Mälardalen University

Numerical Methods with MATLAB

Autumn 2024

Laboratory report

Author(s):
Harry Setiawan Hamjaya
Muhammad Khizar Bin Muzaffar
Somoy SubandhuBarua
Lab Group: 5

Instructors:
Maksat Ashyralyyev
Sergey Korotov

November 13, 2024

1 Solving Nonlinear Algebraic Equation

Consider the following algebraic equation:

$$\tanh(x) = 0$$

where $\tanh(x)$ stands for hyperbolic tangent.

(a) Prove that the given equation has a unique real root at $p = 0$.

(b) Use the Bisection method with the initial interval $[-4, 3]$ to approximate the root of the given equation within an absolute accuracy of 10^{-15}. How many Bisection iterations are required to achieve the approximation within an absolute accuracy of 10^{-15}?

(c) Use the Newton-Raphson method to approximate the root of the given equation within an absolute precision of 10^{-15}. As an initial guess for the Newton-Raphson method, try values $p_0 = 1$ and $p_0 = 1.1$. What do you observe? Explain why it converges/diverges.

(d) Use the Secant method to approximate the root of the given equation within an absolute precision of 10^{-15}. As the initial guesses for the Secant method, try pairs $p_0 = 1, p_1 = 1.2$ and $p_0 = 1, p_1 = 2.4$. What do you observe? Explain why it converges/diverges.

(e) Why does the Bisection method always converge, but the Newton-Raphson and Secant methods sometimes fail to converge?

1.1 Solution & Results

1.1.1 Solution: Part (a)

To prove that the equation $\tanh(x) = 0$ has a unique root, we will follow these steps:

1. **Understanding the function** $\tanh(x)$**:**

   The hyperbolic tangent function is defined as:

   $$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

   We want to find when $\tanh(x) = 0$.

2. **Determine when** $\tanh(x) = 0$**:**

   To find when $\tanh(x) = 0$, set the definition equal to zero:

   $$\frac{\sinh(x)}{\cosh(x)} = 0.$$

This equation is satisfied when $\sinh(x) = 0$, because the denominator $\cosh(x)$ is never zero (since $\cosh(x) = \frac{e^x+e^{-x}}{2} > 0$ for all x).

The function $\sinh(x) = \frac{e^x-e^{-x}}{2}$ equals zero when:

$$e^x - e^{-x} = 0,$$

or equivalently:

$$e^x = e^{-x}.$$

Multiplying both sides by e^x gives:

$$e^{2x} = 1.$$

Taking the natural logarithm of both sides:

$$2x = 0 \quad \Rightarrow \quad x = 0.$$

Thus, $\tanh(x) = 0$ at $x = 0$.

3. **Show the uniqueness of the root:**

   To prove that this is the only root, we examine the behavior of $\tanh(x)$ on the real line. We know the following properties of $\tanh(x)$:

   - $\tanh(x)$ is an odd function, meaning $\tanh(-x) = -\tanh(x)$.
   - As $x \to +\infty$, $\tanh(x) \to 1$, and as $x \to -\infty$, $\tanh(x) \to -1$.

   Additionally, the derivative of $\tanh(x)$ is:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x) = sech^2(x) > 0,$$

   which is always positive for all real x. This shows that $\tanh(x)$ is strictly increasing on the real line.

   Since $\tanh(x)$ is strictly increasing and crosses zero at $x = 0$, it can only have one root, and that root is $x = 0$.

Conclusion: The equation $\tanh(x) = 0$ has a unique root, which is $x = 0$. Figure 1 shows that in the interval, there exists one root for this problem, making it easy to focus on obtaining the solution using the three methods we discussed: Bisection, Newton-Raphson, and Secant methods. All three methods were implemented in Matlab.

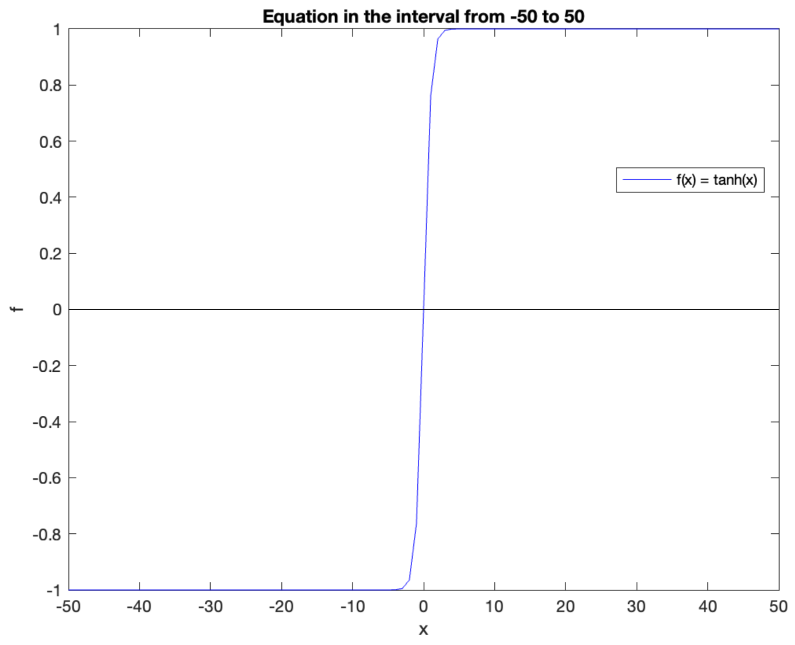

Figure 1: Visualization plot for tanh(x) with interval -4 to 3

1.1.2 Solution: Part (b)

We are tasked with using the Bisection method to approximate the root of the equation $\tanh(x) = 0$ within an absolute accuracy of 10^{-15}, using the initial interval $[-4, 3]$.

The number of iterations required to achieve the desired accuracy can be estimated using the formula:

$$n \geq \log_2 \left(\frac{b - a}{\text{TOL}} \right)$$

where:

- $a = -4$ (left endpoint of the interval),
- $b = 3$ (right endpoint of the interval),
- $\text{TOL} = 10^{-15}$ (the tolerance, or the desired accuracy).

Let's compute each step:
1. Calculate the width of the initial interval:

$$b - a = 3 - (-4) = 7$$

2. Compute the number of iterations required:

$$n \geq \log_2\left(\frac{7}{10^{-15}}\right)$$

Simplifying the expression:

$$n \geq \log_2(7 \times 10^{15}) = \log_2(7) + \log_2(10^{15})$$

3. Now calculate each term:

$$\log_2(7) \approx 2.807$$

$$\log_2(10^{15}) = 15 \times \log_2(10) \approx 15 \times 3.322 = 49.83$$

Thus:

$$n \geq 2.807 + 49.83 = 52.63627$$

Since the number of iterations must be an integer, we round up to the next whole number:

$$n = 53$$

Therefore, we need at least 53 Bisection iterations to approximate the root of $\tanh(x) = 0$ within an absolute accuracy of 10^{-15} specifically 5.5511×10^{-16}.

1.1.3 Solution: Part (c)

The Newton-Raphson method is an iterative numerical method used to approximate the roots of a function. It works by using tangents to the curve of the function. Starting from an initial guess p_0, we find the next approximation by following the tangent line at p_0 to its intersection with the x-axis. The general formula for the Newton-Raphson iteration is:

$$p_{n+1} = p_n - \frac{f(p_n)}{f'(p_n)}, n = 0, 1, 2, ...$$

This process is repeated until the difference between successive approximations is less than the desired tolerance, or until we reach the required accuracy.

In this task, we use the Newton-Raphson method to approximate the root of $\tanh(x) = 0$ within an absolute precision of 10^{-15}. We try two initial guesses: $p_0 = 1$ and $p_0 = 1.1$.

Case 1: $p_0 = 1$

For $p_0 = 1$, the Newton-Raphson method converges to the root after 7 iterations. The root is approximated with an absolute accuracy of 10^{-15}, specifically reaching 0. This indicates the initial $p_0 = 1$ leads to convergence.

Case 2: $p_0 = 1.1$

For $p_0 = 1.1$, however, the Newton-Raphson method fails to converge to the root. This is due to the significant changes in the slope of the tangent near

$x = 1.1$, as seen in Figure 1. The tangent line increases in steepness significantly as we move from $x = 1$ to $x = 1.1$, which leads to divergence in the iteration process.

This divergence is reflected in the results shown in Table 1. After 6 iterations, the value becomes infinity (Inf), and in the next step, it turns into a "Not a Number" (NaN). Therefore, the iteration is stopped.

Table 1: Newton-Raphson Results for $p_0 = 1.1$

Iteration	Estimation	Rounded Absolute Error
0	1.1	1.1
1	-1.12855258526795	1.13
2	1.2341311330391	1.23
3	-1.6951659799228	1.70
4	5.71536010037973	5.72
5	-23021.3564857137	23021.36
6	Inf	Inf
7	NaN	NaN

The reason for this divergence can be seen clearly from Figure 1, which shows how the tangent line changes sharply between $x = 1$ and $x = 1.1$. In such cases, the Newton-Raphson method is sensitive to the choice of the initial guess, and choosing a point where the tangent changes rapidly can lead to non-convergence.

1.1.4 Solution: Part (d)

The Secant Method is an iterative numerical technique used to find the roots of a function. Unlike the Newton-Raphson method, which requires the derivative of the function, the Secant Method approximates the derivative using two initial approximations. This makes it particularly useful when the derivative of the function is difficult to compute.

Starting with two initial guesses, p_0 and p_1, the method generates a sequence of values that converge to a root. Each new approximation, p_{n+1}, is obtained by the formula:

$$p_{n+1} = p_n - f(p_n)\frac{p_n - p_{n-1}}{f(p_n) - f(p_{n-1})}, n = 1, 2, ...$$

The Secant method continues until the difference between successive approximations is less than a specified tolerance level, or the required accuracy is achieved.

In this task, we use the Secant method to approximate the root of $\tanh(x) = 0$ within an absolute precision of 10^{-15}. We try two pairs $p_0 = 1, p_1 = 1.2$ and $p_0 = 1, p_1 = 2.4$.

Case 1: $p_0 = 1, p_1 = 1.2$

For $p_0 = 1, p_1 = 1.2$, the Secant method converges to the root after 6 iterations. The root is approximated with an absolute accuracy of 10^{-15}, specifically reaching $7.48339960419067 \times 10^{-24}$. This indicates the initial $p_0 = 1, p_1 = 1.2$ leads to convergence.

Case 2: $p_0 = 1, p_1 = 2.4$

For $p_0 = 1, p_1 = 2.4$, however, the Secant method fails to converge to the root. This is due to the slope of the line connected points $p_0 = 1, p_1 = 2.4$ becoming steeper, as seen in Table 2. The tangent line deviates significantly as we move from $p_1 = 1.2$ to $p_1 = 2.4$, which leads to divergence in the iteration process.

This divergence is reflected in the results shown in Table 2. It is shown that after the 10^{th} the value becomes infinity (Inf), and in the next step, it turns into a "Not a Number" (NaN). Therefore, the iteration is stopped.

Table 2: Secant Results for $p_0 = 1, p_1 = 2.4$

Iteration	Estimation	Rounded Absolute Error
0	1	1
1	2.4	2.4
2	-3.80110072597107	3.80
3	-0.676581781978972	0.68
4	3.81747567108	3.82
5	0.990780072191173	0.99
6	-7.88375215511537	7.88
7	-2.83479178863368	2.83
8	726.552455785656	726.55
9	360.600893791627	360.60
10	Inf	Inf
11	NaN	NaN

1.2 Conclusions

Overall, the Bisection method took the most iterations to find the root, while both the Secant and Newton-Raphson methods required approximately **nine-fold and eight-fold fewer iterations** respectively. These results confirm the theory that *Bisection* has **linear** convergence, while *Newton-Raphson* exhibits **quadratic** and *Secant* shows nearly **quadratic** convergence. This means that the error decreases at a rate proportional to the square of the previous error, leading to much faster convergence near the root, provided that the initial guess is sufficiently close to the actual root.

By observing the error values, we can also see that the Newton-Raphson method produces no error at all, while the Secant method ranks second in terms of accuracy. Even though the Bisection method takes more steps, it produces a larger error compared to the other two methods.

The reason why the Bisection method always converges, but the Newton-Raphson and Secant methods sometimes fail to converge is because these three numerical methods can be classified into two categories: **Bracketing methods** and **Open methods**. The **Bisection method** falls under the **Bracketing methods**, while both the **Newton-Raphson method** and **Secant method** are considered **Open methods**.

By definition, bracketing methods start with an interval that contains the root, and the procedure is used to reduce this interval while still containing the same root. On the other hand, open methods start with one or several initial guesses and, with each iteration, generate a new guess for the root. Open methods may *fail* to converge if **the initial guesses are chosen poorly**, which is why the choice of initial points is critical for their success. In the experiment, we have shown in some cases both the Newton-Raphson and Secant Methods fail because of wrongly chosen initial points, which leads to the slope in the Newton-Raphson method and the line formed by the points in the Secant method approaching a vertical axis line or in other word it increases in the steepness.

2 Numerical Integration

2.1 Problem Statement

Consider the following integral:

$$I = \int_0^1 e^{-x^2} dx$$

We aim to approximate the integral using two different numerical methods: the Trapezoidal Rule and Simpson's Rule. The exact value of this integral is computed using MATLAB's built-in function `integral()`, which gives us $I = 0.7468$. This exact value will be used as a reference to compute the absolute errors for both numerical methods. The integral is computed for step sizes $h = 2^{-k}, k = 1, 2, \ldots, 15$.

2.2 Solution

Numerical integration is important for approximating the value of definite integrals when an analytical solution is not feasible or too complex to obtain. In this task, we apply the Trapezoidal Rule and Simpson's Rule, which are widely used techniques for such approximations.

2.2.1 Trapezoidal Rule

The Trapezoidal Rule approximates the area under a curve by dividing the interval into small trapezoids and summing up their areas. The formula for the Trapezoidal Rule is:

$$I_{\text{trap}} = \frac{h}{2} \left[f(x_0) + 2 \sum_{k=1}^{n-1} f(x_k) + f(x_n) \right]$$

where h is the step size, x_0 and x_n are the interval boundaries, and $f(x_k)$ represents the function evaluated at the points x_k. The error in the Trapezoidal Rule is of order $O(h^2)$, meaning the error decreases quadratically with the step size.

2.2.2 Simpson's Rule

Simpson's Rule is more accurate than the Trapezoidal Rule and is based on fitting parabolas to segments of the curve. The formula for Simpson's Rule is:

$$I_{\text{simp}} = \frac{h}{3} \left[f(x_0) + 4 \sum_{\text{odd k}} f(x_k) + 2 \sum_{\text{even k}} f(x_k) + f(x_n) \right]$$

Simpson's Rule has an error of order $O(h^4)$, providing significantly higher accuracy as the step size decreases.

2.3 Results and Error Analysis

We calculate the integral using the Trapezoidal Rule and Simpson's Rule for different step sizes $h = 2^{-k}$, where $k = 1, 2, \dots, 15$. The errors for each method are computed by comparing the approximate results with the exact value of $I = 0.7468$. The absolute errors are plotted against the step size h, along with reference lines for h^2 and h^4, as shown in Figure 2.

Table 3: Trapezoidal and Simpson's Rule Results and Errors

h	**I_trap**	**Error_trap**	**I_simp**	**Error_simp**
0.5	0.73137	0.015454	0.74718	0.0003563
0.25	0.74298	0.00384	0.74686	3.1247e-05
0.125	0.74587	0.00095852	0.74683	1.9877e-06
0.0625	0.74658	0.00023954	0.74682	1.2462e-07
0.03125	0.74676	5.9878e-05	0.74682	7.7946e-09
0.015625	0.74681	1.4969e-05	0.74682	4.8725e-10
0.0078125	0.74682	3.7423e-06	0.74682	3.0454e-11
0.0039062	0.74682	9.3557e-07	0.74682	1.9033e-12
0.0019531	0.74682	2.3389e-07	0.74682	1.189e-13
0.0009766	0.74682	5.8473e-08	0.74682	7.2164e-15
0.0004883	0.74682	1.4618e-08	0.74682	4.4409e-16
0.0002441	0.74682	3.6546e-09	0.74682	1.1102e-16
0.0001221	0.74682	9.1364e-10	0.74682	0
6.1035e-05	0.74682	2.2841e-10	0.74682	8.8818e-16
3.0518e-05	0.74682	5.7103e-11	0.74682	5.5511e-16

As shown in Figure 2, the results confirm the theoretical convergence rates. The error for the Trapezoidal Rule follows the reference line for h^2, while the error for Simpson's Rule aligns with the h^4 line.

2.4 Conclusion

The results confirm that the Trapezoidal Rule converges with an error of order $O(h^2)$, while Simpson's Rule converges with an error of order $O(h^4)$, as expected. This is visible in the Figure 2 where both methods follow their respective reference lines for h^2 and h^4.

We can also see that Simpson's Rule consistently provides much lower errors compared to the Trapezoidal Rule for the same step sizes, as shown in Table 3.

Notably, there is an observed increase in the error for Simpson's Rule at very small step sizes. This behavior arises because we are comparing our solution to another approximation, I_{req}, rather than the exact value of the integral. The approximation I_{req} itself contains an error E_1, such that:

$$I_{\text{req}} = I_{\text{exact}} + E_1$$

Similarly, our computed solution using Simpson's Rule can be expressed as:

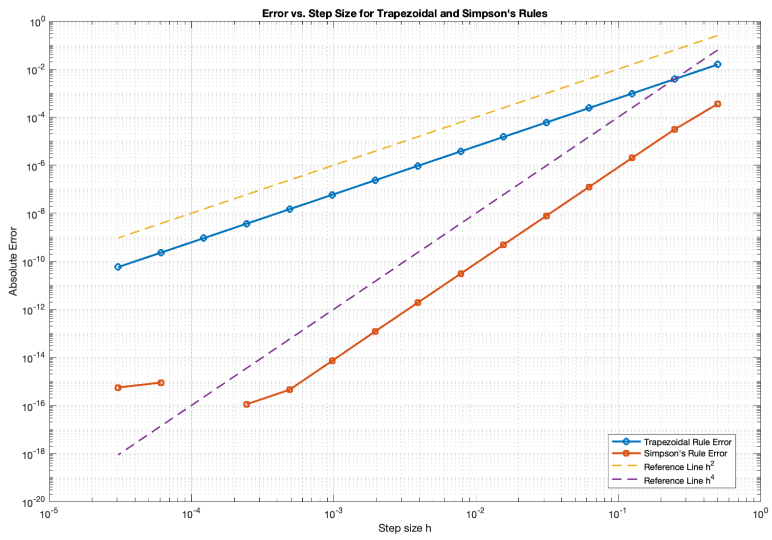

Figure 2: Error vs. Step Size for Trapezoidal and Simpson's Rules

$$I_{\text{simp}} = I_{\text{exact}} + E_2$$

When the errors E_1 and E_2 are combined, they may add up and lead to an increase in the observed error for Simpson's Rule as the step size decreases. This explains why the error for Simpson's Rule does not continue to decrease indefinitely and, instead, starts to increase at very small step sizes.

3 Curve Fitting

It is known that radioactivity decays exponentially with time t, i.e., the intensity of radiation can be described by the following exponential curve:

$$y(t) = \beta e^{-kt}$$

where β and k are constants. The following measurements for the intensity of radiation for a radioactive substance are given in the table:

t_k(years)	y_k
0	1.000
0.5	0.994
1	0.990
1.5	0.985
2	0.979
2.5	0.977
3	0.972
3.5	0.969
4	0.967
4.5	0.960
5	0.956
5.5	0.952

Write a MATLAB script that:

(a) finds the least squares exponential curve for the given data,

(b) computes the Root-Mean-Square (RMS) of the obtained fit,

(c) plots the resulting least squares exponential curve along with the given data points in one frame,

(d) estimates the radioactive half-life of the substance.

3.1 Solution

Least squares curve fitting is commonly used to find the best-fitting curve to a given set of data points by minimizing the sum of the squares of the offsets (the residuals) of the points from the curve. This works well for linear functions, where the relation between the variables can be expressed as a straight line.

However, many functions, such as exponential functions, are not linear, and special techniques are required to handle them. In this task, we applied a technique known as **linearization**. By taking the natural logarithm on both sides of the equation $y(t) = \beta e^{-kt}$, we transform the exponential relation into a linear one. Starting with the original equation:

$$y(t) = \beta e^{-kt}$$

Taking the natural logarithm on both sides:

$$\ln(y) = \ln(\beta e^{-kt}) \tag{1}$$

Using the properties of logarithms, this simplifies to:

$$\ln(y) = \ln(\beta) - kt \tag{2}$$

Letting $Z = \ln(y)$ and $C = \ln(\beta)$, the equation becomes:

$$Z = C - kt \tag{3}$$

This equation is now in a linear form, where Z is analogous to y, C is the intercept, and k is the slope. To find the least squares fit for this linear equation, we use the **normal equations** approach instead of MATLAB's `polyfit` function.

Using Normal Equations The normal equations provide a way to find the best-fitting line by minimizing the sum of squared residuals. Given the linear form $Z = at + b$, where $a = -k$ and $b = \ln(\beta)$, we minimize the sum of squared residuals:

$$S(a, b) = \sum_{i=1}^{n} [Z_i - (at_i + b)]^2$$

Minimizing S with respect to a and b involves setting the partial derivatives of S to zero, resulting in the following normal equations:

$$\begin{cases} nb + \left(\sum_{i=1}^{n} t_i\right) a = \sum_{i=1}^{n} Z_i \\ \left(\sum_{i=1}^{n} t_i\right) b + \left(\sum_{i=1}^{n} t_i^2\right) a = \sum_{i=1}^{n} t_i Z_i \end{cases}$$

The necessary sums to solve these equations are computed as follows: - n: Number of data points. - $\sum t_i$: Sum of all t values. - $\sum Z_i$: Sum of all $Z = \ln(y)$ values. - $\sum t_i^2$: Sum of squares of all t values. - $\sum t_i Z_i$: Sum of the products of t_i and Z_i.

The system of equations is then solved to obtain the values of a and b. From these values, we recover the parameters $k = -a$ and $\beta = e^b$.

Once we have β and k, we can fit this equation to our data points and plot the result. The Root Mean Square (RMS) error of the fit is calculated, and we can estimate the radioactive half-life using the formula:

$$\text{Half-life} = \frac{\ln(2)}{k}$$

3.2 Results

After applying the least squares fit to the data, we obtained the following parameters: $\beta = 0.998416$ and $k = 0.008640$. The final exponential equation that models the intensity of the radioactive decay is:

$$y(t) = 0.998416e^{-0.008640t}$$

The Root Mean Square (RMS) error of the fit was calculated to be 1.136175×10^{-3}, indicating a good approximation of the data by the fitted curve. Additionally, we estimated the radioactive half-life using the equation Half-life $= \frac{\ln(2)}{k}$, which resulted in a half-life of approximately 80.23 years. This suggests that the radioactive substance will decay to half of its original intensity over this period.

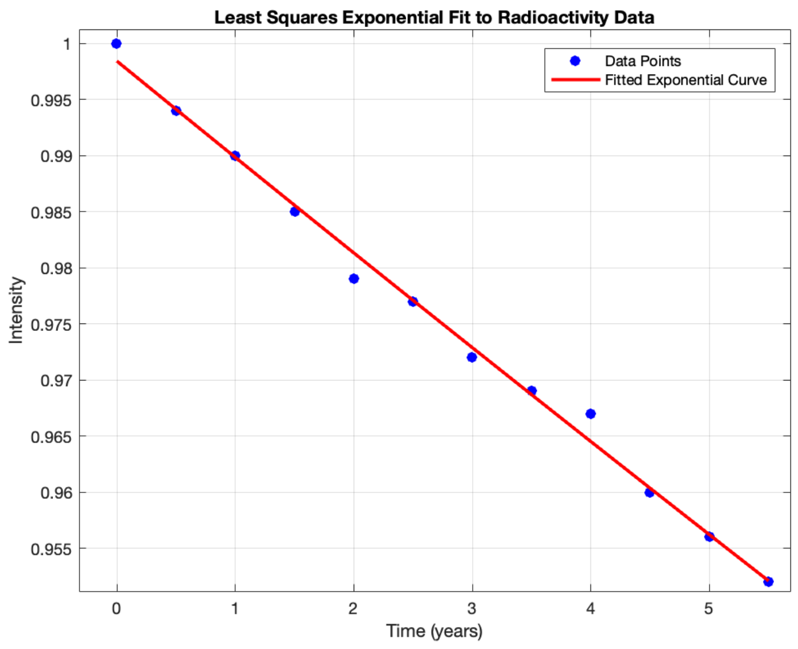

Figure 3: Least Squares Exponential Fit to Radioactivity Data

The plot in Figure 3 shows the least squares exponential curve fitted to the data points, demonstrating the accuracy of the fit.

3.3 Conclusion

Curve fitting is a great tool for understanding relationships between variables, especially when the data follows a pattern like exponential decay. In cases where the function is nonlinear, techniques like linearization can really help by allowing us to use simpler methods like linear regression.

That being said, it's important to remember that curve fitting isn't perfect. We saw this in the RMS error and the slight differences between the fitted curve and the actual data points. While the fit does a good job overall, there will always be some error in these kinds of approximations, as reflected in the graph and the calculated RMS value.

4 Numerical Differentiation

We have to consider the following boundary value problem (BVP):

$$\begin{cases} y''(x) + \frac{1}{4}y(x) = f(x), & 0 < x < \pi, \\ y(0) = y(\pi) = 0, \end{cases}$$

where f is some given function. We want to approximate the solution of this BVP at discrete points $x_m = mh$, $m = 1, 2, \ldots, N-1$, where $h = \frac{\pi}{N}$ and N is some positive integer. Let y_m denote the numerical approximation of $y(x_m)$. Discretizing the differential equation at the points x_m using the second-order central difference formula for y'' results in the system of $N-1$ algebraic equations:

$$\frac{y_{m-1} - 2y_m + y_{m+1}}{h^2} + \frac{1}{4}y_m = f(x_m), \quad m = 1, 2, \ldots, N-1,$$

where $y_0 = y_N = 0$ due to the given zero boundary conditions.

- (a) The aim is to write a MATLAB script that, for a given function $f(x)$ and positive integer N, solves the linear system using LU factorization and computes the error in the obtained numerical solution as follows:

$$E = \max_{1 \le m \le N-1} |y_m - y(x_m)|.$$

- (b) We need to consider BVP with $f(x) = 3 - \frac{6x}{\pi}$. Solve it numerically using your script with $N = N_k = 5 \times 2^k$, $k = 1, 2, \ldots, 10$. The exact solution is:

$$y(x) = 12\left(1 - \cos\left(\frac{x}{2}\right) + \sin\left(\frac{x}{2}\right) - \frac{2x}{\pi}\right).$$

  We plot the errors E_k for $k = 1, 2, \ldots, 10$ against the step sizes $h_k = \frac{\pi}{N_k}$ in log-log scale. Write down our observation Does it confirm the theory for order of convergence?

4.1 Solution & Results

Given the boundary value problem,

$$\begin{cases} y''(x) + \frac{1}{4}y(x) = f(x), \quad 0 < x < \pi, \quad \text{(i)} \\ y(0) = y(\pi) = 0, \quad \text{(ii)} \end{cases}$$

In equation (i), we can simplify the equation utilizing the central difference theorem.

The second derivative $y''(x)$ at point $x = x_m$ can be approximated using the central difference formula:

$$y''(x_m) \approx \frac{y_{m-1} - 2y_m + y_{m+1}}{h^2}$$

where h is the spacing between points, and $y_m \approx y(x_m)$.

Substituting this approximation for $y''(x_m)$ into equation (i):

$$\frac{y_{m-1} - 2y_m + y_{m+1}}{h^2} + \frac{1}{4}y_m = f(x_m)$$

Hence, the resulting finite difference formula is:

$$\frac{y_{m-1} - 2y_m + y_{m+1}}{h^2} + \frac{1}{4}y_m = f(x_m), \quad m = 1, 2, \ldots, N-1.$$

- Which is the equation (2) we have in our question.
Simplifying this further we get,

$$\left(\frac{1}{h^2}\right) y_{m-1} + \left(\frac{-2}{h^2} + \frac{1}{4}\right) y_m + \left(\frac{1}{h^2}\right) y_{m+1} = f(x_m) \text{...(iii)}$$

Point to note that, equation (iii) is similar to the Tridiagonal system of equations for n unknowns,

i.e. A Tridiagonal system for n unknowns may be written as

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i, \text{...(iv)}$$

where $a_1 = 0$ and $c_n = 0$.

In matrix form, this system can be expressed as:

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix}.$$

Such system of equations can be solved using the *Tridiagonal Matrix Algorithm*, also known as *Thomas Algorithm.*

Given equations (iii) and (iv), the coefficients a_i, b_i, and c_i are as follows:

$$a_i = \frac{1}{h^2}, \quad b_i = \frac{-2}{h^2} + \frac{1}{4}, \quad c_i = \frac{1}{h^2}.$$

Expressing this into the matrix form we get,

$$\begin{bmatrix} \frac{-2}{h^2}+\frac{1}{4} & \frac{1}{h^2} & & & 0 \\ \frac{1}{h^2} & \frac{-2}{h^2}+\frac{1}{4} & \frac{1}{h^2} & & \\ & \frac{1}{h^2} & \frac{-2}{h^2}+\frac{1}{4} & \ddots & \\ & & \ddots & \ddots & \frac{1}{h^2} \\ 0 & & & \frac{1}{h^2} & \frac{-2}{h^2}+\frac{1}{4} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{N-1} \end{bmatrix} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_{N-1}) \end{bmatrix}.$$

Therefore, we can solve the systems of equations utilizing the tridiagonal matrix algorithm. First we create the tridiagonal matrix as shown. Then, to solve this system efficiently, we employ LU decomposition with partial pivoting (A = LU),

For our tridiagonal matrix A, we seek to decompose it as $A = LU$ where:

- L is a lower triangular matrix with ones on its diagonal,
- U is an upper triangular matrix.

For our tridiagonal matrix, the LU decomposition has a simple structure:

$$\begin{bmatrix} \frac{-2}{h^2}+\frac{1}{4} & \frac{1}{h^2} & & & 0 \\ \frac{1}{h^2} & \frac{-2}{h^2}+\frac{1}{4} & \frac{1}{h^2} & & \\ & \frac{1}{h^2} & \frac{-2}{h^2}+\frac{1}{4} & \ddots & \\ & & \ddots & \ddots & \frac{1}{h^2} \\ 0 & & & \frac{1}{h^2} & \frac{-2}{h^2}+\frac{1}{4} \end{bmatrix} = \begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ & l_{32} & 1 & \\ & & \ddots & \ddots \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & & \\ & u_{22} & u_{23} & \\ & & u_{33} & \ddots \\ & & & \ddots \end{bmatrix}$$

To process this in MATLAB, we fill up our tridiagonal matrix, A by utilizing the in-built MATLAB function ‘diag()‘.

```
% Diagonal Filling
    A = diag(main_diagonal * ones(N-1, 1))+ ...
        diag(off_diagonal * ones(N-2, 1), 1) + ...
        diag(off_diagonal * ones(N-2, 1), -1);
```

The decomposition process PA = LU includes:

1. The matrix A is decomposed into PA = LU where:
    - P is a permutation matrix that optimizes numerical stability
    - L is a lower triangular matrix with ones on the diagonal
    - U is an upper triangular matrix
2. The permutation matrix P:
    - Implements partial pivoting to improve numerical stability

- In our case, for the tridiagonal matrix, the process of doing PA = LU reduces some additional steps, which would have been taken if we directly used A = LU instead due to a trigger condition in MATLAB's library (according to documentation).

This process is efficiently implemented in MATLAB using the command [L,U,P] = lu(A), which automatically handles the decomposition with partial pivoting. To solve Ay = b, with the LU decomposition PA = LU, we have:

1. PAy = Pb
2. Therefore, LUy = Pb
3. First, solve Lz = Pb by forward substitution:
$$z_i = (Pb)_i - \sum_{j=1}^{i-1} l_{ij} z_j$$
4. Then, solve Uy = z by backward substitution:
$$y_i = \frac{1}{u_{ii}} \left(z_i - \sum_{j=i+1}^{n} u_{ij} y_j \right)$$

Fig 4, presents the numerical results showing the relationship between (N_k), step size (h_k), and the maximum error (E_k) for $k = 1, 2, \ldots, 10$. Starting with $N_1 = 10$, we double the number of points at each step ($N_k = 5 \times 2^k$), consequently halving the step size $h_k = \pi/N_k$. The errors E_k demonstrate the expected second-order convergence, decreasing by approximately a factor of 4 with each halving of the step size. For instance, when $k = 1$, with 10 points, the error is 1.1084×10^{-3}, and by $k = 10$, with $N_{10} = 5120$, the error has reduced to 4.2336×10^{-9}, showing a significant improvement in accuracy with smaller step size (h).

In the Figure 4, we visualize the data from our previous table, specially the errors E_k vs step size h in log scale. We additionally plot a h_k^2 line in the plot to show that it's converging in similar rate.
We also obtain the observed order of convergence by utilizing the `polyfit` function in MATLAB, and get the observed order of 2.0

```
% Observed order of convergence
p = polyfit(log(h_k), log(E_k), 1);
fprintf('Observed order of convergence: %.2f\n', p(1));
```

The error values demonstrate clear second-order convergence. When the step size h_k is halved (N_k doubled), the error E_k decreases by approximately a factor of 4, characteristic of second-order convergence. This can be verified by computing the ratio of consecutive errors:

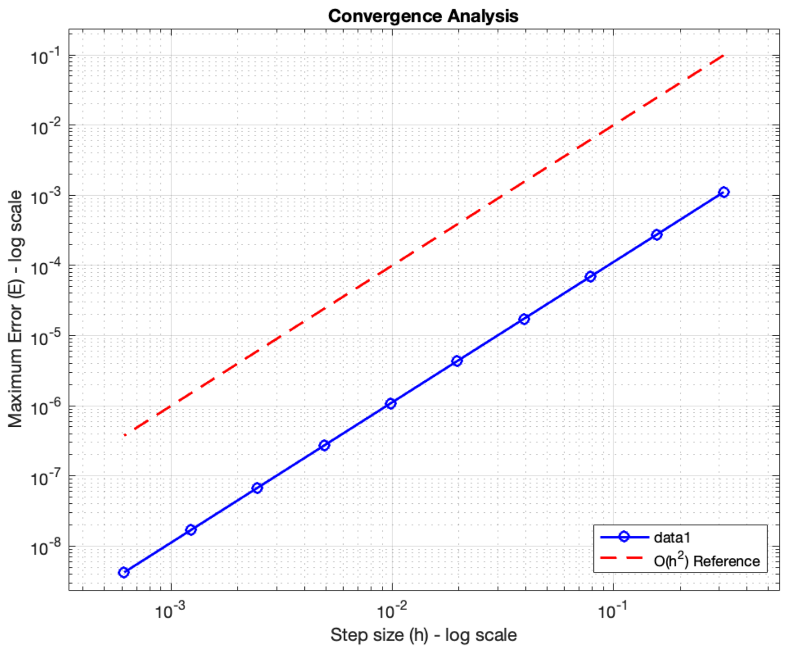

Figure 4: Error vs Step Size Plot demonstrating second-order convergence.

$$\frac{E_k}{E_{k+1}} \approx 4$$

For example, from our data:

$$\frac{1.1084 \times 10^{-3}}{2.7640 \times 10^{-4}} \approx 4.01$$

4.2 Conclusions

The error values observable in the Figure 4 demonstrate clear second-order convergence. When we refer to second-order convergence, we mean:

$$E_k \approx Ch_k^2$$

where C is a constant independent of the step size h_k.

This second-order convergence manifests in two ways:

1. The slope of the log-log plot of error versus step size is approximately 2 (our observed value is 2.00).

2. When the step size h_k is halved (i.e., N_k is doubled), the error E_k decreases by approximately a factor of 4, because:

$$\frac{E_k}{E_{k+1}} \approx \frac{Ch_k^2}{C\left(\frac{h_k}{2}\right)^2} = \frac{Ch_k^2}{\frac{Ch_k^2}{4}} = 4$$

For example, from our data:

$$\frac{1.1084e-03}{2.7640e-04} \approx 4.01$$

We should note that, this factor of 4 reduction in error does not mean fourth-order convergence; rather, it's a characteristic of second-order convergence when the step size is halved. The order of convergence (2.00) is determined by the slope of the log-log plot of errors versus step sizes.

This confirms that our numerical method achieves the expected second-order convergence rate as predicted by the theoretical analysis.

5 Numerical Solution of Differential Equation

Consider the following initial value problem (IVP)

$$\begin{cases} \frac{dy}{dt} = \frac{y(150-y)}{2500}, & t > 0, \\ y(0) = 10, \end{cases}$$

The exact solution to this IVP is

$$y(t) = \frac{150}{1 + 14e^{-0.06t}}, \quad t \geq 0.$$

(a) Use Matlab ODE solver `ode45` to calculate the numerical solution of the given IVP on the interval $[0, 100]$. Plot the numerical solution of $y(t)$ against time. Use the exact solution to find the error of the numerical solution in the maximum norm.

(b) Write the script implementing the third-order Runge-Kutta method (RK3) given by the following Butcher table:

$$\begin{array}{c|ccc} 0 & & & \\ 1 & 1 & & \\ 1/2 & 1/4 & 1/4 & \\ \hline & 1/6 & 1/6 & 2/3 \end{array}$$

Use $RK3$ method with stepsizes $\tau = \tau_k = 2^{-k}, \quad k = 1, 2, 3, 4, 5$, to find the numerical solution of the given IVP on the interval $[0, 100]$. Use the exact solution to find errors of the numerical solutions in the maximum norm for each stepsize $\tau_k = 2^{-k}, k = 1, 2, 3, 4, 5$, and then plot the errors against stepsizes in a log-log scale. What do you observe from the plots? Does this confirm the theory for the order of convergence?

5.1 Solution

An Ordinary Differential Equation (ODE) involves the derivative of an unknown function with respect to one independent variable. When coupled with initial conditions, it forms an Initial Value Problem (IVP). Analytical solutions to ODEs are not always feasible, making numerical methods essential for obtaining solutions.

Among the various numerical methods, the Runge-Kutta family is particularly prominent for solving IVPs. This includes methods like Euler, Heun, and the Trapezoidal method, which can be viewed as specific cases of the general Runge-Kutta approach. The classical third-order Runge-Kutta method, often summarized in a Butcher table, is widely used due to its balance of accuracy and computational efficiency.

$$\begin{array}{c|ccc} 0 & & & \\ c_1 & a_{11} & & \\ c_2 & a_{21} & a_{22} & \\ \hline & b_1 & b_2 & b_3 \end{array}$$

This creates the equation needed for the Runge–Kutta method, which will follow the next pattern, having 3 different values that need to be obtained before obtaining the value of y in a precise step.

$$\begin{aligned}
k_1 &= f(t_{n-1}, y_{n-1})\\
k_2 &= f(t_{n-1} + c_1\tau, y_{n-1} + \tau(a_{11}k_1))\\
k_3 &= f(t_{n-1} + c_2\tau, y_{n-1} + \tau(a_{21}k_1 + a_{22}k_2))\\
y_n &= y_{n-1} + \tau(b_1k_1 + b_2k_2 + b_3k_3)
\end{aligned}$$

Based on our given values, then we will have following equations:

$$\begin{aligned}
k_1 &= f\left(t_{n-1}, y_{n-1}\right)\\
k_2 &= f\left(t_{n-1} + 1 * \tau, y_{n-1} + \tau\left(1 * k_1\right)\right)\\
k_3 &= f\left(t_{n-1} + \frac{1}{2} * \tau, y_{n-1} + \tau\left(\frac{1}{4} * k_1 + \frac{1}{4} * k_2\right)\right)\\
y_n &= y_{n-1} + \tau\left(\frac{1}{6} * k_1 + \frac{1}{6} * k_2 + \frac{2}{3} * k_3\right)
\end{aligned}$$

To implement the solution, the code will repeatedly apply the equations in a loop to calculate the value for the chosen τ. For each specified τ within the interval [0, 100], a set of points will be generated and stored in a vector. The Euclidean norm of this vector can then be calculated, allowing for a comparison of this value with others based on the errors relative to the exact solution.

5.2 Results

The first step is to understand the initial value problem (IVP). For this reason, we use the Matlab ODE solver to calculate the numerical solution on the interval [0,100]. By using both the question equation $\frac{dy}{dt} = y' = \frac{y(150-y)}{2500}$ that we can use to enter it to the ODE45 solver in Matlab.

In figure 5, we can observe the value of $y(t)$ against time obtained in the ODE solver, letting us observe what the pattern of the solution is.

To further understand the difference between the exact solution and the numerical solution given by Matlab, we obtained the error using the infinity norm (also known as the maximum norm) between the absolute difference of the obtained values given by Matlab and the exact solution values. This gives the error results.

$$error = 7.3188 \times 10^{-4}$$

Following the implementation of the Runge-Kutta method in MATLAB, we proceeded to compute the values of y for the initial value problem. This required defining the coefficients from the Butcher tableau and specifying the initial

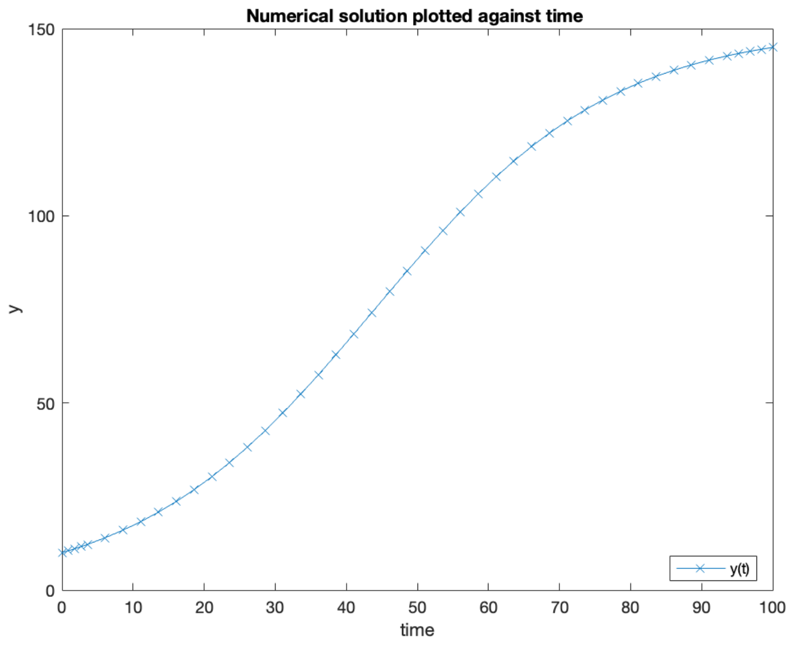

Figure 5: ODE45 Visualization Plot

conditions. We then chose five different step sizes for τ: $2^{-1}, 2^{-2}, 2^{-3}, , 2^{-4}$, and 2^{-5}.

After executing the method for each step size, we calculated the error between the computed values and the exact solution using the infinity norm, allowing us to assess the overall accuracy of the numerical method. The errors were then plotted against the corresponding step sizes τ, as illustrated in Figure 6. Additionally, we included a representation of τ^3 for each step in the graph.

The code used for this implementation can be found in the appendix, where we maintained the same step sizes across the interval $[0, 100]$. The comparison between the exact solution and the computed solution using RK3 is displayed in the figure 6 titled "Runge-Kutta Method Error per stepsize (stepsize = τ)" The graph indicates a minimal difference, a trend that holds across all five values of τ. However, the graphs are similar enough that presenting each individually would not yield significant new insights.

In the log-log plot labeled "Errors for Each τ against h^3," we observe how the errors correlate with the step sizes. An additional line plotting h^3 illustrates the convergence order. The results indicate that even with the largest τ of 2^{-1}, the numerical solution approximates the exact solution reasonably well, maintaining an acceptable error magnitude consistent with cubic error expectations. As anticipated, the errors decrease with smaller τ, showing a significant reduction from 2^{-2} to 2^{-5}, which corresponds to a nearly 512-fold improvement or $(2^3)^3$ which align the cubic magnitude.

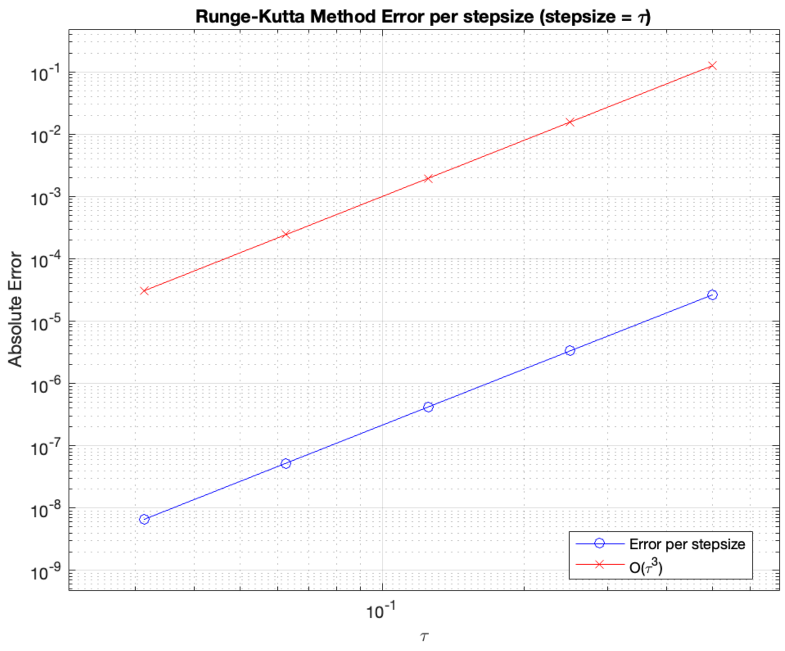

Figure 6: Runge-Kutta Error Plot

5.3 Conclusions

In the case of the differential equation provided in this problem, the smallest τ recommended $\tau = 2^{-5}$ was properly selected to minimize the error obtained. The order of convergence provided by other values of τ is correct, given that their magnitude error follows a similar plot as the specific selected value in its cubic form and therefore is behaving in an $O(\tau^3)$ way and thus giving the third order method of Runge-Kutta sense in its order of computation and its name. These results conclude that the theory of order of convergence is confirmed as the Runge-Kutta method with the third order.

6 References

1. T. Sauer, *Numerical Analysis*, 2nd ed., Harlow, Essex: Pearson, 2014, pp. ii, 608, ISBN: 9781292023588.

2. Documentation - MATLAB & Simulink. Available at: https://www.mathworks.com/help/index.html (Accessed: 20 October 2024).

3. Tridiagonal matrix algorithm (2024) Wikipedia. Available at: https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm (Accessed: 20 October 2024).

4. Finite difference method (2024) Wikipedia. Available at: https://en.wikipedia.org/wiki/Finite_difference_method (Accessed: 20 October 2024).

7 Appendix

You have to include all your MATLAB functions and scripts here. Do not include MATLAB codes as images from a print screen or similar. Enter your code using the syntax shown below.

MATLAB codes for Solving Nonlinear Algebraic Equation

Problem 1 Bisection Method, Newton-Raphson Method, and Secant Method

```
% Matlab script for solving a nonlinear algebraic equation problem by
    visualization plot
function Problem1()
   interval = -50:50;
   f=@(x) tanh(x);
   % (a) Visualizing possible solution
   fh = f(interval);
   plot(interval, fh, 'blue')
   hold on
   plot(interval, zeros(101), 'black')
   title('Equation in the interval from -50 to 50')
   xlabel('x')
   ylabel('f')
   legend('f(x) = tanh(x)')
   legend('Location', 'best');
   saveas(gcf, 'question1.png')
end
```

```
% Matlab script for solving a nonlinear algebraic equation problem with
    bisection method
function [p, n] = bisection(f,a,b,TOL,MaxNiter)
   fa=f(a); fb=f(b);
   for n=1:MaxNiter
      p=(a+b)/2;
       if (b-a)/2<TOL
         break
      end
      fp=f(p);
      if fa*fp<0
         b=p; fb=fp;
      else
         a=p; fa=fp;
      end
   end
end

clear all; clc;
f=@(x) tanh(x);
```

```
a=-4; b=3; Tol=1e-15; MaxIter=100;
[error,number_iteration]=bisection(f,a,b,Tol,MaxIter)
```

```
% Matlab script for solving a nonlinear algebraic equation problem with
    newton-raphson method
function [p, n, x] = newton(f,df,p0,TOL,MaxNiter)
    p=p0;
    x=[p];
    for n=1:MaxNiter
        fp=f(p);
        dfp=df(p);
        dp=fp/dfp;
        p=p-dp;
        x(end+1) = p;
        if abs(dp)<TOL
            break
        end
    end
end

clear all; clc;
f=@(x) tanh(x);
df=@(x) 1-(tanh(x))^2;
Tol=1e-15; MaxIter=7;
% p0=1
p0=1.1
[Error,Iteration,P_val]=newton(f,df,p0,Tol,MaxIter)
```

```
% Matlab script for solving a nonlinear algebraic equation problem with
    secant method
function [p, n, x] = secant(f,a, b,TOL,MaxNiter)
    pa=a;
    pb=b;
    xa=f(pa);
    pvalue=0;
    x = [pa, pb]
    for n=1:MaxNiter
        xb=f(pb);
        p=pb - (pb-pa)*f(pb)/(f(pb)-f(pa));
        x(end+1) = p;
        if abs(p - pvalue)<TOL
            break
        end
        pa = pb;
        pb = p;
        end
    end
clear all; clc;
```

```
f=@(x) tanh(x);
Tol=1e-15; MaxIter=10;
p0=1.0
p1=1.2
% p1=2.4
[Error,Iteration,P_val]=secant(f,p0,p1,Tol,MaxIter)
```

MATLAB codes for Numerical Integration

```
% Function for Trapezoidal Rule
function I_trap = trapezoidal_rule(f, a, b, h)
    x = a:h:b;
    y = f(x);
    I_trap = (h / 2) * (y(1) + 2 * sum(y(2:end-1)) + y(end));
end

% Function for Simpson's Rule
function I_simp = simpson_rule(f, a, b, h)
    n = (b - a) / h;

    x = a:h:b;
    if mod(n, 2) ~= 0
        error('Number of subintervals n must be even for Simpson''s
            rule.');
    end
    y = f(x);

    I_simp = (h / 3) * (y(1) + ...
             4 * sum(y(2:2:end-1)) + ...
             2 * sum(y(3:2:end-2)) + y(end));
end

% Main Script
f = @(x) exp(-x.^2);
a = 0;
b = 1;
I_exact = integral(f, a, b);

K = 1:15;
h_values = 2.^(-K);
errors_trap = zeros(size(K));
errors_simp = zeros(size(K));
I_trap_values = zeros(size(K));
I_simp_values = zeros(size(K));

for idx = 1:length(K)
```

```
    h = h_values(idx);

    I_trap = trapezoidal_rule(f, a, b, h);
    I_trap_values(idx) = I_trap;
    errors_trap(idx) = abs(I_trap - I_exact);

    I_simp = simpson_rule(f, a, b, h);
    I_simp_values(idx) = I_simp;
    errors_simp(idx) = abs(I_simp - I_exact);
end



figure;
loglog(h_values, errors_trap, 'o-', 'LineWidth', 2);
hold on;
loglog(h_values, errors_simp, 's-', 'LineWidth', 2);

loglog(h_values, h_values.^2, '--', 'LineWidth', 1.5);
loglog(h_values, h_values.^4, '--', 'LineWidth', 1.5);

legend('Trapezoidal Rule Error', 'Simpson''s Rule Error', ...
       'Reference Line h^2', 'Reference Line h^4', 'Location', 'best');

xlabel('Step size h');
ylabel('Absolute Error');
title('Error vs. Step Size for Trapezoidal and Simpson''s Rules');
legend('Trapezoidal Rule Error', 'Simpson''s Rule Error', ...
       'Reference Line h^2', 'Reference Line h^4', 'Location', 'best');
hold off;
grid on;

ResultsTable = table(h_values', I_trap_values', errors_trap',
    I_simp_values', errors_simp', ...
    'VariableNames', {'h', 'I_trap', 'Error_trap', 'I_simp',
        'Error_simp'});

disp(ResultsTable);
```

MATLAB codes for Curve Fitting

```
% Given data
t = [0; 0.5; 1; 1.5; 2; 2.5; 3; 3.5; 4; 4.5; 5; 5.5];
y = [1.000; 0.994; 0.990; 0.985; 0.979; 0.977; 0.972; 0.969; 0.967;
    0.960; 0.956; 0.952];

% Part (a): Use Linearization and Manual Normal Equations
```

```
Z = log(y);

% Compute sums needed for the normal equations
n = length(t);
sum_t = sum(t);
sum_z = sum(Z);
sum_tz = sum(t .* Z);
sum_t2 = sum(t .^2);

% Set up the normal equations
% [n      sum_t ] [b] = [sum_z ]
% [sum_t sum_t2] [a] = [sum_tz]

% Create matrices
A = [n, sum_t; sum_t, sum_t2];
b_vec = [sum_z; sum_tz];

% Solve for coefficients
coefficients = A \ b_vec;
b = coefficients(1);
a = coefficients(2);

% Compute k and beta
k = -a; % since Z = a * t + b, and Z = log(y) = -k * t + log(beta)
beta = exp(b);

fprintf('Estimated parameters using manual normal equations:\n');
fprintf('Beta = %.6f\n', beta);
fprintf('k = %.6f\n', k);

% Part (b): Compute the RMS error
y_fitted = beta * exp(-k * t);
residuals = y - y_fitted;
RMS_error = sqrt(mean(residuals.^2));
fprintf('RMS error of the fit: %.6e\n', RMS_error);

% Part (c): Plot the data and the fitted curve
t_fit = linspace(min(t), max(t), 100);
y_fit = beta * exp(-k * t_fit);

figure;
plot(t, y, 'bo', 'MarkerFaceColor', 'b', 'DisplayName', 'Data Points');
hold on;
plot(t_fit, y_fit, 'r-', 'LineWidth', 2, 'DisplayName', 'Fitted
    Exponential Curve');
hold off;
xlabel('Time (years)');
ylabel('Intensity');
title('Least Squares Exponential Fit to Radioactivity Data');
legend('Location', 'best');
```

```
grid on;

% Part (d): Estimate the radioactive half-life
half_life = log(2) / k;
fprintf('Estimated radioactive half-life: %.6f years\n', half_life);
```

MATLAB codes for Numerical Differentiation

```
% MATLAB function script for Boundary Value Problem - BVP
function [x, y, E] = BVP(f, N, y_exact)
   % Solves the boundary value problem(with Tridiagonal Matrix
       Algorithm):
   % Ref: https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm
   %
   % Inputs:
   %   f   - function f(x)
   %   N   - number of discrete points
   %   y_exact - function of exact values of y, for error calculation

   h = pi/N;
   x = (1:N-1) * h;

   % Building Triangular Matrix

   A = zeros(N-1, N-1);


   main_diagonal = -2/h^2 + 1/4;
   off_diagonal = 1/h^2;

   % Diagonal Filling
   A = diag(main_diagonal * ones(N-1, 1))+ ...
       diag(off_diagonal * ones(N-2, 1), 1) + ...
       diag(off_diagonal * ones(N-2, 1), -1);

   % Evaluating f(x) on storing it
   % A.x = b is A.y = Fx here
   b = f(x)';

   [L, U, P] = lu(A);
   y_temp = L\(P * b);
   y = U\y_temp;

   errors = abs(y - y_exact(x)');
   E = max(errors);
end
```

```
% MATLAB script for executing BVP for the given problem in P.4 section
    (b)
format long

k = 1:10;
N_k = 5 * 2.^k; % N_k = 5*2^k for k = 1,2,...,10
h_k = pi./N_k;
E_k = zeros(1, 10);

y0 = 0;
yN = 0;

fx = @(x) 3 - (6*x/ pi);

% Exact Solution
y_exact = @(x) 12*(1-cos(x/2) + sin(x/2) - 2*x/pi);

for i = 1:10
    [x, y, E] = BVP(fx, N_k(i), y_exact);
    E_k(i) = E;
    fprintf('Completed k=%d with error=%e\n', i, E);
end

figure;
loglog(h_k, E_k, 'bo-', 'LineWidth', 1.5);
hold on;
ref_line = h_k.^2;
loglog(h_k, ref_line, 'r--', 'LineWidth', 1.5, 'DisplayName', 'O(h^2)
    Reference');


grid on;
title('Convergence Analysis');
xlabel('Step size (h) - log scale');
ylabel('Maximum Error (E) - log scale');
legend('Location', 'best');

% observed order of convergence
p = polyfit(log(h_k), log(E_k), 1);
fprintf('Observed order of convergence: %.2f\n', p(1));
```

MATLAB codes for Numerical Solution of Differential Equation

```
t_interval = [0,100];
initial = 10;
```

```
%Calculation of numerical solution ODE45
[t,y_ode] = ode45(@eqns , t_interval , initial);
f_correct = @(x) (150./(1 + 14*exp(-0.06*x)));
fh = f_correct(t);
%Plotting of numerical solution ODE45
figure
plot(t,y_ode,'x-')
title('Numerical solution plotted against time ')
xlabel('time')
ylabel('y')
legend('y(t)')
legend('Location', 'best');
%Error of numerical solution ODE45
error_1= abs(fh-y_ode);
error_ode = norm(error_1,inf)
%Initial values of Runge-Kutta method
tao_interval = [1,2,3,4,5];
tao_f = @(n) 2.^-n;
tao = tao_f(tao_interval);
errors = [0,0,0,0,0];
%Runge-Kutta method
for i=1:(length(tao))
    x = 0:tao(i):100;
    y = zeros(1,length(x));
    f = @(t,y) y*(150-y)/2500;
    y(1) = 10;
    for j=1:(length(x)-1)
       k1 = f(x(j),y(j));
       k2 = f(x(j)+1*tao(i),y(j)+tao(i)*1*k1);
       k3 = f((x(j)+(0.5)*tao(i)),(y(j)+tao(i)*0.25*k1+tao(i)*0.25*k2));
       y(j+1) = y(j) + tao(i)*(1/6*k1+1/6*k2+2/3*k3);
    end
    fh = f_correct(x);
    error= abs(fh-y);
    n = norm(error,inf);
    errors(i) = n;
end


%Plotting error of Runge-Kutta method
figure
loglog(tao,errors,'-ob');
hold on
grid on
title('Runge-Kutta Method Error per stepsize (stepsize = {\tau})')
xlabel('{\tau}')
ylabel('Absolute Error')
loglog(tao,tao.^3,'-xr');
legend('Error per stepsize','{\tau}^3')
legend('Location', 'best');
```

```
%Function for IVP
function derivative_y = eqns(t,y)
derivative_y=y*(150-y)/2500;
end
```
