```
# A Minimal Example of the Neural Engineering Framework
#
# The NEF is a method for building large-scale neural models using realistic
#  neurons.  It is a neural compiler: you specify the high-level computations
#  the model needs to compute, and the properties of the neurons themselves,
#  and the NEF determines the neural connections needed to perform those
#  operations.
#
# The standard software for building NEF models is Nengo (http://nengo.ca).
#  Nengo is a cross-platform Java application that provides both a drag-and-
   drop
#  graphical user environment and a Python scripting interface for
#  creating these neural models.  It has been used to model a wide variety of
#  behaviour, including motor control, visual attention, serial recall, action
#  selection, working memory, attractor networks, inductive reasoning, path
#  integration, and planning with problem solving.
#
# However, given the complexity of Nengo, and due to the fact that this is a
#  fairly non-standard approach to neural modelling, we feel it is also useful
#  to have a simple example that shows exactly how the NEF works, from
#  beginning to end.  That is the goal of this script.
#
# This script shows how to build a simple feed-forward network of leaky
#  integrate-and-fire neurons where each population encodes a one-dimensional
#  value and the connection weights between the populations are optimized to
#  compute some arbitrary function.  This same approach is used in Nengo,
#  extended to multi-dimensional representation, multiple populations of
#  neurons, and recurrent connections.
#
# To change the input to the system, change 'input'
# To change the function computed by the weights, change 'function'
#
# The size of the populations and their neural properties can also be adjusted
#  by changing the parameters below.
#
# This script requires Python (http://www.python.org/) and Numpy
#  (http://numpy.scipy.org/) to run, and Matplotlib (http://matplotlib.org/)
   to
#  produce the output graphs.
#
# For more information on the Neural Engineering Framework and the Nengo
#  software, please see http://nengo.ca
```

```
import random
import math

#################################################
# Parameters
#################################################

dt = 0.001       # simulation time step
t_rc = 0.02      # membrane RC time constant
t_ref = 0.002    # refractory period
t_pstc = 0.1     # post-synaptic time constant
N_A = 50         # number of neurons in first population
N_B = 40         # number of neurons in second population
N_samples = 100  # number of sample points to use when finding decoders
rate_A = 25, 75  # range of maximum firing rates for population A
rate_B = 50, 100 # range of maximum firing rates for population B

# the input to the system over time
def input(t):
    return math.sin(t)

# the function to compute between A and B
def function(x):
    return x*x


#################################################
# Step 1: Initialization
#################################################


# create random encoders for the two populations
encoder_A = [random.choice([-1,1]) for i in range(N_A)]
encoder_B = [random.choice([-1,1]) for i in range(N_B)]

def generate_gain_and_bias(count, intercept_low, intercept_high, rate_low,
    rate_high):
    gain = []
    bias = []
    for i in range(count):
        # desired intercept (x value for which the neuron starts firing
        intercept = random.uniform(intercept_low, intercept_high)
        # desired maximum rate (firing rate when x is maximum)
        rate = random.uniform(rate_low, rate_high)

        # this algorithm is specific to LIF neurons, but should
        #  generate gain and bias values to produce the desired
        #  intercept and rate
        z = 1.0 / (1-math.exp((t_ref-(1.0/rate))/t_rc))
        g = (1 - z)/(intercept - 1.0)
        b = 1 - g*intercept
        gain.append(g)
        bias.append(b)
    return gain,bias

# random gain and bias for the two populations
gain_A, bias_A = generate_gain_and_bias(N_A, -1, 1, rate_A[0], rate_A[1])
gain_B, bias_B = generate_gain_and_bias(N_B, -1, 1, rate_B[0], rate_B[1])
```

```
# a simple leaky integrate-and-fire model, scaled so that v=0 is resting
#  voltage and v=1 is the firing threshold
def run_neurons(input,v,ref):
    spikes=[]
    for i in range(len(v)):
        dV = dt * (input[i]-v[i]) / t_rc    # the LIF voltage change equation
        v[i] += dV
        if v[i]<0: v[i]=0                   # don't allow voltage to go below
            0

        if ref[i]>0:                        # if we are in our refractory
            period
            v[i]=0                          #   keep voltage at zero and
            ref[i]-=dt                      #   decrease the refractory period

        if v[i]>1:                          # if we have hit threshold
            spikes.append(True)             #   spike
            v[i] = 0                        #   reset the voltage
            ref[i] = t_ref                  #   and set the refractory period
        else:
            spikes.append(False)
    return spikes

# measure the spike rate of a whole population for a given represented value x
def compute_response(x, encoder, gain, bias, time_limit=0.5):
    N = len(encoder)   # number of neurons
    v = [0]*N          # voltage
    ref = [0]*N        # refractory period

    # compute input corresponding to x
    input = []
    for i in range(N):
        input.append(x*encoder[i]*gain[i]+bias[i])
        v[i]=random.uniform(0,1)  # randomize the initial voltage level

    count = [0]*N    # spike count for each neuron

    # feed the input into the population for a given amount of time
    t = 0
    while t<time_limit:
        spikes=run_neurons(input, v, ref)
        for i,s in enumerate(spikes):
            if s: count[i]+=1
        t += dt
    return [c/time_limit for c in count]  # return the spike rate (in Hz)

# compute the tuning curves for a population
def compute_tuning_curves(encoder, gain, bias):
    # generate a set of x values to sample at
    x_values=[i*2.0/N_samples - 1.0 for i in range(N_samples)]

    # build up a matrix of neural responses to each input (i.e. tuning curves)
    A=[]
    for x in x_values:
        response=compute_response(x, encoder, gain, bias)
        A.append(response)
    return x_values, A

# compute decoders
```

```
import numpy
def compute_decoder(encoder, gain, bias, function=lambda x:x):
    # get the tuning curves
    x_values,A = compute_tuning_curves(encoder, gain, bias)

    # get the desired decoded value for each sample point
    value=numpy.array([[function(x)] for x in x_values])

    # find the optimum linear decoder
    A=numpy.array(A).T
    Gamma=numpy.dot(A, A.T)
    Upsilon=numpy.dot(A, value)
    Ginv=numpy.linalg.pinv(Gamma)
    decoder=numpy.dot(Ginv,Upsilon)/dt
    return decoder

# find the decoders for A and B
decoder_A=compute_decoder(encoder_A, gain_A, bias_A, function=function)
decoder_B=compute_decoder(encoder_B, gain_B, bias_B)

# compute the weight matrix
weights=numpy.dot(decoder_A, [encoder_B])


#################################################
# Step 2: Running the simulation
#################################################

v_A = [0.0]*N_A       # voltage for population A
ref_A = [0.0]*N_A     # refractory period for population A
input_A = [0.0]*N_A   # input for population A

v_B = [0.0]*N_B       # voltage for population B
ref_B = [0.0]*N_B     # refractory period for population B
input_B = [0.0]*N_B   # input for population B

# scaling factor for the post-synaptic filter
pstc_scale=1.0-math.exp(-dt/t_pstc)


# for storing simulation data to plot afterward
inputs=[]
times=[]
outputs=[]
ideal=[]

output=0.0            # the decoded output value from population B
t=0
while t<10.0:
    # call the input function to determine the input value
    x=input(t)

    # convert the input value into an input for each neuron
    for i in range(N_A):
        input_A[i]=x*encoder_A[i]*gain_A[i]+bias_A[i]

    # run population A and determine which neurons spike
    spikes_A=run_neurons(input_A, v_A, ref_A)
```

```
        # decay all of the inputs (implementing the post-synaptic filter)
        for j in range(N_B):
            input_B[j]*=(1.0-pstc_scale)
        # for each neuron that spikes, increase the input current
        #  of all the neurons it is connected to by the synaptic
        #  connection weight
        for i,s in enumerate(spikes_A):
            if s:
                for j in range(N_B):
                    input_B[j]+=weights[i][j]*pstc_scale

        # compute the total input into each neuron in population B
        #  (taking into account gain and bias)
        total_B=[0]*N_B
        for j in range(N_B):
            total_B[j]=gain_B[j]*input_B[j]+bias_B[j]

        # run population B and determine which neurons spike
        spikes_B=run_neurons(total_B, v_B, ref_B)

        # for each neuron in B that spikes, update our decoded value
        #  (also applying the same post-synaptic filter)
        output*=(1.0-pstc_scale)
        for j,s in enumerate(spikes_B):
            if s:
                output+=decoder_B[j][0]*pstc_scale

        print t, output
        times.append(t)
        inputs.append(x)
        outputs.append(output)
        ideal.append(function(x))
        t+=dt


    #################################################
    # Step 3: Plot the results
    #################################################

    x,A = compute_tuning_curves(encoder_A, gain_A, bias_A)
    x,B = compute_tuning_curves(encoder_B, gain_B, bias_B)

    import pylab
    pylab.figure()
    pylab.plot(x, A)
    pylab.title('Tuning curves for population A')

    pylab.figure()
    pylab.plot(x, B)
    pylab.title('Tuning curves for population B')

    pylab.figure()
    pylab.plot(times, inputs, label='input')
    pylab.plot(times, ideal, label='ideal')
    pylab.plot(times, outputs, label='output')
    pylab.title('Simulation results')
    pylab.legend()
    pylab.show()
```