LAB 2

1. Bisection method

Working principle The Bisection Method is a root-finding algorithm that repeatedly bisects an interval and then selects the subinterval in which a root must lie. The method requires two initial points, a and b, such that the function values at these points have opposite signs. The algorithm iteratively narrows down the search interval by calculating the midpoint and evaluating the function at this point until the root is approximated to a desired precision.

Pseudo code:

1. Given initial values a, b, and tolerance ε.
2. Check if f(a) and f(b) have opposite signs.
3. Repeat until the interval length is less than ε:

   a. Find midpoint m = (a + b) / 2.

   b. If f(m) is close to 0, then m is the root.

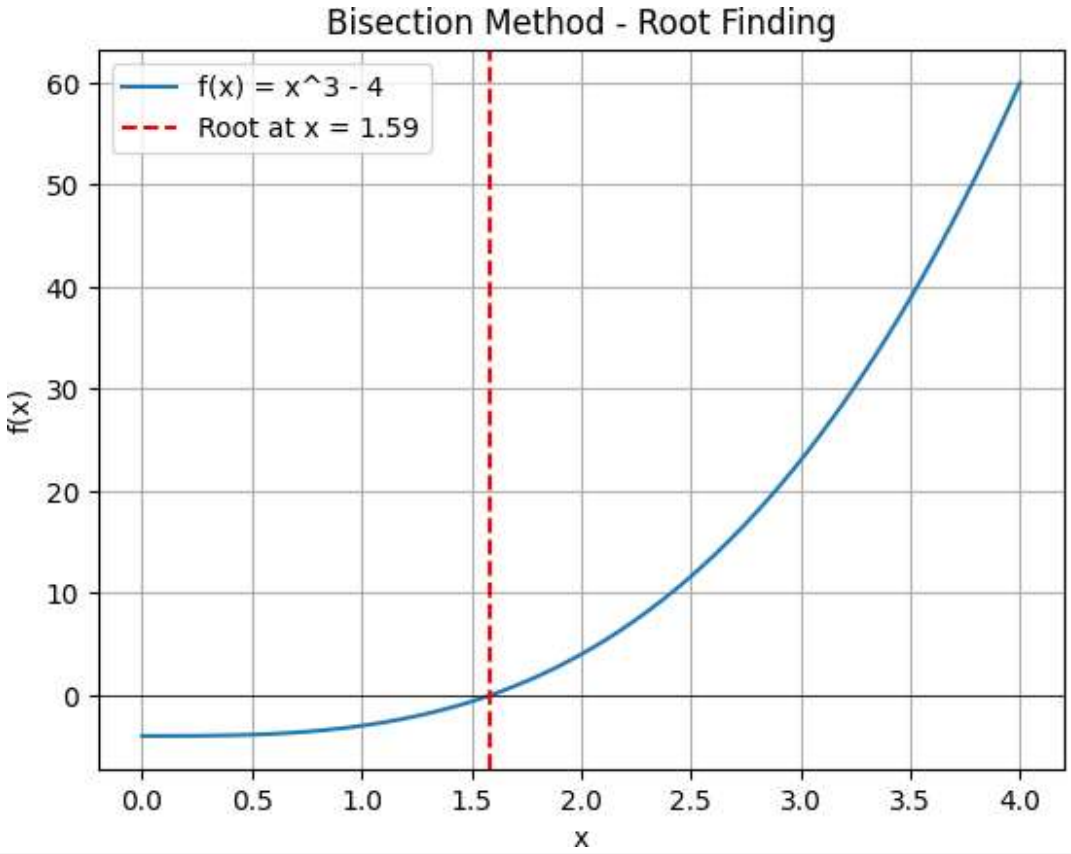
   c. Otherwise, if f(a) and f(m) have opposite signs, set b = m.

   If f(m) and f(b) have opposite signs, set a = m.
4. Return the root approximation m.

Source code:

```
import numpy as np
import matplotlib.pyplot as plt
from cmath import sin
%matplotlib inline

def my_bisection(f, a, b, tol):
    """
    Approximates a root, R, of the function f bounded by a and b to
within tolerance tol.

    Arguments:
    f -- Function whose root we are trying to find
    a -- Left boundary of the interval
    b -- Right boundary of the interval
    tol -- Tolerance value for stopping the iteration

    Returns:
    m -- Approximate root
    """
    # Check if a and b bound a root
    if np.sign(f(a)) == np.sign(f(b)):
        raise Exception("The scalars a and b do not bound a root")

    # Get midpoint
    m = (a + b) / 2

    # Stopping condition, report m as root if f(m) is within tolerance
    if np.abs(f(m)) < tol:
        return m
    elif np.sign(f(a)) == np.sign(f(m)):
        # Case where m is an improvement on a. Make recursive call
with a = m
        return my_bisection(f, m, b, tol)
    elif np.sign(f(b)) == np.sign(f(m)):
        # Case where m is an improvement on b. Make recursive call
with b = m
        return my_bisection(f, a, m, tol)


def f(x):
    return x**3 - 4

# Bisection method
root = my_bisection(f, 1, 3, 0.01)
print("Root:", root)

# Plotting the function and the root
x_vals = np.linspace(0, 4, 400)
y_vals = f(x_vals)
```

```
plt.plot(x_vals, y_vals, label="f(x) = x^3 - 4")
plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(root, color='r', linestyle='--', label=f'Root at x =
{root:.2f}')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Bisection Method - Root Finding')
plt.legend()
plt.grid(True)
plt.show()

Root: 1.587890625
```




```
def func(x):
    return x**3 - 5 * x - 9

def bisection_method(func, a, b, tolerance):
    iteration = 1
    while True:
        midpoint = (a + b) / 2
        f_mid = func(midpoint)
        print(f"Iteration-{iteration}, x = {midpoint:.6f}, f(x) = {f_mid:.6f}")

        # Update the bracket
        if func(a) * f_mid < 0:
            b = midpoint
```

```
        else:
            a = midpoint

        # Check if the root is found within tolerance
        if abs(f_mid) <= tolerance:
            print(f"Required Root is: {midpoint:.8f}")
            return midpoint

        iteration += 1

def main():
    print("BISECTION METHOD IMPLEMENTATION")
    print()

    # Default values
    function_str = "x**3 - 5 * x - 9"
    a, b, tolerance = 2, 3, 0.0001

    print(f"f(x) = {function_str}")
    print(f"a = {a}, b = {b}, tolerance = {tolerance}")
    print()

    # Check if the guesses bracket the root
    if func(a) * func(b) >= 0:
        print("Error: The guesses do not bracket a root. Try again with
different values.")
        return

    # Perform the bisection method
    bisection_method(func, a, b, tolerance)

if __name__ == "__main__":
    main()
```

Output:

BISECTION METHOD IMPLEMENTATION

f(x) = x**3 - 5 * x - 9
a = 2, b = 3, tolerance = 0.0001

Iteration-1, x = 2.500000, f(x) = -5.875000
Iteration-2, x = 2.750000, f(x) = -1.953125
Iteration-3, x = 2.875000, f(x) = 0.388672
Iteration-4, x = 2.812500, f(x) = -0.815186
Iteration-5, x = 2.843750, f(x) = -0.221588
Iteration-6, x = 2.859375, f(x) = 0.081448
Iteration-7, x = 2.851562, f(x) = -0.070592
Iteration-8, x = 2.855469, f(x) = 0.005297

Iteration-9, x = 2.853516, f(x) = -0.032680
Iteration-10, x = 2.854492, f(x) = -0.013700
Iteration-11, x = 2.854980, f(x) = -0.004204
Iteration-12, x = 2.855225, f(x) = 0.000546
Iteration-13, x = 2.855103, f(x) = -0.001829
Iteration-14, x = 2.855164, f(x) = -0.000641
Iteration-15, x = 2.855194, f(x) = -0.000048
Required Root is: 2.85519409

Test cases

Test for f(x) = x^2 - 4 in the interval [1, 3] with tolerance 0.01.

Test for f(x) = cos(x) - x in the interval [0, 1] with tolerance 0.001

System of Non-Linear Equations using Newton- Raphson Method

Working Principle: The Newton-Raphson method can be extended to solve systems of non-linear equations. This is done by solving the Jacobian matrix of the system, which contains the first derivatives of each equation. The system is solved iteratively using a vector form of the Newton-Raphson method, where at each iteration, a correction vector is computed to update the solution.

Pseudo code:

1. Given initial guesses x0 for the system, tolerance ε, and the Jacobian matrix J(x).
   a. Repeat until the difference between successive approximations is less than ε: Compute the function vector F(x) and Jacobian matrix J(x).
   b. Solve the linear system J(x) * delta = -F(x) for delta.

c. Update the guess: x1 = x0 + delta.

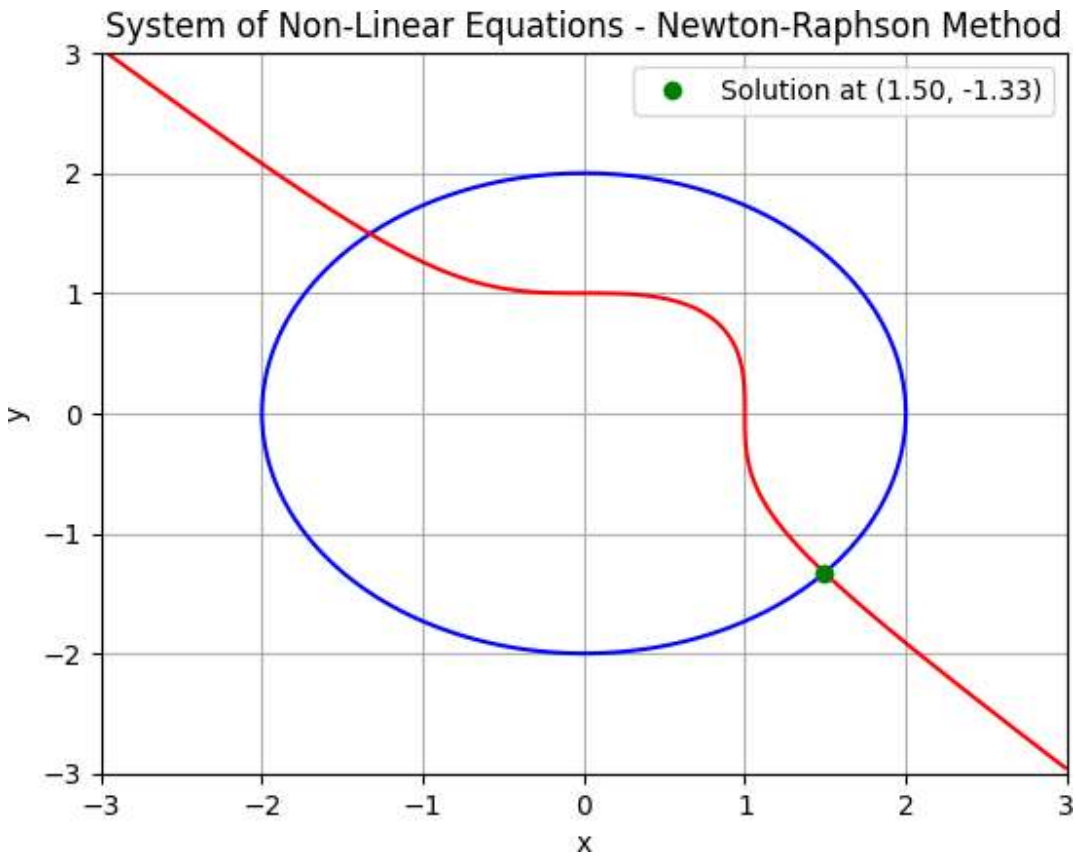
2. Return the solution vector x1.

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

# Example system of non-linear equations
def f1(x):
    return x[0]**2 + x[1]**2 - 4

def f2(x):
    return x[0]**3 + x[1]**3 - 1

# Function vector
def f(x):
    return np.array([f1(x), f2(x)])

# Jacobian matrix with regularization to avoid singularity
def jacobian(f, x, alpha=1e-6):
    n = len(x)
    J = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            h = 1e-5
            x_perturbed = np.copy(x)
            x_perturbed[j] += h
            J[i, j] = (f(x_perturbed)[i] - f(x)[i]) / h
    J += alpha * np.eye(n)  # Add a small value to diagonal to
regularize
    return J

# Newton's method for systems
def newton_system(f, x0, tol=1e-5, max_iter=100):
    iter_count = 0
    while np.linalg.norm(f(x0)) > tol and iter_count < max_iter:
        J = jacobian(f, x0)  # Compute Jacobian
        try:
            delta = np.linalg.solve(J, -f(x0))  # Solve for the update
        except np.linalg.LinAlgError:
            print("Jacobian is singular, cannot solve system.")
            return None
        x0 = x0 + delta  # Update the solution
        iter_count += 1
    if iter_count == max_iter:
        print("Maximum iterations reached without convergence.")
```

```
        return None
    return x0

# Initial guess
x0 = np.array([1.0, 1.0])

# Solve the system
root = newton_system(f, x0)

if root is not None:
    print(f"Solution found: x = {root}")

    # Plotting the root in 2D space
    x_vals = np.linspace(-3, 3, 400)
    y_vals = np.linspace(-3, 3, 400)
    X, Y = np.meshgrid(x_vals, y_vals)
    Z1 = X**2 + Y**2 - 4
    Z2 = X**3 + Y**3 - 1

    # Plotting the system
    plt.contour(X, Y, Z1, levels=[0], colors='b', label='f1(x, y) =
0')
    plt.contour(X, Y, Z2, levels=[0], colors='r', label='f2(x, y) =
0')
    plt.plot(root[0], root[1], 'go', label=f'Solution at
({root[0]:.2f}, {root[1]:.2f})')

    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('System of Non-Linear Equations - Newton-Raphson
Method')
    plt.legend()
    plt.grid(True)
    plt.show()
else:
    print("No solution found.")
```

```
Solution found: x = [ 1.4952723  -1.32821713]
```

```
<ipython-input-34-7a3a3370c2f8>:62: UserWarning: The following kwargs
were not used by contour: 'label'
  plt.contour(X, Y, Z1, levels=[0], colors='b', label='f1(x, y) = 0')
<ipython-input-34-7a3a3370c2f8>:63: UserWarning: The following kwargs
were not used by contour: 'label'
  plt.contour(X, Y, Z2, levels=[0], colors='r', label='f2(x, y) = 0')
```


System of Non-Linear Equations - Newton-Raphson Method
Solution at (1.50, -1.33)
3
2
1
0
−1
−2
−3
y
−3
−2
−1
0
1
2
3
x