MATLAB simulation exercise instructions:

Important note: Make sure you've read the instructions on the wiki and Song *et. al.* paper before you start this simulation. The wiki instructions have important points about how to read the Song *et. al.* paper and how to write up your assignment.

Overview of this assignment: We will use two MATLAB functions modelNeuron.m and stepTime.m to implement the simulation described in Song et. al. To help you get started we also included two examples of simulation implementation: plotSimulation.m, and testParameters.m. First, go through the Getting Started section of these instructions. Then pick one of the simulation exercises that follow to investigate. You will need to modify some code to carry out the exercise; this could be as simple as changing a few parameters and looking at the result, or as complicated as writing a new variation on this simulation. These exercises represent "experiments" that you will run on the model!

Difficulty and collaboration: Depending on your MATLAB comfort level, this simulation may seem daunting. We've tried hard to make it accessible, so you can play with the simulation without being a code warrior. The problems below can be answered using minimal programming; you don't need to write a lot of code to get a good grade on this assignment. We've suggested both "novice" and "advanced" versions of the exercises below. Feel free to choose one that reflects your interest and comfort level, or if you've got your own idea, you can ignore the suggestions and do your own thing. We encourage you to discuss your findings with your classmates, but this exercise is only useful to the extent that you actually roll up your sleeves, get your hands dirty, and dig into the simulation. To that end, please collaborate in English, and not by sharing code.

Time requirement: We urge you to start early. Depending on the exercise you choose, the simulations may require a large amount of computer time to run. In our hands, simulated time can require up to 5x as much real time, but your mileage may vary.

Grading: Your narrative will be graded on the basis of several criteria. (1) Is your narrative organized and logical? Are your results displayed in a clear way in a figure? (2) Do you clearly state the interpretation of your results (i.e., why the model responded the way it did to your "experimental" manipulation)? Is this interpretation reasonable and have you given it some real thought? (3) Did you display creativity in designing your "experiment"? It's fine if you just follow the instructions and make minimal parameter changes, but a really creative angle will also win you some brownie points.

Getting started:

(1) Download the file modelNeuron.zip, and unzip it into a folder called 'modelNeuron'. It contains 4 MATLAB (.m) files: modelNeuron.m, stepTime.m, plotSimulation.m, and testParameters.m. Start the MATLAB environment and set the working directory to this folder.

(2) Open modelNeuron.m, and stepTime.m in separate windows or tabs in MATLAB. modelNeuron**.m** is a function that creates a model neuron with properties you can modify in your code. **stepTime.m** is a function that will simulate changes to your neuron as it takes (discrete) steps through time. These functions will begin to make more sense as you use the example code.

(3) Open plotSimulation.m in a separate window or tab. plotSimulation.m is an example of how you can use the functions from (2) to create a simulation. plotSimulation.m creates the structure `aNeuron`. A structure is a data type that can hold multiple types of information by having different 'fields' where each field has its own value. For example, our `aNeuron` structure has a field called `exSynapses.rate` which is the rate of presynaptic action potentials. After creating `aNeuron`, plotSimulation.m modifies some of the default values of the fields of `aNeuron`. For example, `exSynapses.rate` field is set to `15`. plotSimulation.m then runs the simulation by repeatedly calling stepTime.m. Along the way it reports the values of the membrane voltage and conductances.

(4) Run `plotSimulation()`. You can do this by entering the command: `plotSimulation();` on the MATLAB command line, or by clicking the "Run" button in the Editor window when plotSimulation.m is visible.

(5) You'll see that this script creates 3 figures:

Figure 1 shows traces of the membrane voltage, and total excitatory and inhibitory conductances of the post-synaptic neuron over time. Every few seconds, the windows will update as the simulation calculates a new chunk of time. You'll notice that Vm starts at -60 mV, then drifts higher due to synaptic conductances. When it reaches -54 mV, a spike is fired (the positive-going spike waveform is not shown) and Vm is abruptly reset to -60 mV, the reset voltage. In the bottom plot, you'll notice that the excitatory conductances of the neuron (blue) fluctuate much less than the inhibitory conductances (red). This is because there are fewer inhibitory synapses, and each has a greater weight than the excitatory synapses.

Figure 2 shows the distribution of excitatory synapse strengths. You'll see that initially all of the conductances start at gMax. As the simulation evolves, the STDP rule changes the distribution of synaptic conductances. Eventually, (similar to Fig. 2a in Song et. al.) the distribution of synaptic conductances will become bimodal with peaks at 0 and gMax, and stop changing. This is the "steady-state" that Song et. al. refers to.

Figure 3 shows the firing rate of the post-synaptic cell over the whole simulation time. Initially it fires at a high rate, but the STDP rule reduces the output rate until it reaches a steady state value.

(6) Click in the MATLAB command window, and press [Control]-C (Windows or OSX) to abort the simulation. (Note: you can always do this to stop a MATLAB script that you don't want to wait for to run to completion.) Make a mental note of the firing rates in Figure 3, and then close all the Figure windows to prevent clutter later on.

(7) Look at the code in plotSimulation.m. Scroll down to line 72:

```
72      aNeuron = modelNeuron();
```

This line simply creates the structure (or 'struct') `aNeuron`, by calling the modelNeuron.m function.

Look at the code in **modelNeuron.m.** You will see that modelNeuron.m sets default values for each field of the model neuron. These default values are matched to the simulations in Song et. al. `aNeuron` is a particular example of a model neuron. When you call `aNeuron = modelNeuron(); aNeuron` inherits all of the fields and default values of the model neuron.

(8) Now check out line 74 of plotSimulation.m:

```
74     aNeuron.exSynapses.rate = 15;
```

This line of code modifies the value of the `exSynapses.rate` field of our simulated neuron from the default value of 25 (defined in **modelNeuron.m**) to the new value of 15. Try increasing the value by modifying line 74 in plotSimulation.m:

```
74     aNeuron.exSynapses.rate = 20;
```

Save the file. Now run the simulation again. You should notice that at steady-state the post-synaptic firing rate is slightly higher. This makes sense, because we're increasing the excitatory drive. (You may also notice that the simulation converges to steady-state slightly faster.) If you're interested in what the other default parameters are, open modelNeuron.m. As an alternative to changing line 74 in plotSimulation.m, you could also remove it, and change the default values in modelNeuron.m. If you do this, it is good practice to "comment out" the original line and include a new modified line with your desired code.

(9) Now look at lines 76-77:

```
76    aNeuron.exSynapses.Aplus = .020;
77    aNeuron.exSynapses.Aminus = 1.05*aNeuron.exSynapses.Aplus;
```

Here we're changing the strength of the STDP rule, increasing it from the Song et. al. default of A_{plus} = .005, A_{minus} = (1.05)*.005. We did this because it makes the simulation converge to steady-state faster; this might be useful for your simulations too.

(10) Now scroll down in plotSimulation.m to lines 81-83:

```
81    for n=1:nTimePoints
82
83        aNeuron = stepTime(aNeuron,stepSize);
```

In line 81, we define a loop over all of the simulated time points. Each iteration of the loop calls the stepTime.m function, which takes two parameters: the neuron in its current state, and the size of the step forward in time to make. stepTime.m implements all of the rules in the simulation for figuring out how it changes through time. It's the belly of the beast; where the rubber meets the road; the blood and guts, and probably many other metaphors too. Note, it's important to choose an appropriate step size: too big, and you'll miss important changes that would have happened between steps. Too small, and your simulation will take forever to run. We recommend a step size of 0.1 ms (note stepSize is defined in seconds), which is comfortably shorter than the shortest time constant in the simulation. Read through stepTime.m to see what the simulation does on each time step, and see if you can understand the reason for each line of code. Most simulation exercises you undertake will not require you to modify `stepTime()` (or any part of modelNeuron.m) but feel free to do so if you're feeling ambitious.

(11) Open testParameters.m. This script simulates responses to different excitatory input rates. (See Song et. al., Fig. 2c) Read through the code and comments to see if you can understand how it works.

(12) If you have problems, contact Stephen Holtz (s@holtz.email) or Alex Batchelor (abatchelor@fas.harvard.edu).

Simulation Exercises: (Choose one.)

(1) Threshold:

Song et. al. note: "we find qualitatively new behavior when the intrinsic nonlinearity of the spike-generation mechanism is taken into account." This suggests that the spiking non-linearity is an important feature of their model. Modify the spike generation rule in the model. (Novice version: change threshold, `this.Vthresh` in modelNeuron.m, Advanced version: edit stepTime.m to implement a probabilistic model where spike rate is linearly dependent on membrane voltage.) How does this change the behavior of the simulated neuron? For example, how does it change how the output rate depends on the input rate? Explain why.

(2) Strength of the STDP rule:

Song et. al. see that a striking feature of STDP is a reduction of sensitivity to changes in input: increasing the input rate results in only small changes in post-synaptic firing (Fig. 2c). How does increasing the strength of STDP modify input sensitivity? Explain why this occurs. (Novice version: explore a range of STDP strengths or A_-/A_+ ratios (`aNeuron.exSynapses.Aplus` and `aNeuron.exSynapses.Aminus` in plotSimulation.m), Advanced version: put the A_-/A_+ ratio "under dynamic control of the average post-synaptic firing rate.") Note that the time the simulation takes to converge on a steady-state output rate varies with both firing rate and the magnitude of STDP, so take care to ensure that your simulations have converged to steady-state.

(3) Transient input rates:

Song et. al. note: "Synaptic changes due to STDP take time to develop, so STDP only regulates the long-term average firing rate, and the neuron remains highly sensitive to transient changes of input firing rates." Verify this claim by describing the sensitivity of the neuron to transient changes in input firing rates. (Intermediate version: use a step-increase in presynaptic firing rate that occurs some time after the simulation has reached steady-state, Advanced version: use sinusoidally varying pre-synaptic rates of different frequencies.) It may be interesting to compare the sensitivity to transients with the sensitivity to long term changes during STDP. What happens when STDP is turned off?

Final notes on speed of simulation (optional):

This assignment shares one annoying feature of running modeling experiments outside of class: they can take forever to run. We've chosen to implement the model in this particular way to both try and make every step as easy as possible to understand and having simulations run to completion in a reasonable time. While you don't need to run the code faster to get a good grade (you just need to plan ahead!), there are a few ways you can try and speed up your simulations or test more parameters in a given chunk of time. This NOT required of the assignment:

1) Multiple instances of MATLAB. The simplest, quickest way to increase the number of tests you do is to open MATLAB several times, and in each instance of MATLAB run a different script. This works because your computer has multiple "cores" each of which is capable of performing computations, which allows you to run multiple computations in parallel. The functions described above do not

explicitly parallelize anything to take advantage of your computers multiple cores. By opening two or three instances of MATLAB you can run multiple different scripts in parallel (each script will automatically run on a different core). You just need to make sure that each instance of MATLAB is running a script with a different name e.g. **plotSimulation1.m**, **plotSimulation2.m** etc. So if your computer has two cores and you have one set of parameters in **plotSimulation1.m** and a different set **plotSimulation2.m** you should be able to get the results of these simulations with different parameters twice as fast as if you had run these two scripts sequentially. (Note: you should be able to google instructions to find out how many cores your computer has).

2) **For advanced MATLAB users only:** Parallelization within MATLAB. The 'parallel computing toolbox' allows each core to be used most simply using a 'parfor' loop, roughly speaking each element being iterated over in the for loop gets its own core. So, for instance each element in the parfor loop could be a separate neuron with its own set of parameters. This is analogous to opening an additional instance of MATLAB with each element of the parfor loop (up to the number of cores your computer has, and with a little less overhead).

3) Many, many other ways! Be creative, we welcome suggestions that speed up the code while preserving clarity.