Appendix III – NSLC Extensions

The NSL C++ (NSLC) version includes a number of extensions not included at the moment in NSLM, the common language to both C++ and Java versions. We expect that these extensions will be incorporated into NSLM in the future.

A.III.1 Object Type Extensions

NSLC adds a number of extensions to NSLM object types. Among these the most important ones are the addition of object type arrays, new object types and a number of extensions on module connectivity.

Arrays

NSLC adds a *dimSpec* array specification to any object type definition as follows:

```
    VisiblitySpec ObjectType varName(paramList)dimSpec;
```

For example a single dimension 10 element private array of *ObjX* named *x* can be defined as follows:

```
    private ObjX x()[10];
```

where no instantiation parameters are provided in this example. Additional dimensions are provided by simply adding new brackets with their corresponding element number specification. An extended example of array usage is shown in the "Face Recognition by Dynamic Link Matching" model in chapter 18.

Defined Types

NSLC adds additional defined types besides those described in chapter 6.

String

NSL defines two additional **charString** object types as shown in table A.III.1.

Table A.III.1
Additional charString object types defined in NSLC.

Dimension Type	0	1	2	3	4
charString		NslString1	NslString2		

Ports

NSLC defines additional **charString** port object types as shown in table A.III.2.

Dimension Type	0	1	2	3	4
charString		NslDoutString1	NslDoutString2		
		NslDinString1	NslDinString2		

Table A.III.2
Additional charString port object types defined in NSLC.

Convolution

NSLC adds two additional convolutions methods as shown in table A.III.3 defined for both vectors and matrices of two dimensions. The type and dimension of *z* corresponds to that of *y*.

Method Expression	Description
z = **nslConvW**(*x*,*y*)	wrap-edge convolution
z = **nslConvC**(*x*,*y*)	copy-edge convolution

Table A.III.3
Multiplication method.

Connect

NSLC provides additional **nslConnect** statements enabling *fan-out* and *fan-in* connections between multiple ports at once (NSLM as described in chapter 6 permits single port interconnections). *Fan-out* enables the output of a particular port to be sent out to a number of input ports at the same time using the following format,

```
   nslConnect (port-out, port-in-list);
```

where *port-out* specifies an output port and *port-in-list* specifies a list of input ports separated by commas. Each input port is connected to the same output port. Analogous, *fan-in* enables the output of a list of port to be sent out to a particular input port using the following format,

```
   nslConnect (port-out-list, port-in);
```

where *port-out-list* specifies a list of output ports separated by commas and *port-in* specifies a particular input port. Each output port is connected to the same input port. Note that in this case the input port would queue data from the different output ports according to the order in which they are received.

More generally, a list of output ports may be connected to a list of input ports using the following format,

```
   nslConnect (port-out-list, port-in-list);
```

where *port-out-list* specifies a list of output ports and *port-in* specifies a list of input ports port both separated by commas.

Disconnect

NSLC provides an additional construct, **nslDisconnect**, to delete existing connections. The basic format is as follows,

```
   nslDisconnect (port-out, port-in);
```

where *port-out* specifies an output port and *port-in* specifies an input port.

Similarly to connections, NSLC provides *fan-out*, *fan-in* and the more general disconnection formats as follows,

```
   nslDisconnect (port-out, port-in-list);
   nslDisconnect (port-out-list, port-in);
   nslDisconnect (port-out-list, port-in-list);
```

Relabel

NSLC provides additional **nslRelabel** statements enabling *fan-out* and *fan-in* relabels between multiple ports at once (NSLM as described in chapter 6 permits single port relabels). *Fan-out* enables either a particular output or input port to be relabeled to a number of output or input ports at the same time, respectively. We use either of the following formats,

```
nslRelabel (port-out, port-out-list);
nslRelabel (port-in, port-in-list);
```

Analogous, *fan-in* enables either a list of output or input ports to be relabeled to a particular output or input port at the same time, respectively. We use either of the following formats,

```
nslRelabel (port-out-list, port-out);
nslRelabel (port-in-list, port-in);
```

More generally, a list of either output or input ports may be relabeled to a list of output or input ports, respectively, using the following formats,

```
nslRelabel (port-out1-list, port-out2-list);
nslRelabel (port-in1-list, port-in2-list);
```

Delabel

Analogous to **nslDisconnect** NSLC supports a delabeling (deleting a relabel) construct **nslDelabel**. The basic formats are as follows,

```
nslDelabel (port-out1, port-out2);
nslDelabel (port-in1, port-in2);
```

where *port-out1* and *port-out2* specify output port and *port-in* specifies an input port.

Similarly to disconnections, NSLC provides *fan-out*, *fan-in* and the more general delabel formats as follows,

```
nslDelabel (port-out, port-out-list);
nslDelabel (port-in, port-in-list);
nslDelabel (port-out-list, port-out);
nslDelabel (port-in-list, port-in);
nslDelabel (port-out1-list, port-out2-list);
nslDelabel (port-in1-list, port-in2-list);
```

File Manipulation

As described in Appendix I NSL supports reading and writing into external text files. NSLC additionally supports reading and writing into binary files as shown in chapter 18 with the “Face Recognition with Dynamic Link Architecture” model.

NSLC uses the same basic file manipulation methods described in Appendix I with an additional optional second argument in the **open** method describing the type of file (*file-type*) being manipulated, **text** or **binary**, as shown next:

```
file.open(interaction-spec,file-type);
```

As previously discussed in Appendix I *interaction-type* corresponds to any of the following: ‘**R**’ for read only, ‘**A**’ (all) for both read and write or ‘**W**’ for write only. Note that binary files do not separate values with spaces thus the user must read each byte or character at a time such as in the model described in chapter 18. Since NSLC is based on C++ the user may take advantage of **char** and **unsigned char** types when reading binary files.

A.III.2 Script Extensions

NSLC adds the following script extensions.

Logs

Log files contain the history of previous user model interaction. This is quite useful in generating a previous interaction that has not been stored. Scripts can be logged and saved automatically at the end of the simulation (however, the default is logging false).

```
   nsl set system.log true
```

There is one default log for the complete simulation. The log file name corresponds to the model name followed by a dot and a numeric suffix corresponding to the log version followed by a "log" and it may be specified with a different name by the user. For example,

```
   nsl set system.logfile maxSelectorModel.1.log
```

Besides being able to review the log file, it is possible to reload it and execute it as any other NSLS script.

A.III.3 Input Facility

NSLC includes predefined object classes for the generation of temporal visual stimuli. These types are usually instantiated inside a special visual input module such as the **Visin** module used in the "Retina" model (chapter 10) and the **World** module used in the "Learning to Detour" model (chapter 17). Using these object types different stimuli may be set, with constrains on their location and time when they should appear and disappear. In the following sections we explain these in more detail.

Object Types

Input object types extend their basic semantics from NSLM numeric types while adding special functionality for processing visual stimuli. These types vary according to their dimension and types as shown in table A.III.4.

Dimension Type	0	1	2	3
float	NslInputFloat0	NslInputFloat1	NslInputFloat2	NslInputFloat3
double	NslInputDouble0	NslInputDouble1	NslInputDouble2	NslInputDouble3
int	NslInputInt0	NslInputInt1	NslInputInt2	NslInputInt3

Table A.III.4
Input layer object types defined in NSLC.

Since the input layer object types are derived from the regular numeric layer types have the same instantiation parameters as regular layers. The only exception is the 3-dimensional input array taking four instead of three instantiation arguments. This difference corresponds to the fact that a 3-dimensional input layer is actually a combination of two 2-dimensional input layers corresponding to the *xy* and *xz* space views (see the "Learning to Detour" model in chapter 17 as an example of its usage). Thus input layers may be added with regular layers, and so on. On the other hand the input layer is able to map visual stimulus objects onto the layer. For example, figure A.III.1 shows an **AreaLevel** graph view of a **NslInputFloat2** input layer made of 40x40 elements, containing an object of size 8x4. This example is taken from the **Visin** module in the **Retina** model in chapter 10.

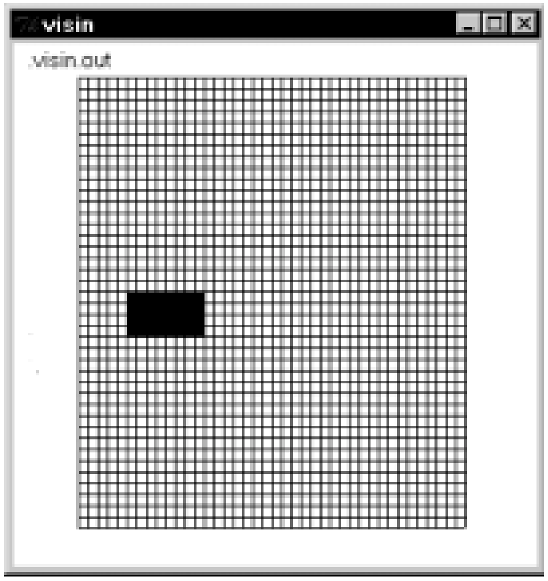

Figure A.III.1
A **NslInputFloat2** input layer of 40x40 elements containing a 8x4 stimulus.

Figure A.III.2 shows a **Temporal** graph view of a **NslInputFloat0** input layer, containing an stimulus appearing at two different time intervals.

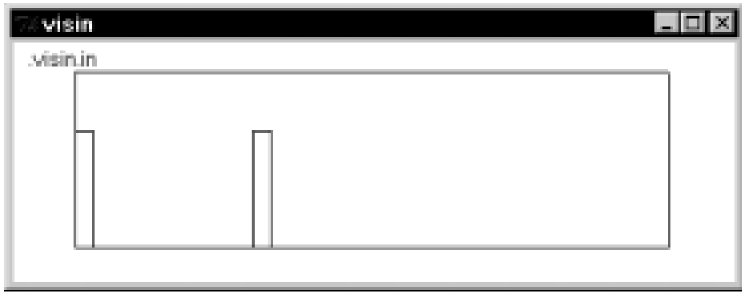

Figure A.III.2
A **NslInputFloat0** with a time varying stimulus.

Input Processing
Actual input layer processing involves "running" the stimuli specified for the particular layer. We show how to interactively specify stimuli in the next section. Input layer processing is achieved by including the following statement inside a module,

```
   input_layer = 0;
   input_layer.run();
```

where *input_layer* specifies the name of the layer, and *run* is the method processing any existing stimuli specification. For example, in the "Retina" model the visual input *in* is processed in the *Visin* module as follows,

```
   in = 0;
   in.run();
```

Note that the input layer is first reset to "0". This is optional since in some case the user may want to leave a trail or history of previous stimuli locations as in the "Learning to Detour" model in chapter 17.

Input Specification
In the current NSLC version all input and stimuli specification takes place interactively using the NSLS script interpreter. Before being able to specify any stimuli one must

understand the coordinate system used in the input layer and stimuli, shown in figure A.III.3.

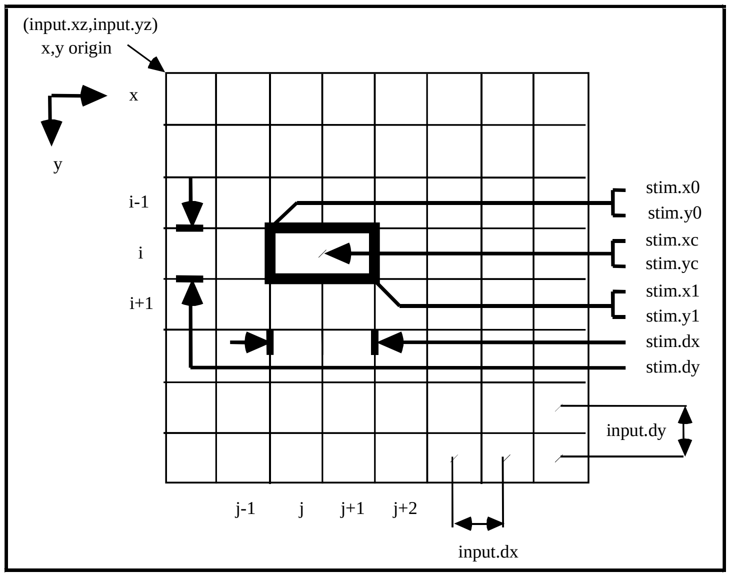

Figure A.III.3
Input layer and stimulus specification details.

There are two aspects to input specification. First the coordinate system in the input layer must be specified. This involves specifying the origin of the coordinate system, (*input.xz*,*input.yz*) and the distance among adjacent elements in the input layer, (*input.dx,input.dy*) as shown in figure A.III.3.

These parameters are specified as follows where *input* in this figure represents the *input_layer* name,

```
   nsl set input_layer.par-name par-value
```

where the different alternatives for *par-name* with their corresponding *par-value* types and descriptions are given in table A.III.5. Note that the input library supports up to 3-dimensional specifications.

Parameter	Type	Description
xz	int	coordinate system *x*-axis origin element
yz	int	coordinate system *y*-axis origin element
zz	int	coordinate system *z*-axis origin element
dx	numeric	distance among adjacent elements in the *x*-axis
dy	numeric	distance among adjacent elements in the *y*-axis
dz	numeric	distance among adjacent elements in the *z*-axis

Table A.III.5
Input layer parameter options.

For example, in the “Retina” model the input layer coordinate system is specified as follows,

```
   nsl set retinaModel.retina.visin.input.xz 0
   nsl set retinaModel.retina.visin.input.yz 20
   nsl set retinaModel.retina.visin.input.dx 2.0
   nsl set retinaModel.retina.visin.input.dy 2.0
```

Once the input layer coordinate system has been specified it is necessary to add stimuli specifications. The general stimulus specification format is as follows,

```
nsl create stim-type stim-name -layer input-name -val val \
   [-x0 x0 -y0 y0 -z0 z0] [-xc xc -yc yc -zc zc] \
-dx dx -dy dy -dz dz -vx vx -vy vy -vz vz -spec_type spec-type
```

These parameters are those shown in figure A.III.3 and specified in more detail in table A.III.6. In the current NSLC version *stim-type* can only be set as **BlockStim**, while *stime-name* and *input-name* are the names of the stimulus and input layer, respectively.

Table A.III.6
Stimulus parameter options.

Parameter	Type	Description
val	numeric	value taken for the complete stimulus
spec_type	string	specification format: **center** [xc,yc] or **corner** [x0,y0]
x0	numeric	stimulus upper left corner *x*-coordinate
y0	numeric	stimulus upper left corner *y*-coordinate
z0	numeric	stimulus upper left corner *z*-coordinate
xc	numeric	stimulus center *x*-coordinate
yc	numeric	stimulus center *y*-coordinate
zc	numeric	stimulus center *z*-coordinate
dx	numeric	stimulus width in *x*-direction
dy	numeric	stimulus depth in *y*-direction
dz	numeric	stimulus height in *z*-direction
vx	numeric	stimulus speed in *x*-direction
vy	numeric	stimulus speed in *y*-direction
vz	numeric	stimulus speed in *z*-direction

The location of the stimulus may be specified either by setting *spec_type* to either corner or center and specifying [*x0,y0,z0*] or [*xc,yc,zc*], respectively. For example, the stimulus shown in figure A.III.1 was specified with the following script,

```
nsl create BlockStim stim -layer retinaModel.retina.visin.in –
val 1.0 \
   -spec_type center -xc 2.0 -yc 0.0 -dx 4.0 -dy 4.0 -vx 7.6
```

Note that the actual figure shows the stimulus situated in a new location according to its initial position, current speed and simulation time elapsed.

Additionally, NSL lets the user define time intervals when a stimulus should appear using the following format,

```
nsl create TimeInterval -stim stim-name -t0 t0 -t1 t1
```

Table A.III.7 describes the two parameters in more detail.

Parameter	Type	Description
t0	numeric	interval starting time
tz	numeric	Interval ending time

Table A.III.7
Time interval parameter options.

For example, the stimulus shown in figure A.III.2 was created using the following script:

```
   nsl create BlockStim stim1 -layer tectum11Model.tectum.in
   nsl create TimeInterval -stim stim1 -t0 0.0 -t1 0.3
   nsl create TimeInterval -stim stim1 -t0 3.0 -t1 3.3
```

This creates a time interval between 0.0 and 0.3 and a second one for the same stimulus between 3.0 and 3.3. Notice that in this example the input layer was actually a scalar thus no other stimulus parameters were given, including stimulus size or location).

A.III.4 Distribution

One additional extension to the NSLC system currently in development is the distributed execution environment to make processing more efficient. See the NSLC web site (*http://www.cannes.itam.mx/*) for the latest developments.