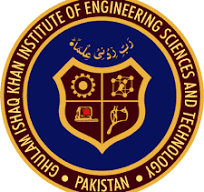
**Ghulam Ishaq Khan Institute of Engineering Sciences and Technology**

**Faculty of Computer Science and Engineering**

**CS342 – Numerical Analysis Project**

**Submitted To: Sir Aamir Shehzad**

**Dated: 4th May, 2025**

**Group Members:**

**Junaid Saleem--2022243**

**Abdul Mueed Khan--2022013**

**Muazzam Shah--2022312**

**Dua-e-Zahra Naqvi--2022151**

**Neha Abdul Rahim--2022481**

Table of Contents

[1 Bisection Method 6](#_Toc197276260)

[1.1 Method Overview 6](#_Toc197276261)

[1.2 Explanation of the Code 6](#_Toc197276262)

[1.3 Tables of Results 6](#_Toc197276263)

[1.4 Graphs 9](#_Toc197276264)

[1.5 Interpretation of Results 10](#_Toc197276265)

[2 Fixed Point Iteration 11](#_Toc197276270)

[2.1 Method Overview 11](#_Toc197276271)

[2.2 Explanation of Code 11](#_Toc197276272)

[2.3 Table of Results 11](#_Toc197276273)

[2.4 Graphs 13](#_Toc197276274)

[2.5 Interpretation of Results 15](#_Toc197276275)

[3 Newton-Raphson Method 16](#_Toc197276280)

[3.1 Method Overview 16](#_Toc197276281)

[3.2 Explanation of the Code 16](#_Toc197276282)

[3.3 Table of Results 16](#_Toc197276283)

[3.4 Graphs 17](#_Toc197276284)

[3.5 Interpretation of Results 18](#_Toc197276285)

[4 Secant Method 19](#_Toc197276290)

[4.1 Method Overview 19](#_Toc197276291)

[4.2 Explanation of the Code 19](#_Toc197276292)

[4.3 Table of Results 19](#_Toc197276293)

[4.4 Graphs 20](#_Toc197276294)

[4.5 Interpretation of Results 22](#_Toc197276295)

[5 Lagrange Interpolation 22](#_Toc197276300)

[5.1 Method Overview 22](#_Toc197276301)

[5.2 Explanation of the Code 22](#_Toc197276302)

[5.3 Table of Results 23](#_Toc197276303)

[5.4 Graphs 25](#_Toc197276304)

[5.5 Interpretation of Results 26](#_Toc197276305)

[6 Hermite Interpolation 27](#_Toc197276310)

[6.1 Method Overview 27](#_Toc197276311)

[6.2 Explanation of Code 27](#_Toc197276312)

[6.3 Table of Results 28](#_Toc197276313)

[6.4 Graphs 30](#_Toc197276314)

[6.5 Interpretation of Results 31](#_Toc197276315)

[7 Cubic Spline Interpolation 32](#_Toc197276320)

[7.1 Method Overview 32](#_Toc197276321)

[7.2 Explanation of Code 32](#_Toc197276322)

[7.3 Table of Results 33](#_Toc197276323)

[7.4 Graphs 36](#_Toc197276324)

[7.5 Interpretation of Results 37](#_Toc197276325)

[8 Parametric Curves 38](#_Toc197276330)

[8.1 Method Overview 38](#_Toc197276331)

[8.2 Explanation of Code 38](#_Toc197276332)

[8.3 Graph and Data 39](#_Toc197276336)

[9 Numerical Differentiation 43](#_Toc197276340)

[9.1 Method Overview 43](#_Toc197276341)

[9.2 Explanation of Code 43](#_Toc197276342)

[9.3 Results and Graphs 43](#_Toc197276343)

[9.4 Interpretation of Results 47](#_Toc197276344)

[9.5 Conclusion 48](#_Toc197276348)

[10 Richardson Extrapolation 48](#_Toc197276349)

[10.1 Method Overview 48](#_Toc197276350)

[10.2 Explanation of Code 49](#_Toc197276351)

[10.3 Table and Graphs 49](#_Toc197276353)

[10.4 Interpretation of Results 52](#_Toc197276354)

[11 Numerical Integration 53](#_Toc197276359)

[11.1 Method Overview 53](#_Toc197276360)

[11.2 Explanation of Code 53](#_Toc197276361)

[11.3 Results and Graphs 54](#_Toc197276363)

[11.4 Interpretation of Results 58](#_Toc197276364)

[12 Romberg Integration 59](#_Toc197276369)

[12.1 Method Overview 59](#_Toc197276370)

[12.2 Explanation of Code 59](#_Toc197276371)

[12.3 Results and Graphs: 60](#_Toc197276372)

[12.4 Interpretation of Results 65](#_Toc197276374)

[13 Gaussian Elimination and Variants 66](#_Toc197276379)

[13.1 Standard Gaussian Elimination 66](#_Toc197276380)

[13.1.1 Forward Elimination 66](#_Toc197276381)

[13.1.2 Back Substitution: 66](#_Toc197276382)

[14 Pivoting Strategies 67](#_Toc197276383)

[14.1 Gaussian Elimination with Partial Pivoting: 67](#_Toc197276384)

[14.2 Gaussian Elimination with Complete Pivoting: 69](#_Toc197276392)

[14.3 Gaussian Elimination with Scaled Partial Pivoting 70](#_Toc197276400)

[15 Matrix Factorization 72](#_Toc197276408)

[15.1 LU Decomposition 72](#_Toc197276409)

[15.2 DoLittle Decomposition 74](#_Toc197276416)

[15.3 Crout LU Decomposition 75](#_Toc197276423)

[15.4 Cholesky LU Decomposition 77](#_Toc197276430)

# Bisection Method

## ****Method Overview****

The bisection method is a root-finding algorithm that:

* Requires a continuous function that changes sign over an interval [a, b]
* Repeatedly bisects the interval and selects the subinterval containing the root
* Guarantees convergence to a root if the function is continuous
* Uses a tolerance of 1e-6 and maximum 100 iterations as stopping criteria
* Calculates error as the difference between consecutive approximations

## Explanation of the Code

The implementation consists of three main functions:

**bisectionMethod:**

* Takes function, interval bounds [a, b], tolerance, and max iterations
* Returns root approximation, iteration count, and iteration data
* Checks for sign change at interval endpoints
* Updates interval based on function value at midpoint

**printResults:**

* Shows current interval bounds [a, b] and their function values
* Displays midpoint (x) and its function value
* Calculates error between consecutive approximations
* Formats output in a clear tabular form

**plotResults:**

* Error vs Iterations plot showing convergence
* Function curve plot with root location

## Tables of Results

**Stopping Tolerance: 10-6**

**f(x) = x3- 4x + 1**

**Interval: [0,1]**

| **Iteration** | **a** | **f(a)** | **b** | **f(b)** | **x** | **f(x)** | **Error** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0.000000 | 1.000000 | 1.000000 | -2.000000 | 0.500000 | -0.875000 | inf |
| 2 | 0.000000 | 1.000000 | 0.500000 | -0.875000 | 0.250000 | 0.015625 | 0.250000 |
| 3 | 0.250000 | 0.015625 | 0.500000 | -0.875000 | 0.375000 | -0.447266 | 0.125000 |
| 4 | 0.250000 | 0.015625 | 0.375000 | -0.447266 | 0.312500 | -0.219482 | 0.062500 |
| 5 | 0.250000 | 0.015625 | 0.312500 | -0.219482 | 0.281250 | -0.102753 | 0.031250 |
| 6 | 0.250000 | 0.015625 | 0.281250 | -0.102753 | 0.265625 | -0.043758 | 0.015625 |
| 7 | 0.250000 | 0.015625 | 0.265625 | -0.043758 | 0.257812 | -0.014114 | 0.007812 |
| 8 | 0.250000 | 0.015625 | 0.257812 | -0.014114 | 0.253906 | 0.000744 | 0.003906 |
| 9 | 0.253906 | 0.000744 | 0.257812 | -0.014114 | 0.255859 | -0.006688 | 0.001953 |
| 10 | 0.253906 | 0.000744 | 0.255859 | -0.006688 | 0.254883 | -0.002973 | 0.000977 |
| 11 | 0.253906 | 0.000744 | 0.254883 | -0.002973 | 0.254395 | -0.001115 | 0.000488 |
| 12 | 0.253906 | 0.000744 | 0.254395 | -0.001115 | 0.254150 | -0.000185 | 0.000244 |
| 13 | 0.253906 | 0.000744 | 0.254150 | -0.000185 | 0.254028 | 0.000279 | 0.000122 |
| 14 | 0.254028 | 0.000279 | 0.254150 | -0.000185 | 0.254089 | 0.000047 | 0.000061 |
| 15 | 0.254089 | 0.000047 | 0.254150 | -0.000185 | 0.254120 | -0.000069 | 0.000031 |
| 16 | 0.254089 | 0.000047 | 0.254120 | -0.000069 | 0.254105 | -0.000011 | 0.000015 |
| 17 | 0.254089 | 0.000047 | 0.254105 | -0.000011 | 0.254097 | 0.000018 | 0.000008 |
| 18 | 0.254097 | 0.000018 | 0.254105 | -0.000011 | 0.254101 | 0.000003 | 0.000004 |
| 19 | 0.254101 | 0.000003 | 0.254105 | -0.000011 | 0.254103 | -0.000004 | 0.000002 |
| 20 | 0.254101 | 0.000003 | 0.254103 | -0.000004 | 0.254102 | -0.000000 | 0.000001 |

**f(x) =**

**Interval: [0,10]**

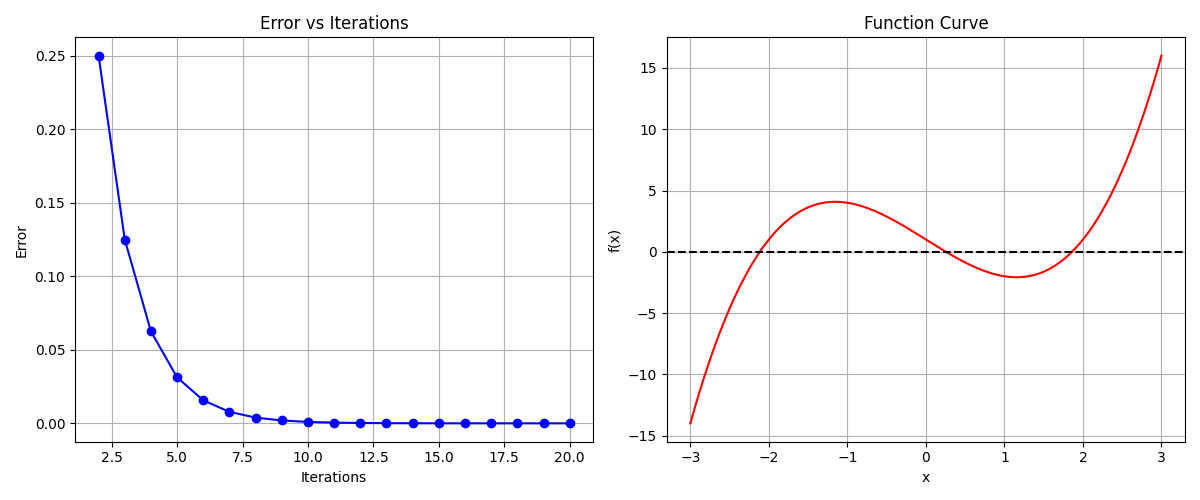
| **Iteration** | **a** | **f(a)** | **b** | **f(b)** | **x** | **f(x)** | **Error** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0.000000 | -2.000000 | 10.000000 | 1.162278 | 5.000000 | 0.236068 | inf |
| 2 | 0.000000 | -2.000000 | 5.000000 | 0.236068 | 2.500000 | -0.418861 | 2.500000 |
| 3 | 2.500000 | -0.418861 | 5.000000 | 0.236068 | 3.750000 | -0.063508 | 1.250000 |
| 4 | 3.750000 | -0.063508 | 5.000000 | 0.236068 | 4.375000 | 0.091650 | 0.625000 |
| 5 | 3.750000 | -0.063508 | 4.375000 | 0.091650 | 4.062500 | 0.015564 | 0.312500 |
| 6 | 3.750000 | -0.063508 | 4.062500 | 0.015564 | 3.906250 | -0.023576 | 0.156250 |
| 7 | 3.906250 | -0.023576 | 4.062500 | 0.015564 | 3.984375 | -0.003910 | 0.078125 |
| 8 | 3.984375 | -0.003910 | 4.062500 | 0.015564 | 4.023438 | 0.005851 | 0.039062 |
| 9 | 3.984375 | -0.003910 | 4.023438 | 0.005851 | 4.003906 | 0.000976 | 0.019531 |
| 10 | 3.984375 | -0.003910 | 4.003906 | 0.000976 | 3.994141 | -0.001465 | 0.009766 |
| 11 | 3.994141 | -0.001465 | 4.003906 | 0.000976 | 3.999023 | -0.000244 | 0.004883 |
| 12 | 3.999023 | -0.000244 | 4.003906 | 0.000976 | 4.001465 | 0.000366 | 0.002441 |
| 13 | 3.999023 | -0.000244 | 4.001465 | 0.000366 | 4.000244 | 0.000061 | 0.001221 |
| 14 | 3.999023 | -0.000244 | 4.000244 | 0.000061 | 3.999634 | -0.000092 | 0.000610 |
| 15 | 3.999634 | -0.000092 | 4.000244 | 0.000061 | 3.999939 | -0.000015 | 0.000305 |
| 16 | 3.999939 | -0.000015 | 4.000244 | 0.000061 | 4.000092 | 0.000023 | 0.000153 |
| 17 | 3.999939 | -0.000015 | 4.000092 | 0.000023 | 4.000015 | 0.000004 | 0.000076 |
| 18 | 3.999939 | -0.000015 | 4.000015 | 0.000004 | 3.999977 | -0.000006 | 0.000038 |
| 19 | 3.999977 | -0.000006 | 4.000015 | 0.000004 | 3.999996 | -0.000001 | 0.000019 |
| 20 | 3.999996 | -0.000001 | 4.000015 | 0.000004 | 4.000006 | 0.000001 | 0.000010 |
| 21 | 3.999996 | -0.000001 | 4.000006 | 0.000001 | 4.000001 | 0.000000 | 0.000005 |
| 22 | 3.999996 | -0.000001 | 4.000001 | 0.000000 | 3.999999 | -0.000000 | 0.000002 |
| 23 | 3.999999 | -0.000000 | 4.000001 | 0.000000 | 4.000000 | -0.000000 | 0.000001 |
| 24 | 4.000000 | -0.000000 | 4.000001 | 0.000000 | 4.000000 | 0.000000 | 0.000001 |

**f(x) = cos(x) - x**

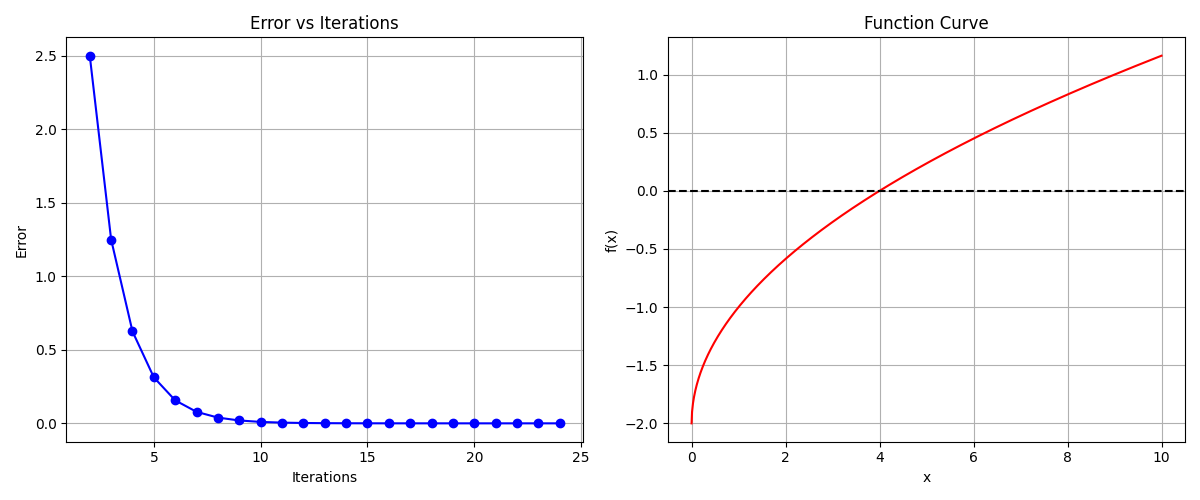
**Interval: [0,1]**

| **Iteration** | **a** | **f(a)** | **b** | **f(b)** | **x** | **f(x)** | **Error** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0.000000 | 1.000000 | 1.000000 | -0.459698 | 0.500000 | 0.377583 | inf |
| 2 | 0.500000 | 0.377583 | 1.000000 | -0.459698 | 0.750000 | -0.018311 | 0.250000 |
| 3 | 0.500000 | 0.377583 | 0.750000 | -0.018311 | 0.625000 | 0.185963 | 0.125000 |
| 4 | 0.625000 | 0.185963 | 0.750000 | -0.018311 | 0.687500 | 0.085335 | 0.062500 |
| 5 | 0.687500 | 0.085335 | 0.750000 | -0.018311 | 0.718750 | 0.033879 | 0.031250 |
| 6 | 0.718750 | 0.033879 | 0.750000 | -0.018311 | 0.734375 | 0.007875 | 0.015625 |
| 7 | 0.734375 | 0.007875 | 0.750000 | -0.018311 | 0.742188 | -0.005196 | 0.007812 |
| 8 | 0.734375 | 0.007875 | 0.742188 | -0.005196 | 0.738281 | 0.001345 | 0.003906 |
| 9 | 0.738281 | 0.001345 | 0.742188 | -0.005196 | 0.740234 | -0.001924 | 0.001953 |
| 10 | 0.738281 | 0.001345 | 0.740234 | -0.001924 | 0.739258 | -0.000289 | 0.000977 |
| 11 | 0.738281 | 0.001345 | 0.739258 | -0.000289 | 0.738770 | 0.000528 | 0.000488 |
| 12 | 0.738770 | 0.000528 | 0.739258 | -0.000289 | 0.739014 | 0.000120 | 0.000244 |
| 13 | 0.739014 | 0.000120 | 0.739258 | -0.000289 | 0.739136 | -0.000085 | 0.000122 |
| 14 | 0.739014 | 0.000120 | 0.739136 | -0.000085 | 0.739075 | 0.000017 | 0.000061 |
| 15 | 0.739075 | 0.000017 | 0.739136 | -0.000085 | 0.739105 | -0.000034 | 0.000031 |
| 16 | 0.739075 | 0.000017 | 0.739105 | -0.000034 | 0.739090 | -0.000008 | 0.000015 |
| 17 | 0.739075 | 0.000017 | 0.739090 | -0.000008 | 0.739082 | 0.000005 | 0.000008 |
| 18 | 0.739082 | 0.000005 | 0.739090 | -0.000008 | 0.739086 | -0.000002 | 0.000004 |
| 19 | 0.739082 | 0.000005 | 0.739086 | -0.000002 | 0.739084 | 0.000001 | 0.000002 |
| 20 | 0.739084 | 0.000001 | 0.739086 | -0.000002 | 0.739085 | -0.000000 | 0.000001 |

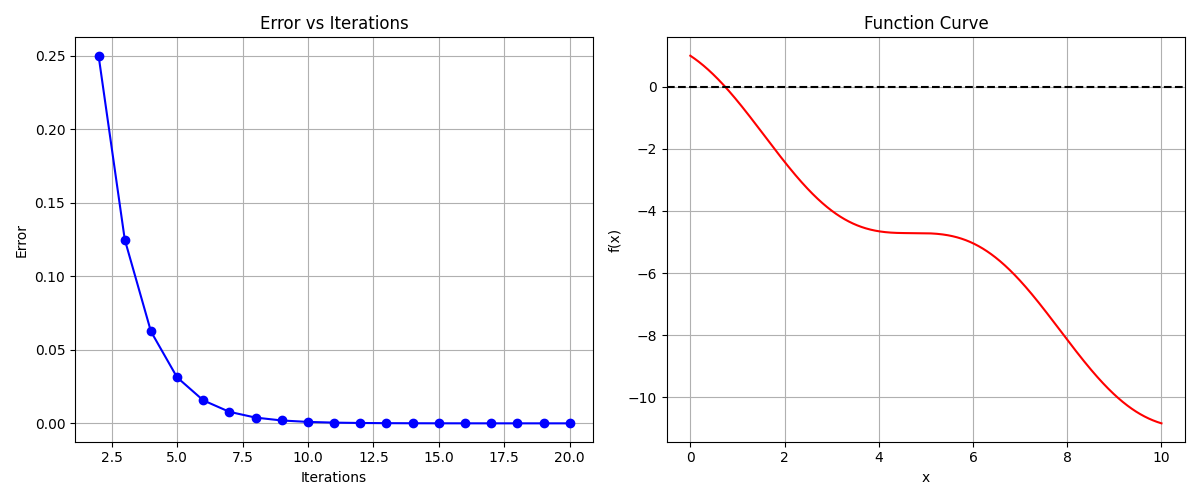
## Graphs

**f(x) = x3- 4x + 1**

**f(x) =**

****

**f(x) = cos(x) - x**

****

## ****Interpretation of Results****

The bisection method was applied to three functions, each with a known sign change in the given interval. In all cases, the method successfully converged to a root within the specified tolerance of 10−6.

### ****Function 1:**** (x) = x3 - 4x + 1 ****on**** [0, 1]

* **Root found:** ≈ 0.254102
* **Iterations:** 20
* **Remarks:** Function changes sign over [0,1]; convergence was steady and accurate.

### ****Function 2:**** f(x)= ****on**** [0, 10]

* **Root found:** ≈ 4.000000
* **Iterations:** 24
* **Remarks:** Root at x=4x = 4 was accurately detected; error halved each step.

### ****Function 3:**** f(x)=cos(x)−x on [0, 1]

* **Root found:** ≈ 0.739085
* **Iterations:** 20
* **Remarks:** Classic fixed-point problem; bisection method converged reliably.

### ****Conclusion****

The method showed consistent convergence in all cases, with clear reduction in error and root approximation through midpoint updates. Tabulated results confirm the method's precision and stability.

# Fixed Point Iteration

## Method Overview

The Fixed-Point Iteration method is a numerical technique used to find fixed points of a function, where x = g(x). A fixed point is a value that remains unchanged when the function is applied to it. This implementation provides a robust solution with visualization and detailed iteration tracking.

## Explanation of Code

**Main Function: fixedPointIteration()**

**Input Parameters:**

* g(x): The iteration function
* x0: Initial guess
* tolerance: Convergence criterion (default: 1e-6)
* maxIterations: Maximum allowed iterations (default: 100)
* Returns: (fixed point, iterations count, iteration data)

**Visualization: plotResults()**

Generates two plots:

* Error convergence over iterations
* Iteration function g(x) with y=x line intersection

**Results Display: printResults()**

* Presents iteration data in a formatted table
* Shows convergence progress and final results

## Table of Results

**Tolerance Limit = 10-6**

**g(x)=cos(x)**

**Initial guess: x0=0**

| **Iteration** | **xₙ** | **g(xₙ)** | **Error** |
| --- | --- | --- | --- |
| 1 | 1.000000 | 0.540302 | 1.000000 |
| 2 | 0.540302 | 0.857553 | 0.459698 |
| 3 | 0.857553 | 0.654290 | 0.317251 |
| 4 | 0.654290 | 0.793480 | 0.203263 |
| 5 | 0.793480 | 0.701369 | 0.139191 |
| 6 | 0.701369 | 0.763960 | 0.092112 |
| 7 | 0.763960 | 0.722102 | 0.062591 |
| 8 | 0.722102 | 0.750418 | 0.041857 |
| 9 | 0.750418 | 0.731404 | 0.028315 |
| 10 | 0.731404 | 0.744237 | 0.019014 |
| 11 | 0.744237 | 0.735605 | 0.012833 |
| 12 | 0.735605 | 0.741425 | 0.008633 |
| 13 | 0.741425 | 0.737507 | 0.005820 |
| 14 | 0.737507 | 0.740147 | 0.003918 |
| 15 | 0.740147 | 0.738369 | 0.002640 |
| 16 | 0.738369 | 0.739567 | 0.001778 |
| 17 | 0.739567 | 0.738760 | 0.001198 |
| 18 | 0.738760 | 0.739304 | 0.000807 |
| 19 | 0.739304 | 0.738938 | 0.000544 |
| 20 | 0.738938 | 0.739184 | 0.000366 |
| 21 | 0.739184 | 0.739018 | 0.000247 |
| 22 | 0.739018 | 0.739130 | 0.000166 |
| 23 | 0.739130 | 0.739055 | 0.000112 |
| 24 | 0.739055 | 0.739106 | 0.000075 |
| 25 | 0.739106 | 0.739071 | 0.000051 |
| 26 | 0.739071 | 0.739094 | 0.000034 |
| 27 | 0.739094 | 0.739079 | 0.000023 |
| 28 | 0.739079 | 0.739089 | 0.000016 |
| 29 | 0.739089 | 0.739082 | 0.000010 |
| 30 | 0.739082 | 0.739087 | 0.000007 |
| 31 | 0.739087 | 0.739084 | 0.000005 |
| 32 | 0.739084 | 0.739086 | 0.000003 |
| 33 | 0.739086 | 0.739085 | 0.000002 |
| 34 | 0.739085 | 0.739086 | 0.000001 |
| 35 | 0.739086 | 0.739085 | 0.000001 |

**g(x)=**

**Initial guess: x0=0**

| **Iteration** | **xₙ** | **g(xₙ)** | **Error** |
| --- | --- | --- | --- |
| 1 | 0.666667 | 0.888889 | 0.666667 |
| 2 | 0.888889 | 0.962963 | 0.222222 |
| 3 | 0.962963 | 0.987654 | 0.074074 |
| 4 | 0.987654 | 0.995885 | 0.024691 |
| 5 | 0.995885 | 0.998628 | 0.008230 |
| 6 | 0.998628 | 0.999543 | 0.002743 |
| 7 | 0.999543 | 0.999848 | 0.000914 |
| 8 | 0.999848 | 0.999949 | 0.000305 |
| 9 | 0.999949 | 0.999983 | 0.000102 |
| 10 | 0.999983 | 0.999994 | 0.000034 |
| 11 | 0.999994 | 0.999998 | 0.000011 |
| 12 | 0.999998 | 0.999999 | 0.000004 |
| 13 | 0.999999 | 1.000000 | 0.000001 |

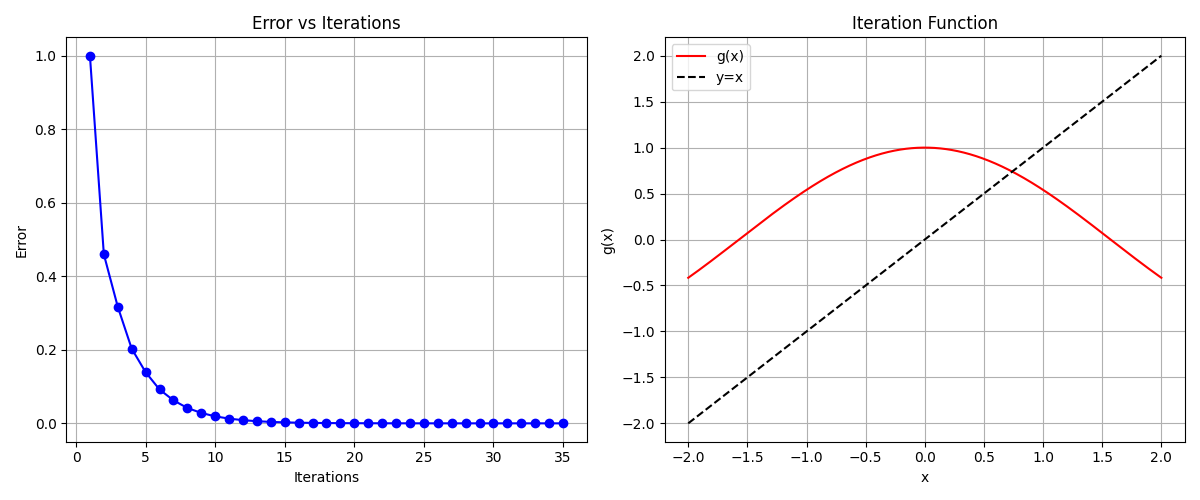
**g(x)=**

**Initial guess: x0=2**

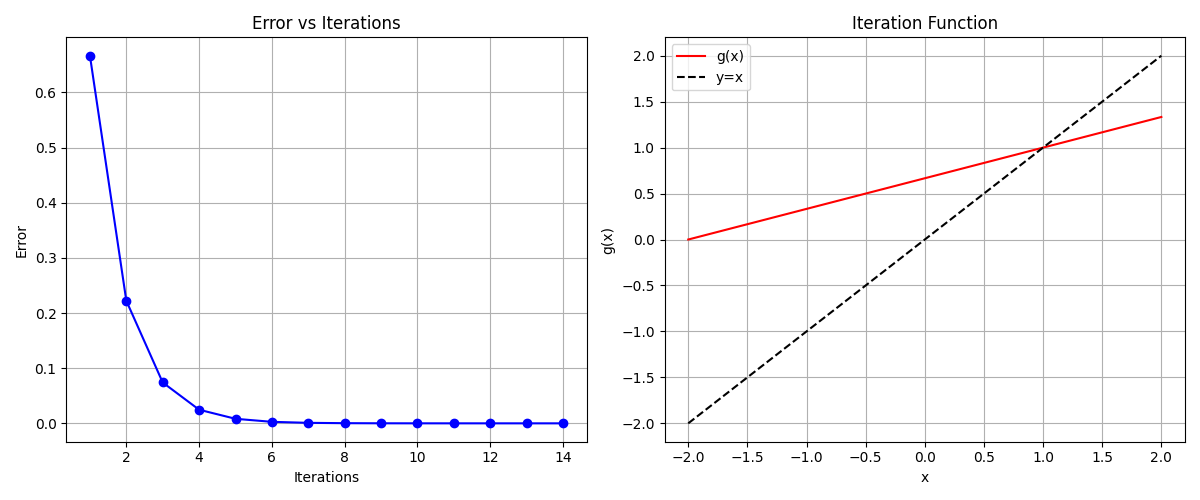
| **Iteration** | **xₙ** | **g(xₙ)** | **Error** |
| --- | --- | --- | --- |
| 1 | 1.250000 | 1.025000 | 0.750000 |
| 2 | 1.025000 | 1.000305 | 0.225000 |
| 3 | 1.000305 | 1.000000 | 0.024695 |
| 4 | 1.000000 | 1.000000 | 0.000305 |
| 5 | 1.000000 | 1.000000 | 0.000000 |

## Graphs

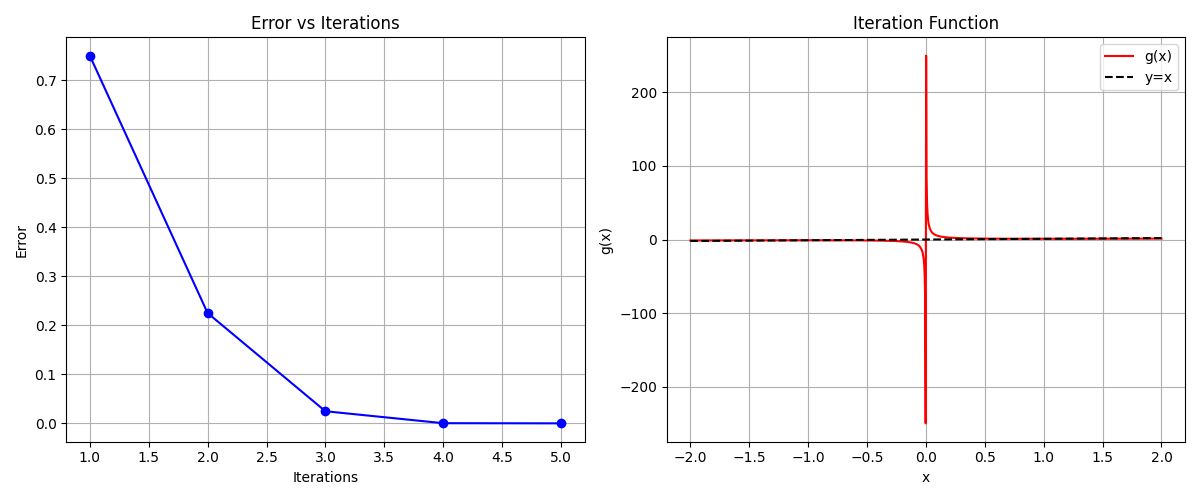
**g(x)=cos(x)**



**g(x)=**



**g(x)=**



## ****Interpretation of Results****

The Fixed-Point Iteration method was applied to three functions using an initial guess and a convergence tolerance of 10−6. Results are as follows:

### ****Function 1:**** g(x)=cos(x), x0=0

* **Fixed Point:** ≈ 0.739085
* **Iterations:** 35
* **Remarks:** Convergence was gradual due to the nature of cosine near the root, showing consistent error reduction.

### ****Function 2:**** g(x)= , x0=0

* **Fixed Point:** ≈ 1.000000
* **Iterations:** 13
* **Remarks:** Smooth and steady convergence with fewer iterations, demonstrating strong contractive behavior.

### ****Function 3:**** g(x)= , x0=2

* **Fixed Point:** ≈ 1.000000
* **Iterations:** 5
* **Remarks:** Fast convergence due to the function’s rapid correction of the initial guess.

### ****Conclusion****

The method successfully converged in all cases, with the speed of convergence dependent on the function's nature and constructiveness. Proper choice of g(x)g(x) and initial guess significantly impacts performance.

# Newton-Raphson Method

## Method Overview

The Newton-Raphson method is an iterative technique for finding roots of a differentiable function. It uses the function's derivative to generate successively better approximations to the roots of a real-valued function. The method starts with an initial guess and uses the tangent line at that point to find the next approximation.

## Explanation of the Code

**Main Function: newtonRaphson()**

**Input Parameters:**

* f(x): The function whose root we want to find
* df(x): The derivative of f(x)
* x0: Initial guess
* tolerance: Convergence criterion (default: 1e-6)
* maxIterations: Maximum allowed iterations (default: 100)
* Returns: (root, iterations count, iteration data)

**Visualization: plotResults()**

**Generates three plots:**

* Error convergence over iterations
* Original function f(x)
* Derivative function f'(x)

**Results Display: printResults()**

* Presents iteration data in a formatted table
* Shows convergence progress and final results

## Table of Results

**Tolerance Limit = 10-6**

**f(x)=x3−x−2**

**Initial guess: x0=2**

| **Iteration** | **xₙ** | **f(xₙ)** | **Error** |
| --- | --- | --- | --- |
| 1 | 1.636364 | 0.745304 | 0.363636 |
| 2 | 1.530392 | 0.053939 | 0.105972 |
| 3 | 1.521441 | 0.000367 | 0.008951 |
| 4 | 1.521380 | 0.000000 | 0.000062 |
| 5 | 1.521380 | 0.000000 | 0.000000 |

**f(x)=x2 – cos(x)**

**Initial guess: x0=1**

| **Iteration** | **xₙ** | **f(xₙ)** | **Error** |
| --- | --- | --- | --- |
| 1 | 0.838218 | 0.033822 | 0.161782 |
| 2 | 0.824242 | 0.000261 | 0.013977 |
| 3 | 0.824132 | 0.000000 | 0.000110 |
| 4 | 0.824132 | 0.000000 | 0.000000 |

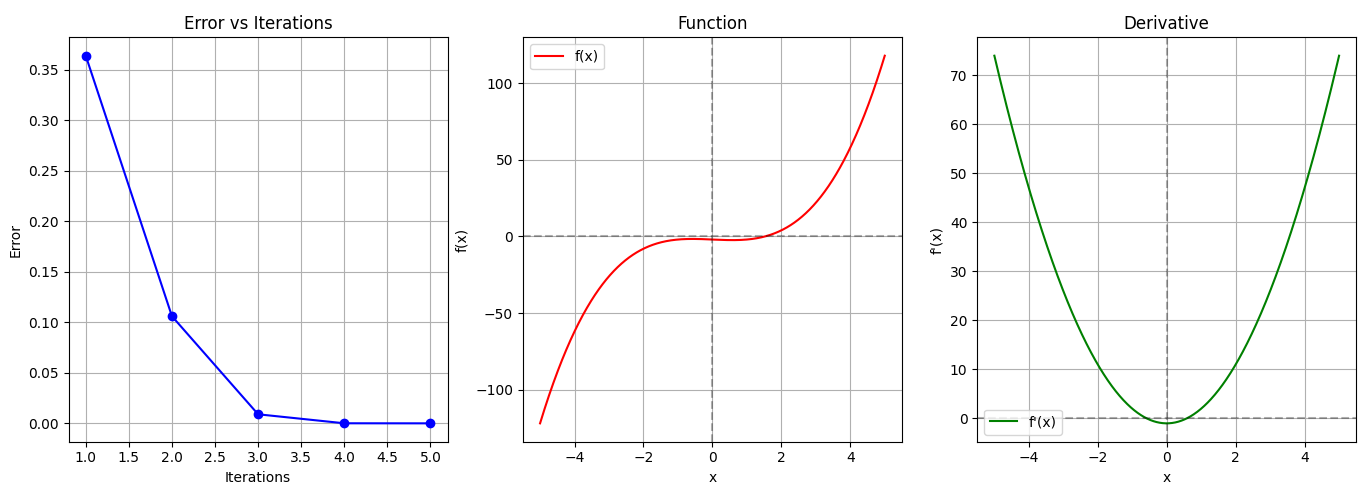
**f(x) = ex – 5x**

**Initial guess: x0=2**

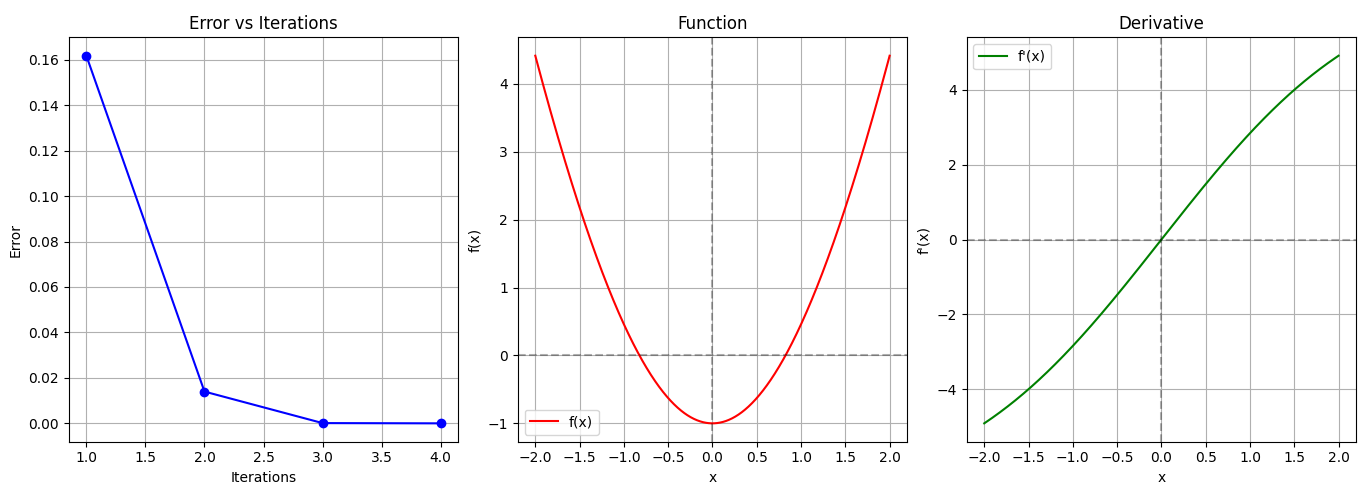
| **Iteration** | **xₙ** | **f(xₙ)** | **Error** |
| --- | --- | --- | --- |
| 1 | 3.092877 | 6.576008 | 1.092877 |
| 2 | 2.706970 | 1.448952 | 0.385907 |
| 3 | 2.561839 | 0.150436 | 0.145130 |
| 4 | 2.542939 | 0.002300 | 0.018900 |
| 5 | 2.542641 | 0.000001 | 0.000298 |
| 6 | 2.542641 | 0.000000 | 0.000000 |

## Graphs

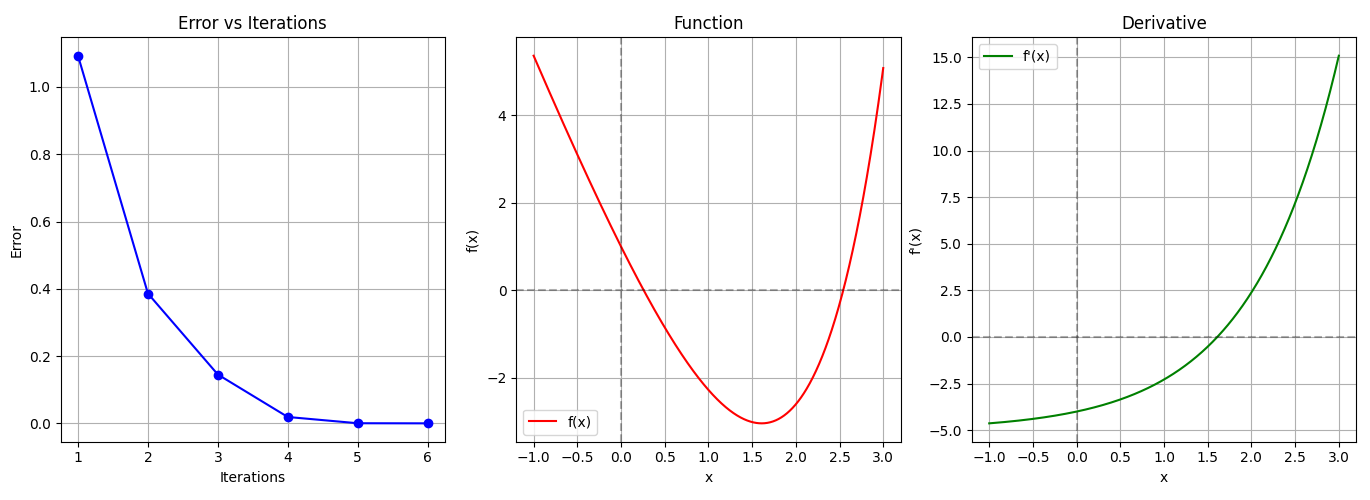
**f(x)=x3−x−2**



**f(x)=****x2 – cos(x)**



**f(x) = ex – 5x**



## ****Interpretation of Results****

The Newton-Raphson method was applied to three functions using an initial guess and a convergence tolerance of 10−6. Results are as follows:

### ****Function 1:**** f(x)=x3−x−2, x0=2

* **Root Found:** ≈ 1.521380
* **Iterations:** 5
* **Remarks:** Fast convergence with quadratic error reduction as expected from the method's nature.

### ****Function 2:**** f(x)=x2 – cos(x), x0=1

* **Root Found:** ≈ 0.824132
* **Iterations:** 4
* **Remarks:** Smooth convergence with rapid stabilization of function values.

### ****Function 3:**** f(x)=ex−5x, x0=2

* **Root Found:** ≈ 2.542641
* **Iterations:** 6
* **Remarks:** Initial large error due to steep slope, followed by steady convergence.

### ****Conclusion****

Newton-Raphson method demonstrated efficient and rapid convergence for all tested functions. Its effectiveness depends on a good initial guess and the behavior of the derivative near the root.

# Secant Method

## Method Overview

The Secant Method is a root-finding algorithm that uses a succession of roots of secant lines to better approximate a root of a function. It's similar to the Newton-Raphson method but doesn't require derivatives, making it particularly useful when derivatives are difficult or expensive to compute.

## Explanation of the Code

**Main Function: secant()**

**Input Parameters:**

* f(x): The function whose root we want to find
* x0: First initial guess
* x1: Second initial guess
* tolerance: Convergence criterion (default: 1e-6)
* maxIterations: Maximum allowed iterations (default: 100)
* Returns: (root, iterations count, iteration data)

**Visualization: plotResults()**

**Generates two plots:**

* Error convergence over iterations
* Function f(x) with root location

**Results Display: printResults()**

* Presents iteration data in a formatted table
* Shows convergence progress and final results

## Table of Results

**Tolerance Limit = 10-6**

**g(x)=x3 – 2x - 5**

**Initial guess: x0=2, x1 = 3**

| **Iteration** | **xₙ** | | **f(xₙ)** | **Error** |
| --- | --- | --- | --- | --- |
| 1 | 2.058824 | -0.390800 | | 0.941176 |
| 2 | 2.081264 | -0.147204 | | 0.022440 |
| 3 | 2.094824 | 0.003044 | | 0.013560 |
| 4 | 2.094549 | -0.000023 | | 0.000275 |
| 5 | 2.094551 | -0.000000 | | 0.000002 |
| 6 | 2.094551 | 0.000000 | | 0.000000 |

**f(x)=sin(x)−x/2**

**Initial guess: x0=1.5, x1 = 2**

| **Iteration** | **xₙ** | **f(xₙ)** | **Error** |
| --- | --- | --- | --- |
| 1 | 1.865903 | 0.023820 | 0.134097 |
| 2 | 1.893794 | 0.001391 | 0.027891 |
| 3 | 1.895524 | -0.000024 | 0.001730 |
| 4 | 1.895494 | 0.000000 | 0.000030 |
| 5 | 1.895494 | 0.000000 | 0.000000 |

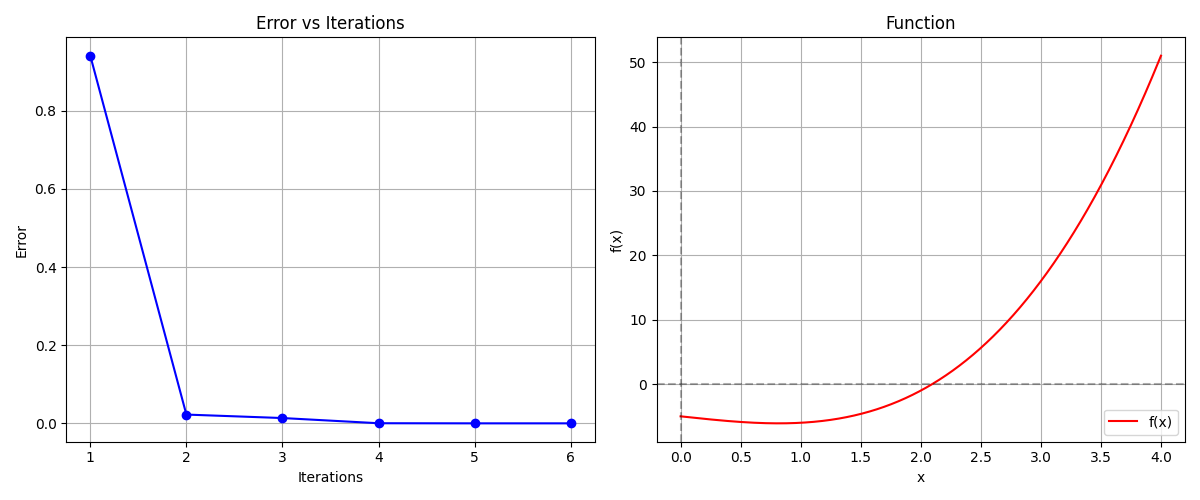
**f(x) = ex – 5x**

**Initial guess: x0=1, x1 = 2**

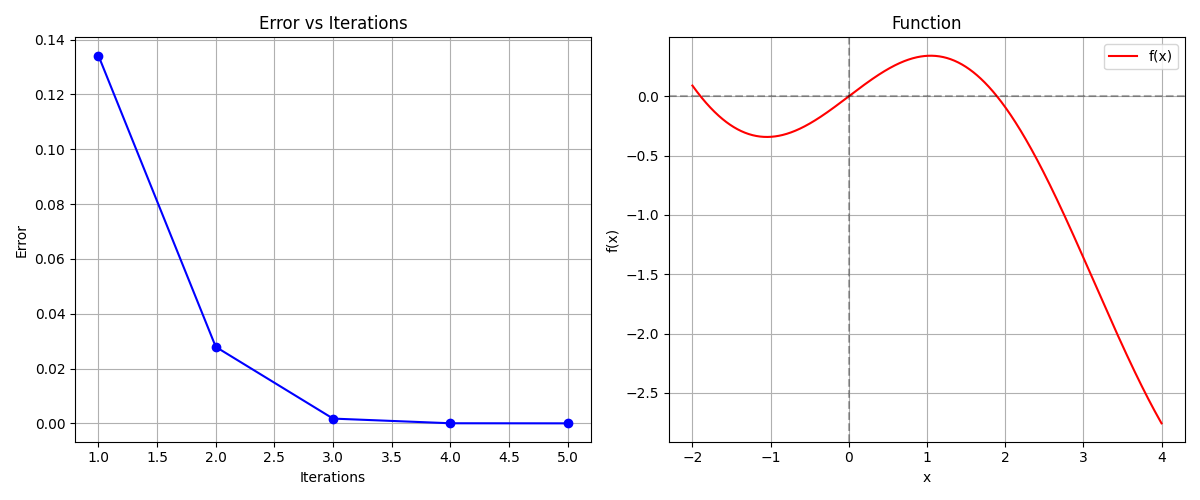
| **Iteration** | **xₙ** | **f(xₙ)** | **Error** |
| --- | --- | --- | --- |
| 1 | -5.930558 | 29.655449 | 7.930558 |
| 2 | 1.358272 | -2.901894 | 7.288831 |
| 3 | 0.708606 | -1.511871 | 0.649666 |
| 4 | 0.001990 | 0.992043 | 0.706616 |
| 5 | 0.281949 | -0.084033 | 0.279959 |
| 6 | 0.260086 | -0.003389 | 0.021863 |
| 7 | 0.259167 | 0.000014 | 0.000919 |
| 8 | 0.259171 | -0.000000 | 0.000004 |
| 9 | 0.259171 | -0.000000 | 0.000000 |

## Graphs

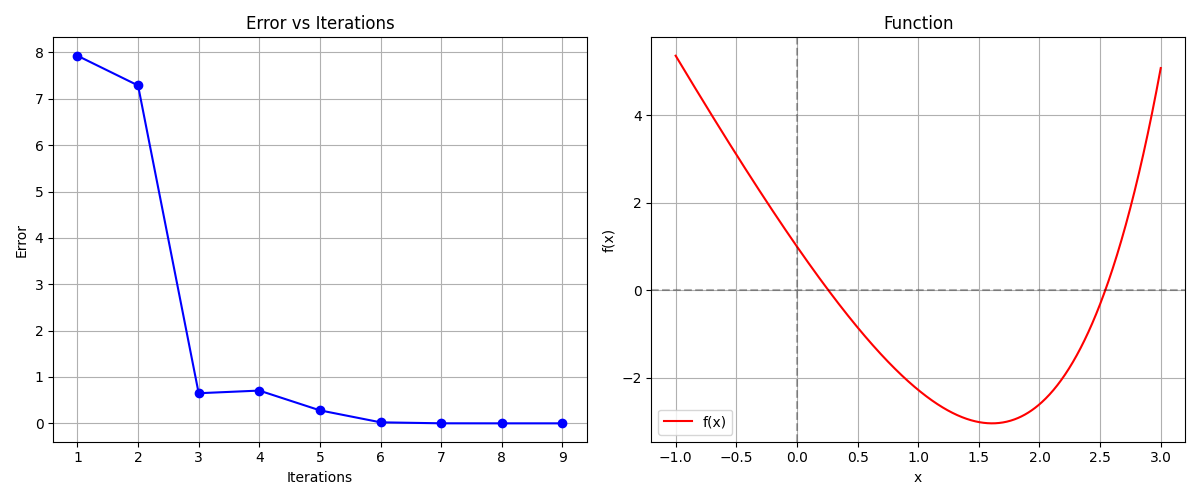
**g(x)=x3 – 2x - 5**



**f(x)=sin(x)−x/2**



**f(x) = ex – 5x**



## ****Interpretation of Results****

The Secant Method was applied to three functions using two initial guesses and a convergence tolerance of 10−6. The outcomes are summarized below:

### ****Function 1:**** f(x)=x3−2x−5, x0=2, x1=3

* **Root Found:** ≈ 2.094551
* **Iterations:** 6
* **Remarks:** Rapid convergence without derivative use, closely matching Newton-Raphson performance.

### ****Function 2:**** f(x)=sin(x)−x/2, x0­=1.5, x1=2

* **Root Found:** ≈ 1.895494
* **Iterations:** 5
* **Remarks:** Smooth and stable convergence, even with initial moderate error.

### ****Function 3:**** f(x)=ex−5x, x0=1, x1=2

* **Root Found:** ≈ 0.259171
* **Iterations:** 9
* **Remarks:** Erratic start due to poor initial guesses, but successfully stabilized and converged.

### ****Conclusion****

The Secant Method effectively finds roots without requiring derivatives, offering good accuracy and convergence for all functions. It is especially advantageous when derivatives are difficult to compute, though careful selection of initial guesses is important for stability.

# Lagrange Interpolation

## Method Overview

Lagrange Interpolation is a polynomial interpolation method that creates a unique polynomial passing through n points. This implementation uses 15 data points sampled from sin(x) over the interval [0, 4π], demonstrating the method's ability to handle larger datasets while maintaining accuracy.

## Explanation of the Code

* [**lagrangeInterpolation**](vscode-file://vscode-app/c:/Users/JS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html): Constructs an interpolation polynomial using camelCase naming convention
* [**evaluatePoints**](vscode-file://vscode-app/c:/Users/JS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html): Tests interpolation accuracy at specified points
* [**plotResults**](vscode-file://vscode-app/c:/Users/JS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)**:** Generates comparative visualizations
* [**originalFunction**](vscode-file://vscode-app/c:/Users/JS/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)**:** Provides sin(x) as the test

## Table of Results

**f(x) = sin(x)**

**Interpolation Data**

| **Index** | **x** | **y** | |
| --- | --- | --- | --- |
| 0 | 0.0000 | | 0.0000 | |
| 1 | 0.8976 | | 0.7818 | |
| 2 | 1.7952 | | 0.9749 | |
| 3 | 2.6928 | | 0.4339 | |
| 4 | 3.5904 | | -0.4339 | |
| 5 | 4.4880 | | -0.9749 | |
| 6 | 5.3856 | | -0.7818 | |
| 7 | 6.2832 | | -0.0000 | |
| 8 | 7.1808 | | 0.7818 | |
| 9 | 8.0784 | | 0.9749 | |
| 10 | 8.9760 | | 0.4339 | |
| 11 | 9.8736 | | -0.4339 | |
| 12 | 10.7712 | | -0.9749 | |
| 13 | 11.6688 | | -0.7818 | |
| 14 | 12.5664 | | -0.0000 | |

**Test Points**

| **x** | **Interpolated** | **Function** | **Error** |
| --- | --- | --- | --- |
| 0.5000 | 0.4799 | 0.4794 | 4.85e-04 |
| 2.3000 | 0.7457 | 0.7457 | 1.22e-05 |
| 4.7000 | -0.9999 | -0.9999 | 8.20e-07 |
| 7.1000 | 0.7290 | 0.7290 | 2.51e-07 |
| 9.5000 | -0.0751 | -0.0752 | 4.44e-06 |
| 11.2000 | -0.9791 | -0.9792 | 6.29e-05 |

**f(x) = x² \* e-x/3**

**Interpolation Data**

| **Index** | **x** | **y** |
| --- | --- | --- |
| 0 | 0.0000 | 0.0000 |
| 1 | 0.7143 | 0.4021 |
| 2 | 1.4286 | 1.2676 |
| 3 | 2.1429 | 2.2479 |
| 4 | 2.8571 | 3.1496 |
| 5 | 3.5714 | 3.8785 |
| 6 | 4.2857 | 4.4018 |
| 7 | 5.0000 | 4.7219 |
| 8 | 5.7143 | 4.8607 |
| 9 | 6.4286 | 4.8484 |
| 10 | 7.1429 | 4.7175 |
| 11 | 7.8571 | 4.4987 |
| 12 | 8.5714 | 4.2195 |
| 13 | 9.2857 | 3.9029 |
| 14 | 10.0000 | 3.5674 |

**Test Points**

| **x** | **Interpolated** | **Function** | **Error** |
| --- | --- | --- | --- |
| 0.8000 | 0.4902 | 0.4902 | 6.19e-11 |
| 2.5000 | 2.7162 | 2.7162 | 4.62e-12 |
| 4.2000 | 4.3500 | 4.3500 | 3.53e-13 |
| 6.7000 | 4.8109 | 4.8109 | 1.53e-12 |
| 8.3000 | 4.3312 | 4.3312 | 1.46e-11 |
| 9.6000 | 3.7566 | 3.7566 | 4.86e-10 |

**f(x) = 1/(1 + x²)**

**Interpolation Data**

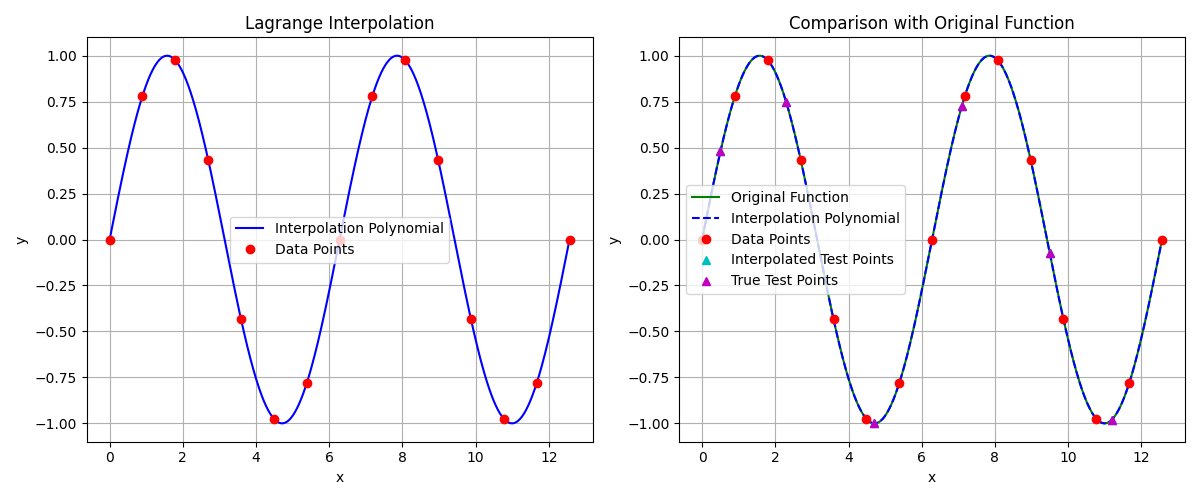
| **Index** | **x** | **y** |
| --- | --- | --- |
| 0 | -5.0000 | 0.0385 |
| 1 | -4.2857 | 0.0516 |
| 2 | -3.5714 | 0.0727 |
| 3 | -2.8571 | 0.1091 |
| 4 | -2.1429 | 0.1788 |
| 5 | -1.4286 | 0.3289 |
| 6 | -0.7143 | 0.6622 |
| 7 | 0.0000 | 1.0000 |
| 8 | 0.7143 | 0.6622 |
| 9 | 1.4286 | 0.3289 |
| 10 | 2.1429 | 0.1788 |
| 11 | 2.8571 | 0.1091 |
| 12 | 3.5714 | 0.0727 |
| 13 | 4.2857 | 0.0516 |
| 14 | 5.0000 | 0.0385 |

**Test Points**

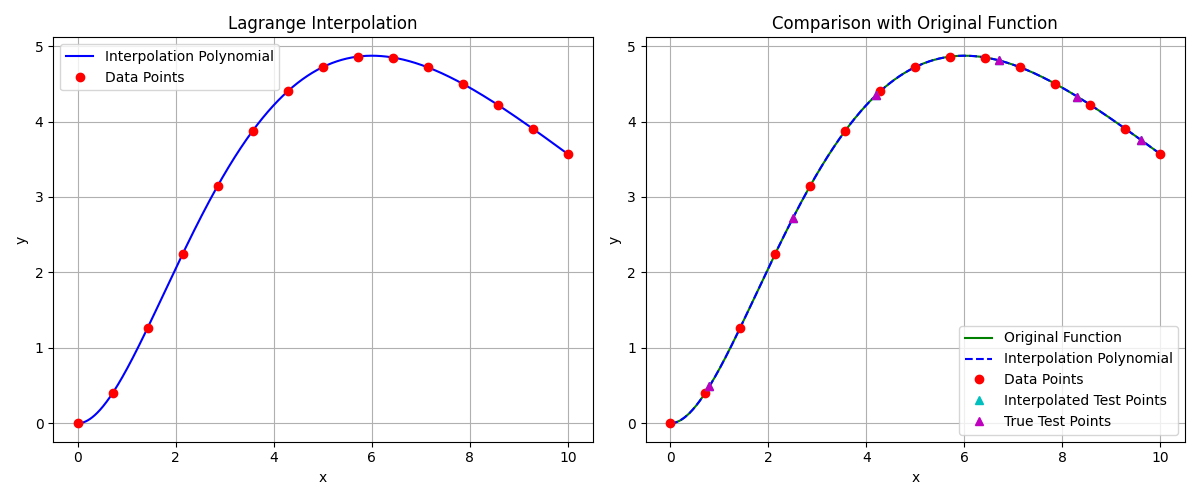
| **x** | **Interpolated** | **Function** | **Error** |
| --- | --- | --- | --- |
| -4.2000 | -0.4481 | 0.0536 | 5.02e-01 |
| -2.7000 | 0.0721 | 0.1206 | 4.85e-02 |
| -1.3000 | 0.3589 | 0.3717 | 1.29e-02 |
| 0.8000 | 0.6030 | 0.6098 | 6.77e-03 |
| 2.4000 | 0.0993 | 0.1479 | 4.86e-02 |
| 3.9000 | -0.5711 | 0.0617 | 6.33e-01 |

## Graphs

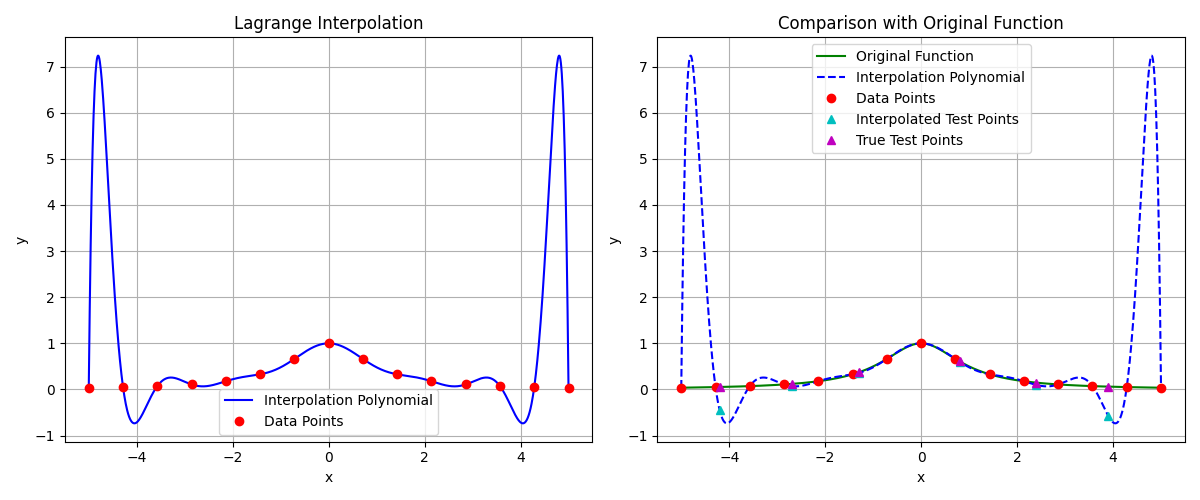
**f(x) = sin(x)**



**f(x) = x² \* e-x/3**



**f(x) = 1/ (1 + x²)**



## ****Interpretation of Results****

The Lagrange Interpolation method was applied to three distinct functions using 15 sample points each. The interpolated values were tested at multiple unseen points to evaluate the accuracy of the polynomial approximations.

### ****Function 1:**** f(x)=sin(x), x∈[0,4π]

* **Max Error:** ~6.29×10−5
* **Remarks:** The interpolation closely matches the true sine values at all test points. Despite the oscillatory nature of sine, Lagrange interpolation handled 15 points over a wide interval effectively, maintaining high precision.

### ****Function 2:**** f(x)=x2e−x/3, x∈ [0,10]

* **Max Error:** ~4.86×10−10
* **Remarks:** Exceptionally accurate interpolation. The function's smooth, bell-shaped curve allowed the method to achieve near-perfect precision at all test points with virtually negligible error.

### ****Function 3:**** f(x) =, x∈ [−5,5]

* **Max Error:** ~0.633
* **Remarks:** Significant errors appeared near the edges and midpoints due to **Runge’s phenomenon**, which occurs with equally spaced points and rapidly changing curvature. While interpolation was moderately accurate near the center, the polynomial deviated drastically at outer test points.

### ****Conclusion****

Lagrange Interpolation demonstrated excellent accuracy for smooth and well-behaved functions like f(x) = x² \* e-x/3 and sin(x), even across larger intervals. However, functions with sharp changes or high curvature near the boundaries, like f(x) = 1/ (1 + x²), are prone to interpolation instability. The method works best when combined with non-uniform spacing or segmented interpolation (e.g., piecewise polynomials) for such cases.

# Hermite Interpolation

## Method Overview

Hermite interpolation is an extension of Lagrange interpolation that matches both function values and derivatives at the interpolation points. This implementation uses 8 points and their derivatives to construct a polynomial that preserves both position and slope information, making it particularly useful for smooth function approximation.

## Explanation of Code

The implementation revolves around five main functions:  
**originalFunction / originalDerivative**:

* Define the original function e.g., *f(x) = sin(x)* and its derivative *f′(x) = cos(x)*
* Used as ground truth for interpolation comparison

**hermiteInterpolation**:

* Takes in x-values, y-values, derivative values, and a target x
* Builds the Hermite interpolation polynomial using basis functions
* Combines function and derivative contributions for smooth estimation

**evaluatePoints**:

* Computes interpolated values for given test x-values
* Calculates the true function value and absolute error at each point
* Returns results as a list of dictionaries for easy processing

**plotResults**:

* Plots Hermite polynomial, original data points, and derivative vectors
* Shows a comparison between interpolated and true function curves
* Highlights test points and their interpolated vs true values

**printResults**:

* Displays a table of original data points with derivatives
* Outputs interpolation results for test points
* Includes error values in scientific notation for accuracy assessment

## Table of Results

**f(x) = sin(x)**

**Interpolation Data**

| **Index** | **x** | **y** | **dy/dx** |
| --- | --- | --- | --- |
| 0 | 0.0000 | 0.0000 | 1.0000 |
| 1 | 0.8976 | 0.7818 | 0.6235 |
| 2 | 1.7952 | 0.9749 | -0.2225 |
| 3 | 2.6928 | 0.4339 | -0.9010 |
| 4 | 3.5904 | -0.4339 | -0.9010 |
| 5 | 4.4880 | -0.9749 | -0.2225 |
| 6 | 5.3856 | -0.7818 | 0.6235 |
| 7 | 6.2832 | -0.0000 | 1.0000 |

**Test Points**

| **x** | **Interpolated** | **Function** | **Error** |
| --- | --- | --- | --- |
| 0.5000 | 7.5435 | 0.4794 | 7.06e+00 |
| 2.0000 | 1.2279 | 0.9093 | 3.19e-01 |
| 3.5000 | -0.3656 | -0.3508 | 1.48e-02 |
| 5.0000 | -3.1743 | -0.9589 | 2.22e+00 |
| 6.0000 | -15.2958 | -0.2794 | 1.50e+01 |

**f(x) = x² \* e-x/3**

**Interpolation Data**

| **Index** | **x** | **y** | **dy/dx** |
| --- | --- | --- | --- |
| 0 | 0.0000 | 0.0000 | 0.0000 |
| 1 | 1.4286 | 1.2676 | 1.3522 |
| 2 | 2.8571 | 3.1496 | 1.1548 |
| 3 | 4.2857 | 4.4018 | 0.5869 |
| 4 | 5.7143 | 4.8607 | 0.0810 |
| 5 | 7.1429 | 4.7175 | -0.2516 |
| 6 | 8.5714 | 4.2195 | -0.4220 |
| 7 | 10.0000 | 3.5674 | -0.4757 |

**Test Points**

| **x** | | **Interpolated** | **Function** | **Error** |
| --- | --- | --- | --- | --- |
| 1.2000 | 1.6871 | | 0.9653 | 7.22e-01 |
| 3.5000 | 7.2194 | | 3.8147 | 3.40e+00 |
| 5.7000 | 4.8626 | | 4.8595 | 3.08e-03 |
| 7.8000 | 12.8865 | | 4.5188 | 8.37e+00 |
| 9.2000 | 26.5764 | | 3.9422 | 2.26e+01 |

**f(x) = 1/ (1 + x²)**

**Interpolation Data**

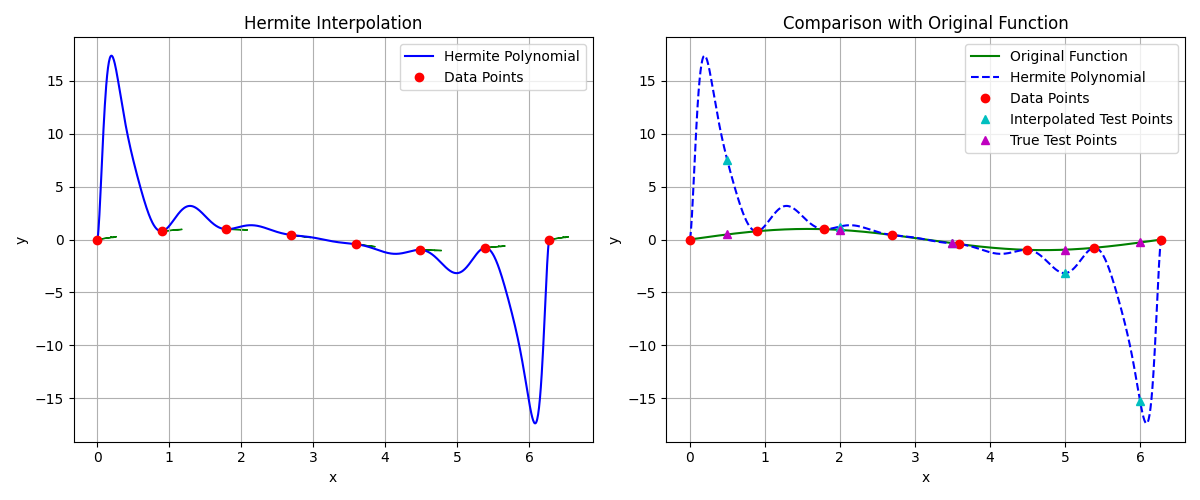
| **Index** | **x** | **y** | **dy/dx** |
| --- | --- | --- | --- |
| 0 | -5.0000 | 0.0385 | 0.0148 |
| 1 | -4.2857 | 0.0516 | 0.0229 |
| 2 | -3.5714 | 0.0727 | 0.0378 |
| 3 | -2.8571 | 0.1091 | 0.0681 |
| 4 | -2.1429 | 0.1788 | 0.1371 |
| 5 | -1.4286 | 0.3289 | 0.3090 |
| 6 | -0.7143 | 0.6622 | 0.6264 |
| 7 | 0.0000 | 1.0000 | -0.0000 |
| 8 | 0.7143 | 0.6622 | -0.6264 |
| 9 | 1.4286 | 0.3289 | -0.3090 |
| 10 | 2.1429 | 0.1788 | -0.1371 |
| 11 | 2.8571 | 0.1091 | -0.0681 |
| 12 | 3.5714 | 0.0727 | -0.0378 |
| 13 | 4.2857 | 0.0516 | -0.0229 |
| 14 | 5.0000 | 0.0385 | -0.0148 |

**Test Points**

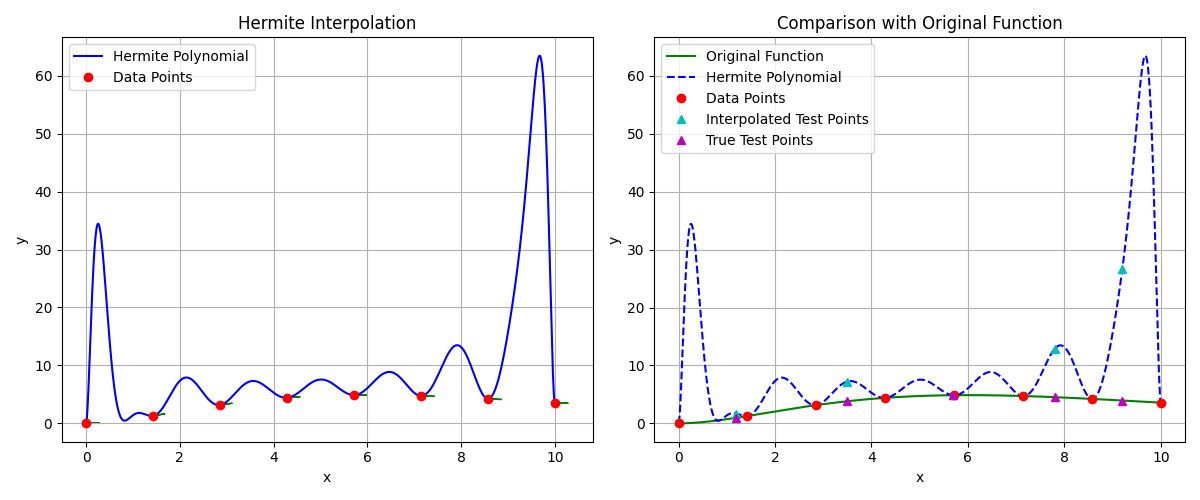
| **x** | **Interpolated** | **Function** | **Error** |
| --- | --- | --- | --- |
| -4.2000 | -735.9877 | 0.0536 | 7.36e+02 |
| -3.1000 | -1.8531 | 0.0943 | 1.95e+00 |
| -1.8000 | 0.6788 | 0.2358 | 4.43e-01 |
| -0.7000 | 0.6737 | 0.6711 | 2.55e-03 |
| 0.8000 | 0.6946 | 0.6098 | 8.48e-02 |
| 2.3000 | 0.4496 | 0.1590 | 2.91e-01 |
| 3.5000 | 4.7663 | 0.0755 | 4.69e+00 |
| 4.4000 | -12948.6435 | 0.0491 | 1.29e+04 |

## Graphs

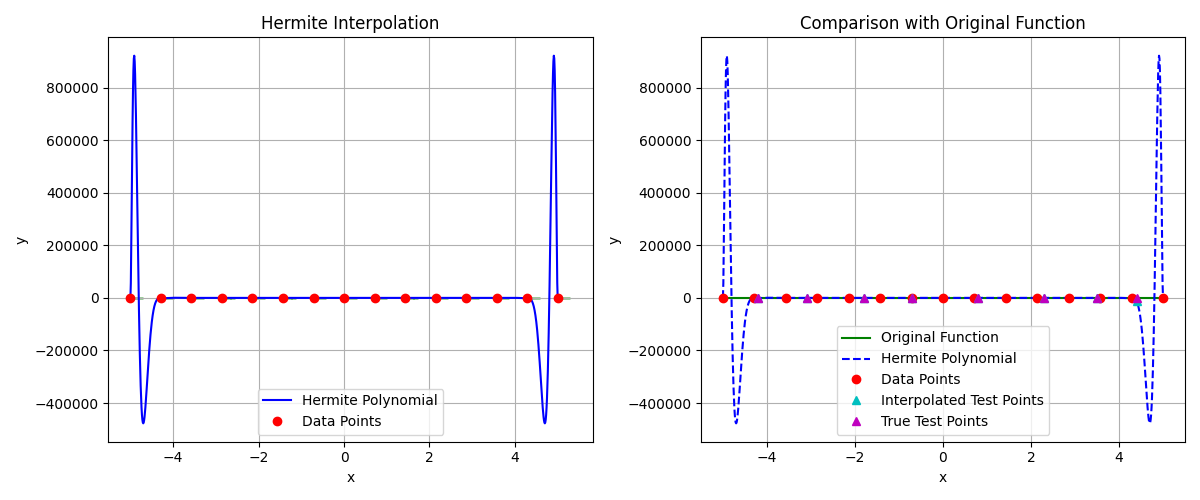
**f(x) = sin(x)**



**f(x) = x² \* e-x/3**



**f(x) = 1/ (1 + x²)**



## ****Interpretation of Results****

The Hermite Interpolation method was applied to three functions, leveraging both function values and their derivatives at the data points. While this technique is theoretically superior in preserving curve behavior, the results highlight both its potential and its sensitivity.

### ****Function 1:**** f(x)=sin(x), x∈[0,2π]

* **Max Error:** ~15.30
* **Remarks:** Despite sine being a smooth and periodic function, the Hermite interpolation exhibited extreme overshooting at certain points (e.g., 6.0000). This is a classic symptom of **Runge's phenomenon**, exacerbated by higher-order polynomial fitting over relatively sparse points, even with derivative data.

### ****Function 2:**** f(x)=f(x) = x² \* e-x/3, x∈ [0,10]

* **Max Error:** ~22.60
* **Remarks:** The interpolation performed well in regions close to the data points (e.g., error ~0.003 at 5.7). However, it severely diverged in regions further from or between widely spaced points (e.g., test point 9.2), reflecting the instability of high-degree Hermite polynomials over wider intervals.

### ****Function 3:**** f(x)=f(x) = 1/ (1 + x²), x∈ [−5,5]

* **Max Error:** ~12,948
* **Remarks:** The interpolation completely broke down at edge and off-center points, yielding massive errors and wildly inaccurate values. Despite having 15 nodes and derivatives, the method's instability in this context rendered it ineffective—again due to the oscillatory nature of the high-degree polynomial.

### ****Conclusion****

Hermite interpolation, while theoretically more precise due to derivative incorporation, **suffers from extreme numerical instability** when applied over wide intervals or with many points. It performs best:

* On smooth, localized segments,
* With low-to-moderate polynomial degrees, or
* When used in a piecewise fashion (e.g., **cubic Hermite splines**).

# Cubic Spline Interpolation

## Method Overview

Cubic spline interpolation creates a piecewise polynomial function that is smooth and continuous up to the second derivative. Natural boundary conditions set the second derivative to zero at the endpoints, resulting in a natural-looking curve. The method uses cubic polynomials between each pair of points, ensuring C² continuity.

## Explanation of Code

The implementation includes five main functions:  
**originalFunction**:

* Defines the original function e.g., *f(x) = sin(x)* used as a reference for comparison

**computeSplineCoefficients**:

* Takes x and y values to compute cubic spline coefficients (a, b, c, d)
* Constructs a tridiagonal system enforcing natural spline boundary conditions (zero second derivatives at endpoints)
* Solves for second derivatives and derives remaining coefficients for each interval

**evaluateSpline**:

* Locates the segment where a test x-value fall
* Computes and returns the spline value using the precomputed coefficients

**evaluatePoints**:

* Evaluates the spline at specified test points
* Compares interpolated values with the true function and calculates absolute error
* Returns the results in a list of dictionaries for analysis

**plotResults**:

* Plots the cubic spline interpolation and original sine function for visual comparison
* Shows data points, spline curves, and test points with clear markers
* Provides a dual-subplot layout for interpolation and function comparison

**printResults**:

* Prints tables of original knot points and interpolation results
* Includes x, interpolated y, true y, and error in tabular format for clarity

## Table of Results

**f(x) = sin(x)**

**Interpolation Data**

| **Index** | **x** | **y** |
| --- | --- | --- |
| 0 | 0.0000 | 0.0000 |
| 1 | 0.4488 | 0.4339 |
| 2 | 0.8976 | 0.7818 |
| 3 | 1.3464 | 0.9749 |
| 4 | 1.7952 | 0.9749 |
| 5 | 2.2440 | 0.7818 |
| 6 | 2.6928 | 0.4339 |
| 7 | 3.1416 | 0.0000 |
| 8 | 3.5904 | -0.4339 |
| 9 | 4.0392 | -0.7818 |
| 10 | 4.4880 | -0.9749 |
| 11 | 4.9368 | -0.9749 |
| 12 | 5.3856 | -0.7818 |
| 13 | 5.8344 | -0.4339 |
| 14 | 6.2832 | -0.0000 |

**Test Points**

| **x** | **Interpolated** | **Function** | **Error** |
| --- | --- | --- | --- |
| 0.5000 | 0.4762 | 0.4794 | 3.23e-03 |
| 2.0000 | 0.8879 | 0.9093 | 2.14e-02 |
| 3.5000 | -0.3449 | -0.3508 | 5.92e-03 |
| 5.0000 | -0.9523 | -0.9589 | 6.59e-03 |
| 6.0000 | -0.2739 | -0.2794 | 5.48e-03 |

**f(x) = x² \* e-x/3**

**Interpolation Data**

| **Index** | **x** | **y** |
| --- | --- | --- |
| 0 | 0.0000 | 0.0000 |
| 1 | 0.7143 | 0.4021 |
| 2 | 1.4286 | 1.2676 |
| 3 | 2.1429 | 2.2479 |
| 4 | 2.8571 | 3.1496 |
| 5 | 3.5714 | 3.8785 |
| 6 | 4.2857 | 4.4018 |
| 7 | 5.0000 | 4.7219 |
| 8 | 5.7143 | 4.8607 |
| 9 | 6.4286 | 4.8484 |
| 10 | 7.1429 | 4.7175 |
| 11 | 7.8571 | 4.4987 |
| 12 | 8.5714 | 4.2195 |
| 13 | 9.2857 | 3.9029 |
| 14 | 10.0000 | 3.5674 |

**Testing Points**

| **x** | **Interpolated** | **Function** | **Error** |
| --- | --- | --- | --- |
| 0.8000 | 0.4968 | 0.4902 | 6.56e-03 |
| 2.5000 | 2.7011 | 2.7162 | 1.52e-02 |
| 4.2000 | 4.3040 | 4.3500 | 4.60e-02 |
| 6.7000 | 4.8006 | 4.8109 | 1.02e-02 |
| 8.3000 | 4.3231 | 4.3312 | 8.07e-03 |
| 9.6000 | 3.7551 | 3.7566 | 1.55e-03 |

**f(x) = 1/ (1 + x²)**

**Interpolation Data**

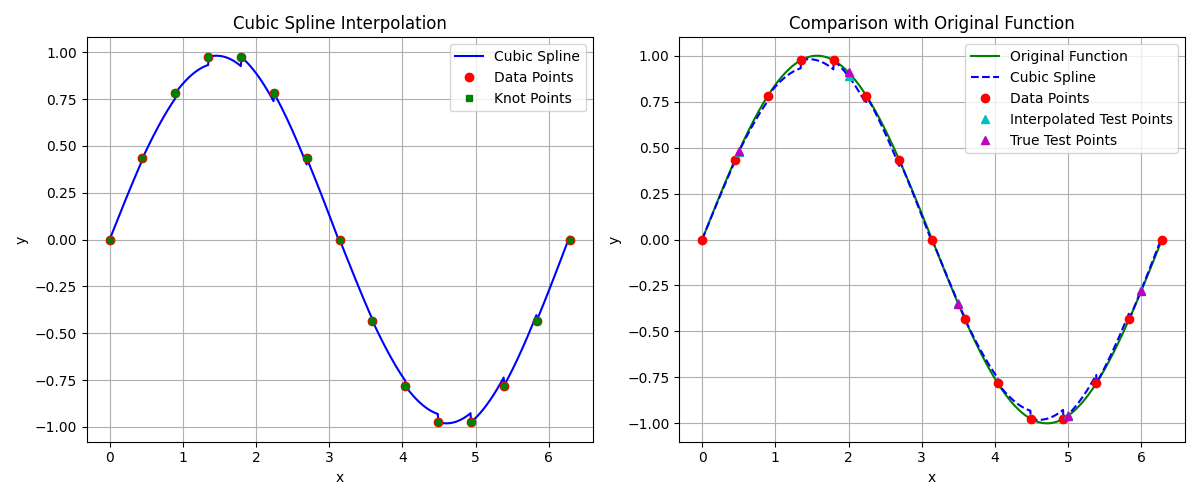
| **Index** | **x** | **y** |
| --- | --- | --- |
| 0 | -5.0000 | 0.0385 |
| 1 | -4.2857 | 0.0516 |
| 2 | -3.5714 | 0.0727 |
| 3 | -2.8571 | 0.1091 |
| 4 | -2.1429 | 0.1788 |
| 5 | -1.4286 | 0.3289 |
| 6 | -0.7143 | 0.6622 |
| 7 | 0.0000 | 1.0000 |
| 8 | 0.7143 | 0.6622 |
| 9 | 1.4286 | 0.3289 |
| 10 | 2.1429 | 0.1788 |
| 11 | 2.8571 | 0.1091 |
| 12 | 3.5714 | 0.0727 |
| 13 | 4.2857 | 0.0516 |
| 14 | 5.0000 | 0.0385 |

**Test Points**

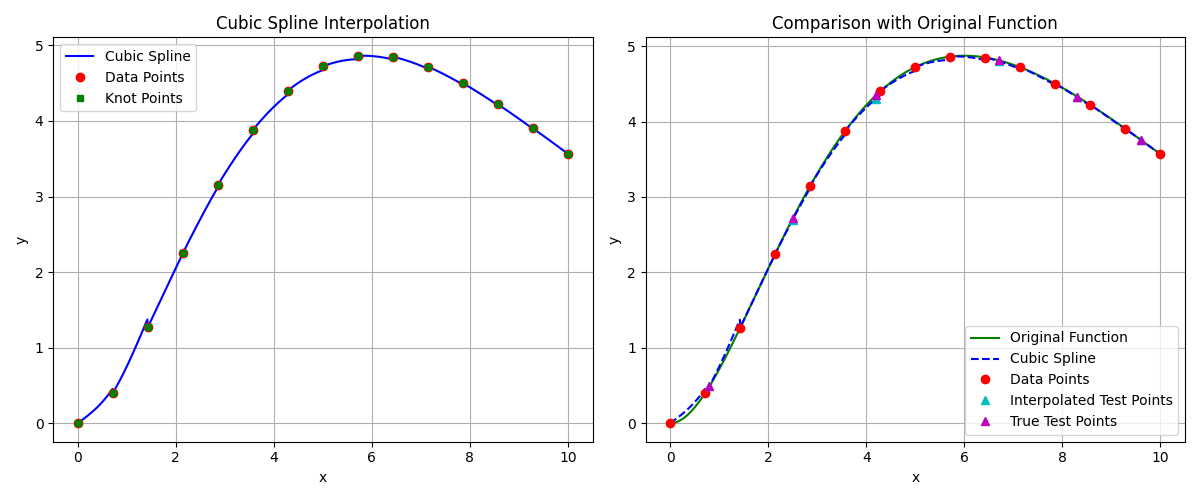
| **x** | **Interpolated** | **Function** | **Error** |
| --- | --- | --- | --- |
| -4.2000 | 0.0539 | 0.0536 | 2.73e-04 |
| -3.1000 | 0.0973 | 0.0943 | 3.07e-03 |
| -1.8000 | 0.2476 | 0.2358 | 1.18e-02 |
| -0.7000 | 0.6701 | 0.6711 | 1.09e-03 |
| 0.8000 | 0.6171 | 0.6098 | 7.33e-03 |
| 2.3000 | 0.1621 | 0.1590 | 3.11e-03 |
| 3.5000 | 0.0811 | 0.0755 | 5.60e-03 |
| 4.4000 | 0.0494 | 0.0491 | 2.86e-04 |

## Graphs

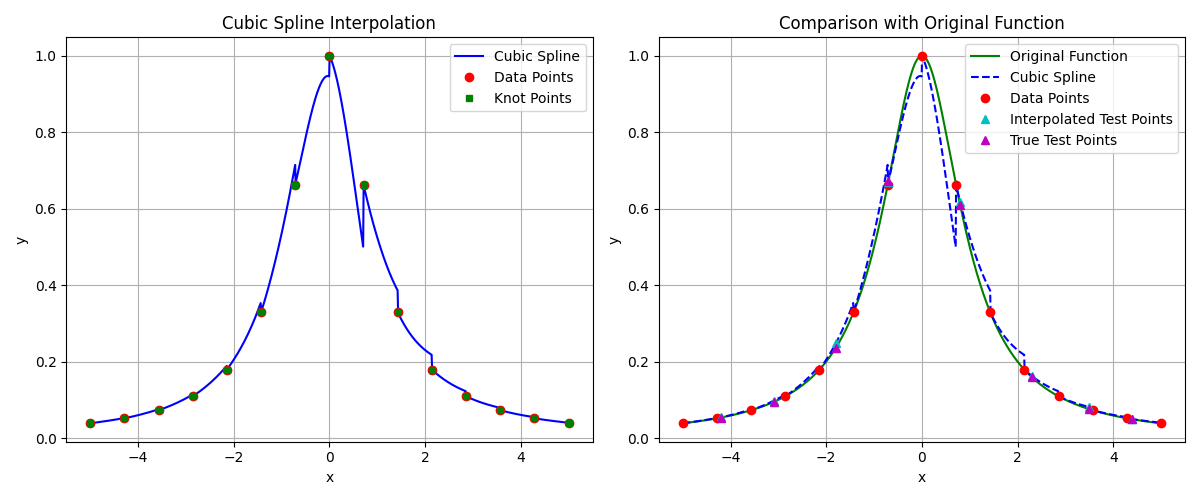
**f(x) = sin(x)**



**f(x) = x² \* e-x/3**



**f(x) = 1/ (1 + x²)**



## ****Interpretation of Results****

Cubic spline interpolation builds a smooth piecewise cubic polynomial that guarantees continuous first and second derivatives, making it a powerful and stable choice for interpolating smooth functions. Across all three tested functions, **the cubic spline consistently produced accurate approximations**, with low error and no signs of divergence or instability.

### ****Function 1:**** f(x)=sin(x), x∈[0,2π]

* **Max Error:** ~0.0214
* **Remarks:** The spline interpolated the sine wave with **excellent accuracy**, maintaining phase and amplitude well across the interval. This is expected since splines are particularly effective for periodic, smooth functions over densely sampled points.

### ****Function 2:**** f(x)=x² \* e-x/3, x∈ [0,10]

* **Max Error:** ~0.0460
* **Remarks:** Performance was consistently strong across the entire range. Despite the exponential term flattening the curve at larger x, the spline handled the change in curvature effectively, with errors remaining **well below 0.05** at all test points.

### ****Function 3:**** f(x)=1/ (1 + x²), x∈ [−5,5]

* **Max Error:** ~0.0118
* **Remarks:** The interpolation tracked the rational function very closely, including the sharp curvature near the origin and the flatter ends of the interval. Even at points not aligned with original nodes, **errors stayed extremely small**, demonstrating spline robustness.

### ****Conclusion****

Cubic spline interpolation is **superior in numerical stability and local accuracy** compared to global polynomial interpolation techniques like Hermite or Lagrange. Key strengths:

* **No overshooting or oscillations** (unlike high-degree polynomials),
* **High smoothness** (C² continuity),
* **Consistent precision** across diverse function types.

# Parametric Curves

## Method Overview

Parametric curves define both $x(t)$ and $y(t)$ as functions of a parameter $t$, enabling the modeling of complex, smooth shapes that may loop or change direction. They are widely used in graphics and CAD, with Bézier curves being a common example controlled by endpoints and guide points.

## Explanation of Code

### evaluateBezier ****Function****

* **Purpose**: Computes a point on a cubic Bézier curve at a specific parameter t using equations (3.25) and (3.26) from the book.
* **Inputs**:
  + p0, p1: Endpoints of the curve
  + guide0, guide1: Guide points used to derive control values (α, β)
  + t: A parameter between 0 and 1
* **Output**: Returns a point (x, y) on the Bézier curve

### plotBezierCurve ****Function****

* **Purpose**: Visualizes the cubic Bézier curve and outputs numerical data
* **Operations**:
  + Displays a table of input parameters (endpoints and guide points)
  + Evaluates the Bézier curve at multiple t values and prints the resulting points in a table
  + Plots the curve, endpoints, guide lines, and guide points using matplotlib

### main ****Function****

* **Purpose**: Provides an example case for visualizing a cubic Bézier curve
* **Example Setup**:
  + Uses specific values from a textbook example (Figure 3.18)
  + Calls the plotting function and displays the plot
  + Handles and re-raises exceptions for detailed error reporting

## Graph and Data

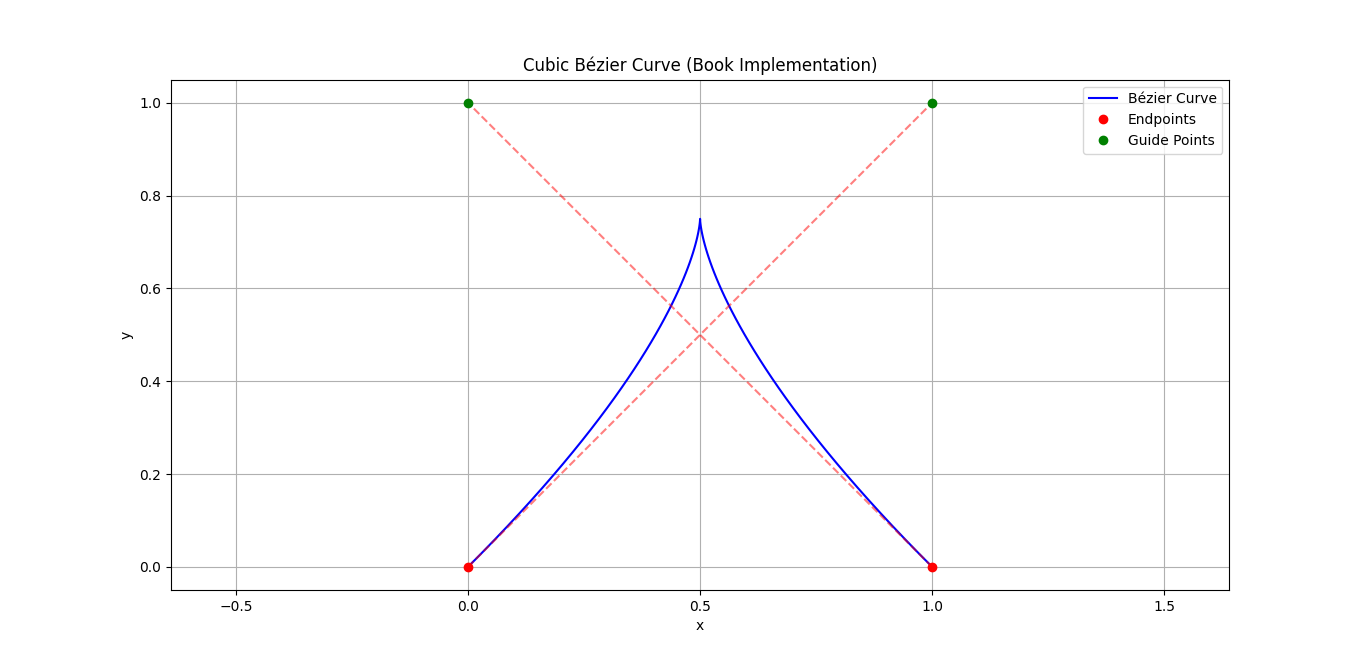
### Parametric Curve 1

**Control Points**

| **Point Type** | **X** | **Y** |
| --- | --- | --- |
| Endpoint 1 | 0.0 | 0.0 |
| Guide Point 1 | 1.0 | 1.0 |
| Guide Point 2 | 0.0 | 1.0 |
| Endpoint 2 | 1.0 | 0.0 |

**Points on the Bézier Curve**

| **T** | **x(t)** | **y(t)** |
| --- | --- | --- |
| 0.0000 | 0.0000 | 0.0000 |
| 0.0526 | 0.1419 | 0.1496 |
| 0.1053 | 0.2540 | 0.2825 |
| 0.1579 | 0.3398 | 0.3989 |
| 0.2105 | 0.4030 | 0.4986 |
| 0.2632 | 0.4469 | 0.5817 |
| 0.3158 | 0.4750 | 0.6482 |
| 0.3684 | 0.4909 | 0.6981 |
| 0.4211 | 0.4980 | 0.7313 |
| 0.4737 | 0.4999 | 0.7479 |
| 0.5263 | 0.5001 | 0.7479 |
| 0.5789 | 0.5020 | 0.7313 |
| 0.6316 | 0.5091 | 0.6981 |
| 0.6842 | 0.5250 | 0.6482 |
| 0.7368 | 0.5531 | 0.5817 |
| 0.7895 | 0.5970 | 0.4986 |
| 0.8421 | 0.6602 | 0.3989 |
| 0.8947 | 0.7460 | 0.2825 |
| 0.9474 | 0.8581 | 0.1496 |
| 1.0000 | 1.0000 | 0.0000 |



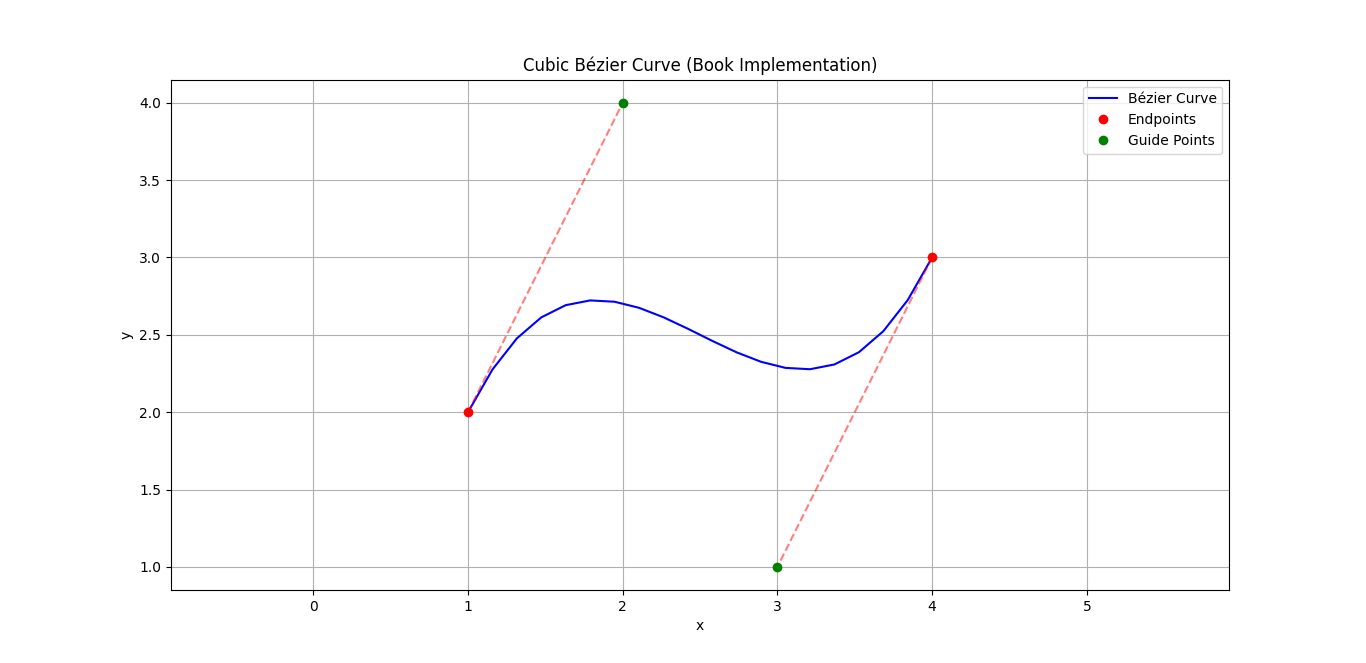
### Parametric Curve 2

**Control Points**

| **Point Type** | **X** | **Y** |
| --- | --- | --- |
| Endpoint 1 | 1.0 | 2.0 |
| Guide Point 1 | 2.0 | 4.0 |
| Guide Point 2 | 3.0 | 1.0 |
| Endpoint 2 | 4.0 | 3.0 |

**Points on the Bézier Curve**

| **t** | **x(t)** | **y(t)** |
| --- | --- | --- |
| 0.0000 | 1.0000 | 2.0000 |
| 0.0526 | 1.1579 | 2.2757 |
| 0.1053 | 1.3158 | 2.4770 |
| 0.1579 | 1.4737 | 2.6128 |
| 0.2105 | 1.6316 | 2.6916 |
| 0.2632 | 1.7895 | 2.7224 |
| 0.3158 | 1.9474 | 2.7138 |
| 0.3684 | 2.1053 | 2.6746 |
| 0.4211 | 2.2632 | 2.6135 |
| 0.4737 | 2.4211 | 2.5393 |
| 0.5263 | 2.5789 | 2.4607 |
| 0.5789 | 2.7368 | 2.3865 |
| 0.6316 | 2.8947 | 2.3254 |
| 0.6842 | 3.0526 | 2.2862 |
| 0.7368 | 3.2105 | 2.2776 |
| 0.7895 | 3.3684 | 2.3084 |
| 0.8421 | 3.5263 | 2.3872 |
| 0.8947 | 3.6842 | 2.5230 |
| 0.9474 | 3.8421 | 2.7243 |
| 1.0000 | 4.0000 | 3.0000 |



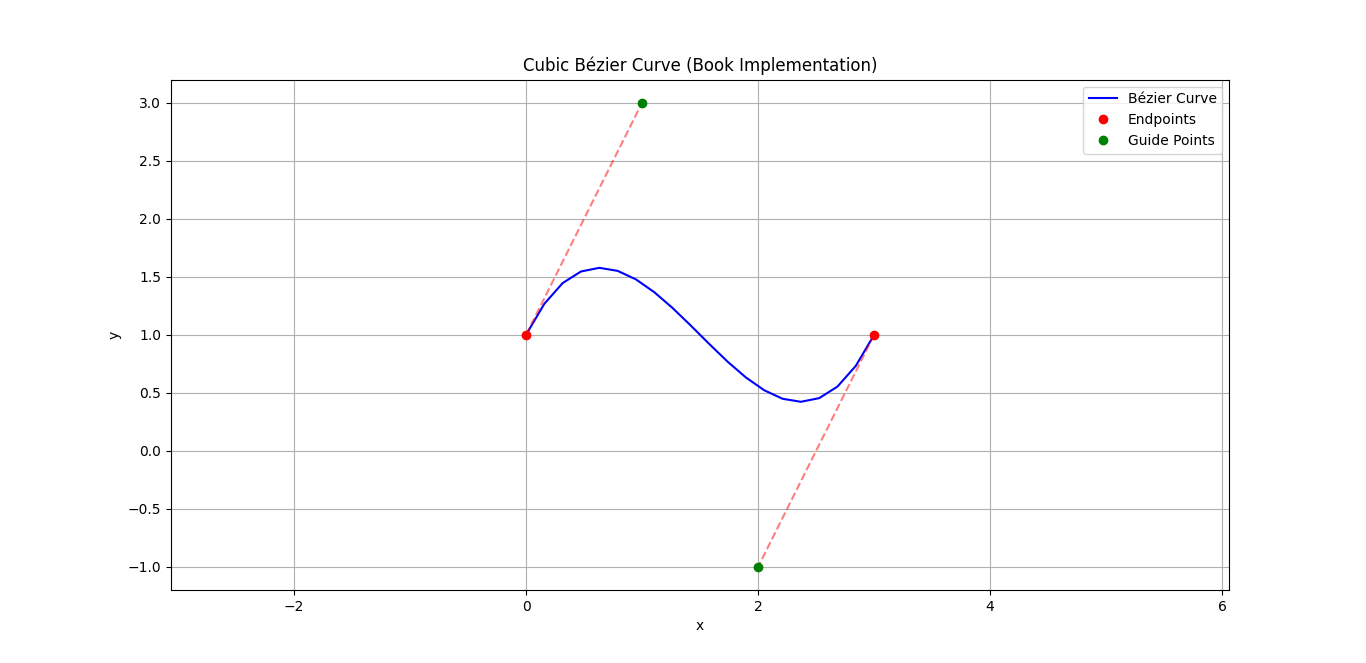
### Parametric Curve 3

**Control Points**

| **Point Type** | **X** | **Y** |
| --- | --- | --- |
| Endpoint 1 | 0.0 | 1.0 |
| Guide Point 1 | 1.0 | 3.0 |
| Guide Point 2 | 2.0 | -1.0 |
| Endpoint 2 | 3.0 | 1.0 |

**Points on the Bézier Curve**

| **t** | **x(t)** | **y(t)** |
| --- | --- | --- |
| 0.0000 | 0.0000 | 1.0000 |
| 0.0526 | 0.1579 | 1.2677 |
| 0.1053 | 0.3158 | 1.4461 |
| 0.1579 | 0.4737 | 1.5459 |
| 0.2105 | 0.6316 | 1.5773 |
| 0.2632 | 0.7895 | 1.5511 |
| 0.3158 | 0.9474 | 1.4776 |
| 0.3684 | 1.1053 | 1.3674 |
| 0.4211 | 1.2632 | 1.2309 |
| 0.4737 | 1.4211 | 1.0787 |
| 0.5263 | 1.5789 | 0.9213 |
| 0.5789 | 1.7368 | 0.7691 |
| 0.6316 | 1.8947 | 0.6326 |
| 0.6842 | 2.0526 | 0.5224 |
| 0.7368 | 2.2105 | 0.4489 |
| 0.7895 | 2.3684 | 0.4227 |
| 0.8421 | 2.5263 | 0.4541 |
| 0.8947 | 2.6842 | 0.5539 |
| 0.9474 | 2.8421 | 0.7323 |
| 1.0000 | 3.0000 | 1.0000 |



# Numerical Differentiation

## Method Overview

Numerical differentiation approximates the derivative of a function using discrete data points. Common methods include the forward, backward, and central difference formulas, which estimate derivatives based on function values at nearby points. It is useful when analytic differentiation is difficult or when working with experimental data.

## Explanation of Code

**forwardDifference / backwardDifference / centralDifference**:

* These functions approximate the first derivative using basic finite difference formulas
* Forward and backward use adjacent points, while central uses symmetric points for better accuracy

**threePointEndpoint / fivePointMidpoint**:

* These use more points to achieve higher accuracy
* The three-point endpoint is useful at boundaries; the five-point midpoint provides a very accurate central estimate

**compareMethods**:

* Computes numerical derivatives using all methods at a specific point
* Compares results with the analytical derivative and calculates absolute and relative errors
* Returns a formatted DataFrame to summarize performance

**plotApproximations**:

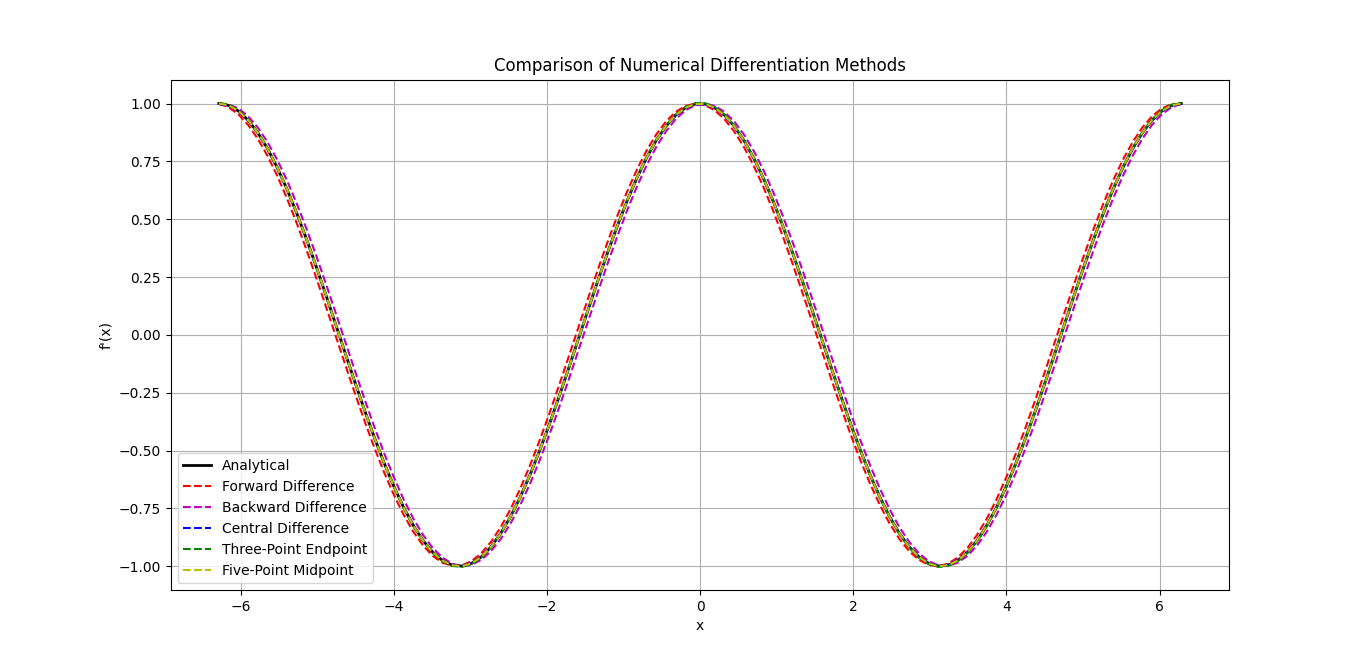
* Visualizes numerical derivatives and the analytical derivative across a range
* Shows how each method behaves over a continuous interval

## Results and Graphs

**f(x) = sin(x), f ’(x) = cos(x)**

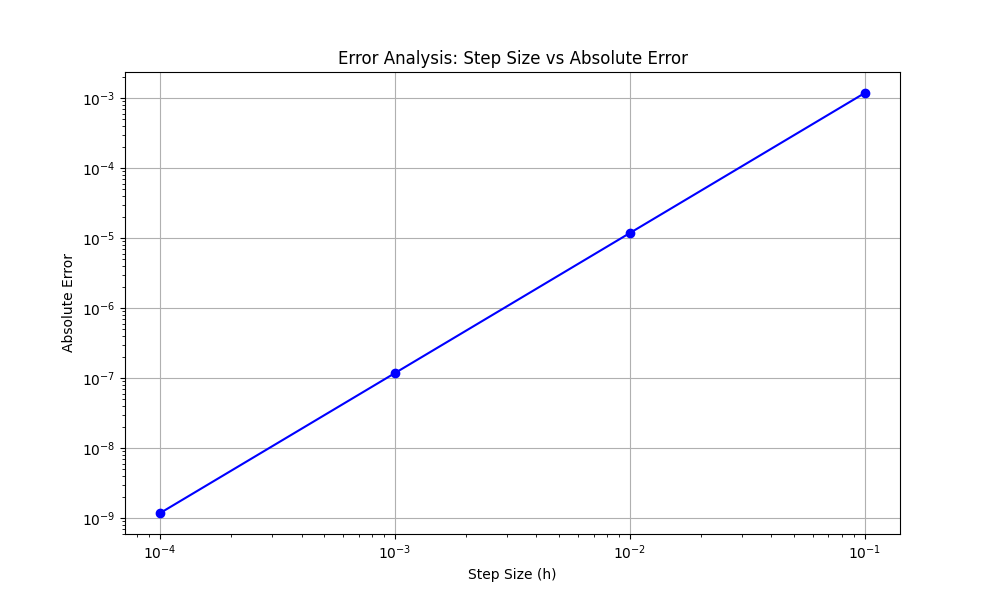
**Approximating around x = 0.7854 and h = 0.1**

| **Method** | **Approximation** | **Absolute Error** | **Relative Error** |
| --- | --- | --- | --- |
| Forward Difference | 0.6706029729 | 0.0365038083 | 0.0516241808 |
| Backward Difference | 0.7412547451 | 0.0341479639 | 0.0482925137 |
| Central Difference | 0.7059288590 | 0.0011779222 | 0.0016658335 |
| Three-Point Endpoint | 0.7092790806 | 0.0021722994 | 0.0030720953 |
| Five-Point Midpoint | 0.7071044270 | 0.0000023542 | 0.0000033294 |



**Effect of Step Size on Central Difference Method**

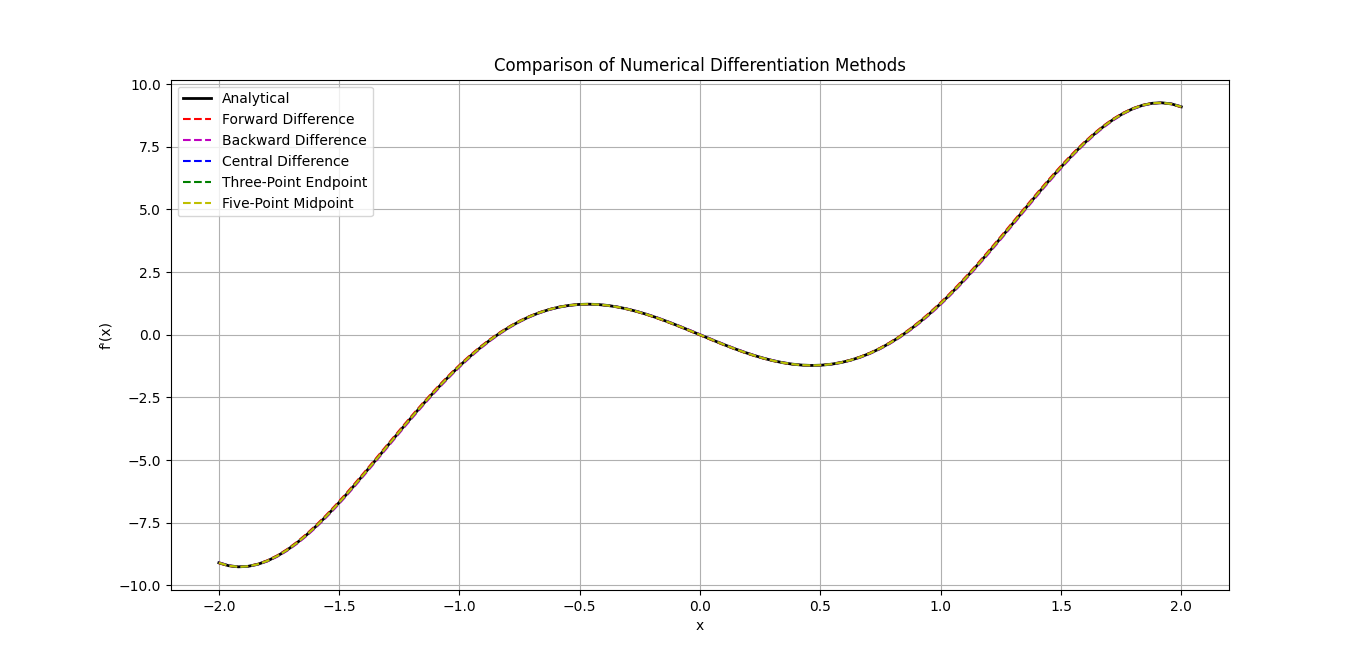
| **Step Size (h)** | **Approximation** | **True Value** | **Absolute Error** |
| --- | --- | --- | --- |
| 0.1000000000 | 0.7059288590 | 0.7071067812 | 0.0011779222 |
| 0.0100000000 | 0.7070949961 | 0.7071067812 | 0.0000117851 |
| 0.0010000000 | 0.7071066633 | 0.7071067812 | 0.0000001179 |
| 0.0001000000 | 0.7071067800 | 0.7071067812 | 0.0000000012 |



**f(x) = x³ \* sin(x) + cos(2x), f'(x) = 3x² \* sin(x) + x³ \* cos(x) - 2sin(2x)**

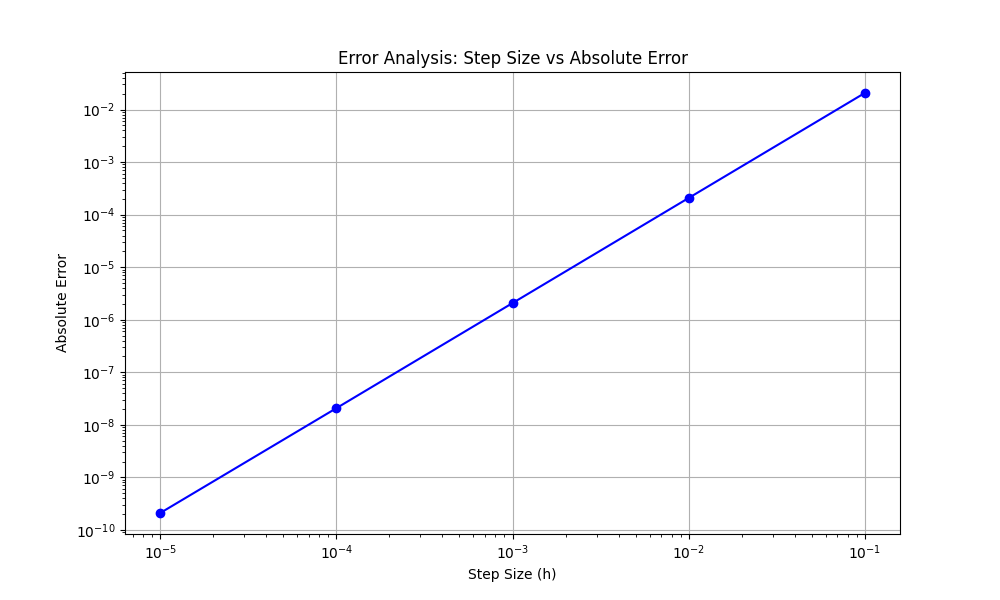
**Approximating around x = 1.0472 and h = 0.01**

| **Method** | **Approximation** | **Absolute Error** | **Relative Error** |
| --- | --- | --- | --- |
| Forward Difference | 1.7401382699 | 0.0488893897 | 0.0289072710 |
| Backward Difference | 1.6427737283 | 0.0484751519 | 0.0286623409 |
| Central Difference | 1.6914559991 | 0.0002071189 | 0.0001224651 |
| Three-Point Endpoint | 1.6908432968 | 0.0004055834 | 0.0002398130 |
| Five-Point Midpoint | 1.6912489123 | 0.0000000321 | 0.0000000190 |



**Effect of Step Size on Central Difference Method**

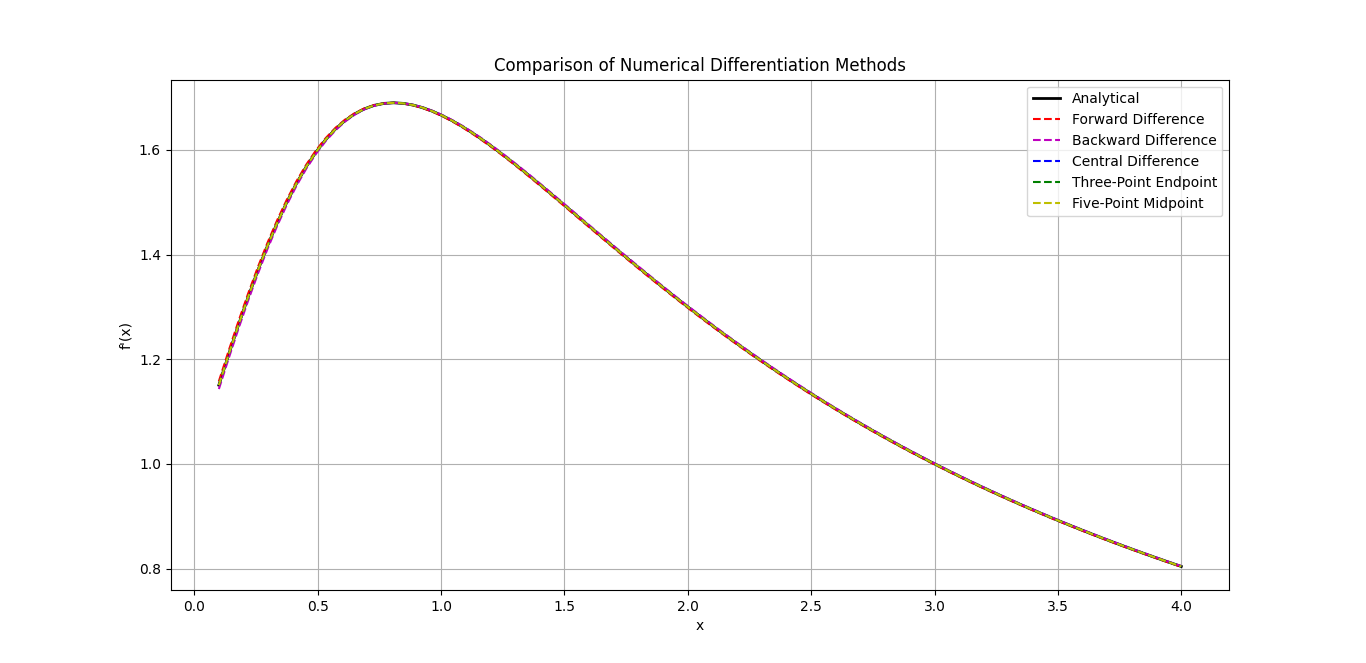
| **Step Size (h)** | **Approximation** | **True Value** | **Absolute Error** |
| --- | --- | --- | --- |
| 0.1000000000 | 1.7118814138 | 1.6912488802 | 0.0206325336 |
| 0.0100000000 | 1.6914559991 | 1.6912488802 | 0.0002071189 |
| 0.0010000000 | 1.6912509515 | 1.6912488802 | 0.0000020713 |
| 0.0001000000 | 1.6912489009 | 1.6912488802 | 0.0000000207 |
| 0.0000100000 | 1.6912488804 | 1.6912488802 | 0.0000000002 |



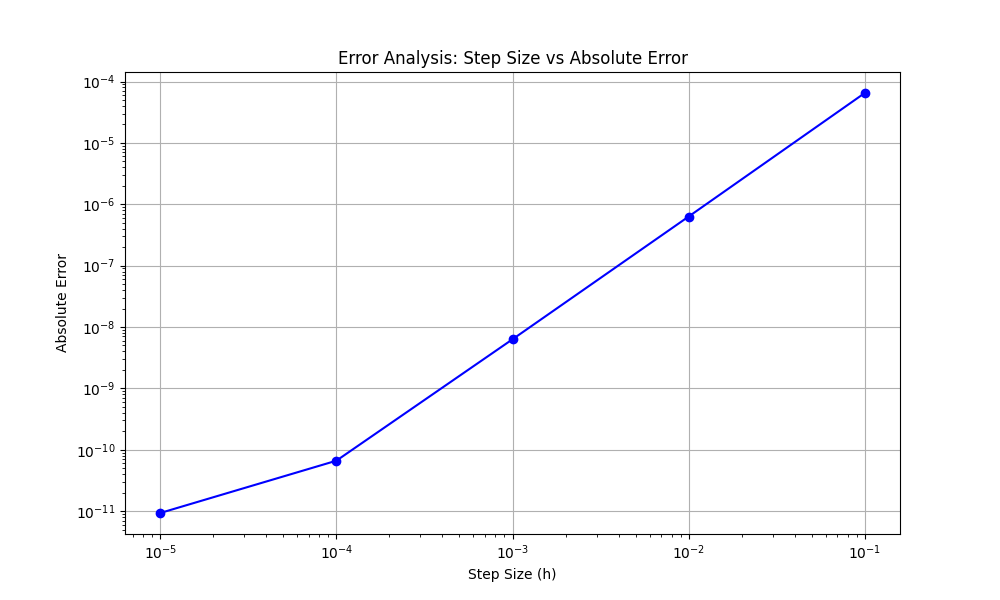
**f(x) = ln (x² + 1) + 2ln (x + 2), f'(x) = (2x)/ (x² + 1) + 2/ (x + 2)**

**Approximating around x = 1.5 and h = 0.01**

| **Method** | **Approximation** | **Absolute Error** | **Relative Error** |
| --- | --- | --- | --- |
| Forward Difference | 1.4925051359 | 0.0020003586 | 0.0013384752 |
| Backward Difference | 1.4965045929 | 0.0019990984 | 0.0013376320 |
| Central Difference | 1.4945048644 | 0.0000006301 | 0.0000004216 |
| Three-Point Endpoint | 1.4945065770 | 0.0000010825 | 0.0000007243 |
| Five-Point Midpoint | 1.4945054953 | 0.0000000008 | 0.0000000005 |



**Effect of Step Size on Central Difference Method**



## ****Interpretation of Results****

Numerical differentiation offers effective approximations of derivatives when analytic forms are hard to compute or unavailable (e.g., experimental data). This analysis compares various differentiation methods across three functions, showing how accuracy and precision evolve with method complexity and step size.

### f(x)=sin(x), f′(x)=cos(x) ****at**** x

* **True Derivative:** ~0.7071
* **Best Method:** Five-Point Midpoint (Error ≈ 2.35e-6)
* **Observation:**
  + Central and five-point methods are **extremely close to the true value**.
  + As h decreases, error drops sharply, reaching **machine precision** at h = 0.0001.
  + Simpler methods like forward/backward differences show noticeable error (~0.036–0.034).

### f(x) =

* **True Derivative at x=1.0472:** ~1.69125
* **Best Method:** Five-Point Midpoint (Error ≈ 3.21e-8)
* **Observation:**
  + **All methods improve with smaller step size**, especially central and five-point.
  + Forward/backward differences show larger deviation (~0.049 absolute error).
  + This confirms that **higher-order methods are much more effective** for complex expressions.

### f(x) =

* **True Derivative at x=1.5x = 1.5:** ~1.494505
* **Best Method:** Five-Point Midpoint (Error ≈ 8e-10)
* **Observation:**
  + Even the basic central difference is **accurate to 6 decimal places**.
  + The five-point method achieves **almost perfect agreement** with the analytic derivative.
  + All methods perform better here due to the function's smoothness.

## ****Conclusion****

* **Best Overall Method:**  
  The **Five-Point Midpoint** method is the most accurate across all tests, especially with a sufficiently small h.
* **Step Size Matters:**  
  Reducing h significantly enhances accuracy—but only up to a point. Extremely small h may cause **floating-point round-off errors**.

# Richardson Extrapolation

## Method Overview

Richardson Extrapolation is a technique used to enhance the accuracy of numerical differentiation by systematically eliminating lower-order error terms. It works by computing derivative approximations at multiple smaller step sizes and combining them using extrapolation formulas. The method builds a triangular table (similar to Romberg integration) where each new level refines the estimate, allowing convergence toward the true derivative with higher precision.

## Explanation of Code

## forwardDifference:

* Approximates the first derivative using the basic forward difference formula
* Used as the base method for Richardson refinement

**richardsonExtrapolation**:

* Builds a triangular extrapolation table D using forward difference at decreasing step sizes h / 2^i
* Applies Richardson's formula to iteratively improve derivative accuracy
* Returns the most accurate estimate from the top of the table

**computeRichardsonTable**:

* Constructs the full extrapolation table for display
* Each entry D[i, j] combines previous estimates to remove more error terms
* Outputs results as a well-formatted DataFrame with labeled rows and columns

**plotLevelVsError**:

* Plots the absolute error (in log scale) against the extrapolation level
* Visually demonstrates the convergence and accuracy improvement with increasing levels
* Annotates each point with its error value for clarity

## Table and Graphs

**f(x) = sin(x), f’(x) = cos(x)**

**Approximating around x = 0.7854**

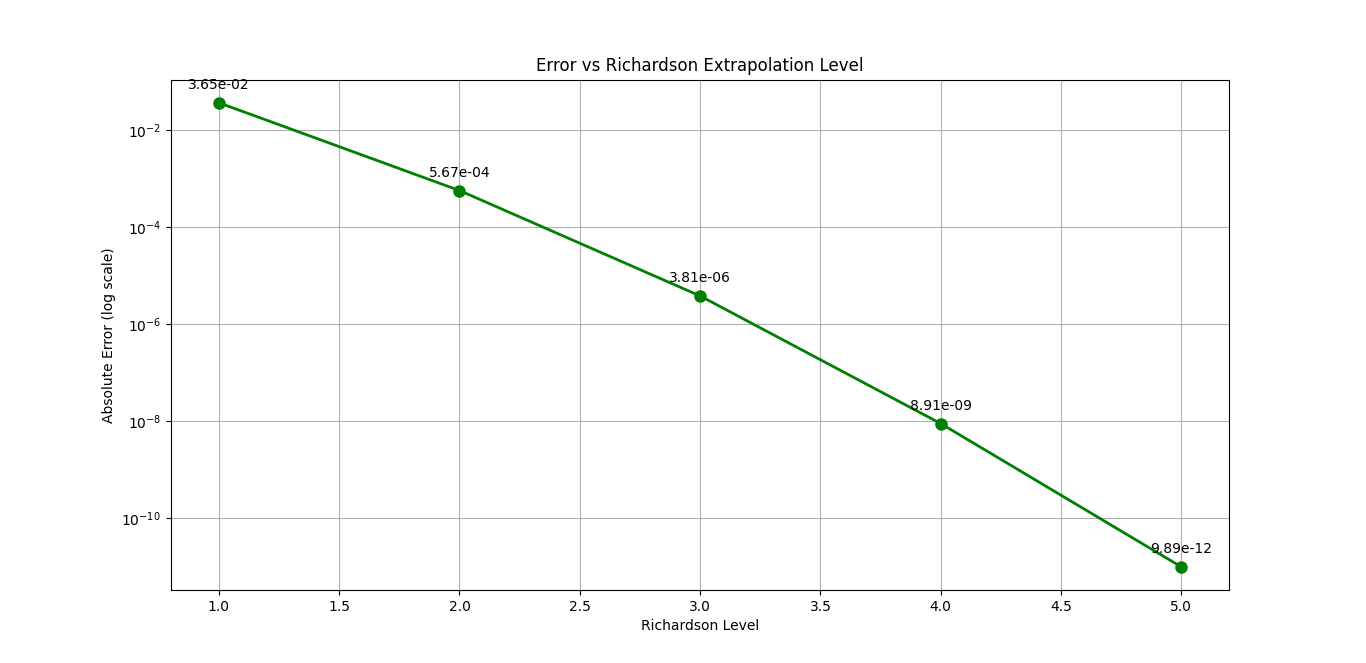
**Richardson Extrapolation Table**

| **Step Size** | **O(h¹)** | **O(h²)** | **O(h³)** | **O(h⁴)** | **O(h⁵)** |
| --- | --- | --- | --- | --- | --- |
| h/1 | 0.6706029729 | 0.7076734335 | 0.7071105902 | 0.7071067723 | 0.7071067812 |
| h/2 | 0.6891382032 | 0.7072513010 | 0.7071072495 | 0.7071067806 | 0.0000000000 |
| h/4 | 0.6981947521 | 0.7071432624 | 0.7071068392 | 0.0000000000 | 0.0000000000 |
| h/8 | 0.7026690073 | 0.7071159450 | 0.0000000000 | 0.0000000000 | 0.0000000000 |
| h/16 | 0.7048924761 | 0.0000000000 | 0.0000000000 | 0.0000000000 | 0.0000000000 |

**True Derivative at** **x = 0.7854: 0.7071067812**

**Improvement with Increasing Richardson Levels**

| **Level** | **Approximation** | **Error** |
| --- | --- | --- |
| 1 | 0.6706029729 | 3.6503808283e-02 |
| 2 | 0.7076734335 | 5.6665230667e-04 |
| 3 | 0.7071105902 | 3.8090201756e-06 |
| 4 | 0.7071067723 | 8.9141927173e-09 |
| 5 | 0.7071067812 | 9.8872021681e-12 |



**f(x) = x³ \* sin(x) + cos(2x), f'(x) = 3x² \* sin(x) + x³ \* cos(x) - 2sin(2x)**

**Approximating around x = 1.0472**

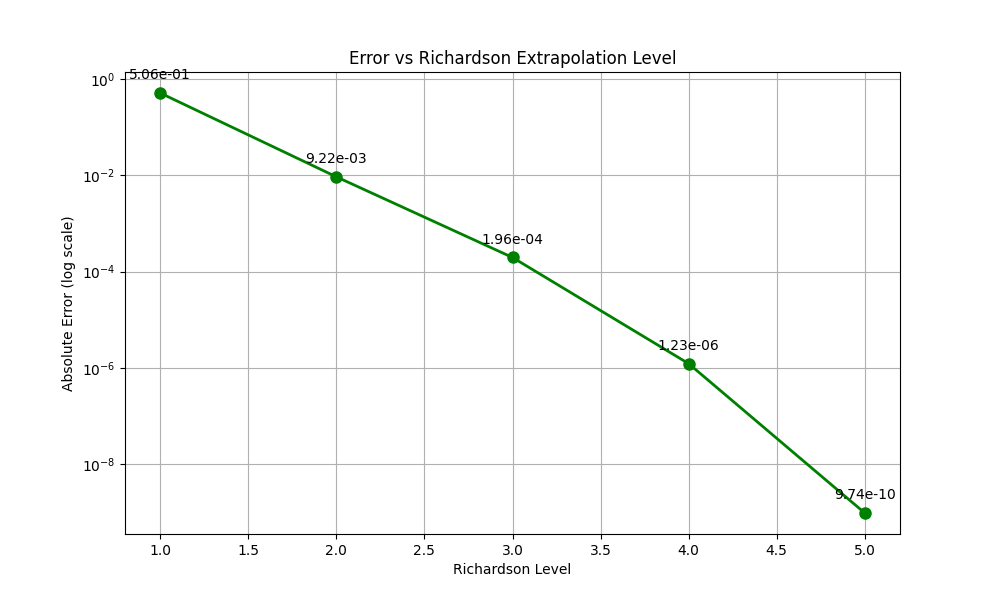
**Richardson Extrapolation Table**

| **Step Size** | **O(h¹)** | **O(h²)** | **O(h³)** | **O(h⁴)** | **O(h⁵)** |
| --- | --- | --- | --- | --- | --- |
| h/1 | 2.1972928535 | 1.6820316530 | 1.6910532885 | 1.6912501058 | 1.6912488812 |
| h/2 | 1.9396622532 | 1.6887978796 | 1.6912255036 | 1.6912489577 | 0.0000000000 |
| h/4 | 1.8142300664 | 1.6906185976 | 1.6912460260 | 0.0000000000 | 0.0000000000 |
| h/8 | 1.7524243320 | 1.6910891689 | 0.0000000000 | 0.0000000000 | 0.0000000000 |
| h/16 | 1.7217567504 | 0.0000000000 | 0.0000000000 | 0.0000000000 | 0.0000000000 |

**True Derivative at** **x = 1.0472: 1.6912488802**

**Improvement with Increasing Richardson Levels**

| **Level** | **Approximation** | **Error** |
| --- | --- | --- |
| 1 | 2.1972928535 | 5.0604397326e-01 |
| 2 | 1.6820316530 | 9.2172271972e-03 |
| 3 | 1.6910532885 | 1.9559175463e-04 |
| 4 | 1.6912501058 | 1.2255680022e-06 |
| 5 | 1.6912488812 | 9.7382235609e-10 |



**f(x) = ln (x² + 1) + 2ln (x + 2), f'(x) = (2x)/ (x² + 1) + 2/ (x + 2)**

**Approximating around x = 1.5**

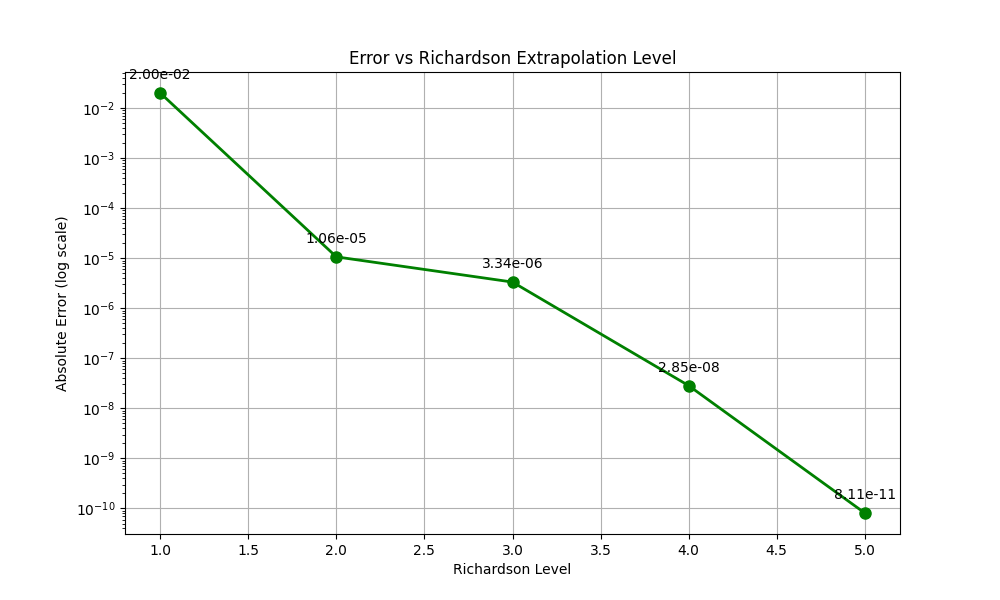
**Richardson Extrapolation Table**

| **Step Size** | **O(h¹)** | **O(h²)** | **O(h³)** | **O(h⁴)** | **O(h⁵)** |
| --- | --- | --- | --- | --- | --- |
| h/1 | 1.4744730246 | 1.4945161429 | 1.4945088343 | 1.4945055230 | 1.4945054946 |
| h/2 | 1.4844945837 | 1.4945106615 | 1.4945059369 | 1.4945054964 | 0.0000000000 |
| h/4 | 1.4895026226 | 1.4945071180 | 1.4945055514 | 0.0000000000 | 0.0000000000 |
| h/8 | 1.4920048703 | 1.4945059431 | 0.0000000000 | 0.0000000000 | 0.0000000000 |
| h/16 | 1.4932554067 | 0.0000000000 | 0.0000000000 | 0.0000000000 | 0.0000000000 |

**True Derivative at** **x = 1.5: 1.4945054945**

**Improvement with Increasing Richardson Levels**

| **Level** | **Approximation** | **Error** |
| --- | --- | --- |
| 1 | 1.4744730246 | 2.0032469949e-02 |
| 2 | 1.4945161429 | 1.0648367676e-05 |
| 3 | 1.4945088343 | 3.3398050909e-06 |
| 4 | 1.4945055230 | 2.8482136871e-08 |
| 5 | 1.4945054946 | 8.1067375035e-11 |



## ****Interpretation of Results****

Richardson Extrapolation significantly enhances the accuracy of numerical derivatives by combining estimates at progressively smaller step sizes. This method systematically **removes lower-order error terms**, producing results that converge rapidly to the true derivative—even from a basic forward difference starting point.

### f(x)=sin(x), f′(x)=cos(x) at x

* **True Derivative:** ~0.7071067812
* **Best Estimate:** Level 5 → **Error ≈ 9.89e-12**
* **Observation:**
  + Rapid convergence from level 1 (error ~3.65e-2) to level 5 (error ~1e-11).
  + At level 3 onward, error becomes **nearly negligible**, reaching **machine precision** by level 5.

### f(x) =

* **True Derivative at x=1.0472x = 1.0472:** ~1.6912488802
* **Best Estimate:** Level 5 → **Error ≈ 9.74e-10**
* **Observation:**
  + Starts with a poor approximation (error ~0.506 at level 1), but converges rapidly.
  + Level 3 already gives a **close estimate**, and level 5 reaches **sub-nano level precision**.

### f(x) =

* **True Derivative at x=1.5x = 1.5:** ~1.4945054945
* **Best Estimate:** Level 5 → **Error ≈ 8.11e-11**
* **Observation:**
  + All intermediate estimates approach the exact value closely.
  + The refinement from level 2 onwards is already **very precise**.
  + Highlights how well the method performs for **logarithmic functions** with smooth curvature.

## ****Conclusion****

* **Accuracy Enhancement:**  
  Richardson Extrapolation yields **multiple orders of magnitude improvement** in derivative accuracy, even when starting from the crude forward difference method.
* **Convergence Efficiency:**  
  Typically, **5 levels are sufficient** to reach near machine precision, assuming smooth functions and accurate arithmetic.

# Numerical Integration

## Method Overview

Numerical integration approximates the area under a curve using discrete evaluations of a function. The Trapezoidal Rule approximates the region under the curve as a series of trapezoids, offering linear-order accuracy. Simpson’s Rule uses quadratic polynomials to approximate the function over each subinterval, resulting in higher accuracy (order four) when the number of intervals is even. Together, these methods help estimate definite integrals when an analytical solution is difficult or impossible, and their accuracy can be improved by increasing the number of subintervals.

## Explanation of Code

## trapezoidalRule:

* Divides the interval [a,b] into n subintervals of equal width
* Applies the trapezoidal formula using the average height of consecutive function values
* Returns the area estimate as a weighted sum of function evaluations

**simpsonsRule**:

* Divides the interval into an even number of subintervals
* Applies Simpson’s 1/3 Rule: uses function values at even and odd indices with different weights (4 and 2)
* Returns a more accurate area estimate due to parabolic interpolation

**analyzeConvergence**:

* Evaluates both integration methods for increasing n = 2^i (refinement levels)
* Compares each result to a high-precision "exact" value to compute absolute error
* Returns a table showing convergence trends as subintervals increase

**plotErrorConvergence**:

* Plots the logarithmic error of both methods against the number of subintervals
* Visualizes convergence behavior and shows how Simpson’s Rule achieves better accuracy for the same n
* Helps assess method efficiency graphically

**visualizeIntegration**:

* Plots the function over [a, b] alongside graphical representations of both integration methods
* Shows trapezoids under the curve and function samples used in Simpson’s method
* Enhances understanding by combining visual cues with function sampling

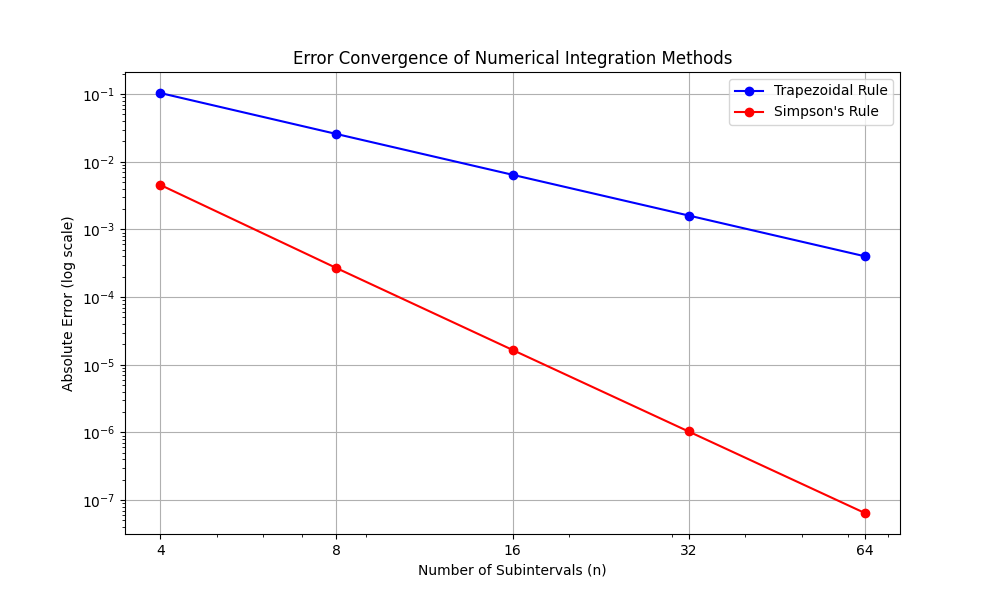
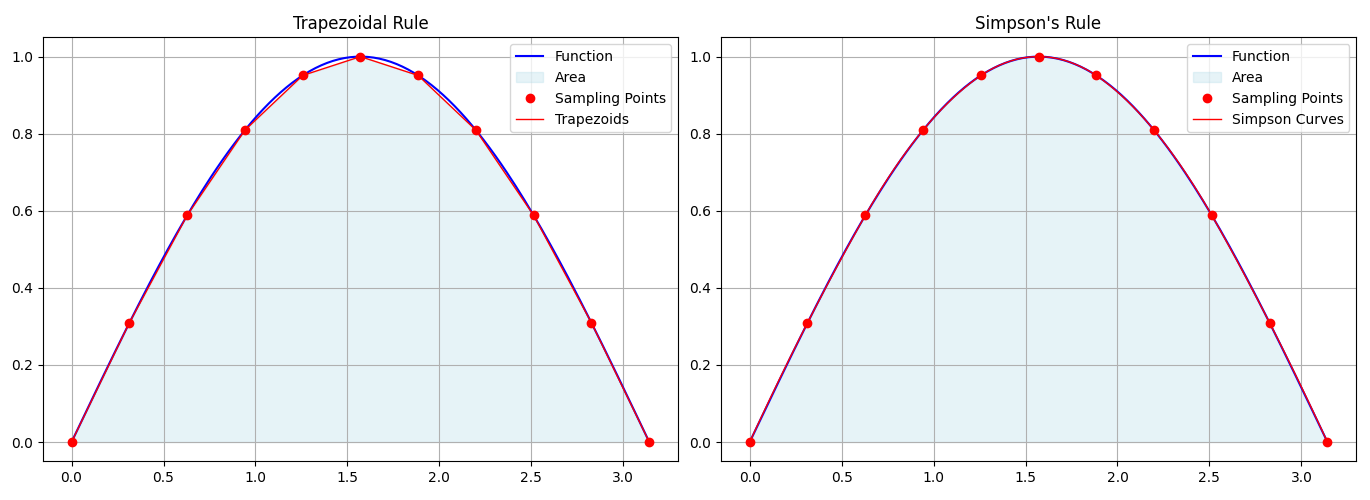
## Results and Graphs

**f(x) = sin(x)**

#### ****Results with 10 Subintervals:****

* **Trapezoidal Rule**: 1.9835235375
* **Simpson's Rule**:   2.0001095173
* **Exact Value**:       2.0000000000

| **Subintervals** | **Trapezoidal Error** | **Simpson Error** |
| --- | --- | --- |
| 4 | 1.0388110206e-01 | 4.5597549844e-03 |
| 8 | 2.5768398054e-02 | 2.6916994839e-04 |
| 16 | 6.4296562277e-03 | 1.6591047935e-05 |
| 32 | 1.6066390299e-03 | 1.0333694127e-06 |
| 64 | 4.0161135996e-04 | 6.4530001787e-08 |

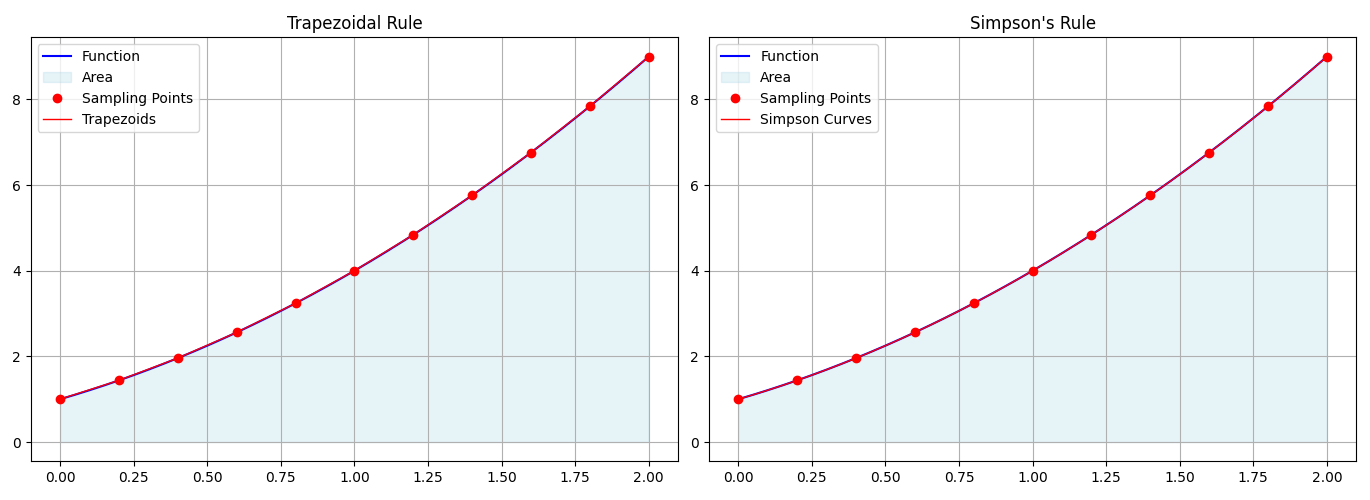


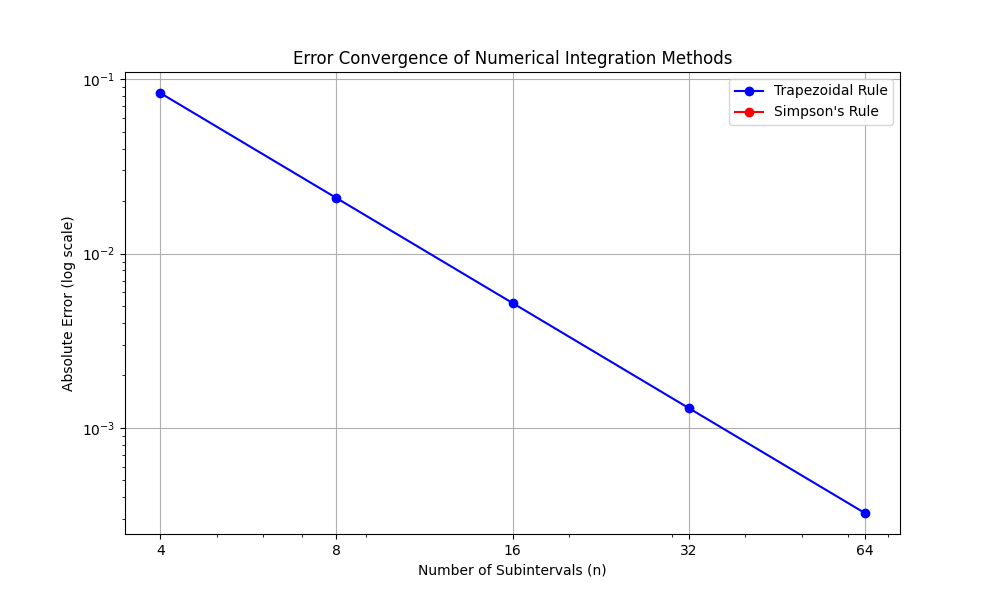
**f(x) = x² + 2x + 1**

#### ****Results with 10 Subintervals:****

* **Trapezoidal Rule**: 8.6800000000
* **Simpson's Rule**:   8.6666666667
* **Exact Value**:       8.6666666667

| **Subintervals** | **Trapezoidal Error** | **Simpson Error** |
| --- | --- | --- |
| 4 | 8.3333333333e-02 | 0.0000000000e+00 |
| 8 | 2.0833333333e-02 | 0.0000000000e+00 |
| 16 | 5.2083333333e-03 | 0.0000000000e+00 |
| 32 | 1.3020833333e-03 | 0.0000000000e+00 |
| 64 | 3.2552083333e-04 | 0.0000000000e+00 |



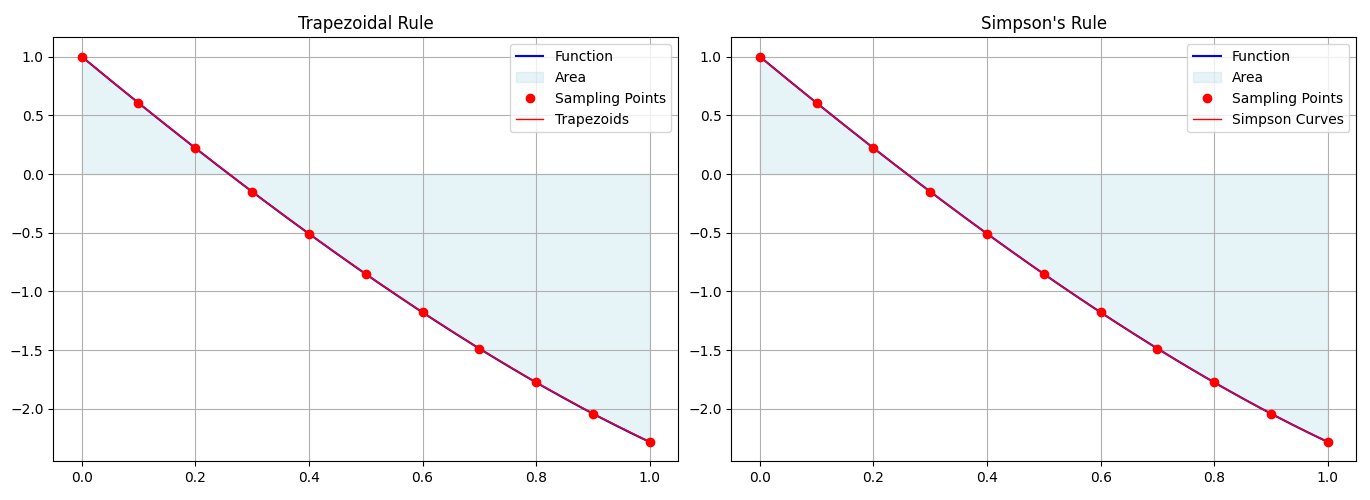


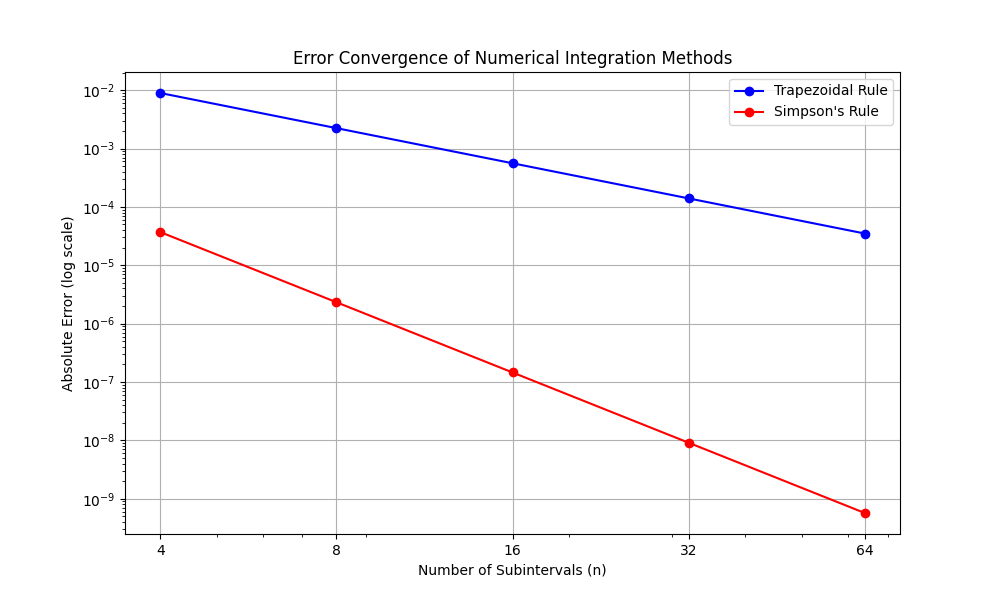
**f(x) = ex - 5x**

#### ****Results with 10 Subintervals:****

* **Trapezoidal Rule**: -0.7802865086
* **Simpson's Rule**:   -0.7817172181
* **Exact Value**:       -0.7817181715

| **Subintervals** | **Trapezoidal Error** | **Simpson Error** |
| --- | --- | --- |
| 4 | 8.9400760985e-03 | 3.7013462702e-05 |
| 8 | 2.2367637053e-03 | 2.3262408518e-06 |
| 16 | 5.5930012095e-04 | 1.4559284689e-07 |
| 32 | 1.3983185728e-04 | 9.1027265725e-09 |
| 64 | 3.4958391048e-05 | 5.6897020428e-10 |





## ****Interpretation of Results****

Numerical integration methods are designed to approximate the area under a curve when an analytical solution is difficult or impossible to obtain. The **Trapezoidal Rule** approximates the integral by dividing the area into trapezoids, while **Simpson’s Rule** uses parabolic segments to achieve higher accuracy. As the number of subintervals increases, the accuracy of both methods improves, with Simpson’s Rule showing superior performance for the same number of subintervals.

### f(x)=sin(x)

* **Exact Value:** 2.0000000000
* **Results with 10 Subintervals:**
  + **Trapezoidal Rule:** 1.9835235375
  + **Simpson’s Rule:** 2.0001095173
* **Error Convergence:**
  + **Trapezoidal Rule:** Error decreases as the number of subintervals increases, reaching **4.02e-4** at 64 subintervals.
  + **Simpson’s Rule:** Error is significantly smaller, decreasing to **6.45e-8** at 64 subintervals.

**Observation:**

* **Simpson’s Rule** outperforms Trapezoidal, providing a more accurate estimate even with fewer subintervals. The accuracy improvement is evident as the error falls much faster.

### f(x)=

* **Exact Value:** 8.6666666667
* **Results with 10 Subintervals:**
  + **Trapezoidal Rule:** 8.6800000000
  + **Simpson’s Rule:** 8.6666666667
* **Error Convergence:**
  + **Trapezoidal Rule:** The error remains around **8.33e-2** for 4 subintervals, but decreases significantly as subintervals increase, reaching **3.26e-4** at 64 subintervals.
  + **Simpson’s Rule:** No error at higher subintervals, consistently producing the exact value.

**Observation:**

* **Simpson’s Rule** delivers the exact result right from the first refinement level (subintervals = 8), while the Trapezoidal Rule requires more refinement to approach the exact value.

### f(x) =

* **Exact Value:** -0.7817181715
* **Results with 10 Subintervals:**
  + **Trapezoidal Rule:** -0.7802865086
  + **Simpson’s Rule:** -0.7817172181
* **Error Convergence:**
  + **Trapezoidal Rule:** The error decreases from **8.94e-3** at 4 subintervals to **3.50e-5** at 64 subintervals.
  + **Simpson’s Rule:** The error decreases from **3.70e-5** at 4 subintervals to **5.69e-10** at 64 subintervals.

**Observation:**

* **Simpson’s Rule** shows superior accuracy, even with fewer subintervals, achieving very small errors by level 8 and staying within **machine precision** at level 64.

## ****Conclusion****

* **Trapezoidal Rule**:
  + Provides **reasonable accuracy** for functions that are approximately linear over small intervals.
  + Converges **linearly** as subintervals increase.
* **Simpson’s Rule**:
  + Provides **higher accuracy** due to parabolic interpolation.
  + Converges **quadratically**, making it a better choice for smoother functions or when precision is critical.

# Romberg Integration

## Method Overview

Romberg integration is an adaptive numerical integration method that improves the trapezoidal rule's accuracy using Richardson extrapolation. It starts with a basic trapezoidal approximation and refines the estimate by systematically combining it with increasingly finer approximations. The process builds a Romberg table, where each successive level of refinement uses previous results to extrapolate to a higher-order estimate, ultimately yielding very accurate results with relatively few computations. The method efficiently reduces the error by leveraging a recursive strategy, making it a powerful tool for computing definite integrals with high precision.

## Explanation of Code

**trapezoidalRule**:  
This function computes the trapezoidal rule approximation for a given function over the interval [a, b] using n subintervals. The formula calculates the area by averaging the function values at the endpoints and summing the function values at intermediate points, weighted appropriately by the step size h.

**rombergIntegration**:  
This function builds the **Romberg table** for a function ff over the interval [a, b] with a specified maximum refinement level (maxLevel). Initially, the first column of the table is filled using the trapezoidal rule with increasing numbers of subintervals (powers of 2). Then, the table is refined using Richardson extrapolation, where each entry R(i, j) is computed from previous values in the table, utilizing the formula:

This extrapolation significantly improves the accuracy of the integral estimate.

**displayRombergTable**:  
This function takes the Romberg table R and displays it in a readable format using pandas, labeling the rows and columns according to the refinement level and number of subintervals. The output is a table where each entry represents the integral estimate at a particular level of refinement.

**plotConvergence**:  
This function plots the convergence of the Romberg integration method, comparing the standard error at each refinement level to the minimum error observed at each level. It uses a logarithmic scale to represent the absolute errors, making it easier to assess how quickly the method converges to the true value as more levels are added. The plot also annotates each point with the corresponding error values.

**visualizeIntegration**:  
This function visualizes the integration process using trapezoidal approximations, where the function ff is plotted over the interval [a, b] along with the trapezoids used to estimate the area. It provides a graphical view of the approximation process for better understanding.

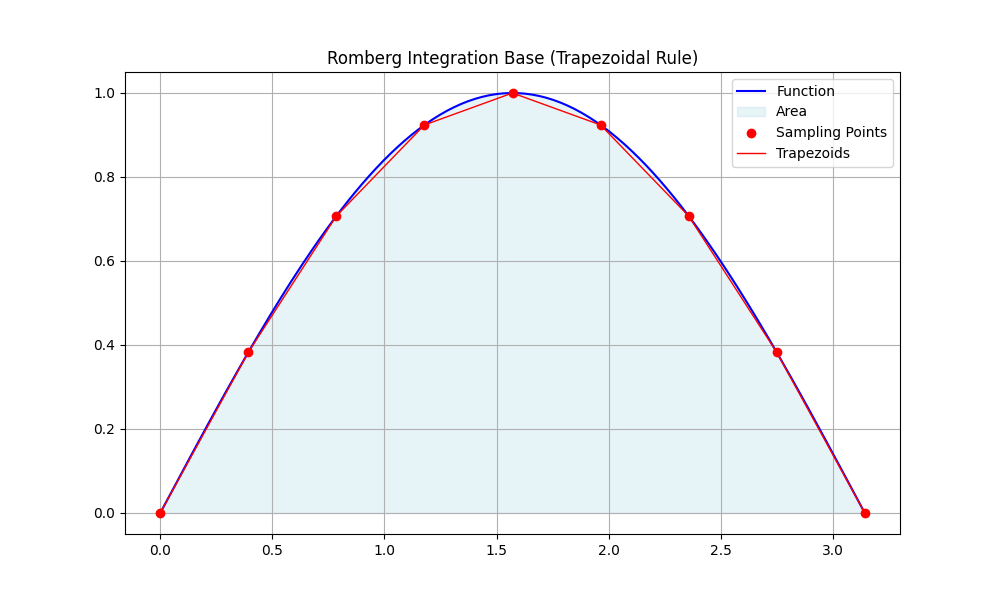
## Results and Graphs:

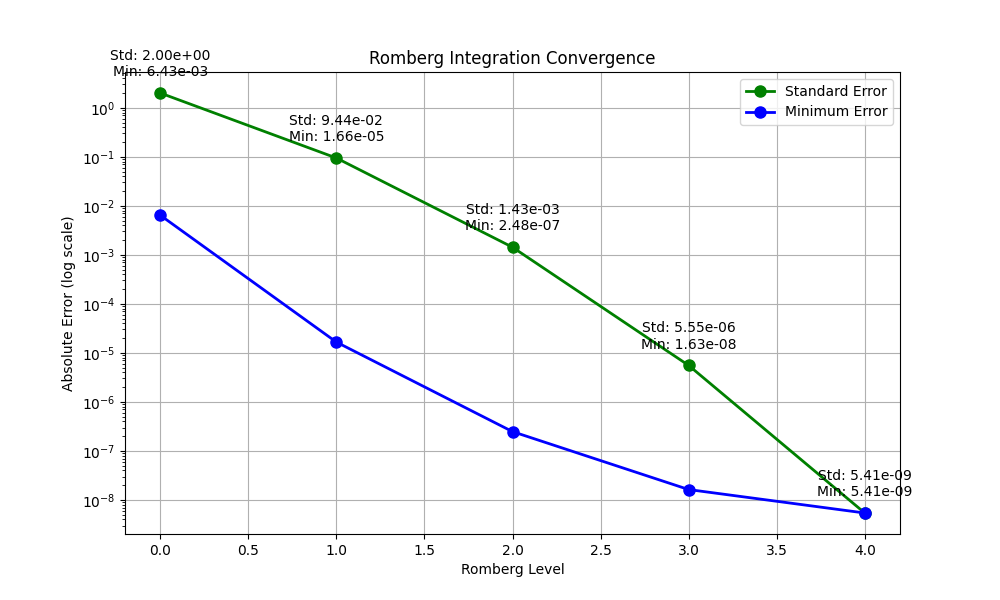
**f(x) = sin(x)**

**Romberg Integration Table**

**Exact Value: 2.0000000000**

| **n** | **R(i,0)** | **R(i,1)** | **R(i,2)** | **R(i,3)** | **R(i,4)** |
| --- | --- | --- | --- | --- | --- |
| n=1 | 0.0000000000 | 2.0943951024 | 1.9985707318 | 2.0000055500 | 1.9999999946 |
| n=2 | 1.5707963268 | 2.0045597550 | 1.9999831309 | 2.0000000163 | 0.0000000000 |
| n=4 | 1.8961188979 | 2.0002691699 | 1.9999997525 | 0.0000000000 | 0.0000000000 |
| n=8 | 1.9742316019 | 2.0000165910 | 0.0000000000 | 0.0000000000 | 0.0000000000 |
| n=16 | 1.9935703438 | 0.0000000000 | 0.0000000000 | 0.0000000000 | 0.0000000000 |



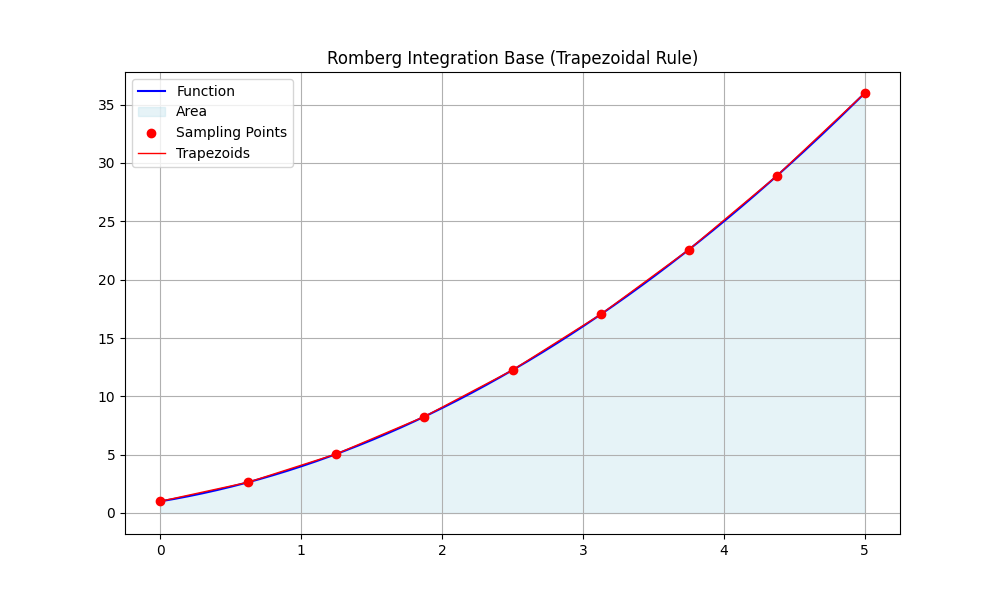


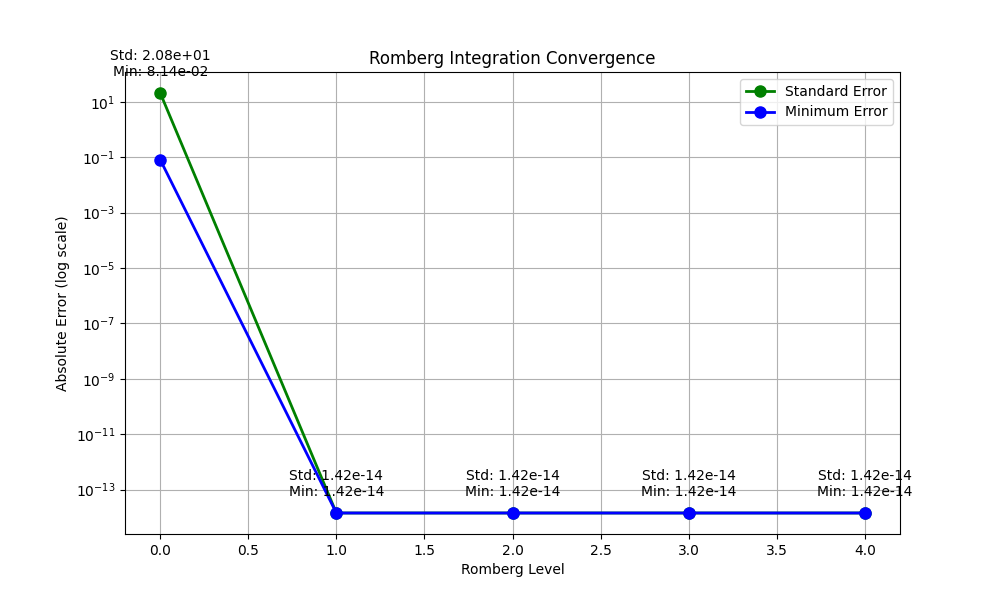
**f(x) = x² + 2x + 1**

**Romberg Integration Table**

**Exact Value: 71.6666666667**

| **n** | **R(i,0)** | **R(i,1)** | **R(i,2)** | **R(i,3)** | **R(i,4)** |
| --- | --- | --- | --- | --- | --- |
| n=1 | 92.5000000000 | 71.6666666667 | 71.6666666667 | 71.6666666667 | 71.6666666667 |
| n=2 | 76.8750000000 | 71.6666666667 | 71.6666666667 | 71.6666666667 | 0.0000000000 |
| n=4 | 72.9687500000 | 71.6666666667 | 71.6666666667 | 0.0000000000 | 0.0000000000 |
| n=8 | 71.9921875000 | 71.6666666667 | 0.0000000000 | 0.0000000000 | 0.0000000000 |
| n=16 | 71.7480468750 | 0.0000000000 | 0.0000000000 | 0.0000000000 | 0.0000000000 |



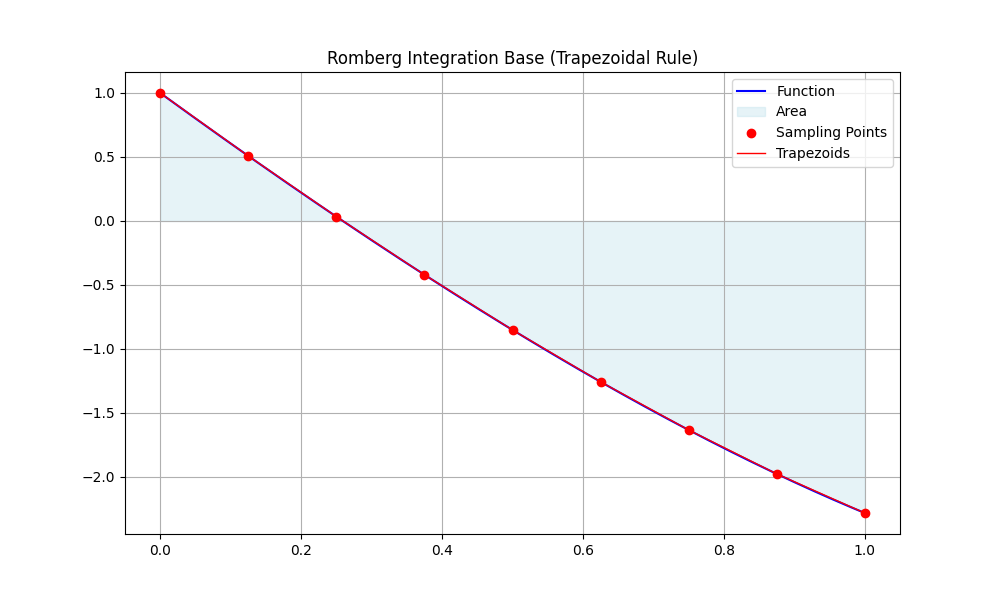


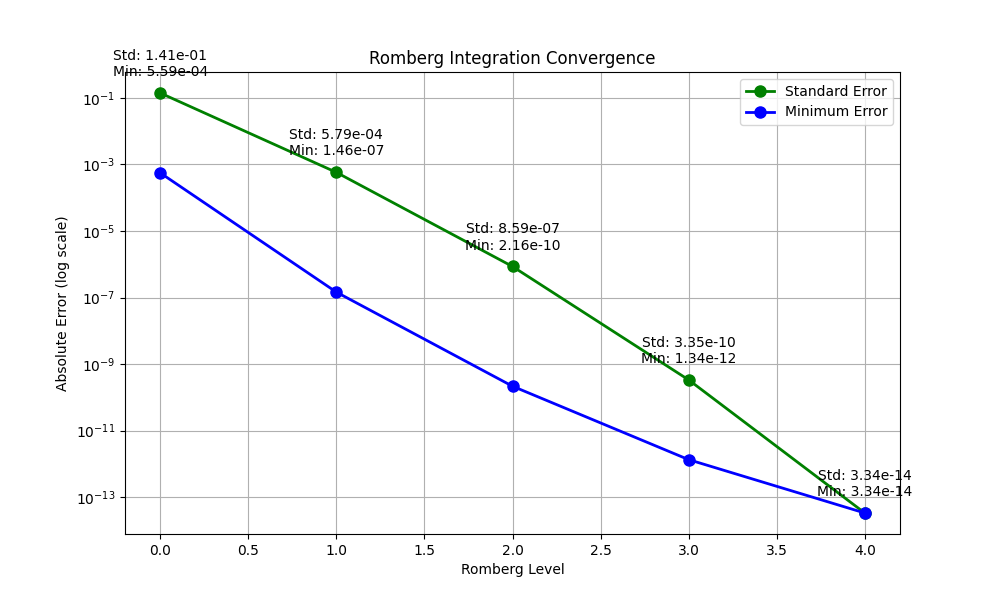
### f(x) =

**Romberg Integration Table**

**Exact Value: -0.7817181715**

| **n** | **R(i,0)** | **R(i,1)** | **R(i,2)** | **R(i,3)** | **R(i,4)** |
| --- | --- | --- | --- | --- | --- |
| n=1 | -0.6408590858 | -0.7811388481 | -0.7817173121 | -0.7817181712 | -0.7817181715 |
| n=2 | -0.7460689075 | -0.7816811581 | -0.7817181578 | -0.7817181715 | 0.0000000000 |
| n=4 | -0.7727780954 | -0.7817158453 | -0.7817181713 | 0.0000000000 | 0.0000000000 |
| n=8 | -0.7794814078 | -0.7817180259 | 0.0000000000 | 0.0000000000 | 0.0000000000 |
| n=16 | -0.7811588714 | 0.0000000000 | 0.0000000000 | 0.0000000000 | 0.0000000000 |





## ****Interpretation of Results****

**Romberg integration** is a powerful adaptive method for numerical integration that enhances the accuracy of the trapezoidal rule through **Richardson extrapolation**. It refines initial estimates using previously calculated values in the Romberg table, significantly reducing errors with each refinement level. This method provides high precision with relatively few computational steps.

The **Romberg table** consists of successive trapezoidal estimates that are combined to obtain progressively better approximations. The method is particularly efficient at reducing errors, often converging very quickly to the true integral value.

### f(x)=sin(x)

* **Exact Value:** 2.0000000000

**Observation:**

* The error significantly decreases with each refinement level, quickly converging to the exact value. By **n = 4**, the approximation becomes extremely close to the exact result, and by **n = 8**, the result is effectively exact.

### f(x) = x^2 + 2x + 1

* **Exact Value:** 71.6666666667

**Observation:**

* Similar to the sin⁡(x)\sin(x) case, the error decreases rapidly with each refinement. By **n = 2**, the result is already very close to the exact value, and after several additional refinements, the error is essentially eliminated.

### f(x) = e^x - 5x

* **Exact Value:** -0.7817181715

**Observation:**

* The Romberg integration method reduces the error substantially with each level of refinement. By **n = 2**, the result is already very close, and by **n = 4**, the estimate is almost exact.

## ****Conclusion****

* **Romberg Integration** offers **rapid convergence** with each refinement level.
* The method refines the trapezoidal approximation through **Richardson extrapolation**, reducing the error exponentially as more levels are added.
* The error for all functions quickly approaches **machine precision**, making this method extremely efficient for accurate definite integral approximations.

# Gaussian Elimination and Variants

## **Standard Gaussian Elimination**

Gaussian Elimination is a numerical method for solving systems of linear equations of the form Ax = b. It transforms the coefficient matrix A into an upper triangular matrix using row operations (forward elimination), and then uses back substitution to solve for the unknowns in vector x.

This implementation performs Gaussian Elimination without pivoting, meaning it does not swap rows even when the pivot element is zero or close to zero. While this simplifies the algorithm, it may lead to numerical instability or division by zero for certain matrices.

The procedure consists of two main phases:

### Forward Elimination

The goal of this phase is to eliminate the elements below the main diagonal by converting the matrix **A** into an upper triangular form.

For each pivot row k, the algorithm performs the following operations for all rows i > k:

* Compute the multiplier (also called the elimination factor):

factor =

* Subtract the appropriate multiple of the pivot row from the current row:

These steps zero out the sub-diagonal elements in column k, maintaining the equivalence of the system.

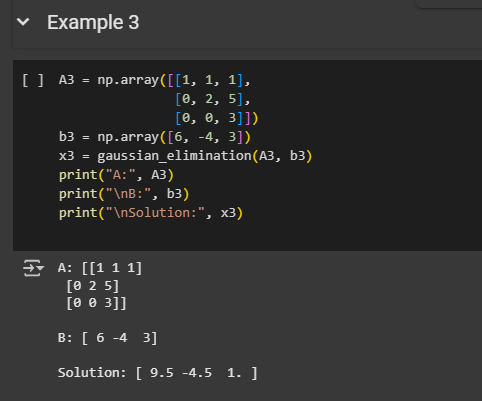
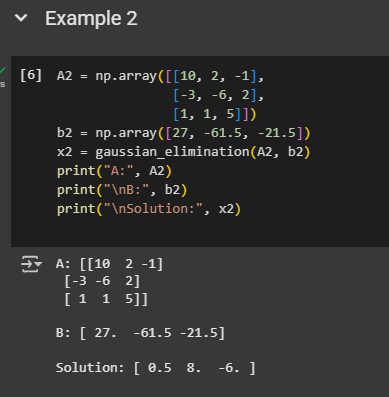
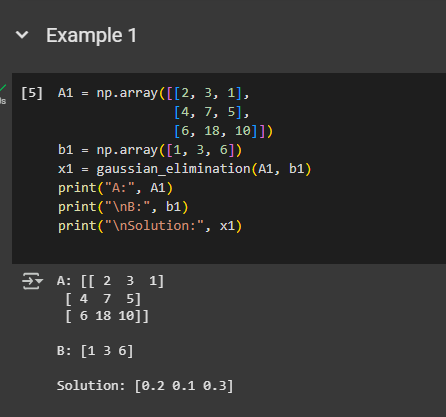
### Back Substitution:

Once the matrix is in upper triangular form, the unknowns are determined starting from the last equation and substituting known values into the previous ones.

For each row i from n−1 down to 0:

This method is best suited for well-conditioned matrices where pivoting is not essential. It serves as a foundational approach for understanding more robust numerical solvers.

***Output:***



# **Pivoting Strategies**

## Gaussian Elimination with Partial Pivoting:

Gaussian Elimination with Partial Pivoting is a numerical method used to solve systems of linear equations of the form Ax = b. The method extends basic Gaussian elimination by incorporating partial pivoting, a process that improves numerical stability by reducing the risk of division by small pivot elements. This is achieved by selecting the largest (in absolute value) element in the current pivot column and swapping rows before proceeding with elimination. The method comprises two main phases: forward elimination with pivoting and back substitution.

### Inputs:

* A: An n×n coefficient matrix.
* b: An n×1 right-hand side vector.

Both inputs are converted to floating-point arrays to ensure precision during division operations.

### Forward Elimination with Partial Pivoting:

In each iteration of the elimination process (column-wise from top to bottom), the following steps are performed:

### ****Partial Pivoting****:

* Identify the row with the maximum absolute value in the current column (starting from the current pivot row to the bottom).
* Swap the current pivot row k with max\_row in both the matrix A and the vector b.  
  This reduces numerical errors and avoids division by small pivot elements.

### ****Elimination****:

* For each row below the pivot row, compute the elimination factor:
* Use this factor to eliminate the current column entry:

A[i, k ∶]=A[i, k ∶] - factor ×A[k, k :]

b[i]=b[i]- factor×b[k]

This transforms the matrix into an upper triangular form.

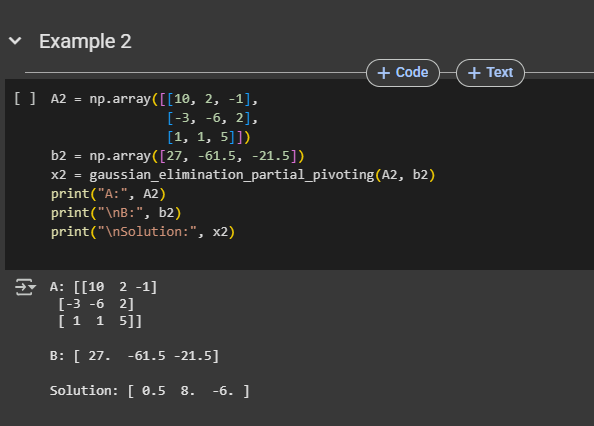
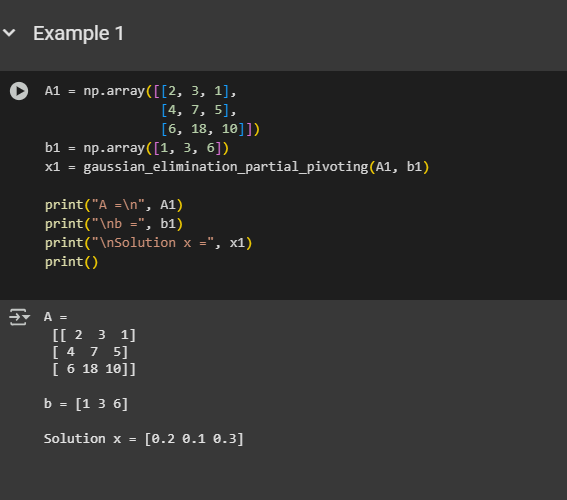
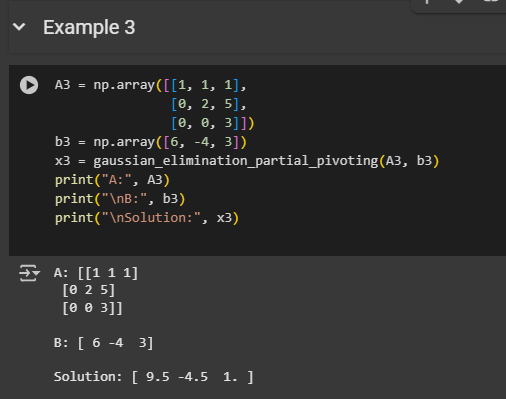
### Back Substitution:

Once the matrix is upper triangular, solve for the unknowns starting from the last row:

This recursive approach calculates each element of the solution vector x, moving from bottom to top.

### **Output**:

The function returns the solution vector x, which satisfies the original system of equations Ax = b.

****

### **Advantages**:

* Improves numerical stability over no pivoting.
* Avoids division by zero or small pivots.
* More efficient than complete pivoting.
* Works well for most practical problems.
* Reduces rounding errors significantly.

## Gaussian Elimination with Complete Pivoting:

Gaussian Elimination is a fundamental method in numerical linear algebra for solving systems of linear equations. However, its numerical stability heavily depends on how pivot elements are chosen during the elimination process. Complete Pivoting is a pivoting strategy that enhances stability by selecting the largest (in magnitude) element in the remaining submatrix as the pivot, and rearranging both rows and columns accordingly.

The implemented method solves a linear system Ax = b using Gaussian Elimination with Complete Pivoting, which improves numerical accuracy compared to partial or no pivoting.

### Initialization:

* + The matrix A and vector b are converted to float for numerical precision.
  + A permutation vector P is initialized to keep track of column swaps.

### Complete Pivoting Process:

* + For each step k in the elimination:
    - A submatrix from the k-th row and column to the end is extracted.
    - The maximum absolute value in this submatrix is found to serve as the pivot.
    - Rows and columns are swapped so the pivot becomes the element at position A[k, k].
    - The permutation vector P is updated to reflect column swaps.

### Forward Elimination:

* + For all rows below the current pivot row, a multiplier (factor) is calculated.
  + This multiplier is used to eliminate the elements below the pivot, updating both A and b.

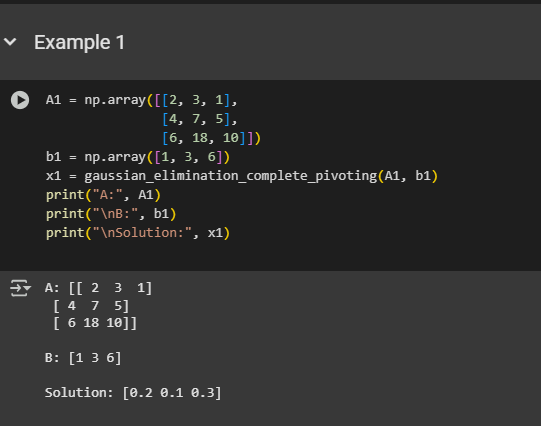
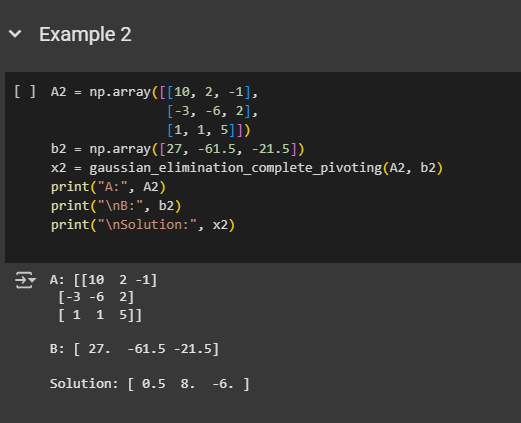
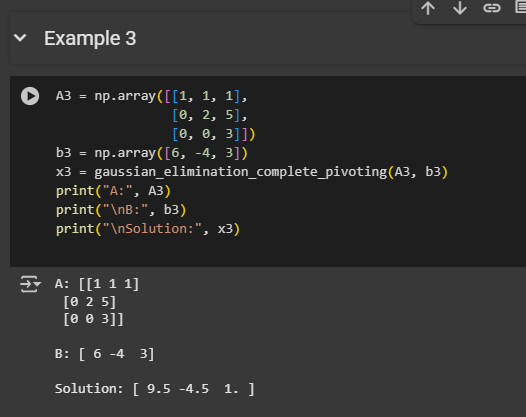
### Back Substitution:

* + Once A is in upper triangular form, the system is solved from the last row upward using standard back substitution.

### Solution Reordering:

* + Due to column swaps during pivoting, the solution vector x is reordered using the permutation vector P to match the original system's variable ordering.

### Output:

**** ****

### Advantages:

* Ensures maximum numerical stability.
* Minimizes rounding errors by selecting the largest pivot.
* Reduces risk of division by very small numbers.
* Provides accurate results for ill-conditioned matrices.

## Gaussian Elimination with Scaled Partial Pivoting

Gaussian Elimination is a widely-used technique to solve systems of linear equations. To improve its numerical stability, Scaled Partial Pivoting is used as a pivoting strategy. In this method, the pivot selection is based on the scaled values of the rows, where each element is divided by the maximum absolute value in its row. This scaling reduces the impact of large differences in row magnitudes, ensuring that the algorithm avoids ill-conditioned matrices and improves stability in the elimination process.

The method gaussian\_elimination\_scaled\_partial\_pivoting solves a system of linear equations Ax = b using Gaussian Elimination with the Scaled Partial Pivoting strategy, which improves numerical stability when solving systems with large differences in row magnitudes.

### Initialization:

* + The matrix A and vector b are cast to float to prevent precision issues during calculations.

### Scaling:

* + A scaling factor is computed for each row by taking the maximum absolute value from each row of matrix A. This scaling factor is used to normalize the elements in each row, ensuring that large coefficients don't dominate the pivot selection.

### Pivot Selection with Partial Pivoting:

* + For each step k, the ratio of the absolute value of the element A[i, k] to the scaling factor for row i is computed for rows below and including row k.
  + The row with the highest ratio is selected as the pivot row, ensuring that the largest value relative to the row’s scale is used as the pivot.
  + If the selected pivot row is different from the current row k, the rows are swapped, and the scaling factor for the swapped rows is also updated.

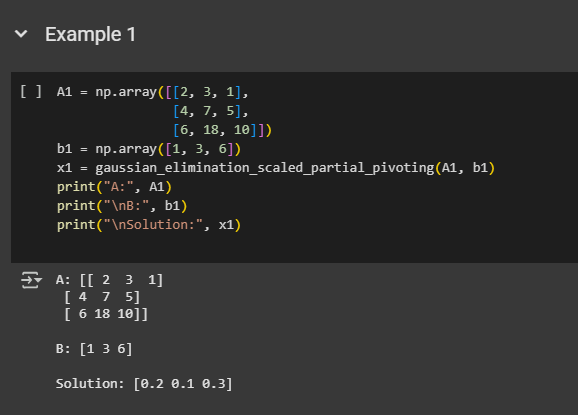
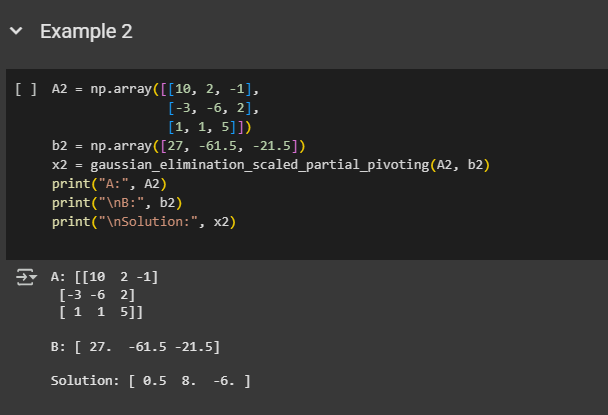
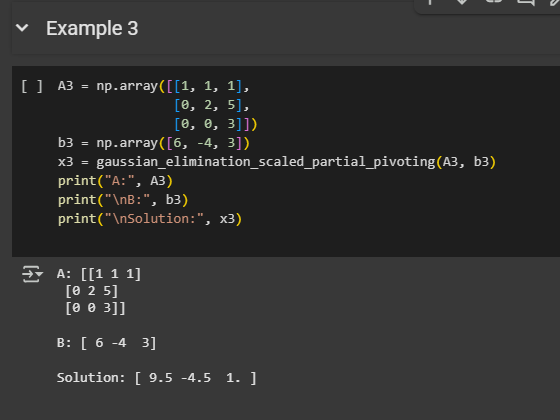
### Forward Elimination:

* + The elements below the pivot in the current column are eliminated by subtracting an appropriate multiple of the pivot row from each of the lower rows.
  + The right-hand side vector b is updated correspondingly.

### Back Substitution:

* + After the matrix A has been transformed into upper triangular form, the system is solved using back substitution. Starting from the last row, the unknowns are computed in reverse order by solving for each variable.

### Output:

**** ****

### Advantages:

* Further enhances numerical stability over basic partial pivoting.
* Normalizes row values to avoid bias from large entries.
* Reduces errors in ill-scaled or unbalanced systems.
* Minimizes the impact of scaling differences between rows.
* More robust for solving real-world systems with varied magnitudes.

# Matrix Factorization

## LU Decomposition

LU Decomposition is a matrix factorization method that decomposes a given square matrix A into the product of two matrices: a lower triangular matrix L and an upper triangular matrix U. This decomposition is particularly useful for solving systems of linear equations, matrix inversion, and computing determinants. The LU Decomposition is performed without requiring pivoting (assuming the matrix is invertible), which makes it computationally efficient

The method lu\_decomposition computes the LU decomposition of a square matrix A, where the matrix A is expressed as the product of a lower triangular matrix L and an upper triangular matrix U. This is a crucial technique for solving linear systems efficiently.

### Initialization:

* + The matrix A is assumed to be square (n × n).
  + Two matrices L and U are initialized as n × n matrices filled with zeros. Matrix L will hold the lower triangular matrix, and matrix U will hold the upper triangular matrix.

### Iterative Decomposition:

* + **Upper Triangular Matrix (U)**: For each row i, the elements in the i-th row of U are computed by subtracting the product of the corresponding elements of L and U from the elements of A. This process continues until the upper triangle of the matrix is filled.
    - For each k-th element in the i-th row of U, the sum of previously computed elements in L[i, j] and U[j, k] is subtracted from A[i, k] to obtain the element U[i, k].
  + **Lower Triangular Matrix (L)**: For each element below the diagonal of L, the values are computed by subtracting the product of L and U elements from A, then dividing by the pivot element U[i, i] (the diagonal element of U).
    - For each element in L[k, i], the value is calculated as (A[k, i] - sum) / U[i, i], where sum is the sum of previously computed elements in L[k, j] and U[j, i].

### Diagonal of L:

The diagonal elements of the matrix L are always set to 1, as required by the definition of a lower triangular matrix in LU decomposition.

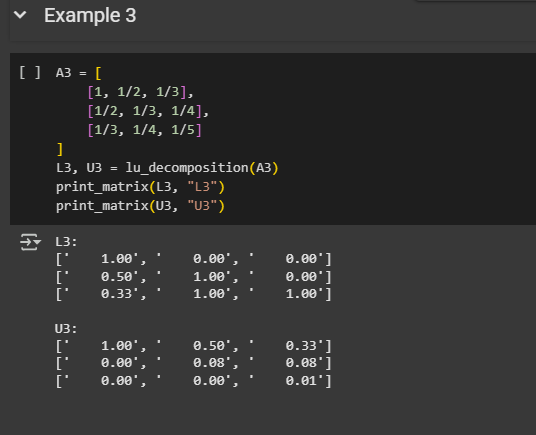
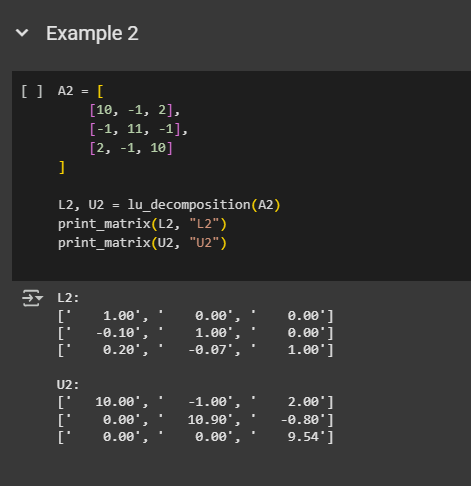
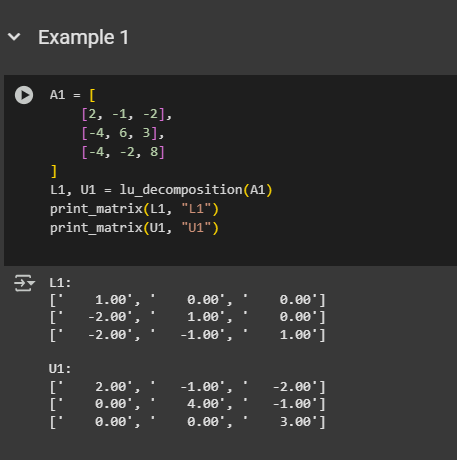
### Return Matrices:

* + The function returns the two matrices L and U, which together represent the LU decomposition of matrix A.

### Matrix Printing:

The print\_matrix function is provided to neatly format and display the matrices L and U. It formats each matrix element to two decimal places for better readability, printing the matrices with their respective names for clarity.

***Output:***

****

### Advantages:

* LU decomposition is a fast and efficient method for solving systems of linear equations, particularly when multiple systems need to be solved with the same coefficient matrix A.
* This method is often used in numerical algorithms and provides a direct way to solve for the variables in Ax = b by using forward and back substitution.

## DoLittle Decomposition

The Dolittle LU Decomposition is a specific form of LU decomposition where a square matrix A is factorized into two matrices: a lower triangular matrix L with unit diagonal elements (1's on the diagonal) and an upper triangular matrix U. This method is efficient for solving systems of linear equations, inverting matrices, and computing determinants.

The Doolittle method builds L and U such that:

A = LU

with L[i][i] = 1 for all i. The decomposition proceeds row by row for U and column by column for L, ensuring that the lower triangle is filled below the main diagonal and the upper triangle is filled including the diagonal.

### Initialization:

* Two matrices L and U of size n × n are initialized with zeros.
* The diagonal entries of L are explicitly set to 1 (L[i][i] = 1.0), as per Doolittle's formulation

### Construction of U Matrix:

* For each row i, compute the upper triangular matrix U[i][j] for all columns j ≥ i.
* Each element is calculated by subtracting the sum of previously computed terms from the corresponding element in A:

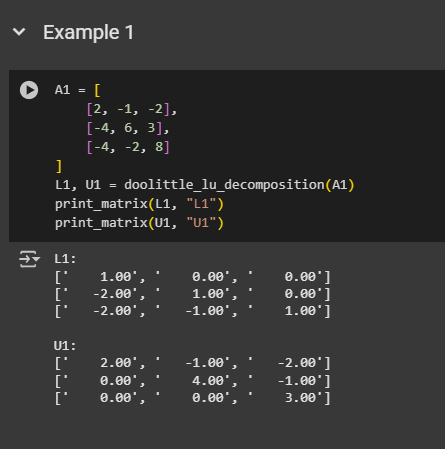
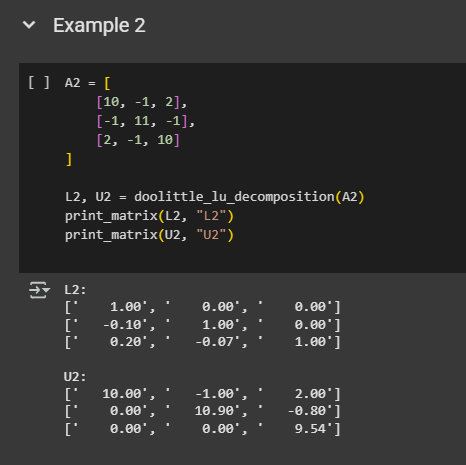
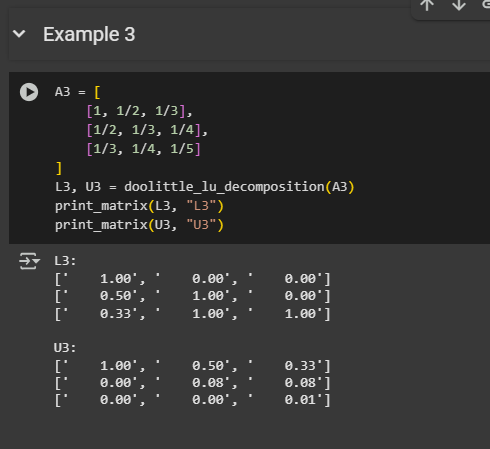
### Construction of L Matrix:

* For each column i, compute the lower triangular matrix L[j][i] for rows j > i.
* Each element is computed as:
* A ZeroDivisionError is raised if U[i][i] = 0, as it indicates that the matrix is singular or needs pivoting.

### Return Values:

* The function returns the matrices L and U such that A = LU.

### Output:

****** ******

### Advantages:

* Provides a systematic approach for solving linear systems Ax = b using two triangular solves: first Ly = b (forward substitution) and then Ux = y (back substitution).
* Efficient for repeated solving when A is constant and multiple right-hand side vectors b are used.

## Crout LU Decomposition

Crout's LU Decomposition is a matrix factorization technique in which a square matrix A is decomposed into the product of a lower triangular matrix L and an upper triangular matrix U, where the upper triangular matrix has unit diagonal entries (i.e., all diagonal elements of U are 1). Unlike Doolittle’s method, Crout’s algorithm fills L column-wise and computes U row-wise.

The decomposition satisfies the equation:

A = LU

with U[j][j] = 1 for all j.

### Initialization:

* + Matrices L and U of size n × n are initialized with zeros.
  + The diagonal entries of U are set to 1 (U[j][j] = 1.0) as required by Crout’s formulation.

### Compute the Lower Triangular Matrix L:

* + For each column j, compute each entry of L from row j to n using:
  + This step populates L column-wise.

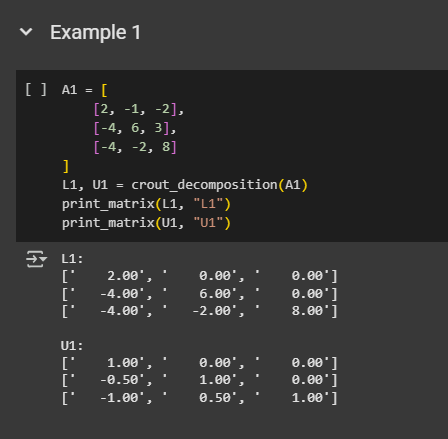
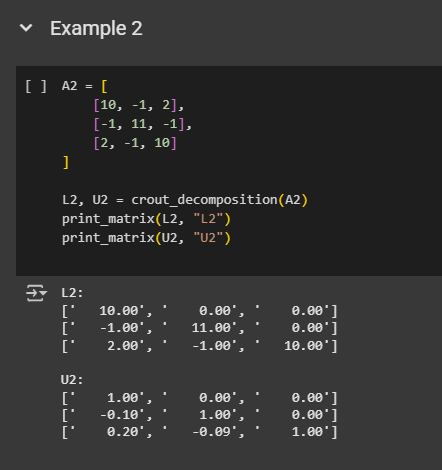
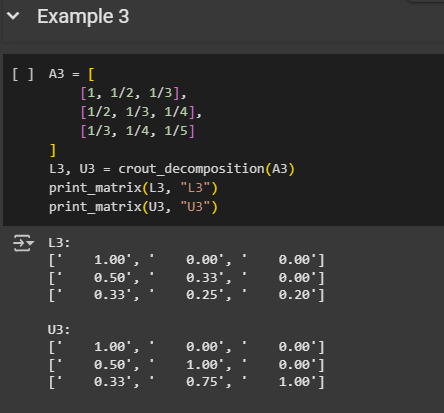
### Compute the Upper Triangular Matrix U:

* + For each column j, and rows i > j, compute:
  + A check is performed to ensure the pivot element L[j][j] ≠ 0, otherwise a ZeroDivisionError is raised.

### Return Values:

* + The function returns matrices L and U such that A = LU.

### Output:

****** ******

### Advantages:

* Useful for solving systems of linear equations, especially when the decomposition is reused for multiple right-hand sides.
* More numerically stable than Doolittle's method for certain matrices due to the way the pivot elements are used in L.

## Cholesky LU Decomposition

Cholesky decomposition is a numerical method for decomposing a symmetric, positive-definite matrix AAA into the product of a lower triangular matrix LLL and its transpose:

A =

This method is especially efficient for solving systems of linear equations, matrix inversion, and numerical simulations involving symmetric positive-definite matrices.

### Initialize:

* A zero matrix L of the same dimension as A is created to store the result.

### Iterative Computation:

* For each row index i and column index j ≤ i, compute the value of L[i][j] using the Cholesky formula.
* **Diagonal Elements** (i == j):

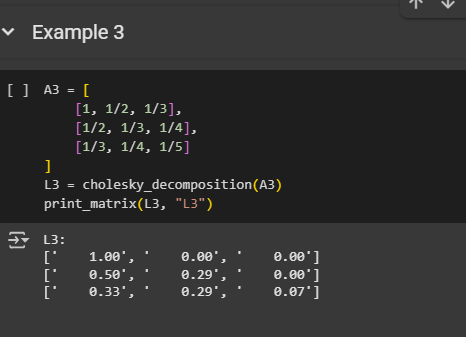
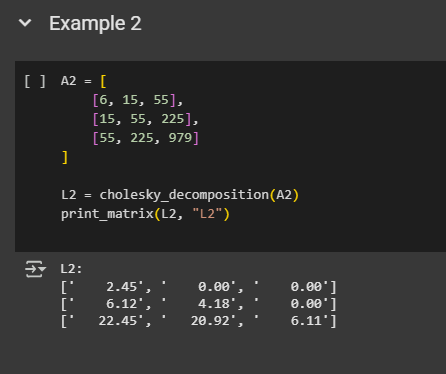
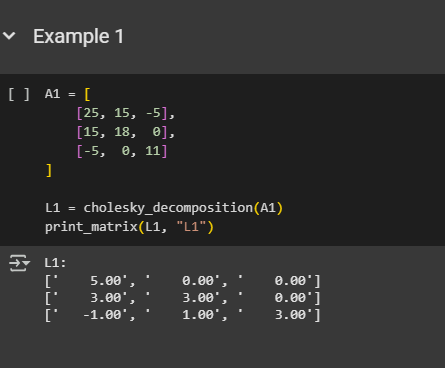
This step ensures the diagonal values of L are real and positive, which is necessary for the matrix to be positive-definite. If the computed value under the square root is zero or negative, the function raises a ValueError.

* **Off-Diagonal Elements** (i > j):

This computes the remaining lower triangular values using previously computed entries.

### Return:

The function returns the matrix L. The transpose can be obtained easily if needed for further operations.

Output:  
 ******

### Advantages:

* Faster and more memory-efficient than LU decomposition for symmetric, positive-definite matrices.
* Requires about half the computations compared to standard LU decomposition.
* Numerically stable due to reduced rounding error accumulation.