Bryan Jay
260738764

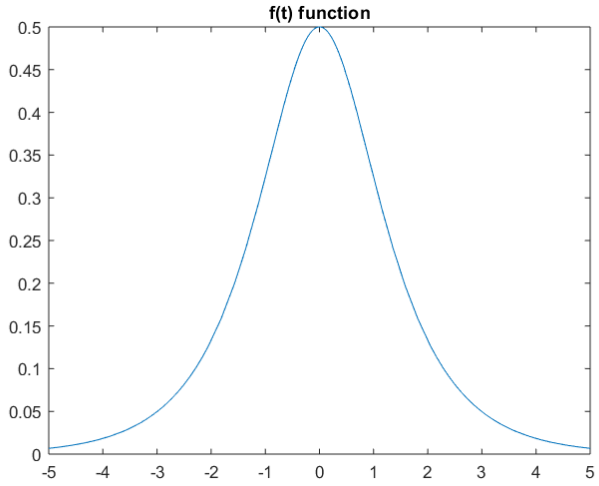
ECSE 443 – Final Project

Question 1

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

$$I = \int_0^{\infty} \frac{1}{e^x + e^{-x}} dx$$

The inner function $f(x) = \frac{1}{e^x + e^{-x}}$, 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 $[6, \infty)$, $f(x) \approx 0$. This is proven by using the Matlab integral function.

Range	I
$[0, \infty)$	0.785398163397448
$[0,20)$	0.785398163397454
$[0,6)$	0.782919416297423

So, the integral with the range $[0, \infty)$ can be adjusted to be approximately $[0,6]$. However, to ensure a higher accuracy we'll take slightly above such value by using range $[0,20]$. The integral to be evaluated can be modified to be:

$$I = \int_0^{20} \frac{1}{e^x + e^{-x}} dx$$

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:

$$\int_a^b P(x)\,dx = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right].$$

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:

$$I = \int_0^1 \frac{x^2}{\sqrt{1-x^2}}\,dx$$

The requirement was to use a step size h = 0.01. The step size is calculated with $h = \frac{b-a}{N}$, where a and b are the boundary conditions and N is the number of segments that will be used. Knowing that $b - a = 1$ and $h = 0.01$, then $N = 100$. 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:

$$M_n = \frac{b-a}{n}(f(m_1) + f(m_2) + \ldots f(m_n)),$$

$$m_k = \frac{t_k + t_{k-1}}{2} = a + \frac{2k-1}{2n}(b-a).$$

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.

$$\int_L A\,dL = \int\int_S \bar{D}\,d\bar{S}$$

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.

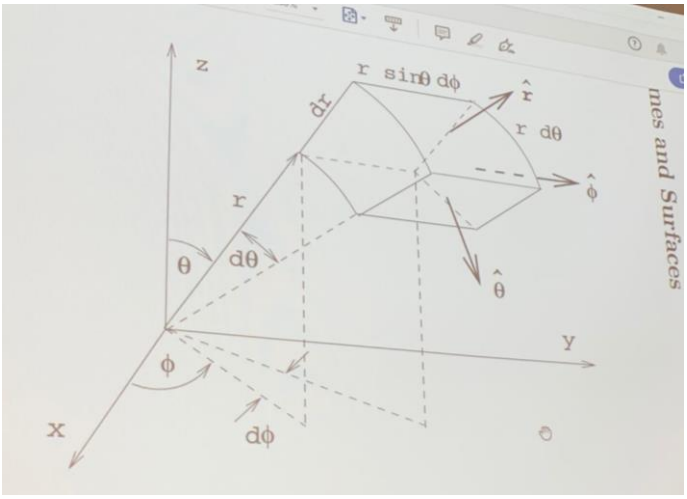
$$\int_{-2}^{2} A\, dL$$

$$2\int_{0}^{2} 13.8\, dL$$

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.

$$\int_{\theta}\int_{\varphi} \overline{D}\, d\varphi d\theta$$

$$\int_{0}^{2\pi}\int_{0}^{\pi} \overline{D} r^2 \sin\theta\, d\varphi d\theta$$

$$\int_{0}^{2\pi}\int_{0}^{\pi} \frac{\lambda}{2\pi r} r^2 \sin\theta\, d\varphi d\theta$$

$$2\int_{0}^{2\pi}\int_{0}^{\frac{\pi}{2}} \frac{\lambda}{2\pi} r \sin\theta\, d\varphi d\theta$$

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.

Step Size	0.01
Number of Segments φ	157
Number of Segments θ	628
Integral Value	55.200000000192055 µC

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

$$2\int_0^{2\pi}\frac{\pi}{2}\frac{\lambda}{2\pi}r\sin\theta\, d\varphi d\theta$$

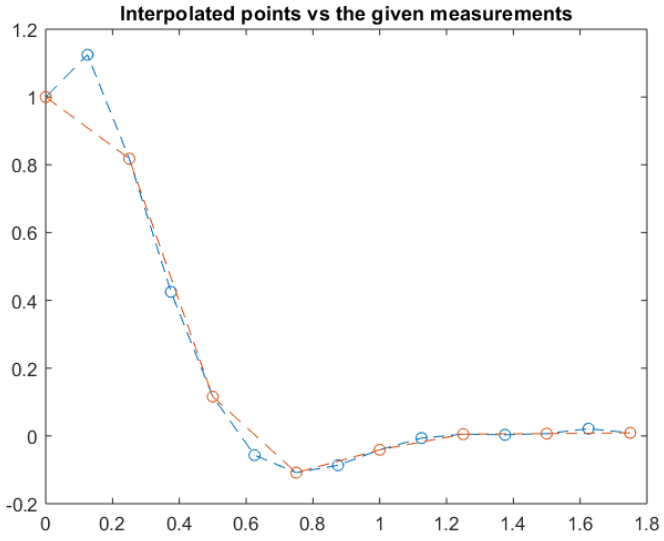
$$\frac{\lambda}{2}r\int_0^{2\pi}\sin\theta\, d\varphi d\theta$$

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	55.200000000192055 µC
Absolute Error	1.920525960485975e-10

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:

$$f'(x) = \frac{\frac{f(i-4h)}{280} - \frac{4f(i-3h)}{105} + \frac{f(i-2h)}{5} - \frac{4f(i-h)}{5} + \frac{4f(i+h)}{5} - \frac{f(i+2h)}{5} + \frac{4f(i+3h)}{105} - \frac{f(i+4h)}{208}}{h}$$

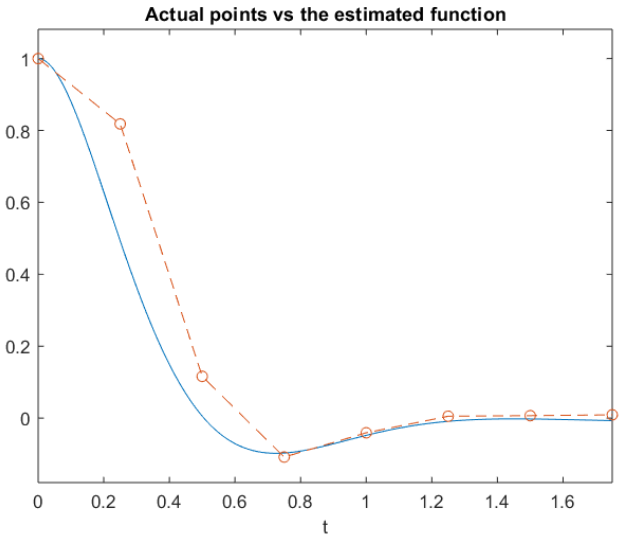
$$f''(x) = \frac{-\frac{f(i-4h)}{560} + \frac{8f(i-3h)}{315} - \frac{f(i-2)}{5} + \frac{8f(i-1)}{5} - \frac{205f(i)}{72} + \frac{8f(i+1)}{5} - \frac{f(i+2)}{5} + \frac{8f(i+3)}{315} - \frac{f(i+4)}{560}}{h^2}$$

$$\begin{bmatrix} y_A''(x) & y_A'(x) & y_A(x) \\ y_B''(x) & y_B'(x) & y_B(x) \\ y_C''(x) & y_C'(x) & y_C(x) \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

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.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

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.

$$-l_{i,n} := -\frac{a_{i,n}^{(n-1)}}{a_{n,n}^{(n-1)}}$$

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:

$$L = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 11 & \frac{103}{3} & 0 & 0 \\ 56 & \frac{346}{3} & -344.54 & 0 \\ 17 & \frac{283}{3} & 148.46 & -9.2698 \end{bmatrix}$$

$$U = \begin{bmatrix} 1 & -\frac{5}{3} & \frac{47}{3} & \frac{20}{3} \\ 0 & 1 & -4.524 & -1.8447 \\ 0 & 0 & 1 & 0.5183 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The unknowns were found by these equations and gave the following results.

$$LY = B$$

$$UX = Y$$

X_1	-1.077179830921188
X_2	1.990205466334715
X_3	1.474706574375536
X_4	-1.906432108560652

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:

$$x_{k+1} = g(x_k)$$

where $x_0 = 0$, 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:

$$g'(x^*) < 1$$

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

$$\begin{bmatrix} 3 & -5 & 47 & 20 \\ 11 & 16 & 17 & 10 \\ 56 & 22 & 11 & -18 \\ 17 & 66 & -12 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 18 \\ 26 \\ 34 \\ 82 \end{bmatrix}$$

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.

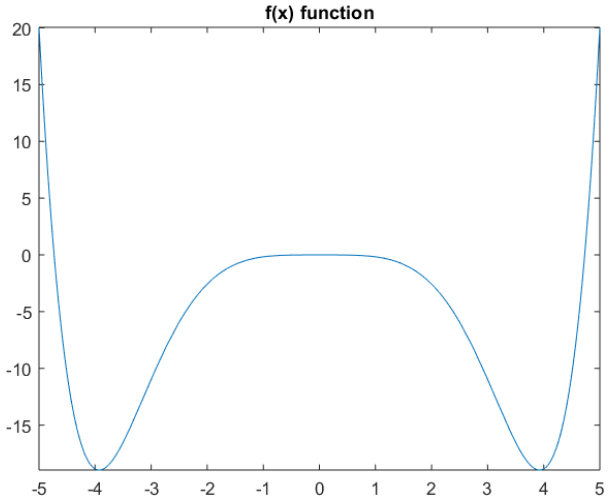
$$\begin{bmatrix} 56 & 22 & 11 & -18 \\ 17 & 66 & -12 & 7 \\ 3 & -5 & 47 & 20 \\ 11 & 16 & 17 & 10 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 34 \\ 82 \\ 18 \\ 26 \end{bmatrix}$$

Using 100 iterations the system of equations converges to:

x_1	x_2	x_3	x_4
-1.077080253093629	1.990142287156950	1.474621130739856	-1.906281024359803

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.

$$RE\ of\ precision = \frac{|calculated - previous|}{previous}$$

$$x_{i+1} = x_i - \frac{f(x_i) * (x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}$$

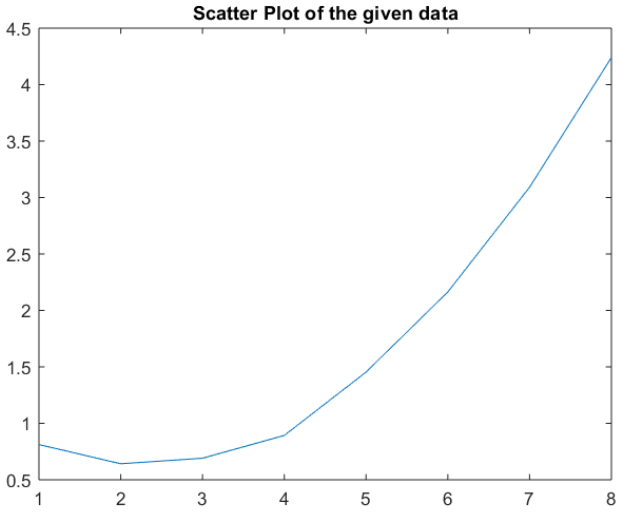
Using the above function, we calculate the roots from until we reach the targeted threshold for error of 10^{-4}. Since the function is even and mirrored on the x-axis the next zero is to the right and the left at the same distance apart but just with a negative.

$$RE = \frac{|actual - calculated|}{actual}$$

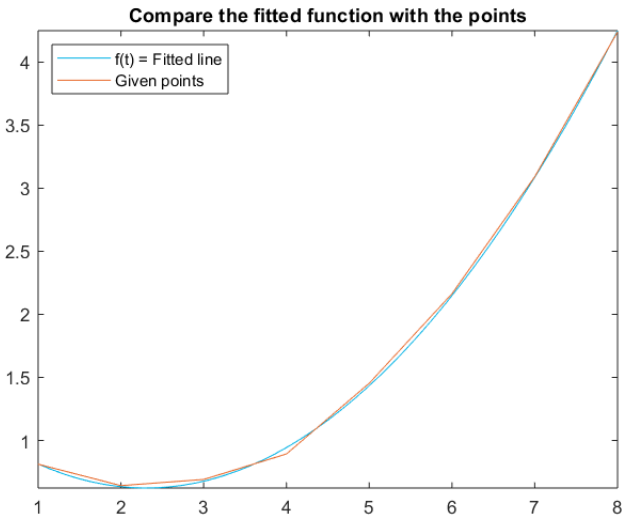
Relative Error	1.786830646479475e-09
Root (positive x-axis)	4.730040736410922
Root (negative x-axis)	- 4.730040736410922

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:



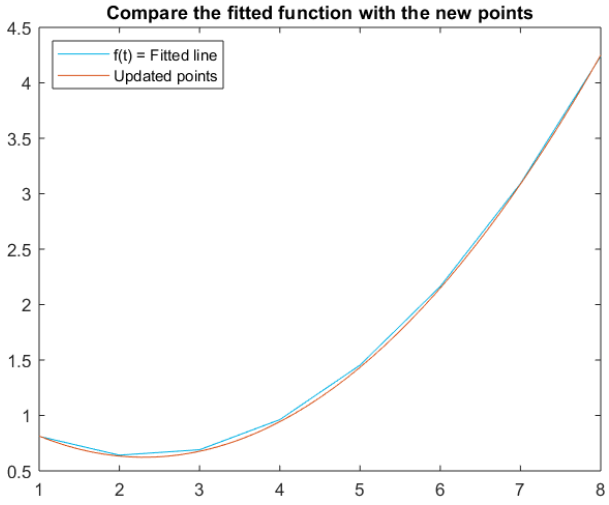
In order to find a point that maybe wrong and out of place, I found the absolute error at each point on the graph. Using the Lagrange Polynomial Interpolation that we used in assignment 3, we can use the similar method to calculate and fix the wrong point. We can find the new point using the interpolation.

$$Absolute\ Error = |actual - calculated|$$

$$\texttt{f\_l} = -\frac{x_l^6}{336000} + \frac{169\,x_l^5}{1680000} - \frac{729\,x_l^4}{560000} + \frac{13547\,x_l^3}{1680000} + \frac{4489\,x_l^2}{52500} - \frac{65291\,x_l}{140000} + \frac{593}{500}$$

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.

$$\frac{d^2y(t)}{dt^2} + 2\frac{dy(t)}{dt} + 4y(t) = 0$$

$$\begin{cases} f = \dfrac{dy}{dt} = z \\ g = \dfrac{dz}{dt} = -2z - 4y \end{cases}$$

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:

$$k_1 = f(x_i, y_i, z_i)$$

$$l_1 = g(x_i, y_i, z_i)$$

$$k_2 = f(x_i + \frac{h}{2}, y_i + \frac{k_1 h}{2}, z_i + \frac{l_1 h}{2})$$

$$l_2 = g(x_i + \frac{h}{2}, y_i + \frac{k_1 h}{2}, z_i + \frac{l_1 h}{2})$$

$$k_3 = f(x_i + \frac{h}{2}, y_i + \frac{k_2 h}{2}, z_i + \frac{l_2 h}{2})$$

$$l_3 = g(x_i + \frac{h}{2}, y_i + \frac{k_2 h}{2}, z_i + \frac{l_2 h}{2})$$

$$k_4 = f(x_i + h, y_i + k_3 h, z_i + l_3 h)$$

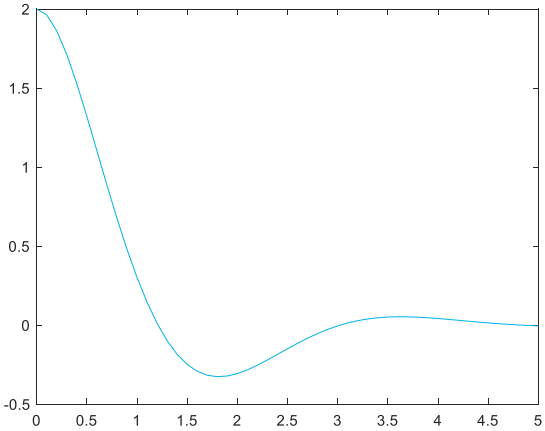
$$l_4 = g(x_i + h, y_i + k_3 h, z_i + l_3 h)$$

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.

$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$z_{i+1} = z_i + \frac{h}{6}(l_1 + 2l_2 + 2l_3 + l_4)$$

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

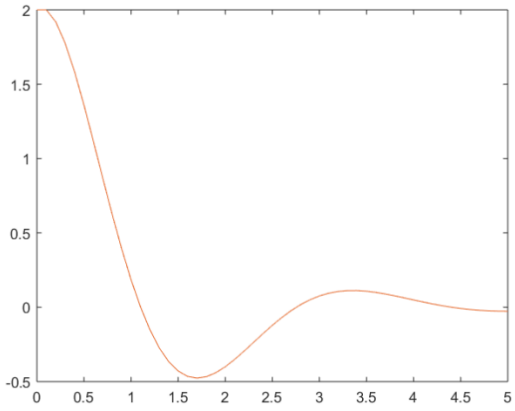


Now to calculate the plot using Euler's method we use the same functions derived at the beginning named f and g. Additionally, we create the same matrices filled with the initial conditions for the first point. The equations associated with Euler's method to calculate the points are much shorter compared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

$$y_{i+1} = y_i + hf(x_i, y_i, z_i)$$

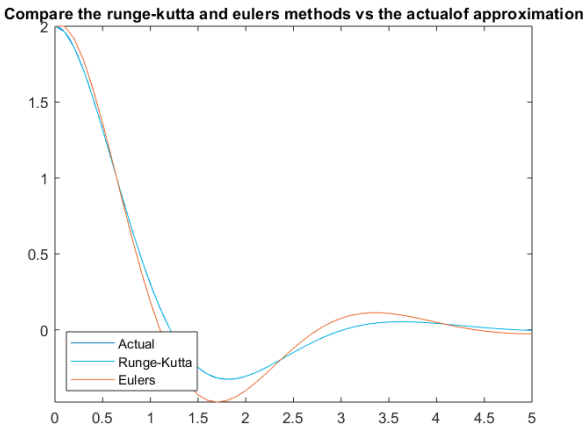
$$z_{i+1} = z_i + hf(x_i, y_i, z_i)$$

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:

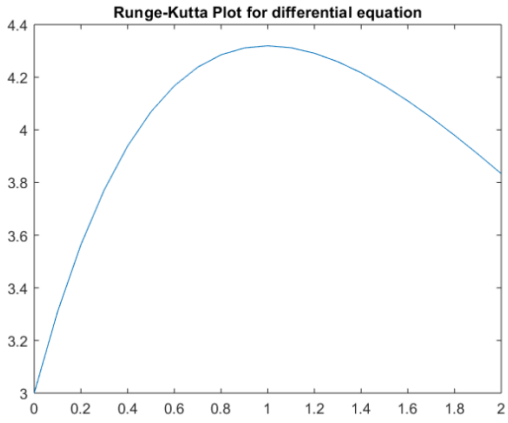
$$k_1 = f(x_i, y_i)$$

$$k_2 = f(x_i + h, y_i + k_1 h)$$

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.

$$y_{i+1} = y_i + \frac{h}{2}(k_1 + k_2)$$

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.

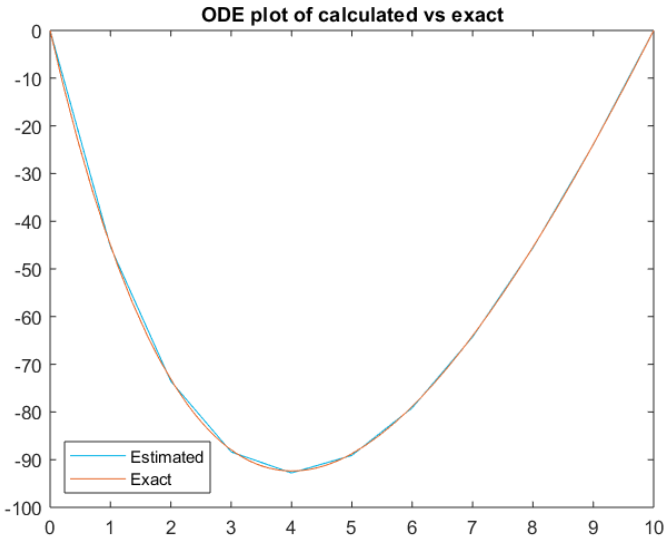
$$\begin{cases} \dfrac{d^2y(t)}{dt^2} \cong \dfrac{y_{i+1} - 2y_i + y_{i-1}}{\Delta t^2} \\ \dfrac{dy(t)}{dt} \cong \dfrac{y_{i+1} - y_{i-1}}{2\Delta t} \end{cases}$$

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

$$\frac{d^2y(t)}{dt^2} + \frac{1}{4}\frac{dy(t)}{dt} = 8$$

$$y_{i+1} = \frac{y_i\left(1 - \frac{\Delta t}{8}\right) + y_{i+2}\left(1 + \frac{\Delta t}{8}\right) - 8\Delta t^2}{2}$$

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.

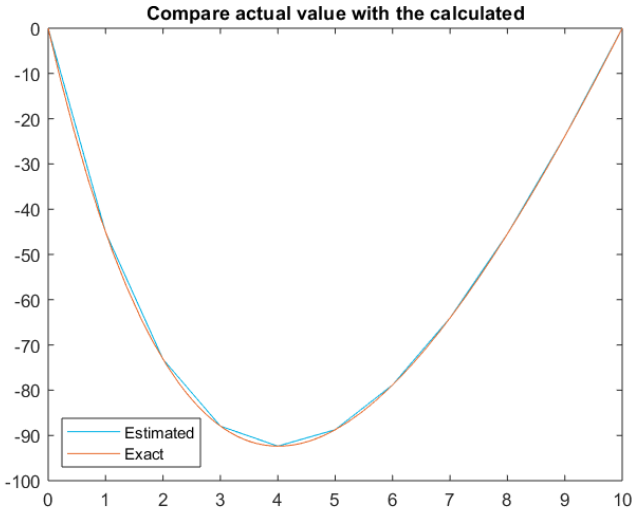


Question 12

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.

$$\begin{cases} \dfrac{dy(t)}{dt} = z \\ \dfrac{dz}{dt} = 8 - \dfrac{z}{4} \end{cases}$$

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 =
2/3
```

```
% 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) = 13.8*2*sin(t)/(2*pi);
% create a function that can be passed to functions
a2 = 0;
b2 = 2*pi;
```

```
c2 = 0;
d2 = pi/2;
% the number of segments/points that will be used , assume >1
N_s_t = 157;
% set S = 0, temp value
S2 = 0;
temp = 0;
% find the step size on the y-axis
dy = (d2-c2)/N_s_t;
% 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
% the number of segments in the other axis
N_s_s = 628;
h(s) = temp;
% find the step size on the x-axis
dx = (b2-a2)/N_s_s;
% calculate the integral using the simpsons method for x-axis
for i=1:N_s_s
    x = a2 + (i-1)*dx;
    S2 = S2 + (dx/6)*(h(x) + 4*h(x+dx/2) + h(x+dx));
end
S2 = 2*double(S2)
```

```
S2 =
  55.200000000192055
```

```
% find the absolute error
absolute_error = abs(S1-S2)
```

```
absolute_error =
     1.920525960485975e-10
```

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 =
```

$$\begin{pmatrix} \frac{128\,(2x_l-1)\,(x_l-1)\left(x_l-\frac{3}{2}\right)\left(x_l-\frac{3}{4}\right)\left(x_l-\frac{5}{4}\right)\left(x_l-\frac{7}{4}\right)}{315} \\ -\frac{256\,x_l\,(x_l-1)\left(x_l-\frac{3}{2}\right)\left(x_l-\frac{3}{4}\right)\left(x_l-\frac{5}{4}\right)\left(x_l-\frac{7}{4}\right)}{15} \\ \frac{512\,x_l\,(x_l-1)\left(x_l-\frac{1}{2}\right)\left(x_l-\frac{3}{2}\right)\left(x_l-\frac{5}{4}\right)\left(x_l-\frac{7}{4}\right)}{9} \\ -\frac{256\,x_l\left(x_l-\frac{1}{2}\right)\left(x_l-\frac{3}{2}\right)\left(x_l-\frac{3}{4}\right)\left(x_l-\frac{5}{4}\right)\left(x_l-\frac{7}{4}\right)}{3} \\ \frac{1024\,x_l\,(x_l-1)\left(x_l-\frac{1}{2}\right)\left(x_l-\frac{3}{2}\right)\left(x_l-\frac{3}{4}\right)\left(x_l-\frac{7}{4}\right)}{15} \\ -\frac{256\,x_l\,(x_l-1)\left(x_l-\frac{1}{2}\right)\left(x_l-\frac{3}{4}\right)\left(x_l-\frac{5}{4}\right)\left(x_l-\frac{7}{4}\right)}{9} \\ \frac{512\,x_l\,(x_l-1)\left(x_l-\frac{1}{2}\right)\left(x_l-\frac{3}{2}\right)\left(x_l-\frac{3}{4}\right)\left(x_l-\frac{5}{4}\right)}{105} \end{pmatrix}$$

f_1 =

$$-\frac{715712\,x_l^6}{196875}+\frac{1486312\,x_l^5}{65625}-\frac{2145226\,x_l^4}{39375}+\frac{1629949\,x_l^3}{26250}-\frac{50029729\,x_l^2}{1575000}+\frac{4304413\,x_l}{1050000}+1$$

```
% 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_1,x_1,0.25)); 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_1,x_1,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
10^2 ×
   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
10^2 ×
   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 = inv(A)*B
```

```
x = 4×1
  -1.077179830921178
   1.990205466334710
   1.474706574375530
  -1.906432108560641
```

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
```

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 =
```

$$\begin{pmatrix} \frac{(x_l-2)\,(x_l-3)\,(x_l-5)\,(x_l-6)\,(x_l-7)\,(x_l-8)}{1680} \\ -\frac{(x_l-1)\,(x_l-3)\,(x_l-5)\,(x_l-6)\,(x_l-7)\,(x_l-8)}{360} \\ \frac{\left(\frac{x_l}{2}-\frac{1}{2}\right)\,(x_l-2)\,(x_l-5)\,(x_l-6)\,(x_l-7)\,(x_l-8)}{120} \\ -\frac{\left(\frac{x_l}{4}-\frac{1}{4}\right)\,(x_l-2)\,(x_l-3)\,(x_l-6)\,(x_l-7)\,(x_l-8)}{36} \\ \frac{\left(\frac{x_l}{5}-\frac{1}{5}\right)\,(x_l-2)\,(x_l-3)\,(x_l-5)\,(x_l-7)\,(x_l-8)}{24} \\ -\frac{\left(\frac{x_l}{6}-\frac{1}{6}\right)\,(x_l-2)\,(x_l-3)\,(x_l-5)\,(x_l-6)\,(x_l-8)}{40} \\ \frac{\left(\frac{x_l}{7}-\frac{1}{7}\right)\,(x_l-2)\,(x_l-3)\,(x_l-5)\,(x_l-6)\,(x_l-7)}{180} \end{pmatrix}$$

```
f_l =
```

$$-\frac{x_l^6}{336000}+\frac{169\,x_l^5}{1680000}-\frac{729\,x_l^4}{560000}+\frac{13547\,x_l^3}{1680000}+\frac{4489\,x_l^2}{52500}-\frac{65291\,x_l}{140000}+\frac{593}{500}$$

```
% 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) =
```

$$\frac{4\sqrt{3}\,e^{-t}\sin\left(\frac{\pi}{3}+\sqrt{3}\,t\right)}{3}$$

```
% 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) =
```

$$32\,x+\frac{320\,e^{\frac{5}{2}-\frac{x}{4}}}{e^{5/2}-1}-\frac{64\,\left(3\,e^{5/2}+2\right)}{e^{5/2}-1}-128$$

```
% 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) =
```

$$32\,x+\frac{320\,e^{\frac{5}{2}-\frac{x}{4}}}{e^{5/2}-1}-\frac{64\,\left(3\,e^{5/2}+2\right)}{e^{5/2}-1}-128$$

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
```