Bioen 485/585: MATLAB ODE Refresher/Tutorial (optional)

Open the matlab program from the start menu. You will see a window that has a command line. If you haven't used matlab at all, so that the instructions here don't make sense to you, call over an instructor. If you feel very comfortable with MATLAB, just read this to see if there is anything new, but if you feel rusty, take time to do the exercises.

Getting help.
Go to "Help>Product Help" and search for "plot", or write 'doc plot' on the command line, and the help window will pop up to that page with instructions and examples of how to use that function. Look at the left panel of the help window and note that you can also browse through sets of related functions.

Scripts, Equations, arrays, and Plotting equations:
Exercise 1: Plot the function $y(t) = e^{-at}\cos(bt)$ between t=0 and t=2 at 0.01 increments for a=2 and b=10. (see solutions at the end of the tutorial).

Open a new m-file, and write on the first line:

```
% plotdecay
```

Save the file as 'plotdecay.m'. Now write the rest of a script that will plot the function, save it, and write 'plotdecay' on the command line (don't write the '.m'), which will run through all the lines of your script in order. First try it on your own using the following hints.
Creating variables and writing simple equations
var = [number]; ←assigns a number to a letter variable

var = [first number]:[increment]:[last number] ←creates an array of evenly spaced numbers between a min and max

Mathematical Operators

[array1].*[array2] ←(.*) does point-wise multiplication; arrays must be same size.

[array1]*[array2] ←(*) does matrix multiplication; inner dimensions must be same.

exp([number]) ←$e^{[number]}$

Simple plots and axes labeling

figure([figure number]) ←initializes a new figure

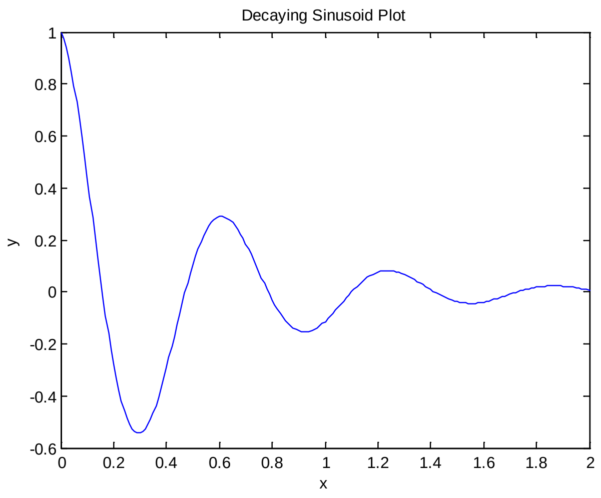
plot([t1 vector],[y1 vector],[t2 vector],[y2 vector]) ←plot multiple sets of variables on the same graph

alternatively: plot([t1 vector],[y1 vector])
hold on ←allows plotting to the same graph
plot([t2 vector],[y2 vector])

xlabel('[x variable name]');
ylabel('[y variable name]');

title('[plot title]');

You should get something like the figure to the right. If you have problems, note that you can write a variable name on the command line and MATLAB will return the value of that variable. You can also test a line of code on the command line before you add it to your script. These can help with debugging, since it helps you identify where things went wrong. In case you need more help, one solution is below:



```
% plotdecay
clear all; close all
%in general, start with these to clear all variable and close all
% figures, so your script is not affected by work you did before.

%define constants
a = 2;
b = 10;
t = 0:.01:2;

%write equation
y = exp(-a*t).*cos(b*t);

%plot
figure(1)
plot(t,y)
xlabel 'x';
ylabel 'y';
title 'Decaying Sinusoid Plot';
```

<u>Functions and loops</u>

In the previous section, you wrote a script and noted that your variables were still defined in the workspace (which is what you access in the command line) after the program quits, even if it quits with an error.

The other type of m-file is a function. MATLAB has lots of packaged functions (such as 'plot') but you can also write your own. Functions don't know the variables that were defined when it was called, and vice versa, with the exception of inputs and outputs that pass to and from the function respectively.

Convert your script to a function by deleting the 'clear all; close all' phrase, and the lines that define a and b, and adding a line that defines the m-file as a function and indicates that a and b will be defined by the user as inputs to the function:

function plotdecay(a,b)

This must be the FIRST line of the m-file, or you will get an error. This means you cannot define a function inside a script (but you can call one, of course.) Save the file, and then write on the command line 'plotdecay(2,10)', and you should see the same plot.

Now write 'plotdecay(1,20)' and you should see a different decay pattern. Note that the new plot replaced the old one. Write 'hold on' after the plot command, save your function and call it a few more times with different inputs. You should see all the curves together in one figure. If you want to clear it, write 'close all' to close the figure or 'hold off' to replace all previous curves with the new one.

Now consider if you want to make a figure that shows the effect of changing a variable. You can do this by altering your function to use a loop to make a curve for each value of a or b. To do this, embed your 'y=…' command and your 'plot…' command in a loop. you can use 'for', or 'while' to make loops. (doc each to find out more if needed.)

example solution, complete with examples of how to label plots with variable label values:

```
function plotdecay2(a,b)
% plotdecay2
t = 0:.01:2;
 colors = {'k','b','c','g','y','r','m'};
close all;
figure(1);
for i =1:length(a)
y = exp(-a(i)*t).*cos(b(1)*t);
plot(t,y,colors{i}); hold on;
end
% create a legend
for i = 1:length(a)
A{i}=strcat('a = ',num2str(a(i)));
end
legend(A);
xlabel 'x';
ylabel 'y';
title (strcat('Decaying Sinusoid Plot for b = ',num2str(b(1))));

figure(2);
for i =1:length(b)
y = exp(-a(1)*t).*cos(b(i)*t);
plot(t,y,colors{i}); hold on;
end
% create a legend
for i = 1:length(b)
B{i}=strcat('b = ',num2str(b(i)));
end
legend(B);
xlabel 'x';
ylabel 'y';
title (strcat('Decaying Sinusoid Plot for a = ',num2str(a(1))));
```

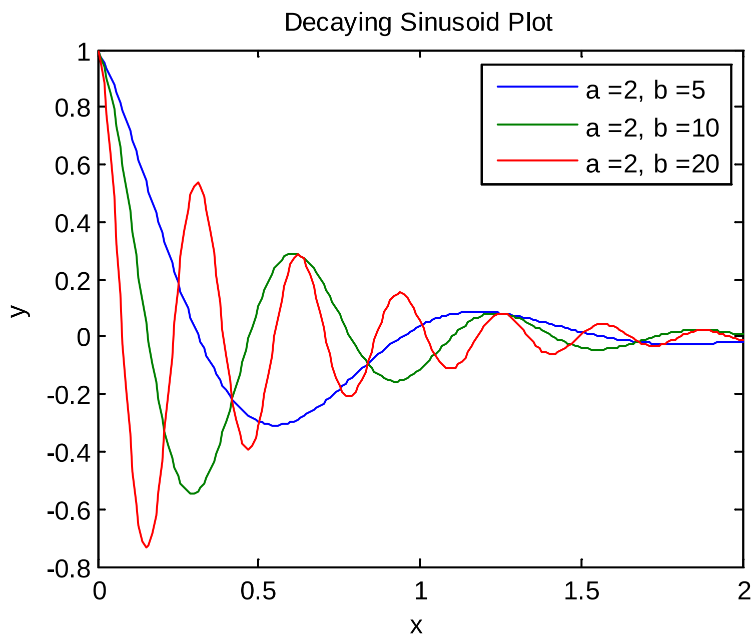
Now call 'plotdecay([1,2,4,8],[5,10,20])'. You should see two figures, each showing the variation in one variable while the other variable keeps its first value.

Note that you can often avoid loops by using array multiplication, which is much faster computationally, but you do need to make sure the inner matrix dimensions agree. Consider the following, and then call it with 'plotdecay3([2,2,2],[5,10,20])'

```
function plotdecay3(a,b)
% plotdecay3
if length(a)~=length(b)
    error('input variable arrays must be same length')
end
t = 0:.01:2;

close all;
figure(1);
y = exp(-a'*t).*cos(b'*t);
plot(t,y)

for i = 1:length(a)
A{i}=strcat('a = ',num2str(a(i)), ', b = ',num2str(b(i)));
end
legend(A)
xlabel 'x';
ylabel 'y';
title 'Decaying Sinusoid Plot';
```



Solving Differential Equations.

You may have used a differential equation solver before. There are three useful things you will need for the first weeks of this class. 1) understand what the ODE solver does. The best way to do this is to write a simple code to solve an ODE yourself, which you can do below. 2) know the syntax for using MATLAB's ODE solving functions. 3) know how to select an ODEsolver and options to efficiently and accurately solve various problems.

Numerical Integration of a differential equation. Consider a differential equation in one variable. This will have the general form $dy(t)/dt = f(y(t))$. If you know you are at position y(t) at time t, solving a differential equation numerically means you will calculate your position y(t+Δt) where Δt is a small increment in time. Then you will do this again and again to step forward in time, predicting the future. Remember from calculus that the definition of a derivative is:

$dy/dt = \lim_{\Delta t \to 0} \dfrac{y(t+\Delta t) - y(t)}{\Delta t}$. Thus, a "first-order" estimation of the derivative is $dy/dt \approx \dfrac{y(t+\Delta t) - y(t)}{\Delta t}$, with the estimation becoming for more accurate as Δt approaches zero. Multiply both sides by Δt to get $\dfrac{dy}{dt}\Delta t = y(t+\Delta t) - y(t)$. Then recall that $dy/dt = f(y(t))$, where the latter is some known function, so you have $f(y(t))\Delta t = y(t+\Delta t) - y(t)$. Now, everything in that equation is a known except y(t+Δt), your desired value, so put it on the left side and everything else on the right:

$$y(t+\Delta t) = y(t) + f(y(t))\Delta t$$

You can write a MATLAB script that solves for y_n iteratively (using a loop of your choosing) for the following differential equation:

$$dp_1(t)/dt = k_{01} - p_1(t)(k_{01} + k_{10})$$

with $k_{01} = 50$ sec^{-1}, $k_{10} = 2$ sec^{-1}, and the initial value of $p_1 = 0$. Solve for the time period from 0 to 1 second with a time step Δt = 0.001 seconds. This is called numerical integration of a differential equation. This equation models the probability of a channel being open as a function of time, where the channel is originally closed but the conditions change at time 0. Plot the proportion of channels open (p_1) against time. Remember to label your axes.

You should be able to solve this particular differential equation analytically to obtain $p_1(t) = \dfrac{k_{01}}{k_{01} + k_{10}}(1 - \exp(-(k_{01} + k_{10}) * t))$, since you've taken differential equations. Plot the analytic together with your numerical solution. Numerical integration should yield a close but not exact solution. Now try a larger step size, such as Δt = 0.01 or worse yet, 0.04.

Solution:

```
clear all
close all

%define constants
k01 = 50;
k10 = 2;
```

```
%experiment parameters
totalt = 1;
deltat = 0.001; %the time step

%establish initial value of p1 and the vector of time values
p1(1) = 0;
t = 0:deltat:totalt;

%for loops to complete iterative stepping for two different
conditions
for i = 1:totalt/deltat
p1(i+1) = p1(i)+deltat*(k01-p1(i)*(k01+k10));
end

%analytical solution
p = k01/(k01 + k10) * (1-exp(-(k01 + k10)*t));

%plot p vs t analytical and interative euler and label the graph
plot(t,p1, t,p)
xlabel 'Time(s)';
ylabel 'Fraction of Channels Open';
title 'Channel Opening vs. Time';
legend 'iterative soln' 'analytical soln';
```

MATLAB ODE solvers. MATLABs ODE solvers (such as ode45) do something similar to this, but they all use smart time stepping, which means they guess a time step, get the new value, and check how much the derivative has changed since the previous. If it changed too much, they use a shorter time step, but if it changed too little, they use a longer one. There is a default requirement for accuracy, or you can specify one. The different solvers use different algorithms that are more efficient for different kinds of functions, but all use essentially the same syntax:

To solve a differential equation, first you need to define the differential equation itself in a different function. For example, for the previous problem, define a function in a separate file. The inputs go in order: time, y-value, parameter list.

```
function dy = flowsolve(t,y,k01,k10)
%equation for change in y using passed k01 and k10 values
dy = k01-y*(k01+k10);
end
```

Now add the following to the end of your script in which you solved the ODE analytically and numerically already:

```
timeframe = [0,totalt];
IC = p1(1);
options = [];
[T,Y] = ode45(@ODEname, timeframe, IC, options, k01, k10);
hold on;
plot(T,Y,'r')
```

The output will be a time vector and a vector of positions. You can plot(T,Y) and you should get the same result as previously. Try this. Now use the comments below to change methods to solve your problem above.

Avoiding Numerical Artifacts with MATLAB ODE solvers. All numerical integrations have some degree of numerical error. We refer to the error as a numerical artifact if the errors is significant enough to affect your conclusions or mislead or distract someone viewing the result. It is necessary to make sure that your calculations have no such artifacts, and are sufficiently efficient to meet your purposes. The best method to achieve these goals depends on the problem.

You can identify numerical artifact in any of three ways:

1) You may suspect an artifact if the graph or result just doesn't seem right; it may be sharp-edged when it should be smooth, smooth when it should be sharp, or oscillate when it shouldn't. However, this method requires that your intuition be correct.
2) The solution may not match something you know from an analytic solution. However, this method requires that your analysis be correct.
3) The solution does not **converge** when you compare different numeric methods. Convergence means the solutions become the same as you vary the method to be more accurate. If methods 1) or 2) lead you to suspect an artifact, testing for convergence is a good way to confirm and solve the problem.

The convergence method involves changing both the algorithm and the accuracy:

a. Trying different algorithms: In MATLAB, this refers to using different ode solvers. This is not a numerical methods class, so you don't need to know how the different algorithms work, although learning this can help you pick the best algorithm for your purpose. The MATLAB documentation provides useful advice on when to use different ones. You will also want to know that you should try a stiff solver (they have an 's' in the name, like ode15s) if the regular solver runs too slowly at the required accuracy. Stiff problems are ones where part of the solution changes quickly and part slowly, and the stiff solvers use a different time step for the different parts of the solution (roughly).

b. Increase the accuracy of the solver for a given algorithm by setting the options using `odeset`. You may want to change the **relative tolerance**, the **absolute tolerance**, or the **maximum time step**. Read the documentation for odeset, but here is a reminder of the syntax used to increase the accuracy by setting the relative tolerance to 1e-4, (the default is 1e-3): `options = odeset('RelTol', 1e-4);`

c. If the only problem is that the output is rough-looking, you can change the number of output points by passing a vector of more than two points for your time span. This also affects the maximum time step for integrations, so can increase accuracy of integration.