Task1:

Analysis:

sparseMatrix.hpp

This implementation uses CSR or Yale format. It has elements, col_index, and row_index to store sparseMatrix.

sparseMatrx(int row, int cols): construct empty sparseMatrix object.

SparseMatrix(Matrix<T> & matrix): construct sparseMatrix from Matrix class. (a conversion)

sparseMatrix(const sparseMatrix<T> & other>: copy constructor.

~sparseMatrix(): default deconstructor

ConvertToDense(sparseMatrix): convert a sparseMatrix to dense Matrix.

Thus, this allows a conversion between sparseMatrix and Matrix class with ease.

templateUtil.h

is_same: using std::is_same, it is a function that checks if two types are the same.

Is_complex: by first initializing the variable is_complex_t to false initially, it returns true if and only if the type inside the angular bracket is std::complex<T>.

is_arithmetic: : using std::is_arithmetic , it is a function that checks if a type is arithmetic.

#define if (…) : macro using std::   enable_if, it enables when the type matches.

Code:

```
#ifndef CS205PROJECT_SPARSEMATRIX_HPP
#define CS205PROJECT_SPARSEMATRIX_HPP
#include "matrix.hpp"
#include <iostream>
#include <vector>
template<typename T>
void printVector(const std::vector<T> &V) {
    using namespace std;
    cout << "[ ";
    for_each(V.begin(), V.end(), [](int a) {
        cout << a << " ";
    });
    cout << "]" << endl;
}
template<typename T>
class sparseMatrix {
private:
    int rows, cols;
    std::vector<int> row_index;
```

```
    std::vector<int> col_index;
    std::vector<T> elements;
    int size;
public:
    sparseMatrix(int row, int col) {
        if (row == 0 || col == 0) {
            throw std::invalid_argument("row or column cannot be zero!");
        }
        rows = row;
        cols = col;
        size = row * col;

    }
    sparseMatrix(Matrix<T> & matrix) {
        int m = matrix.getRowSize();
        int n = matrix.getColumnSize();
        if (m == 0 || n == 0) {
            throw std::invalid_argument("row or column cannot be zero!");
        }

        rows = m;
        cols = n;
        int numberOfNonZeroes = 0;
        for (int i = 0; i < m; i ++) {
            for( int j = 0; j < n; j++) {
                if (matrix.get(i,j)!= 0)
                {
                    elements.push_back(matrix.get(i, j));
                    col_index.push_back(j);
                    numberOfNonZeroes++;
                }
            }
            row_index.push_back(numberOfNonZeroes);
        }
    }
    sparseMatrix(const sparseMatrix<T> & other) {
        if (other.rows == 0 || other.cols == 0) {
            throw std::invalid_argument("row or column cannot be zero!");
        }
        rows = other.rows;
        cols = other.cols;
        row_index(other.row_index);
        col_index(other.col_index);

    }
    ~sparseMatrix() = default;
    Matrix<T> convertToDense(sparseMatrix<T> &sparseMatrix) {
        Matrix<T> mat(sparseMatrix.rows,sparseMatrix.cols);
        T arr[sparseMatrix.rows * sparseMatrix.cols];
//        std::cout << sparseMatrix.rows << std::endl;
//        std::cout << sparseMatrix.cols << std::endl;
        //        T * ptr = &arr;
        int k = 0;
        for( int i = 0; i < sparseMatrix.rows;i++) {
            for( int j = 0; j < sparseMatrix.cols; j++) {
                arr[i*sparseMatrix.cols + j ] = 0;
```

```
            }
        }
        for( int i = 0; i < sparseMatrix.rows; i++) {
            int row_elements;
            if (i == 0) {
                row_elements = sparseMatrix.row_index[0];
            } else {
                row_elements = sparseMatrix.row_index[i] - sparseMatrix.row_index[i-1];
            }

            for( int j = 0; j < row_elements; j++) {
                arr[i*sparseMatrix.cols + col_index[k]] = elements[k];
                k++;
            }
        }
//        for(int i = 0; i < sparseMatrix.rows * sparseMatrix.cols; i ++) {
//            std::cout << arr[i] << ' ';
//        }
        mat.set(sparseMatrix.rows * sparseMatrix.cols, arr);
        return mat;
    }
//    T get(int row, int col);
//    sparseMatrix & set(T val, int row, int col);
    void print() {
        printVector(elements);
        printVector(col_index);
        printVector(row_index);
    }
};







#endif //CS205PROJECT_SPARSEMATRIX_HPP
```

Result and verification:

Sparse Matrix

Sparse Matrix conversion:

```
init
       1       0       0
       0       5       0
       7       0       9
       0       0       0
sparseMatrix Representation
[ 1 5 7 9 ]
[ 0 1 0 2 ]
[ 1 2 4 4 ]
sparseMatrix Conversion to Dense
       1       0       0
       0       5       0
       7       0       9
       0       0       0
```

Template Utilities

Test its functions

```
test some template functions
is_same_t<float, float > : 1
is_same_t <int, double> : 0
is_arithmetic_t<std::complex<int>> : 0
is_arithmetic_t<char *> : 0
is_arithmetic_t<int > : 1
is_complex<int> : 0
is_complex<std::complex<int>> : 1
```

Task 3:

Analysis:

Vector arithmetic such as addition, subtraction, scalar multiplication and scalar division are implemented by overloading corresponding operators. On the other hand, Dot product and cross product are implemented by a static function. Precautions are in place to make sure that the operation are valid. Also, only cross product of vector in three dimensions are supported. Also, bear in mind that row vector and column vector are not differentiated.

Matrix arithmetic such as addition, subtraction, scalar multiplication and scalar division, and matrix-vector multiplication are also implemented by overloading operators. Operations such as transposition, conjugation, element-wise multiplication, matrix-matrix multiplication are implemented by a static function. Also, it’s important to bear in mind that these operations don’t directly make changes to the input matrices, instead they return the result of the operation as a new matrix. In addition, it is assumed that vectors in matrix-vector multiplication are of valid type-that is, row or column vectors depending on the order of the multiplication. Moreover, an assumption was made that in conjugation operation, we only need to conjugate the matrix's elements. As a result, no additional transposition is applied to get the conjugate transpose of a matrix. In addition, note that conjugation operation is only define for complex matrices.

Code:

Vector Arithmetic:

```
friend Vector<T> operator/ (const Vector<T> &a_vec, const T& scalar)
{
   if (scalar == static_cast<T>(0.0))
      throw std::runtime_error("Math error: Attempted to divide by zero");

   Vector<T> quotient_vec(a_vec.getDim());

   for (int i = 0; i < a_vec.getDim(); i++)
      quotient_vec.set(i, a_vec.get(i) / scalar);

   return quotient_vec;
}

friend Vector<T> operator* (const T &scalar, const Vector<T> &a_vec)
{
   Vector<T> product_vec(a_vec.getDim());

   for (int i = 0; i < a_vec.getDim(); i++)
      product_vec.set(i, scalar * a_vec.get(i));

   return product_vec;
}

friend Vector<T> operator* (const Vector<T> &a_vec, const T &scalar)
{
   Vector<T> product_vec(a_vec.getDim());

   for (int i = 0; i < a_vec.getDim(); i++)
      product_vec.set(i, a_vec.get(i) * scalar);

   return product_vec;
}

friend Vector<T> operator+ (const Vector<T> &a_vec, const Vector<T> &b_vec)
{
   if (a_vec.getDim() != b_vec.getDim())
      throw std::invalid_argument("Vectors are compatible");
```

```
    Vector<T> sum_vec(a_vec.getDim());

    for (int i = 0; i < a_vec.getDim(); i++)
        sum_vec.set(i, a_vec.get(i) + b_vec.get(i));

    return sum_vec;
}

friend Vector<T> operator- (const Vector<T> &a_vec, const Vector<T> &b_vec)
{
    if (a_vec.getDim() != b_vec.getDim())
        throw std::invalid_argument("Vectors are compatible");

    Vector<T> difference_vec(a_vec.getDim());

    for (int i = 0; i < a_vec.getDim(); i++)
        difference_vec.set(i, a_vec.get(i) - b_vec.get(i));

    return difference_vec;
}

static T Dot_Product(const Vector<T> &a_vec, const Vector<T> &b_vec)
{
    // Check that the number of dimensions match.
    if (a_vec.getDim() != b_vec.getDim())
        throw std::invalid_argument("Vector dimensions do not match.");

    // Compute the dot product.
    T dot_product = static_cast<T>(0.0);

    for (int i = 0; i < a_vec.getDim(); i++)
        dot_product += (a_vec.get(i) * b_vec.get(i));

    return dot_product;
}

static Vector<T> Cross_Product(const Vector<T>& a_vec, const Vector<T>& b_vec)
{
    if (a_vec.getDim() != b_vec.getDim())
        throw std::invalid_argument("Vector dimensions do not match.");

    if (a_vec.getDim() != 3)
        throw std::invalid_argument("Vectors are not three-dimensional");

    std::vector<T> cross_product;
    cross_product.push_back((a_vec.get(1) * b_vec.get(2)) - (a_vec.get(2) * b_vec.get(1)));
    cross_product.push_back(-((a_vec.get(0) * b_vec.get(2)) - (a_vec.get(2) * b_vec.get(0))));
    cross_product.push_back((a_vec.get(0) * b_vec.get(1)) - (a_vec.get(1) * b_vec.get(0)));

    Vector<T> cross_product_vec(cross_product);
```

```
    return cross_product_vec;
}
```

Matrix Arithmetic:

```
//addition
friend Matrix<T> operator+ (const Matrix<T> &X, const Matrix<T> &Y)
{
    HaveSameDim(X, Y);
    Matrix<T> Sum(X.getRow(), X.getCol());

    for (int i = 0; i < Sum.getRow(); i++)
    {
        for (int j = 0; j < Sum.getCol(); j++)
        {
            Sum.get(i, j) = X.get(i, j) + Y.get(i, j);
        }
    }

    return Sum;
}
//subtraction
friend Matrix<T> operator- (const Matrix<T> &X, const Matrix<T> &Y)
{
    HaveSameDim(X, Y);
    Matrix<T> Difference(X.getRow(), X.getCol());
```

```
    for (int i = 0; i < Difference.getRow(); i++)
    {
        for (int j = 0; j < Difference.getCol(); j++)
        {
            Difference.get(i, j) = X.get(i, j) - Y.get(i, j);
        }
    }

    return Difference;
}
//scalar multiplication
friend Matrix<T> operator* (const Matrix<T> &X, const T &scalar)
{
    Matrix<T> Product(X.getRow(), X.getCol());

    for (int i = 0; i < Product.getRow(); i++)
    {
        for (int j = 0; j < Product.getCol(); j++)
        {
            Product.get(i, j) = X.get(i, j) * scalar;
        }

    }

    return Product;
}
```

```
friend Matrix<T> operator* (const T &scalar, const Matrix<T> &X)
{
    Matrix<T> Product(X.getRow(), X.getCol());

    for (int i = 0; i < Product.getRow(); i++)
    {
        for (int j = 0; j < Product.getCol(); j++)
        {
            Product.get(i, j) = scalar * X.get(i, j);
        }
    }

    return Product;
}

//scalar division
friend Matrix<T> operator/ (const Matrix<T> &X, const T &scalar)
{
    if (scalar == static_cast<T>(0.0))
        throw std::runtime_error("Math error: Attempted to divide by zero");

    Matrix<T> Quotient(X.getRow(), X.getCol());

    for (int i = 0 ; i < Quotient.getRow(); i++)
    {
        for (int j = 0; j < Quotient.getCol(); j++)
        {
            Quotient.get(i, j) = X.get(i, j) / scalar;
```

```
        }
    }

    return Quotient;
}

//transposition
static Matrix<T> Transpose(const Matrix<T> &X)
{
    Matrix<T> Transposed(X.getCol(), X.getRow());

    for (int i = 0; i < X.getRow(); i++)
    {
        for (int j = 0; j < X.getCol(); j++)
        {
            Transposed.get(j, i) = X.get(i, j);
        }
    }

    return Transposed;
}

//conjugation / transpose conjugate?
template <typename U = T, IF(is_complex<U>)>
static Matrix<T> Conjugate(const Matrix<T> &X)
{
    Matrix<T> Conjugated(X.getRow(), X.getCol());

```

```
    for (int i = 0; i < Conjugated.getRow(); i++)

    {

        for (int j = 0; j < Conjugated.getCol(); j++)

        {

            T &complex = X.get(i, j);

            Conjugated.get(i, j) = std::conj(complex);

        }

    }



    return Conjugated;
}
//element-wise multiplication
static Matrix<T> Elementwise_Multiplication(const Matrix<T> &X, const Matrix<T> &Y)
{
    HaveSameDim(X, Y);
    Matrix<T> Product(X.getRow(), X.getCol());

    for (int i = 0; i < Product.getRow(); i++)
    {
        for (int j = 0; j < Product.getCol(); j++)
        {
            Product.get(i, j) = X.get(i, j) * Y.get(i, j);
        }

    }

    return Product;
}
//matrix-matrix multiplication
friend Matrix<T> operator* (const Matrix<T> &X, const Matrix<T> &Y)
{
    IsCompatible(X, Y);
    Matrix<T> Product(X.getRow(), Y.getCol());

    for (int i = 0; i < X.getRow(); i++)
    {
        for (int j = 0; j < Y.getCol(); j++)
        {
            Product.get(i, j) = 0;
            for (int k = 0; k < X.getCol(); k++)
            {
                Product.get(i, j) += X.get(i, k) * Y.get(k, j);
            }
        }
```

```
    }

    return Product;
}

//matrix-vector multiplication
friend Vector<T> operator* (const Matrix<T> &X, const Vector<T> a_vec)
{
    if (X.getCol() != a_vec.getDim())
        throw std::invalid_argument("Matrix and vector are not compatible!");

    Vector<T> product_vec(X.getRow());

    for (int i = 0; i < X.getRow(); i++)
    {
        product_vec.set(i, 0);
        for (int j = 0; j < X.getCol(); j++)
        {
            product_vec.set(i, product_vec.get(i) + X.get(i, j) * a_vec.get(j));
        }
    }

    return product_vec;
}

friend Vector<T> operator* (const Vector<T> &a_vec, const Matrix<T> &X)
{
    if (a_vec.getDim() != X.getRow())
        throw std::invalid_argument("Vector and matrix are not compatible!");

    Vector<T> product_vec(X.getCol());

    for (int j = 0; j < X.getCol(); j++)
    {
        product_vec.set(j, 0);
        for (int i = 0; i < X.getRow(); i++)
        {
            product_vec.set(j, product_vec.get(j) + a_vec.get(i) * X.get(i, j));
        }
    }

    return product_vec;
}
```

Result & Verification:

Matrix Arithmetic:

$$M\_1 = \begin{matrix} 1.0 & 2.0 & -1.0 \\ 3.0 & 4.0 & -7.0 \\ 1.0 & -2.0 & 3.0 \end{matrix} \qquad M\_2 = \begin{matrix} 5.0 & -6.0 & -4.0 \\ 7.0 & 8.0 & -2.0 \\ 4.0 & 6.0 & 5.0 \end{matrix}$$

$$M\_C = \begin{matrix} 1 + 1i & -3 - 2i \\ 0 - 3i & 4 + 0i \end{matrix}$$

Scalar_1 = 9.0 scalar_2 = 2.0

Addition:

M_1 + M_2

```
addition
            6            -4            -5
           10            12            -9
            5             4             8
```

Subtraction:

M_1 – M_2

```
subtraction
           -4             8             3
           -4            -4            -5
           -3            -8            -2
```

Scalar multiplication

M_1 * scalar_1

```
scalar multiplication
              9            18            -9
             27            36           -63
              9           -18            27
```

scalar_1 * M_2

```
             45           -54           -36
             63            72           -18
             36            54            45
```

Scalar division

M_1 / scalar_2

```
            0.5             1          -0.5
            1.5             2          -3.5
            0.5            -1           1.5
```

Transposition

Transpose(M_1)

```
              1             3             1
              2             4            -2
             -1            -7             3
```

Transpose(M_2)

```
              5             7             4
             -6             8             6
             -4            -2             5
```

Conjugate

Conjugate(M_C)

```
conjugate
        (1,-1)        (-3,2)
         (0,3)        (4,-0)
```

Element-wise multiplication:

Elementwise_Multplication(M_1, M_2)

```
element-wise multiplication
             5           -12             4
            21            32            14
             4           -12            15
```

Matrix-matrix multiplication

M_1 * M_2

```
matrix-matrix multiplication
            15             4           -13
            15           -28           -55
             3            -4            15
```

<u>Vector Arithmetic</u>:

$$\text{v\_1\_vec} = \begin{matrix} 1.0 \\ 2.0 \\ -2.0 \end{matrix} \qquad \text{v\_2\_vec} = \begin{matrix} 3.0 \\ -4.0 \\ -1.0 \end{matrix}$$

Addition:

v_1_vec + v_2_vec:

```
addition
             4            -2            -3
```

Subtraction:

v_1_vec – v_2_vec:

```
subtraction
          -2          6          -1
```

Scalar multiplication:

v_1_vec * scalar_1:

```
scalar multiplication
          9          18          -18
```

Scalar_2 * v_2_vec

```
          6          -8          -2
```

Scalar division:

v_1_vec / scalar_2

```
scalar division
          0.5          1          -1
```

Dot Product:

v_1_vec dot v_2_vec

```
dot product
-3
```

Cross Product:

v_1_vec cross v_2_vec

```
cross product
          -10          -5          -10
```

Matrix-vector multiplication:

M_1 * v_1_vec

```
matrix-vector multiplication
              7             25              -9
```

v_1_vec * M_2

```
      11             -2            -18
```

Difficulties & Solutions

_I had some difficulties with conjugate.

Task 4

Analysis

Sum: The sum function simply sums up all value of the elements in the matrix.

The Average function, Min function and Max function have two overloading. The zero-argument version compute the average, min and max for all element in the matrix. While the other version takes one argument called axis. When axis = 0, each function returns their corresponding answer along each column which result in a matrix with 1 row. When axis = 1, each function returns their corresponding answer along each row, result in a matrix of 1 column.

The average function, Min function and Max function also have a slightly different implementation when the matrix typename is not a primitive type. Two version of each function exist. Each called to the function will check if the typename is a reference type. If this is true, it will choose the implementation suited for that datatype. For instance, if the typename is complex, we will compare the modulus of two complex number instead of applying the > operator directly to the object.

Code:

```
template <typename U = T, IF(is_arithmetic_t<U>)>
    T Max() {
        T maxVal = mat_ptr[0];

        for (int i = 0; i < size; ++i) {
            if (maxVal < mat_ptr[i]) {
                maxVal = mat_ptr[i];
            }
        }
        return maxVal;
    }
```

```
template <typename U = T, IF(is_complex<U>)>
U Max() {
    U maxVal = mat_ptr[0];
    for (int i = 0; i < size; ++i) {
        if (std::abs(maxVal) < std::abs(mat_ptr[i])) {
            maxVal = mat_ptr[i];
        }
    }
    return maxVal;
}

template <typename U = T, IF(is_arithmetic_t<U>)>
Matrix<T> Max(int axis) {
    CheckAxis(axis);

    if (axis == 0) {
        Matrix<T> maxMatrix(1, col);

        T maxVal = T();
        for (int i = 0; i < col; ++i) {
            maxVal = get(0, i);
            for (int j = 0; j < row; ++j) {
                if (maxVal < get(j, i)) {
                    maxVal = get(j, i);
                }
            }
            maxMatrix.get(0, i) = maxVal;
        }
        return maxMatrix;
    } else if (axis == 1) {
        Matrix<T> maxMatrix(row, 1);
        T maxVal = T();
        for (int i = 0; i < row; ++i) {
            maxVal = get(i, 0);
            for (int j = 0; j < col; ++j) {
                if (maxVal < get(i, j)) {
                    maxVal = get(i, j);
                }
            }
            maxMatrix.get(i, 0) = maxVal;
        }
        return maxMatrix;
    }


}

template <typename U = T, IF(is_complex<U>)>
Matrix<T> Max(int axis) {
    CheckAxis(axis);

    if (axis == 0) {
        Matrix<T> maxMatrix(1, col);

        T maxVal = T();
```

```
        for (int i = 0; i < col; ++i) {
            maxVal = get(0, i);
            for (int j = 0; j < row; ++j) {
                if (std::abs(maxVal) < std::abs(get(j, i))) {
                    maxVal = get(j, i);
                }
            }
            maxMatrix.get(0, i) = maxVal;
        }
        return maxMatrix;
    }
    else if (axis == 1) {
        Matrix<T> maxMatrix(row, 1);
        T maxVal = T();
        for (int i = 0; i < row; ++i) {
            maxVal = get(i, 0);
            for (int j = 0; j < col; ++j) {
                if (std::abs(maxVal) < std::abs(get(i, j))) {
                    maxVal = get(i, j);
                }
            }
            maxMatrix.get(i, 0) = maxVal;
        }
        return maxMatrix;
    }


}
template <typename U = T, IF(is_arithmetic_t<U>)>
T Min() {
    T maxVal = mat_ptr[0];

    for (int i = 0; i < size; ++i) {
        if (maxVal > mat_ptr[i]) {
            maxVal = mat_ptr[i];
        }
    }
    return maxVal;
}
template <typename U = T, IF(is_complex<U>)>
T Min() {
    T minVal = mat_ptr[0];

    for (int i = 0; i < size; ++i) {
        if (std::abs(minVal) > std::abs(mat_ptr[i])) {
            minVal = mat_ptr[i];
        }
    }
    return minVal;
}

template <typename U = T, IF(is_arithmetic_t<U>)>
Matrix<T> Min(int axis) {
    CheckAxis(axis);

    if (axis == 0) {
```

```
        Matrix<T> minMatrix(1, col);

        T minVal = T();
        for (int i = 0; i < col; ++i) {
            minVal = get(0, i);
            for (int j = 0; j < row; ++j) {
                if (minVal > get(j, i)) {
                    minVal = get(j, i);
                }
            }
            minMatrix.get(0, i) = minVal;
        }
        return minMatrix;
    } else if (axis == 1) {
        Matrix<T> minMatrix(row, 1);
        T minVal = T();
        for (int i = 0; i < row; ++i) {
            minVal = get(i, 0);
            for (int j = 0; j < col; ++j) {
                if (minVal > get(i, j)) {
                    minVal = get(i, j);
                }
            }
            minMatrix.get(i, 0) = minVal;
        }
        return minMatrix;
    }


}

template <typename U = T, IF(is_complex<U>)>
Matrix<T> Min(int axis) {
    CheckAxis(axis);

    if (axis == 0) {
        Matrix<T> minMatrix(1, col);

        T minVal = T();
        for (int i = 0; i < col; ++i) {
            minVal = get(0, i);
            for (int j = 0; j < row; ++j) {
                if (std::abs(minVal) > std::abs(get(j, i))) {
                    minVal = get(j, i);
                }
            }
            minMatrix.get(0, i) = minVal;
        }
        return minMatrix;
    }
    else if (axis == 1) {
        Matrix<T> minMatrix(row, 1);
        T minVal = T();
        for (int i = 0; i < row; ++i) {
            minVal = get(i, 0);
            for (int j = 0; j < col; ++j) {
```

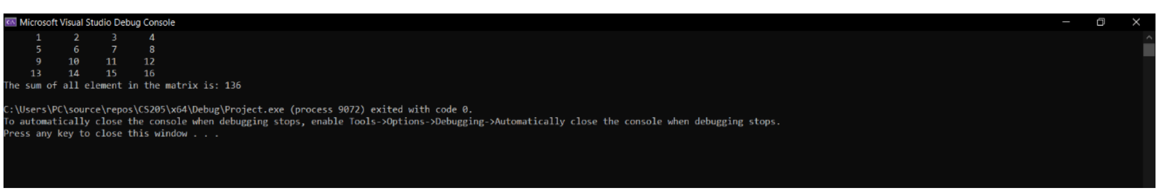
```
                if (std::abs(minVal) > std::abs(get(i, j))) {
                    minVal = get(i, j);
                }
            }
            minMatrix.get(i, 0) = minVal;
        }
        return minMatrix;
    }


}
template <typename U = T, IF(is_complex<U>)>
T Avg() {
    T Avg = T();

    for (int i = 0; i < size; ++i) {
        Avg = Avg +  mat_ptr[i];
    }
    T c (std::real(Avg) / size, std::imag(Avg) / size );
    return c;
}

template <typename U = T, IF(is_arithmetic_t<U>)>
T Avg() {
    T Avg = T();

    for (int i = 0; i < size; ++i) {
        Avg = Avg + mat_ptr[i];
    }
    Avg = Avg / size;
    return Avg;
}

template <typename U = T, IF(is_arithmetic_t<U>)>
Matrix<T> Avg(int axis) {
    CheckAxis(axis);

    if (axis == 0) {
        Matrix<T> AvgMatrix(1, col);

        T avg = T();
        for (int i = 0; i < col; ++i) {
            avg = 0;
            for (int j = 0; j < row; ++j) {


                avg = avg + get(j, i);

            }
            avg = avg / row;
            AvgMatrix.get(0, i) = avg;
        }
        return AvgMatrix;
    } else if (axis == 1) {
        Matrix<T> AvgMatrix(row, 1);
        T avg = T();
```

```
        for (int i = 0; i < row; ++i) {
            avg = 0;
            for (int j = 0; j < col; ++j) {
                avg = avg + get(i, j);

            }
            avg = avg / col;
            AvgMatrix.get(i, 0) = avg;
        }
        return AvgMatrix;
    }

}

template <typename U = T, IF(is_complex<U>)>
Matrix<T> Avg(int axis) {
    CheckAxis(axis);

    if (axis == 0) {
        Matrix<T> AvgMatrix(1, col);

        T avg = T();
        for (int i = 0; i < col; ++i) {
            avg = T();
            for (int j = 0; j < row; ++j) {

                avg = avg + get(j, i);

            }
            T c (std::real(avg)/row, std::imag(avg)/row);
            AvgMatrix.get(0, i) = c;
        }
        return AvgMatrix;
    }
    else if (axis == 1) {
        Matrix<T> AvgMatrix(row, 1);
        T avg = T();
        for (int i = 0; i < row; ++i) {
            avg = T();
            for (int j = 0; j < col; ++j) {

                avg = avg + get(i, j);

            }
            T c(std::real(avg) / col, std::imag(avg) / col);
            AvgMatrix.get(i,0) = c;
        }
        return AvgMatrix;
    }

}

T Sum() {
    T Sum = T();

    for (int i = 0; i < size; ++i) {
```

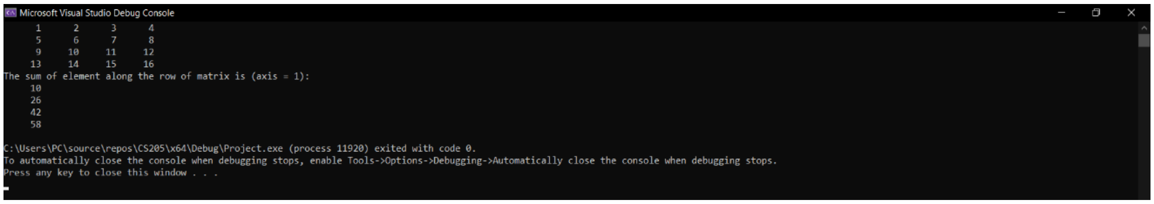
```
        Sum = Sum + mat_ptr[i];
    }

    return Sum;
}

Matrix<T> Sum(int axis) {
    CheckAxis(axis);

    if (axis == 0) {
        Matrix<T> SumMatrix(1, col);

        T sum = T();
        for (int i = 0; i < col; ++i) {
            sum = T();
            for (int j = 0; j < row; ++j) {

                sum = sum + get(j, i);

            }

            SumMatrix.get(0, i) = sum;
        }
        return SumMatrix;
    } else if (axis == 1) {
        Matrix<T> SumMatrix(row, 1);
        T sum = T();
        for (int i = 0; i < row; ++i) {
            sum = T();
            for (int j = 0; j < col; ++j) {

                sum = sum + get(i, j);

            }

            SumMatrix.get(i, 0) = sum;
        }
        return SumMatrix;
    }

}
```

Result of verification:

Result of Sum():

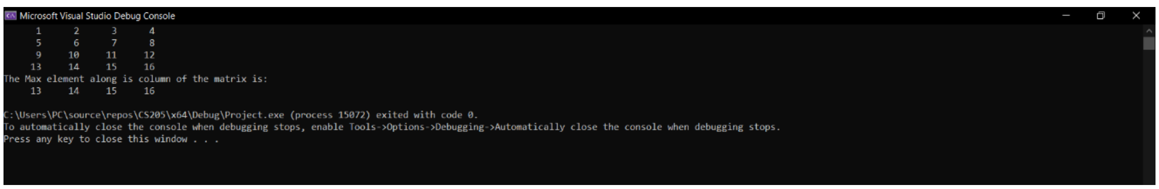


Result of Sum(1):

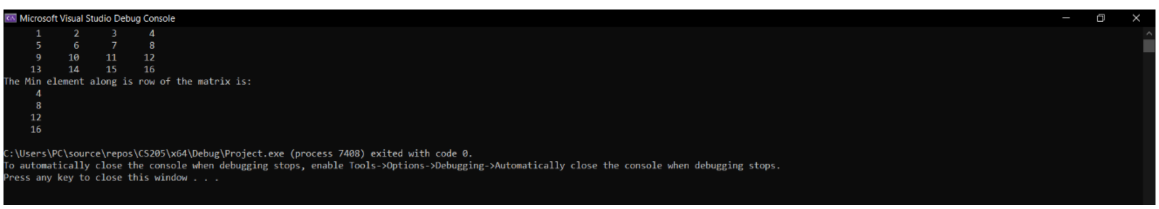


Result of Max():

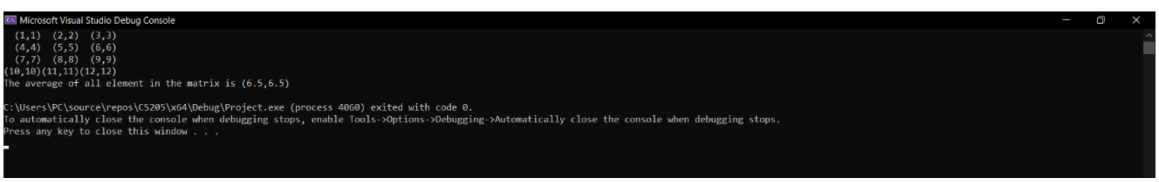
Result of Max(0):



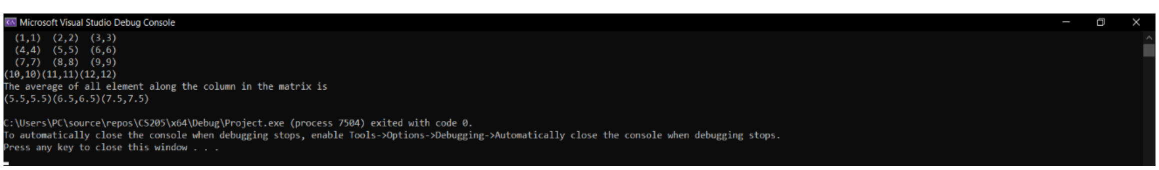
Result of Min(1):



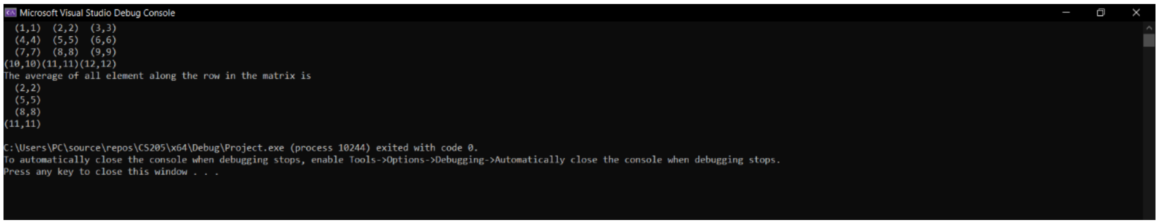
Result of Average() with complex entries:



Result of Average(0) with complex entries:



Result of Average(1) with complex entries:



Problem and Solution:

There was an issue with Average, max and min function when the type name is not primitive type. So, we have to have separate definition of these function for reference type. We will check the type template T to determine which version we should use.

Task 5

Analysis:

Determinant is computed by basically recursively computing determinant of submatrices; specifically, the cofactors. The general formula can be written as

det(A) = $\sum_{j=1}^{n} A_{i,j} C_{i,j}$, where $C_{i,\, j}$ = $(-1)^{i+j}$ * det(A(i | j)) . Note that det(A(i | j)) is the determinant of submatrix of A that is got by ignoring the i-th row and the j-th coloumn.

The inverse can be computed by finding the adjoint matrix. The elements of an adjoint matrix can be found by computing all cofactors for the original matrix and transposing it. Finally, we can use adjoint matrix special property, which is

$Adj(A) \ * \ A \ = \ det(A) \ * \ I$. Thus, we can find the inverse of A by dividing the adjoint of A by its determinant; $A - 1 \ = \ Adj(A) \ / \ det(A)$.

The eigenvalues of a matrix can be computed by QR decomposition. Bear in mind that we can only apply this method reliably for symmetric matrices. The goal of QR decomposition is to decompose a matrix A into a product of an Upper-triangular matrix, R and an orthogonal matrix, Q. To find R, we apply the householder method for each column of A which is finding a reflection matrix P_i such that $P_i*P_{i-1}*\ldots A$ transform the i-th column of A to become column of an upper triangular. The formular for computing P is $P_i = I - 2\vec{n}_i\vec{n}_i^{\,T}$ where $\vec{n}$ is the normalized form of $\vec{u}$. $\vec{u}$ can be computed by $\vec{u} = \vec{a} - [-sign(\vec{a})]\|\vec{a}\|)\vec{b}$ where $\vec{a}$ is the columns vector and $\vec{b}$ is the vector which we want to reflect $\vec{a}$ upon. After reiterating QR_Decompose() and transforming A by A = Q * R, we can get the eigenvalues of A by looking at the diagonal elements of resultant matrix.

After computing the eigenvalues of a matrix, we can compute the corresponding eigenvector by inverse power iteration method. The basic of the method is iteratively applying a formula to a vector, $\vec{v}_{k+1} = \frac{(A-\mu I)^{-1}\,\vec{v}_k}{\|(A-\mu I)^{-1}\,\vec{v}_k\|}$, where μ is the eigenvalue and A is our original matrix. While we can do it by power iteration method, inverse power iteration method can converge faster.

Code:

```
//determinant
static T Determinant(const Matrix<T> &X)
{
   if(!IsSquare(X))
```

```
        throw std::invalid_argument("The matrix is not square! Non-square matrices do not have determinant!");

    T determinant = 0;
    int n = X.getCol();

    if (n == 1)
        return X.get(0, 0);
    else
    {
        for (int j = 0; j < n ; j++)
        {
            determinant += (((0+j) % 2 == 0) ? 1 : -1) * X.get(0, j) * Determinant(SubMatrix(X, 0, j));
        }
    }

    return determinant;
}
//find cofactor
static Matrix<T> SubMatrix(const Matrix<T> &X, int row, int col)
{
    //ignore i = row and j = col
    int m = X.getRow();
    int n = X.getCol();
    Matrix<T> SubMatrix(m-1, n-1);

    for (int i = 0; i < m; i++)
    {
```

```
        for (int j = 0; j < n; j++)
        {
            if (i != row && j != col)
            {
                if (j < col && i < row)
                    SubMatrix.get(i, j) = X.get(i, j);
                else if (j < col && i > row)
                    SubMatrix.get(i-1, j) = X.get(i, j);
                else if (j > col && i < row)
                    SubMatrix.get(i, j-1) = X.get(i, j);
                else if (j > col && i > row)
                    SubMatrix.get(i-1, j-1) = X.get(i, j);

            }
        }
    }

    return SubMatrix;
}

//get adjoint
static Matrix<T> Adjoint(const Matrix<T> &X)
{
    if (!IsSquare(X))
        throw std::invalid_argument("The matrix is not square! Non-square matrices do not have an adjoint matrix!");

    if (IsZero(X))
```

```
        throw std::invalid_argument("The matrix is a Zero matrix! Zero matrices do not have an adjoint 
matrix");


    int n = X.getCol();

    Matrix<T> Adj(n, n);


    if (n == 1)

    {

        Adj.get(0, 0) = static_cast<T>(1.0);

        return Adj;

    }


    for (int i = 0; i < n; i++)

        for (int j = 0; j < n; j ++)

            Adj.get(j, i) = (((i + j) % 2 == 0) ? 1 : -1) * Determinant(SubMatrix(X, i, j));


    return Adj;


}

//find inverse

static Matrix<T> Inverse(const Matrix<T> &X)

{

    if (!IsSquare(X))

        throw std::invalid_argument("The matrix is not square! Non-square matrices do not have an 
inverse!");


    int n = X.getCol();

    Matrix<T> Inv(n, n);

    Matrix<T> Adj(n, n);
```

```
    T det = Determinant(X);


    if (det == static_cast<T>(0.0))

    {

        throw new std::invalid_argument("The matrix is singular! Singular matrices do not have an inverse!");

    }


    Adj = Adjoint(X);


    Inv = Adj / det;


    return Inv;


}


static T Trace(const Matrix<T> &X)

{

    if (!IsSquare(X))

        throw std::invalid_argument("The matrix is not square! Unable to compute trace!");


    int n = X.getCol();

    T trace = static_cast<T>(0);

    for (int i = 0; i < n; i++)

            trace += X.get(i, i);


    return trace;

}

static void QR_Decompose(const Matrix<T> &A, Matrix<T> &Q, Matrix<T> &R)
{
```

```
Matrix<T> A_Copied = A;

if (!IsSquare(A_Copied))
    throw std::invalid_argument("The matrix is not square! Unable to perform QR decomposition!");

int n = A_Copied.getCol();

std::vector<Matrix<T>> Ps;
for (int j = 0; j < n - 1; j++)
{
    Vector<T> a_vec (n - j);
    Vector<T> b_vec (n - j);

    for (int i = j; i < n; i++)
    {
        a_vec.set(i-j, A_Copied.get(i, j));
        b_vec.set(i-j, static_cast<T>(0.0));
    }
    b_vec.set(0, static_cast<T>(1.0));

    //length of a-vector
    T a_norm = Vector<T>::Norm(a_vec);

    //sign
    int sign = -1;
    if (a_vec.get(0) < static_cast<T>(0.0))
        sign = 1;

    //compute n-vector
    Vector<T> n_vec = Vector<T>::Normalize(a_vec - (sign * a_norm * b_vec));

    //convert n-vector to matrix to transpose
    Matrix<T> N_Mat (n - j, 1);
    for (int i = 0; i < n - j; i++)
        N_Mat.get(i, 0) =  n_vec.get(i);

    //transpose n_mat
    Matrix<T> N_Mat_T = Transpose(N_Mat);

    //create an identity matrix
    Matrix<T> I (n - j, n - j);
    SetToIdentity(I);

    //Compute P_Temp
    Matrix<T> P_Temp = I - static_cast<T>(2.0) * N_Mat * N_Mat_T;

    //form the P matrix with original dimensions
    Matrix<T> P (n, n);
    SetToIdentity(P);

    for (int row = j; row < n; row++)
        for (int col = j; col < n; col++)
```

```
            P.get(row, col) = P_Temp.get(row - j, col - j);

        //store result to Ps
        Ps.push_back(P);

        //Apply transformation to inputMatrix
        A_Copied = P * A_Copied;
    }

    //compute Q
    Q = Ps.at(0);

    for (int i = 1; i < n - 1; i++)
        Q = Q * Transpose(Ps.at(i));

    //compute R
    int p_num = Ps.size();
    R = Ps.at(p_num - 1);

    for (int i = p_num - 2; i >= 0; i--)
    {
        R = R * Ps.at(i);
    }

    R = R * A;
}

static void Eigenvalues(const Matrix<T> &A, std::vector<T> &eigenvalues)
{
    //male Copy of A
    Matrix<T> A_Copied = A;

    //verify A is square
    if (!IsSquare(A_Copied))
        throw std::invalid_argument("The matrix is not square! Unable to compute eigenvalues!");

    //verify A is symmetric
    if (!IsSymmetric(A_Copied))
        throw std::invalid_argument("Unable to compute eigenvalues for non-symmetric matrices");

    int n = A_Copied.getCol();

    //create an identity matrix
    Matrix<T> I (n, n);
    SetToIdentity(I);

    //create matrices to store Q and R
    Matrix<T> Q (n, n);
    Matrix<T> R (n, n);

    int max_iteration = 10e3;
    int iteration_cnt = 0;
```

```
    while (iteration_cnt < max_iteration)
    {
        QR_Decompose(A_Copied, Q, R);

        A_Copied = R * Q;

        //check if A is close enough to an upper-triangular
        if (IsCloseToUEnough(A_Copied))
            break;

        iteration_cnt++;
    }

    //eigenvalues is the diagonal elements of A
    for (int i = 0; i < n; i++)
    {
        eigenvalues.push_back(A_Copied.get(i, i));
    }
}

//find eigenvector by inverse power iteration method
static void Eigenvectors(const Matrix<T> &A, const T &eigenvalue, Vector<T> &eigenvector)
{
    //verify A is square
    IsSquare(A);

    std::random_device myRandomDevice;
    std::mt19937 myRandomGenerator(myRandomDevice());
    std::uniform_int_distribution<int> myDistribution(1.0, 10.0);

    int n = A.getCol();

    Matrix<T> I(n, n);
    SetToIdentity(I);

    Vector<T> v_vec(n);
    for (int i = 0; i < n; i++)
        v_vec.set(i, static_cast<T>(myDistribution(myRandomGenerator)));

    int max_iteration = 100;
    int iteration_cnt = 0;
    T min_epsilon = static_cast<T>(1e-9);
    T epsilon = static_cast<T>(1e6);
    Vector<T> prev_vec(n);

    while ((iteration_cnt < max_iteration) && (epsilon > min_epsilon))
    {
        prev_vec = v_vec;

        v_vec = Vector<T>::Normalize(Inverse(A - (eigenvalue * I)) * v_vec);

        epsilon = Vector<T>::Norm((v_vec - prev_vec));
```

```
    iteration_cnt++;
  }

  eigenvector = v_vec;
}
```

Result & Verification:

$$M\_Sym = \begin{matrix} 5.0 & -6.0 & -4.0 \\ -6.9 & 8.0 & -2.0 \\ -4.0 & -2.0 & 5.0 \end{matrix}$$

_Determinant:

Determinant(M_2)

```
determinant
478
```

_Inverse:

Inverse(M_2)

```
Inverse
      0.108787     0.0125523     0.0920502
    -0.0899582     0.0857741    -0.0376569
     0.0209205     -0.112971      0.171548
```

_Trace:

Trace(M_2)

```
Trace
18
```

_Eigenvalues:

Eigenvalues(M_Sym, eigenvalues)

```
eigenvalues
12.8096 7.51677 -2.32638
```

_Eigenvectors:

Eigenvector(M_Sym, eigenvalues.at(i), eigenvector_vec)

```
eigenvectors
     -0.647245       0.749391       0.139597
     -0.316421      -0.430734        0.84519
      0.693507       0.502874       0.515913
```

Note that the three rows correspond to eigenvectors for each eigenvalues

Difficulties & Solutions:

I had difficulties with finding eigenvalues and eigenvectors.

Task6:

Analysis:

Reshaping: in order to reshape we simply change the value of row and col if the product of the new value equal to the original size.

Slicing: The slicing operation is done in two steps. First, we find the index of the element that will be present in the slicing. This is done by finding the all the row and column index in two set. The entries of the slice are the cross product of these two sets, that is (Set of row index) x (Set of column index).

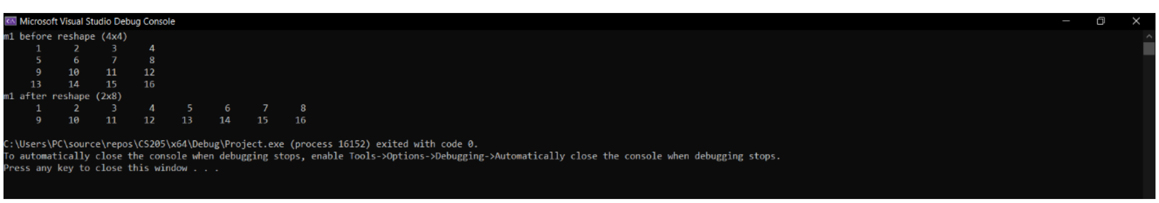
Code:

```
Matrix<T> ReturnSlice(std::vector<int> row, std::vector<int> col) {
    Matrix<T> SliceRes(row.size(), col.size());

    for (int i = 0; i < row.size(); ++i) {
        int r = row[i];
        for (int j = 0; j < col.size(); ++j) {
            int c = col[j];
            SliceRes.get(i, j) = get(r, c);
        }
    }

    return SliceRes;

  }

  void SliceIndex(std::vector<int> &slice, int start, int end, int step) {
    if (step < 0) {
        int tmp = start;
        start = end;
        end = tmp;
    }
```

```
    int i = start;

    while ((step > 0 && i >= start && i <= end) || (step < 0 && i >= end && i <= start)) {
      slice.push_back(i);
      i += step;
    }
  }
Matrix<T> Slice(int rowStart, int rowEnd, int rowStep,
          int colStart, int colEnd, int colStep) {
    if (rowStart > rowEnd) {
      throw std::invalid_argument(
          "slice must start from smaller row to bigger row (use negative step to slice backward)");
    }

    if (colStart > colEnd) {
      throw std::invalid_argument(
          "slice must start from smaller column to bigger row (use negative step to slice backward)");
    }
    ValidRowIndex(rowStart);
    ValidRowIndex(rowEnd);
    ValidColumnIndex(colStart);
    ValidColumnIndex(colEnd);

    std::vector<int> rowIndex;
    SliceIndex(rowIndex, rowStart, rowEnd, rowStep);
    std::vector<int> colIndex;
    SliceIndex(colIndex, colStart, colEnd, colStep);

    return ReturnSlice(rowIndex, colIndex);

    }
void reshape(int row, int col) {
    if (row * col != size) {
      throw std::length_error("Cannot reshape because the two matrix have different shape");
    }
    this->row = row;
    this->col = col;
  }
```

Result and verification:

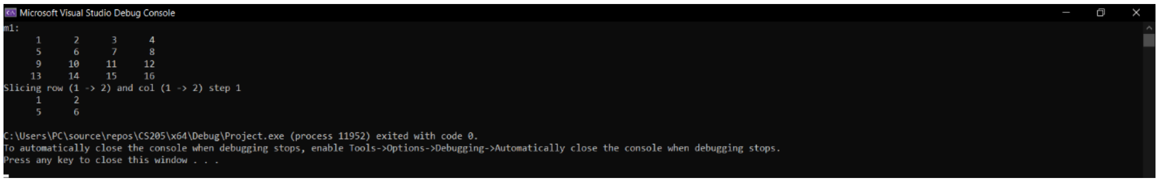
Result of Reshape:



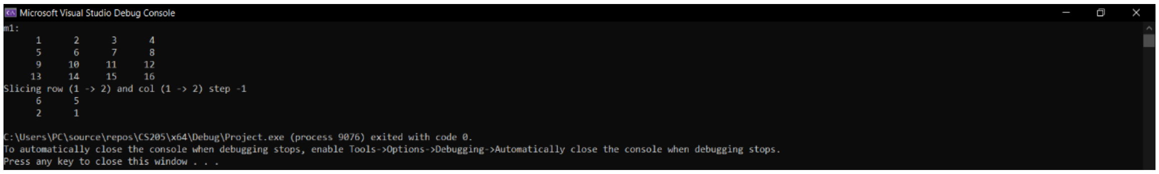
```
Microsoft Visual Studio Debug Console
m1 before reshape (4x4)
     1       2       3       4
     5       6       7       8
     9      10      11      12
    13      14      15      16
m1 after reshape (2x8)
     1       2       3       4       5       6       7       8
     9      10      11      12      13      14      15      16

C:\Users\PC\source\repos\CS205\x64\Debug\Project.exe (process 16152) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Result of Slicing:

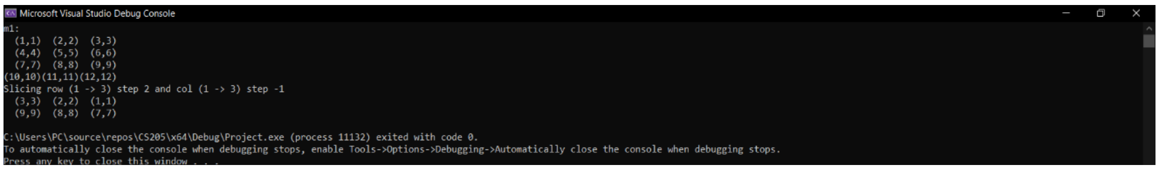
Slice(0,1,-1,0,1-1)



Slice(0,1,-1,0,1-1), both the step for row and column is negative so the slice are arrange backward.



Slice(0,2,2,0,2,-1) with complex entries



Task7:

Analysis:

Convolution: The convolution of two matrix is similar to a dot product between two matrices. In the operation, the second operand is called the kernel whose dimension must be smaller than the first operand. To get each entry of the convolution we place the kernel on the first matrix, then multiply and sum their overlapping element. Then we move the kernel and repeat the process until the kernel overlap the lower right corner of the matrix.

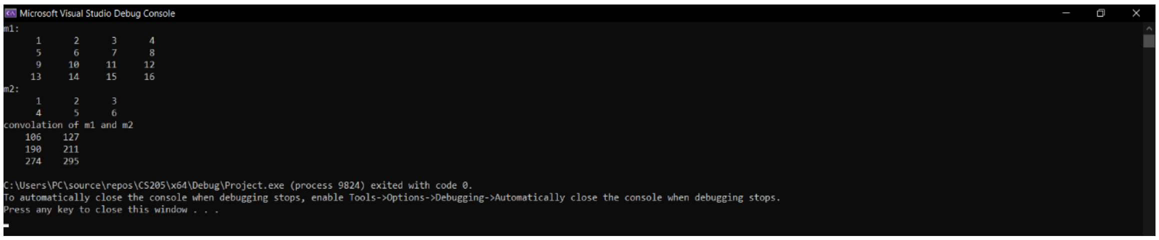
Code:

```
Matrix<T> Convolve(Matrix<T> kernel) {
	if (this->row < kernel.row || this->col < kernel.col) {
	    throw std::length_error("Dimension of kernel is bigger than the left matrix");
	}
	int row = this->row - kernel.row + 1;
	int col = this->col - kernel.col + 1;

	Matrix<T> ans(row, col);

	for (int i = 0; i < row; ++i) {
	    for (int j = 0; j < col; ++j) {
	        T sum = T();
	        for (int ii = 0; ii < kernel.row; ++ii) {
	            for (int jj = 0; jj < kernel.col; ++jj) {
	                sum += kernel.get(ii, jj) * get(i + ii, j + jj);
	            }
	        }
```

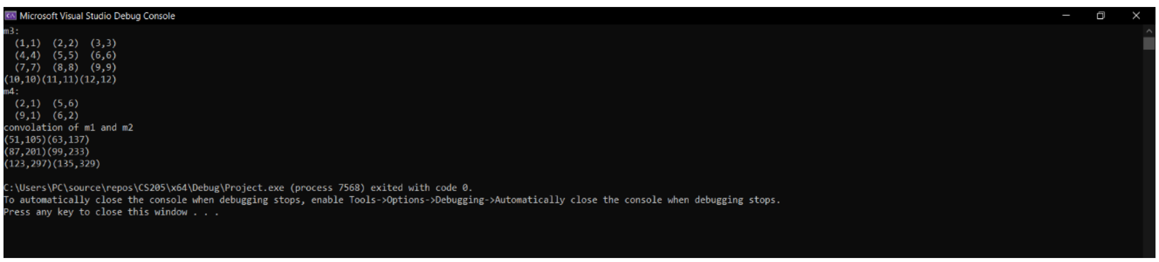
```
            ans.get(i, j) = sum;

        }
    }
    return ans;

}
```

Result and verification:

Convolution of m1 and m2:



Convolution of m3 and m4 (complex entries)



Task8

Analysis:

convertToOpenCV: first check type of matrix using is_same, next we initialize a cv::Mat by row,col, type and pointer of the Matrix class.

convertFromOpenCV: create an array to store the elements then iterate through the whole matrix and get its value. Then, the Matrix class is instantiated with the rows, cols and array.

Code:

```
template <typename T>
cv::Mat convertToOpenCV(Matrix<T> &matrix) {
    int type;
    if (is_same_t<T, uint8_t>) {
        type = 0;
    } else if (is_same_t<T, int8_t>) {
        type = 1;
    } else if (is_same_t<T, uint16_t>) {
        type = 2;
    } else if (is_same_t<T, int16_t>) {
```

```
        type = 3;
    } else if (is_same_t<T, int32_t>) {
        type = 4;
    } else if (is_same_t<T, float>) {
        type = 5;
    } else if (is_same_t<T, double>) {
        type = 6;
    } else {
        type = 7;
    }
    cv::Mat mat(matrix.getRowSize(), matrix.getColumnSize(), type, matrix.getPtr());
    return mat;
};
template <typename T>
Matrix<T> convertFromOpenCV(cv::Mat &mat) {
    T arr[mat.rows * mat.cols * mat.channels() + 1];
    for (int i = 0; i < mat.rows; i++) {
        for (int j = 0; j < mat.cols * mat.channels(); j++) {
            auto *p = mat.ptr(i, j);
            arr[i * mat.rows + j] = *p;
        }
    }
    Matrix<T> matrix(mat.rows, mat.cols * mat.channels());
    matrix.set(mat.rows * mat.cols * mat.channels(), arr);
    return matrix;
}
```

Result and verification:

openCV::mat conversion to and from openCV

```
init
      1      2      3      4
      5      6      7      8
      9     10     11     12
     13     14     15     16
openCVMatrix: convert To OpenCV
[1, 2, 3, 4;
 5, 6, 7, 8;
 9, 10, 11, 12;
 13, 14, 15, 16]
openCVMatrix: convert From OpenCV
      1      2      3      4
      5      6      7      8
      9     10     11     12
     13     14     15     16
```