Running the LDnet simulator (version 1.5)

LDnet simulates a continuous-time, continuous-state dynamical system for performing optimization of a Harmonic grammar defined over tensor-product connectionist representations. Once you define a model (by specifying a **domain** file and **settings** file), LDnet can be used to simulate the dynamics of the model in response to a set (or sequence) of **stimuli** you specify, running the set of stimuli for a specified number of **repetitions**. Results of all repetitions of all stimulus sets (or sequences) are stored in MATLAB arrays to facilitate subsequent analysis.

To create a model and run a simulation:

- Set your MATLAB (or OCTAVE) path (Section 2).
- Create a **domain file** and a **settings file** (Section 3)
- Create a 3D array of **stimuli** (Section 4).
- Call the **runLDnet** function to run the simulation (Section 5).
- Plot and analyze your results (Section 6).

1. Understanding the simulation

This documentation assumes the reader has background knowledge of basic connectionist concepts such as units, activations and weights, as well as tensor-product representations.

LDnet simulates a recurrent network whose units have continuous, real-valued activations in an unbounded range (-Inf, +Inf). The units are connected by pairwise symmetric weights, have biases, and receive external inputs. There are no hidden units. The external inputs are the way that a **stimulus** is provided to the network.

The vector of activations of all units in the network at time *t* is referred to as its **state** and is denoted *s(t)*. For a given set of weights, biases and external inputs, each state has a **Harmony** value that is a measure of its well-formedness, or the degree to which it "agrees" with the weights of the network. The Harmony function is a quadratic function of the state, determined by the weights, biases and external input to the network:

$$H(s) = \left(\frac{1}{2} s^T W s + b^T s + i^T s \right) + bowl(s) \qquad (1)$$

where *s* is the state vector, *W* is the weight matrix, *b* is the vector of biases, and *i* is the stimulus input vector. The user defines the weight matrix and bias vector in a **Domain file** (to be explained in Section 3.1), and defines the stimulus input vector (Section 4). The term *bowl(s)* is a quadratic "upside-down bowl" of Harmony whose sole purpose is to ensure that there is a single Harmony maximum. It is unbiased with respect to the TP states, assigning them equal Harmony. See Section 1.4 for more details.

$$bowl(s) = \left(\frac{1}{2} s^T W_b s + b_b^T s \right) \qquad (2)$$

where

$$W_b = -qP^{-T}P^{-1}$$
$$b_b = qzP^{-1}\vec{1}$$

1.1 Activation dynamics: Optimization plus Quantization

The network activation dynamics are a continuous dynamical system that attempts to find a Tensor-Product representation that maximizes the Harmony function, given the current external input. Because not all states of the network correspond to TP representations, the network must simultaneously solve two problems: (1) find high-Harmony states, and (2) find TP states. It achieves this by combining two dynamical systems that each try to achieve one of the goals. The dynamics are described by a stochastic differential equation with three additive terms:

1. An **optimization** term that attempts to maximize Harmony via gradient ascent on the Harmony function, without regard to which states correspond to TP representations.
2. A **quantization** term that attempts to move the system toward states that corresponds to exact tensor-product representations, without regard to Harmony.
3. A **stochastic** term (simple Brownian motion) that randomly perturbs the state to help the system escape local maxima of the Harmony function.

For each unit *i*,

$$ds_i = \lambda \frac{dH(s)}{ds_i} dt + (1-\lambda)Q(s)_i dt + \sqrt{2T}\, dB_i \qquad (3)$$

where *H(s)* is the Harmony of state *s, Q(s)* is the quantization dynamics, *dB* is the Brownian motion process, and *T* is the **computational temperature** parameter. Note that the first two terms are weighted by two coefficients that sum to one.

In the discrete-time simulation implemented in the program, equation (3) must become:

$$\Delta s_i = \lambda \frac{dH(s)}{ds_i} \Delta t + (1-\lambda)Q(s)_i \Delta t + \sqrt{2T\Delta t}\, N_i(0,1) \qquad (4)$$

where each $N_i(0,1)$ is a unit Normal random variate.

The optimization term (Term 1) is the gradient of the quadratic Harmony function. In the quantization term (Term 2), *Q(s)* is a second-order equation known as the **Competitive Lotka-Volterra Dynamics** (LV dynamics). LV dynamics can be used to implement winner-take-all competition in a network that uses localist, real-valued units. But though a linear transformation, we use it here to implement competition between the distributed fillers within each role. The quantization process therefore consists simply of winner-take-all competition between the distributed fillers within each role, without regard to their Harmony. The stochastic term is present because the combined dynamics of Terms 1 and 2 will in general contain local optima, which should be avoided in favor of global optima. The variance of the stochastic process is controlled by the computational Temperature parameter, which is gradually reduced over time (an instance of Simulated Annealing).

Because not all states correspond to TP representations, the optimization and quantization terms are typically in conflict. The conflict is resolved by gradually shifting the relative influence of the two terms over time, with the optimization term initially having the greater influence, and gradually shifting control to the quantization term over time. This is achieved by gradually reducing the parameter λ from 1 down to 0 over the course of the simulation.

1.2 Stationary distribution

For fixed input to the network, if $\lambda = 1$, then LDnet corresponds exactly to a *Continuous Boltzmann Machine*, also called a *Symmetric Diffusion Network*. In this case, if the weight matrix $\left(W + W_{bowl}\right)$ is negative definite (which is the case if q is sufficiently large), then the network will have a stationary probability distribution over its state space, in which the probability of finding the network in any state *s* is proportional to the exponential of that state's Harmony:

$$P(s) \propto \exp(H(s)/T) \qquad \text{(for fixed input and } \lambda = 1\text{)} \tag{5}$$

Importantly, the addition of the quantization term alters this relationship, introducing the additional preference that the state should be a TP state.

1.3 S-space and c-space

The coordinate system whose axes correspond to the units of the network is referred to as the **state space** of the network, or **s-space** for short. A state of the network is a single point in this space, and can be written as a vector of activation values. Due to the use of TP representations, the meaning of individual activations in LDnet can be obscure. So it is sometimes convenient to view the state of the network in **constituent space,** or **c-space**. In this coordinate system, there is one axis for every constituent (possible role/filler pair), and the meaning of a dimension *c* is the "effective activation" of constituent *c*. Because LDnet requires that all role and filler vectors be linearly independent, there is a one-to-one mapping between s-space states and c-space states, expressed by the linear mapping:

$$s = Pc \tag{6}$$

where the matrix P is a square matrix whose columns consist of the TP representations of the constituents. The effective activation of some constituent c_i is then given by the equation $c = P^{-1}s$. The simulation provides Matlab functions for translating back and forth between s-space and c-space.

During each repetition, LDnet periodically displays the current state of the network in c-space as a [nFillers x nRoles] "effective activation" matrix.

1.4 Equation for the bowl

In *c-space,* the bowl is a radially-symmetric parabolic surface with a single maximum at the point $z\vec{1}$ (the value *z* on every dimension), with curvature $-q$:

$$bowl(c) = q\left(-\frac{1}{2}cc^T + z\vec{1}\right) \tag{7}$$

We refer to q as the *strength* of the bowl. $bowl(s)$ is the transformation of this c-space bowl into s-space in such a way that Harmony is preserved (equation 2).

1.5 Standard Mode and Sequential Mode

As will be discussed more fully in Section 4, the user specifies a list of stimuli to be processed by the network.

In **Standard Mode**, the stimuli are processed independently: A stimulus is applied to the network as external input, and the network is run until a termination condition is met. Then the network is re-initialized and the second stimulus is presented, and so on.

In **Sequential Mode**, the stimuli are processed in sequence, and the network is **not** re-initialized between stimuli, allowing the final state reached in processing one stimulus to serve as the initial state in processing the next. In addition, it is also possible to specify the extent to which the **input** from the previous stimulus continues to be applied to the next stimulus. This is discussed more in Section 3.2.

2. Setting your MATLAB path

To set up Matlab or Octave to run LDnet, first you must add the folders **LDnet/code** and **LDnet/code/utils** to your path. You can do this with the **addpath** command. For example, if the LDnet directory is directly under your home directory, you would execute the command:

addpath '~/LDnet/code' '~/LDnet/code/Utils'

3. Creating a Domain file and a Settings file

To run an LDnet simulation, you must create two MATLAB m-files: a **Domain file**, which defines your Tensor-Product representations and your Harmony function, and a **Settings file** which defines runtime settings for the simulation such as the λ and T-reduction schedules. Each file must define a function returning a MATLAB structure that contains certain fields that LDnet expects.

3.1 The Domain file

A domain file is where your Tensor-Product representations and your Harmony function are defined. It is a Matlab .m file which must define a function that returns a MATLAB structure containing (at least) the fields in the table below.

To define your Tensor-Product representations, you create a matrix **R** whose columns are the TP representations of all the roles in your domain; and you create a matrix **F** whose columns are the TP representations of all the filler in your domain. You place these into the fields **R** and **F** of the domain structure you are creating.

The weights and biases in your domain are specified in the **Hc** and **Hcc** fields of your domain structure. These field names stand for "Harmony of constituents" and "Harmony of pairs of constituents" respectively. When you define the weights and biases, you specify them in **c-space** (Section 1.3). That is, you specify coefficients that determine the amount of Harmony that is contributed when a particular constituent (or pair of constituents) is present in the state. For example, a bias of 2 on some constituent c means that there will be a Harmony increment of 2 whenever constituent c is present in the state. And there will be a Harmony increment of $2a_c$ when constituent c is present in the state with implicit

activation equal to a_c. Similarly, a weight of 2 between constituents *c* and *c'* specifies that there will be a Harmony increment of $2a_c a_{c'}$ whenever constituents *c* and *c'* are simultaneously present in the state with implicit (c-space) activations a_c and $a_{c'}$.

TP representations can be used to encode similarity relations between fillers (or between roles), by specifying non-orthogonal vectors whose dot-product represents the degree of similarity between the two fillers (or roles). When this is the case, two constituents may have a non-zero dot product, in which case there will be what we might call **Harmonic crosstalk** between the constituents: Harmony values specified for one constituent will partially apply to all other constituents to which it is similar (non-orthogonal). For example, an input of 1.5 on a constituent *c* will not only increment the Harmony by $2a_c$ but will also increment it by $2a_{c'}(a_c \cdot a_{c'})$, as though constituent *c'* was also receiving an input. The rule is analogous for weights.

Required fields of the *domain* structure:

Field name	Data Type	Description
`nR`	integer	The number of distinct roles in the domain.
`nF`	integer	The number of distinct fillers in the domain.
`roleNames`	A cell array of strings of length nR	Names for your roles, used to make displays more readable.
`fillerNames`	A cell array of strings of length nF	Names for your fillers, used to make displays more readable.
`R`	A [nR x nR] matrix	The role vectors in your domain are defined to be the columns of this matrix.
`F`	A [nF x nF] matrix	The filler vectors in your domain are defined to be the columns of this matrix.
`Hc`	A [nF x nR] matrix	Single-constituent Harmony values. Hc stands for "Harmony of a constituent". Element Hc(f, r) specifies the Harmony increment that results whenever constituent (<r,f>) is present in the state. However, there will be crosstalk from other constituents: Constituent (<r,f>) will receive additional Harmony from all terms Hc(f',r') in which f is non-orthogonal to f' *and* r is non-orthogonal to r'.
`Hcc`	A 4-dimensional [nF x nR x nF x nR] matrix	Pairwise constituent Harmony values. Hcc stands for "Harmony of two constituents". Element Hcc(f1, r1, f2, r2) specifies the Harmony increment that results whenever the two constituents <r1,f1> and <r2,f2>) are simultaneously present in the state. However, there will be crosstalk from other constituent pairs, analogous to the situation for Hc (see above).
`z`	real number	The center of the "bowl" added to the Harmony function to make it have a single maximum. See equation 7.
`q`	real number > 0	The strength of the bowl. See equation 7.

maxAbsInput	real number > 0	An upper bound on the magnitude of any stimulus component. This is only used to compute a recommended value for q.
axis1Fillers	A [1 x nR] vector of integer [1,nF]	For use in animating a run of the simulation. This is a vector of filler numbers, specifying a state of the system This state vector is used as the "x axis" in the 3D animation plot.
axis2Fillers	A [1 x nR] vector of integer [1,nF]	The "y axis" of the animation plot. See 'axis1Fillers' above.
axis3Fillers	A [1 x nR] vector of integer [1,nF]	The "z axis" of the animation plot. See 'axis1Fillers' above.

Help in creating the Domain file:

You have the full Matlab/octave language to help you create your domain fields, and typically the construction of the **Hc** and **Hcc** fields requires some programming. Here are some tips and functions to help you create your domain file:

Define a function: Remember to define a function in the domain file that returns a structure.

nR and nF: The roles should contain all possible roles in the domain. Likewise for the fillers. LDnet considers all fillers to be candidates for all roles, so if one of your roles needs a restricted set of fillers, you must put weights into the Harmony function to penalize the illegal fillers for that role.

R and F: Often it is desirable to have similarity relationships within the sets of roles and/or fillers. There are a few helper functions for this task:

```
function M = dotProducts (N, dim, s)
```

Creates a set of vectors with a common dot-product: Given scalars N, dim, and s, finds N dim-dimensional unit vectors whose pairwise dot products are all s (and whose self-dot products are 1). Results are returned in the columns of M.

```
function [M itns] = dotProducts2 (N, dim, dpMatrix)
```

Creates a set of vectors with the specified set of pairwise dot-products. Given square matrix dpMatrix of dimension N-by-N, finds N dim-dimensional unit vectors whose pairwise dot products match dpMatrix. Results are returned in the columns of M.

```
function [R R1 R2] = compositionalRoles (nR1, nR2, dpMat1, dpMat2)
```

Creates a compositional set of role vectors: First, creates two sets of "basic" role vectors whose pairwise dot-products match the matrices dpMat1 and dpMat2. These are returned in the matrices R1 and R2. Then, creates a set of "composite" roles such that there is a new role for every pair of basic roles, obtained by taking the tensor product of the two basic role vectors. These role vectors are returned in the columns of R. The (vectorized) tensor product of R1(i) and R2(j) is returned in column ((i-1)*nR2+j) of R.

```
function R = syllablePositionRoles (nSyllables, nPositions, sdp, pdp)
```

Creates a compositional set of role vectors (a special case of `compositionalRoles()`). Designed for a domain with syllables and positions, where we want a role for each <syllable, position> pair, all syllables have a common similarity, and all positions within a syllable have a common similarity. Role vector (s,p) is the tensor product of the vector for s and the vector for p. sdp is the desired common dotproduct between all syllable vectors and sdp is the desired common dotproduct between all position vectors.

Hc: This is a straightforward [nF x nR] matrix of Harmonies for single constituents, where a constituent is a single < filler , role> pair.

Hcc: This is a 4-dimensional [nF x nR x nF x nR] matrix of Harmonies for pairs of constituents, where a constituent is a single [filler, role] pair. Some tips:

- It may help to think of a 4D index (i,j,k,l) as two *pairs* of indices (i,j) and (k,l), each pair indexing a single constituent. The Harmony value at index (i,j,k,l) is the Harmony associated with the conjunction of constituent (i,j) and constituent (k,l).
- Remember that the Hcc must be symmetric at the "top level", i.e., Hcc(i,j, k,l) must equal Hcc(k,l, i,j). (But Hcc(i,j, k,l) need not equal Hcc(j,i, k,l)).

z and **q:** The purpose of the bowl is to give the Harmony function a unique maximum. The **z** parameter specifies the center of the bowl and is typically either 0.5 or approximately *1/nF*. The **q** parameter is the strength of the bowl. It is analogous to the magnitude of an inhibitory self-connection on every unit in c-space. The program will suggest a minimum value of **q** when it is run.

axis1Fillers… axis3Fillers: Recall that these are the three axes that the animation uses to plot the state. Choosing states that are most likely to be visited during a simulationis helpful.

3.2 The Settings file

A settings file must define a function returning a MATLAB structure containing (at least) the fields in table 3.2.1 below. If you wish to run LDnet in **Sequential Mode** (see Section 1.5), there are additional fields that must be specified, listed in table 3.2.2.

3.2.1 Settings that apply to Standard and Sequential Modes:

Field name	Data Type	Description
`randSeed`	integer > 0	A seed to initialize the random number generator. If you use the same seed on two repetitions, you will get the exact same results.
`repetitions`	integer > 0	The number of times to run each stimulus.
`timeStep`	real number > 0	The size of the timestep used to discretize the dynamical system. **Example: 0.005**
`tgtStd`	real number >= 0	The target standard deviation of the stationary distribution over states at $\lambda = 1$. Used to compute recommended initial T. **Example: 0.05**
`TInit`	real number >= 0	The initial Temperature. **Example: 0.05**.
`Tmin`	real number >= 0	The minimum Temperature. **Example: 0**

`TdecayRate`	real number >= 0	The rate of exponential decay of the Temperature toward Tmin, *per unit time*, not per step. **Example: 0.1**
`lambdaInit`	real number strictly between 0 and 1	The initial Lambda. The program will suggest a value when it is run.
`lambdaMin`	real number >= 0	The minimum Lambda. **Example: 0.03**
`lambdaDecayRate`	real number >= 0	The rate of exponential decay of Lambda toward lambdaMin *per unit time*, not per step. **Example: 0.05**
`maxSteps`	integer > 0	The maximum number of steps to run the simulation on a single repetition (only used if the convergence condition below is not met earlier). **Example: 30000**
`emaSpeedTol`	real number >= 0	The convergence condition for each repetition: When the average speed of the state drops *below* this value, the repetition stops and Reaction Time is recorded. Speed is defined as the latest change in the fastest-changing single unit divided by timeStep. emaSpeed is an exponential moving average of Speed. **Example: 0.15**
`emaFactor`	Real number in [0,1]	The factor used to calculate emaSpeed. This is the "momentum" factor per unit time (*not* per timestep). The factor per timestep is $emaFactor^{timeStep}$, e.g., if emaFactor=.001 and timeStep=.01, then $emaFactor^{timeStep}$ = .9333. The ema update rule for one step is then: ema(t)=.9333ema(t-1)*.0667speed(t). **Example: 0.001**
`diary`	boolean	If true, MATLAB's **diary** function is used to capture the output of a simulation. The results are *appended* to the file diary.out in the current directory.
`printInterval`	integer > 0	The number of steps between printings of status information within a repetition. **Example: 1000**
`animate`	boolean	If true, a 3D animation of the current repetition is plotted (warning: very slow!)
`saveFile`	text string	After the entire simulation is completed, the **net**, **domain**, and **settings** structures are saved to the file named by this variable.
`initStateMean`	A [nF x nR] matrix	The mean of the Gaussian-distributed random initial state, in *c-space*. (A new initial state is chosen for each repetition). **Example: `ones(nF, nR)/nF`**
`initStateStdev`	real number >= 0	The standard deviation of the Gaussian-distributed random initial state for each repetition, in *c-space*. **Example: 0.01**
`sequentialMode`	boolean	If true, LDnet will run in sequential mode (Section 1.5).

3.2.2 Additional Settings for Sequential Mode (optional):

In Sequential Mode, it is sometimes desirable to use different settings for the **first** stimulus in the sequence, and the **last** stimulus in the sequence. The settings below allow you to specify settings that apply to the first, or last, stimuli in the sequence, while the settings above apply to the remaining (middle) stimuli in the sequence.

With the exception of the new "`carryOver`" field (the first one in the table below), the fields below have the same name and meaning as those above, except the suffix _FIRST or _LAST is attached to the name in order to indicate the stimulus to which the setting applies.

These variables are only needed if `settings.sequentialMode` is true, and they are all optional, defaulting to the corresponding settings above.

Field name	**Data Type**	**Corresponds to setting**
`carryOver`	real number >= 0	This is a new settings specifying how much of the **input** from the previous stimulus should be added to the current input. Default is 0.
`tgtStd_FIRST`	real number >= 0	`tgtStd`
`tgtStd_LAST`	real number >= 0	`tgtStd`
`TInit_FIRST`	real number >= 0	`TInit`
`TInit_LAST`	real number >= 0	`TInit`
`Tmin_FIRST`	real number >= 0	`Tmin`
`Tmin_LAST`	real number >= 0	`Tmin`
`TdecayRate_FIRST`	real number >= 0	`TdecayRate`
`TdecayRate_LAST`	real number >= 0	`TdecayRate`
`lambdaInit_FIRST`	real number between 0 and 1	`lambdaInit`
`lambdaInit_LAST`	real number between 0 and 1	`lambdaInit`
`lambdaMin_FIRST`	real number >= 0	`lambdaMin`
`lambdaMin_LAST`	real number >= 0	`lambdaMin`
`lambdaDecayRate_FIRST`	real number >= 0	`lambdaDecayRate`
`lambdaDecayRate_LAST`	real number >= 0	`lambdaDecayRate`
`maxSteps_FIRST`	integer > 0	`maxSteps`
`maxSteps_LAST`	integer > 0	`maxSteps`
`emaSpeedTol_FIRST`	real number >= 0	`emaSpeedTol`
`emaSpeedTol_LAST`	real number >= 0	`emaSpeedTol`
`emaFactor_FIRST`	Real number in [0,1]	`emaFactor`
`emaFactor_LAST`	Real number in [0,1]	`emaFactor`
`initStateStdev_FIRST`	real number >= 0	`initStateStdev`
`initStateStdev_LAST`	real number >= 0	`initStateStdev`

4. Creating an array of stimuli

A single **stimulus** is a [nF x nR] matrix that specifies the amount of input that should be given to each constituent, specified in c-space. It is converted to s-space automatically. The actual input pattern in s-

space will be the sum of all of the constituents weighted by their value in the stimulus matrix. Note that there will be crosstalk between constituents: Input specified for constituent <*f,r*> will also apply partially to any other constituent <*f',r'*> if *f* is non-orthogonal to *f'* and *r* is non-orthogonal to *r'*.

To specify a set of stimuli, you need to create a 3D MATLAB array of dimension [nF x nR x nStimuli]. An easy way to do this is to put the stimuli in a text file and read them into MATLAB using the following function:

```
function A = readStimuli (filename)
```

This function returns a 3D array. The format of the file is as follows: Line 1 contains the three dimensions of the array in order (nF, nR, nStimuli) separated by whitespace. The rest of the file contains a sequence of [nF x nR] matrices separated by whitespace. There should be nothing but numbers in the file. For example, here is a file containing a [3 x 2 x 4] matrix (4 stimuli, each of dimension [3 x 2]):

```
3 2 4

0.0568    0.7992
0.1453    0.4464
0.3419    0.7278

0.5280    0.9451
0.3537    0.8386
0.3662    0.8870

0.7772    0.3657
0.0699    0.3511
0.4011    0.6932

0.9057    0.1634
0.9895    0.7826
0.1985    0.3563
```

Here is how this data would be read from a file called 'data.txt':

```
>> A = readStimuli('data.txt')
Opened file data.txt for reading.
Reading 4 matrices of dimension [3 x 2] from data.txt...
A(:,:,1) =
    0.0568    0.7992
    0.1453    0.4464
    0.3419    0.7278
A(:,:,2) =
    0.5280    0.9451
    0.3537    0.8386
    0.3662    0.8870
A(:,:,3) =
```

```
    0.7772    0.3657
    0.0699    0.3511
    0.4011    0.6932
A(:,:,4) =
    0.9057    0.1634
    0.9895    0.7826
    0.1985    0.3563
```

5. Running a simulation with `runLDnet()`

Once you have defined a **domain** file, a **settings** file, and have created a 3D array of **stimuli**, you can run a simulation with the function:

```
function runLDnet (stimuli, domainFile, settingsFile)
```

This function executes the `domainFile` and `settingsFile` to obtain domain and settings objects, then runs the network on all the `stimuli` a number of times specified by `settings.repetitions`.

5.1 Suggested values for q, TInit, and Lambda at runtime

q:

When the program starts, it calculates a lower bound on **q** in order for the Harmony function to have a unique maximum, and therefore for the dynamical system to have a stationary distribution. It also calculates a lower bound on **q** to ensure that the maximum of the Harmony function lies inside the $[0,1]^N$ activation box in c-space. These values are reported at the start of the simulation:

```
Recommended Q > 3.897349 (max in box) > 1.732051
(stationary distribution). dom.q = 4.000000
```

This says that the recommended value of **q** is (at least) `3.897349,` which would place the Harmony maximum in the box, and **q** must be at least `1.732051` to ensure that the system has a stationary distribution at all. It also reports that the value of **q** in the domain file is 4.0.

TInit:

The program calculates the value of **TInit** that is required to make the initial standard deviation of the stationary distribution equal to the value specified in the settings file, **tgtStd,** at $\lambda = 1$. If your initial value of λ is much smaller than 1, then you should set TInit to a larger value. The recommended and current values are reported as follows:

```
Recommended T < 0.005670. settings.TInit = 0.500000
```

If you would like the program to use the recommended **T** value as your **TInit** value, then set **TInit** to a negative number in the settings file. In that case the program reports this

fact and sets **TInit** to the recommended value:

```
Recommended T < 0.005670. settings.TInit = -1.000000
settings.TInit<0 (-1.000000). Setting TInit to produce
tgtStd... TInit = 0.005670.
```

Lambda:

The program provides a rough estimate of the value of λ at which the forces of Harmony maximization and Discretization dynamics will be approximately equal. This can serve as a guide to setting **lambdaInit** and **lambdaDecayRate** in your settings file.

5.3 Monitoring a simulation

While processing each stimulus, LDnet displays a progress report every **settings.printInterval** steps, that looks something like this:

```
[step, time, H, lambda, T, emaSpeed] = [200, 2, 3.9, 0.549, 0.0017, 0.391]
Effective activation of all constituents:
        ''    'left'      'right'     'root'
    'Al '     [0.3309]    [0.3112]    [0.0983]
    'Is '     [0.3038]    [0.2873]    [0.1405]
    ']S '     [0.2793]    [0.2569]    [0.3127]
    ']S2 '    [0.2474]    [0.2341]    [0.2610]
Nearest TP state: Al Al ]S
```

Reported in the first line are the current step, time (which is step/stepSize), Harmony of the current state, T, and emaSpeed (the exponential moving average of the state's speed). After that are the effective activations of all possible constituents. These are the c-space activations, displayed in a (nF x nR) matrix, labeled by the filler and role names.

After a simulation, or if you interrupt a simulation (by using Command-. at the MATLAB command line), you can access the network object via the variable **`net`**. The variables **`domain`** and **`settings`** are also defined and hold the results of executing your domain and settings files.

6. Plotting and analyzing your simulation results

After a simulation is run, LDnet will have stored full activation traces of every repetition of every stimulus inside the '**net**' structure, along with other fields:

Field name	Data Type	Description
finalTPstate	[nStimuli x nRepetitions] cell array of strings	A matrix containing the *name* of the final TP state of each repetition.
finalTPstateNum	[nStimuli x nRepetitions] array of reals	A matrix containing the *number* of the final TP state of each repetition.

finalRT	[nStimuli x nRepetitions] array of reals	A matrix containing the *Reaction Time* (the time to convergence) of each repetition.
divergentP	[nStimuli x nRepetitions] array of booleans	A matrix containing 1's (true) and 0's (false) indicating whether each repetition was divergent.

```
net.fullHTrace:       [nStimuli x nRepetitions x nSteps] matrix
net.fullSpeedTrace:   [nStimuli x nRepetitions x nSteps] matrix
net.fullEmaSpeedTrace: [nStimuli x nRepetitions x nSteps] matrix
net.fullLTrace:       [nStimuli x nRepetitions x nSteps] matrix
net.fullTTrace:       [nStimuli x nRepetitions x nSteps] matrix
net.fullTPTrace:      {nStimuli x nRepetitions x nSteps} cell array
net.fullTPNumTrace:   [nStimuli x nRepetitions x nSteps] matrix
net.fullTPhTrace:     [nStimuli x nRepetitions x nSteps] matrix
net.fullTPdistTrace:  [nStimuli x nRepetitions x nSteps] matrix
net.fullSTrace:       [nStimuli x nRepetitions x nSteps x nUnits] matrix
```

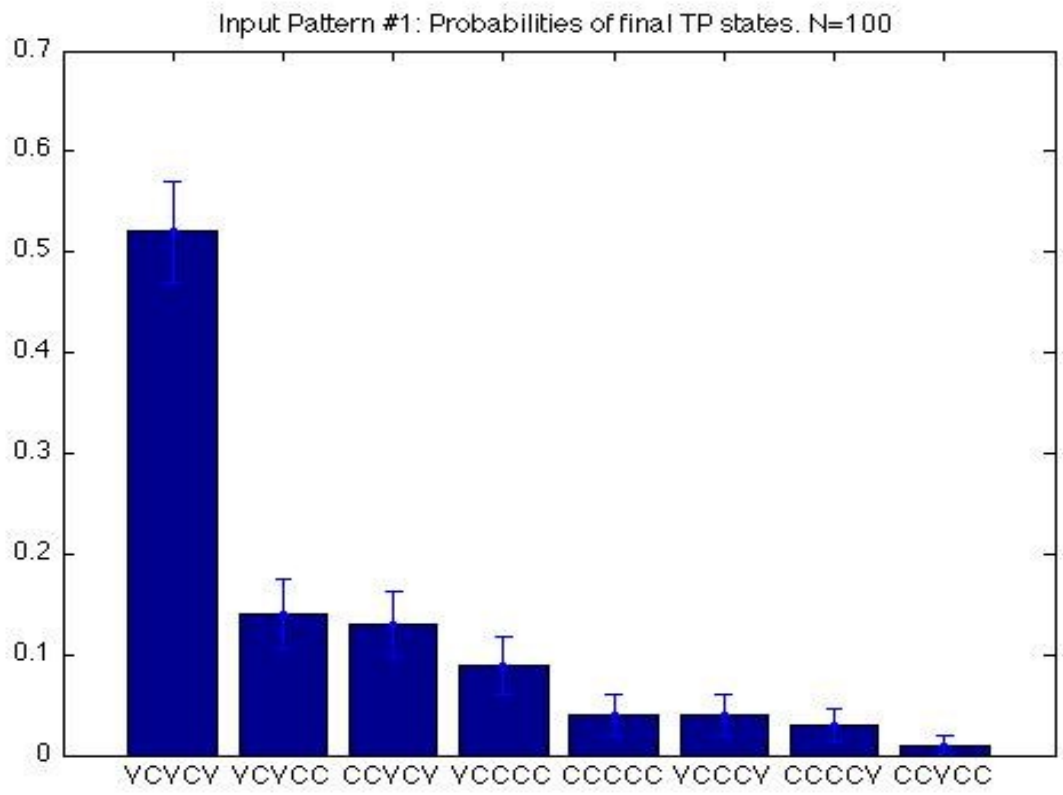
The global variables **net**, **domain**, and **settings** are saved in a MATLAB-loadable .mat file in a filename specified in **settings.saveFile**. You can load these variables back into MATLAB with the **load** command:

```
>> x = load('mySavedFilename')
x =
          net: [1x1 struct]
       domain: [1x1 struct]
     settings: [1x1 struct]
```

To help you calibrate and analyze your simulations, there are a few plotting functions that may be helpful:

```
allSortedCounts = plotFinalTPStates (net, domain, topK)
```

This command creates two charts for every stimulus. One is a bar chart showing the frequency with which the network settled on various TP states, and the other is a scatter plot of the log probability of the final TP state versus their Harmony. For example, if you ran 3 stimuli for 100 repetitions, you would see 3 bar charts, each a histogram of 100 repetitions, and 3 scatter plots, each with 100 points. The return variable, `allSortedCounts`, is a matrix of size [nStimuli x nRepetitions], each row of which shows the counts used to compute the bars in the bar charts.



```
function plotStimulusRepetition (net, repNum, stimNum,
                                 fromStep, toStep)
```

Creates a "ribbon plot" of repetition `repNum` of stimulus `stimNum`, showing the between timeSteps `fromStep` and `toStep`. The horizontal axis is timeStep. The vertical axis is the Harmony of each TP state whose basin was visited during the repetition. The Harmony value and the name of the TP state are labeled. Color indicates the log distance of the network state to the TP state (red is far, blue is close). The exact positions on the vertical axis are sometimes adjusted slightly so the labels don't overlap.

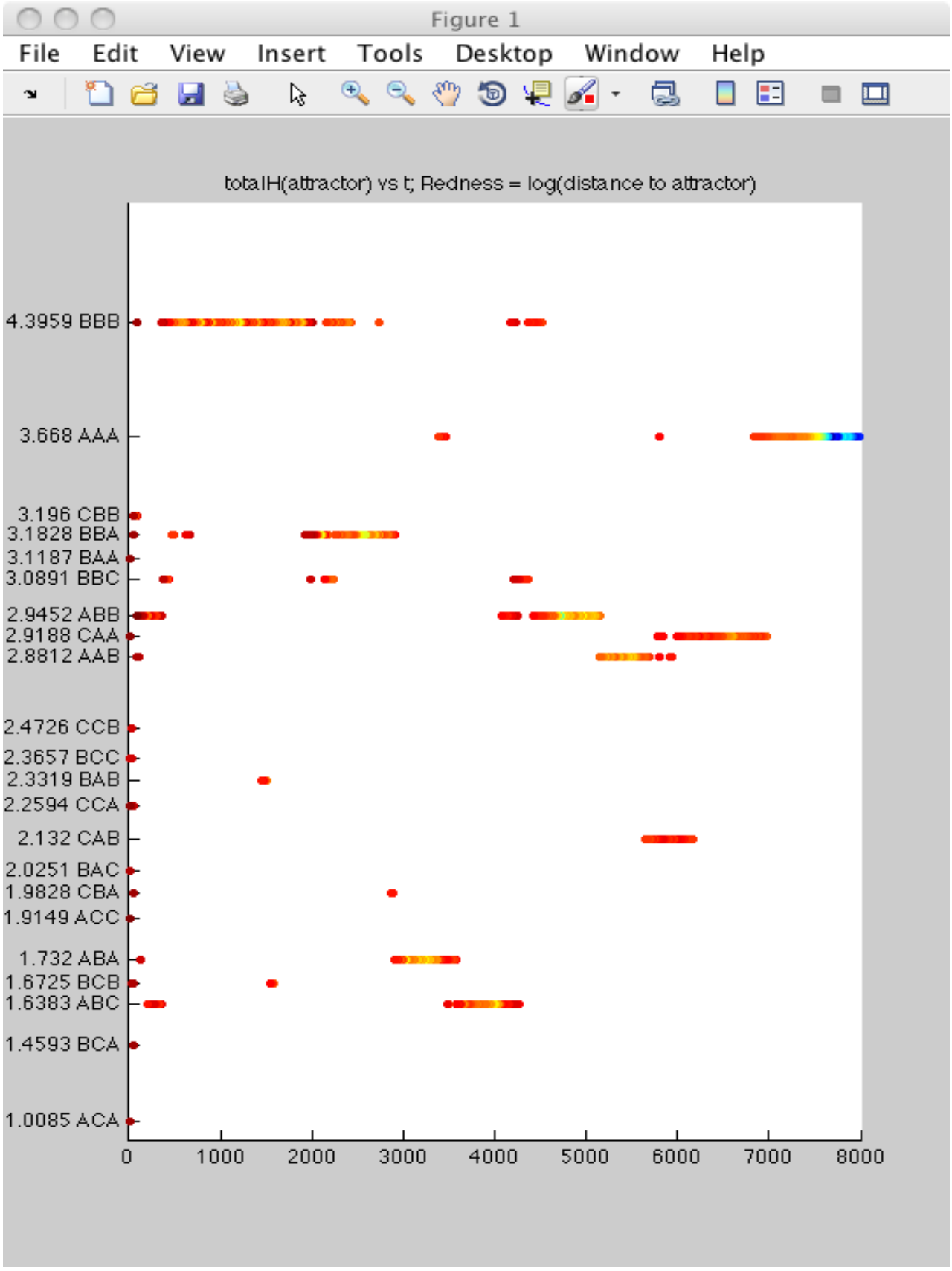
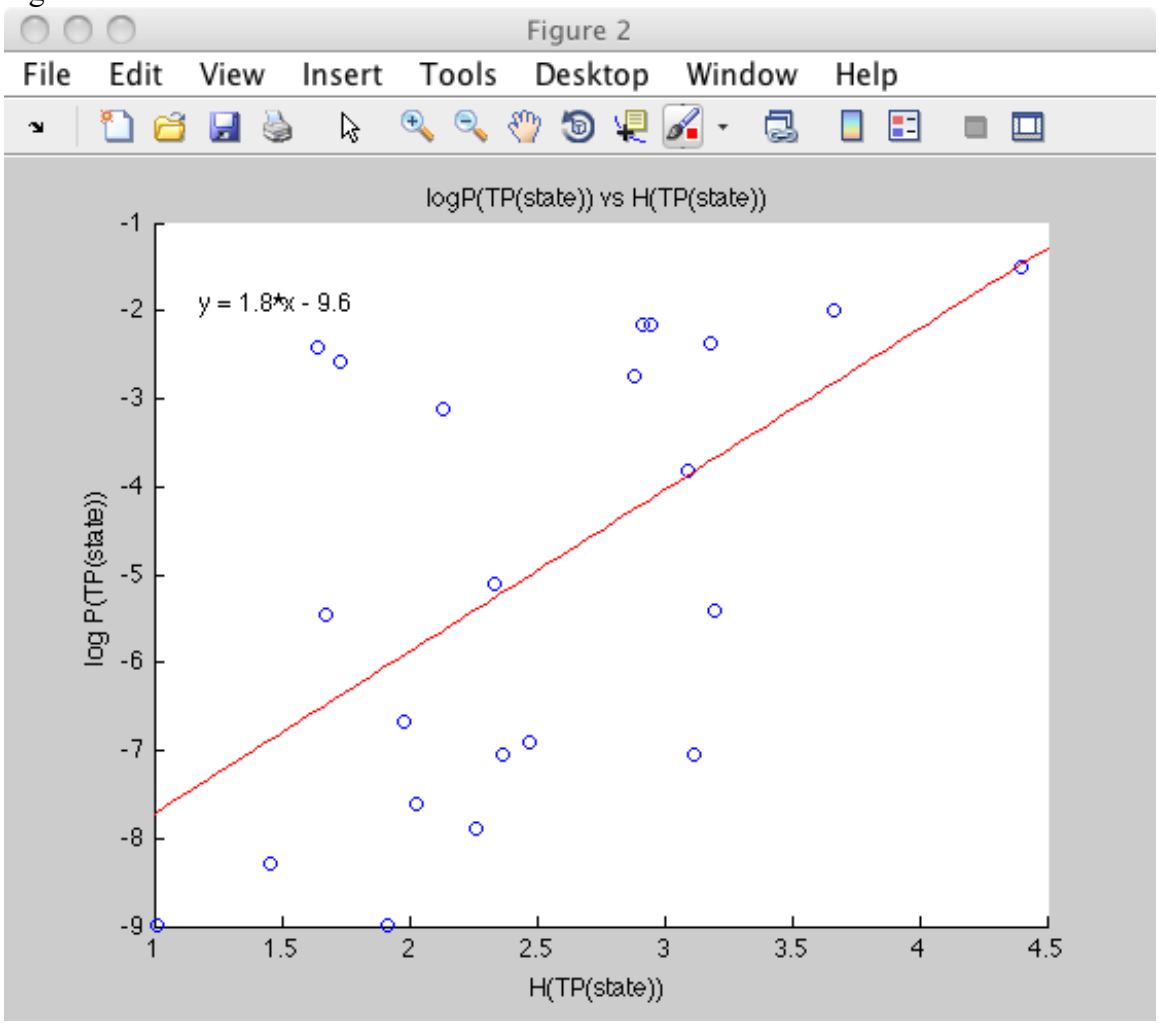


Figure 1
File Edit View Insert Tools Desktop Window Help
totalH(attractor) vs t; Redness = log(distance to attractor)
4.3959 BBB
3.668 AAA
3.196 CBB
3.1828 BBA
3.1187 BAA
3.0891 BBC
2.9452 ABB
2.9188 CAA
2.8812 AAB
2.4726 CCB
2.3657 BCC
2.3319 BAB
2.2594 CCA
2.132 CAB
2.0251 BAC
1.9828 CBA
1.9149 ACC
1.732 ABA
1.6725 BCB
1.6383 ABC
1.4593 BCA
1.0085 ACA
0
1000
2000
3000
4000
5000
6000
7000
8000

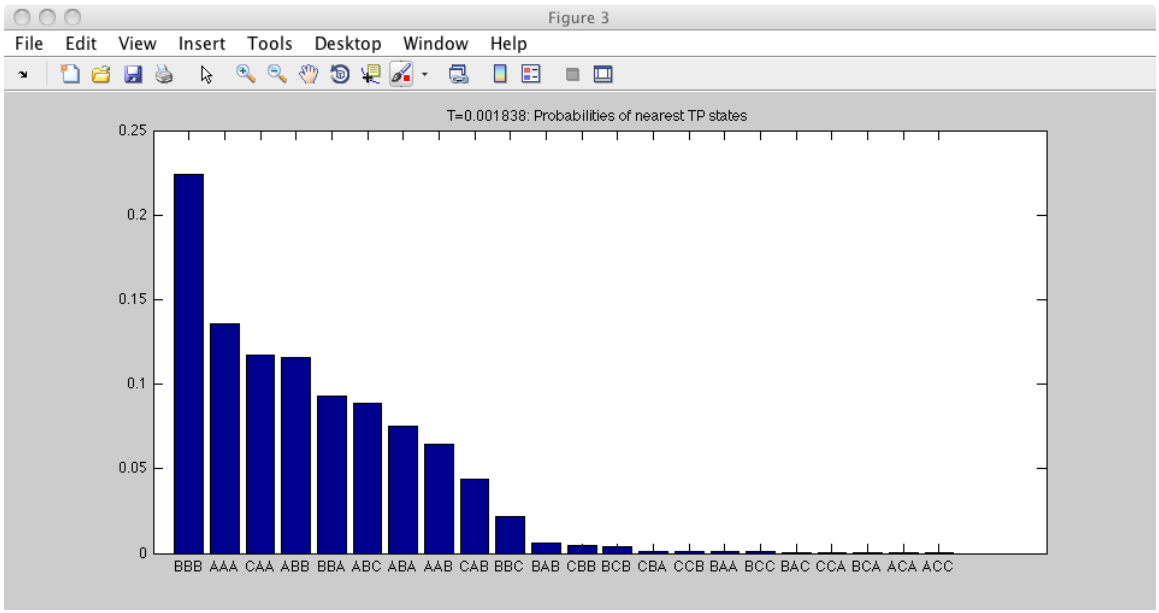
`function plotTPStateScatter (net, domain, repNum, stimNum)`

Creates a scatter plot of the log probability of the network being in each TP state basin during repetition `repNum` of stimulus `stimNum`, vs. the Harmony of the TP state. Also plots a linear regression line:



`function plotTPStateProbs (net, domain, repNum, stimNum)`

Creates two bar charts: One is the probability that each TP state basin was visited, in order of probability. The second is the Harmony of those TP states, in the same order:



In addition to these plotting functions, you can also plot other information about each repetition that LDnet automatically stores. In the net structure, the following fields may be useful to plot: (Note the use of the `squeeze` command to eliminate singleton dimensions from multidimensional array-access results).

Field name	Plotting command	Description
fullHTrace	`figure; plot(squeeze(net.fullHTrace(stimNum, repNum, :)))`	Harmony of the state vs. step.
speedTrace	`figure; plot(squeeze(net.fullSpeedTrace(stimN repNum, :)))`	Speed of the state vs. step.
emaSpeedTrace	`figure; plot(squeeze(net.fullEmaSpeedTrace( . stimNum, repNum, :)))`	Exponential moving average of the speed vs. step.
lTrace	`figure; plot(squeeze(net.fullLTrace(stimNum, repNum, :)))`	Lambda vs. step.
tTrace	`figure; plot(squeeze(net.fullTTrace(stimNum, repNum, :)))`	Temperature vs. step.
TPhTrace	`figure; plot(squeeze(net.fullTPhTrace(stimNum repNum, :)))`	Harmony of the nearest TP state vs. step.
sTrace	`figure; plot(squeeze(net.fullSTrace(stimNum, repNum,:,:)))`	Activation of all the state units vs. step. Most useful when F=R=identity matrix.
TPdistTrace	`figure; plot(squeeze(net.fullTPdistTrace(stim repNum, :)))`	Distance to the nearest TP state vs. step.

7. Example domains

There are two domains defined that you can refer to in defining your domain. Each has a domain file and settings file. It is recommended that you copy these files and modify them to create your domain.

7.1 The trees domain

The trees domain is defined in the files **treesDomain.m** and **treesSettings.m** and **treesInputs.txt**. There are 3 roles, defining the 3 possible positions in a tiny 3-node tree with one root and two leaves. The 4 possible fillers include two suitable for root position (S and S2) and two suitable for leaves (Al and Is). The Harmony function is designed to assign H=0 to the two possible valid trees, and negative H to all other TP states. The valid trees are "Al Is S" and "Is Al S2".

The Harmony function assigns a bias of -1 to each possible leaf constituent and -2 to each possible root constituent, thereby incurring a "debt" if any unit is turned on. The debt can be repaid by turning on the appropriate unit in a connected tree position, because the connections between valid neighbors are all 2.

In this example, 10 repetitions of two stimulus patterns are presented. The first input is all zeros, giving an unbiased input. The result is approximately equal numbers of each of the two possible output trees. The second stimulus gives an input of 1.0 to the component representing "Al" in the Left position. The result is 10/10 trials ending in the correct completion of that input "Al is S".

To run this domain: **Start in the LDnet directory and paste these directly into the MATLAB command window**:

```
cd Trees
I = readStimuli('treesInputs.txt')
runLDnet(I, 'treesDomain', 'treesSettings')
```

To see some results:

(1) Final states and reaction times: Paste the following 2 lines into MATLAB:

```
net.finalTPstate
net.finalRT
```

The first command returned a 2x10 matrix of output sentences. The first row corresponds to the zero stimulkus. You should see roughly equal numbers of the two legal trees. The second row corresponds to the "Al"/Left stimulus. You should see all responses equal to "Al is S".

The second command returned a corresponding matrix of reaction time. Note that the responses for stimulus #2 are faster than for the zero, ambiguous, stimulus.

Now paste this for some plots:

```
plotFinalTPStates(net, domain, 12);
plotStimulusRepetition (net, 1, 1);
```

```
plotTPStateScatter (net, domain, 1, 1);
plotTPStateProbs (net, domain, 1, 1);
```

7.2 The twister domain

The twister domain is defined in the files **twisterDomain.m** and **twisterSettings.m** and **twisterInputs.txt**. This models a task in which subjects must recite two-syllable tongue twisters such as "sag can" or "cag san". Each syllable is of the form CVC (consonant-vowel-consonant). Vowel identity is not modeled. There are 4 roles, corresponding to the 4 positions consonants can occur, in the onset and coda of each syllable. There are 4 fillers representing the consonants k, g, n and s.

To express the idea that some fillers are more similar than others, the vectors representing k and g have a dot-product of 0.5, and similarly for the pair (n, s). Other pairs (e.g., (k, n)) have dot products equal to 0.1. Similarly, the two roles representing the onsets of the two syllables are represented by vectors with a higher dot product than those representing different syllable positions. Finally, roles representing positions within the same syllable also have a nonzero dot-product, but a smaller one than that between similar positions in different syllables. (See the file twisterDomain.m for full details).

The weights and biases in this domain are all zero. The input specifies a particular sequence of syllables. The interesting thing about this model is that it makes errors in a pattern that reflects the detailed similarity structure between the different roles and fillers defined in the domain.

To run this domain: (**Start in the LDnet directory and paste these into the MATLAB command window**)

```
cd Twister;
I = readStimuli('twisterInputs.txt')
runLDnet(I, 'twisterDomain', 'twisterSettings')
```

To see some results:

```
plotFinalTPStates(net, domain, 12);
net.finalTPstate
net.finalRT
plotStimulusRepetition (net, 1, 1);
plotTPStateScatter (net, domain, 1, 1);
plotTPStateProbs (net, domain, 1, 1);
figure;plot([squeeze(net.fullTPhTrace(1,1,:)),squeeze(net.fullHTrace(1,1
,:))]);
figure;plot(squeeze(net.fullSTrace(1, 1,:,:)))
```

7.3 The treesSequential domain

The treesSequential domain is defined in the treesSequential folder. It is defined exactly the same as the trees domain, except that it is run in sequential mode. (Read about the trees domain, above, if you have not already).

In this example, there are two input files. Each contains two stimuli, the first of which is briefly presented, followed by the second. The goal is to examine how the brief stimulus affects the processing of the second stimulus. In the first input file, the first stimulus activates "Is" in the first position, and the second stimulus activates "Al" in the first position. In the second input file, the first stimulus is zero, and the second stimulus activates "Al" in the first position. The settings file runs the pair of stimuli for 10 repetitions.

The result is that, while all repetitions reach the same final state, "Al is S", the runs with the "interfering" stimulus "Is", are significantly slower to converge than those with the neutral, zero stimulus.

To run this domain: **Start in the LDnet directory and paste these directly into the MATLAB command window**:

```
cd TreesSequential

% First run stimulus 1 ("Is"):
I1 = readStimuli('treesSequentialInputs1.txt')
runLDnet(I1, 'treesSequentialDomain', 'treesSequentialSettings')

% Take note of the outputs and RT's at this point:
net.finalTPstate
net.finalRT

% Now run stimulus 2 (zeros):
I2 = readStimuli('treesSequentialInputs2.txt')
runLDnet(I2, 'treesSequentialDomain', 'treesSequentialSettings')

% Take note of the outputs and RT's at this point, and note that they
are faster in this condition:
net.finalTPstate
net.finalRT
```

Written by Don Mathis
mathis@jhu.edu

Version 1.5
September 2011