrices can be assigned single element values by using parenthesis around element indices. For example

```
nsl set s 0
nsl set s(0,4) 1
nsl set s(0,6) 2
nsl set s(1,0) 3
```

is equivalent to the initial example.

- For **Nsl*Type*3** corresponding to a three-dimensional list or vector of matrices, we set the values of a 2x2x9 array using:

```
nsl set s {{{ 0 0 0 0 1 0 2 0 0 } { 3 0 0 0 0 0 0 0 0 }}
      {{ 0 0 0 0 4 0 5 0 0 } { 6 0 0 0 0 0 0 0 0 }}}
```

The expression assigns integer values to a three dimension *s* object: 1 to *s*(0,0,4) 2 to *s*(0,0,6), 3 to *s*(0,0,1), 4 to *s*(1,0,4) 5 to *s*(1,0,6), 6 to *s*(1,0,1) and the rest 0. Again we can assign single element values by using parenthesis around element indices. For example

```
nsl set s 0
nsl set s(0,0,4) 1
```

and so forth.

- For **Nsl*Type*4** corresponding to a four-dimensional list or a vector of three-dimensional matrices, for example, we set a 2x2x2x9 array as

```
nsl set s {{{{ 0 0 0 0 1 0 2 0 0 }{ 3 0 0 0 0 0 0 0 0 }}
   {{ 0 0 0 0 4 0 5 0 0 }{ 6 0 0 0 0 0 0 0 0 }}}
   {{{ 0 0 0 0 7 0 8 0 0 }{ 9 0 0 0 0 0 0 0 0 }}
   {{ 0 0 0 0 10 0 11 0 0 }{ 12 0 0 0 0 0 0 0 0 }}}}
```

The expression assigns integer values to a four dimensional *s* object: 1 to *s*(0,0,0,4) 2 to *s*(0,0,0,6), 3 to *s*(0,0,0,1) and so forth. Again we can assign single element values by using parenthesis around element indices. For example

```
nsl set s 0
nsl set s(0,0,0,4) 1
```

and so forth.

Get

The **get** command is somewhat similar to the **set** command. The main difference lies in that it retrieves the value instead of setting it. Since NSLS does not let the user create new NSLM variables, the result from a get command must be stored into a TCL variable. We

use the following TCL substitution format to retrieve values with the NSLS **get** command into a TCL variable command (notice the brackets below):

```
set tclvar [nsl get object-name]
```

where *tclvar* is the name of the TCL variable storing the resulting value, *object-name* is the name of an existing object. For example,

```
set s [nsl get model.obj1.obj11]
```

Since TCL expressions always return a string, the "[**nsl get** object-name]" will return a string as well, setting the value of *s* to the corresponding return value. For example, if "*model.obj1.obj11*" was a two dimensional 2x2 matrix containing integer values, then "*model.obj1.obj11*" might return the string "{{ 9 5 }{7 4 }}".

If the user tries the "**nsl get** *object-name*" command without assigning the returning string to a variable, a TCL script error will occur. TCL would not know how to interpret the resultant string and would print an "invalid command name" message.

According to the object type and corresponding dimension NSL uses the following format:

- For **Nsl*Type*0** corresponding to single elements, for example, we get a scalar as

```
nsl get tu
```

would return the value stored in *tu*, for example 1.0.

- For **Nsl*Type*1** corresponding to a list or vector, for example, we get a 9-element vector as

```
nsl get s
```

The expression returns a one dimension vector, for example { 0 0 0 0 1 0 2 0 0 }. Using parenthesis around element indices can retrieve single values. For example

```
nsl get s(4)
```

would return 1.

- For **Nsl*Type*2** corresponding to a two-dimensional list or matrix, for example, we get a 2x9 matrix as

```
nsl get s
```

The expression returns a two dimension matrix, for example {{ 0 0 0 0 1 0 2 0 0 }{ 3 0 0 0 0 0 0 0 0 }}. Using parenthesis around element indices can retrieve single values. For example

```
nsl get s(0,4)
```

would return 1.

- For **Nsl*Type*3** corresponding to a three-dimensional list or array of matrices, for example, we get a 2x2x9 array as

```
nsl get s
```

The expression returns a three dimension array, for example {{{ 0 0 0 0 1 0 2 0 0 }{ 3 0 0 0 0 0 0 0 0 }}{{ 0 0 0 0 4 0 5 0 0 }{ 6 0 0 0 0 0 0 0 0 }}}. Using parenthesis around element indices can retrieve single values. For example

```
nsl get s(0,0,4)
```

would return 1.

- For **Nsl*Type4*** corresponding to a four-dimensional list or two-dimensional array of matrices, for example, we get a 2x2x2x9 array as

```
nsl get s
```

The expression returns a four dimensional array, for example {{{{ 0 0 0 0 1 0 2 0 0 }{ 3 0 0 0 0 0 0 0 0 }}{{ 0 0 0 0 4 0 5 0 0 }{ 6 0 0 0 0 0 0 0 0 }}}{{{ 0 0 0 0 7 0 8 0 0 }{ 9 0 0 0 0 0 0 0 0 }}}{{{ 0 0 0 0 10 0 11 0 0 }{ 12 0 0 0 0 0 0 0 0 }}}}. Using parenthesis around element indices can retrieve single values. For example

```
nsl get s(0,0,0,4)
```

would return 1.

Another useful function is the **-dim** option. We can use **-dim** to get the sizes of the dimensions from a NSL type object:

```
nsl get s -dim
```

returns {2 2 2 9} for the four dimensional array mentioned above.

Simulation Methods

NSLS enables the user to call all NSLM simulation methods described in chapter 6 in controlling the overall simulation sequence. Starting with table 7.1 we describe simulation methods that may be called from NSLS as control commands together with additional one. These commands may involve optional parameters and most of these commands can be called from the executive window menus as well (see chapter 5), all requiring the "nsl" prefix in the script. table 7.1 shows the *connection* command called once throughout the execution of the complete system.

Connection Command	Optional Parameters	Description
makeConn	none	Execute the **makeConn** simulation method for all modules in the model.

Table 7.1 Connection command.

Table 7.2 shows the *system* commands called once throughout the execution of the complete system.

System Command	Optional Parameters	Description
initSys	none	Execute the **initSys** simulation method for all modules in the model.
endSys	none	Execute the **endSys** simulation method for all modules in the model.

Table 7.2 System commands.

Table 7.3 shows the *module* commands called once during a complete module simulation.

Module Command	Optional Parameters	Description
initModule	none	Execute the **initModule** simulation method for all modules in the model.
endModule	none	Execute the **endModule** simulation method for all modules in the model.

Table 7.3 Module commands.

Table 7.4 shows the basic **train** commands called in relation to training aspects of the simulation.

Train Command	Optional Parameters	Description
initTrainEpochs	none	Execute the **initTrainEpochs** simulation method for all modules in the model.
endTrainEpochs	none	Execute the **endTrainEpochs** simulation method for all modules in the model.
initTrain	none	Execute the **initTrain** simulation method for all modules in the model.
simTrain	*trainEndTime*	Execute the **simTrain** simulation method for all modules in the model. Simulation starts at *t*=0 until *trainEndTime* (a real number) or until *system.trainEndTime* is reached. The actual number of steps is specified by *trainEndTime* divided by *trainDelta*.
endTrain	none	Execute the **endTrain** simulation method for all modules in the model.

Table 7.4 Basic train commands.

Table 7.5 shows additional **train** commands called in relation to training aspects of the simulation.

Train Command	Optional Parameters	Description
train	*trainEndTime*	Execute **initTrain** once, followed by **simTrain** starting at *t*=0 until reaching *trainEndTime* or *system.trainEndTime* followed by **endTrain** at the end. Simulation takes place for all specified epochs.
doTrainEpochTimes	*numTrainEpochs*	Execute the previous **train** command for *numTrainEpochs* times.
breakEpochs	none	Stop the simulation in between two epochs.
stepEpochs	*numTrainEpochs*	According to the current state of the simulation, execute either the train phase or run phase for all modules in the model *numTrainEpochs*, or once if not specified. If a **breakEpochs** was previously called then start from the next epoch.
contEpochs	*lastTrainEpoch*	According to the current state of the simulation, execute either the train phase or run phase for all modules in the model until *lastTrainEpoch* or until all epochs have been processed. If a **breakTrainEpochs** was previously called then start from the next epoch.
breakCycles	none	Stop the simulation in between two cycles.

Table 7.5 Additional train commands.

Table 7.5 *(continued)*		
Train Command	**Optional Parameters**	**Description**
contCycles	*trainEndTime*	Execute **simTrain** method starting at *t=trainTime* (current training time) until reaching *trainEndTime* (a real number) or *system.trainEndTime* if not specified.
stepCycles	*numTrainCycles*	Execute the **simTrain** method for *numTrainCycles* (an integer). If *numTrainCycles* is not specified, it steps one cycle only.
breakModules	none	Stop the simulation in between modules.
stepModules	*numTrainModules*	According to the current state of the simulation, execute the **simTrain** method for all modules in a model *numTrainModule* times, or once if not specified. If a **breakModule** was previously called then start from the next module.
contModules	*lastTrainModule*	According to the current state of the simulation, execute the **simTrain** for all modules in the model until *numTrainModule*, or until all modules have been processed. If a **breakModule** was previously called then start from the next module.

Table 7.5
Additional train commands.

Table 7.6 shows the basic **run** commands called in relation to running aspects of the simulation.

Run Command	Optional Parameters	Description
initRunEpochs	none	Execute the **initRunEpochs** simulation method for all modules in the model.
endRunEpochs	none	Execute the **endRunEpochs** simulation method for all modules in the model.
initRun	none	Execute the **initRun**simulation method for all modules in the model.
simRun	*runEndTime*	Execute the **simRun** simulation method for all modules in the model. Simulation starts at *t*=0 until *runEndTime* (a real number) or until *system.runEndTime* is reached. The actual number of steps is specified by *runEndTime* divided by *runDelta*.
endRun	none	Execute the **endRun** simulation method for all modules in the model.

Table 7.6
Basic run commands.

Table 7.7 describes additional **run** commands called in relation to running aspects of the simulation.

Command	Optional Parameters	Description
run	*runEndTime*	Execute **initRun** once, followed by **simRun** starting at *t*=0 until reaching *runEndTime* or *system.runEndTime* followed by **endRun** at the end. Simulation takes place for all specified epochs.
doRunEpochTimes	*numRunEpochs*	Execute the previous **run** command for *numRunEpochs* times.
breakEpochs	none	Stop the simulation in between two epochs for all modules in the model.
stepEpochs	*numRunEpochs*	According to the current state of the simulation, execute either the train phase or run phase for all modules in the model *numRunEpochs*, or once if not specified. If a **breakEpochs** was previously called then start from the next epoch.
contEpochs	*lastRunEpoch*	According to the current state of the simulation, execute either the train phase or run phase for all modules in the model until *lastRunEpoch* or until all epochs have been processed. If a **breakEpochs** was previously called then start from the next epoch.
breakCycles	none	Stop the simulation in between two cycles.
contCycles	*runEndTime*	Execute **simRun** method starting at *t=runTime* (current run time) until reaching *runEndTime* (a real number) or *system.runEndTime* if not specified.
stepCycles	*numRunCycles*	Execute the **simRun** method for *numRunCycles* (an integer). If *numRunCycles* is not specified, it steps one cycle only.
breakModules	none	Stop the simulation in between modules.
stepModules	*numRunModules*	According to the current state of the simulation, execute the **simRun** method for all modules in a model *numRunModule* times, or once if not specified. If a **breakModules** was previously called, then start from the next module.
contModules	*lastRunModule*	According to the current state of the simulation, execute the **simRun** for all modules in the model until *numRunModule*, or until all modules have been processed. If a **breakModules** was previously called, then start from the next module.

Table 7.7
Additional NSL run commands

Simulation Parameters

There are a number of simulation parameters that can be specified affecting the overall simulation. These values can be overriden by paremeters passed to the simulation methods as described in the previous section. These parameters are applied to all modules at once when setting them at the system level as follows,

```
nsl set system.parameter value
```

where *parameter* is the corresponding system parameter.

These attributes may also be set by module by specifying the following

```
nsl set module.parameter value
```

These parameters will be described in terms of "train", "run" and "integration" parameters.

Train

The system train parameters are described in table 7.8.

Parameter	Default Value	Description
trainDelta	1.0	Training delta (step size) for the entire system.
trainEndTime	1.0	Training end time for the entire system.
numTrainEpochs	1	Training epochs for the entire system.

Table 7.8
System Train Parameters

For example, to set the value to 5.0 of **trainEndTime** for all modules in the system do the following:

```
nsl set system.trainEndTime 5.0
```

Run

The system run parameters are described in table 7.9.

Parameter	Default Value	Description
runDelta	1.0	Run delta (step size) of the entire system.
runEndTime	1.0	Run end time of the entire system.
numRunEpochs	1	Number of runs (analogous to epochs) for the entire system

Table 7.9
System Run Parameters

For example, to set the value to 5.0 of **runEndTime** for all modules in the system do the following:

```
nsl set system.runEndTime 5.0
```

Integration Approximation Methods

As discussed in chapter 6, NSL provides numerical methods for integration. The involved parameters may be set at the system level or per module. The parameters are described in the following statements, and they are set as follows,

```
nsl set system.approximation.parameter value
```

or for a particular module,

```
nsl set module.approximation.parameter value
```

where *parameter* represents the corresponding integration parameter as shown in table 7.10.

Table 7.10
System Approximation Parameters

Parameter	Default Value	Description
method	Euler	NSL offers the following two numerical method options: **Euler** or **RungeKutta2**.
delta	1.0	The user specifies the approximation step or delta for the complete system.

7.4 Input Output

There are a number of input and output commands dealing with script and data files.

Script Files

Script files store NSLS style command files. These files may be loaded to avoid writing single commands at a time. Additionally, the user may store a complete window interaction (a "log") to be loaded at a alter time without having to duplicate it again. In general, multiple script files can be associated with a single model.

Source

Script files are loaded into the simulator with following command:

```
nsl source file-name
```

(File names must be either relative to the current directory or require an absolute path to the desire file. Additionally, NSL uses the file "SCS_LIBRARY_PATHS" located in at the user's home directory to also search for these files.

Data Files

Besides script files, NSLS also supports reading and writing data as "open format" **ascii text**—text that would need to be read or written in a specific format for the particular model—from/to files or the screen ("standard output"). In particular, it is quite useful to read and store data generated by the simulation into files. Data stored in files can be used as input to new simulations, such as when saving training weights as with *Backpropagation*, or simply as a means of analyzing the simulation output later on in numerical detail. Note that for simplicity these files may actually follow the NSLS script format although NSL gives the user this "open format" added flexibility. Note also that script files are usually loaded (read) all at once while data files are read line by line since only the user knows its particular format. For this reason NSL provides with a number of commands to manipulate files.

Open

Opens a file using a particular access type: read (r), write (w) and append (a) having as default read.

```
open file-name file-access
```

If the file is successfully opened, the command will return a file descriptor that can be saved into a TCL variable using for example

```
set f [open input.dat r]
```

Gets

The *gets* command retrieves the next line from the file associated with the file descriptor passed as an argument. If the file has reached its end, it returns the empty string. For example

```
nsl set hopfieldModel.in.pat [gets $f]
```

Puts

The *puts* command passes a string as argument to the file associated with the file descriptor. It adds a new line character at the end of the string, for example

```
puts $f [nsl get hopfieldModel.hopfield.weights]
```

Eof

The *eof* command tells you if the file associated with the file descriptor has reached its end. If this is true, it returns 1, otherwise 0.

```
while {![eof $f]} {
    nsl set hopfieldModel.in.pat [gets $f]
    nsl run
}
```

Close

The *close* command closes the file associated with the file descriptor passed as an argument.

```
close $f
```

Monitor

The *monitor* command is similar to the "puts" command; however, it continuously "puts" the value of the variable being monitored into the file or screen until specified otherwise and returns a monitor descriptor. To enable an object specified by name to be written into a file do:

```
nsl monitor object-name –file file-descriptor
```

If *file-descriptor* is not specified, then "standard output" is taken as the output file name, which sends the data to the script window, (or where the standard output was redirected):

```
nsl monitor object-name
```

Additional parameters may be included in the monitor command.

```
nsl monitor -parameter value
```

These parameters are given in table 7.11.

parameter	default	description
start	Current time	Start time in the user's specified units of time (usec, msec, seconds, etc).
stop	End time	Stop time in the user's specified units of time.
freq	Delta	Frequency is the number of cycles until the next reporting period. One means report every cycle, two means report every other cycle, etc.

Table 7.11
Parameters for the Monitor Command—start parameter, stop parameter, freq parameter.

All visible NSL objects in a module may be enabled for monitoring by using an asterisk, "*", for example

```
nsl monitor model.module.*
```

In the case of NSL types of 0 dimension, the object data will be written in a single line. In the case of NSL array types, the object data will be written in row major format.

To "unmonitor" a particular variable we can type:

```
nsl unmonitor object-name
```

7.5 Graphics Displays

Another important functionality of NSLS is to support interactive graphical display generation. This is achieved both using the script window and script files. In chapter 5 we discussed how to build the window interface and graphical displays using the NSLM language and how to interact with them from the menu interface. In this chapter we will discuss how to interact using the NSLS language.

Reference Tree for Canvases

In NSLS as in TCL, all windows spring from one parent window—the root window—denoted by ".nsl". In addition to **varpath** (section 7.3), NSLS also provide the **displaypath** variable that acts just like **varpath** but is used to reduce the amount of typing when specifying a display path, for example

```
nsl set displaypath .nsl.frame1.canvas12
```

Create and Configure

NSLS uses a general format in creating new windows and configuring already created ones. Window properties or attributes can be set during their creation or modified afterwards.

To create a new window where initial attributes are specified using the "*-attribute value*" format,

```
nsl create window window-name -attribute value
```

To configure an already created window with attributes specified by the "*-attribute value*" format,

```
nsl configure window window-name -attribute value
```

Note many any attributes may be changed in a single command using multiple "*-attribute value*" pairs in the same line.

NslExecutiveWindow

The first window in the graphics interface is always the **NSL Executive/Script** instantiated by the system and window shown in figure 7.1. This is the root window or console in the NSL window hierarchy and denoted by ".nsl". Each additional window/frame added to the screen should append its name to this executive window name. Since the **executive** window is already instantiated, we can only modify its attribute values shown in table 7.12.

parameter	type	default	description
width	int	100	width in pixels
height	int	100	height in pixels
x0	int	0	left position in x in pixels
y0	int	0	top position in y in pixels

Table 7.12 Executive Window Parameters—width parameter, height parameter, x0 parameter, y0 parameter.

For example, we could change the size of the **NslExecutiveWindow** window by specifying the following

```
nsl configure .nsl -width 400 -height 200
```

NslOutFrame

The user may instantiate multiple **NslOutFrames** representing independent windows on the screen, analogous to the **NslExecutiveWindow**. **NslOutFrames** are used to display **NslOutCanvas** (to be described in the next section) holding actual graphical output. To create a new **NslOutFrame** we can type

```
nsl create NslOutFrame .nsl.frame-name
```

where *frame-name* is used to reference the newly instantiated frame object. This name can then be used for further configuration. The **NslOutFrame** attribute list is shown in table 7.13

parameter	type	default	description
display	charString		frame name
title	charString	display name	any string name to appear in the frame label
rows	int	1	the number of rows
column	int	1	the number of columns
x0	int	0	position in x direction in pixels
y0	int	0	position in y direction in pixels
width	int	100	frame width in pixels
height	int	100	frame height in pixels
font	charString		font used
background	charString		background color
foreground	charString		foreground color
freq	int	1	graphics update frequency in relation to simulation step. Default is 1 corresponding to simulation step

Table 7.13
NslOutFrame Attributes.

Note that NSL can display the output data with frequencies different to those used by the simulator in performing the actual variable updates. Since displaying data may become very slow, modifying this frequency can significantly speed up the overall time or "wall clock time" of the simulation. The only restriction on frequency is that all the variables within an output frame must have the same output display frequency.

To create an output frame named *diddayOut* with width 100 and height 200, we would type:

```
nsl create NslOutFrame .nsl.diddayOut -width 100 -height 200
```

For example, if we later want to change the foreground color to white we would type:

```
nsl configure .nsl.diddayOut –foreground white
```

NslOutCanvas

NSL can instantiate multiple **NslOutCanvas** inside a single independent **NslOutFrame** window. Canvases are not independent windows on the screen, but are always part of a **NslOutFrame**. To instantiate a new **NslOutCanvas** the user has to specify besides a *canvas-name* a variable name "**–var** *var-name*" specifying the particular variable being

output in addition to attribute values. For example, the following line instantiates a new canvas in an existing frame,

```
nsl create NslOutCanvas .nsl.frame-name.canvas-name \
    –var var-name –attributes value
```

Note that the variable name is a required parameter. The **NslOutCanvas** parameter list is given in table 7.14.

parameter	type	default	description
display	charString		frame name
title	charString	display name	canvas label
var	Object type		NSL object to be display in the canvas. Required.
graph	charString		graph type—see table 5.2.
position	charString		position in output frame: first, next, previous, and last.
wymin	float double		low variable value in y direction
wymax	float double		high variable value in y direction
wxmin	float double		low value in the x direction—for temporal plots this is time zero.
wxmax	float double		high value in the x direction—for temporal plots this is the max time.
freq	int		the frequency or time step used for collecting data from the simulation thread
drawcolor	charString		draw color
drawstyle	charString		draw style
xlabel	charString		label placed along x axis
ylabel	charString		label placed along y axis
option	charString		re-scale or shift
grid	boolean	true	grid is drawn

Table 7.14
NslOutCanvas Parameters.

As a general example, to create a display canvas *s* inside an output frame named *maxSelector* with an area level graph displaying the values between -1 and 2 for layer variable *"s"* we would type

```
nsl create NslOutCanvas .nsl.maxSelector.s -var didday.s \
    -wymin -1 -wymax 2 -graph Area
```

When can change for example the **NslOutCanvas** minimum and maximum values as follows:

```
nsl configure .nsl.maxSelector.s -wymin -10 -wymax 20
```

Note that the order in specifying parameters is irrelevant.

NslInFrame

The user may instantiate multiple **NslInFrames**, similar to the **NslOutFrames**. **NslInFrames** are used to display **NslInCanvases** where the user may interact with the simulation by providing input or by changing values as the simulation is running. To isntantiate a new **NslInFrame** we type

```
nsl create NslInFrame .nsl.frame-name
```

where *frame-name* is used for referencing the newly created frame. The NslInFrame attribute list is shown in table 7.15

parameter	type	default	description
display	charString		frame name
title	charString	display name	any string name to appear in the frame label
rows	int	1	the number of rows
column	int	1	the number of columns
x0	int	0	position in x direction in pixels
y0	int	0	position in y direction in pixels
width	int	100	frame width in pixels
height	int	100	frame height in pixels
font	charString		font used
background	charString		background color
foreground	charString		foreground color
freq	charString	1	graphics update frequency in relation to simulation step. Default is 1 corresponding to simulation step

Table 7.15
NslInFrame Attributes.

To create an input frame named *diddayIn* with width 100 and height 200, we would type:

```
nsl create NslInFrame .nsl.diddayIn -width 100 -height 200
```

For example, to change the foreground color to white we would do

```
nsl configure .nsl.diddayIn –foreground white
```

NslInCanvas

NSL can instantiate multiple **NslInCanvases** inside a **NslInFrame** in order to generate graphical input from the simulation. Similar to **NslOutCanvases**, **NslInCanvases** are not independent windows on the screen, but are always part of a **NslInFrame**. To instantiate a new **NslInCanvas** the user has to specify besides a *canvas-name* a variable name "**–var** *var-name*" specifying the particular variable being used for input in addition to attribute values.

```
nsl create NslInCanvas .nsl.frame-name.canvas-name \
    –var var-name –attributes value
```

Note that the variable name is a required parameter. The parameter list is shown in table 7.16

parameter	type	default	description
display	charString		frame name
title	charString	display name	canvas label
var	Object Type		NSL object to be display in the canvas. Required.
graph	charString		graph type—see specified list below
position	charString		position in output frame: first, next, previous, and last.
wymin	float double		low variable value in y direction
wymax	float double		high variable value in y direction
wxmin	float double		low value in the x direction—for temporal plots this is time zero.
wxmax	float double		high value in the x direction—for temporal plots this is the max time.
freq	float double		the frequency or time step used for collecting data from the simulation thread
drawcolor	charString		draw color
drawstyle	charString		draw style
xlabel	charString		label placed along x axis
ylabel	charString		label placed along y axis
option	charString		rescale or shift
grid	boolean	true	grid is drawn

Table 7.16
NslInCavas Attributes.

Input graph types may be specified with one of the following strings: **InputImage** and **NumericEditor** as described in chapter 5.

As a general example, to create a display canvas *s* inside an output frame named *maxSelector* with an inputImage graph displaying the values between 0 and 1 for layer variable "*s*" we would type

```
nsl create NslInCanvas .nsl.didday.s -var maxSelector.s \
    -wymin 0 -wymax 1 –graph inputImage
```

When can change for example the **NslInCanvas** minimum and maximum values as follows:

```
nsl configure .nsl.maxSelector.s -wymin -10 -wymax 20
```

Print

The user can print graphical windows to a file using the PostScript format. The command to do this is:

```
nsl print -name displayname -filename -filename
```

Note that the name parameter can be either a whole frame or a single canvas. The parameters for the **print** command are given in table 7.17.

Parameter	type	default	description
name	charString		Display name: either “screen”, “.ex”, fully extended frame name, or fully extended canvas name
size	int, int	8.5” by 11”	width and height in pixels
position	int, int	centered	x and y position in pixels
orientation	charString	portrait	landscape or portrait

Table 7.17
Print Command Parameters.

7.6 Summary

The NSLS scripting language is a very powerful language whereas we have only described its basics. We showed how to use TCL commands to manipulate values of NSLM variables and how to provide control structure to scripts (*if*, *while*, *for*, and *switch*). We also saw how to get data from NSLM variables and store the information in NSLS variables. A very popular use of the NSLS language is in controlling the simulation with commands such as “nsl trainAndRunAll”, “nsl stepTrain”, “nslBreakCycle”, etc. Finally, we documented how to create new **NslOutFrames** and **NslInFrames**, as well as how to add **NslOutCanvases** and **NslInCanvases** to them.

Notes

1. Note that NSL can be executed in **noDisplay** mode in which case no executive window is brought up.