**Санкт-Петербургский Национальный Исследовательский Университет ИТМО**

**Лабораторная работа №2**

по дисциплине

*«Вычислительная математика»*

Выполнил: Анисимов М. Д.

Группа: Р3233

Преподаватель: Перл О. В.

Санкт-Петербург

2024 г.

**Цель работы**

Разработать программу, которая вычисляет СЛАУ методом Холецкого. В качестве входных данных программа получает размер матрицы n, матрицу элементов и вектор b. Программа должна выводить конечный вектор x и промежуточный вектор y

**Описание метода**

Метод основан на решении СЛАУ вида , где – квадратная матрица размера n и:

Для решения необходимо представить матрицу А в виде произведения нижнедиагональной матрицы и верхнедиагональной матрицы , где на главной диагонали будут содержаться 1. Тогда , где:

Компоненты определяются по формулам:

Далее из формул выразим вектор y:

А после выразим вектор х:

Результатом метода являются два вектора вещественных чисел

**Полный код программы**

import java.io.\*;

import java.util.\*;

import java.util.stream.\*;

import static java.lang.Math.\*;

import static java.util.stream.Collectors.joining;

import static java.util.stream.Collectors.toList;

class Result {

public static boolean isSolutionExists = true;

public static String errorMessage = "The system has no roots of equations or has an infinite set of them.";

public static boolean isInputMatrixSymmetrical(List<List<Double>> inputMatrix) {

if (inputMatrix.size() < inputMatrix.get(0).size() - 1) {

isSolutionExists = false;

return isSolutionExists;

}

return isSolutionExists;

}

public static Double[] getVectorB(List<List<Double>> inputMatrix) {

Double[] vectorB = new Double[inputMatrix.size()];

for (int i = 0; i < inputMatrix.size(); i++) {

for (int j = 0; j < inputMatrix.get(i).size(); j++) {

if (j == inputMatrix.get(i).size() - 1) vectorB[i] = inputMatrix.get(i).get(j);

}

}

return vectorB;

}

public static void fillMatrixes(int n, List<List<Double>> inputMatrix, Double[][] matrixB, Double[][] matrixC){

for(int i=0; i<n; i++){

for(int j=0; j<n; j++){

if(j == 0) {

matrixB[i][j] = inputMatrix.get(i).get(j);

}

else if(i >= j && j > 0){

double sum = 0.0;

for(int k=0; k< j; k++){

sum = sum + matrixB[i][k] \* matrixC[k][j];

}

matrixB[i][j] = inputMatrix.get(i).get(j) - sum;

}

else {

matrixB[i][j]= 0.0;

}

if(0 < i && i < j){

double sum = 0.0;

for(int k=0; k<i; k++){

sum = sum + matrixB[i][k] \* matrixC[k][j];

}

matrixC[i][j] = (inputMatrix.get(i).get(j)- sum) / matrixB[i][i];

}

else if(i == 0){

matrixC[i][j] = inputMatrix.get(i).get(j) / matrixB[i][i];

}

else if(i==j){

matrixC[i][j] = 1.0;

}

else {

matrixC[i][j] = 0.0;

}

}

}

}

public static void makeInversionMatrix(Double[][] matrix, int matrixSize) {

double temp;

Double[][] identityMatrix = new Double[matrixSize][matrixSize];

for (int i = 0; i < matrixSize; i++)

for (int j = 0; j < matrixSize; j++) {

identityMatrix[i][j] = 0.0;

if (i == j) identityMatrix[i][j] = 1.0;

}

for (int k = 0; k < matrixSize; k++) {

temp = matrix[k][k];

for (int j = 0; j < matrixSize; j++) {

if (temp == 0.0) {

isSolutionExists = false;

return;

}

else {

matrix[k][j] /= temp;

identityMatrix[k][j] /= temp;

}

}

for (int i = k + 1; i < matrixSize; i++) {

temp = matrix[i][k];

for (int j = 0; j < matrixSize; j++) {

matrix[i][j] -= matrix[k][j] \* temp;

identityMatrix[i][j] -= identityMatrix[k][j] \* temp;

}

}

}

for (int k = matrixSize - 1; k > 0; k--) {

for (int i = k - 1; i >= 0; i--) {

temp = matrix[i][k];

for (int j = 0; j < matrixSize; j++) {

matrix[i][j] -= matrix[k][j] \* temp;

identityMatrix[i][j] -= identityMatrix[k][j] \* temp;

}

}

}

for (int i = 0; i < matrixSize; i++) {

for (int j = 0; j < matrixSize; j++) matrix[i][j] = identityMatrix[i][j];

}

}

public static Double[] getCalculatedVector(Double[][] matrix, Double[] vectorB) {

Double[] vectorY = new Double[vectorB.length];

Double elementValue = 0.0;

for (int i = 0; i < matrix.length; i++) {

for (int j = 0; j < matrix.length; j++) {

elementValue = elementValue + vectorB[j] \* matrix[i][j];

}

vectorY[i] = elementValue;

elementValue = 0.0;

}

return vectorY;

}

public static List<Double> solveByCholeskyDecomposition(int n, List<List<Double>> matrix) {

if(!isInputMatrixSymmetrical(matrix)){

return null;

}

Double[] vectorB = getVectorB(matrix);

Double[][] matrixB = new Double[n][n];

Double[][] matrixC = new Double[n][n];

fillMatrixes(n, matrix, matrixB, matrixC);

makeInversionMatrix(matrixB, n);

makeInversionMatrix(matrixC, n);

Double[] vectorY = getCalculatedVector(matrixB, vectorB);

Double[] vectorX = getCalculatedVector(matrixC, vectorY);

List<Double> result = new ArrayList<>();

result.addAll(Arrays.stream(vectorY).toList());

result.addAll(Arrays.stream(vectorX).toList());

return result;

}

}

public class Main {

public static void main(String[] args) throws IOException {

BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(System.in));

BufferedWriter bufferedWriter = new BufferedWriter(new OutputStreamWriter(System.out));

int n = Integer.parseInt(bufferedReader.readLine().trim());

int matrixRows = n;

int matrixColumns = n + 1;

List<List<Double>> matrix = new ArrayList<>();

IntStream.range(0, matrixRows).forEach(i -> {

try {

matrix.add(

Stream.of(bufferedReader.readLine().replaceAll("\\s+$", "").split(" "))

.map(Double::parseDouble)

.collect(toList())

);

} catch (IOException ex) {

throw new RuntimeException(ex);

}

});

List<Double> result = Result.solveByCholeskyDecomposition(n, matrix);

if (Result.isSolutionExists) {

bufferedWriter.write(

result.stream()

.map(Object::toString)

.collect(joining("\n"))

+ "\n"

);

} else {

bufferedWriter.write(Result.errorMessage + "\n");

}

bufferedReader.close();

bufferedWriter.close();

}

}

**Примеры работы программы**

(Зелёные цифры – входные данные. Белые цифры – результат)

|  |  |
| --- | --- |
| Пример №1 |  |
| Пример №2 |  |
| Пример №3 |  |
| Пример №4 |  |
| Пример №5 |  |

**Блок-схема работы программы**

<https://github.com/MyximAnisimov/itmo_study/blob/main/2_year/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F%20%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0/%D0%9B%D0%B0%D0%B1%D0%BE%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%BD%D0%B0%D1%8F%202/Labwork_2_new/diagram.md>

**Вывод по работе программы**

Программа и алгоритм работают в том случае, когда матрица является совместной и определённой. При вычислении обратных матриц необходимо, чтобы их определитель не был равен нулю. Так же по условию лабораторной задаётся квадратная матрица. Если же входная матрица не симметричная, то выводится ошибка. Метод Холецкого является аналогом метода Гаусса. Используется для решения СЛАУ, ОДУ и аппроксимации наименьшими квадратами. Метод применим в случае, когда матрицу можно разложить через LU-метод (Когда из входной матрицы можно сделать верхнедиагональную и нижнедиагональную матрицу, произведение которых равно входной матрице). Также при вычислении элементов верхне- и нижнедиагональной матрицы необходимо проверять, нет ли при вычислении элементов нуля в знаменателе (К сожалению, забыл добавить в метод fillMatrixes() данную проверку).  
Алгоритмическая сложность метода составляет O(n3) (В худшем случае).   
Алгоритм вычисляет векторы с небольшими погрешностями в дробной части числа в силу точности вычислений в Джава (Пример: вместо числа 6 программа может вывести в ответ 5.999999999…)