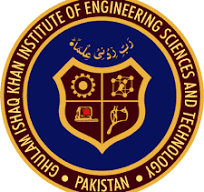
**Ghulam Ishaq Khan Institute of Engineering Sciences and Technology**

**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 3](#_Toc197180865)

[1.1. Method Overview 3](#_Toc197180866)

[1.2. Explanation of the Code 3](#_Toc197180867)

[1.3. Tables of Results 4](#_Toc197180868)

[1.4. Graphs 6](#_Toc197180869)

[1.5. Interpretation of Results 7](#_Toc197180870)

[Function 1: (x) = x3 - 4x + 1 on [0, 1] 8](#_Toc197180871)

[Function 2: f(x)= on [0, 10] 8](#_Toc197180872)

[Function 3: f(x)=cos(x)−x on[0, 1] 8](#_Toc197180873)

[Conclusion 8](#_Toc197180874)

[2. Fixed Point Iteration 8](#_Toc197180875)

[2.1. Method Overview 8](#_Toc197180876)

[2.2. Explanation of Code 8](#_Toc197180877)

[2.3. Table of Results 9](#_Toc197180878)

[2.4. Graphs 11](#_Toc197180879)

[2.5. Interpretation of Results 12](#_Toc197180880)

[Function 1: g(x)=cos(x), x0=0 13](#_Toc197180881)

[Function 2: g(x)= , x0=0 13](#_Toc197180882)

[Function 3: g(x)= , x0=2 13](#_Toc197180883)

[Conclusion 13](#_Toc197180884)

[3. Newton-Raphson Method 13](#_Toc197180885)

[3.1. Method Overview 13](#_Toc197180886)

[3.2. Explanation of the Code 13](#_Toc197180887)

[3.3. Table of Results 14](#_Toc197180888)

[3.4. Graphs 15](#_Toc197180889)

[3.5. Interpretation of Results 16](#_Toc197180890)

[Function 1: f(x)=x3−x−2, x0=2 16](#_Toc197180891)

[Function 2: f(x)=x2 – cos(x), x0=1 16](#_Toc197180892)

[Function 3: f(x)=ex−5x, x0=2 16](#_Toc197180893)

[Conclusion 16](#_Toc197180894)

# 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]**

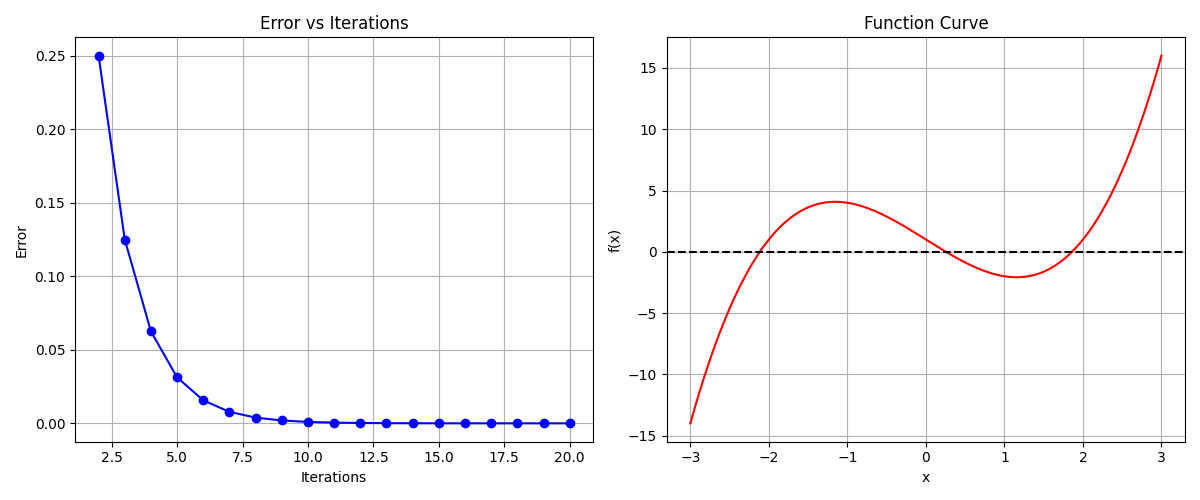
| **Iteration** | **a** | **f(a)** | **b** | **f(b)** | **x** | **f(x)** | **Error** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0.000000 | -2.000000 | 10.000000 | 1.162278 | 5.000000 | 0.236068 | inf |
| 2 | 0.000000 | -2.000000 | 5.000000 | 0.236068 | 2.500000 | -0.418861 | 2.500000 |
| 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**

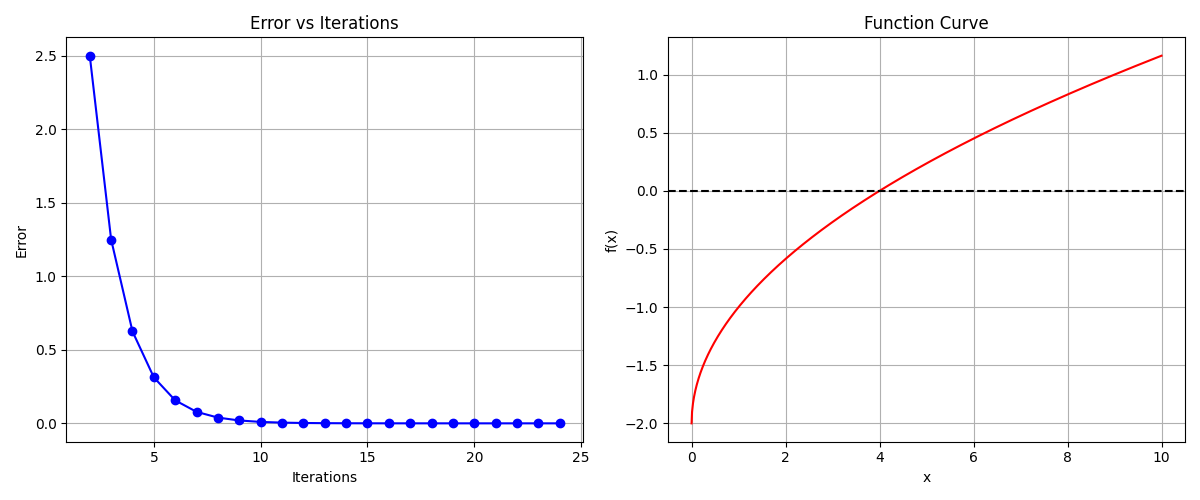
**Interval: [0,1]**

| **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 |

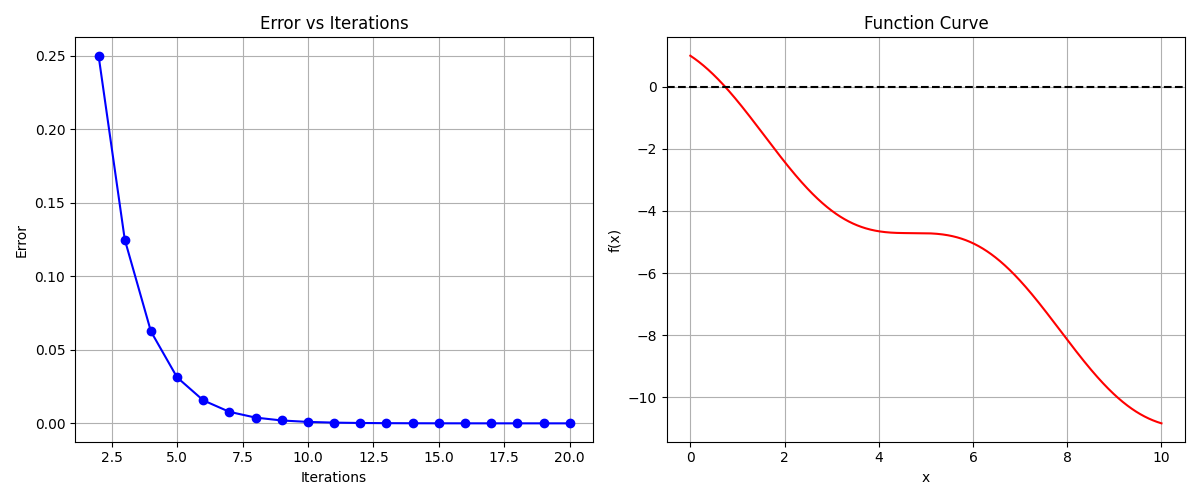
## Graphs

**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 |

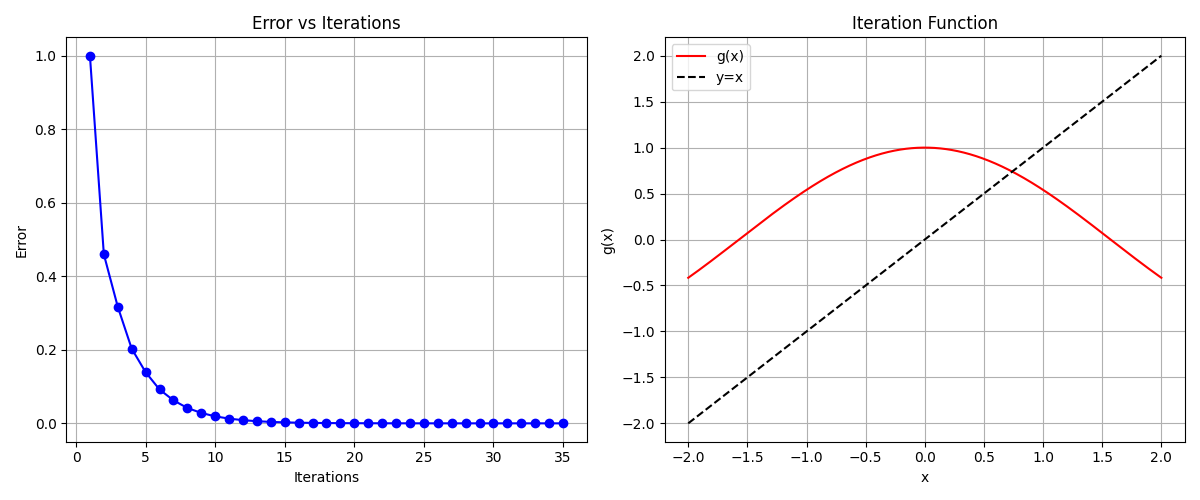
**g(x)=**

**Initial guess: x0=2**

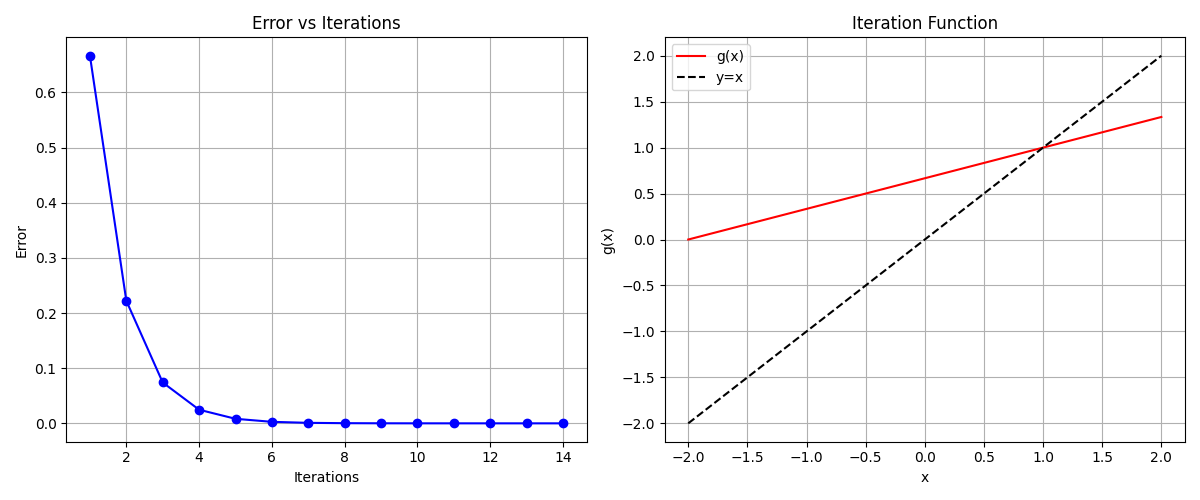
| **Iteration** | **xₙ** | **g(xₙ)** | **Error** |
| --- | --- | --- | --- |
| 1 | 1.250000 | 1.025000 | 0.750000 |
| 2 | 1.025000 | 1.000305 | 0.225000 |
| 3 | 1.000305 | 1.000000 | 0.024695 |
| 4 | 1.000000 | 1.000000 | 0.000305 |
| 5 | 1.000000 | 1.000000 | 0.000000 |

## Graphs

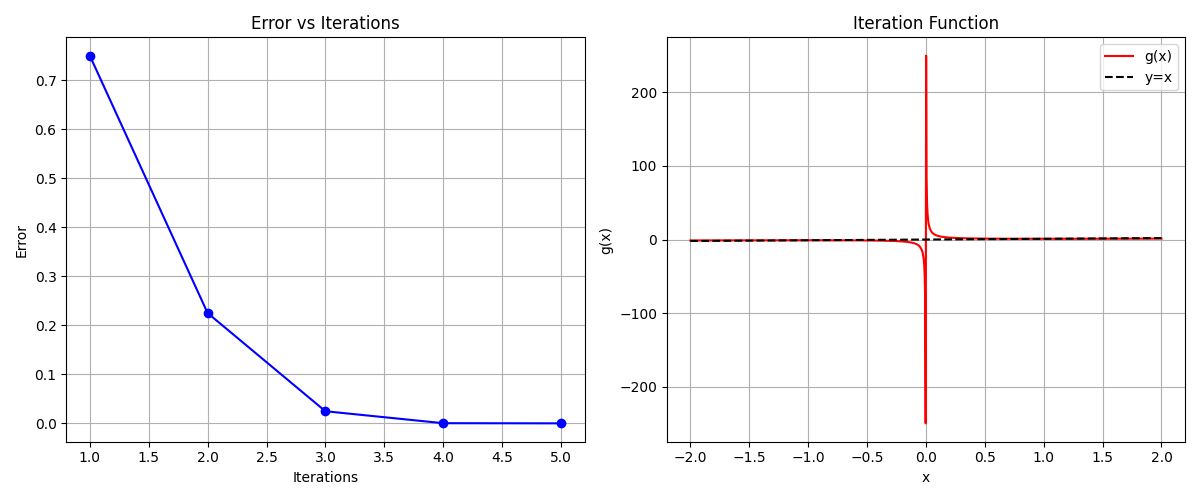
**g(x)=cos(x)**



**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 |

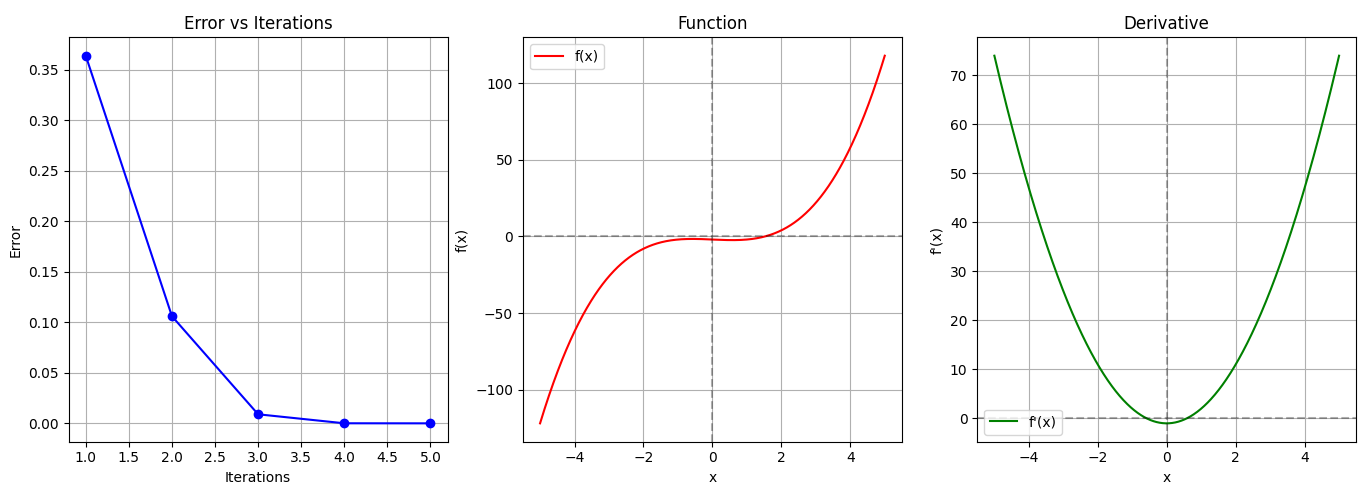
**f(x) = ex – 5x**

**Initial guess: x0=2**

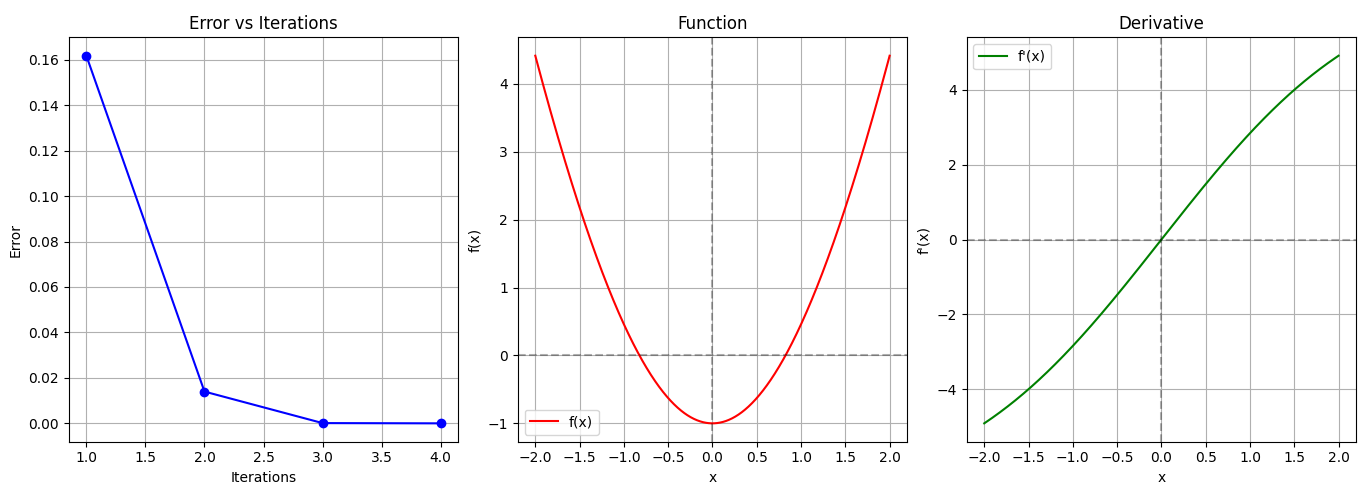
| **Iteration** | **xₙ** | **f(xₙ)** | **Error** |
| --- | --- | --- | --- |
| 1 | 3.092877 | 6.576008 | 1.092877 |
| 2 | 2.706970 | 1.448952 | 0.385907 |
| 3 | 2.561839 | 0.150436 | 0.145130 |
| 4 | 2.542939 | 0.002300 | 0.018900 |
| 5 | 2.542641 | 0.000001 | 0.000298 |
| 6 | 2.542641 | 0.000000 | 0.000000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

## Graphs

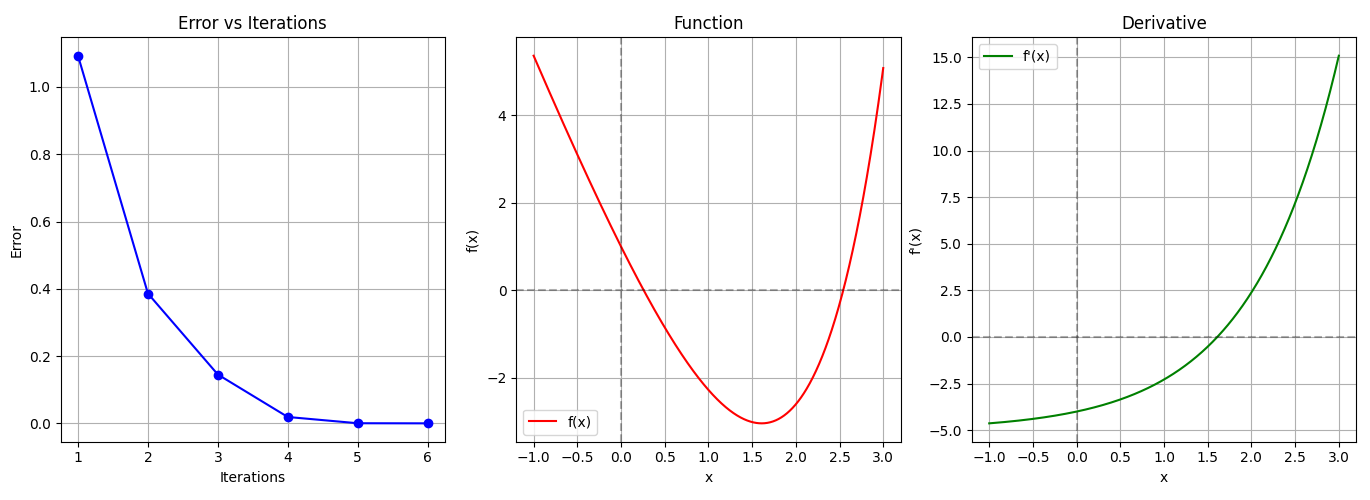
**f(x)=x3−x−2**



**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 |

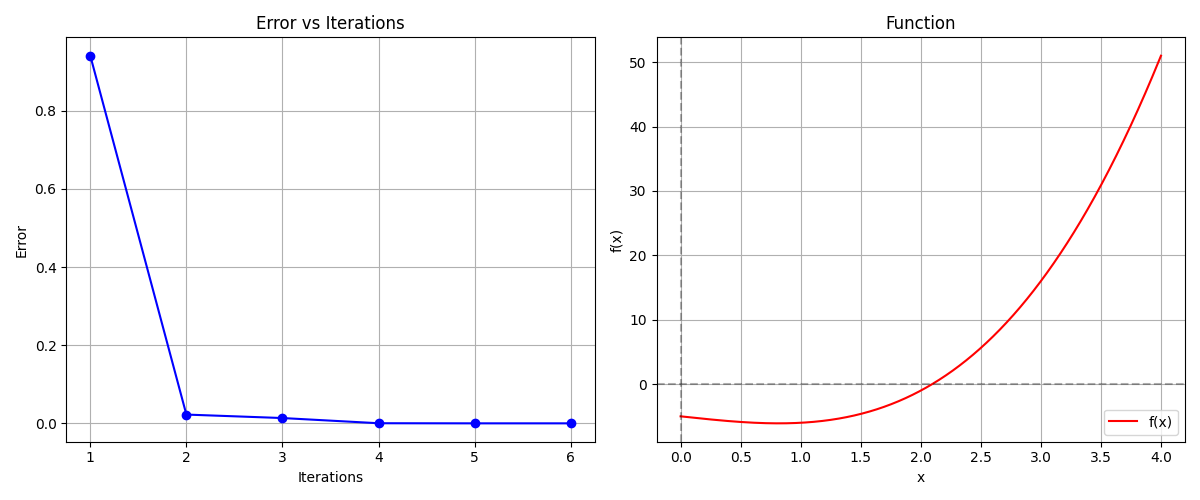
**f(x) = ex – 5x**

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

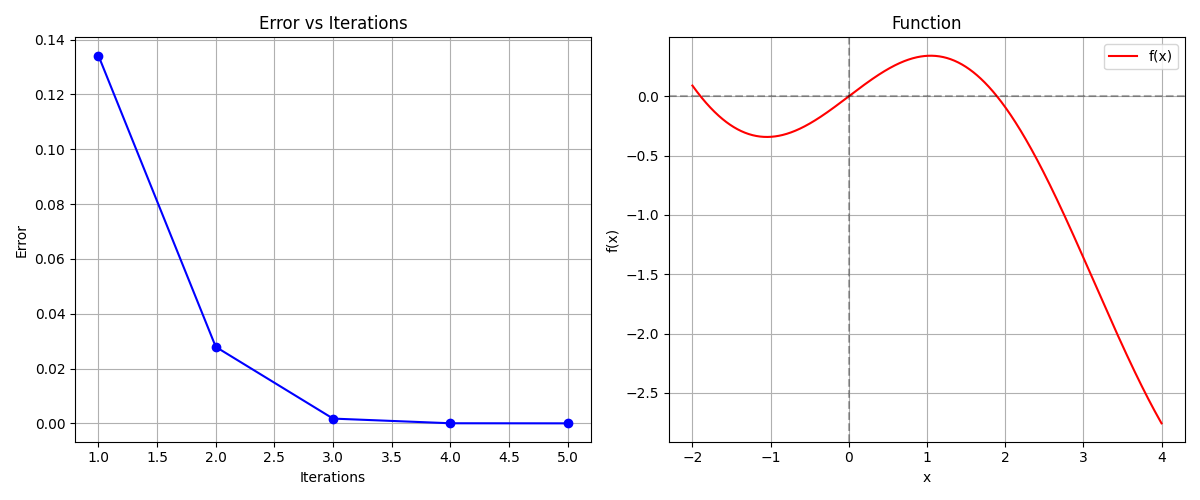
| **Iteration** | **xₙ** | **f(xₙ)** | **Error** |
| --- | --- | --- | --- |
| 1 | -5.930558 | 29.655449 | 7.930558 |
| 2 | 1.358272 | -2.901894 | 7.288831 |
| 3 | 0.708606 | -1.511871 | 0.649666 |
| 4 | 0.001990 | 0.992043 | 0.706616 |
| 5 | 0.281949 | -0.084033 | 0.279959 |
| 6 | 0.260086 | -0.003389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

## Graphs

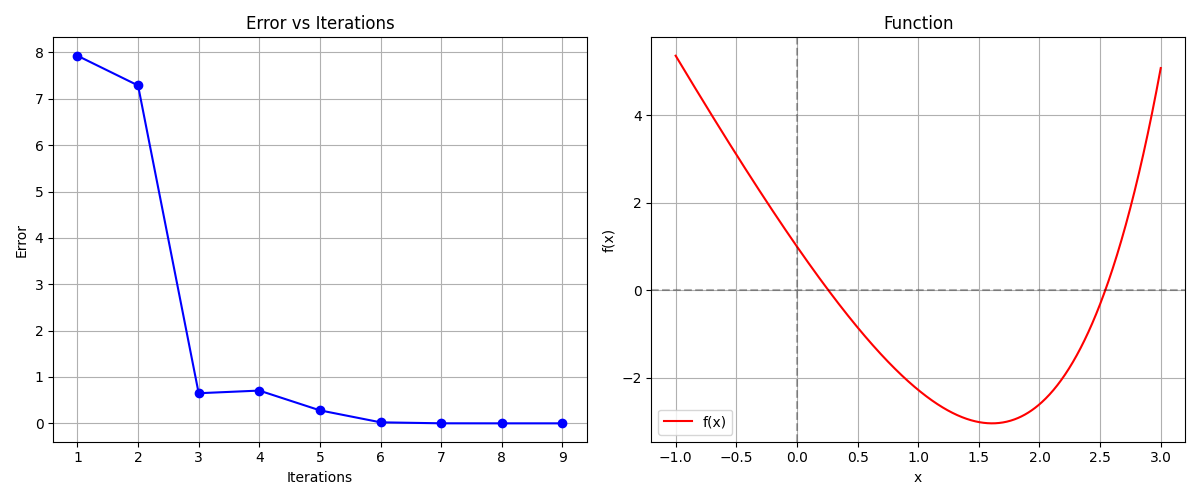
**g(x)=x3 – 2x - 5**



**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**

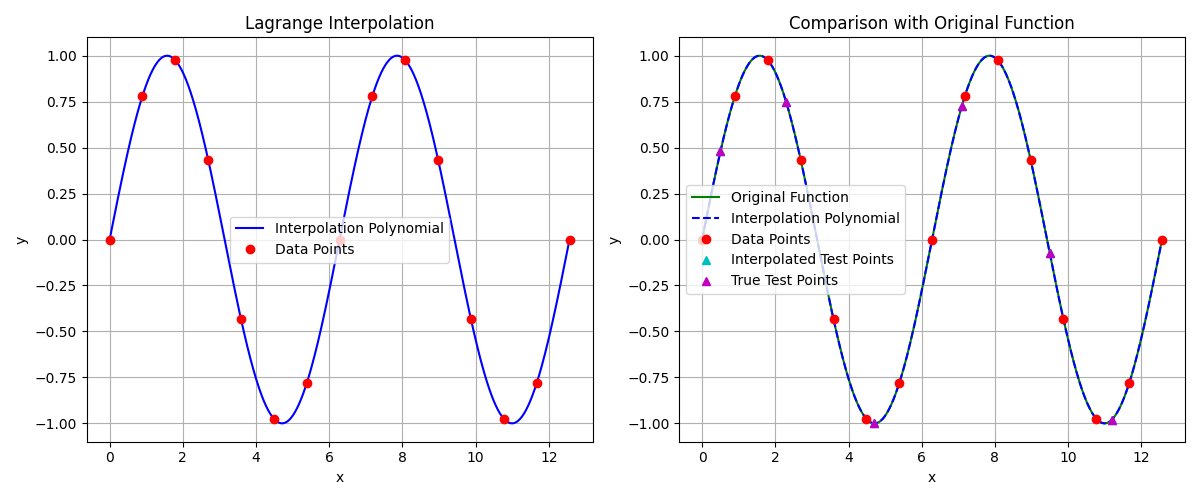
| **Index** | **x** | **y** |
| --- | --- | --- |
| 0 | -5.0000 | 0.0385 |
| 1 | -4.2857 | 0.0516 |
| 2 | -3.5714 | 0.0727 |
| 3 | -2.8571 | 0.1091 |
| 4 | -2.1429 | 0.1788 |
| 5 | -1.4286 | 0.3289 |
| 6 | -0.7143 | 0.6622 |
| 7 | 0.0000 | 1.0000 |
| 8 | 0.7143 | 0.6622 |
| 9 | 1.4286 | 0.3289 |
| 10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

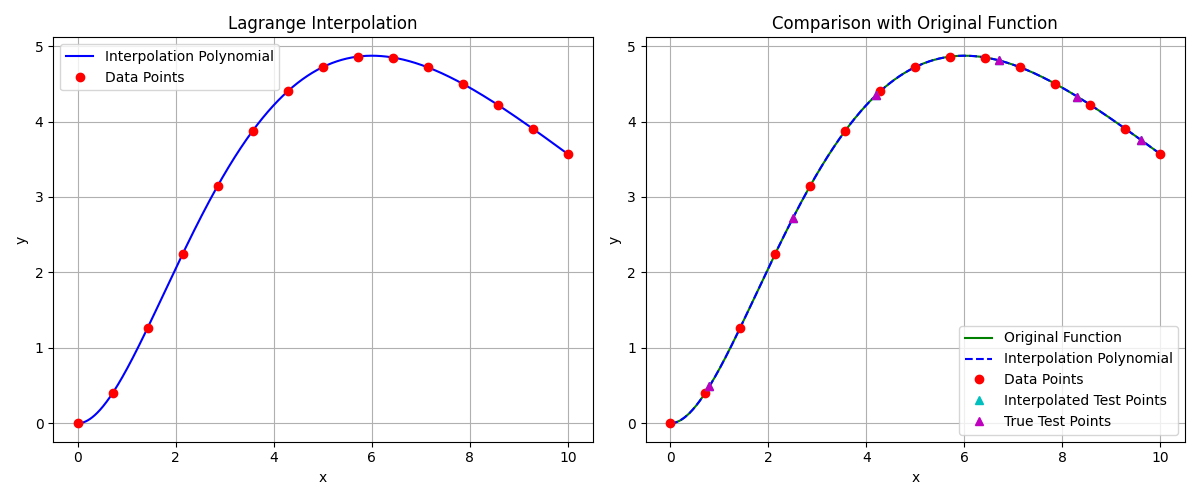
| **x** | **Interpolated** | **Function** | **Error** |
| --- | --- | --- | --- |
| -4.2000 | -0.4481 | 0.0536 | 5.02e-01 |
| -2.7000 | 0.0721 | 0.1206 | 4.85e-02 |
| -1.3000 | 0.3589 | 0.3717 | 1.29e-02 |
| 0.8000 | 0.6030 | 0.6098 | 6.77e-03 |
| 2.4000 | 0.0993 | 0.1479 | 4.86e-02 |
| 3.9000 | -0.5711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

## Graphs

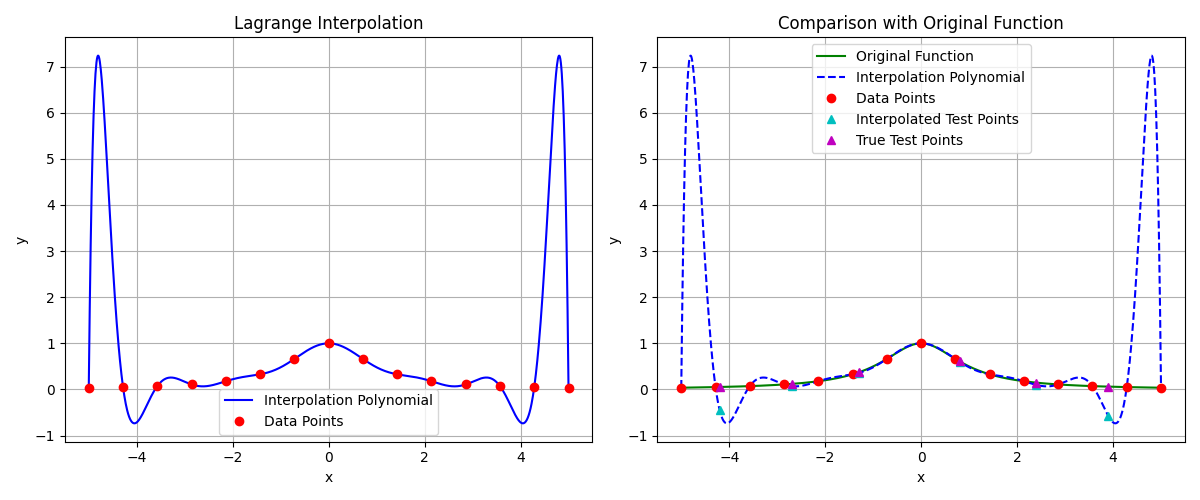
**f(x) = sin(x)**



**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**

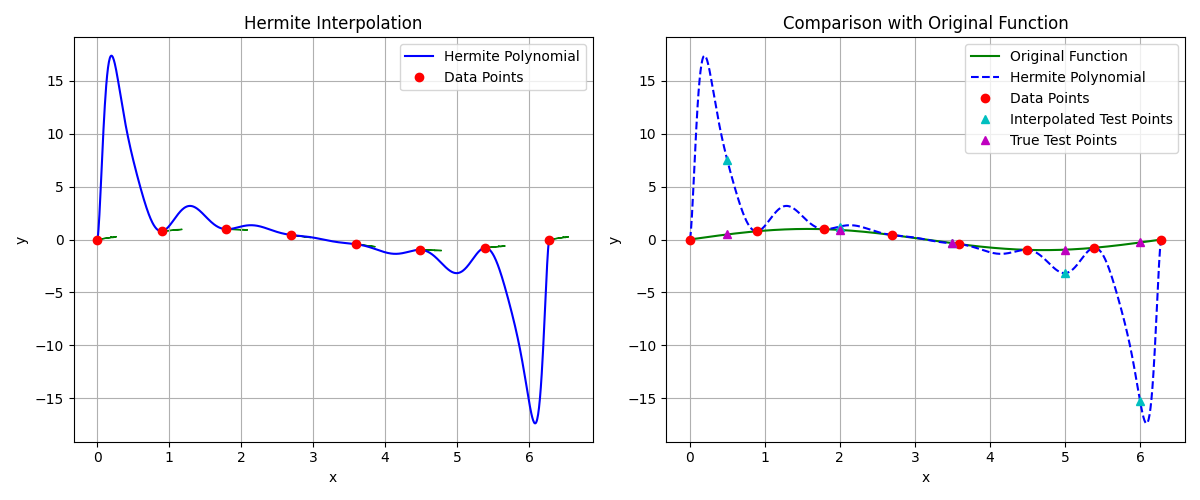
| **Index** | **x** | **y** | **dy/dx** |
| --- | --- | --- | --- |
| 0 | -5.0000 | 0.0385 | 0.0148 |
| 1 | -4.2857 | 0.0516 | 0.0229 |
| 2 | -3.5714 | 0.0727 | 0.0378 |
| 3 | -2.8571 | 0.1091 | 0.0681 |
| 4 | -2.1429 | 0.1788 | 0.1371 |
| 5 | -1.4286 | 0.3289 | 0.3090 |
| 6 | -0.7143 | 0.6622 | 0.6264 |
| 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**

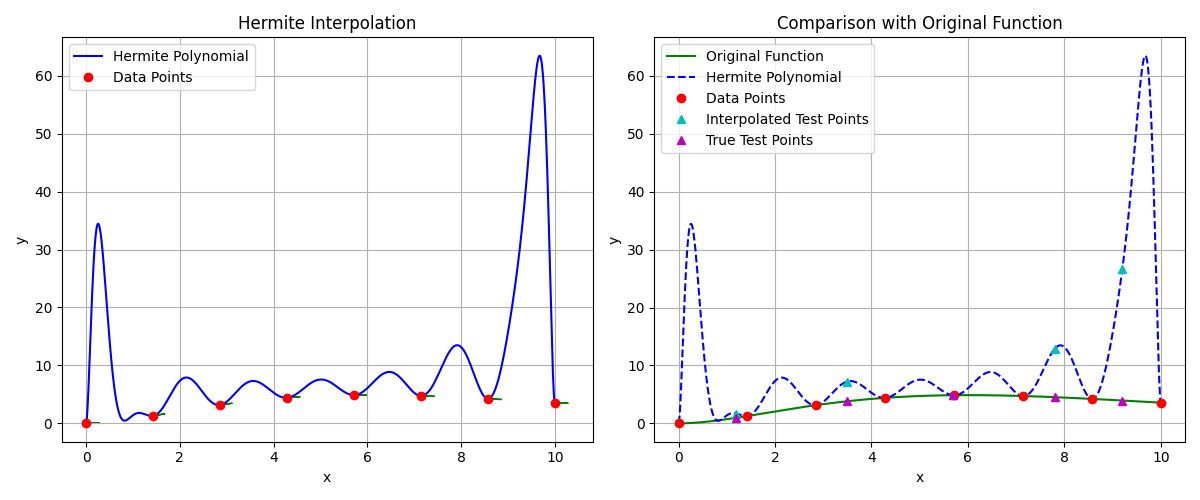
| **x** | **Interpolated** | **Function** | **Error** |
| --- | --- | --- | --- |
| -4.2000 | -735.9877 | 0.0536 | 7.36e+02 |
| -3.1000 | -1.8531 | 0.0943 | 1.95e+00 |
| -1.8000 | 0.6788 | 0.2358 | 4.43e-01 |
| -0.7000 | 0.6737 | 0.6711 | 2.55e-03 |
| 0.8000 | 0.6946 | 0.6098 | 8.48e-02 |
| 2.3000 | 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

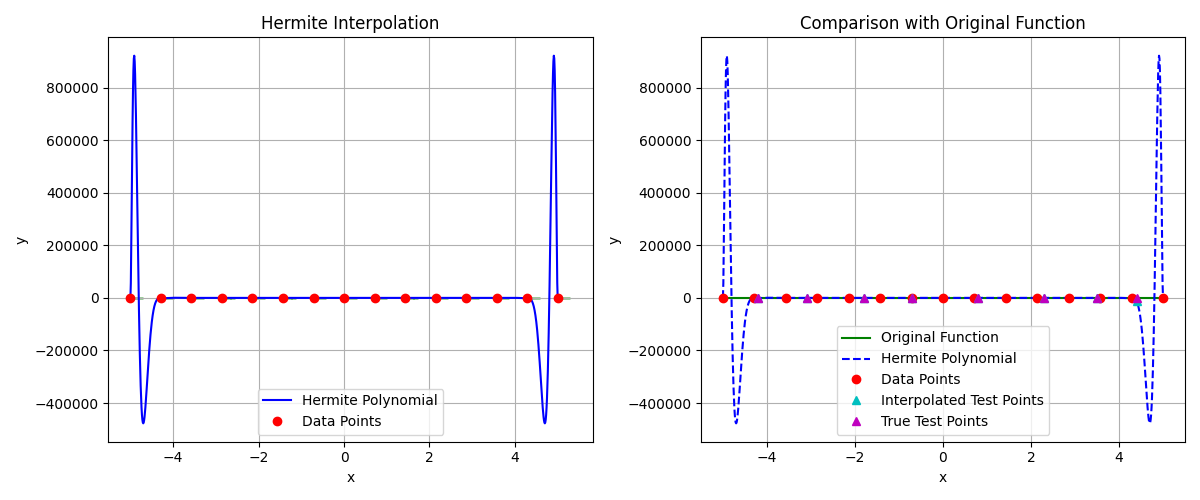
**f(x) = sin(x)**



**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 falls
* 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**

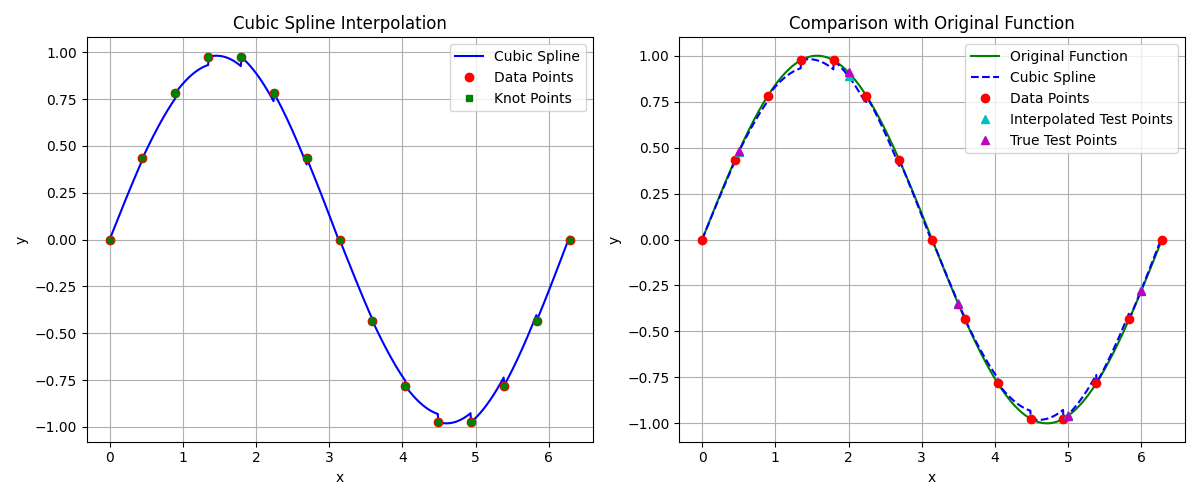
| **Index** | **x** | **y** |
| --- | --- | --- |
| 0 | -5.0000 | 0.0385 |
| 1 | -4.2857 | 0.0516 |
| 2 | -3.5714 | 0.0727 |
| 3 | -2.8571 | 0.1091 |
| 4 | -2.1429 | 0.1788 |
| 5 | -1.4286 | 0.3289 |
| 6 | -0.7143 | 0.6622 |
| 7 | 0.0000 | 1.0000 |
| 8 | 0.7143 | 0.6622 |
| 9 | 1.4286 | 0.3289 |
| 10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

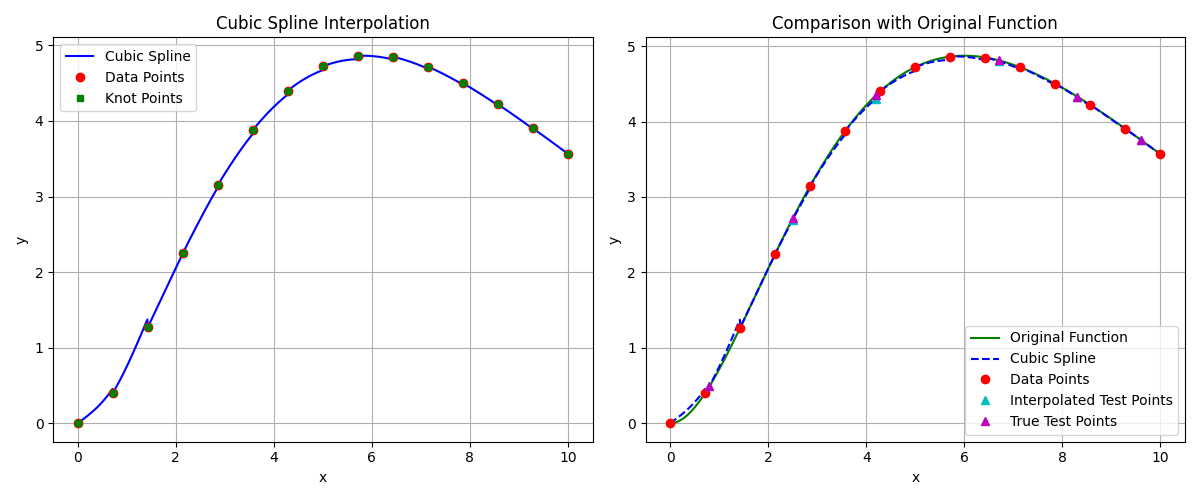
| **x** | **Interpolated** | **Function** | **Error** |
| --- | --- | --- | --- |
| -4.2000 | 0.0539 | 0.0536 | 2.73e-04 |
| -3.1000 | 0.0973 | 0.0943 | 3.07e-03 |
| -1.8000 | 0.2476 | 0.2358 | 1.18e-02 |
| -0.7000 | 0.6701 | 0.6711 | 1.09e-03 |
| 0.8000 | 0.6171 | 0.6098 | 7.33e-03 |
| 2.3000 | 0.1621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

## Graphs

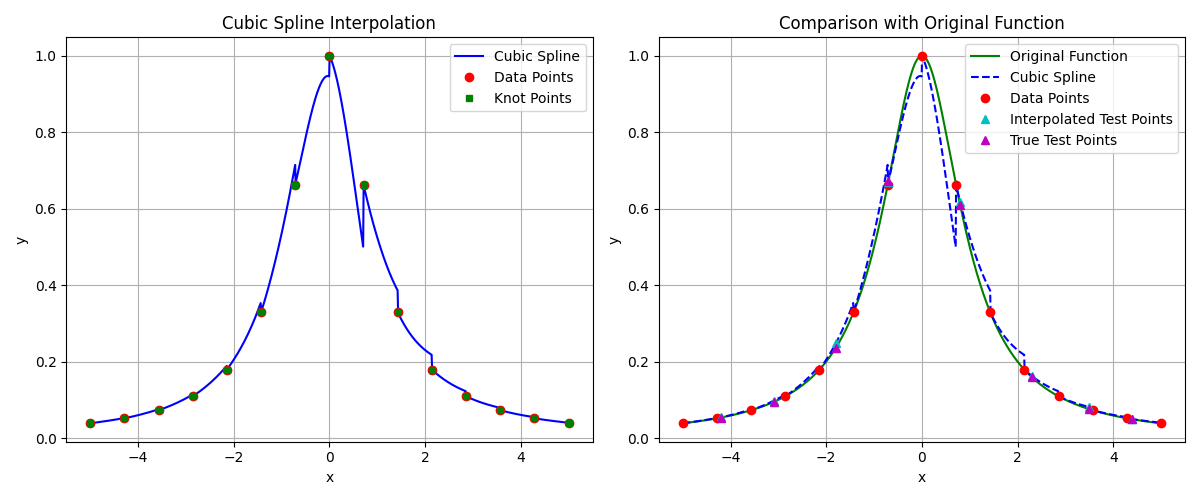
**f(x) = sin(x)**



**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.