**BME 313L: Introduction to Numerical Methods in Biomedical Engineering**

**Lab Report**

**Lab #11: Numerical Differentiation**

**Last name, First name: Liu, Vincent**

**EID: VL5649**

**Lab Section: 14035 (Tuesday 9:30-12:30)**

**Problem 1. From textbook problem 21.16**

Employ the MATLAB function *diff* to

(1) Plot the second derivative vs. for x=-2: 0.1: 2.

(2) Approximate where .

**Things to discuss**: (100 word minimum for each question, 50 word minimum for discussing what you learned, what was reinforced)

(1) What is the difference between diff and gradient?

(2) What is the relationship between the length of increment and truncation error.

**MATLAB code:**

x = -2.1:.1:2.1; %2 additional data points +1 on either end

y = 1/sqrt(2\*pi)\*exp(-x.^2/2); %function

d = diff(y)./diff(x); %first derivative

x = -2.05:.1:2.05; %inbetween x values

d2 = diff(d)./diff(x); %2nd derivative

x = -2:.1:2; %inbetween x values, desired range

plot(x,d2,'b') %plots x = -2: .1: 2 to corresponding d2y/d2x values

x\_infl(1) = fzero(@(T) interp1(x,d2,T,'linear','extrap'),-1); %finds zero

x\_infl(2) = fzero(@(T) interp1(x,d2,T,'linear','extrap'),1);

hold on %keeps plot

grid on %turns on grid

plot(x\_infl,0,'ro') %plots zero

xlabel('x values') %labels

ylabel('d^2y/d^2x')

title('2nd Derivative Values of a Function')

fprintf('The x values where the 2nd derivative is 0 are:\n') %display values

fprintf('%f and %f\n',x\_infl(1),x\_infl(2))

**MATLAB function:**

The purpose of this function was to estimate the derivative of a function, numerically. To do so, we can compute the differences between evenly spaced intervals of values corresponding to the function (twice) to estimate numerical values for the second derivative.

x = -2.1:.1:2.1; %2 additional data points +1 on either end

This first line of code creates an array of x values that we will be plugging into our function. Since 1 point is lost every time we take the difference between points, we add 2 points (1 on either end) so that we end up with the desired range of values.

y = 1/sqrt(2\*pi)\*exp(-x.^2/2); %function’

This line of code corresponds to the function given to us and outputs y values that correspond to the x values in our array.

d = diff(y)./diff(x); %first derivative

This line of code estimates the first derivative by taking the differences between our y values and dividing it by the difference between the corresponding x values.

x = -2.05:.1:2.05; %inbetween x values

This line of code adjusts the corresponding x values to line up with those of our derivative values. These x values are ‘in between’ those of our previous values because we took the differences.

d2 = diff(d)./diff(x); %2nd derivative

This line of code computes the 2nd derivative values in the same way that we calculated the first derivative.

x = -2:.1:2; %inbetween x values, desired range

This line of code adjusts the x values again, so that they line up with our second derivatives.

plot(x,d2,'b') %plots x = -2: .1: 2 to corresponding d2y/d2x values

This line of code plots our 2nd derivative values to their corresponding x’s.

x\_infl(1) = fzero(@(T) interp1(x,d2,T,'linear','extrap'),-1); %finds zero

x\_infl(2) = fzero(@(T) interp1(x,d2,T,'linear','extrap'),1);

These 2 lines of code compute the roots using MATLAB’s fzero function. After looking at the graph, we know that there are 2 roots within the range that we computed so we can take our initial guesses to be close to these points (x = -1 and x = 1).

hold on %keeps plot

This line of code keeps MATLAB from generating a separate plot when we want to plot our calculated root values.

grid on %turns on grid

This line of code turns on the grid for our plot so that we can see where points line up.

plot(x\_infl,0,'ro') %plots zero

This line of code plots our calculated roots onto our existing plot.

xlabel('x values') %labels

ylabel('d^2y/d^2x')

title('2nd Derivative Values of a Function')

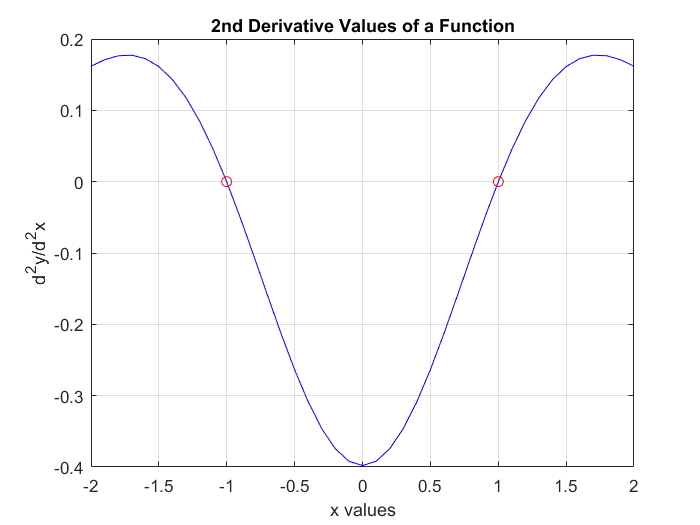
These 3 lines of code add labels to our graph so that it can be more easily understood.

fprintf('The x values where the 2nd derivative is 0 are:\n') %display values

fprintf('%f and %f\n',x\_infl(1),x\_infl(2))

These last 2 lines of code print out the results for our calculated roots into the command window.

**Results:**



****

**Discussion:**

As shown by the results, we can approximate the 2nd derivative of a function numerically by computing the change in the function’s values. This works because the MATLAB function diff takes the difference between values—we can take advantage of this and take the difference of y and divide that by the change in x, the definition of the slope or derivative at a point. By applying this to numerous small points along our function (and doing it multiple times), we can effectively approximate the derivative (or second derivative). The diff function differs from the gradient function in that the diff function only takes the difference between 2 points, the gradient function on the other hand takes the difference between a value above and below it and divides it by 2, akin to using the centered finite difference approximation. The diff function is useful in that it can be used for any set of values regardless of how they’re spaced while the gradient function makes the assumption that the spacing between points is 1. As the length of increment is decreased the truncation error of the approximation is also decreased. This is because we are attempting to approximate an infinitely small number of steps in between with a finite number and by increasing the number of steps (decreasing the length) we make better approximations.

From this problem, we learned how to use the diff function in order to calculate all of the differences between values in an array. We reviewed how to do basic array/matrix operations, element-wise. Additionally, we reviewed how to use the fzero function in order to calculate the zeroes of a function. Lastly, we reviewed how to format and plot arrays and points using MATLAB’s plot function.

**Problem 2. From textbook problem 21.19**

Calculate derivative of at x=2 with four different formulas:

1. Improved forward finite difference approximation
2. Improved backward finite difference approximation
3. Centered finite difference approximation
4. Improved centered finite difference approximation

Change the increment ‘*h*’ from 0.5 to 0.01 with -0.01 intervals (dx=0.5:-0.01:0.01). Generate 4 plots in one graph *(improved forward, improved backward, improved centered, centered finite difference approximation* vs *dx*).

**Things to discuss**: (100 word minimum for each question, 50 word minimum for discussing what you learned, what was reinforced)

(1) How could we improve these finite difference approximations?

(2) Which finite difference approximation could give you the most accurate estimate?

**MATLAB code:**

f =@(x) exp(-2\*x)-x; %function

h = .5:-.01:.01; %step size

n = 2; %point for derivative

%improved forward finite difference approximation

a = (-f(n+2\*h)+4\*f(n+h)-3\*f(n))./(2.\*h);

%improved backward finite difference approximation

b = (3\*f(n)-4\*f(n-h)+f(n-2\*h))./(2.\*h);

%centered finite difference approximation

c = (f(n+h)-f(n-h))./(2.\*h);

%improved centered finite difference approximation

d = (-f(n+2\*h)+8\*f(n+h)-8\*f(n-h)+f(n-2\*h))./(12.\*h);

%plot

plot (h,a,h,b,h,c,h,d) %plot

legend('Improved Forward Finite Difference Approximation','Improved Backward Finite Difference Approximation','Centered Finite Difference Approximation','Improved Centered Finite Difference Approximation') %formatting

xlabel('dx')

ylabel('Derivative Approximation')

title('Plot of Various Finite Difference Approximations of a Function')

**MATLAB function:**

The purpose of this function was to compare 4 different finite difference approximations at converging to the true value of the actual derivative. To do this, we decreased the step sizes of various finite difference approximations of a function and then plotted the values against each other; based on how quickly the functions reached a stable value would determine how effective the approximation is at estimating the derivative.

f =@(x) exp(-2\*x)-x; %function

This first line of code is the function given to us as part of the problem that we are trying to approximate. It is written as an anonymous function so that we can easily write it into our finite difference approximations with different values.

h = .5:-.01:.01; %step size

This line of code creates an array of values that corresponds to the various step sizes that we will be testing in our approximations. This value is continually decreased as though we were taking the limit of a function as it approached 0.

n = 2; %point for derivative

This line of code outlines the value where we are trying to approximate our derivative.

%improved forward finite difference approximation

a = (-f(n+2\*h)+4\*f(n+h)-3\*f(n))./(2.\*h);

%improved backward finite difference approximation

b = (3\*f(n)-4\*f(n-h)+f(n-2\*h))./(2.\*h);

%centered finite difference approximation

c = (f(n+h)-f(n-h))./(2.\*h);

%improved centered finite difference approximation

d = (-f(n+2\*h)+8\*f(n+h)-8\*f(n-h)+f(n-2\*h))./(12.\*h);

These 4 lines of code compute, using the improved forward finite difference approximation, the improved backward finite difference approximation, the centered finite difference approximation, and the improved centered finite difference approximation, respectively, the derivative for various step sizes.

%plot

plot (h,a,h,b,h,c,h,d) %plot

This line of code plots all of our approximations against the step sizes used to compute them.

legend('Improved Forward Finite Difference Approximation','Improved Backward Finite Difference Approximation','Centered Finite Difference Approximation','Improved Centered Finite Difference Approximation') %formatting

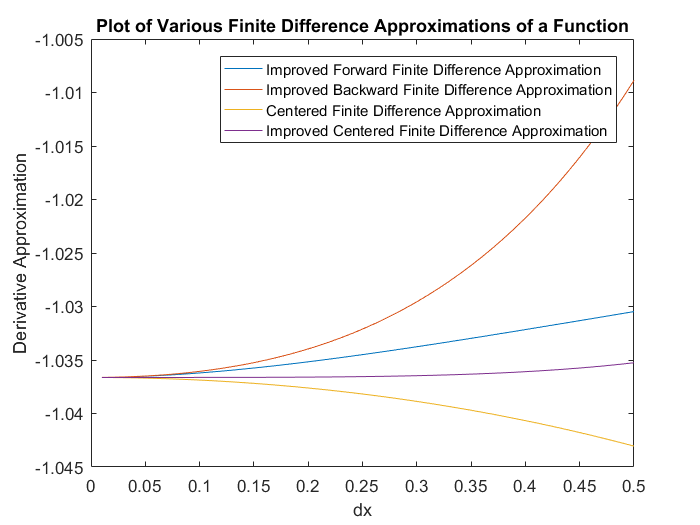
xlabel('dx')

ylabel('Derivative Approximation')

title('Plot of Various Finite Difference Approximations of a Function')

These last 4 lines of code add labels to our plot so that it is more easily understood.

**Results:**



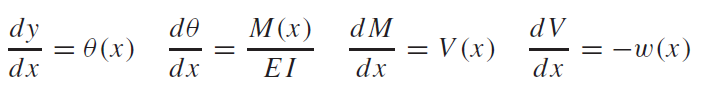
**Discussion:**

As shown by the results, for all 4 different types of finite difference approximations, as the step size, dx, was decreased, all of the values converged on a single point, the true value of the derivative. Of the 4 different types of approximations plotted, 3 of them use improved approximations based on adding additional terms from the Taylor series expansion. When comparing the improved and nonimproved versions (centered in this case), we can see that the improved case yields a much closer approximation at a larger step value. If we were to continue adding terms from the Taylor series expansion we could generate even more accurate approximations of the derivative at larger step values; however, this would be done at the expense of computation speed. How accurate the approximation though, is also in part due to the type of approximation—it can be seen that the centered finite difference approximation is slightly better than the improved backward finite difference approximation, despite it not being the improved version. From the graph, it can be seen that the improved centered finite difference approximation is the best finite difference approximation out of the 4 different approximations compared. The graph of this approximation is nearly straight, starting out at a value already very close to the derivative at a large step size. This is because of the nature of the points that the centered difference approximation uses in addition to being an improved version of the original approximation.

From this problem, we reviewed how to use anonymous functions in order to effectively iterate multiple calculations using the same function. We also reviewed how to use finite difference approximations of different step sizes using basic array operations. Lastly, we reviewed how to format and plot arrays into a plot in order to effectively communicate our results.

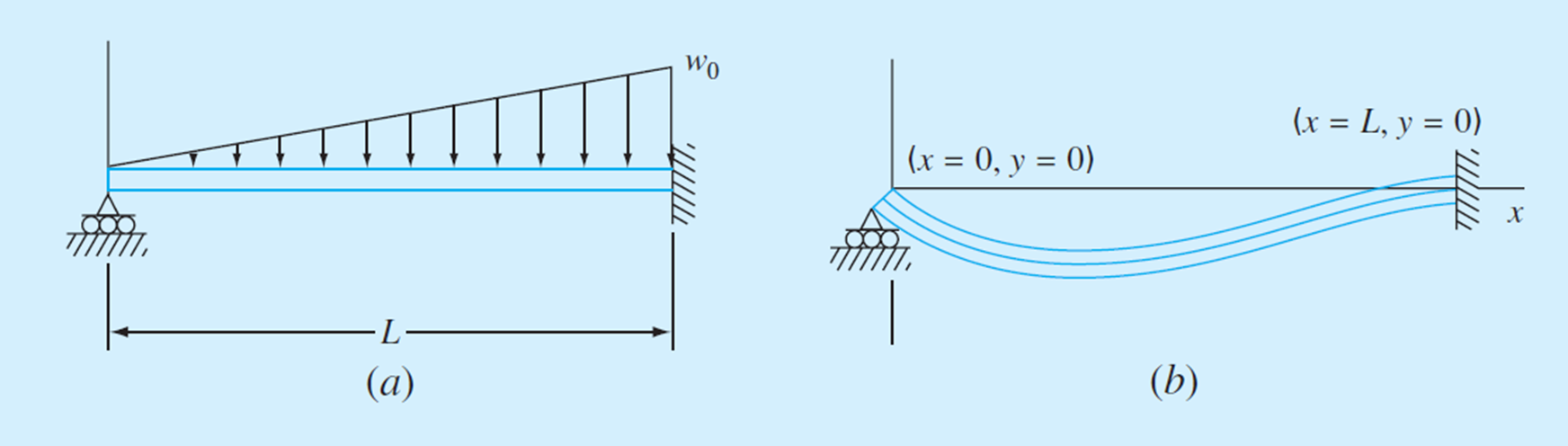
**Problem 3 From textbook problem 21.37**

The following relationships can be used to analyze uniform beams subject to distributed loads:

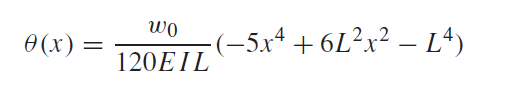


where *x* = distance along beam (m), *y* = deflection (m), (*x*) = slope (m/m), *E* = modulus of elasticity (Pa = N/m2), *I* = moment of inertia (m4), *M*(*x*) = moment (N m), *V*(x) = shear (N), and *w*(*x*) = distributed load (N/m).

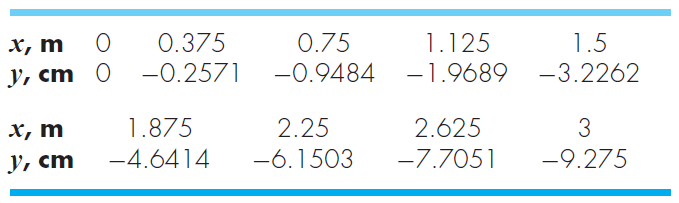
For the case of a linearly increasing load (Fig. 1).



The slope can be computed analytically as



You measure the following deflections along the length of a simply supported uniform beam



Employ numerical differentiation to compute the slope, the moment (in N m), the shear (in N) and the distributed load (in N/m). Use the following parameter values in your computation: E = 200 GPa, and I = 0.0003 m4.

**Things to discuss**: (100 word minimum for each question, 50 word minimum for discussing what you learned, what was reinforced)

(1) What kind of finite difference approximation would you use for the data points on boundaries? Why?

**MATLAB code:**

x = [0;.375;.75;1.125;1.5;1.875;2.25;2.625;3]; %data

y = [0;-.2571;-.9484;-1.9689;-3.2262;-4.6414;-6.1503;-7.7051;-9.275]./100;

E = 200e9;

I = .0003;

n = length(x); %number of terms

%t = thetas

for i = 2:n-1 %only works with points on both sides

t(i) = (y(i+1)-y(i-1))/(2\*.375); %centered finite difference

end

t(1) = (-y(1+2)+4\*y(1+1)-3\*y(1))/(2\*.375);

t(n) = (3\*(y(n))-4\*(y(n-1))+y(n-2))/(2\*.375);

%M = moments

for i = 2:n-1 %only works with points on both sides

M(i) = (t(i+1)-t(i-1))/(2\*.375); %centered finite difference

end

M(1) = (-t(1+2)+4\*t(1+1)-3\*t(1))/(2\*.375);

M(n) = (3\*(t(n))-4\*(t(n-1))+t(n-2))/(2\*.375);

M = M\*E\*I; %dt/dx = M/(EI)

%v = shear

for i = 2:n-1 %only works with points on both sides

V(i) = (M(i+1)-M(i-1))/(2\*.375); %centered finite difference

end

V(1) = (-M(1+2)+4\*M(1+1)-3\*M(1))/(2\*.375);

V(n) = (3\*(M(n))-4\*(M(n-1))+M(n-2))/(2\*.375);

%w = distributed load

for i = 2:n-1 %only works with points on both sides

w(i) = (V(i+1)-V(i-1))/(2\*.375); %centered finite difference

end

w(1) = (-V(1+2)+4\*V(1+1)-3\*V(1))/(2\*.375);

w(n) = (3\*(V(n))-4\*(V(n-1))+V(n-2))/(2\*.375);

w=-w; %dV/dx = -w

A = [x';y';t;M;V;w]; %compiles arrays

fprintf(' x\t\t\ty\t\t Theta(x)\t\t M(x)\t\t\t V(x)\t\t\t w(x)\n') %outputs table

fprintf('%1.3f\t%+f\t%f\t%f\t %f\t %f\n',A)

**MATLAB function:**

The purpose of this function was to approximate various loads within a beam using differentiation. To do so, we could numerically differentiate the x and y values given to us part of the problem multiple times in order to compute shear, moment, etc.

x = [0;.375;.75;1.125;1.5;1.875;2.25;2.625;3]; %data

y = [0;-.2571;-.9484;-1.9689;-3.2262;-4.6414;-6.1503;-7.7051;-9.275]./100;

E = 200e9;

I = .0003;

These first 4 lines of code correspond to data given to us as part of the problem. The first 2 of the lines are the x and corresponding y values at different points along the beam. The last 2 of the lines correspond to the modulus of elasticity and moment of inertia, properties of the beam that were provided as part of the problem.

n = length(x); %number of terms

This line of code computes the number of terms that we have to work with.

%t = thetas

for i = 2:n-1 %only works with points on both sides

t(i) = (y(i+1)-y(i-1))/(2\*.375); %centered finite difference

end

These 3 lines of code form a for loop that works with all of the values in our array except for the 2 end points. For all of these points in the middle, we can compute the centered finite difference, but we can’t apply the same formula to the end points because there aren’t points on both sides of them.

t(1) = (-y(1+2)+4\*y(1+1)-3\*y(1))/(2\*.375);

t(n) = (3\*(y(n))-4\*(y(n-1))+y(n-2))/(2\*.375);

These 2 lines of code apply the improved forward finite difference approximation on the first point and the improved backward finite difference approximation on the last point. We use the improved functions for both of these end points to better approximate the true values.

%M = moments

for i = 2:n-1 %only works with points on both sides

M(i) = (t(i+1)-t(i-1))/(2\*.375); %centered finite difference

end

M(1) = (-t(1+2)+4\*t(1+1)-3\*t(1))/(2\*.375);

M(n) = (3\*(t(n))-4\*(t(n-1))+t(n-2))/(2\*.375);

These 5 lines perform the same operations as before, only instead of calculating theta, the slope, from x, we are calculating the moment, M, from theta.

M = M\*E\*I; %dt/dx = M/(EI)

This line of code finishes the moment calculation. Because we are given that dt/dx = M/(EI), we need to multiply both sides by EI in order to get M on its own.

%v = shear

for i = 2:n-1 %only works with points on both sides

V(i) = (M(i+1)-M(i-1))/(2\*.375); %centered finite difference

end

V(1) = (-M(1+2)+4\*M(1+1)-3\*M(1))/(2\*.375);

V(n) = (3\*(M(n))-4\*(M(n-1))+M(n-2))/(2\*.375);

These 5 lines perform the same operations as before, only instead of calculating the moment, M, from theta, we are calculating shear, v, from the moment.

%w = distributed load

for i = 2:n-1 %only works with points on both sides

w(i) = (V(i+1)-V(i-1))/(2\*.375); %centered finite difference

end

w(1) = (-V(1+2)+4\*V(1+1)-3\*V(1))/(2\*.375);

w(n) = (3\*(V(n))-4\*(V(n-1))+V(n-2))/(2\*.375);

These 5 lines perform the same operations as before, only instead of calculating shear, v, from the moment, we are calculating the distributed load, w, from the shear.

w=-w; %dV/dx = -w

This line of code calculates w from the derivative values that we calculated before; this is because dV/dx = -w so w = -dV/dx.

A = [x';y';t;M;V;w]; %compiles arrays

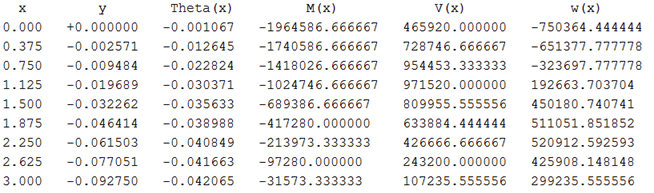
This line of code compiles all of the arrays of values that we calculated and compiles them into a single matrix so that they can be outputted using fprintf.

fprintf(' x\t\t\ty\t\t Theta(x)\t\t M(x)\t\t\t V(x)\t\t\t w(x)\n') %outputs table

fprintf('%1.3f\t%+f\t%f\t%f\t %f\t %f\n',A)

These last 2 lines of code format and output our table into the command window.

**Results:**

****

**Discussion:**

As shown by the results, we can calculate the slope, moment, shear, and distributed load values at points along a simply supported beam, given the deflection of the beam. By using forward, centered, and backward finite differences, we can numerically approximate aforementioned values. For the most part, we chose to use centered finite difference approximations because they of the way the values are spaced out, they tend to offer the most accurate approximation; however, this does not work at the 2 end values because it requires a point on either side to compute. To remedy this, we chose to apply an improved forward finite difference approximation on the first value and an improved backward finite difference approximation on the last value; the improved version of these methods was chosen to better approximate the error.

From this problem, we reviewed how to use for loops to only work with certain elements of an array. We also reviewed basic array operations, used to calculate finite difference approximations using the formulas. Lastly, we also reviewed how to format and print out tables into the command window using MATLAB’s built in fprintf function.