**Московский авиационный институт (национальный исследовательский университет)**

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

**Лабораторная работа по предмету "Численные методы" №2**

Студент: Федоров М.А.

Преподаватель: Ревизников Д.Л.

Группа: М8О-307Б-22

Дата:

Оценка:

Подпись:

Оглавление

[**Задача 1.1** 3](#_Toc192860597)

[**Общий алгоритм решения** 3](#_Toc192860598)

[**Программное решение** 3](#_Toc192860599)

[**Пример работы** 5](#_Toc192860600)

[**Задача 1.2** 7](#_Toc192860601)

[**Общий алгоритм решения** 7](#_Toc192860602)

[**Программное решение** 7](#_Toc192860603)

[**Пример работы для системы из 5-и уравнений** 8](#_Toc192860604)

[**Задача 1.3** 8](#_Toc192860605)

[**Общий алгоритм решения метода итераций** 9](#_Toc192860606)

[**Программное решение** 9](#_Toc192860607)

[**Пример работы** 11](#_Toc192860608)

[**Общий алгоритм решения метода Зейделя** 11](#_Toc192860609)

[**Программное решение** 12](#_Toc192860610)

[**Пример работы** 13](#_Toc192860611)

[**Задача 1.4** 13](#_Toc192860612)

[**Общий алгоритм решения** 14](#_Toc192860613)

[**Программное решение** 15](#_Toc192860614)

[**Пример работы** 16](#_Toc192860615)

[**Задача 1.5** 17](#_Toc192860616)

[**Общий алгоритм решения** 17](#_Toc192860617)

[**Программное решение** 18](#_Toc192860618)

[**Пример работы** 20](#_Toc192860619)

[**Вывод** 21](#_Toc192860620)

# **Задача 2.1**

Реализовать методы простой итерации и Ньютона решения нелинейных уравнений в виде программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически). Проанализировать зависимость погрешности вычислений от количества итераций.

# **Общий алгоритм решения методом Ньютона**

Необходимо решить следующее уравнение:



При нахождении корня уравнения методом Ньютона, итерационный процесс определяется формулой:

Изображение выглядит как Шрифт, текст, линия, белый

Контент, сгенерированный ИИ, может содержать ошибки.

Для начала вычислений требуется задание начального приближения x(0). Условия сходимости метода определяются следующей теоремой:

Пусть на отрезке [a, b] функция f(x) имеет первую и вторую производные постоянного знака и пусть f(a)f(b)<0. Тогда, если f(x(0))f’’(x(0))>0, то начатая с x(0) последовательность x(k) (k=0,1,2,…), определяемая методом Ньютона, монотонно сходится к корню x\*, лежащему в [a, b].

В качестве критерия окончания используется правило |x(n)-x(n-1)|< **ε**

# **Программное решение**

def newton(accuracy: float):

    #задаем функции

    x = sp.symbols('x', real=True)

    expression = x\*\*6-5\*x-2

    first\_diff = sp.diff(expression, x)

    second\_diff = sp.diff(first\_diff, x)

    expression\_labdified = sp.lambdify(x, expression, 'numpy')

    first\_diff\_labdified = sp.lambdify(x, first\_diff, 'numpy')

    second\_diff\_labdified = sp.lambdify(x, second\_diff, 'numpy')

    print("Заданная функция:", expression)

    print("Первая производная:", first\_diff)

    print("Вторая производная:", second\_diff)

    #рисуем график функции и ее производных

    x\_vals = np.linspace(-2, 2, 200)

    y\_expression\_vals = expression\_labdified(x\_vals)

    y\_first\_diff\_vals = first\_diff\_labdified(x\_vals)

    y\_second\_diff\_vals = second\_diff\_labdified(x\_vals)

    plt.figure(figsize=(10, 9))

    plt.plot(x\_vals, y\_expression\_vals, label='f(x)')

    plt.plot(x\_vals, y\_first\_diff\_vals, label="первая производная")

    plt.plot(x\_vals, y\_second\_diff\_vals, label="вторая производная")

    plt.xlabel('x')

    plt.ylabel('y')

    plt.title("График функции и её производной")

    plt.legend()

    plt.ylim(-10, 10)

    plt.grid(True)

    plt.show()

    #по графику видно, что обе производные не меняют знаки на отрезке [1, 2]

    a = 1

    b = 2

    print("Взяли отрезок [a; b]: [", a, ";", b, "]. Значение выражения f(a)\*f(b) должно быть меньше нуля:", expression\_labdified(a)\*expression\_labdified(b))

    if (expression\_labdified(a)\*expression\_labdified(b) >= 0):

        print("Отрезок [a; b] выбран неправильно")

        exit(0)

    x\_0 = 2

    print("Взяли начальную точку x\_0:", x\_0, ". Выражение f(x\_0)\*f''(x\_0) должно быть больше нуля:", expression\_labdified(x\_0)\*second\_diff\_labdified(x\_0))

    if ((expression\_labdified(x\_0)\*second\_diff\_labdified(x\_0) <= 0) or (x\_0 < a) or (x\_0 > b)):

        print("Начальная точка x\_0 выбрана неправильно или не входит в отрезок [a, b]")

        exit(0)

    print("Начинаем вычислять приближения к корню\n")

    x = x\_0 - (expression\_labdified(x\_0) / first\_diff\_labdified(x\_0))

    iteration = 1

    while np.abs(x - x\_0) > accuracy:

        print("Итерация", iteration, "; Значение x:", x, "; Точность вычислений: ", np.abs(x - x\_0), "; значение функции в точке: ", expression\_labdified(x))

        x\_0 = x

        x = x\_0 - (expression\_labdified(x\_0) / first\_diff\_labdified(x\_0))

        iteration += 1

    print("Финальная итерация", iteration, "; Значение x:", x, "; Точность вычислений: ", np.abs(x - x\_0), "; значение функции в точке: ", expression\_labdified(x))

# **Пример работы**

График:ВИзображение выглядит как диаграмма, График, линия, текст

Контент, сгенерированный ИИ, может содержать ошибки.

Вывод программы:

Заданная функция: x\*\*6 - 5\*x - 2

Первая производная: 6\*x\*\*5 - 5

Вторая производная: 30\*x\*\*4

Взяли отрезок [a; b]: [ 1 ; 2 ]. Значение выражения f(a)\*f(b) должно быть меньше нуля: -312

Взяли начальную точку x\_0: 2 . Выражение f(x\_0)\*f''(x\_0) должно быть больше нуля: 24960

Начинаем вычислять приближения к корню

Итерация 1 ; Значение x: 1.7219251336898396 ; Точность вычислений: 0.27807486631016043 ; значение функции в точке: 15.457046557085246

Итерация 2 ; Значение x: 1.5418330872829322 ; Точность вычислений: 0.18009204640690735 ; значение функции в точке: 3.7254168463881054

Итерация 3 ; Значение x: 1.4630388118207538 ; Точность вычислений: 0.07879427546217843 ; значение функции в точке: 0.49178129859087427

Итерация 4 ; Значение x: 1.4490752606517583 ; Точность вычислений: 0.013963551168995503 ; значение функции в точке: 0.013230741842522598

Финальная итерация 5 ; Значение x: 1.4486783692692402 ; Точность вычислений: 0.0003968913825180831 ; значение функции в точке: 1.0414532257030373e-05

# **Общий алгоритм решения методом простой итерации**

Необходимо решить следующее уравнение:



При использовании метода простой итерации начальное уравнение заменяется эквивалентным уравнением с выделенным линейным членом:



Решение ищется путем построения последовательности:  


Начиная с некоторого заданного x(0). Если φ(x) – непрерывная функция, а x(k) (k=0,1,2,…) – сходящаяся последовательность, то предел этой последовательности будет являться решением уравнения.

Условия сходимости метода определяются теоремой:

Пусть функция φ(x) определена и дифференцируема на отрезке [a, b]. Тогда, если ∃q: |φ’(x)| ≤q<1 ∀x ∈ (a, b), то уравнение имеет единственный корень x\* на [a,b]. К этому корню сходится последовательность, определяемая методом простой итерации и начинающаяся с x(0) ∈[a, b]

# **Программное решение**

def simple\_iteration(accuracy: float):

    #задаем функции

    x = sp.symbols('x', real=True)

    expression = x\*\*6-5\*x-2

    equivalent\_expression = np.pow((5\*x+2), 1/6)

    equivalent\_diff = sp.diff(equivalent\_expression, x)

    expression\_lambdified = sp.lambdify(x, expression, 'numpy')

    equivalent\_lambdified = sp.lambdify(x, equivalent\_expression, 'numpy')

    equivalent\_diff\_lambdified = sp.lambdify(x, equivalent\_diff, 'numpy')

    #строим графики (т к один из графиков имеет степень 1/6, то могут выводится предупреждения о наличии отрицательных значений под корнем)

    x\_vals = np.linspace(0, 2, 300)

    y\_expression\_vals = expression\_lambdified(x\_vals)

    y\_equivalent\_vals = equivalent\_lambdified(x\_vals)

    y\_equivalent\_diff\_vals = equivalent\_diff\_lambdified(x\_vals)

    plt.figure(figsize=(10, 9))

    plt.plot(x\_vals, y\_expression\_vals, label='f(x)')

    plt.plot(x\_vals, y\_equivalent\_vals, label="phi(x)")

    plt.plot(x\_vals, y\_equivalent\_diff\_vals, label="phi'(x)")

    plt.xlabel('x')

    plt.ylabel('y')

    plt.title("График функции и её эквивалента")

    plt.legend()

    plt.ylim(-5, 5)

    plt.xticks(np.arange(-7, 7, 1))

    plt.yticks(np.arange(-7, 7, 1))

    plt.grid(True)

    plt.show()

    #задаем a, b и решаем задачу

    a = 0

    b = 2

    print("Выбран отрезок [a; b]: [", a, ";", b,  "]. Проверим условие |phi'(x)| < 1 для любых x, лежащих в [a; b]:")

    for x in np.linspace(a, b, 300):

        if (np.abs(equivalent\_diff\_lambdified(x)) >= 1):

            print("|phi'(x)| > 1 в точке x =", x, "; Значение |phi'(x)| =", np.abs(equivalent\_diff\_lambdified(x)))

            exit(0)

    print("Условие выполняется. Решаем задачу")

    x\_0 = a

    x = equivalent\_lambdified(x\_0)

    iteration = 1

    while np.abs(x - x\_0) > accuracy:

        print("Итерация", iteration, "; Значение x:", x, "; Точность вычислений: ", np.abs(x - x\_0), "; значение функции в точке: ", expression\_lambdified(x))

        x\_0 = x

        x = equivalent\_lambdified(x\_0)

        iteration += 1

    print("Финальная итерация", iteration, "; Значение x:", x, "; Точность вычислений: ", np.abs(x - x\_0), "; значение функции в точке: ", expression\_lambdified(x))

# **Пример работы**

График:Изображение выглядит как текст, График, линия, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Вывод программы:

Выбран отрезок [a; b]: [ 0 ; 2 ]. Проверим условие |phi'(x)| < 1 для любых x, лежащих в [a; b]:

Условие выполняется. Решаем задачу

Итерация 1 ; Значение x: 1.1224620483093732 ; Точность вычислений: 1.1224620483093732 ; значение функции в точке: -5.612310241546863

Итерация 2 ; Значение x: 1.402553425753068 ; Точность вычислений: 0.28009137744369483 ; значение функции в точке: -1.4004568872184437

Итерация 3 ; Значение x: 1.4425903575627261 ; Точность вычислений: 0.04003693180965806 ; значение функции в точке: -0.20018465904825522

Финальная итерация 4 ; Значение x: 1.4478818801808881 ; Точность вычислений: 0.0052915226181620145 ; значение функции в точке: -0.02645761309077166

# **Задача 2.2**

Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них, в котором значения неизвестных являются положительными); начальное приближение определить графически. Проанализировать зависимость погрешности вычислений от количества итераций.

# **Общий алгоритм решения метода Ньютона**

Необходимо найти решение системы:

Если определено начальное приближение x(0)=(x1(0),…, xn(0))T, то итерационный процесс нахождения решения системы (2.11) методом Ньютона можно представить в виде:

Изображение выглядит как текст, Шрифт, линия, рукописный текст

Контент, сгенерированный ИИ, может содержать ошибки.

где значения приращений определяются из решения системы линейных алгебраических уравнений, все коэффициенты которой выражаются через известное предыдущее приближение x(k)

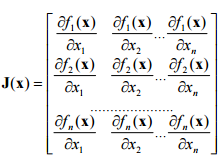
Изображение выглядит как текст, Шрифт, снимок экрана, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Что можно записать в виде:



Где **J(x)** – матрица Якоби:



Как правило, для решения таких систем привлекают самые разные методы. В данном случае использовался метод LU разложения.

Использование метода Ньютона предполагает дифференцируемость функций f1(x),…,fn(x) и невырожденность матрицы Якоби (det **J(x(k))** **≠**0). В случае, если

начальное приближение выбрано в достаточно малой окрестности искомого корня, итерации сходятся к точному решению, причем сходимость квадратичная.

В практических вычислениях в качестве условия окончания итераций обычно используется критерий:  


Где ε – заданная точность

# **Программное решение**

def LU\_for\_newton\_system(J\_lamb: np.matrix, f\_lamb:np.array, x:np.array):

    # Используем метод LU разложения для решения системы f(x\_k)+J\*(delta\_x)=0, в которой ищем вектор приращений delta\_x для решения другой системы методом Ньютона

    # Вычисляем значения J при подставлении x

    J = np.copy(J\_lamb)

    f = np.copy(f\_lamb)

    J[0, 0] = J[0, 0](x[0], x[1])

    J[0, 1] = J[0, 1](x[0], x[1])

    J[1, 0] = J[1, 0](x[0], x[1])

    J[1, 1] = J[1, 1](x[0], x[1])

    # Вычисляем f при подставлении x и умножаем на -1, т к переносим f из правой части в левую

    f[0] = -1\*f[0](x[0], x[1])

    f[1] = -1\*f[1](x[0], x[1])

    U = J.copy()

    f = f.transpose()

    L = np.eye(2, 2)

    p = 0 #кол-во перестановок

    # поиск  LU разложения

    for i in range (0, 2):

        if U[i, i] == 0:

            for j in range(i, 2):

                if U[j, j] != 0:

                    buffer = np.copy(U[i])

                    U[i] = np.copy(U[j])

                    U[j] = np.copy(buffer)

                    p += 1

                    buffer = np.copy(L[i, :i])

                    L[i, :i] = np.copy(L[j, :i])

                    L[j, :i] = np.copy(buffer)

                    buffer = np.copy(f[0, i])

                    f[0, i] = np.copy(f[0, j])

                    f[0, j] = np.copy(buffer)

        for j in range(i+1, 2):

            mu\_j = (U[j, i]/U[i, i])

            L[j, i] = mu\_j

            U[j] = U[j]-U[i]\*mu\_j

    # print("Верхняя матрица:\n", U, "\nНижняя матрица:\n", L)

    z = np.zeros(2)

    delta\_x = np.zeros(2)

    # решение Lz=f

    for i in range(0, 2):

        summ = 0

        for j in range(0, i):

            summ += L[i, j]\*z[j]

        z[i] = f[i]-summ

    # Решение Ux=z

    for i in range(1, -1, -1):

        summ = 0

        for j in range(i+1, 2):

            summ += U[i, j]\*delta\_x[j]

        delta\_x[i] = (1 / U[i, i])\*(z[i]-summ)

    print("Решения СЛАУ в векторе delta\_x: ", delta\_x)

    det = np.pow(-1, p)

    for i in range(0, 2):

        det \*= U[i, i]

    # print(, "Определитель: ", det)

    print("Определитель матрицы Якоби: ", det, "\nПроверка определителя: ", np.linalg.det(np.array(J, dtype=np.float64)))

    if (np.abs(det - np.linalg.det(np.array(J, dtype=np.float64))) > 0.0000001):

        print("Ошибка решения")

        exit(0)

    #проверка правильности решения с помощью вычисления вектора b

    print("Проверка правильности решения (Свободные члены, вычесленные из вектора delta\_x и свободные члены вектора f):")

    for i in range (0, 2):

        answer = 0.

        for j in range(0, 2):

            answer += J[i, j]\*delta\_x[j]

        print(answer, "\t\t", f[i])

        if (np.abs(answer - f[i]) > 0.0000001):

            print("Ошибка решения")

            exit(0)

    return delta\_x

def newton\_system(accuracy: float):

    x = np.array([sp.symbols('x1', real=True), sp.symbols('x2', real=True)])

    f = np.array([3\*x[0]\*\*2-x[0]+x[1]\*\*2-1, x[1]-sp.tan(x[0])])

    J = np.matrix([[sp.diff(f[0], x[0]), sp.diff(f[0], x[1])],

                   [sp.diff(f[1], x[0]), sp.diff(f[1], x[1])]])

    f\_lamb = np.array([sp.lambdify(x, f[0], 'numpy'), sp.lambdify(x, f[1], 'numpy')])

    J\_lamb = np.matrix([[sp.lambdify(x, J[0, 0], 'numpy'), sp.lambdify(x, J[0, 1], 'numpy')],

                        [sp.lambdify(x, J[1, 0], 'numpy'), sp.lambdify(x, J[1, 1], 'numpy')]])

    det = (J[0, 0]\*J[1, 1])-(J[0, 1]\*J[1, 0])

    det\_lamb = sp.lambdify(x, det, 'numpy')

    print("Система уравнений:\n", f[0], "\n", f[1])

    print("Матрица Якоби (1-е производные):\n[", J[0, 0], ",\t", J[0, 1], "]\n[", J[1, 0], ",\t", J[1, 1], "]")

    print("Определитель матрицы Якоби считается по формуле:\n", det)

    #рисуем график функции и ее производных

    x1\_vals = np.linspace(-5, 5, 100)

    x2\_vals = np.linspace(-5, 5, 100)

    X1, X2 = np.meshgrid(x1\_vals, x2\_vals)

    plt.figure(figsize=(9, 6))

    plt.contour(x1\_vals, x2\_vals, f\_lamb[0](X1, X2), levels=[0], colors='r')

    plt.contour(x1\_vals, x2\_vals, f\_lamb[1](X1, X2), levels=[0], colors='b')

    plt.xlabel('x1')

    plt.ylabel('x2')

    plt.title("Графики первого и второго уравнений")

    plt.xticks(np.arange(-5, 5, 1))

    plt.yticks(np.arange(-5, 5, 1))

    plt.grid(True)

    plt.show()

    x\_current = np.array([0.5, 0.5])

    print("\nВычисляем delta\_x для 1 итерации метода Ньютона")

    delta\_x = LU\_for\_newton\_system(J\_lamb, f\_lamb, x\_current)

    x\_current = x\_current + delta\_x

    iteration = 1

    while(np.linalg.norm(delta\_x) >= accuracy and iteration < 30):

        print("Текущее значение x:", x\_current, "\nТочность вычитслений:", np.linalg.norm(delta\_x))

        iteration += 1

        print("\nВычисляем delta\_x для", iteration,  "итерации метода Ньютона")

        delta\_x = LU\_for\_newton\_system(J\_lamb, f\_lamb, x\_current)

        x\_current = x\_current + delta\_x

    print("Итоговое решение x:", x\_current, "\nТочность вычислений:", np.linalg.norm(delta\_x))

# **Пример работы**

График: Изображение выглядит как текст, График, линия, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Вывод программы:

Система уравнений:

3\*x1\*\*2 - x1 + x2\*\*2 - 1

x2 - tan(x1)

Матрица Якоби (1-е производные):

[ 6\*x1 - 1 , 2\*x2 ]

[ -tan(x1)\*\*2 - 1 , 1 ]

Определитель матрицы Якоби считается по формуле:

6\*x1 - 2\*x2\*(-tan(x1)\*\*2 - 1) - 1

Вычисляем delta\_x для 1 итерации метода Ньютона

Решения СЛАУ в векторе delta\_x: [0.13754885 0.2249023 ]

Определитель матрицы Якоби: 3.298446410409525

Проверка определителя: 3.298446410409525

Проверка правильности решения (Свободные члены, вычесленные из вектора delta\_x и свободные члены вектора f):

0.5 0.5

0.046302489843790456 0.046302489843790484

Текущее значение x: [0.63754885 0.7249023 ]

Точность вычитслений: 0.2636299124795697

Вычисляем delta\_x для 2 итерации метода Ньютона

Решения СЛАУ в векторе delta\_x: [-0.02569772 -0.02395945]

Определитель матрицы Якоби: 5.070601082024162

Проверка определителя: 5.0706010820241625

Проверка правильности решения (Свободные члены, вычесленные из вектора delta\_x и свободные члены вектора f):

-0.10734010296286955 -0.10734010296286955

0.015838520646937245 0.01583852064693725

Текущее значение x: [0.61185113 0.70094285]

Точность вычитслений: 0.035134427015122

Вычисляем delta\_x для 3 итерации метода Ньютона

Решения СЛАУ в векторе delta\_x: [-0.00075275 -0.0003884 ]

Определитель матрицы Якоби: 4.763213406470932

Проверка определителя: 4.763213406470932

Проверка правильности решения (Свободные члены, вычесленные из вектора delta\_x и свободные члены вектора f):

-0.0025551731060879934 -0.0025551731060879934

0.0007349785676062928 0.0007349785676062925

Итоговое решение x: [0.61109838 0.70055445]

Точность вычислений: 0.0008470478321929508

# **Общий алгоритм решения методом простой итерации**

Необходимо найти решение системы:

При использовании метода простой итерации система уравнений приводится к эквивалентной системе специального вида:

Изображение выглядит как текст, Шрифт, линия, рукописный текст

Контент, сгенерированный ИИ, может содержать ошибки.

Или в векторной форме:  
Изображение выглядит как текст, Шрифт, рукописный текст, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Где функции ϕ1(x),…, ϕn(x) определены и непрерывны в некоторой окрестности искомого изолированного решения x\*=(x1\*,…, xn\*)T.

Если выбрано некоторое начальное приближение x(0)=(x1(0),…, xn(0))T, то последующие приближения в методе простой итерации находятся по формулам:

Изображение выглядит как текст, Шрифт, линия, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

Или в векторной форме:



Если последовательность векторов x(k)=(x1(k),…, xn(k))T сходится, то она сходится к решению x\*=(x1\*,…, xn\*)T.

Достаточное условие сходимости итерационного процесса формулируется следующим образом:

Пусть вектор-функция ϕ(x) непрерывна, вместе со своей производной:

Изображение выглядит как текст, Шрифт, снимок экрана, число

Контент, сгенерированный ИИ, может содержать ошибки.

В ограниченной выпуклой замкнутой области G и



Где q – постоянная. Если x(0) ∈ G и все последовательные приближения так же содержатся в G, то процесс итерации сходится к единственному решению уравнения x= ϕ(x) в области G.

# **Программное решение**

def simple\_iterations\_system(accuracy: float):

    x = np.array([sp.symbols('x1', real=True), sp.symbols('x2', real=True)])

    f = np.array([3\*x[0]\*\*2-x[0]+x[1]\*\*2-1, x[1]-sp.tan(x[0])])

    phi = np.array([(1+np.pow(13-12\*x[1]\*\*2, 1/2))/6, sp.tan(x[0])])

    # phi = np.array([sp.atan(x[1]), np.pow(1-3\*x[0]\*\*2+x[0], 1/2)])

    phi\_diff = np.matrix([[sp.diff(phi[0], x[0]), sp.diff(phi[0], x[1])],

                          [sp.diff(phi[1], x[0]), sp.diff(phi[1], x[1])]])

    diff\_norm = np.pow(np.abs(phi\_diff[0, 1] \* phi\_diff[1, 0]), 1/2)

    f\_lamb = np.array([sp.lambdify(x, f[0], 'numpy'), sp.lambdify(x, f[1], 'numpy')])

    phi\_lamb = np.array([sp.lambdify(x[1], phi[0], 'numpy'), sp.lambdify(x[0], phi[1], 'numpy')])

    phi\_diff\_lamb = np.matrix([[sp.lambdify(x[1], phi\_diff[0, 0], 'numpy'), sp.lambdify(x[1], phi\_diff[0, 1], 'numpy')],

                          [sp.lambdify(x[0], phi\_diff[1, 0], 'numpy'), sp.lambdify(x[0], phi\_diff[1, 1], 'numpy')]])

    diff\_norm\_lamb = sp.lambdify(x, diff\_norm, 'numpy')

    print("Система уравнений:\n", f[0], "\n", f[1])

    print("Эквивалентные функции:\n", phi[0], "\n", phi[1])

    print("Матрица производных эквивалентных функций:\n[", phi\_diff[0, 0], ",\t", phi\_diff[0, 1], "]\n[", phi\_diff[1, 0], ",\t", phi\_diff[1, 1], "]")

    print("Норма матрицы производных:\n", diff\_norm)

    #рисуем график функции и ее производных (могут вылетать предупреждения о наличии отрицательных чисел под корнем)

    x1\_vals = np.linspace(-2, 2, 300)

    x2\_vals = np.linspace(-2, 2, 300)

    zeros = np.zeros(300)

    X1, X2 = np.meshgrid(x1\_vals, x2\_vals)

    plt.figure(figsize=(9, 6))

    plt.title("Графики первого и второго уравнений")

    plt.contour(x1\_vals, x2\_vals, f\_lamb[0](X1, X2), levels=[0], colors='r')

    plt.contour(x1\_vals, x2\_vals, f\_lamb[1](X1, X2), levels=[0], colors='b')

    plt.xlabel('x1')

    plt.ylabel('x2')

    plt.xticks(np.arange(-2, 2, 0.5))

    plt.yticks(np.arange(-2, 2, 0.5))

    plt.grid(True)

    plt.show()

    plt.figure(figsize=(9, 6))

    plt.title("Графики эквивалентных функций и их производных")

    plt.plot(phi\_lamb[0](x2\_vals), x2\_vals, color='r', linestyle='-', label='x1=phi1(x2)')

    plt.plot(x1\_vals, phi\_lamb[1](x1\_vals), color='b', label='x2=phi2(x1)')

    plt.plot(zeros, x2\_vals, linestyle='--', color='r', label="x1=phi1(x2)'x1")

    plt.plot(phi\_diff\_lamb[0, 1](x2\_vals), x2\_vals, color='r', linestyle='-.', label="x1=phi1(x2)'x2")

    plt.plot(x1\_vals, phi\_diff\_lamb[1, 0](x1\_vals), linestyle='--', color='b', label="x2=phi2(x1)'x1")

    plt.plot(x1\_vals, zeros, color='b', linestyle='-.', label="x2=phi2(x1)'x2")

    plt.xlabel('x1')

    plt.ylabel('x2')

    plt.ylim(-2, 2)

    plt.xlim(-2, 2)

    plt.xticks(np.arange(-2, 2, 0.5))

    plt.yticks(np.arange(-2, 2, 0.5))

    plt.legend()

    plt.grid(True)

    plt.show()

    Z = diff\_norm\_lamb(X1, X2)

    plt.figure(figsize=(9, 6))

    plt.title("График области значений нормы матрицы производных")

    contour = plt.contourf(X1, X2, Z, levels=[0, 1], colors='lightblue', alpha=0.7)

    contour\_line = plt.contour(X1, X2, Z, levels=[1], colors='red', linewidths=2)

    plt.xlabel('x1')

    plt.ylabel('x2')

    plt.ylim(-2, 2)

    plt.xlim(-2, 2)

    plt.xticks(np.arange(-2, 2, 0.2))

    plt.yticks(np.arange(-2, 2, 0.2))

    plt.grid(True)

    plt.show()

    print("Решаем задачу")

    x\_0 = [0.5, 0.5]

    x[0] = phi\_lamb[0](x\_0[1])

    x[1] = phi\_lamb[1](x\_0[0])

    iteration = 1

    while np.linalg.norm(x - x\_0) > accuracy and iteration < 30:

        print("Итерация", iteration, "; Значение x:", x, "; Точность вычислений: ", np.linalg.norm(x - x\_0))

        x\_0 = np.copy(x)

        x[0] = phi\_lamb[0](x\_0[1])

        x[1] = phi\_lamb[1](x\_0[0])

        iteration += 1

    print("Финальная итерация", iteration, "; Значение x:", x, "; Точность вычислений: ", np.linalg.norm(x - x\_0))

# **Пример работы**

Графики:

Изображение выглядит как линия, График, диаграмма, текст

Контент, сгенерированный ИИ, может содержать ошибки. Изображение выглядит как текст, линия, диаграмма, График

Контент, сгенерированный ИИ, может содержать ошибки. Изображение выглядит как График, диаграмма, линия, текст

Контент, сгенерированный ИИ, может содержать ошибки.

Вывод программы:

Система уравнений:

3\*x1\*\*2 - x1 + x2\*\*2 - 1

x2 - tan(x1)

Эквивалентные функции:

(13 - 12\*x2\*\*2)\*\*0.5/6 + 1/6

tan(x1)

Матрица производных эквивалентных функций:

[ 0 , -2.0\*x2/(13 - 12\*x2\*\*2)\*\*0.5 ]

[ tan(x1)\*\*2 + 1 , 0 ]

Норма матрицы производных:

1.4142135623731\*(tan(x1)\*\*2 + 1)\*\*0.5\*Abs(x2)\*\*0.5/Abs((13 - 12\*x2\*\*2)\*\*0.5)\*\*0.5

<lambdifygenerated-3>:2: RuntimeWarning: invalid value encountered in sqrt

return (1/6)\*(13 - 12\*x2\*\*2)\*\*0.5 + 1/6

<lambdifygenerated-6>:2: RuntimeWarning: invalid value encountered in power

return -2.0\*x2\*(13 - 12\*x2\*\*2)\*\*(-0.5)

<lambdifygenerated-9>:2: RuntimeWarning: invalid value encountered in sqrt

return 1.4142135623731\*(tan(x1)\*\*2 + 1)\*\*0.5\*abs(x2)\*\*0.5\*abs((13 - 12\*x2\*\*2)\*\*0.5)\*\*(-0.5)

Решаем задачу

Итерация 1 ; Значение x: [0.6937129433613966 np.float64(0.5463024898437905)] ; Точность вычислений: 0.19916983956279613

Итерация 2 ; Значение x: [np.float64(0.6781634646116158) np.float64(0.8315974464167362)] ; Точность вычислений: 0.2857183902645205

Итерация 3 ; Значение x: [np.float64(0.5280432110293286) np.float64(0.8056283692685717)] ; Точность вычислений: 0.1523498720168072

Итерация 4 ; Значение x: [np.float64(0.5471471803871439) np.float64(0.5832914673071183)] ; Точность вычислений: 0.22315613282865718

Итерация 5 ; Значение x: [np.float64(0.664362824812726) np.float64(0.6091868673461557)] ; Точность вычислений: 0.12004198865932814

Итерация 6 ; Значение x: [np.float64(0.6539122221402597) np.float64(0.7831194322111479)] ; Точность вычислений: 0.17424624017961615

Финальная итерация 7 ; Значение x: [np.float64(0.5625021915279499) np.float64(0.7663959823255929)] ; Точность вычислений: 0.09292721599519745

# **Вывод**

В ходе выполнения лабораторной работы я научился применять алгоритмы, находящие приближения решений нелинейных уравнений и их систем.