Task1

student number:18043309

The header file for this question is attached as Matrix.h with its source code as Matrix.cpp. The code for testing the function is MatrixTest.cpp. The code is compiled, linked and run using:

```
compile: g++ -c Matrix.cpp MatrixTest.cpp
link: g++ Matrix.o MatrixTest.o -o MatrixTest
run: ./MatrixTest
```

In the header file, I created a Matrix class, included three protected member variables, an protected GetIndex function (determine row-column indexing mapping to 1-D indexing, the return type is int), other public member functions and some non-member but friend functions as following.

For case 1, ConstructorsTesting:

```
1.      //constructors
2.      Matrix (const int noOfRow, const int noOfCols);//standard constructor
3.      Matrix (const Matrix& input);//copy constructor
4.      virtual ~Matrix (); //destructor
5.      //static member funtion for named constructors
6.      static Matrix Zeros(const int noOfRows, const int noOfColumns);
7.      static Matrix Ones(const int noOfRows, const int noOfColumns);
```

`Matrix (const int noOfRow, const int noOfCols):`

Standard constructor uses the number of rows and the number of columns as inputs, and allocates memory dynamically, then initialises each entry to 0. Because the constructor is used to initialise, it does not have return value.

`Matrix (const Matrix& input):`

Copy constructor uses another matrix object to initialise it, so each entry will match with the entry of input matrix. Using Matrix& as argument type, it doesn't need the process of dereference in the function.

`virtual ~Matrix ():`

I used "virtual" for destructor, because the subclass in the task 2 will inherit the Matrix class, so as the base class, the destructor should be virtual then objects can be destroyed in a order, firstly derived object then base. Otherwise only base object will be deleted and then there will be memory leak for derived object.

`static Matrix Zeros(const int noOfRows, const int noOfColumns)`
`static Matrix Ones(const int noOfRows, const int noOfColumns)`

Named constructors use appropriate identifiers to indicate its role and behave like constructors, Zeros is used to set each entry of matrix to 0, Ones is used to set to 1. They are static member functions so that they are not associated with each object instance of the class.

For case 2, AssignmentTesting:

```
1.    //assignment operator
2.   Matrix& operator= (const Matrix& rhs);
3.   //operator overloading to output the matrix
4.   friend ostream& operator<< (ostream& out, const Matrix& matrix);
```

```
Matrix& operator= (const Matrix& rhs)
```

I used Matrix& as return tpye, it will return *this, every object can use “this” pointer to access its own address, so after assignment operation, the left hand side matrix will be assigned values by right hand side matrix. In the test file, I considered three cases: when the size of the rhs matrix is the same/ smaller / larger compared with the lhs matrix. In three cases, the left hand side matrix will be the same with right hand matrix after using assignment operator “=”.

```
friend ostream& operator<< (ostream& out, const Matrix& matrix)
```

The function is for overloading output operator “<<”, it is non-member but friend function, “friend” keyword can grant open access non-member function. The return type is ostream&, after overloading “<<”, it can be used to ouput matrix directly.

For case 3: TeoplitzTesting

```
//create Toeplitz matrix
static Matrixz Toeplitz(const double*column,const int noOfRows,const double*row,const int noOfColumns);
//print function
static void Print(const double* column, const int noOfRows,const int noOfColumns)
```

```
static Matrixz Toeplitz(const double*column,const int noOfRows,const double*row,const int noOfColumns);
```

This function is for creating the Teoplitz matrix, when the input dimensions and arrays of column and rowc are given, it will return a toeplitz matrix of Matrix type. I used static keyword so that it does not need an object to call the function, so the syntax as following will work in the test file.

“Matrix toeplitz = Matrix::Toeplitz(column, noOfRows, row, noOfColumns);”

```
static void Print(const double* column, const int noOfRows,const int noOfColumns)
```

Print function is used to print the arrays of column and row which are used as inputs of Toeplitz function, and print the expected array to justify if the toeplitz matrix is correct compared to matlab output. The function does not have return type, it uses “cout” to print arrays inside the function.

For case 4:TransposeTesting

```
1.  //static matrix transpose
2.  static Matrix Transpose(const Matrix& matrix);
3.  //non-static matrix transpose
4.  virtual Matrix& Transpose();
```

```
static Matrix Transpose(const Matrix& matrix)
```

In static Transpose function, it will return the matrix which has transposed the input matrix and the type of it is Matrix, so in the test file, the syntax “ Matrix transpose = Matrix::Transpose(matrix);” will work and the transpose matrix can be printed directly by “cout<<transpose<<endl;”

```
virtual Matrix& Transpose()
```

In non-static Transpose function, I used Matrix& as the return type, so after the object call the function in the test file, it will be changed into transpose matrix of itself. So after “matrix.Transpose()” we can output the matrix as “cout<<matrix<<endl”, it will show the matrix which has been transposed. In addition, because the subclasss in task2 will use transpose function as well, so I used “ virtual” keyword to implement polymorhism.

For case 5:MultiplicationTesting

```
1.  //a member function of multiplication assignment operator
2.  Matrix& operator*= (const Matrix& rhs);
3.  // a non-member function but a friend for "*" operator overloading
4.  friend Matrix operator * (const Matrix& lhs,const Matrix rhs);
```

`friend Matrix operator * (const Matrix& lhs,const Matrix rhs);`

For overloading operator "*" , I used "friend" keyword, which as a means to grant open access to non-member functions, after overloading, it can implement the multiplication of two matrices.

`Matrix& operator*= (const Matrix& rhs);`

The implementation of the operator*= is based on operator "*", I used Matrix& as return type, in the function, I created a matrix as the product of lhs and rhs and return the product matrix using (*this), so when I write " lhs *= rhs" in the test file, the llhs will become the product of "lhs*rhs".

For case 6: RowColumnExchangeTesting

```
1.  //exchange rows or columns of matrices
2.  Matrix ExchangeRows(int row1,int row2);//exchange the whole rows
3.  Matrix ExchangeRows(int row1,int row2,int column1,int column2);
4.  Matrix ExchangeColumns(int column1,int column2);//exchange the whole columns
5.  Matrix ExchangeColumns(int column1,int column2,int row1,int row2);
```

`Matrix ExchangeRows(int row1,int row2) / Matrix ExchangeColumns(int column1,int column2)`

The implenment of exchanging two rows/columns of the matrix, it will return the matrix which has exchanged rows/columns.

`Matrix ExchangeRows(int row1,int row2,int column1,int column2)`
`Matrix ExchangeColumns(int column1,int column2,int row1,int row2)`

The implenment of exchanging two rows/columns of the matrix but only for the specified columns/rows, it will return the matrix which has exchanged rows/columns.

For case 7: OtherTesting

```
1.  //accessors
2.  int GetEntry(int row,int column)const;//get the entries
3.  int GetNoOfRows()const;//get the number of rows
4.  int GetNoOfColumns()const;//get the number of columns
5.
6.  //The function that sets every entry to zero
7.  Matrix& Zeros();
8.  //The function that sets every entry to one
9.  Matrix& Ones();
```

`int GetEntry(int row,int column)const`

The function can get the entries of the matrix, I used "data[GetIndex(row,column)]" as return value so the return type is int. Because the values can not be changed, I used "const" keyword.

`int GetNoOfRows()const / int GetNoOfColumns()const`

The function can get the number of rows/columns of the matrix, I used "this->noOfRows"

/“this->noOfColumns” as the return value and the return type is int, so when objects(zeros) call the function in the test file, the values will be printed by “cout<<zeros.GetNoRows()” / “cout<<zeros.GetNoColumns()”.

`Matrix& Zeros() / Matrix& Ones()`

The function sets every entry to zero/one. I used Matrix& as return type and I used *this to get the number of rows/columns for loop interation, and then return *this, so the values of each entry will be reassigned for the object matrix.

Task 2

1. Foundation of Implement

1.1 Introduction of all files

For this task, I created a subclass SquareMatrix class which inherited Matrix class, the header file is SquareMatrix.h, the source file is SquareMatrix.cpp file, and the test suite SquareMatrixTest.cpp. In the test file, I used Matrix1 to Matrix7 to number matrices as same as the provided Matlab script. The cases can be chosen from 1-9, cases 1-7 are for task 2, cases 8-9 are for task 3. In addition, Matrix.cpp should be linked.

```
compile : g++ SquareMatrix.cpp SquareMatrixTest.cpp Matrix.cpp -c
link : g++ SquareMatrix.o SquareMatrixTest.o Matrix.o -o SquareMatrixTest
run : ./SquareMatrixTest
```

1.2 Introduction of functions in the header file

For subclass SquareMatrix class, it inherited member variables from base class Matrix, and I used noOfRows and noOfCols to represent the number of rows and the number of columns of the object matrix respectively. In addition, in terms of square matrix, so the inputs of noOfRows and noOfCols will always be the same, which are both equal to the dimension.

Because constructors can not be inherited, I defined its own standard constructor and copy constructor in header file, but in the source file, I used syntax as following to minimise code duplication. Also for named constructor functions “ones”, the subclass object called the “ones” function from base class to maximise code re-use, and defined “eye” function in SquareMatrix class.

```
1.  SquareMatrix::SquareMatrix(const int noOfRow, const int noOfCols):Matrix (noOfRow, noOfCols){  }
2.  SquareMatrix::SquareMatrix(const SquareMatrix& input):Matrix(input){  }
```

In order to implement a set of functions like provided Matlab script, in the header file, I defined functions “triu”, “tril”, “lu1” (for algorithm 2.1) , “lu2” (for algorithm 2.2). But for transpose function, the subclass object called the static “Transpose” function from base class, and for assignment operator “=”, I used syntax “using Matrix ::operator =” to maximise code re-use.

In addition, I defined toeplitz function in the SquareMatrix, because I found that the subclass object can not be assigned by base class object, which means if I created a SquareMatrix object, it can call the toeplitz function from Matrix class, but it can not be assigned by the return value. For example, there is the syntax I wrote and the error I got:

```
SquareMatrix matrix = Matrix::Toeplitz(vec1,4,vec2,4);
   [mphy0030@gzhang Part1]$ g++ SquareMatrix.cpp SquareMatrixTest.cpp Matrix.cpp -c
   SquareMatrixTest.cpp: In function ‘void ToeplitzTesting()’:
   SquareMatrixTest.cpp:74: error: conversion from ‘Matrix’ to non-scalar type ‘SquareMatrix’ requested
```

So in order to use the toeplitz matrix easierly, I created two static toeplitz functions in the SquareMatrix class, contained one input matrix and two input matrices respectively.

2. Analysis of Algorithm

2.1. Algorithm 2.1: Gaussian Elimination without Pivoting

This algorithm fails when any of the pivots is equal to zero, because there is a step "$L_{ik} = U_{ik} / U_{kk}$", if any of the pivots is equal to zero, which means $U_{kk} = 0$, but 0 can not be the divisor. So in my programme, I used "if" statement to check the input. Within the lu1Testing() in the SquareMatrixTest.cpp, I used the matrix that has pivots equal to zero to do the test, which can print warning statement in console properly.

```
1.   if (umatrix1.data[GetIndex(i,i)]==0){
2.     cout << "can't use this meathod,because some of the pivots are equal to zero" << endl;
3.     return;}
```

2.2. Algorithm 2.2: Gaussian Elimination with Partial Pivoting

If input matrix is singular matrix, it is not invertible, and its determinant is 0. But an invertible matrix is a necessary condition for LU decomposition.

This is how I handled it for the first Algorithm: I found out that if the input matrix is singular, there will be zero or NAN (not a number) entrice in the lower triangular matrix or upper triangular matrix after the LU decomposition. Therefore, I added "if" statement in the "lu1" function to check, when it does have zero or NAN, the program will give warning and then end directly. Otherwise it will print the correct result for non-singular matrix.This is the output:

```
Case3:when the input matrix is singular

the input matrix =

 4.0000 -2.0000  1.0000
-5.0000  6.0000  0.0000
 7.0000  0.0000  3.0000

please enter non-singular matrix
```

3.Testing of correctness

For the function ones and eye, I initialised a 4*4 matrix of SquareMatrix type, assigning them by call ones/eye function, It worked properly.

For toeplitz, transpose, triu and tril functions, followed the matlab script, I used vec1= [4,3,2,1], vec2=[1,2,3,4] to create toeplitz matrix3 (input matrix is vec1) and toeplitz matrix4 (inputs are vec1 and vec2), and then use matrix4 as the input to test transpose, triu and tril functions. They all worked correctly.

For the lu1 and lu2 functions, I needed to return multiple matrices. So I used "void" as return type, initialzing three matrices (lmatrix, umatrix, pmatrix) in the test file and then pass them into the lu1/lu2 function as arguments. At the end of the function, I reassigned these matrices by proper matrices (which have been assgined through the algorithm). Therefore, in the test file, after calling the lu1/lu2 function, the L,U,P matrices will be printed correctly (It is the same result compared with matlab).

In addition, in the computer system, almost all real numbers are mapped to a set of sparse floating point numbers in a floating-point system, which will produce round-off errors if there is no precision control statements. Therefore, I added "cout.precision(4)" and "cout.setf(ios::fixed)" to set precision. But I also found out even if I set these inside the lu function, it still affected other functions. For example,when I first test ones function, the output is as left side, but after I test "lu" function and then test ones function, it became as right side. So if we want the output of test case 1-5 without decimal, we should rerun it after we test case 6-9.

```
Testing the function ones and eyes
Case 1:Creating a 4x4 matrix of ones with the ones:
matrix1 =
 1       1       1       1
 1       1       1       1
 1       1       1       1
 1       1       1       1
```

```
Testing the function ones and eyes
Case 1:Creating a 4x4 matrix of ones with the ones:
matrix1 =
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000  1.0000  1.0000
```

Task 3

1. Basic description

I implemented the applications in the same header file and source file with task 2. I added "forwardSubstitution" function and "backSubstitution" function for sovling linear equations. The testing is case 8 (when Number of equations = Number of unknowns) and case 9 (when Number of equations > Number of unknowns) in the SquareMatrixTest.cpp.

2.Explaination of algorithms

2.1.Algorithm 3.1:Forward substitution

When Number of equations = Number of unknowns, the system of equations can be written as **Ax=b**, we can use the LU decomposition, so it will be **LUx=Pb**.

In my implementation, the forward substitution solves the "y" for equation **Ly=Pb**.

First, I used the lmatrix from "lu2" function to represent "L", the "Pb" which called a "b_permutation" function to implement the multiplication of "P"(pmatrix from "lu2" function) and "b", represented the right hand side "Pb" of the equation and the dimension "m" of matrix A , they will be passed into "forwardSubstitution" function as arguments.

Then the algorithm calculated the first unknown, and then by looping iteration, it solved every unknown stored in "y" array, and then it will be returned by "forwardSubstitution" function as the return value.

2.2.Algorithm 3.2:Back substitution

After I obtained "y" from "forwardSubstitution" function, I used it as argument of "backSubstitution" function to solve "x" in equation **Ux=y**.

First, I used the umatrix from "lu2" function to represent "U", "y" from "forwardSubstitution" function, and the dimension "m" of matrix A as arguements of "backSubstitution" function.

Then the algorithm calculated the last unknown, and then by looping iteration, it solved every unknown stored in "x", I defined "x" as an dynamic array of double type, after assignment of it, I used "cout" to print the solution at the end of the "backSubstitution" function, which is the solution for linear eqution.

2.3.Least square solution

When Number of equations > Number of unknowns, according to least square solution, $A^T Ax = A^T b$, in this case, I created "lss_ATA" and "lss_ATB" function in the Matrix class to get $A^T A$ (which is $A^T * A$) and $A^T b$ (which is $A^T * b$) respectively to implement least square solution. $A^T A$ will be a square matrix, so put it into the previous algorithm to get the least square solution of the linear system.

3.Justify the testing approach.

For Number of equations = Number of unknowns, there are two test cases in "applicationTesting1()" function. I used toeplitz matrix3 and matrix4 I got before as coefficient matricies to test respectively. I gave array "b[4]={3,3,2,1}" as the right hand side of linear equations, and then called the "forwardSubstitution" and "backSubstitution" function to get the result. In order to test the correctness, I used online matrix caculator and the "linesolve" function of matlab to justify it.

For Number of equations > Number of unknowns, there are two test cases in "applicationTesting2()" function. I gave two random 4*3 coefficient matricies. The first one doesn't need swapping in LU decomposition. The second one requires swapping.

During the testing process, I gave at least 10 different coefficient matrices to caculator different linear equations, and then compared with the matlab "linesovle" function and the online matrix caculator. There were all correct. In the test suite, I chose two typically cases to print the results properly for my program. Also print the expected solution caculated by online matrix caculator as the comparison.