CS205 C/ C++ Programming - Project - matrix

环境：Ubuntu 服务器

编译参数：

```
`pkg-config opencv --cflags`
`pkg-config opencv --libs`
```

Name: 陈松斌(Chen Songbin)，金冬阳(Jin Dongyang)，张颖栋(Zhang Yingdong)

SID: 11812005，11911221，11910709

CPU：Intel Core i5-9300H

内存：8G

测试平台：Ubuntu 7.5.0-3ubuntu1~18.04

编码：UTF-8

C++标准：C++11

Part 1 - Analysis

1) It should support all matrix sizes, for arbitrarily large dense matrices, and even sparse matrices.

We choose to store the matrix in a one-dimensional array and use linked lists to store sparse matrices. Then we realize the mutual transformation of dense matrix and sparse matrix.

2) It can support all standard numeric types, including std::complex, integers, even custom numeric types.

We use generic classes to solve this problem.

3) It supports matrix and vector arithmetic, including addition, subtraction, scalar multiplication, scalar division, transposition, conjugation, element-wise multiplication, matrix-matrix multiplication, matrix-vector multiplication, dot product and cross product.

addition

```
    matrix<T> operator+(const matrix<T>& other) const;
```

We first check whether the two matrices have the same number of rows and columns. If they are equal, we add up the corresponding elements to get the answer.

subtraction

```
    matrix<T> operator-(const matrix<T>& other) const;
```

We first check whether the two matrices have the same number of rows and columns. If they're equal, we get the answer by subtracting the corresponding elements.

scalar multiplication

```
    matrix<T> operator*(T tnum) const;
    template <typename T1>
    friend matrix<T1> operator*(T1 tnum, const matrix<T1>& mat);
```

We get the result by multiplying each element of the matrix by a number. And we use friend classes to allow the scalar multiplication of matrices to be unconstrained by the order of operations.

scalar division

```
    matrix<T> operator/(T tnum) const;
```

We get the result by dividing each element of the matrix by a number.

transposition

```
    matrix<T> transposition() const;
```

We start by creating a matrix. We use the rows of this matrix to store the elements of the original matrix's columns and we use the columns of this matrix to store the elements of the original matrix's rows. Then we return this matrix to get the result.

conjugation

```
    matrix<T> conjugation() const;
```

We get the result by taking the conjugate of each of the elements in the matrix.

element-wise multiplication/dot product

```
    matrix<T> multiply_element_wise(const matrix<T>& other) const;
    matrix<T> dot(const matrix<T>& other) const;
```

We first check whether the two matrices have the same number of rows and columns. If they're equal, we get the answer by multiplying the corresponding elements.

matrix-matrix multiplication/cross multiplication/matrix-vector multiplication

```
    matrix<T> operator*(const matrix<T>& other);
    matrix<T> cross(const matrix<T>& other) const;
    matrix<T> operator*(const matrix<T>& other);
    template <typename T1>
    friend vector<T1> operator*(const vector<T1>& vec, const matrix<T1>& mat);
```

We should first check whether the number of columns in the first matrix is equal to the number of rows in the second matrix. we use the rules of matrix multiplication to calculate the results.

4) It supports basic arithmetic reduction operations, including finding the maximum value, finding the minimum value, summing all items, calculating the average value (all supporting axis-specific and all items).

maximum value

```
    T max(size_type rowi = -1, size_type coli = -1) const;
```

Traverse the matrix to find the maximum value by recording it. (Parameter is the index to limit the traversal range of matrix)

Both coli and rowi are initialized to - 1. If the parameter is passed in, the row / col corresponding to this value will be calculated. If the parameter is not passed in, the whole matrix will be traversed.

minimum value

```
    T min(size_type rowi = -1, size_type coli = -1) const;
```

Traverse the matrix to find the minimum value by recording it. (Parameter is the index to limit the traversal range of matrix)

Both coli and rowi are initialized to - 1. If the parameter is passed in, the row / col corresponding to this value will be calculated. If the parameter is not passed in, the whole matrix will be traversed.

summing all items

```
    T sum(size_type rowi = -1, size_type coli = -1) const;
```

Traverse the matrix to find the sum of matrix elements. (Parameter is the index to limit the traversal range of matrix)

Both coli and rowi are initialized to - 1. If the parameter is passed in, the row / col corresponding to this value will be calculated. If the parameter is not passed in, the whole matrix will be traversed.

the average value

```
    T avg(size_type rowi = -1, size_type coli = -1) const;
```

First call sum () method to calculate sum, and then judge the range of the matrix according to the parameters, so as to calculate the average.

Both coli and rowi are initialized to - 1. If the parameter is passed in, the row / col corresponding to this value will be calculated. If the parameter is not passed in, the whole matrix will be traversed.

5) It supports computing eigenvalues and eigenvectors, calculating traces, computing inverse and computing determinant.

eigenvalues/eigenvectors

```
    template <typename T1>
    friend void QR(const matrix<T1>& A, matrix<T1>& Q, matrix<T1>& R);
    template <typename T1>
    friend void Eig(const matrix<T1>& A,
                    matrix<T1>& eigenVector,
                    matrix<T1>& eigenValue);
    matrix<T> eigenvalues() const;
    matrix<T> eigenvectors() const;
```

QR decomposition to decompose the matrix A. The decomposed orthogonal matrix Q and upper triangular matrix R are obtained.

The eigenvalue is obtained by QR decomposition, and then the eigenvector is calculated by the eigenvalue.

trace

```
    T trace() const;
```

Trace is the sum of diagonal elements, so we need to check whether the matrix is a square matrix first, and then take the diagonal elements to add.

algebraic_cofactor

```
    T algebraic_cofactor(size_type i, size_type j) const;
```

one or more of its rows and columns. Minors obtained by removing just one row and one column from square matrices (first minors) are required for calculating matrix cofactors, which in turn are useful for computing both the determinant and inverse of square matrices. For the convenience of computing determinant and inverse, we write a method of computing algebraic covalent.

inverse

```
    matrix<T> inverse() const;
```

First of all, we need to check whether this matrix is a square matrix. We can use the algebraic cofactor to find the matrix of the algebraic cofactor, and then divide it by the determinant to unite the elements, and then use the transpose method to find the inverse.

determinant

```
    T determinant() const;
```

First of all, we need to check whether this matrix is a square matrix. Because the determinant is the result of the algebraic cofactor of a row of elements and other elements, it is easy to obtain the determinant through the method of algebraic cofactor.

6) It supports the operations of reshape and slicing.

reshape

```
    matrix<T> reshape(size_type row_size, size_type col_size) const;
```

Reshape is a way to change the shape of an array. The array in the matrix is stored as a one bit array, so we only change the values of our row and col.

slice

```
    matrix<T> slice_row(size_type, size_type) const;
    matrix<T> slice_col(size_type, size_type) const;
    matrix<T> slice(size_type, size_type, size_type, size_type) const;
```

Slice is to cut the elements of the matrix. So as to obtain a new matrix method. We divide slice into three methods, the first is row cutting, the second is column cutting, and the third is cutting row and column together. These three methods all need to check whether the parameters in them are out of bounds. The first two slice methods can be called directly for row and column cutting methods.

7) It supports convolutional operations of two matrices.

convolution

```
    enum { FULL, SAME, VALID };
    matrix<T> convolution(const matrix<T>& other, int mode = FULL) const;
```

There are three types of convolution return: full, same and valid. So let's write an enumeration type first. If there is no relevant parameter, the default is convolution of full type. Then the core is rotated 180 degrees, and the convolution is calculated step by step according to the algorithm.

8) It supports to transfer the matrix from OpenCV to the matrix of this library and vice versa.

```
    matrix<T>(const cv::Mat& src);
    cv::Mat toCVMat() const;
```

We can construct the matrix of our library using data from the matrix from OpenCV. And we can transform the matrix of the library to the matrix from OpenCV by using the cv::Mat() constructor.

9) It can supports handling of four different exceptions.

Create four exception classes: ZeroDet, DiffSize, NotSquare, OutOfRange. They can be used to determine whether the determinant of a matrix is zero, whether the number of rows and columns of a matrix meets the requirements, whether the matrix is square, and whether the data is out of range.

10) Build and publish static libraries

Create the static library according to the requirements of the slide.

Part 2 - Code

matrix.cpp

```
#ifndef _CXXPROJECT_MATRIX
#define _CXXPROJECT_MATRIX 1

#include <math.h>
#include <cstring>
```

```
#include <iostream>
#include <opencv2/opencv.hpp>
#include <vector>

typedef int size_type;
#define vector std::vector

class ZeroDet : public std::exception {
    const char* what() const throw() { return "The determinant is zero\n"; }
};
class DiffSize : public std::exception {
    const char* what() const throw() { return "Matrix row or col do not
match\n"; }
};
class NotSquare : public std::exception {
    const char* what() const throw() { return "Matrix is not a square\n"; }
};
class OutOfRange : public std::exception {
    const char* what() const throw() { return "Index out of range\n"; }
};

#define DET_CHECK(cond) \
    if (!(cond))        \
    throw ZeroDet()
#define SIZE_CHECK(cond) \
    if (!(cond))         \
    throw DiffSize()
#define SQUARE_CHECK(cond) \
    if (!(cond))           \
    throw NotSquare()
#define RANGE_CHECK(cond) \
    if (!(cond))          \
    throw OutOfRange()

//*Declare*

template <typename T>
class matrix {
   private:
    T* data;
    size_type row;
    size_type col;

    class data_list {
       public:
        T default_value = (T)0;
        struct node {
            T val;
            size_type i;
            size_type j;
            node* next;
        } *head, *tail;
        data_list() { head = tail = new node{}; }
        ~data_list() {
            node *last, *tmp = head;
            while (tmp != NULL) {
                last = tmp;
                tmp = tmp->next;
                delete last;
            }
        }
        T& get(size_type i, size_type j) {
            node* tmp = head;
            while (tmp != NULL) {
                if (tmp->i == i && tmp->j == j)
                    return tmp->val;
                tmp = tmp->next;
            }
            return default_value;
        }
        void set(T v, size_type i, size_type j) {
            tail = tail->next = new node{v, i, j, NULL};
        }
    } * sparseData;
    bool isSparse = false;

public:
    matrix<T>(size_type row_size = 1, size_type col_size = 1);
    matrix<T>(const T* arrays, size_type row_size, size_type col_size);
    matrix<T>(const matrix<T>& src);
    ~matrix<T>();

    matrix<T>& operator=(const matrix<T>& that);
    inline T& operator()(size_type i, size_type j) { return data[i * col + j]; }
    inline T operator()(size_type i, size_type j) const {
        return (isSparse) ? sparseData->get(i, j) : data[i * col + j];
    }


    // 1)-----------------
    // 稀疏矩阵转换
    bool toSparseMatrix(T default_value = (T)0);
    bool toNormalMatrix();

    // 3)-----------------
    // 矩阵加法
    matrix<T> operator+(const matrix<T>& other) const;
    // 矩阵减法
    matrix<T> operator-(const matrix<T>& other) const;
    // 标量乘法
    matrix<T> operator*(T tnum) const;
    // 标量除法
    matrix<T> operator/(T tnum) const;
    // 友元 标量乘法
    template <typename T1>
    friend matrix<T1> operator*(T1 tnum, const matrix<T1>& mat);

    // 转置
    matrix<T> transposition() const;
    // 共轭
    matrix<T> conjugation() const;
    // 2范数
    T norm2() const;

    // 元素级乘法
    matrix<T> multiply_element_wise(const matrix<T>& other) const;
```

```cpp
    // 矩阵乘法（向量级乘法）
    matrix<T> operator*(const matrix<T>& other);
    // 矩阵-向量乘法
    vector<T> operator*(const vector<T>& other) const;
    // 友元 向量-矩阵乘法
    template <typename T1>
    friend vector<T1> operator*(const vector<T1>& vec, const matrix<T1>& mat);
    // 点积
    matrix<T> dot(const matrix<T>& other) const;
    // 叉积
    matrix<T> cross(const matrix<T>& other) const;

    // 4)-----------------
    // 最大值
    T max(size_type rowi = -1, size_type coli = -1) const;
    // 最小值
    T min(size_type rowi = -1, size_type coli = -1) const;
    // 求和
    T sum(size_type rowi = -1, size_type coli = -1) const;
    // 平均值
    T avg(size_type rowi = -1, size_type coli = -1) const;

    // 5)-----------------
    template <typename T1>
    friend void QR(const matrix<T1>& A, matrix<T1>& Q, matrix<T1>& R);
    template <typename T1>
    friend void Eig(const matrix<T1>& A,
                    matrix<T1>& eigenVector,
                    matrix<T1>& eigenValue);
    // 特征值
    matrix<T> eigenvalues() const;
    // 特征向量
    matrix<T> eigenvectors() const;
    // 迹
    T trace() const;
    // 代数余子式
    T algebraic_cofactor(size_type i, size_type j) const;
    // 逆
    matrix<T> inverse() const;
    // 行列式
    T determinant() const;

    // 6)-----------------
    // 索引
    vector<T> operator[](size_type n) const;
    // 整形
    matrix<T> reshape(size_type row_size, size_type col_size) const;
    // 切片
    matrix<T> slice_row(size_type, size_type) const;
    matrix<T> slice_col(size_type, size_type) const;
    matrix<T> slice(size_type, size_type, size_type, size_type) const;

    // 7)-----------------
    // 卷积
    typedef unsigned int Mode;
    enum { FULL, SAME, VALID };
    matrix<T> convolution(const matrix<T>& other, Mode mode = FULL) const;
    // 8)-----------------
    // 构造器：cv::Mat
    matrix<T>(const cv::Mat& src);
    // 转cv::mat
    cv::Mat toCVMat() const;

    // )-----------------
    // 输出
    template <typename T1>
    friend std::ostream& operator<<(std::ostream& cout, const matrix<T1>& mat);
};

//*Implementation*

// 构造器
template <typename T>
matrix<T>::matrix(size_type row_size, size_type col_size)
    : row(row_size), col(col_size), data(new T[row_size * col_size]()) {}
template <typename T>
matrix<T>::matrix(const T* arrays, size_type row_size, size_type col_size)
    : row(row_size), col(col_size), data(new T[row_size * col_size]()) {
    memcpy(data, arrays, row * col * sizeof(T));
}
template <typename T>
matrix<T>::matrix(const matrix<T>& src)
    : row(src.row), col(src.col), data(new T[src.row * src.col]()) {
    for (size_type i = 0; i < row; i++) {
        for (size_type j = 0; j < col; j++) {
            (*this)(i, j) = src(i, j);
        }
    }
}
template <typename T>
matrix<T>::~matrix() {
    if (isSparse) {
        delete sparseData;
    } else {
        delete[] data;
    }
}

template <typename T>
matrix<T>& matrix<T>::operator=(const matrix<T>& that) {
    SIZE_CHECK(this->row == that.row && this->col == that.col);
    toNormalMatrix();
    for (size_type i = 0; i < row; i++) {
        for (size_type j = 0; j < col; j++) {
            (*this)(i, j) = that(i, j);
        }
    }
    return (*this);
}

// 1)-----------------
// 稀疏矩阵转换
template <typename T>
bool matrix<T>::toSparseMatrix(T default_value) {
    if (!isSparse) {
        sparseData = new data_list();
        sparseData->default_value = default_value;
        for (size_type i = 0; i < row; i++) {
            for (size_type j = 0; j < col; j++) {
                T val = (*this)(i, j);
                if (val != default_value) {
                    sparseData->set(val, i, j);
                }
            }
        }
        delete[] data;
        isSparse = true;
        return true;
    } else {
        return false;
    }
}
template <typename T>
bool matrix<T>::toNormalMatrix() {
    if (isSparse) {
        isSparse = false;
        data = new T[row * col]();
        for (size_type k = 0; k < row * col; k++) {
            data[k] = sparseData->default_value;
        }
        auto* tmp = sparseData->head->next;
        while (tmp != NULL) {
            (*this)(tmp->i, tmp->j) = tmp->val;
            tmp = tmp->next;
        }
        delete sparseData;
        return true;
    } else {
        return false;
    }
}

// 3)-----------------
// 矩阵加法
template <typename T>
matrix<T> matrix<T>::operator+(const matrix<T>& other) const {
    SIZE_CHECK(this->row == other.row && this->col == other.col);
    matrix R(row, col);
    for (size_type i = 0; i < row; i++) {
        for (size_type j = 0; j < col; j++) {
            R(i, j) = (*this)(i, j) + other(i, j);
        }
    }
    return R;
}
// 矩阵减法
template <typename T>
matrix<T> matrix<T>::operator-(const matrix<T>& other) const {
    SIZE_CHECK(this->row == other.row && this->col == other.col);
    matrix R(row, col);
    for (size_type i = 0; i < row; i++) {
        for (size_type j = 0; j < col; j++) {
            R(i, j) = (*this)(i, j) - other(i, j);
        }
    }
    return R;
}
// 标量乘法
template <typename T>
matrix<T> matrix<T>::operator*(T tnum) const {
    matrix R(row, col);
    for (size_type i = 0; i < row; i++) {
        for (size_type j = 0; j < col; j++) {
            R(i, j) = (*this)(i, j) * tnum;
        }
    }
    return R;
}
// 标量除法
template <typename T>
matrix<T> matrix<T>::operator/(T tnum) const {
    matrix R(row, col);
    for (size_type i = 0; i < row; i++) {
        for (size_type j = 0; j < col; j++) {
            R(i, j) = (*this)(i, j) / tnum;
        }
    }
    return R;
}
// 友元 标量乘法
template <typename T>
matrix<T> operator*(T tnum, const matrix<T>& mat) {
    return mat * tnum;
}

// 转置
template <typename T>
matrix<T> matrix<T>::transposition() const {
    matrix R(col, row);
    for (size_type i = 0; i < row; i++) {
        for (size_type j = 0; j < col; j++) {
            R(j, i) = (*this)(i, j);
        }
    }
    return R;
}
// 共轭
template <typename T>
inline T conj(T n) {
    return n;
}
template <typename T>
matrix<T> matrix<T>::conjugation() const {
    matrix R(col, row);
    for (size_type i = 0; i < row; i++) {
        for (size_type j = 0; j < col; j++) {
            R(j, i) = conj((*this)(i, j));
        }
    }
    return R;
}
```

```
template <typename T>
T sqrt(T n) {
    if (n * n == n)
        return n;
    T temp = n / 2;
    while (true) {
        T a = temp;
        temp = (temp + n / 2) / 2;
        if (a - temp < (T)0.00001 && temp < a)
            return temp;
    }
}
// 2范数
template <typename T>
T matrix<T>::norm2() const {
    T t = (T)0;
    for (size_type i = 0; i < row; i++) {
        for (size_type j = 0; j < col; j++) {
            t += (*this)(i, j) * (*this)(i, j);
        }
    }
    T tmp = sqrt(t);
    return tmp;
}

// 元素级乘法
template <typename T>
matrix<T> matrix<T>::multiply_element_wise(const matrix<T>& other) const {
    SIZE_CHECK(this->row == other.row && this->col == other.col);
    matrix R(row, col);
    for (size_type i = 0; i < row; i++) {
        for (size_type j = 0; j < col; j++) {
            R(i, j) = (*this)(i, j) * other(i, j);
        }
    }
    return R;
}
// 矩阵乘法（向量级乘法）
template <typename T>
matrix<T> matrix<T>::operator*(const matrix<T>& other) {
    SIZE_CHECK(this->col == other.row);
    matrix<T> R(this->row, other.col);
    for (size_type i = 0; i < this->row; i++) {
        for (size_type j = 0; j < other.col; j++) {
            for (size_type k = 0; k < this->col; k++) {
                R(i, j) += (*this)(i, k) * other(k, j);
            }
        }
    }
    return R;
}
// 矩阵-向量乘法
template <typename T>
vector<T> matrix<T>::operator*(const vector<T>& other) const {
    SIZE_CHECK(this->col == other.size);
    vector<T> R(this->row);
    for (size_type i = 0; i < this->row; i++) {
        for (size_type k = 0; k < this->col; k++) {
            R[i] += (*this)(i, k) * other[k];
        }
    }
    return R;
}
// 友元 向量-矩阵乘法
template <typename T>
vector<T> operator*(const vector<T>& vec, const matrix<T>& mat) {
    SIZE_CHECK(vec.size == mat.row);
    vector<T> R(mat.col);
    for (size_type j = 0; j < mat.col; j++) {
        for (size_type k = 0; k < mat.row; k++) {
            R[j] += vec[k] * mat(k, j);
        }
    }
    return R;
}
// 点积
template <typename T>
matrix<T> matrix<T>::dot(const matrix<T>& other) const {
    return this->multiply_element_wise(other);
}
// 叉积
template <typename T>
matrix<T> matrix<T>::cross(const matrix<T>& other) const {
    return (*this) * other;
}

// 4)-----------------
// 最大值
template <typename T>
T matrix<T>::max(size_type rowi, size_type coli) const {
    size_type r1 = 0, r2 = row, c1 = 0, c2 = col;
    if (rowi >= 0) {
        r1 = rowi;
        r2 = rowi + 1;
    }
    if (coli >= 0) {
        c1 = coli;
        c2 = coli + 1;
    }
    T t = (*this)(r1, c1);
    for (size_type i = r1; i < r2; i++) {
        for (size_type j = c1; j < c2; j++) {
            if ((*this)(i, j) > t)
                t = (*this)(i, j);
        }
    }
    return t;
}
// 最小值
template <typename T>
T matrix<T>::min(size_type rowi, size_type coli) const {
    size_type r1 = 0, r2 = row, c1 = 0, c2 = col;
    if (rowi >= 0) {
        r1 = rowi;
        r2 = rowi + 1;
    }
    if (coli >= 0) {
        c1 = coli;
        c2 = coli + 1;
    }
    T t = (*this)(r1, c1);
    for (size_type i = r1; i < r2; i++) {
        for (size_type j = c1; j < c2; j++) {
            if ((*this)(i, j) > t)
                t = (*this)(i, j);
        }
    }
    return t;
}
// 求和
template <typename T>
T matrix<T>::sum(size_type rowi, size_type coli) const {
    size_type r1 = 0, r2 = row, c1 = 0, c2 = col;
    if (rowi >= 0) {
        r1 = rowi;
        r2 = rowi + 1;
    }
    if (coli >= 0) {
        c1 = coli;
        c2 = coli + 1;
    }
    T t = (T)0;
    for (size_type i = r1; i < r2; i++) {
        for (size_type j = c1; j < c2; j++) {
            t += (*this)(i, j);
        }
    }
    return t;
}
// 平均值
template <typename T>
T matrix<T>::avg(size_type rowi, size_type coli) const {
    T avg = sum(rowi, coli);
    if (rowi >= 0 && coli >= 0) {
        return avg;
    } else if (rowi >= 0) {
        return avg / col;
    } else if (coli >= 0) {
        return avg / row;
    } else {
        return avg / row / col;
    }
}

// 5)-----------------
// // 特征值
// template <typename T>
// matrix<T> matrix<T>::eigenvalues() const {
//     cv::Mat cvmat = toCVMat();
//     cv::Mat values;
//     eigen(cvmat, values);
//     return matrix(values);
// }
// // 特征向量
// template <typename T>
// matrix<T> matrix<T>::eigenvectors() const {
//     cv::Mat cvmat = toCVMat();
//     cv::Mat values, vectors;
//     eigen(cvmat, values, vectors);
//     return matrix(vectors);
// }

// Refer to https://github.com/HuMeng11/matrix
/*************************************************************************
函数功能：对一个方阵A进行QR分解
输入：需要分解的矩阵A、分解后的正交矩阵Q和上三角矩阵R
输出：无
*************************************************************************/
template <typename T>
void QR(const matrix<T>& A, matrix<T>& Q, matrix<T>& R) {
    SQUARE_CHECK(A.row == A.col);
    matrix<T> col_A(A.row, 1);  //用来存A的每一列
    matrix<T> col_Q(A.row, 1);  //用来存Q的每一列
    //施密特正交化
    for (size_type j = 0; j < A.col; j++) {
        for (size_type i = 0; i < A.row; i++) {  //把A的第j列存入col_A中
            col_A(i, 0) = A(i, j);               // A.slice_col(j, j + 1);
            col_Q(i, 0) = A(i, j);               // A.slice_col(j, j + 1);
        }
        for (size_type k = 0; k < j; k++) {  //计算第j列以前
            R(k, j) = (T)0;
            for (size_type i1 = 0; i1 < col_A.row; i1++) {
                // R=Q'A(Q'即Q的转置) 即Q的第k列和A的第j列做内积
                R(k, j) += col_A(i1, 0) * Q(i1, k);  // Q的第k列
            }
            for (size_type i2 = 0; i2 < A.col; i2++) {
                col_Q(i2, 0) -= R(k, j) * Q(i2, k);
            }
        }
        T temp = col_Q.norm2();
        R(j, j) = temp;
        for (size_type i3 = 0; i3 < Q.col; i3++) {
            //单位化Q
            Q(i3, j) = col_Q(i3, 0) / temp;
        }
    }
}

// 特征值
template <typename T>
matrix<T> matrix<T>::eigenvalues() const {
    SQUARE_CHECK(row == col);
    int NUM = 50;
    matrix<T> temp(*this), temp_Q(row, col), temp_R(row, col);
    for (size_type k = 0; k < NUM; ++k) {
        QR(temp, temp_Q, temp_R);  // A = Q*R
        temp = temp_R * temp_Q;    // A'= R*Q
    }
    matrix<T> values(row, 1);
    for (size_type k = 0; k < temp.col; ++k) {
        values(k, 0) = temp(k, k);
    }
    return values;
}

// Refer to https://github.com/HuMeng11/matrix
/*************************************************************************
函数功能：已知一个矩阵的特征值求对应的特征向量
输入：一个矩阵A、用来存结果的特征向量eigenVector、已知的特征值eigenValue
输出：无
*************************************************************************/
template <typename T>
void Eig(const matrix<T>& A, matrix<T>& eigenVector, matrix<T>& eigenValue) {
    SQUARE_CHECK(A.row == A.col);
    T eValue;
    matrix<T> temp(A.row, A.col);
    for (size_type count = 0; count < A.col; count++) {
        eValue = eigenValue(count, 0);  //当前的特征值
        temp = A;  //这个每次都要重新复制，因为后面会破坏原矩阵
        for (size_type i = 0; i < temp.row; i++) {
            temp(i, i) -= eValue;
        }
        //将temp化为阶梯型矩阵(归一性)对角线值为一
        for (size_type i = 0; i < temp.row - 1; i++) {
            T coe = temp(i, i);
            for (size_type j = i; j < temp.col; j++) {
                temp(i, j) /= coe;  //让对角线值为一
            }
            for (size_type i1 = i + 1; i1 < temp.row; i1++) {
                coe = temp(i1, i);
                for (size_type j1 = i; j1 < temp.col; j1++) {
                    temp(i1, j1) -= coe * temp(i, j1);
                }
            }
        }
        //让最后一行为1
        T sum1 = eigenVector((eigenVector.row - 1), count) = (T)1;
        for (size_type i2 = temp.row - 2; i2 >= 0; i2--) {
            T sum2 = (T)0;
            for (size_type j2 = i2 + 1; j2 < temp.col; j2++) {
                sum2 += temp(i2, j2) * eigenVector(j2, count);
            }
            sum2 = -sum2 / temp(i2, i2);
            sum1 += sum2 * sum2;
            eigenVector(i2, count) = sum2;
        }
        sum1 = sqrt(sum1);  //当前列的模
        for (size_type i = 0; i < eigenVector.row; i++) {
            //单位化
            eigenVector(i, count) /= sum1;
        }
    }
}

// 特征向量
template <typename T>
matrix<T> matrix<T>::eigenvectors() const {
    SQUARE_CHECK(row == col);
    matrix<T> values = eigenvalues();
    matrix<T> vectors(row, col);
    Eig(*this, vectors, values);
    return vectors;
}
// 迹
template <typename T>
T matrix<T>::trace() const {
    SQUARE_CHECK(this->col == this->row);
    T result;
    for (size_type i = 0; i < row; i++) {
        result = result + (*this)(i, i);
    }
    return result;
}
// 代数余子式
template <typename T>
T matrix<T>::algebraic_cofactor(size_type rowi, size_type coli) const {
    matrix submatrix(row - 1, col - 1);
    for (size_type i = 0, si = 0; i < row; i++, si++) {
        if (i == rowi)
            i++;
        for (size_type j = 0, sj = 0; j < col; j++, sj++) {
            if (j == coli)
                j++;
            submatrix(si, sj) = (*this)(i, j);
        }
    }
    if ((rowi + coli) % 2 == 0) {
        return submatrix.determinant();
    } else {
        return -submatrix.determinant();
    }
}
// 逆
template <typename T>
matrix<T> matrix<T>::inverse() const {
    SQUARE_CHECK(this->row == this->col);
    T det = determinant();
    DET_CHECK(det != (T)0);
    matrix adjugate_matrix(row, col);
    for (size_type i = 0; i < row; i++) {
        for (size_type j = 0; j < col; j++) {
            adjugate_matrix(i, j) = algebraic_cofactor(i, j) / det;
        }
    }
    return adjugate_matrix.transposition();
}
// 行列式
template <typename T>
T matrix<T>::determinant() const {
    SQUARE_CHECK(this->row == this->col);
    if (row == 1) {
        return (*this)(0, 0);
    } else if (row == 2) {
        return (*this)(0, 0) * (*this)(1, 1) - (*this)(1, 0) * (*this)(0, 1);
    } else {
        T result;
        for (size_type j = 0; j < col; j++) {
            result += (*this)(0, j) * algebraic_cofactor(0, j);
        }
        return result;
    }
}

// 6)-----------------
// 索引
template <typename T>
vector<T> matrix<T>::operator[](size_type n) const {
    RANGE_CHECK(0 <= n && n < row);
    T* A = this->data;
    vector<T> vec(col);
    for (size_type j = 0; j < col; j++) {
        vec[j] = A[n * this->col + j];
    }
    return vec;
}
// 整形
template <typename T>
matrix<T> matrix<T>::reshape(size_type row_size, size_type col_size) const {
    SIZE_CHECK(row * col == row_size * col_size);
    matrix<T> copy((*this));
    copy.row = row_size;
    copy.col = col_size;
    return copy;
}
// 切片
template <typename T>
matrix<T> matrix<T>::slice_row(size_type r1, size_type r2) const {
    RANGE_CHECK(0 <= r1 && r1 < r2 && r2 <= row);
    matrix R(r2 - r1, col);
    for (size_type i = r1; i < r2; i++) {
        for (size_type j = 0; j < col; j++) {
            R(i - r1, j) = (*this)(i, j);
        }
    }
    return R;
}
template <typename T>
matrix<T> matrix<T>::slice_col(size_type c1, size_type c2) const {
    RANGE_CHECK(0 <= c1 && c1 < c2 && c2 <= col);
    matrix R(row, c2 - c1);
    for (size_type i = 0; i < row; i++) {
        for (size_type j = c1; j < c2; j++) {
            R(i, j - c1) = (*this)(i, j);
        }
    }
    return R;
}
template <typename T>
matrix<T> matrix<T>::slice(size_type c1,
                           size_type c2,
                           size_type r1,
                           size_type r2) const {
    return slice_col(c1, c2).slice_row(r1, r2);
}

// 7)-----------------
```

```cpp
// 卷积
template <typename T>
matrix<T> matrix<T>::convolution(const matrix<T>& other, Mode mode) const {
    size_type r1 = this->row, r2 = other.row, c1 = this->col, c2 = other.col;
    size_type new_r = r1 + r2 - 1, new_c = c1 + c2 - 1;
    matrix<T> result(new_r, new_c);
    for (size_type i = 0; i < new_r; i++) {
        for (size_type j = 0; j < new_c; j++) {
            T temp = 0;
            for (size_type m = 0; m < r1; m++)
                for (size_type n = 0; n < c1; n++)
                    if ((i - m) >= 0 && (i - m) < r2 && (j - n) >= 0 &&
                        (j - n) < c2) {
                        temp = temp + (*this)(m, n) * other((i - m), (j - n));
                    }
            result(i, j) = temp;
        }
    }

    if (mode == SAME) {
        return result.slice((new_r - r1) / 2, new_r - (new_r - r1) / 2,
                            (new_c - c1) / 2, new_c - (new_c - c1) / 2);
    } else if (mode == VALID) {
        return result.slice((new_r - r2) / 2, new_r - (new_r - r2) / 2,
                            (new_c - c2) / 2, new_c - (new_c - c2) / 2);
    } else {
        return result;
    }
}

// 8)-----------------
// 构造器：cv::Mat
template <typename T>
matrix<T>::matrix(const cv::Mat& src)
    : row(src.rows), col(src.cols), data(new T[src.rows * src.cols]()) {
    for (size_type i = 0; i < row; i++) {
        for (size_type j = 0; j < col; j++) {
            (*this)(i, j) = src.at<T>(i, j);
        }
    }
}
// 转cv::mat
template <typename T>
cv::Mat matrix<T>::toCVMat() const {
    int CV_TYPE;
    if (typeid(T) == typeid(unsigned char)) {
        CV_TYPE = CV_8U;
    } else if (typeid(T) == typeid(char)) {
        CV_TYPE = CV_8S;
    } else if (typeid(T) == typeid(unsigned short)) {
        CV_TYPE = CV_16U;
    } else if (typeid(T) == typeid(short)) {
        CV_TYPE = CV_16S;
    } else if (typeid(T) == typeid(int)) {
        CV_TYPE = CV_32S;
    } else if (typeid(T) == typeid(float)) {
        CV_TYPE = CV_32F;
    } else if (typeid(T) == typeid(double)) {
        CV_TYPE = CV_64F;
    } else {
        CV_TYPE = CV_USRTYPE1;
    }
    T* copy = new T[row * col];
    memcpy(copy, data, row * col * sizeof(T));
    return cv::Mat(row, col, CV_TYPE, copy);
}

// 输出
template <typename T>
std::ostream& operator<<(std::ostream& cout, const matrix<T>& mat) {
    if (!mat.isSparse) {
        for (size_type i = 0; i < mat.row; i++) {
            std::cout << ((i == 0) ? '[' : ' ');
            for (size_type j = 0; j < mat.col; j++) {
                std::cout << mat(i, j);
                if (j < mat.col - 1)
                    std::cout << ", ";
            }
            std::cout << ((i < mat.row - 1) ? ";\n" : "]\n");
        }
    } else {
        auto* tmp = mat.sparseData->head->next;
        std::cout << '[';
        while (tmp != NULL) {
            std::cout << '(' << tmp->i << ',' << tmp->j << "):" << tmp->val;
            tmp = tmp->next;
            if (tmp != NULL)
                std::cout << ", ";
        }
        std::cout << "]\n";
    }
    return cout;
}

#endif
```

test.cpp

```
#include <complex>
#include <iostream>
#include "matrix.cpp"
using namespace std;

#define TEST_ALL(ts)                                         \
    for (int i = 0; i < (sizeof(ts) / sizeof(ts[0])); i++) { \
        cout << "\nTEST " << i << " :\n\n";                  \
        ts[i]();                                             \
    }

typedef void (*Test)(void);

void basicTest();
void sparseTest();
void conjTest();
```

```cpp
void operatorTest();
void cvTest();
void determinantTest();
void inverseTest();
void convolutionTest();
void eigenTest();
void exceptionTest();
void avgMaxTest();
void reshapeTest();
void sliceTest();
void traceTest();
void transTest();


static Test tests[] = {basicTest,    sparseTest,     conjTest,
                       operatorTest, cvTest,         determinantTest,
                       inverseTest,  convolutionTest, eigenTest,
                       exceptionTest, avgMaxTest, reshapeTest,
                       sliceTest,  traceTest, transTest};

int main(void) {
    TEST_ALL(tests);
    return 0;
}

void basicTest() {
    double arr0[]{1.1, 2.2, 1.3, 3.3, 6.6, 9.9};
    matrix<double> mat0(arr0, 2, 3);
    cout << mat0;

#ifdef USER_TYPE
    Complex arr1[]{Complex(1, 1), Complex(2, 1), Complex(3, 1),
                   Complex(1, 1), Complex(2, 2), Complex(3, 3)};
    matrix<Complex> mat1(arr1, 2, 3);
    cout << mat1;
#endif

    matrix<int> mat2(3, 3);
    cout << mat2;
}

void sparseTest() {
    int arr0[]{1, 0, 3, 0, 0, 6, 0, 0};
    matrix<int> mat0(arr0, 2, 4);
    cout << mat0;

    mat0.toSparseMatrix();
    cout << mat0;

    mat0.toNormalMatrix();
    cout << mat0;
}

void conjTest() {
    // short
    short arr0[]{1, 2, 1, 3, 6, 3};
    matrix<short> mat0(arr0, 2, 3);
    cout << mat0;
    cout << mat0.conjugation();

#ifdef USER_TYPE
    // Complex (user define type)
    Complex arr1[]{Complex(1, 1), Complex(2, 1), Complex(3, 1),
                   Complex(1, 1), Complex(2, 2), Complex(3, 3)};
    matrix<Complex> mat1(arr1, 2, 3);
    cout << mat1;
    cout << mat1.conjugation();
#endif
    // complex<short>
    complex<short> arr2[]{complex<short>(1, 1), complex<short>(2, 2),
                          complex<short>(1, 1), complex<short>(2, 2),
                          complex<short>(1, 1), complex<short>(3, 3)};
    matrix<complex<short>> mat2(arr2, 2, 3);
    cout << mat2;
    cout << mat2.conjugation();
}

void operatorTest() {
    double arr0[]{1.1, 2.2, 1.3, 3.3, 6.6, 9.9};
    matrix<double> mat0(arr0, 2, 3);
    cout << mat0;

    double arr2[]{1, 2, 1, 3, 6, 3};
    matrix<double> mat2(arr2, 3, 2);
    cout << mat2;
    cout << mat2.avg() << '\n';
    cout << mat2.avg(1) << '\n';
    cout << mat2.avg(1, 1) << '\n';

    cout << mat0 * mat2;
    cout << mat0;

    int arr3[]{1, 2, 1, 3};
    matrix<int> mat3(arr3, 2, 2);
    cout << mat3;

    int arr4[]{2, 2, 3, 3};
    matrix<int> mat4(arr4, 2, 2);
    cout << mat4;

    mat3 = mat4 * mat3;
    mat3 = mat3 * mat4;
    cout << mat3;
}

void cvTest() {
    // Matrix to cv::Mat
    double arr0[]{7.1, 2.2, 7.2, 3.3, 7.3, 4.4};
    matrix<double> mat0(arr0, 3, 2);
    cout << mat0;

    cv::Mat cvmat0 = mat0.toCVMat();
    cout << cvmat0 << '\n';

    // cv::Mat to Matrix
    double arr1[]{7.1, 2.2, 7.2, 3.3, 7.3, 4.4};
    cv::Mat cvmat1(2, 3, CV_64F, arr1);
    cout << cvmat1 << '\n';

    matrix<double> mat1(cvmat1);
    cout << mat1;
}

void determinantTest() {
    double arr0[]{1, 2, 1, 1};
    matrix<double> mat0(arr0, 2, 2);
    cout << mat0;
    cout << mat0.determinant() << '\n';  // -1

    double arr1[]{4, 9, 2, 3, 5, 7, 8, 1, 6};
    matrix<double> mat1(arr1, 3, 3);
    cout << mat1;
    cout << mat1.determinant() << '\n';  // 360

    complex<float> arr2[]{
        complex<float>(1, 1), complex<float>(2, 2), complex<float>(2, 2),
        complex<float>(1, 1), complex<float>(2, 2), complex<float>(3, 3),
        complex<float>(1, 1), complex<float>(3, 3), complex<float>(2, 2)};
    matrix<complex<float>> mat2(arr2, 3, 3);
    cout << mat2;
    cout << mat2.determinant() << '\n';
}

void inverseTest() {
    double arr0[]{1, 2, 1, 1};
    matrix<double> mat0(arr0, 2, 2);
    cout << mat0;
    cout << mat0.inverse();
    /*
    -1  2
     1 -1
    */

    double arr1[]{4, 9, 2, 3, 5, 7, 8, 1, 6};
    // double arr1[]{1, 2, 2, 1, 2, 3, 1, 3, 2};
    matrix<double> mat1(arr1, 3, 3);
    cout << mat1;
    cout << mat1.inverse();
    /*
    0.0639      -0.1444     0.1472
    0.1056      0.0222      -0.0611
    -0.1028     0.1889      -0.0194
    */

    complex<float> arr2[]{
        complex<float>(1, 1), complex<float>(1, 1), complex<float>(1, 1),
        complex<float>(1, 1), complex<float>(2, 2), complex<float>(3, 3),
        complex<float>(1, 1), complex<float>(3, 3), complex<float>(2, 2)};
    matrix<complex<float>> mat2(arr2, 3, 3);
    cout << mat2;
    cout << mat2.inverse();
}

void convolutionTest() {
    double arr0[]{4, 9, 2, 3, 5, 7, 8, 1, 6, 1, 4, 9, 2,
                  3, 5, 7, 8, 1, 6, 1, 7, 8, 1, 6, 1};
    matrix<double> mat0(arr0, 5, 5);
    cout << mat0;

    double arr1[]{1, 0, 0, 0, 1, 0, 0, 1, 0};
    matrix<double> mat1(arr1, 3, 3);
    cout << mat1;

    cout << mat0.convolution(mat1);
}

void eigenTest() {

    double arr0[]{1, 1, 1, 1, 2, 10, 1, 10, 1};
    matrix<double> mat0(arr0, 3, 3);
    cout << mat0;

    cout << mat0.eigenvalues();
    cout << mat0.eigenvectors();

    /*
    特征值:
    特征值1:    11.6993
    特征值2:    0.8133
    特征值3:    -8.5126
    特征向量:
    向量1      向量2      向量3
    0.1310      0.9914      0.0038
    0.7181      -0.0975     0.6890
    0.6835      -0.0876     -0.7247
    */
}

void exceptionTest() {
    try
    {
    double arr0[]{4, 9, 2, 3, 5, 7, 8, 1, 6, 1, 4, 9, 2,
                  3, 5, 7, 8, 1, 6, 1, 7, 8, 1, 6, 1};
    matrix<double> mat0(arr0, 5, 5);
    cout << mat0;

    double arr1[]{1, 0, 0, 0, 1, 0, 0, 1, 0};
    matrix<double> mat1(arr1, 3, 3);
    cout << mat1;

    cout << mat0*mat1;
    }
    catch(std::exception& e)
    {
        std::cerr << e.what() << '\n';
    }

}

void avgMaxTest(){
    double arr0[]{4, 9, 2, 3, 5, 7, 8, 1, 6, 1, 4, 9, 2,
                  3, 5, 7, 8, 1, 6, 1, 7, 8, 1, 6, 1};
    matrix<double> mat0(arr0, 5, 5);
    cout << mat0;
    double a = mat0.avg(0,-1);
    double b = mat0.sum(0,-1);
    cout << "The sum of the first row is: "<< b << endl;
    cout << "The avg of the first row is: "<< a << endl;
}

void reshapeTest() {
    double arr0[] = {1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1};
    matrix<double> mat0(arr0, 4, 4);
    cout << mat0;
    cout << mat0.reshape(8,2);
    cout << mat0.reshape(2,8);
    cout << mat0.reshape(2,8).reshape(4,4);
}


void sliceTest() {
    double arr0[]{4, 9, 2, 3, 5, 7, 8, 1, 6, 1, 4, 9, 2,
                  3, 5, 7, 8, 1, 6, 1, 7, 8, 1, 6, 1};
    matrix<double> mat0(arr0, 5, 5);
    cout << mat0;

    cout << mat0.slice_col(0,2);
    cout << mat0.slice_row(1,3);
    cout << mat0.slice(1,3,1,3);

}


void traceTest() {
    double arr0[]{4, 9, 2, 3, 5, 7, 8, 1, 6, 1, 4, 9, 2,
                  3, 5, 7, 8, 1, 6, 1, 7, 8, 1, 6, 1};
    matrix<double> mat0(arr0, 5, 5);
    cout << mat0;
    cout << mat0.trace() << endl;

    double arr1[]
{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> mat1(arr1, 5, 5);
    cout << mat1;
    cout << mat1.trace() << endl;

}

void transTest(){
    double arr0[]{4, 9, 2, 3, 5, 7, 8, 1, 6, 1, 4, 9, 2,
                  3, 5, 7, 8, 1, 6, 1, 7, 8, 1, 6, 1};
    matrix<double> mat0(arr0, 5, 5);
    cout << mat0;
    cout << mat0.transposition();
}
```

Part 3 - Result & Verification

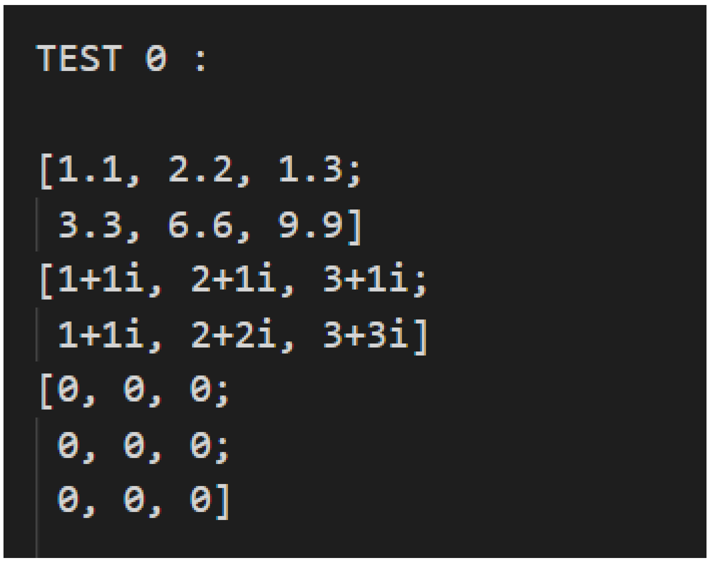
Test case #0 (basicTest)

input:

```
void basicTest() {
    double arr0[]{1.1, 2.2, 1.3, 3.3, 6.6, 9.9};
    matrix<double> mat0(arr0, 2, 3);
    cout << mat0;

#ifdef USER_TYPE
    Complex arr1[]{Complex(1, 1), Complex(2, 1), Complex(3, 1),
                   Complex(1, 1), Complex(2, 2), Complex(3, 3)};
    matrix<Complex> mat1(arr1, 2, 3);
    cout << mat1;
#endif

    matrix<int> mat2(3, 3);
    cout << mat2;
}
```

output:

```
TEST 0 :

[1.1, 2.2, 1.3;
 3.3, 6.6, 9.9]
[1+1i, 2+1i, 3+1i;
 1+1i, 2+2i, 3+3i]
[0, 0, 0;
 0, 0, 0;
 0, 0, 0]
```



```
TEST 0 :

[1.1, 2.2, 1.3;
 3.3, 6.6, 9.9]
[1+1i, 2+1i, 3+1i;
 1+1i, 2+2i, 3+3i]
[0, 0, 0;
 0, 0, 0;
 0, 0, 0]
```

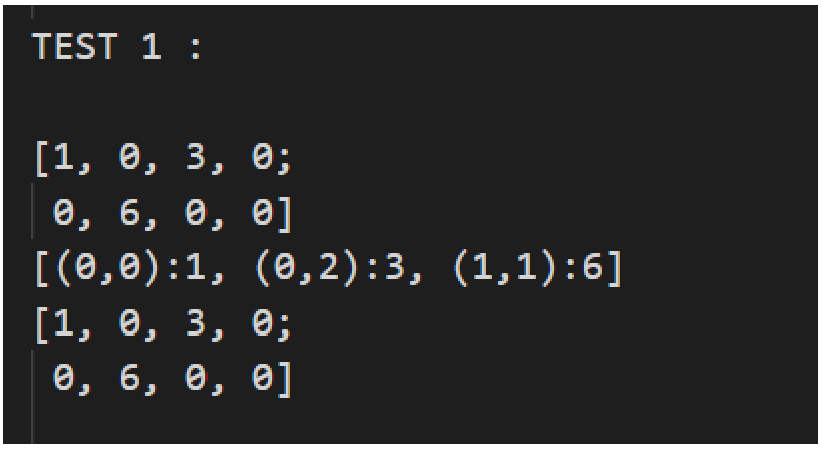
Test case #1 (sparseTest)

input:

```
void sparseTest() {
    int arr0[]{1, 0, 3, 0, 0, 6, 0, 0};
    matrix<int> mat0(arr0, 2, 4);
    cout << mat0;

    mat0.toSparseMatrix();
    cout << mat0;

    mat0.toNormalMatrix();
    cout << mat0;
}
```

output:

```
TEST 1 :

[1, 0, 3, 0;
 0, 6, 0, 0]
[(0,0):1, (0,2):3, (1,1):6]
[1, 0, 3, 0;
 0, 6, 0, 0]
```



```
TEST 1 :

[1, 0, 3, 0;
 0, 6, 0, 0]
[(0,0):1, (0,2):3, (1,1):6]
[1, 0, 3, 0;
 0, 6, 0, 0]
```

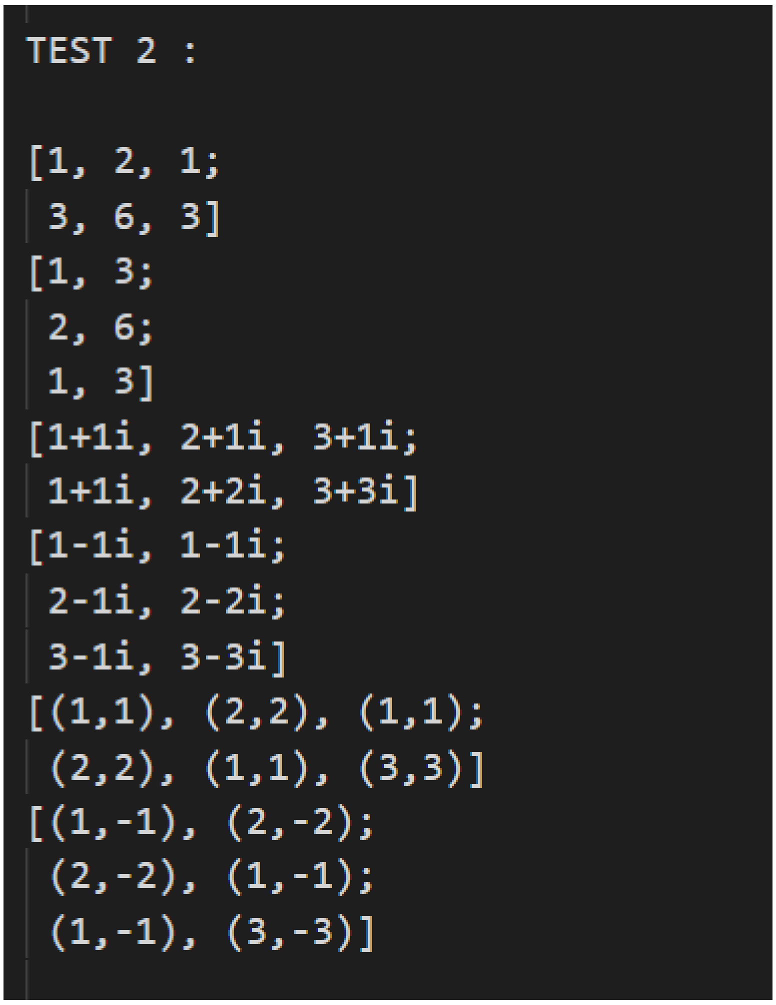
Test case #2 (conjTest)

input:

```
void conjTest() {
    // short
    short arr0[]{1, 2, 1, 3, 6, 3};
    matrix<short> mat0(arr0, 2, 3);
    cout << mat0;
    cout << mat0.conjugation();

#ifdef USER_TYPE
    // Complex (user define type)
    Complex arr1[]{Complex(1, 1), Complex(2, 1), Complex(3, 1),
                   Complex(1, 1), Complex(2, 2), Complex(3, 3)};
    matrix<Complex> mat1(arr1, 2, 3);
    cout << mat1;
    cout << mat1.conjugation();
#endif
    // complex<short>
    complex<short> arr2[]{complex<short>(1, 1), complex<short>(2, 2),
                          complex<short>(1, 1), complex<short>(2, 2),
                          complex<short>(1, 1), complex<short>(3, 3)};
    matrix<complex<short>> mat2(arr2, 2, 3);
    cout << mat2;
    cout << mat2.conjugation();
}
```

output:

```
TEST 2 :

[1, 2, 1;
 3, 6, 3]
[1, 3;
 2, 6;
 1, 3]
[1+1i, 2+1i, 3+1i;
 1+1i, 2+2i, 3+3i]
[1-1i, 1-1i;
 2-1i, 2-2i;
 3-1i, 3-3i]
[(1,1), (2,2), (1,1);
 (2,2), (1,1), (3,3)]
[(1,-1), (2,-2);
 (2,-2), (1,-1);
 (1,-1), (3,-3)]
```



```
TEST 2 :

[1, 2, 1;
 3, 6, 3]
[1, 3;
 2, 6;
 1, 3]
[1+1i, 2+1i, 3+1i;
 1+1i, 2+2i, 3+3i]
[1-1i, 1-1i;
 2-1i, 2-2i;
 3-1i, 3-3i]
[(1,1), (2,2), (1,1);
 (2,2), (1,1), (3,3)]
[(1,-1), (2,-2);
 (2,-2), (1,-1);
 (1,-1), (3,-3)]
```

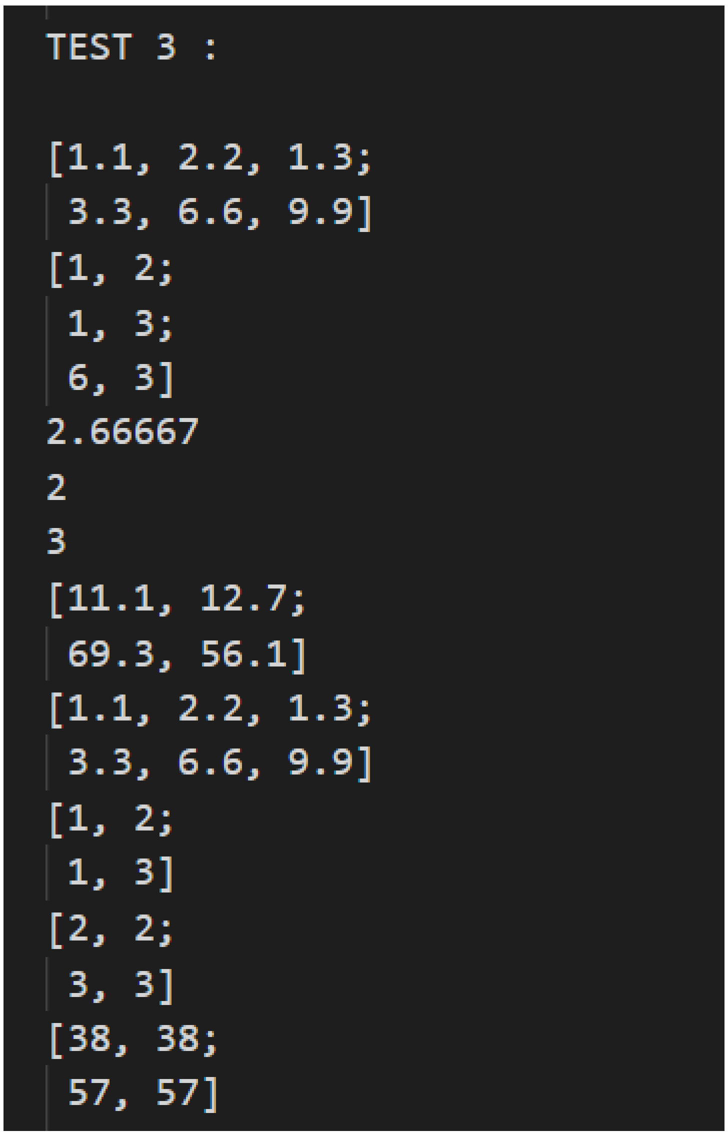
Test case #3 (operatorTest)

input:

```
void operatorTest() {
    double arr0[]{1.1, 2.2, 1.3, 3.3, 6.6, 9.9};
    matrix<double> mat0(arr0, 2, 3);
    cout << mat0;

    double arr2[]{1, 2, 1, 3, 6, 3};
    matrix<double> mat2(arr2, 3, 2);
    cout << mat2;
    cout << mat2.avg() << '\n';
    cout << mat2.avg(1) << '\n';
    cout << mat2.avg(1, 1) << '\n';

    cout << mat0 * mat2;
    cout << mat0;

    int arr3[]{1, 2, 1, 3};
    matrix<int> mat3(arr3, 2, 2);
    cout << mat3;

    int arr4[]{2, 2, 3, 3};
    matrix<int> mat4(arr4, 2, 2);
    cout << mat4;

    mat3 = mat4 * mat3;
    mat3 = mat3 * mat4;
    cout << mat3;
}
```

output:

```
TEST 3 :

[1.1, 2.2, 1.3;
 3.3, 6.6, 9.9]
[1, 2;
 1, 3;
 6, 3]
2.66667
2
3
[11.1, 12.7;
 69.3, 56.1]
[1.1, 2.2, 1.3;
 3.3, 6.6, 9.9]
[1, 2;
 1, 3]
[2, 2;
 3, 3]
[38, 38;
 57, 57]
```



```
TEST 3 :

[1.1, 2.2, 1.3;
 3.3, 6.6, 9.9]
[1, 2;
 1, 3;
 6, 3]
2.66667
2
3
[11.1, 12.7;
 69.3, 56.1]
[1.1, 2.2, 1.3;
 3.3, 6.6, 9.9]
[1, 2;
 1, 3]
[2, 2;
 3, 3]
[38, 38;
 57, 57]
```

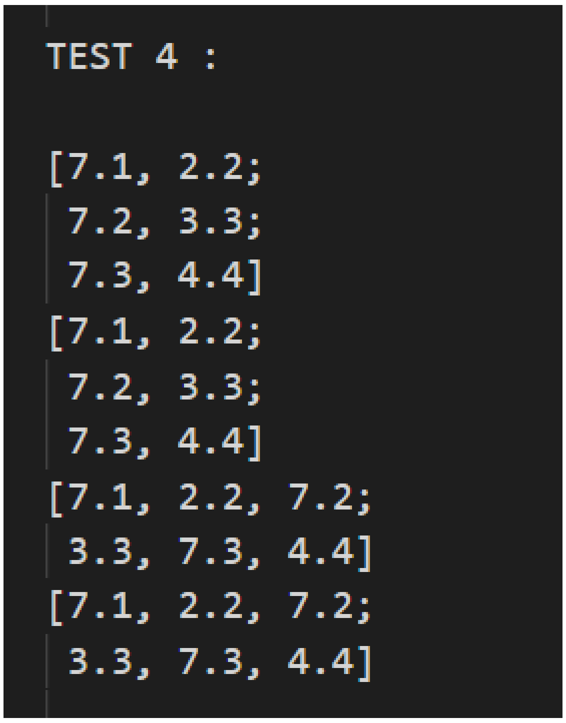
Test case #4 (cvTest)

input:

```
void cvTest() {
    // Matrix to cv::Mat
    double arr0[]{7.1, 2.2, 7.2, 3.3, 7.3, 4.4};
    matrix<double> mat0(arr0, 3, 2);
    cout << mat0;

    cv::Mat cvmat0 = mat0.toCVMat();
    cout << cvmat0 << '\n';

    // cv::Mat to Matrix
    double arr1[]{7.1, 2.2, 7.2, 3.3, 7.3, 4.4};
    cv::Mat cvmat1(2, 3, CV_64F, arr1);
    cout << cvmat1 << '\n';

    matrix<double> mat1(cvmat1);
    cout << mat1;
}
```

output:

```
TEST 4 :

[7.1, 2.2;
 7.2, 3.3;
 7.3, 4.4]
[7.1, 2.2;
 7.2, 3.3;
 7.3, 4.4]
[7.1, 2.2, 7.2;
 3.3, 7.3, 4.4]
[7.1, 2.2, 7.2;
 3.3, 7.3, 4.4]
```



```
TEST 4 :

[7.1, 2.2;
 7.2, 3.3;
 7.3, 4.4]
[7.1, 2.2;
 7.2, 3.3;
 7.3, 4.4]
[7.1, 2.2, 7.2;
 3.3, 7.3, 4.4]
[7.1, 2.2, 7.2;
 3.3, 7.3, 4.4]
```

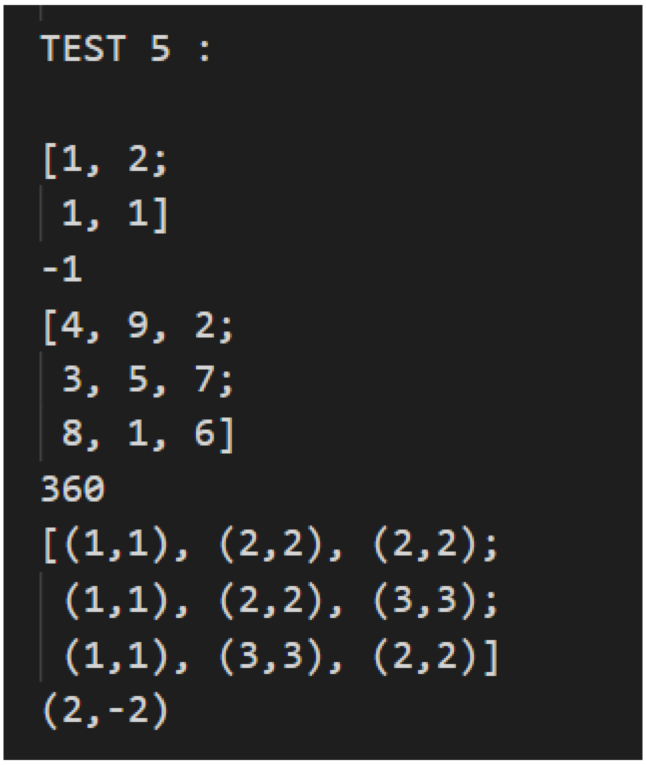
Test case #5 (determinantTest)

input:

```
void determinantTest() {
    double arr0[]{1, 2, 1, 1};
    matrix<double> mat0(arr0, 2, 2);
    cout << mat0;
    cout << mat0.determinant() << '\n';  // -1

    double arr1[]{4, 9, 2, 3, 5, 7, 8, 1, 6};
    matrix<double> mat1(arr1, 3, 3);
    cout << mat1;
    cout << mat1.determinant() << '\n';  // 360

    complex<float> arr2[]{
        complex<float>(1, 1), complex<float>(2, 2), complex<float>(2, 2),
        complex<float>(1, 1), complex<float>(2, 2), complex<float>(3, 3),
        complex<float>(1, 1), complex<float>(3, 3), complex<float>(2, 2)};
    matrix<complex<float>> mat2(arr2, 3, 3);
    cout << mat2;
    cout << mat2.determinant() << '\n';
}
```

output:

```
TEST 5 :

[1, 2;
 1, 1]
-1
[4, 9, 2;
 3, 5, 7;
 8, 1, 6]
360
[(1,1), (2,2), (2,2);
 (1,1), (2,2), (3,3);
 (1,1), (3,3), (2,2)]
(2,-2)
```



```
TEST 5 :

[1, 2;
 1, 1]
-1
[4, 9, 2;
 3, 5, 7;
 8, 1, 6]
360
[(1,1), (2,2), (2,2);
 (1,1), (2,2), (3,3);
 (1,1), (3,3), (2,2)]
(2,-2)
```

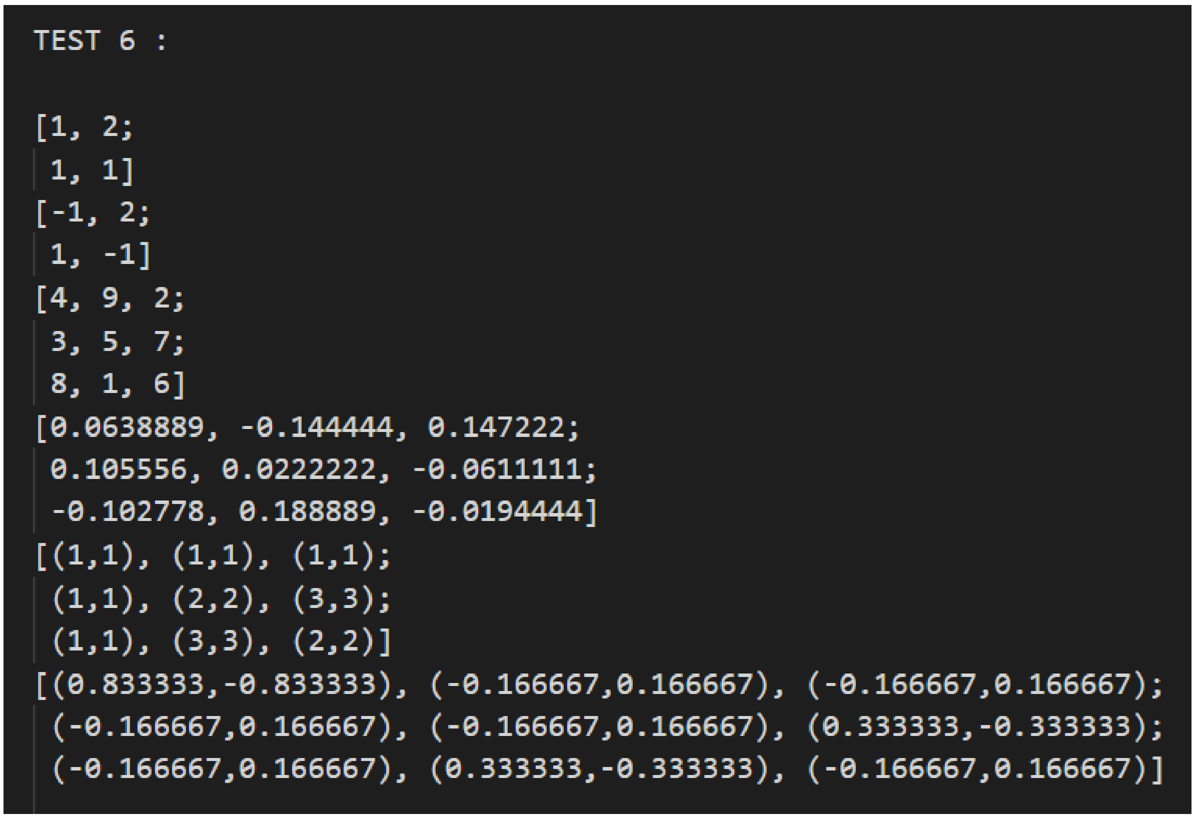
Test case #6 (inverseTest)

input:

```
void inverseTest() {
    double arr0[]{1, 2, 1, 1};
    matrix<double> mat0(arr0, 2, 2);
    cout << mat0;
    cout << mat0.inverse();
    /*
    -1  2
     1 -1
    */

    double arr1[]{4, 9, 2, 3, 5, 7, 8, 1, 6};
    // double arr1[]{1, 2, 2, 1, 2, 3, 1, 3, 2};
    matrix<double> mat1(arr1, 3, 3);
    cout << mat1;
    cout << mat1.inverse();
    /*
    0.0639      -0.1444     0.1472
    0.1056      0.0222      -0.0611
    -0.1028     0.1889      -0.0194
    */

    complex<float> arr2[]{
        complex<float>(1, 1), complex<float>(1, 1), complex<float>(1, 1),
        complex<float>(1, 1), complex<float>(2, 2), complex<float>(3, 3),
        complex<float>(1, 1), complex<float>(3, 3), complex<float>(2, 2)};
    matrix<complex<float>> mat2(arr2, 3, 3);
    cout << mat2;
    cout << mat2.inverse();
}
```

output:

```
TEST 6 :

[1, 2;
 1, 1]
[-1, 2;
 1, -1]
[4, 9, 2;
 3, 5, 7;
 8, 1, 6]
[0.0638889, -0.144444, 0.147222;
 0.105556, 0.0222222, -0.0611111;
 -0.102778, 0.188889, -0.0194444]
[(1,1), (1,1), (1,1);
 (1,1), (2,2), (3,3);
 (1,1), (3,3), (2,2)]
[(0.833333,-0.833333), (-0.166667,0.166667), (-0.166667,0.166667);
 (-0.166667,0.166667), (-0.166667,0.166667), (0.333333,-0.333333);
 (-0.166667,0.166667), (0.333333,-0.333333), (-0.166667,0.166667)]
```



```
TEST 6 :

[1, 2;
 1, 1]
[-1, 2;
 1, -1]
[4, 9, 2;
 3, 5, 7;
 8, 1, 6]
[0.0638889, -0.144444, 0.147222;
 0.105556, 0.0222222, -0.0611111;
 -0.102778, 0.188889, -0.0194444]
[(1,1), (1,1), (1,1);
 (1,1), (2,2), (3,3);
 (1,1), (3,3), (2,2)]
[(0.833333,-0.833333), (-0.166667,0.166667), (-0.166667,0.166667);
 (-0.166667,0.166667), (-0.166667,0.166667), (0.333333,-0.333333);
 (-0.166667,0.166667), (0.333333,-0.333333), (-0.166667,0.166667)]
```

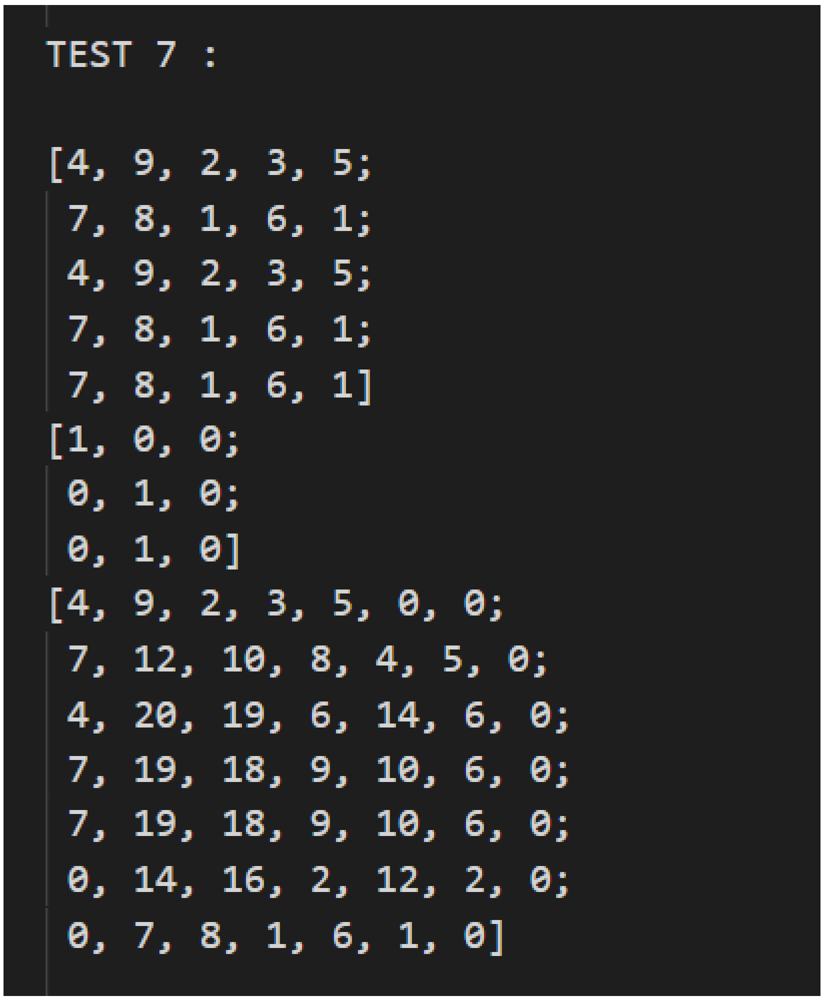
Test case #7 (convolutionTest)

input:

```
void convolutionTest() {
    double arr0[]{4, 9, 2, 3, 5, 7, 8, 1, 6, 1, 4, 9, 2,
                  3, 5, 7, 8, 1, 6, 1, 7, 8, 1, 6, 1};
    matrix<double> mat0(arr0, 5, 5);
    cout << mat0;

    double arr1[]{1, 0, 0, 0, 1, 0, 0, 1, 0};
    matrix<double> mat1(arr1, 3, 3);
    cout << mat1;

    cout << mat0.convolution(mat1);
}
```

output:

```
TEST 7 :

[4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 7, 8, 1, 6, 1]
[1, 0, 0;
 0, 1, 0;
 0, 1, 0]
[4, 9, 2, 3, 5, 0, 0;
 7, 12, 10, 8, 4, 5, 0;
 4, 20, 19, 6, 14, 6, 0;
 7, 19, 18, 9, 10, 6, 0;
 7, 19, 18, 9, 10, 6, 0;
 0, 14, 16, 2, 12, 2, 0;
 0, 7, 8, 1, 6, 1, 0]
```



```
TEST 7 :

[4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 7, 8, 1, 6, 1]
[1, 0, 0;
 0, 1, 0;
 0, 1, 0]
[4, 9, 2, 3, 5, 0, 0;
 7, 12, 10, 8, 4, 5, 0;
 4, 20, 19, 6, 14, 6, 0;
 7, 19, 18, 9, 10, 6, 0;
 7, 19, 18, 9, 10, 6, 0;
 0, 14, 16, 2, 12, 2, 0;
 0, 7, 8, 1, 6, 1, 0]
```

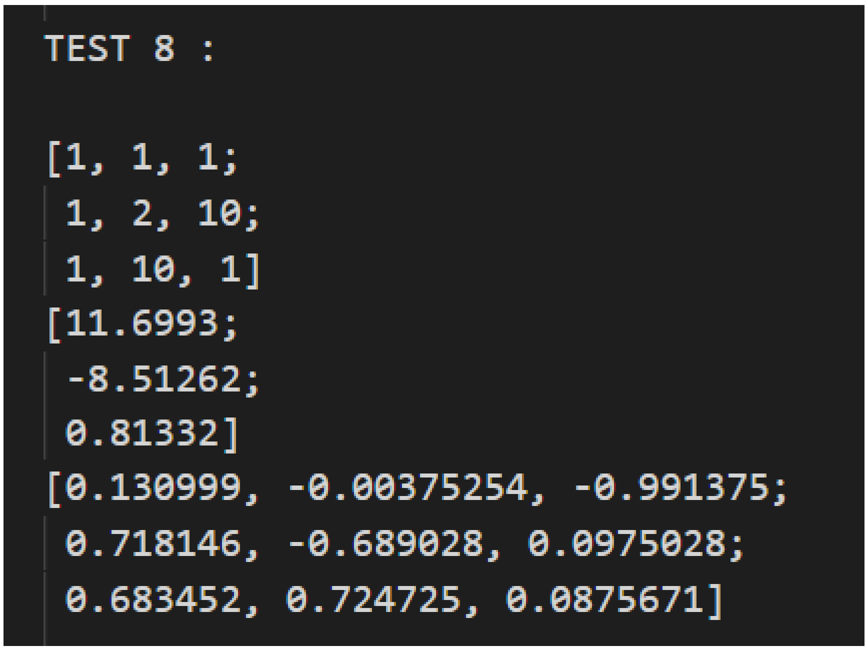
Test case #8 (eigenTest)

input:

```
void eigenTest() {

    double arr0[]{1, 1, 1, 1, 2, 10, 1, 10, 1};
    matrix<double> mat0(arr0, 3, 3);
    cout << mat0;

    cout << mat0.eigenvalues();
    cout << mat0.eigenvectors();
    /*
    特征值:
    特征值1:    11.6993
    特征值2:    0.8133
    特征值3:    -8.5126
    特征向量:
    向量1      向量2      向量3
    0.1310      0.9914      0.0038
    0.7181      -0.0975     0.6890
    0.6835      -0.0876     -0.7247
    */
}
```

output:

```
TEST 8 :

[1, 1, 1;
 1, 2, 10;
 1, 10, 1]
[11.6993;
 -8.51262;
 0.81332]
[0.130999, -0.00375254, -0.991375;
 0.718146, -0.689028, 0.0975028;
 0.683452, 0.724725, 0.0875671]
```



```
TEST 8 :

[1, 1, 1;
 1, 2, 10;
 1, 10, 1]
[11.6993;
 -8.51262;
 0.81332]
[0.130999, -0.00375254, -0.991375;
 0.718146, -0.689028, 0.0975028;
 0.683452, 0.724725, 0.0875671]
```

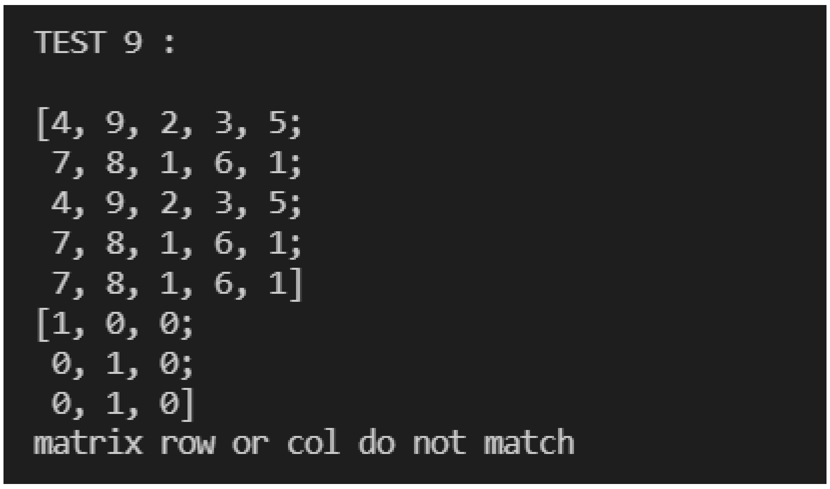
Test case #9 (exceptionTest)

input:

```
void exceptionTest() {
    try
    {
    double arr0[]{4, 9, 2, 3, 5, 7, 8, 1, 6, 1, 4, 9, 2,
                  3, 5, 7, 8, 1, 6, 1, 7, 8, 1, 6, 1};
    matrix<double> mat0(arr0, 5, 5);
    cout << mat0;

    double arr1[]{1, 0, 0, 0, 1, 0, 0, 1, 0};
    matrix<double> mat1(arr1, 3, 3);
    cout << mat1;

    cout << mat0*mat1;
    }
    catch(std::exception& e)
    {
        std::cerr << e.what() << '\n';
    }

}
```

output:

```
TEST 9 :

[4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 7, 8, 1, 6, 1]
[1, 0, 0;
 0, 1, 0;
 0, 1, 0]
matrix row or col do not match
```



```
TEST 9 :

[4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 7, 8, 1, 6, 1]
[1, 0, 0;
 0, 1, 0;
 0, 1, 0]
matrix row or col do not match
```

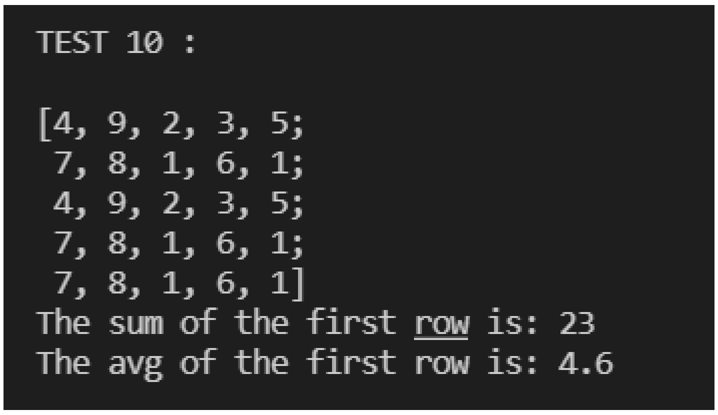
Test case #10 (avgMaxTest)

input:

```
void avgMaxTest(){
    double arr0[]{4, 9, 2, 3, 5, 7, 8, 1, 6, 1, 4, 9, 2,
                  3, 5, 7, 8, 1, 6, 1, 7, 8, 1, 6, 1};
    matrix<double> mat0(arr0, 5, 5);
    cout << mat0;
    double a = mat0.avg(0,-1);
    double b = mat0.sum(0,-1);
    cout << "The sum of the first row is: "<< b << endl;
    cout << "The avg of the first row is: "<< a << endl;
}
```

output:

```
TEST 10 :

[4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 7, 8, 1, 6, 1]
The sum of the first row is: 23
The avg of the first row is: 4.6
```



```
TEST 10 :

[4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 7, 8, 1, 6, 1]
The sum of the first row is: 23
The avg of the first row is: 4.6
```

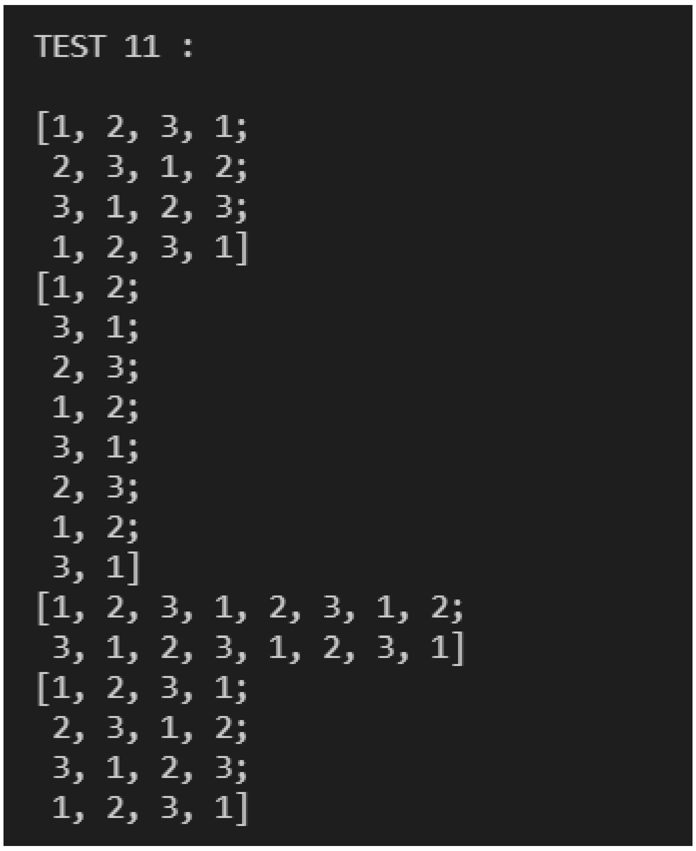
Test case #11 (reshapeTest)

input:

```
void reshapeTest() {
    double arr0[] = {1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1};
    matrix<double> mat0(arr0, 4, 4);
    cout << mat0;
    cout << mat0.reshape(8,2);
    cout << mat0.reshape(2,8);
    cout << mat0.reshape(2,8).reshape(4,4);
}
```

output:

```
TEST 11 :

[1, 2, 3, 1;
 2, 3, 1, 2;
 3, 1, 2, 3;
 1, 2, 3, 1]
[1, 2;
 3, 1;
 2, 3;
 1, 2;
 3, 1;
 2, 3;
 1, 2;
 3, 1]
[1, 2, 3, 1, 2, 3, 1, 2;
 3, 1, 2, 3, 1, 2, 3, 1]
[1, 2, 3, 1;
 2, 3, 1, 2;
 3, 1, 2, 3;
 1, 2, 3, 1]
```


```
TEST 11 :

[1, 2, 3, 1;
 2, 3, 1, 2;
 3, 1, 2, 3;
 1, 2, 3, 1]
[1, 2;
 3, 1;
 2, 3;
 1, 2;
 3, 1;
 2, 3;
 1, 2;
 3, 1]
[1, 2, 3, 1, 2, 3, 1, 2;
 3, 1, 2, 3, 1, 2, 3, 1]
[1, 2, 3, 1;
 2, 3, 1, 2;
 3, 1, 2, 3;
 1, 2, 3, 1]
```

Test case #12 (sliceTest)

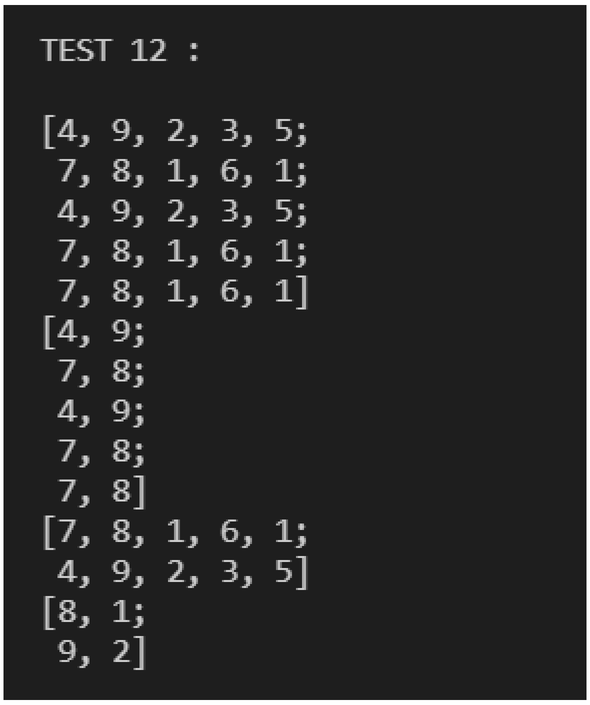
input:

```
void sliceTest() {
    double arr0[]{4, 9, 2, 3, 5, 7, 8, 1, 6, 1, 4, 9, 2,
                  3, 5, 7, 8, 1, 6, 1, 7, 8, 1, 6, 1};
    matrix<double> mat0(arr0, 5, 5);
    cout << mat0;

    cout << mat0.slice_col(0,2);
    cout << mat0.slice_row(1,3);
    cout << mat0.slice(1,3,1,3);

}
```

output:

```
TEST 12 :

[4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 7, 8, 1, 6, 1]
[4, 9;
 7, 8;
 4, 9;
 7, 8;
 7, 8]
[7, 8, 1, 6, 1;
 4, 9, 2, 3, 5]
[8, 1;
```

```
 9, 2]
```



```
TEST 12 :

[4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 7, 8, 1, 6, 1]
[4, 9;
 7, 8;
 4, 9;
 7, 8;
 7, 8]
[7, 8, 1, 6, 1;
 4, 9, 2, 3, 5]
[8, 1;
 9, 2]
```

Test case #13 (traceTest)

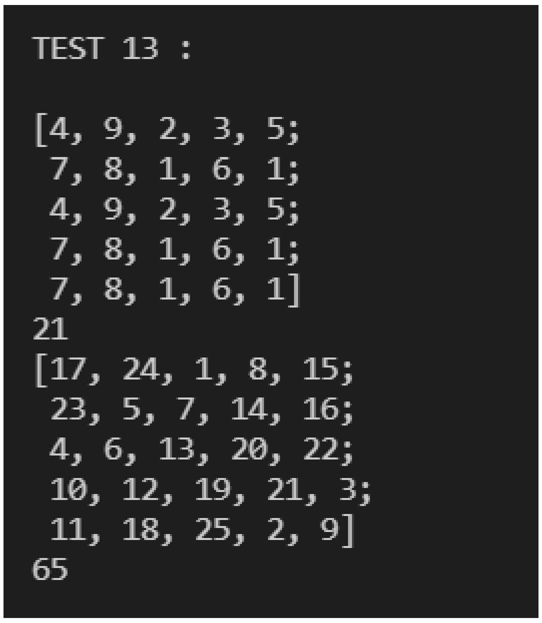
input:

```
void traceTest() {
    double arr0[]{4, 9, 2, 3, 5, 7, 8, 1, 6, 1, 4, 9, 2,
                  3, 5, 7, 8, 1, 6, 1, 7, 8, 1, 6, 1};
    matrix<double> mat0(arr0, 5, 5);
    cout << mat0;
    cout << mat0.trace() << endl;

    double arr1[]
{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> mat1(arr1, 5, 5);
    cout << mat1;
    cout << mat1.trace() << endl;

}
```

output:

```
TEST 13 :

[4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 7, 8, 1, 6, 1]
21
[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]
```

```
65
```



```
TEST 13 :

[4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 7, 8, 1, 6, 1]
21
[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]
65
```

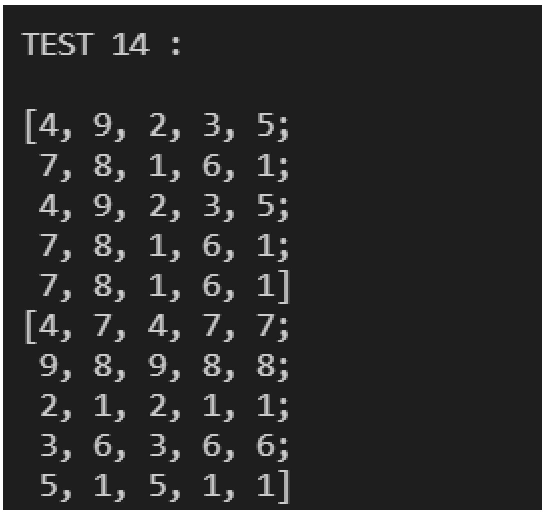
Test case #14 (transTest)

input:

```
void transTest(){
    double arr0[]{4, 9, 2, 3, 5, 7, 8, 1, 6, 1, 4, 9, 2,
                  3, 5, 7, 8, 1, 6, 1, 7, 8, 1, 6, 1};
    matrix<double> mat0(arr0, 5, 5);
    cout << mat0;
    cout << mat0.transposition();
}
```

output:

```
TEST 14 :

[4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 7, 8, 1, 6, 1]
[4, 7, 4, 7, 7;
 9, 8, 9, 8, 8;
 2, 1, 2, 1, 1;
 3, 6, 3, 6, 6;
 5, 1, 5, 1, 1]
```



```
TEST 14 :

[4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 4, 9, 2, 3, 5;
 7, 8, 1, 6, 1;
 7, 8, 1, 6, 1]
[4, 7, 4, 7, 7;
 9, 8, 9, 8, 8;
 2, 1, 2, 1, 1;
 3, 6, 3, 6, 6;
 5, 1, 5, 1, 1]
```

Part 4 - Difficulties & Solutions

（1）When we use the inverse of matrix, determinant and other methods, we report an error when we measure the complex number at the beginning, and then change the number of 1 in the method to T(1); Later, we find that when we decompose the method with LU, some number will be nan or inf. Then we give up the LU decomposition method and use the algebraic cofactor inversion method.

（2）The storage of sparse matrix. Let's first determine whether the matrix is sparse, and if so, use the sparse matrix to store it. We choose to use the form of linked list to store. When we need to use this sparse matrix, we first convert it into a normal matrix for calculation, and then we convert it back to the form of sparse matrix after using it. This saves storage space.