Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Лабораторные работы по курсу «Численные методы»

Выполнил: Попов М. Р.
Преподаватель: Демидова О. Л.
Вариант: 20
Оценка:

Москва, 2023

Оглавление

Лабораторная работа №1

О программе

Программа написана на языке Python версии 3.10. Для решения задач был реализован класс MyMatrix, содержащий основные операции для работы с матрицами, также использовался модуль numpy.

Инструкция к запуску

В аргументах командной строки указываем номера заданий, которые необходимо выполнить. Все входные данные хранятся в директории tests в формате json.

Пример для выполнения всех заданий: python main.py 1 2 3 4 5

Алгоритм LU-разложения матрицы

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:

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

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

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

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

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

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

$$L = \left(1\ 0\ 0\ 0\ 0\ 0\ \mu_2^{(1)}\ 1\ 0\ 0\ 0\ 0\ \mu_3^{(1)}\ \mu_3^{(2)}\ 1\ 0\ 0\ 0\ \ldots\ \ldots\ \mu_{k+1}^{(k)}\ 1\ 0\ 0\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\right.$$

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:

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

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

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

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.

Входные данные:

$$\begin{cases} 7 \cdot x_1 + 8 \cdot x_2 + 4 \cdot x_3 - 6 \cdot x_4 = -126 \\ -x_1 + 6 \cdot x_2 - 2 \cdot x_3 - 6 \cdot x_4 = -42 \\ 2 \cdot x_1 + 9 \cdot x_2 + 6 \cdot x_3 - 4 \cdot x_4 = -115 \\ 5 \cdot x_1 + 9 \cdot x_2 + x_3 + x_4 = -67 \end{cases}$$

Метод для LU-разложения:

```
def decomposite(matrix):
matrix = array(matrix)
result = zeros((matrix.shape[0], matrix.shape[1]), dtype='float64')
result[0] = matrix[0]
for i in range(1, matrix.shape[0]):
    result[i] = matrix[i] + (- matrix[i][0] / result[0][0]) * result[0]
    result[i][0] = matrix[i][0] / result[0][0]
```

```
    for k in range(1, matrix.shape[0] - 1):
        for i in range(k + 1, matrix.shape[0]):
            result[i][k + 1:] = result[i][k + 1:] + (- result[i][k] / result[k][k]) * result[k][k + 1:]
            result[i][k] = result[i][k] / result[k][k]

    return round(result, ROUND)
```

Функция для решения СЛАУ с помощью LU-разложения:

```
def solve_lu_one_matrix(lu, b):
    size = lu.shape[0]
    solutions = array([0] * size, dtype='float64')
    solutions[-1] = b[-1] / lu[-1][-1]
    for i in range(size - 2, -1, -1):
        solutions[i] = b[i]
        for j in range(i + 1, size):
            solutions[i] -= lu[i][j] * solutions[j]
        solutions[i] = round(solutions[i] / lu[i][i], ROUND)
    solutions = round(solutions, ROUND)
    return reshape(solutions, (1, solutions.shape[0]))
```

Результат:

```
----------01-01----------

LU decompose:
L:
1.0                  0                   0                   0
-0.14285714285714285  1.0                 0                   0
0.2857142857142857   0.9400000000000001  1.0                 0
0.7142857142857143   0.46                -0.1935483870967742  1.0
U:
7                    8                   4                   -6
0.0                  7.142857142857142   -1.4285714285714286  -6.857142857142857
0.0                  0.0                 6.200000000000001   4.16
0.0                  0.0                 0.0                 9.24516129032258

LU decompose check: OK decompose correct

Determinant: 2866.0000000000005

Inversed matrix:
0.14654570830425678   -0.05931612002791349   -0.1228192602930914   0.03210048848569434
-0.07048150732728543  0.052337752965806006   0.04954640614096301   0.08932309839497558
-0.00523377529658058  -0.10502442428471735   0.14724354501046752   -0.07257501744591766
-0.0931612002791347   -0.0694347522679693    0.020935101186322403  0.10816468946266575

Inversion check: OK inversion correct

System solution:
 -4.000000000000004
-4.999999999999999
-6.999999999999997
5.0

Solution check: OK solution correct
-------------------------
```

Метод прогонки

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

$$\{b_1x_1 + c_1x_2 = d_1\ a_2x_1 + b_2x_2 + c_2x_3 = d_2\ a_3x_2 + b_3x_3 + c_3x_4 = d_3\ \dots\ a_{n-1}x_{n-2} + b$$

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

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

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

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

$$P_i = \frac{-c_i}{b_i + a_iP_{i-1}};\ \ Q_i = \frac{d_i - a_iQ_{i-1}}{b_i + a_iP_{i-1}}$$

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

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

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

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

.....

$$x_1 = P_1x_2 + Q_1$$

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

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

Входные данные:

$$\begin{cases} -6 \cdot x_1 + 6 \cdot x_2 = 30 \\ 2 \cdot x_1 + 10 \cdot x_2 - 7 \cdot x_3 = -31 \\ -8 \cdot x_2 + 18 \cdot x_3 + 9 \cdot x_4 = 108 \\ 6 \cdot x_3 - 17 \cdot x_4 - 6 \cdot x_5 = -114 \\ 9 \cdot x_4 + 14 \cdot x_5 = 124 \end{cases}$$

Метод решения СЛАУ с трёхдиагональной матрицей:

```
def solve_system_tridiagonal(self, b):
    """
    solves self * x = b, self is tridiagonal matrix
    :param b: free members
    :return: solution in column
    """
    to_use = True
    n = len(self)
    eq = 0
    leq = 0
    for i in range(1, n-1):
        if abs(self[i][1]) < abs(self[i][0]) + abs(self[i][2]) or abs(self[i][1]) < abs(self[i-1][-1]) + abs(self[i+1][0]):
            eq += 1
            to_use = False
        else:
            leq += 1
    to_use *= leq < eq
    p, q = [0] * n, [0] * n
    ans = [0] * n
    p[0] = self[0][1] / -self[0][0]
    q[0] = b[0] / self[0][0]
    for i in range(1, n-1):
        p[i] = -self[i][2] / (self[i][1] + self[i][0]*p[i-1])
        q[i] = (b[i] - self[i][0]*q[i-1]) / (self[i][1] + self[i][0]*p[i-1])
    p[-1] = 0
    q[-1] = (b[-1] - self[-1][0]*q[-2]) / (self[-1][1] + self[-1][0]*p[-2])
    ans[-1] = q[-1]
    for i in range(n-1, 0, -1):
        ans[i-1] = p[i-1] * ans[i] + q[i-1]
    return MyMatrix(ans).transposed(), to_use
```

Результат:

```
----------01-02----------

System solution:
 -5.0
0.0
3.0
6.0
5.0

Solution check: OK solution correct
--------------------------
```

Итерационные методы решения СЛАУ

Рассмотрим СЛАУ

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

с невырожденной матрицей.

Приведем СЛАУ к эквивалентному виду

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

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

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

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

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

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

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

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

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

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 $x_1^{k+1}, x_2^{k+1}, ..., x_{i-1}^{k+1}$, уже вычисленные на (k+1)-й итерации. Тогда метод Зейделя для известного вектора на k-ой итерации имеет вид:

$$\{x_1^{k+1} = \beta_1 + \alpha_{11}x_1^k + \alpha_{12}x_2^k + \ldots + \alpha_{1n}x_n^k \; x_2^{k+1} = \beta_2 + \alpha_{21}x_1^k + \alpha_{22}x_2^k + \ldots + \alpha_{2n}x_n^k \; x_3^k$$

Входные данные:

$$\begin{cases} 10 \cdot x_1 - x_2 - 2 \cdot x_3 + 5 \cdot x_4 = -99 \\ 4 \cdot x_1 + 28 \cdot x_2 + 7 \cdot x_3 + 9 \cdot x_4 = 0 \\ 6 \cdot x_1 + 5 \cdot x_2 - 23 \cdot x_3 + 4 \cdot x_4 = 67 \\ x_1 + 4 \cdot x_2 + 5 \cdot x_3 - 15 \cdot x_4 = 58 \end{cases}$$

Метод простых итераций

```
def solve_system_iterative(a, b, eps):
    """
    Uses iterative method to solve Ax=b
    :param a: system
    :param b: free members
    :param eps:
    :return: x and the number of iterations
    """
    n = a.shape[0]
    alpha = np.zeros_like(a, dtype='float')
    beta = np.zeros_like(b, dtype='float')
    for i in range(n):
        for j in range(n):
            if i == j:
                alpha[i][j] = 0
            else:
                alpha[i][j] = -a[i][j] / a[i][i]
        beta[i] = b[i] / a[i][i]
    iterations = 0
    cur_x = np.copy(beta)
    while True:
        prev_x = np.copy(cur_x)
        cur_x = alpha @ prev_x + beta
        iterations += 1
        if norm(prev_x - cur_x) <= eps:
            break
    return cur_x, iterations
```

Метод Зейделя

```
def solve_system_zeidel(a, b, eps):
    """
    Uses zeidel method to solve ax=b
    :param a: system
    :param b: free members
    :param eps:
    :return: x and number of iterations
    """
    n = a.shape[0]
    alpha = np.zeros_like(a, dtype='float')
```

```
    beta = np.zeros_like(b, dtype='float')
    for i in range(n):
        for j in range(n):
            if i == j:
                alpha[i][j] = 0
            else:
                alpha[i][j] = -a[i][j] / a[i][i]
        beta[i] = b[i] / a[i][i]
    iterations = 0
    cur_x = np.copy(beta)
    while True:
        prev_x = np.copy(cur_x)
        cur_x = zeidel_multiplication(alpha, prev_x, beta)
        iterations += 1
        if norm(prev_x - cur_x) <= eps:
            break
    return cur_x, iterations
```

Результат:

```
----------01-03----------

Iteration method:
-7.9999983802646275
4.000001821811469
-4.999999133973699
-4.999999724733941

Number of iterations:  18

Iterations method check: OK solution correct

Zeidel method:
-7.999999760913948
3.999999747554181
-5.000000081098047
-5.000000078412497

Number of iterations:  7

Zeidel method check: OK solution correct
------------------------
```

Метод вращений

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 вращений:

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:

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

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:

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

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

Входные данные: $\begin{pmatrix} -7 & -9 & 1 \\ -9 & 7 & 2 \\ 1 & 2 & 9 \end{pmatrix}$

Метод вращений:

```
def rotation(A, eps):
    """
    :return: eigen values, eigen vectors, number of iterations
    """
    n = A.shape[0]
    A_i = np.copy(A)
    eigen_vectors = np.eye(n)
    iterations = 0
    while matrix_norm(A_i) > eps:
        i_max, j_max = find_max_upper_element(A_i)
        if A_i[i_max][i_max] - A_i[j_max][j_max] == 0:
            phi = np.pi / 4
        else:
            phi = 0.5 * np.arctan(2 * A_i[i_max][j_max] / (A_i[i_max][i_max] -
                                                           A_i[j_max][j_max]))
        U = np.eye(n)
        U[i_max][j_max] = -np.sin(phi)
        U[j_max][i_max] = np.sin(phi)
        U[i_max][i_max] = np.cos(phi)
        U[j_max][j_max] = np.cos(phi)
        A_i = U.T @ A_i @ U
        eigen_vectors = eigen_vectors @ U
        iterations += 1
    eigen_values = np.array([A_i[i][i] for i in range(n)])
    return eigen_values, eigen_vectors, iterations
```

Результат:

```
----------01-04----------
```

Попов Матвей, М8О-308Б-20

Eigen values:
-11.555975526027277 12.03659015248972 8.519385373537553

Eigen vectors:
0.8926937456893264 -0.36892548536585096 0.2588278629798799
0.44229229437545153 0.8273931428156167 -0.34611864087014355
-0.0864604114219863 0.4234555153369333 0.9017815831937761

Number of iterations: 6

QR-алгоритм нахождения собственных значений матриц

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:

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

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

$$\widetilde{b} = Hb,\ b = \left(b_1, b_2, ..., b_n\right)^T,\ \widetilde{b} = \left(\widetilde{b}_1,\ 0, ..., 0\right)^T$$

Тогда вектор ν определится следующим образом:

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

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: $\left(\sum_{l=m+1}^{n} \left(a_{lm}^{(k)}\right)^2\right)^{\frac{1}{2}} \leq \varepsilon$. При этом соответствующее собственное значение принимается равным диагональному элементу данного столбца.

Входные данные: $\begin{pmatrix} 6 & 5 & -6 \\ 4 & -6 & 9 \\ -6 & 6 & 1 \end{pmatrix}$

Матрица Хаусхолдера:

```
def get_householder_matrix(A, col_num):
    n = A.shape[0]
    v = np.zeros(n)
    a = A[:, col_num]
    v[col_num] = a[col_num] + sign(a[col_num]) * norm(a[col_num:])
    for i in range(col_num + 1, n):
        v[i] = a[i]
    v = v[:, np.newaxis]
    H = np.eye(n) - (2 / (v.T @ v)) * (v @ v.T)
    return H
```

QR-разложение:

```
def qr_decompose(A):
    """
```

```
    A = QR
    :return: Q, R
    """
    n = A.shape[0]
    Q = np.eye(n)
    A_i = np.copy(A)
    for i in range(n - 1):
        H = get_householder_matrix(A_i, i)
        Q = Q @ H
        A_i = H @ A_i
    return Q, A_i
```

Результат:

```
----------01-05----------

Eigen values:
-13.276293090340536     9.861550938470351     4.4147466394824955
-------------------------
```

Лабораторная работа №2

О программе

Программа написана на языке Python версии 3.10. Для реализации решения систем нелинейных уравнений использовался модуль numpy.

Инструкция к запуску

В аргументах командной строки указываем номера заданий, которые необходимо выполнить. Все уравнения заданы в качестве функций внутри программы.

Пример для выполнения всех заданий: python main.py 1 2

Решение нелинейных уравнений

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:

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

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

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

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:

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

Для завершения итерационного процесса используется правило:

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

Уравнение: $\tan tan\, x \; - 5x^2 + 1 = 0, x \in [-1, 1]$

График вблизи корня:

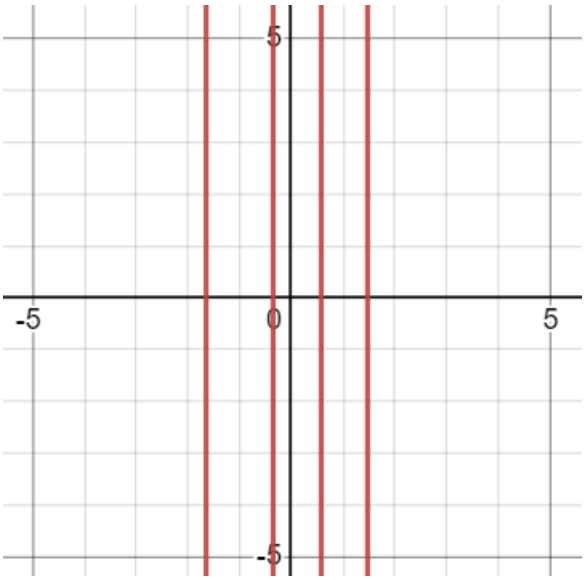

Рисунок 1

Метод простых итераций:

```
def iterations(f, phi, l, r, eps):
    """
    :param f: f(x) = 0
    :param phi: phi(x) = x
    :param l, r: [l, r]
    :return: x and number of iterations
    """
    done = check_iterations(f, phi, l, r)
    x_prev = (l + r) * 0.5
    i = 0
    while i <= 50:
        i += 1
        x = phi(x_prev)
        if abs(f(x) - f(x_prev)) < eps:
            return x, i
        x_prev = x
    return 0, -1, done
```

Метод Ньютона:

```
def newton(f, df, l, r, eps):
    """
    :param f: f(x) = 0
    :param df: f'(x)
    :param l, r: [l, r]
    :return: x and number of iterations
    """
    done = check_newton(f, df, l, r)
    x_prev = (l + r) * 0.5
    i = 0
    while i <= 50:
        i += 1
        x = x_prev - f(x_prev) / df(x_prev)
        if abs(f(x) - f(x_prev)) < eps:
            return x, i
        x_prev = x
    return 0, -1, done
```

Результат:

----------02-01----------

tg(x) - 5x^2 + 1 = 0

Iterations method:
x = 1.4690184716538488
f(x) = 0.0012942886922164831
Number of iterations: 4

Newton method:
x = 1.4690027221370237
f(x) = 2.155161062944444e-07
Number of iterations: 4

Решение систем нелинейных уравнений

Метод простых итераций:

Решение системы нелинейных уравнений вида

$$\{f_1\left(x_1, x_2, x_3, ..., x_n\right) = 0\ f_2\left(x_1, x_2, x_3, ..., x_n\right) = 0\ \ ...\ \ f_n\left(x_1, x_2, x_3, ..., x_n\right) = 0$$

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

Для использования метода итераций система уравнений записывается в эквивалентной форме:

$$\{x_1 = \phi_1\left(x_1, x_2, x_3, ..., x_n\right)\ x_2 = \phi_2\left(x_1, x_2, x_3, ..., x_n\right)\ \ ...\ \ x_n = \phi_n\left(x_1, x_2, x_3, ..., x_n\right),$$

где ϕ_i – итерирующие непрерывно дифференцируемые функции.

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

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

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:

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

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.

$$J\left(X^{(k)}\right) = \left| \frac{\partial f_1\left(X^{(k)}\right)}{\partial x_1} \; \frac{\partial f_1\left(X^{(k)}\right)}{\partial x_2} \; \dots \; \frac{\partial f_1\left(X^{(k)}\right)}{\partial x_n} \; \frac{\partial f_2\left(X^{(k)}\right)}{\partial x_1} \; \frac{\partial f_2\left(X^{(k)}\right)}{\partial x_2} \; \dots \; \frac{\partial f_2\left(X^{(k)}\right)}{\partial x_n} \; \dots \; \dots \; \dots \; \dots \right.$$

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

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

Система: $\begin{cases} x_1^2 - 2\lg x_2 - 1 = 0, \\ x_1^2 - a x_1 x_2 + a = 0. \end{cases}$, $a = 2$

График вблизи корня:

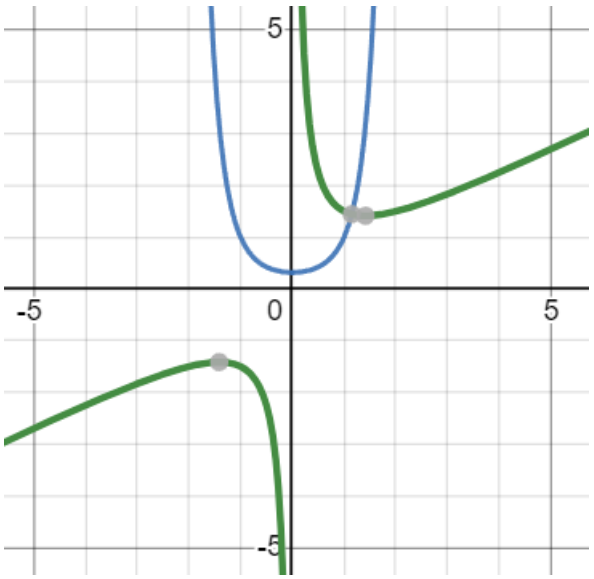

Рисунок 2

Метод простых итераций:

```
def iterations(f1, f2, phi1, phi2, dphi1_dx1, dphi1_dx2, dphi2_dx1, dphi2_dx2, l1, r1, l2, r2, eps):
    if not check_system(dphi1_dx1, dphi1_dx2, dphi2_dx1, dphi2_dx2):
        return 0, -1
    x_prev = [(l1 + r1) * 0.5, (l2 + r2) * 0.5]
    q = get_q(l1, r1, l2, r2, dphi1_dx1, dphi1_dx2, dphi2_dx1, dphi2_dx2)
    if q >= 1:
        return 0, -1
    i = 0
    while i <= 50:
        i += 1
        x = [phi1(x_prev), phi2(x_prev)]
        if q / (1 - q) * l_inf_norm([(x[i] - x_prev[i]) for i in range(len(x))]) < eps:
            return x, i
        x_prev = x
    return 0, -1
```

Метод Ньютона:

```
def newton(f1, f2, df1_dx1, df1_dx2, df2_dx1, df2_dx2, l1, r1, l2, r2, eps):
    x_prev = np.array([(l1 + r1) / 2, (l2 + r2) / 2])
    jacobi = [[df1_dx1(x_prev), df1_dx2(x_prev)],
              [df2_dx1(x_prev), df2_dx2(x_prev)]]
    jacobi_inversed = np.linalg.inv(np.array(jacobi))
    i = 0
    while i <= 50:
        i += 1
        x = x_prev - jacobi_inversed @ np.array([f1(x_prev), f2(x_prev)])
        if l_inf_norm([(x[i] - x_prev[i]) for i in range(len(x))]) < eps:
            return x, i
        x_prev = x

    return 0, -1
```

Метод Зейделя:

```
def zeidel(x_0, eps, phi1, phi2, dphi1_dx1, dphi1_dx2, dphi2_dx1, dphi2_dx2):
    if not check_system(dphi1_dx1, dphi1_dx2, dphi2_dx1, dphi2_dx2):
        return 0, -1
    phi_x_i = x_0
    i = 0
    while i <= 50:
        x_0 = phi_x_i.copy()
        x_k = phi1(x_0)
        phi_x_i = [x_k, phi2([x_k, x_0[1]])]
        i += 1
        if max(abs(x_0[0] - phi_x_i[0]), abs(x_0[1] - phi_x_i[1])) < eps:
            return phi_x_i, i
    return 0, -1
```

Результат:

----------02-02----------

Equation system:
x1^2 - 2lg(x2) - 1 = 0
x1^2 - ax1x2 + a = 0, a = 2

Iterations method:
x1 = 1.147850166884402
x2 = 1.446847372571988
f1(x1, x2) = -0.003285434305210777
f2(x1, x2) = -0.003967990509480046
Number of iterations: 3

Newton method:
x1 = 1.1491300730110081
x2 = 1.440946754815555
f1(x1, x2) = 0.0032040581667982515
f2(x1, x2) = 0.008829425565937132
Number of iterations: 3

Zeidel method:
x1 = 1.149163693172696
x2 = 1.4447798922965516
f1(x1, x2) = 0.000973816257786364
f2(x1, x2) = 0.0
Number of iterations: 3

Лабораторная работа №3

О программе

Программа написана на языке Python версии 3.10. Для отрисовки графиков применялся модуль matplotlib.

Инструкция к запуску

В аргументах командной строки указываем номера заданий, которые необходимо выполнить. Все уравнения заданы в качестве функций внутри программы.

Пример для выполнения всех заданий: python main.py 1 2 3 4 5

Интерполяция

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

$$\varphi\left(x_i, a_0, ..., a_n\right) = f(x_i)$$

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 $l_i\left(x_j\right) = \{1, \text{ при } i = j\ 0, \text{ при } i \neq j$ 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:

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

разделенные разности второго порядка определяются через разделенные разности первого порядка:

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

Разделенная разность $n - k + 2$ определяется соостношениями

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

Интерполяционный многочлен Ньютона может быть представлен в виде:

$$P_n(x) = f\left(x_0\right) + \left(x - x_0\right) f\left(x_1, x_0\right) + \left(x - x_0\right)\left(x - x_1\right) f\left(x_0, x_1, x_2\right) + \ldots + \left(x - x_0\right)\left(x$$

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.

Входные данные: $y = \ x + x$, а) $X_i = -\ 0.4, \ -\ 0.1, \ 0.2, \ 0.5$; б)

$X_i = -\ 0.4, \ 0, \ 0.2, \ 0.5; X^* = 0.1$

Метод Лагранжа:

```
def lagrange(f, x, test_point):
    y = [f(t) for t in x]
    assert len(x) == len(y)
    polynom_str = 'L(x) ='
    polynom_test_value = 0
    for i in range(len(x)):
        cur_enum_str = ''
        cur_enum_test = 1
        cur_denom = 1
        for j in range(len(x)):
            if i == j:
                continue
            cur_enum_str += f'(x-{x[j]:.2f})'
            cur_enum_test *= (test_point[0] - x[j])
            cur_denom *= (x[i] - x[j])

        polynom_str += f'+{(y[i] / cur_denom):.2f}' + cur_enum_str
        polynom_test_value += y[i] * cur_enum_test / cur_denom

    return format_polynom(polynom_str), abs(polynom_test_value - test_point[1])
```

Метод Ньютона:

```
def newton(f, x, test_point):
    y = [f(t) for t in x]
    assert len(x) == len(y)

    n = len(x)
    coefs = [y[i] for i in range(n)]
    for i in range(1, n):
        for j in range(n - 1, i - 1, -1):
            coefs[j] = float(coefs[j] - coefs[j - 1]) / float(x[j] - x[j - i])

    polynom_str = 'P(x) = '
    polynom_test_value = 0

    cur_multipliers_str = ''
    cur_multipliers = 1
    for i in range(n):
        polynom_test_value += cur_multipliers * coefs[i]
        if i == 0:
            polynom_str += f'{coefs[i]:.2f}'
        else:
            polynom_str += '+' + cur_multipliers_str + '*' + f'{coefs[i]:.2f}'

        cur_multipliers *= (test_point[0] - x[i])
        cur_multipliers_str += f'(x-{x[i]:.2f})'

    return format_polynom(polynom_str), abs(polynom_test_value - test_point[1])
```

Результат:

----------03-01----------

Lagrange interpolation:

Points A polynom: L(x) = - 9.77(x + 0.10)(x - 0.20)(x - 0.50) + 29.09(x + 0.40)(x - 0.20)(x - 0.50) - 29.06(x + 0.40)(x + 0.10)(x - 0.50) + 9.55(x + 0.40)(x + 0.10)(x - 0.20)
Error in X*: 0.00011154315104722201

Points B polynom: L(x) = - 7.33(x - 0.00)(x - 0.20)(x - 0.50) + 39.27(x + 0.40)(x - 0.20)(x - 0.50) - 43.60(x + 0.40)(x - 0.00)(x - 0.50) + 11.46(x + 0.40)(x - 0.00)(x - 0.20)

Попов Матвей, М8О-308Б-20

Error in X*: 7.377452117407479e-05

Newton interpolation:

Points A polynom: P(x) = 1.58 + (x + 0.40) - 0.04 + (x + 0.40)(x + 0.10)*0.05 + (x + 0.40)(x + 0.10)(x - 0.20) - 0.19
Error in X*: 0.00011154315104744406

Points B polynom: P(x) = 1.58 + (x + 0.40) - 0.03 + (x + 0.40)(x - 0.00)*0.04 + (x + 0.40)(x - 0.00)(x - 0.20) - 0.19
Error in X*: 7.377452117407479e-05

Сплайн

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:

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

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

$$S(x) = a_i + b_i\left(x - x_{i-1}\right) + c_i\left(x - x_{i-1}\right)^2 + d_i\left(x - x_{i-1}\right)^3,$$

$$x_{i-1} \leq x \leq x_i,\quad i = \overline{1,n}$$

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.

Входные данные: $X^* = 0.1$

i	0	1	2	3	4
x_i	-0.4	-0.1	0.2	0.5	0.8
f_i	1.5823	1.5710	1.5694	1.5472	1.4435

Построение сплайна:

```
def spline_interpolation(x_i, f_i, x_):
    assert len(x_i) == len(f_i)
    n = len(x_i)
    h = [x_i[i] - x_i[i - 1] for i in range(1, len(x_i))]
    A = [[0 for _ in range(len(h)-1)] for _ in range(len(h)-1)]
    A[0][0] = 2 * (h[0] + h[1])
    A[0][1] = h[1]
    for i in range(1, len(A) - 1):
        A[i][0] = h[i-1]
        A[i][1] = 2 * (h[i-1] + h[i])
        A[i][2] = h[i]
    A[-1][-2] = h[-2]
    A[-1][-1] = 2 * (h[-2] + h[-1])

    m = [3.0 * ((f_i[i+1] - f_i[i]) / h[i] - (f_i[i] - f_i[i-1]) / h[i-1])
         for i in range(1, len(h))]

    c = [0] + tridiagonal_solve(A, m)

    a = [f_i[i - 1] for i in range(1, n)]

    b = [(f_i[i] - f_i[i-1]) / h[i-1] - (h[i-1] / 3.0) * (2.0 * c[i-1] + c[i])
         for i in range(1, len(h))]
    b.append((f_i[-1] - f_i[-2]) / h[-1] - (2.0 * h[-1] * c[-1]) / 3.0)

    d = [(c[i] - c[i-1]) / (3.0 * h[i-1]) for i in range(1, len(h))]
    d.append(-c[-1] / (3.0 * h[-1]))

    for interval in range(len(x_i)):
        if x_i[interval] <= x_ < x_i[interval + 1]:
            i = interval
            break
    y_test = s(a[i + 1], b[i + 1], c[i + 1], d[i + 1], x_ - x_i[i])
    return a, b, c, d, y_test
```

Результат:

```
----------03-02----------

[-0.4; -0.1)
s(x) = 1.5823 - 0.0464(x + 0.4000) + 0.0000(x + 0.4000)^2 + 0.0968(x + 0.4000)^3
[-0.1; 0.2)
s(x) = 1.571 - 0.0202(x + 0.1000) + 0.0871(x + 0.1000)^2 - 0.1249(x + 0.1000)^3
[0.2; 0.5)
s(x) = 1.5694 - 0.0017(x - 0.2000) - 0.0252(x - 0.2000)^2 - 0.7196(x - 0.2000)^3
[0.5; 0.8)
s(x) = 1.5472 - 0.2111(x - 0.5000) - 0.6729(x - 0.5000)^2 + 0.7476(x - 0.5000)^3
s(x*) = s(0.1000) = 1.5623

-------------------------
```

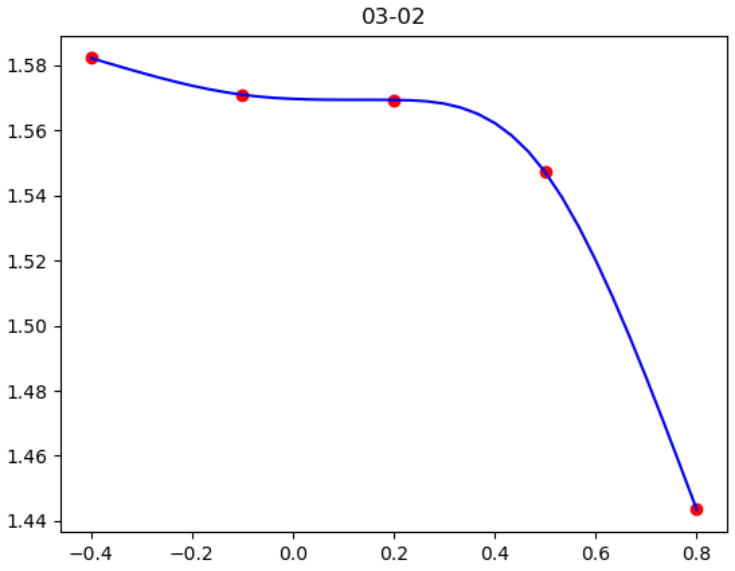

Рисунок 3

Метод наименьших квадратов

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

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

Эту систему для удобства преобразуют к следующему виду:

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

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.

Входные данные:

i	0	1	2	3	4	5
x_i	-0.7	-0.4	-0.1	0.2	0.5	0.8
y_i	1.6462	1.5823	1.571	1.5694	1.5472	1.4435

Метод наименьших квадратов:

```
def least_squares(x, y, n):
    assert len(x) == len(y)
    A = []
    b = []
    for k in range(n + 1):
        A.append([sum(map(lambda x: x**(i + k), x)) for i in range(n + 1)])
        b.append(sum(map(lambda x: x[0] * x[1]**k, zip(y, x))))
    lu = decomposite(A)
    return solve_lu_one_matrix(lu, b)
```

Результат:

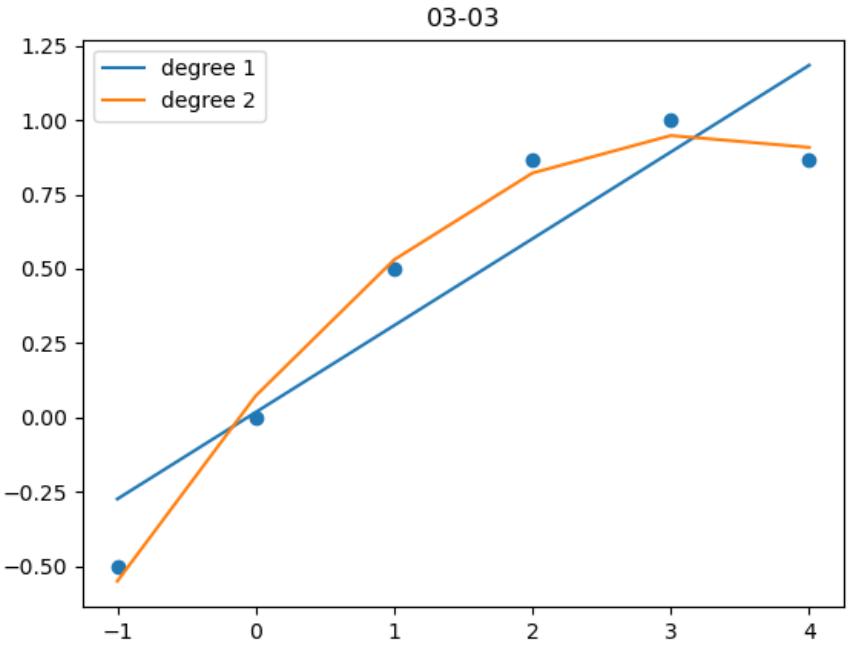

Рисунок 4

```
----------03-03----------

Least squares method, 1st degree
P(x) = 0.0184 + 0.2913x
```

Попов Матвей, М8О-308Б-20

Sum of squared errors = 0.2708179489276191

Least squares method, 2nd degree
P(x) = 0.0735 + 0.5396x - 0.0827x^2
Sum of squared errors = 0.2708179489276191

Численное дифференцирование

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 $\varphi(x, \bar{a})$, $y = \varphi(x, \bar{a}) + R(x)$, где $R(x)$ – остаточный член приближения, $\bar{a}$ – набор коэффициентов, вообще говоря, различный для каждого из рассматриваемых отрезков, и полагают, что $y'(x) \approx \varphi'(x, \bar{a})$. Наиболее часто в качестве приближающей функции $\varphi(x, \bar{a})$ берется интерполяционный многочлен $\varphi(x, \bar{a}) = P_n(x)$, 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

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

Входные данные: $X^* = 1.0$

i	0	1	2	3	4
x_i	-1.0	0.0	1.0	2.0	3.0
y_i	1.3562	1.5708	1.7854	2.4636	3.3218

Дифференциал первого порядка:

```
def df(x, y, x_):
    assert len(x) == len(y)
    for interval in range(len(x)):
        if x[interval] <= x_ < x[interval+1]:
            i = interval
            break

    a1 = (y[i+1] - y[i]) / (x[i+1] - x[i])
    a2 = ((y[i+2] - y[i+1]) / (x[i+2] - x[i+1]) - a1) / (x[i+2] - x[i]) * (2*x_ - x[i] - x[i+1])
    return a1 + a2
```

Дифференциал второго порядка:

```
def d2f(x, y, x_):
    assert len(x) == len(y)
    for interval in range(len(x)):
        if x[interval] <= x_ < x[interval+1]:
            i = interval
            break

    num = (y[i+2] - y[i+1]) / (x[i+2] - x[i+1]) - (y[i+1] - y[i]) / (x[i+1] - x[i])
    return 2 * num / (x[i+2] - x[i])
```

Результат:

```
----------03-04----------

f'(1.0) = 0.5882
f''(1.0) = 0.1800

-------------------------
```

Численное интегрирование

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

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

Входные данные: $y = \frac{\sqrt{x}}{4+3x},\ X_0 = 1,\ X_k = 5,\ h_1 = 1.0,\ h_2 = 0.5$

Метод прямоугольника:

```
def rectangle(f, l, r, h):
    if l > r:
        return None
```

```
    result = 0
    cur_x = l
    while cur_x < r:
        result += h*f((cur_x + cur_x + h)*0.5)
        cur_x += h
    return result
```

Метод трапеции:

```
def trapeze(f, l, r, h):
    if l > r:
        return None
    result = 0
    cur_x = l
    while cur_x < r:
        result += h*0.5*(f(cur_x + h) + f(cur_x))
        cur_x += h
    return result
```

Метод Симпсона:

```
def simpson(f, l, r, h):
    if l > r:
        return None
    result = 0
    cur_x = l + h
    while cur_x < r:
        result += f(cur_x - h) + 4*f(cur_x) + f(cur_x + h)
        cur_x += 2*h
    return result * h / 3
```

Метод Рунге-Ромберга-Ричардсона:

```
def runge_rombert(h1, h2, i1, i2, p):
    return i1 + (i1 - i2) / ((h2 / h1)**p - 1)
```

Результат:

```
----------03-05----------

Rectangle method
Step 1.0: 0.5318189388349329
Step 0.5: 0.5313885217114334

Trapeze method
Step 1.0: 0.5299284993159105
Step 0.5: 0.5308737190754216

Simpson method
Step 1.0: 0.5308999037021467
Step 0.5: 0.531188792328592

Runge Robert method
Rectangle: 0.5313270335509335
Trapeze: 0.5310087504696375
Simpson: 0.5312300621323699

-------------------------
```

Лабораторная работа №4

О программе

Программа написана на языке Python версии 3.10. Для отрисовки графиков применялся модуль matplotlib.

Инструкция к запуску

В аргументах командной строки указываем номера заданий, которые необходимо выполнить. Все уравнения заданы в качестве функций внутри программы.

Пример для выполнения всех заданий: python main.py 1 2

Численные методы решения задачи Коши

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 Ox 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 $y_{1ист}$. При этом допускается погрешность $|y_1 - y_{1ист}|$ 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:

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

Неявный метод Эйлера:

Неявный метод Эйлера (также известный как метод трапеций) является численным методом решения задачи Коши для обыкновенного дифференциального уравнения первого порядка. Он представляет собой один из классических методов решения дифференциальных уравнений, который имеет высокую точность и устойчивость при большом шаге интегрирования.

Идея метода заключается в замене производной функции на приближенную, используя формулу для среднего значения. В результате получается неявное уравнение, которое решается численно.

Метод Рунге-Кутты:

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

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

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

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

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

Метод Рунге-Кутты четвертого порядка:

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

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

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

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

Метод Адамса:

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

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

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

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

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

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 $z = y'(x)$:

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

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

Входные данные:

$x(x-1)y'' + \frac{1}{2}y' - \frac{3}{4}y = 0,$ $y(2) = \sqrt{2},$ $y'(2) = \frac{3}{2}\sqrt{2},$ $x \in [2, 3],\ h = 0.1$	$y = \lvert x \rvert^{3/2}$

Метод Эйлера:

```
def euler(f, g, y0, z0, borders, h):
    l, r = borders
    x = [i for i in np.arange(l, r + h, h)]
    y = [y0]
    z = z0
    for i in range(len(x) - 1):
        z += h * f(x[i], y[i], z)
        y.append(y[i] + h * g(x[i], y[i], z))
    return x, y
```

Неявный метод Эйлера:

```
def implicit_euler(f, y0, z0, borders, h):
    l, r = borders
    n = int((r - l) / h)
    x = [i for i in np.arange(l, r + h, h)]
    y = [y0]
    z = [z0]
    for i in range(1, n+1):
        t_i = l + i * h
        y_i = y[i-1] + h * z[i-1]
        z_i = z[i-1] + h * f(t_i, y_i, z[i-1])
        y.append(y_i)
        z.append(z_i)
    return x, y
```

Метод Рунге-Кутты:

```
def runge_kutta(f, g, y0, z0, borders, h, return_z=False):
    l, r = borders
    x = [i for i in np.arange(l, r + h, h)]
    y = [y0]
    z = [z0]
    for i in range(len(x) - 1):
        K1 = h * g(x[i], y[i], z[i])
        L1 = h * f(x[i], y[i], z[i])
        K2 = h * g(x[i] + 0.5 * h, y[i] + 0.5 * K1, z[i] + 0.5 * L1)
        L2 = h * f(x[i] + 0.5 * h, y[i] + 0.5 * K1, z[i] + 0.5 * L1)
        K3 = h * g(x[i] + 0.5 * h, y[i] + 0.5 * K2, z[i] + 0.5 * L2)
        L3 = h * f(x[i] + 0.5 * h, y[i] + 0.5 * K2, z[i] + 0.5 * L2)
        K4 = h * g(x[i] + h, y[i] + K3, z[i] + L3)
        L4 = h * f(x[i] + h, y[i] + K3, z[i] + L3)
        delta_y = (K1 + 2 * K2 + 2 * K3 + K4) / 6
        delta_z = (L1 + 2 * L2 + 2 * L3 + L4) / 6
        y.append(y[i] + delta_y)
        z.append(z[i] + delta_z)
    if not return_z:
        return x, y
    else:
        return x, y, z
```

Метод Адамса:

```
def adams(f, g, y0, z0, borders, h):
    x_runge, y_runge, z_runge = runge_kutta(f, g, y0, z0, borders, h,
                                            return_z=True)
    x = x_runge
    y = y_runge[:4]
    z = z_runge[:4]
    for i in range(3, len(x_runge) - 1):
        z_i = z[i] + h * (55 * f(x[i], y[i], z[i]) -
                          59 * f(x[i - 1], y[i - 1], z[i - 1]) +
                          37 * f(x[i - 2], y[i - 2], z[i - 2]) -
                          9 * f(x[i - 3], y[i - 3], z[i - 3])) / 24
        z.append(z_i)
        y_i = y[i] + h * (55 * g(x[i], y[i], z[i]) -
                          59 * g(x[i - 1], y[i - 1], z[i - 1]) +
                          37 * g(x[i - 2], y[i - 2], z[i - 2]) -
                          9 * g(x[i - 3], y[i - 3], z[i - 3])) / 24
        y.append(y_i)
    return x, y
```

Результат:

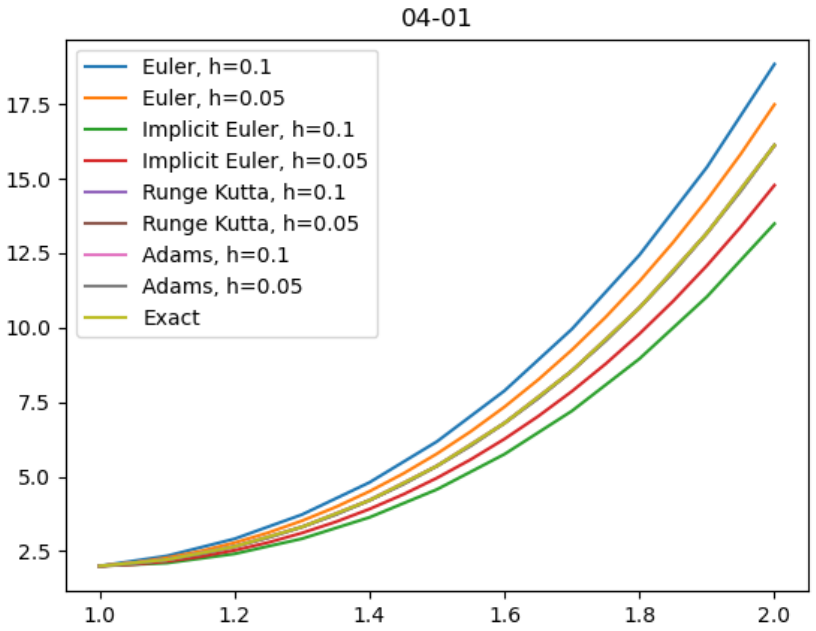

Рисунок 5

----------04-01----------

Errors
h = 0.1
Euler: 1.037467463239314
Implicit Euler: 0.9989754946292955
Runge Kutta: 0.00013213405805423406
Adams: 0.003064751545071283

h = 0.05
Euler: 0.5112637693826623
Implicit Euler: 0.5012695009091112
Runge Kutta: 1.0425143880652516e-05
Adams: 0.0005767686905690226

Runge Romberg
Euler: 0.13062440186188579
Implicit Euler: 0.12394525940744755
Runge Kutta: 3.4464222120195054e-05
Adams: 0.0008347860057395394

Численные методы решения краевой задачи для ОДУ

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

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

Следующее значение искомого корня определяется по соотношению

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

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:

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

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

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

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

Входные данные:

$xy'' - (2x + 1)y' + 2y = 0,$ $y'(0) = 4,$ $y'(1) - 2y(1) = -4$	$y(x) = 2x + 1 + e^{2x}$

Здесь $h = 0.1, p(x) = \frac{-(2x+1)}{x}, q(x) = \frac{2}{x}, f(x) = 0, N = 10$

После замены производных их разностными аналогами получим:

$$\left(1 + \frac{0.1\left(2x_k + 1\right)}{2x_k}\right)y_{k-1} + \left(-2 + 0.01\frac{2}{x_k}\right)y_k + \left(1 - \frac{0.1\left(2x_k + 1\right)}{2x_k}\right)y_{k+1} = 0$$

На левой границе аппроксимируем производную односторонней разностью 1-го порядка:

$$\frac{y_1 - y_0}{0.1} = 0$$

На правой границе аппроксимируем производную односторонней разностью 1-го порядка:

$$\frac{y_{10} - y_9}{0.1} - 2y_{10} = 0$$

Получим систему уравнений:

$$\{-0.2y_1 + 0.4y_2 = -1.6\ 1.35y_1 - 1.9y_2 + 0.65y_3 = 0\ 1.27y_2 - 1.93y_3 + 0.73$$

Решив систему методом прогонки, получаем:
$y_1 = 2.83,\ y_2 = 2.98,\ y_3 = 3.60,\ y_4 = 4.06,\ y_5 = 4.84,\ y_6 = 5.68,\ y_7 = 6.78,\ y_8$

Метод стрельбы:

```
def shoting_method(ddy, borders, bcondition1, bcondition2, h, f):
    y0 = diff_left(bcondition1, h, f)
    eta1 = 0.5
    eta2 = 2.0
    resolve1 = runge_kutta(ddy, borders, y0, eta1, h)[0]
    resolve2 = runge_kutta(ddy, borders, y0, eta2, h)[0]
    Phi1 = resolve1[-1] - diff_right(bcondition2, h, resolve1)
    Phi2 = resolve2[-1] - diff_right(bcondition2, h, resolve2)
    while abs(Phi2 - Phi1) > h/10:
        temp = eta2
        eta2 = eta2 - (eta2 - eta1) / (Phi2 - Phi1) * Phi2
        eta1 = temp
        resolve1 = runge_kutta(ddy, borders, y0, eta1, h)[0]
        resolve2 = runge_kutta(ddy, borders, y0, eta2, h)[0]
        Phi1 = resolve1[-1] - diff_right(bcondition2, h, resolve1)
        Phi2 = resolve2[-1] - diff_right(bcondition2, h, resolve2)

    return runge_kutta(ddy, borders, y0, eta2, h)[0]
```

Конечно-разностный метод:

```
def finite_difference_method(bcondition1, bcondition2, equation, borders, h):
    x = np.arange(borders[0], borders[1] + h, h)
    N = np.shape(x)[0]
    A = np.zeros((N, N))
    b = np.zeros(N)
    A[0][0] = bcondition1['a'] - bcondition1['b']/h
    A[0][1] = bcondition1['b']/h
    b[0] = bcondition1['c']
    for i in range(1, N-1):
        A[i][i-1] = 1/h**2 - equation['p'](x[i])/(2*h)
        A[i][i] = -2/h**2 + equation['q'](x[i])
        A[i][i+1] = 1/h**2 + equation['p'](x[i])/(2*h)
        b[i] = equation['f'](x[i])
    A[N-1][N-2] = -bcondition2['b']/h
    A[N-1][N-1] = bcondition2['a'] + bcondition2['b']/h
    b[N-1] = bcondition2['c']
    return solve(A, b)
```

Результат:

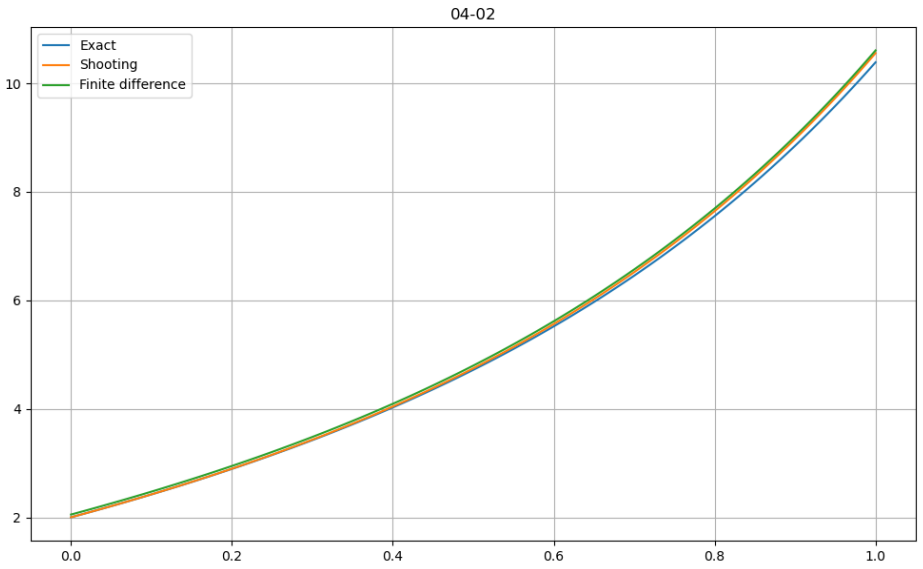

Рисунок 6

----------04-02----------

Runge Rombert errors:
Shooting method: 0.6867871751882191
Finite difference method: 0.7214282951019695

Exact solution errors:
Shooting method: 0.6738944028431726
Finite difference method: 1.0896595413301624
