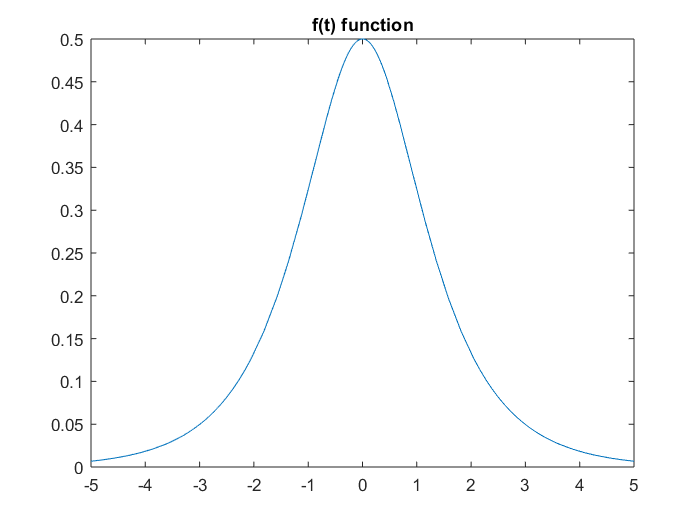
**ECSE 443 – Final Project**

|  |  |
| --- | --- |
| 1 | DONE |
| 2 | DONE |
| 3 | INCOMPLETE |
| 4 | INCOMPLETE |
| 5 | DONE |
| 6 | DONE |
| 7 | DONE |
| 8 | DONE |
| 9 | DONE |
| 10 | DONE |
| 11 | DONE |
| 12 | DONE |

**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.

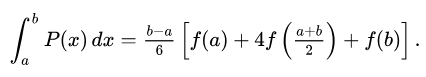


The graph demonstrates that the function, f(x), saturates at approximately 6. Therefore, the values after 5 are approximately 0, which means that from the range , . This is proven by using the Matlab integral function.

|  |  |
| --- | --- |
| **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:

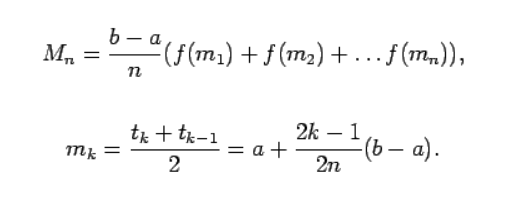


|  |  |
| --- | --- |
| **Number of Segments (Midpoint)** | 19 |
| **I** | 0.785309462232758 |
| **Absolute Error** | 8.870116469052163e-05 |

**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:



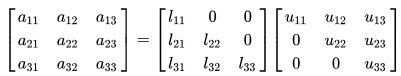
|  |  |
| --- | --- |
| **Number of Segments (Midpoint)** | 100 |
| **I** | 0.623775746323754 |
| **Relative Error** | 0.064336380514477 |

**Question 3**

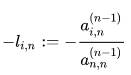
**Question 4**

**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.

|  |  |
| --- | --- |
| **X1** | -1.077179830921188 |
| **X2** | 1.990205466334715 |
| **X3** | 1.474706574375536 |
| **X4** | -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:

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.

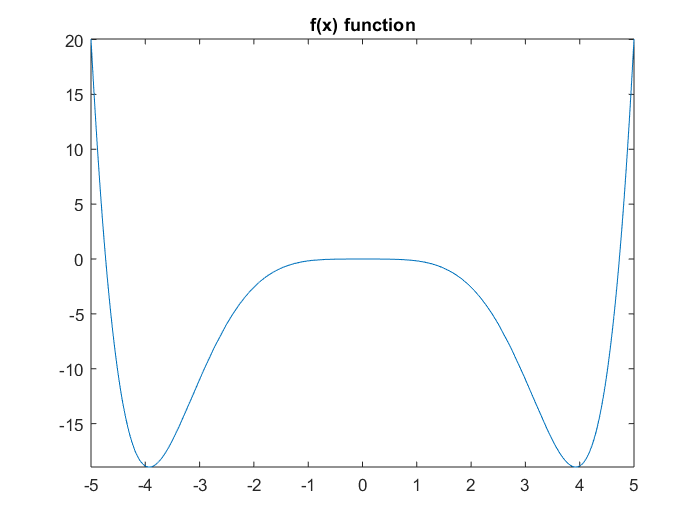
Since the system was not converging the functions needed to be adjusted in order to change which variable was being solved for in each equation. By simply switching the 1st and 3rd row and the 2nd and 4th row, the equations meet the convergence condition.

Using 100 iterations the system of equations converges to:

|  |  |  |  |
| --- | --- | --- | --- |
| **x1** | **x2** | **x3** | **x4** |
| -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.



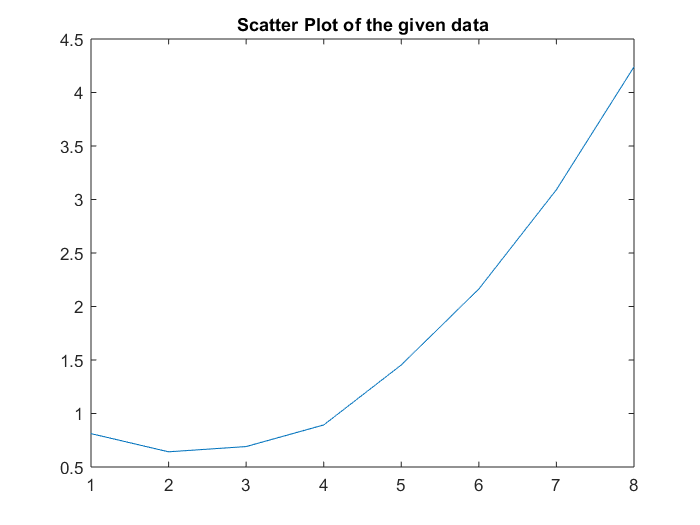
Similarly, to the assignment 2, I look for the places in between points where there is a possible root by checking if the sign changes. I took step sizes of h=0.5 starting from the first root at x=0 and moved throughout the function at by the step size. Then I go through the multiple iterations using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

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.

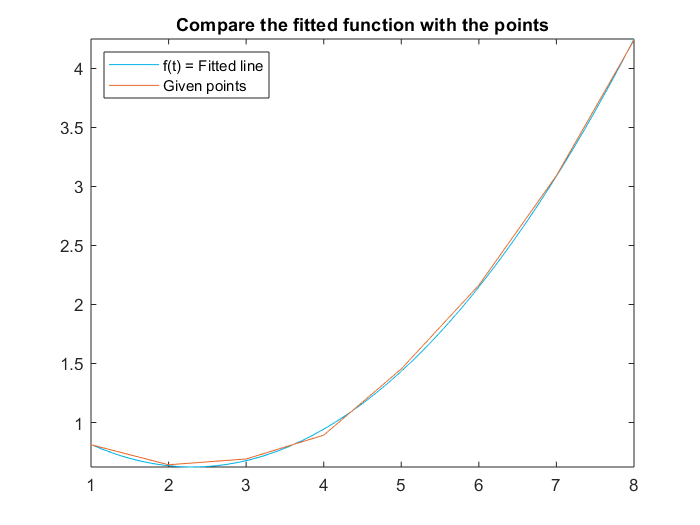
|  |  |
| --- | --- |
| **Relative Error** | 1.786830646479475e-09 |
| **Root (positive x-axis)** | 4.730040736410922 |
| **Root (negative x-axis)** | - 4.730040736410922 |

**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 built in polyfit function from Matlab to fit a second order polynomial to it. Resulting in the combined plot:

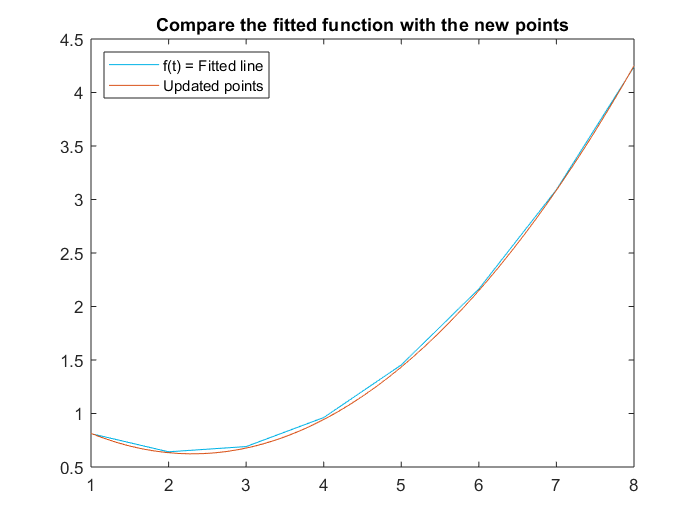


In order to find a point that maybe wrong and out of place, I found the absolute error at each point on the graph. Using the Lagrange Polynomial Interpolation that we used in assignment 3, we can use the similar method to calculate and fix the wrong point. We can find the new point using the interpolation.

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**

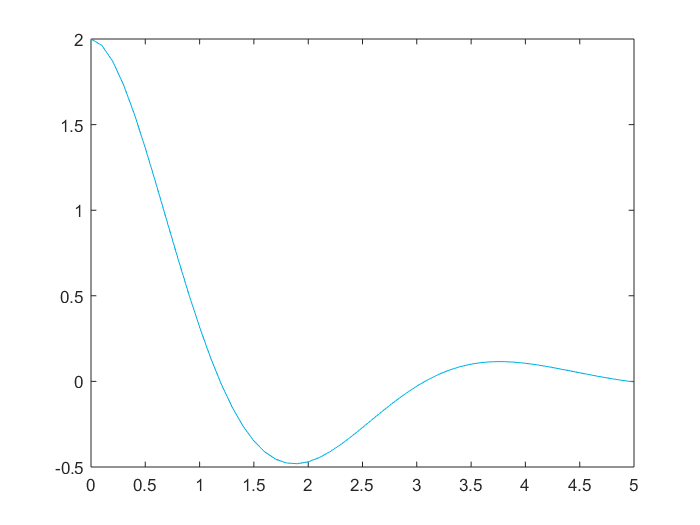
The given equation has the second order derivative, which means that the functions must be transformed so that numerical calculations can be done on it.

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:

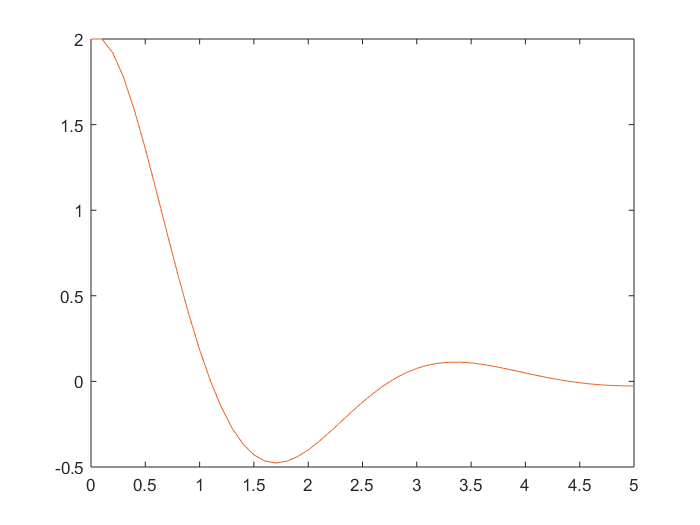
After finding all the coefficients from those functions based on the Taylor series, the coefficients are summed together to calculate the next value of y in the function based off the initial conditions.

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



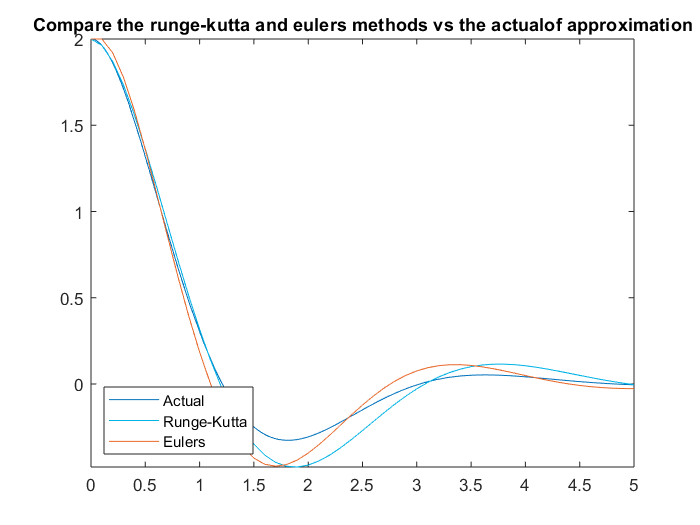
Now to calculate the plot using Euler’s method we use the same functions derived at the beginning named f and g. Additionally, we create the same matrices filled with the initial conditions for the first point. The equations associated with Euler’s method to calculate the points are much shorter compared 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 slightly but 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 | 0.060644375255042 |

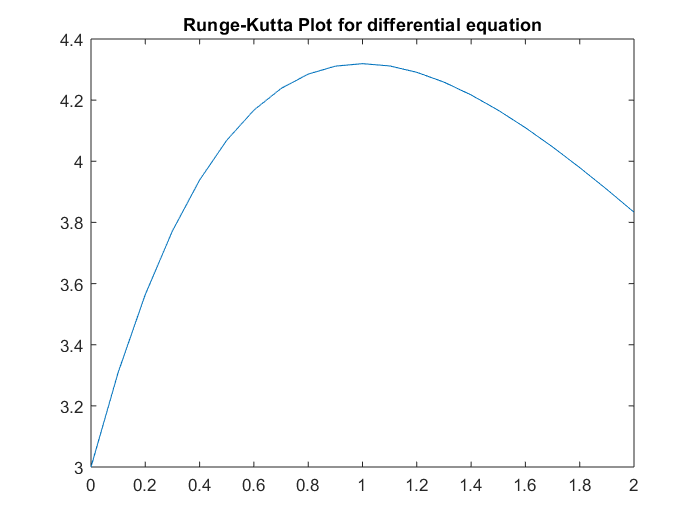


**Question 10**

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.

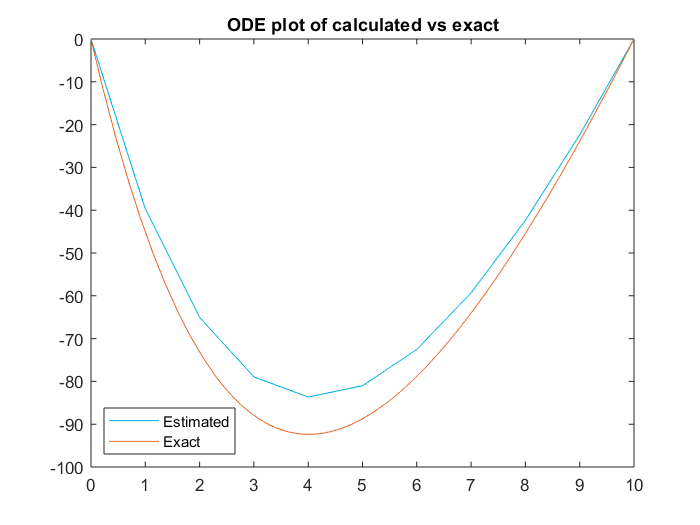


**Question 11**

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 .

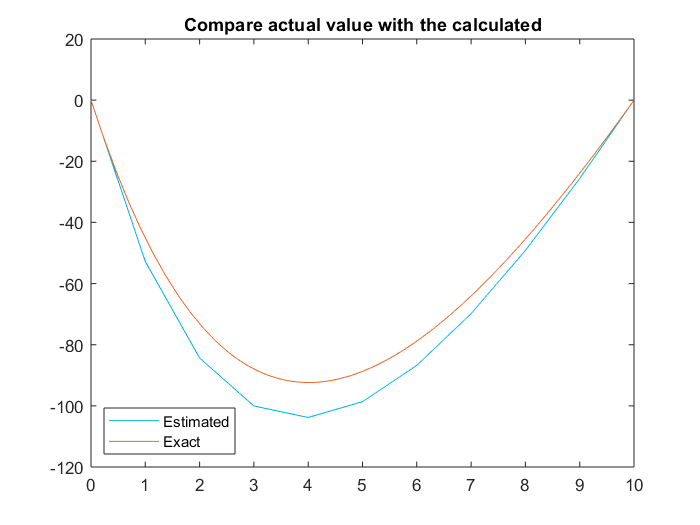
Using this function, we can iterate through the number of coordinates using the boundary conditions that were provided. The function can be plotted vs the actual function and it can be seen that the estimated function provides a close estimate of the real function.



**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 Euler’s 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, but we can assume that the error is accumulated from the Euler’s method that was used. However, if we used a Runge Kutta method with a higher order of precision then it would be possible to calculate better estimate. Additionally, the step size is relatively large.

**Appendix**

**Appendix A – Question 1 Matlab Code**

**Appendix B – Question 2 Matlab Code**

**Appendix C – Question 3 Matlab Code**

**Appendix D – Question 4 Matlab Code**

**Appendix E – Question 5 Matlab Code**

**Appendix F – Question 6 Matlab Code**

**Appendix G – Question 7 Matlab Code**

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

**Appendix I – Question 9 Matlab Code**

**Appendix J – Question 10 Matlab Code**

**Appendix K – Question 11 Matlab Code**

**Appendix L – Question 12 Matlab Code**