ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Дисциплина:

«Вычислительная математика»

Лабораторная работа №2

«Решение СЛАУ»

Выполнил:

Студент группы P3231

Колмаков Дмитрий Владимирович

Преподаватель:

Перл О.В.

Санкт-Петербург

2024г.

Оглавление

Задание

Решите систему линейных алгебраических уравнений, реализуя **метод простых итераций**.

Формат входных данных:
n
a11 a12 ... a1n b1
a21 a22 ... a2n b2
...
an1 an2 ... ann bn

Формат вывода:
x1
x2
...
xn, где x1..xn - значения неизвестных.

Если для текущей матрицы нет диагонального преобладания, вам следует попытаться найти его путем перестановки столбцов или / и строк. Если после такой операции преобладание диагонали по-прежнему отсутствует, должно быть напечатано следующее сообщение:

"The system has no diagonal dominance for this method. Method of the simple iterations is not applicable.". Для этого задайте значение переменной isMethodApplicable и сообщение об ошибке.

Теория

СЛАУ - это система из n уравнений с m неизвестными $x_1, x_2, \dots, x_m$, принадлежащими заданному числовому множеству M. Решить СЛАУ значит найти упорядоченную совокупность чисел $\alpha_1, \alpha_2, \dots, \alpha_m$, каждое из которых принадлежит множеству M, в котором рассматривается система, при подстановке которых вместо соответствующих неизвестных каждое уравнение системы обращается в тождество.

Метод простых итераций относится к итерационным методам решения СЛАУ, суть которых состоит в последовательном приближении к решению. Решение начинается с начального приближения. В результате каждой итерации находят новое приближение. Итерации проводятся до получения решения с требуемой точностью.

Рассмотрим ход метода. Пусть дана СЛАУ с невырожденной матрицей:

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

Выразим неизвестные $x_1, x_2, \dots, x_n$ соответственно из первого, второго и т.д. уравнений системы:

$$\begin{cases} x_1 = \frac{a_{12}}{a_{11}}x_2 + \frac{a_{13}}{a_{11}}x_3 + \dots + \frac{a_{1n}}{a_{11}}x_n - \frac{b_1}{a_{11}} \\ x_2 = \frac{a_{21}}{a_{22}}x_1 + \frac{a_{23}}{a_{22}}x_3 + \dots + \frac{a_{2n}}{a_{22}}x_n - \frac{b_2}{a_{22}} \\ \dots \\ x_n = \frac{a_{n1}}{a_{nn}}x_1 + \frac{a_{n2}}{a_{nn}}x_2 + \dots + \frac{a_{nn-1}}{a_{nn}}x_{n-1} - \frac{b_n}{a_{nn}} \end{cases}$$

Представим систему в сокращенном виде: $x_i = \frac{1}{a_{ii}}\left(\sum_{k=1}^{n} a_{ik}x_k - b_i\right)$.

В качестве начального приближения $x_i^{[0]}$ могут быть использованы:

- 0;
- b_i;
- $\frac{b_i}{a_{ii}}$;
- некоторое предварительно рассчитанное значение (например, из прямых методов);
- любое другое значение.

Возьмем начальное приближение $x_i^{[0]} = 0$. Приближение под номером $j + 1$ будем считать как $x_i^{[j+1]} = x_i^{[j]} - \frac{1}{a_{ii}}\left(\sum_{k=1}^{n} a_{ik}x_k^{[j]} - b_i\right)$. Это и есть рабочая формула метода простых итераций.

Данный метод сходится при выполнении условия доминирования диагонали:

$$|a_{ii}| \geq \sum_{i \neq j} |a_{ij}| \, (i, j = 1, 2, ..., n)$$

При этом хотя бы для одного уравнения неравенство должно выполняться строго. Рассмотрим, как можно добиться выполнения этого условия. Пусть дана система $A \cdot x = b$, при этом квадратная матрица A размера n не удовлетворяет условию доминирования диагонали. Умножим обе части уравнения на матрицу B слева:

$$B \cdot A \cdot x = B \cdot b$$

И потребуем, чтобы $B \cdot A = D$, где D - любая квадратная матрица размера n с диагональным преобладанием. В качестве матрицы D, например, можно использовать матрицу вида:

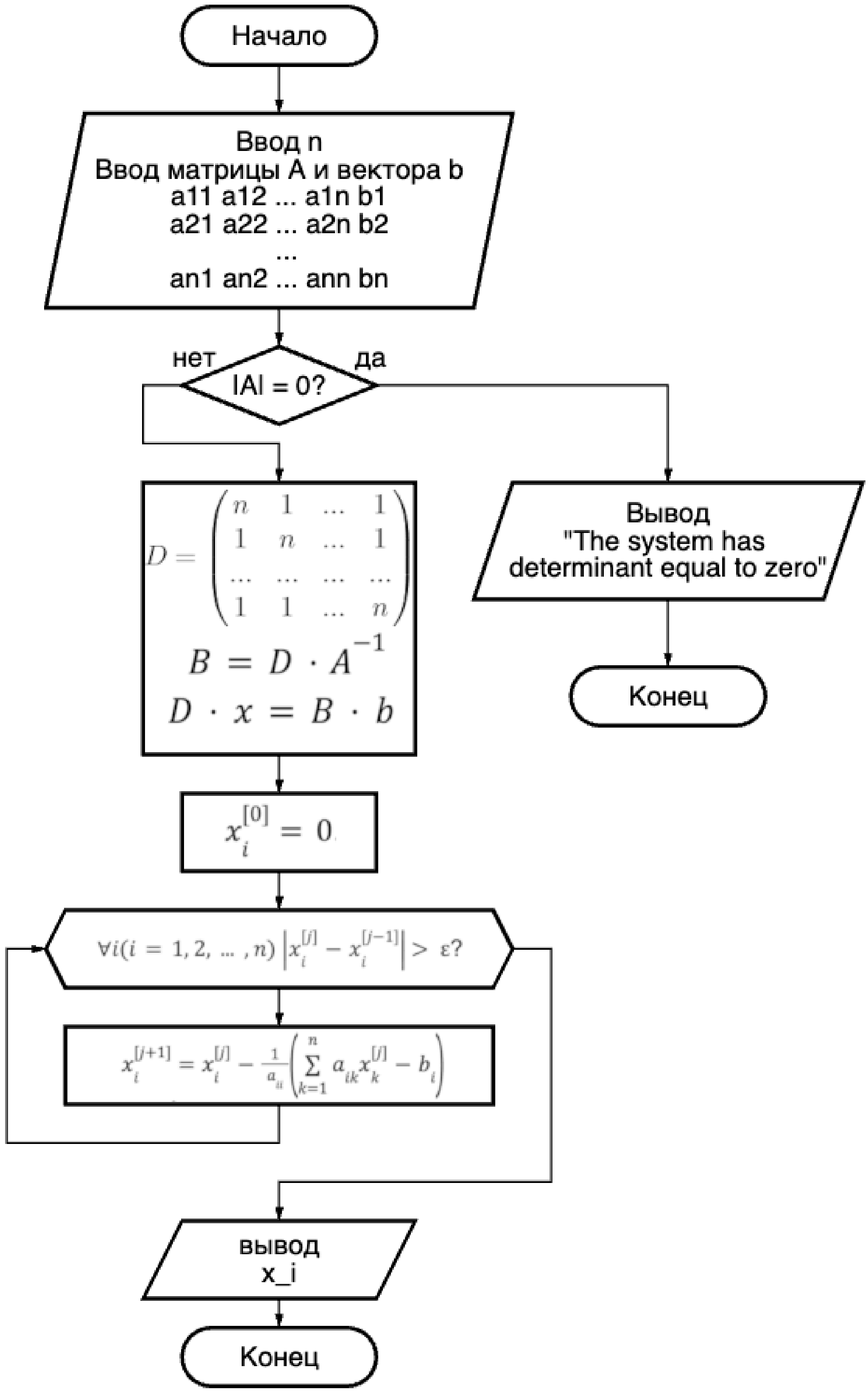
$$\begin{pmatrix} n & 1 & ... & 1 \\ 1 & n & ... & 1 \\ ... & ... & ... & ... \\ 1 & 1 & ... & n \end{pmatrix}$$

Тогда $B = D \cdot A^{-1}$. Система примет вид $D \cdot x = B \cdot b$. Такая система удовлетворяет условию доминирования диагонали и может быть решена методом простых итераций.

Итерационный процесс останавливается, когда для всех x_i изменение по сравнению с предыдущей итерацией составило не больше ε:

$$\forall i (i = 1, 2, \dots , n) \left| x_i^{[j]} - x_i^{[j-1]} \right| > \varepsilon ?$$

Блок-схема



Начало
Ввод n
Ввод матрицы A и вектора b
a11 a12 ... a1n b1
a21 a22 ... a2n b2
...
an1 an2 ... ann bn
нет
да
|A| = 0?
$D = \begin{pmatrix} n & 1 & \dots & 1 \\ 1 & n & \dots & 1 \\ \dots & \dots & \dots & \dots \\ 1 & 1 & \dots & n \end{pmatrix}$
$B = D \cdot A^{-1}$
$D \cdot x = B \cdot b$
Вывод
"The system has
determinant equal to zero"
Конец
$x_i^{[0]} = 0$
$\forall i (i = 1, 2, \dots, n) \left| x_i^{[j]} - x_i^{[j-1]} \right| > \varepsilon ?$
$x_i^{[j+1]} = x_i^{[j]} - \frac{1}{a_{ii}} \left(\sum_{k=1}^{n} a_{ik} x_k^{[j]} - b_i \right)$
вывод
x_i
Конец

Реализация программы

```
import math


class Result:
    isMethodApplicable = True
    errorMessage = ""

    @staticmethod
    def get_triangle_form(n, matrix, epsilon):
        triangle_matrix = [row[:] for row in matrix]
        count_permutations = 0
        for i in range(n - 1):
            for j in range(n - 1, i, -1):
                if triangle_matrix[j][i] == 0:
                    continue
                else:
                    try:
                        ratio = triangle_matrix[j][i] / triangle_matrix[j - 1][i]
                    except ZeroDivisionError:
                        triangle_matrix[j], triangle_matrix[j - 1] = (
                            triangle_matrix[j - 1], triangle_matrix[j])
                        count_permutations += 1
                        continue
                    for k in range(len(triangle_matrix[j])):
                        triangle_matrix[j][k] = (triangle_matrix[j][k] -
                                                 ratio * triangle_matrix[j - 1][k])
                        if abs(triangle_matrix[j][k]) < epsilon ** 2:
                            triangle_matrix[j][k] = 0
        return triangle_matrix, count_permutations

    @staticmethod
    def count_determinant(n, matrix, epsilon):
        determinant = 1
        triangle_matrix, count_permutations = Result.get_triangle_form(n, matrix, epsilon)
        for i in range(n):
            determinant *= triangle_matrix[i][i]
        return determinant * (-1) ** (count_permutations != 0)

    @staticmethod
    def get_transposed_matrix(matrix):
        return list(map(lambda x: list(x), zip(*matrix)))

    @staticmethod
    def get_matrix_minor(matrix, i, j):
        return [row[:j] + row[j + 1:] for row in (matrix[:i] + matrix[i + 1:])]

    @staticmethod
    def get_inverse_matrix(n, matrix, epsilon):
        determinant = Result.count_determinant(n, matrix, epsilon)
        # special case for 2x2 matrix:
        if len(matrix) == 2:
            return [[matrix[1][1] / determinant, -matrix[0][1] / determinant],
                    [-matrix[1][0] / determinant, matrix[0][0] / determinant]]

        # find matrix of cofactors
        cofactors = []
        for r in range(len(matrix)):
            cofactor_row = []
            for c in range(len(matrix)):
                minor = Result.get_matrix_minor(matrix, r, c)
                cofactor_row.append(((-1) ** (r + c)) *
                                    Result.count_determinant(len(minor), minor, epsilon))
            cofactors.append(cofactor_row)
```

```
        cofactors = Result.get_transposed_matrix(cofactors)
        for r in range(len(cofactors)):
            for c in range(len(cofactors)):
                cofactors[r][c] = cofactors[r][c] / determinant
        return cofactors

    @staticmethod
    def multiply_matrix_by_matrix(matrix_a, matrix_b):
        if len(matrix_a[0]) != len(matrix_b):
            return None
        transposed_b = Result.get_transposed_matrix(matrix_b)
        return [[sum(el_a * el_b for el_a, el_b in zip(row_a, col_b))
                 for col_b in transposed_b] for row_a in matrix_a]

    @staticmethod
    def multiply_matrix_by_vector(matrix, vector):
        if len(matrix[0]) != len(vector):
            return None
        result_vector = []
        for i in range(len(matrix[0])):
            summ = 0
            for j in range(len(vector)):
                summ += vector[j] * matrix[i][j]
            result_vector.append(summ)
        return result_vector

    @staticmethod
    def solveBySimpleIterations(n, matrix, epsilon):
        # Check if incorrect matrix
        if n != len(matrix) or any(map(lambda row: len(row) != n + 1, matrix)):
            Result.errorMessage = "The system is invalid"
            Result.isMethodApplicable = False
            return None

        # Check if determinant == 0
        if Result.count_determinant(n, matrix, epsilon) == 0:
            Result.errorMessage = "The system has determinant equal to zero"
            Result.isMethodApplicable = False
            return None

        A = list(map(lambda row: row[:-1], matrix))
        b = list(map(lambda row: row[-1], matrix))
        D = [[n if i == j else 1 for j in range(n)] for i in range(n)]
        A_inversed = Result.get_inverse_matrix(n, A, epsilon)
        B = Result.multiply_matrix_by_matrix(D, A_inversed)
        d = Result.multiply_matrix_by_vector(B, b)

        # Diagonal dominant matrix
        new_matrix = list(zip(*D, d))

        # Check if matrix is not diagonal dominant
        if any(abs(new_matrix[i][i]) < sum(abs(new_matrix[i][j]) if j != i else 0
                                               for j in range(n)) for i in
range(len(new_matrix))):
            Result.errorMessage = ("The system has no diagonal dominance for this method. "
                                   "Method of the simple iterations is not applicable.")
            Result.isMethodApplicable = False
            return None

        prev_x_vector = [math.inf] * len(new_matrix)
        x_vector = [0] * len(new_matrix)

        while any(abs(x_vector[i] - prev_x_vector[i]) > epsilon for i in range(n)):
            prev_x_vector = x_vector
            x_vector = [
                prev_x_vector[i] - (sum([new_matrix[i][j] * prev_x_vector[j]
                for j in range(n)]) - new_matrix[i][-1]) /
```

```
            new_matrix[i][i] for i in range(n)]

    return x_vector
```

Тесты

Тест 1

Ошибочный, неверный формат данных

Входные данные:
3
1 2 3 4
5 6 7
8 9 10 11
0.001

Выходные данные:
The system is invalid

Тест 2

Ошибочный, определитель = 0

Входные данные:
3
1 2 3 4
1 2 3 4
5 6 7 8
0.0001

Выходные данные:
The system has determinant equal to zero

Тест 3

Корректный, матрица с диагональным преобладанием

Входные данные:
3
5 -1 3 5
1 -4 2 20
2 -1 5 10
0.001

Выходные данные:
-0.7349108361764418
-4.48491083510095
1.3974421043884764

Тест 4

Корректный, матрица без диагонального преобладания

Входные данные:
3
0 4 4 1
1 3 2 1
3 2 1 1
0.001

Выходные данные:
0.12528545728256504
0.37528543985963536
-0.12471452529450504

Тест 5

Корректный, та же матрица, более высокая точность

Входные данные:
3
0 4 4 1
1 3 2 1
3 2 1 1
0.00000000001

Выходные данные:
0.12500000000003927
0.3750000000000393
-0.12499999999996073

Тест 6

Корректный, большая матрица, высокая точность

Входные данные:
20
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0.000000000001

Выходные данные:
1.0000000000004798
1.0000000000004798
1.0000000000004798
1.0000000000004798
1.00000000000048
1.00000000000048
1.0000000000004803
1.00000000000048
1.0000000000004803
1.0000000000004803
1.0000000000004805
1.0000000000004805
1.00000000000048
1.0000000000004805
1.0000000000004805
1.0000000000004807
1.0000000000004807
1.0000000000004807
1.0000000000004807
1.000000000000481

Вывод

В ходе проделанной работы я повторил общие сведения о СЛАУ и их решении. Изучил различные методы решения СЛАУ - прямые и итерационные. Узнал, что такое невязка и как ее находить. Подробно изучил метод простых итераций. Работая над ним, выяснил, что:

- метод простых итераций позволяет добиться заданной точности, в отличие от прямых методов, имеющих набегающую погрешность;
- данный метод эффективен по памяти, так как нет необходимости хранить всю матрицу, в отличие от методов Гаусса и Холецкого;
- вычисление каждого уравнения на текущей итерации независимо друг от друга, и поэтому может быть рассчитано параллельно;
- данный метод работает не всегда, для его работы необходимо условие диагонального преобладания, которое на практике встречается достаточно редко, особенно у больших матриц;
- условия диагонального преобладания можно добиться перестановкой строк или столбцов матрицы, но проще всего это сделать, приведя исходную матрицу к матрице, заведомо обладающей диагональным преобладанием;
- заранее неизвестно количество итераций (в отличие от метода Гаусса Гаусса с выбором главного элемента, Холецкого). Скорость сходимости зависит в том числе и от начального приближения - чем оно точнее, тем выше скорость сходимости.
- тем не менее, алгоритмическая сложность данного метода $O(n^2)$ лучше чем у метода Гаусса, Гаусса с выбором главного элемента, Холецкого - $O(n^3)$;
- в сравнении с методом Гаусса-Зейделя, имеет меньшую скорость сходимости, легче параллелизовать, и условие сходимости менее строгое, значит может быть применен в большем количестве случаев.

Также мне удалось реализовать численные методы для работы с матрицами:

- для приведения матрицы к верхнетреугольному виду;
- для вычисления определителя матрицы;
- для получения транспонированной матрицы;
- для получения обратной матрицы;
- для умножения матрицы на матрицу;
- для умножения матрицы на вектор.

А также реализовал численный метод, позволяющий решать СЛАУ с заданной точностью методом простых итераций. На тестовых данных моя реализация работает корректно, находит правильное решение с заданной точностью.