ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ   
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ   
ВЫСШЕГО ОБРАЗОВАНИЯ  
МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Отчёт по лабораторным работам

по курсу «Численные методы»

III курс, VI семестр  
Вариант 2

Студент: Андрюшин Л.Д.

Группа: М8О-312Б-21

Руководитель: Демидова О.Л.

Москва 2025

Оглавление

[Лабораторная работа № 1 2](#_Toc197994949)

[1.1 LU-разложение 2](#_Toc197994950)

[1.2 Метод прогонки 8](#_Toc197994951)

[1.3 Метод простых итераций и метод Зейделя 12](#_Toc197994952)

[1.4 Метод вращения Якоби 19](#_Toc197994953)

[1.5 QR-разложение 25](#_Toc197994954)

[Лабораторная работа № 2 30](#_Toc197994955)

[2.1 Метод простой итерации и метод Ньютона для решения нелинейных уравнений 30](#_Toc197994956)

[2.2 Метод простой итерации и метод Ньютона для решения системы нелинейных уравнений 37](#_Toc197994957)

[Лабораторная работа № 3 37](#_Toc197994958)

[3.1 Построить интерполяционные многочлены Лагранжа и Ньютона 44](#_Toc197994959)

[3.2 Построить кубический сплайн 50](#_Toc197994960)

[3.3 Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены 54](#_Toc197994961)

[3.4 Вычислить первую и вторую производную от таблично заданной функции 59](#_Toc197994962)

[3.5 Вычислить определенный интеграл методами прямоугольников, трапеций, Симпсона 61](#_Toc197994963)

[Лабораторная работа № 4 64](#_Toc197994964)

[4.1 Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка 64](#_Toc197994965)

[4.2 Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ 67](#_Toc197994966)

# Лабораторная работа № 1

## LU-разложение

Реализовать алгоритм LU - разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.

Система уравнений:



**Общая идея LU-разложения**

LU-разложение — это разложение квадратной матрицы AAA в произведение двух треугольных матриц:

A=LU, где:

* L — нижнетреугольная матрица (все элементы выше главной диагонали равны нулю, диагональ состоит из единиц)
* U — верхнетреугольная матрица (все элементы ниже главной диагонали равны нулю).

Разложение удобно для решения систем линейных уравнений, вычисления определителя и нахождения обратной матрицы.

**Алгоритм построения LU-разложения**

Алгоритм LU-разложения состоит из нескольких шагов. Основной идеей является преобразование матрицы A в две матрицы L и U, где матрица L будет содержать элементы, расположенные ниже главной диагонали, а матрица U — элементы на и выше главной диагонали.

1. Для каждого столбца k от 0 до n-1 (где n — размерность матрицы):
   * Для каждого элемента ниже диагонали (где i > k) вычисляется коэффициент, который будет храниться в матрице L:
   * После этого для всех элементов в строках ниже диагонали происходит корректировка матрицы A, чтобы элементы над главной диагональю стали нулями:

, где i > k и j > k.

1. В конце этих операций матрица A будет представлять собой комбинацию матриц L и U.
2. Диагональ матрицы L будет состоять из единиц, и после выполнения этих шагов матрица A преобразуется в две матрицы: одну нижнюю треугольную (L) и одну верхнюю треугольную (U).

**Решение системы с помощью LU-разложения**

После того, как матрица A разложена на две матрицы L и U, можно решать систему линейных уравнений следующим образом:

1. Сначала решаем систему с помощью прямого хода:
   * Для этого поочередно вычисляем элементы вектора y, начиная с первого:

- сумма() для j от 0 до i-1.

1. Затем решаем систему с помощью обратного хода:
   * Для этого поочередно вычисляем элементы вектора x, начиная с последнего:

**Определитель и обратная матрица**

Определитель матрицы A можно вычислить как произведение диагональных элементов матрицы U:

Обратную матрицу можно найти с помощью LU-разложения, решив систему для каждого столбца единичной матрицы (по одному за раз). Каждый такой столбец дает соответствующий столбец матрицы .

Код программы:

def read\_matrix\_from\_file(filename):

    """Считывает матрицу A и вектор b из файла"""

    with open(filename, 'r') as f:

        lines = f.readlines()

    n = int(lines[0])  # Размерность матрицы

    A = [list(map(float, lines[i + 1].split())) for i in range(n)]

    b = [float(lines[n + 1 + i]) for i in range(n)]

    return n, A, b

def lu\_decomposition\_inplace(A):

    """LU-разложение"""

    n = len(A)

    for k in range(n):

        for i in range(k + 1, n):

            A[i][k] /= A[k][k]

            for j in range(k + 1, n):

                A[i][j] -= A[i][k] \* A[k][j]

    return A

def solve\_lu\_system(A, b):

    """Решает систему LUx = b в одном шаге"""

    n = len(A)

    x = [0] \* n

    # Прямой ход: решаем систему L \* y = b

    for i in range(n):

        b[i] -= sum(A[i][j] \* b[j] for j in range(i))

    # Обратный ход: решаем систему U \* x = y

    for i in range(n - 1, -1, -1):

        b[i] = (b[i] - sum(A[i][j] \* b[j] for j in range(i + 1, n))) / A[i][i]

    return b

def determinant(A):

    """Вычисляет определитель через произведение диагональных элементов U"""

    det = 1

    for i in range(len(A)):

        det \*= A[i][i]

    return det

def inverse\_matrix(A):

    """Вычисляет обратную матрицу через LU-разложение"""

    n = len(A)

    I = [[1 if i == j else 0 for j in range(n)] for i in range(n)]  # Единичная матрица

    inv\_A = []

    for e in zip(\*I):  # Берем столбцы единичной матрицы

        b = list(e)

        # Решаем систему LUx = b

        x = solve\_lu\_system(A, b)

        inv\_A.append(x)

    return list(map(list, zip(\*inv\_A)))  # Транспонируем матрицу

def multiply\_matrices(A, B):

    """Перемножает две квадратные матрицы"""

    n = len(A)

    return [[sum(A[i][k] \* B[k][j] for k in range(n)) for j in range(n)] for i in range(n)]

def multiply\_matrix\_vector(A, x):

    """Перемножает матрицу A на вектор x"""

    return [sum(A[i][j] \* x[j] for j in range(len(A))) for i in range(len(A))]

def print\_matrix(matrix, name):

    """Выводим матрицу в консоль"""

    print(f"\n{name}:")

    for row in matrix:

        print(" ".join(map(str, row)))

def solve\_system(filename):

    n, A, b = read\_matrix\_from\_file(filename)

    A\_original = [row[:] for row in A]

    # Выполняем LU-разложение

    lu\_decomposition\_inplace(A)

    # Выводим LU-матрицу

    print\_matrix(A, "LU-разложение")

    # Решаем систему LUx = b

    x = solve\_lu\_system(A, b)

    # Вычисляем определитель

    det\_A = determinant(A)

    # Вычисляем обратную матрицу

    inv\_A = inverse\_matrix(A)

    # Выводим результаты

    print("\nРешение СЛАУ:", x)

    print("Определитель:", det\_A)

    print\_matrix(inv\_A, "Обратная матрица")

    # Вычисляем A \* A^(-1)

    identity\_check = multiply\_matrices(A\_original, inv\_A)

    print\_matrix(identity\_check, "A \* A^(-1)")

    # Вычисляем A \* x

    Ax = multiply\_matrix\_vector(A\_original, x)

    print("\nA \* x:")

    for value in Ax:

        print(value)

# Запуск решения

solve\_system("input.txt")

Результат работы программы:

LU-разложение:

2.0 7.0 -8.0 6.0

2.0 -10.0 16.0 -19.0

-0.5 -0.05 2.8 5.05

4.5 3.85 -9.857142857142858 87.92857142857144

Решение СЛАУ: [8.000000000000004, -3.0000000000000013, 1.9999999999999993, -3.0]

Определитель: -4924.000000000001

Обратная матрица:

0.10235580828594631 0.07189277010560532 0.16125101543460596 0.07432981316003248

0.039805036555645844 0.11129163281884642 0.03493095044679131 -0.05442729488220958

-0.0036555645816409303 0.08671811535337123 0.15495532087733555 -0.02051177904142973

0.08123476848090982 -0.038180341186027617 0.11210398050365555 0.011372867587327374

A \* A^(-1):

1.0 0.0 2.7755575615628914e-17 1.3877787807814457e-17

-1.3877787807814457e-16 1.0000000000000002 1.942890293094024e-16 0.0

5.204170427930421e-17 -1.1102230246251565e-16 1.0 6.938893903907228e-18

-7.762887554996212e-16 1.4988010832439613e-15 -9.159339953157541e-16 0.9999999999999999

A \* x:

-39.0

41.00000000000001

3.9999999999999964

113.00000000000004

## Метод прогонки

Реализовать метод прогонки в виде программы, задавая в качестве входных данных ненулевые элементы матрицы системы и вектор правых частей. Используя разработанное программное обеспечение, решить СЛАУ с трехдиагональной матрицей

Условие:

2.

**Алгоритм метода прогонки:**

1. Прямым ходом находим коэффициенты прогонки с помощью выведенных формул.

,

1. Обратным ходом поочерёдно находим значения иксов, начиная с последнего по формуле.

**Необохдимые и достаточные условия метода прогонки**

Метод прогонки можно применить только к **трехдиагональной матрице**. Это означает, что все элементы матрицы, которые находятся не на главной диагонали и соседних с ней диагоналях, должны быть равны нулю.

Метод прогонки не может работать, если на главной диагонали матрицы присутствуют нули. Это связано с тем, что при решении системы уравнений необходимо делить на элементы главной диагонали. Если элемент равен нулю, это приведет к делению на ноль.

Для того чтобы метод прогонки работал корректно, необходимо, чтобы система уравнений была **диагонально преобладающей**. Это означает, что для каждой строки матрицы сумма модулей элементов, не находящихся на главной диагонали, должна быть меньше или равна модулю элемента на главной диагонали. И хотя бы в одном месте должна быть стого больше. В крайних случаях она должна быть строго больше.

Код программы:

def read\_matrix\_from\_file(filename):

    """Считывает трехдиагональную матрицу A и вектор d из файла."""

    with open(filename, 'r') as f:

        lines = f.readlines()

    n = int(lines[0])

    A = [list(map(float, lines[i + 1].split())) for i in range(n)]

    d = [float(lines[n + 1 + i]) for i in range(n)]

    return n, A, d

def check\_conditions(A):

    """Проверяет необходимые условия для метода прогонки."""

    n = len(A)

    # Проверка на трехдиагональность (все элементы вне главной, верхней и нижней диагоналей должны быть нулевыми)

    for i in range(n):

        for j in range(n):

            if abs(i - j) > 1 and abs(A[i][j]) > 1e-9:

                raise ValueError("Ошибка: матрица не является трехдиагональной.")

    # Проверка наличия нулей на главной диагонали

    if any(abs(A[i][i]) < 1e-9 for i in range(n)):

        raise ValueError("Ошибка: на главной диагонали есть нулевой элемент.")

def check\_sufficient\_conditions(A):

    """Проверяет достаточное условие диагонального преобладания."""

    n = len(A)

    ok = True

    ok2 = False

    for i in range(1, n-1):

        if abs(A[i][i]) <= abs(A[i][i-1]) + abs(A[i][i+1]):

            ok = False

        if abs(A[i][i]) > abs(A[i][i-1]) + abs(A[i][i+1]):

            ok2 = True

    if abs(A[0][0]) <= abs(A[0][1]):

        ok = False

    if abs(A[0][0]) > abs(A[0][1]):

        ok2 = True

    if abs(A[n-1][n-1]) <= abs(A[n-1][n-2]):

        ok = False

    if abs(A[n-1][n-1]) > abs(A[n-1][n-2]):

        ok2 = True

    if not ok or not ok2:

        print("⚠️  Предупреждение: достаточное условие диагонального преобладания не выполнено!")

def extract\_tridiagonal\_vectors(A, d):

    """Извлекает коэффициенты трехдиагональной матрицы в векторы lower, main, upper и d."""

    n = len(A)

    lower = [0] + [A[i][i - 1] for i in range(1, n)]  # Нижняя диагональ

    main = [A[i][i] for i in range(n)]               # Главная диагональ

    upper = [A[i][i + 1] for i in range(n - 1)] + [0] # Верхняя диагональ

    return lower, main, upper, d

def thomas\_algorithm(lower, main, upper, d):

    """Решает СЛАУ с трехдиагональной матрицей методом прогонки."""

    n = len(main)

    # Прямой ход

    p = [0] \* n

    q = [0] \* n

    p[0] = -upper[0] / main[0]

    q[0] = d[0] / main[0]

    for i in range(1, n):

        denominator = main[i] + lower[i] \* p[i - 1]

        p[i] = -upper[i] / denominator if i < n - 1 else 0

        q[i] = (d[i] - lower[i] \* q[i - 1]) / denominator

    # Обратный ход

    x = [0] \* n

    x[n - 1] = q[n - 1]

    for i in range(n - 2, -1, -1):

        x[i] = p[i] \* x[i + 1] + q[i]

    return x

def multiply\_matrix\_vector(A, x):

    """Перемножает матрицу A на вектор x, возвращая получившийся вектор."""

    return [sum(A[i][j] \* x[j] for j in range(len(A))) for i in range(len(A))]

def solve\_system(filename):

    """Читает матрицу, проверяет условия, решает систему и выводит результат."""

    # Считываем данные

    n, A, d = read\_matrix\_from\_file(filename)

    # Проверяем необходимые условия метода прогонки

    check\_conditions(A)

    # Проверяем достаточное условие

    check\_sufficient\_conditions(A)

    # Извлекаем векторы lower, main, upper и d

    lower, main, upper, d = extract\_tridiagonal\_vectors(A, d)

    # Решаем систему методом прогонки

    x = thomas\_algorithm(lower, main, upper, d)

    # Выводим найденный вектор x

    print("\nРешение СЛАУ:")

    for value in x:

        print(f"{value:.1f}")

    # Проверяем результат умножением A \* x

    Ax = multiply\_matrix\_vector(A, x)

    print("\nПроверка: A \* x =")

    for value in Ax:

        print(f"{value:.1f}")

# Запуск решения

solve\_system("input1.2txt")

Вывод программы:

Решение СЛАУ:

-9.0

-6.0

2.0

-7.0

-7.0

Проверка: A \* x =

-120.0

-91.0

5.0

-74.0

-56.0

## Метод простых итераций и метод Зейделя

Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности.

Условие:

2.

При большом числе уравнений прямые методы решения СЛАУ (за исключением метода прогонки) становятся труднореализуемыми на ЭВМ прежде всего из-за сложности хранения и обработки матриц большой размерности. В то же время характерной особенностью ряда часто встречающихся в прикладных задачах СЛАУ является разреженность матриц. Число ненулевых элементов таких матриц мало по сравнению с их размерностью. Для решения СЛАУ с разреженными матрицами предпочтительнее использовать итерационные методы.

Методы последовательных приближений, в которых при вычислении последующего приближения решения используются предыдущие, уже известные приближенные решения, называются итерационными.

**Алгоритм метода простых итераций:**

1. Приводим систему к эквивалентному виду;

Или в векторно-матричной форме

x = +

1. Выбираем точность, которую записываем в переменную epsilon;
2. Заполним вектор beta и матрицу Alpha, исходя из формулы;
3. В качестве нулевого приближения столбца решений принимаем beta;
4. Проводим итерации до того момента, пока формула не будет выполнена.

**Алгоритм метода Зейделя:**

1. Приводим систему к эквивалентному виду.
2. Находим матрицу В — нижнюю-треугольную, С — верхнюю-треугольную, которые следуют из следующего соотношения.

Из этой системы видно, что =+B+C

1. Находим обратную матрицу (Е - В), где Е - единичная матрица.
2. Подставляем полученные значения в итерационную формулу.
3. Проводим итерации до того момента, пока условие выхода не будет выполнено.

Для того чтобы методы простых итераций и Зейделя сходились, необходимо выполнение **достаточного условия диагонального преобладания**.

**Диагональное преобладание**

Условие диагонального преобладания означает, что элементы главной диагонали матрицы должны быть значительно больше по величине, чем сумма абсолютных значений остальных элементов в соответствующей строке или столбце. Это условие важно для того, чтобы итерационные методы сходились, так как оно гарантирует, что на каждом шаге величина ошибки будет уменьшаться.

* **Диагональное преобладание по строкам**: Для каждого элемента главной диагонали матрицы его абсолютное значение должно быть больше суммы абсолютных значений всех остальных элементов в той же строке. То есть, для каждой строки матрицы должен выполняться критерий, что элемент на главной диагонали больше по величине, чем сумма всех других элементов в этой строке.

Если это условие не выполняется по строкам, то мы переходим к **диагональному преобладанию по столбцам**.

* **Диагональное преобладание по столбцам**: В этом случае для каждого столбца матрицы мы проверяем, что элемент на главной диагонали столбца больше суммы всех других элементов в этом столбце.

Если ни по строкам, ни по столбцам не выполняется диагональное преобладание, то методы **простых итераций** и **Зейделя** не сходятся, и решение системы уравнений с помощью этих методов становится невозможным.

Код программы:

def read\_matrix\_from\_file(filename):

    """Считывает матрицу A, вектор B и точность из файла."""

    with open(filename, 'r') as f:

        lines = f.readlines()

    n = int(lines[0].strip())  # Размерность матрицы

    A = [list(map(float, lines[i + 1].split())) for i in range(n)]  # Матрица A

    B = [float(lines[n + 1 + i]) for i in range(n)]  # Вектор B

    eps = float(lines[-1].strip())  # Точность

    return n, A, B, eps

def check\_conditions(A):

    """Проверяет диагональное преобладание."""

    n = len(A)

    has\_strict = False

    for i in range(n):

        sum\_other = sum(abs(A[i][j]) for j in range(n) if i != j)

        if abs(A[i][i]) > sum\_other:

            has\_strict = True

        if abs(A[i][i]) <= sum\_other:

            check\_column\_conditions(A)

            return

    if not has\_strict:

        check\_column\_conditions(A)

def check\_column\_conditions(A):

    """Проверяет диагональное преобладание по столбцам."""

    n = len(A)

    has\_strict = False

    for j in range(n):

        sum\_other = sum(abs(A[i][j]) for i in range(n) if i != j)

        if abs(A[j][j]) > sum\_other:

            has\_strict = True

        if abs(A[j][j]) <= sum\_other:

            print("⚠️  Предупреждение: диагональное преобладание не выполнено!")

            return

    if not has\_strict:

        print("⚠️  Предупреждение: диагональное преобладание не выполнено!")

def simple\_iterations(A, B, eps):

    """Решает систему методом простых итераций."""

    n = len(A)

    x = [0] \* n

    iterations = 0

    print("\n🔹 Метод простых итераций:")

    while True:

        x\_new = [0] \* n

        for i in range(n):

            x\_new[i] = (B[i] - sum(A[i][j] \* x[j] for j in range(n) if j != i)) / A[i][i]

        error = sum((x\_new[i] - x[i]) \*\* 2 for i in range(n)) \*\* 0.5

        x = x\_new

        iterations += 1

        print(f"Шаг {iterations}: x = {x}, ошибка = {error:.6f}")

        if error < eps:

            break

    return x, iterations

def seidel\_method(A, B, eps):

    """Решает систему методом Зейделя."""

    n = len(A)

    x = [0] \* n

    iterations = 0

    print("\n🔹 Метод Зейделя:")

    while True:

        x\_new = x[:]

        for i in range(n):

            x\_new[i] = (B[i] - sum(A[i][j] \* x\_new[j] if j < i else A[i][j] \* x[j] for j in range(n) if j != i)) / A[i][i]

        error = sum((x\_new[i] - x[i]) \*\* 2 for i in range(n)) \*\* 0.5

        x = x\_new

        iterations += 1

        print(f"Шаг {iterations}: x = {x}, ошибка = {error:.6f}")

        if error < eps:

            break

    return x, iterations

def multiply\_matrix\_vector(A, x):

    """Перемножает матрицу A на вектор x, возвращая получившийся вектор."""

    return [sum(A[i][j] \* x[j] for j in range(len(A))) for i in range(len(A))]

def solve\_system(filename):

    """Считывает матрицу, проверяет условия, решает систему и выводит результаты."""

    n, A, B, eps = read\_matrix\_from\_file(filename)

    # Проверка диагонального преобладания

    check\_conditions(A)

    # Метод простых итераций

    x\_si, iters\_si = simple\_iterations(A, B, eps)

    print("\nРешение методом простых итераций:", x\_si)

    print("Число итераций:", iters\_si)

    print("Проверка A \* x =", multiply\_matrix\_vector(A, x\_si))

    # Метод Зейделя

    x\_seidel, iters\_seidel = seidel\_method(A, B, eps)

    print("\nРешение методом Зейделя:", x\_seidel)

    print("Число итераций:", iters\_seidel)

    print("Проверка A \* x =", multiply\_matrix\_vector(A, x\_seidel))

# Запуск решения

solve\_system("input1\_3.txt")

Вывод программы:  
🔹 Метод простых итераций:

Шаг 1: x = [-0.375, 2.814814814814815, -4.157894736842105, 5.384615384615385], ошибка = 7.371973

Шаг 2: x = [2.1026453241365526, 2.933535762483131, -7.122432148747938, 7.311366021892338], ошибка = 4.318990

Шаг 3: x = [3.3093729694606893, 2.833162559770747, -7.25925794393015, 8.803504350400438], ошибка = 1.926516

Шаг 4: x = [3.900093575407627, 2.27395554193609, -7.434149194880308, 9.16777526988361], ошибка = 0.908264

Шаг 5: x = [4.112444296858397, 2.11355483286278, -7.18936429753473, 9.17481535353476], ошибка = 0.361650

Шаг 6: x = [4.0876535710927575, 1.9922724621451442, -7.079344926140906, 9.121972634180898], ошибка = 0.173841

Шаг 7: x = [4.059607853662558, 1.9769260065735306, -7.016810590414501, 9.042308567193805], ошибка = 0.106203

Шаг 8: x = [4.0205903105523, 1.9823328221547345, -6.991096212689517, 9.013345632212415], ошибка = 0.055242

Шаг 9: x = [4.004992912348348, 1.989820520182513, -6.992440796226482, 8.99748569161934], ошибка = 0.023509

Шаг 10: x = [3.998645557046457, 1.9972094350372491, -6.993868771672776, 8.995876664537338], ошибка = 0.009976

Шаг 11: x = [3.9976644248938604, 1.9994006238473867, -6.99780806522822, 8.9970950552608], ошибка = 0.004771

Шаг 12: x = [3.9985952712735546, 2.0005150978110295, -6.999321980145707, 8.998545001115158], ошибка = 0.002550

Шаг 13: x = [3.9992984472915496, 2.0004346002530884, -7.000053142320262, 8.99960942419896], ошибка = 0.001473

Шаг 14: x = [3.99982617444023, 2.0002584410970012, -7.000207345160646, 8.999963555184033], ошибка = 0.000677

Шаг 15: x = [3.9999993539627035, 2.000108162353318, -7.000133903269437, 9.000093764517528], ошибка = 0.000274

Шаг 16: x = [4.000048465376203, 2.000018982047189, -7.000075287898994, 9.000072302878902], ошибка = 0.000119

Шаг 17: x = [4.000038079725488, 1.9999954701356046, -7.000020621691901, 9.000039587341364], ошибка = 0.000069

Решение методом простых итераций: [4.000038079725488, 1.9999954701356046, -7.000020621691901, 9.000039587341364]

Число итераций: 17

Проверка A \* x = [-8.999533919156947, -76.00017872252724, -79.00034284591504, -70.00032310078205]

🔹 Метод Зейделя:

Шаг 1: x = [-0.375, 2.8981481481481484, -5.457115009746589, 7.643237366921578], ошибка = 9.835602

Шаг 2: x = [3.159220835207677, 2.031191513525822, -6.7616880509950175, 8.698300389341913], ошибка = 4.007169

Шаг 3: x = [3.8445446950419018, 2.030979033394713, -6.9504976748969245, 8.952658997833227], ошибка = 0.754994

Шаг 4: x = [3.971415150554055, 2.002205055883396, -6.991996517465805, 8.990205802617739], ошибка = 0.141619

Шаг 5: x = [3.9948095075690033, 2.0009585288372587, -6.998403421901499, 8.9984031492436], ошибка = 0.025634

Шаг 6: x = [3.999055207213495, 2.0000917494988673, -6.999733266452556, 8.99968302890046], ошибка = 0.004710

Шаг 7: x = [3.99982903445486, 2.000029398203279, -6.999948274800503, 8.999946765633641], ошибка = 0.000848

Шаг 8: x = [3.9999689663957594, 2.0000034002306526, -6.999991154213896, 8.99998971764446], ошибка = 0.000155

Шаг 9: x = [3.999994386466435, 2.0000009114461403, -6.999998318503692, 8.9999982352775], ошибка = 0.000028

Решение методом Зейделя: [3.999994386466435, 2.0000009114461403, -6.999998318503692, 8.9999982352775]

Число итераций: 9

Проверка A \* x = [-9.000110293425553, -75.99999379147987, -78.99994889420175, -69.99999999999999]

## Метод вращения Якоби

Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.

Условие:

2.

Метод вращений Якоби — это итерационный алгоритм, используемый для нахождения собственных значений и собственных векторов симметричных матриц. Он является одним из классических методов для решения задачи спектрального разложения матриц, то есть нахождения собственных значений и собственных векторов матрицы. Этот метод особенно полезен для больших симметричных матриц, где использование прямых методов (например, метод Гаусса) может быть вычислительно сложным.

**Принцип работы**

Метод вращений Якоби основывается на идее, что можно получить диагональную матрицу, проводя последовательные преобразования матрицы с помощью так называемых **поворотных матриц** (или матриц вращения). Эти матрицы выполняют вращение пространства, что позволяет постепенно "выравнивать" матрицу до диагональной формы.

Алгоритм включает в себя следующие шаги:

1. **Поиск максимального элемента в матрице**: На каждом шаге алгоритм ищет максимальный элемент вне главной диагонали. Это позволяет найти тот элемент, который в следующем шаге будет "поворотным" и который приводит к наибольшему изменению в матрице.
2. **Поворотная матрица**: После того как найден максимальный элемент , вычисляется так называемый **угол поворота** θ, который зависит от значений ​ (элементы на главной диагонали) и (элемент вне диагонали). Используя этот угол, строится поворотная матрица, которая выполнит поворот для данной матрицы.
3. **Применение поворота**: Поворотная матрица используется для преобразования исходной матрицы. Она не меняет диагональные элементы, но приводит к нулевым значениям в местах, где ранее находились элементы вне диагонали.
4. **Итерации**: Алгоритм повторяет этот процесс несколько раз. После каждого шага элементы вне главной диагонали уменьшаются, а элементы на главной диагонали становятся всё более точными собственными значениями. Итерации продолжаются до тех пор, пока элементы вне главной диагонали не станут достаточно малыми (меньше заданной точности).
5. **Конвергенция**: В конечном итоге, после достаточного количества итераций, матрица становится диагональной, а элементы на диагонали становятся собственными значениями. Вектор столбцов поворотных матриц, которые были использованы в каждой итерации, представляет собственные векторы исходной матрицы.

Так же в начале программы существует проверка на симметричность матрицы.

Код программы:

import math

def find\_max\_upper\_element(X):

    """

    Находит наибольший внедиагональный элемент в верхнем треугольнике матрицы.

    Возвращает индексы i, j и значение максимального элемента.

    """

    n = len(X)

    i\_max, j\_max = 0, 1

    max\_elem = abs(X[0][1])

    for i in range(n):

        for j in range(i + 1, n):

            if abs(X[i][j]) > max\_elem:

                max\_elem = abs(X[i][j])

                i\_max = i

                j\_max = j

    return i\_max, j\_max, max\_elem

def is\_symmetric(A):

    """

    Проверяет, является ли матрица A симметричной.

    """

    n = len(A)

    for i in range(n):

        for j in range(i + 1, n):

            if A[i][j] != A[j][i]:

                return False

    return True

def rotation\_method(A, eps):

    """

    Метод вращений (Якоби) для нахождения собственных значений и собственных векторов симметричной матрицы.

    """

    if not is\_symmetric(A):

        raise ValueError("Матрица A должна быть симметричной")

    n = len(A)

    A\_i = [row[:] for row in A]  # Копия матрицы A

    eigen\_vectors = [[1.0 if i == j else 0.0 for j in range(n)] for i in range(n)]  # Единичная матрица

    iterations = 0

    while True:

        # Находим наибольший внедиагональный элемент

        i\_max, j\_max, max\_off\_diag = find\_max\_upper\_element(A\_i)

        if max\_off\_diag < eps:

            break

        # Вычисляем угол поворота

        if A\_i[i\_max][i\_max] - A\_i[j\_max][j\_max] == 0:

            phi = math.pi / 4

        else:

            phi = 0.5 \* math.atan(2 \* A\_i[i\_max][j\_max] / (A\_i[i\_max][i\_max] - A\_i[j\_max][j\_max]))

        cos\_phi = math.cos(phi)

        sin\_phi = math.sin(phi)

        # Формируем матрицу поворота

        H = [[1.0 if i == j else 0.0 for j in range(n)] for i in range(n)]

        H[i\_max][i\_max] = cos\_phi

        H[j\_max][j\_max] = cos\_phi

        H[i\_max][j\_max] = -sin\_phi

        H[j\_max][i\_max] = sin\_phi

        print(f"Итерация {iterations}:")

        print("Матрица поворота H:")

        for row in H:

            print(row)

        # Обновляем элементы матрицы A

        for k in range(n):

            if k != i\_max and k != j\_max:

                A\_ik = A\_i[i\_max][k] \* cos\_phi + A\_i[j\_max][k] \* sin\_phi

                A\_jk = -A\_i[i\_max][k] \* sin\_phi + A\_i[j\_max][k] \* cos\_phi

                A\_i[i\_max][k] = A\_ik

                A\_i[j\_max][k] = A\_jk

                A\_i[k][i\_max] = A\_ik

                A\_i[k][j\_max] = A\_jk

        # Обновляем диагональные элементы

        A\_ii = A\_i[i\_max][i\_max] \* cos\_phi\*\*2 + 2 \* A\_i[i\_max][j\_max] \* cos\_phi \* sin\_phi + A\_i[j\_max][j\_max] \* sin\_phi\*\*2

        A\_jj = A\_i[i\_max][i\_max] \* sin\_phi\*\*2 - 2 \* A\_i[i\_max][j\_max] \* cos\_phi \* sin\_phi + A\_i[j\_max][j\_max] \* cos\_phi\*\*2

        A\_ij = 0

        A\_i[i\_max][i\_max] = A\_ii

        A\_i[j\_max][j\_max] = A\_jj

        A\_i[i\_max][j\_max] = A\_ij

        A\_i[j\_max][i\_max] = A\_ij

        # Обновляем собственные векторы

        for k in range(n):

            eig\_ik = eigen\_vectors[k][i\_max] \* cos\_phi + eigen\_vectors[k][j\_max] \* sin\_phi

            eig\_jk = -eigen\_vectors[k][i\_max] \* sin\_phi + eigen\_vectors[k][j\_max] \* cos\_phi

            eigen\_vectors[k][i\_max] = eig\_ik

            eigen\_vectors[k][j\_max] = eig\_jk

        print("Обновленная матрица A:")

        for row in A\_i:

            print(row)

        print("----------------------------------")

        iterations += 1

    eigen\_values = [A\_i[i][i] for i in range(n)]

    return eigen\_values, eigen\_vectors, iterations

def create\_matrix(eigen\_values, eigen\_vectors):

    """

    Восстанавливает матрицу по собственным значениям и собственным векторам.

    """

    n = len(eigen\_values)

    A = [[0.0] \* n for \_ in range(n)]

    for i in range(n):

        for j in range(n):

            for k in range(n):

                A[i][j] += eigen\_vectors[i][k] \* eigen\_values[k] \* eigen\_vectors[j][k]

    return A

def print\_matrix(matrix):

    """

    Выводит матрицу в удобочитаемом формате.

    """

    for row in matrix:

        print(' '.join(f'{val:8.4f}' for val in row))

if \_\_name\_\_ == '\_\_main\_\_':

    # Чтение входных данных из файла

    with open('input1\_4.txt', 'r') as file:

        lines = file.readlines()

    n = int(lines[0].strip())  # Размер матрицы

    A = [list(map(float, lines[i + 1].split())) for i in range(n)]  # Матрица

    eps = float(lines[n + 1].strip())  # Точность

    # Выполняем метод вращений

    eig\_values, eig\_vectors, iters = rotation\_method(A, eps)

    print('Собственные значения:', eig\_values)

    print('Собственные векторы:')

    for row in eig\_vectors:

        print(row)

    print('Число итераций:', iters)

    # Восстанавливаем исходную матрицу по собственным векторам и значениям

    reconstructed\_matrix = create\_matrix(eig\_values, eig\_vectors)

    print("\nВосстановленная матрица из собственных значений и векторов:")

    print\_matrix(reconstructed\_matrix)

Вывод программы:

Итерация 0:

Матрица поворота H:

[1.0, 0.0, 0.0]

[0.0, 0.7071067811865476, -0.7071067811865476]

[0.0, 0.7071067811865476, 0.7071067811865476]

Обновленная матрица A:

[-9.0, 8.485281374238571, -1.414213562373095]

[8.485281374238571, 17.000000000000004, 0]

[-1.414213562373095, 0, -1.0]

----------------------------------

Итерация 1:

Матрица поворота H:

[0.9584893359659982, 0.2851283795756921, 0.0]

[-0.2851283795756921, 0.9584893359659982, 0.0]

[0.0, 0.0, 1.0]

Обновленная матрица A:

[-11.524174696260026, 0, -1.3555086183130964]

[0, 19.524174696260026, -0.40323242141340754]

[-1.3555086183130964, -0.40323242141340754, -1.0]

----------------------------------

Итерация 2:

Матрица поворота H:

[0.9920650923603646, 0.0, -0.12572530580603616]

[0.0, 1.0, 0.0]

[0.12572530580603616, 0.0, 0.9920650923603646]

Обновленная матрица A:

[-11.695959528598655, -0.050696519493109105, 0]

[-0.050696519493109105, 19.524174696260026, -0.4000328093921856]

[0, -0.4000328093921856, -0.8282151676613712]

----------------------------------

Итерация 3:

Матрица поворота H:

[1.0, 0.0, 0.0]

[0.0, 0.9998070390999847, 0.019643944769869214]

[0.0, -0.019643944769869214, 0.9998070390999847]

Обновленная матрица A:

[-11.695959528598655, -0.05068673704708007, -0.0009958796289472331]

[-0.05068673704708007, 19.532034435296183, 0]

[-0.0009958796289472331, 0, -0.8360749066975252]

----------------------------------

Итерация 4:

Матрица поворота H:

[0.9999986827525899, -0.001623112160348037, 0.0]

[0.001623112160348037, 0.9999986827525899, 0.0]

[0.0, 0.0, 1.0]

Обновленная матрица A:

[-11.696041798966293, 0, -0.000995878317127371]

[0, 19.53211670566382, 1.616424335987145e-06]

[-0.000995878317127371, 1.616424335987145e-06, -0.8360749066975252]

----------------------------------

Итерация 5:

Матрица поворота H:

[0.9999999957953917, 0.0, -9.170178090271196e-05]

[0.0, 1.0, 0.0]

[9.170178090271196e-05, 0.0, 0.9999999957953917]

Обновленная матрица A:

[-11.69604189029011, 1.4822899030450484e-10, 0]

[1.4822899030450484e-10, 19.53211670566382, 1.6164243291907139e-06]

[0, 1.6164243291907139e-06, -0.8360748153737095]

----------------------------------

Собственные значения: [-11.69604189029011, 19.53211670566382, -0.8360748153737095]

Собственные векторы:

[0.9513385684751464, 0.2858968116149279, -0.1149693056431899]

[-0.28785659279593123, 0.6913737726511047, -0.6626770619797369]

[-0.10997049656874465, 0.6635249201036814, 0.7400278172378568]

Число итераций: 6

Восстановленная матрица из собственных значений и векторов:

-9.0000 7.0000 5.0000

7.0000 8.0000 9.0000

5.0000 9.0000 8.0000

## QR-разложение

Реализовать алгоритм QR – разложения матриц в виде программы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений. С использованием разработанного программного обеспечения найти собственные значения матрицы.

Условие:

2.

**Основная идея QR-алгоритма:**

QR-алгоритм опирается на две ключевые операции: **QR-разложение** и **итерации** с использованием этих разложений.

1. **QR-разложение**: Это разложение матрицы на произведение двух матриц: Q (ортогональная или унитарная) и R (верхняя треугольная матрица). Ортогональность Q гарантирует, что матрица не изменяет нормы и углы при умножении, что является важным для сохранения стабильности вычислений.
2. **Итерационный процесс**: В каждом шаге QR-алгоритма исходная матрица A заменяется на новую матрицу, которая получается после применения разложения QR и перемножения матриц R и Q в обратном порядке. Этот процесс повторяется, пока матрица не станет практически диагональной. Когда это происходит, элементы на диагонали матрицы становятся собственными значениями исходной матрицы.
3. **Сходимость**: После достаточного количества итераций QR-алгоритм конвергирует к диагональной матрице, где элементы на главной диагонали — это собственные значения исходной матрицы. Собственные векторы можно получить из соответствующих столбцов матриц Q, полученных на каждом шаге разложения.

**Алгоритм QR-метода:**

1. Найдём матрицу Хаусхолдера по формуле:

где v находится по формуле

,

где – евклидова норма вектора, .

1. Перемножим Н на А, в результате чего поддиагональные элементы первого столбца матрицы А станут равными нулю.
2. Продолжаем процесс до того момента, пока матрица А не станет диагональной.
3. Матрица Q представляет из себя произведение матриц Хаусхолдера на каждой итерации. Матрица R представляет из себя матрицу А на последней итерации.
4. Перемножая полученные матрицы в обратном порядке, получаем матрицу А'.
5. Продолжаем процесс, состоящий из пунктов 1-5, до того момента, пока корень из суммы квадратов поддиагональных элементов не станет меньше epsilon.
6. На диагонали финальной матрицы будут находиться собственные значения исходной матрицы. Если после пункта 6 поддиагональные элементы будут недостаточно малы, собственные значения содержат комплексно-сопряжённые корни, смотри п.8.
7. В случае матрицы 3х3 необходимо решить квадратное уравнение

Код программы:

def read\_matrix(filename):

    with open(filename, 'r') as f:

        lines = f.readlines()

    n = int(lines[0])

    A = [list(map(float, line.strip().split())) for line in lines[1:n+1]]

    return A

def transpose(M):

    return [list(row) for row in zip(\*M)]

def matmul(A, B):

    return [[sum(a \* b for a, b in zip(row, col)) for col in zip(\*B)] for row in A]

def identity(n):

    return [[float(i == j) for j in range(n)] for i in range(n)]

def norm\_squared(v):

    return sum(x\*\*2 for x in v)

def householder\_reflection(a):

    n = len(a)

    e1 = [1.0] + [0.0] \* (n - 1)

    sign = 1 if a[0] >= 0 else -1

    norm\_a = norm\_squared(a) \*\* 0.5

    v = [a[i] + sign \* norm\_a \* e1[i] for i in range(n)]

    v\_norm\_sq = norm\_squared(v)

    return v, v\_norm\_sq

def construct\_H(v, v\_norm\_sq, n, k):

    H = identity(n)

    for i in range(k, n):

        for j in range(k, n):

            H[i][j] -= 2 \* v[i - k] \* v[j - k] / v\_norm\_sq

    return H

def qr\_decomposition(A):

    n = len(A)

    R = [row[:] for row in A]

    Q = identity(n)

    for k in range(n):

        x = [R[i][k] for i in range(k, n)]

        v, v\_norm\_sq = householder\_reflection(x)

        H\_k = construct\_H(v, v\_norm\_sq, n, k)

        R = matmul(H\_k, R)

        Q = matmul(Q, transpose(H\_k))  # Q = H1ᵗ \* H2ᵗ \* ... \* Hnᵗ

    return Q, R

def print\_matrix(M, name):

    print(f"\n{name} =")

    for row in M:

        print("  ", " ".join(f"{x:.10f}" for x in row))

def max\_off\_diagonal\_change(old\_diag, new\_diag):

    return max(abs(old\_diag[i] - new\_diag[i]) for i in range(len(old\_diag)))

def qr\_find\_eigenvalues(A, eps=0.01, max\_iter=1000):

    n = len(A)

    Ak = [row[:] for row in A]

    iter\_count = 0

    while iter\_count < max\_iter:

        Q, R = qr\_decomposition(Ak)

        Ak\_next = matmul(R, Q)

        old\_diag = [Ak[i][i] for i in range(n)]

        new\_diag = [Ak\_next[i][i] for i in range(n)]

        if max\_off\_diagonal\_change(old\_diag, new\_diag) < eps:

            break

        Ak = Ak\_next

        iter\_count += 1

    print(f"\nИтераций: {iter\_count}")

    return [Ak[i][i] for i in range(n)]

# === Основной запуск ===

A = read\_matrix('input1\_5.txt')

Q, R = qr\_decomposition(A)

print\_matrix(Q, "Q")

print\_matrix(R, "R")

QR = matmul(Q, R)

print\_matrix(QR, "Q \* R (проверка)")

eigenvalues = qr\_find\_eigenvalues(A, eps=0.01)

print("\nСобственные значения:")

for val in eigenvalues:

    print(val)

Вывод программы:

Q =

-0.6359987280 0.4706034953 0.6115864356

-0.7419985160 -0.5906554074 -0.3171188925

-0.2119995760 0.6554834399 -0.7248431829

R =

9.4339811321 -0.6359987280 6.6779866440

-0.0000000000 -9.3592470647 -0.4537962276

-0.0000000000 0.0000000000 7.2937345281

Q \* R (проверка) =

-6.0000000000 -4.0000000000 -0.0000000000

-7.0000000000 6.0000000000 -7.0000000000

-2.0000000000 -6.0000000000 -7.0000000000

Итераций: 43

Собственные значения:

-11.33454613194149

10.007042670403361

-5.6724965384618695

# Лабораторная работа № 2

## 2.1 Метод простой итерации и метод Ньютона для решения нелинейных уравнений

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.

Условие:

.

Первая производная:

Вторая производная:

Достаточные условия метода Ньютона:

**Алгоритм метода простых итераций:**

1. Составляем уравнение вида x = f(x)
2. Проверяем условие сходимости:

Пусть функция φ(x) определена и дифференцируема на отрезке [a,b]. Тогда если выполняются условия:

1. φ(x) ,
2. ,

То уравнение имеет и притом единственный на [a,b] корень

1. Решение ищется путем построения последовательности: x(k+1) = f(x(k)).

**Алгоритм метода Ньютона:**

1. Выбираем начальное приближение х0;
2. Проверяем условие сходимости:

Пуста на отрезке [a,b] функция f(x) имеет первую и вторую производные постоянного знака и пусть f(a)f(b) ˂ 0.

Тогда если точка выбрана на [a,b] так, что

*,*

то начатая с нее последовательность (k=0,1,2…), определяемая методом Ньютона монотонно сходится к корню

1. Запускаем итерационный процесс по формуле:

Код программы

import math

import matplotlib.pyplot as plt

# Определение функции и её производной

def f(x):

    return math.log(x + 2) - x \*\* 2

def f\_prime(x):

    return 1 / (x + 2) - 2 \* x

def f\_double\_prime(x):

    return -1 / (x + 2) \*\* 2 - 2  # Вторая производная функции

# Проверка на разные знаки на концах отрезка

def check\_signs(a, b):

    f\_a = f(a)

    f\_b = f(b)

    if f\_a \* f\_b > 0:

        print("Значения функции на концах отрезка имеют одинаковый знак!")

        return False

    return True

# Проверка на f(x) \* f''(x) > 0

def condition\_1(x):

    return f(x) \* f\_double\_prime(x) > 0

# Проверка на f(x) \* f''(x) <= (f'(x))^2

def condition\_2(x):

    return abs(f(x) \* f\_double\_prime(x)) <= (f\_prime(x)) \*\* 2

# Явно заданная функция φ(x) и её производная

def phi(x):

    return math.sqrt(math.log(x + 2))

def phi\_prime(x):

    return 1 / (2 \* math.sqrt(math.log(x + 2)) \* (x + 2))

# Метод Ньютона

def newton\_method(a, b, epsilon, max\_iterations=100):

    # Проверка на разные знаки

    if not check\_signs(a, b):

        return None, 0

    # Рассматриваем правую точку (b), если она не удовлетворяет условиям, то рассматриваем левую точку (a)

    if condition\_1(b) and condition\_2(b):

        x0 = b

    elif condition\_1(a) and condition\_2(a):

        x0 = a

    else:

        print(f"Условия сходимости не выполняются ни для {a}, ни для {b}.")

        return None, 0

    iterations = 0

    while iterations < max\_iterations:

        fx = f(x0)

        fpx = f\_prime(x0)

        # Проверка на слишком маленькую производную

        if abs(fpx) < 1e-12:

            print(f"Производная в точке {x0} слишком мала (f'({x0}) = {fpx}), метод Ньютона может не сойтись.")

            return None, iterations

        x1 = x0 - fx / fpx

        if abs(x1 - x0) < epsilon:

            return x1, iterations + 1

        x0 = x1

        iterations += 1

    return x0, iterations

# Метод простых итераций

def simple\_iterations\_manual\_phi(a, b, epsilon, max\_iter=1000):

# Проверка на разные знаки

    if not check\_signs(a, b):

        return None, 0

    # Проверим условие сходимости |φ'(x)| < 1 на концах

    if abs(phi\_prime(a)) >= 1 or abs(phi\_prime(b)) >= 1:

        print("Условие сходимости |φ'(x)| < 1 не выполняется на концах отрезка.")

        return None, 0

    x0 = (a + b) / 2

    iterations = 0

    while iterations < max\_iter:

        try:

            x1 = phi(x0)

        except ValueError:

            print(f"Ошибка вычисления φ({x0})")

            return None, iterations

        if abs(x1 - x0) < epsilon:

            return x1, iterations + 1

        x0 = x1

        iterations += 1

    return x0, iterations

# Построение графика функций

def plot\_functions():

    x\_values = []

    y1\_values = []

    y2\_values = []

    x = -1.9

    while x <= 2:

        x\_values.append(x)

        y1\_values.append(math.log(x + 2))

        y2\_values.append(x \*\* 2)

        x += 0.01

    plt.plot(x\_values, y1\_values, label='ln(x + 2)', color='blue')

    plt.plot(x\_values, y2\_values, label='x²', color='red')

    plt.title('Графики функций ln(x + 2) и x²')

    plt.grid(True)

    plt.legend()

    plt.show()

# Анализ зависимости числа итераций от точности

def plot\_iterations\_vs\_epsilon(a, b):

    epsilons = [10 \*\* (-i) for i in range(1, 15)]

    eps\_labels = [f"1e-{i}" for i in range(1, 15)]

    iterations\_newton = []

    iterations\_simple = []

    for eps in epsilons:

        # Метод Ньютона

        \_, iters\_n = newton\_method(a, b, eps)

        iterations\_newton.append(iters\_n)

        # Метод простых итераций

        \_, iters\_s = simple\_iterations\_manual\_phi(a, b, eps)

        iterations\_simple.append(iters\_s)

    # Строим график

    plt.figure(figsize=(10, 5))

    plt.plot(eps\_labels, iterations\_newton, marker='o', linestyle='-', color='green', label='Метод Ньютона')

    plt.plot(eps\_labels, iterations\_simple, marker='s', linestyle='--', color='blue', label='Метод простых итераций')

    plt.xlabel('Точность ε')

    plt.ylabel('Число итераций')

    plt.title('Зависимость числа итераций от точности ε')

    plt.grid(True)

    plt.legend()

    plt.xticks(rotation=45)

    # Обеспечиваем адекватный масштаб

    ymin = min(iterations\_newton + iterations\_simple)

    ymax = max(iterations\_newton + iterations\_simple)

    plt.yticks(range(ymin, ymax + 1))

    plt.tight\_layout()

    plt.show()

# Основная программа

if \_\_name\_\_ == "\_\_main\_\_":

    plot\_functions()

    try:

        print("Введите интервал, в котором находится x:")

        a, b = map(float, input().split())

        print("Введите точность вычислений ε:")

        epsilon = float(input())

        # Метод Ньютона

        root, iterations = newton\_method(a, b, epsilon)

        if root is not None:

            print("\n[Метод Ньютона]")

            print("Найденный x:", root)

            print("Значение функции:", f"{f(root):.30f}")

            print("Число итераций:", iterations)

        else:

            print("Метод Ньютона не сработал.")

        # Метод простых итераций

        root\_iter, iter\_iter = simple\_iterations\_manual\_phi(a, b, epsilon)

        if root\_iter is not None:

            print("\n[Метод простых итераций]")

            print("Найденный x:", root\_iter)

            print("Значение функции:", f"{f(root\_iter):.30f}")

            print("Число итераций:", iter\_iter)

        else:

            print("Метод простых итераций не сработал.")

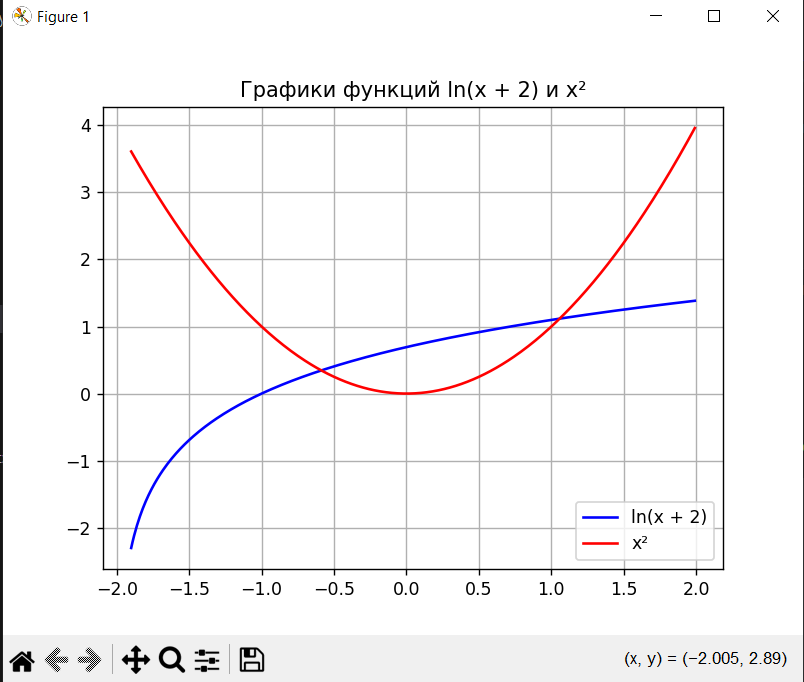
        # График зависимости числа итераций от точности

        plot\_iterations\_vs\_epsilon(a, b)

    except Exception as e:

        print("Ошибка ввода или вычислений:", e)

Вывод программы:

  
Введите интервал, в котором находится x:

1 1.2

Введите точность вычислений ε:

0.00001

[Метод Ньютона]

Найденный x: 1.057103549994738

Значение функции: 0.000000000000000000000000000000

Число итераций: 4

[Метод простых итераций]

Найденный x: 1.0571041314286642

Значение функции: -0.000001039080978149087286510621

Число итераций: 6



## 2.2 Метод простой итерации и метод Ньютона для решения системы нелинейных уравнений

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.

Условие:



 = 3

После подстановки получаем такие уравнения:

Частые производные для метода Ньютона:

φ-функции для метода простых итераций:

Частные производные φ-функций (для сходимости):

**Алгоритм метода простых итераций:**

1. Приводим систему к эквивалентному виду:
2. Проверяем условие сходимости:

Пусто вектор-функция непрерывна, вместе со своей производной

в ограниченной выпуклой замкнутой области G и

1. Запускаем итерационный процесс по формуле:

**Алгоритм метода Ньютона:**

1. Выбираем начальное приближение - вектор х0;
2. Запускаем итерационный процесс по формуле:

где дельта — это приращение, которое находится из формулы:

1. Условие окончания итераций:

Код программы

import matplotlib.pyplot as plt

import math

# Уравнения

def f1(x, y):

    return (x\*\*2 + 9) \* y - 27

def f2(x, y):

    return (x - 1.5)\*\*2 + (y - 1.5)\*\*2 - 9

# Частные производные

def df1\_dx(x, y):

    return 2 \* x \* y

def df1\_dy(x, y):

    return x\*\*2 + 9

def df2\_dx(x, y):

    return 2 \* (x - 1.5)

def df2\_dy(x, y):

    return 2 \* (y - 1.5)

# Решение системы 2x2 линейных уравнений

def solve\_linear\_2x2(a11, a12, a21, a22, b1, b2):

    det = a11 \* a22 - a12 \* a21

    if abs(det) < 1e-12:

        return None, "Якобиан вырожден (определитель близок к нулю)"

    dx = b1 \* a22 - b2 \* a12

    dy = a11 \* b2 - a21 \* b1

    return dx / det, dy / det

# Метод Ньютона

def newton\_system(x0, y0, epsilon=1e-5, max\_iter=100):

    iterations = 0

    while iterations < max\_iter:

        f1\_val = f1(x0, y0)

        f2\_val = f2(x0, y0)

        a11 = df1\_dx(x0, y0)

        a12 = df1\_dy(x0, y0)

        a21 = df2\_dx(x0, y0)

        a22 = df2\_dy(x0, y0)

        result = solve\_linear\_2x2(a11, a12, a21, a22, -f1\_val, -f2\_val)

        if result is None or isinstance(result, str):

            return None, iterations, result

        dx, dy = result

        x1 = x0 + dx

        y1 = y0 + dy

        if abs(dx) < epsilon and abs(dy) < epsilon:

            return (x1, y1), iterations + 1, None

        x0, y0 = x1, y1

        iterations += 1

    return (x0, y0), iterations, None

def phi\_y(x):

    return 27 / (x \*\* 2 + 9)

def phi\_x(y, switch):

    radicand = 9 - (y - 1.5) \*\* 2

    if radicand < 0:

        return None

    return 1.5 + switch \* math.sqrt(radicand)

# Частные производные φ-функций для оценки сходимости

def dphi\_y\_dx(x):

    return -54 \* x / (x\*\*2 + 9)\*\*2

def dphi\_x\_dy(y, switch):

    radicand = 9 - (y - 1.5)\*\*2

    if radicand <= 0:

        return None

    return switch \* (y - 1.5) / math.sqrt(radicand)

def check\_convergence\_condition(x0, y0, switch):

    try:

        dphi\_y = abs(dphi\_y\_dx(x0))

        dphi\_x = abs(dphi\_x\_dy(y0, switch))

        if dphi\_x is None:

            return None

        norm = max(dphi\_x, dphi\_y)

        return norm < 1

    except:

        return None

def simple\_iterations(x0, y0, epsilon, switch=1, max\_iter=100):

    x, y = x0, y0

    for iteration in range(1, max\_iter + 1):

        y\_new = phi\_y(x)

        x\_new = phi\_x(y\_new, switch)

        if x\_new is None:

            return None, iteration, "Выход за область определения φ-функции"

        if abs(x\_new - x) < epsilon and abs(y\_new - y) < epsilon:

            return (x\_new, y\_new), iteration, None

        x, y = x\_new, y\_new

    return (x, y), max\_iter, "Превышено число итераций"

def draw\_graph():

    x\_vals = [i \* 0.05 for i in range(-100, 101)]

    y\_vals = [i \* 0.05 for i in range(-100, 101)]

    X = [[x for x in x\_vals] for \_ in y\_vals]

    Y = [[y for \_ in x\_vals] for y in y\_vals]

    Z1 = [[f1(x, y) for x in x\_vals] for y in y\_vals]

    Z2 = [[f2(x, y) for x in x\_vals] for y in y\_vals]

    plt.figure(figsize=(8, 6))

    plt.contour(x\_vals, y\_vals, Z1, levels=[0], colors='red', linewidths=2)

    plt.contour(x\_vals, y\_vals, Z2, levels=[0], colors='blue', linewidths=2)

    plt.axhline(0, color='red', linewidth=0.5)

    plt.axvline(0, color='blue', linewidth=0.5)

    plt.xlabel("x")

    plt.ylabel("y")

    plt.grid(True)

    plt.title("Графики уравнений")

    plt.legend(["(x²+9)y−27 = 0", "(x−1.5)²+(y−1.5)²−9 = 0"])

    plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    draw\_graph()

    try:

        x0\_str, y0\_str = input("Введите начальные приближения x0 и y0: ").split()

        x0, y0 = float(x0\_str), float(y0\_str)

        epsilon = float(input("Введите точность ε: "))

        # Метод Ньютона

        print("\n[Метод Ньютона]")

        result\_newton, iterations\_newton, error\_newton = newton\_system(x0, y0, epsilon)

        if error\_newton:

            print(f"Ошибка: {error\_newton}")

        else:

            print(f"Решение найдено за {iterations\_newton} итераций:")

            print(f"x = {result\_newton[0]}")

            print(f"y = {result\_newton[1]}")

            print(f"Проверка: f1 = {f1(result\_newton[0], result\_newton[1])}, f2 = {f2(result\_newton[0], result\_newton[1])}")

        # Метод простых итераций

        print("\n[Метод простых итераций]")

        # Определяем, какую φ-функцию использовать

        switch = -1 if x0 < 0 else 1

        # Проверка достаточного условия сходимости

        conv = check\_convergence\_condition(x0, y0, switch)

        if conv is None:

            print("Предупреждение: невозможно вычислить норму φ' в точке — вне области определения.")

        elif not conv:

            print("Предупреждение: достаточное условие сходимости ||φ'(x₀)|| < 1 не выполнено.")

        else:

            print("OK: достаточное условие сходимости выполнено (||φ'(x₀)|| < 1).")

        result\_iter, iterations\_iter, error\_iter = simple\_iterations(x0, y0, epsilon, switch)

        if error\_iter:

            print(f"Ошибка: {error\_iter}")

        else:

            print(f"Решение найдено за {iterations\_iter} итераций:")

            print(f"x = {result\_iter[0]}")

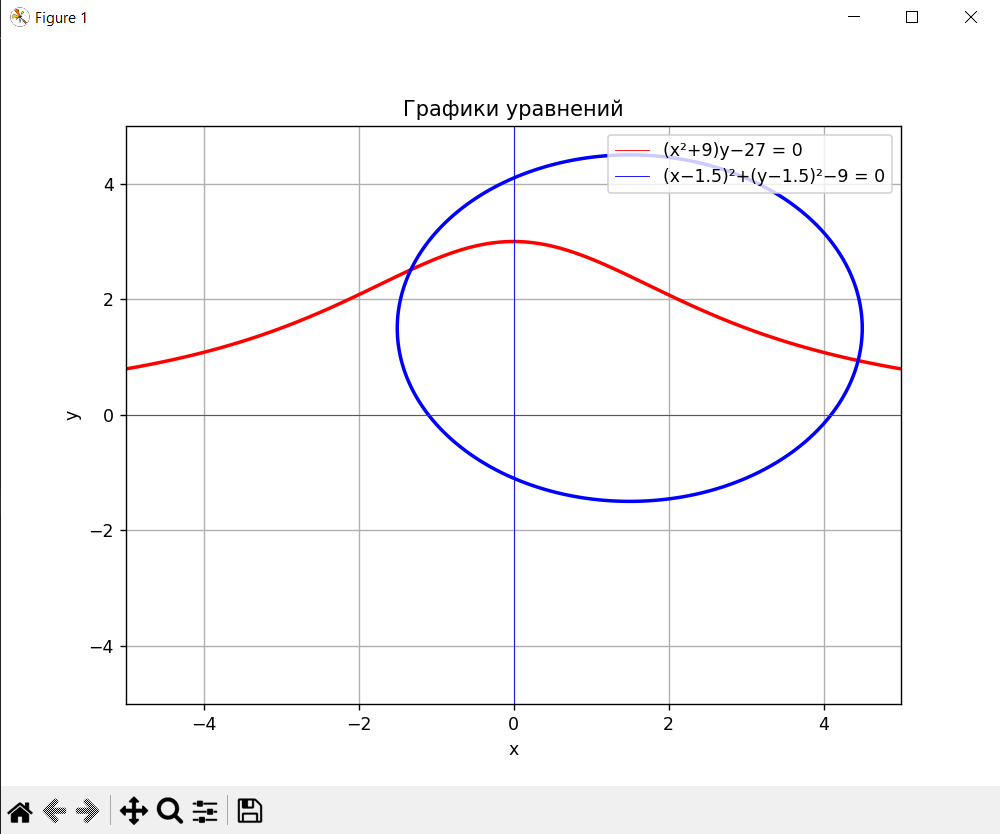
            print(f"y = {result\_iter[1]}")

            print(f"Проверка: f1 = {f1(result\_iter[0], result\_iter[1])}, f2 = {f2(result\_iter[0], result\_iter[1])}")

    except Exception as e:

        print(f"\nОшибка: {e}")

Вывод программы:



Введите начальные приближения x0 и y0: 4 2

Введите точность ε: 0.000000001

[Метод Ньютона]

Решение найдено за 6 итераций:

x = 4.446947067680284

y = 0.9383034802577345

Проверка: f1 = -3.552713678800501e-15, f2 = -1.7763568394002505e-15

[Метод простых итераций]

OK: достаточное условие сходимости выполнено (||φ'(x₀)|| < 1).

Решение найдено за 9 итераций:

x = 4.446947067682337

y = 0.9383034802685014

Проверка: f1 = 3.269526871463313e-10, f2 = 1.7763568394002505e-15

# Лабораторная работа № 3

## 3.1 Построить интерполяционные многочлены Лагранжа и Ньютона

Используя таблицу значений  функции , вычисленных в точках  построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки . Вычислить значение погрешности интерполяции в точке .

Условие:

, a)  ; б) ; .

**Интерполяция** — это метод приближённого представления функции, при котором по заданному набору точек

строится функция f(x), проходящая через все эти точки:

*Цель интерполяции* — восстановить или аппроксимировать (**аппроксимация** - приближённое представление одной функции с помощью другой, более простой функции, которая не обязательно проходит через все заданные точки, но **наилучшим образом** приближает их в некотором смысле) функцию между известными точками на основе ограниченного набора данных.

**Алгоритм решения поставленной задачи**

Пусть на отрезке [a,b] задано дискретное множество несовпадающих точек , которые будем называть узлами и в которых известны значения функции Потребуем, чтобы приближающая функция φ(x,a) совпадала с приближаемой f(x) в (n+1) узле таблицы, т.е. потребуем выполнения равенства

Этот способ построения приближающей функции, при котором в узлах значения приближаемой и приближающей функции совпадают, называется интерполяцией, или лагранжевой интерполяцией. Наиболее распространен способ линейной интерполяции, в случае которой приближающая функция ищется в виде линейной комбинации некоторых базисных функций

Система функций должна быть линейно зависимой, так как в противном случае число параметров и членов в сумме можно было бы уменьшить и, кроме того,

**Алгоритм построения интерполяционного многочлена Лагранжа:**

* 1. Заполняем вектор точек и вектор значений функции в этих точках
  2. Введем функцию:
  3. Запишем выражение интерполяционного многочлена Лагранжа в виде:

**Алгоритм построения интерполяционного многочлена Ньютона:**

1. Введем понятие разделенной разности. Разделенные разности нулевого порядка совпадают со значениями функции в узлах. Разделенные разности первого порядка обозначаются и определяются через разделенные разности нулевого порядка:
2. разделенные разности второго порядка определяются через разделенные разности первого порядка:
3. Разделенная разность порядка n - k + 2 определяется соотношениями
4. Интерполяционный многочлен, значения которого в узлах интерполяции совпадают со значениями функции f(x) запишем в виде:

Код программы:

import math

pi = math.pi

points\_A = [0, pi/6, 2\*pi/6, 3\*pi/6]

points\_B = [0, pi/6, 5\*pi/12, pi/2]

x\_star = pi/4

def f(x):

    return math.cos(x)

def get\_y(points):

    return [f(x) for x in points]

def lagrange\_polynomial(X, Y, x\_star):

    n = len(X)

    def L\_i(i, x):

        res = 1.0

        for j in range(n):

            if j != i:

                res \*= (x - X[j]) / (X[i] - X[j])

        return res

    value = 0.0

    for i in range(n):

        value += Y[i] \* L\_i(i, x\_star)

    return value

def lagrange\_poly\_str(X, Y):

    n = len(X)

    terms = []

    for i in range(n):

        coef = Y[i]

        factors = ''

        for j in range(n):

            if j != i:

                factors += f"(x-{X[j]:.2f})"

        sign = '+' if coef >= 0 else '-'

        terms.append(f" {sign} {abs(coef):.2f}\*{factors}")

    return "L(x) =" + "".join(terms)

def divided\_differences(X, Y):

    n = len(X)

    dd\_table = [y for y in Y]

    coeffs = [dd\_table[0]]

    for level in range(1, n):

        new\_dd = []

        for i in range(n - level):

            val = (dd\_table[i+1] - dd\_table[i]) / (X[i+level] - X[i])

            new\_dd.append(val)

        coeffs.append(new\_dd[0])

        dd\_table = new\_dd

    return coeffs

def newton\_poly\_value(X, coeffs, x):

    n = len(coeffs)

    res = coeffs[0]

    prod = 1.0

    for i in range(1, n):

        prod \*= (x - X[i-1])

        res += coeffs[i] \* prod

    return res

def newton\_poly\_str(X, coeffs):

    n = len(coeffs)

    terms = [f"{coeffs[0]:.2f}"]

    for i in range(1, n):

        factors = ''.join([f"(x-{X[j]:.2f})" for j in range(i)])

        sign = '+' if coeffs[i] >= 0 else '-'

        terms.append(f" {sign} {abs(coeffs[i]):.2f}\*{factors}")

    return "P(x) = " + " +".join(terms)

def abs\_error(true\_val, approx\_val):

    return abs(true\_val - approx\_val)

def process\_points(name, X):

    Y = get\_y(X)

    true\_val = f(x\_star)

    print(f"Точки {name}")

    # Лагранж

    L\_val = lagrange\_polynomial(X, Y, x\_star)

    print("Полином:")

    print(lagrange\_poly\_str(X, Y))

    print(f"Значение полинома в точке X\* = {L\_val}")

    print(f"Значение функции в точке X\* = {true\_val}")

    print(f"Абсолютная погрешность в точке = {abs\_error(true\_val, L\_val)}\n")

    # Ньютона

    coeffs = divided\_differences(X, Y)

    N\_val = newton\_poly\_value(X, coeffs, x\_star)

    print("Полином:")

    print(newton\_poly\_str(X, coeffs))

    print(f"Значение полинома в точке X\* = {N\_val}")

    print(f"Значение функции в точке X\* = {true\_val}")

    print(f"Абсолютная погрешность в точке = {abs\_error(true\_val, N\_val)}\n")

process\_points("A", points\_A)

process\_points("B", points\_B)

Вывод программы:  
Точки A

Полином:

L(x) = + 1.00\*(x-0.52)(x-1.05)(x-1.57) + 0.87\*(x-0.00)(x-1.05)(x-1.57) + 0.50\*(x-0.00)(x-0.52)(x-1.57) + 0.00\*(x-0.00)(x-0.52)(x-1.05)

Значение полинома в точке X\* = 0.7058892896287468

Значение функции в точке X\* = 0.7071067811865476

Абсолютная погрешность в точке = 0.0012174915578008205

Полином:

P(x) = 1.00 + - 0.26\*(x-0.00) + - 0.42\*(x-0.00)(x-0.52) + + 0.11\*(x-0.00)(x-0.52)(x-1.05)

Значение полинома в точке X\* = 0.7058892896287468

Значение функции в точке X\* = 0.7071067811865476

Абсолютная погрешность в точке = 0.0012174915578008205

Точки B

Полином:

L(x) = + 1.00\*(x-0.52)(x-1.31)(x-1.57) + 0.87\*(x-0.00)(x-1.31)(x-1.57) + 0.26\*(x-0.00)(x-0.52)(x-1.57) + 0.00\*(x-0.00)(x-0.52)(x-1.31)

Значение полинома в точке X\* = 0.7048104798998414

Значение функции в точке X\* = 0.7071067811865476

Абсолютная погрешность в точке = 0.002296301286706215

Полином:

P(x) = 1.00 + - 0.26\*(x-0.00) + - 0.40\*(x-0.00)(x-0.52) + + 0.12\*(x-0.00)(x-0.52)(x-1.31)

Значение полинома в точке X\* = 0.7048104798998415

Значение функции в точке X\* = 0.7071067811865476

Абсолютная погрешность в точке = 0.002296301286706104

## 3.2 Построить кубический сплайн

Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при  и . Вычислить значение функции в точке .

Условия:

1.5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | 0.0 | 1.0 | 2.0 | 3.0 | 4.0 |
|  | 1.0 | 0.86603 | 0.5 | 0.0 | -0.5 |

**Сплайн** — это кусочно-гладкая функция, используемая для интерполяции. Обычно это совокупность многочленов невысокой степени, определённых на каждом из отрезков между узлами​, причём они «гладко» стыкуются между собой.

Наиболее распространённый вариант — **кубический сплайн**, то есть интерполяционная функция S(x), которая на каждом промежутке определяется кубическим многочленом:

Сплайн удовлетворяет следующим условиям:

* 1. Проходит через заданные точки:
  2. Обеспечивает непрерывность первой производной:
  3. Обеспечивает непрерывность второй производной:

Наиболее широко применяемым является случай, когда между любыми двумя точками разбиения исходного отрезка строится многочлен n-й степени:

который в узлах интерполяции принимает значения аппроксимируемой функции и непрерывен вместе со своими (n-1) производными. Такой кусочно-непрерывный интерполяционный многочлен называется сплайном. Его коэффициенты находятся из условий равенства в узлах сетки значений сплайна и приближаемой функции, а также равенства n-1 производных соответствующих многочленов. На практике наиболее часто используется интерполяционный многочлен третьей степени, который удобно представить как:

,

Алгоритм построения кубического сплайна:

* 1. Составляем трехдиагональную матрицу вида:
  2. Найдет коэффициенты решив систему уравнений из пункта 1.
  3. Найдем коэффицианты , , по формулам:

4. Вычисляем значение функции в выбранной точке и строим график

Код программы:

import matplotlib.pyplot as plt

x = [0, 1, 2, 3, 4]

f = [1.00000, 0.86603, 0.50000, 0.00000, -0.50000]

n = len(x) - 1

h = x[1] - x[0]

X\_star = 1.5

a = [0]  # нижняя диагональ

b = [1]  # главная диагональ

c = [0]  # верхняя диагональ

d = [0]  # правая часть

for i in range(1, n):

    a.append(1)

    b.append(4)

    c.append(1)

    d.append(3 \* (f[i + 1] - 2 \* f[i] + f[i - 1]))

# Метод прогонки для нахождения c

c\_vals = [0] \* (n + 1)  # с0 и сn = 0

# Прямой ход

for i in range(1, n):

    w = a[i] / b[i - 1]

    b[i] = b[i] - w \* c[i - 1]

    d[i] = d[i] - w \* d[i - 1]

# Обратный ход

c\_vals[n - 1] = d[n - 1] / b[n - 1]

for i in range(n - 2, 0, -1):

    c\_vals[i] = (d[i] - c[i] \* c\_vals[i + 1]) / b[i]

# c0 и cn по краям равны нулю (условие натурального сплайна)

c\_vals[0] = 0

c\_vals[n] = 0

# Вычисляем коэффициенты a, b, d

a\_coeff = []

b\_coeff = []

d\_coeff = []

for i in range(1, n + 1):

    ai = f[i - 1]

    ci = c\_vals[i]

    ci\_prev = c\_vals[i - 1]

    di = (ci - ci\_prev) / (3 \* h)

    bi = (f[i] - f[i - 1]) / h - (h / 3) \* (2 \* ci\_prev + ci)

    a\_coeff.append(ai)

    b\_coeff.append(bi)

    d\_coeff.append(di)

# Найдём нужный интервал

for i in range(1, n + 1):

    if x[i - 1] <= X\_star <= x[i]:

        dx = X\_star - x[i - 1]

        S = (a\_coeff[i - 1] +

             b\_coeff[i - 1] \* dx +

             c\_vals[i - 1] \* dx \*\* 2 +

             d\_coeff[i - 1] \* dx \*\* 3)

        print(f"S({X\_star}) = {S}")

        break

# Построение графика

points\_x = []

points\_y = []

for i in range(1, n + 1):

    xi = x[i - 1]

    xf = x[i]

    a\_i = a\_coeff[i - 1]

    b\_i = b\_coeff[i - 1]

    c\_i = c\_vals[i - 1]

    d\_i = d\_coeff[i - 1]

    step = 0.01

    t = xi

    while t <= xf:

        dx = t - xi

        y = a\_i + b\_i \* dx + c\_i \* dx \*\* 2 + d\_i \* dx \*\* 3

        points\_x.append(t)

        points\_y.append(y)

        t += step

plt.figure(figsize=(8, 5))

plt.plot(points\_x, points\_y, label='Cubic spline', color='blue')

plt.scatter(x, f, color='red', label='Data points')

plt.scatter([X\_star], [S], color='green', s=80, zorder=5, label=f'S({X\_star}) = {S:.5f}')

plt.title('Natural Cubic Spline Interpolation')

plt.xlabel('x')

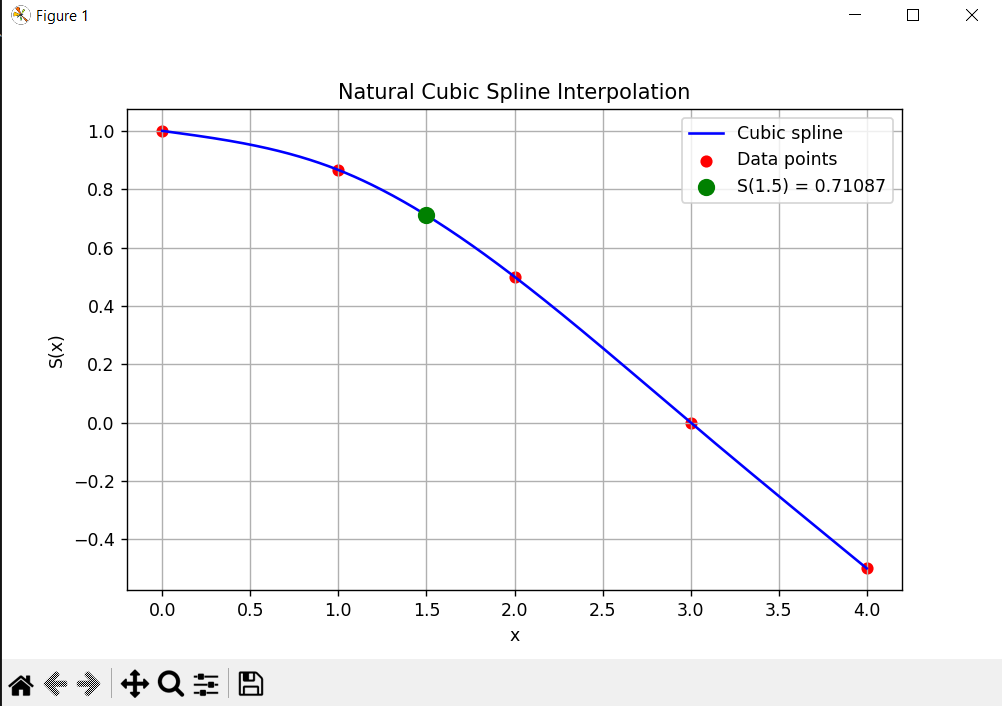
plt.ylabel('S(x)')

plt.legend()

plt.grid(True)

plt.show()

Вывод программы:



S(1.5) = 0.7108741517857143

## 3.3 Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены

Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены a) 1-ой и б) 2-ой степени. Для каждого из приближающих многочленов вычислить сумму квадратов ошибок. Построить графики приближаемой функции и приближающих многочленов.

Условие:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
|  | -1.0 | 0.0 | 1.0 | 2.0 | 3.0 | 4.0 |
|  | 0.86603 | 1.0 | 0.86603 | 0.50 | 0.0 | -0.50 |

Алгоритм нахождения приближающих многочленов:

* + 1. Составляем матрицу следующей системы:

Для 1-ой степени:

Для 2-ой степени:

* + 1. Решаем систему методом Крамера

Для многочлена 1-ой степени:

Для многочлена 2-ой степени:

(аналогично для b и c)

* + 1. Рассчитываем сумму квадратов ошибок

Для многочлена 1-ой степени:

Для многочлена 2-ой степени:

Код программы:

import matplotlib.pyplot as plt

# Исходные данные

x\_vals = [-1, 0, 1, 2, 3, 4]

y\_vals = [0.86603, 1, 0.86603, 0.5, 0, -0.5]

n = len(x\_vals)

# МНК для многочлена первой степени: y = a + b\*x

def least\_squares\_linear(x, y):

    sum\_x = sum(x)

    sum\_y = sum(y)

    sum\_x2 = sum(xi\*\*2 for xi in x)

    sum\_xy = sum(x[i] \* y[i] for i in range(n))

    A = [[n, sum\_x],

         [sum\_x, sum\_x2]]

    B = [sum\_y, sum\_xy]

    # Решим систему методом Крамера

    detA = A[0][0] \* A[1][1] - A[0][1] \* A[1][0]

    detA1 = B[0] \* A[1][1] - A[0][1] \* B[1]

    detA2 = A[0][0] \* B[1] - B[0] \* A[1][0]

    a = detA1 / detA

    b = detA2 / detA

    # Сумма квадратов ошибок

    error = sum((y[i] - (a + b \* x[i]))\*\*2 for i in range(n))

    return a, b, error

# МНК для многочлена второй степени: y = a + b\*x + c\*x^2

def least\_squares\_quadratic(x, y):

    sum\_x = sum(x)

    sum\_x2 = sum(xi\*\*2 for xi in x)

    sum\_x3 = sum(xi\*\*3 for xi in x)

    sum\_x4 = sum(xi\*\*4 for xi in x)

    sum\_y = sum(y)

    sum\_xy = sum(x[i] \* y[i] for i in range(n))

    sum\_x2y = sum((x[i]\*\*2) \* y[i] for i in range(n))

    A = [

        [n, sum\_x, sum\_x2],

        [sum\_x, sum\_x2, sum\_x3],

        [sum\_x2, sum\_x3, sum\_x4]

    ]

    B = [sum\_y, sum\_xy, sum\_x2y]

    # Решим систему методом Крамера

    def determinant(m):

        return (

            m[0][0] \* (m[1][1] \* m[2][2] - m[1][2] \* m[2][1])

            - m[0][1] \* (m[1][0] \* m[2][2] - m[1][2] \* m[2][0])

            + m[0][2] \* (m[1][0] \* m[2][1] - m[1][1] \* m[2][0])

        )

    def replace\_column(matrix, col\_idx, new\_col):

        return [

            [new\_col[i] if j == col\_idx else matrix[i][j] for j in range(3)]

            for i in range(3)

        ]

    detA = determinant(A)

    detA0 = determinant(replace\_column(A, 0, B))

    detA1 = determinant(replace\_column(A, 1, B))

    detA2 = determinant(replace\_column(A, 2, B))

    a = detA0 / detA

    b = detA1 / detA

    c = detA2 / detA

    # Сумма квадратов ошибок

    error = sum((y[i] - (a + b \* x[i] + c \* x[i]\*\*2))\*\*2 for i in range(n))

    return a, b, c, error

# Расчет коэффициентов и ошибок

a1, b1, err1 = least\_squares\_linear(x\_vals, y\_vals)

a2, b2, c2, err2 = least\_squares\_quadratic(x\_vals, y\_vals)

# Вывод результатов

print(f"Многочлен 1-й степени: f(x) = {a1:.5f} + {b1:.5f} \* x")

print(f"Многочлен 2-й степени: f(x) = {a2:.5f} + {b2:.5f} \* x + {c2:.5f} \* x^2\n")

print(f"Сумма квадратов ошибок (1-й степени): {err1:.5f}")

print(f"Сумма квадратов ошибок (2-й степени): {err2:.5f}")

# Создание значений x для построения графика

x\_min = min(x\_vals) - 1

x\_max = max(x\_vals) + 1

x\_plot = [x\_min + i \* (x\_max - x\_min) / 300 for i in range(301)]

# Вычисление значений многочленов

y\_lin = [a1 + b1 \* x for x in x\_plot]

y\_quad = [a2 + b2 \* x + c2 \* x\*\*2 for x in x\_plot]

# Построение графика

plt.plot(x\_plot, y\_lin, label='1-я степень', color='blue')

plt.plot(x\_plot, y\_quad, label='2-я степень', color='green')

plt.scatter(x\_vals, y\_vals, label='Точки', color='red')

plt.legend()

plt.title("Аппроксимация МНК")

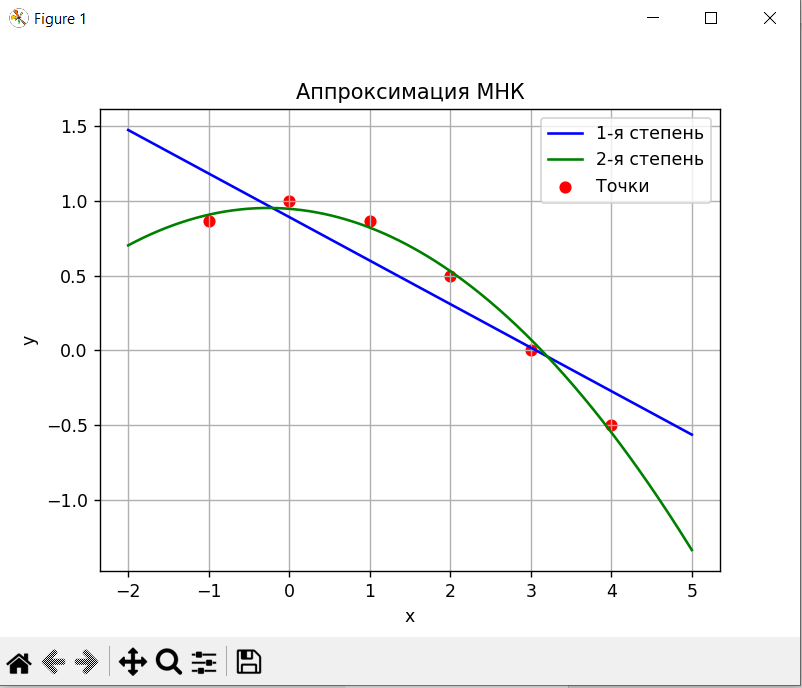
plt.grid(True)

plt.xlabel("x")

plt.ylabel("y")

plt.show()

Вывод программы:



Многочлен 1-й степени: f(x) = 0.89232 + -0.29132 \* x

Многочлен 2-й степени: f(x) = 0.94749 + -0.04307 \* x + -0.08275 \* x^2

Сумма квадратов ошибок (1-й степени): 0.27082

Сумма квадратов ошибок (2-й степени): 0.01518

## 3.4 Вычислить первую и вторую производную от таблично заданной функции

Вычислить первую и вторую производную от таблично заданной функции в точке .

Условие:

 1.0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | -1.0 | 0.0 | 1.0 | 2.0 | 3.0 |
|  | -0.5 | 0.0 | 0.5 | 0.86603 | 1.0 |

Алгоритм действий:

* + - 1. Вычисляем правостороннюю производную
      2. Вычисляем левостороннюю производную

3. Вычисляем значение производной второго порядка точности

4. Вычисляем значение производной второго порядка

Код программы:

def get\_i(x, x0):

    for i in range(len(x)):

        if x[i] == x0:

            return i

    return -1

def first\_der(x, y, id\_x, x0):

    if x0 in x and x0 != x[0] and x0 != x[-1]:

        print("\nТочка совпадает с правой границей отрезка")

        dxLeft = (y[id\_x] - y[id\_x - 1]) / (x[id\_x] - x[id\_x - 1])

        print("Значение левосторонней производной:", dxLeft)

        dxRight = (y[id\_x + 1] - y[id\_x]) / (x[id\_x + 1] - x[id\_x])

        print("Значение правосторонней производной:", dxRight)

        dxSecondAccuracy = (dxLeft + dxRight) / 2

        print("Значение производной второго порядка точности:", dxSecondAccuracy)

    else:

        print("\nТочка находится внутри отрезка. Найдем производную с первым порядком точности:")

        dx = (y[id\_x] - y[id\_x - 1]) / (x[id\_x] - x[id\_x - 1])

        print("Значение производной:", dx)

        if id\_x < len(x) - 1:

            dxLeft = (y[id\_x] - y[id\_x - 1]) / (x[id\_x] - x[id\_x - 1])

            dxRight = (y[id\_x + 1] - y[id\_x]) / (x[id\_x + 1] - x[id\_x])

            dxSecondAccuracy = dxLeft + (dxRight - dxLeft) / (x[id\_x + 1] - x[id\_x - 1]) \* (2 \* x0 - x[id\_x - 1] - x[id\_x])

            print("Значение производной второго порядка точности:", dxSecondAccuracy)

def second\_der(x, y, id\_x):

    if 0 < id\_x < len(x) - 1:

        dxdx = (2 \* ((y[id\_x + 1] - y[id\_x]) / (x[id\_x + 1] - x[id\_x]) -

                     (y[id\_x] - y[id\_x - 1]) / (x[id\_x] - x[id\_x - 1])) /

                (x[id\_x + 1] - x[id\_x - 1]))

        print("Значение второй производной:", dxdx)

    else:

        print("Невозможно вычислить производную второго порядка")

# Данные

x = [-1, 0, 1, 2, 3]

y = [-0.5, 0, 0.5, 0.86603, 1]

x0 = 1

id\_x = get\_i(x, x0)

if id\_x == -1:

    print("Точка не найдена в массиве x")

else:

    first\_der(x, y, id\_x, x0)

    second\_der(x, y, id\_x)

Вывод программы:

Точка совпадает с правой границей отрезка

Значение левосторонней производной: 0.5

Значение правосторонней производной: 0.36602999999999997

Значение производной второго порядка точности: 0.433015

Значение второй производной: -0.13397000000000003

## 3.5 Вычислить определенный интеграл методами прямоугольников, трапеций, Симпсона

Вычислить определенный интеграл , методами прямоугольников, трапеций, Симпсона с шагами . Оценить погрешность вычислений, используя Ме­тод Рунге-Ромберга:

Условие:

, ;

Алгоритм:

* + - * 1. Вычисляем интеграл методом прямоугольников
        2. Вычисляем интеграл методом трапеций

3. Вычисляем интеграл методом Симпсона

4. Оцениваем погрешность методом Рунге-Ромберга

Код программы:

def f(x):

    return x / (3 \* x + 4)\*\*2

def rectangle\_method(a, b, h):

    n = int((b - a) / h)

    result = 0

    for i in range(n):

        xi = a + i \* h

        xi1 = xi + h

        midpoint = (xi + xi1) / 2

        result += h \* f(midpoint)

    return result

def trapezoid\_method(a, b, h):

    n = int((b - a) / h)

    result = 0

    for i in range(1, n):

        x = a + i \* h

        result += f(x)

    result += (f(a) + f(b)) / 2

    return result \* h

def simpson\_method(a, b, h):

    n = int((b - a) / h)

    if n % 2 == 1:

        n += 1

    h = (b - a) / n

    result = f(a) + f(b)

    for i in range(1, n):

        x = a + i \* h

        coef = 4 if i % 2 == 1 else 2

        result += coef \* f(x)

    return result \* h / 3

def runge\_romberg(Fh, Fkh, k, p):

    return Fh + (Fh - Fkh) / (k\*\*p - 1)

# Параметры задачи

a = 0

b = 4

h1 = 1

h2 = 0.5

k = int(h1 / h2)

# Прямоугольники

R1 = rectangle\_method(a, b, h1)

R2 = rectangle\_method(a, b, h2)

R\_rr = runge\_romberg(R2, R1, k, 2)

R\_err = abs(R\_rr - R2)

# Трапеции

T1 = trapezoid\_method(a, b, h1)

T2 = trapezoid\_method(a, b, h2)

T\_rr = runge\_romberg(T2, T1, k, 2)

T\_err = abs(T\_rr - T2)

# Симпсон

S1 = simpson\_method(a, b, h1)

S2 = simpson\_method(a, b, h2)

S\_rr = runge\_romberg(S2, S1, k, 4)

S\_err = abs(S\_rr - S2)

# Результаты

print("Прямоугольники:")

print(f" h1={h1}: {R1}, h2={h2}: {R2}, уточнённый: {R\_rr}, ошибка: {R\_err}")

print("\nТрапеции:")

print(f" h1={h1}: {T1}, h2={h2}: {T2}, уточнённый: {T\_rr}, ошибка: {T\_err}")

print("\nСимпсон:")

print(f" h1={h1}: {S1}, h2={h2}: {S2}, уточнённый: {S\_rr}, ошибка: {S\_err}")

Вывод программы:

Прямоугольники:

h1=1: 0.07284061196629331, h2=0.5: 0.0713276669809035, уточнённый: 0.07082335198577355, ошибка: 0.0005043149951299425

Трапеции:

h1=1: 0.06597214255524694, h2=0.5: 0.06940637726077012, уточнённый: 0.07055112216261118, ошибка: 0.0011447449018410633

Симпсон:

h1=1: 0.0694211900736626, h2=0.5: 0.07055112216261118, уточнённый: 0.07062645096854109, ошибка: 7.532880592990565e-05

# Лабораторная работа № 4

## 4.1 Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка

Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки . С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

Условие:

|  |  |
| --- | --- |
| Задача Коши | Точное решение |
| , |  |

**Точное решение:**

**Дифференциальное уравнение второго порядка:**

Пусть

Тогда система будет выглядеть так:

**Метод Эйлера:**

Метод Эйлера — простейший численный метод решения систем обыкновенных дифференциальных уравнений. Метод Эйлера является явным, одношаговым методом первого порядка точности. Он основан на аппроксимации интегральной кривой кусочно-линейной функцией — так называемой ломаной Эйлера.

Переход к произвольным индексам дает формулу метода Эйлера:

**Метод Рунге-Кутты четвертого порядка:**

Промежуточные коэффициенты:

Формула обновления:

**Метод Адамса:**

При использовании интерполяционного многочлена 3-ей степени построенного по значениям подынтегральной функции в последних четырех узлах получим метод Адамса четвертого порядка точности:

где значение подынтегральной функции в узле

Метод Адамса как и все многошаговые методы не являются самостартующими, то есть для того, чтобы использовать метод Адамса необходимо иметь решения в первых четырех узлах. В узле решение известно из начальных условий, а в других трех узлах решения можно получить с помощью подходящего одношагового метода, например: метода Рунге-Кутты четвертого порядка.

**Абсолютная погрешность**:

**Погрешность Рунге-Ромберта:**

Код программы:

import math

import matplotlib.pyplot as plt

# Точное решение

def y\_exact(x):

    return x \* math.sin(x) + math.cos(x)

# Правая часть: y'' = 2cos(x) - y

def f(x, y, y1):

    return y1

def g(x, y, y1):

    return 2 \* math.cos(x) - y

# Метод Эйлера

def euler\_method(x0, xn, h, y0, y1\_0):

    X = []

    Y = []

    x = x0

    y = y0

    y1 = y1\_0

    while x <= xn + 1e-10:

        X.append(x)

        Y.append(y)

        y1\_new = y1 + h \* g(x, y, y1)

        y\_new = y + h \* f(x, y, y1)

        y = y\_new

        y1 = y1\_new

        x += h

    return X, Y

# Метод Рунге-Кутты 4-го порядка

def runge\_kutta\_method(x0, xn, h, y0, y1\_0):

    X = []

    Y = []

    x = x0

    y = y0

    y1 = y1\_0

    while x <= xn + 1e-10:

        X.append(x)

        Y.append(y)

        K1 = h \* f(x, y, y1)

        L1 = h \* g(x, y, y1)

        K2 = h \* f(x + h/2, y + K1/2, y1 + L1/2)

        L2 = h \* g(x + h/2, y + K1/2, y1 + L1/2)

        K3 = h \* f(x + h/2, y + K2/2, y1 + L2/2)

        L3 = h \* g(x + h/2, y + K2/2, y1 + L2/2)

        K4 = h \* f(x + h, y + K3, y1 + L3)

        L4 = h \* g(x + h, y + K3, y1 + L3)

        y += (K1 + 2\*K2 + 2\*K3 + K4) / 6

        y1 += (L1 + 2\*L2 + 2\*L3 + L4) / 6

        x += h

    return X, Y

# Метод Адамса 4-го порядка (предиктор-корректор)

def adams\_method(x0, xn, h, y0, y1\_0):

    X = []

    Y = []

    # Сначала получим первые 4 точки методом Рунге-Кутты

    x = x0

    y = y0

    y1 = y1\_0

    x\_vals = [x]

    y\_vals = [y]

    y1\_vals = [y1]

    for \_ in range(3):  # нужно 3 дополнительных шага

        K1 = h \* f(x, y, y1)

        L1 = h \* g(x, y, y1)

        K2 = h \* f(x + h/2, y + K1/2, y1 + L1/2)

        L2 = h \* g(x + h/2, y + K1/2, y1 + L1/2)

        K3 = h \* f(x + h/2, y + K2/2, y1 + L2/2)

        L3 = h \* g(x + h/2, y + K2/2, y1 + L2/2)

        K4 = h \* f(x + h, y + K3, y1 + L3)

        L4 = h \* g(x + h, y + K3, y1 + L3)

        y += (K1 + 2\*K2 + 2\*K3 + K4) / 6

        y1 += (L1 + 2\*L2 + 2\*L3 + L4) / 6

        x += h

        x\_vals.append(x)

        y\_vals.append(y)

        y1\_vals.append(y1)

    X.extend(x\_vals)

    Y.extend(y\_vals)

    # Применяем формулы Адамса

    while x + h <= xn + 1e-10:

        f\_vals = [f(x\_vals[i], y\_vals[i], y1\_vals[i]) for i in range(-1, -5, -1)]

        g\_vals = [g(x\_vals[i], y\_vals[i], y1\_vals[i]) for i in range(-1, -5, -1)]

        y\_next = y\_vals[-1] + h / 24 \* (55 \* f\_vals[0] - 59 \* f\_vals[1] + 37 \* f\_vals[2] - 9 \* f\_vals[3])

        y1\_next = y1\_vals[-1] + h / 24 \* (55 \* g\_vals[0] - 59 \* g\_vals[1] + 37 \* g\_vals[2] - 9 \* g\_vals[3])

        x += h

        x\_vals.append(x)

        y\_vals.append(y\_next)

        y1\_vals.append(y1\_next)

        X.append(x)

        Y.append(y\_next)

    return X, Y

# Построение графиков и вывод ошибок

def plot\_solutions(method\_name, X, Y, X2, Y2, order):

    Y2\_interp = [Y2[i \* 2] for i in range(len(Y))]

    abs\_errors = [abs(y\_exact(X[i]) - Y[i]) for i in range(len(X))]

    abs\_error\_max = max(abs\_errors)

    rr\_errors = [abs(Y2\_interp[i] - Y[i]) / (2 \*\* order - 1) for i in range(len(Y))]

    rr\_error\_max = max(rr\_errors)

    plt.figure(figsize=(10, 6))

    plt.plot(X, [y\_exact(x) for x in X], label='Точное решение', linewidth=2)

    plt.plot(X, Y, label=f'{method\_name}', linestyle='--', marker='o')

    plt.plot(X, Y2\_interp, label=f'{method\_name} (Рунге-Ромберг)', linestyle='--', marker='x')

    plt.xlabel('x')

    plt.ylabel('y')

    plt.title(f'Сравнение решений ОДУ ({method\_name})')

    plt.grid(True)

    plt.legend()

    plt.tight\_layout()

    plt.show()

    print(f"{method\_name}:")

    print("Абсолютная погрешность:", abs\_error\_max)

    print("Погрешность (Рунге-Ромберг):", rr\_error\_max)

# Параметры

x0 = 0

xn = 1

h = 0.1

y0 = 1

y1\_0 = 0

# Метод Эйлера

X\_euler, Y\_euler = euler\_method(x0, xn, h, y0, y1\_0)

\_, Y\_euler\_h2 = euler\_method(x0, xn, h / 2, y0, y1\_0)

plot\_solutions("Метод Эйлера", X\_euler, Y\_euler, \_, Y\_euler\_h2, order=1)

# Метод Рунге-Кутты

X\_rk, Y\_rk = runge\_kutta\_method(x0, xn, h, y0, y1\_0)

\_, Y\_rk\_h2 = runge\_kutta\_method(x0, xn, h / 2, y0, y1\_0)

plot\_solutions("Метод Рунге-Кутты 4-го порядка", X\_rk, Y\_rk, \_, Y\_rk\_h2, order=4)

# Метод Адамса 4-го порядка

X\_adams, Y\_adams = adams\_method(x0, xn, h, y0, y1\_0)

\_, Y\_adams\_h2 = adams\_method(x0, xn, h / 2, y0, y1\_0)

plot\_solutions("Метод Адамса 4-го порядка", X\_adams, Y\_adams, \_, Y\_adams\_h2, order=4)

Вывод программы:

Метод Эйлера:

Абсолютная погрешность: 0.020573108256691564

Погрешность (Рунге-Ромберг): 0.01092609401290523

Метод Рунге-Кутты 4-го порядка:

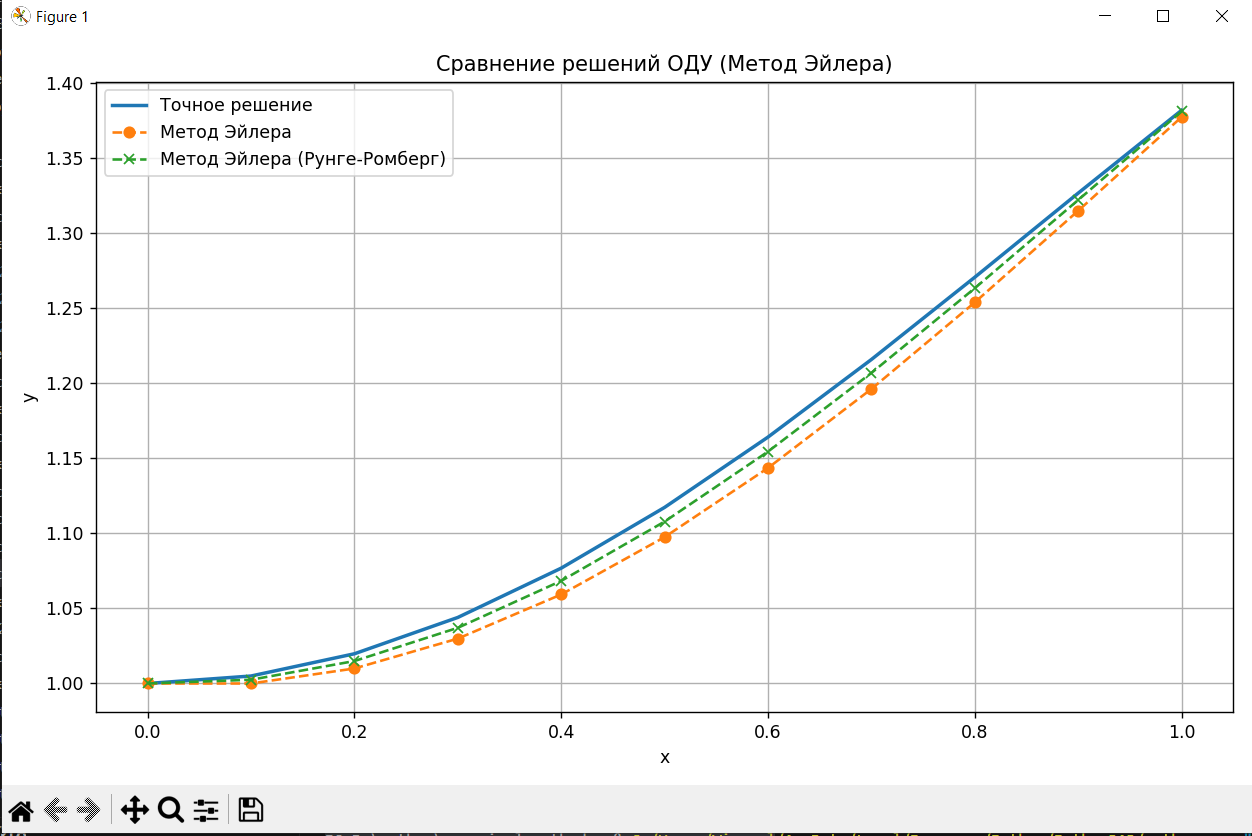
Абсолютная погрешность: 1.1613422221667946e-06

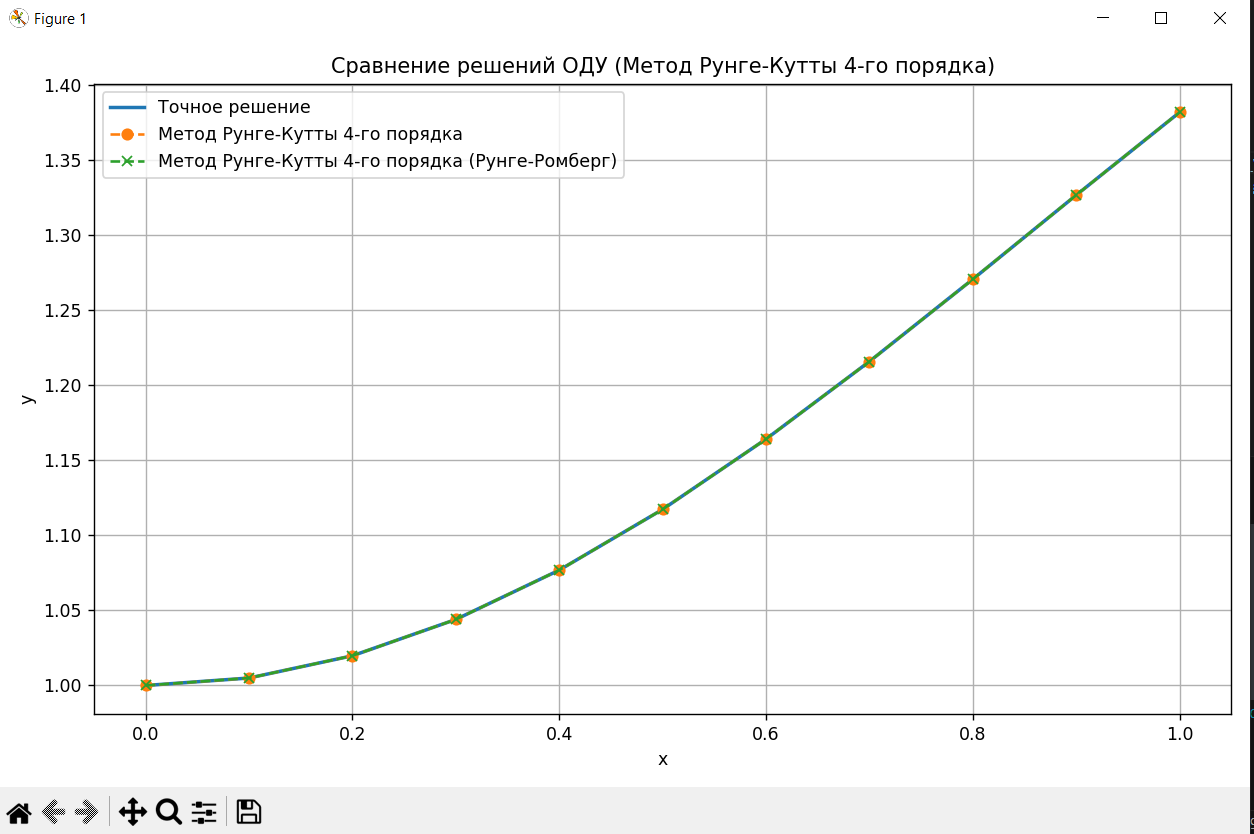
Погрешность (Рунге-Ромберг): 7.251929046899383e-08

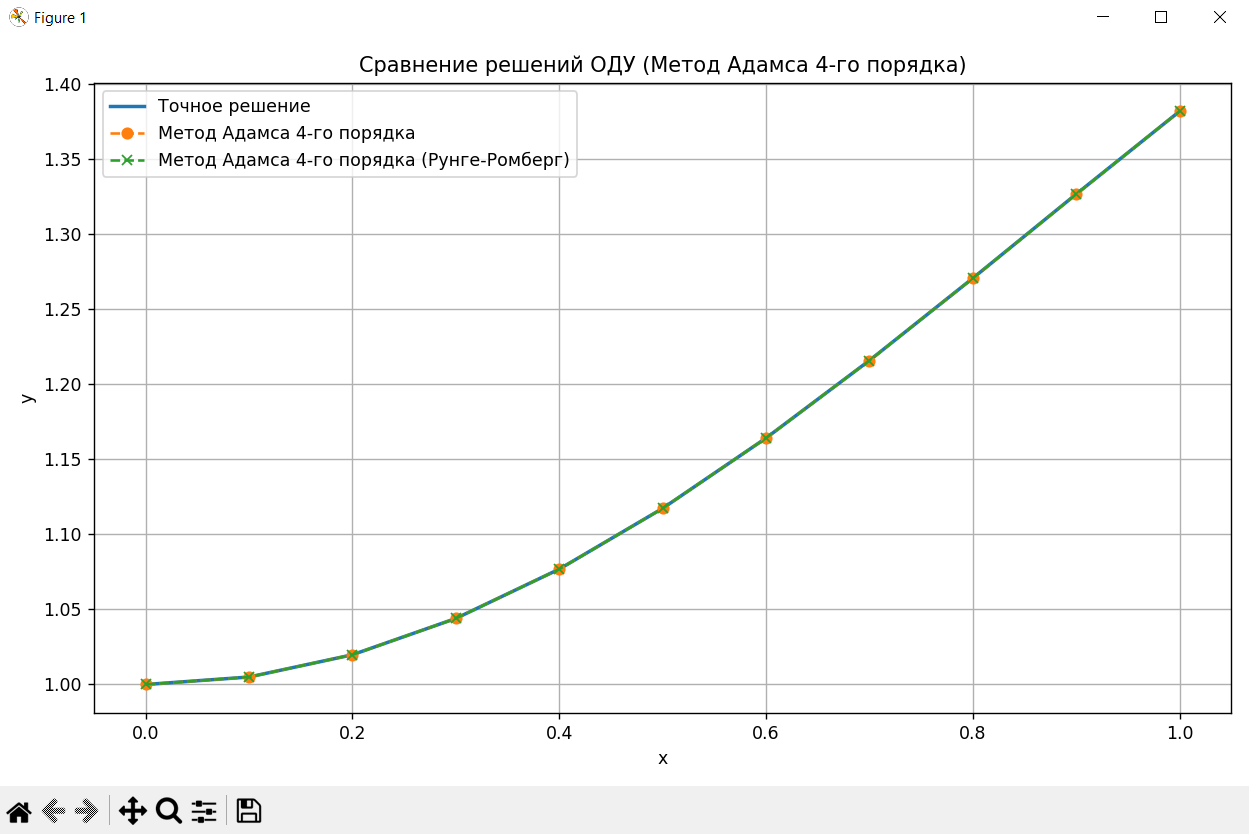
Метод Адамса 4-го порядка:

Абсолютная погрешность: 8.843624887044932e-05

Погрешность (Рунге-Ромберг): 5.419835073242988e-06







## 

## 4.2 Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ

Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

Условие:

|  |  |
| --- | --- |
| Краевая задача | Точное решение |
| xy″+2y′-xy=0,  y (1)=e-1,  y(2)=0,5e-2 |  |

Преобразуем наше дифференциальное уравнение к подобному виду:

Граничные условия:

Функция для отклонения метода стрельбы:

**Метод стрельбы:**

Суть метода заключена в многократном решении задачи Коши для приближенного нахождения решения краевой задачи.

Пусть надо решить краевую задачу на отрезке [a,b]. Вместо исходной задачи формулируется задача Коши с начальными условиями:

Где η – некоторое значение тангенса угла наклона касательной к решению в точке x=a.

Положим сначала некоторое начальное значение параметру η =

После чего решим каким-либо методом задачу Коши. Пусть

решение этой задачи на интервале [a,b], тогда сравнивая значение функции со значением в правом конце отрезка можно получить информацию для корректировки угла наклона касательной к решению в левом конце отрезка. Решая задачу Коши для нового значения η = , получим другое решение со значением на правом конце.

Другими словами, решение исходной задачи эквивалентно нахождению корня уравнения:

где

Следующее значение искомого корня определяется по соотношению:

Итерации по данной формуле выполняются до удовлетворения заданной точности.

**Конечно-разностный метод:**

Рассмотрим двухточечную краевую задачу для линейного дифференциального уравнения второго порядка на отрезке [a,b]

Введем разностную сетку на отрезке [a,b]

h = |b-a| / N. Решение данной задачи будем искать в виде сеточной функции

*,* предлагая, что решение существует и единственно. Введем разностную аппроксимацию производных следующим образом:

Подставляя аппроксимации производных в исходную задачу получим систему уравнений для нахождения :

Приводя подобные и учитывая, что при задании граничных условий первого рода два неизвестных , уже фактические определены, получим систему линейных алгебраических уравнений с трехдиагональной матрицей коэффициентов:

Для этой системы при достаточно малых шагах сетки h и выполнены условия преобладания диагональных элементов:

В случае использования граничных условий второго и третьего рода аппроксимация производных проводится с помощью односторонних разностей первого и второго порядков.

Код программы

import math

import sys

import matplotlib.pyplot as plt

# y1 = y

# y2 = y'

# y1' = y2

# y2' = (x \* y1 - 2 \* y2) / x

def f(x, y1, y2):

    return (x \* y1 - 2 \* y2) / x

# Метод Рунге-Кутты 4-го порядка для системы на шаге h

def runge\_kutta\_step(x, y1, y2, h):

    k1\_y1 = y2

    k1\_y2 = f(x, y1, y2)

    k2\_y1 = y2 + 0.5 \* h \* k1\_y2

    k2\_y2 = f(x + 0.5 \* h, y1 + 0.5 \* h \* k1\_y1, y2 + 0.5 \* h \* k1\_y2)

    k3\_y1 = y2 + 0.5 \* h \* k2\_y2

    k3\_y2 = f(x + 0.5 \* h, y1 + 0.5 \* h \* k2\_y1, y2 + 0.5 \* h \* k2\_y2)

    k4\_y1 = y2 + h \* k3\_y2

    k4\_y2 = f(x + h, y1 + h \* k3\_y1, y2 + h \* k3\_y2)

    y1\_next = y1 + (h / 6.0) \* (k1\_y1 + 2\*k2\_y1 + 2\*k3\_y1 + k4\_y1)

    y2\_next = y2 + (h / 6.0) \* (k1\_y2 + 2\*k2\_y2 + 2\*k3\_y2 + k4\_y2)

    return y1\_next, y2\_next

# Решение задачи Коши для заданного s = y'(1)

def solve\_ode(s, h=0.01):

    x = 1.0

    y1 = math.exp(-1)  # y(1) = e^{-1}

    y2 = s             # y'(1) = s

    xs = [x]

    ys = [y1]

    while x < 2.0:

        y1, y2 = runge\_kutta\_step(x, y1, y2, h)

        x += h

        xs.append(x)

        ys.append(y1)

    return xs, ys

# Функция для подсчёта отклонения в точке x=2

def F(s, h=0.01):

    xs, ys = solve\_ode(s, h)

    y2\_val = ys[-1]

    return y2\_val - 0.5 \* math.exp(-2)

# Метод секущих для нахождения s

def shooting\_method(s0, s1, eps=1e-6, max\_iter=100):

    F0 = F(s0)

    F1 = F(s1)

    # Проверка на смену знака

    if F0 \* F1 > 0:

        print("Значения функции F(s) на начальных приближениях не имеют разных знаков.")

        print("Метод стрельбы не применим. Завершение программы.")

        sys.exit(1)

    for i in range(max\_iter):

        if abs(F1) < eps:

            return s1

        # Метод секущих

        s2 = s1 - F1 \* (s1 - s0) / (F1 - F0)

        s0, s1 = s1, s2

        F0, F1 = F1, F(s1)

    print("Метод не сошёлся за заданное число итераций.")

    sys.exit(1)

# Точное решение

def exact\_solution(x):

    return math.exp(-x) / x

# Ошибка Рунге-Ромберга для метода порядка 4

def runge\_romberg(y\_h, y\_h2, h, p=4):

    error = 0.0

    n = min(len(y\_h), len(y\_h2) // 2)

    for i in range(n):

        err\_i = abs(y\_h[i] - y\_h2[2\*i]) / (2\*\*p - 1)

        if err\_i > error:

            error = err\_i

    return error

s0 = -1.0

s1 = 0.0

s\_found = shooting\_method(s0, s1)

h = 0.01

xs, ys = solve\_ode(s\_found, h)

h2 = h / 2

xs2, ys2 = solve\_ode(s\_found, h2)

# Вычисляем ошибку Рунге-Ромберга

error\_rr = runge\_romberg(ys, ys2, h)

print(f"Найденное значение y'(1) = {s\_found}")

print(f"Ошибка Рунге-Ромберга (приближённая) = {error\_rr}")

plt.figure(figsize=(10,6))

# Численное решение — точки с маркерами и линиями

plt.plot(xs, ys, 'bo-', label="Численное решение (метод стрельбы)", markersize=4, linewidth=1)

# Точное решение — гладкая красная пунктирная линия

x\_exact = [1 + i \* 0.001 for i in range(1001)]

y\_exact = [exact\_solution(x) for x in x\_exact]

plt.plot(x\_exact, y\_exact, 'r--', label="Точное решение", linewidth=2)

plt.xlabel("x")

plt.ylabel("y")

plt.title("Решение краевой задачи методом стрельбы")

plt.legend(fontsize=12)

plt.grid(True)

plt.tight\_layout()

plt.show()

import math

import matplotlib.pyplot as plt

# Параметры задачи

a, b = 1.0, 2.0

ya = math.exp(-a)          # y(1) = e^{-1}

yb = 0.5 \* math.exp(-b)    # y(2) = 0.5 e^{-2}

# Функции p(x), q(x), f(x)

def p(x): return 2 / x

def q(x): return -1

def f(x): return 0.0  # однородное уравнение

def solve\_fdm(n):

    h = (b - a) / (n + 1)

    x = [a + (i+1)\*h for i in range(n)]

    A = [[0.0]\*n for \_ in range(n)]

    d = [0.0]\*n

    for i in range(n):

        xi = x[i]

        pi = p(xi)

        qi = q(xi)

        A[i][i] = h\*\*2 \* qi - 2

        if i > 0:

            A[i][i-1] = 1 - h \* pi / 2

        if i < n - 1:

            A[i][i+1] = 1 + h \* pi / 2

        d[i] = h\*\*2 \* f(xi)

    d[0]   -= (1 - h \* p(x[0]) / 2) \* ya

    d[-1] -= (1 + h \* p(x[-1]) / 2) \* yb

    def lu\_decomposition(M):

        n = len(M)

        L = [[0.0]\*n for \_ in range(n)]

        U = [[0.0]\*n for \_ in range(n)]

        for i in range(n):

            L[i][i] = 1.0

        for j in range(n):

            for i in range(j+1):

                s = sum(U[k][j]\*L[i][k] for k in range(i))

                U[i][j] = M[i][j] - s

            for i in range(j+1, n):

                s = sum(U[k][j]\*L[i][k] for k in range(j))

                if abs(U[j][j]) < 1e-15:

                    raise ZeroDivisionError("LU decomposition failed: zero pivot")

                L[i][j] = (M[i][j] - s) / U[j][j]

        return L, U

    def forward\_substitution(L, b):

        n = len(b)

        y = [0.0]\*n

        for i in range(n):

            y[i] = b[i] - sum(L[i][j]\*y[j] for j in range(i))

        return y

    def backward\_substitution(U, y):

        n = len(y)

        x = [0.0]\*n

        for i in reversed(range(n)):

            if abs(U[i][i]) < 1e-15:

                raise ZeroDivisionError("Back substitution failed: zero pivot")

            x[i] = (y[i] - sum(U[i][j]\*x[j] for j in range(i+1, n))) / U[i][i]

        return x

    L, U = lu\_decomposition(A)

    y\_internal = backward\_substitution(U, forward\_substitution(L, d))

    x\_full = [a] + x + [b]

    y\_full = [ya] + y\_internal + [yb]

    return x\_full, y\_full

# Основное решение с шагом h

n1 = 10

x1, y1 = solve\_fdm(n1)

# Решение с шагом h/2

n2 = 2 \* n1

\_, y2 = solve\_fdm(n2)

# Расчёт ошибки Рунге–Ромберга (только для внутренних точек)

p\_order = 2

errors = []

for i in range(n1):

    y1\_i = y1[i+1]

    y2\_i = y2[2\*i + 1]

    rr\_error = abs(y1\_i - y2\_i) / (2\*\*p\_order - 1)

    errors.append(rr\_error)

max\_error = max(errors)

print(f"Ошибка Рунге–Ромберга: {max\_error:.6e}")

# Точное решение

def exact\_solution(x): return math.exp(-x) / x

x\_exact = [a + i\*0.01 for i in range(int((b - a)/0.01)+1)]

y\_exact = [exact\_solution(xi) for xi in x\_exact]

# График

plt.plot(x\_exact, y\_exact, label="Точное решение", color="red")

plt.scatter(x1, y1, color="blue", label="Численное решение (n=10)", s=25)

plt.xlabel("x")

plt.ylabel("y")

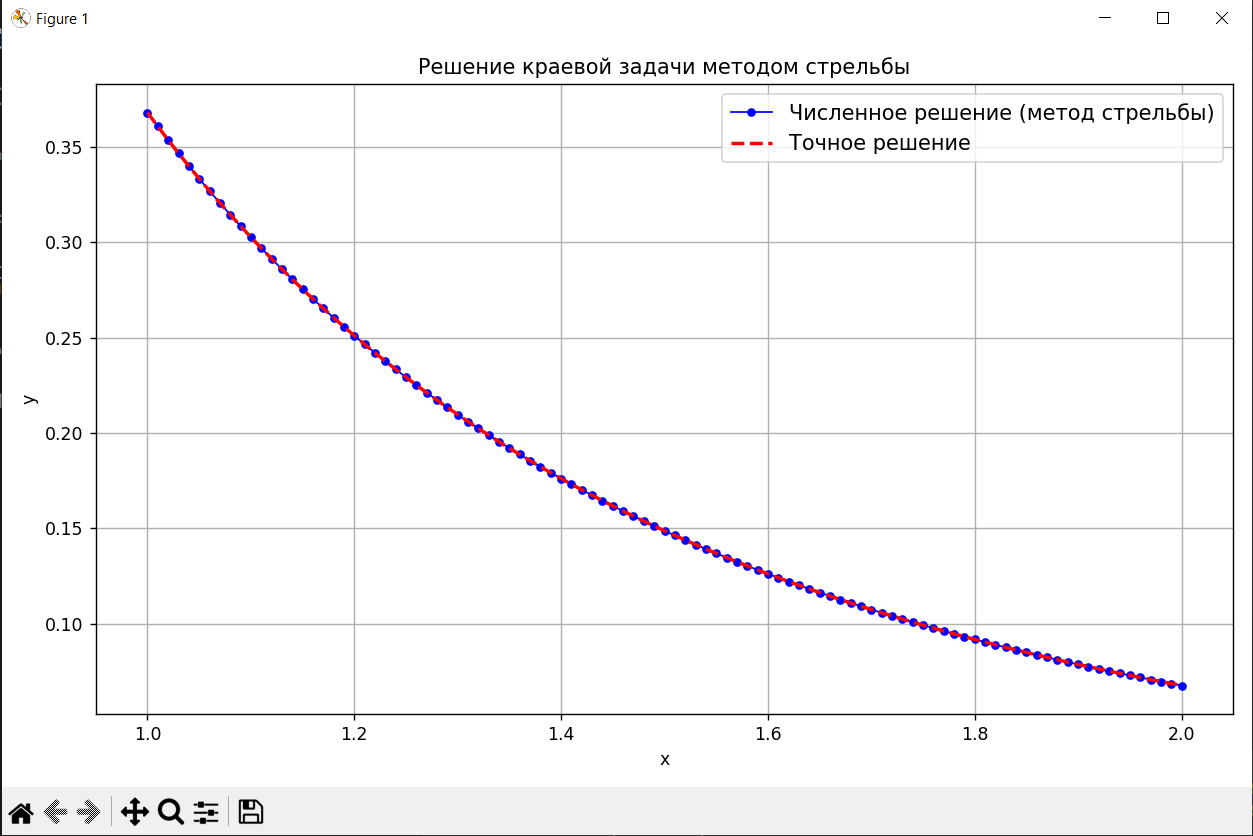
plt.title("Конечно-разностный метод")

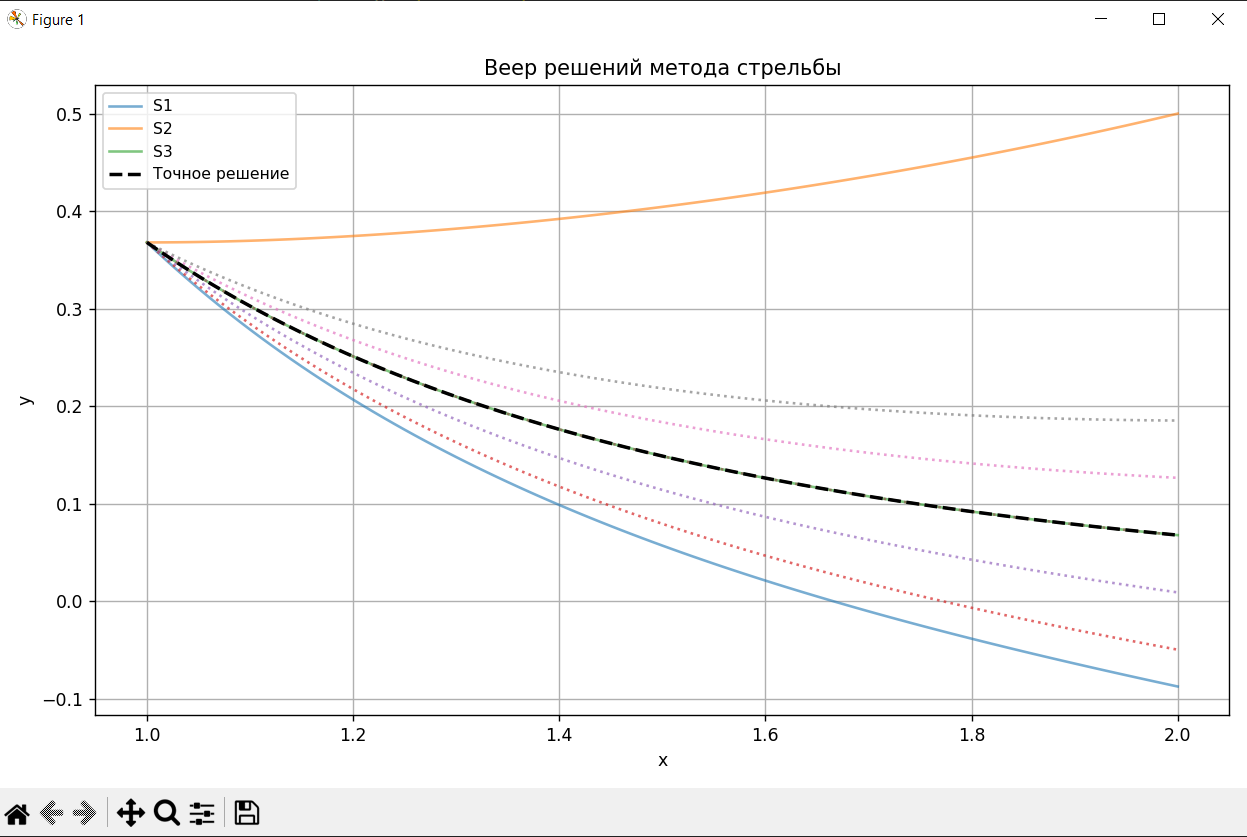
plt.legend()

plt.grid(True)

plt.show()

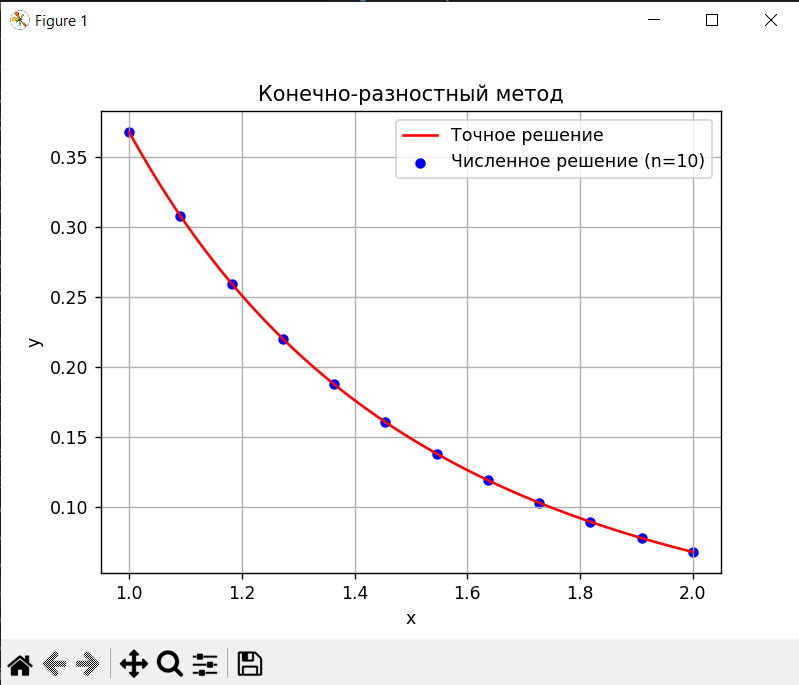
Вывод программы





Найденное значение y'(1) = -0.7357588830106639

Ошибка Рунге-Ромберга = 2.687941906091377e-11



Ошибка Рунге–Ромберга: 8.968012e-03