Nengo Tutorial: Multiplication

This tutorial covers building a neural model capable of performing multiplication. We start with two neural groups representing the two values to be multiplied, create a new neural group which stores both values, and then produce a final neural group representing the product of the original values.

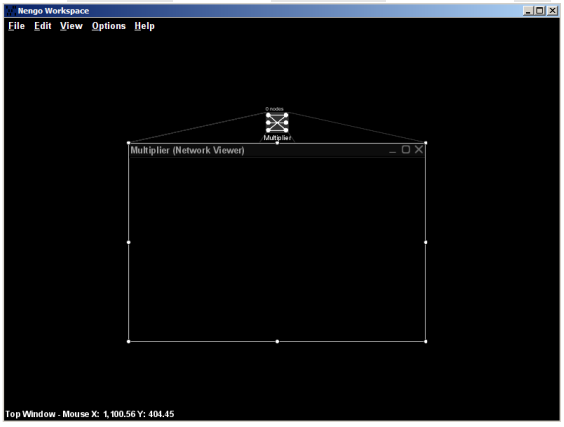
The following techniques are covered by this tutorial:

- Defining neural groups (ensembles)
- Defining external inputs to a network
- Combining values from two neural groups into one
- Non-linear transformations
- Recording data from the network and plotting it
- Using the scripting interface

Step 1: Create a Network

When Nengo is run, we are presented with an empty workspace. The first thing to do is to create a Network to contain all of the neural ensembles we will be creating.

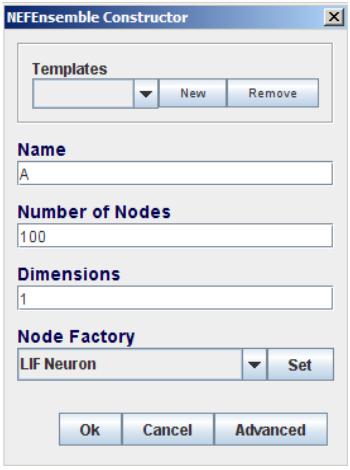
To create the Network, go to File->New->Network in the top menu. A dialog box will appear that allows you to enter a name for the Network. Enter Multiplier and click Ok.



Step 2: Represent the Input

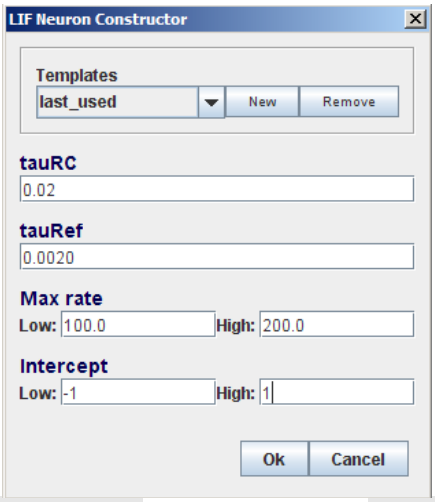
We now need two neural populations to represent the two values we are multiplying. These will be standard Leaky-Integrate-and-Fire (LIF) neural ensembles, with 100 neurons per ensemble. Each population will represent a single 1-dimensional value, since we are just multiplying scalar values in this tutorial.

Right-click inside the Multiplier Network and select Create New->NEFEnsemble. Set the name to A, the number of nodes to 100, and the Dimensions to 1.

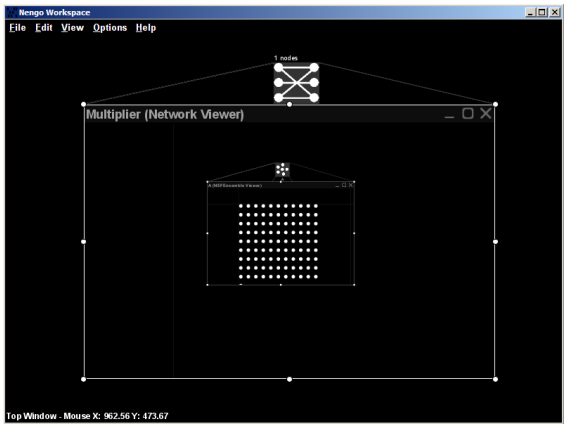


Before clicking Ok, we need to specify the neural properties. This is done via the Node Factory. From the drop-down menu, choose LIF Neuron. Now click Set to set its properties.

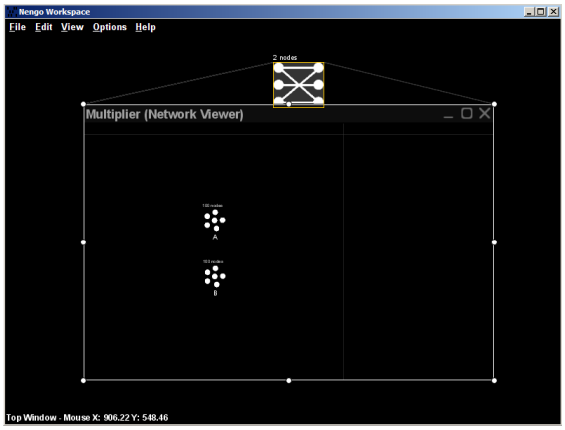
The new dialog box allows you to configure tauRC (the neurons' membrane time constant), tauRef (the spike refractory period), the range of maximum firing rates across the population, and the range of intercepts across the population. For this tutorial, we will use standard values. Set tauRC to 0.02 (20 milliseconds) and tauRef to 0.002 (2 milliseconds). By setting the low max rate to 100 and the high max rate to 200, we specify that the neurons in this neural ensemble will have their maximum firing rates chosen from a uniform distribution between 100Hz and 200Hz. Since we want these neurons to be equally accurate across the full range of the represented varable, we set the intercepts to be between -1 and 1 (the intercept is the represented value that is just large enough for the neuron to start to respond).



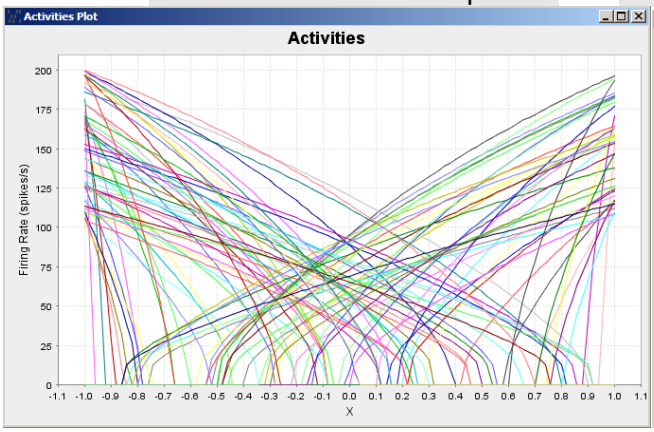
Now press Ok for the LIF Neuron Constructor and again for the NEF Ensemble Constructor. The neural population will now be created.



Repeat this process to create a second neural ensemble named B. Note that you can close the NEFEnsemble Viewers, as we do not need to examine the individual neurons.



Given these neural ensembles, we can produce two useful plots. Right click on one of the ensembles and choose Plot->Constant Rate Responses and Plot->Distortion: X.



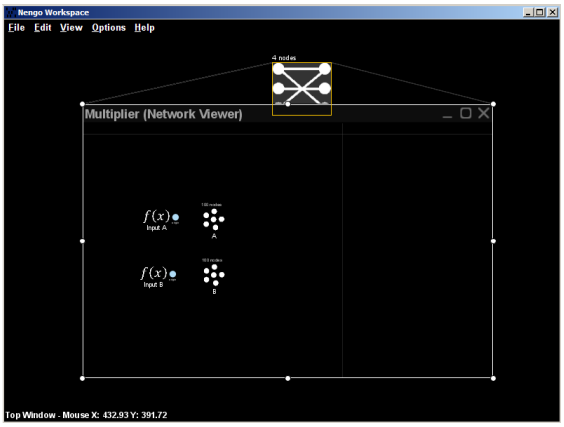
These graphs allow us to confirm that the neurons have the expected tuning curves and are capable of representing values accurately.

Step 3: Provide Input

In order for our multiplier to do anything, we will need to be able to specify the input: the two values to be multiplied. Since neural models are inherently temporal models, these inputs can change over time,

and so are treated by Nengo as functions of time.

Create a function by right-clicking on the Network and choosing Create New->Function Input. Give it a name of Input A and set its Dimensions to 1. Now click on Set Functions, which allows us to specify the function that will be used. Choose Constant Function from the list (to indicate that we want a function that stays at the same value over time) and click Set. We can now specify the value that we want as the input. Since our neurons are representing a range between -1 and 1, we choose a value within that range, such as 0.8. Now click Ok for each dialog box and the function will be created. Repeat this process to create a second input (Input B) with a constant value of 0.5.

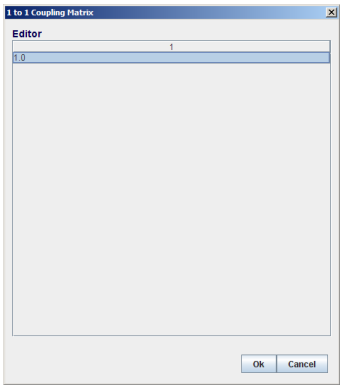


Step 4: Connect Inputs to Neural Ensembles

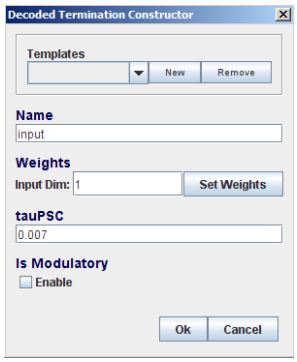
We now need to feed the inputs into the corresponding neural ensembles. In Nengo, this is done through Origins and Terminations. The two functions already have origins defined (the blue circles on their right). We need to add terminations to the two neural ensembles. These will be simple terminations that do not transform the represented value in any way.

Right-click on NEFEnsemble A and choose Add decoded termination. A dialog box will appear allowing us to configure the termination. Set its name to input and its input dimensions to 1.

Now click on Set Weights to specify the coupling matrix for this termination. This can be any matrix, allowing for any linear transformation to occur. In this case, since we have a single dimension at the origin and a single dimension at the termination, we have a 1x1 matrix. Since we do not want to transform the value in any way, we want the 1x1 identity matrix, which is simply the single number 1. Change the coupling matrix so that it just contains the number 1 and then click Ok.

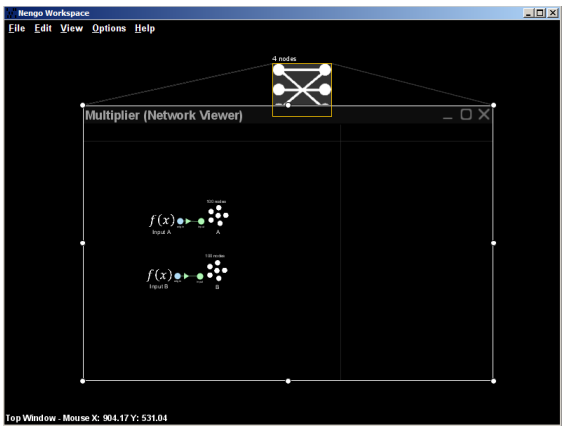


The Decoded Termination Constructor should now reappear. The final value to set is tauPSC, the post synaptic time constant. Set it to a value of 0.007 (7 milliseconds), since this is what we will be using later in the tutorial.



A green circle indicating the new termination will appear to the left of NEFEnsemble A. Connect the origin from Input A to the new termination by clicking on the origin, holding the mouse button down, and dragging to the new termination. When the mouse button is released, the connection will be made.

Repeat the above process to create a termination for NEFEnsemble B and connect it to input B.

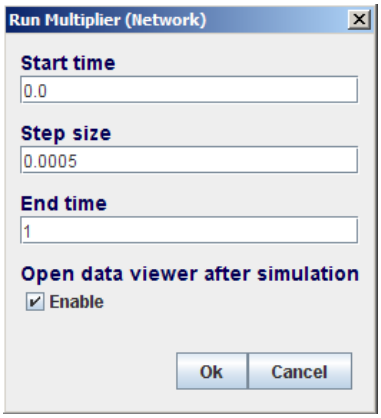


Step 5: Run the Simulation

To confirm that our system is working as expected, we can run the simulation as it stands. This should result in neural population A encoding a value of 0.8 and neural population B encoding a value in 0.5.

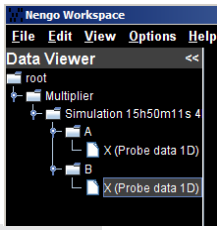
The first thing we need to do is to add two probes to the model. These will record the values encoded in the spiking activity of neural ensembles A and B. Right-click on A and select Add probe->X - Function of NEFEnsemble state. (X refers to the value being represented by any given NEF population). Do the same for B.

Right-click on the Network and select Run Multiplier. A dialog box will appear where we can set the timing for this simulation. Set the Start time to 0, the End time to 1, and the Step size to 0.0005 (0.5 milliseconds). Since we will want to examine the data after running the simulation, make sure Open data viewer after simulation is enabled.

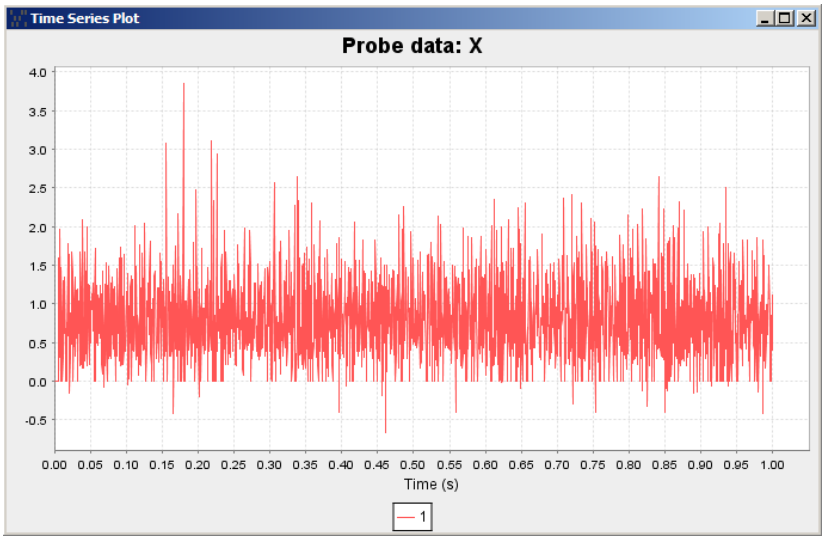


Now click Ok to run the simulation.

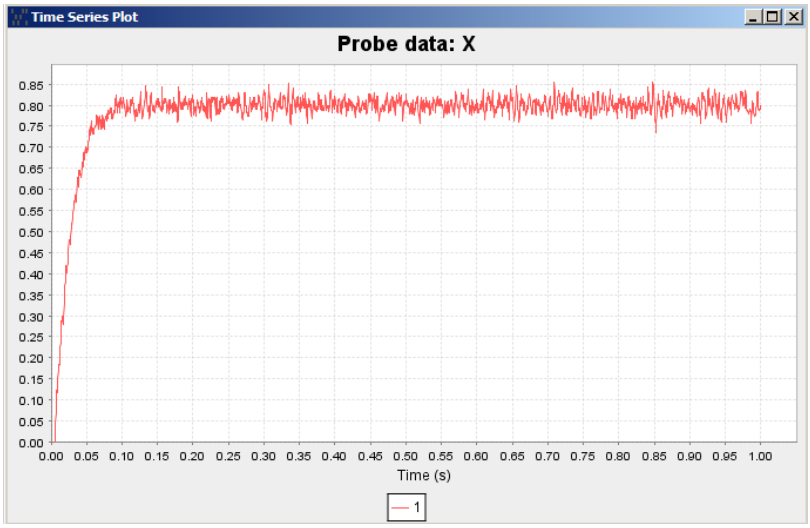
After it has run, a Data Viewer will appear on the left side of the screen (if it does not, select View->Toggle Data Viewer from the top menu, or press Ctrl-2). Click on Multiplier and expand the tree to find the data from the two probes.



If we double-click on one of the X (Probe data 1D) lines, we will get the raw data of what is being represented by that neural ensemble over the simulation. Since this is a spiking neuron model, this data is actually just a weighted sum of spike impulses. Since we are only using 100 neurons, this will be a noisy value, but should average around the expected data (0.8 for A, 0.5 for B).



To get a more useful graph, we can apply an averaging filter to the plot. Do this by right-clicking on the line in the Data Viewer and choosing Plot with options. Set the time constant to 0.02 and the subsampling to 0.

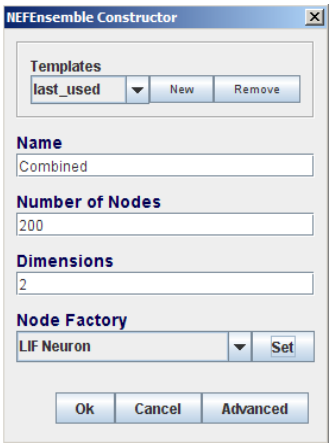


The graph indicates that the neural ensemble is accurately representing the value 0.8. Note that the gradual increase from 0 to 0.8 at the beginning of the graph is an artifact of the filtering process. The graph for the other neural ensemble should look similar, but with a value around 0.5.

Step 6: Combine the Inputs

Since multiplication is a non-linear operation, the first step in performing it is to form a new neural population that represents both of the values. This combined representation allows an accurate approximation of multiplication to be derived.

We thus create a new neural ensemble comprising 200 neurons and having 2 dimensions. Follow the same process as in Step 2, using the same neural parameters. Call the new NEFEnsemble Combined.

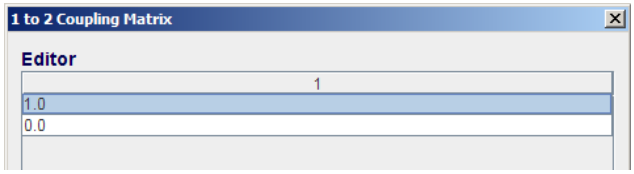


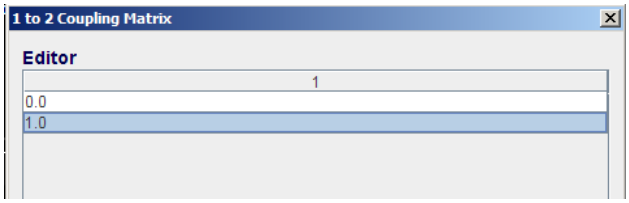
Now we need to connect A and B to this new neural population. We want the value from A to be stored in the first dimension and the value from B to be stored in the second dimension. To do this, we use the coupling matrix, as mentioned in Step 4, where it was a one-dimensional identity matrix.

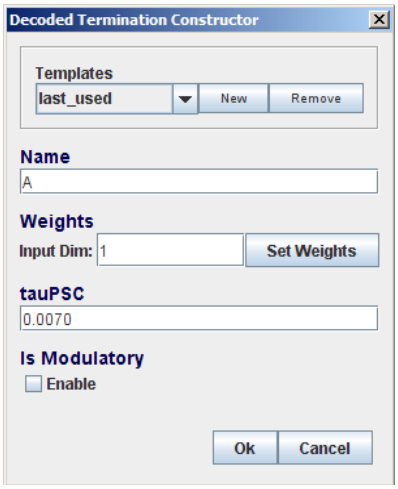
For the connection from A to the Combined population, we use the coupling matrix [1,0], and for the connection from B we use the coupling matrix [0,1]. We can see how this works by imagining the 1-dimensional inputs being multiplied by these coupling matrices before they are stored in the combined population:

[A]*[1,0]+[B]*[0,1]
= [A,0]+[0,B]
= [A,B]

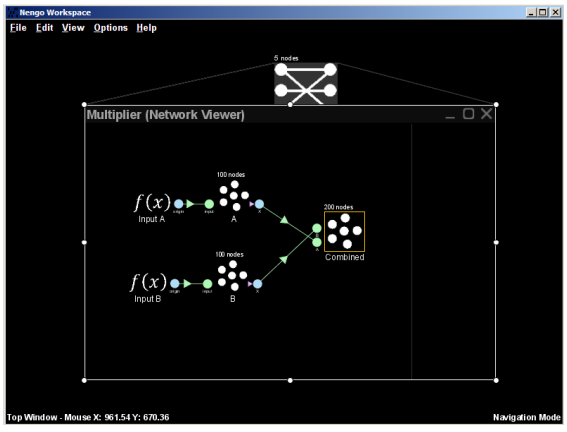
To create these connections, we need two new terminations on the Combined population. Follow the same steps as in Step 4, but call one of them A and the other one B. Make sure the input dimension is 1 and that you set the weights of A to be [1,0] and B to be [0,1].







Now that the terminations exist, connect the origins from A and B to the new terminations. If the origins from A and B are not visible, right click on them and select Origins and Terminations->Show origin->X.

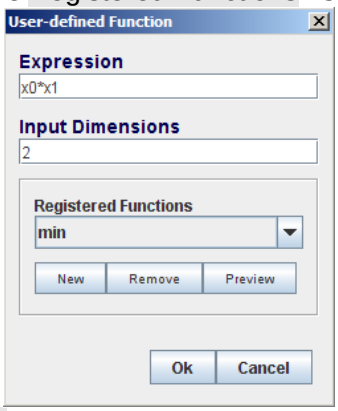


Step 7: Perform the Multiplication

Now that we have both values represented in a single population, we can multiply them. We do this by creating a new decoded origin, which we can specify should extract the product of the two values. As per NEF theory, this is done using exactly the same method as we have been using to extract X, which

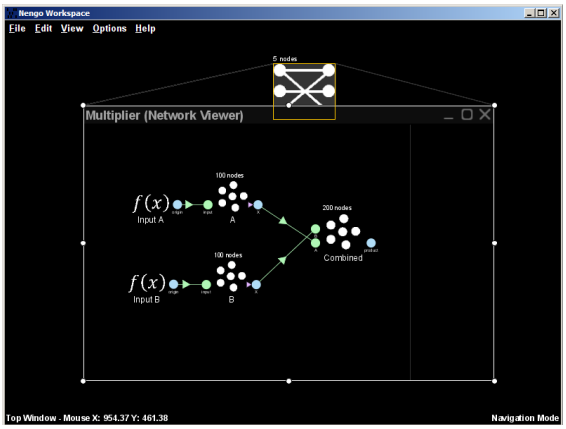
is just the basic value being represented.

We do this by creating a new origin for the Combined population. Right-click on it and choose Add decoded origin. Set its name to product and the number of output dimensions to 1. Then click on Set Functions to define the function that should be applied. Choose User-defined Function from the list and click Set to define it. We can now specify a numerical expression that defines the function we want to calculate. Type x0*x1 into the Expression box. This will multiply the two values in the represented vector together. Note that you can define any function you like here, and can make use of the standard math functions listed in the Registered Functions list.



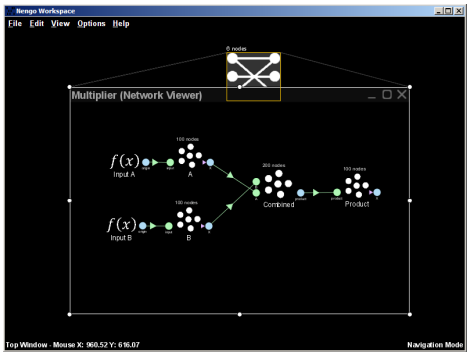
Click Ok to define the function and Ok again to set it.

IMPORTANT NOTE: Before clicking Ok to complete the creation of the decoded origin, make sure that the Node Origin Name is set to AXON, not current. We want to decode based on the spike trains produced by the neurons, not the somatic current.

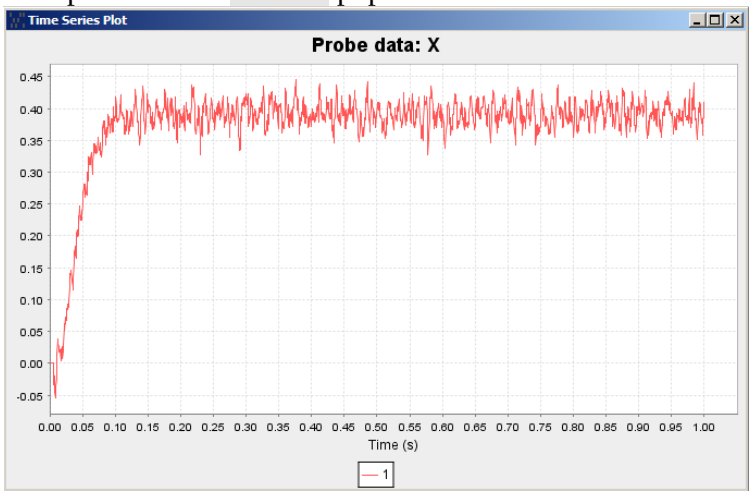


Step 8: Store the Product

Finally, we can create a new NEFEnsemble to represent the result of the multiplication. Create this exactly as A and B were created, using 100 neurons, 1 dimension, and the name Product. Add a decoded termination (with a coupling matrix that is just the number 1, as no transformation is needed). Connect it to the product origin of the Combined population and add a probe so that we can observe its value.



If we now run the simulation as in Step 5, multiplication should be performed. We can observe this by graphing the data from the probe on the Product population.



As can be seen, the network has correctly calculated that 0.8 times 0.5 is 0.4.

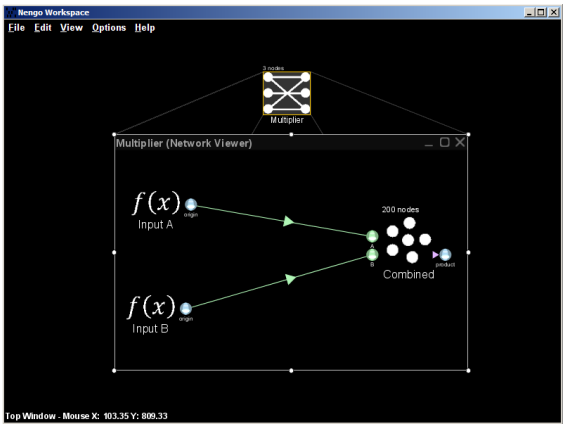
Step 9: Simplify the Model

The model we have just built performs multiplication successfully. However, it turns out that we can simplify this model considerably. Three of the neural ensembles we have defined (A, B, and Product) merely store a value. The Combined neural ensemble is where the multiplication actually occurs.

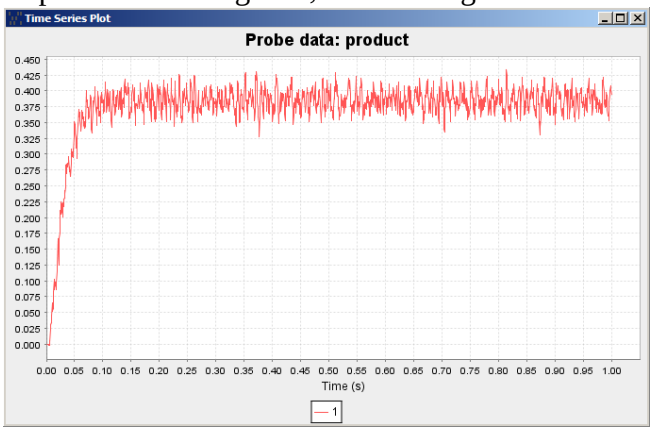
In a larger, more complex model (where more than just multiplication occurs), these three storage neural ensembles may be needed. However, for the purposes of this model, we can actually remove them.

Right-click on A and choose Remove model. Answer Yes to the "Are you sure?" question. This will delete the neural ensemble. Do the same to B and Product. Now directly connect the origin of Input A to the A termination on the Combined NEFEnsemble. Do the same for Input B (to the B termination). This is now a (much simpler) multiplier network.

To view the result of the multiplication, we need a new probe. Right-click on the Combined neural ensemble and select Add probe->product.



If we run the simulation and plot the resulting data, we should get a similar result as before.

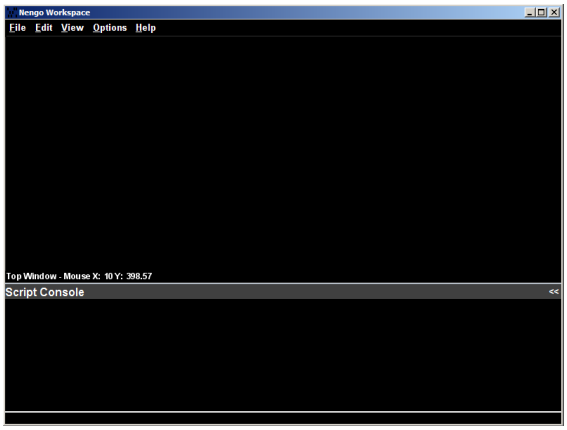


Since the data passes through fewer neural ensembles, the result should be slightly more accurate than before. When creating any sort of Nengo model, it is often useful to look through it for neural ensembles that can be removed in this way. In general, neural ensembles that only store a value and do not themselves transform it are good candidates for removal.

Step 10: Use the Scripting Interface

Instead of doing all of the above steps through the graphical user interface, we can also create models using a more traditional coding approach. Nengo is written in Java, so models can be created by directly using the Java API. However, since it is useful to be able to combine coding with the graphical interface, a scripting language interface has been added to Nengo. This uses the Python syntax and provides interaction with the Java objects in the model thanks to the Jython project. The practical result is that you can modify and inspect from the scripting console a model that you have created and run using the graphical user interface, and vice versa.

For now, we are going to use the Python interface to build our multiplier model. Start with a clean Nengo environment by either restarting Nengo or deleting any network you have built. Open the script console by pressing Ctrl-P.



We can now type into the bottom line in the script console. In theory, we could simply type a complete program in, line-by-line, executing it as we go. This is good for doing quick things, but often we will want to save our program as a script and run it all at once. We can do this by creating a file with any text editor, or using the built in script editor available by pressing Ctrl-E. The file should be saved in the main Nengo directory (on Windows, this is `C:\Program Files\CTN\Nengo` by default). Scripts are run by typing `run filename.py` into the script console.

Here is the full program:

```
# Get access to the Nengo objects
from ca.nengo.model import *
from ca.nengo.model.impl import *
from ca.nengo.model.nef.impl import *
from ca.nengo.math.impl import *


# Create a network and add it to the world
network=NetworkImpl()
network.name="Multiplier"
world.add(network)


# Configure the neuron parameters
ef=NEFEnsembleFactoryImpl()
ef.nodeFactory.tauRC=0.02
ef.nodeFactory.tauRef=0.002
ef.nodeFactory.maxRate=IndicatorPDF(100,200)
ef.nodeFactory.intercept=IndicatorPDF(-1,1)


# Create input neural ensembles
a=ef.make("A",100,1)
network.addNode(a)

b=ef.make("B",100,1)
network.addNode(b)


# Provide the input
inputA=FunctionInput("Input A", [ConstantFunction(1,0.8)], Units.UNK)
inputB=FunctionInput("Input B", [ConstantFunction(1,0.5)], Units.UNK)
network.addNode(inputA)
network.addNode(inputB)


# Connect input to neural ensembles
a.addDecodedTermination("input",[[1.0]],0.007,False)
network.addProjection(inputA.getOrigin('origin'),a.getTermination("input"))

b.addDecodedTermination("input",[[1.0]],0.007,False)
network.addProjection(inputB.getOrigin('origin'),b.getTermination("input"))
```

```
# Combine inputs
combined=ef.make("Combined",200,2)
network.addNode(combined)

combined.addDecodedTermination("A",[[1.0],[0.0]],0.007,False)
network.addProjection(a.getOrigin('X'),combined.getTermination("A"))

combined.addDecodedTermination("B",[[0.0],[1.0]],0.007,False)
network.addProjection(b.getOrigin('X'),combined.getTermination("B"))


# Perform the multiplication
interpreter=DefaultFunctionInterpreter()
combined.addDecodedOrigin("product",[interpreter.parse("x0*x1",2)],'AXON')

product=ef.make("Product",100,1)
network.addNode(product)
product.addDecodedTermination("product",[[1.0]],0.007,False)
network.addProjection(combined.getOrigin('product'),product.getTermination('product'))


# Record the results
network.simulator.addProbe("A",a,"X",True)
network.simulator.addProbe("B",b,"X",True)
network.simulator.addProbe("Product",product,"X",True)
```

The first section adds the important classes from the Nengo API into the local namespace. This makes it easier to access them in our code.

```
# Get access to the Nengo objects
from ca.nengo.model import *
from ca.nengo.model.impl import *
from ca.nengo.model.nef.impl import *
from ca.nengo.math.impl import *
```

An alternative approach would be to do this instead:

```
import ca.nengo
```

However, this approach would mean we would have to use the full name of the Nengo objects, so instead of:

```
network=NetworkImpl()
```

we would have to write:

```
network=ca.nengo.model.impl.NetworkImpl()
```

Next, we create the network.

```
# Create a network and add it to the world
network=NetworkImpl()
network.name="Multiplier"
world.add(network)
```

One special trick occurring here is the introduction of the `world` object. This refers to the whole content of the user interface, and allows the network to appear in it.

Now we configure the neuron parameters. Since we will often want to use the same sorts of neurons throughout a simulation, we can configure them once with a Factory that will allow us to produce neurons and neural ensembles with the desired properties.

```
# Configure the neuron parameters
ef=NEFEnsembleFactoryImpl()
ef.nodeFactory.tauRC=0.02
ef.nodeFactory.tauRef=0.002
```

```
ef.nodeFactory.maxRate=IndicatorPDF(100,200)
ef.nodeFactory.intercept=IndicatorPDF(-1,1)
```

Expert users may note that we are making use of a simplifying feature of Jython: it is aware of JavaBean properties. That is, we could have written the above code in the following way, which will be more familiar to Java programmers

```
ef.getNodeFactory().setTauRC(0.02)
```

The Jython interface automatically translates the direct code to call the correct `get` and `set` methods.

Next, we create the neural ensembles by using the NEFEnsembleFactory.

```
# Create input neural ensembles
a=ef.make("A",100,1)
network.addNode(a)

b=ef.make("B",100,1)
network.addNode(b)
```

The parameters to the `make` command are the name, the number of neurons, and the number of dimensions. We have to be sure to add the newly created node to the network so that it becomes part of our simulation.

To create the Function Inputs, we do the following.

```
# Provide the input
inputA=FunctionInput("Input A", [ConstantFunction(1,0.8)], Units.UNK)
inputB=FunctionInput("Input B", [ConstantFunction(1,0.5)], Units.UNK)
network.addNode(inputA)
network.addNode(inputB)
```

Notice that we can specify an array of functions, which would be treated as multiple dimensions. The unit specification at the end indicates that we are not treating this value as being in any particular units (seconds, Newtons, millivolts, etc.).

To connect the inputs to the relevant neural ensemble, we need to create a termination and form the projection between the two.

```
# Connect input to neural ensembles
a.addDecodedTermination("input",[[1.0]],0.007,False)
network.addProjection(inputA.getOrigin('origin'),a.getTermination("input"))

b.addDecodedTermination("input",[[1.0]],0.007,False)
network.addProjection(inputB.getOrigin('origin'),b.getTermination("input"))
```

When creating a termination, we specify the coupling matrix, the post-synaptic time constant, and whether this is just a modulatory input (used for advanced models where the connection affects some other property of the neuron without directly affecting its current).

We follow a similar approach to create the Combined neural ensemble.

```
# Combine inputs
combined=ef.make("Combined",200,2)
network.addNode(combined)

combined.addDecodedTermination("A",[[1.0],[0.0]],0.007,False)
network.addProjection(a.getOrigin('X'),combined.getTermination("A"))

combined.addDecodedTermination("B",[[0.0],[1.0]],0.007,False)
network.addProjection(b.getOrigin('X'),combined.getTermination("B"))
```

Notice that we have changed the number of neurons, the dimensionality, and the coupling matrices.

To perform the multiplication, we need to create an origin and specify its transformation function. We can do this by creating a new Python object (i.e. by subclassing Function), allowing us to define any possible function. However, for this case, we will make use of the same user-defined function system used above.

```
# Perform the multiplication
interpreter=DefaultFunctionInterpreter()
combined.addDecodedOrigin("product",[interpreter.parse("x0*x1",2)],'AXON')

product=ef.make("Product",100,1)
network.addNode(product)
product.addDecodedTermination("product",[[1.0]],0.007,False)
network.addProjection(combined.getOrigin('product'),product.getTermination('product'))
```

Finally, we can add probes to our model so we can observe its behaviour

```
# Record the results
network.simulator.addProbe("A",a,"X",True)
network.simulator.addProbe("B",b,"X",True)
network.simulator.addProbe("Product",product,"X",True)
```

The result should be a model identical to the one created using the graphical approach.