**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра вычислительной математики**

**Отчёт**

**Лабораторная работа**

“Метод Данилевского. Степенной метод”

Вариант №2

Коноваловой Алины Николаевны

студентки 3 курса, 3 группы

специальности «Информатика»

дисциплина «Численные методы»

Минск, 2024

Содержание:

Постановка задачи...................................................................................................3

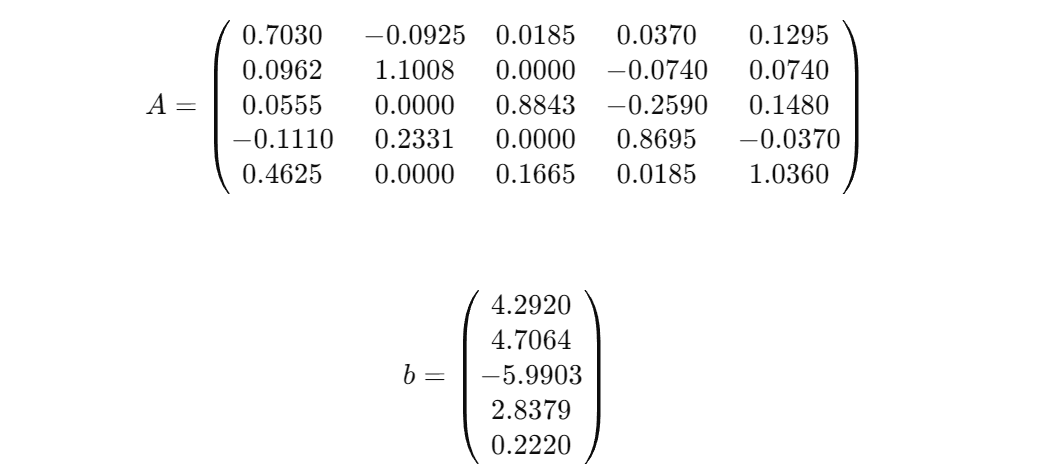
Алгоритм решения..................................................................................................4

Листинг программы.................................................................................................6

Результат и его анализ...........................................................................................10

**Постановка задачи**

Построить собственный многочлен матрицы , найти минимальное собственное значение и собственный вектор, соответствующий этому собственному значению, где:



Заданная точность: eps=1e-5.

**Необходимо:**

1. Построить: – собственный многочлен матрицы , найдя коэффициенты используя метод Данилевского.
2. Проверить точность вычисления вычислив:

;

.

1. Реализовать степенной метод через скалярное произведение;
2. Найти минимальное собственное значение ;
3. Найти собственный вектор x, соответствующий найденному собственному значению;
4. Оценить точность:

;

;

**Алгоритм решения**

Работаем с матрицей

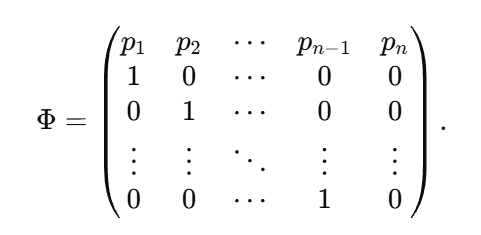
1. **Алгоритм метода Данилевского**

Метод Данилевского относится к прямым методам решения проблемы

собственных значений. Метод основан на подобном преобразовании матрицы: преобразованиями матрица приводится к канонической форме Фробениуса, которая содержит коэффициенты характеристического многочлена.

Матрица приводится к Ф, в результате последовательного домножения справа на и слева на , k = 1, ..., n -1.

Таким образом справедлива формула: , где , положим =. В итоге получим = Ф.



где pi - соответствующий коэффициент (с противоположным знаком) собственного многочлена матрицы , i=.

Для матрицы справедливы следующие формулы:

*, , ,* где

Для матрицы имеем:

1. **Алгоритм степенного метода**

Для того, чтобы найти для матрицы , сведём задачу к нахождению для матрицы .

Теперь перед нами стоит задача нахождения наибольшего по модулю собственного значения и соответствующего ему собственного вектора.

Обозначим собственные значения следующим образом: λ1, . . ., λ𝑛. Будем считать, что все собственные значения матрицы A перенумерованы в порядке невозрастания модулей:

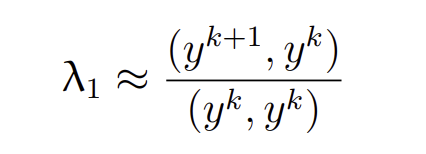
|λ1| > |λ2| ≥ … ≥ |λn|.

Пусть – произвольный ненулевой вектор (положим =[1, 0, …, 0]).

Перейдём к итерационному процессу:

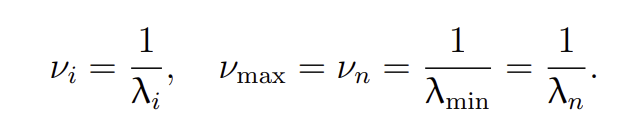
, k = 0, 1, 2, …

В качестве λ1 можно взять:



В качестве критерия для остановки итерационного процесса берём:

Обозначим через , 𝑖 = собственные значения матрицы . Далее, зная соотношения между матрицей и обратной к ней, получаем



Таким образом минимальное собственное значение будет найдено по формуле: .

Собственный вектор x = .

**Листинг кода**

import java.util.Arrays;

import org.apache.commons.math3.linear.\*;

import static org.apache.commons.math3.stat.StatUtils.sum;

public class Main {

public static void main(String[] args) {

int n = 5;

double eps = 1e-5;

double[][] A = {

{0, 0, 0, 0, 0, 0},

{0, 0.7030, -0.0925, 0.0185, 0.0370, 0.1295},

{0, 0.0962, 1.1008, 0.0000, -0.0740, 0.0740},

{0, 0.0555, 0.0000, 0.8843, -0.2590, 0.1480},

{0, -0.1110, 0.2331, 0.0000, 0.8695, -0.0370},

{0, 0.4625, 0.0000, 0.1665, 0.0185, 1.0360}

};

double[][] At = transpose(A);

A = matrixMultiply(At, A);

double[][] A\_inv = invertMatrix(A);

double[] p = danilevskyMethod(A, n);

System.out.println("Коэффициенты собственного многочлена матрицы А:" + Arrays.toString(p));

double[] y0 = new double[n + 1];

y0[1] = 1;

double[] result = powerMethod(A\_inv, y0, eps);

double l\_min = 1/ result[0];

System.out.println("Наименьшее по модулю собственное значение λ1: " + l\_min);

double[] x = Arrays.copyOfRange(result, 1, result.length - 1);

double[] xExtended = new double[x.length + 1];

xExtended[0] = 0;

System.arraycopy(x, 0, xExtended, 1, x.length);

System.out.println("Собственный вектор x: " + Arrays.toString(xExtended));

int iterations = (int) result[result.length - 1];

System.out.println("Количество итераций: " + iterations);

RealMatrix matrix = MatrixUtils.createRealMatrix(A);

RealMatrix subMatrix = matrix.getSubMatrix(1, matrix.getRowDimension() - 1, 1, matrix.getColumnDimension() - 1);

LUDecomposition luDecomposition = new LUDecomposition(subMatrix);

double determinant = luDecomposition.getDeterminant();

EigenDecomposition eigenDecomposition = new EigenDecomposition(subMatrix);

double[] eigenvalues = eigenDecomposition.getRealEigenvalues();

double r1 = p[1] - sum(eigenvalues);

double r2 = p[p.length-1] - determinant;

double r3 = calcP(p, l\_min, 5);

double[] vector1 = matrixVectorMultiply(A,xExtended);

double[] vector2 = scalarMultiply(l\_min, xExtended);

double[] r4 = new double[n+1];

for (int i = 1; i <= 5; i++) {

r4[i] = vector1[i] - vector2[i];

}

double norm = calculateNorm(r4);

System.out.println("r1 = " + r1);

System.out.println("r2 = " + r2);

System.out.println("r3 = " + r3);

System.out.println("r4 = " + Arrays.toString(r4));

System.out.println("||r4|| = " + norm);

}

public static double[] danilevskyMethod(double[][] A, int n) {

double[][] A\_cur = deepCopy(A);

for (int k = 1; k < n; k++) {

double[][] M = new double[n + 1][n + 1];

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

M[i][i] = 1;

}

M[0][0] = 0;

double[][] M\_inv = new double[n + 1][n + 1];

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

M\_inv[i][i] = 1;

}

M\_inv[0][0] = 0;

for (int j = 1; j <= n; j++) {

if (j != n - k) {

M[n - k][j] = -A\_cur[n - k + 1][j] / A\_cur[n - k + 1][n - k];

} else {

M[n - k][j] = 1 / A\_cur[n - k + 1][n - k];

}

M\_inv[n - k][j] = A\_cur[n - k + 1][j];

}

double[][] B = matrixMultiply(A\_cur, M);

A\_cur = matrixMultiply(M\_inv, B);

}

return A\_cur[1];

}

public static double[] powerMethod(double[][] A, double[] y0, double eps) {

int n = 5;

double[] yCur = Arrays.copyOf(y0, n + 1);

double[] yOld;

double lambdaOld = 0.0, lambdaNew;

int iterCount = 0;

while (true) {

iterCount++;

yOld = Arrays.copyOf(yCur, n + 1);

yCur = matrixVectorMultiply(A, yOld);

lambdaNew = dotProduct(yCur, yOld) / dotProduct(yOld, yOld);

if (Math.abs(Math.abs(lambdaNew) - Math.abs(lambdaOld)) <= eps) {

break;

}

lambdaOld = lambdaNew;

}

double[] result = new double[n + 2];

result[0] = lambdaNew;

double[] yNorm = norm(yOld);

System.arraycopy(yNorm, 1, result, 1, n);

result[n + 1] = iterCount;

return result;

}

private static double[] norm(double[] vec) {

double norm = 0.0;

for (double value : vec) {

norm += value \* value;

}

norm = Math.sqrt(norm);

double[] normalizedVec = new double[vec.length];

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

normalizedVec[i] = vec[i] / norm;

}

return normalizedVec;

}

private static double[][] invertMatrix(double[][] matrix) {

int n = matrix.length - 1;

double[][] augmented = new double[n + 1][2 \* n + 1];

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= n; j++) {

augmented[i][j] = matrix[i][j];

}

augmented[i][n + i] = 1;

}

for (int i = 1; i <= n; i++) {

int maxRow = i;

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

if (Math.abs(augmented[k][i]) > Math.abs(augmented[maxRow][i])) {

maxRow = k;

}

}

double[] temp = augmented[i];

augmented[i] = augmented[maxRow];

augmented[maxRow] = temp;

double pivot = augmented[i][i];

for (int j = 1; j <= 2 \* n; j++) {

augmented[i][j] /= pivot;

}

for (int k = 1; k <= n; k++) {

if (k != i) {

double factor = augmented[k][i];

for (int j = 1; j <= 2 \* n; j++) {

augmented[k][j] -= factor \* augmented[i][j];

}

}

}

}

double[][] inverse = new double[n + 1][n + 1];

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= n; j++) {

inverse[i][j] = augmented[i][n + j];

}

}

return inverse;

}

private static double[][] deepCopy(double[][] original) {

int n = original.length;

double[][] copy = new double[n][];

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

copy[i] = original[i].clone();

}

return copy;

}

private static double[][] transpose(double[][] matrix) {

int rows = matrix.length;

int cols = matrix[0].length;

double[][] result = new double[cols][rows];

for (int i = 1; i < rows; i++) {

for (int j = 1; j < cols; j++) {

result[j][i] = matrix[i][j];

}

}

return result;

}

private static double[][] matrixMultiply(double[][] A, double[][] B) {

int n = A.length - 1;

int m = B[0].length - 1;

double[][] result = new double[n + 1][m + 1];

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= m; j++) {

for (int k = 1; k <= B.length - 1; k++) {

result[i][j] += A[i][k] \* B[k][j];

}

}

}

return result;

}

private static double[] matrixVectorMultiply(double[][] A, double[] B) {

int n = A.length - 1;

double[] result = new double[n + 1];

for (int i = 1; i <= 5; i++) {

for (int j = 1; j <= 5; j++) {

result[i] += A[i][j] \* B[j];

}

}

return result;

}

private static double dotProduct(double[] v1, double[] v2) {

double result = 0.0;

for (int i = 1; i <= 5; i++) {

result += v1[i] \* v2[i];

}

return result;

}

public static double calcP(double[] P, double lambd, int n) {

double answer = 0;

answer = Math.pow(-1,5)\*(Math.pow(lambd,5) - P[1]\*Math.pow(lambd,4) -

P[2]\*Math.pow(lambd,3) - P[3]\*Math.pow(lambd,2) -

P[4]\*Math.pow(lambd,1) - P[5]);

return answer;

}

public static double[] scalarMultiply(double scalar, double[] vector) {

double[] result = new double[6];

for (int i = 1; i <= 5; i++) {

result[i] = scalar \* vector[i];

}

return result;

}

private static double calculateNorm(double[] vec) {

double sum = 0.0;

for (double value : vec) {

sum += value \* value;

}

return Math.sqrt(sum);

}

}

**Результаты**

Коэффициенты собственного многочлена матрицы А:

[4.7666, -8.4419, 6.8556, -2.51277, 0.3251].

Наименьшее по модулю собственное значение λ1: 0.2849.

Собственный вектор x: [-1.2506, -0.0147, -0.3534, -0.2387, 1]

Количество итераций: 10

r1= -2.6645E-15

r2 = -1.0547E-15

r3 = -2.3497E-8

r4 = [7.5807E-5, 1.8596E-5, -1.7442E-4, -1.2958E-4, 2.6550E-6]

||r4|| = 2.3089251478022145E-4

На основе результатов вычислений можно сделать следующие выводы. Значения r1 и r2, которые близки к нулю и имеют порядок величины 1e-15, подтверждают корректность найденных коэффициентов собственного многочлена матрицы А.

Значение r3​, имеющее порядок 1e-8, свидетельствует о высокой точности вычислений собственных значений и векторов.

Норма вектора r4 имеет порядок 1e-4, это может быть связано с накоплением погрешностей в результате вычислений.