CS205 C/C++ Programming - Project - Building a Library for Matrix Computation

Name&SID: 董骁 11813104, 许博清 11812910

Part 1 - Analysis

Matrix is an important concept introduced in linear algebra. Matrix calculation is widely used in many practical applications, such as image processing and machine learning. Programmers can indeed use many different existing libraries, and in certain cases, programmers are required to design their own matrix calculation libraries for specific implementations. This project will build a new library (do not attempt to directly copy codes from other existing library) that can perform the following operations on the matrix:

1) It supports all matrix sizes, from small fixed-size matrices to arbitrarily large dense matrices, and even sparse matrices (Add: try to use efficient ways to store the sparse matrices). (10 points)

We decided to use `vector` to realize this matrix, and each `vector` contains a few of `vectors` with the same size to simulate the tow-dimensional matrices.

When it comes to the initialization of sparse matrices, we give a solution that we can input all the non-zero value's index and their value to set the matrix and make other places to be 0.

2) It supports all standard numeric types, including std::complex, integers, and is easily extensible to custom numeric types. (10 points)

We decided make our Matrix class support two kinds of types to represent all the other types. The fires type is `double` that can deal with the condition when we want to input numbers in `int`, `float`, `long long` and `double`. Another type is the `std::complex` that can support all numbers.

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. (20 points)

For addition, subtraction, scalar multiplication and scalar division, we can use the operator `+`, `-`, `*` and `/` respectively to realize so that it can be easier to understand. Other complicated calculations we used functions to realize.

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). (10 points)

It is to some extent much easier than the last problem, we just need to traverse all elements in the matrix and get the result we want.

5) It supports computing eigenvalues and eigenvectors, calculating traces, computing inverse and computing determinant. (10 points)

6) It supports the operations of reshape and slicing. (10 points)

The main point is new a matrix and then set its value to realize the reshape and slicing.

7) It supports convolutional operations of two matrices. (10 points)

8) It supports to transfer the matrix from OpenCV to the matrix of this library and vice versa. (10 points)

9) It should process likely exceptions as much as possible. (10 points)

We have tried our best to think about all the exceptions during the input and other situations, and will give hints if some exceptions occurs, then the whole program will end.

Part 2 - Code

```
//
// Created by ccyys on 2021/5/22.
//

#ifndef MATRIX_H
#define MATRIX_H

#include <iostream>
#include <cmath>
#include <algorithm>
#include <vector>

using namespace std;
```

```cpp
template<class T>
class Matrix {
public:
    void setValue(vector<vector<T>>);

    void showMatrix();

    Matrix elementWiseMultiple(Matrix &a);

    T findMax();

    T findMaxByRow(int rowNum);

    T findMaxByCol(int colNum);

    T findMin();

    T findMinByRow(int rowNum);

    T findMinByCol(int colNum);

    T getSum();

    T getSumByRow(int rowNum);

    T getSumByCol(int colNum);

    T getAverage();

    T getAverageByRow(int rowNum);

    T getAverageByCol(int colNum);

    Matrix operator+(Matrix &a);

    Matrix operator-(Matrix &a);

    Matrix operator*(Matrix &a);

    Matrix operator*(vector<T> a);

    Matrix operator/(T a);

    Matrix operator*(T a);

    Matrix transposition();

    Matrix conjugation();
    T dotProduct(Matrix &other);

    T crossProduct(Matrix &other);

    bool isVector() const;

    T det(Matrix &a);


    T eigenValue();

    Matrix convolution(Matrix &a);

    Matrix inverse();


    friend Matrix<T> operator*(T a, const Matrix &other) {
        Matrix res(other.row, other.col);
        for (int i = 0; i < other.row; i++) {
            for (int j = 0; j < other.col; j++) {
                res.mat[i][j] = other.mat[i][j] * a;
            }
        }
        return res;
    };

    //friend function must be write inside the class
    friend Matrix<T> operator*(vector<T> vec, Matrix &a) {
        if (vec.size() != a.mat.size()) {
            cout << "This vector cannot be multiplied with
a matrix.";
            exit(0);
        }
        Matrix<T> res(1, a.mat[0].size());
        for (int i = 0; i < a.mat[0].size(); i++) {
            res.mat[0][i] = 0;
            for (int j = 0; j < a.mat.size(); j++) {
                res.mat[0][i] += vec[j] * a.mat[j][i];
            }
        }
        return res;
    }

    vector<vector<T>> mat;
    int row{};
    int col{};
    Matrix(int row, int col);

    void swap(int i, int j, int row);

    int find(int i, int row);
};

template<class T>
Matrix<T> Matrix<T>::conjugation() {
    Matrix res(col, row);
    res = this->transposition();
    if(typeid(T)==typeid(complex<double>)){
        for(int j = 0;j<col;j++){
            for(int i = 0;i<row;i++){
                res.mat[j][i] = conj(res.mat[j][i]);
            }
        }
    }
    return res;

}

template<class T>
Matrix<T> Matrix<T>::transposition() {
    Matrix res(col, row);
    for (int i = 0; i < col; i++) {
        for (int j = 0; j < row; j++) {
            res.mat[i][j] = mat[j][i];
        }
    }
    return res;
}

template<class T>
Matrix<T>::Matrix(int row, int col) {
    this->row = row;
    this->col = col;
    vector<T> temp(col);
    this->mat.resize(row, temp);
}

template<class T>
void Matrix<T>::setValue(vector<vector<T>> valueMat) {
    if (valueMat.size() != mat.size() ||
valueMat[0].size() != mat[0].size()) {
        cout << "The input value's size does not match the
matrix!";
        exit(0);
```

```cpp
    }
    for (int i = 0; i < valueMat.size(); i++) {
        for (int j = 0; j < valueMat[0].size(); j++) {
            mat[i][j] = valueMat[i][j];
        }
    }
}

template<class T>
void Matrix<T>::showMatrix() {
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            cout << mat[i][j] << " ";
        }
        cout << endl;
    }
}

template<class T>
Matrix<T> Matrix<T>::elementWiseMultiple(Matrix<T> &a) {
    if (mat.size() != a.mat.size() || mat[0].size() !=
a.mat[0].size()) {
        cout << "Matrix size does not match! Element wise
multiplication operation failed.";
        exit(0);
    }
    Matrix<T> res(row, col);
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            res.mat[i][j] = mat[i][j] * a.mat[i][j];
        }
    }
    return res;
}

template<class T>
Matrix<T> Matrix<T>::operator+(Matrix<T> &a) {
    if (mat.size() != a.mat.size() || mat[0].size() !=
a.mat[0].size()) {
        cout << "Matrix size does not match! Plus
operation failed.";
        exit(0);
    }
    Matrix<T> res(row, col);
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            res.mat[i][j] = mat[i][j] + a.mat[i][j];
        }
    }
    return res;
}

template<class T>
Matrix<T> Matrix<T>::operator-(Matrix<T> &a) {
    if (mat.size() != a.mat.size() || mat[0].size() !=
a.mat[0].size()) {
        cout << "Matrix size does not match! Subtraction
operation failed.";
        exit(0);
    }
    Matrix<T> res(row, col);
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            res.mat[i][j] = mat[i][j] - a.mat[i][j];
        }
    }
    return res;
}

template<class T>
Matrix<T> Matrix<T>::operator*(Matrix<T> &a) {
    if (mat[0].size() != a.mat.size()) {
        cout << "Matrix size does not match! Two matrix
multiplication failed.";
        exit(0);
    }
    Matrix<T> res(row, a.col);
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < a.col; j++) {
            res.mat[i][j] = 0;
            for (int k = 0; k < col; k++) {
                res.mat[i][j] += mat[i][k] * a.mat[k][j];
            }
        }
    }
    return res;
}

template<class T>
Matrix<T> Matrix<T>::operator*(vector<T> vec) {
    if (mat[0].size() != 1) {
        cout << "This matrix cannot be multiplied with a
vector.";
        exit(0);
    }
    Matrix<T> res(row, vec.size());
```

```cpp
    for (int i = 0; i < mat.size(); i++) {
        for (int j = 0; j < vec.size(); j++) {
            res.mat[i][j] = mat[i][0] * vec[j];
        }
    }
    return res;
}

template<class T>
Matrix<T> Matrix<T>::operator*(T a) {
    Matrix res(row, col);
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            res.mat[i][j] = mat[i][j] * a;
        }
    }
    return res;
}


template<class T>
Matrix<T> Matrix<T>::operator/(T a) {

    if (a == complex<double>(0,0)) {
        cout << "Divisor can not be zero! Division
operation failed.";
        exit(0);
    }
    Matrix res(row, col);
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            res.mat[i][j] = mat[i][j] / a;
        }
    }
    return res;
}


template<class T>
T Matrix<T>::findMax() {
    if(typeid(T) == typeid(complex<double>)){
        cout << "complex can not compare";
        exit(0);
    }
    T max = INT32_MIN;

    for (int i = 0; i < mat.size(); i++) {
        for (int j = 0; j < mat[0].size(); j++) {
                if (mat[i][j] > max) {
                    max = mat[i][j];
                }
            }
        }
        return max;
    }

template<class T>
T Matrix<T>::findMaxByRow(int rowNum) {
    if(typeid(T) == typeid(complex<double>)){
        cout << "complex can not compare";
        exit(0);
    }
    if (rowNum > row) {
        cout << "Your input row number is too large";
        exit(0);
    }
    T max = INT32_MIN;
    for (int i = 0; i < mat[0].size(); i++) {
        if (mat[rowNum][i] > max) {
            max = mat[rowNum][i];
        }
    }
    return max;
}

template<class T>
T Matrix<T>::findMaxByCol(int colNum) {
    if(typeid(T) == typeid(complex<double>)){
        cout << "complex can not compare";
        exit(0);
    }
    if (colNum > col) {
        cout << "Your input col number is too large";
        exit(0);
    }
    T max = INT32_MIN;
    for (int i = 0; i < mat.size(); i++) {
        if (mat[i][colNum] > max) {
            max = mat[i][colNum];
        }
    }
    return max;
}

template<class T>
T Matrix<T>::findMin() {
    if(typeid(T) == typeid(complex<double>)){
        cout << "complex can not compare";
        exit(0);
    }
    T min = INT32_MAX;
    for (int i = 0; i < mat.size(); i++) {
        for (int j = 0; j < mat[0].size(); j++) {
            if (mat[i][j] < min) {
                min = mat[i][j];
            }
        }
    }
    return min;
}

template<class T>
T Matrix<T>::findMinByRow(int rowNum) {
    if(typeid(T) == typeid(complex<double>)){
        cout << "complex can not compare";
        exit(0);
    }
    if (rowNum > row) {
        cout << "Your input row number is too large";
        exit(0);
    }
    T min = INT32_MAX;
    for (int i = 0; i < mat[0].size(); i++) {
        if (mat[rowNum][i] < min) {
            min = mat[rowNum][i];
        }
    }
    return min;
}

template<class T>
T Matrix<T>::findMinByCol(int colNum) {
    if(typeid(T) == typeid(complex<double>)){
        cout << "complex can not compare";
        exit(0);
    }
    if (colNum > col) {
        cout << "Your input col number is too large";
        exit(0);
    }
    T min = INT32_MAX;
    for (int i = 0; i < mat.size(); i++) {
        if (mat[i][colNum] < min) {
            min = mat[i][colNum];
        }
    }
    return min;
}

template<class T>
T Matrix<T>::getSum() {
    T sum = 0;
    for (int i = 0; i < mat.size(); i++) {
        for (int j = 0; j < mat[0].size(); j++) {
            sum += mat[i][j];
        }
    }
    return sum;
}

template<class T>
T Matrix<T>::getSumByRow(int rowNum) {
    T sum = 0;
    for (int i = 0; i < mat[0].size(); i++) {
        sum += mat[rowNum][i];
    }
    return sum;
}


template<class T>
T Matrix<T>::getSumByCol(int colNum) {
    T sum = 0;
    for (int i = 0; i < mat.size(); i++) {
        sum += mat[i][colNum];
    }
    return sum;
}

template<class T>
T Matrix<T>::getAverage() {
    return Matrix::getSum() / complex<double>((row *
col),0);
}

template<class T>
T Matrix<T>::getAverageByRow(int rowNum) {
    return Matrix::getSumByRow(rowNum) / complex<double>
((row * col),0);
}

template<class T>
```

```
T Matrix<T>::getAverageByCol(int colNum) {
    return Matrix::getSumByCol(colNum) / complex<double>
((row * col),0);
}


template<class T>
T Matrix<T>::dotProduct(Matrix &other) {
    if (!this->isVector() || !other.isVector()) {
        cout << "The dot product requires two vectors! Dot
product operation failed.";
        exit(0);
    }
    double res=0;
    for (int j = 0; j < col; j++) {
        res+=mat[0][j]+other.mat[0][j];
    }
    return res;
}

template<class T>
T Matrix<T>::crossProduct(Matrix &other) {
    if (!this->isVector() || !other.isVector()) {
        cout << "The cross product requires two vectors!
Cross product operation failed.";
        exit(0);
    }

    return nullptr;
}

template<class T>
bool Matrix<T>::isVector() const {
    if (col == 1)return true;
    else return false;
}

template<class T>
T det(Matrix<T> &a) {
    if (a.mat.size() == 1) {
        return a.mat.front().front();
    }
    uint32_t size_m = a.mat.size();
    Matrix<T> submatrix(a.row, a.col);
    submatrix.mat = vector<vector<T>>(size_m - 1,
vector<T>(size_m - 1, static_cast<T>(0)));
    T will_return(0);
    for (uint32_t i = 0; i < size_m; ++i) {
        for (uint32_t j = 0; j < size_m - 1; ++j) {
            for (uint32_t k = 0; k < size_m - 1; ++k) {
                submatrix.mat[j][k] = a.mat[(((i > j) ? 0
: 1) + j)][k + 1];
            }
        }
        will_return += ((i % 2) ? -1 : 1) *
a.mat[i].front() * det(submatrix);
    }
    return will_return;
}




template<class T>
int Matrix<T>::find( int i, int row) {
    for(int m=i+1;m<row;m++)
        if(mat[m][i]==1.0)
            return m;
    return 0;
}

template<class T>
void Matrix<T>::swap(int i, int j, int row) {
    for(int m=0;m<row;m++){
        mat[i][m]+=mat[j][m];
        mat[j][m]=mat[i][m]-mat[j][m];
        mat[i][m]-=mat[j][m];
    }
}

template<class T>
Matrix<T> Matrix<T>::convolution(Matrix &kernel){
    T temp;
    Matrix<T> result(row, col);
    int kLength = kernel.col;
    int kWidth = kernel.row;

    for (int i = 0; i < col; i++) {
        for (int j = 0; j < row; j++) {
            temp = 0;
            for (int m = 0; m < kLength; m++) {
                for (int n = 0; n < kWidth; n++) {
                    if ((i - m) >= 0 && (i - m) < col &&
(j - n) >= 0 && (j - n) < row) {
                        temp += kernel.mat[m][n] * this-
>mat[i - m][j - n];
```

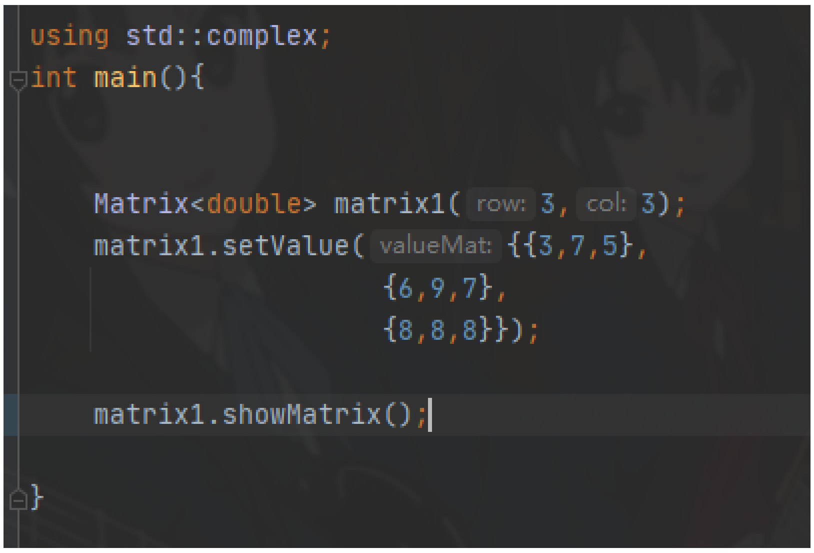
```
                    }
                }
            }
            result.mat[i][j] = temp;
        }
    }
    return result;
}

template<class T>
Matrix<T> inverse(Matrix<T> &a){
        //可逆条件
        if (a.row==a.col && a.row > 1 && det(a) != 0) {
            static constexpr T zeroNumber{0};
            static constexpr T oneElement{1};
            std::vector<std::vector<T>> result_vector(
a.row, std::vector<T>( a.col, zeroNumber));
            for (int32_t i = 0; i < a.row; i++) {
                for (int32_t j = 0; j < a.col; j++) {
                    //vector<vector<T>> submatrix(this-
>rows() - 1, vector<T>(this->cols() - 1, static_cast<T>
(0)));
                    Matrix<T> submatrix(a.row,a.col);
                    submatrix.setValue(vector<vector<T>>
(a.mat));

 submatrix.mat.erase(submatrix.mat.begin() + i);
                    for (int m = 0; m <  a.row - 1; ++m) {

 submatrix.mat[m].erase(submatrix.mat[m].begin() + j);
                    }
                    result_vector[j][i] =
                            (((i + j) % 2) ? -1 : 1) *
det(submatrix);//子矩阵展开得到伴随矩阵
                }
            }
            Matrix<T> will_return(a.row,a.col);
            will_return.setValue(result_vector);
            will_return = will_return / det(a);
            //will_return[i][j] = res[i][j] / this-
>determinant();//逆
            return will_return;
        }
        return Matrix<T>(1, 1);
    }
```

```
#endif
```

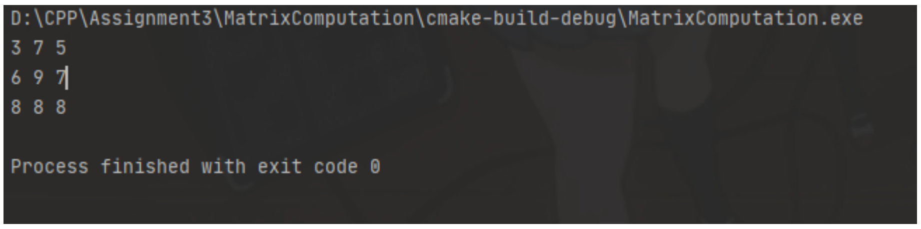
Part 3 - Result & Verification

1) It supports all matrix sizes, from small fixed-size matrices to arbitrarily large dense matrices, and even sparse matrices (Add: try to use efficient ways to store the sparse matrices). (10 points)

small fixed-size matrix

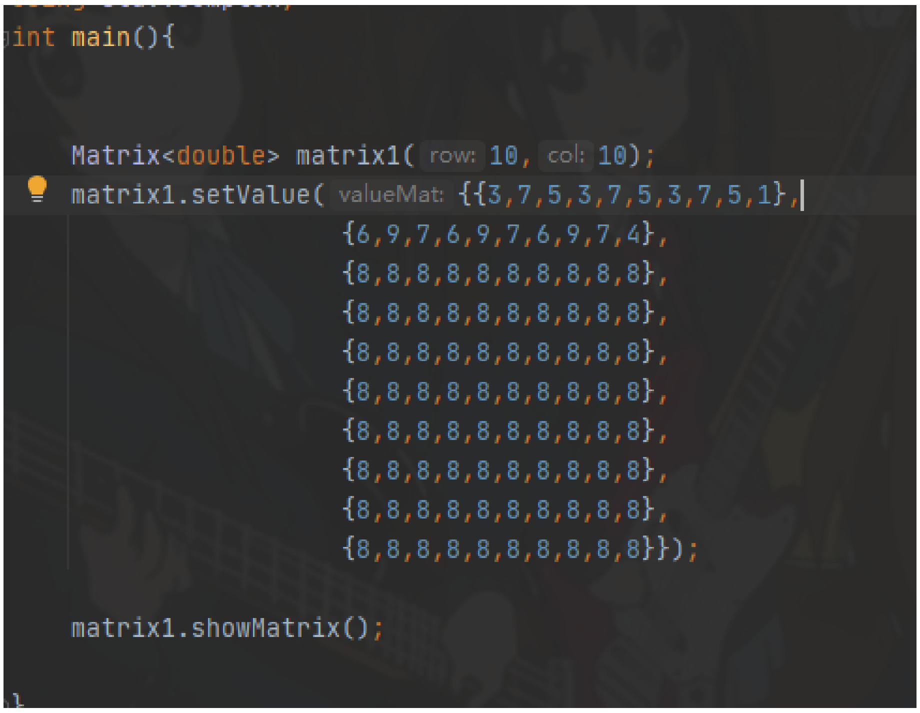


```
using std::complex;
int main(){


    Matrix<double> matrix1(row: 3, col: 3);
    matrix1.setValue(valueMat: {{3,7,5},
                                {6,9,7},
                                {8,8,8}});

    matrix1.showMatrix();

}
```

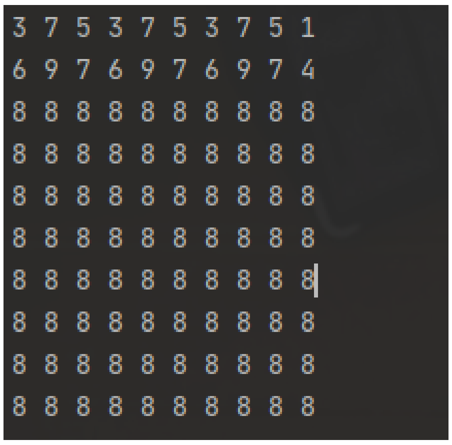


```
D:\CPP\Assignment3\MatrixComputation\cmake-build-debug\MatrixComputation.exe
3 7 5
6 9 7
8 8 8

Process finished with exit code 0
```

arbitrarily large dense matrices



```
int main(){

    Matrix<double> matrix1( row: 10, col: 10);
    matrix1.setValue( valueMat: {{3,7,5,3,7,5,3,7,5,1},
                        {6,9,7,6,9,7,6,9,7,4},
                        {8,8,8,8,8,8,8,8,8,8},
                        {8,8,8,8,8,8,8,8,8,8},
                        {8,8,8,8,8,8,8,8,8,8},
                        {8,8,8,8,8,8,8,8,8,8},
                        {8,8,8,8,8,8,8,8,8,8},
                        {8,8,8,8,8,8,8,8,8,8},
                        {8,8,8,8,8,8,8,8,8,8},
                        {8,8,8,8,8,8,8,8,8,8}});

    matrix1.showMatrix();
}
```

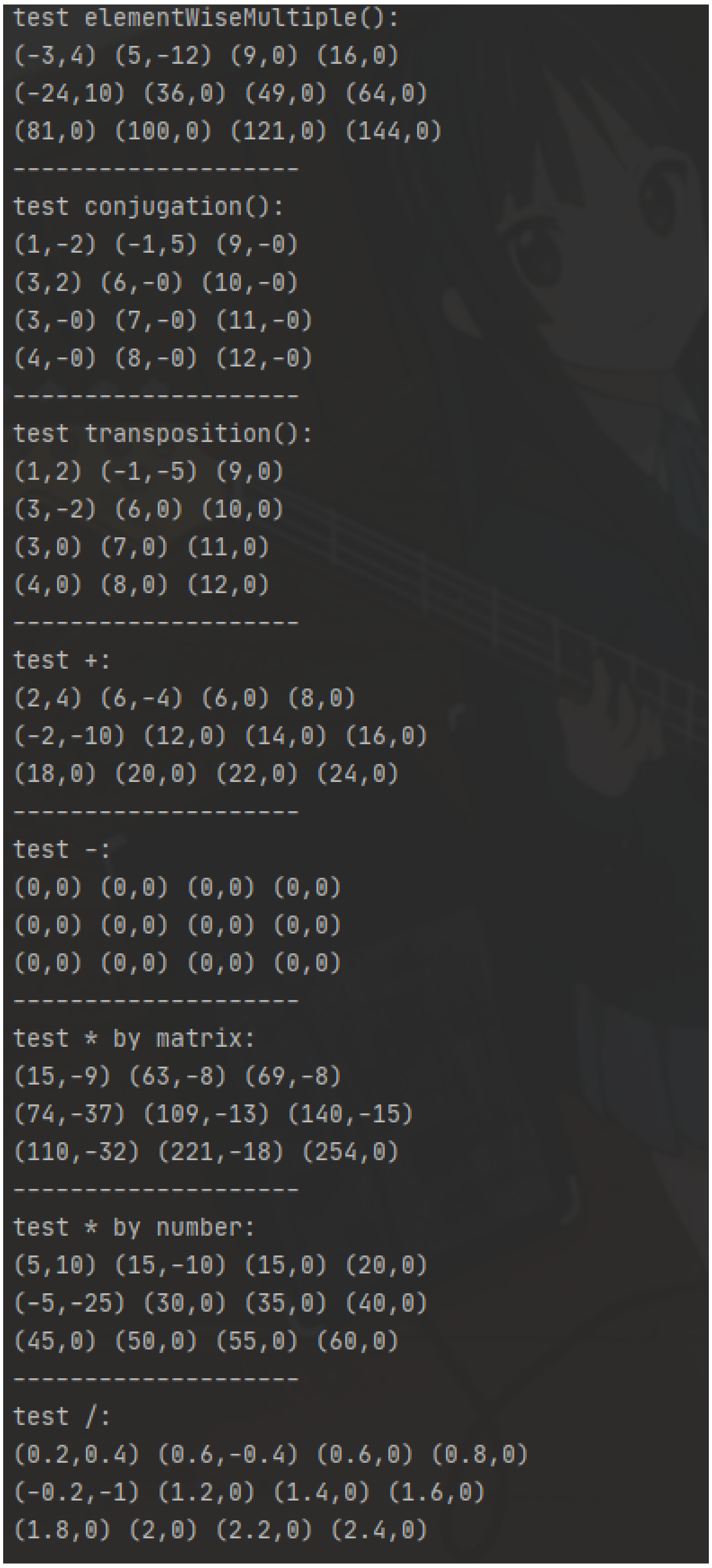


```
3 7 5 3 7 5 3 7 5 1
6 9 7 6 9 7 6 9 7 4
8 8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8 8
```

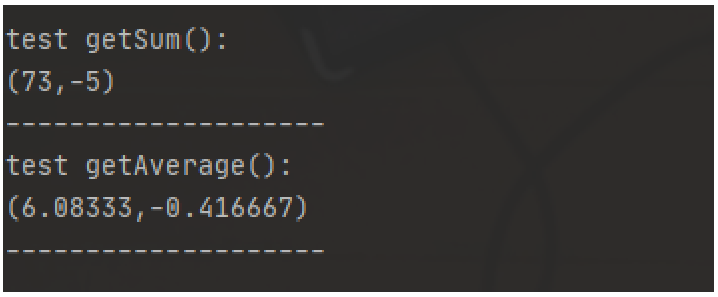
2) It supports all standard numeric types, including std::complex, integers, and is easily extensible to custom numeric types. (10 points)

std::complex: testComplex.cpp

```
//
// Created by ccyys on 2021/5/22.
//

#include <complex>
#include "../Matrix.hpp"

using std::complex;
int main(){
    //test elementWiseMultiple()
    cout << "test elementWiseMultiple(): " << endl;
    Matrix<complex<double>> matrix1(3,4);
    matrix1.setValue({{complex<double>
(1,2),complex<double>(3,-2),3,4},
                      {complex<double>(-1,-5),6,7,8},
                      {9,10,11,12}});

    Matrix<complex<double>> matrix2(3,4);
    matrix2.setValue({{complex<double>
(1,2),complex<double>(3,-2),3,4},
                      {complex<double>(-1,-5),6,7,8},
                      {9,10,11,12}});
    matrix1.elementWiseMultiple(matrix2).showMatrix();
    cout << "---------------------"<< endl;

    //test conjugation()
    cout << "test conjugation(): " << endl;
    matrix1.conjugation().showMatrix();
    cout << "---------------------"<< endl;

    //test transposition()
    cout << "test transposition(): " << endl;
    matrix1.transposition().showMatrix();
    cout << "---------------------"<< endl;

    //test +
    cout << "test +: " << endl;
    Matrix<complex<double>> matrix3(3,4);
    (matrix1+matrix2).showMatrix();
    cout << "---------------------"<< endl;

    //test -
    cout << "test -: " << endl;
    (matrix1-matrix2).showMatrix();
    cout << "---------------------"<< endl;

    //test * by matrix
    cout << "test * by matrix: " << endl;
    Matrix<complex<double>> matrix4(4,3);
    matrix4.setValue({{complex<double>
(1,2),complex<double>(3,-2),3},
                      {complex<double>(-1,-5),6,7},
                      {9,10,11},
                      {1,2,3}});
    (matrix1*matrix4).showMatrix();
    cout << "---------------------"<< endl;
    //test * by number
    cout << "test * by number: " << endl;
    (matrix1*5).showMatrix();
    cout << "--------------------"<< endl;


    //test /
    cout << "test /: " << endl;
    (matrix1/5).showMatrix();
    cout << "--------------------"<< endl;

    //test getSum()
    cout << "test getSum(): " << endl;
    cout << matrix1.getSum() << endl;
    cout << "--------------------"<< endl;


    //test getAverage()
    cout << "test getAverage(): " << endl;
    cout << matrix1.getAverage() << endl;
    cout << "--------------------"<< endl;



}
```

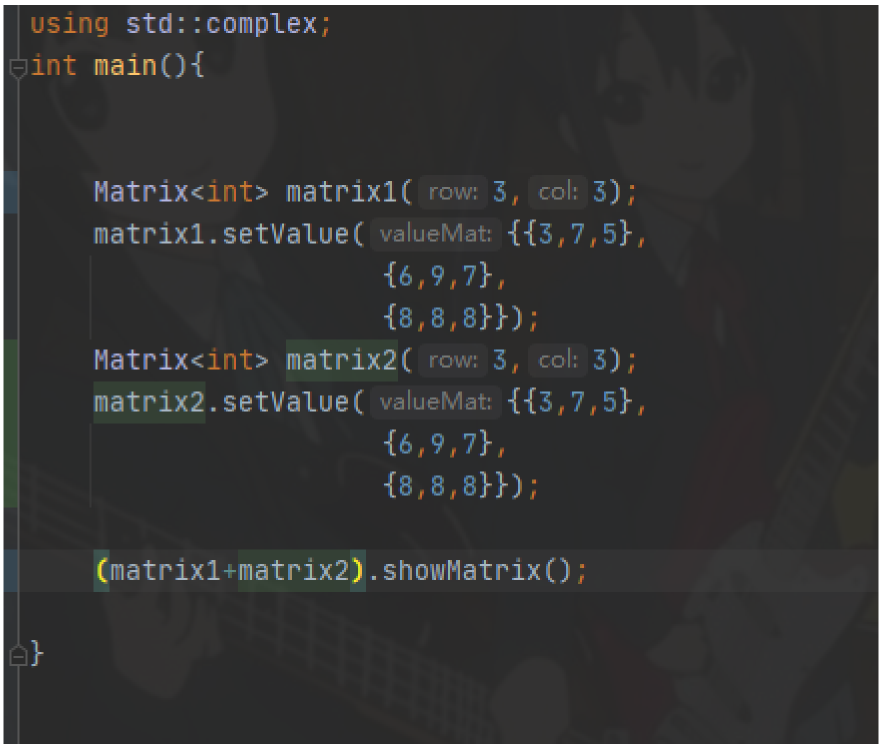


```
test elementWiseMultiple():
(-3,4) (5,-12) (9,0) (16,0)
(-24,10) (36,0) (49,0) (64,0)
(81,0) (100,0) (121,0) (144,0)
--------------------
test conjugation():
(1,-2) (-1,5) (9,-0)
(3,2) (6,-0) (10,-0)
(3,-0) (7,-0) (11,-0)
(4,-0) (8,-0) (12,-0)
--------------------
test transposition():
(1,2) (-1,-5) (9,0)
(3,-2) (6,0) (10,0)
(3,0) (7,0) (11,0)
(4,0) (8,0) (12,0)
--------------------
test +:
(2,4) (6,-4) (6,0) (8,0)
(-2,-10) (12,0) (14,0) (16,0)
(18,0) (20,0) (22,0) (24,0)
--------------------
test -:
(0,0) (0,0) (0,0) (0,0)
(0,0) (0,0) (0,0) (0,0)
(0,0) (0,0) (0,0) (0,0)
--------------------
test * by matrix:
(15,-9) (63,-8) (69,-8)
(74,-37) (109,-13) (140,-15)
(110,-32) (221,-18) (254,0)
--------------------
test * by number:
(5,10) (15,-10) (15,0) (20,0)
(-5,-25) (30,0) (35,0) (40,0)
(45,0) (50,0) (55,0) (60,0)
--------------------
test /:
(0.2,0.4) (0.6,-0.4) (0.6,0) (0.8,0)
(-0.2,-1) (1.2,0) (1.4,0) (1.6,0)
(1.8,0) (2,0) (2.2,0) (2.4,0)
```

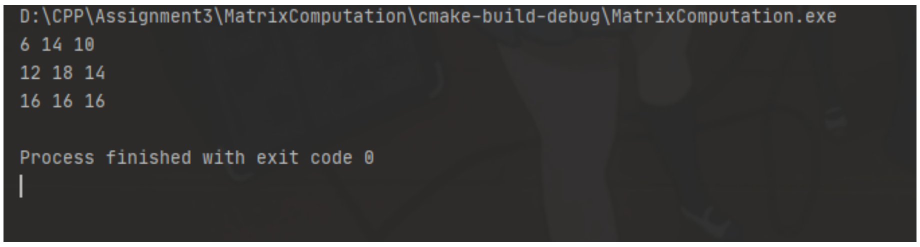


```
test getSum():
(73,-5)
---------------------
test getAverage():
(6.08333,-0.416667)
---------------------
```

integer:

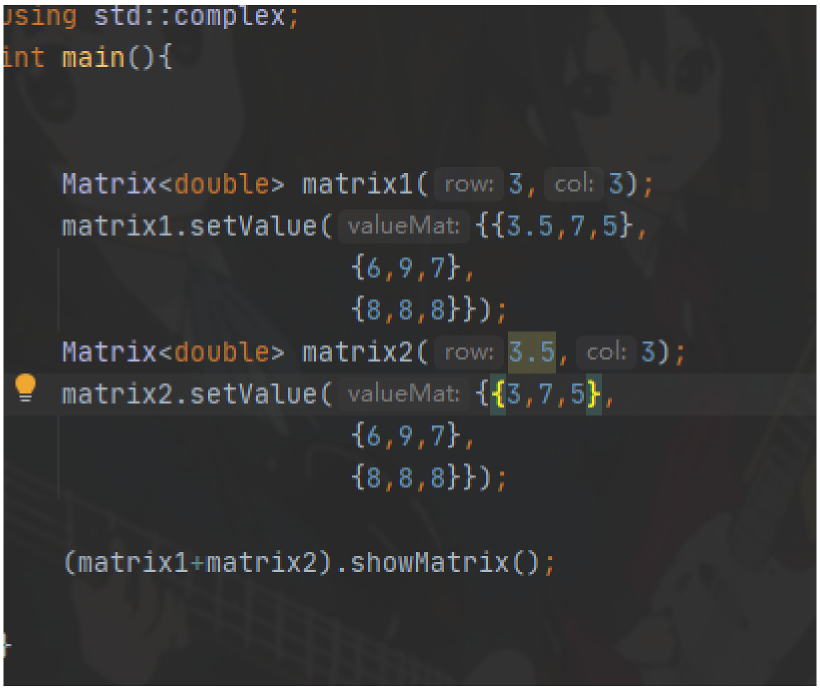


```
using std::complex;
int main(){


    Matrix<int> matrix1( row: 3, col: 3);
    matrix1.setValue( valueMat: {{3,7,5},
                                  {6,9,7},
                                  {8,8,8}});
    Matrix<int> matrix2( row: 3, col: 3);
    matrix2.setValue( valueMat: {{3,7,5},
                                  {6,9,7},
                                  {8,8,8}});

    (matrix1+matrix2).showMatrix();

}
```

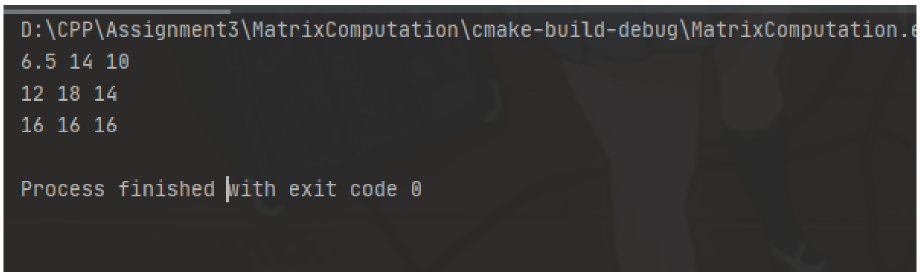


```
D:\CPP\Assignment3\MatrixComputation\cmake-build-debug\MatrixComputation.exe
6 14 10
12 18 14
16 16 16

Process finished with exit code 0
|
```

double



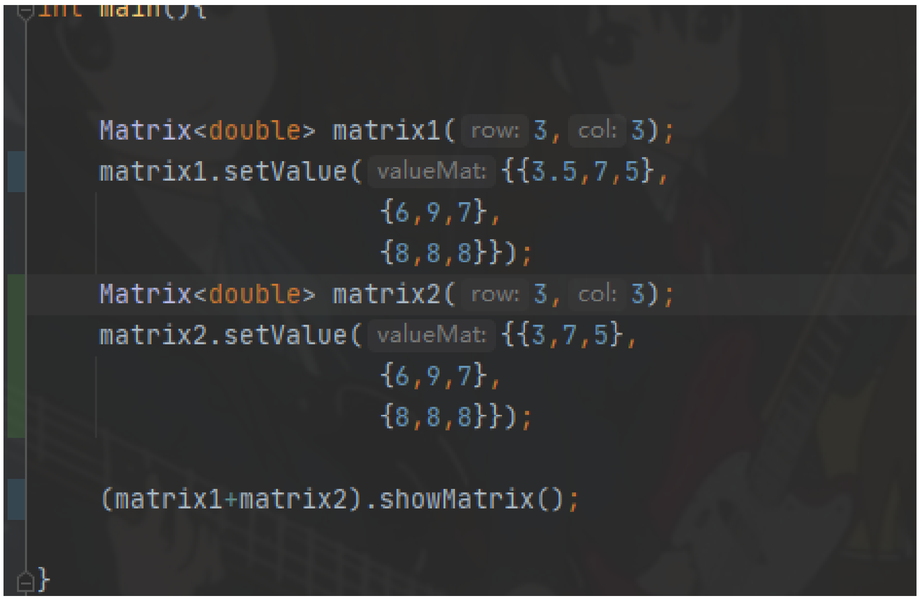
```
using std::complex;
int main(){

    Matrix<double> matrix1( row: 3, col: 3);
    matrix1.setValue( valueMat: {{3.5,7,5},
                                 {6,9,7},
                                 {8,8,8}});
    Matrix<double> matrix2( row: 3.5, col: 3);
    matrix2.setValue( valueMat: {{3,7,5},
                                 {6,9,7},
                                 {8,8,8}});

    (matrix1+matrix2).showMatrix();

}
```



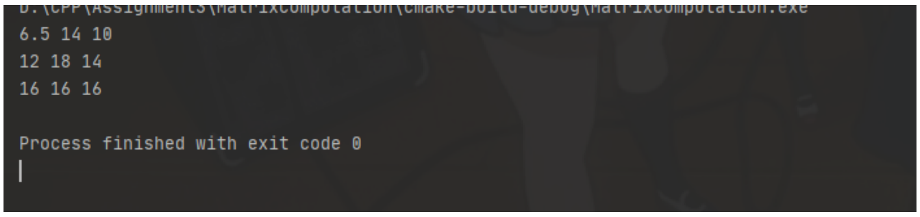
```
D:\CPP\Assignment3\MatrixComputation\cmake-build-debug\MatrixComputation.e
6.5 14 10
12 18 14
16 16 16

Process finished with exit code 0
```

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. (20 points)

addition

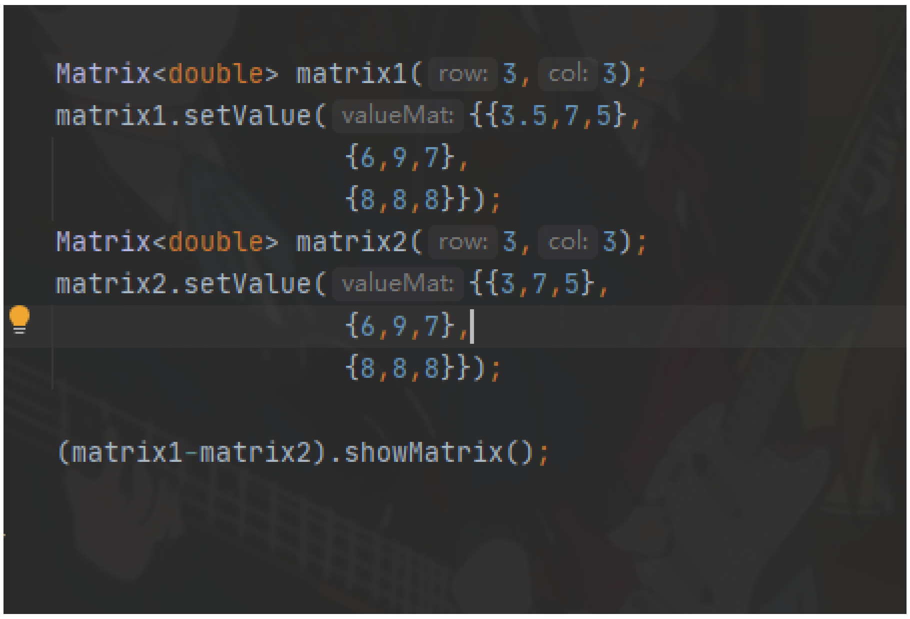


```
int main(){

    Matrix<double> matrix1( row: 3, col: 3);
    matrix1.setValue( valueMat: {{3.5,7,5},
                                 {6,9,7},
                                 {8,8,8}});
    Matrix<double> matrix2( row: 3, col: 3);
    matrix2.setValue( valueMat: {{3,7,5},
                                 {6,9,7},
                                 {8,8,8}});

    (matrix1+matrix2).showMatrix();

}
```

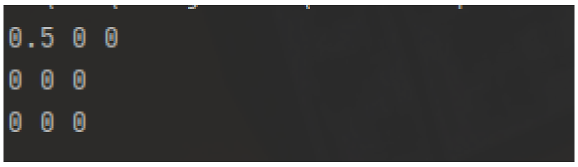


```
D:\CPP\Assignments3\MatrixComputation\cmake-build-debug\MatrixComputation.exe
6.5 14 10
12 18 14
16 16 16

Process finished with exit code 0
```

subtraction

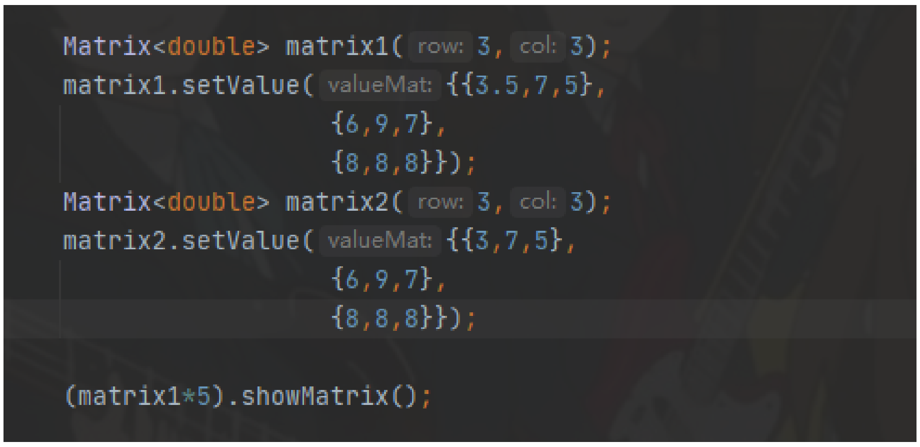


```
Matrix<double> matrix1( row: 3, col: 3);
matrix1.setValue( valueMat: {{3.5,7,5},
                  {6,9,7},
                  {8,8,8}});
Matrix<double> matrix2( row: 3, col: 3);
matrix2.setValue( valueMat: {{3,7,5},
                  {6,9,7},
                  {8,8,8}});

(matrix1-matrix2).showMatrix();
```

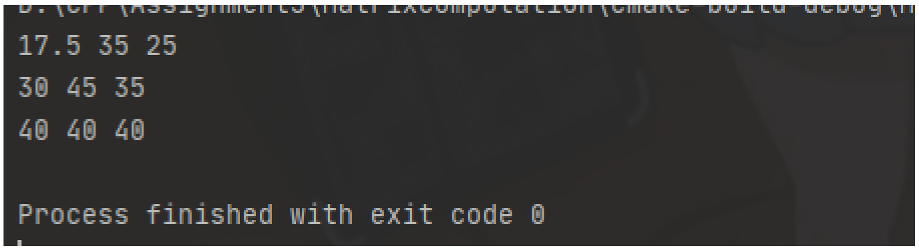


```
0.5 0 0
0 0 0
0 0 0
```

scalar multiplication

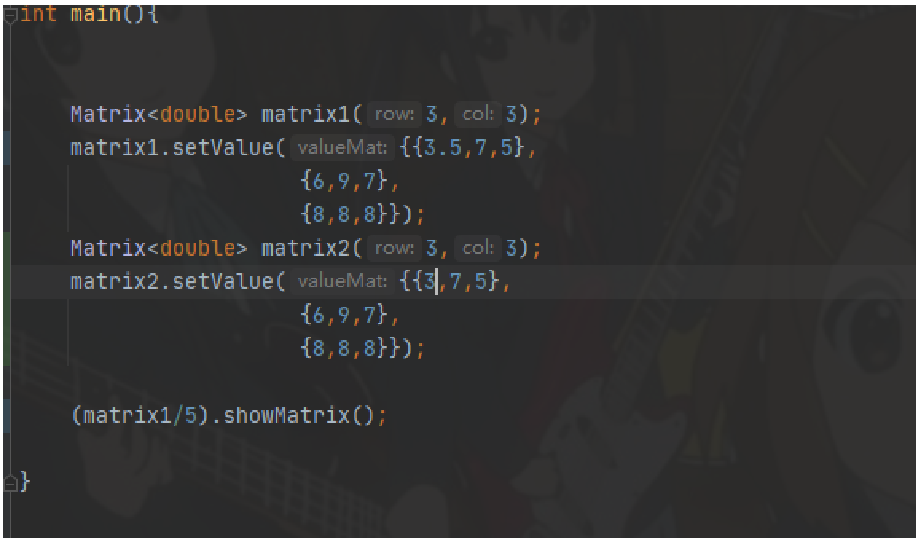


```
Matrix<double> matrix1( row: 3, col: 3);
matrix1.setValue( valueMat: {{3.5,7,5},
                  {6,9,7},
                  {8,8,8}});
Matrix<double> matrix2( row: 3, col: 3);
matrix2.setValue( valueMat: {{3,7,5},
                  {6,9,7},
                  {8,8,8}});

(matrix1*5).showMatrix();
```

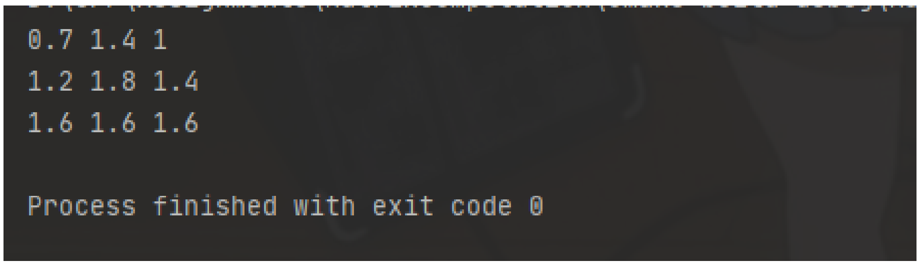


```
17.5 35 25
30 45 35
40 40 40

Process finished with exit code 0
```

scalar division

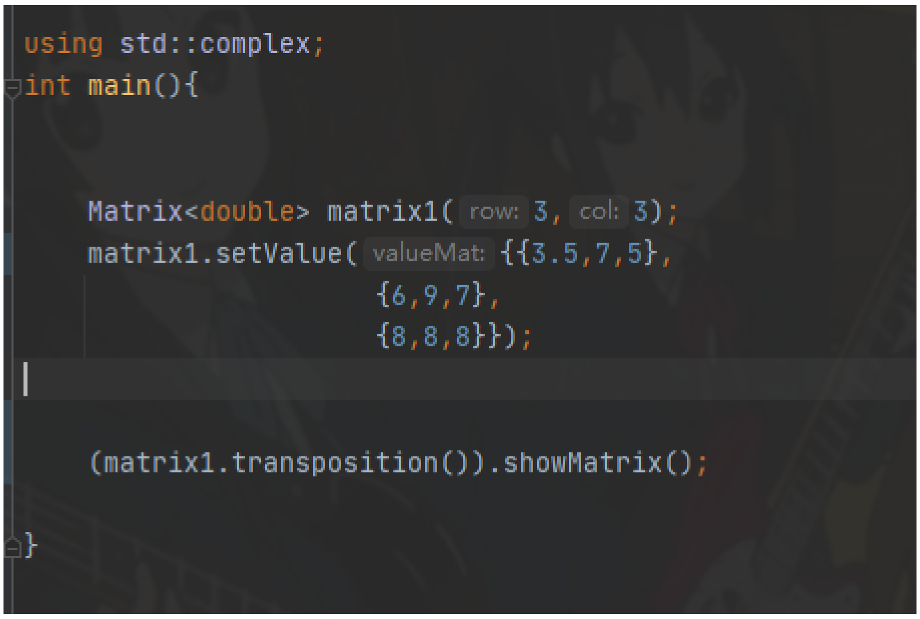


```
int main(){


    Matrix<double> matrix1( row: 3, col: 3);
    matrix1.setValue( valueMat: {{3.5,7,5},
                                 {6,9,7},
                                 {8,8,8}});
    Matrix<double> matrix2( row: 3, col: 3);
    matrix2.setValue( valueMat: {{3,7,5},
                                 {6,9,7},
                                 {8,8,8}});

    (matrix1/5).showMatrix();

}
```

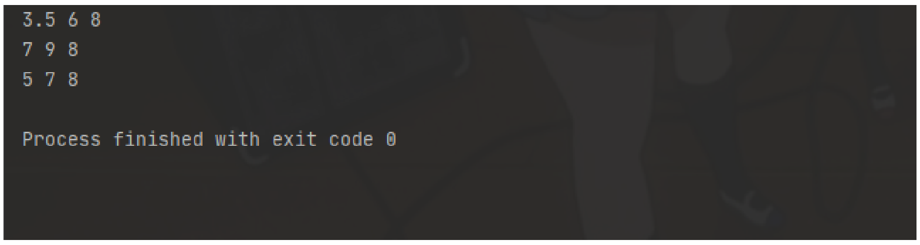


```
0.7 1.4 1
1.2 1.8 1.4
1.6 1.6 1.6

Process finished with exit code 0
```

transposition

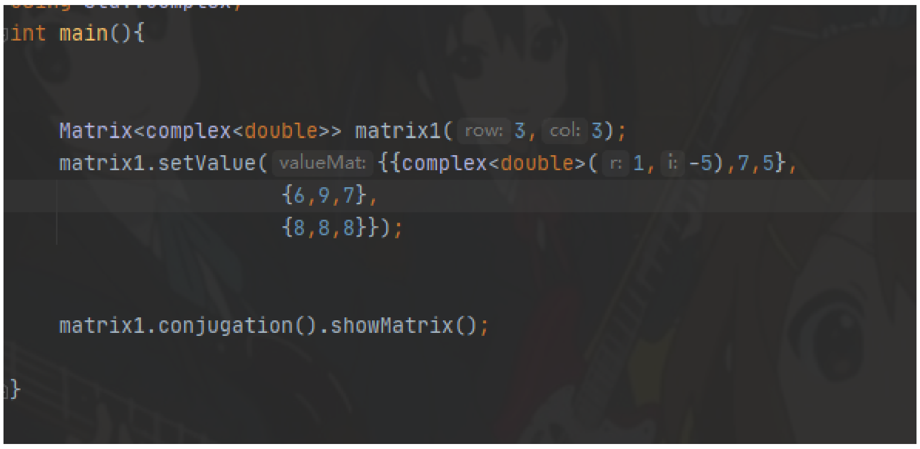


```
using std::complex;
int main(){


    Matrix<double> matrix1( row: 3, col: 3);
    matrix1.setValue( valueMat: {{3.5,7,5},
                                 {6,9,7},
                                 {8,8,8}});


    (matrix1.transposition()).showMatrix();

}
```

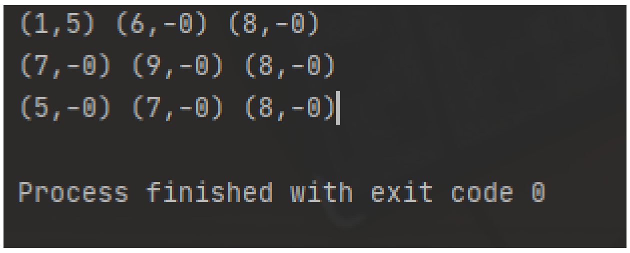


```
3.5 6 8
7 9 8
5 7 8

Process finished with exit code 0
```

conjugation

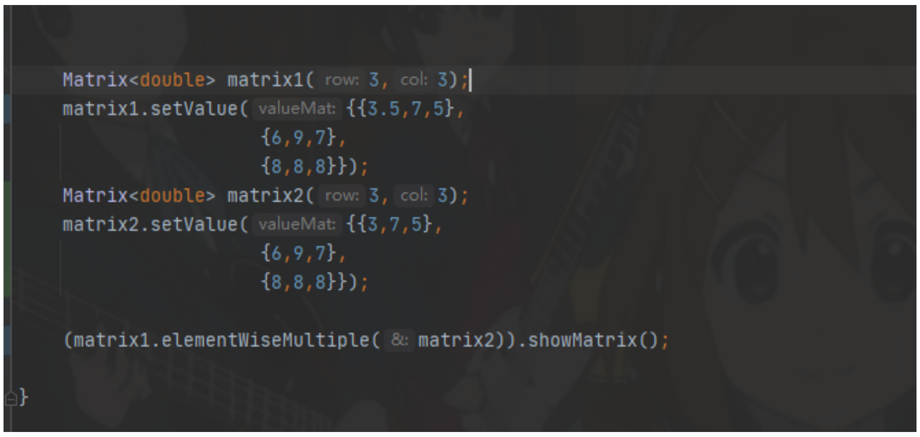


```
using std::complex;
int main(){

    Matrix<complex<double>> matrix1( row: 3, col: 3);
    matrix1.setValue( valueMat: {{complex<double>( r: 1, i: -5),7,5},
                        {6,9,7},
                        {8,8,8}});


    matrix1.conjugation().showMatrix();

}
```

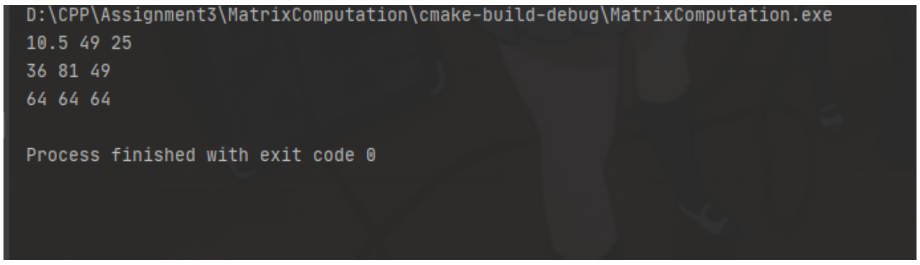


```
(1,5) (6,-0) (8,-0)
(7,-0) (9,-0) (8,-0)
(5,-0) (7,-0) (8,-0)

Process finished with exit code 0
```

element-wise multiplication

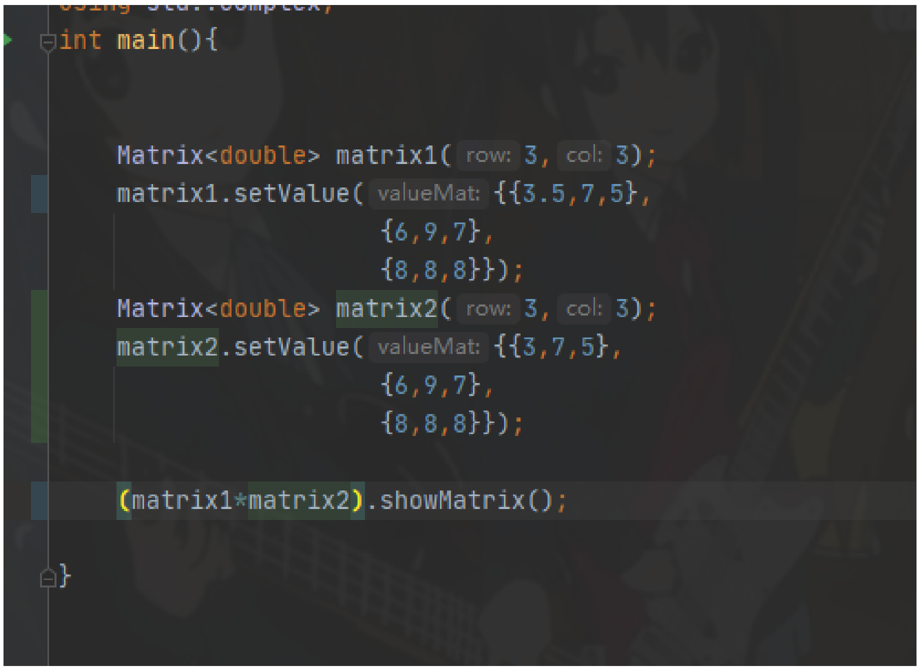


```
    Matrix<double> matrix1( row: 3, col: 3);
    matrix1.setValue( valueMat: {{3.5,7,5},
                      {6,9,7},
                      {8,8,8}});
    Matrix<double> matrix2( row: 3, col: 3);
    matrix2.setValue( valueMat: {{3,7,5},
                      {6,9,7},
                      {8,8,8}});

    (matrix1.elementWiseMultiple( &: matrix2)).showMatrix();

}
```

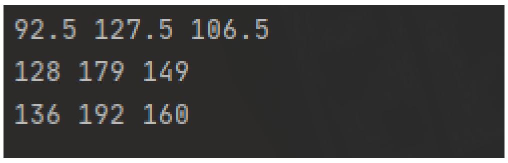


```
D:\CPP\Assignment3\MatrixComputation\cmake-build-debug\MatrixComputation.exe
10.5 49 25
36 81 49
64 64 64

Process finished with exit code 0
```

matrix-matrix multiplication

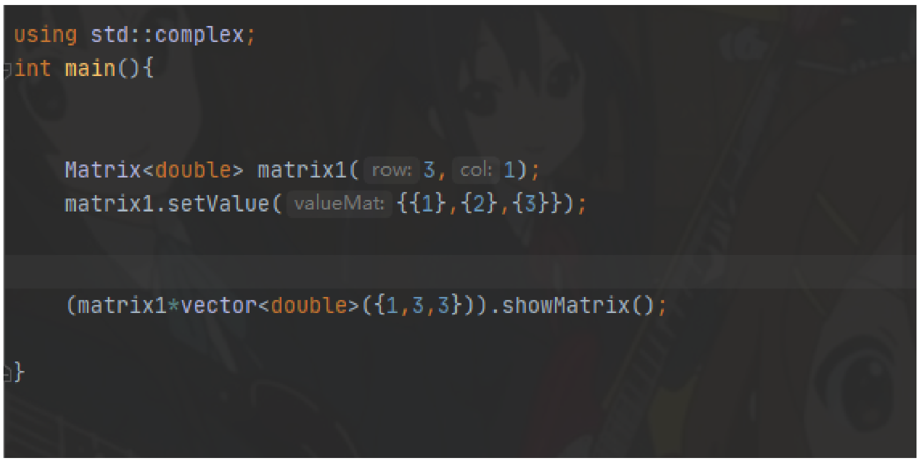


```
int main(){


    Matrix<double> matrix1( row: 3, col: 3);
    matrix1.setValue( valueMat: {{3.5,7,5},
                                 {6,9,7},
                                 {8,8,8}});
    Matrix<double> matrix2( row: 3, col: 3);
    matrix2.setValue( valueMat: {{3,7,5},
                                 {6,9,7},
                                 {8,8,8}});

    (matrix1*matrix2).showMatrix();

}
```

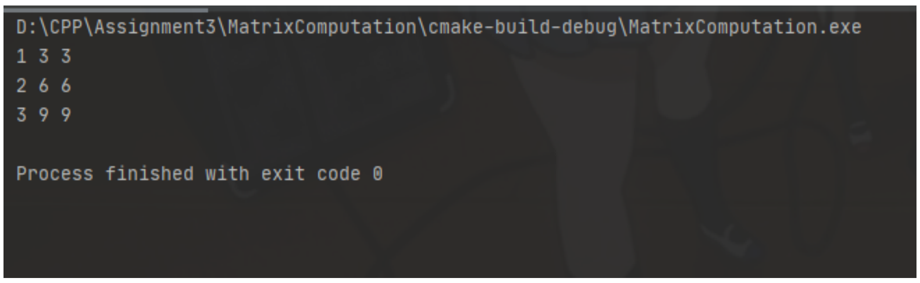


```
92.5 127.5 106.5
128 179 149
136 192 160
```

matrix-vector multiplication

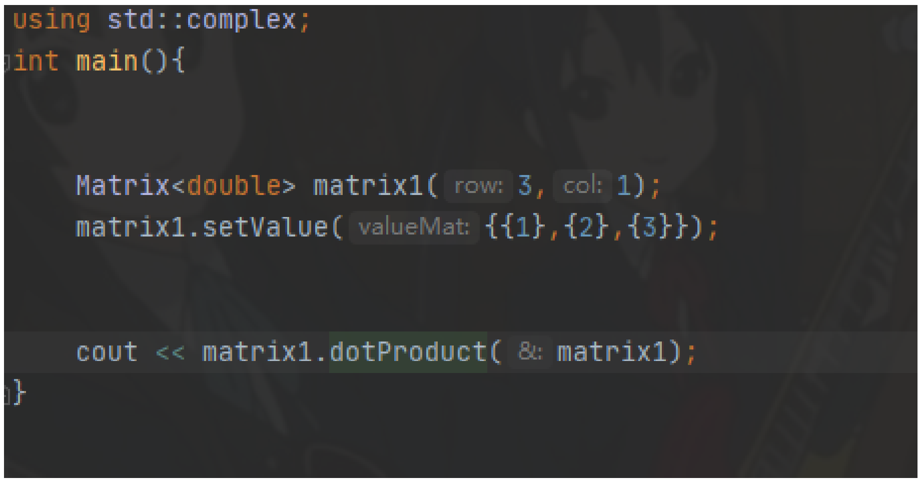


```
using std::complex;
int main(){


    Matrix<double> matrix1( row: 3, col: 1);
    matrix1.setValue( valueMat: {{1},{2},{3}});


    (matrix1*vector<double>({1,3,3})).showMatrix();

}
```

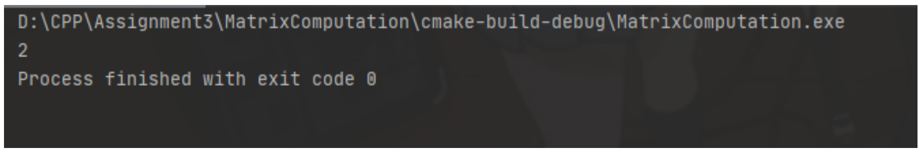


```
D:\CPP\Assignment3\MatrixComputation\cmake-build-debug\MatrixComputation.exe
1 3 3
2 6 6
3 9 9

Process finished with exit code 0
```

dot production



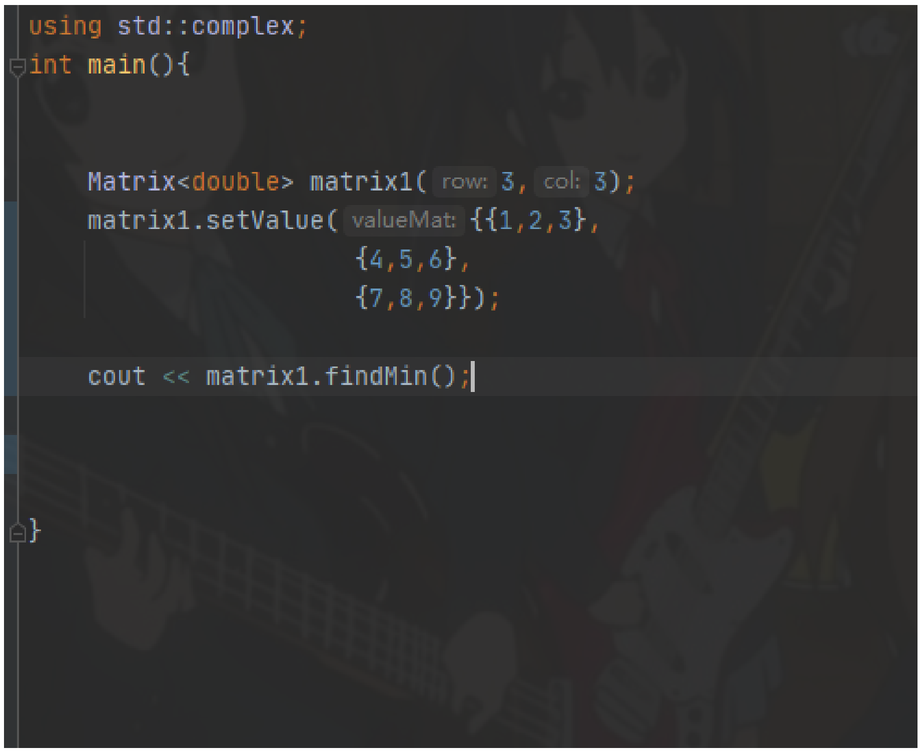
```
using std::complex;
int main(){

    Matrix<double> matrix1( row: 3, col: 1);
    matrix1.setValue( valueMat: {{1},{2},{3}});

    cout << matrix1.dotProduct( &: matrix1);
}
```



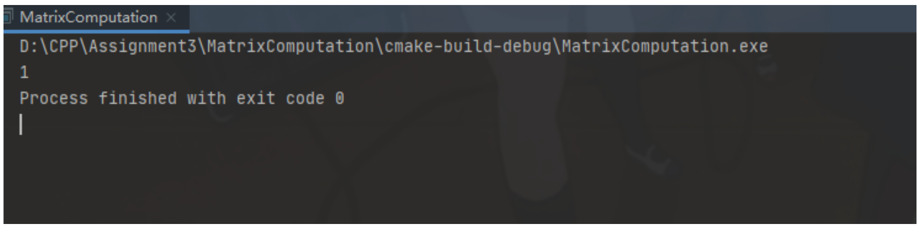
```
D:\CPP\Assignment3\MatrixComputation\cmake-build-debug\MatrixComputation.exe
2
Process finished with exit code 0
```

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). (10 points)

find min

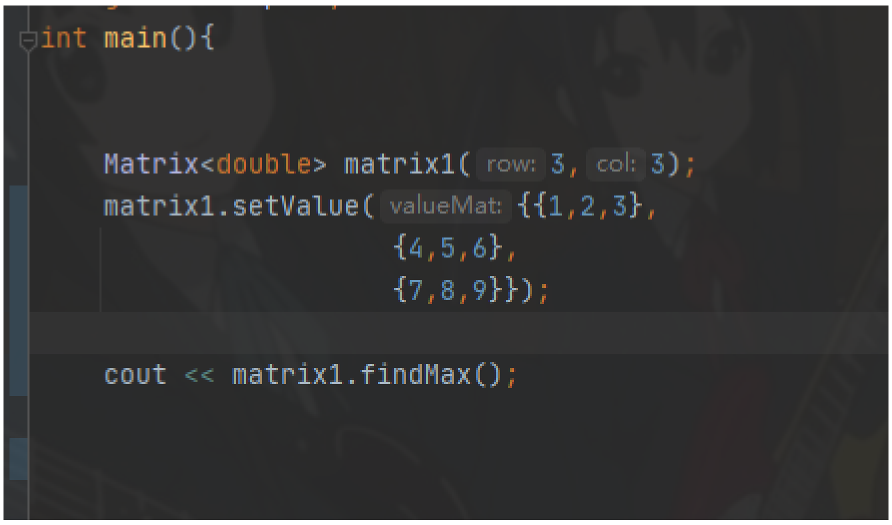


```
using std::complex;
int main(){

    Matrix<double> matrix1( row: 3, col: 3);
    matrix1.setValue( valueMat: {{1,2,3},
                      {4,5,6},
                      {7,8,9}});

    cout << matrix1.findMin();


}
```

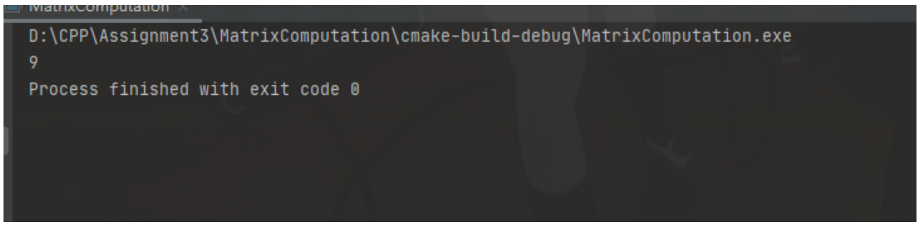

MatrixComputation

```
D:\CPP\Assignment3\MatrixComputation\cmake-build-debug\MatrixComputation.exe
1
Process finished with exit code 0
```

find max

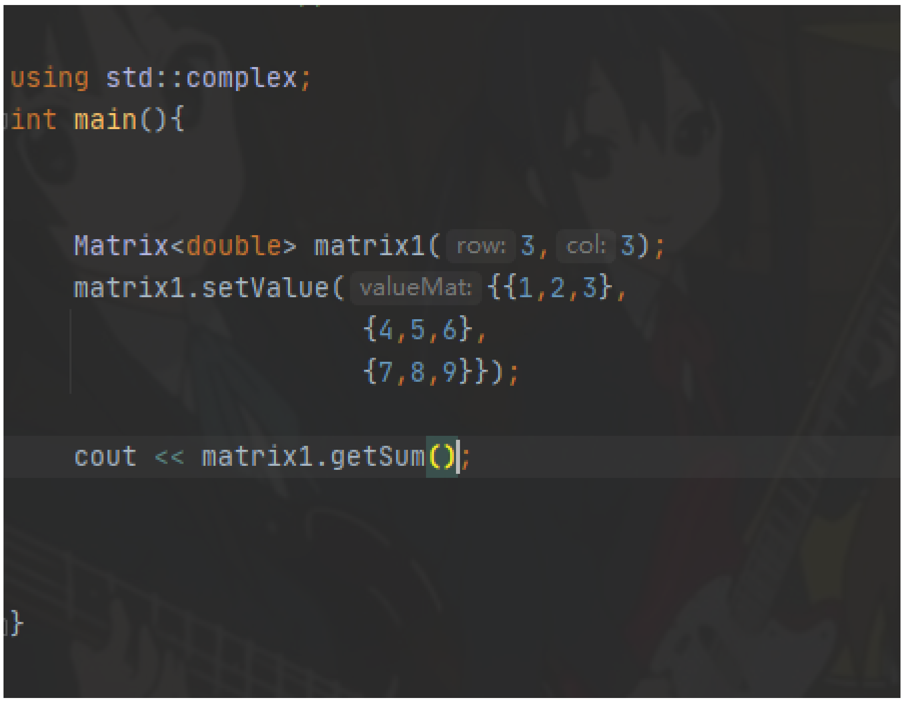


```
int main(){

    Matrix<double> matrix1( row: 3, col: 3);
    matrix1.setValue( valueMat: {{1,2,3},
                                 {4,5,6},
                                 {7,8,9}});

    cout << matrix1.findMax();
```


MatrixComputation

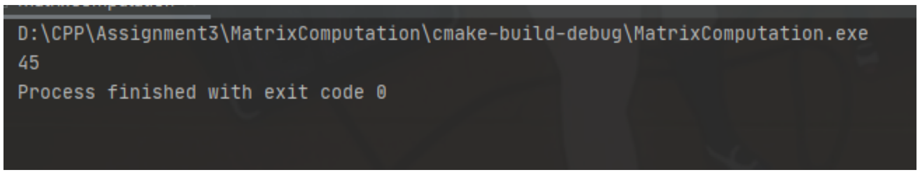
```
D:\CPP\Assignment3\MatrixComputation\cmake-build-debug\MatrixComputation.exe
9
Process finished with exit code 0
```

sum

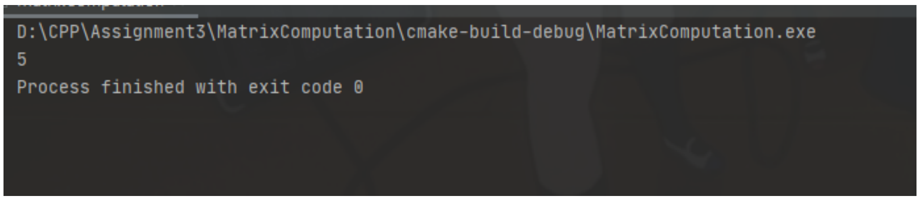


```
using std::complex;
int main(){

    Matrix<double> matrix1( row: 3, col: 3);
    matrix1.setValue( valueMat: {{1,2,3},
                                 {4,5,6},
                                 {7,8,9}});

    cout << matrix1.getSum();



}
```

average

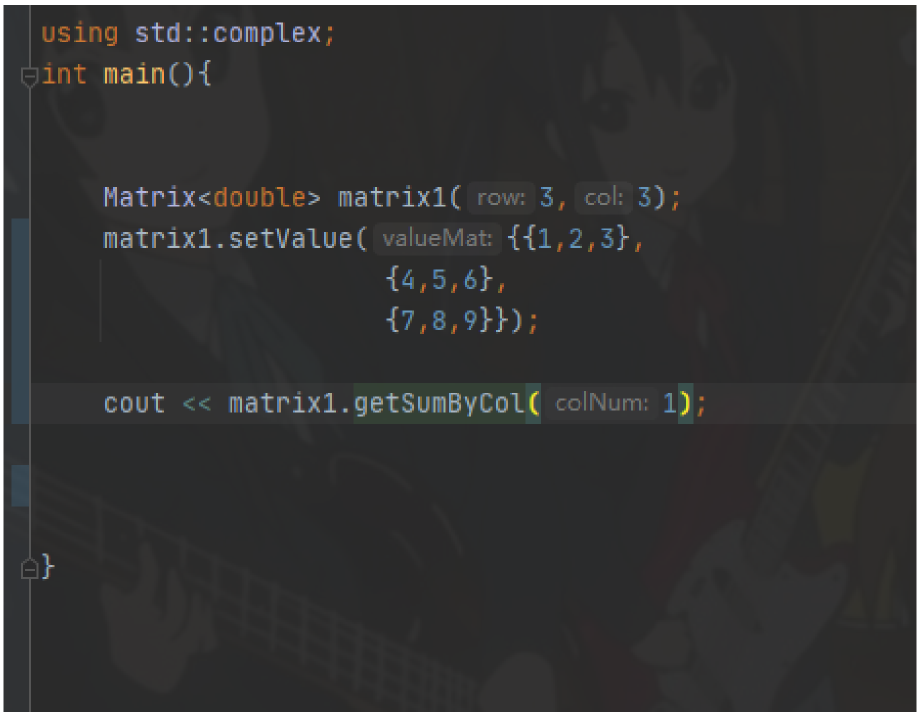


```
D:\CPP\Assignment3\MatrixComputation\cmake-build-debug\MatrixComputation.exe
45
Process finished with exit code 0
```

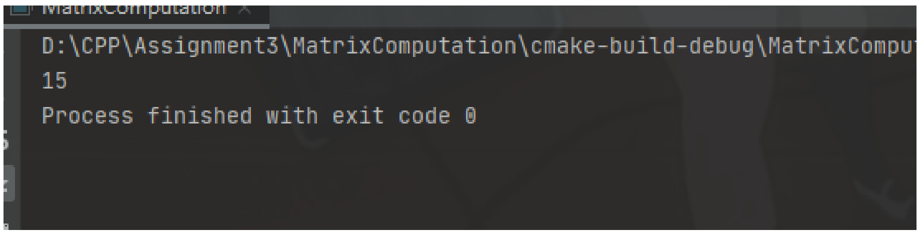


```
D:\CPP\Assignment3\MatrixComputation\cmake-build-debug\MatrixComputation.exe
5
Process finished with exit code 0
```

axis-specific

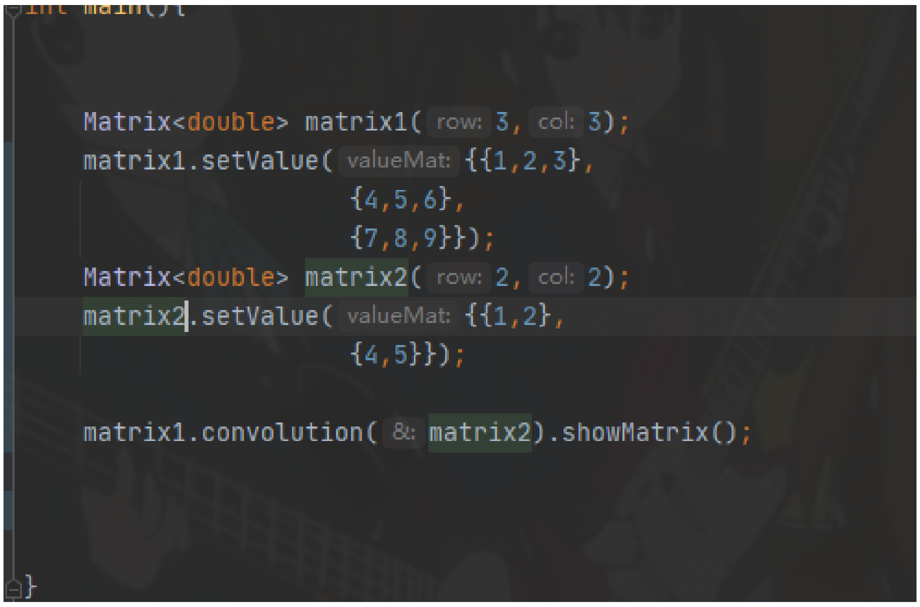


```
using std::complex;
int main(){


    Matrix<double> matrix1( row: 3, col: 3);
    matrix1.setValue( valueMat: {{1,2,3},
                                 {4,5,6},
                                 {7,8,9}});

    cout << matrix1.getSumByCol( colNum: 1);




}
```

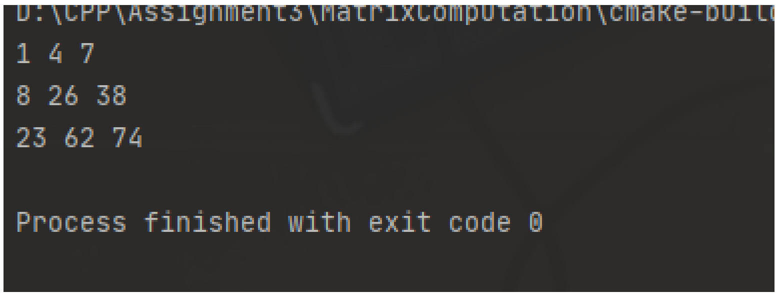


```
D:\CPP\Assignment3\MatrixComputation\cmake-build-debug\MatrixCompu
15
Process finished with exit code 0
```

7) It supports convolutional operations of two matrices. (10 points)



```
int main(){


    Matrix<double> matrix1( row: 3, col: 3);
    matrix1.setValue( valueMat: {{1,2,3},
                      {4,5,6},
                      {7,8,9}});
    Matrix<double> matrix2( row: 2, col: 2);
    matrix2.setValue( valueMat: {{1,2},
                      {4,5}});

    matrix1.convolution( &: matrix2).showMatrix();



}
```



```
D:\CPP\Assignment3\MatrixComputation\cmake-buil
1 4 7
8 26 38
23 62 74

Process finished with exit code 0
```

Part 4 - Difficulties & Solutions

1) Supports all standard numeric types

We found one difficulty is that how to make the matrix support multiple types but still not need to write different functions for each kind of types respectively.

The solution is that we use the genericity to deal with this problem. And in this case, most of the functions can be supported to use in different types except only a few of functions require special modification. In this case, the whole project's code became more concise.

2) Supports computing eigenvalues and eigenvectors

We found in this requirement the difficulty is that the calculation methods of computing eigenvalues and eigenvectors in C++ can not permit the correctness. You can get different results from different calculation methods. We thought it is because the C++ itself can not support such complex calculation well.

The solution is that we use the OpenCV to realize do the calculations.