Midterm A1 / A2 / B1 / B2
Section 4 / 2 / 4 / 2

November 9, 2014

Problem 2 / 3 [10pt]: Finish the following function. For input: f is **a** function so that $f(x_j)$ returns the value of **an** interpolating polynomial at x_j and x is a row vector of values. For output: y is a row vector of the values $f(x_j)$ for each element in x. Assume that $f(x_j)$ only accepts *one value at a time*.

function y = evaluateF(f, x)

end

This is a *very* tricky problem. When I sat down to take this exam, I almost missed this one completely. In the previous problem, you either wrote the function $f(x)$ as a linear system (A1 / A2) or you found a Lagrangian polynomial named $f(x)$ (B1 / B2). In any case, you've done a lot of work to accurately describe $f(x)$. If this problem were stated like this

Problem 2 / 3 [10pt]: Finish the following MATLAB function named "evaluateF." For input: g is a function so that $g(t_j)$ returns the value of **an** interpolating polynomial at t_j and t is a row vector of values. For output: y is a row vector of the values $g(t_j)$ for each element in t. Assume that $g(x_j)$ only accepts one value at a time.

```
function y = evaluateF( g, t )


end
```

then what we were asking may have been a bit more clear. The first part of solving this problem is recognizing that the f in this problem is *not necessarily* the same f from the last problem. With that in mind, let's look back over the statement of **Problem 2 / 3** so that we can identify all of our given information, and decipher what output we want to produce.

The problem statement tells us that we are given two input variables. One is a function f which takes only one value at a time, the other is a row vector, x. Since f takes only one value at a time, we can't do this $f([1, 2])$, or this $f([x_1, x_2, x_3])$, or this $f([x_1, x_2, \ldots, x_n])$, or so on. The best we can hope to do is throw one value into f at a time, like this $f(1), f(2)$, or this $f(x_1), f(x_2), f(x_3)$, or this, $f(x_1), f(x_2), \ldots, f(x_n)$.

We also know that x is a row vector. But we don't know how many entries it has. So as I'm getting my head around this problem, I'm just going to write down what x looks like in general. We have

$$x = [x_1, x_2, \ldots, x_m]$$

for some integer m with $m \geq 1$. That is, x may have 100 entries, it may have 1000 entries, or it may have only one entry. In any case, we want our code to be able to take a given row vector (of any length) and compute ... what? Well, the problem says that the output, y, needs to be a row vector of values $f(x_j)$ for *each* element in x. Since I just wrote $x = [x_1, x_2, \ldots, x_m]$ above, it's pretty obvious how y will look. We'll get

$$y = [f(x_1), f(x_2), \ldots, f(x_m)]$$

Stop! After doing some analysis, let's review the problem statement and make sure we're on the right track. Have we accounted for every bit of information given in the statement above? We've looked at x, f, and we've got a descent candidate for y. Since the statement tells us that y is a row vector of the values $f(x_j)$ for *each* element in x, and since x has the form $x = [x_1, x_2, \ldots, x_m]$, then we can conclude that the vector y (which we want to make our output) needs to have the form $y = [f(x_1), f(x_2), \ldots, f(x_m)]$. But we know that we can't simply do this

$$y = f(x)$$

because $x = [x_1, x_2, \ldots, x_m]$. We were told *explicitly* that $f(x) = f([x_1, x_2, \ldots, x_m])$ can **not** work because f takes only one value at a time. Then we'll need to come up with another way to fill a row vector y with the m values $f(x_j)$. Notice that we know how big our row vector y has to be. Each entry in y corresponds to a given entry in x. Then y and x must be row vectors of the same size. In summary, we have

$$x = [x_1, x_2, \ldots, x_m]$$

and we want

$$y = [f(x_1), f(x_2), \ldots, f(x_m)].$$

Stop! Did we get all of the information we need from the problem statement? I think so. The two relations above describe the entire problem. We have a row vector with m entries and we want to construct a row vector y with m entries.

How big is m? Well, it could be 1. It could be 2. It could be $2^{16} - 1$. *We don't know how large m is.* The only thing we *do* know is that m is the number of elements in x. On page 2 of the exam, you were given the MATLAB command `size(x)`. The exam states that for a vector `x`, we have `D = size(x)` , where `D` is the number of elements in `x`. This is actually inaccurate. If `x` is a vector or matrix (of any size), then `D = size(x)`, where `D` is a 1×2 row vector, whose entries are the dimensions of `x`. For example, suppose that `x` is the row vector given by `x = [1, 3, 5, 7]`. Then `x` is 1×4, so that `D = [1, 4] = size(x)`. If `x` were a matrix given by `x = [1, 2, 3; 4, 5 6]`, then x is 2×3 so that `D = [2, 3] = size(x)`. There is a function, `length`, which has the property that if `x` is is a row vector or a column vector, then `L = length(x)` where `L` is the number of elements in `x`. We'll come back to this below.

Okay. Let's consider our given vector $x = [x_1, x_2, \ldots, x_m]$. There are two ways to get the value m from x by using the function `size`. One way it to assign `size(x)` to a variable like this `D = size(x)`. Then we have `[1, m] = D`, so that `m = D(2)`, or equivalently, `m = D(1,2)`. A better way to access this number is by throwing a second argument into `size`. Since we know that x is a row vector, then we want its second dimension for m. To do this, we can write `m = size(x,2)`. If we wanted to assign the first dimension of x a variable named `n`, we would write `n = size(x,1)`. For grading purposes, I treated your usage of `size` as either returning a vector or returning number of entries of x depending on the context of your code. That being said, I took off points for *inconsistent* usages of variables and functions in your code.

Earlier, I noted that we did a lot of work in the previous problem in order to get information about some $f(x)$, and that the f in this problem is not *necessarily* the same f. Maybe your boss gave you some data and told you to work on getting an f to fit that data. Then later that morning, you get an email from your boss with a vector x and some function f with some weird stipulations on it. It may or may not have anything to do with the f that she told you to work on earlier! I agree that this is a very tricky problem, but that is precisely the point. Computational science and math require careful attention to detail.

Suppose that the following code is the mysterious f, which happens to have nothing to do with your previous work.

```
function out  = f( z )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   FUNCTION:    f(z)
%
%      INPUT:     z    a real or complex scalar value with
%                      real part larger than -1
%
%     OUTPUT:    out   Gamma(x+1)
%
%DESCRIPTION:
%
% Given a scalar, z, f(z) returns the value
% of the Gamma function evaluated at z+1.
%
% The Gamma function is the continuous
% extension of the factorial function to real
% (and complex) numbers. For any positive
% integer, n, we have
%
%     Gamma(n - 1) = n!
%
% You can think of Gamma as the function
% which "fills in" all the points between
% the integers. For instance, we know that
%
%     3! = 3 * 2 * 1
% and
%
%     4! = 4 * 3 * 2 * 1
%
% But how do we get a (3.5)!, or a (pi)! ?
% It turns out that if z is one of those
% "missing points" between the integers,
% we can compute
%
%     Gamma(z) = (z-1)!
%
% But I don't want that. I want a new
% function f(z) which gives me
%
%         f(z) =  z!
%
% So I want f(z) = z! = Gamma(z+1).
```

```
% Looking up the formula for the Gamma
% function on wikipedia, I find that
%
%Gamma(z) = integral(@(t) t.^(z-1).*exp(-t),0,Inf)
%
% Then I want
%
% output = integral(@(t) t.^(z).*exp(-t),0,Inf) = Gamma(x+1)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

out = integral(@(t) t.^(z).*exp(-t),0,Inf);

end
```

I urge you to either (a) copy and paste everything (except the page number) from "`function out =...`" all the way to "`end`" into a blank file. Name this file "f.m" and make sure it is in your current working directory or (b) in your MATLAB environment, select the drop down arrow under "New" and select "Function." Once you do that, a new file titled `untitledN.m` will appear. `N` will depend on the number of untitled files you're currently working on. If you have none, then your file will be named "`untitled.m`" and will look like this.

```
function [ output_args ] = untitled( input_args )
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here


end
```

From here, you can change your new file to look like the one I made. You only have to copy and paste one line and change the first line of your file a little. A MATLAB function must have the same filename as the function name. If you've changed `untitled` to `f`, then when you go to save the file, MATLAB will already suggest that its name should be f.m. If you changed the top line so that it reads "`function out = myWickedCoolFunction( x )`, then MATLAB will suggest that its name should be `myWickedCoolFunction.m`, and so on. You need to name the file so that the two names are the same. So if I did this

```
function [ OUT ] = continuousFactorial( IN )

% OUT = Gamma(IN+1) = (IN)!
%     (for input = IN = a + i*b with a > 0)

OUT = integral(@(t) t.^(IN).*exp(-t),0,Inf);

end
```

then I would need to make sure I save my file as `continuousFactorial.m`.

Let's move on as though you've saved your function as `f.m`. Making sure that the file `f.m` is in your Current Folder (one of the windows in the MATLAB environment, also called your current or working directory), then you can type `f(3)` into the command window. The returned value should be 6.000, if you're only showing 4 decimal places. You can type `format long` into the command window, hit return, then type `f(3)` again. You'll see that this is a *very* good numerical approximation to the true value, $3! = 6$. Now try this. Define a vector x by typing `x = 0:0.01:2;` Then try `f(x)` and you'll get all kinds of scary errors. This is what we mean by "$f(x_j)$ only accepts one value at a time."

There are many ways to complete the function `evaluateF`. The most straightforward way to do this is to use a `for` loop. We can use this method because MATLAB gives us a way to access the number of elements in the given variable x via the `size` or the `length` functions. Below, we'll see some examples as we continue our exploration of the Gamma function, $\Gamma(x)$, which we define right now for all complex numbers x with real part greater than zero.

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt \tag{1}$$

Then we have our crazy $f(z)$ given by $f(z) = \Gamma(z+1)$. Substituting, we get

$$f(z) = \int_0^\infty t^{z} e^{-t} dt \tag{2}$$

which, therefore, must be defined for all complex z with real part greater than -1. Now let's plot this function on the interval $[0, 2]$ at 201 equally spaced points. That is, I'm giving you a row vector, `x`, which I've defined as `x = 0:0.01:2`. I'm also giving you a function `f.m`. We'd like to simply do this `plot(x,f(x))` to look at our f, but that will give us scary errors (remember calling `f(x)` earlier?). That's why we want a vector `y = [f(x(1)), f(x(2)), ..., f(x(m))]`, so that we can write `plot(x,y)` and look at f. (It would have been nice if your boss had mentioned this, but she can be terse in the mornings.) We also want to plot many different intervals of different lengths. Maybe immediately after looking at f on `x = 0:0.01:2`, I realize that I want way better resolution and need to chop up $[0, 2]$ into a finer partition, `x = 0:0.0001:2`. Perhaps there's an interesting region around `x = 0.46` that I want to look at. It will be useful to make sure that `evaluateF` can handle whatever interval `x` that I give it. The script below is divided into sections by the lines which read `%% Plot on the interval...`. You'll see that each section contains a block which begins and ends with `%%% evaluateF(f, x) %%%`. The code sandwiched between these lines are examples of perfect answers, although some ways are better than others.

```
clear all;close all;clc

% Plotting f(x) = Gamma(x+1)

%% Plot on the interval 0 <= x <= 2

x = 0:0.01:2;  %splits [0, 2] into 201 points

%%%%%% evaluateF(f,x) %%%%%%
y = zeros(size(x));

for i = 1:size(y,2)

    y(i) = f(x(i));

end
%%%%%% evaluateF(f,x) %%%%%%

plot(x,y)

%% Plot on the interval 0.2 <= x <= 0.8
```

```
x = 0.2:0.0005:0.8;  %splits [.2, .8] into 1201 points

%%%%%% evaluateF(f,x) %%%%%%
y = zeros(1,length(x));

for i = 1:length(y)

    y(1,i) = f(x(1,i));

end
%%%%%% evaluateF(f,x) %%%%%%


plot(x,y)

%% Plot on the interval .45 <= x <= .5

x = 0.45:0.00005:0.5;   %splits interval into 1001 points

%%%%%% evaluateF(f,x) %%%%%%
s = 0;
y = [];   %NOT computationally efficient
while s < size(x,2)

    s = s + 1;

    y = [y f(x(s))];
end
%%%%%% evaluateF(f,x) %%%%%%


plot(x,y)

%% Plot on the interval .461 <= x <= .462

x = linspace(0.461,0.462,2000);   %splits interval into 2000 points

%%%%%% evaluateF(f,x) %%%%%%
y = zeros(size(x,1),size(x,2));

for i = 1:size(x,1)
    for j = 1:size(x,2)

        y(i,j) = f(x(i,j));

    end
end
%%%%%% evaluateF(f,x) %%%%%%


plot(x,y)
```

All of these methods return the vector `y = [f(x(1)), f(x(2)), ..., f(x(m))]`. Take a look at the second to last method. This way gives exactly the `y` that we want, but every time we go through the while loop, the size of `y` changes. Then on the first loop, MATLAB has to take a second and say, "okay hold on.

I need to change the size of `y` from a 0×0 to a 1×1." Then it goes along until the next time when it has to change `y` from a 1×1 to a 1×2. And so on until `y` is a $1 \times m$. Then MATLAB will have performed a bunch of *extra* steps it didn't need to. Since we know ahead of time that `y` will end up being a $1 \times m$, then we can knock out all of that unnecessary updating ahead of time. That's why we call `y = zeros(size(x))` or something similar at the beginning of each example above.

Copy and paste the script above (but not the page number) into a new, empty MATLAB file. The `%%`'s will section off the script so that you can run only one section at a time. Run each section. You'll notice right away that f has a local minimum around 0.46. Interesting, isn't it? Curious that there's a minimum between 0! and 1!. Then the derivative of f with respect to z should be zero somewhere near $z \approx 0.46$, right? Of course.