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

**Lab Report**

**Lab\_10: Numerical Integration**

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

**EID: VL5649**

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

**Problem 1.**

Two very important quantities in studying the growth of microorganisms in fermentation processes are the carbon dioxide evolution rate and the oxygen uptake rate. These are calculated from experimental analysis of the inlet and exit gases of the fermentor, and the flow rates, temperature, and pressure of these gases. The ratio of carbon dioxide evolution rate to oxygen update rate yields the respiratory quotient, which is a good barometer of the metabolic activity of the microorganism. In addition, the above rates can be integrated to obtain the total amounts of carbon dioxide produced and oxygen consumed during the fermentation. These total amounts form the basis of the material balancing techniques used in modeling of fermentation processes. Table 1 shows a set of rates calculated from the fermentation of *Penicillium chrysogenum*, which produces penicillin antibiotics.

Write a general MATLAB function named *‘MySimpsons.m’* for integrating experimental data using Simpson’s 1/3 rule, and use it to calculate the total amounts of carbon dioxide produced and oxygen consumed during the ten-hour period of fermentation. Compare the results of this function and those obtained by using the existing MATLAB function *trapz* (trapezoidal rule).

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time of Fermentation (h) | 140 | 141 | 142 | 143 | 144 | 145 | 146 | 147 | 148 | 149 | 150 |
| Carbon dioxide evolution rate (g/h) | 15.72 | 15.53 | 15.19 | 16.56 | 16.21 | 17.39 | 17.36 | 17.42 | 17.60 | 17.75 | 18.95 |
| Oxygen uptake rate (g/h) | 15.49 | 16.16 | 15.35 | 15.13 | 14.20 | 14.23 | 14.29 | 12.74 | 14.74 | 13.68 | 14.51 |

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

1. Discuss the Simpson’s 1/3 rule and Trapezoidal rule in brief.
2. Do the results match? Explain your answer.
3. Will you use simple or composite Simpson’s 1/3 rule? Why?

**MATLAB code:**

**Function:**

function a = MySimpsons\_VL(x,y)

n = length(x); %number of terms

if length(y)~=n, error('x and y must be same length'); end

w = 0; %initializes summation values

v = 0;

for i = 2:2:n-1 %summation of even index values

w = w + y(i);

end

for i = 3:2:n-2 %summation of odd index values

v = v + y(i);

end

a = (x(n)-x(1)) \* (y(1)+4\*w+2\*v+y(n))/(3\*(n-1)); %composite simpson's 1/3 rule

**Mainscript:**

t = [140;141;142;143;144;145;146;147;147;149;150]; %given values

C = [15.72;15.53;15.19;16.56;16.21;17.93;17.36;17.41;17.60;17.75;18.95];

O = [15.49;16.16;15.35;15.13;14.20;14.23;14.29;12.74;14.74;13.68;14.51];

f = MySimpsons\_VL(t,C); %solutions using Simpson's 1/3 rule

g = MySimpsons\_VL(t,O);

h = trapz(t,C); %solutions using trapezoidal rule

i = trapz(t,O);

fprintf('The CO2 and O2 consumed calculated using the Simpson''s 1/3 rule is\n%f and %f, respectively\n',f,g) %outputs result

fprintf('The CO2 and O2 consumed calculated using the trapezoidal rule is \n%f and %f, respectively\n',h,i)

**MATLAB function:**

The purpose of this function was to compare the calculated results of the total amounts of CO2 and Oxygen consumed of penicillin over the given time, using 2 different integration methods—Simpson’s 1/3 rule and the trapezoidal rule. To do so, we had to implement the Simpson’s rule using our own function and then compare our results with those of MATLAB’s built in trapz function.

function a = MySimpsons\_VL(x,y)

This first line of code outlines the variables used as the inputs of our function and the variable to be used as the result.

n = length(x); %number of terms

This line of code finds the number of terms in our data.

if length(y)~=n, error('x and y must be same length'); end

This line of code creates an error if we don’t have the same number of terms in our ‘x and y’ (input) arrays.

w = 0; %initializes summation values

v = 0;

These 2 lines of code initialize our ‘w’ and ‘v’ variables which will be used to store the summation values that are used in the composite simpson’s 1/3 rule calculation.

for i = 2:2:n-1 %summation of even index values

w = w + y(i);

end

These 3 lines of code from a for loop that sums up all of the ‘even’ indexed values of y.

for i = 3:2:n-2 %summation of odd index values

v = v + y(i);

end

Like the 3 lines of code before it, these 3 lines of code form a for loop that sums up all of the ‘odd’ indexed values excluding the first and final values.

a = (x(n)-x(1)) \* (y(1)+4\*w+2\*v+y(n))/(3\*(n-1)); %composite simpson's 1/3 rule

This line of code estimates the integral using the composite Simpson’s 1/3 rule. This formula works by dividing the interval that we are integrating on and weighting each segment with different weights (4 for the ‘odd’ intervals or even indexes and 2 for the ‘even’ intervals or odd indexes) and averaging the value.

t = [140;141;142;143;144;145;146;147;147;149;150]; %given values

C = [15.72;15.53;15.19;16.56;16.21;17.93;17.36;17.41;17.60;17.75;18.95];

O = [15.49;16.16;15.35;15.13;14.20;14.23;14.29;12.74;14.74;13.68;14.51];

These first 3 lines of the mainscript correspond to the experimental data given to us as part of the problem.

f = MySimpsons\_VL(t,C); %solutions using simpson's 1/3 rule

g = MySimpsons\_VL(t,O);

These 2 lines of code call the “MySimpsons\_VL” function that we wrote to apply Simpson’s 1/3 rule to the data we are given.

h = trapz(t,C); %solutions using trapezoidal rule

i = trapz(t,O);

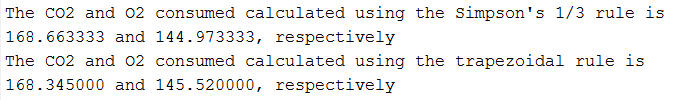
These 2 lines of code call MATLAB’s built in trapz function that applies trapezoidal numerical integration on the data.

fprintf('The CO2 and O2 consumed calculated using the Simpson''s 1/3 rule is\n%f and %f, respectively\n',f,g) %outputs result

fprintf('The CO2 and O2 consumed calculated using the trapezoidal rule is \n%f and %f, respectively\n',h,i)

These last 2 lines of code print out our results in the command window, using fprintf.

**Results:**

****

\*\*done under the assumption that x values are in order and evenly spaced

**Discussion:**

As shown by the results, the total amount of CO2 consumed by the penicillin was a value close to 168 and the total amount of oxygen consumed was somewhere around 145. Our estimates of integration for Simpson’s 1/3 rule and the trapezoidal rule are very close to each other (off by a value of .3183 and .5467, respectively which is <.38% relative error) and likely very close to the true value, a result of them being composite functions. The Simpson’s 1/3 rule works by assuming that the function that we are trying to integrate is polynomial in nature. The simple takes 3 points—the end points, weighted once, and the midpoint, weighted 4x, in order to estimate the area of a parabola made of points along our function. This idea is extended in the composite function by weighting every other value by 4 and those in between by 2 in order to more effectively estimate in integral. The trapezoidal function works by drawing a line between the 2 end points that we are integrating on and finding the area of the trapezoid underneath. This value is very easy to computer; however, it is often very far from the true value. The trapz function uses the composite trapezoidal function which works by making the same trapezoids, between even intervals along the function. By having additional trapezoids, we can better approximate the true value of the integral. We choose to use the composite version of Simpson’s 1/3 rule first, because trapz is the composite trapezoidal rule so to effectively compare the rules we need to use the equivalent composite rule in Simpson’s. Additionally, the simple functions are very rough approximations of the integrals and often very far from the true value—by using the composite functions we are able to attain a better estimation of the integral (higher order of accuracy).

From this problem, we learned how to implement Simpson’s 1/3 rule in a function that we wrote, applying array manipulation that we have used before. Additionally, we learned how to use MATLAB’s built in trapz function in order to apply the trapezoidal rule to estimate an integral. Lastly, we reviewed how to output results into the command window, so that they can be easily read by the user, using MATLAB’s fprintf function.

**Problem 2.**

The force on a sailboat mast can be represented by the following function:



where *z* = the elevation above the deck and *H* = the height of the mast. Compute *F* for the case where *H* = 30 using:

1. Romberg integration to a tolerance of using the provided MATLAB functions *romberg.m*. and *trap.m*. Report the integral estimates and the approximate percentage errors at every iteration.
2. Two-point Gauss-Legendre formula. Besides the integral value, also report the variable transformation used for the two-point Gauss-Legendre formula.
3. MATLAB *quad* and *quadl* function

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

1. Discuss Romberg Integration technique in brief. Discuss its efficiency over Trapezoidal rule.
2. Why do we need transformation to apply Two-point Gauss-Legendre formula? Discuss the transformation.
3. What is the difference between *quad* and *quadl* function?

**MATLAB code:**

**Function:**

function [q,Error,iter] = romberg\_VL(func,a,b,es,maxit,varargin)

% romberg: Romberg integration quadrature

% q = romberg(func,a,b,es,maxit,p1,p2,...):

% Romberg integration.

% input:

% func = name of function to be integrated

% a, b = integration limits

% es = desired relative error (default = 0.000001%)

% maxit = maximum allowable iterations (default = 30)

% pl,p2,... = additional parameters used by func

% output:

% q = integral estimate

% ea = approximate relative error (%)

% iter = number of iterations

if nargin<3,error('at least 3 input arguments required'),end

if nargin<4||isempty(es), es=0.000001; end %default values

if nargin<5||isempty(maxit), maxit=50; end

n = 1; %initializes number of segments

I(1,1) = trap\_VL(func,a,b,n,varargin{:}); %simple trapezoid, first value

iter = 0; %iteration counter

while iter<maxit

iter = iter+1; %update iteration coutner

n = 2^iter; %updates segments

I(iter+1,1) = trap\_VL(func,a,b,n,varargin{:}); %Adds another O(h^2) value from richardson extrapolation

for k = 2:iter+1 %column position counter

j = 2+iter-k; %row position coutner

I(j,k) = (4^(k-1)\*I(j+1,k-1)-I(j,k-1))/(4^(k-1)-1); %romberg integration algorithm

end

ea = abs((I(1,iter+1)-I(2,iter))/I(1,iter+1))\*100; %calculates percent approximate error

Error(iter) = ea; %stores error

if ea<=es, break; end %error criterion

end

q = I; %final result

**Function:**

function I = trap\_VL(func,a,b,n,varargin)

% trap: composite trapezoidal rule quadrature

% I = trap(func,a,b,n,pl,p2,...):

% composite trapezoidal rule

% input:

% func = name of function to be integrated

% a, b = integration limits

% n = number of segments (default = 100)

% pl,p2,... = additional parameters used by func

% output:

% I = integral estimate

if nargin<3,error('at least 3 input arguments required'),end

if ~(b>a),error('upper bound must be greater than lower'),end %bounds must be from low to high

if nargin<4|isempty(n),n=100;end %default values

x = a; h = (b - a)/n; %initializes 'x', h is the spacing between segments

s=func(a,varargin{:}); %plugs lower limit in

for i = 1 : n-1

x = x + h; %next segment

s = s + 2\*func(x,varargin{:}); %weighted 2x every value except lower and upper limits

end

s = s + func(b,varargin{:}); %adds upper limit

I = (b - a) \* s/(2\*n); %height x length (trapezoid) for area

**Mainscript:**

H = 30; %height of mast

F = @(z) 100\*(z./(5+z)).\*exp(-2\*z/H); %function

%romberg

[a,error,n] = romberg\_VL(F,0,H,.5);

%Gauss-Legendre

w = [-1/sqrt(3) 1/sqrt(3)]; %weights in 2-point gauss-legendre

x = @(xd) ((H+0)+(H-0).\*xd)./2; %x->xd

v = (H-0)/2; %dx -> dxd

I = sum(F(x(w))\*v); %transformed Gauss-Legendre

%quads

d = quad(F,0,H,0.5); %simpson quad

e = quadl(F,0,H,0.5); %lobatto quad

iter = [0,1:n]'; %iteration vector

er = [inf,error]'; %error vector

A = [iter,a',er,]'; %matrix for output

fprintf('Using Romberg Integration:\nIteration\t\t\tIntegral Estimate Calculations\t\t\t Approximate Error(%%)\n') %output

fprintf('\t%d\t\t%f\t%f\t%f\t%f\t\t %f\n',A)

fprintf('(The first value of every row is the Integral Estimate for the corresponding Iteration)\n')

fprintf('The value of the integral using Two-point Gauss-Legendre is \n%f.\n',I)

fprintf('The value of the integral using quad and quadl is\n%f and %f, respectively.\n',d,e)

**MATLAB function:**

The purpose of this function was to compare different integration estimation methods used to calculate the force on a sailboat mast. To do so, we created our function as an anonymous function and passed it to various functions both written in scripts and built into MATLAB, and then compared our results for each.

function [q,Error,iter] = romberg\_VL(func,a,b,es,maxit,varargin)

This first line of the Romberg function outlines the outputs and inputs of the function.

if nargin<3,error('at least 3 input arguments required'),end

This line of code produces an error if there are less than 3 inputs. This is necessary because an integral cannot be calculated without a lower and upper limit and the function we are taking an integral for.

if nargin<4||isempty(es), es=0.000001; end %default values

if nargin<5||isempty(maxit), maxit=50; end

These 2 lines of code fill in default values for the error that we are trying to reach and the maximum number of iterations, if they are not inputted by the user

n = 1; %initializes number of segments

This line of code initializes the number of segments that we will be using in the trapezoidal function, with the value corresponding to 2^0 as part of the Richardson extrapolation.

I(1,1) = trap\_VL(func,a,b,n,varargin{:}); %simple trapezoid, first value

This line of code essentially uses the simple trapezoid to get our initial estimate of the area underneath the curve we are trying to estimate (very poor estimate).

iter = 0; %iteration counter

while iter<maxit

These 2 lines of code initialize our iteration counter and then form a while loop that repeats while the iteration counter is below the maximum number of iterations specified earlier..

iter = iter+1; %update iteration coutner

n = 2^iter; %updates segments

These 2 lines of code update our iteration counter and then update the number of segments to use as part of the Richardson extrapolation.

I(iter+1,1) = trap\_VL(func,a,b,n,varargin{:}); %Adds another O(h^2) value from richardson extrapolation

This line of code adds another ‘O(h^2)’ value to our first column. These values are obtained by estimating the integral using the number of segments calculated in the line before, as part of the Richardson extrapolation.

for k = 2:iter+1 %column position counter

j = 2+iter-k; %row position coutner

These 2 lines of code are the counter for our row and column counters (j,k). As part of a for loop, these counters move us along a diagonal that we calculate values for.

I(j,k) = (4^(k-1)\*I(j+1,k-1)-I(j,k-1))/(4^(k-1)-1); %romberg integration algorithm

end

This line of code moves along the diagonal specified by the counters in the previous 2 lines, and calculates higher order approximations of our integral using previous values (the initial values are our Richardson extrapolation), as part of the Romberg integration method.

ea = abs((I(1,iter+1)-I(2,iter))/I(1,iter+1))\*100; %calculates percent approximate error

Error(iter) = ea; %stores error

These 2 lines of code calculate the approximate relative error and then store it into an array to be outputted.

if ea<=es, break; end %error criterion

This line of code breaks the while loop if a sufficient error criterion (specified earlier or set to a default value) is met, before the maximum number of iterations is met.

end

This line of code closes the while loop with our maximum iterations as the conditional.

q = I; %final result

This line of code sets the value ‘q’ as the value I, our calculated result. In doing so, we ensure that the result is not outputted into the script that calls it, prematurely, which would introduce a plethora of errors.

function I = trap\_VL(func,a,b,n,varargin)

This first line of the trap function outlines the single output and multiple inputs.

if nargin<3,error('at least 3 input arguments required'),end

if ~(b>a),error('upper bound must be greater than lower'),end %bounds must be from low to high

These 2 lines of code produce errors. The first of the 2 lines produces an error if there aren’t enough inputs for the trap function to work with (you need the function that you’re working with as well as the upper and lower bounds of integration otherwise the function can’t be computer). The second of the 2 lines produces an error if the upper bound of integration is lower than the lower bound (because that doesn’t make sense).

if nargin<4|isempty(n),n=100;end %default values

This line of code sets the default number of segments to use in calculation if it is not defined by the user.

x = a; h = (b - a)/n; %initializes 'x', h is the spacing between segments

This line of code initializes our x value as the first value we will need to perform a calculation with—the lower limit. The variable ‘h’ is set as the spacing between every one of our points as they are all equal.

s=func(a,varargin{:}); %plugs lower limit in

This line of code plugs the lower limit of integration into the function. This line of code is only specific to the lower limit because the rest of the corresponding ‘s’ values that will be added later are weighted twice and this one is weighted only once (as per the composite trapezoidal rule).

for i = 1 : n-1

x = x + h; %next segment

s = s + 2\*func(x,varargin{:}); %weighted 2x every value except lower and upper limits

end

These 4 lines form a for loop that weights each of the func(x) values x2 as part of the trapezoidal rule. This is done for all of the values between the limits of our integral and summed up as the variable ‘s’.

s = s + func(b,varargin{:}); %adds upper limit

This line of code adds the function value of our upper limit to the sum of the rest of the values. Like the first value, this value must also be done separately because the limits are only weighted once.

I = (b - a) \* s/(2\*n); %height x length (trapezoid) for area

This line of code takes the summed up function values with their weights that we calculated and averages them. It then multiplies it by our lower limit subtracted from our limit. This effectively is multiplied the height (function values) by the length (limit where we’re integrating) to calculate the area of our ‘trapezoid’.

H = 30; %height of mast

This first line of code in our mainscript is the top of the mast which is the highest place we can calculate the force for. This is used in our function and is also our upper limit of integration.

F = @(z) 100\*(z./(5+z)).\*exp(-2\*z/H); %function

This line of code creates the function that we are trying to integrate as an anonymous function so that it can be passed to various functions.

%romberg

[a,error,n] = romberg\_VL(F,0,H,.5);

This line of code calls the Romberg function that we have, outputting all of our integral estimates, error value at each iteration, and the number of total iterations.

%Gauss-Legendre

w = [-1/sqrt(3) 1/sqrt(3)]; %weights in 2-point gauss-legendre

These 2 values are the weights of the endpoints in 2-point gauss-legendre method which will eventually be substituted as the ‘xd’ value after we transform our function.

x = @(xd) ((H+0)+(H-0).\*xd)./2; %x->xd

This line of code creates an anonymous function that transforms x to xd as part of our limit transformation.

v = (H-0)/2; %dx -> dxd

This line of code transforms dx to dxd which is also part of our limit transformation.

I = sum(F(x(w))\*v); %transformed Gauss-Legendre

This line of code plugs our weights, w, as xd in our transformation of x to xd. This is then plugged into our function, F, effectively transforming our function. The value is then multiplied by v, our transformation of dx to complete the transformation. Because we get 2 values, 1 for each of the points, we sum these values together, as part of the Gauss-Legendre method to get our integral estimate, I.

%quads

d = quad(F,0,H,0.5); %simpson quad

e = quadl(F,0,H,0.5); %lobatto quad

These 2 lines of code call MATLAB’s quad and quadl functions which correspond to the simpson and lobatto quadratures used to estimate integrals.

iter = [0,1:n]'; %iteration vector

This line of code generates our iteration vector. Because the O(h^2) values are values from Richardson extrapolation, they are not actually values that are part of our iteration calculations so the value ‘0’ is added before.

er = [inf,error]'; %error vector

This line of code generates our error vector. While the Romberg function outputs a vector of our errors, it calculates using percent relative error and does not include the first row (as there is no previous value). Therefore, we set the first error as ‘inf’ as it is technically not calculatable.

A = [iter,a',er,]'; %matrix for output

This line of code compiles our arrays and matrix, calculated by the Romberg Integration, so that we can output the values using fprintf.

fprintf('Using Romberg Integration:\nIteration\t\t\tIntegral Estimate Calculations\t\t\t Approximate Error(%%)\n') %output

fprintf('\t%d\t\t%f\t%f\t%f\t%f\t\t %f\n',A)

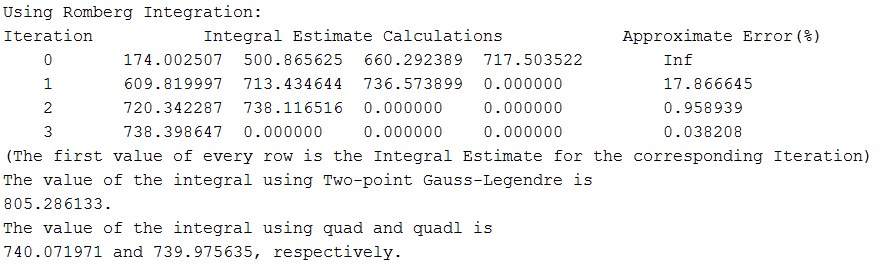
fprintf('(The first value of every row is the Integral Estimate for the corresponding Iteration)\n')

fprintf('The value of the integral using Two-point Gauss-Legendre is \n%f.\n',I)

fprintf('The value of the integral using quad and quadl is\n%f and %f, respectively.\n',d,e)

These last 5 lines of code format and output the results of all of our calculations, into the command window.

**Results:**

****

**Discussion:**

As shown by the results, we can approximate integration using different functions, to varying results. Functions such as Romberg integration are efficient, utilizing simple trapezoidal rule calculations and approximating more accurate results using less accurate ones, leading to results close to the true value. Functions such as the two-point gauss-legendre method are less accurate (the least accurate of all of the functions we tested) having a low relative order of accuracy. These methods, however can be extended to yield better results (such as increasing the number of points to three-point, etc.). Functions such as quad and quadl which user quadrature to ‘polynomial-ly’ estimate the value of our integral are also able to yield values very close to the true value because of a high order of accuracy as well. For all of the functions that we tested, how close we are to the true value can be improved be increasing the number of intervals within the limits that we calculate each method on, but in doing so we are also increasing the number of FLOPS needed to compute each. We need to perform limit transformation to utilize the two-point gauss-legendre formula because the derivation of the Gauss-Legendre formula uses the limits [-1, 1] and the limits of our function are [0, 30]. By applying the transformation, we can transform the integration interval without changing the value of the integral, allowing us to apply the Gauss-Legendre formula. The function quad uses recursive adaptive Simpson quadrature while quadl uses recursive adaptive Lobatto quadrature (L for Lobatto). Both functions approximate within an error of 10^-6 but quad is usually better for low accuracies or non-smooth functions while quadl is usually better for high accuracies or smooth functions. Both functions work by successively increasing the number of segments used in their respective calculation until the sufficient error is reached.

From this problem, we learned how to implement a large variety of integral approximating methods in MATLAB. We reviewed how to utilize loops (both for and conditional loops) in order to iterate calculations to reach a desired criterion. Additionally, we reviewed basic array and matrix operations in MATLAB. Furthermore, we reviewed how to use anonymous functions so that they can be passed to functions. Lastly, we reviewed how to output results into the command window using fprintf.