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

**Lab Report**

**Lab #9 Chapter 17: Polynomial Interpolation**

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

**EID: VL5649**

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

**Problem 1**

Bessel functions often arise in advanced engineering analyses such as the study of electric fields. Here are some selected values for the zero-order Bessel function of the first kind

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1.8 | 2.0 | 2.2 | 2.4 | 2.6 |
|  | 0.5815 | 0.5767 | 0.5560 | 0.5202 | 0.4708 |

Estimate using third and fourth-order interpolating polynomials with MATLAB’s built-in *polyfit* and *polyval*. Determine the percent relative error for each case based on the true value, which can be determined with *besselj*. Plot Bessel function of the first kind, and the polynomials you find for x=linspace(-2.6, 2.6, 50).

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

(1) For 5 data points, there is one and only one polynomial of fourth-order that passes through all the points. Why?

(2) To attain the best possible accuracy for the estimates, the point should be centered around as close as possible to the unknown. Why?

(3) Do higher order polynomials always give better interpolations?

(The following is your answer)

**MATLAB code:**

format long %makes it so you can tell small differences

x = [1.8;2.0;2.2;2.4;2.6]; %given data

y = [.5815;.5767;.5560;.5202;.4708];

a = polyfit(x(1:4),y(1:4),3); %3rd order polynomial fit

b = polyfit(x,y,4); %4th order polynomial fit

j3 = polyval(a,2.1); %calculates 3rd order at 2.1

j4 = polyval(b,2.1); %calculates 4th order at 2.1

bt = besselj(1,2.1); %calculates besselj at 2.1

error3 = abs(((bt-j3)/bt))\*100; %calculates errors

error4 = abs(((bt-j4)/bt))\*100; %calculates errors

xl = linspace(-2.6,2.6,50); %generates potential x's

p3 = polyval(a,xl); %3rd order polynomial

p4 = polyval(b,xl); %4th order polynomial

bj = besselj(1,xl); %besselj function

fprintf('The estimated Bessel value at 2.1 using a 3rd order polynomial is:\n%f\n',j3) %displays results

fprintf('The estimated Bessel value at 2.1 using a 4th order polynomial is:\n%f\n',j4)

fprintf('The true value of the Bessel function at 2.1 is:\n%f\n',bt)

fprintf('The percent relative error for a 3rd order interpolating polynomial is:\n%f%%\n',error3)

fprintf('The percent relative error for a 4th order interpolating polynomial is:\n%f%%\n',error4)

plot(xl,bj,xl,p3,xl,p4) %plot

legend('Bessel function','3rd Order Approximation','4th Order Approximation') %formatting

xlabel('x')

ylabel('J\_1(x) values')

title('Polynomial Approximations of Bessel function of First Kind')

**MATLAB function:**

The purpose of this function was to approximate the Bessel function of the first kind, given selected values from the Bessel function, using 3rd and 4th order polynomials. To do so, we could take advantage of MATLAB’s built in polyfit, polyval, and besselj functions, to reach the values we needed and calculate a percentage error to determine which order fit was better.

format long %makes it so you can tell small differences

This first line of code makes MATLAB format numbers so that they display more values so that it can be easier to distinguish very small differences in close numbers.

x = [1.8;2.0;2.2;2.4;2.6]; %given data

y = [.5815;.5767;.5560;.5202;.4708];

These 2 lines of code are the given data points of the Bessel function.

a = polyfit(x,y,3); %3rd order polynomial fit

b = polyfit(x,y,4); %4th order polynomial fit

These 2 lines of code fit a 3rd order and 4th order polynomial, respectively, to the given data.

j3 = polyval(a,2.1); %calculates 3rd order at 2.1

j4 = polyval(b,2.1); %calculates 4th order at 2.1

These 2 lines of code calculate the values of 2.1 using the 3rd and 4th order approximations of the Bessel function.

bt = besselj(1,2.1); %calculates besselj at 2.1

This line of code uses MATLAB’s built in Bessel J function to calculate the true value at x = 2.1.

error3 = abs((bt-j3/bt))\*100; %calculates errors

error4 = abs((bt-j4/bt))\*100; %calculates errors

These 2 lines of code calculate the percent relative errors in the 3rd order and 4th order polynomial approximations of the Bessel function.

xl = linspace(-2.6,2.6,50); %generates potential x's

This line of code generates potential x values to be plugged into our polynomial approximations as well as MATLAB’s besselj function to be graphed.

p3 = polyval(a,xl); %3rd order polynomial

p4 = polyval(b,xl); %4th order polynomial

bj = besselj(1,xl); %besselj function

These 3 lines of code calculate, using the values of x from the line before, values using the besselj function of MATLAB as well as our polynomial approximations.

fprintf('The estimated Bessel value at 2.1 using a 3rd order polynomial is:\n%f\n',j3) %displays results

fprintf('The estimated Bessel value at 2.1 using a 4th order polynomial is:\n%f\n',j4)

fprintf('The true value of the Bessel function at 2.1 is:\n%f\n',bt)

fprintf('The percent relative error for a 3rd order interpolating polynomial is:\n%f%%\n',error3)

fprintf('The percent relative error for a 4th order interpolating polynomial is:\n%f%%\n',error4)

These 5 lines of code format and print out our results to the command window.

plot(xl,bj,xl,p3,xl,p4) %plot

This line of code plots all of our values (true value of Bessel J function + 3rd and 4th order approximations) onto the same graph so that we can compare how the values line up.

legend('Bessel function','3rd Order Approximation','4th Order Approximation') %formatting

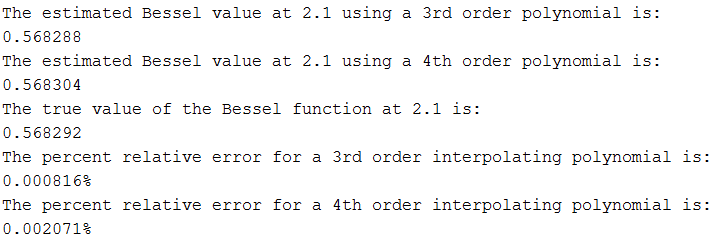
xlabel('x')

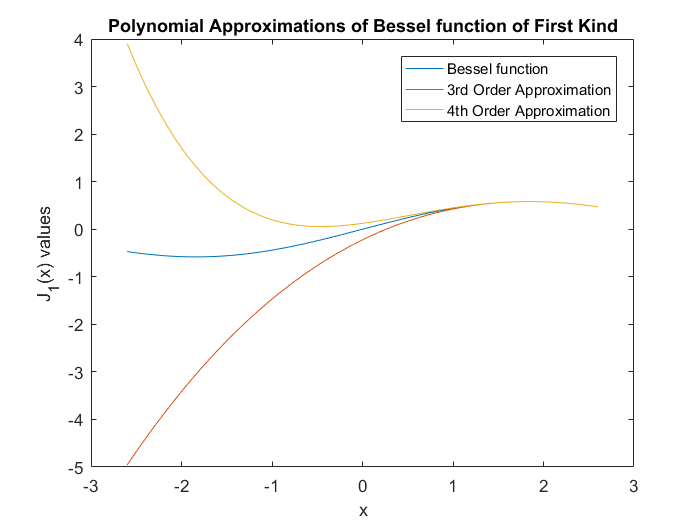
ylabel('J\_1(x) values')

title('Polynomial Approximations of Bessel function of First Kind')

These last 4 lines of code add labels to the plot so that it can be more easily interpreted.

**Results:**





**Discussion:**

As shown by the results, we were able to approximate the Bessel function (to some extent) by interpolating polynomials given a set of data points. Our data points were largely centered around the value that we were trying to calculate, resulting in a very good estimation regardless of what order to polynomial was (with the larger of the errors being ~.002%). This is because essentially, we ‘bracketed’ the value that we were trying to calculate for with closer values resulting in better estimates. For values that were not within our data set (all of the negative values on the graph), it can be seen that our polynomial estimates are very far off. For every n+1 data points, there is only one polynomial of order n—for polynomials of order greater than n, MATLAB uses the least squares fit to estimate a polynomial. This is because for every coefficient you need a data point, resulting in the maximum order being one value less than the number of data points. As illustrated by the percent relative errors between the 3rd and 4th order polynomial, a higher order does not necessarily guarantee a better value. This is likely because having a higher order than is necessary introduces oscillations that increase the error.

From this problem, we learned how to use MATLAB’s built in polyfit and polyval functions in order to approximate and then calculate a polynomial and a value within the polynomial. We also learned how to use MATLAB’s besselj function in order to calculate values for the Bessel function of first kind. Furthermore, we reviewed how to calculate errors and use linspace in order to generate a set of potential values. Lastly, we reviewed how to format and plot graphs.

**Problem 2 from textbook Problem 17.5**

Given the data

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 5 | 6 |
|  | 7 | 4 | 5.5 | 40 | 82 |

Calculate using Newton’s interpolating polynomials of order 1 through 4 with function *Newtint*. Choose your base points to attain good accuracy. That is, the points should be centered around and as close as possible to the unknown. Repeat the problem using the Lagrange polynomial of order 1 through 4 with function *Lagrange.*

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

(1) What do your results indicate regarding the order of the polynomial used to generate the data in the table?

(2) Which method could give you better accuracy in this problem?

(The following is your answer)

**MATLAB code:**

**Function:**

function yint = Lagrange\_VL(x,y,xx)

% Lagrange: Lagrange interpolating polynomial

% yint = Lagrange(x,y,xx): Uses an (n - 1)-order

% Lagrange interpolating polynomial based on n data points

% to determine a value of the dependent variable (yint) at

% a given value of the independent variable, xx.

% input:

% x = independent variable

% y = dependent variable

% xx = value of independent variable at which the

% interpolation is calculated

% output:

% yint = interpolated value of dependent variable

n = length(x); %number of terms

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

s = 0; %initializes sums

for i = 1:n

product = y(i); %loads current y

for j = 1:n

if i ~= j

product = product\*(xx-x(j))/(x(i)-x(j)); %lagrange interpolations

end

end

s = s+product; %sums interpolations

end

yint = s; %outputs results

**Function:**

function yint = Newtint\_VL(x,y,xx)

% Newtint: Newton interpolating polynomial

% yint = Newtint(x,y,xx): Uses an (n - 1)-order Newton

% interpolating polynomial based on n data points (x, y)

% to determine a value of the dependent variable (yint)

% at a given value of the independent variable, xx.

% input:

% x = independent variable

% y = dependent variable

% xx = value of independent variable at which

% interpolation is calculated

% output:

% yint = interpolated value of dependent variable

% compute the finite divided differences in the form of a

% difference table

n = length(x); %number of terms

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

b = zeros(n,n); %initializes matrix of size n

% assign dependent variables to the first column of b.

b(:,1) = y(:); % the (:) ensures that y is a column vector.

for j = 2:n

for i = 1:n-j+1

b(i,j) = (b(i+1,j-1)-b(i,j-1))/(x(i+j-1)-x(i)); %finite divided differences

end

end

% use the finite divided differences to interpolate

xt = 1; %initializes xt for first value in loop

yint = b(1,1); %b\_1 = f(x\_1)

for j = 1:n-1

xt = xt\*(xx-x(j)); %interpolation

yint = yint+b(1,j+1)\*xt;

end

**Mainscript:**

for i = 1:4

x = [3;5;2;6;1]; %rearranged data

y = [5.5;40;4;82;7];

a(i) = Newtint\_VL(x(1:i+1),y(1:i+1),4); %calls newtint

b(i) = Lagrange\_VL(x(1:i+1),y(1:i+1),4); %calls lagrange

end

fprintf('For both Newton and Lagrange, the values of order 1-4 are:\n') %output results

fprintf('1st\t\t2nd\t\t3rd\t\t4th\n')

fprintf('%1.2f\t',a)

fprintf('\n')

**MATLAB function:**

The purpose of this function was to compare the efficiency of the Newton Interpolating Method and Lagrange methods in reaching an unknown values, as the order of the polynomial was increased. To do this, we had to call the Newtint and Lagrange functions using n+1 data points to get a polynomial of order n. We could then take these polynomials and solve for the unknown. These values could then be compared to determine which value reached the unknown, faster.

function yint = Lagrange\_VL(x,y,xx)

This first line of code specifies the outputs and inputs of the Lagrange function, with x and y being the data given and xx being the value that we wanted to calculate.

n = length(x); %number of terms

This line of code calculates the number of terms that we are working with—this is important because the order of the polynomial is one less than this value

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

This line of code creates an error when x and y aren’t the same value because you need for every value of x to have a corresponding value of y.

s = 0; %initializes sums

This line initializes the ‘sums’ of the interpolations so that it starts at 0 there are no issues when re-running the function.

for i = 1:n

product = y(i); %loads current y

These 2 lines of code from the outer loop portion of our loops and loads a value of y to use for the calculations in the inner loop.

for j = 1:n

if i ~= j

product = product\*(xx-x(j))/(x(i)-x(j)); %lagrange interpolations

These 3 lines iterate the lagrange interpolation calculations for all of the values of x, in estimating our desired value

end

end

These 2 lines of code close the inner loop.

s = s+product; %sums interpolations

This line of code sums up the interpolations from the inner loop as part of the outer loop

end

This line of code closes the outer loop.

yint = s; %outputs results

This line of code outputs the result after the calculations are done.

function yint = Newtint\_VL(x,y,xx)

Similar to the first line of the Lagrange function, this first line of the Newtint function specifies the same inputs and output.

n = length(x); %number of terms

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

Similar to the Lagrange function, the newtint function also calculates the number of terms to be used in calculations and outputs an error if the terms do not line up.

b = zeros(n,n); %initializes matrix of size n

This line of code initializes a matrix of 0’s of size ‘n’ (the number of terms). By initializing this matrix, MATLAB saves time by not having to constantly update the size of the matrix and by setting the values to 0, unused values are ‘0’ and do not affect the calculations.

b(:,1) = y(:); % the (:) ensures that y is a column vector.

for j = 2:n

for i = 1:n-j+1

b(i,j) = (b(i+1,j-1)-b(i,j-1))/(x(i+j-1)-x(i)); %finite divided differences

These 4 lines of code put y as the first column in our b matrix and then calculate the finite divided differences used in the later calculations.

end

end

These 2 lines of code close the inner and outer loops used to calculate the finite divided differences

xt = 1; %initializes xt for first value in loop

This line of code initializes our ‘xt’ value to 1 so that we can interpolate our finite divided differences without having to rewrite the code for just the first loop.

yint = b(1,1); %b\_1 = f(x\_1)

This line of code sets our ‘yint’ (our answer) as b(1,1) which is the same as b\_1 or f(x\_1) from our calculations in the newton method. This is also the first term when calculating our interpolations.

for j = 1:n-1

xt = xt\*(xx-x(j)); %interpolation

yint = yint+b(1,j+1)\*xt;

end

These 4 lines of the newtint function form the section that calculates the interpolations, term by term. Each of the terms is then summed up in our ‘yint’ value, which, after all the calculations are over, is the output of the function (the result).

for i = 1:4

This first line of the main script creates a for loop because we want to calculate the 1st-4th order approximations of a value.

x = [3;5;2;6;1]; %rearranged data

y = [5.5;40;4;82;7];

These 2 lines of code are the x and y values given to us as part of the problem. They have been rearranged so that we can use the for loop to work with a specific subset of the values.

a(i) = Newtint\_VL(x(1:i+1),y(1:i+1),4); %calls newtint

b(i) = Lagrange\_VL(x(1:i+1),y(1:i+1),4); %calls lagrange

These 2 lines of code call the Newtint and Lagrange functions, respectively. Because of the way the for loop is written, these calculations are always performed with x and y values that are centered around the value that we are trying to calculate. The results, in order of increasing order, are then stored as the variables a and b.

end

This line of code closes the for loop that we used to iterate the newton and lagrange methods.

fprintf('For both Newton and Lagrange, the values of order 1-4 are:\n') %output results

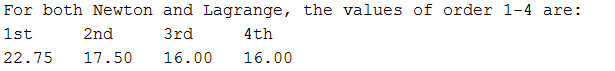
fprintf('1st\t\t2nd\t\t3rd\t\t4th\n')

fprintf('%1.2f\t',a)

fprintf('\n')

These last 4 lines of code print out the results of our calculations into the command window.

**Results:**



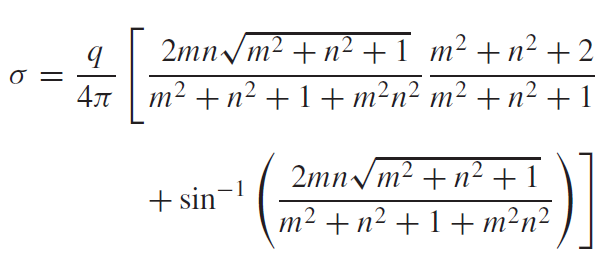
**Discussion:**

As shown by the results, both the Newton Interpolation and Lagrange Interpolation functions are equally efficient at reaching our desired value, in this scenario, with both functions having the same values for every order estimation. As the order of the polynomial increased, the general trend for both functions is for the approximated values to approach the true value. This can be seen as our 1st order polynomial produced a value that was 6.75 off (42% relative error) but our 2nd order polynomial was able to bring that down to 1.5 (9.4% relative error). In this problem, neither method gives better accuracy as both methods produce the same values at the same orders; however, it has been shown that Newton’s method is approximately 1.174574 times better than Lagrange’s (Srivastava 2012).

From this problem, we learned how to apply the Newton and Lagrange Interpolation methods in order to calculate a value given a set of data. We reviewed how to perform column and row specific matrix operations in MATLAB as well as basic matrix operations. Additionally, we reviewed how to user inner and outer loops in order to iterate specific operations. Lastly, we reviewed how to format and display results in the command window using MATLAB’s fprintf function.

**Problem 3 from textbook Problem 17.16**

The vertical stress σz under the corner of a rectangular area subjected to a uniform load of intensity q is given by the solution of Boussinesq’s equation:



Because this equation is inconvenient to solve manually, it has been reformulated as



where fz(m, n) is called the influence value, and m and n are dimensionless ratios, with **m = a/z** and **n = b/z** and **a** and **b** are defined in Fig. 1. The influence value is then tabulated, a portion of which is given in Table 1. If a = 5.6 and b = 12, use a third-order interpolating polynomial to compute σz at a depth 10 m below the corner of a rectangular footing that is subject to a total load of 100 t (metric tons). Express your answer in tonnes per square meter. What is the vertical stress when (a, b) = (4.8, 14) or (4.2, 16) (the rest of parameters are fixed)? Note that q is equal to the load per area.

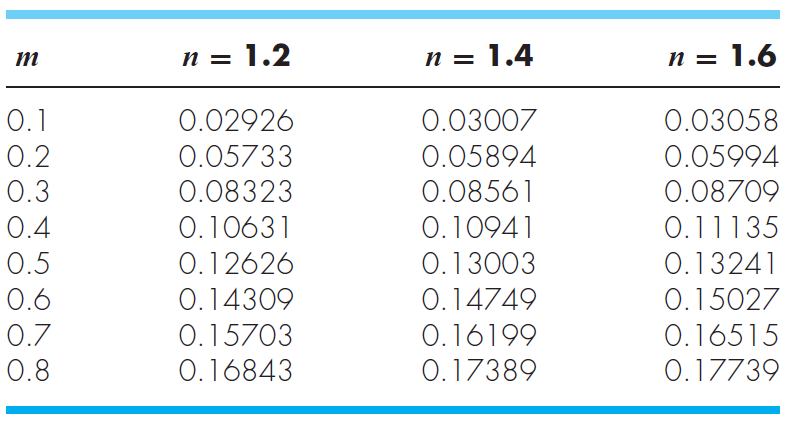
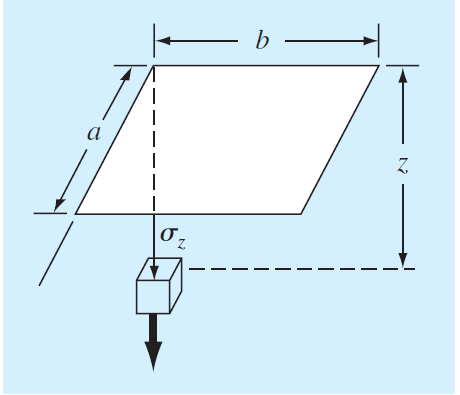


Table 1.

Fig. 1

**Things to discuss**

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

(1) Discuss the relationship between aspect ratio of rectangular plate and vertical stress.

(The following is your answer)

**MATLAB code:**

m = [.1;.2;.3;.4;.5;.6;.7;.8]; %data

n12 = [.02926;.05733;.08323;.10631;.12626;.14309;.15703;.16843];

n14 = [.03007;.05894;.08561;.10941;.13003;.14749;.16199;.17389];

n16 = [.03058;.05994;.08709;.11135;.13241;.15027;.16515;.17739];

p12 = polyfit(m,n12,3); %fitting curves

p14 = polyfit(m,n14,3);

p16 = polyfit(m,n16,3);

f12 = polyval(p12,.56); %solving function values using fitted curves

f14 = polyval(p14,.48);

f16 = polyval(p16,.42);

s12 = f12 \* 100/(5.6\*12); %calculating sigmas

s14 = f14 \* 100/(4.8\*14);

s16 = f16 \* 100/(4.2\*16);

ratio = [5.6/12;4.8/14;4.2/16]; %compiles ratios

sigma = [s12;s14;s16]; %compiles sigmas

xval = 0:.01:1; %potential aspect ratios

p = polyfit(ratio,sigma,2); %polynomial relating aspect ratio to stress

y = polyval(p,xval); %potential vertical stersses

plot(xval,y,ratio,sigma,'o') %plot

legend('Potential Values','Calculated Values')

xlabel('Aspect Ratio') %formatting

ylabel('Sigma (tonnes/m^2)')

title('Vertical Stress as a Function of Aspect Ratio')

fprintf('If a = 5.6 and b = 12, the vertical stress is %f at a depth 10m\n',s12)

fprintf('below the corner of a rectangular footing that is subjected to 100 tons.\n')

fprintf('When (a,b) = (4.8,14) or (4.2,16), the vertical stress is:\n')

fprintf('%f and %f, respectively (with fixed parameters).\n',s14,s16)

**MATLAB function:**

The purpose of this function was to calculate vertical stress given that it is a function of m and n multiplied by the distributed load. To calculate the function of m and n, we had to interpolate a polynomial from a given table of values relating m and n. Using this value, we could then calculate specific vertical stresses given that the other parameters were held constant (and then go one step further and relate vertical stresses to aspect ratios).

m = [.1;.2;.3;.4;.5;.6;.7;.8]; %data

n12 = [.02926;.05733;.08323;.10631;.12626;.14309;.15703;.16843];

n14 = [.03007;.05894;.08561;.10941;.13003;.14749;.16199;.17389];

n16 = [.03058;.05994;.08709;.11135;.13241;.15027;.16515;.17739];

These first 4 lines of code are the columns of the table given to us formatted into individual vectors.

p12 = polyfit(m,n12,3); %fitting curves

p14 = polyfit(m,n14,3);

p16 = polyfit(m,n16,3);

These 3 lines of code take the columns from before and solve for polynomials relating m to n = 1.2, n = 1.4, and n = 1.6, respectively.

f12 = polyval(p12,.56); %solving function values using fitted curves

f14 = polyval(p14,.48);

f16 = polyval(p16,.42);

These 3 lines of code take the polynomials that we solved for before and solve for specific values (m = .56 for n = 1.2, m = .48 for n = 1.4, and m = .42 for n = 1.6). These values correspond to the ratios of a to b that we want to solve vertical stress for, as part of the problem.

s12 = f12 \* 100/(5.6\*12); %calculating sigmas

s14 = f14 \* 100/(4.8\*14);

s16 = f16 \* 100/(4.2\*16);

These 3 lines of code take the function value relating m and n and solve for the vertical stress, sigma at that point. This is obtained by multiplying the function value by the distributed load (load/area).

ratio = [5.6/12;4.8/14;4.2/16]; %compiles ratios

sigma = [s12;s14;s16]; %compiles sigmas

These 2 lines of code compile the sigma values and their corresponding aspect ratios into 2 separate column vectors so that we can perform a regression on them.

xval = 0:.01:1; %potential aspect ratios

This line of code produces potential ‘x values’ or ‘aspect ratios’ that the vertical stresses will correspond to.

p = polyfit(ratio,sigma,2); %polynomial relating aspect ratio to stress

This line of code performs a 2nd order polynomial fit to the data, relating the aspect ratio to the the vertical stress.

y = polyval(p,xval); %potential vertical stersses

This line of code takes our potential aspect ratios vector that we made earlier and plugs it into the polynomial function that we just solved for in the line before, generating corresponding vertical stresses.

plot(xval,y,ratio,sigma,'o') %plot

This line of code plots the 3 stresses that we calculated vertical stresses to, and plots a function for the regression performed on those 3 points.

legend('Potential Values','Calculated Values')

xlabel('Aspect Ratio') %formatting

ylabel('Sigma (tonnes/m^2)')

title('Vertical Stress as a Function of Aspect Ratio')

These 4 lines of code add labels to the plot so that it can be more easily understood.

fprintf('If a = 5.6 and b = 12, the vertical stress is %f at a depth 10m\n',s12)

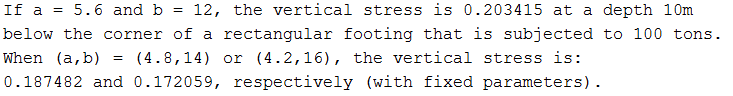
fprintf('below the corner of a rectangular footing that is subjected to 100 tons.\n')

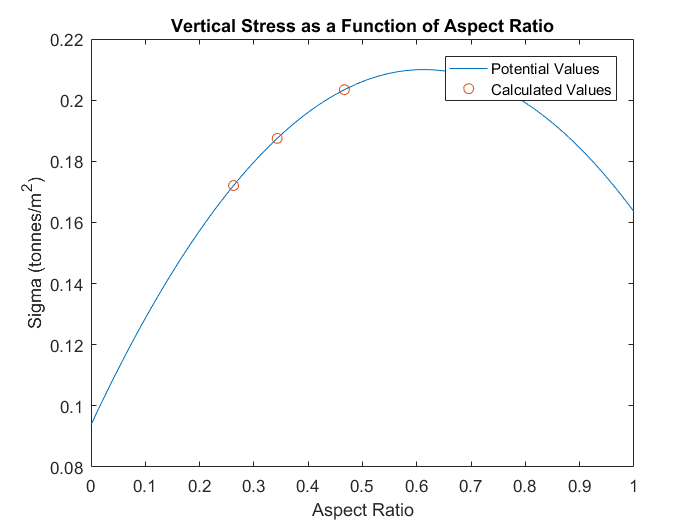
fprintf('When (a,b) = (4.8,14) or (4.2,16), the vertical stress is:\n')

fprintf('%f and %f, respectively (with fixed parameters).\n',s14,s16)

These last 4 lines of code print out or results for vertical stresses into the command window.

**Results:**





**Discussion:**

As shown by the results, the vertical stresses when (a,b) = (5.6,12), (4.8,14), and (4.2,16) are .203415, .187482, and .172059, respectively. By performing a regression on the regressed values of the table, we get a function that relates the vertical stresses to the aspect ratio of the rectangle. As the aspect ratio increases (a/b gets larger), the vertical stress generally increases until around .6-.7, when the vertical stress begins to go back down. It is possible to do all of this because performing interpolation assuming that n is a fixed value. We can calculate for values (a,b) = (5.6,12), etc. despite not having a model for n = 12 because f(m,n) is a ratio (i.e. a = m/z, b = n/z, a/b = m/n so (5.6,12) = (.56,1.2) as long as you keep the ratios). We can then calculate a regression from sigma values we calculated to find the relation between sigma and the aspect ratio.

From this problem, we reviewed how to use MATLAB’s built in polyfit and polyval functions in order to fit a polynomial to a set of data and then calculate a specific value for that polynomial. We reviewed how to work with basic arrays and basic array manipulation in MATLAB. We also reviewed how to plot sets of data against each other using the plot function and to label it. Lastly, we reviewed how to format and output results into the command window using the fprintf function.