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

**Lab Report**

**Lab\_2: Chapter 1. Mathematical Modeling, Numerical Methods, and Problem Solving**

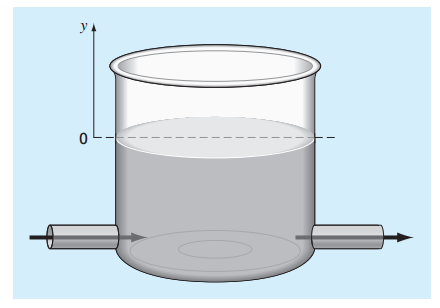
**Chapter 4. Round off and Truncation Errors**

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

**EID: VL5649**

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

**Problem 1**

A storage tank contains a liquid at depth ‘y’ where y=0 when the tank is half full. Liquid is withdrawn at a flow rate dependent on the depth as . The contents are resupplied at a sinusoidal rate = . The conservation equation can be written as:

Use Euler method to solve for the depth y from t = 0 to 10 days with a step size of 0.5 day. The parameter values are A = 1000 m2, Q = 400 m3/day, and α = 50. Assume that the initial condition is y = 0.

1. Generate a table in MATLAB using fprintf with columns as time (from t = 0 to 10 days with step size of 0.5 day) and depth y (at time t = 0 to 10 day)
2. Plot depth y, depth rate vs. time ***t*** (show plots of y vs. time and depth rate vs. time in one graph)

**Possible Things to discuss: (you can discuss beyond this scope)**

1. The plot for the volume vs. time is expected to be an oscillating curve, why?
2. Will the tank be filled or drained in a long term?

(The following is your answer)

**MATLAB Code:**

t = 0:.5:10; %days vector

A = 1000; %given parameters

Q = 400;

a = 50;

y = zeros(1,length(t)); %creates vector of matching length

yi = 0; %initial y

y(1) = yi; %sets initial y

for i = 2:length(t) %for loop to use previous values

dy = (((3\*(Q/A)\* (cos(t(i))).^2) - (a/A)\*(1 + y(i)).^1.5)); %the change in y for any given time

yf = yi + (.5 \* dy); %euler method for y

y(i) = yf;

yi = yf;

end

dy = 3\*Q/A\*(cos(t)).^2-a/A\*(1+y).^1.5; %same as before but independent of i

A = [t;y]; %Creates matrix

fprintf('Time(days)\tHeight of water\n');%title for table

fprintf('%.1f %f\n',A);%parameters for table

plot(t,y,t,dy,'--')%creates plot

legend('y vs time','dy/dt vs time')%labels plot

xlabel('Time (days)')

ylabel('Height (units)')

title('Height of water and Change in height with respect to Time')

**MATLAB Function:**

The purpose of this question was to model the water level in a tank given the equation for the change of the water level over time. To approach this, we first generate a vector for the days in the model. Using this vector, we can then calculate the height of the water for each given day with the equation given. These values can then be plotted against the original days to generate a graph that models the water level in the tank.

t = 0:.5:10; %days vector

This first line of code creates a vector for ‘days’ that lapse over the course of the model.

A = 1000; %given parameters

Q = 400;

a = 50;

These 3 lines of code specify the parameters to be used to calculate the change in height. These values, given as 1000, 400, and 50, are stored as the variables A, Q, and a, respectively.

y = zeros(1,length(t)); %creates vector of matching length

This line of code creates a vector of 0’s from 1 to the length of the vector t. the purpose of this line is so that y is the same length as t.

yi = 0; %initial y

y(1) = yi; %sets initial y

These 2 lines set the first value in the y vector as 0

for i = 2:length(t) %for loop to use previous values

This line of code specifies that it works from the 2nd value of the vector until the end of the vector (which is specified as the length of t.

dy = (((3\*(Q/A)\* (cos(t(i))).^2) - (a/A)\*(1 + y(i)).^1.5)); %the change in y for any given time

This line of code, the equation for dy, is given to us as part of the question. It calculates the change in height of the water by giving us the in flow and outflow of the water, accounting for the area of the tank.

yf = yi + (.5 \* dy); %euler method for y

y(i) = yf;

yi = yf;

These 3 lines of code use the Euler method and the change in y (dy) to calculate y at a given time. It calculates using the previous value, yi, and then stores its result as yi, to be used for the next calculation. It does this over the course over the for loop to generate the values of the vector, y (excluding the very first y defined earlier).

end

This line of code is self-explanatory— it closes the for loop.

dy = 3\*Q/A\*(cos(t)).^2-a/A\*(1+y).^1.5; %same as before but independent of i

Because of the way dy was written before, dy had to be redefined by this line in order to generate the vector result that we wanted.

A = [t;y]; %Creates matrix

This line of code creates a matrix using the vectors, t and y, and stores it to be used by fprintf later on.

fprintf('Time(days)\tHeight of water\n');%title for table

fprintf('%.1f %f\n',A);%parameters for table

These 2 lines of code format the table to be outputted by fprintf. The first line creates titles for each of the columns and the second line specifies that matrix ‘A’ should be used with the floating point specified.

plot(t,y,t,dy,'--')%creates plot

This line plots both y and dy against the time, t. The second plot, t vs dy is specified to be a dashed line by ‘--‘ in the code.

legend('y vs time','dy/dt vs time')%labels plot

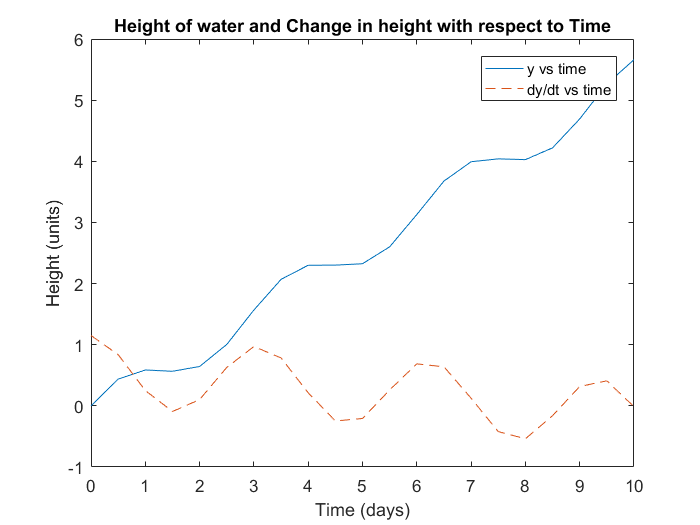
title('Height of water and Change in height with respect to Time')

xlabel('Time (days)')

ylabel('Height (units)')

These last 4 lines of code label the x and y axis, label the plots with a legend differentiating between the 2 graphs, and add a title to the graph.

**Results:**



**Discussion:**

As shown by the results, the height of the water in the tank increases at an oscillating rate, with respect to time. The height of the water starts at 0 relative to a point, y, as given by the problem, and gradually increases while oscillating. The same oscillation in y can be seen in the graph of dy/dt as the values of y are dependent on dy/dt. These graphs oscillate because they are dependent on cos(t)^2 which is an oscillating graph. As shown by the results, the tank will gradually fill up with water over time with no upper bound, because the value of dt/dy is never less than 0 for very long. The important aspect to be taken away from this problem was the ability to model using MATLAB, an equation given its derivative and to plot that equation to model a real life scenario.

From this we learned how to be able to solve for an equation given its derivative, using Euler’s method, in MATLAB. We also learned how to utilize a for loop in order to solve for values that build on previous ones (e.g. Euler’s method). Furthermore we learned how to create and format tables in MATLAB using the fprintf function. Lastly, we refreshed our knowledge on how to create and label various details of plots.

**Problem 2**

If |x|<1, the Maclaurin series expansion for is:



Starting with the simplest version, , add terms one at a time to estimate the value of . After each term is added, compute the true and approximate percent relative errors. Add terms until the absolute value of the approximate error estimate falls below an error criterion e = 0.05. Write a Matlab function named ‘my\_maclaurin.m’ that takes as inputs the variables ‘x’ and the error criterion ‘e’. The outputs from the function should be the following two things:

1. A table shown on command window with the format same as in EXAMPLE 4.1 on page 93 (columns should be #terms, result, true percent relative error and approximate percent relative error).
2. The number of terms (including 1) required in the expression until the absolute error value falls below the error criterion ‘e’

In the problem script file, run the function for x=0.7 and e=0.05 and report the results.

**Possible Things to discuss: (you can discuss beyond this scope)**

1. Does the error decrease linearly or exponentially when we add more term in the Maclaurin series expansion? Why?

(The following is your answer)

**MATLAB Code:**

**Function:**

function output = my\_maclaurin\_VL(x,e)

N = 1 / (1 - x); %equation being compared to

n = 0; %initializes counter

M = 1; %initializes series

Result(1) = M; %sets first value of result

error = abs((N - M)/N); %calculates initial error

TrueError(1) = error; %puts error in vector

ApproxError(1) = 1; %starts approxerror vector

while error > e %loop to repeat series

n = n + 1; %updates counter

M = M + x^n; %adds next value to series

Result(end+1) = M; %stores result in vector

error = abs((N - M)/N); %calculates error

TrueError(end+1) = error; %stores error in vector

ApproxError(end+1) = (Result(end)-Result(end-1))/Result(end); %calculates approximate error

end %closes loop

Terms = (1:(n+1))'; %makes all the vectors vertical to be used in a table

TrueError = TrueError';

Result = Result';

ApproxError = ApproxError';

table(Terms,Result,TrueError,ApproxError) %creates table

disp('Number of terms used: ') %displays how many terms were used

disp(n+1)

end %closes function

**Main script:**

x = .7; %variables

e = 0.05;

my\_maclaurin\_VL(x,e) %calls function

**MATLAB Function:**

The purpose of this question was to iterate terms in a given Maclaurin series until the % error was below a certain threshold. These successive terms would then be outputted in a table displaying each iteration.

function output = my\_maclaurin\_VL(x,e)

This 1st line of code sets the variables that the function will take, x and e, as well as sets the name of the function.

N = 1 / (1 - x); %equation being compared to

This line of code is just the equation that we have a Maclaurin series expansion for. The result, stored as the variable N, will be the exact value that we compare our series to.

n = 0; %initializes counter

This line of code initializes the counter that I use to keep track of the number of iterations

M = 1; %initializes series

This line of code is the first value of the Maclaurin series that the successive iterations will add on to.

Result(1) = M; %sets first value of result

This line of code stores the value of the Maclaurin series (in this case, 1) as the first value in the vector named Result.

error = abs((N - M)/N); %calculates initial error

TrueError(1) = error; %puts error in vector

These 2 lines take the absolute value of the calculated error and sore it as a vector. In this case, this line was only meant to initialize the first value of the error vector and as such, only sets the first value.

ApproxError(1) = 1; %starts approxerror vector

This line of code was just meant to be more of a placeholder than anything, as approximate error is calculated using previous calculations, but there are no previous calculations for the first calculation. The percent error would also be 100 if you take any value before the first value to be 0.

while error > e %loop to repeat series

This line of code creates a while loop to repeat calculations until the condition error > e, where error is the calculated true error and e is a value defined in the main script.

n = n + 1; %updates counter

This line adds 1 to the counter each time it is looped. This allows it to keep track of the number of iterations, as well as to provide the correct value for which to add on to the series

M = M + x^n; %adds next value to series

Result(end+1) = M; %stores result in vector

These 2 lines update the Maclaurin series and update the Result vector accordingly.

error = abs((N - M)/N); %calculates error

TrueError(end+1) = error; %stores error in vector

These 2 lines update the error and update the error vector accordingly.

ApproxError(end+1) = (Result(end)-Result(end-1))/Result(end); %calculates approximate error

This line calculates the approximate error and stores it as a vector.

end %closes loop

This line terminates the while loop.

Terms = (1:(n+1))'; %makes all the vectors vertical to be used in a table

This line creates the Terms vector, using the number of iterations that was kept track of with my counter. The value needs a 1 added to it because we need to account for the very first value, 1, that was not a part of the loop.

TrueError = TrueError';

Result = Result';

ApproxError = ApproxError';

These 3 lines of code rotate the TrueError, Result, and ApproxError vectors so that they can be used as part of MATLAB’s table function

table(Terms,Result,TrueError,ApproxError) %creates table

This line creates a table with columns containing the terms, result, trueerror, and approxerror vectors

disp('Number of terms used: ') %displays how many terms were used

disp(n+1)

These 2 lines of code output the number of iterations (or terms) needed in order to fall below to specified error limit, in an easily understandable fashion.

end %closes function

This line closes the function.

x = .7; %variables

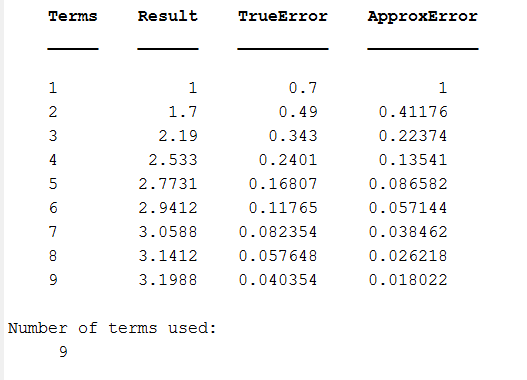
e = 0.05;

These 2 lines of code outline the variables given in the problem. X is any value that is used in the formula to compare and e is the error criterion that we have to get below.

my\_maclaurin\_VL(x,e) %calls function

This last line of code calls the ‘my\_maclaurin\_VL(x,e)’ function outlined earlier.

**Results:**



**Discussion:**

As shown by the results, it takes 9 terms of the Maclaurin series in order to fall below the specified .05 error. The values for the true error and approximate error can be seen as large and exponentially decreasing as successive terms are added because with each term in the series, the values are approaching 0 and the sum gets closer to the function. With more iterations, the value could be even closer and error even smaller.

From this problem, we learned how to use a while loop until a certain condition was met (in this case, the error falling below a threshold). We also learned how to continually add values to a vector so that the length is variable. Furthermore, we learned how to print strings in order to increase user understandability. We also refreshed our ability to set and call functions from other scripts. Lastly, we refreshed our knowledge on how to create tables using MATLAB’s built in table function.

**Problem 3.**

Consider the function  on the interval [-2,2] with h=0.25. Use the forward, backward and centered finite difference approximations for the first derivative so as to graphically illustrate which approximation is most accurate. Report the following:

1. Four plots have to be in one graph and create different lines for each plot. Set line width = 2 for plots.
   1. Theoretical difference: Solid black
   2. Backward difference: Dotted blue line
   3. Centered difference: Dash dotted green line
   4. Forward difference: Dashed red line
2. Generate a table containing following entries using *fprintf*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x | f(x) | f(x-1) | f(x+1) | f '(x)theory | f '(x)back | εback (%) | f '(x)cent | εcent (%) | f '(x)forw | εforw (%) |
|  |  |  |  |  |  |  |  |  |  |  |

**Possible Things to discuss: (you can discuss beyond this scope)**

1. **Which finite difference approximation is most accurate? Why?**

(The following is your answer)

**MATLAB Code:**

x = (-2:.25:2); %makes vector

h = .25;

f = 3 \* (x) .^3 - 2 \* (x) + 6; %original function

fprev = 3 \* (x-1) .^3 - 2 \* (x-1) + 6; %function with values x-1

fnext = 3 \* (x+1) .^3 - 2 \* (x+1) + 6; %function with values x+1

theoretical = 9 \* x .^2 - 2; %theoretical change in x (derivative)

forward = ((3 \* ((x+h) .^ 3) - 2 \* (x+h) + 6) - (3 \* x .^ 3 - 2 \* x + 6))/h; %forward difference

backward = ((3 \* x .^ 3 - 2 \* x + 6) - (3 \* (x-h) .^ 3 - 2 \* (x-h) + 6))/h; %backward difference

centered = ((3 \* (x+.5\*h) .^ 3 - 2 \* (x+.5\*h) + 6) - (3 \* (x-.5\*h) .^ 3 - 2 \*(x-.5\*h) + 6))/h; %centered difference

eback = (theoretical - backward)./theoretical; %calculated errors

ecent = (theoretical - centered)./theoretical;

eforw = (theoretical - forward)./theoretical;

A =[x;f;fprev;fnext;theoretical;backward;eback;centered;ecent;forward;eforw]; %creates matrix of all values

fprintf('x\t f(x)\t f(x-1)\t f(x+1)\t f''(x)theory\t f''(x)back Eback(%%) f''(x)cent Ecent(%%) f''(x)forw Eforw(%%)\n'); %outputs table

fprintf('%+.2f %+.3f\t %+f %+f\t%+f\t %+f\t %+f\t %+f\t %+f\t %+f\t %+f\n',A);

plot(x,theoretical,'b',x,backward,x,centered,x,forward); %plots

xlabel('x') %axis labels

ylabel('y')

legend('theoretical','backward approximation','centered approximation','forward approximation');

**MATLAB Function:**

The purpose of this

x = (-2:.25:2); %makes vector

This first line of code creates a vector of ‘x’ values from -2 to 2 in increments of .25 (we already know this because the step, h, was defined as .25)

h = .25;

This line of code just defines the step, h, as .25. Despite being used as part of the x vector, it is useful to have and will be used in later calculations.

f = 3 \* (x) .^3 - 2 \* (x) + 6; %original function

This line of code is merely the given function. Because x is a vector, the output f, is also a vector, and a .^ must be used in order to avoid any errors.

fprev = 3 \* (x-1) .^3 - 2 \* (x-1) + 6; %function with values x-1

fnext = 3 \* (x+1) .^3 - 2 \* (x+1) + 6; %function with values x+1

These 2 lines calculate f(x+1) and f(x-1) to be used as part of the table later on.

theoretical = 9 \* x .^2 - 2; %theoretical change in x (derivative)

This line of code is the derivative of the original function. It calculates the theoretical change in x at any given t. The output of this vector is also a vector and a .^ must also be used.

forward = ((3 \* ((x+h) .^ 3) - 2 \* (x+h) + 6) - (3 \* x .^ 3 - 2 \* x + 6))/h; %forward difference

backward = ((3 \* x .^ 3 - 2 \* x + 6) - (3 \* (x-h) .^ 3 - 2 \* (x-h) + 6))/h; %backward difference

centered = ((3 \* (x+.5\*h) .^ 3 - 2 \* (x+.5\*h) + 6) - (3 \* (x-.5\*h) .^ 3 - 2 \*(x-.5\*h) + 6))/h; %centered difference

These 3 lines of code create vectors of the forward, backward, and centered finite difference approximations, respectively. By subtracting the difference between 2 values in the function and dividing by the step, the change between those 2 points can be calculated.

eback = (theoretical - backward)./theoretical; %calculated errors

ecent = (theoretical - centered)./theoretical;

eforw = (theoretical - forward)./theoretical;

These 3 lines of code calculate the % error between the finite difference approximations and the theoretical change (given by the derivative). These values are all stored in a vector to be used later in a table.

A = [x;f;fprev;fnext;theoretical;backward;eback;centered;ecent;forward;eforw]; %creates matrix of all values

This line creates a matrix using all of the above listed vectors. The end result is a giant 11x17 matrix of all the calculated values.

fprintf('x\t f(x)\t f(x-1)\t f(x+1)\t f''(x)theory\t f''(x)back Eback(%%) f''(x)cent Ecent(%%) f''(x)forw Eforw(%%)\n'); %outputs table

fprintf('%+.2f %+.3f\t %+f %+f\t%+f\t %+f\t %+f\t %+f\t %+f\t %+f\t %+f\n',A);

These 2 lines create the table using the fprintf function. The first line titles each of the columns in the table and the second line sets the matrix A (as well as outlines the parameters of each of the values) to be used.

plot(x,theoretical,'b',x,backward,x,centered,x,forward); %plots

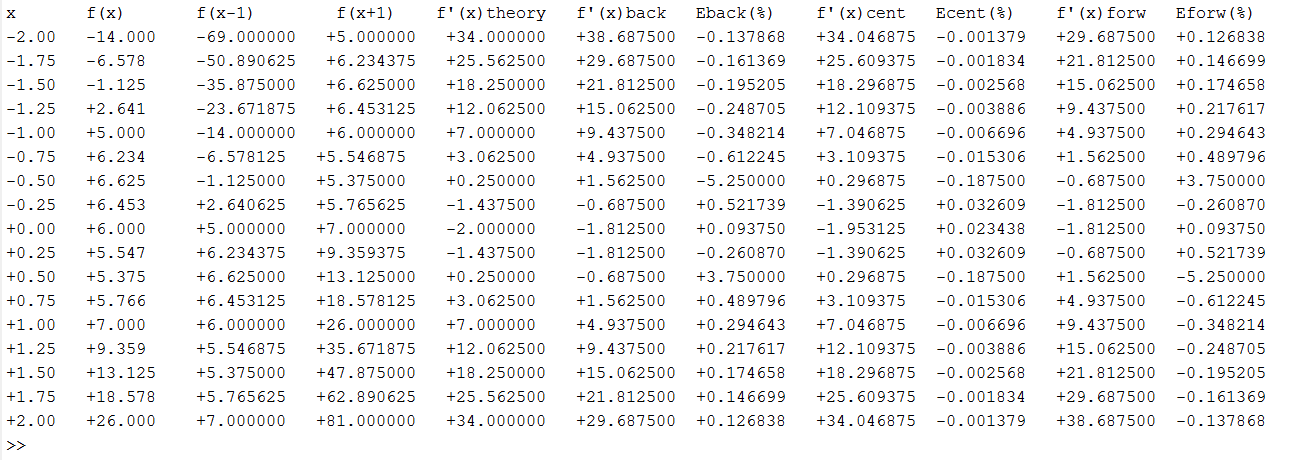
xlabel('x') %axis labels

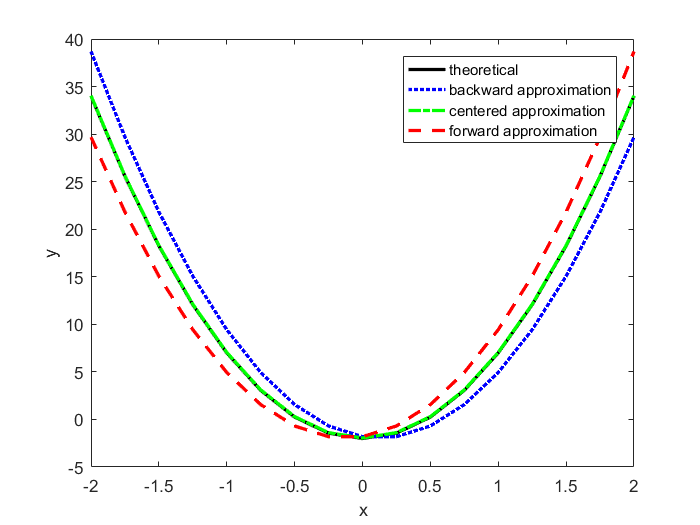
ylabel('y')

legend('theoretical','backward approximation','centered approximation','forward approximation');

These last lines of code create the plot and specify which line is of which values. The x and y axes are also labeled for better readability.

**Results:**





**Discussion:**

As shown by the results, the centered finite approximation for a function is the best approximation for the derivative of a function. This result is echoed in both the table and the plot of approximations. This is likely because the forward and backward approximations both account for too little or too much whereas the centered finite approximation is a happy medium.

For this problem, we reviewed our knowledge on how to manipulate vectors in order to output a set of values given an equation to work with. Furthermore, we refreshed our knowledge on how to create tables using the fprintf function, on a much larger scale than before. Lastly, we refreshed our knowledge on how to distinguish between graphs on the same plot in order to express the greatest understandability for the reader.