BioE 332 Assignment 1: Persistent activity in a spatial working memory model

Kwabena Boahen, TA: Tatiana Engel

Assignment due: April 12, 2013

1 Goals

In this assignment we will build a cortical network model that captures neuronal persistent activity, similar to the persistent activity recorded in the prefrontal cortex of animals performing a spatial working memory task. We will see how stable activity patterns emerge from the recurrent network dynamics and persist on time scales greatly exceeding biophysical time constants of constituent neurons and synapses. In the language of dynamical systems, the spatial working memory model implements a line attractor. We will explore the sensitivity of this line attractor to the spiking noise and to the heterogeneity of the network parameters. By completion of this assignment, we will learn and understand the following:

Scientifically:

1. Leaky integrate-and-fire (LIF) model of a neuron.
2. Conductance-based models of biophysical synapses: AMPA, NMDA and GABA.
3. Line attractor and persistent activity in the spatial working memory model.
4. Random drift of activity along the line attractor, sensitivity to noise.

Technically:

1. Neural network simulations with Brian.
2. Superposition of inputs on linear synapses.
3. Implementation of non-linear NMDA synapses.

Readings:

1. Working memory model that will be implemented in this assignment, Ref. [1].
2. Extension question, robustness of the line attractor to the heterogeneity of network parameters, Ref. [2].

2 Getting started with Brian

To simulate neural networks in software, we will use Brian, a python-based neural network simulator (www.briansimulator.org). Brian is as versatile as it is easy to use – it can directly interpret mathematical equations of the model you would like to simulate. So, type your equations in and you are ready to go!

Installing Brian is easy too. First, you need a working Python distribution with a couple of required packages, such as NumPy and SciPy. The easiest way to get everything is to download the Enthought Python distribution from http://www.enthought.com/products/epd.php. If you register on their web-site with your academic e-mail address, you will get a free Python distribution with all required packages already included. After you install that, you can just type into the terminal

```
easy_install brian
```

and Brian will be installed. For manual installation instructions see the Brian web-site.

Now open a terminal and start an interactive Python interpreter:

```
ipython --pylab
```

Here you can interactively type in Python commands, or if you put your code into a script with the .py extension, you can execute it with the command

```
execfile('my_script.py')
```

Brian comes with two tutorials and plenty of code examples. First, complete both tutorials to learn how to implement simple integrate-and-fire models in Brian, how to set up synaptic connections in a network, and how to record spikes or any variable of your interest.

3 Building the neural circuit model of working memory

We will reproduce the working memory model from Ref. [1]. The model is a recurrent neural network, that comprises $N_\mathrm{E} = 2048$ excitatory and $N_\mathrm{I} = 512$ inhibitory neurons. We will build the model in sequential steps, by first wiring up the network of inhibitory neurons only, then adding excitatory neurons to it, and finally creating spatially structured synaptic connectivity. We begin by setting up the basic network components: neurons and synapses.

3.1 Integrate-and-fire neurons

Individual neurons are modeled with a simple leaky integrate-and-fire (LIF) model:

$$C_\mathrm{m}\frac{dV}{dt} = -g_\mathrm{L}(V - E_\mathrm{L}) + I, \tag{1}$$

where variable V represents the membrane voltage, and parameters are: C_m - membrane capacitance, g_L - leak conductance, E_L - leak reversal potential. I is the input current to the neuron. The "integrate" part of the LIF neuron refers the sub-threshold integration of the input current with the time constant $\tau_\mathrm{m} = g_\mathrm{L}/C_\mathrm{m}$. In the absence of any input the voltage decays to the steady-state resting potential $V = E_{\mathrm{L}}$. The “fire” part of the LIF neuron refers to the fixed voltage threshold V_{th}: each time the voltage reaches this threshold a spike is elicited and the voltage is reset to a fixed reset potential V_{res}.

3.2 Linear conductance-based synapses

In a network, neuronal input currents arise from spikes arriving through the synapses. Each synapse contains a population of embedded receptor channels that conduct current when they bind neurotransmitter (e.g. AMPA, NMDA or GABA). The more receptors bind neurotransmitter, the higher is the synaptic conductance.

When a spike arrives, the neurotransmitter is released from presynaptic vesicles, the receptors bind the released neurotransmitter and synaptic conductance increases. Thereafter receptors unbind neurotransmitter, whereby the fraction of activated receptors decays exponentially with the synaptic time constant τ_{syn}. If spikes arrive at the synapse at a very high rate, the concentration of neurotransmitter can increase so much that binding rate exceeds the unbinding rate. In this case all receptors bind neurotransmitter and synaptic conductance saturates. All chemical synapses exhibit this nonlinear saturation due to the finite number of postsynaptic receptors.

Since the time constants for AMPA ($\tau_{\mathrm{syn}} \sim 2$ms) and GABA ($\tau_{\mathrm{syn}} \sim 10$ms) receptors are relatively short, i.e. their unbinding rates are fast, the conductance of these synapses does not saturate for realistic input firing rates, and their dynamics can be modeled by a simple linear equation:

$$\begin{aligned} I_{\mathrm{syn}} &= -G_{\mathrm{syn}}\, s\, (V - E_{\mathrm{syn}}), & (2) \\ \frac{ds}{dt} &= -\frac{1}{\tau_{\mathrm{syn}}} s + \sum_i \delta(t - t_i). & (3) \end{aligned}$$

Here, the current I_{syn} is proportional to the product of the voltage and the synaptic conductance G_{syn}. The dimensionless variable s represents the fraction of open channels and is called the synaptic gating variable. At the time of each presynaptic spike (t_i), the gating variable s is instantaneously incremented by a fixed amount (in our case 1), and exponentially decays between spikes with the time constant τ_{syn}.

An important characteristic of the linear synapse model Eq. 3 is a property of superposition. As for any linear system, the net response caused by two or more input spike trains is the sum of the responses which would have been caused by each spike train individually. The superposition property of linear synapses gives us a great advantage for network simulations. We only need to model one synaptic gating variable s for each receptor type of each neuron. All input spike trains arriving from different neurons can be fed to the same gating variable. In this way, the number of gating variables that we need to model is equal to the number of neurons (N), and not to the number of synapses ($\sim N^2$).

3.3 Recurrent sub-network of inhibitory neurons

Now we are ready to implement the first building block of the working memory model: a population of $N_{\mathrm{I}} = 512$ recurrently connected inhibitory neurons that receive external excitatory inputs through AMPA synapses. Both AMPA and GABA synapses are modeled by a linear model Eq. 3. As in Ref. [1], the background input is modeled as uncorrelated Poisson spike trains to each neuron at rate of $\nu_{\text{ext}} = 1.8$kHz per cell.

First, we need to set up the model equations:

```
eqs_i = '''
dv/dt = (-gl_i*(v-El_i)-g_ext_i*s_ext*(v-E_ampa)- \
Gii*s_gaba*(v-E_gaba))/Cm_i: volt
ds_ext/dt = -s_ext/t_ampa : 1
ds_gaba/dt = -s_gaba/t_gaba : 1
'''
```

For single neuron model and for AMPA and GABA synapses use the parameter values specified in Ref. [1], and do not forget to specify the units.

Now we generate a neural population (NeuronGroup in Brian) using these equations:

```
Pi = NeuronGroup(NI, eqs_i, threshold=Vth_i, \
reset=Vr_i, refractory=tr_i)
```

To deliver Poisson input to each neuron, we can use the class PoissonGroup (a group that generates independent Poisson spike trains) and the class IdentityConnection (a connection between two groups of the same size, where neuron i in the source group is connected to neuron i in the target group):

```
PGi  = PoissonGroup(NI, fext)
Cpi = IdentityConnection(PGi, Pi, 's_ext', weight=1.0)
```

Now we only need to set up the recurrent inhibitory connections:

```
Cii = Connection(Pi, Pi, 's_gaba', weight=1.0)
```

and then we can run the network and make the raster plots of activity. You should see asynchronous firing of neurons at a low firing rate.

3.4 Specify integration method and time step

Before we move on to build the rest of the network, we need to specify some details about the algorithm used for numerical integration of model equations. For accurate numerical integration, we will use the second order Runge-Kutta method with small integration time step $dt = 0.02$ ms. By default Brian uses Euler method, but this can be changed by providing the argument order=2 to the NeuronGroup constructor.

The integration time step can be specified using Brian's clock object. Many Brian objects store a clock that specifies at which times the object will be updated. If no clock is specified, the program uses the global default clock. Each clock has the time step attribute dt, which you can set to the desired value. We will create our own clock simulation_clock, and use it to update the neuron groups as well as the external Poisson groups. The specification of the inhibitory and Poisson neuronal groups will be slightly changed:

```
simulation_clock=Clock(dt=0.02*ms)
```

```
Pi = NeuronGroup(NI, eqs_i, threshold=Vth_i, reset=Vr_i, \
refractory=tr_i, clock=simulation_clock, order=2)

PGi = PoissonGroup(NI, fext, clock=simulation_clock)
```

3.5 Saturating non-linearity of NMDA synapses

Modeling dynamics of NMDA synapses is a little trickier because the NMDA conductance saturates at high presynaptic firing rates. The saturation of NMDA receptors is due to their long time constant, which exceeds ~ 50 times the AMPA time constant, and ~ 10 times the GABA time constant. Both, the long synaptic time constant of NMDA receptors and their saturating nonlinearity are crucial for the stability of persistent firing states in the working memory model.

Non-linear dynamical equations for NMDA gating variable are given in Ref. [1]. Since NMDA dynamics is non-linear, we cannot make use of the superposition principle to implement it with a single postsynaptic gating variable, as we did for AMPA and GABA synapses. Still it is possible to implement NMDA synapses with only N gating variables for the network of N neurons. The trick is to integrate NMDA gating variable *presynaptically* for each neuron projecting NMDA synapses. This presynaptic value of NMDA gating is then used to compute the input currents to all postsynaptic targets.

The Brian equations for the excitatory population in the working memory model read:

```
eqs_e = '''
dv/dt = (-gl_e*(v-El_e) - g_ext_e*s_ext*(v-E_ampa) \
- Gie*s_gaba*(v-E_gaba) \
- Gee*s_tot*(v-E_nmda)/(1+b*exp(-a*v)))/Cm_e: volt
ds_ext/dt = -s_ext/t_ampa : 1
ds_gaba/dt = -s_gaba/t_gaba : 1
ds_nmda/dt = -s_nmda/t_nmda+alfa*x*(1-s_nmda) : 1
dx/dt = -x/t_x : 1
s_tot : 1
'''
```

Note another non-linear term in the NMDA synaptic current equation, which models voltage-dependence of NMDA channels. In this implementation, the variable *s_nmda* represents the *presynaptic* NMDA gating variable of a neuron, whereas the variable *s_tot* is a *postsynaptic* NMDA gating for this neuron, that is a weighted sum of NMDA gatings from all its presynaptic inputs. To finish implementation of NMDA dynamics, we need to modify the reset mechanism to increment the auxiliary variable x at the time of each spike:

```
reset='''
v=Vr_e
x+=1*1
'''
```

and we also need to compute *s_tot* for each neuron on each time step. To this end, Brian has a mechanism called network_operation, which allows to specify arbitrary operations that will be performed on each time step. We will make use of it in the next section.

3.6 Recurrent network with uniform connectivity

Now we can implement the entire working memory model, with the only simplification that the recurrent excitatory connections are not structured but uniform for now. To do this follow these steps:

1. Add the input current through NMDA conductance to the inhibitory (I) population.
2. Create a neural group for the excitatory (E) population with $N_{\mathrm{E}} = 2048$.
3. Add the external Poisson drive to the E population.
4. Add the recurrent inhibition from I to E cells with a uniform connection strengths and weight equal to 1.

Now we will use the network_operation mechanism to implement the update of NMDA conductance *s_tot* on each time step. For simplicity, we just add uniform connections from each excitatory neuron to all E cells and to all I cells with the weight 1. This means that each neuron receives just the sum of presynaptic NMDA gatings of all E cells. The Brian code to implement this reads

```
@network_operation(clock=simulation_clock, when='start')
def update_nmda(clock=simulation_clock):
    s_NMDA = Pe.s_nmda.sum()
    Pe.s_tot = s_NMDA
    Pi.s_tot = s_NMDA
```

You can run the network now and observe asynchronous activity at low firing rate in both E and I populations.

3.7 Structured recurrent excitation

The final step is to implement the structured recurrent excitation. Here we will use another simple trick, that helps to speed up the simulation and to simplify implementation. The trick takes advantage of the circular symmetry of the working memory model, the so called "ring" architecture. The connectivity strength depends only on the difference between preferred directions of two neurons, but not on the absolute value of their preferred directions. Hence the postsynaptic weighted sum of NMDA gating is a convolution of the presynaptic gating with the connectivity profile $W(\theta_i - \theta_j)$:

$$s\_tot(\theta) = (s\_nmda * W)(\theta). \tag{4}$$

Convolution is efficiently computed in the Fourier domain as the product of the Fourier transforms of *s_nmda* and W. Therefore we will precompute the Fourier transform of the connectivity profile $fweight$, and then use it on each time step to update the NMDA gating *s_tot*. The mathematical expression for the connectivity profile together with the normalization condition from Ref. [1] can be implement as

```
from scipy.special import erf
from numpy.fft import rfft, irfft

# Jm_ee is calculated from the normalization condition
tmp   = sqrt(2*pi)*sigma_ee*erf(360.*0.5/sqrt(2.)/sigma_ee)/360.
Jm_ee = (1.-Jp_ee*tmp)/(1.-tmp)

weight=lambda i:(Jm_ee+(Jp_ee-Jm_ee)* \
exp(-0.5*(360.*min(i,NE-i)/NE)**2/sigma_ee**2))

weight_e=zeros(NE)
for i in xrange(NE):
    weight_e[i]=weight(i)

fweight = rfft(weight_e) # Fourier transform
```

Now change the network_operation for the NMDA update that we created earlier, in order to incorporate the structured excitation. Use precomputed $fweight$ and the inverse Fourier transform. Please note that Ref. [1] has a typo: σ_{EE} should be 14.4° instead of 18° as specified in the paper.

The network model is now complete. You can run it and observe an asynchronous spontaneous activity state with low firing rate.

4 Simulating stimulus-specific persistent activity

As you know from Ref. [1], the neural circuit model that we just have built, possess not only a homogeneous low activity state, but also the whole family of stable attractor states with a localized profile of high activity ("bump"). To trigger a transition from the low activity state to the bump attractor, we will stimulate the network for a brief period of time with a spatially selective input current, the same way as specified in Ref. [1]. To inject the stimulation current, add an additional input current term I_e to the equations for the excitatory population.

We first precompute the the spatially-selective input current to all neurons for the desired cue direction:

```
# return the distance between neurons on the ring
def circ_distance(deltaTheta):
    if (deltaTheta > 0):
        return min(deltaTheta,360-deltaTheta)
    else:
        return max(deltaTheta,deltaTheta-360)

currents = lambda i,j: i_cue_amp* \
exp(-0.5*circ_distance((i-j)*360./NE)**2/i_cue_width**2)

# precompute input current for the cue direction i_cue_ang
```

```
current_e=zeros(NE)
j = i_cue_ang*NE/360.
for i in xrange(NE):
    current_e[i]=currents(i,j)
```

Then we use the network_operation to switch this current on only during the relevant period of time:

```
current_clock = Clock(dt=50*ms)

@network_operation(current_clock, when='start')
def update_currents(current_clock):
    if (current_clock.t>tc_start and current_clock.t<tc_stop):
        Pe.I_e = current_e
    else:
        Pe.I_e = 0
```

Here we introduced another clock that we called current_clock with the larger time step dt = 50 ms. This coarser time resolution is sufficient to determine when to switch the input on and off, it is not necessary to perform this check on every integration time step.

Now generate the raster plot of activity in excitatory and inhibitory populations [**Deliverable**]. An example is shown in Fig. 1, where a cue stimulus at 180° was presented at 1 s for duration of 250 ms.

5 Random activity drift on a line attractor

The energy landscape for the line attractor represents a one dimensional valley, with a flat landscape along the trough. All states along the trough correspond to the same minimal energy and are stable. Random fluctuations can therefore move the activity from state to state along the trough of the minimal energy. This will result in the random drift of activity bumps.

We would like to explore these random activity drifts. To quantify the position of the activity profile in the network, we will use the population vector measure specified in Ref. [1]. You can think about it as decoding the direction stored in the working memory network.

Run simulations of the working memory model for six seconds, repeat the simulation one hundred times for three network sizes ($N_{\rm E}$ =1024, 2048 and 4096). Plot the angle of the population vector as a function of time for twenty trials [**Deliverable**], an example is shown in Fig. 2 (upper panels). Remember, that when you change the network size, you have to scale the synaptic conductances inversely proportionally to N and to keep $N_{\rm I}/N_{\rm E}$ ratio fixed. Use different random seeds for each repetition of the simulation. Compute and plot the variance for the angle of the population vector at each moment in time for three network sizes [**Deliverable**], example is shown in Fig. 2 (lower panel).

Hint. You can implement a version of the population vector measure, that is based not on the spike count in a fixed time window, but rather on the exponential filter of all previous spike times with a short time constant (e.g. 10 ms). This way you can implement

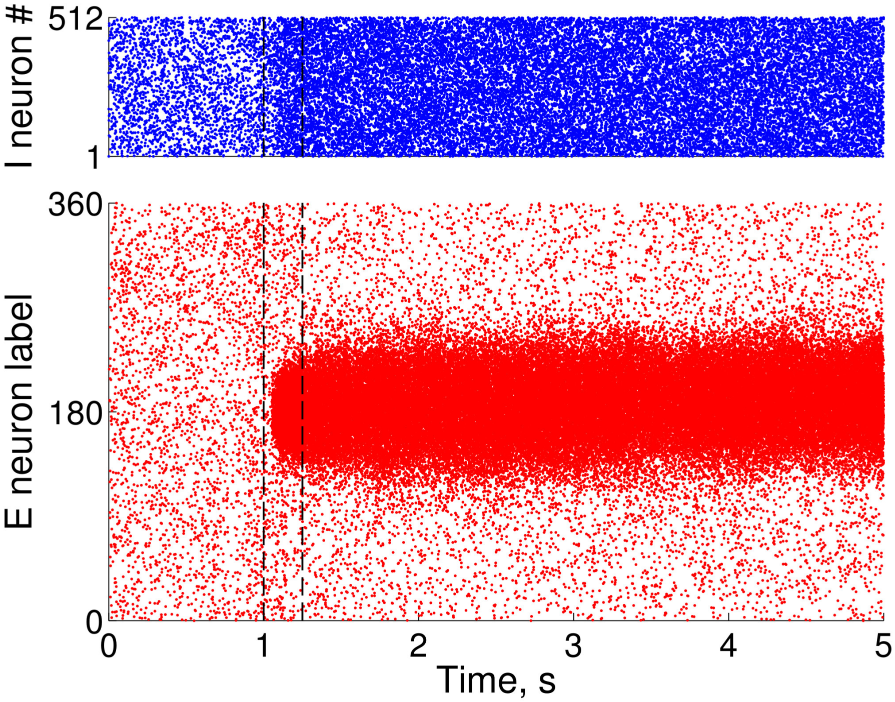

Figure 1: Working memory maintained by a tuned network activity state, reproduced Fig. 2 from Ref. [1].

the population vector computation directly in Brian, and do not need to store spike times for all the multiple model runs.

Hint. You can run multiple simulations on the Stanford Barley cluster (www.stanford.edu/group/farmshare/cgi-bin/wiki/index.php). A convenient way to run multiple similar simulations is to submit an array job. Here is a minimal script to run 100 simulations as an array job:

```
#!/bin/bash

#$ -t 1-100

$PYTHON_EXE $EXE --k $SGE_TASK_ID
```

Here PYTHON_EXE is your Python path, EXE is your executable script. The environmental variable SGE_TASK_ID contains the ID number for each simulation (1 to 100 in this case). In this example SGE_TASK_ID is passed as a command-line argument to the python script. In your Python code, you can use the Python module argparse to parse the command-line arguments. This example shows how to set the simulation seed using the parsed simulation ID number:

```
import argparse
```

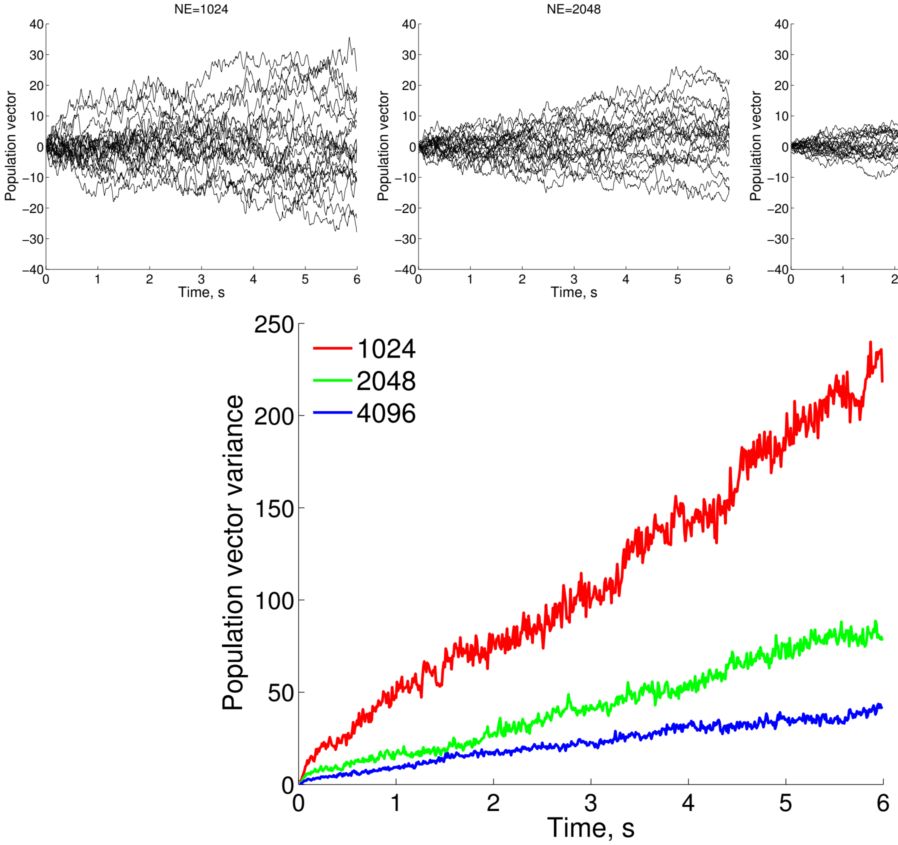

Figure 2: The network pattern of persistent activity drifts randomly in time due to noise, reproduced Fig. 5 from Ref. [1].

```
parser = argparse.ArgumentParser()
parser.add_argument('--k', type = int, default = 0)
pa = parser.parse_args()
numpy.random.seed(pa.k)
```

6 Extension question

To understand why activity drift is slower in networks of larger size, investigate the recurrent input current for a few representative neurons in the network. Examine the distributions of the total recurrent current, recurrent excitation, recurrent inhibition and the background excitatory current. Based on your measurements explain results in Fig. 2 [**Deliverable**].

Introduce heterogeneity in the network: instead of using the exact same parameters for all neurons, draw the leak reversal potential V_{L} randomly from a Gaussian distribution with the variance 1 mV. Investigate how heterogeneity influences the network's ability to maintain the memory of the stimulus location [**Deliverable**]. You can use Ref. [2] for your inspiration.

References

[1] A Compte, N Brunel, PS Goldman-Rakic, and Wang X-J. Synaptic mechsnisms and network dynamics underlying spatial working memory in a cortical network model. *Cerebral Cortex*, 10:910–923, 2000.

[2] A Renart, P Song, and X-J Wang. Robust spatial working memory through homeostatic synaptic scaling in heterogeneous cortical networks. *Neuron*, 38:473–85, 2003.