МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра №806 «Вычислительная математика и программирование»

Отчёт
по предмету «Численные методы»

Выполнил: Иванченко М. Д.
Группа: 8О-310Б
Преподаватель: О. Л. Демидова

Москва 2025

Содержание

1 Решение СЛАУ

1.1 Задание 1. Метод Гаусса

1.1.1 Теоретическая часть

Решить систему линейных алгебраических уравнений методом Гаусса. Вычислить определитель матрицы. Для заданной матрицы методом Гаусса вычислить обратную матрицу. Система представлена в формуле (1.1.1.1).

$$\begin{cases} 6x_1 + 3x_2 + 4x_3 + 7x_4 + x_5 = 63 \\ 3x_1 + 2x_2 + x_3 - 5x_4 + 2x_5 = 69 \\ 7x_1 + 5x_3 + 2x_4 - 7x_5 = 30 \\ 2x_1 + 5x_2 - 4x_3 + x_4 + 3x_5 = 13 \\ 3x_1 + 4x_2 - 5x_3 + 2x_4 + 5x_5 = 16 \end{cases} \quad (1.1.1.1)$$

Алгоритм работает в 2 прохода по матрице, прямой ход приводит матрицу к верхнедиагональному виду, обратный ход вычисляет решение системы уравнений, определитель и обратную матрицу.

Для каждом i-ом шаге, ведущий элемент a_{ii} используется для обнуления всех элементов a_{ki} $(k > i)$. Первым шагом нормализуем строку по формуле 1.1.1.2.

$$a'_{ij} = \frac{a_{ij}}{a_{ii}}, j = i, i+1, \dots, n \quad (1.1.1.2)$$

Для каждой строки $k \neq i$ элемент a_{ki} обнуляется путем вычитания строки i, умноженной на коэффициент a_{ki} по формуле 1.1.1.3.

$$a'_{kj} = a_{kj} - a_{ki} \cdot a'_{ij}, j = i, i+1, \dots, n, k > i \quad (1.1.1.3)$$

После приведения матрицы к верхнедиагональному виду вычисляем определитель по формуле 1.1.1.4 как произведение элементов на диагонали.

$$\det(A) = \prod_{i=1}^{n} a'_{ii} \quad (1.1.1.4)$$

Для получения обратной матрицы нужно привести главную матрицы к единичной, исключив элементы выше главной диагонали, тогда на месте первоначальной единичной матрицы получим обратную, а на месте столбца b получим решение СЛАУ. Приведение главной матрицы к единичной производится по формуле 1.1.1.5.

$$a'_{ik} = a_{ik} - a_{jk} \cdot a'_{ij}, k = 1,2, \dots, n, i = n-1, n-2, \dots, 0 \quad (1.1.1.5)$$

1.1.2 Основной код программы

Метод Гаусса реализован в двух функциях. Функция forward_step реализует прямой проход алгоритма, а функция gauss_back реализует обратный проход алгоритма. Код функций представлен в листинге 1.1.1.2.1 и 1.1.1.2.2 соответственно.

Листинг 1.1.2.1 Прямой проход метода Гаусса

```
def to_step_matrix(coeffs, reversed):
det = 1

for i in range(len(coeffs)):
    tmp = coeffs[i][i]
    det *= coeffs[i][i]
    for j in range(len(coeffs[i])):
        coeffs[i][j] /= tmp
        if j < len(coeffs[i]) - 1:
            reversed[i][j] /= tmp
    #
    for n in range(i + 1, len(coeffs)):
        tmp2 = coeffs[n][i]
        for j in range(len(coeffs[n])):
            coeffs[n][j] -= coeffs[i][j] * tmp2
            if j < len(coeffs[n]) - 1:
                reversed[n][j] -= reversed[i][j] * tmp2
    # print_matrix(coeffs)
    # print_matrix(reversed_matrix)
return round(det, 3)
```

Листинг 1.1.2.2. Обратный ход метода Гаусса

```
def gauss_back(coeffs, reversed):
n = len(coeffs)
res = []
for _ in range(n):
    res.append(0)

for k in range(n - 1, -1, -1):
    res[k] = coeffs[k][-1]

    for j in range(k, n - 1):
        res[k] -= coeffs[k][j + 1] * res[j + 1]
```

```
            for m in range(len(reversed[k])):
                reversed[k][m] -= coeffs[k][j + 1] * reversed[j + 1][m]

    for i in range(len(reversed)):
        for j in range(len(reversed[i])):
            reversed[i][j] = round(reversed[i][j], 4)

    res = [round(i, 3) for i in res]
    return res
```

1.1.3 Результаты работы программы

Результаты работы программы представлены на рисунке 1.

```
X: [6.0, 4.0, 9.0, -4.0, 7.0]
Определитель: -7208.0
Обратная матрица
-0.0314 0.0533 0.0993 -0.1956 0.2414
0.0566 0.0189 -0.0377 0.4151 -0.3208
0.1287 0.0821 -0.076 0.0422 -0.1903
0.0608 -0.1121 -0.0111 -0.0103 0.0233
0.078 0.0799 -0.101 -0.1684 0.1121
```

Рисунок 1, результаты работы метода Гаусса

1.2 Задание 2. LU-разложение.

1.2.1 Теоретическая часть

Выполнить LU-разложение и найти решение системы линейных алгебраических уравнений. Вычислить определитель матрицы и построить обратную матрицу. Исходная система представлена в формуле (1.2.1.1).

$$\begin{cases} 6x_1 + 3x_2 + 4x_3 + 7x_4 + x_5 = 63 \\ 3x_1 + 2x_2 + x_3 - 5x_4 + 2x_5 = 69 \\ 7x_1 + 5x_3 + 2x_4 - 7x_5 = 30 \\ 2x_1 + 5x_2 - 4x_3 + x_4 + 3x_5 = 13 \\ 3x_1 + 4x_2 - 5x_3 + 2x_4 + 5x_5 = 16 \end{cases} \quad (1.2.1.1)$$

LU-разложение матрицы A представляет собой представление матрицы в виде произведения двух матриц: A = LU, где L — нижняя треугольная матрица с единицами на главной диагонали, а U — верхняя треугольная матрица. Процесс

разложения заключается в нахождении матриц L и U, которые позволяют выразить систему линейных уравнений в более удобной форме для решения.

Система линейных уравнений, представленная в виде (1.2.1.2):

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{cases} \quad (1.2.1.2)$$

может быть решена с использованием LU-разложения. Для этого матрицу A представляют как произведение матриц L и U, где L — это нижняя треугольная матрица с единичной диагональю, а U — верхняя треугольная матрица. Визуальный вид этих матриц представлен в формулах (1.2.1.3) и (1.2.1.4).

$$L = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{pmatrix} \quad (1.2.1.3)$$

и

$$U = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{pmatrix} \quad (1.2.1.4)$$

Эти матрицы вычисляются поэлементно. Для первой строки разложения элементы матрицы A прямо соответствуют элементам матрицы U и имеют вид (1.2.1.5).

$$a_{1j} = u_{1j} (j = 1, \dots, n) \quad (1.2.1.5)$$

Для второй строки элементы матрицы A выражаются через элементы L и U при помощи формулы (1.2.1.6).

$$a_{21} = l_{21}u_{11}, a_{22} = l_{21}u_{12} + u_{22}, \dots, a_{2n} = l_{21}u_{1n} + u_{2n} \quad (1.2.1.6)$$

Для третьей строки выражения становятся более сложными, так как включают элементы обеих матриц:

$$a_{31} = l_{31}u_{11}, a_{32} = l_{31}u_{12} + l_{32}u_{22}, \dots, a_{3n} = l_{31}u_{1n} + l_{32}u_{2n} + u_{3n} \quad (1.2.1.7)$$

Общий элемент для строки i и столбца j выражается через сумму произведений соответствующих элементов матриц L и U:

$$a_{ij} = \sum_{k=1}^{\min(i,j)} l_{ik}\, u_{kj} \quad (1.2.1.8)$$

при этом условие $l_{ii} = 1$ обязательно выполняется, так как на главной диагонали матрицы L всегда стоят единицы.

Когда LU-разложение получено, система уравнений A x = b может быть решена с использованием следующего алгоритма. Сначала решается система вида:

$$Ly = b \quad (1.2.1.9)$$

относительно y методом прямой подстановки, так как матрица L — нижняя треугольная. После этого решается система вида:

$$Ux = y \quad (1.2.1.10)$$

относительно x методом обратной подстановки, так как матрица U — верхняя треугольная. Это позволяет эффективно решать систему линейных уравнений без необходимости вычислять обратную матрицу.

1.2.2 Основной код программы

Функция ProcessLU принимает расширенную матрицу, являющуюся объединением матрицы A, единичной матрицы и столбца b, возвращает преобразованную матрицу, включающую в себя матрицы L и U. Код функции представлен в листинге 1.2.2.1.

Листинг 1.2.2.1.

```
def ProcessLU(extended_matrix: Matrix) -> Matrix:
    n = len(extended_matrix)
    total_cols = len(extended_matrix[0])  # 2*n + 1
    for i in range(n):
        for j in range(i + 1, n):
            if abs(extended_matrix[i][i]) < 1e-15:
                raise ZeroDivisionError("Матрица вырождена")
            factor = extended_matrix[j][i] / extended_matrix[i][i]
            extended_matrix[j][i] = factor
            for k in range(i + 1, total_cols):
                extended_matrix[j][k] -= factor * extended_matrix[i][k]
    return extended_matrix
```

Функция SolveBackward используется для обратного хода метода Гаусса, функция возвращает решение системы уравнений и обратную матрицу. Код функции представлен в листинге 1.2.2.2.

Листинг 1.2.2.2

```
def SolveBackward(extended_matrix: Matrix) -> Matrix:
    n = len(extended_matrix)
    total_cols = len(extended_matrix[0])
    for i in range(n - 1, -1, -1):
        diag_element = extended_matrix[i][i]
        for j in range(i, total_cols):
            extended_matrix[i][j] /= diag_element
        for k in range(i):
            factor = extended_matrix[k][i]
            for j in range(i, total_cols):
                extended_matrix[k][j] -= factor * extended_matrix[i][j]
    return extended_matrix
```

1.2.3 Результаты работы программы

Результаты работы программы представлены на рисунке 2.

```
Начальная расширенная матрица [A | b | I]:
[ 6.0  3.0  4.0  7.0  1.0  63.0  1.0  0.0  0.0  0.0  0.0 ]
[ 3.0  2.0  1.0  -5.0  2.0  69.0  0.0  1.0  0.0  0.0  0.0 ]
[ 7.0  0.0  5.0  2.0  -7.0  30.0  0.0  0.0  1.0  0.0  0.0 ]
[ 2.0  5.0  -4.0  1.0  3.0  13.0  0.0  0.0  0.0  1.0  0.0 ]
[ 3.0  4.0  -5.0  2.0  5.0  16.0  0.0  0.0  0.0  0.0  1.0 ]

Матрица после LU-разложения:
[ 6.0  3.0  4.0  7.0  1.0  63.0  1.0  0.0  0.0  0.0  0.0 ]
[ 0.5  0.5  -1.0  -8.5  1.5  37.5  -0.5  1.0  0.0  0.0  0.0 ]
[ 1.1667  -7.0  -6.6667  -65.6667  2.3333  219.0  -4.6667  7.0  1.0  0.0  0.0 ]
[ 0.3333  8.0  -0.4  40.4  -8.4  -220.4  1.8  -5.2  0.4  1.0  0.0 ]
[ 0.5  5.0  0.3  1.5025  8.9208  62.4455  0.6955  0.7129  -0.901  -1.5025  1.0 ]

Определитель A:
-7208.0

Матрица после обратной подстановки:
[ 1.0  0.0  0.0  0.0  0.0  6.0  -0.0314  0.0533  0.0993  -0.1956  0.2414 ]
[ 0.5  1.0  0.0  0.0  0.0  4.0  0.0566  0.0189  -0.0377  0.4151  -0.3208 ]
[ 1.1667  -7.0  1.0  -0.0  -0.0  9.0  0.1287  0.0821  -0.076  0.0422  -0.1903 ]
[ 0.3333  8.0  -0.4  1.0  0.0  -4.0  0.0608  -0.1121  -0.0111  -0.0103  0.0233 ]
[ 0.5  5.0  0.3  1.5025  1.0  7.0  0.078  0.0799  -0.101  -0.1684  0.1121 ]

Решение СЛАУ x:
[        6         4         9        -4         7 ]
```

Рисунок 2, результаты работы LU разложения

1.3 Задание 3. Метод прогонки

1.3.1 Теоретическая часть

Методом прогонки найти решение системы линейных алгебраических уравнений. Вычислить определитель матрицы. Система представлена в формуле (1.3.1.1).

$$\begin{cases} 7x_1 + 3x_2 = 50 \\ 2x_1 - 7x_2 - 4x_3 = -68 \\ 2x_2 + 11x_3 + 5x_4 = 11 \\ -4x_3 + 12x_4 - 7x_5 = 53 \\ 5x_4 + 8x_5 + 3x_6 = 49 \\ -5x_5 + 10x_6 + 4x_7 = 73 \\ 3x_6 + 7x_7 + 2x_8 = 53 \\ 4x_7 + 9x_8 = 89 \end{cases} \quad (1.3.1.1)$$

Метод прогонки — это специализированный, численно устойчивый и крайне эффективный алгоритм для решения систем линейных уравнений, чья матрица коэффициентов является трехдиагональной. Он основан на идее исключения Гаусса, но использует лишь два цикла, значительно сокращая вычислительную сложность до $O(n)$ (вместо $O(n^3)$ для общего метода Гаусса).

Метод состоит из двух этапов:

1) вычисляются прогоночные коэффициенты P_i и Q_i;

2) по найденным коэффициентам P_i и Q_i.последовательно находятся неизвестные x_i, начиная с последнего x_n.

Система представляется в трехдиагональной форме, которая выглядит следующим образом:

$$\begin{cases} a_1x_1 + c_1x_2 = d_1 \\ a_2x_1 + b_2x_2 + c_2x_3 = d_2 \\ a_3x_2 + b_3x_3 + c_3x_4 = d_3 \\ \vdots \\ a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n = d_{n-1} \\ a_nx_{n-1} + b_nx_n = d_n \end{cases} \quad (1.3.1.2)$$

Здесь P_i и Q_i прогоночные коэффициенты, определяемые по нижеприведенным формулам (1.3.1.3), (1.3.1.4), (1.3.1.5), (1.3.1.6).

$$P_1 = -\frac{c_1}{b_1} \quad (1.3.1.3)$$

$$Q_1 = \frac{d_1}{b_1} \quad (1.3.1.4)$$

$$Q_i = \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}}, i = 2,3, \dots, n \quad (1.3.1.5)$$

$$Q_i = \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}}, i = 2,3, \dots, n \quad (1.3.1.6)$$

После того как будут найдены прогоночные коэффициенты (прямой ход), можно вычислить значения неизвестных путем обратной подстановки (обратный ход) по формуле (1.3.1.7).

$$x_i = P_i x_{i+1} + Q_i, i = n-1, n-2, \dots, 1 \quad (1.3.1.7)$$

Для последней неизвестной нужно воспользоваться формулой (1.3.1.8).

$$x_n = Q_n \quad (1.3.1.8)$$

Достаточным условием корректности метода прогонки и устойчивости его к погрешностям вычислений является условие преобладания диагональных коэффициентов (1.3.1.9).

$$|b_i| \geq |a_i| + |c_i| \quad (1.3.1.9)$$

Поскольку прямой ход метода прогонки эквивалентен прямому ходу Гаусса, определитель матрицы A равен произведению элементов на главной диагонали преобразованной матрицы:

$$\det(A) = \prod_{i=1}^{n} (b_i + a_i P_{i-1}) \quad (1.3.1.10)$$

1.3.2 Основной код программы

Алгоритм реализован в функции progonka, функция принимает полную матрицу и возвращает вектор x и определитель матрицы. Код функции представлен в листинге 1.3.2.1.

Листинг 1.3.2.1

```
def progonka(coeffs) -> tuple:
    det = 1.0
    prev_p = 0
    prev_q = 0
    p_list = []
    q_list = []
    for n in range(len(coeffs)):
        d = coeffs[n][-1]
```

```
        if n == (len(coeffs) - 1):
            a, b, c = coeffs[n][n - 1], coeffs[n][n], 0
        elif n == 0:
            a, b, c = 0, coeffs[n][n], coeffs[n][n + 1]
        else:
            a, b, c = coeffs[n][n - 1], coeffs[n][n], coeffs[n][n + 1]
        p = -c / (b + a * prev_p)
        q = (d - a * prev_q) / (b + a * prev_p)
        det *= (b + a * prev_p)
        p_list.append(p)
        q_list.append(q)
        prev_q = q
        prev_p = p
    res = [0.0] * len(coeffs)
    res[-1] = q_list[-1]
    for n in range(len(coeffs) - 2, -1, -1):
        x = p_list[n] * res[n + 1] + q_list[n]
        res[n] = x
    return (res, det)
```

1.3.3 Результаты работы программы

Результаты работы программы представлены на рисунке 3

```
Решение:
[[2.0], [12.0], [-3.0], [4.0], [1.0], [7.0], [2.0], [9.0]]

Определитель: -42020398.0
```

Рисунок 3, метод прогонки

1.4 Задание 4. Метод простых итераций

1.4.1 Теоретическая часть

Методом простых итераций, Зейделя решить систему линейных алгебраических уравнений с погрешностью $\varepsilon = 0{,}0001$. Указать количество итераций. Исходная система имеет вид (1.4.1.1).

$$\begin{cases} 6x_1 - 5x_2 - 19x_3 + 7x_4 = 46 \\ -5x_1 - 8x_2 + 4x_3 + 18x_4 = 68 \\ -4x_1 + 13x_2 - 2x_3 + 5x_4 = 61 \\ 15x_1 + 6x_2 - 4x_3 - 3x_4 = 89 \end{cases} \quad (1.4.1.1)$$

Итерационные методы (простых итераций, Зейделя) гарантированно сходятся, если матрица А удовлетворяет условию диагонального преобладания.

Таким образом, условие сходимости выглядит следующим образом, причем выполняться оно должно для строк или для столбцов:

$$| a_{ii} |> \sum_{j \neq i} | a_{ij} | \text{ для всех} i = 1,2, \dots, n \quad (1.4.1.2)$$

Преобразуем исходную систему, чтобы выполнялось условие сходимости:

$$\begin{cases} 15x_1 + 6x_2 - 4x_3 - 3x_4 = 89 \\ 6x_1 - 5x_2 - 1 - 4x_1 + 13x_2 - 2x_3 + 5x_4 = 61 \\ 9x_3 + 7x_4 = 46 \\ -5x_1 - 8x_2 + 4x_3 + 18x_4 = 68 \end{cases} \quad (1.4.1.3)$$

Для вычисления вводят итерационную формулу (1.4.1.4).

$$x_i^{(k+1)} = \frac{1}{A_{ii}} (b_i - \sum_{j=1, j \neq i}^{n} A_{ij} \cdot x_j^{(k)}) \quad (1.4.1.4)$$

На каждой итерации производится проверка критерия остановы (формула (1.4.1.5)) для заданной изначально точности.

$$\max_i | x_i^{(k+1)} - x_i^{(k)} |< \epsilon \quad (1.4.1.5)$$

Если критерий остановы выполнился, алгоритм можно останавливать, так как необходимая точность достигнута.

1.4.2 Основной код программы

Функция Check проверяет достаточное условие сходимости метода итерации и метода Зейделя. Код программы представлен в листинге 1.4.2.1.

Листинг 1.4.2.1. Проверка достаточного условия итерационных методов

```
def chek(A) -> bool:
    n = len(A)
    for i in range(n):
        mx = abs(A[i][i])
        sm_row, sm_str = 0, 0
        for j in range(n):
            if i != j:
                sm_row += abs(A[i][j])
                sm_str += abs(A[j][i])
        if mx < sm_row or mx < sm_str:
            return False
    return True
```

Функция Solve решает систему методом простых итераций. Код функции представлен в листинге 1.4.2.2.

Листинг 1.4.2.2. Метод простых итераций

```
def solve(A):
    n = len(A)
    x = [A[i][n] for i in range(n)]
    x_new = [0.0] * n
    for i in range(n):
        if A[i][i] == 0:
            raise ValueError(f"A[i][i] = 0 error")
    for iteration in range(10000000):
        for i in range(n):
            s = 0.0
            for j in range(n):
                if i != j:
                    s += A[i][j] * x[j]
            x_new[i] = (A[i][n] - s) / A[i][i]
        max_diff = 0.0
        for i in range(n):
            diff = abs(x_new[i] - x[i])
            if diff > max_diff:
                max_diff = diff
        if max_diff < tol:
            print(f"Solusion find after {iteration + 1} iteration.")
            return x_new
        x = x_new.copy()
```

1.4.3 Результаты работы программы

Результаты работы метода простой итерации представлены на рисунке 4.

```
Solution has been found after 25 iteration.
X: [6.000045129743023, 3.9999789429997814, 1.000028144524437, 6.999946646423413]
calculated  |  expected |  error
89.000598 |      89.0 |  0.000598
60.999223 |      61.0 |  0.000777
45.999468 |      46.0 |  0.000532
67.999095 |      68.0 |  0.000905
Max error: 0.000905
```

Рисунок 4, метод простых итераций

1.5 Задание 5. Метод Зейделя

Метод Зейделя во многом схож с методом простых итераций. В частности, он обладает тем же условием сходимости:

$$| a_{ii} | > \sum_{j \neq i} | a_{ij} | \text{ для всех} i = 1,2, \dots, n \quad (1.5.1)$$

Для решения системы методом Зейделя, используется формула 1.5.2.

$$x_i^{(k+1)} = \frac{1}{A_{ii}} (b_i - \sum_{j=1}^{i-1} A_{ij} \cdot x_j^{(k+1)} - \sum_{j=i+1}^{n} A_{ij} \cdot x_j^{(k)}) \quad (1.5.2)$$

Метод Зейделя использует уже новые, пересчитанные значения в той же итерации для ускорения сходимости.

Функция solve решает систему методом Зейделя, отличие от метода простых итераций заключается в использовании уже посчитанных на текущей итерации значений. Код функции представлен в листинге 1.5.1.

Листинг 1.5.1. Метод Зейделя

```
def solve(A):
n = len(A)
x = [A[i][n] for i in range(n)]

for i in range(n):
    if A[i][i] == 0:
        raise ValueError(f"A[i][i] = 0 error")

for iteration in range(10000000):
    max_diff = 0.0

    for i in range(n):
        s = 0.0
        for j in range(i):
            s += A[i][j] * x[j]
        for j in range(i + 1, n):
            s += A[i][j] * x[j]

        new_val = (A[i][n] - s) / A[i][i]
        diff = abs(new_val - x[i])
        if diff > max_diff:
            max_diff = diff
        x[i] = new_val
```

```
        if max_diff < tol:
            print(f"Solution has been found after {iteration + 1}
iteration.")
            return x

    return x
```

Результаты работы метода Зейделя представлены на рисунке 5.

```
Solution has been found after 22 iteration.
X: [6.000015381932146, 4.000019927216311, 0.9999883813869133, 7.000015711213531]
computed  |  expected |  error
89.000350 |      89.0 |  0.000350
61.000299 |      61.0 |  0.000299
46.000323 |      46.0 |  0.000323
68.000000 |      68.0 |  0.000000
Max error: 0.000350
```

Рисунок 5, метод Зейделя

Как мы видим, методу Зейделя потребовалось на несколько итераций меньше для сходимости.

1.6 Задание 6. QR-разложение

1.6.1 Теоретическая часть

Определить собственные значения матрицы с точностью не более $\varepsilon = 0{,}0001$. Реализовать алгоритм QR – разложения матриц. Исходная матрица представлена в формуле (1.6.1.1).

$$\begin{vmatrix} 1 & -5 & -4 & 7 & 3 \\ 4 & 12 & 1 & 2 & 4 \\ -2 & 3 & 4 & 7 & 5 \\ 2 & 5 & -4 & 11 & 3 \\ 5 & 4 & -5 & -4 & -2 \end{vmatrix} \quad (1.6.1.1)$$

Собственные значения матрицы — это числа, удовлетворяющие уравнению $Ax = \lambda x$. Для нахождения собственных числе необходимо решить характеристическое уравнение $\det(A - \lambda I) = 0$.

Алгоритм поиска собственных значений методом QR-разложения с использованием преобразований Хаусхолдера начинается с того, что данную матрицу Aнужно привести к верхней треугольной форме Rс помощью

преобразования Хаусхолдера. Для первого столбца матрицы формируем вектор $\bar{v_1}$, который используется для создания матрицы Хаусхолдера H_1, с помощью которой обнуляется все элементы первого столбца ниже главной диагонали. Для этого вычисляем вектор $\bar{v_1}$, который включает в себя элементы первого столбца и первый элемент, который равен:

$$a_{11}^0 + \text{sign}(a_{11}^0) \cdot \sqrt{\sum_{j=1}^{n} (a_{j1}^0)^2} \tag{1.6.1.2}$$

Для каждого следующего столбца k (где $k = 2,3,\dots,n-1$), аналогично формируется вектор $\bar{v_k}$ по следующей формуле:

$$\bar{v_k} = \left(0,\dots,0, a_{kk}^0 + \text{sign}(a_{kk}^0) \sqrt{\sum_{j=k}^{n} (a_{jk}^0)^2}, a_{k+1,k}^0, \dots, a_{nk}^0\right)^T \tag{1.6.1.3}$$

Он используется для создания новой матрицы Хаусхолдера H_k. Она формируется следующим образом:

$$H_k = E - 2\frac{\bar{v_k} \cdot \bar{v_k}^T}{\bar{v_k}^T \cdot \bar{v_k}} \tag{1.6.1.4}$$

Применяя H_kк матрице A_k^0, мы обнуляем элементы k-го столбца ниже диагонали:

$$A_{k+1}^0 = H_k \cdot A_k^0 \tag{1.6.1.5}$$

Перебрав все столбцы кроме последнего, мы формируем матрицы Q и R следующим образом:

$$R^0 = A_{n-1}^0 \tag{1.6.1.6}$$

$$Q^0 = H_1 \cdot H_2 \cdot \dots \cdot H_{n-1} \tag{1.6.1.7}$$

После того как матрица A представлена как произведение матриц Q^0и R^0, переходим к QR-алгоритму для поиска собственных значений. В итерационном процессе для текущей матрицы A_0^mнаходим QR-разложение, т.е. $A_0^m = Q^m \cdot R^m$, после чего формируем новую матрицу $A_0^{m+1} = R^m \cdot Q^m$. Этот процесс повторяется до тех пор, пока матрица A_0^kне станет квазитреугольной, что означает, что она имеет блоки на главной диагонали и на поддиагонали, причем элементы поддиагонали стремятся к нулю.

Когда матрица становится квазитреугольной, собственные значения могут быть извлечены из диагональных элементов. Если есть блоки размером 2×2, то собственные значения для них вычисляются решением уравнения для матрицы:

$$\begin{bmatrix} a_{jj} & a_{j,j+1} \\ a_{j+1,j} & a_{j+1,j+1} \end{bmatrix} \quad (1.6.1.7)$$

собственные значения λ для которого находят из уравнения:

$$(a_{jj} - \lambda)(a_{j+1,j+1} - \lambda) = a_{j,j+1} \cdot a_{j+1,j} \quad (1.6.1.8)$$

Алгоритм продолжается до тех пор, пока разница между собственными значениями на каждой итерации не станет меньше заданной погрешности:

$$| \lambda^{(n+1)} - \lambda^{(n)} | < \varepsilon \quad (1.6.1.9)$$

Таким образом, метод QR с использованием преобразований Хаусхолдера позволяет эффективно находить собственные значения матрицы с помощью последовательных итераций и разложения на матрицы Qи R.

1.6.2 Основной код программы

В листинге 1.6.2.1 представлен основной цикл программы и обработка результатов после итерационного процесса.

Листинг 1.6.2.1. Основной цикл QR-разложения.

```
def GetQR(matrix):
r = matrix.copy()
n = len(matrix)
q = get_e_matrix(n)
for i in range(n - 1):
    vi = [0.0 for _ in range(i)]

    subcolumn_norm = sqrt(sum(matrix[j][i] ** 2 for j in range(i, n)))

    sign = 1 if matrix[i][i] >= 0 else -1
    first_element = matrix[i][i] + sign * subcolumn_norm
    vi.append(first_element)

    for j in range(i + 1, n):
        vi.append(matrix[j][i])

    vi = transpose_matrix([vi])

    # print(vi)
```

```
        vi1Matrix = matrix_multiply(vi, transpose_matrix(vi))
        vi2Matrix = matrix_multiply(transpose_matrix(vi), vi)

        # print(vi1Matrix)
        # print(vi2Matrix)

        vi2Matrix = new_matrix_from_digit(vi2Matrix[0][0], len(vi1Matrix),
len(vi1Matrix))

        H = piecemeal_subtraction(get_e_matrix(len(vi2Matrix)),
                                  matrux_multiply_number(2,
divide_matrix_by_element(vi1Matrix, vi2Matrix)))
        # print(H)
        r = matrix_multiply(H, r)
        q = matrix_multiply(q, H)
        matrix = r.copy()
    return q, r
```

В листинге 1.6.2.2 представлена функция, которая выполняет одну итерацию, QR-разложение и перемножение матриц Q и R.

Листинг 1.6.2.2. Итерация QR-разложения.

```
while True:
q, r = GetQR(new_matrix)
new_matrix = matrix_multiply(r, q)
blocks = extract_blocks(new_matrix, eps)
l = []
for block in blocks:
    if len(block) == 1:
        l.append(block[0][0])

    else:
        t1, t2 = get2x2_eigenvalues(block)
        l.append(t1)
        l.append(t2)

if len(prev_l) == 0:
    prev_l = l.copy()
    continue

mx_diff = 0.0
for i in range(len(l)):
    # print(l[i])
```

```
            # print(prev_l[i])
            mx_diff = max(abs(prev_l[i] - l[i]), mx_diff)
        if mx_diff < eps:
            break
        else:
            prev_l = l.copy()
            it += 1
```

В листинге 1.6.2.3 представлена функция, которая решает квадратное уравнение для нахождения собственных значений матрицы 2 на 2.

Листинг 1.6.2.3. Собственные значения матрицы 2 на 2.

```
def get2x2_eigenvalues(matrix) -> list:
    if len(matrix) != 2:
        raise ValueError("Matrix should be 2x2")

    a, b, c, d = matrix[0][0], matrix[0][1], matrix[1][0], matrix[1][1]
    discriminant = (a + d) ** 2 - 4 * (a * d - b * c)

    if discriminant > 0:
        return [((a + d) + sqrt(discriminant)) / 2, ((a + d) -
sqrt(discriminant)) / 2]
    elif discriminant < 0:
        return [((a + d) + cmath.sqrt(discriminant)) / 2, ((a + d) -
cmath.sqrt(discriminant)) / 2]
    else:
        return [(a + d) / 2, (a + d / 2)]
```

1.6.3 Результаты работы программы

Результаты работы алгоритма QR для нахождения собственных значений матрицы представлены на рисунке 6.

```
Решение найдено после 18 итераций
Полученная матрица
11.4324 -8.8133   7.597 -1.8027 -0.8245
  0.761 10.2967  -1.757  6.8846 -4.0291
    0.0    -0.0  5.1494 10.2308  6.8825
   -0.0    -0.0    -0.0 -0.6778 -4.7484
    0.0    -0.0    -0.0  0.2448 -0.2007

Собственные значения:
(10.864549611496429+2.5266661670198043j)
(10.864549611496429-2.5266661670198043j)
5.1494217808626885
(-0.43926050192776106+1.0514857074742348j)
(-0.43926050192776106-1.0514857074742348j)
```

Рисунок 6, резльтаты работы QR алгоритма

1.7 Вывод

В ходе выполнения работ по решению систем линейных алгебраических уравнений были реализованы и протестированы шесть ключевых методов. На практике удалось наглядно сравнить их эффективность, выявить сильные стороны и ограничения для разных типов задач.

Прямые методы, такие как метод Гаусса и LU-разложение, проявили себя как надежный инструмент для получения точного решения систем небольшой и средней размерности. Их главное преимущество — детерминированное количество операций, не зависящее от условных параметров, как у итерационных методов. При реализации метод Гаусса показал свою концептуальную простоту, в то время как LU-разложение, хотя и требует больше подготовительных вычислений, оказалось невероятно эффективным при необходимости решить одну систему с разными правыми частями, так как дорогостоящий этап факторизации выполняется только один раз.

Особняком среди прямых методов стоит метод прогонки. Его реализация для трехдиагональных систем впечатлила своей эффективностью — количество

операций имеет порядок O(n), что на несколько порядков быстрее стандартных методов для плотных матриц. Этот метод однозначно является лучшим выбором для СЛАУ со специфической, но часто встречающейся в прикладных задачах (например, при решении дифференциальных уравнений) трехдиагональной структурой.

Среди итерационных методов было проведено сравнение метода простых итераций и метода Зейделя. Оба метода требуют приведения системы к виду с преобладанием диагональных элементов для гарантии сходимости. На тестовых примерах метод Зейделя стабильно демонстрировал более высокую скорость сходимости по сравнению с методом простых итераций, что полностью подтверждает теорию, так как он использует уже вычисленные на текущей итерации уточненные значения. Однако, общим недостатком этих методов является невозможность предсказать точное количество итераций, а также риск расходимости при неудачном выборе начального приближения или нарушении условий сходимости.

Реализация QR-разложения, в отличие от предыдущих, была направлена не на решение конкретной системы, а на приведение матрицы к удобному виду. Этот алгоритм оказался наиболее сложным в программной реализации из-за необходимости организации последовательных ортогональных преобразований Хаусхолдера. Его основная мощь заключается в устойчивости и применении для решения более общих задач, таких как поиск собственных значений и решение переопределенных систем методом наименьших квадратов.

Таким образом, проведенный анализ позволяет сделать следующий практический вывод: не существует «универсально лучшего» метода. Выбор алгоритма критически зависит от условий задачи. Для плотных систем малой размерности оптимальны методы Гаусса или LU-разложения. Для трехдиагональных систем — исключительно метод прогонки. Для больших разреженных систем, где прямое вычисление неприемлемо по ресурсам, используются итерационные методы (Зейделя как более быстрый вариант), а для задач спектрального анализа и обработки данных незаменимым инструментом является QR-разложение. Каждый из реализованных методов занял свою важную нишу в арсенале средств для решения СЛАУ.

Прямые и итерационные методы решения систем линейных уравнений представляют собой две основные группы подходов, отличающиеся по вычислительным затратам, точности и области применения. Прямые методы, такие как метод Гаусса или LU-разложение, решают систему за конечное

количество шагов с точностью, которая зависит от численных погрешностей. Эти методы обычно требуют больше памяти, особенно для больших и плотных матриц, и имеют вычислительную сложность порядка (O(n^3)), что делает их неэффективными для очень крупных систем. Однако они всегда дают точное решение, если система хорошо обусловлена.

Итерационные методы, такие как метод простых итераций или метод Зейделя, начинают с приближённого решения и итеративно уточняют его, пока не достигнут заданной точности. Эти методы обычно требуют меньше памяти и вычислительных ресурсов, что делает их удобными для работы с большими и разреженными системами. Однако они не всегда дают точное решение, а зависят от начальных приближений и сходимости процесса, что может быть проблемой для плохо обусловленных задач. Итерационные методы могут быть более эффективными для больших задач, где требуется высокая скорость вычислений и низкие требования к памяти, но их точность зависит от числа итераций и состояния системы.

2 Решение нелинейных уравнений

2.1 Задание 7. Решение нелинейного уравнения

Методом простой итерации, методом Ньютона, методом секущих, методом хорд и методом дихотомии найти и уточнить, по крайней мере, два корня нелинейного уравнения с точность $\varepsilon = 0{,}0001$. Проверить достаточное условие сходимости каждого метода. Указать количество итераций. Начальное приближение определить графически. Уравнение представлено ниже (2.1.1).

$$5 - e^x - x^3 - 3x^2 + 4x = 0 \qquad (2.1.1)$$

График представлен на рисунке 7. По нему можно определить, что уравнение имеет три корня, для которых можно выбрать следующие интервалы: [-3.79, -3.5], [-1.2, -0.5], [1.2, 1.3].

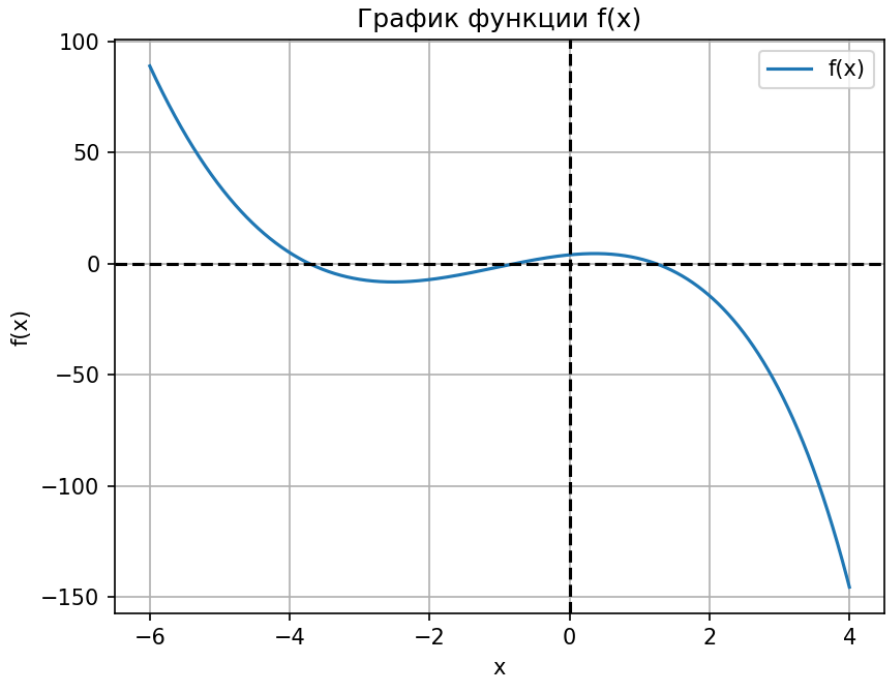

Рисунок 7, график исходной функции

2.1.1 Метод простой итерации

Метод простой итерации (или метод фиксированной точки) является одним из основных численных методов решения нелинейных уравнений. Его суть заключается в преобразовании исходного уравнения к эквивалентному виду:

$$x = \varphi(x), \qquad (2.1.1.1)$$

После этого решение ищется как фиксированная точка этой функции. Далее строится последовательность приближений ($x_0 = x$):

$$x_i = \varphi(x_{i-1}) \qquad (2.1.1.2)$$

которая при выполнении условий сходимости стремится к корню уравнения. Для применения метода необходимо выбрать такую функцию φ(x), чтобы на некотором интервале, содержащем корень, выполнялось условие:

$$| \varphi'(x) | < 1 \qquad (2.1.1.3)$$

Это условие гарантирует, что функция является сжимающей и последовательность итераций будет сходиться. Также требуется выбрать начальное приближение x_0, достаточно близкое к предполагаемому корню. Итерационный процесс продолжают до тех пор, пока разность между соседними приближениями не станет меньше заданной точности.

Одним из распространённых подходов к построению итерационной функции является выбор итерирующего оператора вида:

$$\varphi_i(\mathrm{x}) = x_i + \lambda f_i(\mathrm{x}) \qquad (2.1.1.4)$$

где λ— числовой параметр, обычно не превышающий по модулю 0,5 и подбираемый таким образом, чтобы обеспечить сходимость процесса. что делает итерации устойчивыми.

Ниже представлена реализация итерационного процесса метода простой итерации. В качестве значений λ для корней используются соответственно значения 0.1, 0.1, -0.1.

Листинг 2.1.1.1. Метод простой итерации

```
while True:
    it += 1
    x_new = phi(x, l)
    if abs(x - x_new) < EPSILON:
        return x_new, it
    x = x_new
```

Листинг 2.1.1.2. Итерирующая функция

```
def phi(x: float, l) -> float:
 return x + l * f(x)

def dphi(x: float, l) -> float:
 return 1 + l * df(x)
```

Характеристики метода:

1. итерационный метод первого порядка, обладает низкой скоростью сходимости;

2. рекомендуется применять для разреженных матриц или слабо заполненных матриц;
3. используется для решения систем высокого порядка;
4. контролирует точность получения результата;
5. требует обязательной проверки достаточного условия сходимости;
6. может использоваться при проектировании параллельных вычислений.

Результат работы программы представлен на рисунке 8.

```
Простая итерация:
    Корень -3.716116 был найден за 12 итераци
     Проверка: f(root) = 0.0
    Корень 1.238694 был найден за 5 итераци
     Проверка: f(root) = -0.0
    Корень -0.7911111 был найден за 9 итераци
     Проверка: f(root) = -0.0
```

Рисунок 8, резльтаты работы метода простой итерации

2.1.2 Метод Ньютона

Метод Ньютона (или метод касательных) является одним из наиболее эффективных численных методов решения нелинейных уравнений вида $f(x) = 0$. Основная идея метода заключается в замене нелинейной функции её линейной аппроксимацией в окрестности текущего приближения. Для этого используется разложение функции в ряд Тейлора с ограничением первыми членами. В результате итерационная формула метода имеет вид:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad (2.1.2.1)$$

Метод Ньютона позволяет получать приближения к корню значительно быстрее, чем метод простой итерации, благодаря квадратичной сходимости: при достаточно хорошем начальном приближении число верных знаков удваивается на каждой итерации.

Для корректной работы метода необходимо, чтобы функция $f(x)$ была достаточно гладкой в окрестности корня, а её производная $f'(x)$не обращалась в нуль. Начальное приближение x_0 выбирают таким образом, чтобы точка

находилась рядом с корнем, иначе метод может расходиться. Условие сходимости метода Ньютона можно записать следующим образом:

$$\left|\frac{f(x)f''(x)}{(f'(x))^2}\right| < 1 \qquad (2.1.2.2)$$

Итерационный процесс продолжают до выполнения критерия остановки, который обычно формулируется через разность двух соседних приближений или через значение функции в текущей точке:

$$| x_{n+1} - x_n | < \varepsilon \qquad (2.1.2.3)$$

Ниже приведен листинг кода, реализующего основную часть метода Ньютона.

Листинг 2.1.2.1. Реализация метода Ньютона вместе с проверкой достаточного условия сходимости

```
def newton(a, b):
    if f(a) * f(b) >= 0:
        raise Exception("Не выполнены условия для метода")

    if df(a) == 0 or df(b) == 0:
        raise Exception("Не выполнены условия для метода")

    if (f(a) * d2f(a) < df(a) ** 2) and \
            (f(b) * d2f(a) < df(b) ** 2):
        if abs(f(a)) < abs(f(b)):
            x = a
        else:
            x = b
    elif f(a) * d2f(a) < df(a) ** 2:
        x = a
    elif f(b) * d2f(b) < df(b) ** 2:
        x = b
    else:
        raise Exception("Не выполнены условия для метода")

    it = 0
    while True:
        x_new = x - f(x) / df(x)
        it += 1
        if abs(x_new - x) < EPSILON:
            return x_new, it
        x = x_new
```

Достоинства метода:

1. метод предназначен для уточнения отделенных корней нелинейных уравнений;
2. одношаговый метод второго порядка точности;
3. алгоритмичен и прост в реализации;
4. быстро сходится к решению, если начальное приближение выбрано близко к корню.

Недостатки метода:

1. применяется только для дифференцируемых функций;
2. не определяет кратность корней.

Результаты работы метода Ньютона приведены на рисунке 9.

```
Метод Ньютона:
    Корень -3.716088 был найден за 3 итераци
     Проверка: f(root) = 0.0
    Корень 1.238692 был найден за 3 итераци
     Проверка: f(root) = -0.0
    Корень -0.7910747 был найден за 3 итераци
     Проверка: f(root) = -0.0
```

Рисунок 9, результаты работы метода Ньютона

2.1.3 Метод секущих

Метод секущих является численным методом решения нелинейных уравнений вида $f(x) = 0$ и представляет собой модификацию метода Ньютона, в которой производная функции не вычисляется напрямую. Вместо этого её значение аппроксимируется с помощью разностного отношения, построенного по двум последним приближениям. Благодаря этому метод секущих сочетает в себе высокую скорость сходимости и отсутствие необходимости в аналитическом вычислении производной.

Так как метод секущих является модификацией метода Ньютона, их достаточные условия сходимости совпадают и имеют вид (2.1.2.2).

Итерационная формула метода секущих имеет вид:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \qquad (2.1.3.1)$$

В качестве начального значения x_0 выбирают конец отрезка, на котором выполняется условие сходимости. Если оба конца подходят, то мы выбираем тот, который ближе к корню. Скорость сходимости метода секущих лежит между скоростью метода простой итерации и квадратичной скоростью метода Ньютона. Это делает метод привлекательным в ситуациях, когда производная сложно вычисляется или не существует в аналитическом виде.

Условие остановки для метода секущих обычно формулируется через модуль разности соседних приближений:

$$| x_{n+1} - x_n | < \varepsilon \quad (2.1.3.2)$$

Метод секущих обладает рядом преимуществ: не требует вычисления производной, сходится быстрее метода простой итерации и проще в реализации, чем метод Ньютона. Однако метод может быть менее устойчивым, чем метод Ньютона, и при неудачном выборе начальных точек возможна расходимость или замедление сходимости. Несмотря на эти ограничения, метод секущих широко используется в вычислительной математике благодаря своей универсальности и эффективности.

Реализация метода приведена в листинге 2.1.3.1.

Листинг 2.1.3.1. *Реализация метода секущих*

```
def secant(a, b):
    if f(a) * f(b) >= 0:
        raise Exception("Не выполнены условия для метода")

    if df(a) == 0 or df(b) == 0:
        raise Exception("Не выполнены условия для метода")

    if (f(a) * d2f(a) < df(a) ** 2) and \
            (f(b) * d2f(a) < df(b) ** 2):
        if abs(f(a)) < abs(f(b)):
            x = a
        else:
            x = b

    if f(a) * d2f(a) < df(a) ** 2:
        x = a
    elif f(b) * d2f(b) < df(b) ** 2:
        x = b
    else:
        raise Exception("Не выполнены условия для метода")
```

```
it = 0
x_prev = x

x = x_prev - f(x_prev) / df(x_prev)

while abs(x - x_prev) > EPSILON:
    x_new = x - f(x) * (x - x_prev) / (f(x) - f(x_prev))
    x, x_prev = x_new, x
    it += 1

return x, it
```

Результат работы программы представлен на рисунке 10.

```
Метод секущих:
    Корень -3.716088 был найден за 3 итераци
     Проверка: f(root) = 0.0
    Корень 1.238692 был найден за 2 итераци
     Проверка: f(root) = 0.0
    Корень -0.7910747 был найден за 2 итераци
     Проверка: f(root) = -0.0
```

Рисунок 10, результаты работы метода секущих

2.1.4 Метод хорд

Метод хорд относится к итерационным методам, основанным на замене нелинейной функции линейной аппроксимацией. В отличие от метода секущих, где используются два изменяющихся приближения, метод хорд фиксирует одну из точек и строит все последующие приближения с использованием одной постоянной точки и текущего значения. Такой подход делает метод несколько устойчивее, но снижает скорость сходимости по сравнению с методом секущих.

Так как метод хорд является модификацией метода Ньютона, их достаточные условия сходимости совпадают и имеют вид (2.1.2.2).

Итерационная формула метода хорд имеет вид:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_0}{f(x_n) - f(x_0)} \qquad (2.1.4.1)$$

где x_0— фиксированная начальная точка, а x_n— текущее приближение. В качестве начальной фиксированной точки x_0 выбирают конец отрезка, на котором

выполняется условие сходимости. Если оба конца подходят, то мы выбираем тот, который ближе к корню. Метод основан на построении хорды (прямой линии), соединяющей точку $\big(x_0, f(x_0)\big)$ с точкой $(x_n, f(x_n))$. Точка пересечения этой хорды с осью абсцисс и является очередным приближением.

Критерий остановки метода обычно формулируется следующим образом:

$$| x_{n+1} - x_n | < \varepsilon \qquad (2.1.4.2)$$

Метод хорд обладает хорошей устойчивостью и часто используется в случаях, когда производная функции недоступна или её вычисление затруднено. По скорости сходимости он превосходит метод простой итерации, но уступает методу Ньютона и методу секущих. Его недостаток заключается в необходимости выбора удачной фиксированной точки, что может повлиять на скорость и даже возможность сходимости. Несмотря на это, метод хорд остаётся одним из базовых и часто применяемых методов в задачах численного решения нелинейных уравнений.

Реализация метода приведена в листинге 2.1.4.1.

Листинг 2.1.4.1. Реализация метода хорд

```
def hord(a, b):
if f(a) * f(b) >= 0:
    raise Exception("Не выполнены условия для метода")

if df(a) == 0 or df(b) == 0:
    raise Exception("Не выполнены условия для метода")

if (f(a) * d2f(a) < df(a) ** 2) and \
        (f(b) * d2f(a) < df(b) ** 2):
    if abs(f(a)) < abs(f(b)):
        x = a
        z = b
    else:
        x = b
        z = a

if f(a) * d2f(a) < df(a) ** 2:
    x = a
    z = b
elif f(b) * d2f(b) < df(b) ** 2:
    x = b
```

```
    z = a
else:
    raise Exception("Не выполнены условия для метода")

it = 0
while True:
    it += 1
    x_new = x - f(x) * (z - x) / (f(z) - f(x))
    if abs(x_new - x) < EPSILON:
        return x_new, it
    x = x_new
```

Результат работы программы представлен на рисунке 11.

```
Метод хорд:
    Корень -3.716086 был найден за 5 итераци
     Проверка: f(root) = -0.0
    Корень 1.238689 был найден за 3 итераци
     Проверка: f(root) = 0.0
    Корень -0.7910772 был найден за 4 итераци
     Проверка: f(root) = -0.0
```

Рисунок 11, результаты работы метода хорд

2.2 Задание 8. Решение системы нелинейных уравнений

Требуется решить систему нелинейных уравнений методом Ньютона, методом простой итерации и методом Зейделя с точностью $\varepsilon = 10^{-4}$. Начальные приближения получить как точки пересечения графиков эквивалентных функций (несколько корней). Проверить условия сходимости заданного метода. Указать количество итераций. Исходная система имеет вид:

$$\begin{cases} e^{x_1 x_2} + x_1 - 4 = 0 \\ {x_1}^2 - 4{x_2}^2 + 1 = 0 \end{cases} \quad (2.2.1)$$

На рисунке 12 представлен график искомой системы.

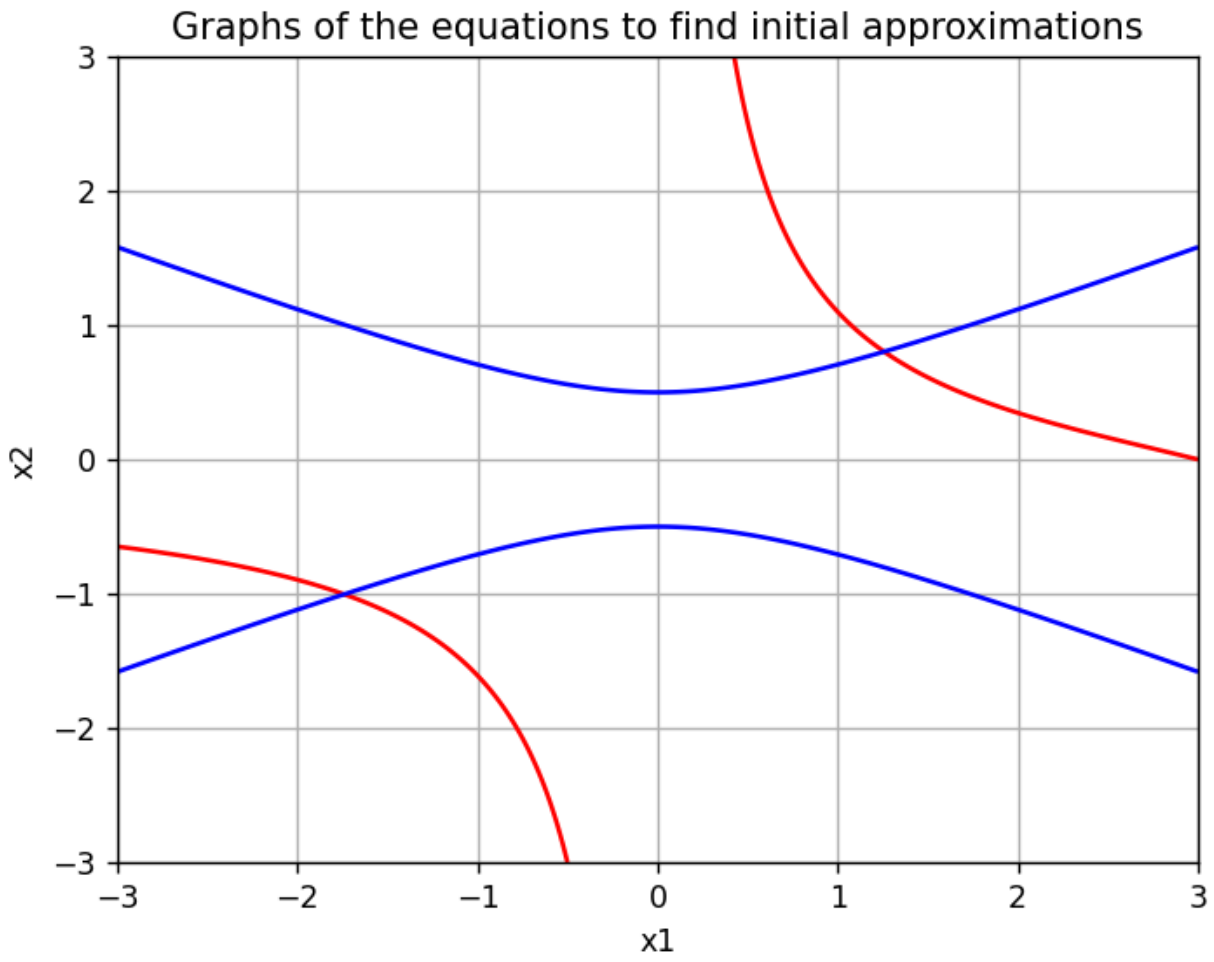

Рисунок 12, график функции

2.2.1 Метод простых итераций

Метод простых итераций для решения систем нелинейных уравнений является обобщением одноимённого метода, применяемого к одиночным уравнениям. Пусть дана система вида:

$$\mathrm{F}(\mathrm{x}) = 0 \qquad (2.2.1.1)$$

где $\mathrm{x} = (x_1, x_2, \dots, x_n)^T$, а $\mathrm{F} = (f_1, f_2, \dots, f_n)^T$.

Идея метода заключается в преобразовании данной системы к эквивалентному:

$$\mathrm{x} = \varphi(\mathrm{x}) \qquad (2.2.1.2)$$

После этого решение ищется как фиксированная точка отображения φ.

Итерационный процесс имеет вид:

$$\mathrm{x}^{(\mathrm{k}+1)} = \varphi(\mathrm{x}^{(\mathrm{k})}) \qquad (2.2.1.3)$$

где $x^{(k)}$— вектор очередного приближения. Для сходимости метода отображение φдолжно быть сжимающим в некоторой окрестности корня, что гарантирует устойчивое уменьшение погрешности на каждой итерации. Условие сжимаемости формулируется через нормы якобиана функции $\varphi(x)$и требует выполнения неравенства:

$$\| \varphi'(x) \| < 1 \quad (2.2.1.4)$$

Реализация вычисления нормы якобиана приведена в листинге 2.2.1.1.

Листинг 2.2.1.1. Реализация проверки сходимости метода простых итераций

```
def check_convergence(x1, x2):
    a, b, c, d = (dphi1_dx1(x1, x2),
                  dphi1_dx2(x1, x2),
                  dphi2_dx1(x1, x2),
                  dphi2_dx2(x1, x2))
    # print(a, b, c, d)
    q1 = abs(a) + abs(b)
    q2 = abs(c) + abs(d)

    q = max(q1, q2)

    if int(q) == 1:
        print("q == 1, сходимость не гарантирована")
        return True

    return q < 1
```

Одним из распространённых подходов к построению итерационной функции является выбор итерирующего оператора вида:

$$\varphi_i(x) = x_i + \lambda f_i(x) \quad (2.2.1.5)$$

где λ— числовой параметр, обычно не превышающий по модулю 0,5 и подбираемый таким образом, чтобы обеспечить сходимость процесса. что делает итерации устойчивыми. В качестве значений λ для корней используются соответственно значения две пары значений: 0.02, -0.1 и -0.05, 0.1

Критерий остановки метода формулируется через разность двух соседних векторов:

$$\| x^{(k+1)} - x^{(k)} \| < \varepsilon \quad (2.2.1.6)$$

Метод простых итераций для систем нелинейных уравнений отличается простотой реализации и отсутствием необходимости вычислять или обращать матрицу Якоби, что делает его удобным для задач, где вычисление производных затруднено. Однако скорость сходимости обычно ниже, чем у методов Ньютона и секущих, и сильно зависит от выбора функции φ и параметра λ. Несмотря на это, метод широко применяется, особенно как предварительный или вспомогательный метод в сложных вычислительных алгоритмах.

В листинге 2.2.1.2 представлена реализация метода простых итераций для системы. Там используется функция check_convergence, описанная выше.

Листинг 2.2.1.2. Реализация проверки сходимости метода простых итераций

```
def simple_iteration(x1, x2, eps=1e-4):
    if not check_convergence(x1, x2):
        raise Exception("Метод неприменим")

    it = 0
    while True:
        it += 1
        x1_new = phi1(x1, x2)
        x2_new = phi2(x1, x2)

        if max(abs(x1 - x1_new), abs(x2 - x2_new)) <= eps:
            return x1_new, x2_new, it
        x1 = x1_new
        x2 = x2_new

        if it > MAX_ITERS:
            raise Exception(f"Метод не сошелся за {it} итераций")
```

На рисунках 13, 14 представлены результаты работы метода простых итераций для первого и для второго корней.

```
Простой итерации: x1=-1.741275, x2=-1.004052, итераций=15
f1=0.0, f2=-0.0
```

Рисунок 13, результаты работы метода простых итераций для системы (первый корень)

```
Простой итерации: x1=1.256476, x2=0.802851, итераций=15
f1=-0.0, f2=0.0
```

Рисунок 14, результаты работы метода простых итераций для системы (второй корень)

2.2.2 Метод Зейделя

Метод Зейделя для решения систем нелинейных уравнений является модификацией метода простых итераций и основан на последовательном уточнении компонент вектора неизвестных. Пусть задана система нелинейных уравнений:

$$\begin{cases} f_1(x_1, x_2, \dots, x_n) = 0 \\ f_2(x_1, x_2, \dots, x_n) = 0 \\ \vdots \\ f_n(x_1, x_2, \dots, x_n) = 0 \end{cases} \quad (2.2.2.1)$$

где $x = (x_1, x_2, \dots, x_n)^T$.

Система преобразуется к эквивалентному виду:

$$\begin{cases} x_1 = \varphi_1(x_1, x_2, \dots, x_n) \\ x_2 = \varphi_2(x_1, x_2, \dots, x_n) \\ \vdots \\ x_n = \varphi_n(x_1, x_2, \dots, x_n) \end{cases} \quad (2.2.2.2)$$

что позволяет организовать итерационный процесс. В методе Зейделя новые значения компонент подставляются в последующие уравнения немедленно по мере вычисления, что ускоряет сходимость по сравнению с обычным методом простых итераций.

Итерационная схема метода имеет вид:

$$\begin{aligned} x_1^{(k+1)} &= \varphi_1(x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)}), \\ x_2^{(k+1)} &= \varphi_2(x_1^{(k+1)}, x_2^{(k)}, \dots, x_n^{(k)}), \\ &\vdots \\ x_n^{(k+1)} &= \varphi_n(x_1^{(k+1)}, x_2^{(k+1)}, \dots, x_{n-1}^{(k+1)}, x_n^{(k)}). \end{aligned} \quad (2.2.2.3)$$

Для построения итерационного процесса часто используется функция вида (2.2.1.5).

В методе Зейделя обновлённые значения подставляются в формулы сразу после их вычисления.

Критерий завершения итераций обычно задаётся условием (2.2.1.6), как и в методе простых итераций. Условие сходимости также такое же, как и в методе простых итераций (2.2.1.4)

Метод Зейделя, по сравнению с методом простых итераций, часто обладает лучшей скоростью сходимости, однако сохраняет требования к корректному выбору итерационной функции и параметра λ. Он широко применяется благодаря своей эффективности и относительной простоте реализации.

В листинге 2.2.2.3 представлена реализация метода Зейделя.

Листинг 2.2.2.3. Реализация метода Зейделя

```
def seidel_method(x1, x2, eps=1e-4):
    if not check_convergence(x1, x2):
        raise Exception("Метод неприменим")

    it = 0
    while True:
        it += 1
        x1_new = phi1(x1, x2)
        x2_new = phi2(x1_new, x2)

        if max(abs(x1 - x1_new), abs(x2 - x2_new)) <= eps:
            return x1_new, x2_new, it
        x1 = x1_new
        x2 = x2_new

        if it > MAX_ITERS:
            raise Exception(f"Метод не сошелся за {it} итераций")
```

На рисунке 15, 16 представлены результаты работы метода Зейделя для первого и для второго корней.

```
Метод зейделя: x1=-1.741348, x2=-1.004040, итераций=16
f1=0.0, f2=-0.0
```

Рисунок 15, результаты работы метода Зейделя для системы (первый корень)

```
Метод зейделя: x1=1.256432, x2=0.802881, итераций=16
f1=-0.0, f2=0.0
```

Рисунок 16, результаты работы метода Зейделя для системы (второй корень)

2.2.3 Метод Ньютона

Метод Ньютона для решения системы нелинейных уравнений является обобщением одноимённого метода для одного уравнения. Пусть дана система нелинейных уравнений (2.2.2.1), которую представим в виде $F(x) = 0$.

Основная идея метода Ньютона состоит в том, чтобы на каждом шаге аппроксимировать систему линейной системой, используя разложение Тейлора для функций. На основе этого на каждом шаге метода вычисляется новый вектор приближений как:

$$\mathrm{x}^{(k+1)} = \mathrm{x}^{(k)} - [\mathrm{J}(\mathrm{x}^{(k)})]^{-1}\mathrm{F}(\mathrm{x}^{(k)}), \quad (2.2.3.1)$$

где J(x)— матрица Якоби системы, которая вычисляется как:

$$J(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix} \quad (2.2.3.2)$$

Этот процесс повторяется до тех пор, пока изменение вектора приближений на двух последовательных шагах не станет меньше заданной точности:

$$\| \mathrm{x}^{(k+1)} - \mathrm{x}^{(k)} \| < \varepsilon \quad (2.2.3.3)$$

Проверка условия сходимости для метода ньютона при решении систем нелинейных уравнений довольно-таки сложна, поэтому в практических задачах обычно ее не выполняют.

В листинге 2.2.3.1 представлена реализация метода Ньютона.

Листинг 2.2.3.1. Реализация метода Ньютона для решения систем нелинейных уравнений

```
def newton_method(x1, x2, eps=1e-4):
    J = [[df1_dx1(x1, x2), df1_dx2(x1, x2)],
         [df2_dx1(x1, x2), df2_dx2(x1, x2)]]

    det = J[0][0] * J[1][1] - J[0][1] * J[1][0]
    if abs(det) == 0:
        raise Exception("Матрица Якоби вырождена!")

    it = 0
    x1_pred = x1
    x2_pred = x2
    while True:
        it += 1
        ff1 = f1(x1, x2)
```

```
ff2 = f2(x1, x2)

dx = (ff1 * J[1][1] - ff2 * J[0][1]) / det
dy = (J[0][0] * ff2 - J[1][0] * ff1) / det
x1 -= dx
x2 -= dy
if max(abs(x1 - x1_pred), abs(x2 - x2_pred)) <= eps:
    return x1, x2, it
x1_pred = x1
x2_pred = x2

if it > MAX_ITERS:
    raise Exception(f"Метод не сошелся за {it} итераций")
```

На рисунке 17, 18 представлены результаты работы метода Ньютона для первого и для второго корней.

```
Метод Ньютона: x1=-1.740926, x2=-1.003845, итераций=2
f1=-0.0, f2=-0.0
```

Рисунок 17, результаты работы метода Ньютона для системы (первый корень)

```
Метод Ньютона: x1=1.256712, x2=0.803014, итераций=3
f1=-0.0, f2=-0.0
```

Рисунок 18, результаты работы метода Ньютона для системы (второй корень)

2.3 Вывод

Практическая реализация и сравнение методов решения нелинейных уравнений позволили глубоко изучить их поведение, сильные стороны и принципиальные ограничения. Работа была разделена на две ключевые части: решение скалярных уравнений и систем нелинейных уравнений.

При решении одного нелинейного уравнения все рассмотренные методы показали зависимость результата от выбора начального приближения, но с разной степенью критичности. Метод простой итерации оказался самым простым в реализации, но и самым капризным. Его сходимость и скорость полностью определяются удачным выбором эквивалентного преобразования уравнения `x = φ(x)`. На практике часто требовалось выполнять несколько проб, чтобы найти сходящийся итерационный процесс, что делает метод не самым надежным инструментом без предварительного анализа.

Метод Ньютона продемонстрировал эталонную квадратичную скорость сходимости вблизи корня, что делало его самым быстрым из всех испытанных методов. Однако эта мощь имеет свою цену: метод требователен к начальному приближению (при неудачном выборе может легко разойтись) и требует вычисления производной на каждом шаге. Его реализация на практике сталкивается с двумя проблемами: необходимостью аналитического задания производной (что не всегда удобно) и потенциальными ошибками, если производная близка к нулю в окрестности итераций.

Методы секущих и хорд стали компромиссным решением. Они сохраняют высокую скорость сходимости (сверхлинейную для секущих), но избавляют от необходимости вычислять производную аналитически, заменяя ее разностной аппроксимацией (секущие) или используя фиксированный наклон (хорды). Метод секущих на тестах часто сходился почти так же быстро, как Ньютон, но оказался чуть менее стабильным. Метод хорд, напротив, проявил себя как более надежный и устойчивый, особенно при работе на фиксированном интервале, гарантирующем наличие корня, хотя и сошёлся за большее число итераций. Эти методы оказались наиболее практичными для случаев, когда производную вычислить сложно или невозможно.

Переход к системам нелинейных уравнений существенно усложнил задачу. Метод простых итераций для систем унаследовал все проблемы скалярного случая: необходимость построения сходящегося итерационного процесса, которое на практике часто превращалось в нетривиальную задачу. Его сходимость наблюдалась только при очень слабой нелинейности и диагональном преобладании в соответствующей матрице Якоби.

Метод Зейделя для систем, как и для СЛАУ, показал ускорение сходимости по сравнению с простыми итерациями за счет использования обновленных значений переменных на текущей же итерации, но не избавил от фундаментальных проблем сходимости.

Наиболее мощным и универсальным инструментом для систем подтвердил себя метод Ньютона. Его реализация, хотя и была самой трудоемкой (требовалось вычислять матрицу Якоби и решать вспомогательную СЛАУ на каждом шаге), обеспечивала быструю и точную сходимость при удачном начальном векторе. Главными практическими сложностями стали: трудоемкость вычисления или аппроксимации матрицы частных производных, высокие вычислительные затраты на каждом шаге, чувствительность к начальному приближению. В ходе работы стало очевидно, что успешное применение метода Ньютона часто требует

предварительного «грубого» нахождения области решения с помощью более простых, но устойчивых методов.

Итоговый вывод по разделу заключается в следующем: для одиночных уравнений метод Ньютона является оптимальным по скорости, когда производная доступна, а начальное приближение известно. Методы секущих и хорд — отличные практичные замены. Для систем уравнений метод Ньютона, несмотря на свою сложность, остается основным рабочим инструментом, тогда как итерационные методы служат скорее для анализа или решения специальных, хорошо обусловленных задач.

3 Интерполяция и аппроксимация

3.1 Задание 9. Интерполяция многочленами Лагранжа

Для таблично заданной функции построить интерполяционные многочлены Лагранжа второй и третьей степени. Проверить многочлен в любой узловой точке. Вычислить значение функции в точке x^*, учитывая положение заданной точки на интервале.

Вариант 32 $x^* = 4{,}016$

i	0	1	2	3	4	5	6	7	8
x_i	3,05	3,43	3,81	4,19	4,57	4,95	5,33	5,71	6,09
y_i	1,8571	2,1247	3,6456	2,6842	2,3539	0,3431	1,6577	2,8982	1,4326

Многочлен Лагранжа $L_n(x)$степени n строится как сумма произведений табличных значений f_iна базисные полиномы Лагранжа $l_i(x)$.

Общая формула многочлена Лагранжа:

$$L_n(x) = \sum_{i=0}^{n} f_i \cdot l_i(x) \tag{3.1.1}$$

Базисный полином Лагранжа:

$$l_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} \tag{3.1.2}$$

Базисный полином Лагранжа обладает свойством:

$$l_i(x_j) = \begin{cases} 1, & \text{при } i = j, \\ 0, & \text{при } i \neq j \end{cases} \tag{3.1.3}$$

Остаточный член многочлена Лагранжа имеет вид:

$$R_n(x) = \frac{max \mid f^{(n+1)}(\xi) \mid}{(n+1)!} \omega_{n+1}(x), \xi \in [a, b] \tag{3.1.4}$$

где $\omega_{n+1}(x)$ имеет вид:

$$\omega_{n+1}(x) = (x - x_0)(x - x_1)(x - x_2) \cdots (x - x_n) \tag{3.1.5}$$

Функция $l_i(x, x_vals, i)$реализует вычисление базисного полинома $l_i(x)$, код которой приведён в листинге 3.1.1.

Листинг 3.1.1

```
def l_i(x, x_vals, i):
```

```
    """
    Lagrange basis polynomial l_i(x).
    """
    n = len(x_vals)
    prod = 1.0
    for j in range(n):
        if j != i:
            prod *= (x - x_vals[j]) / (x_vals[i] - x_vals[j])
    return prod
```

Функция lagrange реализует общую формулу $L_n\ (x)$ путём суммирования $y_i * l_i\ (x)$ код которой приведен в листинге 3.1.2.

Листинг 3.1.2

```
def lagrange(x_vals, y_vals, x):
    """
    Interpolation polynomial L_n(x).
    """
    n = len(x_vals)
    result = 0.0
    for i in range(n):
        result += y_vals[i] * l_i(x, x_vals, i)
    return result
```

Для проверки многочлена, его значение вычисляется в одной из узловых точек x_k, где оно должно совпадать с y_i.

В программе вычисляются значения многочлена Лагранжа в точке интерполяции для всех возможных вариантов выбора наборов x_i, учитывая расположение точки x^*. Для многочлена второй степени подходящие варианты узловых точек: x_1, x_2, x_3; x_2, x_3, x_4.

Для многочлена третьей степени подходят варианты x_0, x_1, x_2, x_3; x_1, x_2, x_3, x_4; x_2, x_3, x_4, x_5.

Результаты работы программы показаны на рисунках 19, 20.

```
Value of L_2(x*) = 3.43250712326869 94
Check: L_2(3.81) = 3.6456 (expected: 3.6456)
|ω(x*)| = 0.021004584000000038

Value of L_2(x*) = 3.0460919238227158
Check: L_2(4.19) = 2.6842 (expected: 2.6842)
|ω(x*)| = 0.019857576000000047
```

Рисунок 19, результаты вычисления через многочлен Лагранжа (вторая степень).

```
Value of L_3(x*) = 2.906668135486224
Check: L_3(4.19) = 2.6842 (expected: 2.6842)
|ω(x*)| = 0.018546975984000005

Value of L_3(x*) = 3.233876152325413
Check: L_3(3.81) = 3.6456 (expected: 3.6456)
|ω(x*)| = 0.011636539536000027

Value of L_3(x*) = 3.670833665009478
Check: L_3(3.43) = 2.1247 (expected: 2.1247)
|ω(x*)| = 0.020290428144000004
```

Рисунок 20 результаты вычисления через многочлен Лагранжа (третья степень).

Графическая интерпретация результатов показана на рисунках 21, 22.

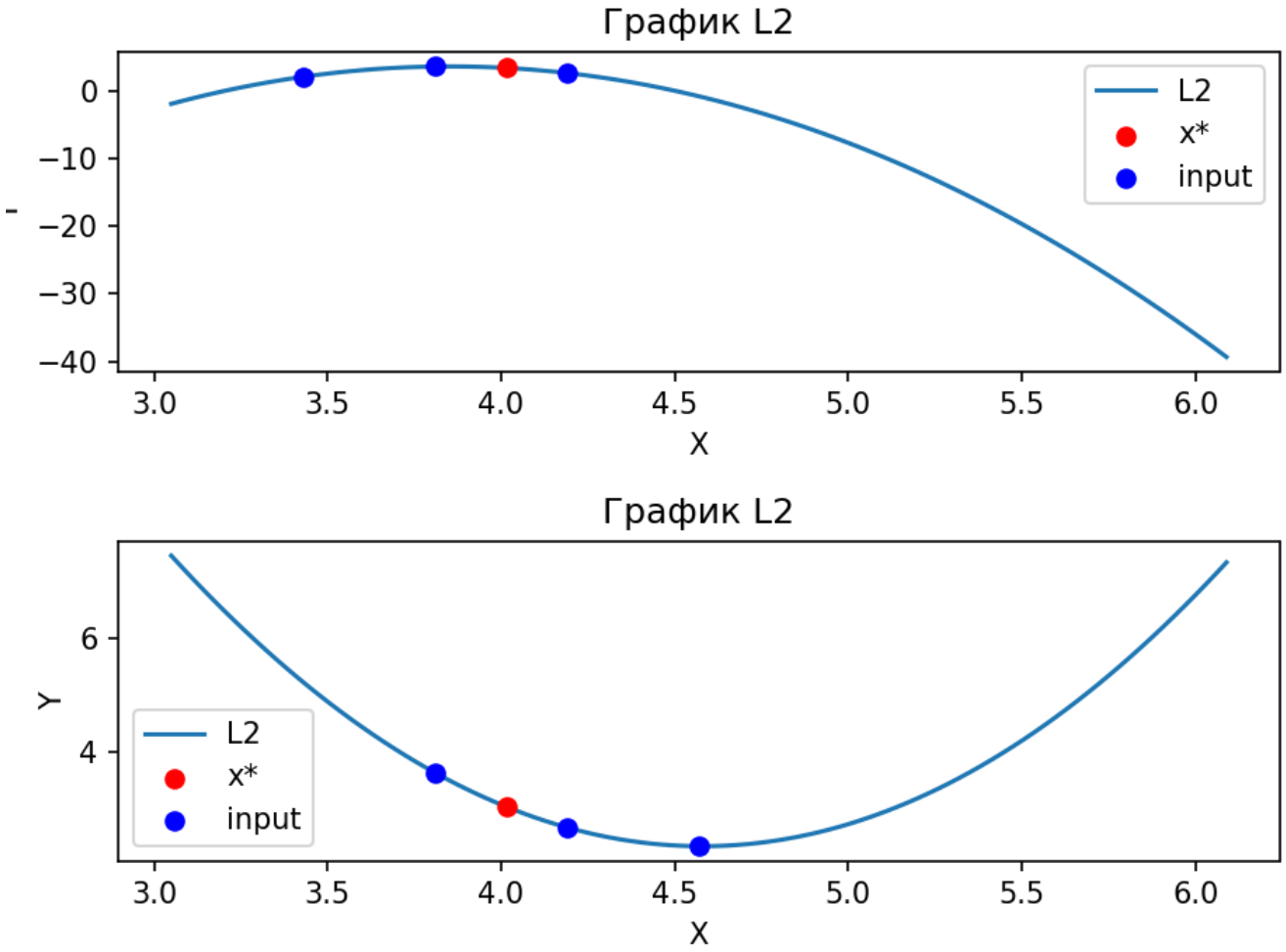

Рисунок 21, графики интерполяционных многочленов Лагранжа (2 степень)

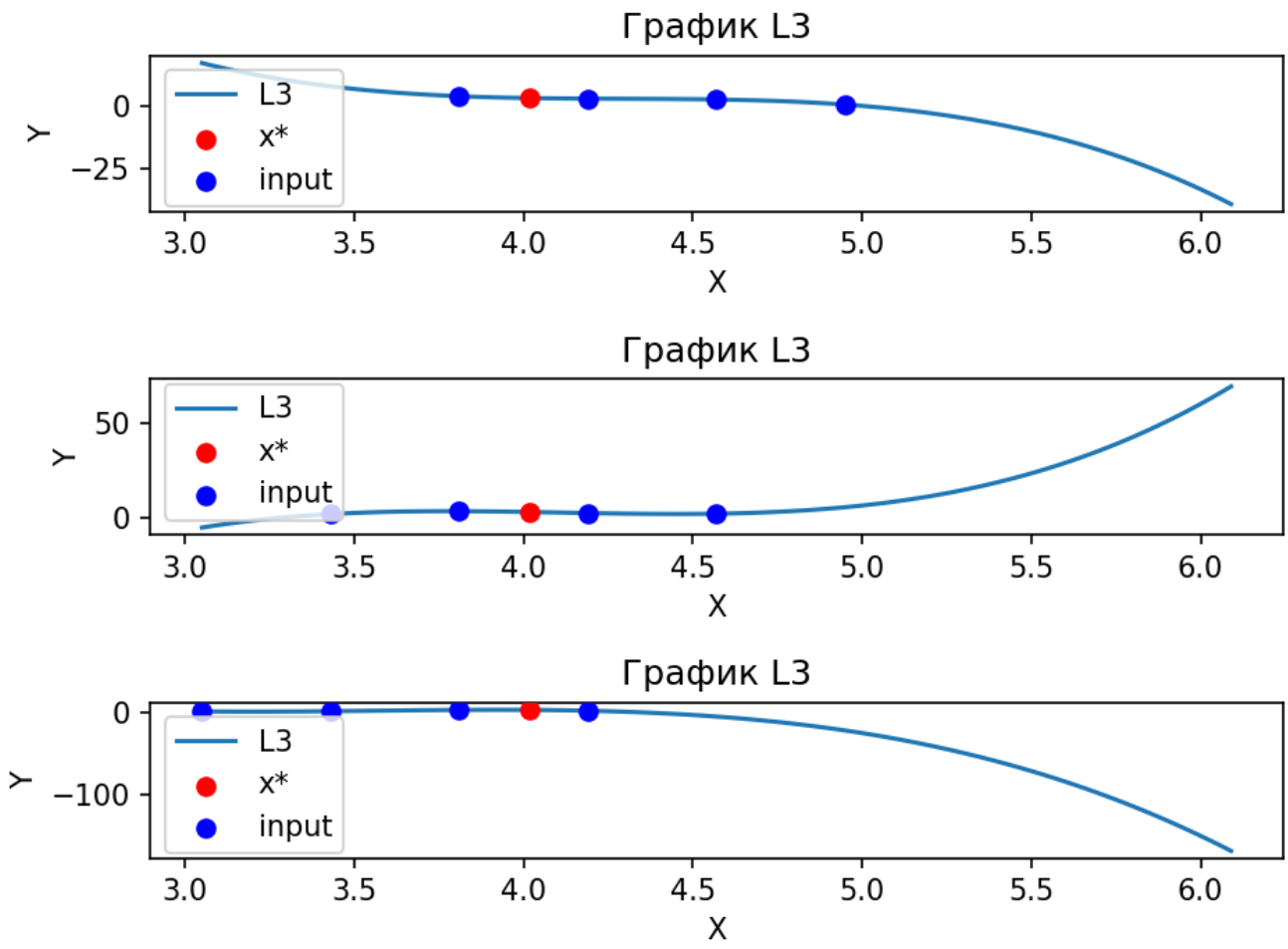

Рисунок 22, графики интерполяционных многочленов Лагранжа (3 степень)

3.2 Интерполяционный многочлен Ньютона

Многочлен Ньютона используется для интерполяции данных, и его форма отличается от многочлена Лагранжа. Он строится с использованием разностей разделённого разложения, что позволяет эффективно вычислять коэффициенты многочлена, используя табличные значения.

Общая формула многочлена Ньютона для степени n имеет вид:

$$P_n(x) = f_0 + (x - x_0) \cdot \Delta f_0 + (x - x_0)(x - x_1) \cdot \Delta^2 f_0 + \cdots + (x - x_0)(x - x_1) \cdot \cdots \cdot (x - x_{n-1}) \cdot \Delta^n f_0 \quad (3.2.1)$$

где $\Delta^k f_0$— это k-ая разделённая разность для значений функции в узловых точках.

Многочлен Ньютона второй степени строится по формуле:

$$P_2(x) = f_0 + (x - x_0) \cdot \Delta f_0 + (x - x_0)(x - x_1) \cdot \Delta^2 f_0 \quad (3.2.2)$$

где Δf_i имеет вид:

$$\Delta f_0 = \frac{f_1 - f_0}{x_1 - x_0}, \Delta^2 f_0 = \frac{\Delta f_1 - \Delta f_0}{x_2 - x_0}, \Delta^3 f_0 = \frac{\Delta^2 f_1 - \Delta^2 f_0}{x_3 - x_0} \quad (3.2.3)$$

Для многочлена третьей степени подходящие узловые точки: x_0, x_1, x_2, x_3.

Многочлен Ньютона третьей степени строится по формуле:

$$P_3(x) = f_0 + (x - x_0) \cdot \Delta f_0 + (x - x_0)(x - x_1) \cdot \Delta^2 f_0 + (x - x_0)(x - x_1)(x - x_2) \cdot \Delta^3 f_0 \quad (3.2.4)$$

Ошибка для полиномов Ньютона оценивается так же, как и для полинома Лагранжа. Можно записать это следующим образом через разделенные разности:

$$\Delta^n f_0 \cdot \omega_{n+1}(x) \quad (3.2.5)$$

$\omega_{n+1}(x)$ имеет вид (3.1.5)

На листинге 3.2.1 представлена функция построения полинома Ньютона.

Листинг 3.2.1. Функция построения полинома Ньютона

```
def build_newton(x_t, y_t, x_star, start, stop, degree):
    x_quad = x_t[start:stop]
    y_quad = y_t[start:stop]

    x_check = x_quad[1]
    y_check = y_quad[1]

    dd = divided_differences(x_quad, y_quad)
    p2_star = newton_eval(x_quad, dd, x_star, degree)
    p2_check = newton_eval(x_quad, dd, x_check, degree)

    x = np.linspace(x_t[0], x_t[-1], 100)
    y = []
    for i in x:
        y.append(newton_eval(x_quad, dd, i, degree))

    err = get_newton_error(x_quad, x_star, degree)
    print(f"Value of L_{degree}(x*) = {p2_star}")
    print(f"Check: L_{degree}({x_check}) = {p2_check} (expected:
{y_check})")
    print(f"Approximate error  = {err}")

    return x, y, err, p2_star, x_quad, y_quad
```

На листинге 3.2.2 представлена функция вычисления разделенных разностей, необходимых для построения многочлена Ньютона.

Листинг 3.2.2. Функция вычисления разделенных разностей

```
def divided_differences(x_vals, y_vals):
    """
    Computes the divided difference table.
    dd[k][j] = f[x_j, ..., x_{j+k}]
    """
    m = len(x_vals)
    dd = [[0.0] * m for _ in range(m)]
    for j in range(m):
        dd[0][j] = y_vals[j]
    for k in range(1, m):
        for j in range(m - k):
            dd[k][j] = (dd[k - 1][j + 1] - dd[k - 1][j]) / (x_vals[j + k] - x_vals[j])
    return dd
```

Результаты работы программы для вычисления многочлена Ньютона второй и третьей степени показаны на рисунке 23 и рисунке 24 соответственно.

```
Value of L_2(x*) = 4.993956879501385
Check: L_2(3.43) = 2.1247 (expected: 2.1247)
Approximate error  = 0.1166116559999998
Value of L_2(x*) = 3.43250712326869 94
Check: L_2(3.81) = 3.6456 (expected: 3.6456)
Approximate error  = 0.02100458400000038
Answer with min err is 3.43250712326869 94
```

Рисунок 23, графики интерполяционных многочленов Ньютона (2 степень)

```
Value of L_3(x*) = 3.6708336650094777
Check: L_3(3.43) = 2.1247 (expected: 2.1247)
Approximate error  = 0.020290428144000004
Value of L_3(x*) = 3.233876152325413
Check: L_3(3.81) = 3.6456 (expected: 3.6456)
Approximate error  = 0.011636539536000027
Value of L_3(x*) = 2.906668135486223
Check: L_3(4.19) = 2.6842 (expected: 2.6842)
Approximate error  = 0.01854697598400005
Answer with min err is 3.233876152325413
```

Рисунок 24, графики интерполяционных многочленов Ньютона (3 степень)

На рисунках 25, 26 приведены графики полиномов Ньютона, построенные на различных точках.

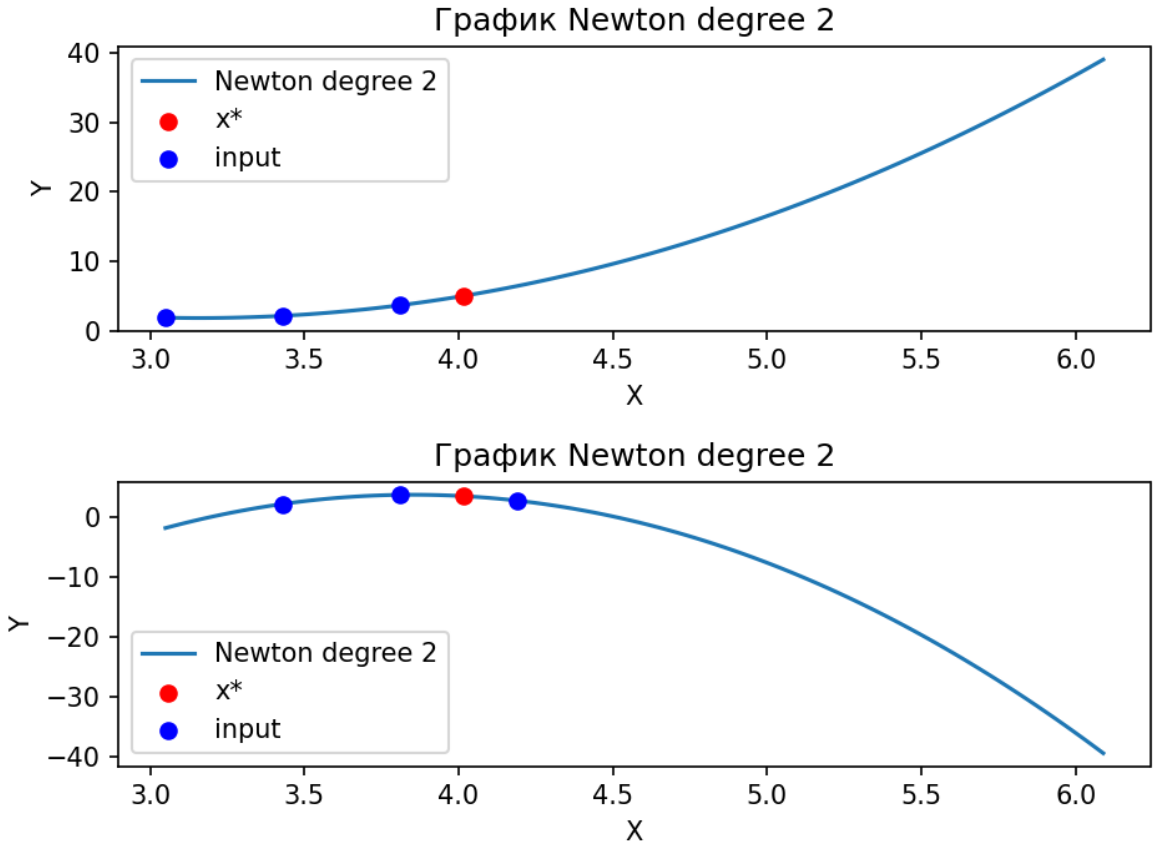

Рисунок 25, график многочленов Ньютона второй степени

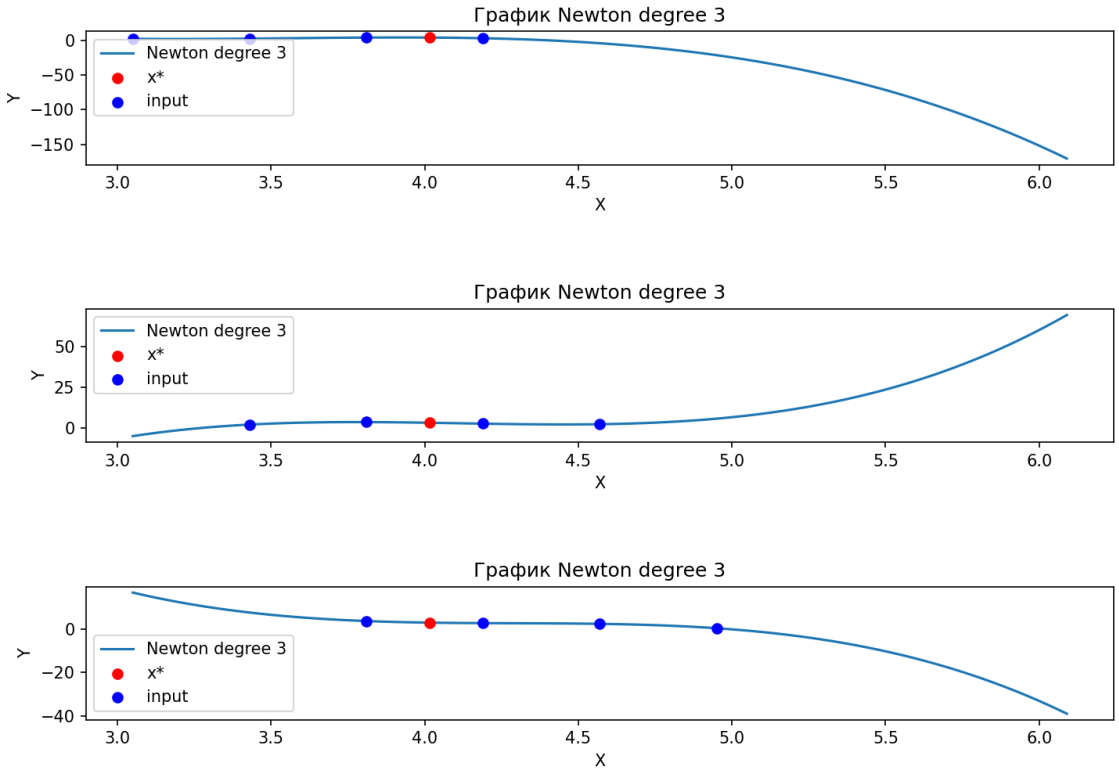

Рисунок 26, график многочленов Ньютона третьей степени

3.3 Сплайны

Для таблично заданной функции с неравномерной сеткой построить естественный кубический сплайн дефекта 1. Вычислить значение сеточной функции в заданной точке x^* Указать коэффициенты сплайна на отрезке, включающим точку x^*.

Вариант 32 $x^* = 1{,}045$

i	0	1	2	3	4	5	6	7	8	9	10
x_i	0,25	0,488	0,760	1,168	1,576	1,882	2,120	2,494	2,902	3,344	3,65
y_i	0,853	1,047	0,498	0,354	0,216	0,052	0,314	0,792	0,921	1,157	0,824

Сплайн — это определенный на некотором интервале многочлен n-й степени, непрерывно l раз дифференцируемый на каждом промежутке между узлами. Разность между степенью сплайна n и показателем его гладкости l называется дефектом сплайна.

Для построения естественного кубического сплайна с краевыми условиями, дан отрезок $[a, b]$, узлы $x_0, x_1, \dots, x_n$ и значения $y_0, y_1, \dots, y_n$ в этих узлах. На каждом интервале $[x_{i-1}, x_i]$ требуемый сплайн имеет вид полинома третьей степени:

$$S_3(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3 \qquad (3.3.1)$$

где $i = 1,2,\dots,n$. Для нахождения коэффициентов a_i, b_i, c_i, и d_iнеобходимо удовлетворить краевым условиям. Рассмотрим их немного более подробно.

Первым краевым условием является задание первых производных на концах:

$$S_3'(x_0) = f'(a), S_3'(x_n) = f'(b) \tag{3.3.2}$$

Вторым краевым условием является задание вторых производных на концах:

$$S_3''(x_0) = f''(a), S_3''(x_n) = f''(b) \tag{3.3.3}$$

Третьим краевым условием является задание первых производны на концах каждого интервала:

$$S_3'(x_0) = S_3'(x_n), S_3''(x_0) = S_3''(x_n) \tag{3.3.4}$$

Четвертое краевое условие задается в разной форме, в данной конкретной задаче, так как мы строим естественный сплайн, оно заключается в следующем. Для естественных краевых условий устанавливаются нулевые значения второй производной на концах интервала, то есть $S_3''(x_0) = 0$ и $S_3''(x_n) = 0$.

На основании данных краевых условий, можно построить трехдиагональную систему уравнений, которая имеет вид:

$$\begin{cases} c_1 = 0, \\ 2(h_1 + h_2)c_2 + h_2c_3 = 3\left[\frac{y_2 - y_1}{h_2} - \frac{y_1 - y_0}{h_1}\right], \\ h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_ic_{i+1} = 3\left[\frac{y_i - y_{i-1}}{h_i} - \frac{y_{i-1} - y_{i-2}}{h_{i-1}}\right], \\ \text{для } i = 3,4,\dots,n-1, \\ h_{n-1}c_{n-1} + 2(h_{n-1} + h_n)c_n = 3\left[\frac{y_n - y_{n-1}}{h_n} - \frac{y_{n-1} - y_{n-2}}{h_{n-1}}\right]. \end{cases} \tag{3.3.5}$$

Ее можно решить методом прогонки и найти коэффициенты c. После нахождения c_i, можно вычислить остальные коэффициенты. Для всех отрезков с $i = 1,2,\dots,n-1$значения коэффициентов вычисляются по формулам:

$$a_i = y_{i-1}, \tag{3.3.6}$$

$$b_i = \frac{y_i - y_{i-1}}{h_i} - \frac{1}{3}h_i(c_{i+1} + 2c_i), \tag{3.3.7}$$

$$d_i = \frac{c_{i+1} - c_i}{3h_i}. \tag{3.3.8}$$

Для последнего отрезка ($i = n$) коэффициенты вычисляются как:

$$d_n = -\frac{c_n}{3h_n}. \tag{3.3.9}$$

Таким образом, можно построить естественный кубический сплайн, решив трёхдиагональную систему для c_iи вычислив остальные коэффициенты. Реализация сплайна состоит из нескольких этапов. В листинге 3.3.1 приведен код, реализующий естественный кубический сплайн дефекта 1.

Листинг 3.3.1

```
    def main():
    x = [0.25, 0.488, 0.760, 1.168, 1.576, 1.882, 2.120, 2.494, 2.902,
3.344, 3.65]
    y = [0.853, 1.047, 0.498, 0.354, 0.216, 0.052, 0.314, 0.792, 0.921,
1.157, 0.824]
    x_star = 1.045
    n = len(x) - 1  # number of segments

    # --- Step 1: Calculate step sizes h_i ---
    # Using 1-based indexing for h, A, B, C, D for convenience
    # h[i] = x_i - x_{i-1}
    h = [0.0] * (n + 1)
    for i in range(1, n + 1):
        h[i] = x[i] - x[i - 1]

    # --- Step 2: Form the SLAE for c_i ---
    # We are solving the system for c_2, ..., c_n (n-1 unknowns)
    # c_1 = 0 by the natural spline condition
    N_system = n - 1  # System size

    a_tdma = [0.0] * N_system  # Lower diagonal
    b_tdma = [0.0] * N_system  # Main diagonal
    c_tdma = [0.0] * N_system  # Upper diagonal
    d_tdma = [0.0] * N_system  # Right-hand side

    # Calculate the RHS: D_i = 3 * ( (y_i - y_{i-1})/h_i - (y_{i-1} -
y_{i-2})/h_{i-1} )
    RHS = [0.0] * (n + 1)
    for i in range(2, n + 1):
        term1 = (y[i] - y[i - 1]) / h[i]
        term2 = (y[i - 1] - y[i - 2]) / h[i - 1]
        RHS[i] = 3 * (term1 - term2)
```

```
# Filling A and d matrices for TDMA
for i_tdma in range(N_system):
    i = i_tdma + 2

    # Main diagonal: 2 * (h_{i-1} + h_i)
    b_tdma[i_tdma] = 2 * (h[i - 1] + h[i])

    # Right-hand side
    d_tdma[i_tdma] = RHS[i]

    # Lower diagonal: h_{i-1}
    if i_tdma > 0:  # a_tdma[0] is unused
        a_tdma[i_tdma] = h[i - 1]

    # Upper diagonal: h_i
    if i_tdma < N_system - 1:  # c_tdma[N_system-1] is unused
        c_tdma[i_tdma] = h[i]

# --- Step 3: Solve the SLAE ---
# c_solved contains [c_2, c_3, ..., c_n]
c_solved = solve_tdma(a_tdma, b_tdma, c_tdma, d_tdma)

# Forming the full C array (1-based index)
C = [0.0] * (n + 1)
C[1] = 0.0  # c_1 = 0
for i in range(N_system):
    C[i + 2] = c_solved[i]
# Now C = [0.0, c_1, c_2, ..., c_n]

# --- Step 4: Calculate coefficients A, B, D ---
A = [0.0] * (n + 1)
B = [0.0] * (n + 1)
D = [0.0] * (n + 1)

for i in range(1, n + 1):
    # a_i = y_{i-1}
    A[i] = y[i - 1]

    if i < n:
        # d_i = (c_{i+1} - c_i) / (3 * h_i)
        D[i] = (C[i + 1] - C[i]) / (3.0 * h[i])
        # b_i = (y_i - y_{i-1})/h_i - h_i/3 * (c_{i+1} + 2*c_i)
```

```
            B[i] = (y[i] - y[i - 1]) / h[i] - (h[i] / 3.0) * (C[i + 1] +
2.0 * C[i])
        else:  # i == n (last segment)
            # d_n = -c_n / (3 * h_n)
            D[n] = -C[n] / (3.0 * h[n])
            # b_n = (y_n - y_{n-1})/h_n - 2*h_n/3 * c_n
            B[n] = (y[n] - y[n - 1]) / h[n] - (2.0 * h[n] / 3.0) * C[n]

    # --- Step 5: Calculate S(x*) and find coefficients ---
    # Find the segment i to which x* belongs
    target_i = -1
    y_star = 0.0
    for i in range(1, n + 1):
        if x[i - 1] <= x_star <= (x[i]):
            target_i = i
            break

    # Theoretical error factor
    # |R(x)| <= (5/384) * H^4 * max|f^(4)(x)|
    h = [0.0] * (n + 1)
    H = 0.0
    for i in range(1, n + 1):
        h[i] = x[i] - x[i - 1]
        if h[i] > H:
            H = h[i]
    H_power_4 = H ** 4
    max_error_factor = H_power_4
```

На рисунке 27 приведен график сплайнов, интерполирующих заданную табличную функцию.

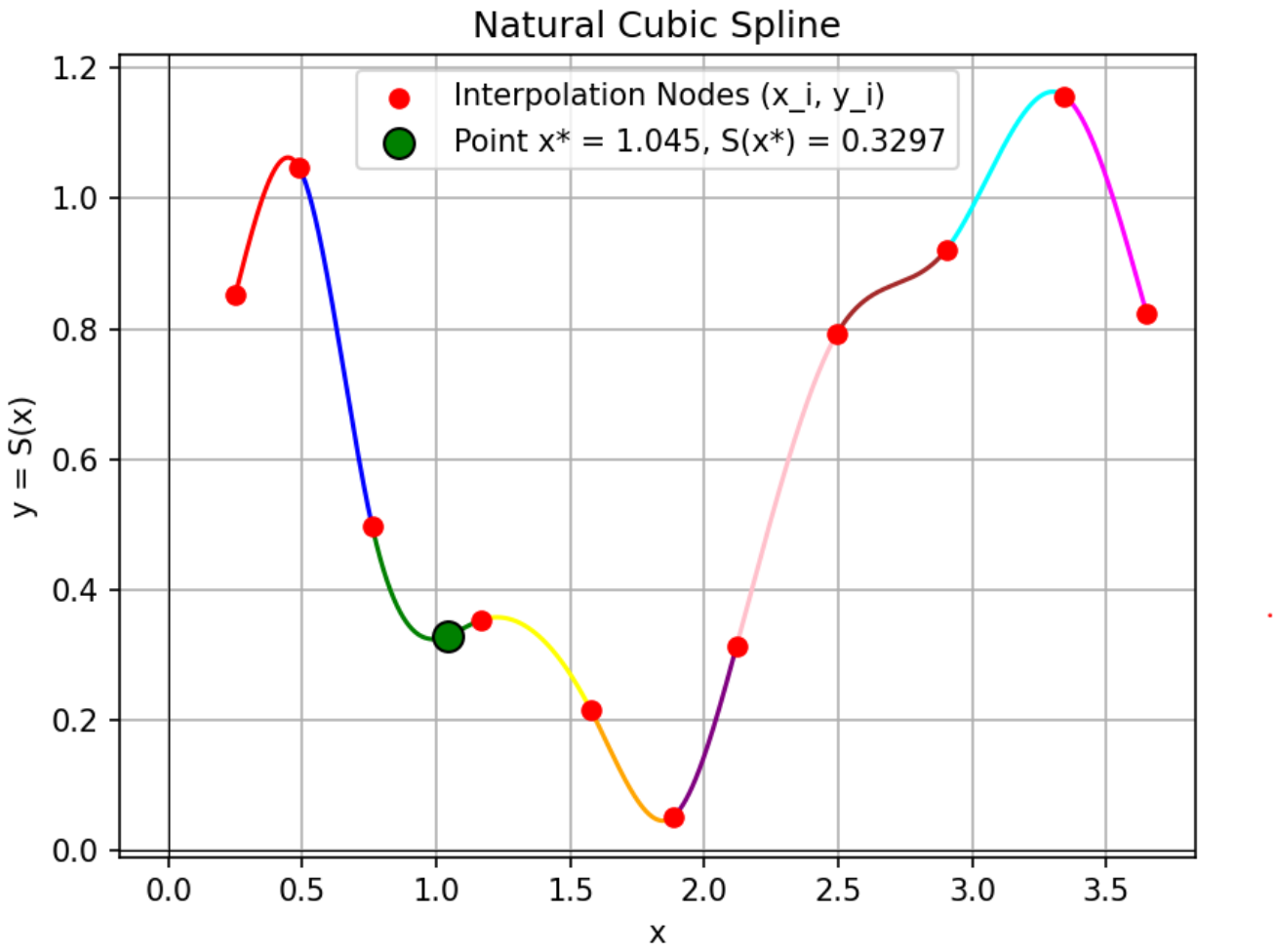

Рисунок 27, график сплайнов

3.4 Метод наименьших квадратов

Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены первой, второй и третьей степени. Для каждого из приближающих многочленов вычислить сумму квадратов ошибок. Построить общий график всех многочленов и приближаемой функции. Вычислить значения всех приближающих многочленов в точке x^*.

Вариант 32 $x^* = -1{,}338$

i	0	1	2	3	4	5	6	7	8	9
x_i	−3,17	−2,54	−1,91	−1,28	−0,65	−0,02	0,61	1,24	1,87	2,50
y_i	2,2457	3,7921	4,8164	4,6415	3,5386	0,2581	−3,8714	−5,9872	−6,7853	−4,9621

Метод наименьших квадратов (МНК) является одним из самых широко используемых методов для решения задач аппроксимации, когда требуется найти наилучшее приближение для данных. Этот метод позволяет минимизировать сумму квадратов отклонений между заданными значениями и аппроксимирующей функцией, что делает его полезным инструментом для обработки данных, подверженных шуму или погрешности измерений. Основное его преимущество

заключается в том, что он предоставляет эффективный способ нахождения параметров модели, которые наилучшим образом соответствуют данным.

Общая цель метода заключается в нахождении такого приближения функции f(x), которое минимизирует ошибку аппроксимации, выраженную через сумму квадратов отклонений. Для этого предполагается, что есть набор данных, состоящий из n точек (x_i, y_i), где x_i— это независимые переменные, а y_i— зависимые переменные, для которых нужно найти приближающую функцию.

Формула для аппроксимации функции f(x)методом наименьших квадратов имеет вид:

$$S(a) = \sum_{i=1}^{n} (y_i - f(x_i, a))^2 \qquad (3.4.1)$$

где a— вектор параметров, который необходимо определить, а $f(x_i, a)$— значение аппроксимирующей функции в точке x_i. Цель метода — минимизировать сумму квадратов отклонений между реальными значениями y_iи значениями $f(x_i, a)$.

При этом задача сводится к решению системы нормальных уравнений. Если функция f(x) линейна относительно параметров a, то для её аппроксимации минимизируется следующая функция:

$$S(a) = \sum_{i=1}^{n} (y_i - a_0 - a_1 x_i - a_2 x_i^2 - \cdots - a_m x_i^m)^2 \qquad (3.4.2)$$

Задача сводится к нахождению таких коэффициентов $a_0, a_1, \ldots, a_m$, которые минимизируют S(a). Для решения этой задачи используется метод нахождения производных функции ошибки по каждому из параметров и приравнивания их к нулю, что приводит к системе линейных уравнений:

$$\begin{pmatrix} \sum_{i=1}^{n} 1 & \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \ldots & \sum_{i=1}^{n} x_i^m \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i^3 & \ldots & \sum_{i=1}^{n} x_i^{m+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{n} x_i^m & \sum_{i=1}^{n} x_i^{m+1} & \sum_{i=1}^{n} x_i^{m+2} & \ldots & \sum_{i=1}^{n} x_i^{2m} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} y_i x_i \\ \sum_{i=1}^{n} y_i x_i^2 \\ \vdots \\ \sum_{i=1}^{n} y_i x_i^m \end{pmatrix} \qquad (3.4.3)$$

Решение этой системы уравнений даёт искомые коэффициенты $a_0, a_1, \ldots, a_m$, которые минимизируют ошибку аппроксимации.

Метод наименьших квадратов идеально подходит для задач, где требуется аппроксимировать функции с локальными особенностями, так как он позволяет сгладить колебания данных, исключив влияние случайных отклонений или ошибок. Это особенно полезно при работе с большими объёмами данных, где важна точность, но при этом необходимо исключить влияние шума, который может существенно искажать результаты.

Ниже приведен листинг кода, в котором представлена реализация метода МНК. В нем используются некоторые вспомогательные функции, например calculate_sums, позволяющая посчитать суммы, и calculate_errors, позволяющая посчитать ошибки.

Листинг 3.4.1

```
for m in m_values:
print(f"--- Polynomial of degree m = {m} ---")

# 4.1. Building the Normal System
A, B = calculate_sums(m, x_data, y_data)

print("Normal LSM System (A|B):")
for i in range(m + 1):
    equation = ""
    for j in range(m + 1):
        equation += f"{A[i][j]:.4f}*a{j} "
        if j < m:
            equation += "+ "
    equation += f"= {B[i]:.4f}"
    print(equation)

print()

# 4.2. Solving the Normal System
a_coeffs = gauss_solve(A, B)

# 4.3. Forming the Approximating Polynomial
polynomial_str = f"F_{m}(x) = "
for i, a in enumerate(a_coeffs):
    if i == 0:
        polynomial_str += f"{a:.4f}"
    elif i == 1:
        # Use just x for x^1
        term = f"{abs(a):.4f}x"
```

```
            polynomial_str += f" + {term}" if a >= 0 else f" - {term}"
        else:
            term = f"{abs(a):.4f}x^{i}"
            polynomial_str += f" + {term}" if a >= 0 else f" - {term}"

    # 4.4. Calculating Errors
    Phi_m, Sigma_m = calculate_errors(a_coeffs, m, x_data, y_data,
N_points)

    # 4.5. Calculating value at x*
    Fm_x_star = polynomial_value(a_coeffs, x_star)
```

На рисунке 28 представлены графики для первой, второй и третьей степени полиномов соответственно.

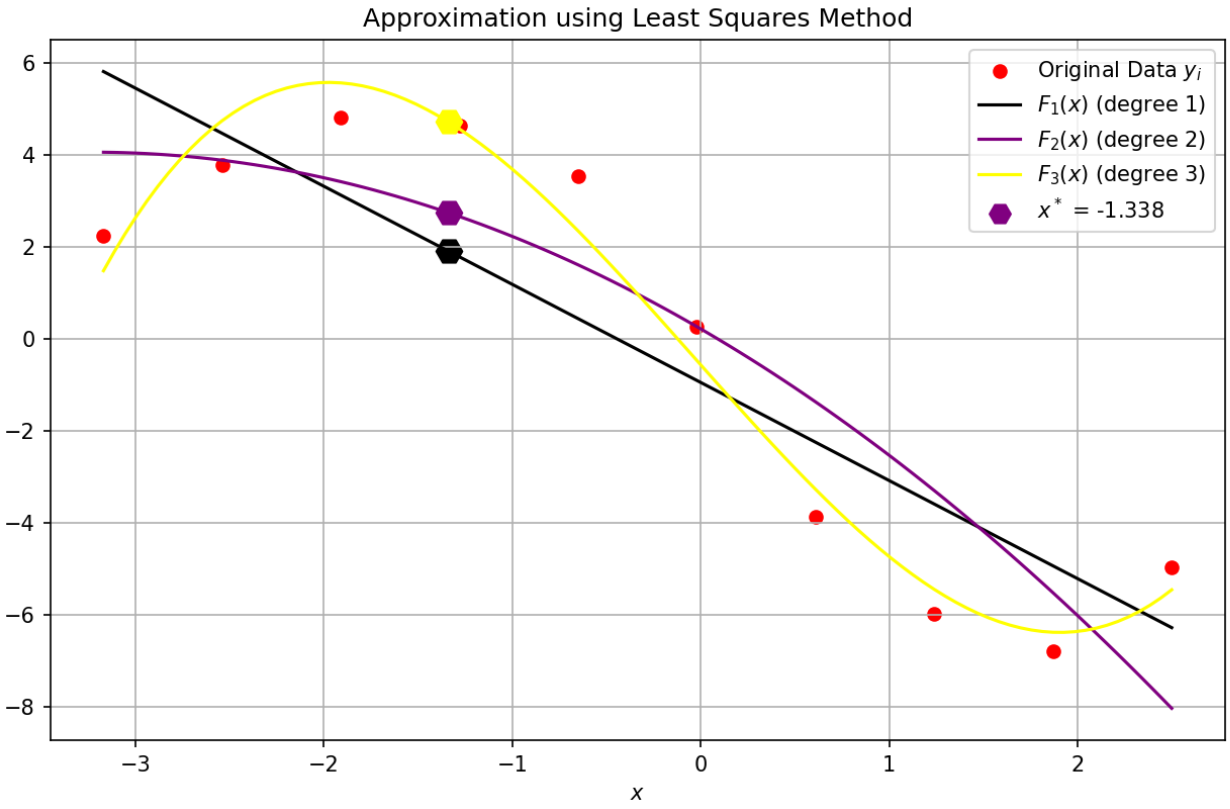

Рисунок 28, график для МНК

На рисунке 28 представлены результаты вычислений для исходных данных.

```
Sequence    Sum of Squared Errors    ASD             Value in x*
1           48.7063                  2.4674          1.9083
2           37.4122                  2.3118          2.7442
3           5.0738                   0.9196          4.7178

Approximating polynomials:
m=1: F_1(x) = -0.9460 - 2.1332x
m=2: F_2(x) = 0.2193 - 2.3801x - 0.3685x^2
m=3: F_3(x) = -0.5625 - 4.6217x + 0.0428x^2 + 0.4092x^3
```

Рисунок 29, результаты аппроксимации методом наименьших квадратов

В качестве дополнительной информации программа также выводит построенную систему уравнений. С ней можно ознакомиться на рисунке 30.

```
--- Polynomial of degree m = 1 ---
Normal LSM System (A|B):
10.0000*a0 + -3.3500*a1 = -2.3136
-3.3500*a0 + 33.8665*a1 = -69.0759

--- Polynomial of degree m = 2 ---
Normal LSM System (A|B):
10.0000*a0 + -3.3500*a1 + 33.8665*a2 = -2.3136
-3.3500*a0 + 33.8665*a1 + -33.2839*a2 = -69.0759
33.8665*a0 + -33.2839*a1 + 212.5685*a2 = 8.3153

--- Polynomial of degree m = 3 ---
Normal LSM System (A|B):
10.0000*a0 + -3.3500*a1 + 33.8665*a2 + -33.2839*a3 = -2.3136
-3.3500*a0 + 33.8665*a1 + -33.2839*a2 + 212.5685*a3 = -69.0759
33.8665*a0 + -33.2839*a1 + 212.5685*a2 + -331.2628*a3 = 8.3153
-33.2839*a0 + 212.5685*a1 + -331.2628*a2 + 1626.8909*a3 = -312.1413
```

Рисунок 30, построенная система уравнений

3.5 Вывод

Практическая работа по методам интерполяции и аппроксимации позволила сравнить их сильные стороны и ограничения на реальных данных. Реализация интерполяционных многочленов Лагранжа и Ньютона показала, что они дают идентичный полиномиальный результат для одного набора узлов. Многочлен

Лагранжа понятен в реализации, но неэффективен при многократных вычислениях, так как требует пересчета всей суммы для каждой новой точки. Многочлен Ньютона, основанный на разделенных разностях, оказался значительно удобнее в вычислительном плане. После однократного построения его коэффициентов добавление нового узла интерполяции не требует пересчета всей схемы с нуля, а значение в точке вычисляется по эффективной схеме Горнера.

Альтернативой выступили кубические сплайны. Их реализация оказалась более сложной, так как потребовала решения системы линейных уравнений для нахождения коэффициентов, но результат полностью оправдал усилия. Сплайновая интерполяция устранила проблему неустойчивых колебаний, обеспечив гладкое соединение узлов с непрерывностью не только самой функции, но и ее первой и второй производных. Это делает сплайны идеальным инструментом для построения плавных кривых по плотным данным, например, в компьютерной графике или при обработке результатов измерений.

В ситуациях, когда данные зашумлены или содержат погрешности, требование точного прохождения через все точки становится не физичным. Для таких случаев был применен метод наименьших квадратов, который строит не интерполирующую, а аппроксимирующую функцию. На практике было подтверждено, что этот метод эффективно сглаживает случайные ошибки, находя функцию, которая минимизирует суммарное квадратичное отклонение от данных. Особенно наглядно это проявилось при построении линейной и полиномиальной регрессии, где можно было управлять степенью аппроксимирующего полинома, находя баланс между точностью описания тренда и избеганием переобучения шумам.

Таким образом, полученный опыт четко очерчивает области применения каждого метода. Полиномы Лагранжа и Ньютона хороши для аналитической работы с малым числом точных узлов. Сплайны незаменимы для гладкой интерполяции плотных данных. Метод наименьших квадратов является основным инструментом для анализа и сглаживания экспериментальных данных, позволяя выявлять устойчивые закономерности.

4 Численное дифференцирование и интегрирование

4.1 Задание 13. Численное дифференцирование

Выполнить сравнение различных схем численного дифференцирования, вычислив значения первой и второй производной для заданной функции $y = f(x)$. В исследовании использовать 6 различных схем численного дифференцирования первого порядка и 4 схемы для второго порядка, построенных на двух, трех, четырех и пяти точечных шаблонах с разным порядком точности. Расчеты выполниться для шага h и для шага $h/2$. Графически сравнить результаты с аналитическими значениями.

Функция	Интервал и начальный шаг сетки
$y = \frac{\ln(5x^4 - 3x^2)}{2x^2 + \sin\frac{x}{4}} + \frac{3x^2 - \cos 3x}{\ln(x^3 + 3)}$	$[1{,}0;\ 4{,}0], \quad h = 0{,}15$

Численное дифференцирование представляет собой метод приближенного вычисления производных функции, когда аналитическое вычисление производных затруднительно или невозможно. Это важный инструмент численных методов, который применяется во многих областях науки и инженерии, таких как численное моделирование, обработка данных и решение дифференциальных уравнений. Вместо того чтобы вычислять производные в привычном аналитическом виде, численное дифференцирование позволяет получить приближенные значения производных с использованием конечных разностей.

Процесс численного дифференцирования заключается в замене производных на приближенные выражения, получаемые через разности значений функции в соседних точках. Эти выражения называются разностями. Разности могут быть первого порядка, второго порядка и даже высших порядков в зависимости от количества точек, используемых для аппроксимации. Различные схемы численного дифференцирования могут использовать разные подходы к вычислению этих разностей, что влияет на точность и эффективность метода.

Одной из самых простых схем для численного вычисления первой производной является схема двухточечной разности первого порядка. Эта схема использует разность значений функции в точках x и $x - h$, где h— шаг. Формула для этой схемы:

$$\frac{df}{dx}(x) \approx \frac{f(x) - f(x - h)}{h} \tag{4.1.1}$$

предполагает, что разность значений функции между точкой xи точкой $x - h$ делится на шаг h, что позволяет аппроксимировать первую производную. Эта схема обладает точностью первого порядка $O(h)$, и её ошибка будет пропорциональна величине шага h, как показано в функции two_point_1_error.

Для вычисления первой производной можно также использовать схему, известную как обратная разность. В отличие от прямой разности, эта схема вычисляет разницу между значениями функции в точке $x + h$и в точке x. Формула для этой схемы выглядит следующим образом:

$$\frac{df}{dx}(x) \approx \frac{f(x+h) - f(x)}{h} \tag{4.1.2}$$

Как и схема прямой разности, схема обратной разности имеет точность первого порядка $O(h)$, и ошибка будет зависеть от первой производной функции. В отличие от предыдущей схемы, она использует информацию только с правой стороны от точки x, что делает её менее точной, чем центральная разность.

Схема центральной разности для вычисления первой производной является более точной, так как она использует значения функции как с правой, так и с левой стороны точки x. Эта схема вычисляется по следующей формуле:

$$\frac{df}{dx}(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{4.1.3}$$

Центральная разность обладает точностью второго порядка $O(h^2)$, что означает, что ошибка схемы уменьшается быстрее по сравнению с прямой и обратной разностями. Ошибка вычисления первой производной с помощью этой схемы будет пропорциональна квадрату шага h^2, как показано в функции two_point_3_error.

Для повышения точности можно использовать более сложные схемы, такие как схема с четырьмя точками. Эта схема используется для вычисления производной, используя значения функции в точках x, $x + h$и $x + 2h$. Формула для этой схемы выглядит следующим образом:

$$\frac{df}{dx}(x) \approx \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h} \tag{4.1.4}$$

Схема с четырьмя точками также имеет точность второго порядка $O(h^2)$, но она более точна по сравнению с центральной разностью для вычисления первой производной, так как использует данные о значениях функции в нескольких точках. Ошибка этой схемы оценивается через третью производную функции, как показано в функции three_point_4_error.

Для ещё более высокой точности можно использовать схему с девятью точками, которая включает значения функции в пяти точках и записывается следующим образом:

$$\frac{df}{dx}(x) \approx \frac{-2f(x-h) - 3f(x) + 6f(x+h) - f(x+2h)}{6h} \quad (4.1.5)$$

Эта схема также имеет точность второго порядка $O(h^2)$, но из-за использования большего количества точек ошибка будет значительно меньше при малых значениях шага h, что делает её полезной для более точных вычислений.

Для достижения ещё большей точности можно использовать схему с девятнадцатью точками, которая обладает точностью четвёртого порядка $O(h^4)$. Формула для этой схемы выглядит так:

$$\frac{df}{dx}(x) \approx \frac{-25f(x) + 48f(x+h) - 36f(x+2h) + 16f(x+3h) - 3f(x+4h)}{12h} \quad (4.1.6)$$

Этот метод значительно снижает ошибку, особенно при малых шагах h, и её ошибка оценивается как $O(h^5)$, что делает её наиболее точной из рассмотренных.

Для вычисления второй производной функции также используются разностные схемы. Одной из таких схем является схема центральной разности второго порядка для второй производной. Формула для этой схемы выглядит следующим образом:

$$\frac{d^2f}{dx^2}(x) \approx \frac{f(x-2h) - 2f(x-h) + f(x)}{h^2} \quad (4.1.7)$$

Эта схема использует три точки и обладает точностью второго порядка O(h^2)). Для второй производной также можно использовать схему с точками \(x + h и $x - h$. Формула для этой схемы:

$$\frac{d^2f}{dx^2}(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \quad (4.1.8)$$

Она также имеет точность второго порядка $O(h^2)$, но её ошибка оценивается как $O(h^4)$, что позволяет повысить точность вычислений по сравнению с предыдущей схемой.

Для ещё более точных вычислений второй производной можно использовать схемы, которые включают четыре точки. Например, схема с четырьмя точками для второй производной выглядит следующим образом:

$$\frac{d^2f}{dx^2}(x) \approx \frac{2f(x) - 5f(x+h) + 4f(x+2h) - f(x+3h)}{h^2} \quad (4.1.9)$$

Эта схема также обладает точностью второго порядка $O(h^2)$, но её ошибка оценивается через четвёртую производную функции, что делает её более точной.

Для ещё более точных вычислений второй производной можно использовать схему с пятью точками, которая записана как:

$$\frac{d^2f}{dx^2}(x) \approx \frac{-f(x+2h) + 16f(x+h) - 30f(x) + 16f(x-h) - f(x-2h)}{12h^2} \quad (4.1.10)$$

Эта схема обладает точностью четвёртого порядка $O(h^4)$, что позволяет достичь более точных результатов при малых шагах h.

Точность схем численного дифференцирования определяется как порядок ошибки. Порядок ошибки для схемы численного дифференцирования первого порядка обычно равен $O(h)$, что означает, что ошибка уменьшается линейно с уменьшением шага h. Для схем второго порядка ошибка уменьшается как $O(h^2)$, а для схем более высокого порядка, таких как $O(h^4)$, ошибка уменьшается еще быстрее.

Таким образом, использование схем более высокого порядка, таких как схема на 4 или 5 точках, может значительно повысить точность вычислений, особенно при малых значениях шага h.

Для выполнения сравнения различных схем численного дифференцирования для первой и второй производных функции можно использовать несколько стандартных методов, таких как ошибка аппроксимации (разница между аналитическим значением производной и численно вычисленным значением) и графическое сравнение. Расчёты выполняются для различных шагов hи $h/2$, чтобы исследовать, как уменьшение шага влияет на точность результатов.

Ожидается, что при уменьшении шага ошибка будет снижаться для всех методов, однако, схемы с более высоким порядком точности должны показывать более быстрое уменьшение ошибки по сравнению с методами первого порядка.

Для первой производной можно сравнить такие схемы, как прямая разность, обратная разность и центральная разность, а для второй производной — центральную разность второго порядка, а также схемы, использующие большее количество точек, например, на основе 4-х или 5-ти точек.

Графически сравнив результаты с аналитическими значениями, можно убедиться в преимуществах использования схем с более высоким порядком

точности. Для этого можно построить графики, где на оси xбудет отклонение от аналитической функции, а на оси y— ошибка вычисления производной.

Численное дифференцирование является мощным инструментом для приближенного вычисления производных. Выбор схемы зависит от требований к точности и скорости вычислений. Для первой производной метод центральной разности второго порядка часто оказывается наиболее точным и стабильным. Для второй производной схемы на основе четырёх и пяти точек дают более высокую точность и позволяют минимизировать ошибку при вычислениях.

Функция $f(x)$представляет собой сложное выражение, включающее логарифмы, синусы и косинусы. Мы будем вычислять производные этой функции:

Листинг 4.1.1

```
def f(x):
    return (math.log(5 * x ** 4 - 3 * x ** 2) / (2 * x ** 2 + 
math.sin(x / 4))) + (
                (3 * x ** 2 - math.cos(3 * x)) / math.log(x ** 3 + 3))
```

Для вычисления первой и второй производных используется численное дифференцирование.

Для вычисления первой производной используются различные схемы, такие как схемы с двумя точками, тремя точками, четырьмя точками и пятью точками. Например, схема с двумя точками вычисляет первую производную по формуле:

Листинг 4.1.2

```
def two_point_1(x, h):
    if round(x - h, 4) < round(x_start, 4):
        return None
    return (f(x) - f(x - h)) / h
Ошибка для этой схемы оценивается как:
def two_point_1_error(h):
    return f"{h / 2} * f''"
```

Для вычисления второй производной также используются схемы с различным числом точек. Например, схема с тремя точками для второй производной выглядит следующим образом:

Листинг 4.1.3

```
def three_point_d2f_7(x, h):
    if round(x - 2 * h, 4) < round(x_start, 4):
        return None
    return (f(x - 2 * h) - 2 * f(x - h) + f(x)) / (h * h)
```

```
def three_point_d2f_7_error(h):
    return f"{h} * f³"
```

Для второй производной используются схемы с тремя, четырьмя и пятью точками. Например, схема с тремя точками для второй производной:

Листинг 4.1.4

```
def three_point_d2f_8(x, h):
    if round(x - h, 4) < round(x_start, 4) or round(x + h, 4) > round(x_end, 4):
        return None
    return (f(x + h) - 2 * f(x) + f(x - h)) / (h * h)

def three_point_d2f_8_error(h):
    return f"{h * h / 12} * f⁴"
```

Программа выполняет вычисления для заданных схем и выводит результаты в виде таблиц и графиков. Для каждой схемы производится вычисление производной, а также выводится ошибка.

Листинг 4.1.5

```
def print_values(derivative_schemes, top_header, x_values, order, h):
    names = [i[1] for i in derivative_schemes]
    col_width = 20
    header = "{:<10}".format("x")
    print(f"{top_header}. Шаг сетки h = {h}")

    for name in names:
        header += "{:<{width}}".format(name, width=col_width)
    print(header)

    num_values = {name: [] for name in names}
    for x in x_values:
        res_str = "{:<10.2f}".format(x)

        for func, name, error in derivative_schemes:
            try:
                res = derivative(func, x, h)
                res_str += "{:<{width}.8f}".format(res, width=col_width)
                num_values[name].append((x, res))
            except Exception:
                res_str += "{:<{width}}".format("-", width=col_width)
```

```
        print(res_str)
    print('\n')
    print("Погрешность:")
    res_str = "{:<{width}}".format("", width=col_width / 2)
    for __, _, error in derivative_schemes:
        res_str += "{:<{width}}".format(error(h), width=col_width)

    print(res_str + "\n")

    for name in names:
        x = [i[0] for i in num_values[name]]
        y = [i[1] for i in num_values[name]]
        plt.plot(x, y, label=name)

    if order == 1:
        df_values = [df(x) for x in x_values]
    elif order == 2:
        df_values = [d2f(x) for x in x_values]
    else:
        raise Exception("Неверный порядок")


    plt.plot(x_values, df_values, label=f"Analytical", marker='o')

    plt.title(f"{top_header}. Шаг сетки h = {h}")

    plt.xlabel("x")

    if order == 1:

        plt.ylabel("f'(x)")

    elif order == 2:

        plt.ylabel("f''(x)")

    plt.grid(True)

    plt.legend()

    plt.show()
```

Здесь для каждой схемы выполняются вычисления производных, затем результаты выводятся в виде таблиц. Также строятся графики, на которых можно увидеть, как значения численных производных сопоставляются с аналитическими значениями.

Ошибка каждой схемы вычисляется по аналитической формуле, которая зависит от шага сетки h и производной функции. Для этого в коде предусмотрены функции для оценки ошибок для каждой схемы. Например, для схемы с тремя точками для второй производной ошибка вычисляется как:

Листинг 4.1.6

```
def three_point_d2f_8_error(h):
    return f"{h * h / 12} * f⁴"
```

В основной функции программы задаются схемы для вычисления первой и второй производных, затем выполняются вычисления для шагов hи $h/2$, результаты выводятся в таблицах и на графиках. Также отображаются ошибки для каждой из схем.

Листинг 4.1.7

```
def main():
    h = 0.15

    num_points = int((x_end - x_start) / h) + 1
    x_values = np.linspace(x_start, x_end, num_points)

    derivative_schemes = [
        (two_point_1, "One-point var 1", two_point_1_error),
        (two_point_2, "One-point var 2", two_point_1_error),
        (two_point_3, "One-point var 3", two_point_3_error),
        (three_point_4, "Three-point var 4", three_point_4_error),
        (four_point_9, "Four-point var 9", four_point_9_error),
        (five_point_19, "Five-point var 19", five_point_19_error)
    ]

    print_values(derivative_schemes, "Первая производная", x_values,
1, h)
    print_values(derivative_schemes, "Первая производная", x_values,
1, h / 2.0)

    derivative_schemes2 = [
        (three_point_d2f_7, "Three-point var 7",
three_point_d2f_7_error),
        (three_point_d2f_8, "Three-point var 8",
three_point_d2f_8_error),
        (four_point_d2f_16, "Four-point var 16",
four_point_d2f_16_error),
```

```
            (five_point_d2f_23, "Five-point var 23",
five_point_d2f_23_error)
        ]
        print_values(derivative_schemes2, "Вторая производная", x_values,
2, h)
        print_values(derivative_schemes2, "Вторая производная", x_values,
2, h / 2.0)
```

На рисунке 31 приведены графики производных для различных схем для первой производной с шагом 0.15.

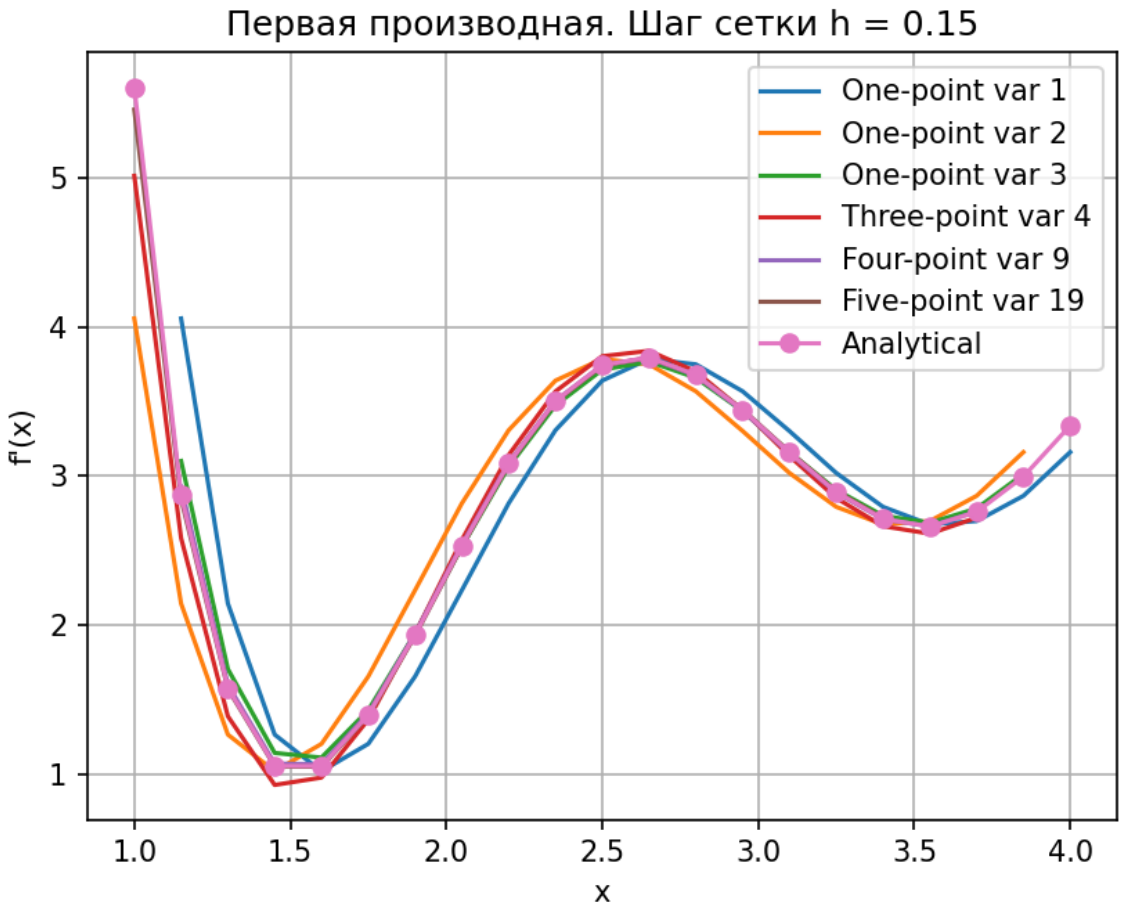

Рисунок 31, графики первых производных

На рисунке 32 приведены графики производных для различных схем для первой производной с шагом 0.075.

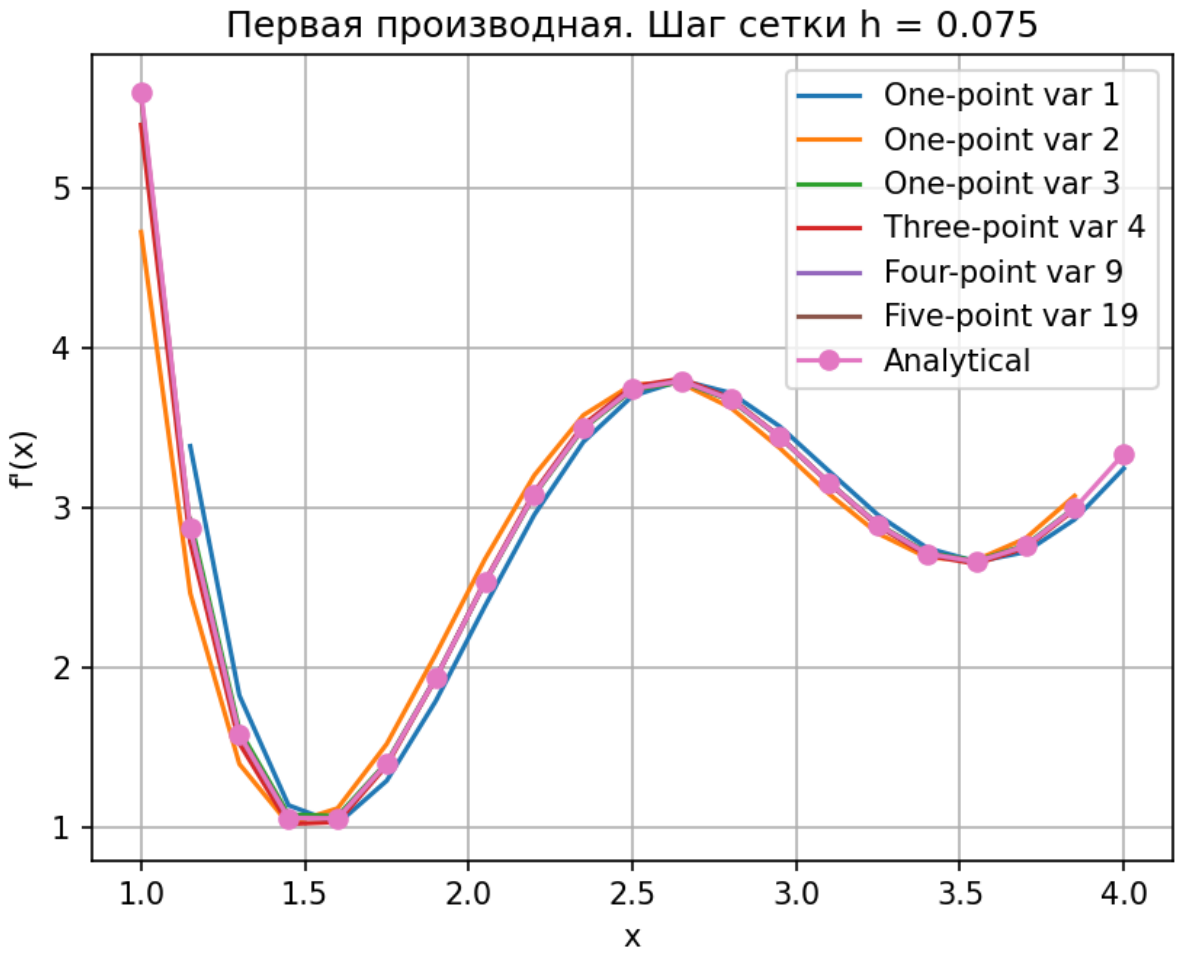

Рисунок 32, графики первых производных

На рисунке 33 приведены графики вторых производных для различных схем для первой производной с шагом 0.15.

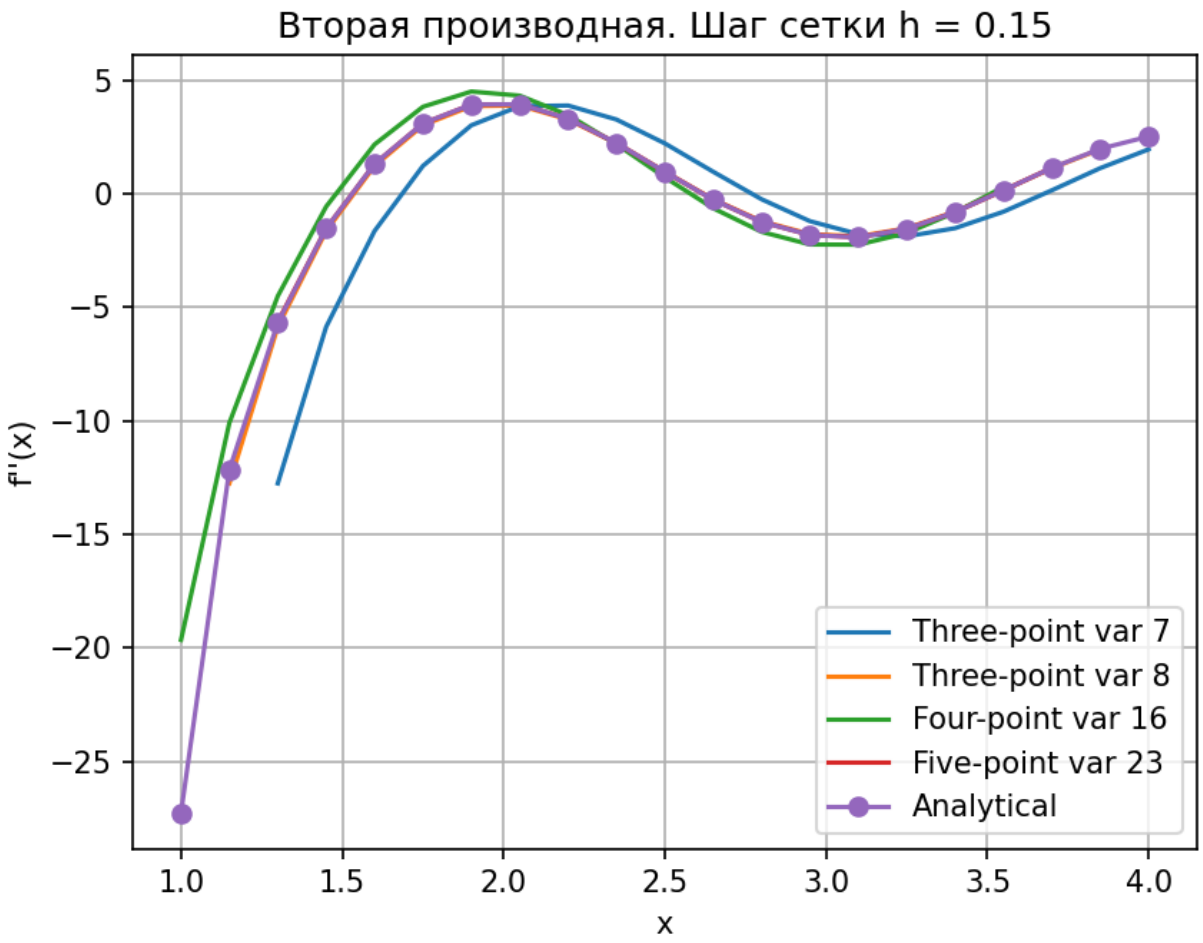

Рисунок 33, графики вторых производных

На рисунке 34 приведены графики вторых производных для различных схем для первой производной с шагом 0.075.

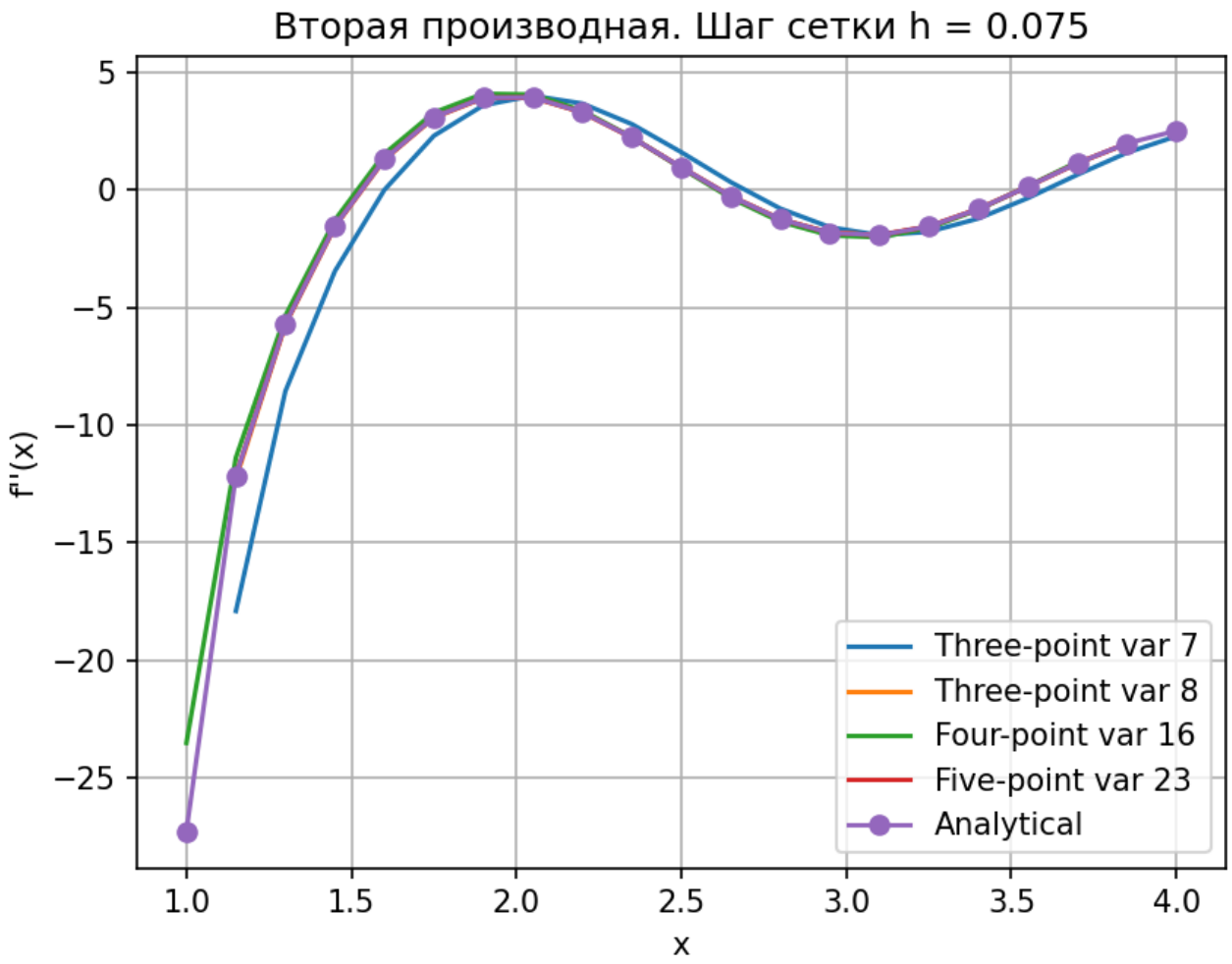

Рисунок 34, графики вторых производных

4.2 Задание 14. Численное интегрирование

Вычислить определенный интеграл по формулам средних прямоугольников, трапеций, Симпсона и Эйлера на равномерной сетке. Шаг интегрирования выбрать таким образом, чтобы на заданном отрезке можно было применить все перечисленные методы. Оценить погрешности методов интегрирования. Уменьшить шаг в два раза и повторить вычисление интегралов. Для каждого метода уточнить полученные значения, используя метод Рунге–Ромберга.

Искомый интеграл выглядит следующим образом:

$$F = \int_0^1 \frac{sh(\sin 3x)}{(x+1)^2}\,dx \qquad (4.2.1)$$

Численное интегрирование играет ключевую роль в математике и прикладных дисциплинах, когда аналитическое решение интеграла невозможно или слишком сложное. Основная цель численного интегрирования — найти приближенное значение определённого интеграла на основе значений функции в

дискретных точках на некотором интервале. Существует множество методов численного интегрирования, среди которых методы средних прямоугольников, трапеций, Симпсона и Эйлера считаются наиболее распространёнными. Рассмотрим их по порядку.

Метод средних прямоугольников основывается на аппроксимации графика функции с помощью прямоугольников, в которых высота каждого прямоугольника равна значению функции в середине интервала. Математически это можно выразить следующим образом. Пусть необходимо вычислить интеграл от функции $f(x)$на интервале $[a, b]$с шагом h, то есть интеграл имеет вид:

$$I = \int_a^b f(x)\, dx \tag{4.2.2}$$

В методе средних прямоугольников интервал $[a, b]$разбивается на nравных частей, и для каждого интервала $[x_i, x_{i+1}]$вычисляется площадь прямоугольника с высотой $f(\frac{x_i+x_{i+1}}{2})$. Формула для вычисления интеграла будет выглядеть так:

$$I \approx h \sum_{i=0}^{n-1} f(\frac{x_i + x_{i+1}}{2}) \tag{4.2.3}$$

где x_i— это узловые точки сетки, а h— шаг сетки, равный $\frac{b-a}{n}$.

Метод трапеций является одной из самых простых и широко используемых схем численного интегрирования. Он основан на аппроксимации функции линейными отрезками, соединяющими точки на графике функции. Для вычисления интеграла методом трапеций интервал $[a, b]$также разбивается на nравных частей, но вместо прямоугольников используются трапеции, площадь которых вычисляется по формуле:

$$I \approx \frac{h}{2}[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)] \tag{4.2.4}$$

где $x_0 = a$, $x_n = b$, а x_i— это промежуточные узловые точки. Как и в методе средних прямоугольников, h— это шаг сетки.

Метод Симпсона является более точным методом, который использует аппроксимацию функции параболами. В отличие от предыдущих методов, Симпсон использует не только значения функции в узловых точках, но и её значения в промежуточных точках, а также предполагает, что функция можно

аппроксимировать кусочными параболами. Для интервала $[a, b]$с шагом hформула для вычисления интеграла методом Симпсона будет следующей:

$$I \approx \frac{h}{3}[f(a) + 4 \sum_{\text{нечётные } i} f(x_i) + 2 \sum_{\text{чётные } i} f(x_i) + f(b)] \quad (4.2.5)$$

Здесь $x_0 = a$, $x_n = b$, а x_i— промежуточные узлы. В отличие от предыдущих методов, где для вычисления используется только два соседних значения функции, метод Симпсона использует тройки точек для аппроксимации функции.

Метод Эйлера — это один из наиболее простых методов численного интегрирования, основанный на идее аппроксимации функции с использованием её касательных. Суть метода Эйлера заключается в том, чтобы аппроксимировать функцию с помощью линейной функции, соединяющей соседние точки. Для вычисления интеграла методом Эйлера на интервале $[a, b]$с шагом hприменяется следующая формула:

$$I \approx h \sum_{i=0}^{n-1} f(x_i) \quad (4.2.6)$$

В отличие от метода средних прямоугольников, где высота прямоугольника равна значению функции в середине интервала, в методе Эйлера высота прямоугольника равна значению функции на левом конце интервала.

Метод Рунге-Ромберга используется для уточнения результатов численного интегрирования и позволяет значительно повысить точность решения, используя результаты, полученные для различных шагов сетки. Метод основан на том, что погрешность численного метода можно аппроксимировать некоторой степенной зависимостью от шага сетки h. Метод Рунге-Ромберга использует два вычисления интегралов для шагов hи $h/2$и их комбинацию для получения более точного результата.

Если I_h— это результат численного интегрирования для шага h, то более точная аппроксимация интеграла I_{exact} может быть вычислена по формуле Рунге-Ромберга:

$$I_{\text{exact}} \approx I_{h/2} + \frac{I_{h/2} - I_h}{2^p - 1} \quad (4.2.7)$$

где p— это порядок метода (например, для метода трапеций $p = 2$, для метода Симпсона $p = 4$, и так далее).

Метод Рунге-Ромберга использует идею о том, что при уменьшении шага сетки погрешность метода уменьшается в степени, связанной с порядком метода.

Применяя это правило, можно получить более точное значение интеграла, улучшив результаты для любого метода численного интегрирования.

Для исследования погрешностей методов интегрирования на заданном интервале $[a, b]$с выбором шага hи $h/2$, необходимо вычислить приближённые значения интегралов с помощью каждого из методов — средних прямоугольников, трапеций, Симпсона и Эйлера. Затем эти значения сравниваются с аналитическим решением, и на основе полученных данных оцениваются погрешности. После этого шаг уменьшен в два раза, и вычисления повторяются. Для уточнения результатов используется метод Рунге-Ромберга, который позволяет с большей точностью определить значение интеграла, корректируя погрешности каждого из методов численного интегрирования.

Таким образом, численное интегрирование — это мощный инструмент для нахождения приближённых значений определённых интегралов, а использование методов Рунге-Ромберга даёт возможность улучшить точность этих приближений, что особенно важно при решении сложных задач в численных методах.

Рассмотрим программную реализацию данной задачи. Функция $f(x)$, под интегралом, представлена как сложная комбинация гиперболического синуса и косинуса:

Листинг 4.2.1. Функция f

```
def f(x):
    return math.sinh(math.sin(3 * x)) / ((x + 1) ** 2)
```

Также определены её производные: первая, вторая и четвёртая, которые используются для вычисления погрешностей методов интегрирования. Например, для первой производной:

Листинг 4.2.2. Функция df

```
def df(x):
    return (3 * math.cosh(math.sin(3 * x)) * math.cos(3 * x)) / (x + 1) ** 2 - (2 * math.sinh(math.sin(3 * x))) / (x + 1) ** 3
```

Методы интегрирования, такие как метод средних прямоугольников, метод трапеций, метод Симпсона и метод Эйлера, реализованы в следующих функциях, представленных в листингах ниже.

Листинг 4.2.3. Метод прямоугольников

```
def midpoint_rule(a, b, h):
```

```
    n = int((b - a) / h)
    total = 0
    for i in range(n):
        x_mid = a + (i + 0.5) * h
        total += f(x_mid)
    return total * h
```

Листинг 4.2.4. Метод трапеций

```
def trapezoidal_rule(a, b, h):
    n = int((b - a) / h)
    total = (f(a) + f(b)) / 2
    for i in range(1, n):
        x_i = a + i * h
        total += f(x_i)
    return total * h
```

Листинг 4.2.5. Метод Симпсона

```
def simpson_rule(a, b, h):
    n = int((b - a) / h)
    if n % 2 != 0:
        raise Exception("Некорректный шаг")

    total = f(a) + f(b)
    for i in range(1, n):
        x_i = a + i * h
        if i % 2 == 0:
            total += 2 * f(x_i)
        else:
            total += 4 * f(x_i)
    return total * h / 3
```

Листинг 4.2.6. Метод Эйлера

```
def euler_rule(a, b, h):
    n = int((b - a) / h)
    total = (f(a) + f(b)) / 2.0
    for i in range(1, n):
        x_i = a + i * h
        total += f(x_i)
    total = total * h
    return total + 1.0 / 12.0 * h * h * (df(a) - df(b))
```

Для каждого метода численного интегрирования оценивается погрешность с использованием максимальных значений производных функции. Например, для метода средних прямоугольников погрешность вычисляется через вторую производную:

Листинг 4.2.7. Функция подсчета ошибки

```
def calculate_mistake(method, a, b, h):
    if method == midpoint_rule:
        mx = find_max(lambda x: abs(d2f(x)), a, b, h)
        return mx * (b - a) * h * h / 24
    elif method == trapezoidal_rule:
        mx = find_max(lambda x: abs(d2f(x)), a, b, h)
        return mx * (b - a) * h * h / 12
    elif method == simpson_rule:
        mx = find_max(lambda x: abs(d4f(x)), a, b, h)
        return mx * (b - a) * h ** 4 / 180
    elif method == euler_rule:
        mx = find_max(lambda x: abs(d4f(x)), a, b, h)
        return mx * h ** 4 / 720
    else:
        raise Exception("Неизвестный метод")
```

Метод Рунге-Ромберга используется для уточнения значения интеграла, полученного с шагом h, на основе вычислений для шага $h/2$. Метод вычисляется по следующей формуле:

$$I_{\text{refined}} = I_{h/2} + \frac{I_{h/2} - I_h}{2^p - 1} \tag{4.2.8}$$

где p— это порядок метода численного интегрирования. Для методов Симпсона и Эйлера $p = 4$, а для методов трапеций и средних прямоугольников $p = 2$.

Листинг 4.2.8. Функция, реализующая метод Рунге-Ромберга

```
def runge_romberg(I_h, I_h2, p):
    return I_h2 + (I_h2 - I_h) / (2 ** p - 1)
```

В основной программе выполняются вычисления для всех четырёх методов численного интегрирования на заданном интервале $[a, b]$с шагом $h = 0.1$, а затем с шагом $h/2$. Результаты уточняются с помощью метода Рунге-Ромберга.

Листинг 4.2.9. Основная программа

```
a, b = 0, 1
h = 0.1
methods = [
    ("Средние прямоугольники", midpoint_rule, 2),
    ("Трапеции", trapezoidal_rule, 2),
    ("Симпсона", simpson_rule, 4),
    ("Эйлера", euler_rule, 4)
]

print(f"{'Метод':<25} {'I(h)':>14} {'I(h/2)':>14} {'Уточнённое':>16} {'Погр. при h':>14} {'Погр. при h/2':>14}")
print("=" * 105)

for name, method, p in methods:
    I_h = method(a, b, h / 2)
    I_h2 = method(a, b, h)
    I_refined = runge_romberg(I_h2, I_h, p)
    m1 = calculate_mistake(method, a, b, h)
    m2 = calculate_mistake(method, a, b, h / 2)

    print(f"{name:<25}"
          f"{I_h:14.8f} "
          f"{I_h2:14.8f} "
          f"{I_refined:16.8f} "
          f"{m1:16.8e} "
          f"{m2:16.8e}")
```

В результате работы программы выводятся таблицы, в которых для каждого метода численного интегрирования приводятся значения интегралов, уточнённые результаты и погрешности для шагов hи $h/2$. Эти таблицы приведены на рисунке 35.

Метод	I(h)	I(h/2)	Уточнённое	Погр. при h	Погр. при h/2
Средние прямоугольники	0.34219613	0.34337253	0.34180400	4.72403983e-04	1.18100996e-04
Трапеции	0.34101415	0.33865578	0.34180028	9.44807966e-04	2.36201991e-04
Симпсона	0.34180028	0.34177019	0.34180229	4.57921236e-06	2.86200772e-07
Эйлера	0.34180276	0.34181019	0.34180226	1.14480309e-06	7.15501931e-08

Рисунок 35, графики вторых производных

4.3 Вывод

Практическая работа по численному дифференцированию позволила детально исследовать влияние выбора разностной схемы на точность вычисления производных. Были реализованы и сравнены шесть схем для первой производной и четыре схемы для второй производной, построенные на шаблонах разной ширины (от двух до пяти точек) и имеющие разный порядок точности. На графиках было наглядно видно, как увеличение количества точек в шаблоне и использование схем с более высоким порядком точности (например, центральные разности второго порядка) позволяют существенно снизить погрешность по сравнению с простейшими односторонними разностями первого порядка. Особенно показательным было сравнение результатов для шага h и шага h/2. Уменьшение шага закономерно повышало точность всех схем, однако для схем низкого порядка этот выигрыш был менее значительным, а погрешность могла оставаться заметной даже при малых шагах из-за влияния погрешности округления. Работа наглядно продемонстрировала фундаментальный компромисс численного дифференцирования: использование более точных многоточечных схем требует более гладкой функции и сталкивается с трудностями на краях интервала, где не хватает точек для построения симметричного шаблона. Таким образом, выбор оптимальной схемы всегда зависит от конкретных условий — доступности данных, требуемой точности и поведения самой функции.

В разделе, посвященном численному интегрированию, были применены и проанализированы четыре классические квадратурные формулы на равномерной сетке: средних прямоугольников, трапеций, Симпсона и Эйлера. Расчеты, выполненные с двумя различными шагами интегрирования, подтвердили теоретические оценки их порядка точности. Метод средних прямоугольников и метод трапеций показали сходимость второго порядка относительно шага, в то время как более сложная формула Симпсона, использующая квадратичную интерполяцию, продемонстрировала сходимость четвертого порядка, что сделало ее наиболее точной на сравнительно крупной сетке. Реализация уточнения по методу Рунге–Ромберга стала ключевым этапом, позволившим существенно повысить точность результатов без дополнительных вычислительных затрат на

еще более мелкое разбиение. Этот метод эффективно оценил ведущую погрешность и построил уточненное значение, которое для схем второго порядка (прямоугольники, трапеции) дало результат с точностью, сопоставимой с результатом формулы Симпсона на исходной сетке. Практика подтвердила, что формула Симпсона является оптимальным выбором для интегрирования гладких функций, сочетая высокую точность с умеренной вычислительной сложностью. В то же время, для функций с разрывами или быстрыми осцилляциями более надежными могут оказаться простые методы, такие как трапеций, либо адаптивные алгоритмы. Общий вывод заключается в том, что численное интегрирование, в отличие от дифференцирования, является устойчивой и хорошо управляемой процедурой, где точность может планомерно повышаться как за счет уменьшения шага, так и за счет применения более точных формул и методов уточнения результатов

5 Решение дифференциальных уравнений

5.1 Задание 15. Явные и неявные схемы

5.1.1 Теоретическая часть

Решить задачу Коши для обыкновенного дифференциального уравнения первого порядка. Полученное численное решение сравнить с аналитическим решением. Определить погрешность решения. Написать программу для реализации явного и неявного методов решения ОДУ по общим таблицам Бутчера. В качестве тестовых примеров использовать предложенные схемы разного порядка

Задача Коши	Аналитическое решение
$xy' + \sqrt{y^2 + x^2} - y = 0$ $y(0{,}4) = 0{,}42$ $x \in [0{,}4;\ 3], \quad h = 0{,}2$	$y = x \cdot sh\left(ln\frac{1}{x}\right)$

Решение задачи Коши для обыкновенного дифференциального уравнения первого порядка — классическая задача численной математики: задано начальное значение $y(t_0) = y_0$и уравнение:

$$y' = f(t, y), t \geq t_0, \tag{5.1.1.1}$$

требуется построить приближённую функцию $y_n \approx y(t_n)$на сетке $t_n = t_0 + nh$. Подходы к решению делятся на явные и неявные схемы; ключевые вопросы — точность (порядок), устойчивость (включая поведение для жёстких задач), экономичность вычислений (число вызовов правой части и, для неявных схем, число итераций для решения алгебраических систем), и контроль погрешности.

Любая схема Рунге–Кутты задаётся таблицей Бутчера. Для s стадий она задаётся матрицей $A = (a_{ij})$, вектором сдвигов $c = (c_i)$и вектором весов $b = (b_i)$. Один шаг длины hот (t_n, y_n) даёт стадии:

$$k_i = f\left(t_n + c_i h,\ y_n + h\sum_{j=1}^{s} a_{ij}k_j\right), i = 1 \dots s, \tag{5.1.1.2}$$

Обновление имеет вид:

$$y_{n+1} = y_n + h\sum_{i=1}^{s} b_i k_i \tag{5.1.1.3}$$

Если A строго верхнетреугольная (все $a_{ij} = 0$при $j \geq i$), то схема явная: вычисление k_i не требует решения уравнений и делается последовательно. Типичный пример явной схемы — классический RK4 (четырёхстадийный,

порядок 4). Если же в A есть ненулевые элементы на диагонали или ниже, стадии k_iопределяются неявно и требуют решения системы нелинейных уравнений на каждой стадии. Неявные схемы требуют итерационных решателей (например, Ньютон или фиксированная точка) для нахождения векторов k.

Неявные схемы повышают устойчивость. Неявные схемы, такие как обратный Эйлер и трапеций, часто являются A-устойчивыми (вся левополуплоскость входит в область устойчивости); некоторые неявные схемы ещё и L-устойчивы (гашение жестких составляющих). Практический вывод: для нестрогих задач удобны явные RK высоких порядков (например, RK4 или встроенные адаптивные схемы с оценкой погрешности), для жёстких задач выбирают неявные.

Реализация «общих таблиц Бутчера». Написание общей функции шага RK, принимающей таблицу A, c, b, позволяет единожды реализовать и явные, и неявные схемы. Для явных схем алгоритм сводится к простому циклу: для i вычислить (15.2). Для неявных схем необходимо решать систему векторов $K = (k_1, \dots, k_s)$из нелинейных уравнений:

$$G_i(K) := k_i - f\left(t_n + c_i h,\ y_n + h\sum_j a_{ij} k_j\right) = 0, i = 1 \dots s, \qquad (5.1.1.4)$$

или компактно $G(K) = 0$. Эту систему обычно решают методом Ньютона.

Когда аналитическое решение $y_{\text{exact}}(t)$известно, сравнение тривиально: вычисляется норма ошибки в узлах сетки $e_n = y_n - y_{\text{exact}}(t_n)$ и оценивается максимум, среднее из этих ошибок.

5.1.2 Практическая часть

Задача Коши представляет собой дифференциальное уравнение первого порядка:

$$\mathrm{y}' = \frac{\sqrt{\mathrm{y}^2 + \mathrm{x}^2} - \mathrm{y}}{-\mathrm{x}}, \qquad (5.1.2.1)$$

с начальными условиями $y(x_0) = y_0$, где $x_0 = 0.4$— начальная точка на оси x, $x_{\text{end}} = 3.0$— конечная точка, $y_0 = 0.42$— значение функции yв точке x_0.

В коде определена функция ode_rhs, которая представляет правую часть дифференциального уравнения:

Листинг 5.1.2.1. Правая часть

```
def ode_rhs(x: float, y: float) -> float:
```

```
    return (math.sqrt(y ** 2 + x ** 2) - y) / -x
```

Также имеется функция analytic_solution, которая задает аналитическое решение задачи (если оно известно):

Листинг 5.1.2.2. Аналитическое решение

```
def analytic_solution(x: float) -> float:
    return x * math.sinh(math.log(1 / x))
```

В коде определены две группы методов Рунге-Кутты — явные и неявные схемы. Каждая схема описана с помощью таблицы Бутчера, которая включает в себя вектор сдвигов (c), матрица коэффициентов (a), вектор весов (b).

Для явных методов Рунге-Кутты используется несколько схем различного порядка. Пример:

Листинг 5.1.2.3. Пример таблицы Бутчера для явного метода

```
{
    "name": "Метод Хойна (2 порядок)",
    "order": 2,
    "c": (0.0, 0.5),
    "a": ((0.0, 0.0), (0.5, 0.0)),
    "b": (0.0, 1.0),
    "implicit": False,
}
```

Этот метод имеет два этапа (стадии), и его схема решает систему с использованием значений y_n на предыдущих шагах.

Неявные методы, такие как неявный Эйлер, задаются аналогично, но для них требуется решение системы нелинейных уравнений на каждом шаге. Пример неявного метода:

Листинг 5.1.2.4. Пример таблицы Бутчера для неявного метода

```
{
    "name": "Неявный Эйлер",
    "order": 1,
    "c": (1.0,),
    "a": ((1.0,),),
    "b": (1.0,),
    "implicit": True,
}
```

Функция build_grid строит сетку значений x от x_0 до x_{end} с шагом h:

Листинг 5.1.2.5. Пример таблицы Бутчера для неявного метода

```
def build_grid(a: float, b: float, h: float) -> List[float]:
    xs: List[float] = []
    n = 0
    while True:
        x = a + n * h
        if x > b + 1e-12:
            break
        xs.append(x)
        n += 1
    if abs(xs[-1] - b) > 1e-12:
        xs.append(b)
    return xs
```

Для неявных методов необходимо решать систему линейных уравнений на каждом шаге. Это выполняется функцией solve_linear_system, которая использует метод Ньютона.

Листинг 5.1.2.6. Функция решения системы уравнений

```
    def solve_linear_system(matrix: Sequence[Sequence[float]], rhs:
Sequence[float]) -> List[float]:
    n = len(rhs)
    a = [list(row) for row in matrix]
    b = list(rhs)
    for col in range(n):
        pivot_row = max(range(col, n), key=lambda i: abs(a[i][col]))
        if abs(a[pivot_row][col]) < 1e-15:
            raise RuntimeError("Матрица плохо обусловлена")
        if pivot_row != col:
            a[col], a[pivot_row] = a[pivot_row], a[col]
            b[col], b[pivot_row] = b[pivot_row], b[col]
        pivot = a[col][col]
        inv_pivot = 1.0 / pivot
        for j in range(col, n):
            a[col][j] *= inv_pivot
        b[col] *= inv_pivot
        for row in range(col + 1, n):
            factor = a[row][col]
            if factor == 0.0:
                continue
            for j in range(col, n):
                a[row][j] -= factor * a[col][j]
            b[row] -= factor * b[col]
```

```
    x = [0.0] * n
    for i in reversed(range(n)):
        x[i] = b[i] - sum(a[i][j] * x[j] for j in range(i + 1, n))
    return x
```

Основной функционал, который выполняет один шаг численного метода Рунге-Кутты, — это функция rk_step. Она использует таблицу Бутчера, чтобы вычислить новое значение yна основе текущего состояния xи y.

Для явных методов вычисление стадий происходит последовательно, а для неявных — через решение системы нелинейных уравнений:

Листинг 5.1.2.7. Шаг метода Рунге-Кутты

```
def rk_step(
        tableau: Tableau,
        h: float,
        x_cur: float,
        y_cur: float,
        rhs: Callable[[float, float], float],
) -> float:
    explicit = not tableau["implicit"]
    stages = solve_stages(tableau, h, x_cur, y_cur, explicit, rhs)
    increment = sum(b_i * k_i for b_i, k_i in zip(tableau["b"], stages))
    return y_cur + h * increment
```

Для решения задачи Коши на отрезке от x_0до x_{end}с шагом hиспользуется функция solve_cauchy. Она поочередно применяет шаг метода Рунге-Кутты для каждого интервала на сетке:

Листинг 5.1.2.8. Функция solve_cauchy

```
def solve_cauchy(
        tableau: Tableau,
        x0: float,
        x_end: float,
        y0: float,
        h: float,
        rhs: Callable[[float, float], float],
) -> Tuple[List[float], List[float]]:
```

```
    xs = build_grid(x0, x_end, h)
    ys = [y0]
    for idx in range(len(xs) - 1):
        x_cur = xs[idx]
        y_cur = ys[-1]
        y_next = rk_step(tableau, h, x_cur, y_cur, rhs)
        ys.append(y_next)
    return xs, ys
```

Для оценки ошибки сравнивается численное решение с аналитическим. Функция collect_results собирает результаты для каждого метода, вычисляет ошибку и возвращает информацию.

Листинг 5.1.2.9. Функция collect_results

```
def collect_results(
        tableaus: Sequence[Tableau],
        h: float,
        x0: float,
        x_end: float,
        y0: float,
        rhs: Callable[[float, float], float],
) -> List[dict]:
    results = []
    for tableau in tableaus:
        xs, ys = solve_cauchy(tableau, x0, x_end, y0, h, rhs)
        exact = [analytic_solution(x) for x in xs]
        errors = [abs(y - y_exact) for y, y_exact in zip(ys, exact)]
        results.append(
            {
                "tableau": tableau,
                "xs": xs,
                "ys": ys,
                "exact": exact,
                "errors": errors,
                "max_error": max(errors),
                "mean_error": sum(errors) / len(errors),
                "final_error": errors[-1],
            }
        )
    return results
```

Функция print_summary выводит таблицу с результатами для разных методов, а plot_results строит графики для явных и неявных схем.

Листинг 5.1.2.10. Функции вывода

```
    def print_summary(title: str, results: Sequence[dict]) -> None:
    print(f"\n{title}")
    header = f"{'Метод':35s} {'p':>3s} {'max|ош|':>12s} {'ср.ош':>12s}
{'|ош| в x=5':>12s}"
    print(header)
    print("*" * len(header))
    for res in results:
        t = res["tableau"]
        print(
            f"{t['name']:35s} {t['order']:3d} "
            f"{res['max_error']:12.5e} "
            f"{res['mean_error']:12.5e} "
            f"{res['final_error']:12.5e}"
        )

def plot_results(explicit_results: Sequence[dict], implicit_results:
Sequence[dict]) -> None:
    smooth_x = np.arange(0.4, 3.0 + 1e-9, 0.01)
    smooth_y = [analytic_solution(val) for val in smooth_x]

    fig, axes = plt.subplots(1, 2, figsize=(14, 6), sharey=True)
    for ax, res_set, title in zip(
            axes,
            (explicit_results, implicit_results),
            ("Явные схемы", "Неявные схемы"),
    ):
        ax.plot(smooth_x, smooth_y, color="black", label="Аналитика",
linestyle="--")
        for res in res_set:
            ax.plot(
                res["xs"],
                res["ys"],
                marker="o",
                linestyle=":",
                label=res["tableau"]["name"],
            )
        ax.set_title(title)
        ax.set_xlabel("x")
        ax.set_ylabel("y(x)")
        ax.grid(True, linestyle=":", alpha=0.6)
        ax.legend(fontsize=8)
```

```
    fig.suptitle("Сравнение результатов решения задачи Коши")
    plt.tight_layout()
    plt.show()
```

Основная функция main запускает все процессы: собирает результаты для явных и неявных методов, выводит статистику и строит графики.

Листинг 5.1.2.11. Функция main

```
def main() -> None:
    x0 = 0.4
    x_end = 3.0
    y0 = 0.42
    h = 0.2

    explicit_results = collect_results(EXPLICIT_TABLEAUS, h, x0, x_end, y0, ode_rhs)
    implicit_results = collect_results(IMPLICIT_TABLEAUS, h, x0, x_end, y0, ode_rhs)

    print_summary("Явные методы", explicit_results)
    print_summary("Неявные методы", implicit_results)
    plot_results(explicit_results, implicit_results)
```

5.1.3 Результаты работы программы

На рисунках 36 и 37 представлены графики для явных и неявных методов различного порядка.

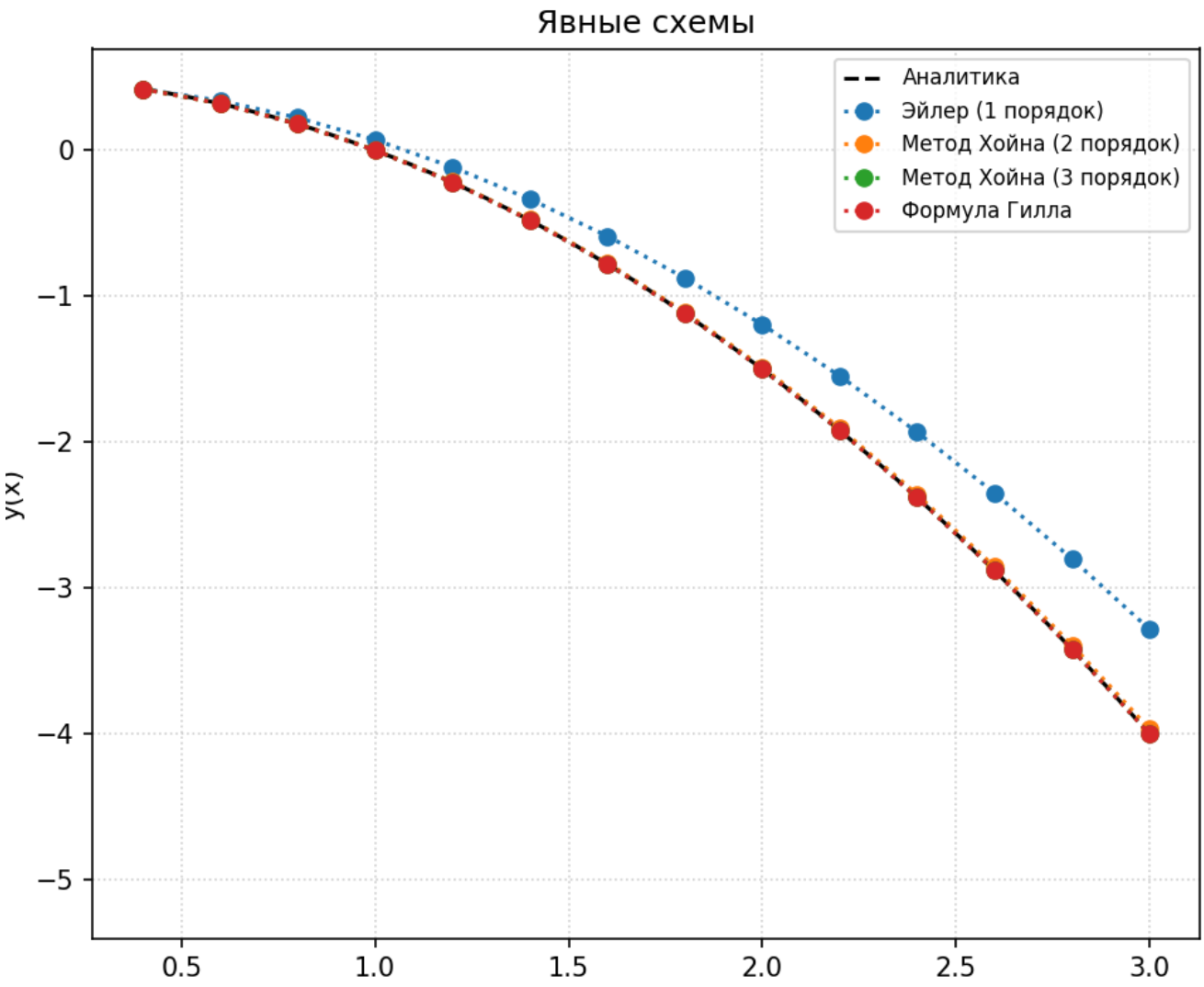

Рисунок 36, графическое сравнение явных методов с аналитическим решением

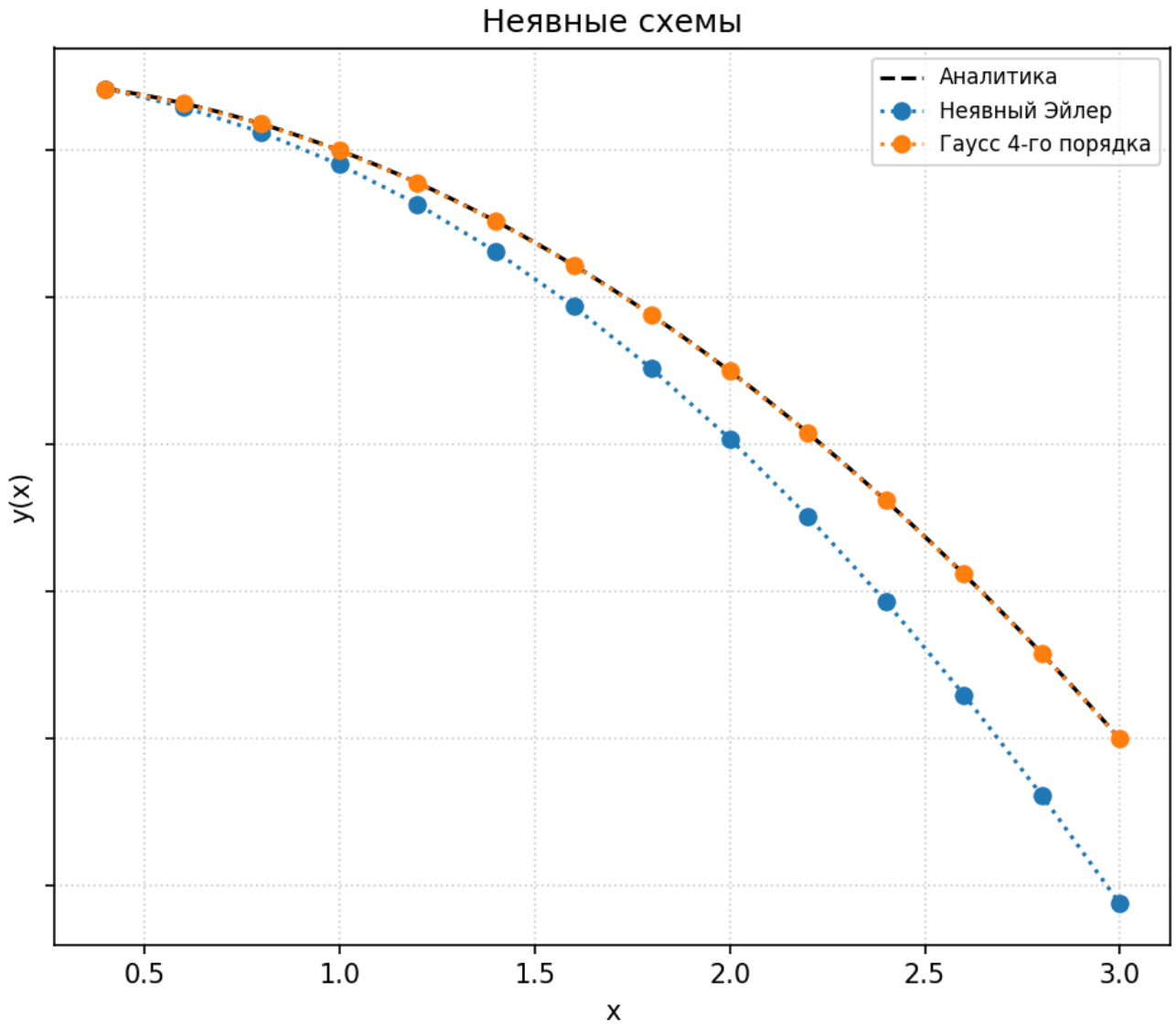

Рисунок 37, графическое сравнение неявных методов с аналитическим решением

На рисунке 38 приведен вывод программы.

```
Численное решение задачи Коши y' = x(x+2)y^3 + (x+3)y^2, y(1) = -2/3
Шаг сетки h = 0.20, отрезок [0.4; 3.0]

Явные методы
Метод                                p      max|ош|        ср.ош    |ош| в x=5
********************************************************************************
Эйлер (1 порядок)                    1  7.17950e-01  2.72964e-01  7.17950e-01
Метод Хойна (2 порядок)              2  3.45586e-02  1.32036e-02  3.45586e-02
Метод Хойна (3 порядок)              3  1.37741e-03  5.34548e-04  1.37741e-03
Формула Гилла                        4  7.10644e-05  2.58856e-05  7.10644e-05

Неявные методы
Метод                                p      max|ош|        ср.ош    |ош| в x=5
********************************************************************************
Неявный Эйлер                        1  1.12081e+00  4.13331e-01  1.12081e+00
Гаусс 4-го порядка                   4  1.02496e-12  4.28964e-13  1.02496e-12
```

Рисунок 38, вывод программы

5.2 Задание 16 (17). Краевая задача. Конечно-разностный метод

5.2.1 Теоретическая часть

Решить краевую задачу для линейного дифференциального уравнения второго порядка. Использовать конечно-разностный метод. Граничные условия аппроксимировать с первым и со вторым порядком точности. Полученные численные решения сравнить с аналитическим решением. Определить погрешность решения.

Краевая задача	Аналитическое решение
$x^2(3x+2)y'' - 2x(3x+4)y' + 6(x+2)y = 0$ $y(0{,}5) + y'(0{,}5) = 0{,}515$ $y(6{,}5) = 14{,}738 \qquad h = 0{,}5$	$y(x) = \frac{x^2(1+x)}{3x+2}$

Для решения краевой задачи для линейного дифференциального уравнения второго порядка с использованием конечно-разностного метода, важно правильно понимать, как дифференциальное уравнение и граничные условия аппроксимируются с помощью численных методов. Обозначим первое граничное условие как LEFT_BC, а второе граничное условие как RIGHT_BC.

Рассмотрим краевую задачу для линейного дифференциального уравнения второго порядка. Общий вид уравнения выглядит следующим образом:

$$\frac{d^2u}{dx^2} + p(x)\frac{du}{dx} + q(x)u(x) = f(x) \qquad (5.2.1.1)$$

Для численного решения задачи с использованием метода конечных разностей сначала необходимо аппроксимировать производные и саму задачу с помощью разностных схем. Мы делим интервал $[a, b]$на равномерные части, создавая сетку, на которой будем вычислять значения функции. Эта сетка состоит из узлов $x_0, x_1, x_2, \dots, x_n$, где шаг между соседними узлами h вычисляется как $h = \frac{b-a}{n}$, и количество шагов n зависит от требуемой точности.

Далее мы аппроксимируем производную во внутренних точках интервала и на концах интервала. Во внутренних точках аппроксимируем формулами второго порядка:

$$y'_k = \frac{y_{k+1} - y_{k-1}}{2h} + O(h^2) \quad (5.2.1.2)$$

$$y''_k = \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + O(h^2) \quad (5.2.1.3)$$

В граничных точках, исходя из условия задачи, мы попробуем два варианта аппроксимации: формулами первого и второго порядков. Они имеют следующий вид.

Первый порядок:

$$y'_0 = \frac{y_1 - y_0}{h} + O(h) \quad (5.2.1.4)$$

$$y'_n = \frac{y_n - y_{n-1}}{h} + O(h) \quad (5.2.1.5)$$

Второй порядок:

$$y'_0 = \frac{-3y_0 + 4y_1 - y_2}{2h} + O(h^2) \quad (5.2.1.6)$$

$$y'_n = \frac{y_{n-2} - 4y_{n-1} + 3y_n}{2h} + O(h^2) \quad (5.2.1.7)$$

После этого эти аппроксимации подставляются в исходное уравнение. В результате мы получаем систему вида Ay = b. Для нашей задачи эта матрица выглядит следующим образом.

Для первого порядка аппроксимации:

$$\begin{pmatrix} 1-\frac{1}{H} & \frac{1}{H} & 0 & 0 & \dots & 0 \\ \frac{1}{H^2}-\frac{p(x_1)}{2H} & -\frac{2}{H^2}+q(x_1) & \frac{1}{H^2}+\frac{p(x_1)}{2H} & 0 & \dots & 0 \\ 0 & \frac{1}{H^2}-\frac{p(x_2)}{2H} & -\frac{2}{H^2}+q(x_2) & \frac{1}{H^2}+\frac{p(x_2)}{2H} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \frac{1}{H^2}-\frac{p(x_{n-2})}{2H} & -\frac{2}{H^2}+q(x_{n-2}) & \frac{1}{H^2}+\frac{p(x_{n-2})}{2H} \\ 0 & 0 & 0 & \dots & 0 & 1 \end{pmatrix} \quad (5.2.1.8)$$

Для второго порядка аппроксимации:

$$\begin{pmatrix} 1-\frac{3}{2H} & \frac{4}{2H} & -\frac{1}{2H} & 0 & \dots & 0 \\ \frac{1}{H^2}-\frac{p(x_1)}{2H} & -\frac{2}{H^2}+q(x_1) & \frac{1}{H^2}+\frac{p(x_1)}{2H} & 0 & \dots & 0 \\ 0 & \frac{1}{H^2}-\frac{p(x_2)}{2H} & -\frac{2}{H^2}+q(x_2) & \frac{1}{H^2}+\frac{p(x_2)}{2H} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \frac{1}{H^2}-\frac{p(x_{n-2})}{2H} & -\frac{2}{H^2}+q(x_{n-2}) & \frac{1}{H^2}+\frac{p(x_{n-2})}{2H} \\ 0 & 0 & 0 & \dots & 0 & 1 \end{pmatrix} \quad (5.2.1.9)$$

Правая часть в обоих случаях одинакова:

$$\mathrm{b} = \begin{pmatrix} \text{LEFT\_BC} \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{n-2}) \\ \text{RIGHT\_BC} \end{pmatrix} \quad (5.2.1.10)$$

Для решения этой системы используется метод Гаусса или другие численные методы решения систем линейных уравнений. По полученным значениям $y_0, y_1, \dots, y_n$ и соответствующим им узлам сетки $x_0, x_1, \dots, x_n$ можно построить функцию, которая является численным решением краевой задачи.

5.2.2 Практическая часть

В начале программы импортируются необходимые библиотеки и выполняются базовые настройки. Подключаются модули для математических операций, работы с системой ввода-вывода и для рисования графиков.

Далее задаются константы, которые определяют параметры задачи. Переменные A и B — это левая и равая границы интервала, H — шаг сетки, а LEFT_BC и RIGHT_BC — это граничные условия на левой и правой границе задачи.

Листинг 5.2.2.1. Входные данные

```
A = 0.5  # левая граница
```

```
B = 6.5  # правая граница
H = 0.5  # шаг сети
LEFT_BC = 0.515
RIGHT_BC = 14.738
```

Далее определяются функции для вычисления коэффициентов дифференциального уравнения, а также аналитическое решение. Функции a(x), p(x), q(x) возвращают коэффициенты уравнения второго порядка, а функция f(x) — правую часть уравнения, которая в данном случае равна нулю.

Листинг 5.2.2.2. Входные данные

```
def a(x: float):
    return x * x * (3 * x + 2)

def p(x: float) -> float:
    return (-2.0 * x * (3 * x + 4)) / a(x)

def q(x: float) -> float:
    return 6 * (x + 2) / a(x)

def f(x: float) -> float:
    return 0.0
```

Функция analytic_solution(x) задает аналитическое решение задачи, которое будет использоваться для сравнения с численными результатами.

Листинг 5.2.2.3. Аналитическое решение

```
def analytic_solution(x: float) -> float:
    return (x * x * (1 + x)) / (3 * x + 2)
```

Затем идет функция build_system, которая создает систему линейных уравнений для решения задачи методом конечных разностей. В зависимости от переданного порядка аппроксимации для производной на левой границе, строится соответствующая система.

Листинг 5.2.2.4. Функция построения системы

```
def build_system(order: int) -> Tuple[List[List[float]], List[float]]:
    n_intervals = round((B - A) / H)  # количество интервалов
    size = n_intervals + 1  # количество узлов
    matrix = [[0.0 for _ in range(size)] for _ in range(size)]
    rhs = [0.0 for _ in range(size)]

    # Левая граница: y(a) + y'(a) = LEFT_BC
    if order == 1:
```

```
        # Односторонняя разность для y'(a)
        matrix[0][0] = 1.0 - 1.0 / H  # для y(a)
        matrix[0][1] = 1.0 / H  # для y_1 (разность для y')
        rhs[0] = LEFT_BC
    elif order == 2:
        # Второй порядок аппроксимации для y'(a)
        matrix[0][0] = 1.0 - 3.0 / (2 * H)
        matrix[0][1] = 4.0 / (2 * H)
        matrix[0][2] = -1.0 / (2 * H)
        rhs[0] = LEFT_BC
    else:
        raise ValueError("Допустимые порядки аппроксимации: 1 или 2.")

    # Основная часть для внутренних точек
    for k in range(1, size - 1):
        xk = A + k * H
        matrix[k][k - 1] = 1.0 / (H * H) - p(xk) / (2.0 * H)
        matrix[k][k] = -2.0 / (H * H) + q(xk)
        matrix[k][k + 1] = 1.0 / (H * H) + p(xk) / (2.0 * H)
        rhs[k] = f(xk)

    # Правая граница: y(b) = RIGHT_BC
    matrix[-1][-1] = 1.0  # только для y(b)
    rhs[-1] = RIGHT_BC
    return matrix, rhs
```

Функция gaussian_elimination реализует метод Гаусса для решения системы линейных уравнений. Сначала выбирается ведущий элемент (поворотный элемент) для каждой строки, затем выполняется приведение матрицы к верхнетреугольному виду и обратный ход для получения решения.

Листинг 5.2.2.5. Функция gaussian_elimination

```
def gaussian_elimination(matrix: List[List[float]], rhs: List[float])
-> List[float]:
    n = len(matrix)
    a = [row[:] for row in matrix]
    b = rhs[:]

    for i in range(n):
        pivot = max(range(i, n), key=lambda r: abs(a[r][i]))
        if abs(a[pivot][i]) < 1e-12:
            raise ValueError("Обнаружен нулевой поворотный элемент.")
        if pivot != i:
```

```
            a[i], a[pivot] = a[pivot], a[i]
            b[i], b[pivot] = b[pivot], b[i]

        pivot_val = a[i][i]
        for j in range(i, n):
            a[i][j] /= pivot_val
        b[i] /= pivot_val

        for r in range(i + 1, n):
            factor = a[r][i]
            if factor == 0.0:
                continue
            for j in range(i, n):
                a[r][j] -= factor * a[i][j]
            b[r] -= factor * b[i]


    x = [0.0 for _ in range(n)]
    for i in range(n - 1, -1, -1):
        x[i] = b[i] - sum(a[i][j] * x[j] for j in range(i + 1, n))
    return x
```

Функция solve_bvp(order: int) вызывает функцию build_system для создания системы и решает ее с помощью метода Гаусса.

Листинг 5.2.2.6. Функция solve_bvp

```
def solve_bvp(order: int) -> List[float]:
    matrix, rhs = build_system(order)
    return gaussian_elimination(matrix, rhs)
```

Затем идет функция print_table, которая выводит таблицу с результатами. В таблице показываются: значения точных решений для каждой точки сетки, численные решения для первого и второго порядков аппроксимации, абсолютные погрешности для каждого из решений.

Листинг 16.7. Функции вывода

```
    def print_table(xs: List[float], exact: List[float], sol1:
List[float], sol2: List[float]) -> None:
        header = (
```

```
        f"{'k':>3} {'x':>8} {'y_точно':>15} "
        f"{'y_1-й П.':>15} {'|ε_1|':>15} "
        f"{'y_2-й П.':>15} {'|ε_2|':>15}"
    )
    print(header)
    print("-" * len(header))
    for i, x in enumerate(xs):
        err1 = abs(sol1[i] - exact[i])
        err2 = abs(sol2[i] - exact[i])
        print(
            f"{i:3d} {x:8.3f} {exact[i]:15.6f}"
            f"{sol1[i]:15.6f} {err1:15.6f}"
            f"{sol2[i]:15.6f} {err2:15.6f}"
        )
```

Функция plot_solutions рисует график, на котором сравниваются аналитическое решение и численные решения для первого и второго порядков аппроксимации.

5.2.3 Результаты работы программы

На рисунке 39 представлено графическое сравнение аналитического решение с численным. Из него можно сделать вывод, что граничные условия второго порядка дают большую точность и приближенность к аналитическому решению

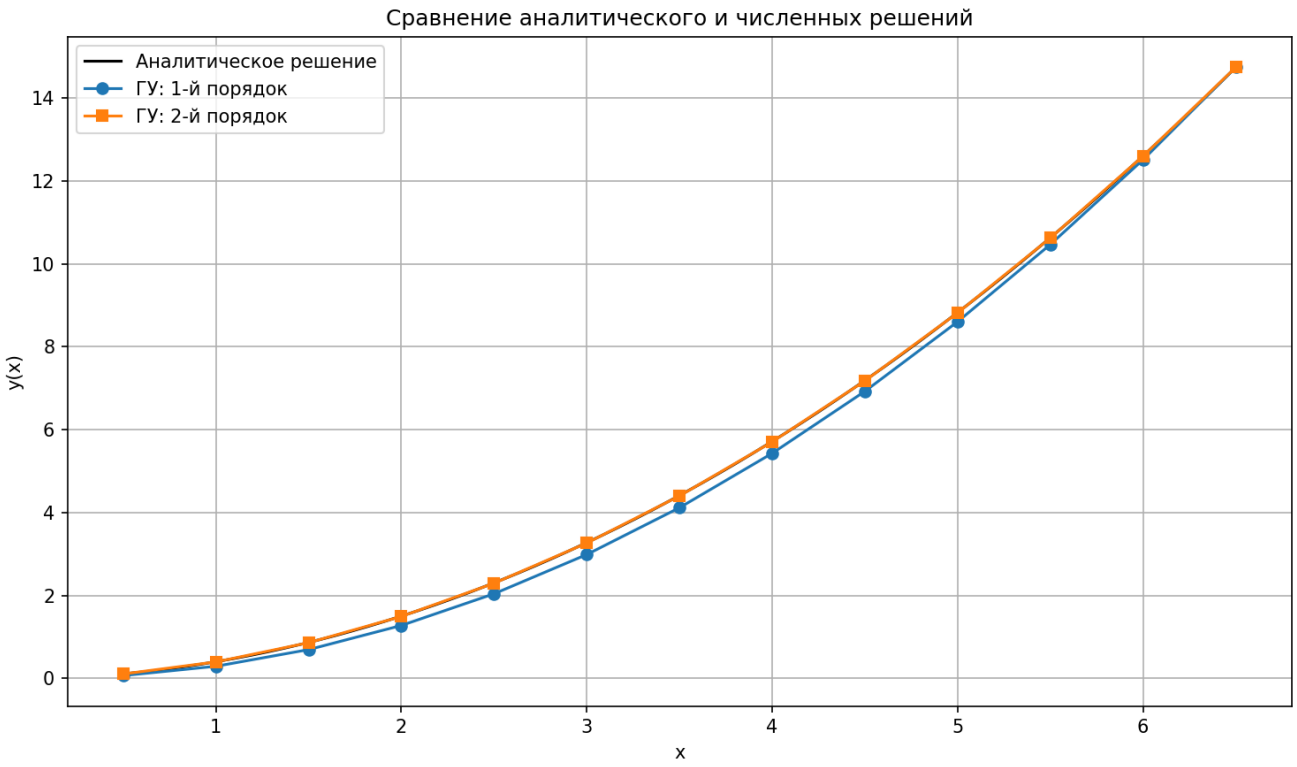

Рисунок 39, сравнение результатов

5.3 Задание 17 (26). Краевая задача. Метод стрельбы

5.3.1 Теоретическая часть

Решить краевую задачу для линейного дифференциального уравнения второго порядка. Использовать метод стрельбы. Полученные численные решения сравнить с аналитическим решением. Определить погрешность решения.

Краевая задача	Аналитическое решение
$xy'' - 2(x\ tg\ x + 1)y' + 2ytg\ x = 0$ $y(2) = -7{,}555$ $y(4{,}2) = 6{,}044 \qquad h = 0{,}11$	$y(x) = x\ tg\ x + tg\ x - x + 1$

Для решения краевой задачи для линейного дифференциального уравнения второго порядка методом стрельбы, начнём с теоретического обоснования метода и описания самого процесса решения.

Линейное дифференциальное уравнение второго порядка в стандартной форме записывается как:

$$y''(x) + p(x)y'(x) + q(x)y(x) = f(x), a \le x \le b \qquad (5.3.1.1)$$

с заданными граничными условиями:

$$y(a) = \alpha, y(b) = \beta \qquad (5.3.1.2)$$

Метод стрельбы применяется к таким задачам, когда аналитическое решение дифференциального уравнения невозможно получить непосредственно из-за сложной формы правой части или других причин. Этот метод сводит решение задачи к системе задач Коши.

Суть метода заключается в том, чтобы начать решение дифференциального уравнения с некоторого "стрельбы" — выбором начальных значений для производных в точке $x = a$. Предполагается, что для исходных значений $y(a) = \alpha$ и $y'(a) = \eta$ (где η— это параметр, который мы будем варьировать), можно решить задачу Коши для дифференциального уравнения и получить численное решение для функции $y(x)$на интервале $[a, b]$. После этого сравнивается значение $y(b)$с известным значением β, которое требуется по граничному условию. Если $y(b)$близко к β, то выбранное значение ηможно считать подходящим. Если же результат значительно отличается от β, то необходимо подбирать новое значение η, повторяя процесс, пока не достигнем нужной точности.

Процесс решения задачи методом стрельбы можно разделить на несколько этапов. На первом этапе мы выбираем начальное значение для производной $y'(a)$, обозначаем его как η_0. С помощью этого значения решаем задачу Коши для уравнения. Решение задачи Коши состоит в нахождении функции $y(x)$, которая является решением дифференциального уравнения с начальными условиями $y(a) = \alpha$и $y'(a) = \eta_0$. Для этого можно использовать численные методы, такие как метод Эйлера, метод Рунге-Кутты или другие методы интегрирования.

После получения численного решения для функции $y(x)$на интервале $[a, b]$, вычисляем значение $y(b)$. Если это значение совпадает или достаточно близко к требуемому граничному условию $y(b) = \beta$, то значение η_0можно считать подходящим. Однако, если результат не совпадает с β, то метод стрельбы заключается в следующем шаге: необходимо скорректировать значение η_0и повторить решение задачи Коши.

На практике корректировка параметра η_0может осуществляться с помощью метода итераций, в данной задаче используется метод секущих. Этот метод позволяет эффективно сужать диапазон поиска для параметра η_0, улучшая точность решения.

После того как будет найдено такое значение η, которое дает решение $y(x)$, удовлетворяющее обоим граничным условиям, можно проверить погрешность численного решения, сравнив его с аналитическим решением (если оно существует). Погрешность определяется как разность между численным

решением и точным аналитическим решением в некоторой точке или на интервале.

Таким образом, метод стрельбы позволяет решать краевые задачи для линейных дифференциальных уравнений второго порядка, даже в тех случаях, когда аналитическое решение невозможно или крайне сложно получить. Этот метод широко используется в численных методах и имеет большую практическую ценность для решения различных инженерных и физико-математических задач.

5.3.2 Практическая часть

Для начала необходимо вычислить правую часть дифференциального уравнения для задачи Коши. Это делается с помощью функции rhs, которая вычисляет производные функции и её первой производной:

Листинг 5.3.2.1. Правая часть

```
    def rhs(x: float, y: float, v: float) -> tuple[float, float]:
        dy = v
        dv = (2.0 * (x * math.tan(x) + 1.0) * v - 2.0 * y * math.tan(x)) /
x
        return dy, dv
```

Функция возвращает значения $y'(x)$и $y''(x)$, которые необходимы для дальнейшего численного интегрирования.

Для численного решения уравнения на каждом шаге используется метод Рунге-Кутты 4-го порядка, который даёт точное приближение для значений функции и её производных. В функции rk4_step рассчитываются четыре промежуточных значения для производных и их взвешенная сумма:

Листинг 5.3.2.2. Шаг численного интегрирования

```
    def rk4_step(x: float, y: float, v: float, h: float) -> tuple[float,
float]:
        k1_y, k1_v = rhs(x, y, v)
        k2_y, k2_v = rhs(x + h / 2, y + h * k1_y / 2, v + h * k1_v / 2)
        k3_y, k3_v = rhs(x + h / 2, y + h * k2_y / 2, v + h * k2_v / 2)
        k4_y, k4_v = rhs(x + h, y + h * k3_y, v + h * k3_v)

        y_next = y + (h / 6) * (k1_y + 2 * k2_y + 2 * k3_y + k4_y)
        v_next = v + (h / 6) * (k1_v + 2 * k2_v + 2 * k3_v + k4_v)
        return y_next, v_next
```

Метод Рунге-Кутты позволяет получать точное приближение функции с высоким порядком точности, что критично для задач с переменными коэффициентами.

Функция integrate решает задачу Коши для заданных начальных условий. Она использует метод Рунге-Кутты для получения значений $y(x)$и $y'(x)$на сетке:

Листинг 5.3.2.3. Функция интегрирования

```
    def integrate(alpha: float, x0: float, y0: float, h: float, steps: int)
-> tuple[list[float], list[float], list[float]]:
        xs = [x0]
        ys = [y0]
        vs = [alpha]
        x = x0
        y = y0
        v = alpha

        for _ in range(steps):
            y, v = rk4_step(x, y, v, h)
            x = round(x + h, 12)
            xs.append(x)
            ys.append(y)
            vs.append(v)

        return xs, ys, vs
```

Здесь создается сетка значений xи интегрируются значения функции и её производной. Количество шагов $steps$рассчитывается как количество делений интервала $[x_0, x_b]$.

Основная часть решения задачи реализована в функции shooting. Эта функция подбирает начальный наклон $\alpha = y'(x_0)$, чтобы решить краевую задачу методом стрельбы. Для этого используется метод секущих, который позволяет итерационно уточнять начальный наклон, основываясь на вычисленной ошибке в конце интервала $x = b$:

Листинг 5.3.2.4. Функция стрельбы

```
def shooting(
        x0: float,
        xb: float,
        y0: float,
        yb: float,
        h: float,
```

```
        alpha0: float,
        alpha1: float,
        tol: float = 1e-8,
        max_iter: int = 30,
    ) -> tuple[list[float], list[float], list[float], float, list[dict]]:
        steps = int(round((xb - x0) / h))
        history: list[dict] = []

        xs_prev, ys_prev, _ = integrate(alpha0, x0, y0, h, steps)
        res_prev = ys_prev[-1] - yb
        history.append(
            {
                "iter": 0,
                "alpha": alpha0,
                "y_end": ys_prev[-1],
                "residual": res_prev,
                "xs": xs_prev,
                "ys": ys_prev,
            }
        )

        xs_curr, ys_curr, vs_curr = integrate(alpha1, x0, y0, h, steps)
        res_curr = ys_curr[-1] - yb
        history.append(
            {
                "iter": 1,
                "alpha": alpha1,
                "y_end": ys_curr[-1],
                "residual": res_curr,
                "xs": xs_curr,
                "ys": ys_curr,
            }
        )

        iter_idx = 1
        while abs(res_curr) > tol and iter_idx < max_iter:
            denom = res_curr - res_prev
            if abs(denom) < 1e-14:
                raise RuntimeError("Метод секущих остановлен из-за малого
знаменателя.")

            alpha_next = alpha1 - res_curr * (alpha1 - alpha0) / denom
```

```
            alpha0, res_prev = alpha1, res_curr
            xs_prev, ys_prev = xs_curr, ys_curr

            alpha1 = alpha_next
            xs_curr, ys_curr, vs_curr = integrate(alpha1, x0, y0, h, steps)
            res_curr = ys_curr[-1] - yb

            iter_idx += 1
            history.append(
                {
                    "iter": iter_idx,
                    "alpha": alpha1,
                    "y_end": ys_curr[-1],
                    "residual": res_curr,
                    "xs": xs_curr,
                    "ys": ys_curr,
                }
            )

        if abs(res_curr) > tol:
            raise RuntimeError(
                "Метод стрельбы не достиг требуемой точности за допустимое число итераций."
            )

        return xs_curr, ys_curr, vs_curr, alpha1, history
```

На каждом шаге метода секущих вычисляется новая начальная производная α, и процесс повторяется до тех пор, пока погрешность не станет достаточно малой.

После того как метод стрельбы найдет оптимальное значение наклона α, вычисляется численное решение на сетке x_0до x_b. Сравнение численного решения с аналитическим даёт возможность оценить погрешность. Для этого используется функция exact_solution, которая реализует аналитическое решение задачи:

Листинг 5.3.2.5. Функция аналитического решения

```
def exact_solution(x: float) -> float:
    return x * math.tan(x) + math.tan(x) - x + 1
```

Графически результаты решения выводятся с помощью функций plot_solutions и plot_shooting_iterations. Первая функция отображает сравнение

аналитического и численного решения, а вторая — процесс итераций метода стрельбы.

5.3.3 Результаты работы программы

На рисунке 40 представлен веер решений метода стрельбы. В нем наглядно показано, как происходил подбор параметра для этой конкретной задачи.

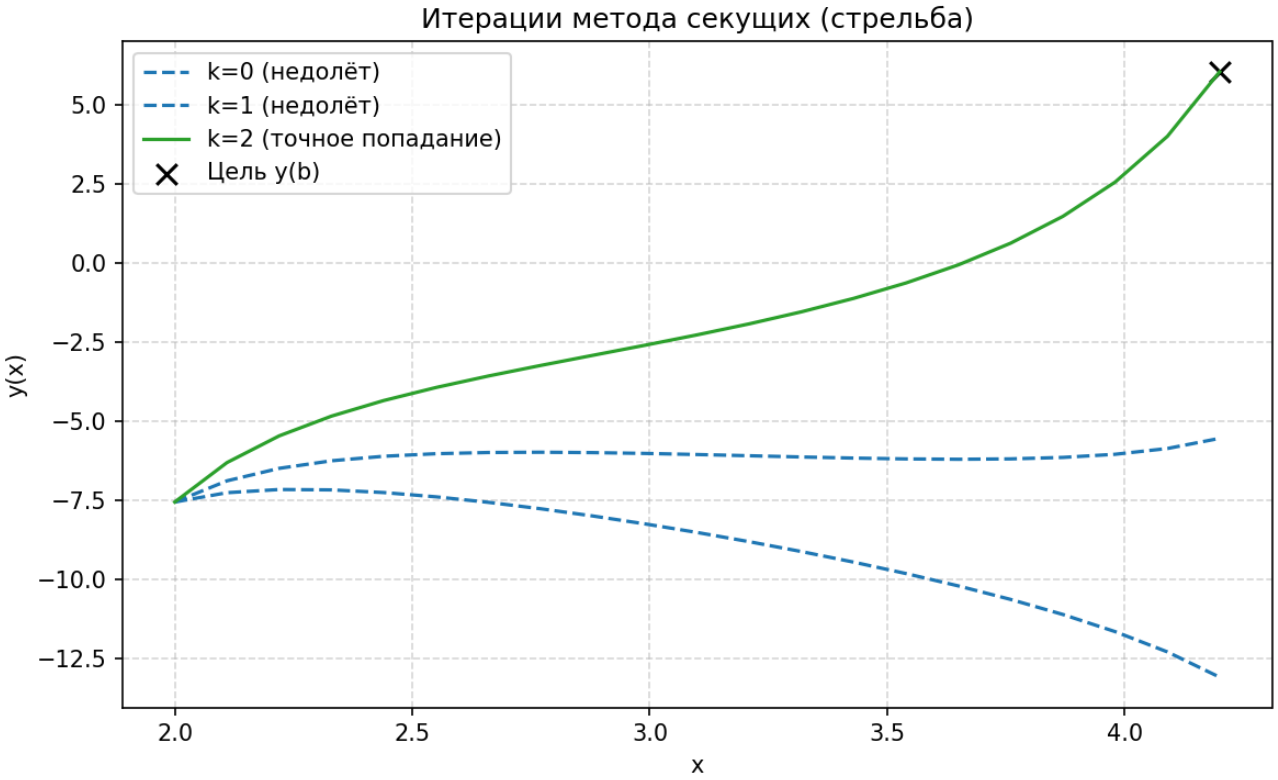

Рисунок 40, веер решений метода стрельбы

На рисунке 40 изображено графическое сравнение метода стрельбы с аналитическим решением. Как мы видим, они достаточно близки, чтобы можно было говорить об эффективности этого метода решения краевой задачи

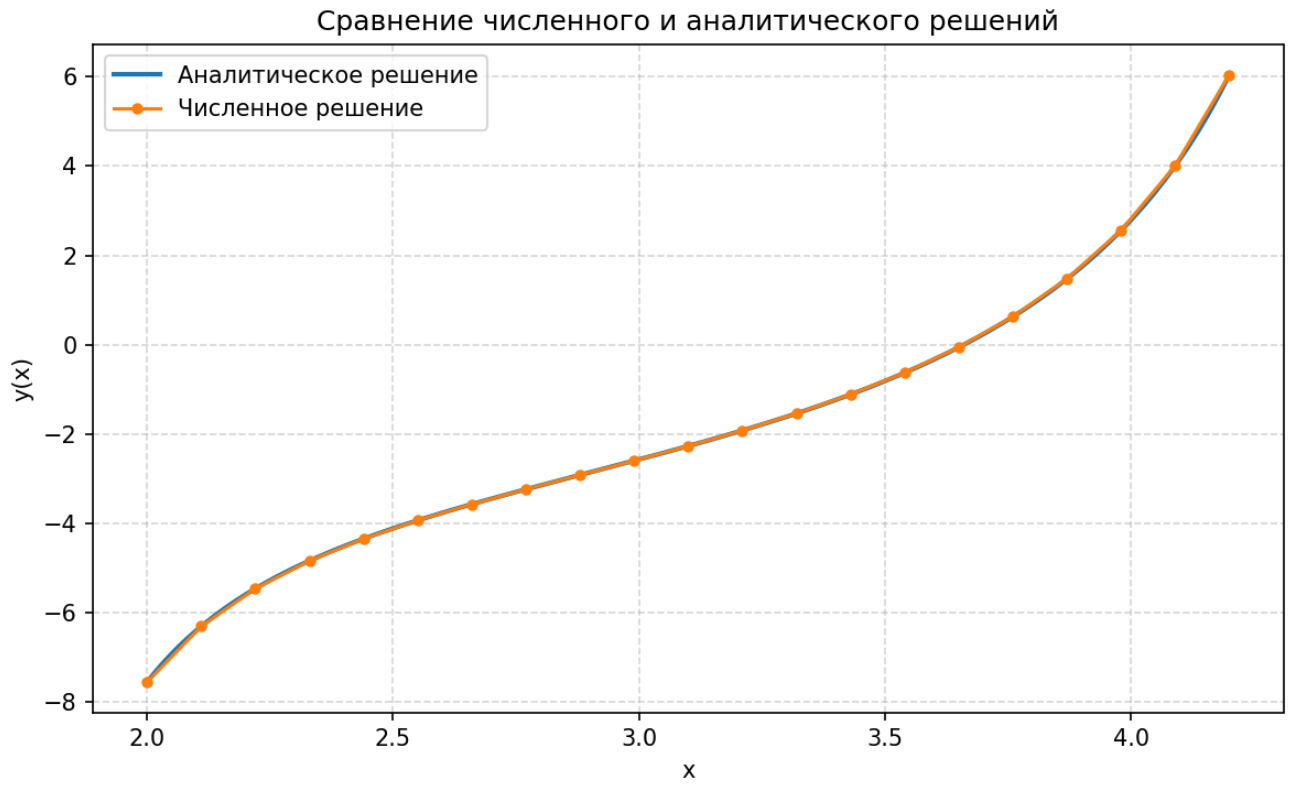

Рисунок 41, сравнение с аналитическим решением

5.3 Вывод

Практическое исследование методов решения дифференциальных уравнений позволило сравнить подходы к двум принципиально разным классам задач: задаче Коши с начальными условиями и краевой задаче. Работа с явными и неявными схемами для обыкновенных дифференциальных уравнений первого порядка (задача Коши) наглядно продемонстрировала ключевой компромисс между вычислительной сложностью и устойчивостью. Явный метод Эйлера оказался предельно прост в реализации, но предъявил жесткое ограничение на шаг интегрирования для обеспечения устойчивости. Напротив, неявный метод Эйлера и более точная схема Кранка-Николсон (или метод трапеций), требующие на каждом шаге решения уравнения (часто нелинейного) относительно следующего значения, показали абсолютную устойчивость. Это позволило использовать существенно большие шаги интегрирования без риска получить расходящееся решение, что критически важно для жестких систем. Основной практический вывод – явные схемы применимы для некритичных по скорости расчетов с плавными решениями, в то время как для сложных, быстро меняющихся или жестких систем неявные схемы становятся безальтернативным инструментом, несмотря на увеличение вычислительных затрат на шаг.

При переходе к краевым задачам были изучены и реализованы два фундаментальных подхода: конечно-разностный метод и метод стрельбы. Конечно-разностный метод основан на непосредственной дискретизации дифференциального уравнения и краевых условий во всех внутренних точках сетки, что приводит к необходимости решения системы линейных (для линейного уравнения) или нелинейных алгебраических уравнений. Его основное преимущество, подтвержденное на практике, – прямота подхода и устойчивость. Решение находится сразу для всей сетки, а итоговая СЛАУ с хорошо структурированной матрицей (например, трехдиагональной) эффективно решается методом прогонки. Этот метод надежен и часто является методом выбора для линейных задач.

Метод стрельбы, напротив, сводит краевую задачу к последовательности задач Коши. Путем параметризации недостающего начального условия и последующего применения методов решения нелинейных уравнений (например, метода Ньютона или дихотомии) находится такое начальное значение, при котором решение «промахнувшегося» начального условия попадает во вторую граничную точку. Преимущество метода стрельбы проявилось в его концептуальной простоте и возможности использования высокоточных и адаптивных решателей для задачи Коши. Однако его главный недостаток – потенциальная неустойчивость. Для жестких или чувствительных к начальным данным уравнений небольшая погрешность в определении начального параметра может привести к экспоненциальному росту ошибки на другом конце интервала, делая метод неприменимым. В ходе работы это было подтверждено на соответствующих тестовых примерах.

Таким образом, сравнительный анализ позволяет сделать следующие заключения. Для задач Коши выбор между явной и неявной схемой определяется требованиями к устойчивости и ресурсами. Для краевых задач конечно-разностный метод является более универсальным и устойчивым, особенно для линейных уравнений, в то время как метод стрельбы может быть эффективен для нелинейных задач, не обладающих жесткостью, и там, где важна высокая точность на адаптивной сетке. Полученный опыт подчеркивает, что в вычислительной практике не существует единого лучшего метода; оптимальный алгоритм всегда выбирается исходя из специфики конкретной математической модели.

6 Список литературы

1) Пирумов, У. Г. Численные методы: теория и практика / У. Г. Пирумов. — Москва: Издательство Юрайт, 2012. — 421 с.

2) Формалев, В. Ф., Ревизников, Д. Л. Численные методы / В. Ф. Формалев, Д. Л. Ревизников. — Москва: Издательство Юрайт, 2015. — 422 с.