

**Fall 2019**

**ENGR 3202-02 (Instructor: Dr. Mostafa Youssef)**

**Final Project: Properties of Pure Substances (Thermodynamics)**

**Team Members:**

Mohamed Abdel Hamed 900163202

Mahmoud Ahmed 900172747

Khloud Maged 900162914

Eslam A. Soliman 900163257

**Tasks and Contribution**

**-** numerical differentiation and integration (Mahmoud)

- analytical differentiation and integration (Khloud)

- cubic splines (Mohamed)

- quadratic and linear splines (Eslam)

- utility functions for getting the required range, dataset, F(x), etc. (Mohamed)

- main module integration (Mohamed)

- plotting functions (Mahmoud, Mohamed)

**Abstract**

This project basically is about computing the thermodynamic boundary work and the bulk modulus using experimentally determined data for a very specific thermodynamic process in which water remains in the vapor state at the edge of condensation. As known, water as any other substance has two characteristic lines that determine the boiling behavior of this substance; always its graph representation has a critical point that has on side of it the zone of water being liquid and the other water being vapor. Mainly, thermodynamics is the science concerned with studying energy and in this project, we are trying to compute the so-called boundary work and the corresponding pressures; the developed software mainly the project will be computing those in different methods: using cubic splines in calculating the pressure and for the boundary work will be using two different methods: integration of the cubic splines and standard numerical integration algorithms, specifically, trapezoidal method for its efficiency.

**Background on the Addressed Thermodynamic Problem**

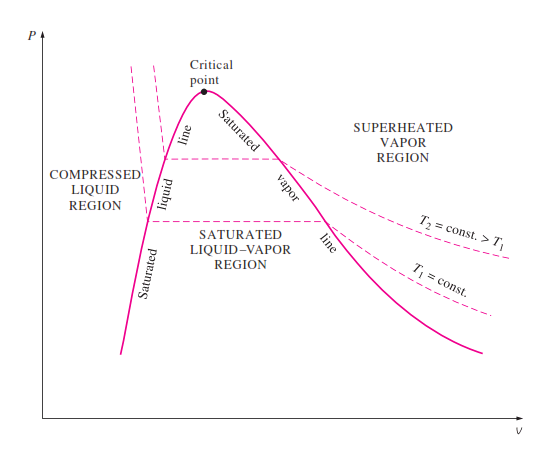
Basically, the project is based on the subject of the phases of pure substances, more specifically, with the properties of pure substances during the process of vaporizing the substance (transforming the substance from the liquid state to the vapor one). This specific situation has three sub-states itself. 1- the substance being in the liquid state, 2- the substance being in the vapor state, 3- the state in between where the substance is in an inseparable state between both being liquid and vapor. The problem in hand is more concerned with the bridge between the 2nd and 3rd state. In other words, about the properties of a certain pure substance, which is water, during the transition from being in the inseparable state to being in the vapor state. Thermodynamics provides a diagram for describing both the pressure and volume of the substance during all of these states. This diagram is the *p-v diagram for pure substance (figure 1)*, which provides information about the relation between both the pressure and the volume of the substance during its transitions between the liquid and vapor states. Another useful information that can be extracted from this relation are; the boundary work and the bulk modulus of the substance, which are the work exerted by the substance on its surroundings while expanding during the transition (Boundary Work), and a constant which represents how resistant to compression the substance is (Bulk modulus), respectively. The goal of the project is to get these two pieces of information for water from the data that represents the p-v relation using analytical and numerical methods of interpolating, integration and differentiation on these data.

Figure 1

**Algorithms Background**

**Cubic splines:**

In many engineering problems, we normally have measured values for a specific value collected from precise experiments. However, as shown in this problem, we might need to evaluate the function on a value other than the ones used in the experiment. In order to do this, we use splines and for accurate results we use cubic spline. The method is based on cutting the measured value to intervals. Each interval has its own equation based on coefficients and their second derivatives. The main advantage of this method is that it provides high accuracy for the values within each interval.

**Thomas algorithm:**

Thomas algorithm or as it called: Tri-Diagonal Matrix Algorithm (TDMA), and a Tri-Diagonal matrix is just a band matrix that has nonzero elements only on the main diagonal, and lower diagonal below, and the upper diagonal above the main one**.** This algorithm is basically the result of applying Gaussian elimination to a Tri-Diagonal system. It is very powerful when it comes to computation as it is very quick method, and it is also very important if there is a very large number of unknowns. On the other hand, it has some limitations as it becomes unstable in case if the Tri-diagonal matrix is singular (the determinant is zero) and in this case, it does not have an inverse. According to the large given data we will be using this method in our implementation. It is used into the calculation of the second derivatives needed for the cubic splines.

**Numerical integration:**

Usually, analytical integration can be used for simple problems. However, there are problems where integrating the given problem’s function analytically would be time consuming and might be too difficult. Therefore, using numerical integration would be more time efficient and easier to work with. In order to evaluate the numerical integration, there are 3 known methods for doing so, trapezoidal rule, 1/3 Simpson and 3/8 Simpson. In this project, we chose to use the trapezoidal method for its efficiency in data given as points.

**Numerical differentiation:**

Similar to the numerical integration, evaluating the derivative at a certain value can be complex and difficult especially if the values are given in tabular form without knowing the function beforehand. In this case, using numerical differentiation methods can make the process much easier with accurate results depending on the error type of the equation used. The main method for evaluating the derivative is through the absolute difference between the value of the function at two points surrounding the wanted value divided by the difference between the two points. For higher derivatives, the expression of the result is obtained from the Taylor expansion of the function.

**Code Functionality**

numerical\_int( x, y):

The function takes the values of the volumes as x and the corresponding pressures as y, with the first and last value being calculated from the splines equations. Afterwards, it calculated the values of the integral using the trapezoidal method by calculating the partitioning the values into segments and calculating each one of them separately and add them up in the end to get the integral.

numerical\_spline\_int( splines,values,x1, x2):

The function takes the coefficients of each spline interval, the values of the volume, and the two point between which we want to calculate the integral and a value representing the integration of the pressure along the value v1 and v2. The function uses the trapezoidal method since it gives accurate and efficient results.

numerical\_diff(splines,values):

The function takes the coefficients of each spline interval, the values of the volume and returns a vector containing the derivatives of the given points. The function uses a difference of 1e-12 and handles three cases. The first case is if the given point is the last in the dataset, in this case it uses the backward difference. The second case is if the given point is the first in the dataset, in this case it uses the forward difference. Otherwise, it uses the centered difference method.

compute\_2nd\_deriv(Vg, P):

the function take the values of both the volume and their corresponding pressures and returns a vector containing the second derivative of each point. This is using the cubic splines assumption that the first and last points’ derivatives are both zero. The function uses another Thomas function to solve the acquired equations.

linear\_spline(Vg, P):

The function takes as input the values of both the volume and their corresponding pressures and returns a matrix in which each row represents the coefficients of the corresponding linear spline. The function directly calculates the coefficients using the slope between every two points.

quadratic\_splines(Vg, P):

The function takes as input the values of both the volume and their corresponding pressures and returns a matrix in which each row represents the coefficients of the corresponding quadratic splines. The function uses gauss method to solve the generated system of equations

cubic\_spline(Vg, P, d2y):

The function takes as input the values of both the volume and their corresponding pressures in addition to the value of the second derivative at each given volume. It then returns a matrix in which each row represents the coefficients of the corresponding cubic splines.

valid\_range(v1, v2, data\_start, data\_end):

A Boolean function which takes as input the two inputted values, the start of the data , and the end of it. It returns 1 if both values are within the given dataset obtained from the table.

F(f, x):  
This function takes as input the coefficients of the spline containing the value x, along with x which we want to compute the function’s value for it.

analyt\_int\_sliced(inner\_splines, X):  
the function takes a matrix containing the coefficients of the splines and the vector of the given volumes X. It then loops over the given values and coefficients and computes the analytical integration of each segment.

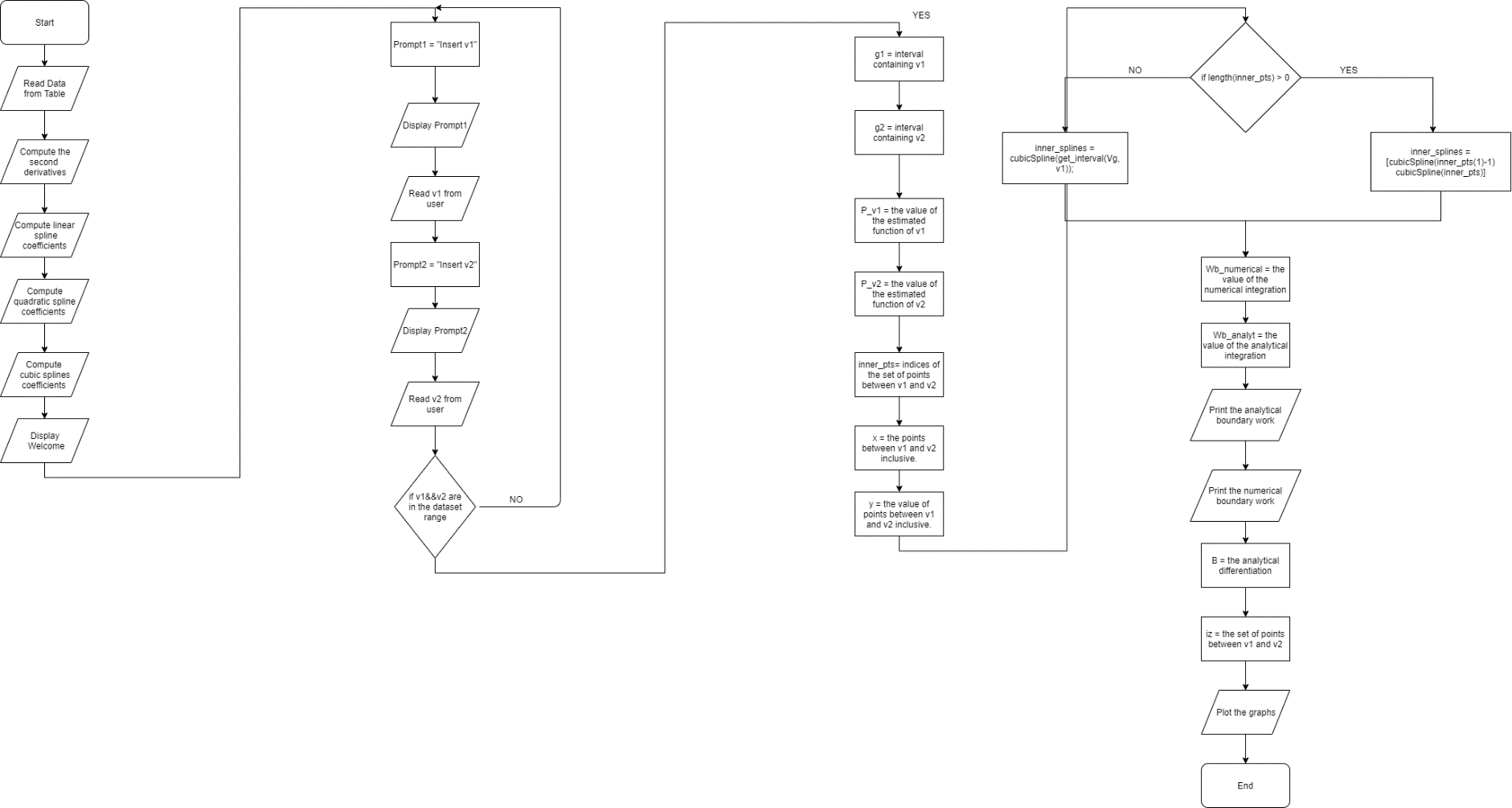
analyt\_int(f(i), X(i), X(i+1)):  
The function take a vector containing the coefficients of the spline containing the other two inputs, x(i) and x(i+1) representing the volumes at the end of each segment.

analyt\_diff\_sliced(inner\_splines, X):  
the function takes a matrix containing the coefficients of the splines and the vector of the given volumes X. It then loops over the given values and coefficients and computes the analytical differentiation of each segment.

analyt\_diff\_sliced(f, X):  
The function take a vector containing the coefficients of the spline containing the other input, X representing the volumes at which we want to get the derivative.

**Code FlowChart**

The following flow chart is flipped to be easier to read on paper.



**Results**

Requirements:

1. **Calculate the values of p1 and p2:**

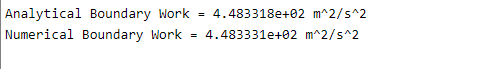
Using the cubic spline functions we managed to calculate the pressures corresponding to both v1 and v2, respectively and output them to the user

1. **Calculate the boundary work using the equation**

Using the splines’ equations we managed to calculate the boundary work, both analytically and numerically.

The results were surprisingly close which shows how efficient numerical methods are.

Below are the computed values for v1=0.05 and v2=0.5

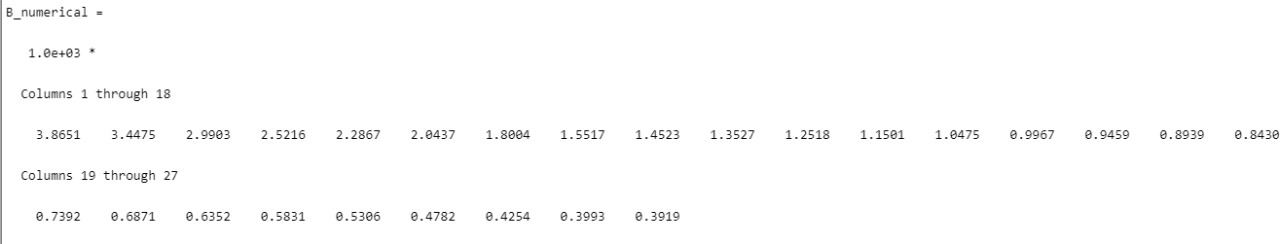


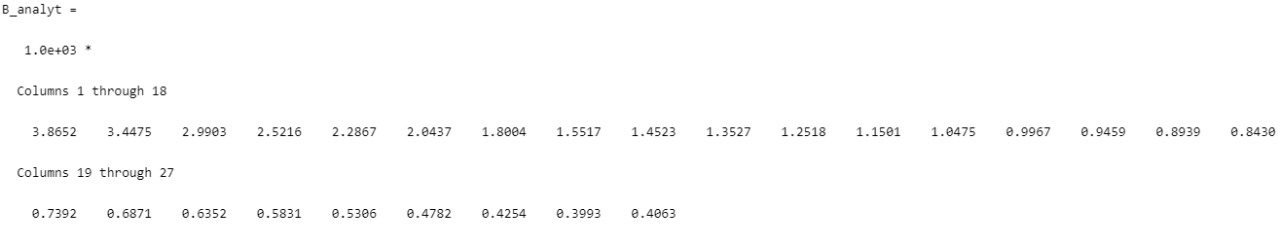
1. **Calculate the bulk modulus using the equation**

Using the splines’ equations we managed to calculate the derivative of the pressure with respect to the volume, both numerically and analytically.

For the numerical differentiation, we chose to use an h equal to 1e-12 to guarantee high accuracy after trying several other values. The results were indeed accurate and almost the same as the analytical ones.

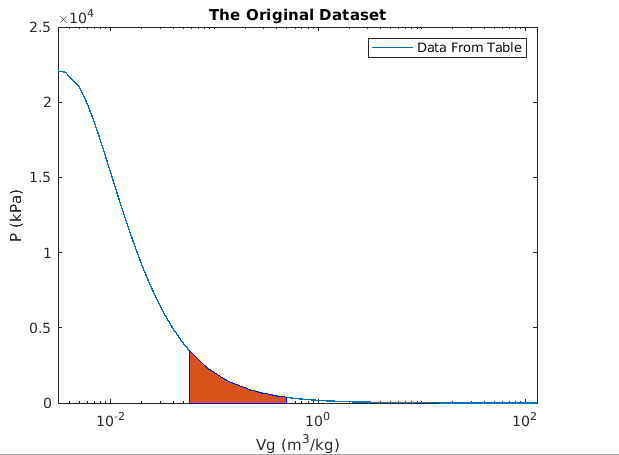
Below are the computed values for v1=0.05 and v2=0.5

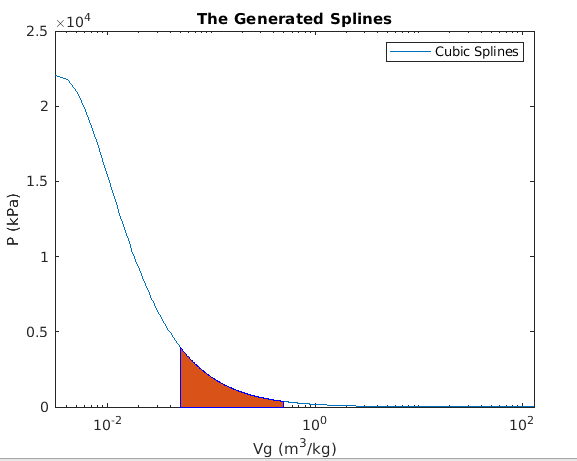


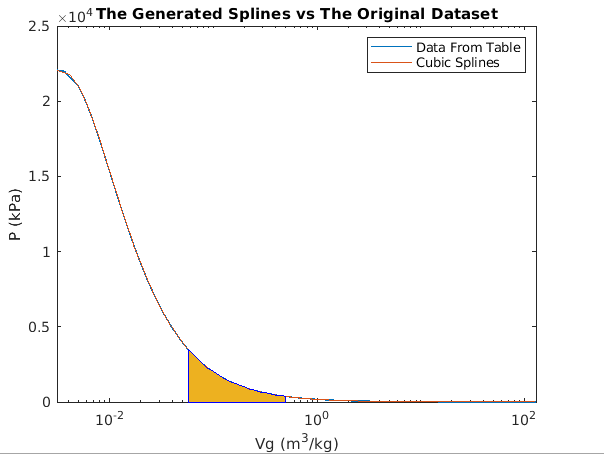


1. **Plots**

The graphs below are for v1 = 0.05, v2 = 0.5

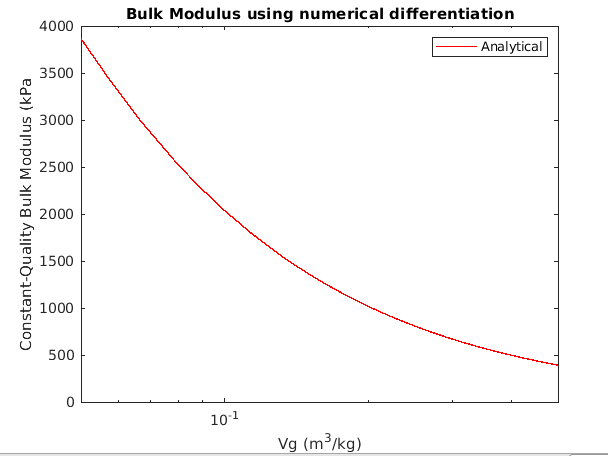


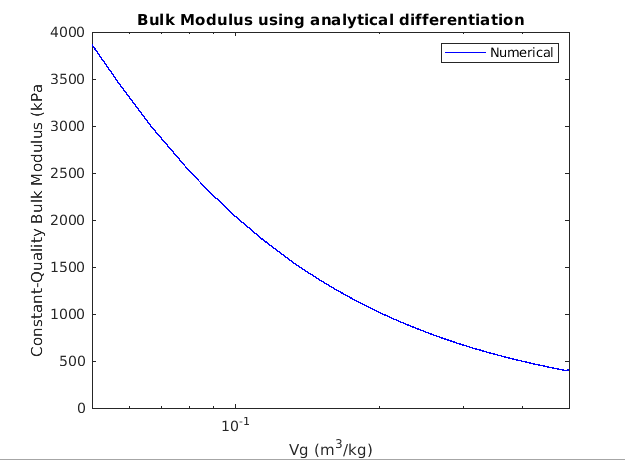


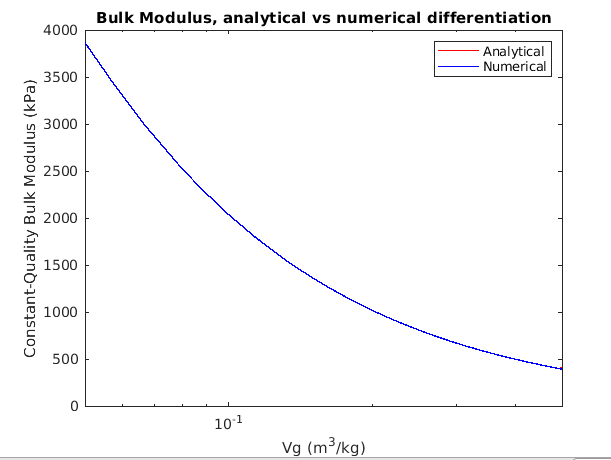


As shown, the cubic splines manage to fit the dataset smoothly allowing for accurate predictions of the pressure.

**Bulk Modulus Graphs:**







As shown, the results are very accurate and close the analytical figures.

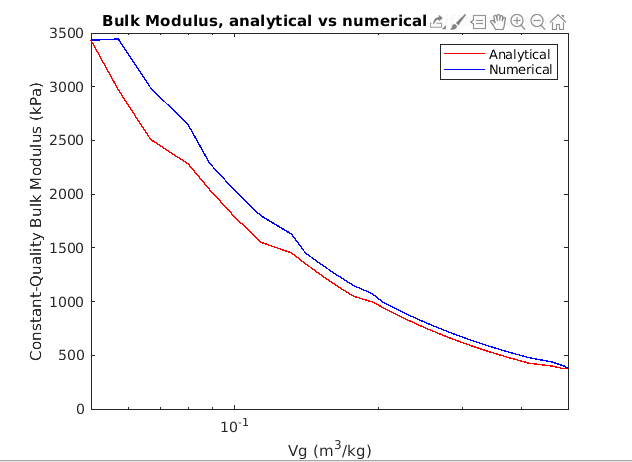
***Bonus:***

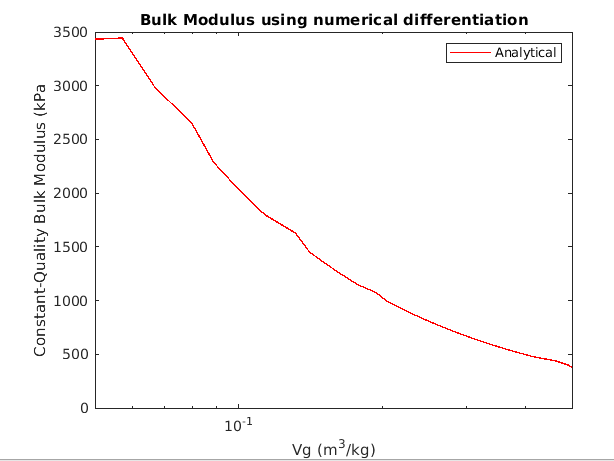
For the bonus part of our project, we modified the main flow of the program to be in form of 4 modes of operations described in the project statement:

1. Using cubic splines in the fitting, integration and differentiation.
2. Using cubic splines in the fitting but integration and differentiation are done using method (trapezoidal (or Simpson), and finite difference, respectively).
3. Using quadratic splines in the fitting, integration, and differentiation [Here you are asked to use Gauss elimination with partial pivoting to find the coefficients of the quadratic splines.]
4. Using linear splines in the fitting, integration and differentiation.

After each mode iteration, the user is asked whether he wishes to display the P-Vg graphs or the Bulk Modulus graphs.

Below are a few snapshots from mode 4 (v1 = 0.05, v2 = 0.5).





**Conclusion and Future Work**

The conclusion of the project is our software that is capable of extracting both the boundary work and the bulk modulus of water based on two points that are in the range of the data provided. One main takeaway for us was how accurate the results were using numerical methods compared to analytical methods. Possible future work might include performing extrapolation on the given data to be able to take any possible values for v even if it is not in the given data’s range. Also, we could import the data for different substances to be able to get the same information for several substances other than water.

**References**

Boundary Work. (n.d.). Retrieved from <http://www.mhhe.com/engcs/mech/cengel/notes/BoundaryWork.html>.

Bulk modulus. (2019, November 11). Retrieved from <https://en.wikipedia.org/wiki/Bulk_modulus>.

**APPENDIX A:**

**main.m**

clear; clc;

T = readtable('Table.dat');

P = table2array(T(:, 1));

P = flip(P');

Vg = table2array(T(:, 2));

Vg = flip(Vg');

d2y = compute\_2nd\_deriv(Vg, P);

linSpline = linear\_spline(Vg, P);

quadSpline = quadratic\_spline(Vg, P);

cubicSpline = cubic\_spline(Vg, P, d2y);

data\_start = Vg(1)

data\_end = Vg(end)

disp('Welcome!');

[v1, v2, neg] = take\_input(data\_start, data\_end);

% Fitting Modes:

modeFitSpline{1} = cubicSpline;

modeFitSpline{2} = cubicSpline;

modeFitSpline{3} = quadSpline;

modeFitSpline{4} = linSpline;

splineType{1} = "Cubic Splines";

splineType{2} = "Cubic Splines";

splineType{3} = "Quadratic Splines";

splineType{4} = "Linear Splines";

for m = 1:4

fprintf('You Are On Mode %d \n', m);

fitSpline = modeFitSpline{m};

P1 = F(fitSpline(get\_interval(Vg, v1), :), v1)

P2 = F(fitSpline(get\_interval(Vg, v2), :), v2)

inner\_pts = extract\_inner\_pts(Vg, v1, v2);

X = [v1 Vg(inner\_pts) v2];

Y = [P1 P(inner\_pts) P2];

if length(inner\_pts) > 0

inner\_eqs = [fitSpline(inner\_pts(1)-1, :); fitSpline(inner\_pts, :)];

else

inner\_eqs = fitSpline(get\_interval(Vg, v1), :);

end

if m ~= 2 % 2 is the only mode requiring all diff & int be done not through the splines

Wb\_numerical = neg\*numerical\_spline\_int(fitSpline, Vg, v1, v2);

else

Wb\_numerical = neg\*numerical\_int(X, Y);

end

Wb\_analyt = neg\*analyt\_int\_sliced(inner\_eqs, X);

fprintf('Analytical Boundary Work = %d m^2/s^2\n', Wb\_analyt);

fprintf('Numerical Boundary Work = %d m^2/s^2\n', Wb\_numerical);

B\_analyt = -X.\*analyt\_diff\_sliced(inner\_eqs, X);

B\_numerical = -X.\*numerical\_diff(inner\_eqs, X);

% Generate many data points for splines to show the slight difference

% with the table data points

Xspline = data\_start:0.001:data\_end;

first = 1;

for i = 1: length(Xspline)

Yspline(i) = F(fitSpline(get\_interval(Vg, Xspline(i)), :), Xspline(i));

if Xspline(i) >= v1 && Xspline(i) <= v2

if first == 1

spline\_pts(1) = i;

first = 0;

else

spline\_pts = [spline\_pts i];

end

end

end

prompt = 'Do you wish to display the P-Vg Graphs? insert [Y]es/[N]o ';

decision = input(prompt, 's');

if strcmp(decision,'Y') == 1 || strcmp(decision,'y') == 1

plot\_dataset\_alone(Vg, P, inner\_pts);

plot\_splines\_alone(Xspline, Yspline, spline\_pts, splineType{m});

plot\_splines\_with\_dataset(Xspline, Yspline, Vg, P, inner\_pts, splineType{m});

end

prompt = 'Do you wish to Bulk Modulus Graphs? insert [Y]es/[N]o ';

decision = input(prompt, 's');

if strcmp(decision,'Y') == 1 || strcmp(decision,'y') == 1

plot\_Bulk\_numerical(X,B\_numerical);

plot\_Bulk\_analyt(X, B\_analyt);

plot\_Bulk\_numerical\_with\_analyt(X, B\_numerical, B\_analyt);

end

end

function [coff] = analyt\_diff(f)

n = length(f);

coff(1) = 0;

for i = 2:n

coff(i-1) = f(i)\*(i-1);

end

end

function [diff] = analyt\_diff\_sliced(f, X)

n = size(f, 1)

for i = 1:n

diff(i) = F(analyt\_diff(f(i, :)), X(i));

end

diff(n+1) = F(analyt\_diff(f(n, :)), X(n));

end

function [val] = analyt\_int(f, x1, x2)

val = 0;

pow\_x1 = x1;

pow\_x2 = x2;

for i = 1:length(f)

diff = f(i)\*(pow\_x2-pow\_x1)/i;

val = val + diff;

pow\_x1 = pow\_x1\*x1;

pow\_x2 = pow\_x2\*x2;

end

end

function [val] = analyt\_int\_sliced(f, X)

n = size(f, 1);

val = 0;

for i = 1:n

val = val + analyt\_int(f(i, :), X(i), X(i+1));

end

end

function [d2y] = compute\_2nd\_deriv(x, y)

n = length(x);

d2y(1) = 0;

f(1) = 2\*(x(3) - x(1));

g(1) = (x(3) - x(2));

r(1) = (6\*(y(3) - y(2)))/(x(3) - x(2)) + (6\*(y(1) - y(2))/(x(2) - x(1)));

for k = 3:n-2

e(k-1) = x(k) - x(k-1);

f(k-1) = 2\*(x(k+1) - x(k-1));

g(k-1) = (x(k+1) - x(k));

r(k-1) = (6\*(y(k+1) - y(k)))/(x(k+1) - x(k)) + (6\*(y(k-1) - y(k))/(x(k) - x(k-1)));

end

e(n-2) = (x(n-1) - x(n-2));

f(n-2) = 2\*(x(n) - x(n-2));

r(n-2) = (6\*(y(n) - y(n-1)))/(x(n) - x(n-1)) + (6\*(y(n-2) - y(n-1))/(x(n-1) - x(n-2)));

d2y = [d2y thomas(g, f, e, r)];

d2y(n) = 0;

end

function [coff] = cubic\_spline(x, y, d2y)

n = length(x);

for i = 1:n-1

t1 = d2y(i)/(6\*(x(i+1) - x(i)));

t2 = d2y(i+1)/(6\*(x(i+1) - x(i)));

t3 = y(i)/(x(i+1) - x(i));

t4 = d2y(i)\*(x(i+1)-x(i))/6;

t5 = y(i+1)/(x(i+1) - x(i));

t6 = d2y(i+1)\*(x(i+1)-x(i))/6;

coff(i, 4) = t2-t1;

coff(i, 3) = 3\*(x(i+1)\*t1 - x(i)\*t2);

rem = 3\*(x(i)\*x(i)\*t2 - x(i+1)\*x(i+1)\*t1);

coff(i, 2) = rem + (t4 + t5 - (t3 + t6));

pow3\_1 = x(i)\*x(i)\*x(i);

pow3\_2 = x(i+1)\*x(i+1)\*x(i+1);

rem1 = pow3\_2\*t1 - pow3\_1\*t2;

coff(i, 1) = rem1 + x(i+1)\*(t3 - t4) - x(i)\*(t5 - t6);

end

end

function [pts] = extract\_inner\_pts(T, v1, v2)

n = length(T);

st = 1;

en = 1;

for i = 2:n

st = i+1;

if T(i) > v1

st = i

break;

elseif T(i) == v1

break;

end

end

for i = st-1:n

en = i-1;

if T(i) > v2

%en = i

break;

elseif T(i) == v2

break;

end

end

if st > en

pts = [];

else

pts = st:1:en;

end

end

% This function evaluates a given polynomial f at value x

function [poly] = F(f, x)

n = length(x);

for k = 1:n

poly(k) = 0;

pow\_x = 1;

for i = 1:length(f)

poly(k) = poly(k) + pow\_x\*f(i);

pow\_x = pow\_x\*x(k);

end

end

end

function x=gauss(A,B)

n = length(A);

% List of the input

% A is an nxn matrix

%B is an nx1 vector

% n is the number of linear equations

if det(A)==0

display('Coefficient Matrix is Singular; system might be inconsistent or have infinite solutions');

end

%PHASE 1: Elimination

for k=1:n-1 % Pivot equation

%Partial Pivoting

p = k;

max = abs(A(k,k));

for s=k+1:n

if(abs(A(s,k)) > max)

max = abs(A(s,k));

p = s;

end

end

if p ~= k

for ii=1:n

temp = A(k, ii);

A(k, ii) = A(p, ii);

A(p, ii) = temp;

end

temp = B(p, 1);

B(p, 1) = B(k, 1);

B(k, 1) = temp;

%A([k p], :) = A([p k], :); % Swap the two rows

end

for i=k+1:n % equation in which we want to eliminate xk

factor = A(i,k)/A(k,k); % A(k,k) is the pivot

for j=k+1:n

A(i,j)=A(i,j)-factor\*A(k,j); %cancellation of xk

end

B(i)=B(i)-factor\*B(k); % also apply the subtraction to R.H.S.

end % ended elimination of xk

end

% PHASE 2: Back substitution

x(n)=B(n)/A(n,n);

for i=n-1:-1:1 % Work on equations n-1 to 1

sum=B(i);

for j=i+1:n

sum=sum-A(i,j)\*x(j);

end

x(i)=sum/A(i,i);

end

function [idx] = get\_interval(T, x)

idx = -1;

if x < T(1) || x > T(end) %out of bounds

return

end

for i = 1:length(T)-1

if x >= T(i) && x <= T(i+1)

idx = i;

break;

end

end

end

function [coff] = linear\_spline(x, y)

n = length(x);

for i = 1:n-1

slope = (y(i+1) - y(i))/(x(i+1) - x(i));

coff(i, 1) = y(i) - slope\*x(i);

coff(i, 2) = slope;

end

end

function [res] = numerical\_diff(splines,values)

% corner cases:

% if the point is the first in the interval, we should use the forward

% difference

% if it's the last, we should use the backward differnce

for i=1:length(values)

flag =0;

h = 1e-6;

x1 = values(i)-h;

x2= values(i)+h;

n = size(values,2);

if(values(i)==values(n)) % backward

idx = get\_interval(values,x1);

if(idx~=-1)

spline = splines(idx,:);

y1 = F(spline,x1 );

idx = get\_interval(values,values(i));

if(idx~=-1)

spline = splines(idx,:);

y2 = F(spline,values(i) );

res(i) = (y2-y1)/(h);

else

flag=1;

end

else

flag=1;

end

elseif(values(i)==values(1)) % forward

idx = get\_interval(values,values(i));

if(idx~=-1)

spline = splines(idx,:);

y1 = F(spline,values(i) );

idx = get\_interval(values,x2);

if(idx~=-1)

spline = splines(idx,:);

y2 = F(spline,x2 );

res(i) = (y2-y1)/(h);

else

flag=1;

end

else

flag=1;

end

else

idx = get\_interval(values,x1); %centered

if(idx~=-1)

spline = splines(idx,:);

y1 = F(spline,x1 );

idx = get\_interval(values,x2);

if(idx~=-1)

spline = splines(idx,:);

y2 = F(spline,x2 );

res(i) = (y2-y1)/(2\*h);

else

flag=1;

end

else

flag=1;

end

end

if(flag==1)

fprintf("Cannot differintiate, will return -1");

res=-1;

end

end

end

function [res] = numerical\_int( x, y)

% this function uses the trapizoidal method to calculate the value of the

% integral

n = length(x);

sum =0;

for i=2:n-2

h= (x(i+1)-x(i))/2;

sum = sum + h\*(y(i)+y(i+1));

end

h1= (x(2)-x(1))/2;

val1= h1\*(y(1)+y(2));

hn= (x(n)-x(n-1))/2;

valn= hn\*(y(n)+y(n-1));

res = val1+valn+sum;

end

function [res] = numerical\_spline\_int( splines,values,x1, x2)

% this function uses the trapizoidal method to calculate the value of the

% integral

n= 1000;

flag = 0;

h = (x2-x1)/n;

sum =0;

for i=1:n-1

x = x1+i\*h;

idx = get\_interval(values,x);

if(idx~=-1)

spline = splines(idx,:);

sum = sum +F(spline,x );

else % handle out of range

flag=1;

break;

end

end

if(flag==0)

idx = get\_interval(values,x1);

if(idx~=-1)

spline = splines(idx,:);

y1 = F(spline,x1 );

idx = get\_interval(values,x2);

if(idx~=-1)

spline = splines(idx,:);

y2 = F(spline,x2 );

res = (h/2)\*(y1+y2+2\*sum);

else

flag =1;

end

else

flag=1;

end

end

if(flag==1)

fprintf("Cannot integrate, will return -1");

res=-1;

end

end

function [res] = range\_diff( splines,values,x1,x2)

count =1;

idx1=get\_interval(values,x1);

idx2=get\_interval(values,x2);

if(idx1~=-1 && idx2~=-1)

for i=x1:0.1:x2

res(count) = numerical\_diff(splines,values,i);

count = count+1;

end

else

fprintf("Error, Range is not in the table. Will return -1");

res=-1;

end

end

function [] =plot\_Bulk\_analyt(X, B\_numerical)

figure;

semilogx(X, B\_numerical, 'LineWidth', 1, 'Color', 'b')

title("Bulk Modulus using analytical differentiation");

xlabel("Vg (m^3/kg)")

ylabel("Constant-Quality Bulk Modulus (kPa")

legend("Numerical")

end

function [] =plot\_Bulk\_numerical(X, B\_analyt)

figure;

semilogx(X, B\_analyt, 'LineWidth', 1, 'Color', 'r')

title("Bulk Modulus using numerical differentiation");

xlabel("Vg (m^3/kg)")

ylabel("Constant-Quality Bulk Modulus (kPa")

legend("Analytical")

end

function []= plot\_Bulk\_numerical\_with\_analyt(X, B\_numerical, B\_analyt)

fg = figure;

ax = axes(fg);

set(0,'DefaultLegendAutoUpdate','off');

semilogx(ax, X, B\_analyt, 'LineWidth', 1, 'Color', 'r');

title("Bulk Modulus, analytical vs numerical differentiation");

xlabel("Vg (m^3/kg)");

ylabel("Constant-Quality Bulk Modulus (kPa)");

hold on;

semilogx(ax, X, B\_numerical, 'LineWidth', 1, 'Color', 'b');

hold off;

legend("Analytical", "Numerical");

end

function [] = plot\_dataset\_alone(Vg, P, shade\_range)

fg = figure;

ax = axes(fg);

semilogx(ax, Vg, P, 'LineWidth', 1);

title("The Original Dataset");

set(0,'DefaultLegendAutoUpdate','off')

legend("Data From Table");

xlabel("Vg (m^3/kg)")

ylabel("P (kPa)")

hold on;

area(ax, Vg(shade\_range), P(shade\_range), 'EdgeColor', 'b')

hold off

end

function [] = plot\_splines\_alone(Xspline, Yspline, shade\_range, leg)

fg = figure;

ax = axes(fg);

semilogx(ax, Xspline, Yspline, 'LineWidth', 1);

title("The Generated Splines");

set(0,'DefaultLegendAutoUpdate','off')

legend(leg);

xlabel("Vg (m^3/kg)")

ylabel("P (kPa)")

hold on;

area(ax, Xspline(shade\_range), Yspline(shade\_range), 'EdgeColor', 'b')

hold off

end

function [] = plot\_splines\_with\_dataset(Xspline, Yspline, Vg, P, shade\_range, leg)

fg = figure;

ax = axes(fg);

set(0,'DefaultLegendAutoUpdate','off')

semilogx(ax, Vg, P, 'LineWidth', 1);

title("The Generated Splines vs The Original Dataset");

xlabel("Vg (m^3/kg)")

ylabel("P (kPa)")

hold on

semilogx(ax, Xspline, Yspline, 'LineWidth', 1);

hold off

legend("Data From Table", leg);

hold on;

area(ax, Vg(shade\_range), P(shade\_range), 'EdgeColor', 'b')

hold off

end

function [c] = quadratic\_spline(fx, x)

n = int16(length(x) - 1); %number of splines

a = zeros(3 \* n - 1);

b = zeros(3 \* n - 1, 1);

%1st condition

for i = 1:n

if i == 1

a(2 \* i - 1, 1) = x(i);

a(2 \* i - 1, 2) = 1;

b(2 \* i - 1) = fx(i);

a(2 \* i, 1) = x(i + 1);

a(2 \* i, 2) = 1;

b(2 \* i) = fx(i + 1);

else

a(2 \* i - 1, 3 \* i - 3) = x(i) \* x(i);

a(2 \* i - 1, 3 \* i - 2) = x(i);

a(2 \* i - 1, 3 \* i - 1) = 1;

b(2 \* i - 1) = fx(i);

a(2 \* i, 3 \* i - 3) = x(i + 1) \* x(i + 1);

a(2 \* i, 3 \* i - 2) = x(i + 1);

a(2 \* i, 3 \* i - 1) = 1;

b(2 \* i) = fx(i + 1);

end

end

%2nd condition

offset = 2 \* n;

for i = 1 : n - 1

if i == 1

a(i + offset, 1) = 1;

a(i + offset, 3) = -2 \* x(i + 1);

a(i + offset, 4) = -1;

else

t = (i - 1) \* 3;

a(i + offset, t) = 2 \* x(i + 1);

a(i + offset, t + 1) = 1;

a(i + offset, t + 3) = -2 \* x(i + 1);

a(i + offset, t + 4) = -1;

end

end

x = gauss(a, b);

c = zeros(n, 3);

for i = 1:n

if i == 1

c(i, 2) = x(1);

c(i, 3) = x(2);

else

t = (i - 1) \* 3 - 1;

for j = 1:3

c(i, j) = x(t + j);

end

end

end

function [v1, v2, neg] = take\_input(data\_start, data\_end)

while 1

neg = 1; % negating intergral in case endpoints are flipped

prompt1 = 'Insert v1: ';

v1 = input(prompt1);

prompt2 = 'Insert v2: ';

v2 = input(prompt2);

if v1 > v2

temp = v1;

v1 = v2;

v2 = temp;

neg = -1;

end

if valid\_range(v1, v2, data\_start, data\_end) == 1

break;

else

disp('Ops, inserted range is not within our data set, try again')

end

end

end

function [x] = thomas(g, f, e, r)

n = length(f);

for k = 2:n

e(k) = e(k)/f(k-1);

f(k) = f(k) - e(k)\*g(k-1);

end

% Fwd Subst.

for k = 2:n

r(k) = r(k) - e(k)\*r(k-1);

end

x(n) = r(n)/f(n);

for k = n-1:-1:1

x(k) = (r(k) - g(k)\*x(k+1))/f(k);

end

end

function [valid] = valid\_range(v1, v2, data\_start, data\_end)

if v1 >= data\_start && v2 <= data\_end

valid = 1;

else

valid = 0;

end

end