Reading assignment:

[NSLbook]: Chapters 0, 1, 2

A. Weitzenfeld, M.A. Arbib and A. Alexander, 2000,
NSL Neural Simulation Language, MIT Press (in press).
[www-hbp.usc.edu/_Documentation/NSL/Book/TOC.htm]
NSL Neural Simulation Language
Version 3.0.n

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 graphical interface.
* Temporal and spatial displays.
* 2D and 3D graphics.
* The flexibility of a system written in the Java™ environment.
User Expertise

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

- the basic level requires familiarity with NSL 'high level' language and there is no need to know Java™, C or C++;

- the advanced level requires some basic understanding of Java™, allowing great flexibility on the type of models which can be described, as well as permitting communication with other software tools.

In this class: we will use both basic and advanced levels.
- Compiled language **NSLM** for definition of modules and models. Relies heavily on Java™ (or C++) syntax;

- Scripting language **NSLS** for automation of simulations. Extends Tcl/Tk scripting language;

- Graphical interface for running simulations;

- Schematic editor **SCS** to graphically define and interconnect modules.
A Basic Neuron

Its membrane potential $m$ has time course described by a differential equation

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

depending on its input $S_m$. For example, the 'leaky integrator' is described by the differential equation

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

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 $\sigma$ could be, for example, a sigmoid function.
The most common formula for the input to a neuron is

$$Sm = \sum_{i=1}^{n} wi \cdot M_i[t]$$

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 $wi$ is the weight on that link.

$$S1 = W21 \cdot M2 + W31 \cdot M3$$

Neurons may be modeled with different levels of detail.
Layers and Masks

When a model has many neurons, naming and describing each one becomes impractical.

We often find neural networks structured into two-dimensional layers, with regular connection patterns between various layers.

\[ B = W \ast A \]

i.e.,

\[ B(i,j) = \sum_{k=-d}^{d} \sum_{l=-d}^{d} W(k,l)A(i+k,j+l) \]
Modular Model Definition in NSL 3.0

Module Level 1

Module Level 2

Module Level

Neural Network Level
This modular, object-oriented approach allows principled and organized development of possibly large-scale neural network models.

PreFrontal Cortex Module:
NSL 3.0 API

Allows programmer to concentrate on neural network architecture rather then on implementation details, in particular by providing:

- Neuron models and numerical methods for simulation of differential equations
- Simulation control and graphing tools (NSL book Chap 5)
Java™ Refresher in Four Slides

- Object-oriented programming: a “class” defines data structures as well as methods (functions) to operate on those structures

```java
public class Foo {
    int x, y; // integer data
    float z[]; // float data array

    public Foo() { init(); } // constructor

    private void init() {
        x = 0; y = 1; z = new float[100];
        for (int i=0; i<100; i++) z[i]=0;
    }
}
```

- Inheritance using “extends” keyword: a new class which extends an existing one has the data and methods from the existing class built-in; these inherited methods can be overloaded by defining new ones with same names

```java
public class Bar extends Foo {
    public reset() { // new method in Bar
        init(); // inherited from Foo
    }
}
```
- No prototypes and no `#include`. Shorthand references to other classes (not methods!) possible using `import`

```java
java.rmi.server.RemoteObject ro; // full ref

or

import java.rmi.server.*; // at top of file
...
RemoteObject ro; // compiler searches imported packages for that class
```

- Memory allocation either when first declaring a variable or later using keyword `new`. Automatic garbage collection and no need to free memory.

- No pre-processor, no macros and no `#define`. Constants are declared as `final` variables.

- Basic data types, operators, control statements, conversions, casting and scoping like in C and C++.

- No pointers; function arguments implicitly passed by ref.

```java
private float func(int x[]) {
    int y = some_function(x[3]);
    float z = ((float)y) * 1.2F;
    return z;
}
```
Basic types: byte, short, int, long, char, float, double, boolean

Arithmetic operators: +, -, *, /, %, ++, --, +=, -=, *=, /=

Comparison operators: <, >, ==, !=, <=, >=

Bitwise operators: &,, |, ^, <<, >>, >>>, ~,
 &=, |=, ^=, <<=, >>=, >>>=

Logical operators: &&, |, ^, !

Control statements:
if (expr) block [else if (expr) block] [...] [else block]

expr ? block : block;

while (expr) block

do block while (expr);

switch (expr) { [case expr: [...] block [break;]]
 [default: block] }

for (init; test; incr) block

For more information:
NSL Book Chapter 6
http://java.sun.com/docs/books/tutorial/java/index.html
http://www.afu.com/javafaq.html
A Simple (empty) NSLM Module

NSLM is very similar to Java, with some added macro-constructs to be translated by the NSLM compiler into semantically heavier Java constructs. Also, the NSLM API provides neural-oriented classes, operators and functions.

```java
nsllImport NslAllImports;  // NSLM macro-import

nsllModule MyModule(int s)extends NslModule() {
    public NslDouble1 x(s);  // 1D array

    // the following is called when module
    // is instantiated:
    public void initModule() {
        // do some initialization
    }

    // the following is called at beginning
    // of a simulation run:
    public void initRun() {
        x = 0.0;  // NSLM array init
    }

    // the following is called at each time
    // step during simulation:
    public void simRun() {
        // update time-dependent vars
    }
}
```
Numerical methods

The 'Euler' method replaces the differential equation

\[ \tau m \frac{dm(t)}{dt} = f(S_m,m,t) \]

by

\[ \tau m \frac{m(t+\Delta t)-m(t)}{\Delta t} = f(S_m,m,t), \quad \text{i.e.,} \]

\[ m(t+\Delta t) = m(t) + \frac{\Delta t}{\tau m} f(S_m,m,t) \]

Very simple and easy to implement, but only first order accuracy and computationally inefficient.
The second-order Runge-Kutta method uses an Euler step to help compute the function and its derivative at $t+\Delta t/2$, then uses that derivative across the whole step.

Its error is $O(\Delta t^3)$, i.e., second-order method. But at the additional cost of one more evaluation of the function and its derivative (at the mid-point).

In NSL, users can specify integration method and time step.
NSL Scheduler

We saw that models are simulated using discrete time steps. “Multi-clock” scheduler in NSL allows various modules to be simulated with various time steps.

1. For all modules execute `initSys` followed by `callFromConstructorTop`, `makeInst`, and `callFromConstructorBottom`.
2. For all modules execute `makeConn`.
3. For all modules execute `initModule`.
4. For all modules execute `initTrainEpochs`.
5. For all modules execute as many `numTrainEpochs` as specified in `initSys`, `initModule`, or `initTrainEpochs`.
   - Execute the `initTrain` method once per epoch.
   - Execute the `simTrain` method `numTrainCycles`.
   - Execute the `endTrain` method once per epoch.
6. For all modules execute `endTrainEpochs`.
7. For all modules execute `initRunEpochs`.
8. For all modules execute as many `numTrainEpochs` as specified in `initSys`, `initModule`, or `initRunEpochs`.
   - Execute the `initRun` method once per epoch.
   - Execute the `simRun` method `numRunCycles`.
   - Execute the `endRun` method once per epoch.
9. For all modules execute `endRunEpochs`.
10. For all modules execute the `endModule` method for each module.
11. For all modules execute the `endSys` method for each module.
Learning Methods

An important part of neural network modeling is to be able to introduce learning in a model.

Hebbian learning

reinforce connection between co-activated neurons

Back propagation

reinforce connections which positively contributed to a correct answer

We will analyze these techniques in subsequent lectures. NSL provides an environment to setup and execute training and simulation runs in an automated manner.
A Simple NSLM Module

```java
import NslAllImports;  // NSLM macro-import

module MyModule() extends NslModule() {
    public NslDinDouble1 in();  // input arr
    public NslDoutDouble0 f();  // output
    private NslDouble0 v();  // membrane pot
    private double tau;  // time const

    public void initSys() {
        system.setRunEndTime(10.0);
        system.nslSetRunDelta(0.1);
    }

    // called when simulation run starts:
    public void initRun() {
        v = 0.0; f = 0.0; tau = 1.0;
    }

    // called each time step during simul:
    public void simRun() {
        // tau*dv/dt = -v + sum(in):
        v = nslDiff(v, tau, -v + nslSum(in));

        // apply step fct to v to get f:
        f = nslStep(v);
    }
}
```
NSLM Method Example:
Activation Functions

See Appendix of NSL Book for complete references!
Instantiation & Execution of Complex Models

The NSL executive:

- Instantiates top-level module (model), and its sub-modules, in a depth-recursive manner.
- Adds each module to the execution queue as it is instantiated.
NSLS: The Scripting Language of NSL

Let’s start all over, but now with the TCL language as a base!

When the NSL environment is started, it includes an “NSL Executive” TCL shell window that can accept interactive user commands in NSLS language.

NSLS provides the following functionality:

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

```
nsls% command        # a TCL command
...                   ...
nsls% nsl command     # an NSLS command
...                   ...
nsls% nsl help command # NSLS online help
```
TCL Basics in Two Slides

- Sloppy shell-style interpreted scripting language, non-typed (char or number type of variables determined by their initial assignment; no need to declare variables).

```tcl
set x 1  # numerical assignment
set s "TCL example" # string assignment
set y $x  # assignment from variable
puts "x is $x, y is $y and s is $s"
set t \[expr $x * 100 / (5 + cos($y))\]
```

- Arrays are not indexed by numerical values, but by strings (associative arrays; index does not need quotes)

```tcl
set arr(test) 5
set arr(1,2) 10  # fake 2D index
                 # [CAUTION: string index]
```

- or you can use lists

```tcl
set vec {10 5 20 12}  # constants
set vec2 [list 1 2 $x] # variables/evals
set v3 [lindex $vec 3] # access element
puts "vec[3] is $v3"
```
Basic types: untyped! (actually, string, int or double)

Arithmetic operators: +, -, *, /, %

Comparison operators: <, >, ==, !=, <=, >=

Bitwise operators: &, |, ^, <<, >>, ~

Logical operators: &&, ||, ^, !

Control statements:
  if {exp} block [elseif {exp} block] [...] [else block]

  expr { exp ? block : block }

  while {exp} block

  switch {exp} { [exp block]
     [default block] }

  for {init} {test} {inc} block

  foreach var {list} block

Procedures:

  proc name {args} block

For more information:
NSL Book Chapter 7
http://hegel.ittc.ukans.edu/topics/tcltk/tutorial-noplugin/
http://www.cujo.com/tcl_tut.html
NSLS Adds Neural Net Functions to TCL

Set/get MSLM variables, scalar or arrays using

“nsl set” and “nsl get”

Access NSLM objects via absolute reference

```
nsl get model.obj1.obj11.obj111
```

or relative reference from `varpath`:

```
nsl set varpath model.obj1.obj11
nsl get obj111
```

Control simulations

```
nsl initRun
nsl simRun
...
```

Modify input, read/write data from/to files, etc…

We will explore these uses of NSLS in further lectures.
Next time…

We will use the winner-take-all model of Amari and Arbib as our first example of NSLM code being acted upon using the NSL graphical environment and NSLS.