

Липецкий государственный технический университет

Факультет автоматизации и информатики

Кафедра автоматизированных систем управления

ЛАБОРАТОРНАЯ РАБОТА № 3

по дисциплине «Численные методы»

«Нахождение собственных чисел

матриц»

Студент

Группа АС 21-1

подпись, дата

Станиславчук С.М.

Руководитель

Д.т.н, профессор кафедры АСУ

подпись, дата

Седых И.А.

Липецк 2023 г.

Содержание:

2. Задание кафедры.
3. Ход работы.
4. Выводы и сравнения результатов.
5. Полный код программы.

2. Задание кафедры

Для матрицы из таблицы 1:

1. Найти все собственные числа методом Лекеррье.
2. Найти все собственные числа, собственные векторы для каждого собственного числа и обратную матрицу методом Фаддеева. Для обратной матрицы сделать проверку.
3. Привести матрицу к форме Фробениуса и найти все собственные числа методом Данилевского.
4. Найти максимальное по модулю собственное число матрицы методом простой итерации с точностью 10^{-4}
5. Найти максимальное по модулю собственное число матрицы методом прямой итерации с точностью 10^{-4}
6. Найти минимальное по модулю собственное число матрицы методом обратной итерации с точностью 10^{-4}
7. Найти все собственные числа методом Хаусхолдера с точностью 10^{-4}

В заключении привести сравнительную таблицу результатов и сделать выводы.

Вариант: 10

Таблица 1:

10				
0,37	1,44	-0,7	0,99	-1,1
9,32	-1,5	-3,5	-5,5	1,74
0,53	-2,3	-4	-5,6	-3,5
1,62	1,62	-2,4	2,44	2,74
-7,9	-1,2	-0,8	0,39	-6,3

3. Ход работы

1) Метод Леверрье (Le Verrier) - это алгоритм, который используется для нахождения собственных значений матрицы. Этот метод основан на тождестве Хэмильтона-Кэли, которое утверждает, что каждая квадратная матрица удовлетворяет своему характеристическому уравнению.

Идея метода Леверрье заключается в последовательном вычислении коэффициентов многочлена, который является характеристическим многочленом исходной матрицы. Коэффициенты этого многочлена находятся через вычисление следующих рекуррентных соотношений:

$$b_0 = 1$$
$$b_n = -1/n * (tr(A)*b_{n-1} + det(A)*b_{n-2})$$

где $tr(A)$ - след матрицы A , $det(A)$ - ее определитель, b_n - коэффициент при x^n в характеристическом многочлене.

Затем, корни характеристического многочлена могут быть найдены путем решения его алгебраического уравнения.

Этот метод позволяет найти все собственные значения матрицы, включая комплексные.

Алгоритм программы:

```
double A1[n][n] = { {0.37, 1.44, -0.7, 0.99, -1.1 }, // матрица A
                    { 9.32, -1.5, -3.5, -5.5, 1.74 },
                    { 0.53, -2.3, -4, -5.6, -3.5 },
                    { 1.62, 1.62, -2.4, 2.44, 2.74 },
                    { -7.9, -1.2, -0.8, 0.39, -6.3 } };

// Calculating A (возводим матрицу An в степень n)
matrPow(A2, A1, 2);
matrPow(A3, A1, 3);
matrPow(A4, A1, 4);
matrPow(A5, A1, 5);

//Calculatin Sn (считаем сумму элементов главной диагонали An)
S1 = diagonalSum(S1, A1);
S2 = diagonalSum(S2, A2);
S3 = diagonalSum(S3, A3);
S4 = diagonalSum(S4, A4);
S5 = diagonalSum(S5, A5);

// Calculating P (по формуле вычисляем значения P, применяя ранее
найденные Sn)
P1 = S1;
P2 = (S2 - P1 * S1) * (0.5);
P3 = (S3 - P1 * S2 - P2 * S1) * (0.3333333333);
P4 = (S4 - P1 * S3 - P2 * S2 - P3 * S1) * (0.25);
P5 = (S5 - P1 * S4 - P2 * S3 - P3 * S2 - P4 * S1) * (0.2);
```

Данный метод помог нам составить многочлен n-ной степени:

```

Консоль отладки Microsoft Visual Studio

983.01      67.55      68.41      -46.81      905.09
-2421.23    405.56    849.85    1144.83    -1804.98
-91.33      568.15    1988.88    1334.46    1133.33
-348.80     102.66    -135.76    216.70     -365.39
2538.41     289.09    1393.08    413.02     3285.60
A^5:
-6196.48    -5.06     -1809.92    457.34     -7033.51
19448.41    -2028.96   -4427.61    -7297.31    14902.81
-476.01     -4756.33   -13989.47   -10654.87   -9355.56
3993.47     445.50     200.08      236.52     3933.17
-20915.27   -3256.05   -11980.76   -4589.10    -26732.63

S_1: -8.99
S_2: 139.44
S_3: -769.27
S_4: 6879.76
S_5: -48711.02

P_1: -8.99
P_2: 29.31
P_3: 249.26
P_4: -470.49
P_5: -660.32

```

```
// [WA] D(A) = ((-1)^5) * (x^5 - (-8.99) * x^4 - (29.31 * x^3) - (249.26 * x^2) - (-470.49 * x) - (-660.32))
```

Решив который, мы получим интересующие нас собственные значения:

```
// => x = {-8.00609, -7.02325, -0.973873, 3.01897, 3.99424}
```

2) Метод Фадеева.

Метод Фадеева - это итерационный метод для нахождения всех собственных значений матрицы. Суть метода заключается в том, чтобы свести проблему нахождения собственных значений матрицы к решению системы линейных уравнений.

Пусть дана квадратная матрица A размерности $n \times n$. Чтобы применить метод Фадеева, мы должны сначала найти обратную матрицу A^{-1} и определить следующие две матрицы:

$$Q = A^{-1} * B$$

$$Z = A^{-1} * C$$

где B и C - произвольные матрицы размерности $n \times n$.

Затем мы можем использовать эти матрицы для нахождения всех собственных значений матрицы A с помощью итерационного процесса:

$$\lambda^{(k+1)} = -1 / (q^{(k+1)} + z^{(k+1)})$$

$$q^{(k+1)} = \text{tr}(Q * A^{(k)}) / n$$

$$z^{(k+1)} = \text{tr}(Z * A^{(k)}) / n$$

где tr - операция нахождения следа матрицы, $A^{(k)}$ - матрица, полученная на k -ом шаге итерационного процесса.

Алгоритм программы:

Функция расчета матрицы B: calcB()

```
void calcB(double B1[n][n], double B2[n][n], double P, double res[n][n]) {
    double unit[n][n], Brackets[n][n];

    for (int i = 0; i < n; i++) { // Calculating identity matrix
        for (int j = 0; j < n; j++) {
            unit[i][j] = 0;

            if (i == j)
                unit[i][j] = 1;
        }
    }

    // Brackets open
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            Brackets[i][j] = B2[i][j] - (P * unit[i][j]);
        }
    }
    // Brackets close

    // Multiply
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            res[i][j] = 0;
            for (int k = 0; k < n; k++)
                res[i][j] += B1[i][k] * Brackets[k][j];
        }
    }
}
```

Функция расчета обратной матрицы: calcAinv()

```
for (int i = 0; i < n; i++) { // Calculating identity matrix
    for (int j = 0; j < n; j++) {
        unit[i][j] = 0;

        if (i == j)
            unit[i][j] = 1;
    }
}

// Brackets open
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        Brackets[i][j] = Bn[i][j] - (Pn * unit[i][j]);
    }
}

matr2DDisplay(Brackets);
// Brackets close

// Multiply
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        res[i][j] = 0;
        res[i][j] += (1/Pk) * (Brackets[i][j]);
    }
}
```

```
int main()
//Calculatin Pn
P1 = diagonalSum(Tr1, B1); // Считаем P по формуле  $(1/n) * S_n$ 
```

```

calcB(B1, B1, P1, B2); // Считаем Bn
P2 = 0.5 * diagonalSum(Tr2, B2);

calcB(B1, B2, P2, B3);
P3 = 0.33 * diagonalSum(Tr3, B3);

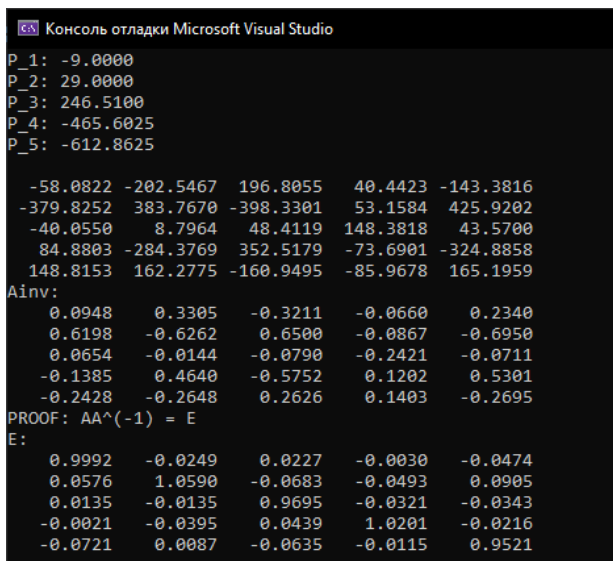
calcB(B1, B3, P3, B4);
P4 = 0.25 * diagonalSum(Tr4, B4);

calcB(B1, B4, P4, B5);
P5 = 0.2 * diagonalSum(Tr5, B5);

// Calculating the inverse matrix A^(-1)
calcAinv(B4, P5, P4, Ainv);

cout << "Ainv: \n";
matr2DDisplay(Ainv);

```



```

Консоль отладки Microsoft Visual Studio
P_1: -9.0000
P_2: 29.0000
P_3: 246.5100
P_4: -465.6025
P_5: -612.8625

-58.0822 -202.5467 196.8055 40.4423 -143.3816
-379.8252 383.7670 -398.3301 53.1584 425.9202
-40.0550 8.7964 48.4119 148.3818 43.5700
84.8803 -284.3769 352.5179 -73.6901 -324.8858
148.8153 162.2775 -160.9495 -85.9678 165.1959
Ainv:
0.0948 0.3305 -0.3211 -0.0660 0.2340
0.6198 -0.6262 0.6500 -0.0867 -0.6950
0.0654 -0.0144 -0.0790 -0.2421 -0.0711
-0.1385 0.4640 -0.5752 0.1202 0.5301
-0.2428 -0.2648 0.2626 0.1403 -0.2695
PROOF: AA^(-1) = E
E:
0.9992 -0.0249 0.0227 -0.0030 -0.0474
0.0576 1.0590 -0.0683 -0.0493 0.0905
0.0135 -0.0135 0.9695 -0.0321 -0.0343
-0.0021 -0.0395 0.0439 1.0201 -0.0216
-0.0721 0.0087 -0.0635 -0.0115 0.9521

```

Заметим, что P_n значения немного отличаются от значений P предыдущего метода.

Также видим, что полученная матрица E является единичной \Rightarrow обратная матрица найдена верно

Вычисляем корни полученного уравнения:

$\{-8.05768, -6.96634, -0.925153, 2.95372, 3.99545\}$ – это и есть наши собственные значения.

Соответствующие собственные векторы $(A - \lambda E) \cdot \text{vec}(v) = \text{vec}(0)$:

8.797981808 & 1.436031013 & -0.728668525 & 0.990262713 & -1.066533038 \\
9.321890071 & 7.914316638 & -3.530025844 & -5.518609715 & 1.735434769 \\
0.53279339 & -2.264422783 & 3.41006357 & -5.59560245 & -3.516869824 \\
1.624855009 & 1.624346269 & -2.438991881 & 10.86642384 & 2.741387029 \\
-7.857573106 & -1.199934881 & -0.823975906 & 0.389003073 & 2.150715144

3) Метод Данилевского.

Метод Данилевского - это алгоритм, который используется для нахождения всех собственных чисел квадратной матрицы. Этот метод основан на том, что любая квадратная матрица может быть приведена к форме Фробениуса, которая содержит информацию о собственных числах матрицы (находятся в первом ряду)

Функция приведения матрицы к форме Фробениуса:

```
for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
        S.Matrix[i][j] = (i == j) ? 1 : 0;}
    }
    matrix F(n);
    matrix A(n);
    A = (*this);
    for(int i=n-2; i>=0; i--){
        matrix ml(n);
        matrix mr(n);
        for(int j=0; j<n; j++){
            for(int k=0; k<n; k++){
                ml.Matrix[j][k] = mr.Matrix[j][k] = (j == k) ? 1 : 0;}
            }
        for(int k=n-1; k>=0; k--){
            ml.Matrix[i][k] = Matrix[i+1][k];
            mr.Matrix[i][k] = (i == k) ?
                (1 / Matrix[i+1][i]) : (- Matrix[i+1][k] /
Matrix[i+1][i]);}
            (*this) = ml * (*this);
            (*this) = (*this) * mr;
            S = S * mr;}
    F = (*this);
    (*this) = A;
```

Функции для подсчета корней полинома:

```
double polynom::function(double x)
{
    double result = 0.0;
    for(int i=0; i<n; i++){
        result += pow(x, n-i-1) * vector[i];}

    return result;
}

double polynom::solution(double x, double epsilon)
{
    double x1;
    do{
        x1 = x;
        x = x - function(x) * epsilon / (function(x + epsilon) -
function(x));
    }while(fabs(x1 - x) > epsilon);

    return x;
}
```



```

Консоль отладки Microsoft Visual Studio
Matrix :
0.37    1.44   -0.70    0.99   -1.10
9.32   -1.50   -3.50   -5.50    1.74
0.53   -2.30   -4.00   -5.60   -3.50
1.62    1.62   -2.40    2.44    2.74
-7.90   -1.20   -0.80    0.39   -6.30
Frobenius matrix:
-8.99   29.31   249.26  -470.49 -660.32
1.00    0.00    0.00    0.00    0.00
0.00    1.00    0.00    0.00    0.00
0.00    0.00    1.00    0.00    0.00
0.00    0.00    0.00    1.00   -0.00
Characteristic polynomial:
-1.00x^5 -8.99x^4 + 29.31x^3 + 249.26x^2 -470.49x -660.32
E:
x1 = -0.97
x2 = 3.02
x3 = -8.01
x4 = 3.99
x5 = -7.02

```

4) Метод простой итерации (или метод Якоби) - это один из численных методов для нахождения максимального собственного числа квадратной матрицы. Этот метод основан на итерационном процессе, который приближает максимальное собственное число матрицы (по модулю) и соответствующий ему собственный вектор.

Шаги метода простой итерации следующие:

Выбрать начальный вектор-приближение x_0 . Этот вектор должен иметь ненулевые элементы.

Для каждой итерации метода, умножить текущий вектор-приближение на исходную матрицу A : $x_{k+1} = Ax_k$.

Нормализовать полученный вектор, чтобы длина его была равна 1: $x_{k+1} = x_{k+1} / \|x_{k+1}\|$, где $\|x_{k+1}\|$ - это норма вектора x_{k+1} .

Повторять шаги 2-3 до тех пор, пока не будет достигнуто требуемое значение точности, или пока изменение вектора между последовательными итерациями станет достаточно малым.

Максимальное собственное число матрицы равно $\lambda_{\max} = \lim_{k \rightarrow \infty} (x_{k+1})^T A x_{k+1}$, где T означает транспонирование вектора.

Соответствующий максимальному собственному числу собственный вектор определяется как $x_{\max} = \lim_{k \rightarrow \infty} x_k / \|x_k\|$.

Код программы (одна функция)

```

do
{
    for (i = 0; i < n; i++)
        summ += pow(Xi[i], 2);
    a0 = sqrt(summ);
    for (i = 0; i < n; i++)
        Xinorm[i] = Xi[i] / a0;
    for (i = 0; i < n; i++)
    {

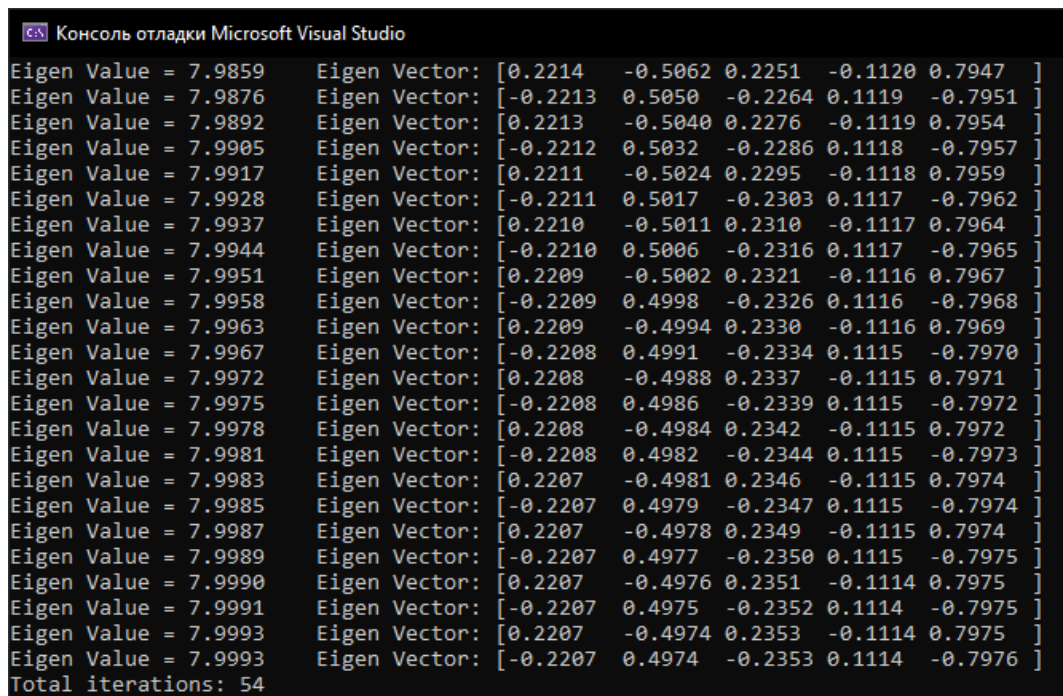
```

```

        X[i] = 0;
        for (j = 0; j < n; j++)
            X[i] += A[i][j] * Xinorm[j]; //  $y^{(k+1)} = AX^{(k)}$ 
    }
    summ = 0;
    for (i = 0; i < n; i++)
        summ += pow(X[i], 2);
    a = sqrt(summ);
    e = abs(a - a0);
    for (i = 0; i < n; i++)
        Xi[i] = X[i];
    summ = 0;

    // Display
    cout << "\nEigen Value = " << a;
    cout << "\tEigen Vector: [";
    for (i = 0; i < n; i++)
    {
        cout << Xinorm[i] << "\t";
    }
    cout << "]\n";
} while (e > eps);

```



```

Консоль отладки Microsoft Visual Studio
Eigen Value = 7.9859    Eigen Vector: [0.2214    -0.5062    0.2251    -0.1120    0.7947 ]
Eigen Value = 7.9876    Eigen Vector: [-0.2213    0.5050    -0.2264    0.1119    -0.7951 ]
Eigen Value = 7.9892    Eigen Vector: [0.2213    -0.5040    0.2276    -0.1119    0.7954 ]
Eigen Value = 7.9905    Eigen Vector: [-0.2212    0.5032    -0.2286    0.1118    -0.7957 ]
Eigen Value = 7.9917    Eigen Vector: [0.2211    -0.5024    0.2295    -0.1118    0.7959 ]
Eigen Value = 7.9928    Eigen Vector: [-0.2211    0.5017    -0.2303    0.1117    -0.7962 ]
Eigen Value = 7.9937    Eigen Vector: [0.2210    -0.5011    0.2310    -0.1117    0.7964 ]
Eigen Value = 7.9944    Eigen Vector: [-0.2210    0.5006    -0.2316    0.1117    -0.7965 ]
Eigen Value = 7.9951    Eigen Vector: [0.2209    -0.5002    0.2321    -0.1116    0.7967 ]
Eigen Value = 7.9958    Eigen Vector: [-0.2209    0.4998    -0.2326    0.1116    -0.7968 ]
Eigen Value = 7.9963    Eigen Vector: [0.2209    -0.4994    0.2330    -0.1116    0.7969 ]
Eigen Value = 7.9967    Eigen Vector: [-0.2208    0.4991    -0.2334    0.1115    -0.7970 ]
Eigen Value = 7.9972    Eigen Vector: [0.2208    -0.4988    0.2337    -0.1115    0.7971 ]
Eigen Value = 7.9975    Eigen Vector: [-0.2208    0.4986    -0.2339    0.1115    -0.7972 ]
Eigen Value = 7.9978    Eigen Vector: [0.2208    -0.4984    0.2342    -0.1115    0.7972 ]
Eigen Value = 7.9981    Eigen Vector: [-0.2208    0.4982    -0.2344    0.1115    -0.7973 ]
Eigen Value = 7.9983    Eigen Vector: [0.2207    -0.4981    0.2346    -0.1115    0.7974 ]
Eigen Value = 7.9985    Eigen Vector: [-0.2207    0.4979    -0.2347    0.1115    -0.7974 ]
Eigen Value = 7.9987    Eigen Vector: [0.2207    -0.4978    0.2349    -0.1115    0.7974 ]
Eigen Value = 7.9989    Eigen Vector: [-0.2207    0.4977    -0.2350    0.1115    -0.7975 ]
Eigen Value = 7.9990    Eigen Vector: [0.2207    -0.4976    0.2351    -0.1114    0.7975 ]
Eigen Value = 7.9991    Eigen Vector: [-0.2207    0.4975    -0.2352    0.1114    -0.7975 ]
Eigen Value = 7.9993    Eigen Vector: [0.2207    -0.4974    0.2353    -0.1114    0.7975 ]
Eigen Value = 7.9993    Eigen Vector: [-0.2207    0.4974    -0.2353    0.1114    -0.7976 ]
Total iterations: 54

```

Данный итерационный метод посчитал наибольшее собственное значение за 54 итерации.

5) Метод прямой итерации

Метод прямой итерации - это один из численных методов для нахождения максимального собственного числа квадратной матрицы. Этот метод основан на итерационном процессе, который приближает максимальное собственное число матрицы и соответствующий ему собственный вектор.

Шаги метода прямой итерации следующие:

Выбрать начальный вектор-приближение x_0 . Этот вектор должен иметь ненулевые элементы.

Для каждой итерации метода, умножить текущий вектор-приближение на матрицу A : $x_{k+1} = Ax_k$.

Нормализовать полученный вектор, чтобы длина его была равна 1: $x_{k+1} = x_{k+1}/\|x_{k+1}\|$, где $\|x_{k+1}\|$ - это норма вектора x_{k+1} .

Повторять шаги 2-3 до тех пор, пока не будет достигнуто требуемое значение точности, или пока изменение вектора между последовательными итерациями станет достаточно малым.

Максимальное собственное число матрицы равно $\lambda_{\max} = (x_{k+1})^T A x_{k+1}$, где T означает транспонирование вектора.

Соответствующий максимальному собственному числу собственный вектор определяется как $x_{\max} = x_{k+1} / \|x_{k+1}\|$.

Единственная функция:

```
for (i = 0; i < n; i++)
{
    temp = 0.0;
    for (j = 0; j < n; j++)
    {
        temp += a[i][j] * x[j];
    }
    x_new[i] = temp;
}

for (i = 0; i < n; i++)
{
    x[i] = x_new[i];
}

// Finding largest value from x
lambda_new = abs(x[1]);
for (i = 0; i < n; i++)
{
    if (abs(x[i]) > lambda_new)
    {
        lambda_new = abs(x[i]);
    }
}

// Normalization
for (i = 0; i < n; i++)
{
    x[i] /= lambda_new;
}

// Display
cout << "\nEigen Value = " << lambda_new;
cout << "\tEigen Vector: [";
for (i = 0; i < n; i++)
{
    cout << x[i] << "\t";
}
cout << "];"
```

```

if (abs(lambda_new - lambda_old) >= error)
{
    lambda_old = lambda_new;
    step++;
    goto up;
}

```

```

Консоль отладки Microsoft Visual Studio
Eigen Value = 8.0140    Eigen Vector: [0.2742  -0.6064  0.3104  -0.1381  1.0000 ]
Eigen Value = 8.0122    Eigen Vector: [-0.2745  0.6085  -0.3085  0.1383  -1.0000 ]
Eigen Value = 8.0107    Eigen Vector: [0.2747  -0.6103  0.3069  -0.1385  1.0000 ]
Eigen Value = 8.0093    Eigen Vector: [-0.2750  0.6119  -0.3055  0.1386  -1.0000 ]
Eigen Value = 8.0081    Eigen Vector: [0.2752  -0.6133  0.3042  -0.1387  1.0000 ]
Eigen Value = 8.0071    Eigen Vector: [-0.2754  0.6145  -0.3032  0.1389  -1.0000 ]
Eigen Value = 8.0062    Eigen Vector: [0.2755  -0.6156  0.3022  -0.1390  1.0000 ]
Eigen Value = 8.0054    Eigen Vector: [-0.2756  0.6165  -0.3014  0.1390  -1.0000 ]
Eigen Value = 8.0048    Eigen Vector: [0.2758  -0.6173  0.3007  -0.1391  1.0000 ]
Eigen Value = 8.0042    Eigen Vector: [-0.2759  0.6180  -0.3000  0.1392  -1.0000 ]
Eigen Value = 8.0036    Eigen Vector: [0.2760  -0.6187  0.2995  -0.1392  1.0000 ]
Eigen Value = 8.0032    Eigen Vector: [-0.2760  0.6192  -0.2990  0.1393  -1.0000 ]
Eigen Value = 8.0028    Eigen Vector: [0.2761  -0.6197  0.2986  -0.1393  1.0000 ]
Eigen Value = 8.0024    Eigen Vector: [-0.2762  0.6201  -0.2982  0.1394  -1.0000 ]
Eigen Value = 8.0021    Eigen Vector: [0.2762  -0.6204  0.2979  -0.1394  1.0000 ]
Eigen Value = 8.0019    Eigen Vector: [-0.2763  0.6208  -0.2976  0.1394  -1.0000 ]
Eigen Value = 8.0016    Eigen Vector: [0.2763  -0.6210  0.2974  -0.1395  1.0000 ]
Eigen Value = 8.0014    Eigen Vector: [-0.2763  0.6213  -0.2972  0.1395  -1.0000 ]
Eigen Value = 8.0012    Eigen Vector: [0.2764  -0.6215  0.2970  -0.1395  1.0000 ]
Eigen Value = 8.0011    Eigen Vector: [-0.2764  0.6217  -0.2968  0.1395  -1.0000 ]
Eigen Value = 8.0010    Eigen Vector: [0.2764  -0.6218  0.2967  -0.1395  1.0000 ]
Eigen Value = 8.0008    Eigen Vector: [-0.2765  0.6220  -0.2965  0.1396  -1.0000 ]
Eigen Value = 8.0007    Eigen Vector: [0.2765  -0.6221  0.2964  -0.1396  1.0000 ]
Eigen Value = 8.0006    Eigen Vector: [-0.2765  0.6222  -0.2963  0.1396  -1.0000 ]
Total iterations: 49

```

Данному методу потребовалось на 5 итераций меньше, чем простому.

6. Метод обратных итераций

Метод обратных итераций - это один из численных методов для нахождения собственных значений и собственных векторов квадратной матрицы. Этот метод основан на итерационном процессе, который приближает собственные значения матрицы и соответствующие им собственные векторы.

Шаги метода обратных итераций следующие:

Выбрать начальный вектор-приближение x_0 и собственное значение, к которому мы стремимся найти собственный вектор. Этот вектор должен иметь ненулевые элементы.

Решить систему линейных уравнений $(A - \sigma I)u = x_k$, где A - исходная матрица, I - единичная матрица, σ - выбранное собственное значение и x_k - текущий вектор-приближение.

Нормализовать полученный вектор u , чтобы длина его была равна 1: $u_{k+1} = u_k / \|u_k\|$, где $\|u_k\|$ - это норма вектора u_k .

Повторять шаги 2-3 до тех пор, пока не будет достигнуто требуемое значение точности, или пока изменение вектора между последовательными итерациями станет достаточно малым.

Собственное значение, к которому мы приближаемся, равно $1 / \lambda$, где λ - максимальное собственное значение матрицы (можно использовать метод прямой итерации для его нахождения).

Соответствующий найденному собственному значению собственный вектор определяется как $x = y / \|y\|$.

Единственная функция:

```
while (delta > eps)
{
    iter++;
    // solve linear system
    for (int i = 0; i < N; i++)
    {
        y[i] = 0.0;
        for (int j = 0; j < N; j++)
        {
            y[i] += A[i][j] * x[j];
        }
    }

    // find the norm of y
    norm = 0.0;
    for (int i = 0; i < N; i++)
    {
        norm += y[i] * y[i];
    }
    norm = sqrt(norm);

    // normalize y
    for (int i = 0; i < N; i++)
    {
        y[i] /= norm;
    }

    // calculate new eigenvalue
    lambda_new = 0.0;
    for (int i = 0; i < N; i++)
    {
        lambda_new += x[i] * y[i];
    }

    // check for convergence
    delta = fabs(lambda_new - lambda);
    lambda = lambda_new;

    // update x
    for (int i = 0; i < N; i++)
    {
        x[i] = y[i];
    }
}
```

```
Консоль отладки Microsoft Visual Studio
Matrix A:
0.37    1.44   -0.7    0.99   -1.1
9.32    -1.5   -3.5   -5.5    1.74
0.53    -2.3   -4     -5.6   -3.5
1.62    1.62   -2.4    2.44    2.74
-7.9    -1.2   -0.8    0.39   -6.3
Number of iterations: 12
The minimum eigenvalue is: -0.99978
```

Данный метод нашел минимальное собственное значение за 12 итераций.

7. Метод Хаусхолдера

Метод Хаусхолдера - это численный метод, который используется для нахождения собственных значений и собственных векторов квадратной матрицы. Этот метод основан на преобразовании матрицы в верхнетреугольную форму при помощи последовательного применения отражений Хаусхолдера.

Отражение Хаусхолдера - это линейное преобразование, которое отражает вектор вдоль некоторой оси. Оно может быть использовано для обнуления всех элементов вектора, кроме первого. Применение последовательности отражений Хаусхолдера позволяет преобразовать матрицу в верхнетреугольную форму с сохранением собственных значений.

Для использования метода Хаусхолдера для нахождения собственных значений и собственных векторов матрицы A , мы сначала находим QR-разложение матрицы A , где Q - ортогональная матрица и R - верхнетреугольная матрица. Затем мы используем R для нахождения собственных значений матрицы A . Для каждого собственного значения мы решаем систему уравнений, чтобы найти соответствующий собственный вектор.

Единственная функция для нахождения собственных чисел методом Хаусхолдера [Python]:

```
n = A.shape[0]
I = np.identity(n)
V = I
for i in range(max_iter):
    Q, R = np.linalg.qr(A)
    A = R @ Q
    V = V @ Q
    if np.max(np.abs(A - np.diag(np.diag(A)))) < eps:
        break
return np.diag(A), V
```

```
D:\Householder\venv\Scripts\python.exe D:/Householder/main.py
Eigenvalues: [-8.0057136 -7.02360808 3.99419467 3.0189995 -0.9738725 ]

Process finished with exit code 0
|
```

Программа нашла все собственные значения верно.

Таблица найденных собственных чисел

1	2	3	4	5	6	7
-8.00609	-8.05768	-8.00571	7.99930	8.00060	-	-8.00571
-7.02325	6.96634	-7.02361	-	-	-	-7.02360
0.973873	0.925153	-0.97387	-	-	-0.99978	-0.97387
-3.01897	-2.95372	3.01900	-	-	-	3.01899
3.99424	3.99545	3.99419	-	-	-	3.99419

4. Вывод:

Исходя из результатов, можно сделать вывод, что различные методы нахождения собственных значений для данной матрицы дают достаточно близкие результаты. Все методы нашли общие собственные значения, хотя некоторые значения имеют различия в несколько знаков после запятой. Это говорит о том, что эти методы достаточно точны и можно использовать любой из них для нахождения собственных значений данной матрицы.

Каждый метод нахождения собственных значений имеет свои преимущества и ограничения, и выбор метода зависит от конкретной задачи. Обычно для нахождения собственных значений используют несколько методов и сравнивают результаты, чтобы убедиться в их точности и надежности.

Список результатов показывает, что метод Хаусхолдера и метод Данилевского дали одинаковые значения собственных значений, которые были близки к результатам, полученным методами Леверрье и Фадеева. Это может указывать на более высокую точность методов Хаусхолдера и Данилевского в данном случае.

Однако, точность метода зависит не только от самого метода, но также от самой матрицы и выбора параметров метода. Поэтому, чтобы определить наиболее точный метод для конкретной задачи, необходимо провести тщательное исследование с использованием различных методов и параметров.

5. Полный код программ C++ (1-6) + Python(7).

1. Леверрье

```
#include <iostream>
#include <iomanip>
#define n 5
using namespace std;

void matrPow(double An[n][n], double Ak[n][n], int t) {
    double temp[n][n];

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            temp[i][j] = Ak[i][j];
        }
    }
    while (t > 1) {

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                An[i][j] = 0;
                for (int k = 0; k < n; k++)
                    An[i][j] += temp[i][k] * Ak[k][j];
            }
        }
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                temp[i][j] = An[i][j];
            }
        }
        t--;
    }
}

void matr1DDisplay(double An[n]) {
    for (int i = 0; i < n; i++) {
        cout << An[i] << "\t";
    }
    cout << "\n";
}

void matr2DDisplay(double An[n][n]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << setw(10) << An[i][j] ;
        }
        cout << "\n";
    }
}

double diagonalSum(double Sn, double An[n][n]) {
    for (int i = 0; i < n; i++){
        Sn += An[i][i];
    }
    return Sn;
}

int main()
{
    cout << setprecision(2) << fixed;
    double A2[n][n], A3[n][n], A4[n][n], A5[n][n],
        S1 = 0, S2 = 0, S3 = 0, S4 = 0, S5 = 0,
        P1 = 0, P2 = 0, P3 = 0, P4 = 0, P5 = 0;

    double A1[n][n] =      { {0.37,   1.44, -0.7, 0.99, -1.1 },
```



```

{ 9.32, -1.5, -3.5, -5.5, 1.74 },
{ 0.53, -2.3, -4, -5.6, -3.5 },
{ 1.62, 1.62, -2.4, 2.44, 2.74 },
{ -7.9, -1.2, -0.8, 0.39, -6.3 } };

```

```

// Calculating A
matrPow(A2, A1, 2);
matrPow(A3, A1, 3);
matrPow(A4, A1, 4);
matrPow(A5, A1, 5);

```

```

// Displaying A
cout << "A^1: \n";
matr2DDisplay(A1);
cout << "A^2: \n";
matr2DDisplay(A2);
cout << "A^3: \n";
matr2DDisplay(A3);
cout << "A^4: \n";
matr2DDisplay(A4);
cout << "A^5: \n";
matr2DDisplay(A5);
cout << "\n";

```

```

//Calculatin Sn
S1 = diagonalSum(S1, A1);
S2 = diagonalSum(S2, A2);
S3 = diagonalSum(S3, A3);
S4 = diagonalSum(S4, A4);
S5 = diagonalSum(S5, A5);

```

```

// Displaying Sn
cout << "S_1: "; cout << S1 << "\n";
cout << "S_2: "; cout << S2 << "\n";
cout << "S_3: "; cout << S3 << "\n";
cout << "S_4: "; cout << S4 << "\n";
cout << "S_5: "; cout << S5 << "\n\n";

```

```

// Calculating P
P1 = S1;
P2 = (S2 - P1 * S1) * (0.5);
P3 = (S3 - P1 * S2 - P2 * S1) * (0.3333333333);
P4 = (S4 - P1 * S3 - P2 * S2 - P3 * S1) * (0.25);
P5 = (S5 - P1 * S4 - P2 * S3 - P3 * S2 - P4 * S1) * (0.2);

```

```

//Displaying P
cout << "P_1: " << P1 << "\n";
cout << "P_2: " << P2 << "\n";
cout << "P_3: " << P3 << "\n";
cout << "P_4: " << P4 << "\n";
cout << "P_5: " << P5 << "\n\n";

```

```

// [WA] D(A) = ((-1)^5)*(x^5-(-8.99)*x^4-(29.31*x^3)-(249.26*x^2)-(-
470.49*x)-(-660.32))

```

```

// => x = {-8.00609, -7.02325, -0.973873, 3.01897, 3.99424)

```

```

return 0;

```

```

}

```

```

2. Фадеев
#include <iostream>
#include <iomanip>
#define n 5
using namespace std;

void matrPow(double An[n][n], double Ak[n][n], int t) {
    double temp[n][n];

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            temp[i][j] = Ak[i][j];
        }
    }
    while (t > 1) {

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                An[i][j] = 0;
                for (int k = 0; k < n; k++)
                    An[i][j] += temp[i][k] * Ak[k][j];
            }
        }
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                temp[i][j] = An[i][j];
            }
        }
        t--;
    }
}

void matr2DDisplay(double An[n][n]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << setw(10) << An[i][j];
        }
        cout << "\n";
    }
}

double diagonalSum(double Sn, double An[n][n]) {
    for (int i = 0; i < n; i++) {
        Sn += An[i][i];
    }
    return Sn;
}

void calcB(double B1[n][n], double B2[n][n], double P, double res[n][n]) {
    double unit[n][n], Brackets[n][n];

    for (int i = 0; i < n; i++) { // Calculating identity matrix
        for (int j = 0; j < n; j++) {
            unit[i][j] = 0;

            if (i == j)
                unit[i][j] = 1;
        }
    }

    // Brackets open
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            Brackets[i][j] = B2[i][j] - (P * unit[i][j]);
        }
    }
}

```

```

    }
    // Brackets close

    // Multiply
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            res[i][j] = 0;
            for (int k = 0; k < n; k++)
                res[i][j] += B1[i][k] * Brackets[k][j];
        }
    }
}

void calcAinv(double Bn[n][n], double Pk, double Pn, double res[n][n]) {
    double unit[n][n], Brackets[n][n];

    for (int i = 0; i < n; i++) { // Calculating identity matrix
        for (int j = 0; j < n; j++) {
            unit[i][j] = 0;

            if (i == j)
                unit[i][j] = 1;
        }
    }

    // Brackets open
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            Brackets[i][j] = Bn[i][j] - (Pn * unit[i][j]);
        }
    }

    matr2DDisplay(Brackets);
    // Brackets close

    // Multiply
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            res[i][j] = 0;
            res[i][j] += (1/Pk) * (Brackets[i][j]);
        }
    }
}

int main()
{
    cout << setprecision(4) << fixed;
    double B1[n][n], B2[n][n], B3[n][n], B4[n][n], B5[n][n], Ainv[n][n],
E[n][n],
        Tr1 = 0, Tr2 = 0, Tr3 = 0, Tr4 = 0, Tr5 = 0,
        P1 = 0, P2 = 0, P3 = 0, P4 = 0, P5 = 0;

    double A[n][n] = { {0.369080808,      1.436031013, -0.728668525,
0.990262713, -1.066533038},
{9.321890071, -1.513583362, -3.530025844, -
5.518609715,      1.735434769,},
{0.53279339, -2.264422783, -4.01783643, -
5.59560245, -3.516869824, },
{1.624855009,      1.624346269, -2.438991881,
2.43852384, 2.741387029,},
{-7.857573106, -1.199934881, -0.823975906,
0.389003073, -6.276184856} };

    // Displaying A
    cout << "A^1: \n"; matr2DDisplay(A);
}

```

```

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        B1[i][j] = A[i][j];
    }
}

//Calculatin Pn
cout << "B1: \n"; matr2DDisplay(B1);
P1 = diagonalSum(Tr1, B1);

calcB(B1, B1, P1, B2);
cout << "B2: \n"; matr2DDisplay(B2);
P2 = 0.5 * diagonalSum(Tr2, B2);

calcB(B1, B2, P2, B3);
cout << "B3: \n"; matr2DDisplay(B3);
P3 = 0.33 * diagonalSum(Tr3, B3);

calcB(B1, B3, P3, B4);
cout << "B4: \n"; matr2DDisplay(B4);
P4 = 0.25 * diagonalSum(Tr4, B4);

calcB(B1, B4, P4, B5);
cout << "B5: \n"; matr2DDisplay(B5);
P5 = 0.2 * diagonalSum(Tr5, B5);

//Displaying P
cout << "\nP_1: " << P1 << "\n";
cout << "P_2: " << P2 << "\n";
cout << "P_3: " << P3 << "\n";
cout << "P_4: " << P4 << "\n";
cout << "P_5: " << P5 << "\n\n";

// Calculating the inverse matrix A(-1)
calcAinv(B4, P5, P4, Ainv);

cout << "Ainv: \n";
matr2DDisplay(Ainv);

cout << "PROOF: AA(-1) = E\n";
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        E[i][j] = 0;
        for (int k = 0; k < n; k++)
            E[i][j] += A[i][k] * Ainv[k][j];
    }
}
cout << "E: \n";
matr2DDisplay(E);
// [WA] D(A) = ((-1)5*(x5-(-9)*x4-(29*x3)-(246.51*x2)-(-465.60*x)-(-612.8625))

// => x = {-8.05768, -6.96634, -0.925153, 2.95372, 3.99545)
// 8.797981808 & 1.436031013 & -0.728668525 & 0.990262713 & -1.066533038 \
// 9.321890071 & 7.914316638 & -3.530025844 & -5.518609715 & 1.735434769 \
// 0.53279339 & -2.264422783 & 3.41006357 & -5.59560245 & -3.516869824 \
// 1.624855009 & 1.624346269 & -2.438991881 & 10.86642384 & 2.741387029 \
// -7.857573106 & -1.199934881 & -0.823975906 & 0.389003073 & 2.150715144
double x[5] = { -8.05768, -6.96634, -0.925153, 2.95372, 3.99545 };

return 0;
}

```

3. Данилевский

```

#include <iostream>
#include <iomanip>
#include "matrix.h"
#include "polynom.h"

using namespace std;

int main()
{
    cout << setprecision(5) << fixed;

    int n = 5;
    matrix A;

    double m[5][5] = { {0.37, 1.44, -0.7, 0.99, -1.1 },
                        { 9.32, -1.5, -3.5, -5.5, 1.74 },
                        { 0.53, -2.3, -4, -5.6, -3.5 },
                        { 1.62, 1.62, -2.4, 2.44, 2.74 },
                        { -7.9, -1.2, -0.8, 0.39, -
6.3 } };

    matrix B(m);
    cout << "Matrix : \n";
    cout << B;
    A = B;

    matrix S(n);
    matrix F = A.toFrobenius(S);
    cout << "Frobenius matrix: \n";
    cout << F;
    polynom P(n + 1);
    polynom Q(2);
    P.setPolynom(F.getPolynom());

    cout << "Characteristic polynomial: \n";
    cout << P;
    cout << "E: " << endl;
    double* roots = new double[n];
    for (int i = 0; i < n; i++) {
        roots[i] = P.solution(0.0, 1e-9);
        cout << 'x' << i + 1 << " = " << roots[i] << "\n";
        Q.setPolynom(1, -roots[i]);
        P = P.dividing(Q);
    }

    delete[] roots;

    return 0;
}

```

```

4. Простые итерации
// Метод простых итераций
#include <iostream>
#include <iomanip>
#define n 5
using namespace std;

int main()
{
    float A[n][n] = { {0.369080808,      1.436031013, -0.728668525,
                      0.990262713, -1.066533038},
                      {9.321890071, -1.513583362, -3.530025844, -5.518609715,
                      1.735434769},
                      {0.53279339, -2.264422783, -4.01783643, -5.59560245, -
3.516869824, },
                      {1.624855009,      1.624346269, -2.438991881,
                      2.43852384, 2.741387029},
                      {-7.857573106, -1.199934881, -0.823975906,
                      0.389003073, -6.276184856} };

    float Xi[n], X[n], summ = 0, Xinorm[n], e, a, a0, eps = 0.0001;
    int i, j, k, count = 0;

    cout << setprecision(4) << fixed;

    Xi[0] = 1;
    for (i = 1; i < n; i++)
        Xi[i] = 0;
    do
    {
        for (i = 0; i < n; i++)
            summ += pow(Xi[i], 2);
        a0 = sqrt(summ);
        for (i = 0; i < n; i++)
            Xinorm[i] = Xi[i] / a0;
        for (i = 0; i < n; i++)
        {
            X[i] = 0;
            for (j = 0; j < n; j++)
                X[i] += A[i][j] * Xinorm[j]; //  $y^{(k+1)} = AX^{(k)}$ 
        }
        summ = 0;
        for (i = 0; i < n; i++)
            summ += pow(X[i], 2);
        a = sqrt(summ);
        e = abs(a - a0);
        for (i = 0; i < n; i++)
            Xi[i] = X[i];
        summ = 0;

        // Display
        cout << "\nEigen Value = " << a;
        cout << "\tEigen Vector: [";
        for (i = 0; i < n; i++)
        {
            cout << Xinorm[i] << "\t";
        }
        cout << "]\n";
        count++;
    } while (e > eps);
    cout << "\nTotal iterations: " << count;
    return 0;
}

```

```

5. Прямые итерации
// Метод прямых итераций
#include<iostream>
#include<iomanip>
#include<stdio.h>

#define SIZE 5

using namespace std;

int main()
{
    float a[SIZE][SIZE] = { {0.369080808,      1.436031013, -0.728668525,
    0.990262713, -1.066533038},
    {9.321890071, -1.513583362, -3.530025844, -
5.518609715,      1.735434769,},
    {0.53279339, -2.264422783, -
4.01783643, -5.59560245, -3.516869824, },
    {1.624855009,      1.624346269, -
2.438991881,      2.43852384, 2.741387029,},
    {-7.857573106, -1.199934881, -
0.823975906,      0.389003073, -6.276184856} }, x[SIZE], x_new[SIZE];
    float temp, lambda_new, lambda_old, error;
    int i, j, n, step = 1;

    cout << setprecision(4) << fixed;

    n = SIZE;

    error = 0.0001;

    for (i = 0; i < n; i++)
    {
        x[i] = 1;
    }

    lambda_old = 1;

up:
    for (i = 0; i < n; i++)
    {
        temp = 0.0;
        for (j = 0; j < n; j++)
        {
            temp += a[i][j] * x[j];
        }
        x_new[i] = temp;
    }

    for (i = 0; i < n; i++)
    {
        x[i] = x_new[i];
    }

    // Finding largest value from x
    lambda_new = abs(x[1]);
    for (i = 0; i < n; i++)
    {
        if (abs(x[i]) > lambda_new)
        {
            lambda_new = abs(x[i]);
        }
    }
}

```

```

// Normalization
for (i = 0; i < n; i++)
{
    x[i] /= lambda_new;
}

// Display
cout << "\nEigen Value = " << lambda_new;
cout << "\tEigen Vector: [";
for (i = 0; i < n; i++)
{
    cout << x[i] << "\t";
}
cout << "]\n";

if (abs(lambda_new - lambda_old) >= error)
{
    lambda_old = lambda_new;
    step++;
    goto up;
}
cout << "\nTotal iterations: " << step;
return(0);
}

```



```

6. Обратные итерации
#include <iostream>
#include <cmath>

using namespace std;

const int N = 5; // matrix dimension
const double eps = 0.0001; // accuracy

void print_matrix(double A[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            cout << A[i][j] << "\t";
        }
        cout << endl;
    }
}

void inverse_iteration(double A[N][N], double& lambda)
{
    double x[N], x_new[N], y[N], y_new[N];
    double norm = 0, lambda_new, delta = 1;
    int iter = 0;

    // initial guess
    for (int i = 0; i < N; i++)
    {
        x[i] = 1.0;
    }

    while (delta > eps)
    {
        iter++;
        // solve linear system
        for (int i = 0; i < N; i++)
        {
            y[i] = 0.0;
            for (int j = 0; j < N; j++)
            {
                y[i] += A[i][j] * x[j];
            }
        }

        // find the norm of y
        norm = 0.0;
        for (int i = 0; i < N; i++)
        {
            norm += y[i] * y[i];
        }
        norm = sqrt(norm);

        // normalize y
        for (int i = 0; i < N; i++)
        {
            y[i] /= norm;
        }

        // calculate new eigenvalue
        lambda_new = 0.0;
        for (int i = 0; i < N; i++)
        {
            lambda_new += x[i] * y[i];
        }
    }
}

```

```

        // check for convergence
        delta = fabs(lambda_new - lambda);
        lambda = lambda_new;

        // update x
        for (int i = 0; i < N; i++)
        {
            x[i] = y[i];
        }
    }

    cout << "Number of iterations: " << iter << endl;
}

int main()
{
    double A[N][N] = { {0.37,      1.44, -0.7, 0.99, -1.1 },
                        { 9.32, -1.5, -3.5, -5.5, 1.74 },
                        { 0.53, -2.3, -4,   -5.6, -3.5 },
                        { 1.62,      1.62, -2.4, 2.44, 2.74 },
                        { -7.9, -1.2, -0.8,      0.39, -6.3 } };

    double lambda = 0.0;

    cout << "Matrix A:\n";
    print_matrix(A);

    inverse_iteration(A, lambda);

    cout << "The minimum eigenvalue is: " << lambda << endl;

    return 0;
}

```

7. Хаусхолдер

```
import numpy as np
```

```

def qr_eig(A, eps=1e-4, max_iter=100):
    n = A.shape[0]
    I = np.identity(n)
    V = I
    for i in range(max_iter):
        Q, R = np.linalg.qr(A)
        A = R @ Q
        V = V @ Q
        if np.max(np.abs(A - np.diag(np.diag(A)))) < eps:
            break
    return np.diag(A), V

# matrix A
A = np.array([[0.37, 1.44, -0.7, 0.99, -1.1],
              [9.32, -1.5, -3.5, -5.5, 1.74],
              [0.53, -2.3, -4, -5.6, -3.5],
              [1.62, 1.62, -2.4, 2.44, 2.74],
              [-7.9, -1.2, -0.8, 0.39, -6.3]])

eigvals, eigvecs = qr_eig(A)
print("Eigenvalues:", eigvals)

```