Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»

Факультет Программной Инженерии и Компьютерной Техники

Лабораторная работа №2

«Численное решение нелинейных уравнений и систем»

по дисциплине «Вычислительная математика»

Вариант: **9**

Преподаватель:
Машина Екатерина Алексеевна

Выполнил:
Пронкин Алексей Дмитриевич
Группа: P3208

Санкт-Петербург, 2025 г.

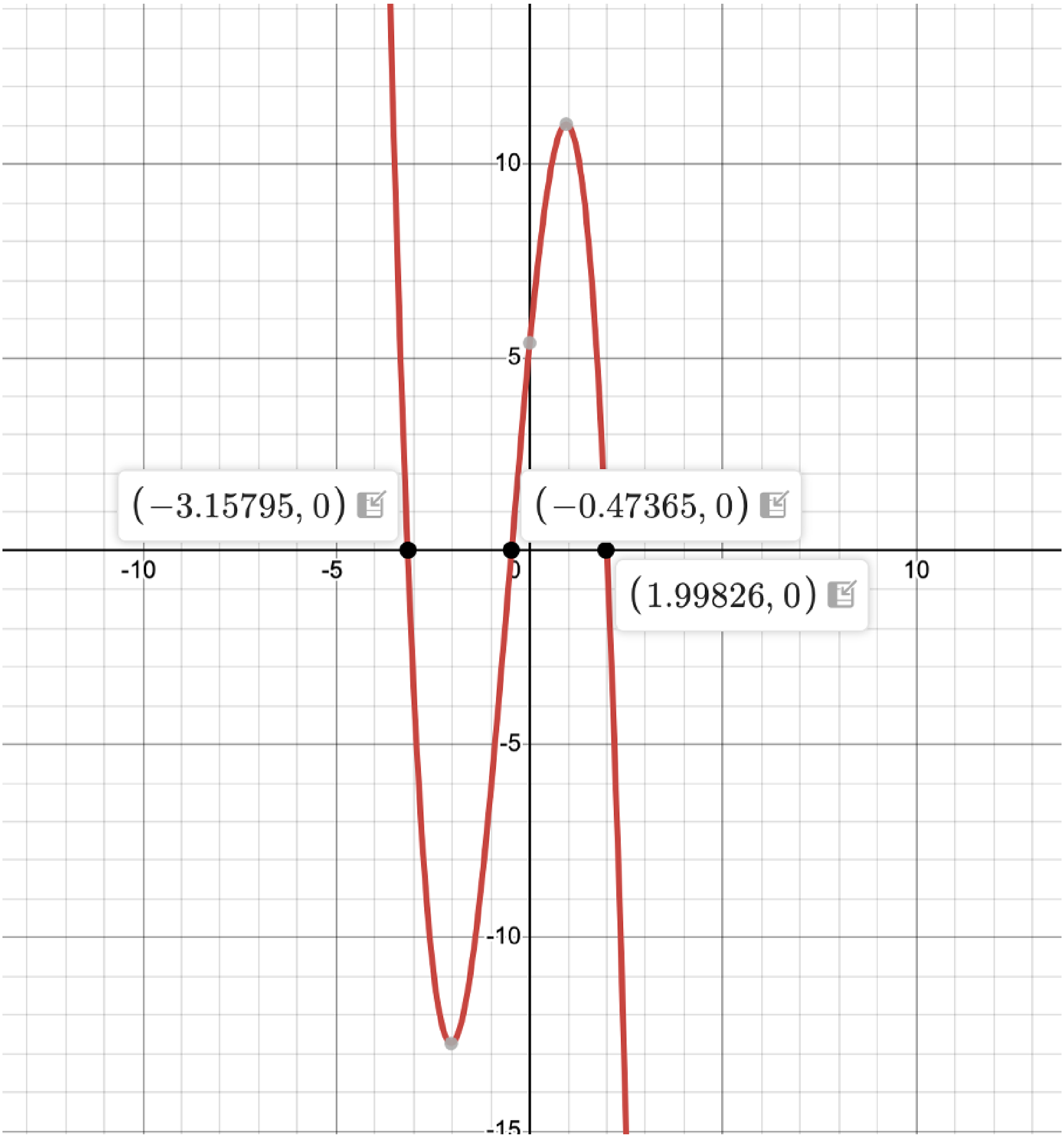
Цель работы

Изучить численные методы решения нелинейных уравнений и их систем, найти корни заданного нелинейного уравнения/системы нелинейных уравнений, выполнить программную реализацию методов.

1. 1 часть. Решение нелинейного уравнения

1.1. Отделение корней заданного нелинейного уравнения графически

$$-1.8x^3 - 2.94x^2 + 10.37x + 5.38$$



1.2. Определить интервалы изоляции корней.

Для определения интервалов изоляции корней данного уравнения можно воспользоваться методом интервалов знакопеременности. Найдём интервалы, в которых происходит смена знака, что свидетельствует о наличии корня (теорема Больцано).

Крайний левый интервал:
Исследуем значения функции при $x = -4$ и $x = -3$. При $x = -4$ функция принимает отрицательное значение, а при $x = -3$ – положительное. Это означает, что на интервале $(-4, -3)$ функция переходит через 0, и там лежит один корень.

Центральный интервал:
Рассмотрим $x = -1$ и $x = 0$. При $x = -1$ значение оказывается положительным, а при $x = 0$ отрицательным. Значит, на $(-1, 0)$ есть корень.

Крайний правый интервал:
Проверим $x = 1$ и $x = 2$. При $x = 1$ функция отрицательна, а при $x = 2$ – положительна. Следовательно, существует корень на $(1, 2)$.

Таким образом, интервалы изоляции корней данного уравнения:

1. $(-4, -3)$
2. $(-1,\ 0)$
3. $(1,\ 2)$

1.3. Уточнить корни нелинейного уравнения с точностью $\varepsilon = 10^{-2}$

1. $(-4, -3)$
2. $(-1,\ 0)$
3. $(1,\ 2)$

1.4. Методы уточнения

Крайний левый интервал – метод простой итерации

1. **Итерационная функция**

$$x_{k+1} \ = \ \varphi(x_k) \ = \ \sqrt[3]{\frac{-2.94\,x_k^2 \ + \ 10.37\,x_k \ + \ 5.38}{1.8}}.$$

2. **Начальное приближение**

$$x_0 = -3.20.$$

3. **На каждом шаге** вычислялось: - $x_{k+1} = \varphi(x_k)$, - $f(x_{k+1})$ подстановкой x_{k+1} в исходное уравнение

$$f(x) \ = \ -1.8\,x^3 \ - \ 2.94\,x^2 \ + \ 10.37\,x \ + \ 5.38,$$

- разность $|x_{k+1} - x_k|$.

4. **Окончание итераций** — при $|x_{k+1} - x_k| < 0.01$.

№ итерации	x_k	x_{k+1}	$f(x_{k+1})$	$\lvert x_{k+1} - x_k\rvert$
1	-3.20	-3.18	0.76	0.02
2	-3.18	-3.17	-0.38	0.01
3	-3.17	-3.162	0.22	0.008
4	-3.162	-3.157	-0.006	0.005

Итоговое приближение (после 4 итераций) находится в окрестности -3.16.
Сходимость:

$$\varphi(x) = \sqrt[3]{\frac{-2.94x^2 + 10.37x + 5.38}{1.8}}, \quad \varphi'(x) = \frac{1}{3} \cdot g(x)^{-2/3} \cdot g'(x)$$

$$g(x) = \frac{-2.94x^2 + 10.37x + 5.38}{1.8}, \quad g'(x) = \frac{-5.88x + 10.37}{1.8}$$

$$|\varphi'(-3.15)| \approx 0.537 < 1$$

Условие сходимости ВЫПОЛНЯЕТСЯ

Центарльный интервал – метод секущих

1. **Выбираем два начальных приближения** x_0 и x_1, которые дают разные знаки функции:

$$x_0 = -1, \quad x_1 = 0.$$

Проверяем:

$$f(-1) = -6.13, \quad f(0) = 5.38,$$

действительно $f(x_0)$ и $f(x_1)$ имеют разные знаки, значит корень лежит между -1 и 0.

2. **Формула метода секущих** для нахождения очередного приближения x_{k+1}:

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

3. **Выполняем итерации** до достижения требуемой точности $|x_{k+1} - x_k| < 0.01$.
Итерационный процесс (краткий расчёт)

1. **Итерация 1** ($k = 1$):

$$x_2 = x_1 - f(x_1) \frac{x_1 - x_0}{f(x_1) - f(x_0)} = 0 - 5.38 \frac{0 - (-1)}{5.38 - (-6.13)} \approx -0.467.$$

$f(-0.467) \approx 0.0746.$

2. **Итерация 2** ($k = 2$):

$$x_3 = x_2 - f(x_2) \frac{x_2 - x_1}{f(x_2) - f(x_1)} = -0.467 - 0.0746 \frac{-0.467 - 0}{0.0746 - 5.38} \approx -0.474.$$

$f(-0.474) \approx -0.00075.$

3. **Итерация 3** ($k = 3$):

$$x_4 = x_3 - f(x_3) \frac{x_3 - x_2}{f(x_3) - f(x_2)} = -0.474 - (-0.00075) \frac{-0.474 - (-0.467)}{-0.00075 - 0.0746} \approx -0.47393.$$

$f(-0.47393) \approx -0.00069.$

Замечаем, что уже на **итерации 2** $|x_3 - x_2| = 0.007 < 0.01$, то есть точность 0.01 достигнута. Для полноты приведена и третья итерация.

№ итерации	x_{k-1}	x_k	x_{k+1}	$f(x_{k+1})$	$\lvert x_{k+1} - x_k \rvert$
1	-1.00	0.00	-0.467	0.0746	0.467
2	0.00	-0.467	-0.474	-0.00075	0.007
3	-0.467	-0.474	-0.47393	-0.00069	0.00007

Итог: $x \approx -0.474$.

Крайний правый интервал – метод половинного деления

1. **Задаём начальный отрезок** $[a, b]$ так, чтобы $f(a)$ и $f(b)$ имели **разные знаки**. Для $(1, 2)$:

$$a = 1, \quad b = 2, \quad f(1) \approx 11.01 > 0, \quad f(2) \approx -0.04 < 0.$$

2. **Основная формула** на каждом шаге:

$$x = \frac{a+b}{2}, \quad \text{затем проверяем знак } f(x).$$

Если $f(x)$ имеет **тот же знак**, что и $f(a)$, то сдвигаем левую границу: $a \leftarrow x$.

Иначе сдвигаем правую границу: $b \leftarrow x$.

3. **Окончание** итераций, когда длина отрезка $|b-a|$ становится меньше требуемой точности (здесь 0.01).

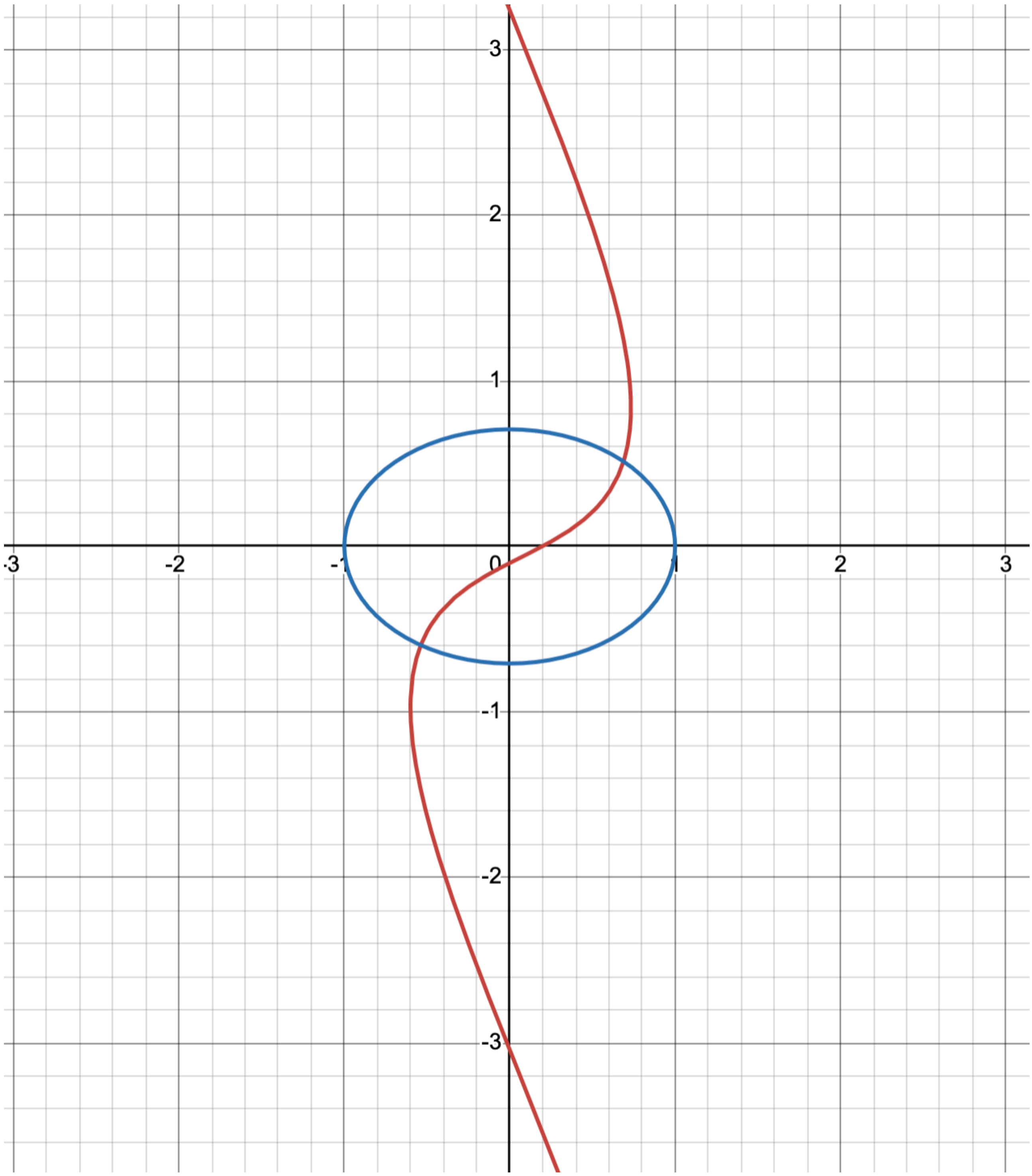
№ шага	$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{x = \frac{a+b}{2}}$	$\boldsymbol{f(a)}$	$\boldsymbol{f(b)}$	$\boldsymbol{f(x)}$	$\boldsymbol{\lvert a-b \rvert}$
1	1.0000	2.0000	1.5000	11.01	-0.04	8.245	1.0000
2	1.5000	2.0000	1.7500	8.245	-0.04	4.88	0.5000
3	1.7500	2.0000	1.8750	4.88	-0.04	2.62	0.2500
4	1.8750	2.0000	1.9375	2.62	-0.04	1.34	0.1250
5	1.9375	2.0000	1.9688	1.34	-0.04	0.65	0.0625
6	1.9688	2.0000	1.9844	0.65	-0.04	0.30	0.0313
7	1.9844	2.0000	1.9922	0.30	-0.04	0.10	0.0156
8	1.9922	2.0000	1.9961	0.14	-0.04	0.05	0.0078

Таким образом, методом половинного деления на интервале $(1, 2)$ получаем правый корень уравнения с точностью 0.01:

$$\boxed{x \approx 2.00}$$

2. 2 часть. Решение системы нелинейных уравнений

$$\begin{cases} \sin(x+y) = 1.5x - 0.1 \\ x^2 + 2y^2 = 1 \end{cases}, \qquad \text{Метод Ньютона}$$



Обозначим:

$$F_1(x, y) = \sin(x+y) - \big(1.5x - 0.1\big) = 0,$$
$$F_2(x, y) = x^2 + 2y^2 - 1 = 0.$$

Для применения метода Ньютона необходимо вычислить якобиан:

$$J(x,y) = \begin{pmatrix} \frac{\partial F_1}{\partial x} & \frac{\partial F_1}{\partial y} \\ \frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial y} \end{pmatrix} = \begin{pmatrix} \cos(x+y) - 1.5 & \cos(x+y) \\ 2x & 4y \end{pmatrix}.$$

На каждом шаге ищется поправка $\Delta = (\Delta x,\ \Delta y)^T$ из системы

$$J(x_k, y_k)\Delta = -\begin{pmatrix} F_1(x_k, y_k) \\ F_2(x_k, y_k) \end{pmatrix},$$

после чего обновляются приближения:

$$x_{k+1} = x_k + \Delta x, \quad y_{k+1} = y_k + \Delta y.$$

Анализируя уравнения и учитывая ограничения (так как $\sin(x+y)$ принимает значения от -1 до 1, а $1.5x - 0.1$ должен попадать в этот диапазон), можно ожидать наличие двух решений. При выборе разумных начальных приближений, например:

Для одного решения: $(x_0, y_0) = (0.7,\ 0.5)$,

Для второго решения: $(x_0, y_0) = (-0.55,\ -0.6)$,

последовательные итерации метода Ньютона приводят к следующим корням с точностью до 0.01:

Первое решение:

$$x \approx 0.68, \quad y \approx 0.52.$$

Проверим: - $x + y \approx 1.20 \Rightarrow \sin(1.20) \approx 0.93$, - $1.5 \cdot 0.68 - 0.1 \approx 1.02 - 0.1 = 0.92$, разница около 0.01; - $x^2 + 2y^2 \approx 0.4624 + 2 \cdot 0.2704 \approx 0.4624 + 0.5408 \approx 1.0032$.

Второе решение:

$$x \approx -0.53, \quad y \approx -0.60.$$

Проверим: - $x+y \approx -1.13 \Rightarrow \sin(-1.13) \approx -0.90$, - $1.5\cdot(-0.53) - 0.1 \approx -0.795 - 0.1 \approx -0.895$, разница порядка 0.005–0.01; - $x^2 + 2y^2 \approx 0.2809 + 2 \cdot 0.36 \approx 0.2809 + 0.72 \approx 1.0009$.

Итоговый ответ

Приближённые корни системы с точностью до 0.01:

$$\boxed{(x,y) \approx (0.68,\ 0.52) \quad \text{и} \quad (x,y) \approx (-0.53,\ -0.60).}$$

Эти результаты получены с использованием метода Ньютона.

3. Листинг программы

main.py

```python
def main():
    print("Выберите, что решать:")
    print("1 - Одно нелинейное уравнение")
    print("2 - Система нелинейных уравнений")
    choice = input("Введите номер задачи (1 или 2): ").strip()

    if choice == "1":
        try:
            import single_equations
            single_equations.main()
```

```
        except ImportError:
            print("Ошибка импорта модуля для решения уравнений.")
    elif choice == "2":
        try:
            import systems
            systems.main()
        except ImportError:
            print("Ошибка импорта модуля для решения систем уравнений.")
    else:
        print("Неверный выбор. Запустите программу снова и введите 1 или 2.")


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

single_equations.py

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


# ---------------------------
# Определение функций и их производных
# ---------------------------

# Функция 1: -1.8x^3 - 2.94x^2 + 10.37x + 5.38
def f1(x):
    return -1.8 * x ** 3 - 2.94 * x ** 2 + 10.37 * x + 5.38


def df1(x):
    return -5.4 * x ** 2 - 5.88 * x + 10.37


def f1_2(x):
    return -10.8 * x - 5.88


# Функция 2: x^3 + 2.84x^2 - 5.606x - 14.766
def f2(x):
    return x ** 3 + 2.84 * x ** 2 - 5.606 * x - 14.766


def df2(x):
    return 3 * x ** 2 + 5.68 * x - 5.606


def f2_2(x):
    return 6 * x + 5.68


# Функция 3: -1.38x^3 - 5.42x^2 + 2.57x + 10.95
```

```python
def f3(x):
    return -1.38 * x ** 3 - 5.42 * x ** 2 + 2.57 * x + 10.95


def df3(x):
    return -4.14 * x ** 2 - 10.84 * x + 2.57


def f3_2(x):
    return -8.28 * x - 10.84


# Функция 4 (трансцендентная): cos(x) - x
import math


def f4(x):
    return np.cos(x) - x


def df4(x):
    return -np.sin(x) - 1


def f4_2(x):
    return -np.cos(x)


# Словарь с функциями для выбора
functions = {
    "1": {
        "name": "f1(x) = -1.8x^3 - 2.94x^2 + 10.37x + 5.38",
        "f": f1,
        "df": df1,
        "f2": f1_2
    },
    "2": {
        "name": "f2(x) = x^3 + 2.84x^2 - 5.606x - 14.766",
        "f": f2,
        "df": df2,
        "f2": f2_2
    },
    "3": {
        "name": "f3(x) = -1.38x^3 - 5.42x^2 + 2.57x + 10.95",
        "f": f3,
        "df": df3,
        "f2": f3_2
    },
    "4": {
        "name": "f4(x) = cos(x) - x",
        "f": f4,
        "df": df4,
```

```
        "f2": f4_2
    }
}


# ---------------------------
# Классы-решатели для методов
# ---------------------------
class FixedPointIterationSolver:
    def __init__(self, f, df, a: float, b: float, eps: float):
        """
        Инициализация решателя методом простой итерации.
        Параметры:
          f    - функция, корень которой ищем
          df   - её производная
          a, b - границы интервала (при a > b они меняются местами)
          eps - требуемая точность
        """
        self.f = f
        self.df = df
        self.eps = eps
        self.a, self.b = (a, b) if a <= b else (b, a)
        self.lambda_val = None

    def count_sign_changes(self, num_points=1000) -> int:
        xs = np.linspace(self.a, self.b, num_points)
        f_vals = self.f(xs)
        signs = np.sign(f_vals)
        changes = 0
        for i in range(1, len(signs)):
            if signs[i] == 0 or signs[i - 1] == 0:
                continue
            if signs[i] != signs[i - 1]:
                changes += 1
        return changes

    def check_root_existence(self) -> None:
        sign_changes = self.count_sign_changes()
        if sign_changes != 1:
            raise ValueError(
                f"На интервале [{self.a}, {self.b}] обнаружено {sign_changes} смен
            )

    def compute_lambda(self) -> None:
        xs = np.linspace(self.a, self.b, 1000)
        df_vals = self.df(xs)
        m = np.min(df_vals)
        M = np.max(df_vals)
        if m * M < 0:
            raise ValueError("Достаточное условие сходимости не выполнено: произво
        self.lambda_val = 2 / (m + M)
        if not np.all(np.abs(1 - self.lambda_val * self.df(xs)) < 1):
```

```python
                raise ValueError("Достаточное условие сходимости не выполнено для выбра

    def choose_initial_approximation(self) -> float:
        return self.a if abs(self.f(self.a)) < abs(self.f(self.b)) else self.b

    def solve(self) -> (float, float, int):
        """
        Выполняет итерационный процесс до достижения заданной точности.
        Возвращает кортеж: (найденный корень, значение функции в корне, число итера
        """
        self.check_root_existence()
        self.compute_lambda()
        x_prev = self.choose_initial_approximation()
        iteration = 0

        while True:
            x_next = x_prev - self.lambda_val * self.f(x_prev)
            iteration += 1
            if abs(x_next - x_prev) < self.eps:
                break
            x_prev = x_next
            if iteration > 10000:
                raise RuntimeError("Превышено максимальное число итераций.")
        return x_next, self.f(x_next), iteration


class ChordMethodSolver:
    def __init__(self, f, f2, a: float, b: float, eps: float):
        """
        Инициализация решателя методом хорд с фиксированным концом.
        Параметры:
          f    - функция
          f2   - вторая производная функции f (для выбора фиксированного конца)
          a, b - границы интервала (при a > b они меняются местами)
          eps  - требуемая точность
        """
        self.f = f
        self.f2 = f2
        self.eps = eps
        self.a, self.b = (a, b) if a <= b else (b, a)
        self.fixed_endpoint = None
        self.variable_endpoint = None

    def count_sign_changes(self, num_points=1000) -> int:
        xs = np.linspace(self.a, self.b, num_points)
        f_vals = self.f(xs)
        signs = np.sign(f_vals)
        changes = 0
        for i in range(1, len(signs)):
            if signs[i] == 0 or signs[i - 1] == 0:
                continue
            if signs[i] != signs[i - 1]:
```

```python
                changes += 1
        return changes

    def check_root_existence(self) -> None:
        sign_changes = self.count_sign_changes()
        if sign_changes != 1:
            raise ValueError(
                f"На интервале [{self.a}, {self.b}] обнаружено {sign_changes} смен
            )

    def choose_fixed_endpoint(self) -> None:
        """
        Выбирает фиксированный конец по условию: f(x)*f''(x) > 0.
        Если условие выполняется для a, фиксированный конец a, переменная - b; ина
        """
        if self.f(self.a) * self.f2(self.a) > 0:
            self.fixed_endpoint = self.a
            self.variable_endpoint = self.b
        elif self.f(self.b) * self.f2(self.b) > 0:
            self.fixed_endpoint = self.b
            self.variable_endpoint = self.a
        else:
            raise ValueError(
                "Невозможно выбрать фиксированный конец, не удовлетворены условия 

    def solve(self) -> (float, float, int):
        """
        Выполняет итерационный процесс по схеме метода хорд с фиксированным концом
          x_(n+1) = x_n - (x_n - c)*f(x_n)/(f(x_n)-f(c))
        где c - выбранный фиксированный конец.
        Возвращает кортеж: (найденный корень, значение функции в корне, число итер
        """
        self.check_root_existence()
        self.choose_fixed_endpoint()
        c = self.fixed_endpoint
        x_prev = self.variable_endpoint
        iteration = 0
        while True:
            denominator = self.f(x_prev) - self.f(c)
            if denominator == 0:
                raise ZeroDivisionError("Деление на ноль в методе хорд.")
            x_next = x_prev - (x_prev - c) * self.f(x_prev) / denominator
            iteration += 1
            if abs(x_next - x_prev) < self.eps:
                break
            x_prev = x_next
            if iteration > 10000:
                raise RuntimeError("Превышено максимальное число итераций.")
        return x_next, self.f(x_next), iteration


class NewtonMethodSolver:
```

```
    def __init__(self, f, df, f2, a: float, b: float, eps: float):
        """
        Инициализация решателя методом Ньютона.
        Параметры:
          f    - функция
          df   - её производная
          f2   - вторая производная функции f (используется для выбора начального пр
          a, b - границы интервала (при a > b они меняются местами)
          eps - требуемая точность
        """
        self.f = f
        self.df = df
        self.f2 = f2
        self.eps = eps
        self.a, self.b = (a, b) if a <= b else (b, a)

    def count_sign_changes(self, num_points=1000) -> int:
        xs = np.linspace(self.a, self.b, num_points)
        f_vals = self.f(xs)
        signs = np.sign(f_vals)
        changes = 0
        for i in range(1, len(signs)):
            if signs[i] == 0 or signs[i - 1] == 0:
                continue
            if signs[i] != signs[i - 1]:
                changes += 1
        return changes

    def check_root_existence(self) -> None:
        sign_changes = self.count_sign_changes()
        if sign_changes != 1:
            raise ValueError(
                f"На интервале [{self.a}, {self.b}] обнаружено {sign_changes} смен
            )

    def choose_initial_approximation(self) -> float:
        """
        Выбирает начальное приближение для метода Ньютона.
        Рекомендуется выбрать ту границу, для которой выполнено условие f(x)*f''(x
        """
        if self.f(self.a) * self.f2(self.a) > 0:
            return self.a
        elif self.f(self.b) * self.f2(self.b) > 0:
            return self.b
        else:
            raise ValueError(
                "Невозможно выбрать начальное приближение: условие f(x)*f''(x) > 0

    def solve(self) -> (float, float, int):
        """
        Выполняет итерационный процесс метода Ньютона:
          x_(n+1) = x_n - f(x_n)/f'(x_n)
```

```
        Возвращает кортеж: (найденный корень, значение функции в корне, число итер
        """
        self.check_root_existence()
        x_prev = self.choose_initial_approximation()
        iteration = 0
        while True:
            derivative = self.df(x_prev)
            if derivative == 0:
                raise ZeroDivisionError("Производная равна нулю при x = " + str(x_
            x_next = x_prev - self.f(x_prev) / derivative
            iteration += 1
            if abs(x_next - x_prev) < self.eps:
                break
            x_prev = x_next
            if iteration > 10000:
                raise RuntimeError("Превышено максимальное число итераций.")
        return x_next, self.f(x_next), iteration


# ---------------------------
# Функция для построения графика
# ---------------------------
def plot_function(f, a: float, b: float, root: float = None):
    """
    Строит график функции f на интервале [a, b].
    Если root задан, отмечает его на графике.
    """
    xs = np.linspace(a, b, 1000)
    ys = f(xs)

    plt.figure()
    plt.plot(xs, ys, label="f(x)")
    if root is not None:
        plt.scatter([root], [f(root)], color='red', zorder=5, label="Найденный кор
    plt.xlim(a, b)
    plt.xlabel("x")
    plt.ylabel("f(x)")
    plt.title("График функции на интервале [{}, {}]".format(a, b))
    plt.grid(True)
    plt.legend()
    plt.show()


# ---------------------------
# Основная функция
# ---------------------------
def main():
    # Выбор функции
    print("Выберите функцию для поиска корня:")
    for key, func in functions.items():
        print(f"{key} - {func['name']}")
    func_choice = input("Введите номер функции (1-4): ").strip()
```

```
    if func_choice not in functions:
        print("Неверный выбор функции.")
        return
    selected = functions[func_choice]
    f_selected = selected["f"]
    df_selected = selected["df"]
    f2_selected = selected["f2"]

    # Ввод интервала и погрешности
    try:
        a = float(input("Введите левую границу интервала a: "))
        b = float(input("Введите правую границу интервала b: "))
        eps = float(input("Введите требуемую погрешность epsilon: "))
    except ValueError:
        print("Некорректный ввод.")
        return

    # Выбор метода
    print("Выберите метод для поиска корня:")
    print("1 - Метод простой итерации")
    print("2 - Метод хорд")
    print("3 - Метод Ньютона")
    method_choice = input("Введите номер метода (1, 2 или 3): ").strip()

    try:
        if method_choice == "1":
            solver = FixedPointIterationSolver(f_selected, df_selected, a, b, eps)
        elif method_choice == "2":
            solver = ChordMethodSolver(f_selected, f2_selected, a, b, eps)
        elif method_choice == "3":
            solver = NewtonMethodSolver(f_selected, df_selected, f2_selected, a, b
        else:
            print("Неверный выбор метода.")
            return

        root, f_at_root, iterations = solver.solve()
    except (ValueError, RuntimeError, ZeroDivisionError) as e:
        print(f"Ошибка: {e}")
        return

    print("\nРезультаты вычислений:")
    print(f"Выбранная функция: {selected['name']}")
    print(f"Найденный корень: {root}")
    print(f"Значение функции в корне: {f_at_root}")
    print(f"Количество итераций: {iterations}")

    # Построение графика выбранной функции на заданном интервале
    plot_function(f_selected, a, b, root)


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

systmes.py

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


# ---------------------------------------
# Класс-решатель для системы методом простых итераций
# ---------------------------------------
class SystemFixedPointSolver:
    def __init__(self, phi1, phi2, dphi1_dx, dphi1_dy, dphi2_dx, dphi2_dy, x0: flo
                 max_iter: int = 1000):
        """
        Инициализация решателя системы методом простых итераций.
        Параметры:
          phi1, phi2       - функции итерационного преобразования
          dphi1_dx, dphi1_dy, dphi2_dx, dphi2_dy - частные производные (аналитичес
          x0, y0           - начальные приближения
          tol              - требуемая погрешность
          max_iter         - максимальное число итераций
        """
        self.phi1 = phi1
        self.phi2 = phi2
        self.dphi1_dx = dphi1_dx
        self.dphi1_dy = dphi1_dy
        self.dphi2_dx = dphi2_dx
        self.dphi2_dy = dphi2_dy
        self.x0 = x0
        self.y0 = y0
        self.tol = tol
        self.max_iter = max_iter

    def check_convergence_condition(self):
        """
        Проверяет достаточное условие сходимости:
        Вычисляется норма якобиана F(x,y) = (phi1, phi2) в начальной точке,
        где норма оценивается как max{ |dphi1/dx|+|dphi1/dy|, |dphi2/dx|+|dphi2/dy
        Возвращает (True, norm) если условие выполнено (norm < 1).
        """
        x, y = self.x0, self.y0
        J1 = np.abs(self.dphi1_dx(x, y)) + np.abs(self.dphi1_dy(x, y))
        J2 = np.abs(self.dphi2_dx(x, y)) + np.abs(self.dphi2_dy(x, y))
        norm = max(J1, J2)
        return (norm < 1), norm

    def solve(self):
        """
        Выполняет итерационный процесс, пока разница между соседними приближениями
        не станет меньше tol по обеим координатам.
        Возвращает кортеж:
            ((x, y), [|Δx|, |Δy|], итераций)
        """
        x_old, y_old = self.x0, self.y0
```

```
        iterations = 0
        while iterations < self.max_iter:
            iterations += 1
            x_new = self.phi1(x_old, y_old)
            y_new = self.phi2(x_old, y_old)
            err_x = np.abs(x_new - x_old)
            err_y = np.abs(y_new - y_old)
            if max(err_x, err_y) < self.tol:
                return (x_new, y_new), [err_x, err_y], iterations
            x_old, y_old = x_new, y_new
        raise RuntimeError("Метод не сошелся за заданное число итераций.")


# ---------------------------------------
# Функция для построения графика контуров
# ---------------------------------------
def plot_system(system_funcs, solution, x_range: tuple, y_range: tuple):
    """
    Строит нулевые контуры двух функций системы и отмечает найденное решение.
    system_funcs - кортеж (F1, F2), где F1(x,y)=0 и F2(x,y)=0.
    """
    F1, F2 = system_funcs
    X, Y = np.meshgrid(np.linspace(x_range[0], x_range[1], 400),
                       np.linspace(y_range[0], y_range[1], 400))
    Z1 = F1(X, Y)
    Z2 = F2(X, Y)

    plt.figure()
    cp1 = plt.contour(X, Y, Z1, levels=[0], colors='blue', linewidths=2)
    cp2 = plt.contour(X, Y, Z2, levels=[0], colors='green', linewidths=2)
    plt.clabel(cp1, fmt='F1=0', fontsize=10)
    plt.clabel(cp2, fmt='F2=0', fontsize=10)
    plt.plot(solution[0], solution[1], 'ro', markersize=8, label='Найденное решени
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title("Нулевые контуры функций системы и найденное решение")
    plt.legend()
    plt.grid(True)
    plt.show()


# ---------------------------------------
# Основная функция
# ---------------------------------------
def main():
    print("Выберите систему нелинейных уравнений:")
    print("1. Система:")
    print("   sin(x+y) = 1.5x - 0.1")
    print("   x^2 + 2y^2 = 1")
    print()
    print("2. Система:")
    print("   tan(x*y + 0.1) = x^2")
```

```
    print("   x^2 + 2y^2 = 1")
    system_choice = input("Введите номер системы (1 или 2): ").strip()
    if system_choice not in ["1", "2"]:
        print("Неверный выбор системы.")
        return

    try:
        x0 = float(input("Введите начальное приближение x0: "))
        y0 = float(input("Введите начальное приближение y0: "))
        tol = float(input("Введите требуемую погрешность (например, 0.01): "))
    except ValueError:
        print("Некорректный ввод числовых значений.")
        return

    # Определяем знаки для итерационных преобразований
    s_x = 1 if x0 >= 0 else -1
    s_y = 1 if y0 >= 0 else -1

    # В зависимости от выбора, задаём phi-функции, их производные и функции для по
    if system_choice == "1":
        # Система 1: sin(x+y) = 1.5x - 0.1,  x^2 + 2y^2 = 1
        # Итерационные преобразования:
        phi1 = lambda x, y: (np.sin(x + y) + 0.1) / 1.5
        phi2 = lambda x, y: s_y * np.sqrt((1 - x ** 2) / 2)
        # Производные:
        dphi1_dx = lambda x, y: np.cos(x + y) / 1.5
        dphi1_dy = lambda x, y: np.cos(x + y) / 1.5
        dphi2_dx = lambda x, y: - s_y * x / (np.sqrt(2) * np.sqrt(1 - x ** 2))
        dphi2_dy = lambda x, y: 0
        # Функции для графика (нулевые контуры):
        F1 = lambda x, y: np.sin(x + y) - (1.5 * x - 0.1)
        F2 = lambda x, y: x ** 2 + 2 * y ** 2 - 1
        system_funcs = (F1, F2)
    else:
        # Система 2: tan(x*y+0.1) = x^2,  x^2+2y^2=1
        # Итерационные преобразования:
        # Для phi1: x = s_x*sqrt(tan(x*y+0.1))
        phi1 = lambda x, y: s_x * np.sqrt(np.tan(x * y + 0.1)) if np.tan(x * y + 0
            (_ for _ in ()).throw(ValueError("tan(x*y+0.1) получило отрицательное з
        phi2 = lambda x, y: s_y * np.sqrt((1 - x ** 2) / 2)
        # Производные для phi1:
        # d/dx sqrt(tan(x*y+0.1)) = (1/(2*sqrt(tan(x*y+0.1)))) * sec^2(x*y+0.1)*y
        dphi1_dx = lambda x, y: s_x * (y * (1 / np.cos(x * y + 0.1) ** 2) / (2 * np
        dphi1_dy = lambda x, y: s_x * (x * (1 / np.cos(x * y + 0.1) ** 2) / (2 * np
        dphi2_dx = lambda x, y: - s_y * x / (np.sqrt(2) * np.sqrt(1 - x ** 2))
        dphi2_dy = lambda x, y: 0
        # Функции для графика:
        F1 = lambda x, y: np.tan(x * y + 0.1) - x ** 2
        F2 = lambda x, y: x ** 2 + 2 * y ** 2 - 1
        system_funcs = (F1, F2)

    # Создаём экземпляр решателя
```

```
    solver = SystemFixedPointSolver(phi1, phi2, dphi1_dx, dphi1_dy, dphi2_dx, dphi2

    # Проверка достаточного условия сходимости
    convergent, norm_val = solver.check_convergence_condition()
    if not convergent:
        print("Достаточное условие сходимости не выполнено.")
        print("Норма якобиана в начальной точке равна {:.4f} (должна быть < 1).".fo
        return
    else:
        print("Достаточное условие сходимости выполнено (норма якобиана = {:.4f}).

    # Решаем систему
    try:
        (x_sol, y_sol), errors, iterations = solver.solve()
    except RuntimeError as e:
        print("Ошибка:", e)
        return
    except ValueError as ve:
        print("Ошибка в вычислениях:", ve)
        return

    # Вывод результатов
    print("\nНайденное решение системы:")
    print("  x = {:.8f}".format(x_sol))
    print("  y = {:.8f}".format(y_sol))
    print("Количество итераций:", iterations)
    print("Вектор погрешностей последней итерации: |Δx| = {:.8e},  |Δy| = {:.8e}".

    # Подстановка найденного решения в исходную систему для проверки
    # Вычисляем значения функций F1 и F2 (для выбранной системы)
    val1 = system_funcs[0](x_sol, y_sol)
    val2 = system_funcs[1](x_sol, y_sol)
    print("\nПогрешности подстановки (значения функций в решении):")
    print("  F1(x, y) = {:.8e}".format(val1))
    print("  F2(x, y) = {:.8e}".format(val2))

    # Определяем диапазоны для графика (с учетом найденного решения)
    x_range = (x_sol - 1, x_sol + 1)
    y_range = (y_sol - 1, y_sol + 1)
    plot_system(system_funcs, (x_sol, y_sol), x_range, y_range)


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