Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №1

по «**Вычислительной математике**»

Системы линейных алгебраических уравнений

Выполнил:

Студент группы P3233

Нгуен Нгок Дык

Преподаватели:

Перл О.В.

Санкт-Петербург

2022

1. **Описание метода. Расчетные формулы.**

Метод последовательных итераций — метод, позволяющий находить значения вектора неизвестных с заданной точностью. В его основе лежит итерационный подход: для нахождения значений на k + 1 шаге необходимо знать значения на k шаге. Также его можно оптимизировать, взяв в качестве неизвестных уже вычисленные значения на данном шаге.

<u>Расчетные формулы:</u>

Пусть дана СЛАУ:

$$\left.\begin{array}{l} a_{11}x_1 + a_{12}x_2 + \ \dots\ + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \ \dots\ + a_{2n}x_n = b_2, \\ \dots\dots\dots\dots\dots\dots\dots \\ a_{n1}x_1 + a_{n2}x_2 + \ \dots\ + a_{nn}x_n = b_n. \end{array}\right\}$$

Для вычисления системы решений необходимо привести СЛАУ к виду: $x = \beta + \alpha x.$

$$\alpha = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \dots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \dots & \alpha_{2n} \\ \dots & \dots & \dots & \dots \\ \alpha_{n1} & \alpha_{n2} & \dots & \alpha_{nn} \end{bmatrix} \quad \text{и} \quad \beta = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \cdot \\ \cdot \\ \cdot \\ \beta_n \end{bmatrix},$$

$$\beta_i = \frac{b_i}{a_{ii}}; \qquad \alpha_{ij} = -\frac{a_{ij}}{a_{ii}} \qquad \text{при } i \neq j$$

Для этого запишем исходное СЛАУ в матричном виде:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \qquad x = \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}, \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ \cdot \\ \cdot \\ \cdot \\ b_n \end{bmatrix},$$

После того, как привели матрицу к виду, где максимальные элементы стоят на главной диагонали, разделим по строкам матрицу A:

$$[a_{i,1}, a_{i,2}, \dots, a_{i,n}]$$

Сохраним в переменную значение $a_{i,i}$ и занулим данный элемент в строке. Затем каждый элемент матрицы умножим на $-a_{i,i}^{-1}$. Затем произведем операцию матричного умножения:

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

В результате получим значения матрицы неизвестных на k-ом шаге.

Для вычисления столбца погрешностей необходимо найти разность столбца неизвестных на k-ом и k+1 шагах.

2. Листинг

2.1 Solver

```
package com.company.commands;

import com.company.receivers.Matrix;

public class JacobiMethodSolver implements Command{
    private final Matrix matrix;

    public JacobiMethodSolver(Matrix matrix) { this.matrix = matrix; }
    @Override
    public void execute() {
        System.out.println("> Initial matrix:");
        matrix.printMatrix();
        if(this.matrix.isStrictDiagonalDominant()){
            System.out.println("> Matrix is strictly diagonally dominant");
        } else{
            System.out.println("> Matrix is NOT strictly diagonally dominant");
            if(matrix.canSearchDiagonalDominant()){
                if(matrix.isStrictDiagonalDominant()){
                    System.out.println("> Interchanging row...");
                    System.out.println("> New matrix:");
                    matrix.printMatrix();
                } else{
                    System.out.println("> Can't find solution!");
                    System.exit( status: 0);
                }
            } else{
                System.out.println("> Can't find solution!");
                System.exit( status: 0);
            }
        }
        matrix.setTransformedMatrix();
        System.out.println();
        System.out.print("Number of iterations: " + matrix.findSolution());
        matrix.printSolution();
    }
}
```

2.2 Matrix implementation

```
public class SimpleItrMatrixImpl implements Matrix{

    private int size;
    private double[][] matrix;
    private double[] coefficients;
    private double[][] transformedMatrix;
    private double accuracy;
    private double[] solution;
    private double[] error;


    public SimpleItrMatrixImpl(int size, double[][] matrix, double[] coefficients, double accuracy){
        this.size = size;
        this.matrix = matrix;
        this.coefficients = coefficients;
        this.accuracy = accuracy;
    }

    public SimpleItrMatrixImpl(){};
```

```java
@Override
public void read(ReadOption option) {
    Scanner sc = new Scanner(System.in);
    try {
        switch (option) {
            case CLI:
                System.out.println("> Read matrix from console...");
                break;
            case FILE:
                System.out.println("> Reading from file...");
                System.out.print("> Please enter file name: ");
                File file = new File(sc.next());
                sc = new Scanner(file);
                break;
            default: throw new UnexpectedException("Can not parse input option");
        }
    } catch (UnexpectedException | FileNotFoundException e) {
        e.printStackTrace();
        System.exit( status: 0);
    }
    try {
        if (option == ReadOption.CLI) System.out.print("> Please enter the accuracy: ");
        this.accuracy = Double.parseDouble(sc.next());
        if (option == ReadOption.CLI) System.out.print("> Please enter the matrix's size: ");
        this.size = sc.nextInt();
        if (option == ReadOption.CLI) System.out.println("> Please enter the matrix: ");
        matrix = new double[this.size][this.size];
        coefficients = new double[this.size];
        for (int row = 0; row < this.size; ++row) {
            for (int col = 0; col < this.size + 1; ++col) {
                String val = sc.next();
                val = val.replace( target: ",",  replacement: ".");
                if(col == this.size) this.coefficients[row] = Double.parseDouble(val);
                else this.matrix[row][col] = Double.parseDouble(val);
            }
        }
    } catch(NumberFormatException e){
        System.out.println("Input is incorrect format, please check again!");
        System.exit( status: 0);
    }
}
```

```java
@Override
public boolean canSearchDiagonalDominant() {
    Matrix copy = new SimpleItrMatrixImpl(this.size, new double[this.size][this.size], new double[this.size], this.accuracy);
    boolean[] done = new boolean[this.size];
    for (int i = 0; i < size; i++) done[i] = false;

    for(int i = 0; i < this.size; i++){
        double sum = 0;
        for (int j = 0 ; j < this.size; j++){
            sum += Math.abs(this.matrix[i][j]);
        }
        boolean flag = false;
        int index = 0;
        for (int j = 0; j < size; j++){
            if(Math.abs(matrix[i][j]) >= sum - Math.abs(matrix[i][j])){
                flag = true;
                index = j;
                break;
            }
        }
        if (flag){
            if (!done[index]){
                for (int k = 0; k < size; k++){
                    copy.getMatrix()[index][k] = this.matrix[i][k];
                }
                copy.getCoefficients()[index] = this.coefficients[i];
                done[index] = true;
            } else return false;
        } else return false;
    }
    this.matrix = copy.getMatrix();
    this.coefficients = copy.getCoefficients();
    return true;
}
```

```java
@Override
public void setTransformedMatrix() {
    this.transformedMatrix = new double[size][size + 1];
    for(int i = 0; i < size; i++){
        for(int j = 0; j < size + 1; j++){
            if (i == j) transformedMatrix[i][j] = 0;
            else {
                if(j == size) transformedMatrix[i][j] = this.coefficients[i] / this.matrix[i][i];
                else transformedMatrix[i][j] = -1 * (this.matrix[i][j] / (this.matrix[i][i]));
            }
        }
    }
}

@Override
public int findSolution() {
    double[] newApproxVector = new double[this.size];
    solution = new double[this.size];
    error = new double[this.size];
    int iteration = 0;
    for (int i = 0; i < size; i++) {
        solution[i] = 0;
    }
    while (true) {
        for (int i = 0; i < size; i++) {
            double x = 0;
            for (int j = 0; j < size; j++) {
                x += this.transformedMatrix[i][j] * solution[j];
                if (j == size - 1) {
                    x += this.transformedMatrix[i][j + 1];
                }
            }
            error[i] = Math.abs(solution[i] - (x));
            newApproxVector[i] = x;
            if (i == size - 1) {
                for (int l = 0; l < size; l++) {
                    solution[l] = newApproxVector[l];
                }
                iteration++;
                boolean flag = true;
                for (int k = 0; k < size; k++) {
                    if (error[k] > accuracy) {
                        flag = false;
                        break;
                    }
                }
                if (flag) return iteration;
            }
        }
    }
}
```

3. Тестовые данные

```
0.0001
8
4 -1 -1 0 0 0 0 0 18
-1 4 -1 -1 0 0 0 0 18
0 -1 4 -1 -1 0 0 0 4
0 0 -1 4 -1 -1 0 0 4
0 0 0 -1 4 -1 -1 0 26
0 0 0 0 -1 4 -1 -1 16
0 0 0 0 0 -1 4 -1 10
0 0 0 0 0 0 -1 4 32
```

```
> Welcome to simple iterative method for solving linear systems!!

> Usage: [options]
> Options:
 -h        Help
 -f        Read data from file
 -r        Randomly generate data

> Reading from file...
> Please enter file name: Lab_1/8_strict.txt
> Initial matrix:
> Matrix accuracy: 1.0E-4
> Matrix size: 8
> Matrix:
> 4.000000 | -1.000000 | -1.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 18.000000 |
> -1.000000 | 4.000000 | -1.000000 | -1.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 18.000000 |
> 0.000000 | -1.000000 | 4.000000 | -1.000000 | -1.000000 | 0.000000 | 0.000000 | 0.000000 | 4.000000 |
> 0.000000 | 0.000000 | -1.000000 | 4.000000 | -1.000000 | -1.000000 | 0.000000 | 0.000000 | 4.000000 |
> 0.000000 | 0.000000 | 0.000000 | -1.000000 | 4.000000 | -1.000000 | -1.000000 | 0.000000 | 26.000000 |
> 0.000000 | 0.000000 | 0.000000 | 0.000000 | -1.000000 | 4.000000 | -1.000000 | -1.000000 | 16.000000 |
> 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | -1.000000 | 4.000000 | -1.000000 | 10.000000 |
> 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | -1.000000 | 4.000000 | 32.000000 |
> Matrix is strictly diagonally dominant

Number of iterations: 24
Vector Solution:
9.999899 | 11.999873 | 9.999885 | 9.999912 | 13.999941 | 11.999965 | 7.999982 | 9.999992 |
Vector Error:
0.000068 | 0.000086 | 0.000077 | 0.000059 | 0.000040 | 0.000024 | 0.000012 | 0.000005 |

Process finished with exit code 0
```

Main.java × App.java ×

```
0.001
3
1 1 1 3
2 2 2 6
3 3 3 9
```

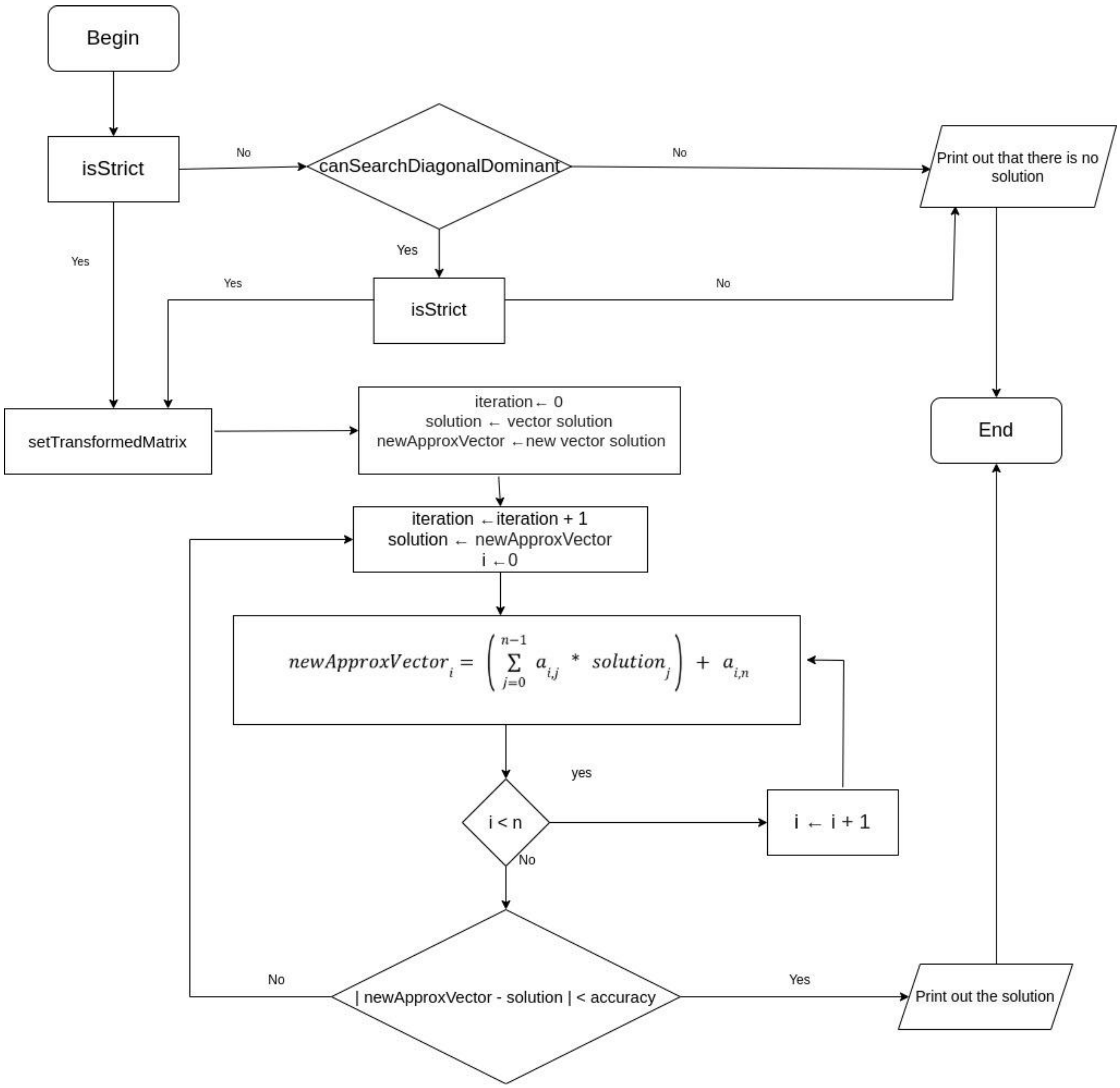
```
> Welcome to simple iterative method for solving linear systems!!

> Usage: [options]
> Options:
 -h        Help
 -f        Read data from file
 -r        Randomly generate data

> Reading from file...
> Please enter file name: Lab_1/3_impossible.txt
> Initial matrix:
> Matrix accuracy: 0.001
> Matrix size: 3
> Matrix:
> 1.000000 | 1.000000 | 1.000000 | 3.000000 |
> 2.000000 | 2.000000 | 2.000000 | 6.000000 |
> 3.000000 | 3.000000 | 3.000000 | 9.000000 |
> Matrix is NOT strictly diagonally dominant
> Can't find solution!

Process finished with exit code 0
```

4. Схема програмы



5. Вывод

При выполнению лабораторной работе, я изучил метод простой итерации для решения системы линейных уравнений.

По сравнению с методом Гаусса - Зейделя, этот метод требуется новый вектор для хранения нового элемента, но его можно вычислить с помощью параллельных вычислений, что нельзя сделать с помощью метода Гаусса-Зейделя, поскольку вычисления новых элементов также зависят от новый итерации. С другой стороны, для метода Гаусса - Зейделя использует меньше памяти, поскольку он помещает новые элементы прямо в текущий вектор итерации.

По сравнению с точным методом, этот метод не только дает менее точные результаты, но также не может точно определить случай, когда нет решения или существует бесконечный набор решений. Но если проблема не в точности, то итерационный метод может быть выполнен быстрее за несколько итераций. Точный метод всегда должен найти решение во временной сложности O(n^3), которая на самом деле огромна при больших n