Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторные работы по курсу «Численные методы»

Студент:	М. Д. Жилин
Преподаватель:	Д. Е. Пивоваров
Группа:	М8О-303Б-21
Дата:	
Оценка:	
Подпись:	

Москва, 2024

1.1 LU - разложение матриц

1 Постановка задачи

Реализовать алгоритм LU - разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.

Вариант: 9

$$\begin{cases} -2x_1 - 9x_2 - 3x_3 + 7x_4 = -26 \\ -7x_1 + 8x_2 + 2x_3 + 5x_4 = -25 \\ -6x_1 + 2x_2 = -16 \\ -3x_2 + 8x_3 - 3x_4 = -5 \end{cases}$$

2 Результаты работы

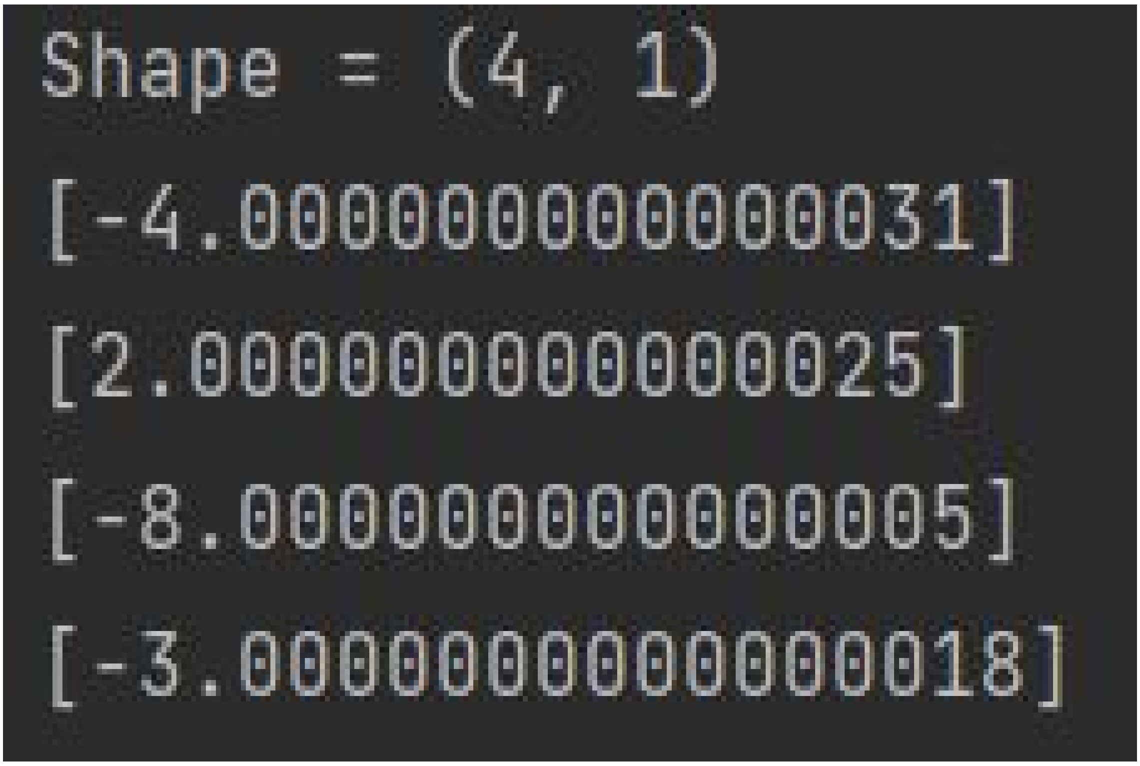

Рис. 1: Вывод программы в консоли

3 Исходный код

```python
class Vector:
    def __init__(self, values: list[None | int | float]):
        self.values = [value for value in values]
        self.n = len(values)

    def __str__(self) -> str:
        return self.values.__str__()

    def copy(self):
        return Vector([value for value in self.values])

    def __len__(self) -> int:
        return self.values.__len__()

    def __getitem__(self, index: int):
        return self.values[index]

    def __setitem__(self, index: int, value: int | float):
        self.values[index] = value

    def upload_values(self, values: list[list[None | int | float]]):
        self.values = [value for value in values]


class Matrix:
    def __init__(self, values: list[list[None | int | float] | Vector]):
        if type(values[0]) == Vector:
            self.values = [vector.copy() for vector in values]
        else:
            self.values = [Vector(row) for row in values]
        self.n: int = len(self.values)
        self.m: int = len(self.values[0])

    def __str__(self) -> str:
        m = '\n'.join([row.__str__() for row in self.values])
        return f"Shape = {self.get_shape()} \n{m}"

    def get_shape(self) -> tuple[int, int]:
        return self.n, self.m

    def copy(self):
        return Matrix([vect.copy() for vect in self.values])

    def upload_values(self, values: list[list[None | int | float]]):
        self.values = [[val for val in row] for row in values]

    def transposed(self):
```

```
        res = [[None] * self.n for _ in range(self.m)]
        for i in range(self.n):
            for j in range(self.m):
                res[j][i] = self.values[i][j]
        return Matrix(res)

    def __setitem__(self, index: int, value: Vector):
        self.values[index] = value

    def __getitem__(self, index: int):
        return self.values[index]


def multiple_matrix(matrix1: Matrix, matrix2: Matrix) -> None | Matrix:
    n1, m1 = matrix1.get_shape()
    n2, m2 = matrix2.get_shape()
    if m1 != n2: # TODO: implement custom Error for matrices
        print("Incorrect shapes of matrices")
        return
    res: list[list[None | int | float]] = [[None]*m2 for _ in range(n1)]
    n, m, h = n1, m2, m1
    del n1, n2, m1, m2
    for i in range(n):
        for j in range(m):
            cntr = 0
            for k in range(h):
                cntr += matrix1.values[i][k] * matrix2.values[k][j]
            res[i][j] = cntr

    return Matrix(res)


def plus_matrix(matrix1: Matrix, matrix2: Matrix) -> Matrix:
    res = [[matrix1[i][j]+matrix2[i][j] for j in range(matrix1.m)] for i in range(matrix1.n)]
    return Matrix(res)
```

```
from matrix import *


def lu_decomposition(coefficients: Matrix, results: Matrix | None = None) -> tuple[Matrix, Matrix]:
    # coefficients and results might be changed in outer code
    L = Matrix([[0] * coefficients.m for _ in range(coefficients.n)])
    U = Matrix([[value for value in row] for row in coefficients.values])

    # Straight Gaussian stroke
    for k in range(coefficients.n):
        if U[k][k] == 0:
            for i in range(k+1, coefficients.n):
```

```
                if U[i][k] != 0:
                    U[k], U[i] = U[i], U[k]
                    L[k], L[i] = L[i], L[k]
                    coefficients[k], coefficients[i] = coefficients[i], coefficients[k]
                    if results:
                        results[k], results[i] = results[i], results[k]
                    break
            else:
                print("There are not solutions") # TODO: create custom Exception
                raise Exception
        L[k][k] = 1
        for i in range(k+1, coefficients.n):
            L[i][k] = U[i][k]/U[k][k]
            if U[i][k] == 0:
                continue
            for j in range(k, coefficients.m):
                U[i][j] -= L[i][k]*U[k][j]

    return L, U


def get_determinant(coefficients: Matrix) -> int | float:
    _, U = lu_decomposition(coefficients)
    det = 1
    for i in range(coefficients.n):
        det *= U[i][i]
    return det


def calculate_decisions(coefficients: Matrix, results: Matrix) -> Matrix:
    L, U = lu_decomposition(coefficients, results)
    res = results.copy()
    for k in range(res.m):
        for i in range(res.n):
            for j in range(i):
                res[i][k] -= res[j][k]*L[i][j]

    for k in range(res.m):
        for i in range(coefficients.n-1, -1, -1):
            for j in range(i+1, results.n):
                res[i][k] -= res[j][k]*U[i][j]
            res[i][k] /= U[i][i]
    return res


def get_inverse_matrix(matrix: Matrix) -> Matrix:
    E = Matrix([[1 if i == j else 0 for j in range(matrix.n)]for i in range(matrix.n)])
    return calculate_decisions(matrix, E)
```

```python

if __name__ == "__main__":
    coefficient_matrix = Matrix([
        [-7, -9, 1, -9],
        [-6, -8, -5, 2],
        [-3, 6, 5, -9],
        [-2, 0, -5, -9]
    ])

    equation_roots = Matrix([
        [29],
        [42],
        [11],
        [75]
    ])

    print(calculate_decisions(coefficient_matrix, equation_roots))
```

1.2 Метод прогонки

4 Постановка задачи

Реализовать метод прогонки в виде программы, задавая в качестве входных данных ненулевые элементы матрицы системы и вектор правых частей. Используя разработанное программное обеспечение, решить СЛАУ с трехдиагональной матрицей.

Вариант: 9

$$\begin{cases} -12x_1 - 7x_2 = -102 \\ -7x_1 - 11x_2 - 3x_3 = -92 \\ -7x_2 + 21x_3 - 8x_4 = -65 \\ 4x_3 - 13x_4 + 5x_5 = 38 \\ -6x_4 + 14x_5 = -12 \end{cases}$$

5 Результаты работы

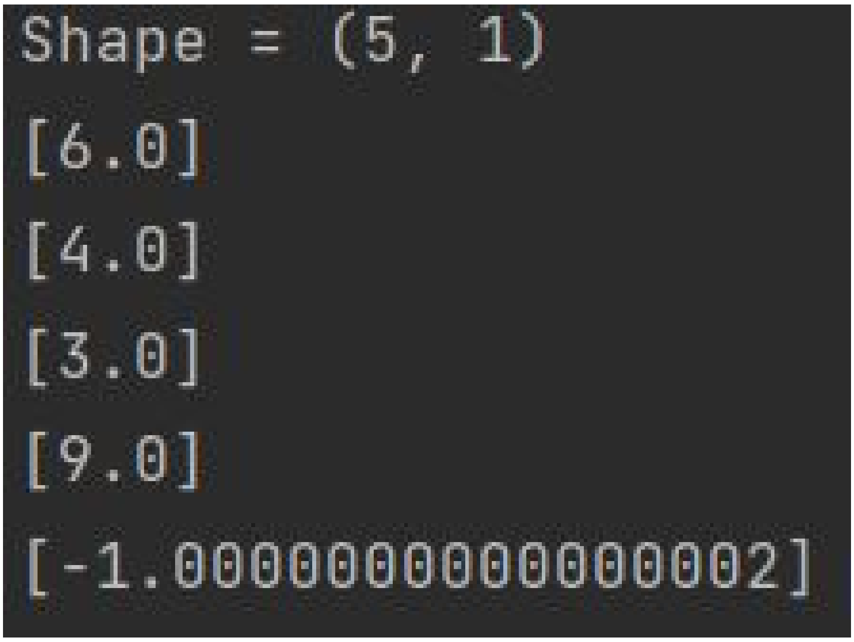

Рис. 2: Вывод программы в консоли

6 Исходный код

```
#include <bits/stdc++.h>

using namespace std;
using matrix = vector<vector<double> >;


matrix tridiagonal_algorithm(matrix& coefficients, matrix& results) {
    double a, b, c, d;
    a = 0;
    b = coefficients[0][0];
    c = coefficients[0][1];
    d = results[0][0];
    vector<double> P(coefficients[0].size(), 0), Q(coefficients[0].size(), 0);

    P[0] = -c/b;
    Q[0] = d/b;
    for (int i=1; i < coefficients.size() - 1; i++){
        a = coefficients[i][i-1];
        b = coefficients[i][i];
        c = coefficients[i][i+1];
        d = results[i][0];

        P[i] = -c/(b + a*P[i-1]);
        Q[i] = (d - a*Q[i-1])/(b + a*P[i-1]);
    }

    a = coefficients[coefficients.size()-1][coefficients[0].size()-2];
    b = coefficients[coefficients.size()-1][coefficients[0].size()-1];
    c = 0;
    d = results[results.size()-1][0];

    Q[Q.size()-1] = (d - a * Q[Q.size()-2]) / (b + a * P[P.size()-2]);

    matrix decision(results.size());
    for(int i=0; i<decision.size(); i++)
        decision[i].push_back(0);

    decision[decision.size()-1][0] = Q[Q.size()-1];
    for (int i = decision.size()-2; i > -1; i--)
        decision[i][0] = P[i]*decision[i+1][0] + Q[i];

    return decision;
}


void print_matrix(matrix& matrix1) {
    for(const auto& vect: matrix1) {
```

```
        for (auto x: vect)
            cout << x << " ";
        cout << endl;
    }
}

int main() {
    matrix coefficient_matrix{
        {8, -4, 0, 0, 0},
        {-2, 12, -7, 0, 0},
        {0, 2, -9, 1, 0},
        {0, 0, -8, 17, -4},
        {0, 0, 0, -7, 13}
    };

    matrix equation_roots = {
        {32},
        {15},
        {-10},
        {133},
        {-76}
    };

    matrix res_matrix = tridiagonal_algorithm(coefficient_matrix, equation_roots);

    cout << "Desicion" << endl;
    for (int i=0; i<res_matrix.size(); i++)
        printf("x%d = %f \n", i+1, res_matrix[i][0]);

    return 0;
}
```

```
from matrix import *


def tridiagonal_algorithm(coefficients: Matrix, results: Matrix) -> Matrix:
    if coefficients.n < 3 or coefficients.m < 3:
        print("Incorrect shapes of matrices")
        raise Exception
    a, b, c, d = 0, coefficients[0][0], coefficients[0][1], results.values[0][0]
    P, Q = [0]*coefficients.m, [0]*coefficients.m

    P[0], Q[0] = -c/b, d/b
    for i in range(1, coefficients.n-1):
        a, b, c, d = coefficients[i][i-1], coefficients[i][i], coefficients[i][i+1], results[i][0]
        P[i] = -c/(b + a*P[i-1])
        Q[i] = (d - a*Q[i-1])/(b + a*P[i-1])
    a, b, c, d = coefficients[-1][-2], coefficients[-1][-1], 0, results[-1][0]
    Q[-1] = (d - a * Q[-2]) / (b + a * P[-2])
```

```python

    decisions = [0]*results.n
    decisions[-1] = Q[-1]
    for i in range(decisions.__len__()-2, -1, -1):
        decisions[i] = P[i]*decisions[i+1] + Q[i]

    return Matrix([[i] for i in decisions])


if __name__ == "__main__":
    coefficient_matrix = Matrix([
        [8, -4, 0, 0, 0],
        [-2, 12, -7, 0, 0],
        [0, 2, -9, 1, 0],
        [0, 0, -8, 17, -4],
        [0, 0, 0, -7, 13]
    ])

    equation_roots = Matrix([
        [32],
        [15],
        [-10],
        [133],
        [-76]
    ])

    print(tridiagonal_algorithm(coefficient_matrix, equation_roots))
```

1.3 Метод простых итераций. Метод Зейделя

7 Постановка задачи

Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности.

Вариант: 9

$$\begin{cases} 18x_1 - 2x_3 + 7x_4 = 50 \\ -x_1 + 14x_2 - 3x_3 + 2x_4 = 2 \\ 5x_1 + 5x_2 + 26x_3 + 7x_4 = 273 \\ -2x_1 - 6x_2 + 9x_3 + 24x_4 = 111 \end{cases}$$

8 Результаты работы

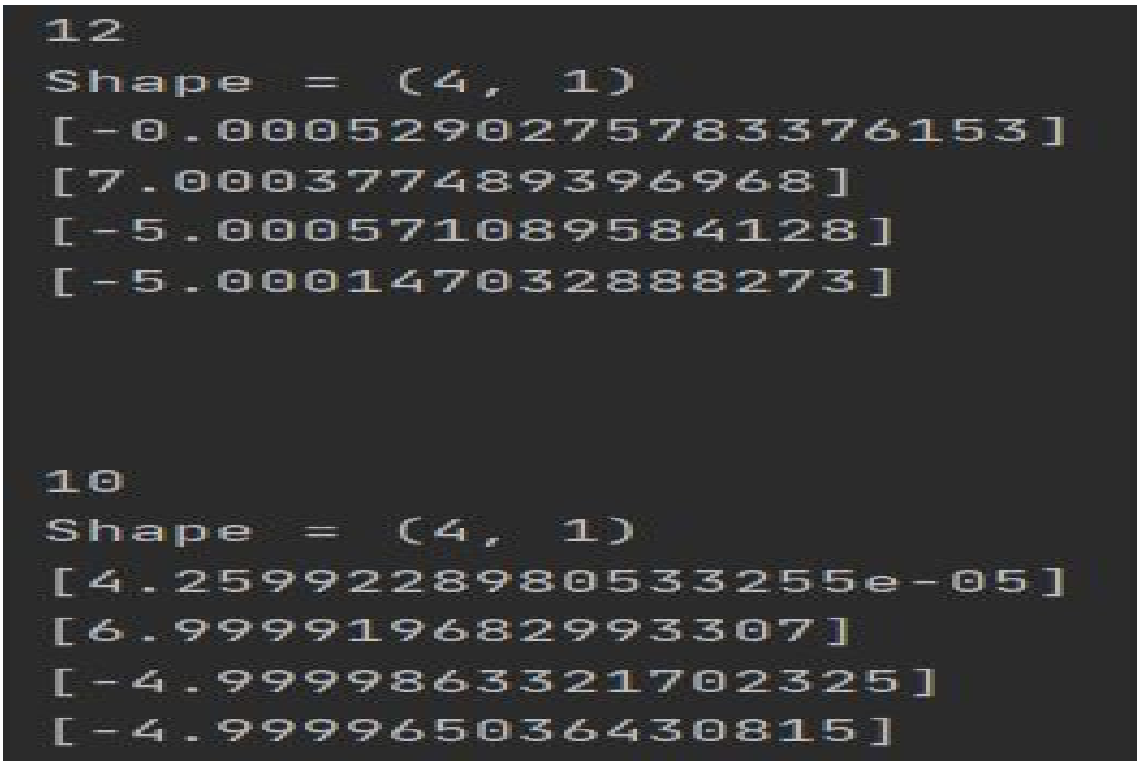

Рис. 3: Вывод программы в консоли

9 Исходный код

```
from matrix import *


def preprocessing(coefficients: Matrix, results: Matrix) -> None:
    n, m = coefficients.get_shape()
    for i in range(n):
        a = coefficients[i][i]
        if a == 0:
            continue
        results[i][0] /= a
        for j in range(m):
            if i == j:
                coefficients[i][j] = 0
            else:
                coefficients[i][j] /= -a


def get_current_eps(vect1: Matrix, vect2: Matrix) -> float:
    eps = 0
    for i in range(vect1.get_shape()[0]):
        eps += (vect1[i][0] - vect2[i][0]) ** 2
    return eps ** 0.5


def simple_iterations(a: Matrix, b: Matrix, EPS0: float) -> tuple[int, Matrix]:
    x_previous = Matrix(b.values)
    k = 0
    while get_current_eps(x_current := plus_matrix(b, multiple_matrix(a, x_previous)), x_previous) > EPS0:
        k += 1
        x_current, x_previous = None, x_current
    return k, x_previous


def seidel_method(a: Matrix, b: Matrix, EPS0: float) -> tuple[int, Matrix]:
    x_previous = Matrix(b.values)
    x_current = x_previous.copy()
    flag = True
    k, eps = 0, 0
    while flag or eps > EPS0:
        k += 1
        flag = False
        for i in range(b.get_shape()[0]):
            x_current[i][0] = 0
            for j in range(b.get_shape()[0]):
                if i == j:
                    x_current[i][0] += b[i][0]
```

```
                elif i < j:
                    x_current[i][0] += a[i][j]*x_previous[j][0]
                else:
                    x_current[i][0] += a[i][j]*x_current[j][0]
        eps = get_current_eps(x_current, x_previous)
        x_current, x_previous = x_current, x_current.copy()
    return k, x_previous


if __name__ == "__main__":
    coefficient_matrix = Matrix([
        [12, -3, -1, 3],
        [5, 20, 9, 1],
        [6, -3, -21, -7],
        [8, -7, 3, -27]
    ])

    equation_roots = Matrix([
        [-31],
        [90],
        [119],
        [71]
    ])

    preprocessing(coefficient_matrix, equation_roots)

    last_iteration, res_matrix = simple_iterations(coefficient_matrix, equation_roots, 0.001)
    print(last_iteration)
    print(res_matrix)

    print('\n\n')

    last_iteration, res_matrix = seidel_method(coefficient_matrix, equation_roots, 0.001)
    print(last_iteration)
    print(res_matrix)
```

```
#include <bits/stdc++.h>

using namespace std;
using matrix = vector<vector<double> >;


matrix multiply_matrix(const matrix& matrix1, const matrix& matrix2) {
    int n1 = matrix1.size();
    int m1 = matrix1[0].size();
    int n2 = matrix2.size();
    int m2 = matrix2[0].size();

```

```
    if (m1 != n2) {
        cout << "Incorrect shapes of matrices" << endl;
        return matrix();
    }

    matrix res(n1, vector<double>(m2, 0));

    for (int i = 0; i < n1; ++i) {
        for (int j = 0; j < m2; ++j) {
            double cntr = 0;
            for (int k = 0; k < m1; ++k) {
                cntr += matrix1[i][k] * matrix2[k][j];
            }
            res[i][j] = cntr;
        }
    }

    return res;
}


matrix plus_matrix(const matrix& matrix1, const matrix& matrix2) {
    int n = matrix1.size();
    int m = matrix1[0].size();

    matrix res(n, vector<double>(m));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            res[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }

    return res;
}

matrix minus_matrix(const matrix& matrix1, const matrix& matrix2) {
    int n = matrix1.size();
    int m = matrix1[0].size();

    matrix res(n, vector<double>(m));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            res[i][j] = matrix1[i][j] - matrix2[i][j];
        }
    }

    return res;
}

```

```

void preprocessing(matrix& coefficients, matrix& results) {
    double n = coefficients.size(), m = coefficients[0].size();
    for(int i=0; i < n; i++) {
        double a = coefficients[i][i];
        if (a == 0)
            continue;
        results[i][0] /= a;
        for (int j=0; j < m; j++)
            if (i == j)
                coefficients[i][j] = 0;
            else
                coefficients[i][j] /= -a;
    }
}


double get_current_eps(const matrix& vect1, const matrix& vect2) {
    double eps = 0;
    for(int i=0; i<vect1.size(); i++)
        eps += pow(vect1[i][0] - vect2[i][0], 2);
    return sqrt(eps);
}


void print_matrix(const matrix& matrix1) {
    for(const auto& vect: matrix1) {
        cout << '\t';
        for (auto x: vect)
            cout << x << " ";
        cout << endl;
    }
}


pair<int, matrix> simple_iterations(matrix& a, matrix& b, double EPS0) {
    matrix x_previous = b;
    matrix x_current;
    int k = 0;
    while (get_current_eps(x_current = plus_matrix(b, multiply_matrix(a, x_previous)),
        x_previous) > EPS0) {
        k += 1;
        x_previous = x_current;
    }
    return make_pair(k, x_previous);
}


pair<int, matrix> seidel_method(const matrix& a, const matrix& b, double EPS0) {
```

```
    matrix x_previous = b;
    matrix x_current = x_previous;
    bool flag = true;
    int k = 0;
    double eps = 0;
    while (flag or eps > EPS0) {
        k += 1;
        flag = false;
        for(int i = 0; i < b.size(); i++) {
            x_current[i][0] = 0;
            for(int j=0; j < b.size(); j++) {
                if (i == j)
                    x_current[i][0] += b[i][0];
                else if (i < j)
                    x_current[i][0] += a[i][j]*x_previous[j][0];
                else
                    x_current[i][0] += a[i][j]*x_current[j][0];
            }
        }
        eps = get_current_eps(x_current, x_previous);
        x_previous = x_current;
    }
    return make_pair(k, x_previous);
}


int main() {
    matrix coefficient_matrix = {
        {12, -3, -1, 3},
        {5, 20, 9, 1},
        {6, -3, -21, -7},
        {8, -7, 3, -27}
    };

    matrix equation_roots = {
        {-31},
        {90},
        {119},
        {71}
    };

    preprocessing(coefficient_matrix, equation_roots);

    int last_iteration;
    matrix res_matrix;

    tie(last_iteration, res_matrix) = simple_iterations(coefficient_matrix,
        equation_roots, 0.001);
    cout << "Simple iterations method:" << endl;
```

```
    cout << "\tNumber of iterations = " << last_iteration << endl;
    cout << "\tSolution=" << endl;
    print_matrix(res_matrix);
    cout << endl << endl;



    tie(last_iteration, res_matrix) = seidel_method(coefficient_matrix, equation_roots,
         0.001);
    cout << "Seidel's method:" << endl;
    cout << "\tNumber of iterations = " << last_iteration << endl;
    cout << "\tSolution = " << endl;
    print_matrix(res_matrix);

    return 0;
}
```

1.4 Метод вращений

10 Постановка задачи

Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.

Вариант: 9

$$\begin{pmatrix} 4 & 7 & -1 \\ 7 & -9 & -6 \\ -1 & -6 & -4 \end{pmatrix}$$

11 Результаты работы

```
Собственные значения:
λ_1 = 8.071918520499581
λ_2 = -14.623607766264337
λ_3 = -2.4483107542352407

Собственные векторы:
x_1 = [0.8440417011154279, 0.449064021683711, -0.2931469106223472]
x_2 = [-0.2930937368771562, 0.8440481813931533, 0.44908654944263804]
x_3 = [0.4490987287685852, -0.2931282516513245, 0.8440297150588233]
```

Рис. 4: Вывод программы в консоли

12 Исходный код

```
import math

from matrix import *
from matrix import Vector, Matrix


def get_indexes(matrix1: Matrix) -> tuple[int, int]:
    """
    return indexes of maximum non-diagonal element
    """
    m: float = matrix1[0][1]
    indexes: tuple[int, int] = (0, 1)
    for i in range(matrix1.n):
        for j in range(i+1, matrix1.m):
            if abs(matrix1[i][j]) > m:
                m = abs(matrix1[i][j])
                indexes = (i, j)
    return indexes


def get_phi(a_ij: float, a_ii: float, a_jj: float) -> float:
    if a_ii == a_jj:
        return math.pi/4
    return math.atan(2*a_ij/(a_ii-a_jj)) / 2


def get_U_matrix(matrix1: Matrix) -> Matrix:
    res = [[1 if i == j else 0 for j in range(matrix1.n)] for i in range(matrix1.n)]
    i_max, j_max = get_indexes(matrix1)
    phi = get_phi(a_ij=matrix1[i_max][j_max], a_ii=matrix1[i_max][i_max], a_jj=matrix1[
        j_max][j_max])
    res[i_max][i_max] = math.cos(phi)
    res[j_max][j_max] = math.cos(phi)
    res[i_max][j_max] = -math.sin(phi)
    res[j_max][i_max] = math.sin(phi)
    return Matrix(res)


def get_eps(matrix1: Matrix) -> float:
    eps = 0
    for i in range(matrix1.n):
        for j in range(i+1, matrix1.m):
            eps += matrix1[i][j]**2
    return math.sqrt(eps)


def Jacobi_method(coeff_matrix: Matrix, EPS: float) -> tuple[Vector, Matrix]:
```

```python
    eigenvectors = Matrix([[1 if i == j else 0 for j in range(coeff_matrix.n)]for i in range(coeff_matrix.n)])
    while get_eps(coeff_matrix) > EPS:
        U = get_U_matrix(coeff_matrix)
        eigenvectors = multiple_matrix(eigenvectors, U)
        coeff_matrix = multiple_matrix(multiple_matrix(U.transposed(), coeff_matrix), U)
    eigenvalues = Vector([coeff_matrix[i][i] for i in range(coeff_matrix.n)])
    return eigenvalues, eigenvectors


if __name__ == "__main__":
    coefficient_matrix = Matrix([
        [4, 7, -1],
        [7, -9, -6],
        [-1, -6, -4]
    ])

    values, vectors = Jacobi_method(coefficient_matrix, EPS=0.01)
    print(" :", *[f'_{i+1} = {values[i]}' for i in range(values.n)], sep='\n', end='\n\n')

    print(" :")
    for i in range(vectors.n):
        print(f'x_{i+1} = {[vectors[j][i] for j in range(vectors.n)]}')
```

1.5 QR – разложение матриц

13 Постановка задачи

Реализовать алгоритм QR – разложения матриц в виде программы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений. С использованием разработанного программного обеспечения найти собственные значения матрицы.

Вариант: 26

$$\begin{pmatrix} -5 & -8 & 4 \\ 4 & 2 & 6 \\ -2 & 5 & -6 \end{pmatrix}$$

14 Результаты работы

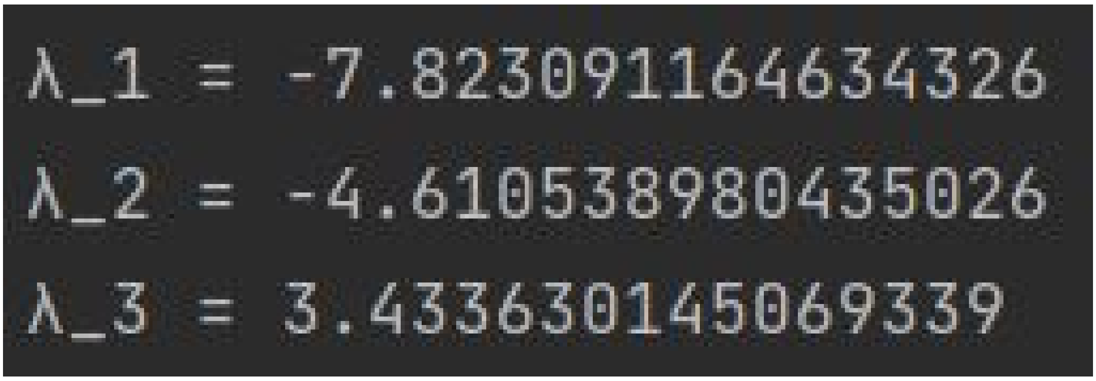

Рис. 5: Вывод программы в консоли

15 Исходный код

```
#include <bits/stdc++.h>

using namespace std;
using matrix = vector<vector<double> >;


void print_matrix(const matrix& matrix1) {
    for(const auto& vect: matrix1) {
        for (auto x: vect)
            cout << x << " ";
        cout << endl;
    }
}


matrix plus_matrix(const matrix& matrix1, const matrix& matrix2) {
    matrix res;
    int n = matrix1.size();
    for (int i = 0; i < n; i++) {
        vector<double> row;
        for (int j = 0; j < n; j++) {
            row.push_back(matrix1[i][j] + matrix2[i][j]);
        }
        res.push_back(row);
    }

    return res;
}


matrix transposed(const matrix& matrix1){
    int n = matrix1.size(), m = matrix1[0].size();
    matrix res(m, vector<double>(n));
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            res[j][i] = matrix1[i][j];
    return res;
}

matrix multiple_matrix(matrix& matrix1, matrix& matrix2) {
    int n1 = matrix1.size(), m1 = matrix1[0].size(), m2 = matrix2[0].size();
    matrix res(n1);
    for (int i=0; i<n1; i++)
        for (int j=0; j<m2; j++)
            res[i].push_back(0);

    for (int i=0; i<n1; i++) {
```

```
        for (int j=0; j<m2; j++) {
            double cntr = 0;
            for (int k=0; k<m1; k++)
               cntr += matrix1[i][k] * matrix2[k][j];
            res[i][j] = cntr;
        }
    }
    return res;
}


int sign(double a){
    return (a >= 0) ? 1 : -1;
}


double get_eps(const matrix& matrix1){
    double eps = 0;
    int n = matrix1.size();
    for(int i=0; i<n; i++)
        for(int j=0; j<i-1; j++)
            eps += matrix1[i][j]*matrix1[i][j];
    return sqrt(eps);
}


matrix get_E_matrix(int n){
    matrix E(n, vector<double>(n, 0));
    for(int i=0; i<n; i++)
        E[i][i] = 1;
    return E;
}


matrix get_H_matrix(const matrix& coefficients, int ind){
    int n = coefficients.size();
    matrix v(n, vector<double>(1));

    for(int i=0; i<n; i++){
        if (i < ind)
            v[i][0] = 0;
        else if (i == ind){
            double sum = 0;
            for (int j=ind; j < n; j++)
                sum += coefficients[j][i]*coefficients[j][i];
            v[i][0] = coefficients[i][i] + sign(coefficients[i][i]) * sqrt(sum);
        }
        else
            v[i][0] = coefficients[i][ind];
```

```
    }

    matrix transposed_v = transposed(v);
    double k = -multiple_matrix(transposed_v, v)[0][0]/2;
    v = multiple_matrix(v, transposed_v);
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            v[i][j] /= k;

    matrix E = get_E_matrix(n);
    return plus_matrix(E, v);
}


pair<matrix, matrix> QR_decomposition(const matrix& coeff){
    matrix coefficients = coeff;
    matrix Q = get_H_matrix(coefficients, 0);
    coefficients = multiple_matrix(Q, coefficients);
    int n = coefficients.size();
    for (int i=1; i<n-1; i++){
        matrix H = get_H_matrix(coefficients, i);
        Q = multiple_matrix(Q, H);
        coefficients = multiple_matrix(H, coefficients);
    }
    return make_pair(Q, coefficients);
}


vector<double> get_eigenvalues(matrix& coefficients, double EPS){
    while (get_eps(coefficients) > EPS){
        pair<matrix, matrix> QR = QR_decomposition(coefficients);
        coefficients = multiple_matrix(QR.second, QR.first);
    }
    int n = coefficients.size();
    vector<double> result(n);
    for (int i=0; i<n; i++)
        result[i] = coefficients[i][i];
    return result;
}


int main() {
    matrix coefficient_matrix{
        {-5, -8, 4},
        {4, 2, 6},
        {-2, 5, -6}
    };


```

```
    vector<double> eigenvalues = get_eigenvalues(coefficient_matrix, 0.01);
    int n = eigenvalues.size();

    for (int i=0; i<n; i++)
        cout << "lambda_" << i+1 << " = " << eigenvalues[i] << endl;

    return 0;
}
```

```
from matrix import *
from matrix import Vector, Matrix


coefficient_matrix: Matrix


def get_eps(matrix1: Matrix) -> float:
    eps = 0
    for i in range(matrix1.n):
        for j in range(i-1):
            eps += matrix1[i][j]**2
    return eps**0.5


def sign(a: float) -> int:
    return 1 if a >= 0 else -1


def get_H_matrix(coefficients: Matrix, ind: int) -> Matrix:
    E = Matrix([[1 if i == j else 0 for j in range(coefficients.n)] for i in range(
        coefficients.n)])
    v = []
    for i in range(coefficients.n):
        if i < ind:
            v.append([0])
        elif i == ind:
            v.append([coefficients[i][i] + sign(coefficients[i][i]) * (sum([
                coefficients[j][i]**2 for j in range(ind, coefficients.n)])**0.5)])
        else:
            v.append([coefficients[i][ind]])
    v = Matrix(v)
    k = -multiple_matrix(v.transposed(), v)[0][0]/2
    V = multiple_matrix(v, v.transposed())
    for i in range(V.n):
        for j in range(V.n):
            V[i][j] /= k
    H = plus_matrix(E, V)
    return H


```

```python
def QR_decomposition(coefficients: Matrix) -> tuple[Matrix, Matrix]:
    coefficients = coefficients.copy()
    Q = get_H_matrix(coefficients, 0)
    coefficients = multiple_matrix(Q, coefficients)
    for i in range(1, coefficients.n-1):
        H = get_H_matrix(coefficients, i)
        Q = multiple_matrix(Q, get_H_matrix(coefficients, i))
        coefficients = multiple_matrix(H, coefficients)
    return Q, coefficients


def get_eigenvalues(coefficients: Matrix, EPS: float) -> list[float]:
    while get_eps(coefficients) > EPS:
        Q, R = QR_decomposition(coefficients)
        coefficients = multiple_matrix(R, Q)
    return [coefficients[i][i] for i in range(coefficients.n)]


if __name__ == "__main__":
    coefficient_matrix = Matrix([
        [-5, -8, 4],
        [4, 2, 6],
        [-2, 5, -6]
    ])

    eigenvalues = get_eigenvalues(coefficient_matrix, 0.01)

    print(*[f'_{i+1} = {eigenvalues[i]}' for i in range(eigenvalues.__len__())], sep='\n')
```