**Белорусский государственный университет**

**Факультет прикладной математики и информатики**

Лабораторная работа №5,6,7,8

Вариант 8

Проблема собственных значений

**Выполнил:**

Студент 2 курса 7 группы ПМ ФПМИ

Шевцов Евгений

Преподаватель:

Будник Анатолий Михайлович

**Минск – 2021**

***Для всех методов B = ATA***

**Описание метода Крылова нахождения собственного многочлена**

Построим систему векторов c(n+1) следующим образом: возьмём вектор c(0) = (1, 0, …, 0)T, остальные векторы системы найдём по формуле c(k + 1) = Bc(k), k = . Получив   
n + 1 векторов с учётом, что матрица имеет размерность n, можем разложить последний вектор по базису из первых n векторов. Таким образом, получим систему:

Решив которую методом Гаусса (что было и сделано в программе) мы получим q1, …, qn, которые совпадают с коэффициентом характеристического многочлена λn – p1λn-1 - … - pn, корни которого являются собственными значениями.

Для проверки найденных коэффициентов вычислим корни многочлена с помощью программы Wolfram Alpha. Подставив найденные собственные значения в многочлен, мы получим невязку:

**Описание метода Данилевского нахождения системы собственных векторов**

В методе Данилевского исходная матрица подобными преобразованиями приводится к нормальной форме Фробениуса. Таким образом, преобразование примет вид Ф = S-1BS.

Для нахождения матрицы S рассмотрим матрицы, с помощью которых B приводится к НФФ. Для этого на каждом шаге k матрица B умножается справа на Mn-k =

а слева на на матрицу Mn-k-1, равную:

Таким образом S = Mn-1\*…\*M1 (в программе произведение происходило параллельно преобразованию матрицы B).

В итоге первая строка матрицы B будет состоять из коэффициентов характеристического многочлена матрицы. Используя найденные собственные значения методом Крылова (коэффициенты получатся аналогичными, т.к. оба методы являются точными, значит, можно взять и их) строим вектор yi = (λin-1, λin-2, …, 1) и находим собственный вектор для i-го собственного значения по формуле:

В матричном виде: x = Sy.

i-й компонент невязки же считаем по формуле:

ri = Bxi – λixi, i =

**Описание итерационного метода вращений (Якоби) нахождение спектра и системы собственных векторов**

Для данного метода используется преобразования вращения вида TklTATkl, последовательность которых стремится к диагональной матрице, состоящей из собственных значений. Последовательное же произведение матриц Tkl даст нам матрицу, столбцы которых образуют систему собственных векторов.

Распишем одну итерацию.

Вначале находится максимальный по модулю недиагональный элемент bkl. В программе это делается в самой итерации. Далее идёт подсчёт tg2φ = . Затем находятся элементы матрицы вращения:

Матрица вращения при этом принимает вид:

Далее происходит ранее упомянутое преобразование подобия и последовательное произведение Tkl до тех пор, пока не выполнится условие:

По условию = 10-5.

**Описание степенного метода нахождения минимального собственного значения**

Для нахождения минимального собственного значения будем использовать вариацию степенного метода – метод обратных итераций. Для него нам требуется найти обратную матрицу для матрицы B. В программе это сделано методом Гаусса, реализованного в первой лабораторной.

Далее берётся произвольный вектор y(0) (в программе взят вектор (1, …, 1)T) в качестве начального. На k-й итерации формула выглядит следующим образом:

y(k+1) = B-1y(k)

Приближение собственных значений вычисляется как среднее арифметическое отношения координат y(k+1) и y(k), т.е. по формуле:

Данный процесс выполняем, пока |λk – λk+1| > ε. В конце получим максимальное собственное значение B-1. Возведя его в -1 степень, получим минимальное собственное значение B.

Собственным вектором же будет являться y(k+1).

**Листинг**

**Общие функции:**

#include <iostream>

#include <vector>

#include <iomanip>

std::vector<std::vector<double>> transpositionMatrix(std::vector<std::vector<double>> M) {

for (int i = 0; i < 4; ++i) {

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

std::swap(M[i][j], M[j][i]);

}

}

return M;

}

std::vector<std::vector<double>> multMatrix(const std::vector<std::vector<double>> M1, const std::vector<std::vector<double>> M2) {

std::vector<std::vector<double>> M3 = {

{0.0, 0.0, 0.0, 0.0, 0.0},

{0.0, 0.0, 0.0, 0.0, 0.0},

{0.0, 0.0, 0.0, 0.0, 0.0},

{0.0, 0.0, 0.0, 0.0, 0.0},

{0.0, 0.0, 0.0, 0.0, 0.0} };

for (int i = 0; i < 5; ++i) {

for (int j = 0; j < 5; ++j) {

for (int k = 0; k < 5; ++k) {

M3[i][j] += M1[i][k] \* M2[k][j];

}

}

}

return M3;

}

std::vector<double> multVec(const std::vector<std::vector<double>> M, const std::vector<double> f) {

std::vector<double> b = { 0., 0., 0., 0., 0. };

for (int i = 0; i < 5; ++i) {

for (int j = 0; j < 5; ++j) {

b[i] += M[i][j] \* f[j];

}

}

return b;

}

double getLambda(const std::vector<double> f1, const std::vector<double> f2) {

double res = 0.;

for (int i = 0; i < 5; ++i) {

res += f1[i]/ f2[i];

}

return res / 5.;

}

std::vector<double> normV(std::vector<double> f) {

double max = 0;

for (int i = 0; i < 5; ++i) {

if (abs(f[i]) > abs(max)) {

max = f[i];

}

}

for (int i = 0; i < 5; ++i) {

f[i] /= max;

}

return f;

}

//Для Якоби

double getEps(std::vector<std::vector<double>> M) {

double eps = 0.;

for (int i = 0; i < 5; ++i) {

for (int j = 0; j < 5; ++j) {

if (i == j) {

continue;

}

eps += pow(M[i][j], 2);

}

}

return sqrt(eps);

}

//Для степенного

double getEps(double lk, double lk1) {

return abs(lk - lk1);

}

**Метод Крылова**

std::vector<std::vector<double>> A\_Result = multMatrix(A, transpositionMatrix(A));

std::vector<std::vector<double>> C;

C.push\_back({1., 0., 0., 0., 0.});

for (int i = 0; i < 5; ++i) {

std::vector<double> Ck = multVec(A\_Result, C[i]);

C.push\_back(Ck);

}

std::vector<std::vector<double>> C\_result = {

{C[4][0], C[3][0], C[2][0], C[1][0], C[0][0], C[5][0]},

{C[4][1], C[3][1], C[2][1], C[1][1], C[0][1], C[5][1]},

{C[4][2], C[3][2], C[2][2], C[1][2], C[0][2], C[5][2]},

{C[4][3], C[3][3], C[2][3], C[1][3], C[0][3], C[5][3]},

{C[4][4], C[3][4], C[2][4], C[1][4], C[0][4], C[5][4]} };

for (int i = 0; i < 4; ++i) {

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

for (int k = i + 1; k < 6; ++k) {

C\_result[j][k] -= C\_result[j][i] / C\_result[i][i] \* C\_result[i][k];

}

}

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

C\_result[j][i] = 0.;

}

}

std::vector<double> answer = { 0., 0., 0., 0., 0. };

for (int k = 4; k >= 0; --k) {

double sum = 0;

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

sum += C\_result[k][j] \* answer[j];

}

answer[k] = (C\_result[k][5] - sum) / C\_result[k][k];

}

**Метод Данилевского**

std::vector<std::vector<double>> F = multMatrix(A, transpositionMatrix(A));

std::vector<std::vector<double>> S = {

{1.0, 0.0, 0.0, 0.0, 0.0},

{0.0, 1.0, 0.0, 0.0, 0.0},

{0.0, 0.0, 1.0, 0.0, 0.0},

{0.0, 0.0, 0.0, 1.0, 0.0},

{0.0, 0.0, 0.0, 0.0, 1.0} };

for (int i = 3; i >= 0; --i) {

std::vector<std::vector<double>> M = {

{1.0, 0.0, 0.0, 0.0, 0.0},

{0.0, 1.0, 0.0, 0.0, 0.0},

{0.0, 0.0, 1.0, 0.0, 0.0},

{0.0, 0.0, 0.0, 1.0, 0.0},

{0.0, 0.0, 0.0, 0.0, 1.0} };

std::vector<std::vector<double>> M1 = {

{1.0, 0.0, 0.0, 0.0, 0.0},

{0.0, 1.0, 0.0, 0.0, 0.0},

{0.0, 0.0, 1.0, 0.0, 0.0},

{0.0, 0.0, 0.0, 1.0, 0.0},

{0.0, 0.0, 0.0, 0.0, 1.0} };

for (int j = 0; j < 5; ++j) {

M1[i][j] = F[i + 1][j];

if (i != j) {

M[i][j] = -F[i + 1][j] / F[i + 1][i];

}

}

M[i][i] = 1 / F[i + 1][i];

F = multMatrix(M1, multMatrix(F, M));

S = multMatrix(S, M);

}

std::vector<double> lambda = { 0.23075, 0.31026, 0.691624, 0.936767, 1.17864 };

**Метод Якоби (Итерационный метод вращений)**

std::vector<std::vector<double>> M = multMatrix(A, transpositionMatrix(A));

const double EPS = 0.00001;

int counter = 0;

std::vector<std::vector<double>> resultT = {

{1.0, 0.0, 0.0, 0.0, 0.0},

{0.0, 1.0, 0.0, 0.0, 0.0},

{0.0, 0.0, 1.0, 0.0, 0.0},

{0.0, 0.0, 0.0, 1.0, 0.0},

{0.0, 0.0, 0.0, 0.0, 1.0} };

while (getEps(M) > EPS) {

counter++;

double max = 0;

int maxI = 0;

int maxJ = 0;

for (int i = 0; i < 5; ++i) {

for (int j = 0; j < 5; ++j) {

if (i == j) {

continue;

}

else {

if (abs(M[i][j]) > abs(max)) {

max = M[i][j];

maxI = i;

maxJ = j;

}

}

}

}

double tg = 2 \* max / (M[maxI][maxI] - M[maxJ][maxJ]);

double cos = sqrt((1 + 1 / sqrt(1 + pow(tg, 2))) / 2);

double sin = sqrt(1 - pow(cos, 2));

std::vector<std::vector<double>> T = {

{1.0, 0.0, 0.0, 0.0, 0.0},

{0.0, 1.0, 0.0, 0.0, 0.0},

{0.0, 0.0, 1.0, 0.0, 0.0},

{0.0, 0.0, 0.0, 1.0, 0.0},

{0.0, 0.0, 0.0, 0.0, 1.0} };

T[maxI][maxI] = cos;

T[maxJ][maxJ] = cos;

T[maxI][maxJ] = -sin;

T[maxJ][maxI] = sin;

M = multMatrix(transpositionMatrix(T), multMatrix(M, T));

resultT = multMatrix(resultT, T);

}

**Степенной метод**

const double EPS = 0.00001;

std::vector<std::vector<double>> A1 = {

{1.67953, 0.291252, 0.513653, 0.32638, -0.737916},

{0.291252, 3.87842, 0.500924, 0.384012, -0.280008 },

{0.513653, 0.500924, 1.5147, 0.051476, -0.903941},

{-0.32638, 0.384012, 0.051476, 1.24609, 0.169331},

{-0.737916, -0.280008, -0.903941, 0.169331, 2.59987} };

std::vector<double> y = {1, 1, 1, 1, 1};

double l = 1;

double l1 = 0;

while (getEps(l, l1) > EPS) {

std::vector<double> y1 = multVec(A1, y);

l = l1;

l1 = getLambda(y1, y);

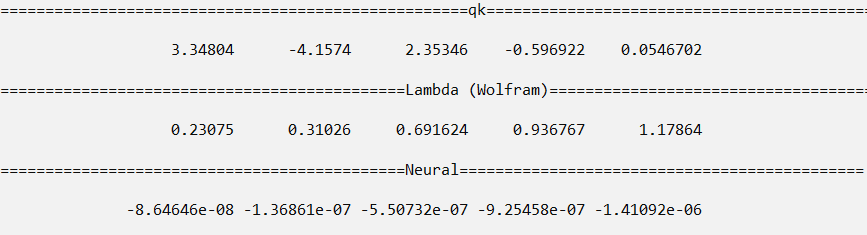
y = normV(y1);

}

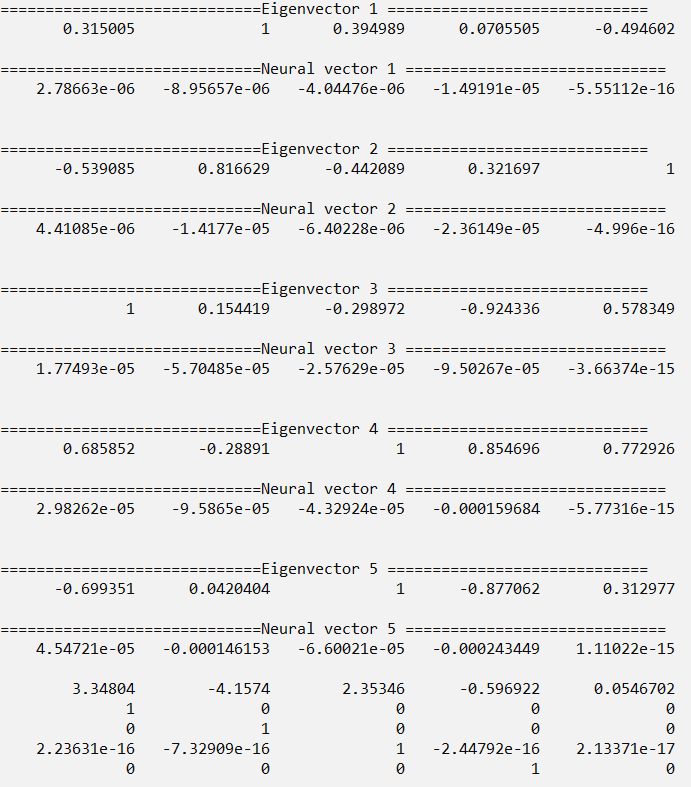
l1 = 1 / l1;

**Полученные значения**

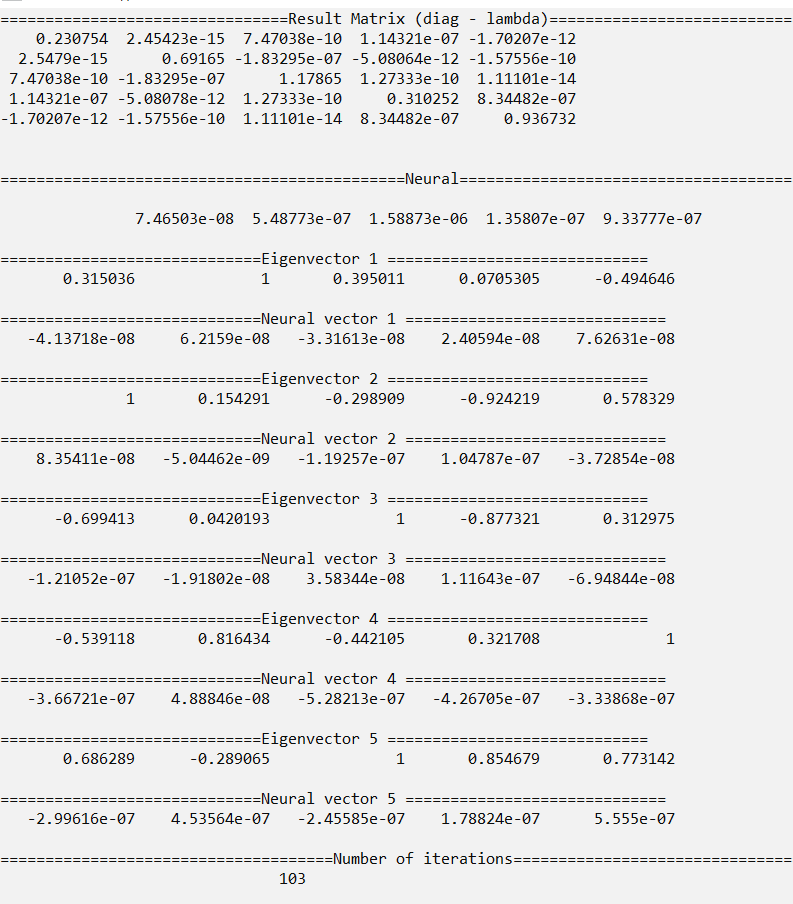
**Метод Крылова**



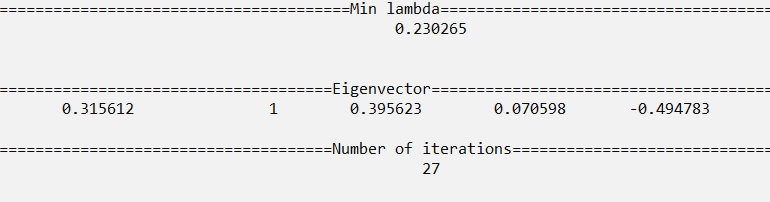
**Метод Данилевского**



**Метод Якоби (Итерационный метод вращений)**



**Степенной метод**



**Вывод**

В точных методах Крылова и Данилевского были найдены коэффициенты характеристического многочлена. Однако ни тот, ни другой метод не находит его корни, следовательно, они искались «за пределами» данных методов, а именно: с помощью WolframAlpha, что привело к погрешностям вычислений, т.к. данная программа выдаёт значения с определённой точностью. Таким образом, несмотря на то, что эти методы являются точными, мы имеем погрешность корней порядка 10-6 (и там, и там будет погрешность одна и та же, т.к. коэффициенты по итогу многочленов получились одинаковыми). Аналогичная причина погрешности получена в нахождении собственных векторов (вплоть до 10-4). Также следует заметить, что нормальная форма Фробениуса была найдена с небольшой (порядка 10-16) погрешностью, связанной с округлениями в вычислениях.

Если бы точные значения вычислялись без погрешности программы, то их погрешность была бы связана на прямую с погрешностью найденных коэффициентов. В случае метода Крылова (напомню: там решается система методом Гаусса) систему можно было бы решить с помощью выбора главного элемента по матрице, а т.ж. ортогональными методами, не влияющие на обусловленность системы, с целью нахождения более точных коэффициентов. В случае метода Данилевского преобразования подобия сильно ухудшают число обусловленности матрицы, что также может сказаться на дальнейшем поиске собственных значений и векторов этим методом.

В методе Якоби, который является итерационным, собственные значения и вектора найдены с погрешностью порядка 10-6. В случае с собственными векторами у нас получились более точные значения в связи с использованием не точных решений из WolframAlpha и возможных погрешностях при вычислениях. При этом в методе Якоби не накапливается погрешность, т.к. каждый новый процесс нивелирует погрешность старого.

В степенном методе изначально находилась обратная матрица B, что сопровождалось накоплением ошибок, т.к. находилась она методом Гаусса. Метод так же является итерационным, но используется для нахождения максимального (с помощью B-1 – минимального) собственного значения и соответствующего ему вектора. Собственное значение совпадает с точностью до 10-4. Аналогичная ситуация и с собственным вектором, соответствующего этому собственному значению.

Погрешность метода Якоби и степенного связана с различным условием выхода из итерационного процесса (в случае Якоби – сумма диагональных элементов < ε, в случае степенного – модуль разность искомого собственного значения на соседних шагах < ε). Несмотря на это, из-за округлений в вычислениях собственных значений у точных методов погрешность получилась, вообще говоря, меньше, чем в точных.

Для сравнения собственных векторов они нормировались по кубической норме. Значения их координат во всех методах совпадали с точность 10-4.