Week 1 Assignment: Neuron Models

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
```

Helper functions

In [2]:

```
def PlotSpikeRaster(st, y_range=[0, 1.]):
    '''
    PlotSpikeRaster(spiketimes, y_range=[0, 1.])

    Plots a spike raster plot for a list of arrays of spike times.

    Input:
      spiketimes is a list of arrays of spike times, like that returned
          by the function Stim2Spikes.
      y_range is a 2-tuple that holds the y-values that the raster ticks
          should be drawn between
    '''
    N = len(st)  # number of neurons

    levels = np.linspace(y_range[1], y_range[0], N+1, endpoint=True)
    for n in range(N):
        nspikes = len(st[n])
        y = [ [levels[n+1]]*nspikes , [levels[n]]*nspikes ]
        plt.plot(np.vstack((st[n],st[n])), y, color=np.random.rand(3))
    plt.ylim(y_range)
    plt.xlabel('Time (s)')
    return


def GenerateSpikeTrain(rates, T, jitter=0.):
    '''
    spike_times = GenerateSpikeTrain(rates, T)

    Creates a spike train (as an array of time stamps).

    Input:
    rates is an array or list of firing rates (in Hz), one
        firing rate for each interval.
    T is an array or list (the same size as 'rates') that gives
        the ending time for each interval
    jitter is a scalar that determines how much the spikes
        are randomly moved

    Output:
    spike_times is an array of times when spikes occurred

    Example: To create a spike train of 10Hz for 0.5s, followed
             by 25Hz that starts at 0.5s and ends at 2s, use

               GenerateSpikeTrain([10, 25], [0.5, 2])
    '''
    s = []
    t = 0.
    for idx in range(0,len(rates)):
        Trange = T[idx] - t
        if rates[idx]!=0:
            delta = 1. / rates[idx]
            N = rates[idx] * Trange
            times = np.arange(t+delta/2., T[idx], delta)
            times += np.random.normal(scale=delta*jitter, size=np.shape(times))
            s.extend(times)
        t = T[idx]
    s.sort()
    return np.array(s)
```

```
def spikes_between(spiketrain, t_start, t_end):
    '''
     numspikes = spikes_between(spiketrain, t_start, t_end)

     Returns the number of times between t_start and t_end.
     Specifically, it counts a spike if it occurred at t, where
     t_start <= t < t_end

     Inputs:
       spiketrain   array-like list of spike times
       t_start      start time
       t_end        end time

     Output:
       numspikes    number of spikes, where t_start <= t < t_end
    '''
    sp_bool = np.logical_and( np.array(spiketrain)>=t_start, np.array(spiketrain)<t_end )
    return np.sum(sp_bool)
```

Classes

Neuron class

This is the base class for different types of neurons.

In [3]:

```
class Neuron(object):
    '''
     neur = Neuron()

     This is an abstract base class for Neuron objects.
    '''
    def __init__(self):
        self.t = 0.        # current time
        self.axon = []     # list of outgoing connections

    def slope(self):
        return 0.

    def step(self, dt):
        raise NotImplementedError

    def send_spike(self, n=1):
        for a in self.axon:
            a.transmit(n)
```

Synapse class

This class represents a connection between two neurons.

In [4]:

```
class Synapse(object):
    '''
     The Synapse class represents a connection between a pre-synaptic neuron and
     post-synaptic neuron. This class implements the presence and strength (weight)
     of the connection, but does NOT model the dynamics of the connection.
    '''
    def __init__(self, pre, post, w):
        self.pre = pre                 # pre-synaptic neuron (object)
        self.post = post               # post-synaptic neuron (object)
        self.pre.axon.append(self)     # record this synapse in the pre-syn neuron
        self.w = w                     # connection weight

    def transmit(self, n=1):
        '''
         syn.transmit(n=1)

         Transmit n spikes through this synapse, from the
         pre-syn neuron to the post-syn neuron. The spikes get multiplied
         by this Synapse's connection weight.
        '''
        self.post.receive_current(n*self.w)
```

`InputNeuron` class

Derived from the `Neuron` class, this class is for generating input to feed into a network.

In [5]:

```
class InputNeuron(Neuron):
    '''
     InputNeuron(spiketrain)

     Constructor for InputNeuron class.

     InputNeuron is a class of neuron that can be used to inject spikes into
     the network. When involved in a simulation, an InputNeuron will generate
     spikes at the times specified during its construction.

     Inputs:
       spiketrain is an array or list of spike times
    '''
    def __init__(self, spiketrain):
        super().__init__()
        self.spikes = np.array(spiketrain)

    def step(self, slopes, dt):
        n_spikes = spikes_between(self.spikes, self.t, self.t+dt)
        self.t += dt
        if n_spikes>0:
            self.send_spike(n_spikes)
```

`LIFNeuron` class

Derived from the `Neuron` class, this class implements the Leaky Integrate-and-Fire (LIF) neuron.

In [6]:

```
class LIFNeuron(Neuron):

    def __init__(self, Tau_m=0.02, Tau_ref=0.002, Tau_s=0.05):
        '''
         LIFNeuron(Tau_m=0.02, Tau_ref=0.002, Tau_s=0.05)

         Constructor for LIFNeuron class

         Inputs:
           Tau_m   membrane time constant, in seconds (s)
           Tau_ref refractory period (s)
           Tau_s   synaptic time constant (s)
        '''
        super().__init__()
        # self.t and self.axon are defined in the super-class, Neuron.
        self.tau_m = Tau_m      # membrane time constant
        self.tau_ref = Tau_ref  # refractory period
        self.tau_s = Tau_s      # synaptic time constant
        self.v = 0.             # sub-threshold membrane potential (voltage)
        self.s = 0.             # post-synaptic current (PSC)

        self.weighted_incoming_spikes = 0.  # weighted sum of incoming spikes (for one time step)
        self.ref_remaining = 0. # amount of time remaining in the refractory period

        # For plotting
        self.v_history = []     # records v over time
        self.s_history = []
        self.spikes = []        # list of times when this neuron spiked

    def slope(self):
        '''
         LIFNeuron.slope()

         Evaluates the right-hand side of the differential equations that
         govern v and s.

         Output
           [dvdt, dsdt]  the slopes, in a list
        '''
        dvdt = ( self.s - self.v ) / self.tau_m # [1]
        dsdt = -self.s / self.tau_s  # [1]

        return [dvdt, dsdt]


    def step(self, slopes, dt):
        '''
         LIFNeuron.step(dt)

         Updates the LIF neuron state by taking an Euler step in v and s.
         The length of the step is dt seconds.

         Input
          slopes   list-like, containing the slopes of v and s
          dt       time step (in seconds)

         If v reaches the threshold of 1, the neuron fires an action potential
         (spike). Linear interpolation is used to estimate the time that v=1.
         The spike time is appended to the list self.spikes, and v
```

```
         is set to zero. After a spike, the neuron is dormant for self.tau_ref
         seconds.
        '''
        dvdt, dsdt = slopes
        # Update input current, included newly-arrived spikes
        self.s += dt*dsdt + self.weighted_incoming_spikes/self.tau_s

        v_previous = self.v
        t = self.t
        dt_integrate = dt

        # Implement refractory period
        if dt-self.ref_remaining>0:
            dt_integrate = max(0, dt-self.ref_remaining)
            t = self.t + self.ref_remaining
            self.v += dt_integrate*dvdt  # Euler step
            self.ref_remaining = 0
        else:
            self.v = 0.
            self.ref_remaining -= dt

        # Detect spike: if v reaches 1, spike
        if self.v>=1.0:
            # SPIKE!

            # Interpolate spike time
            v0 = v_previous
            v1 = self.v
            tstar = t + dt_integrate * (1.-v0) / (v1-v0)
            self.spikes.append(tstar)  # Record spike time

            self.v = 1.  # Set v to 1 (or zero)

            self.ref_remaining = self.tau_ref - (dt - (tstar-self.t))

            # Broadcast the spike to downstream neurons
            self.send_spike()

        # Store v (for plotting), and reset incoming spike accumulator
        self.v_history.append(self.v)
        self.s_history.append(self.s)
        self.weighted_incoming_spikes = 0.
        self.t += dt

    def receive_current(self, c):
        '''
         LIFNeuron.receive_current(c)

         Registers the arrival of current from a synapse. The
         member variable self.total_injected_current keeps track of all
         the incoming current for a time step.

         It is sufficient to add all currents together to tabulate the
         total incoming current (from all presynaptic neurons).

         Input:
           c    incoming current
        '''
        self.weighted_incoming_spikes += c
```

```
    def __repr__(self):
        '''
         print(neur)

         Prints the current time, membrane potential, input current, and
         remaining refractory time.
        '''
        return '{0:6.4f}s: s={1:5.3f}, v={2:6.4f}, ref remaining={3:7.5f}'.format(self.t, self.s, s
```

`SpikingNetwork` class

This class represents a collection of neurons and their connections to each other. Add neurons, connect them, and then simulate the network.

In [7]:

```
class SpikingNetwork(object):
    '''
     SpikingNetwork()

     Constructor for SpikingNetwork class.

     The SpikingNetwork class contains a collection of neurons,
     and the connections between those neurons.
    '''
    def __init__(self):
        self.neur = []          # List of neurons (of various kinds)
        self.conn = []          # List of connections
        self.t_history = []     # List of time stamps for the Euler steps
                                # (Useful for plotting)

    def add_neuron(self, neur):
        '''
         SpikingNetwork.add_neuron(neuron)

         Adds a neuron to the network.

         Input:
           neuron is an object of type LIFNeuron or InputNeuron
        '''
        self.neur.append(neur)

    def connect(self, pre, post, w):
        '''
         SpikingNetwork.connect(pre, post, w)

         Connects neuron 'pre' to neuron 'post' with a connection
         weigth of w.

         where
          pre    is the pre-synaptic neuron object,
          post   is the post-synaptic neuron object, and
          weight is the connection weight.
        '''
        syn = Synapse(pre, post, w)
        self.conn.append(syn)

    def simulate(self, T, dt):
        '''
         SpikingNetwork.simulate(T, dt)

         Simulates the network for T seconds by taking Euler steps
         of size dt.

         Inputs:
           T    how long to integrate for
           dt   time step for Euler's method
        '''

        current = 0 if len(self.t_history)==0 else self.t_history[-1]
        t_segment = np.arange(current, current+T, dt)

        for tt in t_segment:
            self.t_history.append(tt)
```

```
            # Compute slopes for all neurons first...
            slopes = []
            for neur in self.neur:
                slopes.append(neur.slope())

            # ... then update the neurons using an Euler step.
            for neur,slope in zip(self.neur, slopes):
                neur.step(slope, dt)

    def all_spike_times(self):
        '''
         SpikingNetwork.AllSpikeTimes()

         Returns all the spikes of all the neurons in the network.
         Useful for making spike-raster plots of network activity.

         Output:
           all_spikes  a list of sublists, where each sublist holds
                       the spike times of one of the neurons
        '''
        all_spikes = []
        for neur in self.neur:
            all_spikes.append(np.array(neur.spikes))
        return all_spikes
```

Assignment Questions

Q1: Two LIF Neurons

(a)

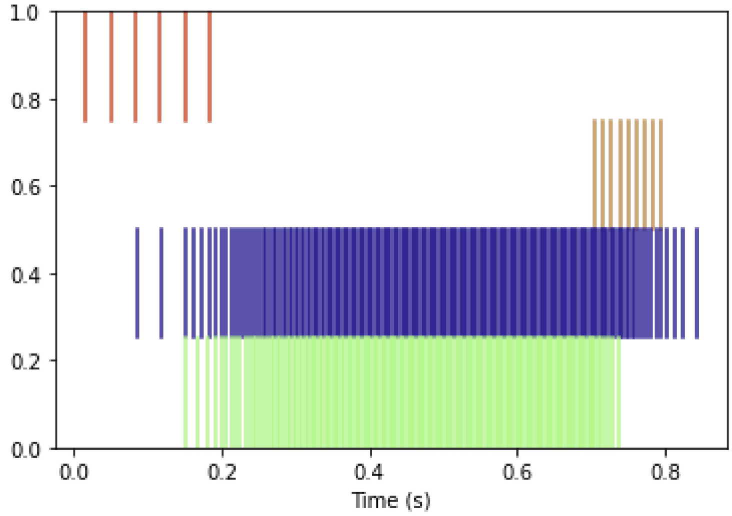
In [8]:

```
# ===== YOUR CODE HERE =====
InA = InputNeuron( GenerateSpikeTrain([30], [0.2]) )
InB = InputNeuron( GenerateSpikeTrain([0, 90], [0.7, 0.8]) )
LIFA = LIFNeuron()
LIFB = LIFNeuron()
network = SpikingNetwork()
network.add_neuron(InA)
network.add_neuron(InB)
network.add_neuron(LIFA)
network.add_neuron(LIFB)
network.connect(InA, LIFA, 0.05)
network.connect(InB, LIFB, -0.25)
network.connect(LIFA, LIFB, 0.05)
network.connect(LIFB, LIFA, 0.05)
```

(b)

In [9]:

```
# ===== YOUR CODE HERE =====
network.simulate(1, 0.001)
st = network.all_spike_times()
PlotSpikeRaster(st)
```



(c)

<< PLACE YOUR ANSWER BY DOUBLE-CLICKING HERE >>

B is most similarto the interaction bewteen neurons A and B. Because the frequency pattern of A and B are very similar. So they should have some connections like autdio feedback from holding a microphone too close to its loudspeaker.

Q2: Ring Oscillator

(a)

In [10]:

```
# ===== YOUR CODE HERE =====
LIFA = LIFNeuron(0.05, 0.002, 0.1)
LIFB = LIFNeuron(0.05, 0.002, 0.1)
LIFC = LIFNeuron(0.05, 0.002, 0.1)
LIFD = LIFNeuron(0.05, 0.002, 0.1)
LIFE = LIFNeuron(0.05, 0.002, 0.1)
LIFF = LIFNeuron(0.05, 0.002, 0.1)
LIFG = LIFNeuron(0.05, 0.002, 0.1)
LIFH = LIFNeuron(0.05, 0.002, 0.1)

network = SpikingNetwork()
network.add_neuron(LIFA)
network.add_neuron(LIFB)
network.add_neuron(LIFC)
network.add_neuron(LIFD)
network.add_neuron(LIFE)
network.add_neuron(LIFF)
network.add_neuron(LIFG)
network.add_neuron(LIFH)

network.connect(LIFA, LIFB, 0.2)
network.connect(LIFB, LIFC, 0.2)
network.connect(LIFC, LIFD, 0.2)
network.connect(LIFD, LIFE, 0.2)
network.connect(LIFE, LIFF, 0.2)
network.connect(LIFF, LIFG, 0.2)
network.connect(LIFG, LIFH, 0.2)
network.connect(LIFH, LIFA, 0.2)
```

(b)

In [11]:

```
# ===== YOUR CODE HERE =====
network.connect(LIFB, LIFA, -0.4)
network.connect(LIFC, LIFB, -0.4)
network.connect(LIFD, LIFC, -0.4)
network.connect(LIFE, LIFD, -0.4)
network.connect(LIFF, LIFE, -0.4)
network.connect(LIFG, LIFF, -0.4)
network.connect(LIFH, LIFG, -0.4)
network.connect(LIFA, LIFH, -0.4)
```

(c)

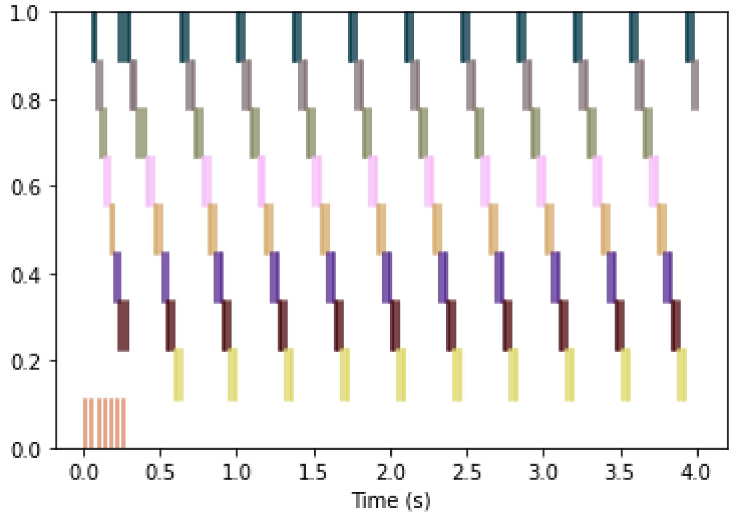
In [12]:

```
# ===== YOUR CODE HERE =====
In = InputNeuron( GenerateSpikeTrain([25], [0.3]) )
network.add_neuron(In)
network.connect(In, LIFA, 0.2)
```

(d)

In [13]:

```
# ===== YOUR CODE HERE =====
network.simulate(4, 0.001)
st = network.all_spike_times()
PlotSpikeRaster(st)
```



(e)

<< PLACE YOUR ANSWER BY DOUBLE-CLICKING HERE >>

It takes 0.59758285 seconds for the wave activity to go around the ring. I get this answer by running all_spike_times().

In [14]:

```
network.all_spike_times()
```

Out[14]:

```
[array([0.06323586, 0.08640641, 0.2387654 , 0.26527412, 0.28428486,
        0.30856503, 0.64232758, 0.65878263, 0.67876108, 1.00750106,
        1.02454622, 1.04546954, 1.37351186, 1.39074862, 1.41198143,
        1.73895918, 1.75616687, 1.77735844, 2.10492516, 2.12212006,
        2.1432906 , 2.47087011, 2.48804399, 2.50917981, 2.8368685 ,
        2.8540418 , 2.87517666, 3.20286846, 3.22004174, 3.24117658,
        3.56886846, 3.58604174, 3.60717658, 3.93486846, 3.95204174,
        3.97317658]),
 array([0.09331822, 0.11556013, 0.30961805, 0.32424854, 0.34145653,
        0.68631267, 0.70338719, 0.72435838, 1.05333346, 1.07068185,
        1.09209911, 1.41922204, 1.43652885, 1.45787788, 1.78482753,
        1.80206433, 1.82330374, 2.15077427, 2.16799089, 2.18919686,
        2.51670707, 2.53390035, 2.55506475, 2.88270506, 2.8998977 ,
        2.92106093, 3.24870501, 3.26589763, 3.28706084, 3.61470501,
        3.63189763, 3.65306083, 3.98070501, 3.99789763]),
 array([0.12262893, 0.14483031, 0.36042191, 0.38189736, 0.41096011,
        0.73408854, 0.7520944 , 0.7746193 , 1.10028822, 1.1178111 ,
        1.13952128, 1.46498142, 1.48219698, 1.50340148, 1.83070038,
        1.84789138, 1.86905169, 2.19646392, 2.21368301, 2.23488786,
        2.56237374, 2.57956036, 2.60071358, 2.92837111, 2.94555684,
        2.96670863, 3.29437104, 3.31155675, 3.3327085 , 3.66037104,
        3.67755675, 3.6987085 ]),
 array([0.15162893, 0.17383031, 0.41765017, 0.43681605, 0.46137693,
        0.78259365, 0.80031114, 0.8223446 , 1.14579713, 1.16277146,
        1.18358176, 1.51051577, 1.52764226, 1.54869755, 1.87629081,
        1.89344703, 1.91455155, 2.24178708, 2.25893111, 2.28001499,
        2.60780085, 2.62494955, 2.64604184, 2.97380124, 2.99095007,
        3.01204259, 3.33980125, 3.35695008, 3.37804261, 3.70580125,
        3.72295008, 3.74404261]),
 array([0.18062893, 0.20283031, 0.46668733, 0.48406353, 0.50553231,
        0.82429094, 0.8398508 , 0.85844456, 1.19009945, 1.20699772,
        1.22768813, 1.55575097, 1.57288312, 1.59394743, 1.92175387,
        1.93888662, 1.95995188, 2.28740579, 2.30460423, 2.3257763 ,
        2.65342685, 2.67063271, 2.69181658, 3.01942743, 3.0366335 ,
        3.05781767, 3.38542745, 3.40263352, 3.4238177 , 3.75142745,
        3.76863352, 3.7898177 ]),
 array([0.20962893, 0.23183031, 0.52350516, 0.54547979, 0.8676451 ,
        0.8849207 , 0.90622212, 1.23515494, 1.25235663, 1.27353644,
        1.60096428, 1.61817393, 1.63936873, 1.96687124, 1.98404553,
        2.00518202, 2.33286853, 2.35004184, 2.37117671, 2.69886846,
        2.71604174, 2.73717658, 3.06486846, 3.08204174, 3.10317658,
        3.43086846, 3.44804174, 3.46917658, 3.79686846, 3.81404174,
        3.83517658]),
 array([0.23862893, 0.26083031, 0.2914672 , 0.54733147, 0.56702491,
        0.59256053, 0.91458773, 0.93222037, 0.95411016, 1.28105005,
        1.29829127, 1.31953693, 1.64676796, 1.66398244, 1.6851846 ,
        2.01270689, 2.02990011, 2.0510644 , 2.37870506, 2.39589769,
        2.41706092, 2.74470501, 2.76189763, 2.78306083, 3.11070501,
        3.12789763, 3.14906083, 3.47670501, 3.49389763, 3.51506083,
        3.84270501, 3.85989763, 3.88106083]),
 array([0.59758285, 0.61486983, 0.63618876, 0.96112999, 0.97830338,
        0.99943731, 1.32679493, 1.34394158, 1.3650302 , 1.69245388,
        1.70966947, 1.73086878, 2.05842823, 2.07563456, 2.09681916,
        2.42442747, 2.44163354, 2.46281774, 2.79042745, 2.80763352,
        2.8288177 , 3.15642745, 3.17363352, 3.1948177 , 3.52242745,
```

```
        3.53963352,  3.5608177 ,  3.88842745,  3.90563352,  3.9268177 ]),
 array([0.02,  0.06,  0.1 ,  0.14,  0.18,  0.22,  0.26])]
```