Computational Mathematics, Assignment 1: Root-Finding Methods

Rakhmetollayev Askhat

Group: SE-2416

Selected function:x ** 3 -x-1=0

Imports and helper functions:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

def f(x):
    return x**3 - x - 1

def df(x):
    return 3*x**2 - 1

def g(x):
    return (x + 1)**(1/3)

def print_table(table, header=("n", "x_n", "f(x_n)", "error")):
    print(f"{header[0]}\t{header[1]}\t\t{header[2]}\t\t{header[3]}")
    for r in table:
        print(f"{r[0]}\t{r[1]:.6f}\t{r[2]:+.6f}\t{r[3]:.6f}")

def plot_function_and_root(f, root, title, xmin=0, xmax=2.0):
    X = np.linspace(xmin, xmax, 400)
    plt.figure(figsize=(7,5))
    plt.plot(X, f(X))
    plt.axhline(0, color="black")
    if root is not None:
        plt.scatter(root, f(root), color="red")
        plt.text(root, f(root)+0.02, f"root ≈ {root:.6f}", ha="center")
    plt.title(title)
    plt.xlabel("x")
    plt.ylabel("f(x)")
    plt.grid()
    plt.show()

def plot_convergence(errors, title, ylabel="Error"):
    plt.figure(figsize=(6,4))
    plt.plot(errors, marker='o')
    plt.yscale("log")
    plt.xlabel("Iteration")
    plt.ylabel(ylabel)
    plt.title(title)
    plt.grid()
    plt.show()
```

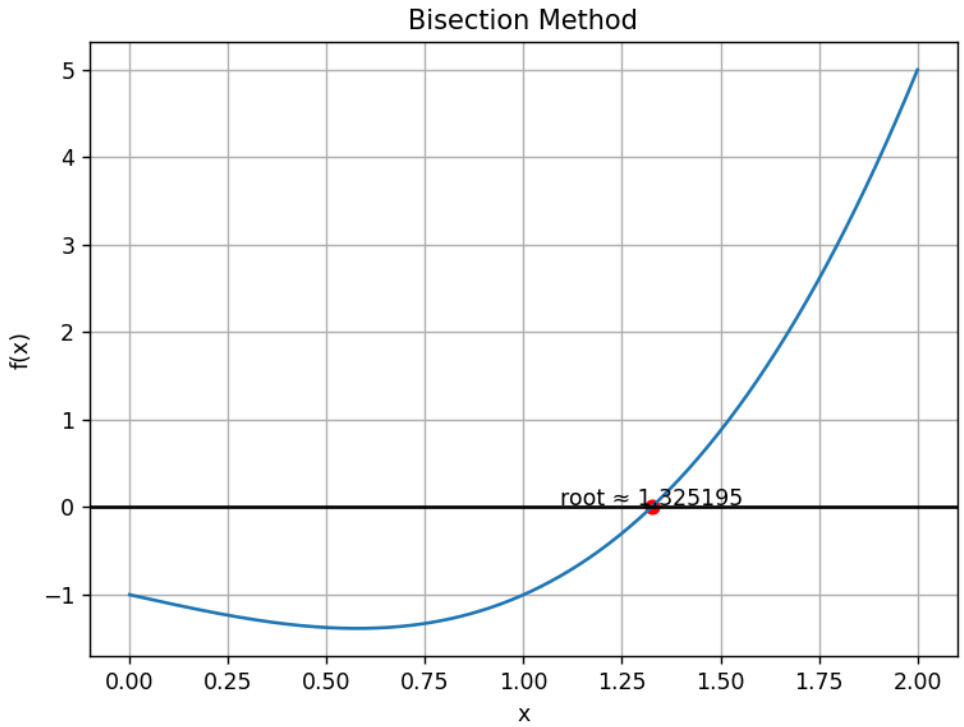
Bisection Method

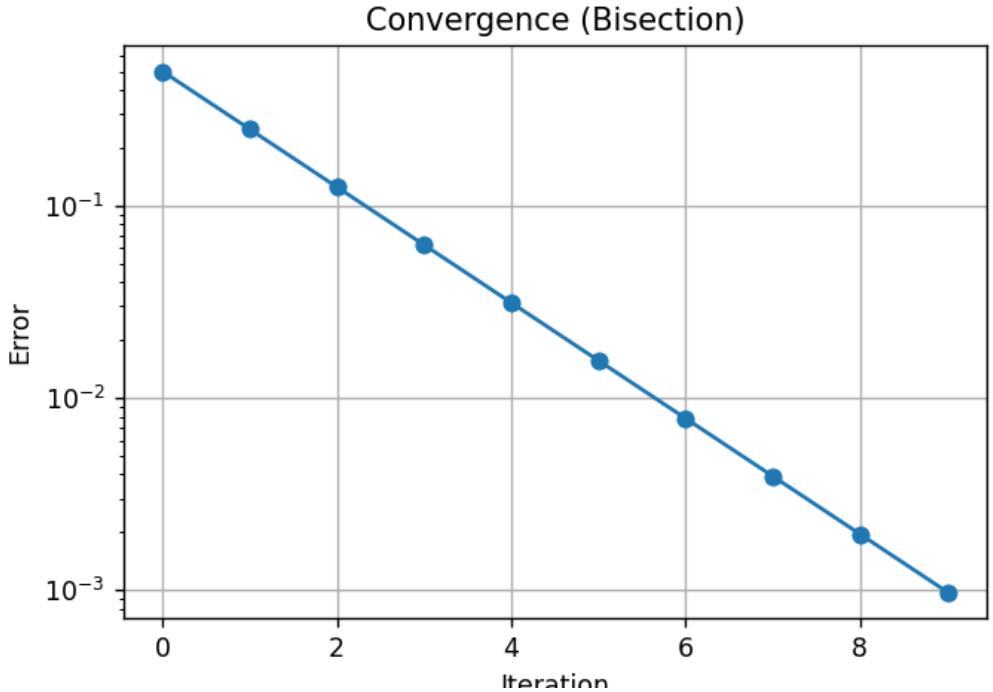
The Bisection Method is a bracketing method that repeatedly halves the interval containing the root. It always converges if the initial interval is valid.The Bisection Method is a robust and reliable root-finding algorithm because it guarantees convergence as long as the initial interval contains a sign change. However, its main limitation is slow convergence. The interval width is reduced by a fixed factor of 2 at each iteration, which results in linear convergence. As a consequence, a relatively large number of iterations is required to achieve high accuracy compared to open methods. Additionally, the method requires prior knowledge of an interval where the function changes sign, which may not always be easy to determine in practice.

```
def bisection(f, a, b, eps=1e-3, Nmax=100):
    if a >= b:
        return None, [], "invalid interval order"
    if eps <= 0:
        return None, [], "invalid tolerance"
    if Nmax <= 0:
        return None, [], "invalid max iterations"
    if f(a) * f(b) > 0:
        return None, [], "invalid interval (no sign change)"

    table = []

    for n in range(1, Nmax + 1):
        c = (a + b) / 2
        fc = f(c)
        error = (b - a) / 2
        table.append([n, c, fc, error])

        if error < eps or abs(fc) < eps:
            return c, table, "tolerance reached"

        if f(a) * fc < 0:
            b = c
        else:
            a = c

    return c, table, "max iterations reached"

root, table_b, reason = bisection(f, 1.0, 2.0)

print("Bisection Method")
print("Root estimate:", root)
print("Iterations:", len(table_b))
print("Stop reason:", reason, "\n")

print_table(table_b)
```

```
plot_function_and_root(f, root, "Bisection Method", xmin=0.0, xmax=2.0)
plot_convergence([r[3] for r in table_b], "Convergence (Bisection)")
```

```
C:\Users\Асхат\PycharmProjects\COMPUT\.venv\Scripts\python.exe C:\Users\Асхат\PycharmProjects\COMPUT\bisection.py
Bisection Method
Root estimate: 1.3251953125
Iterations: 10
Stop reason: tolerance reached

n   x_n     f(x_n)      error
1   1.500000    +0.875000   0.500000
2   1.250000    -0.296875   0.250000
3   1.375000    +0.224609   0.125000
4   1.312500    -0.051514   0.062500
5   1.343750    +0.082611   0.031250
6   1.328125    +0.014576   0.015625
7   1.320312    -0.018711   0.007812
8   1.324219    -0.002128   0.003906
9   1.326172    +0.006209   0.001953
10  1.325195    +0.002037   0.000977

Process finished with exit code 0
```



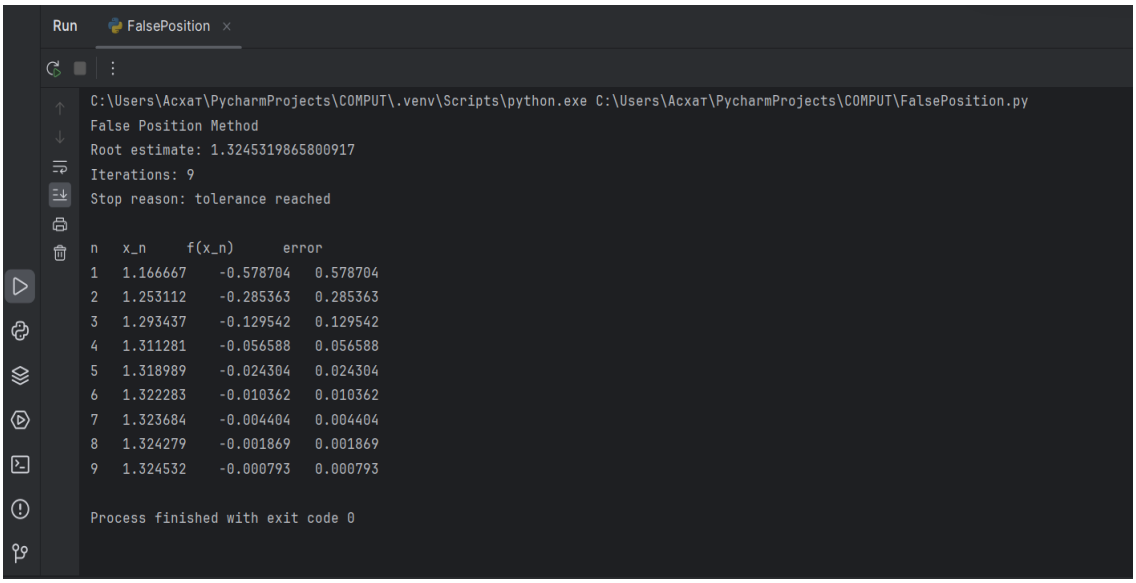


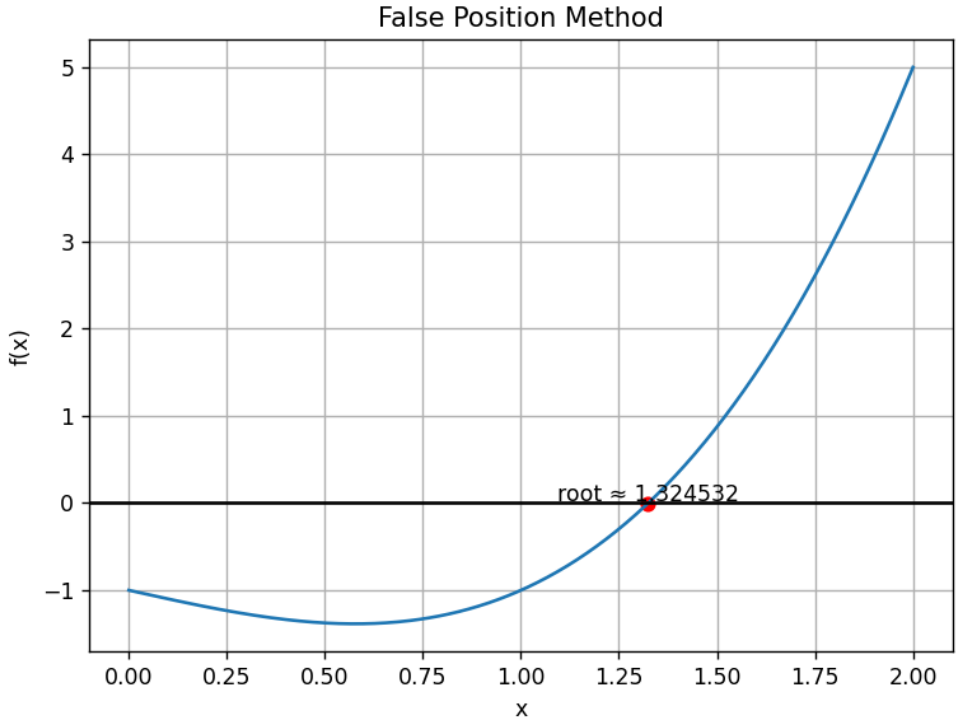
False Position (Regula Falsi) Method

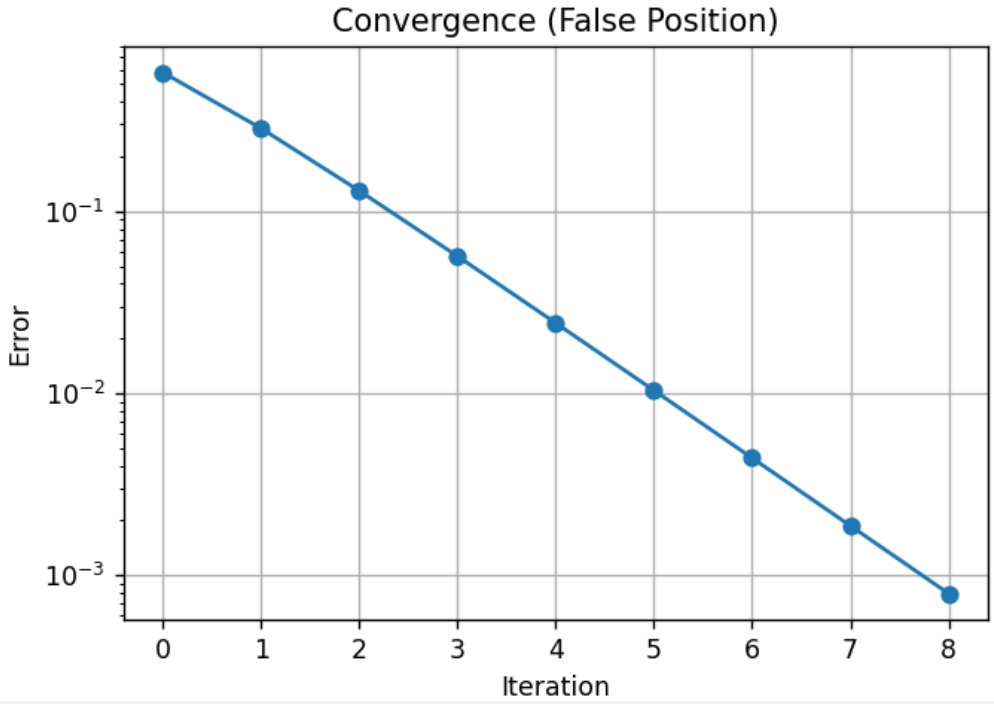
The False Position (Regula Falsi) method is a bracketing method that uses linear interpolation instead of midpoint selection.The False Position Method improves upon the Bisection Method by using a linear interpolation instead of a midpoint, often resulting in faster convergence. Nevertheless, the method may suffer from endpoint stagnation, where one endpoint of the interval remains fixed for many iterations. This can slow down convergence significantly, especially when the function is highly nonlinear near one endpoint. Like all bracketing methods, it also requires an initial interval with opposite signs of the function values.

```
def false_position(f, a, b, eps=1e-3, Nmax=100):
    if a >= b:
        return None, [], "invalid interval order"
    if eps <= 0:
        return None, [], "invalid tolerance"
    if Nmax <= 0:
        return None, [], "invalid max iterations"
    if f(a) * f(b) > 0:
        return None, [], "invalid interval (no sign change)"

    table = []
    fa, fb = f(a), f(b)
```

```
    for n in range(1, Nmax + 1):
        if fb - fa == 0:
            return None, table, "division by zero"
        c = (a * fb - b * fa) / (fb - fa)
        fc = f(c)
        error = abs(fc)
        table.append([n, c, fc, error])

        if abs(fc) < eps:
            return c, table, "tolerance reached"

        if fa * fc < 0:
            b, fb = c, fc
        else:
            a, fa = c, fc

    return c, table, "max iterations reached"


root_f, table_f, reason_f = false_position(f, 1.0, 2.0)

print("False Position Method")
print("Root estimate:", root_f)
print("Iterations:", len(table_f))
print("Stop reason:", reason_f, "\n")
print_table(table_f)

plot_function_and_root(f, root_f, "False Position Method", xmin=0.0, xmax=2.0)
plot_convergence([r[3] for r in table_f], "Convergence (False Position)")
```







Fixed-Point Iteration Method

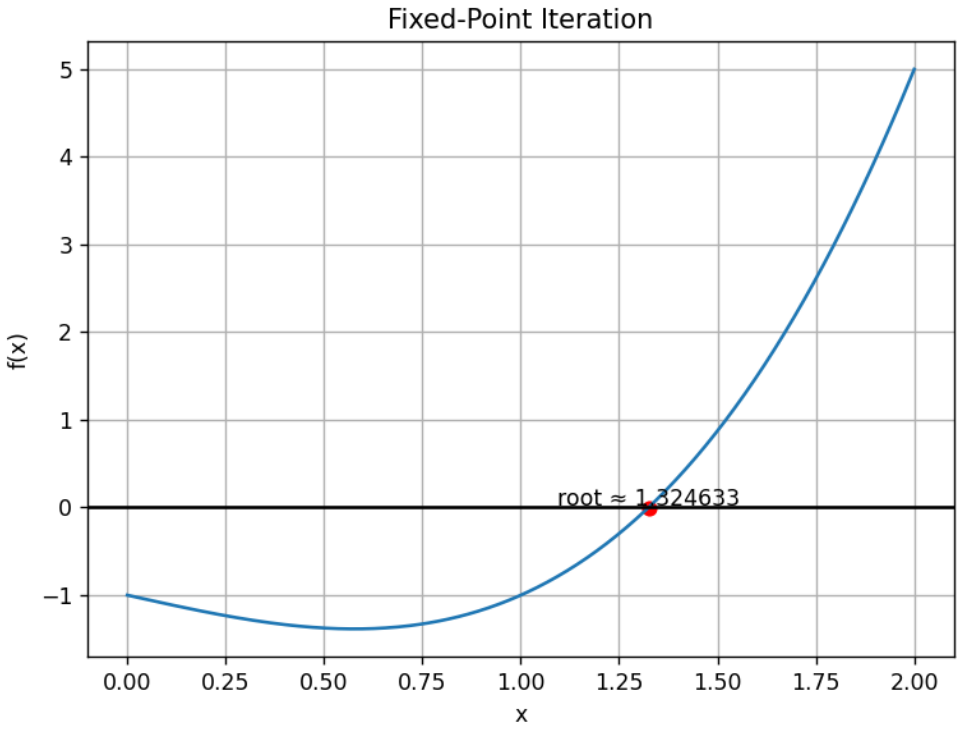
This method rewrites the equation in the form (x = g(x)) and iteratively applies the function until convergence.The Fixed Point Iteration method is simple to implement and computationally inexpensive per iteration. However, its convergence strongly depends on the choice of the iteration function g(x). In the condition |g'(x)|<1 is not satisfied near the root, the method may converge very slowly or even diverge. Compared to other methods, Fixed Point Iteration typically converges slower and is more sensitive to the initial guess, making it less reliable without proper theoretical analysis.

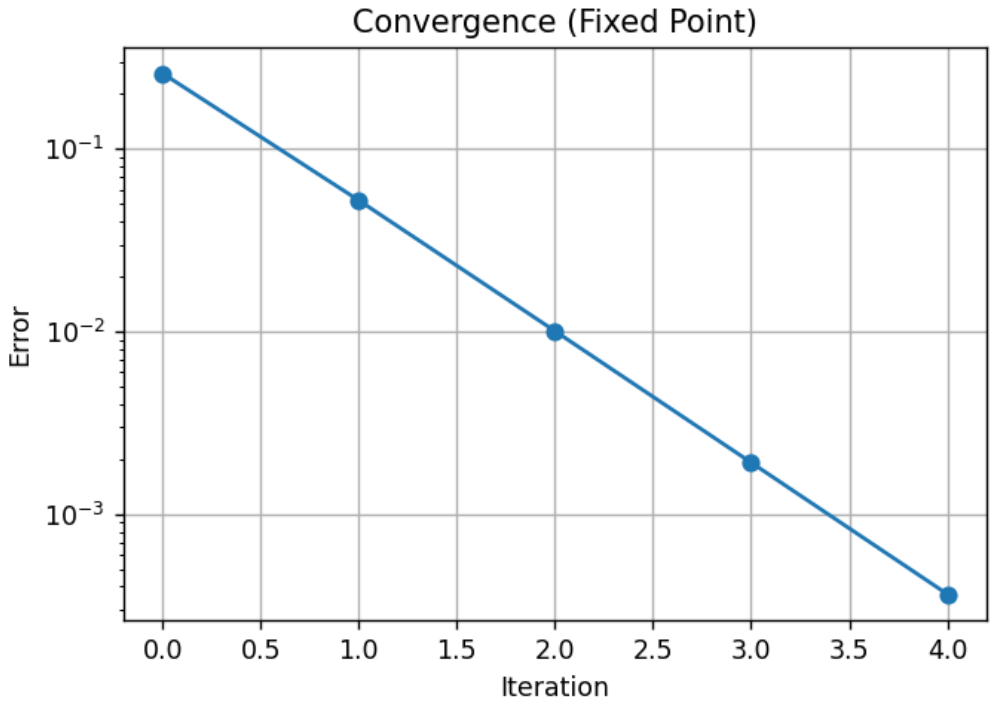
```
def fixed_point(g, f, x0, eps=1e-3, Nmax=100):
    if eps <= 0:
        return None, [], "invalid tolerance"
    if Nmax <= 0:
        return None, [], "invalid max iterations"

    table = []
    x = x0

    for n in range(1, Nmax + 1):
        x_new = g(x)
        fx_new = f(x_new)
        error = abs(x_new - x)
        table.append([n, x_new, fx_new, error])

        if error < eps:
            return x_new, table, "tolerance reached"

        x = x_new

    return x, table, "max iterations reached"


root_fp, table_fp, reason_fp = fixed_point(g, f, x0=1.0, eps=1e-3, Nmax=100)

print("Fixed-Point Iteration")
print("Root estimate:", root_fp)
print("Iterations:", len(table_fp))
print("Stop reason:", reason_fp, "\n")
print_table(table_fp)

plot_function_and_root(f, root_fp, "Fixed-Point Iteration", xmin=0.0, xmax=2.0)
plot_convergence([r[3] for r in table_fp], "Convergence (Fixed Point)")
```

Run Fixed-point

```
C:\Users\Асхат\PycharmProjects\COMPUT\.venv\Scripts\python.exe C:\Users\Асхат\PycharmProjects\COMPUT\Fixed-point.py
Fixed-Point Iteration
Root estimate: 1.3246326252509202
Iterations: 5
Stop reason: tolerance reached

n   x_n     f(x_n)      error
1   1.259921    -0.259921   0.259921
2   1.312294    -0.052373   0.052373
3   1.322354    -0.010060   0.010060
4   1.324269    -0.001915   0.001915
5   1.324633    -0.000364   0.000364

Process finished with exit code 0
```

MPUT > Fixed-point.py 6



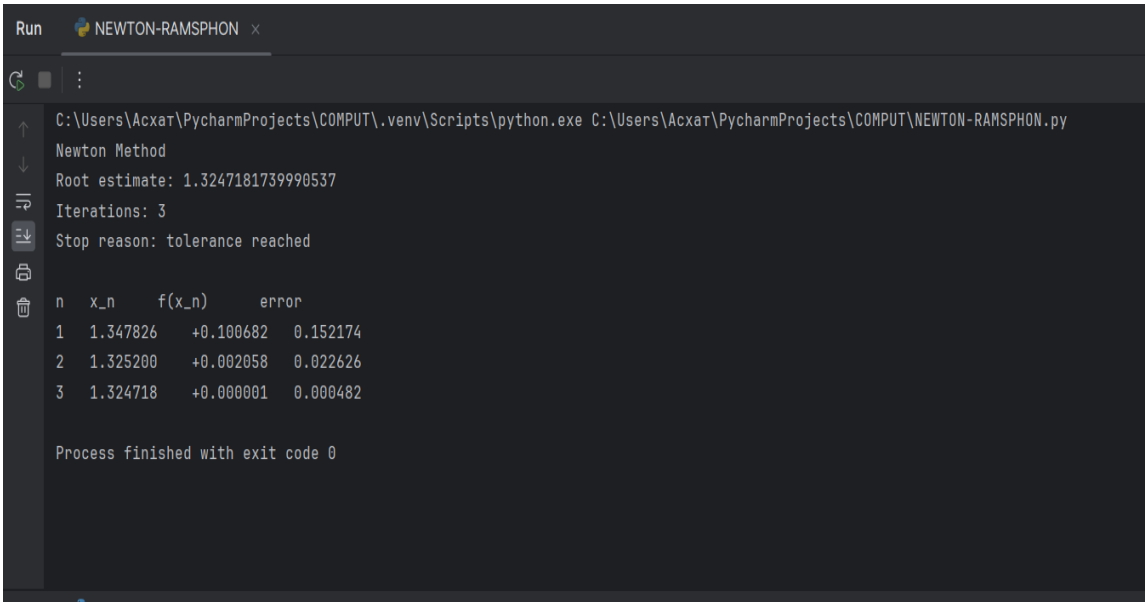


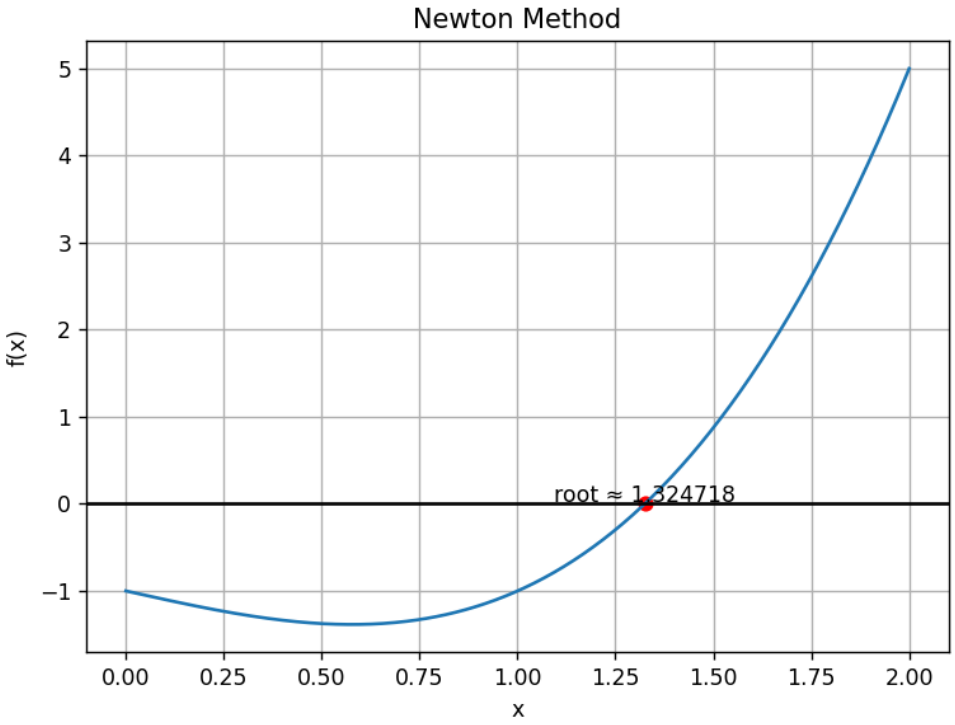
Newton–Raphson Method

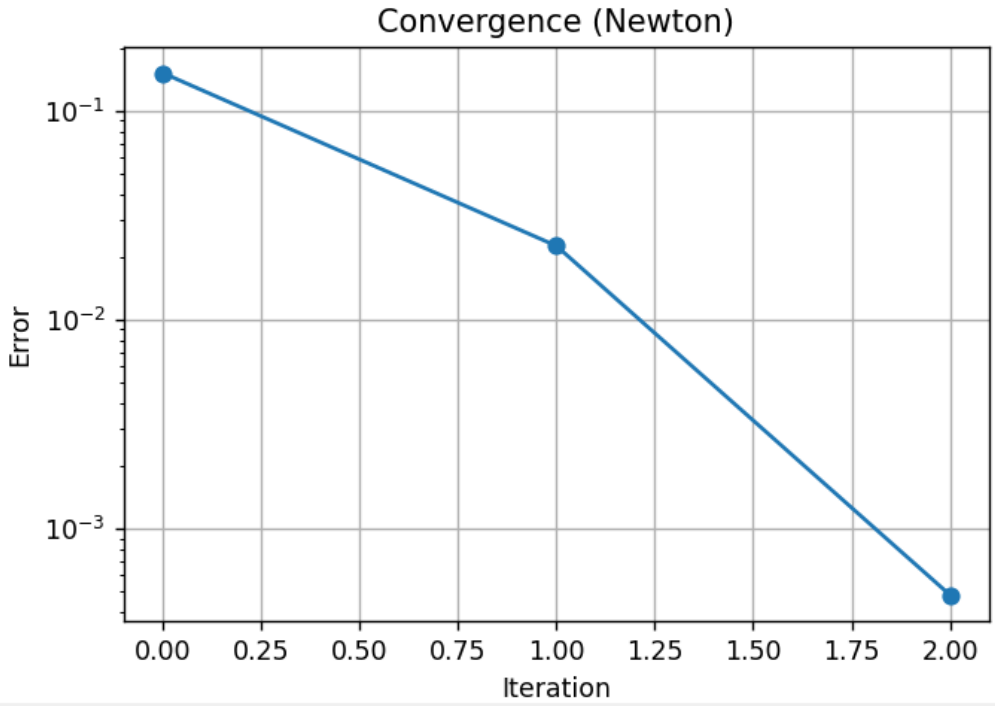
Newton's method uses the derivative of the function to achieve quadratic convergence near the root.The Newton–Raphson Method is highly efficient and exhibits quadratic convergence when the initial guess is sufficiently close to the root. However, it requires the computation of the derivative, which may be difficult or impossible for some functions. The method may also fail if the derivative is zero or very small near the current approximation. Additionally, a poor initial guess can lead to divergence or convergence to an unintended root.

```
def newton(f, df, x0, eps=1e-3, Nmax=100):
    if eps <= 0:
        return None, [], "invalid tolerance"
    if Nmax <= 0:
        return None, [], "invalid max iterations"

    table = []
    x = x0

    for n in range(1, Nmax + 1):
        dfx = df(x)
        if dfx == 0:
            return None, table, "division by zero in derivative"

        x_new = x - f(x) / dfx
        fx_new = f(x_new)
```

```
        error = abs(x_new - x)
        table.append([n, x_new, fx_new, error])

        if error < eps or abs(fx_new) < eps:
            return x_new, table, "tolerance reached"

        x = x_new

    return x, table, "max iterations reached"

root_n, table_n, reason_n = newton(f, df, x0=1.5, eps=1e-3, Nmax=100)

print("Newton Method")
print("Root estimate:", root_n)
print("Iterations:", len(table_n))
print("Stop reason:", reason_n, "\n")
print_table(table_n)

plot_function_and_root(f, root_n, "Newton Method", xmin=0.0, xmax=2.0)
plot_convergence([r[3] for r in table_n], "Convergence (Newton)")
```







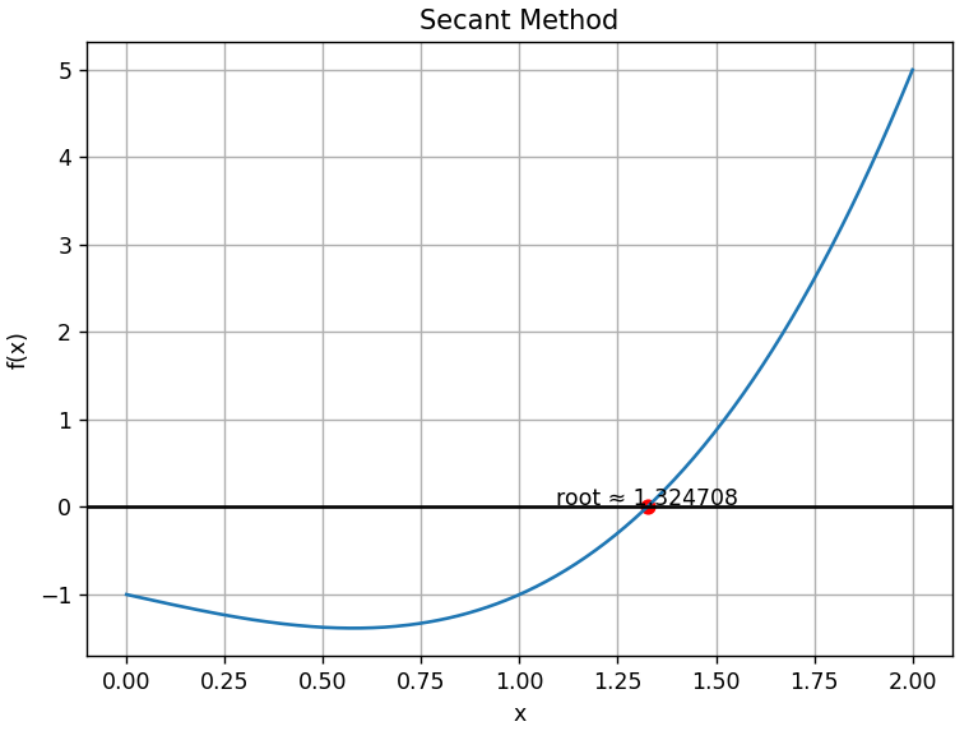
Secant Method

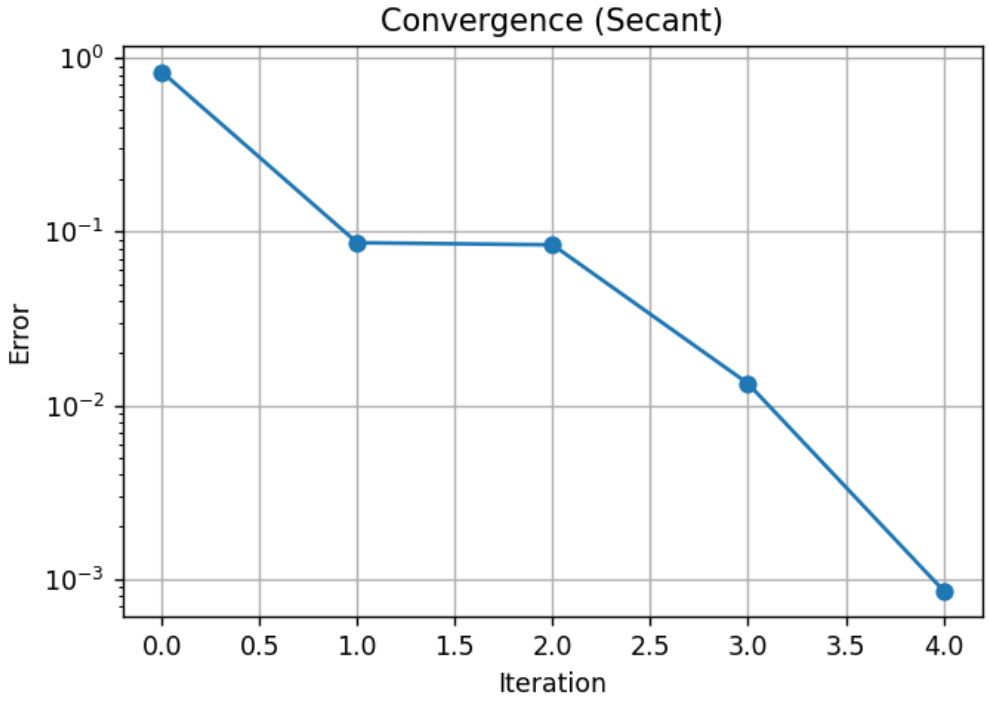
The Secant Method approximates the derivative numerically using two previous points.The Secant Method eliminates the need for explicit derivative computation by approximating it numerically, making it more practical than Newton's method in some cases. However, it does not guarantee convergence and is sensitive to the choice of initial guesses. If consecutive function values become equal, division by zero may occur. Although its convergence is faster than Fixed Point Iteration, it is generally slower and less stable than Newton's method.

```
def secant(f, x0, x1, eps=1e-3, Nmax=100):
    if eps <= 0:
        return None, [], "invalid tolerance"
    if Nmax <= 0:
        return None, [], "invalid max iterations"

    table = []
    x_prev, x = x0, x1

    for n in range(1, Nmax + 1):
        f_prev, f_curr = f(x_prev), f(x)
        if x == x_prev:
            return None, table, "division by zero in secant"

        x_new = x - f_curr * (x - x_prev) / (f_curr - f_prev)
        fx_new = f(x_new)
        error = abs(x_new - x)
        table.append([n, x_new, fx_new, error])

        if error < eps or abs(fx_new) < eps:
            return x_new, table, "tolerance reached"

        x_prev, x = x, x_new

    return x, table, "max iterations reached"

root_s, table_s, reason_s = secant(f, x0=1.0, x1=2.0, eps=1e-3, Nmax=100)

print("Secant Method")
print("Root estimate:", root_s)
print("Iterations:", len(table_s))
print("Stop reason:", reason_s, "\n")
print_table(table_s)

plot_function_and_root(f, root_s, "Secant Method", xmin=0.0, xmax=2.0)
plot_convergence([r[3] for r in table_s], "Convergence (Secant)")
```

Run Secant

```
C:\Users\Асхат\PycharmProjects\COMPUT\.venv\Scripts\python.exe C:\Users\Асхат\PycharmProjects\COMPUT\Secant.py
Secant Method
Root estimate: 1.324707936532088
Iterations: 5
Stop reason: tolerance reached

n   x_n     f(x_n)      error
1   1.166667    -0.578704   0.833333
2   1.253112    -0.285363   0.086445
3   1.337206    +0.053881   0.084094
4   1.323850    -0.003698   0.013356
5   1.324708    -0.000043   0.000858

Process finished with exit code 0
```



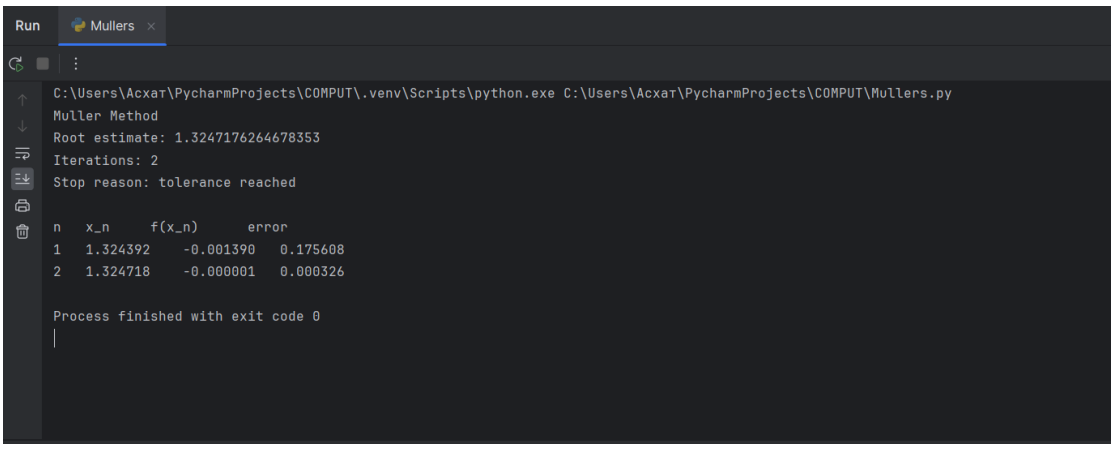


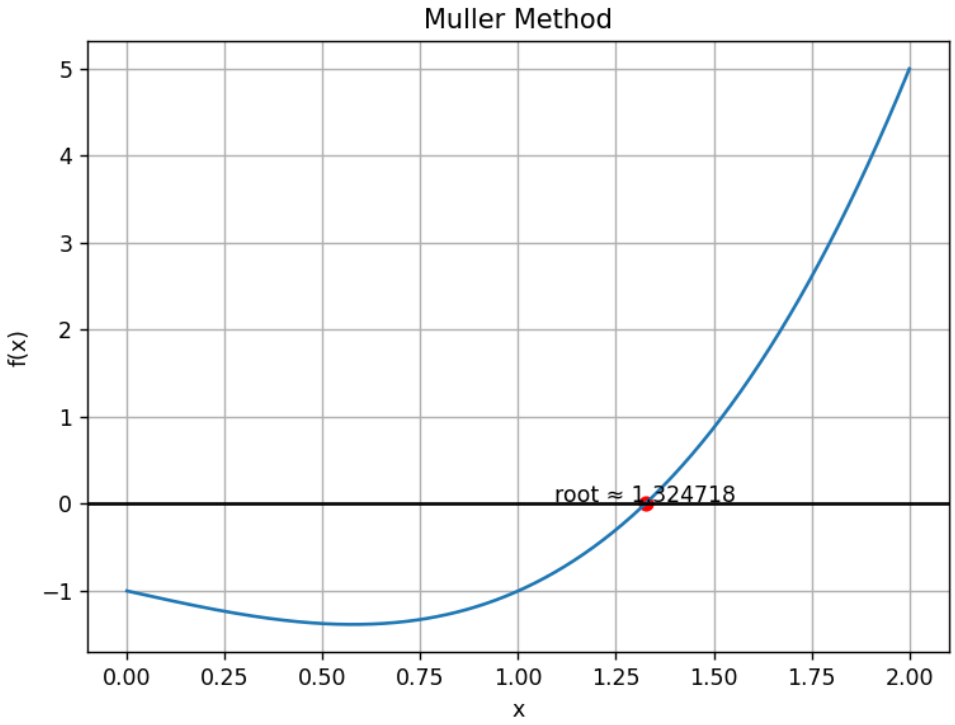
Muller's Method

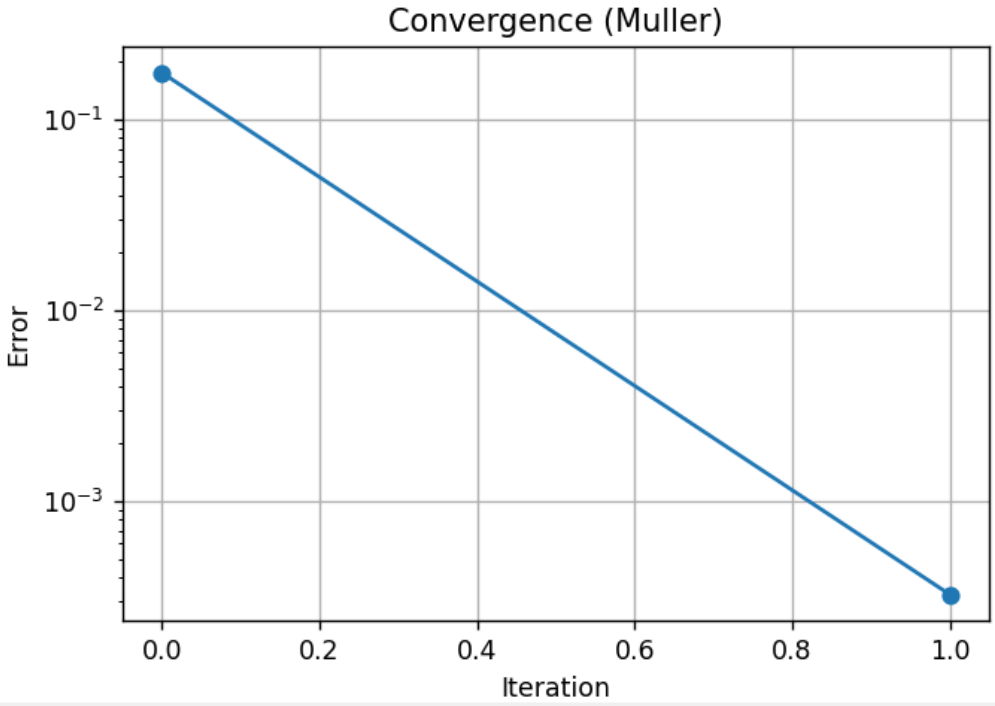
Muller’s Method uses quadratic interpolation and can converge faster than the Secant Method.Muller’s Method uses quadratic interpolation and often converges faster than the Secant Method. Despite its efficiency, the algorithm is more complex and computationally expensive per iteration. The method may generate complex intermediate values if the discriminant becomes negative, which can complicate implementation and interpretation. Additionally, Muller’s Method requires three initial points, and poor selection of these points can lead to instability or divergence.

```
def muller(f, x0, x1, x2, eps=1e-3, Nmax=100):

    # basic input checks
    if not (x0 < x1 < x2):
        return None, [], "invalid initial points order"
    if eps <= 0:
        return None, [], "invalid tolerance"
    if Nmax <= 0:
        return None, [], "invalid max iterations"

    table = []

    for n in range(1, Nmax + 1):
        f0, f1, f2 = f(x0), f(x1), f(x2)
        h0, h1 = x1 - x0, x2 - x1
```

```
        if h0 == 0 or h1 == 0:
            return None, table, "division by zero in h"

        d0 = (f1 - f0) / h0
        d1 = (f2 - f1) / h1

        a = (d1 - d0) / (h1 + h0)
        b = a * h1 + d1
        c = f2

        D2 = b*b - 4*a*c
        if D2 < 0:
            return None, table, "negative discriminant"

        D = np.sqrt(D2)
        denom = b + D if abs(b + D) > abs(b - D) else b - D
        if denom == 0:
            return None, table, "division by zero in step"

        x3 = x2 - 2*c / denom
        error = abs(x3 - x2)
        table.append([n, x3, f(x3), error])

        if error < eps:
            return x3, table, "tolerance reached"

        x0, x1, x2 = x1, x2, x3

    return x3, table, "max iterations reached"

root_m, table_m, reason_m = muller(f, 1.0, 1.3, 1.5)

print("Muller Method")
print("Root estimate:", root_m)
print("Iterations:", len(table_m))
print("Stop reason:", reason_m, "\n")
print_table(table_m)

plot_function_and_root(f, root_m, "Muller Method", xmin=0.0, xmax=2.0)
plot_convergence([r[3] for r in table_m], "Convergence (Muller)")
```




Muller Method
f(x)
x
root ≈ 1.324718
5
4
3
2
1
0
−1
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


Convergence (Muller)
Error
Iteration
10^{-1}
10^{-2}
10^{-3}
0.0
0.2
0.4
0.6
0.8
1.0

Comparison and Conclusion

Bisection-10 Iterations

False Position-9 Iterations

Fixed-Point-5 Iterations

Newton-3 Iterations

Secant-5 Iterations

Muller-2 Iterations

I have used all 6 methods.Bracketing methods are robust but slow, while open methods converge faster but depend on initial guesses or derivatives.The numerical results confirm the theoretical convergence behavior of each method

Github:https://github.com/Askhat111/ComputMath1