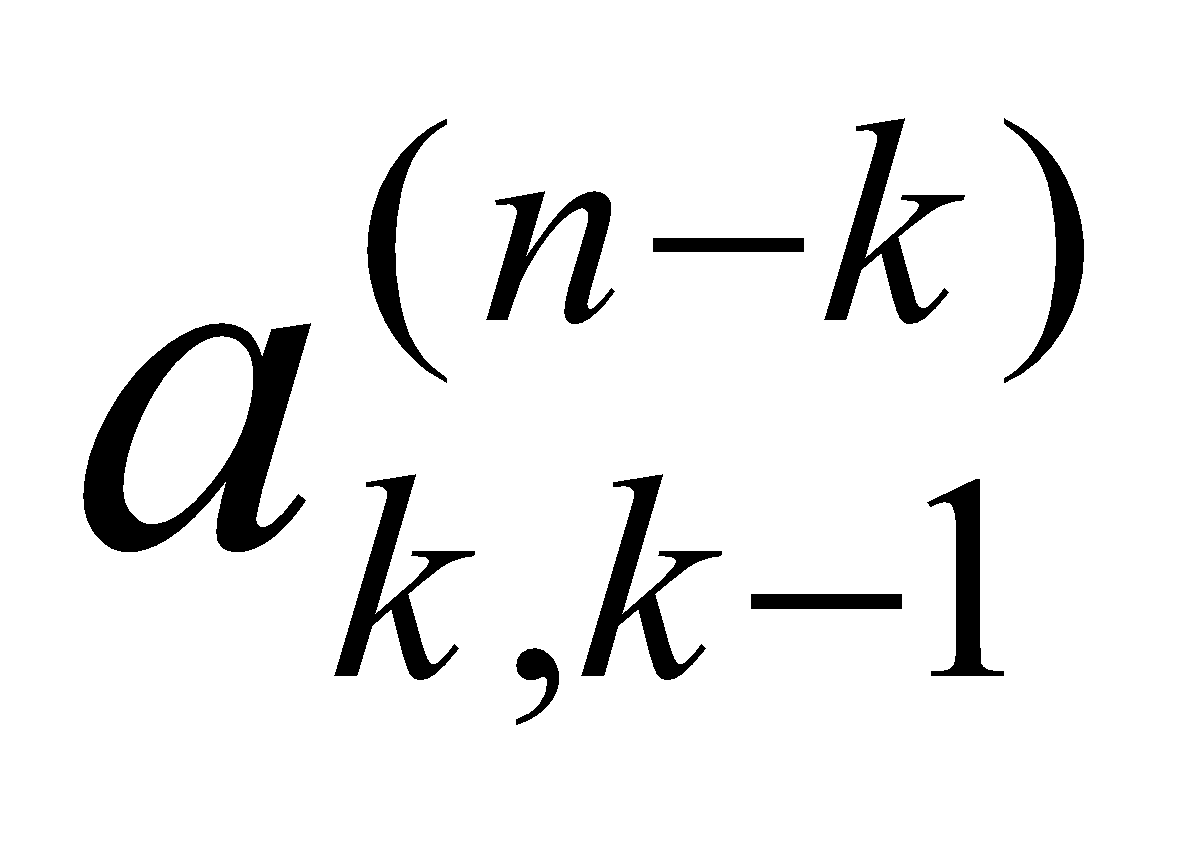
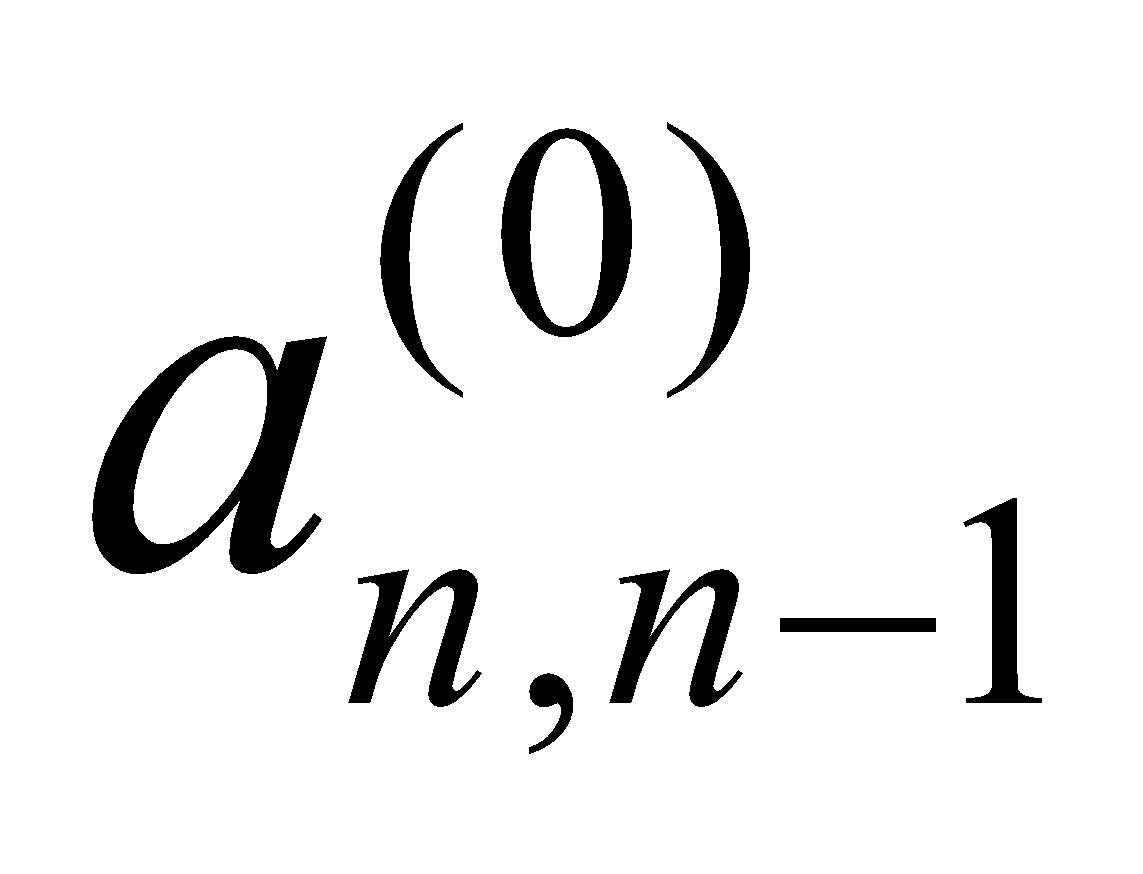
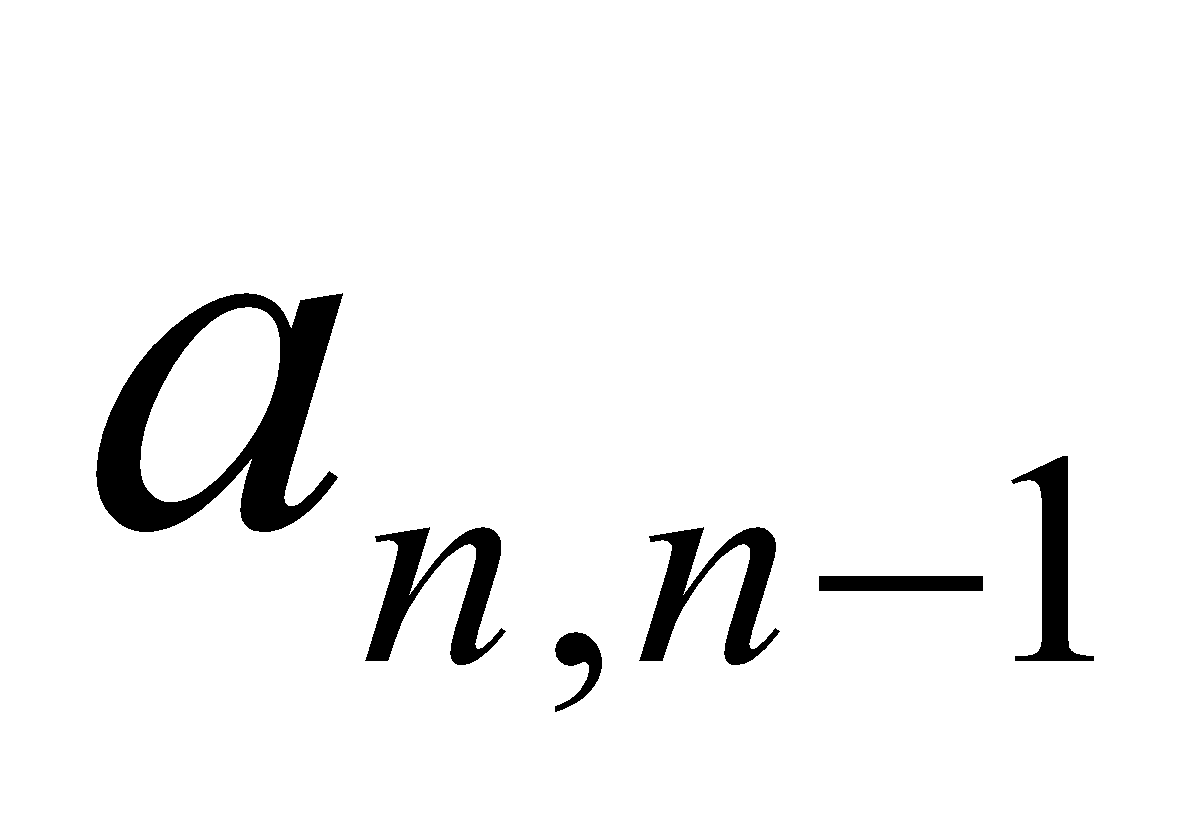
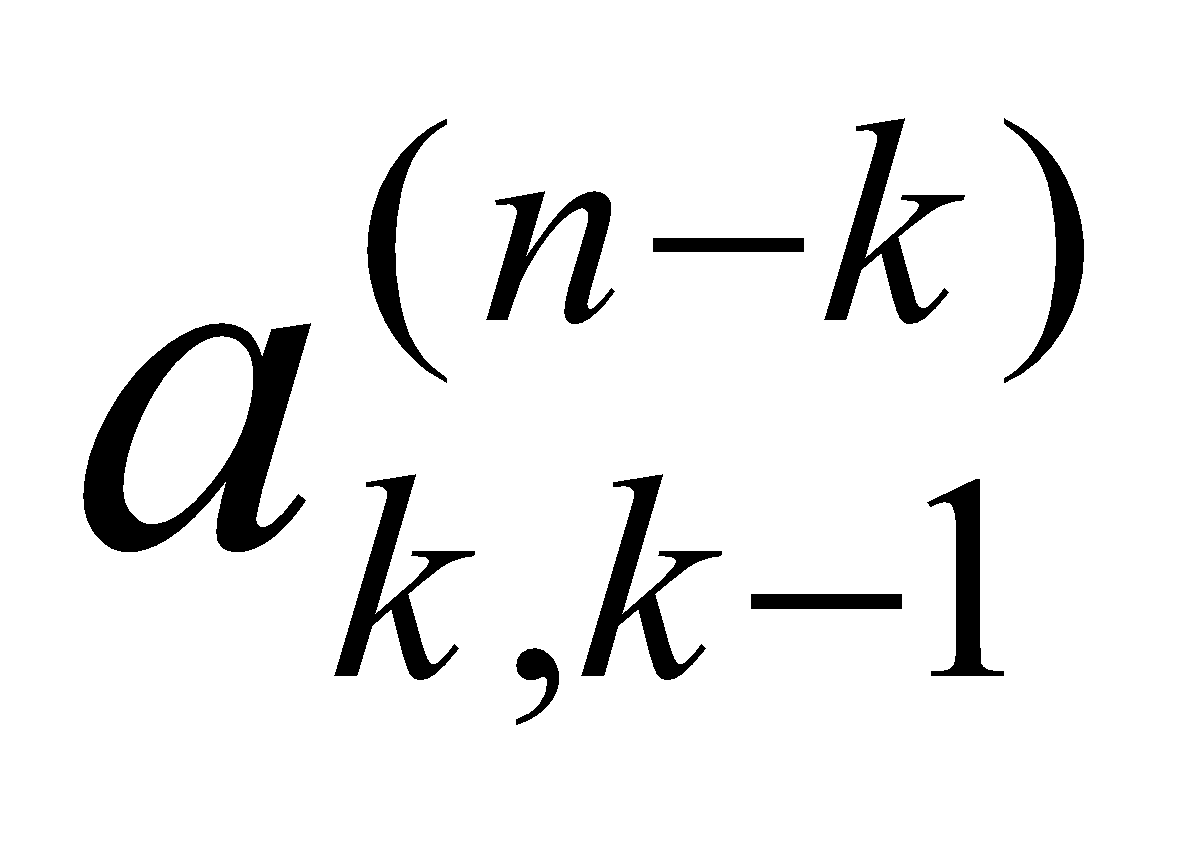
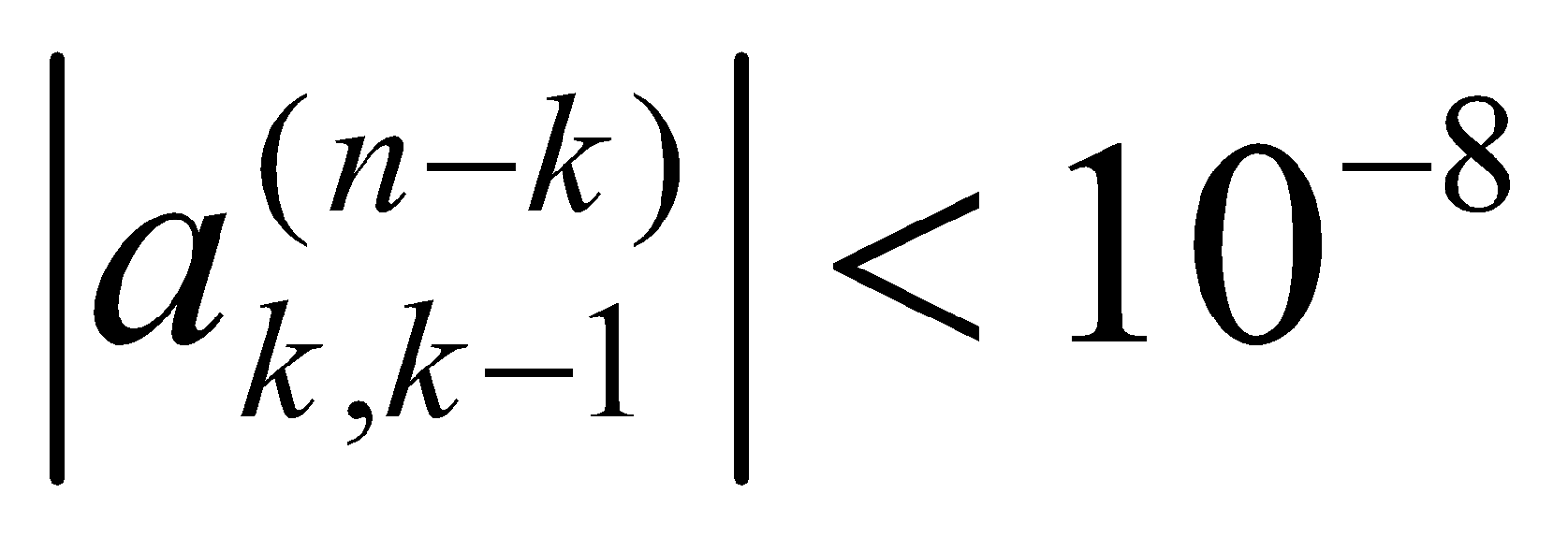
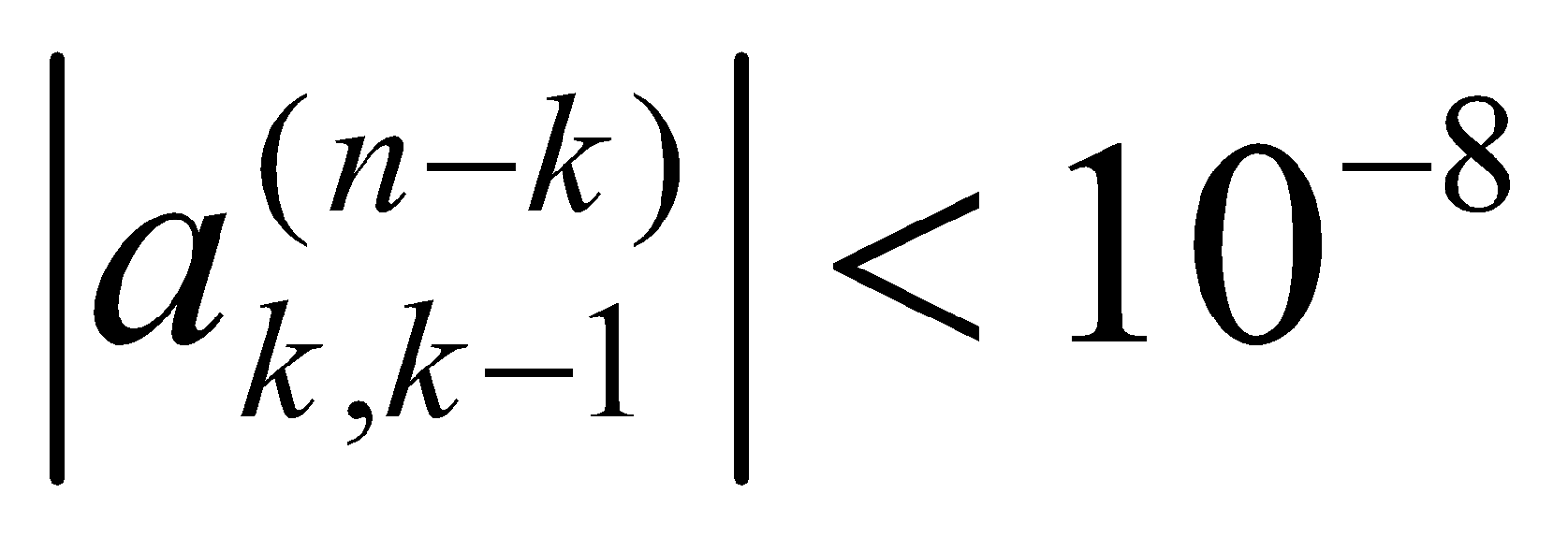
**Лабораторная работа 7 «Метод Данилевского»**

Выполнил студент 3 курса 4 группы ФПМИ БГУ Видевич Александр

**Задача.**

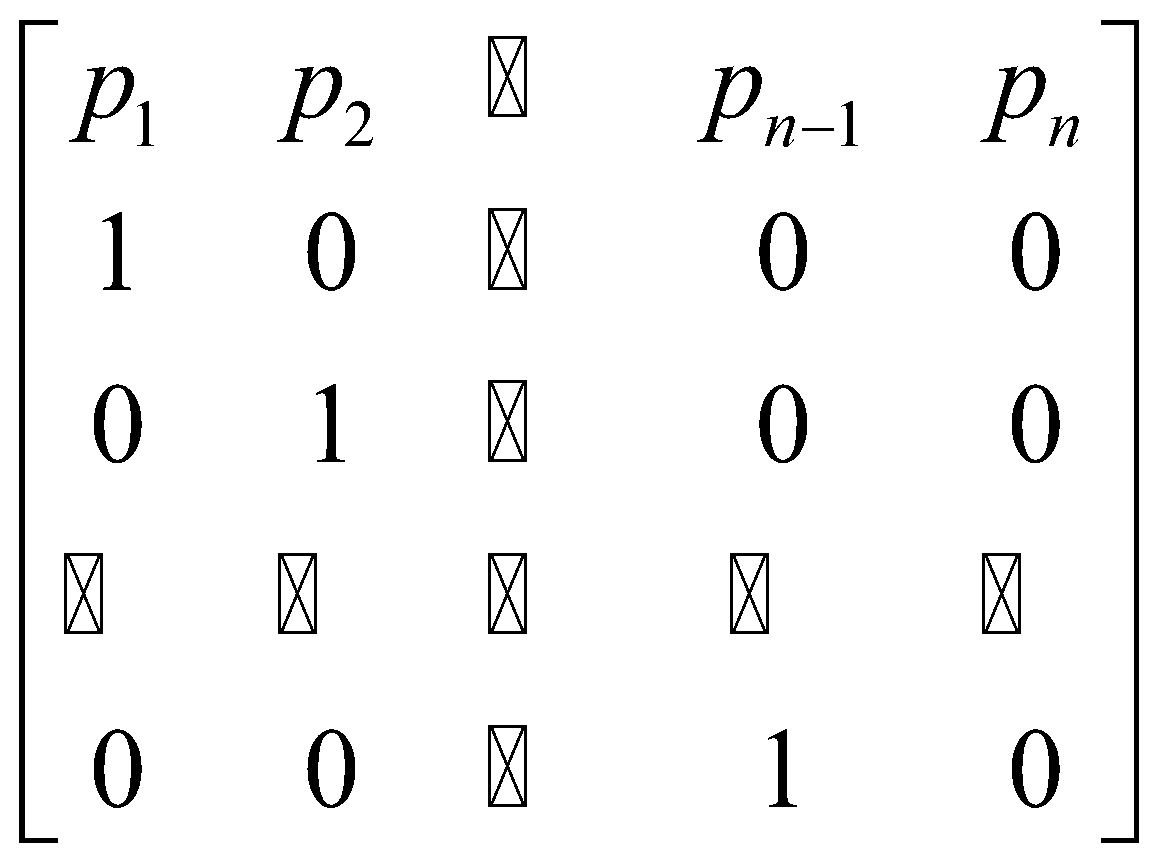
Изучить основы метода Данилевского. разработать программу приведения матрицы к канонической форме Фробениуса методом Данилевского (регулярный случай); сохранить матрицы, используемые для получения собственных векторов.

Для вычислений:

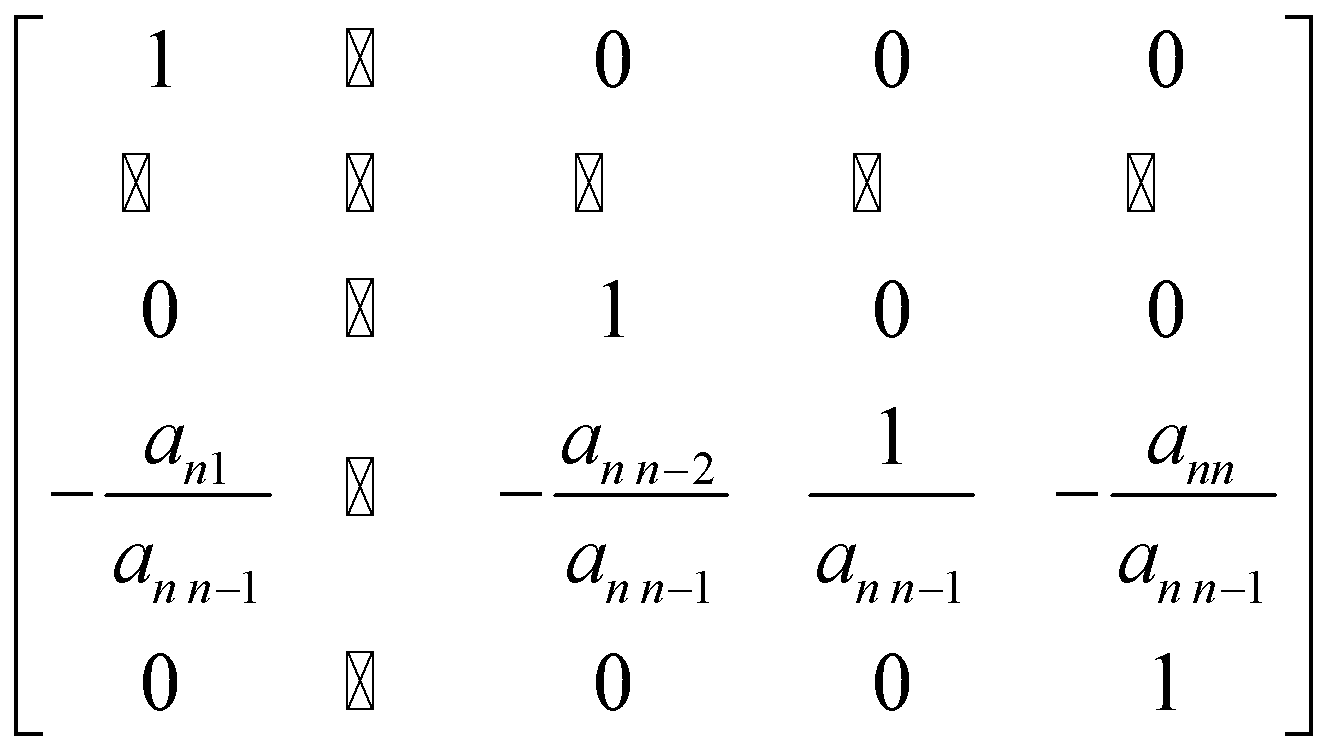
* *n=*4 – порядок матрицы;
* при заполнении матрицы *A* использовать случайные числа из диапазона от −50 до 50;
* использовать тип float;
* «ведущий элемент» на (*n*–*k*+1)-м шаге, *k=n*, *n*–1,... , 2, (=) должен быть отличным от нуля (рассматривается только такой регулярный случай);
* считать равным нулю, если ;
* если , то выдать соответствующее сообщение и заново заполнить матрицу *A*.

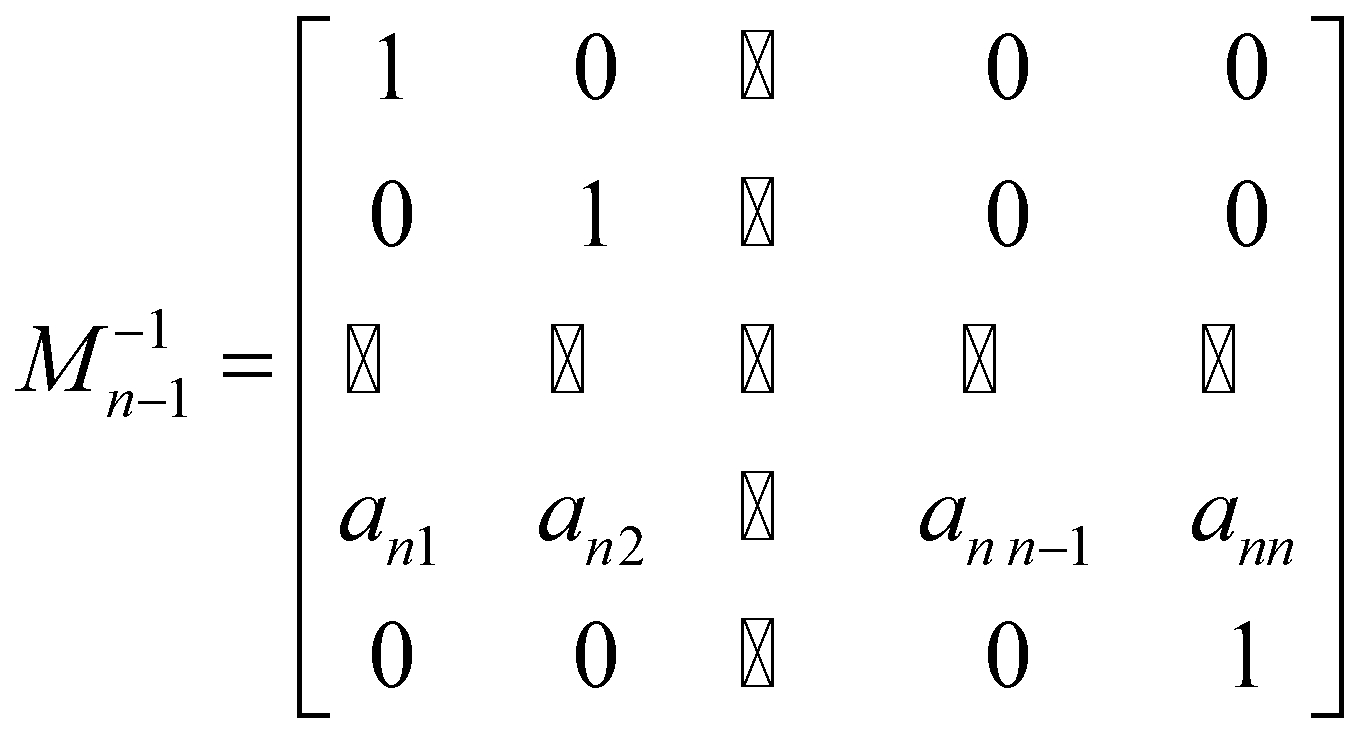
Программно реализовать для рассматриваемого примера алгоритм приведения матрицы к канонической форме Фробениуса. Порядок матрицы *n* должен быть в коде параметром (это требование не является обязательным). Сохранить матрицы *Mn–*1, *Mn–*2,... , *M*1, используемые для получения собственных векторов (понадобятся для другой работы). Вывести на печать матрицу *A* (входные данные), полученную каноническую форму Фробениуса Φ, матрицы *Mn–*1, *Mn–*2, ... , *M*1, коэффициент *p*1 (полученный из формы Фробениуса), след матрицы Sp *A*

Форма Фробениуса:

Φ*=**.*

Матрицы для приведения к форме Фробениуса:

*Mn–*1*=**.*

.

**Входные данные.**

n = 4

**Листинг программы.**

*#include* <iostream>

*#include* <vector>

*#include* <iomanip>

*#include* <optional>

int n = *4*;

float eps = *1e*-*8*;

std::*vector*<std::*vector*<std::*vector*<float>>> matrices;

std::*vector*<std::*vector*<float>> GenerateMatrix()

{

    std::*vector*<std::*vector*<float>> result(n, std::*vector*<float>(n));

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

    {

*for* (int j = *0*; j < n; ++j)

        {

            result[i][j] = (float((rand() % *100*) - *50*));

        }

    }

*return* result;

}

void PrintMatrix(*const* std::*vector*<std::*vector*<float>> *&*matrix)

{

*for* (*const* auto &r : matrix)

    {

*for* (auto c : r)

        {

            std::cout << std::setprecision(*5*) << c << " ";

        }

        std::cout << '*\n*';

    }

    std::cout << '*\n*';

}

float GetTrace(*const* std::*vector*<std::*vector*<float>> *&*matrix)

{

    float trace = *0*;

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

    {

        trace += matrix[i][i];

    }

*return* trace;

}

std::*vector*<std::*vector*<float>> MultiplyMatrices(*const* std::*vector*<std::*vector*<float>> *&*a,

*const* std::*vector*<std::*vector*<float>> *&*b)

{

    std::*vector*<std::*vector*<float>> result(n, std::*vector*<float>(n, *0*));

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

    {

*for* (int j = *0*; j < n; ++j)

        {

*for* (int k = *0*; k < n; ++k)

            {

                result[i][j] += a[i][k] \* b[k][j];

            }

        }

    }

*return* result;

}

std::*vector*<std::*vector*<float>> CreateM(*const* std::*vector*<std::*vector*<float>> *&*matrix, int order)

{

    std::*vector*<std::*vector*<float>> m(n, std::*vector*<float>(n, *0*));

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

    {

        m[i][i] = *1*;

*if* (i != order)

        {

*continue*;

        }

*for* (int j = *0*; j < n; ++j)

        {

*if* (j == order)

            {

                m[i][j] = (*1* / matrix[order + *1*][order]);

            }

*else*

            {

                m[i][j] = -(matrix[order + *1*][j] / matrix[order + *1*][order]);

            }

        }

    }

*return* m;

}

std::*vector*<std::*vector*<float>> CreateInvertM(*const* std::*vector*<std::*vector*<float>> *&*matrix, int order)

{

    std::*vector*<std::*vector*<float>> m(n, std::*vector*<float>(n, *0*));

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

    {

        m[i][i] = *1*;

*if* (i == order)

        {

            m[i] = matrix[i + *1*];

        }

    }

*return* m;

}

std::*vector*<std::*vector*<float>> RunAlgorithm(*const* std::*vector*<std::*vector*<float>> *&*matrix)

{

    std::*vector*<std::*vector*<float>> a = matrix;

    matrices.clear();

*for* (int i = n - *1*; i > *0*; --i)

    {

*if* (a[n - i][n - i - *1*] < eps)

        {

*return* {};

        }

        std::*vector*<std::*vector*<float>> m = CreateM(a, i - *1*);

        std::*vector*<std::*vector*<float>> inverse = CreateInvertM(a, i - *1*);

        std::*vector*<std::*vector*<float>> temp = MultiplyMatrices(inverse, a);

        a = MultiplyMatrices(temp, m);

        matrices.push\_back(m);

    }

*return* a;

}

int main()

{

    srand(time(*nullptr*));

    std::ios\_base::sync\_with\_stdio(*false*);

    std::cin.tie(*nullptr*);

    std::*vector*<std::*vector*<float>> matrix;

    std::*vector*<std::*vector*<float>> frob;

*while* (frob.empty())

    {

        std::cout << "Generated new matrix*\n*";

        matrix = GenerateMatrix();

        frob = RunAlgorithm(matrix);

    }

    std::cout << "Matrix A: " << std::endl;

    PrintMatrix(matrix);

    std::cout << "Trace: " << GetTrace(matrix) << std::endl;

    std::cout << "Matrix Frobenius: " << std::endl;

    PrintMatrix(frob);

    std::cout << "M matrices: *\n*"

              << std::endl;

*for* (*const* std::*vector*<std::*vector*<float>> &mtrx : matrices)

    {

        PrintMatrix(mtrx);

        std::cout << std::endl;

    }

*return* *0*;

}

**Выходные данные.**

Generated new matrix

Matrix A:

-18 27 -4 -37

18 -34 -38 10

-23 6 -1 26

6 0 43 -21

Trace: -74

Matrix Frobenius:

-74 -531 1.06e+05 1.2167e+06

1 7.6294e-06 0 0

0 1 0 0

5.6583e-12 7.1566e-10 1 -7.2702e-07

M matrices:

1 0 0 0

0 1 0 0

-0.13953 -0 0.023256 0.48837

0 0 0 1

1 0 0 0

2.5896 0.002381 0.05371 -2.0554

0 0 1 0

0 0 0 1

-1.1866e-05 -0.0015008 -0.020778 1.5247

0 1 0 0

0 0 1 0

0 0 0 1