M. Kramer

**NSCI 360, Fall 2019**

**Computational Assignment 2: Programs that simulate behavior & brain activity**

**DUE SUNDAY, APRIL 7, 9 pm**

As in Assignment 1, computational assignments contain questions for all background levels of programming and mathematical experience (Levels 1, 2 and 3). You will be graded only on those questions that correspond to your agreed-upon background level (if you don’t remember your level, ask me). However, it may be necessary to answer Level 1 questions (or at least be *able* to answer them) in order to answer Level 2 questions. Similarly, Level 2 capabilities may be needed to answer Level 3 questions.

In general, there is one place you should always check first before asking questions of the instructor or even other students, which is the Help documentation that comes with Matlab. For example, to get help using Matlab’s **plot** command, you can type “help plot” or "doc plot" at the command prompt (looks like this: >>), or type “doc” and then search for **plot** in the Search box that you’ll see. Try copying the little snippets of code examples that are included in the doc articles. Paste them into the Command Window and hit Return, and you’ll see what the code does. Then change it a little bit, and see what that does. That seems to be the best, practical way to learn how to do what you need to do.

**Collaboration Policy:** I expect students to try to do these problems individually, and to submit their answers in their own writing (or their own code). Do not copy and paste from any other student. However, you should feel free to discuss problems with other students, and you should feel free to modify code from the doc articles as you see fit, as long as you do that yourself (i.e., don’t copy code from the doc and then send it to another student for their use – they should look that information up themselves).

How to submit your work: Just copy and paste your code, your figures, or your written answers into the spaces provided in this document. Then email to me by the deadline at:

[psimen@oberlin.edu](mailto:psimen@oberlin.edu)

Please name the file “*YourLastName*\_*YourFirstName*\_NSCI360\_HW2”. Files can be in Word, text, Pages (Apple), or PDF format.

**LEVEL 1 ONLY**

**1. *For* loops.** Use a **for** loop to print out the even numbers from 2 to 10. Here’s the tricky part: Inside the loop, the only command should be the name of the variable you use in the **for** command.

*Hint: look up the* ***:***  *(the colon) operator in Matlab. This is really useful in Matlab for creating sequences of numbers, which is what Matlab is all about.*

**Paste your code here:**

**. . . . . . . .**

**2.** **Plot out a signal.** First, create a vector of numbers called **t** representing different moments in time, equally spaced from time 0 to time 10 in steps of size 0.1 (you could think of these numbers as representing time in seconds, or minutes, or whatever). *Hint: use a* ***for*** *loop, or better yet, Matlab’s “:” notation for creating arrays.*  Create a variable of the same size as **t** called **Sig1**, which is equal to the sum of two sine waves with length equal to the length of **t**, *each with different frequencies*. Now do the whole thing over again, and divide time into much smaller time steps of size 0.001.

*Recommendation: use the* ***sin*** *function and look it up for help on its use; look up* ***plot*** *and the* ***:*** *operator.*

**Paste your code here**, and try to place a copy of your output plot figure after the code (save the figure as a pdf file and insert it). To save your figure as a pdf, which you should then paste into this document, use the command:

saveas( gcf, 'Question4a', 'pdf' );

**Paste code here:**

**. . . . . . . .**

**Paste figure here:**

**. . . . . . . .**

**3.** **Plot out a more complex signal.** As in Problem 8, create **Sig2** to be the sum of three sine waves, each with a different frequency and phase, and with a small amount of random noise added to it.

*Hint: Frequency is determined by one of the arguments you provide to the* ***sin*** *command; you can change the phase by adding something to the inputs. “Frequency” is how many times per second a sine wave goes from maximum amplitude to minimum and back to maximum; “phase” specifies how far a signal is from its maximum value at some given time (such as time = 0 seconds). If you still don’t understand, look up what frequency and phase are by searching in the doc or wikipedia. Ask me about them if all of that fails. Finally, use* ***normrnd*** *or* ***randn*** *to create the random noise to add to the signal (one noise sample per time step).*

**Paste your code here**, and try to place a copy of your output plot figure after the code (save the figure as a pdf file and insert it):

**Paste code here:**

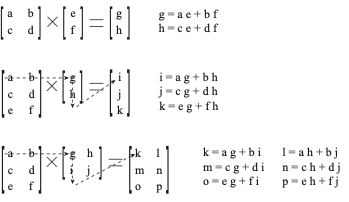
**. . . . . . . .**

**Paste figure here:**

**. . . . . . . .**

**LEVEL 1 AND LEVEL 2**

**4. Matrix multiplication:** If you haven’t taken linear algebra, then you have probably not seen matrix multiplication before. But it is a relatively simple concept, and a very useful one – one could argue that it’s the most important mathematical operation that people ever compute in science. Yet it just depends on doing multiplication and addition in the right order. That’s all there is to it. Below I give you three examples of matrix multiplication. The basic algorithm you use to figure out the answer to a matrix multiplication problem is to go row by row, and multiply each element in a row of the matrix on the left with each element of a column of the matrix on the right. Then you add all these products together. In accordance with this algorithm, the equations to the right of each matrix product below show how the value of each element on the righthand side of the equals sign is computed.



**Steps for Problem 4:**

1. Below I list a Matlab command that defines a 2 x 2 matrix (**A**):

A = [ cos(pi/2) -sin(pi/2); sin(pi/2) cos(pi/2) ]

**A** has two rows and two columns. This problem requires you to multiply **A** against a vector of numbers (**x**). Plot out lines that show how the vector **x** is changed (or as mathematicians say, “linearly transformed”) by matrix multiplication.You should create a vector of 2 rows and 1 column, call it **x**, and then compute **A\*x**.

* Do the **A\*x** operation for 3 different 2-D vector inputs **x** of your choosing (for example, the vector [ 0; 1 ] ).
* Each time, plot the original vector **x** as a line.
* Then, plot it’s transformed version **A\*x**, also as a line.

The way to plot a 2-D vector **x** is as follows:

figure

hold on

plot( [ 0 x(1)], [ 0 x(2) ], '.-'); % Can you figure out why this works?

The command 'figure' opens a new figure window. 'hold on' makes it so that anything you plot in the window stays there after the next plotting command. Otherwise, the next plotting command will erase whatever’s in the figure.

1. In the end, you should have 3 different vectors and their 3 transformed versions plotted in your figure, for a total of 6 vectors. To save your figure as a pdf, which you should then paste into this document, use the command:

saveas( gcf, 'Question4a', 'pdf' );

Now you should be able to paste a figure into this document in any Word processing program.

**Paste the code that you used to generate the figure here:**

**. . . . . . . .**

**Paste the corresponding figure here:**

**. . . . . . . .**

1. Now, create a second figure, and in it, plot the same thing. Only this time, figure out how to rotate the vectors to 180 degrees away from where the original **A** matrix rotates vectors. *(Hint: Note that* ***A*** *was originally created by taking sines and cosines of pi/2.)* At the same time as the rotation, shrink the output vectors to half their size in the first figure. *(Hint: Multiplying a matrix* ***A*** *= [ a b; c d ] by any number, such as 2, leads to a new matrix equal to [ 2a, 2b; 2c 2d ].)*

**Paste the code that you used to generate the figure here:**

**. . . . . . . .**

**Paste the corresponding figure here:**

**. . . . . . . .**

**5. Simple neural network:** Here is the code for a simple neural network model, but it doesn’t quite work yet. Your job is to add just a few lines to this code to make the simulation work. You need to connect up the different components of the model, and you need to define the weights of the connections between inputs and outputs so that a particular function – the AND function – is computed by the network. *(Hint: think in terms of matrix multiplication, as in Problem 4, only here, the matrix will only have two elements and will be a row vector.)*

DEFINITION: The AND function of two binary input values is defined to be 1 if both inputs are 1, and 0 if either of two inputs is 0.

*{NOTE: Although you only have to make your network compute the right output for the four possible binary vectors of two elements (i.e., [0;0], [0;1], [1;0], & [1;1]), your network will also produce perfectly good outputs for non-binary vectors (for example, the vector [ 0.9; 0.2 ]). That’s good, because it appears that the brain doesn’t always represent quantities as 0s or 1s. And what cognitive neuroscientists ultimately want is a good model of the brain.}*

In particular, make sure that the weighted sum of inputs is assigned to the output unit (called "output\_unit") properly. Plot the input vectors, as in Problem 4, for each of the four different possible, binary input patterns: [1,0], [1,1], [0,1], [0,0]. *(Hint: Use the command “hold on” to make sure that whatever you plot does not erase whatever you previously plotted.)*

**Save the following code into its own .m file (select New from Matlab’s File menu, and replace any code in the file that appears in the editor with the following code). Then submit this modified function separately from your word processing document. Call the function “neural\_net.m”.**

function thresholded\_output\_unit = neural\_net( input\_pattern )

% 11/2/12, PAS: This function is supposed to compute the output of a simple neural

% network. I have defined a weight matrix (with undefined weights), and I have

% defined an input vector (input\_pattern) and an output value (output\_unit).

% The output\_unit value represents the 'activation' of two neural network units.

% The user must specify an input pattern (assumed to be one of the following

% vectors: [0;0], [0;1], [1;0], [1;1]). Use the 'figure' command at the command window

% prompt (>>) to open up a new figure window. Then type 'hold on' to allow you to plot

% multiple things in the window. Then call this function 4 times, once each with one

% of the 4 possible input patterns.

output\_unit = 0; % represents the level of activation in a single output % unit, with an initial value of 0. You will overwrite this

% value a few lines down.

% The weight matrix (your job is to figure out what values to assign to a & b

% so as to compute the AND function).

weights = [ a b ]; % <-------- Fill in the values

output\_unit = % <--- Fill in the blank; i.e., assign something to output\_unit

% Plot out the input vector as in Problem 4. You can assign colors by plotting this

% way:

% plot( [0, 1, 2 ], [4, 2, 3], 'r.-')

% The 'r.-' plots everything in red (r), with a little circle at the end (.) and

% with a line connecting the different points (-). You can plot a dashed line with

% (--) instead of (-).

% Use 'r', 'g', 'b' and 'k' for your plotting colors.

% ------ Place your code after this line ------ :

% There is one trick you will need, which is to convert your output\_unit value to a

% 1 or 0. You may find that the output\_unit has an activation that is a

% decimal-valued number, depending on the weight values you chose for a and b.

%

% Here is one way to do the conversion, equivalent to applying a `threshold’ to

% the values of the output\_unit: If the value or activation of a unit is greater than

% the threshold, you set it to 1.

threshold = 2;

% Figure out how to convert graded values for output\_units into 1s and 0s, using the

% threshold value above. Save the thresholded value of the output\_unit in a new vector

% called `thresholded\_output\_unit’. Notice that this variable is what gets returned

% by this function.

% ------ Place your code after this line ------

**6. Modify a spike-train model:** Here is the code for a simple model of action potentials in a neuron. Your job is to increase the rate at which spikes are produced, and to create a spike raster plots of several different epochs of spike recordings. *Hint: look up the* exprnd *function*

function spike\_model();

% 11/2/12, PAS: This function plots out a spike raster for a sequence of spikes. It

% does this for one trial of spike time data. You should refine it so that it plots

% out different spike sequences in different vertical positions.

figure

hold on % Create a figure for plotting, & don’t let stuff get erased.

number\_of\_trials = 1;

number\_of\_spikes = 15;

for i = 1:number\_of\_trials

interspike\_durations = exprnd(1,1,number\_of\_spikes);

spike\_times = cumsum( interspike\_durations );

for j = 1:number\_of\_spikes

plot( [ spike\_times(j), spike\_times(j) ], [0 1] );

ylim( [ -1 2 ]);

% 'ylim' changes the vertical size of the graph. Watch out when

% using this command -- you may find that you plot something and

% can't see it, because it's outside the limits set by ylim. To

% fix such a problem, just call the ylim command again with

% different lower and upper limits.

end

end

The trick to this problem is to modify the trial number variable and the plotting technique so that you can see different spike sequences as different rows of spike blips, one on top of the other. Generate a figure, and save it using the saveas command from Problem 2.

**Separately submit your modified function and call it “spike\_model.m”.**

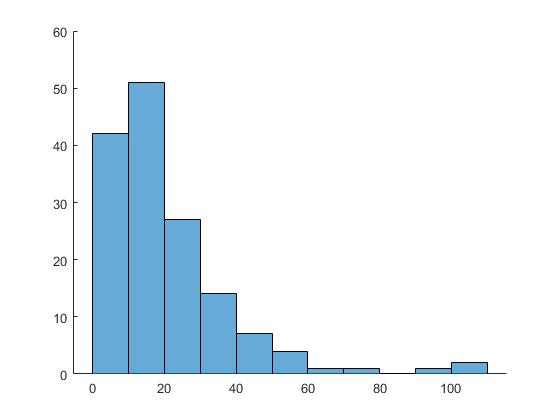
**Paste the corresponding figure here: . . . . . . . .**

**LEVEL 2 AND LEVEL 3**

**7. Response Times generated by a random-walk model:** Using for-loops and a random number generator, compute a random walk: a variable whose initial value is 0, and which gets incremented by both a deterministic increment (0.01) and a random increment (normal with mean 0, std of 0.25) at each “time” step. Stop the process when it exceeds the value 1, or decreases below the value -1. Keep track of the values of this variable, and plot out the resulting trajectory. Also, keep track of *when* (at which time step) the variable exceeds 1, or decreases below -1. Do this 150 times. Then create a histogram of those threshold-crossing times (“response times”, or “RTs”), and save the response time data to a .mat file. Also, save a vector of 1s and -1s that indicate which threshold was crossed first.

**Submit your function code as a separate file called random\_walk.m. Also, paste your histogram here:**

**. . . . . . . .**



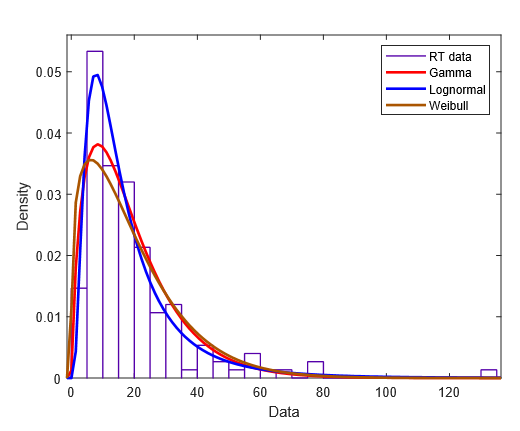
**8. Use Matlab’s distribution fitting tool to fit the response time (RT) distributions:** Load up the RT data from Problem 7, and use the dfittool distribution fitting tool in Matlab to fit a distribution to the data (simply type `dfittool’ at the command prompt >>). Try fitting the gamma, lognormal and Weibull distributions to the data. Which one fits best?

For assistance with using dfittool, use the doc, and search for “Distribution Fitting Tool”. There is a large help page that documents how to use this interactive tool for fitting different probability distributions to the data you’ve generated. But the overall process is pretty simple:

1. Make sure the display type is set to “Density (PDF)”
2. Make sure you have your RT data as a variable in the Workspace
3. Click the Data button
4. In the popup menu, select your RT data for the “Data” field, then click “Create Data Set” (you can close the Data window when this is done)
5. Then, back in the main dfittool window, click NewFit
6. Make sure the Data pulldown menu is listing your RT data
7. Select whichever distribution you like from the pulldown list of many different sorts of probability distributions
8. Then click “Apply”, and observe how well that probability density can be made to fit your data
9. Report the “log likelihood” score that you observe in the bottom of the popup window. The less negative this number is, the closer the fit to the data is.

**Save your fitting window to a pdf file, by selecting from the dfittool’s File menu the Print to Figure option. Then save that figure as a pdf, and paste it here.**

**. . . . . . . .**



**Type here the name of the distribution that fit your data best, and report the log likelihood score of the fit.**

**. . . . . . . .**

Lognormal fit best: Log Likelihood = -568.957

**9. Compute micro-SAT (speed-accuracy tradeoff) plots:** Take the RT data and response type data (1 or -1) from Problem 7, and examine the proportion of 1s and -1s that occur in each of 10 consecutive time windows, or “bins”. Plot out the proportion of 1s in each bin. Is it flat, increasing, or decreasing? Now, what about the average RT for all 1-crossings? Is it different than the average RT for all -1-crossings?

**Type your answers here, and paste a figure that shows your micro-SAT.**

**. . . . . . . .**

**Paste your code here:**

**. . . . . . . .**

%Problem 9: micro-SAT computations%

[sorted\_rt, index] = sort (RT); %sort the RT times

sorted\_Thresholds = zeros(1,150); %init list of thresholds

for i = 1:150 %populate sorted thresholds

sorted\_Thresholds(i) = Thresholds(index(i));

end

figure;

hold on;

xlim([0, 9]);

prop\_pos = zeros(1,10);

prop\_neg = zeros(1,10);

for i = 0:9 %window counter

pos = 0;

neg = 0;

for a = 1:15 %index within window

if sorted\_Thresholds(i \* 15 + a) == 1

pos = pos + 1;

else

neg = neg + 1;

end

total = pos + neg;

end

prop\_pos(i+1) = pos / total;

prop\_neg (i+1) = neg / total;

end

plot(prop\_pos, 'r.-') %plot prop of 1s

negOneCross = 0;

numNegCrosses = 0;

OneCross = 0;

numCrosses = 0;

for i = 1:150 %average RT for 1 trials and -1 trials

if sorted\_Thresholds(i) == 1

OneCross = OneCross + sorted\_rt(i);

numCrosses = numCrosses + 1;

else

negOneCross = negOneCross + sorted\_rt(i);

numNegCrosses = numNegCrosses + 1;

end

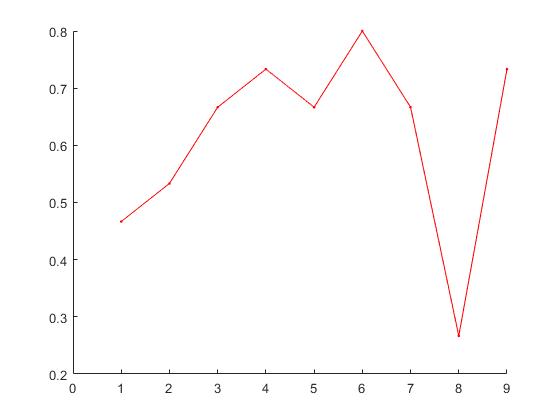
end

avgOneCross = OneCross / numCrosses;

avgOneNegCross = negOneCross / numNegCrosses;

disp(avgOneCross)

disp(avgOneNegCross)



Average 1 crossing RT: 20.5495 Average -1 crossing RT: 20.2034

**LEVEL 3 ONLY**

**10. More complex neural network:** For this problem, you should complete Problem 5 first. Then the problem should be modified so that there are 3 input values (a 3-dimensional vector), and two output units. Think of each output unit as a category (e.g., if the output is (1,0), then the system is “thinking of” a cat; if it is (0,1), it is thinking of a dog). The 3 input values (a column vector) can be thought of as the values of different visual features (facial shape, body shape, type of movement). Do the thresholding operation with a threshold of 2 again. Your main job is just to design the weight matrix necessary to compute a function that corresponds to the following “truth table”:

IN1 IN2 IN3 OUT1 OUT2

0 0 0 0 0

0 0 1 0 1

0 1 0 0 1

0 1 1 0 1

1 0 0 0 0

1 0 1 1 0

1 1 0 1 0

1 1 1 1 1

For your information, what you are doing in this problem is essentially finding a plane that divides the three-dimensional input space into two categories, and classifying input vectors that exist on one side of the plane as a “0”, and those that fall on the other as a “1”. And you are doing this separately for each output unit. Note that it is possible (in fact, easy!) to define functions that are not “linearly separable” in this way. One such linearly inseparable function of two binary variables, call them A & B, is the exclusive-or (XOR) operation, defined as follows: if A is 1, or B is 1, but not both, then the output is 1, otherwise the output is 0. When this was recognized in the 1960s, early neural network research took a huge hit in popularity, funding, and overall progress. We talked about the solution to this problem in class (i.e., using more than just two layers of units), but we will not implement that solution now.

**Save your code into its own .m file. Call it “complex\_net.m”. Make it accept 3-dimensional input vectors, representing how cat-like (how close to 0) or dog-like (how close to 1) a given animal is in terms of the three features. Also, try out some non-binary input vectors, and see how they get classified (they too should produce network decisions that an input animal is a cat, a dog, neither, or even both, whatever that means.)**

function thresholded\_output\_unit = complex\_net( input\_pattern )

output\_unit = 0;% represents the level of activation in a single output unit, with an initial value of 0.

%You will overwrite this value a few lines down.

% The weight matrix

weights = [ 1.6 0.5 0.5; -0.5 2.4 2.4 ];% <-------- Fill in the values

output\_unit = weights \* input\_pattern; % <--- Fill in the blank; i.e., assign something to output\_unit

% Figure out how to convert graded values for output\_units into 1s and 0s, using the

% threshold value above. Save the thresholded value of the output\_unit in a new vector

% called `thresholded\_output\_unit’. Notice that this variable is what gets returned

% by this function.

threshold = 2;

% ------ Place your code after this line ------

if output\_unit(1) > threshold

output\_unit(1) = 1;

else

output\_unit(1) = 0;

end

if output\_unit(2) > threshold

output\_unit(2) = 1;

else

output\_unit(2) = 0;

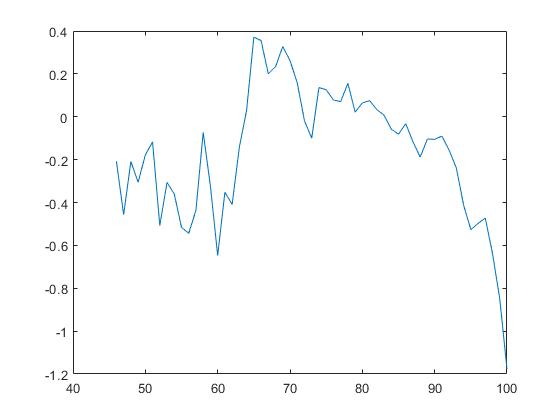
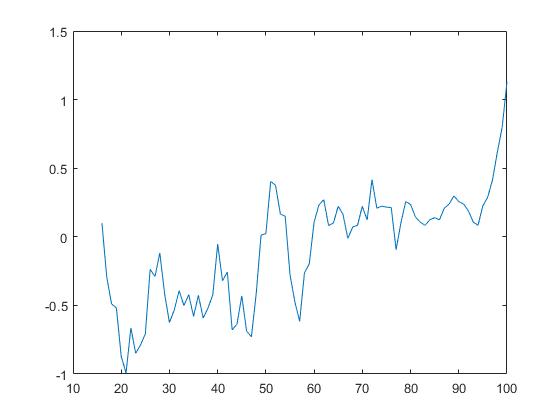
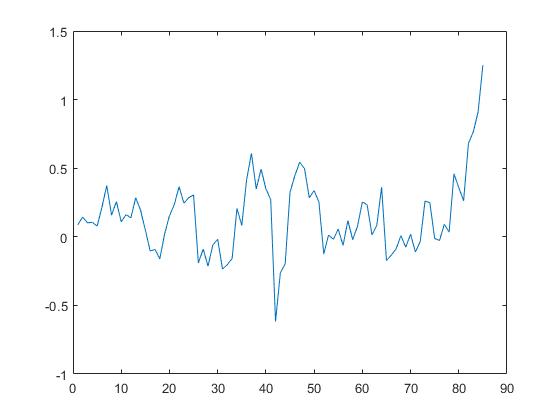
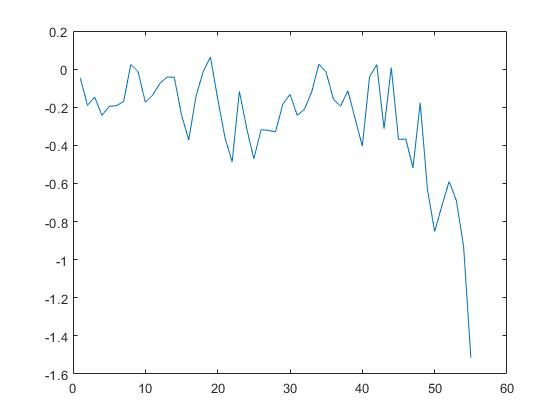
end

thresholded\_output\_unit = output\_unit;

**11. Response-locked vs. stimulus-locked averaging:** For this problem, you first need to save several random walk trajectories from Problem 7. You then need to create a single, average signal that averages at each time step across all the separate random walk trials. Do this separately for those trajectories that terminate at the 1-boundary, and then for those that terminate at the -1-boundary. First, make sure that all the trajectories *begin* at the same time: this is an example of stimulus-locked averaging. Second, make sure that all the trajectories *end* at the same time: this is an example of response-locked averaging. These two averages will not be the same, since the random walk takes different amounts of time to cross threshold on different trials.

**Submit your function code as a separate file called “averaging.m”. Have it create four figures (one each for the stimulus-locked averages of 1-crossing and -1-crossing trajectories, and one each for the response-locked average of 1 and -1-crossing trajectories). Paste those as pdf figures here:**

**. . . . . . . .**



**12. Rate-varying Poisson processes:** Create a spike-generating process as in the previous assignment (note the simple way you can do this that is shown in Problem 6). However, do it in such a way that the rate of spiking changes three times over the course of a simulation (rising and falling over time). Create many simulations of such a process, with the spike-rate changing at roughly the same time on each simulation. Then, create spike rasters as requested in Problem 6. Finally, create a spike-time histogram. This histogram should show how many spikes occurred within a set of small time windows across all simulations.

Break the output figure into two “subplots”, by using the “subplot” command. By using the following commands, you are telling Matlab first to draw into the top subplot out of a column of two subplots. The second command tells it to draw into the bottom subplot. Spike rasters should go into the top subplot; histograms should go into the bottom subplot.

**Submit your function code as a separate file called “non\_homog\_Poisson.m” (which stands for “non-homogeneous Poisson”, meaning that the rate of Poisson spikes changes over time). Have the function create a single figure. Paste the pdf figure here:**

**. . . . . . . .**

NOTE: I was unable to get the subplot command working properly, the rasterplot would disappear as soon as I called the subplot command regardless of xlim and ylim being preset.

