БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Лабораторная работа №4

**Итерационные методы решения проблемы собственных значений**

**Выполнил:**

Крючков Василий

2 курс 9 группа

**Преподаватель:**

Горбачева Ю.Н.

Минск, 2021

**Постановка задачи**

**Задание 1**

Написать и отладить программу нахождения степенным методом (методом скалярных произведений) наибольшего по модулю собственного значения и соответствующего ему собственного вектора вещественной диагонализируемой матрицы A порядка n. Вычислительный процесс проводить снормировкой векторов итерационной последовательности (использовать евклидову норму). В качестве критерия остановки итерационного процесса использовать. Предусмотреть сообщение о выходе из итерационного процесса, если он расходится.

**Задание 2**

Написать и отладить программу нахождения итерационным методом вращений (Якоби) всех собственных значений и соответствующих им собственных векторов вещественной симметрической матрицы A порядка n. В качестве критерия остановки итерационного процесса использовать .

**Краткие теоретические сведения**

Степенной метод с нормировкой:

Матрица – симметрическая и диагонализируемая.

– любой ненулевой вектор

,

Метод вращений Якоби:

, где – матрица вращения для наибольшего по модулю элемента над диагональю матрицы .

Диагональные элементы полученной диагональной матрицы и будут искомыми собственными значениями.

Столбцы матрицы представляют собой собственные векторы матрицы A.

**Листинг программы**

**Lab.java**

import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.Formatter;  
import java.util.Random;  
  
public class Lab {  
 public static void main(String[] args) {  
 Program pr = new Program();  
 try {  
 pr.base();  
 }catch (Error e){  
 System.*out*.println(e.getMessage());  
 }  
 }  
}  
class Program {  
 static int *kIt* = 0;  
 void base() throws Error {  
 double e = 1e-7;  
 final int kMax = 5000;  
 final int n = 10;  
 double[][] mtrA = generateMatrix(n);  
 double[] initialColumnX = generateInitialColumn(n);  
 double properValue = powerMethod(mtrA, initialColumnX, kMax, e).properValue;  
 double[] properColumn = powerMethod(mtrA, initialColumnX, kMax, e).properColumn;  
 Formatter f;  
 System.*out*.println("Степенной метод");  
 System.*out*.println("Матрица А ");  
 for (int i = 0; i < mtrA.length; i++) {  
 f = new Formatter();  
 for (int j = 0; j < mtrA.length; j++) {  
 f.format("%4.0f ", mtrA[i][j]);  
 }  
 System.*out*.println(f);  
 }  
 System.*out*.println("Начальное приближение");  
 f = new Formatter();  
 for (int j = 0; j < initialColumnX.length; j++)  
 f.format("%10.0f%n", initialColumnX[j]);  
 System.*out*.print(f);  
 System.*out*.println("Номер итерации q, при которой достигнута требуемая точность = " + *kIt*);  
 System.*out*.println("Приближенное наибольшее по модулю собственное значение = " + properValue);  
 System.*out*.print("Соответствующий ему нормированный собственный вектор = ");  
 for (double i : properColumn)  
 System.*out*.print(i + " ");  
 System.*out*.println();  
 System.*out*.print("Вектор невязки = ");  
 double[] discrepancyVector = stmDiscrepancy(mtrA, properColumn, properValue);  
 for (double i : discrepancyVector)  
 System.*out*.print(i + " ");  
 System.*out*.println();  
 System.*out*.println("Норма невязки = " + secondNorm(stmDiscrepancy(mtrA, properColumn, properValue)));  
 *kIt* = 0;  
 e = 1e-7;  
 System.*out*.println();  
 ArrayList<MyResult1> lp = jacobiMethod(mtrA.clone(),kMax,e);  
 System.*out*.println("Метод вращений Якоби");  
 System.*out*.println("Номер итерации q, при которой достигнута требуемая точность = " + *kIt*);  
 for (int i=0;i<lp.size();i++)  
 System.*out*.println(lp.get(i));  
 System.*out*.println("Векторы невязок:");  
 for (int i=0;i<lp.size();i++)  
 System.*out*.println( Arrays.*toString*(stmDiscrepancy(mtrA, lp.get(i).properColumn, lp.get(i).properValue)));  
 }  
  
 //Генерация матрицы  
 double[][] generateMatrix(int n){  
 final int k = 6;  
 double[][] mtrA = new double[n][n];  
 Random rand = new Random();  
 for (int i = 0;i<n; i++)  
 for (int j = 0; j < n; j++) {  
 if (i < j) {  
 mtrA[i][j] = -100 + rand.nextInt(200+1);  
 mtrA[j][i] = mtrA[i][j];  
 }  
 }  
 for (int i = 0;i<n; i++) {  
 double dTemp = 0.;  
 for (int j = 0; j < n; j++) {  
 dTemp += Math.*abs*(mtrA[i][j]);  
 }  
 mtrA[i][i] = dTemp + k + rand.nextInt(9\*k+1);  
 }  
 return mtrA;  
 }  
  
  
 double[] generateInitialColumn(int n){  
 Random rand = new Random();  
 double[] initialColumn = new double[n];  
 for(int i = 0;i<n;i++){  
 initialColumn[i] = -100 + rand.nextInt(200+1);  
 }  
 return initialColumn;  
 }  
  
  
  
 //Степенной метод  
 MyResult1 powerMethod ( double[][] mtrA,double [] initialColumnX, int kMax, double e){  
 double [] columnX1;  
 double [] columnX2 = initialColumnX.clone();  
 double properValue;  
 int k = 0;  
 do {  
 if (k > kMax)  
 throw new Error("Итерационный процесс расходится");  
 columnX2 = divisionColumn(columnX2, secondNorm(columnX2));  
 columnX1 = columnX2;  
 columnX2 = multiplicationMatrixColumn(mtrA, columnX2);  
 properValue = scalarMultiplication(columnX2, columnX1) / scalarMultiplication(columnX1, columnX1);  
 k++;  
 *kIt*++;  
 }while (secondNorm(stmDiscrepancy(mtrA,columnX2,properValue))>e);  
 columnX2 = divisionColumn(columnX2, secondNorm(columnX2));  
 return new MyResult1(properValue,columnX2);  
 }  
 class MyResult1 {  
 public double properValue;  
 public double [] properColumn;  
 public MyResult1(double properValue,double [] properColumn){  
 this.properColumn=properColumn;  
 this.properValue=properValue;  
 }  
  
 @Override  
 public String toString() {  
 return "Собственное значение = " + properValue +  
 " соответствующий ему нормированный собственный вектор = " + Arrays.*toString*(properColumn);  
 }  
 }  
 public static void arrayCopy(double[][] aSource, double[][] aDestination) {  
 for (int i = 0; i < aSource.length; i++) {  
 System.*arraycopy*(aSource[i], 0, aDestination[i], 0, aSource[i].length);  
 }  
 }  
 ArrayList<MyResult1> jacobiMethod(double[][] mtrA, int kMax, double e) {  
 ArrayList<MyResult1> lp= new ArrayList();  
 double[][] mtr = new double[mtrA.length][mtrA.length];  
 *arrayCopy*(mtrA,mtr);  
 int k = 0;  
 int i = 0;  
 int j = 0;  
 double u = 0;  
 double cos;  
 double sin;  
 double[][] tm = new double[mtr.length][mtr.length];  
 for(int g=0;g<tm.length;g++)  
 tm[g][g]=1;  
 double[][] tTemp = new double[mtr.length][mtr.length];  
 double[][] mtrTemp = new double[mtr.length][mtr.length];  
 while (secondNormND(mtr) > e) {  
 double t = 0.;  
 for(int d =0;d<mtr.length;d++)  
 for (int b = d+1;b<mtr[0].length;b++)  
 if(Math.*abs*(mtr[d][b])>t)  
 {  
 t= Math.*abs*(mtr[d][b]);  
 i=d;  
 j=b;  
 }  
 if(mtr[i][i] == mtr[j][j]) {  
 cos = 1 / Math.*sqrt*(2);  
 sin = cos \* (-1.);  
 }  
 else  
 {  
 u = 2\*mtr[i][j]/(mtr[i][i]-mtr[j][j]);  
 double v = 1 / (Math.*sqrt*(1 + u \* u));  
 cos = Math.*sqrt*((1+v)/2);  
 sin = Math.*signum*(u)\*Math.*sqrt*((1-v)/2);  
 }  
 *arrayCopy*(mtr,mtrTemp);  
 *arrayCopy*(tm,tTemp);  
 for(int d =0;d<mtr.length;d++){  
 mtr[d][i]=mtrTemp[d][i]\*cos+mtrTemp[d][j]\*sin;  
 mtr[d][j]=mtrTemp[d][i]\*sin\*(-1.)+mtrTemp[d][j]\*cos;  
 tm[d][i]=tTemp[d][i]\*cos+tTemp[d][j]\*sin;  
 tm[d][j]=tTemp[d][i]\*sin\*(-1.)+tTemp[d][j]\*cos;  
 }  
 *arrayCopy*(mtr,mtrTemp);  
 for(int d =0;d<mtr.length;d++){  
 mtr[i][d]=mtrTemp[i][d]\*cos+mtrTemp[j][d]\*sin;  
 mtr[j][d]=mtrTemp[i][d]\*sin\*(-1.)+mtrTemp[j][d]\*cos;  
 }  
 k++;  
 *kIt*++;  
 if (k > kMax)  
 throw new Error("Привышен параметр k max");  
 }  
 for(int d=0;d<mtrA.length;d++) {  
 double[] properColumn = new double[mtrA.length];  
 for(int g=0;g<mtrA.length;g++)  
 properColumn[g]=tm[g][d];  
 lp.add(new MyResult1(mtr[d][d],properColumn));  
 }  
 return lp;  
 }  
 //Подсчет второй нормы  
 double secondNorm(double []columnX){  
 double dTemp =0.;  
 for (double x : columnX) dTemp += x \* x;  
 return Math.*sqrt*(dTemp);  
 }  
  
 double secondNormND(double [][] mtrA){  
 double dTemp =0.;  
 for(int i =0;i<mtrA.length;i++)  
 for (int j = i+1;j<mtrA[0].length;j++)  
 dTemp += mtrA[i][j] \* mtrA[i][j];  
 return dTemp;  
 }  
 //Подсчет нормы  
  
 //Подсчет скалярного произведение векторов  
 double scalarMultiplication(double []columnX1,double []columnX2){  
 double temp =0.;  
 for (int i = 0;i<columnX1.length;i++){  
 temp += columnX1[i]\*columnX2[i];  
 }  
 return temp;  
 }  
  
 double[] divisionColumn(double []columnX1,double x){  
 double[] columnX = columnX1.clone();  
 for (int i = 0;i<columnX1.length;i++){  
 columnX[i] = columnX[i]/x;  
 }  
 return columnX;  
 }  
  
  
 double[] stmDiscrepancy(double[][] mtrA,double [] columnX, double properValue){  
 double[] r = multiplicationMatrixColumn(mtrA,columnX);  
 for(int i=0;i<r.length;i++)  
 r[i]= r[i] - columnX[i]\*properValue;  
 return r;  
 }  
  
  
 double[] multiplicationMatrixColumn(double[][] mtrA,double[] columnF){  
 double [] columnX = new double[columnF.length];  
 for(int i = 0; i < mtrA.length; i++)  
 for(int j = 0; j < mtrA.length; j++)  
 columnX[i] += columnF[j] \* mtrA[i][j];  
 return columnX;  
 }  
 double[][] multiplicationMatrix(double[][] mtrA1,double[][] mtrA2){  
 double [][] newMtr = new double[mtrA1.length][mtrA2.length];  
 for (int i=0; i< mtrA1.length; ++i)  
 for (int j=0; j<mtrA2.length; ++j)  
 for (int k=0; k<mtrA1[0].length; ++k)  
 newMtr[i][j] += mtrA1[i][k] \* mtrA2[k][j];  
 return newMtr;  
 }  
}

**Результаты**

Степенной метод

Матрица А

299 -6 10 0 -55 -58 -56 -39 -33 24

-6 345 -11 36 -25 48 15 83 64 -34

10 -11 357 58 -81 -61 -12 24 67 2

0 36 58 505 -79 -6 -65 -95 -78 49

-55 -25 -81 -79 558 49 67 16 76 85

-58 48 -61 -6 49 386 5 16 89 20

-56 15 -12 -65 67 5 369 24 63 -29

-39 83 24 -95 16 16 24 423 33 -64

-33 64 67 -78 76 89 63 33 536 1

24 -34 2 49 85 20 -29 -64 1 321

Начальное приближение

59

60

48

71

74

-90

-10

25

62

13

Номер итерации q, при которой достигнута требуемая точность = 192

Приближенное наибольшее по модулю собственное значение = 790.5533258329178

Соответствующий ему нормированный собственный вектор = -0.1784998051824704 0.09358099076023654 -0.11888171626597213 -0.4500040657908341 0.5384060853612427 0.24993567067862796 0.2778058163344959 0.24703648851920493 0.5021620164567591 -0.005015453602766897

Вектор невязки = 1.0800249583553523E-12 -3.127809122815961E-11 -3.0880187296133954E-11 1.0516032489249483E-11 6.048139766789973E-11 9.947598300641403E-13 -4.291678123991005E-12 -4.132516551180743E-11 -3.427658157306723E-11 3.518918489930911E-11

Норма невязки = 9.920684425199838E-11

Метод вращений Якоби

Номер итерации q, при которой достигнута требуемая точность = 117

Собственное значение = 216.17146076675695 соответствующий ему нормированный собственный вектор = [0.6167703622785277, -0.2148597999068837, 0.2548736500497738, 0.14786477020334454, 0.22783091718788775, 0.4092110559870543, 0.16725892506950754, 0.04396823049213069, -0.11471237785572772, -0.47347304118794326]

Собственное значение = 305.3335091631321 соответствующий ему нормированный собственный вектор = [0.5322481008127763, 0.6008919524164991, -0.3433430942309842, 0.03193126953513288, 0.08704244905349674, -0.38078684999339457, 0.1787364824553291, -0.1945231059980372, 0.1173131144969204, -0.024766621617232303]

Собственное значение = 207.4520529391038 соответствующий ему нормированный собственный вектор = [-0.17517616765983407, 0.49319223401736967, 0.5239995018695544, -0.3422513312825579, 0.15841242758504198, 0.14029561263661883, -0.10511123981724277, -0.4166287067766232, -0.314184135294081, -0.0790341686203236]

Собственное значение = 532.5162703445793 соответствующий ему нормированный собственный вектор = [-0.06557635324298508, 0.18164811294625965, 0.2313445855374979, 0.6194272259767173, 0.05214213550225114, 0.30408966327619025, -0.07655071095877167, -0.281115883417171, 0.5286478092077458, 0.25680598975815394]

Собственное значение = 589.2575527705754 соответствующий ему нормированный собственный вектор = [0.010710407872350234, -0.3148934581694487, -0.3110472877790152, 0.10565138538541245, 0.6107632346060952, 0.010358069835278098, -0.04295547015405348, -0.4163527516256497, -0.34434036204034607, 0.3548389454485796]

Собственное значение = 387.7946929134416 соответствующий ему нормированный собственный вектор = [0.15226731099552138, -0.08823963957309365, -0.40887620842233824, -0.367799540444159, -0.39990024281808817, 0.48784786593958696, -0.22820392498983116, -0.432661963691557, 0.1689292458913017, -0.029084968868193367]

Собственное значение = 362.8001213772097 соответствующий ему нормированный собственный вектор = [-0.35210991097041083, -0.09432142924202112, -0.10196944779412515, 0.13406309576112266, -0.17853970165050134, -0.022978837570590077, 0.7580350697825357, -0.3804955877989662, -0.04381239856676448, -0.2916047993247779]

Собственное значение = 446.8414662352248 соответствующий ему нормированный собственный вектор = [-0.23820752292014877, 0.434286167202368, -0.4173944600354169, 0.3038604369108309, 0.029131398605399747, 0.4564948274339198, -0.047520398344185845, 0.33273405387955063, -0.3891590309346758, -0.12021790435067282]

Собственное значение = 790.5533258329101 соответствующий ему нормированный собственный вектор = [-0.17849977942464365, 0.09358097557710653, -0.1188817672007527, -0.4500041107801539, 0.5384060364869965, 0.24993573908668257, 0.277805792637101, 0.24703643026398464, 0.5021620360832691, -0.0050154642397868794]

Собственное значение = 260.2795476570646 соответствующий ему нормированный собственный вектор = [0.2557985537621018, 0.05767999008003493, 0.16241982564995508, -0.13055208173049815, -0.2390508924888463, 0.256725624467462, 0.46389192732922746, 0.17481449392322118, -0.20557276893137888, 0.6904684688806305]

Векторы невязок:

[3.64448874847767E-5, -7.877665463240646E-5, 7.824394758415565E-5, -4.9877368091699736E-5, -1.5620350914957726E-5, -7.800305752425629E-5, 8.694954217958184E-6, -5.6264642548597976E-5, 6.562023143530382E-5, 1.6781872673732323E-5]

[-5.185947856034545E-5, -9.86483306064656E-6, -3.7096496868116446E-5, 1.93025901467081E-5, 3.898367625509991E-5, -4.6468251113651604E-5, -9.159295346705676E-5, -3.8292196457234695E-5, 4.149455325830331E-5, -1.269176752494161E-4]

[-5.468922523732545E-7, -1.6138288145839397E-6, -2.1916519301612425E-6, 1.0984800695723607E-6, 4.197549586137939E-6, -1.0207491278890757E-6, -1.4801539975906053E-6, -2.738520919365328E-6, -1.2544002458980685E-6, -1.5346448023478843E-7]

[1.0004456072465473E-5, -1.2418936705671513E-6, 7.604255259252568E-6, -6.148966122054844E-6, -5.230341983519793E-6, 3.287093676362929E-6, 7.979189923901231E-6, 3.89937619615921E-6, -2.0830413518524438E-6, 1.9519062931294684E-5]

[-4.580633240713894E-6, 1.8020104164406803E-6, 3.3004016870563646E-6, -4.786994423966462E-6, -1.406458181918424E-6, -5.526368957120553E-6, -1.7736370985232952E-6, -4.747496518575645E-6, 1.4460140675964794E-6, 4.255989608736854E-6]

[1.5779262824366924E-5, 1.701873742376847E-6, 2.4316880740116176E-6, 2.3810785677369495E-5, -2.679937165339652E-5, -1.732379894292535E-5, -1.4166962984063503E-5, -1.4277602360834862E-5, -2.4672561750094246E-5, 3.0082752466853435E-7]

[3.3106596220022766E-6, -2.0059942897887595E-6, 1.018384431006325E-6, -7.976012312838066E-7, 4.4509880581244943E-7, -1.164671738607126E-6, 9.143400347966235E-7, 1.9057135318689689E-6, -4.66458820369553E-6, -3.6612326539398055E-6]

[-1.0653885692590848E-4, 3.79097718621324E-5, -4.400814918881224E-5, -3.0763804602429445E-5, -4.3620612563799455E-5, -7.32722810141695E-5, -2.8189896244867896E-5, -3.0726079671694606E-6, 1.817006224769102E-5, 7.810770332383754E-5]

[-1.1663133932415803E-5, 6.4835375326310896E-6, 1.986399209386036E-5, 1.78191589270682E-5, 1.9428358200457296E-5, -2.8431879314894104E-5, 9.069335590083938E-6, 2.3270903682259814E-5, -7.773336733407632E-6, 5.471786513222554E-6]

[-1.016714413566433E-4, -1.1045762805217407E-4, 6.925734292195784E-5, 1.0547686002837509E-5, -1.5425810978797472E-5, 7.891763148393238E-5, -3.543779774872746E-5, 3.127166850447338E-5, -8.116110748801475E-6, 1.1388272810108901E-5]

**Выводы**

Степенной метод является эффективным решениям для решения задачи нахождения наибольшего собственного значения и соответствующего ему собственного значения с заданной точностью. А метод вращений Якоби является продуктивным методом нахождения собственных значений и соответствующих им собственных векторов. Однако для достаточно низких точностей эти и подобные им алгоритмы решения проблемы собственных значений обладает довольно большой сложностью.