DUE 12/11, 21:00

NSCI 360, Fall 2019

Computational Assignment 3: Brain/behavior simulations, Experiments, and Data Analysis

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" at the command prompt (looks like this: >>), or type "helpdesk" 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 Help 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 Helpdesk articles as you see fit, as long as you do that yourself (i.e., don't copy code from the Helpdesk 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
Please name the file "*YourLastName_YourFirstName*_NSCI360_HW3". Files can be in Word, text, Pages (Apple), or PDF format.

LEVEL 1 ONLY

1. 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.

LEVEL 2 ONLY

2. Linear regression. Follow the instructions for the example in the Matlab helpdesk on the 'regress' function. This example shows how you can predict the fuel-efficiency (miles per gallon, or MPG) of a car based on its Weight and the Horsepower. You do this by finding a set of weights that produces weighted sums of Weight and Horsepower values that are as close as possible to the actual MPG values recorded for a set of 100 cars (specifically, that minimize the sum total of the differences between predictions and observations, squared – a.k.a., the "sum of squared errors" or "SSE"). The data can be found in carsmall.mat, which should already be installed along with Matlab (if it's not, you can also find it on Blackboard in the Matlab Assignments folder).

The example code in the regress.m helpdesk section is as follows:

```
% Load data on cars; identify weight and horsepower as predictors, mileage
% as the response (or "prediction"):
load carsmall
x1 = Weight;
x2 = Horsepower; % Contains NaN data
y = MPG;


% Compute regression coefficients for a linear model with an interaction
% term:
X = [ones(size(x1)) x1 x2 x1.*x2];
b = regress(y,X) % Removes NaN data



% Plot the data and the model. There are some lines of code here that may
% seem very obscure. The use of 'mesh' and 'meshgrid' are to help define
% a matrix of all possible Weight and Horsepower combinations (X1FIT and
% X2FIT). Once you have that set of possible combinations, you can plot
% a surface defined by the weights.
scatter3(x1,x2,y,'filled')
hold on
x1fit = min(x1):100:max(x1);
x2fit = min(x2):10:max(x2);
[X1FIT,X2FIT] = meshgrid(x1fit,x2fit);
YFIT = b(1) + b(2)*X1FIT + b(3)*X2FIT + b(4)*X1FIT.*X2FIT;
mesh(X1FIT,X2FIT,YFIT)
xlabel('Weight')
ylabel('Horsepower')
```

```
zlabel('MPG')
view(50,10)
```

Once this is done, try to compute the SSE between the fitted surface and the actual MPG data values. *(Hint: Find the difference between the predicted MPG – i.e., the weighted values of the Weight and Horsepower data points – and the actual MPG values. Also, instead of using the function 'sum' to add up the prediction errors, use 'nansum', since there are some values equal to NaN (Not a Number) in the car data, which represent missing data points.)*

Paste your code for computing the SSE here:

. . . .

Now try to fit a simpler model to the data – one that does ***not*** contain the "interaction term" (i.e., Weight x Horsepower). Is the fit of this simpler model better or worse?

Enter your fit error values (the SSE) for both the complex model containing an interaction between Weight and Horsepower values, and for the one without an interaction term.

Complex model SSE:

Simple model SSE:

Plot out the best fitting surface for each model, and paste both of those figures below. Set the title of each plot by using the command "title", e.g.:

```
title('Simple Model').
```

. . . .

Finally, consider the independent variables (Weight and Horsepower), the dependent variable (MPG), and the predictors (b) as defining a neural network. Which components of the regression problem correspond to the input layer, weights and output unit of a simple neural network?

Independent variables correspond to (enter answer here):

The dependent variable corresponds to (enter answer here):

Predictors correspond to (enter answer here):

LEVEL 3 ONLY

3. Compute time-solutions of a dynamical system / neural network

In this problem, you will work with a classic equation for simulating the average firing rate (V) of a population of neurons. Eq. 1 specifies the rate (dV/dt) at which V changes over time:

$$dV/dt = -V + f(I) \qquad (1)$$

Use the following definition of the sigmoidal function f:

$$f(I) = 1./(1+\exp(-4*(I - 0.5)));$$

Eq. 1 only *implicitly* defines what the firing rate V is; but Eq. 1 can be "solved" to give V as an explicit function of t.

According to Eq. 1, V changes over time in response to its input (I), and $I(t)$ itself can change over time. Just to let you know, Eq. 1 specifies a system that changes *continuously* as time progresses – there is no "smallest" unit of time for a differential equation. But for simplicity, you can assume that time *does* change in tiny little time steps of size dt (for example, dt = 0.001). When you make that assumption, Equation (1) is essentially the same as the following equation:

$$V\_new = V\_old + (f(I) - V\_old) * dt \qquad (2)$$

In Equation (2), you compute a new value for V (V_new) by adding a tiny increment defined by the rate of change dV/dt in Equation (1), multiplied by the time step size, dt. If dt is very small, then Eq. 2 approximates Eq. 1 very closely, but making dt small makes it take a long time to simulate V_new changing over some given time window.

It turns out that Matlab is very good at efficiently solving these differential equations, so if you know what the initial state of a system is, you can figure out (almost) exactly what that system will do in the future. That is, Matlab can find the solution as long as the system you are modeling is "deterministic", or noiseless, and for this question, we will indeed assume determinism.

Look up the function 'ode45' in the Helpdesk to find out how to get Matlab to solve the differential equation. Below, I have pasted images of two key examples from this help-page on solver functions for O.D.E.'s (which stands for "ordinary differential equations"). The examples use the notation y' in place of dy/dt. But if you want to copy the Example code and paste it into Matlab, which I always recommend, you'll need to go to the help page itself.

Example 1

An example of a nonstiff system is the system of equations describing the motion of a rigid body without external forces.

$$y'_1 = y_2 y_3 \qquad y_1(0) = 0$$
$$y'_2 = -y_1 y_3 \qquad y_2(0) = 1$$
$$y'_3 = -0.51 y_1 y_2 \qquad y_3(0) = 1$$

To simulate this system, create a function `rigid` containing the equations

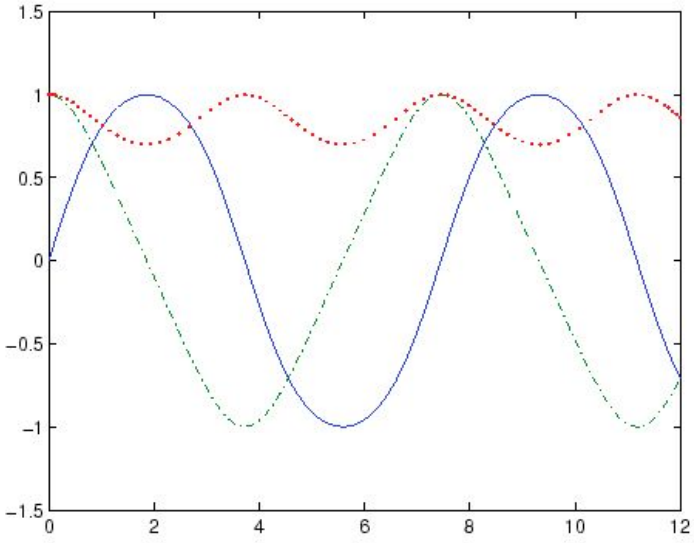
```
function dy = rigid(t,y)
dy = zeros(3,1);    % a column vector
dy(1) = y(2) * y(3);
dy(2) = -y(1) * y(3);
dy(3) = -0.51 * y(1) * y(2);
```

In this example we change the error tolerances using the `odeset` command and solve on a time interval `[0 12]` with an initial condition vector `[0 1 1]` at time `0`.

```
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4 1e-5]);
[T,Y] = ode45(@rigid,[0 12],[0 1 1],options);
```

Plotting the columns of the returned array `Y` versus `T` shows the solution

```
plot(T,Y(:,1),'-',T,Y(:,2),'-.',T,Y(:,3),'.')
```



Implement Eq. 1 as an ODE function called **NN_ODE.m**, following the examples. You can consider the input to be a single number called "I" that changes over time. Use a sine wave as the signal, over a time span ranging from time 0 to time 100. One way to do that is to compute the sine of the current time within the ODE function, and incorporate that value as the input term of Eq. 1.

Example 3

This example solves an ordinary differential equation with time-dependent terms.

Consider the following ODE, with time-dependent parameters defined only through the set of data points given in two vectors:

```
y'(t) + f(t)y(t) = g(t)
```

The initial condition is `y(0) = 0`, where the function `f(t)` is defined through the `n-by-1` vectors `tf` and `f`, and the function `g(t)` is defined through the `m-by-1` vectors `tg` and `g`.

First, define the time-dependent parameters `f(t)` and `g(t)` as the following:

```
ft = linspace(0,5,25); % Generate t for f
f = ft.^2 - ft - 3; % Generate f(t)
gt = linspace(1,6,25); % Generate t for g
g = 3*sin(gt-0.25); % Generate g(t)
```

Write an M-file function to interpolate the data sets specified above to obtain the value of the time-dependent terms at the specified time:

```
function dydt = myode(t,y,ft,f,gt,g)
f = interp1(ft,f,t); % Interpolate the data set (ft,f) at time t
g = interp1(gt,g,t); % Interpolate the data set (gt,g) at time t
dydt = -f.*y + g; % Evalute ODE at time t
```

Call the derivative function `myode.m` within the MATLAB `ode45` function specifying time as the first input argument :

```
Tspan = [1 5]; % Solve from t=1 to t=5
IC = 1; % y(t=0) = 1
[T Y] = ode45(@(t,y) myode(t,y,ft,f,gt,g),Tspan,IC); % Solve ODE
```

Plot the solution `y(t)` as a function of time:

```
plot(T, Y);
title('Plot of y as a function of time');
xlabel('Time'); ylabel('Y(t)');
```

Now you can call the ode45 function, as in Example 3, and plot the resulting solution, V(t). Send me a function called NN.m that includes the sine wave definition and the code that calls the ODE function, NN_ODE.m. It should also plot out the sine wave input and the output V.

Paste the plot here:

. . . .

In words, explain how the output V(t) is related to the input, I(t), paying particular attention to how it is different from I(t):

. . . .