ECE 7650 ASSIGNMENT REPORT

(Advance Matrix Algorithm)

Author: Jamiu Babatunde Mojolagbe

Department: Electrical and Computer Engineering

Student ID: #7804719

Email: mojolagm@myumanitoba.ca

Course: ECE 7650

Homework: 3 & 4

Sub. Date: December 5, 2016

QUESTION ONE (1)

The main goal here is to implement Breadth First Search (BFS) and Reverse Cuthill-McKee (RCM) ordering algorithms. However, for purpose of compactness and code simplicity, two additional functions were created to assist in this.

The two (2) additional functions are:

1. "**csr.m**": It implements Compressed Sparse Row format of an input matrix A. The parameters to be supplied to the function as well as the return values are shown in the function implementation shown below:

```
function [IA, JA] = csr(A)
%% csr.m
%   Implements Compress sparse row
%       of an input matrix A
%
%       Parameters:
%           A:      A matrix
%       Returns:
%            IA:    Row indices in CSR
%            JA:    Column indices
%
%   Author: Jamiu Babatunde Mojolagbe

    %% set parameters
    [n, m] = size(A);
    IA = ones(1, n+1);

    %% loop through the rows and compile the CSR parameters
    for i = 1:n;
        nzr = nnz(A(i ,:));
        IA(i+1) = nzr;
        if i == 1;  JA = [find(A(i ,:))];
        else JA = [JA find(A(i ,:))];
        end
    end
    IA = cumsum(IA);
end
```

2. "**getLowestDegree.m**": This function gets the index of row of with the lowest degree of freedom of an input matrix A. The required parameters and return values are shown in the presented function implementation shown below:

```
function [r, n, m] = getLowestDegree(A)
%% getLowestDegree.m
%   Get index of the row with lowest degree of freedom, i. e, neighbors
%
%       Parameters:
%           A:   A sparse matrix in compressed-row format with each row sorted
%       Returns:
%            r:     Index of the row with lowest degree of freedom
%            n:     Number of rows in A
%            m:     Number of columns in A
%
    %% set parameters
    [n, m] = size(A);        % obtain the number of rows and cols of A

    %% loop through the rows of A and get the non-zeros
    for i = 1:n
        nz(i) = nnz(A(i, :));
    end
    r = find(nz(:) == min(nz));
end
```

Implementation Note on Breadth First Search (BFS)

Breadth First Search (BFS) algorithm was implemented in the file named "**bfs.m**". Two important issues were noticed that was not in the pseudocode and were careful fixed accordingly in the implementation.

The first issue is the choice of "i" – which is the initial search row index. The question here is "What happens if the choice of 'i' is beyond the maximum number of rows in A?". To answer this question one line of code was added to ascertain that we are not going beyond the maximum number of rows.

Again, the second important issue here is the issue of 'π'. The implemented BFS function is not only returning 'π' but also permutation matrix P. Hence, from proper study of the implementation of LU in MATLAB, it was observed that when built in MATLAB **lu( )** function is instructed to return the permutation matrix P, it returns P as a sparse matrix rather than a dense one. Therefore, in this implementation of BFS, a parameter is added to check if P is to be returned as a dense P or as a sparse one.

The required parameters or arguments for the function and return variable are shown in the function implementation code presented below:

```
function [P, pi] = bfs(A, i, retSparse)
%% bfs.m
%   Implements Breadth First Search for
%       adjacency graph traversal reordering permutation
%
%       Parameters:
%           A: A sparse matrix in compressed-row format with each row sorted
%           i:      An index of the first vertex (row) to start at
%           retSparse: Boolean value whether to return P as sparse matrix
%       Returns:
%            P:     A permutation matrix
%            pi:    A permutation list based on the ordering of the vertices traversed
%
%   Author: Jamiu Babatunde Mojolagbe

    %% obtain the dimension of A inorder to obtain number of rows
    [n, m] = size(A);
    assert(i <= n, 'Specified vertex does not exist');
    pi = [i];
    count = 1;
    marked = zeros(n, 1);
    marked(i) = 1;
    S = [i];
    [IA, JA] = csr(A);

    while count < n
        Snew = [];
        for i = S
            row_start = IA(i);
            row_stop = IA(i+1);

            for j = row_start:row_stop - 1
                column = JA(j);
                if marked(column) == 0
                    marked(column) = 1;
                    Snew = [Snew column];
                    pi = [pi column];
                    count = count + 1;
                end
            end
        end
        S = Snew;
    end
    P = zeros(n, n);
    for i = 1:n
        P(i, pi(i)) = 1;
    end
    if retSparse; P = sparse(P); end
end
```

Implementation Note on Reverse Cuthill-McKee (RCM)

This algorithm was implemented as a function in the file named "**rcm.m**". As with the BFS function presented above, RCM implementation also take care of returning the permutation as 'π' as well as a permutation matrix P. Again, there is also choice whether to return P as a full matrix or as a sparse one. The arguments needed by this function is shown in the implementation code below:

```
function [P, pi] = rcm(A, retSparse)
%% rcm.m
%   Implements Reverse Cuthill-McKee (RCM) Ordering of
%       adjacency graph traversal reordering permutation
%
%       Parameters:
%           A:          A sparse matrix in compressed-row format with each row sorted
%           retSparse:  Boolean value whether to return P as sparse matrix
%       Returns:
%            P:         A permutation matrix
%            pi:        A permutation list based on the ordering of the vertices
traversed
%
%   Author: Jamiu Babatunde Mojolagbe

    %% obtain the dimension of A inorder to obtain number of rows
    [r, n, m] = getLowestDegree(A);
    i = min(r);
    pi = [i];
    count = 1;
    marked = zeros(n, 1);
    marked(i) = 1;
    S = [i];
    [IA, JA] = csr(A);

    while count < n
        Snew = [];
        for i = S
            row_start = IA(i);
            row_stop = IA(i+1);

            % loop over the adjacency nodes
            for j = row_start:row_stop - 1
                column = JA(j);
                if marked(column) == 0
                    marked(column) = 1;
                    Snew = [Snew column];
                    count = count + 1;
                end
            end
        end
        pi = [pi Snew];
        S = Snew;
    end
    pi = fliplr(pi);
    P = zeros(n, n);
    for i = 1:n
        P(i, pi(i)) = 1;
    end
    if retSparse; P = sparse(P); end
end
```

QUESTION TWO (2)

While the main implementation file for this question is "**Q2.m**", however, it has additional two (2) files "**Q2reportDataPloting.m**" and "**Q2visualize.m**". According to the question, the task here is to test the efficiency and/or time taken to solve an input matrix A, when the matrix A is:

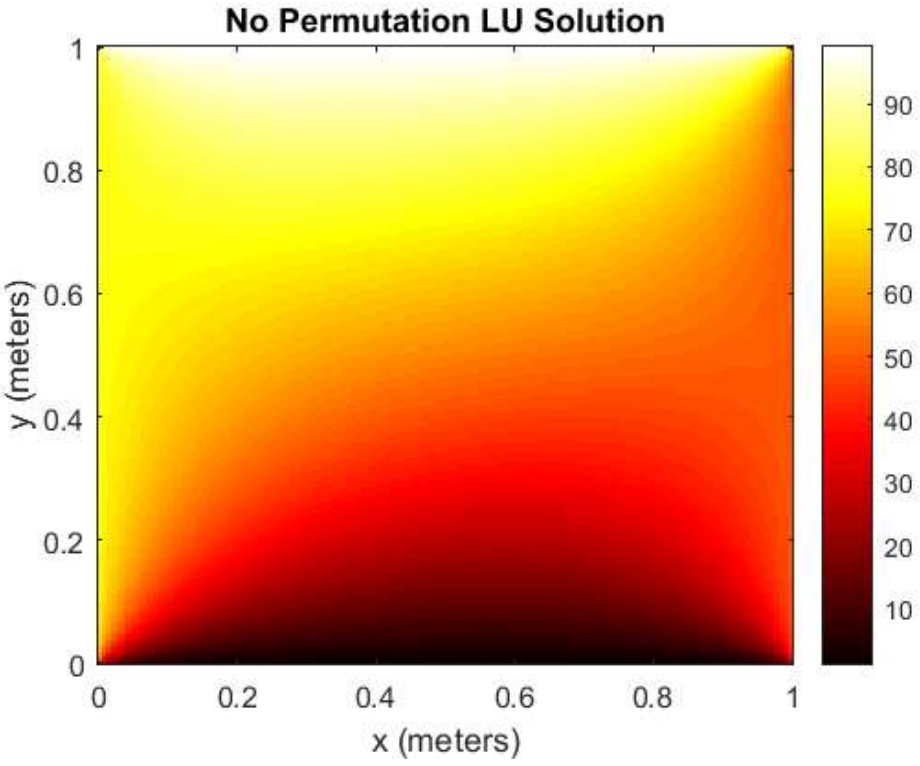
i. Original unpermuted
ii. Bread First Search (BFS) symmetric permuted
iii. Reverse Cuthill-McKee symmetric permuted
iv. MATLAB Built-in Routine permuted

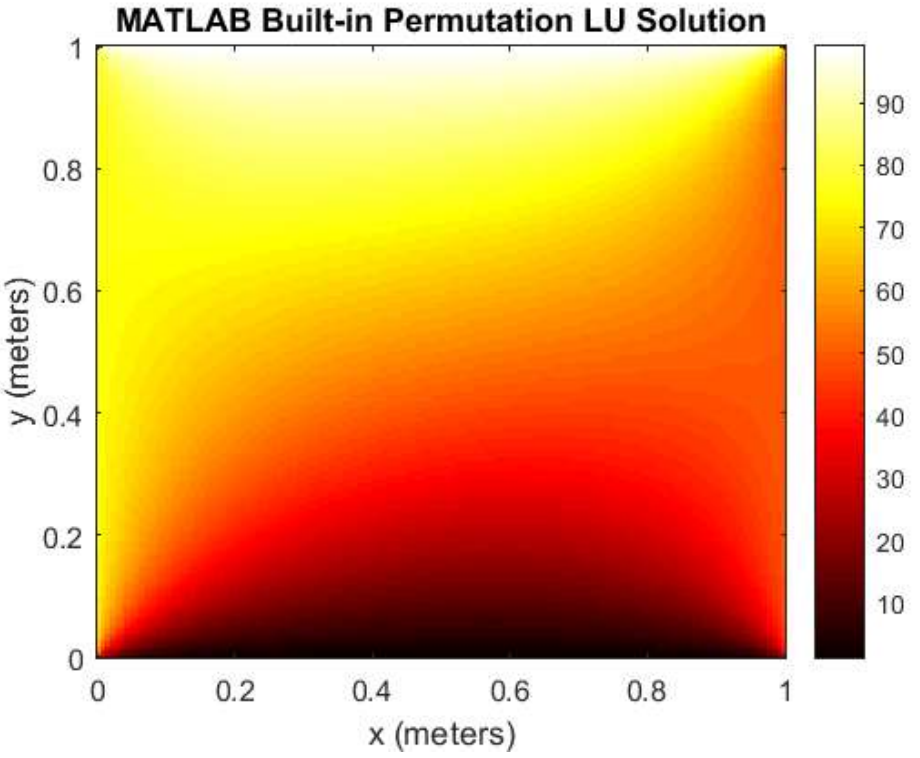
The above four conditions were implemented and test as contained in "**Q2.m**". However, for the purpose of this report, the graph showing the time taken and data used are replicated in "**Q2reportDataPloting.m**" (though, all results can also be obtained from "**Q2.m**"). To show that the implementation works and solutions obtained were correct, the plotting of the solution was done to show the distribution using the given function "**plotLaplaceSolution.m**". The obtained results are shown below:

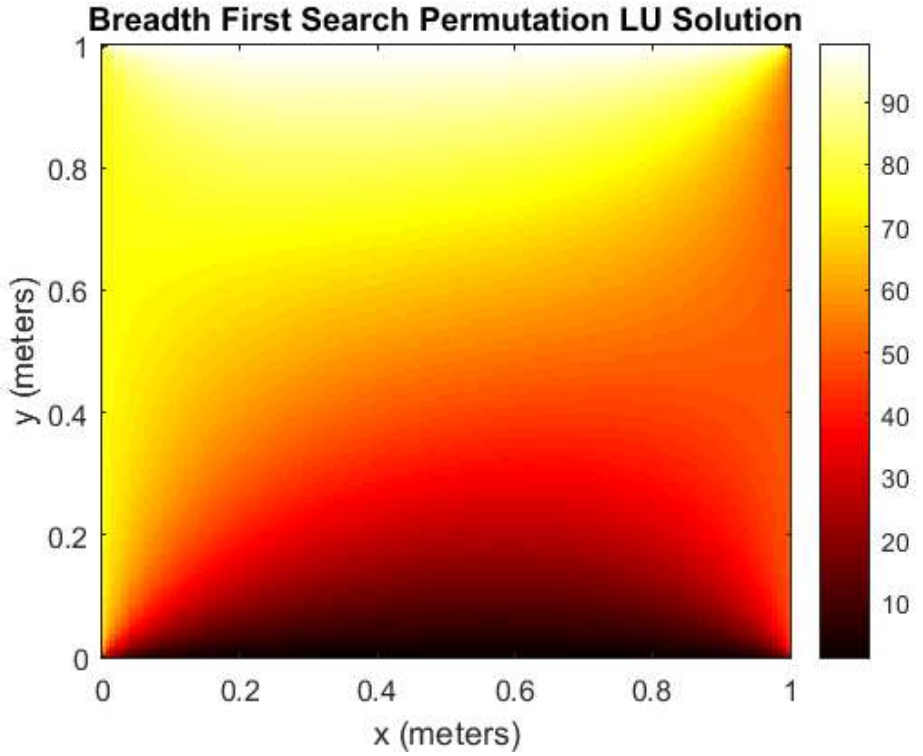
The table showing time taken in seconds for the four conditions tested

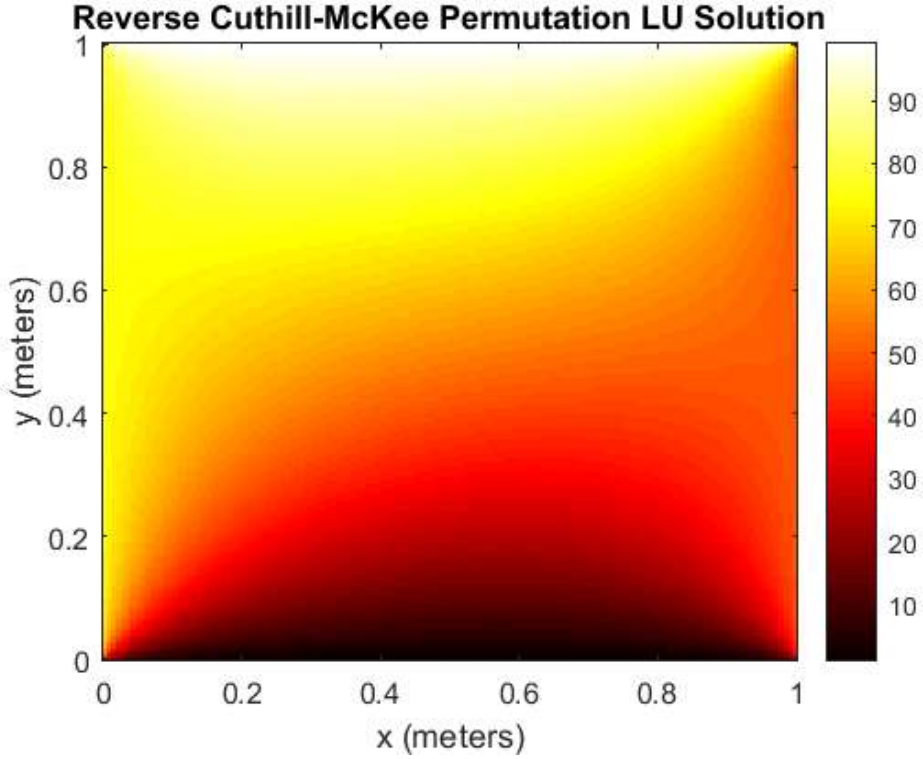
Dim (nxn)	Unpermuted (s)	Built-In Routine(s)	BFS (s)	RCM (s)
100x100	0.001058	0.002111	0.000452	0.000504
400x400	0.002605	0.002769	0.001558	0.001309
900x900	0.008263	0.008247	0.002709	0.003878
1600x1600	0.017850	0.008233	0.006643	0.009283
2500x2500	0.034908	0.019326	0.017465	0.018680
3600x3600	0.055706	0.028997	0.028436	0.030630
4900x4900	0.083240	0.034866	0.050838	0.048135
6400x6400	0.137913	0.045793	0.087834	0.097644
8100x8100	0.217990	0.053236	0.131986	0.141438
10000x10000	0.308947	0.067594	0.202453	0.264267
12100x12100	0.584909	0.112153	0.305484	0.398970
14400x14400	0.650227	0.103781	0.447355	0.370886
16900x16900	0.756885	0.102010	0.508035	0.456723
19600x19600	1.166409	0.173606	0.834445	0.799378

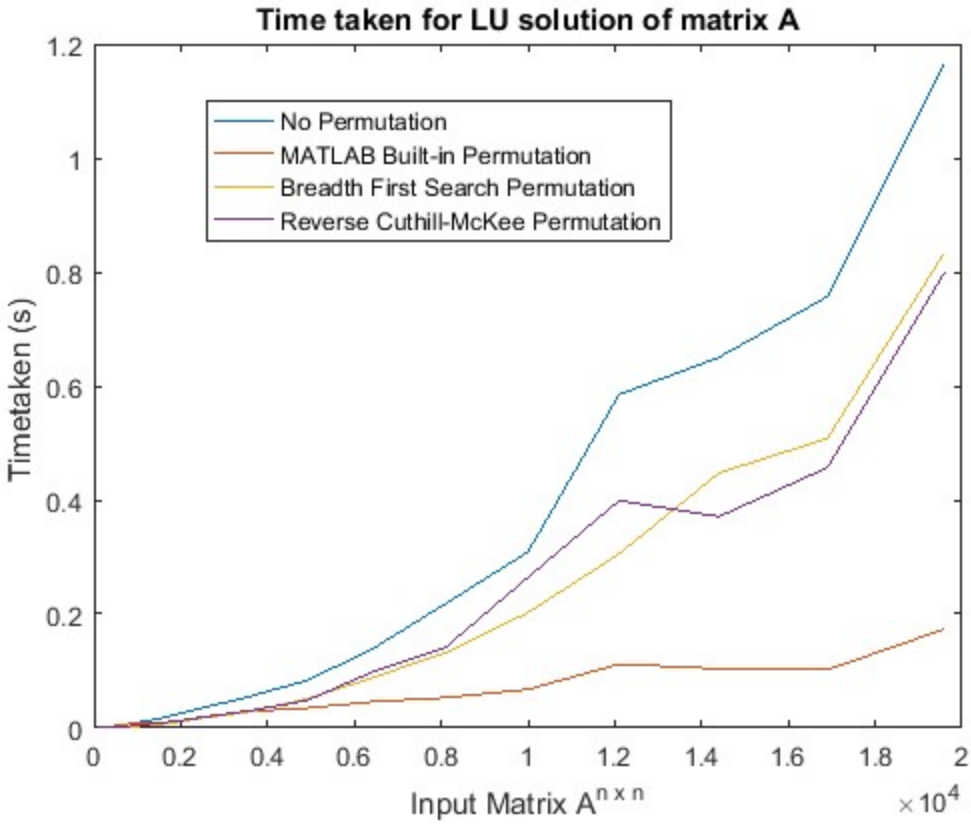
The plot of solutions obtained from the four (4) condition for dimension (19600x19600)



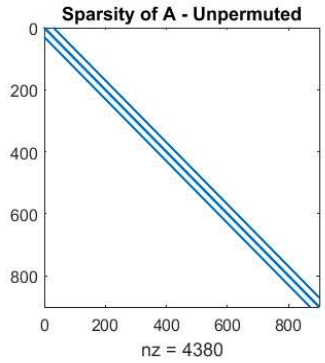


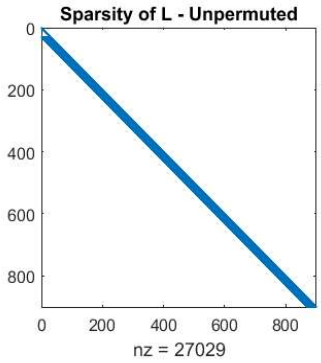

Breadth First Search Permutation LU Solution
y (meters)
x (meters)
0
0.2
0.4
0.6
0.8
1
10
20
30
40
50
60
70
80
90

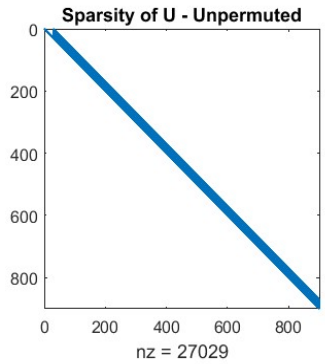

Reverse Cuthill-McKee Permutation LU Solution
y (meters)
x (meters)
0
0.2
0.4
0.6
0.8
1
10
20
30
40
50
60
70
80
90

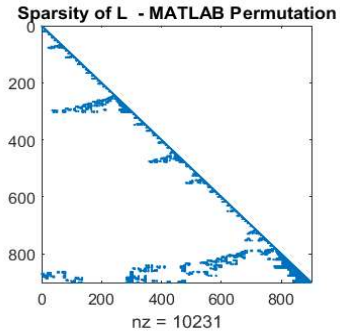


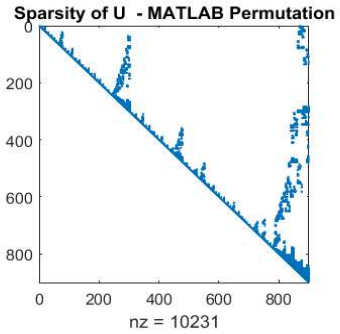
For visualizing the effect of three (3) applied permutation on the input matrix A, file named "**Q2visualize.m**" was used for this purpose and the following graphs were obtained.

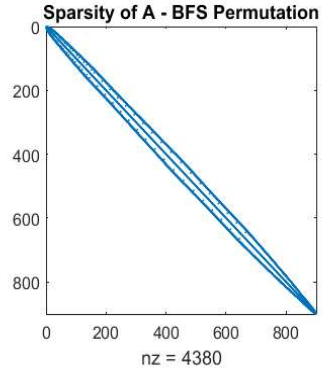


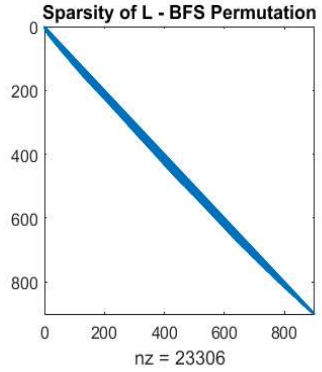


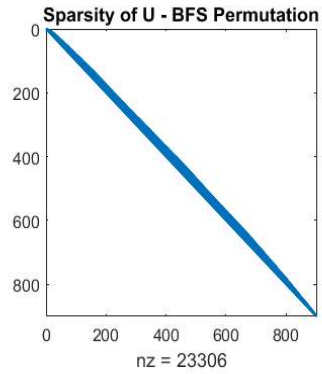


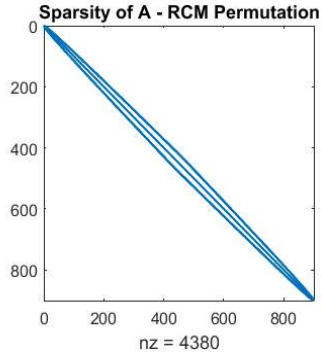

Sparsity of L - MATLAB Permutation
0
200
400
600
800
0 200 400 600 800
nz = 10231

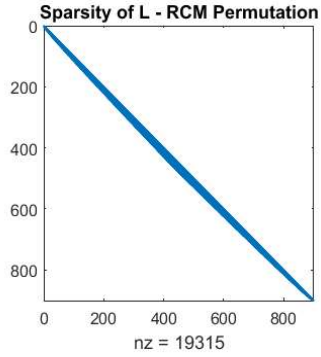

Sparsity of U - MATLAB Permutation
0
200
400
600
800
0 200 400 600 800
nz = 10231

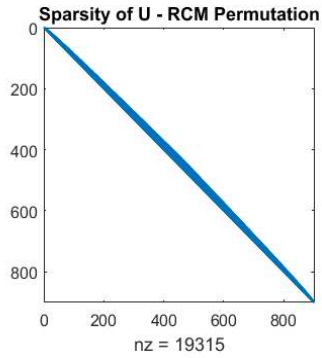

Sparsity of A - BFS Permutation
0
200
400
600
800
0 200 400 600 800
nz = 4380


Sparsity of L - BFS Permutation
0
200
400
600
800
0 200 400 600 800
nz = 23306


Sparsity of U - BFS Permutation
0
200
400
600
800
0 200 400 600 800
nz = 23306







Observation:

From the table and the time taken graph shown above, it can be observed that when the input matrix A was set to lower values there was no consistent behavior of time taken. However, when the matrix dimension was set to high to very high values, it was observed that the time seemed to be consistent, in that, it was clear that the fastest of the permutations is MATLAB built-in routine while the slowest is the original matrix without permutation. Again, observing BFS and RCM, it can be seen that at certain dimensions (1600 - 12100) BFS tends to be faster than RCM, but at dimensions (14400 – 19600) RCM tends to be faster than BFS. In overall, it can be stated that, BFS and RCM time taken are between time taken for the unpermuted and MATLAB built-in routine permuted.

Other important that was observed was that when a certain matrix - obtained from solution of Finite Element Method solution of a square plate - was subjected to both RCM and BFS, it was observed that both algorithm enter into infinite loop and thus RCM and BFS was not applicable. This showed that both algorithms can only be used for mainly diagonally dominant matrices. The matrix used is contained in folder named '**data**'.

In conclusion, permuting a matrix makes its LU solution faster and saves time.

QUESTION THREE (3)

Block Jacobi iterative technique was implemented as a function in the file named "**blockJacobi.m**" while Gauss-Seidel was implemented in the file "**blockGaussSeidel.m**". The driver programs for this question are "**Q3.m**" and "**Q3Driver.m**".

Implementation Note

As contained in the question, the two algorithms were implemented such that no overlap would occur. This was done such that it carefully handles uneven block sizes. The parameters to be supplied to the function and the return values from the functions are contained in the implementation files because the functions are fully documented for ease of use.

While "**Q3Driver.m**" was used to confirm that the implementations of the two algorithms – Block Jacobi and Block Gauss – work, "**Q3.m**" contains comparison between the two algorithms when in input matrix is permuted or not.

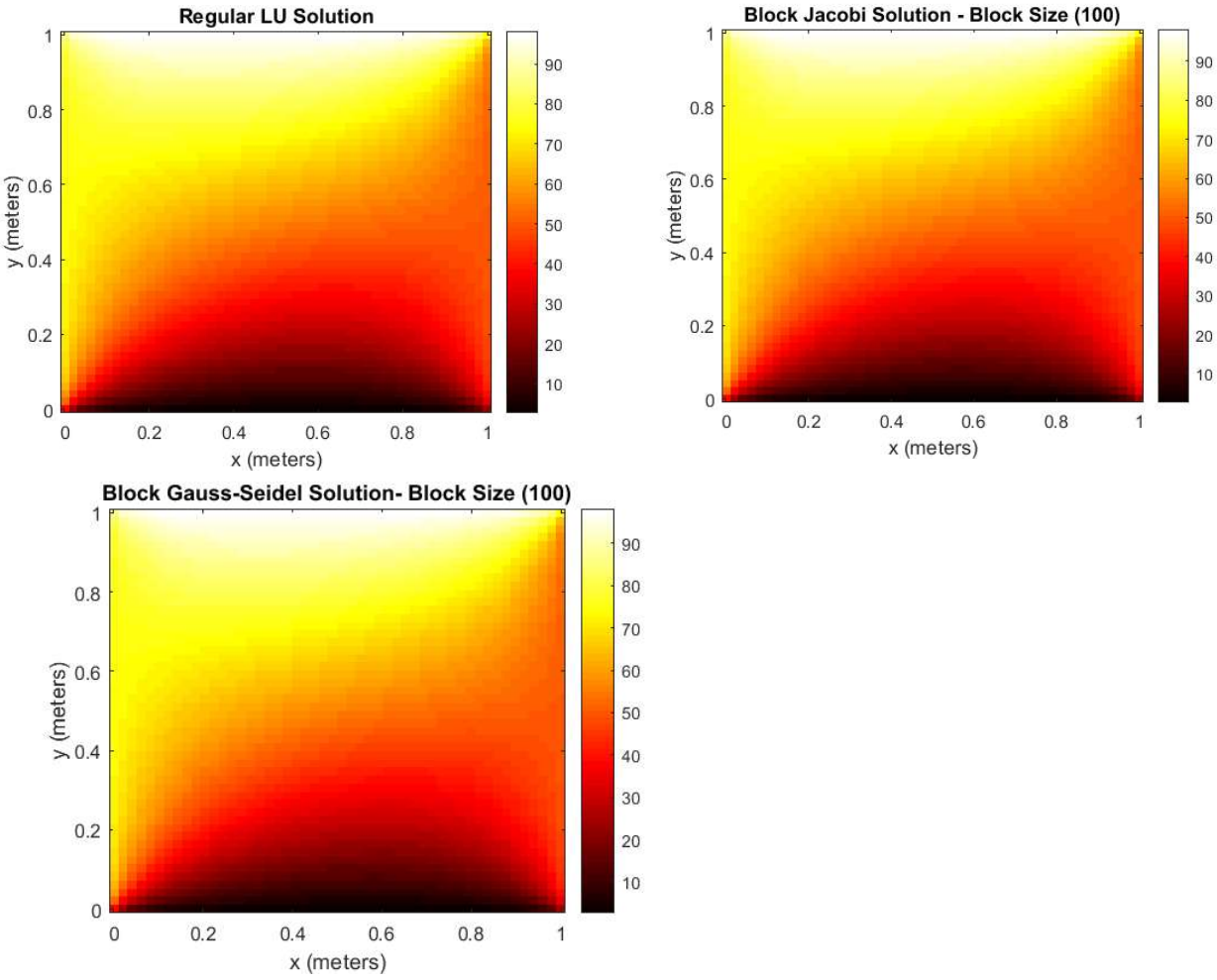
Results and Discussions

From "**Q3Driver.m**" using matrix of dimension **2500x2500** and at different block size, the following result was obtained:

Block Size	**BLOCK JACOBI**		**BLOCK GAUSS-SEIDEL**	
	Relative Error	**Iteration Count**	**Relative Error**	**Iteration Count**
100	9.9466e-09	700	9.9852e-09	370
200	9.767e-09	373	9.5584e-09	197
500	9.5232e-09	169	9.4163e-09	89
1000	8.771e-09	87	8.532e-09	46
1600	7.4876e-31	1	7.4876e-31	1

The table above shows that block Gauss-Seidel tend to converge faster than block Jacobi, but in overall the convergence is faster as the block size increase down the table. From the relative error section, which are the results of the true solution minus the calculated values by each of the algorithms, it can be seen that the error tends to be very small and almost insignificant; thus it can be confirmed that the implementations worked.

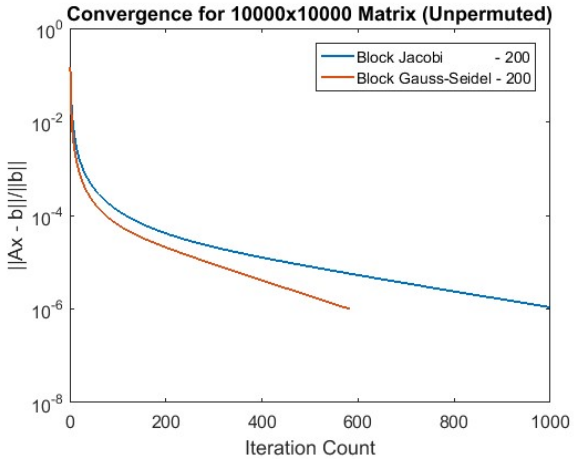
However, while the report above was not the only result obtained from "**Q3Driver.m**", the other interesting part is the plot of the solutions obtained at each block size shown above. Using the function given to us - "**plotLaplaceSolution.m**", the following graph was obtained for a block size (others are not presented to save space):

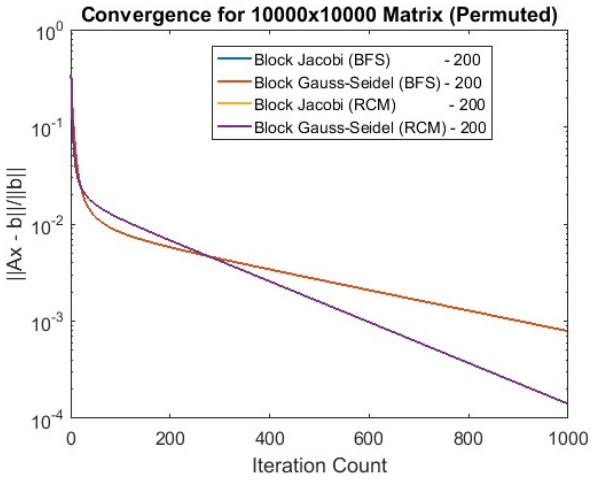


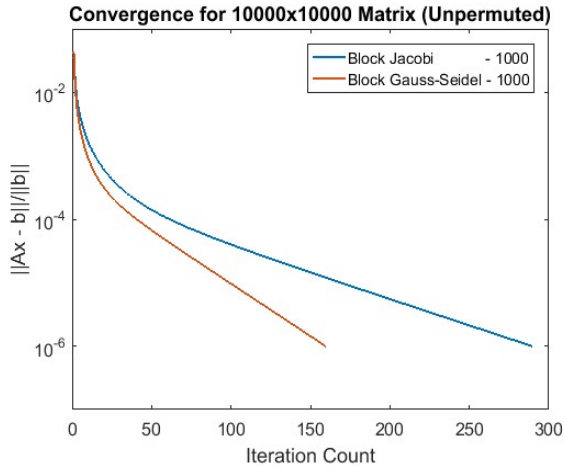
Comparing the temperature distributions, that is the solution plot, above, it can be seen that the distributions (solutions) are the same, this again validates that both block Jacobi and block Gauss – Seidel implementation gave correct solutions, and thus worked.

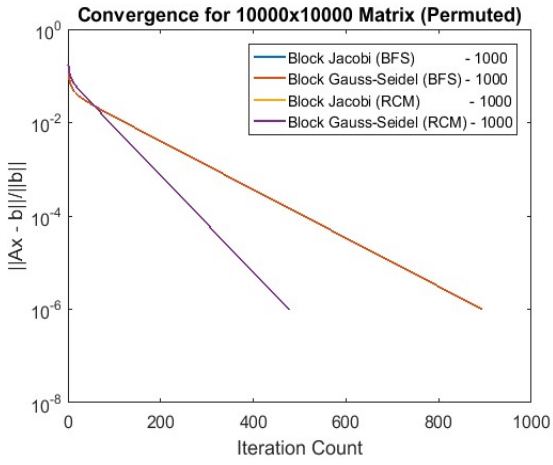
Now, going into "**Q3.m**", where the main comparison was done between the two algorithm implementations, different test cases were done. For the sake of this report, **10000x10000** input matrix was used with maximum iteration set to **1000** and tolerance of $\mathbf{1x10^{-6}}$, below are the results from the different test scenarios used:

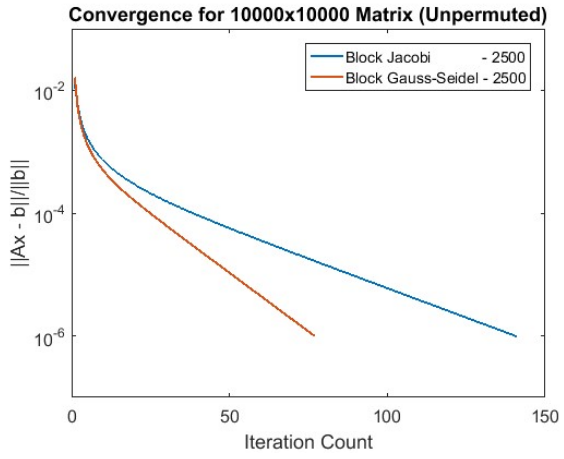
Case 1: When a single block size was tested for both Bread First Search (BFS) and Reverse Cuthill-McKee permuted input matrix A and when not permuted. The following results were obtained:

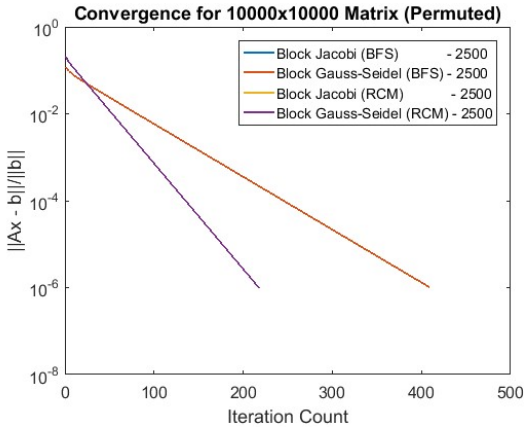








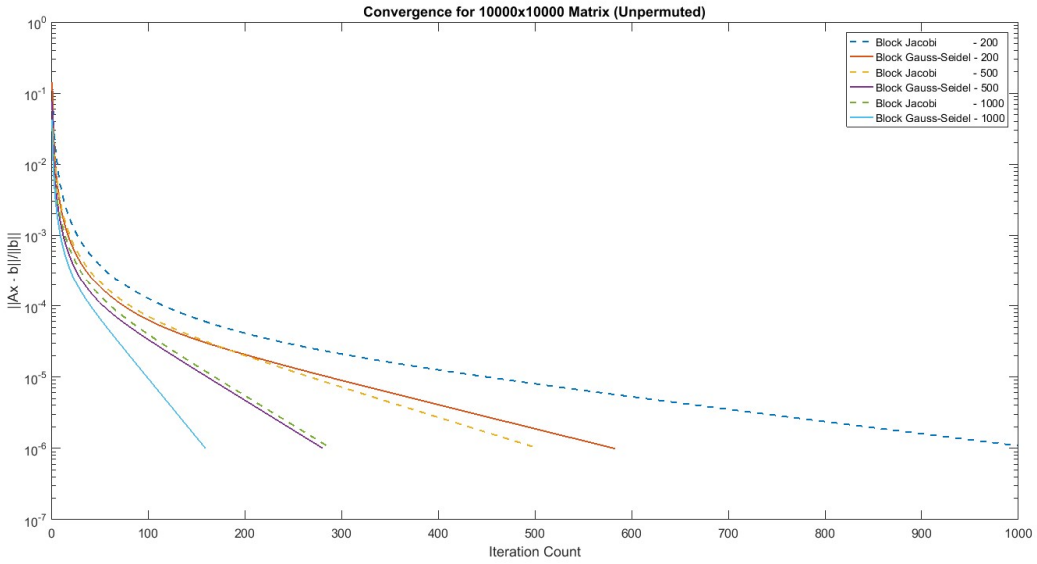


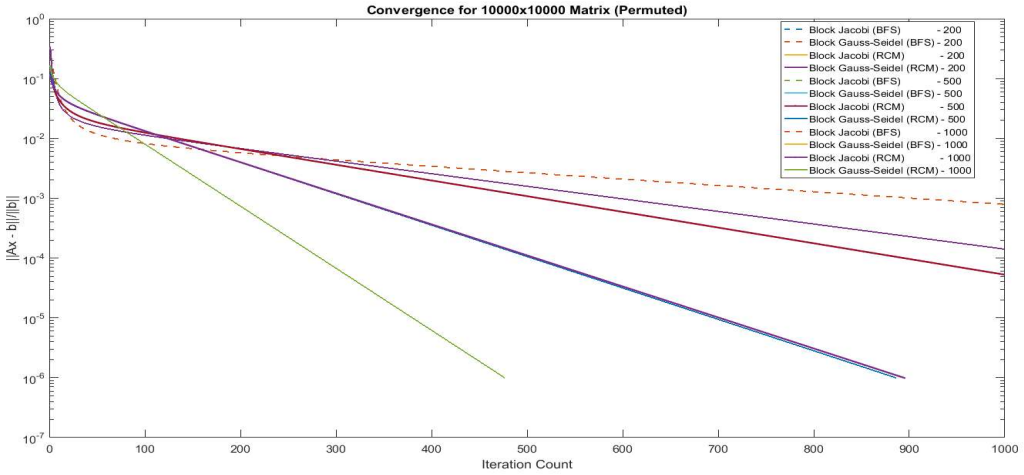


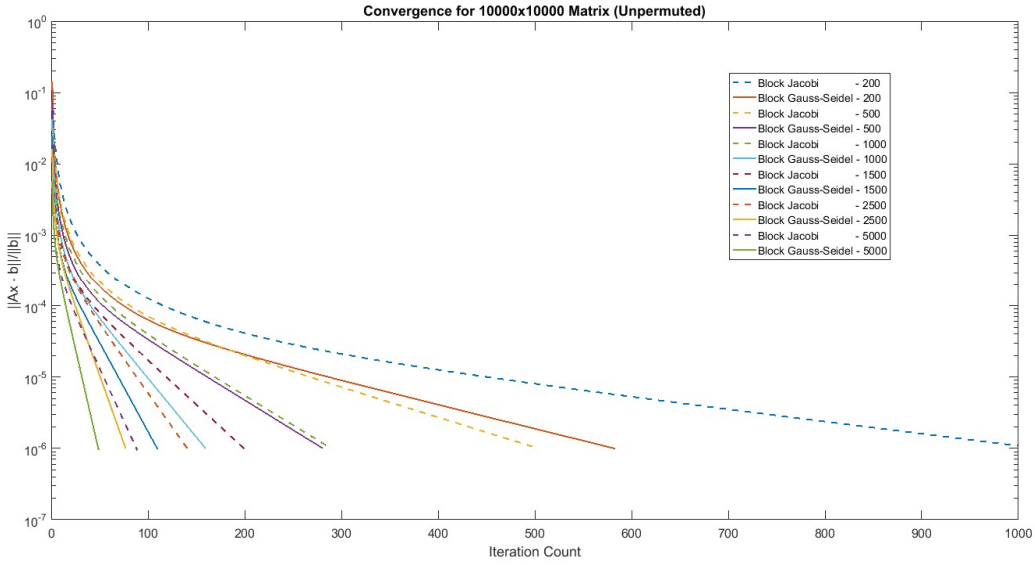
With a block size selected as shown above, for **200**, **1000** and **2500** block sizes, it can be noted from the graph that block Gauss-Seidel is faster in overall than block Jacobi. Whereas this is important, the other noteworthy thing here are: Firstly, considering the left side (unpermuted results) and the right side (permuted results), the iteration counts tend to decrease as the block size increases - that is, faster convergence - and in fact it was 1 when block size was **10000**. Secondly, with permutations (**BFS** and **RCM**) applied to input matrix **A**, in all the presented cases above the convergence rate tend to be worse compared to their respective unpermuted ones. For example,

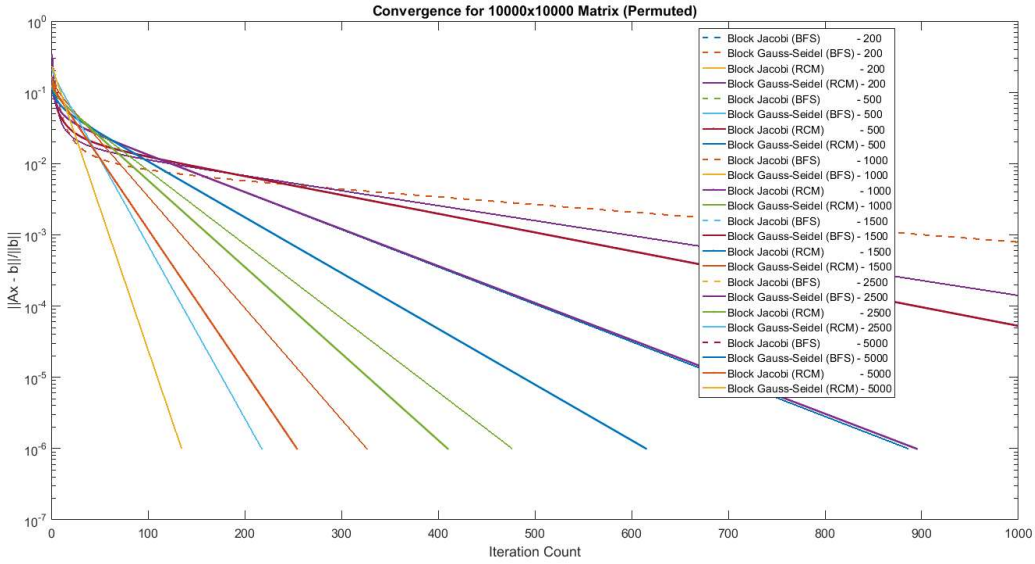
while unpermuted one converges for Jacobi and Gauss-Seidel (block size 2500) at 141 and 77 iterations respectively, permuted ones converged for the same block size at 410 and 218 iterations respectively. One last thing here, observing the right hand side section of the graph presented above (that is, permuted ones), it can be seen that two (2) lines are overlapping so instead of four (4) lines we ended up having two (2). Why this behavior? From the returned results, it was revealed that the result for both **RCM** or **BFS** applied to matrix A will make no difference when block Jacobi or block Gauss-Seidel are used to solve the matrix; as whichever of the two applied to matrix will return the same result and converge at the same iteration count, however permuting the matrix has bad effect on the convergence as shown above.

Case 2: When multiple block sizes was tested for both Bread First Search (BFS) and Reverse Cuthill-McKee permuted input matrix A and when not permuted. The following results were obtained:









As noted for Case 1 previously presented, studying the above graphs also confirmed those observations in Case 1. It shown that as the block sizes increases so also the solutions converge faster. Again, in overall, permuting a matrix tend to worsen its convergence for block Jacobi and Gauss-Seidel iterative techniques. While in overall, the block Gauss-Seidel has better convergence rate than block Jacobi.

Case 3: When block size was 1, that is 10000, the effect of permutation was test by taken into account time taken to compute unpermuted ones and permuted ones. The following results were obtained:

Iterative Technique	Permutation Applied	Time Taken (s)
Block Jacobi	No	0.028938
	Breadth First Search	0.036742
	Reverse Cuthill-McKee	0.030952
Block Gauss-Seidel	No	0.035920
	Breadth First Search	0.025536
	Reverse Cuthill-McKee	0.027780

From the table above, it can be stated that, block Jacobi is no better at all when permuted as the time taken increase when permutated while for block Gauss-Seidel the time tend to improve as the matrix was permuted.

Conclusion and Final Comments

From the forgoing, it can be concluded that in overall block Gauss-Seidel iterative technique converges faster than its counterpart block Jacobi. Also, as the block sizes increase so also the convergence rate increases. On the same note, permuting a matrix decreases its convergence rate for block Jacobi and Gauss-Seidel iterative techniques.

Meanwhile, when a block size is 1, that is the same size as dimension, permuting a matrix with BFS or RCM and solving it with block Jacobi is not in any way advised. While permuting a matrix with BFS or RCM can be said to save time as it turned out to be little bit faster when permuted (though time taken to compute and generate a permutation matrix itself if considered, then this assertion can be given a second thought and may not be an option any longer, thus may be avoided in its entirety).

Finally, permutation or permuting a matrix is intended for LU factorization as it helps in solving and LU decomposed system. WHY? Because Gaussian Elimination method is the precursor of LU, and its purpose is to transform a matrix into row echelon form or upper triangular matrix form which can be easily solved with backward substitution. But during this process of transforming a matrix, issues like ill-conditioning and pivoting (when zero, in worst case, or a smaller value is at the pivot location) comes up, which will require row and/or column swapping which is the function of a permutation matrix. So, all these algorithms were developed to help solve these issues and as well obtain the efficient path transverse by the system and obtain appropriate permutation for the same thus saves times in solving such a sparse system.