



Nengo Documentation

Release 1.3

Centre for Theoretical Neuroscience, University of Waterloo

July 04, 2012

CONTENTS

1	Nengo Tutorial	3
1.1	One-Dimensional Representation	3
1.2	Linear Transformations	8
1.3	Non-Linear Transformations	19
1.4	Feedback and Dynamics	24
1.5	Cognitive Models	30
2	Nengo Demos	39
2.1	Introductory Demos	39
2.2	Simple Transformations	43
2.3	Dynamics	50
2.4	Basal Ganglia Based Simulations	55
2.5	Learning	66
2.6	Miscellaneous	72
3	Nengo Scripting	83
3.1	Scripting Interface	83
3.2	Scripting Basics	84
3.3	Configuring Neural Ensembles	86
3.4	Speeding up Network Creation (and Making Consistent Models)	88
3.5	Adding Arbitrary Code to a Model	89
3.6	Connection Weights	91
3.7	Scripting Tips	93
3.8	List of Classes	97
4	Advanced Nengo Usage	111
4.1	Interactive Plots Layout Files	111
4.2	Creating a Drag And Drop Template	112
4.3	Running Experiments in Nengo	113
4.4	Running Nengo in Matlab	115
4.5	Generating Large Ensembles (500 to 5000 neurons)	116
4.6	GPU Computing in Nengo	118
4.7	Integrating with IPython Notebook	122
5	Java API Documentation	127

Nengo (<http://nengo.ca>) is a software package for simulating large-scale neural systems.

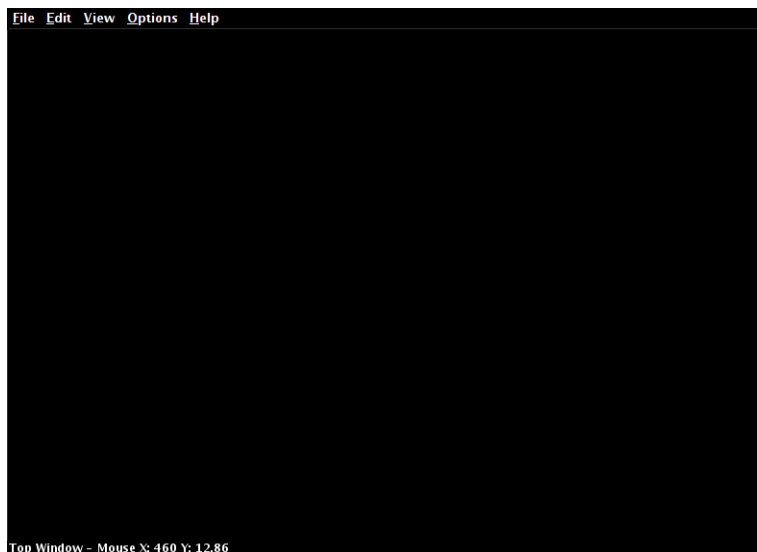
To use it, you define groups of neurons in terms of what they represent, and then form connections between neural groups in terms of what computation should be performed on those representations. Nengo then uses the Neural Engineering Framework (NEF) to solve for the appropriate synaptic connection weights to achieve this desired computation. This makes it possible to produce detailed spiking neuron models that implement complex high-level cognitive algorithms.

NENGO TUTORIAL

1.1 One-Dimensional Representation

1.1.1 Installing and Running Nengo

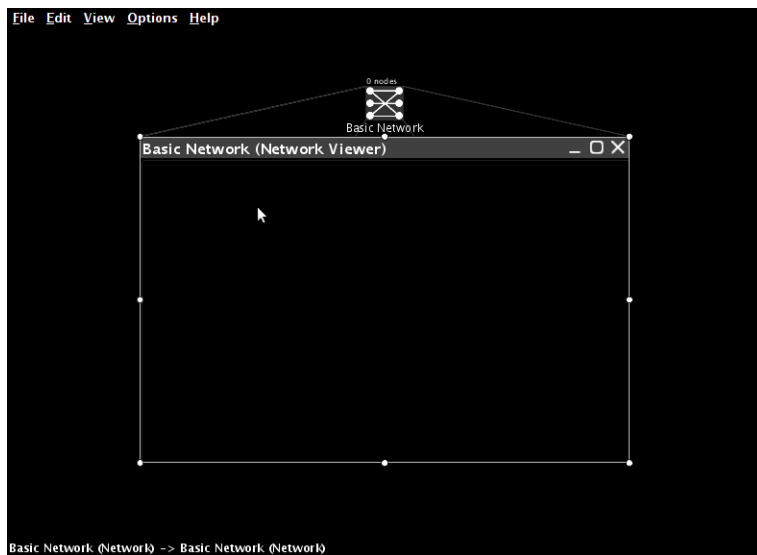
- Nengo runs on Linux, OS X, and Windows. The only requirement is that you have [Java](http://java.com) (<http://java.com>) already installed on your computer. Most computers already do have this installed.
- To download Nengo, download the file from <http://nengo.ca>
- To install Nengo, just unzip the file.
- To run Nengo, either:
 - Double-click on `nengo.bat` (in Windows)
 - run `./nengo` (in OS X and Linux)



1.1.2 Creating Networks

- When creating an NEF model, the first step is to create a Network. This will contain all of the neural ensembles and any needed inputs to the system.
 - File->New Network

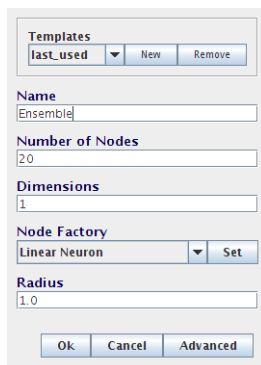
- Give the network a name



- You can create networks inside of other networks. This can be useful for hierarchical organization of models.

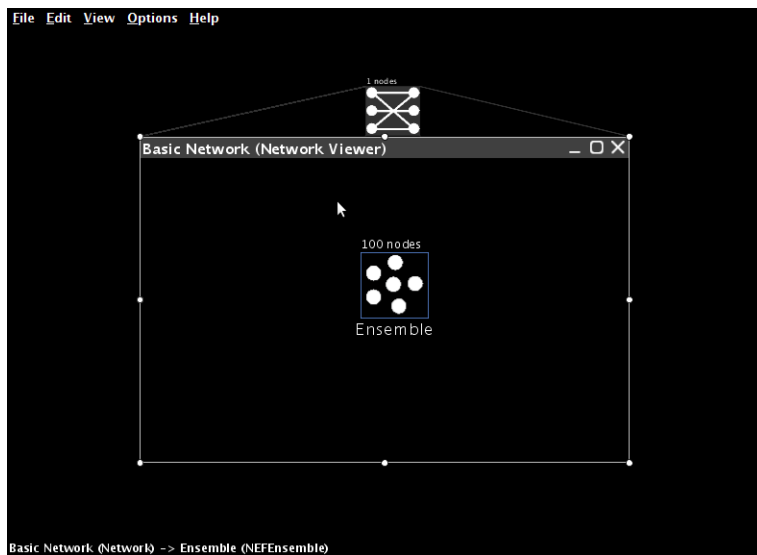
1.1.3 Creating an Ensemble

- Ensembles must be placed inside networks in order to be used
- Right-click inside a network
 - Create New->NEF Ensemble

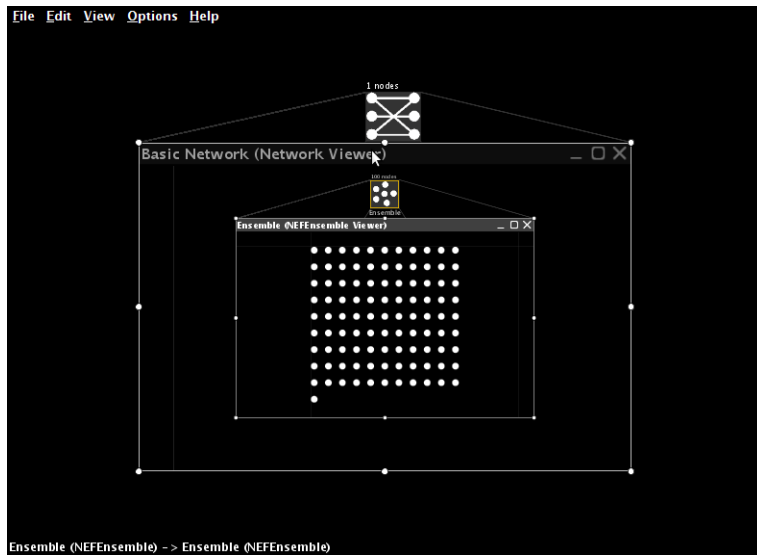


- Here the basic features of the ensemble can be configured
 - Name
 - Number of nodes (i.e. neurons)
 - Dimensions (the number of values in the vector encoded by these neurons; leave at 1 for now)
 - Radius (the range of values that can be encoded; for example, a value of 100 means the ensemble can encode numbers between -100 and 100)
- Node Factory (the type of neuron to use)

- For this tutorial (and for the majority of our research), we use LIF Neuron, the standard Leaky-Integrate-and-Fire neuron. Clicking on Set allows for the neuron parameters to be configured * tauRC (RC time constant for the neuron membrane; usually 0.02) * tauRef (absolute refractory period for the neuron; usually 0.002) * Max rate (the maximum firing rate for the neurons; each neuron will have a maximum firing rate chosen from a uniform distribution between low and high) * Intercept (the range of possible x-intercepts on the tuning curve graph; normally set to -1 and 1)
- Because there are many parameters to set and we often choose similar values, Nengo will remember your previous settings. Also, you can save templates by setting up the parameters as you like them and clicking on New in the Templates box. You will then be able to go back to these settings by choosing the template from the drop-down box.

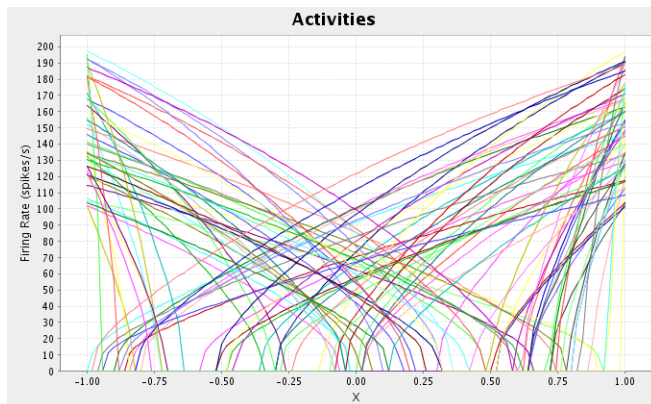


- You can double-click on an ensemble to view the individual neurons within it



1.1.4 Plotting Tuning Curves

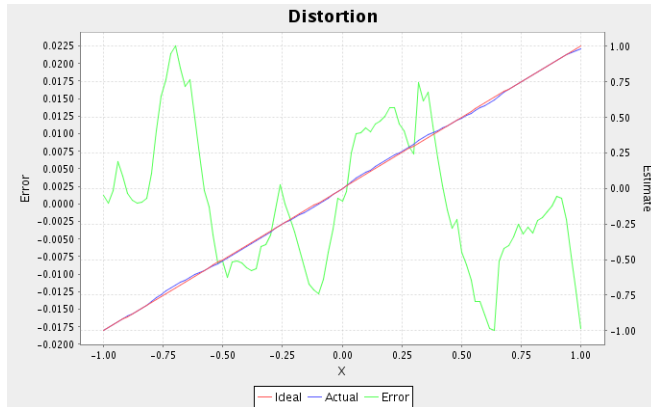
- This shows the behaviour of each neuron when it is representing different values (i.e. the tuning curves for the neurons)
- Right-click on the ensemble, select Plot->Constant Rate Responses



- tauRC affects the linearity of the neurons (smaller values are more linear)
- Max rate affects the height of the curves at the left and right sides
- Intercept affects where the curves hit the x-axis (i.e. the value where the neuron starts firing)

1.1.5 Plotting Representation Error

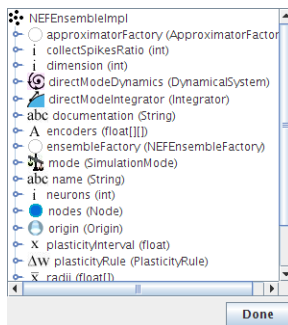
- We often want to determine the accuracy of a neural ensemble.
- Right-click on the ensemble, select Plot->Plot Distortion:X



- Mean Squared Error (MSE) is also shown (at the top)
- MSE decreases as the square of the number of neurons (so RMSE is proportional to $1/N$)
- Can also affect representation accuracy by adjusting the range of intercepts. This will cause the system to be more accurate in the middle of the range and less accurate at the edges.

1.1.6 Adjusting an Ensemble

- After an ensemble is created, we can inspect and modify many of its parameters
- Right-click on an ensemble and select Configure



- neurons (number of neurons; this will rebuild the whole ensemble)
- radii (the range of values that can be encoded; can be different for different dimensions)
- encoders (preferred direction vectors for each neuron)

1.1.7 The Script Console

- Nengo also allows users to interact with the model via a scripting interface using the Python language. This can be useful for writing scripts to create components of models that you use often.
- You can also use it to inspect and modify various aspects of the model.
- Press Ctrl-P or choose View->Toggle Script Console to show the script interface
 - The full flexibility of the Python programming language is available in this console. It interfaces to the underlying Java code of the simulation using Jython, making all Java methods available.
- If you click on an object in the GUI (so that it is highlighted in yellow), this same object is available by the name `that` in the script console.

- Click on an ensemble
- Open the script console
- type `print that.neurons`
- type `that.neurons=50`
- You can also run scripts by typing `run [scriptname.py]`

1.2 Linear Transformations

1.2.1 Creating Terminations

- Connections between ensembles are built using Origins and Terminations. The Origin from one ensemble can be connected to the Termination on the next ensemble
- Create two ensembles. They can have different neural properties and different numbers of neurons, but for now make sure they are both one-dimensional.
- Right-click on the second ensemble and select Add Decoded Termination
 - Provide a name (for example, `input`)
 - Set the input dimension to 1 and use Set Weights to set the connection weight to 1
 - Set `tauPSC` to 0.01 (this synaptic time constant differs according to which neurotransmitter is involved. 10ms is the time constant for AMPA (5-10ms).

Templates

Name
input

Weights
Input Dim: 1 Set Weights

tauPSC
0.01

Is Modulatory
☐ Enable

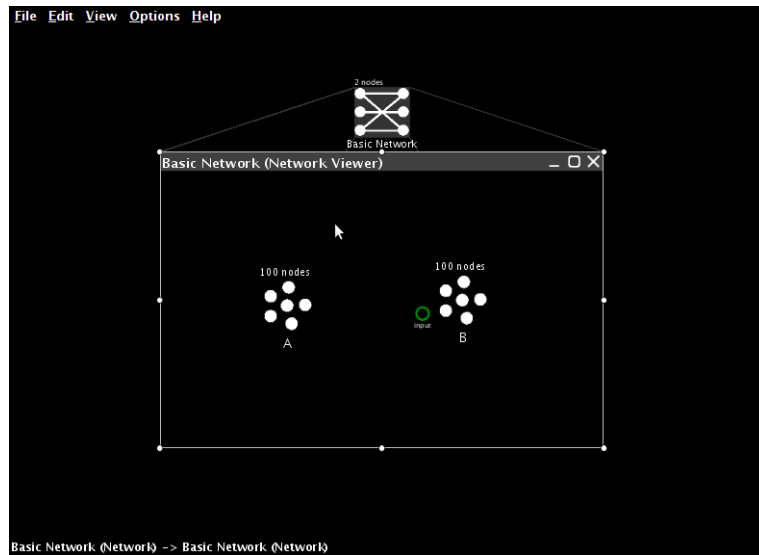
OK Cancel

Editor

1

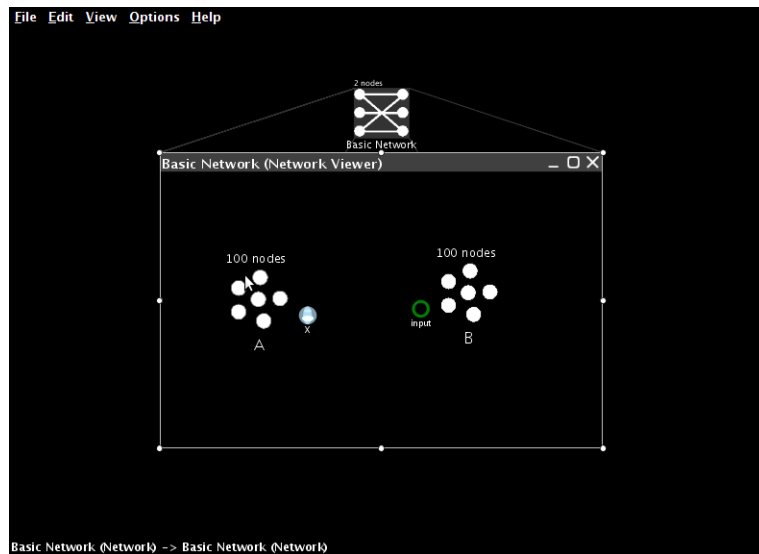
1.0

OK Cancel

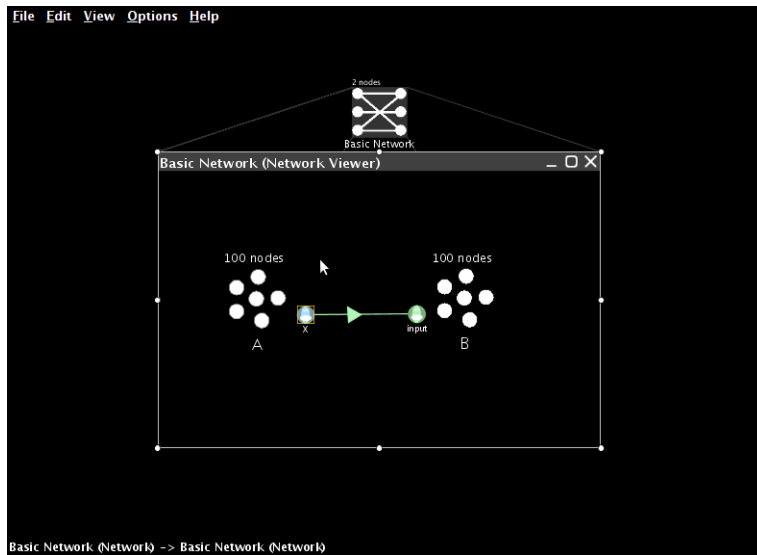


1.2.2 Creating Projections

- We can now connect the two neural ensembles.
- Every ensemble automatically has an origin called X. This is an origin suitable for building any linear transformation. In Part Three we will show how to create origins for non-linear transformations.



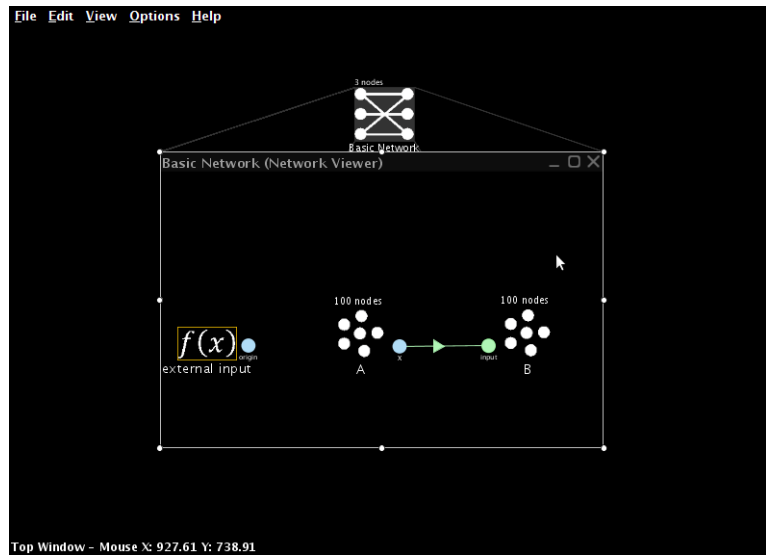
- Click and drag from the origin to the termination. This will create the desired projection.



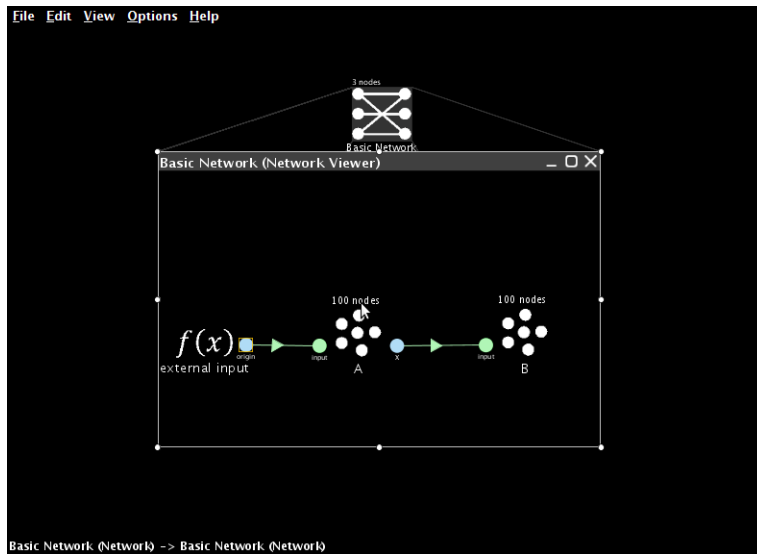
1.2.3 Adding Inputs

- In order to test that this projection works, we need to set the value encoded by the first neural ensemble. We do this by creating an input to the system. This is how all external inputs to Nengo models are specified.
- Right-click inside the Network and choose Create New->Function Input.
- Give it a name (for example, `external input`)
- Set its output dimensions to 1

- Press Set Function to define the behaviour of this input
- Select Constant Function from the drop-down list and then press Set to define the value itself. For this model, set it to 0.5.

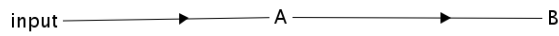


- Add a termination on the first neural ensemble and create a projection from the new input to that ensemble.

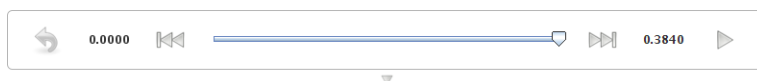
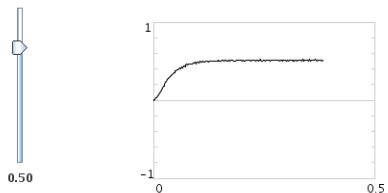
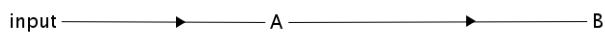


1.2.4 Interactive Plots

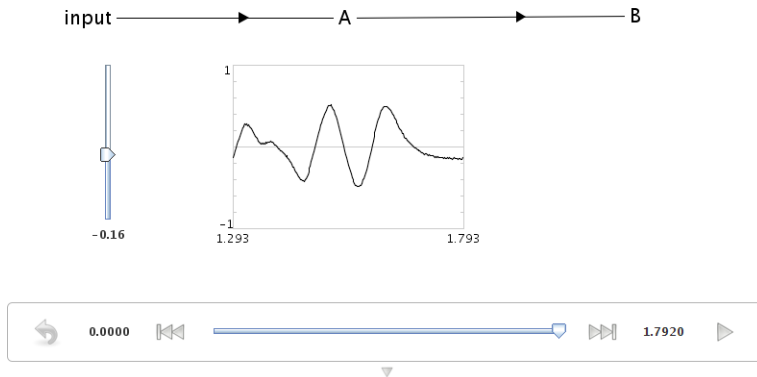
- To observe the performance of this model, we now switch over to Interactive Plots. This allows us to both graph the performance of the model and adjust its inputs on-the-fly to see how this affects behaviour.
- Start Interactive Plots by right-clicking inside the Network and selecting Interactive Plots



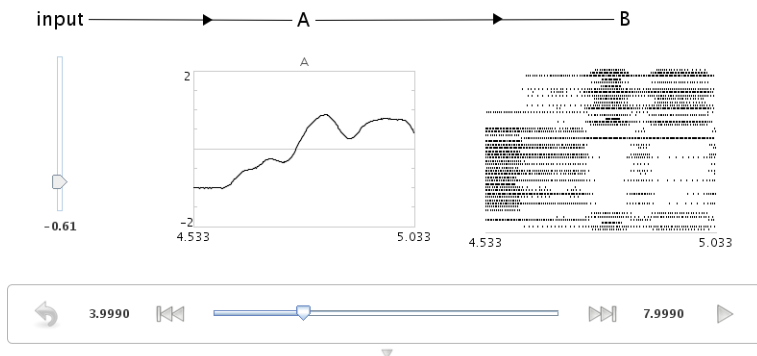
- The text shows the various components of your model, and the arrows indicate the synaptic connections between them.
 - You can move the components by left-click dragging them, and you can move all the components by dragging the background.
 - You can hide a component by right-clicking on it and selecting “hide”
 - To show a hidden component, right click on the background and select the component by name
- The bottom of the window shows the controls for running the simulation.
 - The simulation can be started and stopped by pressing the Play or Pause button at the bottom right. Doing this right now will run the simulation, but no data will be displayed since we don’t have any graphs open yet!
 - The reset button on the far left clears all the data from the simulation and puts it back to the beginning.
 - In the middle is a slider that shows the current time in the simulation. Once a simulation has been run, we can slide this back and forth to observe data from different times in the simulation.
- Right-clicking on a component also allows us to select a type of data to show about that component.
 - Right-click on A and select “value”. This creates a graph that shows the value being represented by the neuron in ensemble A. You can move the graph by left-click dragging it, and you can resize it by dragging near the corners or using a mouse scroll wheel.
 - Press the Play button at the bottom-right of the window and confirm that this group of neurons successfully represents its input value, which we previously set to be 0.5.



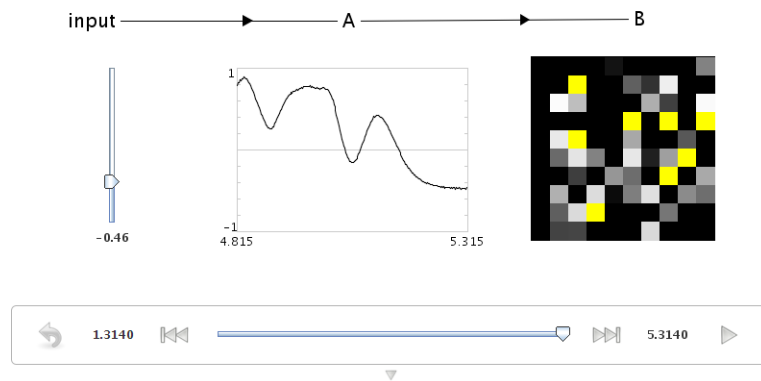
- Now let us see what happens if we change the input. Right-click on the input and select “control”. This lets us vary the input while the simulation is running.
- Drag the slider up and down while the simulation is running (press Play again if it is paused). The neurons in ensemble A should be able to successfully represent the changing values.



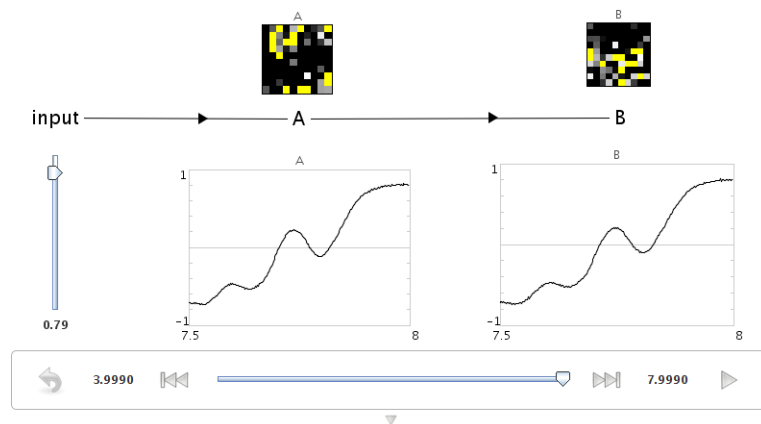
- We can also view what the individual neurons are doing during the simulation. Right-click on A and choose “spike raster”. This shows the individual spikes coming from the neurons. Since there are 100 neurons in ensemble A, the spikes from only a sub-set of these are shown. You can right-click on the spike raster graph and adjust the proportion of spikes shown. Change it to 50%.
- Run the simulation and change the input. This will affect the neuron firing patterns.



- We can also see the voltage levels of all the individual neurons. Right-click on A and choose “voltage grid”. Each neuron is shown as a square and the shading of that square indicates the voltage of that neuron’s cell membrane, from black (resting potential) to white (firing threshold). Yellow indicates a spike.
- The neurons are initially randomly ordered. You can change this by right-clicking on the voltage grid and selecting “improve layout”. This will attempt to re-order the neurons so that neurons with similar firing patterns are near each other, as they are in the brain. This does not otherwise affect the simulation in any way.
- Run the simulation and change the input. This will affect the neuron voltage.



- So far, we have just been graphing information about neural ensemble A. We have shown that these 100 neurons can accurately represent a value that is directly input to them.
- For this to be useful for constructing cognitive models, we need to also show that the spiking output from this group of neurons can be used to transfer this information from one neural group to another.
 - In other words, we want to show that B can represent the same thing as A, where B's only input is the neural firing from group A. For this to happen, the correct synaptic connection weights between A and B (as per the Neural Engineering Framework) must be calculated.
 - Nengo automatically calculates these weights whenever an origin is created.
- We can see that this communication is successful by creating graphs for ensemble B.
 - Do this by right-clicking on B and selecting “value”, and then right-clicking on B again and selecting “voltage grid”.
 - To aid in identifying which graph goes with which ensemble, right click on a graph and select “label”.
 - Graphs can be moved (by dragging) and resized (by dragging near the edges and corners or by the mouse scroll wheel) as desired.

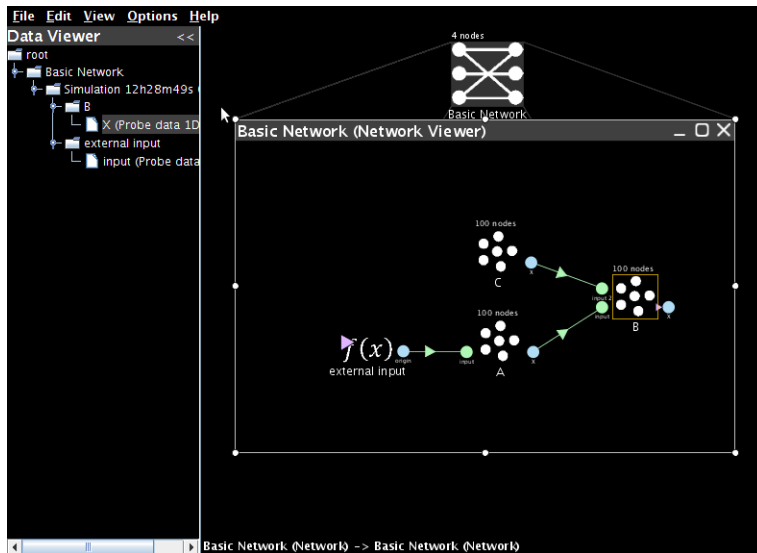


- Notice that the neural ensembles can be representing the same value, but have a different firing pattern.
- Close the Interactive Plots when you are finished.

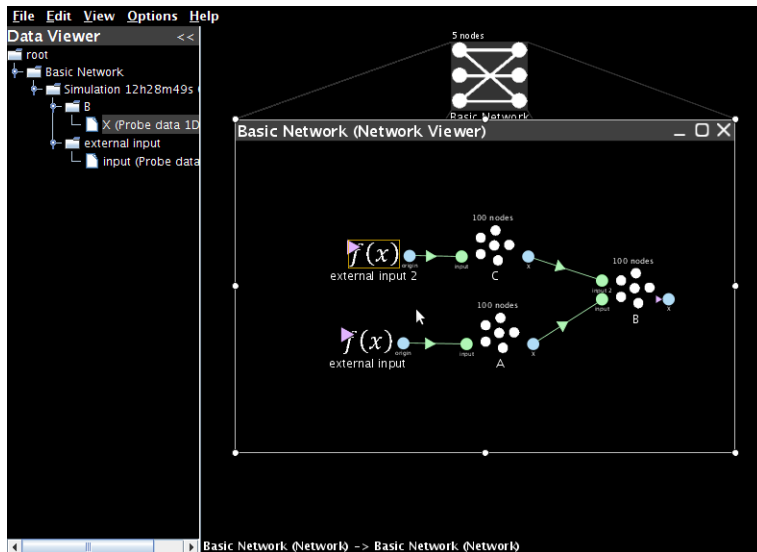
1.2.5 Adding Scalars

- If we want to add two values, we can simply add another termination to the final ensemble and project to it as well.

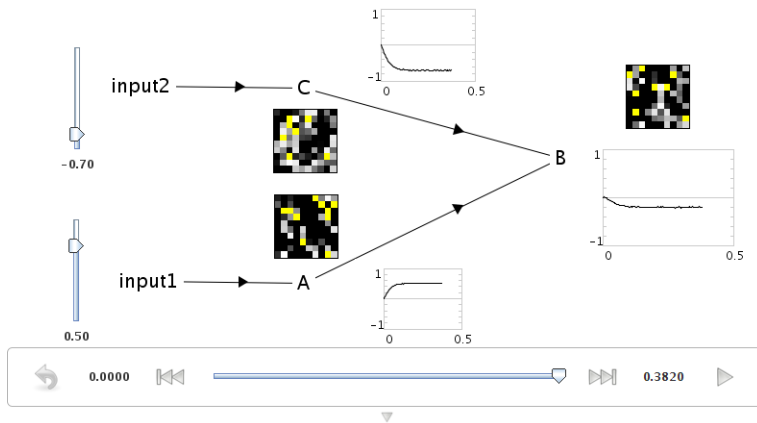
- Create a termination on the second ensemble called “input 2”
- Create a new ensemble
- Create a projection from the X origin to input 2



- Create a new Function input and set its value to -0.7
- Add the required termination and projection to connect it to the new ensemble



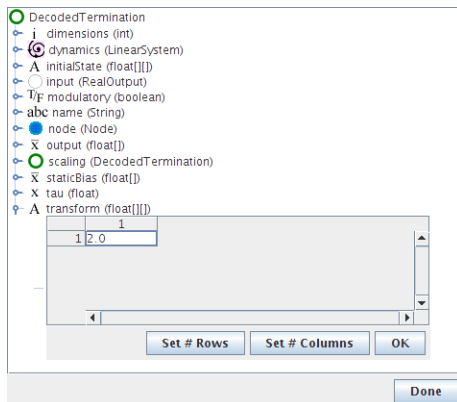
- Switch to Interactive Plots.
- Show the controls for the two inputs
- Create value graphs for the three neural ensembles
- Press Play to start the simulation. The value for the final ensemble should be $0.5 - 0.7 = -0.2$
- Use the control sliders to adjust the input. The output should still be the sum of the inputs.



- This will be true for most values. However, if the sum is outside of the radius that was set when the neural group was formed (in this case, from -1 to 1), then the neurons may not be able to fire fast enough to represent that value (i.e. they will saturate). Try this by computing $1+1$. The result will only be around 1.3.
- To accurately represent values outside of the range -1 to 1, we need to change the radius of the output ensemble. Return to the standard black editing mode and right-click on ensemble B. Select “Configure” and change its radii to 2. Now return to the Interactive Plots. The network should now accurately compute that $1+1=2$.

1.2.6 Adjusting Transformations

- So far, we have only considered projections that do not adjust the values being represented in any way. However, due to the NEF derivation of the synaptic weights between neurons, we can adjust these to create arbitrary linear transformations (i.e. we can multiply any represented value by a matrix).
- Each termination in Nengo has an associated transformation matrix. This can be adjusted as desired. In this case, we will double the weight of the original value, so instead of computing $x+y$, the network will compute $2x+y$.
- Right-click on the first termination in the ensemble that has two projections coming into it. Select Configure. Double-click on transform.
- Double-click on the 1.0 and change it to 2.0



- Click on OK and then Done
- Now run the simulation. The final result should be $2(0.5)-0.7=0.3$

1.2.7 Multiple Dimensions

- Everything discussed above also applies to ensembles that represent more than one dimension.
- To create these, set the number of dimensions to 2 when creating the ensemble

Templates
last_used ▼ New Remove

Name
D

Number of Nodes
200

Dimensions
2

Node Factory
LIF Neuron ▼ Set

Radius
1.0

Ok Cancel Advanced

- When adding a termination, the input dimension can be adjusted. This defines the shape of the transformation matrix for the termination, allowing for projections that change the dimension of the data

Templates
last_used ▼ New Remove

Name
Input

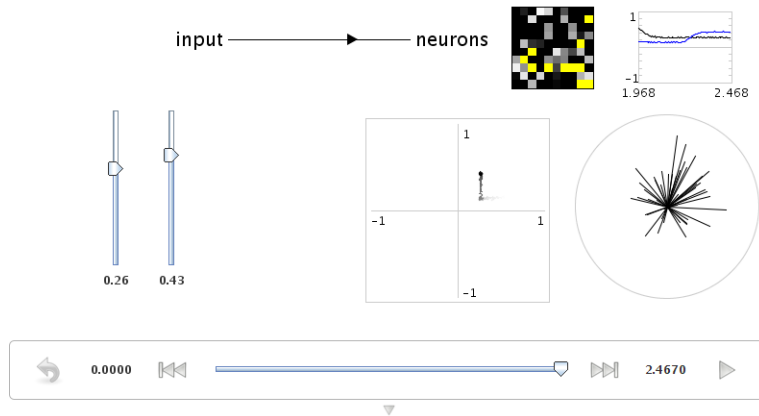
Weights
Input Dim: 5 Set Weights

tauPSC
0.01

Is Modulatory
☐ Enable

Ok Cancel

- For example, two 1-dimensional values can be combined into a single two-dimensional ensemble. This would be done with two terminations: one with a transformation (or coupling) matrix of $\begin{bmatrix} 1 & 0 \end{bmatrix}$ and the other with $\begin{bmatrix} 0 & 1 \end{bmatrix}$. If the two inputs are called a and b, this will result in the following calculation:
 - $a * \begin{bmatrix} 1 & 0 \end{bmatrix} + b * \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} a & 0 \end{bmatrix} + \begin{bmatrix} 0 & b \end{bmatrix} = \begin{bmatrix} a & b \end{bmatrix}$
 - This will be useful for creating non-linear transformations, as discussed further in the next section.
- There are additional ways to view 2D representations in the interactive plots
 - Including plotting the activity of the neurons along their preferred direction vectors
 - Plotting the 2D decoded value of the representation



1.2.8 Scripting

- Along with the ability to construct models using this point-and-click interface, Nengo also provides a Python scripting language interface for model creation. These examples can be seen in the “demo” directory.
- To create the communication channel through the scripting interface, go to the Script Console (Ctrl-P) and type:

```
run demo/communication.py
```

- The actual code for this can be seen by opening the communication.py file in the demo directory:

```
import nef

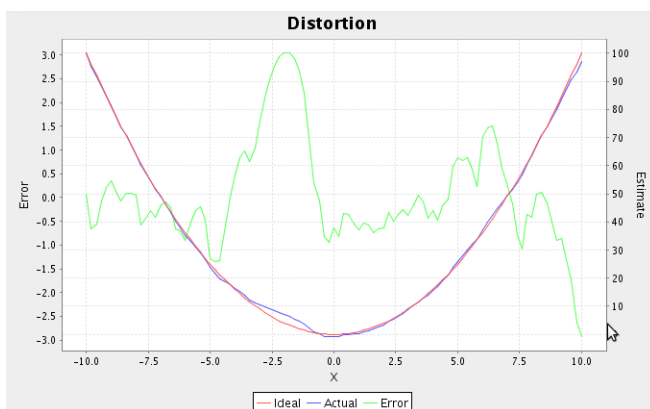
net=nef.Network('Communications Channel')
input=net.make_input('input',[0.5])
A=net.make('A',100,1)
B=net.make('B',100,1)
net.connect(input,A)
net.connect(A,B)
net.add_to(world)
```

- The following demo scripts create models similar to those seen in this part of the tutorial:
 - demo/singleneuron.py shows what happens with an ensemble with only a single neuron on it (poor representation)
 - demo/twoneurons.py shows two neurons working together to represent
 - demo/manyneurons.py shows a standard ensemble of 100 neurons representing a value
 - demo/communication.py shows a communication channel
 - demo/addition.py shows adding two numbers
 - demo/2drepresentation.py shows 100 neurons representing a 2-D vector
 - demo/combining.py shows two separate values being combined into a 2-D vector

1.3 Non-Linear Transformations

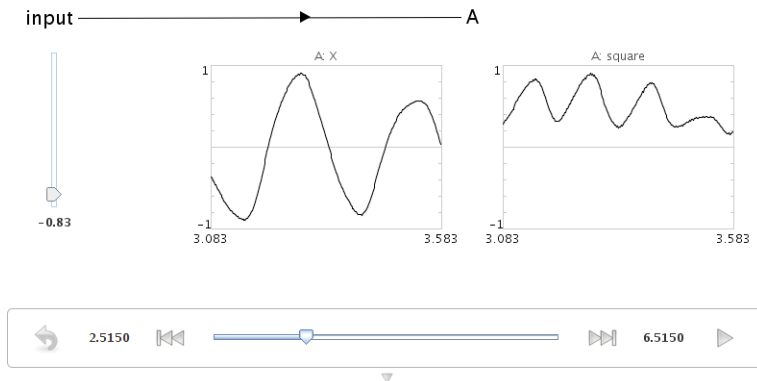
1.3.1 Functions of one variable

- We now turn to creating nonlinear transformations in Nengo. The main idea here is that instead of using the X origin, we will create a new origin that estimates some arbitrary function of X. This will allow us to estimate any desired function.
 - The accuracy of this estimate will, of course, be dependent on the properties of the neurons.
- For one-dimensional ensembles, we can calculate various 1-dimensional functions:
 - $f(x) = x^2$
 - $f(x) = \theta(x)$ (thresholding)
 - $f(x) = \sqrt{x}$
- To perform a non-linear operation, we need to define a new origin
 - The X origin just uses $f(x) = x$.
 - Create a new ensemble and a function input. The ensemble should be one-dimensional with 100 neurons and a radius of 1. Use a Constant Function input set two 0.5.
 - Create a termination on the ensemble and connect the function input to it
 - Now create a new origin that will estimate the square of the value.
 - * Right-click on the combined ensemble and select Add decoded origin
 - * Set the name to square
 - * Click on Set Functions
 - * Select User-defined Function and press Set
 - * For the Expression, enter $x0 * x0$. We refer to the value as $x0$ because when we extend this to multiple dimensions, we will refer to them as $x0$, $x1$, $x2$, and so on.
 - * Press OK, OK, and OK.
 - You can now generate a plot that shows how good the ensemble is at calculating the non-linearity. Right-click on the ensemble and select Plot->Plot distortion:square.



- Start Interactive Plots.
- Create a control for the input, so you can adjust it while the model runs (right-click on the input and select “control”)

- Create a graph of the “square” value from the ensemble. Do this by right-clicking on the ensemble in the Interactive Plots window and selecting “square->value”.
- For comparison, also create a graph for the standard X origin by right-clicking on the ensemble and selecting “X->value”. This is the standard value graph that just shows the value being represented by this ensemble.
- Press Play to run the simulation. With the default input of 0.5, the squared value should be near 0.25. Use the control to adjust the input. The output should be the square of the input.

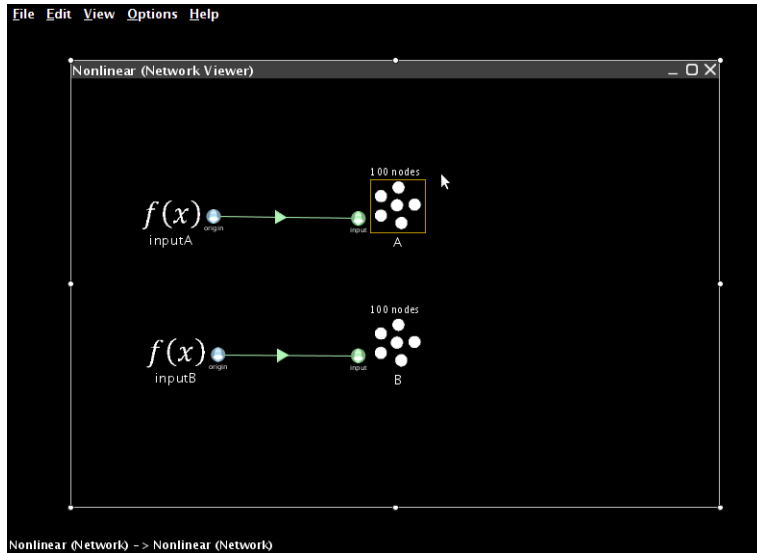


- You can also run this example using scripting:

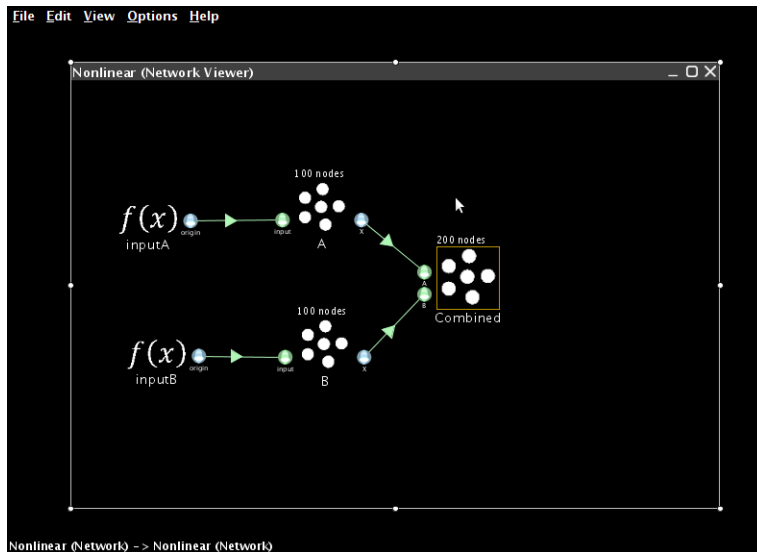
```
run demo/squaring.py
```

1.3.2 Functions of multiple variables

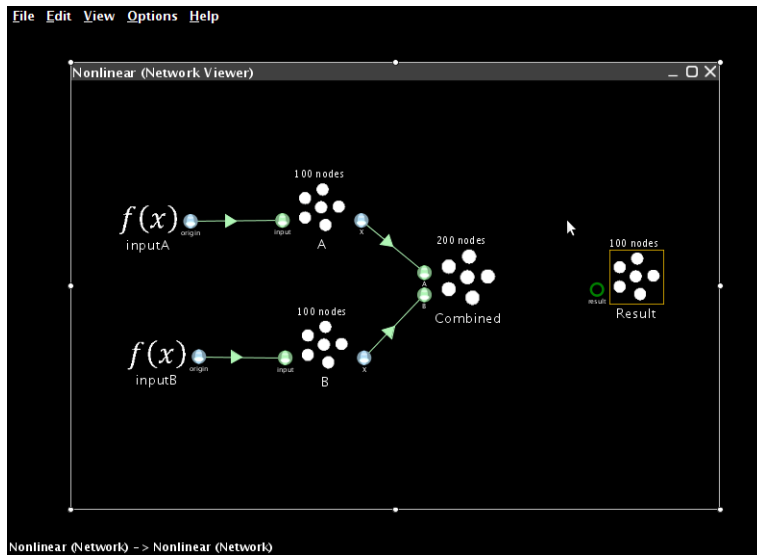
- Since X (the value being represented by an ensemble) can also be multidimensional, we can also calculate these sorts of functions
 - $f(x) = x_0 * x_1$
 - $f(x) = \max(x_0, x_1)$
- To begin, we create two ensembles and two function inputs. These will represent the two values we wish to multiply together.
 - The ensembles should be one-dimensional, use 100 neurons and have a radius of 10 (so they can represent values between -10 and 10)
 - The two function inputs should be constants set to 8 and 5
 - The terminations you create to connect them should have time constants of 0.01 (AMPA)



- Now create a two-dimensional neural ensemble with a radius of 15 called Combined
 - Since it needs to represent multiple values, we increase the number of neurons it contains to 200
- Add two terminations to Combined
 - For each one, the input dimensions are 1
 - For the first one, use Set Weights to make the transformation be [1 0]
 - For the second one, use Set Weights to make the transformation be [0 1]
- Connect the two other ensembles to the Combined one



- Next, create an ensemble to store the result. It should have a radius of 100, since it will need to represent values from -100 to 100. Give it a single one-dimensional termination with a weight of 1.



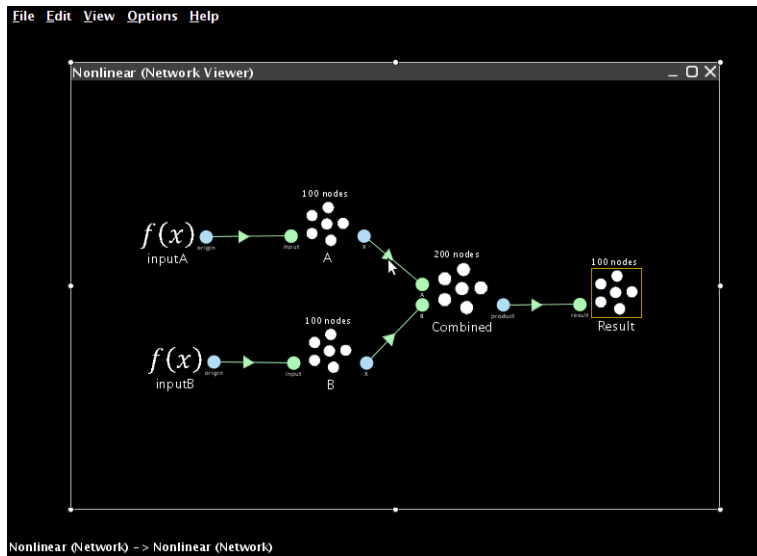
- Now we need to create a new origin that will estimate the product between the two values stored in the combined ensemble.
 - Right-click on the combined ensemble and select Add decoded origin.
 - Set the name to `product`
 - Set Output dimensions to 1

- Click on Set Functions
- Select User-defined Function and press Set.

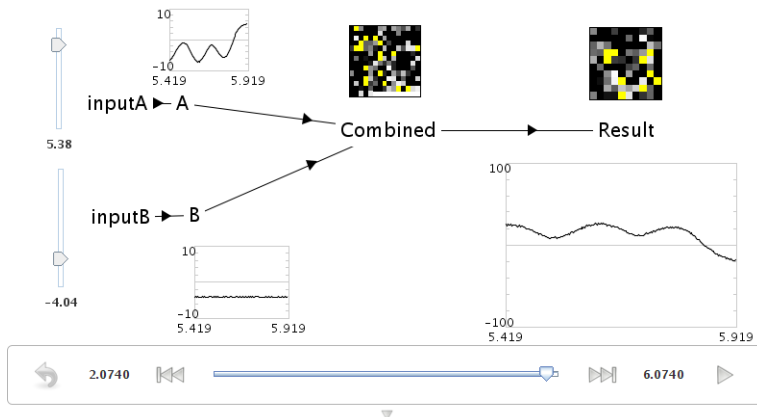
- For the Expression, enter `x0 * x1`

- Press OK, OK, and OK to finish creating the origin

- Connect the new origin to the termination on the result ensemble



- Add a probe to the result ensemble and run the simulation
- The result should be approximately 40.
- Adjust the input controls to multiple different numbers together.



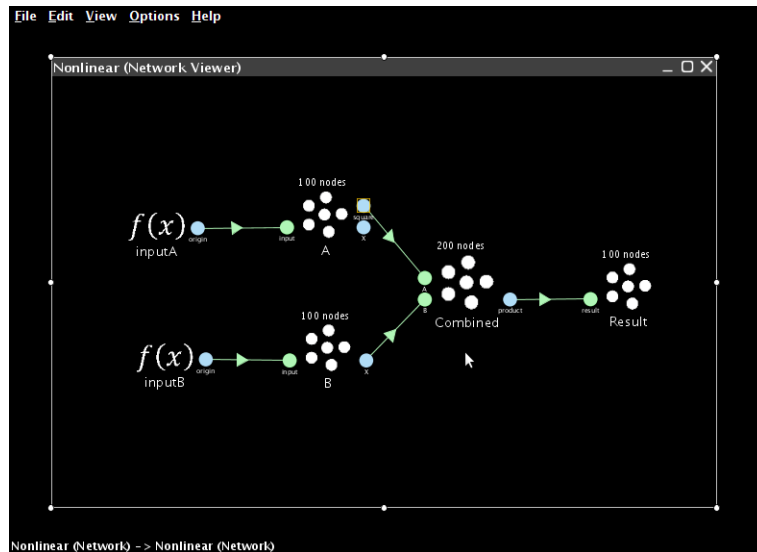
- You can also run this example using scripting:

```
run demo/multiplication.py
```

1.3.3 Combined Approaches

- We can combine these two approaches in order to compute more complex functions, such as x^2y
 - Right-click on the ensemble representing the first of the two values and select Add decoded origin.
 - Give it the name “square”, set its output dimensions to 1, and press Set Functions.
 - As before, select the User-defined Function and press Set.
 - Set the Expression to be “x0*x0”.
 - Press OK, OK, and OK to finish creating the origin.

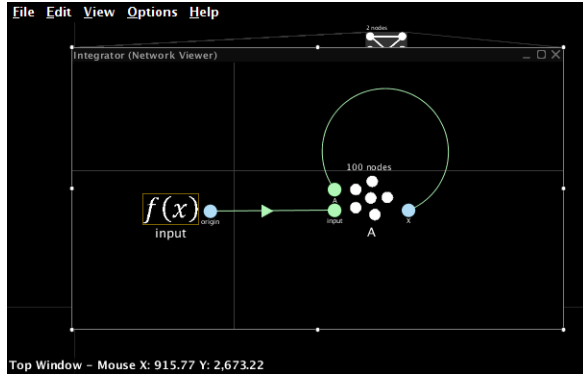
- This new origin will calculate the square of the value represented by this ensemble.
- If you connect this new origin to the Combined ensemble instead of the standard X origin, the network will calculate x^2y instead of xy .



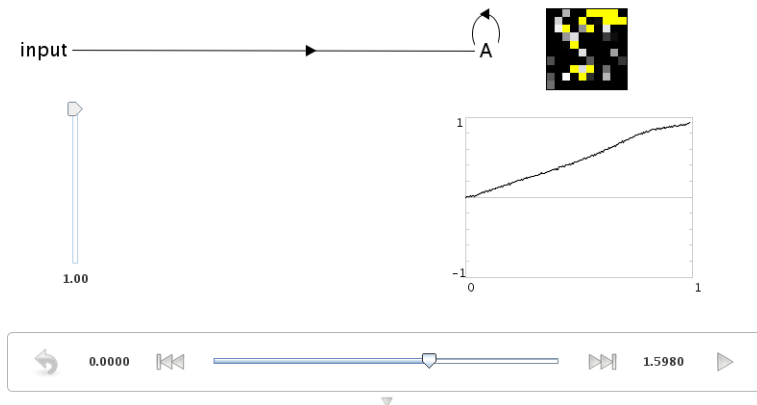
1.4 Feedback and Dynamics

1.4.1 Storing Information Over Time: Constructing an Integrator

- The basis of many of our cognitive models is the integrator. Mathematically, the output of this network should be the integral of the inputs to this network.
 - Practically speaking, this means that if the input to the network is zero, then its output will stay at whatever value it is currently at. This makes it the basis of a neural memory system, as a representation can be stored over time.
 - Integrators are also often used in sensorimotor systems, such as eye control
- For an integrator, a neural ensemble needs to connect to itself with a transformation weight of 1, and have an input with a weight of τ , which is the same as the synaptic time constant of the neurotransmitter used.
- Create a one-dimensional ensemble called Integrator. Use 100 neurons and a radius of 1.
- Add two terminations with synaptic time constants of 0.1s. Call the first one “input” and give it a weight of 0.1. Call the second one “feedback” and give it a weight of 1.
- Create a new Function input using a Constant Function with a value of 1.
- Connect the Function input to the input termination
- Connect the X origin of the ensemble back to its own feedback termination.

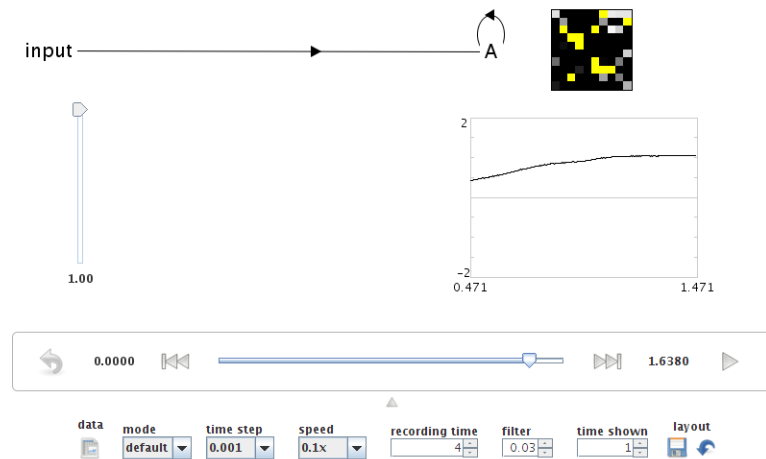


- Go to Interactive Plots. Create a graph for the value of the ensemble (right-click on the ensemble and select “value”).
- Press Play to run the simulation. The value stored in the ensemble should linearly increase, reaching a value of 1 after approximately 1 second.
 - You can increase the amount of time shown on the graphs in Interactive Plots. Do this by clicking on the small downwards-pointing arrow at the bottom of the window. This will reveal a variety of settings for Interactive Plots. Change the “time shown” to 1.

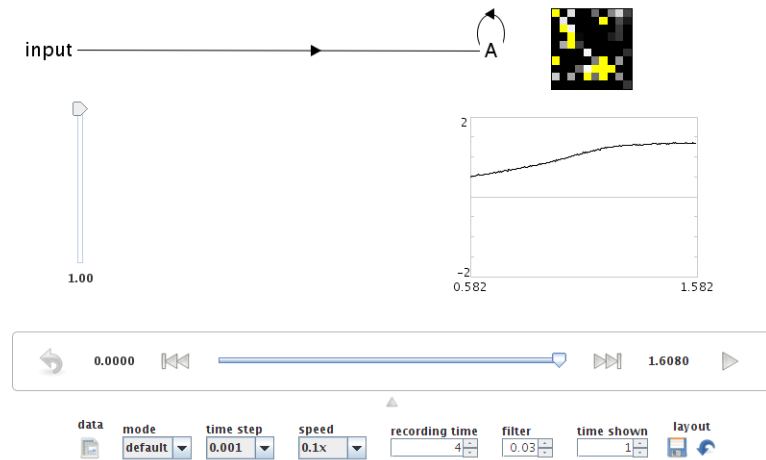


1.4.2 Representation Range

- What happens if the previous simulation runs for longer than one second?
- The value stored in the ensemble does not increase after a certain point. This is because all neural ensembles have a range of values they can represent (the radius), and cannot accurately represent outside of that range.



- Adjust the radius of the ensemble to 1.5 using either the Configure interface or the script console (that.`radii=[1.5]`). Run the model again. It should now accurately integrate up to a maximum of 1.5.

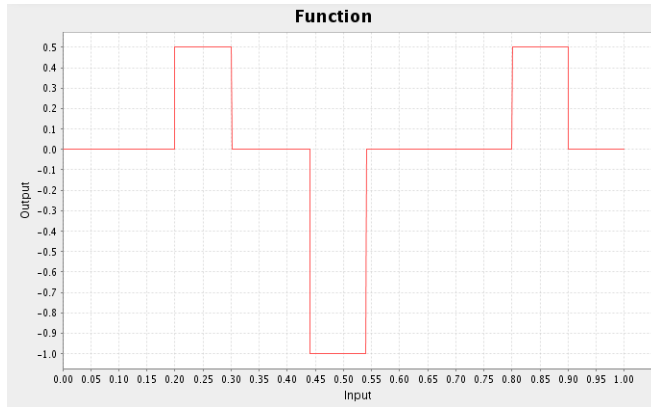


1.4.3 Complex Input

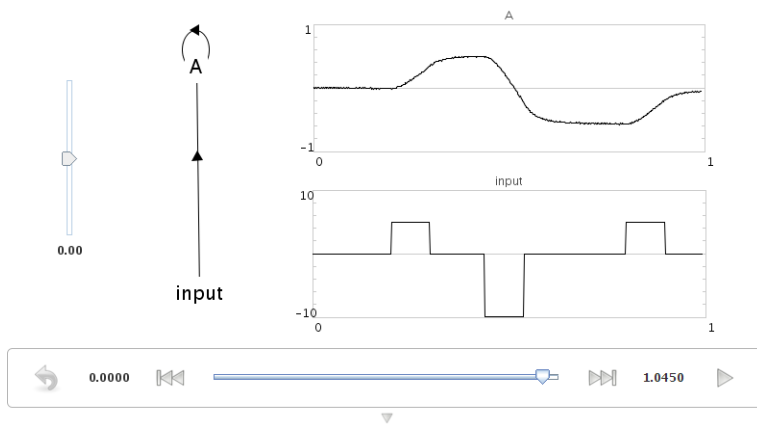
- We can also run the model with a more complex input. Change the Function input using the following command from the script console (after clicking on it in the black model editing mode interface). Press Ctrl-P to show the script console:

```
that.functions=[ca.nengo.math.impl.PiecewiseConstantFunction([0.2,0.3,0.44,0.54,0.8,0.9],[0,5,0,
```

- You can see what this function looks like by right-clicking on it in the editing interface and selecting “Plot”.

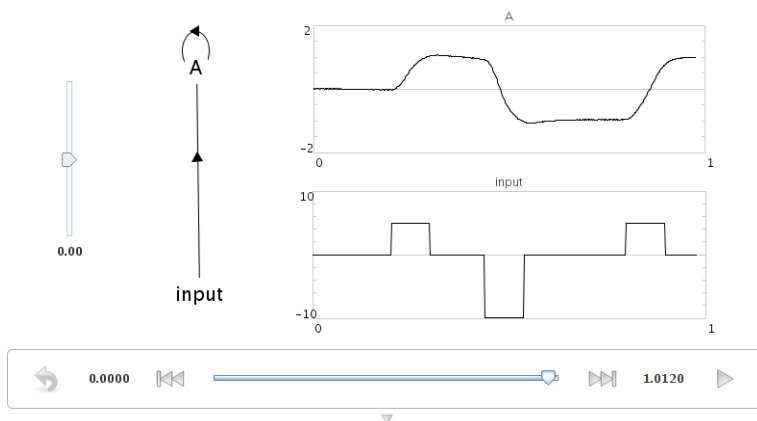


- Return to Interactive Plots and run the simulation.



1.4.4 Adjusting Synaptic Time Constants

- You can adjust the accuracy of an integrator by using different neurotransmitters.
- Change the input termination to have a tau of 0.01 (10ms: GABA) and a transform to be 0.01. Also change the feedback termination to have a tau of 0.01 (but leave its transform at 1).



- By using a shorter time constant, the network dynamics are more sensitive to small-scale variation (i.e. noise).
- This indicates how important the use of a particular neurotransmitter is, and why there are so many different types with vastly differing time constants.

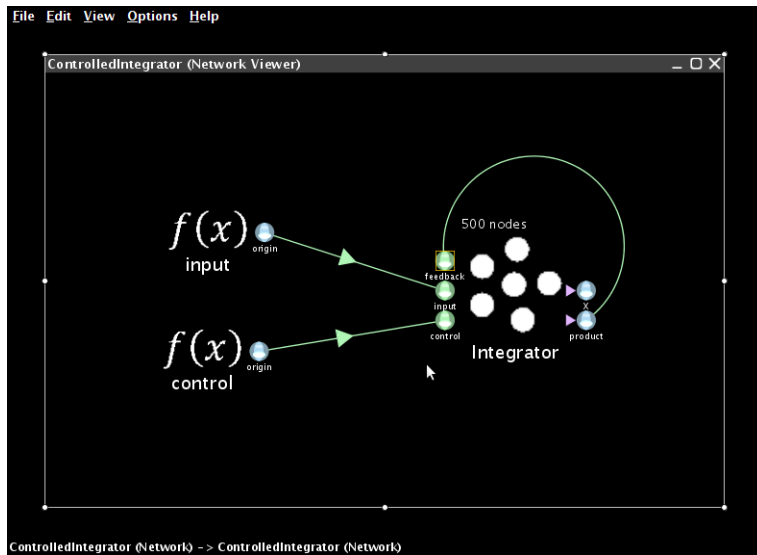
- AMPA: 2-10ms
 - GABA:subscript:A: 10-20ms
 - NMDA: 20-150ms
 - The actual details of these time constants vary across the brain as well. We are collecting empirical data on these from various sources at <http://ctn.uwaterloo.ca/~cnrglab/?q=node/505>
- You can also run this example using scripting:

```
run demo/integrator.py
```

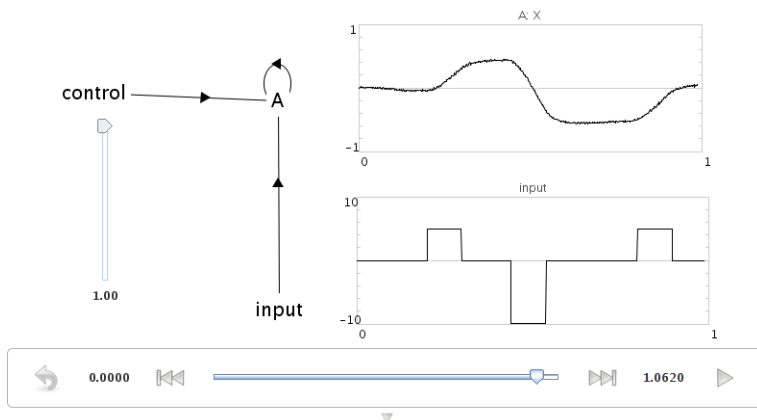
1.4.5 Controlled Integrator

- We can also build an integrator where the feedback transformation (1 in the previous model) can be controlled.
 - This allows us to build a tunable filter.
- This requires the use of multiplication, since we need to multiply two stored values together. This was covered in the previous part of the tutorial.
- We can efficiently implement this by using a two-dimensional ensemble. One dimension will hold the value being represented, and the other dimension will hold the transformation weight.
- Create a two-dimensional neural ensemble with 225 neurons and a radius of 1.5.
- Create the following three terminations:
 - `input`: time constant of 0.1, 1 dimensional, with a transformation matrix of $\begin{bmatrix} 0.1 & 0 \end{bmatrix}$. This acts the same as the input in the previous model, but only affects the first dimension.
 - `control`: time constant of 0.1, 1 dimensional, with a transformation matrix of $\begin{bmatrix} 0 & 1 \end{bmatrix}$. This stores the input control signal into the second dimension of the ensemble.
 - `feedback`: time constant of 0.1, 1 dimensional, with a transformation matrix of $\begin{bmatrix} 1 & 0 \end{bmatrix}$. This will be used in the same manner as the feedback termination in the previous model.
- Create a new origin that multiplies the values in the vector together
 - This is exactly the same as the multiplier in the previous part of this tutorial
 - This is a 1 dimensional output, with a User-defined Function of $x_0 \times x_1$
- Create two function inputs called `input` and `control`. Start with Constant functions with a value of 1
 - Use the script console to set the `input` function by clicking on it and entering the same input function as used above:

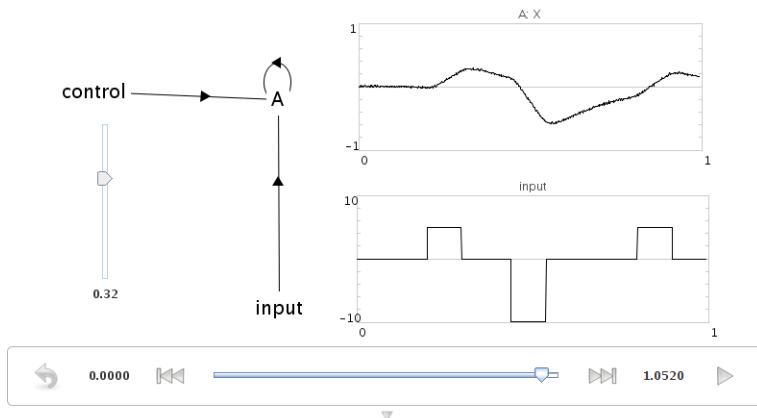
```
that.functions=[ca.nengo.math.impl.PiecewiseConstantFunction([0.2,0.3,0.44,0.54,0.8,0.9],[0,
```
- Connect the input function to the input termination, the control function to the control termination, and the product origin to the feedback termination.



- Go to Interactive Plots and show a graph for the value of the ensemble (right-click->X->value). If you run the simulation, this graph will show the values of both variables stored in this ensemble (the integrated value and the control signal). For clarity, turn off the display of the control signal by right-clicking on the graph and removing the checkmark beside “v[1]”.
- The performance of this model should be similar to that of the non-controlled integrator.



- Now adjust the control input to be 0.3 instead of 1. This will make the integrator into a leaky integrator. This value adjusts how quickly the integrator forgets over time.



- You can also run this example using scripting:

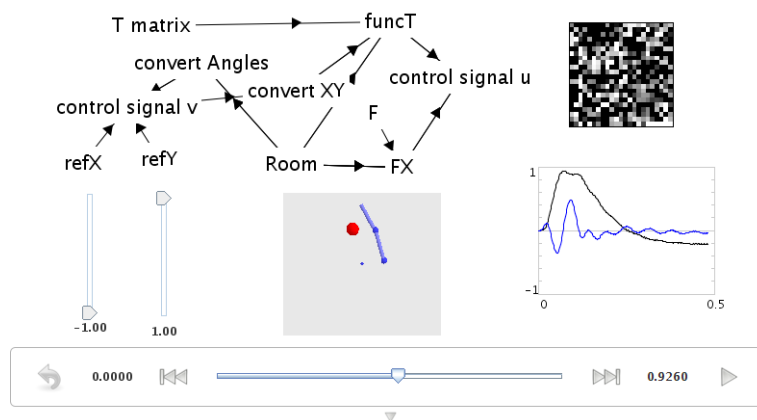
```
run demo/controlledintegrator.py
```

1.5 Cognitive Models

1.5.1 Larger Systems

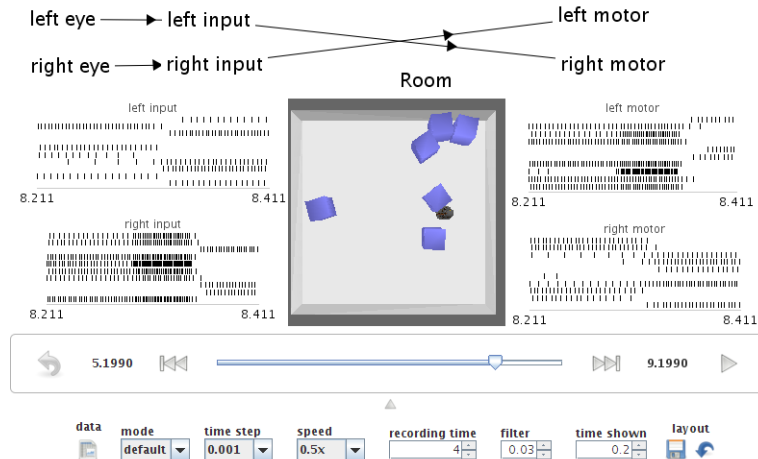
- So far, we've seen how to implement the various basic components
 - representations
 - linear transformation
 - non-linear transformation
 - feedback
- The goal is to use these components to build full cognitive models using spiking neurons
 - Constrained by the actual properties of real neurons in real brains (numbers of neurons, connectivity, neurotransmitters, etc)
 - Should be able to produce behavioural predictions in terms of timing, accuracy, lesion effects, drug treatments, etc
- Some simple examples
 - Motor control
 - * take an existing engineering control model for what angles to move joints to to place the hand at a particular position:

```
run demo/armcontrol.py
```



- Braitenberg vehicle
 - * connect range sensors to opposite motors on a wheeled robot:

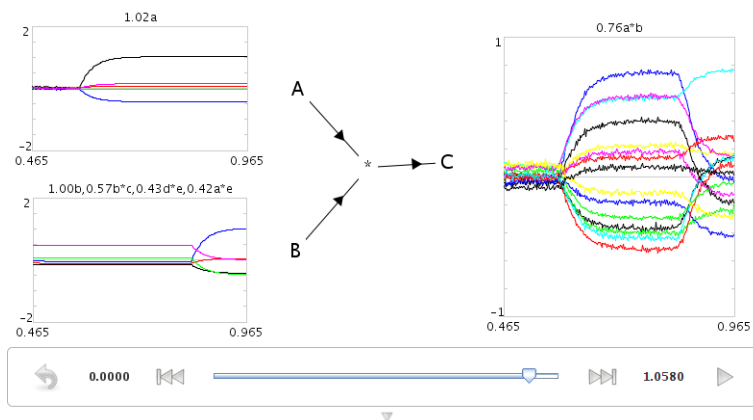
```
run demo/vehicle.py
```



1.5.2 Binding Semantic Pointers (SPs)

- We want to manipulate sophisticated representational states (this is the purpose of describing the semantic pointer architecture (SPA))
- The main operation to manipulate representations in the SPA is circular convolution (for binding)
- Let's explore a binding circuit for semantic pointers
- Input: Two semantic pointers (high-dimensional vectors)
- Output: One semantic pointer (binding the original two)
- Implementation: element-wise multiplication of DFT (as described in slides):

```
run demo/convolve.py
```

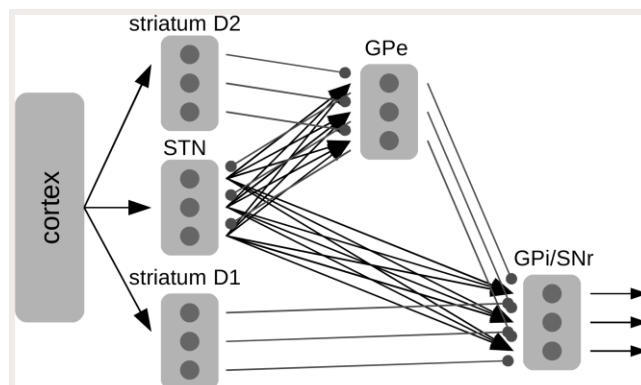


- To deal with high-dimensional vectors, we don't want to have to set each individual value for each vector
 - would need 100 controls to configure a single 100-dimensional vector
- Nengo has a specialized “semantic pointer” graph for these high-dimensional cases
 - Instead of showing the value of each element in the vector (as with a normal graph), it shows the similarity between the currently represented vector and all the known vectors
 - “How much like CAT is this? How much like DOG? How much like RED? How much like TRIANGLE?”
 - You can configure which comparisons are shown using the right-click menu

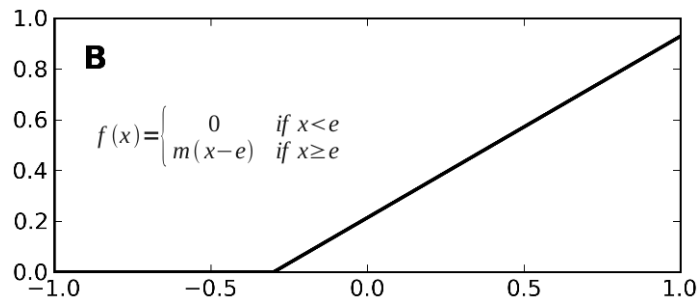
- You can also use it to `_set_` the contents of a neural group by right-clicking and choosing “set value”. This will force the neurons to represent the given semantic pointer. You can go back to normal behaviour by selecting “release value”.
- Use the right-click menu to set the input values to “a” and “b”. The output should be similar to “a*b”.
 - This shows that the network is capable of computing the circular convolution operation, which binds two semantic pointers to create a third one.
- Use the right-click menu to set the input values to “a” and “~a*b”. The output should be similar to “b”.
 - This shows that convolution can be used to transform representations via binding and unbinding, since “a*(~a*b)” is approximately “b”.

1.5.3 Control and Action Selection: Basal Ganglia

- Pretty much every cognitive model has an action selection component
 - Out of many possible things you could do right now, pick one
 - Usually mapped on to the basal ganglia
 - Some sort of winner-take-all calculation based on how suitable the various possible actions are to the current situation
- Input: A vector representing how good each action is (for example, [0.2, 0.3, 0.9, 0.1, 0.7])
- Output: Which action to take ([0, 0, 1, 0, 0])
 - Actually, the output from the basal ganglia is inhibitory, so the output is more like [1, 1, 0, 1, 1]
- Implementation
 - Could try doing it as a direct function
 - * Highly non-linear function
 - * Low accuracy
 - Could do it by setting up inhibitory interconnections
 - * Like the integrator, but any value above zero would also act to decrease the others
 - * Often used in non-spiking neural networks (e.g. PDP++) to do k-winner-take-all
 - * But, you have to wait for the network to settle, so it can be rather slow
 - Gurney, Prescott, & Redgrave (2001)
 - * Model of action selection constrained by the connectivity of the basal ganglia



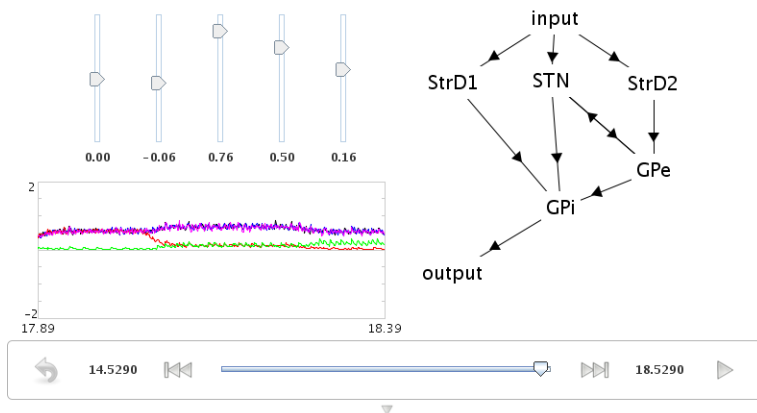
- Each component computes the following function



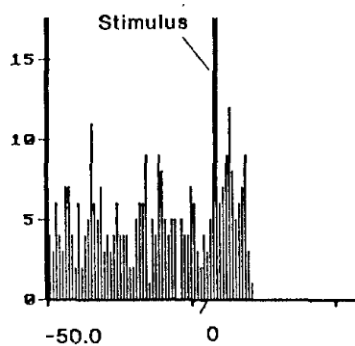
- Their model uses unrealistic rate neurons with that function for an output
- We can use populations of spiking neurons and compute that function
- We can also use correct timing values for the neurotransmitters involved:

```
run demo/basalganglia.py
```

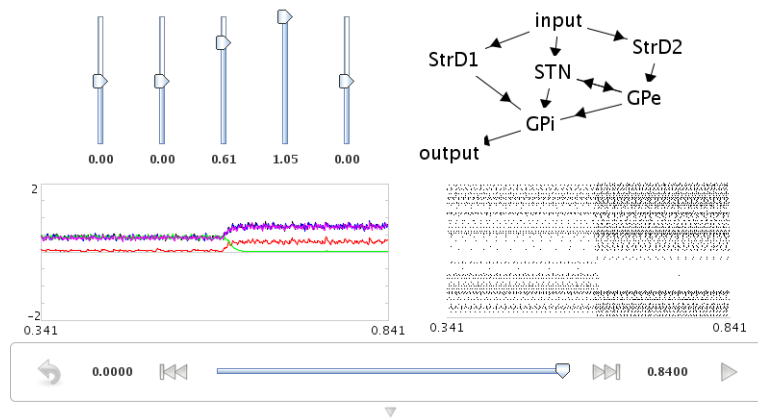
- Adjust the input controls to change the five utility values being selected between
- Graph shows the output from the basal ganglia (each line shows a different action)
- The selected action is the one set to zero



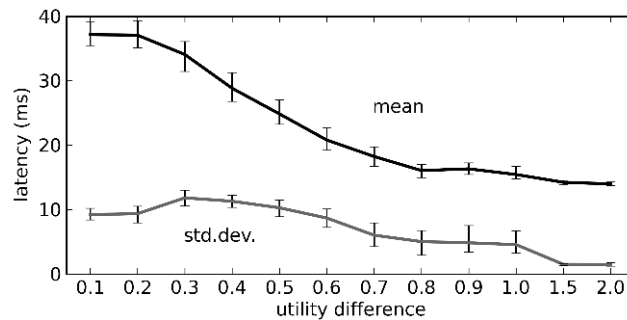
- Comparison to neural data
 - Ryan & Clark, 1991
 - Stimulate regions in medial orbitofrontal cortex, measure from GPi, see how long it takes for a response to occur



- To replicate
 - Set the inputs to [0, 0, 0.6, 0, 0]
 - Run simulation for a bit, then pause it
 - Set the inputs to [0, 0, 0.6, 1, 0]
 - Continue simulation
 - Measure how long it takes for the neurons for the fourth action to stop firing



- In rats: 14-17ms. In model: 14ms (or more if the injected current isn't extremely large)



1.5.4 Sequences of Actions

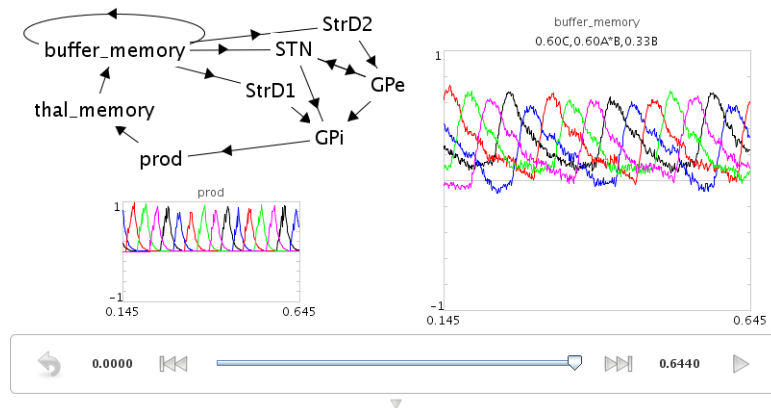
- To do something useful with the action selection system we need two things
 - A way to determine the utility of each action given the current context
 - A way to take the output from the action selection and have it affect behaviour
- We do this using the representations of the semantic pointer architecture
 - Any cognitive state is represented as a high-dimensional vector (a semantic pointer)
 - Working memory stores semantic pointers (using an integrator)
 - Calculate the utility of an action by computing the dot product between the current state and the state for the action (i.e. the IF portion of an IF-THEN production rule)
 - * This is a linear operation, so we can directly compute it using the connection weights between the cortex and the basal ganglia

- The THEN portion of a rule says what semantic pointers to send to what areas of the brain. This is again a linear operation that can be computed on the output of the thalamus using the output from the basal ganglia

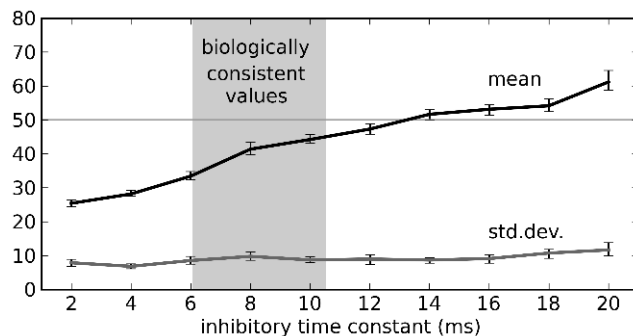
- Simple example:

- Five possible states: A, B, C, D, and E
- Rules for IF A THEN B, IF B THEN C, IF C THEN D, IF D THEN E, IF E THEN A
- Five *production rules* (semantic pointer mappings) cycling through the five states:

```
run demo/sequence.py
```



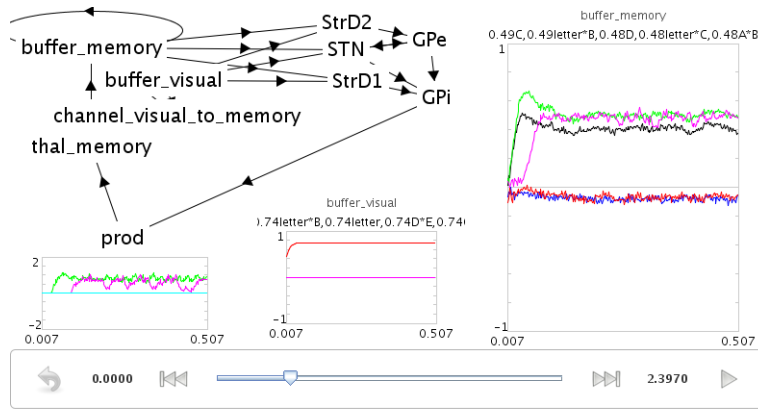
- Can set the contents of working memory in Interactive Plots by opening an SP graph, right-clicking on it, and choosing “set value” (use “release value” to allow the model to change the contents)
- Cycle time is around 40ms, slightly faster than the standard 50ms value used in ACT-R, Soar, EPIC, etc.
 - This depends on the time constant for the neurotransmitter GABA



1.5.5 Routing of Information

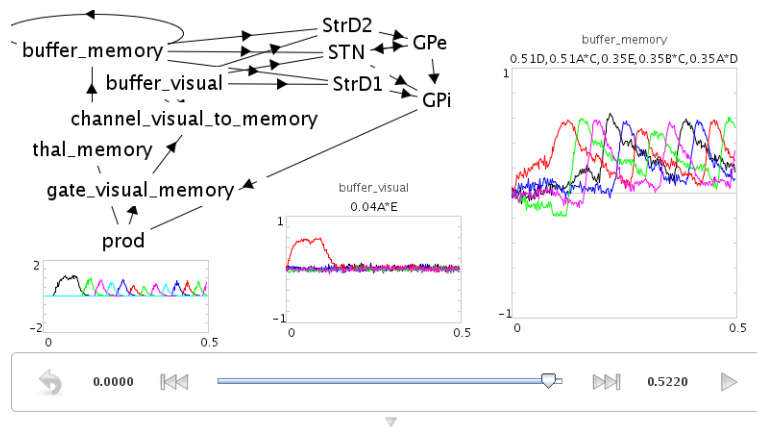
- What about more complex actions?
 - Same model as above, but we want visual input to be able to control where we start the sequence
 - Simple approach: add a visual buffer and connect it to the working memory:

```
run demo/sequencenogate.py
```

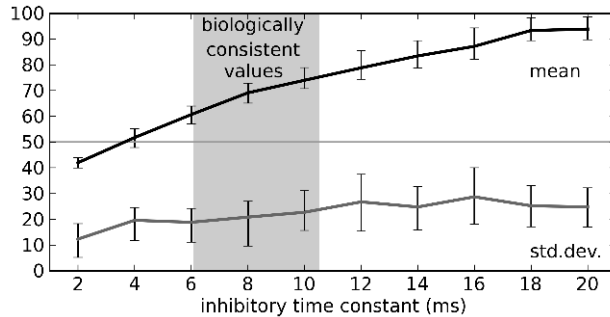


- Problem: If this connection always exists, then the visual input will always override what's in working memory. This connection needs to be controllable
- Solution
 - Actions need to be able to control the flow of information between cortical areas.
 - Instead of sending a particular SP to working memory, we need “IF X THEN transfer the pattern in cortex area Y to cortex area Z”?
 - In this case, we add a rule that says “IF it contains a letter, transfer the data from the visual area to working memory”
 - We make the utility of the rule lower than the utility of the sequence rules, so that it will only transfer that information (open that gate) when no other action applies:

```
run demo/sequencerouted.py
```



- The pattern in the visual buffer is successfully transferred to working memory, then the sequence is continued from that letter.

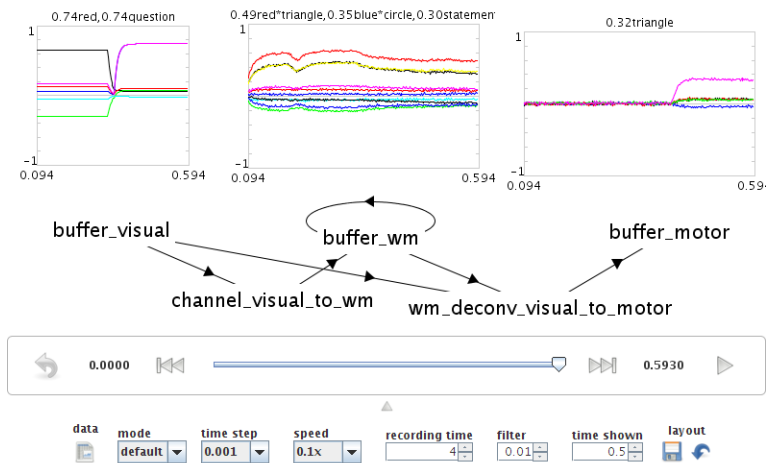


- Takes longer (60-70ms) for these more complex productions to occur

1.5.6 Question Answering




- The control signal in the previous network can also be another semantic pointer that binds/unbinds the contents of the visual buffer (instead of just a gating signal)
 - This more flexible control does not add processing time
 - Allows processing the representations while routing them
- This allows us to perform arbitrary symbol manipulation such as “take the contents of buffer X, unbind it with buffer Y, and place the results in buffer Z”
- Example: Question answering
 - System is presented with a statement such as “red triangle and blue circle”
 - * a semantic pointer representing this statement is placed in the visual cortical area
 - * `statement+red*triangle+blue*circle`
 - Statement is removed after a period of time
 - Now a question is presented, such as “What was Red?”
 - * `question+red` is presented to the same visual cortical area as before
 - Goal is to place the correct answer in a motor cortex area (in this case, “triangle”)
- This is achieved by creating two action rules:
 - If a statement is in the visual area, move it to working memory (as in the previous example)
 - If a question is in the visual area, unbind it with working memory and place the result in the motor area
- This example requires a much larger simulation than any of the others in this tutorial (more than 50,000 neurons). If you run this script, Nengo may take a long time (hours!) to solve for the decoders and neural connection weights needed. We have pre-computed the larger of these networks for you, and they can be downloaded at <http://ctn.uwaterloo.ca/~cnrglab/f/question.zip>:

```
run demo/question.py
```



NENGO DEMOS

This section describes the collection of demos that comes with Nengo. To use any of these demo scripts in Nengo, do the following:

- **Open** Open any <demo>.py file by clicking on the  icon (or going to File->Open from file in the menu) and selecting the file from demos directory in your Nengo installation.
- **Run** Run the demo by selecting the network created in the previous step and then clicking the  icon in the upper right corner of the Nengo main window. Alternatively, right-click on the network and select 'Interactive Plots'. Click the  arrow to start the simulation.

Note: You don't need to select the network if there is only one available to run.

- **Delete** Remove the demo network after using it by right clicking and selecting 'Remove model'.

Note: You don't need to remove a model if you reload the same script again, it will automatically be replaced.

More sophisticated examples can be found in the Model Archive at <http://models.nengo.ca>.

2.1 Introductory Demos

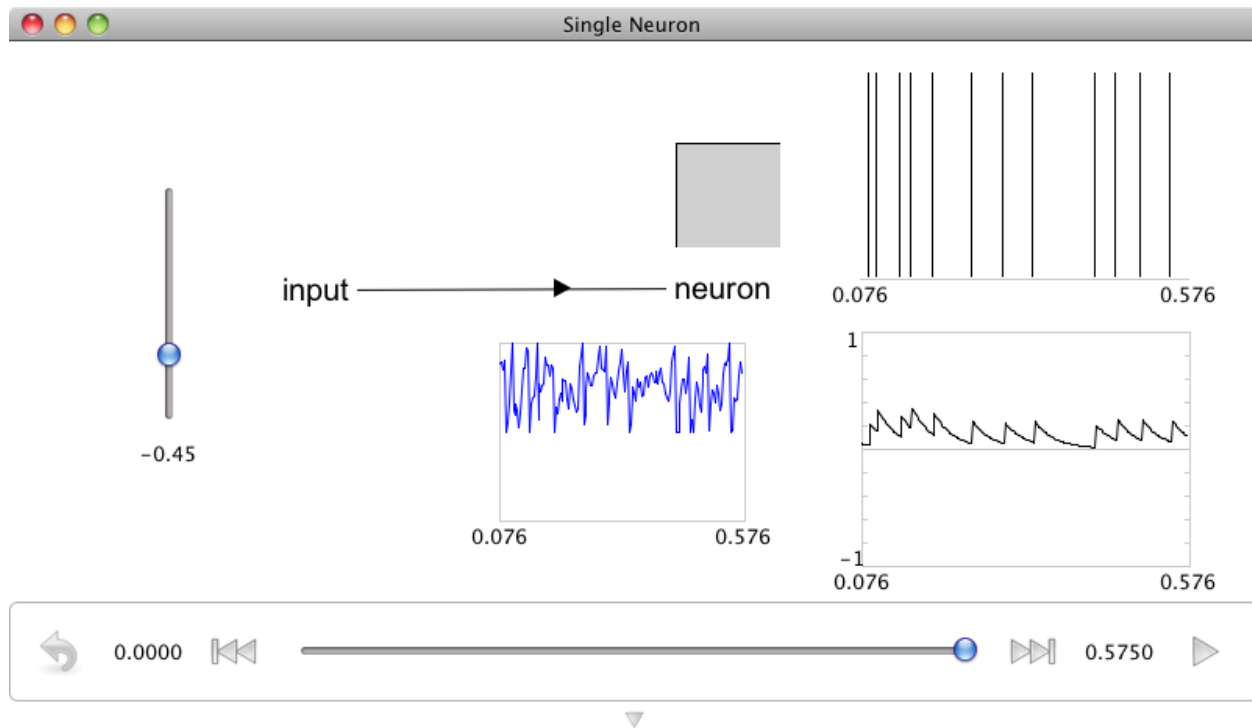
2.1.1 A Single Neuron

Purpose: This demo shows how to construct and manipulate a single neuron.

Comments: This leaky integrate-and-fire (LIF) neuron is a simple, standard model of a spiking single neuron. It resides inside a neural 'population', even though there is only one neuron.

Usage: Grab the slider control and move it up and down to see the effects of increasing or decreasing input. This neuron will fire faster with more input (an 'on' neuron).

Output: See the screen capture below



Code:

```
import nef

net=nef.Network('Single Neuron') #Create the network object
input=net.make_input('input', [-0.45]) #Create a controllable input function
                                         #with a starting value of -.45
neuron=net.make('neuron',1,1,max_rate=(100,100),intercept=(-0.5,-0.5),
               encoders=[[1]],noise=3) #Make 1 neuron, 1 dimension, a max firing
                                         #rate evenly distributed between 100 and 100,
                                         #an x-intercept evenly distributed between -.5
                                         #and -.5, an encoder of 1 and noise at every
                                         #step with a variance of 3
net.connect(input,neuron) #Connect the input to the neuron
net.add_to_nengo()
```

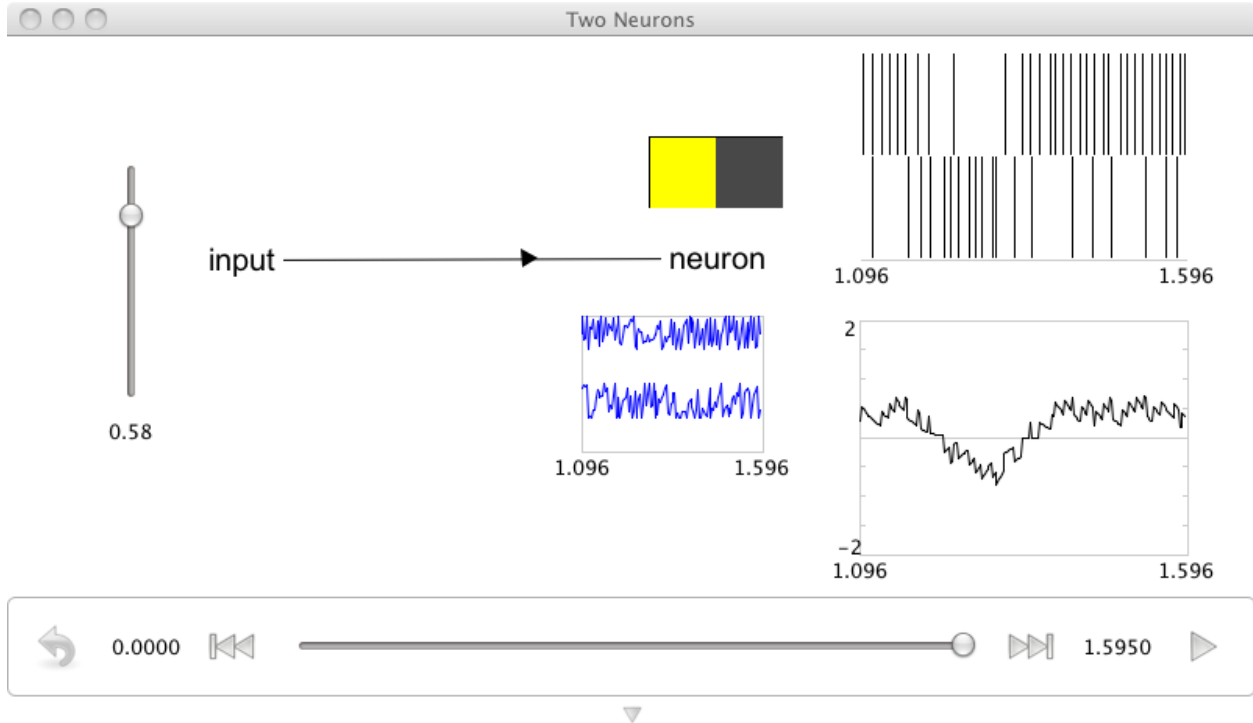
2.1.2 Two Neurons

Purpose: This demo shows how to construct and manipulate a complementary pair of neurons.

Comments: These are leaky integrate-and-fire (LIF) neurons. The neuron tuning properties have been selected so there is one 'on' and one 'off' neuron.

Usage: Grab the slider control and move it up and down to see the effects of increasing or decreasing input. One neuron will increase for positive input, and the other will decrease. This can be thought of as the simplest population to give a reasonable representation of a scalar value.

Output: See the screen capture below



Code:

```
import nef

net=nef.Network('Two Neurons') #Create the network object
input=net.make_input('input', [-0.45]) #Create a controllable input function
                                         #with a starting value of -.45
neuron=net.make('neuron', 2, 1, max_rate=(100, 100), intercept=(-0.5, -0.5),
               encoders=[[1], [-1]], noise=3) #Make 2 neurons, 1 dimension, max firing
                                              #rates evenly distributed between 100 and
                                              #100, x-intercepts evenly distributed
                                              #between -.5 and -.5, encoders of 1 and
                                              #-1 and noise at every step with a
                                              #variance of 3
net.connect(input, neuron) #Connect the input to the neurons
net.add_to_nengo()
```

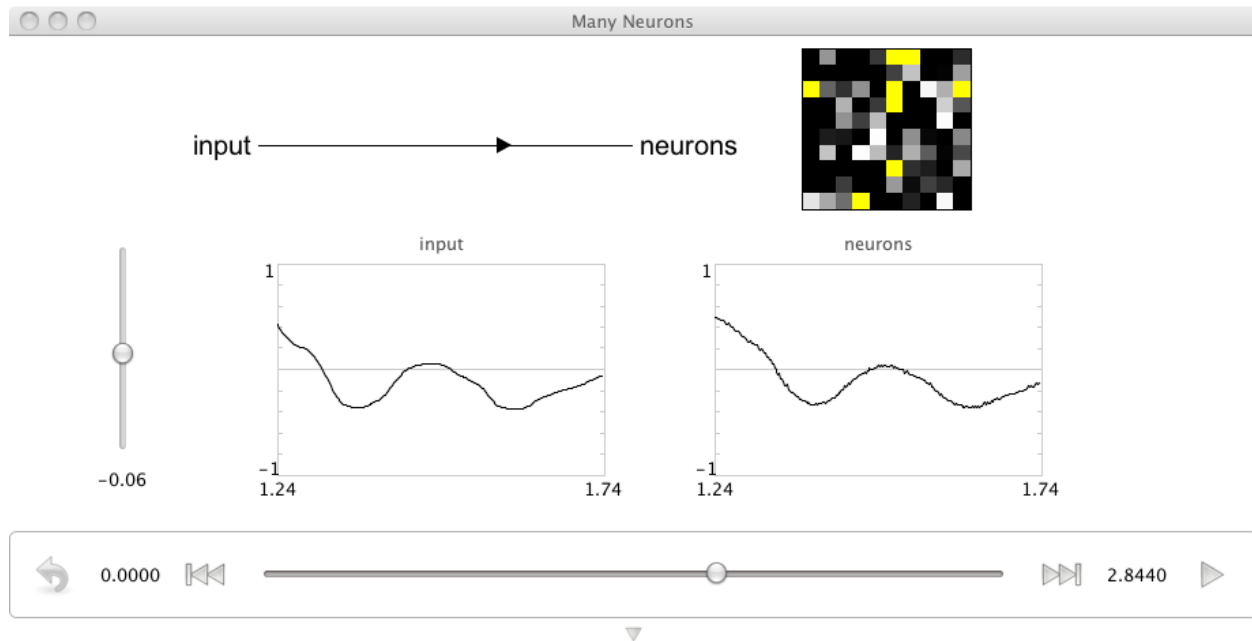
2.1.3 Population of Neurons

Purpose: This demo shows how to construct and manipulate a population of neurons.

Comments: These are 100 leaky integrate-and-fire (LIF) neurons. The neuron tuning properties have been randomly selected.

Usage: Grab the slider control and move it up and down to see the effects of increasing or decreasing input. As a population, these neurons do a good job of representing a single scalar value. This can be seen by the fact that the input graph and neurons graphs match well.

Output: See the screen capture below



Code:

```
import nef

net=nef.Network('Many Neurons') #Create the network object
input=net.make_input('input', [-0.45]) #Create a controllable input function
                                         #with a starting value of -.45
neuron=net.make('neurons', 100, 1, noise=1,
               quick=True) #Make a population with 100 neurons, 1 dimensions, and noise
                           #variance of 1 (added at every step)
net.connect(input, neuron) #Connect the input to the population
net.add_to_nengo()
```

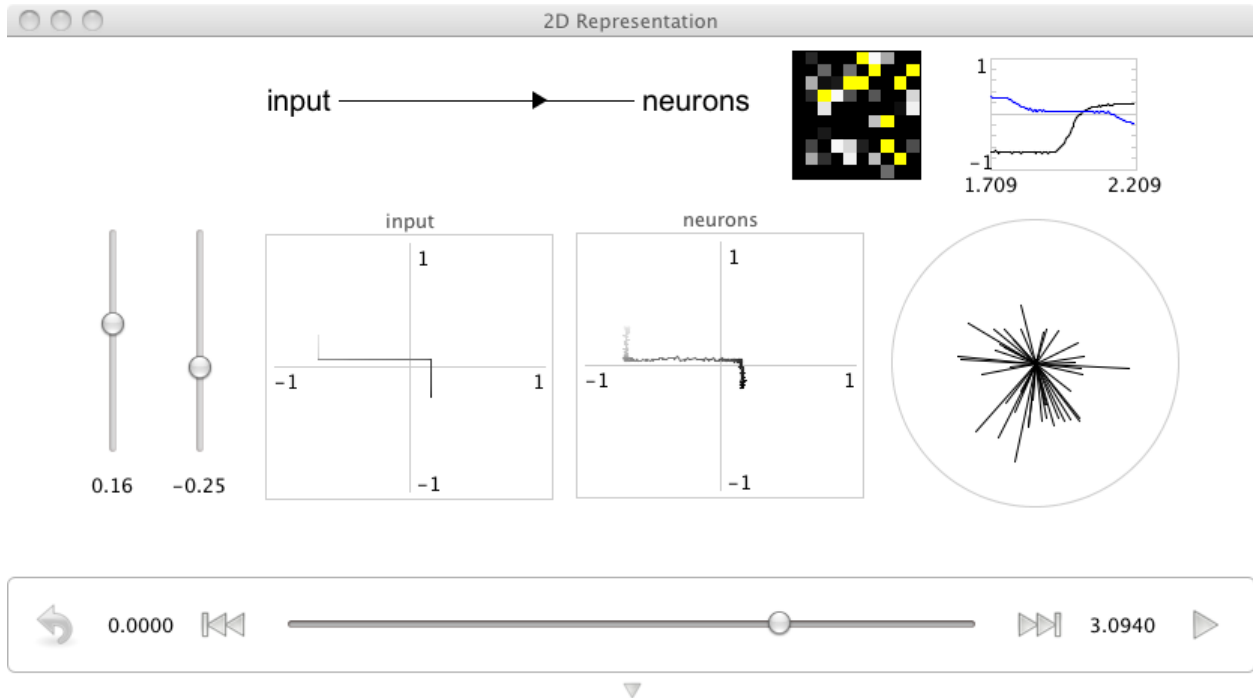
2.1.4 2D Representation

Purpose: This demo shows how to construct and manipulate a population of 2D neurons.

Comments: These are 100 leaky integrate-and-fire (LIF) neurons. The neuron tuning properties have been randomly selected to encode a 2D space (i.e. each neuron has an encoder randomly selected from the unit circle).

Usage: Grab the slider controls and move then up and down to see the effects of shifting the input throughout the 2D space. As a population, these neurons do a good job of representing a 2D vector value. This can be seen by the fact that the input graph and neurons graphs match well.

Output: See the screen capture below. The 'circle' plot is showing the preferred direction vector of each neuron multiplied by its firing rate. This kind of plot was made famous by Georgopoulos et al.



Code:

```
import nef

net=nef.Network('2D Representation') #Create the network object
input=net.make_input('input',[0,0]) #Create a controllable input function
                                     #with a starting value of 0 and 0 in
                                     #the two dimensions

neuron=net.make('neurons',100,2,quick=True) #Make a population with 100
                                              #neurons, 2 dimensions

net.connect(input,neuron) #Connect the input object to the neuron object,
                          #with an identity matrix by default

net.add_to_nengo()
```

2.2 Simple Transformations

2.2.1 Communication Channel

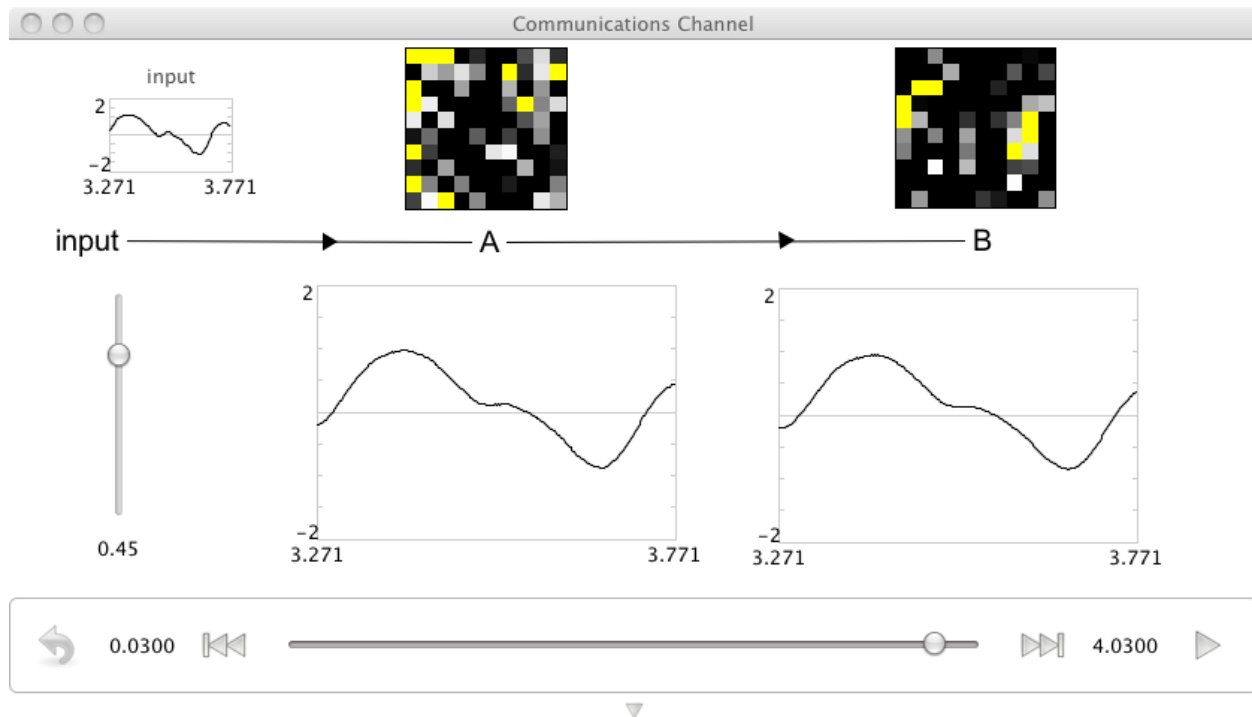
Purpose: This demo shows how to construct a simple communication channel.

Comments: A communication channel attempts to take the information from one population and put it in the next one. The ‘transformation’ is thus the identity $f(x) = x$.

Notably, this is the simplest ‘neural circuit’ in the demos. This is because the connection from the first to second population is only connection weights that are applied to postsynaptic currents (PSCs) generated by incoming spikes.

Usage: Grab the slider control and move it up and down to see the effects of increasing or decreasing input. Both populations should reflect the input, but note that the second population only gets input from the first population through synaptic connections.

Output: See the screen capture below



Code:

```
import nef

net=nef.Network('Communications Channel') #Create the network object
input=net.make_input('input',[0.5]) #Create a controllable input function
                                     #with a starting value of 0.5
A=net.make('A',100,1,quick=True) #Make a population with 100 neurons,
                                  #1 dimensions
B=net.make('B',100,1,quick=True,
           storage_code='B') #Make a population with 100 neurons, 1 dimensions
                             #(storage codes work with 'quick' to load already made
                             #populations if they exist
net.connect(input,A) #Connect all the relevant objects
net.connect(A,B)
net.add_to_nengo()
```

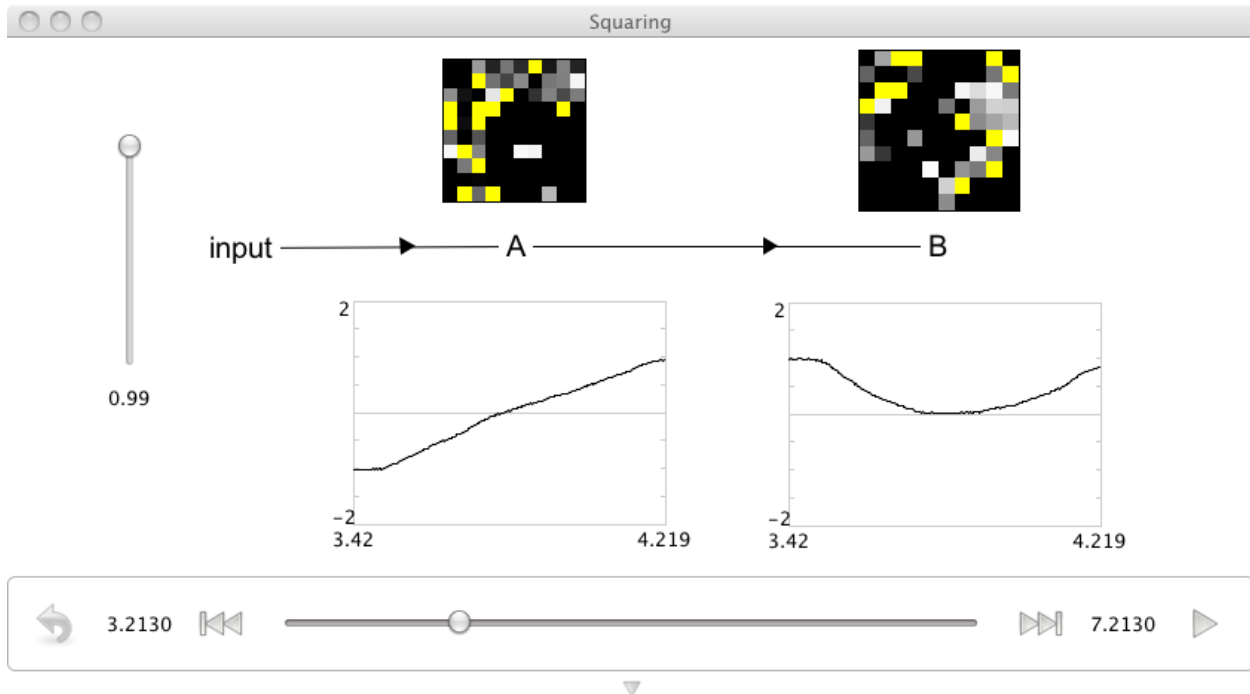
2.2.2 Squaring the Input

Purpose: This demo shows how to construct a network that squares the value encoded in a first population in the output of a second population.

Comments: This is a simple nonlinear function being decoded in the connection weights between the cells. Previous demos are linear decodings.

Usage: Grab the slider control and move it up and down to see the effects of increasing or decreasing input. Notice that the output value does not go negative even for negative inputs. Dragging the input slowly from -1 to 1 will approximately trace a quadratic curve in the output.

Output: See the screen capture below



Code:

```
import nef

net=nef.Network('Squaring') #Create the network object
input=net.make_input('input',[0]) #Create a controllable input function
                                     #with a starting value of 0
A=net.make('A',100,1,quick=True) #Make a population with 100 neurons,
                                     #1 dimensions
B=net.make('B',100,1,quick=True,storage_code='B') #Make a population with
                                                     #100 neurons, 1 dimensions

net.connect(input,A) #Connect the input to A
net.connect(A,B,func=lambda x: x[0]*x[0]) #Connect A and B with the
                                           #defined function approximated
                                           #in that connection

net.add_to_nengo()
```

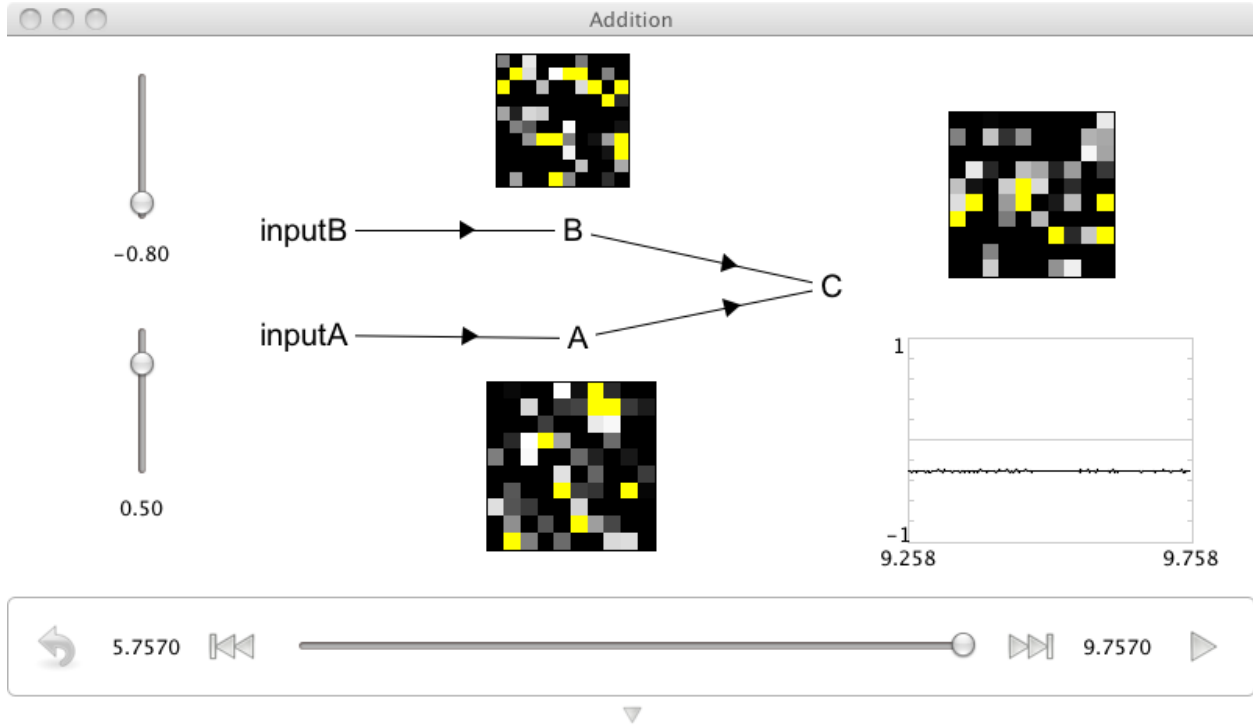
2.2.3 Addition

Purpose: This demo shows how to construct a network that adds two inputs.

Comments: Essentially, this is two communication channels into the same population. Addition is thus somewhat ‘free’, since the incoming currents from different synaptic connections interact linearly (though two inputs don’t have to combine in this way: see the combining demo).

Usage: Grab the slider controls and move them up and down to see the effects of increasing or decreasing input. The C population represents the sum of A and B representations. Note that the ‘addition’ is a description of neural firing in the decoded space. Neurons don’t just add all the incoming spikes (the NEF has determined appropriate connection weights to make the result in C interpretable (i.e., decodable) as the sum of A and B).

Output: See the screen capture below



Code:

```
import nef

net=nef.Network('Addition') #Create the network object
inputA=net.make_input('inputA',[0]) #Create a controllable input function
                                     #with a starting value of 0
inputB=net.make_input('inputB',[0]) #Create another controllable input
                                     #function with a starting value of 0
A=net.make('A',100,1,quick=True) #Make a population with 100 neurons,
                                  #1 dimensions
B=net.make('B',100,1,quick=True,
           storage_code='B') #Make a population with 100 neurons, 1 dimensions
                             #(storage codes work with 'quick' to load already made
                             #populations if they exist)
C=net.make('C',100,1,quick=True,storage_code='C') #Make a population with
                                                    #100 neurons, 1 dimensions

net.connect(inputA,A) #Connect all the relevant objects
net.connect(inputB,B)
net.connect(A,C)
net.connect(B,C)
net.add_to_nengo()
```

2.2.4 Combining 1D Representations into a 2D Representation

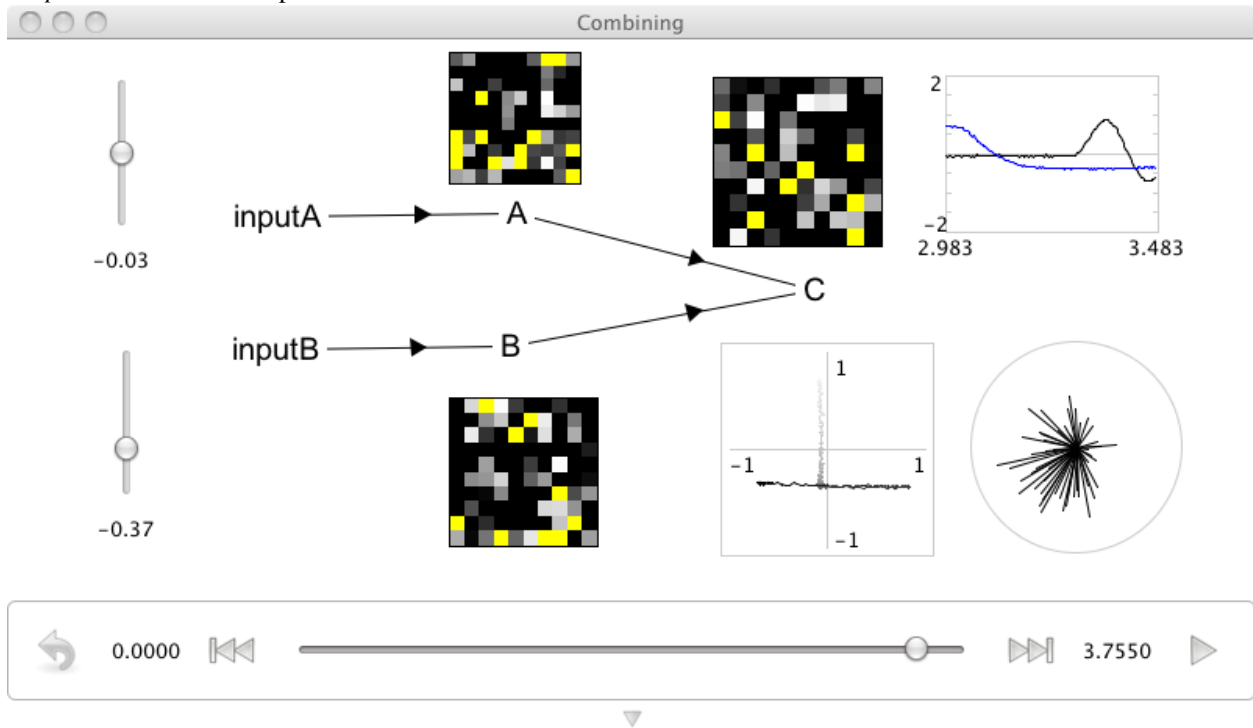
Purpose: This demo shows how to construct a network that combines two 1D inputs into a 2D representation.

Comments: This can be thought of as two communication channels projecting to a third population, but instead of combining the input (as in addition), the receiving population represents them as being independent.

Usage: Grab the slider controls and move them up and down to see the effects of increasing or decreasing input. Notice that the output population represents both dimensions of the input independently, as can be seen by the fact that

each input slider only changes one dimension in the output.

Output: See the screen capture below



Code:

```
import nef

net=nef.Network('Combining') #Create the network object
inputA=net.make_input('inputA',[0]) #Create a controllable input function
#with a starting value of 0
inputB=net.make_input('inputB',[0]) #Create another controllable input
#function with a starting value of 0
A=net.make('A',100,1,quick=True) #Make a population with 100 neurons,
#1 dimensions
B=net.make('B',100,1,quick=True,
    storage_code='B') #Make a population with 100 neurons, 1 dimensions
#(storage codes work with 'quick' to load already made
#populations if they exist)
C=net.make('C',100,2,quick=True,
    radius=1.5) #Make a population with 100 neurons, 2 dimensions, and set a
#larger radius (so 1,1 input still fits within the circle of
#that radius)
net.connect(inputA,A) #Connect all the relevant objects (default connection
#is identity)
net.connect(inputB,B)
net.connect(A,C,transform=[1,0]) #Connect with the given 1x2D mapping matrix
net.connect(B,C,transform=[0,1])
net.add_to_nengo()
```

2.2.5 Performing Multiplication

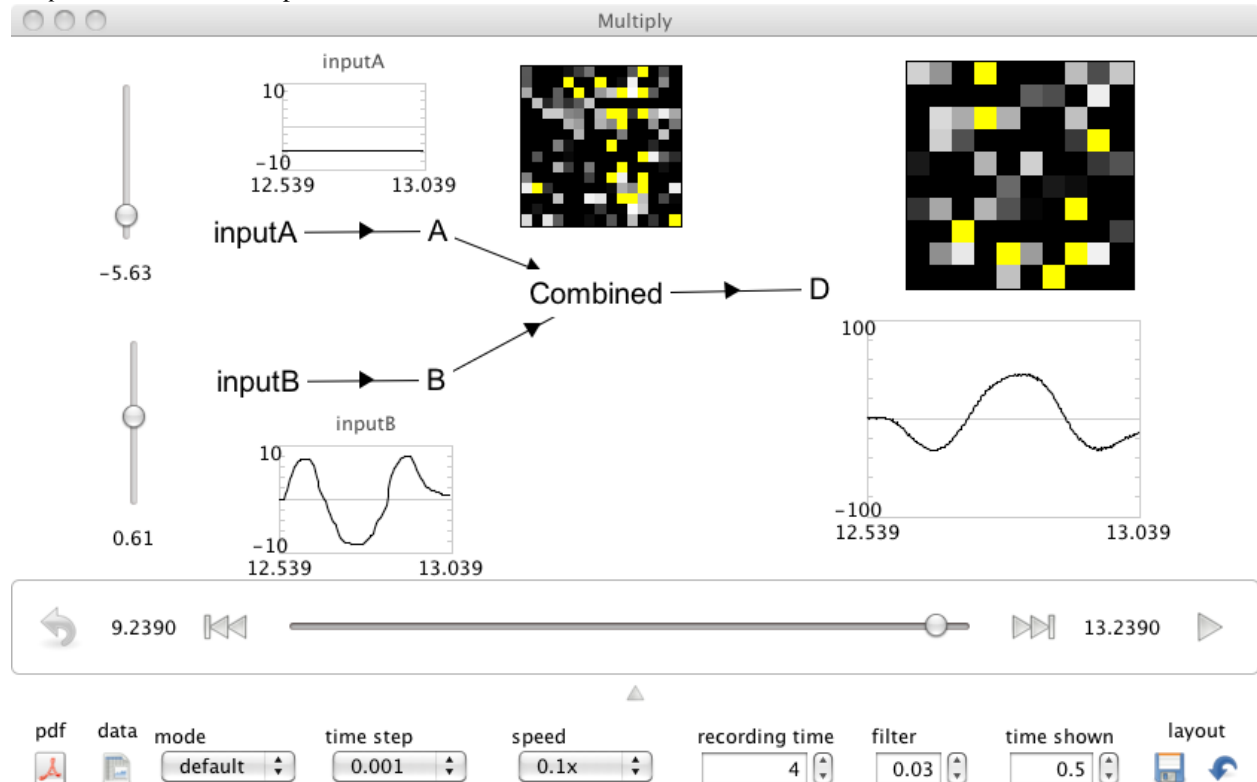
Purpose: This demo shows how to construct a network that multiplies two inputs.

Comments: This can be thought of as a combination of the combining demo and the squaring demo. Essentially, we project both inputs independently into a 2D space, and then decode a nonlinear transformation of that space (the product of the first and second vector elements).

Multiplication is extremely powerful. Following the simple usage instructions below suggests how you can exploit it to do gating of information into a population, as well as radically change the response of a neuron to its input (i.e. completely invert its ‘tuning’ to one input dimension by manipulating the other).

Usage: Grab the slider controls and move them up and down to see the effects of increasing or decreasing input. The output is the product of the inputs. To see this quickly, leave one at zero and move the other. Or, set one input at a negative value and watch the output slope go down as you move the other input up.

Output: See the screen capture below



Code:

```
import nef

net=nef.Network('Multiplication') #Create the network object
inputA=net.make_input('inputA',[8]) #Create a controllable input function
#with a starting value of 8
inputB=net.make_input('inputB',[5]) #Create a controllable input function
#with a starting value of 5
A=net.make('A',100,1,radius=10,quick=True) #Make a population with 100 neurons,
#1 dimensions, a radius of 10
#(default is 1)
B=net.make('B',100,1,radius=10,quick=True,
storage_code='B') #Make a population with 100 neurons, 1 dimensions, a
#radius of 10 (default is 1), storage_code works with
#quick to reuse an appropriate population if created
#before
C=net.make('Combined',225,2,radius=15,
```

```

    quick=True) #Make a population with 225 neurons, 2 dimensions, and set a
                #larger radius (so 10,10 input still fits within the circle
                #of that radius)
D=net.make('D',100,1,radius=100,quick=True,
           storage_code='D') #Make a population with 100 neurons, 1 dimensions, a
                             #radius of 10 (default is 1)
net.connect(inputA,A) #Connect all the relevant objects
net.connect(inputB,B)
net.connect(A,C,transform=[1,0]) #Connect with the given 1x2D mapping matrix
net.connect(B,C,transform=[0,1])
def product(x):
    return x[0]*x[1]
net.connect(C,D,func=product) #Create the output connection mapping the
                              #1D function 'product'
net.add_to_nengo()

```

2.2.6 Circular Convolution

Purpose: This demo shows how to exploit the hrr library to do binding of vector representations.

Comments: The binding operator we use is circular convolution. This example is in a 10-dimensional space.

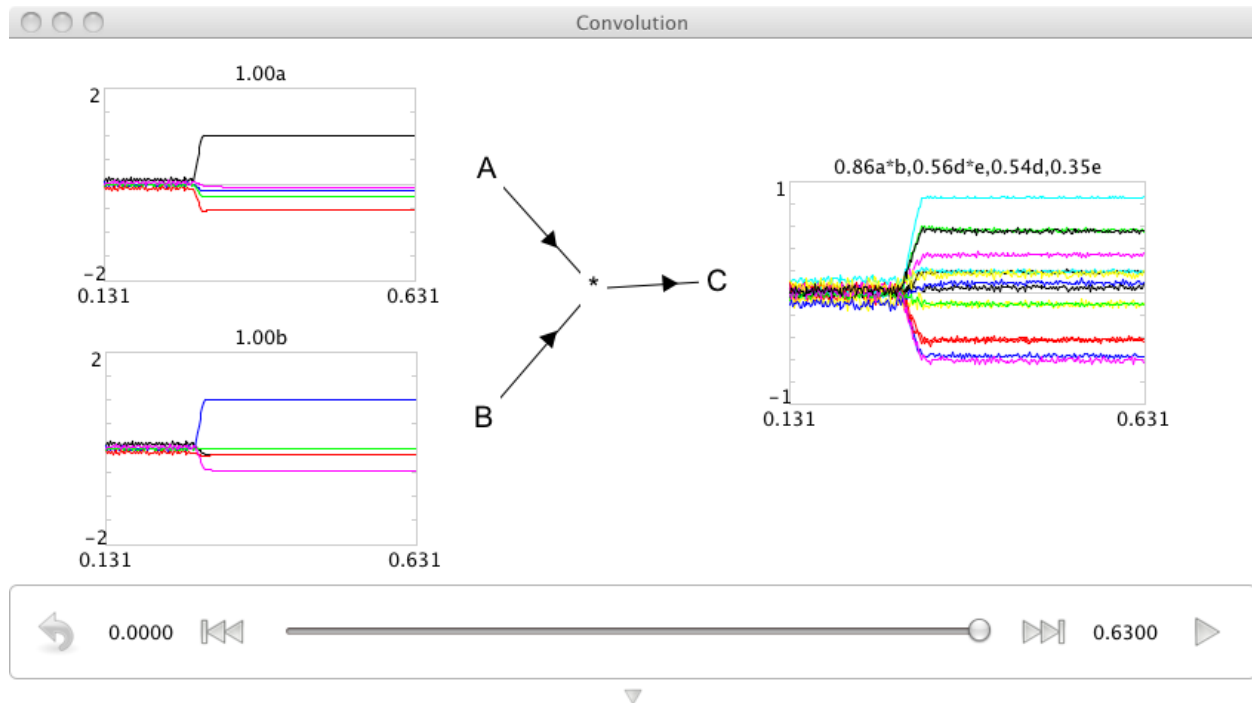
This (or any similar) binding operator (see work on vector symbolic architectures (VSAs)) is important for cognitive models. This is because such operators lets you construct structured representations in a high-dimensional vector space.

Usage: The best way to change the input is to right-click the ‘semantic pointer’ graphs and choose ‘set value’ to set the value to one of the elements in the vocabulary (defined as a, b, c, d, or e in the code) by typing it in. Each element in the vocabulary is a randomly chosen vector. Set a different value for the two input graphs.

The ‘C’ population represents the output of a neurally-computed circular convolution (i.e., binding) of the ‘A’ and ‘B’ input vectors. The label above each semantic pointer graph displays the name of the vocabulary vectors that are most similar to the vector represented by that neural ensemble. The number preceding the vector name is the value of the normalized dot product between the two vectors (i.e., the similarity of the vectors).

In this simulation, the ‘most similar’ vocabulary vector for the ‘C’ ensemble is ‘a*b’. The ‘a*b’ vector is the analytically-calculated circular convolution of the ‘a’ and ‘b’ vocabulary vectors. This result is expected, of course. Also of note is that the similarity of the ‘a’ and ‘b’ vectors alone is significantly lower. Both of the original input vectors should have a low degree of similarity to the result of the binding operation. The ‘show pairs’ option controls whether bound pairs of vocabulary vectors are included in the graph.

Output: See the screen capture below



Code:

```
import nef
import nef.convolution
import hrr

D=10

vocab=hrr.Vocabulary(D, include_pairs=True)
vocab.parse('a+b+c+d+e')

net=nef.Network('Convolution') #Create the network object
A=net.make('A', 300, D, quick=True) #Make a population of 300 neurons and
                                     #10 dimensions
B=net.make('B', 300, D, quick=True)
C=net.make('C', 300, D, quick=True)
conv=nef.convolution.make_convolution(net, '*', A, B, C, 100,
                                       quick=True) #Call the code to construct a convolution network using
                                                    #the created populations and 100 neurons per dimension

net.add_to_nengo()
```

2.3 Dynamics

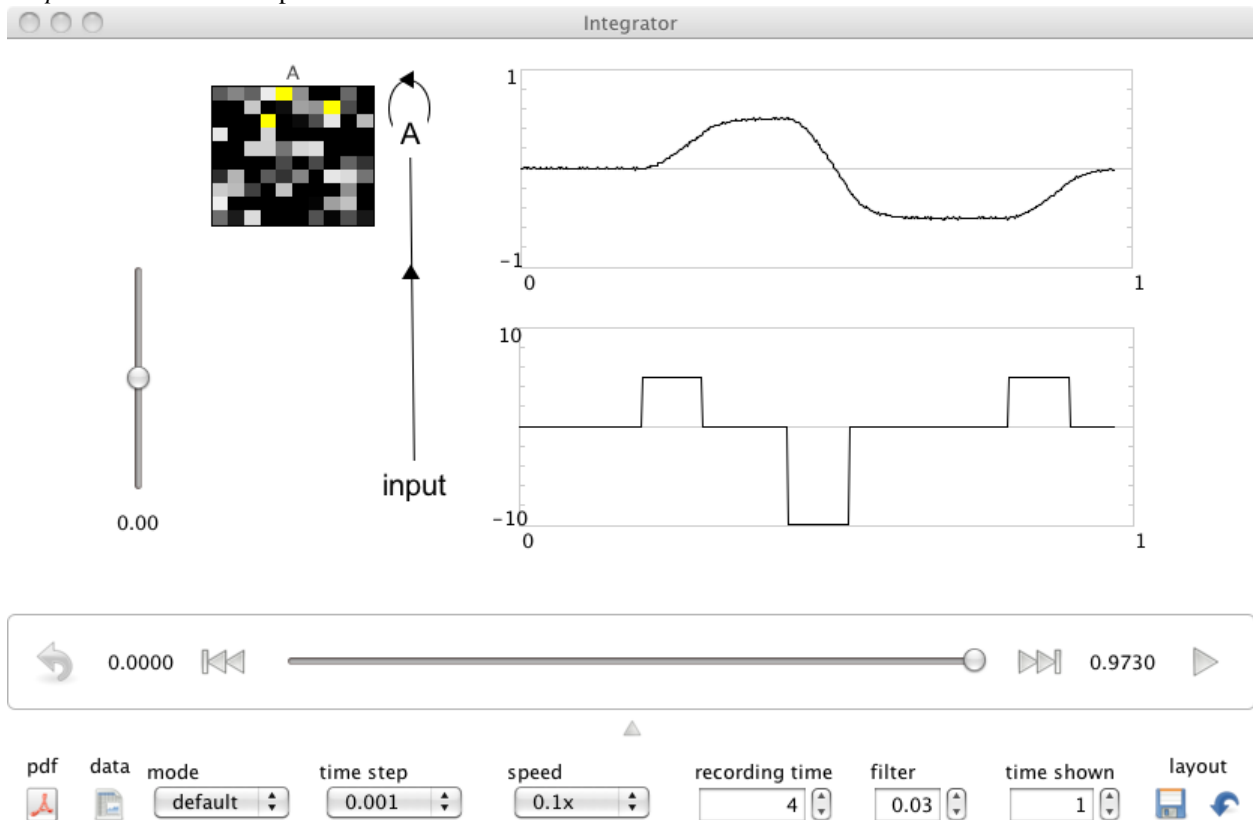
2.3.1 A Simple Integrator

Purpose: This demo implements a one-dimensional neural integrator.

Comments: This is the first example of a recurrent network in the demos. It shows how neurons can be used to implement stable dynamics. Such dynamics are important for memory, noise cleanup, statistical inference, and many other dynamic transformations.

Usage: When you run this demo, it will automatically put in some step functions on the input, so you can see that the output is integrating (i.e. summing over time) the input. You can also input your own values. Note that since the integrator constantly sums its input, it will saturate quickly if you leave the input non-zero. This reminds us that neurons have a finite range of representation. Such saturation effects can be exploited to perform useful computations (e.g. soft normalization).

Output: See the screen capture below



Code:

```
import nef

net=nef.Network('Integrator') #Create the network object
input=net.make_input('input',[0]) #Create a controllable input function
                                     #with a starting value of 0
input.functions=[ca.nengo.math.impl.PiecewiseConstantFunction(
    [0.2,0.3,0.44,0.54,0.8,0.9],
    [0,5,0,-10,0,5,0])] #Change the input function (that was 0) to this
                           #piecewise step function
A=net.make('A',100,1,quick=True) #Make a population with 100 neurons,
                                   #1 dimensions
net.connect(input,A,weight=0.1,pstc=0.1) #Connect the input to the integrator,
                                           #scaling the input by .1; postsynaptic
                                           #time constant is 10ms
net.connect(A,A,pstc=0.1) #Connect the population to itself with the
                           #default weight of 1
net.add_to_nengo()
```

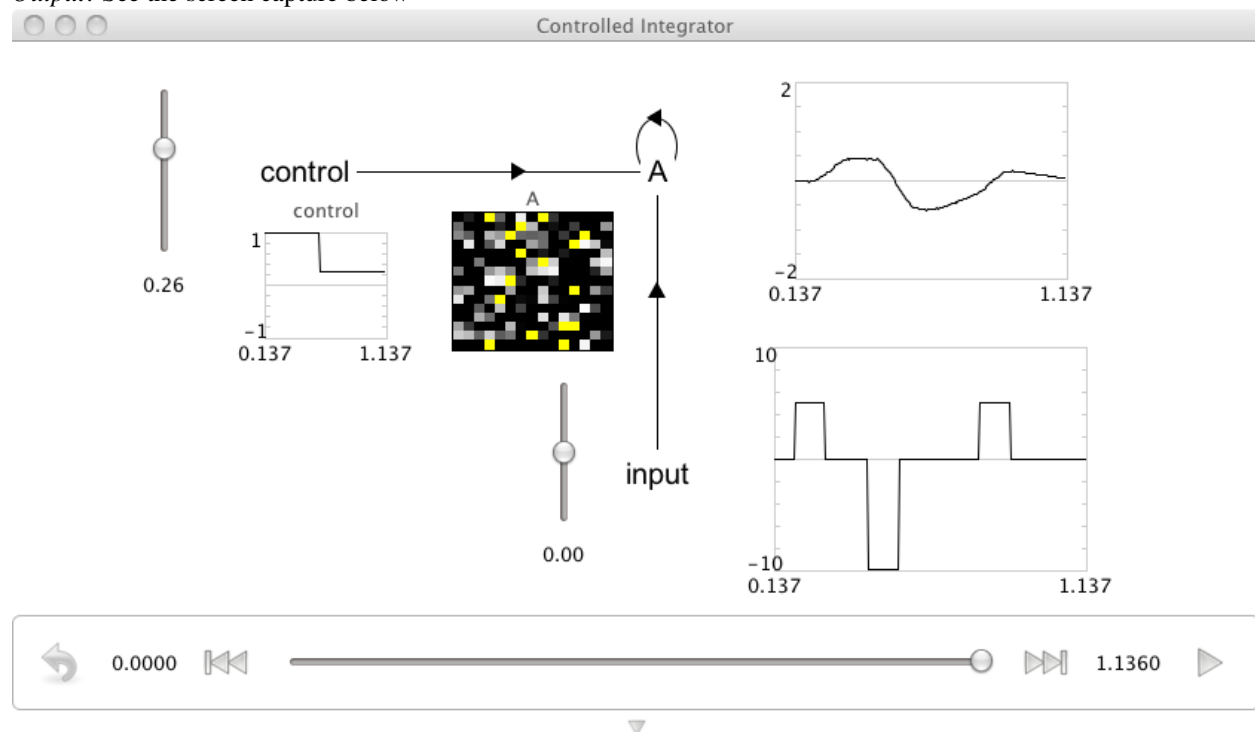
2.3.2 Controlled Integrator

Purpose: This demo implements a controlled one-dimensional neural integrator.

Comments: This is the first example of a controlled dynamic network in the demos. This is the same as the integrator circuit, but we now have introduced a population that explicitly effects how well the circuit ‘remembers’ its past state. The ‘control’ input can be used to make the integrator change from a pure communication channel (0) to an integrator (1) with various low-pass filtering occurring in between.

Usage: When you run this demo, it will automatically put in some step functions on the input, so you can see that the output is integrating (i.e. summing over time) the input. You can also input your own values. It is quite sensitive like the integrator. But, if you reduce the ‘control’ input below 1, it will not continuously add its input, but slowly allow that input to leak away. It’s interesting to rerun the simulation from the start, but decreasing the ‘control’ input before the automatic input starts to see the effects of ‘leaky’ integration.

Output: See the screen capture below



Code:

```
import nef

net=nef.Network('Controlled Integrator') #Create the network object
input=net.make_input('input',[0]) #Create a controllable input function with
                                   #a starting value of 0
input.functions=[ca.nengo.math.impl.PiecewiseConstantFunction(
    [0.2,0.3,0.44,0.54,0.8,0.9],
    [0,5,0,-10,0,5,0])] #Change the input function (that was 0) to this
                        #piecewise step function
control=net.make_input('control',[1]) #Create a controllable input function
                                       #with a starting value of 1
A=net.make('A',225,2,radius=1.5,
    quick=True) #Make a population with 225 neurons, 2 dimensions, and a
                #larger radius to accommodate large simultaneous inputs
net.connect(input,A,transform=[0.1,0],pstc=0.1) #Connect all the relevant
```



```

#objects with the relevant 1x2
#mappings, postsynaptic time
#constant is 10ms

net.connect(control,A,transform=[0,1],pstc=0.1)
def feedback(x):
    return x[0]*x[1]
net.connect(A,A,transform=[1,0],func=feedback,pstc=0.1) #Create the recurrent
                                                         #connection mapping the
                                                         #1D function 'feedback'
                                                         #into the 2D population
                                                         #using the 1x2 transform

net.add_to_nengo()

```

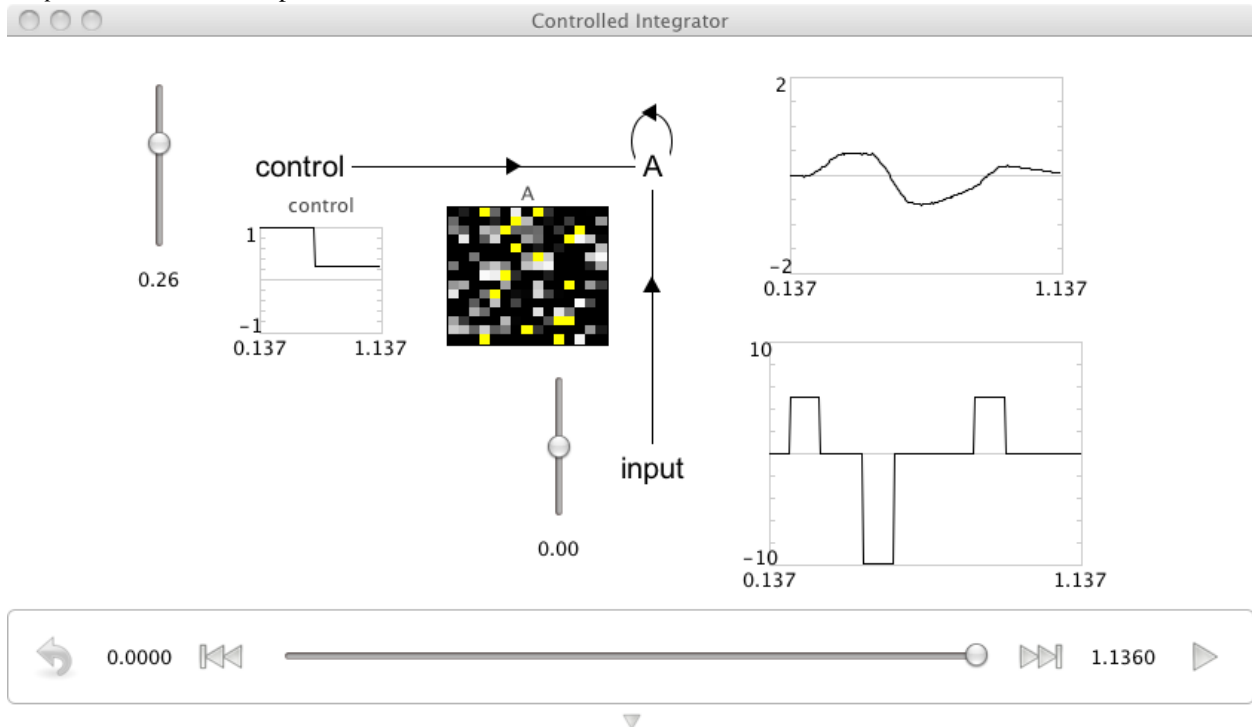
2.3.3 Controlled Integrator 2

Purpose: This demo implements a controlled one-dimensional neural integrator.

Comments: This is functionally the same as the other controlled integrator. However, the control signal is zero for integration, less than one for low-pass filtering, and greater than 1 for saturation. This behavior maps more directly to the differential equation used to describe an integrator (i.e. $\dot{x}(t) = Ax(t) + Bu(t)$). The control in this circuit is A in that equation. This is also the controlled integrator described in the book “How to build a brain.”

Usage: When you run this demo, it will automatically put in some step functions on the input, so you can see that the output is integrating (i.e. summing over time) the input. You can also input your own values. It is quite sensitive like the integrator. But, if you reduce the ‘control’ input below 0, it will not continuously add its input, but slowly allow that input to leak away. It’s interesting to rerun the simulation from the start, but decreasing the ‘control’ input before the automatic input starts to see the effects of ‘leaky’ integration.

Output: See the screen capture below



Code:

```
import nef

#This implements the controlled integrator described in the
#book "How to build a brain"
net=nef.Network('Controlled Integrator 2') #Create the network object
input=net.make_input('input',[0]) #Create a controllable input function with
                                     #a starting value of 0
input.functions=[ca.nengo.math.impl.PiecewiseConstantFunction(
    [0.2,0.3,0.44,0.54,0.8,0.9],
    [0,5,0,-10,0,5,0])] #Change the input function (that was 0) to this
                           #piecewise step function
control=net.make_input('control',[0]) #Create a controllable input function
                                       #with a starting value of 0

A=net.make('A',225,2,radius=1.5,
    quick=True) #Make a population with 225 neurons, 2 dimensions, and a
                #larger radius to accommodate large simultaneous inputs
net.connect(input,A,transform=[0.1,0],pstc=0.1) #Connect all the relevant
                                                #objects with the relevant 1x2
                                                #mappings, postsynaptic time
                                                #constant is 10ms

net.connect(control,A,transform=[0,1],pstc=0.1)
def feedback(x):
    return x[0]*x[1]+x[0] #Different than the other controlled integrator
net.connect(A,A,transform=[1,0],func=feedback,pstc=0.1) #Create the recurrent
                                                         #connection mapping the
                                                         #1D function 'feedback'
                                                         #into the 2D population
                                                         #using the 1x2 transform

net.add_to_nengo()
```

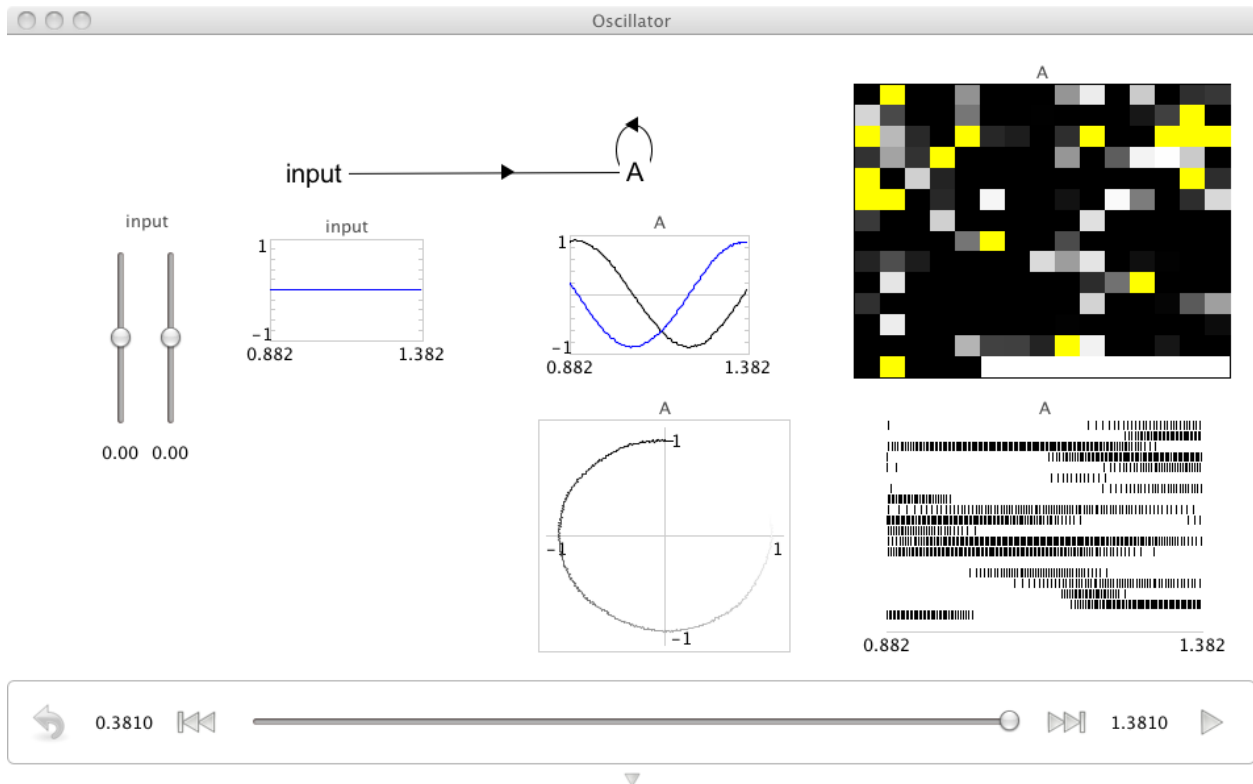
2.3.4 A Simple Harmonic Oscillator

Purpose: This demo implements a simple harmonic oscillator in a 2D neural population.

Comments: This is more visually interesting on its own than the integrator, but the principle is the same. Here, instead of having the recurrent input just integrate (i.e. feeding the full input value back to the population), we have two dimensions which interact. In Nengo there is a 'Linear System' template which can also be used to quickly construct a harmonic oscillator (or any other linear system).

Usage: When you run this demo, it will sit at zero while there is no input, and then the input will cause it to begin oscillating. It will continue to oscillate without further input. You can put inputs in to see the effects. It is very difficult to have it stop oscillating. You can imagine this would be easy to do by either introducing control as in the controlled integrator demo, or by changing the tuning curves of the neurons (hint: so none represent values between -.3 and 3, say).

Output: See the screen capture below. You will get a sine and cosine in the 2D output.



Code:

```
import nef

net=nef.Network('Oscillator') #Create the network object
input=net.make_input('input',[1,0],
    zero_after_time=0.1) #Create a controllable input function with a
    #starting value of 1 and zero, then make it go
    #to zero after .1s
A=net.make('A',200,2) #Make a population with 200 neurons, 2 dimensions
net.connect(input,A)
net.connect(A,A,[[1,1],[-1,1]],pstc=0.1) #Recurrently connect the population
    #with the connection matrix for a
    #simple harmonic oscillator mapped
    #to neurons with the NEF

net.add_to_nengo()
```

2.4 Basal Ganglia Based Simulations

2.4.1 Basal Ganglia

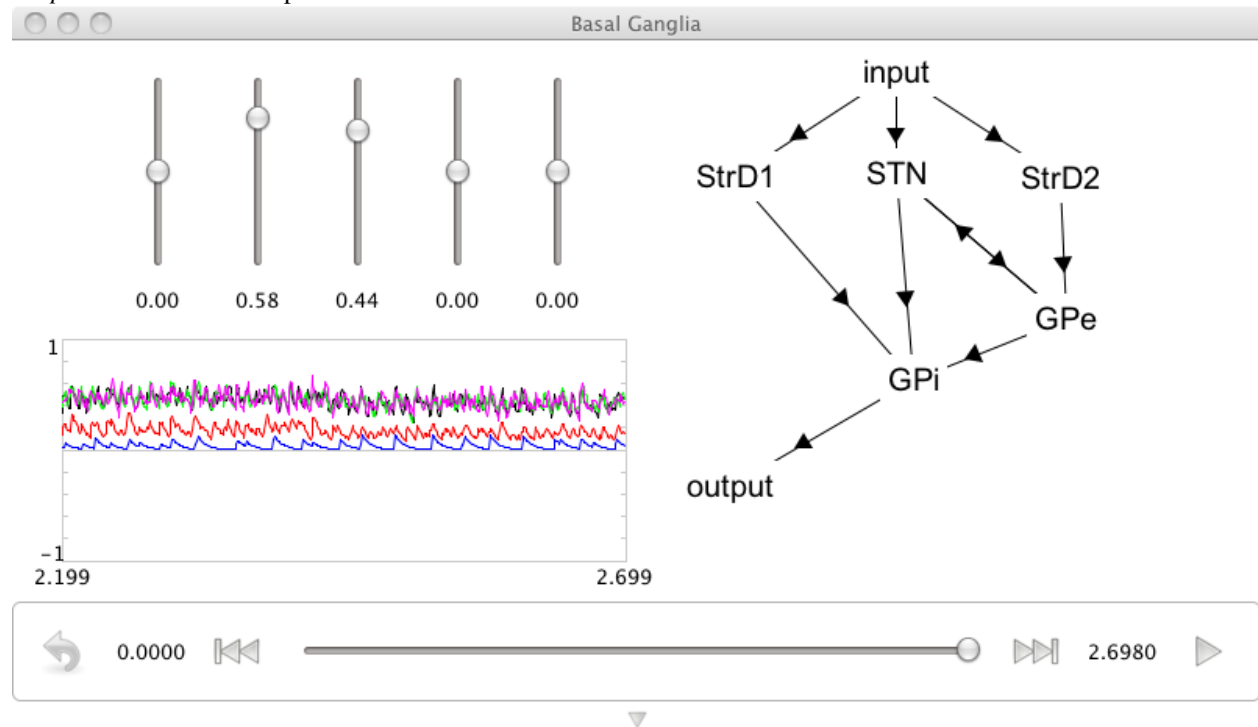
Purpose: This demo introduces the basal ganglia model that the SPA exploits to do action selection.

Comments: This is just the basal ganglia, not hooked up to anything. It demonstrates that this model operates as expected, i.e. supressing the output corresponding to the input with the highest input value.

This is an extension to a spiking, dynamic model of the Redgrave et al. work. It is more fully described in several CNRG lab publications. It exploits the 'nps' class from Nengo.

Usage: After running the demo, play with the 5 input sliders. The highest slider should always be selected in the output. When they are close, interesting things happen. You may even be able to tell that things are selected more quickly for larger differences in input values.

Output: See the screen capture below.



Code:

```
import nef
import nps

D=5
net=nef.Network('Basal Ganglia') #Create the network object
input=net.make_input('input',[0]*D) #Create a controllable input function
                                     #with a starting value of 0 for each of D
                                     #dimensions
output=net.make('output',1,D,mode='direct',
               quick=True) #Make a population with 100 neurons, 5 dimensions, and set
                           #the simulation mode to direct
nps.basalganglia.make_basal_ganglia(net,input,output,D,same_neurons=False,
                                   N=50) #Make a basal ganglia model with 50 neurons per action
net.add_to_nengo()
```

2.4.2 Cycling Through a Sequence

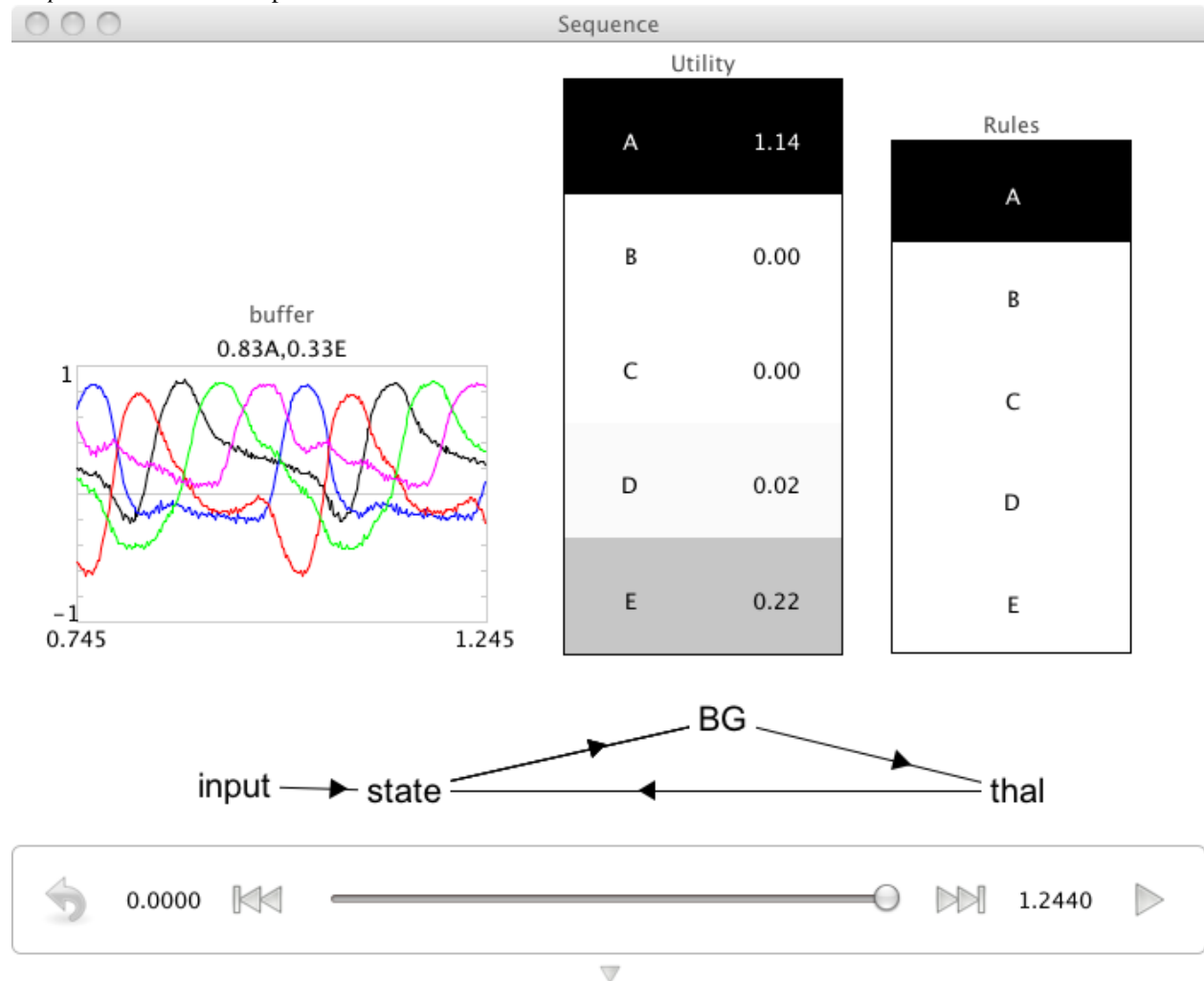
Purpose: This demo uses the basal ganglia model to cycle through a 5 element sequence.

Comments: This basal ganglia is now hooked up to a memory. This allows it to update that memory based on its current input/action mappings. The mappings are defined in the code such that A->B, B->C, etc. until E->A completing a loop. This uses the 'spa' module from Nengo.

The 'utility' graph shows the utility of each rule going into the basal ganglia. The 'rule' graph shows which one has been selected and is driving thalamus.

Usage: When you run the network, it will go through the sequence forever. It's interesting to note the distance between the 'peaks' of the selected items. It's about 40ms for this simple action. We like to make a big deal of this.

Output: See the screen capture below.



Code:

```
from spa import *

D=16

class Rules: #Define the rules by specifying the start state and the
    #desired next state
    def A(state='A'): #e.g. If in state A
        set(state='B') # then go to state B
    def B(state='B'):
        set(state='C')
    def C(state='C'):
        set(state='D')
    def D(state='D'):
        set(state='E')
    def E(state='E'):
        set(state='A')
```

```
class Sequence(SPA): #Define an SPA model (cortex, basal ganglia, thalamus)
    dimensions=16

    state=Buffer() #Create a working memory (recurrent network) object:
                  #i.e. a Buffer
    BG=BasalGanglia(Rules()) #Create a basal ganglia with the prespecified
                           #set of rules
    thal=Thalamus(BG) # Create a thalamus for that basal ganglia (so it
                     # uses the same rules)

    input=Input(0.1,state='D') #Define an input; set the input to
                              #state D for 100 ms

seq=Sequence()
```

2.4.3 Routed Sequencing

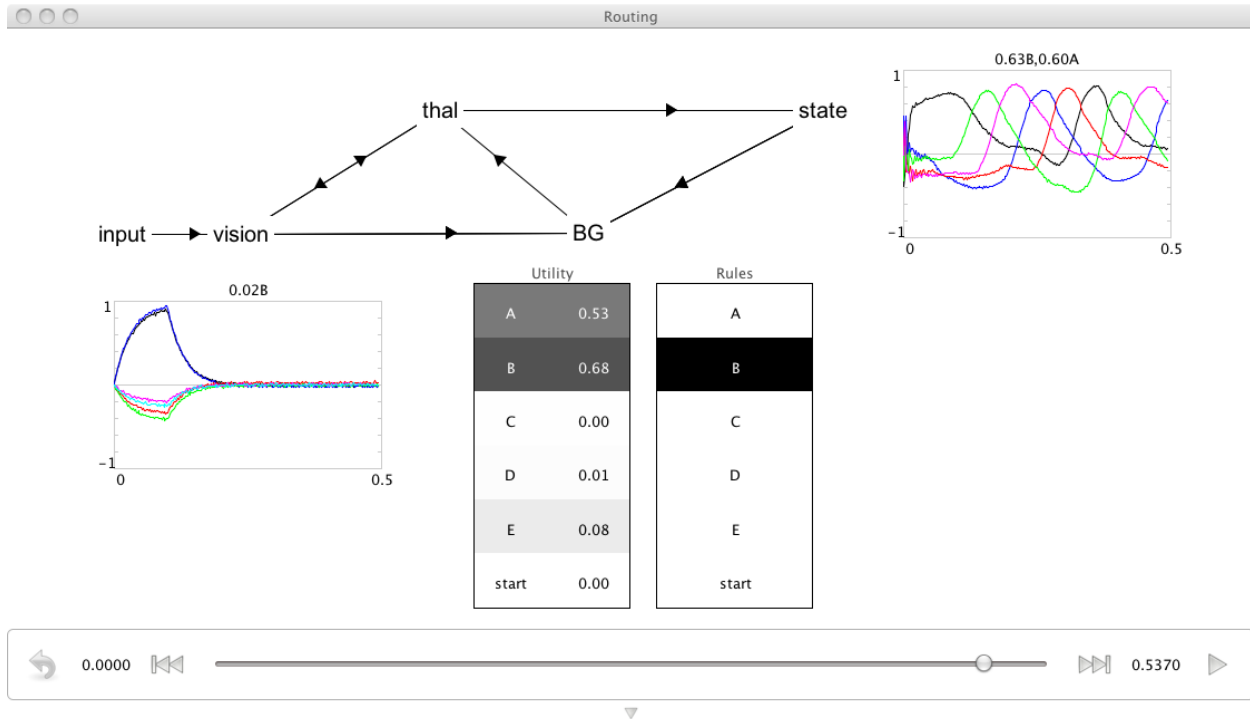
Purpose: This demo uses the basal ganglia model to cycle through a 5 element sequence, where an arbitrary start can be presented to the model.

Comments: This basal ganglia is now hooked up to a memory and includes routing. The addition of routing allows the system to choose between two different actions: whether to go through the sequence, or be driven by the visual input. If the visual input has its value set to $0.8 * \text{START} + D$ (for instance), it will begin cycling through at D->E, etc. The 0.8 scaling helps ensure start is unlikely to accidentally match other SPAs (which can be a problem in low dimensional examples like this one).

The ‘utility’ graph shows the utility of each rule going into the basal ganglia. The ‘rule’ graph shows which one has been selected and is driving thalamus.

Usage: When you run the network, it will go through the sequence forever, starting at D. You can right-click the SPA graph and set the value to anything else (e.g. ‘ $0.8 * \text{START} + B$ ’) and it will start at the new letter and then begin to sequence through. The point is partly that it can ignore the input after its first been shown and doesn’t persevere on the letter as it would without gating.

Output: See the screen capture below.

**Code:**

```

from spa import *

D=16

class Rules: #Define the rules by specifying the start state and the
    #desired next state
    def start(vision='START'):
        set(state=vision)
    def A(state='A'): #e.g. If in state A
        set(state='B') # then go to state B
    def B(state='B'):
        set(state='C')
    def C(state='C'):
        set(state='D')
    def D(state='D'):
        set(state='E')
    def E(state='E'):
        set(state='A')

class Routing(SPA): #Define an SPA model (cortex, basal ganglia, thalamus)
    dimensions=16

    state=Buffer() #Create a working memory (recurrent network)
    #object: i.e. a Buffer
    vision=Buffer(feedback=0) #Create a cortical network object with no
    #recurrence (so no memory properties, just
    #transient states)
    BG=BasalGanglia(Rules) #Create a basal ganglia with the prespecified
    #set of rules

```

```
thal=Thalamus(BG) # Create a thalamus for that basal ganglia (so it
                  # uses the same rules)

input=Input(0.1,vision='0.8*START+D') #Define an input; set the input
                                       #to state 0.8*START+D for 100 ms

model=Routing()
```

2.4.4 A Question Answering Network

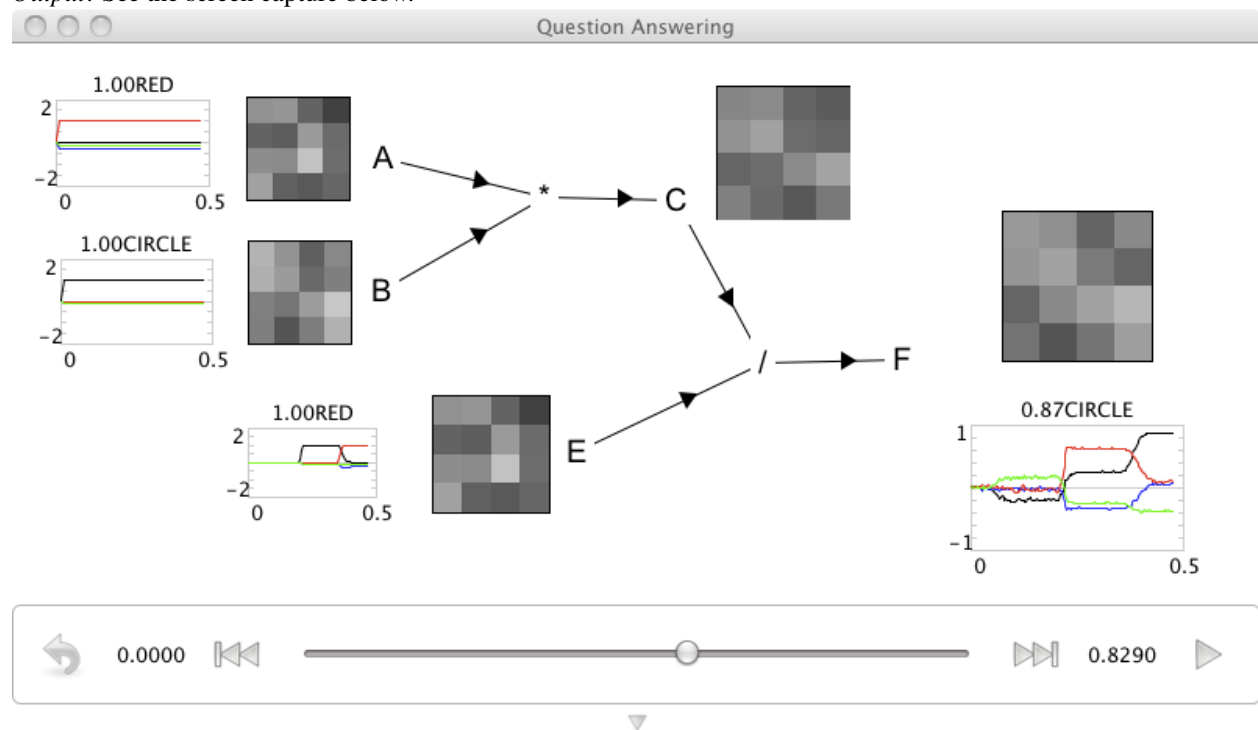
Purpose: This demo shows how to do question answering using binding (i.e. see the convolution demo).

Comments: This example binds the A and B inputs to give C. Then the E input is used to decode the contents of C and the result is shown in F. Essentially showing unbinding gives back what was bound.

Note: The b/w graphs show the decoded vector values, not neural activity. So, the result in F should visually look like the A element if B is being unbound from C.

Usage: When you run the network, it will start by binding 'RED' and 'CIRCLE' and then unbinding 'RED' from that result, and the output will be 'CIRCLE'. Then it does the same kind of thing with BLUE SQUARE. You can set the input values by right-clicking the SPA graphs and setting the value by typing something in. If you type in a vocabulary word that is not there, it will be added to the vocabulary.

Output: See the screen capture below.



Code:

```
D=16
subdim=4
N=100
seed=7

import nef
```



```

import nef.convolution
import hrr
import math
import random

random.seed(seed)

vocab=hrr.Vocabulary(D,max_similarity=0.1)

net=nef.Network('Question Answering') #Create the network object
A=net.make('A',1,D,mode='direct') #Make some pseudo populations (so they
                                   #run well on less powerful machines):
                                   #1 neuron, 16 dimensions, direct mode
B=net.make('B',1,D,mode='direct')
C=net.make_array('C',N,D/subdim,dimensions=subdim,quick=True,radius=1.0/math.sqrt(D),storage_cod
                 #array element and D/subdim elements in the array
                 #each with subdim dimensions, set the radius as
                 #appropriate for multiplying things of this
                 #dimension
E=net.make('E',1,D,mode='direct')
F=net.make('F',1,D,mode='direct')

conv1=nef.convolution.make_convolution(net,'*',A,B,C,N,
                                       quick=True) #Make a convolution network using the construct populations
conv2=nef.convolution.make_convolution(net,'/',C,E,F,N,
                                       invert_second=True,quick=True) #Make a 'correlation' network (by using
                                                                       #convolution, but inverting the second
                                                                       #input)

CIRCLE=vocab.parse('CIRCLE') #Add elements to the vocabulary to use
BLUE=vocab.parse('BLUE')
RED=vocab.parse('RED')
SQUARE=vocab.parse('SQUARE')
ZERO=[0]*D

class Input(nef.SimpleNode): #Make a simple node to generate
                             #interesting input for the network
    def origin_A(self):
        t=(self.t_start)%1.0
        if 0<t<0.5: return RED.v
        if 0.5<t<1: return BLUE.v
        return ZERO
    def origin_B(self):
        t=(self.t_start)%1.0
        if 0.0<t<0.5: return CIRCLE.v
        if 0.5<t<1: return SQUARE.v
        return ZERO
    def origin_E(self):
        t=(self.t_start)%1.0
        if 0.2<t<0.35: return CIRCLE.v
        if 0.35<t<0.5: return RED.v
        if 0.7<t<0.85: return SQUARE.v
        if 0.85<t<1: return BLUE.v
        return ZERO

input=Input('input')

```

```

net.add(input)
net.connect(input.getOrigin('A'),A) #Connect the origins in the simple node
                                   #to the populations they are input to
net.connect(input.getOrigin('B'),B)
net.connect(input.getOrigin('E'),E)

net.add_to_nengo()

```

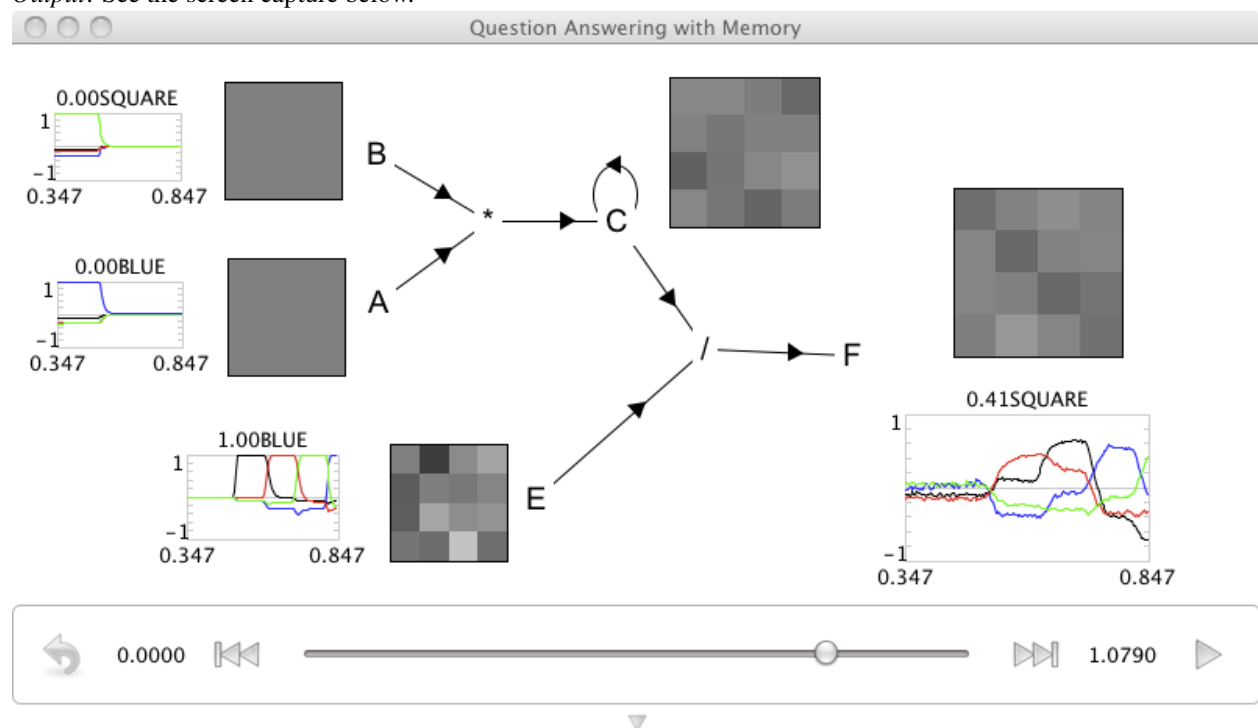
2.4.5 A Question Answering Network with Memory

Purpose: This demo performs question answering based on storing items in a working memory.

Comments: This example is very much like the question answering demo (it would be good to read that). However, it answers questions based on past input, not the immediate input.

Usage: When you run the network, it will start by binding 'RED' and 'CIRCLE' and then binding 'BLUE' and 'SQUARE' so the memory essentially has RED*CIRCLE + BLUE*SQUARE. The input from A and B is turned off, and E is then run through the vocabulary. The appropriately unbound element for each vocabulary word shows up in F as appropriate.

Output: See the screen capture below.



Code:

```

D=16
subdim=4
N=100
seed=7

import nef
import nef.convolution
import hrr

```

```

import math
import random

random.seed(seed)

vocab=hrr.Vocabulary(D,max_similarity=0.1)

net=nef.Network('Question Answering with Memory') #Create the network object
A=net.make('A',1,D,mode='direct') #Make some pseudo populations (so they run
                                   #well on less powerful machines): 1 neuron,
                                   #16 dimensions, direct mode
B=net.make('B',1,D,mode='direct')
C=net.make_array('C',N,D/subdim,dimensions=subdim,quick=True,radius=1.0/math.sqrt(D),storage_coef=0.5)
                                   #array element and D/subdim elements in the array
                                   #each with subdim dimensions, set the radius as
                                   #appropriate for multiplying things of this
                                   #dimension
E=net.make('E',1,D,mode='direct')
F=net.make('F',1,D,mode='direct')

conv1=nef.convolution.make_convolution(net,'*',A,B,C,N,
                                       quick=True) #Make a convolution network using the construct populations
conv2=nef.convolution.make_convolution(net,'/',C,E,F,N,
                                       invert_second=True,quick=True) #Make a 'correlation' network (by using
                                       #convolution, but inverting the second input)

net.connect(C,C,pstc=0.4) # Recurrently connect C so it acts as a memory

CIRCLE=vocab.parse('CIRCLE') #Create a vocabulary
BLUE=vocab.parse('BLUE')
RED=vocab.parse('RED')
SQUARE=vocab.parse('SQUARE')
ZERO=[0]*D

class Input(nef.SimpleNode): #Make a simple node to generate interesting
                             #input for the network

    def origin_A(self):
        t=(self.t_start)
        if 0<t<0.25: return RED.v
        if 0.25<t<0.5: return BLUE.v
        return ZERO
    def origin_B(self):
        t=(self.t_start)
        if 0.0<t<0.25: return CIRCLE.v
        if 0.25<t<0.5: return SQUARE.v
        return ZERO
    def origin_E(self):
        t=self.t_start
        if t<0.5: return ZERO
        t=t%0.5
        if 0.0<t<0.1: return CIRCLE.v
        if 0.1<t<0.2: return RED.v
        if 0.2<t<0.3: return SQUARE.v
        if 0.3<t<0.4: return BLUE.v
        return ZERO

```

```

input=Input('input')
net.add(input)
net.connect(input.getOrigin('A'),A) #Connect the origins in the simple node
                                   #to the populations they are input to

net.connect(input.getOrigin('B'),B)
net.connect(input.getOrigin('E'),E)

net.add_to_nengo()

```

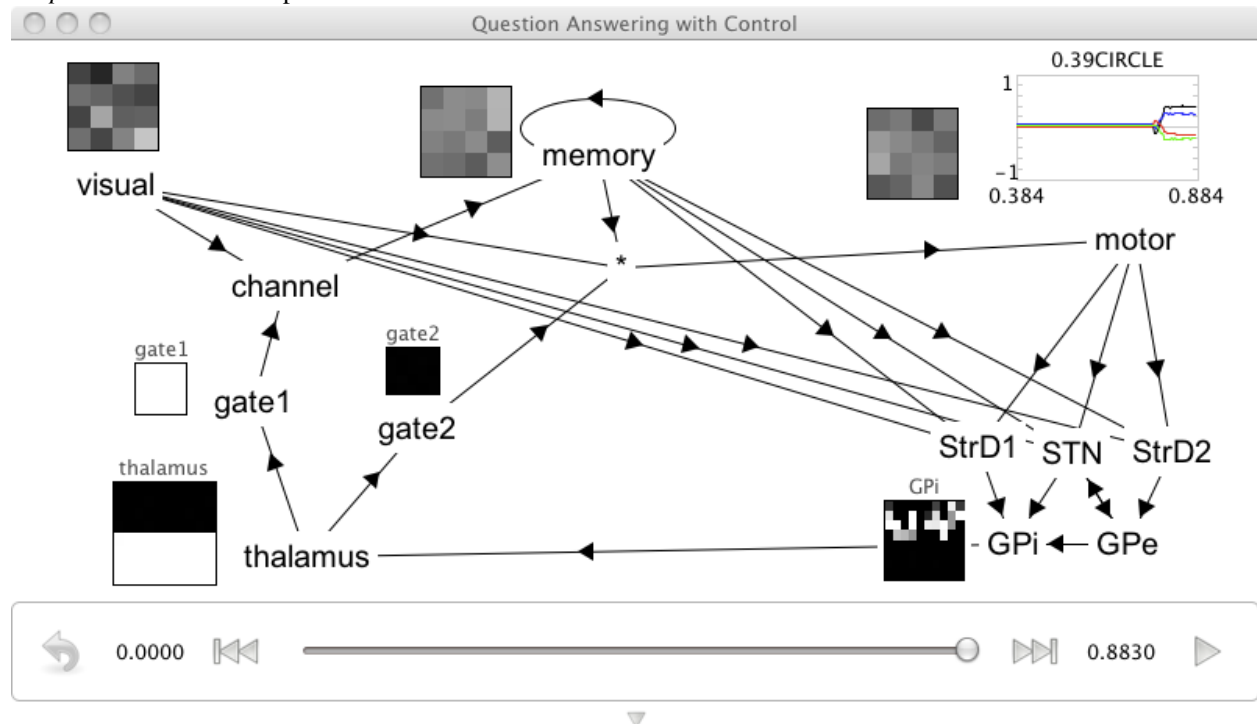
2.4.6 A Controlled Question Answering Network

Purpose: This demo performs question answering based on storing items in a working memory, while under control of a basal ganglia.

Comments: This example is very much like the question answering with memory demo (it would be good to read that). However, both the information to bind and the questions to answer come in through the same visual channel. The basal ganglia decides what to do with the information in the visual channel based on its content (i.e. whether it is a statement or a question).

Usage: When you run the network, it will start by binding 'RED' and 'CIRCLE' and then binding 'BLUE' and 'SQUARE' so the memory essentially has RED*CIRCLE + BLUE*SQUARE. It does this because it is told that RED*CIRCLE is a STATEMENT (i.e. RED*CIRCLE+STATEMENT in the code) as is BLUE*SQUARE. Then it is presented with something like QUESTION+RED (i.e., 'What is red?'). The basal ganglia then reroutes that input to be compared to what is in working memory and the result shows up in the motor channel.

Output: See the screen capture below.



Code:

```

D=16
subdim=4

```

```

N=100
seed=3

import nef
import nps
import nef.convolution
import hrr
import math
import random

random.seed(seed)

net=nef.Network('Question Answering with Control') #Create the network object

for i in range(50): #This is here to create a vocabulary with sufficient
                    #lack of similarity in the items so we can work in such
                    #a low dimensional space; for high dimensional spaces
                    #this can be skipped
    vocab=hrr.Vocabulary(D,max_similarity=0.05)
    vocab.parse('CIRCLE+BLUE+RED+SQUARE')
    a=vocab.parse('(RED*CIRCLE+BLUE*SQUARE)*~(RED)').compare(
        vocab.parse('CIRCLE'))
    b=vocab.parse('(RED*CIRCLE+BLUE*SQUARE)*~(SQUARE)').compare(
        vocab.parse('BLUE'))
    if min(a,b)>0.65: break

class Input(nef.SimpleNode): #Make a simple node to generate interesting
                             #input for the network
    def __init__(self,name):
        self.zero=[0]*D
        nef.SimpleNode.__init__(self,name)
        self.v1=vocab.parse('STATEMENT+RED*CIRCLE').v
        self.v2=vocab.parse('STATEMENT+BLUE*SQUARE').v
        self.v3=vocab.parse('QUESTION+RED').v
        self.v4=vocab.parse('QUESTION+SQUARE').v
    def origin_x(self):
        t=self.t_start
        if t<0.5:
            if 0.1<self.t_start<0.3:
                return self.v1
            elif 0.35<self.t_start<0.5:
                return self.v2
            else:
                return self.zero
        else:
            t=(t-0.5)%0.6
            if 0.2<t<0.4:
                return self.v3
            elif 0.4<t<0.6:
                return self.v4
            else:
                return self.zero

inv=Input('inv')
net.add(inv)

prods=nps.ProductionSet() #This is an older way of implementing an SPA
                         #(see SPA routing examples), using the nps

```

```
#code directly
prods.add(dict(visual='STATEMENT'),dict(visual_to_wm=True))
prods.add(dict(visual='QUESTION'),dict(wm_deconv_visual_to_motor=True))

model=nps.NPS(net,prods,D,direct_convolution=False,direct_buffer=['visual'],
    neurons_buffer=N/subdim,subdimensions=subdim)
model.add_buffer_feedback(wm=1,pstc=0.4)

net.connect(inv.getOrigin('x'),'buffer_visual')

#Rename objects for display purposes
net.network.getNode('prod').name='thalamus'
net.network.getNode('buffer_visual').name='visual'
net.network.getNode('buffer_wm').name='memory'
net.network.getNode('buffer_motor').name='motor'
net.network.getNode('channel_visual_to_wm').name='channel'
net.network.getNode('wm_deconv_visual_to_motor').name='*'
net.network.getNode('gate_visual_wm').name='gate1'
net.network.getNode('gate_wm_visual_motor').name='gate2'

net.add_to_nengo()
```

2.5 Learning

2.5.1 Learning a Communication Channel

Purpose: This is the first demo showing learning in Nengo. It learns the same circuit constructed in the Communication Channel demo.

Comments: The particular connection that is learned is the one between the ‘pre’ and ‘post’ populations. This particular learning rule is a kind of modulated Hebb-like learning (see Bekolay, 2011 for details).

Note: The red and blue graph is a plot of the connection weights, which you can watch change as learning occurs (you may need to zoom in with the scroll wheel; the learning a square demo has a good example). Typically, the largest changes occur at the beginning of a simulation. Red indicates negative weights and blue positive weights.

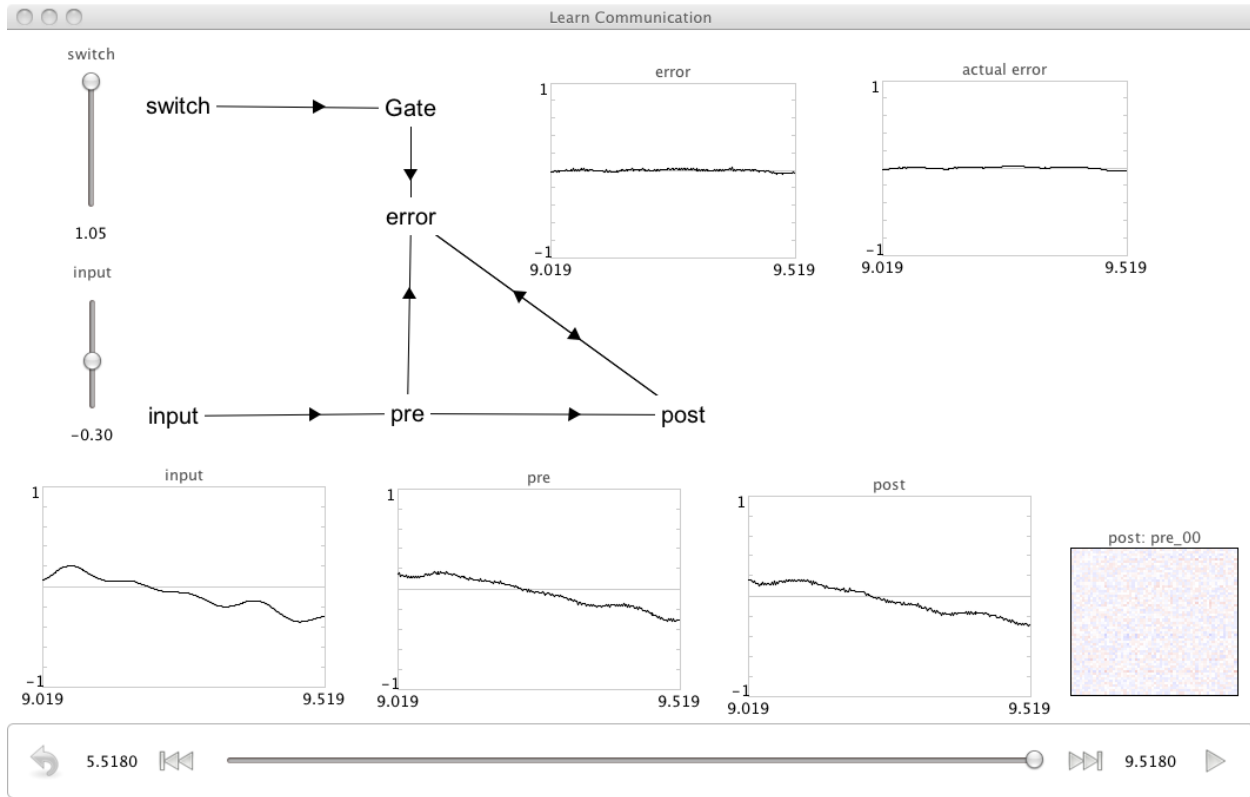
Usage: When you run the network, it automatically has a random white noise input injected into it. So the input slider moves up and down randomly. However, learning is turned off, so there is little correlation between the representation of the pre and post populations.

Turn learning on: To allow the learning rule to work, you need to move the ‘switch’ to +1. Because the learning rule is modulated by an error signal, if the error is zero, the weights won’t change. Once learning is on, the post will begin to track the pre.

Monitor the error: When the switch is 0 at the beginning of the simulation, there is no ‘error’, though there is an ‘actual error’. The difference here is that ‘error’ is calculated by a neural population, and used by the learning rule, while ‘actual error’ is computed mathematically and is just for information.

Repeat the experiment: After a few simulated seconds, the post and pre will match well. You can hit the ‘reset’ button (bottom left) and the weights will be reset to their original random values, and the switch will go to zero. For a different random starting point, you need to re-run the script.

Output: See the screen capture below.

**Code:**

```

N=60
D=1

import nef
import nef.templates.learned_termination as learning
import nef.templates.gate as gating
import random

from ca.nengo.math.impl import FourierFunction
from ca.nengo.model.impl import FunctionInput
from ca.nengo.model import Units

random.seed(27)

net=nef.Network('Learn Communication') #Create the network object

# Create input and output populations.
A=net.make('pre',N,D) #Make a population with 60 neurons, 1 dimensions
B=net.make('post',N,D) #Make a population with 60 neurons, 1 dimensions

# Create a random function input.
input=FunctionInput('input',[FourierFunction(
    .1, 10,.5, 12)],
    Units.UNK) #Create a white noise input function .1 base freq, max
               #freq 10 rad/s, and RMS of .5; 12 is a seed
net.add(input) #Add the input node to the network
net.connect(input,A)

```

```
# Create a modulated connection between the 'pre' and 'post' ensembles.
learning.make(net,errName='error', N_err=100, preName='pre', postName='post',
              rate=5e-7) #Make an error population with 100 neurons, and a learning
                        #rate of 5e-7

# Set the modulatory signal.
net.connect('pre', 'error')
net.connect('post', 'error', weight=-1)

# Add a gate to turn learning on and off.
net.make_input('switch',[0]) #Create a controllable input function
                             #with a starting value of 0
gating.make(net,name='Gate', gated='error', neurons=40,
            pstc=0.01) #Make a gate population with 100 neurons, and a postsynaptic
                     #time constant of 10ms
net.connect('switch', 'Gate')

# Add another non-gated error population running in direct mode.
actual = net.make('actual error', 1, 1,
                  mode='direct') #Make a population with 1 neurons, 1 dimensions, and
                                #run in direct mode
net.connect(A,actual)
net.connect(B,actual,weight=-1)

net.add_to_nengo()
```

2.5.2 Learning Multiplication

Purpose: This demo shows learning a familiar nonlinear function, multiplication.

Comments: The set up here is very similar to the other learning demos. The main difference is that this demo learns a nonlinear projection from a 2D to a 1D space (i.e. multiplication).

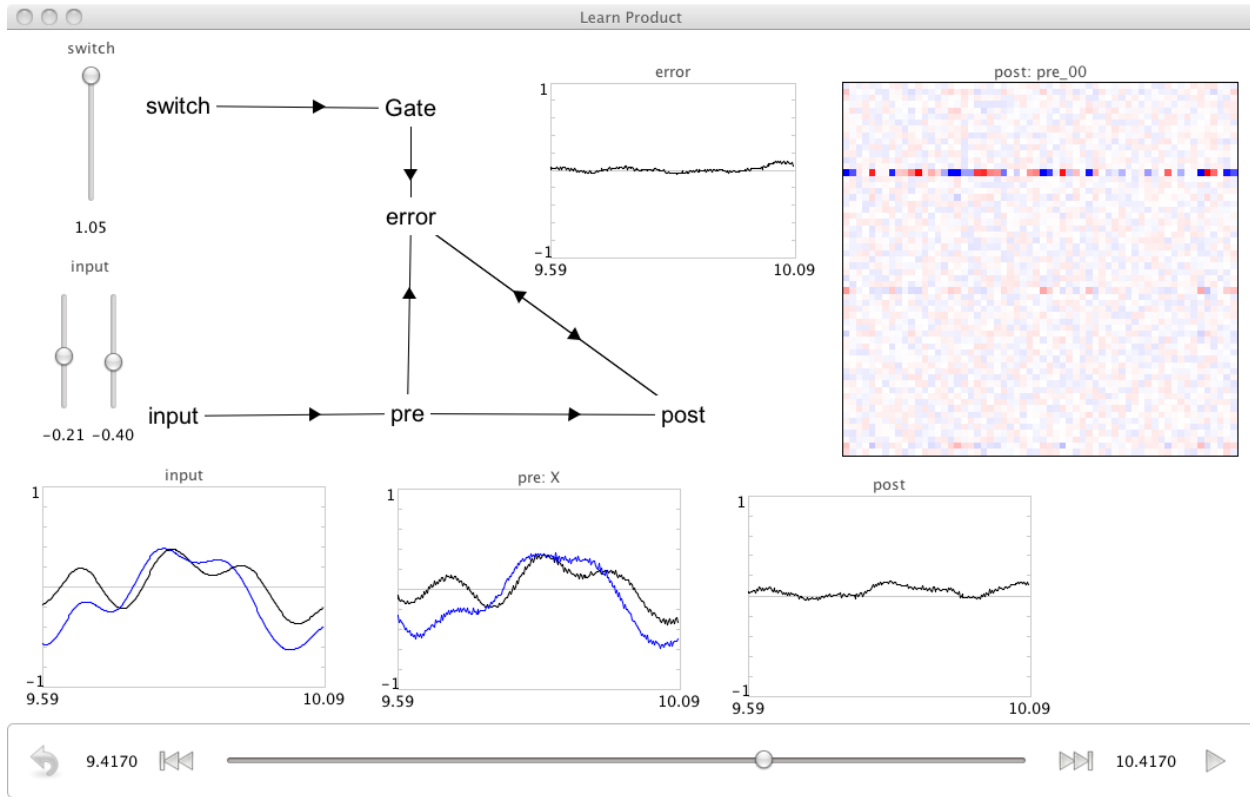
Usage: When you run the network, it automatically has a random white noise input injected into it in both dimensions.

Turn learning on: To allow the learning rule to work, you need to move the 'switch' to +1.

Monitor the error: When the simulation starts and learning is on, the error is high. After about 10s it will do a reasonable job of computing the product, and the error should be quite small.

Is it working? To see if the right function is being computed, compare the 'pre' and 'post' population value graphs. You should note that if either dimension in the input is small, the output will be small. Only when both dimensions have larger absolute values does the output go away from zero (see the screen capture below).

Output: See the screen capture below.

**Code:**

```

N=60
D=2

import nef
import nef.templates.learned_termination as learning
import nef.templates.gate as gating
import random

from ca.nengo.math.impl import FourierFunction
from ca.nengo.model.impl import FunctionInput
from ca.nengo.model import Units

random.seed(37)

net=nef.Network('Learn Product') #Create the network object

# Create input and output populations.
A=net.make('pre',N,D) #Make a population with 60 neurons, 1 dimensions
B=net.make('post',N,1) #Make a population with 60 neurons, 1 dimensions

# Create a random function input.
input=FunctionInput('input',[FourierFunction(
    .1, 8,.4,i,0) for i in range(D)],
    Units.UNK) #Create a white noise input function .1 base freq, max
               #freq 8 rad/s, and RMS of .4; i makes one for each dimension;
               #0 is the seed
net.add(input) #Add the input node to the network
net.connect(input,A)

```

```
# Create a modulated connection between the 'pre' and 'post' ensembles.
learning.make(net,errName='error', N_err=100, preName='pre', postName='post',
              rate=5e-7) #Make an error population with 100 neurons, and a learning
                        #rate of 5e-7

# Set the modulatory signal to compute the desired function
def product(x):
    product=1.0
    for xx in x: product*=xx
    return product

net.connect('pre', 'error', func=product)
net.connect('post', 'error', weight=-1)

# Add a gate to turn learning on and off.
net.make_input('switch',[0]) #Create a controllable input function with
                             #a starting value of 0 and 0 in the two
                             #dimensions
gating.make(net,name='Gate', gated='error', neurons=40,
            pstc=0.01) #Make a gate population with 40 neurons, and a postsynaptic
                     #time constant of 10ms
net.connect('switch', 'Gate')

net.add_to_nengo()
```

2.5.3 Learning to Compute the Square of a Vector

Purpose: This demo shows learning a nonlinear function of a vector.

Comments: The set up here is very similar to the Learning a Communication Channel demo. The main difference is that this demo works in a 2D vector space (instead of a scalar), and that it is learning to compute a nonlinear function (the element-wise square) of its input.

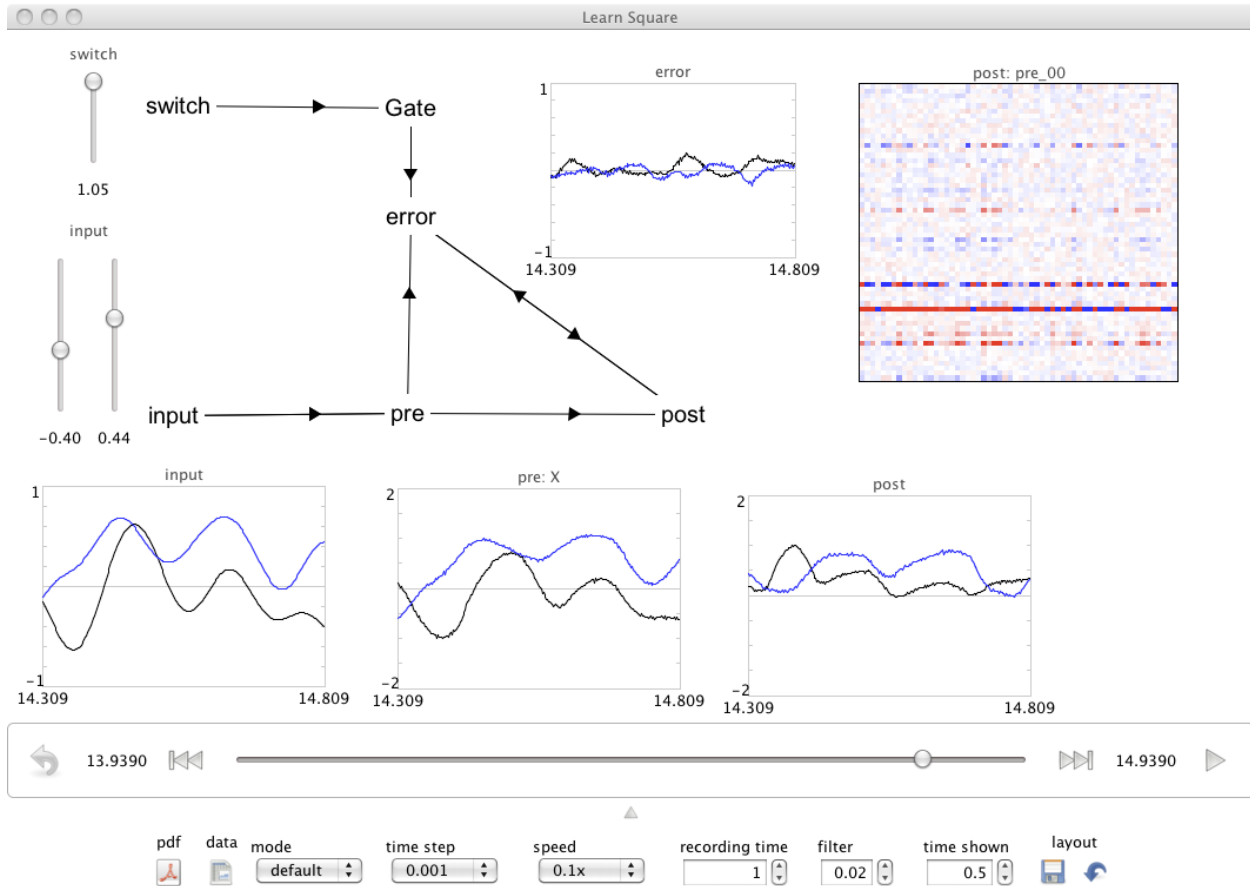
Usage: When you run the network, it automatically has a random white noise input injected into it in both dimensions.

Turn learning on: To allow the learning rule to work, you need to move the 'switch' to +1.

Monitor the error: When the simulation starts and learning is on, the error is high. The average error slowly begins to decrease as the simulation continues. After 15s or so of simulation, it will do a reasonable job of computing the square, and the error in both dimensions should be quite small.

Is it working? To see if the right function is being computed, compare the 'pre' and 'post' population value graphs. You should note that 'post' looks kind of like an absolute value of 'pre', the 'post' will be a bit squashed. You can also check that both graphs of either dimension should hit zero at about the same time.

Output: See the screen capture below.

**Code:**

```

N=60
D=2

import nef
import nef.templates.learned_termination as learning
import nef.templates.gate as gating
import random

from ca.nengo.math.impl import FourierFunction
from ca.nengo.model.impl import FunctionInput
from ca.nengo.model import Units

random.seed(27)

net=nef.Network('Learn Square') #Create the network object

# Create input and output populations.
A=net.make('pre',N,D) #Make a population with 60 neurons, 1 dimensions
B=net.make('post',N,D) #Make a population with 60 neurons, 1 dimensions

# Create a random function input.
input=FunctionInput('input',[FourierFunction(
    .1, 8,.4,i, 0) for i in range(D)],
    Units.UNK) #Create a white noise input function .1 base freq,
               #max freq 8 rad/s, and RMS of .4; i makes one for
               #each dimension; 0 is the seed

```

```
net.add(input) #Add the input node to the network
net.connect(input,A)

# Create a modulated connection between the 'pre' and 'post' ensembles.
learning.make(net,errName='error', N_err=100, preName='pre', postName='post',
              rate=5e-7) #Make an error population with 100 neurons, and a learning
                        #rate of 5e-7

# Set the modulatory signal to compute the desired function
def square(x):
    return [xx*xx for xx in x]

net.connect('pre', 'error', func=square)
net.connect('post', 'error', weight=-1)

# Add a gate to turn learning on and off.
net.make_input('switch',[0]) #Create a controllable input function with
                             #a starting value of 0 and 0 in the two
                             #dimensions
gating.make(net,name='Gate', gated='error', neurons=40,
            pstc=0.01) #Make a gate population with 40 neurons, and a postsynaptic
                     #time constant of 10ms
net.connect('switch', 'Gate')

net.add_to_nengo()
```

2.6 Miscellaneous

2.6.1 Simple Vehicle

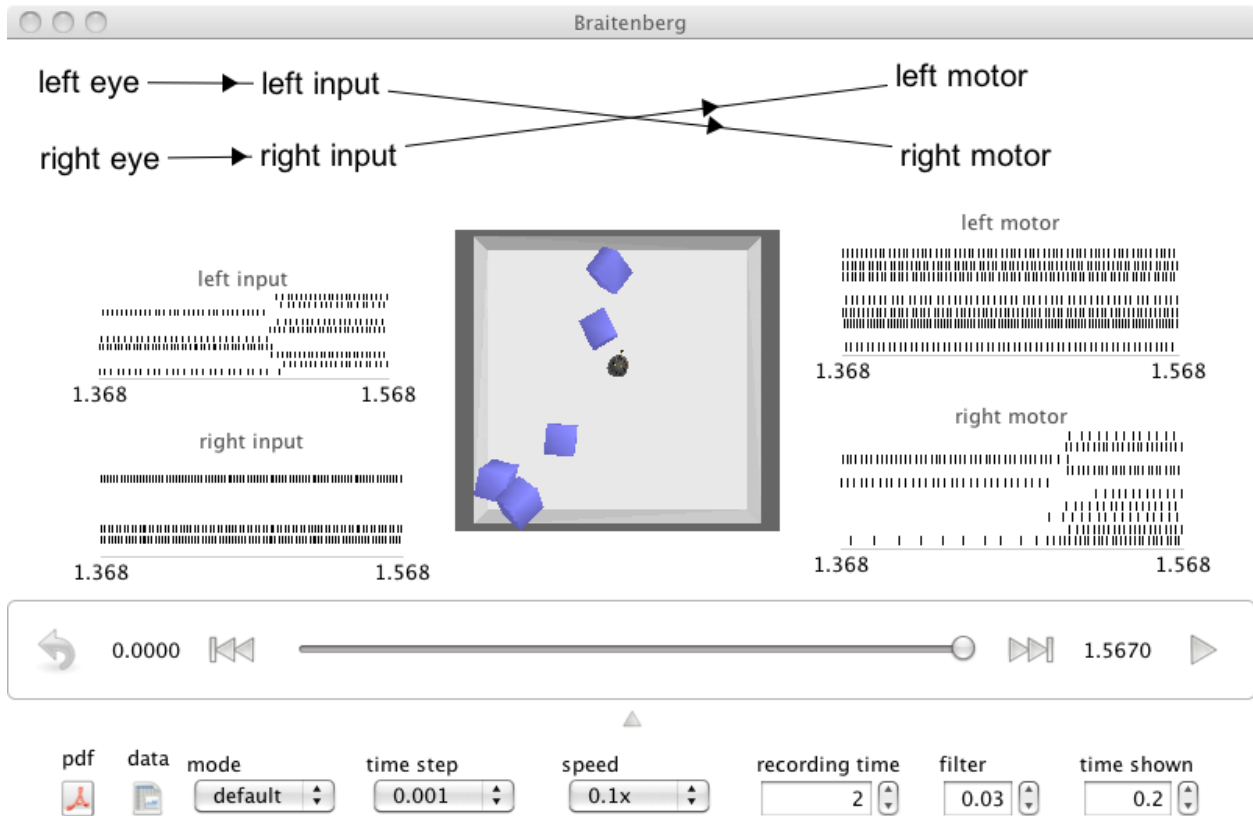
Purpose: This demo shows an example of integrating Nengo with a physics engine renderer.

Comments: The demo shows a simple Braitenburg vehicle controller attached to a simulated robot (a Dalek from Dr. Who), that drives in a physical simulation.

Most of the code is demonstrating how to construct a simple robot and a simple room in the physical simulator. Only the last few lines are used to construct the neural controller.

Usage: Hit play and the robot will drive around, avoiding the blocks and walls. The spikes are generated by measurements of distance from obstacles, and motor commands to the wheels.

Output: See the screen capture below.



Code:

```
from __future__ import generators
import nef
import space
from java.awt import Color
import ccm
import random

dt=0.001
N=10

class Bot (space.MD2) :
    def __init__(self) :
        space.MD2.__init__(self, 'python/md2/dalekx/tris.md2', 'python/md2/dalekx/imperial.png',
                           scale=0.02, mass=800, overdraw_scale=1.4)

        z=0
        s=0.7
        r=0.7
        self.wheels=[space.Wheel(-s,0,z,radius=r),
                     space.Wheel(s,0,z,radius=r),
                     space.Wheel(0,s,z,friction=0,radius=r),
                     space.Wheel(0,-s,z,friction=0,radius=r)]

    def start(self) :
        self.sch.add(space.MD2.start, args=(self,))
        self.range1=space.RangeSensor(0.3,1,0,maximum=6)
        self.range2=space.RangeSensor(-0.3,1,0,maximum=6)
        self.wheel1=0
        self.wheel2=0
```

```
while True:
    r1=self.range1.range
    r2=self.range2.range
    input1.functions[0].value=r1-1.8
    input2.functions[0].value=r2-1.8

    f1=motor1.getOrigin('X').getValues().getValues()[0]
    f2=motor2.getOrigin('X').getValues().getValues()[0]

    self.wheels[1].force=f1*600
    self.wheels[0].force=f2*600
    yield dt

class Room(space.Room):
    def __init__(self):
        space.Room.__init__(self, 10, 10, dt=0.01)
    def start(self):
        self.bot=Bot()
        self.add(self.bot, 0, 0, 1)
        #view=space.View(self, (0, -10, 5))

    for i in range(6):
        self.add(space.Box(1, 1, 1, mass=1, color=Color(0x8888FF),
            flat_shading=False), random.uniform(-5, 5), random.uniform(-5, 5),
            random.uniform(4, 6))

    self.sch.add(space.Room.start, args=(self,))

from ca.nengo.util.impl import NodeThreadPool, NEFGPUInterface

net=nef.Network('Braitenberg')

input1=net.make_input('right eye', [0])
input2=net.make_input('left eye', [0])

sense1=net.make("right input", N, 1)
sense2=net.make("left input", N, 1)
motor1=net.make("right motor", N, 1)
motor2=net.make("left motor", N, 1)

net.connect(input1, sense1)
net.connect(input2, sense2)
net.connect(sense2, motor1)
net.connect(sense1, motor2)

net.add_to_nengo()

r=ccm.nengo.create(Room)
net.add(r)
```

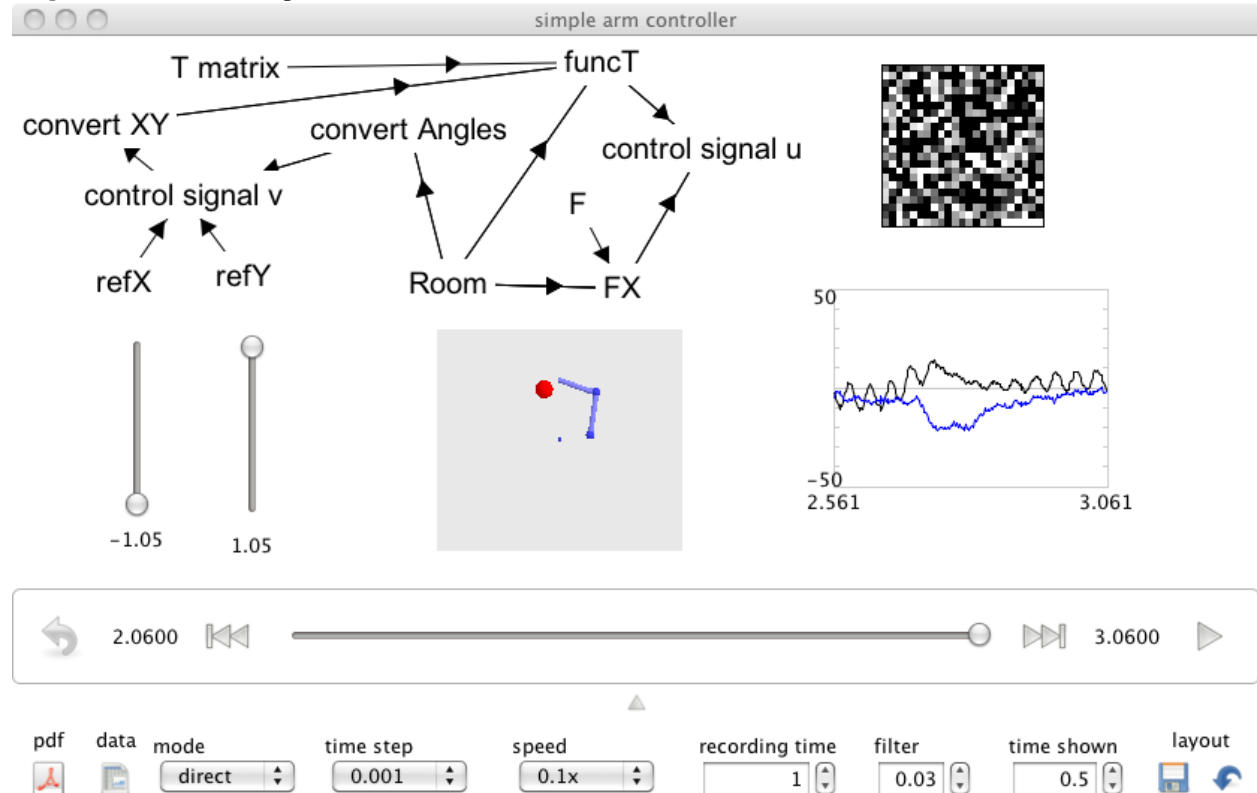
2.6.2 Arm Control

Purpose: This demo shows an example of having an interaction between a neural and non-neural dynamical simulation.

Comments: The majority of the simulation is a non-neural dynamical simulation, with just one crucial population being neural. That population is plotted in the visualizer, as are the generated control signals. The control signals are used to drive the arm which is run in the physics simulator.

Usage: When you run the network, it will reach to the target (red ball). If you change refX and refY, that will move the ball, and the arm will reach for the new target location.

Output: See the screen capture below.



Code:

```
from __future__ import generators
import sys
import nef

import space

from ca.nengo.math.impl import *
from ca.nengo.model.plasticity.impl import *
from ca.nengo.util import *
from ca.nengo.plot import *

from com.bulletphysics import *
from com.bulletphysics.linearmath import *
from com.bulletphysics.dynamics.constraintsolver import *
from javax.vecmath import Vector3f
from math import *
import java
from java.awt import Color

import ccm
import random
```

```
random.seed(11)

from math import pi
from com.threed.jpct import SimpleVector
from com.bulletphysics.linearmath import Transform
from javax.vecmath import Vector3f

dt=0.001
N=1
pstc=0.01

net = nef.Network('simple arm controller')

class getShoulder(ca.nengo.math.Function):
    def map(self,X):
        x = float(X[0])
        y = float(X[1])
        # make sure we're in the unit circle
        if sqrt(x**2+y**2) > 1:
            x = x / (sqrt(x**2+y**2))
            y = y / (sqrt(x**2+y**2))

        L1 = .5
        L2 = .5
        EPS = 1e-10
        D = (x**2 + y**2 - L1**2 - L2**2) / (2*L1*L2) # law of cosines
        if (x**2+y**2) < (L1**2+L2**2):
            D = -D

        # find elbow down angles from shoulder to elbow

        #java.lang.System.out.println("x: %f    y:%f"%(x,y))
        if D < 1 and D > -1:
            elbow = acos(D)
        else:
            elbow = 0

        if (x**2+y**2) < (L1**2+L2**2):
            elbow = pi - elbow

        if x==0 and y==0: y = y+EPS

        inside = L2*sin(elbow)/(sqrt(x**2+y**2))
        if inside > 1: inside = 1
        if inside < -1: inside = -1

        if x==0:
            shoulder = 1.5708 - asin(inside) # magic numbers from matlab
        else:
            shoulder = atan(y/x) - asin(inside)
        if x < 0: shoulder = shoulder + pi

        return shoulder
    def getDimension(self):
        return 2

class getElbow(ca.nengo.math.Function):
    def map(self,X):
```



```

x = float(X[0])
y = float(X[1])
# make sure we're in the unit circle
if sqrt(x**2+y**2) > 1:
    x = x / (sqrt(x**2+y**2))
    y = y / (sqrt(x**2+y**2))
L1 = .5
L2 = .5
D = (x**2 + y**2 - L1**2 - L2**2) / (2*L1*L2) # law of cosines
if (x**2+y**2) < (L1**2+L2**2):
    D = -D

# find elbow down angles from shoulder to elbow

if D < 1 and D > -1:
    elbow = acos(D)
else:
    elbow = 0

if (x**2+y**2) < (L1**2+L2**2):
    elbow = pi - elbow

return elbow
def getDimension(self):
    return 2

class getX(ca.nengo.math.Function):
    def map(self,X):
        shoulder = X[0]
        elbow = X[1]
        L1 = .5
        L2 = .5

        return L1*cos(shoulder)+L2*cos(shoulder+elbow)

    def getDimension(self):
        return 2

class getY(ca.nengo.math.Function):
    def map(self,X):
        shoulder = X[0]
        elbow = X[1]
        L1 = .5
        L2 = .5

        return L1*sin(shoulder)+L2*sin(shoulder+elbow)

    def getDimension(self):
        return 2

# input functions
refX=net.make_input('refX', [-1])
refY=net.make_input('refY', [1])
Tfunc=net.make_input('T matrix', [1,0,0,1])
F=net.make_input('F', [-1,0,-1,0,0,-1,0,-1])

```

```
# neural populations
convertXY=net.make("convert XY",N,2)
convertAngles=net.make("convert Angles",N,2)
funcT=net.make("funcT",N,6)
FX=net.make("FX",N,12)
controlV=net.make("control signal v",N,2) # calculate 2D control signal
controlU=net.make("control signal u",500,2, quick=True) # calculates
                                                         #jkoint torque control
                                                         #signal

# add terminations
convertXY.addDecodedTermination('refXY',[[1,0],[0,1]],pstc,False)
convertAngles.addDecodedTermination('shoulder',[[1],[0]],pstc,False)
convertAngles.addDecodedTermination('elbow',[[0],[1]],pstc,False)

FX.addDecodedTermination('inputFs',[[1,0,0,0,0,0,0,0],[0,1,0,0,0,0,0,0],
    [0,0,1,0,0,0,0,0], \
    [0,0,0,1,0,0,0,0],[0,0,0,0,1,0,0,0],[0,0,0,0,0,1,0,0],[0,0,0,0,0,0,1,0],
    [0,0,0,0,0,0,0,1], \
    [0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0]],
    pstc,False)
FX.addDecodedTermination('X1',[[0],[0],[0],[0],[0],[0],[0],[0],[0],[1],[0],[0],
    [0]],pstc,False)
FX.addDecodedTermination('X2',[[0],[0],[0],[0],[0],[0],[0],[0],[0],[0],[1],[0],
    [0]],pstc,False)
FX.addDecodedTermination('X3',[[0],[0],[0],[0],[0],[0],[0],[0],[0],[0],[0],[1],
    [0]],pstc,False)
FX.addDecodedTermination('X4',[[0],[0],[0],[0],[0],[0],[0],[0],[0],[0],[0],[0],
    [1]],pstc,False)

funcT.addDecodedTermination('shoulderRef',[[1],[0],[0],[0],[0],[0]],pstc,False)
funcT.addDecodedTermination('elbowRef',[[0],[1],[0],[0],[0],[0]],pstc,False)
funcT.addDecodedTermination('shoulder',[[0],[0],[0],[0],[0],[0]],pstc,False)
funcT.addDecodedTermination('elbow',[[0],[0],[0],[0],[0],[0]],pstc,False)
funcT.addDecodedTermination('inputTs',[[0,0,0,0],[0,0,0,0],[1,0,0,0],[0,1,0,0],
    [0,0,1,0],[0,0,0,1]],pstc,False)

controlV.addDecodedTermination('inputCurrentX',[[-1],[0]],pstc,False)
controlV.addDecodedTermination('inputCurrentY',[[0],[-1]],pstc,False)
controlV.addDecodedTermination('inputRefX',[[1],[0]],pstc,False)
controlV.addDecodedTermination('inputRefY',[[0],[1]],pstc,False)

controlU.addDecodedTermination('inputFuncT1',[[1],[0]],pstc,False)
controlU.addDecodedTermination('inputFuncT2',[[0],[1]],pstc,False)
controlU.addDecodedTermination('inputFX1',[[1],[0]],pstc,False)
controlU.addDecodedTermination('inputFX2',[[0],[1]],pstc,False)

# add origins
interpreter=DefaultFunctionInterpreter()

convertXY.addDecodedOrigin('elbowRef',[getElbow()], "AXON")
convertXY.addDecodedOrigin('shoulderRef',[getShoulder()], "AXON")

convertAngles.addDecodedOrigin('currentX',[getX()], "AXON")
convertAngles.addDecodedOrigin('currentY',[getY()], "AXON")

FX.addDecodedOrigin('FX1',[interpreter.parse("x0*x8+x1*x9+x2*x10+x3*x11",12)],
    "AXON")
```

```

FX.addDecodedOrigin('FX2', [interpreter.parse("x4*x8+x5*x9+x6*x10+x7*x11", 12)],
    "AXON")

funcT.addDecodedOrigin('funcT1', [interpreter.parse("x0*x2+x1*x3", 6)], "AXON")
funcT.addDecodedOrigin('funcT2', [interpreter.parse("x0*x4+x1*x5", 6)], "AXON")

controlU.addDecodedOrigin('u1', [interpreter.parse("x0", 2)], "AXON")
controlU.addDecodedOrigin('u2', [interpreter.parse("x1", 2)], "AXON")

# add projections
net.connect(controlV.getOrigin('X'), convertXY.getTermination('refXY'))

net.connect(refX.getOrigin('origin'), controlV.getTermination('inputRefX'))
net.connect(refY.getOrigin('origin'), controlV.getTermination('inputRefY'))

net.connect(convertAngles.getOrigin('currentX'), controlV.getTermination(
    'inputCurrentX'))
net.connect(convertAngles.getOrigin('currentY'), controlV.getTermination(
    'inputCurrentY'))

net.connect(F.getOrigin('origin'), FX.getTermination('inputFs'))

net.connect(convertXY.getOrigin('shoulderRef'), funcT.getTermination(
    'shoulderRef'))
net.connect(convertXY.getOrigin('elbowRef'), funcT.getTermination('elbowRef'))

net.connect(Tfunc.getOrigin('origin'), funcT.getTermination('inputTs'))

net.connect(funcT.getOrigin('funcT1'), controlU.getTermination('inputFuncT1'))
net.connect(funcT.getOrigin('funcT2'), controlU.getTermination('inputFuncT2'))
net.connect(FX.getOrigin('FX1'), controlU.getTermination('inputFX1'))
net.connect(FX.getOrigin('FX2'), controlU.getTermination('inputFX2'))

net.add_to_nengo()

class Room(space.Room):
    def __init__(self):
        space.Room.__init__(self, 10, 10, gravity=0, color=[Color(0xFFFFFF),
            Color(0xFFFFFF), Color(0xEEEEEE), Color(0xDDDDDD),
            Color(0xCCCCCC), Color(0xBBBBBB)])
    def start(self):

        self.target=space.Sphere(0.2, mass=1, color=Color(0xFF0000))
        self.add(self.target, 0, 0, 2)

        torso=space.Box(0.1, 0.1, 1.5, mass=100000, draw_as_cylinder=True,
            color=Color(0x4444FF))
        self.add(torso, 0, 0, 1)

        upperarm=space.Box(0.1, 0.7, 0.1, mass=0.5, draw_as_cylinder=True,
            color=Color(0x8888FF), overdraw_radius=1.2, overdraw_length=1.2)
        self.add(upperarm, 0.7, 0.5, 2)
        upperarm.add_sphere_at(0, 0.5, 0, 0.1, Color(0x4444FF), self)
        upperarm.add_sphere_at(0, -0.5, 0, 0.1, Color(0x4444FF), self)

        lowerarm=space.Box(0.1, 0.75, 0.1, mass=0.1, draw_as_cylinder=True,
            color=Color(0x8888FF), overdraw_radius=1.2, overdraw_length=1.1)

```

```
self.add(lowerarm,0.7,1.5,2)

shoulder=HingeConstraint(torso.physics,upperarm.physics,
                        Vector3f(0.7,0.1,1),Vector3f(0,-0.5,0),
                        Vector3f(0,0,1),Vector3f(0,0,1))

elbow=HingeConstraint(upperarm.physics,lowerarm.physics,
                    Vector3f(0,0.5,0),Vector3f(0,-0.5,0),
                    Vector3f(0,0,1),Vector3f(0,0,1))

shoulder.setLimit(-pi/2,pi/2+.1)
elbow.setLimit(-pi,0)

self.physics.addConstraint(elbow)
self.physics.addConstraint(shoulder)

#upperarm.physics.applyTorqueImpulse(Vector3f(0,0,300))
#lowerarm.physics.applyTorqueImpulse(Vector3f(0,0,300))

self.sch.add(space.Room.start,args=(self,))
self.update_neurons()
self.upperarm=upperarm
self.lowerarm=lowerarm
self.shoulder=shoulder
self.elbow=elbow
self.hinge1=self.shoulder.hingeAngle
self.hinge2=self.elbow.hingeAngle
self.upperarm.physics.setSleepingThresholds(0,0)
self.lowerarm.physics.setSleepingThresholds(0,0)

def update_neurons(self):
    while True:
        scale=0.0003
        m1=controlU.getOrigin('u1').getValues().getValues()[0]*scale
        m2=controlU.getOrigin('u2').getValues().getValues()[0]*scale
        v1=Vector3f(0,0,0)
        v2=Vector3f(0,0,0)
        #java.lang.System.out.println("m1: %f    m2:%f"%(m1,m2))

        self.upperarm.physics.applyTorqueImpulse(Vector3f(0,0,m1))
        self.lowerarm.physics.applyTorqueImpulse(Vector3f(0,0,m2))

        self.hinge1=-(self.shoulder.hingeAngle-pi/2)
        self.hinge2=-self.elbow.hingeAngle
        #java.lang.System.out.println("angle1: %f
        #angle2:%f"%(self.hinge1,self.hinge2))

        self.upperarm.physics.getAngularVelocity(v1)
        self.lowerarm.physics.getAngularVelocity(v2)
        # put bounds on the velocity possible
        if v1.z > 2:
            self.upperarm.physics.setAngularVelocity(Vector3f(0,0,2))
        if v1.z < -2:
            self.upperarm.physics.setAngularVelocity(Vector3f(0,0,-2))
        if v2.z > 2:
            self.lowerarm.physics.setAngularVelocity(Vector3f(0,0,2))
        if v2.z < -2:
            self.lowerarm.physics.setAngularVelocity(Vector3f(0,0,-2))
```

```

self.upperarm.physics.getAngularVelocity(v1)
self.lowerarm.physics.getAngularVelocity(v2)

wt=Transform()
#self.target.physics.motionState.getWorldTransform(wt)
wt.setIdentity()

tx=controlV.getTermination('inputRefX').input
if tx is not None:
    wt.origin.x=tx.values[0]+0.7
else:
    wt.origin.x=0.7
ty=controlV.getTermination('inputRefY').input
if ty is not None:
    wt.origin.y=ty.values[0]+0.1
else:
    wt.origin.y=0.1
wt.origin.z=2

self.target.physics.motionState.worldTransform=wt

self.vel1=v1.z
self.vel2=v2.z

yield 0.0001

r=ccm.nengo.create(Room)
net.add(r)

# need to make hinge1, hinge2, vell, and vel external nodes and hook up
# the output to the FX matrix
r.exposeOrigin(r.getNode('hinge1').getOrigin('origin'),'shoulderAngle')
r.exposeOrigin(r.getNode('hinge2').getOrigin('origin'),'elbowAngle')
r.exposeOrigin(r.getNode('vell').getOrigin('origin'),'shoulderVel')
r.exposeOrigin(r.getNode('vel2').getOrigin('origin'),'elbowVel')

net.connect(r.getOrigin('shoulderAngle'),FX.getTermination('X1'))
net.connect(r.getOrigin('elbowAngle'),FX.getTermination('X2'))
net.connect(r.getOrigin('shoulderVel'),FX.getTermination('X3'))
net.connect(r.getOrigin('elbowVel'),FX.getTermination('X4'))

net.connect(r.getOrigin('shoulderAngle'),convertAngles.getTermination(
    'shoulder'))
net.connect(r.getOrigin('elbowAngle'),convertAngles.getTermination('elbow'))
net.connect(r.getOrigin('shoulderAngle'),funcT.getTermination('shoulder'))
net.connect(r.getOrigin('elbowAngle'),funcT.getTermination('elbow'))

# put everything in direct mode
net.network.setMode(ca.nengo.model.SimulationMode.DIRECT)
# except the last population
controlU.setMode(ca.nengo.model.SimulationMode.DEFAULT)

```


NENGO SCRIPTING


This documentation describes the scripting libraries available for use in Nengo.

3.1 Scripting Interface


Many researchers prefer to create models by writing scripts rather than using a graphical user interface. To accomplish this, we have embedded a scripting system within Nengo.

The scripting language used in Nengo is Python. Detailed documentation on Python syntax can be found at <http://www.python.org/>. Nengo uses Jython (<http://jython.org>) to interface between the script and the underlying Java implementation of Nengo. This allows us to use Python syntax and still have access to all of the underlying Java code.

3.1.1 Running scripts from a file

You can create scripts using your favourite text editor. When saved, they should have a `.py` extension. To run the script, click on the  icon, or select `File->Open from file` from the main menu.

3.1.2 Running scripts from the console

You can also use scripting to interact with a model within Nengo. Click on the  icon to toggle the script console (or press `Ctrl-P` to jump to it). You can now type script commands that will be immediately run. For example, you can type this at the console:

```
print "Hello from Nengo"
```

When using the script console, you can refer to the currently selected object using the word `that`. For example, if you click on a component of your model (it should be highlighted in yellow to indicate it is selected), you can get its name by typing:

```
print that.name
```

You can also change aspects of the model this way. For example, click on an ensemble of neurons and change the number of neurons in that ensemble by typing:

```
that.neurons=75
```

You can also run a script file from the console with this command, with `script_name` replaced by the name of the script to run:


```
run script_name.py
```

Pressing the up and down arrows will scroll through the history of your console commands.

3.1.3 Running scripts from the command line

You can also run scripts from the command line. This allows you to simply run the model without running Nengo itself. To do this, instead of running `nengo` with `nengo` (or `nengo.bat` on Windows), do:

```
nengo-cl script_name.py
```

Of course, since this bypasses the Nengo graphical interface, you won't be able to click on the  icon to show the model. Instead, you should add this to the end of your model:

```
net.view()
```

3.1.4 Importing other libraries

The scripting system allows you to make use of any Java library and any 100% Python library. There are two methods for doing this in Python.

First, you can do it this way:

```
import ca.nengo.utils
```

but then when you use the library, you will have to provide its full name:

```
y=ca.nengo.utils.DataUtils.filter(x,0.005)
```

The other option is to import this way:

```
from ca.nengo.utils import *
```

This allows you to just do:

```
y=DataUtils.filter(x,0.005)
```

3.2 Scripting Basics

The Jython scripting interface in Nengo provides complete access to all of the underlying Java functionality. This provides complete flexibility, but requires users to manually create all of the standard components (Origins, Terminations, Projections, etc), resulting in fairly repetitive code.

To simplify the creation of Nengo models, we have developed the `nef` module as a wrapper around the most common functions needed for creating Nengo networks. As an example, the following code will create a new network with an input that feeds into ensemble A which feeds into ensemble B:

```
import nef
net=nef.Network('Test Network')
net.add_to_nengo()
input=net.make_input('input', values=[0])
```



```
A=net.make('A',neurons=100,dimensions=1)
B=net.make('B',neurons=100,dimensions=1)
net.connect(input,A)
net.connect(A,B)
```

These scripts can be created with any text editor, saved with a `.py` extension, and run in Nengo by choosing File->Open from the menu or clicking on the blue *Open* icon in the upper-left. All of the examples in the demo directory have been written using this system.

3.2.1 Creating a network

The first thing you need to do is to create your new network object and tell it to appear in the Nengo user interface:

```
import nef
net=nef.Network('My Network')
net.add_to_nengo()
```

3.2.2 Creating an ensemble

Now we can create ensembles in our network. You must specify the name of the ensemble, the number of neurons, and the number of dimensions:

```
A=net.make('A',neurons=100,dimensions=1)
B=net.make('B',1000,2)
C=net.make('C',50,10)
```

You can also specify the radius for the ensemble. The neural representation will be optimized to represent values inside a sphere of the specified radius. So, if you have a 2-dimensional ensemble and you want to be able to represent the value $(10, -10)$, you should have a radius of around 15:

```
D=net.make('D',100,2,radius=15)
```

3.2.3 Creating an input

To create a simple input that has a constant value (which can then be controlled interactive mode interface), do the following:

```
inputA=net.make_input('inputA',values=[0])
```

The name can be anything, and the values is an array of the required length. So, for a 5-dimensional input, you can do:

```
inputB=net.make_input('inputB',values=[0,0,0,0,0])
```

or:

```
inputC=net.make_input('inputC',values=[0]*5)
```

You can also use values other than 0:

```
inputD=net.make_input('inputD',values=[0.5,-0.3, 27])
```

3.2.4 Computing linear functions

To have components be useful, they have to be connected to each other. To assist this process, the `nengo.Network.connect()` function will create the necessary Origins and/or Terminations as well as the Projection:

```
A=net.make('A',100,1)
B=net.make('B',100,1)
net.connect(A,B)
```

You can refer to networks by name or by reference:

```
net.make('A',100,1)
net.make('B',100,1)
net.connect('A','B')
```

You can also specify a transformation matrix (to allow for the computation of any linear function) and a post-synaptic time constant:

```
A=net.make('A',100,2)
B=net.make('B',100,3)
net.connect(A,B,transform=[[0,0.5],[1,0],[0,0.5]],pstc=0.03)
```

3.2.5 Computing nonlinear functions

To compute nonlinear functions, you can specify a function to compute and an origin will automatically be created:

```
A=net.make('A',100,1)
B=net.make('B',100,1)
def square(x):
    return x[0]*x[0]
net.connect(A,B,func=square)
```

This also works for highly complex functions:

```
A=net.make('A',100,5)
B=net.make('B',100,1)
import math
def strange(x):
    if x[0]<0.4: return 0.3
    elif x[1]*x[2]<0.3: return math.sin(x[3])
    else: return x[4]
net.connect(A,B,func=strange)
```

3.3 Configuring Neural Ensembles

When creating a neural ensemble, a variety of parameters can be adjusted. Some of these parameters are set to reflect the physiological properties of the neurons being modelled, while others can be set to improve the accuracy of the transformations computed by the neurons.

3.3.1 Membrane time constant and refractory period

The two parameters for Leaky-Integrate-and-Fire neurons are the membrane time constant (`tau_rc`) and the refractory period (`tau_ref`). These parameters are set when creating the ensemble, and default to 0.02 seconds for the

membrane time constant and 0.002 seconds for the refractory period:

```
D=net.make('D',100,2,tau_rc=0.02,tau_ref=0.002)
```

Empirical data on the membrane time constants for different types of neurons in different parts of the brain can be found at <http://ctn.uwaterloo.ca/~cnrglab/?q=node/547>.

3.3.2 Maximum firing rate

You can also specify the maximum firing rate for the neurons. It should be noted that it will always be possible to force these neurons to fire faster than this specified rate. Indeed, the actual maximum firing rate will always be $1/\tau_{ref}$, since if enough current is forced into the simulated neuron, it will fire as fast as its refractory period will allow. However, what we can specify with this parameter is the *normal* operating range for the neurons. More technically, this is the maximum firing rate *assuming that the neurons are representing a value within the ensemble's radius*.

In most cases, we specify this by giving a range of maximum firing rates, and each neuron will have a maximum chosen uniformly from within this range. This gives a somewhat biologically realistic amount of diversity in the tuning curves. The following line makes neurons with maximums between 200Hz and 400Hz:

```
E=net.make('E',100,2,max_rate=(200,400))
```

Alternatively, we can specify a particular set of maximum firing rates, and each neuron will take on a value from the provided list. If there are more neurons than elements in the list, the provided values will be re-used:

```
F=net.make('F',100,2,max_rate=[200,250,300,350,400])
```

Note: The type of brackets used is important!! Python has two types of brackets for this sort of situation: round brackets `()` and square brackets `[]`. Round brackets create a *tuple*, which we use for indicating a range of values to randomly choose within, and square brackets create a *list*, which we use for specifying a list of particular value to use.

3.3.3 Intercept

The intercept is the point on the tuning curve graph where the neuron starts firing. For example, for a one-dimensional ensemble, a neuron with a preferred direction vector of `[1]` and an intercept of 0.3 will only fire when representing values above 0.3. If the preferred direction vector is `[-1]`, then it will only fire for values below 0.3. In general, the neuron will only fire if the dot product of `x` (the value being represented) and the preferred direction vector, divided by the radius, is greater than the intercept. Note that since we divide by the radius, the intercepts will always be normalized to be between -1 and 1.

While this parameter can be used to help match the tuning curves observed in the system being modelled, one important other use is to build neural models that can perfectly represent the value 0. For example, if a 1-dimensional neural ensemble is built with intercepts in the range (0.3,1), then no neurons at all will fire for values between -0.3 and 0.3. This means that any value in that range (i.e. any small value) will be rounded down to exactly 0. This can be useful for optimizing thresholding and other functions where many of the output values are zero.

By default, intercepts are uniformly distributed between -1 and 1. The intercepts can be specified by providing either a range, or a list of values:

```
G=net.make('G',100,2,intercept=(-1,1))
H=net.make('H',100,2,intercept=[-0.8,-0.4,0.4,0.8])
```

Note: The type of brackets used is important!! Python has two types of brackets for this sort of situation: round brackets `()` and square brackets `[]`. Round brackets create a *tuple*, which we use for indicating a range of values to randomly choose within, and square brackets create a *list*, which we use for specifying a list of particular value to use.

3.3.4 Encoders (a.k.a. preferred direction vectors)

You can specify the encoders (preferred direction vectors) for the neurons. By default, the encoders are chosen uniformly from the unit sphere. Alternatively, you can specify those encoders by providing a list. The encoders given will automatically be normalized to unit length:

```
F=net.make('F',100,2,encoders=[[1,0],[-1,0],[0,1],[0,-1]])
G=net.make('G',100,2,encoders=[[1,1],[1,-1],[-1,1],[-1,-1]])
```

This allows you to make complex sets of encoders by creating a list with the encoders you want. For example, the following code creates an ensemble with 100 neurons, half of which have encoders chosen from the unit circle, and the other half of which are aligned on the diagonals:

```
import random
import math

encoders=[]                                # create an empty list to store the encoders
for i in range(50):
    theta=random.uniform(-math.pi,pi)      # choose a random direction between -pi and pi
    encoders.append([math.sin(theta),math.cos(theta)]) # add the encoder
for i in range(50):
    encoders.append(random.choice([[1,1],[1,-1],[-1,1],[-1,-1]])) # add an aligned encoder

G=net.make('G',100,2,encoders=encoders)    # create the ensemble
```

3.4 Speeding up Network Creation (and Making Consistent Models)

Whenever Nengo creates an ensemble, it needs to compute a decoder. This is done via the NEF method of doing a least-squares minimization computation. For large ensembles (~500 neurons or more), this can take some time, since it needs to invert an NxN matrix.

By default, an ensemble consists of neurons with randomly generated encoders, intercepts, and maximum firing rates (withing the specified ranges). This means that a new decoder must be computed for every new ensemble. However, if we specify a random number seed, all of these parameters will be consistent: that is, if we run the script again and re-create the same ensemble with the same random number seed, Nengo will detect this and re-use the previously computed decoder. This greatly speeds up running the script. (Of course, the first time the script is run, it will still take the same amount of time).

There are two ways to specify the random number seed. The first is to set the `fixed_seed` parameter when creating the Network:

```
net=Network('My Network',fixed_seed=5)
```

This tells Nengo to use the random seed 5 for every single ensemble that is created within this network. In other words, if you now do the following:

```
A=net.make('A',neurons=300,dimensions=1)
B=net.make('B',neurons=300,dimensions=1)
```

then the neurons within ensembles A and B will be *identical*. This will also be true for ensembles within a `NetworkArray`:

```
C=net.make_array('C',300,10)
```

which will create an array of ten identical ensembles, each with 300 identical neurons.

3.4.1 Avoiding Identical Ensembles

Using `fixed_seed` allows you to make networks very quickly, since Nengo takes advantage of these identical ensembles and does not need to re-compute decoders. However, it leads to neurally implausible models, since neurons are not identical in this way. As an alternative, you can use the second way of specifying seeds, the `seed` parameter:

```
net=Network('My Network',seed=5)
```

With this approach, Nengo will create a new seed for every ensemble, based on your initial seed value. There will be no identical neurons in your model. However, if you re-run your script, Nengo will re-generate exactly the same neurons, and so will be able to re-use the previously computed decoders.

3.4.2 Sharing Consistent Models

A useful side-effect of the `seed` parameter is that it allows us to be sure that *exactly* the same model is generated on different computers. That is, if you build a model and set a specific seed, then other researchers can run that same script and will get exactly the same model as you created. In this way we can ensure consistent performance.

3.4.3 Individual Differences

One way to think about the `seed` parameter is as a way to generate neural models with individual differences. The same script run with a different `seed` value will create a slightly different model, due to the low-level neural differences. This may affect the overall behaviour.

3.5 Adding Arbitrary Code to a Model

Nengo models are composed of Nodes. Each Node has Origins (outputs) and Terminations (inputs), allowing them to be connected up to perform operations. Nengo has built-in Node types for Neurons, Ensembles (groups of Neurons), Networks, Arrays, and so on. However, when creating a complex model, we may want to create our own Node type. This might be to provide custom input or output from a model, or it can be used to have a non-neural component within a larger model.

Technically, the only requirement for being a Node is that an object supports the `ca.nengo.model.Node` interface. However, for the vast majority of cases, it is easier to make use of the `nef.SimpleNode` wrapper class.

3.5.1 SimpleNode

You create a `nef.SimpleNode` by subclassing, defining functions to be called for whatever Origins and/or Terminations you want for this object. The functions you define will be called once every time step (usually 0.001 seconds). These functions can contain arbitrary code, allowing you to implement anything you want to. For example, the following code creates a Node that outputs a sine wave:

```
import nef
import math

net=nef.Network('Sine Wave')
```

```
# define the SimpleNode
class SineWave(nef.SimpleNode):
    def origin_wave(self):
        return [math.sin(self.t)]
wave=net.add(SineWave('wave'))

neurons=net.make('neurons',100,1)

# connect the SimpleNode to the group of neurons
net.connect(wave.getOrigin('wave'),neurons)

net.add_to_nengo()
```

3.5.2 Outputs (a.k.a. origins)

You can create as many outputs as you want from a SimpleNode, as long as each one has a distinct name. Each origin consists of a single function that will get called once per time-step and must return an array of floats.

When defining this function, it is often useful to know the current simulation time. This can be accessed as `self.t`, and is the time (in seconds) of the beginning of the current time-step (the end of the current time step is `self.t_end`):

```
class ManyOrigins(nef.SimpleNode):

    # an origin that is 0 for t<0.5 and 1 for t>=0.5
    def origin_step(self):
        if self.t<0.5: return [0]
        else: return [1]

    # a triangle wave with period of 1.0 seconds
    def origin_triangle(self):
        x=self.t%1.0
        if x<0.5: return [x*2]
        else: return [2.0-x*2]

    # a sine wave and a cosine with frequency 10 Hz
    def origin_circle(self):
        theta=self.t*(2*math.pi)*10
        return [math.sin(theta),math.cos(theta)]
```

When connecting a SimpleNode to other nodes, we need to specify which origin we are connecting. The name of the origin is determined by the function definition, of the form `origin_<name>`:

```
A=net.make('A',100,1)
B=net.make('B',200,2)
many=net.add(ManyOrigins('many'))
net.connect(many.getOrigin('triangle'),A)
net.connect(many.getOrigin('circle'),B)
```

3.5.3 Inputs (a.k.a. Terminations)

To provide input to a SimpleNode, we define terminations. These are done in a similar manner as origins, but these functions take an input value (usually denoted `x`), which is an array of floats containing the input.

When defining the termination, we also have to define the number of dimensions expected. We do this by setting the `dimensions` parameter (which defaults to 1). We can also specify the post-synaptic time constant at this termination by setting the `pstc` parameter (default is None).

For example, the following object takes a 5-dimensional input vector and outputs the largest of the received values:

```
class Largest(nef.SimpleNode):
    def init(self):
        self.largest=0
    def termination_values(self,x,dimensions=5,pstc=0.01):
        self.largest=max(x)
    def origin_largest(self):
        return [self.largest]

net=nef.Network('largest')
input=net.make_input('input',[0]*5)
largest=net.add(Largest('largest'))
net.connect(input,largest.getTermination('values'))
```

Note: When making a component like this, make sure to define an initial value for `largest` (or whatever internal parameter is being used to map inputs to outputs) inside the `init(self)` function. This function will be called before the origins are evaluated so that there is a valid `self.largest` return value.

3.5.4 Arbitrary Code

You can also define a function that will be called every time step, but which is *not* tied to a particular Origin or Termination. This function is called `tick`. Here is a simple example where this function simply prints the current time:

```
class Time(nef.SimpleNode):
    def tick(self):
        print 'The current time in the simulation is:',self.t
```

As a more complex example, here is a `tick` function used to save spike raster information to a text file while the simulation runs:

```
class SpikeSaver(nef.SimpleNode):
    def tick(self):
        f=file('data.csv','a+')
        data=A.getOrigin('AXON').getValues().getValues()
        f.write('%1.3f,%s\n'%(self.t,list(data)))
        f.close()

net=nef.Network('Spike Saver example')
A=net.make('A',50,1)
saver=net.add(SpikeSaver('saver'))
```

3.6 Connection Weights

3.6.1 Viewing synaptic weights

For most Nengo models, we do not need to worry about the exact connection weights, as Nengo solves for the optimal connection weights using the Neural Engineering Framework. However, we sometimes would like to have direct access to this weight matrix, so we can modify it in various ways. We do this using the `weight_func` argument in the `nef.Network.connect()` function. For example, to simply print the solved-for optimal weights, we can do the following:

```
A=net.make('A',100,1)
B=net.make('B',100,1)

def print_weights(w):
    print w
    return w

net.connect(A,B,weight_func=print_weights)
```

3.6.2 Adjusting weights

We can also adjust these weights by modifying them and returning the new matrix. The following code randomly adjusts each connection weight by an amount sampled from a normal distribution of standard deviation 0.001:

```
A=net.make('A',100,1)
B=net.make('B',100,1)

def randomize(w):
    for i in range(len(w)):
        for j in range(len(w[i])):
            w[i][j]+=random.gauss(0,0.001)
    return w

net.connect(A,B,weight_func=randomize)
```

3.6.3 Sparsification

We can also use this approach to enforce sparsity constraints on our networks. For example, one could simply find all the small weights in the *w* matrix and set them to zero. The following code takes a slightly different approach that we have found to be a bit more robust. Here, if we want 20% connectivity (i.e. each neuron in population A is only connected to 20% of the neurons in population B), we simply randomly select 80% of the weights in the matrix and set them to zero. To make up for this reduction in connectivity, we also increase the remaining weights by scaling them by 1.0/0.2:

```
A=net.make('A',100,1)
B=net.make('B',100,1)

p=0.2
def sparsify(w):
    for i in range(len(w)):
        for j in range(len(w[i])):
            if random.random()<p:
                w[i][j]/=p
            else:
                w[i][j]=0.0
    return w

net.connect(A,B,weight_func=sparsify)
```


3.7 Scripting Tips

3.7.1 Common transformation matrices

To simplify creating connection matrices for high-dimensional ensembles, you can use three additional parameters in the `nengo.Network.connect()` function: `weight`, `index_pre`, and `index_post`. `weight` specifies the overall gain on the connection across all dimensions, and defaults to 1. For example:

```
A=net.make('A',100,3)
B=net.make('B',100,3)
net.connect(A,B,weight=0.5)  # makes a transform matrix of
                             # [[0.5,0,0],[0,0.5,0],[0,0,0.5]]
```

Note that the system by default assumes the identity matrix for the connection.

If you don't want the identity matrix, and would prefer some other connectivity, specify `index_pre` and `index_post`. These indicate which dimensions in the first ensemble should be mapped to which dimensions in the second ensemble. For example:

```
A=net.make('A',100,3)
B=net.make('B',100,1)
net.connect(A,B,index_pre=2)
                             # makes a transform matrix of
                             # [[0,0,1]]
```

```
A=net.make('A',100,1)
B=net.make('B',100,3)
net.connect(A,B,index_post=0)
                             # makes a transform matrix of
                             # [[1],[0],[0]]
```

```
A=net.make('A',100,4)
B=net.make('B',100,2)
net.connect(A,B,index_pre=[1,2])
                             # makes a transform matrix of
                             # [[0,1,0,0],[0,0,1,0]]
                             # which makes B hold the 2nd and 3rd element of A
```

```
A=net.make('A',100,4)
B=net.make('B',100,3)
net.connect(A,B,index_pre=[1,2],index_post=[0,1])
                             # makes a transform matrix of
                             # [[0,1,0,0],[0,0,1,0],[0,0,0,0]]
                             # which makes B hold the 2nd and 3rd element of A
                             # in its first two elements
```

3.7.2 Adding noise to the simulation

To make the inputs to neurons noisy, you can specify an amount of noise and a noise frequency (how often a new noise value is sampled from the uniform distribution between `-noise` and `+noise`). Each neuron will sample from this distribution at this rate, and add the resulting value to its input current. The frequency defaults to 1000Hz:

```
H=net.make('H',50,1,noise=0.5,noise_frequency=1000)
```

3.7.3 Random inputs

Here is how you can convert an input to provide a randomly changing value, rather than a constant:

```
input=net.make_input('input',[0])
input.functions=[FourierFunction(0.1,10,0.5,0,0)]
```

This will produce a randomly varying input. This input will consist of random sine waves varying from 0.1Hz to 10Hz, in 0.5Hz increments. The random number seed used is 0.

3.7.4 Changing modes: spiking, rate, and direct

You can set an ensemble to be simulated as spiking neurons, rate neurons, or directly (no neurons). The default is spiking neurons:

```
J=net.make('J',neurons=1,dimensions=100,mode='direct')
K=net.make('K',neurons=50,dimensions=1,mode='rate')
```

One common usage of direct mode is to quickly test out algorithms without worrying about the neural implementation. This can be especially important when creating algorithms with large numbers of dimensions, since they would require large numbers of neurons to simulate. It can often be much faster to test the algorithm without neurons in direct mode before switching to a realistic neural model.

Note: When using direct mode, you may want to decrease the number of neurons in the population to 1, as this makes it much faster to create the ensemble.

3.7.5 Arrays of ensembles

When building models that represent large numbers of dimensions, it is sometimes useful to break an ensemble down into sub-ensembles, each of which represent a subset of dimensions. Instead of building one large ensemble to represent 100 dimensions, we might have 10 ensembles that represent 10 dimensions each, or 100 ensembles representing 1 dimension each.

The main advantage of this is speed: It is much faster for the NEF methods to compute decoders for many small ensembles, rather than one big one.

However, there is one large disadvantage: you cannot compute nonlinear functions that use values in two different ensembles. One of the core claims of the NEF is that we can only approximate nonlinear functions of two (or more) variables if there are neurons that respond to *both* dimensions. However, it is still possible to compute any linear function.

We create an array by specifying its length and (optionally) the number of dimensions per ensemble (the default is 1):

```
M=net.make_array('M',neurons=100,length=10,dimensions=1)
```

You can also use all of the parameters available in `nef.Network.make()` to configure the properties of the neurons.

Note: The *neurons* parameter specifies the number of neurons *in each ensemble*, not the total number of neurons!

The resulting array can be used just like a normal ensemble. The following example makes a single 10-dimensional ensemble and a network array of 5 two-dimensional ensembles and connects one to the other:

```
A=net.make_array('A',neurons=100,length=5,dimensions=2)
B=net.make('B',neurons=500,dimensions=10)
net.connect(A,B)
```

When computing nonlinear functions with an array, the function is applied to *each ensemble separately*. The following computes the products of five pairs of numbers, storing the results in a single 5-dimensional array:

```
A=net.make_array('A',neurons=100,length=5,dimensions=2)
B=net.make('B',neurons=500,dimensions=5)
def product(x):
    return x[0]*x[1]
net.connect(A,B,func=product)
```

3.7.6 Matrix operations

To simplify the manipulation of matrices, we have added a version of JNumeric to Nengo. This allows for a syntax similar to Matlab, but based on the NumPy python module.

To use this for matrix manipulation, you will first have to convert any matrix you have into an array object:

```
a=[[1,2,3],[4,5,6]]          # old method
a=array([[1,2,3],[4,5,6]])    # new method
```

You can also specify the storage format to be used as follows:

```
a=array([[1,2,3],[4,5,6]],typecode='f')
# valid values for the typecode parameter:
#      'i'  int32
#      's'  int16
#      'l'  int64
#      '1'  int8
#      'f'  float32
#      'd'  float64
#      'F'  complex64
#      'D'  complex128
```

The first important thing you can do with this array is use full slice syntax. This is the `[:]` notation used to access part of an array. A slice is a set of three values, all of which are optional. `[a:b:c]` means to start at index a, go to index b (but not include index b), and have a step size of c between items. The default for a is 0, for b is the length of the array, and c is 1. For multiple dimensions, we put a comma between slices for each dimension. The following examples are all for a 2D array. Note that the order of the 2nd and 3rd parameters are reversed from matlab, and it is all indexed starting at 0:

```
a[0]      # the first row
a[0,:]    # the first row
a[:,0]    # the first column
a[0:3]    # the first three rows
a[:,0:3]  # the first three columns
a[:, :3]  # the first three columns (the leading zero is optional)
a[:,2:]   # all columns from the 2nd to the end (the end value is optional)
a[:, :-1] # all columns except the last one (negative numbers index from the end)
a[::2]    # just the even-numbered rows (skip every other row)
a[::3]    # every third row
a[::-1]   # all rows in reverse order
a[:, ::2] # just the even-numbered columns (skip every other column)
```

```
a[:,::-1] # all columns in reverse order

a.T      # efficient transpose (doesn't use any more memory)
```

With such an array, you can perform element-wise operations as follows:

```
c=a+b      # same as .+ in matlab
c=a*b      # same as .* in matlab
b=cos(a)   # computes cosine of all values in a
# other known functions: add, subtract, multiply, divide, remainder, power,
# arccos, arccosh, arcsinh, arctan, arctanh, ceil, conjugate, imaginary,
# cos, cosh, exp, floor, log, log10, real, sin, sinh, sqrt, tan, tanh,
# maximum, minimum, equal, not_equal, less, less_equal, greater,
# greater_equal, logical_and, logical_or, logical_xor, logical_not,
# bitwise_and, bitwise_or, bitwise_xor
```

You can also create particular arrays:

```
arange(5)   # same as array(range(5))==[0,1,2,3,4]
arange(2,5) # same as array(range(2,5))==[2,3,4]

eye(5)      # 5x5 identity matrix
ones((3,2)) # 3x2 matrix of all 1
ones((3,2),dtype='f') # 3x2 matrix of all 1.0 (floating point values)
zeros((3,2)) # 3x2 matrix of all 0
```

The following functions help manipulate the shape of a matrix:

```
a.shape      # get the current size of the matrix
b=reshape(a, (3,4)) # convert to a 3x4 matrix (must already have 12 elements)
b=resize(a, (3,4)) # convert to a 3x4 matrix (can start at any size)
b=ravel(a)    # convert to a 1-D vector
b=diag([1,2,3]) # create a diagonal matrix with the given values
```

Some basic linear algebra operations are available:

```
c=dot(a,b)
c=dot(a,a.T)
c=innerproduct(a,a)
c=convolve(a,b)
```

And a Fourier transform:

```
b=fft(a)
a=ifft(b)
```

The following functions also exist:

```
# argmax, argsort, argmin, asarray, bitwise_not, choose, clip, compress,
# concatenate, fromfunction, indices, nonzero, searchsorted, sort, take
# where, tostring, fromstring, trace, repeat, diagonal
# sum, cumsum, product, cumproduct, alltrue, sometrue
```

The vast majority of the time, you can use these objects the same way you would a normal list of values (i.e. for specifying transformation matrices). If you ever need to explicitly convert one back into a list, you can call `.tolist()`:

```
a=array([1,2,3])
b=a.tolist()
```

These functions are all available at the Nengo console and in any script called using the `run` command. To access them in a separate script file, you need to call:

```
from numeric import *
```

3.8 List of Classes

3.8.1 nef.Network

class `nef.Network` (*name*, *quick*=False, *seed*=None, *fixed_seed*=None)

Wraps a Nengo network with a set of helper functions for simplifying the creation of Nengo models.

This system is meant to allow short, concise code to create Nengo models. For example, we can make a communication channel like this:

```
import nef
net=nef.Network('Test Network')
input=net.make_input('input', values=[0])
A=net.make('A', neurons=100, dimensions=1)
B=net.make('B', neurons=100, dimensions=1)
net.connect(input, A)
net.connect(A, B)
net.add_to_nengo()
```

This will automatically create the necessary origins, terminations, ensemble factories, and so on needed to create this network.

Parameters

- **name** (*string or NetworkImpl*) – If a string, create and wrap a new `NetworkImpl` with the given *name*. If an existing `NetworkImpl`, then create a wrapper around that network.
- **quick** (*boolean*) – Default setting for the *quick* parameter in `nef.Network.make()`. Note: the use of this parameter is not encouraged any more: use *seed*=<number> or *fixed_seed*=<number> instead.
- **fixed_seed** (*int*) – random number seed to use for creating ensembles. Every ensemble will use this seed value, resulting in identical neurons in ensembles that have the same parameters. Automatically makes use of the *quick* system to avoid re-computing ensembles.
- **seed** (*int*) – random number seed to use for creating ensembles. This one seed is used only to start the random generation process, so each neural group created will be different (unlike the *fixed_seed* parameter).

make (*name*, *neurons*, *dimensions*, *tau_rc*=0.02, *tau_ref*=0.002, *max_rate*=(200, 400), *intercept*=(-1, 1), *radius*=1, *encoders*=None, *decoder_noise*=0.10000000000000001, *eval_points*=None, *noise*=None, *noise_frequency*=1000, *mode*='spike', *add_to_network*=True, *node_factory*=None, *decoder_sign*=None, *seed*=None, *quick*=None, *storage_code*='')

Create and return an ensemble of neurons.

Parameters

- **name** (*string*) – name of the ensemble (must be unique)
- **neurons** (*integer*) – number of neurons in the ensemble
- **dimensions** (*integer*) – number of dimensions to represent
- **tau_rc** (*float*) – membrane time constant
- **tau_ref** (*float*) – refractory period

- **max_rate** (*tuple or list*) – range for uniform selection of maximum firing rate in Hz (as a 2-tuple) or a list of maximum rate values to use
- **intercept** (*tuple or list*) – normalized range for uniform selection of tuning curve x-intercept (as 2-tuple) or a list of intercept values to use
- **radius** (*float*) – representational range
- **encoders** (*list*) – list of encoder vectors to use (if None, uniform distribution around unit sphere). The provided encoders will be automatically normalized to unit length.
- **decoder_noise** (*float*) – amount of noise to assume when calculating decoders
- **eval_points** (*list*) – list of points to do optimization over
- **noise** (*float*) – current noise to inject, chosen uniformly from (-noise,noise)
- **noise_frequency** (*float*) – sampling rate (how quickly the noise changes)
- **mode** (*string*) – simulation mode ('direct', 'rate', or 'spike')
- **node_factory** (*ca.nengo.model.impl.NodeFactory*) – a factory to use instead of the default LIF factory (for creating ensembles with neurons other than LIF)
- **decoder_sign** (*None, +1, or -1*) – +1 for positive decoders, -1 for negative decoders. Set to None to allow both.
- **quick** (*boolean or None*) – if True, saves data from a created ensemble and will re-use it in the future when creating an ensemble with the same parameters as this one. If None, uses the Network default setting. Note: the use of this parameter is not encouraged any more: use `seed=<number>` or `fixed_seed=<number>` in the Network constructor instead.
- **seed** (*int*) – random number seed to use. Will be passed to both `random.seed()` and `ca.nengo.math.PDFTools.setSeed()`. If this is None and the Network was constructed with a seed parameter, a seed will be randomly generated.
- **storage_code** (*string*) – an extra parameter to allow different quick files even if all other parameters are the same
- **add_to_network** (*boolean*) – flag to indicate if created ensemble should be added to the network

Returns the newly created ensemble

make_array (*name, neurons, length, dimensions=1, **args*)

Create and return an array of ensembles. This acts like a high-dimensional ensemble, but actually consists of many sub-ensembles, each one representing a separate dimension. This tends to be much faster to create and can be more accurate than having one huge high-dimensional ensemble. However, since the neurons represent different dimensions separately, we cannot compute nonlinear interactions between those dimensions.

Note: When forming neural connections from an array to another ensemble (or another array), any specified function to be computed will be computed on each ensemble individually (with the results concatenated together). For example, the following code creates an array and then computes the sum of the squares of each value within it:

```
net=nef.Network('Squaring Array')
input=net.make_input('input',[0,0,0,0,0])
A=net.make_array('A',neurons=100,length=5)
B=net.make('B',neurons=100,dimensions=1)
net.connect(input,A)
def square(x):
```

```

return x[0]*x[0]
net.connect(A,B,transform=[1,1,1,1,1],func=square)

```

All of the parameters from `nef.Network.make()` can also be used.

If the `storage_code` parameter is used, you may use `%d` (or variants such as `%02d`) in the storage code, which will be replaced by the index number of the ensemble in the array. Thus, `storage_code='a%02d'` will become `a00` for the first ensemble, `a01` for the second, `a02` for the third, and so on.

If the `encoders` parameter is used, you can provide either the standard array of encoders (e.g. `[[1], [-1]]`) or a list of sets of encoders for each ensemble (e.g. `[[[1]], [[-1]]]`).

Parameters

- **name** (*string*) – name of the ensemble array (must be unique)
- **neurons** (*integer*) – number of neurons in each ensemble
- **length** (*integer*) – number of ensembles in the array
- **dimensions** (*integer*) – number of dimensions each ensemble represents

Returns the newly created `nef.array.NetworkArray`

make_input (*name, values, zero_after_time=None*)

Create and return a `FunctionInput` of dimensionality `len(values)` with `values` as its constants. Python functions can be provided instead of fixed values.

Parameters

- **name** (*string*) – name of created node
- **values** (*list, function, string, or dict*) – numerical values for the function. If a list, can contain a mixture of floats and functions (floats are fixed input values, and functions are called with the current time and must return a single float). If `values` is a function, will be called with the current time and can return either a single float or a list of floats. If a string, will be treated as a filename of a csv file with the first number in each row indicating the time and the other numbers giving the value of the function. If a dictionary, the keys,value pairs in the dictionary will be treated as time,value pairs.
- **zero_after_time** (*float or None*) – if not `None`, any fixed input value will change to 0 after this amount of time

Returns the created `FunctionInput`

make_fourier_input (*name, dimensions=None, base=1, high=10, power=0.5, seed=None*)

Create and return a `FunctionInput` that randomly varies. The variation is generated by randomly generating fourier components with frequencies that are multiples of `base` up to `high`, and normalized to have an rms power of `power`.

Parameters

- **name** (*string*) – name of created node
- **dimensions** (*int or None*) – dimensionality of the input. If `None`, will default to the longest of any of the lists given in the other parameters, or 1 if there are no lists.
- **base** (*float or list*) – fundamental (lowest) frequency for the fourier series. If a list, will use different values for each dimension. Default is 1Hz
- **high** (*float or list*) – maximum frequency for the fourier series. If a list, will use different values for each dimension. Default is 10Hz

- **power** (*float or list*) – RMS power for the random function. If a list, will use different values for each dimension. Default is 0.5
- **seed** (*int or list or None.*) – random number seed to use. If a list, will use different values for each dimension. If None, a random seed will be chosen.

Returns the created FunctionInput

compute_transform (*dim_pre, dim_post, weight=1, index_pre=None, index_post=None*)

Helper function used by `nef.Network.connect()` to create a *dim_pre* by *dim_post* matrix. All values are either 0 or *weight*. *index_pre* and *index_post* are used to determine which values are non-zero, and indicate which dimensions of the pre-synaptic ensemble should be routed to which dimensions of the post-synaptic ensemble.

For example, with *dim_pre*=2 and *dim_post*=3, *index_pre*=[0,1], *index_post*=[0,1] means to take the first two dimensions of pre and send them to the first two dimensions of post, giving a transform matrix of `[[1, 0], [0, 1], [0, 0]]`. If an index is None, the full range [0,1,2,...,N] is assumed, so the above example could just be *index_post*=[0,1]

Parameters

- **dim_pre** (*integer*) – first dimension of transform matrix
- **dim_post** (*integer*) – second dimension of transform matrix
- **weight** (*float*) – the non-zero value to put into the matrix
- **index_pre** (*list of integers or a single integer*) – the indexes of the pre-synaptic dimensions to use
- **index_post** (*list of integers or a single integer*) – the indexes of the post-synaptic dimensions to use

Returns a two-dimensional transform matrix performing the requested routing

connect (*pre, post, transform=None, weight=1, index_pre=None, index_post=None, pstc=0.01, func=None, weight_func=None, expose_weights=False, origin_name=None, modulatory=False, plastic_array=False, create_projection=True*)

Connect two nodes in the network.

pre and *post* can be strings giving the names of the nodes, or they can be the nodes themselves (FunctionInputs and NEFEnsembles are supported). They can also be actual Origins or Terminations, or any combination of the above. If *post* is set to an integer or None, an origin will be created on the *pre* population, but no other action will be taken.

pstc is the post-synaptic time constant of the new Termination

If *transform* is not None, it is used as the transformation matrix for the new termination. You can also use *weight*, *index_pre*, and *index_post* to define a transformation matrix instead. *weight* gives the value, and *index_pre* and *index_post* identify which dimensions to connect (see `nef.Network.compute_transform()` for more details). For example:

```
net.connect(A,B,weight=5)
```

with both A and B as 2-dimensional ensembles, will use `[[5, 0], [0, 5]]` as the transform. Also, you can do:

```
net.connect(A,B,index_pre=2,index_post=5)
```

to connect the 3rd element in A to the 6th in B. You can also do:

```
net.connect(A,B,index_pre=[0,1,2],index_post=[5,6,7])
```


to connect multiple elements.

If *func* is not None, a new Origin will be created on the pre-synaptic ensemble that will compute the provided function. The name of this origin will taken from the name of the function, or *origin_name*, if provided. If an origin with that name already exists, the existing origin will be used rather than creating a new one.

If *weight_func* is not None, the connection will be made using a synaptic connection weight matrix rather than a DecodedOrigin and a Decoded Termination. The computed weight matrix will be passed to the provided function, which is then free to modify any values in that matrix, returning a new one that will actually be used. This allows for direct control over the connection weights, rather than just using the once computed via the NEF methods. If you do not want to modify these weights, but do want Nengo to compute the weight matrix, you can just set *expose_weights* to True.

Parameters

- **pre** – The item to connect from. Can be a string (the name of the ensemble), an Ensemble (made via `nef.Network.make()`), an array of Ensembles (made via `nef.Network.make_array()`), a FunctionInput (made via `nef.Network.make_input()`), or an Origin.
- **post** – The item to connect to. Can be a string (the name of the ensemble), an Ensemble (made via `nef.Network.make()`), an array of Ensembles (made via `nef.Network.make_array()`), or a Termination.
- **transform** (*array of floats*) – The linear transform matrix to apply across the connection. If *transform* is T and *pre* represents *x*, then the connection will cause *post* to represent *Tx*. Should be an N by M array, where N is the dimensionality of *pre* and M is the dimensionality of *post*, but a 1-dimensional array can be given if either N or M is 1.
- **pstc** (*float*) – post-synaptic time constant for the neurotransmitter/receptor implementing this connection
- **weight** (*float*) – scaling factor for a transformation defined with *index_pre* and *index_post*. Ignored if *transform* is not None. See `nef.Network.compute_transform()`
- **index_pre** (*list of integers or a single integer*) – the indexes of the pre-synaptic dimensions to use. Ignored if *transform* is not None. See `nef.Network.compute_transform()`
- **index_post** (*list of integers or a single integer*) – the indexes of the post-synaptic dimensions to use. Ignored if *transform* is not None. See `nef.Network.compute_transform()`
- **func** (*function*) – function to be computed by this connection. If None, computes $f(x) = x$. The function takes a single parameter *x* which is the current value of the *pre* ensemble, and must return wither a float or an array of floats. For example:

```
def square(x):
    return x[0]*x[0]
net.connect(A,B,func=square)

def powers(x):
    return x[0],x[0]^2,x[0]^3
net.connect(A,B,func=powers)

def product(x):
    return x[0]*x[1]
net.connect(A,B,func=product)
```

- **origin_name** (*string*) – The name of the origin to create to compute the given function. Ignored if `func` is `None`. If an origin with this name already exists, the existing origin is used instead of creating a new one.
- **weight_func** (*function or None*) – if not `None`, converts the connection to use an explicit connection weight matrix between each neuron in the ensembles. This is mathematically identical to the default method (which simply uses the stored encoders and decoders for the ensembles), but much slower, since we are no longer taking advantage of the factorable weight matrix. However, using `weight_func` also allows explicit control over the individual connection weights, as the computed weight matrix is passed to `weight_func`, which can make changes to the matrix before returning it. If `weight_func` is a function taking one argument, it is passed the calculated weight matrix. If it is a function taking two arguments, it is passed the encoder and decoder.
- **expose_weights** – if `True`, set `weight_func` to the identity function. This makes the connection use explicit connection weights, but doesn't modify them in any way. Ignored if `weight_func` is not `None`.
- **modulatory** (*boolean*) – whether the created connection should be marked as modulatory, meaning that it does not directly affect the input current to the neurons, but instead may affect internal parameters in the neuron model.
- **plastic_array** (*boolean*) – configure the connection to be learnable. See `nef.Network.learn()`.
- **create_projection** (*boolean*) – flag to disable actual creation of the connection. If `False`, any needed Origin and/or Termination will be created, and the return value will be the tuple `(origin, termination)` rather than the created projection object.

Returns the created Projection, or `(origin, termination)` if `create_projection` is `False`.

learn (*post, learn_term, mod_term, rate=4.9999999999999998e-07, **kwargs*)

Apply a learning rule to a termination of the *post* ensemble. The *mod_term* termination will be used as an error signal that modulates the changing connection weights of *learn_term*.

Parameters

- **post** (*string or Ensemble*) – the ensemble whose termination will be changing, or the name of this ensemble
- **learn_term** (*string or Termination*) – the termination whose transform will be modified by the learning rule. This termination must be created with `plastic_array=True` in `nef.Network.connect()`
- **mod_term** (*string or Termination*) – the modulatory input to the learning rule; while this is technically not required by the plasticity functions in Nengo, currently there are no learning rules implemented that do not require modulatory input.
- **rate** (*float*) – the learning rate that will be used in the learning functions.

Todo

(Possible enhancement: make this 2D for stdp mode, different rates for `in_fcn` and `out_fcn`)

If *stdp* is `True`, a triplet-based spike-timing-dependent plasticity rule is used, based on that defined in:

Pfister, J. and Gerstner, W. (2006) Triplets of Spikes in a Model of Spike Timing-Dependent Plasticity. *J. Neurosci.*, 26: 9673 – 9682.

The parameters for this learning rule have the following defaults, and can be set as keyword arguments to this function call:

```
(a2Minus=5.0e-3, a3Minus=5.0e-3, tauMinus=70, tauX=70,
 a2Plus=5.0e-3, a3Plus=5.0e-3, tauPlus=70, tauY=70,
 decay=None, homeostatis=None)
```

If *stdp* is False, a rate-mode error minimizing learning rule is applied. The only parameter available here is whether or not to include an oja normalization term:

```
(oja=True)
```

learn_array (*array*, *learn_term*, *mod_term*, *rate*=4.999999999999998e-07, ***kwargs*)

Apply a learning rule to a termination of a `nef.array.NetworkArray` (an array of ensembles, created using `nef.Network.make_array()`).

See `nef.Network.learn()` for parameters.

add_to_nengo ()

Add the network to the Nengo user interface. If there is no user interface (i.e. if Nengo is being run via the command line only interface `nengo-cl`), then do nothing.

add_to (*world*=None)

Add the network to the given Nengo world object. If there is a network with that name already there, remove the old one. If *world* is None, it will attempt to find a running version of Nengo to add to. Deprecated since version 1.3: Use `nef.Network.add_to_nengo()` instead.

view (*play*=False)

Creates the interactive mode viewer for running the network

Parameters *play* (False or float) – Automatically starts the simulation running, stopping after the given amount of time

set_layout (*view*, *layout*, *control*)

Defines the graphical layout for the interactive plots

You can use this to specify a particular layout. This will replace the currently saved layout (if any). Useful when running a script on a new computer that does not have a previously saved layout (saving you from also copying over that layout file).

The arguments for this function call are generally made by opening up interactive plots, making the layout you want, saving the layout, and then copying the text in `layouts/<networkname>.layout`.

Parameters

- **view** (*dictionary*) – parameters for the window position
- **layout** (*list*) – list of all components to be shown and their parameters
- **control** (*dictionary*) – configuration parameters for the simulation

add (*node*)

Add the node to the network.

This is generally only used for manually created nodes, not ones created by calling `nef.Network.make()` or `nef.Network.make_input()` as these are automatically added. A common usage is with `nef.SimpleNode` objects, as in the following:

```
node=net.add(MyNode('name'))
```

Parameters *node* – the node to be added

Returns *node*

get (*name*, *default*=<type 'exceptions.Exception'>, *require_origin*=False)

Return the node with the given *name* from the network

releaseMemory ()

Attempt to release extra memory used by the Network. Call only after all connections are made.

getNeuronCount ()

Return the total number of neurons in this network

run (*time*, *dt*=0.001)

Run the simulation.

If called twice, the simulation will continue for *time* more seconds. To reset the simulation, call `nef.Network.reset()`. Typical use cases are to either simply call it once:

```
net.run(10)
```

or to call it multiple times in a row:

```
t=0
dt=0.1
while t<10:
    net.run(dt)
    t+=dt
```

Parameters

- **time** (*float*) – the amount of time (in seconds) to run for
- **dt** (*float*) – the size of the time step to use

reset ()

Reset the simulation.

Should be called if you have previously called `nef.Network.run()`, but now want to reset the simulation to its initial state.

log (*name*=None, *dir*=None, *filename*='%(*name*)s-%(*time*)s.csv', *interval*=0.001, *tau*=0.01)

Creates a `nef.Log` object which dumps data to a .csv file as the model runs.

See the `nef.Log` documentation for details.

Parameters

- **name** (*string*) – The name of the model. Defaults to the name of the Network object.
- **dir** (*string*) – The directory to place the .csv file into
- **filename** (*string*) – The filename to use. .csv will be added if it is not already there. Can use % (*name*) s to refer to the *name* of the model and % (*time*) s to refer to the start time of the model. Defaults to % (*name*) s-% (*time*) s.csv.
- **interval** (*float*) – The time interval between each row of the log. Defaults to 0.001 (1ms).
- **tau** (*float*) – The default filter time for data. Defaults to 0.01 (10ms).

set_view_function_1d (*node*, *basis*, *label*='1D function', *origin*='X', *minx*=-1, *maxx*=1, *miny*=-1, *maxy*=1)

Define a function representation for the given node.

This has no effect on the model itself, but provides a useful display in the interactive plots visualizer. The vector represented by the function is plotted by treating the vector values as weights for a set of basis functions. So, if a vector is (2,0,3) and the basis functions are x^2 , x , and 1, we get the polynomial $2*x^2+3$.

The provided basis function should accept two parameters: an index value indicating which basis function should be computed, and x , indicating the x value to compute the basis function at. For example, for polynomials, the basis functions would be computed as:

```
def polynomial_basis(index, x):
    return x**index
```

Parameters

- **node** (*Node*) – The Nengo component that represents the vector
- **basis** (*function*) – The set of basis functions to use. This is a single function accepting two parameters: the basis index and the x value. It should return the corresponding y value.
- **origin** (*string*) – Which origin to use. Defaults to X .
- **label** (*string*) – The text that will appear in the pop-up menu to activate this view
- **minx** (*float*) – minimum x value to plot
- **maxx** (*float*) – maximum x value to plot
- **miny** (*float*) – minimum y value to plot
- **maxy** (*float*) – maximum y value to plot

3.8.2 nef.SimpleNode

`class nef.SimpleNode(name)`

A SimpleNode allows you to put arbitrary code as part of a Nengo model.

This object has Origins and Terminations which can be used just like any other Nengo component. Arbitrary code can be run every time step, making this useful for simulating sensory systems (reading data from a file or a webcam, for example), motor systems (writing data to a file or driving a robot, for example), or even parts of the brain that we don't want a full neural model for (symbolic reasoning or declarative memory, for example).

Origins and terminations are defined by subclassing SimpleNode. For example, the following code creates a node that takes a single input and outputs the square of that input:

```
class SquaringNode(nef.SimpleNode):
    def init(self):
        self.value=0
    def termination_input(self, x):
        self.value=x[0]*x[0]
    def origin_output(self):
        return [self.value]
square=net.add(SquaringNode('square'))
net.connect(A, square.getTermination('input'))
net.connect(square.getOrigin('output'), B)
```

You can have as many origins and terminations as you like. The dimensionality of the origins are set by the length of the returned vector of floats. The dimensionality of the terminations can be set by specifying the dimensionality in the method definition:

```
class SquaringFiveValues(nef.SimpleNode):
    def init(self):
        self.value=0
    def termination_input(self, x, dimensions=5):
        self.value=[xx*xx for xx in x]
```

```
def origin_output(self):  
    return [self.value]
```

You can also specify a post-synaptic time constant for the filter on the terminations in the method definition:

```
class SquaringNode(nef.SimpleNode):  
    def init(self):  
        self.value=0  
    def termination_input(self, x, pstc=0.01):  
        self.value=x[0]*x[0]  
    def origin_output(self):  
        return [self.value]
```

There is also a special method called `tick()` that is called once per time step. It is called after the terminations but before the origins:

```
class HelloNode(nef.SimpleNode):  
    def tick(self):  
        print 'Hello world'
```

The current time can be accessed via `self.t`. This value will be the time for the beginning of the current time step. The end of the current time step is `self.t_end`:

```
class TimeNode(nef.SimpleNode):  
    def tick(self):  
        print 'Time:', self.t
```

Parameters `name` (*string*) – the name of the created node

create_origin (*name, func*)

Adds an origin to the SimpleNode.

Every timestep the function *func* will be called. It should return a vector which is the output value at this origin.

Any member functions of the form `origin_name` will automatically be created in the constructor, so the following two nodes are equivalent:

```
class Node1(nef.SimpleNode):  
    def origin_test(self):  
        return [0]  
node1=Node1('node1')  
  
node2=nef.SimpleNode('node2')  
def test():  
    return [0]  
node2.create_origin('test', test)
```

Parameters

- **name** (*string*) – the name of the origin
- **func** (*function*) – the function to call

Note: The function *func* will be called once by `create_origin` to determine the dimensionality it returns.

create_termination (*name, func*)

Adds a termination to the SimpleNode.

Every timestep the function *func* will be called. It must accept a single parameter, which is a list of floats representing the current input to the termination.

Any member functions of the form `termination_name` will automatically be created in the constructor, so the following two nodes are equivalent:

```
class Node1(nef.SimpleNode):
    def termination_test(self, x):
        self.data=x[0]
node1=Node1('node1')

node2=nef.SimpleNode('node2')
def test(x):
    node2.data=x[0]
node2.create_termination('test', test)
```

By default, the termination will be 1 dimensional. To change this, specify a different value in the function definition:

```
class Node3(nef.SimpleNode):
    def termination_test(self, x, dimensions=4):
        self.data=x[0]*x[1]+x[2]*x[3]
```

By default, no post-synaptic filter is applied. To change this, specify a `pstc` value:

```
class Node4(nef.SimpleNode):
    def termination_test(self, x, pstc=0.01):
        self.data=x[0]
```

Parameters

- **name** (*string*) – the name of the termination
- **func** (*function*) – the function to call.

Note: The function *func* will be called once by `create_termination` with an input of all zeros.

`tick()`

An extra utility function that is called every time step.

Override this to create custom behaviour that isn't necessarily tied to a particular input or output. Often used to write spike data to a file or produce some other sort of custom effect.

`setTau(name, tau)`

Change the post-synaptic time constant for a termination.

Parameters

- **name** (*string*) – the name of the termination to change
- **tau** (*float*) – the desired post-synaptic time constant

`init()`

Initialize the node.

Override this to initialize any internal variables. This will also be called whenever the simulation is reset:

```
class DoNothingNode(nef.SimpleNode):
    def init(self):
        self.value=0
    def termination_input(self, x, pstc=0.01):
```

```
self.value=x[0]
def origin_output(self):
    return [self.value]
```

3.8.3 nef.array.NetworkArray

class `nef.array.NetworkArray` (*name, nodes*)

Collects a set of NEFEnsembles into a single network.

Create a network holding an array of nodes. An ‘X’ Origin is automatically created which concatenates the values of each internal element’s ‘X’ Origin.

This object is meant to be created using `nef.Network.make_array()`, allowing for the efficient creation of neural groups that can represent large vectors. For example, the following code creates a NetworkArray consisting of 50 ensembles of 1000 neurons, each of which represents 10 dimensions, resulting in a total of 500 dimensions represented:

```
net=nef.Network('Example Array')
A=net.make_array('A',neurons=1000,length=50,dimensions=10,quick=True)
```

The resulting NetworkArray object can be treated like a normal ensemble, except for the fact that when computing nonlinear functions, you cannot use values from different ensembles in the computation, as per NEF theory.

Parameters

- **name** (*string*) – the name of the NetworkArray to create
- **nodes** (*list of NEFEnsembles*) – the nodes to combine together

createEnsembleOrigin (*name*)

Create an Origin that concatenates the values of internal Origins.

Parameters **name** (*string*) – The name of the Origin to create. Each internal node must already have an Origin with that name.

addDecodedOrigin (*name, functions, nodeOrigin*)

Create a new Origin. A new origin is created on each of the ensembles, and these are grouped together to create an output.

This method uses the same signature as `ca.nengo.model.nef.NEFEnsemble.addDecodedOrigin()`

Parameters

- **name** (*string*) – the name of the newly created origin
- **functions** (*list of ca.nengo.math.Function objects*) – the functions to approximate at this origin
- **nodeOrigin** (*string*) – name of the base Origin to use to build this function approximation (this will always be ‘AXON’ for spike-based synapses)

addTermination (*name, matrix, tauPSC, isModulatory*)

Create a new termination. A new termination is created on each of the ensembles, which are then grouped together. This termination does not use NEF-style encoders; instead, the matrix is the actual connection weight matrix. Often used for adding an inhibitory connection that can turn off the whole array (by setting *matrix* to be all -10, for example).

Parameters

- **name** (*string*) – the name of the newly created origin

- **matrix** (*2D array of floats*) – synaptic connection weight matrix (NxM where M is the total number of neurons in the NetworkArray)
- **tauPSC** (*float*) – post-synaptic time constant
- **isModulatory** (*boolean*) – False for normal connections, True for modulatory connections (which adjust neural properties rather than the input current)

addPlasticTermination (*name, matrix, tauPSC, decoder, weight_func=None*)

Create a new termination. A new termination is created on each of the ensembles, which are then grouped together.

If decoders are not known at the time the termination is created, then pass in an array of zeros of the appropriate size (i.e. however many neurons will be in the population projecting to the termination, by number of dimensions).

addDecodedTermination (*name, matrix, tauPSC, isModulatory*)

Create a new termination. A new termination is created on each of the ensembles, which are then grouped together.

addIndexTermination (*name, matrix, tauPSC, isModulatory=False, index=None*)

Create a new termination. A new termination is created on the specified ensembles, which are then grouped together. This termination does not use NEF-style encoders; instead, the matrix is the actual connection weight matrix. Often used for adding an inhibitory connection that can turn off selected ensembles within the array (by setting *matrix* to be all -10, for example).

Parameters

- **name** (*string*) – the name of the newly created origin
- **matrix** (*2D array of floats*) – synaptic connection weight matrix (NxM where M is the total number of neurons in the ensembles to be connected)
- **tauPSC** (*float*) – post-synaptic time constant
- **isModulatory** (*boolean*) – False for normal connections, True for modulatory connections (which adjust neural properties rather than the input current)
- **index** – The indexes of the ensembles to connect to. If set to None, this function behaves exactly like `nef.NetworkArray.addTermination()`.

ADVANCED NENGO USAGE

4.1 Interactive Plots Layout Files

The interactive plots mode in Nengo shows a wide variety of information about a running Nengo model, including graphs to interpret the value being represented within a network and interactive controls which allow the inputs to the system to be varied.

The exact configuration of this view is saved in a text file in the `layouts` directory, using the name of the network as an identifier. For example, the layout for the multiplication demo is saved as `layouts/Multiply.layout` and looks like:

The first line gives the size and location of the window, the last line gives the setting of the various simulation parameters, and the middle lines define the various plots that are displayed.

While this saved file format is human-readable, it is not meant to be hand-coded. The best way to create a layout is to open up the Interactive plots window, create the layout you want, and save it. A corresponding file will be created in the `layouts` folder.

4.1.1 Specifying a Layout

If you want to, you can define a layout in a script by cutting and pasting from a `.layout` file. For this, you can use the `nef.Network.set_layout()` function.

For example, here we define a simple network and directly specify the layout to use:

```
import nef

net=nef.Network('My Test Model')
input=net.make_input('input',[0,0])
neuron=net.make('neurons',100,2,quick=True)
net.connect(input,neuron)
net.add_to_nengo()

net.set_layout({'height': 473, 'x': -983, 'width': 798, 'state': 0, 'y': 85},
[('neurons', None, {'x': 373, 'height': 32, 'label': 0, 'width': 79, 'y': 76}),
('input', None, {'x': 53, 'height': 32, 'label': 0, 'width': 51, 'y': 76}),
('neurons', 'voltage grid', {'x': 489, 'height': 104, 'auto_improve': 0,
                             'label': 0, 'width': 104, 'rows': None, 'y': 30}),
('neurons', 'value|X', {'x': 601, 'height': 105, 'sel_dim': [0, 1], 'label': 0,
                        'width': 158, 'autozoom': 0, 'last_maxy': 1.0, 'y': 29}),
('neurons', 'preferred directions', {'x': 558, 'height': 200, 'label': 0,
```

```
        'width': 200, 'y': 147})),
('neurons', 'XY plot|X', {'width': 200, 'autohide': 1, 'last_maxy': 1.0,
                          'sel_dim': [0, 1], 'y': 148, 'label': 0, 'x': 346,
                          'height': 200, 'autozoom': 1}),
('input', 'control', {'x': 67, 'height': 200, 'range': 1.0, 'label': 1,
                      'limits': 1, 'width': 120, 'limits_w': 0, 'y': 145}]),
{'sim_spd': 4, 'rcd_time': 4.0, 'filter': 0.03, 'dt': 0, 'show_time': 0.5})
```

This ability is useful when sending a model to someone else, so that they will automatically see the particular set of graphs you specify. This can be easier than also sending the `.layout` file.

4.2 Creating a Drag And Drop Template

Nengo comes with a variety of templates: pre-built components that can be used to build your models. These are the various icons on the left side of the screen that can be dragged in to your model.

These components are defined in `python/nef/templates`. There is one file for each item, and the following example uses `thalamus.py`.

The file starts with basic information, including the full name (title) of the component, the text to be used in the interface (label), and an image to use as an icon. The image should be stored in `/images/nengoIcons`:

```
title='Thalamus'
label='Thalamus'
icon='thalamus.png'
```

Next, we define the parameters that should be set for the component. These can be strings (`str`), integers (`int`), real numbers (`float`), or checkboxes (`bool`). For each one, we must indicate the name of the parameter, the label text, the type, and the help text:

```
params=[
    ('name', 'Name', str, 'Name of thalamus'),
    ('neurons', 'Neurons per dimension', int, 'Number of neurons to use'),
    ('D', 'Dimensions', int, 'Number of actions the thalamus can represent'),
    ('useQuick', 'Quick mode', bool,
     'If true, the same distribution of neurons will be used for each action'),
]
```

Next, we need a function that will test if the parameters are valid. This function will be given the parameters as a dictionary and should return a string containing the error message if there is an error, or not return anything if there is no error:

```
def test_params(net, p):
    try:
        net.network.getNode(p['name'])
        return 'That name is already taken'
    except:
        pass
```

Finally, we define the function that actually makes the component. This function will be passed in a `nef.Network` object that corresponds to the network we have dragged the template into, along with all of the parameters specified in the `params` list above. This script can now do any scripting calculations desired to build the model:

```
def make(net, name='Network Array', neurons=50, D=2, useQuick=True):
    thal = net.make_array(name, neurons, D, max_rate=(100, 300),
                          intercept=(-1, 0), radius=1, encoders=[[1]],
                          quick=useQuick)
```

```
def addOne(x):
    return [x[0]+1]
net.connect(thal, None, func=addOne, origin_name='xBiased',
            create_projection=False)
```

The last step to make the template appear in the Nengo interface is to add it to the list in `python/nengo/templates/__init__.py`.

4.3 Running Experiments in Nengo

Once a model has been designed, we often want to run controllable experiments to gather statistics about the performance of the model. As an example, we may want to read the input data from a file and save the corresponding outputs to a separate file. This allows us to automate the process of running these simulations, rather than using the interactive plots viewer.

4.3.1 Inputs

Input can be provided for the simulation using the `nengo.SimpleNode` approach, covered in more detail in [Adding Arbitrary Code to a Model](#). Here, we create an origin that has a fixed set of input stimuli, and shows each one for 100 milliseconds each:

```
inputs=[[ 1, 1],
        [ 1,-1],
        [-1,-1],
        [-1, 1],
        ]
dt=0.001
steps_per_input=100

class Input(nengo.SimpleNode):
    def origin_input(self, dimensions=2):
        step=int(round(self.t/dt))           # find time step we are on
        index=(step/steps_per_input)%len(inputs) # find stimulus to show
        return inputs[index]
```

This origin will cycle through showing the values 1, 1, 1, -1, -1, -1, and -1, 1 for 0.1 seconds each. Instead of manually specifying the inputs, we could also have read these values from a file:

```
inputs=[]
for line in file('inputs.csv').readlines():
    row=[float(x) for x in line.strip().split(',') ]
    inputs.append(row)
```

4.3.2 Outputs

When running experiments, we often don't want the complete record of every output the network makes over time. Instead, we're interested in what it is doing at specific points. For this case, we want to find out what the model's output is for each of the inputs. In particular, we want its output value just before we change the input to the next one in the list. The following code will collect these values and save them to a file called `experiments.csv`:

```
class Output(nengo.SimpleNode):
    def termination_save(self, x, dimensions=1, pstc=0.01):
        step=int(round(self.t/dt))
```

```
if step%steps_per_input==(steps_per_input-1):
    f=file('experiment.csv','a+')
    f.write('%d,%1.3f\n'%(step/steps_per_input,x[0]))
f.close()
```

4.3.3 Connecting the Model

For this particular example, here is a model that simply computes the product of its inputs. The inputs to the model are connected to the Input node and the outputs go to the Output node to be saved:

```
net=nef.Network('Experiment Example')
input=net.add(Input('input'))           # create the input node
output=net.add(Output('output'))        # create the output node
A=net.make('A',100,2,radius=1.5)
B=net.make('B',50,1)
net.connect(input.getOrigin('input'),A)  # connect the input
net.connect(B,output.getTermination('save')) # connect the output

def multiply(x):
    return x[0]*x[1]
net.connect(A,B,func=multiply)

net.add_to_nengo()
```

4.3.4 Running the Simulation

The model so far should run successfully within Nengo using the standard approach of going into the interactive plots mode and clicking the run button. However, we can also have the model automatically run right from within the script. This bypasses the visual display, making it run faster. The following commands runs the simulation for 2 seconds (this is 2 simulated seconds, of course – the actual time needed to run the simulation is dependent on your computer's speed and the complexity of the network):

```
net.network.simulator.run(0,2.0,dt)
```

The three parameters are the start time of the simulation (almost always 0), the end time of the simulation (2.0 seconds in this case), and the size of the time step to use (usually 0.001 seconds).

As an alternative method, you can also run the simulation like this, producing an equivalent result:

```
t=0
while t<=2.0:
    net.network.run(t,t+dt)
    t+=dt
```

With either approach, the simulation will be automatically run when you run the script. No visual display will be given, but the data will be saved to the `experiment.csv` file.

With this approach, you can even run a script without using the Nengo user interface at all. Instead, you can run the model from the command line. Instead of running `nengo` (or `nengo.bat` on Windows), you can do:

```
nengo-cl experiment.py
```

This will run whatever script is in `experiment.py`.

4.4 Running Nengo in Matlab

Since Nengo is a Java application, it can be directly embedded within Matlab. This allows for stimuli and data analysis to be performed within Matlab, which may be more familiar to some users.

4.4.1 Step 1: Set up your Matlab classpath

Matlab needs to know where to find all the java files needed for running Nengo. You do this by adding the names of all the `.jar` files to Matlab's `classpath.txt` file. You can edit this file from within Matlab by typing `edit classpath.txt`.

You need to add a bunch of lines to the end of this file that list all the `.jar` files found in your installation of Nengo and their full path. For example:

```
D:\MatNengo\nengo-1074\nengo-1074.jar
D:\MatNengo\nengo-1074\lib\Blas.jar
D:\MatNengo\nengo-1074\lib\colt.jar
D:\MatNengo\nengo-1074\lib\commons-collections-3.2.jar
D:\MatNengo\nengo-1074\lib\formsrt.jar
D:\MatNengo\nengo-1074\lib\iText-5.0.5.jar
D:\MatNengo\nengo-1074\lib\Jama-1.0.2.jar
D:\MatNengo\nengo-1074\lib\jcommon-1.0.0.jar
D:\MatNengo\nengo-1074\lib\jfreechart-1.0.1.jar
D:\MatNengo\nengo-1074\lib\jmatio.jar
D:\MatNengo\nengo-1074\lib\jung-1.7.6.jar
D:\MatNengo\nengo-1074\lib\jython.jar
D:\MatNengo\nengo-1074\lib\log4j-1.2.14.jar
D:\MatNengo\nengo-1074\lib\piccolo.jar
D:\MatNengo\nengo-1074\lib\piccolox.jar
D:\MatNengo\nengo-1074\lib\qdox-1.6.3.jar
D:\MatNengo\nengo-1074\lib\ssj.jar
D:\Matnengo\nengo-1074\python\jar\jbullet.jar
D:\Matnengo\nengo-1074\python\jar\jpct.jar
D:\Matnengo\nengo-1074\python\jar\vecmath.jar
```

This is technically all that needs to be done to run Nengo inside Matlab. Once you restart Matlab, you should be able to do:

```
import ca.nengo.model.impl.*;
network=NetworkImpl()
network.setName('test')
```

All the basic functionality of Nengo is exposed in this way.

You may also want to increase the memory available to Java. To do this, create a file called *java.opts* in the Matlab startup directory with the following command in it:

```
-Xmx1500m
```

This will give Java a maximum of 1500MB of memory.

4.4.2 Step 2: Connecting Python and Matlab

Since we're used to creating models using the Python scripting system, we want to do the same in Matlab. To set this up, we do the following in Matlab:

```
import org.python.util.*;
python=PythonInterpreter();
python.exec('import sys; sys.path.append("nengo-1074/python")')
```

The path specified on the last line must be your path to the python directory in your Nengo installation.

You can now run the same Python scripts that you can in the Nengo scripting system:

```
python.execfile('addition.py')
```

or, if the file is in another directory:

```
python.execfile('nengo\demo\addition.py')
```

We can also execute single python commands like this:

```
python.exec('net.view(play=0.5)')
```

4.4.3 Step 3: Running a simulation and gathering data

However, what we really want to do is use Matlab to run a simulation, and gather whatever data we need. Here is how we can do that.

First, we need to know how to get access to objects created in the python script. For example, if we made an ensemble called C with `C=net.make('C', 100, 1)` in the script, we can get this object in Matlab as follows:

```
C=javaMethod('__tojava__',python.get('C'), java.lang.Class.forName('java.lang.Object'));
```

Now we can get the value of its origin by something like:

```
C.getOrigin('X').getValues().getValues()
```

With this in mind, the following Matlab code runs a simulation and gathers the output from C's X origin:

```
python.execfile('addition.py');

% get the ensemble 'C'
C=javaMethod('__tojava__',python.get('C'), java.lang.Class.forName('java.lang.Object'));
```

```
% run the simulation
t=0.0;
dt=0.001;
data=[];
w=waitbar(0,'Running simulation...');
while(t<1)
    waitbar(t);
    command=sprintf('net.network.simulator.run(%f,%f,%f)',t,t+dt,dt);
    python.exec(command);
    data=[data C.getOrigin('X').getValues().getValues()];
    t=t+dt;
end
close(w);
```

4.5 Generating Large Ensembles (500 to 5000 neurons)

With the standard Nengo install, creating neural population with more than 500 neurons takes a very long time. This is because Nengo needs to solve for the decoders, which involves (pseudo-)inverting an $N \times N$ matrix, where N is the

number of neurons.

By default, Nengo is a pure Java application, so in order to perform this matrix pseudo-inversion, it uses a Java implementation of Singular Value Decomposition, which is quite slow. To speed this up, we can tell Nengo to call out to another program (Python) to perform this operation.

4.5.1 Step 1: Install Python, SciPy, and NumPy

Python is a freely available programming language, and it has two extensions (SciPy and NumPy) which provide most of the high-speed matrix operations found in Matlab. For Windows and OS X, download it from the following sites:

- Python: <http://www.python.org/getit/>
- SciPy: <http://sourceforge.net/projects/scipy/files/scipy/>
- NumPy: <http://sourceforge.net/projects/numpy/files/NumPy/>

For Linux, it may already be installed, but a command like `sudo apt-get install python python-numpy python-scipy` should install it on many standard Linux distributions (such as Ubuntu).

Important Note: When you install it, use Python 2.7 (or 2.6), rather than 3.2. As it says on the Python web page, “start with Python 2.7; more existing third party software is compatible with Python 2 than Python 3 right now.”

4.5.2 Step 2: Tell Nengo where Python is

For Linux and OS X, Nengo should automatically be able to find Python to run it. However, this seems not to be the case for some versions of Windows. Under Windows, you will have to edit the file `external/pseudoInverse.bat` by changing this:

```
python pseudoInverse %1 %2 %3 %4
```

into this:

```
C:\python27\python.exe pseudoInverse %1 %2 %3 %4
```

Where `C:\python27\` is the directory you installed Python into (this is the default).

4.5.3 Step 3: Testing

You should now be able to create larger ensembles in Nengo. A population with 500 neurons should take ~5-10 seconds. 1000 neurons should take 15-30 seconds. The largest we recommend using this technique is 5000 neurons, which may take up to a half hour. As always, you can make use of the *Speeding up Network Creation (and Making Consistent Models)* option to re-use ensembles once they have been created.

To confirm that Nengo is using this faster approach, you can examine the console output from Nengo. This is the window Nengo is being run from, and on Windows this is a black window showing a lot of text as ensembles are created. This is not the `Script Console` at the bottom of the main Nengo window.

If the external script is working, when you create an ensemble you should see something like this:

```
INFO [Configuring NEFEnsemble:ca.nengo.util.Memory]: Used: 62454552 Total: 101449728 Max: 810942464
INFO [Configuring NEFEnsemble:ca.nengo.util.Memory]: Used: 71208944 Total: 101449728 Max: 810942464
INFO [Configuring NEFEnsemble:ca.nengo.util.Memory]: Used: 69374440 Total: 164933632 Max: 810942464
```

If the external script is not working, when you create an ensemble you should see something like this:

```
INFO [Configuring NEFEnsemble:ca.nengo.util.Memory]: Used: 108627944 Total: 164933632 Max: 810942464
INFO [Configuring NEFEnsemble:ca.nengo.util.Memory]: Used: 108627944 Total: 164933632 Max: 810942464
File not found: java.io.FileNotFoundException: external\matrix_-3645035487712329947.inv (The system c
INFO [Configuring NEFEnsemble:ca.nengo.math.impl.WeightedCostApproximator]: Using 53 singular values
INFO [Configuring NEFEnsemble:ca.nengo.util.Memory]: Used: 75885904 Total: 164933632 Max: 810942464
```

4.6 GPU Computing in Nengo

Since neurons are parallel processors, Nengo can take advantage of the parallelization offered by GPUs to speed up simulations. Currently, only NEF Ensembles and Network Arrays containing NEF Ensembles can benefit from GPU acceleration (and they must be composed solely of leaky integrate-and-fire (LIF) neurons). You can still use the GPU for Networks which contain other types of nodes, but only nodes that meet these criteria will actually be executed on the GPU(s), the rest will run on the CPU. This restriction is necessary because GPUs take advantage of a computing technique called Single-Instruction Multiple-Data, wherein we have many instances of the same code running on different data. If we are only using NEF Ensembles containing LIF neurons then we are well within this paradigm: we want to execute many instances of the code simulating the LIF neurons, but each neuron has different parameters and input. On the other hand, a SimpleNode that you have defined yourself in a Python script cannot run on the GPU because SimpleNodes can contain arbitrary code.

The GPU can also be used to speed up the process of creating NEF Ensembles. The dominant component in creating an NEF Ensemble in terms of runtime is solving for the decoders, which requires performing a Singular Value Decomposition (SVD) on a matrix with dimensions (number of neurons) x (number of neurons). If the number of neurons in the ensemble is large, this can be an extremely computationally expensive operation. You can use CUDA to perform this SVD operation much faster.

The Nengo GPU implementation requires the CUDA developer driver and runtime libraries for your operating system. You may also want to install the CUDA code samples which let you test whether your machine can access and communicate with the GPU and whether the GPU is performing up to standards. Installers for each of these can be downloaded from here: <http://developer.nvidia.com/cuda-toolkit-40>. As a first step, you should download a copy of each of these.

The SVD computation requires a third party GPU linear algebra toolkit called CULA Dense, in addition to the CUDA toolkit. CULA Dense can be downloaded free of charge (though does require a quick registration) from here: <http://culatools.com/downloads/dense/>. Download version R13a as it is the latest release that is compatible with version 4.0 of the CUDA toolkit. CULA is NOT required if you only want to run simulations on the GPU.

Note: Most of the following steps have to be performed for both NengoGPU, the library for running Nengo simulations on the GPU, and NengoUtilsGPU, the library for performing the SVD on the GPU. Here I will detail the process of installing NengoGPU, but the process of installing NengoUtilsGPU is almost identical. I will add comments about how to adjust the process for installing NengoUtilsGPU, and will mark these with *******. The main difference is that NengoUtilsGPU relies on CULA whereas NengoGPU has no such dependency, and this manifests itself in several places throughout the process.

4.6.1 Linux

Step 1: Install CUDA Developer Driver

1. Be sure you have downloaded the CUDA developer driver installer for your system from the link provided above. Note where the file gets downloaded.
2. In a shell, enter the command:

```
sudo gdm service stop
```

This stops the X server. The X server relies on the GPU driver, so the former can't be running while the latter is being changed/updated.

3. Hit `Ctrl-Alt-F1`. This should take you to a login shell. Enter your credentials and then `cd` into the directory where the driver installer was downloaded.
4. To start the installer, run the command:

```
sudo sh <driver-installer-name>
```

5. Answer yes to the queries from the installer, especially the one that asks to change `xorg.conf`.
6. Once installation has finished, restart the X server with:

```
sudo gdm service start
```

Step 2: Install CUDA Toolkit

1. Be sure you have downloaded the CUDA toolkit installer for your system from the link provided above. Note where the file gets downloaded.
2. Run the installer with:

```
sudo sh <toolkit-installer-name>
```

3. The installer will ask where you want to install CUDA. The default location, `/usr/local/cuda`, is the most convenient since parts of the NengoGPU implementation assume it will be there (however, we can easily change these assumptions by changing some text files, so just note where you install it).
4. At the end, the installer gives a message instructing you to set the values of certain environment variables. Be sure to do this. The best way to do it permanently is to set them in your `~/.bashrc` file. For example, to change the `PATH` variable to include the path to the `bin` directory of the `cuda` installation, add the following lines to the end of your `~/.bashrc`:

```
PATH=$PATH:<path-to-cuda-bin-dir>
export PATH
```

and then restart your bash shell. You can type `printenv` to see whether the changes have taken effect.

Step 3: Install CUDA code samples (optional)

This step is not strictly necessary, but can help to ensure that your driver is installed properly and that CUDA code has access to the GPU, and can be useful in troubleshooting various other problems.

1. Be sure you have downloaded the CUDA code sample installer for your system from the link provided above. Note where the file gets downloaded.
2. Run the installer with:

```
sudo sh <samples-installer-name>
```

Your home directory is generally a good place to install it. The installer creates a folder called `NVIDIA_GPU_Computing_SDK` in the location you chose. `NVIDIA_GPU_COMPUTING_SDK/C/src` contains a series of subdirectories, each of which contains CUDA source code which can be compiled into binary files and then executed.

3. `cd` into `NVIDIA_GPU_Computing_SDK/C` and enter the command: `make`. This will compile the many CUDA source code samples in the `C/src` directory, creating a series of executable binaries in `C/bin/linux/release`. Sometimes `make` may fail to compile one of the programs, which will halt the

entire compilation process. Thus, all programs which would have been compiled after the failed program will remain uncompiled. To get around this, you can either fix the compilation issues with the failed program, or you can compile each individual code sample on its own (there are a lot of them, so you probably won't want to compile ALL of them this way, just the ones that seem interesting). Just `cd` into any of the directories under `C/src` and type `make` there. If compilation succeeds, a binary executable file will be created in `C/bin/linux/release`.

4. To run any sample program, `cd` into `C/bin/linux/release` and type `./<name of program>`. If the program in question was compiled properly, you should see a bunch of output about what computations are being performed, as well as either a PASS or a FAIL. FAIL's are bad.
5. Some useful samples are:

`deviceQueryDrv` - Simple test to make sure CUDA programs have access to the GPU. Also displays useful information about the CUDA-enabled GPUs on your system, if it can find and access them.

`bandwidthTest` - Tests bandwidth between CPU and GPU. This bandwidth can sometimes be a bottleneck of the NengoGPU implementation. Online you can usually find bandwidth benchmarks which say roughly what the bandwidth should be for a given card. If your bandwidth is much lower than the benchmark for your card, there may be a problem with your setup.

`simpleMultiGPU` - Useful if your system has multiple GPUs. Tests whether they can all be used together.

*** Step 4: Install CULA Dense (only required if installing NengoUtilsGPU)

This step is very similar to Step 2: `Install CUDA Toolkit`.

1. Be sure you have downloaded the CULA toolkit installer for your system as mentioned in the introduction. Note where the file gets downloaded.
2. Run the installer with:

```
sudo sh <CULA-installer-name>
```
3. The installer will ask where you want to install CULA. Again, the default location `/usr/local/cula`, is the most convenient since parts of the GPU implementation assume it will be there (however, we can easily change these assumptions by changing some text files, so just note where you install it).
4. Be sure to set the environment variables as recommended by the installer. See Step 2: `Install CUDA Toolkit` for the best way to do this.

Step 5: Compiling the shared libraries

1. `cd` into the directory `NengoGPU`. For developers this is can be found in `simulator/src/c/NengoGPU`. For users of the prepackaged version of Nengo, it should just be a subdirectory of the main Nengo folder.
2. Type `./configure` to run the configure script that sets things up.
3. If you installed CUDA in a location other than the default, open the file `Makefile` with your favourite text editor and edit it so that the variables `CUDA_INC_PATH` and `CUDA_LIB_PATH` point to the correct locations. If you installed CUDA in the default location, you don't have to change anything.
4. Type `make` to compile the code in the directory. If successful, this creates a shared library called `libNengoGPU.so`. This is the native library that Nengo will call to perform the neural simulations on the GPU(s).
5. *** Redo steps 1-3 in the `NengoUtilsGPU` directory, which should be located in the same directory as the `NengoGPU` directory. In this case, there are two additional variables in the `Makefile` that you might have to

edit which point to CULA libraries and include files: `CULA_INC_PATH` and `CULA_LIB_PATH`. Again, you only have to edit these if you installed CULA in a location other than the default.

6. We have make sure that the CUDA libraries, which are referenced by `libNengoGPU.so`, can be found at runtime. To acheive this, cd into `/etc/ld.so.conf.d/`. Using your favourite text editor and ensuring you have root privileges, create a text file called `cuda.conf` (eg. `sudo vim cuda.conf`). In this file type the lines:

```
<absolute-path-to-CUDA-dir>/lib/
<absolute-path-to-CUDA-dir>/lib64/
```

So, for example, if you installed CUDA in the default location you should have:

```
/usr/local/cuda/lib/
/usr/local/cuda/lib64/
```

*** If you are installing NengoUtilsGPU as well, then you also have to add the lines:

```
<absolute-path-to-CUDA-dir>/lib/
<absolute-path-to-CUDA-dir>/lib64/
```

Save the file and exit. Finally, run `sudo ldconfig`. This populates the file `/etc/ld.so.cache` using the files in `/etc/ld.so.conf.d/`. `ld.so.cache` tells the machine were to look for shared libraries at runtime (in addition to the default locations like `/usr/lib` and `/usr/local/lib`).

7. This step is only for developers running Nengo through an IDE like Eclipse. Those using a prepackaged version of Nengo can skip this step.

The Java Virtual Machine has to be told where to look for native libraries. Edit the JVM arguments in your Run and Debug configurations so that they contains the following text:


```
-Djava.library.path=<absolute-path-to-NengoGPU-dir>
```

*** If you are also installing NengoUtilsGPU, then you must also add:

```
<absolute-path-to-NengoUtilsGPU-dir>
```

using a colon (:) as the separator between paths.

Step 6: Using NengoGPU and NengoUtilsGPU

1. Now open up Nengo. Click on the  icon on the right side of the tool bar at the top of the Nengo UI. This will open up a menu which lets you configure the parallelization of Nengo.
2. If the previous steps worked properly, then the field `Number of GPUs for Simulation` will be enabled and you will be able to choose the number of GPUs to use for Nengo simulations. This field will not let you choose more GPUs than Nengo can detect on your system. If the `libNengoGPU` library wasn't found (either because building it didn't succeed or Java doesn't know where to find it) or no CUDA enabled GPUs can be detected on your system, then this field will be grayed out and an error message will appear to the right of the field. In this case, revisit the relevant step above.
3. After setting the `Number of GPUs for Simulation` field to a non-zero value, any simulations you run that contain NEF Ensembles should run those NEF Ensembles on the GPU! There should be no change at all in the way you simulate networks except, of course, that they will run faster. You can still set probes, collect spikes, etc, in the same way you did before.
4. NengoGPU provides support for running certain NEF Ensembles on the CPU while the rest are simulated on the GPU(s). Right click on the NEF Ensembles that you want to stay on the CPU and select the `configure` option. Set the `useGPU` field to false, and the ensemble you are configuring will run on the CPU no matter

what. You can also edit the same field on a Network object, and it will force all NEF Ensembles within the Network to run on the CPU.

5. You can also set the number of GPUs to use for simulation in a python script. This is useful if you want to ensure that a given network, created by a script (and maybe even run in that script), always runs with the same number of devices. To achieve this, add the following line to your script:

```
ca.nengo.util.impl.NEFGPUInterface.setRequestedNumDevices(x)
```

where x is the number of devices you want to use for the resulting network.

6. GPU simulations can be combined with CPU multithreading. In the parallelization dialog, it lets you select the number of CPU threads to use. All NEF Ensembles that are set to run on the GPU will run there, and the rest of the nodes in the Network will be parallelized via multithreading. This is especially useful for speeding up Simple Nodes that do a lot of computation. The optimal number of threads will vary greatly depending on the particular network you are running and the specs of your machine, and generally takes some experimentation to get right. However, using a number of threads equivalent to the number of cores on your machine is usually a good place to start.
7. *** If you installed `libNengoUtilsGPU` and it succeeded, then the parallelization dialog will have the `Use GPU for Ensemble Creation` checkbox enabled. If you check the box and press OK, then all NEF Ensembles you create afterwards will use the GPU for the Singular Value Decomposition, and this process should be significantly faster, especially for larger ensembles. If the install failed, Nengo cannot detect a CUDA-enabled GPU, or you simply chose not to install `NengoUtilsGPU`, then the box will be disabled and an error message will appear to its right. Note that the SVD implementation cannot take advantage of multiple GPUs, which is why there is no option to select the number of GPUs for ensemble creation.

4.7 Integrating with IPython Notebook

We are currently working on experimental support for [IPython](http://ipython.org/) (<http://ipython.org/>)'s new Notebook system. The `iPython Notebook` gives a browser-based interface to Python that is similar to the one in Mathematica.

However, to make use of IPython's graphing features, it must be run from Python, not Jython (which is what Nengo uses). To address this, we can make slight modifications to IPython to tell it to run Nengo scripts using Nengo, but everything else in normal Python.

4.7.1 Installing IPython

- Download and install IPython using the instructions at <http://ipython.org/download.html>
- You should now be able to activate the notebook system by running `ipython notebook --pylab`. (The `--pylab` argument is optional, but initializes the `pylab` graphic system so it is recommended). A browser window will appear with the notebook interface.

4.7.2 Configuring IPython Notebook

- We need to customize IPython a bit to tell it about Nengo. Create a configuration profile with the command `ipython profile create`.
- Open the newly-created file `profile_default/ipython_config.py`. It should be off your new IPython directory. On Linux, this is usually `~/.config/ipython`. For other operating systems, see <http://ipython.org/ipython-doc/stable/config/overview.html#configuration-file-location>
- Add the following lines to the bottom of the file:

```
import sys
sys.path.append('python')
```

This will give IPython access to Nengo's python directory. This will let you import things from that directory; most importantly, the `stats` module will be useful for running Nengo models and extracting data from the logs files.

4.7.3 Telling IPython to use Nengo

- We now need to make a small change to the IPython interface. We do this by editing one of the javascript files it uses to create its interface.
- First, find your python install's site-packages directory. This varies by computer, but you can get a list by doing:

```
import sys
for f in sys.path:
    if f.endswith('packages'): print f
```

- Find the IPython directory `IPython/frontend/html/notebook/static/js`
- Edit `notebook.js`. Find the line containing `Notebook.prototype.execute_selected_cell`. About 10 lines below that should be:

```
var code = cell.get_code();
var msg_id = that.kernel.execute(cell.get_code());
```

Change it to:

```
var code = cell.get_code();
if (code.indexOf("nef.Network")!=-1) {
    code='from stats.ipython import run_in_nengo\nrun_in_nengo("""'+code+'""");
}'
var msg_id = that.kernel.execute(code);
```

4.7.4 Running a Nengo model from IPython notebook

- Go to your Nengo directory. You must run IPython from the same directory that Nengo is in!
- Run the IPython interface with `ipython notebook --pylab`
- Create a New Notebook
- Enter the following code to make a simple model:

```
import nef
net=nef.Network('Test')
A=net.make('A',50,1)
input=net.make_input('input',{0: -1, 0.2: -0.5, 0.4:0, 0.6:0.5, 0.8:1.0})
net.connect(input,A)
log=net.log()
log.add('input',tau=0)
log.add('A')
net.run(1.0)
```

- Run the model by pressing Ctrl-Enter. It should think for a bit and then output "finished running experiment-Test.py". It creates this name based on the name of the Network being run.

- As with running a model in Nengo, the logfile should be created as `Test-<date>-<time>.csv` inside your Nengo directory.

4.7.5 Analyzing data from a Nengo run

- As with Nengo, you can use the `stats` module to extract data from a log file. Create a new cell below your current one in the Notebook and enter the following:

```
import stats
data=stats.Reader()
plot(data.time,data.input)
plot(data.time,data.A)
```

4.7.6 Running a model with different parameter settings

- First we need to identify the parameters we might want to vary in the Nengo model. Do this by defining them as variables at the top of the code. For example:

```
# here are my parameters
N=50
```

```
import nef
net=nef.Network('Test')
A=net.make('A',N,1)
input=net.make_input('input',{0: -1, 0.2: -0.5, 0.4:0, 0.6:0.5, 0.8:1.0})
net.connect(input,A)
log=net.log()
log.add('input',tau=0)
log.add('A')
net.run(1.0)
```

The parameters should be defined before any import statements, but can be after any comments at the top of the file.

- Re-run the model (Ctrl-Enter).
- Make a new cell with the following code:

```
import stats
stats.run('experiment-Test',3,N=[5,10,20,50])
```

The model will be run 3 times at each parameter setting (N=5, N=10, N=20, and N=50), for a total of 12 runs. Each parameter combination has its own directory inside the `experiment-Test` directory.

4.7.7 Plotting data from varying parameter settings

- We use the `stats.Stats` class to access the data from simulation runs. For example, to get the data from the runs where N=5, we can do:

```
import stats
s=stats.Stats('experiment-Test')

data=s.data(N=5)

for i in range(len(data)):
    plot(data.time[:,i],data.A[:,i])
```


- We can use a similar approach to plot the average activity for varying values of N:

```
import stats
s=stats.Stats('experiment-Test')

for N in [5,10,20,50]:
    data=s.data(N=N)

    plot(data.time[:,0],numpy.mean(data.A,axis=1),label='%d'%N)
legend(loc='best')
```

4.7.8 Computing summary data

- We often want to look at data that's more high-level than the raw time-varying output. For example, we might want to determine the representation accuracy of the model. We can do this by writing a function that does our analysis. It should expect 2 inputs: a Reader object and a dictionary holding any other computed results.
- This particular example computes the mean-squared error between the input and A values within 5 different 100ms time windows:

```
import stats
s=stats.Stats('experiment-Test')

def error(data,computed):
    errors=[]
    for t in [0.1,0.3,0.5,0.7,0.9]:
        A=data.get('A',(t,t+0.1))
        ideal=data.get('input',(t,t+0.1))
        errors.append(sqrt(mean((ideal-A)**2)))
    return mean(errors)

s.compute(error=error)
```

- We can now see how the error changes as N varies by doing:

```
import stats
s=stats.Stats('experiment-Test')

N=[5,10,20,50]
plot(N,[mean(s.computed(N=n).error) for n in N])
```


JAVA API DOCUMENTATION

The Java API documentation is not a part of the User Manual, but can be found here: <http://www.nengo.ca/javadoc/>

- *genindex*
- *search*