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:	Г. С. Будайчиев
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: 3

$$\begin{cases} 9x_1 - 5x_2 - 6x_3 + 3x_4 = -8 \\ x_1 - 7x_2 + x_3 = 38 \\ 3x_1 - 4x_2 + 9x_3 = 47 \\ 6x_1 - x_2 + 9x_3 + 8x_4 = -8 \end{cases}$$

2 Результаты работы

```
Solving system:
3.94746e-16
-5
3
-5

Determinant:
-4239
InverseMatrix:
0.111347      -0.149328     0.132578      -0.0417551
0.0113234     -0.167728     0.0304317     -0.00424628
-0.032083     -0.02477      0.0804435     0.0120311
-0.0460014    0.118896      -0.186129     0.142251
```

Рис. 1: Вывод программы в консоли

3 Исходный код

Общий файл для всех подзадач 1 лабораторной работы:

```
#ifndef MATRIX_H
#define MATRIX_H

#include <vector>
#include <map>
#include <utility>
#include <memory>
#include <cmath>
#include <string>
#include <fstream>
#include <complex>

namespace numeric {

    template<class T>
    class EigenResult {
    public:
        std::vector<T> eigenValues;
        std::vector<std::vector<T>> eigenVectors;

        explicit EigenResult(size_t size) : eigenValues(size), eigenVectors(size, std::
            vector<T>(size)) {}
    };

    template<class T, template<typename> class Container = std::vector>
    class AbstractMatrix {
    protected:
        size_t _rows;
        size_t _cols;
    public:
        virtual ~AbstractMatrix() = default;

        virtual size_t rows() const;

        virtual size_t cols() const;

        virtual Container<T>& operator[](size_t i) = 0;

        virtual const Container<T>& operator[](size_t i) const = 0;

    };

    template<class T, template<typename> class Container>
    size_t AbstractMatrix<T, Container>::rows() const {
        return _rows;
    }
```

```
template<class T, template<typename> class Container>
size_t AbstractMatrix<T, Container>::cols() const {
    return _cols;
}

template<class T>
class Matrix : public AbstractMatrix<T, std::vector> {
protected:
    std::vector<std::vector<T>> data;
    size_t _rows;
    size_t _cols;
public:
    explicit Matrix(const std::vector<std::vector<T>>& data);

    explicit Matrix(size_t rows, size_t cols);

    Matrix(const Matrix& other);

    Matrix<T>& operator=(const Matrix& other);

    Matrix<T>& operator+=(const Matrix<T>& rhs);

    Matrix<T>& operator-=(const Matrix<T>& rhs);

    Matrix<T>& operator*=(const Matrix<T>& rhs);

    size_t rows() const override;

    size_t cols() const override;

    std::vector<T>& operator[](size_t i);

    const std::vector<T>& operator[](size_t i) const;

    Matrix<T> transpose() const;

    ~Matrix() override;

    static Matrix<T> eye(size_t size);
};

template<class T>
class SquareMatrix : public AbstractMatrix<T, std::vector> {
private:
    std::vector<std::vector<T>> data;
    size_t size;
public:
    explicit SquareMatrix(const std::vector<std::vector<T>>& data);
```

```
    explicit SquareMatrix(size_t size);

    SquareMatrix(const SquareMatrix& other);

    SquareMatrix<T>& operator=(const SquareMatrix& other);

    SquareMatrix<T>& operator+=(const SquareMatrix<T>& rhs);

    SquareMatrix<T>& operator-=(const SquareMatrix<T>& rhs);

    SquareMatrix<T>& operator*=(const SquareMatrix<T>& rhs);

    size_t rows() const override;

    size_t cols() const override;

    std::vector<T>& operator[](size_t i);

    const std::vector<T>& operator[](size_t i) const;

    SquareMatrix<T> transpose() const;

    explicit operator Matrix<T>() const;

    ~SquareMatrix() override;

};

template<class T>
SquareMatrix<T>::operator Matrix<T>() const {
    Matrix<T> result(this->size, this->size);
    for (size_t i = 0; i < this->size; ++i) {
        for (size_t j = 0; j < this->size; ++j) {
            result[i][j] = this->data[i][j];
        }
    }
    return result;
}

template<class T>
SquareMatrix<T>& SquareMatrix<T>::operator=(const SquareMatrix& other) {
    if (this != &other) {
        data = other.data;
        size = other.size;
    }
    return *this;
}
```

```
template<class T>
SquareMatrix<T>::SquareMatrix(const SquareMatrix& other) : data(other.data), size(
    other.cols()) {

}

template<class T>
Matrix<T>& Matrix<T>::operator=(const Matrix& other) {
    if (this != &other) {
        data = other.data;
        _rows = other._rows;
        _cols = other._cols;
    }
    return *this;
}

template<class T>
Matrix<T>::Matrix(const Matrix& other) : data(other.data), _rows(other._rows),
    _cols(other._cols) {

}

template<typename T>
class Row {
private:
    std::map<size_t, T> row_data;

public:
    T& operator[](size_t col) {
        return row_data[col];
    }

    const T& operator[](size_t col) const {
        auto it = row_data.find(col);
        if (it != row_data.end()) {
            return it->second;
        }
        static const T defaultValue{};
        return defaultValue;
    }
};

template<class T>
class SparseMatrix : public AbstractMatrix<T, Row> {
private:
    size_t _rows;
    size_t _cols;

```

```
        std::map<size_t, Row<T>> data;


    public:
        explicit SparseMatrix(size_t rows, size_t cols);

        SparseMatrix(const SparseMatrix& other);

        SparseMatrix<T>& operator=(const SparseMatrix& other);

        SparseMatrix<T>& operator+=(const SparseMatrix<T>& rhs);

        SparseMatrix<T>& operator-=(const SparseMatrix<T>& rhs);

        SparseMatrix<T>& operator*=(const SparseMatrix<T>& rhs);

        size_t rows() const override;

        size_t cols() const override;

        Row<T>& operator[](size_t row) override;

        const Row<T>& operator[](size_t row) const override;

        SparseMatrix<T> transpose() const;

        ~SparseMatrix() override;
    };

    template<class T>
    Matrix<T>::Matrix(size_t rows, size_t cols) : data(rows, std::vector<T>(cols)), _rows(rows), _cols(cols) {

    }

    template<class T>
    Matrix<T>::Matrix(const std::vector<std::vector<T>>& data)
        : data(data), _rows(data.size()), _cols(data.begin()->size()) {
    }


    template<class T>
    size_t Matrix<T>::rows() const {
        return _rows;
    }

    template<class T>
    size_t Matrix<T>::cols() const {
        return _cols;
```

```
}

template<class T>
std::vector<T>& Matrix<T>::operator[](size_t i) {
    return data[i];
}

template<class T>
const std::vector<T>& Matrix<T>::operator[](size_t i) const {
    return data[i];
}

template<class T>
Matrix<T> Matrix<T>::transpose() const {
    Matrix<T> transposedMatrix(_cols, _rows);
    for (size_t i = 0; i < rows(); ++i) {
        for (size_t j = 0; j < cols(); ++j) {
            transposedMatrix[j][i] = (*this)[i][j];
        }
    }
    return transposedMatrix;
}

template<class T>
Matrix<T>::~Matrix() = default;

template<class T>
Matrix<T> Matrix<T>::eye(size_t size) {
    Matrix<T> matrix(size, size);
    for (int i = 0; i < size; ++i) {
        matrix[i][i] = 1;
    }
    return matrix;
}

template<class T>
Matrix<T> operator+(const Matrix<T>& lhs, const Matrix<T>& rhs) {
    if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols()) {
        throw std::invalid_argument("Matrix dimensions must match for addition.");
    }

    Matrix<T> result(lhs.rows(), lhs.cols());
    for (size_t i = 0; i < lhs.rows(); ++i) {
        for (size_t j = 0; j < lhs.cols(); ++j) {
            result[i][j] = lhs[i][j] + rhs[i][j];
        }
    }
    return result;
}
```

```

template<class T>
std::vector<T> operator*(const std::vector<T>& vec, T scalar) {
    std::vector<T> result(vec.size());
    for (size_t i = 0; i < vec.size(); ++i) {
        result[i] = vec[i] * scalar;
    }
    return result;
}

template<class T>
std::vector<T> operator*(T scalar, const std::vector<T>& vec) {
    return vec * scalar;
}

template<class T>
Matrix<T> operator-(const Matrix<T>& lhs, const Matrix<T>& rhs) {
    if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols()) {
        throw std::invalid_argument("Matrix dimensions must match for subtraction.");
    }

    Matrix<T> result(lhs.rows(), lhs.cols());
    for (size_t i = 0; i < lhs.rows(); ++i) {
        for (size_t j = 0; j < lhs.cols(); ++j) {
            result[i][j] = lhs[i][j] - rhs[i][j];
        }
    }
    return result;
}

template<class T>
Matrix<T> operator*(const Matrix<T>& lhs, const Matrix<T>& rhs) {
    if (lhs.cols() != rhs.rows()) {
        throw std::invalid_argument(
            "The number of columns in the first matrix must match the number of rows in the second.");
    }

    Matrix<T> result(lhs.rows(), rhs.cols());
    for (size_t i = 0; i < lhs.rows(); ++i) {
        for (size_t j = 0; j < rhs.cols(); ++j) {
            T sum = T();
            for (size_t k = 0; k < lhs.cols(); ++k) {
                sum += lhs[i][k] * rhs[k][j];
            }
            result[i][j] = sum;
        }
    }
```

```
        return result;
    }

    template<class T>
    Matrix<T> operator*(const Matrix<T>& matrix, T scalar) {
        Matrix<T> result(matrix.rows(), matrix.cols());
        for (size_t i = 0; i < matrix.rows(); ++i) {
            for (size_t j = 0; j < matrix.cols(); ++j) {
                result[i][j] = matrix[i][j] * scalar;
            }
        }
        return result;
    }

    template<class T>
    Matrix<T> operator*(T scalar, const Matrix<T>& matrix) {
        return matrix * scalar;
    }

    template<class T>
    Matrix<T> operator/(const Matrix<T>& matrix, T scalar) {
        if (scalar == 0) {
            throw std::invalid_argument("Zero division");
        }
        Matrix<T> result(matrix.rows(), matrix.cols());
        for (size_t i = 0; i < matrix.rows(); ++i) {
            for (size_t j = 0; j < matrix.cols(); ++j) {
                result[i][j] = matrix[i][j] / scalar;
            }
        }
        return result;
    }

    template<class T>
    std::vector<T> operator*(const Matrix<T>& matrix, const std::vector<T>& vec) {
        if (matrix.cols() != vec.size()) {
            throw std::invalid_argument("The number of columns in the matrix must match
                 the size of the vector.");
        }

        std::vector<T> result(matrix.rows(), T());
        for (size_t i = 0; i < matrix.rows(); ++i) {
            for (size_t k = 0; k < matrix.cols(); ++k) {
                result[i] += matrix[i][k] * vec[k];
            }
        }
        return result;
    }

```

```
template<class T>
std::vector<T> operator*(const std::vector<T>& vec, const Matrix<T>& matrix) {
    if (vec.size() != matrix.rows()) {
        throw std::invalid_argument("The size of the vector must match the number of rows in the matrix.");
    }

    std::vector<T> result(matrix.cols(), T());
    for (size_t j = 0; j < matrix.cols(); ++j) {
        for (size_t i = 0; i < matrix.rows(); ++i) {
            result[j] += vec[i] * matrix[i][j];
        }
    }
    return result;
}




template<class T>
Matrix<T>& Matrix<T>::operator+=(const Matrix<T>& rhs) {
    if (_rows != rhs._rows || _cols != rhs._cols) {
        throw std::invalid_argument("Matrix dimensions must match for addition.");
    }
    for (size_t i = 0; i < _rows; ++i) {
        for (size_t j = 0; j < _cols; ++j) {
            data[i][j] += rhs.data[i][j];
        }
    }
    return *this;
}

template<class T>
Matrix<T>& Matrix<T>::operator-=(const Matrix<T>& rhs) {
    if (_rows != rhs._rows || _cols != rhs._cols) {
        throw std::invalid_argument("Matrix dimensions must match for subtraction.");
    }
    for (size_t i = 0; i < _rows; ++i) {
        for (size_t j = 0; j < _cols; ++j) {
            data[i][j] -= rhs.data[i][j];
        }
    }
    return *this;
}

template<class T>
Matrix<T>& Matrix<T>::operator*=(const Matrix<T>& rhs) {
    if (_cols != rhs._rows) {
```

```
        throw std::invalid_argument(
            "The number of columns in the first matrix must match the number of rows in the second for multiplication.");
    }

    Matrix<T> result(_rows, rhs._cols);
    for (size_t i = 0; i < _rows; ++i) {
        for (size_t j = 0; j < rhs._cols; ++j) {
            for (size_t k = 0; k < _cols; ++k) {
                result.data[i][j] += data[i][k] * rhs.data[k][j];
            }
        }
    }

    *this = std::move(result);
    return *this;
}

template<class T>
SquareMatrix<T>::SquareMatrix(const std::vector<std::vector<T>>& data) : data(data), size(data.size()) {
    if (data.size() != data.begin()->size()) {
        throw std::invalid_argument("Not square matrix");
    }
}

template<class T>
SquareMatrix<T>::SquareMatrix(size_t size) : size(size), data(size, std::vector<T>(size)) {
}

template<class T>
size_t SquareMatrix<T>::rows() const {
    return size;
}

template<class T>
size_t SquareMatrix<T>::cols() const {
    return size;
}

template<class T>
std::vector<T>& SquareMatrix<T>::operator[](size_t i) {
    return data[i];
}

template<class T>
const std::vector<T>& SquareMatrix<T>::operator[](size_t i) const {
    return data[i];
```

```
    }

    template<class T>
    SquareMatrix<T> SquareMatrix<T>::transpose() const {
        SquareMatrix<T> transposedMatrix(size);
        for (size_t i = 0; i < rows(); ++i) {
            for (size_t j = 0; j < cols(); ++j) {
                transposedMatrix[j][i] = (*this)[i][j];
            }
        }
        return transposedMatrix;
    }

    template<class T>
    SquareMatrix<T>::~SquareMatrix() = default;

    template<class T>
    SquareMatrix<T> operator+(const SquareMatrix<T>& lhs, const SquareMatrix<T>& rhs) {
        if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols()) {
            throw std::invalid_argument("Matrix dimensions must match for addition.");
        }

        SquareMatrix<T> result(lhs.cols());
        for (size_t i = 0; i < lhs.rows(); ++i) {
            for (size_t j = 0; j < lhs.cols(); ++j) {
                result[i][j] = lhs[i][j] + rhs[i][j];
            }
        }
        return result;
    }

    template<class T>
    SquareMatrix<T> operator-(const SquareMatrix<T>& lhs, const SquareMatrix<T>& rhs) {
        if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols()) {
            throw std::invalid_argument("Matrix dimensions must match for subtraction."
                );
        }

        SquareMatrix<T> result(lhs.cols());
        for (size_t i = 0; i < lhs.rows(); ++i) {
            for (size_t j = 0; j < lhs.cols(); ++j) {
                result[i][j] = lhs[i][j] - rhs[i][j];
            }
        }
        return result;
    }

    template<class T>
    SquareMatrix<T> operator*(const SquareMatrix<T>& lhs, const SquareMatrix<T>& rhs) {
```

```
    if (lhs.cols() != rhs.rows()) {
        throw std::invalid_argument(
            "The number of columns in the first matrix must match the number of rows in the second.");
    }

    SquareMatrix<T> result(rhs.cols());
    for (size_t i = 0; i < lhs.rows(); ++i) {
        for (size_t j = 0; j < rhs.cols(); ++j) {
            T sum = T();
            for (size_t k = 0; k < lhs.cols(); ++k) {
                sum += lhs[i][k] * rhs[k][j];
            }
            result[i][j] = sum;
        }
    }
    return result;
}

template<class T>
SquareMatrix<T>& SquareMatrix<T>::operator+=(const SquareMatrix<T>& rhs) {
    if (size != rhs.size || size != rhs.size) {
        throw std::invalid_argument("Matrix dimensions must match for addition.");
    }
    for (size_t i = 0; i < size; ++i) {
        for (size_t j = 0; j < size; ++j) {
            data[i][j] += rhs[i][j];
        }
    }
    return *this;
}

template<class T>
SquareMatrix<T>& SquareMatrix<T>::operator-=(const SquareMatrix<T>& rhs) {
    if (size != rhs.size || size != rhs.size) {
        throw std::invalid_argument("Matrix dimensions must match for subtraction.");
    }
    for (size_t i = 0; i < size; ++i) {
        for (size_t j = 0; j < size; ++j) {
            data[i][j] -= rhs[i][j];
        }
    }
    return *this;
}

template<class T>
SquareMatrix<T>& SquareMatrix<T>::operator*=(const SquareMatrix<T>& rhs) {
    if (size != rhs.size) {
```

```
            throw std::invalid_argument(
                "The number of columns in the first matrix must match the number of rows
                     in the second for multiplication.");
        }

        SquareMatrix<T> result(size);
        for (size_t i = 0; i < size; ++i) {
            for (size_t j = 0; j < rhs.size; ++j) {
                for (size_t k = 0; k < size; ++k) {
                    result.data[i][j] += data[i][k] * rhs.data[k][j];
                }
            }
        }

        *this = std::move(result);
        return *this;
    }


    template<class T>
    SparseMatrix<T>::SparseMatrix(size_t rows, size_t cols) : _rows(rows), _cols(cols)
        {
    }

    template<class T>
    size_t SparseMatrix<T>::rows() const {
        return _rows;
    }

    template<class T>
    size_t SparseMatrix<T>::cols() const {
        return _cols;
    }

    template<class T>
    Row<T>& SparseMatrix<T>::operator[](size_t row) {
        return data[row];
    }

    template<class T>
    const Row<T>& SparseMatrix<T>::operator[](size_t row) const {
        auto it = data.find(row);
        if (it != data.end()) {
            return it->second;
        }
        else {
            static const Row<T> emptyRow{};
            return emptyRow;
        }
```

```
}

template<typename T>
T operator*(const std::vector<T>& v1, const std::vector<T>& v2) {
    if (v1.size() != v2.size()) {
        throw std::invalid_argument("Vectors must be of the same length.");
    }

    T result = 0;
    for (size_t i = 0; i < v1.size(); ++i) {
        result += v1[i] * v2[i];
    }
    return result;
}


template<class T>
SparseMatrix<T> SparseMatrix<T>::transpose() const {
    SparseMatrix<T> transposedMatrix(cols(), rows());
    for (size_t i = 0; i < rows(); ++i) {
        for (size_t j = 0; j < cols(); ++j) {
            transposedMatrix[j][i] = (*this)[i][j];
        }
    }
    return transposedMatrix;
}

template<class T>
SparseMatrix<T>::~SparseMatrix() = default;

template<class T>
SparseMatrix<T> operator+(const SparseMatrix<T>& lhs, const SparseMatrix<T>& rhs) {
    if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols()) {
        throw std::invalid_argument("Matrix dimensions must match for addition.");
    }

    SparseMatrix<T> result(lhs.rows(), lhs.cols());
    for (size_t i = 0; i < lhs.rows(); ++i) {
        for (size_t j = 0; j < lhs.cols(); ++j) {
            result[i][j] = lhs[i][j] + rhs[i][j];
        }
    }
    return result;
}

template<class T>
SparseMatrix<T> operator-(const SparseMatrix<T>& lhs, const SparseMatrix<T>& rhs) {
    if (lhs.rows() != rhs.rows() || lhs.cols() != rhs.cols()) {
```

```
            throw std::invalid_argument("Matrix dimensions must match for subtraction."
                );
        }

        SparseMatrix<T> result(lhs.rows(), lhs.cols());
        for (size_t i = 0; i < lhs.rows(); ++i) {
            for (size_t j = 0; j < lhs.cols(); ++j) {
                result[i][j] = lhs[i][j] - rhs[i][j];
            }
        }
        return result;
    }

    template<class T>
    SparseMatrix<T> operator*(const SparseMatrix<T>& lhs, const SparseMatrix<T>& rhs) {
        if (lhs.cols() != rhs.rows()) {
            throw std::invalid_argument(
                "The number of columns in the first matrix must match the number of rows
                     in the second.");
        }

        SparseMatrix<T> result(lhs.rows(), rhs.cols());
        for (size_t i = 0; i < lhs.rows(); ++i) {
            for (size_t j = 0; j < rhs.cols(); ++j) {
                T sum = T();
                for (size_t k = 0; k < lhs.cols(); ++k) {
                    sum += lhs[i][k] * rhs[k][j];
                }
                result[i][j] = sum;
            }
        }
        return result;
    }

    template<typename T>
    Matrix<T> outerProduct(const std::vector<T>& v1, const std::vector<T>& v2) {
        Matrix<T> matrix(v1.size(), v2.size());
        for (size_t i = 0; i < v1.size(); ++i) {
            for (size_t j = 0; j < v2.size(); ++j) {
                matrix[i][j] = v1[i] * v2[j];
            }
        }
        return matrix;
    }

    template<class T>
    SparseMatrix<T>::SparseMatrix(const SparseMatrix& other) : data(other.data), _rows(
        other._rows),
        _cols(other._cols) {
```

```

}

template<class T>
SparseMatrix<T>& SparseMatrix<T>::operator=(const SparseMatrix& other) {
    if (this != &other) {
        data = other.data;
        _rows = other._rows;
        _cols = other._cols;
    }
    return *this;
}

template<class T>
SparseMatrix<T>& SparseMatrix<T>::operator+=(const SparseMatrix<T>& rhs) {
    if (_rows != rhs._rows || _cols != rhs._cols) {
        throw std::invalid_argument("Matrix dimensions must match for addition.");
    }
    for (size_t i = 0; i < _rows; ++i) {
        for (size_t j = 0; j < _cols; ++j) {
            data[i][j] += rhs[i][j];
        }
    }
    return *this;
}

template<class T>
SparseMatrix<T>& SparseMatrix<T>::operator-=(const SparseMatrix<T>& rhs) {
    if (_rows != rhs._rows || _cols != rhs._cols) {
        throw std::invalid_argument("Matrix dimensions must match for addition.");
    }
    for (size_t i = 0; i < _rows; ++i) {
        for (size_t j = 0; j < _cols; ++j) {
            data[i][j] -= rhs[i][j];
        }
    }
    return *this;
}

template<class T>
SparseMatrix<T>& SparseMatrix<T>::operator*=(const SparseMatrix<T>& rhs) {
    if (_cols != rhs._rows) {
        throw std::invalid_argument(
            "The number of columns in the first matrix must match the number of rows
                 in the second for multiplication.");
    }

    SparseMatrix<T> result(_rows, rhs._cols);
    for (size_t i = 0; i < _rows; ++i) {
```

```
        for (size_t j = 0; j < rhs._cols; ++j) {
            for (size_t k = 0; k < _cols; ++k) {
                result[i][j] += data[i][k] * rhs[k][j];
            }
        }
    }

    *this = std::move(result);
    return *this;
}

template<class T>
class LUMatrix {
private:
    void init(const Matrix<T>& matrix);

public:
    Matrix<T> L;
    Matrix<T> U;

    explicit LUMatrix(const Matrix<T>& matrix);

    Matrix<T> solve();

    T determinant();
};

template<class T>
T LUMatrix<T>::determinant() {
    T deter = 1;
    for (size_t i = 0; i < U.rows(); ++i) {
        deter *= U[i][i];
    }

    return deter;
}

template<class T>
Matrix<T> LUMatrix<T>::solve() {
    Matrix<T> result(U.rows(), U.cols() - U.rows());
    for (int k = 0; k < U.cols() - U.rows(); ++k) {
        for (int i = U.rows() - 1; i >= 0; --i) {
            T sum = 0.0;
            for (int j = i + 1; j < U.rows(); ++j) {
                sum += U[i][j] * result[j][k];
            }
            result[i][k] = (U[i][U.rows() + k] - sum) / U[i][i];
        }
    }
```

```
    return result;
}

template<class T>
void LUMatrix<T>::init(const Matrix<T>& matrix) {
    U = matrix;
    for (size_t i = 0; i < matrix.rows(); ++i) {
        T max = 0;
        size_t row = i;
        for (size_t k = i; k < matrix.rows(); ++k) {
            if (std::fabs(matrix[k][i]) > max) {
                max = std::fabs(matrix[k][i]);
                row = k;
            }
        }

        std::swap(U[i], U[row]);
        std::swap(L[i], L[row]);

        L[i][i] = 1;

        for (size_t j = i + 1; j < matrix.rows(); ++j) {
            double factor = U[j][i] / double(U[i][i]);
            L[j][i] = factor;
            for (size_t k = i; k < matrix.cols(); ++k) {
                U[j][k] -= factor * U[i][k];
            }
        }
    }
}

template<class T>
LUMatrix<T>::LUMatrix(const Matrix<T>& matrix) : L(matrix.rows(), matrix.cols()), U(matrix.rows(), matrix.cols()) {
    init(matrix);
}

template<class T>
Matrix<T> inputMatrix(const std::string& path) {
    std::ifstream fin(path);
    size_t n, m;
    fin >> n >> m;
    Matrix<T> matrix(n, m);
    for (size_t i = 0; i < n; ++i) {
        for (size_t j = 0; j < m; ++j) {
            fin >> matrix[i][j];
        }
    }
```

```
    return matrix;
}

template<class T>
std::vector<T> inputVector(const std::string& path) {
    std::ifstream fin(path);
    size_t n;
    fin >> n;
    std::vector<T> vec(n);
    for (size_t i = 0; i < n; ++i) {
        fin >> vec[i];
    }
    return vec;
}

template<class T>
void printMatrix(const AbstractMatrix<T>& matrix) {
    for (size_t i = 0; i < matrix.rows(); ++i) {
        for (size_t j = 0; j < matrix.cols(); ++j) {
            std::cout << matrix[i][j] << "\t";
        }
        std::cout << "\n";
    }
}

template<class T>
void printVector(const std::vector<T>& vector) {
    for (size_t i = 0; i < vector.size(); ++i)
        std::cout << vector[i] << " ";
}

template<class T>
std::vector<T> tridiagonalSolve(const AbstractMatrix<T>& matrix, const std::vector<
    T>& b) {
    size_t n = matrix.rows();
    if (matrix.cols() != n || b.size() != n) {
        throw std::invalid_argument("Matrix must be square and the size of vector b
             must match.");
    }

    std::vector<T> C(n, 0);
    std::vector<T> D(n, 0);
    std::vector<T> x(n);

    C[0] = matrix[0][1] / matrix[0][0];
    D[0] = b[0] / matrix[0][0];

    for (size_t i = 1; i < n; ++i) {
        T m = 1 / (matrix[i][i] - matrix[i][i - 1] * C[i - 1]);
```

```
            C[i] = i < n - 1 ? matrix[i][i + 1] * m : 0;
            D[i] = (b[i] - matrix[i][i - 1] * D[i - 1]) * m;
        }

        x[n - 1] = D[n - 1];

        for (int i = n - 2; i >= 0; --i) {
            x[i] = D[i] - C[i] * x[i + 1];
        }

        return x;
    }

    template<class T>
    double norm(const AbstractMatrix<T>& matrix) {
        double norm = 0;
        for (size_t i = 0; i < matrix.rows(); ++i) {
            double currentSum = 0;
            for (size_t j = 0; j < matrix.cols(); ++j) {
                currentSum += fabs(matrix[i][j]);
            }
            norm = fmax(currentSum, norm);
        }

        return norm;
    }

    template<class T>
    double norm(const std::vector<T>& vector) {
        double norm = 0;
        for (size_t i = 0; i < vector.size(); ++i) {
            norm += pow(vector[i], 2);
        }
        return sqrt(norm);
    }

    template<class T>
    std::vector<T> diffVector(const std::vector<T>& lhs, const std::vector<T>& rhs) {
        if (lhs.size() != rhs.size())
            throw std::invalid_argument("invalid args");
        std::vector<T> result(lhs.size());
        for (size_t i = 0; i < lhs.size(); ++i) {
            result[i] = lhs[i] - rhs[i];
        }
        return result;
    }

    template<class T>
```

```
    std::vector<T> iterationSolve(const AbstractMatrix<T>& matrix, const std::vector<T
        >& b, T eps) {
        std::size_t n = matrix.rows();
        std::vector<T> beta(n, T());
        Matrix<T> alpha(n, n);
        for (std::size_t i = 0; i < n; ++i) {
            beta[i] = b[i] / matrix[i][i];
            for (std::size_t j = 0; j < n; ++j) {
                if (i == j) {
                    alpha[i][j] = 0;
                }
                else {
                    alpha[i][j] = -matrix[i][j] / matrix[i][i];
                }
            }
        }

        std::vector<T> x = beta;
        std::vector<T> x_next(n, T());

        bool continueIteration = true;

        double a = norm(alpha);

        std::cout << "\nNorm of matrix:\n";
        std::cout << a << "\n";

        size_t iter = 0;
        while (continueIteration) {
            continueIteration = false;

            for (std::size_t i = 0; i < n; ++i) {
                T sum = beta[i];
                for (std::size_t j = 0; j < n; ++j) {
                    sum += alpha[i][j] * x[j];
                }
                x_next[i] = sum;
            }

            if (a < 1) {
                if (a / (1 - a) * norm(diffVector(x_next, x)) > eps) {
                    continueIteration = true;
                }
            }
            else {
                if (norm(diffVector(x_next, x)) > eps) {
                    continueIteration = true;
                }
            }
```

```

        x = x_next;
        ++iter;
    }
    std::cout << "\nCount of iterations: " << iter << "\n";
    return x;
}

template<class T>
std::vector<T> SeidelSolve(const AbstractMatrix<T>& matrix, const std::vector<T>& b
    , T eps) {
    std::size_t n = matrix.rows();
    std::vector<T> beta(n, T());
    Matrix<T> alpha(n, n);
    Matrix<T> CMatrix(n, n);
    for (std::size_t i = 0; i < n; ++i) {
        beta[i] = b[i] / matrix[i][i];
        for (std::size_t j = 0; j < n; ++j) {
            if (i == j) {
                alpha[i][j] = 0;
            }
            else {
                alpha[i][j] = -matrix[i][j] / matrix[i][i];
            }
        }
    }

    for (size_t i = 0; i < n; ++i) {
        for (size_t j = i; j < n; ++j) {
            CMatrix[i][j] = alpha[i][j];
        }
    }
    std::vector<T> x = beta;
    std::vector<T> x_next(n, T());

    bool continueIteration = true;

    double a = norm(alpha);
    double c = norm(CMatrix);
    std::cout << "\nNorm of matrix:\n";
    std::cout << a << "\n";
    std::cout << "\nNorm of C matrix:\n";
    std::cout << c << "\n";

    size_t iter = 0;
    while (continueIteration) {
        continueIteration = false;

        x_next = x;
```

```
            for (std::size_t i = 0; i < n; ++i) {
                T sum = beta[i];
                for (size_t j = 0; j < n; ++j) {
                    sum += alpha[i][j] * x_next[j];
                }
                x_next[i] = sum;
            }

            if (a < 1) {
                if (c / (1 - a) * norm(diffVector(x_next, x)) > eps) {
                    continueIteration = true;
                }
            }
            else {
                if (norm(diffVector(x_next, x)) > eps) {
                    continueIteration = true;
                }
            }

            x = x_next;
            ++iter;
        }
        std::cout << "\nCount of iterations: " << iter << "\n";
        return x;
    }

    template<class T>
    void applyRotation(AbstractMatrix<T>& matrix, std::vector<std::vector<T>>&
        eigenVectors, size_t p, size_t q, T c, T s) {
        size_t n = matrix.rows();
        for (size_t i = 0; i < n; ++i) {
            T mpi = matrix[i][p];
            T mqi = matrix[i][q];
            matrix[i][p] = c * mpi + s * mqi;
            matrix[i][q] = -s * mpi + c * mqi;

            T epi = eigenVectors[i][p];
            T eqi = eigenVectors[i][q];
            eigenVectors[i][p] = c * epi + s * eqi;
            eigenVectors[i][q] = -s * epi + c * eqi;
        }

        for (size_t j = 0; j < n; ++j) {
            T mpj = matrix[p][j];
            T mqj = matrix[q][j];
            matrix[p][j] = c * mpj + s * mqj;
            matrix[q][j] = -s * mpj + c * mqj;
        }
    }
```

```
template<class T>
EigenResult<T> findEigenvaluesAndEigenvectors(Matrix<T>& inputMatrix, double eps) {
    size_t n = inputMatrix.rows();
    EigenResult<T> result(n);
    auto& eigenVectors = result.eigenVectors;
    auto& eigenValues = result.eigenValues;
    Matrix<T> matrix = inputMatrix;

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

    double offDiagonalNorm;
    do {
        offDiagonalNorm = 0.0;
        for (size_t p = 0; p < n; ++p) {
            for (size_t q = p + 1; q < n; ++q) {
                offDiagonalNorm += matrix[p][q] * matrix[p][q];
            }
        }

        if (sqrt(offDiagonalNorm) < eps)
            break;

        for (size_t p = 0; p < n; ++p) {
            for (size_t q = p + 1; q < n; ++q) {
                T apq = matrix[p][q];
                if (fabs(apq) > eps) {
                    T app = matrix[p][p];
                    T aqq = matrix[q][q];
                    T tau = (aqq - app) / (2 * apq);
                    T t = (tau / fabs(tau)) * (1.0 / (fabs(tau) + sqrt(1.0 + tau *
                        tau)));
                    T c = 1 / sqrt(1 + t * t);
                    T s = t * c;

                    applyRotation(matrix, eigenVectors, p, q, c, s);
                }
            }
        }
    } while (true);

    for (size_t i = 0; i < n; ++i) {
        eigenValues[i] = matrix[i][i];
    }

    return result;
```

```
}

template<class T>
class QRMatrix {
public:
    Matrix<T> Q;
    Matrix<T> R;

    explicit QRMatrix(const Matrix<T>& matrix) : Q(Matrix<T>::eye(matrix.rows())), R(matrix) {
        init();
    }

private:
    void init() {
        size_t m = R.rows();
        size_t n = R.cols();

        for (size_t j = 0; j < n - 1; ++j) {
            T norm_x = 0;
            for (size_t i = j; i < m; ++i) {
                norm_x += R[i][j] * R[i][j];
            }
            norm_x = std::sqrt(norm_x);

            std::vector<T> v(m, 0);
            T alpha = R[j][j] > 0 ? -norm_x : norm_x;
            for (size_t i = j; i < m; ++i) {
                v[i] = R[i][j] - ((i == j) ? alpha : 0);
            }

            Matrix<T> H = Matrix<T>::eye(n) - 2.0 * (outerProduct(v, v) / (v * v));

            R = H * R;
            Q *= H;
        }
    }
};

template<class T>
EigenResult<std::complex<T>> findEigenvaluesAndEigenvectorsByQR(Matrix<T>& inputMatrix, double eps1, double eps2) {
    bool continueIteration = true;
    Matrix<T> A = inputMatrix;
    std::vector<std::complex<T>> prevEigenvalues(inputMatrix.cols());

    while (continueIteration) {
        QRMatrix<T> QR(A);
        A = QR.R * QR.Q;
```

```

            for (int i = 0; i < A.cols(); ++i) {
                T underDiagonal = 0;
                for (int j = i + 1; j < A.rows(); ++j) {
                    underDiagonal += A[j][i] * A[j][i];
                }
                underDiagonal = std::sqrt(underDiagonal);
                if (i < A.cols() - 1 && underDiagonal > eps1) {
                    T a = A[i][i], b = A[i][i + 1], c = A[i + 1][i], d = A[i + 1][i + 1];
                    T tr = a + d;
                    T det = a * d - b * c;
                    T s = std::sqrt(std::abs(tr * tr / 4 - det));
                    if (std::abs(std::complex<T>(tr / 2, s) - prevEigenvalues[i]) < eps2) {
                        continueIteration = false;
                    }
                    prevEigenvalues[i] = std::complex<T>(tr / 2, s);
                }
                else if (i < A.cols() - 1) {
                    if (underDiagonal < eps1) {
                        continueIteration = false;
                    }
                }
            }
        }

        size_t i = 0;
        EigenResult<std::complex<T>> result(A.cols());
        while (i < A.cols()) {
            T underDiagonal = 0;
            for (int j = i + 1; j < A.rows(); ++j) {
                underDiagonal += A[j][i] * A[j][i];
            }
            underDiagonal = std::sqrt(underDiagonal);
            if (i < A.cols() - 1 && underDiagonal > eps1) {
                T a = A[i][i], b = A[i][i + 1], c = A[i + 1][i], d = A[i + 1][i + 1];
                T tr = a + d;
                T det = a * d - b * c;
                T s = std::sqrt(std::abs(tr * tr / 4 - det));
                result.eigenValues[i] = std::complex<T>(tr / 2, s);
                result.eigenValues[i + 1] = std::complex<T>(tr / 2, -s);
                i += 2;
            }
            else {
                result.eigenValues[i] = std::complex<T>(A[i][i], 0);
                i++;
            }
        }
```

```

        return result;
    }
} // numeric

#endif //MATRIX_H
```

Коэффициенты перед иксами системы:

```
9 -5 -6 3
1 -7 1 0
3 -4 9 0
6 -1 9 8
```

Вектор b:

```
-8
38
47
-8
```

```
#include <iostream>
#include "Matrix.h"
using namespace std;
using namespace numeric;

int main() {
    Matrix<double> matrix = inputMatrix<double>("input1Matrix.txt");
    vector<double> b = inputVector<double>("input1Vector.txt");
    Matrix<double> LinearSystem(matrix.rows(), matrix.rows() + 1);
    Matrix<double> InverseSystem(matrix.rows(), matrix.rows() + matrix.rows());
    for (size_t i = 0; i < matrix.rows(); ++i) {
        for (size_t j = 0; j < matrix.cols(); ++j) {
            LinearSystem[i][j] = matrix[i][j];
            InverseSystem[i][j] = matrix[i][j];
        }
    }
    for (size_t i = 0; i < b.size(); ++i) {
        LinearSystem[i][LinearSystem.rows()] = b[i];
    }
    for (size_t i = 0; i < matrix.cols(); ++i) {
        InverseSystem[i][i + matrix.rows()] = 1;
    }
    LUMatrix<double> LinearSystemLU(LinearSystem);
    LUMatrix<double> LU(matrix);
    LUMatrix<double> InverseSystemLU(InverseSystem);
    cout << "\nSolving system:\n";
    printMatrix(LinearSystemLU.solve());
    cout << "\nDeterminant:\n";
    cout << LinearSystemLU.determinant();
    cout << "\nInverseMatrix:\n";
```

```
    printMatrix(InverseSystemLU.solve());
    cout << "\nU:\n";
    printMatrix(LU.U);
    cout << "\nL:\n";
    printMatrix(LU.L);
    cout << "\nMatrix:\n";
    printMatrix(LU.L * LU.U);
    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: 3

$$\begin{cases} 13x_1 - 5x_2 = -66 \\ -4x_1 + 9x_2 - 5x_3 = -47 \\ -x_2 - 12x_3 - 6x_4 = -43 \\ 6x_3 + 20x_4 - 5x_5 = -74 \\ 4x_4 + 5x_5 = 14 \end{cases}$$

5 Результаты работы

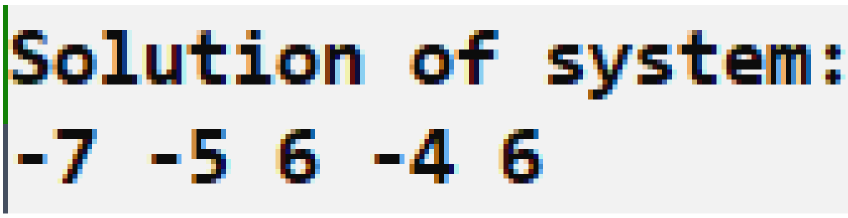

Рис. 2: Вывод программы в консоли

6 Исходный код

Коэффициенты перед исками системы:

```
13 -5 0 0 0
-4 9 -5 0 0
0 -1 -12 -6 0
0 0 6 20 -5
0 0 0 4 5
```

Вектор b:

```
-66
-47
-43
-74
14
```

```cpp
#include <iostream>
#include "Matrix.h"
using namespace std;
using namespace numeric;

int main() {
    Matrix<double> matrix = inputMatrix<double>("input2Matrix.txt");
    vector<double> b = inputVector<double>("input2Vector.txt");
    cout << "\nMatrix:\n";
    printMatrix(matrix);
    cout << "\nVector b:\n";
    printVector(b);
    cout << "\nSolution of system:\n";
    printVector(tridiagonalSolve(matrix, b));
    return 0;
}
```

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: 3

$$\begin{cases} -23x_1 - 7x_2 + 5x_3 + 2x_4 = -26 \\ -7x_1 - 21x_2 + 4x_3 + 9x_4 = -55 \\ 9x_1 + 5x_2 - 31x_3 - 8x_4 = -58 \\ x_2 - 2x_3 + 10x_4 = -24 \end{cases}$$

8 Результаты работы

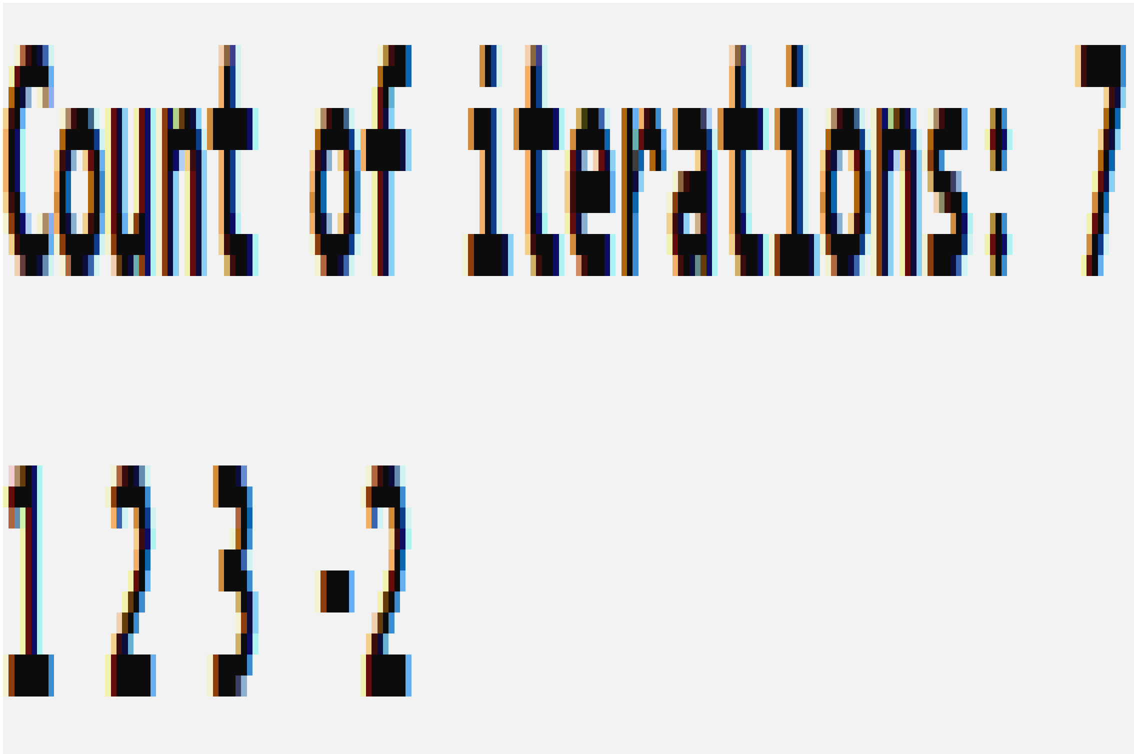

Рис. 3: Вывод программы в консоли

9 Исходный код

Коэффициенты перед иксами системы:

```
-23 -7 5 2
-7 -21 4 9
9 5 -31 -8
0 1 -2 10
```

Вектор b:

```
-26
-55
-58
-24
```

```cpp
#include <iostream>
#include "Matrix.h"
using namespace std;
using namespace numeric;

int main() {
    Matrix<double> matrix = inputMatrix<double>("input3Matrix.txt");
    vector<double> b = inputVector<double>("input3Vector.txt");
    cout << "\nMatrix:\n";
    printMatrix(matrix);
    cout << "\nVector b:\n";
    printVector(b);
    cout << "\nSolution of system by iterations\n";
    printVector(iterationSolve<double>(matrix, b, 0.001));
    cout << "\nSolution of system by Seidel\n";
    printVector(SeidelSolve<double>(matrix, b, 0.001));


    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: 3

$$\begin{pmatrix} 5 & 5 & 3 \\ 5 & -4 & 1 \\ 3 & 1 & 2 \end{pmatrix}$$

11 Результаты работы

```
Eigen values
-6.23937 8.70547 0.533906
Eigen vectors
0.415191 -0.830359 0.371646
-0.908818 -0.360255 0.210393
-0.0408142 -0.425112 -0.90422
```

Рис. 4: Вывод программы в консоли

12 Исходный код

Матрица:

```
5 5 3
5 -4 1
3 1 2
```

```
#include <iostream>
#include "Matrix.h"
using namespace std;
using namespace numeric;

int main() {
    Matrix<double> matrix = inputMatrix<double>("input4Matrix.txt");
    cout << "\nMatrix:\n";
    printMatrix(matrix);
    cout << "\nEigen values\n";
    auto res = findEigenvaluesAndEigenvectors(matrix, 0.001);
    printVector(res.eigenValues);
    cout << "\nEigen vectors\n";
    for(const auto& x: res.eigenVectors) {
        printVector(x);
        cout << "\n";
    }
    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: 3

$$\begin{pmatrix} 5 & -5 & -6 \\ -1 & -8 & -5 \\ 2 & 7 & -3 \end{pmatrix}$$

14 Результаты работы

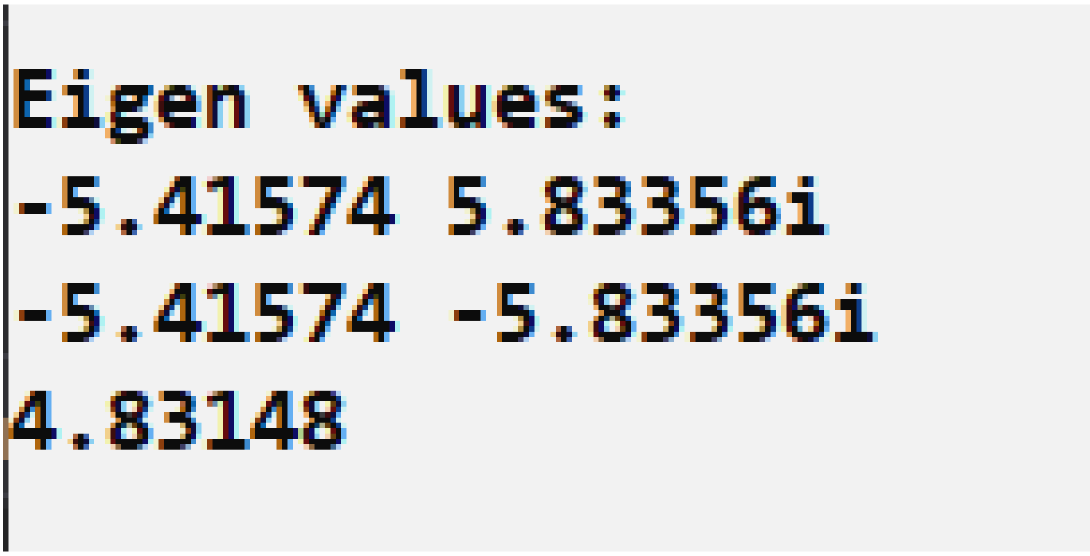

Рис. 5: Вывод программы в консоли

15 Исходный код

Матрица:

```
5 -5 -6
-1 -8 -5
2 7 -3
```

```
#include <iostream>
#include "Matrix.h"
using namespace std;
using namespace numeric;

int main() {
    Matrix<double> matrix = inputMatrix<double>("input5Matrix.txt");
    cout << "\nMatrix:\n";
    printMatrix(matrix);
    QRMatrix<double> QRDecomposition(matrix);
    cout << "\nEigen values:\n";
    auto res = findEigenvaluesAndEigenvectorsByQR(matrix, 0.001, 0.0001);
    for(auto c: res.eigenValues) {
        if(c.imag() == 0) {
            cout << c.real() << "\n";
        } else {
            cout << c.real() << " " << c.imag() << "i\n";
        }
    }
    return 0;
}
```