Численные методы линейной алгебры
Лабораторная работа №1 - СЛАУ

Студент: Косов В.В.

Группа: М8О-311Б-23

Вариант: 7

Задание 1.

Формулировка задачи:

Методом Гаусса решить систему линейных алгебраических уравнений.
Вычислить определитель матрицы.
Найти обратную матрицу методом Гаусса.

Теоретическая часть:

Система линейных алгебраических уравнений (СЛАУ) общего вида может быть записана в матричной форме как $A \cdot x = b$, где A — матрица коэффициентов, x — вектор неизвестных, b — вектор правой части. Для нахождения решения системы, вычисления определителя и обратной матрицы может использоваться метод Гаусса.

Идея метода Гаусса заключается в последовательном исключении переменных: матрица коэффициентов преобразуется элементарными строковыми операциями так, чтобы в итоге получить эквивалентную систему с единичной матрицей слева и преобразованным вектором справа. Этот процесс включает выбор ведущего элемента (pivot), нормализацию строки и обнуление остальных элементов в текущем столбце.

Вычисление определителя производится на основе тех же преобразований: матрица A приводится к верхнетреугольному виду, после чего определитель равен произведению диагональных элементов с учётом перестановок строк.
Для нахождения обратной матрицы используется метод Гаусса-Жордана. К матрице A приписывается справа единичная матрица I, образуя расширенную матрицу $[A|I]$. После полного исключения переменных матрица принимает вид $[I|A^{-1}]$, где правая часть и есть обратная матрица.

Идеи реализации в коде:

Решение СЛАУ реализовано функцией gaussian_elimination(A, b):

1) поиск ведущего элемента в столбце;
2) перестановка строк при необходимости;
3) нормализация строки так, чтобы на диагонали стояла 1;
4) обнуление остальных элементов в столбце;
5) результат — вектор решений.

Вычисление определителя реализовано функцией determinant(A):

1) выбор ведущего элемента;
2) преобразование матрицы к верхнетреугольному виду;
3) произведение элементов главной диагонали с учётом числа перестановок строк.

Нахождение обратной матрицы реализовано функцией inverse_matrix(A):

1) формирование расширенной матрицы [A|I];
2) применение метода Гаусса-Жордана;
3) извлечение правой части как обратной матрицы.

Система уравнений:

3x1 + 6x2 - 4x3 + 3x4 + 2x5 = 15
4x1 + 2x2 + x3 + 3x4 + 5x5 = 58
-2x1 + 3x2 + 3x3 + 2x4 + 9x5 = 72
2x1 - 5x2 - 4x3 + 3x5 = 39
9x1 - 4x2 + 5x3 + x4 - 2x5 = 24

Код программы:

```
def gaussian_elimination(A, b):
    """
    Метод Гаусса для решения системы линейных уравнений Ax = b
    """
    n = len(A)

    # Преобразуем в расширенную матрицу
    for i in range(n):
        A[i] = A[i] + [b[i]]

    # Прямой ход
    for col in range(n):
        # Поиск ведущего элемента
        max_row = col
        for i in range(col + 1, n):
            if abs(A[i][col]) > abs(A[max_row][col]):
                max_row = i
        # Перестановка строк
        if max_row != col:
            A[col], A[max_row] = A[max_row], A[col]

        # Нормализация ведущей строки
        pivot = A[col][col]
        if abs(pivot) < 1e-12:
            raise ValueError("Система не имеет единственного решения")
        for j in range(col, n + 1):
            A[col][j] /= pivot

        # Обнуление остальных строк
        for i in range(n):
```

```
            if i != col:
                factor = A[i][col]
                for j in range(col, n + 1):
                    A[i][j] -= factor * A[col][j]

    # Решение
    x = [A[i][n] for i in range(n)]
    return x


def determinant(A):
    """
    Вычисление определителя методом Гаусса
    """
    n = len(A)
    M = [row[:] for row in A]  # копия матрицы
    det = 1
    swap_count = 0

    for col in range(n):
        # Поиск ведущего элемента
        max_row = col
        for i in range(col + 1, n):
            if abs(M[i][col]) > abs(M[max_row][col]):
                max_row = i
        if abs(M[max_row][col]) < 1e-12:
            return 0  # вырожденная матрица

        # Перестановка строк
        if max_row != col:
            M[col], M[max_row] = M[max_row], M[col]
            swap_count += 1

        # Диагональный элемент
        pivot = M[col][col]
        det *= pivot

        # Нормализация строки
        for j in range(col + 1, n):
            M[col][j] /= pivot

        # Обнуление под диагональю
        for i in range(col + 1, n):
            factor = M[i][col]
            for j in range(col + 1, n):
                M[i][j] -= factor * M[col][j]

    # Учитываем перестановки строк
    if swap_count % 2 == 1:
        det = -det

    return det
```

```
def inverse_matrix(A):
    """
    Метод Гаусса-Жордана для нахождения обратной матрицы
    """
    n = len(A)
    # Формируем расширенную матрицу [A | I]
    augmented = []
    for i in range(n):
        row = A[i][:] + [1 if j == i else 0 for j in range(n)]
        augmented.append(row)

    # Прямой и обратный ход
    for col in range(n):
        # Поиск ведущего элемента
        max_row = col
        for i in range(col + 1, n):
            if abs(augmented[i][col]) > abs(augmented[max_row][col]):
                max_row = i
        if max_row != col:
            augmented[col], augmented[max_row] = augmented[max_row],
augmented[col]

        # Нормализация ведущей строки
        pivot = augmented[col][col]
        if abs(pivot) < 1e-12:
            raise ValueError("Матрица вырожденная, обратной нет")
        for j in range(2 * n):
            augmented[col][j] /= pivot

        # Обнуление остальных строк
        for i in range(n):
            if i != col:
                factor = augmented[i][col]
                for j in range(2 * n):
                    augmented[i][j] -= factor * augmented[col][j]

    # Извлекаем обратную матрицу
    inv = []
    for i in range(n):
        inv.append(augmented[i][n:])
    return inv


if __name__ == "__main__":
    # Пример системы
    A = [
        [3, 6, -4, 3, 2],
        [4, 2, 1, 3, 5],
        [-2, 3, 3, 2, 9],
        [2, -5, -4, 0, 3],
```

```
    [9, -4, 5, 1, -2]
]
b = [15, 58, 72, 39, 24]

print("Решение системы:")
x = gaussian_elimination([row[:] for row in A], b[:])
print(x)

print("\nОпределитель:")
print(determinant([row[:] for row in A]))

print("\nОбратная матрица:")
inv = inverse_matrix([row[:] for row in A])
for row in inv:
    print(row)
```

Результат работы программы:

```
Решение системы:
[1.9999999999999947,
-3.0000000000000053,
 0.9999999999999998,
5.000000000000021,
7.999999999999995]

Определитель:
-1781.999999999997

Обратная матрица:
[0.5258136924803598, -1.0033670033670048, 0.4887766554433228,
0.15881032547699236, 0.455106621773289]

[0.49551066217732964, -0.9124579124579141, 0.4584736251402927,
0.03759820426487118, 0.3338945005611678]

[-0.10774410774410778, 0.10101010101010104, 0.0033670033670033517,
-0.09764309764309767, 0.013468013468013462]

[-1.5482603815937175, 3.4410774410774465, -1.6964085297418663,
-0.47081930415263834, -1.285634118967454]

[0.33164983164983225, -0.7171717171717183, 0.4427609427609434,
0.1599326599326601 2,0.2710437710437715]
```

Задание 2

Формулировка задачи:

Методом LU-разложения с выбором главного элемента (с частичным pivoting) найти решение системы линейных алгебраических уравнений. Вычислить матрицы L и U, а также решение системы Ax = b.

Теоретическая часть:

LU-разложение — это метод представления квадратной матрицы A в виде произведения двух матриц: A = LU, где L — нижняя треугольная матрица с единицами на диагонали, а U — верхняя треугольная матрица. Если матрица невырожденная и допустимо переставлять строки для устойчивости, можно использовать частичный выбор главного элемента.

Идея метода состоит в следующем:

1. На прямом ходе вычисляем элементы L и U так, чтобы выполнялось LU = PA, где P — перестановочная матрица, отражающая выбранные строки.
2. На обратном ходе решаем систему Ly = Pb (прямая подстановка), затем Ux = y (обратная подстановка).
3. Использование рациональных чисел (Fraction) позволяет избежать ошибок округления и получать точное решение.

Идеи реализации в коде:

1. Перевод всех коэффициентов матрицы и вектора b в объект Fraction для точной арифметики.
2. Реализация LU-разложения с частичным выбором главного элемента (pivoting), формирование матриц L и U и вектора перестановок P.
3. Прямая подстановка для решения Ly = Pb.
4. Обратная подстановка для решения Ux = y.
5. Вывод матриц L, U и решения системы x.

Система уравнений:

3x1 + 6x2 - 4x3 + 3x4 + 2x5 = 15
4x1 + 2x2 + x3 + 3x4 + 5x5 = 58
-2x1 + 3x2 + 3x3 + 2x4 + 9x5 = 72
2x1 - 5x2 - 4x3 + 3x5 = 39
9x1 - 4x2 + 5x3 + x4 - 2x5 = 24

Код программы:

```
from fractions import Fraction

def lu_decomposition_with_pivoting(A):
    n = len(A)
    A_frac = [[Fraction(x) for x in row] for row in A]
    L = [[Fraction(0) for _ in range(n)] for _ in range(n)]
    U = [[Fraction(0) for _ in range(n)] for _ in range(n)]
    P = list(range(n))

    for i in range(n):
        # выбор главного элемента
        max_row = i
```

```
        for k in range(i, n):
            if abs(A_frac[k][i]) > abs(A_frac[max_row][i]):
                max_row = k
        if max_row != i:
            A_frac[i], A_frac[max_row] = A_frac[max_row], A_frac[i]
            P[i], P[max_row] = P[max_row], P[i]

        # вычисление U
        for j in range(i, n):
            s = sum(L[i][k] * U[k][j] for k in range(i))
            U[i][j] = A_frac[i][j] - s

        # диагональ L = 1
        L[i][i] = Fraction(1)

        # вычисление L
        for j in range(i + 1, n):
            s = sum(L[j][k] * U[k][i] for k in range(i))
            L[j][i] = (A_frac[j][i] - s) / U[i][i]

    return P, L, U


def solve_lu(P, L, U, b):
    n = len(b)
    b_permuted = [b[P[i]] for i in range(n)]

    # прямая подстановка (Ly = Pb)
    y = [Fraction(0) for _ in range(n)]
    for i in range(n):
        s = sum(L[i][j] * y[j] for j in range(i))
        y[i] = b_permuted[i] - s

    # обратная подстановка (Ux = y)
    x = [Fraction(0) for _ in range(n)]
    for i in range(n - 1, -1, -1):
        s = sum(U[i][j] * x[j] for j in range(i + 1, n))
        x[i] = (y[i] - s) / U[i][i]

    return x


# Пример
mas = [
    [3, 6, -4, 3, 2],
    [4, 2, 1, 3, 5],
    [-2, 3, 3, 2, 9],
    [2, -5, -4, 0, 3],
    [9, -4, 5, 1, -2]
]

b = [15, 58, 72, 39, 24]
```

```
A_frac = [[Fraction(x) for x in row] for row in mas]
b_frac = [Fraction(x) for x in b]

P, L, U = lu_decomposition_with_pivoting(mas)

print("Матрица U:")
for row in U:
    print([round(float(x), 4) for x in row])

print("\nМатрица L:")
for row in L:
    print([round(float(x), 4) for x in row])

x = solve_lu(P, L, U, b_frac)
x_float = [float(x_i) for x_i in x]

print("\nРешение системы:")
print([round(val, 6) for val in x_float])
```

Результат работы программы:

```
Матрица U:
[9.0, -4.0, 5.0, 1.0, -2.0]
[0.0, 7.7778, -6.2222, 2.5556, 2.8889]
[0.0, 0.0, -1.2, -0.4714, 1.7714]
[0.0, 0.0, 0.0, 4.6786, 4.9048]
[0.0, 0.0, 0.0, 0.0, 11.6056]

Матрица L:
[1.0, 0.0, 0.0, 0.0, 0.0]
[0.4444, 1.0, 0.0, 0.0, 0.0]
[-0.2222, 0.2714, 1.0, 0.0, 0.0]
[0.2222, -0.5286, 1.1667, 1.0, 0.0]
[0.3333, 0.4286, -1.6667, -0.0458, 1.0]

Решение системы:
[9.342469, -11.682964, -14.505591, -11.145144, 11.570489]
```

Задание 3

Формулировка задачи:

Методом прогонки найти решение системы линейных алгебраических уравнений. Вычислить определитель матрицы.

Теоретическая часть:

Система уравнений имеет трёхдиагональную матрицу коэффициентов. Это означает, что в каждом уравнении участвуют только три неизвестные: текущая, соседняя слева и соседняя справа. Такая структура позволяет использовать специальный метод решения — метод прогонки (алгоритм Томаса).

Идея метода прогонки состоит в том, чтобы на прямом ходе преобразовать систему так, чтобы каждое неизвестное выражалось через следующее. Для этого вводятся прогоночные коэффициенты p и q, которые вычисляются по рекуррентным формулам. В результате получаем представление:
x[i] = p[i] * x[i+1] + q[i]

На обратном ходе, начиная с последнего уравнения, подставляем найденные значения и последовательно восстанавливаем все решения. Таким образом, система решается за линейное время O(n).

Определитель трёхдиагональной матрицы можно вычислить по рекуррентной формуле. Для подматриц определитель выражается через два предыдущих:
D1 = b0
D2 = b0*b1 – a1*c0
Dk = b[k-1]*D[k-1] – a[k-1]*c[k-2]*D[k-2]

Идеи реализации в коде:

1. Сбор системы уравнений для выделения коэффициентов трёхдиагональной матрицы a, b, c и вектора правой части d.
2. Прямой ход метода прогонки: вычисление массивов коэффициентов p и q.
3. Обратный ход: нахождение решений x, начиная с последнего элемента.
4. Проверка правильности решения подстановкой найденных значений в исходную систему.
5. Вычисление определителя по рекуррентной формуле, зависящей только от диагоналей матрицы.

Система уравнений:

7x1 - 3x2 = 26
3x1 + 5x2 - 2x3 = 28
2x2 + 9x3 - x4 = 15
-2x3 + 7x4 - 3x5 = 7
3x4 + 8x5 + x6 = -23
-5x5 + 9x6 + 4x7 = 24
3x6 - 6x7 - 2x8 = -3
3x7 + 8x8 = 24

Код программы:

```
def tridiagonal_matrix_algorithm(a, b, c, d):
    """
    Метод прогонки (алгоритм Томаса) для решения системы с
трехдиагональной матрицей
    a - поддиагональ
    b - главная диагональ
    c - наддиагональ
    d - правая часть системы
    """
    n = len(d)

    p = [0.0] * n  # прогоночные коэффициенты
    q = [0.0] * n  # прогоночные коэффициенты

    # Прямой ход
    if abs(b[0]) < 1e-15:
        raise ValueError("Нулевой элемент на диагонали b[0]")

    p[0] = -c[0] / b[0]
    q[0] = d[0] / b[0]

    for i in range(1, n):
        denom = b[i] + a[i] * p[i-1]
        if abs(denom) < 1e-15:
            raise ValueError(f"Нулевой знаменатель в строке {i+1}")

        if i < n-1:
            p[i] = -c[i] / denom
        else:
            p[i] = 0.0  # для последнего элемента

        q[i] = (d[i] - a[i] * q[i-1]) / denom

    print("\nОбратный ход (нахождение решения):")

    # Обратный ход
    x = [0.0] * n
    x[n-1] = q[n-1]
    print(f"x[{n-1}] = q[{n-1}] = {x[n-1]:.6f}")

    for i in range(n-2, -1, -1):
        x[i] = p[i] * x[i+1] + q[i]
        print(f"x[{i}] = p[{i}]*x[{i+1}] + q[{i}] = {p[i]:.6f}*{x[i+1]:.6f} + {q[i]:.6f} =
{x[i]:.6f}")

    print()
    return x
```

```
def determinant_tridiagonal(a, b, c):
    """
    Вычисление определителя трехдиагональной матрицы
    """
    n = len(b)
    if n == 1:
        return b[0]
    if n == 2:
        return b[0] * b[1] - a[1] * c[0]

    det_prev_prev = 1.0
    det_prev = b[0]

    for i in range(2, n+1):
        det_current = b[i-1] * det_prev - a[i-1] * c[i-2] * det_prev_prev
        det_prev_prev = det_prev
        det_prev = det_current

    return det_prev


def parse_tridiagonal_equations(equations):
    """
    Парсинг трехдиагональной системы уравнений
    Возвращает коэффициенты a, b, c, d для метода прогонки
    """
    n = len(equations)
    a = [0.0] * n
    b = [0.0] * n
    c = [0.0] * n
    d = [0.0] * n

    for i, eq in enumerate(equations):
        left, right = eq.split('=')
        d[i] = float(right.strip())

        import re
        terms = re.findall(r'([+-]?\s*\d*\.?\d*)\s*x(\d+)', left)

        for coeff_str, var_str in terms:
            var_num = int(var_str) - 1

            # Убираем все пробелы из коэффициента
            coeff_str = coeff_str.replace(' ', '')

            if coeff_str == '' or coeff_str == '+':
                coeff = 1.0
            elif coeff_str == '-':
                coeff = -1.0
            else:
                coeff = float(coeff_str)
```

```
            if var_num == i - 1:
                a[i] = coeff
            elif var_num == i:
                b[i] = coeff
            elif var_num == i + 1:
                c[i] = coeff

    return a, b, c, d




def check_solution(equations, x):
    """
    Проверка найденного решения подстановкой в исходные уравнения
    """
    print("\nПроверка решения:")
    a, b, c, d = parse_tridiagonal_equations(equations)

    for i, eq in enumerate(equations):
        # Вычисляем левую часть уравнения
        left_value = 0.0
        if i > 0:
            left_value += a[i] * x[i-1]  # коэффициент при x[i-1]
        left_value += b[i] * x[i]        # коэффициент при x[i]
        if i < len(x) - 1:
            left_value += c[i] * x[i+1]  # коэффициент при x[i+1]

        right_value = d[i]  # правая часть

        print(f"Уравнение {i+1}: {left_value:.6f} = {right_value:.6f}")


def solve_tridiagonal_from_equations(equations):
    """
    Решение трехдиагональной системы уравнений методом прогонки
    """
    print("=== Решение трехдиагональной системы методом прогонки
===")
    for eq in equations:
        print(eq)
    print()

    a, b, c, d = parse_tridiagonal_equations(equations)

    # Решение
    x = tridiagonal_matrix_algorithm(a, b, c, d)
    print("Решение системы:")
    for i, xi in enumerate(x, 1):
        print(f"x{i} = {xi:.6f}")

    # Определитель
```

```python
    det = determinant_tridiagonal(a, b, c)
    print(f"\nОпределитель матрицы: {det:.6f}")

    # Проверка решения
    check_solution(equations, x)


def test_tridiagonal_method():
    equations = [
        "7x1 - 3x2 = 26",
        "3x1 + 5x2 - 2x3 = 28",
        "2x2 + 9x3 - x4 = 15",
        "-2x3 + 7x4 - 3x5 = 7",
        "3x4 + 8x5 + x6 = -23",
        "-5x5 + 9x6 + 4x7 = 24",
        "3x6 - 6x7 - 2x8 = -3",
        "3x7 + 8x8 = 24"
    ]
    solve_tridiagonal_from_equations(equations)


if __name__ == "__main__":
    test_tridiagonal_method()
```

Результат работы программы:

```
=== Решение трехдиагональной системы методом прогонки ===
7x1 - 3x2 = 26
3x1 + 5x2 - 2x3 = 28
2x2 + 9x3 - x4 = 15
-2x3 + 7x4 - 3x5 = 7
3x4 + 8x5 + x6 = -23
-5x5 + 9x6 + 4x7 = 24
3x6 - 6x7 - 2x8 = -3
3x7 + 8x8 = 24

Обратный ход (нахождение решения):
x[7] = q[7] = 3.000000
x[6] = p[6]*x[7] + q[6] = -0.275544*3.000000 + 0.826633 = -0.000000
x[5] = p[5]*x[6] + q[5] = -0.419455*-0.000000 + 1.000000 = 1.000000
x[4] = p[4]*x[5] + q[4] = -0.107239*1.000000 + -2.892761 = -3.000000
x[3] = p[3]*x[4] + q[3] = 0.441667*-3.000000 + 1.325000 = -0.000000
x[2] = p[2]*x[3] + q[2] = 0.103774*-0.000000 + 1.000000 = 1.000000
x[1] = p[1]*x[2] + q[1] = 0.318182*1.000000 + 2.681818 = 3.000000
x[0] = p[0]*x[1] + q[0] = 0.428571*3.000000 + 3.714286 = 5.000000

Решение системы:
x1 = 5.000000
x2 = 3.000000
x3 = 1.000000
```

x4 = -0.000000
x5 = -3.000000
x6 = 1.000000
x7 = -0.000000
x8 = 3.000000

Определитель матрицы: -13334544.000000

Проверка решения:
Уравнение 1: 26.000000 = 26.000000
Уравнение 2: 28.000000 = 28.000000
Уравнение 3: 15.000000 = 15.000000
Уравнение 4: 7.000000 = 7.000000
Уравнение 5: -23.000000 = -23.000000
Уравнение 6: 24.000000 = 24.000000
Уравнение 7: -3.000000 = -3.000000
Уравнение 8: 24.000000 = 24.000000

Задание 4 - 5

Формулировка задачи

Методом **простых итераций (4)** и **методом Зейделя (5)** решить систему линейных алгебраических уравнений с погрешностью $\varepsilon=0,0001$. Сравнить количество итераций

Теоретическая часть (метод простых итераций):

Исходная система Ax = b преобразуется к виду x = Bx + c, где
B_{ij} = 0 при i = j, B_{ij} = -a_{ij}/a_{ii} при i ≠ j, и c_i = b_i / a_{ii}.
Итерационный процесс:
x^{(k+1)} = B x^{(k)} + c.
Критерий остановки: ||x^{(k+1)} - x^{(k)}||∞ < *ε.*
Достаточное условие сходимости — строгая диагональная доминантность: |*a*{ii}| > Σ_{j≠i} |a_{ij}| для всех i .

Теоретическая часть (метод Зейделя):

Метод Зейделя — модификация метода простых итераций. При вычислении x^{(k+1)} компоненты используются сразу же по мере их вычисления:
x_i^{(k+1)} = (1 / a_{ii}) (b_i - Σ_{j < i} a_{ij} x_j^{(k+1)} - Σ_{j > i} a_{ij} x_j^{(k)}).
Для j < i используются уже обновлённые значения x_j^{(k+1)}, для j > i — старые x_j^{(k)}. На практике метод Зейделя обычно даёт более быструю сходимость по сравнению с простыми итерациями при одинаковых условиях сходимости (например, при диа.

Идеи реализации в коде:

1. Подготовить матрицу A и вектор b, задать ε.
2. Проверить достаточное условие сходимости (диагональное доминирование) и вывести результат проверки.

3. Для простых итераций: построить B и c по формулам B_{ij} = -a_{ij}/a_{ii}, c_i = b_i/a_{ii}; выполнить итерации x_{new} = B x + c.
4. Для метода Зейделя: выполнить поэлементное обновление x_new[i] по формуле Зейделя, использовать в расчётах уже обновлённые компоненты.
5. На каждой итерации вычислять max_diff = max_i |x_new[i] - x[i]| и останавливаться при max_diff < ε или при достижении max_iterations.
6. Проверить полученный вектор x подстановкой в исходную систему (вывести левую и правую части уравнений для каждой строки).

Система уравнений:

-7x1 - 12x2 + 4x4 = -119
3x1 + 4x2 + 13x3 - 2x4 = 162
-4x1 + 3x2 - 8x3 + 17x4 = -125
13x1 + 3x2 + 5x3 + x4 = 125

Код программы:

```
def check_convergence_condition(A):
    """
    Проверка достаточного условия сходимости метода простых итераций
    Проверяем диагональное доминирование: |a_ii| > sum(|a_ij|) для j != i
    """
    n = len(A)
    is_diagonally_dominant = True
    weak_rows = []

    print("Проверка условия сходимости (диагональное доминирование):")
    for i in range(n):
        diagonal = abs(A[i][i])
        off_diagonal_sum = sum(abs(A[i][j]) for j in range(n) if j != i)

        print(f"Строка {i+1}: |a_{i+1}{i+1}| = {diagonal:.6f}, Σ|a_{i+1}j| =
{off_diagonal_sum:.6f}")

        if diagonal <= off_diagonal_sum:
            is_diagonally_dominant = False
            weak_rows.append(i+1)
            print(f"  Слабое доминирование в строке {i+1}")
        else:
            print(f"  Строгое доминирование в строке {i+1}")

    if is_diagonally_dominant:
        print(" Матрица имеет строгое диагональное доминирование -
сходимость гарантирована")
    else:
        print(f"   Матрица не имеет строгого диагонального доминирования")
        print(f"   Слабые строки: {weak_rows}")
        print("   Сходимость не гарантирована, но метод может сойтись")

    print()
```

```
    return is_diagonally_dominant


def simple_iteration(A, b, epsilon=1e-4, max_iterations=10000):
    """
    Метод простых итераций для решения системы Ax = b
    Преобразуем систему к виду x = Bx + c
    """
    n = len(A)
    x = [0.0] * n  # начальное приближение

    # Проверка условия сходимости
    check_convergence_condition(A)

    # Формируем матрицу B и вектор c для итерационного процесса
    B = [[0.0] * n for _ in range(n)]
    c = [0.0] * n

    for i in range(n):
        if abs(A[i][i]) < 1e-15:
            raise ValueError("Нулевой элемент на диагонали - метод не
применим")

        c[i] = b[i] / A[i][i]
        for j in range(n):
            if i != j:
                B[i][j] = -A[i][j] / A[i][i]

    # Итерационный процесс
    for iteration in range(max_iterations):
        x_new = [sum(B[i][j] * x[j] for j in range(n)) + c[i] for i in range(n)]

        # Проверка сходимости
        max_diff = max(abs(x_new[i] - x[i]) for i in range(n))
        if max_diff < epsilon:
            print(f"Сходимость достигнута на итерации {iteration + 1}")
            print(f"Максимальная разность: {max_diff:.2e} < ε = {epsilon:.2e}")
            return x_new, iteration + 1

        x = x_new

    raise ValueError(f"Метод не сошелся за {max_iterations} итераций.
Последняя разность: {max_diff:.2e}")

def seidel_method(A, b, epsilon=1e-4, max_iterations=10000):
    """
    Метод Зейделя для решения системы Ax = b
    Использует уже вычисленные значения на текущей итерации
    """
    n = len(A)
    x = [0.0] * n  # начальное приближение
```

```
    # Проверка условия сходимости
    check_convergence_condition(A)

    for iteration in range(max_iterations):
        x_new = x[:]  # копируем текущее приближение

        for i in range(n):
            # Сумма с уже обновленными значениями (метод Зейделя)
            s1 = sum(A[i][j] * x_new[j] for j in range(i))
            # Сумма со старыми значениями
            s2 = sum(A[i][j] * x[j] for j in range(i+1, n))

            if abs(A[i][i]) < 1e-15:
                raise ValueError("Нулевой элемент на диагонали - метод не
применим")

            x_new[i] = (b[i] - s1 - s2) / A[i][i]

        # Проверка сходимости
        max_diff = max(abs(x_new[i] - x[i]) for i in range(n))
        if max_diff < epsilon:
            print(f"Сходимость достигнута на итерации {iteration + 1}")
            print(f"Максимальная разность: {max_diff:.2e} < ε = {epsilon:.2e}")
            return x_new, iteration + 1

        x = x_new

    raise ValueError(f"Метод не сошелся за {max_iterations} итераций.
Последняя разность: {max_diff:.2e}")



def print_matrix(M):
    """Вывод матрицы в удобном формате"""
    for row in M:
        print("  ".join(f"{v:12.6f}" for v in row))


def test_simple_iteration():
    """
    Тестирование метода простых итераций на данных из задания
    """
    print("=== Задание 4: Метод простых итераций ===")
    print("Система уравнений:")
    print("-7x1 + -12x2 + 4x4 = -119")
    print("3x1 + 4x2 + 13x3 - 2x4 = 162")
    print("-4x1 + 3x2 - 8x3 + 17x4 = -125")
    print("13x1 + 3x2 + 5x3 + x4 = 125")
    print()
```

```
    # Переупорядочиваем систему для улучшения диагонального 
доминирования
    A = [
    [13,  3,  5,  1],   # наибольший диагональный элемент для столбца 1
    [-7, -12, 0,  4],   # наибольший по столбцу 2
    [3,   4, 13, -2],   # наибольший по столбцу 3
    [-4,  3, -8, 17]    # наибольший по столбцу 4
    ]

    b = [125, -119, 162, -125]

    try:
        # Решение системы методом простых итераций
        x, iterations = simple_iteration(A, b, epsilon=1e-4)

        print("Решение системы:")
        for i, xi in enumerate(x, 1):
            print(f"x{i} = {xi:.6f}")
        print()

        print(f"Количество итераций: {iterations}")
        print(f"Точность: ε = 0.0001")

        # Проверка решения
        check_solution(A, b, x)

    except ValueError as e:
        print(f"Ошибка: {e}")


def test_seidel_method():
    """
    Тестирование метода Зейделя на данных из задания
    """
    print("=== Задание 5: Метод Зейделя ===")

    # Переупорядоченная система для улучшения диагонального 
доминирования
    A = [
    [13,  3,  5,  1],   # наибольший диагональный элемент для столбца 1
    [-7, -12, 0,  4],   # наибольший по столбцу 2
    [3,   4, 13, -2],   # наибольший по столбцу 3
    [-4,  3, -8, 17]    # наибольший по столбцу 4
    ]

    b = [125, -119, 162, -125]

    try:
        # Решение системы методом Зейделя
        x, iterations = seidel_method(A, b, epsilon=1e-4)

        print("Решение системы:")
```

```
        for i, xi in enumerate(x, 1):
            print(f"x{i} = {xi:.6f}")
        print()

        print(f"Количество итераций: {iterations}")
        print(f"Точность: ε = 0.0001")
        print()

        check_solution_zed(A, b, x)
    except ValueError as e:
        print(f"Ошибка: {e}")

def check_solution(A, b, x):
    """
    Проверка найденного решения подстановкой в исходные уравнения
    """
    print("\nПроверка решения:")
    n = len(A)

    for i in range(n):
        left_value = sum(A[i][j] * x[j] for j in range(n))
        right_value = b[i]
        print(f"Уравнение {i+1}: {left_value:.6f} = {right_value:.6f}")
    print()


def check_solution_zed(A, b, x):
    """
    Проверка найденного решения подстановкой в исходные уравнения
    """
    print("\nПроверка решения:")
    n = len(A)

    for i in range(n):
        left_value = sum(A[i][j] * x[j] for j in range(n))
        right_value = b[i]
        print(f"Уравнение {i+1}: {left_value:.6f} = {right_value:.6f}")

if __name__ == "__main__":
    test_simple_iteration()
    test_seidel_method()
```

Результат программы:

```
=== Задание 4: Метод простых итераций ===
Система уравнений:
-7x1 + -12x2 + 4x4 = -119
3x1 + 4x2 + 13x3 - 2x4 = 162
-4x1 + 3x2 - 8x3 + 17x4 = -125
13x1 + 3x2 + 5x3 + x4 = 125

Проверка условия сходимости (диагональное доминирование):
```

Строка 1: |a_11| = 13.000000, Σ|a_1j| = 9.000000
Строгое доминирование в строке 1
Строка 2: |a_22| = 12.000000, Σ|a_2j| = 11.000000
Строгое доминирование в строке 2
Строка 3: |a_33| = 13.000000, Σ|a_3j| = 9.000000
Строгое доминирование в строке 3
Строка 4: |a_44| = 17.000000, Σ|a_4j| = 15.000000
Строгое доминирование в строке 4
Матрица имеет строгое диагональное доминирование - сходимость гарантирована

Сходимость достигнута на итерации 30
Максимальная разность: 9.27e-05 < ε = 1.00e-04

Решение системы:
x1 = 4.999971
x2 = 5.999963
x3 = 8.999967
x4 = -2.999977

Количество итераций: 30
Точность: ε = 0.0001

Проверка решения:
Уравнение 1: 124.999373 = 125.000000
Уравнение 2: -118.999259 = -119.000000
Уравнение 3: 161.999295 = 162.000000
Уравнение 4: -124.999337 = -125.000000

=== Задание 5: Метод Зейделя ===
Проверка условия сходимости (диагональное доминирование):
Строка 1: |a_11| = 13.000000, Σ|a_1j| = 9.000000
Строгое доминирование в строке 1
Строка 2: |a_22| = 12.000000, Σ|a_2j| = 11.000000
Строгое доминирование в строке 2
Строка 3: |a_33| = 13.000000, Σ|a_3j| = 9.000000
Строгое доминирование в строке 3
Строка 4: |a_44| = 17.000000, Σ|a_4j| = 15.000000
Строгое доминирование в строке 4
Матрица имеет строгое диагональное доминирование - сходимость гарантирована

Сходимость достигнута на итерации 8
Максимальная разность: 3.77e-05 < ε = 1.00e-04
Решение системы:
x1 = 4.999991
x2 = 6.000007
x3 = 9.000001
x4 = -3.000003

Количество итераций: 8

Точность: ε = 0.0001

Проверка решения:
Уравнение 1: 124.999907 = 125.000000
Уравнение 2: -119.000030 = -119.000000
Уравнение 3: 162.000015 = 162.000000
Уравнение 4: -125.000000 = -125.000000

Сравнение методов:
Метод Зейделя сходится быстрее чем метод простых итераций
Метод Зейделя - 8 итераций
Метод простых итераций - 30 итераций

Задание 6.

Формулировка задачи:
Определить собственные значения матрицы с заданной точностью. Реализовать алгоритм QR-разложения матриц и приведение к форме Шура.

Теоретическая часть:
Собственные значения матрицы A — это числа λ, для которых существует ненулевой вектор v, удовлетворяющий уравнению Av = λv. Для вычисления собственных значений используется QR-алгоритм, который состоит в последовательных разложениях матрицы на произведение A_k = Q_kR_k и переходе к новой матрице A_{k+1} = R_kQ_k.

QR-алгоритм с помощью преобразований Хаусхолдера позволяет привести матрицу к верхнетреугольной форме (форме Шура). Верхнетреугольная матрица сохраняет собственные значения исходной матрицы на диагонали и в блоках 2x2, если встречаются комплексные сопряжённые пары. Проверка сходимости осуществляется по норме поддиагональных элементов матрицы.

Формулы:
QR-разложение:
A_k = Q_k * R_k
Итерация QR-алгоритма:
A_{k+1} = R_k * Q_k

Определение собственных значений из формы Шура:
Если поддиагональный элемент блока приблизительно равен нулю, λ = A_{ii}.
Если блок 2x2:
[a b; c d] → λ_{1,2} = (a + d ± sqrt((a + d)^2 - 4*(ad - bc)))/2
При отрицательном дискриминанте значения комплексные:
λ = (a + d)/2 ± isqrt(-(a + d)^2 + 4(ad - bc))/2

Идеи реализации в коде:

1. Реализованы функции для базовых операций с матрицами и векторами: умножение, вычитание, умножение на скаляр, норма вектора, единичная матрица, внешнее произведение.
2. QR-разложение реализовано с помощью отражений Хаусхолдера. Создается отражение H = I - 2*v*v^T / (v^T*v), которое применяется к подматрице для обнуления поддиагональных элементов.
3. QR-алгоритм выполняет итерации A_{k+1} = R_k*Q_k до тех пор, пока норма поддиагональных элементов не станет меньше заданного eps, или пока не будет достигнут максимум итераций.
4. После достижения верхнетреугольной формы извлекаются собственные значения: диагональные элементы — вещественные собственные значения, блоки 2x2 — решение квадратного уравнения для комплексных значений.
5. Собственные значения возвращаются в виде списка вещественных и комплексных чисел.

Матрица:

[3 -5 -4 7 -1
-1 17 1 2 2
-2 3 4 -1 5
2 -1 -4 1 3
1 3 -5 1 - 2]

Код программы:

```
def matrix_mult(A, B):
    """Умножение матриц"""
    rows_A, cols_A = len(A), len(A[0])
    rows_B, cols_B = len(B), len(B[0])
    if cols_A != rows_B:
        raise ValueError("Несовместимые размеры матриц для умножения")
    result = [[0 for _ in range(cols_B)] for _ in range(rows_A)]
    for i in range(rows_A):
        for j in range(cols_B):
            for k in range(cols_A):
                result[i][j] += A[i][k] * B[k][j]
    return result

def vector_dot(u, v):
    """Скалярное произведение векторов"""
    if len(u) != len(v):
        raise ValueError("Векторы должны иметь одинаковую длину")
    return sum(u[i] * v[i] for i in range(len(u)))

def vector_norm(v):
    """Евклидова норма"""
    return sum(x*x for x in v) ** 0.5

def identity_matrix(n):
    """Единичная матрица n x n"""
```

```
    I = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        I[i][i] = 1
    return I

def outer_product(u, v):
    """Внешнее произведение векторов"""
    rows, cols = len(u), len(v)
    result = [[0 for _ in range(cols)] for _ in range(rows)]
    for i in range(rows):
        for j in range(cols):
            result[i][j] = u[i] * v[j]
    return result

def scalar_mult(c, A):
    """Умножение матрицы на скаляр"""
    rows, cols = len(A), len(A[0])
    result = [[0 for _ in range(cols)] for _ in range(rows)]
    for i in range(rows):
        for j in range(cols):
            result[i][j] = c * A[i][j]
    return result

def matrix_sub(A, B):
    """Вычитание матриц"""
    rows, cols = len(A), len(A[0])
    if rows != len(B) or cols != len(B[0]):
        raise ValueError("Матрицы должны иметь одинаковый размер")
    result = [[0 for _ in range(cols)] for _ in range(rows)]
    for i in range(rows):
        for j in range(cols):
            result[i][j] = A[i][j] - B[i][j]
    return result

def householder_qr(A):
    """QR-разложение с помощью отражений Хаусхолдера"""
    n = len(A)
    Q = identity_matrix(n)
    R = [row[:] for row in A]

    for k in range(n - 1):
        x = [R[i][k] for i in range(k, n)]
        norm_x = vector_norm(x)
        if norm_x < 1e-15:
            continue
        sign = 1 if x[0] >= 0 else -1
        v = x[:]
        v[0] += sign * norm_x
        beta = 2.0 / vector_dot(v, v)
        v_vt = outer_product(v, v)
        H = matrix_sub(identity_matrix(len(v)), scalar_mult(beta, v_vt))
```

```
        # Применяем H к подматрице R
        R_sub = [[R[i][j] for j in range(k, n)] for i in range(k, n)]
        H_R = matrix_mult(H, R_sub)
        for i in range(k, n):
            for j in range(k, n):
                R[i][j] = H_R[i - k][j - k]

        # Обновляем Q
        Q_sub = [[Q[i][j] for j in range(k, n)] for i in range(n)]
        Q_H = matrix_mult(Q_sub, H)
        for i in range(n):
            for j in range(k, n):
                Q[i][j] = Q_H[i][j - k]

    return Q, R

def qr_algorithm(A, eps=1e-10, max_iter=1000):
    """QR-алгоритм для нахождения формы Шура"""
    n = len(A)
    A_k = [row[:] for row in A]

    for iteration in range(max_iter):
        Q, R = householder_qr(A_k)
        A_next = matrix_mult(R, Q)
        A_k = A_next
        # проверка сходимости по поддиагонали
        converged = True
        for m in range(n - 1):
            sub_norm = vector_norm([A_k[i][m] for i in range(m + 1, n)])
            if sub_norm >= eps:
                converged = False
                break
        if converged:
            break
    return A_k

def extract_eigenvalues_from_schur(schur_form, eps=1e-10):
    """Извлечение собственных значений из верхнетреугольной матрицы
Шура"""
    n = len(schur_form)
    eigenvalues = []
    i = 0
    while i < n:
        if i == n - 1 or abs(schur_form[i+1][i]) < eps:
            eigenvalues.append(schur_form[i][i])
            i += 1
        else:
            a = schur_form[i][i]
            b = schur_form[i][i+1]
            c = schur_form[i+1][i]
            d = schur_form[i+1][i+1]
            trace = a + d
```

```
            det = a*d - b*c
            discriminant = trace**2 - 4*det
            if discriminant >= 0:
                lambda1 = (trace + discriminant**0.5)/2
                lambda2 = (trace - discriminant**0.5)/2
                eigenvalues.extend([lambda1, lambda2])
            else:
                real_part = trace / 2
                imag_part = (-discriminant)**0.5 / 2
                eigenvalues.extend([complex(real_part, imag_part),
                                    complex(real_part, -imag_part)])
            i += 2
    return eigenvalues

# Пример использования
A = [
    [3, -5, -4, 7, -1],
    [-1, 17, 1, 2, 2],
    [-2, 3, 4, -1, 5],
    [2, -1, -4, 1, 3],
    [1, 3, -5, 1, 2]
]

print("Исходная матрица:")
for row in A:
    print(row)

schur_form = qr_algorithm(A, eps=1e-4, max_iter=10000)

print("\nФорма Шура:")
for row in schur_form:
    print([f"{x:.6f}" for x in row])

our_eigenvalues = extract_eigenvalues_from_schur(schur_form)

print("\nСобственные значения:")
for val in our_eigenvalues:
    if isinstance(val, complex):
        print(f"{val.real:.6f} + {val.imag:.6f}i")
    else:
        print(f"{val:.6f}")
```

Результат работы программы:

```
Исходная матрица:
[3, -5, -4, 7, -1]
[-1, 17, 1, 2, 2]
[-2, 3, 4, -1, 5]
[2, -1, -4, 1, 3]
```

[1, 3, -5, 1, 2]

Форма Шура:
['17.547415', '-5.913174', '-0.809283', '0.938790', '1.466201']
['0.000000', '8.004202', '5.384931', '1.963709', '4.260833']
['0.000000', '-0.000000', '2.496931', '-3.898920', '3.598356']
['-0.000000', '-0.000000', '4.134745', '0.588976', '-2.975247']
['0.000000', '0.000000', '-0.000000', '0.000000', '-1.637524']

Собственные значения:
17.547415
8.004202
1.542953 + 3.900124i
1.542953 + -3.900124i
-1.637524