

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

Институт информационных технологий и прикладной математики
«Кафедра вычислительной математики и программирования»

**Отчёт по лабораторным работам по курсу
«Численные методы» III курс, VI семестр**

Вариант 15

Студент: Кострюков Е.С.

Преподаватель: Киндинова В.В.

Группа: М8О-307Б-22

Дата: 28.05.2025

Оценка:

Подпись:

Лабораторная работа № 1

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

LU – разложение матрицы A представляет собой разложение матрицы A в произведение нижней и верхней треугольных матриц, т.е.

$$A = LU,$$

где L - нижняя треугольная матрица (матрица, у которой все элементы, находящиеся выше главной диагонали равны нулю, $l_{ij} = 0$ при $i < j$), U - верхняя треугольная матрица (матрица, у которой все элементы, находящиеся ниже главной диагонали равны нулю, $u_{ij} = 0$ при $i > j$).

LU – разложение может быть построено с использованием описанного выше метода Гаусса. Рассмотрим k -ый шаг метода Гаусса, на котором осуществляется обнуление поддиагональных элементов k -го столбца матрицы $A^{(k-1)}$. Как было описано выше, с этой целью используется следующая операция:

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \mu_i^{(k)} a_{kj}^{(k-1)}, \quad \mu_i^{(k)} = \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}, \quad i = \overline{k+1, n}, \quad j = \overline{k, n}.$$

В терминах матричных операций такая операция эквивалентна умножению $A^{(k)} = M_k A^{(k-1)}$, где элементы матрицы M_k определяются следующим образом

$$m_{ij}^k = \begin{cases} 1, & i = j \\ 0, & i \neq j, \quad j \neq k \\ -\mu_{j+1}^{(k)}, & i \neq j, \quad j = k \end{cases}.$$

Т.е. матрица M_k имеет вид

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -\mu_{k+1}^{(k)} & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & -\mu_n^{(k)} & 0 & 0 & 1 \end{pmatrix}.$$

При этом выражение для обратной операции запишется в виде $A^{(k-1)} = M_k^{-1} A^{(k)}$, где

$$M_k^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \mu_{k+1}^{(k)} & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \mu_n^{(k)} & 0 & 0 & 1 \end{pmatrix}$$

При этом выражение для обратной операции запишется в виде $A^{(k-1)} = M_k^{-1}A^{(k)}$, где

$$M_k^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \mu_{k+1}^{(k)} & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \mu_n^{(k)} & 0 & 0 & 1 \end{pmatrix}$$

В результате прямого хода метода Гаусса получим $A^{(n-1)} = U$,

$$A = A^{(0)} = M_1^{-1}A^{(1)} = M_1^{-1}M_2^{-1}A^{(2)} = M_1^{-1}M_2^{-1}\dots M_{n-1}^{-1}A^{(n-1)},$$

где $A^{(n-1)} = U$ - верхняя треугольная матрица, а $L = M_1^{-1}M_2^{-1}\dots M_{n-1}^{-1}$ - нижняя треугольная

$$\text{матрица, имеющая вид } L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ \mu_2^{(1)} & 1 & 0 & 0 & 0 & 0 \\ \mu_3^{(1)} & \mu_3^{(2)} & 1 & 0 & 0 & 0 \\ \dots & \dots & \dots \mu_{k+1}^{(k)} & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \mu_n^{(1)} & \mu_n^{(2)} & \mu_n^{(k)} & \mu_n^{(k+1)} & \dots & \mu_n^{(n-1)} & 1 \end{pmatrix}.$$

Таким образом, искомое разложение $A = LU$ получено.

Условие:

$$15. \begin{cases} -9 \cdot x_1 + 8 \cdot x_2 + 8 \cdot x_3 + 6 \cdot x_4 = -81 \\ -7 \cdot x_1 - 9 \cdot x_2 + 5 \cdot x_3 + 4 \cdot x_4 = -50 \\ -3 \cdot x_1 - x_2 + 8 \cdot x_3 = -69 \\ 3 \cdot x_1 - x_2 - 4 \cdot x_3 - 5 \cdot x_4 = 48 \end{cases}$$

Код программы:

```

1 import numpy as np
2
3 def find_main_element(matrix, row):
4     ...
5     Ищет индекс главного элемента в указанном столбце
6     ...
7     main_string_index = row
8     main_number = abs(matrix[row, row])
9
10    for string in range(row + 1, len(matrix)):
11        if abs(matrix[string, row]) > main_number:
12            main_number = abs(matrix[string, row])
13            main_string_index = string
14    return main_string_index
15
16 def lu_decompose(matrix):
17     ...
18     Вычисляет LU-разложение
19     ...
20     n = len(matrix)
21     L = np.zeros((n, n))
22     U = matrix.copy()
23     P = np.arange(n) # Массив перестановок строк (по умолчанию [0, 1, 2, ...])
24     swaps = 0

```

```

25     for column in range(n):
26         string_with_max_element = find_main_element(U, column)
27         # Если главный элемент не на диагонали
28         if string_with_max_element != column:
29             # Меняем строки местами
30             U[[column, string_with_max_element]] = U[[string_with_max_element, column]]
31             P[column], P[string_with_max_element] = P[string_with_max_element], P[column]
32             L[[column, string_with_max_element]] = L[[string_with_max_element, column]]
33             swaps += 1
34             # You, в прошлом месяце • added new labs ...
35
36         for string in range(column + 1, n):
37             nulling_coefficient = U[string, column] / U[column, column]
38             L[string, column] = nulling_coefficient
39             U[string, column:] -= nulling_coefficient * U[column, column:]
40
41     np.fill_diagonal(L, 1.0)
42     return L, U, P, swaps
43
44 def determinant(U, swaps):
45     ...
46     Вычисляет определитель
47     ...
48     det = np.prod(np.diag(U)) # Произведение диагональных элементов
49     return det if swaps % 2 == 0 else -det
50
51 def solve_system(L, U, P, b):
52     ...
53     Решает систему уравнений LUx = b
54     ...
55     n = len(L)
56     b_permuted = b[P] # Переставляем b в новом порядке
57
58     # Решение y[i] + sum(Ly) = Pb (прямой ход) - на диагонали единицы
59     y = np.zeros(n)
60     for i in range(n):
61         # np.dot() - сумма произведений элементов L и уже найденных y
62         y[i] = b_permuted[i] - np.dot(L[i, :i], y[:i])
63
64     # Решение Ux = y (обратный ход)
65     x = np.zeros(n)
66     for i in range(n - 1, -1, -1):
67         # np.dot() - сумма произведений элементов U и уже найденных x
68         x[i] = (y[i] - np.dot(U[i, i+1:], x[i+1:])) / U[i, i]
69
70     return x
71
72 def compute_inverse_matrix(L, U, P):
73     ...
74     Вычисляет обратную матрицу
75     ...
76     n = len(L)
77     inv = np.zeros((n, n))
78     for i in range(n):
79         e = np.zeros(n)
80         e[i] = 1
81         inv[:, i] = solve_system(L, U, P, e) # Решаем систему уравнений LUx = e[i]
82     return inv

```

Результат:

```
Детерминант: -2739
Решение системы: -1 0 -9 -3
Обратная матрица:
-0.139832 -0.105148 0.079591 -0.251917
0.057320 -0.061336 -0.009127 0.019715
-0.045272 -0.047097 0.153706 -0.092004
-0.059146 -0.013143 -0.073384 -0.281490
```

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

Метод прогонки является одним из эффективных методов решения СЛАУ с трехдиагональными матрицами, возникающих при конечно-разностной аппроксимации задач для обыкновенных дифференциальных уравнений (ОДУ) и уравнений в частных производных второго порядка и является частным случаем метода Гаусса. Рассмотрим следующую СЛАУ:

$$\left\{ \begin{array}{l} a_1 = 0 \quad b_1 x_1 + c_1 x_2 = d_1 \\ a_2 x_1 + b_2 x_2 + c_2 x_3 = d_2 \\ a_3 x_2 + b_3 x_3 + c_3 x_4 = d_3 \\ \dots \\ a_{n-1} x_{n-2} + b_{n-1} x_{n-1} + c_{n-1} x_n = d_{n-1} \\ a_n x_{n-1} + b_n x_n = d_n, \quad c_n = 0, \end{array} \right. \quad (1.1)$$

решение которой будем искать в виде

$$x_i = P_i x_{i+1} + Q_i, \quad i = \overline{1, n}$$

(1.2)

где $P_i, Q_i, i = \overline{1, n}$ - прогоночные коэффициенты, подлежащие определению. Для их определения выразим из первого уравнения СЛАУ (1.1) x_1 через x_2 , получим:

$$x_1 = \frac{-c_1}{b_1} x_2 + \frac{d_1}{b_1} = P_1 x_2 + Q_1, \quad (1.3)$$

откуда

$$P_1 = \frac{-c_1}{b_1}, \quad Q_1 = \frac{d_1}{b_1}.$$

Из второго уравнения СЛАУ (1.1) с помощью (1.3) выразим x_2 через x_3 , получим:

$$x_2 = \frac{-c_2}{b_2 + a_2 P_1} x_3 + \frac{d_2 - a_2 Q_1}{b_2 + a_2 P_1} = P_2 x_3 + Q_2,$$

откуда

$$P_2 = \frac{-c_2}{b_2 + a_2 P_1}, \quad Q_2 = \frac{d_2 - a_2 Q_1}{b_2 + a_2 P_1}.$$

Продолжая этот процесс, получим из i -го уравнения СЛАУ (1.1):

$$x_i = \frac{-c_i}{b_i + a_i P_{i-1}} x_{i+1} + \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}},$$

следовательно

$$P_i = \frac{-c_i}{b_i + a_i P_{i-1}}, \quad Q_i = \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}}.$$

Из последнего уравнения СЛАУ имеем

$$x_n = \frac{-c_n}{b_n + a_n P_{n-1}} x_{n+1} + \frac{d_n - a_n Q_{n-1}}{b_n + a_n P_{n-1}} = 0 \cdot x_{n+1} + Q_n,$$

то есть

$$P_n = 0 \text{ (т.к. } c_n = 0), \quad Q_n = \frac{d_n - a_n Q_{n-1}}{b_n + a_n P_{n-1}} = x_n.$$

Таким образом, прямой ход метода прогонки по определению прогоночных коэффициентов $P_i, Q_i, i = \overline{1, n}$ завершен. В результате прогоночные коэффициенты вычисляются по следующим формулам:

$$P_i = \frac{-c_i}{b_i + a_i P_{i-1}}, \quad Q_i = \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}}, \quad i = \overline{2, n-1};$$

(1.4)

$$P_1 = \frac{-c_1}{b_1}, \quad Q_1 = \frac{d_1}{b_1}, \text{ так как } a_1 = 0, \quad i = 1;$$

(1.5)

$$P_n = 0, \text{ т.к. } c_n = 0, \quad Q_n = \frac{d_n - a_n Q_{n-1}}{b_n + a_n P_{n-1}}, \quad i = n.$$

Обратный ход метода прогонки осуществляется в соответствии с выражением (1.2)

$$\begin{cases} x_n = P_n x_{n+1} + Q_n = 0 \cdot x_{n+1} + Q_n = Q_n \\ x_{n-1} = P_{n-1} x_n + Q_{n-1} \\ x_{n-2} = P_{n-2} x_{n-1} + Q_{n-2} \\ \dots \\ x_1 = P_1 x_2 + Q_1. \end{cases}$$

(1.7)

Условие:

$$15. \begin{cases} 16 \cdot x_1 - 8 \cdot x_2 = 0 \\ -7 \cdot x_1 - 16 \cdot x_2 + 5 \cdot x_3 = -123 \\ 4 \cdot x_2 + 12 \cdot x_3 + 3 \cdot x_4 = -68 \\ -4 \cdot x_3 + 12 \cdot x_4 - 7 \cdot x_5 = 104 \\ -x_4 + 7 \cdot x_5 = 20 \end{cases}$$

Код программы:

```
1 import numpy as np
2
3 a = np.array([0, -7, 4, -4, -1], dtype=float) # Нижняя диагональ (a1 = 0)
4 b = np.array([16, -16, 12, 12, 7], dtype=float) # Главная диагональ
5 c = np.array([-8, 5, 3, -7, 0], dtype=float) # Верхняя диагональ (c5 = 0)
6 d = np.array([0, -123, -68, 104, 20], dtype=float) # Правая часть
7
8 n = len(b)
9
10 is_stable = True
11 for i in range(n):
12     ai = a[i] if i > 0 else 0
13     ci = c[i] if i < n - 1 else 0
14     if abs(b[i]) < abs(ai) + abs(ci):
15         is_stable = False
16         print(f"Условие устойчивости нарушено на i={i+1}: |b[{i}]| < |a[{i}]| + |c[{i}]|")
17
18 if is_stable:
19     print("Матрица удовлетворяет условиям устойчивости.")
20
21 # Прямой ход
22 P = np.zeros(n, dtype=float)
23 Q = np.zeros(n, dtype=float)
24 x = np.zeros(n, dtype=float)
25
26 P[0] = -c[0] / b[0]
27 Q[0] = d[0] / b[0]
28 for i in range(1, n):
29     denominator = b[i] + a[i] * P[i - 1]
30     P[i] = -c[i] / denominator if i < n - 1 else 0 # Учитываем, что cn = 0
31     Q[i] = (d[i] - a[i] * Q[i - 1]) / denominator
32 You, в прошлом месяце • added new labs
33 # Обратный ход
34 x[-1] = Q[-1] # Т.к. cn = 0
35 for i in range(n - 2, -1, -1):
36     x[i] = P[i] * x[i + 1] + Q[i]
37
38 print("Решение СЛАУ:")
39 for i, xi in enumerate(x, 1):
40     print(f"x{i} = {xi:.6f}")
41 else:
42     print("Решение не выполняется из-за нарушения условий устойчивости.")
```

Результат:

```
Матрица удовлетворяет условиям устойчивости.
Решение СЛАУ:
x1 = 2.000000
x2 = 4.000000
x3 = -9.000000
x4 = 8.000000
x5 = 4.000000
```

1.3. Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное

обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности

Метод простых итераций

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

их размерностью. Для решения СЛАУ с разреженными матрицами предпочтительнее использовать итерационные методы.

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

Метод простых итераций

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

их размерностью. Для решения СЛАУ с разреженными матрицами предпочтительнее использовать итерационные методы.

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

Рассмотрим СЛАУ

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{cases} \quad (1.16)$$

с невырожденной матрицей ($\det A \neq 0$).

Приведем СЛАУ к эквивалентному виду

$$\begin{cases} x_1 = \beta_1 + \alpha_{11}x_1 + \alpha_{12}x_2 + \dots + \alpha_{1n}x_n \\ x_2 = \beta_2 + \alpha_{21}x_1 + \alpha_{22}x_2 + \dots + \alpha_{2n}x_n \\ \dots \\ x_n = \beta_n + \alpha_{n1}x_1 + \alpha_{n2}x_2 + \dots + \alpha_{nn}x_n \end{cases} \quad (1.17)$$

или в векторно-матричной форме

$$x = \beta + \alpha x.$$

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad \beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_n \end{pmatrix}, \quad \alpha = \begin{pmatrix} \alpha_{11} & \dots & \alpha_{1n} \\ \vdots & \dots & \vdots \\ \alpha_{n1} & \dots & \alpha_{nn} \end{pmatrix}.$$

Такое приведение может быть выполнено различными способами. Одним из наиболее распространенных является следующий.

Такое приведение может быть выполнено различными способами. Одним из наиболее распространенных является следующий.

Разрешим систему (1.16) относительно неизвестных при ненулевых диагональных элементах $a_{ii} \neq 0$, $i = \overline{1, n}$ (если какой-либо коэффициент на главной диагонали равен нулю, достаточно соответствующее уравнение поменять местами с любым другим уравнением). Получим следующие выражения для компонентов вектора β и матрицы α эквивалентной системы:

$$\beta_i = \frac{b_i}{a_{ii}}; \quad \alpha_{ij} = -\frac{a_{ij}}{a_{ii}}, \quad i, j = \overline{1, n}, \quad i \neq j; \quad \alpha_{ii} = 0, \quad i = j, \quad i = \overline{1, n}. \quad (1.18)$$

При таком способе приведения исходной СЛАУ к эквивалентному виду метод простых итераций носит название метода Якоби.

В качестве нулевого приближения $x^{(0)}$ вектора неизвестных примем вектор правых частей $x^{(0)} = \beta$ или $(x_1^{(0)} \quad x_2^{(0)} \quad \dots \quad x_n^{(0)})^T = (\beta_1 \quad \beta_2 \quad \dots \quad \beta_n)^T$. Тогда *метод простых итераций* примет вид:

$$\begin{cases} x^{(0)} = \beta \\ x^{(1)} = \beta + \alpha x^{(0)} \\ x^{(2)} = \beta + \alpha x^{(1)} \\ \dots \\ x^{(k)} = \beta + \alpha x^{(k-1)}. \end{cases} \quad (1.19)$$

Из (1.19) видно преимущество итерационных методов по сравнению, например, с рассмотренным выше методом Гаусса. В вычислительном процессе участвуют только произведения матрицы на вектор, что позволяет работать только с ненулевыми элементами матрицы, значительно упрощая процесс хранения и обработки матриц.

Имеет место следующее достаточное условие сходимости метода простых итераций.

Метод простых итераций (1.19) сходится к единственному решению СЛАУ (1.17) (а следовательно и к решению исходной СЛАУ (1.16)) при любом начальном приближении $x^{(0)}$, если какая-либо норма матрицы α эквивалентной системы меньше единицы $\|\alpha\| < 1$.

Если используется метод Якоби (выражения (1.18) для эквивалентной СЛАУ), то достаточным условием сходимости является *диагональное преобладание матрицы* A , т.е.

$$|a_{ii}| > \sum_{j=1, j \neq i}^n |a_{ij}| \quad \forall i \quad (\text{для каждой строки матрицы } A \text{ модули элементов, стоящих на главной}$$

диагонали, больше суммы модулей недиагональных элементов). Очевидно, что в этом случае $\|\alpha\|_c$ меньше единицы и, следовательно, итерационный процесс (1.19) сходится.

Приведем также необходимое и достаточное условие сходимости метода простых итераций. Для сходимости итерационного процесса (1.19) необходимо и достаточно, чтобы спектр матрицы α эквивалентной системы лежал внутри круга с радиусом, равным единице.

При выполнении достаточного условия сходимости оценка погрешности решения на k -ой итерации дается выражением:

$$\|x^{(k)} - x^*\| \leq \varepsilon^{(k)} = \frac{\|\alpha\|}{1-\|\alpha\|} \|x^{(k)} - x^{(k-1)}\|, \quad (1.20)$$

где x^* - точное решение СЛАУ.

Процесс итераций останавливается при выполнении условия $\varepsilon^{(k)} \leq \varepsilon$, где ε - задаваемая вычислителем точность.

Принимая во внимание, что из (1.20) следует неравенство $\|x^{(k)} - x^*\| \leq \frac{\|\alpha\|^k}{1-\|\alpha\|} \|x^{(1)} - x^{(0)}\|$, можно получить априорную оценку необходимого для достижения заданной точности числа итераций. При использовании в качестве начального приближения вектора β такая оценка определится неравенством:

$$\frac{\|\alpha\|^{k+1}}{1-\|\alpha\|} \|\beta\| \leq \varepsilon,$$

откуда получаем априорную оценку числа итераций k при $\|\alpha\| < 1$

$$k + 1 \geq \frac{\lg \varepsilon - \lg \|\beta\| + \lg(1 - \|\alpha\|)}{\lg \|\alpha\|}.$$

Следует подчеркнуть, что это неравенство дает завышенное число итераций k , поэтому редко используется на практике.

Условие:

$$15. \begin{cases} -14 \cdot x_1 + 6 \cdot x_2 + x_3 - 5 \cdot x_4 = 95 \\ -6 \cdot x_1 + 27 \cdot x_2 + 7 \cdot x_3 - 6 \cdot x_4 = -41 \\ 7 \cdot x_1 - 5 \cdot x_2 - 23 \cdot x_3 - 8 \cdot x_4 = 69 \\ 3 \cdot x_1 - 8 \cdot x_2 - 7 \cdot x_3 + 26 \cdot x_4 = 27 \end{cases}$$

Код программы:

```

1 import numpy as np
2
3 def transform_system(A, b):
4     ...
5     # Преобразует систему Ax = b в вид x = αx + β.
6     ...
7     n = len(b)
8     alpha = -A / A.diagonal()[:, None] # Делим каждую строку на A_ii и меняем знак
9     np.fill_diagonal(alpha, 0) # Зануляем главную диагональ (A_ii -> 0)
10    beta = b / A.diagonal()
11    return alpha, beta
12
13 def calculate_matrix_norm(alpha):
14     ...
15     # Вычисляет норму матрицы
16     ...
17     norms = {
18         '1': np.max(np.sum(np.abs(alpha), axis=0)), # Столбцовая норма
19         'C': np.max(np.sum(np.abs(alpha), axis=1)), # Строковая норма
20         '2': np.sqrt(np.sum(alpha**2))
21     }
22     norm_type, min_norm = min(norms.items(), key=lambda x: x[1])
23     print(f"Выбрана норма {norm_type} для матрицы альфа")
24     return min_norm, norm_type
25
26 def calculate_vector_norm(vec, norm_type):
27     ...
28     # Вычисляет норму вектора в соответствии с нормой матрицы.
29     ...
30     return {
31         '1': np.sum(np.abs(vec)),
32         '2': np.sqrt(np.sum(vec**2)),
33         'C': np.max(np.abs(vec))
34     }[norm_type]
35
36 def check_convergence(x, x_prev, epsilon, norm_alpha, norm_type, use_matrix_norm):
37     ...
38     # Проверяет условие сходимости:
39     # - Если use_matrix_norm=True, учитывается норма матрицы.
40     # - Если False, используется только норма разности векторов.
41     ...
42     error_norm = calculate_vector_norm(x - x_prev, norm_type)
43     return (norm_alpha * error_norm / (1 - norm_alpha)) <= epsilon if use_matrix_norm else error_norm
44
45 def iterative_solver(alpha, beta, epsilon, norm_alpha, norm_type, update_rule, use_matrix_norm):
46     ...
47     # Универсальный итерационный метод (простые итерации / Зейдель).
48     # update_rule: функция, определяющая обновление x (обычные итерации / Зейдель).
49     ...
50     n = len(beta)
51     x = np.zeros(n)
52     iterations = 0
53
54     while True:
55         x_prev = x.copy()
56         x = update_rule(alpha, beta, x, x_prev)
57         iterations += 1

```

```

58         if check_convergence(x, x_prev, epsilon, norm_alpha, norm_type, use_matrix_norm):
59             break
60
61     return x, iterations
62
63
64 def simple_iteration_update(alpha, beta, x, x_prev):
65     '''Правило обновления для метода простых итераций'''
66     return beta + np.dot(alpha, x_prev)
67
68 def seidel_update(alpha, beta, x, x_prev):
69     '''Правило обновления для метода Зейделя'''
70     n = len(beta)
71     for i in range(n):
72         sum1 = np.dot(alpha[i, :i], x[:i])
73         sum2 = np.dot(alpha[i, i+1:], x_prev[i+1:])
74         x[i] = beta[i] + sum1 + sum2
75
76     return x

```

Результат:

```

Выбрана норма С для матрицы альфа
Решение (простые итерации): [-7.99957441 -2.00032821 -5.00015187  0.00031017], итераций: 14
Решение (Зейдель): [-7.99988274 -1.999729   -5.00007891  0.00004861], итераций: 6

```

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

Метод вращений Якоби применим только для симметрических матриц $A_{n \times n}$ ($A = A^T$) и решает полную проблему собственных значений и собственных векторов таких матриц. Он основан на отыскании с помощью итерационных процедур матрицы U в преобразовании подобия $\Lambda = U^{-1}AU$, а поскольку для симметрических матриц A матрица преобразования подобия U является ортогональной ($U^{-1} = U^T$), то $\Lambda = U^TAU$, где Λ - диагональная матрица с собственными значениями на главной диагонали

$$\Lambda = \begin{pmatrix} \lambda_1 & \dots & 0 \\ \dots & \ddots & \dots \\ 0 & \dots & \lambda_n \end{pmatrix}.$$

Пусть дана симметрическая матрица A . Требуется для нее вычислить с точностью ε все собственные значения и соответствующие им собственные векторы. Алгоритм метода вращения следующий:

Пусть известна матрица $A^{(k)}$ на k -й итерации, при этом для $k=0$ $A^{(0)} = A$.

1. Выбирается максимальный по модулю недиагональный элемент $a_{ij}^{(k)}$ матрицы $A^{(k)}$ ($|a_{ij}^{(k)}| = \max_{l < m} |a_{lm}^{(k)}|$).

2. Ставится задача найти такую ортогональную матрицу $U^{(k)}$, чтобы в результате преобразования подобия $A^{(k+1)} = U^{(k)T}A^{(k)}U^{(k)}$ произошло обнуление элемента $a_{ij}^{(k+1)}$ матрицы $A^{(k+1)}$. В качестве ортогональной матрицы выбирается матрица вращения, имеющая следующий вид:

$$U^k = \begin{pmatrix} 1 & & \vdots & & \vdots & & \\ & \ddots & & \vdots & & \vdots & 0 \\ & & 1 & \vdots & & \vdots & \\ \cdots & \cos \varphi^{(k)} & \cdots & -\sin \varphi^{(k)} & \cdots & & i \\ & \vdots & 1 & & \vdots & & \\ & & & \ddots & & \vdots & \\ & & & & 1 & \vdots & \\ \cdots & \sin \varphi^{(k)} & \cdots & \cos \varphi^{(k)} & \cdots & & j \\ & \vdots & & & \vdots & 1 & \\ 0 & \vdots & & & \vdots & & \ddots \\ & & & & & & 1 \end{pmatrix},$$

В матрице вращения на пересечении i -й строки и j -го столбца находится элемент $u_{ij}^{(k)} = -\sin \varphi^{(k)}$, где $\varphi^{(k)}$ - угол вращения, подлежащий определению. Симметрично относительно главной диагонали (j -я строка, i -й столбец) расположен элемент $u_{ji}^{(k)} = \sin \varphi^{(k)}$; Диагональные элементы $u_{ii}^{(k)}$ и $u_{jj}^{(k)}$ равны соответственно $u_{ii}^{(k)} = \cos \varphi^{(k)}$, $u_{jj}^{(k)} = \cos \varphi^{(k)}$; другие диагональные элементы $u_{mm}^{(k)} = 1, m = \overline{1, n}, m \neq i, m \neq j$; остальные элементы в матрице вращения $U^{(k)}$ равны нулю.

Угол вращения $\varphi^{(k)}$ определяется из условия $a_{ij}^{(k+1)} = 0$:

$$\varphi^{(k)} = \frac{1}{2} \operatorname{arctg} \frac{2a_{ij}^{(k)}}{a_{ii}^{(k)} - a_{jj}^{(k)}},$$

причем если $a_{ii}^{(k)} = a_{jj}^{(k)}$, то $\varphi^{(k)} = \frac{\pi}{4}$.

3. Строится матрица $A^{(k+1)}$

$$A^{(k+1)} = U^{(k)T} A^{(k)} U^{(k)},$$

в которой элемент $a_{ij}^{(k+1)} \approx 0$.

В качестве критерия окончания итерационного процесса используется условие малости суммы квадратов внедиагональных элементов:

$$t(A^{(k+1)}) = \left(\sum_{l,m;l < m} (a_{lm}^{(k+1)})^2 \right)^{1/2}.$$

Если $t(A^{(k+1)}) > \varepsilon$, то итерационный процесс

$$A^{(k+1)} = U^{(k)T} A^{(k)} U^{(k)} = U^{(k)T} U^{(k-1)T} \dots U^{(0)T} A^{(0)} U^{(0)} U^{(1)} \dots U^{(k)}$$

продолжается. Если $t(A^{(k+1)}) < \varepsilon$, то итерационный процесс останавливается, и в качестве искомых собственных значений принимаются $\lambda_1 \approx a_{11}^{(k+1)}$, $\lambda_2 \approx a_{22}^{(k+1)}$, ..., $\lambda_n \approx a_{nn}^{(k+1)}$.

Координатными столбцами собственных векторов матрицы A в единичном базисе будут столбцы матрицы $U = U^{(0)} U^{(1)} \dots U^{(k)}$, т.е.

$$(x^1)^T = (u_{11} u_{21} \dots u_{n1}), \quad (x^2)^T = (u_{12} u_{22} \dots u_{n2}), \quad (x^n)^T = (u_{1n} u_{2n} \dots u_{nn}),$$

причем эти собственные векторы будут ортогональны между собой, т.е. $(x^l, x^m) \approx 0$, $l \neq m$.

Условие:

$$15. \begin{pmatrix} -3 & -1 & 3 \\ -1 & 8 & 1 \\ 3 & 1 & 5 \end{pmatrix}$$

Код программы:

```

1  import math
2  |      You, в прошлом месяце • added new labs
3  def is_symmetric(matrix):
4      """Проверяет, является ли матрица симметричной."""
5      n = len(matrix)
6      for i in range(n):
7          for j in range(n):
8              if matrix[i][j] != matrix[j][i]:
9                  return False
10     return True
11
12 def find_max_off_diagonal(matrix):
13     """Находит индекс максимального по модулю внедиагонального элемента."""
14     n = len(matrix)
15     max_value = 0
16     i_max, j_max = 0, 1
17     for i in range(n):
18         for j in range(i + 1, n):
19             if abs(matrix[i][j]) > abs(max_value):
20                 max_value = matrix[i][j]
21                 i_max, j_max = i, j
22     return i_max, j_max, max_value
23

```

```

24 def sum_off_diagonal_squares(matrix):
25     """Вычисляет сумму квадратов внедиагональных элементов матрицы."""
26     n = len(matrix)
27     total = 0
28     for i in range(n):
29         for j in range(n):
30             if i != j:
31                 total += matrix[i][j] ** 2
32     return total
33
34 def create_rotation_matrix(n, i_max, j_max, phi):
35     """Создает матрицу поворота U."""
36     U = [[1 if i == j else 0 for j in range(n)] for i in range(n)]
37     U[i_max][i_max] = math.cos(phi)
38     U[j_max][j_max] = math.cos(phi)
39     U[i_max][j_max] = -math.sin(phi)
40     U[j_max][i_max] = math.sin(phi)
41     return U
42
43 def multiply_matrices(A, B):
44     """Перемножает две матрицы."""
45     rows, cols = len(A), len(B[0])
46     common_dim = len(B)
47     result = [[0] * cols for _ in range(rows)]
48     for i in range(rows):
49         for j in range(cols):
50             for k in range(common_dim):
51                 result[i][j] += A[i][k] * B[k][j]
52     return result
53
54 def transpose_matrix(matrix):
55     """Транспонирует матрицу."""
56     n, m = len(matrix), len(matrix[0])
57     return [[matrix[j][i] for j in range(n)] for i in range(m)]
58
59 def jacobi_eigenvalues_and_vectors(matrix, epsilon=1e-5, max_iterations=100):
60     """
61     Вычисляет собственные значения и собственные векторы симметрической матрицы методом Якоби.
62     """
63     if not is_symmetric(matrix):
64         raise ValueError("Матрица не является симметричной")
65
66     n = len(matrix)
67     eigenvectors = [[1 if i == j else 0 for j in range(n)] for i in range(n)]
68
69     iterations = 0
70     while True:
71         if sum_off_diagonal_squares(matrix) < epsilon or iterations >= max_iterations:
72             break
73
74         i_max, j_max, max_off_diag = find_max_off_diagonal(matrix)
75         iterations += 1
76         phi = 0.5 * math.atan(2 * max_off_diag / (matrix[i_max][i_max] - matrix[j_max][j_max]))
77
78         U = create_rotation_matrix(n, i_max, j_max, phi)
79         matrix = multiply_matrices(multiply_matrices(transpose_matrix(U), matrix), U) # После этого внедиагональный элемент a[ij] стремится к нулю
80         eigenvectors = multiply_matrices(eigenvectors, U) # Обновление матрицы собственных векторов
81
82     eigenvalues = [matrix[i][i] for i in range(n)]
83     return eigenvalues, eigenvectors
84
85     matrix = [[-3, -1, 3],
86               [-1, 8, 1],
87               [3, 1, 5]]
88
89     eigenvalues, eigenvectors = jacobi_eigenvalues_and_vectors(matrix)

```

Решение:

```
Собственные значения:
```

```
λ_0 = -4.132310
```

```
λ_1 = 8.303678
```

```
λ_2 = 5.828632
```

```
Собственные векторы:
```

```
v_0 = [0.9414623110639988, -0.010296356972543275, 0.3369609797574905]
```

```
v_1 = [0.10403030829920959, 0.9596256016610862, -0.2613357985270473]
```

```
v_2 = [-0.3206655762647518, 0.28109195945400506, 0.9045224698862391]
```

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

В основе QR-алгоритма лежит представление матрицы в виде $A = QR$, где Q - ортогональная матрица ($Q^{-1} = Q^T$), а R - верхняя треугольная. Такое разложение существует для любой квадратной матрицы. Одним из возможных подходов к построению QR разложения является использование преобразования Хаусхолдера, позволяющего обратить в нуль группу поддиагональных элементов столбца матрицы.

Преобразование Хаусхолдера осуществляется с использованием матрицы Хаусхолдера, имеющей следующий вид:

$$H = E - \frac{2}{v^T v} vv^T, \quad (1.29)$$

где v - произвольный ненулевой вектор-столбец, E - единичная матрица, vv^T - квадратная матрица того же размера.

Легко убедиться, что любая матрица такого вида является симметричной и ортогональной. При этом произвол в выборе вектора v дает возможность построить матрицу, отвечающую некоторым дополнительным требованиям.

Рассмотрим случай, когда необходимо обратить в нуль все элементы какого-либо вектора кроме первого, т.е. построить матрицу Хаусхолдера такую, что

$$\tilde{b} = Hb, \quad b = (b_1, b_2, \dots, b_n)^T, \quad \tilde{b} = (\tilde{b}_1, 0, \dots, 0)^T.$$

Тогда вектор v определится следующим образом:

$$v = b + sign(b_1) \|b\|_2 e_1, \quad (1.30)$$

где $\|b\|_2 = \left(\sum_i b_i^2 \right)^{1/2}$ - евклидова норма вектора, $e_1 = (1, 0, \dots, 0)^T$.

Применяя описанную процедуру с целью обнуления поддиагональных элементов каждого из столбцов исходной матрицы, можно за фиксированное число шагов получить ее QR – разложение.

Условие:

$$15. \begin{pmatrix} 1 & 7 & -1 \\ -2 & 2 & -2 \\ 9 & -7 & 3 \end{pmatrix}$$

Код программы:

```

1 import numpy as np
2 |     You, в прошлом месяце • added new labs ...
3 def euclidean_norm(a):
4     """Вычисляет евклидову норму вектора a."""
5     return np.sqrt(np.sum(a**2))
6
7 def householder_reflection(a, i, n):
8     """Вычисляет полную матрицу Хаусхолдера для вектора a, начиная с позиции i."""
9     v = np.zeros(n)
10    v[i:] = a
11    v[i] += np.sign(a[0]) * euclidean_norm(a)
12    beta = np.dot(v, v)
13    H = np.eye(n) - 2 * np.outer(v, v) / beta
14    return H
15
16 def qr_decomposition(A):
17     """Выполняет QR-разложение матрицы A с использованием преобразования Хаусхолдера."""
18     n = A.shape[0]
19     Q = np.eye(n)
20     R = A.copy()
21
22     for i in range(n - 1):
23         # вектор a, содержит текущий диагональный элемент и все элементы ниже него
24         a = R[i:, i]
25
26         # Вычисляем полную матрицу Хаусхолдера
27         H = householder_reflection(a, i, n)
28
29         # Обновляем R и Q
30         R = np.dot(H, R) # зануляет нужные поддиагональные элементы
31         Q = np.dot(Q, H)
32
33     return Q, R
34
35 def extract_eigenvalues(A, eps=1e-10, max_iterations=100):
36     n = A.shape[0]
37     eigenvalues = []
38     i = 0
39     while i < n:
40         if i == n - 1 or np.abs(A[i+1, i]) < eps:
41             # 1x1 блок (вещественное собственное значение)
42             eigenvalues.append(A[i, i])
43             i += 1

```

```

44     else:
45         # 2x2 блок (комплексно-сопряженная пара)
46         block = A[i:i+2, i:i+2]
47         lambda_prev = None
48         converged = False
49
50         for _ in range(max_iterations):
51             current_eigvals = np.linalg.eigvals(block)
52
53             if lambda_prev is not None:
54                 if all(np.abs(current_eigvals - lambda_prev) < eps):
55                     converged = True
56                     break
57
58             lambda_prev = current_eigvals
59             Q, R = qr_decomposition_2x2(block)
60             block = R @ Q
61
62             if not converged:
63                 print(f"Блок 2x2 в позиции {i} не сошёлся за {max_iterations} итераций")
64
65             eigenvalues.extend(lambda_prev)
66             i += 2
67
68     return np.array(eigenvalues)
69
70
71 def qr_decomposition_2x2(A):
72     """QR-разложение 2x2 матрицы с использованием преобразований Хаусхолдера."""
73     n = 2
74     Q = np.eye(n)
75     R = A.copy()
76
77     a = R[:, 0]
78     norm_a = euclidean_norm(a)
79     v = a.copy()
80     v[0] += np.sign(a[0]) * norm_a
81     v = v / euclidean_norm(v)
82
83     H = np.eye(n) - 2 * np.outer(v, v)
84     R = H @ R
85     Q = Q @ H
86
87     return Q, R
88
89
90 def qr_algorithm(A, epsilon, max_iterations=1000):
91     """Находит собственные значения и векторы матрицы A с использованием QR-алгоритма."""
92     n = A.shape[0]
93     Ak = A.copy()
94     Q_total = np.eye(n)
95
96     for _ in range(max_iterations):
97         Q, R = qr_decomposition(Ak)
98         Q_total = Q_total @ Q
99         Ak = R @ Q
100
101        # Проверка сходимости (сумма квадратов поддиагональных элементов)
102        off_diag_sum = np.sqrt(np.sum(np.tril(Ak, -1)**2))
103        if off_diag_sum < epsilon:
104            break
105
106    eigenvalues = extract_eigenvalues(Ak, eps=epsilon)
107    eigenvectors = Q_total
108    return eigenvalues, eigenvectors

```

Решение:

Собственные значения:

$$\lambda_1 = 4.13 + 4.66i$$

$$\lambda_2 = 4.13 - 4.66i$$

$$\lambda_3 = -2.27$$

Собственные векторы:

$$v_1 = [-0.87, -0.36, 0.34]$$

$$v_2 = [0.42, -0.17, 0.89]$$

$$v_3 = [-0.26, 0.92, 0.30]$$

Лабораторная работа №2

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

Условие:

15. $\sin x - x^2 + 1 = 0$.

Метод Ньютона (метод касательных). При нахождении корня уравнения (2.1) методом Ньютона, итерационный процесс определяется формулой

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})} \quad k = 0, 1, 2, \dots \quad (2.2)$$

Для начала вычислений требуется задание начального приближения $x^{(0)}$.

Условия сходимости метода определяются следующей теоремой [2]:

Теорема 2.2. Пусть на отрезке $[a,b]$ функция $f(x)$ имеет первую и вторую производные постоянного знака и пусть $f(a)f(b) < 0$.

Тогда если точка $x^{(0)}$ выбрана на $[a,b]$ так, что

$$f(x^{(0)})f''(x^{(0)}) > 0, \quad (2.3)$$

то начатая с нее последовательность $x^{(k)}$ ($k = 0, 1, 2, \dots$), определяемая методом Ньютона (2.2), монотонно сходится к корню $x^{(*)} \in (a, b)$ уравнения (2.1).

В качестве условия окончания итераций в практических вычислениях часто используется правило $|x^{(k+1)} - x^{(k)}| < \varepsilon \Rightarrow x^{(*)} \approx x^{(k+1)}$.

Метод простой итерации. При использовании метода простой итерации уравнение (2.1) заменяется эквивалентным уравнением с выделенным линейным членом

$$x = \varphi(x) \quad (2.5)$$

Решение ищется путем построения последовательности

$$x^{(k+1)} = \varphi(x^{(k)}) \quad k = 0,1,2,\dots \quad (2.6)$$

начиная с некоторого заданного значения $x^{(0)}$. Если $\varphi(x)$ - непрерывная функция, а $x^{(k)} \quad (k = 0,1,2,\dots)$ - сходящаяся последовательность, то значение $x^{(*)} = \lim_{k \rightarrow \infty} x^{(k)}$ является решением уравнения (2.5).

Условия сходимости метода и оценка его погрешности определяются теоремой [2]:

Теорема 2.3. Пусть функция $\varphi(x)$ определена и дифференцируема на отрезке $[a,b]$. Тогда если выполняются условия:

- 1) $\varphi(x) \in [a,b] \quad \forall x \in [a,b],$
- 2) $\exists q : |\varphi'(x)| \leq q < 1 \quad \forall x \in (a,b),$

то уравнение (2.5) имеет и притом единственный на $[a,b]$ корень $x^{(*)}$;

к этому корню сходится определяемая методом простой итерации последовательность $x^{(k)} \quad (k = 0,1,2,\dots)$, начинающаяся с любого $x^{(0)} \in [a,b]$.

При этом справедливы оценки погрешности ($\forall k \in N$):

$$\begin{aligned} |x^{(*)} - x^{(k+1)}| &\leq \frac{q}{1-q} |x^{(k+1)} - x^{(k)}| \\ |x^{(*)} - x^{(k+1)}| &\leq \frac{q^{k+1}}{1-q} |x^{(1)} - x^{(0)}|. \end{aligned} \quad (2.7)$$

Код программы:

```

1 import math
2 | You, 4 недели назад • new labs added
3 EPSILON = 1e-6
4 MAX_ITERATIONS = 100
5
6 def f(x):
7 |     return math.log10(x + 1) - x + 0.5
8
9 def df(x):
10 |    return 1 / ((x + 1) * math.log(10)) - 1
11
12 def d2f(x):
13 |    return -1 / ((x + 1) ** 2 * math.log(10))
14

```

```

15 def phi(x):
16     return math.log10(x + 1) + 0.5
17
18 def phi_derivative_max(a, b):
19     #  $\phi'(x) = 1 / ((x + 1) * \ln(10))$  - максимум в а
20     return 1 / ((a + 1) * math.log(10))
21
22 def find_root_interval(f, start, end, step):
23     x_prev = start
24     f_prev = f(x_prev)
25     x = start + step
26     while x <= end:
27         f_curr = f(x)
28         if f_prev * f_curr <= 0:
29             return [x_prev, x]
30         x_prev = x
31         f_prev = f_curr
32         x += step
33     return None
34
35 def newton_method(f, df, d2f, x0, a, b, eps):
36     print("\nМетод Ньютона:")
37
38     # Условие сходимости  $f(x_0)*f''(x_0) > 0$ 
39     check_value = f(x0) * d2f(x0)
40     if check_value <= 0:
41         print(f"Предупреждение: f(x0)*f''(x0) = {check_value:.2f} <= 0 (условие сходимости не выполняется)")
42
43     x = x0
44     print(f"iter {0:2d}: x = {x:.8f}")
45
46     for iter in range(1, MAX_ITERATIONS + 1):
47         fx = f(x)
48         dfx = df(x)
49
50         if abs(dfx) < 1e-12:
51             print("Ошибка: производная слишком близка к нулю!")
52             return
53
54         x_next = x - fx / dfx
55
56         if x_next < a or x_next > b:
57             print("Ошибка: выход за границы интервала!")
58             return
59
60         delta = abs(x_next - x)
61         print(f"iter {iter:2d}: x = {x_next:.8f} | Δ = {delta:.2e}")
62
63         if delta < eps:
64             print("Достигнута заданная точность")
65             return
66
67         x = x_next
68
69     print("Достигнуто максимальное число итераций!")
70
71 def iteration_method(phi, x0, a, b, eps):
72     print("\nМетод простой итерации:")
73
74     # Условие сходимости существует такой что  $abs(phi'(x)) \leq q < 1$  для всех x на интервале

```

```

75     q = phi_derivative_max(a, b)
76     if q >= 1:
77         print(f"Ошибка: условие сходимости не выполняется (q = {q:.2f})")
78         return
79     print(f"Параметр сходимости q = {q:.6f}")
80
81     x = x0
82     print(f"iter {0:2d}: x = {x:.8f}")
83
84     for iter in range(1, MAX_ITERATIONS + 1):
85         x_next = phi(x)
86
87         if x_next < a or x_next > b:
88             print("Ошибка: выход за границы интервала!")
89             return
90
91         # Условие окончания abs(x* - x(k+1)) <= (q/(1-q)) * abs(x(k+1)-x(k))
92         delta = abs(x_next - x)
93         error_estimate = q * delta / (1 - q)
94         print(f"iter {iter:2d}: x = {x_next:.8f} | Δ = {delta:.2e} | оценка погрешности = {error_estimate:.2e}")
95
96         if error_estimate < eps:
97             print("Достигнута заданная точность")
98             return
99
100        x = x_next
101
102    print("Достигнуто максимальное число итераций!")

```

Результат:

```

Установлена точность ε = 1.0e-06
Найден интервал с корнем: [0.7400, 0.7500]
Начальное приближение: x0 = 0.7450000

```

Метод Ньютона:

```

iter 0: x = 0.74500000
iter 1: x = 0.74073362 | Δ = 4.27e-03
iter 2: x = 0.74073188 | Δ = 1.73e-06
iter 3: x = 0.74073188 | Δ = 2.87e-13

```

Достигнута заданная точность

Метод простой итерации:

Параметр сходимости q = 0.249595

```

iter 0: x = 0.74500000
iter 1: x = 0.74179543 | Δ = 3.20e-03 | оценка погрешности = 1.07e-03
iter 2: x = 0.74099715 | Δ = 7.98e-04 | оценка погрешности = 2.66e-04
iter 3: x = 0.74079806 | Δ = 1.99e-04 | оценка погрешности = 6.62e-05
iter 4: x = 0.74074839 | Δ = 4.97e-05 | оценка погрешности = 1.65e-05
iter 5: x = 0.74073600 | Δ = 1.24e-05 | оценка погрешности = 4.12e-06
iter 6: x = 0.74073291 | Δ = 3.09e-06 | оценка погрешности = 1.03e-06
iter 7: x = 0.74073214 | Δ = 7.71e-07 | оценка погрешности = 2.57e-07

```

Достигнута заданная точность

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

Проанализировать зависимость погрешности вычислений от количества итераций.

Условие:

13	2	$\begin{cases} x_1^2/a^2 + x_2^2/(a/2)^2 - 1 = 0, \\ ax_2 - e^{x_1} - x_1 = 0. \end{cases}$
14	3	
15	4	

Метод Ньютона. Если определено начальное приближение $\mathbf{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)})^T$, итерационный процесс нахождения решения системы (2.11) методом Ньютона можно представить в виде

$$\begin{cases} x_1^{(k+1)} = x_1^{(k)} + \Delta x_1^{(k)} \\ x_2^{(k+1)} = x_2^{(k)} + \Delta x_2^{(k)} \\ \dots \\ x_n^{(k+1)} = x_n^{(k)} + \Delta x_n^{(k)} \end{cases} \quad k = 0, 1, 2, \dots \quad (2.13)$$

где значения приращений $\Delta x_1^{(k)}, \Delta x_2^{(k)}, \dots, \Delta x_n^{(k)}$ определяются из решения системы линейных алгебраических уравнений, все коэффициенты которой выражаются через известное предыдущее приближение $\mathbf{x}^{(k)} = (x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)})$

$$\begin{cases} f_1(\mathbf{x}^{(k)}) + \frac{\partial f_1(\mathbf{x}^{(k)})}{\partial x_1} \Delta x_1^{(k)} + \frac{\partial f_1(\mathbf{x}^{(k)})}{\partial x_2} \Delta x_2^{(k)} + \dots + \frac{\partial f_1(\mathbf{x}^{(k)})}{\partial x_n} \Delta x_n^{(k)} = 0 \\ f_2(\mathbf{x}^{(k)}) + \frac{\partial f_2(\mathbf{x}^{(k)})}{\partial x_1} \Delta x_1^{(k)} + \frac{\partial f_2(\mathbf{x}^{(k)})}{\partial x_2} \Delta x_2^{(k)} + \dots + \frac{\partial f_2(\mathbf{x}^{(k)})}{\partial x_n} \Delta x_n^{(k)} = 0 \\ \dots \\ f_n(\mathbf{x}^{(k)}) + \frac{\partial f_n(\mathbf{x}^{(k)})}{\partial x_1} \Delta x_1^{(k)} + \frac{\partial f_n(\mathbf{x}^{(k)})}{\partial x_2} \Delta x_2^{(k)} + \dots + \frac{\partial f_n(\mathbf{x}^{(k)})}{\partial x_n} \Delta x_n^{(k)} = 0 \end{cases} \quad (2.14)$$

В векторно-матричной форме расчетные формулы имеют вид

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta \mathbf{x}^{(k)} \quad k = 0, 1, 2, \dots \quad (2.15)$$

где вектор приращений $\Delta \mathbf{x}^{(k)} = \begin{pmatrix} \Delta x_1^{(k)} \\ \Delta x_2^{(k)} \\ \dots \\ \Delta x_n^{(k)} \end{pmatrix}$ находится из решения уравнения

$$\mathbf{f}(\mathbf{x}^{(k)}) + \mathbf{J}(\mathbf{x}^{(k)}) \Delta \mathbf{x}^{(k)} = \mathbf{0} \quad (2.16)$$

Здесь $\mathbf{J}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \frac{\partial f_1(\mathbf{x})}{\partial x_2} & \dots & \frac{\partial f_1(\mathbf{x})}{\partial x_n} \\ \frac{\partial f_2(\mathbf{x})}{\partial x_1} & \frac{\partial f_2(\mathbf{x})}{\partial x_2} & \dots & \frac{\partial f_2(\mathbf{x})}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n(\mathbf{x})}{\partial x_1} & \frac{\partial f_n(\mathbf{x})}{\partial x_2} & \dots & \frac{\partial f_n(\mathbf{x})}{\partial x_n} \end{bmatrix}$ - матрица Якоби первых производных вектор-функции $\mathbf{f}(\mathbf{x})$.

Выражая из (2.16) вектор приращений $\Delta \mathbf{x}^{(k)}$ и подставляя его в (2.15), итерационный процесс нахождения решения можно записать в виде

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{J}^{-1}(\mathbf{x}^{(k)}) \mathbf{f}(\mathbf{x}^{(k)}) \quad k = 0, 1, 2, \dots \quad (2.17)$$

где $\mathbf{J}^{-1}(\mathbf{x})$ - матрица, обратная матрице Якоби. Формула (2.17) есть обобщение формулы (2.2) на случай систем нелинейных уравнений.

При реализации алгоритма метода Ньютона в большинстве случаев предпочтительным является не вычисление обратной матрицы $\mathbf{J}^{-1}(\mathbf{x}^{(k)})$, а нахождение из системы (2.14) значений приращений $\Delta x_1^{(k)}, \Delta x_2^{(k)}, \dots, \Delta x_n^{(k)}$ и вычисление нового приближения по (2.13). Для решения таких линейных систем можно привлекать самые разные методы, как прямые, так и итерационные (см. раздел 1.1), с учетом размерности n решаемой задачи и специфики матриц Якоби $\mathbf{J}(\mathbf{x})$ (например, симметрии, разреженности и т.п.).

Использование метода Ньютона предполагает дифференцируемость функций $f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_n(\mathbf{x})$ и невырожденность матрицы Якоби ($\det \mathbf{J}(\mathbf{x}^{(k)}) \neq 0$). В случае, если начальное приближение выбрано в достаточно малой окрестности искомого корня, итерации сходятся к точному решению, причем сходимость квадратичная.

В практических вычислениях в качестве условия окончания итераций обычно используется критерий [2,5]

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| \leq \varepsilon, \quad (2.18)$$

где ε - заданная точность.

Код:

```

1  import math
2  import numpy as np
3
4  EPSILON = 1e-6
5  MAX_ITER = 100
6
7  # Уравнения системы
8  def f1(x1, x2):
9      return 2 * x1 ** 2 - x2 + x2 ** 2 - 2
10
11 def f2(x1, x2):
12     return x1 - math.sqrt(x2 + 2) + 1
13
14 # Фи-функции для метода простой итерации
15 def phi1(x1, x2):
16     return math.sqrt(x2 + 2) - 1
17
18 def phi2(x1, x2):
19     return 2 * x1 ** 2 + x2 ** 2 - 2
20
21 # Производные фи-функций
22 def dphi1_dx2(x2):
23     return 1 / (2 * math.sqrt(x2 + 2))
24
25 def dphi2_dx1(x1):
26     return 4 * x1
27
28 def dphi2_dx2(x2):
29     return 2 * x2
30
31 # Поиск приближённого пересечения - Ищем такую точку, чтобы значения обеих функций в этой точке были близки к 0
32 def find_root_interval(from_, to, step):
33     minD = 0.1
34     best = None
35
36     x_range = np.arange(from_, to + step, step)
37     for x1 in x_range:
38         for x2 in x_range:
39             d = math.sqrt(f1(x1, x2)**2 + f2(x1, x2)**2)
40             if d < minD:
41                 minD = d
42                 best = (x1, x2)
43
44     if minD < 0.1:
45         print(f"Найдено приближённое пересечение: x1 = {best[0]:.4f}, x2 = {best[1]:.4f}, D = {minD:.6f}")
46         return best
47     else:
48         print(f"Пересечения не найдено в диапазоне [{from_}; {to}], minD = {minD:.6f}")
49         return None
50
51 # Проверка условия сходимости
52 def check_convergence(x1, x2):
53     J = [
54         [0, dphi1_dx2(x2)],
55         [dphi2_dx1(x1), dphi2_dx2(x2)]
56     ]
57
58     # Условие сходимости: вектор функция phi непрерывна вместе со своей производной в области G и max||phi'(x)|| <= q < 1.
59
60     max_row_sum = max(sum(abs(j) for j in row) for row in J)
61     print(f"Проверка сходимости: q = {max_row_sum:.4f} {'< 1 - сходимость есть' if max_row_sum < 1 else '>= 1 - сходимости НЕТ'}")
62     return max_row_sum < 1

```

```

63
64     # Метод простой итерации
65     def simple_iteration(x1_0, x2_0):
66         x1, x2 = x1_0, x2_0
67         iter_count = 0
68
69         if not check_convergence(x1, x2):
70             print("Метод простой итерации может не сойтись.")
71
72         while iter_count < MAX_ITER:
73             x1_new = phi1(x1, x2)
74             x2_new = phi2(x1, x2)
75
76             error = max(abs(x1_new - x1), abs(x2_new - x2))
77             if error < EPSILON:
78                 break
79
80             x1, x2 = x1_new, x2_new
81             iter_count += 1
82
83         print(f"Простая итерация: x1 = {x1:.6f}, x2 = {x2:.6f} за {iter_count} итераций")
84
85     # Метод Ньютона
86     def newton(x1_0, x2_0):
87         x1, x2 = x1_0, x2_0
88         iter_count = 0
89
90         while iter_count < MAX_ITER:
91             f1v = f1(x1, x2)
92             f2v = f2(x1, x2)
93
94             df1_dx1 = 4 * x1
95             df1_dx2 = -1 + 2 * x2
96             df2_dx1 = 1
97             df2_dx2 = -1 / (2 * math.sqrt(x2 + 2))
98
99             detJ = df1_dx1 * df2_dx2 - df1_dx2 * df2_dx1
100            detA1 = f1v * df2_dx2 - df1_dx2 * f2v
101            detA2 = df1_dx1 * f2v - f1v * df2_dx1
102
103            if abs(detJ) < 1e-12:
104                print("Матрица Якоби вырождена, метод Ньютона остановлен.")
105                return
106
107            # xn(k+1) = xn(k) - detAn(k)/detJ(k), для каждого xn
108            x1_new = x1 - detA1 / detJ
109            x2_new = x2 - detA2 / detJ
110
111            # Условие окончания: ||x(k-1)=x(k)|| = max по i |xi(k+1) - xi(k)|.
112            error = max(abs(x1_new - x1), abs(x2_new - x2))
113            if error < EPSILON:
114                break
115
116            x1, x2 = x1_new, x2_new
117            iter_count += 1
118
119        print(f"Ньютон: x1 = {x1:.6f}, x2 = {x2:.6f} за {iter_count} итераций")
120
121    # Главная функция
122    def main():
123        interval = find_root_interval(0, 2, 0.1)
124        if interval is None:
125            print("Не удалось найти интервал, содержащий корень.")
126            return
127
128        x1_0, x2_0 = interval
129        print(f"Найдено начальное приближение: x1 = {x1_0:.4f}, x2 = {x2_0:.4f}")
130
131        simple_iteration(x1_0, x2_0)
132        newton(x1_0, x2_0)

```

Результат:

```
Найдено приближённое пересечение: x1 = 0.8000, x2 = 1.5000, D = 0.076920
Найдено начальное приближение: x1 = 0.8000, x2 = 1.5000
Проверка сходимости: q = 6.2000 >= 1 - сходимости НЕТ
Метод простой итерации может не сойтись.
c:\Users\Евгений\Desktop\Evgeny\Study\NumMethods\solutions\lab2\lab2.2.py:19: RuntimeWarning: overflow encountered in scalar power
    return 2 * x1 ** 2 + x2 ** 2 - 2
c:\Users\Евгений\Desktop\Evgeny\Study\NumMethods\solutions\lab2\lab2.2.py:76: RuntimeWarning: invalid value encountered in scalar subtract
    error = max(abs(x1_new - x1), abs(x2_new - x2))
Простая итерация: x1 = inf, x2 = inf за 100 итераций
Ньютона: x1 = 0.845466, x2 = 1.405745 за 3 итераций
```


Лабораторная работа №3

3.1. Используя таблицу значений Y_i функции $y = f(x)$, вычисленных в точках X_i , $i = 0, \dots, 3$ построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки $\{X_i, Y_i\}$. Вычислить значение погрешности интерполяции в точке X^* .

3.1. ИНТЕРПОЛЯЦИЯ

Пусть на отрезке $[a, b]$ задано множество несовпадающих точек x_i (интерполяционных узлов), в которых известны значения функции $f_i = f(x_i)$, $i = 0, \dots, n$.

Приближающая функция $\varphi(x, a)$ такая, что выполняются равенства

$$\varphi(x_i, a_0, \dots, a_n) = f(x_i) = f_i, \quad i = 0, \dots, n. \quad (3.1)$$

называется интерполяционной.

Наиболее часто в качестве приближающей функции используют многочлены степени n :

$$P_n(x) = \sum_{i=0}^n a_i x^i \quad (3.2)$$

Введем понятие разделенной разности. Разделенные разности нулевого порядка совпадают со значениями функции в узлах. Разделенные разности первого порядка обозначаются $f(x_i, x_j)$ и определяются через разделенные разности нулевого порядка:

$$f(x_i, x_j) = \frac{f_i - f_j}{x_i - x_j},$$

разделенные разности второго порядка определяются через разделенные разности первого порядка:

$$f(x_i, x_j, x_k) = \frac{f(x_i, x_j) - f(x_j, x_k)}{x_i - x_k}.$$

Разделенная разность порядка $n - k + 2$ определяется соотношениями

$$f(x_i, x_j, x_k, \dots, x_{n-1}, x_n) = \frac{f(x_i, x_j, x_k, \dots, x_{n-1}) - f(x_j, x_k, \dots, x_n)}{x_i - x_n}. \quad (3.7)$$

Таким образом, для $(n+1)$ -й точки могут быть построены разделенные разности до n -го порядка; разделенные разности более высоких порядков равны нулю.

Пусть известны значения аппроксимируемой функции $f(x)$ в точках x_0, x_1, \dots, x_n .

Интерполяционный многочлен, значения которого в узлах интерполяции совпадают со значениями функции $f(x)$ может быть записан в виде:

$$\begin{aligned} P_n(x) = & f(x_0) + (x - x_0)f(x_1, x_0) + (x - x_0)(x - x_1)f(x_0, x_1, x_2) + \dots + \\ & + (x - x_0)(x - x_1)\dots(x - x_n)f(x_0, x_1, \dots, x_n). \end{aligned} \quad (3.8)$$

Запись многочлена в формуле (3.8) есть так называемый интерполяционный многочлен Ньютона. Если функция $f(x)$ не есть многочлен n -й степени, то формула (3.8) для $P_n(x)$ приближает функцию $f(x)$ с некоторой погрешностью. Отметим, что при

Условие:

$$15. \quad y = \operatorname{ctg}(x) + x, \text{a) } X_i = \frac{\pi}{8}, \frac{2\pi}{8}, \frac{3\pi}{8}, \frac{4\pi}{8}; \text{б) } X_i = \frac{\pi}{8}, \frac{\pi}{3}, \frac{3\pi}{8}, \frac{\pi}{2}; \quad X^* = \frac{3\pi}{16}.$$

Код программы:

```

1   import numpy as np
2
3   # Функция f(x) = cot(x) + x
4   def f(x):
5       return 1 / np.tan(x) + x
6
7   # Интерполяционный многочлен Лагранжа
8   def lagrange(xi, yi, x):
9       n = len(xi)
10
11      # omega(x) = произведение (x - xi)
12      omega = 1.0
13      for i in range(n):
14          omega *= (x - xi[i])
15
16      result = 0.0
17      for i in range(n):
18          omega_prime = 1.0
19          for j in range(n):
20              if j != i:
21                  omega_prime *= (xi[i] - xi[j])
22          result += (yi[i] / omega_prime) * (omega / (x - xi[i]))
23
24      return result
25
26  # Таблица разделённых разностей (для метода Ньютона)
27  def divided_differences(xi, yi):
28      n = len(xi)
29      table = np.zeros((n, n))
30      table[:, 0] = yi
31
32      for j in range(1, n):
33          for i in range(n - j):
34              table[i][j] = (table[i][j - 1] - table[i + 1][j - 1]) / (xi[i] - xi[i + j])
35
36      return table
37
38  # Интерполяционный многочлен Ньютона
39  def newton(xi, dd_table, x):
40      result = dd_table[0][0]
41      product = 1.0
42      for i in range(1, len(xi)):
43          product *= (x - xi[i - 1])
44          result += dd_table[0][i] * product
45
46      return result
47
48  # Абсолютная погрешность
49  def absolute_error(real_value, approx_value):
50      return abs(real_value - approx_value)
51
52  # Основной метод решения задачи
53  def solve(xi, x_star, label):
54      print(f"\nРешение для набора {label}:")
55      yi = [f(x) for x in xi]
56
57      print("Точки (X, Y):")
58      for i, (x_val, y_val) in enumerate(zip(xi, yi)):
59          print(f"x[{i}] = {x_val:.4f}, y[{i}] = {y_val:.6f}")
60
61      lagrange_value = lagrange(xi, yi, x_star)
62      dd_table = divided_differences(xi, yi)
63      newton_value = newton(xi, dd_table, x_star)
64      real_value = f(x_star)
65
66      print(f"\nВычисления в точке X* = {x_star}:")
67      print(f"Истинное значение: f({x_star:.2f}) = {real_value:.6f}")
68      print(f"Лагранжев интерполянт: L({x_star:.2f}) = {lagrange_value:.6f}")
69      print(f"Ньютонов интерполянт: N({x_star:.2f}) = {newton_value:.6f}")
70
71      print("\nАбсолютные погрешности:")
72      print(f"|f(x*) - L(x*)| = {absolute_error(real_value, lagrange_value):.6f}")
73      print(f"|f(x*) - N(x*)| = {absolute_error(real_value, newton_value):.6f}")

```

Результат:

Решение для набора а) $\{\pi/8, 2\pi/8, 3\pi/8, 4\pi/8\}$:

Точки (X, Y):

$x[0] = 0.3927, y[0] = 2.806913$
 $x[1] = 0.7854, y[1] = 1.785398$
 $x[2] = 1.1781, y[2] = 1.592311$
 $x[3] = 1.5708, y[3] = 1.570796$

Вычисления в точке $X^* = 0.5890486225480862$:

Истинное значение: $f(0.59) = 2.085654$

Лагранжев интерполянт: $L(0.59) = 2.151549$

Ньютона интерполянт: $N(0.59) = 2.151549$

Абсолютные погрешности:

$|f(x^*) - L(x^*)| = 0.065894$
 $|f(x^*) - N(x^*)| = 0.065894$

Решение для набора б) $\{\pi/8, \pi/3, 3\pi/8, \pi/2\}$:

Точки (X, Y):

$x[0] = 0.3927, y[0] = 2.806913$
 $x[1] = 1.0472, y[1] = 1.624548$
 $x[2] = 1.1781, y[2] = 1.592311$
 $x[3] = 1.5708, y[3] = 1.570796$

Вычисления в точке $X^* = 0.5890486225480862$:

Истинное значение: $f(0.59) = 2.085654$

Лагранжев интерполянт: $L(0.59) = 2.200593$

Ньютона интерполянт: $N(0.59) = 2.200593$

Абсолютные погрешности:

$|f(x^*) - L(x^*)| = 0.114938$
 $|f(x^*) - N(x^*)| = 0.114938$

3.2. Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну $x = x_0$ и $x = x_4$. Вычислить при

значение функции в $x = X^*$.

точке

Использование одной интерполяционной формулы на большом числе узлов нецелесообразно. Интерполяционный многочлен может проявить свои колебательные свойства, его значения между узлами могут сильно отличаться от значений интерполируемой функции. Одна из возможностей преодоления этого недостатка заключается в применении *сплайн-интерполяции*. Суть сплайн-интерполяции заключается в определении интерполирующей функции по формулам одного типа для различных непересекающихся промежутков и в стыковке значений функции и её производных на их границах.

Для построения кубического сплайна необходимо построить n многочленов третьей степени, т.е. определить $4n$ неизвестных a_i, b_i, c_i, d_i . Эти коэффициенты ищутся из условий в узлах сетки.

$$\begin{aligned} S(x_{i-1}) &= a_i = a_{i-1} + b_{i-1}(x_{i-1} - x_{i-2}) + c_{i-1}(x_{i-1} - x_{i-2})^2 + d_{i-1}(x_{i-1} - x_{i-2})^3 = f_{i-1} \\ S'(x_{i-1}) &= b_i = b_{i-1} + 2c_{i-1}(x_{i-1} - x_{i-2}) + 3d_{i-1}(x_{i-1} - x_{i-2})^2, \\ S''(x_{i-1}) &= 2c_i = 2c_{i-1} + 6d_{i-1}(x_{i-1} - x_{i-2}), \quad i = 2, 3, \dots, n \\ S(x_0) &= a_1 = f_0 \\ S''(x_0) &= c_1 = 0 \\ S(x_n) &= a_n + b_n(x_n - x_{n-1}) + c_n(x_n - x_{n-1})^2 + d_n(x_n - x_{n-1})^3 = f_n \\ S''(x_n) &= 2c_n + 6d_n(x_n - x_{n-1}) = 0 \end{aligned}, \quad (3.12)$$

Условие:

15. $X^* = 2.66666667$

i	0	1	2	3	4
x_i	1.0	1.9	2.8	3.7	4.6
f_i	2.8069	1.8279	1.6091	1.5713	1.5663

Код программы:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def solve_tridiagonal(a, b, c, d):
5     """Решает трёхдиагональную систему линейных уравнений методом прогонки.
6         Шаги:
7             - Прямой ход – преобразуем систему к верхнетреугольному виду;
8             - Обратный ход – находим решения x_i по формулам.
9     """
10    n = len(b)
11    P = np.zeros(n)
12    Q = np.zeros(n)
13    x = np.zeros(n)
14
15    P[0] = -c[0] / b[0]
16    Q[0] = d[0] / b[0]
```

```

18     # Прямой ход
19     for i in range(1, n):
20         denom = b[i] + a[i] * P[i - 1]
21         P[i] = 0 if i == n - 1 else -c[i] / denom
22         Q[i] = (d[i] - a[i] * Q[i - 1]) / denom
23
24     # Обратный ход
25     x[-1] = Q[-1]
26     for i in range(n - 2, -1, -1):
27         x[i] = P[i] * x[i + 1] + Q[i]
28
29     return x
30
31 def cubic_spline_interpolation(x, f, x_star):
32     """
33     Строит кубический сплайн по точкам и вычисляет значение сплайна в заданной точке X*.
34     """
35     n = len(x) - 1
36     h = np.zeros(n + 1)
37     for i in range(1, n + 1):
38         h[i] = x[i] - x[i - 1]
39
40     # Формирование СЛАУ для коэффициентов с (вторая производная сплайна)
41     A = np.zeros(n - 1)
42     B = np.zeros(n - 1)
43     C = np.zeros(n - 1)
44     D = np.zeros(n - 1)
45
46     for i in range(2, n + 1):
47         idx = i - 2
48         A[idx] = 0.0 if i == 2 else h[i - 1]
49         B[idx] = 2 * (h[i - 1] + h[i])
50         C[idx] = 0.0 if i == n else h[i]
51         D[idx] = 3 * ((f[i] - f[i - 1]) / h[i] - (f[i - 1] - f[i - 2]) / h[i - 1])
52
53     solutionC = solve_tridiagonal(A, B, C, D)
54
55     c = np.zeros(n + 2) # индексируем с 1 до n + 1
56     c[1] = 0.0
57     c[n + 1] = 0.0
58     for i in range(2, n + 1):
59         c[i] = solutionC[i - 2]
60
61     a = np.zeros(n + 1)
62     b = np.zeros(n + 1)
63     d = np.zeros(n + 1)
64
65     # Вычисление всех коэффициентов сплайна
66     for i in range(1, n + 1):
67         a[i] = f[i - 1]
68         d[i] = (c[i + 1] - c[i]) / (3 * h[i])
69         b[i] = (f[i] - f[i - 1]) / h[i] - (h[i] / 3) * (2 * c[i] + c[i + 1])
70
71     # Определение нужного интервала для x*
72     interval = 1
73     for i in range(1, n + 1):
74         if x_star >= x[i - 1] and x_star <= x[i]:
75             interval = i
76             break
77

```

```

78     # Вычисление значения сплайна в x*
79     dx = x_star - x[interval - 1]
80     result = a[interval] + b[interval] * dx + c[interval] * dx**2 + d[interval] * dx**3
81
82     print(f"Значение функции в точке X* = {x_star:.3f}: {result:.6f}")
83     for i in range(1, n + 1):
84         print(f"[{x[i-1]:.1f}, {x[i]:.1f}]: a={a[i]:.6f}, b={b[i]:.6f}, c={c[i]:.6f}, d={d[i]:.6f}")
85
86     # Построение графика сплайна
87     spline_x = []
88     spline_y = []
89     for i in range(1, n + 1):
90         xi = x[i - 1]
91         xi_next = x[i]
92         dx_range = np.linspace(0, xi_next - xi, 100)
93         yi = a[i] + b[i] * dx_range + c[i] * dx_range**2 + d[i] * dx_range**3
94         spline_x.extend(xi + dx_range)
95         spline_y.extend(yi)
96
97     plt.figure(figsize=(8, 5))
98     plt.plot(spline_x, spline_y, label='Кубический сплайн', color='blue')
99     plt.plot(x, f, 'o', label='Узлы интерполяции', color='red')
100    plt.plot(x_star, result, 's', label=f'S(x*) = {result:.3f}', color='green')
101    plt.title('Интерполяция кубическим сплайном')
102    plt.xlabel('x')
103    plt.ylabel('f(x)')
104    plt.grid(True)
105    plt.legend()
106    plt.show()      You, 2 недели назад • added new labs ...
107
108    return result

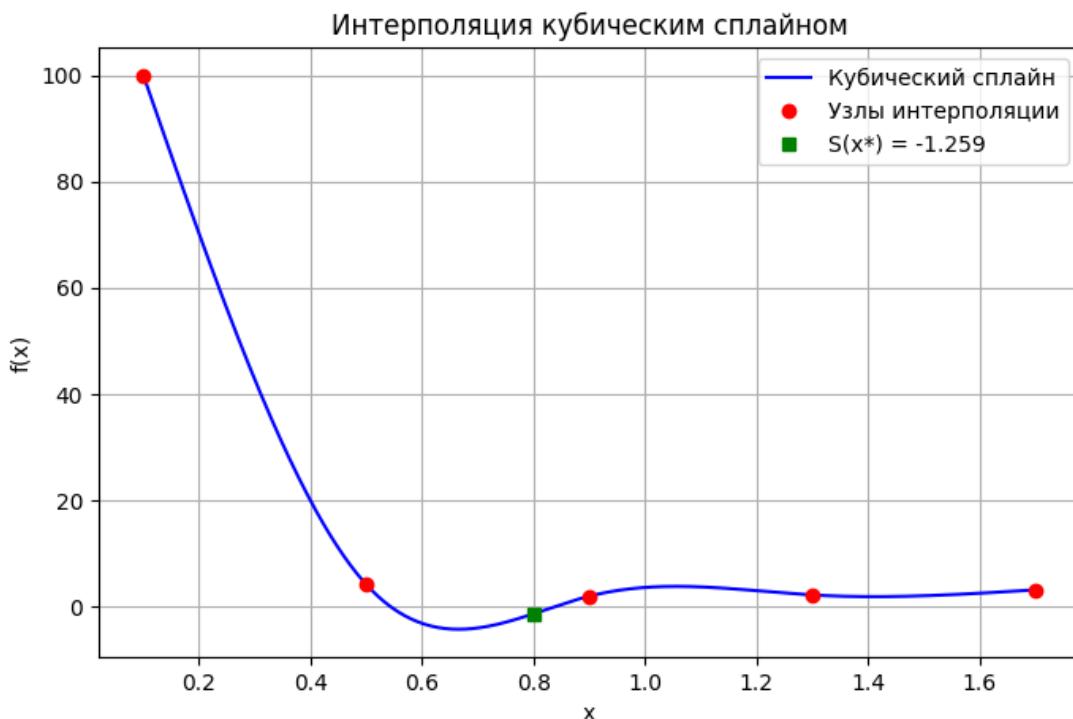
```

Результат:

```

Значение функции в точке X* = 0.800: -1.258917
[0.1, 0.5]: a=100.010000, b=-301.644027, c=0.000000, d=389.025167
[0.5, 0.9]: a=4.250000, b=-114.911946, c=466.830201, d=-483.335212
[0.9, 1.3]: a=2.044600, b=26.551312, c=-113.172054, d=120.689118
[1.3, 1.7]: a=2.281700, b=-6.055554, c=31.654888, d=-26.379074

```



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

Пусть задана таблично в узлах x_j функция $y_j = f(x_j)$, $j = 0, 1, \dots, N$. При этом значения функции y_j определены с некоторой погрешностью, также из физических соображений известен вид функции, которой должны приближенно удовлетворять табличные точки, например: многочлен степени n , у которого неизвестны коэффициенты a_i , $F_n(x) = \sum_{i=0}^n a_i x^i$. Неизвестные коэффициенты будем находить из условия минимума

квадратичного отклонения многочлена от таблично заданной функции.

$$\Phi = \sum_{j=0}^N [F_n(x_j) - y_j]^2. \quad (3.15)$$

Условие:

15.

i	0	1	2	3	4	5
x_i	1.0	1.9	2.8	3.7	4.6	5.5
y_i	3.4142	2.9818	3.3095	3.8184	4.3599	4.8318

Код программы:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # === 1. Входные данные ===
5 points = [
6     (1.0, 3.4142),
7     (1.9, 2.9818),
8     (2.8, 3.3095),
9     (3.7, 3.8184),
10    (4.6, 4.3599),
11    (5.5, 4.8318)
12 ]
13 | You, 2 недели назад • added new labs
14 x_vals = np.array([p[0] for p in points])
15 y_vals = np.array([p[1] for p in points])
16 n = len(points)
17
18 # === 2. Решение нормальных систем МНК ===
19

```

```

20 def solve_least_squares(x, y, degree):
21     """Находит коэффициенты приближающего многочлена степени degree методом наименьших квадратов (МНК)."""
22     # Составляем матрицу Вандермонда для подстановки соответствующих  $x_i^j$  в полином
23     A = np.vander(x, N=degree+1, increasing=True) # Строит матрицу, где каждая строка – это степенные комби-
24     # Решаем нормальную систему  $A \cdot T \cdot A = A \cdot T \cdot y$ 
25     ATA = A.T @ A
26     ATy = A.T @ y
27     coeffs = np.linalg.solve(ATA, ATy)
28     return coeffs # Коэффициенты многочлена, минимизирующие ошибку
29
30 # === 3. Вычисление ошибки ===
31
32 def compute_error(x, y, coeffs):
33     """Считает сумму квадратов ошибок."""
34     y_pred = np.polyval(coeffs[:-1], x) # Вычисляет значение многочлена на заданных x - вместо наивной под-
35     return np.sum((y - y_pred)**2)
36
37 # === 4. Построение графика ===
38
39 def plot_results(x, y, coeffs_list, labels):
40     """Строит график исходных точек и аппроксимаций."""
41     plt.figure(figsize=(10, 6))
42
43     # Исходные точки
44     plt.scatter(x, y, color='black', label='Исходные данные')
45
46     x_plot = np.linspace(min(x) - 0.2, max(x) + 0.2, 500)
47
48     colors = ['blue', 'red']
49     for coeffs, label, color in zip(coeffs_list, labels, colors):
50         y_plot = np.polyval(coeffs[:-1], x_plot)
51         plt.plot(x_plot, y_plot, label=label, color=color)
52
53     plt.title("Аппроксимация методом наименьших квадратов")
54     plt.xlabel("x")
55     plt.ylabel("y")
56     plt.grid(True)
57     plt.legend()
58     plt.tight_layout()
59     plt.show()
60
61 # === 5. Основная логика ===
62
63 # Линейная аппроксимация (1-я степень)
64 coeffs_linear = solve_least_squares(x_vals, y_vals, degree=1)
65 error_linear = compute_error(x_vals, y_vals, coeffs_linear)
66
67 # Квадратичная аппроксимация (2-я степень)
68 coeffs_quadratic = solve_least_squares(x_vals, y_vals, degree=2)
69 error_quadratic = compute_error(x_vals, y_vals, coeffs_quadratic)
70
71 # === 6. Вывод результатов ===
72
73 def print_polynomial(coeffs):
74     terms = [f"{coeff:.6f}·x^{i}" for i, coeff in enumerate(coeffs)]
75     print(" + ".join(terms))
76
77 print("Многочлен 1-й степени:")
78 print_polynomial(coeffs_linear)
79 print(f"Сумма квадратов ошибок: {error_linear:.6f}\n")
80
81 print("Многочлен 2-й степени:")
82 print_polynomial(coeffs_quadratic)
83 print(f"Сумма квадратов ошибок: {error_quadratic:.6f}\n")
84
85 # === 7. График ===
86 plot_results(x_vals, y_vals, [coeffs_linear, coeffs_quadratic], ["1-я степень", "2-я степень"])

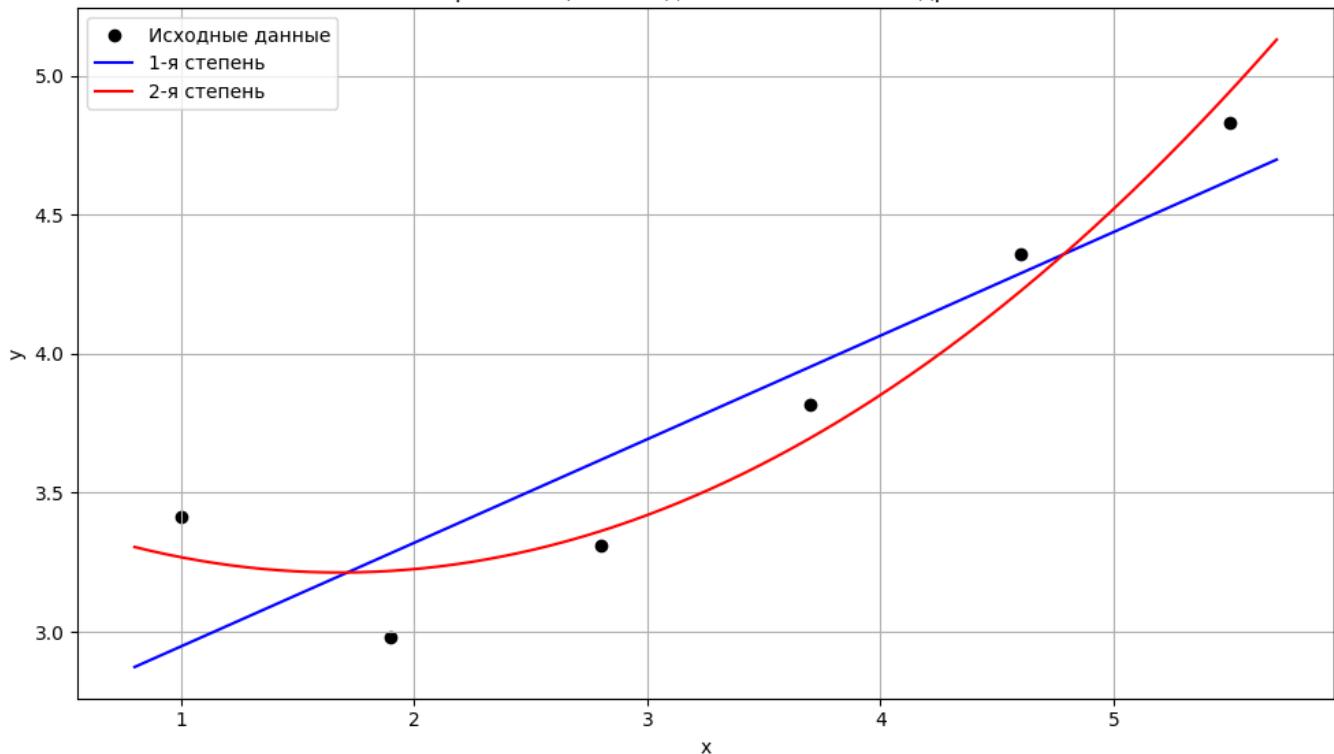
```

Результат:

Многочлен 1-й степени:
 $2.575571 \cdot x^0 + 0.372419 \cdot x^1$
Сумма квадратов ошибок: 0.470119

Многочлен 2-й степени:
 $3.547550 \cdot x^0 + -0.398052 \cdot x^1 + 0.118534 \cdot x^2$
Сумма квадратов ошибок: 0.125965

Аппроксимация методом наименьших квадратов



3.4. Вычислить первую и вторую производную от таблично заданной функции

Формулы численного дифференцирования в основном используются при нахождении производных от функции $y = f(x)$, заданной таблично. Исходная функция $y_i = f(x_i)$, $i = 0, 1, \dots, M$ на отрезках $[x_j, x_{j+k}]$ заменяется некоторой приближающей, легко вычисляемой функцией $\varphi(x, \bar{a})$, $y = \varphi(x, \bar{a}) + R(x)$, где $R(x)$ – остаточный член приближения, \bar{a} – набор коэффициентов, вообще говоря, различный для каждого из рассматриваемых отрезков, и полагают, что $y'(x) \approx \varphi'(x, \bar{a})$. Наиболее часто в качестве приближающей функции $\varphi(x, \bar{a})$ берется интерполяционный многочлен $\varphi(x, \bar{a}) = P_n(x) = \sum_{i=0}^n a_i x^i$, а производные соответствующих порядков определяются дифференцированием многочлена.

При решении практических задач, как правило, используются аппроксимации первых и вторых производных.

В первом приближении, таблично заданная функция может быть аппроксимирована отрезками прямой $y(x) \approx \varphi(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i)$, $x \in [x_i, x_{i+1}]$. В этом случае:

$$y'(x) \approx \varphi'(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} = \text{const}, \quad x \in [x_i, x_{i+1}], \quad (3.18)$$

производная является кусочно-постоянной функцией и рассчитывается, по формуле (3.18) с первым порядком точности в крайних точках интервала, и со вторым порядком точности в средней точке интервала [1].

При использовании для аппроксимации таблично заданной функции интерполяционного многочлена второй степени имеем:

$$y(x) \approx \varphi(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) + \frac{\frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}}{\frac{x_{i+2} - x_{i+1}}{x_{i+2} - x_i}}(x - x_i)(x - x_{i+1}), \quad x \in [x_i, x_{i+1}] \quad (3.19)$$

$$y'(x) \approx \varphi'(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} + \frac{\frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}}{\frac{x_{i+2} - x_{i+1}}{x_{i+2} - x_i}}(2x - x_i - x_{i+1}), \quad x \in [x_i, x_{i+1}] \quad (3.20)$$

При равностоценных точках разбиения, данная формула обеспечивает второй порядок точности.

Для вычисления второй производной, необходимо использовать интерполяционный многочлен, как минимум второй степени. После дифференцирования многочлена получаем

$$y''(x) \approx \varphi''(x) = 2 \frac{\frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}}{x_{i+2} - x_i}, \quad x \in [x_i, x_{i+1}] \quad (3.21)$$

Условие:

15. $X^* = 0.4$

i	0	1	2	3	4
x_i	0.0	0.2	0.4	0.6	0.8
y_i	1.0	1.4214	1.8918	2.4221	3.0255

Код:

```

1 import matplotlib.pyplot as plt
2
3 def compute_derivatives(x, y, x_star):
4     i = -1
5     for j in range(len(x) - 1):
6         if x[j] <= x_star <= x[j + 1]:
7             i = j + 1
8             break
9
10    if i == -1 or i + 1 >= len(x):
11        print("Ошибка: x* вне диапазона данных или недостаточно точек.")
12        return
13
14    # Первая производная (левосторонняя и правосторонняя)
15    left_diff = (y[i] - y[i - 1]) / (x[i] - x[i - 1])
16    right_diff = (y[i + 1] - y[i]) / (x[i + 1] - x[i])
17
18    # Формулы 3.20 (приближение первой производной в x*)
19    temp = (right_diff - left_diff) / (x[i + 1] - x[i - 1])
20    first_derivative = left_diff + temp * (2 * x_star - x[i] - x[i - 1])
21
22    # Формула 3.21 (вторая производная в x*)
23    second_derivative = 2 * temp
24
25    print(f"Левосторонняя производная: {left_diff:.10f}")
26    print(f"Правосторонняя производная: {right_diff:.10f}")
27    print(f"Первая производная в x*={x_star}: {first_derivative:.10f}")
28    print(f"Вторая производная в x*={x_star}: {second_derivative:.10f}")
29
30    # Визуализация
31    plt.figure(figsize=(10, 6))

```

```

32     # График точек
33     plt.plot(x, y, 'bo-', label='Табличные значения')
34
35
36     # Левая секущая
37     x_left = [x[i - 1], x[i]]
38     y_left = [y[i - 1], y[i]]
39     plt.plot(x_left, y_left, 'g--', label='Левосторонняя секущая')
40
41     # Правая секущая
42     x_right = [x[i], x[i + 1]]
43     y_right = [y[i], y[i + 1]]
44     plt.plot(x_right, y_right, 'r--', label='Правосторонняя секущая')
45
46     # Приближённая касательная (в x*)
47     x_tangent = [x_star - 0.05, x_star + 0.05]
48     y_tangent = [y[i] + first_derivative * (xi - x[i]) for xi in x_tangent]
49     plt.plot(x_tangent, y_tangent, 'm--', label='Касательная (1-я производная)')
50
51     # Вертикальная линия на x*
52     plt.axvline(x_star, color='k', linestyle=':', label='x*')
53
54     # Настройки
55     plt.title("Приближение производных по табличным данным")
56     plt.xlabel("x")
57     plt.ylabel("y")
58     plt.legend()
59     plt.grid(True)
60     plt.show()
61
62     # Пример данных
63     x = [0, 0.1, 0.2, 0.3, 0.4]
64     y = [1, 1.1052, 1.2214, 1.3499, 1.4918]
65     x_star = 0.2
66
67     compute_derivatives[x, y, x_star]      You, 2 недели назад • added new labs ...

```

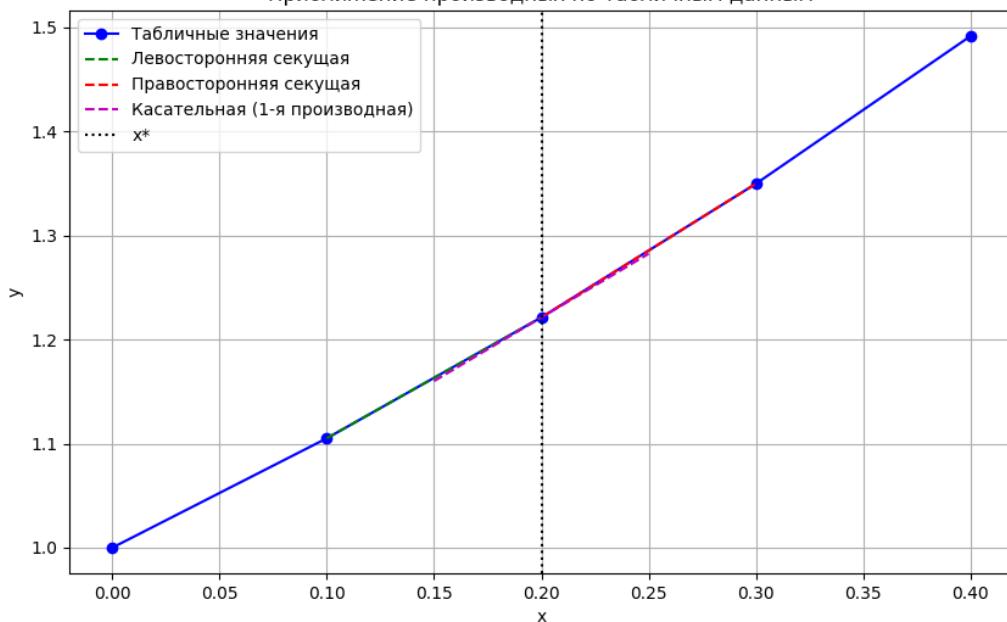
Результат:

```

Левосторонняя производная: 1.1620000000
Правосторонняя производная: 1.2850000000
Первая производная в x*=0.2: 1.2235000000
Вторая производная в x*=0.2: 1.2300000000

```

Приближение производных по табличным данным



3.5. Вычислить определенный интеграл $F = \int_{x_0}^{x_1} y dx$, методами прямоугольников, трапеций, Симпсона с шагами h_1, h_2 . Оценить погрешность вычислений, используя Метод Рунге-Ромберга:

3.4. ЧИСЛЕННОЕ ИНТЕГРИРОВАНИЕ

Формулы численного интегрирования используются в тех случаях, когда вычислить аналитически определенный интеграл $F = \int_a^b f(x) dx$ не удается. Отрезок

$[a, b]$ разбивают точками x_0, \dots, x_N , так что $a = x_0 \leq x_1 \leq \dots \leq x_N = b$ с достаточно мелким шагом $h_i = x_i - x_{i-1}$ и на одном или нескольких отрезках h_i подынтегральную функцию $f(x)$ заменяют такой приближающей $\varphi(x)$, так что она, во-первых, близка $f(x)$, а, во-вторых, интеграл от $\varphi(x)$ легко вычисляется. Рассмотрим наиболее простой и часто применяемый способ, когда подынтегральную функцию заменяют на интерполяционный многочлен $P_n(x) = \sum_{j=0}^n a_j x^j$, причем коэффициенты многочлена a_j , вообще говоря, различны на каждом отрезке $[x_i, x_{i+k}]$ и определяются из условия $\varphi(x_j) = f(x_j)$, $j = i, \dots, i+k$, т.е. многочлен P_n зависит от параметров a_j - $P_n(x, \bar{a}_i)$, тогда

$$f(x) = P_n(x, \bar{a}_i) + R_n(x, \bar{a}_i), \quad x \in [x_i, x_{i+k}], \quad (3.22)$$

где $R_n(x, \bar{a}_i)$ – остаточный член интерполяции. Тогда $F = \sum_{i=1}^N \int_{x_{i-1}}^{x_i} P_n(x, \bar{a}_i) dx + R$,

где $R = \sum_{i=1}^N \int_{x_{i-1}}^{x_i} R_n(x, \bar{a}_i) dx$ – остаточный член формулы численного интегрирования или её погрешность.

Заменим подынтегральную функцию, интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка – точку $\bar{x}_i = (x_{i-1} + x_i) / 2$, получим формулу прямоугольников.

$$\int_a^b f(x) dx \approx \sum_{i=1}^N h_i f\left(\frac{x_{i-1} + x_i}{2}\right) \quad (3.23)$$

В случае постоянного шага интегрирования $h_i = h, i = 1, 2, \dots, N$ и существования $f''(x), x \in [a, b]$, имеет место оценка остаточного члена формулы прямоугольников

$$R \leq \frac{1}{24} h^2 M_2 (b - a), \quad (3.24)$$

где $M_2 = \max |f''(x)|_{[a, b]}$.

В случае таблично заданных функций удобно в качестве узлов интерполяции выбрать начало и конец отрезка интегрирования, т.е. заменить функцию $f(x)$ многочленом Лагранжа первой степени.

$$F = \int_a^b f(x) dx \approx \frac{1}{2} \sum_{i=1}^N (f_i + f_{i-1}) h_i \quad (3.25)$$

Эта формула носит название формулы трапеций.

В случае постоянного шага интегрирования величина остаточного члена оценивается

$$R \leq \frac{b-a}{12} h^2 M_2, \quad (3.26)$$

где $M_2 = \max |f''(x)|_{[a,b]}.$

Для повышения порядка точности формулы численного интегрирования заменим подынтегральную кривую параболой – интерполяционным многочленом второй степени, выбрав в качестве узлов интерполяции концы и середину отрезка интегрирования:

$$x_{i-1}, x_{\frac{i-1}{2}} = (x_{i-1} + x_i)/2, x_i.$$

Для случая $h_i = \frac{x_i - x_{i-1}}{2}$, получим формулу Симпсона (парабол)

$$F = \int_a^b f(x) dx \approx \frac{1}{3} \sum_{i=1}^N (f_{i-1} + 4f_{\frac{i-1}{2}} + f_i) h_i \quad (3.27)$$

В случае постоянного шага интегрирования $h_i = h, i = 1, 2, \dots, N$, формула Симпсона принимает вид.

$$F \approx \frac{h}{3} \left[f_0 + 4f_{\frac{1}{2}} + 2f_1 + 4f_{\frac{3}{2}} + 2f_2 + \dots + 2f_{\frac{N-1}{2}} + 4f_{\frac{N-1}{2}} + f_N \right], \quad (3.28)$$

при этом количество интервалов на которое делится отрезок интегрирования, равно $2N$.

В том случае если существует $f^{IV}(x)$, $x \in [a,b]$, для оценки величины погрешности справедлива мажорантная оценка

$$R \leq \frac{(b-a)}{180} h^4 M_4, \quad (3.29)$$

где $M_4 = \max \left| f^{IV}(x) \right|_{[a,b]}$.

Метод Рунге-Ромберга-Ричардсона позволяет получать более высокий порядок точности вычисления. Если имеются результаты вычисления определенного интеграла на сетке с шагом h - $F = F_h + O(h^p)$ и на сетке с шагом kh - $F = F_{kh} + O((kh)^p)$, то

$$F = \int_a^b f(x) dx = F_h + \frac{F_h - F_{kh}}{k^p - 1} + O(h^{p+1}) \quad (3.30)$$

Условие:

$$15. \quad y = \frac{x}{x^4 + 81}, \quad X_0 = 0, \quad X_k = 2, \quad h_1 = 0.5, \quad h_2 = 0.25;$$

Код программы:

```

1 import numpy as np
2 import sympy as sp
3 import matplotlib.pyplot as plt
4 from scipy.interpolate import BarycentricInterpolator
5
6 def f(x):
7     """
8         Возвращает заданную функцию
9     """
10    return x / (x**4 + 81)
11
12 def exact_integral():
13     """
14         Вычисляет точное значение определённого интеграла с помощью библиотеки SymPy
15     """
16    x = sp.symbols('x')
17    integrand = x / (x**4 + 81)
18    result = sp.integrate(integrand, (x, 0, 2))
19    return float(result.evalf())
20

```

```

21 def generate_x(a, b, h):
22     """
23     Генерирует массив точек на отрезке [a, b] с шагом h
24     """
25     n = int((b - a) / h)
26     return np.linspace(a, b, n + 1)
27
28 def evaluate_function(x):
29     """
30     Вычисляет значения функции f(x) в каждой точке массива x
31     """
32     return f(x)
33
34 def rectangle_rule(x, h):
35     """
36     Вычисляет определённый интеграл методом прямоугольников.
37     Использует средние точки на каждом интервале
38     """
39     sum_val = 0.0
40     for i in range(len(x) - 1):
41         mid = (x[i] + x[i + 1]) / 2
42         sum_val += f(mid)
43     return h * sum_val
44
45 def trapezoid_rule(y, h):
46     """
47     Вычисляет определённый интеграл методом трапеций
48     """
49     return h * (y[0] / 2 + y[-1] / 2 + np.sum(y[1:-1]))
50
51 def simpson_rule(y, h):
52     """
53     Вычисляет определённый интеграл методом Симпсона
54     (работает только при чётном числе интервалов)
55     """
56     n = len(y)
57     if (n - 1) % 2 != 0:
58         print("Ошибка: нечетное число интервалов для Симпсона.")
59         return float('nan')
60     sum_val = y[0] + y[-1]
61     for i in range(1, n - 1):
62         sum_val += 4 * y[i] if i % 2 != 0 else 2 * y[i]
63     return h / 3 * sum_val
64
65 def rect_residual(a, b, h):
66     """
67     Оценивает остаточный член для метода прямоугольников
68     """
69     M2 = 0.01 # максимум |f''(x)| на [a, b], оценка вручную
70     return (M2 * (b - a) * h**2) / 24
71
72 def trap_residual(a, b, h):
73     """
74     Оценивает остаточный член для метода трапеций
75     """
76     M2 = 0.01
77     return (M2 * (b - a) * h**2) / 12
78
79 def simpson_residual(a, b, h):
80     """
81     Оценивает остаточный член для метода Симпсона
82     """
83     M4 = 0.05 # оценка производной f^(4)(x)
84     return (M4 * (b - a)**5) / (180 * ((b - a)/h)**4)
85

```

```

86     def runge_romberg(method, a, b, h1, h2, p):
87         """
88             Вычисляет уточнённое значение интеграла и абсолютную погрешность по методу Рунге–Ромберга
89         """
90         x1 = generate_x(a, b, h1)
91         x2 = generate_x(a, b, h2)
92         y1 = evaluate_function(x1)
93         y2 = evaluate_function(x2)
94
95         if method == "Прямоугольники":
96             I1 = rectangle_rule(x1, h1)
97             I2 = rectangle_rule(x2, h2)
98         elif method == "Трапеции":
99             I1 = trapezoid_rule(y1, h1)
100            I2 = trapezoid_rule(y2, h2)
101        elif method == "Симпсон":
102            I1 = simpson_rule(y1, h1)
103            I2 = simpson_rule(y2, h2)
104        else:
105            raise ValueError("Неизвестный метод")
106
107        k = h2 / h1
108        refined = I1 + (I1 - I2) / (k**p - 1)
109        error = abs(refined - exact_integral())
110
111        print(f"{method} → уточнённое значение: {refined:.8f}, абсолютная погрешность: {error:.8f}")
112
113 # ===== ГРАФИКИ =====
114 def plot_function(a, b):
115     """
116         Строит график функции f(x)
117     """
118     x_vals = np.linspace(a, b, 500)
119     y_vals = f(x_vals)
120     plt.figure(figsize=(8, 4))
121     plt.plot(x_vals, y_vals, label='f(x)', color='blue')
122     plt.title('График функции f(x)')
123     plt.xlabel('x')
124     plt.ylabel('f(x)')
125     plt.grid(True)
126     plt.legend()
127     plt.show()
128
129 def plot_rectangle_method(x, h):
130     """
131         Визуализирует метод прямоугольников
132     """
133     plt.figure(figsize=(8, 4))
134     x_vals = np.linspace(x[0], x[-1], 500)
135     plt.plot(x_vals, f(x_vals), 'b', label='f(x)')
136     for i in range(len(x) - 1):
137         mid = (x[i] + x[i + 1]) / 2
138         height = f(mid)
139         plt.bar(mid, height, width=h, align='center', alpha=0.4, edgecolor='black')
140     plt.title("Метод прямоугольников")
141     plt.grid(True)
142     plt.legend()
143     plt.show()
144
145 def plot_trapezoid_method(x, y):
146     """
147         Визуализирует метод трапеций
148     """
149     plt.figure(figsize=(8, 4))
150     plt.plot(x, y, 'b', label='f(x)')

```

```

151     for i in range(len(x) - 1):
152         plt.fill([x[i], x[i], x[i+1], x[i+1]],
153                  [0, y[i], y[i+1], 0],
154                  'orange', edgecolor='black', alpha=0.4)
155     plt.title("Метод трапеций")
156     plt.grid(True)
157     plt.legend()
158     plt.show()
159
160 def plot_simpson_method(x, y):
161     """
162     Строит график парабол, аппроксимирующих f(x) по методу Симпсона
163     """
164     plt.figure(figsize=(10, 5))
165     plt.title("Метод Симпсона: приближение параболами")
166     plt.xlabel("x")
167     plt.ylabel("f(x)")
168
169     colors = plt.cm.viridis(np.linspace(0.3, 0.9, (len(x) - 1) // 2))
170
171     # Для каждой тройки узлов строим параболу и закрашиваем площадь под ней
172     for i in range(0, len(x) - 2, 2):
173         xi = x[i:i+3]
174         yi = y[i:i+3]
175
176         # Интерполяция параболой через 3 точки
177         parabola = BarycentricInterpolator(xi, yi)
178
179         xf = np.linspace(xi[0], xi[-1], 100)
180         yf = parabola(xf)
181
182         plt.plot(xf, yf, color=colors[i // 2], label=f"Парабола [{xi[0]}:{.2f}, {xi[-1]}:{.2f}]")
183         plt.fill_between(xf, yf, alpha=0.2, color=colors[i // 2])
184
185         # Узлы параболы
186         plt.plot(xi, yi, 'ko')
187
188     plt.legend(loc="upper right", fontsize=8)
189     plt.grid(True)
190     plt.tight_layout()
191     plt.show()

```

Результат:

```

===== Визуализация функции =====

===== Шаг h = 0.5 =====
Прямоугольники: 0.02332650
Трапеции: 0.02305084
Симпсон: 0.02323298
Оценка остатка (Rect): 0.00020833
Оценка остатка (Trap): 0.00041667
Оценка остатка (Simpson): 0.00003472

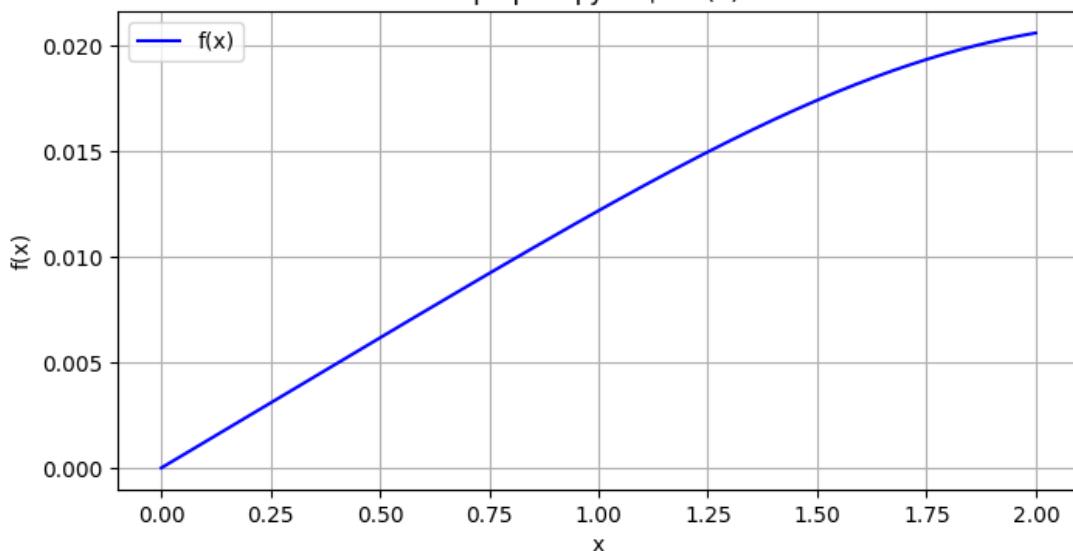
===== Шаг h = 0.25 =====
Прямоугольники: 0.02325769
Трапеции: 0.02318867
Симпсон: 0.02323461
Оценка остатка (Rect): 0.00005208
Оценка остатка (Trap): 0.00010417
Оценка остатка (Simpson): 0.00000217

===== Метод Рунге–Ромберга–Ричардсона =====
Прямоугольники → уточнённое значение: 0.02323475, абсолютная погрешность: 0.00000006
Трапеции → уточнённое значение: 0.02323461, абсолютная погрешность: 0.00000007
Симпсон → уточнённое значение: 0.02323472, абсолютная погрешность: 0.00000004

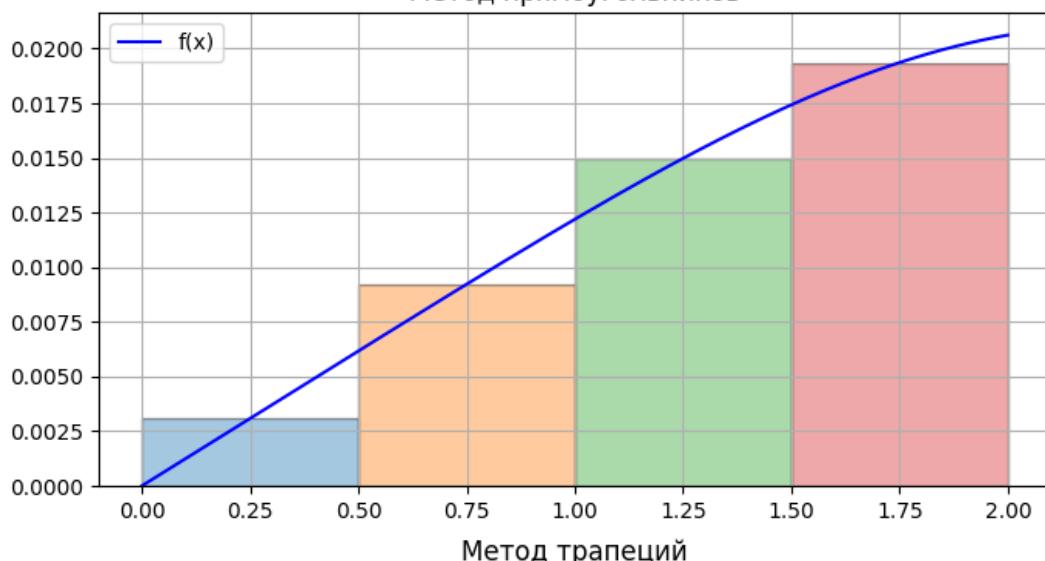
===== Точное значение интеграла =====
Точное значение: 0.02323468

```

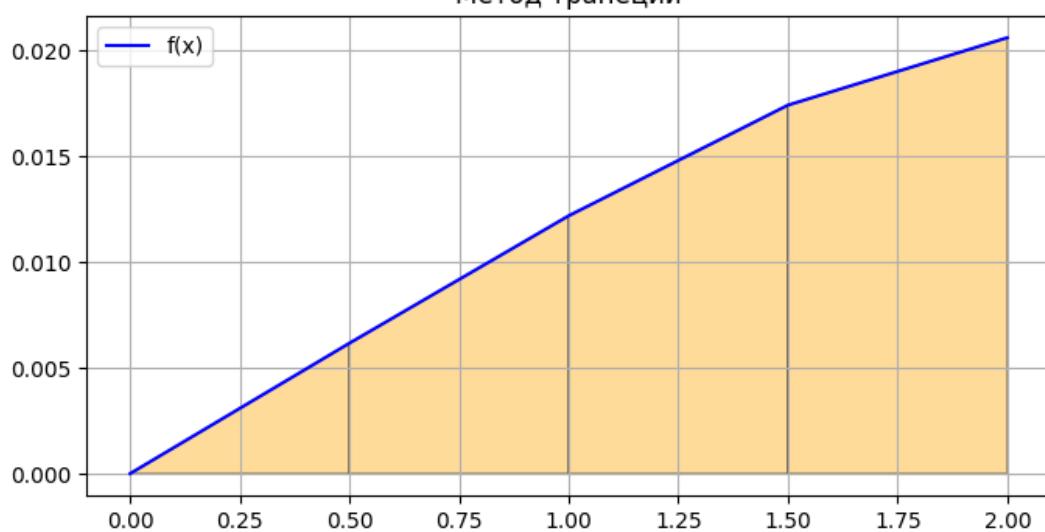
График функции $f(x)$



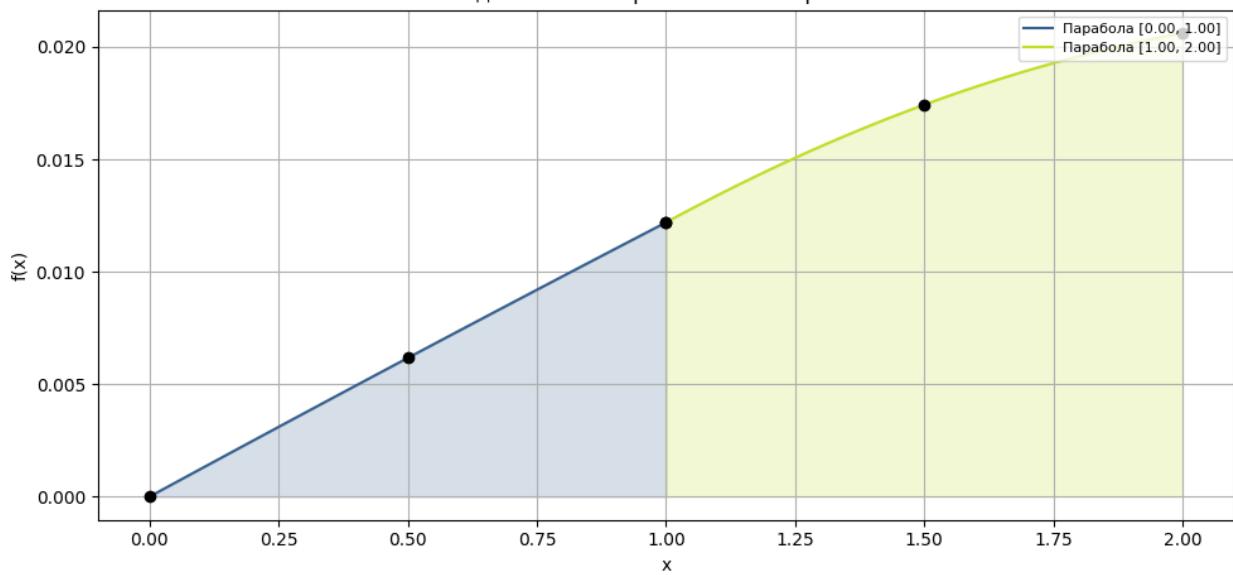
Метод прямоугольников



Метод трапеций



Метод Симпсона: приближение параболами



Лабораторная работа №4

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

Условие:

15	$xy'' + y' = 0$, $y(1) = 1$, $y'(1) = 1$, $x \in [1, 2], h = 0.1$	$y = 1 + \ln x $
----	---	------------------

График функции $y^{(h)}$, которая является решением задачи Коши (1), представляет собой гладкую кривую, проходящую через точку (x_0, y_0) согласно условию $y(x_0) = y_0$, и имеет в этой точке касательную. Тангенс угла наклона касательной к оси Ох равен значению производной от решения в точке x_0 и равен значению правой части дифференциального уравнения в точке (x_0, y_0) согласно выражению $y'(x_0) = f(x_0, y_0)$. В случае небольшого шага разностной сетки h график функции и график касательной не успевают сильно разойтись друг от друга и можно в качестве значения решения в узле x_1 принять значение касательной y_1 , вместо значения неизвестного точного решения $y_{1\text{иск}}$. При этом допускается погрешность $|y_1 - y_{1\text{иск}}|$ геометрически представлена отрезком CD на рис.4.1. Из прямоугольного треугольника ABC находим $CB=BA \ tg(CAB)$ или $\Delta y = hy'(x_0)$. Учитывая, что $\Delta y = y_1 - y_0$ и заменяя производную $y'(x_0)$ на правую часть дифференциального уравнения, получаем соотношение $y_1 = y_0 + hf(x_0, y_0)$. Считая теперь точку (x_1, y_1) начальной и повторяя все предыдущие рассуждения, получим значение y_2 в узле x_2 .

Переход к произвольным индексам дает формулу метода Эйлера:

$$y_{k+1} = y_k + hf(x_k, y_k) \quad (4.2)$$

Погрешность метода Эйлера.

На каждом шаге метода Эйлера допускается *локальная* погрешность по отношению к точному решению, график которого проходит через крайнюю левую точку отрезка. Геометрически локальная погрешность изображается отрезком CD на первом шаге, C'D' на втором и т.д. Кроме того, на каждом шаге, начиная со второго, накапливается *глобальная* погрешность представляющая собой разность между численным решением и точным решением исходной начальной задачи (а не локальной). Глобальная погрешность на втором шаге изображена отрезком C'E' на рис.4.1.

Локальная ошибка на каждом шаге выражается соотношением $\varepsilon_k^h = \frac{y''(\xi)}{2} h^2$, где

$\xi \in [x_{k-1}, x_k]$. Глобальная погрешность метода Эйлера $\varepsilon_{\text{гл}}^h = Ch$ в окрестности $h=0$ ведет себя как линейная функция, и, следовательно, метод Эйлера имеет первый порядок точности относительно шага h .

Метод Адамса

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

$$y_{k+1} = y_k + \frac{h}{24} (55f_k - 59f_{k-1} + 37f_{k-2} - 9f_{k-3}), \quad (4.25)$$

где f_k значение подынтегральной функции в узле x_k .

Метод Адамса (4.25) как и все многошаговые методы не является самостартующим, то есть для того, что бы использовать метод Адамса необходимо иметь решения в первых четырех узлах. В узле x_0 решение y_0 известно из начальных условий, а в других трех узлах x_1, x_2, x_3 решения y_1, y_2, y_3 можно получить с помощью подходящего одношагового метода, например: метода Рунге-Кутты четвертого порядка (4.10).

Методы Рунге-Кутты

Все рассмотренные выше явные методы являются вариантами методов Рунге-Кутты.

Семейство явных методов Рунге-Кутты p -го порядка записывается в виде совокупности формул:

$$\begin{aligned}y_{k+1} &= y_k + \Delta y_k \\ \Delta y_k &= \sum_{i=1}^p c_i K_i^k \\ K_i^k &= h f(x_k + a_i h, y_k + h \sum_{j=1}^{i-1} b_{ij} K_j^k) \\ i &= 2, 3, \dots, p\end{aligned}\tag{4.8}$$

Параметры a_i, b_{ij}, c_i подбираются так, чтобы значение y_{k+1} , рассчитанное по соотношению (4.8) совпадало со значением разложения в точке x_{k+1} точного решения в ряд Тейлора с погрешностью $O(h^{p+1})$

Код программы:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Правая часть системы ( $y'' = f(x, y, y')$ ) для уравнения:  $xy'' + y' = 0 \Rightarrow y'' = -y'/x$ 
5
6 def f(x, y, z):
7     return -z / x
8
9 # Точное решение
10 exact_solution = lambda x: 1 + np.log(np.abs(x))
11
12 # Метод Эйлера
13 def euler_method(f, x0, y0, z0, h, n):
14     x_vals, y_vals, z_vals = [x0], [y0], [z0]
15     for _ in range(n):
16         x, y, z = x_vals[-1], y_vals[-1], z_vals[-1]
17         x_next = x + h
18         y_next = y + h * z
19         z_next = z + h * f(x, y, z)
20         x_vals.append(x_next)
21         y_vals.append(y_next)
22         z_vals.append(z_next)
23
24 return x_vals, y_vals
```

```

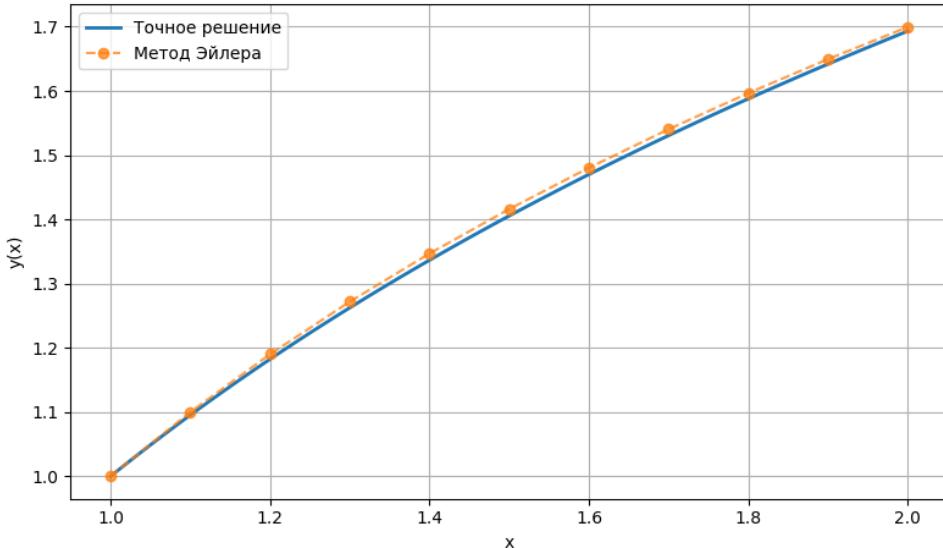
24
25 # Метод Рунге-Кутты 4-го порядка
26 def runge_kutta_4(f, x0, y0, z0, h, n):
27     x_vals, y_vals, z_vals = [x0], [y0], [z0]
28     for _ in range(n):
29         x, y, z = x_vals[-1], y_vals[-1], z_vals[-1]
30
31         k1 = h * z
32         l1 = h * f(x, y, z)
33
34         k2 = h * (z + l1 / 2)
35         l2 = h * f(x + h / 2, y + k1 / 2, z + l1 / 2)
36
37         k3 = h * (z + l2 / 2)
38         l3 = h * f(x + h / 2, y + k2 / 2, z + l2 / 2)
39
40         k4 = h * (z + l3)
41         l4 = h * f(x + h, y + k3, z + l3)
42
43         x_vals.append(x + h)
44         y_vals.append(y + (k1 + 2*k2 + 2*k3 + k4) / 6)
45         z_vals.append(z + (l1 + 2*l2 + 2*l3 + l4) / 6)
46     return x_vals, y_vals, z_vals
47
48 # Метод Адамса 4-го порядка
49 def adams_method(f, x0, y0, z0, h, n):
50     x_vals, y_vals, z_vals = runge_kutta_4(f, x0, y0, z0, h, 3)
51
52     for i in range(3, n):
53         x0, x1, x2, x3 = x_vals[-4:]
54         y0, y1, y2, y3 = y_vals[-4:]
55         z0, z1, z2, z3 = z_vals[-4:]
56
57         f0 = z0
58         f1 = z1
59         f2 = z2
60         f3 = z3
61
62         g0 = f(x0, y0, z0)
63         g1 = f(x1, y1, z1)
64         g2 = f(x2, y2, z2)
65         g3 = f(x3, y3, z3)
66
67         y_next = y3 + h * (55*f3 - 59*f2 + 37*f1 - 9*f0) / 24
68         z_next = z3 + h * (55*g3 - 59*g2 + 37*g1 - 9*g0) / 24
69
70         x_vals.append(x3 + h)
71         y_vals.append(y_next)
72         z_vals.append(z_next)
73
74     return x_vals, y_vals
75
76 # Метод Рунге-Ромберга для оценки погрешности
77 def runge_romberg(y_h, y_h2, p):
78     y_h2_thinned = y_h2[::2]
79     return [(y2 - y1) / (2**p - 1) for y1, y2 in zip(y_h, y_h2_thinned)]
80
81 # Отрисовка отдельных графиков для каждого метода
82 def plots(x_exact, y_exact, results):
83     for label, (x_vals, y_vals) in results.items():
84         plt.figure(figsize=(8, 5))
85         plt.plot(x_exact, y_exact, label='Точное решение', linewidth=2)
86         plt.plot(x_vals, y_vals, '--o', label=label, alpha=0.75)
87         plt.title(f"Сравнение: {label} и точное решение")
88         plt.xlabel("x")
89         plt.ylabel("y(x)")
90         plt.grid(True)
91         plt.legend()
92         plt.tight_layout()
93         plt.show()

```

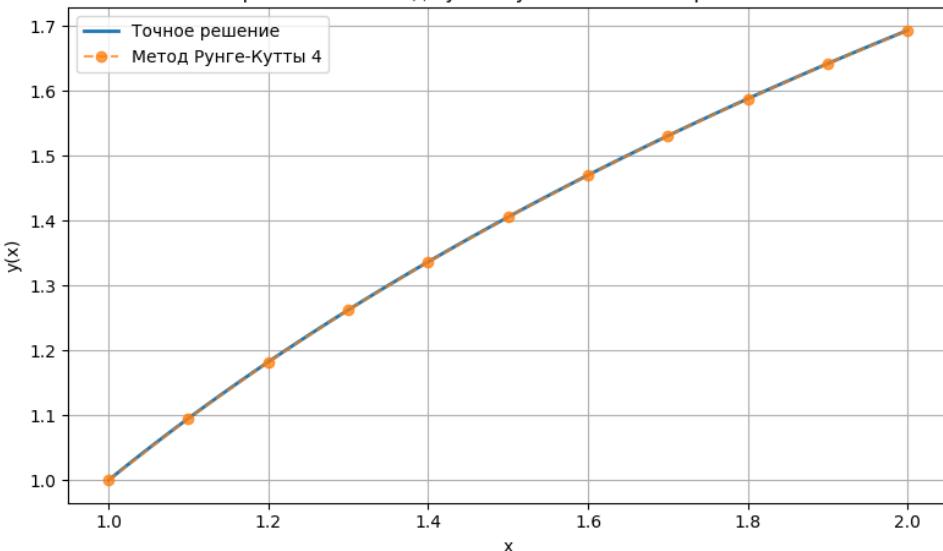
Результат:

Оценка ошибки (метод Рунге–Ромберга) в $x = 2.0$:
Эйлер (порядок 1): -0.00337160
Рунге–Кутта 4 порядка: 0.00000005
Адамс–Бэнфорд 4 порядка: -0.00000348

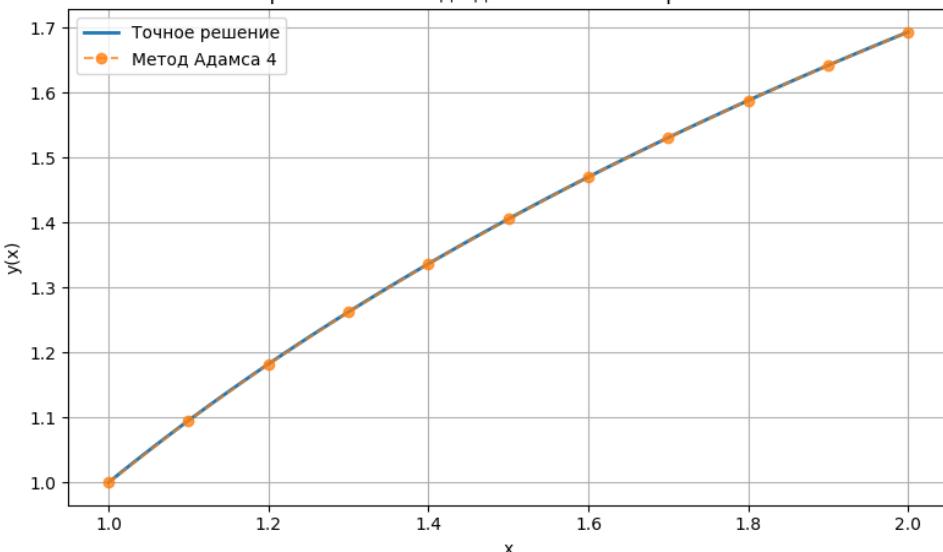
Сравнение: Метод Эйлера и точное решение



Сравнение: Метод Рунге–Кутты 4 и точное решение



Сравнение: Метод Адамса 4 и точное решение



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

Условие:

15	$x^2 \ln x y'' - xy' + y = 0,$ $y'(-1) = 0,$ $y'(1) - y(1) = 0$	$y(x) = 1 + x + \ln x$
----	---	------------------------

1)Метод стрельбы

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

Пусть надо решить краевую задачу (4.28), (4.29) на отрезке $[a, b]$. Вместо исходной задачи формулируется задача Коши с уравнением (4.28) и с начальными условиями

$$\begin{aligned} y(a) &= y_0 \\ y'(b) &= \eta \end{aligned} \quad , \quad (4.32)$$

где η - некоторое значение тангенса угла наклона касательной к решению в точке $x = a$.

Положим сначала некоторое начальное значение параметру $\eta = \eta_0$, после чего решим каким либо методом задачу Коши (4.28),(4.32). Пусть $y = y_0(x, y_0, \eta_0)$ решение этой задачи на интервале $[a, b]$, тогда сравнивая значение функции $y_0(b, y_0, \eta_0)$ со значением y_1 в правом конце отрезка можно получить информацию для корректировки угла наклона касательной к решению в левом конце отрезка. Решая задачу Коши для нового значения $\eta = \eta_1$, получим другое решение со значением $y_1(b, y_0, \eta_1)$ на правом конце. Таким образом, значение решения на правом конце $y(b, y_0, \eta)$ будет являться функцией одной переменной η . Задачу можно сформулировать таким образом: требуется найти такое значение переменной η^* , чтобы решение $y(b, y_0, \eta^*)$ в правом конце отрезка совпало со значением y_1 из (4.29). Другими словами решение исходной задачи эквивалентно нахождению корня уравнения

$$\Phi(\eta) = 0, \quad (4.33)$$

где $\Phi(\eta) = y(b, y_0, \eta) - y_1$.

Уравнение (4.33) является “алгоритмическим” уравнением, так как левая часть его задается с помощью алгоритма численного решения соответствующей задачи Коши. Но методы решения уравнения (4.33) аналогичны методам решения нелинейных уравнений, изложенным в разделе 2. Следует заметить, что так как невозможно вычислить производную функции $\Phi(\eta)$, то вместо метода Ньютона следует использовать метод секущих, в котором производная от функции заменена ее разностным аналогом. Данный разностный аналог легко вычисляется по двум приближениям, например η_k и η_{k+1} . Следующее значение искомого корня определяется по соотношению

$$\eta_{j+2} = \eta_{j+1} - \frac{\eta_{j+1} - \eta_j}{\Phi(\eta_{j+1}) - \Phi(\eta_j)} \Phi(\eta_{j+1}) \quad (4.34)$$

Итерации по формуле (4.34) выполняются до удовлетворения заданной точности.

2) Конечно-разностный метод

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

$$y'' + p(x)y' + q(x)y = f(x) \quad (4.35)$$

$$y(a) = y_0, y(b) = y_1. \quad (4.36)$$

Введем разностную сетку на отрезке $[a, b]$ $\Omega^{(h)} = \{x_k = x_0 + hk\}, k = 0, 1, \dots, N$, $h = |b - a| / N$. Решение задачи (4.35), (4.36) будем искать в виде сеточной функции $y^{(h)} = \{y_k, k = 0, 1, \dots, N\}$, предполагая, что решение существует и единственno. Введем разностную аппроксимацию производных следующим образом:

$$\begin{aligned} y'_k &= \frac{y_{k+1} - y_{k-1}}{2h} + O(h^2); \\ y''_k &= \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + O(h^2); \end{aligned} \quad (4.37)$$

Подставляя аппроксимации производных из (4.37) в (4.35),(4.36) получим систему уравнений для нахождения y_k :

$$\begin{cases} y_0 = y_a \\ \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + p(x_k) \frac{y_{k+1} - y_{k-1}}{2h} + q(x_k)y_k = f(x_k), k = 1, N-1 \\ y_N = y_b \end{cases} \quad (4.38)$$

Приводя подобные и учитывая, что при задании граничных условий первого рода два неизвестных y_0, y_N уже фактически определены, получим систему линейных алгебраических уравнений с трехдиагональной матрицей коэффициентов

$$\begin{cases} (-2 + h^2 q(x_1))y_1 + (1 + \frac{p(x_1)h}{2})y_2 = h^2 f(x_1) - (1 - \frac{p(x_1)h}{2})y_a \\ (1 - \frac{p(x_k)h}{2})y_{k-1} + (-2 + h^2 q(x_k))y_k + (1 + \frac{p(x_k)h}{2})y_{k+1} = h^2 f(x_k), k = 2, \dots, N-2 \\ (1 - \frac{p(x_{N-1})h}{2})y_{N-1} + (-2 + h^2 q(x_{N-1}))y_N = h^2 f(x_{N-1}) - (1 + \frac{p(x_{N-1})h}{2})y_b \end{cases} \quad (4.39)$$

Для системы (4.39) при достаточно малых шагах сетки h и $q(x_k) < 0$ выполнены условия преобладания диагональных элементов

$$|-2 + h^2 q(x_k)| > \left|1 - \frac{p(x_k)h}{2}\right| + \left|1 + \frac{p(x_k)h}{2}\right|, \quad (4.39)$$

что гарантирует устойчивость счета и корректность применения метода прогонки для решения этой системы.

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

$$\begin{aligned} y'_0 &= \frac{y_1 - y_0}{h} + O(h); \\ y'_N &= \frac{y_N - y_{N-1}}{h} + O(h) \end{aligned} \quad (4.40)$$

$$\begin{aligned} y'_0 &= \frac{-3y_0 + 4y_1 - y_2}{2h} + O(h^2); \\ y'_N &= \frac{y_{N-2} - 4y_{N-1} + 3y_N}{2h} + O(h^2); \end{aligned} \quad (4.41)$$

Код программы:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Правая часть уравнения
5 def rhs(x, y, dy):
6     return 2 * y * (1 + np.tan(x) ** 2)
7
8 # Точное решение
9 def analytical(x):
10    return -np.tan(x)
11
12 # Метод Рунге-Кутты 4-го порядка
13 # Решает задачу Коши методом Рунге-Кутты (используется внутри метода стрельбы).
14 def rk4_solver(f, x_start, y_start, dy_start, step, steps):
15     x_data, y_data, dy_data = [x_start], [y_start], [dy_start]
16     for _ in range(steps):
17         x, y, dy = x_data[-1], y_data[-1], dy_data[-1]
18
19         k1 = step * dy
20         l1 = step * f(x, y, dy)
21
22         k2 = step * (dy + l1 / 2)
23         l2 = step * f(x + step / 2, y + k1 / 2, dy + l1 / 2)
24
25         k3 = step * (dy + l2 / 2)
26         l3 = step * f(x + step / 2, y + k2 / 2, dy + l2 / 2)
27
28         k4 = step * (dy + l3)
29         l4 = step * f(x + step, y + k3, dy + l3)
30
31         y_next = y + (k1 + 2*k2 + 2*k3 + k4) / 6
32         dy_next = dy + (l1 + 2*l2 + 2*l3 + l4) / 6
33
34         x_data.append(x + step)
35         y_data.append(y_next)
36         dy_data.append(dy_next)
37
38     return x_data, y_data, dy_data
39
40 # Метод стрельбы
41 # Подбирает нужное начальное значение производной, чтобы в конце отрезка получить y(pi/6)=-3^(1/2)/3
42 def shooting(f, x0, x1, y_start, y_end, h, guess1, guess2, tol=1e-8):
43     n_steps = int((x1 - x0) / h)
44
45     def boundary_miss(eta):
46         _, y, _ = rk4_solver(f, x0, y_start, eta, h, n_steps)
47         return y[-1] - y_end
48
49     eta_prev, eta_curr = guess1, guess2
50     while True:
51         # Для каждого значения решаем задачу Коши методом Рунге-Кутты
52         val_prev, val_curr = boundary_miss(eta_prev), boundary_miss(eta_curr)
53         # Смотрим, насколько далеко мы промахнулись в точке x=b
54         if abs(val_curr) < tol or val_curr == val_prev:
55             break
56         # Используем метод секущих для подбора нового приближения
57         eta_next = eta_curr - val_curr * (eta_curr - eta_prev) / (val_curr - val_prev)
58         eta_prev, eta_curr = eta_curr, eta_next
59
60     x_vals, y_vals, _ = rk4_solver(f, x0, y_start, eta_curr, h, n_steps)
61     return x_vals, y_vals
62
63 # Конечно-разностный метод
64 # Заменяет производные на разности – создает систему линейных уравнений и решает её
65 def difference_scheme(x0, x1, y0, yN, n):
66     h = (x1 - x0) / n
67     # Разбиваем отрезок [a,b] на n узлов:
68     x_points = np.linspace(x0, x1, n + 1)
69
```

```

70     # Формируем трёхдиагональную матрицу коэффициентов
71     main_diag = -2 / h**2 + 2 * (1 + np.tan(x_points[1:-1])**2)
72     lower_diag = upper_diag = np.ones(n - 1) / h**2
73     rhs_vector = np.zeros(n - 1)
74     rhs_vector[0] -= y0 / h**2
75     rhs_vector[-1] -= yN / h**2
76
77     alpha = np.zeros(n - 1)
78     beta = np.zeros(n - 1)
79
80     # Решаем систему линейных уравнений методом прогонки
81     alpha[0] = -upper_diag[0] / main_diag[0]
82     beta[0] = rhs_vector[0] / main_diag[0]
83     for i in range(1, n - 1):
84         denom = main_diag[i] + lower_diag[i] * alpha[i - 1]
85         alpha[i] = -upper_diag[i] / denom
86         beta[i] = (rhs_vector[i] - lower_diag[i] * beta[i - 1]) / denom
87
88     y_sol = np.zeros(n + 1)
89     y_sol[0], y_sol[-1] = y0, yN
90     y_sol[-2] = beta[-1]
91     for i in range(n - 3, -1, -1):
92         y_sol[i + 1] = alpha[i] * y_sol[i + 2] + beta[i]
93
94     return x_points, y_sol
95
96     # Метод Рунге-Ромберга
97     # Оценивает погрешность: запускает метод с двумя шагами h и h/2, и сравнивает результат.
98     def rr_error(y_coarse, y_fine, order):
99         y_c = np.array(y_coarse)
100        y_f = np.array(y_fine)[::2][:len(y_c)]
101        err_final = abs((y_f[-1] - y_c[-1]) / (2**order - 1))
102        err_max = np.max(np.abs((y_f - y_c) / (2**order - 1)))
103        return err_final, err_max
104
105    # Функция визуализации
106    def plot_solution(x_num, y_num, method_name, x_exact=None, y_exact=None):
107        plt.figure(figsize=(10, 5))
108        if x_exact is not None and y_exact is not None:
109            plt.plot(x_exact, y_exact, 'k-', label='Точное решение', linewidth=2)
110            plt.plot(x_num, y_num, 'o--', label=method_name, markersize=4)
111            plt.title(f'Решение методом: {method_name}')
112            plt.xlabel('x')
113            plt.ylabel('y(x)')
114            plt.grid(True, linestyle='--', alpha=0.6)
115            plt.legend()
116            plt.tight_layout()
117            plt.show()

```

Результат:

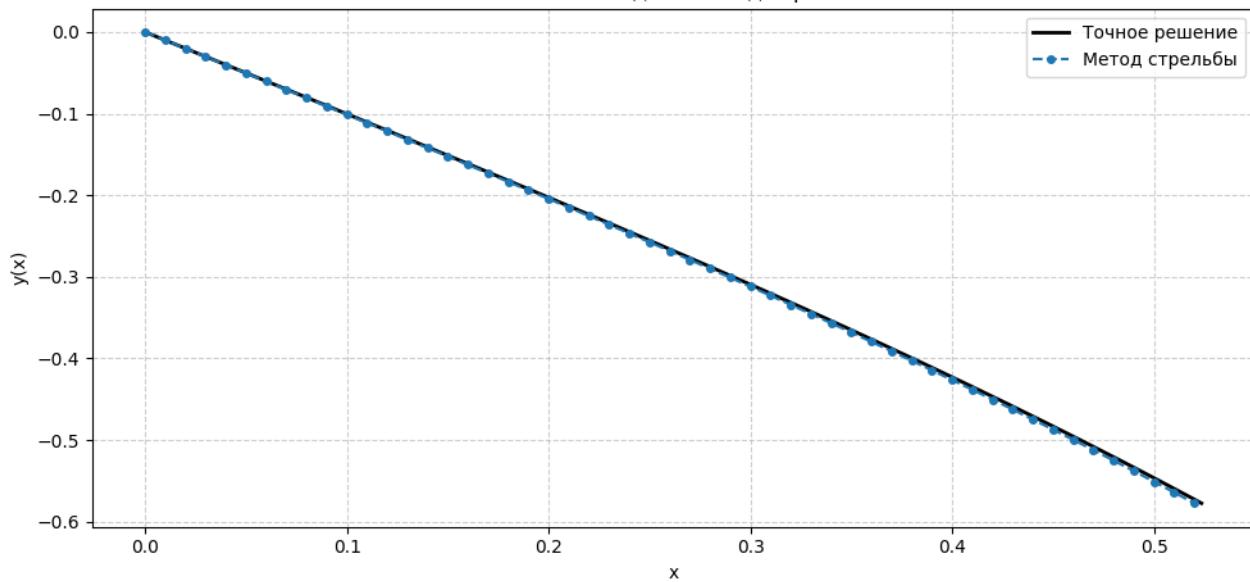
```

== Метод стрельбы ==
Погрешность в последней точке: 0.000e+00
Максимальная погрешность:      4.708e-12

== Конечно-разностный метод ==
Погрешность в последней точке: 0.000e+00
Максимальная погрешность:      2.343e-07

```

Решение методом: Метод стрельбы



Решение методом: Конечно-разностный метод

