NSL

Neural Simulation Language[1]

Version 2.1

Alfredo Weitzenfeld

Technical Report 91-05

August 1991

Brain Simulation Laboratory
Center for Neural Engineering
University of Southern California

[1] Preparation of this document was supported in part by grant 1R01 NS24926 from NINDS of the National Institutes of Health (Michael A. Arbib, Principal Investigator), and in part by a grant from Rockwell International to the USC Center for Neural Engineering.

NSL - Neural Simulation Language
Version 2.1

The information in this document is subject to change. The Brain Simulation Laboratory assumes no responsibility for any errors that may appear in this document. Users are requested to advise Alfredo Weitzenfeld at the address given below of any comments or suggestions for improvements.

To get copies of the software contact :

Alfredo Weitzenfeld
Brain Simulation Laboratory
Center for Neural Engineering
University of Southern California
Los Angeles, CA 90089-2520
e-mail : alfredo@usc.edu

The lab will charge a fee of $50 for tape, user's manual and delivery.
(For FTP, send a request to the above e-mail address.)

Table of Contents

1. Introduction

NSL (pronounced "Nissl"), Neural network Simulation Language, is a powerful yet easy to use simulation language and development system. The simulator is designed and implemented following an object-oriented paradigm; and includes NSL's high level language for describing neural networks, an interactive command interpreter, and powerful visualization tools for analyzing models in different ways.

The goal of NSL is to facilitate the description of models in relatively few 'human readable' lines which be understood and appreciated both by users with little programming background and by those with more extensive programming expertise. For this purpose we have designed the language in the form of basic mathematical equations where the user may either program at a high level, or include more sophisticated programming code.

Although NSL has been principally developed with a certain type of biological modelling in mind, we have kept it as general as possible, letting the user define diverse types of networks, including those concerned with learning. As will be discussed in more detail in later sections, any network that can be described as a set of equations to be updated after each simulation iteration can be modelled by NSL.

The system includes the following features :

* An object-oriented simulation language with built-in model object classes.
* A library with the common neural network functions.
* A library with sample neural network models.
* Methodology for extending the libraries.
* A command language interpreter for describing the simulation environment.
* Interactive and batch processing of commands and data.
* Management of simulation versions.
* An interactive X windows graphical interface.
* Temporal and spatial displays.
* 2D and 3D graphics.
* The flexibility of a system written in the UNIX/C/C++ environment.

NSL 2.1, offered as public domain software, is the second release for the second generation NSL language. NSL 2.0 [Weitzenfeld 1990] has been widely used in varied simulation applications giving rise to an accompanying document with sample neural network models simulated in NSL. These second generation versions are direct successors to the original NSL 1.0 [Weitzenfeld 1989] which has also been extensively used in our research lab, and by students taking brain theory course given at USC.

2. NSL System Overview

This section gives an overview of NSL in terms of the requirements for its use, and an overview of the overall system design.

2.1. System Requirements

NSL 2.1, like NSL 2.0, is implemented in C++ [Stroustrup 1987], while NSL 1.0 was implemented in C [Kernighan and Ritchie 1978][1].

The current system requires:

- an AT&T C++ (2.0) compiler,
- an X (X11R4) server[2].

2.2. User Expertise

There are two levels of user expertise when developing NSL models:

- the basic level requires familiarity with NSL 'high level' language as described in the manual's next sections; and there is no need to know either C or C++;
- the advanced level requires some basic understanding of C and C++, allowing great flexibility on they type of models which can be described, as well as permitting communication with other software tools[3].

2.3. System Design

The system is composed of several modules, whose functionality is basically the same in the different NSL versions, differing most importantly mainly in the object-oriented emphasis given to the second generation system, which includes the evolution from C into C++ as the system's implementation language. Two aspects of the simulator have been greatly enhanced by the *objectization* of the system; first, the internal design now follows a very natural object-oriented design and implementation, and second, the development of neural networks models is enhanced by the incorporation of special object classes into the simulation language.

The structure of the simulator is shown in Figure 1. The system is divided into two independent sub-systems:

- the Simulator, where model interaction and processing takes place;
- the Window Interface, where all graphics interaction takes place.

The Simulator is composed of:

- the Processing Module, where the network is fully processed;
- the Model Language Compiler, which together with the Model Language Libraries, translates and links the user's Model File, loading it into the Processing Module;
- the Commands Interpreter executing the user's Command Files for setting up and controlling the simulation.

The development of a simulation is then done in two separate steps by the creation of two different types of files:

- Model File: The user describes the network model (a '.c' file to be compiled), which may include any C or C++ code[4].
- Command Files: The user describes the simulation environment (a '.nsl' file to be interpreted), containing all the graphics descriptions, data assignments, and any other run time specifications[5].

---

1 NSL has been developed on a SUN workstation platform.

2 The simulator may be run independent from a graphical interface, in that case the X server is not required. The X windows system has been developed utilizing the XVIEW library.

3 NSL advanced programming section includes explanations directed to more advanced users with some basic knowledge of C and C++.

4 C and C++ supporting code may be stored in separate files to be linked together with the model file.

5 Several command files may be assigned to a single model file.

The Window Interface is composed of:
- the Graphical Displays, where all window and graphics interaction takes places;
- the Graphics Libraries, containing all display functions and object libraries.

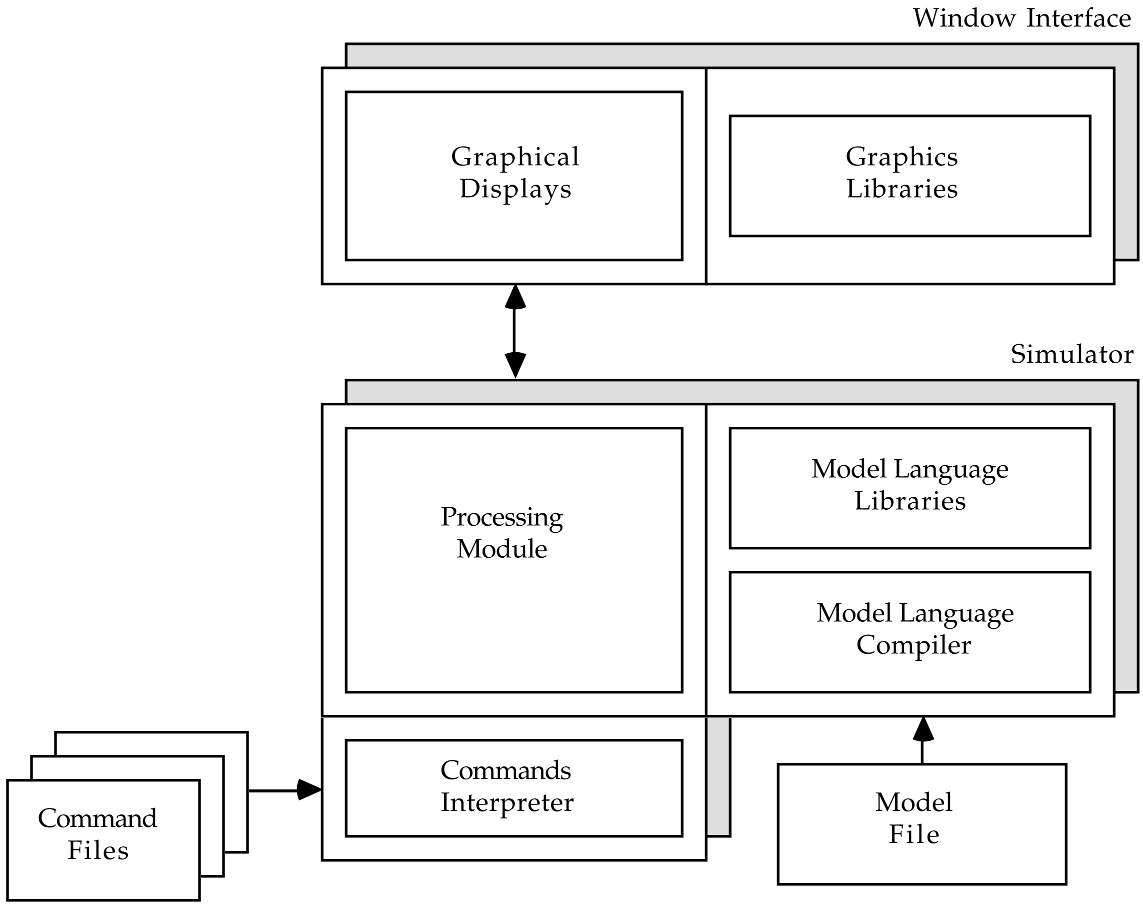


Figure 1
NSL Simulation System

One of the key consideration when designing these modules has been that of extensibility, where new functions and objects may be added into any of these libraries, as will be described in later sections.

3. Neural Network Simulation

In general, neural network simulation can be divided into two distinct categories, biological neural networks, designed to model the brain in a faithful way, and artificial neural networks, intended to apply neural networks computing techniques, including different learning algorithms, into various technological applications [Rumelhart and and McClelland 1986].

Different languages have been written for the simulation of neural networks, with different modelling capabilities, and for different application areas [Goddard et al. 1987, Teeters 1989, Wilson et al. 1989, Wang and Hsu 1990]. One important contrasting point is how networks are syntactically described; in our particular simulation language, we describe neural networks in the form of mathematical equations.

3.1. Neurons and Links

A neural network is, in our abstraction, a set of interconnected concurrent processing units (neurons) having a non-linear dynamic behavior. Therefore, a model description includes (1) declarations of the units in the model, (2) connections between the units, (3) descriptions of inputs external to the network, and (4) descriptions of network outputs.

A basic neuron is shown in Figure 2. It may receive input from many different neurons, while it has only a single output.

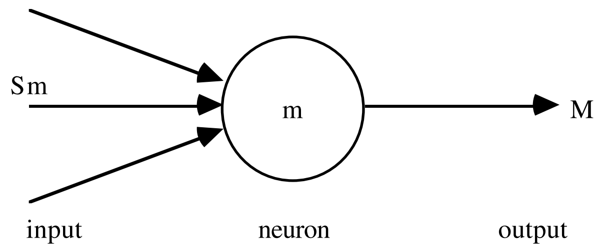


Figure 2
A Basic Neuron

We will focus on neurons whose internal state is described by a single scalar quantity, its membrane potential m, whose time course is described by a differential equation

$$\tau_m \frac{dm(t)}{dt} = f(S_m, m, t)$$

depending on its input S_m. The choice of f defines the particular neural model utilized, including the dependence of m on the neuron's previous history.

The *firing rate* or output of the neuron, M, is obtained by applying some "threshold function" to the neuron's membrane potential,

$$M(t) = \sigma(m(t))$$

where σ could be, for example, a non-linear sigmoid saturation function.

When building neural networks, the output of a neuron serves as input to other neurons. Links among neurons carry a connection weight which describes how neurons affect each other. Links are called excitatory or inhibitory depending on whether the weight is positive or negative. The most common formula for the input to a neuron is

$$S_m = \sum_{i=1}^{n} w_i M_i[t]$$

where $M_i[t]$ is the firing rate of the neuron whose output is connected to the i^{th} input line to the neuron, and w_i is the weight on that link.

It is important to realize that neurons may be modelled with different levels of detail, from sophisticated biophysical models to simple binary models where the neuron is either *on* or *off* at each time step, such as the McCulloch-Pitts neuron model [McCulloch and Pitts 1943]. The Leaky Integrator (see section 3.3) provides a simple yet more natural neuron model. In terms of more detailed neuron models we have *compartmental* models [Kandel and Schwartz 1985], and the Hodgkin-Huxley model [Hodgkin and Huxley 1952]. In any event, neurons are best suited to be treated as *objects* whose internal details are completely hidden away from the rest of the network.

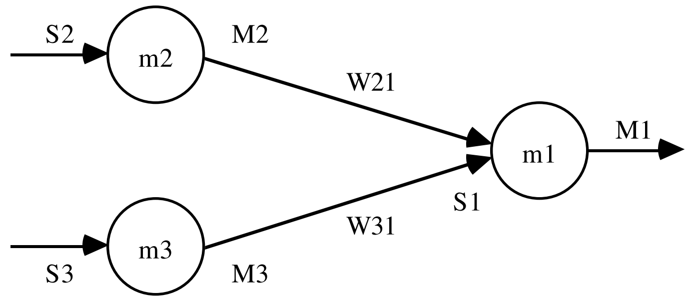


Figure 3
A network of interconnected neurons.

For example, Figure 3 shows a network composed of three neurons, where the input to neuron *m1* is given by specifying *S1* in the following manner,

*S1 = W21*M2 + W31*M3*

where *W21* and *W31* specify the connection weights for the links between *m2* and *m1*, and the links between *m3* and *m1* respectively. *S1* adds together ('+') the factors from the output of neurons *M2* and *M3* multiplied ('*') by their respective connection weights.

3.2. Layers and Masks

As part of our modelling primitives, we have extended the basic neuron abstraction into neuron layers and connection masks, describing spatial arrangements among homogeneous neurons and their connections, respectively. The reason for defining such abstractions is that, in the brain as well as in many neural engineering applications, we often find neural networks structured into two-dimensional layers, with regular connection patterns between various layers.

The computational advantage of introducing such concepts when describing a neural network is that neural layers and interconnection masks can then be concisely described as higher level data structures. Instead of describing neurons on a one by one basis, a layer can be described as an array and, similarly, the connections between layers can be described by a mask storing synaptic weights. An interconnection among neurons would then be processed by computing a spatial convolution of a mask and a layer. For example, if *A* represents an array of outputs from one layer of neurons, and *B* represents the array of inputs to another layer, and if the mask $W(k,l)$ (for $-d \le k,l \le d$) represents the synaptic weight from the $A(i+k,j+l)$ (for $-m \le k,l \le m$) elements to $B(i,j)$element for each *i* and *j*, we then have

$$B(i,j) = \sum_{k=-d}^{d} \sum_{l=-d}^{d} W(k,l)A(i+k,j+l)$$

which can be expressed by the single array operation of convolution

$B = W{*}A$

giving greater computing power to a simple descriptive expression.

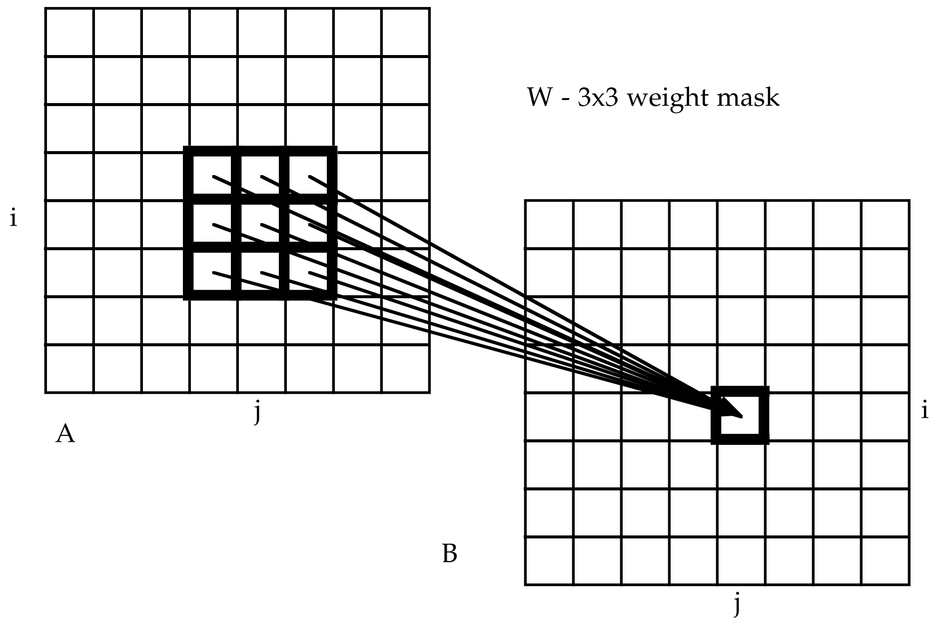


Figure 4
A discrete spatial convolution among mask W and layer A, with B=W∗A.

As shown in Figure 4, mask 'W' stores the different connection or synaptic weights performing a discrete spatial convolution over 'A' to obtain 'B'. In this example both the layers and the mask are square although they could be of any size (as a matter of fact, a layer may have any shape, while 2D rectangular shapes are the most commonly used ones).

In terms of syntax, the equations for the network shown in Figure 2 would be exactly the same if it was describing neuron layers instead of single neurons. The only difference would be that now the factors representing connection weights will now represent connection masks, and the variables representing neurons will now represent neuron layers. The multiplication operator now means convolution.

It is important to realize that there are different ways for treating edge effects, which is basically the convolution with out of bound layer elements. We will address the following three main alternatives:

- treat edges effects as 'zero',
- do a wrap around of layer elements,
- replicate boundary layer elements.

NSL deals with these three types of convolution, as will be described more details in later sections.

3.3. Numerical Methods

Up until now we have left out the internal neural model detail. One widely used neuron model is the 'leaky integrator' [Arbib 1989], whose membrane potential is described by the following differential equation

$$\tau_m \frac{dm(t)}{dt} = - m(t) + S_m(t)$$

where $S_m(t)$ incorporates the input from all the other cells, and τ_m is the time constant. The overall dynamics will depend upon the actual choice of excitatory and inhibitory weights in each $S_m(t)$ and of the time constants.

While different numerical methods may be used to solve a particular neuron model, the neural network architecture and connection weights should *not* have to change depending on this. Different numerical methods keep on evolving and they may be more appropriate according to the sophistication of the model and the processing power of the computing machine.

Many methods use a constant time step Δt. As an example consider two different methods, the 'Euler' method and 'interpolation' method, for solving the previous differential equation. The 'Euler' method replaces the differential equation by

$$m(t+\Delta t) = (1 - \frac{\Delta t}{\tau_m})\, m(t) + \frac{\Delta t}{\tau_m}\, S_m(t)$$

while the 'interpolation' method replaces the differential equation by

$$m(t+\Delta t) = (p)\, m(t) + (1 - p)\, S_m(t)$$

where $p = e^{-\Delta t/t}$

3.4. Learning Methods

An important part of neural network modelling is to be able to introduce learning in a model. There are many different learning algorithms commonly utilized in neural network simulation, among the most popular being *back propagation* [Rumelhart et al. 1986]. This learning algorithm is *not* biological and thus differentiates biological modelling, which is primarily concerned with modelling the brain in a faithful way, from the study of artificial neural networks, otherwise known as connectionism or neural engineering, where the main concern is in applying neural network computing techniques to varied technological applications. Artificial neural networks take advantage of newly developed neural learning algorithms to approach problem domains where traditional programming approaches may not have been very successful. Biological learning can be exemplified by Hebbian learning [Hebb 1949].

4. Amari/Arbib Maximum Selector (Didday) Model

In this section we present the 'Maximum Selector' model [Amari and Arbib 1977] based on the Didday model for prey selection [Didday 1976]. This model uses a competition mechanism to obtain a single winner in the network, and will serve as sample network to be simulated in NSL. Figure 5 shows the network's connectivity among the different layers.

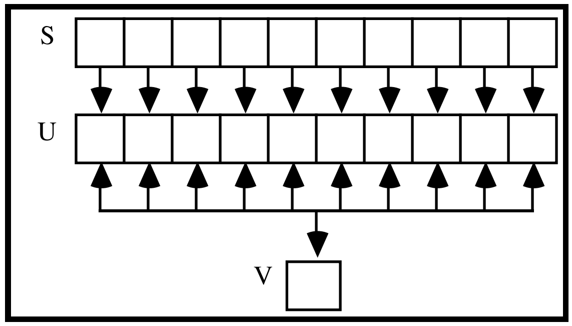


Figure 5

Layers and interconnections as defined by the sample network.

The model represents an array of 10 cells with membrane potential *u* and firing rate *U*, connected to a single inhibitory cell, with firing rate *V* and membrane potential *v*. The circuit is driven by an input array *S*.

The mathematical description of the model is

$$\tau_u \frac{du_i(t)}{dt} = -u_i + w_1 f(u_i) - w_2 g(v) - h_1 + s_i \qquad 1 \le i \le n$$

$$\tau_v \frac{dv}{dt} = -v + \sum_{i=1}^{n} f(u_i) - h_2$$

where

$$f(u) = \begin{cases} 1 & u > 0 \\ 0 & u \le 0 \end{cases}$$

and

$$g(S_i) = \begin{cases} S_i & S_i > 0 \\ 0 & S_i \le 0 \end{cases}$$

The model has the parameters w1, w2, h1, h2, with the restriction that 0 ≤ hl, and 0 ≤ h2 < 1, and n = 10 (see Arbib 1989, Sec. 4.4 for a full exposition).

4.1. didday.c

Let's take the previously described 'Didday' model as our sample model.

(As in C++ comments are preceded by either '//', and take effect until the end of the line; or as in both C and C++ they start with '/*' until a '*/' is used to close the comments. ';' marks the end of the expression.)

4.1.1. Declarations

We start by providing comments to name the program, a line to tell the compiler that it should use the library 'nsl_include.h'[6], and a line to name the program, declaring that it is a network and that its name is DIDDAY.

In NSL we use two layers of elements to represent one layer of neurons. The first layer holds the membrane potentials, while the second layer holds the firing rates. The mapping from the first layer to the second is simply the neuron's nonlinearity (e.g. a step, ramp, or sigmoidal function). A convention is to use the same name for each layer, in lower case for the membrane potential layer and in upper case for the firing rate layer. This convention aids the user, but is not enforced by the compiler. In NSL 2.1, an array x is declared as DATA(x) if it has a single element, VECTOR(x,n) if it one-dimensional with n elements, and ARRAY(x,m,n) if it is a 2-dimensional mxn array. Each declaration is terminated by a semi-colon. Thus the "introduction" to the model looks as follows:

```
/********************************/
/*          didday.c            */
/********************************/

# include "nsl_include.h"

NETWORK(DIDDAY);

VECTOR(S,10);   // network input
VECTOR(u,10);   // membrane potential
VECTOR(U,10);   // firing rate

DATA(v);        // membrane potential
DATA(V);        // firing rate
```

We next declare some of the constants needed to define the model (whose values will be set in didday.nsl). If a constant is "really" constant, we may write the numerical value directly into the program. However it is often good practice to name these model parameters (a) so that the user may better understand the role that it plays in the model; and (b) to make it easy to change values when running experiments on how different features of the model affect its behavior.

```
DATA(k);
DATA(tu);
DATA(tv);
DATA(h1);       // threshold for u
DATA(h2);       // threshold for inhibitory unit
DATA(w1);
DATA(w2);
```

The description of the model is now given in two parts, the INIT_MODULE and the RUN_MODULEs:

---

6 The current release requires the user to include the line:

include "nsl_include.h"

at the beginning of the file. This line tells NSL to incorporate all object declarations and macro definitions.

4.1.2. INIT_MODULE

INIT_MODULE(didday_init) tells the program how to initialize appropriate variables and constants [we shall later discuss how other parameters are initialized]. INIT_MODULE runs just once, at the beginning of a run of the model [and we shall later discuss how the length and "grain" of the run is specified]. Following the header, INIT_MODULE(didday_init) starts with { and ends with }. Each line is terminated with a semi-colon. Note that, in the spirit of array processing, we may, where appropriate, specify all values of an array in a single assignment.

```
INIT_MODULE(didday_init)
{
    u = 0;
    v = 0;
    U = 0;
    V = 0;
}
```

These expressions are called every time the model is 're-run' and should include any model initialization lines such as resetting layer values to zero.

4.1.3. RUN_MODULEs

RUN_MODULE(didday_x) has as many parts as there are "natural" submodules which comprise the module. We could have an x for every variable in the module, or one x for the whole module. However, in this example, we specify two RUN_MODULEs, one called didday_U to specify how u and U are updated, and another called didday_V for v and V.

```
RUN_MODULE(didday_U)
{
    DIFF.eq(u,tu) = - u + w1 * U - w2 * V - h1 + S;
    U = NSLstep(u,k);
}
```

```
RUN_MODULE(didday_V)
{
    DIFF.eq(v,tv) = - v + U.sum() - h2;
    V = NSLramp(v);
}
```

Note the NSL syntax for a differential equation. We simply rewrite

```
tau*du/dt = f(u,x)
```

as

```
DIFF.eq(u,tau) = f(u,x)
```

where the differentiated variable u is followed by its time constant tau on the left hand side, and f(u,x) is simply the right hand side of the differential equation rewritten in NSL syntax. In the present example, the only convention to be noted is that $\sum_{i=1}^{n} f(u_i)$ — which is just $\sum_{i=1}^{n} U_i$ — is rewritten as U.sum() with the sigma/summator written as a layer 'member' function.

The nonlinear functions that transform a membrane potential into its firing rate are provided by functions that are included in the NSL library. Thus

$$f(u) = \begin{cases} 1 & u > 0 \\ 0 & u \leq 0 \end{cases}$$

is given by NSLstep(u,0), while

$$g(S_i) = \begin{cases} S_i & S_i > 0 \\ 0 & S_i \leq 0 \end{cases}$$

is given by NSLramp(v).

This completes didday.c, the actual specification of the model in NSL. The INIT_MODULE initializes the state of the computer for the run. An actual run has a given time step and a given duration and proceeds by running each of the RUN_MODULEs at each time step in order to update the values of all the variables from the start to the end of the time step. The user may specify which numerical method is to be used in approximating the solution of each differential equation in going from u(t) to u(t+Δt), etc. The code in didday.nsl, to be described in section 4.2, provides the missing information for a given run of the model, as well as the information on how to set up the graphics for displaying the results of the simulation.

4.1.4. Compilation

We will name the model file 'didday.c', although more generally a model file should be called '*file*.c'; where '*file*' is a totally arbitrary name, and what's important is the '.c' suffix. Compilation[7] and linkage of this file is done by executing the following line (the file 'nsl_main.c' should be present in the directory where compilation is taking place[8]):

nsl_link *file*.c

The system will then create an executable file called 'nsl', if no errors were made in the model file, which is ready for running[9]. (Several model files may be linked with NSL by simply adding their names to the 'nsl_link' command file list[10].)

At this point the model has been linked with NSL and it can be run by typing 'nsl',

---

7 This compilation requires the availability of a C++ compiler.

8 'nsl_main.c' contains the C/C++ 'main' routine where nsl initialization takes place. Linking to any other C/C++ programs can be included inside this file as explained in the advanced programming section. A copy of this file should exist in the directory where compilation is taking place, and can be obtained by running 'nsl_init' which also copies the sample didday model files.

9 If running graphics remotely run nsl as follows: 'nsl -display [hostname]:0' where [hostname] is the name of the machine physically displaying the graphics.

10 A single 'nsl_main.c' is required even if multiple model files are compiled together.

nsl

The NSL prompt will appear,

nsl>

Although we could set up the simulation environment interactively, it is more convenient to have it stored in a batch file (a '.nsl' file).

4.2. didday.nsl

The simulation set up is provided in the form of simulation parameters, model parameters, including network input, and graphics.

4.2.1. Simulation Parameters

In the first part of didday.nsl, we tell the compiler to set up the network DIDDAY (i.e., it should link the DIDDAY network stored in didday.c to the material that follows), to use the EULER integration method, and to run the model from time 0 to time 20.0 using a time step delta of 0.1, thus making a total of 200 iterations of the model.

```
// didday.nsl simulation set up

set network DIDDAY

// network
set integration EULER
set delta 0.1
set end_time 20.0
```

4.2.2. Model Parameters

We also tell it to set those network parameters whose values were not specified in didday.c.

```
set data_value tu
1.0
set data_value tv
1.0
set data_value k
0.1
set data_value w1
1.0
set data_value w2
1.0
set data_value h1
0.1
set data_value h2
0.5
```

Next, we specify the input to be used in this simulation. In this case, we hold the input constant throughout the run. All values of the S array are 0 except for 1.0 in the fourth entry (arrays start at the zeroth entry) and 0.5 in the sixth entry.

```
set data_value S
0 0 0 0 1.0 0 0.5 0 0 0
```

4.2.3. Graphics

Finally, we specify the windows to be used in displaying the input S and the membrane potential u.

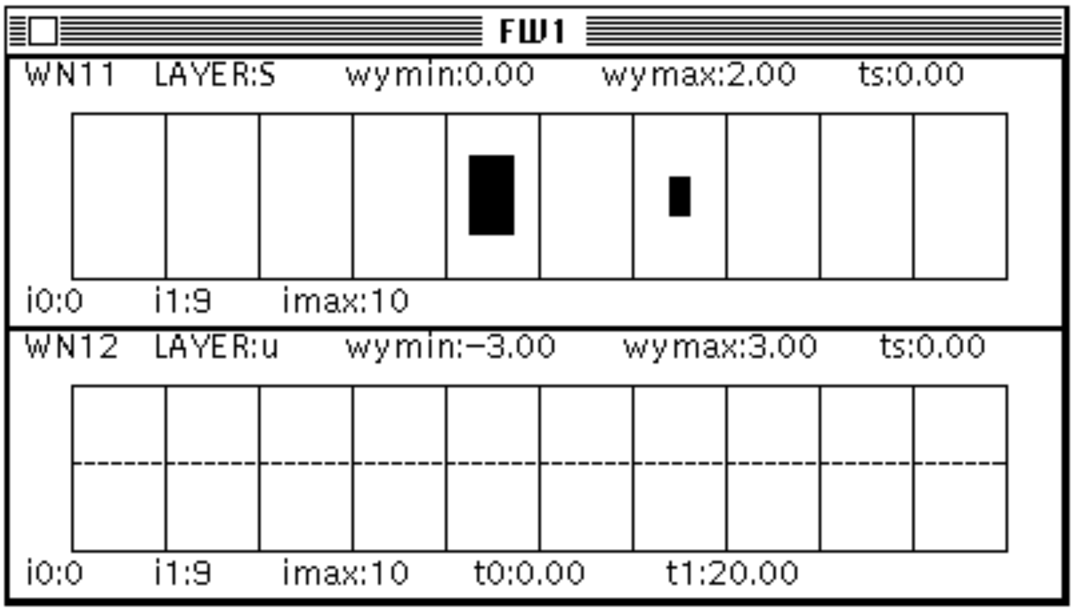


Figure 6
The Didday model window set up

We specify that both displays will be contained in a frame called FW1, which will have two rows:

```
// graphics

create window_interface

set frame_name FW1
set frame_X0 250
set frame_Y0 100
set frame_width 510
set frame_height 335
set frame_rows 2
create display_frame
```

The first row contains a window WN11 which displays the vector S, with each element shown as an "area level graph"; while the second row contains a window WN12 which displays the vector u, with each element shown as a "temporal graph".

```
set window_name WN11
set window_layer S
set window_graph area_level_graph
set window_wymin 0
set window_wymax 2
create display_window

set window_name WN12
set window_layer u
set window_graph temporal_graph
set window_wymin -3
set window_wymax 3
set window_t1 20
create display_window
```

4.2.4. Running the Simulation

The previous commands could have been loaded interactively into NSL's command interpreter. If the commands were stored into a file called 'didday.nsl', then this file can be loaded by typing

nsl> load didday

To run the model simply type (the output of a simulation is shown in Figure 7)

nsl> run

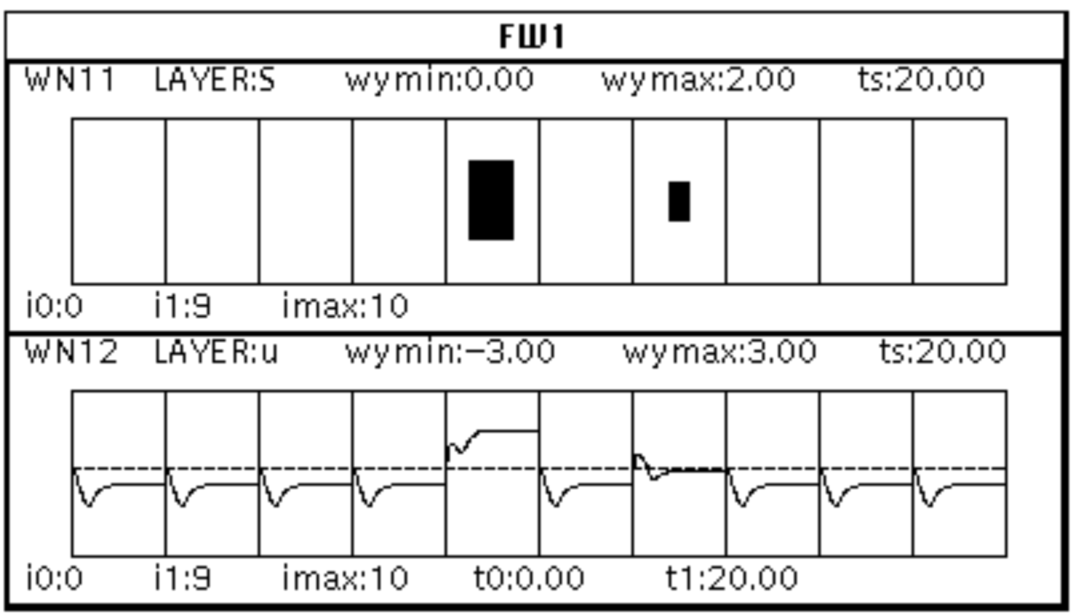


Figure 7
The output of Didday model

The user may stop the simulation at any time by typing '^C'. To resume the simulation from the interrupt point type,

```
nsl> cont
```

5. NSL Model Language

NSL is built following an object-oriented programming paradigm. NSL provides a set of basic network object classes intended to give the user simple yet powerful constructs without restricting in any way the complexity and the type of models which can be described. It is the goal of NSL to provide tools in a bottom-up fashion, where future system versions will incorporate more sophisticated network structures built on top of the currently available ones. The following sections will give the user an introduction to NSL basic object structures.

5.1. Object-Oriented Programming

Since NSL is designed as an object-oriented language, it is worth understanding the concepts behind this kind of programming methodology. We will put emphasis on C++ due to its important role as NSL's implementation language. Specifically the reader may want to become familiar with C++ as a programming language by referencing C++ from Stroustrup [1989] or any other C++ textbook.

5.2. Model Creation

Writing a simulation in NSL consists of several steps. The first step is to describe the model as a set of NSL equations, stored into the model file ('.c' file). The second step consists of writing a simulation environment and stored into simulation files, as many as needed ('.nsl' files). Simulation files contain input and parameter assignments which may vary during different simulation runs. They also include any graphical set ups.

5.3. Model Object Classes

NSL basic network objects are of three types: NETWORK, MODULE, and LAYER. These classes serve as super-classes (utilizing C++ terminology) for other derived classes, meaning that any other classes inherit their respective characteristics. These names define macros, which get translated into internal NSL object class declaration, as will be explained in the advanced programming section in further detail.

(In what follows, characters shown in **bold** letters are meant to be typed exactly as they appear, and should not be used as variable names; words appearing in *italics* are user supplied strings, either names or values.)

5.3.1. Network

A model is described by a **NETWORK** object, which points to a list of LAYERs and processing MODULEs. Figure 8 shows the network structure.

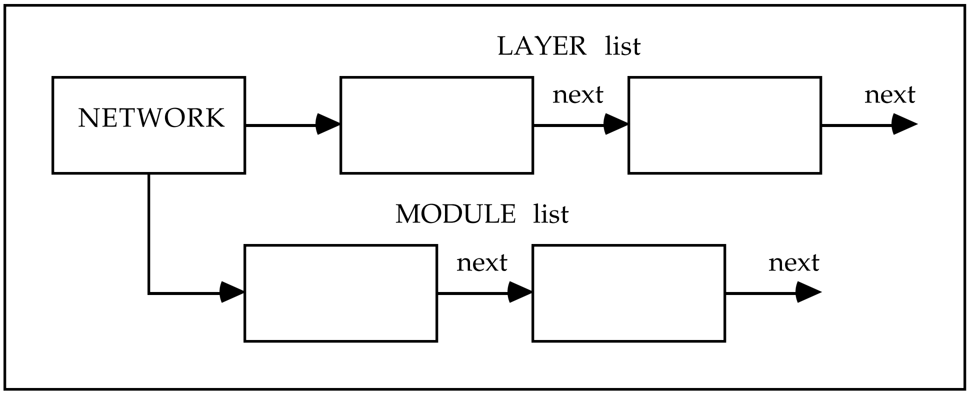


Figure 8
NETWORK class.

The declaration of the network object should always precede all other declarations in the model file,

NETWORK(*model_name*);

where *model_name* is the name to be used when identifying this particular model[11].

For our sample network, the first line of the '.c' file would be:

NETWORK(DIDDAY);

(Note that this entry is required in every model file. Without it the model file will not be translated.)

5.3.2. Layer

The **LAYER** is a super-class from which three data layer classes are derived: **DATA**, **VECTOR**, and **MATRIX** classes. These three classes vary according to the number of dimensions they contain. A data layer represents a layer with a single unit, a vector layer represents a one dimensional array of units, and a matrix layer represents a two dimensional array of units. (Higher numbers of dimensions may be incorporated into new data structures, if so desired; similarly, a user may define classes of objects which may have a structure other than the traditional rectangular ones, although this is beyond the scope of this manual.) Figure 9 shows the layer class hierarchy.

---

[11] *model_name* doesn't have to be the same as the name of the file. *model_name* is used to refer to the model from the simulation environment.

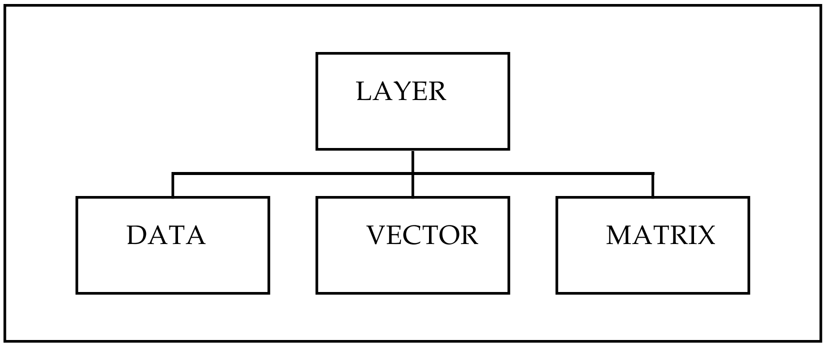


Figure 9
NSL layer classes.

The **LAYER** super-class is not directly utilized in specifying a network, instead the user utilizes the derived layer classes, **DATA**, **VECTOR**, and **MATRIX**. The following is the syntax for each class declaration,

Data layer:

DATA(*layer_name*);

```
DATA(S);                // a data layer with a single element
```

Vector layer:

VECTOR(*layer_name*,*size*)[12];

```
VECTOR(INP,10);         // a 10 element vector layer
```

Matrix layer:

MATRIX(*layer_name*,*rows*,*columns*);

```
MATRIX(A,10,8);         // a 10x8 element layer
```

where *layer_name* specifies the name of the layer, *size* specifies the length of the vector, *rows* specifies the number of rows of a matrix layer, and *columns* represent the number of columns of a matrix layer.

For our sample model, we should enter the following declarations after the NETWORK line.

```
VECTOR(S,10);       // network input S
VECTOR(u,10);       // membrane potential u
VECTOR(U,10);       // firing rate U

DATA(v);            // membrane potential v
DATA(V);            // firing rate V
```

[12] Vectors may also be declared as column vectors,
COL_VEC(*layer_name*,*size*);
or row vectors (which corresponds to the default VECTOR declaration type),
ROW_VEC(*layer_name*,*size*);

S is the network input layer while u and U represent the membrane potential and firing rate layer respectively, and v and V represent the membrane potential and firing rate layer respectively. (The convention followed here, which is not enforced, is to name membrane potential layers with lower case letters, while firing layers are named with upper case letters.)

The following seven declarations, although similar to those of v and V, represent network parameters.

```
DATA(tu);          // u time constant
DATA(tv);          // v time constant
DATA(k);           // step function threshold
DATA(w1);
DATA(w2);
DATA(h1);
DATA(h2);
```

These parameters will be assigned values through the '.nsl' file to be read by the commands interpreter.

5.3.3. Module

The **MODULE** class is utilized by declaring any of the following (in the form of macro declarations): **MODULE**, **INIT_MODULE**, and **RUN_MODULE**. The module contains the code describing the equations that are to be processed by NSL, differing only in the time intervals on which the modules are applied, depending on the above declaration types. (If there are no modules, no processing will be done !!) There may be as many modules as desired, having any number of equations in each of them, including having a single module for all the equations, or one module per equation[13]. The order of processing is the same order in which the modules are declared. (Currently these modules are serially processed, although the goal in the future is to distribute them and take advantage of their concurrent nature.) In what follows *module_name* is the name of the module[14].

MODULEs are processed for every time iteration, "t≥0".

MODULE(*module_name*)
{
    ...layer dynamics...
}

INIT_MODULEs are processed only at "t=0" and should contain initialization code which gets called every time the model is 'reset'.

INIT_MODULE(*module_name*)
{
    ...layer initialization...
}

---

[13] The current version doesn't support layer declarations inside a module, only expressions.

[14] Since several model files may be compiled together, it is required to name each module differently (this requirement is for the present system release, and will be discarded in the future). Different naming can be achieved by concatenating the particular model name with a local separate name. For example, for a model called 'didday', a module could be called 'didday_mod1'.

RUN_MODULEs are processed during the entire simulation with the exception of the initialization time, when "t>0".

```
RUN_MODULE(module_name)
{
    ...layer dynamics...
}
```

For our sample model, and after having declared the layers in the network, we define the processing modules.

The first one is an INIT_MODULE resetting the dynamic layers to zero every time the network is re-computed.

```
INIT_MODULE(didday_init)
{
    u = 0;
    v = 0;
    U = 0;
    V = 0;
}
```

Two RUN_MODULEs are declared for the didday model, one for layer U and another one for layer V. Each module calculates new membrane potentials and firing rates.

The layer dynamics are all in the form of mathematical descriptions. These expressions will be explained in the next sections.

```
RUN_MODULE(didday_U)
{
    DIFF.eq(u,tu) = - u + w1*U - w2*V -h1 + S;
    U = NSLstep(u,k);
}
RUN_MODULE(didday_V)
{
    DIFF.eq(v,tv) =  - v + U.sum() - h2;
    V = NSLramp(v);
}
```

Processing is sequential, so in this case layer U would get computed before layer V. If we wanted to simulate parallel neural update, we can do this by defining the modules in a different way, so input to layers will get processed first, and only after all layers have their input processed, will the output then get computed.

```
RUN_MODULE(didday_mem_pot)
{
    DIFF.eq(u,tu) = - u + w1*U - w2*V - h1 + S;
    DIFF.eq(v,tv) =  - v + U.sum() - h2;
}
RUN_MODULE(didday_firing)
{
    U = NSLstep(u,k);
    V = NSLramp(v);
}
```

By combining the update of all neuron firing layers layers into the last module, we simulate parallel update, since no output values will be computed until every layer has its input processed.

5.4. Expressions

Layer dynamics, in the form of mathematical equations, describe the interactions among the different layers in the network. On the left hand side of the equation the layer being described is specified. On the right hand side of the equation, an expression is given specifying the particular input for that layer, where the expression may be composed of sub-expressions. Positive sub-expressions describe an excitatory effect while negative sub-expressions describe an inhibitory effect on the layer.

For example, equations can be as follows:

```
DIFF.eq(u,tu) = -u + w1*U - w2*V - h1 + S;
U = NSLstep(u,k);
```

The general form of the equation is,

lhs= expression

where *lhs* is the left hand side of the equation and may be one of the following,

lhs **->** *layer*
lhs **->** *var*
lhs **->** *diff*

where

layer is either a data, vector or matrix structure,
var is a low level C/C++ variable structure such as 'float',
diff is a differential type of equation, which will be further described in the numerical method section.[15]

expression is defined as follows,

expression **->** (*expression*)
expression **->** *expression operator expression*
expression **->** *term*

operator is defined as follows,

operator **->** + | - | * | ^ | /

where,

'+' is the layer addition operator
'-' is the layer subtraction operator
'*' is the layer convolution operator
'^' is the pointwise layer multiplication operator[16]

---

[15] Both sides of the equality should return compatible value types, in particular, if the left hand side is of 'var' type, the right hand side should also return a 'var' type.

[16] Due to its current implementation, the '^' has the lowest precedence, requiring parenthesis around the partial expression

'/' is the pointwise layer division operator

term is defined as any of the following:

term **->** *layer*
term **->** *var*
term **->** *val*
term **->** *function*

where,

layer -> *data* | *vector* | *matrix*
var is a variable returning a 'float' type
val is a 'float' number
function **->** *function_name(par1, par2, ..., parN)*

where *parj* ($1 \leq j \leq N$) can be any layer type (data, vector, or matrix), or a number value depending on the function argument specification.

(All operators have a corresponding function performing the same operation. In the C++ implementation, operators are overloaded to provide the right functionality. For example adding two matrices, 'a+b' operators is similar to calling 'NSLadd(a,b)' which returns a matrix containing the added pointwise matrix values.)

Operations not covered here by the above operators (e.g. regular matrix multiplication) are achieved by function calling. Some of these functions provide the nonlinear mappings (between the neuron's *membrane potential* and *firing rate*) which are needed in neural net modeling.

Direct value assignment to layers may be achieved in two ways:

- By specifying a single value for the whole layer, available to all layer types (as included in the above description).

layer_name = *value*

e.g. S = 1.5;

- By specifying a value for a particular layer unit, and depending on the type of layer.

Data layer:

layer_name.**elem**() = *value*

Vector layer:

layer_name.**elem**(*index*) = *value*

Matrix layer:

layer_name.**elem**(*row_index*,*col_index*) = *value*

---

in order to ensure proper operation. This shortcoming should be overcome in the next NSL release.

where *index* specifies a vector element, *row_index* specifies a matrix row element, and *col_index* specifies a matrix column element. (All indices start from '0').

5.5. Layer Class Functions

There are two types of functions, layer class functions and layer library functions. Layer class functions are of the form: *layer.function*(), while layer library function are of the form: *function*(*layer*). Layer class function are basically functions applied to a particular layer, while layer library functions are routines taking any number of layer arguments (not all arguments have to be of layer type). The following sections describe the most relevant layer class functions.

5.5.1. Transformations

While there are three basic types of layers, it is important to be able to do transformations among them. Figure 10 shows row and column transformations among vectors and matrices.

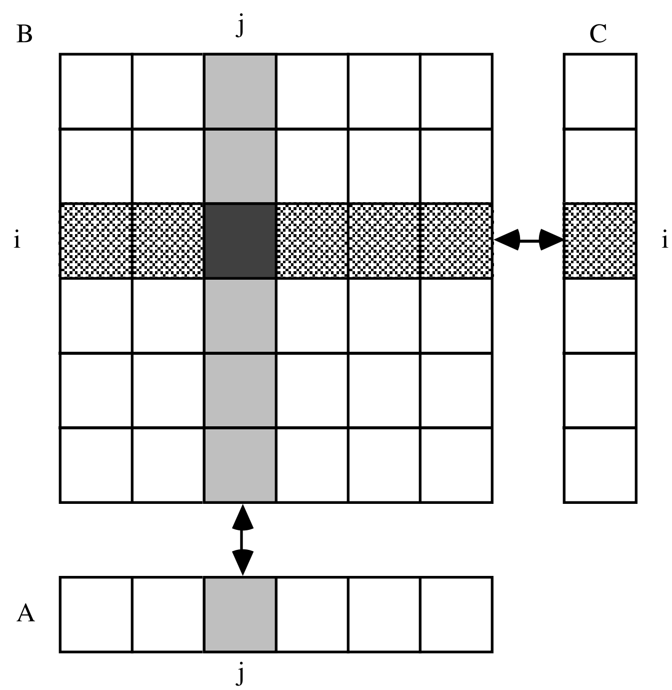


Figure 10
Vector and Matrix Transformations

Transforming from vectors into matrices (either as a row or column vector expansion)[17].

matrix = *row_vector*.expand_row(*n*);
matrix =*col_vector*.expand_col(*n*);

From Figure 10:

B = A.expand_row(*n*)

[17] Previously know as NSLcol_vec_to_mat and NSLrow_vec_to_mat.

B(i,j) = A(j)

B = C.expand_col(*n*)

B(i,j) = C(i)

Transforming from matrices into vectors (either as a row or column vector reduction)[18].

row_vector = *matrix*.reduce_row();
col_vector = *matrix*.reduce_col();

From Figure 10:

A = B.reduce_row()

$$A(j) = \sum_i B(i,j)$$

C = B.reduce_col()

$$C(i) = \sum_j B(i,j)$$

Since vectors may be row-wise or col-wise, it is important to have a transformation from one to the other[19]. To transpose row vectors and columns vectors[20]:

To set a vector as a row vector, do,

row_vector = *vector*.transpose_row()

To set a vector as a column vector, do

col_vector = *vector*.transpose_col()

To switch a vector from its current form, row or col, to the opposite one, do,

vector = *vector*.transpose()

(applying it twice consecutively brings the vector to its original form).

Other functions set or get matrix rows and columns:

vector = *matrix*.get_row(*i*)

vector = *matrix*.get_col(*j*)

18 Previously known as NSLmat_to_row_vec and NSLmat_to_col_vec.

19 NSL will automatically draw vectors on the screen in the direction they are defined. Internally all vectors are stored a unified way, having simply an internal flag which tells what type they are row or column vectors.

20 Currently there is no matrix transpose function.

matrix = *matrix*.set_row(*vector*,*i*)

matrix = *matrix*.set_col(*vector*,*j*)

where *i* is a row index, and *j* is a column index.

For example, if A = B.set_row(b,i) then

A[i,j] = b[j]

while

A[k,j] = B[k,j] for k ≠ i

5.5.2. Sectors

Sector functions deal with parts of layers, either for setting them or to get part of them.

'get_sector' returns a sector of the respective vector or matrix, while 'put_sector' returns a larger vector or matrix with a copy of the specified vector or matrix inside, and the rest of the elements left to zero. The return layers are of size equal to the layer in front of the member function. (Currently the sector's starting point is specified and the whole sector is either copied or retrieved from the layer.)

Vector sectors:

vector = *vector*.get_sector(*i0*,*j0*);

vector = *vector*.put_sector(*vector0*,*i0*);

where *i0*,*i1* specifies a vector sector.

Matrix sectors:

matrix = *matrix*.get_sector(*i0*,*i1*,*j0*,*j1*);

matrix = *matrix*.put_sector(*matrix0*,*i0*,*j0*);

where *i0*,*i1*,*j0*,*j1* specifies a matrix sector.

5.5.3. Other

To get the sum of all elements in either a data, vector or matrix layer ,returning a 'num_type' ('float') value:

num_type = *layer*.sum();

To get the max or min values from either data, vector, or matrix layers, and returning a 'num_type' ('float') value:

num_type = *layer*.max();
num_type = *layer*.min();

5.6. Layer Library Functions

The layer library includes a list of functions which usually take more than one layer as argument to the functions. These functions are independent from the layer classes themselves, and provide further extensibility. Among these functions there are arithmetic functions, including convolution, and threshold functions.

Efficiency of these functions is of major importance since the system spends most of its CPU time around these functions. All functions have a 'NSL' prefix to distinguish them from any user defined functions.

The user may define additional functions, to be written in either C++ [Stroustrup 1987], or C [Kernighan and Ritchie 1978], as will be explained in the advanced NSL programming section.

5.6.1. Addition

Addition (operator '+') is a pointwise addition among corresponding elements of similar layers. The NSL function corresponding to it is 'NSLadd(*a,b*)'.

Vectors can be added with other vectors of the same size, or with data and with 'float' values.
Matrices can be added with other matrices of the same size, with data and with 'float' values.
Vectors and matrices cannot be added together.
When a vector or matrix is added to a data or 'num_type' value, this value is added to every component of the vector or matrix.

For example, having matrices X and Y[21]

```
X =     1,1,1          Y =     2,2,2
        1,2,1                  1,3,1
        1,1,2                  2,2,2

X + Y = 3,3,3          X + 1 =  2,2,2
        2,5,2                   2,3,2
        3,3,4                   2,2,3
```

5.6.2. Subtraction

Subtraction (operator '-') is a pointwise subtraction among corresponding elements of similar layers. The NSL function corresponding to it is 'NSLsub(*a,b*)'.

Vectors can be subtracted from other vectors of the same size, from data and from 'float' values.
Matrices can be subtracted from other matrices of the same size, from data and from 'float' values.
Vectors and matrices cannot be subtracted from each other.
When a vector or matrix is subtracted from/to a data or 'num_type' value, this value is subtracted from/to every component of the vector or matrix.

5.6.3. Multiplication

Multiplication can be of different forms.

---

[21] Matrix values are entered row by row.

Pointwise multiplication (operator '^'[22]) among corresponding elements of similar layers. The NSL function corresponding to it is 'NSLmult(*a,b*)'.

Vectors can be multiplied with other vectors of the same size, with data and with 'float' values.
Matrices can be multiplied with other matrices of the same size, with data and with 'float' values.
Vectors and matrices cannot be pointwise multiplied together .
When a vector or matrix is pointwise multiplied to a data or 'num_type' value, this value is multiplied to every component of the vector or matrix.

NSL also supports vector dot product and matrix multiplication (scalar multiplication is supported by both). The NSL function corresponding to it is 'NSLprod(*a,b*)'.

If X and Y are two vectors of the same size, then

NSLprod(X ,Y) = $X^T Y$

returns a data value (size 1) equal to the dot product of the two vectors.

If X and Y are two matrices of sizes nxm and mxr, respectively, then

NSLprod(X ,Y) = X Y

returns a matrix of size nxr.

5.6.4. Division

Pointwise division (operator '/') among corresponding elements of similar layers. The NSL function corresponding to it is 'NSLdiv(*a,b*)'.

Vectors can be divided with other vectors of the same size, with data and with 'float' values.
Matrices can be divided with other matrices of the same size, with data and with 'float' values.
Vectors and matrices cannot be pointwise divided together .
When a vector or matrix is pointwise divided with/to a data or 'num_type' value, this value is divided with/to every component of the vector or matrix.

For example, having matrices X and Y

```
X =   1,1,1          Y =   2,2,2
      1,2,1                1,3,1
      1,1,2                2,2,2

X / Y =  0.5,0.5,0.5
          1,0.67,1
         0.5,0.5,1

X / 2 =  0.5,0.5,0.5
         0.5,1,0.5
         0.5,0.5,1
```

[22] Due to the current implementation, the '^' has lower precedence than the other arithmetic operators, making it necessary to add parentheses around the multiplication expression.

5.6.5. Convolution

Three types of convolution edge effect are treated by NSL:

zero effect

The edge effect is treated as '0'. (Default action.)

For example, having layer X and mask M as follows

```
X =     1,1,1,1,1
        1,2,2,2,2
        1,2,4,4,4
        1,2,4,8,8
        1,2,4,8,0

M =     1,1,1
        1,2,1
        1,1,1
```

will result in Y = M*X as follows,

- first a larger matrix is created with 0 values for the edges (the size of the new matrix depends on both the size of the mask and the convolved matrix; for example for a (2d+1)x(2d+1) mask, the border of zeroes must be d-deep):

```
X0 =    0,0,0,0,0,0,0
        0,1,1,1,1,1,0
        0,1,2,2,2,2,0
        0,1,2,4,4,4,0
        0,1,2,4,8,8,0
        0,1,2,4,8,0,0
        0,0,0,0,0,0,0
```

- second we overlap M on the left top corner of X0 with M[0,0] on top of X0[0,0] so the first convolution will be given by

Y[0,0] = 0*1 + 0*1 + 0*1 + 0*1 + 1*2 + 1*1 + 0*1 + 1*1 + 2*1 = 6

and so on for the other elements.

The function for this kind of effect would be called as:

```
Y = NSLconv_zero(M,X);
```

or

```
Y = M*X
```

since the '*' default operation is the zero-edge convolution.

wrap around

The layer is replicated successively so values are wrapped around.

For example, having layer X and mask M as before will result in Y = M*X as follows

- first a larger matrix is created with wrapped around values for the edges (the size of the new matrix depends on both the size of the mask and the convolved matrix)[23]:

```
X0 =    0,1,2,4,8,0,0
        1,1,1,1,1,1,1
        2,1,2,2,2,2,1
        4,1,2,4,4,4,1
        8,1,2,4,8,8,1
        0,1,2,4,8,0,1
        0,1,1,1,1,1,0
```

- second we overlap M on the left top corner of X0 with M[0,0] on top of X0[0,0] so the first convolution will be given by

Y[0,0] = 0*1 + 1*1 + 2*1 + 1*1 + 1*2 + 1*1 + 2*1 + 1*1 + 2*1 = 12

and so on for the other elements.

The function for this kind of effect would be called as:

```
Y = NSLconv_wrap(M,X);
```

<u>copy edge</u>

The edge element values of the layers are replicated as if it was an infinite layer.

For example, having layer X and mask M as before will result in Y = M*X as follows

- first a larger matrix is created with copied values for the edges (the size of the new matrix depends on both the size of the mask and the convolved matrix):

```
X0 =    1,1,1,1,1,1,1
        1,1,1,1,1,1,1
        1,1,2,2,2,2,2
        1,1,2,4,4,4,4
        1,1,2,4,8,8,8
        1,1,2,4,8,0,0
        1,1,2,4,8,0,0
```

- second we overlap M on the left top corner of X0 with M[0,0] on top of X0[0,0] so the first convolution will be given by

Y[0,0] = 1*1 + 1*1 + 1*1 + 1*1 + 1*2 + 1*1 + 1*1 + 1*1 + 2*1 = 11

and so on for the other elements.

---

23 Edge corners are currently assigned zero.

The function for this kind of effect would be called as:

Y = NSLconv_copy(M,X);

As can be seen when comparing the results, and as the name implies, the differences are shown on the edges; the further away from the edges, the less the importance of edge effects.

5.6.6. Threshold

Figure 11 shows the available threshold functions. These functions are applied to every element in the layer independently. The return layer type is similar to the first argument given in the function, where B indicates the returning value, while A indicates the input value. The function may be overloaded, i.e. its arguments may be of different types. The different argument types are given in Appendix 13.2.

step

NSLstep(*layer0*)

For example,

B = NSLstep(A)

corresponds to an iteration over all i,j for

if A[i,j] < 0 B[i,j] = 0
else if A[i,j] ≥ 0 B[i,j] = 1

ramp

NSLramp(*layer0*)

For example

B = NSLramp(A)

corresponds to an iteration over all i,j for

if A[i,j] < 0 B[i,j] = 0
else if A[i,j] ≥ 0 B[i,j] = A[i,j]

saturation

NSLsaturation(*layer0*)

For example,

B = NSLsaturation(A)

corresponds to an iteration over all i,j for

if A[i,j] < 0 B[i,j] = 0
else if 0 ≤ A[i,j] < 1 B[i,j] = A[i,j]

else if $A[i,j] \geq 1$ $B[i,j] = 1$

sigmoid

NSLsigmoid(*layer0*)

For example,

B = NSLsigmoid(A)

corresponds to an iteration over all i,j for

if $A[i,j] < 0$ $B[i,j] = 0$
else if $0 \leq A[i,j] < 1$ $B[i,j] = A[i,j]^2(3 - 2A[i,j])$
else if $A[i,j] \geq 1$ $B[i,j] = 1$

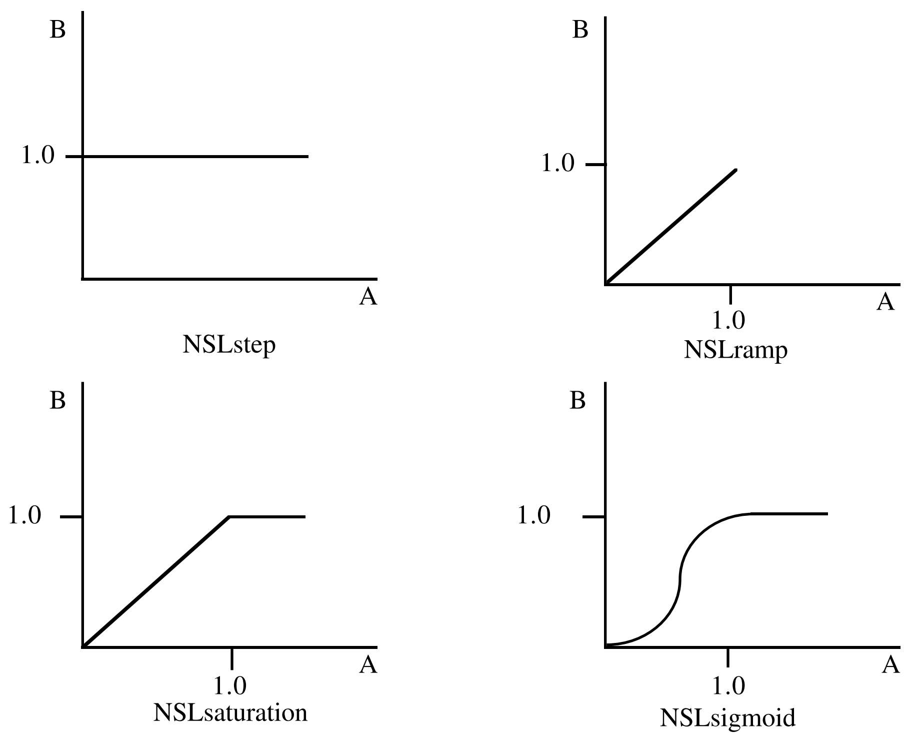


Figure 11
NSL threshold functions

To get arbitrary threshold functions as shown in Figure 12, the following function call can be given:

step

B = (ky2-ky1)*NSLstep(A-kx1) + ky1, or
B = NSLstep(A,kx1,ky1,ky2)

corresponds to an iteration over all i,j for

if $A[i,j] < kx1$ $B[i,j] = ky1$

else if A[i,j] ≥ kx1 B[i,j] = ky2

ramp

B = (ky2-ky1)*NSLramp(A-kx1) + ky1, or
B = NSLramp(A,kx1,ky1,ky2)

corresponds to an iteration over all i,j for

if A[i,j] < kx1 B[i,j] = ky1
else if A[i,j] ≥ kx1 B[i,j] = A[i,j] - kx1 + ky2

saturation

B = (ky2-ky1)*NSLsaturation((A-kx1)/(kx2-kx1)) + ky1, or
B = NSLsaturation(A,kx1,kx2,ky1,ky2)

corresponds to an iteration over all i,j for

if A[i,j] < kx1 B[i,j] = ky1
else if kx1 ≤ A[i,j] < kx2 B[i,j] = (ky2-ky1)(A[i,j]-kx1)/(kx2-kx1) + ky1
else if A[i,j] ≥ kx2 B[i,j] = ky2

sigmoid

B = (ky2-ky1)*NSLsigmoid(s^2*(3 - 2s)) + ky1, where s = (A-kx1)/(kx2-kx1), or
B = NSLsigmoid(A,kx1,kx2,ky1,ky2)

corresponds to an iteration over all i,j for

if A[i,j] < kx1 B[i,j] = ky1
else if kx1 ≤ A[i,j] < kx2 B[i,j] = s^2(3 - 2s)
where s = (A[i,j]-kx1)/(kx2-kx1)
else if A[i,j] ≥ kx2 B[i,j] = ky2

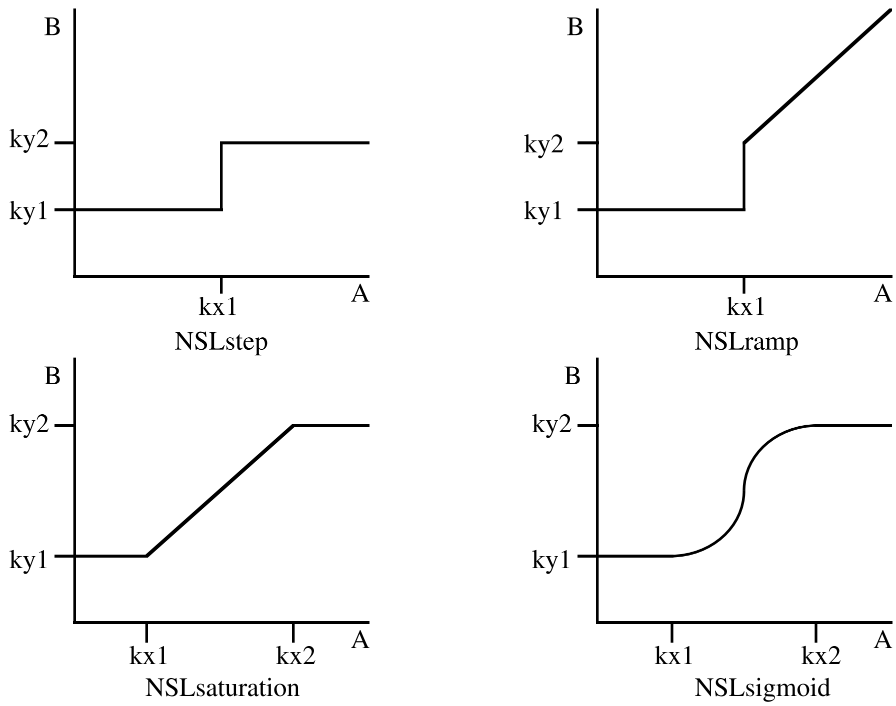


Figure 12
Arbitrary threshold functions

5.6.7. Other

Other functions include the max and min between to layers of the same size:

<u>max</u>

Function returning a layer containing the pointwise maxima, where both function arguments have to be of the same size,

NSLmax(*layer1*,*layer2*)

For example,

B = NSLmax(A1,A2)

corresponds to an iteration over all i,j of

B[i,j] = max(A1[i,j],A2[i,j])

<u>min</u>

Functions returning a layer containing the pointwise minima, where both function arguments have to be of the same size,

NSLmin(*layer1*,*layer2*)

For example,

B = NSLmin(A1,A2)

corresponds to an iteration over all i,j of

B[i,j] = min(A1[i,j],A2[i,j])

5.7. Numerical Methods

NSL provides a library of numerical methods for integration of differential equations. The syntax for a differential equation is as follows:

DIFF.eq(u,tm) = f(u)

which translates to

tm*du/dt = f(u).

NSL will then apply the appropriate numerical method (euler, etc.) at run time, as specified by the user.

For example, in the case of the leaky integrator model, we would write

DIFF.eq(u,tm) = -u+S

for

f(u) = -u + S,

where S is the input to the neuron layer.

While different numerical methods may be used to solve a particular neuron model, the neural network architecture and connection weights should *not* have to change depending on this. For this reason we treat numerical methods as *orthogonal* object classes totally independent from network specification. A numerical method is instantiated only after the neural network has been completely described. Different numerical methods keep on evolving and they may be more appropriate according to the sophistication of the model and the processing power of the computing machine.

5.7.1. Euler

The 'Euler' method, which is the default NSL method, replaces the above differential equation by

$m(t+dt) = (1 - dt/t_m)) m(t) + (dt/t_m) S_m(t)$

where dt is the integration time step (delta).

5.7.2. Interpolation

The 'interpolation' method replaces the above differential equation by

$m(t+dt) = (p) m(t) + (1 - p) S_m(t)$

where $p = \exp(-dt/t)$

5.7.3. Runge-Kutta

The 'Runge-Kutta' numerical method is included as a separate document.

5.8. Learning Methods

Learning methods are included as NSL libraries built on top of the main layer structures. Back-propagation sample networks are included as a separate document.

6. NSL Simulation Commands

In order to process and interact with the model, it is necessary to set up a suitable simulation environment, including the window interface for graphics displays. This section will concentrate on how model parameters are interactively assigned, the simulation is run, and other simulation related commands are used; the window interface will be described in the next section.

Commands may be read from batch files, or interactively entered; and any number of command files can be associated with a single model file. ('//' precedes all comments.)

The most important commands will be described in these sections, while a summary of all simulation commands is included at the end of the manual.

There are three important commands worth mentioning here, before getting into any of the other ones:

command	description
help	on-line help.
quit or **exit**	exit the simulator.
load *file*	load a '.nsl' file to be interpreted.

help

Provides extensive on-line information, where information is available as a general command listing,

help

or per command, giving more detailed information on each command,

help *command*

exit/quit

Provides the way of exiting the simulator[24].

---

[24] 'control-C' ('^C') is used to interrupt simulation runs.

load

The load command is used to read in any '.nsl' files containing simulation commands.

For example, a file called 'didday.nsl' may be loaded as follows (the '.nsl' suffix may be omitted):

load didday

6.1. Model Object Classes

Interaction with NSL's model language objects is done via the simulation environment.

In these sections we will provide a sample simulation file for the previously described 'DIDDAY' model (stored in 'didday.c').

6.1.1. network

The following are commands affecting initial network setup.

set

To set up the desired simulation model include the following line,

set network *name*

where *name* matches the one in 'NETWORK(*name*)' for the corresponding model.

Since this file is interpreted, the order of commands is very important (the 'set network' command should precede all other commands in the file).

The simulation file for the previously described 'DIDDAY' model should then begin with,

set network DIDDAY

status

The interpreter provides status information on different aspects of the system. If several models have been compiled together, the 'status' command can give a listing of them, showing their names and indices, and information on which one is currently enabled[25].

In what follows 'nsl>' is the system's prompt,

nsl> status system

A sample response would look as follows,

```
// total linked models: 11
// 1: TUTORIAL  (disabled)
// 2: TECTUM_88 (disabled)
// 3: TECTUM_11 (disabled)
// 4: STRETCH   (disabled)
```

[25] Currently, NSL 2.1 lets only one model be run at any time; in the future several model could be run together.

```
// 5: RETINA    (disabled)
// 6: LOOM21    (disabled)
// 7: LIMULUS   (disabled)
// 8: DIDDAY    (enabled)
// 9: DEV       (disabled)
// 10: CUE_INTERACTION   (disabled)
// 11: BEKESY   (disabled)
```

6.1.2. layer

The following are commands affecting network layers.

set

Layers can be assigned data interactively at any time during the simulation. Data may also be assigned through the graphics interface, as will be explained in the window interface section.

To assign data,

set data_value *name value [value, ...]*

where *name* is the name of an existing layer. Immediately following is a list of values (separated by spaces), with the list size matching the number of elements in the layer. In the case of a data layer only one value would be entered, in the case of a vector, a list of values matching the number of elements in the vector would be entered, and in the case of a matrix, a list of values, row wise, and starting with row '0', followed by row '1', and son on would be entered, until all the elements have been assigned.[26]

For example, for the didday model one can assign the following values to its variables,

```
set data_value tu 1.0
set data_value tv 1.0
set data_value k 0.1
set data_value w1 1.0
set data_value w2 1.0
set data_value h1 0.1
set data_value h2 0.5

set data_value S
0 0 0 0 1.0 0 .5 0 0 0
```

assigning 1.0 to S[4] and .5 to S[6].

Layers can be assigned a single data value for the complete layer by using,

set all_data_value *name value*

For example

```
set all_data_value S 1
```

[26] NSL 2.2 will include assignment operators at the interpreter level, so the same operation would be written as 'name = value' for data, 'name = { value, ... }' for vectors, and 'name = {{value, ...},...}' for matrices.

is equivalent to

set data_value S
1 1 1 1 1 1 1 1 1 1

Layers can be assigned single element data values by

set elem_i *index*
set elem_j *index*
set elem_data_value *name value*

elem_i specifies the vector element index or the matrix row index, while elem_j specifies the matrix column index.[27]

For example

set all_data_value S 0
set elem_i 4
set elem_data_value S 1
set elem_i 6
set elem_data_value S 0.5

is equivalent to

set data_value S
0 0 0 0 1.0 0 .5 0 0 0

status

To show the data values,

status data_value *name*

For example, if we want to show the values of layer S then,

nsl> status data_value S

The system response would be as follows,

set data_value S
0 0 0 0 1 0 0.5 0 0 0

6.1.3. module

The following are commands affecting network modules.

Initially all modules are enabled for processing, yet the user has the option to disable or enable individual modules from being processed.

enable

To enable module processing,

[27] NSL 2.2 will have single element assignment in the form of v[i]=val for vectors, and m[i][j]=val for matrices.

enable module *name*

where *name* is optional, and should match an existing module name.

disable

To disable module processing,

disable module *name*

where *name* is optional, and should match an existing module name.

6.2. Simulation Processing

The following are commands affecting network simulation processing.

set

To set the simulation time step or delta,

set delta *value*

value specifies the simulation delta time step

To set the simulation end time,

set end_time *value*

value specifies the simulation end time, which divided by 'delta', gives the total number of steps the simulation is going to run.

For example, for the didday model we have,

set delta 0.1
set end_time 20.0

which gives a total of 200 (20.0/0.1) simulation steps.

processing

To process a simulation there are the following commands,

command	description
init	process 'INIT_MODULE's and 'MODULE's for t = 0.
run [*n*]	process all modules from t=0 (similar to 'init'), and all 'RUN_MODULE's and 'MODULE's for t>0 until t=*n* or t = *end_time*, if no *n* is specified.
step [*n*]	process modules *n* steps or step for one iteration, if no *n* is specified.

cont [*n*] continue module processing from current t until t=*n* or t = *end_time*, if no *n* is specified.

'Control-C' ('^C') serves as processing interrupt.

6.3. Data files

Data produced by a simulation may be stored into external files. The data stored in these files can be used as input to other simulations or simply as a means of analyzing the simulation output in numerical detail.

Besides the 'load' command, the user has available commands for opening and closing files, either for input or output.

<u>output data file</u>

In order to create a a file for output the user may execute the following command:

open OUTPUT *file_name*

where file_name is the name of the file to be created. If the command is executed without a file name, then the current model name is used as file name. If the 'file_auto_name' flag is 'ON' ('set file_auto_name ON') then NSL will append an increasing numerical index suffix to every newly opened data file.

For example, if the currently simulated model is called 'DIDDAY' then a call to 'open' without arguments will create an output data file named 'DIDDAY.nsl.0'; the next call to 'open' without arguments will create an output data file named 'DIDDAY.nsl.1'; and so on.

The information stored into these data files is basically all the data values computed during the simulation. NSL will store data from all those layers currently 'enabled', where enable layer refers to which layer is enabled for file interaction. To enable a layer specified by *name* to be written, do,

file_enable layer *name*

(Layers may be disabled by using the 'file_disable' command in a similar way. The 'ALL' reserved string serves as name modifier specifying all layers in the network).

To close file execute the following command:

close *file_name*

where file_name is the name of the file to be close. If the command is executed without a file name, then the current open file is closed.

<u>input data file</u>

In order to create a a file for input the user may execute the following command:

open INPUT *file_name*

where file_name is the name of the file to be created. If the command is executed without a file name, then the current model name is used as file name.

The information to be read is basically depends on what's stored in the file and what layers are currently 'enabled'. NSL will read data to all those layers currently 'enabled', where enable layer refers to which layer is enabled for file interaction. To enable a layer specified by *name* to be read, do,

file_enable layer *name*

The only other aspect to be considered is that if there are modules which originally computed those layers, they should be 'disabled' from processing.

To close the file execute the following command:

close *file_name*

where file_name is the name of the file to be close. If the command is executed without a file name, then the current open file is closed.

6.4. Simulation Management

One important aspect of the system is the ability to record automatically any interaction done by the user, so it can be reproduced at any time in the future.[28]

6.5. Numerical Methods

NSL current release provides two different numerical methods for integration.[29]

set

To choose the appropriate numerical method NSL provides with the following command,

set integration *type*

where type may be EULER or INTERPOLATION.

For example, to use the didday model with the EULER method, which is also the default one, write

set integration EULER

6.6. Other

Other commands are included in section 10.

[28] Simulation management is under current development and will be available with the next system release.

[29] Future releases will extend upon the number and types of numerical methods available. Yet, the user may include new numerical methods with any NSL model.

7. NSL Window Interface

The window interface comprises a set of display object classes and graphics libraries. In order to be able to efficiently interact with the window interface the user should become familiar with the different types of display objects available.

In the following sections we will explain how to design such a graphics interface by either storing all commands on a '.nsl' file or by interactively typing into the nsl interpreter.

Figure 12 shows the window interface created for the didday model, at the end of the simulation.

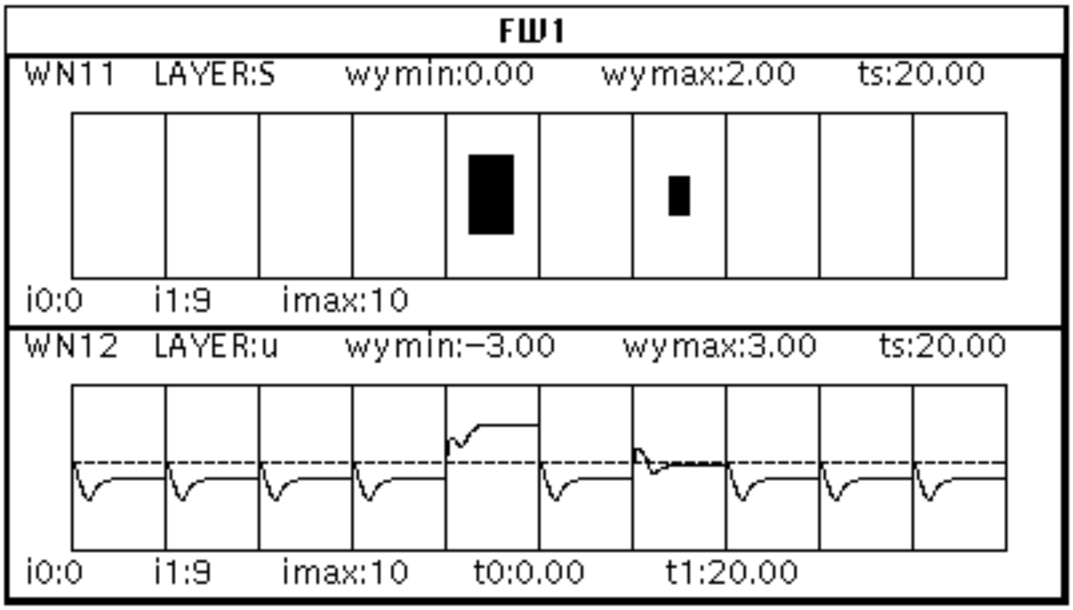


Figure 12
The window interface for the didday simulation model

The code creating this frame is as follows:

```
set frame_name FW1
set frame_X0 200
set frame_Y0 100
set frame_width 500
set frame_height 335
set frame_rows 2
create display_frame

set window_name WN11
set window_layer S
set window_graph area_level_graph
set window_wymin 0
set window_wymax 2
create display_window

set window_name WN12
set window_layer u
set window_graph temporal_graph
set window_wymin -3
```

set window_wymax 3
set window_t1 20
create display_window

The next sections will describe the different interface object classes and describe their manipulation via the command interpreter. A summary of all available window interface commands is given in section 11.

7.1. Display Object Classes

There are five types of display object classes: **window_interface**, **display_frame**, **display_window**, **display_canvas**, and **display_panel**.

7.1.1. window_interface

The **window_interface** is a 'non-visual' object type, instantiated automatically by the system's graphical environment. Figure 13 shows the window interface structure, composed of a list of display frames. Each display frame is made up of a list of display windows, which themselves are composed of a display canvas where graphs are drawn, and a data panel where the user can interface with the window control attributes.

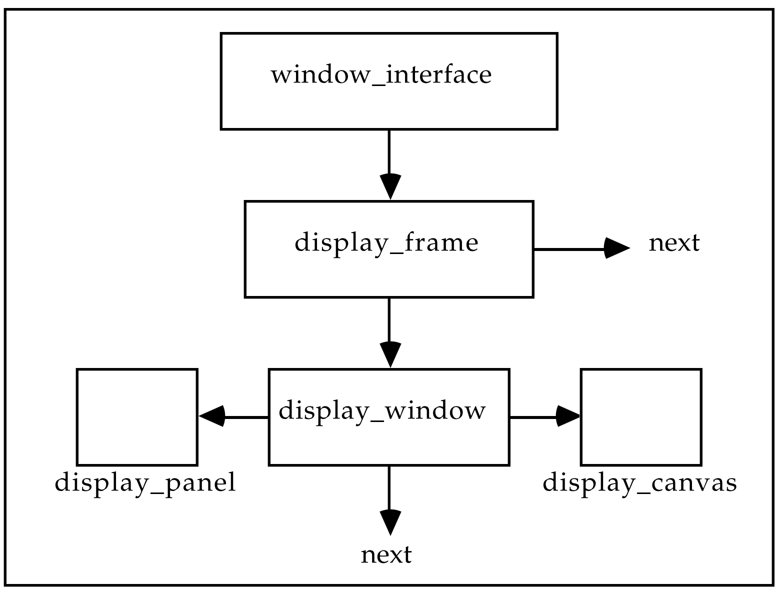


Figure 13
NSL display classes.

In order to initialize a window interface the following call should be entered (if the call doesn't appear, new frames will be added to any currently existing ones),

create window_interface

7.1.2. display_frame

A **display_frame** is a 'visual' display unit manipulated as a single entity on the screen which can be moved, resized, and printed. A display frame can be physically composed of any number of display windows as will be explained in the next section.

For example, the display frame on Figure 12, FW1, is created by the following commands:

```
set frame_name FW1
set frame_X0 200
set frame_Y0 100
set frame_width 500
set frame_height 335
set frame_rows 2
create display_frame
```

<u>set</u>

The set command assigns values to the different environment variables:

- **frame_name** specifies the name to be used for the display frame.
- **frame_width** specifies the width of the display frame.
- **frame_height** specifies the height of the display frame.
- **frame_X0** specifies the left pixel location of the display frame, where the coordinate is zero on the left of the screen and increments towards the right until the monitor's maximum width is reached.
- **frame_Y0** specifies the top pixel location of the display frame, where the coordinate is zero on the top of the screen and increments towards the bottom until the monitor's maximum height is reached.
- **frame_rows** specifies the number of rows of display windows to be included inside the display frame (graphs are assigned row-wise).
- **frame_cols** specifies the number of columns of display windows to be included inside the display frame.

<u>create</u>

The create command creates a new display frame according to the specified setting.

<u>update</u>

The update command updates an existing display frame according to the specified settings.

7.1.3. display_window

A **display_window** is a 'non-visual' object containing two types of of 'visual' objects: a **display_canvas**, where all graphics takes place, and a **display_panel**, a special window used for controlling the display canvas. The display window also contains an internal database storing all the window parameters.

In the the sample frame of Figure 12, two display windows are created with the following commands:

```
set window_name WN11
```

```
set window_layer S
set window_graph area_level_graph
set window_wymin 0
set window_wymax 2
create display_window

set window_name WN12
set window_layer u
set window_graph temporal_graph
set window_wymin -3
set window_wymax 3
set window_t1 20
create display_window
```

The first window, WN11, will display layer S as an 'area_level_graph' with a y axis value range between 0 and 2. The second window, WN12, will display layer u, as a 'temporal_graph' with y axis value range between -3 and 3. The time scale for the temporal graph is from 0 to 20,

Each window automatically opens a display panel (control panel) on an independent screen frame[30].

set

The set command assigns values to the different environment variables:

- **window_name** specifies the name to be used for the display window.
- **window_width** specifies the width of the display canvas contained in the display window.
- **window_height** specifies the height of the display canvas contained in the display window.
- **window_X0** specifies the left pixel location of the display canvas contained in the display window, where the coordinate is zero on the left of the frame and increments towards the right until the frame's maximum width is reached.
- **window_Y0** specifies the top pixel location of the display canvas contained in the display window, where the coordinate is zero on the top of the frame and increments towards the bottom until the frame's maximum height is reached.
- **window_layer** specifies the particular layer to be displayed.
- **window_graph** specifies the particular type of graph to utilize for displaying.

create

The create command creates a new display window according to the current settings.

update

The update command update an existing display window according to the current settings.

7.1.4. display_canvas

A **display_canvas** is the 'visual' display object where graphics are actually drawn. There are different kinds of graphics items which may be drawn on a canvas: graphs, grids, lines, and text, as shown by the class structure tree in Figure 14.

[30] When we talk about a frame, we talk about an X windows screen frame; while a display frame refers to NSL's window interface display entity.

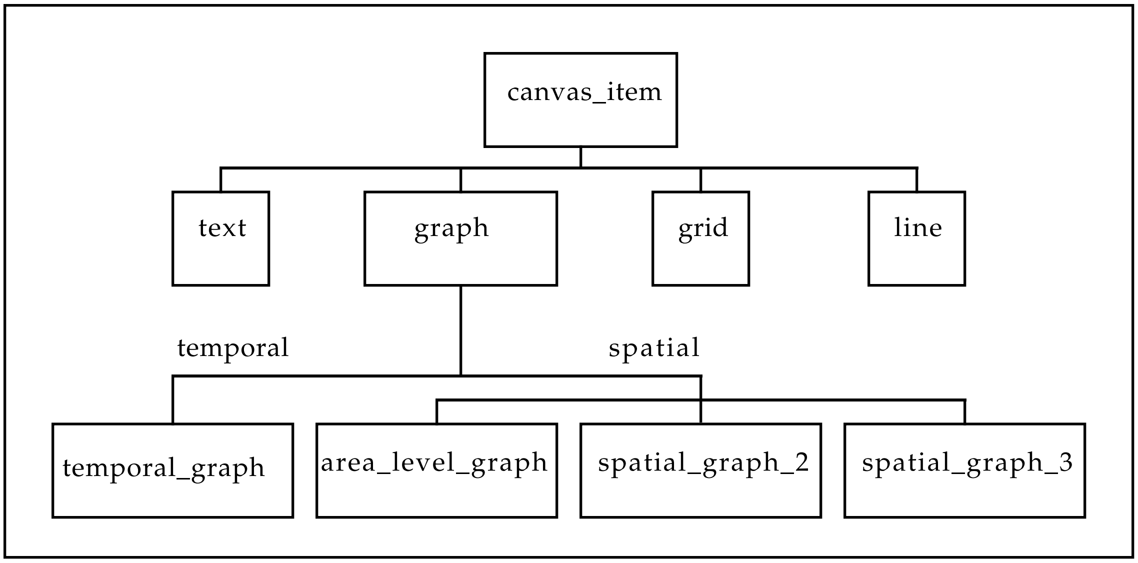


Figure 14
NSL display canvas item classes.

set

The set command assigns canvas related parameters:

- **window_graph** specifies the type of graph to be used for the display window. Graphs may be either temporal or spatial, and there are three different types of spatial graphs.

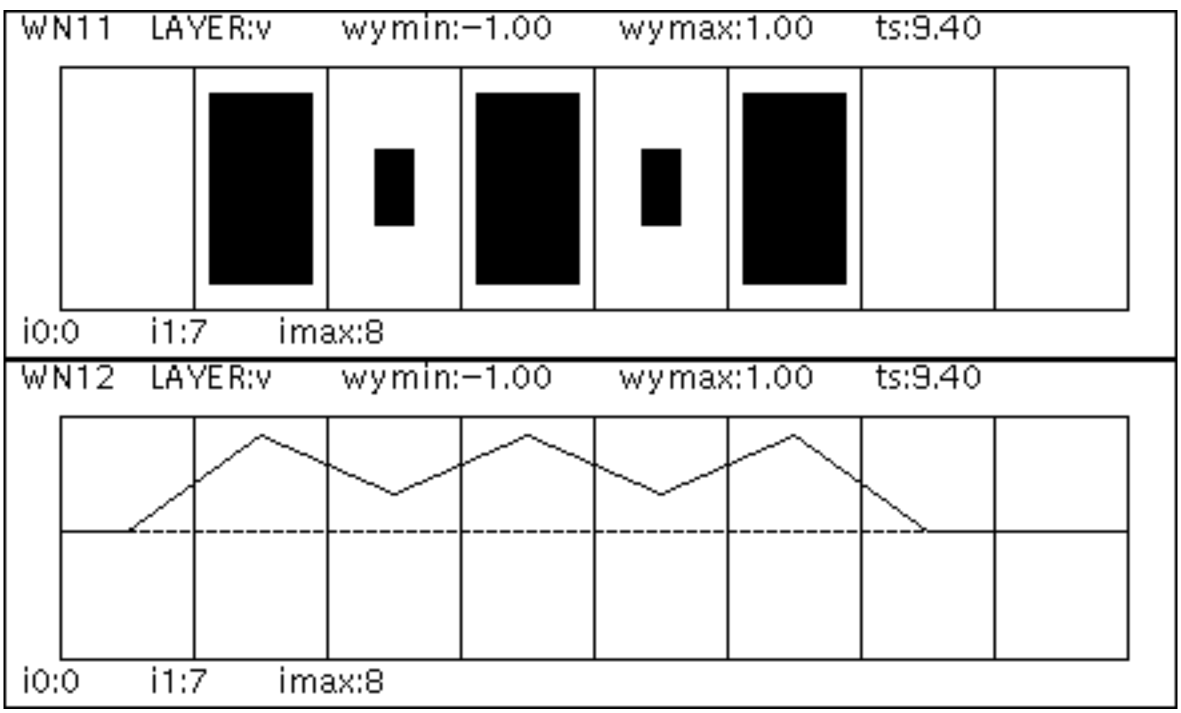


Figure 15

vector area_level_graph vs. spatial_graph_2

Spatial graphs:

- **area_level_graph** displays a layer as levels of rectangular area representing the activity of the different elements in the layer[31].
- **spatial_graph_2** displays a layer as a two dimensional graph (only vector layers should be displayed with this option). Figure 15 shows an area_level_graph and a spatial_graph_2 for a similar vector layer.
- **spatial_graph_3** displays a layer as a three dimensional graph (only matrix layers should be displayed with this option). Figure 16 shows an area_level_graph and a spatial_graph_3 for a similar matrix layer.

Temporal Graphs (a vector temporal graph is shown in Figure 12 inside window WN12):

- **temporal_graph** displays temporal layer information. Every layer element is displayed as an independent temporal graph, where the complete graph will contain as many rows and columns as there are elements in that layer (i0<->i1, j0<->j1), each graph being a temporal display (t0<->t1) of the respective layer's element.[32]

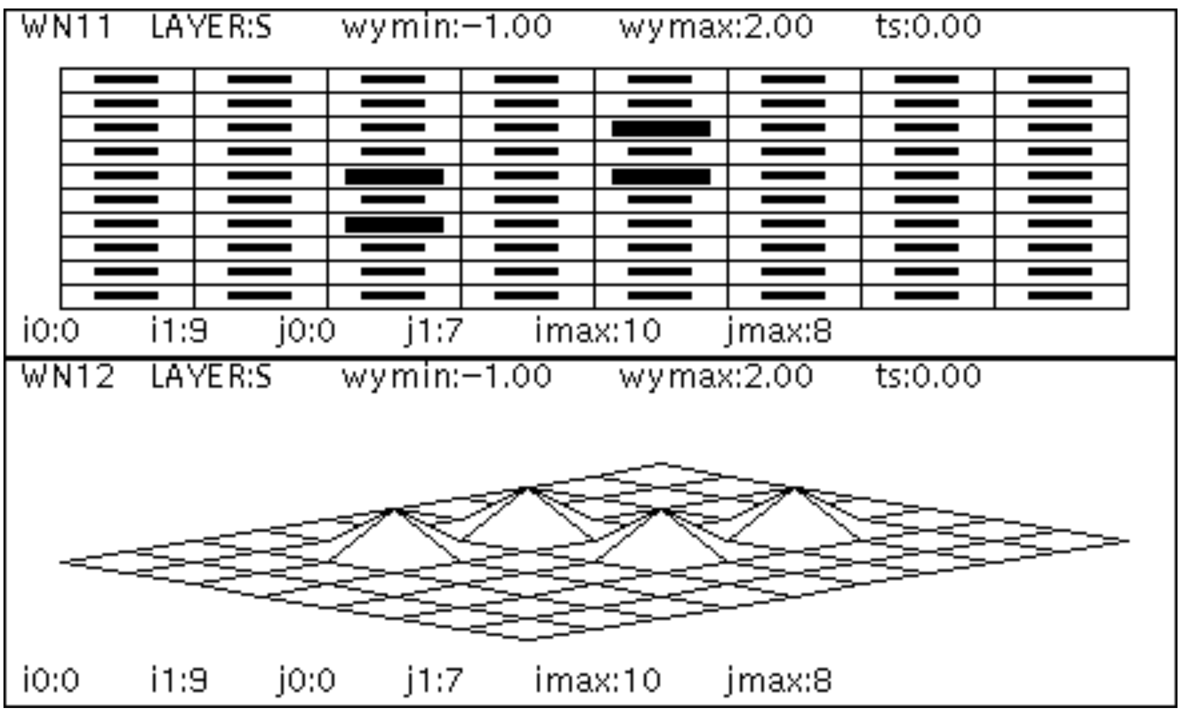


Figure 16
matrix area_level_graph vs. spatial_graph_3

---

31 An *area_level_graph* may be used both for displaying and reading input as will be described later in more detail.

32 For efficiency considerations, the model data produced is not automatically stored by the system. For this reason, temporal graphs can only be fully displayed while the simulation is being run. At the end of a simulation all layer temporal data is lost.

7.1.5. display_panel

The **display_panel** contains varied window control items. There are four classes of panel items: *text*, *menus*. *buttons*, and *messages*, The panel class structure is shown in Figure 17.

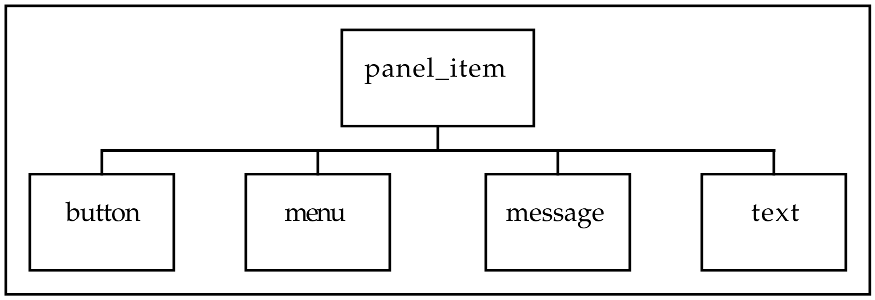
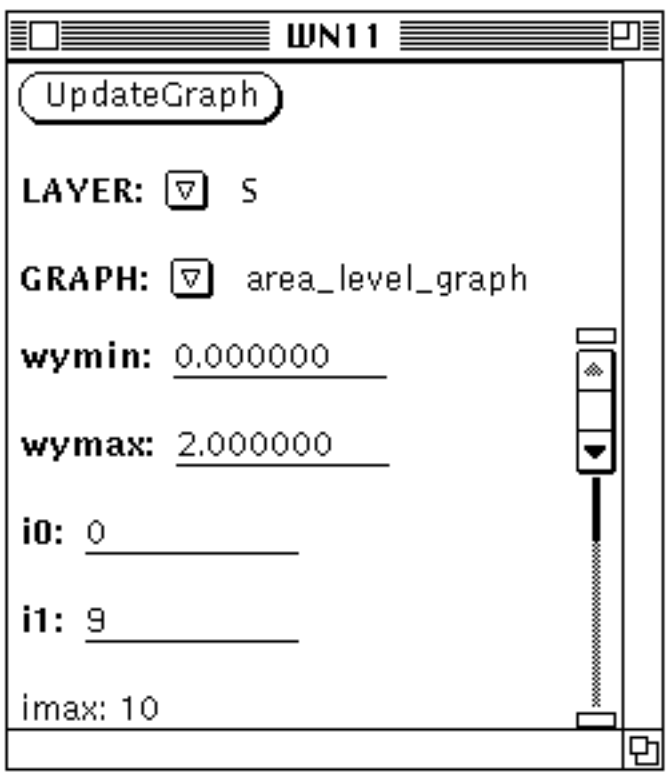


Figure 17
NSL display panel item classes.

A typical control panel, as the one controlling window WN11 (in Figure 12) is shown in Figure 18. The different entries are control parameters which can also be set from the window interface interpreter[33].



[33] These parameters are preceded by 'window_' when called from the commands interpreter. The control panel functions and parameters are a subset from those available from the commands interpreter.

Figure 18
NSL display window control panel

To disable the control panel from the command interpreter do,

disable display_panel *name*

and to enable it do,

enable display_panel *name*

where *name* is the name of the window to which the control panel belongs.

7.2. Window Mouse Interaction

Until now we have described the window interface display objects, and how they can be manipulated from the command interpreter. In these sections we explain how to interact with the window interface in a graphical way.

7.2.1. Drawing Frame

Every frame includes a *frame menu* along its boundary. The Frame Menu contains the regular X windows menu (e.g. for moving, resizing, and closing, the frame).

<u>resize</u>

The graphs in the different displays may be automatically resized when the frame itself is resized. For this purpose, ff 'set frame_auto_resize' is 'ON' (which is the default), then every time a frame is resized, all the windows will be automatically resized according to the number of rows and columns that have been specified for the particular frame. If the option is 'OFF' no internal window resizing will take place, even though the frame will change its size.

7.2.2. Drawing Canvas

The following are different types of interactive commands which may be activated through the help of the mouse while inside a display canvas.

<u>Obtaining a Window Dump</u>

A 'Window Dump', a display picture snapshot, may be obtained from any display frame at any time. A window dump can be saved into a raster file ('ras'), a postscript file ('ps'), or directly be sent to a printer. The menu containing these options is found by moving the mouse over the desired frame, and pressing the mouse's right button which will show a menu with three entries: <u>WinDump</u>, <u>UpdateGraph</u>, and <u>ControlPanel</u>.

Drag the mouse to the right of the <u>WinDump</u> menu entry, while still holding down the right button, a new menu will appear with the following choices:

- <u>WindowDump -> Printer</u> - Send the window dump directly to the printer without having to store it first in a file.

- WindowDump -> RasterFile - Save the window dump into a rasterfile ('ras'), specified by the *dump_file* variable. 'raster' format is the local workstation picture format.
- WindowDump -> PostScript - Save the window dump into a postscript file ('ps'), specified by the *dump_file* variable. 'postscript' is the standard printer language.

where *dump_file* is a file name automatically generated by NSL, composed of a number suffix, an inffix of 'ras' or 'ps' depending on what type of file it represents, and a preffix containing the model's name.

For example the first time a raster ('ras') is created from the didday model, the following name will be generated,

didday.ras.1

The other two entries in the menu are as follows,

- The UpdateGraph entry is used for updating the window with any new information.

- The ControlPanel entry is used for recalling the Control Panel window if not present.

Interactive input assignment

The user may interactively modify the values ('data_value') from any layer in the network. This can be done with the help of the two mouse buttons when pressed over any window displaying a layer as an 'area_level_graph' type of graph. By clicking the left button over a layer element, the value specified by the window control panel's 'val_in' entry is assigned; while by clicking the middle button, the layer's element value is reset to 0.0. To assign a value other than the one currently shown in 'val_in', simply modify the text entry to the right of 'val_in' and the new value will be used instead. To find out the values, or the element's index, of the different layer elements, simply drag the mouse, without clicking any button, over the different units, and their respective values with their respective unit locations will be shown on 'val_out' (this entry value cannot be modified by the user).

7.2.3. Control Panel

The control panel, as shown in Figure 14, contains two sections, a 'static' top section, and a 'dynamic' bottom section.

The top section contains the following entries:

- The UpdateGraph button used for updating the corresponding display window with new information.
- The Layer menu from which a layer is selected.
- The Graph menu from which a graph is selected.

The lower section is scrollable and contains different entries depending on the Layer and Graph type currently selected.

8. NSL Input Facility

NSL 2.1 includes a facility for dealing with a special type of INPUT_LAYER. An input layer serves as the 'world' where many different objects or stimuli may be set, with constrains on their location and time on which they are valid. While this functionality may be obtained using regular NSL language constructs, such a task is greatly simplified by the use of NSL's input facility. In the next sections we explain how to interact with this module.

For example, Figure 19 shows an 'area_level_graph' of an input layer ('INPUT_MAT') made of 32x32 elements, containing an object of size 8x4.

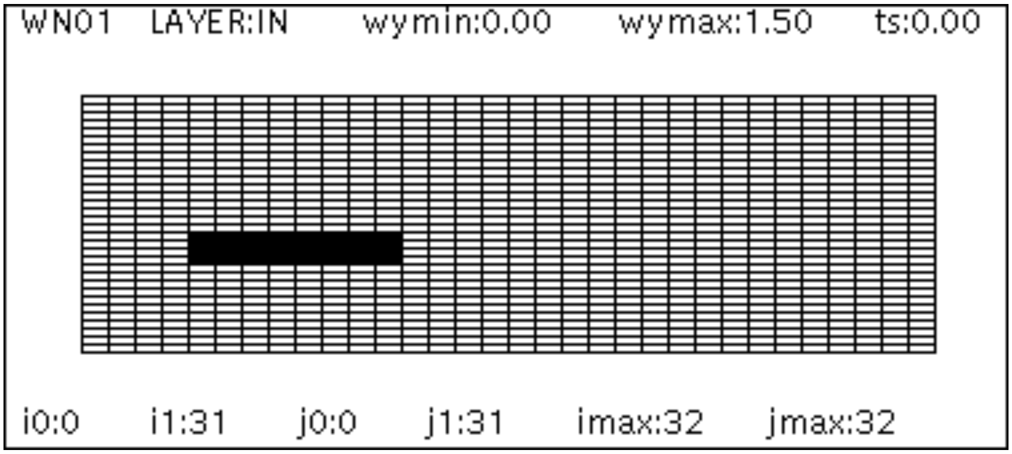


Figure 19
NSL input stim objects.

Figure 20 shows a 'temporal graph' of an input layer ('INPUT_DAT') made of a single element, containing an object appearing at two different time intervals.

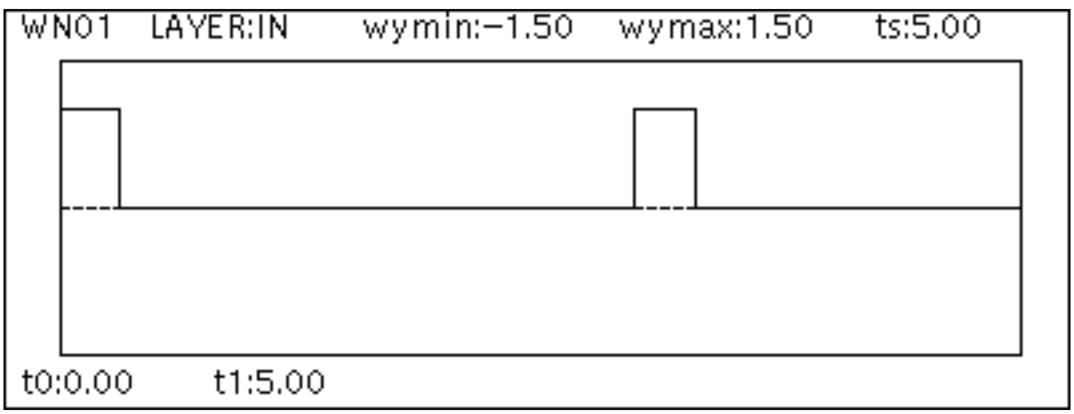


Figure 20
NSL input stim objects.

These and many other object specifications may be done with the help of NSL's input facility, as will be shown in the following sections.

8.1. Model Language

In this section we will describe the input layer class declarations and expressions included in NSL model language ('.c' file).

8.1.1. input_layer

The **input_layer** classes are derived from the basic layer classes, data, vector, and matrix, giving rise to: **INPUT_DAT**, **INPUT_VEC**, and **INPUT_MAT** classes. These three classes vary according to the number of dimensions they contain. Figure 21 shows the layer class hierarchy.

Since the input_layer is derived from the regular layer, it includes all the basic functionality of the layer, extended with its own private functionality. Thus, for example, an input layer may be added with regular layers, and so on. What's characteristic to the input layer is its ability to be map external objects into itself. These objects, or stimuli, have special characteristics, such as movement, and may be assigned time intervals on when to appear. This facility thus provides a means for assigning external object specifications into a layer which at any moment may be incorporated into regular layer expressions.

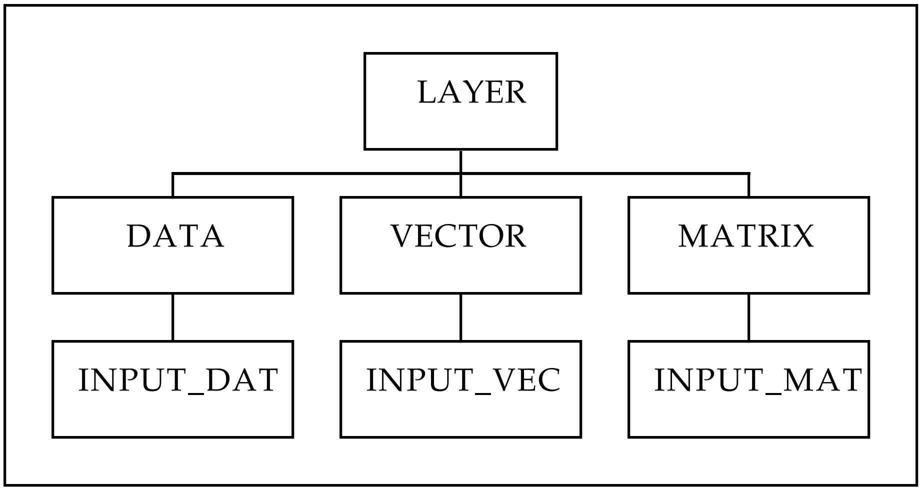


Figure 21
NSL input layer classes.

The following are the different classes of input layers.

Data layer:

INPUT_DAT(*layer_name*)

e.g INPUT_DAT(S); // a data layer

Vector layer:

INPUT_VEC(*layer_name*,*size*)

e.g. INPUT_VEC(INP,10); // a 10 unit vector layer

Matrix layer:

INPUT_MAT(*layer_name*,*rows*,*columns*)

e.g INPUT_MAT(A,10,8); // a 10x8 layer

where *layer_name* specifies the name of the layer, *size* specifies the length of the vector, *rows* specifies the number of rows of a matrix layer, and *columns* represent the number of columns of a matrix layer.

8.1.2. processing

Processing of any stimuli specifications (done from the '.nsl' file) is achieved by including the following line inside a RUN_MODULE ,

input_layer.run();

where *layer_name* specifies the name of the layer, and 'run' is the function which will process any existing stimuli specification.

8.2. Simulation Commands

Once an input layer has been declared in the model file, it is necessary to give the input object specifications in the command file. For example, the object shown in Figure 19 was created with the following script:

```
set input_layer IN
set input_xz 0
set input_yz 16
set input_dx 1
set input_dy 1
reset stim_list
set stim_val 1.5
set stim_x0 4
set stim_y0 0
set stim_dx 8
set stim_dy 4
create block_stim
```

The object shown in Figure 20 was created with the following script:

```
set input_layer IN
reset stim_list
set stim_val 1.0
create block_stim
set stim_t0 0.0
set stim_t1 0.3
create time_stim
set stim_t0 3.0
set stim_t1 3.3
create time_stim
```

These lines will be explained in the next sections.

8.2.1. input_layer

The first step in describing an input object is to set the appropriate input layer, e.g.

```
set input_layer IN
```

then, the mapping characteristics between the layer and the input stimulus coordinates have to be given.

For example,

set input_xz 0
set input_yz 16
set input_dx 1
set input_dy 1

specify the origin of the coordinate system to lie over IN[0,16] (element 0,16 of layer IN), while the distance among adjacent layer elements is equivalent to '1' either in x or y. Figure 22 shows the general schematics of an input layer and input stimulus specification characteristics.

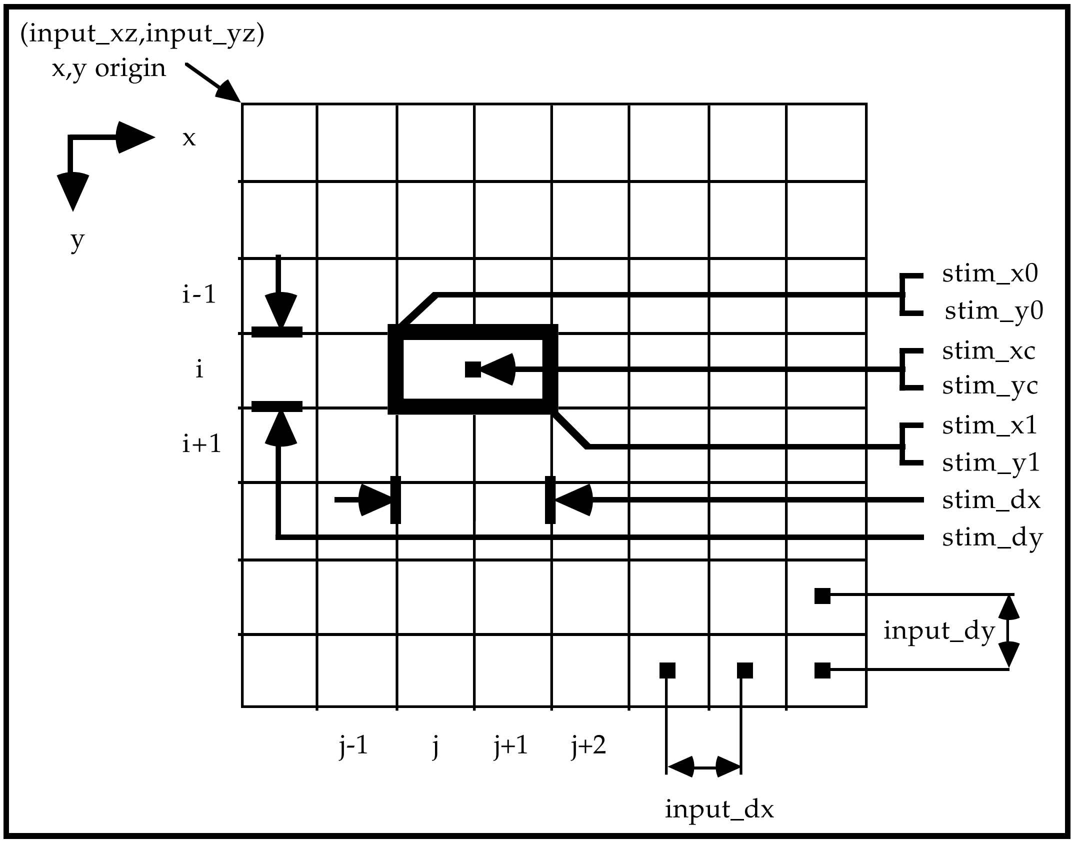


Figure 22
NSL input stim objects.

The summary of input layer specifications is given in section 10 (the input stimulus specifications will be explained in the next section).

8.2.2. input_stim

The following is the input object specification as shown in Figure 19,

reset stim_list

which resets the stimuli list to a new one,

set stim_val 1.5

which assigns the value of 1.5 to the input layer elements corresponding to the current input object location,

```
set stim_x0 4
set stim_y0 0
```

which sets the input object upper left corner location to (4,0) on the input layer coordinate system, and which maps to layer elements [4,16],

```
set stim_dx 8
set stim_dy 4
```

which sets the input object size to 8x4, and

```
create block_stim
```

which actually creates the specified object.

The object shown in figure 20 is specified by,

```
reset stim_list
```

which resets the stimuli list to a new one,

```
set stim_val 1.0
```

which assigns the value of 1.0, which is really the default value,

```
create block_stim
```

which creates the specified object (since this is a single element layer, there is no need to specify stimulus size or location),

```
set stim_t0 0.0
set stim_t1 0.3
create time_stim
```

this creates a time interval between 0.0 and 0.3 (simulated time),

```
set stim_t0 3.0
set stim_t1 3.3
create time_stim
```

while this creates a time interval between 3.0 and 3.3. The temporal output is shown in Figure 20.

The location of the stimulus may be specified either by using its corner or its center as reference position, where the relation among them in a any rectangular object is as follows,

```
stim_xc = stim_x0 + (stim_dx)/2
stim_yc = stim_y0 + (stim_dy)/2
```

Objects may also be specified with velocity,

```
stim_vx
```

stim_vy

where the object's velocity is calculated as distance per simulated time, or in other words, the current stimulus position at any given time is computed by (in the case of the corner location) :

tx0 = ts*stim_vx + stim_x0
ty0 = ts*stim_vy + stim_y0

where 'ts' is the current simulated time.

The summary of input stimulus specifications is given in section 10.

9. Advanced NSL Programming

To write more developed code in NSL, the user is encouraged to become familiar with the C++ language. The current release compiles NSL an extended version of C++, so user written C++ code should compile in a straight forward manner, and may be integrated with any NSL code. In these sections we will describe NSL classes and their most important public methods. Calls to these methods can be done from the model language, by writing any C++ expressions inside the different processing modules, or by writing totally independent C++ routines which may be called from an expression.

Many files may be linked together, the only requirement is that each contain an '# include "nsl_include.h"' line, which contains all the header and macro definitions necessary for linking NSL code.

9.1. Model Language

Following C++ conventions, an object's public method is called as follows,

object.method();

where *object* is the name of the object and *method* is the desired 'member' function. (If the object is referenced as a pointer to it, then the call would be: '*object*->*method*()'.)

9.1.1. network

The macro declaration of 'NETWORK(*name*)' translates to a C++ declaration of
nsl_network *name*("*name*")
where **nsl_network** is the name of the object class, and *name* is the name of the instantiation variable. "*name*" is used the name to be stored internally for linking the compiled code with the run time interpreted environment.

The network class definition is as follows:

```
class nsl_network
{
    char name[NAME_SIZE];          // model name

    float DELTA;                   // time step
    float CURRENT_TIME;            // current time
    float END_TIME;                // end time

    int layer_total;               // total number of layers
    int module_total;              // total number of modules
```

```
public:
    nsl_network(char*);
    ~nsl_network();

    float  get_delta();              // get delta time step
    float  get_time();               // get current simulation time
    float  get_end_time();           // get simulation end time

    void  set_delta(float);          // set time step
    void  set_time(float);           // set current simulation time
    void  set_end_time(float);       // set simulation end time

    void  reset_time();              // reset time to 0

    int        get_layer_total();    // total number of layers
    nsl_layer*    get_layer(char*);  // name search
    nsl_layer*    get_layer(int); // index search

    int  get_module_total();         // total number of modules

    void  print_status();            // print model information
};
```

For example, to get the value for the delta do (and having a network called 'DIDDAY'),

```
float dt = DIDDAY.get_delta();
```

9.1.2. modules

The macro declaration of 'MODULE(*name*)' translates to a C++ declaration of

nsl_module *name*("*name*")

where **nsl_module** is the name of the object class, and *name* is the name of the instantiation variable. "*name*" is used the name to be stored internally for linking the compiled code with the run time interpreted environment.

The module class definition is as follows:

```
class nsl_module
{
    char name[NAME_SIZE];            // module name
    int index;                       // module index number
public:
    nsl_module();
    ~nsl_module();

    char*    get_name();             // return module name
    int      get_index();            // return module index number
};
```

9.1.3. layers

The layer class, which is the super-class for the derived data, vector, and matrix classes, definition is as follows:

```
class nsl_layer
{
    char name[NAME_SIZE];
    int index;
    int layer_type; // 1 - nsl_data, 2 - nsl_vector, 3 - nsl_matrix
    public:
    nsl_layer(int);
    nsl_layer(char*,int);

    void    set_name(char*) ;
    void    set_index(int);
    void    set_layer_type(int);

    char*   get_name();
    int     get_index();
    int     get_layer_type();
};
```

9.1.3.1. data

The macro declaration 'DATA(*name*)' translates to

nsl_data *name*("*name*")

where **nsl_data** is the name of the object class, and *name* is the name of the instantiation variable. "*name*" is used for linking the compiled code with the run time interpreted environment.

The data structure is as follows:

```
class nsl_data : public nsl_layer
{
    num_type v;
public:
    nsl_data();
    nsl_data(nsl_data&);
    nsl_data(nsl_model&,char*);
    nsl_data(nsl_model*,char*);
    nsl_data(char*);
    ~nsl_data();

    num_type&     get_data();

    num_type&     elem();

    void    operator=(nsl_data&);
    void    operator=(num_type);

    void    print();
    void    print_status();
};
```

9.1.3.2. vector

The macro declaration of 'VECTOR(*name,size*)' translates to

nsl_vector *name*("*name*",*size*)

where **nsl_vector** is the name of the object class, and *name* is the name of the instantiation variable. "*name*" is used for linking the compiled code with the run time interpreted environment. *size* is simply the size of the vector which also gets passed as argument to the 'nsl_vector' class constructor.

The vector structure is as follows:

```
class nsl_vector : public nsl_layer
{
    num_type* v;
    int i0,i1;
    int size;
public:
    nsl_vector(int);
    nsl_vector(nsl_vector&);
    nsl_vector(nsl_model*,char*,int);
    nsl_vector(nsl_model&,char*,int);
    nsl_vector(char*,int);
    ~nsl_vector();

    int         get_i0();
    int         get_i1();
    int         get_size();

    num_type*   get_vector();
    num_type&   elem(int);

    void        operator=(nsl_vector&);
    void        operator=(nsl_data&);
    void        operator=(num_type);

    num_type    max();
    num_type    min();

    num_type    sum();

    nsl_vector  get_sector(int,int);
    nsl_vector  put_sector(nsl_vector&,int);

    void        print();
    void        print(int,int);
    void        print_status();
};
```

9.1.3.3. matrix

The macro declaration of MATRIX(*name,rows,columns*) translates to

nsl_matrix *name*("name",*rows,columns*)

where **nsl_matrix** is the name of the object class, and *name* is the name of the instantiation variable. "*name*" is used for linking the compiled code with the run time interpreted environment. *rows* and *columns* are simply the size of the matrix which also gets passed as arguments to the 'nsl_matrix' class constructor.

The matrix structure is as follows:

```
class nsl_matrix : public nsl_layer
```

```
{
    num_type** v;
    int i0,i1,j0,j1;
    int rows,cols;
public:
    nsl_matrix(int,int);
    nsl_matrix(nsl_matrix&);
    nsl_matrix(nsl_model*,char*,int,int);
    nsl_matrix(nsl_model&,char*,int,int);
    nsl_matrix(char*,int,int);
    ~nsl_matrix();

    int     get_i0();
    int     get_i1();
    int     get_j0();
    int     get_j1();
    int     get_rows();
    int     get_cols();

    num_type**    get_matrix();
    num_type&     elem(int,int);

    void    operator=(nsl_matrix&);
    void    operator=(nsl_data&);
    void    operator=(num_type);

    num_type      max();
    num_type      min();

    num_type      sum();

    nsl_matrix    get_sector(int,int,int,int);
    nsl_matrix    put_sector(nsl_matrix&,int,int);

    void    print();                // print layer data
    void    print(int,int,int,int);         // print layer data sector
    void    print_status();             // print layer information
};
```

9.1.4. input_layers

There are three types of input layers, derived from the regular layer classes.

9.1.4.1. input_data

The macro 'INPUT_DAT(*name*)' translates to
nsl_input_data *name*("*name*")
where **nsl_input_data** is the name of the object class, and *name* is the name of instantiation variable. "*name*" is utilized for linking the compiled code with the run time interpreted environment.

The data structure is as follows:

```
class nsl_input_data : public nsl_data
{
    int stim_total;
```

```
        nsl_stim_list*  stim_list;
    public:
        nsl_input_data(char*);
        ~nsl_input_data();

        int  get_stim_total();
    };
```

9.1.4.2. input_vector

The macro 'INPUT_VEC(*name,size*) translates to

nsl_input_vector *name*("*name*",*size*)

where **nsl_input_vector** is the name of the object class, and *name* is the name of instantiation variable. "*name*" is utilized for linking the compiled code with the run time interpreted environment. *size* is simply the size of the vector which get also passed as argument to the 'nsl_vector' class constructor.

The vector structure is as follows:

```
    class nsl_input_vector : public nsl_vector
    {
        int  stim_total;
        nsl_stim_list*  stim_list;
    public:
        nsl_input_vector(char*,int);
        ~nsl_input_vector();

        int  get_stim_total();
    };
```

9.1.4.3. input_matrix

The macro 'INPUT_MAT(*name,rows,columns*)' translates to

nsl_input_matrix *name*("*name*",*rows,columns*)

where **nsl_input_matrix** is the name of the object class, and *name* is the name of instantiation variable. "*name*" is utilized for linking the compiled code with the run time interpreted environment. *rows* and *columns* simply the size of the matrix which get also passed as arguments to the 'nsl_matrix' class constructor.

The matrix structure is as follows:

```
    class nsl_input_matrix : public nsl_matrix
    {
        int  stim_total;
        nsl_stim_list*  stim_list;
    public:
        nsl_input_matrix(char*,int,int);
        ~nsl_input_matrix();

        int  get_stim_total();
    };
```

9.2. Adding NSL Language Functions

New NSL language functions can be written by following C++ syntax. Any NSL data structure can be used inside functions and as arguments to the function. The only restriction is that no NSL macros may be declared (DATA, VECTOR, MATRIX, etc.), only the raw nsl structures (nsl_data, nsl_vector, nsl_matrix, etc.). It is also important that any nsl structures be declared by specifying its size and not the optional name, thus making it unknown from the global nsl structures[34].

For example, to write a function that takes two nsl_matrix arguments, and,
- checks if both matrix arguments are of the same size,
- declares a new matrix m of equal size,
- assigns to the corresponding elements of the new matrix either '1' if both argument matrices have an equal corresponding element value or '0' otherwise,
- the newly created matrix containing the result is then returned.

```
nsl_matrix function(nsl_matrix& m1,nsl_matrix& m2)
{
    int rows = m1.get_rows();
    int cols = m1.get_cols();

    nsl_matrix m(rows,cols);        // initializes the matrix with '0'

    if ((rows != m2.get_rows() || cols != m2.get_cols())
        cmd_error("Bad matrix sizes in function");
    else
    {
        for (int i = 0; i < rows; i++)          // 'i' index go from '0' to 'rows-1'
            for (int j = 0; j < cols; j++)      // 'j' index goes from '0' to 'cols-1'
                if (m1.elem(i,j) == m2.elem(i,j))
                    m.elem(i,j) = 1;
                else  // this 'else' part is not really necessary since 'm' was initialized to '0'
                    m.elem(i,j) = 0;
    }

    return m;
}
```

Note that structures can be passed by value or by reference. In this case the matrices were passed by reference, permitting any changes on the argument matrices to take effect outside the function, although this was not the case. Passing by reference is also more efficient than passing by value, where the structure gets copied (for passing by value omit the '&' in the argument declaration).

9.3. Adding Simulation Commands

New simulation commands may be added to NSL command interpreter. This is achieved by writing the following command reading function in a file to be linked via 'nsl_link'.

```
# include "nsl_include.h"

extern int new_command_function(istream&);  // C++ declaration of your command function

int my_command_function(istream& in)
{
    string str;
    int status = 0;
```

34 NSL keeps global structures independent from locally declared structures.

```
    if (nsl_get_str(in,str) == 0)
    {
        cmd_error("Empty command string");
        return -1;
    }

    if (strcmp(str,"new_command") == 0)
        status = new_command_function(in);
    else
        cmd_error("Unknown command: ",str);

    return status;
}
```

Every time 'nsl_get_str' gets called, a new string is read from the interpreter. 'new_command_function' will contain the code that actually performs the command, including reading any other command line parameters via more 'nsl_get_str'. The name of the command is 'new_command'.

The file 'nsl_main.c' has to be edited so it would look as follows:

```
# include "nsl_include.h"

extern int my_command_function(istream&);

main(int argc,char** argv)
{
    NSLinit(argc,argv);

    NSL_USER_CMD = my_command_function;

    NSLmain(argc,argv);
}
```

These commands may then be called from the interpreter with the following syntax,

user new_command [optional modifiers]

9.4. Adding Simulation Functions in C++

In order to permit greater flexibility from the command interpreter, NSL permits the addition of general purpose simulation command functions to be written in C++ code. This is achieved by declaring a special type of module in the model file, called **FUNC_MODULE**. The purpose of this module is to provide the link between the model language and the command interpreter. A FUNC_MODULE may include any C++ code, or it may call regular simulation commands (see next section). The interpreter itself would call the function by 'exec *function_name*'. (No arguments may be passed to the function.) Any C++ expressions may be included inside this module, such as conditionals and iterative loops.

The syntax is as follows,

FUNC_MODULE(*function_name*)
{
    ...function specification...

}

Calling simulation commands from inside a module is described in the next section.

9.4. Calling Simulation Commands from C++

Simulation commands may be called from the model file in the following manner,

NSL << *command* [<< *command*...] **<< ENDL;**

where NSL is the command interpreter, *command* is the desired command, and ENDL specifies the end of an expression (without ENDL the command wouldn't be processed). There may be as many '<< *command*' as necessary; this is particularly useful for sending commands composed of part string and part numerical values.

For example you could execute a status command from inside a FUNC_MODULE by having the following expression:

NSL << "status data_value ALL" << ENDL;

which will show the data values for all layers when the function containing this line gets executed.

9.6. Adding Graphics

New graphics libraries may be incorporated as part of NSL[35].

10. NSL Simulation Command Summary

The next sections provide a summary of all NSL simulation commands.

10.1. help

help 'command' ['command option']

NSL help on simulation related commands

command	description
help	- help command
create	- create model objects
disable	- disable objects
enable	- enable objects
exec	- execute a particular function module
exit,quit	- exit NSL
file	- file info
file_enable	- enable layer filing
file_disable	- disable layer filing
close	- close a file

35 A separate document will show a sample creation of new types of graphics libraries.

load,read	- read a file
open	- open a file
write	- write a file
proc	- processing info
cont	- process modules starting at current time
init	- process modules for t=0
run	- process modules for t≥0
step	- process once through all modules
reset	- reset commands
set	- set commands
shell	- unix shell commands
cd	- change model directory
csh	- execute a c-shell command
sh	- execute a shell command
status	- print status information on the model
update	- update model objects
user	- call a user defined command

10.2. create

The create command is utilized for creating model class objects. The syntax is as follows,

create 'type'

type	description
block_stim	create block-type stim object
icon_stim	create icon-type stim object
time_stim	create time interval for existing stim object

10.3. disable

disable 'type' ['option']

disable processing for object type

type	option	description
network		disable current model
	'name'	disable a particular model by name
	'index'	disable a particular model by index
module		disable current module
	'name'	disable a particular module by name
	'index'	disable a particular module by index
layer		disable current layer update
	'name'	disable a particular layer update by name
	'index'	disable a particular layer update by index
input_layer		disable current input layer
	'name'	disable an input layer by name
	'index'	disable an input layer by index
input_stim		disable current input stim
	'name'	disable an input stim by name

'index' disable an input stim by index

- 'name' is either a string name or 'ALL'
- 'index' is an integer

10.4. enable

enable 'type' ['option']

enable processing for object type

type	option	description
network		enable current model
	'name'	enable a particular model by name
	'index'	enable a particular model by index
module		enable current module
	'name'	enable a particular module by name
	'index'	enable a particular module by index
layer		enable current layer update
	'name'	enable a particular layer update by name
	'index'	enable a particular layer update by index
input_layer		enable current input layer
	'name'	enable an input layer by name
	'index'	enable an input layer by index
input_stim		enable current input stim
	'name'	enable an input stim by name
	'index'	enable an input stim by index

- 'name' is either a string name or 'ALL'
- 'index' is an integer

10.5. exec

exec - execute a function module defined in the model file

10.6. exit

exit,quit - exit NSL

10.7. file

'cmd' ['option'] 'name'

commands for dealing with external i/o

cmd	option	description

file_enable		enable layer 'name' for file in/out
file_disable		disable layer 'name' for file in/out
load,read		read from file 'name'
write		write into 'filename'
open		open file 'name' according to current 'file_type'
	INPUT	open file 'name' for input
	OUTPUT	open file 'name' for output
close		close file 'name'

10.8. proc

'cmd' ['option']

process model

cmd	option	description
init		process modules for t=0
run		process modules for 0<=t<end_time
	'ts'	process modules for 0<=t<=ts
cont		process modules for current time<t<=end_time
	'ts'	process modules for current_time<t<ts
step		process modules 1 iteration
	'n'	process modules n iterations

10.9. reset

reset 'type'

reset object type

type	description
stim_list	reset stimuli list

10.10. set

set 'type' ['option']

set buffer for objects in the next categories

type class	description
network	network information
module	module information
layer	layer information
input_layer	input layer information
input_stim	stimulus information

network

set 'modifier' 'option'

modifier	option	description
network		current network
	'name'	network by name
	'index'	network by index
end_time		current end_time
	'value'	set simulation 'end_time' to 'value'
current_time		current time
	'value'	set simulation 'current_time' to 'value'
delta		current delta time step
	'value'	set simulation 'delta' to 'value'
integration		current integration numerical method
	'method'	set simulation numerical method to 'method'

module

set 'modifier' 'option'

modifier	option	description
module		current module
	'name'	module by name
	'index'	module by index

layer

set 'modifier' 'option'

modifier	option	description
layer		current layer
	'name'	layer by name
	'index'	layer by index
data_value	'name'	layer name followed by all values
all_data_value	'name'	layer name followed by single data value for whole layer
elem_i	'index'	layer row element index
elem_j	'index'	layer col element index
elem_data_value	'name'	layer name followed by elem data value used in conjunction with elem_i and elem_j

input_layer

set 'modifier' 'option'

modifier	option	description
input_layer		current input layer
	'name'	input layer by name
	'index'	input layer by index
input_xz	'index'	zero coordinate x layer element, default is 0
input_yz	'index'	zero coordinate y layer element, default is 0
input_dx	'value'	distance between adjacent x layer elements,

		default is 1.0
input_dy	'value'	distance between adjacent y layer elements, default is 1.0

input_stim

set 'modifier' 'option'

modifier	option	description
input_stim		current stim
	'index'	stim by index
stim_index 'index'		stim index
stim_dx	'value'	stim x size (width), default is 1.0
stim_dy	'value'	stim y size (height), default is 1.0
stim_xc	'value'	stim x center initial location, default is 0.0
stim_yc	'value'	stim y center initial location, default is 0.0
stim_x0	'value'	stim x left initial location, default is 0.0
stim_y0	'value'	stim y top initial location, default is 0.0
stim_spec	'type'	CENTER or CORNER
stim_type	'type'	BLOCK or ICON
stim_val	'value'	stim value (used for BLOCK type), default is 1.0
stim_mat	'matrix'	stim icon matrix (used for ICON type)
stim_vx	'value'	stim velocity, positive going right
stim_vy	'value'	stim velocity, positive going up
stim_t0	'value'	beginning of stim time interval
stim_t1	'value'	end of stim time interval

10.11. shell

'cmd' 'option'

execute a unix shell command

cmd	option	description
cd	'dir_name'	change the model directory to 'dir_name'
csh	'cmd_line'	execute a csh 'cmd_line'
sh	'cmd_line'	execute a sh 'cmd_line'

10.12. status

status 'type' ['option']

print status information for object type

type	option	description
system		list all linked models
files		list all active data files
current		current system status
network		status on current model

	'name'	status on a model by name
	'index'	status on a model by index
module		status on current module
	'name'	status on a module by name
	'index'	status on a module by index
layer		status on current layer
	'name'	status on a layer by name
	'index'	status on a layer by index
data_value		current layer data values
	'name'	current layer data values by name
	'index'	current layer data values by index
input_layer		status on current input_layer
	'name'	status on an input_layer by name
	'index'	status on an input_layer by index
input_stim		print all input layer stimuli
	'name'	print stimuli of an input layer by name
	'index'	print stimuli of an input layer by index

- 'name' is either a string name or 'ALL'
- 'index' is an integer

10.13. update

The update command is utilized for updating model class objects. The syntax is as follows,

update 'type'

type	description
block_stim	update block-type stim object
icon_stim	update icon-type stim object
time_stim	update time interval for existing stim object

10.14. user

user 'cmd' user written command

11. NSL Window Interface Command Summary

There are several types of display class commands including the creation of objects, and the modification of object parameters.

11.1. help

help 'command' ['command option']

NSL help on window interface commands

command	description

help	- this command
create	- create model objects
disable	- disable objects
enable	- enable objects
print	- printing commands
print_dump	- print a screen dump
win_dump	- get a screen dump
reset	- reset options
set	- set options
status	- status information on display objects
update	- update display objects

11.2. create

The create command is utilized for creating display class objects. The syntax is as follows,

create 'type'

type	description
display_frame	create display frame
display_window	create display window

11.3. disable

disable 'type' ['option']

disable window object type

type	option	description
display_panel		disable current display panel
	'name'	disable display panel by name
	'index'	disable display panel by index
canvas_item		disable current canvas item
	'index'	disable canvas item by index
panel_item		disable current panel item
	'index'	disable panel item by index

- 'name' is either a string name or 'ALL'
- 'index' is an integer

11.4. enable

enable 'type' ['option']

enable window object type

type	option	description

display_panel		enable current display panel
	'name'	enable display panel by name
	'index'	enable display panel by index
canvas_item		enable current canvas item
	'index'	enable canvas item by index
panel_item		enable current panel item
	'index'	enable panel item by index

- 'name' is either a string name or 'ALL'
- 'index' is an integer

11.5. print

A window dump may also be generated from the command interpreter as follows,

'cmd' 'option'

cmd	option	description
win_dump		do a window dump on the current 'frame_name'
	'frame_name'	do a window dump on the specified 'frame_name'
print_dump		print the current 'dump_file'
	'dump_file'	print the specified 'dump_file'

The win_dump command may be instructed to generate automatic file names by using the following commands,

set 'modifier' 'option'

modifier	option	description
dump_file	'name'	name of file used for window dumps.
dump_type	PRINTER	output will be sent directly to printer
	RASTER	output will be saved as raster ('ras') file
	POSTSCRIPT	output will be saved as postscript ('ps') file
dump_ras_index	'n'	'ras' dump file suffix number.
dump_ps_index	'n'	'ps' dump file suffix number.
dump_auto_name	ON	output file name is composed: 'dump_file.dump_type.index'
	OFF	output file name is same as 'dump_file'.

11.6. reset

reset 'type'

reset display objects

type	description
window_interface	reset window interface

11.7. set

set 'type' ['option']

set buffer for objects in the next categories

type class	description
display_frame	display frame information
display_window	display window information
window_item	window item information
canvas_item	canvas item information
panel_item	panel item information

display_frame

set 'modifier' 'option'

modifier	option	description
display_frame		current display frame
	'name'	display frame by name
	'index'	display frame by index
frame_name	'name'	frame's name.
frame_X0	'n'	frame's left corner's location on the screen (# pixels).
frame_Y0	'n'	frame's top corner's location on the screen (# pixels).
frame_width	'n'	frame's width (# pixels).
frame_height	'n'	frame's height (# pixels).
frame_auto_resize	'type'	ON - automatic resize, OFF - no automatic resize.
frame_rows	'n'	number of window rows (automatic redraw).
frame_cols	'n'	number of window columns (automatic redraw).

display_window

set 'modifier' 'option'

modifier	option	description
display_window		current display window
	'name'	display frame by name
	'index'	display frame by index
window_name	'name'	window's name
window_X0	'n'	window's left corner's location on the screen (# pixels)
window_Y0	'n'	window's top corner's location on the screen (# pixels)
window_width	'n'	window's width (# pixels)
window_height	'n'	window's height (# pixels)
window_step	'n'	window update step (default 1)
window_layer	'name'	layer's name to be displayed
window_layer_i0	'n'	layer's i0 unit to be displayed
window_layer_i1	'n'	layer's i1 unit to be displayed
window_layer_j0	'n'	layer's j0 unit to be displayed
window_layer_j1	'n'	layer's j1 unit to be displayed
window_graph	'type'	area_level_graph

			spatial_graph_2
			spatial_graph_3
			temporal_graph
window_wymin		'value'	graph's min layer unit value
window_wymax		'value'	graph's max layer unit value
window_t0	'value'	temporal graph starting time	
window_t1	'value'	temporal graph ending time	
window_val_in		'value'	interactive mouse input value
window_pulse		'type'	draw using 'pulse' type lines (ON/OFF)
window_vec_type		'type'	vector drawing type (HORIZONTAL/VERTICAL)
window_hidden_line		'type'	draw 3D with hidden line removal (ON/OFF)
window_ax	'value'	3D graph x reference point	
window_by	'value'	3D graph y reference point	
window_cz	'value'	3D graph z reference point	
window_sx	'value'	3D graph x scale	
window_sy	'value'	3D graph y scale	
window_sz	'value'	3D graph z scale	

11.8. status

status 'type' ['option']

print status information for object type

type	option	description
window_interface		interface status information
display_frame		frame status information
	'name'	display frame status by name
	'index'	display frame status by index
display_window		window status information
	'name'	display window status by name
	'index'	display window status by index
display_canvas		canvas status information
	'name'	display canvas status by name
	'index'	display canvas status by index
display_panel		panel status information
	'name'	display panel status by name
	'index'	display panel status by index
window_item		window item status information
	'name'	window item status by name
	'index'	window item status by index
canvas_item		canvas item status information
	'name'	canvas item status by name
	'index'	canvas item status by index
panel_item		panel item status information
	'name'	panel item status by name
	'index'	panel item status by index

- 'name' is either a string name or 'ALL'
- 'index' is an integer

11.9. update

The update command updates current display information,

update 'type'

type	description
window_interface	update all display frames and display windows
display_frame	update display frame
display_window	update display window
display_canvas	update display canvas
display_panel	update display panel
canvas_item	update canvas item
panel_item	update panel item

12. References

Amari, S., Arbib, M.A., 1977, *Competition and Cooperation in Neural Nets*, Systems Neuroscience (J. Metzler, ed.), pp. 119-165, Academic Press.

Arbib, M.A., 1989, *The Metaphorical Brain 2: Neural Networks and Beyond*, Wiley.

Didday, R.L., 1976, *A Model of Visuomotor Mechanisms in the Frog Optic Tectum*, Math. Biosci., Vol. 30, pp. 169-180.

Goddard, N., Lynne, K.J., Mintz, T., 1987, *Rochester Connectionist Simulator*, (User's Manual), University of Rochester, Department of Computer Science.

Hebb, D.O., 1949, *The Organization of Behavior*, Wiley.

Hodgkin, A.L, Huxley, A.F., 1952, *A Quantitative Descriptionof Membrane Current and its Application to Conduction and Excitation in Nerve*, J. Physiol., London, Vol. 117, pp. 500-544.

Kandel, E.R., Schwartz, J.H., 1985, *Principles of Neural Science*, Elsevier.

Kernighan, B.W., Ritchie, D.M., 1978, *The C Programming Language*, Prentice-Hall.

McCulloch, W.S., Pitts, W.H., 1943, *A Logical Calculus of the Ideas Immanent in Nervous Activities*, Bull. Math. Biophys., Vol. 5, pp. 115-133.

Rumelhart, D.E., McClelland, J.L., 1986, *Parallel Distributed Processing*, Vol I & II, MIT Press.

Stroustrup, B., 1987, *The C++ Programming Language*, Addison Wesley.

Teeters, J., 1989,*A Simulation System for Neural Networks and Model for the Anuran Retina*, TR 89-01 (PhD Thesis), Center for Neural Engineering, University of Southern California.

Wang, D., Hsu, C., 1990, *SLONN: A Simulation Language for modeling of Neural Networks*, Simulation pp. 69-83.

Weitzenfeld, A., 1989, *NSL, Neural Simulation Language, Version 1.0*, TR 89-02, Center for Neural Engineering, University of Southern California.

Weitzenfeld, A., 1990, *NSL, Neural Simulation Language, Version 2.0*, TR 90-01, Center for Neural Engineering, University of Southern California.

Wilson, M.A., Bhalla, U.S., Uhley, J.D., Bower, J.M., 1989, *GENESIS: A System for Simulating Neural Networks*, Advances in Neural Network Information Processing System, Morgan Kauffman.

13. Appendices

The following appendices give have been added to the manual to give further explanations.

13.1. NSL Environment

To start an NSL session, one including graphics, make sure you have initialized the appropriate windowing environment. (Due to the fact that NSL's current window interface release runs under the X window environment. The regular 'xstart' or 'xinit' can be used to enter the environment. In a Sun, there is also the option to run the openwinodws environment by executing 'openwin'. 'nsl' can be run from any 'c-shell' window, or any 'xterm'.)

If this is the first time you are going to run a model in NSL, execute first the 'nsl_init' command to create default settings, and copy the sample models from the NSL directories.

13.2. NSL Library Functions

The following is a list of the different library functions available in NSL describing the number and type of arguments accepted by each one of them. *nsl_data*, *nsl_vector*, and *nsl_matrix* are NSL layer classes, while *num_type* is a simple data type, currently defined to be a 'float'.

// **arithmetic**

```
// addition
nsl_data     operator+(nsl_data&);
nsl_data     operator+(nsl_data&,nsl_data&);
nsl_data     operator+(nsl_data&,num_type);
nsl_data     operator+(num_type,nsl_data&);
nsl_vector   operator+(nsl_vector&,nsl_vector&);
nsl_vector   operator+(nsl_vector&);
nsl_vector   operator+(nsl_vector&,num_type);
nsl_vector   operator+(num_type,nsl_vector&);
nsl_vector   operator+(nsl_vector&,nsl_data&);
nsl_vector   operator+(nsl_data&,nsl_vector&);
nsl_matrix   operator+(nsl_matrix&);
nsl_matrix   operator+(nsl_matrix&,nsl_matrix&);
nsl_matrix   operator+(nsl_matrix&,nsl_data&);
nsl_matrix   operator+(nsl_data&,nsl_matrix&);
nsl_matrix   operator+(nsl_matrix&,num_type);
nsl_matrix   operator+(num_type,nsl_matrix&);

// subtraction
nsl_data     operator-(nsl_data&);
nsl_data     operator-(nsl_data&,nsl_data&);
nsl_data     operator-(nsl_data&,num_type);
nsl_data     operator-(num_type,nsl_data&);
nsl_vector   operator-(nsl_vector&);
nsl_vector   operator-(nsl_vector&,nsl_vector&);
nsl_vector   operator-(nsl_vector&,num_type);
nsl_vector   operator-(num_type,nsl_vector&);
nsl_vector   operator-(nsl_vector&,nsl_data&);
nsl_vector   operator-(nsl_data&,nsl_vector&);
nsl_matrix   operator-(nsl_matrix&);
nsl_matrix   operator-(nsl_matrix&,nsl_matrix&);
nsl_matrix   operator-(nsl_matrix&,nsl_data&);
nsl_matrix   operator-(nsl_data&,nsl_matrix&);
```

```
nsl_matrix     operator-(nsl_matrix&,num_type);
nsl_matrix     operator-(num_type,nsl_matrix&);

// pointwise multiplication
nsl_vector     operator^(nsl_vector&,nsl_vector&);
nsl_vector     operator^(nsl_vector&,num_type);
nsl_vector     operator^(num_type,nsl_vector&);
nsl_vector     operator^(nsl_vector&,nsl_data&);
nsl_vector     operator^(nsl_data&,nsl_vector&);
nsl_matrix     operator^(nsl_matrix&,nsl_matrix&);
nsl_matrix     operator^(nsl_matrix&,nsl_data&);
nsl_matrix     operator^(nsl_data&,nsl_matrix&);
nsl_matrix     operator^(nsl_matrix&,num_type);
nsl_matrix     operator^(num_type,nsl_matrix&);

// pointwise division
nsl_data       operator/(nsl_data&,nsl_data&);
nsl_data       operator/(nsl_data&,num_type);
nsl_data       operator/(num_type,nsl_data&);
nsl_vector     operator/(nsl_vector&,nsl_vector&);
nsl_vector     operator/(nsl_vector&,num_type);
nsl_vector     operator/(num_type,nsl_vector&);
nsl_vector     operator/(nsl_vector&,nsl_data&);
nsl_vector     operator/(nsl_data&,nsl_vector&);
nsl_matrix     operator/(nsl_matrix&,nsl_matrix&);
nsl_matrix     operator/(nsl_matrix&,nsl_data&);
nsl_matrix     operator/(nsl_data&,nsl_matrix&);
nsl_matrix     operator/(nsl_matrix&,num_type);
nsl_matrix     operator/(num_type,nsl_matrix&);

// convolution
nsl_data       operator*(nsl_data&,nsl_data&);
nsl_data       operator*(nsl_data&,num_type);
nsl_data       operator*(num_type,nsl_data&);
nsl_vector     operator*(nsl_vector&,nsl_vector&);
nsl_matrix     operator*(nsl_vector&,nsl_matrix&);
nsl_vector     operator*(nsl_vector&,nsl_data&);
nsl_vector     operator*(nsl_data&,nsl_vector&);
nsl_vector     operator*(nsl_vector&,num_type);
nsl_vector     operator*(num_type,nsl_vector&);
nsl_matrix     operator*(nsl_matrix&,nsl_matrix&);
nsl_matrix     operator*(nsl_vector&,nsl_matrix&); // row vector convolution
nsl_matrix     operator*(nsl_matrix&,nsl_vector&); // column vector convolution
nsl_matrix     operator*(nsl_matrix&,nsl_data&);
nsl_matrix     operator*(nsl_data&,nsl_matrix&);
nsl_matrix     operator*(nsl_matrix&,num_type);
nsl_matrix     operator*(num_type,nsl_matrix&);
```

// **threshold functions**

```
// NSLmax
num_type       NSLmax(nsl_vector&);
num_type       NSLmax(nsl_matrix&);
nsl_data       NSLmax(nsl_data&,nsl_data&);
```

```
nsl_data      NSLmax(nsl_data&,num_type);
nsl_vector    NSLmax(nsl_vector&,nsl_vector&);
nsl_vector    NSLmax(nsl_vector&,nsl_data&);
nsl_vector    NSLmax(nsl_vector&,num_type);
nsl_matrix    NSLmax(nsl_matrix&,nsl_matrix&);
nsl_matrix    NSLmax(nsl_matrix&,nsl_data&);
nsl_matrix    NSLmax(nsl_matrix&,num_type);

// NSLmin
num_type      NSLmin(nsl_vector&);
num_type      NSLmin(nsl_matrix&);
nsl_data      NSLmin(nsl_data&,nsl_data&);
nsl_data      NSLmin(nsl_data&,num_type);
nsl_vector    NSLmin(nsl_vector&,nsl_vector&);
nsl_vector    NSLmin(nsl_vector&,nsl_data&);
nsl_vector    NSLmin(nsl_vector&,num_type);
nsl_matrix    NSLmin(nsl_matrix&,nsl_matrix&);
nsl_matrix    NSLmin(nsl_matrix&,nsl_data&);
nsl_matrix    NSLmin(nsl_matrix&,num_type);

// NSLstep
num_type      NSLstep(num_type);
nsl_data      NSLstep(nsl_data&);
nsl_vector    NSLstep(nsl_vector&);
nsl_matrix    NSLstep(nsl_matrix&);

num_type      NSLstep(num_type,num_type,num_type,num_type);
nsl_data      NSLstep(nsl_data&,nsl_data&,nsl_data&,nsl_data&);
nsl_vector    NSLstep(nsl_vector&,nsl_vector&,nsl_vector&,nsl_vector&);
nsl_vector    NSLstep(nsl_vector&,nsl_vector&,nsl_data&,nsl_data&);
nsl_vector    NSLstep(nsl_vector&,nsl_data&,nsl_data&,nsl_data&);
nsl_matrix    NSLstep(nsl_matrix&,nsl_matrix&,nsl_matrix&,nsl_matrix&);
nsl_matrix    NSLstep(nsl_matrix&,nsl_matrix&,nsl_data&,nsl_data&);
nsl_matrix    NSLstep(nsl_matrix&,nsl_data&,nsl_data&,nsl_data&);

// NSLramp
num_type      NSLramp(num_type);
nsl_data      NSLramp(nsl_data&);
nsl_vector    NSLramp(nsl_vector&);
nsl_matrix    NSLramp(nsl_matrix&);

num_type      NSLramp(num_type,num_type,num_type,num_type);
nsl_data      NSLramp(nsl_data&,nsl_data&,nsl_data&,nsl_data&);
nsl_vector    NSLramp(nsl_vector&,nsl_vector&,nsl_vector&,nsl_vector&);
nsl_vector    NSLramp(nsl_vector&,nsl_vector&,nsl_data&,nsl_data&);
nsl_vector    NSLramp(nsl_vector&,nsl_data&,nsl_data&,nsl_data&);
nsl_matrix    NSLramp(nsl_matrix&,nsl_matrix&,nsl_matrix&,nsl_matrix&);
nsl_matrix    NSLramp(nsl_matrix&,nsl_matrix&,nsl_data&,nsl_data&);
nsl_matrix    NSLramp(nsl_matrix&,nsl_data&,nsl_data&,nsl_data&);

// NSLsaturation
num_type      NSLsaturation(num_type);
nsl_data      NSLsaturation(nsl_data&);
nsl_vector    NSLsaturation(nsl_vector&);
```

```
nsl_matrix    NSLsaturation(nsl_matrix&);

num_type      NSLsaturation(num_type,num_type,num_type,num_type,num_type);
nsl_data      NSLsaturation(nsl_data&,nsl_data&,nsl_data&,nsl_data&,nsl_data&);
nsl_vector    NSLsaturation(nsl_vector&,nsl_vector&,nsl_vector&,nsl_vector&,nsl_vector&);
nsl_vector    NSLsaturation(nsl_vector&,nsl_vector&,nsl_vector&,nsl_data&,nsl_data&);
nsl_vector    NSLsaturation(nsl_vector&,nsl_data&,nsl_data&,nsl_data&,nsl_data&);
nsl_matrix    NSLsaturation(nsl_matrix&,nsl_matrix&,nsl_matrix&,nsl_matrix&,nsl_matrix&);
nsl_matrix    NSLsaturation(nsl_matrix&,nsl_matrix&,nsl_matrix&,nsl_data&,nsl_data&);
nsl_matrix    NSLsaturation(nsl_matrix&,nsl_data&,nsl_data&,nsl_data&,nsl_data&);

// NSLsigmoid
num_type      NSLsigmoid(num_type);
nsl_data      NSLsigmoid(nsl_data&);
nsl_vector    NSLsigmoid(nsl_vector&);
nsl_matrix    NSLsigmoid(nsl_matrix&);

num_type      NSLsigmoid(num_type,num_type,num_type,num_type,num_type);
nsl_data      NSLsigmoid(nsl_data&,nsl_data&,nsl_data&,nsl_data&,nsl_data&);
nsl_vector    NSLsigmoid(nsl_vector&,nsl_vector&,nsl_vector&,nsl_vector&,nsl_vector&);
nsl_vector    NSLsigmoid(nsl_vector&,nsl_vector&,nsl_vector&,nsl_data&,nsl_data&);
nsl_vector    NSLsigmoid(nsl_vector&,nsl_data&,nsl_data&,nsl_data&,nsl_data&);
nsl_matrix    NSLsigmoid(nsl_matrix&,nsl_matrix&,nsl_matrix&,nsl_matrix&,nsl_matrix&);
nsl_matrix    NSLsigmoid(nsl_matrix&,nsl_matrix&,nsl_matrix&,nsl_data&,nsl_data&);
nsl_matrix    NSLsigmoid(nsl_matrix&,nsl_data&,nsl_data&,nsl_data&,nsl_data&);
```