Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

**ВЫЧИСЛЕНИЕ СОБСВЕННЫХ ЧИСЕЛ И СОБСТВЕННЫХ ВЕКТОРОВ**

**Отчет по лабораторной работе №3**

**По дисциплине**

**«Численные методы»**

Студент гр. 430-2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.А. Лузинсан

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_ 2022 г.

Проверил: ст. преп. каф. АСУ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.Е. Косова

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_ 2022 г.

Томск 2022

Оглавление

[1 Цели и задачи 3](#_Toc100062076)

[1.1 Формат входных данных 3](#_Toc100062077)

[1.2 Формат выходных данных 3](#_Toc100062078)

[2 Теория 3](#_Toc100062079)

[2.1 Метод Данилевского 4](#_Toc100062080)

[2.1.1 Вычисление собственных чисел 4](#_Toc100062081)

[2.1.2 Вычисление собственных векторов 5](#_Toc100062082)

[2.2 Определение кратности собственных чисел и векторов 5](#_Toc100062083)

[3 Ход выполнения лабораторной работы 8](#_Toc100062084)

[4 Тестирование 9](#_Toc100062085)

[4.1 Unit 1 9](#_Toc100062086)

[4.2 Unit 2 9](#_Toc100062087)

[Вывод 10](#_Toc100062088)

[Список использованных источников 11](#_Toc100062089)

[**Листинг программы** 12](#_Toc100062090)

# Цели и задачи

В ходе данной лабораторной работы необходимо реализовать поиск собственных чисел и собственных при помощи метода Данилевского.

## 1.1 Формат входных данных

|  |  |  |
| --- | --- | --- |
| m | – тип задачи (1 – поиск собственных чисел, 2 – векторов); | |
| n | – порядок матрицы; | |
| a11…a1n [b1]  a21…a2n [b2]  …………..  an1…ann [bn] | | – коэффициенты матрицы. |

## 1.2 Формат выходных данных

P – матрица Фробениуса;

λi – i-е собственное число;

|A-λiE| – проверка i-го собственного числа (при m = 1);

xi – i-й собственный вектор (при m = 2);

Axi-λixi – проверка i-го собственного вектора (при m = 2);

ki – кратность i-го собственного числа/вектора;

… И т.д. для всех i = 1, 2, …, m.

# Теория

Собственные числа и вектора квадратной матрицы являются её важными характеристиками, использующимися в различных формах математического анализа. Собственное число матрицы *λi* и соответствующий ему собственный вектор *xi* удовлетворяют соотношению 2.1:

|  |  |
| --- | --- |
| *Axi = λixi* | (2.1) |

У квадратной матрицы размерности *n* имеется *n* собственных чисел и векторов. Некоторые из них могут быть кратными (т.е. совпадающими). Таким образом, квадратная матрица размерности *n* имеет *m* различных собственных чисел *λi* и соответствующих им собственных векторов *xi* кратности *ki*. При этом используется формула 2.2:

|  |  |
| --- | --- |
|  | (2.2) |

Отметим также, что от умножения собственного вектора матрицы на скаляр *c* он не перестаёт быть её собственным вектором (2.3):

|  |  |
| --- | --- |
| *A(cxi) = λi(cxi) ⇒ cAxi = cλixi ⇒ Axi = λixi* | (2.3) |

При аналитическом решении собственные числа матрицы находятся из решения уравнения:

|  |  |
| --- | --- |
| *D(λ) = 0* | (2.4) |

где *D(λ) = det(A – λE)* – характеристический полином матрицы. После этого, согласно (2.1), можно найти собственные вектора, решая СЛАУ (2.5):

|  |  |
| --- | --- |
| *(A – λE)x = 0* | (2.5) |

## Метод Данилевского

### **Вычисление собственных чисел**

В данной практической работе для поиска собственных чисел и векторов мы будем использовать метод Данилевского. Суть его состоит в том, что исходная матрица A преобразуется в подобную ей матрицу Фробениуса P, имеющую следующий вид:

*P =*

Делается это при помощи следующего преобразования подобия:

|  |  |
| --- | --- |
| P = S-1AS, | (2.6) |

Где

Таким образом, можно последовательно находить n–1 матрицу A(k):

|  |  |
| --- | --- |
|  | (2.7) |

А можно найти матрицы S (прямую и обратную) и затем сразу вычислить *P* по формуле (2.6). Такой способ эффективнее, т.к. не нужно хранить множество матриц *M*, произведение которых еще понадобятся для вычисления собственных векторов.

Матрицы *M* строятся следующим образом:

|  |  |
| --- | --- |
|  | (2.8) |
|  | (2.9) |

Несложно доказать, что у подобных матриц собственные числа совпадают. Далее для матрицы *P* строится характеристический полином:

|  |  |
| --- | --- |
| *D(λ) = det(P – λE) = (–1)n [λn – p1λn–1 – p2λn–2 – … – pn]* | (2.10) |

Это полином степени *n*. Очевидно, что он имеет n корней *λ1, λ2, …, λn*. Некоторые из них могут быть кратными, при этом выполняется соотношение (2.2). Необходимо не только найти все корни полинома, но и определить их кратность (см. п. 2.1.2).

### Вычисление собственных векторов

Далее для каждого собственного числа вычисляется соответствующий ему собственный вектор. Собственные вектора *у* подобных матриц не совпадают. Если *yi* – это собственный вектор матрицы *P*, соответствующий собственному числу *λi*, то:

|  |  |
| --- | --- |
|  | (2.11) |

При этом собственный вектор матрицы *P* выглядит следующим образом:

|  |  |
| --- | --- |
|  | (2.12) |

## Определение кратности собственных чисел и векторов

При поиске кратных корней возникают некоторые сложности. Дело в том, что если кратность корня четная, то в этой точке наблюдается экстремум (минимум или максимум) характеристического полинома, а если нечетная – то полином просто меняет знак. Пример приведен на рис. 2.2.1.

Согласно определению, корень уравнения ξ имеет кратность k, если не только функция в точке ξ принимает нулевое значение, но и k–1 её производных:

|  |  |
| --- | --- |
|  | (2.13) |

При i = 0 имеем саму функцию. Таким образом, получаем k нулей функции и ее производных.

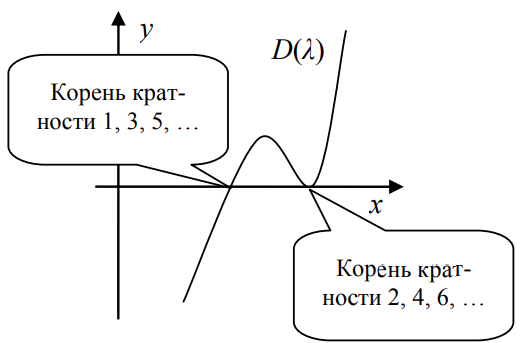


Рисунок 2.2.1 – Поведение характеристического полинома

Учитывая погрешности вычислений на ЭВМ, при четной кратности корня характеристический полином может пройти либо выше, либо ниже нулевой отметки (рис. 2.2.2).

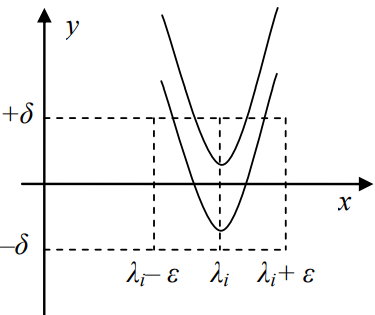


Рисунок 2.2.2 – Погрешности при вычислении собственных чисел

Здесь ε и δ – достаточно малые числа. Т.о., программа может либо вообще не найти корня, либо найти сразу два. Поэтому договоримся считать корнем любое число λi , для которого | f(λi)| < δ. При этом, если два корня λi1 и λi2 расположены близко друг к другу (т.е. |λi1 – λi2| < 2ε), то корнем следует считать только один из них, либо за корень принять число, расположенное ближе всего в оси абсцисс.

Поиск собственных чисел продолжается до тех пор, пока не будут найдены все, т.е. пока не выполнится условие (2.2).

# Ход выполнения лабораторной работы

В качестве задания исходной матрицы, матрицы Фробениуса, вспомогательной матрицы S использовался собственный класс Matrix, в котором уже определены необходимые операции. Для превращения исходной матрицы в полином использовался класс Polynomial, который позволяет преобразовать полином в строчный вид для дальнейших вычислений. Элементами класса Polynom выступают экземпляры класса Fraction, который имеет метод toFraction(), позволяя представлять в качестве элементов полинома не только целые числа, но и вещественные, преобразованный в дробный вид, так как предоставленная библиотека PolStr умеет находить польскую строку только для выражений с целочисленными элементами.

В качестве метода для определения значения определителя матрицы был взят метод из предыдущей лабораторной работы, а именно метод Гаусса.

Данные считываются из файла. Далее, в зависимости от выбранного типа задачи (задаётся в файле), программа высчитывает собственные числа или соответствующие собственные вектора. И в том, и в другом случае выполняется подсчёт собственных чисел.

Для подсчёта собственных чисел получаем матрицу Фробениуса. Таким образом:

1. сначала инициализируем пока не построенную матрицу Фробениуса исходной матрицей;
2. запоминаем промежуточную исходную матрицу, состояние которой понадобится при вычислении матрицы Фробениуса;
3. инициализируем промежуточные матрицы M и M-1;
4. инициализируем вспомогательную единичную матрицу.

Запускаем цикл, в котором:

1. сначала получаем матрицу M,
2. перемножаем матрицу M справа на матрицу Фробениуса, результат записываем в матрицу Фробениуса,
3. а также перемножаем матрицу M с матрицей S, которая понадобится при вычислении собственных чисел,
4. вычисляем матрицу M-1;
5. и наконец, домножаем только что полученную матрицу M-1 слева на матрицу Фробениуса, записываем результат произведения в матрицу Фробениуса.

Вычисленная матрица Фробениуса нужна для получения собственных чисел методом Данилевского. Инициализируем полином, коэффициенты которого составлены из нулевой строки матрицы Фробениуса. Далее следует вычислить корни полинома. Это реализовано простым перебором корней с шагом eps в промежутке, заданном в программе препроцессорными константами со значениями -10 и 10 соответственно. Значение считается корнем, если оно входит в допустимый промежуток по оси ординат, заданный также .

Если некоторый корень удовлетворяет всем условиям, то это ещё не значит, что он является фактическим корнем, т.к., как показано на рис. 2.2.2., нужно удостоверится, не будут ли последующие корни ближе к оси абсцисс, чем текущая. В программе реализован такой подход, что, если последующий корень ближе по модулю к оси абсцисс, чем предыдущий, то на текущем промежутке (удовлетворяющем условиям погрешности), следует считать его корнем. Корень установится, как только последующий корень по модулю будет больше предыдущего, боже если он всё ещё входит в промежуток заданной погрешности.

После вычисления текущего корня нужно проверить его кратность. Для этого вычислим производную функции в данном корне несколько раз, до тех пор, пока производная не станет равной нулю. Количество раз, которое мы вычислим производную, не встретив нулевую, и будет считаться кратностью корня. При этом стоит заметить, что каждая последующая вычисленная производная в этой точке считается нулевой по несколько большему промежутку по оси ординат, по причине увеличивающейся погрешности вычислений в ЭВМ. В ходе экспериментов было выяснено, что коэффициентом корректировки можно считать значение 10.

Вычисление собственных векторов исходит из значений собственных чисел непосредственно. Причём между собственными векторами матрицы Фробениуса и собственными векторами исходной матрицы существует прямая связь. Найдя собственные вектор по простой формуле 2.12, получаем собственный вектор исходной матрицы, умножив вычисленную на этапе получения матрицы Фробениуса матрицу M на текущий собственный вектор матрицы Фробениуса.

# Тестирование

## Unit 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| А | Собственные числа () | Кратность () |  | Собственные вектора () |  |
|  | 3 | 1 | 3.41061e-12 |  |  |
| -1 | 2 | 6.31089e-30 |  |  |

## Unit 2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| А | Собственные числа () | Кратность () |  | Собственные вектора () |  |
|  | 1.7343 | 1 | -0.0197881 |  |  |
| 30.9817 | 1 | -0.0517757 |  |  |
| -13.716 | 1 | -0.00289032 |  |  |

# Вывод

В ходе данной лабораторной работы я реализовала поиск собственных чисел и собственных векторов с помощью метода Данилевского.

# Список использованных источников

1. Вычислительные методы: учебное пособие / А.А.Мицель. — Томск : Эль Контент, 2013. — 198 с. (дата обращения: 15.04.2022).

**Листинг программы**

#ifndef EIGENVALUE\_H

#define EIGENVALUE\_H

#include <iostream>

#include <fstream>

#include <vector>

#include <cmath>

#include "../../../2/Lab2Chm/Lab2Chm/Matrix.h"

#include "../../../2/Lab2Chm/Lab2Chm/Vector.h"

#include "Fraction.h"

#include "Polynomial.h"

#include "PolStr.h"

#define EPS\_B 1E-1

#define EPS 1E-4

#define MAX 31

namespace luMath

{

double unit\_matrix\_initer(size\_t m, size\_t n, size\_t r, size\_t c)

{

return r == c;

}

double zero\_vector\_initer(double m, double n)

{

return 0;

}

template<class T>

class Eigenvalue

{

public:

enum class TASK

{

EIGENVALUES = 1,

EIGENVECTORS,

ROOTS

};

private:

TASK \_task;

int m; // размерность квадратной матрицы

Matrix<T>\* A; // исходная матрица

Matrix<T>\* P; // матрица Фробениуса

Matrix<T>\* S; // вспомогательная матрица при вычислении матрицы Фробениуса

Vector<T>\* x; // собственные вектора

Vector<T>\* eigenvalues; // собственные числа

std::vector<T> k; // кратность собственных чисел/векторов

std::ofstream\* \_fout;

public:

Eigenvalue()

{

std::ifstream \_fin("input.txt");

\_fout = new std::ofstream("output.txt");

int c;

\_fin >> c; // считывается тип задачи

\_task = static\_cast<TASK>(c);

\_fin >> m;

T\* array = new T[m \* m];

for (int i = 0; i < m \* m; i++)

\_fin >> array[i];

\_fin.close();

A = new Matrix<T>(m, array);

delete[] array;

P = new Matrix<T>(m);

x = new Vector<T>[m];

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

x[i] = Vector<T>(m);

eigenvalues = new Vector<T>(m);

S = new Matrix<T>(m,unit\_matrix\_initer);

}

~Eigenvalue()

{

delete A;

delete P;

delete[] x;

delete eigenvalues;

delete S;

}

TASK getTask() { return \_task; }

static T getDeterminant(const Matrix<T>& \_Matrix) // Через метод Гаусса

{

Matrix<T> A(\_Matrix);

T determinant = 1;

// Прямой ход метода Гаусса - преобразование матрицы к треугольному виду

for (int i = 0; i < A.getRows(); i++) // проходим по всем строкам

{

T coeff = A[i][i]; // запоминаем коэффициент по диагонали

determinant \*= coeff;

for (int j = i; j < A.getRows(); j++) // проходим по всем элементам текущей строки, включая вектор коэффициентов

A[i][j] /= coeff;

for (int j = i + 1; j < A.getRows(); j++)

{

coeff = A[j][i];

for (int k = i; k < A.getCols(); k++)

A[j][k] -= coeff \* A[i][k];

}

}

return determinant;

}

// Получение собственных чисел методом Данилевского

Vector<T> getEigenvalues()

{

\*\_fout << "\t\tМетод Данилевского для нахождения собственных чисел.\n";

\*P = GetFrobenius();

\*\_fout << "\n\tМатрица Фробениуса:\n" << std::setw(10) << \*P;

int sign = (m % 2 == 0) ? 1 : -1;

Fraction<int>\* item = new Fraction<int>[m+1];

item[m] = sign;

for (int i = m - 1; i >= 0; i--)

item[i] = sign \* -(\*P)[0][m - i - 1];

Polynomial<Fraction<int>> pol(m + 1, item);

delete[] item;

\*\_fout << "\n\tПолученный полином:\t" << pol;

std::string ss = pol.to\_string();

const char\* polStr = CreatePolStr(ss.c\_str(), 0);

if (GetError() == ERR\_OK && polStr)

{

int n = 0; // номер корня

double x0 = 0; // Начальное приближение

double x1 = x0;

bool flag = false; // было ли на предыдущей итерации найдено приближение корня в промежутке (-EPS, EPS)

T root1 = EvalPolStr(polStr, x0, 0);

T root2 = root1;

for (x1 = EPS; n < m && x1 < MAX ; x1 += EPS)

{

root1 = root2;

root2 = EvalPolStr(polStr, x1, 0);

if (abs(root2) <= EPS\_B && abs(root2) < abs(root1))

{

x0 = x1;

flag = true;

}

else if (flag == true || root1\*root2<0)

{

int multiplicity = 0;

flag = false;

(\*eigenvalues)[n] = x0;

multiplicity++; n++;

double eps\_b = EPS\_B;

while (abs(EvalPolStr(polStr, x0, multiplicity)) < eps\_b)

{

multiplicity++;

eps\_b \*= 10;

}

for (int i = 0; i < multiplicity - 1; i++)

(\*eigenvalues)[n++] = x0;

}

}

for (x1 = EPS; n < m && x1 > -MAX; x1 -= EPS)

{

root1 = root2;

root2 = EvalPolStr(polStr, x1, 0);

if (abs(root2) <= EPS\_B && abs(root2) < abs(root1))

{

x0 = x1;

flag = true;

}

else if (flag == true)

{

int multiplicity = 0;

flag = false;

(\*eigenvalues)[n] = x0;

multiplicity++; n++;

double eps\_b = EPS\_B;

while (abs(EvalPolStr(polStr, x0, multiplicity)) < eps\_b)

{

multiplicity++;

eps\_b \*= 10;

}

for (int i = 0; i < multiplicity-1; i++)

(\*eigenvalues)[n++] = x0;

}

}

delete polStr;

n = 1;

int \_k;

int i;

Matrix<T> E(m, unit\_matrix\_initer);

for (i = 1; i < m; i++)

{

\_k = 1;

if ((\*eigenvalues)[i] == (\*eigenvalues)[i-1])

\_k++;

else

{

\*\_fout << "\n\tСобственное число #" << n++ << ": " << (\*eigenvalues)[i - 1] << "\n\t-> Кратность: " << \_k;

k.push\_back(\_k);

\*\_fout << "\n\tПроверка: " << getDeterminant(((\*A) - E \* (\*eigenvalues)[i - 1]));

\_k = 1;

}

}

\*\_fout << "\n\n\tСобственное число #" << n << ": " << (\*eigenvalues)[i - 1] << "\n\t-> Кратность: " << \_k;

k.push\_back(\_k);

\*\_fout << "\n\tПроверка: " << getDeterminant(((\*A) - E \* (\*eigenvalues)[i - 1])) << "\n";

}

return (\*eigenvalues);

}

// Получение собственных векторов методом Данилевского

Vector<T>\* getEigenvectors()

{

Vector<T>\* y = new Vector<T>[m];

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

{

y[i] = Vector<T>(m);

// Находим собственные вектора для матрицы Фробениуса

for (int j = m - 1; j >= 0; j--)

y[i][j] = pow((\*eigenvalues)[i], m-j-1);

y[i].transposition();

// Вычисляем собственные вектора исходной матрицы

x[i] = (\*S) \* y[i];

}

delete[] y;

int index = 0;

for (int i = 0; i < m; i += k[index++])

{

\*\_fout << "\n\tСобственное число #" << index + 1 << " : " << (\*eigenvalues)[i] << "\n\t-> Кратность: " << k[index]

<< "\n\tСоответствующий собственный вектор:\n" << x[i];

\*\_fout << "\nПроверка:\n" << (\*A) \* x[i] - (\*eigenvalues)[i] \* x[i];

}

return x;

}

Matrix<T> GetRoots()

{

std::ifstream \_fin("input2.txt");

int n;

double \_k;

\_fin >> n;

m = 0;

delete eigenvalues;

eigenvalues = new Vector<T>(n);

k.clear();

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

{

\_fin >> (\*eigenvalues)[i] >> \_k;

k.push\_back(\_k);

m += \_k;

}

Polynomial<T> pol({1});

std::string ss = "";

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

{

\*\_fout << "\n\tСобственное число #" << i+1 << ": " << (\*eigenvalues)[i] << "\n\t-> Кратность: " << k[i] << "\n";

Polynomial<T> temp\_pol({ -(\*eigenvalues)[i], 1 });

pol \*= temp\_pol;

for (int j = 1; j < k[i]; j++)

{

temp\_pol = Polynomial<T>({ -(\*eigenvalues)[i], 1 });

pol \*= temp\_pol;

}

}

int sign;

if(pol[m] < 0)

sign = (m % 2 == 0) ? 1 : -1;

else

sign = ((m + 1) % 2 == 0) ? 1 : -1;

Matrix<T> F(m, zero\_vector\_initer);

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

{

F[0][i] = sign \* -pol[m-i-1];

if(i < m-1)

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

}

\*\_fout << "\n\tПолученная матрица Фробениуса:\n" << std::setw(10) << F;

return F;

}

Matrix<T> GetFrobenius()

{

Matrix<T> P(\*A);

Matrix<T> tempA(\*A);

Matrix<T> M(m), M\_1(m);

Matrix<T> E(m, unit\_matrix\_initer);

for (int k = m - 1; k >= 0; k--)

{

// Вычисляем M(k)

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

for (int j = 0; j < m; j++)

{

if (i == (k-1) && j == (k-1)) // m(k,k) = 1 / a^(n-k-1)\_(k+1,k)

M[i][j] = 1 / tempA[k][k-1];

else if (i == (k-1)) // m(k,j) = -a^(n-k-1)\_(k+1,j) / a^(n-k-1)\_(k+1,k); j = 1,2,...,n; j != k

M[i][j] = -tempA[k][j] / tempA[k][k-1];

else //m(i,j) = e(i,j); i = 1,2,...,n; j = 1,2,...,n; i!=k

M[i][j] = E[i][j];

}

P \*= M; // ~A^k = A^(k-1) \* M\_(n-k)

(\*S) \*= M;

// Вычисляем M^-1(k)

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

for (int j = 0; j < m; j++)

{

if (i == (k - 1)) // m(k,j) = a^(n-k-1)\_(k+1,j); j = 1,2,...,n

M\_1[i][j] = tempA[k][j];

else //m(i,j) = e(i,j); i = 1,2,...,n; j = 1,2,...,n; i!=k

M\_1[i][j] = E[i][j];

}

P = M\_1 \* P; // A^k = M^(-1)\_(n-k) \* ~A^k

tempA = P;

}

return P;

}

};

}

#endif