МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

Кафедра математической физики и вычислительной математики

ОТЧЁТ
ПО ПРАКТИЧЕСКОЙ ПОДГОТОВКЕ

по дисциплине «Методы вычислений»

студента 3 курса 351 группы

направления 09.03.04 — Программная инженерия

факультета компьютерных наук и информационных технологий

Карасева Вадима Дмитриевича

Проверено: _____ _____

Саратов 2025

1 Построение интерполяционного многочлена в общем виде

Необходимо найти интерполяционный многочлен в общем виде.

x	0	1	2	3
$f(x)$	2	3	10	29

Код

```
 1 import numpy
 2 # входные данные
 3 x_data = [0, 1, 2, 3]
 4 f_data = [2, 3, 10, 29]
 5 x_np_data = numpy.array(x_data)
 6 f_np_data = numpy.array(f_data)
 7 print("Исходные данные:")
 8 for i in range (0, 4):
 9     print('X' + str(i) + ': ' + str(x_data[i]) + '\tF' + str(i) + ': ' + str(f_data[i]))
10 # сторим матрицу
11 matrix = numpy.vander(x_np_data, len(x_data), increasing=True)
12 matrix_flipped = numpy.flip(matrix, axis=1)
13 print("\nСистема построена:")
14 for i in range(0, 4):
15     s = ""
16     for j in range (0, 4):
17         s += str(matrix_flipped[i][j]) + ' * a' + str(4 - j - 1) + ' + '
18     print(s[:-2] + ' = ' + str(f_data[i]))
19 print()
20 # решение системы уравнений для нахождения коэффициентов многочлена
21 coefficients = numpy.linalg.solve(matrix, f_np_data)
22 print("\nКоэффициенты найдены:")
23 for i in range (0, 4):
24     print('a' + str(i) + ': ' + str(coefficients[i]))
25 print("\nP3 (x): ")
26 for i in range (0, 7):
27     print('x = ' + str(i / 2) + ': ' + str(
28         coefficients[3] * (i / 2) * (i / 2) * (i / 2)
29         + coefficients[2] * (i / 2) * (i / 2)
30         + coefficients[1] * (i / 2)
31         + coefficients[0]
32         ))
```

Результат

```
 1 Исходные данные:
 2 X0: 0   F0: 2
 3 X1: 1   F1: 3
 4 X2: 2   F2: 10
 5 X3: 3   F3: 29
 6 Система построена:
 7 0 * a3 + 0 * a2 + 0 * a1 + 1 * a0  = 2
 8 1 * a3 + 1 * a2 + 1 * a1 + 1 * a0  = 3
 9 8 * a3 + 4 * a2 + 2 * a1 + 1 * a0  = 10
10 27 * a3 + 9 * a2 + 3 * a1 + 1 * a0  = 29
11 Коэффициенты найдены:
12 a0: 2.0
13 a1: 0.0
14 a2: -0.0
15 a3: 1.0
```

```
P3 (x):
x = 0.0: 2.0
x = 0.5: 2.125
x = 1.0: 3.0
x = 1.5: 5.375
x = 2.0: 10.0
x = 2.5: 17.625
x = 3.0: 29.0
```

2 Интерполяционный многочлен в форме Лагранжа

По данным интерполяции из предыдущего задания построить интерполяционный многочлен в форме Лагранжа.

x	0	1	2	3
$f(x)$	2	3	10	29

Код

```
import numpy
def lagrange_fundamental(k, x_nodes, z):
        Lk = 1.0
        for i, xi in enumerate(x_nodes):
            if i != k:
                Lk *= (z - xi) / (x_nodes[k] - xi)
        return Lk

def lagrange_interpolation(x_nodes, y_nodes, z):
        P = 0.0
        for k in range(len(x_nodes)):
            P += y_nodes[k] * lagrange_fundamental(k, x_nodes, z)
        return P
x_data = [0, 1, 2, 3]
f_data = [2, 3, 10, 29]
x_np_data = numpy.array(x_data)
f_np_data = numpy.array(f_data)
print("Входные данные:")
for i in range (0, 4):
    print('X' + str(i) + ': ' + str(x_data[i]) + '\tF' + str(i) + ': ' + str(f_data[i]))
print("\nL3 (x): ")
for i in range (0, 7):
    print('x = ' + str(i / 2) + ': ' + str(
        lagrange_interpolation(x_data, f_data, i / 2)
        ))
```

Результат

```
Входные данные:
X0: 0   F0: 2
X1: 1   F1: 3
X2: 2   F2: 10
X3: 3   F3: 29
L3 (x):
x = 0.0: 2.0
x = 0.5: 2.125
x = 1.0: 3.0
x = 1.5: 5.375
x = 2.0: 10.0
```

```
x = 2.5: 17.625
x = 3.0: 29.0
```

3 Интерполяционный многочлен в форме Ньютона

По данным интерполяции из предыдущего задания построить интерполяционный многочлен в форме Ньютона.

x	0	1	2	3
$f(x)$	2	3	10	29

Код

```
import numpy as np
import pandas as pd
# Входные данные
x_data = np.array([0, 1, 2, 3])
f_data = np.array([2, 3, 10, 29])
print("Исходные данные:")
print(pd.DataFrame({'X': x_data, 'F': f_data}))
# Построение таблицы разделённых разностей
def divided_difference_table(x, y):
    n = len(x)
    table = np.zeros((n, n))
    table[:,0] = y
    for j in range(1, n):
        for i in range(n - j):
            table[i,j] = (table[i+1,j-1] - table[i,j-1]) / (x[i+j] - x[i])
    return table
dd_table = divided_difference_table(x_data, f_data)
# Вывод таблицы разделённых разностей
dd_df = pd.DataFrame(dd_table, columns=[f"f[x0..x{j}]" for j in range(len(x_data))])
print("\nТаблица разделённых разностей (матрица Ньютона):")
print(dd_df)
# Коэффициенты Ньютона (верхняя строка таблицы)
newton_coeffs = dd_table[0,:]
print("\nКоэффициенты многочлена Ньютона:")
print(newton_coeffs)

# Функция для вычисления значения многочлена Ньютона
def newton_polynomial(x, coeffs, x_data):
    n = len(coeffs) - 1
    p = coeffs[n]
    for k in range(1, n + 1):
        p = coeffs[n-k] + (x - x_data[n-k]) * p
    return p
# Вычисление значений на промежуточных точках
x_half = [(x_data[i] + x_data[i+1])/2 for i in range(len(x_data)-1)]
f_half_newton = [newton_polynomial(x, newton_coeffs, x_data) for x in x_half]
# Объединяем узлы и промежуточные точки
x_combined = np.concatenate([x_data, x_half])
y_newton = np.concatenate([f_data, f_half_newton])
# Сортируем по X
sort_idx = np.argsort(x_combined)
x_combined = x_combined[sort_idx]
y_newton = y_newton[sort_idx]
```

```
# Создаем итоговую таблицу
newton_df = pd.DataFrame({
    'X': x_combined,
    'F (Ньютон)': y_newton
})
print("\nЗначения многочлена Ньютона:")
print(newton_df.to_string(index=False))
```

Результат

```
Исходные данные:
   X   F
0  0   2
1  1   3
2  2  10
3  3  29
Таблица разделённых разностей (матрица Ньютона):
   f[x0..x0]  f[x0..x1]  f[x0..x2]  f[x0..x3]
0        2.0        1.0        3.0        1.0
1        3.0        7.0        6.0        0.0
2       10.0       19.0        0.0        0.0
3       29.0        0.0        0.0        0.0
Коэффициенты многочлена Ньютона:
[2. 1. 3. 1.]
Значения многочлена Ньютона:
  X  F (Ньютон)
0.0       2.000
0.5       2.125
1.0       3.000
1.5       5.375
2.0      10.000
2.5      17.625
3.0      29.000
```

4 Интерполяция кубическими сплайнами

Необходимо построить интерполяционный многочлен с помощью кубических сплайнов (алгебраических многочленов третьей степени, где сплайн — фрагмент, отрезок чего-либо).

$\boldsymbol{x}$	0	1	2	3
$\boldsymbol{f(x)}$	2	3	10	29

Код

```
import numpy as np
import pandas as pd
def create_spline_matrix(x, f):
    n = len(x) - 1  # количество интервалов
    # инициализируем матрицу 12x12 и вектор правой части
    a = np.zeros((4*n, 4*n))
    b = np.zeros(4*n)
    # уравнение 1: интерполяция в левых концах
    # s_i(x_i) = f_i
    for i in range(n):
        row = i
        a[row, 4*i] = 1  # a_i
        b[row] = f[i]
```

```
    # уравнение 2: интерполяция в правых концах
    # s_i(x_{i+1}) = f_{i+1}
    for i in range(n):
        row = n + i
        h = x[i+1] - x[i]
        a[row, 4*i] = 1          # a_i
        a[row, 4*i + 1] = h      # b_i
        a[row, 4*i + 2] = h**2   # c_i
        a[row, 4*i + 3] = h**3   # d_i
        b[row] = f[i+1]
    # уравнение 3: непрерывность первых производных
    # s'_i(x_{i+1}) = s'_{i+1}(x_{i+1})
    for i in range(n-1):
        row = 2*n + i
        h_i = x[i+1] - x[i]
        h_ip1 = x[i+2] - x[i+1]
        a[row, 4*i + 1] = 1          # b_i
        a[row, 4*i + 2] = 2*h_i      # 2*c_i*h_i
        a[row, 4*i + 3] = 3*h_i**2   # 3*d_i*h_i^2
        a[row, 4*(i+1) + 1] = -1     # -b_{i+1}
        b[row] = 0
    # уравнение 4: непрерывность вторых производных
    # s''_i(x_{i+1}) = s''_{i+1}(x_{i+1})
    for i in range(n-1):
        row = 3*n - 1 + i
        h_i = x[i+1] - x[i]
        a[row, 4*i + 2] = 2         # 2*c_i
        a[row, 4*i + 3] = 6*h_i     # 6*d_i*h_i
        a[row, 4*(i+1) + 2] = -2    # -2*c_{i+1}
        b[row] = 0
    # граничные условия (естественный сплайн)
    # s''_0(x_0) = 0 и s''_{n-1}(x_n) = 0
    row1 = 4*n - 2
    a[row1, 2] = 2  # 2*c_0 = 0
    row2 = 4*n - 1
    h_last = x[n] - x[n-1]
    a[row2, 4*(n-1) + 2] = 2         # 2*c_{n-1}
    a[row2, 4*(n-1) + 3] = 6*h_last  # 6*d_{n-1}*h_{n-1}
    return a, b
def solve_spline_coefficients(x, f):
    a, b = create_spline_matrix(x, f)
    coefficients = np.linalg.solve(a, b)
    return coefficients.reshape(-1, 4)  # преобразуем в матрицу [n x 4]
# данные из условия
x = np.array([0, 1, 2, 3])
f = np.array([2, 3, 10, 29])
# создаем матрицу и решаем систему
a, b = create_spline_matrix(x, f)
coefficients = solve_spline_coefficients(x, f)
# создаем dataframe для красивого отображения матрицы
matrix_df = pd.DataFrame(a,
                         columns=[f'coef_{i}' for i in range(12)],
                         index=[f'eq_{i}' for i in range(12)])
# выводим результаты
print("Матрица системы 12x12:")
print(matrix_df)
print("\nВектор правой части:")
print(pd.Series(b, index=[f'eq_{i}' for i in range(12)]))
print("\nКоэффициенты сплайнов (по строкам: [a_i, b_i, c_i, d_i] для каждого интервала):")
```

```
print(pd.DataFrame(coefficients,
                   columns=['a_i', 'b_i', 'c_i', 'd_i'],
                   index=[f'interval {i}' for i in range(len(x)-1)]))
# проверка размерности
print(f"\nРазмер матрицы: {a.shape}")
print(f"Количество интервалов: {len(x)-1}")
```

Результат

```
Матрица системы 12x12:
     coef_0 coef_1 coef_2 coef_3  coef_4  coef_5 coef_6  coef_7  coef_8  coef_9  coef_10 coef_11
eq_0  1.0    0.0    0.0    0.0     0.0     0.0    0.0     0.0     0.0     0.0     0.0    0.0
eq_1  0.0    0.0    0.0    0.0     1.0     0.0    0.0     0.0     0.0     0.0     0.0    0.0
eq_2  0.0    0.0    0.0    0.0     0.0     0.0    0.0     0.0     1.0     0.0     0.0    0.0
eq_3  1.0    1.0    1.0    1.0     0.0     0.0    0.0     0.0     0.0     0.0     0.0    0.0
eq_4  0.0    0.0    0.0    0.0     1.0     1.0    1.0     1.0     0.0     0.0     0.0    0.0
eq_5  0.0    0.0    0.0    0.0     0.0     0.0    0.0     0.0     1.0     1.0     1.0    1.0
eq_6  0.0    1.0    2.0    3.0     0.0    -1.0    0.0     0.0     0.0     0.0     0.0    0.0
eq_7  0.0    0.0    0.0    0.0     0.0     1.0    2.0     3.0     0.0    -1.0     0.0    0.0
eq_8  0.0    0.0    2.0    6.0     0.0     0.0   -2.0     0.0     0.0     0.0     0.0    0.0
eq_9  0.0    0.0    0.0    0.0     0.0     0.0    2.0     6.0     0.0     0.0    -2.0    0.0
eq_10 0.0    0.0    2.0    0.0     0.0     0.0    0.0     0.0     0.0     0.0     0.0    0.0
eq_11 0.0    0.0    0.0    0.0     0.0     0.0    0.0     0.0     0.0     0.0     2.0    6.0
Вектор правой части:
eq_0      2.0
eq_1      3.0
eq_2     10.0
eq_3      3.0
eq_4     10.0
eq_5     29.0
eq_6      0.0
eq_7      0.0
eq_8      0.0
eq_9      0.0
eq_10     0.0
eq_11     0.0
```

```
Коэффициенты сплайнов (по строкам: [a_i, b_i, c_i, d_i] для каждого интервала):
              a_i   b_i  c_i  d_i
interval 0   2.0   0.2  0.0  0.8
interval 1   3.0   2.6  2.4  2.0
interval 2  10.0  13.4  8.4 -2.8
Размер матрицы: (12, 12)
Количество интервалов: 3
```

5 Метод Гаусса решения СЛАУ

Решить следующую СЛАУ методом Гаусса: метод Гаусса должен решать уравнения вида $Ax = b$, где

$$A = \begin{pmatrix} 5 & 0.05 & 0.05 & 0.05 & 0.05 \\ 0.06 & 6 & 0.06 & 0.06 & 0.06 \\ 0.07 & 0.07 & 7 & 0.07 & 0.07 \\ 0.08 & 0.08 & 0.08 & 8 & 0.08 \\ 0.09 & 0.09 & 0.09 & 0.09 & 9 \end{pmatrix} \quad b = \begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{pmatrix}. \tag{1}$$

Код

```
import numpy as np
A = np.array([[5, 0.05, 0.05, 0.05, 0.05],
              [0.06, 6, 0.06, 0.06, 0.06],
              [0.07, 0.07, 7, 0.07, 0.07],
              [0.08, 0.08, 0.08, 8, 0.08,],
              [0.09, 0.09, 0.09, 0.09, 9]])
print("1) Матрица A:\n", A)
print("Определитель матрицы A = ", np.linalg.det(A))
b = np.zeros(len(A))
print("\nКолонка b:")
for i in range(len(A)):
    b[i] = A[i][i]
    print("[", b[i], "]")
def forward_elimination(A, b):
    n = len(b)
    for k in range(n - 1):
        for i in range(k + 1, n):
            if A[i, k] != 0:
                factor = A[i, k] / A[k, k]
                A[i, k+1:n] = A[i, k+1:n] - factor * A[k, k+1:n]
                A[i, k] = 0  # Explicitly zero out for clarity
                b[i] = b[i] - factor * b[k]
    return A, b
A_tmp, b_tmp = forward_elimination(A, b)
print("Матрица после прямого прохода:")
print(A_tmp)
def backward_substitution(U, b):
    n = len(b)
    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        x[i] = b[i] - np.dot(U[i, i+1:], x[i+1:])
        x[i] /= U[i, i]
    return x
solution = backward_substitution(A_tmp, b_tmp)
print("Решение (A|b) методом Гаусса")
print("Вектор решения после обратного прохода: ")
for i in range(len(solution)):
    print("x", str(i), "=", solution[i])
print(solution, "\n 4) x =")
for i in range(len(solution)):
    print("[", solution[i], "]")
b = np.dot(A, b)
```

Результат

```
1) Матрица A:
 [[5.   0.05 0.05 0.05 0.05]
 [0.06 6.   0.06 0.06 0.06]
 [0.07 0.07 7.   0.07 0.07]
 [0.08 0.08 0.08 8.   0.08]
 [0.09 0.09 0.09 0.09 9.  ]]
Определитель матрицы A =  15105.180138048006
Колонка b:
[ 5.0 ]
[ 6.0 ]
[ 7.0 ]
[ 8.0 ]
[ 9.0 ]
```

```
Матрица после прямого прохода:
[[5.         0.05       0.05       0.05       0.05      ]
 [0.         5.9994     0.0594     0.0594     0.0594    ]
 [0.         0.         6.99861386 0.06861386 0.06861386]
 [0.         0.         0.         7.99764706 0.07764706]
 [0.         0.         0.         0.         8.99650485]]
Решение (A|b) методом Гаусса
Вектор решения после обратного прохода:
x 0 = 0.9615384615384615
x 1 = 0.9615384615384616
x 2 = 0.9615384615384616
x 3 = 0.9615384615384616
x 4 = 0.9615384615384612
[0.96153846 0.96153846 0.96153846 0.96153846 0.96153846]
 4) x =
[ 0.9615384615384615 ]
[ 0.9615384615384616 ]
[ 0.9615384615384616 ]
[ 0.9615384615384616 ]
[ 0.9615384615384612 ]
```

6 Метод прогонки решения СЛАУ (трехдиагональных)

В данном случае решается система линейных уравнений вида $Ax = b$, где

$$A = \begin{pmatrix} 5 & 0.05 & 0.05 & 0.05 & 0.05 \\ 0.06 & 6 & 0.06 & 0.06 & 0.06 \\ 0.07 & 0.07 & 7 & 0.07 & 0.07 \\ 0.08 & 0.08 & 0.08 & 8 & 0.08 \\ 0.09 & 0.09 & 0.09 & 0.09 & 9 \end{pmatrix} \quad b = \begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{pmatrix}. \tag{2}$$

Код

```
import numpy as np
A = np.array([[5, 0.05, 0, 0, 0],
              [0.06, 6, 0.06, 0, 0.],
              [0, 0.07, 7, 0.07, 0],
              [0, 0, 0.08, 8, 0.08],
              [0, 0, 0, 0.09, 9]])
# Вычисляем вектор b
b = np.zeros(len(A))
for i in range(len(A)):
    b[i] = A[i][i]
b = np.dot(A, b.reshape(-1,1))
print("Матрица A:")
print(A)
print("Столбец b:")
print(b)
Q = np.zeros(len(A))
P = np.zeros(len(A) - 1)
P[0] = -A[0][1]/A[0][0]
Q[0] = b[0][0]/A[0][0]
print("Прямая прогонка")
print("Список Pi и Qi")
for i in range(1, len(P)):
    P[i] = (A[i][i+1])/(-A[i][i] - A[i][i-1] * P[i-1])
for i in range(1, len(Q)):
    Q[i] = (A[i][i-1] * Q[i-1] - b[i][0]) / (-A[i][i] - A[i][i-1] * P[i-1])
print(P, Q)
```

```
print("Обратная прогонка")
x = np.zeros(len(A))
x[len(A)-1] = Q[len(A)-1]
print("x 5 =",x[len(A)-1])
for i in range(len(x) - 2, -1, -1):
    x[i] = P[i] * x[i + 1] + Q[i]
    print("x",i + 1,"= ", x[i])
print(x.reshape(-1,1))
```

Результат

```
Матрица A:
[[5.   0.05 0.   0.   0.  ]
 [0.06 6.   0.06 0.   0.  ]
 [0.   0.07 7.   0.07 0.  ]
 [0.   0.   0.08 8.   0.08]
 [0.   0.   0.   0.09 9.  ]]
Столбец b:
[[25.3 ]
 [36.72]
 [49.98]
 [65.28]
 [81.72]]
Прямая прогонка
Список Pi и Qi
[-0.01 -0.010001 -0.010001 -0.010001] [5.06 6.070007 7.080008 8.090009 9.]
Обратная прогонка
x 5 = 9.0
x 4 =  8.000000000000002
x 3 =  6.999999999999999
x 2 =  6.0
x 1 =  5.000000000000001
[[5.]
 [6.]
 [7.]
 [8.]
 [9.]]
```

7 Метод простой итерации

При решении СЛАУ вида $Ax = b$, где A — квадратная матрица, мы можем преобразовать ее к эквивалентному виду:

$$\begin{pmatrix} 0 & -\frac{a_{12}}{a_{11}} & \dots & -\frac{a_{1n}}{a_{11}} \\ -\frac{a_{21}}{a_{22}} & 0 & \dots & -\frac{a_{2n}}{a_{22}} \\ \vdots & \vdots & \ddots & \vdots \\ -\frac{a_{n1}}{a_{nn}} & -\frac{a_{n2}}{a_{nn}} & \dots & 0 \end{pmatrix} x = \begin{pmatrix} \frac{b_1}{a_{11}} \\ \frac{b_2}{a_{22}} \\ \vdots \\ \frac{b_n}{a_{nn}} \end{pmatrix}. \tag{3}$$

Таким образом исходная система допускает представление в виде:

$$\alpha x + \beta = x, \tag{4}$$

а критерий остановки вычислений:

$$\|x^k - x^{k-1}\| < e. \tag{5}$$

Код

```
import numpy as np
A = np.array([[5, 0.05, 0.05, 0.05, 0.05],
              [0.06, 6, 0.06, 0.06, 0.06],
              [0.07, 0.07, 7, 0.07, 0.07],
              [0.08, 0.08, 0.08, 8, 0.08,],
              [0.09, 0.09, 0.09, 0.09, 9]])
print("Матрица A:")
print(A)
print("Определитель матрицы A:", np.linalg.det(A))
# Вычисляем вектор b
b = np.zeros(len(A))
for i in range(len(A)):
    b[i] = A[i][i]
b = np.dot(A, b.reshape(-1,1))
print("Столбец b:")
print(b)
alpha = A.copy()
beta = b.copy()
for i in range(len(A)):
    for j in range(len(A)):
        if i == j:
            alpha[i][j] = 0
        else:
            alpha[i][j] = - A[i][j]/A[i][i]
print("Матрица alpha:")
print(alpha)
for i in range(len(A)):
    beta[i][0] = b[i][0]/A[i][i]
print("Столбец beta:")
print(beta)
epsilon = 10**-9
print("Считаем до точности epsilon=", epsilon)
xk = np.zeros(len(A)).reshape(-1,1)
print("x^(0) = ", xk)
def normStop (xk, xkp1, epsilon):
    return max(np.abs(np.add(xk, -xkp1))) < epsilon
for i in range(17):
    xkp1 = np.add(np.dot(alpha, xk), beta)
    print(f"x^({i+1})=", xkp1)
    if normStop(xk, xkp1, epsilon):
        break
    xk = xkp1
```

Результат

```
Матрица A:
[[5.   0.05 0.05 0.05 0.05]
 [0.06 6.   0.06 0.06 0.06]
 [0.07 0.07 7.   0.07 0.07]
 [0.08 0.08 0.08 8.   0.08]
 [0.09 0.09 0.09 0.09 9.  ]]
Определитель матрицы A: 15105.180138048006
Столбец b:
[[26.5 ]
 [37.74]
 [50.96]
 [66.16]
 [83.34]]
```

```
Матрица alpha:
[[ 0.   -0.01 -0.01 -0.01 -0.01]
 [-0.01  0.   -0.01 -0.01 -0.01]
 [-0.01 -0.01  0.   -0.01 -0.01]
 [-0.01 -0.01 -0.01  0.   -0.01]
 [-0.01 -0.01 -0.01 -0.01  0.  ]]
Столбец beta:
[[5.3 ]
 [6.29]
 [7.28]
 [8.27]
 [9.26]]
Считаем до точности epsilon= 1e-09
x^(0) =  [[0.]
 [0.]
 [0.]
 [0.]
 [0.]]
x^(1)= [[5.3 ]
 [6.29]
 [7.28]
 [8.27]
 [9.26]]
x^(2)= [[4.989 ]
 [5.9889]
 [6.9888]
 [7.9887]
 [8.9886]]
x^(3)= [[5.00045 ]
 [6.000449]
 [7.000448]
 [8.000447]
 [9.000446]]
x^(4)= [[4.9999821 ]
 [5.99998209]
 [6.99998208]
 [7.99998207]
 [8.99998206]]
x^(5)= [[5.00000072]
 [6.00000072]
 [7.00000072]
 [8.00000072]
 [9.00000072]]
x^(6)= [[4.99999997]
 [5.99999997]
 [6.99999997]
 [7.99999997]
 [8.99999997]]
x^(7)= [[5.]
 [6.]
 [7.]
 [8.]
 [9.]]
x^(8)= [[5.]
 [6.]
 [7.]
 [8.]
 [9.]]
x^(9)= [[5.]
 [6.]
 [7.]
 [8.]
 [9.]]
```

8 Задача Коши методами Эйлера

Решить задачу Коши: а) методом Эйлера; б) усовершенствованным методом Эйлера:

$$\begin{cases} y'(x) = 2Vx + Vx^2 - y(x) \\ y(1) = V \end{cases} \tag{6}$$

где $y_{\text{точн}}(x) = Vx^2$, V — номер варианта.

Код

```
import numpy as np
import pandas as pd
def f(x, y, V=5):
    return 2 * V * x + V * x**2 - y
# Метод Эйлера
def euler_method(x0, y0, h, n, V):
    x = np.zeros(n + 1)
    y = np.zeros(n + 1)
    x[0] = x0
    y[0] = y0
    for i in range(n):
        x[i + 1] = x[i] + h
        y[i + 1] = y[i] + h * f(x[i], y[i], V)
    return x, y
# Усовершенствованный метод Эйлера
def improved_euler_method(x0, y0, h, n, V):
    x = np.zeros(n + 1)
    y = np.zeros(n + 1)
    x[0] = x0
    y[0] = y0
    for i in range(n):
        x[i + 1] = x[i] + h
        y_half = y[i] + (h / 2) * f(x[i], y[i], V)
        x_half = x[i] + h / 2
        y[i + 1] = y[i] + h * f(x_half, y_half, V)
    return x, y
def exact_solution(x, V=5):
    return V * x**2
# Параметры
x0 = 1
V = y0 = 5
h = 0.001 # шаг
n = 10
# Вычисление решений
x_euler, y_euler = euler_method(x0, y0, h, n, V)
x_improved, y_improved = improved_euler_method(x0, y0, h, n, V)
y_exact = exact_solution(x_euler)
# Вычисление погрешностей
error_euler = np.abs(y_euler - y_exact)
error_improved = np.abs(y_improved - y_exact)
print("\nМетод Эйлера: ")
print("-" * 130)
print("x:      ", " ".join(f"{x:>10.7f}" for x in x_euler))
print("y_M:    ", " ".join(f"{y:>10.7f}" for y in y_euler))
print("y_T:    ", " ".join(f"{y:>10.7f}" for y in y_exact))
print("Погрешн:", " ".join(f"{e:>10.7f}" for e in error_euler))
print("-" * 130)
```

```
print("\nУсовершенствованный метод Эйлера: ")
print("-" * 130)
print("x:      ", " ".join(f"{x:>10.7f}" for x in x_improved))
print("y_M:    ", " ".join(f"{y:>10.7f}" for y in y_improved))
print("y_T:    ", " ".join(f"{y:>10.7f}" for y in y_exact))
print("Погрешн:", " ".join(f"{e:>10.7f}" for e in error_improved))
print("-" * 130)
```

Результат

Метод Эйлера:

```
x:       1.0000000  1.0010000  1.0020000  1.0030000  1.0040000  1.0050000  1.0060000  1.0070000  1.0080000  1.0090000  1.0100000
y_M:     5.0000000  5.0100000  5.0200100  5.0300300  5.0400600  5.0501000  5.0601501  5.0702101  5.0802801  5.0903602  5.1004502
y_T:     5.0000000  5.0100050  5.0200200  5.0300450  5.0400800  5.0501250  5.0601800  5.0702450  5.0803200  5.0904050  5.1005000
Погрешн: 0.0000000  0.0000050  0.0000100  0.0000150  0.0000200  0.0000250  0.0000299  0.0000349  0.0000399  0.0000448  0.0000498
```

Усовершенствованный метод Эйлера:

```
x:       1.0000000  1.0010000  1.0020000  1.0030000  1.0040000  1.0050000  1.0060000  1.0070000  1.0080000  1.0090000  1.0100000
y_M:     5.0000000  5.0100050  5.0200200  5.0300450  5.0400800  5.0501250  5.0601800  5.0702450  5.0803200  5.0904050  5.1005000
y_T:     5.0000000  5.0100050  5.0200200  5.0300450  5.0400800  5.0501250  5.0601800  5.0702450  5.0803200  5.0904050  5.1005000
Погрешн: 0.0000000  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000
```

9 Решить краевую задачу разностным методом.

$$\begin{cases} y'' + x^2 y' + xy = 4Vx^4 - 3VTx^3 + 6Vx - 2VT \\ y'(0) = y(T) = 0 \\ V = T = 5 \end{cases} \tag{7}$$

Код

```
V = 5 # номер варанта
def derivative(x):
    return -(4 * V * x**4 - 3 * V**2 * x**3 + 6 * V * x - 2 * V**2)
def Y(x):
    return V * x**2 * (x - V)
def p(x):
    return -x**2
def q(x):
    return -x
def main():
    n = 10
    x0 = 0
    h = V / n
    x = [x0 + i * h for i in range(n + 1)]
    exact = [Y(xi) for xi in x]
    f = [0.0] * (n + 1)
    s = [0.0] * (n + 1)
    t = [0.0] * (n + 1)
    r = [0.0] * (n + 1)
    f1 = [0.0] * (n + 1)
    s1 = [0.0] * (n + 1)
    y = [0.0] * (n + 1)
    e = [0.0] * (n + 1)
    for i in range(1, n):
        f[i] = 0.5 * (1 + 0.5 * h * p(x[i]))
        s[i] = 0.5 * (1 - 0.5 * h * p(x[i]))
```

```
        t[i] = 1 + 0.5 * h**2 * q(x[i])
        r[i] = 0.5 * h**2 * derivative(x[i])
    f1[1] = 0.0
    s1[1] = 0.0
    for j in range(1, n):
        denom = t[j] - f[j] * f1[j]
        f1[j + 1] = s[j] / denom
        s1[j + 1] = (r[j] + f[j] * s1[j]) / denom
    # Граничное условие y[n] = 0 в оригинале неявно, принимаем y[n]=0
    y[n] = 0.0
    for j in range(n - 1, 0, -1):
        y[j] = f1[j + 1] * y[j + 1] + s1[j + 1]
    max_e = 0.0
    max_e_index = 0
    for i in range(n + 1):
        e[i] = abs(y[i] - exact[i])
        if (e[i] > max_e):
            max_e = e[i]
            max_e_index = i
    print("x\t y\t\t exact\t e")
    for i in range(n + 1):
        print(f"{x[i]:.2f}\t {y[i]:.2f} \t{exact[i]:.2f}\t {e[i]:.8f}")
    print("Максимальный e: ", max_e, "Номер максимального e: ", max_e_index)
if __name__ == "__main__":
    main()
```

Результат

```
x          y           exact          e
0.00      0.00         -0.00       0.00000000
0.50     -0.09         -5.62       5.53017248
1.00     -10.31        -20.00      9.68559849
1.50     -29.38        -39.38      9.99173440
2.00     -52.77        -60.00      7.22943620
2.50     -73.33        -78.12      4.79707715
3.00     -86.60        -90.00      3.39879569
3.50     -89.59        -91.88      2.28687417
4.00     -78.58        -80.00      1.41652296
4.50     -49.97        -50.62      0.65542909
5.00      0.00          0.00       0.00000000
Максимальный e: 9.991734404070492
Номер максимального e: 3
```

10 Краевая задача методом неопределенных коэффициентов

$$\begin{cases} y'' + x^2 y' + xy = 4Vx^4 - 3VTx^3 + 6Vx - 2VT \\ y'(0) = y(T) = 0 \\ V = T = 5 \end{cases} \tag{8}$$

Код

```
import numpy as np
V = 5 # номер варианта
h = 0.1 # шаг
n = int(V / h)
maxx = V
# y точный
def ytoch(x):
    return V * x * x * (x - maxx)
```

```
def f(x):
    return 4 * V * (x ** 4) - 3 * V * V * (x ** 3) + 6 * V * x - 2 * V * V
def p(x):
    return x * x
def q(x):
    return x
def phi_k(x, k):
    return x ** k * (x - V)
def dphi_k(x, k):
    return (k + 1) * x ** k - V * k * x ** (k - 1)
def ddphi_k(x, k):
    return k * (k + 1) * x ** (k - 1) - V * k * (k - 1) * x ** (k - 2)
# Создаем сетку
xk = [x * h for x in range(n + 1)]
ykToch = [ytoch(x) for x in xk]
print("Проверка точного решения в ключевых точках:")
test_points = [0, 1, 2, 3, 4, 5]
for x in test_points:
    print(f"x = {x}: y_toch = {ytoch(x):.2f}")
# Инициализация матрицы A и вектора b
A = np.zeros((n, n))
b = np.zeros(n)
# Заполнение матрицы A и вектора b для ВНУТРЕННИХ точек (i=1..n-1)
for i in range(1, n + 1):
    b[i - 1] = f(xk[i])
    for k in range(1, n + 1):
        A[i - 1][k - 1] = ddphi_k(xk[i], k) + p(xk[i]) * dphi_k(xk[i], k) + q(xk[i]) *
        phi_k(xk[i], k)
print(f"\nРазмерность матрицы A: {A.shape}")
print(f"Размерность вектора b: {b.shape}")
# Метод Гаусса для решения СЛАУ A * c = b
def gauss_elimination(A, b):
    n = len(b)
    # Прямой ход метода Гаусса
    for i in range(n):
        # Поиск максимального элемента в столбце для улучшения устойчивости
        max_row = i
        max_val = abs(A[i, i])
        for j in range(i + 1, n):
            if abs(A[j, i]) > max_val:
                max_val = abs(A[j, i])
                max_row = j
        # Обмен строк, если необходимо
        if max_row != i:
            A[[i, max_row]] = A[[max_row, i]]
            b[i], b[max_row] = b[max_row], b[i]
        # Проверка на нулевой диагональный элемент
        if abs(A[i, i]) < 1e-12:
            print(f"Внимание: малый диагональный элемент A[{i},{i}] = {A[i, i]}")
            A[i, i] = 1e-12
        # Нормировка строки
        pivot = A[i, i]
        for j in range(i, n):
            A[i, j] /= pivot
        b[i] /= pivot
        # Исключение переменной из нижележащих строк
        for j in range(i + 1, n):
            factor = A[j, i]
            for k in range(i, n):
                A[j, k] -= factor * A[i, k]
            b[j] -= factor * b[i]
```

```
    # Обратный ход метода Гаусса
    c = np.zeros(n)
    for i in range(n - 1, -1, -1):
        c[i] = b[i]
        for j in range(i + 1, n):
            c[i] -= A[i, j] * c[j]
    return c
# Решение системы
print("\nРешение СЛАУ методом Гаусса")
c = gauss_elimination(A.copy(), b.copy())
print("\nПервые 10 коэффициентов a_k:")
for i in range(min(10, len(c))):
    print(f"a_{i+1} = {c[i]:.6e}")
# Построение приближенного решения
def y_approx(x, c):
    result = 0
    for k in range(1, len(c) + 1):
        result += c[k - 1] * phi_k(x, k)
    return result
# Сравнение точного и приближенного решений
print("\nСравнение решений в некоторых точках:")
print("x\t\tТочное y\tПриближенное y\t\tОтносительная погрешность")
for x in range(V + 1):
    y_exact = ytoch(x)
    y_appr = y_approx(x, c)
    rel_error = abs((y_appr - y_exact) / y_exact) if abs(y_exact) > 1e-12
    else abs(y_appr)
    print(f"{x}\t\t{y_exact:.4f}\t\t{y_appr:.4f}\t\t{rel_error:.2e}")
```

Результат

```
Проверка точного решения в ключевых точках:
x = 0: y_toch = 0.00
x = 1: y_toch = -20.00
x = 2: y_toch = -60.00
x = 3: y_toch = -90.00
x = 4: y_toch = -80.00
x = 5: y_toch = 0.00
Размерность матрицы A: (50, 50)
Размерность вектора b: (50,)
Решение СЛАУ методом Гаусса
Первые 10 коэффициентов a_k:
a_1 = 1.507017e-12
a_2 = 5.000000e+00
a_3 = 2.660881e-10
a_4 = -1.744293e-09
a_5 = 7.754601e-09
a_6 = -2.451899e-08
a_7 = 5.696992e-08
a_8 = -9.949613e-08
a_9 = 1.324989e-07
a_10 = -1.354025e-07
Сравнение решений в некоторых точках:
x               Точное y        Приближенное y          Относительная погрешность
0                0.0000             0.0000                    0.00e+00
1              -20.0000           -20.0000                    2.84e-14
2              -60.0000           -60.0000                    3.67e-15
3              -90.0000           -90.0000                    4.74e-16
4              -80.0000           -80.0000                    1.78e-16
5                0.0000             0.0000                    0.00e+00
```

11 Метод неопределенных коэффициентов

Решите следующее интегральное уравнение:

$$y(x) + 1 * \int_0^1 (xt + x^2t^2 + x^3t^3)y(t)dt = V\left(\frac{4}{3}x + \frac{1}{4}x^2 + \frac{1}{5}x^3\right) \quad (9)$$

Код

```
import numpy as np
from scipy import integrate
def rhs_func(x, variant): # правая часть уравнения
    return variant * (4 / 3 * x + 1 / 4 * x**2 + 1 / 5 * x**3)
def build_alpha(size):  # матрица коэффициентов a_ij
    alpha = np.zeros((size, size))
    for i in range(size):
        for j in range(size):
            def integrand(t):
                if i == 0:
                    ai = t
                elif i == 1:
                    ai = t**2
                else:
                    ai = t**3
                if j == 0:
                    bj = t
                elif j == 1:
                    bj = t**2
                else:
                    bj = t**3
                return ai * bj
            alpha[i, j], _ = integrate.quad(integrand, 0, 1)
    return alpha
def build_gamma(size, variant): # вектор gamma_i
    gamma = np.zeros(size)
    for i in range(size):
        def integrand(t):
            if i == 0:
                bi = t
            elif i == 1:
                bi = t**2
            else:
                bi = t**3
            return rhs_func(t, variant) * bi
        gamma[i], _ = integrate.quad(integrand, 0, 1)
    return gamma
def gauss_method(A, b):# метод Гаусса
    A = A.astype(float)
    b = b.astype(float)
    n = len(b)
    for i in range(n): # прямой ход
        max_row = i + np.argmax(abs(A[i:, i])) # поиск максимального элемента для выбора
        главного элемента
        if A[max_row, i] == 0:
            raise ValueError("Система не имеет единственного решения (нулевой ведущий
            элемент).")
        if max_row != i: # Перестановка строк
            A[[i, max_row]] = A[[max_row, i]]
            b[[i, max_row]] = b[[max_row, i]]
        pivot = A[i, i] # нормализация ведущей строки
```

```
        A[i] = A[i] / pivot
        b[i] = b[i] / pivot
        for j in range(i + 1, n): # обнуление элементов под ведущим
            factor = A[j, i]
            A[j] = A[j] - factor * A[i]
            b[j] = b[j] - factor * b[i]
    x = np.zeros(n) # Обратный ход
    for i in range(n - 1, -1, -1):
        x[i] = b[i] - np.dot(A[i, i + 1:], x[i + 1:])
    return x
def fredholm_solver(variant, rank=3):
    alpha = build_alpha(rank)
    gamma = build_gamma(rank, variant)
    system_matrix = np.eye(rank) + alpha
    coeffs = gauss_method(system_matrix, gamma)
    step = 0.1
    x_vals = np.arange(0, 1 + step, step)
    y_num = np.zeros_like(x_vals)
    for i, x in enumerate(x_vals):
        y_val = rhs_func(x, variant)
        for j in range(rank):
            if j == 0:
                aj = x
            elif j == 1:
                aj = x**2
            else:
                aj = x**3
            y_val -= coeffs[j] * aj
        y_num[i] = y_val
    y_true = variant * x_vals
    err = np.abs(y_num - y_true)
    return x_vals, y_num, y_true, err
def main():
    V = 5
    x, y_calc, y_exact, error = fredholm_solver(V)
    print("\nРешение интегрального уравнения Фредгольма (вырожденное ядро)")
    print("x:           ", " ".join(f"{float(xi):7.3f}" for xi in x))
    print("y_мет:       ", " ".join(f"{float(yi):7.3f}" for yi in y_calc))
    print("y_точн:      ", " ".join(f"{float(yi):7.3f}" for yi in y_exact))
    print("погрешн: ", " ".join(f"{ei:7.3f}" for ei in error))
if __name__ == "__main__":
    main()
```

Результат

```
Решение интегрального уравнения Фредгольма (вырожденное ядро)
x:       0.000   0.100   0.200   0.300   0.400   0.500   0.600   0.700   0.800   0.900   1.000
y_мет:   0.000   0.500   1.000   1.500   2.000   2.500   3.000   3.500   4.000   4.500   5.000
y_точн:  0.000   0.500   1.000   1.500   2.000   2.500   3.000   3.500   4.000   4.500   5.000
погрешн: 0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
```

12 Метод квадратур

Решите следующее интегральное уравнение:

$$y(x) + 1 * \int_0^1 (xt + x^2t^2 + x^3t^3)y(t)dt = V\left(\frac{4}{3}x + \frac{1}{4}x^2 + \frac{1}{5}x^3\right) \qquad (10)$$

Код

```
import numpy as np
import pandas as pd
from scipy.integrate import quad
from scipy.linalg import solve
def solve_fredholm_degenerate_correct(V):
    a, b = 0, 1
    n = 3
    lam = 1
    def y_exact(x):
        return V * x
    def f(x):
        return V * (4 / 3 * x + 1 / 4 * x**2 + 1 / 5 * x**3)
    a_funcs = [lambda x: x, lambda x: x**2, lambda x: x**3]
    b_funcs = [lambda t: t, lambda t: t**2, lambda t: t**3]
    alpha = np.zeros((n, n))
    gamma = np.zeros(n)
    for i in range(n):
        for k in range(n):
            alpha[i, k], _ = quad(lambda x: a_funcs[i](x) * b_funcs[k](x), a, b)
        gamma[i], _ = quad(lambda x: f(x) * b_funcs[i](x), a, b)
    A = np.eye(n) + lam * alpha.T
    q = solve(A, gamma)
    def y_numerical(x):
        result = f(x)
        for i in range(n):
            result -= lam * q[i] * a_funcs[i](x)
        return result
    x_test = np.linspace(a, b, 10)
    y_num_vals = [y_numerical(x) for x in x_test]
    y_ex_vals = [y_exact(x) for x in x_test]
    errors = [abs(y_num_vals[i] - y_ex_vals[i]) for i in range(len(x_test))]
    df = pd.DataFrame(
        {"x": x_test, "y_метода": y_num_vals, "y_точн": y_ex_vals, "eps": errors}
    )
    print(df.to_string(index=False))
    print("\n")
    return y_numerical, y_exact
if __name__ == "__main__":
    V = 5  # номер варанта
    solve_fredholm_degenerate_correct(V)
```

Результат

```
        x       y_метода    y_точн        eps
 0.000000    0.000000   0.000000   0.000000e+00
 0.111111    0.555556   0.555556   1.110223e-16
 0.222222    1.111111   1.111111   0.000000e+00
 0.333333    1.666667   1.666667   2.220446e-16
 0.444444    2.222222   2.222222   0.000000e+00
 0.555556    2.777778   2.777778   4.440892e-16
 0.666667    3.333333   3.333333   1.332268e-15
 0.777778    3.888889   3.888889   0.000000e+00
 0.888889    4.444444   4.444444   0.000000e+00
 1.000000    5.000000   5.000000   0.000000e+00
```