MPHYGB24 C/C++ Coursework 2 - Parts 2 & 3 - Written Report

PART 2

The `SquareMatrix` class (which inherits from the `Matrix` class) does not define any additional class member variables, and implements all of the functionality provided in `example.m`. The class maximises code re-use and minimises code duplication by making as much use of the functions from the class `Matrix` as possible.

Where necessary, however, new functions had to be implemented in the `SquareMatrix` class for algorithms which had not been previously defined in the `Matrix` class. These included the eye constructor, the triangular matrix extraction functions, and the two Gaussian elimination algorithms. Two static functions were added to the `Matrix` class: one to enable the construction of matrices whose individual entries can be specified in `int main` (the `Test` function), and another to replicate the default MATLAB command window output style with fixed 4.d.p. accuracy for doubles with decimal places (the `PrintMATLAB` function).

A. Ones and Eye constructors

All 8 of the constructors declared and implemented in the `SquareMatrix` class directly invoke their relevant base class functions, apart from the eye constructor (which has no analogous relation within the base class). This is done by inputting the constructor arguments as `Matrix::Zeros(dim,dim)`, rather than `Matrix::Zeros(noOfRows, noOfCols)`, for example, and converting the returned `Matrix` object to a `SquareMatrix` object using the second default copy constructor. When invoked, this version of the copy constructor calls the private `checkSquare` function, which displays an error message and exits the program if the user attempts to copy a `Matrix` object containing a non-square matrix into a `SquareMatrix` object. All of these constructors (apart from the default constructor) are tested in this section of the test suite, and are capable of correctly producing both `matrix1` (case 3) and `matrix2` (cases 4 and 5), as detailed in the MATLAB script `example.m`.

B. Toeplitz functions

These two functions were implemented in a very similar fashion to the above constructors. They invoke their base class analogue using `Matrix::Toeplitz (row, dim, row, dim)` when only the first row is specified, and using `Matrix::Toeplitz (column, dim, row, dim)` when both the first row and the first column are specified. This ensures that these constructors can only produce square matrices. Both of these functions are tested in the test suite; the latter function is tested twice (cases 2 and 3) to ensure that a warning message is *only* displayed when the first elements of the first row and first column are not equal. Case 1 correctly produces `matrix3`, and case 3 produces `matrix4` from `example.m`.

C. Transpose functions

Both of the base class functions are reused here, and it was not necessary to declare any new functions in class `SquareMatrix`. The self-modifying base class function is tested in the test suite (case 1), as is the base class static transpose function (case 2). This emphasises the polymorphic nature of a `SquareMatrix` object, which can behave as either a `SquareMatrix` or a `Matrix` object dependent on context.

D. Upper and lower triangular extraction functions

These two functions were relatively straightforward to implement. They are invoked by a `SquareMatrix` object, and return a new `SquareMatrix` object in which all entries below or above the main diagonal of the invoking matrix have been converted to zeros, respectively. This is demonstrated in the test suite: case 1 produces `matrix6`, and case 2 produces `matrix7`. These functions, as well as others in the `SquareMatrix` class, made use of the `GetIndex` function from the base class.

E. Algorithm 2.1 - Gaussian elimination without pivoting

At the beginning of this algorithm, the upper triangular matrix, U, is initialised with the input matrix, A. The entries on the main diagonal of U (denoted by U_{kk} in this context) at any point throughout the algorithm are known as the pivots. If U_{kk} is ever zero, then line 5 of the pseudo-code in the question will result in division by zero. This will result in the production of `±infs` and subsequent `±NaNs`, resulting in failure of the algorithm. This can be forced to occur by setting $A_{0,0}$ (and therefore initialising $U_{0,0}$) to zero on the first iteration of the outer for loop (case 3).

But this will not necessarily occur when any of $U_{1:m-1,1:m-1}$ is initially set to zero, as the pivots are adjusted from their non-zero value by the first and subsequent iterations of the outer for loop. This is demonstrated in case 2, which shows that this algorithm is robust to having all but the first of its main diagonal entries set to zero.

Therefore the function (`LUDecompositionOne`) returns early if at any point in the algorithm $U_{kk} = 0$. In this situation an empty `SquareMatrix` object is returned (thus also testing the default constructor), an error message is displayed, and the program does not crash. This error message is tested in the test suite in case 3. With regard to checking the correctness of this algorithm, the multiplication member function from the base class is invoked to calculate L * U, which should be equal to A. This is confirmed in cases 1 and 2. Case 1 also provides the same values for `lmatrix1` and `umatrix1` that are produced by execution of `example.m`, which demonstrates that this algorithm has been correctly implemented.

This function has an input argument called `output`, which determines which object the function returns when it is invoked. This technique was also made use of in other functions which are required to return one of multiple objects, including the functions `LUDecompositionTwo` and `LeastSquaresHelper`. This has the disadvantage that the same function is invoked more than once for multiple object output, but this technique seemed to be the most straightforward way to allow a function to one of a range of objects.

F. Algorithm 2.2 - Gaussian elimination with partial pivoting

Unlike algorithm 2.1, this algorithm can handle setting $A_{0,0}$ (and therefore initialising $U_{0,0}$) to zero. This is because the non-singularity of A ensures there is a non-zero pivot in each column, which can be placed on the main diagonal by carrying out the appropriate row swap (this row swap was achieved by reusing both of the overloaded variants of the `ExchangeRows` function from the base class). This is shown in case 3, in which *all* diagonal entries of U are initialised to zero.

This robustness can be extended to there being all but one zero entries in the first column of A and all zeros on the main diagonal (case 4), as the matrix is still non-singular. However, when all entries in the first column and main diagonal are set to zero, the resultant singularity of the matrix is a deal-breaker and $U_{k,k}$ becomes zero in the first iteration of the outer for loop, resulting in the failure of the algorithm. The implementation of this algorithm handles this possibility appropriately in a similar fashion to algorithm 2.1. The error message produced in this situation is tested in case 5.

With regard to the correctness of the implementation, the test suite calculates the same LUP decompositions as `example.m` (as is demonstrated in cases 1 and 2), and also calculates and displays P * A and L * U in cases 1, 2, 3 and 4, so that the user can visually confirm that the two are equal.

PART 3

The functions for this part were implemented partially in the `Matrix` class and partially in the `SquareMatrix` class, as detailed below.

G. Algorithm 3.1 – Forward Substitution

This algorithm is a straightforward way to solve for y from the expression L * y = b, when L is a lower triangular matrix. This is best demonstrated using a mathematical example, as shown below:

$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & -1 & 0 \\ 4 & 1 & -3 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 16 \\ 43 \\ 57 \end{bmatrix}$$

Multiplying the top row of L by y and rearranging shows that y_1 is simply 16 / 1 = 16, which is equivalent to the expression b_1 / L_{11} in line 1 of the pseudo-code. Multiplying the second row of L by y yields the expression $3y_1 – y_2 = 43$, which can be rearranged to give $y_2 = 3y_1 – 43$. This expression is equivalent to the expression $y_2 = [43 – (3y_1)] / -1 = 5$, which is computed in line 3 of the pseudo-code for k = 2.

This is then repeated row by row up to and including the last row of L. For each row an equation with one unknown is formed and solved to yield all the entries of the column vector b in a stepwise fashion. For the sake of completeness, the algorithm computes y_3 as follows: $y_3 = [57 – (4y_1) – (y_2)] / -3 = 4$.

H. Algorithm 3.2 – Backward Substitution

This algorithm is very similar to forward substitution, except that this time the expression U * x = y is solved for x. Here U is an upper triangular matrix, and y is the column vector output of the above forward substitution:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 16 \\ 5 \\ 4 \end{bmatrix}$$

Starting from the bottom row of U, and working upwards row by row to the top row of U, the respective entries of the column vector x can be calculated stepwise.

These two algorithms (3.1 and 3.2) were both implemented successfully in class `Matrix`. Checks within cases 1 of the test suite for each algorithm confirm that L * y = b and that U * x = y, respectively, which indicates that these algorithms have been implemented correctly. Error checking also occurs in these functions (within the scope of class `SquareMatrix`), which ensures that the input matrices L/U are indeed and lower/upper rectangular, that the number of rows in the column vector b/y matches the dimension of L/U, and that none of the diagonal entries of L/U is zero (which would cause the algorithms to fail). If any of these errors is detected, an appropriate error message is displayed (cases 2, 3 and 4).

If these error checks are passed, the function invokes its `virtual` relation in class `Matrix` for calculation of the above two algorithms. This was because class `SquareMatrix` inherits publicly from class `Matrix`, and thus has *access* to the `public` and `protected` member variables of class `Matrix`. The derived class can therefore *get* and *compare* these base class member variables; this is done here using the `GetNoOfRows` base class function (the output of which is the operand of an inequality operator). However, the derived class cannot perform arithmetic operations with these values. This was therefore done within the scope of class `Matrix`. (If the member variables were declared `public` in class `Matrix`, then these functions could have been implemented within the scope of class `SquareMatrix`).

It was important to declare the function prototypes for `ForwardSub` and `BackwardSub` as `virtual` within class `Matrix` in order to control the polymorphic behaviour of a `SquareMatrix` object, which can act as either of type `Matrix` or type `SquareMatrix`. Declaring these two function prototypes as `virtual` within the base class ensures that they are 'hidden', and that any overriding variant in a derived class is preferentially executed when invoked by an object of the derived class.

I. Solving uniquely determined systems of linear equations

This was achieved using the two functions `SolveUnique`. Within the scope of class `SquareMatrix`, this function first performs an error check, before invoking the function `LUDecompositionTwo` in order to decompose A into L, U and P matrices using algorithm 3.2. These three square matrices, together with the column vector b, are then passed to the `Matrix` class version of `SolveUnique`, which solves for y in L * y = P * b, and then solves for x in U * x = b, using class `Matrix` functions `ForwardSub` and `BackwardSub`, respectively. There was no need to declare the `SolveUnique` prototype as `virtual` within class `Matrix`, as the two versions of the function do not have exactly the same signatures. Also the keyword `const` was not used to modify the function argument b, as b is altered in this particular implementation.

Column vector x is then returned to `int main` and displayed as the result of solving for x in A * x = b, before checking that A * x does indeed equal b. This is tested in cases 1 and 2, which confirm that the function has been correctly implemented. Cases 3 and 4 test the error messages of both the `SolveUnique` and `LUDecompositionTwo` functions, which are displayed if the number of columns in b does not match the dimension of A, or if LU decomposition fails due to A being singular.

J. Solving overdetermined systems of linear equations

This was achieved by implementing the function `LeastSquaresHelper`, which was an addition to the original `Matrix` class. This function is invoked in `int main`, and returns the `SquareMatrix` object A' * A and the `Matrix` object A' * b. The number of rows and columns in A' * A are then displayed, to assure the user that they are equal, before displaying both A' * A and A' * b. These two objects are then used by the `SolveUnique` functions, which this time solve for x in (A' * A * x = A' * b). x is then displayed in the screen output, as is the expected result from the same calculation performed in MATLAB. This serves as a check to confirm that the algorithm has been correctly implemented, as is shown in cases 1 and 2. (A * x was not calculated and displayed in this cases, as A * x ≠ b for a least squares solution).

A warning message is shown in the case that the matrix A is square, and advises that an exact solution, not a least squares solution will be provided, and that the `SolveUnique` function should be used instead. This is tested in case 3 (A * x was calculated and displayed in this case, as A * x = b for an exact solution).

Error checking occurs within this function to ensure that the number of rows is greater than the number of columns in A, and that the length of the column vector b matches the number of columns in A. The resultant error messages are tested in cases 4 and 5. (This algorithm will also fail if A is singular, as a singular matrix multiplied by the transpose of itself is also singular, resulting in the subsequent failure of algorithm 2.2).