CS205 C/ C++ Programming - Project

Name: 李亦琛 (Li Yichen), 孙畅(Sun Chang), 邱泽宇(Qiu zeyu)

SID: 11912433, 11910437, 11912415

Part 1 - Analysis

We need to design a matrix library, this library should contain the following functions:

1. Because the matrix should be variably-sized, so we need to use `vector` to build matrix.
2. Because the type of matrix is vague, so we need to use `template<typename T>`.
3. We need to define the common operations between matrices, vectors and constants, such as <<, =, +, -, *, /...
4. We need to define the common functions between matrices, vectors and constants, such as transposion, conjugation, element-wise-multiplication and so on.
5. We need to define some special functions based on the characteristics of matrix, such as reshape, slicing and so on.
6. We need to check the aplicability of functions with different input. We use `try & catch` to throw different messages when meet diffent type of mistakes.
7. We write examples to test whether the error handalling works well for each function.

Part 2 - Code

Matrix.h

```
#pragma once
#ifndef PROJECT_MATRIX_MATRIX_H
#define PROJECT_MATRIX_MATRIX_H
#include <iostream>
#include <vector>
#include <complex>
#include <algorithm>
using namespace std;
template<typename T>
class Matrix {
private:
    vector<vector<T>> matrix;
    int row;
    int col;
private:
    T get_det(vector<vector<T>> m, int n);
    void QR(Matrix<T>& A, Matrix<T>& Q, Matrix<T>& R);
    bool check_df(const string& T_str);
public:
    //constructor
    Matrix(int row = 0, int cow = 0) {
        //initialize
        if (row > 0 && cow > 0) {
            this->row = row;
            this->col = cow;
            for (int i = 0; i < row; ++i) {
                vector<T> r;
                for (int j = 0; j < col; ++j) {
                    r.emplace_back(0);
                }
                matrix.emplace_back(r);
            }
        }
        else {
            this->row = 0;
            this->col = 0;
        }
    };
    explicit Matrix(vector<vector<T>> m) {
        if (m.size() > 0) {
            if (m[0].size() > 0) {
                this->matrix = m;
                row = m.size();
                col = m[0].size();
            }
            else {
                row = 0;
                col = 0;
            }
        }
        else {
            row = 0;
            col = 0;
        }
    }
    Matrix(const Matrix<T>& m) {
        matrix = m.matrix;
        row = m.row;
        col = m.col;
    }
    //getter & setter
    int getRow() { return row; }
    int getCol() { return col; }
    vector<vector<T>> getMatrix() { return matrix; }
    void setMatrix(vector<vector<T>> m) {
        if (m.size() > 0) {
            if (m[0].size() > 0) {
                this->matrix = m;
                row = m.size();
                col = m[0].size();
            }
            else {
                row = 0;
                col = 0;
            }
        }
        else {
            row = 0;
            col = 0;
        }
    }
    bool un_initialized();
    //Q3
    template<typename S>
    friend ostream& operator<<(ostream& os, const Matrix<S>& other);
    template<typename S>
    friend ostream& operator<<(ostream& os, const vector<S>& other);
    Matrix<T>& operator=(const Matrix<T>& next);
    Matrix<T> operator+(const Matrix<T>& next);
    template<typename S>
    friend vector<S> operator+(const vector<S>& v1, const vector<S>& v2);
    Matrix<T> operator-(const Matrix<T>& next);
    template<typename S>
    friend vector<S> operator-(vector<S>& v1, vector<S>& v2);
    Matrix<T> operator*(T val);
    template<typename S>
    friend Matrix<S> operator*(S val, const Matrix<S>& other);
    template<typename S>
    friend vector<S> operator*(vector<S>& v, S val);
    template<typename S>
    friend vector<S> operator*(S val, vector<S>& v);
    Matrix<T> operator/(T val);
    template<typename S>
    friend vector<S> operator/(vector<S>& v, S val);
    static Matrix<T> transposition(const Matrix<T>& m);
    Matrix<T> transposition();
    static Matrix<T> conjugation(const Matrix<T>& m);
    Matrix<T> conjugation();
    Matrix<T> element_wise_mul(const Matrix<T>& m);
    static Matrix<T> element_wise_mul(const Matrix<T>& m1, const Matrix<T>& m2);
    static vector<T> element_wise_mul(vector<T>& v1, vector<T>& v2);
    Matrix<T> operator*(const Matrix<T>& m);
    vector<T> operator*(vector<T>& vec);
    template<typename S>
    friend vector<S> operator*(vector<S>& vec, const Matrix<S>& other);
    template<typename S>
    friend S operator*(vector<S> v1, vector<S> v2);
    static T dot_product(Matrix<T>& m1, Matrix<T>& m2);
    T dot_product(Matrix<T>& m);
    static T dot_product(vector<T>& v1, vector<T>& v2);
    static vector<T> cross_product(vector<T>& v1, vector<T>& v2);
    //Q4
    bool check_type_axis(int type, int axis);
    T find_max();
    T find_max(int type, int axis);
    T find_min();
    T find_min(int type, int axis);
    T sum();
    T sum(int type, int axis);
    T avg();
    T avg(int type, int axis);
    //Q5
    bool check_square();
    T trace();
    T determinant();
    Matrix<T> adjugate();
    bool has_inverse();
    Matrix<T> inverse();
    T norm2();
    static T norm2(Matrix<T>& m);
    static T norm2(vector<T>& v);
    vector<T> eigenvalues();
    vector<vector<T>> eigenvectors();
    void print_eigenvectors();
    //Q6
    static Matrix<T> reshape(Matrix<T> m, int r, int c);
    Matrix<T> reshape(int r, int c);
    static Matrix<T> slicing(Matrix<T> m, int row_begin, int row_end, int
col_begin, int col_end, int row_interval, int col_interval);
    static Matrix<T> row_slicing(Matrix<T> m, int row_begin, int row_end, int
interval);
    static Matrix<T> col_slicing(Matrix<T> m, int col_begin, int col_end, int
interval);
    Matrix<T> slicing(int row_begin, int row_end, int col_begin, int col_end,
int row_interval, int col_interval);
    Matrix<T> row_slicing(int row_begin, int row_end, int interval);
    Matrix<T> col_slicing(int col_begin, int col_end, int interval);
    //Q7
    static Matrix<T> convolution(Matrix<T> m1, Matrix<T> m2);
};

//解决复数编译报错问题，如果泛型类型是complex会自动调用该库
template<typename D>
class Matrix<complex<D>> {
private:
    int row;
    int col;
    vector<vector<complex<D>>> matrix;
private:
    complex<D> get_det(vector<vector<complex<D>>> m, int n);
public:
    //constructor
    Matrix(int row = 0, int cow = 0) {
        //initialize
        if (row > 0 && cow > 0) {
            this->row = row;
            this->col = cow;
            for (int i = 0; i < row; ++i) {
                vector<complex<D>> r;
                for (int j = 0; j < col; ++j) {
                    r.emplace_back(0);
                }
                matrix.emplace_back(r);
            }
        }
        else {
            this->row = 0;
            this->col = 0;
        }
    };
    explicit Matrix(vector<vector<complex<D>>> m) {
        if (m.size() > 0) {
            if (m[0].size() > 0) {
                this->matrix = m;
                row = m.size();
                col = m[0].size();
            }
            else {
                row = 0;
                col = 0;
            }
        }
        else {
            row = 0;
            col = 0;
        }
    }
    Matrix(const Matrix<complex<D>>& m) {
        matrix = m.matrix;
        row = m.row;
        col = m.col;
    }
    //getter & setter
    int getRow() { return row; }
    int getCol() { return col; }
    vector<vector<complex<D>>> getMatrix() { return matrix; }
    void setMatrix(vector<vector<complex<D>>> m) {
        if (m.size() > 0) {
            if (m[0].size() > 0) {
                this->matrix = m;
                row = m.size();
                col = m[0].size();
            }
            else {
                row = 0;
                col = 0;
            }
        }
        else {
            row = 0;
            col = 0;
        }
    }
    bool un_initialized();
    //Q3
    template<typename S>
    friend ostream& operator<<(ostream& os, const Matrix<complex<S>>& other);
    template<typename S>
    friend ostream& operator<<(ostream& os, const vector<complex<S>>& other);
    Matrix<complex<D>>& operator=(const Matrix<complex<D>>& next);
    Matrix<complex<D>> operator+(const Matrix<complex<D>>& next);
    template<typename S>
    friend vector<complex<S>> operator+(const vector<complex<S>>& v1, const
vector<complex<S>>& v2);
    Matrix<complex<D>> operator-(const Matrix<complex<D>>& next);
    template<typename S>
    friend vector<complex<S>> operator-(vector<complex<S>>& v1,
vector<complex<S>>& v2);
    Matrix<complex<D>> operator*(complex<D> val);
    template<typename S>
    friend Matrix<complex<S>> operator*(complex<S> val, const
Matrix<complex<S>>& other);
    template<typename S>
    friend vector<complex<S>> operator*(vector<complex<S>>& v, complex<S> val);
    template<typename S>
    friend vector<complex<S>> operator*(complex<S> val, vector<complex<S>>& v);
    Matrix<complex<D>> operator/(complex<D> val);
    template<typename S>
    friend vector<complex<S>> operator/(vector<complex<S>>& v, complex<S> val);
    static Matrix<complex<D>> transposition(const Matrix<complex<D>>& m);
    Matrix<complex<D>> transposition();
    static Matrix<complex<D>> conjugation(const Matrix<complex<D>>& m);
    Matrix<complex<D>> conjugation();
    Matrix<complex<D>> element_wise_mul(const Matrix<complex<D>>& m);
    static Matrix<complex<D>> element_wise_mul(const Matrix<complex<D>>& m1,
const Matrix<complex<D>>& m2);
    static vector<complex<D>> element_wise_mul(vector<complex<D>>& v1,
vector<complex<D>>& v2);
    Matrix<complex<D>> operator*(const Matrix<complex<D>>& m);
    vector<complex<D>> operator*(vector<complex<D>>& vec);
    template<typename S>
    friend vector<complex<S>> operator*(vector<complex<S>>& vec, const
Matrix<complex<S>>& other);
    template<typename S>
    friend complex<S> operator*(vector<complex<S>> v1, vector<complex<S>> v2);
    static complex<D> dot_product(Matrix<complex<D>>& m1, Matrix<complex<D>>&
m2);
    complex<D> dot_product(Matrix<complex<D>>& m);
    static complex<D> dot_product(vector<complex<D>>& v1, vector<complex<D>>&
v2);
    static vector<complex<D>> cross_product(vector<complex<D>>& v1,
vector<complex<D>>& v2);
    //Q4
    bool check_type_axis(int type, int axis);
    complex<D> sum();
    complex<D> sum(int type, int axis);
    complex<D> avg();
    complex<D> avg(int type, int axis);
    //Q5
    bool check_square();
    complex<D> trace();
    complex<D> determinant();
    Matrix<complex<D>> adjugate();
    bool has_inverse();
    Matrix<complex<D>> inverse();
    D norm2();
    static D norm2(Matrix<complex<D>>& m);
    static D norm2(vector<complex<D>>& v);
    //Q6
    static Matrix<complex<D>> reshape(Matrix<complex<D>> m, int r, int c);
    Matrix<complex<D>> reshape(int r, int c);
    static Matrix<complex<D>> slicing(Matrix<complex<D>> m, int row_begin, int
row_end, int col_begin, int col_end, int row_interval, int col_interval);
    static Matrix<complex<D>> row_slicing(Matrix<complex<D>> m, int row_begin,
int row_end, int interval);
    static Matrix<complex<D>> col_slicing(Matrix<complex<D>> m, int col_begin,
int col_end, int interval);
    Matrix<complex<D>> slicing(int row_begin, int row_end, int col_begin, int
col_end, int row_interval, int col_interval);
    Matrix<complex<D>> row_slicing(int row_begin, int row_end, int interval);
    Matrix<complex<D>> col_slicing(int col_begin, int col_end, int interval);
    //Q7
    static Matrix<complex<D>> convolution(Matrix<complex<D>> m1,
Matrix<complex<D>> m2);
};
#endif //PROJECT_MATRIX_MATRIX_H
```

Matrix.cpp

```
#include "Matrix.h"
//检查T是不是浮点数
template<typename T>
bool Matrix<T>::check_df(const string& T_str) {
    string s1 = typeid(double).name();
    string s2 = typeid(double_t).name();
    string s3 = typeid(float).name();
    string s4 = typeid(float_t).name();
    if ((T_str == s1) || (T_str == s2) || (T_str == s3) || (T_str == s4)) return
true;
    return false;
}
//检查初始化
template<typename T>
bool Matrix<T>::un_initialized() {
    if (row == 0 || col == 0)return true;
    else return false;
}
//overload cout<<
template<typename T>
ostream& operator<<(ostream& os, const Matrix<T>& other) {
    try {
        if (other.row == 0 || other.col == 0) {
            throw 1;
        }
        else {
            for (int i = 0; i < other.row; ++i) {
                for (int j = 0; j < other.col; ++j) {
                    os << other.matrix[i][j] << " ";
                }
                os << endl;
            }
        }
    }
    catch (int i)
    {
        os << "The matrix isn't initialized." << endl;
    }
    return os;
}
template<typename T>
ostream& operator<<(ostream& os, const vector<T>& other) {
    try
    {
        if (other.size() == 0) {
            throw 0;
        }
        else {
            os << "[";
            for (int i = 0; i < other.size() - 1; ++i) {
                os << other[i] << ",";
            }
            os << other[other.size() - 1] << "]"<<endl;
        }
    }
    catch (int i)
    {
        os << "The vector isn't initialized." << endl;
    }
    return os;
}
//overload =
template<typename T>
Matrix<T>& Matrix<T>::operator=(const Matrix<T>& next) {
    if (this != &next) {
        this->row = next.row;
        this->col = next.col;
        this->matrix = next.matrix;
    }
    return *this;
}
//A+B
template<typename T>
Matrix<T> Matrix<T>::operator+(const Matrix<T>& next) {
    try {
        if (un_initialized() || next.row == 0 || next.col == 0) {
            throw 0;
        }
        vector<vector<T>> add;
        //矩阵大小匹配
        if (this->row == next.row && this->col == next.col) {
            for (int i = 0; i < this->row; ++i) {
                vector<T> r;
                for (int j = 0; j < this->col; ++j) {
                    r.emplace_back(this->matrix[i][j] + next.matrix[i][j]);
                }
                add.emplace_back(r);
            }
            return Matrix<T>(add);
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a)
    {
        //矩阵大小不匹配
        cout << "The sizes of matrices are mismatched." << endl;
    }
}
//v1+v2
template<typename T>
vector<T> operator+(const vector<T>& v1, const vector<T>& v2) {
    vector<T> v;
    try {
        if (v1.size() == v2.size()) {
            for (int i = 0; i < v1.size(); ++i) {
                v.emplace_back(v1[i] + v2[i]);
            }
            return v;
```

```cpp
        }
        throw 0;
    }
    catch (int i)
    {
        cout << "The sizes of vectors are mismatched." << endl;
    }
}
//A-B
template<typename T>
Matrix<T> Matrix<T>::operator-(const Matrix<T>& next) {
    try {
        if (un_initialized() || next.row == 0 || next.col == 0) {
            throw 0;
        }
        vector<vector<T>> sub;
        //矩阵大小匹配
        if (this->row == next.row && this->col == next.col) {
            for (int i = 0; i < this->row; ++i) {
                vector<T> r;
                for (int j = 0; j < this->col; ++j) {
                    r.emplace_back(this->matrix[i][j] - next.matrix[i][j]);
                }
                sub.emplace_back(r);
            }
            return Matrix<T>(sub);
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a)
    {
        //矩阵大小不匹配
        cout << "The sizes of matrices are mismatched." << endl;
    }
}
//v1-v2
template<typename T>
vector<T> operator-(vector<T>& v1, vector<T>& v2) {
    try {
        if (v1.size() == 0 || v2.size() == 0) {
            throw 0;
        }
        vector<T> v;
        if (v1.size() == v2.size()) {
            for (int i = 0; i < v1.size(); ++i) {
                v.emplace_back(v1[i] - v2[i]);
            }
            return v;
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The vector isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of vectors are mismatched." << endl;
    }
}
//scalar multiplication
template<typename T>
Matrix<T> Matrix<T>::operator*(T val) {
    try {
        if (un_initialized()) {
            throw 0;
        }
        vector<vector<T>> mul;
        for (int i = 0; i < this->row; ++i) {
            vector<T> r;
            for (int j = 0; j < this->col; ++j) {
                r.emplace_back(this->matrix[i][j] * val);
            }
            mul.emplace_back(r);
        }
        return Matrix<T>(mul);
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
}
template<typename T>
Matrix<T> operator*(T val, const Matrix<T>& other) {
    try {
        if (other.row == 0 || other.col == 0) {
            throw 0;
        }
        vector<vector<T>> mul;
        for (int i = 0; i < other.row; ++i) {
            vector<T> r;
            for (int j = 0; j < other.col; ++j) {
                r.emplace_back(val * other.matrix[i][j]);
            }
            mul.emplace_back(r);
        }
        return Matrix<T>(mul);
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }

}
template<typename T>
vector<T> operator*(vector<T>& v, T val) {
    try {
        if (v.size() == 0) {
            throw 0;
        }
        vector<T> mul;
        for (int i = 0; i < v.size(); ++i) {
            mul.emplace_back(v[i] * val);
        }
        return mul;
    }
    catch (int i)
    {
        cout << "The vector isn't initialized." << endl;
    }

}
template<typename T>
vector<T> operator*(T val, vector<T>& v) {
    try {
        if (v.size() == 0) {
            throw 0;
        }
        vector<T> mul;
        for (int i = 0; i < v.size(); ++i) {
            mul.emplace_back(v[i] * val);
        }
        return mul;
    }
    catch (int i)
    {
        cout << "The vector isn't initialized." << endl;
    }

}
//scalar division
template<typename T>
Matrix<T> Matrix<T>::operator/(T val) {
    try {
        if (un_initialized()) {
            throw 0;
        }
        vector<vector<T>> div;
        for (int i = 0; i < this->row; ++i) {
            vector<T> r;
            for (int j = 0; j < this->col; ++j) {
                r.emplace_back(this->matrix[i][j] / val);
            }
            div.emplace_back(r);
        }
        return Matrix<T>(div);
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }

}
template<typename T>
vector<T> operator/(vector<T>& v, T val) {
    try {
        if (v.size() == 0) {
            throw 0;
        }
        vector<T> div;
        for (int i = 0; i < v.size(); ++i) {
            div.emplace_back(v[i] / val);
        }
        return div;
    }
    catch (int i)
    {
        cout << "The vector isn't initialized." << endl;
    }

}
//transposition
template<typename T>
Matrix<T> Matrix<T>::transposition(const Matrix<T>& m) {
    try {
        if (m.row == 0 || m.col == 0) {
            throw 0;
        }
        vector<vector<T>> trans;
        for (int i = 0; i < m.col; ++i) {
            vector<T> trans_r;
            for (int j = 0; j < m.row; ++j) {
                trans_r.emplace_back(m.matrix[j][i]);
            }
            trans.emplace_back(trans_r);
        }
        return Matrix<T>(trans);
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
}
template<typename T>
Matrix<T> Matrix<T>::transposition() {
    try {
        if (un_initialized()) {
            throw 0;
        }
        vector<vector<T>> trans;
        for (int i = 0; i < this->col; ++i) {
            vector<T> trans_r;
            for (int j = 0; j < this->row; ++j) {
                trans_r.emplace_back(this->matrix[j][i]);
            }
            trans.emplace_back(trans_r);
        }
        return Matrix<T>(trans);
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }

};
//conjugation 共轭
template<typename T>
Matrix<T> Matrix<T>::conjugation(const Matrix<T>& m) {
    return transposition(m);
}
template<typename T>
Matrix<T> Matrix<T>::conjugation() {
    return transposition();
}
//element-wise multiplication
template<typename T>
Matrix<T> Matrix<T>::element_wise_mul(const Matrix<T>& m) {
    try {
        if (m.row == 0 || m.col == 0 || row == 0 || col == 0) {
            throw 0;
        }
        vector<vector<T>> mul;
        if (row == m.row && col == m.col) {
            for (int i = 0; i < row; ++i) {
                vector<T> r;
                for (int j = 0; j < col; ++j) {
                    r.emplace_back(matrix[i][j] * m.matrix[i][j]);
                }
                mul.emplace_back(r);
            }
            return Matrix<T>(mul);
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of matrices are mismatched." << endl;
    }
}
template<typename T>
Matrix<T> Matrix<T>::element_wise_mul(const Matrix<T>& m1, const Matrix<T>& m2)
{
    try {
        if (m1.row == 0 || m1.col == 0 || m2.row == 0 || m2.col == 0) {
            throw 0;
        }
        vector<vector<T>> mul;
        if (m1.row == m2.row && m1.col == m2.col) {
            for (int i = 0; i < m1.row; ++i) {
                vector<T> r;
                for (int j = 0; j < m1.col; ++j) {
                    r.emplace_back(m1.matrix[i][j] * m2.matrix[i][j]);
                }
                mul.emplace_back(r);
            }
            return Matrix<T>(mul);
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of matrices are mismatched." << endl;
    }
}
template<typename T>
vector<T> Matrix<T>::element_wise_mul(vector<T>& v1, vector<T>& v2) {
    try {
        if (v1.size() == 0 || v2.size() == 0) {
            throw 0;
        }
        vector<T> mul;
        if (v1.size() == v2.size()) {
            for (int i = 0; i < v1.size(); ++i) {
                mul.emplace_back(v1[i] * v2[i]);
            }
            return mul;
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The vector isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of vectors are mismatched." << endl;
    }
}
//matrix-matrix multiplication
template<typename T>
Matrix<T> Matrix<T>::operator*(const Matrix<T>& m) {
    try {
        if (row == 0 || col == 0 || m.row == 0 || m.col == 0) {
            throw 0;
        }
        vector<vector<T>> mul;
        if (col == m.row) {
            //initialize
            for (int i = 0; i < row; ++i) {
                vector<T> r;
                for (int j = 0; j < m.col; ++j) {
                    r.emplace_back(0);
                }
                mul.emplace_back(r);
            }
            for (int i = 0; i < row; ++i) {
                for (int j = 0; j < m.col; ++j) {
                    for (int k = 0; k < col; ++k) {
                        mul[i][j] += matrix[i][k] * m.matrix[k][j];
                    }
                }
            }
            return Matrix<T>(mul);
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of matrices are mismatched." << endl;
    }
}
//matrix-vector multiplication
template<typename T>
vector<T> Matrix<T>::operator*(vector<T>& vec) {
    try {
        if (row == 0 || col == 0 || vec.size() == 0) {
            throw 0;
        }
        vector<T> v;
        if (col == vec.size()) {
            //initialize
            for (int i = 0; i < row; ++i) {
                v.emplace_back(0);
            }
            for (int i = 0; i < row; ++i) {
                for (int j = 0; j < col; ++j) {
                    v[i] += matrix[i][j] * vec[j];
                }
            }
            return v;
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix or the vector isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The size of matrix and the size of vector are mismatched." <<
endl;
    }
}
template<typename T>
vector<T> operator*(vector<T>& vec, const Matrix<T>& other) {
    try {
        if (other.row == 0 || other.col == 0 || vec.size() == 0) {
            throw 0;
        }
        vector<T> v;
        if (vec.size() == other.row) {
            //initialize
            for (int i = 0; i < other.col; ++i) {
                v.emplace_back(0);
            }
            for (int i = 0; i < other.col; ++i) {
                for (int j = 0; j < other.row; ++j) {
                    v[i] += other.matrix[j][i] * vec[j];
                }
            }
            return v;
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix or the vector isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The size of matrix and the size of vector are mismatched." <<
endl;
    }
}
template<typename T>
T operator*(vector<T> v1, vector<T> v2) {
    try {
        if (v1.size() == 0 || v2.size() == 0) {
            throw 0;
        }
        T mul = 0;
        if (v1.size() == v2.size()) {
            for (int i = 0; i < v1.size(); ++i) {
                mul += v1[i] * v2[i];
            }
            return mul;
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The vector isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of vectors are mismatched." << endl;
    }
}
//dot product 内积
template<typename T>
T Matrix<T>::dot_product(Matrix<T>& m1, Matrix<T>& m2) {
    try {
        if (m1.un_initialized() || m2.un_initialized()) {
            throw 0;
        }
        T dot_prod = 0;
        if (m1.row == m2.row && m1.col == m2.col) {
            for (int i = 0; i < m1.row; ++i) {
                for (int j = 0; j < m1.col; ++j) {
                    dot_prod += m1.matrix[i][j] * m2.matrix[i][j];
                }
            }
            return dot_prod;
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of matrices are mismatched." << endl;
    }
}
template<typename T>
T Matrix<T>::dot_product(Matrix<T>& m) {
    try {
        if (m.un_initialized() || un_initialized()) {
            throw 0;
        }
        T dot_prod = 0;
        if (row == m.row && col == m.col) {
            for (int i = 0; i < row; ++i) {
                for (int j = 0; j < col; ++j) {
                    dot_prod += matrix[i][j] * m.matrix[i][j];
                }
            }
            return dot_prod;
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of matrices are mismatched." << endl;
    }
}
template<typename T>
T Matrix<T>::dot_product(vector<T>& v1, vector<T>& v2) {
    return v1 * v2;
}
//cross product n=3
template<typename T>
vector<T> Matrix<T>::cross_product(vector<T>& v1, vector<T>& v2) {
    try {
        vector<T> prod;
        if (v1.size() == 3 && v2.size() == 3) {
            prod.emplace_back(v1[1] * v2[2] - v1[2] * v2[1]);
            prod.emplace_back(v1[2] * v2[0] - v1[0] * v2[2]);
            prod.emplace_back(v1[0] * v2[1] - v1[1] * v2[0]);
            return prod;
        }
        throw 0;
    }
    catch (int i)
    {
        cout << "The sizes of vectors are not 3 and have no cross product." <<
endl;
    }
}
//check type and axis
template<typename T>
```

```cpp
bool Matrix<T>::check_type_axis(int type, int axis) {
    if (un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return false;
    }
    else {
        if (type == 0) {//row
            if (axis >= 0 && axis < row) return true;
            else {
                cout << "Invalid axis." << endl;
                return false;
            }
        }
        else if (type == 1) {//col
            if (axis >= 0 && axis < col) return true;
            else {
                cout << "Invalid axis." << endl;
                return false;
            }
        }
        else {
            cout << "Invalid type." << endl;
            return false;
        }
    }
}
//find the maximum value
template<typename T>
T Matrix<T>::find_max() {
    if (un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return 0;
    }
    T max = matrix[0][0];
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            if (max < matrix[i][j]) {
                max = matrix[i][j];
            }
        }
    }
    return max;
}
template<typename T>
T Matrix<T>::find_max(int type, int axis) {
    if (check_type_axis(type, axis)) {
        T max = 0;
        if (type == 0) {//row
            int r = axis;
            max = matrix[r][0];
            for (int i = 0; i < col; ++i) {
                if (max < matrix[r][i]) {
                    max = matrix[r][i];
                }
            }
        }
        else {//type==1, col
            int c = axis;
            max = matrix[0][c];
            for (int i = 0; i < row; ++i) {
                if (max < matrix[i][c]) {
                    max = matrix[i][c];
                }
            }
        }
        return max;
    }
    else return 0;
}
//find the minimum value
template<typename T>
T Matrix<T>::find_min() {
    if (un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return 0;
    }
    T min = matrix[0][0];
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            if (min > matrix[i][j]) {
                min = matrix[i][j];
            }
        }
    }
    return min;
}
template<typename T>
T Matrix<T>::find_min(int type, int axis) {
    if (check_type_axis(type, axis)) {
        T min = 0;
        if (type == 0) {//row
            int r = axis;
            min = matrix[r][0];
            for (int i = 0; i < col; ++i) {
                if (min > matrix[r][i]) {
                    min = matrix[r][i];
                }
            }
        }
        else {//type==1, col
            int c = axis;
            min = matrix[0][c];
            for (int i = 0; i < row; ++i) {
                if (min > matrix[i][c]) {
                    min = matrix[i][c];
                }
            }
        }
        return min;
    }
    else return 0;
}
//sum
template<typename T>
T Matrix<T>::sum() {
    T sum = 0;
    if (row > 0 && col > 0) {
        for (int i = 0; i < row; ++i) {
            for (int j = 0; j < col; ++j) {
                sum += matrix[i][j];
            }
        }
    }
    else cout << "The matrix isn't initialized." << endl;
    return sum;
}
template<typename T>
T Matrix<T>::sum(int type, int axis) {
    T sum = 0;
    if (check_type_axis(type, axis)) {
        if (type == 0) {//row
            int r = axis;
            for (int i = 0; i < col; ++i) {
                sum += matrix[r][i];
            }
        }
        else {//type==1, col
            int c = axis;
            for (int i = 0; i < row; ++i) {
                sum += matrix[i][c];
            }
        }
    }
    return sum;
}
//average value
template<typename T>
T Matrix<T>::avg() {
    T s = sum();
    if (row > 0 && col > 0) {
        T ele = row * col;
        return s / ele;
    }
    else return 0;
}
template<typename T>
T Matrix<T>::avg(int type, int axis) {
    T s = sum(type, axis);
    if (row > 0 && col > 0) {
        if (type == 0) {//row
            T c = col;
            return s / c;
        }
        else {//type==1, col
            T r = row;
            return s / r;
        }
    }
    else return 0;
}

//check square
template<typename T>
bool Matrix<T>::check_square() {
    if (un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return false;
    }
    else {
        if (row == col) return true;
        else {
            cout << "The matrix isn't a square matrix." << endl;
            return false;
        }
    }
}
//trace
template<typename T>
T Matrix<T>::trace() {
    if (check_square()) {
        T tr = 0;
        for (int i = 0; i < row; ++i) {
            tr += matrix[i][i];
        }
        return tr;
    }
    else return 0;
}
//determinant
template<typename T>
T Matrix<T>::get_det(vector<vector<T>> m, int n) {
    if (n == 1) return m[0][0];
    T det = 0;
    vector<vector<T>> tmp;
    //initialize
    for (int j = 0; j < n - 1; ++j) {
        vector<T> r;
        for (int k = 0; k < n - 1; ++k) {
            r.emplace_back(0);
        }
        tmp.emplace_back(r);
    }
    int i;
    for (i = 0; i < n; ++i) {
        for (int j = 0; j < n - 1; ++j) {
            for (int k = 0; k < n - 1; ++k) {
                if (k >= i) tmp[j][k] = m[j + 1][k + 1];
                else tmp[j][k] = m[j + 1][k];
            }
        }
        T det_tmp = get_det(tmp, n - 1);
        if (i % 2 == 0) det += m[0][i] * det_tmp;
        else det -= m[0][i] * det_tmp;
    }
    return det;
}
template<typename T>
T Matrix<T>::determinant() {
    if (check_square()) {
        T det = get_det(matrix, row);
        return det;
    }
    else return 0;
}
//adjugate matrix 伴随矩阵 A*
template<typename T>
Matrix<T> Matrix<T>::adjugate() {
    if (check_square()) {
        int n = row;
        if (n == 1) {
            vector<vector<T>> m;
            vector<T> r;
            r.emplace_back(1);
            m.emplace_back(r);
            return Matrix<T>(m);
        }
        vector<vector<T>> adj;
        vector<vector<T>> tmp;
        //initialize
        for (int i = 0; i < n - 1; ++i) {
            vector<T> r;
            for (int j = 0; j < n - 1; ++j) {
                r.emplace_back(0);
            }
            tmp.emplace_back(r);
        }
        for (int i = 0; i < n; ++i) {
            vector<T> r;
            for (int j = 0; j < n; ++j) {
                r.emplace_back(0);
            }
            adj.emplace_back(r);
        }
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                for (int k = 0; k < n - 1; ++k) {
                    for (int l = 0; l < n - 1; ++l) {
                        int k1 = (k >= i) ? k + 1 : k;
                        int l1 = (l >= j) ? l + 1 : l;
                        tmp[k][l] = matrix[k1][l1];
                    }
                }
                adj[j][i] = get_det(tmp, n - 1);//转置
                if ((i + j) % 2 == 1) adj[j][i] = -adj[j][i];
            }
        }
        return Matrix<T>(adj);
    }
    else return Matrix<T>(0);
}
//inverse AA*=|A|I
template<typename T>
bool Matrix<T>::has_inverse() {
    if (check_square()) {
        T det = get_det(matrix, row);
        if (det == 0) {
            cout << "The determinant of the matrix is 0, and the matrix has no
inverse." << endl;
            return false;
        }
        else return true;
    }
    else return false;
}
template<typename T>
Matrix<T> Matrix<T>::inverse() {
    string T_str = typeid(T).name();
    if (check_df(T_str) == false) {
        cout << "Warning: T is not a floating point number and it will cause
precision loss, so it can not return correct result." << endl;
    }
    if (has_inverse()) {
        T det = get_det(matrix, row);
        Matrix<T> adj = adjugate();
        Matrix<T> inv = adj / det;
        return inv;
    }
    else return Matrix<T>(0);
}
//eigenvalue and eigenvector, QR分解
//2的范数
template<typename T>
T Matrix<T>::norm2() {
    if (un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return 0;
    }
    T norm = 0;
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            norm += matrix[i][j] * matrix[i][j];
        }
    }
    return sqrt(norm);
}
template<typename T>
T Matrix<T>::norm2(Matrix<T>& m) {
    if (m.un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return 0;
    }
    T norm = 0;
    for (int i = 0; i < m.row; ++i) {
        for (int j = 0; j < m.col; ++j) {
            norm += m.matrix[i][j] * m.matrix[i][j];
        }
    }
    return sqrt(norm);
}
template<typename T>
T Matrix<T>::norm2(vector<T>& v) {
    if (v.size() == 0) {
        cout << "The vector isn't initialized." << endl;
        return 0;
    }
    T norm = 0;
    for (int i = 0; i < v.size(); ++i) {
        norm += v[i] * v[i];
    }
    return sqrt(norm);
}
template<typename T>
void Matrix<T>::QR(Matrix<T>& A, Matrix<T>& Q, Matrix<T>& R) {
    int n = A.row;
    vector<T> a, b;
    for (int i = 0; i < n; ++i) {
        a.emplace_back(0);
        b.emplace_back(0);
    }
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            a[i] = A.matrix[i][j];
            b[i] = a[i];//第j列
        }
        for (int i = 0; i < j; ++i) {
            R.matrix[i][j] = 0;
            for (int k = 0; k < n; ++k) {
                R.matrix[i][j] += a[k] * Q.matrix[k][i];
            }
            for (int k = 0; k < n; ++k) {
                b[k] -= R.matrix[i][j] * Q.matrix[k][i];
            }
        }
        T norm = norm2(b);
        R.matrix[j][j] = norm;
        for (int i = 0; i < n; ++i) {
            Q.matrix[i][j] = b[i] / norm;
        }
    }
}
template<typename T>
vector<T> Matrix<T>::eigenvalues() {
    string T_str = typeid(T).name();
    if (check_df(T_str) == false) {
        cout << "Warning: T is not a floating point number and it will cause
precision loss, so it can not return correct result." << endl;
    }
    vector<T> evalue;
    if (check_square()) {
        if (determinant() != 0) {
            int n = row;
            const int accu = (1000 > n) ? 1000 : n;
            vector<vector<T>> tmp(n, vector<T>(n));
            vector<vector<T>> q(n, vector<T>(n));
            vector<vector<T>> r(n, vector<T>(n));
            tmp = matrix;
            Matrix<T> TMP(tmp);
            Matrix<T> Q(q);
            Matrix<T> R(r);
            for (int i = 0; i < accu; ++i) {
                QR(TMP, Q, R);
                TMP = R * Q;
            }
            for (int i = 0; i < n; ++i) {
                evalue.emplace_back(TMP.matrix[i][i]);
            }
            vector<T> evalue_copy;
            //去掉重复特征值
            sort(evalue.begin(), evalue.end());
            evalue_copy.emplace_back(evalue[0]);
            for (int i = 1; i < evalue.size(); ++i) {
                if (evalue[i] != evalue[i - 1]) {
                    evalue_copy.emplace_back(evalue[i]);
                }
            }
            return evalue_copy;
        }
        else {
            cout << "The determinant of the matrix is 0, eigenvalues can not be
found by QR decomposition." << endl;
            cout << "The matrix must have eigenvalue 0." << endl;
            evalue.emplace_back(0);
            return evalue;
        }
    }
    return evalue;
}
//eigenvectors
template<typename T>
vector<vector<T>> Matrix<T>::eigenvectors() {
    vector<vector<T>> evector;
    if (check_square()) {
        vector<T> evalue = eigenvalues();
        for (int i = 0; i < evalue.size(); ++i) {
            T e = evalue[i];
            //A-eI
            int n = row;
            Matrix<T> tmp(matrix);
            int zero_flag = 0;
            for (int j = 0; j < n; ++j) {
                tmp.matrix[j][j] -= e;
                if (tmp.matrix[j][j] == 0) zero_flag = 1;
            }
            if (zero_flag == 1) {
                cout << "The elements on the main diagonal of the matrix
contains 0 and can't use Gaussian method." << endl;
                return evector;
            }
            //高斯消元，将tmp化成上三角矩阵
            for (int j = 0; j < n - 1; ++j) {
                T md = tmp.matrix[j][j];
                for (int k = j; k < n; ++k) {
                    tmp.matrix[j][k] /= md;
                }
                for (int k = j + 1; k < n; ++k) {
                    T num = tmp.matrix[k][j];
                    for (int l = j; l < n; ++l) {
                        tmp.matrix[k][l] -= num * tmp.matrix[j][l];
                    }
                }
            }
            vector<T> ev;//单位向量
            //initialize 1*n
            for (int j = 0; j < n; ++j) {
                ev.emplace_back(0);
            }
            //A*ev=0 从最后一行算到第一行给ev[j]赋值
            ev[n - 1] = 1;
            T ev_sum = 1;
            for (int j = n - 2; j >= 0; --j) {
                T sum = 0;
                for (int k = j + 1; k < n; ++k) {
                    sum += tmp.matrix[j][k] * ev[k];
                }
                ev[j] = -sum / tmp.matrix[j][j];
                ev_sum += ev[j] * ev[j];
            }
            ev_sum = sqrt(ev_sum);
            for (int j = 0; j < n; ++j) {
                ev[j] /= ev_sum;
            }
            evector.emplace_back(ev);
        }
        return evector;
    }
    return evector;
}
template<typename T>
void Matrix<T>::print_eigenvectors() {
    vector<vector<T>> evector = eigenvectors();
    for (int i = 0; i < evector.size(); ++i) {
        cout << "[";
        for (int j = 0; j < evector[0].size() - 1; ++j) {
            cout << evector[i][j] << ",";
        }
        cout << evector[i][evector[0].size() - 1] << "]" << endl;
    }
}
//reshape 整型
template<typename T>
Matrix<T> Matrix<T>::reshape(Matrix<T> m, int r, int c) {
    if (m.un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return Matrix<T>(0);
    }
    vector<vector<T>> res;//r*c
    if (m.row * m.col == r * c) {
        //initialize
        for (int i = 0; i < r; ++i) {
            vector<T> ro;
            for (int j = 0; j < c; ++j) {
                ro.emplace_back(0);
            }
            res.emplace_back(ro);
        }
        for (int i = 0; i < r * c; ++i) {
            res[i % r][i / r] = m.matrix[i % m.row][i / m.row];
        }
        return Matrix<T>(res);
    }
    else {
        cout << "The sizes of row and column of new matrix are mismatched." <<
endl;
        return Matrix<T>(0);
    }
}
template<typename T>
Matrix<T> Matrix<T>::reshape(int r, int c) {
    return reshape(Matrix<T>(matrix), r, c);
}
//slicing 切片
template<typename T>
Matrix<T> Matrix<T>::slicing(Matrix<T> m, int row_begin, int row_end, int
col_begin, int col_end, int row_interval, int col_interval) {
    if (m.un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return Matrix<T>(0);
    }
    if ((0 <= row_begin && row_begin <= row_end && row_end < m.row) && (0 <=
col_begin && col_begin <= col_end && col_end < m.col)
        && (0 <= row_interval && row_interval < row_end - row_begin + 1) && (0
<= col_interval && col_interval < col_end - col_begin + 1)) {
        Matrix<T> sli_1(row_end - row_begin + 1, col_end - col_begin + 1);
        for (int i = 0; i < sli_1.row; ++i) {
            for (int j = 0; j < sli_1.col; ++j) {
                sli_1.matrix[i][j] = m.matrix[row_begin + i][col_begin + j];
            }
        }
        int row_period = row_interval + 1;
        int col_period = col_interval + 1;
        int row_time = sli_1.row / row_period;
        int col_time = sli_1.col / col_period;
        int sli_2_row = ((sli_1.row % row_period == 0)) ? row_time : (row_time +
1);
        int sli_2_col = ((sli_1.col % col_period) == 0) ? col_time : (col_time +
1);
        Matrix<T> sli_2(sli_2_row, sli_2_col);
        int i = 0, j = 0, k = 0, t = 0;
        for (i = 0, k = 0; i < sli_2_row; ++i, k = k + row_period) {
            for (j = 0, t = 0; j < sli_2_col; ++j, t = t + col_period) {
                sli_2.matrix[i][j] = sli_1.matrix[k][t];
            }
        }
        return sli_2;
    }
    else {
        cout << "The range of input is error." << endl;
        return Matrix<T>(0);
    }
}
template<typename T>
Matrix<T> Matrix<T>::row_slicing(Matrix<T> m, int row_begin, int row_end, int
interval) {
    return slicing(m, row_begin, row_end, 0, m.col - 1, interval, 0);
}
template<typename T>
Matrix<T> Matrix<T>::col_slicing(Matrix<T> m, int col_begin, int col_end, int
interval) {
    return slicing(m, 0, m.row - 1, col_begin, col_end, 0, interval);
```

```cpp
}
template<typename T>
Matrix<T> Matrix<T>::slicing(int row_begin, int row_end, int col_begin, int
col_end, int row_interval, int col_interval) {
    return slicing(Matrix<T>(matrix), row_begin, row_end, col_begin, col_end,
row_interval, col_interval);
}
template<typename T>
Matrix<T> Matrix<T>::row_slicing(int row_begin, int row_end, int interval) {
    return row_slicing(Matrix<T>(matrix), row_begin, row_end, interval);
}
template<typename T>
Matrix<T> Matrix<T>::col_slicing(int col_begin, int col_end, int interval) {
    return col_slicing(Matrix<T>(matrix), col_begin, col_end, interval);
}
//convolution 卷积
template<typename T>
Matrix<T> Matrix<T>::convolution(Matrix<T> m1, Matrix<T> m2) {
    if (m1.un_initialized() || m2.un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return Matrix<T>(0);
    }
    Matrix<T> big;
    Matrix<T> small;
    big = (m1.row * m1.col > m2.row * m2.col) ? m1 : m2;
    small = (m1.row * m1.col <= m2.row * m2.col) ? m1 : m2;
    Matrix<T> extend(big.row + 2 * (small.row - 1), big.col + 2 * (small.col -
1));
    Matrix<T> full(big.row + small.row - 1, big.col + small.col - 1);
    Matrix<T> small_180(small.row, small.col);
    for (int i = 0; i < small.row; ++i) {
        for (int j = 0; j < small.col; ++j) {
            small_180.matrix[small.row - 1 - i][small.col - 1 - j] =
small.matrix[i][j];
        }
    }
    for (int i = 0; i < big.row; ++i) {
        for (int j = 0; j < big.col; ++j) {
            extend.matrix[small.row - 1 + i][small.col - 1 + j] = big.matrix[i]
[j];
        }
    }
    //small_180矩阵起点在extend矩阵的位置，对应full矩阵的i,j
    for (int i = 0; i < full.row; ++i) {
        for (int j = 0; j < full.col; ++j) {
            T sum = 0;
            for (int k = 0; k < small_180.row; ++k) {
                for (int l = 0; l < small_180.col; ++l) {
                    sum += small_180.matrix[k][l] * extend.matrix[i + k][j + l];
                }
            }
            full.matrix[i][j] = sum;
        }
    }
    return full;
}

//Matrix<complex<D>>
```

```
//检查初始化
template<typename D>
bool Matrix<complex<D>>::un_initialized() {
    if (row == 0 || col == 0)return true;
    else return false;
}
//overload cout<<
template<typename D>
ostream& operator<<(ostream& os, const Matrix<complex<D>>& other) {
    if (other.row == 0 || other.col == 0) {
        os << "The matrix isn't initialized." << endl;
    }
    else {
        for (int i = 0; i < other.row; ++i) {
            for (int j = 0; j < other.col; ++j) {
                os << other.matrix[i][j] << " ";
            }
            os << endl;
        }
    }
    return os;
}
template<typename D>
ostream& operator<<(ostream& os, const vector<complex<D>>& other) {
    if (other.size() == 0) {
        os << "The vector isn't initialized." << endl;
    }
    else {
        os << "[";
        for (int i = 0; i < other.size() - 1; ++i) {
            os << other[i] << ",";
        }
        os << other[other.size() - 1] << "]";
    }
    return os;
}
//overload =
template<typename D>
Matrix<complex<D>>& Matrix<complex<D>>::operator=(const Matrix<complex<D>>&
next) {
    if (this != &next) {
        this->row = next.row;
        this->col = next.col;
        this->matrix = next.matrix;
    }
    return *this;
}
//A+B
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::operator+(const Matrix<complex<D>>& next)
{
    try {
        if (un_initialized() || next.row == 0 || next.col == 0) {
            throw 0;
        }
        vector<vector<complex<D>>> add;
        //矩阵大小匹配
        if (this->row == next.row && this->col == next.col) {
            for (int i = 0; i < this->row; ++i) {
                vector<complex<D>> r;
                for (int j = 0; j < this->col; ++j) {
                    r.emplace_back(this->matrix[i][j] + next.matrix[i][j]);
                }
                add.emplace_back(r);
            }
            return Matrix<complex<D>>(add);
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a)
    {
        //矩阵大小不匹配
        cout << "The sizes of matrices are mismatched." << endl;
    }
}
//v1+v2
template<typename D>
vector<complex<D>> operator+(const vector<complex<D>>& v1, const
vector<complex<D>>& v2) {
    try {
        vector<complex<D>> v;
        if (v1.size() == v2.size()) {
            for (int i = 0; i < v1.size(); ++i) {
                v.emplace_back(v1[i] + v2[i]);
            }
            return v;
        }
        throw 0;
    }
    catch (int i)
    {
        cout << "The sizes of vectors are mismatched." << endl;
    }
}
//A-B
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::operator-(const Matrix<complex<D>>& next)
{
    try {
        if (un_initialized() || next.row == 0 || next.col == 0) {
            throw 0;
        }
        vector<vector<complex<D>>> sub;
        //矩阵大小匹配
        if (this->row == next.row && this->col == next.col) {
            for (int i = 0; i < this->row; ++i) {
                vector<complex<D>> r;
                for (int j = 0; j < this->col; ++j) {
                    r.emplace_back(this->matrix[i][j] - next.matrix[i][j]);
                }
                sub.emplace_back(r);
            }
```

```cpp
            return Matrix<complex<D>>(sub);
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a)
    {
        //矩阵大小不匹配
        cout << "The sizes of matrices are mismatched." << endl;
    }
}
//v1-v2
template<typename D>
vector<complex<D>> operator-(vector<complex<D>>& v1, vector<complex<D>>& v2) {
    try {
        if (v1.size() == 0 || v2.size() == 0) {
            throw 0;
        }
        vector<complex<D>> v;
        if (v1.size() == v2.size()) {
            for (int i = 0; i < v1.size(); ++i) {
                v.emplace_back(v1[i] - v2[i]);
            }
            return v;
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The vector isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of vectors are mismatched." << endl;
    }
}
//scalar multiplication
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::operator*(complex<D> val) {
    try {
        if (un_initialized()) {
            throw 0;
        }
        vector<vector<complex<D>>> mul;
        for (int i = 0; i < this->row; ++i) {
            vector<complex<D>> r;
            for (int j = 0; j < this->col; ++j) {
                this->matrix[i][j] *= val;
                r.emplace_back(this->matrix[i][j] * val);
            }
            mul.emplace_back(r);
        }
        return Matrix<complex<D>>(mul);
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
}
template<typename D>
Matrix<complex<D>> operator*(complex<D> val, const Matrix<complex<D>>& other) {
    try {
        if (other.row == 0 || other.col == 0) {
            throw 0;
        }
        vector<vector<complex<D>>> mul;
        for (int i = 0; i < other.row; ++i) {
            vector<complex<D>> r;
            for (int j = 0; j < other.col; ++j) {
                r.emplace_back(val * other.matrix[i][j]);
            }
            mul.emplace_back(r);
        }
        return Matrix<complex<D>>(mul);
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }

}
template<typename D>
vector<complex<D>> operator*(vector<complex<D>>& v, complex<D> val) {
    try {
        if (v.size() == 0) {
            throw 0;
        }
        vector<complex<D>> mul;
        for (int i = 0; i < v.size(); ++i) {
            mul.emplace_back(v[i] * val);
        }
        return mul;
    }
    catch (int i)
    {
        cout << "The vector isn't initialized." << endl;
    }

}
template<typename D>
vector<complex<D>> operator*(complex<D> val, vector<complex<D>>& v) {
    try {
        if (v.size() == 0) {
            throw 0;
        }
        vector<complex<D>> mul;
        for (int i = 0; i < v.size(); ++i) {
            mul.emplace_back(v[i] * val);
        }
        return mul;
    }
    catch (int i)
    {
```

```cpp
        cout << "The vector isn't initialized." << endl;
    }

}
//scalar division
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::operator/(complex<D> val) {
    try {
        if (un_initialized()) {
            throw 0;
        }
        vector<vector<complex<D>>> div;
        for (int i = 0; i < this->row; ++i) {
            vector<complex<D>> r;
            for (int j = 0; j < this->col; ++j) {
                r.emplace_back(this->matrix[i][j] / val);
            }
            div.emplace_back(r);
        }
        return Matrix<complex<D>>(div);
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }

}
template<typename D>
vector<complex<D>> operator/(vector<complex<D>>& v, complex<D> val) {
    try {
        if (v.size() == 0) {
            throw 0;
        }
        vector<complex<D>> div;
        for (int i = 0; i < v.size(); ++i) {
            div.emplace_back(v[i] / val);
        }
        return div;
    }
    catch (int i)
    {
        cout << "The vector isn't initialized." << endl;
    }

}
//transposition
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::transposition(const Matrix<complex<D>>&
m) {
    try {
        if (m.row == 0 || m.col == 0) {
            throw 0;
        }
        vector<vector<complex<D>>> trans;
        for (int i = 0; i < m.col; ++i) {
            vector<complex<D>> trans_r;
            for (int j = 0; j < m.row; ++j) {
                trans_r.emplace_back(m.matrix[j][i]);
            }
            trans.emplace_back(trans_r);
        }
        return Matrix<complex<D>>(trans);
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
}
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::transposition() {
    try {
        if (un_initialized()) {
            throw 0;
        }
        vector<vector<complex<D>>> trans;
        for (int i = 0; i < this->col; ++i) {
            vector<complex<D>> trans_r;
            for (int j = 0; j < this->row; ++j) {
                trans_r.emplace_back(this->matrix[j][i]);
            }
            trans.emplace_back(trans_r);
        }
        return Matrix<complex<D>>(trans);
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }

};
//conjugation 共轭
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::conjugation(const Matrix<complex<D>>& m)
{
    Matrix<complex<D>> coj = transposition(m);
    for (int i = 0; i < coj.row; ++i) {
        for (int j = 0; j < coj.col; ++j) {
            coj.matrix[i][j] = conj(coj.matrix[i][j]);
        }
    }
    return coj;
}
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::conjugation() {
    Matrix<complex<D>> coj = transposition();
    for (int i = 0; i < coj.row; ++i) {
        for (int j = 0; j < coj.col; ++j) {
            coj.matrix[i][j] = conj(coj.matrix[i][j]);
        }
    }
    return coj;
}
//element-wise multiplication
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::element_wise_mul(const
Matrix<complex<D>>& m) {
    try {
        if (m.row == 0 || m.col == 0 || row == 0 || col == 0) {
            throw 0;
        }
        vector<vector<complex<D>>> mul;
        if (row == m.row && col == m.col) {
            for (int i = 0; i < row; ++i) {
                vector<complex<D>> r;
                for (int j = 0; j < col; ++j) {
                    r.emplace_back(matrix[i][j] * m.matrix[i][j]);
                }
                mul.emplace_back(r);
            }
            return Matrix<complex<D>>(mul);
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of matrices are mismatched." << endl;
    }

}
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::element_wise_mul(const
Matrix<complex<D>>& m1, const Matrix<complex<D>>& m2) {
    try {
        if (m1.row == 0 || m1.col == 0 || m2.row == 0 || m2.col == 0) {
            throw 0;
        }
        vector<vector<complex<D>>> mul;
        if (m1.row == m2.row && m1.col == m2.col) {
            for (int i = 0; i < m1.row; ++i) {
                vector<complex<D>> r;
                for (int j = 0; j < m1.col; ++j) {
                    r.emplace_back(m1.matrix[i][j] * m2.matrix[i][j]);
                }
                mul.emplace_back(r);
            }
            return Matrix<complex<D>>(mul);
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of matrices are mismatched." << endl;
    }
}
template<typename D>
```

```
vector<complex<D>> Matrix<complex<D>>::element_wise_mul(vector<complex<D>>& v1,
vector<complex<D>>& v2) {
    try {
        if (v1.size() == 0 || v2.size() == 0) {
            throw 0;
        }
        vector<complex<D>> mul;
        if (v1.size() == v2.size()) {
            for (int i = 0; i < v1.size(); ++i) {
                mul.emplace_back(v1[i] * v2[i]);
            }
            return mul;
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The vector isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of vectors are mismatched." << endl;
    }
}
//matrix-matrix multiplication
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::operator*(const Matrix<complex<D>>& m) {
    try {
        if (row == 0 || col == 0 || m.row == 0 || m.col == 0) {
            throw 0;
        }
        vector<vector<complex<D>>> mul;
        if (col == m.row) {
            //initialize
            for (int i = 0; i < row; ++i) {
                vector<complex<D>> r;
                for (int j = 0; j < m.col; ++j) {
                    r.emplace_back(0);
                }
                mul.emplace_back(r);
            }
            for (int i = 0; i < row; ++i) {
                for (int j = 0; j < m.col; ++j) {
                    for (int k = 0; k < col; ++k) {
                        mul[i][j] += matrix[i][k] * m.matrix[k][j];
                    }
                }
            }
            return Matrix<complex<D>>(mul);
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of matrices are mismatched." << endl;
    }
}
//matrix-vector multiplication
template<typename D>
vector<complex<D>> Matrix<complex<D>>::operator*(vector<complex<D>>& vec) {
    try {
        if (row == 0 || col == 0 || vec.size() == 0) {
            throw 0;
        }
        vector<complex<D>> v;
        if (col == vec.size()) {
            //initialize
            for (int i = 0; i < row; ++i) {
                v.emplace_back(0);
            }
            for (int i = 0; i < row; ++i) {
                for (int j = 0; j < col; ++j) {
                    v[i] += matrix[i][j] * vec[j];
                }
            }
            return v;
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix or the vector isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The size of matrix and the size of vector are mismatched." <<
endl;
    }
}
template<typename D>
vector<complex<D>> operator*(vector<complex<D>>& vec, const Matrix<complex<D>>&
other) {
    try {
        if (other.row == 0 || other.col == 0 || vec.size() == 0) {
            throw 0;
        }
        vector<complex<D>> v;
        if (vec.size() == other.row) {
            //initialize
            for (int i = 0; i < other.col; ++i) {
                v.emplace_back(0);
            }
            for (int i = 0; i < other.col; ++i) {
                for (int j = 0; j < other.row; ++j) {
                    v[i] += other.matrix[j][i] * vec[j];
                }
            }
            return v;
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix or the vector isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The size of matrix and the size of vector are mismatched." <<
endl;
    }
}
template<typename D>
complex<D> operator*(vector<complex<D>> v1, vector<complex<D>> v2) {
    try {
        if (v1.size() == 0 || v2.size() == 0) {
            throw 0;
        }
        complex<D> mul = 0;
        if (v1.size() == v2.size()) {
            for (int i = 0; i < v1.size(); ++i) {
                mul += v1[i] * v2[i];
            }
            return mul;
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The vector isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of vectors are mismatched." << endl;
    }
}
//dot product 内积
template<typename D>
complex<D> Matrix<complex<D>>::dot_product(Matrix<complex<D>>& m1,
Matrix<complex<D>>& m2) {
    try {
        if (m1.un_initialized() || m2.un_initialized()) {
            throw 0;
        }
        complex<D> dot_prod = 0;
        if (m1.row == m2.row && m1.col == m2.col) {
            for (int i = 0; i < m1.row; ++i) {
                for (int j = 0; j < m1.col; ++j) {
                    dot_prod += m1.matrix[i][j] * m2.matrix[i][j];
                }
            }
            return dot_prod;
        }
        throw 'a';
    }
    catch (int i) {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a) {
        cout << "The sizes of matrices are mismatched." << endl;
    }
```

```
}
template<typename D>
complex<D> Matrix<complex<D>>::dot_product(Matrix<complex<D>>& m) {
    try {
        if (m.un_initialized() || un_initialized()) {
            throw 0;
        }
        complex<D> dot_prod = 0;
        if (row == m.row && col == m.col) {
            for (int i = 0; i < row; ++i) {
                for (int j = 0; j < col; ++j) {
                    dot_prod += matrix[i][j] * m.matrix[i][j];
                }
            }
            return dot_prod;
        }
        throw 'a';
    }
    catch (int i)
    {
        cout << "The matrix isn't initialized." << endl;
    }
    catch (char a)
    {
        cout << "The sizes of matrices are mismatched." << endl;
    }
}
template<typename D>
complex<D> Matrix<complex<D>>::dot_product(vector<complex<D>>& v1,
vector<complex<D>>& v2) {
    return v1 * v2;
}
//cross product n=3
template<typename D>
vector<complex<D>> Matrix<complex<D>>::cross_product(vector<complex<D>>& v1,
vector<complex<D>>& v2) {
    try {
        vector<complex<D>> prod;
        if (v1.size() == 3 && v2.size() == 3) {
            prod.emplace_back(v1[1] * v2[2] - v1[2] * v2[1]);
            prod.emplace_back(v1[2] * v2[0] - v1[0] * v2[2]);
            prod.emplace_back(v1[0] * v2[1] - v1[1] * v2[0]);
            return prod;
        }
        throw 0;
    }
    catch (int i)
    {
        cout << "The sizes of vectors are not 3 and have no cross product." <<
endl;
    }
}
//check type and axis
template<typename D>
bool Matrix<complex<D>>::check_type_axis(int type, int axis) {
    if (un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
```

```
            return false;
        }
        else {
            if (type == 0) {//row
                if (axis >= 0 && axis < row) return true;
                else {
                    cout << "Invalid axis." << endl;
                    return false;
                }
            }
            else if (type == 1) {//col
                if (axis >= 0 && axis < col) return true;
                else {
                    cout << "Invalid axis." << endl;
                    return false;
                }
            }
            else {
                cout << "Invalid type." << endl;
                return false;
            }
        }
}
//sum
template<typename D>
complex<D> Matrix<complex<D>>::sum() {
    complex<D> sum = 0;
    if (row > 0 && col > 0) {
        for (int i = 0; i < row; ++i) {
            for (int j = 0; j < col; ++j) {
                sum += matrix[i][j];
            }
        }
    }
    else cout << "The matrix isn't initialized." << endl;
    return sum;
}
template<typename D>
complex<D> Matrix<complex<D>>::sum(int type, int axis) {
    complex<D> sum = 0;
    if (check_type_axis(type, axis)) {
        if (type == 0) {//row
            int r = axis;
            for (int i = 0; i < col; ++i) {
                sum += matrix[r][i];
            }
        }
        else {//type==1, col
            int c = axis;
            for (int i = 0; i < row; ++i) {
                sum += matrix[i][c];
            }
        }
    }
    return sum;
}
//average value
template<typename D>
```

```cpp
complex<D> Matrix<complex<D>>::avg() {
    complex<D> s = sum();
    if (row > 0 && col > 0) {
        complex<D> ele = row * col;
        return s / ele;
    }
    else return 0;
}
template<typename D>
complex<D> Matrix<complex<D>>::avg(int type, int axis) {
    complex<D> s = sum(type, axis);
    if (row > 0 && col > 0) {
        if (type == 0) {//row
            complex<D> c = col;
            return s / c;
        }
        else {//type==1, col
            complex<D> r = row;
            return s / r;
        }
    }
    else return 0;
}

//check square
template<typename D>
bool Matrix<complex<D>>::check_square() {
    if (un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return false;
    }
    else {
        if (row == col) return true;
        else {
            cout << "The matrix isn't a square matrix." << endl;
            return false;
        }
    }
}
//trace
template<typename D>
complex<D> Matrix<complex<D>>::trace() {
    if (check_square()) {
        complex<D> tr = 0;
        for (int i = 0; i < row; ++i) {
            tr += matrix[i][i];
        }
        return tr;
    }
    else return 0;
}
//determinant
template<typename D>
complex<D> Matrix<complex<D>>::get_det(vector<vector<complex<D>>> m, int n) {
    if (n == 1) return m[0][0];
    complex<D> det = 0;
    vector<vector<complex<D>>> tmp;
    //initialize
    for (int j = 0; j < n - 1; ++j) {
        vector<complex<D>> r;
        for (int k = 0; k < n - 1; ++k) {
            r.emplace_back(0);
        }
        tmp.emplace_back(r);
    }
    int i;
    for (i = 0; i < n; ++i) {
        for (int j = 0; j < n - 1; ++j) {
            for (int k = 0; k < n - 1; ++k) {
                if (k >= i) tmp[j][k] = m[j + 1][k + 1];
                else tmp[j][k] = m[j + 1][k];
            }
        }
        complex<D> det_tmp = get_det(tmp, n - 1);
        if (i % 2 == 0) det += m[0][i] * det_tmp;
        else det -= m[0][i] * det_tmp;
    }
    return det;
}
template<typename D>
complex<D> Matrix<complex<D>>::determinant() {
    if (check_square()) {
        complex<D> det = get_det(matrix, row);
        return det;
    }
    else return 0;
}
//adjugate matrix 伴随矩阵 A*
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::adjugate() {
    if (check_square()) {
        int n = row;
        if (n == 1) {
            vector<vector<complex<D>>> m;
            vector<complex<D>> r;
            r.emplace_back(1);
            m.emplace_back(r);
            return Matrix<complex<D>>(m);
        }
        vector<vector<complex<D>>> adj;
        vector<vector<complex<D>>> tmp;
        //initialize
        for (int i = 0; i < n - 1; ++i) {
            vector<complex<D>> r;
            for (int j = 0; j < n - 1; ++j) {
                r.emplace_back(0);
            }
            tmp.emplace_back(r);
        }
        for (int i = 0; i < n; ++i) {
            vector<complex<D>> r;
            for (int j = 0; j < n; ++j) {
                r.emplace_back(0);
            }
            adj.emplace_back(r);
        }
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                for (int k = 0; k < n - 1; ++k) {
                    for (int l = 0; l < n - 1; ++l) {
                        int k1 = (k >= i) ? k + 1 : k;
                        int l1 = (l >= j) ? l + 1 : l;
                        tmp[k][l] = matrix[k1][l1];
                    }
                }
                adj[j][i] = get_det(tmp, n - 1);//转置
                if ((i + j) % 2 == 1) adj[j][i] = -adj[j][i];
            }
        }
        return Matrix<complex<D>>(adj);
    }
    else return Matrix<complex<D>>(0);
}
//inverse AA*=|A|I
template<typename D>
bool Matrix<complex<D>>::has_inverse() {
    if (check_square()) {
        complex<D> det = get_det(matrix, row);
        if (det == 0) {
            cout << "The determinant of the matrix is 0, and the matrix has no
inverse." << endl;
            return false;
        }
        else return true;
    }
    else return false;
}
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::inverse() {
    string T_str = typeid(complex<D>).name();
    if (has_inverse()) {
        complex<D> det = get_det(matrix, row);
        Matrix<complex<D>> adj = adjugate();
        Matrix<complex<D>> inv = adj / det;
        return inv;
    }
    else return Matrix<complex<D>>(0);
}
//2的范数
template<typename D>
D Matrix<complex<D>>::norm2() {
    if (un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return 0;
    }
    D norm = 0;
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            norm += matrix[i][j].imag() * matrix[i][j].imag() + matrix[i]
[j].real() * matrix[i][j].real();//|x|^2
        }
    }
    return sqrt(norm);
}
```

```
template<typename D>
D Matrix<complex<D>>::norm2(Matrix<complex<D>>& m) {
    if (m.un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return 0;
    }
    D norm = 0;
    for (int i = 0; i < m.row; ++i) {
        for (int j = 0; j < m.col; ++j) {
            norm += m.matrix[i][j].imag() * m.matrix[i][j].imag() + m.matrix[i]
[j].real() * m.matrix[i][j].real();
        }
    }
    return sqrt(norm);
}
template<typename D>
D Matrix<complex<D>>::norm2(vector<complex<D>>& v) {
    if (v.size() == 0) {
        cout << "The vector isn't initialized." << endl;
        return 0;
    }
    D norm = 0;
    for (int i = 0; i < v.size(); ++i) {
        norm += v[i].imag() * v[i].imag() + v[i].real() * v[i].real();
    }
    return sqrt(norm);
}
//reshape 整型
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::reshape(Matrix<complex<D>> m, int r, int
c) {
    if (m.un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return Matrix<complex<D>>(0);
    }
    vector<vector<complex<D>>> res;//r*c
    if (m.row * m.col == r * c) {
        //initialize
        for (int i = 0; i < r; ++i) {
            vector<complex<D>> ro;
            for (int j = 0; j < c; ++j) {
                ro.emplace_back(0);
            }
            res.emplace_back(ro);
        }
        for (int i = 0; i < r * c; ++i) {
            res[i % r][i / r] = m.matrix[i % m.row][i / m.row];
        }
        return Matrix<complex<D>>(res);
    }
    else {
        cout << "The sizes of row and column of new matrix are mismatched." <<
endl;
        return Matrix<complex<D>>(0);
    }
}
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::reshape(int r, int c) {
    return reshape(Matrix<complex<D>>(matrix), r, c);
}
//slicing 切片
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::slicing(Matrix<complex<D>> m, int
row_begin, int row_end, int col_begin, int col_end, int row_interval, int
col_interval) {
    if (m.un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return Matrix<complex<D>>(0);
    }
    if ((0 <= row_begin && row_begin <= row_end && row_end < m.row) && (0 <=
col_begin && col_begin <= col_end && col_end < m.col)
        && (0 <= row_interval && row_interval < row_end - row_begin + 1) && (0
<= col_interval && col_interval < col_end - col_begin + 1)) {
        Matrix<complex<D>> sli_1(row_end - row_begin + 1, col_end - col_begin +
1);
        for (int i = 0; i < sli_1.row; ++i) {
            for (int j = 0; j < sli_1.col; ++j) {
                sli_1.matrix[i][j] = m.matrix[row_begin + i][col_begin + j];
            }
        }
        int row_period = row_interval + 1;
        int col_period = col_interval + 1;
        int row_time = sli_1.row / row_period;
        int col_time = sli_1.col / col_period;
        int sli_2_row = ((sli_1.row % row_period == 0)) ? row_time : (row_time +
1);
        int sli_2_col = ((sli_1.col % col_period) == 0) ? col_time : (col_time +
1);
        Matrix<complex<D>> sli_2(sli_2_row, sli_2_col);
        int i = 0, j = 0, k = 0, t = 0;
        for (i = 0, k = 0; i < sli_2_row; ++i, k = k + row_period) {
            for (j = 0, t = 0; j < sli_2_col; ++j, t = t + col_period) {
                sli_2.matrix[i][j] = sli_1.matrix[k][t];
            }
        }
        return sli_2;
    }
    else {
        cout << "The range of input is error." << endl;
        return Matrix<complex<D>>(0);
    }
}
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::row_slicing(Matrix<complex<D>> m, int
row_begin, int row_end, int interval) {
    return slicing(m, row_begin, row_end, 0, m.col - 1, interval, 0);
}
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::col_slicing(Matrix<complex<D>> m, int
col_begin, int col_end, int interval) {
    return slicing(m, 0, m.row - 1, col_begin, col_end, 0, interval);
}
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::slicing(int row_begin, int row_end, int
col_begin, int col_end, int row_interval, int col_interval) {
    return slicing(Matrix<complex<D>>(matrix), row_begin, row_end, col_begin,
col_end, row_interval, col_interval);
}
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::row_slicing(int row_begin, int row_end,
int interval) {
    return row_slicing(Matrix<complex<D>>(matrix), row_begin, row_end,
interval);
}
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::col_slicing(int col_begin, int col_end,
int interval) {
    return col_slicing(Matrix<complex<D>>(matrix), col_begin, col_end,
interval);
}
//convolution 卷积
template<typename D>
Matrix<complex<D>> Matrix<complex<D>>::convolution(Matrix<complex<D>> m1,
Matrix<complex<D>> m2) {
    if (m1.un_initialized() || m2.un_initialized()) {
        cout << "The matrix isn't initialized." << endl;
        return Matrix<complex<D>>(0);
    }
    Matrix<complex<D>> big;
    Matrix<complex<D>> small;
    big = (m1.row * m1.col > m2.row * m2.col) ? m1 : m2;
    small = (m1.row * m1.col <= m2.row * m2.col) ? m1 : m2;
    Matrix<complex<D>> extend(big.row + 2 * (small.row - 1), big.col + 2 *
(small.col - 1));
    Matrix<complex<D>> full(big.row + small.row - 1, big.col + small.col - 1);
    Matrix<complex<D>> small_180(small.row, small.col);
    for (int i = 0; i < small.row; ++i) {
        for (int j = 0; j < small.col; ++j) {
            small_180.matrix[small.row - 1 - i][small.col - 1 - j] =
small.matrix[i][j];
        }
    }
    for (int i = 0; i < big.row; ++i) {
        for (int j = 0; j < big.col; ++j) {
            extend.matrix[small.row - 1 + i][small.col - 1 + j] = big.matrix[i]
[j];
        }
    }
    //small_180矩阵起点在extend矩阵的位置，对应full矩阵的i,j
    for (int i = 0; i < full.row; ++i) {
        for (int j = 0; j < full.col; ++j) {
            complex<D> sum = 0;
            for (int k = 0; k < small_180.row; ++k) {
                for (int l = 0; l < small_180.col; ++l) {
                    sum += small_180.matrix[k][l] * extend.matrix[i + k][j + l];
                }
            }
            full.matrix[i][j] = sum;
        }
    }
    return full;
}
```

Main.cpp

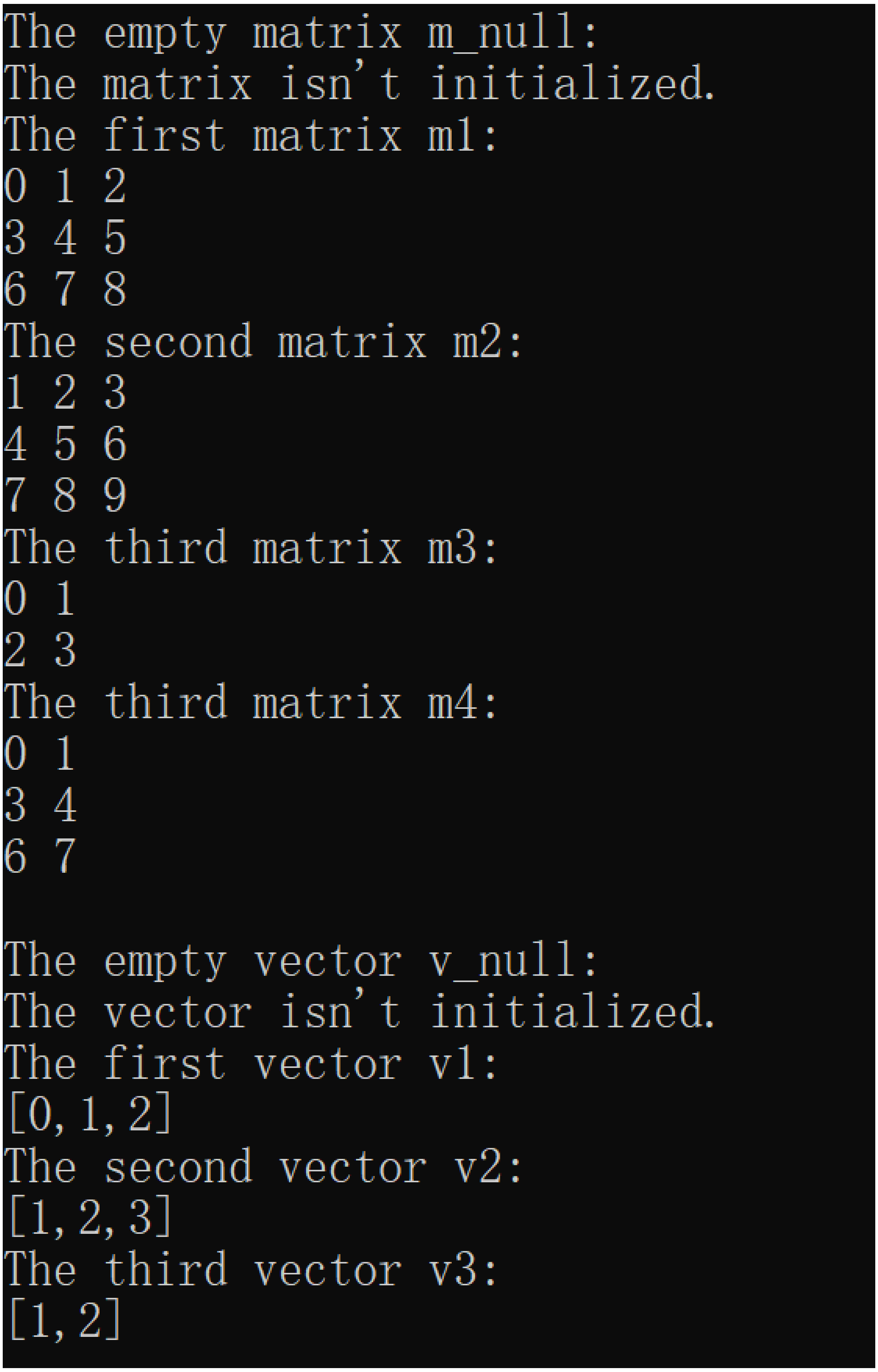
```cpp
#include <iostream>
#include <vector>
#include "Matrix.cpp"
using namespace std;

int main() {
    int n, m;
    n = 3;
    m = 2;
    vector<vector<double>> vv1(n, vector<double>(n));
    vector<vector<double>> vv2(n, vector<double>(n));
    vector<vector<double>> vv3(m, vector<double>(m));
    vector<vector<double>> vv4(n, vector<double>(m));
    vector<double> v1(n);
    vector<double> v2(n);
    vector<double> v3(m);
    vector<double> v_null;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            vv1[i][j] = (double)i * n + j;
        }
        v1[i] = (double)i;
    }
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            vv2[i][j] = (double)i * n + j + 1;
        }
        v2[i] = (double)i + 1;
    }

    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < m; j++)
        {
            vv3[i][j] = (double)i * m + j;
        }
        v3[i] = (double)i + 1;
    }
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            vv4[i][j] = (double)i * n + j;
        }
    }

    Matrix<double> m1(vv1);
    Matrix<double> m2(vv2);
    Matrix<double> m3(vv3);
    Matrix<double> m4(vv4);
    Matrix<double> m_null;

    // <<
    cout << "The empty matrix m_null:" << endl;
    cout << m_null;
    cout << "The first matrix m1:" << endl;
    cout << m1;
    cout << "The second matrix m2:" << endl;
    cout << m2;
    cout << "The third matrix m3:" << endl;
    cout << m3;
    cout << "The third matrix m4:" << endl;
    cout << m4;
    cout << endl;
    cout << "The empty vector v_null:" << endl;
    cout << v_null;
    cout << "The first vector v1:" << endl;
    cout << v1;
    cout << "The second vector v2:" << endl;
    cout << v2;
    cout << "The third vector v3:" << endl;
    cout << v3;

    cout << endl;

    // +
    cout << "m1 + m_null:" << endl;
    Matrix<double> m_plus1 = m1 + m_null;
    cout << "m1 + m3:" << endl;
    Matrix<double> m_plus2 = m1 + m3;
    cout << "m1 + m2:" << endl;
    Matrix<double> m_plus3 = m1 + m2;
    cout << m_plus3;
    cout << endl;
    cout << "v1 + v_null:" << endl;
    vector<double> v_plus1 = v1 + v_null;
    cout << "v1 + v3:" << endl;
    vector<double> v_plus2 = v1 + v3;
    cout << "v1 + v2:" << endl;
    vector<double> v_plus3 = v1 + v2;
    cout << v_plus3;

    cout << endl;

    // -
    cout << "m1 - m_null:" << endl;
    Matrix<double> m_minus1 = m1 - m_null;
    cout << "m1 - m3:" << endl;
    Matrix<double> m_minus2 = m1 - m3;
    cout << "m1 - m2:" << endl;
    Matrix<double> m_minus3 = m1 - m2;
    cout << m_minus3;
    cout << endl;
    cout << "v1 - v_null:" << endl;
    vector<double> v_minus1 = v1 - v_null;
    cout << "v1 - v3:" << endl;
    vector<double> v_minus2 = v1 - v3;
    cout << "v1 - v2:" << endl;
    vector<double> v_minus3 = v1 - v2;
    cout << v_minus3;

    cout << endl;

    // *
    cout << "m_null * 2:" << endl;
    Matrix<double> m_mul1 = m_null * 2.0;
    cout << "m1 * 2:" << endl;
    Matrix<double> m_mul2 = m1 * 2.0;
    cout << m_mul2;
    cout << "2 * m1:" << endl;
    Matrix<double> m_mul3 = 2.0 * m1;
    cout << m_mul3;
    cout << endl;
    cout << "v_null * 2:" << endl;
    vector<double> v_mul1 = v_null * 2.0;
    cout << "v1 * 2:" << endl;
    vector<double> v_mul2 = v1 * 2.0;
    cout << v_mul2;
    cout << "2 * v1:" << endl;
    vector<double> v_mul3 = 2.0 * v1;
    cout << v_mul3;

    cout << endl;

    // /
    cout << "m_null / 2:" << endl;
    Matrix<double> m_div1 = m_null / 2.0;
    cout << "m1 / 2" << endl;
    Matrix<double> m_div2 = m1 / 2.0;
    cout << m_div2;
    cout << endl;
    cout << "v_null / 2:" << endl;
    vector<double> v_div1 = v_null / 2.0;
    cout << "v1 / 2" << endl;
    vector<double> v_div2 = v1 / 2.0;
    cout << v_div2;

    cout << endl;

    // transposition
    cout << "transposition of m_null:" << endl;
    Matrix<double> m_trans1 = m_null.transposition();
    cout << "transposition of m1:" << endl;
    Matrix<double> m_trans2 = m1.transposition();
    cout << m_trans2;
    cout << "transposition of m3:" << endl;
    Matrix<double> m_trans3 = m1.transposition(m3);
    cout << m_trans3;

    cout << endl;

    // conjugation
    cout << "conjugation of m_null:" << endl;
    Matrix<double> m_conj1 = m_null.conjugation();
    cout << "conjugation of m1:" << endl;
    Matrix<double> m_conj2 = m1.conjugation();
    cout << m_conj2;
    cout << "conjugation of m3:" << endl;
    Matrix<double> m_conj3 = m1.conjugation(m3);
    cout << m_conj3;

    cout << endl;

    // element_wise_mul
    cout << "element_wise_mul of m1 and m_mul:" << endl;
    Matrix<double> m_element_wise_mul1 = m1.element_wise_mul(m_null);
    cout << "element_wise_mul of m1 and m3:" << endl;
    Matrix<double> m_element_wise_mul2 = m1.element_wise_mul(m3);
    cout << "element_wise_mul of m1 and m2:" << endl;
    Matrix<double> m_element_wise_mul3 = m1.element_wise_mul(m1, m2);
    cout << m_element_wise_mul3;
    cout << endl;
    cout << "element_wise_mul of v1 and v_null:" << endl;
    vector<double> v_element_wise_mul1 = m1.element_wise_mul(v1, v_null);
    cout << "element_wise_mul of v1 and v3:" << endl;
    vector<double> v_element_wise_mul2 = m1.element_wise_mul(v1, v3);
    cout << "element_wise_mul of v1 and v2:" << endl;
    vector<double> v_element_wise_mul3 = m1.element_wise_mul(v1, v2);
    cout << v_element_wise_mul3;

    cout << endl;

    // *
    cout << "m1 * m_null:" << endl;
    Matrix<double> m_mulm1 = m1 * m_null;
    cout << "m1 * m3:" << endl;
    Matrix<double> m_mulm2 = m1 * m3;
    cout << "m1 * m2:" << endl;
    Matrix<double> m_mulm3 = m1 * m2;
    cout << m_mulm3;
    cout << endl;
    cout << "v1 * v_null:" << endl;
    vector<double> m_mulv1 = m1 * v_null;
    cout << "v1 * v3:" << endl;
    vector<double> m_mulv2 = m1 * v3;
    cout << "v1 * v2:" << endl;
    vector<double> m_mulv3 = m1 * v2;
    cout << m_mulv3;

    cout << endl;

    // dot_product
    cout << "m1 dot_product m_null:" << endl;
    double m_dot_product1 = m1.dot_product(m_null);
    cout << "m1 dot_product m3:" << endl;
    double m_dot_product2 = m1.dot_product(m3);
    cout << "m1 dot_product m2:" << endl;
    double m_dot_product3 = m1.dot_product(m1, m2);
    cout << m_dot_product3 << endl;
    cout << endl;
    cout << "v1 dot_product v_null:" << endl;
    double v_dot_product1 = m1.dot_product(v1, v_null);
    cout << "v1 dot_product v3:" << endl;
    double v_dot_product2 = m1.dot_product(v1, v3);
    cout << "v1 dot_product v2:" << endl;
    double v_dot_product3 = m1.dot_product(v1, v2);
    cout << v_dot_product3 << endl;


    cout << endl;

    // cross_product
    cout << "v1 cross_product v3:" << endl;
    vector<double> v_cross_product1 = m1.cross_product(v1, v3);
    cout << "v1 cross_product v2:" << endl;
    vector<double> v_cross_product2 = m1.cross_product(v1, v2);
    cout << v_cross_product2;

    cout << endl;

    // check_type_axis
    cout << "check_type_axis of m_null:" << endl;
    bool check_type_axis1 = m_null.check_type_axis(0, 0);
    cout << "check_type_axis of m1:" << endl;
    bool check_type_axis2 = m1.check_type_axis(0, n + 1);
    bool check_type_axis3 = m1.check_type_axis(0, n - 1);
    cout << "valid axis of m1:" << n - 1 << endl;

    cout << endl;

    // find_max
    cout << "find_max of m_null:" << endl;
    double m_max1 = m_null.find_max();
    cout << "find_max of m1:" << endl;
    double m_max2 = m1.find_max();
    cout << m_max2 << endl;
    cout << "find_max of m1 row1:" << endl;
    double m_max3 = m1.find_max(0, 0);
    cout << m_max3 << endl;
    cout << "find_max of m1 col1:" << endl;
    double m_max4 = m1.find_max(1, 0);
    cout << m_max4 << endl;

    cout << endl;

    // find_min
    cout << "find_min of m_null:" << endl;
    double m_min1 = m_null.find_min();
    cout << "find_min of m1:" << endl;
    double m_min2 = m1.find_min();
    cout << m_min2 << endl;
    cout << "find_min of m1 row2:" << endl;
    double m_min3 = m1.find_min(0, 1);
    cout << m_min3 << endl;
    cout << "find_min of m1 col2:" << endl;
    double m_min4 = m1.find_min(1, 1);
    cout << m_min4 << endl;

    cout << endl;

    // sum
    cout << "sum of m_null:" << endl;
    double m_sum1 = m_null.sum();
    cout << "sum of m1:" << endl;
    double m_sum2 = m1.sum();
    cout << m_sum2 << endl;
    cout << "sum of m1 row1:" << endl;
    double m_sum3 = m1.sum(0, 0);
    cout << m_sum3 << endl;
    cout << "sum of m1 col1:" << endl;
    double m_sum4 = m1.sum(1, 0);
    cout << m_sum4 << endl;

    cout << endl;

    // avg
    cout << "avg of m_null:" << endl;
    double m_avg1 = m_null.avg();
    cout << "avg of m1:" << endl;
    double m_avg2 = m1.avg();
    cout << m_avg2 << endl;
    cout << "avg of m1 row1:" << endl;
    double m_avg3 = m1.avg(0, 0);
    cout << m_avg3 << endl;
    cout << "avg of m1 col1:" << endl;
    double m_avg4 = m1.avg(1, 0);
    cout << m_avg4 << endl;

    cout << endl;

    // check_square
    cout << "check_square of m_null:" << endl;
    bool check_square1 = m_null.check_square();
    cout << "check_square of m4:" << endl;
    bool check_square2 = m4.check_square();
    cout << "check_square of m1:" << endl;
    bool check_square3 = m1.check_square();
    cout << check_square3 << endl;

    cout << endl;

    // trace
    cout << "trace of m_null:" << endl;
    double trace1 = m_null.trace();
    cout << "trace of m4:" << endl;
    double trace2 = m4.trace();
    cout << "trace of m1:" << endl;
    double trace3 = m1.trace();
    cout << trace3 << endl;

    cout << endl;

    //determinant
    cout << "determinant of m1:" << endl;
    cout << m1.determinant() << endl;

    cout << endl;

    // inverse
    cout << "inverse of m1:" << endl;
    Matrix<double> inverse1 = m1.inverse();

    // norm2
    cout << "norm2 of m_null:" << endl;
    double norm21 = m_null.norm2();
    cout << "norm2 of m1:" << endl;
    double norm22 = m1.norm2();
    cout << norm22 << endl;

    cout << endl;

    // eigenvalues
    cout << "eigenvalues of m3:" << endl;
    vector<double> eigenvalues1 = m3.eigenvalues();
    cout << eigenvalues1 << endl;

    cout << endl;

    // print_eigenvectors
    cout << "print_eigenvectors of m3:" << endl;
    m3.print_eigenvectors();

    cout << endl;

    // reshape
    cout << "reshape of m1:" << endl;
    Matrix<double> reshape1 = m1.reshape(1, m1.getRow() * m1.getCol());
    cout << reshape1;

    cout << endl;

    // slicing
    cout << "testcase of slicing:" << endl;
    vector<vector<double>> s(4, vector<double>(4));
    s = { {1,2,3,4},{5,6,7,8},{1,3,5,7},{2,4,6,8} };
    Matrix<double> S(s);
    cout << S;
    cout << Matrix<double>::slicing(S, 0, 3, 1, 2, 1, 0);
    cout << S.slicing(2, 3, 0, 3, 0, 0);

    cout << endl;

    // convolution
    cout << "testcase of convolution:" << endl;
    vector<vector<double>> a(5, vector<double>(5));
    a = { {17,24,1,8,15},{23,5,7,14,16},{4,6,13,20,22},{10,12,19,21,3},
{11,18,25,2,9} };
    Matrix<double> A(a);
    vector<vector<double>> b(3, vector<double>(3));
    b = { {1,3,1},{0,5,0},{2,1,2} };
    Matrix<double> B(b);
    cout << Matrix<double>::convolution(A, B);
    cout << endl;

    //complex number
    vector<vector<complex<double>>> p(2, vector<complex<double>>(2));
    complex<double> c(1, 1);
```

```cpp
    p = { {1,c},{c,2} };
    Matrix<complex<double>> P(p);

    //Matrix<complex<T>>
    complex<double> c1(1, 1);
    complex<double> c2(0, 1);
    vector<vector<complex<double>>> cm;
    vector<complex<double>> r1, r2;
    r1.emplace_back(c1);
    r1.emplace_back(c2);
    r2.emplace_back(c1);
    r2.emplace_back(c1);
    // <<
    cout << "The first complex c1:" << endl;
    cout << c1 << endl;
    cout << "The second complex c2:" << endl;
    cout << c2 << endl;
    cout << endl;
    cout << "The first complex vector r1:" << endl;
    cout << r1 << endl;
    cout << "The second complex vector r2:" << endl;
    cout << r2 << endl;
    cout << endl;

    //vector +
    cout << "r1 + r2:" << endl;
    cout << r1 + r2 << endl;
    //vector -
    cout << "r1 - r2:" << endl;
    cout << r1 - r2 << endl;
    //vector *
    cout << "r1 * r2:" << endl;
    cout << r1 * r2 << endl;
    //vector / val
    cout << "r1 / c2:" << endl;
    cout << r1 / c2 << endl;
    cout << endl;
    cm.emplace_back(r1);
    cm.emplace_back(r2);
    Matrix<complex<double>> C(cm);
    //print C
    cout << "The complex matrix C:" << endl;
    cout << C << endl;
    //matrix calculation
    cout << "The calculation C + 2 * c1 * C:" << endl;
    cout << C + (complex<double>)2 * c1 * C << endl;
    //conjugation
    cout << "The conjugation of complex matrix C:" << endl;
    cout << C.conjugation();
    cout << endl;
    //norm2
    cout << "The norm of complex matrix C:" << endl;
    cout << C.norm2() << endl;
    cout << "The norm of complex vector r1:" << endl;
    cout << Matrix<complex<double>>::norm2(r1) << endl;
    cout << endl;

    return 0;
```

```
}
```

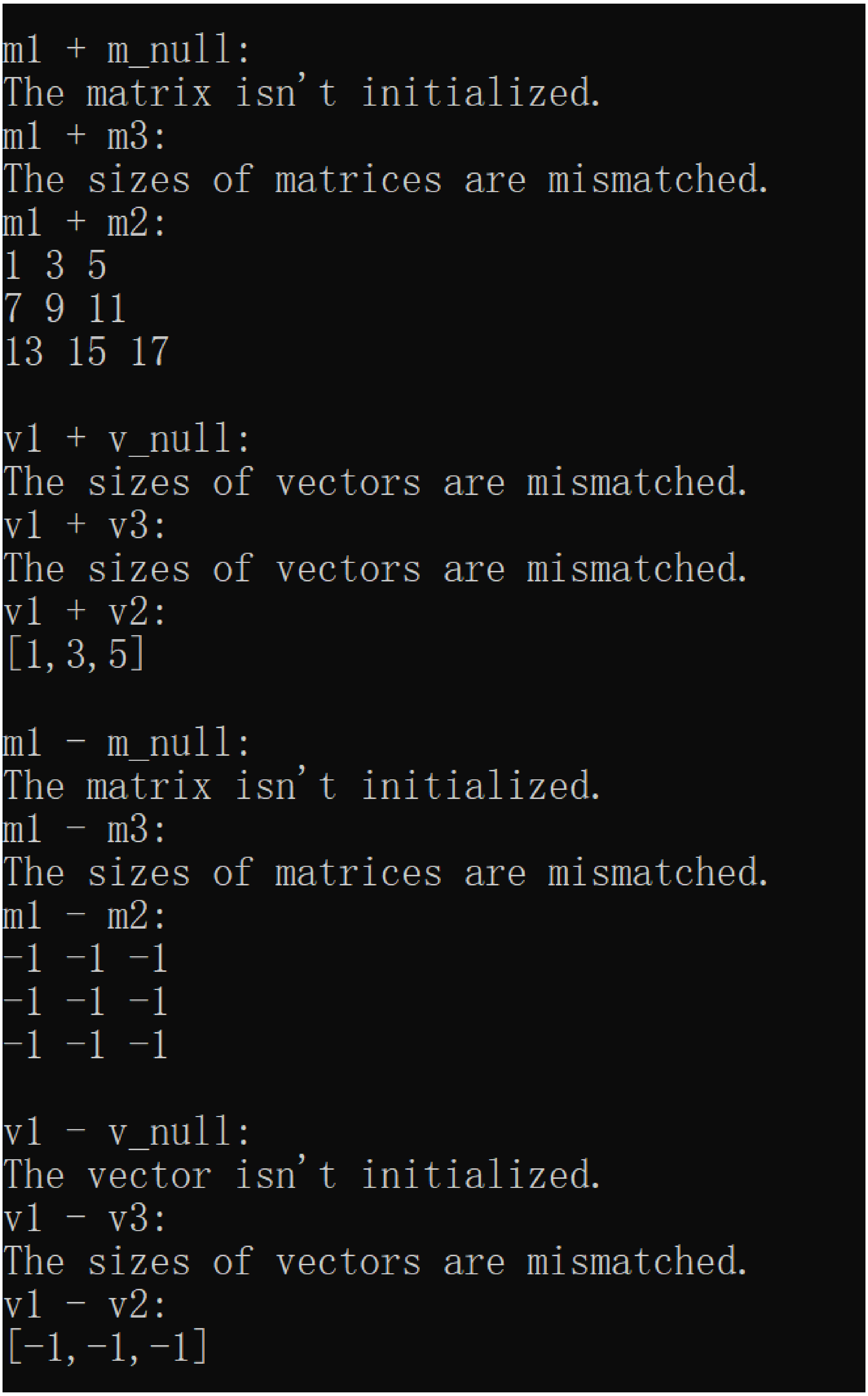
Part 3 - Result & Verification

Test operator<<



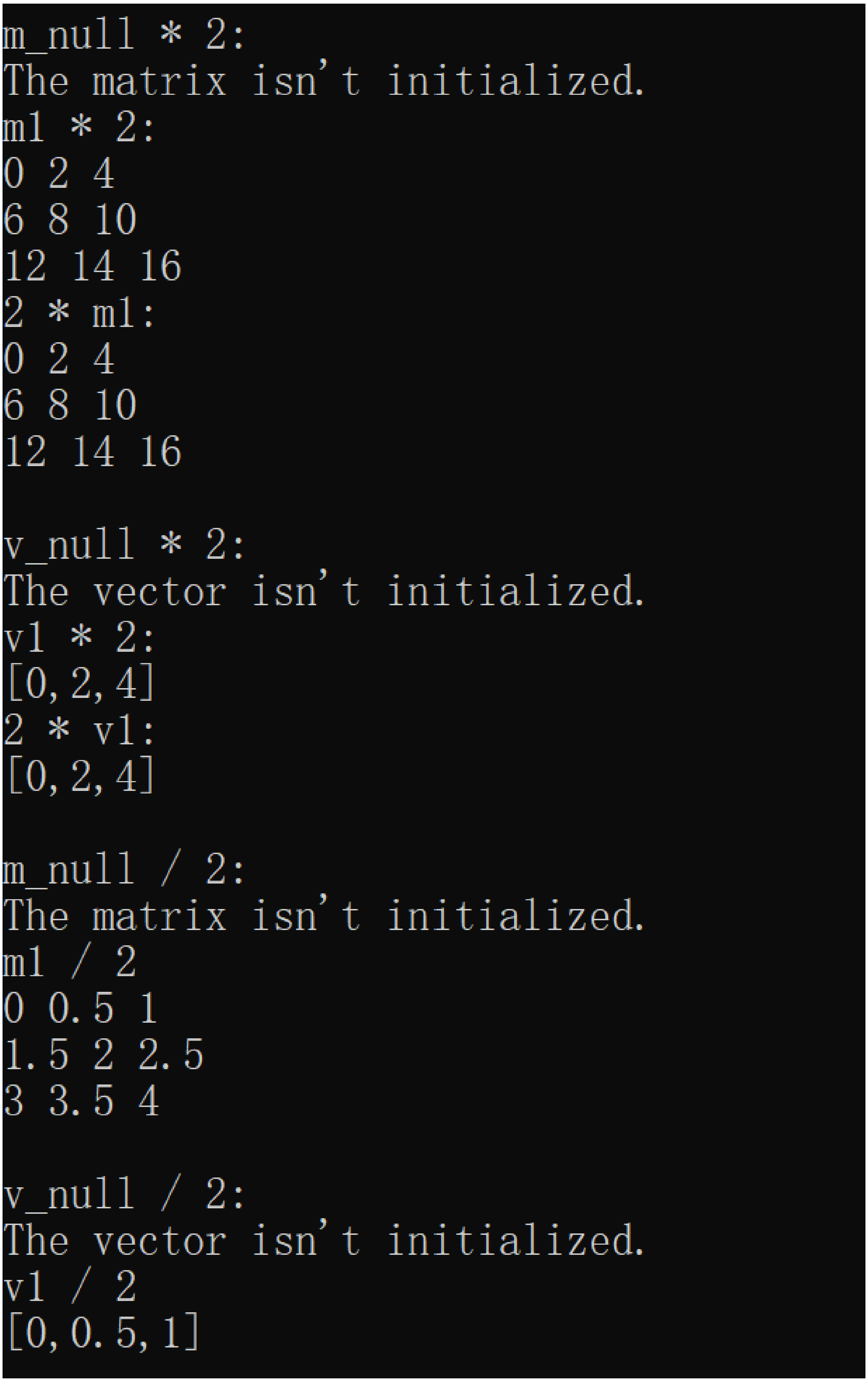
```
The empty matrix m_null:
The matrix isn't initialized.
The first matrix m1:
0 1 2
3 4 5
6 7 8
The second matrix m2:
1 2 3
4 5 6
7 8 9
The third matrix m3:
0 1
2 3
The third matrix m4:
0 1
3 4
6 7

The empty vector v_null:
The vector isn't initialized.
The first vector v1:
[0,1,2]
The second vector v2:
[1,2,3]
The third vector v3:
[1,2]
```

Test operator+, -



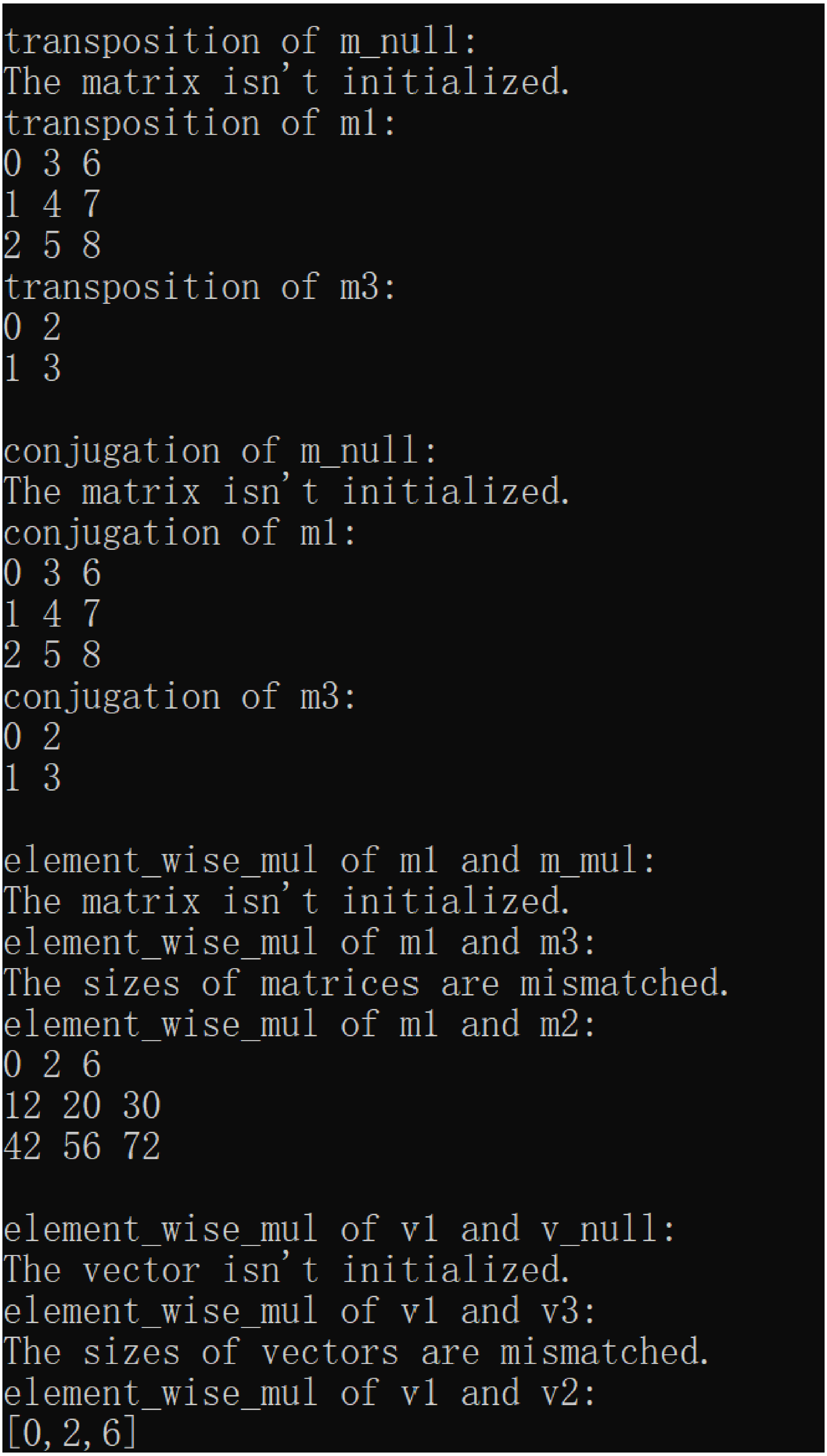
```
m1 + m_null:
The matrix isn't initialized.
m1 + m3:
The sizes of matrices are mismatched.
m1 + m2:
1 3 5
7 9 11
13 15 17

v1 + v_null:
The sizes of vectors are mismatched.
v1 + v3:
The sizes of vectors are mismatched.
v1 + v2:
[1,3,5]

m1 - m_null:
The matrix isn't initialized.
m1 - m3:
The sizes of matrices are mismatched.
m1 - m2:
-1 -1 -1
-1 -1 -1
-1 -1 -1

v1 - v_null:
The vector isn't initialized.
v1 - v3:
The sizes of vectors are mismatched.
v1 - v2:
[-1,-1,-1]
```

Test operator *, / with constant

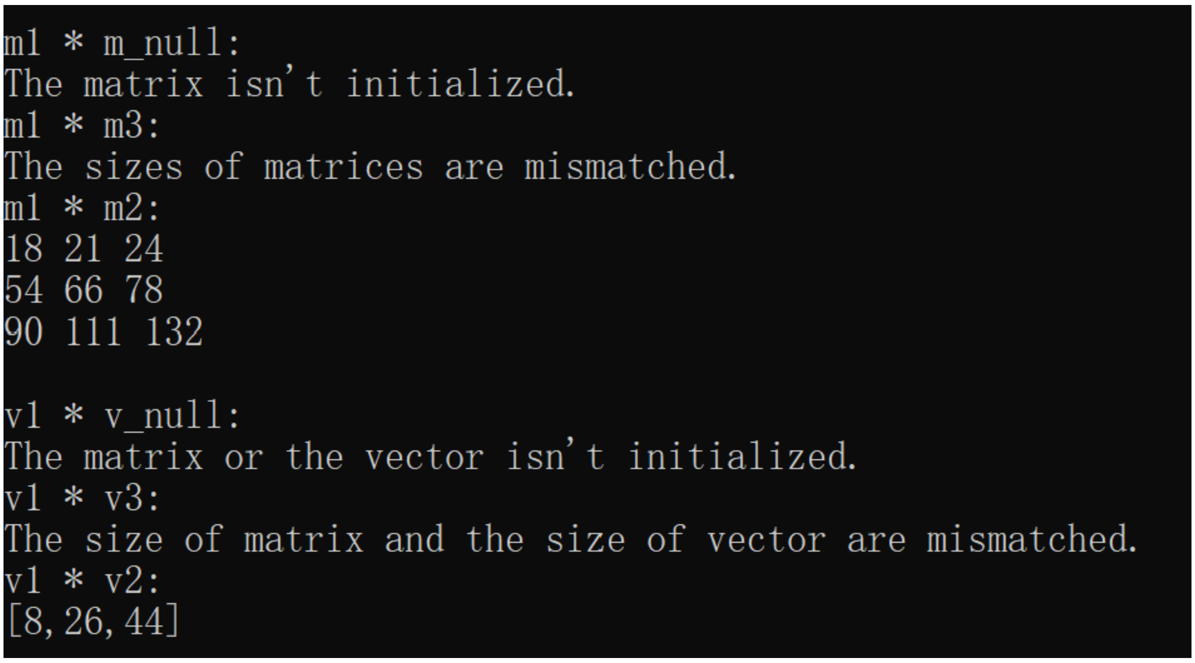


```
m_null * 2:
The matrix isn't initialized.
m1 * 2:
0 2 4
6 8 10
12 14 16
2 * m1:
0 2 4
6 8 10
12 14 16

v_null * 2:
The vector isn't initialized.
v1 * 2:
[0,2,4]
2 * v1:
[0,2,4]

m_null / 2:
The matrix isn't initialized.
m1 / 2
0 0.5 1
1.5 2 2.5
3 3.5 4

v_null / 2:
The vector isn't initialized.
v1 / 2
[0,0.5,1]
```

Test transposion, conjugation, element_wise_multiplication

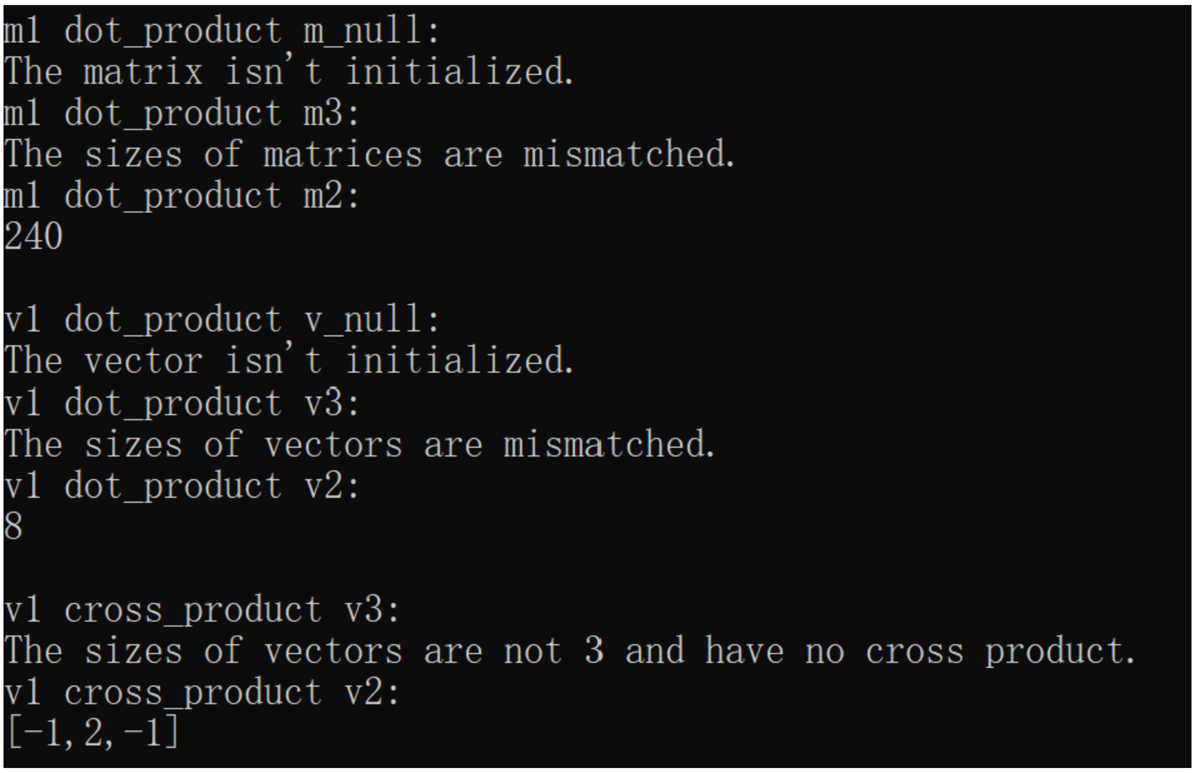

```
transposition of m_null:
The matrix isn't initialized.
transposition of m1:
0 3 6
1 4 7
2 5 8
transposition of m3:
0 2
1 3

conjugation of m_null:
The matrix isn't initialized.
conjugation of m1:
0 3 6
1 4 7
2 5 8
conjugation of m3:
0 2
1 3

element_wise_mul of m1 and m_mul:
The matrix isn't initialized.
element_wise_mul of m1 and m3:
The sizes of matrices are mismatched.
element_wise_mul of m1 and m2:
0 2 6
12 20 30
42 56 72

element_wise_mul of v1 and v_null:
The vector isn't initialized.
element_wise_mul of v1 and v3:
The sizes of vectors are mismatched.
element_wise_mul of v1 and v2:
[0,2,6]
```

Test operator* (matrix mul matrix, vector mul vector)



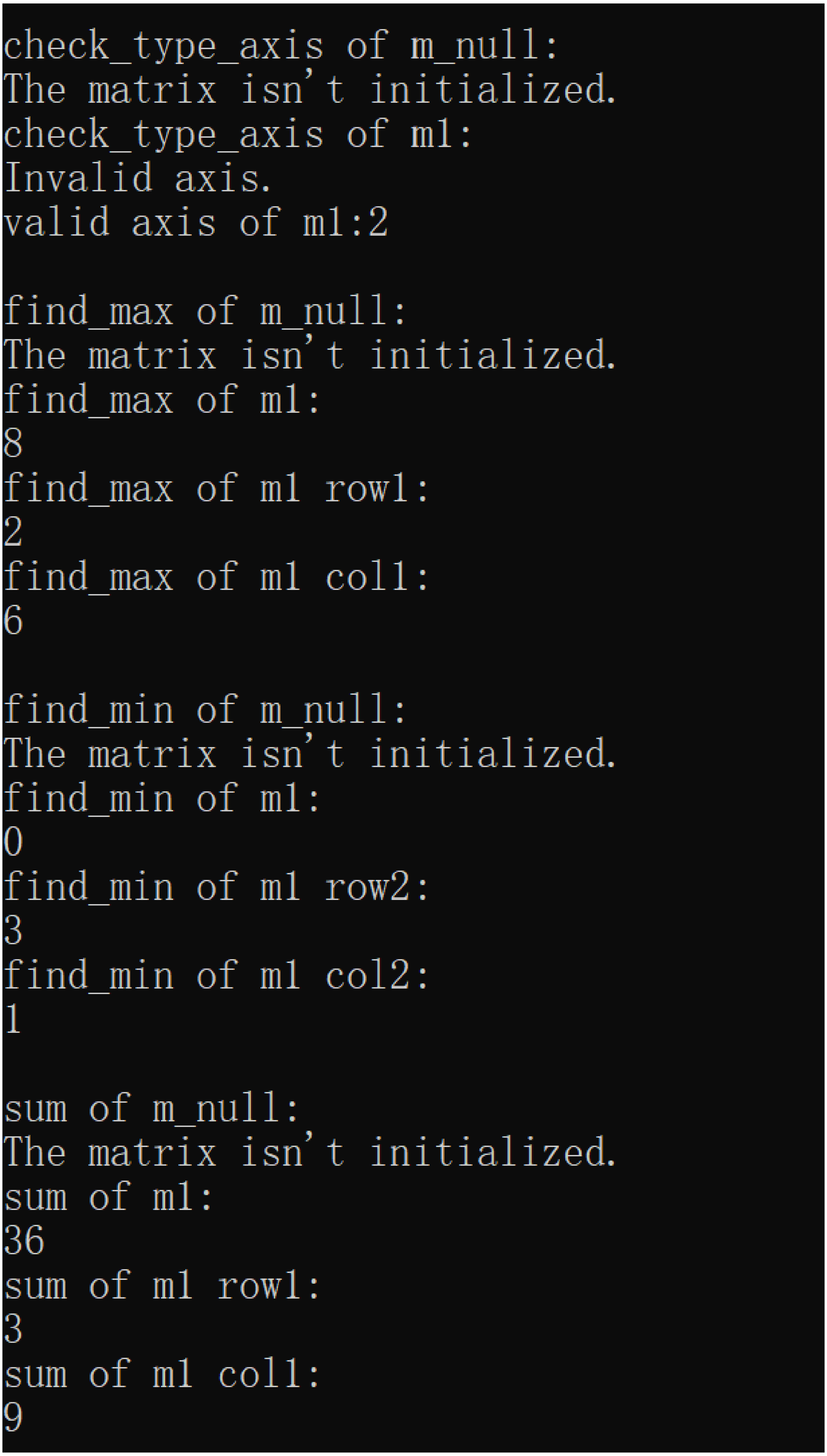
```
m1 * m_null:
The matrix isn’t initialized.
m1 * m3:
The sizes of matrices are mismatched.
m1 * m2:
18 21 24
54 66 78
90 111 132

v1 * v_null:
The matrix or the vector isn’t initialized.
v1 * v3:
The size of matrix and the size of vector are mismatched.
v1 * v2:
[8,26,44]
```

Test dot_product, cross_product

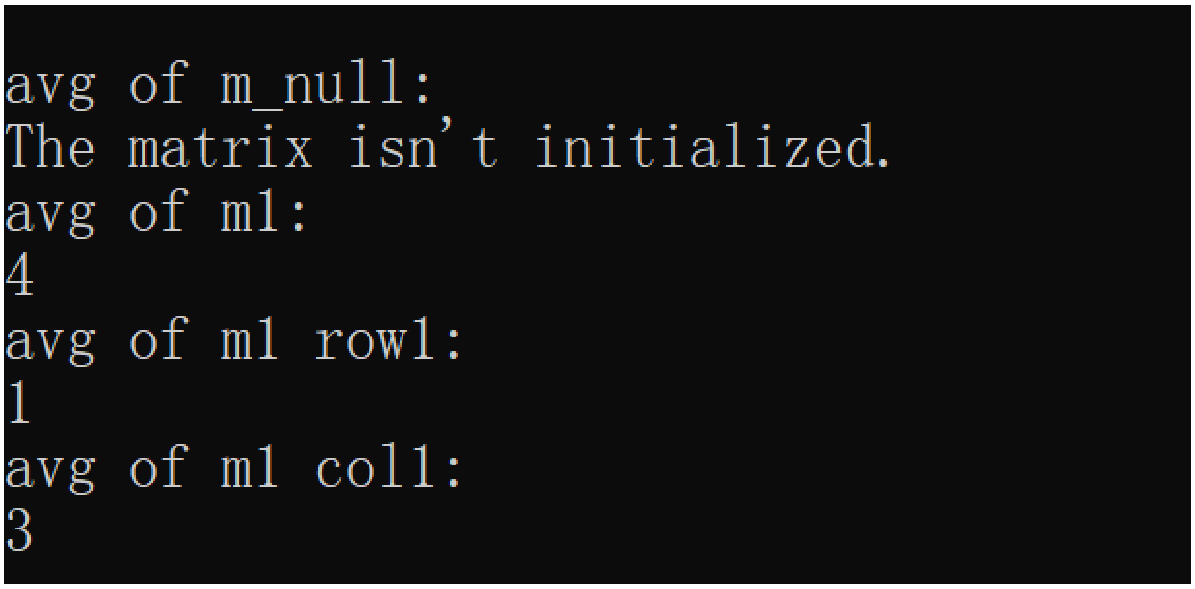


```
m1 dot_product m_null:
The matrix isn’t initialized.
m1 dot_product m3:
The sizes of matrices are mismatched.
m1 dot_product m2:
240

v1 dot_product v_null:
The vector isn’t initialized.
v1 dot_product v3:
The sizes of vectors are mismatched.
v1 dot_product v2:
8

v1 cross_product v3:
The sizes of vectors are not 3 and have no cross product.
v1 cross_product v2:
[-1,2,-1]
```

Test check_type_axis, find _max, find_min, sum, average

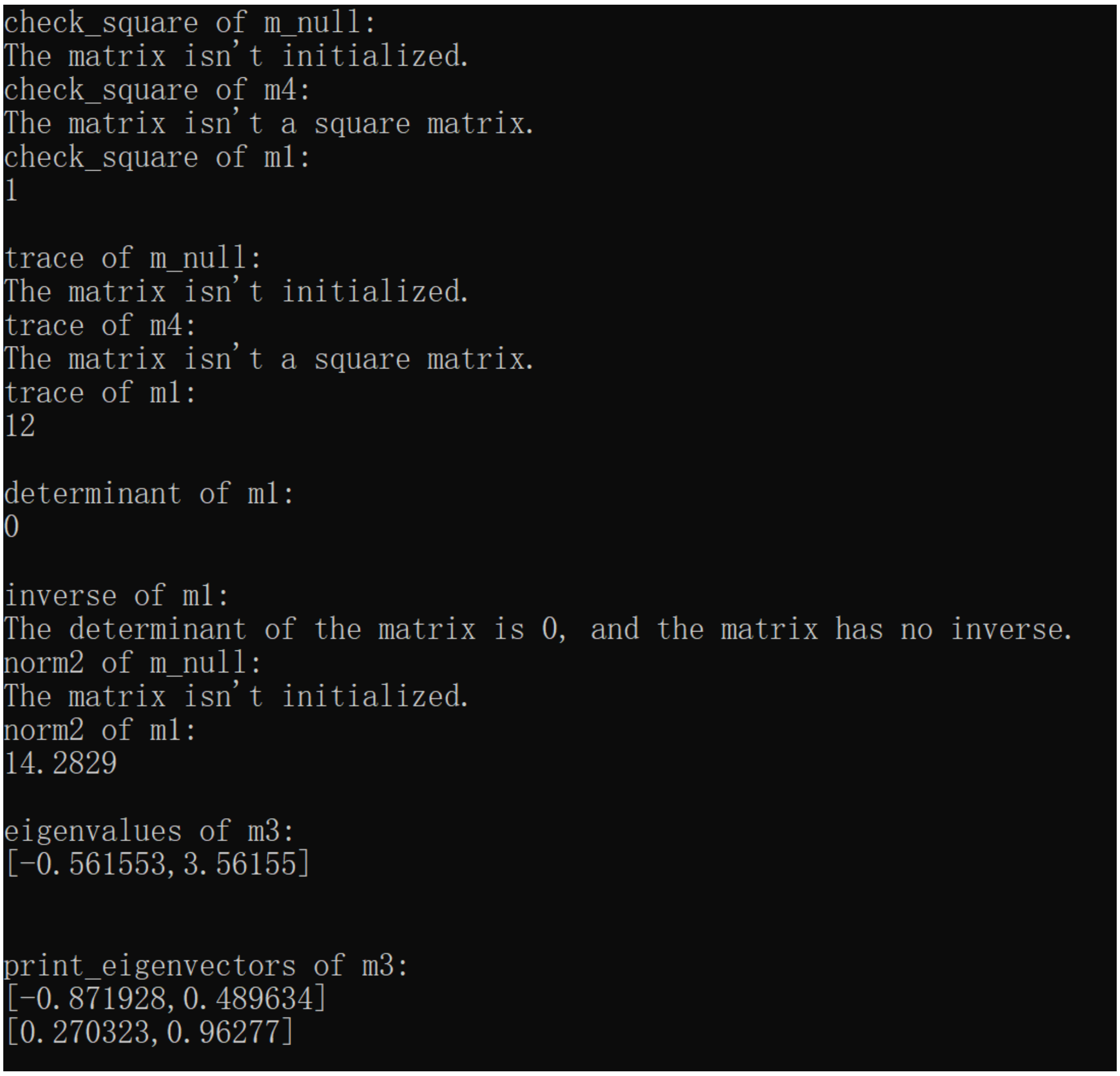


```
check_type_axis of m_null:
The matrix isn't initialized.
check_type_axis of m1:
Invalid axis.
valid axis of m1:2

find_max of m_null:
The matrix isn't initialized.
find_max of m1:
8
find_max of m1 row1:
2
find_max of m1 col1:
6

find_min of m_null:
The matrix isn't initialized.
find_min of m1:
0
find_min of m1 row2:
3
find_min of m1 col2:
1

sum of m_null:
The matrix isn't initialized.
sum of m1:
36
sum of m1 row1:
3
sum of m1 col1:
9
```



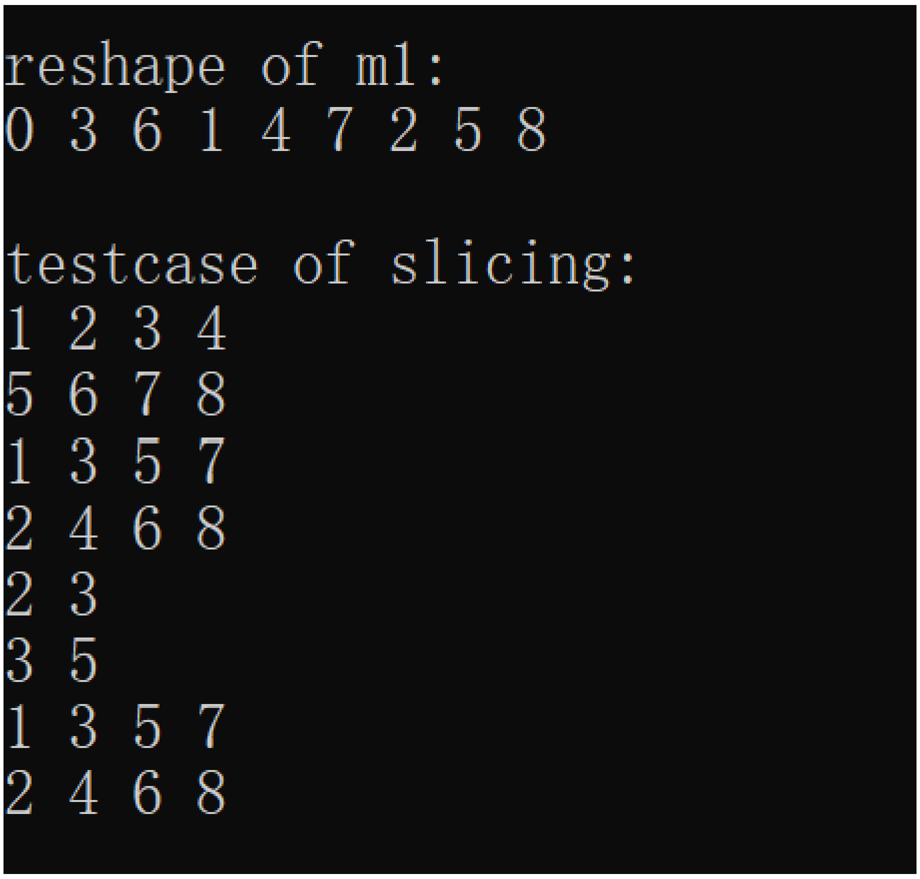
```
avg of m_null:
The matrix isn’t initialized.
avg of m1:
4
avg of m1 row1:
1
avg of m1 col1:
3
```

Test check_square, trace, determinant, inverse, eigenvalues, eigenvectors



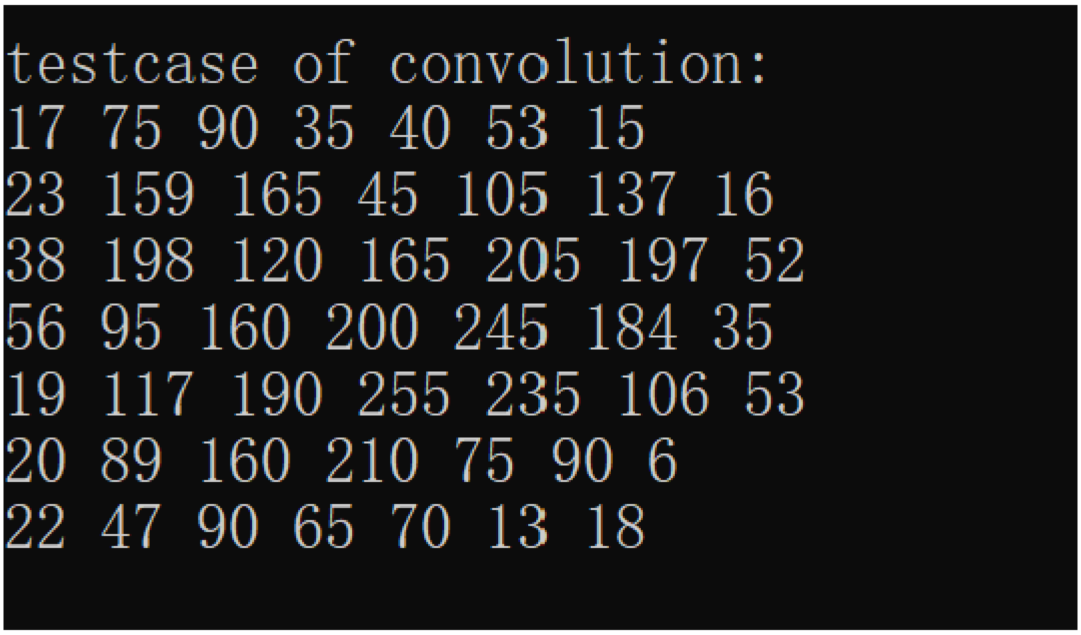
```
check_square of m_null:
The matrix isn’t initialized.
check_square of m4:
The matrix isn’t a square matrix.
check_square of m1:
1

trace of m_null:
The matrix isn’t initialized.
trace of m4:
The matrix isn’t a square matrix.
trace of m1:
12

determinant of m1:
0

inverse of m1:
The determinant of the matrix is 0, and the matrix has no inverse.
norm2 of m_null:
The matrix isn’t initialized.
norm2 of m1:
14.2829

eigenvalues of m3:
[-0.561553,3.56155]


print_eigenvectors of m3:
[-0.871928,0.489634]
[0.270323,0.96277]
```

Test reshape, slicing



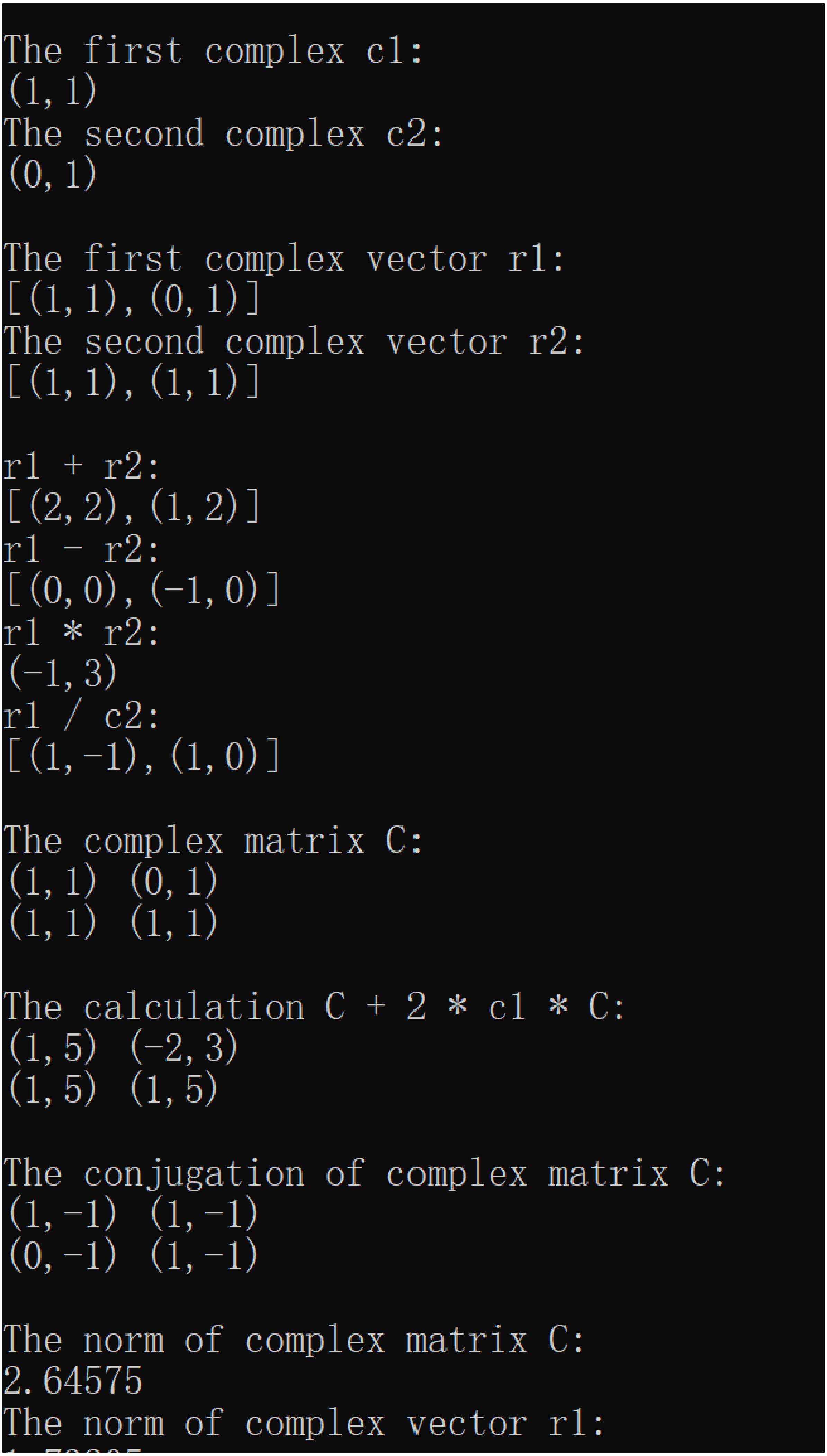
```
reshape of m1:
0 3 6 1 4 7 2 5 8

testcase of slicing:
1 2 3 4
5 6 7 8
1 3 5 7
2 4 6 8
2 3
3 5
1 3 5 7
2 4 6 8
```

Test convolutional



```
testcase of convolution:
17 75 90 35 40 53 15
23 159 165 45 105 137 16
38 198 120 165 205 197 52
56 95 160 200 245 184 35
19 117 190 255 235 106 53
20 89 160 210 75 90 6
22 47 90 65 70 13 18
```

Test complex



```
The first complex c1:
(1,1)
The second complex c2:
(0,1)

The first complex vector r1:
[(1,1),(0,1)]
The second complex vector r2:
[(1,1),(1,1)]

r1 + r2:
[(2,2),(1,2)]
r1 - r2:
[(0,0),(-1,0)]
r1 * r2:
(-1,3)
r1 / c2:
[(1,-1),(1,0)]

The complex matrix C:
(1,1) (0,1)
(1,1) (1,1)

The calculation C + 2 * c1 * C:
(1,5) (-2,3)
(1,5) (1,5)

The conjugation of complex matrix C:
(1,-1) (1,-1)
(0,-1) (1,-1)

The norm of complex matrix C:
2.64575
The norm of complex vector r1:
```

```
1. 73205
```

Part 4 - Difficulties & Solutions

Difficulties: Because the complex numbers can not be compared, some function will cause compile error when doing complex matrix operation.

Solutions: Using template specialization, call the template more explicitly so that when the compiler compiles, it selects the specialized template first.

Template specialization:

```
template<typename D>
class Matrix<complex<D>> {};
```

We don't define functions that are illegal for complex numbers in this specialized template class, and it will not lead to compile error.