Part1

Introduction

A pre-written “MatrixTest.cpp” file was provided with functions to perform different testing shown in the selection interface below. The implementation of the codes and the corresponding result are provided in the section: Method 1.1-1.2 (Task1-Task6). The original “MatrixTest.cpp” has not been modified, except the assignment operator testing (which was required in Coursework2 Question Sheet). The exact implementation of the assignment operator testing is provided in section Mathod 1.3 (Task7). This report only provides the relevant part in “MatrixTest.cpp”, and the rest of codes of the “MatrixTest.cpp” are provided in Codelisting.

```
[mphy0030@gzhang MPHY0030]$ g++ -c Matrix.cpp
[mphy0030@gzhang MPHY0030]$ g++ -c MatrixTest.cpp
^[[A[mphy0030@gzhang MPHY0030]$ g++ Matrix.o MatrixTest.o -o MatrixTest
[mphy0030@gzhang MPHY0030]$ ./MatrixTest
Choose to test one of the following:
  Enter '1' for the constructors
  Enter '2' for the assignment operator
  Enter '3' for the Toeplitz function
  Enter '4' for the Transpose function
  Enter '5' for the multiplication
  Enter '6' for the Row or Column Exchange functions
  Enter '7' for the other functions
```

Method

1.1 Macroscopic structure of the class “Matrix”

In “Matrix.h” file, start by introducing the wanted library and the relevant namespace.

```
 8//prevent multiple includes
 9#ifndef MATRIX_H_
10#define MATRIX_H_
11
12#include <iostream>//choose the library that we want to use
13using namespace std;//it declares that the program will be assessing entities who are part of the namespace called "std."
```

```
102#endif /* MATRIX_H_ */
```

C/C++/ObjC Header Tab Width: 4 Ln 12, Col 60 INS

Then, define the class “Matrix”: essential member variables like noOfRows, noOfColumns and element data which defines a matrix, and member function which returns the index of the element position are protected. Accessible methods to other class or functions are declared and defined in the area of “public”. These methods will be introduced based on their functions and purposes.

```
24// Description:
25// A class for modelling noOfRows-by-noOfColumns  matrices.
26class Matrix {
27protected:
28
29// These are the only member variables allowed! And these member variable are protected.
30
31    int noOfRows; // the member variable that stores the number of rows, the data type is supposed to be integer.
32    int noOfColumns;// the member variable that stores the number of columns, the data type is supposed to be integer.
33    double *data;// the member variable that stores the address to the 1-D array of the matrix entries arranged column-wise,
   the correponding data type has to be double, as the storing capacity of double outperforms other data types and it can store
   non-integer number data.
34
35    int GetIndex (const int rowIdx, const int columnIdx) const;// the member function that determines the position (index)
   along the 1-D array (data) of a matrix entry in the row specified by rowIdx and the column specified by columnIdx. In short,
   get the index of the position from the matrix entry. The data type of GetIndex is an integer, and has to be consistent with
   the arguments,as it returns an index, so its data type is int. The const-ness in the end, means the function promises not to
   change the result of GetIndex, whether or not you require that promises.
36
37public:
38//These are accessible methods! Public grants their accessibility.
```

In this program, there are many prototypes and special symbols are used in declaration and construction of a specific function. Below are the general rules applied to the using of the common appeared prototypes and symbols. For using static prototype, it means that the methods attached to the static prototype are only available to the class itself in which the function is in (Matrix in this case).

```
15    // General rule for const: the const-ness of the corresponding arguments in all the functions means the function promises not to change the argument, whether or not you require that promises
16    // General rule for using double: double can store both integer and non-integer numbers and the storing capacity of double outperforms other data types like int. And it is particularly useful for address of an array like *data.
17    // General rule for using int: good for integer-based index data, like row and column etc.
18    // General rule of &: & has the use of dereferencing the address stored in the input, which returns the output as visualised matrix elements(number).
19    //General rule of *, which is known as pointer which can store data as address. & can dereference those data stored by the pointer.
20    //General rule of using virtual: virtual allows the child to be called, rather than calling the parents, decalre the function that needs polymorphisms as virtual
21    //General rule of using friend: friend grants open access to non-member functions.
22    // General rule of using static: objects declared as static have a scope till the lifetime of program.
```

1.2 Member function and Methods

(1) Member function in “protected” region

Create member function GetIndex which returns the index of the element position in “Matrix.cpp”.

```
11int Matrix::GetIndex (const int rowIdx, const int columnIdx) const {
12
13    return rowIdx*noOfColumns + columnIdx;
14}// This determines the position (index) along the 1-D array (data) of a matrix entry in the row specified by rowIdx and the column specified by columnIdx.
```

(2) Constructors and destructor testing (task1)

In the ConstructorsTesting sesson of the default codes provided in the “MatrixTest.cpp”, operations of applying the standard constructor, the assignment constructor, Matrix::Zeros(2,4), Matrix::Ones(2,4) and the copy constructor are involved. A destructor will also be used to free the memory stored by the dynamic array.

To perform the tasks shown above, the following steps are taken:

Step1: In “Matrix.h”, create headers of the functions used for the constructor testing

```
38// Constructors and destructors
39//task1
40    Matrix (const int row, const int col); //default constructor (M1), which constructs a standard matrix of values with noOfRows rows and with noOfCols columns. The return type of the output has to be a matrix.
41
42    Matrix (const Matrix& mat); //copy constructor (M2), which stores data from the input matrix (mat). const ensures that the argument will not be changed by the function. & has the use of dereferencing the address stored in mat, which returns the output as visualised matrix elements.
43
44    Matrix& operator= (const Matrix& mat_a);// assignment constructor(M3), which assigns the value of the right hand side input matrix(mat_a) to the left hand side matrix. As = is introduced for this constructor, so word operator is needed. & has the use of dereferencing the address stored in both lhs and rhs matrix, which returns the output as visualised matrix elements.
45
46    virtual ~Matrix ();// destructor, which frees the memory address stored occcupied by the dynamical array.
47
48// static member functions for named constructors, objects declared as static have a scope till the lifetime of program.
49
50    static Matrix Ones(const int row1, const int col1);// which returns a matrix filled with value 1 of row1 rows and with col1 columns.
51
52    static Matrix Zeros(const int row0, const int col0);//which returns a matrix filled with value 0 of row0 rows and with col0 columns.
```

```
91    friend ostream& operator<< (ostream& out,const Matrix& rhs);// Main function: print the output in the established format. friend grants open access to non-member function ostream&, in which output stream objects (ostream& out) can write sequences of characters and represent other kinds of data. Non-class function operator<< grants access to the private member functions of Matrix, inserts formatted output.
```

This ostream& operator << function which prints the output in the established format, for any operations of which involves the operator << . Non-class function operator<< grants access to the private member functions of Matrix, inserts formatted output. And this applies to all the testing, not only the ConstructorTesting in the “MatrixTest.cpp”.

Step2: In “Matrix.cpp”, create the relevant constructors and the destructor, establish the desire format for Matrix::Zeros, Matrix::Ones and output printing.

```cpp
Matrix::Matrix (const int row, const int col)  {
    noOfRows = row; // assign const input integer row to noOfRows, data will have row rows
    noOfColumns = col;// assign const input integer col to noOfColumns, data will have col columns
    data = new double[row*col*3];// assign data to a dynamical double array which can store a number of row*col*3 data points.
    for (int i = 0; i < row*col; ++i){
        data[i] = 0.0;
    }
}// standard constructor(M1), which constructs a standard matrix of 0 with row rows and with col columns. The return type of the output is a matrix.

Matrix::Matrix (const Matrix& mat) {
    noOfRows = mat.noOfRows;// assign the noOfRows of the input Matrix mat to noOfRows
    noOfColumns = mat.noOfColumns;// assign the noOfColumns of the input Matrix mat to noOfColumns
    data = new double[mat.noOfRows*mat.noOfColumns*3];//assign data to a dynamical double array which can store a number of mat.noOfRows*mat.noOfColumns*3 data points.
    for (int i = 0; i < mat.noOfRows*mat.noOfColumns; ++i) {
        data[i] = mat.data[i];// asssigns the element value in the input Matrix mat to the element at ith position in data
    }
}// copy constructor(M2), which stores data from the input Matrix (mat).


Matrix& Matrix::operator= (const Matrix& mat_a) {
    noOfRows = mat_a.noOfRows;
    noOfColumns = mat_a.noOfColumns;
    for (int i = 0; i < mat_a.noOfRows*mat_a.noOfColumns; ++i) {
        data[i] = mat_a.data[i];
    }
}
// assignment constructor(M3), which assigns the noOfRows, noOfColumns, the element value of the right hand side input matrix (mat_a) to the corresponding quantities in the Matrix in the left hand side of operator =

Matrix::~Matrix () {
    delete[] data;
}
//destructor, which frees the memory address stored occcupied by the dynamical array.
```

```cpp
Matrix Matrix::Zeros (const int row0, const int col0) {
    Matrix matrix(row0, col0);
    return matrix;
}// As standard constructor returns a row0 rows and a col0 columns of zeros, so it is basically creating a matrix of zeros with standard constructor and return itself.

Matrix Matrix::Ones (const int row1, const int col1) {
    Matrix matrix(row1, col1);
    for (int i = 0; i < row1*col1; ++i){
        matrix.data[i] = 1.0;
    }
    return matrix;
}// create a standard matrix of zeros, and assign 1.0 to every element in the standard matrix

ostream& operator<< (ostream& out, const Matrix& rhs) {

    for (int i = 0; i < rhs.noOfRows; ++i) {
        for (int j = 0; j < rhs.noOfColumns; ++j){
            cout << "        ";
            if (rhs.data[rhs.GetIndex(i,j)] < 0){
                out << rhs.data[rhs.GetIndex(i,j)] << "    ";
            }
            else{
                out << " " << rhs.data[rhs.GetIndex(i,j)] << "    ";
            }
        }
        out << endl;
    }
    return out;
}// Take the input Matrix rhs (the matrix in the right hand side of operator =), transform it into the desired display output, and return the final display output
```

Step3: Display the results for task1

```
Testing the Matrix constructors:
Case 1: Creating a 2x4 matrix of zeros with the standard constructor:
          0             0             0             0
          0             0             0             0

Press any key to continue ...
```

```
Case 2: Creating a 2x4 matrix of zeros with the static Zeros:
         0            0            0            0
         0            0            0            0

Press any key to continue ...

Case 3: Creating a 2x4 matrix of ones with the static Ones:
         1            1            1            1
         1            1            1            1

Press any key to continue ...

Case 4: Copying a 2x4 matrix of ones with the copy constructor:
The input matrix =
         1            1            1            1
         1            1            1            1

The copy =
         1            1            1            1
         1            1            1            1

Press any key to continue ...

Enter '0' to exit or '1' to choose another test
```

(3) Toeplitz Testing(Task2)

ToeplitzTestingHelper is the main function which executes operations for ToeplitzTesting in "MatrixTest.cpp". On the foundation of the previously created functions, there are operations in ToeplitzTestingHelper, require functions to print the input const double array in a desired format and also to create a Toeplitz function.

Hence, there are 3 steps involved in this testing.

Step1: Create the print and Toeplitz function header in "Matrix.h". Static means these functions are only available to the Matrix class.

```
53    static void Print(const double *array, const int noOfRows, const int noOfColumns);// Static Print, which displays the
  *array in the form of a 2-D matrix, with noOfRows rows and noOfColumns columns.
54
55    static Matrix Toeplitz(const double *column, const int noOfRows,const double *row, const int noOfColumns);//Static
  Toeplitz, which returns a toeplitz matrix with noOfRows rows and noOfColumns columns, based on the values stored in *column
  and *row.
```

Step2: Create the corresponding print and Toeplitz functions in "Matrix.cpp".

```
80//task 2
81void Matrix::Print (const double *array, const int noOfRows, const int noOfColumns) {
82    double Expected_M[noOfRows][noOfColumns];
83        for (int i = 0; i < noOfRows; ++i){
84            for (int j = 0; j < noOfColumns; ++j){
85                    Expected_M[i][j] = array[j* noOfRows+ i];
86                    cout << "         ";
87                    if(Expected_M[i][j] < 0){
88                        cout << Expected_M[i][j] << "   ";
89                    }
90                    else{
91                        cout <<" ";
92                        cout << Expected_M[i][j] <<"   ";
93                    }
94            }
95        cout << "\n";
96        }
97}// This function takes a const double *array as a input, and create a new double Expected_M 2-D matrix, which stores
  element data with a corresponding position from input array, and print it out in an established format (a matrix with
  noOfRows rows and noOfColumns columns).
98
99Matrix Matrix::Toeplitz(const double *column, const int noOfRows, const double *row, const int noOfColumns) {
100    Matrix Mat(noOfRows, noOfColumns);// Create a Matrix Mat with noOfRows rows and noOfColumns columns
101    int num =0;
102    for (int i = 0; i < noOfRows; ++i) {
103        for (int j = 0; j < noOfColumns; ++j) {
104            if (i == 0) {//Case:create a Toeplitz Matrix based on the values stored in row
105                Mat.data[j] = row[j];
106            }
107            if (j == 0) {//Case:create a Toeplitz Matrix based on the values stored in column
108                Mat.data[i*noOfColumns] = column[i];
109            }
110            if (i == j) {//Case: assign the first value of column to the main diagonal during Toeplitz Matrix creation,
  where row[0] = column[0] in this case
111                Mat.data[i*noOfColumns + j] = column[0];
112            }
```

```
            }
            if (i > j && j > 0) {// Case: the first value of row is not equal to that of column where noOfRows < noOfColumns
   in this case,assign the values after the first element in column to the second onward elements in the first column of the
   Toeplitz Matrix. All other sub diagnol are correspond to the values along row. Then display the warning for one time
                num = num + i;
                if(column[0] != row[0] && num == 1){
                    cout << "Warning: First element of input column does not match first element of input row!" <<endl;
                    cout <<"\t";
                    cout << "Row wins the diagonal war." <<endl;
                }
                Mat.data[i*noOfColumns + j] = column[i - j];
                }
            if (i < j && i > 0) {// Case: the first value of row is not equal to that of column where noOfRows > noOfColumns
   in this case, assign the values after the first element in row to the second onward elements in the first row of the
   Toeplitz Matrix. All other sub diagnol are correspond to the values along column. Then display the warning for one time
                num = num + i;
                if(column[0] != row[0] && num == 1){
                    cout << "Warning: First element of input column does not match first element of input row!" <<endl;
                    cout <<"\t";
                    cout << "Column wins the diagonal war." <<endl;
                }
                Mat.data[i*noOfColumns + j] = row[j - i];
            }
        }
    }
    return Mat;
}//This function takes const double-type row and const double-type columnas input arrays, which creates and returns a
   Toepltz matrix as a Matrix with noOfRows rows and noOfColumns columns.
```

Step3: Display the result for Toeplitz Testing

```
Testing the static function Matrix::Toeplitz:
Case 1: When the number of columns is larger than the number of rows
The 1st column =
         2
         1
         0
        -1

The 1st row =
         2           0          -1

The matrix created by the toeplitz function in MATLAB =
         2           0          -1
         1           2           0
         0           1           2
        -1           0           1

The matrix created by Matrix::Toeplitz =
         2           0          -1
         1           2           0
         0           1           2
        -1           0           1

Press any key to continue ...
```

```
Case 2: When the number of columns is less than the number of rows
The 1st column =
         2
         0
        -1

The 1st row =
         2           1           0          -1

The matrix created by the toeplitz function in MATLAB =
         2           1           0          -1
         0           2           1           0
        -1           0           2           1

The matrix created by Matrix::Toeplitz =
         2           1           0          -1
         0           2           1           0
        -1           0           2           1

Press any key to continue ...
```

```
Case 3: When the 1st entries of the 1st column and row are different:
The 1st column =
         2
         1
         0
        -1

The 1st row =
         1            0           -1

The matrix created by the toeplitz function in MATLAB =
         2            0           -1
         1            2            0
         0            1            2
        -1            0            1

Warning: First element of input column does not match first element of input row
!
        Column wins the diagonal war.
The matrix created by Matrix::Toeplitz =
         2            0           -1
         1            2            0
         0            1            2
        -1            0            1
```

(4) Transpose function testing(Task3)

In “MatrixTest.cpp”, the transpose function has two additional featured functions compared to previous testings, which are static and non-static Transpose functions.

Step1: Define the headers of these Transpose functions in “Matrix.h”, and they are distinguished by keyword “static”.

```
57    static Matrix Transpose (const Matrix& mat_tr);//Static Transpose, which transposes the rows and columns from the input matrix mat_tr.
58    virtual Matrix& Transpose ();// non-static transpose,which transposes the rows and columns of the matrix itself. i.e. data.Transpose() transposes the rows and columns of the matrix data,return the resultant matrix as matrix.
```

Step2 and 3: Implement the static and non-static functions in “Matrix.cpp”, and display the result.

```
136//task4
137Matrix Matrix::Transpose (const Matrix& mat_tr) {
138    Matrix Transpose(mat_tr.noOfColumns, mat_tr.noOfRows);
139    for (int i = 0; i < mat_tr.noOfColumns; ++i) {
140        for (int j = 0; j < mat_tr.noOfRows; ++j) {
141        Transpose.data[Transpose.GetIndex(i,j)] = mat_tr.data[mat_tr.GetIndex(j,i)];
142        }
143    }
144    return Transpose;
145}// This static Transpose function takes a const Matrix mat_tr as a input matrix, create and return a Transposed Matrix with its number of rows equals to noOfColumns of mat_tr and the number of columns equals to noOfRows of mat_tr, where the element data at position of (jthrow, ithcolumn) of mat_tr are assigned to the element data at position of (ithrow, jthcolumn) of Transposed Matrix.
146
147Matrix& Matrix::Transpose(){
148
149    double temp[noOfRows][noOfColumns];
150    for (int i = 0; i < noOfRows; ++i) {
151        for (int j = 0; j < noOfColumns; ++j) {
152        temp[i][j] = data[GetIndex(i,j)];
153        }
154    }
155    // Store the data at position(i,j) in the Matrix itself to that at position[i][j] in a new defined double matrix temp.
156    int cols = noOfRows;
157    noOfRows = noOfColumns;
158    noOfColumns = cols;
159    // Swap the column and rows, as it is transposed
160    for (int i = 0; i < noOfRows; ++i) {
161        for (int j = 0; j < noOfColumns; ++j) {
162        data[GetIndex(i,j)] = temp[j][i];
163        }
164    }
165    return *this;
166    // assign the data at new transposed position(i,j) to that stored at position of (original row, original column) in temp, and return the transposed data as a Matrix
167}// Transpose the matrix itself and return the transposed version
```

```
Case 1: the static Transpose function
The original Matrix =
         2             0            -1
         1             2             0
         0             1             2
        -1             0             1

The transposed version =
         2             1             0            -1
         0             2             1             0
        -1             0             2             1

Press any key to continue ...
```

```
Case 2: the non-static Transpose function
The original Matrix =
         2             0            -1
         1             2             0
         0             1             2
        -1             0             1

The transposed version =
         2             1             0            -1
         0             2             1             0
        -1             0             2             1

Press any key to continue ...
```

(Code) (Result)

(5) Multiplication Function Testing(Task4)

Rather than previously defined functions, this session of “MatrixTest.cpp” mainly tests the non-member * and member *= operators functions defined in the Matrix class.

Step1: define the headers of * and *= operators in “Matrix.h”

```
63//task4
64// Matrix multiplication operation
65    Matrix& operator*= (const Matrix& rhs_pe);// Matrix multiplication, which executes matrix operation of  *=, e.g. a *= b
  is new matrix a = old matrix a * matrix b and returns new matrix a as the final output matrix. b is the input matrix rhs_pe.
66
67// friend grants open access to non-member functions
68    friend Matrix operator* (const Matrix& lhs_pe, const Matrix& rhs_pe);//friend grants open access to non-member
  functions, in which * operator performs matrix multiplication of Matrix lhs(left hand side) and Matrix rhs(right hand
  side).The const-ness of corresponding arguments(Matrix& rhs and Matrix& lhs) means the function promises not to change the
  arguments, whether or not you require that promises.
```

Step2 and 3: create functions for operators * and *= in “Matrix.cpp”. Display the result

```
Matrix operator* (const Matrix& lhs, const Matrix& rhs){
    if (lhs.noOfRows == rhs.noOfColumns||lhs.noOfColumns == rhs.noOfRows){
        Matrix multiple (lhs.noOfRows, rhs.noOfColumns);
        for (int i = 0; i < multiple.noOfRows; ++i){
            for (int j =0; j < multiple.noOfColumns; ++j){
                for(int k = 0; k < lhs.noOfColumns; ++k){
                    multiple.data[multiple.GetIndex(i,j)] += lhs.data[lhs.GetIndex(i,k)] * rhs.data[rhs.GetIndex(k,j)];
                    // the data at position of (ith row, jth column) in Matrix multiple equals to the sum of the product of
  the elements along the ith row in lhs and the elements along the jth column in rhs.
                }
            }
        }
    return multiple;
    }// Perform element-wise matrix multiplication, which only works when noOfRows of the matrix lhs in the left hand side
  of the operator = euquals to either noOfColumns or noOfRowsof the matrix rhs in the right hand side of the operator.
    else{
        cout << "lhs.noOfRows = " << lhs.noOfRows << endl;
        cout << "rhs.noOfColumns = " << rhs.noOfColumns << endl;
        cerr << " The number of rows in the lhs is supposed to be equal to either the number of columns or the number of
  rows in the rhs. " << endl;
        exit(1);
    }// otherwise, the matrix multiplication cannot be performed. Then, display the errors and exit the program.
}// perform the matrix multiplication lhs*rhs, and return the prodcut matrix which has the noOfRows rows of the lhs, and the
  noOfColumns columns of the rhs, as a Matrix type.

Matrix& Matrix::operator*= (const Matrix& rhs_me){
    if (noOfRows == rhs_me.noOfColumns){// Inner matrix dimension must agree
        Matrix product = (*this) * rhs_me;
        (*this) = product;
        return *this;
    }// (*this) simply means the matrix itself, in which this function is like performing A *= rhs_me, where A = A * rhs_me.
    else{
        cerr << " The number of rows in the lhs is supposed to be equal to the number of columns in the rhs. " << endl;
        exit(1);
    }//otherwise, display the error and exit the program
}
```

```
Testing the multiplication functions:
Case 1: the non-member multiplication function
The L.H.S. Matrix =
         2          0         -1
         1          2          0
         0          1          2
        -1          0          1

The R.H.S. version =
         2          1          0         -1
         0          2          1          0
        -1          0          2          1

The expected product =
         5          2         -2         -3
         2          5          2         -1
        -2          2          5          2
        -3         -1          2          2
The actual product =
         5          2         -2         -3
         2          5          2         -1
        -2          2          5          2
        -3         -1          2          2
```

```
Case 2: the member multiplication function
The L.H.S. Matrix =
         2          0         -1
         1          2          0
         0          1          2
        -1          0          1

The R.H.S. version =
         2          1          0         -1
         0          2          1          0
        -1          0          2          1

The expected product =
         5          2         -2         -3
         2          5          2         -1
        -2          2          5          2
        -3         -1          2          2
The actual product =
         5          2         -2         -3
         2          5          2         -1
        -2          2          5          2
        -3         -1          2          2
```

(6) Row or Column Exchange Function Testing(Task5)

This testing session of “MAtrixTest.cpp” mainly testes the exchange functions for cases shown in the result.

Step1: define the headers of the following 4 exchange functions for rows or columns in “Matrix.h”

```
    Matrix& ExchangeRows(const int rowIdx_1,const int rowIdx_2);// Exchange elements in rows specifed by rowIdx_1 and
  rowIdx_2,return the resultant matrix as matrix.

    Matrix& ExchangeRows(const int rowIdx_1,const int rowIdx_2,const int colIdx_1,const int colIdx_2);// Exchange elements
  in rows specifed by rowIdx_1 and rowIdx_2, and from columns specifed by colIdx_1 to that specified colIdx_2.

    //Exchange columns
    Matrix& ExchangeColumns(const int colIdx_1,const int colIdx_2);//Exchange elements in columns specifed by colIdx_1 and
  colIdx_2, return the resultant matrix as matrix.

    Matrix& ExchangeColumns(const int colIdx_1,const int colIdx_2,const int rowIdx_1,const int rowIdx_2);//Exchange elements
  in columns specifed by colIdx_1 and colIdx_2, from rows specified by rowIdx_1 to that specified by rowIdx_2.
```

Step2 and 3: Create the corresponding function in “Matrix.cpp”, display the result.

```
204Matrix& Matrix::ExchangeRows(const int rowIdx_1,const int rowIdx_2){
205    for (int j = 0; j < noOfColumns; ++j){
206            double tmp = data[GetIndex(rowIdx_1,j)];
207            data[GetIndex(rowIdx_1,j)] = data[GetIndex(rowIdx_2,j)];
208            data[GetIndex(rowIdx_2,j)] = tmp;
209    }
210    return *this;
211}// Exchange elements in rows specifed by rowIdx_1 and rowIdx_2,return the resultant matrix as Matrix.
212
213Matrix& Matrix::ExchangeRows(const int rowIdx_1,const int rowIdx_2,const int colIdx_1,const int colIdx_2){
214    for(int k = colIdx_1; k < colIdx_2+1; ++k){
215            double tmp = data[GetIndex(rowIdx_1,k)];
216            data[GetIndex(rowIdx_1,k)] = data[GetIndex(rowIdx_2,k)];
217            data[GetIndex(rowIdx_2,k)] = tmp;
218    }
219    return *this;
220}//Exchange elements in rows specifed by rowIdx_1 and rowIdx_2, and from columns specifed by colIdx_1 to that specified
   colIdx_2
221Matrix& Matrix::ExchangeColumns(const int colIdx_1,const int colIdx_2){
222    for (int i = 0; i < noOfRows; ++i){
223            double tmp = data[GetIndex(i,colIdx_1)];
224            data[GetIndex(i,colIdx_1)] = data[GetIndex(i,colIdx_2)];
225            data[GetIndex(i,colIdx_2)] = tmp;
226    }
227    return *this;
228}//Exchange elements in columns specifed by colIdx_1 and colIdx_2, return the resultant matrix as Matrix.
229
230Matrix& Matrix::ExchangeColumns(const int colIdx_1,const int colIdx_2,const int rowIdx_1,const int rowIdx_2){
231    for(int k = rowIdx_1; k < rowIdx_2+1; ++k){
232            double tmp = data[GetIndex(k,colIdx_1)];
233            data[GetIndex(k,colIdx_1)] = data[GetIndex(k,colIdx_2)];
234            data[GetIndex(k,colIdx_2)] = tmp;
235    }
236    return *this;
237}//Exchange elements in columns specifed by colIdx_1 and colIdx_2, from rows specified by rowIdx_1 to that specified by
   rowIdx_2.
```

```
Testing the Row or Column Exchange functions:
Case 1: the row exchange function
The original Matrix =
         2            0           -1
         1            2            0
         0            1            2
        -1            0            1

Exchange the 2nd and 4th rows gives
         2            0           -1
        -1            0            1
         0            1            2
         1            2            0

Exchange the 1st and 4th rows, but only from the 2nd to the 3rd columns, gives
         2            2            0
        -1            0            1
         0            1            2
         1            0           -1

Case 2: the column exchange function
The original Matrix =
         2            0           -1
         1            2            0
         0            1            2
        -1            0            1

Exchange the 1st and 2nd columns gives
         0            2           -1
         2            1            0
         1            0            2
         0           -1            1

Exchange the 2nd and 3rd columns, but only from the 2nd to the 4th rows, gives
         0            2           -1
         2            0            1
         1            2            0
         0            1           -1
```

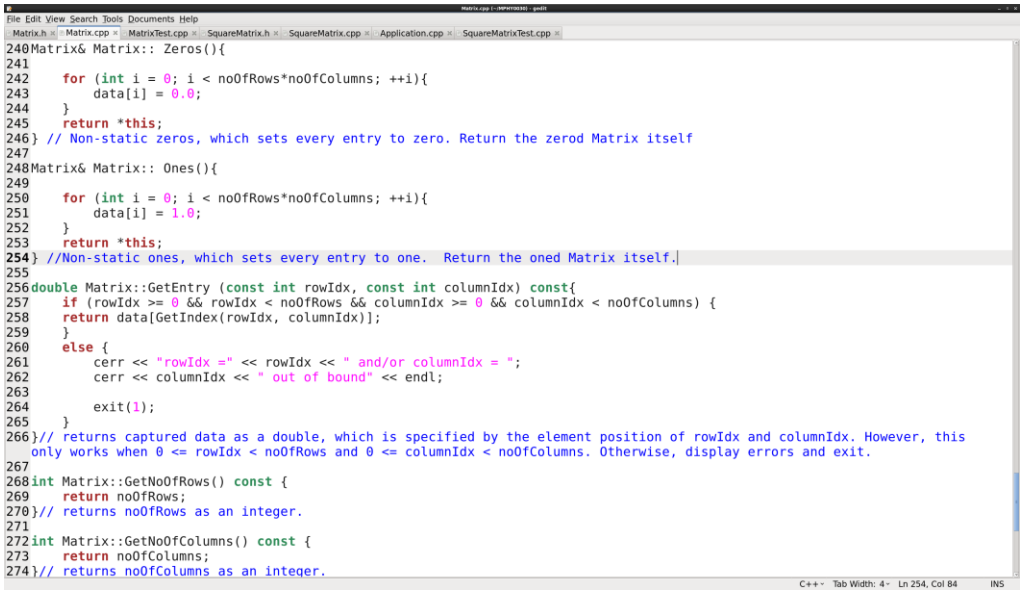
(7) Other Testing(Task6)

Rather than static functions, this time non-static Zeros and Ones are implemented and tested. Functions like GetNoOfRows, GetNoOfColumns and GetEntry are introduced to get the NoOfRows, NoOfColumns and specific element at a input position.

Step1: define the headers of the above functions in “Matrix.h”

```
79    // accessors, which returns the desired data in the desired data type.
80    int GetNoOfRows() const;// returns NoOfRows as an integer.
81
82    int GetNoOfColumns() const;// returns NoOfColumns as an integer.
83
84    double GetEntry(const int rowIdx, const int columnIdx) const;// returns captured data as a double, which is specified by
   the element position of rowIdx and columnIdx.
85
86    Matrix& Zeros ();// Non-static zeros, which sets every entry to zero.
87
88//virtual allows the child to be called, rather than calling the parents, decalre the function that needs polymorphisms as
   virtual
89    virtual Matrix& Ones ();//Non-static ones, which sets every entry to one. This function will be further used in
   SquareMatrix (child class), so use virtual to support polymorphisms.
```

Step2 and 3: create the corresponding functions in “Matrix.cpp”.



```
File Edit View Search Tools Documents Help
Matrix.h × Matrix.cpp × MatrixTest.cpp × SquareMatrix.h × SquareMatrix.cpp × Application.cpp × SquareMatrixTest.cpp ×
240 Matrix& Matrix:: Zeros(){
241
242     for (int i = 0; i < noOfRows*noOfColumns; ++i){
243         data[i] = 0.0;
244     }
245     return *this;
246 } // Non-static zeros, which sets every entry to zero. Return the zerod Matrix itself
247
248 Matrix& Matrix:: Ones(){
249
250     for (int i = 0; i < noOfRows*noOfColumns; ++i){
251         data[i] = 1.0;
252     }
253     return *this;
254 } //Non-static ones, which sets every entry to one.  Return the oned Matrix itself.
255
256 double Matrix::GetEntry (const int rowIdx, const int columnIdx) const{
257     if (rowIdx >= 0 && rowIdx < noOfRows && columnIdx >= 0 && columnIdx < noOfColumns) {
258     return data[GetIndex(rowIdx, columnIdx)];
259     }
260     else {
261         cerr << "rowIdx =" << rowIdx << " and/or columnIdx = ";
262         cerr << columnIdx << " out of bound" << endl;
263
264         exit(1);
265     }
266 }// returns captured data as a double, which is specified by the element position of rowIdx and columnIdx. However, this
    only works when 0 <= rowIdx < noOfRows and 0 <= columnIdx < noOfColumns. Otherwise, display errors and exit.
267
268 int Matrix::GetNoOfRows() const {
269     return noOfRows;
270 }// returns noOfRows as an integer.
271
272 int Matrix::GetNoOfColumns() const {
273     return noOfColumns;
274 }// returns noOfColumns as an integer.
C++ Tab Width: 4 Ln 254, Col 84 INS
```

```
Testing the miscellaneous functions:
The function that sets every entry to zero
The input matrix =
         1            1            1            1
         1            1            1            1

After setting every entry to zero, we have
         0            0            0            0
         0            0            0            0

Press any key to continue ...

The function that sets every entry to one
The input matrix =
         0            0            0            0
         0            0            0            0

After setting every entry to one, we have
         1            1            1            1
         1            1            1            1

Press any key to continue ...

Find out the number of rows and columns
The input matrix =
         0            0            0            0
         0            0            0            0

The number of rows = 2 ; the number of columns = 4

Press any key to continue ...

Find out the value of a particular entry
The original Matrix =
         2            0           -1
         1            2            0
         0            1            2
        -1            0            1

The entry at the 2nd row and the 3rd column is 0
The entry at the 3rd row and the 2nd column is 1
```

1.3 Implementation of Assignment Testing(Task 7)

To investigate how the assignment operator changes the size of matrix in the left hand side(lhs) of =, with the size of matrix in the right hand side(rhs) of =, three cases were considered and implemented in the “MatrixTest.cpp”(shown in the following figures). From the result, it has been shown that the size of lhs matrix after assignment operation is determined by the size of rhs matrix. The variance in size of lhs matrix will not affect the final matrix size in rhs.

```
57 void AssignmentTesting () {
58     cout << "Testing the assignment operator:" << endl;
59     cout << endl;
60     cout << "Case 1: when the size of the LHS matrix is larger than that of the RHS matrix" << endl;
61     {
62         // the same matrix as in ToeplitzTesting
63         double column_l[2] = {1, 2};
64         double row_l[3] = {3, 0, -1};
65         double column_r[2] = {1, 2};
66         double row_r[2] = {3, 0};
67         Matrix lhs = Matrix::Toeplitz(column_l, 2, row_l, 3);
68         cout << "The L.H.S. Matrix = " << endl;
69         cout << lhs << endl;
70         Matrix rhs = Matrix::Toeplitz(column_r, 2, row_r, 2);
71         cout << "The R.H.S. version = " << endl;
72         cout << rhs << endl;
73         cout << "The New LHS matrix = " << endl;
74         //  1     0
75         //  2     1
76         lhs = rhs;
77         cout << lhs << endl;
78         cout << "Press any key to continue ..." << flush;
79         system("read");
80         cout << endl;
81     }
```

```
Testing the assignment operator:

Case 1: when the size of the LHS matrix is larger than that of the RHS matrix
Warning: First element of input column does not match first element of input row!
        Column wins the diagonal war.
The L.H.S. Matrix =
         1            0           -1
         2            1            0

The R.H.S. version =
         1            0
         2            1

The New LHS matrix =
         1            0
         2            1

Press any key to continue ...
```

```
83     cout << "Case 2: when the size of the LHS matrix is smaller than that of the RHS matrix" << endl;
84         {
85         // the same matrix as in ToeplitzTesting
86         double column_l2[2] = {1, 2};
87         double row_l2[2] = {3, 0};
88         double column_r2[2] = {1, 2};
89         double row_r2[3] = {3, 0, 4};
90         Matrix lhs = Matrix::Toeplitz(column_l2, 2, row_l2, 2);
91         cout << "The L.H.S. Matrix = " << endl;
92         cout << lhs << endl;
93         Matrix rhs = Matrix::Toeplitz(column_r2, 2, row_r2, 3);
94         cout << "The R.H.S. version = " << endl;
95         cout << rhs << endl;
96         //  1     0     4
97         //  2     1     0
98         cout << "The New LHS matrix =  " << endl;
99         lhs = rhs;
100        cout << lhs << endl;
101        cout << "Press any key to continue ..." << flush;
102        system("read");
103        cout << endl;
104    }
```

```
Case 2: when the size of the LHS matrix is smaller than that of the RHS matrix
The L.H.S. Matrix =
         1            0
         2            1

Warning: First element of input column does not match first element of input row!
        Column wins the diagonal war.
The R.H.S. version =
         1            0            4
         2            1            0

The New LHS matrix =
         1            0            4
         2            1            0

Press any key to continue ...
```

```
106     cout << "Case 3: when the size of the LHS matrix is equal to that of the RHS matrix" << endl;
107         {
108         // the same matrix as in ToeplitzTesting
109         double column_l3[2] = {2, 3};
110         double row_l3[2] = {4, 0};
111         double column_r3[2] = {1, 5};
112         double row_r3[2] = {0, 4};
113         Matrix lhs = Matrix::Toeplitz(column_l3, 2, row_l3, 2);
114         cout << "The L.H.S. Matrix = " << endl;
115         cout << lhs << endl;
116         Matrix rhs = Matrix::Toeplitz(column_r3, 2, row_r3, 2);
117         cout << "The R.H.S. version = " << endl;
118         cout << rhs << endl;
119         //  1     0
120         //  5     4
121         cout << "The New LHS matrix =  " << endl;
122         lhs = rhs;
123         cout << lhs << endl;
124         cout << "Press any key to continue ..." << flush;
125         system("read");
126         cout << endl;
127     }
```

```
Case 3: when the size of the LHS matrix is equal to that of the RHS matrix
The L.H.S. Matrix =
          2             0
          3             2

The R.H.S. version =
          1             4
          5             1

The New LHS matrix =
          1             4
          5             1
```

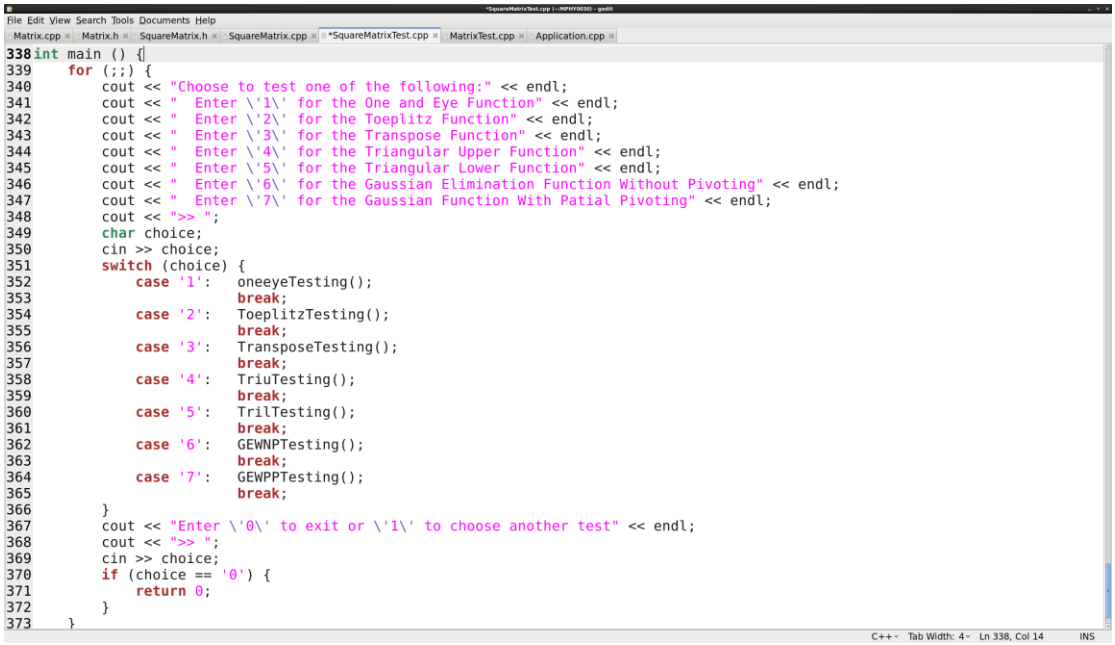
Part2

Introduction

A Matlab script file example.m was provided, and the aim in part 2 is to develop a set of functions that can reproduce the individual tasks carried out in the script. The below are the selection interface implemented using SquareMatrix class for different testing required in Matlab and its codes in the test file.

```
[mphy0030@gzhang MPHY0030]$ ./SquareMatrixTest
Choose to test one of the following:
  Enter '1' for the One and Eye Function
  Enter '2' for the Toeplitz Function
  Enter '3' for the Transpose Function
  Enter '4' for the Triangular Upper Function
  Enter '5' for the Triangular Lower Function
  Enter '6' for the Gaussian Elimination Function Without Pivoting
  Enter '7' for the Gaussian Function With Patial Pivoting
```

(interface)




```
338 int main () {
339     for (;;) {
340         cout << "Choose to test one of the following:" << endl;
341         cout << "  Enter \'1\' for the One and Eye Function" << endl;
342         cout << "  Enter \'2\' for the Toeplitz Function" << endl;
343         cout << "  Enter \'3\' for the Transpose Function" << endl;
344         cout << "  Enter \'4\' for the Triangular Upper Function" << endl;
345         cout << "  Enter \'5\' for the Triangular Lower Function" << endl;
346         cout << "  Enter \'6\' for the Gaussian Elimination Function Without Pivoting" << endl;
347         cout << "  Enter \'7\' for the Gaussian Function With Patial Pivoting" << endl;
348         cout << ">> ";
349         char choice;
350         cin >> choice;
351         switch (choice) {
352             case '1':   oneeyeTesting();
353                         break;
354             case '2':   ToeplitzTesting();
355                         break;
356             case '3':   TransposeTesting();
357                         break;
358             case '4':   TriuTesting();
359                         break;
360             case '5':   TrilTesting();
361                         break;
362             case '6':   GEWNPTesting();
363                         break;
364             case '7':   GEWPPTesting();
365                         break;
366         }
367         cout << "Enter \'0\' to exit or \'1\' to choose another test" << endl;
368         cout << ">> ";
369         cin >> choice;
370         if (choice == '0') {
371             return 0;
372         }
373     }
```

(Codes)

Method

2.1 Macroscopic structure of the class “SquareMatrix”

The construction of the macroscopic structure is pretty similar to one created for the class “Matrix”. The class “SquareMatrix” is a child class of the class “Matrix”, so no additional member variables are defined and “Matrix.h” is included. Same general rules as those in the parent class apply to the prototypes in the child class. The only difference is that those functions starting with prototypes “static” are specific to the child class and some prototypes like friend and virtual are not needed. The child class itself inherits some public methods from its parent.

```
#ifndef SQUAREMATRIX_H_
#define SQUAREMATRIX_H_

#include <iostream>//choose the library that we want to use
#include "Matrix.h"// To inherit methods from the class Matrix, "Matrix.h" is therefore included
using namespace std;//it declares that the program will be assessing entities who are part of the namespace called "std."

    // General rule for const: the const-ness of the corresponding arguments in all the functions means the function promises not to change the argument, whether or not you require that promises
    // General rule for using double: double can store both integer and non-integer numbers and the storing capacity of double outperforms other data types like int. And it is particularly useful for address of an array like *data.
    // General rule for using int: good for integer-based index data, like row and column etc.
    // General rule of &: & has the use of dereferencing the address stored in the input, which returns the output as visualised matrix elements(number).
    //General rule of *, which is known as pointer which can store data as address. & can dereference those data stored by the pointer.
    // General rule of using static: objects declared as static have a scope till the lifetime of program.

// Description:
// SquareMatrix: A class for modelling dim-by-dim  square matrices, and it is also a child class inherits public methods from class Matrix
class SquareMatrix : public Matrix {

    protected://No additional class member variables are defined.

    public:

//These are accessible methods! Public grants their accessibility.
```

```
};

#endif /* SQUAREMATRIX_H_ */
```

C/C++/ObjC Header · Tab Width: 4 · Ln 92, Col 3 INS

2.2 Implementation of individual tasks carried out in Matlab

In the aim stated at the very beginning of part 2, implementation of functionalities that can reproduced in the Matlab script are required.

(1) Oneeyetesting (task1)

The 1st part of the Matlab codes require to reproduce ones and eye functions by taking dim as a input.

```
dim = 4;
matrix1 = ones(dim)
matrix2 = eye(dim)
```

(Matlab code)

Therefore, the first part in ”SquareMatrixTest.cpp” is to start with creating a square matrix taking one input (dim), then achieve the testing for ones and eye functions by creating SquareMatrices which takes the same input. In this case, dim = 4.

```cpp
void oneeyeTesting () {
    cout << "Testing the Square Matrix standard constructor:" << endl;
    cout << "\n";
    cout << "Case 1: Creating a 4 by 4 square matrix of Zeros with the Sqaure Matrix standard constructor" << endl;
    cout << "\n";
    {
        SquareMatrix Sqmatrix(4);
        cout << "The square matrix created by the standard construtor: " << endl;
        cout << Sqmatrix << endl;
        cout << "Press any key to continue ..." << flush;
        system("read");
        cout << endl;
    }

    cout << "Case 2: Creating a 4 by 4 square matrix with Ones:" << endl;
    {
        SquareMatrix onesmatrix(4);
        cout << onesmatrix.ones() << endl;
        cout << "Press any key to continue ..." << flush;
        system("read");
        cout << endl;
    }

    cout << "Case 3: Creating a 4 by 4 identity square matrix with Ones on the main diagonal and zeros elsewhere:" << endl;
    {
        SquareMatrix eyematrix = SquareMatrix::eye(4);
        cout << eyematrix << endl;
        cout << "Press any key to continue ..." << flush;
        system("read");
        cout << endl;
    }
}
```

(Testfile codes)

It can be seen that standard constructor, copy constructor, assignment constructor from SquareMatrix to SquareMatrix, destructor(inherited from Matrix class), non-static ones and static eye are required. Hence, create their corresponding headers and codes.

```cpp
//task1
    SquareMatrix (const int dim); //Default constructor (M1), which constructs a standard square matrix of values with dim rows and dim columns. The return type of the output has to be a SqaureMatrix.
// static member functions for named constructors, objects declared as static have a scope till the lifetime of program.
    static SquareMatrix eye(const int dim);//which returns a Squarematrix of dim dimensions with its diagonal filled with 1, and zeros elsewhere. Due to it's SquareMatrix's own feature, so here a prototype "static" is used.
    SquareMatrix& ones(); //Non-static ones, which sets every entry to one. This function is inherited from the parent class (Matrix).
    ~SquareMatrix ();// destructor, which frees the memory address stored occcupied by the dynamical array.

    SquareMatrix (const SquareMatrix& mat); //Copy constructor (M2), which stores data from the input SquareMatrix (mat). const ensures that the argument will not be changed by the function. & has the use of dereferencing the address stored in mat, which returns the output as visualised SquareMatrix elements.
    SquareMatrix& operator = (const SquareMatrix& mat);// Assignment constructor(M3), which assigns the value of the right hand side input SquareMatrix(mat) to the left hand side SquareMatrix. As = is introduced for this constructor, so word operator is needed. & has the use of dereferencing the address stored in both lhs and rhs SquareMatrix, which returns the output as visualised SquareMatrix elements.
```

(Headers)

Matrix.cpp | Matrix.h | SquareMatrix.h | SquareMatrix.cpp | SquareMatrixTest.cpp | MatrixTest.cpp | Application.cpp

```cpp

#include "SquareMatrix.h"
#include <cstdlib>
#include <cmath>

using namespace std;
//Task 1
SquareMatrix::SquareMatrix (const int dim): Matrix(dim,dim){}//Default constructor (M1), which constructs a standard square matrix of 0 with dim rows and dim columns. The return type of the output has to be a SqaureMatrix. Due to the difference of SquareMatrix only takes one input const integer and its return type, hence we have to declare SquareMatrix with its own Default constructor to ensure right type of return.

SquareMatrix& SquareMatrix::ones(){
    this -> Matrix::Ones();
    return *this;
}//Non-static ones, which sets every entry to one. This function is inherited from the parent class(Matrix& Matrix::ones()).Due to the difference of return type, hence we have to declare SquareMatrix with its own ones function to ensure right type of return.

SquareMatrix::~SquareMatrix (){}// destructor, which frees the memory address stored occcupied by the dynamical array. It is a polymorphism of the destrutor in the parent class Matrix.

SquareMatrix SquareMatrix::eye (const int dim){
    SquareMatrix matrix(dim);
    for (int i =0; i < dim; i++){
        matrix.data[matrix.GetIndex(i,i)] = 1.0;
    }
    return matrix;
}//This returns a Squarematrix of dim dimensions with its diagonal filled with 1, and zeros elsewhere.


SquareMatrix::SquareMatrix (const SquareMatrix& mat): Matrix(mat){}//Copy constructor (M2), which stores data from the input SquareMatrix (mat). const ensures that the argument will not be changed by the function. & has the use of dereferencing the address stored in mat, which returns the output as visualised SquareMatrix elements.Due to the difference of input type and return type, hence we have to declare SquareMatrix with its own copy constructors to ensure right type of return.
```

```cpp
SquareMatrix& SquareMatrix::operator = (const SquareMatrix& mat){
    this -> Matrix::operator = (mat);
    return *this;
}// Assignment constructor(M3), which assigns the value of the right hand side input SquareMatrix(mat) to the left hand side SquareMatrix. This constructor inherits from the assignment operator(M3) in the parent class, and the only difference is the return type and the input type.(Both are SquareMatrix format.) Due to these differences, hence we have to declare SquareMatrix with its own assignment constructors to ensure right type of return and input.
```

C++ Tab Width: 4 Ln 413, Col 439 INS

(CPP codes)

Conclusion from the results: the results in c++ matches that in Matlab.

```
Testing the Square Matrix standard constructor:

Case 1: Creating a 4 by 4 square matrix of Zeros with the Sqaure Matrix standard
 constructor

The square matrix created by the standard construtor:
        0           0           0           0
        0           0           0           0
        0           0           0           0
        0           0           0           0

Press any key to continue ...

Case 2: Creating a 4 by 4 square matrix with Ones:
        1           1           1           1
        1           1           1           1
        1           1           1           1
        1           1           1           1

Press any key to continue ...

Case 3: Creating a 4 by 4 identity square matrix with Ones on the main diagonal
and zeros elsewhere:
        1           0           0           0
        0           1           0           0
        0           0           1           0
        0           0           0           1

Press any key to continue ...

Enter '0' to exit or '1' to choose another test
```

(C++ Task1 Result)

```
matrix1 =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1


matrix2 =

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

(Matlab Task 1 Result)

(2) ToeplitzTesting (task2)

The second task is to create Toeplitz SquareMatrices by inputting single vector or double input vectors with same dimensions.

```
vec1 = [4, 3, 2, 1];
vec2 = [1, 2, 3, 4];
matrix3 = toeplitz(vec1)
matrix4 = toeplitz(vec2, vec1)
```

(Matlab Code)

Hence, create a test file in “SquareMatrixTest.cpp” considering the above two cases.

```
Matrix.cpp | Matrix.h | SquareMatrix.h | SquareMatrix.cpp | SquareMatrixTest.cpp | MatrixTest.cpp | Application.cpp
69 void ToeplitzTesting () {
70     cout << "Testing the static function SquareMatrix::toeplitz:" << endl;
71     cout << "\n";
72     cout << "Case 1: Creating 4 by 4 square matrix with single input argument of an array" << endl;
73     {
74         double vec2[4] = {4, 3, 2, 1};
75         bool inside = true;
76         // matrix should be stored, in 1-D, column-wise
77         //  4     3     2     1
78         //  3     4     3     2
79         //  2     3     4     3
80         //  1     2     3     4
81         double expected[16] = {4, 3, 2, 1, 3, 4, 3, 2, 2, 3, 4, 3, 1, 2, 3, 4};
82         ToeplitzTestingHelper(vec2, vec2, 4, expected,inside);
83         cout << "Press any key to continue ..." << flush;
84         system("read");
85         cout << endl;
86     }
87
88     cout << "Case 2: Creating 4 by 4 square matrix with double inputs arguments of arrays with the same dimension" << endl;
89     {
90         double vec1[4] = {1, 2, 3, 4};
91         double vec2[4] = {4, 3, 2, 1};
92         bool inside = true;
93         // matrix should be stored, in 1-D, column-wise
94         //  1     3     2     1
95         //  2     1     3     2
96         //  3     2     1     3
97         //  4     3     2     1
98         double expected[16] = {1, 2, 3, 4, 3, 1, 2, 3, 2, 3, 1, 2, 1, 2, 3, 1};
99         ToeplitzTestingHelper(vec1,vec2,4, expected,inside);
00         cout << "Press any key to continue ..." << flush;
01         system("read");
02         cout << endl;
03     }
04 }
```

(C++ Test file codes: main part of Toeplitz testing)

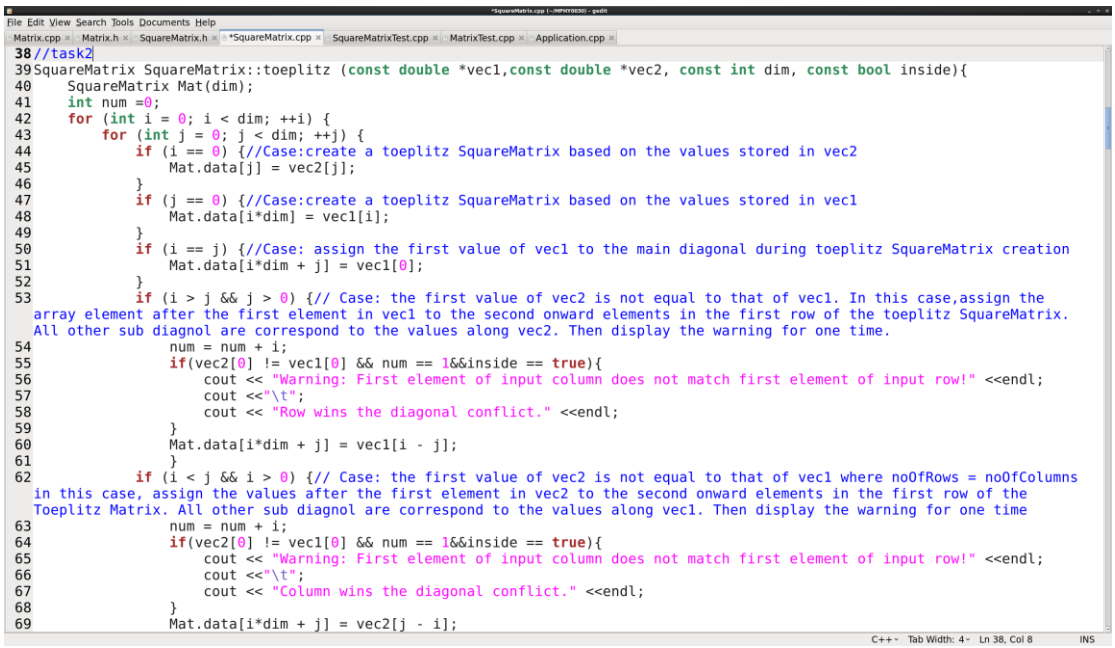
```
47 void ToeplitzTestingHelper (const double *vec1,const double *vec2,const int dim, const double *expected, const bool inside) {
48     cout << "The 1st vector = " << endl;
49     Matrix::Print(vec1, 1, dim);
50     cout << endl;
51     cout << "The 2nd vector = " << endl;
52     Matrix::Print(vec2, 1, dim);
53     cout << endl;
54     cout << "The matrix created by the toeplitz function in MATLAB = " << endl;
55     Matrix::Print(expected, dim, dim);
56     cout << endl;
57     //Case2: Double input vectors
58     if (vec1 != vec2){
59         SquareMatrix Toeplitz = SquareMatrix::toeplitz(vec1,vec2,dim,inside);
60         cout << "The matrix created by SquareMatrix::toeplitz = " << endl;
61         cout << Toeplitz << endl;
62     }
63     //Case1: Single input vector
64     else{
65         SquareMatrix Toeplitz = SquareMatrix::toeplitz(vec1,dim,inside);
66         cout << "The matrix created by SquareMatrix::toeplitz = " << endl;
67         cout << Toeplitz << endl;
68     }
69 }
```

(C++ Test file codes: function used in Toeplitz testing)

In this testing, two types of toeplitz functions with different input formats are needed. All the previously defined constructors and destructors are also employed in this testing.

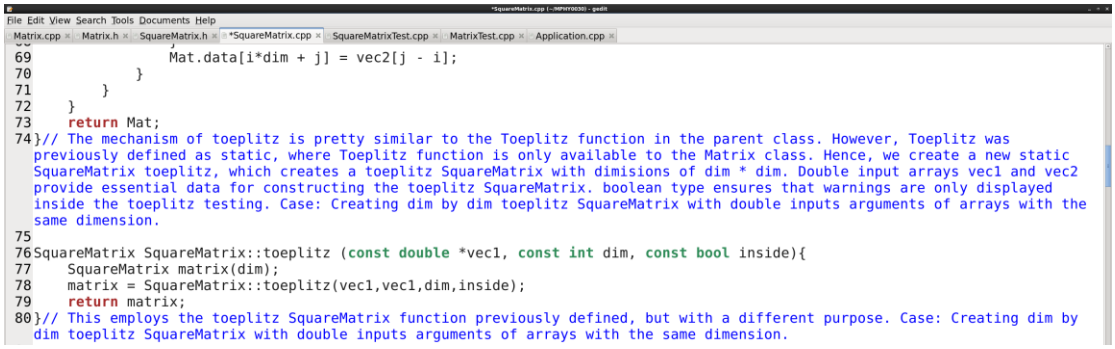
```
44 //task2
45     static SquareMatrix toeplitz(const double *vec1,const int dim, const bool inside);//Static toeplitz, which returns a
   toeplitz Squarematrix with dim rows and dim columns, based on the values stored in *vec1 (input vector). bool inside is used
   to justify whether this function is inside the toeplitz testing function in "SquareMatrixTesting.cpp", true means inside and
   false means outside.
46
47     static SquareMatrix toeplitz(const double *vec2, const double *vec1,const int dim, const bool inside);//Static toeplitz,
   which returns a toeplitz Squarematrix with dim rows and dim columns, based on the values stored in *vec1 (input vector1) and
   *vec2 (input vector2). bool inside is used to justify whether this function is inside the toeplitz testing function in
   "SquareMatrixTesting.cpp", true means inside and false means outside.
```

(Headers for task2)



```
38 //task2
39 SquareMatrix SquareMatrix::toeplitz (const double *vec1,const double *vec2, const int dim, const bool inside){
40     SquareMatrix Mat(dim);
41     int num =0;
42     for (int i = 0; i < dim; ++i) {
43         for (int j = 0; j < dim; ++j) {
44             if (i == 0) {//Case:create a toeplitz SquareMatrix based on the values stored in vec2
45                 Mat.data[j] = vec2[j];
46             }
47             if (j == 0) {//Case:create a toeplitz SquareMatrix based on the values stored in vec1
48                 Mat.data[i*dim] = vec1[i];
49             }
50             if (i == j) {//Case: assign the first value of vec1 to the main diagonal during toeplitz SquareMatrix creation
51                 Mat.data[i*dim + j] = vec1[0];
52             }
53             if (i > j && j > 0) {// Case: the first value of vec2 is not equal to that of vec1. In this case,assign the
   array element after the first element in vec1 to the second onward elements in the first row of the toeplitz SquareMatrix.
   All other sub diagnol are correspond to the values along vec2. Then display the warning for one time.
54                 num = num + i;
55                 if(vec2[0] != vec1[0] && num == 1&&inside == true){
56                     cout << "Warning: First element of input column does not match first element of input row!" <<endl;
57                     cout <<"\t";
58                     cout << "Row wins the diagonal conflict." <<endl;
59                 }
60                 Mat.data[i*dim + j] = vec1[i - j];
61                 }
62             if (i < j && i > 0) {// Case: the first value of vec2 is not equal to that of vec1 where noOfRows = noOfColumns
   in this case, assign the values after the first element in vec2 to the second onward elements in the first row of the
   Toeplitz Matrix. All other sub diagnol are correspond to the values along vec1. Then display the warning for one time
63                 num = num + i;
64                 if(vec2[0] != vec1[0] && num == 1&&inside == true){
65                     cout << "Warning: First element of input column does not match first element of input row!" <<endl;
66                     cout <<"\t";
67                     cout << "Column wins the diagonal conflict." <<endl;
68                 }
69                 Mat.data[i*dim + j] = vec2[j - i];
```

(Class Codes for task2)



```
69                 Mat.data[i*dim + j] = vec2[j - i];
70             }
71         }
72     }
73     return Mat;
74 }// The mechanism of toeplitz is pretty similar to the Toeplitz function in the parent class. However, Toeplitz was
   previously defined as static, where Toeplitz function is only available to the Matrix class. Hence, we create a new static
   SquareMatrix toeplitz, which creates a toeplitz SquareMatrix with dimisions of dim * dim. Double input arrays vec1 and vec2
   provide essential data for constructing the toeplitz SquareMatrix. boolean type ensures that warnings are only displayed
   inside the toeplitz testing. Case: Creating dim by dim toeplitz SquareMatrix with double inputs arguments of arrays with the
   same dimension.
75
76 SquareMatrix SquareMatrix::toeplitz (const double *vec1, const int dim, const bool inside){
77     SquareMatrix matrix(dim);
78     matrix = SquareMatrix::toeplitz(vec1,vec1,dim,inside);
79     return matrix;
80 }// This employs the toeplitz SquareMatrix function previously defined, but with a different purpose. Case: Creating dim by
   dim toeplitz SquareMatrix with double inputs arguments of arrays with the same dimension.
```

(Class Codes for task2)

Conclusion: The results match.

```
Case 1: Creating 4 by 4 square matrix with single input argument of an array
The 1st vector =
         4           3           2           1

The 2nd vector =
         4           3           2           1

The matrix created by the toeplitz function in MATLAB =
         4           3           2           1
         3           4           3           2
         2           3           4           3
         1           2           3           4

The matrix created by SquareMatrix::toeplitz =
         4           3           2           1
         3           4           3           2
         2           3           4           3
         1           2           3           4

Press any key to continue ...

Case 2: Creating 4 by 4 square matrix with double inputs arguments of arrays with the same dimension
The 1st vector =
         1           2           3           4

The 2nd vector =
         4           3           2           1

The matrix created by the toeplitz function in MATLAB =
         1           3           2           1
         2           1           3           2
         3           2           1           3
         4           3           2           1

Warning: First element of input column does not match first element of input row!
        Column wins the diagonal conflict.
The matrix created by SquareMatrix::toeplitz =
         1           3           2           1
         2           1           3           2
         3           2           1           3
         4           3           2           1

Press any key to continue ...
```

(Task2 C++ result)

```
matrix3 =

     4     3     2     1
     3     4     3     2
     2     3     4     3
     1     2     3     4

Warning: First element of input column does not match first
element of input row.
        Column wins diagonal conflict.
> In toeplitz (line 31)
  In example (line 11)

matrix4 =

     1     3     2     1
     2     1     3     2
     3     2     1     3
     4     3     2     1
```

(Task2 Matlab result)

(3) Transpose Testing (task3)

The transpose testing is based on the result from the Toeplitz function using double input vectors.

```
vec1 = [4, 3, 2, 1];
vec2 = [1, 2, 3, 4];
matrix3 = toeplitz(vec1)
matrix4 = toeplitz(vec2, vec1)

matrix5 = transpose(matrix4)
```

(Matlab codes)

Hence, create a test file in “SquareMatrixTest.cpp” considering the above case.

```
void TransposeTesting () {
    cout << "Testing the Transpose functions:" << endl;
    cout << "\n";
    cout << "Case 1: the non-static Transpose function" << endl;
    {
        // the same matrix as in ToeplitzTesting
        double vec1[4] = {1, 2, 3, 4};
        double vec2[4] = {4, 3, 2, 1};
        double expected[16] = {1, 2, 3, 4, 3, 1, 2, 3, 2, 3, 1, 2, 1, 2, 3, 1};
        bool inside = false;
        SquareMatrix matrix = SquareMatrix::toeplitz(vec1,vec2,4,inside);
        cout << "The original Matrix = " << endl;
        cout << matrix << endl;
        matrix.transpose();
        cout << "The transposed version = " << endl;
        cout << matrix << endl;
        cout << "Press any key to continue ..." << flush;
        system("read");
        cout << endl;
    }
}
```

(C++ test file codes for task3)

Implement a non-static transpose function in Class SquareMatrix, which is inherited from its parent class (that with prototype virtual to support polymorphism.

```
//task3
    SquareMatrix& transpose();// non-static transpose,which transposes the rows and columns of the SquareMatrix itself. i.e.
  data.Transpose() transposes the rows and columns of the SquareMatrix data,return the resultant SquareMatrix as SquareMatrix.
```

(Headers for task3)

```
SquareMatrix& SquareMatrix::transpose(){
    this -> Matrix::Transpose();
    return *this;
}// non-static transpose,which transposes the rows and columns of the SquareMatrix itself. i.e. data.Transpose() transposes
  the rows and columns of the SquareMatrix data,return the resultant SquareMatrix as SquareMatrix.This function is inherited
  from the parent class(Matrix& Matrix::Transpose()).Due to the difference of return type, hence we have to declare
  SquareMatrix with its own transpose function to ensure right type of return.
```

(C++ class codes for task3)

Conclusion: The results match.

```
Testing the Transpose functions:

Case 1: the non-static Transpose function
The original Matrix =
        1           3           2           1
        2           1           3           2
        3           2           1           3
        4           3           2           1

The transposed version =
        1           2           3           4
        3           1           2           3
        2           3           1           2
        1           2           3           1

Press any key to continue ...

Enter '0' to exit or '1' to choose another test
>>
```

```
matrix4 =

     1     3     2     1
     2     1     3     2
     3     2     1     3
     4     3     2     1


matrix5 =

     1     2     3     4
     3     1     2     3
     2     3     1     2
     1     2     3     1
```

(Task 3 C++ result) (Task 3 Matlab result)

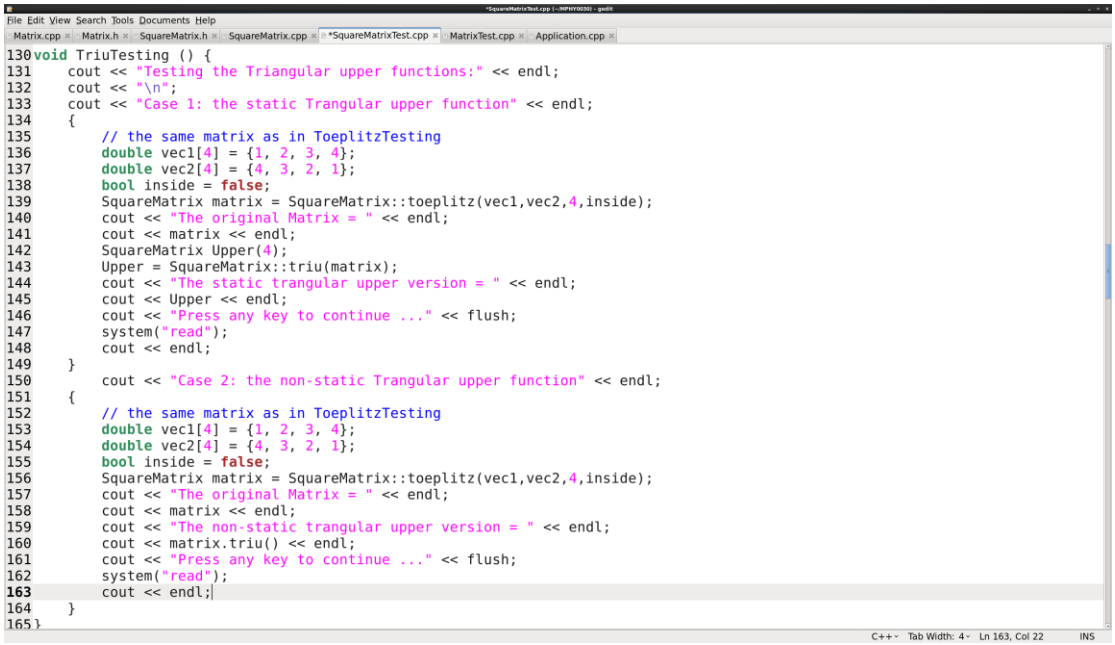
(4) Triangular upper and lower Function Testings(Task 4)

These functions return lower and upper parts of an input SquareMatrix. The triu testing and the tril testing are based on the result from the Toeplitz function using double input vectors shown in Matlab. And both static and non-static cases are considered for these testings.

```
matrix6 = triu(matrix4)
```

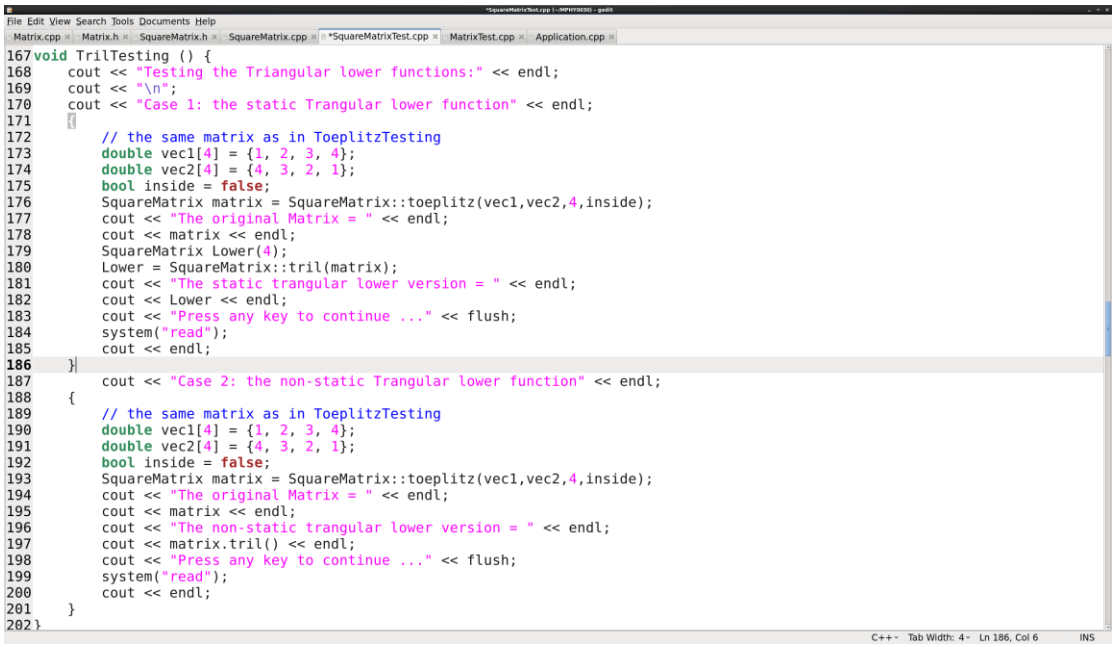
```
matrix7 = tril(matrix4)
```

(Matlab code task4)



```
130 void TriuTesting () {
131     cout << "Testing the Triangular upper functions:" << endl;
132     cout << "\n";
133     cout << "Case 1: the static Trangular upper function" << endl;
134     {
135         // the same matrix as in ToeplitzTesting
136         double vec1[4] = {1, 2, 3, 4};
137         double vec2[4] = {4, 3, 2, 1};
138         bool inside = false;
139         SquareMatrix matrix = SquareMatrix::toeplitz(vec1,vec2,4,inside);
140         cout << "The original Matrix = " << endl;
141         cout << matrix << endl;
142         SquareMatrix Upper(4);
143         Upper = SquareMatrix::triu(matrix);
144         cout << "The static trangular upper version = " << endl;
145         cout << Upper << endl;
146         cout << "Press any key to continue ..." << flush;
147         system("read");
148         cout << endl;
149     }
150         cout << "Case 2: the non-static Trangular upper function" << endl;
151     {
152         // the same matrix as in ToeplitzTesting
153         double vec1[4] = {1, 2, 3, 4};
154         double vec2[4] = {4, 3, 2, 1};
155         bool inside = false;
156         SquareMatrix matrix = SquareMatrix::toeplitz(vec1,vec2,4,inside);
157         cout << "The original Matrix = " << endl;
158         cout << matrix << endl;
159         cout << "The non-static trangular upper version = " << endl;
160         cout << matrix.triu() << endl;
161         cout << "Press any key to continue ..." << flush;
162         system("read");
163         cout << endl;
164     }
165 }
```

(Task4: Triangular upper function testing)



```
167 void TrilTesting () {
168     cout << "Testing the Triangular lower functions:" << endl;
169     cout << "\n";
170     cout << "Case 1: the static Trangular lower function" << endl;
171     {
172         // the same matrix as in ToeplitzTesting
173         double vec1[4] = {1, 2, 3, 4};
174         double vec2[4] = {4, 3, 2, 1};
175         bool inside = false;
176         SquareMatrix matrix = SquareMatrix::toeplitz(vec1,vec2,4,inside);
177         cout << "The original Matrix = " << endl;
178         cout << matrix << endl;
179         SquareMatrix Lower(4);
180         Lower = SquareMatrix::tril(matrix);
181         cout << "The static trangular lower version = " << endl;
182         cout << Lower << endl;
183         cout << "Press any key to continue ..." << flush;
184         system("read");
185         cout << endl;
186     }
187         cout << "Case 2: the non-static Trangular lower function" << endl;
188     {
189         // the same matrix as in ToeplitzTesting
190         double vec1[4] = {1, 2, 3, 4};
191         double vec2[4] = {4, 3, 2, 1};
192         bool inside = false;
193         SquareMatrix matrix = SquareMatrix::toeplitz(vec1,vec2,4,inside);
194         cout << "The original Matrix = " << endl;
195         cout << matrix << endl;
196         cout << "The non-static trangular lower version = " << endl;
197         cout << matrix.tril() << endl;
198         cout << "Press any key to continue ..." << flush;
199         system("read");
200         cout << endl;
201     }
202 }
```

(Task4: Triangular lower function testing)

Create the corresponding headers and class codes.

```
52     static SquareMatrix tril(const SquareMatrix& mat_tri);// returns the lower triangular part of the input SquareMatrix
   mat_tri
53
54     static SquareMatrix triu(const SquareMatrix& mat_tri);// returns the upper triangular part of the input SquareMatrix
   mat_tri
55
56     SquareMatrix& triu();// non-static triu, which returns the upper triangular part of the SquareMatrix itself
57
58     SquareMatrix& tril();// non-static tril, which returns the lower triangular part of the SquareMatrix itself
```

(Headers for task 4)

```cpp
//task4
SquareMatrix SquareMatrix::tril(const SquareMatrix& mat_tri){
    int dim = mat_tri.noOfRows;
    for (int i = 0; i < dim; ++i){
        for (int j = 0; j < dim; ++j){
            if (i < j && i >= 0) {
                mat_tri.data[i*dim + j] = 0.0;
            }
        }
    }
    return mat_tri;
}//static-tril. This function returns the lower triangular part of the input SquareMatrix mat_tri,by setting the elements in the upper part to 0.

SquareMatrix SquareMatrix::triu(const SquareMatrix& mat_tri){
    int dim = mat_tri.noOfRows;
    for (int i = 0; i < dim; ++i){
        for (int j = 0; j < dim; ++j){
            if (i > j && j >= 0) {
                mat_tri.data[i*dim + j] = 0.0;
            }
        }
    }
    return mat_tri;
}//static-triu. This function returns the upper triangular part of the input SquareMatrix mat_tri,by setting the elements in the lower part to 0.

SquareMatrix& SquareMatrix::tril(){
    for (int i = 0; i < noOfRows; ++i){
        for (int j = 0; j < noOfColumns; ++j){
            if (i < j && i >= 0) {
                data[i*noOfColumns + j] = 0.0;
            }
        }
    }
    return *this;
```

(C++ class codes for task4)

```cpp
    return *this;
}// non-static tril, which returns the lower triangular part of the SquareMatrix itself, through setting the elements of the upper triangular part of the SquareMatrix to zero.

SquareMatrix& SquareMatrix::triu(){
    for (int i = 0; i < noOfRows; ++i){
        for (int j = 0; j < noOfColumns; ++j){
            if (i > j && j >= 0) {
                data[i*noOfColumns + j] = 0.0;
            }
        }
    }
    return *this;
}// non-static triu, which returns the upper triangular part of the SquareMatrix itself, through setting the elements of the lower triangular part of the SquareMatrix to zero.
```

(C++ class codes for task4)

Conclusion: The results for both static and non-static match that in Matlab. Matrix 6 is the result of triu and matrix 7 is the result of tril.

```
Testing the Triangular upper functions:

Case 1: the static Trangular upper function
The original Matrix =
     1     3     2     1
     2     1     3     2
     3     2     1     3
     4     3     2     1

The static trangular upper version =
     1     3     2     1
     0     1     3     2
     0     0     1     3
     0     0     0     1

Press any key to continue ...

Case 2: the non-static Trangular upper function
The original Matrix =
     1     3     2     1
     2     1     3     2
     3     2     1     3
     4     3     2     1

The non-static trangular upper version =
     1     3     2     1
     0     1     3     2
     0     0     1     3
     0     0     0     1

Press any key to continue ...

Enter '0' to exit or '1' to choose another test
>>
```

(C++ Triu testing)

```
Testing the Triangular lower functions:

Case 1: the static Trangular lower function
The original Matrix =
     1     3     2     1
     2     1     3     2
     3     2     1     3
     4     3     2     1

The static trangular lower version =
     1     0     0     0
     2     1     0     0
     3     2     1     0
     4     3     2     1

Press any key to continue ...

Case 2: the non-static Trangular lower function
The original Matrix =
     1     3     2     1
     2     1     3     2
     3     2     1     3
     4     3     2     1

The non-static trangular lower version =
     1     0     0     0
     2     1     0     0
     3     2     1     0
     4     3     2     1

Press any key to continue ...

Enter '0' to exit or '1' to choose another test
>>
```

(C++ Tril testing)

```
matrix6 =

     1     3     2     1
     0     1     3     2
     0     0     1     3
     0     0     0     1


matrix7 =

     1     0     0     0
     2     1     0     0
     3     2     1     0
     4     3     2     1
```

(Matlab testing)

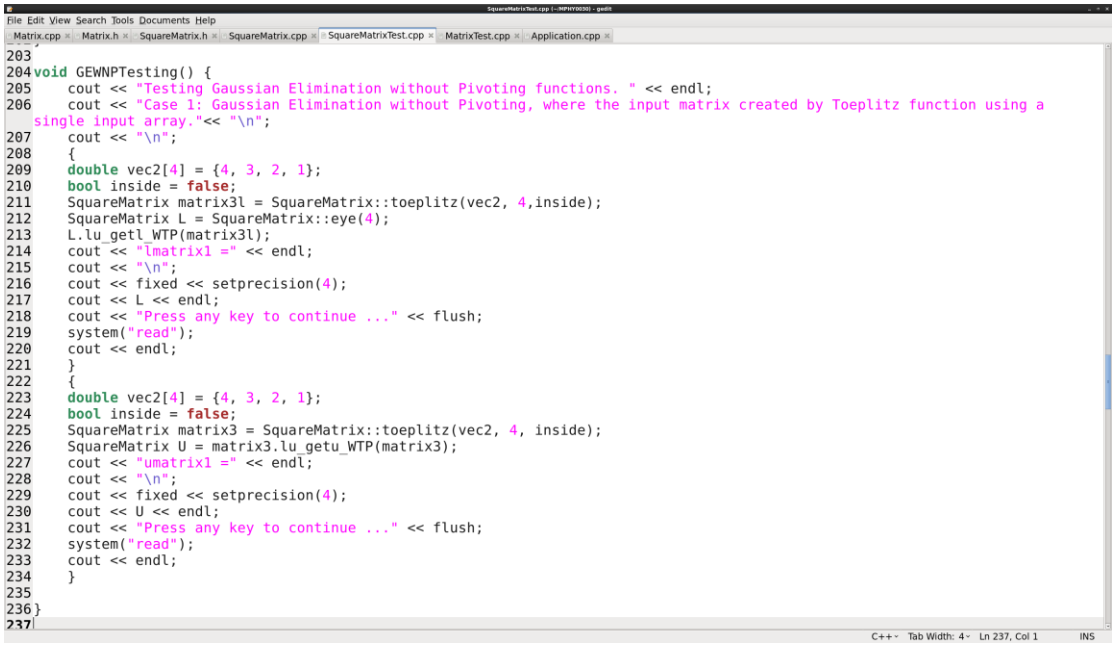
(5) Gaussian Elimination Function Without Pivoting Testing (task5)

Gaussian Elimination Function Without Pivoting is an important algorithm in solving system of linear equations and other numerical applications. The first step starts with working on LU decomposition, lu function in matlab. It can be observed that lu takes matrix3 as input, where matrix 3 is the Toeplitz created by a single input array.

```
vec1 = [4, 3, 2, 1];
vec2 = [1, 2, 3, 4];
matrix3 = toeplitz(vec1)
matrix4 = toeplitz(vec2, vec1)

matrix5 = transpose(matrix4)

matrix6 = triu(matrix4)

matrix7 = tril(matrix4)

[lmatrix1, umatrix1] = lu(matrix3)
```

（Matlab codes）

Hence, create a testing function (Gaussian Elimination without pivoting (GEWNP)Testing) considering the above case.




```
203
204void GEWNPTesting() {
205    cout << "Testing Gaussian Elimination without Pivoting functions. " << endl;
206    cout << "Case 1: Gaussian Elimination without Pivoting, where the input matrix created by Toeplitz function using a
   single input array."<< "\n";
207    cout << "\n";
208    {
209    double vec2[4] = {4, 3, 2, 1};
210    bool inside = false;
211    SquareMatrix matrix3l = SquareMatrix::toeplitz(vec2, 4,inside);
212    SquareMatrix L = SquareMatrix::eye(4);
213    L.lu_getl_WTP(matrix3l);
214    cout << "lmatrix1 =" << endl;
215    cout << "\n";
216    cout << fixed << setprecision(4);
217    cout << L << endl;
218    cout << "Press any key to continue ..." << flush;
219    system("read");
220    cout << endl;
221    }
222    {
223    double vec2[4] = {4, 3, 2, 1};
224    bool inside = false;
225    SquareMatrix matrix3 = SquareMatrix::toeplitz(vec2, 4, inside);
226    SquareMatrix U = matrix3.lu_getu_WTP(matrix3);
227    cout << "umatrix1 =" << endl;
228    cout << "\n";
229    cout << fixed << setprecision(4);
230    cout << U << endl;
231    cout << "Press any key to continue ..." << flush;
232    system("read");
233    cout << endl;
234    }
235
236}
237
```

It can be observed that lmatrix1 and umatrix1 are returned as outputs in lu(matrix3) in Matlab. Hence, create headers and relevant functions which returns LU, U and L without pivoting. abs is function used in both task5 and 6, which returns an absolute value. If pivot is zero, divide by this pivot will lead to singular values in matrix, therefore a singular matrix is resultant.

```
61//task5
62
63    SquareMatrix& lu_WTP(const SquareMatrix& mat_lu);// By inputting a SquareMatrix(mat_lu, aslo known as A), this function
   returns a non-singular square full or sparse Matrix, which can be factorised into its upper triangular matrix U and a
   permuted lower triangular matrix L. A= LU. LU Decomposition Case: Gaussian Elimination without Pivotting.
64
65    SquareMatrix& lu_getu_WTP(const SquareMatrix& mat_lu); //By inputting a SquareMatrix(mat_lu, also known as A), this
   function returns the factor of A, U, which is the upper triangular part of A. LU Decomposition Case: Gaussian Elimination
   without Pivotting.
66
67    SquareMatrix& lu_getl_WTP(const SquareMatrix& mat_lu);//By inputting a SquareMatrix(mat_lu, also known as A), this
   function returns the factor of A, L, which is the lower triangular part of A. LU Decomposition Case: Gaussian Elimination
   without Pivotting.
68
69//task6
70    static double abs(const double val);// This function obtains the absolute value of the input const double val, and return
   it as a double
71
```

(Headers for task5)

```cpp
//task5
SquareMatrix& SquareMatrix::lu_WTP(const SquareMatrix& mat_lu){
    int dim = mat_lu.noOfRows;
    for(int i = 0; i < dim*dim; i++){
        data[i] = mat_lu.data[i];
    }// Initialisation of data, assign value of the elements in input SquareMatrix(mat_lu) to every element of the SquareMatrix(data)

    for(int i = 0; i < dim; ++i){
        if(data[GetIndex(i,i)] == 0){// Case: To avoid having 0 as pivot, display the error and exit the program.
            cerr << "Error: the coefficient has 0 as pivot, divide by 0 will result in a singular value. Please fix this." << endl;
            exit(1);
        }
        else if(isnan(abs(data[GetIndex(i,i)])) == 1){// Case: Check there are singular elements inside the SquareMatrix, if there is any singular element, display the error and exit the program.
            cerr << "Error: the input matrix has a singular element, divide by it will result in a value tends toward 0. And divide by 0, will lead to a singular value. Please fix this." << endl;
            exit(1);
        }
        else{// Case: If there are non singular elements and have non-zero pivots, perform Gaussian elimination.
            for (int j = i +1; j < dim; ++j){
                for (int k = i +1; k < dim; ++k){
                    data[GetIndex(j,k)] -= data[GetIndex(i,k)]* (data[GetIndex(j,i)]/data[GetIndex(i,i)]);
                }
            }
        }
    }//Perform Gaussian elimination without pivotting for U, where U(j,k) = U(j,k)-(U(jk)/U(kk))*U(i,k). where U is the upper triangular part of LU or A
    return *this;
}// By inputting a SquareMatrix(mat_lu, aslo known as A), this function returns a non-singular square full or sparse Matrix LU, which can be factorised into its upper triangular matrix U and a permuted lower triangular matrix L. A= L*U. LU Decomposition Case: Gaussian Elimination without Pivotting.

```


(Class code for LU for task 5)

```cpp

SquareMatrix& SquareMatrix::lu_getu_WTP(const SquareMatrix& mat_lu){
    lu_WTP(mat_lu);
    (*this).triu();
    return *this;
}//By inputting a SquareMatrix(mat_lu, also known as A), this function obtains the values form lu_WTP, then get rid off the lower triangular part, to return the factor of A, U, which is the upper triangular part of A. LU Decomposition Case: Gaussian Elimination without Pivotting.

SquareMatrix& SquareMatrix::lu_getl_WTP(const SquareMatrix& mat_lu){
    int dim = mat_lu.noOfRows;
    for(int i = 0; i < dim; ++i){
        for (int j = i +1; j < dim; ++j){
            for (int k = i +1; k < dim; ++k){
                data[GetIndex(j,i)] = mat_lu.data[mat_lu.GetIndex(j,i)]/mat_lu.data[mat_lu.GetIndex(i,i)];
                mat_lu.data[mat_lu.GetIndex(j,k)] -= mat_lu.data[mat_lu.GetIndex(i,k)]* (mat_lu.data[mat_lu.GetIndex(j,i)]/mat_lu.data[mat_lu.GetIndex(i,i)]);
            }
        }
    }//Perform Gaussian elimination without pivotting, to get values for the lower triangular part of A, where L = U(j,k)/U(k,k);
    return *this;
}//By inputting a SquareMatrix(mat_lu, also known as A), this function returns the factor of A, L, which is the lower triangular part of A. LU Decomposition Case: Gaussian Elimination without Pivotting.


double SquareMatrix::abs(const double val) {
    if (val >= 0){
        return val;
    }
    else{
        return -val;
    }
}// This function obtains the absolute value of the input const double val(means value), and return it as a double
```

(Class codes for U, L and abs)

Conclude: results for task 5 shown in C++ match that in Matlab.

```
Testing Gaussian Elimination without Pivoting functions.
Case 1: Gaussian Elimination without Pivoting, where the input matrix created by Toeplitz function using a single input array.

lmatrix1 =

         1.0000            0.0000            0.0000            0.0000
         0.7500            1.0000            0.0000            0.0000
         0.5000            0.8571            1.0000            0.0000
         0.2500            0.7143            0.8333            1.0000

Press any key to continue ...

umatrix1 =

         4.0000            3.0000            2.0000            1.0000
         0.0000            1.7500            1.5000            1.2500
         0.0000            0.0000            1.7143            1.4286
         0.0000            0.0000            0.0000            1.6667

Press any key to continue ...

Enter '0' to exit or '1' to choose another test
```

(C++ task5 result)

```
lmatrix1 =

    1.0000         0         0         0
    0.7500    1.0000         0         0
    0.5000    0.8571    1.0000         0
    0.2500    0.7143    0.8333    1.0000


umatrix1 =

    4.0000    3.0000    2.0000    1.0000
         0    1.7500    1.5000    1.2500
         0         0    1.7143    1.4286
         0         0         0    1.6667
```

(Matlab task 5 result)

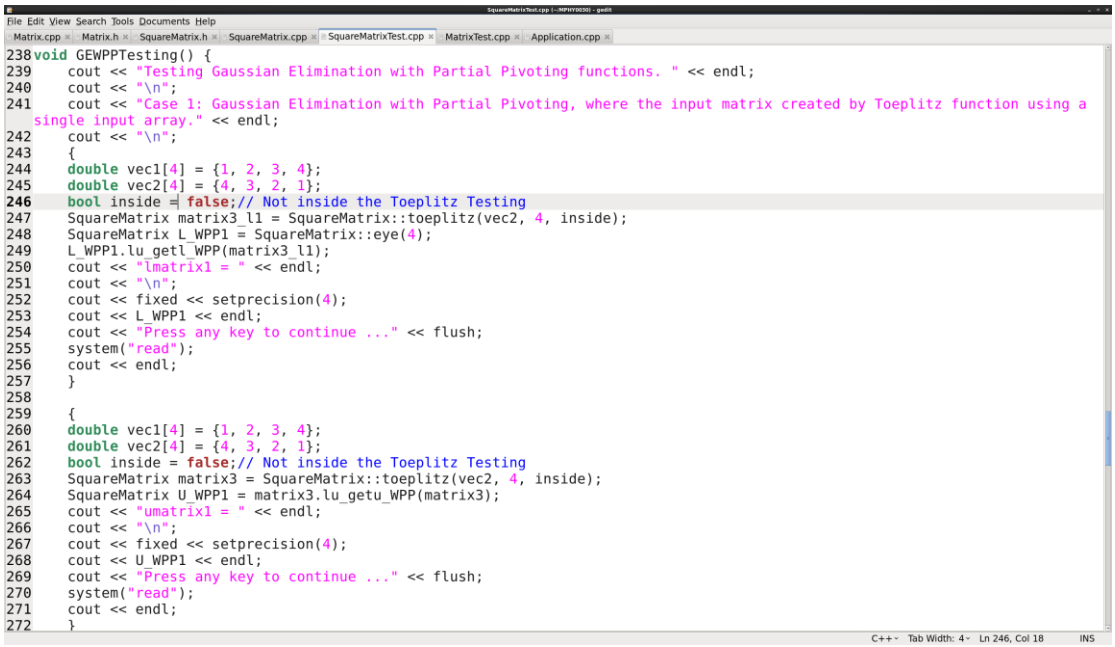
(6) Gaussian Elimination with Partial Pivotting (GEWWP)Testing (task 6)

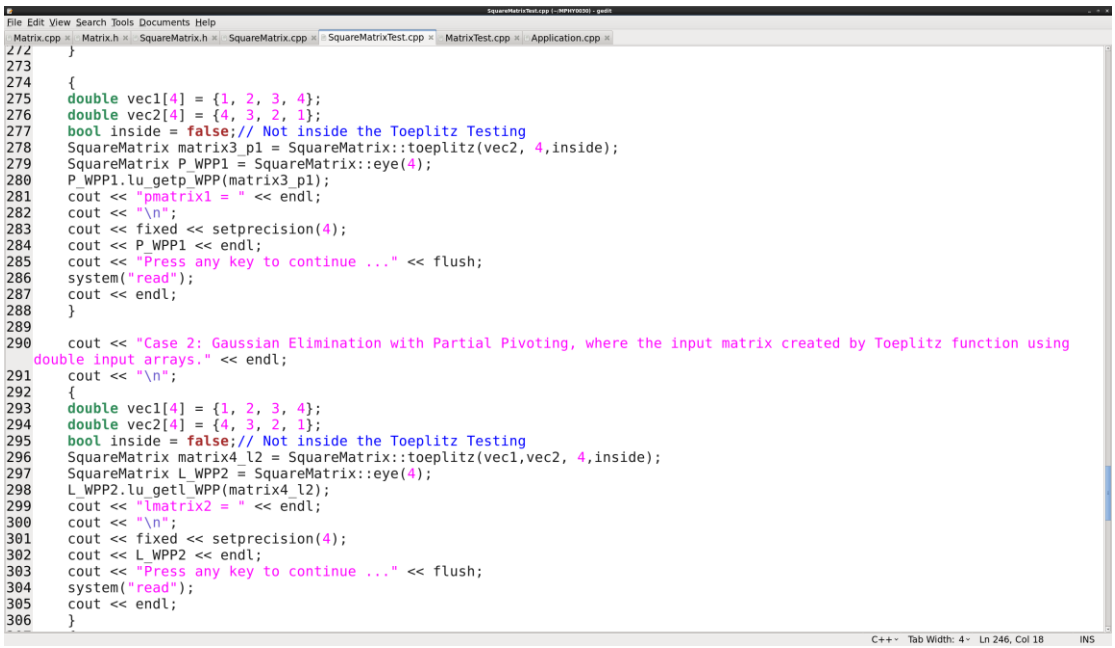
The strategy of GEWWP is quite similar to that of GEWNP. The difference of GEWWP with GEWNP is the partial pivoting, which involves the interchanges of U,L and P(exact details please see class codes). This algorithm is more stable, due to its non-singularity. If the input matrix is singular, this may lead to 0 to be chosen as pivot, and the issue of choosing pivot as 0 leading to singular matrix will occur again. In Matlab, function lu takes matrix3 and matrix 4 as input in two different cases, and corresponding lmatrix, umatrix and pmatrix are produced.

```
[lmatrix1, umatrix1, pmatrix1] = lu(matrix3)

[lmatrix2, umatrix2, pmatrix2] = lu(matrix4)
```

(Matlab code)

Hence, create the testing function considering different input cases.





```
File Edit View Search Tools Documents Help
Matrix.cpp | Matrix.h | SquareMatrix.h | SquareMatrix.cpp | SquareMatrixTest.cpp | MatrixTest.cpp | Application.cpp
272     }
273
274     {
275     double vec1[4] = {1, 2, 3, 4};
276     double vec2[4] = {4, 3, 2, 1};
277     bool inside = false;// Not inside the Toeplitz Testing
278     SquareMatrix matrix3_p1 = SquareMatrix::toeplitz(vec2, 4,inside);
279     SquareMatrix P_WPP1 = SquareMatrix::eye(4);
280     P_WPP1.lu_getp_WPP(matrix3_p1);
281     cout << "pmatrix1 = " << endl;
282     cout << "\n";
283     cout << fixed << setprecision(4);
284     cout << P_WPP1 << endl;
285     cout << "Press any key to continue ..." << flush;
286     system("read");
287     cout << endl;
288     }
289
290     cout << "Case 2: Gaussian Elimination with Partial Pivoting, where the input matrix created by Toeplitz function using
    double input arrays." << endl;
291     cout << "\n";
292     {
293     double vec1[4] = {1, 2, 3, 4};
294     double vec2[4] = {4, 3, 2, 1};
295     bool inside = false;// Not inside the Toeplitz Testing
296     SquareMatrix matrix4_l2 = SquareMatrix::toeplitz(vec1,vec2, 4,inside);
297     SquareMatrix L_WPP2 = SquareMatrix::eye(4);
298     L_WPP2.lu_getl_WPP(matrix4_l2);
299     cout << "lmatrix2 = " << endl;
300     cout << "\n";
301     cout << fixed << setprecision(4);
302     cout << L_WPP2 << endl;
303     cout << "Press any key to continue ..." << flush;
304     system("read");
305     cout << endl;
306     }
                                                    C++ Tab Width: 4 Ln 246, Col 18 INS
```

```
308     {
309     double vec1[4] = {1, 2, 3, 4};
310     double vec2[4] = {4, 3, 2, 1};
311     bool inside = false;
312     SquareMatrix matrix4 = SquareMatrix::toeplitz(vec1,vec2, 4,inside);
313     SquareMatrix U_WPP2 = matrix4.lu_getu_WPP(matrix4);
314     cout << "umatrix2 = " << endl;
315     cout << fixed << setprecision(4);
316     cout << U_WPP2 << endl;
317     cout << "Press any key to continue ..." << flush;
318     system("read");
319     cout << endl;
320     }
321
322     {
323     double vec1[4] = {1, 2, 3, 4};
324     double vec2[4] = {4, 3, 2, 1};
325     bool inside = false;// Not inside the Toeplitz Testing
326     SquareMatrix matrix4_p2 = SquareMatrix::toeplitz(vec1,vec2, 4,inside);
327     SquareMatrix P_WPP = SquareMatrix::eye(4);
328     P_WPP.lu_getp_WPP(matrix4_p2);
329     cout << "pmatrix2 = " << endl;
330     cout << "\n";
331     cout << fixed << setprecision(4);
332     cout << P_WPP << endl;
333     cout << "Press any key to continue ..." << flush;
334     system("read");
335     cout << endl;
336     }
337 }
338
```

(Task6: Testing codes for two input array case)

Create headers and implement functions that apearred in the test file, abs is used for finding the partial pivotting.

```
69 //task6
70     static double abs(const double val);// This function obtains the absolute value of the input const double val, and return
   it as a double
71
72     SquareMatrix& lu_WPP(const SquareMatrix& mat_lu);// By inputting a SquareMatrix(mat_lu, also known as A), this function
   returns a square full or sparse Matrix A, which can return a permutation matrix P and can be factorised into its upper
   triangular matrix U and a lower triangular matrix L. A= P'*L*U. LU Decomposition Case: Gaussian Elimination with Patial
   Pivotting.
73
74     SquareMatrix& lu_getu_WPP(const SquareMatrix& mat_lu);//By inputting a SquareMatrix(mat_lu, also known as A), this
   function returns the factor of A, U, which is the upper triangular part of A. LU Decomposition Case: Gaussian Elimination
   with Partial Pivotting.
75
76     SquareMatrix& lu_getl_WPP(const SquareMatrix& mat_lu);//By inputting a SquareMatrix(mat_lu, also known as A), this
   function returns the factor of A, L, which is the lower triangular part of A. LU Decomposition Case: Gaussian Elimination
   with Partial Pivotting.
77
78     SquareMatrix& lu_getp_WPP(const SquareMatrix& mat_lu);// By inputting a SquareMatrix(mat_lu, also known as A), this
   returns a permutation matrix P such that A = P'*L*U. LU Decomposition Case: Gaussian Elimination with Partial Pivotting.
79
```

(Headers for task 6)

```
File Edit View Search Tools Documents Help
Matrix.cpp | Matrix.h | SquareMatrix.h | SquareMatrix.cpp | SquareMatrixTest.cpp | MatrixTest.cpp | Application.cpp
```

```
SquareMatrix& SquareMatrix::lu_WPP(const SquareMatrix& mat_lu){
    int dim = mat_lu.noOfRows;//Define the dimension as the dimension of the input matrix

    for(int i = 0; i < dim*dim; i++){
        data[i] = mat_lu.data[i];
    }// Initialisation of data, assign value of the elements in input SquareMatrix(mat_lu) to every element of the SquareMatrix(data)
    for(int i = 0; i < dim; ++i){
        if(isnan(abs(data[GetIndex(i,i)])) == 1){// Case: Check there are singular elements inside the SquareMatrix, if there is any singular element, display the error and exit the program.
            cerr << "Error: the input matrix has a singular element, divide by it will result in a value tends toward 0. And divide by 0, will lead to a singular value. Please fix this." << endl;
            exit(1);
        }
        else{// Case: If there are non singular elements and have non-zero pivots(avoided by this algorithm), perform Gaussian elimination.
            double max = 0;//initialise maximum value
            int p = i; // initialise position
            for(int a = i; a < dim; ++a){
                double abs_val = abs(data[GetIndex(a,i)]);
                if(max < abs_val){
                    max = abs_val;
                    p = a;
                }
            }//select i >= a to maximise |data ia|, or say determine determine the entry on or below its diagnal with the largest absolute value for each column. This chosen as pivot.
            if(p != i){// if the position is not at the initial position, then interchanges are taking place
                for (int a = i; a < dim; ++a){
                    double tmp = data[GetIndex(i,a)];
                    data[GetIndex(i,a)] = data[GetIndex(p,a)];
                    data[GetIndex(p,a)] = tmp;
                }
                for (int a = 0; a < i; ++a){
                    double tmp = data[GetIndex(i,a)];
                    data[GetIndex(i,a)] = data[GetIndex(p,a)];
                    data[GetIndex(p,a)] = tmp;
                }//interchange the row containing the diagnal of the column(data (i,a)) with the row containing the new pivot (data (p,a)).Case: interchange data(i,a) and data(p,a), like interchange LU(i,a) and LU(p,a)
            }
            for (int j = i +1; j < dim; ++j){
                for (int k = i +1; k < dim; ++k){
                    data[GetIndex(j,k)] -= data[GetIndex(i,k)]* (data[GetIndex(j,i)]/data[GetIndex(i,i)]);
                }
            }//Perform Gaussian elimination with pivotting for U, where U(j,k) = U(j,k) - (U(jk)/U(kk))*U(i,k).U, upper triangular part of LU
        }
    }
    return *this;
}//By inputting a SquareMatrix(mat_lu, also known as A), this function returns a square full or sparse Matrix LU, which can return a permutation matrix P and can be factorised into its upper triangular matrix U and a lower triangular matrix L. LU= P'*L*U. LU Decomposition Case: Gaussian Elimination with Patial Pivotting.


SquareMatrix& SquareMatrix::lu_getu_WPP(const SquareMatrix& mat_lu){
    lu_WPP(mat_lu);
    (*this).triu();
    return *this;
}//By inputting a SquareMatrix(mat_lu, also known as A), this function obtains the values form lu_WPP, then get rid off the lower triangular part, to return the factor of A, U, which is the upper triangular part of LU. LU Decomposition Case: Gaussian Elimination with Partial Pivotting.
```

(Class code for obtaining LU and U for task 6)

```
SquareMatrix& SquareMatrix::lu_getl_WPP(const SquareMatrix& mat_lu){
    lu_WPP(mat_lu);
    int dim = mat_lu.noOfRows;
    double tmp[dim][dim];
    for(int i = 0; i < dim; ++i){
        for(int j = 0; j < dim; ++j){
            tmp[i][j] = data[GetIndex(i,j)];
        }
    }
    for(int i = 0; i < dim; ++i){
        for(int j = 0; j < dim; ++j){
            data[GetIndex(j,i)] = tmp[j][i] / tmp[i][i];
        }
    }
    (*this).tril();

    return *this;
}//Perform Gaussian elimination with pivotting, to get values for the lower triangular part of A, where L = U(j,k)/U(k,k).Then get rid off the upper triangular part, return L;
```

(Class code for obtaining L for task 6)

```cpp
SquareMatrix& SquareMatrix::lu_getp_WPP(const SquareMatrix& mat_lu){

    int dim = mat_lu.noOfRows;//Define the dimension as the dimension of the input matrix

    for(int i = 0; i < dim; ++i){
        double max = 0;//initialise maximum value
        int p = i; // initialise position
        for(int a = i; a < dim; ++a){
            double abs_val = mat_lu.abs(mat_lu.data[mat_lu.GetIndex(a,i)]);
            if(max <= abs_val){
                max = abs_val;
                p = a;
            }
        }
        if(p != i){
            for (int a = 0; a < dim; ++a){
                double tmp = data[GetIndex(i,a)];
                data[GetIndex(i,a)] = data[GetIndex(p,a)];
                data[GetIndex(p,a)] = tmp;
            }
        }
    }// interchange P(a,:) and P(p,:)
    return *this;
}// By inputting a SquareMatrix(mat_lu, also known as A), this returns a permutation matrix P such that A = P'*L*U. This P
   SquareMatrix records the swapping of rows. LU Decomposition Case: Gaussian Elimination with Partial Pivotting.
```

(Class code for obtaining P for task 6)

Conclusion: Result matches that in matlab

```
Testing Gaussian Elimination with Partial Pivoting functions.

Case 1: Gaussian Elimination with Partial Pivoting, where the input matrix created by Toeplitz function using a single input array.

lmatrix1 =

        1.0000            0.0000            0.0000            0.0000
        0.7500            1.0000            0.0000            0.0000
        0.5000            0.8571            1.0000            0.0000
        0.2500            0.7143            0.8333            1.0000

Press any key to continue ...

umatrix1 =

        4.0000            3.0000            2.0000            1.0000
        0.0000            1.7500            1.5000            1.2500
        0.0000            0.0000            1.7143            1.4286
        0.0000            0.0000            0.0000            1.6667

Press any key to continue ...

pmatrix1 =

        1.0000            0.0000            0.0000            0.0000
        0.0000            1.0000            0.0000            0.0000
        0.0000            0.0000            1.0000            0.0000
        0.0000            0.0000            0.0000            1.0000

Press any key to continue ...
```

(Task 6 case1 result in c++)

```
lmatrix1 =

    1.0000         0         0         0
    0.7500    1.0000         0         0
    0.5000    0.8571    1.0000         0
    0.2500    0.7143    0.8333    1.0000


umatrix1 =

    4.0000    3.0000    2.0000    1.0000
         0    1.7500    1.5000    1.2500
         0         0    1.7143    1.4286
         0         0         0    1.6667


pmatrix1 =

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

(Task 6 case1 result in Matlab)

```
Case 2: Gaussian Elimination with Partial Pivoting, where the input matrix created by Toeplitz function using double input arrays.

lmatrix2 =

         1.0000            0.0000            0.0000            0.0000
         0.2500            1.0000            0.0000            0.0000
         0.5000           -0.2222            1.0000            0.0000
         0.7500           -0.1111           -0.1429            1.0000

Press any key to continue ...

umatrix2 =
         4.0000            3.0000            2.0000            1.0000
         0.0000            2.2500            1.5000            0.7500
         0.0000            0.0000            2.3333            1.6667
         0.0000            0.0000            0.0000            2.5714

Press any key to continue ...

pmatrix2 =

         0.0000            0.0000            0.0000            1.0000
         1.0000            0.0000            0.0000            0.0000
         0.0000            1.0000            0.0000            0.0000
         0.0000            0.0000            1.0000            0.0000

Press any key to continue ...

Enter '0' to exit or '1' to choose another test
```

(Task 6 case2 result in c++)

```
lmatrix2 =

    1.0000         0         0         0
    0.2500    1.0000         0         0
    0.5000   -0.2222    1.0000         0
    0.7500   -0.1111   -0.1429    1.0000


umatrix2 =

    4.0000    3.0000    2.0000    1.0000
         0    2.2500    1.5000    0.7500
         0         0    2.3333    1.6667
         0         0         0    2.5714


pmatrix2 =

     0     0     0     1
     1     0     0     0
     0     1     0     0
     0     0     1     0
```

(Task 6 case2 result in Matlab)

Part3

Introduction

This part uses the implementation of part2 LU decomposition to solve systems of linear equations. The below are the selection interface implemented using both Matrix and SquareMatrix class for different testing created in “Application.cpp”.

Selection interface, where it consders two cases: number of equations = number of unknowns or number of equations > number of unknowns.

```
Choose to test one of the following:
  Enter '1' for the case where number of equations equals to number of unknowns
  Enter '2' for the case where number of equations is larger than number of unknowns
>>
```

The corresponding codes are:

```
126int main () {
127
128    for (;;) {
129        cout << "Choose to test one of the following:" << endl;
130        cout << "  Enter \'1\' for the case where number of equations equals to number of unknowns" << endl;
131        cout << "  Enter \'2\' for the case where number of equations is larger than number of unknowns" << endl;
132        cout << ">> ";
133        char choice;
134        cin >> choice;
135        switch (choice) {
136            case '1':   LUdecomTesting();
137                        break;
138            case '2':   LSQTesting();
139                        break;
140        }
141        cout << "Enter \'0\' to exit or \'1\' to choose another test" << endl;
142        cout << ">> ";
143        cin >> choice;
144        if (choice == '0') {
145            return 0;
146        }
147    }
148
149}
150
```

Method

A system of linear equations are shown in the left hand side of the following figure, where the corresponding Matrix form are shown in right hand side : a SquareMatrix A stores the coefficients of unknown parameters, b is the final matrix product and x is a vector which stores the value of unknowns.

$$\begin{aligned} 2w + x + y &= 1 \\ 4w + 3x + 3y + z &= 0 \\ 8w + 7x + 9y + 5z &= 0 \\ 6w + 7x + 9z + 8z &= 0 \end{aligned} \quad \longleftrightarrow \quad \underset{\mathbf{A}}{\begin{pmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{pmatrix}} \underset{\mathbf{x}}{\begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix}} = \underset{\mathbf{b}}{\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}}$$

In order to solve this system and obtain x values, there are two typical methods to solve x, they are LU Decomposition method and Least Square Method.

(1) LU decomposition method (Task1)

Case: number of equations = number of unknowns. LU decomposition simply starts with LUx = Pb, where Pb is the Matrix product of the permutation matrix P of input coefficient Matrix A and the resultant vector b. x can be found by y = Pb/L (forward substitution) and then x = y/U(back substitution), where L is the lower triangular Matrix of A or LU and U is the upper triangular Matrix of A or LU. A = LU, so they are basically same thing.

By using the forward and backward built function in the matlab, a testing code in matlab is:

```
A = [2, 1, 1, 0; 4, 3, 3, 1; 8, 7, 9, 5; 6, 7, 9, 8]

[L, U, P] = lu(A);

b= [1, 0, 0, 0]'

a = P * b
y = forward(L, a)
x = backward(U, y)
```

Hence, create a test file for LU decomposition method testing.

```cpp
#include "SquareMatrix.h"
#include "Matrix.h"
#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;
// LU Decomposition Method Testing (Forward and backward substitution method)
void LUdecomTesting(){
    cout << "Testing for the case where number of equations equals to number of unknowns:" << endl;
    {
    //Create a SquareMatrix A, which stores data from a double input array with a size of 16. b is a double input array with a size of 4.
    double input[16] = {2, 1, 1, 0, 4, 3, 3, 1, 8, 7, 9, 5, 6, 7, 9, 8};
    double b[4] = {1, 0, 0, 0};
    SquareMatrix A(4);
    A = input;
    cout << "A =" << "\n";
    cout << "\n";
    cout << fixed << setprecision(4);
    cout << A << endl;

    //Transfer data and convert format from Matrix (A) to SquareMatrix(Li), as the input matrix in lu_getl has to be a SquareMatrix;
    //Declare a SquareMatrix U to store the lower SquareMatrix data obatained through Gaussian Elimination and LU Decomposition with Partial Pivot
    SquareMatrix Li = A;
    SquareMatrix L(4);
    L.lu_getl_WPP(Li);
    cout << "L = " << endl;
    cout << "\n";
    cout << fixed << setprecision(4);
    cout << L << endl;

    //Transfer data and convert format from Matrix (A) to SquareMatrix(Ui), as the input matrix in lu_getu has to be a SquareMatrix;
```

As no swapping occurred, so P is initialised with an identity SquareMatrix.

```cpp
    //Transfer data and convert format from Matrix (A) to SquareMatrix(Ui), as the input matrix in lu_getu has to be a SquareMatrix;
    //Declare a SquareMatrix U to store the Upper SquareMatrix data obatained through Gaussian Elimination and LU Decomposition with Partial Pivot
    SquareMatrix Ui = A;
    SquareMatrix U(4);
    U = Ui.lu_getu_WPP(Ui);
    cout << "U = " << endl;
    cout << U << endl;

    //Transfer data and convert format from Matrix (A) to SquareMatrix(Pi), as the input matrix in lu_getp has to be a SquareMatrix;
    //Declare an identity SquareMatrix P to store the Upper SquareMatrix data obatained through Gaussian Elimination and LU Decomposition with Partial Pivot
    SquareMatrix Pi = A;
    SquareMatrix P = SquareMatrix::eye(4);
    P.lu_getp_WPP(Pi);
    cout << "P = " << endl;
    cout << P << endl;

    // Declare a Matrix x with diemsions of 4*1, store the final solution (x value) obatined through getxsubstitution method to the Matrix, and display the final result
    Matrix x(4,1);
    double *x1 = A.getxsubstitution(L,P,U,b,1);
    x = x1;
    cout << "\n";
    cout << "The final solution, x = " << endl;
    cout << fixed << setprecision(4);
    cout << x << endl;
    cout << "\n";
    cout << "Press any key to continue ..." << flush;
    system("read");
    cout << endl;
    }
}

```

(C++ Test codes for task1)

In this test file, for both methods, it involves the storing of data from the double-type address of an input array to a Matrix or a SquareMatrix via assignment operator, also from Matrix to a SquareMatrix. Hence the headers and functions of these assignment operator are defined as below.

```cpp
    Matrix& operator= (const double *input);// Assignment operator (M4),This assigns the values stored by double-type adress of input in the right hand side to the matrix in the left hand side of the = operator

};

#endif /* MATRIX_H_ */
```

(Headers for task1 and 2 in Matrix class)

```
292Matrix& Matrix::operator= (const double *input) {
293
294    for (int i = 0; i < noOfRows*noOfColumns; ++i) {
295        data[i] = input[i];
296    }
297}
298// Used in Part2 and 3 of the coursework, assignment constructor(M4),in which the left hand side Matrix stores the data from
   the data stored in the address of the input array in the right hand side of = operator
```


(Class code for task1 and 2 in Matrix class)

```
87    SquareMatrix& operator = (const Matrix& mat);// Assignment constructor(M4), which assigns the value of the right hand
  side input Matrix(mat) to the left hand side SquareMatrix.
88    SquareMatrix& operator = (const double *input);// Assignment constructor(M5), which assigns the value of the right hand
  side input array(input) to the left hand side SquareMatrix. & dereferences the address of the input array, which returns the
  array elements in SquareMatrix type.
```

(Headers for task1 and 2 in SquareMatrix class)

```
410SquareMatrix& SquareMatrix::operator = (const Matrix& mat){
411    this -> Matrix::operator = (mat);
412    return *this;
413}// Assignment constructor(M4), which assigns the value of the right hand side input Matrix(mat) to the left hand side
   SquareMatrix. This constructor inherits from the assignment operator(M3) in the parent class, and the only difference is the
   return type is in SquareMatrix format.
414
415SquareMatrix& SquareMatrix::operator = (const double *input){
416    this -> Matrix::operator = (input);
417    return *this;
418}// Assignment constructor(M5), which assigns the value of the right hand side input the address of the double-type input to
   the left hand side SquareMatrix. This constructor inherits from the assignment operator(M4) in the parent class, and the
   only difference is the return type is in SquareMatrix format. Due to this difference, hence we have to declare SquareMatrix
   with its own assignment constructors to ensure right type of return.
```


(Class code for task1 and 2 in SquareMatrix class)

In LUdecomTesting, the main method of getting x is via getxsubstitution, and this method is a combination of forward and backward substitution. Hence, their headers and functions are:

```
80// Part3
81//task1
82    double* getxsubstitution(const SquareMatrix& L, const SquareMatrix& P, const SquareMatrix& U, const double *b,const int
  ncol);// By giving const SquareMatrices include L(lower triangular part of A),P(permutation SquareMatrix returned by A) and U
  (upper triangular part of A), integer ncol (number of columns) and double input array b as inputs to this function, a double-
  type address of the final solution x is returned as the final result of solving systems of linear equations. In short, solves
  the x in Ax = b. Case: Number of equations = Number of unknowns
83
84    SquareMatrix backsubstitution(const SquareMatrix& U, const SquareMatrix& ATB,const int nrow);// This function solves the
  upper triangular system using back substitution method, the inputs are upper triangular SquareMatrix U, integer nrow (number
  of rows) and the product SquareMatrix in the right handside of = operator , ATB, where Ux = ATB. Then, this function returns
  the final solution x as a SquareMatrix with a dismenion of nrow * nrow.
85
86    SquareMatrix forwardsubstitution(const SquareMatrix& L, const SquareMatrix& Pb, const int nrow, const int ncol);// This
  function solves the lower triangular system using forward substitution method, the inputs are lower triangular SquareMatrix
  U, integer nrow (number of rows), integer ncol(number of columns) and the product SquareMatrix in the right handside of =
  operator, Pb, where Ly = Pb. y is equal to ATB in the backsubstitution. Then, this function returns the solution y as a
  SquareMatrix with a dismenion of nrow * nrow. ncol here determines the index of column that we want to store the value in.
```

(Headers for task1 in SquareMatrix class)

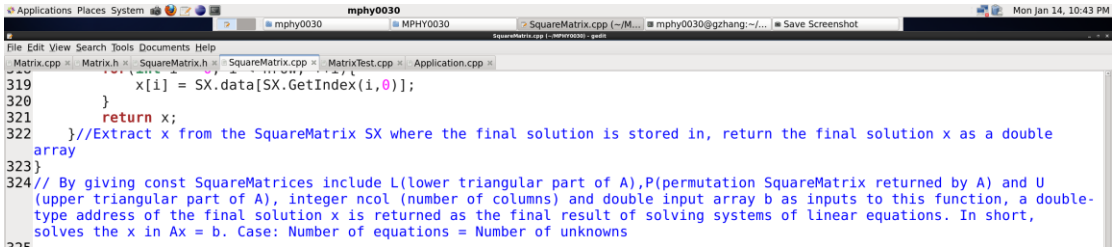
```
SquareMatrix SquareMatrix::forwardsubstitution(const SquareMatrix& L, const SquareMatrix& Pb, const int nrow, const int ncol)
{
    SquareMatrix y(nrow);
    cout << "\n";
    cout << "Applying the forward substitution method!" << "\n";
    y.data[y.GetIndex(0,0)] = Pb.data[Pb.GetIndex(0,0)]/L.data[L.GetIndex(0,0)];//The first element is equal to the product SquareMatrix in the right hand side divieded by the first element in L, Ly = Pb, so y[0,0] = Pb[0,0]/L[0,0]
    for (int a = 1; a < nrow; ++a){
        double diff = 0;
        if(L.data[L.GetIndex(a,a)] == 0){// avoid singular matrix

            cerr << " Matrix is singular!" << endl;
            exit(1);
        }
        else{
            for( int b = 0; b <=a-1; ++b){
                diff += y.data[y.GetIndex(b,0)]*L.data[L.GetIndex(a,b)];
                y.data[y.GetIndex(a,0)] = (Pb.data[Pb.GetIndex(a,0)] - diff)/L.data[L.GetIndex(a,a)];
            }// Get the sum of the products of L(a,a-1)*y(a-1,0), where the value of solution was stored in the first column of SquareMatrix y. Then use Pb(a,0) - the sum of the product of L and y from index = 1 to a-1.(accumulation mode)
        }
    }
    return y;
}
// This function solves the lower triangular system using forward substitution method, the inputs are lower triangular SquareMatrix U, integer nrow (number of rows), integer ncol(number of columns) and the product SquareMatrix in the right handside of = operator, Pb, where Ly = Pb. y is equal to ATB in the backsubstitution. Then, this function returns the solution y as a SquareMatrix with a dismenion of nrow * nrow. ncol here determines the index of column that we want to store the value in.
```

(Class code in SquareMatrix: Forward substitution for task 1: getxsubstitution)

```
SquareMatrix SquareMatrix::backsubstitution(const SquareMatrix& U, const SquareMatrix& ATB,const int nrow){
        SquareMatrix x(nrow);
        cout << "\n";
        cout << "Then, applying the backward substitution method!" << "\n";
        x.data[x.GetIndex(nrow,0)] = ATB.data[ATB.GetIndex(nrow,0)]/U.data[U.GetIndex(nrow,nrow)];//The first element is equal to the product SquareMatrix in the right hand side divieded by the first element in U, Ux = y, so x[nrow,0] = y[nrow,0]/U[nrow,nrow].
        for (int m = nrow-1; m >= 0; m--){
            double dif = 0;
            if(U.data[U.GetIndex(m,m)] == 0){// avoid singular matrix

                cerr << " Matrix is singular!" << endl;
                exit(1);
            }
            else(U.data[U.GetIndex(m,m)] != 0);{
                for(int j = m+1; j < nrow; j++){
                dif += U.data[U.GetIndex(m,j)]*x.data[x.GetIndex(j,0)];
                }
                x.data[x.GetIndex(m,0)] = (ATB.data[ATB.GetIndex(m,0)] - dif)/U.data[U.GetIndex(m,m)];

            }// Get the sum of the products of U(m,j)*x(j,0), where the value of solution was stored in the first column of SquareMatrix x. Then use ATB(m,0) - the sum of the product of L and y from index m+1 to nrow.(accumulation mode)
        }
    return x;
}// This function solves the upper triangular system using back substitution method, the inputs are upper triangular SquareMatrix U, integer nrow (number of rows) and the product SquareMatrix in the right handside of = operator , ATB, where Ux = ATB. Then, this function returns the final solution x as a SquareMatrix with a dismenion of nrow * nrow.
```

(Class code in SquareMatrix: Backward substitution for task 1: getxsubstitution)

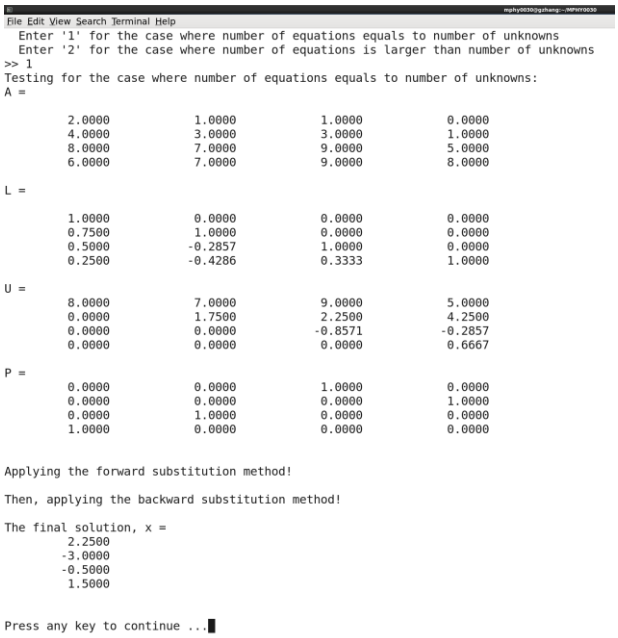
```
//part3
//task1
double* SquareMatrix::getxsubstitution(const SquareMatrix& L, const SquareMatrix& P, const SquareMatrix& U, const double *b,const int ncol){
    int nrow = P.noOfRows;
    int prow = P.noOfRows;
    int pcol = P.noOfColumns;
    int lrow = L.noOfRows;
    int lcol = L.noOfColumns;
    int urow = U.noOfRows;
    int ucol = U.noOfColumns;
    SquareMatrix Pb(nrow);
    double x[nrow];
    double y[nrow];
    //Initialisations
    if(prow != pcol||lrow != lcol|urow != ucol){
        cerr << "Input Matrix of P, U and L must be Square!" << endl;
        exit(1);
    }// Ensure that L,P and U are SquareMatrix
    else{
        for (int i = 0; i < nrow; ++i){
            Pb.data[Pb.GetIndex(i,0)] = 0;
            for (int j =0; j < ncol; ++j){
                for(int k = 0; k < nrow; ++k){
                    Pb.data[Pb.GetIndex(i,0)] += P.data[P.GetIndex(i,k)] * b[j*nrow + k];
                }
            }
        }
    // Returns a SqaureMatrix which performs element-wise multiplication and stores the Product of Permuation SquareMatrix P and the 1-D array b.
        SquareMatrix Y(nrow);
        SquareMatrix y = Y.forwardsubstitution(L, Pb, nrow, ncol);
        SquareMatrix X(nrow);
        SquareMatrix SX = X.backsubstitution(U, y, nrow);
        for(int i = 0; i < nrow; ++i){
            x[i] = SX.data[SX.GetIndex(i,0)];
```



```
319             x[i] = SX.data[SX.GetIndex(i,0)];
320         }
321         return x;
322     }//Extract x from the SquareMatrix SX where the final solution is stored in, return the final solution x as a double
    array
323 }
324 // By giving const SquareMatrices include L(lower triangular part of A),P(permutation SquareMatrix returned by A) and U
    (upper triangular part of A), integer ncol (number of columns) and double input array b as inputs to this function, a double-
    type address of the final solution x is returned as the final result of solving systems of linear equations. In short,
    solves the x in Ax = b. Case: Number of equations = Number of unknowns
```

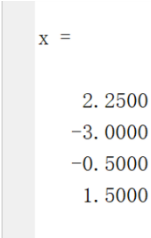
(Class code in SquareMatrix for task 1: getxsubstitution)

Conclusion: The x value matches the solution given in matlab and given by Matrix Calculator.



```
  Enter '1' for the case where number of equations equals to number of unknowns
  Enter '2' for the case where number of equations is larger than number of unknowns
>> 1
Testing for the case where number of equations equals to number of unknowns:
A =

    2.0000    1.0000    1.0000    0.0000
    4.0000    3.0000    3.0000    1.0000
    8.0000    7.0000    9.0000    5.0000
    6.0000    7.0000    9.0000    8.0000

L =

    1.0000         0         0         0
    0.7500    1.0000         0         0
    0.5000   -0.2857    1.0000         0
    0.2500   -0.4286    0.3333    1.0000

U =
    8.0000    7.0000    9.0000    5.0000
         0    1.7500    2.2500    4.2500
         0         0   -0.8571   -0.2857
         0         0         0    0.6667

P =
     0     0     1     0
     0     0     0     1
     0     1     0     0
     1     0     0     0

Applying the forward substitution method!

Then, applying the backward substitution method!

The final solution, x =
    2.2500
   -3.0000
   -0.5000
    1.5000

Press any key to continue ...
```

(Task1 result of x in c++)



```
x =

    2.2500
   -3.0000
   -0.5000
    1.5000
```

(Task1 result of x in Matlab)

(2) Least Square (LSQ)Method

Case: number of equations > number of unknowns. All the headers and functions defined in C++ previously, will be used by LSQTesting. The difference is that LSQTesting will start with ATAx = ATb, where AT is the transposed version of A. So it is necessary to obtain ATA and ATB. A test code in Matlab is produced.

```
A1 =[2, 5, 1;1, 2, 2;6, 1, 2;1, 5, 2]
At = A1'
Ap = At*A1
b1 =[3, 3, 2, 1]'

[L1, U1, P1] = lu(Ap)
Atb = At * b1

y1 = forward(L1, Atb)
x2 = backward(U1, y1)
```

(codes in Matlab)

```
File Edit View Search Tools Documents Help
Matrix.cpp | Matrix.h | SquareMatrix.h | SquareMatrix.cpp | MatrixTest.cpp | *Application.cpp
69//Least Square Testing
70void LSQTesting(){
71    cout << "Testing for the case where number of equations is larger than number of unknowns:" << endl;
72    {
73    //Create a Matrix A with a dimension of 4*3, which stores data from a double input array with a size of 12.
74    double input[12] = {2,5,1,1,2,2,6,1,2,1,5,2};
75    Matrix A(4,3);
76    A = input;
77    cout << "Input Matrix A, with Dimension of 4 * 3." << endl;
78    cout << fixed << setprecision(4);
79    cout << A << endl;
80
81    //Create a Matrix B with a dimension of 4*1, which stores data from b, which is a double input array with a size of 4.
82    double b[4] = {3,3,2,1};
83    Matrix B(4,1);
84    B = b;
85    cout << "Input Matrix B, with Dimension of 4 * 1." << endl;
86    cout << fixed << setprecision(4);
87    cout << B << endl;
88
89    //Transfer data and convert format from Matrix (A) to Matrix(A1), as the input matrix in getATA has to be a Matrix and
   this can avoid that data stored in A will not be changed by futher functions;
90    //Declare a SquareMatrix ATA to store the data of the product of A's transpose AT and A
91    Matrix A1(4,3);
92    A1 = A;
93    SquareMatrix ATA(3);
94    ATA = A1.getATA(A1);
95    cout << "Product, ATA = AT * A. Where AT is the transposed A and A is the 4 * 3 input Matrix." << endl;
96    cout << "\n";
97    cout << "ATA = " << endl;
98    cout << fixed << setprecision(4);
99    cout << ATA << endl;
100
101   //Transfer data and convert format from Matrix (A) to Matrix(A2), as the input matrix in getATA has to be a Matrix and
   this can avoid that data stored in A will not be changed by futher functions;
102   //Declare a SquareMatrix ATB to store the data of the product of A's transpose AT and B
C++ Tab Width: 4 Ln 99, Col 25 INS
```

```
File Edit View Search Tools Documents Help
Matrix.cpp | Matrix.h | SquareMatrix.h | SquareMatrix.cpp | MatrixTest.cpp | *Application.cpp
   this can avoid that data stored in A will not be changed by futher functions;
102   //Declare a SquareMatrix ATB to store the data of the product of A's transpose AT and B
103   Matrix A2(4,3);
104   A2 = A;
105   SquareMatrix ATB(3);
106   ATB = A2.getATB(A2,B);
107   cout << "Product, ATB = AT * B. Where AT is the transposed A, A is the 4 * 3 input Matrix, and B is the 4 * 1 input
   Matrix " << "\n";
108   cout << "\n";
109   cout << "ATB(Pb) = " << endl;
110   cout << fixed << setprecision(4);
111   cout << ATB << endl;
112
113   // Declare a Matrix x with diemsions of 4*1, store the final solution (x value) obatined through getxsubstitution method
   to the Matrix, and display the final result
114   Matrix x(3,1);
115   double *x1 = ATA.LSQ(ATA, ATB, A, 3, 1);
116   x = x1;
117   cout << "\n";
118   cout << "The final solution obatined using Least Square Method, x = " << endl;
119   cout << "\n";
120   cout << fixed << setprecision(4);
121   cout << x << endl;
122   }
123}
124
```

(C++ Test codes for task2)

The new headers and functions of LSQTesting compared to DecomTesting are defined as below. ATA and ATB in Matrix format are first obtained, then the SquareMatrix ATA and ATB copies data into them. Finally, the LSQ take them as inputs.

```
//Part 3
    Matrix getATA(const Matrix& A);// This returns the product of the input Matrix A and its transposed Matrix, ATA

    Matrix getATB(const Matrix& A, const Matrix& B);// This returns the product of the input Matrix A and the input Matrix B as a Matrix
```

(Headers for task2 in Matrix class)

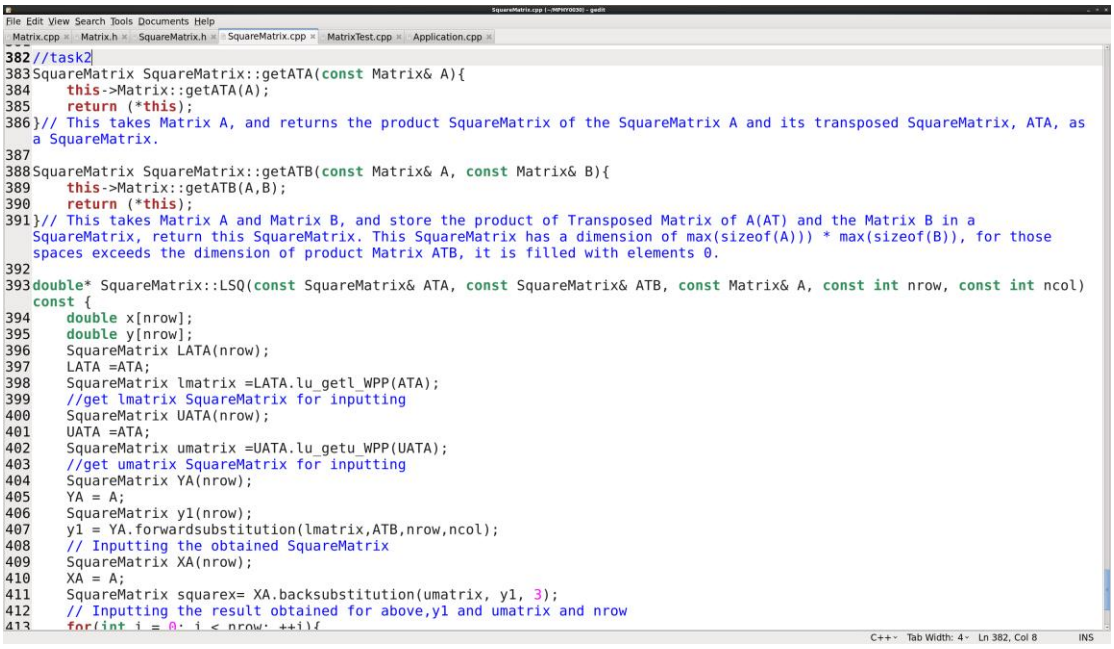
```
    SquareMatrix getATA(const Matrix& A);// This takes Matrix A, and returns the product SquareMatrix of the SquareMatrix A and its transposed SquareMatrix, ATA, as a SquareMatrix.
    SquareMatrix getATB(const Matrix& A, const Matrix& B);// This takes Matrix A and Matrix B, and store the product of Transposed Matrix of A(AT) and the Matrix B in a SquareMatrix, return this SquareMatrix. This SquareMatrix has a dimension of max(sizeof(A))) * max(sizeof(B)), for those spaces exceeds the dimension of product Matrix ATB, it is filled with elements 0.
    double* LSQ(const SquareMatrix& ATA, const SquareMatrix& ATB, const Matrix& A,const int nrow, const int ncol) const;// This function employs Least Square Method for solving systems of linear equations. It takes inputs of Matrix A, the product SquareMatrix in the right hand side of =, ATB and the product SquareMatrix of SquareMatrix version of A and its transpose (AT). such that ATA*x=ATB. This function returns the address of the final solution, x, as a double. ncol is for the use in the inside forwardsubstitution. Case: Number of equations > Number of unknowns.
};


#endif /* SQUAREMATRIX_H_ */
```

C/C++/ObjC Header · Tab Width: 4 · Ln 81, Col 8 · INS

(Headers for task2 in SquareMatrix class)

```
// Part3
Matrix Matrix::getATA(const Matrix& A){
    Matrix AP(A.noOfColumns,A.noOfColumns);

    Matrix AT = Matrix::Transpose(A);
    AP = AT*A;
    return AP;
}// This returns the product of the input Matrix A and its transposed Matrix, ATA

Matrix Matrix::getATB(const Matrix& A, const Matrix& B){
    Matrix ATB(A.noOfColumns,B.noOfColumns);
    Matrix AT = Matrix::Transpose(A);
    ATB = AT*B;
    return ATB;
}// This returns the product of the input Matrix A and the input Matrix B as a Matrix
```

(Class code in Matrix for task 2: getATA and getATB, parent class)




```
//task2
SquareMatrix SquareMatrix::getATA(const Matrix& A){
    this->Matrix::getATA(A);
    return (*this);
}// This takes Matrix A, and returns the product SquareMatrix of the SquareMatrix A and its transposed SquareMatrix, ATA, as a SquareMatrix.

SquareMatrix SquareMatrix::getATB(const Matrix& A, const Matrix& B){
    this->Matrix::getATB(A,B);
    return (*this);
}// This takes Matrix A and Matrix B, and store the product of Transposed Matrix of A(AT) and the Matrix B in a SquareMatrix, return this SquareMatrix. This SquareMatrix has a dimension of max(sizeof(A))) * max(sizeof(B)), for those spaces exceeds the dimension of product Matrix ATB, it is filled with elements 0.

double* SquareMatrix::LSQ(const SquareMatrix& ATA, const SquareMatrix& ATB, const Matrix& A, const int nrow, const int ncol) const {
    double x[nrow];
    double y[nrow];
    SquareMatrix LATA(nrow);
    LATA =ATA;
    SquareMatrix lmatrix =LATA.lu_getl_WPP(ATA);
    //get lmatrix SquareMatrix for inputting
    SquareMatrix UATA(nrow);
    UATA =ATA;
    SquareMatrix umatrix =UATA.lu_getu_WPP(UATA);
    //get umatrix SquareMatrix for inputting
    SquareMatrix YA(nrow);
    YA = A;
    SquareMatrix y1(nrow);
    y1 = YA.forwardsubstitution(lmatrix,ATB,nrow,ncol);
    // Inputting the obtained SquareMatrix
    SquareMatrix XA(nrow);
    XA = A;
    SquareMatrix squarex= XA.backsubstitution(umatrix, y1, 3);
    // Inputting the result obtained for above,y1 and umatrix and nrow
    for(int i = 0; i < nrow; ++i){
```

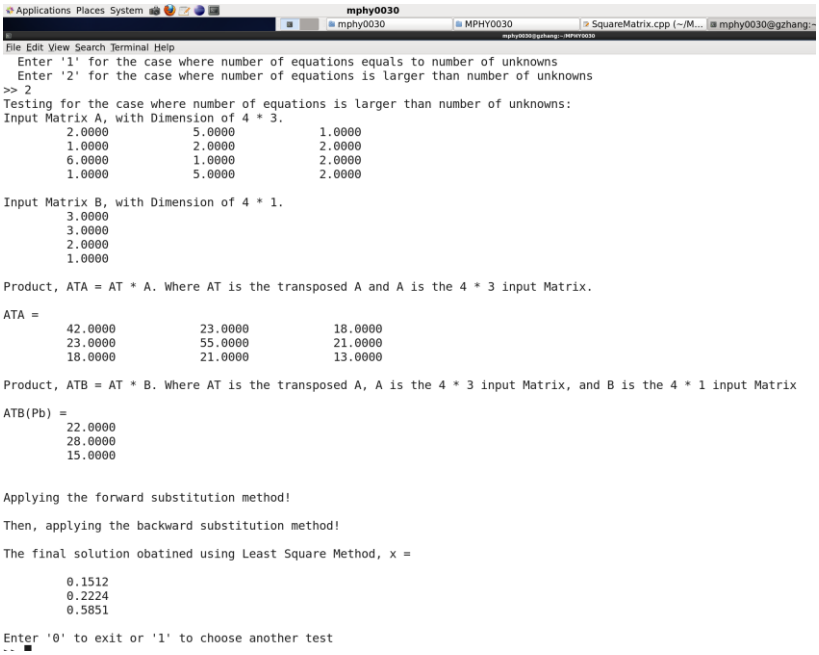
C++ · Tab Width: 4 · Ln 382, Col 8 · INS

```
    for(int i = 0; i < nrow; ++i){
        x[i] = squarex.data[squarex.GetIndex(i,0)];
    }
    return x;
    //Extract x from the SquareMatrix squarex where the final solution is stored in, return the final solution x as a double array
}
//This function employs Least Square Method for solving systems of linear equations. It takes inputs of Matrix A, the product SquareMatrix in the right hand side of =, ATB and the product SquareMatrix of SquareMatrix version of A and its transpose (AT). such that ATA*x=ATB. This function returns the address of the final solution, x, as a double. ncol is for the use in the inside forwardsubstitution. Case: Number of equations > Number of unknowns.


```

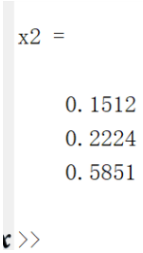
C++ · Tab Width: 4 · Ln 382, Col 8 · INS

(Class code in SquareMatrix for task 2: getATA, getATB, LSQ)

Conclusion: Result of LSQ matches that in Matlab and Matrixcalculator



(Result for task 2 in c++)



(Result in Matlab)