Московский авиационный институт

(Национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторные работы

по курсу «Численные методы»

Вариант 23

Выполнил: Фомин И. Д.

Группа: М8О-309Б-22

Проверил: Ревизников Д. Л.

Дата:

Оценка:

Москва, 2022

Оглавление

Лабораторная работа №1

Задание 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.

$$23.\ \begin{cases} 2\cdot x_1 - 7\cdot x_2 + 8\cdot x_3 - 4\cdot x_4 = 57 \\ -x_2 + 4\cdot x_3 - x_4 = 24 \\ 3\cdot x_1 - 4\cdot x_2 + 2\cdot x_3 - x_4 = 28 \\ -9\cdot x_1 + x_2 - 4\cdot x_3 + 6\cdot x_4 = 12 \end{cases}$$

Теоретические сведения

LU-разложение матрицы представляет собой разложение матрицы A в произведение нижней и верхней треугольных матриц, т. е. $A = LU$, где L — нижняя треугольная матрица, U — верхняя треугольная матрица.

LU-разложение может быть построено с использованием метода Гаусса. Рассмотрим k-й шаг метода Гаусса, на котором осуществляется обнуление поддиагональных элементов k-го столбца матрицы $A^{(k-1)}$. С этой целью используется операция:

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \mu_i^{(k)} a_{kj}^{(k-1)}, \text{где } \mu_i^{(k)} = \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}};\ i = \underline{k+1, n};\ j = \underline{k, n}$$

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

$$m_{ij}^k = \{1,\ \ i = j\ 0,\ \ i \neq j,\ \ j \neq k\ -\mu_{k+1}^{(k)},\ \ \ i \neq j,\ \ j = k$$

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

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

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

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

В дальнейшем LU-разложение может быть эффективно использовано при решении систем линейных алгебраических уравнений вида $Ax = b$. Действительно, подставляя LU-разложение в СЛАУ, получим $LUx = b$, или $Ux = L^{-1}b$. Т.е. процесс решения СЛАУ сводится к двум простым этапам:

1) На первом этапе решается СЛАУ $Ly = b$. Поскольку матрица системы – нижняя треугольная, решение можно записать в явном виде:

$$y_1 = b_1, \;\; y_i = b_i - \sum_{j=1}^{i-1} l_{ij}y_j, \;\; i = \underline{2,n}$$

2) На втором этапе решается СЛАУ $Ux = y$ с верхней треугольной матрицей. Здесь, как и на предыдущем этапе, решение представляется в явном виде:

$$x_1 = \frac{y_n}{u_{nn}}, \;\; x_i = \frac{1}{u_{ii}}(y_i - \sum_{j=i+1}^{n} u_{ij}y_j, \;\; i = \underline{n-1,1}$$

Зная LU-разложение легко также получить определитель матрицы A и обратную ей матрицу. Определитель находится как произведение элементов на главных диагоналях матриц L и U:

$$detA = \prod_{i=1}^{n} l_{ii} \cdot u_{ii}$$

Обратную матрицу можно найти из отношения $AA^{-1} = LUA^{-1} = E$. Это уравнение также можно решить методом LU-разложения.

Код программы

```
import numpy as np
```

```
from scipy.linalg import lu as scipy_lu



def lu(A):
  """
  Функция LU-разложения

  :param A: входная квадратная матрица (A из системы Ax = b)
  :returns: P, L, U, где\n
    P - перестановочная матрица (P⁻¹ = Pᵗ),\n
    L - нижняя треугольная матрица с единицами на диагонали,\n
    U - верхняя треугольная матрица\n
    swap_count - число перестановок, надо для детерминанта
  """
  n = len(A)
  P = np.eye(n)
  L = np.eye(n)
  U = A.copy()
  swap_count = 0

  for i in range(n):
    max_row = np.argmax(np.abs(U[i:, i])) + i
    if max_row != i:
      U[[i, max_row]] = U[[max_row, i]]
      P[[i, max_row]] = P[[max_row, i]]
      L[[i, max_row], :i] = L[[max_row, i], :i]

    swap_count += 1

    for j in range(i + 1, n):
      L[j, i] = U[j, i] / U[i, i]
      U[j, i:] -= L[j, i] * U[i, i:]

  return P, L, U, swap_count
```

```
def solve(P, L, U, b):
  """
  Решает систему вида Ax = b, используя разложение PA = LU\n
  \n
  PA = LU, Ax = b =>\n
  PAx = Pb =>\n
  LUx = Pb\n
  Пусть y = Ux, тогда нам надо решить:\n
  1) Ly = Pb, 2) Ux = y

  :param P: перестановочная матрица (P⁻¹ = Pᵗ)
  :param L: нижняя треугольная матрица с единицами на диагонали
  :param U: верхняя треугольная матрица
  :param b: числовые значения в Ax = b
  """
  n = len(b)
  b_permuted = P @ b
  x = np.zeros(n)
  y = np.zeros(n)

  for i in range(n):
    y[i] = b_permuted[i] - np.dot(L[i, :i], y[:i])

  for i in range(n - 1, -1, -1):
    x[i] = (y[i] - np.dot(U[i, i + 1:], x[i + 1:])) / U[i, i]

  return x



def lu_det(A, swap_count):
  """
    Находит детерминант A
  """
  det_A = np.prod(np.diag(A))
  sign = (-1) ** swap_count
  return sign * det_A
```

```
def inv(A):
  """
    Вычисляет обратную матрицу через решение A X = I.
  """
  P, L, U, _ = lu(A)
  n = len(A)
  inv_A = np.zeros((n, n))
  for i in range(n):
    e = np.zeros(n)
    e[i] = 1
    inv_A[:, i] = solve(P, L, U, e)
  return inv_A


A = np.array([
  [2, -7, 8, -4],
  [0, -1, 4, -1],
  [3, -4, 2, -1],
  [-9, 1, -4, 6]],
  dtype=float
)

b = np.array([57, 24, 28, 12], dtype=float)
P, L, U, swap_count = lu(A)
P_sc, L_sc, U_sc = scipy_lu(A)

print("P = \n", P)
print("L = \n", L)
print("U = \n", U)

print("x = ", solve(P, L, U, b))
print("det A = ", lu_det(A, swap_count))
print("inv A = \n", inv(A))
```

```
print("Решение совпадает с библиотечным:", np.allclose(solve(P, L, U, b), np.linalg.solve(A, b)))
```

Вывод программы

```
A:  [[ 2. -7.  8. -4.]
 [ 0. -1.  4. -1.]
 [ 3. -4.  2. -1.]
 [-9.  1. -4.  6.]]
b =  [57. 24. 28. 12.]
x =  [ 1. -5.  7.  9.]
det A =  -24.0
inv A =
 [[-0.2173913   0.22360248  0.31055901 -0.05590062]
 [-0.13043478  0.2484472  -0.09937888 -0.0621118 ]
 [-0.13043478  0.46273292  0.11490683  0.00931677]
 [-0.39130435  0.60248447  0.55900621  0.09937888]]
Решение совпадает с библиотечным: True
```

Задание 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.

$$23.\ \begin{cases} 7\cdot x_1 - 5\cdot x_2 = 38 \\ -6\cdot x_1 + 19\cdot x_2 - 9\cdot x_3 = 14 \\ 6\cdot x_2 - 18\cdot x_3 + 7\cdot x_4 = -45 \\ -7\cdot x_3 - 11\cdot x_4 - 2\cdot x_5 = 30 \\ 5\cdot x_4 - 7\cdot x_5 = 48 \end{cases}$$

Теоретические сведения

Метод прогонки является частным случаем метода Гаусса. Он применяется для решения СЛАУ с трехдиагональными матрицами. Рассмотрим СЛАУ:

$$\{b_1x_1 + c_1x_2 = d_1\ a_2x_1 + b_2x_2 + c_2x_3 = d_2\ a_3x_2 + b_3x_3 + c_3x_4 = d_3\ \dots\ a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n = d_{n-1}\ a_nx_{n-1} + b_nx_n = d_n$$

При этом будем полагать, что $a_1 = 0$ и $c_n = 0$. Решение системы можно искать в виде:

$$x_i = P_i x_{i+1} + Q_i, \;\; i = \underline{1, n}$$

Здесь P_i и Q_i – прогоночные коэффициенты, определяемые по формулам:

$$P_1 = \frac{-c_1}{b_1}; \;\; Q_1 = \frac{d_1}{b_1}$$

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

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

$$x_n = P_n x_{n+1} + Q_n = 0 \cdot x_{n+1} + Q_n = Q_n$$

$$x_{n-1} = P_{n-1} x_n + Q_{n-1}$$

$$x_{n-2} = P_{n-2} x_{n-1} + Q_{n-2}$$

$$\dots \dots \dots \dots \dots \dots \dots$$

$$x_1 = P_1 x_2 + Q_1$$

Достаточным условием корректности метода прогонки и устойчивости его к погрешностям вычислений является условие преобладания диагональных коэффициентов:

$$|b_i| \geq |a_i| + |c_i|$$

Код программы

```
import numpy as np



def progonka(a, b, c, d):
  """
  Решает СЛАУ с трехдиагональной матрицей методом прогонки

  Параметры:
  :param a: Элементы под главной диагональю (коэффициенты при x[i-1] в i-м уравнении, i = 1..n-1).
  :param b: Элементы на главной диагонали (коэффициенты при x[i] в i-м уравнении).
```

```
  :param c: Элементы над главной диагональю (коэффициенты при x[i+1] в i-м
уравнении, i = 0..n-2).
  :param d: Вектор правых частей системы.

  поскольку матрица трехдиагональная, то мы можем выразить в каждой строке i-
й элемент через i+1-й
  """
  n = len(d)
  P = np.zeros(n-1)
  Q = np.zeros(n)

  P[0] = -c[0] / b[0]
  Q[0] = d[0] / b[0]

  for i in range(1, n-1):
    P[i] = -c[i] / (b[i] + a[i - 1] * P[i - 1])
    Q[i] = (d[i] - a[i - 1] * Q[i - 1]) / (b[i] + a[i - 1] * P[i - 1])

  Q[n - 1] = (d[n - 1] - a[n - 2] * Q[n - 2]) / (b[n - 1] + a[n - 2] * P[n -
2])

  x = np.zeros(n)
  x[n - 1] = Q[n - 1]
  for i in range(n - 2, -1, -1):
    x[i] = Q[i] + P[i] * x[i + 1]

  return x

#  7  -5  0  0   0 | 38
# -6  19 -9  0   0 | 14
#  0   6 -18 7   0 | -45
#  0   0 -7 -11 -2 | 30
#  0   0  0  5  -7 | 48
a = np.array([-6, 6, -7, 5])
b = np.array([7, 19, -18, -11, -7])
c = np.array([-5, -9, 7, -2])
d = np.array([38, 13, -45, 30, 48])
x = progonka(a, b, c, d)
```

```
print("Решение системы:", x)
```

Вывод программы

```
Решение системы: [ 8.94170672  4.91838941  2.97768427 -2.98743137 -8.99102241]
```

Задание 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.

$$23.\ \begin{cases} -24\cdot x_1 - 6\cdot x_2 + 4\cdot x_3 + 7\cdot x_4 = 130 \\ -8\cdot x_1 + 21\cdot x_2 + 4\cdot x_3 - 2\cdot x_4 = 139 \\ 6\cdot x_1 + 6\cdot x_2 + 16\cdot x_3 = -84 \\ -7\cdot x_1 - 7\cdot x_2 + 5\cdot x_3 + 24\cdot x_4 = -165 \end{cases}$$

Теоретические сведения

Рассмотрим СЛАУ

$$\{a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2\ \ldots\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n$$

с невырожденной матрицей.

Приведем СЛАУ к эквивалентному виду

$$\{x_1 = \beta_1 + \alpha_{11}x_1 + \alpha_{12}x_2 + \cdots + \alpha_{1n}x_n\ x_2 = \beta_2 + \alpha_{21}x_1 + \alpha_{22}x_2 + \cdots + \alpha_{2n}x_n\ \ldots\ x_n = \beta_n + \alpha_{n1}x_1 + \alpha_{n2}x_2 + \cdots + \alpha_{nn}x_n$$

или в векторно-матричной форме $x = \beta + \alpha x$.

Такое приведение может быть выполнено различными способами. Одним из наиболее распространенных является следующий. Разрешим систему относительно неизвестных при ненулевых диагональных элементах $a_{ii} \neq 0, i = \underline{1,n}$ (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:

$$\beta_i = \frac{b_i}{a_{ii}}$$

$$\alpha_{ij} = -\frac{a_{ij}}{a_{ii}}, \;\; i \neq j; \; \alpha_{ij} = 0, \;\; i = j$$

В качестве нулевого приближения $x^{(0)}$ вектора неизвестных примем вектор правых частей $x^{(0)} = \beta$. Тогда **метод простых итераций** примет вид:

$$\{x^{(0)} = \beta \; x^{(1)} = \beta + \alpha x^{(0)} \; x^{(2)} = \beta + \alpha x^{(1)} \; \dots \; x^{(k)} = \beta + \alpha x^{(k-1)}$$

Достаточным условием сходимости является диагональное преобладание матрицы A по строкам или по столбцам:

$$|a_{ii}| > \sum_{j=1, i \neq j}^{n} |a_{ij}|$$

Критерием окончания итерационного процесса может служить неравенство $\left\| x^{(k)} - x^{(k-1)} \right\| \leq \varepsilon$.

Метод простых итераций довольно медленно сходится. Для его ускорения существует **метод Зейделя**, заключающийся в том, что при вычислении компонента x_i^{k+1} вектора неизвестных на (k+1)-й итерации используются $x_1^{k+1}, x_2^{k+1}, \dots, x_{i-1}^{k+1}$, уже вычисленные на (k+1)-й итерации. Тогда метод Зейделя для известного вектора на k-ой итерации имеет вид:

$$\begin{aligned}\{x_1^{k+1} &= \beta_1 + \alpha_{11}x_1^k + \alpha_{12}x_2^k + \cdots + \alpha_{1n}x_n^k \; x_2^{k+1} \\ &= \beta_2 + \alpha_{21}x_1^k + \alpha_{22}x_2^k + \cdots + \alpha_{2n}x_n^k \; x_3^{k+1} \\ &= \beta_3 + \alpha_{31}x_1^k + \alpha_{32}x_2^k + \cdots + \alpha_{3n}x_n^k \; \dots \; x_n^{k+1} \\ &= \beta_n + \alpha_{n1}x_1^{k+1} + \alpha_{n2}x_2^{k+1} + \cdots + \alpha_{nn-1}x_n^{k+1} + \alpha_{nn}x_n^k\end{aligned}$$

Код программы

```
def matrix_inf_norm(alpha):
  """
  Вычисляет бесконечную (строчную) норму матрицы\n
  она используется для проверки условия сходимости итерационных методов

  :param alpha: квадратная матрица размера (n, n).
  """
```

```
  return max(np.sum(np.abs(alpha), axis=1))



def solve_simple_iteration(A, b, tolerance=1e-6, max_iterations=1000):
  """
  Решает систему линейных уравнений Ax = b методом простых итераций.

  :param A: квадратная матрица коэффициентов (n, n), диагональные элементы !=
0.
  :param b: вектор правой части (n,).
  :param tolerance: точность решения.
  :param max_iterations: максимальное число итераций.
  """
  n = len(b)
  x = np.zeros(n)
  iterations = 0

  alpha = np.zeros((n, n))
  beta = np.zeros(n)
  for i in range(n):
    beta[i] = b[i] / A[i, i]
    for j in range(n):
      if i != j:
        alpha[i, j] = -A[i, j] / A[i, i]

  alpha_norm = matrix_inf_norm(alpha)
  coefficient = alpha_norm / (1 - alpha_norm)
  for _ in range(max_iterations):
    x_new = np.dot(alpha, x) + beta
    diff_norm = np.max(np.abs(x_new - x))

    if alpha_norm <= 1:
      if coefficient * diff_norm < tolerance:
        break
    else:
      if diff_norm < tolerance:
        break
```

```
    x = x_new
    iterations += 1

  return x, iterations



def solve_zeidel(A, b, tolerance=1e-10, max_iterations=1000):
  """
  Решает систему линейных уравнений Ax = b методом Зейделя

  :param A: квадратная матрица коэффициентов (n, n), диагональные элементы !=
0.
  :param b: вектор правой части (n,).
  :param tolerance: точность решения.
  :param max_iterations: максимальное число итераций.
  """
  n = len(b)
  x = np.zeros(n)
  iterations = 0

  alpha = np.zeros((n, n))
  beta = np.zeros(n)
  for i in range(n):
    beta[i] = b[i] / A[i, i]
    for j in range(n):
      if i != j:
        alpha[i, j] = -A[i, j] / A[i, i]

  alpha_norm = matrix_inf_norm(alpha)
  coefficient = alpha_norm / (1 - alpha_norm)
  for _ in range(max_iterations):
    x_new = np.copy(x)
    for i in range(n):
      x_new[i] = (b[i] - np.dot(A[i, :i], x_new[:i]) - np.dot(A[i, i+1:],
x_new[i+1:])) / A[i, i]
```

```
    diff_norm = np.max(np.abs(x_new - x))
    if alpha_norm >= 1:
      if diff_norm < tolerance:
        break
    else:
      if coefficient * diff_norm < tolerance:
        break

    x = x_new
    iterations += 1

  return x, iterations



A = np.array([
  [-24, -6, 4, 7],
  [-8, 21, 4, -2],
  [6, 6, 16, 0],
  [-7, -7, 5, 24]
], dtype=float)
b = np.array([130, 139, -84, -165], dtype=float)

print("A = ", A)

x_simple, iter_simple = solve_simple_iteration(A, b, tolerance=1e-6)
print("Метод простых итераций:")
print("Решение:", x_simple)
print("Количество итераций:", iter_simple)

x_seidel, iter_zeidel = solve_zeidel(A, b, tolerance=1e-6)
print("\nМетод Зейделя:")
print("Решение:", x_seidel)
print("Количество итераций:", iter_zeidel)
```

Вывод программы

```
A =  [[-24.  -6.   4.   7.]
 [ -8.  21.   4.  -2.]
 [  6.   6.  16.   0.]
 [ -7.  -7.   5.  24.]]
Метод простых итераций:
Решение: [-9.          3.00000017 -3.00000019 -7.99999974]
Количество итераций: 17

Метод Зейделя:
Решение: [-8.99999999  3.00000004 -3.00000002 -7.99999998]
Количество итераций: 11
```

Задание 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.

$$23.\begin{pmatrix} 9 & -5 & -6 \\ -5 & 1 & -8 \\ -6 & -8 & -3 \end{pmatrix}$$

Теоретические сведения

Метод вращений Якоби применим только для симметрических матриц ($A^T = A$) 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 $\Lambda = U^T AU$, где Λ – диагональная матрица с собственными значениями на главной диагонали.

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 $A^{(k)}(|a_{ij}^{(k)}| = \left|a_{lm}^{(k)}\right|)$.
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:

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

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

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

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

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

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

Код программы

```
import numpy as np
import matplotlib.pyplot as plt


def jacobi_eigenvalue(A, tolerance=1e-20):
  """
  Вычисляет собственные значения и собственные векторы симметричной матрицы
  используя повороты к оператору

  :param A: Входная симметричная матрица, должна удовлетворять A == A^t
  :param tolerance: Порог сходимости: алгоритм останавливается,
    когда сумма квадратов всех внедиагональных элементов становится меньше tolerance.
  """
  n = A.shape[0]
  V = np.eye(n)
  error_history = []

  while True:
    off_diag_sum = np.sum(np.square(A - np.diag(np.diag(A))))
    error_history.append(off_diag_sum)

    if off_diag_sum < tolerance:
      break

    without_diag = A - np.diag(np.diag(A))
    max_abs_of_non_diag = np.argmax(np.abs(without_diag))
    p, q = np.unravel_index(max_abs_of_non_diag, A.shape)
```

```
    theta = 0.5 * np.arctan(2 * A[p, q] / (A[p, p] - A[q, q])) if A[p, p] !=
A[q, q] else np.pi / 4


    J = np.eye(n)
    J[p, p] = np.cos(theta)
    J[q, q] = np.cos(theta)
    J[p, q] = -np.sin(theta)
    J[q, p] = np.sin(theta)

    A = J.T @ A @ J
    V = V @ J

  eigenvalues = np.diag(A)
  eigenvectors = V

  return eigenvalues, eigenvectors, error_history


A = np.array([
  [9, -5, -6],
  [-5, 1, -8],
  [-6, -8, -3]
])
eigenvalues, eigenvectors, error_history = jacobi_eigenvalue(A, tolerance=1e-
10)


print("Собственные значения:")
print(eigenvalues)
print("\nСобственные векторы:")
print(eigenvectors)


plt.plot(error_history)
plt.xlabel('Итерации')
plt.ylabel('Сумма квадратов внедиагональных элементов')
plt.title('Зависимость суммы квадратов внедиагональных элементов от числа
итераций')
plt.show()
```

Вывод программы

```
Собственные значения:
[ 11.89563801   7.2349017  -12.13053971]

Собственные векторы:
[[ 0.936918    0.04544384  0.34658263]
 [-0.25237568  0.77396189  0.58076631]
 [-0.2418495  -0.63159944  0.73660775]]
```

Задание 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.

$$23.\begin{pmatrix} 1 & 5 & -6 \\ 9 & -7 & -9 \\ 6 & -1 & -9 \end{pmatrix}$$

Теоретические сведения

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:

$$H = E - \frac{2}{v^T v} v v^T, \text{где } v - \text{произвольный ненулевой столбец.}$$

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

$$\tilde{b} = Hb, b = (b_1, b_2, \dots, b_n)^T, \tilde{b} = \left(\widetilde{b_1}, 0, \dots, 0\right)^T$$

Тогда вектор v определится следующим образом:

$$v = b + sign(b_1)\left|\left|b\right|\right|_2 e_1$$

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

Процедура QR-разложения многократно используется в QR-алгоритме вычисления собственных значений. Строится следующий итерационный процесс:

$$A^{(0)} = A,$$

$A^{(0)} = Q^{(0)}R^{(0)}$ – производится QR-разложение,

$A^{(1)} = R^{(0)}Q^{(0)}$ – производится перемножение матриц,

……..

$A^{(k)} = Q^{(k)}R^{(k)}$ – разложение

$A^{(k+1)} = R^{(k)}Q^{(k)}$ – перемножение

Таким образом, каждая итерация реализуется в два этапа. На первом этапе осуществляется разложение матрицы $A^{(k)}$ в произведение ортогональной $Q^{(k)}$ и верхней треугольной $R^{(k)}$ матриц, а на втором – полученные матрицы перемножаются в обратном порядке.

При отсутствии у матрицы кратных собственных значений последовательность $A^{(k)}$ сходится к верхней треугольной матрице (в случае, когда все собственные значения вещественны) или к верхней квазитреугольной матрице (если имеются комплексносопряженные пары собственных значений).

Таким образом, каждому вещественному собственному значению будет соответствовать столбец со стремящимися к нулю поддиагональными элементами и в качестве критерия сходимости итерационного процесса для таких собственных значений можно использовать следующее неравенство: $(\sum_{l=m+1}^{n} (a_{lm}^{(k)})^2)^{\frac{1}{2}} \leq \varepsilon$. При этом соответствующее собственное значение принимается равным диагональному элементу данного столбца.

Код программы

```
import numpy as np



def qr_decomposition(A):
  """
  Выполняет QR-разложение квадратной матрицы A с использованием метода отражений Хаусхолдера

  :param A: квадратная матрица размера n x n

  :returns:
    Q: ортогональная матрица (n x n)
    R: верхнетреугольная матрица (n x n), равная преобразованной A
  """
```

```
  n = len(A)
  A = np.copy(A)
  Q = np.eye(n)

  for i in range(n):
    v = np.zeros((n, 1))
    norm = np.sqrt(sum([A[j][i] ** 2 for j in range(i, n)]))
    v[i] = A[i][i] + np.sign(A[i][i]) * norm

    for j in range(i + 1, n):
      v[j] = A[j][i]

    vT = np.transpose(v)
    denominator = np.dot(vT, v)
    if denominator == 0:
      H = np.eye(n)
    else:
      H = np.eye(n) - 2 / denominator * np.dot(v, vT)

    Q = np.dot(Q, H)
    A = np.dot(H, A)

  return Q, A



def solve_qr(A, eps):
  """
  Реализует QR-алгоритм для нахождения собственных значений квадратной
матрицы A

  :returns:
    lambdas, где:
      lambdas[i][0] — вещественная часть i-го собственного значения,
      lambdas[i][1] — мнимая часть i-го собственного значения
    iterations: количество выполненных итераций
  """
  n = len(A)
```

```
A = np.copy(A)
iterations = 0
lambdas = np.empty((n, 2))

while True:
  iterations += 1
  Q, R = qr_decomposition(A)
  A = np.dot(R, Q)

  flg = True
  skip = False
  for i in range(n):
    if skip:
      skip = False
      continue

    if i < n - 1:
      a = A[i][i]
      d = A[i + 1][i + 1]
      b = A[i][i + 1]
      c = A[i + 1][i]
      D = a ** 2 + d ** 2 - 2 * a * d + 4 * b * c

      if D < 0:
        re = (a + d) / 2.0
        im = np.sqrt(-D) / 2.0
        lambda_ = np.sqrt(re ** 2 + im ** 2)
        if iterations > 1:
          lambdaPrev = np.sqrt(lambdas[i][0] ** 2 + lambdas[i][1] ** 2)
          if abs(lambda_ - lambdaPrev) > eps:
            flg = False

        lambdas[i][0] = re
        lambdas[i][1] = im
        lambdas[i + 1][0] = re
        lambdas[i + 1][1] = -im
```

```
          skip = True
          continue

      lambdas[i][0] = A[i][i]
      lambdas[i][1] = 0.0

      sum_ = np.sqrt(sum([A[j][i] ** 2 for j in range(i + 1, n)]))
      if sum_ > eps:
        flg = False

    if flg:
      break

  return lambdas, iterations

A = np.array([
  [1, 5, -6],
  [9, -7, -9],
  [6, -1, -9]
])

Q, R = qr_decomposition(A)
print("Q:")
print(Q)
print("R:")
print(R)

Q_my, R_my = qr_decomposition(A)
Q_lib, R_lib = np.linalg.qr(A)

ortho_my = np.allclose(Q_my.T @ Q_my, np.eye(len(A)), atol=1e-10)
ortho_lib = np.allclose(Q_lib.T @ Q_lib, np.eye(len(A)), atol=1e-10)

print(f"\nОртогональность Q (своя): {ortho_my}")
print(f"Ортогональность Q (библиотека): {ortho_lib}")
```

```
lam, iterations = solve_qr(A, 0.01)
print("Собственные значения (вещественная и мнимая части):")
print(lam)
print("Количество итераций:", iterations)
```

Вывод программы

```
Q:
[[-0.09205746  0.87318679  0.47861285]
 [-0.82851716 -0.333787    0.44960601]
 [-0.55234477  0.35514937 -0.75417782]]
R:
[[-1.08627805e+01  5.89167755e+00  1.29801021e+01]
 [-1.44250819e-15  6.34729356e+00 -5.43138204e+00]
 [-1.21204133e-15  4.15968353e-18 -1.30530776e-01]]

Ортогональность Q (своя): True
Ортогональность Q (библиотека): True
Собственные значения (вещественная и мнимая части):
[[-12.7862646    0.        ]
 [ -1.82935075   0.        ]
 [ -0.38438465   0.        ]]
Количество итераций: 5
```

Лабораторная работа №2

Задание 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.

$$23. \qquad \ln(x+2) - x^4 + 0.5 = 0\,.$$

Теоретические сведения

Метод простых итераций:

Пусть требуется решить трансцендентное уравнение вида $f(x) = 0$. Предположим, что корень уравнения отделен и находится на отрезке $[a, b]$. Кроме того, функция удовлетворяет некоторым дополнительным условиям:

- на концах отрезка функция имеет разные знаки;
- $\forall x \in [a, b]: f'(x) \neq 0$;
- первая и вторая производные имеют постоянные знаки.

Для того чтобы построить итерационный процесс, согласно методу простой итерации уравнение $f(x) = 0$ заменяется эквивалентным уравнением: $x = \varphi(x)$, причем $\varphi(x)$ – непрерывная функция. Выберем некоторое нулевое приближение x^0, а затем организуем итерационный процесс по схеме:

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

Условие $\left|\varphi'(x)\right| \leq q < 1, \forall x \in [a, b]$ является достаточным условием сходимости итераций, если начальное приближение выбрано в некоторой окрестности корня.

Метод Ньютона:

Предположим, что на интервале $[a, b]$ требуется определить корень уравнения $f(x) = 0$. Для того чтобы построить итерационный процесс согласно методу Ньютона, непрерывная функция $f(x)$ на интервалах $x \in [a, b]$ должна удовлетворять условиям, аналогичным условиям в методе итераций:

Правило построения итерационной последовательности получается путем замены нелинейной функции $f(x)$ ее линейной моделью на основе формулы Тейлора. Выберем в окрестности решения уравнения две соседние точки, так что $x^{(k+1)} = x^{(k)} + \varepsilon$, и запишем разложение функции в ряд Тейлора:

$$f\left(x^{(k+1)}\right) = f\left(x^{(k)}\right) + f'\left(x^{(k)}\right)\left(x^{(k+1)} - x^{(k)}\right) + \frac{1}{2}f''\left(x^{(k)}\right)\left(x^{(k+1)} - x^{(k)}\right)^2 + \cdots$$

Учитывая, что $f(x_{k+1}) \approx 0$ и оставляя только линейную часть разложения ряда, запишем соотношение метода Ньютона:

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$$

Используя условие сходимости метода простых итераций, получим достаточное условие сходимости метода Ньютона в форме

$$|f(x) \cdot f''(x)| < (f'(x))^2, x \in [a, b]$$

Для выбора начального приближения $x^{(0)}$ используется теорема, которая гласит, что в качестве начального приближения нужно выбрать тот конец интервала, где знак функции совпадает со знаком второй производной:

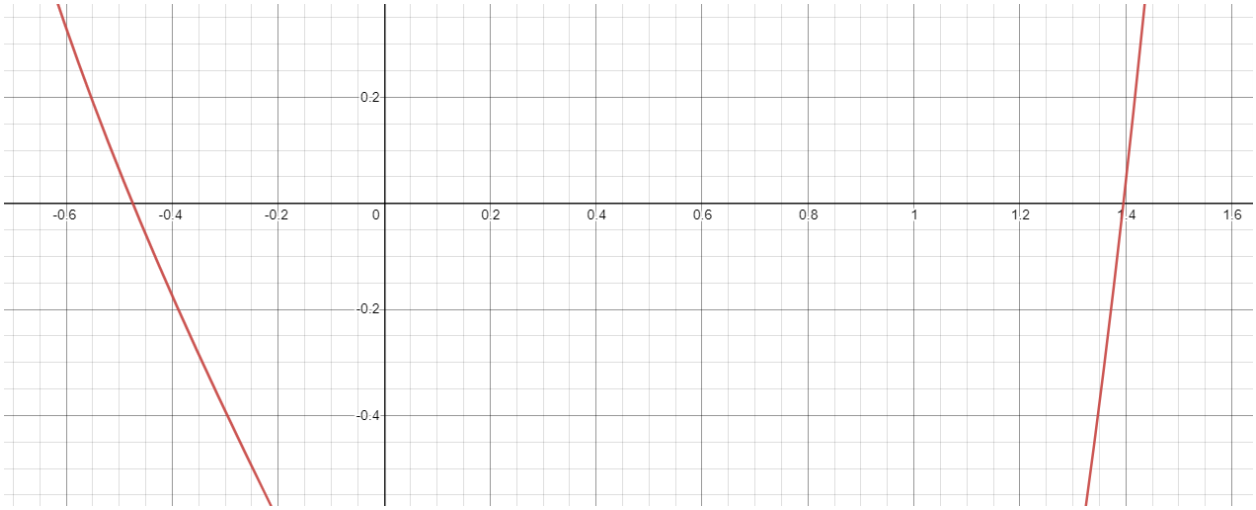
$$x^{(0)} = \{a, \text{если } f(a) \cdot f''(a) > 0 \; b, \text{если } f(b) \cdot f''(b) > 0$$

Для завершения итерационного процесса используется правило:

$$\left|x^{(k+1)} - x^{(k)}\right| < \varepsilon$$

Подготовка уравнения к решению методами Ньютона и простой итерации:

1. Графически определим интервалы корней уравнения $f(x) = 0$:



$$x_1 \epsilon [-0.6, -0.4], x_2 \epsilon [1.3, 1.5]$$

2. Найдем 1 и 2 производные функции $f(x)$:

$$f'(x) = 4x^3 - 2$$

$$f''(x) = 12x^2$$

3. Подберем подходящие функции $\varphi(x)$ и найдем их производные:

$$x = \frac{x^4 - 1}{2} = \varphi_1(x); \; x = \sqrt[4]{2x + 1} = \varphi_2(x)$$

$$\varphi_1'(x) = 2x^3;\ \varphi_2'(x) = \frac{2}{\sqrt[4]{(2x+1)^3}}$$

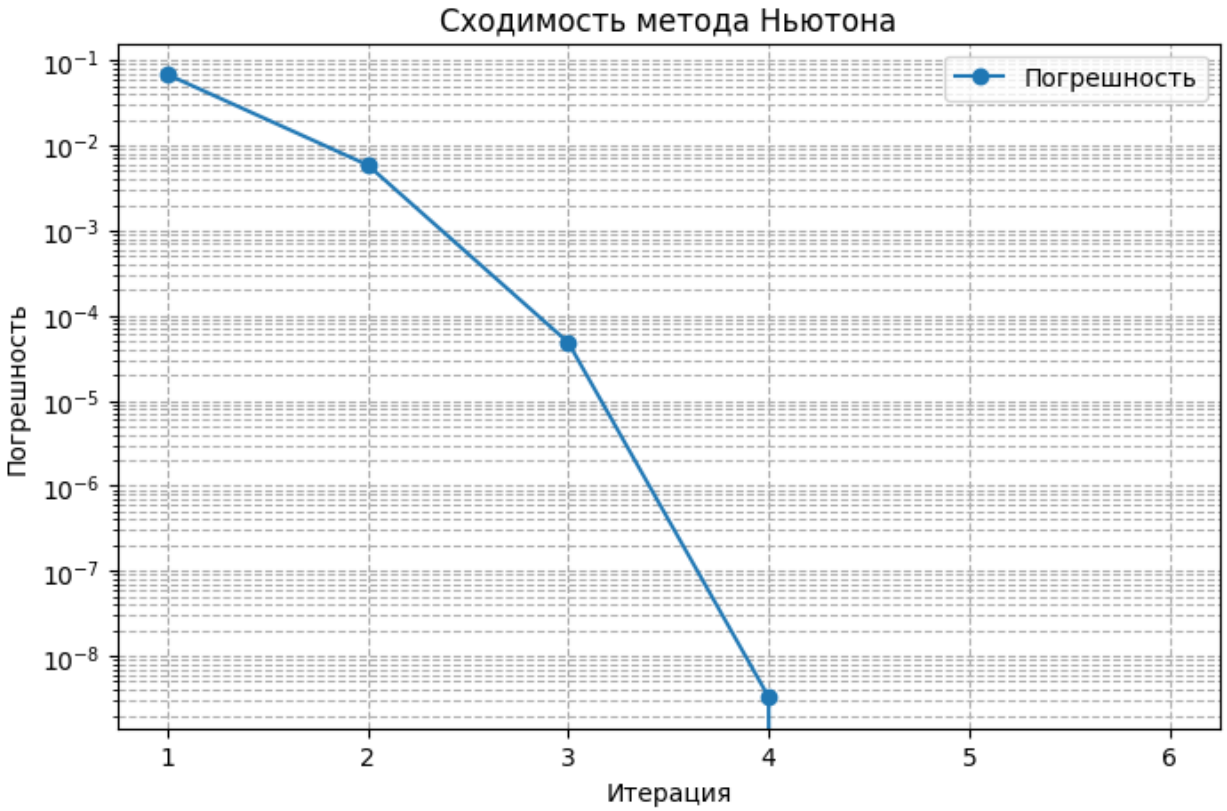
Код программы

```
import math
from scipy.optimize import root_scalar

def f(x): return math.log(x + 2) - x ** 4 + 0.5
def df_dx(x): return 1 / (x + 2) - 4 * x ** 3
def d2f_dx(x): return -1 / ((x + 2) ** 2) - 12 * x ** 2
def phi(x): return (math.log(x + 2) + 0.5) ** (1 / 4)

def solve_newton(x0, eps):
  if (f(x0) * d2f_dx(x0) <= 0):
    raise Exception("Последовательность может не сойтись")

  xPrev = x0
  iterations = 0
  while (True):
    iterations += 1
    xCur = xPrev - f(xPrev) / df_dx(xPrev)
    if abs(xCur - xPrev) < eps:
      break
    xPrev = xCur
  return xCur, iterations



def solve_simple_iterations(x0, q, eps):
  xPrev = x0
  iterations = 0
  while (True):
    iterations += 1
    xCur = phi(xPrev)
    error = q / (1 - q) * abs(xCur - xPrev)
    if error < eps:
      break
    xPrev = xCur
```

```
  return xCur, iterations



eps = 1e-10
x0 = 1.15

newton_res, iterations = solve_newton(x0, eps)
print("Метод Ньютона")
print("\tКорень: ", newton_res)
print("\tКоличество итераций: ", iterations)

x_one = 1
q = 0.25 / ((x_one + 2) * ((math.log(x_one + 2) + 0.5) ** (3 / 4)))
simpleIterationsAns, iterations = solve_simple_iterations(x0, q, eps)
print("Метода простых итераций")
print("\tКорень: ", simpleIterationsAns)
print("\tКоличество итераций: ", iterations)

sol = (root_scalar(f, bracket=[1.1, 1.2], method='brentq', xtol=eps)).root
print("Библиотечный метод:")
print(f"\tКорень: {sol:.12f}")
```

Вывод программы

```
Метод Ньютона
        Корень:  1.131931474415741
        Количество итераций:  4
Метода простых итераций
        Корень:  1.1319314744432623
        Количество итераций:  7
Библиотечный метод:
        Корень: 1.131931474413
```



Задание 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.

22	1	$\begin{cases} ax_1^2 - x_1 + x_2^2 - 1 = 0, \\ x_2 - \operatorname{tg} x_1 = 0. \end{cases}$
23	2	
24	3	

a = 2

Теоретические сведения

Метод простых итераций:

Решение системы нелинейных уравнений вида

$$\{f_1(x_1, x_2, x_3, \dots, x_n) = 0\ f_2(x_1, x_2, x_3, \dots, x_n) = 0\ \dots\ f_n(x_1, x_2, x_3, \dots, x_n) = 0$$

возможно, если все функции в системе непрерывны и дифференцируемы в окрестности решения.

Для использования метода итераций система уравнений записывается в эквивалентной форме:

$$\{x_1 = \phi_1(x_1, x_2, x_3, \dots, x_n)\ x_2 = \phi_2(x_1, x_2, x_3, \dots, x_n)\ \dots\ x_n = \phi_n(x_1, x_2, x_3, \dots, x_n),$$

где ϕ_i – итерирующие непрерывно дифференцируемые функции.

Тогда, если известно начальное приближение $X^{(0)}$, то можно построить алгоритм метода простых итераций:

$$\{x_1^{(k+1)} = \phi_1\left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \dots, x_n^{(k)}\right)\ x_2^{(k+1)} = \phi_2\left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \dots, x_n^{(k)}\right)\ \dots\ x_n^{(k+1)} = \phi_n\left(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \dots, x_n^{(k)}\right)$$

Предложение формулировки достаточного условия сходимости для многомерного случая выглядит следующим образом. Метод простых итераций сходится к решению системы нелинейных уравнений, если какая-либо норма матрицы Якоби $J(X^{(k)})$, построенная по правым частям ϕ_i эквивалентной системы в замкнутой области G, меньше единицы на каждой итерации:

$$J\left(X^{(k)}\right) = \left[\frac{\partial \phi_i\left(X^{(k)}\right)}{\partial x_j}\right];\ \ \|J\left(X^{(k)}\right)\| \le q < 1$$

Для практических расчетов чаще всего используют матричную норму, определенную в области решения G, которую обязательно проверяют в начальном приближении:

$$\|\phi'(x)\| = \left\{\sum_{j=1}^{n} \left|\frac{\partial \phi_i(X)}{\partial x_j}\right|\right\}$$

В практических вычислениях в качестве условия окончания итераций обычно используется критерий

$$\left|X^{(k+1)} - X^{(k)}\right| < \varepsilon$$

Метод Ньютона:

Пусть дана система нелинейных уравнений. Все функции системы непрерывны на некотором интервале $[a, b]$ и дифференцируемы вплоть до вторых производных.

Итерационный процесс нахождения решения, который носит название метода Ньютона для систем, записывается в виде решения матричного уравнения:

$$X^{(k+1)} = X^{(k)} - J^{-1}\left(X^{(k)}\right) \cdot f\left(X^{(k)}\right),$$

где $J\left(X^{(k)}\right)$ – матрица Якоби.

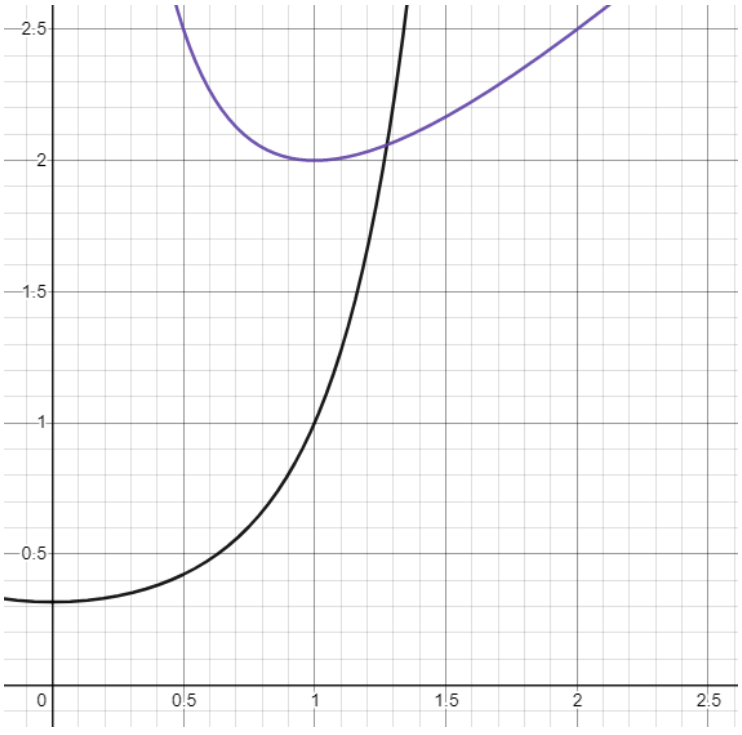
$$J\left(X^{(k)}\right) = \begin{vmatrix} \frac{\partial f_1\left(X^{(k)}\right)}{\partial x_1} & \frac{\partial f_1\left(X^{(k)}\right)}{\partial x_2} & \dots & \frac{\partial f_1\left(X^{(k)}\right)}{\partial x_n} \\ \frac{\partial f_2\left(X^{(k)}\right)}{\partial x_1} & \frac{\partial f_2\left(X^{(k)}\right)}{\partial x_2} & \dots & \frac{\partial f_2\left(X^{(k)}\right)}{\partial x_n} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_n\left(X^{(k)}\right)}{\partial x_1} & & & \end{vmatrix}$$

В практических вычислениях в качестве условия окончания итераций обычно используют критерий, выполняющийся для всех переменных системы:

$$\left|X^{(k+1)} - X^{(k)}\right| < \varepsilon$$

Подготовка системы к решению методами Ньютона и простой итерации

1. Графически определим примерное расположение корня системы:



В качестве начального приближения возьмем точку $X^{(0)} = (1, 1)$.

Вычислим матрицу Якоби для данной системы уравнений:

$$J(X) = \begin{vmatrix} 2x_1 & -\frac{2}{ln10 \cdot x_2} \\ 2x_1 - x_2 & -x_1 \end{vmatrix}$$

2. Подберем подходящие итерирующие функции ϕ_i и найдем матрицу Якоби для системы, записанной в эквивалентной форме:

$$\{x_1 = \sqrt{2lgx_2 + 1} = \phi_1 \; x_2 = x_1 + \frac{1}{x_1} = \phi_2$$

$$J(\phi) = \begin{vmatrix} 0 & \frac{1}{x_2\sqrt{ln10 \cdot (2lnx_2 + ln10)}} \\ 1 - \frac{1}{x_1^2} & 0 \end{vmatrix}$$

Код программы

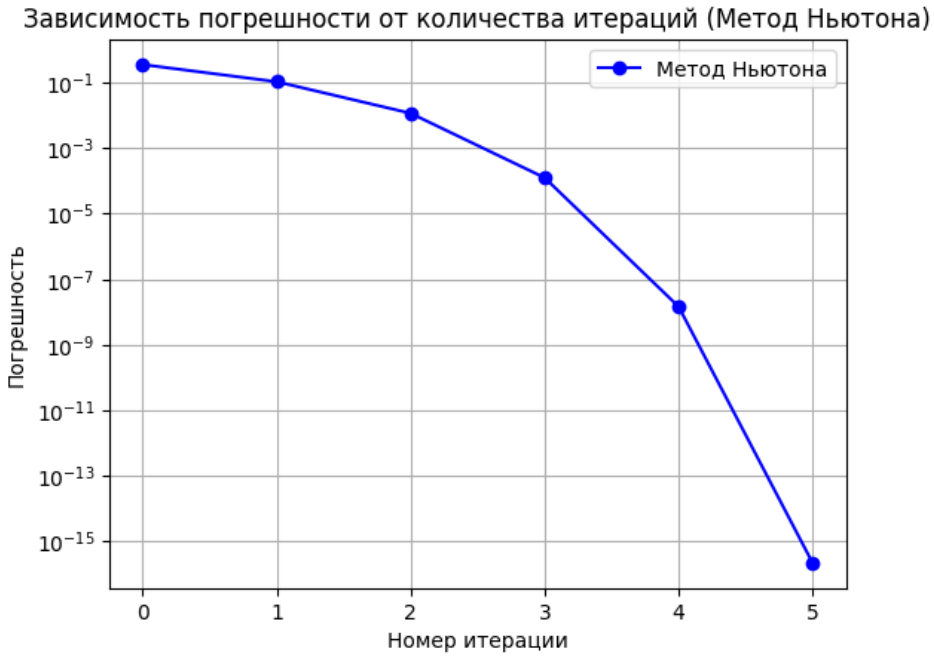
```
import numpy as np
from chislaki.lab1.solution import solve_lu, lu
import matplotlib.pyplot as plt


def f(x):
  return np.array([
    2 * x[0]**2 - x[0] + x[1]**2 - 1,
    x[1] - np.tan(x[0])
  ])
def df_dx(x):
  cos_x1 = np.cos(x[0])
  sec2_x1 = 1.0 / (cos_x1 ** 2)
  return np.array([
    [4 * x[0] - 1, 2 * x[1]],
    [-sec2_x1, 1]
  ])
def phi(x):
  x2_new = np.tan(x[0])
  x1_new = (1 + np.sqrt(9 - 8 * x2_new**2)) / 4
  return np.array([x1_new, x2_new])



def newton_with_history(x0, eps):
  xPrev = x0
  iterations = 0
  history = [xPrev.copy()]
  while True:
```

```
    iterations += 1
    P, L, U, _ = lu(df_dx(xPrev))
    xDelta = solve_lu(P, L, U, -f(xPrev))
    xCur = xPrev + xDelta
    history.append(xCur.copy())
    if np.max(np.abs(xCur - xPrev)) < eps:
      break
    xPrev = xCur
    if iterations > 50:
      break
  return xCur, iterations, history

def solve_simple_iterations(x0, eps, lam=0.08):
  def phi(x):
    return x - lam * f(x)

  xPrev = x0.copy()
  iterations = 0
  while True:
    iterations += 1
    xCur = phi(xPrev)
    if np.max(np.abs(xCur - xPrev)) < eps:
      break
    xPrev = xCur
    if iterations > 10000:
      raise RuntimeError("Простые итерации не сошлись")
  return xCur, iterations



eps = 1e-10
x0 = np.array([0.5, 0.5])
exact_solution, _, _ = newton_with_history(x0, eps)
eps_plot = 1e-6
_, _, newton_history = newton_with_history(x0, eps_plot)
```

```
newton_errors = []
for x in newton_history:
    err = np.linalg.norm(x - exact_solution, ord=np.inf)
    newton_errors.append(err)


plt.semilogy(range(len(newton_errors)), newton_errors, 'bo-', label='Метод Ньютона')
plt.xlabel('Номер итерации')
plt.ylabel('Погрешность')
plt.title('Зависимость погрешности от количества итераций (Метод Ньютона)')
plt.grid(True, which="both", ls="-")
plt.legend()
plt.show()


newton_ans, iterations, _ = newton_with_history(x0, eps)
print("Метод Ньютона")
print("\tКорень: ", newton_ans)
print("\tКоличество итераций: ", iterations)


simple_ans, iterations = solve_simple_iterations(x0, 1e-6, lam=0.08)
print("Метода простых итераций")
print("\tКорень: ", simple_ans)
print("\tКоличество итераций: ", iterations)
```

Вывод программы

```
Метод Ньютона
        Корень:  [0.70224654 0.84613601]
        Количество итераций:  6
Метода простых итераций
        Корень:  [0.70225254 0.84613642]
        Количество итераций:  102
```



Лабораторная работа №3

Задание 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^*.

23. $y = \frac{1}{x}$, а) $X_i = 0.1, 0.5, 0.9, 1.3$; б) $X_i = 0.1, 0.5, 1.1, 1.3$; $X^* = 0.8$.

Теоретические сведения

Пусть на отрезке $[a, b]$ задано множество несовпадающих точек x_i (интерполяционных узлов), в которых известны значения функции $f_i = f(x_i)$. Приближающая функция $\varphi(x, a)$ такая, что выполняются равенства

$$\varphi(x_i, a_0, \dots, a_n) = f(x_i)$$

называется интерполяционной.

Наиболее часто в качестве приближающей функции используют многочлены степени n:

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

Произвольный многочлен может быть записан в виде

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

Здесь $l_i(x)$ – многочлены степени n, так называемые лагранжевы многочлены влияния, которые удовлетворяют условию $l_i(x_j) = \{1, \text{при } i = j \; 0, \text{при } i \neq j$ и, соответственно,

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

а интерполяционный многочлен запишется в виде

$$L_n(x) = \sum_{i=0}^{n} f_i \prod_{j=0, j\neq i}^{n} \frac{(x - x_i)}{(x_i - x_j)}$$

Интерполяционный многочлен, записанный в этой форме, называется **интерполяционным многочленом Лагранжа**.

Недостатком интерполяционного многочлена Лагранжа является необходимость полного пересчета всех коэффициентов в случае добавления дополнительных интерполяционных узлов. Чтобы избежать указанного недостатка используют интерполяционный многочлен в форме Ньютона.

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:

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

Разделенная разность $n - k + 2$ определяется соостношениями

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

Интерполяционный многочлен Ньютона может быть представлен в виде:

$$P_n(x) = f(x_0) + (x - x_0)f(x_1, x_0) + (x - x_0)(x - x_1)f(x_0, x_1, x_2) + \cdots \\ + (x - x_0)(x - x_1) \dots (x - x_n)f(x_0, x_1, \dots, x_n)$$

Отметим, что при добавлении новых узлов первые члены многочлена Ньютона остаются неизменными.

Для повышения точности интерполяции в сумму могут быть добавлены новые члены, что требует подключения дополнительных интерполяционных узлов. При этом безразлично, в каком порядке подключаются новые узлы. Этим формула Ньютона выгодно отличается от формулы Лагранжа.

Код программы

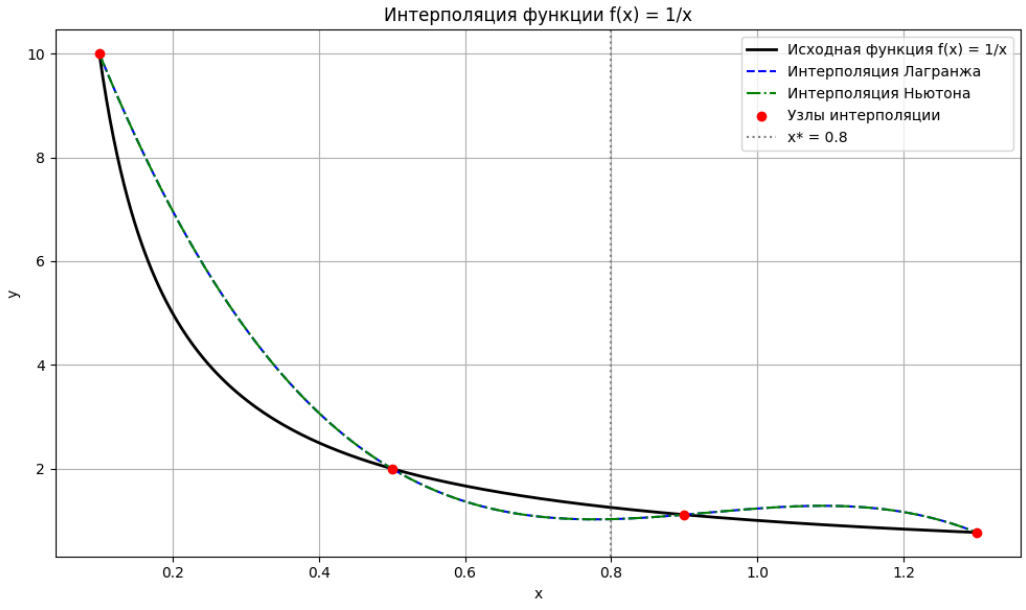
```
import numpy as np
import matplotlib.pyplot as plt


xi = np.array([0.1, 0.5, 1.1, 1.3])


yi = 1 / xi
x_star = 0.8
f_real = 1 / x_star


def lagrange_interpolation(x, xi, yi):
    n = len(xi)
    Lx = 0
    for i in range(n):
        li = 1
        for j in range(n):
            if i != j:
                li *= (x - xi[j]) / (xi[i] - xi[j])
        Lx += yi[i] * li
    return Lx
```

```
def newton_coefficients(xi, yi):
    n = len(xi)
    coef = np.copy(yi)
    for j in range(1, n):
        coef[j:n] = (coef[j:n] - coef[j - 1:n - 1]) / (xi[j:n] - xi[0:n - j])
    return coef

def newton_interpolation(x, xi, coef):
    n = len(coef)
    Nx = coef[0]
    mult_term = 1.0
    for i in range(1, n):
        mult_term *= (x - xi[i - 1])
        Nx += coef[i] * mult_term
    return Nx

lagrange_result = lagrange_interpolation(x_star, xi, yi)
newton_coef = newton_coefficients(xi, yi)
newton_result = newton_interpolation(x_star, xi, newton_coef)

lagrange_error = abs(f_real - lagrange_result)
newton_error = abs(f_real - newton_result)

print(f"Точное значение f({x_star}) = {f_real}")
print(f"Интерполяция Лагранжа: {lagrange_result}, погрешность:
{lagrange_error}")
print(f"Интерполяция Ньютона: {newton_result}, погрешность: {newton_error}")



x_vals = np.linspace(0.1, 1.3, 500)
f_vals = 1 / x_vals
lagrange_vals = [lagrange_interpolation(x, xi, yi) for x in x_vals]
newton_vals = [newton_interpolation(x, xi, newton_coef) for x in x_vals]
```

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

plt.plot(x_vals, f_vals, label='Исходная функция f(x) = 1/x', color='black', linewidth=2)

plt.plot(x_vals, lagrange_vals, label='Интерполяция Лагранжа', linestyle='--', color='blue')

plt.plot(x_vals, newton_vals, label='Интерполяция Ньютона', linestyle='-.', color='green')

plt.scatter(xi, yi, color='red', label='Узлы интерполяции', zorder=5)

plt.axvline(x_star, color='gray', linestyle=':', label=f'x* = {x_star}')


plt.title('Интерполяция функции f(x) = 1/x')

plt.xlabel('x')

plt.ylabel('y')

plt.legend()

plt.grid(True)

plt.tight_layout()

plt.show()
```

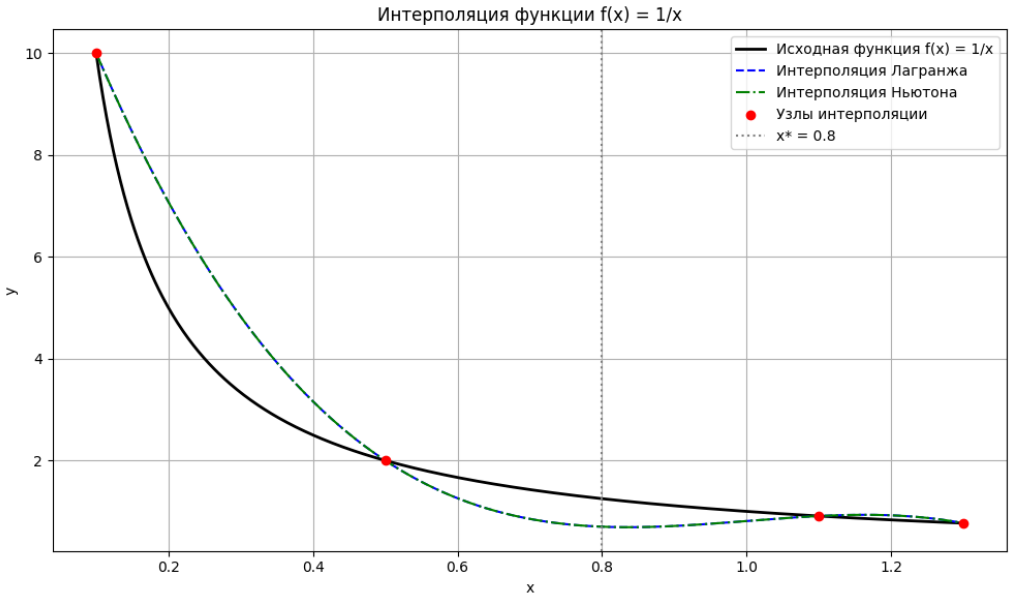
Вывод программы

первый набор точек




```
Точное значение f(0.8) = 1.25
Интерполяция Лагранжа: 1.0256410256410258, погрешность: 0.22435897435897423
Интерполяция Ньютона: 1.025641025641025, погрешность: 0.2243589743589749
```

второй набор точек




```
Точное значение f(0.8) = 1.25
Интерполяция Лагранжа: 0.6993006993006992, погрешность: 0.5506993006993008
Интерполяция Ньютона: 0.6993006993006993, погрешность: 0.5506993006993007
```

Задание 2

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

23. X^* =0.8

i	0	1	2	3	4
x_i	0.1	0.5	0.9	1.3	1.7
f_i	10.0	2.0	1.1111	0.76923	0.58824

Теоретические сведения

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.

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

$$S(x) = \sum_{k=0}^{n} a_{ik}x^k, x_{i-1} \le x \le x_i, \ \ i = \underline{1,n}$$

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

$$S(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3,$$
$$x_{i-1} \le x \le x_i, \ \ i = \underline{1,n}$$

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.

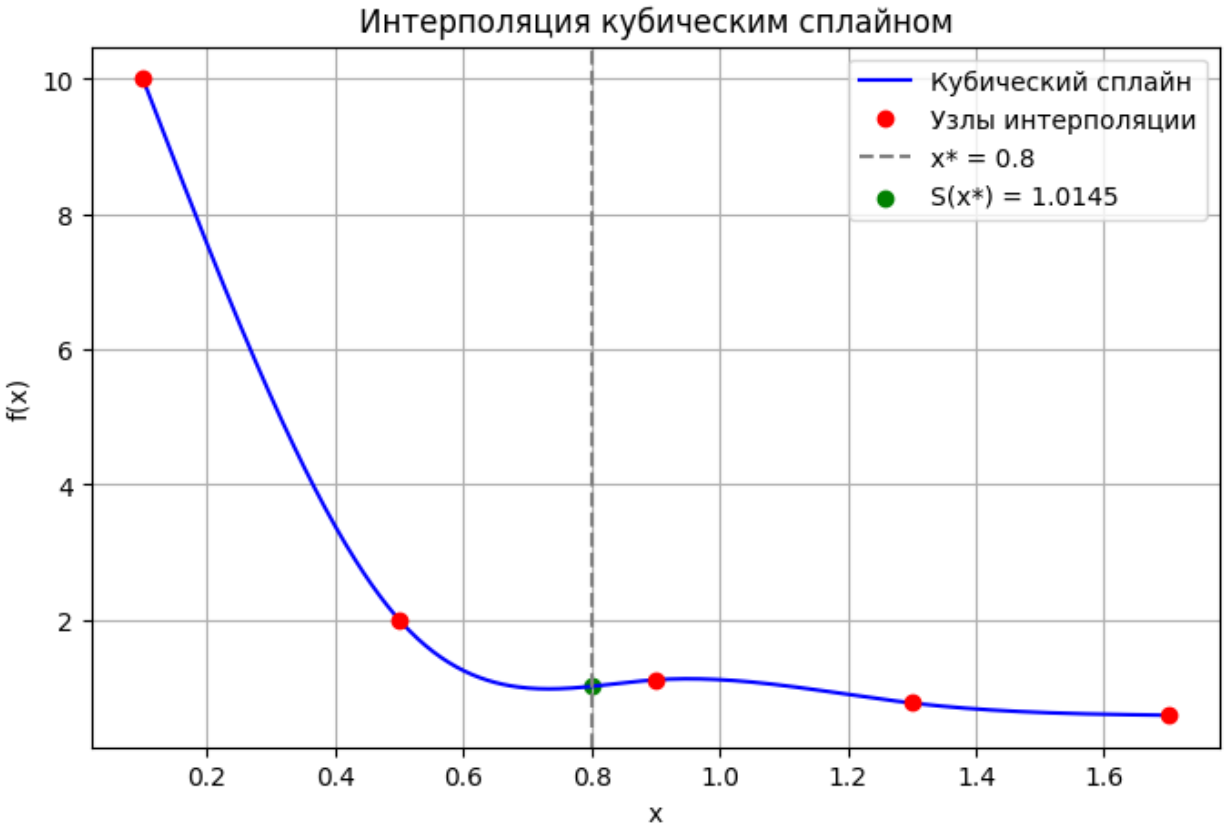
Код программы

```
import numpy as np
import matplotlib.pyplot as plt
from lab1.solution import progonka


x = np.array([0.1, 0.5, 0.9, 1.3, 1.7])
f = np.array([10, 2, 1.1111, 0.76923, 0.58824])
x_star = 0.8


n = len(x)
h = np.diff(x)


a_ = h[:-1]
b_ = 2 * (h[:-1] + h[1:])
c_ = h[1:]
d_ = 3 * ((f[2:] - f[1:-1]) / h[1:] - (f[1:-1] - f[:-2]) / h[:-1])
```

```
c_vals = np.zeros(n)
c_vals[1:-1] = progonka(a_, b_, c_, d_)


a_coef = f[:-1]
b_coef = np.zeros(n - 1)
d_coef = np.zeros(n - 1)


for i in range(1, n):
  b_coef[i - 1] = (f[i] - f[i - 1]) / h[i - 1] - h[i - 1] * (c_vals[i] + 2 *
c_vals[i - 1]) / 3
  d_coef[i - 1] = (c_vals[i] - c_vals[i - 1]) / (3 * h[i - 1])


def spline_value_methodical(x_val):
  for i in range(n - 1):
    if x[i] <= x_val <= x[i + 1]:
      dx = x_val - x[i]
      return (
        a_coef[i]
        + b_coef[i] * dx
        + c_vals[i] * dx**2
        + d_coef[i] * dx**3
      )
  return None


s_val = spline_value_methodical(x_star)
print(f"Значение сплайна в точке x* = {x_star} : {s_val:.6f}")


X_dense = np.linspace(x[0], x[-1], 400)
Y_dense = [spline_value_methodical(xi) for xi in X_dense]


plt.figure(figsize=(8, 5))
plt.plot(X_dense, Y_dense, label="Кубический сплайн", color="blue")
plt.plot(x, f, "ro", label="Узлы интерполяции")
plt.axvline(x_star, color='gray', linestyle='--', label=f"x* = {x_star}")
```

```
plt.scatter([x_star], [s_val], color='green', label=f"S(x*) = {s_val:.4f}")
plt.legend()
plt.title("Интерполяция кубическим сплайном")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid(True)
plt.show()
plt.savefig("SPLINE")
```

Вывод программы




```
Значение сплайна в точке x* = 0.8 : 1.014535
```

Задание 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.

23.

i	0	1	2	3	4	5
x_i	0.1	0.5	0.9	1.3	1.7	2.1
y_i	10.	2.0	1.1111	0.76923	0.58824	0.47619

Теоретические сведения

Пусть задана таблично в узлах x_j функция $y_i = f(x_i)$. При этом значения функции y_i 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.

$$\Phi = \sum_{j=0}^{n} [F_n(x_j) - y_i]^2$$

Минимума Ф можно добиться только за счет изменения коэффициентов многочлена $F_n(x)$. Необходимые условия экстремума имеют вид

$$\frac{\partial \Phi}{\partial a_k} = 2 \sum_{j=0}^{N} \left[\sum_{j=0}^{n} a_i x_j^i - y_j \right] x_j^k = 0, \quad k = \underline{0,n}$$

Эту систему для удобства преобразуют к следующему виду:

$$\sum_{i=0}^{n} a_i \sum_{j=0}^{N} x_j^{k+1} = \sum_{j=0}^{N} y_j x_j^k, \quad k = \underline{0,n}$$

Эта система называется нормальной системой метода наименьших квадратов (МНК) и представляет собой систему линейных алгебраических уравнений относительно коэффициентов a_i. Решив систему, построим многочлен $F_n(x)$, приближающий функцию $f(x)$ и минимизирующий квадратичное отклонение.

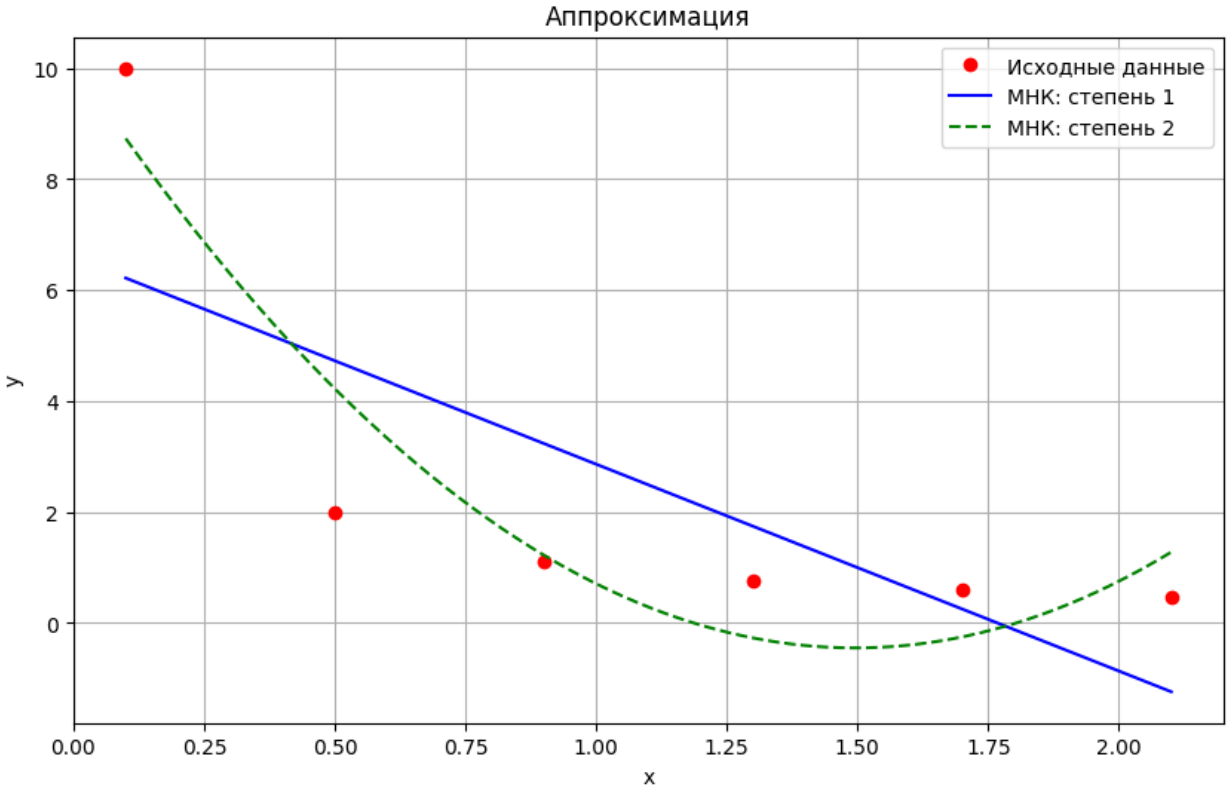
Код программы

```
x = np.array([0.1, 0.5, 0.9, 1.3, 1.7, 2.1])
y = np.array([10, 2, 1.1111, 0.76923, 0.58824, 0.47619])
```

```
def build_normal_system(x, y, degree):
  A = np.zeros((degree + 1, degree + 1))
  b = np.zeros(degree + 1)

  for k in range(degree + 1):
    for i in range(degree + 1):
      A[k, i] = np.sum(x ** (i + k))
    b[k] = np.sum(y * (x ** k))

  return A, b



def fit_polynomial(x, y, degree):
  A, b = build_normal_system(x, y, degree)
  P, L, U, _ = lu(A)
  coeffs = solve_lu(P,L,U, b)
  y_pred = np.zeros_like(x, dtype=float)
  for i in range(len(x)):
    for j in range(degree + 1):
      y_pred[i] += coeffs[j] * (x[i] ** j)

  sse = np.sum((y - y_pred) ** 2)
  return coeffs, sse



def manual_polyval(coeffs, x):
  y = 0
  for power, coeff in enumerate(coeffs):
    y += coeff * (x ** power)
  return y



coeffs_deg1, sse_deg1 = fit_polynomial(x, y, degree=1)
coeffs_deg2, sse_deg2 = fit_polynomial(x, y, degree=2)
```

```
print("Коэффициенты многочлена 1-й степени:", coeffs_deg1)
print("Сумма квадратов ошибок (1-я степень):", sse_deg1)
print("Коэффициенты многочлена 2-й степени:", coeffs_deg2)
print("Сумма квадратов ошибок (2-я степень):", sse_deg2)

x_plot = np.linspace(min(x), max(x), 500)

y_deg1 = manual_polyval(coeffs_deg1, x_plot)
y_deg2 = manual_polyval(coeffs_deg2, x_plot)

plt.figure(figsize=(10, 6))
plt.plot(x, y, 'ro', label='Исходные данные')
plt.plot(x_plot, y_deg1, 'b-', label='МНК: степень 1')
plt.plot(x_plot, y_deg2, 'g--', label='МНК: степень 2')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Аппроксимация')
plt.legend()
plt.grid(True)
plt.show()
```

Вывод программы




```
Коэффициенты многочлена 1-й степени: [ 6.59192333 -3.7283    ]
Сумма квадратов ошибок (1-я степень): 30.254114148333336
Коэффициенты многочлена 2-й степени: [ 10.0988132  -14.10743594   4.71778906]
Сумма квадратов ошибок (2-я степень): 8.98184733247501
```

Задание 4

Вычислить первую и вторую производную от таблично заданной функции $y_i = f(x_i),\ i = 0,1,2,3,4$ в точке $x = X^*$.

23. $X^* = 2.0$

i	0	1	2	3	4
x_i	1.0	1.5	2.0	2.5	3.0
y_i	2.0	2.1667	2.5	2.9	3.3333

Теоретические сведения

Формулы численного дифференцирования в основном используются при нахождении производных от функции $y = f(x)$, заданной таблично. Исходная функция $y_i = f(x_i)$ на отрезках$[x_j, x_{j+k}]$, заменяется некоторой приближающей, легко вычисляемой функцией $\varphi(x, \underline{a}), y = \varphi(x, \underline{a}) + R(x)$, где $R(x)$ – остаточный член приближения, $\underline{a}$ – набор коэффициентов, вообще

говоря, различный для каждого из рассматриваемых отрезков, и полагают, что $y'(x) \approx \varphi'(x, \underline{a})$. Наиболее часто в качестве приближающей функции $\varphi(x, \underline{a})$ берется интерполяционный многочлен $\varphi(x, \underline{a}) = P_n(x)$, 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. В этом случае:

$$y' \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i} = const$$

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

$$y'(x) \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i} + \frac{\frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}}{x_{i+2} - x_i}(2x - x_i - x_{i+1})$$

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

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

$$y''(x) \approx 2\frac{\frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}}{x_{i+2} - x_i}$$

Код программы

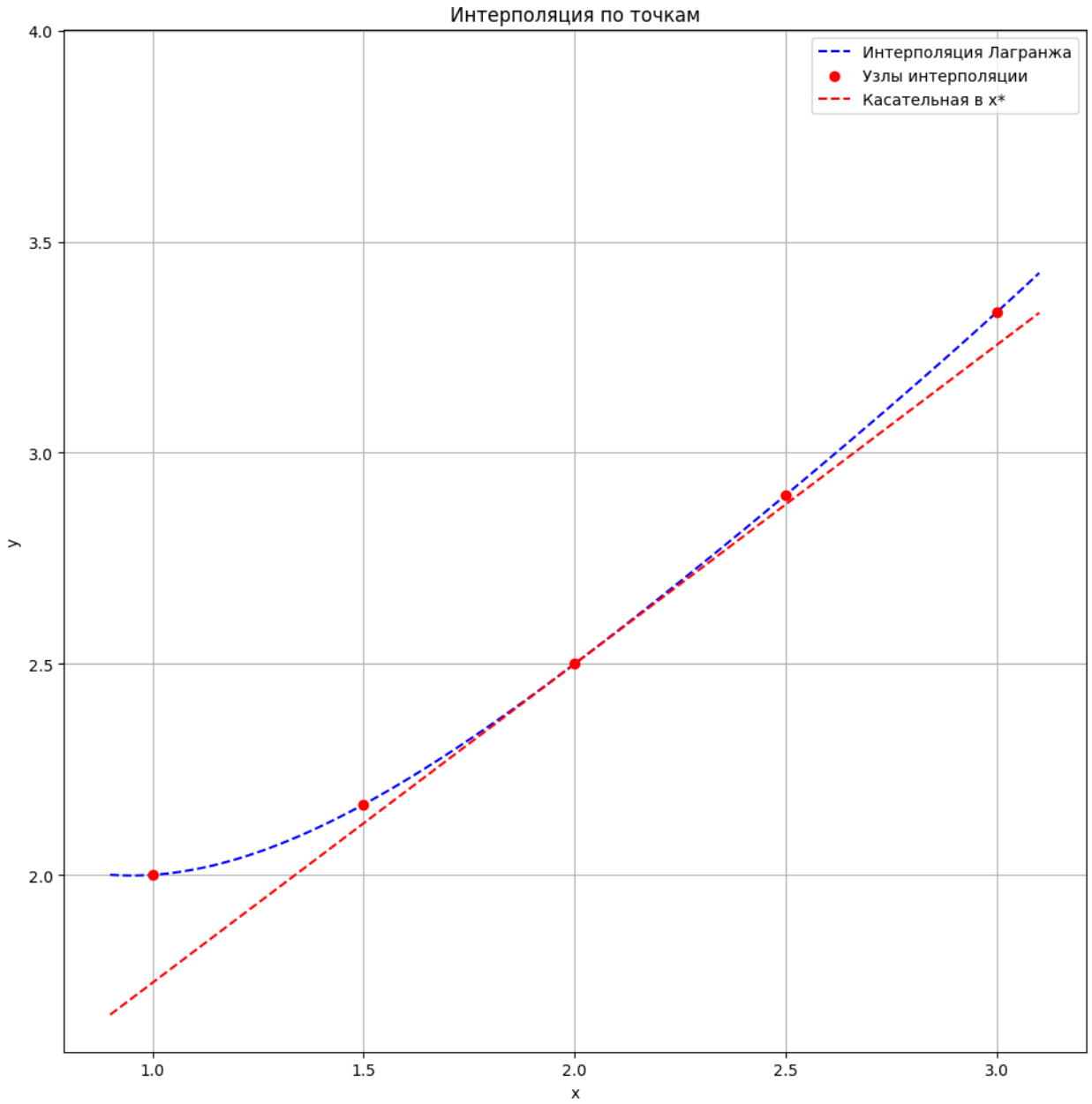
```
import numpy as np
import matplotlib.pyplot as plt



def lagrange_interpolation(x, xi, yi):
  n = len(xi)
  Lx = 0
  for i in range(n):
    li = 1
    for j in range(n):
```

```
      if i != j:
        li *= (x - xi[j]) / (xi[i] - xi[j])
    Lx += yi[i] * li
  return Lx



x = np.array([1, 1.5, 2.0, 2.5, 3.0])
y = np.array([2, 2.1667, 2.5, 2.9, 3.3333])

i = 2
x_star = 2.0

def get_index(x_star, x):
  for i in range(len(x) - 2):
    if x[i] <= x_star <= x[i + 1]:
      return i

i = get_index(x_star, x)

dy1 = (y[i + 1] - y[i]) / (x[i + 1] - x[i])
dy2 = (y[i + 2] - y[i + 1]) / (x[i + 2] - x[i + 1])
correction = (dy2 - dy1) / (x[i + 2] - x[i]) * (2 * x_star - x[i] - x[i + 1])

first_derivative = dy1 + correction
second_derivative = 2 * (dy2 - dy1) / (x[i + 1] - x[i - 1])

print(f"Первая производная в x* = {x_star}: {first_derivative:.6f}")
print(f"Вторая производная в x* = {x_star}: {second_derivative:.6f}")

tan = lambda x_star: (lagrange_interpolation(x_star + 1e-3, x, y) -
lagrange_interpolation(x_star, x, y)) / 1e-3
print(f"Первая производная по функции интерполяции = {tan(x_star)}")

x_vals = np.linspace(0.9, 3.1, 500)
tangent_line = tan(x_star)*(x_vals-x_star) + lagrange_interpolation(x_star,
x, y)
```

```
lagrange_vals = [lagrange_interpolation(point, x, y) for point in x_vals]


plt.figure(figsize=(10, 10))
plt.plot(x_vals, lagrange_vals, label='Интерполяция Лагранжа', linestyle='--
', color='blue')
plt.scatter(x, y, color='red', label='Узлы интерполяции', zorder=5)
plt.plot(x_vals, tangent_line, '--', label='Касательная в x*', color='red')


plt.title('Интерполяция по точкам')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.axis('square')
plt.show()
```

Вывод программы

```
Первая производная в x* = 2.0: 0.733300
Вторая производная в x* = 2.0: 0.266800
Первая производная по функции интерполяции = 0.7556388945109127
```



Задание 5

Вычислить определенный интеграл $F = \int_{X_0}^{X_1} ydx$, методами прямоугольников, трапеций, Симпсона с шагами h_1, h_2. Оценить погрешность вычислений, используя Метод Рунге-Ромберга:

23. $y = \dfrac{1}{\sqrt{(2x+7)(3x+4)}}$, $X_0 = 0$, $X_k = 4$, $h_1 = 1.0$, $h_2 = 0.5$;

Теоретические сведения

Формулы численного интегрирования используются в тех случаях, когда вычислить аналитически определенный интеграл $F = \int_a^b f(x)dx$ не удается. Рассмотрим наиболее простой и часто применяемый способ, когда подынтегральную функцию заменяют на интерполяционный многочлен.

При использовании интерполяционных многочленов различной степени, получают формулы численного интегрирования различного порядка точности.

Заменим подынтегральную функцию, интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка – точку $\underline{x_i} = \frac{x_{i-1}+x_i}{2}$, получим **формулу прямоугольников**:

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

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.

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

Эта формула носит название **формулы трапеций**.

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: $x_{i-1},\ x_{i-\frac{1}{2}} = \frac{x_{i-1}+x_i}{2},\ x_i$.

Для случая $h_i = \frac{x_i - x_{i-1}}{2}$, получим **формулу Симпсона**:

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

Метод Рунге-Ромберга-Ричардсона позволяет получать более высокий порядок точности вычисления. Если имеются результаты вычисления определенного интеграла порядка точности p на сетке с шагом h: F_h и на сетке с шагом kh: F_{kh}, то

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

Код программы

```
import numpy as np


def f(x):
    return 1 / np.sqrt((2 * x + 7) * (3 * x + 4))


def generate_points(a, b, h):
    points = []
    current = a
    while current < b:
        points.append(current)
        current += h
    points.append(b)
    return points


def rectangle(a, b, h):
    total = 0
    xs = generate_points(a, b, h)
    for i in range(1, len(xs)):
        mid = (xs[i - 1] + xs[i]) / 2
        total += h * f(mid)
    return total


def trapez(a, b, h):
    total = 0
    xs = generate_points(a, b, h)
    for i in range(1, len(xs)):
        total += 0.5 * h * (f(xs[i - 1]) + f(xs[i]))
    return total


def simpson(a, b, h):
    total = 0
```

```
    xs = generate_points(a, b, h)
    for i in range(1, len(xs)):
        left = xs[i - 1]
        right = xs[i]
        mid = (left + right) / 2
        total += (1/3) * (h / 2) * (f(left) + 4 * f(mid) + f(right))
    return total

def runge_romberg_error(results, steps, order):
    ratio = steps[0] / steps[1]
    return (results[1] - results[0]) / (ratio ** order - 1)

def runge_romberg_refinement(results, steps, order):
    return results[1] + runge_romberg_error(results, steps, order)

a, b = 0, 4
h1 = 1
h2 = 0.5
hs = [h1, h2]

methods = {
    'Прямоугольники': (rectangle, 2),
    'Трапеции': (trapez, 2),
    'Симпсон': (simpson, 4)
}

true_value = (2 / np.sqrt(6)) * np.log(
    (np.sqrt(2 * (3 * b + 4)) + np.sqrt(3 * (2 * b + 7))) /
    (np.sqrt(2 * (3 * a + 4)) + np.sqrt(3 * (2 * a + 7)))
)

print(f"Точное значение интеграла на [{a}, {b}]: {true_value:.8f}\n")

for name, (method, p) in methods.items():
    I_h1 = method(a, b, h1)
```

```
    I_h2 = method(a, b, h2)
    RR_error = runge_romberg_error([I_h1, I_h2], hs, p)
    RR_value = runge_romberg_refinement([I_h1, I_h2], hs, p)
    abs_error = abs(true_value - RR_value)
    print(f"{name}:")
    print(f"  I(h1={h1}) = {I_h1:.6f}")
    print(f"  I(h2={h2}) = {I_h2:.6f}")
    print(f"  Runge-Romberg уточнение = {RR_value:.6f}")
print(f"  Погрешность RR ≈ {RR_error:.6f}")
print(f"  Абсолютная погрешность ≈ {abs_error:.6f}\n")
```

Вывод программы

```
Точное значение интеграла на [0, 4]: 0.41797182

Прямоугольники:
  I(h1=1) = 0.414533
  I(h2=0.5) = 0.417075
  Runge-Romberg уточнение = 0.417922
  Погрешность RR ≈ 0.000847
  Абсолютная погрешность ≈ 0.000050

Трапеции:
  I(h1=1) = 0.425023
  I(h2=0.5) = 0.419778
  Runge-Romberg уточнение = 0.418030
  Погрешность RR ≈ -0.001748
  Абсолютная погрешность ≈ 0.000058

Симпсон:
  I(h1=1) = 0.418030
  I(h2=0.5) = 0.417976
  Runge-Romberg уточнение = 0.417972
  Погрешность RR ≈ -0.000004
  Абсолютная погрешность ≈ 0.000001
```

Лабораторная работа №4

Задание 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.

23	$x^2y''+xy'-y-3x^2=0,$ $y(1)=3,$ $y'(1)=2,$ $x\hat{I}\ [1,2], h=0.1$ Equation.3	$y=x^2+x+\frac{1}{x}$

Теоретические сведения

Рассматривается задача Коши для одного дифференциального уравнения первого порядка разрешенного относительно производной

$$y' = f(x, y)$$

$$y(x_0) = y_0$$

Требуется найти решение на отрезке $[a, b]$, где $x_0 = a$.

Введем разностную сетку на отрезке $[a, b]$: $\Omega^{(k)} = \{x_k = x_0 + hk\}$.

Точки x_i называются узлами разностной сетки, расстояния между узлами — *шагом разностной сетки*, а совокупность значений какой-либо величины заданных в узлах сетки называется *сеточной функцией*.

Приближенное решение задачи Коши будем искать численно в виде сеточной функции.

Метод Эйлера (явный):

Метод Эйлера играет важную роль в теории численных методов решения ОДУ, хотя и не часто используется в практических расчетах из-за невысокой точности. Вывод расчетных соотношений для этого метода может быть произведен несколькими способами: с помощью геометрической интерпретации, с использованием разложения в ряд Тейлора, конечно разностным методом (с помощью разностной аппроксимации производной), квадратурным способом (использованием эквивалентного интегрального уравнения).

Рассмотрим вывод соотношений метода Эйлера геометрическим способом. Решение в узле x_0 известно из начальных условий. Рассмотрим процедуру получения решения в узле x_1.

График функции $y^{(h)}$, которая является решением задачи Коши, 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 $y_{1\text{ист}}$. При этом допускается погрешность $|y_1 - y_{1\text{ист}}|$ геометрически представленная отрезком CD. 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:

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

Метод Рунге-Кутты:

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

$$y_{k+1} = y_k + \Delta y_k$$

$$\Delta y_k = \sum_{i=1}^{p} c_i K_i^k$$

$$K_i^k = hf(x_k + a_i h, y_k + h \sum_{j=1}^{i-1} b_{ij} K_j^k)$$

$$i = \underline{2, p}$$

Метод Рунге-Кутты четвертого порядка:

$$a = \left(0, \frac{1}{2}, \frac{1}{2}, 1\right); b = (0\ 0\ 0\ 0\ 0.5\ 0\ 0\ 0\ 0\ 0.5\ 0\ 0\ 0\ 0\ 0.5\ 0); c = \left(\frac{1}{6}, \frac{1}{3}, \frac{1}{3}, \frac{1}{6}\right)$$

$$\Delta y_k = \frac{1}{6}\left(K_1^k + K_2^k + K_3^k + K_4^k\right)$$

$$K_1^k = hf(x_k, y_k);\ K_2^k = hf(x_k + 0.5h, y_k + 0.5K_1^k)$$

$$K_3^k = hf(x_k + 0.5h, y_k + 0.5K_2^k);\ K_4^k = hf(x_k + h, y_k + K_3^k)$$

Метод Адамса:

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

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

Метод Адамса 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.

Решение задачи Коши для системы обыкновенных дифференциальных уравнений:

Рассматривается задача Коши для системы дифференциальных уравнений первого порядка разрешенных относительно производной

$$\{y_1' = f_1(x, y, y_1, y_2, \dots, y_n)\ y_2' = f_2(x, y, y_1, y_2, \dots, y_n)\ \dots\ y_n' = f_n(x, y, y_1, y_2, \dots, y_n)$$

$$y_1(x_0) = y_{01}\ y_2(x_0) = y_{02}\ \dots\ y_n(x_0) = y_{0n}$$

К системе дифференциальных уравнений можно применить все методы рассмотренные выше. Уравнения решаются по порядку.

Для задачи Коши 2-го порядка $y'' = f(x, y, y')$ можно применить следующее разложение в систему (используя замену $z = y'(x)$:

$$\{z' = f(x, y, z)\ y' = z(x)$$

$$y(x_0) = y_0\ y'(x_0) = z_0$$

Код программы

```
import numpy as np
import matplotlib.pyplot as plt



def generate_points(a, b, h):
  points = []
  current = a
  while current < b:
    points.append(current)
    current += h
  points.append(b)
  return points



def f(x, y1, y2):
  return (1 / x**2) * (y1 + 3 * x**2 - x * y2)



def exact_solution(x):
  return x**2 + x + 1/x



def euler_method(a, b, h, y1_0, y2_0):
  xs = generate_points(a, b, h)
  y1s = [y1_0]
  y2s = [y2_0]
  for i in range(1, len(xs)):
    x = xs[i - 1]
    y1 = y1s[-1]
    y2 = y2s[-1]
    y1_next = y1 + h * y2
    y2_next = y2 + h * f(x, y1, y2)
    y1s.append(y1_next)
    y2s.append(y2_next)
```

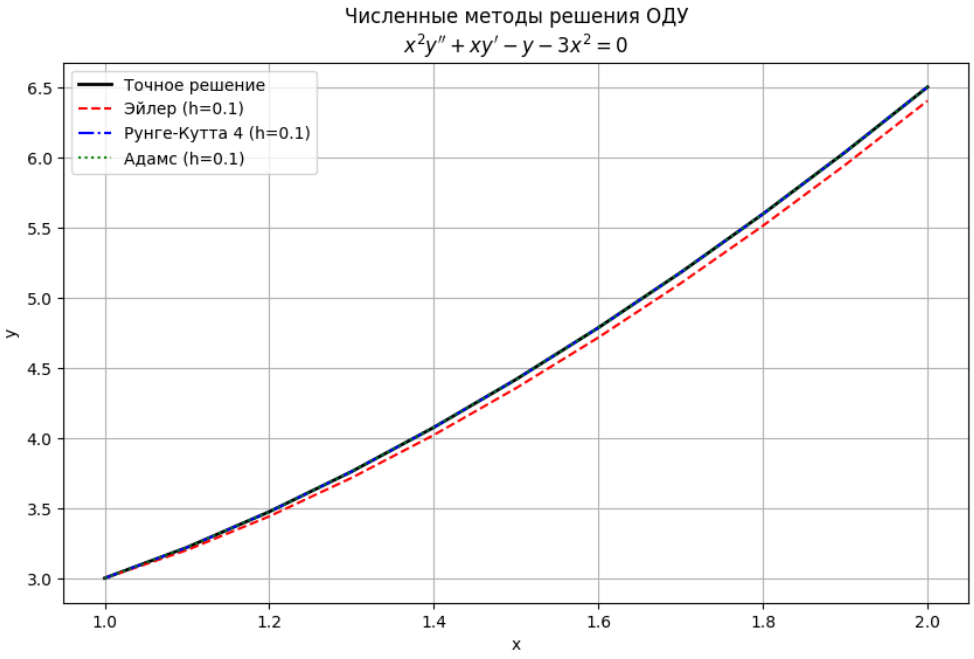
```
  return xs, y1s


def runge_kutta_method(a, b, h, y1_0, y2_0):
  xs = generate_points(a, b, h)
  y1s = [y1_0]
  y2s = [y2_0]
  for i in range(1, len(xs)):
    x = xs[i - 1]
    y1 = y1s[-1]
    y2 = y2s[-1]

    k1 = h * y2
    l1 = h * f(x, y1, y2)

    k2 = h * (y2 + l1 / 2)
    l2 = h * f(x + h / 2, y1 + k1 / 2, y2 + l1 / 2)

    k3 = h * (y2 + l2 / 2)
    l3 = h * f(x + h / 2, y1 + k2 / 2, y2 + l2 / 2)

    k4 = h * (y2 + l3)
    l4 = h * f(x + h, y1 + k3, y2 + l3)

    y1_next = y1 + (k1 + 2 * k2 + 2 * k3 + k4) / 6
    y2_next = y2 + (l1 + 2 * l2 + 2 * l3 + l4) / 6

    y1s.append(y1_next)
    y2s.append(y2_next)

  return xs, y1s


def adams_method(a, b, h, y1_0, y2_0):
  xs = generate_points(a, b, h)
```

```
  y1s = [y1_0]
  y2s = [y2_0]

  for i in range(3):
    x = xs[i]
    y1 = y1s[-1]
    y2 = y2s[-1]

    k1 = h * y2
    l1 = h * f(x, y1, y2)

    k2 = h * (y2 + l1 / 2)
    l2 = h * f(x + h / 2, y1 + k1 / 2, y2 + l1 / 2)

    k3 = h * (y2 + l2 / 2)
    l3 = h * f(x + h / 2, y1 + k2 / 2, y2 + l2 / 2)

    k4 = h * (y2 + l3)
    l4 = h * f(x + h, y1 + k3, y2 + l3)

    y1_next = y1 + (k1 + 2 * k2 + 2 * k3 + k4) / 6
    y2_next = y2 + (l1 + 2 * l2 + 2 * l3 + l4) / 6

    y1s.append(y1_next)
    y2s.append(y2_next)

  for i in range(3, len(xs) - 1):
    x = xs[i]
    y1_next = y1s[i] + h / 24 * (55 * y2s[i] - 59 * y2s[i - 1] + 37 * y2s[i -
2] - 9 * y2s[i - 3])
    y2_next = y2s[i] + h / 24 * (
      55 * f(xs[i], y1s[i], y2s[i]) -
      59 * f(xs[i - 1], y1s[i - 1], y2s[i - 1]) +
      37 * f(xs[i - 2], y1s[i - 2], y2s[i - 2]) -
      9 * f(xs[i - 3], y1s[i - 3], y2s[i - 3])
```

```
    )
    y1s.append(y1_next)
    y2s.append(y2_next)

  return xs, y1s


def runge_romberg(y_h, y_h2, p):
  return [(abs(yh - yh2) / (2**p - 1)) for yh, yh2 in zip(y_h, y_h2)]


a, b = 1, 2
h = 0.1
y1_0 = 3
y2_0 = 2

x_euler, y_euler = euler_method(a, b, h, y1_0, y2_0)
x_rk, y_rk = runge_kutta_method(a, b, h, y1_0, y2_0)
x_adams, y_adams = adams_method(a, b, h, y1_0, y2_0)

_, y_euler2 = euler_method(a, b, h/2, y1_0, y2_0)
rr_euler = runge_romberg(y_euler, y_euler2[::2], 1)

_, y_runge_kutta2 = runge_kutta_method(a, b, h/2, y1_0, y2_0)
rr_runge_kutta = runge_romberg(y_rk, y_runge_kutta2[::2], 4)

_, y_adams2 = adams_method(a, b, h/2, y1_0, y2_0)
rr_adams = runge_romberg(y_adams, y_adams2[::2], 4)

y_exact = [exact_solution(x) for x in x_rk]

errors_e = [abs(ye - yt) for ye, yt in zip(y_euler, y_exact)]
errors_rk = [abs(ye - yt) for ye, yt in zip(y_rk, y_exact)]
errors_ad = [abs(ye - yt) for ye, yt in zip(y_adams, y_exact)]
```

```
print(f"{'x':>5} {'Точное':>12} {'Эйлер':>12} {'РК4':>12} {'Адамс':>12}
{'|Эйл - точн|':>15} {'|РК4 - точн|':>15} {'|Адм - точн|':>15} {'Эйл
corr':>15} {'РК4 corr':>15} {'Адм corr':>15}")

for i in range(len(x_rk)):

    print(f"{x_rk[i]:5.2f} {y_exact[i]:12.6f} {y_euler[i]:12.6f}
{y_rk[i]:12.6f} {y_adams[i]:12.6f} {errors_e[i]:15.6e} {errors_rk[i]:15.6e}
{errors_ad[i]:15.6e}  {rr_euler[i]:15.6e}  {rr_runge_kutta[i]:15.6e}
{rr_adams[i]:15.6e}")



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

plt.plot(x_rk, y_exact, label="Точное решение", linewidth=2, color='black')

plt.plot(x_euler, y_euler, label="Эйлер (h=0.1)", linestyle='--',
color='red')

plt.plot(x_rk, y_rk, label="Рунге-Кутта 4 (h=0.1)", linestyle='-.',
color='blue')

plt.plot(x_adams, y_adams, label="Адамс (h=0.1)", linestyle=':',
color='green')

plt.legend()

plt.grid(True)

plt.xlabel("x")

plt.ylabel("y")

plt.title("Численные методы решения ОДУ\n$x^2 y'' + x y' - y - 3x^2 = 0$")

plt.show()
```

Вывод программы



x	Точное	Эйлер	РК4	Адамс	|Эйл - точн|
1.00	3.000000	3.000000	3.000000	3.000000	0.000000e+00
1.10	3.219091	3.200000	3.219093	3.219093	1.909091e-02
1.20	3.473333	3.440000	3.473336	3.473336	3.333333e-02
1.30	3.759231	3.714628	3.759235	3.759235	4.460267e-02
1.40	4.074286	4.020259	4.074290	4.074440	5.402630e-02
1.50	4.416667	4.354361	4.416671	4.416738	6.230599e-02
1.60	4.785000	4.715109	4.785005	4.785076	6.989092e-02
1.70	5.178235	5.101160	5.178241	5.178194	7.707498e-02
1.80	5.595556	5.511502	5.595561	5.595417	8.405381e-02
1.90	6.036316	5.945357	6.036321	6.036048	9.095928e-02
2.00	6.500000	6.402119	6.500006	6.499612	9.788096e-02

|РК4 - точн|	|Адм - точн|	Эйл corr	РК4 corr	Адм corr
0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
1.881464e-06	1.881464e-06	1.000000e-02	1.178046e-07	1.178046e-07
3.018253e-06	3.018253e-06	1.729266e-02	1.889614e-07	3.149632e-07
3.755626e-06	3.755626e-06	2.294970e-02	2.351029e-07	4.589787e-07
4.268592e-06	1.544152e-04	2.760475e-02	2.671923e-07	9.843802e-06
4.650710e-06	7.083490e-05	3.164648e-02	2.910899e-07	4.778423e-06
4.954280e-06	7.598180e-05	3.532041e-02	3.100708e-07	5.755915e-06
5.209657e-06	4.126861e-05	3.878525e-02	3.260357e-07	1.361864e-06
5.435027e-06	1.388410e-04	4.214542e-02	3.401232e-07	7.138624e-06
5.641580e-06	2.674216e-04	4.547052e-02	3.530338e-07	1.497322e-05
5.836358e-06	3.875474e-04	4.880733e-02	3.652082e-07	2.224575e-05

Задание 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.

23	$x(x^2+6)y^2-4(x^2+3)y' +6xy=0,$ $y'(0)=0,$ $y(4)-y'(4)=26$	$y(x)=x^3+x^2+2$

Теоретические сведения

Примером краевой задачи является двухточечная краевая задача для обыкновенного дифференциального уравнения второго порядка.

$$y'' = f(x, y, y')$$

с граничными условиями, заданными на концах отрезка $[a, b]$.

$$y(a) = y_0\, y(b) = y_1 \ - \text{граничные условия 1 рода}$$

Следует найти такое решение $y(x)$ на этом отрезке, которое принимает на концах отрезка значения y_0, y_1.

Кроме граничных условий первого рода, используются еще условия на производные от решения на концах - граничные условия второго рода:

$$y'(a) = \hat{y}_0\, y'(b) = \hat{y}_1$$

или линейная комбинация решений и производных – граничные условия третьего рода:

$$\alpha y(a) + \beta y'(a) = \ \hat{y}_0\, \delta y(b) + \gamma y'(b) = \ \hat{y}_1$$

Возможно на разных концах отрезка использовать условия различных типов.

Метод стрельбы:

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

Пусть надо решить краевую задачу краевыми условиями 1-го рода на отрезке $[a, b]$. Вместо исходной задачи формулируется задача Коши с уравнением $y'' = f(x, y, y')$ и с начальными условиями

$$y(a) = y_0\, y'(b) = \eta$$

Положим сначала некоторое начальное значение параметру$\eta = \eta_0$, после чего решим каким либо методом задачу Коши. 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. 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. Другими словами решение исходной задачи эквивалентно нахождению корня уравнения

$$\Phi(\eta) = y(b, y_0, \eta) - y_1 = 0$$

Следующее значение искомого корня определяется по соотношению

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

Конечно-разностный метод:

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

$$y'' + p(x)y' + q(x)y = f(x)$$

$$y(a) = y_0, y(b) = y_1$$

Введем разностную аппроксимацию производных следующим образом:

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

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

Подставляя аппроксимации производных в уравнение, получим систему уравнений для нахождения y_k:

$$\{y_0 = y_a \; \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} \; y_N = y_b \; + p(x_k)\frac{y_{k+1} - y_{k-1}}{2h} + q(x_k)y_k = f(x_k), \;\; k = \underline{1, N-1}$$

Код программы

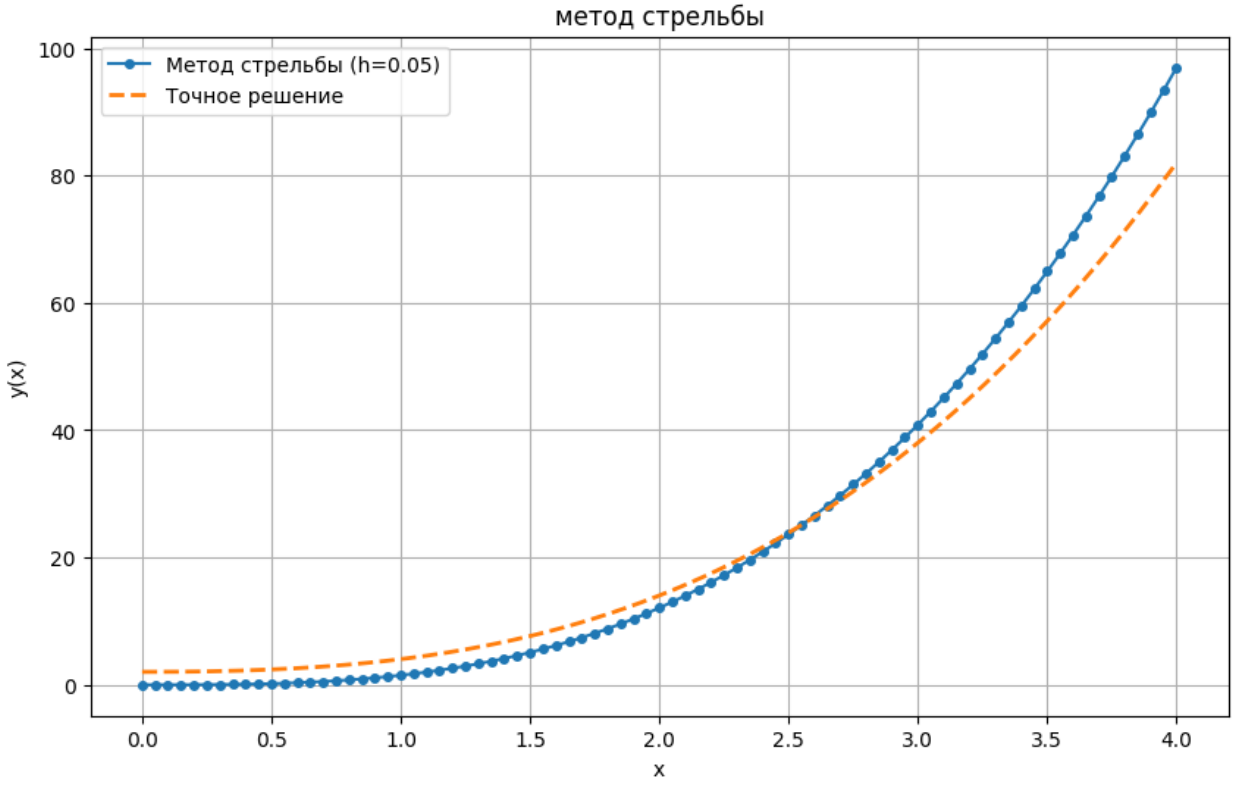
```
import numpy as np
import matplotlib.pyplot as plt


def generate_points(a, b, h):
  points = []
  current = a
  while current < b:
    points.append(current)
    current += h
  points.append(b)
  return points


def runge_romberg(y_h, y_h2, p):
  return [abs(yh - yh2) / (2**p - 1) for yh, yh2 in zip(y_h, y_h2)]
```

```
# Правая часть уравнения: y'' = [4(x^2+3)y' - 6x y] / [x(x^2+6)]
def f(x, y1, y2):
  if abs(x) < 1e-12:
    return -y1
  return (4 * (x**2 + 3) * y2 - 6 * x * y1) / (x * (x**2 + 6))

def exact_solution(x):
  return x**3 + x**2 + 2

def runge_kutta_method(a, b, h, y1_0, y2_0):
  xs = generate_points(a, b, h)
  y1s = [y1_0]
  y2s = [y2_0]
  for i in range(1, len(xs)):
    x = xs[i - 1]
    y1 = y1s[-1]
    y2 = y2s[-1]

    k1 = h * y2
    l1 = h * f(x, y1, y2)

    k2 = h * (y2 + l1 / 2)
    l2 = h * f(x + h / 2, y1 + k1 / 2, y2 + l1 / 2)

    k3 = h * (y2 + l2 / 2)
    l3 = h * f(x + h / 2, y1 + k2 / 2, y2 + l2 / 2)

    k4 = h * (y2 + l3)
    l4 = h * f(x + h, y1 + k3, y2 + l3)

    y1_next = y1 + (k1 + 2 * k2 + 2 * k3 + k4) / 6
    y2_next = y2 + (l1 + 2 * l2 + 2 * l3 + l4) / 6

    y1s.append(y1_next)
    y2s.append(y2_next)
```

```
  return xs, y1s, y2s


def shooting_method(h, beta, a=0, b=4):
  def phi(s):
    xs, y1s, y2s = runge_kutta_method(a, b, h, y1_0=s, y2_0=0.0)
    y_b = y1s[-1]
    y_prime_b = y2s[-1]
    return y_b - y_prime_b - beta

  s0, s1 = 0.0, 5.0
  for iter_count in range(20):
    f0, f1 = phi(s0), phi(s1)
    if abs(f1) < 1e-12:
      break
    s2 = s1 - f1 * (s1 - s0) / (f1 - f0)
    s0, s1 = s1, s2

  xs, y1s, y2s = runge_kutta_method(a, b, h, y1_0=s1, y2_0=0.0)
  return iter_count, xs, y1s, s1

h1 = 0.1
h2 = h1 / 2

iter1, x1, y1, s1 = shooting_method(h1, beta=26)
iter2, x2, y2, s2 = shooting_method(h2, beta=26)

y2_on_h1_grid = [y2[i] for i in range(0, len(y2), 2)]
rr_error = runge_romberg(y1, y2_on_h1_grid, p=4)

x_exact = np.linspace(0, 4, 200)
y_exact = exact_solution(x_exact)

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

```
plt.plot(x2[:-1], y2[:-1], 'o-', label='Метод стрельбы (h=0.05)',
markersize=4)

plt.plot(x_exact, y_exact, '--', label='Точное решение', linewidth=2)

plt.xlabel('x')

plt.ylabel('y(x)')

plt.title('метод стрельбы')

plt.grid(True)

plt.legend()

plt.show()


print("Сравнение решений на сетке h=0.1:")

for i in range(len(x1)):

  x_val = x1[i]

  y_num = y1[i]

  y_true = exact_solution(x_val)

  abs_err = abs(y_true - y_num)

  rr_est = rr_error[i] if i < len(rr_error) else float('nan')

  print(f"x = {x_val:4.1f} | y_точн = {y_true:8.5f} | y_числ = {y_num:8.5f} |
"

      f"абс. ош. = {abs_err:.2e} | P-P ош. = {rr_est:.2e}")
```

Вывод программы




```
Сравнение решений на сетке h=0.1:
x =  0.0 | y_точн =  2.00000 | y_числ = -0.07171 | абс. ош. = 2.07e+00 | P-P ош. = 2.58e-03
x =  0.1 | y_точн =  2.01100 | y_числ = -0.07040 | абс. ош. = 2.08e+00 | P-P ош. = 2.58e-03
x =  0.2 | y_точн =  2.04800 | y_числ = -0.05994 | абс. ош. = 2.11e+00 | P-P ош. = 2.56e-03
x =  0.3 | y_точн =  2.11700 | y_числ = -0.03044 | абс. ош. = 2.15e+00 | P-P ош. = 2.46e-03
x =  0.4 | y_точн =  2.22400 | y_числ =  0.02800 | абс. ош. = 2.20e+00 | P-P ош. = 2.23e-03
x =  0.5 | y_точн =  2.37500 | y_числ =  0.12526 | абс. ош. = 2.25e+00 | P-P ош. = 1.83e-03
x =  0.6 | y_точн =  2.57600 | y_числ =  0.27123 | абс. ош. = 2.30e+00 | P-P ош. = 1.20e-03
x =  0.7 | y_точн =  2.83300 | y_числ =  0.47579 | абс. ош. = 2.36e+00 | P-P ош. = 2.82e-04
x =  0.8 | y_точн =  3.15200 | y_числ =  0.74882 | абс. ош. = 2.40e+00 | P-P ош. = 9.66e-04
x =  0.9 | y_точн =  3.53900 | y_числ =  1.10021 | абс. ош. = 2.44e+00 | P-P ош. = 2.60e-03
x =  1.0 | y_точн =  4.00000 | y_числ =  1.53984 | абс. ош. = 2.46e+00 | P-P ош. = 4.66e-03
x =  1.1 | y_точн =  4.54100 | y_числ =  2.07760 | абс. ош. = 2.46e+00 | P-P ош. = 7.21e-03
x =  1.2 | y_точн =  5.16800 | y_числ =  2.72338 | абс. ош. = 2.44e+00 | P-P ош. = 1.03e-02
x =  1.3 | y_точн =  5.88700 | y_числ =  3.48705 | абс. ош. = 2.40e+00 | P-P ош. = 1.40e-02
x =  1.4 | y_точн =  6.70400 | y_числ =  4.37850 | абс. ош. = 2.33e+00 | P-P ош. = 1.83e-02
x =  1.5 | y_точн =  7.62500 | y_числ =  5.40762 | абс. ош. = 2.22e+00 | P-P ош. = 2.33e-02
x =  1.6 | y_точн =  8.65600 | y_числ =  6.58428 | абс. ош. = 2.07e+00 | P-P ош. = 2.90e-02
x =  1.7 | y_точн =  9.80300 | y_числ =  7.91838 | абс. ош. = 1.88e+00 | P-P ош. = 3.56e-02
x =  1.8 | y_точн = 11.07200 | y_числ =  9.41980 | абс. ош. = 1.65e+00 | P-P ош. = 4.30e-02
x =  1.9 | y_точн = 12.46900 | y_числ = 11.09843 | абс. ош. = 1.37e+00 | P-P ош. = 5.12e-02
x =  2.0 | y_точн = 14.00000 | y_числ = 12.96414 | абс. ош. = 1.04e+00 | P-P ош. = 6.05e-02
x =  2.1 | y_точн = 15.67100 | y_числ = 15.02682 | абс. ош. = 6.44e-01 | P-P ош. = 7.07e-02
x =  2.2 | y_точн = 17.48800 | y_числ = 17.29636 | абс. ош. = 1.92e-01 | P-P ош. = 8.20e-02
x =  2.3 | y_точн = 19.45700 | y_числ = 19.78265 | абс. ош. = 3.26e-01 | P-P ош. = 9.43e-02
...
x =  3.7 | y_точн = 66.34300 | y_числ = 82.88372 | абс. ош. = 1.65e+01 | P-P ош. = 4.12e-01
x =  3.8 | y_точн = 71.31200 | y_числ = 89.80726 | абс. ош. = 1.85e+01 | P-P ош. = 4.47e-01
x =  3.9 | y_точн = 76.52900 | y_числ = 97.10569 | абс. ош. = 2.06e+01 | P-P ош. = 4.84e-01
x =  4.0 | y_точн = 82.00000 | y_числ = 104.78889 | абс. ош. = 2.28e+01 | P-P ош. = 5.22e-01
```