NSL
NEURAL SIMULATION LANGUAGE[1]

Alfredo Weitzenfeld and Michael A. Arbib

Brain Simulation Laboratory, Center for Neural Engineering
University of Southern California, Los Angeles, CA 90089-2520
email: [alfredo, arbib]@cs.usc.edu
tel: 213/740-1176, fax: 213/746-2863

ABSTRACT

NSL, Neural Simulation Language, is a general purpose neural network simulation language and development system. NSL includes a high level language for describing neural networks, an interactive command interpreter, and powerful visualization tools. The simulator is designed and implemented using object-oriented programming methodologies. NSL provides a simulation platform for different types of applications, including both biological and artificial neural network based models, some of which are presented here. Presently, a number of research sites are involved in the development of new NSL based models, as well as in the extension and development of new libraries. The close interaction between these sites, together with the utilization of the system for teaching purposes, has provided a key part in the evolution of the system.

1 Introduction

NSL, Neural Simulation Language, is part of a broad research effort in the simulation of neural network systems at the Brain Simulation Laboratory of the University of Southern California. The NSL system, in development for several years, integrates state-of-the-art object-oriented programming methodologies in its design and implementation, providing (1) fast prototyping of new neural object types, (2) orthogonal object specification and implementation, (3) object data

[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.

encapsulation, and (4) code reuse. NSL provides a high-level language with many constructs and libraries developed to ease the specification of potentially large neural networks. NSL enables the modeling of any network which can be described as a set of dynamic equations to be updated in a continuous fashion. NSL also offers a simulation environment for users with both little programming background, as well as those with more extensive programming expertise, who are given the option to use C and C++ to extend its functionality. The flexibility of the system has been proven in its use in a variety of applications, including modeling of both biological and artificial neural networks.

NSL 2.1 [27] is offered as public domain software[2], being this the second release of the second generation NSL system, and currently the latest version. NSL 2.0 [26] and the original NSL 1.0 [25] are direct predecessors to NSL2.1. NSL has been extensively used at USC, as well as in several other research labs inside and outside the U.S.. The system has also been used as a teaching tool for neural network related courses both at USC and outside. We are currently in collaboration with the University of Bochum, Germany, and the University of Granada, Spain, in extending NSL and porting it to a more extensive number of platforms.

The latest release, NSL2.1, is characterized by

- an object-oriented simulation language with pre-defined object classes;
- a library with the most common neural network functions;
- numerical and learning methods libraries;
- methodologies for extending any of the libraries;
- a command language interpreter for interactive and batch processing;
- an interactive X windows graphical interface;
- temporal and spatial displays, 2D and 3D graphics.

2 System Overview

The system is composed of two main modules, shown in Figure 1,

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

The Simulator is composed of the following sub-modules,

- the **Processing Module**, where the network is fully processed (parallelism is simulated through synchronous update of neural components);
- 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 in an interactive fashion, including the graphics environment.

[2] NSL is available through anonymous FTP from 128.125.37.35. Please contact alfredo@usc.edu for further information, including system documentation.

The development of a simulation is carried out by the creation of two different types of files:

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

The Window Interface is composed of the following sub-modules,

- the **Graphical Displays**, where all window and graphics interaction takes place;
- the **Graphics Libraries**, containing all display functions and object libraries.

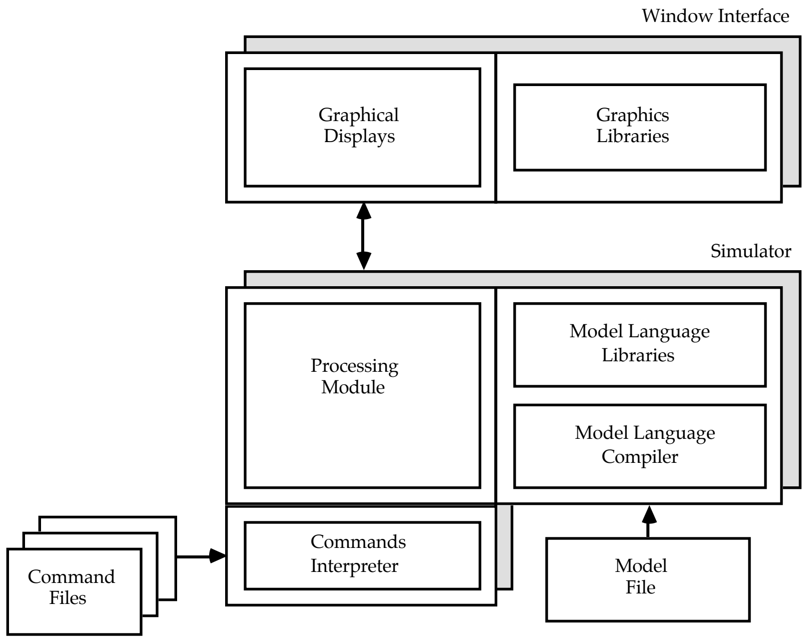

Figure 1 NSL Simulation System The system includes the Simulator, composed of the Processing Module, the Model Language Compiler and Model Language Libraries, and the Commands Interpreter. NSL also includes the Window Interface composed of the Graphical Displays Unit and the Graphics Libraries. Model and Command Files complement the system's functionality.

3 Example

As an introduction to the high-level simulation language, we present the 'Maximum Selector' network based on the Didday [4] model. This model uses competition mechanisms to obtain a single winner out of a set of external network inputs.

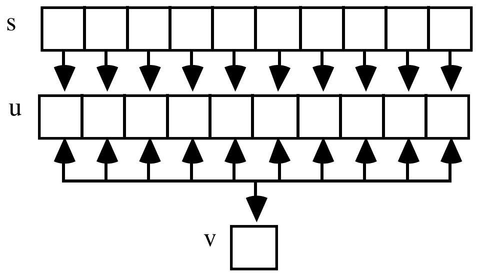

Figure 2 Didday model layers and interconnections Layer s, composed of 10 elements, stores the external input of the network. Layer u, also composed of 10 elements, receives point-wise excitatory input from Layer s, while receiving inhibitory input from the single element layer v, which itself gets excitatory inputs from layer u through a feedback loop.

The model includes an array of 10 cells, u, connected to a single inhibitory cell, v, having the circuit driven by a 10 cell input array s. Figure 2 shows the network.

The mathematical description of the model is given by

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

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

where

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

and

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

The model has the parameters w_1, w_2, h_1, h_2, τ_u, τ_v, with the restriction that $0 \le h_1$, and $0 \le h_2 < 1$, and $n = 10$ (see [2] for a full exposition).

3.1 Model File

In order to represent this and other models, we extend the basic neuron abstraction into neuron layers and connection masks, describing spatial arrangements among homogeneous neurons and their connections, respectively. One of 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. Furthermore, 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, while enabling efficient processing of large number of homogeneous interconnections as single primitives without requiring explicit element by element programming.

A model is represented by a set of NSL layer structures, and layer equations described in a number of modules; all these are stored in a file called '*model*.c', where *model* is the name of the model. In this example, the model is stored in a file called 'didday.c'. (In what follows, comments are preceded by '//', taking effect until the end of the line. ';' is the statement separator.)

3.1.1 Layers

The layer is the basic NSL structure, where two simple layers are used 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 non-linearity (e.g. a step, ramp, or sigmoidal function). A notational 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 (or layer) *x* is declared as DATA(*x*) if it has a single element, VECTOR(*x*,*n*) if it is one-dimensional with *n* elements, and MATRIX(*x*,*m*,*n*) if it is a 2-dimensional *m*x*n* array. Thus the layers in the model are declared as follows (preceded by the network's name, in this case 'DIDDAY'):

```
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
```

Declarations also include constants needed to define the model (whose values will be set in the interpreted file 'didday.nsl').

```
DATA(tu);
DATA(tv);
DATA(h1);
DATA(h2);
DATA(w1);
DATA(w2);
```

If a constant is "really" constant, we may write the numerical value directly into the program. However it is often good practice to name such model parameter (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.

3.1.2 Masks

The mask provides, through a single structure, the specification of regular interconnections among corresponding neural layers, thus enabling large number of neurons to be linked and processed at once by computing the 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 convolution operation

$$B = W*A$$

giving great computing power to a simple descriptive expression.

3.1.3 Integration

Integration in NSL is accomplished through the specification of the model layer dynamics by first-order differential equations of the form

$$\tau\, du/dt = f(u,x)$$

which corresponds in NSL to

$$\text{DIFF.eq}(u,\tau) = f(u,x)$$

where the differentiated variable u is followed by its time constant τ on the left hand side, and $f(u,x)$ is simply the right hand side of the differential equation rewritten in NSL syntax. The user specifies (from the '.nsl' file) alternative numerical methods to be used in approximating the solution of each differential equation in going from $u(t)$ to $u(t+\Delta t)$, such as *euler* or other.

3.1.4 Thresholding

The most common nonlinear functions that transform a neuron's membrane potential into its firing rate are provided by *thresholding* functions that are included in the NSL library. For example, in NSL,

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

is given by NSLstep(u), while

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

is given by NSLramp(s).

Other thresholding functions are available in NSL.

3.1.5 Modules

The description of the complete model is arranged into *modules*. NSL contains two classes of modules, the INIT_MODULE, and the RUN_MODULE. The INIT_MODULE contains model re-initialization statements, such as resetting layer values to zero. All modules begin with '{' and end with '}'.

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

The RUN_MODULE contains statements which are continuously executed as part of the simulation. In the DIDDAY network example, we specify the layer dynamics through two RUN_MODULEs, one called 'U' to specify how layers 'u' and 'U' are updated, and another called 'V' for layers 'v' and 'V'.

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

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

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.

In the example, the 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 class "method", using object-oriented methodologies.

3.2 Command Files

The simulation set up is provided in 'didday.nsl' in the form of simulation parameters, model parameters, including network input, and graphics.[3]

3.2.1 Simulation Parameters

In the first part of didday.nsl, the appropriate network is referenced, in this case 'DIDDAY'. The other entries include 'EULER' corresponds to the integration method, and a specification 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.

```
set network DIDDAY
set integration EULER
set delta 0.1
set end time 20.0
```

3.2.2 Model Parameters

The command file includes the setting of those network parameters whose values were not specified in the model file, didday.c.

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

[3] Upcoming NSL releases will extend the window interface to include further graphical interfaces to complement the existing command line interpreter.

Next, we specify the input to be used in the 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.[4]

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

3.2.3 Graphics

The output graphics are specified by windows to be used in displaying the input 'S' and the membrane potential 'u'. The network output is shown in Figure 3.

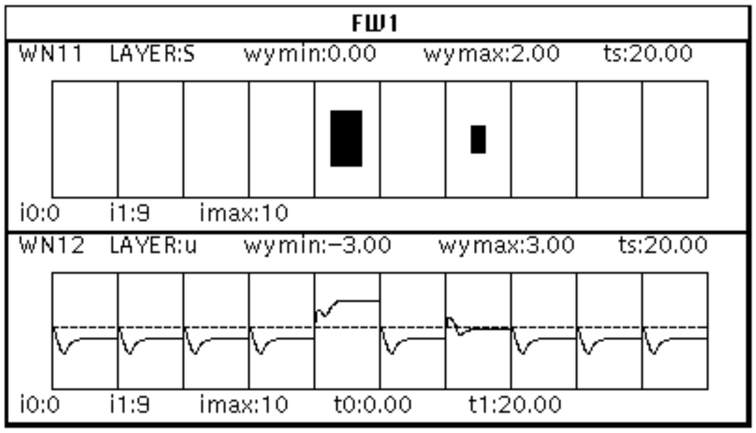
FW1
WN11 LAYER:S wymin:0.00 wymax:2.00 ts:20.00
i0:0 i1:9 imax:10
WN12 LAYER:u wymin:-3.00 wymax:3.00 ts:20.00
i0:0 i1:9 imax:10 t0:0.00 t1:20.00

Figure 3 Didday model output The window frame called 'FW1', contains a window 'WN11 '(top) which displays the vector 'S', with each element shown as an "area level graph"; spatial graph, and window 'WN12' (bottom) which displays the vector 'u', with each element shown as a "temporal graph", displaying the membrane potential values for each neuron in the layer through time.

4 Model Language

NSL is built following an object-oriented programming paradigm, incorporating a set of basic network object classes intended to give the user simple yet powerful building tools. These objects are complemented by expressions defined on them, as well as functions which can be applied to them. The language is then complemented by numerical and learning methods libraries.[5]

[4] NSL also includes a special input facility for bitmap inputs so complete array values do not have to be individually assigned.

[5] Refer to Weitzenfeld [27] for an extended description.

4.1 Object Classes

The three basic object types in NSL are: **NETWORK**, **MODULE**, and **LAYER**. **LAYER** is a super-class from which three data layer classes are derived: **DATA**, **VECTOR**, and **MATRIX** classes, differing according to the number of dimensions they have. **MODULE** is a super-class to **INIT_MODULE** and **RUN_MODULE**. **NETWORK** has no derived classes[6].

4.2 Expressions

Layer dynamics in NSL correspond to mathematical equations, describing the interactions among the different elements in the neural network. On the left hand side of the equation the layer being updated 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. Expressions may involve different kind of operations and may include function calls.

4.3 Library Functions

The layer function library includes arithmetic functions, such as convolutions, and neuron related ones, such as threshold functions. The user may define additional functions to the library.

4.4 Numerical Methods

NSL provides a library of numerical methods for the integration of differential equations

$$\mathbf{DIFF.eq}(u,t_u) = f(u,x)$$

where, again, the differentiated variable u is followed by its time constant t_u on the left hand side, and $f(u,x)$ is simply the right hand side of the differential equation rewritten in NSL syntax. NSL will then apply an appropriate numerical method (*euler*, etc.) at run time, as specified by the user.

While different numerical methods may be used to solve a particular neuron model, the neural network architecture and connection weights do *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

[6] Future NSL releases will provide for NETWORK derived classes useful in constructing more specialized new networks.

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.

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

$$\text{DIFF.eq}(u,t_u) = -u + S$$

for

$$f(u) = -u + S,$$

where 'S' is the input to the neuron layer.

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

$$u(t+dt) = (1 - dt/t_u))\, u(t) + (dt/t_u)\, S(t)$$

where 'dt' is the integration time step (delta).

Another numerical method, the 'interpolation' method replaces the above differential equation by

$$u(t+dt) = (p)\, u(t) + (1 - p)\, S(t)$$

where

$$p = \exp(-dt/t_u)$$

Other numerical methods are added as external libraries, such as Runge-Kutta.

4.5 Learning Methods

Learning methods, such as the back-propagation algorithm, are treated as external system libraries. As with numerical methods, other learning methods can be added to the existing NSL libraries by the user.

5 Applications

In the following sections some of the applications in which NSL has been used are described, including both biological as well as artificial neural networks models. Yet, these are only a few of many projects in which NSL is used.

5.1 Biological Neural Networks

The biological systems described here are most of them based on the *leaky integrator* neuron model[7]. We overview the frog's predator avoidance model [17], the frog's prey snapping model [18], the anuran retina model [23], and the saccade generation model [5]. Other models include the retino-tectal model [11], and an activity dependent model for the origin of receptive fields [1], where NSL is used to integrate a deterministic model for the development of the ganglion on-center cell receptive field dependent on both spontaneous activity and interactions among proximal cells.

[7] Other neuron models can also be simulated in NSL, such as compartmental ones.

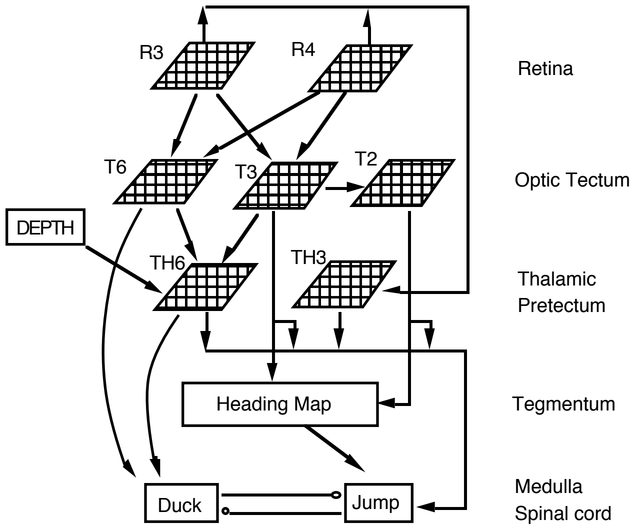

Figure 4 The looming-detection model The visual stimulus is transmitted to the network via R3 and R4 ganglion cells. T3 neurons detect movement of a looming object and T6 neurons monitor stimulus activity in the upper visual field. These tectal signals, along with depth information, converge onto the TH6 neurons which determine whether the visual stimulus is a looming threat. T2 neurons detects temporo-nasal movement whereas TH3 neurons are more sensitive to larger stimuli. The spatial signals conveyed by T2 and T3 neurons are integrated and transformed in the tegmentum to form a motor heading map which specifies the escape direction. The motor output depends on the size and elevation of a stimulus.

5.1.1 Frog's Predator Avoidance

The frog's predator avoidance model [17] is described by a neural network accounting for the detection and response to a looming (predator) stimulus. The generation of an appropriate response includes five tasks: detection of a looming stimulus, localization of the stimulus position, computation of the direction of the stimulus movement, determination of escape direction, and selection of a proper motor action. The detection of a looming stimulus is achieved based on the expansion of the retinal image and depth information. The spatial location of the stimulus is encoded by a population of neurons. The direction of the looming stimulus is computed by monitoring the shift of the peak of neuronal activity in this population. The signal encoding the stimulus location is gated by the direction-selective neurons onto a motor heading map which specifies the escape direction. The selection of a proper action is achieved through competition among different groups of motor neurons. The model is based on the analysis of predator-avoidance in frog and toad, but leads to a comparative analysis of mammalian visual systems. The model diagram is shown in Figure 4. Figure 5 shows the synaptic interconnections in the NSL model as well as a typical response to a looming stimulus.

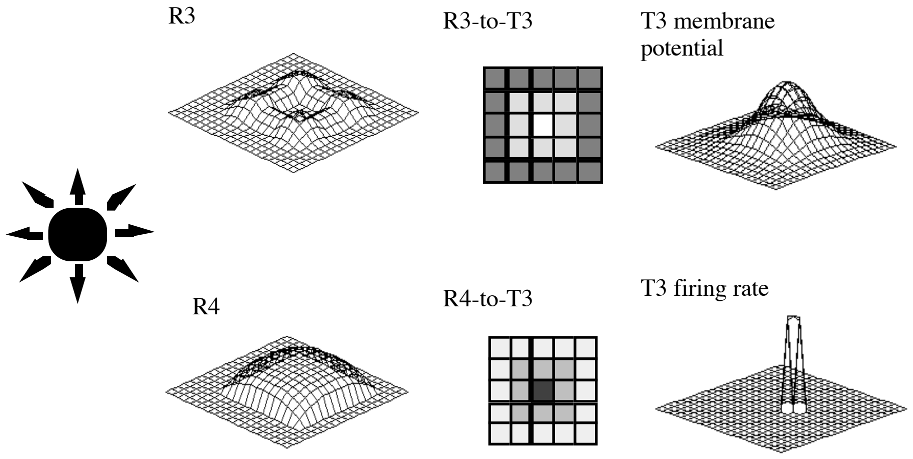

Figure 5 Synaptic connectivity of T3 neurons The R3 signals are projected through a radial mask to T3 neurons. The R3 cells respond to the leading edges of a stimulus in which the highest activity concentrates on the peripheral region (top of the second column). A radially arranged connectivity pattern (off-center, on-surround) is suitable to integrate activity from the periphery (top of the third column, the shading of each square indicate the synaptic weight). The R4 signals, which indicate the general dimming in the visual field (bottom of the second column), are projected to T3 neurons through a Gaussian mask (bottom of the third column). The position of a looming object is determined by localizing its center. The top graph in the last column is the membrane potential, and the lower one is the firing rate, of an array of T3 neurons. The T3 neuron with its center of mask aligned with the center of the looming stimulus would have the highest activation. Neighboring neurons

would have decaying activation as one moves away from the center. Thus the spatial position of a looming stimulus is encoded in a population of T3 neurons where the more central neurons have higher activations[8].

5.1.2 Frog's Prey Snapping

Snapping in anurans is, to a first approximation, a stereotyped motor pattern which follows transformations from visual processing to motor pattern generation [18]. This behavior is controlled by several extrinsic tongue muscles for flipping the tongue and by jaw muscles for opening and closing the mouth. The motoneurons controlling the tongue muscles are located in the hypoglossal nucleus in medulla. Prey recognition is performed in the optic tectum where the activity of T5.2 neurons is closely related to prey related visual stimuli. The transformation of the sensory signals into appropriate spatio-temporal patterns of activity in the motoneurons is postulated to be carried out in part by a motor pattern generator (MPG) in the medial reticular formation in the rostral medulla. Figure 6 shows a typical NSL simulation output.

[8] The 3-D graphical output taken from NSL is shown without the hidden line removal option.

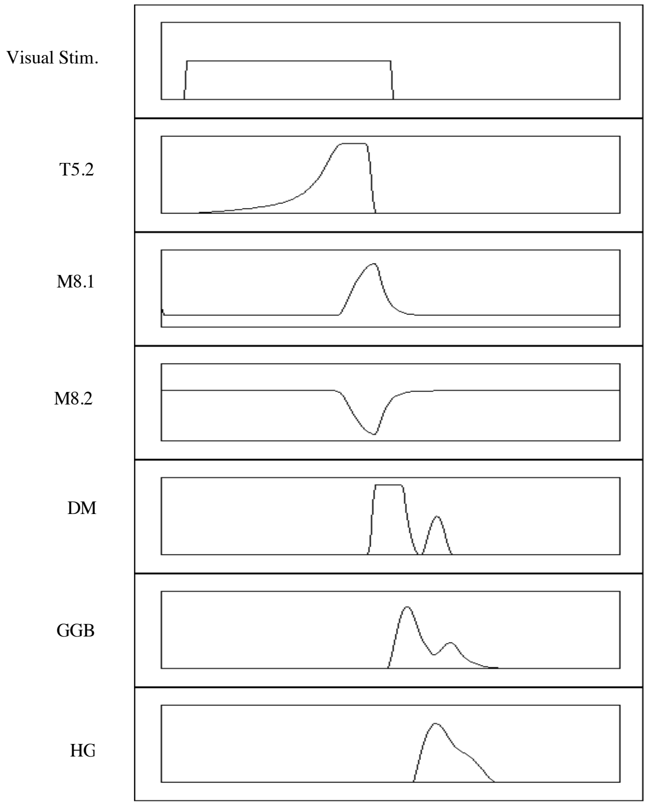

Figure 6 Behavior of snapping network The activity of the network in response to a prey-like stimulus lasting 450 ms. Note the slow build up of the discharge of T5.2 neurons which terminates before the peak of the activation of the protractor motoneurons, GGB. The temporal ordering of the activities in DM, GGB, and HG correspond closely to observed EMG data. Immediately following the peak in HG, a second, lower, activation in DM occurs, followed by a small activation in GGB.

5.1.3 Retina Model The anuran retina model [23] reproduces experimental data on the toad's R2, R3, and R4 ganglion cell in their response to moving worm, antiworm, and square-shaped stimuli of various edge lengths. In general, the average response of anuran ganglion cells to a moving stimulus depends on stimulus configuration, size, and velocity. A long thin bar moving in the direction of its long axis (a "worm" stimulus) will normally give a different response than the same sized stimulus moving perpendicular to its long axis ("antiworm"). Likewise, a square shaped stimulus will often generate a different response than do worm or antiworm stimuli.

5.1.4 Saccade Generation The saccade model [5] presents a biological-based system for the analysis of cortical and subcortical interactions in the control of saccadic eye movements. The system models the control of voluntary saccades to visual and remembered targets in terms of in teractions between posterior parietal cortex (PP), frontal eye fields (FEF), the basal ganglia (caudate and substantia nigra), superior colliculus (SC), mediodorsal thalamus, and the saccade generator of the brainstem. Interactions include the modulation of eye movement motor error maps by topographic inhibitory projections; dynamic remapping of spatial target representations in saccade motor error maps and temporal target representations in the brainstem to correct for intervening eye movements; and sustained neural activity that embodies spatial memory. The saccade experiments consist of a monkey seated in a primate chair with its head fixed and eyes free to move. Illuminated fixation points and saccade targets are presented on a visual screen in front of the primate. Microelectrodes record (and stimulate) neural activity during the tasks, while eye movements are concurrently monitored and recorded. The simulations include simple saccades, memory saccades, double saccades, compensatory saccades, lesion studies, and colliding saccades.

5.2 Artificial Neural Networks

Among the artificial neural network applications in which NSL has been utilized, we overview the following models: back-propagation [10], Dynamic Link Architecture [20], reaching and grasping [12], including dynamic programming and optimal control models [13; 3; 14], conditional learning in monkeys for visuo-motor behavior [7], and a model which integrates NSL as part of a complex robotics system [8]. Other models include topological maps [21], motor-program learning within a reinforcement-producing environment [9], a neural network model for extracting visual motion reversal invariant events [19], learning and recognition of complex temporal sequences [22], and the classification of nonlinear processes [24].

5.2.1 Back-Propagation

One of the most widely used learning models for artificial neural networks, the back-propagation algorithm, is included in NSL [10]. Without getting into the particulars of the model, we emphasize the generality of NSL by incorporating back-propagation as part the system's external libraries. Other "classical" learning algorithms are currently in development.

5.2.2 Dynamic Link Architecture

The Dynamic Link Architecture (DLA) [20] has been simulated in NSL, in its principal application to face recognition. In contrast to other "learning" systems, DLA is based on a stochastic self-organization process occurring within a single pattern perception and on rapidly modifiable links used to express adaptive binding of features[9].

[9] DLA simulations also by M. Misra and J.-M. Fellous.

5.2.3 Reaching and Grasping

A model of two-dimensional reach and grasp, under normal and perturbed conditions, including preshape and enclose of the hand [12].

In the implementation, NSL modules have been used to conveniently partition controller, plants, and duration computation elements. Normal and perturbed transport and preshape trajectories are based on optimality principles for movement efficiency (smoothness) with a penalty for aperture added to preshape. Delays in information flow between sensorimotor programs for reach and grasp affect timing and kinematics of these actions. In reaching to grasp and reaching to point, the different tasks put different constraints on the final state of the hand. A mathematical *orientation constraint* at the end of reach realistically models reaching in the context of prehension.

The feedback controller in the model has been implemented in different ways. One approach involves the use of linear quadratic dynamic programming [13], in which the Ricatti equation is utilized, integrated in reverse time, to give the optimal feedback control, then used through forward integration to simulate optimal control of a linear plant. The model integrates a model of variability in movement with a model of continuous control under delayed feedback to reproduce findings on accuracy and trajectory in movements of constrained accuracy. An alternative to this model [3] where an adaptive controller generates an optimal trajectory given cost feedback, rather than through computed trajectories. Figure 7 shows sample output from the NSL simulations.

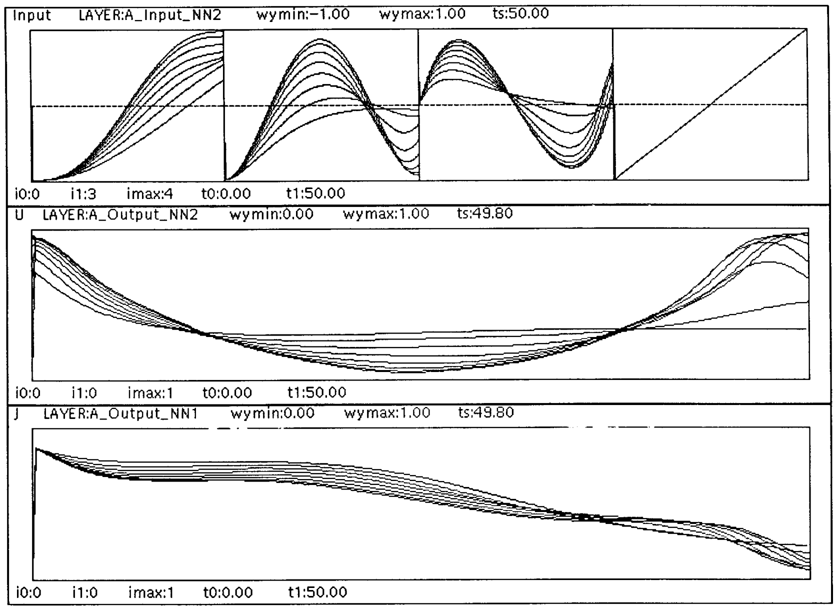

Figure 7 Reaching and Grasping Model Overlay graphs of the NSL simulation of HDP (Heuristic Adaptive Critic) architecture learning minimum jerk trajectory. Six superimposed iterations show convergence to solution. Top, left to right: Position, velocity, acceleration, and time. Middle: Jerk. Bottom: Cost-to-go.

Another related model is reaching with a dynamic limb [14]. In this model, trajectories were generated using a "minimum-torque-change" optimization method, which was solved numerically through the integration to NSL of an external numerical package written in Fortran. Furthermore, a numerical two-point boundary value problem solver was added to NSL.

5.2.4 Conditional Learning

An interesting model for primate visuo-motor conditional learning [7]. The model matches observations of behavior and neural activity in premotor cortex of monkeys learning to pair an arbitrary visual stimulus with one of a set of previously learned behaviors by simulating a network comprising a large number of "motor selection columns." Reinforcement-learning is used to recognize new visual patterns and acquire the appropriate visual-motor conditions. The architecture employs a distributed representation in which a single pattern is coded by a small subset of columns. A column is initially able to respond to many different inputs; as it learns to trigger a motor program, its responses become more narrowly defined. Each column's output is a set of votes for the various motor programs. The votes for each program are collected by "selection units" which drive a winner-take-all circuit to determine whether or not a particular motor program is executed. The model is successful in reproducing the sequence of behavioral responses given by the monkeys, as well as a number of phenomena that have been observed at the single-unit level.

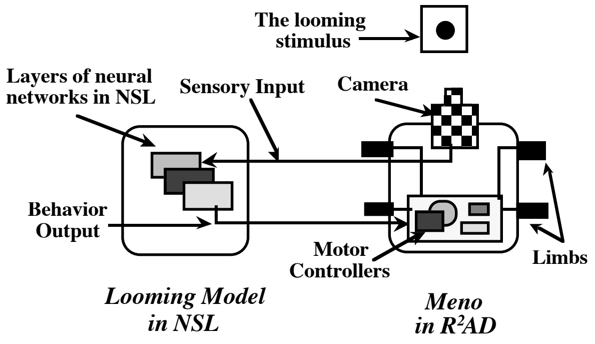

Figure 8 Distributed Robotics System This schematic drawing illustrates the interaction between the perception and action systems. To the left is the sensory processing looming perception model in NSL which receives visual inputs from Meno[10] on the right and produces neural activities in the presence of a looming object. This population firing is then sent to Meno's robot controller which in term generates a command to move the limbs synchronously to avoid the moving object.

5.2.5 Robotics

Besides simulation of "stand-alone" systems, NSL has been used as one component in more complex systems. In particular, a model of sensorimotor integration in the development of real-time robotics application is presented in [8]. This work discusses realistic biological behaviors by providing a close integration between the perception of the external environment and the generation of actions. In this setup, shown in Figure 8, NSL models classes of neural networks performing different aspects of sensory processing, while the R^{2}AD [6] system furnishes a flexible robotics application development platform to monitor dynamically activated processes during robot program executions. Although these two developmental systems were designed and implemented independently, the internal software architecture in each system allows external access and exchange of data and control information. The perception aspect is based on the looming model [16] where NSL extracts the center of the looming object and perceives its looming direction in order to activate the proper robot behavior procedure.

6 Conclusions

NSL is an evolving project at the University of Southern California with cooperative efforts at the University of Bochum, Germany, and the University of Granada, Spain. The present system, NSL2.1, is part of a growing effort in the simulation of complex neural-based models. NSL2.2, to be released

[10] Meno is a four legged autonomous robot developed in the Robotics laboratory at USC.

summer 1993 will incorporate many new features, particularly a more extensive window interface and the ability to run on a larger number of platforms. The next generation of the system, NSL3.0, already in development, will offer a distributed and parallel simulation environment. The simulation model is based on research described in [29], where neural entities are actually mapped to different object classes [28]. The system will present a unified computational model for *schemas* (coarse-grained computational agents) and neural networks (fined-grained computational agents) based on concurrent object-oriented programming methodologies. This will provide the modeling abstraction required for enabling the simulation of complex hierarchical applications, while permitting the integration of different programming paradigms with that of neural networks. Furthermore, independently developed neural networks could be integrated under this design. As a last note, we would like to emphasize the generality of the system which transcends the area of neural networks simulation, with applications in, e.g., genetic algorithms [15].

7 Acknowledgments

We wish to thank all the members of the USC Brain Simulation Laboratory, the USC Robotics Laboratory, and the rest of our contributors and users who have made this article possible, and who have provided a key part in the evolution of NSL.

8 References

[1] Andrade, M.A., Moran, F., 1993, Activity Dependent Model for the Origin of Receptive Fields, Technical Report, Dept. of Biophysics, Universidad Complutense, Madrid, Spain (in preparation).

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

[3] Borghese, N.A., Hoff, B., Arbib, M.A., 1993, Optimal Control of Reaching Movements Using Neural Networks (submitted).

[4] Didday, R. L, 1976, A model of visuomotor mechanisms in the frog optic tectum, Math. Biosci. 30:169-180.

[5] Dominey, P.F., Arbib, M.A., 1992, A Cortico-Subcortical Model for Generation of Spatially Accurate Sequential Saccades, Cerebral Cortex, 2:153-175.

[6] Fagg, A.H., Lewis, M.A., Iberall, T., Bekey, G., 1991, R^{2}AD: Rapid Robotics Application Development Environment, Proc. of the IEEE Conf. of Robotics and Automation, pp. 1420 - 1426.

[7] Fagg, A.H., Arbib, M.A., 1992, A Model of Primate Visual-Motor Conditional Learning, Adaptive Behavior, 1:3-37.

[8] 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, Proc. of IJCNN '92, Baltimore, MD.

[9] Fagg, A.H., 1993, Motor Program Learning within A Reinforcement-Producing Environment, Technical Report, Center for Neural Engineering, USC, Los Angeles, California (in preparation).

[10] Fagg, A.H., 1993, Back-Propagation in NSL, Technical Report, Center for Neural Engineering, USC, Los Angeles, California (in preparation).

[11] Fellous, J.-M., 1993, Retino-Tectal Model in NSL, Technical Report, Center for Neural Engineering, USC, Los Angeles, California (in preparation).

[12] Hoff, B., Arbib, M.A., 1993, Simulation of Interaction of Hand Transport and Preshape During Visually Guided Reaching to Perturbed Targets, J. Motor Behavior (in press).

[13] Hoff, B., Arbib, M.A., 1993, Stochastic Optimal Control Model of the Effects of Movement Time on Accuracy in Reaching Movements (in preparation).

[14] Hoff, B., 1992, A Computational Description of the Organization of Human Reaching and Prehension, Tech Report USC-CS-92-523, USC-CNE-92-02.

[15] Lewis, M.A., Fagg, A.H., Solidum, A., 1992, A Genetic Approach to the Development of a Neural Network for the Control of a Walking Machine, Proc. of the IEEE Conference of Robotics and Automation, pp. 2618-23, Nice, France.

[16] Liaw, J.-S., Arbib, M.A., 1990, A Neural Network Model for Response to Looming Objects by Frog and Toad, in Visual Structures and Integrated Functions, (M.A. Arbib and J.-P. Ewert, Eds.), Research Notes in Neural Computing, Springer-Verlag.

[17] Liaw, J.-S., Arbib, M.A., 1993, Neural Mechanisms Underlying Direction-Selective Avoidance Behavior, Adaptive Behavior (in press).

[18] Liaw, J.-S., Weerasuriya, A., Arbib, M.A., 1993, A Neural Network Model for Snapping in Frogs and Toads (submitted).

[19] King, I.K., Arbib, M.A., 1992, A Neural Network Model for Extracting Visual Motion Reversal Invariant Events, Proc. of IJCNN Conf, Beijing, China.

[20] Konen, W., von der Malsburg, C., 1992, Learning to Generalize from Single Examples in the Dynamic Link Architecture, Technical Report, University of Bochum, Germany.

[21] Merelo, J.J., 1993, Topological Maps in NSL, Technical Report, University of Granada, Spain (in preparation).

[22] Tanaka, T., 1992, A Complex Sequence Recognition Model, Proceedings of IJCNN '92, IV:202-207, Baltimore, MD.

[23] Teeters, J.L., Arbib M.A., Corbacho F., Lee H. B., 1993, Quantitative modeling of Anuran Retina: Stimulus Shape and Size Dependency (submitted).

[24] Vermeersch, L., Vanrolleghem, P., 1993 Feature Based Model Identification of Non-Linear Biotechnological Processes, Proc. of ISEM 8th Intl. Conf. on the State-of-the-Art in Ecological Modeling, Kiel, Germany (in press).

[25] Weitzenfeld, A., 1989, NSL: Neural Simulation Language, Version 1.0, CNE-TR 89-02, University of Southern California, Center for Neural Engineering, Los Angeles, CA.

[26] Weitzenfeld, A., 1990, NSL: Neural Simulation Language, Version 2.0, CNE-TR 90-01, University of Southern California, Center for Neural Engineering, Los Angeles, CA.

[27] Weitzenfeld, A., 1991, NSL: Neural Simulation Language, Version 2.1, CNE-TR 91-05, University of Southern California, Center for Neural Engineering, Los Angeles, CA.

[28] Weitzenfeld, A., Arbib, M., 1991, A Concurrent Object-Oriented Framework for the Simulation of Neural Networks, Proceedings of ECOOP/OOPSLA '90 Workshop on Object-Based Concurrent Programming, OOPS Messenger, 2(2):120-124, April.

[29] Weitzenfeld, A., 1992, A Unified Computational Model for Schemas and Neural Networks in Concurrent Object-Oriented Programming, PhD Thesis, TR 92-03, Center for Neural Engineering, University of Southern California, Los Angeles, California, September.