NSL
Neural Simulation Language

Alfredo Weitzenfeld
Departamento Académico de Computación
Instituto Tecnológico Autónomo de México
Río Hondo #1, Tizapán San Angel 01000
México D.F., México

Correspondence:

Alfredo Weitzenfeld
Departamento Académico de Computación
Instituto Tecnológico Autónomo de México
Rio Hondo #1, Tizapán San Angel 01000
México D.F., México
Phone: +52-55-56284060
Fax: +52-55-56162211
email: alfredo@itam.mx

INTRODUCTION

Neural simulation plays an essential role in understanding the brain. While many neural simulators exist today (see NEUROSIMULATORS for a listing of the most important ones), design considerations can be quite different. For example, systems supporting very detailed neural elements can simulate only a few neurons at a time (see NEURON and GENESIS), while systems supporting coarser elements can usually simulate larger neural populations. In this article we describe the *Neural Simulation Language* (NSL) (Weitzenfeld et al., 2002), an *object-oriented* system (Wegner, 1990) primarily designed to support simulation of large neural networks. The system addresses the needs of a wide range of users, from novice users requiring friendly user interfaces to advanced users requiring advanced programming and integration to other systems. Two versions of the system exist today, one in Java (Gosling et al., 2000) and the other one in C++ (Stroustrup, 2000). Both of these can run in a wide range of computer platforms making the system quite independent from the actual computing environment.

MODULARITY IN NEURAL SYSTEMS

A particular aspect that distinguishes NSL from comparable simulators is its special focus on *modularity*, a well-known software development strategy in dealing with large and complex systems. As neural models become large and complex, they become hard to manage. Moreover, modularization of biological neural networks is further motivated by taking into consideration the way we analyze the brain as a set of different brain regions, as seen by the example shown in Figure 1.

The general methodology for understanding a complex neural system involves two basic approaches. One is to focus on some particular brain region or module and carry out studies of that region in detail. The other is to step back and look at higher levels of organization in which the details of particular modules are hidden. Full understanding comes as we cycle back and forth between different levels of detail in analyzing different subsystems, sometimes simulating modules in isolation, at other times designing computer experiments that help us follow the dynamics of the interactions between the various modules.

MODELING IN NSL

There are two ways to describe a model in NSL: (i) by direct programming in NSLM, the NSL (compiled) Modeling language; and (ii) by using the *Schematic Capture System* (SCS), a visual programming interface to NSLM supporting the description of module assemblages. In general, NSL supports the two levels of modeling, *modules* and *neural networks*, as described next.

Modules

Modules in NSL are hierarchical structures organized in a tree fashion having root module, the *model* and multiple levels of *module assemblages*. Modules may be implemented in different ways and independently from each other in a top-down and bottom-up fashion, an important benefit from modular design. In particular, *neural* modules are implemented with neural networks, corresponding to leaves in the tree. In general, the external interface to a module is described by a set of unidirectional input and output *data ports*, representing module entry or exit points, where data is sent or received, usually in the form of numerical values with varying dimension, i.e., a single scalar, a one-dimensional array of values (*vector*), a two-dimensional array (*matrix*), or higher ones. In order to communicate, modules require interconnections among ports belonging to different modules. The following is sample NSLM code describing module assemblages:

```
nslModel Model()
{
    private StimulusModule stimulus();
    private MainModule main();
    private OutputModule output();

    public void makeConn(){
           nslConnect(stimulus.sout,main.in);
           nslConnect(stimulus.sout,output.sin);
           nslConnect(main.out,output.oin);
    }
}
```

The description is analogous to a class specification in object-oriented programming. The attribute section describes a three-module assemblage consisting of a “stimulus”, “main” and “output” modules, while the *makeConn* method specifies module interconnections using the *nslConnect* statement (see Weitzenfeld et al., 2002 for a more extensive description of all NSLM commands.) This sample NSLM code could be automatically generated from SCS as well. Figure 2 shows sample schematics for a module assemblage within a higher-level module.

Neural Networks

Modules representing brain regions can be anatomically or physiologically divided until reaching *neural modules*, modules described by neural arrays. In order to model a complete neural network it is necessary to describe (1) the particular neuron model, i.e., the desired neural level of detail, (2) the neurons making up the network, (3) the set of interconnections among neurons, and (3) network parameters, such as inputs and connection weights. Without precluding the importance of other neural models, we focus here on the *leaky integrator* (Arbib, 1989) neuron model, a single-compartment neuron, having one output and many inputs. The internal state of the neuron is described by a single scalar quantity, its membrane potential *mp* that depends on the neuron's inputs and past history. The output is described by another single scalar

quantity, its firing rate *mf*, and may serve as input to multiple neurons, including itself. As the input to a neuron varies the membrane potential and firing rate vary as well.

In NSL two numerical structures (**NslDouble0** data type) are required to represent such a neuron, one corresponds to the membrane potential and the other one to its firing rate:

```
private NslDouble0 mf();
private NslDouble0 mp();
```

In many cases we may want the value of *mf* to be communicated to other modules. If such is the case, the declaration for *mf* should be modified from a private variable to a public output port (note the *Dout* keyword):

```
public NslDoutDouble0 mf();
```

The *membrane potential* for *mp* is described by a first-order differential equation with dependence on its previous history and input s_m

$$\tau_m \frac{dmp(t)}{dt} = f(s_m, mp, t)$$

Variable τ_m represents the time constant, while the choice of *f* defines the particular neural model utilized. The *leaky integrator* model is described by $f(s_m,mp,t) = -mp(t) + s_m(t)$, or

$$\tau_m \frac{dmp(t)}{dt} = -mp(t) + s_m(t)$$

In addition to the membrane potential and firing rate descriptions, we also need to specify the input to the neuron, s_m, internal to the module or obtained from another module. In the latter case input s_m would be specified as an input port (note the "Din" keyword):

```
public NslDinDouble0 sm();
```

where *sm* holds a weighted spatial summation of all input to the corresponding neuron.

While neural networks are continuous in their nature, their simulated state is approximated by discrete time computations. For this reason we must specify an integration or approximation method to generate as faithfully as possible the corresponding neural state. The dynamics for *mp* are described by the following statement,:

```
mp = nslDiff(mp,tau,-mp+sm);
```

Fucntion *nslDiff* defines a first-degree differential equation equal to "*-mp+sm*" as described by the leaky integrator model. Different methods can be used to approximate the differential equation, such as *Euler* and *Runge-Kutta*. The choice of method may affect both the computation time and its precision. The specific method to use is chosen during simulation and not as part of the model architecture.

The firing rate *mf*, the output of the neuron, is obtained by applying a *threshold*, typically a *ramp*, *step*, *saturation* or *sigmoidal* function, to the neuron's membrane potential,

$$mf(t) = \sigma(mp(t))$$

where σ is usually a non-linear function.

For example, if σ is set to a *step* threshold function, the equation for the firing rate *mf* would be described by

```
mf = nslStep(mp);
```

where *nslStep* is the corresponding NSL *step* threshold function.

The previous definition specifies a single neuron without any interconnections. An actual neural network is made of a number of interconnected neurons where the output of one neuron serves as input to the others. In the leaky integrator neural model, interconnections are very simple structures. On the other hand, *synapses*, the links among neurons, are – in biological systems – complex electrochemical systems and may be modeled in exquisite detail. However, many models have succeeded with a very simple synaptic model where each synapse carries a connection weight that describes how neurons affect each other. The most common formula for the input *sv* to a neuron *v* is given by

$$sv_j = \sum_{i=0}^{n-1} w_{ji} uf_i$$

where uf_i is the firing of neuron u_i whose output is connected to the *j*th input line of neuron v_j, and w_{ji} is the weight for that link, as shown in Figure 3 (*up* and *vp* are analogous to *mp*, while *uf* and *vf* are analogous to *mf*).

Expanding the summation, input to neuron v_j (identified by its corresponding membrane potential vp_j) is given by sv_j defined as

$$sv_j = w_{j0}uf_0 + w_{j1}uf_1 + w_{j2}uf_2 + ... + w_{jn\text{-}1}uf_{n\text{-}1}$$

While module interconnections are specified in NSL via a *nslConnect* method call, doing this with neurons would in general be prohibitively expensive considering there may be thousands or millions of neurons and even more connections in a single neural network. Instead we use mathematical expressions similar to those used for their representation. For example, the input to neuron v_j, represented by *svj*, would be the sum for all outputs of neuron *ufi* multiplied (using the '*' operator) by connection weight *wji,* correspondingly, as shown next:

```
svj = wj0*uf0 + wj1*uf1 + wj2*uf2 + ...;
```

Note that there exist *m* such equations in the network. We could describe each membrane potential and firing rate individually or else we could make all u_i and v_j neuron vector structures. The first approach would be very long, inefficient, and prone to typing errors; thus we present the second approach, using *neuron arrays* and *connection masks* representing spatial arrangements among homogeneous neurons and their connections, respectively. We consider uf_i the output

from a single neuron in an array of neurons and sv_j the input to a single neuron in another array of neurons.

If mask w_{jk} (for $-d \le k \le d$) represents the synaptic weights from the uf_{j+k} (for $-d \le k \le d$) elements to v_j, for every j, we then have

$$sv_j = \sum_{k=-d}^{d} w_{jk} uf_{j+k}$$

where the same mask w is applied to the output of each neuron uf_{i+k} to obtain input sv_j. In NSL, the *convolution* operation is described by a single symbol '@'.

```
sv= w@uf;
```

This kind of representation results in great conciseness, an important concern when working with large numbers of interconnected neurons. Note that this is possible as long as connections are regular. Otherwise, single neurons would still need to be connected separately on a one by one basis. This also suggests that the operation is best defined when the number of v and u neurons matches, although a non-matching number of units can be processed using a more complex notation.

SIMULATION IN NSL

Simulation involves interactively specifying aspects of the model that tend to change, in particular parameter values, input patterns, simulation control and visualization. It is not only important to design a good model, it is also important to design good graphical interfaces, both input and output. In terms of input, NSL offers a number of approaches: (i) by interactively writing code in NSLS, the NSL (interpreted) Scripting language; (ii) by loading NSLS scripts stored in files; and (iii) by designing custom input interfaces. In terms of output, NSL enables the user to specify various forms of graphical and textual output, including temporal and spatial 2D and 3D graphics (see Weitzenfeld et al., 2002 for input and output visualization examples as well as more extensive description of NSLS commands.)

DISCUSSION

In this article we have overviewed modeling and simulation using NSL, a system primarily designed to simulate modular neural systems, both biological and artificial (see BACKPROPAGATION for an example of artificial neural networks). The underlying NSL computational model is based on the *Abstract Schema Language* (ASL) (Weitzenfeld, 1993) inspired by the work on SCHEMA THEORY (q.v.), *actors* (Agha, 1986) and more generally on object-based concurrent programming (Yonezawa and Tokoro, 1987).

There are a number of issues worth discussing from our experience with NSL. While user interactivity plays an essential role while creating and testing new neural models, as model

becomes more stable, simulation efficiency becomes a primary concern. This is a very important issue if we consider that neural network execution can consume extensive amounts of processing time, possibly hours or even days depending on the size and architecture of the network. For example, we have processed in NSL "simple" biological network, such as the RETINA (q.v.) model involving ten thousand neurons, consuming just a few seconds. On the other hand, a more complex network such as the one described in Figure 1 could take several minutes if implemented by "faithful" neural components. The general solution to this problem is to use parallelism and distributed computing facilities in speeding up computation. While a number of neural systems have been ported to supercomputers, we are currently developing a distributed simulation environment to run on networks of low cost computers (Weitzenfeld et al., 2000). In general, the client-server distributed architecture has become quite pervasive thanks to the Internet.

A Web-based simulation interface (Alexander et al., 1999) brings additional possibilities to the process of neural modeling. For example, users could be offered shared model repositories in creating new models or in addressing experimental data linked to it. These two thrusts are part of a project known as *Brain Models on the Web* (*BMW*), a model repository where model assumptions, empirical data, and simulation results are stored (see DATABASES FOR NEUROSCIENCE).

An important issue arising from the sharing of module libraries is how to reuse portions of different models in creating new ones. An important consideration is to provide a general module interconnection specification to be followed by all modelers. This specification should deal with issues such as "edges" in the block diagrams as the one shown in Figure 1, where module interconnections and the corresponding ports, are designed to deal only with primitive data without any temporal considerations. Additionally, the specification could address the relationship to the particular experimental protocol on which the model is based. These aspects need to be defined and then specified as a "meta-level" that will separate the internal module characteristics from the external ones.

Another consideration is the extensibility the system. Since not all users use similar simulation systems, thus it is important to offer interoperability of data and model descriptions to be shared by multiple simulation systems and applications in general. Additionally, integration with simulated or real-time REACTIVE ROBOTIC SYSTEMS (q.v.) is of particular interest, in particular BIOLOGICALLY INSPIRED ROBOTICS (q.v.) as exemplified by a number of NSL-based neural architectures developed to control mobile robots (Fagg et al., 1992; Weitzenfeld, 2000). Since many approaches exist today to mobile robotics, it is an interesting challenge to design new architectures integrating non-neural and neural based approaches (Arkin et al. 2000).

REFERENCES

Agha, G., 1986, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press.

Alexander, A., Arbib, M.A., Weitzenfeld, A., 1999, Web Simulation of Brain Models, in Proc. of SCS 1999 International Conference on Web-Based Modelling and Simulation, January 17-20, San Francisco, California.

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

Arkin, R.C., Ali, K., Weitzenfeld, A., and Cervates-Perez, F., 2000, Behavioral Models of the Praying Mantis as a Basis for Robotic Behavior, in Journal of Robotics and Autonomous Systems, 32 (1) pp. 39-60, Elsevier.

Crowley, M., Oztop, E., Mármol, S., 2002, Crowley-Arbib Saccade Model, in The Neural Simulation Language: A System for Brain Modeling, (A. Weitzenfeld, M. Arbib and A. Alexander), MIT Press.

Fagg, A.H., King, I.K., Lewis, M.A., Liaw, J.S., Weitzenfeld, A., 1992, A Neural Network Based Testbed for Modeling Sensorimotor Integration in Robotics Applications, in Proc. of IJCNN '92, Baltimore, MD.

Gosling, J., Joy, B., Steele, G, and Bracha., 2000, G., The Java Language Specification, 2nd Edition, Addison-Wesley.

Stroustrup, B., 200, The C++ Programming Language, Special Edition, Addison-Wesley.

*Wegner, P., 1990, Concepts and Paradigms of Object-Oriented Programming, in SIGPLAN OOPS Messenger, 1(1):7-87, Aug.

Weitzenfeld, A., 1993, ASL: Hierarchy, Composition, Heterogeneity, and Multi-Granularity in Concurrent Object-Oriented Programming, in Proc. on Neural Architectures and Distributed AI: From Schema Assemblages to Neural Networks Workshop, Oct 19-20, Center for Neural Engineering, USC, Los Angeles, CA.

Weitzenfeld A., 2000, A Multi-level Approach to Biologically Inspired Robotic Systems, in Proc of NNW 2000 10th International Conference on Artificial Neural Networks and Intelligent Systems, Prague, Czech Republic, July 9-12.

*Weitzenfeld, A., Arbib, M.A., and Alexander, A., 2002, The Neural Simulation Language NSL, A System for Brain Modeling, MIT Press (http://www.neuralsimulationlanguage.org).

Weitzenfeld, A., Peguero, O., and Gutiérrez, S., 2000, NSL/ASL: Distributed Simulation of Modular Neural Networks, in Proc of MICAI 2000: Advances on Artificial Intelligence, Acapulco, Mexico, April 10-14, LNCS 1796.

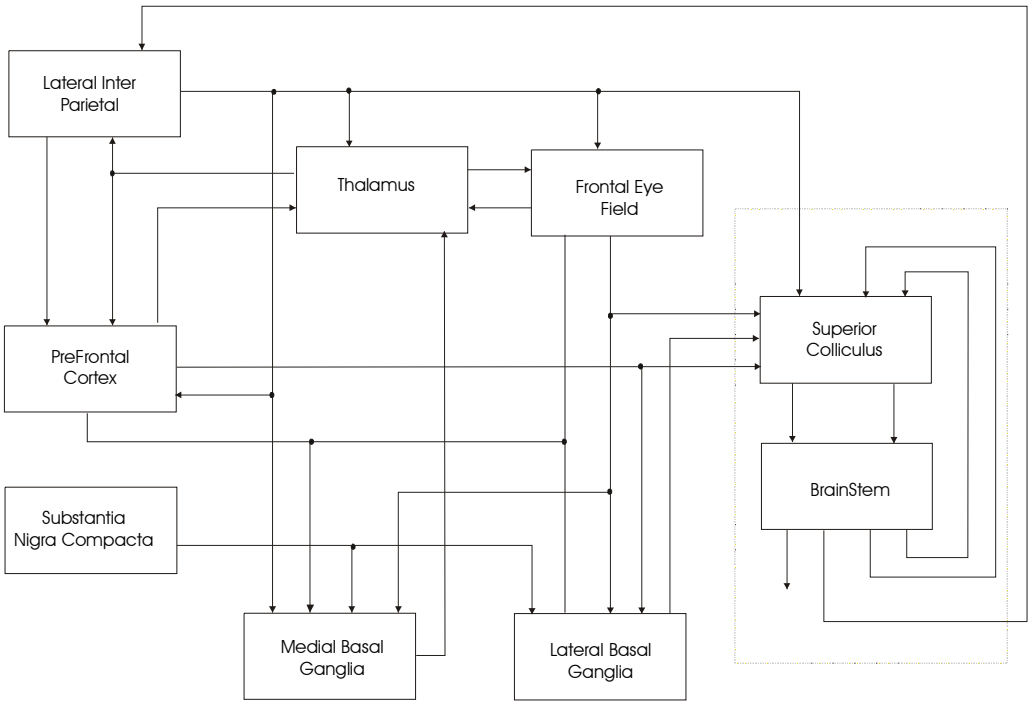
Yonezawa, A. and Tokoro, M., Eds., 1987, Object-Oriented Concurrent Programming, MIT Press.

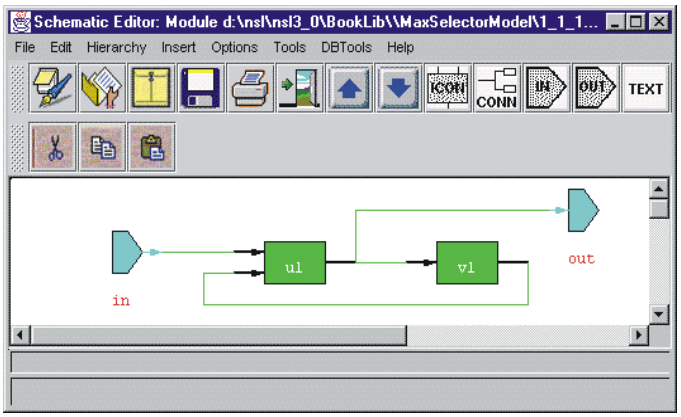
Figure Captions

Figure 1. The smaller outlined diagram shows a basic model for control of eye movements consisting of a *Superior Colliculus* (*SC*) and *Brainstem* modules, each representing a single brain region, responsible for generation of SACCADES (q.v.). As an example of the benefits of modularization, the *SC* and *BrainStem* modules can be embedded into the much larger and far more complex model of interacting brain regions, such as the *Crowley-Arbib* model of BASAL GANGLIA (q.v.) (Crowley et al., 2002).

Figure 2. The window shows the *Schematic Capture System* (SCS) view of the schematics of a sample module consisting of a two-module assemblage and two data ports. Modules are represented by rectangles, while entry (left) and exit (right) ports are represented by pentagon shaped icons.

Figure 3. The diagram shows a sample two-layer fully connected neural organization (see BACKPROPAGATION for an example of networks using such architectures). Each neuron is described by a single compartment represented by a value *up* or *vp*, its membrane potential respectively, and a value *uf* or *vf*, to its firing, the output from the neuron respectively. Input to the first neural layer is represented by *s*. Additionally, weights *w* have been added to the different connections.


Lateral Inter Parietal
Thalamus
Frontal Eye Field
PreFrontal Cortex
Superior Colliculus
BrainStem
Substantia Nigra Compacta
Medial Basal Ganglia
Lateral Basal Ganglia


Schematic Editor: Module d:\nsl\nsl3_0\BookLib\\MaxSelectorModel\1_1_1...
File Edit Hierarchy Insert Options Tools DBTools Help
ICON
CONN
IN
OUT
TEXT
ul
vl
in
out