Министерство образования и науки Российской Федерации федеральное государственное автономное образовательное учреждение высшего образования
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет «Программной инженерии и компьютерной техники.»

Вычислительная математика

Лабораторная работа №2
“Численное решение нелинейных уравнений и систем”
Вариант №3

Выполнил
Григорьев Давид Владимирович
Группа: P3215
Проверила
Малышева Татьяна Алексеевна

Санкт-Петербург 2025

Содержание

1 Цель работы

Изучить численные методы решения нелинейных уравнений и их систем. Найти корни заданного нелинейного уравнения и системы уравнений. Выполнить программную реализацию методов.

2 Порядок выполнения работы

1. Отделение корней графически
2. Определение интервалов изоляции корней
3. Уточнение корней с заданной точностью
4. Решение системы нелинейных уравнений
5. Программная реализация методов

3 Рабочие формулы методов

3.1 Метод половинного деления

$$x_{k+1} = \frac{a_k + b_k}{2}, \quad \text{если } f(a_k)f(b_k) < 0$$

3.2 Метод Ньютона

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

3.3 Метод простой итерации

$$x_{k+1} = \varphi(x_k), \quad \text{где } |\varphi'(x)| < 1$$

4 График функции

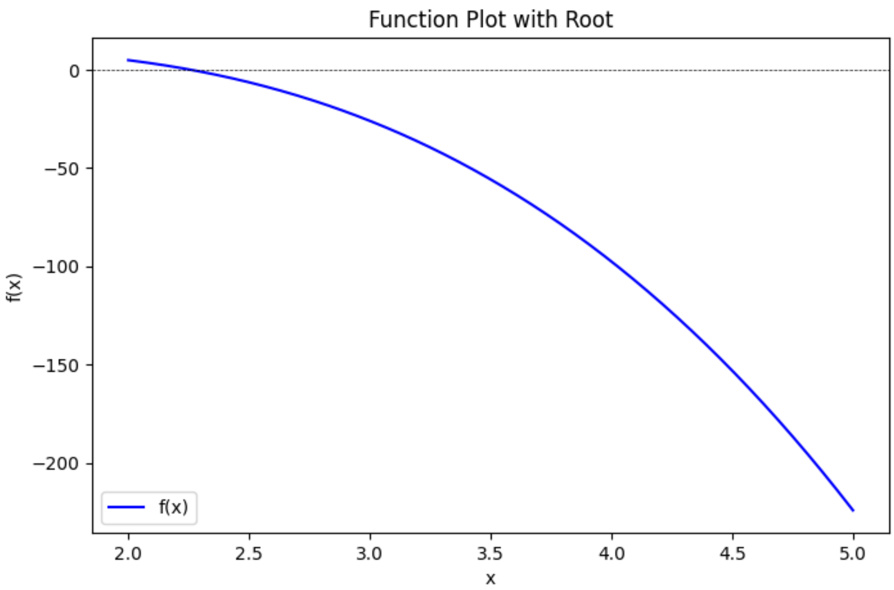

Рис. 1: График функции $f(x)$

5 Уточнение корней уравнения

5.1 Крайний правый корень (метод половинного деления)

Iteration	a	b	c	$f(c)$
1	2.000000	5.000000	3.500000	-55.772500
2	2.000000	3.500000	2.750000	-15.008125
3	2.000000	2.750000	2.375000	-2.822969
4	2.000000	2.375000	2.187500	1.552578
5	2.187500	2.375000	2.281250	-0.501997
6	2.187500	2.281250	2.234375	0.557849
7	2.234375	2.281250	2.257812	0.036158
8	2.257812	2.281250	2.269531	-0.230850
9	2.257812	2.269531	2.263672	-0.096830
10	2.257812	2.263672	2.260742	-0.030207
11	2.257812	2.260742	2.259277	0.003008

Таблица 1: Метод половинного деления

5.2 Крайний левый корень (метод простой итерации)

Iteration	x_n	x_{n+1}	$f(x_{n+1})$	Error
1	3.500000	-0.131022	1.241855	3.63e+00
2	-0.131022	-0.050172	1.894060	8.08e-02
3	-0.050172	0.073139	2.927297	1.23e-01
4	0.073139	0.263718	4.558991	1.91e-01
5	0.263718	0.560528	6.972363	2.97e-01
6	0.560528	1.014457	9.592246	4.54e-01
7	1.014457	1.638953	8.857079	6.24e-01
8	1.638953	2.215585	0.964319	5.77e-01
9	2.215585	2.278366	-0.434893	6.28e-02
10	2.278366	2.250053	0.210687	2.83e-02
11	2.250053	2.263770	-0.099059	1.37e-02
12	2.263770	2.257321	0.047277	6.45e-03
13	2.257321	2.260398	-0.022408	3.08e-03
14	2.260398	2.258940	0.010656	1.46e-03
15	2.258940	2.259633	-0.005059	6.94e-04

Таблица 2: Метод простой итерации

5.3 Центральный корень (метод Ньютона)

Iteration	x_n	$f(x_n)$	$f'(x_n)$	x_{n+1}
1	3.500000	-55.772500	-70.950000	2.713918
2	2.713918	-13.621160	-37.777184	2.353352
3	2.353352	-2.262765	-25.538014	2.264749
4	2.264749	-0.121382	-22.816962	2.259429

Таблица 3: Метод Ньютона

6 Решение системы нелинейных уравнений

$$\begin{cases} f_1(x, y) = 0 \\ f_2(x, y) = 0 \end{cases}$$

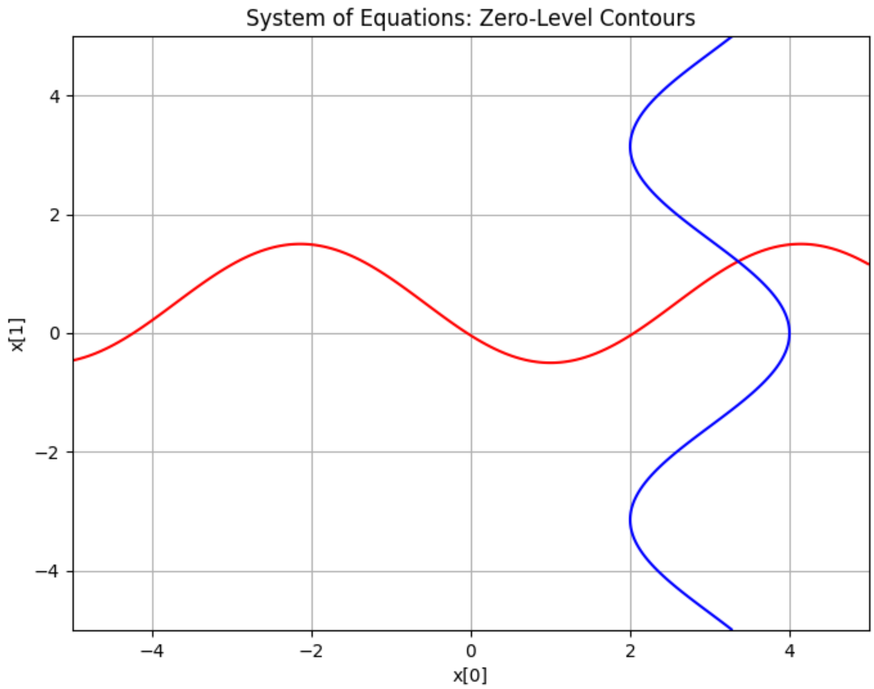

Рис. 2: Графическое решение системы

6.1 Метод простой итерации для системы

Проверка условия сходимости:

$$|\partial\varphi_1/\partial x| + |\partial\varphi_1/\partial y| < 1, \quad |\partial\varphi_2/\partial x| + |\partial\varphi_2/\partial y| < 1$$

7 Листинг программы

7.1 main.py

```
import math
from latex import (
    generate_bisection_latex_table,
    generate_newton_latex_table,
    generate_simple_iter_latex_table,
)
from plotter import plot_bisection, plot_graph, plot_newton, plot_simple_iteration
from zero_finder import ZeroFinder


# Predefined equations with f, df, and d2f
equations = [
    {
        "id": 1,
        "name": "-2.4x³ + 1.27x² + 8.36x + 2.31",
        "f": lambda x: -2.4 * x**3 + 1.27 * x**2 + 8.36 * x + 2.31,
        "df": lambda x: -7.2 * x**2 + 2.54 * x + 8.36,
    },
    {
        "id": 2,
        "name": "5.74x³ - 2.95x² - 10.28x - 3.23",
        "f": lambda x: 5.74 * x**3 - 2.95 * x**2 - 10.28 * x - 3.23,
        "df": lambda x: 17.22 * x**2 - 5.9 * x - 10.28,
    },
    {
        "id": 3,
        "name": "x³ + 2.64x² - 5.41x - 11.76",
        "f": lambda x: x**3 + 2.64 * x**2 - 5.41 * x - 11.76,
        "df": lambda x: 3 * x**2 + 5.28 * x - 5.41,
    },
    {
        "id": 4,
        "name": "sin(x) - e^(-x)",
        "f": lambda x: math.sin(x) - math.exp(-x),
        "df": lambda x: math.cos(x) + math.exp(-x),
    },
    {
        "id": 5,
        "name": "x³ + 2.84x² - 5.606x - 14.766",
        "f": lambda x: x**3 + 2.84 * x**2 - 5.606 * x - 14.766,
        "df": lambda x: 3 * x**2 + 5.68 * x - 5.606,
    },
]


# Function to let the user select a function from the list
def select_function():
    print("\nAvailable Functions:")
    for eq in equations:
        print(f"{eq['id']}. {eq['name']}")
    while True:
```

```python
        try:
            choice = int(input("Select function by ID (1-5): "))
            for eq in equations:
                if eq["id"] == choice:
                    print(f"Selected function: {eq['name']}")
                    return eq["f"], eq["df"], eq["name"]
            print("Invalid ID. Please enter a number between 1 and 5.")
        except ValueError:
            print("Invalid input. Please enter a number.")


def get_interval():
    while True:
        try:
            a = float(input("Enter left endpoint of interval (a): "))
            b = float(input("Enter right endpoint of interval (b): "))
            if a >= b:
                print("Error: a must be less than b")
                continue
            return (a, b)
        except ValueError:
            print("Please enter valid numbers")


def get_epsilon():
    while True:
        try:
            eps = float(input("Enter tolerance (epsilon): "))
            if eps <= 0:
                print("Error: epsilon must be positive")
                continue
            return eps
        except ValueError:
            print("Please enter a valid number")


if __name__ == "__main__":
    f, df, name = select_function()
    interval = get_interval()
    epsilon = get_epsilon()
    zero_finder = ZeroFinder(f, df, interval, "output/")
    plot_graph(zero_finder, "output/graph_plot.png")

    try:
        print("\nRunning Bisection method:")
        root = zero_finder.bisection_method(tolerance=epsilon, debug=True)
        print(f"Bisection root: {root:.6f}")
        print(f"Bisection value: {f(root)}")
        print(f"Bisection iterations: {len(zero_finder.bisection_data)}")

        latex_str = generate_bisection_latex_table(zero_finder)
        with open("output/bisection.tex", "w") as file:
            file.write(latex_str)
        plot_bisection(zero_finder)
```

```
    except ValueError as e:
        print(f"Bisection error: {e}")
    except OverflowError as e:
        print(f"Bisection error: {e}")

    try:
        print("\nRunning Newton method:")
        root = zero_finder.newton_method(tolerance=epsilon, debug=True)
        print(f"Newton root: {root:.6f}")
        print(f"Newton value: {f(root)}")
        print(f"Newton iterations: {len(zero_finder.newton_data)}")

        latex_str = generate_newton_latex_table(zero_finder)
        with open("output/newton.tex", "w") as file:
            file.write(latex_str)
        plot_newton(zero_finder)

    except ValueError as e:
        print(f"Newton error: {e}")
    except OverflowError as e:
        print(f"Newton error: {e}")

    try:
        print("\nRunning Iterative method:")
        root = zero_finder.simple_iteration_method(tolerance=epsilon, debug=True)
        print(f"Iterative root: {root:.6f}")
        print(f"Iterative value: {f(root)}")
        print(f"Iterative iterations: {len(zero_finder.simple_iter_data)}")

        latex_str = generate_simple_iter_latex_table(zero_finder)
        with open("output/simple_iteration.tex", "w") as file:
            file.write(latex_str)
        plot_simple_iteration(zero_finder)

    except ValueError as e:
        print(f"Iterative error: {e}")
    except OverflowError as e:
        print(f"Iterative error: {e}")
```

7.2 latex.py

```
from zero_finder import ZeroFinder


def generate_bisection_latex_table(zero_finder: ZeroFinder):
    if not zero_finder.bisection_data:
        return ""

    headers = ["Iteration", "$a$", "$b$", "$c$", "$f(c)$"]
    latex = []
    latex.append(r"\begin{tabular}{|c|c|c|c|c|}")
    latex.append(r"\hline")
    latex.append(" & ".join(headers) + r" \\")
    latex.append(r"\hline")
```

```python
    for i, entry in enumerate(zero_finder.bisection_data):
        row = [
            str(i + 1),
            f"{entry['left']:.6f}",
            f"{entry['right']:.6f}",
            f"{entry['mid']:.6f}",
            f"{entry['f_mid']:.6f}",
        ]
        latex.append(" & ".join(row) + r" \\")
        latex.append(r"\hline")

    latex.append(r"\end{tabular}")
    return "\n".join(latex)


def generate_newton_latex_table(zero_finder: ZeroFinder):
    if not zero_finder.newton_data:
        return ""

    headers = ["Iteration", "$x_n$", "$f(x_n)$", "$f'(x_n)$", "$x_{n+1}$"]
    latex = []
    latex.append(r"\begin{tabular}{|c|c|c|c|c|}")
    latex.append(r"\hline")
    latex.append(" & ".join(headers) + r" \\")
    latex.append(r"\hline")

    for i, entry in enumerate(zero_finder.newton_data):
        row = [
            str(i + 1),
            f"{entry['x']:.6f}",
            f"{entry['fx']:.6f}",
            f"{entry['dfx']:.6f}",
            f"{entry['x_new']:.6f}",
        ]
        latex.append(" & ".join(row) + r" \\")
        latex.append(r"\hline")

    latex.append(r"\end{tabular}")
    return "\n".join(latex)


def generate_simple_iter_latex_table(zero_finder: ZeroFinder):
    if not zero_finder.simple_iter_data:
        return ""

    headers = ["Iteration", "$x_n$", "$x_{n+1}$", "$f(x_{n+1})$", "Error"]
    latex = [
        r"\begin{tabular}{|c|c|c|c|c|}",
        r"\hline",
        " & ".join(headers) + r" \\",
        r"\hline",
    ]

    for entry in zero_finder.simple_iter_data:
        row = [
```

```
            str(entry["iteration"]),
            f"{entry['x_prev']:.6f}",
            f"{entry['x_next']:.6f}",
            f"{entry['f_x_next']:.6f}",
            f"{entry['error']:.2e}",
        ]
        latex.append(" & ".join(row) + r" \\")
        latex.append(r"\hline")

    latex.append(r"\end{tabular}")
    return "\n".join(latex)


def generate_newton_system_latex_table(solver):
    """
    Generate a LaTeX table for Newton's method iterations.

    Parameters:
    - solver: Instance of SystemSolver after running Newton's method

    Returns:
    - LaTeX string representation of the table
    """
    iterations = solver.iterations
    n_vars = len(solver.initial_guess)

    # Start LaTeX table
    latex = (
        "\\begin{table}[H]\n\\centering\n\\begin{tabular}{|c|"
        + "c|" * n_vars
        + "c|c|}\n\\hline\n"
    )
    latex += "Iteration"

    for var_idx in range(n_vars):
        latex += f" & x_{var_idx + 1}"

    latex += " & $\\|\\Delta x\\|$ & $\\|F(x)\\|$ \\\\ \\hline\n"

    # Add rows
    for iter_data in iterations:
        latex += f"{iter_data['iteration']} & "
        x_values = " & ".join(f"{xi:.6f}" for xi in iter_data["x"])
        latex += f"{x_values} & {iter_data['delta_norm']:.3e} & {iter_data['f_norm']:.3e

    latex += "\\end{tabular}\n\\caption{Newton's Method Iterations for System of Equatio
    return latex
```

7.3 plotter.py

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

from system_solver import SystemSolver
from zero_finder import ZeroFinder
```

```python
def plot_bisection(zero_finder: ZeroFinder):
    if not zero_finder.bisection_data:
        print("Run iterative_method with debug=True first")
        return

    left, right = zero_finder.a, zero_finder.b
    x = np.linspace(left, right, 1000)
    y = [zero_finder.func(xi) for xi in x]

    plt.figure(figsize=(12, 7))
    plt.plot(x, y, label="Function", color="navy")
    plt.axhline(0, color="black", linestyle="--", alpha=0.5)

    # Plot intervals and midpoints
    colors = plt.cm.viridis(np.linspace(0, 1, len(zero_finder.bisection_data)))
    for i, (data, color) in enumerate(zip(zero_finder.bisection_data, colors)):
        plt.axvspan(data["left"], data["right"], alpha=0.1, color=color)
        plt.scatter(
            data["mid"],
            0,
            color=color,
            s=50,
            zorder=2,
            label=f"Iter {i+1}" if i < 3 else None,
        )

    # Final result
    final_x = zero_finder.bisection_data[-1]["mid"]
    plt.scatter(
        final_x, 0, color="red", marker="*", s=200, zorder=3, label="Final Result"
    )

    plt.title("Bisection Method Visualization")
    plt.xlabel("x")
    plt.ylabel("f(x)")
    plt.legend(bbox_to_anchor=(1.05, 1), loc="upper left")
    plt.grid(True)

    if zero_finder.plot_path:
        plt.savefig(zero_finder.plot_path + "bisection.pdf", bbox_inches="tight")
        plt.savefig(zero_finder.plot_path + "bisection.png", bbox_inches="tight")
    plt.close()


def plot_newton(zero_finder: ZeroFinder):
    if not zero_finder.newton_data:
        print("Run newton_method with debug=True first")
        return

    plt.figure(figsize=(12, 7))
    x_vals = np.linspace(zero_finder.a, zero_finder.b, 1000)
    f_vals = [zero_finder.func(x) for x in x_vals]
```

```python
    # Create single plot
    fig, ax = plt.subplots(figsize=(12, 7))

    # Plot function and iterations
    ax.plot(x_vals, f_vals, label="f(x)", color="blue")
    ax.axhline(0, color="black", linestyle="--", alpha=0.5)

    colors = plt.cm.plasma(np.linspace(0, 1, len(zero_finder.newton_data)))
    for i, (data, color) in enumerate(zip(zero_finder.newton_data, colors)):
        # Function plot annotations
        ax.scatter(data["x"], data["fx"], color=color, s=80, zorder=3)
        ax.plot(
            [data["x"], data["x_new"]],
            [data["fx"], 0],
            linestyle="--",
            color=color,
            alpha=0.7,
            label=f"Iter {i+1}" if i == 0 else "",
        )

    # Final result marker
    final_x = zero_finder.newton_data[-1]["x_new"]
    ax.scatter(final_x, 0, color="red", marker="*", s=200, zorder=4, label="Root")

    ax.set_title("Newton-Raphson Method Convergence")
    ax.legend()
    ax.grid(True)

    plt.tight_layout()
    if zero_finder.plot_path:
        plt.savefig(zero_finder.plot_path + "newton.pdf", bbox_inches="tight")
        plt.savefig(zero_finder.plot_path + "newton.png", bbox_inches="tight")
    plt.close()


def plot_simple_iteration(zero_finder: ZeroFinder):
    """Visualize the simple iteration method convergence steps"""
    if not zero_finder.simple_iter_data:
        print("Run simple_iteration_method with debug=True first")
        return

    plt.figure(figsize=(12, 7))
    x_vals = np.linspace(zero_finder.a, zero_finder.b, 1000)
    f_vals = [zero_finder.func(x) for x in x_vals]

    # Main function plot
    plt.plot(x_vals, f_vals, label="f(x)", color="blue")
    plt.axhline(0, color="black", linestyle="--", alpha=0.5, linewidth=1)

    # Iteration visualization
    colors = plt.cm.plasma(np.linspace(0, 1, len(zero_finder.simple_iter_data)))

    for i, (entry, color) in enumerate(zip(zero_finder.simple_iter_data, colors)):
        x_prev = entry["x_prev"]
        x_next = entry["x_next"]
```

```python
        f_x_prev = zero_finder.func(x_prev)

        # Plot iteration step components
        plt.plot([x_prev, x_prev], [f_x_prev, 0], color=color, linestyle=":", alpha=0.5)
        plt.plot([x_prev, x_next], [0, 0], color=color, linestyle="-", alpha=0.7)
        plt.scatter(
            x_prev,
            f_x_prev,
            color=color,
            s=80,
            zorder=3,
            label=f"Iter {i+1}" if i == 0 else "",
        )
        plt.scatter(x_next, 0, color=color, marker="X", s=100, zorder=3)

    # Final root marker
    final_x = zero_finder.simple_iter_data[-1]["x_next"]
    plt.scatter(final_x, 0, color="red", marker="*", s=200, zorder=4, label="Root")

    plt.title("Simple Iteration Method Convergence")
    plt.xlabel("x")
    plt.ylabel("f(x)")
    plt.legend()
    plt.grid(True)

    # Save plots if path specified
    if zero_finder.plot_path:
        plt.savefig(f"{zero_finder.plot_path}simple_iteration.pdf", bbox_inches="tight")
        plt.savefig(f"{zero_finder.plot_path}simple_iteration.png", bbox_inches="tight")
    plt.close()


def plot_newton_system(solver):
    """
    Plot the convergence behavior of Newton's method for systems.

    Parameters:
    - solver: Instance of SystemSolver after running Newton's method
    """
    iterations = solver.iterations
    delta_norms = [iter_data["delta_norm"] for iter_data in iterations]
    f_norms = [iter_data["f_norm"] for iter_data in iterations]
    iters = list(range(1, len(iterations) + 1))

    plt.figure(figsize=(10, 5))
    plt.semilogy(iters, delta_norms, label="||x||", marker="o")
    plt.semilogy(iters, f_norms, label="||F(x)||", marker="s")
    plt.xlabel("Iteration")
    plt.ylabel("Norm")
    plt.title("Convergence of Newton's Method for System of Equations")
    plt.legend()
    plt.grid(True)
    if solver.output_dir:
        plt.savefig(f"{solver.output_dir}newton_system_convergence.png")
        plt.savefig(f"{solver.output_dir}newton_system_convergence.pdf")
```

```python
    plt.close()


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


def plot_system(output_dir, F):
    # Generate a grid of x and y values
    x_vals = np.linspace(-5, 5, 400)
    y_vals = np.linspace(-5, 5, 400)
    X, Y = np.meshgrid(x_vals, y_vals)

    # Evaluate the system of equations on the grid
    Z = F([X, Y])  # F should be vectorized
    Z1, Z2 = Z[0], Z[1]  # Extract the two equations

    # Create the plot
    plt.figure(figsize=(8, 6))
    contour1 = plt.contour(X, Y, Z1, levels=[0], colors="red")
    contour2 = plt.contour(X, Y, Z2, levels=[0], colors="blue")

    # Add titles and labels
    plt.xlabel("x[0]")
    plt.ylabel("x[1]")
    plt.title("System of Equations: Zero-Level Contours")
    plt.grid(True)

    # Ensure the output directory exists
    os.makedirs(output_dir, exist_ok=True)
    plot_path = os.path.join(output_dir, "system_plot.png")

    # Save and close the plot
    plt.savefig(plot_path)
    plt.close()


def plot_graph(zero_finder: ZeroFinder, output_path: str):
    """
    Plot the function over the interval [a, b], and optionally mark the root.
    If a plot path is provided, the plot is saved to that location.
    """
    # Generate x values from a to b
    x = np.linspace(zero_finder.a, zero_finder.b, 400)
    y = [zero_finder.func(xi) for xi in x]

    # Create the plot
    plt.figure(figsize=(8, 5))
    plt.plot(x, y, label="f(x)", color="blue")
    plt.axhline(0, color="black", linestyle="--", linewidth=0.5)

    plt.legend()
    plt.title("Function Plot with Root")
```

```
    plt.xlabel("x")
    plt.ylabel("f(x)")

    # Save the plot if a path is provided
    if output_path:
        # Ensure the directory exists
        os.makedirs(os.path.dirname(output_path), exist_ok=True)
        plt.savefig(output_path)
        print(f"Plot saved to {output_path}")
    plt.close()
```

7.4 system_main.py

```
from latex import generate_newton_system_latex_table
from plotter import plot_newton_system, plot_system
from system_solver import SystemSolver
import numpy as np

systems = [
    {
        "id": 1,
        "name": "sin(x+1) - y = 1.2; 2x + cos(y) = 2",
        "F": lambda x: np.array(
            [np.sin(x[0] + 1) - x[1] - 1.2, 2 * x[0] + np.cos(x[1]) - 2]
        ),
        "J": lambda x: np.array([[np.cos(x[0] + 1), -1], [2, -np.sin(x[1])]]),
    },
    {
        "id": 2,
        "name": "sin(x) + 2y = 2; x + cos(y-1) = 0.7",
        "F": lambda x: np.array(
            [np.sin(x[0]) + 2 * x[1] - 2, x[0] + np.cos(x[1] - 1) - 0.7]
        ),
        "J": lambda x: np.array([[np.cos(x[0]), 2], [1, -np.sin(x[1] - 1)]]),
    },
    {
        "id": 3,
        "name": "sin(x+y) = 1.5x -0.1; x² + 2y² =1",
        "F": lambda x: np.array(
            [np.sin(x[0] + x[1]) - 1.5 * x[0] + 0.1, x[0] ** 2 + 2 * x[1] ** 2 - 1]
        ),
        "J": lambda x: np.array(
            [[np.cos(x[0] + x[1]) - 1.5, np.cos(x[0] + x[1])], [2 * x[0], 4 * x[1]]]
        ),
    },
    {
        "id": 4,
        "name": "sin(x-1) + y = 0.5; x - cos(y) = 3",
        "F": lambda x: np.array(
            [np.cos(x[0] - 1) + x[1] - 0.5, x[0] - np.cos(x[1]) - 3]
        ),
        "J": lambda x: np.array([[-np.sin(x[0] - 1), 1], [1, np.sin(x[1])]]),
    },
]
```

```python
def get_initial_guess():
    while True:
        try:
            x0 = float(input("Enter initial guess for x: "))
            y0 = float(input("Enter initial guess for y: "))
            return [x0, y0]
        except ValueError:
            print("Please enter valid numbers.")


def get_epsilon():
    while True:
        try:
            eps = float(input("Enter tolerance (epsilon): "))
            if eps <= 0:
                print("Epsilon must be positive.")
                continue
            return eps
        except ValueError:
            print("Please enter a valid number.")


if __name__ == "__main__":
    # Вывод списка доступных систем
    print("Available systems:")
    for system in systems:
        print(f"{system['id']}. {system['name']}")

    # Выбор системы пользователем
    while True:
        try:
            system_id = int(input("Select system by ID: "))
            selected_system = next((s for s in systems if s["id"] == system_id), None)
            if selected_system is None:
                print("Invalid ID. Please select again.")
            else:
                break
        except ValueError:
            print("Please enter a valid integer.")

    initial_guess = get_initial_guess()
    epsilon = get_epsilon()
    max_iterations = 100  # Could also ask user for this

    # Initialize the system solver
    system_solver = SystemSolver(
        F=selected_system["F"],
        J=selected_system["J"],
        initial_guess=initial_guess,
        output_dir="output/",
    )

    try:
        print("\nRunning Newton's method for system of equations:")
```

```
        root = system_solver.newton_method(
            tolerance=epsilon, max_iterations=max_iterations, debug=True
        )
        print(f"Root found: {root}")
        print(f"Function value at root=", system_solver.F(root))
        print(f"iterations=", len(system_solver.iterations))

        # Generate LaTeX table
        latex_table = generate_newton_system_latex_table(system_solver)
        with open("output/newton_system.tex", "w") as f:
            f.write(latex_table)

        # Plot convergence
        plot_newton_system(system_solver)
        plot_system("output/", system_solver.F)

    except RuntimeError as e:
        print(f"Error during Newton's method: {e}")
```

7.5 system_solver.py

```
import numpy as np


class SystemSolver:
    def __init__(self, F, J, initial_guess, output_dir="output/"):
        """
        Initialize the system solver for Newton's method.

        Parameters:
        - F: Function that returns the vector of residuals
        - J: Function that returns the Jacobian matrix
        - initial_guess: Initial guess for the solution vector
        - output_dir: Directory to save output files
        """
        self.F = F
        self.J = J
        self.initial_guess = np.array(initial_guess, dtype=float)
        self.output_dir = output_dir
        self.iterations = []
        self.root = None
        self.converged = False

    def newton_method(self, tolerance=1e-6, max_iterations=100, debug=False):
        """
        Perform Newton-Raphson iterations to solve the system.

        Parameters:
        - tolerance: Convergence threshold
        - max_iterations: Maximum number of iterations
        - debug: Whether to record iteration data

        Returns:
        - x: Final solution vector
        """
```

```
        x = self.initial_guess.copy()
        for i in range(max_iterations):
            F_val = self.F(x)
            J_val = self.J(x)

            try:
                delta = np.linalg.solve(J_val, -F_val)
            except np.linalg.LinAlgError:
                raise RuntimeError("Jacobian is singular and cannot be inverted.")

            if debug:
                iteration_data = {
                    "iteration": i + 1,
                    "x": x.copy(),
                    "delta_norm": np.linalg.norm(delta),
                    "f_norm": np.linalg.norm(F_val),
                }
                self.iterations.append(iteration_data)

            x += delta
            print("step: ", delta)

            if np.linalg.norm(delta) < tolerance:
                if debug:
                    iteration_data = {
                        "iteration": i + 1,
                        "x": x.copy(),
                        "delta_norm": np.linalg.norm(delta),
                        "f_norm": np.linalg.norm(F_val),
                    }
                    self.iterations.append(iteration_data)

                self.root = x
                self.converged = True
                return x

        raise RuntimeError("Maximum number of iterations reached without convergence.")
```

7.6 zero_finder.py

```
class ZeroFinder:
    def __init__(self, func, derivative, interval, plot_path=""):
        self.func = func
        self.derivative = derivative
        self.a, self.b = interval
        self.plot_path = plot_path
        self.bisection_data = []
        self.newton_data = []
        self.simple_iter_data = []

        if self.a >= self.b:
            raise ValueError("Interval must be in the form [a, b] where a < b")

    def bisection_method(self, tolerance=1e-6, max_iterations=1000, debug=False):
        self.bisection_data = []
```

```
        a, b = self.a, self.b
        fa = self.func(a)
        fb = self.func(b)

        if fa * fb >= 0:
            raise ValueError("Function must have opposite signs at endpoints")

        for _ in range(max_iterations):
            c = (a + b) / 2
            fc = self.func(c)

            if debug:
                self.bisection_data.append(
                    {"left": a, "right": b, "mid": c, "f_mid": fc}
                )

            if abs(fc) < tolerance and (b - a) / 2 < tolerance:
                print("function value return:", abs(fc) < tolerance)
                print("function argument return:", (b - a) / 2 < tolerance)

                return c

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

        return (a + b) / 2

    def newton_method(
        self, initial_guess=None, tolerance=1e-6, max_iterations=1000, debug=False
    ):
        self.newton_data = []
        x = initial_guess if initial_guess else (self.a + self.b) / 2

        for _ in range(max_iterations):
            fx = self.func(x)
            dfx = self.derivative(x)
            if dfx == 0:
                raise ValueError("Zero derivative encountered")
            x_new = x - fx / dfx

            if debug:
                self.newton_data.append({"x": x, "fx": fx, "dfx": dfx, "x_new": x_new})

            if abs(x_new - x) < tolerance:
                return x_new
            x = x_new

        return x

    def simple_iteration_method(
        self, initial_guess=None, tolerance=1e-6, max_iterations=1000, debug=False
    ):
        self.simple_iter_data = []
```

```
x0 = initial_guess if initial_guess else (self.a + self.b) / 2

# Validate contraction condition
try:
    df_a = self.derivative(self.a)
    df_b = self.derivative(self.b)
    if abs(df_a) >= 1 or abs(df_b) >= 1:
        print("Derivative condition not satisfied (|phi'| < 1 required)")

    M = max(df_a, df_b)
    print("M =", M)
    if M < 0:
        lam = -1 / M
    else:
        lam = 1 / M
    print("lambda =", lam)
except ZeroDivisionError:
    raise ValueError("Cannot compute  - zero derivative at boundaries")

phi = lambda x: x + lam * self.func(x)
x_prev = x0

phi_prime = lambda x: 1 + lam * self.derivative(x)

print("phi'(a)=", phi_prime(self.a))
print("phi'(b)=", phi_prime(self.b))

for iter_count in range(max_iterations):
    x_next = phi(x_prev)
    error = abs(x_next - x_prev)

    if debug:
        self.simple_iter_data.append(
            {
                "iteration": iter_count + 1,
                "x_prev": x_prev,
                "x_next": x_next,
                "f_x_next": self.func(x_next),
                "error": error,
            }
        )

    if error < tolerance and abs(self.func(x_next)) < tolerance:
        return x_next

    x_prev = x_next

raise ValueError(f"No convergence in {max_iterations} iterations")
```

8 Вывод программы

```
Available Functions:
1. -2.4x³ + 1.27x² + 8.36x + 2.31
2. 5.74x³ - 2.95x² - 10.28x - 3.23
3. x³ + 2.64x² - 5.41x - 11.76
4. sin(x) - e^(-x)
5. x³ + 2.84x² - 5.606x - 14.766
Select function by ID (1-5): 1
Selected function: -2.4x³ + 1.27x² + 8.36x + 2.31
Enter left endpoint of interval (a): 2
Enter right endpoint of interval (b): 5
Enter tolerance (epsilon): 0.01
Plot saved to output/graph_plot.png

Running Bisection method:
function value return: True
function argument return: True
Bisection root: 2.259277
Bisection value: 0.0030075054429459236
Bisection iterations: 11

Running Newton method:
Newton root: 2.259429
Newton value: -0.0004251722422687898
Newton iterations: 4

Running Iterative method:
Derivative condition not satisfied (|phi'| < 1 required)
M = -15.36
lambda = 0.06510416666666667
phi'(a)= 0.0
phi'(b)= -9.34765625
Iterative root: 2.259633
Iterative value: -0.005059379576943801
Iterative iterations: 15
```