Karan Patel

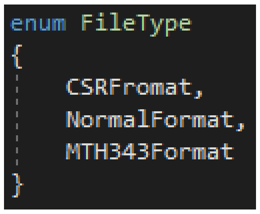
MTH 343

Final Project Report

The first part of the programming project asked us to implement a library of functions containing the following functionalities:

1. Reading in a sparse matrix from a file which contains the i (row number in matrix), j (column number in matrix) and data entries aij for all non-zero entries of the matrix.

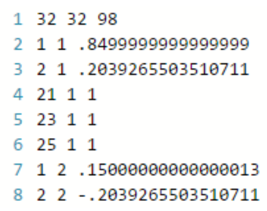
   The piece of functionality described above is implemented in class Matrix_CSR_Format which is located in Matrix_CSR_Format.cs file. My implementation can read matrices in the following 3 formats defined by the enum (which is again located in file Matrix_CSR_Format.cs) below:



```
enum FileType
{
    CSRFromat,
    NormalFormat,
    MTH343Format
}
```

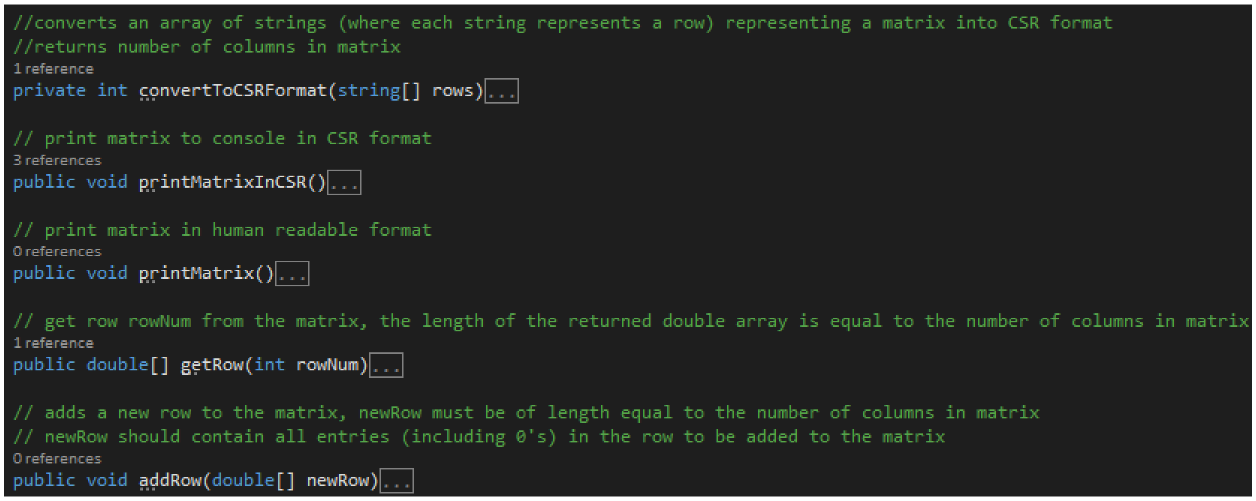
   The format of file (containing matrix data) described above is represented by the FileType.MTH343Format enum constant in my implementation. A sample data file for a matrix stored in FileType.MTH343Format can be found here: https://1drv.ms/t/s!Arc54q14bwOLgeI5OPsTrYCov9Ep1A

   It looks as follows:



```
32 32 98
1 1 .8499999999999999
2 1 .2039265503510711
21 1 1
23 1 1
25 1 1
1 2 .15000000000000013
2 2 -.2039265503510711
```

   The first line contains number of rows, columns and non-zero entries in matrix and the remaining lines contain information about non-zero entries. Matrix data is extracted and stored in Compressed Sparse Row (CSR) format by Matrix_CSR_Format class's constructor. The Matrix_CSR_Format class in Matrix_CSR_Format.cs file is responsible for storing a matrix in CSR format and defining the following functions:



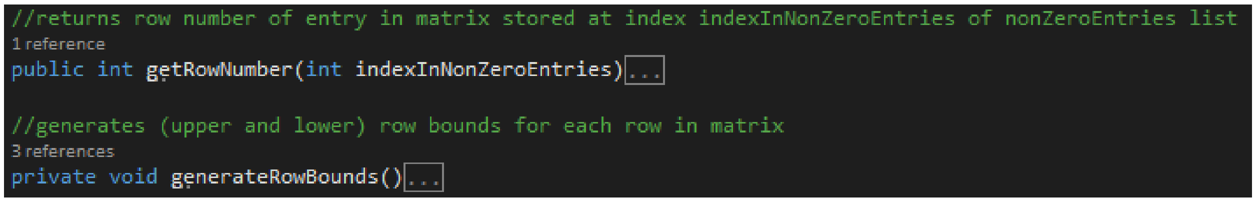
```
//converts an array of strings (where each string represents a row) representing a matrix into CSR format
//returns number of columns in matrix
1 reference
private int convertToCSRFormat(string[] rows)...

// print matrix to console in CSR format
3 references
public void printMatrixInCSR()...

// print matrix in human readable format
0 references
public void printMatrix()...

// get row rowNum from the matrix, the length of the returned double array is equal to the number of columns in matrix
1 reference
public double[] getRow(int rowNum)...

// adds a new row to the matrix, newRow must be of length equal to the number of columns in matrix
// newRow should contain all entries (including 0's) in the row to be added to the matrix
0 references
public void addRow(double[] newRow)...
```



```
//returns row number of entry in matrix stored at index indexInNonZeroEntries of nonZeroEntries list
1 reference
public int getRowNumber(int indexInNonZeroEntries)...

//generates (upper and lower) row bounds for each row in matrix
3 references
private void generateRowBounds()...
```

2. The second part of task 1 was to compute matrix (given in CSR format) times a vector. This functionality is implemented in class Matrix_CSR_Format_Operations (by MatrixTimesComlumVector(Matrix_CSR_Format m, double[] columnVector) method) located in Matrix_CSR_Format_Operations.cs. The function takes in a matrix in CSR format and a vector, computes their product (which is a column vector) and outputs the resulting vector as an array of doubles (64-bit floating-point values with an approximate range of $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$).
3. The third part of task 1 was to implement an algorithm which, given a matrix in CSR format, would efficiently calculate its transpose in the same format. In order to do so, my algorithm first creates dynamically expanding lists of non-zero elements and their corresponding column location for each new row of the matrix. After that, every element in the original matrix is traversed and is added to the correct lists corresponding to its row in the transpose. At the end, all of these lists are merged to a final list and the rowInfo array is created. rowInfo array is populated as follows: rowInfo[0] = 0,

rowInfo [*i*] = rowInfo [*i* − 1] + (number of nonzero elements on the (*i* − 1)-th row in the original matrix). `public static Matrix_CSR_Format Transpose(Matrix_CSR_Format m)` method in Matrix_CSR_Format_Operations.cs is used to compute transpose.

4. The final part of task 1 is to implement an algorithm which, given two matrices In CSR format, computes their product in CSR format. The algorithm is implement by `public static Matrix_CSR_Format Multiply(Matrix_CSR_Format m1, Matrix_CSR_Format m2) method in` Matrix_CSR_Format_Operations.cs. The algorithm is heavily commented so that it can be easily understood. Please take a look at the comments to get a feeling of how it is implemented.

A test driver for all of the fucntions listed above is implemented as `static void demoCSROperations(String dataFilePath)` method located in Program.cs. It first reads in a matrix from a data file, computes its product with a vector, computes transpose of it and multiplies the matrix with its transpose. Resulting vectors and matrices (in CSR format) are printed out to the console. Matrix can be printed out in easy to read format using the printMatrix() method of Matrix_CSR_Format class. Following file (https://1drv.ms/t/s!Arc54q14bwOLgeI7wQieGrbIqH7VdA) contains output obtained from running the test driver on matrix stored in file https://1drv.ms/t/s!Arc54q14bwOLgeI5OPsTrYCov9Ep1A

Task 2

The second half of the programming project instructed us to implement a given GMRES algorithm. The generalized minimal residual (GMRES) method is designed to solve nonsymmetric linear systems. The most popular form of GMRES implemented in this project is based on a modified Gram-Schmidt orthonormalization procedure and uses restarts to control storage requirements. The algorithm is implemented in GMRES.cs file.

The algorithm requires to be passed in the following information:

1. A nonsymmetrix sparse matrix A in CSR format
2. Vector b such that Ax = b
3. A random vector x
4. Tolerance, t
5. Maximum number of iterations, itrMax
6. The number of iterations between each restart, itrBetweenEachRestart

All information listed above needs to be passed on to the `public GMRES(double tolerance, int maxIterations, int restartAfterEveryNIterations, Matrix_CSR_Format A, double[] b, double[] initialX)` constructor of GMRES class.

The GMRES class contains the following two important methods:

1. `private double[] solveEachRestart(ref bool convergenceReached)`

   The method above computes residuals for iterations between each restart. Marix P (containing vectors orthogonal to all previous vectors) as well as matrix B (which is factorized into QR) both are discarded at the end of method.

2. `public static void solveUsingGMRES(double tolerance, int maxIterations, int restartAfterEveryNIterations, Matrix_CSR_Format A, double[] b, double[] initialX)`

   This method calls the `solveEachRestart` method described above until either the following happens: convergence has been reached i.e. residual is within tolerance t, current iteration equals the maximum number of iterations, itrMax.

The algorithm prints out residual for each iteration required to reach convergence (if it is possible to converge in the first place). The time and number of iterations taken to converge are also printed out. The test driver for task 2 is `GMRES_demo(String dataFilePath) in Program.cs`

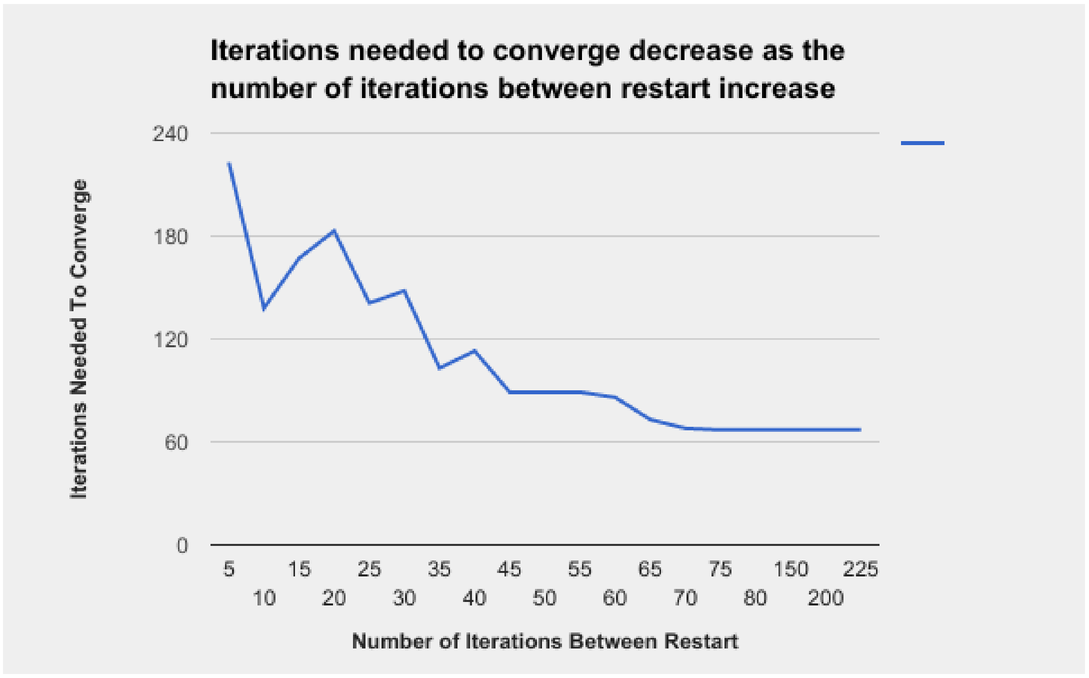
I tested out my implementation of the GMRES algorithm on a real unsymmetric sparse matrix with 225 rows, 225 columns and 1065 non-zero entries. The data file for matrix can be found here: https://1drv.ms/t/s!Arc54q14bwOLgeI6PxQFMxwPPBSDyQ. Vector b was computed by multiplying Matrix A (In CSR format) with a vector (each entry in vector was 75). Tolerance was set to 10^-6. Vector x or initial_X was set to a 0 vector. Maximum number of iterations or itrMax was set to 225. The number of iterations between each restart, itrBetweenEachRestart was set to multiples of 5 (i.e. 5, 10, 15, …) for each run.

Following are the results on running the algorithm with varying itrBetweenEachRestart:

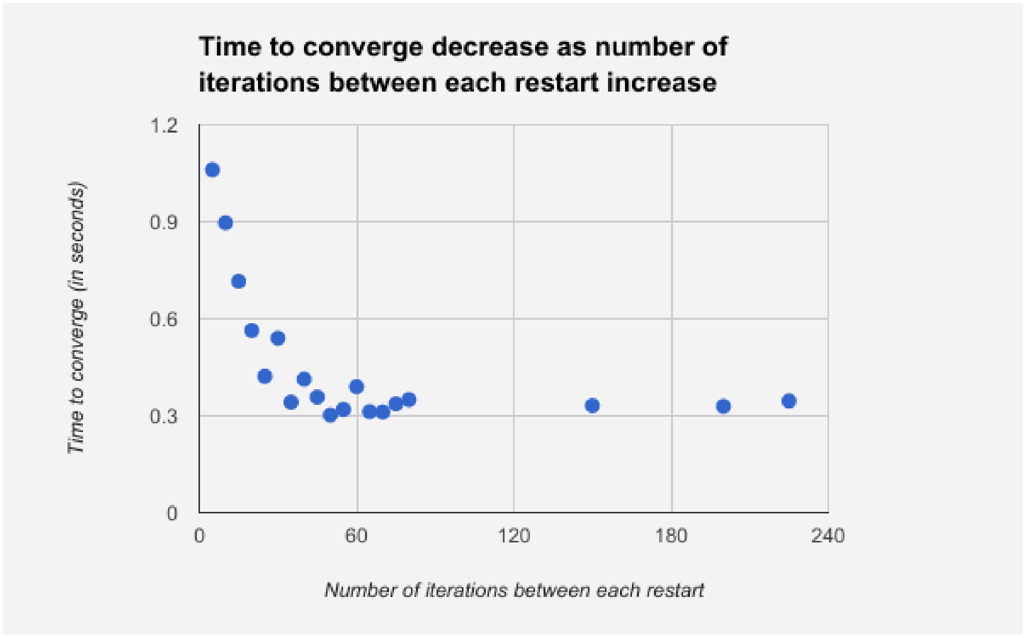
Note: The data table attached below can also be obtained at https://1drv.ms/x/s!Arc54q14bwOLgeI8IWODH-NWgVPVdw. Output from program when itrBetweenEachRestart = 5 can be obtained at https://1drv.ms/t/s!Arc54q14bwOLgeI9KyuISrkhY_wx3A. Output from program when itrBetweenEachRestart = 225 can be obtained at https://1drv.ms/t/s!Arc54q14bwOLgeI-sw6VbBS2zTNbFA.

maxItr	itrBetweenEachRestart	Iterations needed to converge	Time to converge (in seconds)	Resedual	Tolerance
225	5	223	1.0607164	9.64E-06	1.00E-05
225	10	138	0.8965985	9.67E-06	1.00E-05
225	15	167	0.7154875	9.48E-06	1.00E-05
225	20	183	0.5637636	8.45E-06	1.00E-05
225	25	141	0.4222074	8.22E-06	1.00E-05
225	30	148	0.5399833	8.75E-06	1.00E-05
225	35	103	0.3420633	8.74E-06	1.00E-05
225	40	113	0.4133072	7.07E-06	1.00E-05
225	45	89	0.3577183	6.34E-06	1.00E-05
225	50	89	0.3021176	7.84E-06	1.00E-05
225	55	89	0.3196303	8.13E-06	1.00E-05
225	60	86	0.3899375	9.44E-06	1.00E-05
225	65	73	0.3126623	9.76E-06	1.00E-05
225	70	68	0.3116222	8.13E-06	1.00E-05
225	75	68	0.336918	8.13E-06	1.00E-05
225	80	68	0.3498312	8.13E-06	1.00E-05
225	150	68	0.3315073	8.13E-06	1.00E-05
225	200	68	0.3294082	8.13E-06	1.00E-05
225	225	68	0.3456829	8.13E-06	1.00E-05

From the data above, I can conclude the following:



From data table above, we can see that the algorithm needs 68 iterations to converge (assuming no need for a restart). From the graph and data table, we can see that the relationship between number of iterations needed to converge and number of iterations between each restart are inversely proportional.



I can also conclude that time needed to converge decrease as number of iterations between each restart increase.

The main reason for using restart in the algorithm is to address the issue of memory / space complexity. If no restarts are used, GMRES will converge in no more than n steps (assuming exact arithmetic). This is of no practical value when n is large; moreover, the storage and computational requirements in the absence of restarts are prohibitive. The crucial element for successful application of GMRES(m) revolves around the decision of when to restart, i.e. the choice of m.

One major drawback to GMRES is the amount of work and storage required per iteration rises linearly with the iteration count. Unless one is fortunate enough to obtain extremely fast convergence, the cost will rapidly become prohibitive. The usual way to overcome this limitation is by restarting the iteration. After a chosen number of iterations m, the accumulated data are cleared and the intermediate results are used as the initial data for the next m iterations. This procedure is repeated until convergence is achieved. The difficulty is in choosing an appropriate value for m. If m is too small, GMRES(m) may be slow to converge, or fail to converge entirely. A value of m that is larger than necessary involves excessive work (and uses more storage).

Sources and Citations

Matrix data obtained from

http://www.cise.ufl.edu/research/sparse/matrices/list_by_nnz.html

Other sources:

https://en.wikipedia.org/wiki/Sparse_matrix

http://mathworld.wolfram.com/GeneralizedMinimalResidualMethod.html