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:	И. Б. Белов
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: 2

$$\begin{cases} -2x_1 + 7x_2 - 8x_3 + 6x_4 = -39 \\ 4x_1 + 4x_2 - 7x_4 = 41 \\ -x_1 - 3x_2 + 6x_3 + 3x_4 = 4 \\ 9x_1 - 7x_2 - 2x_3 - 8x_4 = 113 \end{cases}$$

2 Результаты работы

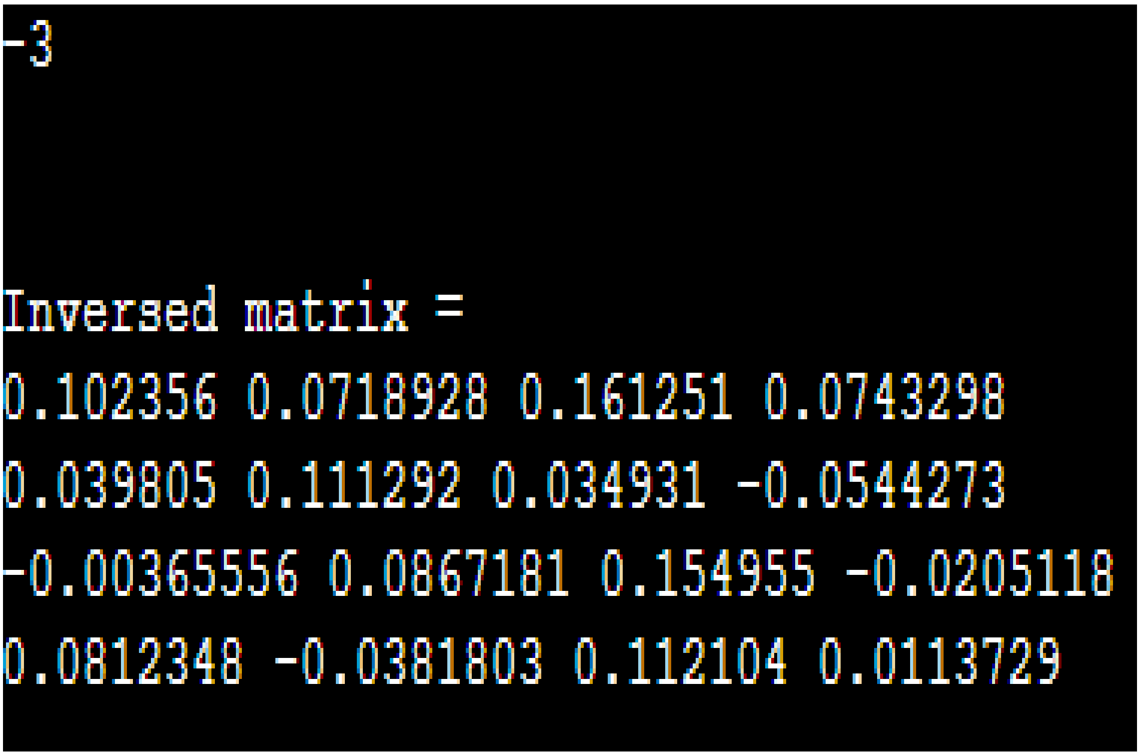

Рис. 1: Вывод программы в консоли

3 Исходный код

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

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



pair<matrix, matrix> lu_decomposition(matrix& coefficients, matrix& results) {
    int n1=coefficients.size(), m1=coefficients[0].size(), m2 = results[0].size();
    matrix L(n1), U = coefficients;
    for (int i=0; i<n1; i++)
        for (int j=0; j<m1; j++)
            L[i].push_back(0);

    for (int k=0; k<n1; k++) {
        if (U[k][k] == 0) {
            for (int i=k+1; i<n1; i++) {
                if (U[i][k] != 0) {
                    swap(U[k], U[i]);
                    swap(L[k], L[i]);
                    swap(coefficients[k], coefficients[i]);
                    swap(results[k], results[i]);
                    break;
                }
            }
        }
        L[k][k] = 1;
        for (int i=k+1; i<n1; i++) {
            L[i][k] = U[i][k]/U[k][k];
            if (U[i][k] == 0)
                continue;
            for(int j=k; j<m1; j++)
                U[i][j] -= L[i][k]*U[k][j];

        }
    }

    return make_pair(L, U);
}


double get_determinant(matrix& coefficients, matrix& results) {
    auto [_, U] = lu_decomposition(coefficients, results);
    double det = 1;
    for (int i=0; i<coefficients.size(); i++)
        det *= U[i][i];
    return det;
```

```
}


matrix calculate_decisions(matrix& coefficients, matrix& results) {
    auto [L, U] = lu_decomposition(coefficients, results);
    matrix res = results;

    for (int k=0; k<res[0].size(); k++)
        for (int i=0; i<res.size(); i++)
            for (int j=0; j<i; j++)
                res[i][k] -= res[j][k]*L[i][j];
    for (int k=0; k<res[0].size(); k++) {
        for (int i=coefficients.size()-1; i>-1; i--) {
            for (int j=i+1; j<results.size(); j++) {
                res[i][k] -= res[j][k]*U[i][j];
            }
            res[i][k] /= U[i][i];
        }
    }

    return res;
}


matrix get_inverse_matrix(matrix& matrix1) {
    matrix E(matrix1.size());
    for (int i=0; i<matrix1.size(); i++)
        for (int j=0; j<matrix1.size(); j++)
            E[i].push_back((i == j) ? 1 : 0);
    return calculate_decisions(matrix1, E);
}

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

int main() {
    matrix coefficient_matrix{
            {2, 7, -8, 6},
            {4, 4, 0, -7},
            {-1, -3, 6, 3},
            {9, -7, -2, -8}
    };

    matrix equation_roots = {
```

```
            {-39},
            {41},
            {4},
            {113}
    };

    auto [l, u] = lu_decomposition(coefficient_matrix, equation_roots);

    cout << endl << "L =" << endl;
    print_matrix(l);

    cout << endl << endl;
    cout << "U =" << endl;
    print_matrix(u);

    cout << endl << endl;
    cout << "det = " << get_determinant(coefficient_matrix, equation_roots) << endl;

    cout << endl << endl << "Decisions =" << endl;
    matrix decisions = calculate_decisions(coefficient_matrix, equation_roots);
    print_matrix(decisions);

    cout << endl << endl << "Inversed matrix =" << endl;
    matrix inversed = get_inverse_matrix(coefficient_matrix);
    print_matrix(inversed);
    return 0;
}
```

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: 2

$$\begin{cases} 10x_1 + 5x_2 = -120 \\ 3x_1 + 10x_2 - 2x_3 = -91 \\ 2x_2 - 9x_3 - 5x_4 = 5 \\ 5x_3 - 16x_4 - 4x_5 = -74 \\ -8x_4 + 16x_5 = -56 \end{cases}$$

5 Результаты работы

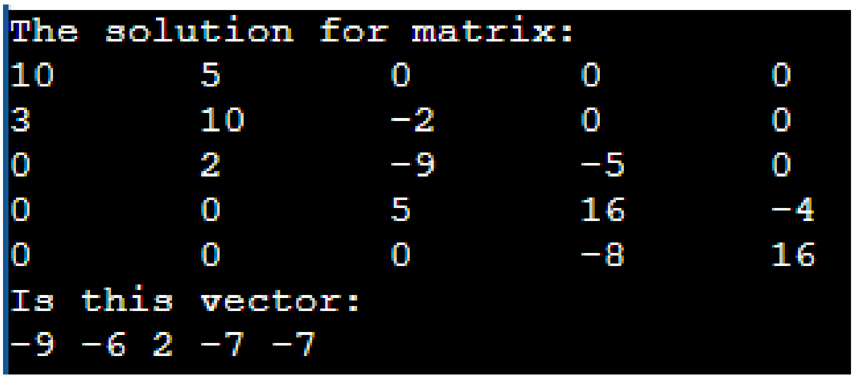

Рис. 2: Вывод программы в консоли

6 Исходный код

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
#include <cmath>

class NotANeededMatrix : public std::exception {
public:
    const char* what() const noexcept override {
        return "Sorry this matrix doesn't fit the requirements!\n";
    }
};

double find_det(const std::vector<std::vector<double>>& matrix) {
    double det = 1;
    for (size_t i = 0; i < matrix.size(); ++i) {
        det *= matrix[i][i];
    }
    return det;
}

void check_coefficients(double a = 0, double b = 0, double c = 0) {
    return;
}

std::vector<double> tridiagonal_solution(const std::vector<std::vector<double>>&
    matrix, const std::vector<double>& vector_b) {
    check_coefficients(matrix[0][0], matrix[0][1]);
    std::vector<double> vector_alphas = {-matrix[0][1] / matrix[0][0]};
    std::vector<double> vector_betas = {vector_b[0] / matrix[0][0]};

    for (size_t i = 1; i < matrix.size() - 1; ++i) {
        check_coefficients(matrix[i][i - 1], matrix[i][i], matrix[i][i + 1]);
        double y_i = matrix[i][i] + matrix[i][i - 1] * vector_alphas[i - 1];
        double a_i = -matrix[i][i + 1] / y_i;
        double b_i = (vector_b[i] - matrix[i][i - 1] * vector_betas[i - 1]) / y_i;
        vector_alphas.push_back(a_i);
        vector_betas.push_back(b_i);
    }

    check_coefficients(matrix.back()[matrix.back().size() - 2], matrix.back().back());
    double y_n = (matrix.back().back() + matrix.back()[matrix.back().size() - 2] *
        vector_alphas.back());
    std::vector<double> vector_x = {round((vector_b.back() - matrix.back()[matrix.back
        ().size() - 2] * vector_betas.back()) / y_n)};

    for (int i = static_cast<int>(matrix.size()) - 2; i >= 0; --i) {
```

```
        vector_x.insert(vector_x.begin(), round(vector_alphas[i] * vector_x[0] +
            vector_betas[i]));
    }

    return vector_x;
}

int main() {

    std::vector<std::vector<double>> matrix = {
            {10.0, 5.0, 0, 0, 0},
            {3.0, 10.0, -2.0, 0.0, 0},
            {0, 2, -9, -5, 0},
            {0, 0, 5, 16, -4},
            {0, 0, 0, -8, 16}
    };

    std::vector<double> vector = {-120, -91, 5, -74, -56};
    find_det(matrix);
    auto solution = tridiagonal_solution(matrix, vector);
    std::cout << "The solution for matrix:\n";
    for (const auto& row : matrix) {
        for (double val : row) {
            std::cout << val << "\t";
        }
        std::cout << std::endl;
    }
    std::cout << "Is this vector:\n";
    for (double val : solution) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
}
```

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: 2

$$\begin{cases} 24x_1 + 2x_2 + 4x_3 - 9x_4 = -9 \\ -6x_1 - 27x_2 - 8x_3 - 6x_4 = -76 \\ -4x_1 + 8x_2 + 19x_3 + 6x_4 = -79 \\ 4x_1 + 5x_2 - 3x_3 - 13x_4 = -70 \end{cases}$$

8 Результаты работы

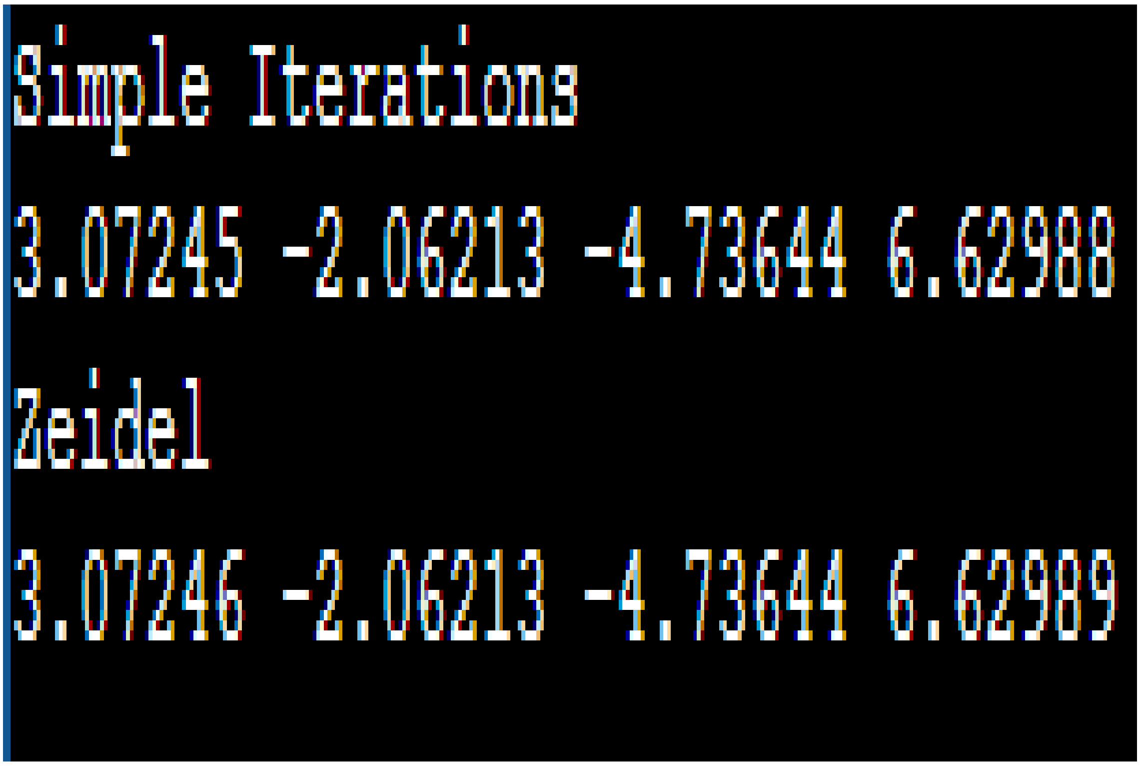

Рис. 3: Вывод программы в консоли

9 Исходный код

```cpp
#include <fstream>
#include <sstream>
#include <iostream>
#include <vector>
#include <cmath>

std::vector<std::vector<double>> deepcopy(const std::vector<std::vector<double>>&
    matrix) {
    std::vector<std::vector<double>> result(matrix.size(), std::vector<double>(matrix
        [0].size()));
    for (size_t i = 0; i < matrix.size(); ++i) {
        for (size_t j = 0; j < matrix[i].size(); ++j) {
            result[i][j] = matrix[i][j];
        }
    }
    return result;
}



std::pair<std::vector<std::vector<double>>, std::vector<double>> read_matrix_from_file
    (const std::string& filename) {
    std::ifstream file(filename);
    std::vector<std::vector<double>> matrix;
    std::vector<double> vector;
    std::string line;
    while (std::getline(file, line)) {
        if (!line.empty()) {
            std::istringstream iss(line);
            std::vector<double> row;
            double value;
            while (iss >> value) {
                row.push_back(value);
            }
            if (vector.empty()) {
                vector = row;
            } else {
                matrix.push_back(row);
            }
        }
    }
    return std::make_pair(matrix, vector);
}

double matrix_norm1(const std::vector<std::vector<double>>& matrix) {
    double max_sum = 0.0;
    for (const auto& row : matrix) {
```

```
        double row_sum = 0.0;
        for (double val : row) {
            row_sum += std::abs(val);
        }
        if (row_sum > max_sum) {
            max_sum = row_sum;
        }
    }
    return max_sum;
}


double dot_product(const std::vector<double>& a, const std::vector<double>& b) {
    double result = 0.0;
    for (size_t i = 0; i < a.size(); ++i) {
        result += a[i] * b[i];
    }
    return result;
}

std::vector<double> subtract_vectors(const std::vector<double>& a, const std::vector<
    double>& b) {
    std::vector<double> result(a.size());
    for (size_t i = 0; i < a.size(); ++i) {
        result[i] = a[i] - b[i];
    }
    return result;
}

double norm(const std::vector<double>& vector) {
    double max_val = vector[0];
    for (size_t i = 1; i < vector.size(); ++i) {
        if (vector[i] > max_val) {
            max_val = vector[i];
        }
    }
    return max_val;
}

std::pair<std::vector<std::vector<double>>, std::vector<double>> normal_view(const std
    ::vector<std::vector<double>>& matrix, const std::vector<double>& vector) {
    std::vector<std::vector<double>> res = matrix;
    std::vector<double> res_v = vector;
    for (size_t i = 0; i < res.size(); ++i) {
        double delim = res[i][i];
        for (size_t j = 0; j < res.size(); ++j) {
            res[i][j] /= -delim;
        }
        res_v[i] /= delim;
```

```
        res[i][i] = 0;
    }
    return std::make_pair(res, res_v);
}

std::vector<double> sum_vectors(const std::vector<double>& vect1, const std::vector<double>& vect2) {
    std::vector<double> result(vect1.size());
    for (size_t i = 0; i < vect1.size(); ++i) {
        result[i] = vect1[i] + vect2[i];
    }
    return result;
}

std::vector<double> prod_matrix(const std::vector<std::vector<double>>& a, const std::vector<double>& vector) {
    std::vector<double> result(a.size());
    for (size_t i = 0; i < a.size(); ++i) {
        double sum = 0.0;
        for (size_t j = 0; j < a[i].size(); ++j) {
            sum += a[i][j] * vector[j];
        }
        result[i] = sum;
    }
    return result;
}

std::vector<double> gauss_seidel(const std::vector<std::vector<double>>& a, const std::vector<double>& b, double epsilon) {
    auto [a_norm, b_norm] = normal_view(a, b);
    double alpha_norm = matrix_norm1(a_norm);
    std::vector<double> x_start(a_norm.size(), 0);
    std::vector<double> x_new = b_norm;

    while (true) {
        if (alpha_norm / (1 - alpha_norm) * norm(subtract_vectors(x_new, x_start)) <= epsilon) {
            break;
        }
        x_start = x_new;
        for (size_t j = 0; j < a_norm.size(); ++j) {
            double x_res = 0;
            for (size_t l = 0; l < a_norm.size(); ++l) {
                x_res += x_new[l] * a_norm[j][l];
            }
            x_res += b_norm[j];
            x_new[j] = x_res;
        }
    }
```

```
    return x_new;
}

std::vector<double> simple_iteration(const std::vector<std::vector<double>>& a, const std::vector<double>& b, double epsilon) {
    auto [a_norm, b_norm] = normal_view(a, b);
    double alpha_norm = matrix_norm1(a_norm);
    std::vector<double> x_start(a_norm.size(), 0);
    size_t max_iters = 100000;
    std::vector<double> x_new = b_norm;

    for (size_t j = 0; j < max_iters; ++j) {
        if (alpha_norm / (1 - alpha_norm) * norm(subtract_vectors(x_new, x_start)) > epsilon) {
            x_start = x_new;
            x_new = sum_vectors(prod_matrix(a_norm, x_new), b_norm);
        } else {
            break;
        }
    }
    return x_new;
}
int main() {
    std::vector<std::vector<double>> matrix = {
            {24.0, 2.0, 4.0, -9.0},
            {-6.0, 27.0, -8.0, -6.0},
            {-4.0, 8.0, 19.0, 6.0},
            {4.0, 5.0, -3.0, -13.0}
    };

    //   1x4
    std::vector<double> vector = {-9.0, -76.0, -79.0, -70.0};
    double epsilon = 0.0001;
    auto result = simple_iteration(matrix, vector, epsilon);
    auto result2 = gauss_seidel(matrix, vector, epsilon);
    std::cout << "Simple Iterations\n";
    for (double element : result) {
        std::cout << element << " ";
    }
    std::cout << std::endl;
    std::cout << "Zeidel\n";
    for (double element : result2) {
        std::cout << element << " ";
    }

    std::cout << std::endl;
    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: 2

$$\begin{pmatrix} -9 & 7 & 5 \\ 7 & 8 & 9 \\ 5 & 9 & 8 \end{pmatrix}$$

11 Результаты работы

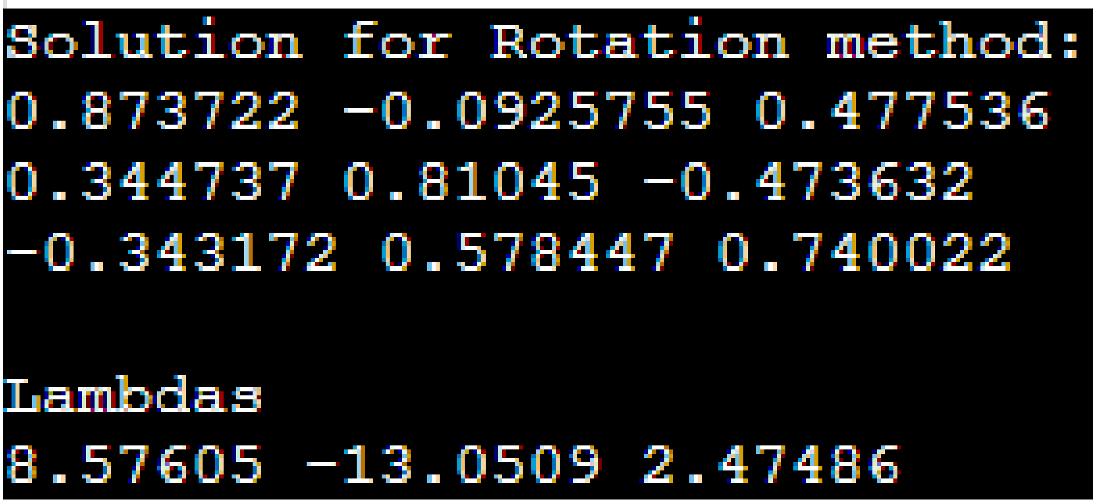

Рис. 4: Вывод программы в консоли

12 Исходный код

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>

std::vector<std::vector<double>> find_transport_matrix(const std::vector<std::vector<double>>& matrix) {
    std::vector<std::vector<double>> transport_matrix = matrix;
    for (size_t i = 0; i < matrix.size(); ++i) {
        for (size_t j = i + 1; j < matrix.size(); ++j) {
            std::swap(transport_matrix[i][j], transport_matrix[j][i]);
        }
    }
    return transport_matrix;
}

std::vector<std::vector<double>> make_e_matrix(int size) {
    std::vector<std::vector<double>> e_matrix(size, std::vector<double>(size, 0));
    for (int i = 0; i < size; ++i) {
        e_matrix[i][i] = 1.0;
    }
    return e_matrix;
}

double scholar_multiply(const std::vector<double>& vector_1, const std::vector<double>& vector_2) {
    double result = 0;
    for (size_t i = 0; i < vector_1.size(); ++i) {
        result += vector_1[i] * vector_2[i];
    }
    return result;
}


std::vector<std::vector<double>> multiply_matrix(const std::vector<std::vector<double>>& matrix_1, const std::vector<std::vector<double>>& matrix_2) {
    size_t n = matrix_1.size();
    size_t m = matrix_1[0].size();
    size_t p = matrix_2[0].size();
    if (m != matrix_2.size()) {
        throw std::invalid_argument("Matrix dimensions do not match.");
    }

    std::vector<std::vector<double>> result(n, std::vector<double>(p, 0));
    for (size_t i = 0; i < n; ++i) {
        for (size_t j = 0; j < p; ++j) {
            for (size_t k = 0; k < m; ++k) {
```

```
                result[i][j] += matrix_1[i][k] * matrix_2[k][j];
            }
        }
    }
    return result;
}


std::vector<int> find_max_non_diagonal(const std::vector<std::vector<double>>& matrix)
     {
    double max_val = matrix[0][1];
    std::vector<int> max_indices = {0, 1};
    for (size_t i = 0; i < matrix.size(); ++i) {
        for (size_t j = i + 1; j < matrix[i].size(); ++j) {
            if (std::abs(matrix[i][j]) > std::abs(max_val)) {
                max_val = matrix[i][j];
                max_indices = {static_cast<int>(i), static_cast<int>(j)};
            }
        }
    }
    return max_indices;
}

double find_phi(const std::vector<std::vector<double>>& matrix, int i, int j) {
    if (matrix[i][i] == matrix[j][j]) {
        return M_PI / 4;
    }
    return 0.5 * std::atan((2 * matrix[i][j]) / (matrix[i][i] - matrix[j][j]));
}

std::vector<std::vector<double>> make_rotation_matrix(double angle, size_t size, int
    i_, int j_) {
    std::vector<std::vector<double>> res = make_e_matrix(size);
    res[i_][j_] = -std::sin(angle);
    res[j_][i_] = std::sin(angle);
    res[i_][i_] = res[j_][j_] = std::cos(angle);
    return res;
}

double find_lim(const std::vector<std::vector<double>>& matrix) {
    double res = 0;
    for (size_t i = 0; i < matrix.size() - 1; ++i) {
        for (size_t j = i + 1; j < matrix[i].size(); ++j) {
            res += matrix[i][j] * matrix[i][j];
        }
    }
    return std::sqrt(res);
}

```

```cpp


std::pair<std::vector<std::vector<double>>, std::vector<double>> rotation_solution(
    const std::vector<std::vector<double>>& matrix, double epsilon) {
    std::vector<std::vector<double>> matrix_a = matrix;
    std::vector<std::vector<double>> matrix_res_u = make_e_matrix(matrix.size());
    while (std::abs(find_lim(matrix_a)) > epsilon) {
        std::vector<int> res_ij = find_max_non_diagonal(matrix_a);
        double phi = find_phi(matrix_a, res_ij[0], res_ij[1]);
        std::vector<std::vector<double>> matrix_u = make_rotation_matrix(phi, matrix_a.
            size(), res_ij[0], res_ij[1]);
        matrix_res_u = multiply_matrix(matrix_res_u, matrix_u);
        matrix_a = multiply_matrix(multiply_matrix(find_transport_matrix(matrix_u),
            matrix_a), matrix_u);
    }
    std::vector<std::vector<double>> result_vector;
    std::vector<double> lambdas(matrix_a.size());
    for (size_t i = 0; i < matrix_a.size(); ++i) {
        result_vector.push_back({matrix_res_u[i]});
        lambdas[i] = matrix_a[i][i];
    }

    return std::make_pair(matrix_res_u, lambdas);
}

int main() {
    std::vector<std::vector<double>> matrix = {
            {7.0, 3.0, -1.0},
            {3.0, -7.0, -8.0},
            {-1.0, -8.0, -2.0}
    };


    auto rotation_solution_result = rotation_solution(matrix, 0.01);
    auto matrix_res_u = rotation_solution_result.first;
    auto lambdas = rotation_solution_result.second;

    std::cout << "Solution for Rotation method:\n";
    for (const auto& row : matrix_res_u) {
        for (double val : row) {
            std::cout << val << " ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
    std::cout << "Lambdas\n";
    for (auto const& var: lambdas) {
        std::cout << var << " ";
    }
```

```

    return 0;
}
```

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: 2

$$\begin{pmatrix} -6 & -4 & 0 \\ -7 & 6 & -7 \\ -2 & -6 & -7 \end{pmatrix}$$

14 Результаты работы

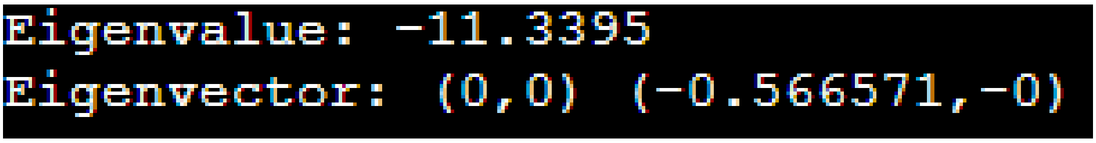

Рис. 5: Вывод программы в консоли

15 Исходный код

```cpp
#include <iostream>
#include <utility>
#include <vector>
#include <cmath>
#include <complex>

double row_col_mul(const std::vector<double>& row, const std::vector<double>& col) {
    double result = 0;
    for (size_t i = 0; i < row.size(); ++i) {
        result += row[i] * col[i];
    }
    return result;
}

std::pair<std::complex<double>, std::complex<double>> solve_equation(double a, double b, double c) {
    std::complex<double> delta = std::sqrt(std::complex<double>(b * b - 4 * a * c, 0));
    std::complex<double> root1 = (b + delta) / (2 * a);
    std::complex<double> root2 = (b - delta) / (2 * a);
    return std::make_pair(root1, root2);
}

std::vector<std::vector<double>> col_row_mul(const std::vector<double>& col, const std::vector<double>& row) {
    std::vector<std::vector<double>> result(col.size(), std::vector<double>(row.size(), 0));
    for (size_t i = 0; i < col.size(); ++i) {
        for (size_t j = 0; j < row.size(); ++j) {
            result[i][j] = col[i] * row[j];
        }
    }
    return result;
}

std::vector<std::vector<double>> matrix_matrix_mul(const std::vector<std::vector<double>>& matrix1, const std::vector<std::vector<double>>& matrix2) {
    std::vector<std::vector<double>> result(matrix1.size(), std::vector<double>(matrix2[0].size(), 0));
    for (size_t i = 0; i < matrix1.size(); ++i) {
        for (size_t j = 0; j < matrix2[0].size(); ++j) {
            for (size_t k = 0; k < matrix1[0].size(); ++k) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
    return result;
}
```

```
std::vector<std::vector<double>> get_e_matrix(size_t size) {
    std::vector<std::vector<double>> result(size, std::vector<double>(size, 0));
    for (size_t i = 0; i < size; ++i) {
        result[i][i] = 1;
    }
    return result;
}

int sign(double element) {
    if (element > 0) {
        return 1;
    } else if (element < 0) {
        return -1;
    }
    return 0;
}

double vector_second_norm_2(const std::vector<double>& vector) {
    double result = 0;
    for (double val : vector) {
        result += val * val;
    }
    return std::sqrt(result);
}

std::vector<std::vector<double>> get_h(const std::vector<double>& vector) {
    double lower = row_col_mul(vector, vector);
    auto upper = col_row_mul(vector, vector);
    auto E = get_e_matrix(upper.size());
    std::vector<std::vector<double>> second(upper.size(), std::vector<double>(upper.
        size(), 0));
    for (size_t i = 0; i < upper.size(); ++i) {
        for (size_t j = 0; j < upper.size(); ++j) {
            second[i][j] = -2 / lower * upper[i][j];
        }
    }
    std::vector<std::vector<double>> result(upper.size(), std::vector<double>(upper.
        size(), 0));
    for (size_t i = 0; i < upper.size(); ++i) {
        for (size_t j = 0; j < upper.size(); ++j) {
            result[i][j] = E[i][j] + second[i][j];
        }
    }
    return result;
}

std::pair<std::vector<std::vector<double>>, std::vector<std::vector<double>>> gen_qr(
    std::vector<std::vector<double>> matrix) {
```

```cpp
    auto A = matrix;
    std::vector<std::vector<double>> Q(matrix.size(), std::vector<double>(matrix.size()
        , 0));
    std::vector<std::vector<std::vector<double>>> H_all;
    std::vector<double> vector(matrix.size());
    for (size_t i = 0; i < matrix.size() - 1; ++i) {
        vector.clear();
        std::vector<double> row;
        for (const auto& col : A) {
            row.push_back(col[i]);
        }
        vector.insert(vector.end(), i, 0);
        vector.push_back(A[i][i] + sign(A[i][i]) * vector_second_norm_2(row));
        vector.insert(vector.end(), row.begin() + i + 1, row.end());
        auto H = get_h(vector);
        A = matrix_matrix_mul(H, A);
        H_all.push_back(H);
    }
    Q = H_all[0];
    for (size_t i = 1; i < H_all.size(); ++i) {
        Q = matrix_matrix_mul(Q, H_all[i]);
    }
    return std::make_pair(Q, A);
}

std::pair<double, std::pair<std::complex<double>, std::complex<double>>> qr_solve(std
    ::vector<std::vector<double>> matrix, double eps) {
    auto A = std::move(matrix);
    while (sqrt(A[1][0] * A[1][0] + A[2][0] * A[2][0]) > eps) {
        auto qr = gen_qr(A);
        A = matrix_matrix_mul(qr.second, qr.first);
    }
    auto roots = solve_equation(A[1][1], A[2][2], A[1][2] * A[2][1]);
    return std::make_pair(A[0][0], roots);
}

int main() {
    // Example usage
    std::vector<std::vector<double>> matrix = {{-6, -4, 0},
                                               {-7, 6, -7},
                                               {-2, -6, -7}};
    double epsilon = 0.0001;
    auto result = qr_solve(matrix, epsilon);
    std::cout << "Eigenvalue: " << result.first << std::endl;
    std::cout << "Eigenvector: ";
    std::cout << result.second.first << " " << result.second.second;
    std::cout << std::endl;
    return 0;
}
```