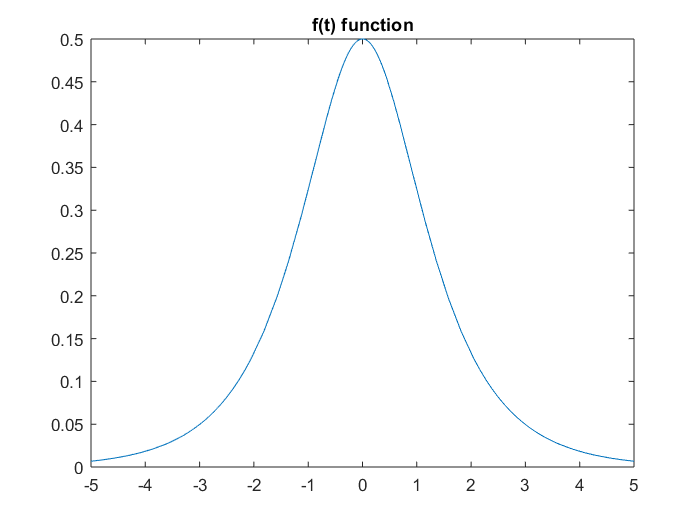
**ECSE 443 – Final Project**

**Question 1**

Please refer to Appendix A, for the Matlab code. The given equation to be evaluates using Simpson’s rule is the following:

The inner function , can be described by this graph.

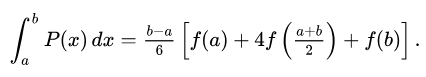


The graph demonstrates that the function, f(x), saturates at approximately 6. Therefore, the values after 5 are approximately 0, which means that from the range , . This is proven by using the Matlab integral function.

|  |  |
| --- | --- |
| **Range** | **I** |
|  | 0.785398163397448 |
|  | 0.785398163397454 |
|  | 0.782919416297423 |

So, the integral with the range can be adjusted to be approximately . However, to ensure a higher accuracy we’ll take slightly above such value by using range . The integral to be evaluated can be modified to be:

The method to calculate Simpson’s rule was derived for assignment 4. The following formula was followed to calculate he area under the curve at each segment using the Simpsons rule:

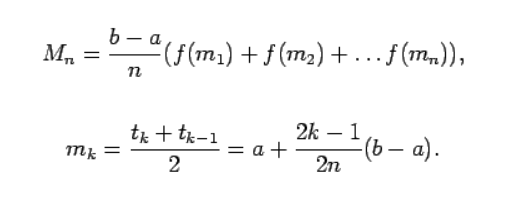


|  |  |
| --- | --- |
| **Number of Segments (Midpoint)** | 19 |
| **I** | 0.785309462232758 |
| **Absolute Error** | 8.870116469052163e-05 |

**Question 2**

Please refer to Appendix B, for the Matlab code. The given equation to be evaluates using the midpoint rule is the following:

The requirement was to use a step size h = 0.01. The step size is calculated with , where a and b are the boundary conditions and N is the number of segments that will be used. Knowing that and , then . Following the method to calculate the midpoint rule which was derived in assignment 4, the area under the curve at each segment can be found with this formula:



|  |  |
| --- | --- |
| **Number of Segments (Midpoint)** | 100 |
| **I** | 0.623775746323754 |
| **Relative Error** | 0.064336380514477 |

**Question 3**

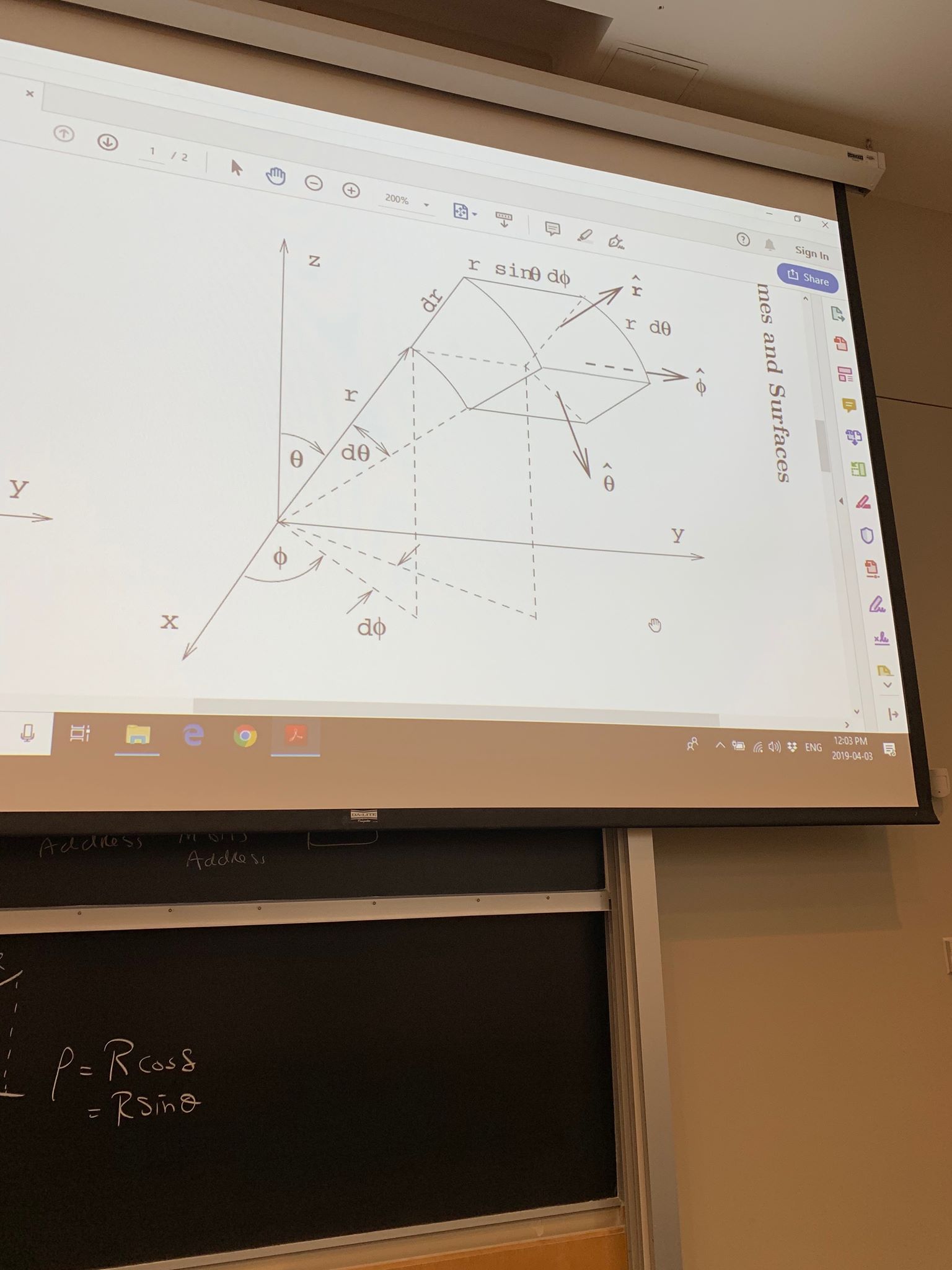
Refer to Appendix C for the corresponding Matlab code. We want to verify Gauss’ Law with numerical integration and Gauss’ Law refers to a set equation with an integral on each side.

Taking a look first at the left side. It is a simple integral to calculate considering is a straight line of infinite charge. Since, the line charge is infinite it is not used to consider the boundary of the conditions and we instead use the sphere’s limits. This is due to the fact that the portion of the infinite line charge outside of the sphere has a net electrical flux of 0 and can be therefore ignored. So, the left equation can be simplified to the following.

To calculate a step size of 0.01 we need to set the number of segments to 200 which equals the step size since a = 0 and b = 2. Simpson’s method was used because it generally used less segments than the other methods, we learnt which meant that it converged faster so it was more precise.

|  |  |
| --- | --- |
| **Step Size** | 0.01 |
| **Number of Segments** | 200 |
| **Integral Value** | 55.200000000000003 μC |

Now taking a look at the right side of the integral. We have a sphere encapsulating the line charge at certain points along the line. We did the calculation in the spherical coordinate system, which was shown in class. Using the spherical coordinated we can set the bounds corresponding to this coordinate system.



So, we can breakdown the integral into the spherical coordinates first.

We can calculate it using Simpson’s method using both integrals to get this. The bounds were different in this case, so a different number of segments had to be used for each part. Simpson’s method was repeated until there was a relative error of 10% (meaning the value of the right side differed by 10% of the value on the left).

|  |  |
| --- | --- |
| **Step Size** | 0.01 |
| **Number of Segments** | 2 |
| **Integral Value** | 55.325849237570026 μC |

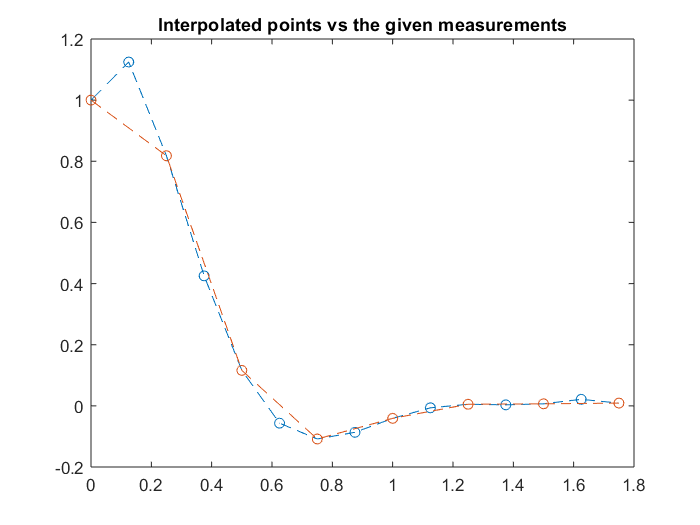
At this point you have the integral form to be solved but it can be simplified more to eliminate the second integral.

But the results would have been the exact same whichever method was used. When comparing both sides of the equation now we find that the results are very similar and the absolute error low. This clearly demonstrates Gauss’ law.

|  |  |
| --- | --- |
| **Value S\_left** | 55.200000000000003 μC |
| **Value S\_right** | 55.325849237570026 μC |
| **Absolute Error** | 0.125849237570023 |

**Question 4**

Refer to the Appendix D for the corresponding Matlab code and the Lagrange polynomials function that was used. The goal is to find the coefficients of the 2nd order differential function corresponding to the given measurements that were performed. Due to the limited number of points, I interpolated more points in the function using Lagrange’s method which we had written in Assignment 3. I interpolated points for the domain of x ϵ [0,1.75] with a step size of 0.125. Therefore, it gave a total of 15 points and the following reflects the points.

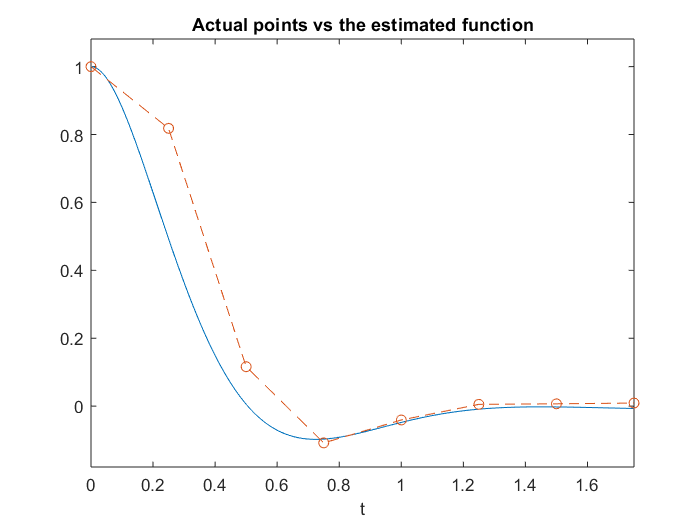


In assignment 4, we learnt about central finite difference as a method for finding the values of derivatives. So, we can use this knowledge of central difference and apply it to find the derivative values from the points with 8th order error because using a higher order allows for less error. The functions that were used are the following:

The central difference was calculated using these functions and were put into the above matrix in order to find the coefficients of the given points. Since, there were 15 points available and only 9 points were needed as a time there 7 possible starting points for the central difference. The coefficients were found to be:

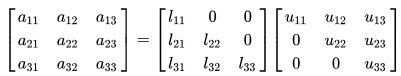
|  |  |
| --- | --- |
| **A** | 3.3586704235173 |
| **B** | 22.5951996658204 |
| **C** | 101.0425447721065 |

These coefficients give forth the following plot:

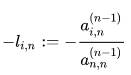


**Question 5**

Refer to Appendix E for the Matlab code corresponding to the LU factorization code. The LU factorization was solved in order to find the upper and lower triangle matrices corresponding to A.



The matrices were solved by looking for the row with the highest values per column and moving it to the diagonal. This was done using row reduction methods which eliminated the values below the diagonal for the upper, but the same steps were done on the identify matrix to obtain the lower triangle. This was used to eliminate matrix elements.



After performing the row echelon reduction on the A matrix, U then, the lower matrix has the L operations to it and U has the reduced A. The L and U were found to be:

The unknowns were found by these equations and gave the following results and the square error used the following equation.

|  |  |
| --- | --- |
| **X1** | -1.077179830921188 |
| **X2** | 1.990205466334715 |
| **X3** | 1.474706574375536 |
| **X4** | -1.906432108560652 |
| **Squared Error** | 2.641697956358863e-28 |

We can see that there is remotely no error using this method and that the error is negligible.

**Question 6**

Refer to Appendix F, for the Matlab code corresponding to fixed point iteration function. The given matrix corresponds to 4 systems of equations with 4 unknown values. The fixed point iteration method iterates through the system of equations and the unknowns will converge to the answer after a number of iterations. The following form was used to solve the equations:

where , and fixed points are g

The issues with fixed point iteration are that the function does not always converge. Therefore, it is important to ensure that the right functions are formed in order for the unknowns to converge to the result. The convergence condition is:

Therefore, the given system of equations had some issues with convergence.

Since the system was not converging the functions needed to be adjusted in order to change which variable was being solved for in each equation. By simply switching the 1st and 3rd row and the 2nd and 4th row, the equations meet the convergence condition. The swapping of equations ensures switches which variables are being solved the system of equations.

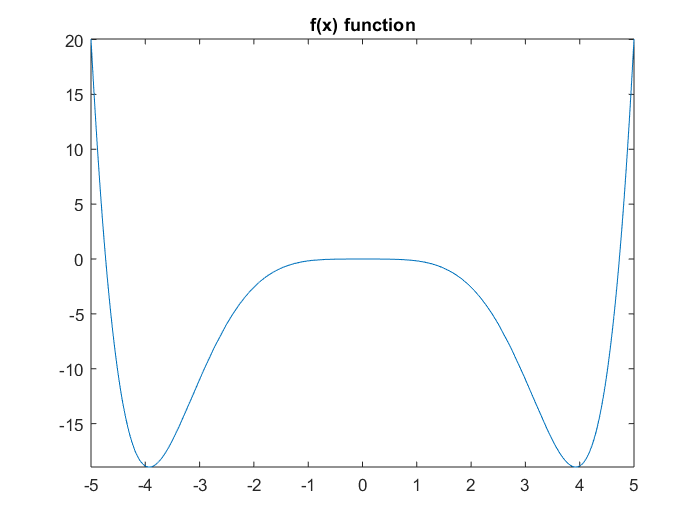
Using 100 iterations the system of equations converges to with the squared error calculated by the function set in 5:

|  |  |
| --- | --- |
| **x1** | -1.077080253093629 |
| **x2** | 1.990142287156950 |
| **x3** | 1.474621130739856 |
| **x4** | -1.906281024359803 |
| **Squared Error** | 4.403440286156683e-08 |

We can observe that the error for fixed point iteration has 20 magnitudes of error than the LU factorization method. So we can conclude that the LU method is the ideal method to use in terms of reducing error.

**Question 7**

Refer to **Appendix G**, for the function to calculate the roots using the secant method. The points given represent this function, as seen below.



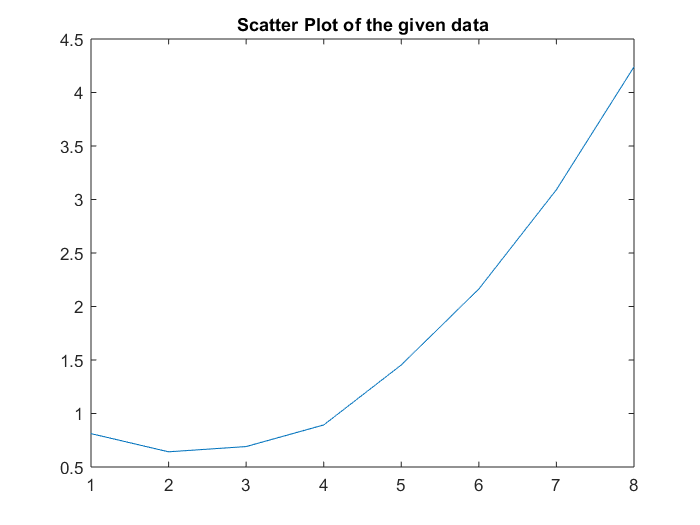
Similarly, to the assignment 2, I look for the places in between points where there is a possible root by checking if the sign changes. I took step sizes of h=0.5 starting from the first root at x=0 and moved throughout the function at by the step size. Then I go through the multiple iterations using the following function and the relative error measured in precision as a limit to find the root.

Using the above function, we calculate the roots from until we reach the targeted threshold for error of 10-4. Since the function is even and mirrored on the x-axis the next zero is to the right and the left at the same distance apart but just with a negative.

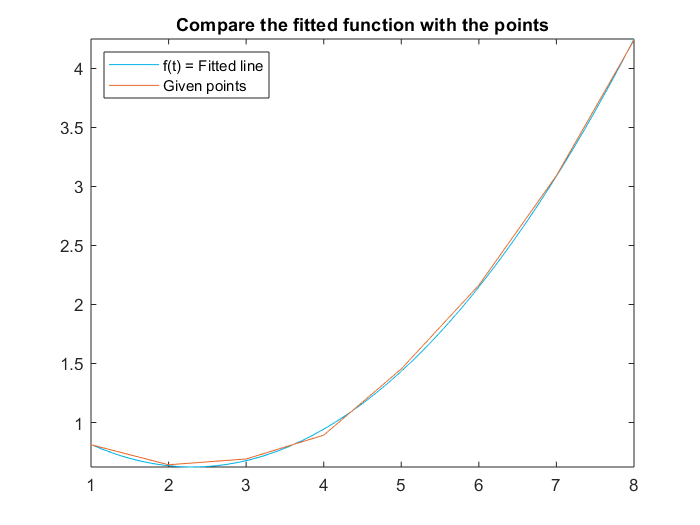
|  |  |
| --- | --- |
| **Relative Error** | 1.786830646479475e-09 |
| **Root (positive x-axis)** | 4.730040736410922 |
| **Root (negative x-axis)** | - 4.730040736410922 |

**Question 8**

Refer to the Appendix H, for the corresponding Matlab code. The points provided to us in the assignment correspond to the following plot.



The graph looks similar to a second degree polynomial; therefore, I used the a polynomial fit using the normal equations method that was used in assignment 2, question 1 to fit a second order polynomial to a set of data points. Resulting in the combined plot:

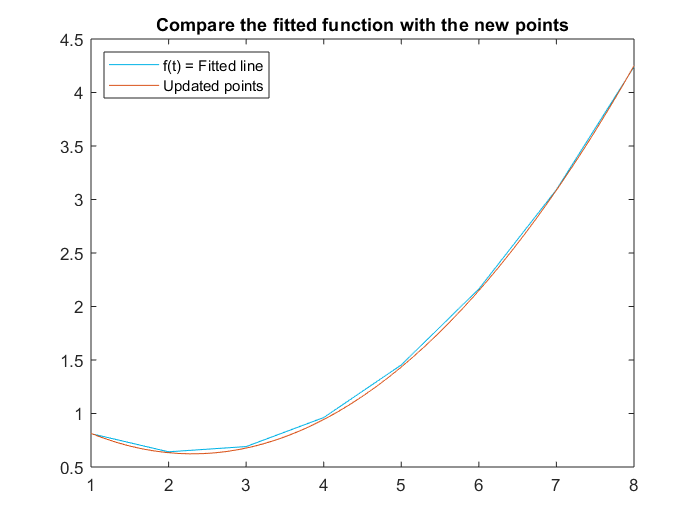


In order to find a point that maybe wrong and out of place, I found the absolute error at each point on the graph. Using the Lagrange Polynomial Interpolation that we used in assignment 3, we can use the similar method to calculate and fix the wrong point. We can find the new point using the interpolation.

f\_l = C:\Users\bjay2\AppData\Local\Temp\ConnectorClipboard6739360047510106805\image15542276322750.png

|  |  |
| --- | --- |
| **Max Absolute Error** | 0.050642857142857 |
| **Index** | 4 |
| **New point y** | 0.962257142857143 |

The new curve against the fitted function previously found it below and is closer.



**Question 9**

The given equation has the second order derivative, which means that the functions must be transformed so that numerical calculations can be done on it.

We create a matrix containing all the y and z values that will be calculated and we use the initial conditions as starting points for those values. Therefore, we solve both equations using both Runge-Kutta and Euler’s method.

Refer to Appendix I, for the corresponding Matlab code. In order to calculate the Runge-Kutta to the 4th order, we needed to consider finding 4 coefficients for every iteration of the function. Therefore, the functions were calculated to be:

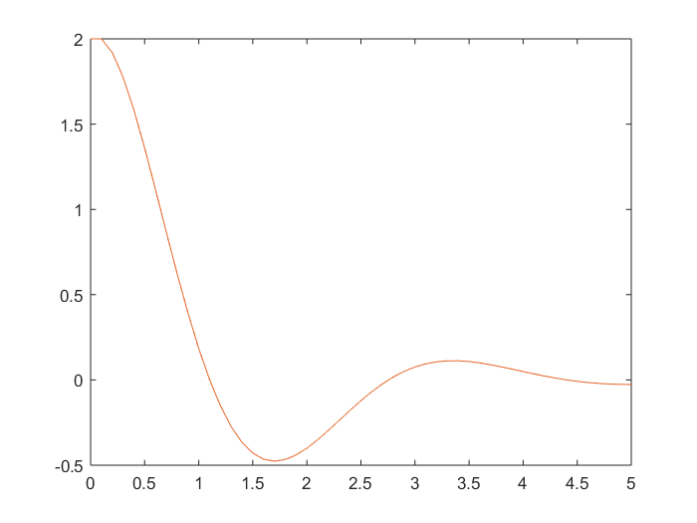
After finding all the coefficients from those functions based on the Taylor series, the coefficients are summed together to calculate the next value of y in the function based off the initial conditions.

After this is done for each y-coordinate it produces the following:



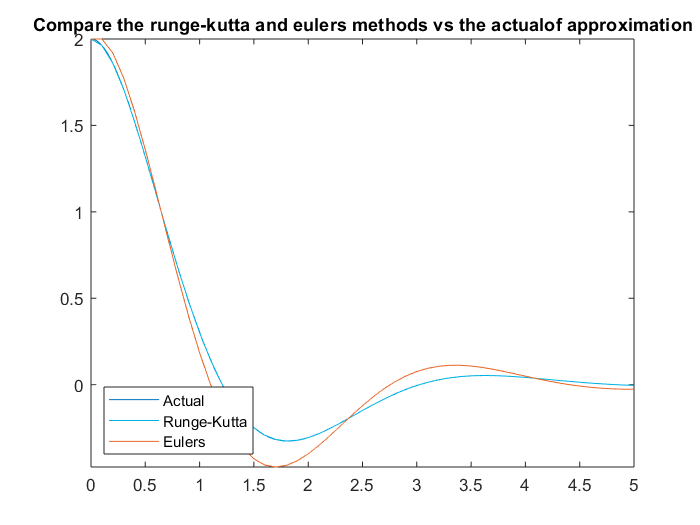
Now to calculate the plot using Euler’s method we use the same functions derived at the beginning named f and g. Additionally, we create the same matrices filled with the initial conditions for the first point. The equations associated with Euler’s method to calculate the points are much shorter compared to the ones of Runge-Kutta.

After this is done for each y-coordinate it produces the following:



The plots were plotted together, and they differ slightly after x = 2.5. Runge Kutta method provides a approximations because it is much higher order than Euler’s method. The RK method a 4th order method compared to Euler’s which is only a 1st order one. Therefore, the graphs differ and it is clear that RK is a better approximation than Euler. Additionally, the average absolute error if higher for Euler’s method compared to Runge Kutta.

|  |  |
| --- | --- |
| **Method of Approximation** | **Average Absolute Error** |
| Euler’s | 0.063496337297686 |
| Runge Kutta | 7.562639331815784e-06 |

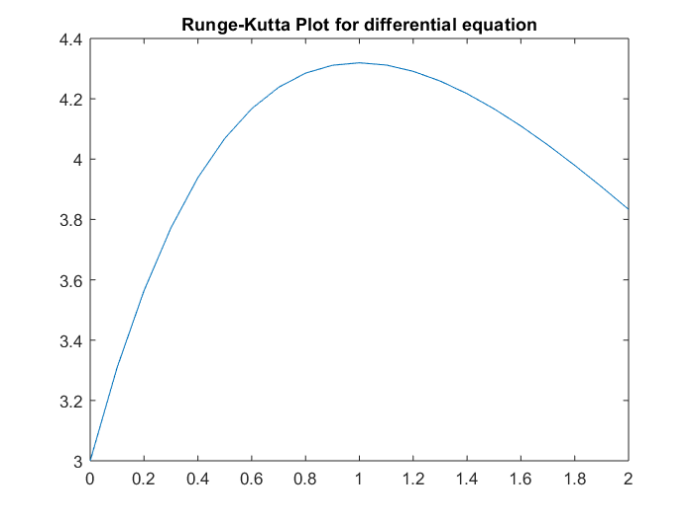


**Question 10**

Refer to Appendix J, for the corresponding Matlab code. In order to calculate the Runge-Kutta to the 2nd order, we needed to consider finding 2 coefficients for every iteration of the function. Therefore, the functions with the Taylor series function were calculated to be:

After finding all the coefficients from those functions based on the Taylor series, the coefficients are summed together to calculate the next value of y in the function based off the initial condition.

With the given step value and initial conditions, the function can be solved and graphed.

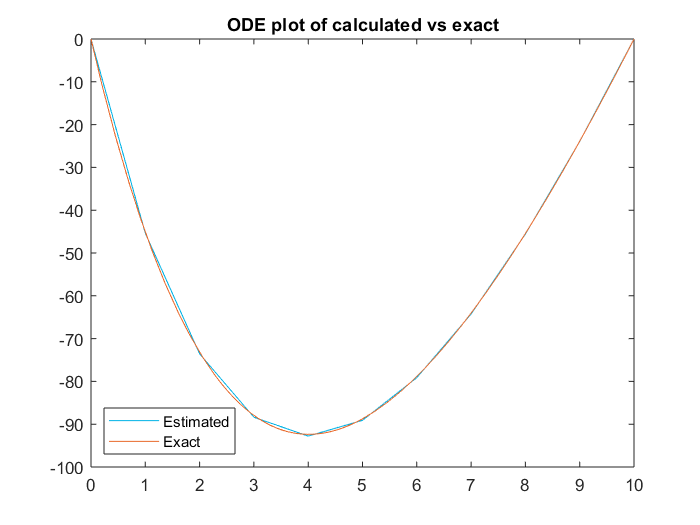


**Question 11**

Refer to Appendix K, for the corresponding Matlab code. Use the given values of delta x and initial points to set the boundary conditions and set the unknowns list. We know based on the differential equation that we can create the following functions.

With these functions we can substitute them into the original function and isolate .

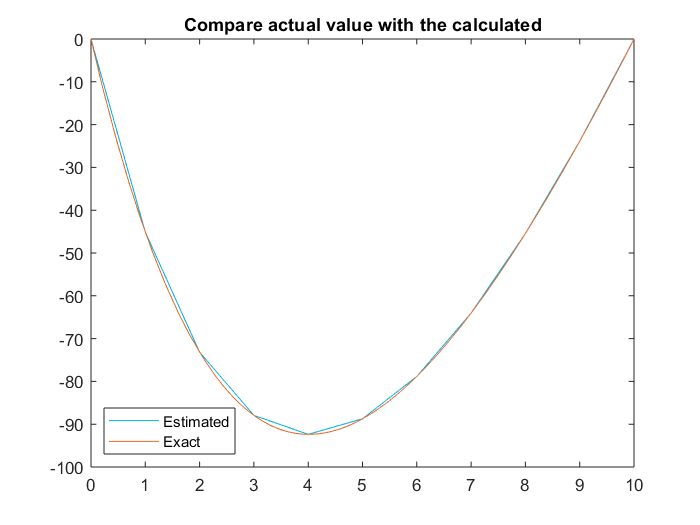
Using this function, we can iterate through the number of coordinates using the boundary conditions that were provided. The function can be plotted vs the actual function and it can be seen that the estimated function provides a close estimate of the real function.



**Question 12**

Refer to Appendix L, for the corresponding Matlab code. The shooting function is used in this occasion similar to what was demonstrated in the provided notes. In the first step we set the ODE to a system of 2 equations. Similarly to question 9 we use the same Runge Kutta method to calculate the ideal function.

Next, we need to define the shooting function that will approximate the points by trying several different points. The given initial conditions set the coordinates that are used to tested. The methodology chosen to find the ideal point initial condition for z was to start at point 0 and point 1 as the bounds and increase by a step count of 5 at each iteration until there is a sign change at the boundary condition we are testing against. In this case we have y(10) = 0 therefore, we only needed to detect a sign change then.



The results are relatively close, to the generated matlab built in Runge Kutta method that was used to compare. Additionally, Runge Kutta method is a better method of approximations which is why it was chosen to work with the shooting method.

**Appendix**

**Appendix A – Question 1 Matlab Code**

format long

syms f(t)

% the given function and the decorator

f(t) = 1./(exp(t)+exp(-t));

p = @(t) 1./(exp(t)+exp(-t));

% the threshold for error

thresh = 0.0001;

% plot the function to see where and if it saturates

fplot(f(t))

title('f(t) function')

% find the actual answer for the function going to infinity

actual = integral(p,0,Inf)

actual =

0.785398163397448

integral(p,0,6)

ans =

0.782919416297423

integral(p,0,20)

ans =

0.785398163397454

% since the function saturates at [6,inf) we can find finite boundaries

a = 0;

b = 20;

% the number of segments/points that will be used , assume >1

N\_s = 1; % actual 47

% set S = 0

S = 0;

% check if the absolute errorv is larger than the threshold

while(abs(S-actual) > thresh)

% reset the sum of of area segments under the curve

S = 0;

% find the step size

dx = (b-a)/N\_s;

for i=1:N\_s

% calculate the integral using the simpsons method

x = a + (i-1)\*dx;

y = (dx/6)\*(f(x) + 4\*f(x+dx/2) + f(x + dx));

S = S + y;

end

% increment the number of segments

N\_s = N\_s + 1;

end

N\_s = N\_s - 1

N\_s =

19

absoluteError = double(abs(S-actual))

absoluteError =

8.870116469052163e-05

double(S)

ans =

0.785309462232758

**Appendix B – Question 2 Matlab Code**

format long

syms f(t)

% the given function and the decorator

f(t) = t.^3./(1-t.^2).^(1/2);

p = @(t) t.^3./(1-t.^2).^(1/2);

% step size

h = 0.01;

% set the bounds to the given points

a = 0;

b = 1;

% plot the function

fplot(f(t))

title('f(t) function')

% find the actual answer for the function going to from the bounds

actual = integral(p,0,1)

actual =

0.666666666666744

actual1 = int(f,0,1)

actual1 =

C:\Users\bjay2\AppData\Local\Temp\ConnectorClipboard1362037288450451872\image15550260699711.png

% to have h = 0.01, that means N\_m = 100

h = 0.01;

% so the number of segments is equal

N\_m = 100;

% set the sum of areas to 0

M = 0;

% do the one run for the set values

% find the step size which should = 0.01

dx = (b-a)/N\_m;

for i=1:N\_m

% calculate the midpoint formula at each segment and sum the area

% under the curve

x = a + 0.5\*(2\*i-1)\*dx;

M = M + f(x)\*dx;

end

% display the error and the answer

relativeError = double(abs(M-actual)/actual)

relativeError =

0.064336380514477

M = double(M)

M =

0.623775746323754

**Appendix C – Question 3 Matlab Code**

syms f1(t) f2(s,t) h(s)

format long

% create the function variable and its integration endpoints

f1(t) = 13.8;

% create a function that can be passed to functions

a1 = 0;

b1 = 2;

% set S = 0

S1 = 0;

% the number of segments/points that will be used , assume >1

N\_s = 200;

% check if the value vary by a large enough threshold

% reset the sum of of area segments under the curve

S1 = 0;

% find the step size

dx = (b1-a1)/N\_s;

for i=1:N\_s

% calculate the integral using the simpsons method

x = a1 + (i-1)\*dx;

y = (dx/6)\*(f1(x) + 4\*f1(x+dx/2) + f1(x + dx));

S1 = S1 + y;

end

% multiply by 2 to get the final approximation

S1 = 2\*double(S1)

S1 =

55.200000000000003

RIGHT SIDE OF THE EQN

% create the function variable and its integration endpoints

f2(s,t) = sin(t);

% create a function that can be passed to functions

a2 = 0;

b2 = 2\*pi;

c2 = 0;

d2 = pi/2;

% want to check that the integral on the right is within 10% of the

% integral on the left

thresh = 0.1;

% the number of segments/points that will be used , assume >1

N\_s\_t = 1;

% set S = 0, temp value

S2 = 0;

temp = 0;

% find the step size on the y-axis

dy = (d2-c2)/N\_s\_t;

while abs(S1-S2)/S1 > 0.1

% calculate the integral using the simpsons method for hte y-axis

for i=1:N\_s\_t

y = c2 + (i-1)\*dy;

temp = temp + (dy/6)\*(f2(s,y) + 4\*f2(s,y+dy/2) + f2(s,y+dy));

end

h(s) = temp;

% find the step size on the x-axis

dx = (b2-a2)/N\_s\_t;

% calculate the integral using the simpsons method for x-axis

for i=1:N\_s\_t

x = a2 + (i-1)\*dx;

S2 = S2 + (dx/6)\*(h(x) + 4\*h(x+dx/2) + h(x+dx));

end

N\_s\_t = N\_s\_t + 1;

S2 = 13.8\*2\*2\*double(S2)/(2\*pi);

end

S2

S2 =

55.325849237570026

% find the absolute error

absolute\_error = abs(S1-S2)

absolute\_error =

0.125849237570023

**Appendix D – Question 4 Matlab Code & Lagrange Method**

format long

syms x\_l f\_l f\_d(s,t,z) g\_d(s,t,z)

% import the data we have

data = importdata('FP\_data\_Q4.txt');

x = data(:,1);

y = data(:,2);

% set the step size

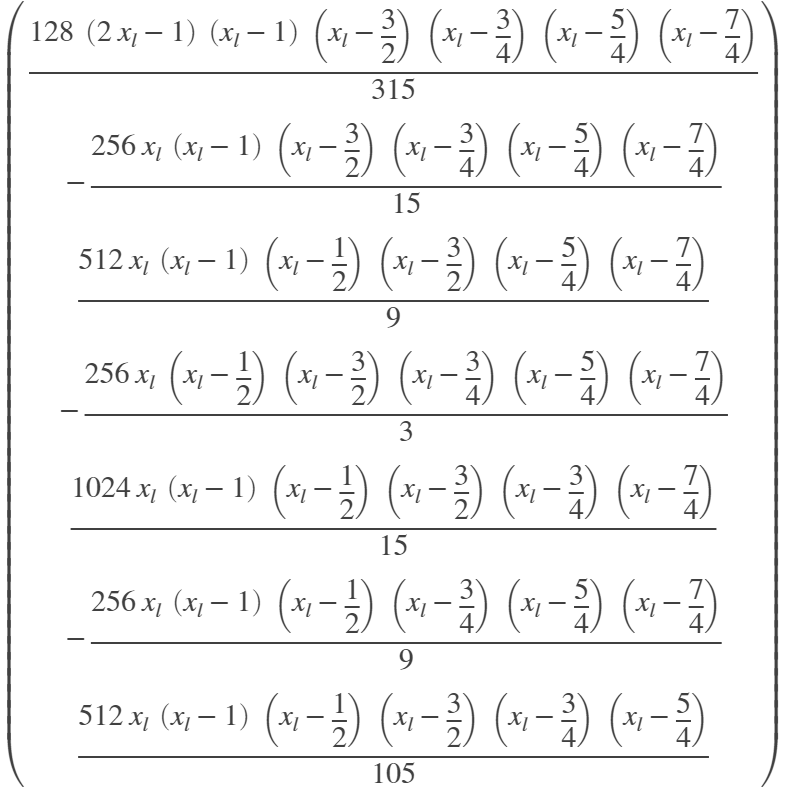
h = 0.125;

% interpolate more points

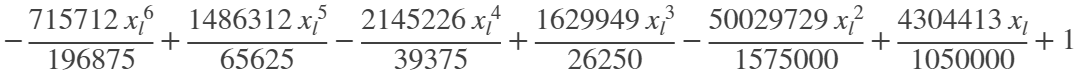
% find the lagrange function to interpolate more points

f\_l = LagrangePolynomial(x\_l,x,y)

L\_coeffs =



f\_l =



% put the new point back into the points

x = [x(1:2-1); 0.25; x(2:end)];

y = [y(1:2-1); double(subs(f\_l,x\_l,0.25)); y(2:end)];

% interpolate new set of points

x\_new = [x(1):h:x(length(x))];

y\_new = zeros(length(x\_new),1);

% calculate the new points that will be

for i=1:length(x\_new)

% for all the points we already have just insert them in

if rem(x\_new(i),0.25) == 0

y\_new(i) = y(find(x==x\_new(i)));

end

y\_new(i) = double(subs(f\_l,x\_l,x\_new(i)));

end

% plot the new points vs the old

plot(x\_new,y\_new,'--o')

hold on

plot(x,y,'--o')

title('Interpolated points vs the given measurements')

hold off

% initialize the matrices for the system of equations

B = [1;1;1];

A = zeros(3);

% find the inputs to the matrix using hte 8th order central difference

count = 0;

for i=5:10

cd\_1 = (y\_new(i-4)/280 - y\_new(i-3)\*4/105 + y\_new(i-2)/5 - y\_new(i-1)\*4/5 + y\_new(i+1)\*4/5 - y\_new(i+2)/5 + y\_new(i+3)\*4/105 - y\_new(i+4)/280)/h;

cd\_2 = (-y\_new(i-4)/560 + y\_new(i-3)\*8/315 - y\_new(i-2)/5 + y\_new(i-1)\*8/5 - y\_new(i)\*205/72 + y\_new(i+1)\*8/5 - y\_new(i+2)/5 + y\_new(i+3)\*8/315 - y\_new(i+4)/560)/h^2;

row = rem(count,3) + 1;

% take the average of the central calculated central difference and the

% points if they are next to each other

if rem(count,2)

A(row,1) = (A(row,1)+cd\_2)/2;

A(row,2) = (A(row,2)+cd\_1)/2;

A(row,3) = (A(row,3)+y\_new(i))/2;

else

A(row,1) = cd\_2;

A(row,2) = cd\_1;

A(row,3) = y\_new(i);

end

count = count + 1;

end

% find the coefficients

coeffs = inv(A)\*B

coeffs = 3×1

102 ×

0.033586704235173  
 0.225951996658204  
 1.010425447721065

% find the function of ode in order to plot it

syms y\_d(t)

coeffs = coeffs\*-1;

Dy = diff(y\_d);

D2y = diff(y\_d,2);

ode = coeffs(1)\*D2y + coeffs(2)\*Dy + coeffs(3)\*y\_d == 1;

ySol = dsolve(ode,y\_d(0)==1,Dy(0)==0);

% plot the calculated function vs the points

ezplot(ySol,[0,1.75])

hold on

plot(x,y,'--o')

title('Actual points vs the estimated function')

hold off

**Lagrange Method**

Function to Calculate the Lagrange Polynomial Interpolation.

inputs: x\_l: symbolic variable that will be used

x: x points

y: y points

return: polynomial function

function f = LagrangePolynomial(x\_l, x, y)

L\_coeffs = sym(ones(length(x),1));

syms x\_l

% find all the lagrange polynomials associated with the data

for i=1:length(x)

temp = 1;

for j=1:length(x)

if j ~= i

temp = temp \* (x\_l - x(j))/(x(i)-x(j));

end

end

L\_coeffs(i) = temp;

end

L\_coeffs

f = 0;

for i=1:length(x)

temp = y(i)\*L\_coeffs(i);

f = f + temp;

end

f = simplify(f);

end

**Appendix E – Question 5 Matlab Code**

format long

% input the matrices

A = [3 -5 47 20; 11 16 17 10; 56 22 11 -18; 17 66 -12 7];

B = [18; 26; 34; 82];

% get size of matrix

n = length(A);

% initialize the values of L and U to be A matrix so it can be reduced

L = eye(n);

U = A;

% compute the lower and upper triangle matrices

for i=1:n

% Row reducing

if U(i,i)==0

% always want the row with the largest value at pt (i,i)

max = max(abs(U(i:end,1)));

for j=1:n

% swap the rows to have the max value in the column at the

% diagonal

if max == abs(U(j,i))

temp = U(1,:);

U(1,:) = U(j,:);

U(j,:) = temp;

end

end

end

% check if the diagonal value is 1

if U(i,i)~=1

% perform the same changes to the L matrix

temp = eye(n);

temp(i,i)=U(i,i);

L = L \* temp;

% divide the row of by the value of diagonal so that its 1

U(i,:) = U(i,:)/U(i,i);

end

% for the other rows make sure the rows below the diagonal are 0

if i~=n

for j=i+1:n

% subtract 1 row by the diagonal row so that we can get 0

temp = eye(n);

temp(j,i) = U(j,i);

% manipulate the L matrix to mirror the changes

L = L\*temp;

U(j,:) = U(j,:)-U(j,i)\*U(i,:);

end

end

end

L

L = 4×4

102 ×

0.030000000000000 0 0 0  
 0.110000000000000 0.343333333333333 0 0  
 0.560000000000000 1.153333333333333 -3.445339805825242 0  
 0.170000000000000 0.943333333333333 1.484563106796116 -0.092698452954603

U

U = 4×4

1.000000000000000 -1.666666666666667 15.666666666666666 6.666666666666667  
 0 1.000000000000000 -4.524271844660193 -1.844660194174757  
 0 0 1.000000000000000 0.518330656296672  
 0 0 0 1.000000000000000

% solve for the unknowns

% LY = B

Y = inv(L)\*B

Y = 4×1

6.000000000000000  
 -1.165048543689320  
 0.486544368360245  
 -1.906432108560652

% UX = Y

X = inv(U)\*Y

X = 4×1

-1.077179830921188  
 1.990205466334715  
 1.474706574375536  
 -1.906432108560652

% compare with the other unknowns

x\_actual = inv(A)\*B

x\_actual = 4×1

-1.077179830921178  
 1.990205466334710  
 1.474706574375530  
 -1.906432108560641

% calculate the square error

SE = 0;

for i=1:length(X)

SE = SE + (x\_actual(i)-X(i))^2;

end

SE

SE =

2.641697956358863e-28

**Appendix F – Question 6 Matlab Code**

format long

% input the matrices

A = [3 -5 47 20; 11 16 17 10; 56 22 11 -18; 17 66 -12 7];

B = [18; 26; 34; 82];

A([1 3],:)=A([3 1],:);

A([2 4],:)=A([4 2],:)

A = 4×4

56 22 11 -18  
 17 66 -12 7  
 3 -5 47 20  
 11 16 17 10

B([1 3],:)=B([3 1],:);

B([2 4],:)=B([4 2],:)

B = 4×1

34  
 82  
 18  
 26

% get length on matrix

n = length(A);

% start off with coefficients/unkowns equal to 0

x = zeros(n,1);

% set number of iterations to be 100

N = 100;

for i=1:N

temp = zeros(n,1);

temp(1) = (B(1)-A(1,2)\*x(2)-A(1,3)\*x(3)-A(1,4)\*x(4))/A(1,1);

temp(2) = (B(2)-A(2,1)\*x(1)-A(2,3)\*x(3)-A(2,4)\*x(4))/A(2,2);

temp(3) = (B(3)-A(3,2)\*x(2)-A(3,1)\*x(1)-A(3,4)\*x(4))/A(3,3);

temp(4) = (B(4)-A(4,2)\*x(2)-A(4,3)\*x(3)-A(4,1)\*x(1))/A(4,4);

x = temp;

end

x

x = 4×1

-1.077080253093629  
 1.990142287156950  
 1.474621130739856  
 -1.906281024359803

actual = inv(A)\*B

actual = 4×1

-1.077179830921178  
 1.990205466334710  
 1.474706574375529  
 -1.906432108560640

% calculate the square error

SE = 0;

for i=1:length(X)

SE = SE + (actual(i)-x(i))^2;

end

SE

SE =

4.403440286156683e-08

**Appendix G – Question 7 Matlab Code**

Q7

format long

syms f(x)

% the function to be solved

f(x) = cos(x)\*cosh(x)-1;

% the threshold used for the relative error

thresh = 10^-4;

% display function to visualize zeros

fplot(f(x))

title('f(x) function')

% find the value of the real root to calculate relative error

p = @(x) cos(x)\*cosh(x)-1;

actual = fzero(p,[4, 5])

actual =

4.730040744862704

% continue until it finds the root

root = 0;

% set the step size

h = 0.5;

% since we don't want to catch the point at 0 we start after it

i = 0.51;

while 1

% we know there is a root if there is a sign change in between points

if f(i)\*f(i-h) < 0

x\_n = i;

x\_1 = i-h;

% check to see if root is within the relative error accepted by thresh

while abs(x\_n - x\_1)/x\_1 > thresh

% calculate the change in x at each iteration

dx = (f(x\_n)\*(x\_n-x\_1)/(f(x\_n)-f(x\_1)));

x\_1 = x\_n;

% calculate the new x

x\_n = x\_n - dx;

end

root = x\_n;

break;

end

i = i + h;

end

% display the value

relativeError = double(abs(root - actual)/actual)

relativeError =

1.786830646479475e-09

root = double(root)

root =

4.730040736410922

**Appendix H – Question 8 Matlab Code & Lagrange Method**

format long

% initialize variables for all the functions

syms f(t) x\_l

% import data

data = importdata('FP\_data\_Q8.txt');

x = data(:,1);

y = data(:,2);

% plot the given data

plot(x,y)

title('Scatter Plot of the given data')

hold off

lenData = length(data);

A1 = ones(lenData, 3);

B1 = zeros(lenData, 1);

% create the desired matrices

for i=1:lenData

A1(i,2) = x(i);

A1(i,3) = x(i)^2;

B1(i) = y(i);

end

% find the normal matrices on each side of the eqn

A\_T1 = transpose(A1);

sqr1 = A\_T1\*A1;

ATB1 = A\_T1\*B1;

% calculate the upper and lower triangle using cholosky

L1 = chol(sqr1, 'lower');

L\_T1 = chol(sqr1, 'upper');

z1 = inv(L1)\*ATB1;

x1 = inv(L\_T1)\*z1;

x1 = x1

x1 = 3×1

1.218357142857155  
 -0.516250000000007  
 0.111892857142858

% compare with the using the equation

x1 = inv(sqr1)\*ATB1

x1 = 3×1

1.218357142857143  
 -0.516249999999996  
 0.111892857142858

% show the final function

f(t) = x1(1) + x1(2)\*t + x1(3)\*t^2;

% function plotted against the points

fplot(f(t), [1,8], 'Color',[0,0.7,0.9])

hold on

plot(x,y,'Color',[0.9100,0.4100,0.1700])

legend({'f(t) = Fitted line', 'Given points'},'Location','Northwest')

title('Compare the fitted function with the points')

hold off

% find the point where there is the highest absolute error

max\_abs\_error = 0;

max\_index = 0;

% keep the index with the highest abs error

for i=1:length(x)

abs\_error = abs(f(x(i)) - y(i));

if abs\_error > max\_abs\_error

max\_index = i;

max\_abs\_error = abs\_error;

end

end

max\_index

max\_index =

4

max\_abs\_error = double(max\_abs\_error)

max\_abs\_error =

0.050642857142857

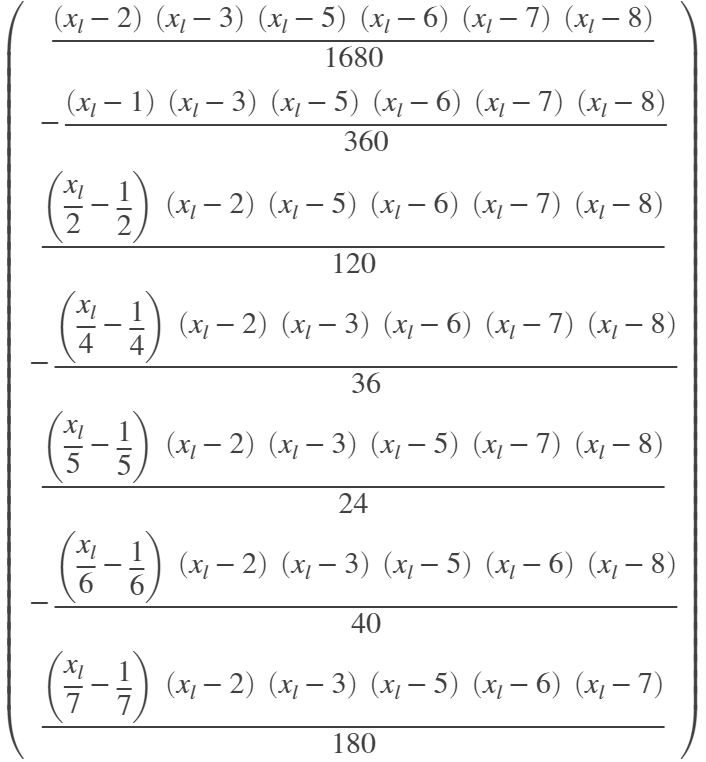
x(max\_index) = [];

y(max\_index) = [];

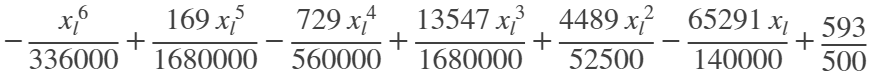
% take the index with the high abs and interpolate it with Lagrange

f\_l = LagrangePolynomial(x\_l, x, y)

L\_coeffs =



f\_l =



% interpolate the point of max error

ypt\_new = double(subs(f\_l,x\_l,max\_index))

ypt\_new =

0.962257142857143

% put the new point back into the points

x\_new = [x(1:max\_index-1); max\_index; x(max\_index:end)];

y\_new = [y(1:max\_index-1); ypt\_new; y(max\_index:end)];

% plot the new functions vs the orinal poly fit

plot(x\_new,y\_new,'Color',[0,0.7,0.9])

hold on

fplot(f(t),[1,8])

legend({'f(t) = Fitted line', 'Updated points'},'Location','Northwest')

title('Compare the fitted function with the new points')

hold off

**Lagrange Method**

Function to Calculate the Lagrange Polynomial Interpolation.

inputs: x\_l: symbolic variable that will be used

x: x points

y: y points

return: polynomial function

function f = LagrangePolynomial(x\_l, x, y)

L\_coeffs = sym(ones(length(x),1));

syms x\_l

% find all the lagrange polynomials associated with the data

for i=1:length(x)

temp = 1;

for j=1:length(x)

if j ~= i

temp = temp \* (x\_l - x(j))/(x(i)-x(j));

end

end

L\_coeffs(i) = temp;

end

L\_coeffs

f = 0;

for i=1:length(x)

temp = y(i)\*L\_coeffs(i);

f = f + temp;

end

f = simplify(f);

end

**Appendix I – Question 9 Matlab Code**

format long

syms y(t) f\_d(s,t,z) g\_d(s,t,z)

% find the exact solutions function first

Dy = diff(y);

% plug in the ode equation

ode = diff(y,t,2) + 2\*diff(y,t,1) + 4\*y == 0x`;

% input the ode boundary conditions

cond1 = y(0) == 2;

cond2 = Dy(0) == 0;

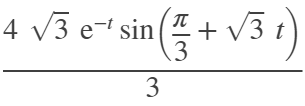
% solve for the pde

conds = [cond1 cond2];

ySol(t) = dsolve(ode,conds);

ySol = simplify(ySol)

ySol(t) =



% set the step size

h = 0.1;

% set the range for the data points

x = 0:h:5;

% input the derrivative functions associated the ODE

f\_d(s,t,z) = z;

g\_d(s,t,z) = -2\*z - 4\*t;

Runge Kutta Method

% create the list of points and put in the initial condition y = 2, z = 0

y\_rk = zeros(length(x),1);

y\_rk(1) = 2;

z\_rk = zeros(length(x),1);

z\_rk(1) = 0;

% compute the runge kutta

for i=1:length(y\_rk)-1

l1 = g\_d(x(i),y\_rk(i),z\_rk(i));

k1 = f\_d(x(i),y\_rk(i),z\_rk(i));

l2 = g\_d(x(i)+h/2,y\_rk(i)+k1\*h/2,z\_rk(i)+l1\*h/2);

k2 = f\_d(x(i)+h/2,y\_rk(i)+k1\*h/2,z\_rk(i)+l1\*h/2);

l3 = g\_d(x(i)+h/2,y\_rk(i)+k2\*h/2,z\_rk(i)+l2\*h/2);

k3 = f\_d(x(i)+h/2,y\_rk(i)+k2\*h/2,z\_rk(i)+l2\*h/2);

l4 = g\_d(x(i)+h,y\_rk(i)+k3\*h,z\_rk(i)+l3\*h);

k4 = f\_d(x(i)+h,y\_rk(i)+k3\*h,z\_rk(i)+l3\*h);

y\_rk(i+1) = y\_rk(i) + h\*(k1+2\*k2+2\*k3+k4)/6;

z\_rk(i+1) = z\_rk(i) + h\*(l1+2\*l2+2\*l3+l4)/6;

end

Eulers Method

% create the list of y, z points that will be used and insert the ICs

y\_e = zeros(length(x),1);

y\_e(1) = 2;

z\_e = zeros(length(x),1);

z\_e(1) = 0;

for i=1:length(x)-1

y\_e(i+1) = y\_e(i) + h\*f\_d(x(i),y\_e(i),z\_e(i));

z\_e(i+1) = z\_e(i) + h\*g\_d(x(i),y\_e(i),z\_e(i));

end

% compare plots

fplot(ySol,[0,5])

hold on

plot(x,y\_rk,'Color',[0,0.7,0.9])

hold on

plot(x,y\_e,'Color',[0.9100,0.4100,0.1700])

legend({'Actual','Runge-Kutta', 'Eulers'},'Location','Southwest')

title('Compare the runge-kutta and eulers methods vs the actualof approximation')

hold off

avg\_absErr\_eu = 0;

avg\_absErr\_rk = 0;

for i=1:length(x)

avg\_absErr\_eu = avg\_absErr\_eu + abs(y\_e(i) - ySol(x(i)));

avg\_absErr\_rk = avg\_absErr\_rk + abs(y\_rk(i) - ySol(x(i)));

end

avg\_absErr\_eu = double(avg\_absErr\_eu/length(x))

avg\_absErr\_eu =

0.063496337297686

avg\_absErr\_rk = double(avg\_absErr\_rk/length(x))

avg\_absErr\_rk =

7.562639331815784e-06

**Appendix J – Question 10 Matlab Code**

format long

syms f\_d(s,t)

% where s = x and y = y

% set the step size

h = 0.1;

% set the range of points to step through

x = 0:h:2;

% input the derrivative function in to be solved

f\_d(s,t) = -1.2\*t + 7\*exp(-0.3\*s);

% create the list of points and put in the initial condition y = 3

y = zeros(length(x),1);

y(1) = 3;

% compute the runge kutta-4

% for i=1:length(y)-1

% k1 = f\_d(x(i),y(i));

% k2 = f\_d(x(i)+h/2,y(i)+k1\*h/2);

% k3 = f\_d(x(i)+h/2,y(i)+k2\*h/2);

% k4 = f\_d(x(i)+h,y(i)+k3\*h);

%

% y(i+1) = y(i) + h\*(k1+2\*k2+2\*k3+k4)/6;

% end

for i=1:length(y)-1

k1 = f\_d(x(i),y(i));

k2 = f\_d(x(i)+h,y(i)+k1\*h);

y(i+1) = y(i) + h\*(k1+k2)/2;

end

y

y = 21×1

3.000000000000000  
 3.309255936741978  
 3.564486678164910  
 3.772351972056787  
 3.938743627523325  
 4.068872632596348  
 4.167346430372343  
 4.238237464230580  
 4.285143977397482  
 4.311243940973443

plot(x,y)

title('Runge-Kutta 2nd order Plot for differential equation')

**Appendix K – Question 11 Matlab Code**

format long

syms y(x)

% find the exact solutions function first

Dy = diff(y);

% plug in the ode equation

ode = diff(y,x,2) + 0.25\*diff(y,x,1) == 8;

% input the ode boundary conditions

cond1 = y(0) == 0;

cond2 = y(10) == 0;

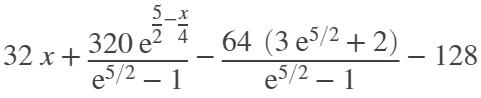
% solve for the pde

conds = [cond1 cond2];

ySol(x) = dsolve(ode,conds);

ySol = simplify(ySol)

ySol(x) =



% set the delta x to 1 and find the points in the domain

del\_X = 1;

% set last number of pts

N = 10;

% set the boundary conditions

x = 0:del\_X:N;

% convergence conditions

accuracy = 10^-4;

% create the unknown matrices for y

y = zeros(N+1,1);

y(1) = 0;

y(N+1) = 0;

% allocate memory for temp value

temp = zeros(11,1);

% stopping condition based on chose accuracy

stop = 1;

while stop>0

% calculate the new values of the unknowns

for i=1:N-1

temp(i+1) = (y(i+2)\*(1+del\_X/8)+y(i)\*(1-del\_X/8)-8\*del\_X^2)/2;

end

temp(1) = y(1);

temp(N+1) = y(N+1);

stop = 0;

% stop the function when it hits the threshold of accuracy

for i=1:N+1

if(abs(y(i)-temp(i)) > accuracy)

stop = 1;

break;

end

end

y = temp;

end

% plot functions

plot(x,y,'Color',[0,0.7,0.9])

hold on

fplot(ySol, [0,10],'Color',[0.9100,0.4100,0.1700])

legend({'Estimated','Exact'},'Location','southwest')

title('ODE plot of calculated vs exact')

hold off

**Appendix L – Question 12 Matlab Code**

format long

syms y(x) f\_d(s,t,z) g\_d(s,t,z)

% find the exact solutions function first

Dy = diff(y);

% plug in the ode equation

ode = diff(y,x,2) + 0.25\*diff(y,x,1) == 8;

% input the ode boundary conditions

cond1 = y(0) == 0;

cond2 = y(10) == 0;

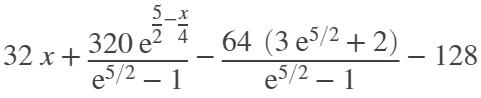
% solve for the pde

conds = [cond1 cond2];

ySol(x) = dsolve(ode,conds);

ySol = simplify(ySol)

ySol(x) =



Compute the shooting method

% set the step size

h = 1;

% set the range for the data points

x = 0:h:10;

% boundary conditions

y\_0 = 0;

y\_10 = 0;

% input the derrivative functions associated the ODE

f\_d(s,t,z) = z;

g\_d(s,t,z) = 8 - z/4;

% guess random points to start so we can interpolate a distance

yd\_top = 1;

y\_top = 0;

yd\_bottom = 0;

y\_bottom = 0;

while 1

% do the process for the top pt

% create the list of y, z points that will be used and insert the ICs

y\_rk\_top = zeros(length(x),1);

y\_rk\_top(1) = y\_0;

z\_rk\_top = zeros(length(x),1);

z\_rk\_top(1) = yd\_top;

% compute the runge kutta

for i=1:length(y\_rk\_top)-1

l1 = g\_d(x(i),y\_rk\_top(i),z\_rk\_top(i));

k1 = f\_d(x(i),y\_rk\_top(i),z\_rk\_top(i));

l2 = g\_d(x(i)+h/2,y\_rk\_top(i)+k1\*h/2,z\_rk\_top(i)+l1\*h/2);

k2 = f\_d(x(i)+h/2,y\_rk\_top(i)+k1\*h/2,z\_rk\_top(i)+l1\*h/2);

l3 = g\_d(x(i)+h/2,y\_rk\_top(i)+k2\*h/2,z\_rk\_top(i)+l2\*h/2);

k3 = f\_d(x(i)+h/2,y\_rk\_top(i)+k2\*h/2,z\_rk\_top(i)+l2\*h/2);

l4 = g\_d(x(i)+h,y\_rk\_top(i)+k3\*h,z\_rk\_top(i)+l3\*h);

k4 = f\_d(x(i)+h,y\_rk\_top(i)+k3\*h,z\_rk\_top(i)+l3\*h);

y\_rk\_top(i+1) = y\_rk\_top(i) + h\*(k1+2\*k2+2\*k3+k4)/6;

z\_rk\_top(i+1) = z\_rk\_top(i) + h\*(l1+2\*l2+2\*l3+l4)/6;

end

y\_top = y\_rk\_top(length(x));

% repeat for bottom point

% create the list of y, z points that will be used and insert the ICs

y\_rk\_bottom = zeros(length(x),1);

y\_rk\_bottom(1) = y\_0;

z\_rk\_bottom = zeros(length(x),1);

z\_rk\_bottom(1) = yd\_bottom;

% compute the runge kutta

for i=1:length(y\_rk\_bottom)-1

l1 = g\_d(x(i),y\_rk\_bottom(i),z\_rk\_bottom(i));

k1 = f\_d(x(i),y\_rk\_bottom(i),z\_rk\_bottom(i));

l2 = g\_d(x(i)+h/2,y\_rk\_bottom(i)+k1\*h/2,z\_rk\_bottom(i)+l1\*h/2);

k2 = f\_d(x(i)+h/2,y\_rk\_bottom(i)+k1\*h/2,z\_rk\_bottom(i)+l1\*h/2);

l3 = g\_d(x(i)+h/2,y\_rk\_bottom(i)+k2\*h/2,z\_rk\_bottom(i)+l2\*h/2);

k3 = f\_d(x(i)+h/2,y\_rk\_bottom(i)+k2\*h/2,z\_rk\_bottom(i)+l2\*h/2);

l4 = g\_d(x(i)+h,y\_rk\_bottom(i)+k3\*h,z\_rk\_bottom(i)+l3\*h);

k4 = f\_d(x(i)+h,y\_rk\_bottom(i)+k3\*h,z\_rk\_bottom(i)+l3\*h);

y\_rk\_bottom(i+1) = y\_rk\_bottom(i) + h\*(k1+2\*k2+2\*k3+k4)/6;

z\_rk\_bottom(i+1) = z\_rk\_bottom(i) + h\*(l1+2\*l2+2\*l3+l4)/6;

end

y\_bottom = y\_rk\_bottom(length(x));

if y\_top\*y\_bottom < 0

break;

else

yd\_bottom = yd\_bottom - 10;

yd\_top = yd\_top + 1;

end

end

p = yd\_top + (yd\_bottom-yd\_top)\*(y\_10 - y\_top)/(y\_bottom-y\_top)

p =

-55.154820877021550

% create the list of points and put in the initial condition y = 2, z = 0

y\_rk = zeros(length(x),1);

y\_rk(1) = y\_0;

z\_rk = zeros(length(x),1);

z\_rk(1) = p;

% compute the runge kutta

for i=1:length(y\_rk)-1

l1 = g\_d(x(i),y\_rk(i),z\_rk(i));

k1 = f\_d(x(i),y\_rk(i),z\_rk(i));

l2 = g\_d(x(i)+h/2,y\_rk(i)+k1\*h/2,z\_rk(i)+l1\*h/2);

k2 = f\_d(x(i)+h/2,y\_rk(i)+k1\*h/2,z\_rk(i)+l1\*h/2);

l3 = g\_d(x(i)+h/2,y\_rk(i)+k2\*h/2,z\_rk(i)+l2\*h/2);

k3 = f\_d(x(i)+h/2,y\_rk(i)+k2\*h/2,z\_rk(i)+l2\*h/2);

l4 = g\_d(x(i)+h,y\_rk(i)+k3\*h,z\_rk(i)+l3\*h);

k4 = f\_d(x(i)+h,y\_rk(i)+k3\*h,z\_rk(i)+l3\*h);

y\_rk(i+1) = y\_rk(i) + h\*(k1+2\*k2+2\*k3+k4)/6;

z\_rk(i+1) = z\_rk(i) + h\*(l1+2\*l2+2\*l3+l4)/6;

end

plot(x,y\_rk,'Color',[0,0.7,0.9])

hold on

fplot(ySol,[0,10],'Color',[0.9100,0.4100,0.1700])

legend({'Estimated','Exact'},'Location','southwest')

title('Compare actual value with the calculated')

hold off