Week 5 - Evaluating Integrals, Iterative Solving

Reminder - the best place to learn MATLAB (or anything, really) is the internet! StackOverflow and MathWorks' own MATLAB Exchange are filled to the brim with people asking and answering questions about MATLAB. MATLAB's own documentation is also extensive and extremely helpful. It includes descriptions of how to call functions as well as usage examples.

Download this page as a PDF here (this document is generated automatically - print this page to PDF from your browser for the best result).

Return to Lectures

Functions of Functions

Like in many other languages, you can evaluate functions in terms of other functions. This enables you to build complex, multi-variate functions which are *much* easier to debug. Let's walk through an example using the ideal gas law.

Assume that we want to model the pressure of a system where it is appropriate to use the ideal gas law, and we know we are going to vary the temperature in the system in a sinusoidal (according to the sin function) manner. We could write one function where we use *sin* in the ideal gas law, but that is going to make it unrecognizable at a glance and add a level of complexity we don't need. `matlab temperature = @(time) 115 + 5 * sin(time);` This function is array-valued, though none of the operations we are doing require use of the **.** operator.

Now let's set up the ideal gas law. Here we are going to use molar volume rather than extensive volume so *n* is not included. `matlab % define the gas constant R = 8.314; % define our function pressure_IGL = @(volume, time) R * temperature(time) / volume; % descriptive name!` Notice that our variable names can be long and descriptive, and that we can use previously defined variables when setting up the function. When we call temperature, we pass through *time*, which we gave to *pressure_IGL*.

Try using this function to plot over some range of times, then make and label a plot appropriately.

Evaluating Integrals

MATLAB has the capability to do both symbolic (indefinite) and numeric (definite) integrals, using *int* and *integral*, respectively. If you do a good job keeping your numeric and symbolic functions separated, this difference will be a non-issue, so do so!

The intricacies of integration in MATLAB are best learned through practice, so I will preproduce some examples from the documentation here, but I definitely recommend solving practice problems to really learn.

Numeric

```matlab % evaluate the integral of e^(-x^2) * log(x)^2 (not something we would want to do by hand) fun = @(x) exp(-x.^2).*log(x).^2; % MATLAB allows for the evaluation of equations at infinity and negative infinity q = integral(fun,0,Inf)

    1.9475 ```

### Symbolic

As the documentation explains, symbolic integration can prove a lot more complicated for a myriad of reasons. It's rare that you will have to do it in chemical engineering, but knowing how to will help you out tremendously in those rare cases. ```matlab % let's find the indefinite integral of sin(*atheta + b) % declare needed symbols syms a b theta % define our function, remembering not to provide an @(input) f = sin(*atheta+b); % perform integration int(f)

    ans = -cos(b + a*theta)/a ```

## Iterative Solving

There are quite a few circumstances in chemical engineering where you may want to iteratively calculate two values until they converge, particularly in the case of stability calculations. While there are methods which will automatically perform iteration (see vpa from Week 4), they conceal a lot of the relevant math behind the systems and, while excellent tools once you have mastered the basics, can interfere with your understanding. This may seem medieval, but we recommend setting up systems which require iterative calculation using **for** or **while** loops, at least until you really understand how they work.

For an example, let's try and solve *sin(x)=1*. Pretend for now that inverse sin does not exist, because you won't normally have a closed-form, analytical solution.

### Pseudocode - studying the workflow

First, let's do some pseudocode so that we can understand the workflow. ```matlab % set up our symbols and function - for iterative calculation, we use numeric functions func = @(x) sin... % we need to decide how accurate we want to be, how many times we want to try at max, and a starting guess acceptable_error = 0.005 initial_guess = 2.0 for attempt = 1:max_number_attempts: % evaluate sin at our current guess temp = evaluate...

```
% check to see if we are acceptably close to the answer
if abs(1 - temp) < acceptable_error:
   % done

elseif % vale is too small:
```
```

```
   % make our guess smaller
else: % our value must be too big
   % make the guess bigger

```

Code - studying the implementation

And now let's formalize this into something a little more syntactically correct. `matlab func = @(x) sin(x); acceptable_error = 0.005 initial_guess = 2.0 for attempt = [1:max_number_attempts]: temp = func(guess); if abs(1 - temp) < acceptable_error: disp(guess) break elseif temp<1: guess = guess * 0.95; % reduce the size of guess by 5% of its current value else: guess = guess * 1.05;` If this never reaches the acceptable error, you can add more iterations or restart the process with a better first guess. Odds are this won't work the first time, which is alright! Every time you revisit the problem, you learn more.

Writing the guess-modifying *if* statements is the hardest part of iterative equation solving. It requires understanding the nature of the function you are trying to solve - speaking of which, can you think of a limitation of trying to solve a periodic function with iterative calculation?