МИНОБРНАУКИ РОССИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «ВГУ»)**

Факультет прикладной математики, информатики и механики

Кафедра математического обеспечения ЭВМ

**Численные методы решения**

**спектральных задач линейной алгебры**

Отчёт по лабораторной работе

Вариант 16

Студент 3 курса                                                 \_\_\_\_\_\_\_ М.О. Курченков

Преподаватель                                                   \_\_\_\_\_\_\_     О.А. Махинова

Воронеж 2023

**Содержание**

[Постановка задачи 3](#_Toc151595658)

[Анализ задачи 4](#_Toc151595659)

Этапы решения [5](#_Toc151595659)

Тестирование алгоритма6

[Вычислительный эксперимент](#_Toc151595660) 8

[Приложение](#_Toc151595664) 10

ThirdLabApplication.java10

Matrix[.java 1](#_Toc151595666)1

JacobiRotationMethod[.java 1](#_Toc151595667)5

ThirdLabTests.java17

# Постановка задачи

1. Реализовать метод вращений Якоби для поиска собственных значений симметричных матриц.
2. Провести вычислительные эксперименты.
3. Проанализировать полученные результаты.

# Анализ задачи

Матрица в данном варианте представлена как 2 массива: diagValues ∈ Rn, values ∈ Rn x Rn и как N — размерность матрицы. Выполнение метода вращений Якоби ограничено целочисленным значением M и действительным положительным числом eps:

* N – размерность системы;
* values – двумерный массив размерности N\*N, содержащий значения элементов матрицы;
* diagValues – одномерный массив размерности N, содержащий значения диагональных элементов матрицы;
* M – максимальное число вращений;
* eps – максимальное по модулю значение внедиагональных элементов преобразованной матрицы.

**Этапы решения**

Решение представлено одним этапом – циклом вращений исходной матрицы до исчерпания количества вращений или условием выхода при достаточно низких значениях внедиагональных элементов:

1. Вращение Якоби:

1.1) Найдём максимальный наддиагональный элемент матрицы A и его индексы в матрице (i, j).

1.2) Вычислим угол поворота φ, используя формулу:

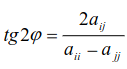


Рис. 1 — формула угла поворота

1.3) Создадим единичную матрицу T и изменим некоторые её значения, как указано ниже:

T[i,i] = T[j,j] = cos(φ),

T[i,j] = -sin(φ),

T[j,i] = sin(φ).

1.4) Применим формулу матричного умножения к A. Результатом вычисления будет являться матрица после одной итерации вращения Якоби:



Рис. 2 — формула матричного умножения

# Тестирование алгоритма

Используем заранее сформированную матрицу небольшой размерности и вычислим её собственные значения:



Рис. 3 — значения тестовой матрицы

Используя онлайн-калькулятор, вычислим её собственные значения:

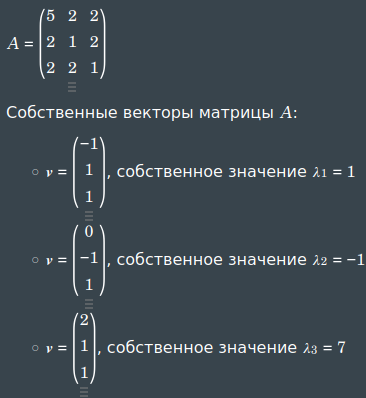


Рис. 4 — собственные значения согласно онлайн-калькулятору

Как нетрудно заметить при проверке, значения действительно являются собственными и при подстановке образуют зависимые строки. Теперь используем функцию JacobiRotationMethod.apply(...) для поиска собственных значений:

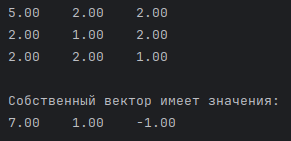
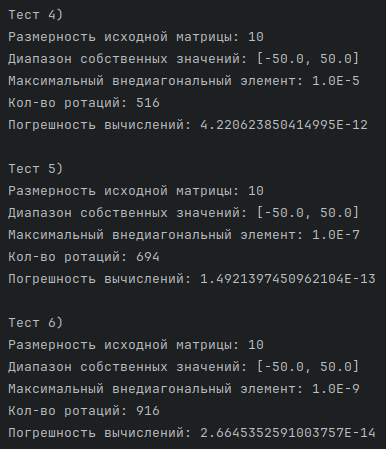
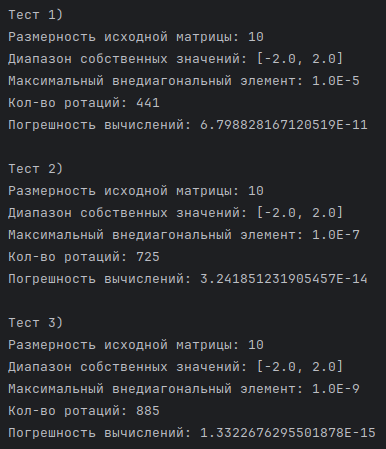


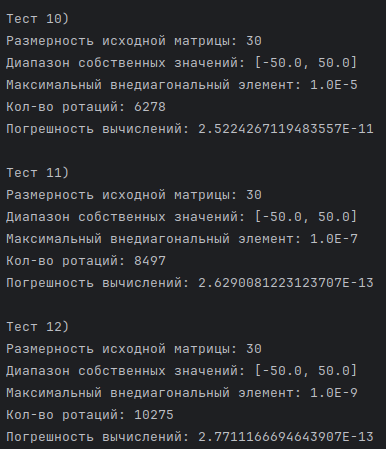
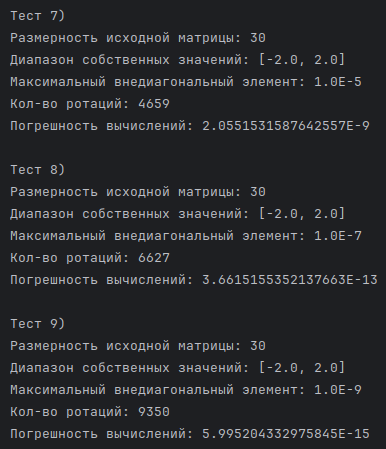
Рис. 5 – результаты выполнения метода

Как мы видим по результатам выполнения метода, собственные значения найдены верно.

# Вычислительный эксперимент

В ходе данного вычислительного эксперимента формируется матрица с заранее известными собственными значениями с использованием вспомогательной матрицы Хаусхолдера. Затем к матрице применяется программная реализация метода вращений Якоби, благодаря которой находится вектор собственный значений. Затем исследуется погрешность вычисления. Эксперимент проходит для различных значений, по которым формируется исследуемая матрица: размерность системы (10, 30) и диапазоны собственных значений ([-2, 2], [-50, 50]). Помимо этого, варьируется параметр eps, отвечающий за критерий останова при уменьшении значений наддиагональных элементов, разных порядков (5, 7, 9). Результаты представлены на рис. 6-9:

Рис. 6-7 — результаты тестов 1-6

Рис. 8-9 — результаты тестов 7-12

Как мы видим по результатам тестов, на погрешность негативно влияют увеличение размерности исходной матрицы, расширение диапазона значений и повышение порога критерия останова, выраженном параметром eps. По полученным результатам можно выдвинуть гипотезу о том, что увеличение значения максимального внедиагонального элемента, ограничивающего количество итераций алгоритма, наносит наибольший ущерб вычислениям, сильнее всего влияя на погрешность.

# **Приложение**

**SecondLabApplication,java**

package com.k4r3l1ns;  
  
import com.k4r3l1ns.models.Matrix;  
import com.k4r3l1ns.service.JacobiRotationMethod;  
  
import java.io.FileInputStream;  
import java.io.IOException;  
  
public class ThirdLabApplication {  
  
 public static final String *SOURCE\_DIRECTORY* = "/home/k4r3l1ns/Desktop/NM\_labs/third\_lab/src/main/resources/";  
  
 public static final String *MATRIX\_FILENAME* = "matrix.txt";  
  
 public static final int *ROTATION\_LIMIT* = 1000000000;  
 public static final double *EPSILON* = 1e-3;  
  
 public static void main(String[] args) {  
  
 try {  
  
 Matrix symmetricMatrix =  
 Matrix.*read*(new FileInputStream(*SOURCE\_DIRECTORY* + *MATRIX\_FILENAME*));  
  
 symmetricMatrix.print();  
  
 double[] eigenvector = JacobiRotationMethod.*apply*(symmetricMatrix, *ROTATION\_LIMIT*, *EPSILON*);  
  
 System.*out*.println("\nСобственный вектор имеет значения:");  
 for (double value : eigenvector) {  
 System.*out*.printf("%.2f\t", value);  
 }  
 System.*out*.println();  
  
 } catch (IOException ex) {  
 ex.printStackTrace();  
 }  
 }  
}

**Matrix.****java**

package com.k4r3l1ns.models;  
  
import lombok.Getter;  
import lombok.NoArgsConstructor;  
  
import java.io.InputStream;  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Scanner;  
import java.util.concurrent.ThreadLocalRandom;  
  
@Getter  
@NoArgsConstructor  
public class Matrix {  
  
 private double[][] values;  
 private double[] diagValues;  
 int size = -1;  
  
 public Matrix(int size) {  
 this.size = size;  
 values = new double[size][size];  
 diagValues = new double[size];  
 }  
  
 public static Matrix read(InputStream inputStream) {  
  
 Scanner scanner = new Scanner(inputStream);  
  
 int lineIndex = 0;  
 Matrix result = new Matrix();  
 result.size = -1;  
  
 while (  
 scanner.hasNextLine() &&  
 (result.size == -1 || lineIndex < result.size)  
 ) {  
 String line = scanner.nextLine();  
  
 List<Double> row = new ArrayList<>();  
 Scanner scanner2 = new Scanner(line);  
  
 while (scanner2.hasNext()) {  
 row.add(scanner2.nextDouble());  
 }  
  
 if (result.size == -1) {  
 result = new Matrix(row.size());  
 } else if (result.size != row.size()) {  
 throw new RuntimeException("Входные данные не являются матрицей");  
 }  
  
 result.values[lineIndex] = row.stream().mapToDouble(x -> x).toArray();  
  
 ++lineIndex;  
 }  
  
 for (int index = 0; index < result.size; ++index) {  
 result.diagValues[index] = result.values[index][index];  
 }  
  
 *throwIfNotSymmetric*(result);  
  
 return result;  
 }  
  
 public void print() {  
  
 for (int i = 0; i < size; ++i) {  
 for (var value : values[i]) {  
 System.*out*.printf("%.2f\t", value);  
 }  
 System.*out*.println();  
 }  
 }  
  
 private static void throwIfNotSymmetric(Matrix matrix) {  
  
 if (!*isSymmetric*(matrix)) {  
 throw new RuntimeException("Матрица не симметрична");  
 }  
 }  
  
 private static boolean isSymmetric(Matrix matrix) {  
  
 if (matrix.size == -1) {  
 return false;  
 }  
  
 for (int lineIndex = 0; lineIndex < matrix.size; ++lineIndex) {  
 for (int columnIndex = lineIndex + 1; columnIndex < matrix.size; ++columnIndex) {  
 if (  
 matrix.values[lineIndex][columnIndex] !=  
 matrix.values[columnIndex][lineIndex]  
 ) {  
 return false;  
 }  
 }  
 }  
  
 return true;  
 }  
  
 public static Matrix copyOf(Matrix matrix) {  
  
 Matrix copy = new Matrix(matrix.size);  
  
 System.*arraycopy*(matrix.values, 0, copy.values, 0, copy.size);  
 System.*arraycopy*(matrix.diagValues, 0, copy.diagValues, 0, copy.size);  
  
 return copy;  
 }  
  
 public double findNorm() {  
  
 double norm = 0;  
  
 for (int lineIndex = 0; lineIndex < size; ++lineIndex) {  
 for (int columnIndex = lineIndex + 1; columnIndex < size; ++columnIndex) {  
 double currentValue = Math.*abs*(values[lineIndex][columnIndex]);  
 if (currentValue > norm) {  
 norm = currentValue;  
 }  
 }  
 }  
  
 return norm;  
 }  
  
 public int[] findMaxElemIndexes() {  
  
 int[] result = { -1, -1 };  
 double norm = 0;  
  
 for (int lineIndex = 0; lineIndex < size; ++lineIndex) {  
 for (int columnIndex = lineIndex + 1; columnIndex < size; ++columnIndex) {  
 double currentValue = Math.*abs*(values[lineIndex][columnIndex]);  
 if (currentValue > norm) {  
 norm = currentValue;  
 result[0] = lineIndex;  
 result[1] = columnIndex;  
 }  
 }  
 }  
  
 return result;  
 }  
  
 public static Matrix identityMatrix(int size) {  
  
 Matrix result = new Matrix(size);  
 for (int i = 0; i < size; ++i) {  
 result.values[i][i] = 1;  
 result.diagValues[i] = 1;  
 }  
 return result;  
 }  
  
 public Matrix transpose() {  
  
 Matrix result = new Matrix(size);  
  
 for (int i = 0; i < size; ++i) {  
 for (int j = 0; j < size; ++j) {  
 result.values[j][i] = values[i][j];  
 }  
 }  
  
 result.diagValues = diagValues.clone();  
  
 return result;  
 }  
  
 public static Matrix multiply(Matrix matrix1, Matrix matrix2) {  
  
 int size = matrix1.size;  
 Matrix result = new Matrix(size);  
  
 for (int i = 0; i < size; ++i) {  
 for (int j = 0; j < size; ++j) {  
 for (int k = 0; k < size; ++k) {  
 result.values[i][j] += matrix1.values[i][k] \* matrix2.values[k][j];  
 }  
 }  
 }  
  
 for (int i = 0; i < size; ++i) {  
 result.diagValues[i] = result.values[i][i];  
 }  
  
 return result;  
 }  
  
 */\*\**  
 *\* Матрица, полученная преобразованием Хаусхолдера по вектору собственных значений*  
 *\**  
 *\* @param eigenvector Вектор собственных значений*  
 *\* @return Матрица с заранее известными собственными значениями*  
 *\*/*  
public static Matrix predefinedMatrix(double[] eigenvector, double[] range) {  
  
 int size = eigenvector.length;  
  
 Matrix result = new Matrix(size);  
 for (int i = 0; i < size; ++i) {  
 result.values[i][i] = result.diagValues[i] = eigenvector[i];  
 }  
  
 double z = 0;  
 double[] randomVector = new double[size];  
 for (int i = 0; i < size; ++i) {  
 randomVector[i] = ThreadLocalRandom.*current*().nextDouble(range[0], range[1]);  
 z += randomVector[i] \* randomVector[i];  
 }  
 z = Math.*pow*(z, 0.5);  
 for (int i = 0; i < size; ++i) {  
 randomVector[i] /= z;  
 }  
  
 Matrix householderMatrix = new Matrix(size);  
 for (int i = 0; i < size; ++i) {  
 for (int j = 0; j < size; ++j) {  
 if (i == j) {  
 householderMatrix.values[i][i] =  
 householderMatrix.diagValues[i] = 1 - 2 \* randomVector[i] \* randomVector[i];  
 } else {  
 householderMatrix.values[i][j] = -2 \* randomVector[i] \* randomVector[j];  
 }  
 }  
 }  
  
 result = Matrix.*multiply*(householderMatrix, result);  
 result = Matrix.*multiply*(result, Matrix.*copyOf*(householderMatrix).transpose());  
  
 return result;  
 }  
}

**JacobiRotationMethod****.****java**

package com.k4r3l1ns.service;  
  
import com.k4r3l1ns.models.Matrix;  
import lombok.Getter;  
import lombok.Setter;  
  
@Getter  
@Setter  
public class JacobiRotationMethod {  
  
  
 */\*\**  
 *\* Максимальное по модулю значение внедиагональных*  
 *\* элементов преобразованной матрицы*  
 *\*/*  
double eps;  
  
 */\*\**  
 *\* Поиск собственных значений симметричной матрицы методом вращения Якоби*  
 *\**  
 *\* @param symmetricMatrix Симметричная матрица*  
 *\* @param rotationLimit Максимальное допустимое число вращений*  
 *\* @param eps Максимальное по модулю значение внедиагональных элементов преобразованной матрицы*  
 *\* @return Собственный вектор*  
 *\*/*  
public static double[] apply(  
 Matrix symmetricMatrix,  
 int rotationLimit,  
 double eps  
 ) {  
 Matrix matrix = Matrix.*copyOf*(symmetricMatrix);  
 int size = matrix.getSize();  
  
 double[] eigenvector = matrix.getDiagValues();  
  
 int rotationCount;  
 for (rotationCount = 0; rotationCount < rotationLimit; ++rotationCount) {  
  
 double norm = matrix.findNorm();  
 if (norm < eps) {  
 break;  
 }  
  
 int[] maxElemIndexes = matrix.findMaxElemIndexes();  
 int p = maxElemIndexes[0];  
 int q = maxElemIndexes[1];  
  
 var diagValues = matrix.getDiagValues();  
 double rotationAngle = 0.5 \* Math.*atan2*(  
 2 \* norm, (diagValues[q] - diagValues[p])  
 );  
  
 Matrix matrixT = Matrix.*identityMatrix*(size);  
 var matrixTDiagValues = matrixT.getDiagValues();  
 var matrixTValues = matrixT.getValues();  
 matrixTDiagValues[p] = matrixTDiagValues[q] =  
 matrixTValues[p][p] =  
 matrixTValues[q][q] = Math.*cos*(rotationAngle);  
 matrixTValues[p][q] = -Math.*sin*(rotationAngle);  
 matrixTValues[q][p] = Math.*sin*(rotationAngle);  
  
 matrix = Matrix.*multiply*(  
 Matrix.*copyOf*(matrixT).transpose(),  
 matrix  
 );  
 matrix = Matrix.*multiply*(matrix, matrixT);  
  
 eigenvector = matrix.getDiagValues();  
 }  
  
// System.out.println("Вычисление выполнено успешно.");  
 System.*out*.println("Кол-во ротаций: " + rotationCount);  
 return eigenvector;  
 }  
}

**ThirdLabTests****.****java**

import com.k4r3l1ns.models.Matrix;  
import com.k4r3l1ns.service.JacobiRotationMethod;  
import org.junit.jupiter.api.Test;  
  
import java.util.Arrays;  
import java.util.concurrent.ThreadLocalRandom;  
  
public class ThirdLabTests {  
  
 public int[] SIZES = { 10, 30 };  
  
 public double[][] RANGES = {  
 { -2, 2 },  
 { -50, 50 }  
 };  
  
 public double[] EPSILONS = { 1e-5, 1e-7, 1e-9 };  
  
 private static int *testNum* = 1;  
  
 @Test  
 public void testAll() {  
  
 for (var size : SIZES) {  
 for (var range : RANGES) {  
 for (var eps : EPSILONS) {  
  
 System.*out*.println("Тест " + *testNum*++ + ")");  
 System.*out*.println("Размерность исходной матрицы: " + size);  
 System.*out*.println("Диапазон собственных значений: " + Arrays.*toString*(range));  
 System.*out*.println("Максимальный внедиагональный элемент: " + eps);  
// System.out.println("Выполнение программы...");  
 var preciseEigenvector = randomVector(size, range);  
 var matrix = Matrix.*predefinedMatrix*(preciseEigenvector, range);  
 var eigenvector = JacobiRotationMethod.*apply*(matrix, 1000000000, eps);  
 System.*out*.println("Погрешность вычислений: " + findFault(eigenvector, preciseEigenvector) + "\n");  
 }  
 }  
 }  
 }  
  
 private double findFault(double[] vector1, double[] vector2) {  
  
 Arrays.*sort*(vector1);  
 Arrays.*sort*(vector2);  
  
 double result = 0;  
 for (int i = 0; i < vector1.length; ++i) {  
 double value = Math.*abs*(vector1[i] - vector2[i]);  
 if (value > result) {  
 result = value;  
 }  
 }  
  
 return result;  
 }  
  
 private double[] randomVector(int size, double[] range) {  
  
 double[] result = new double[size];  
 for (int i = 0; i < size; ++i) {  
 result[i] = ThreadLocalRandom.*current*().nextDouble(range[0], range[1]);  
 }  
 return result;  
 }  
}