**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

1. “**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:

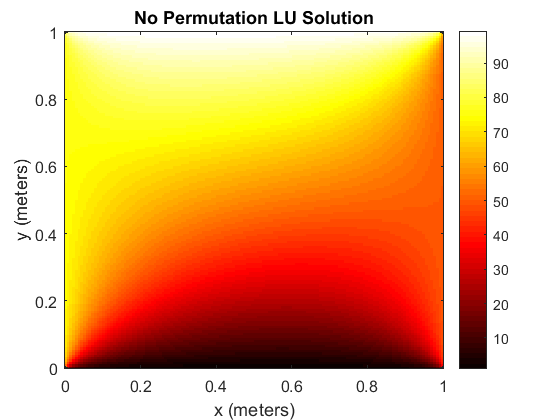
1. Original unpermuted
2. Bread First Search (BFS) symmetric permuted
3. Reverse Cuthill-McKee symmetric permuted
4. MATLAB Built-in Routine permuted

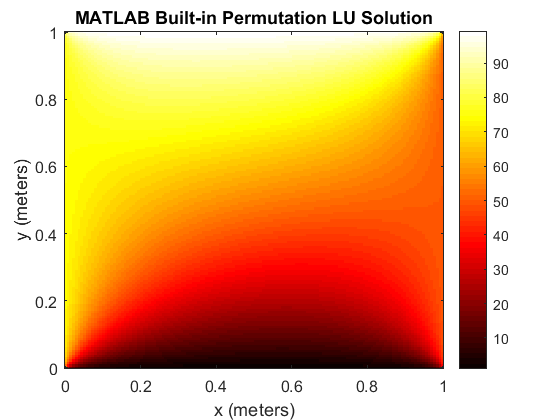
The above four conditions were implemented and test as contained in “**Q2.m**”. However, for the purpose of this report, the graph showing the time taken and data used are replicated in “**Q2reportDataPloting.m**” (though, all results can also be obtained from “**Q2.m**”). To show that the implementation 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:

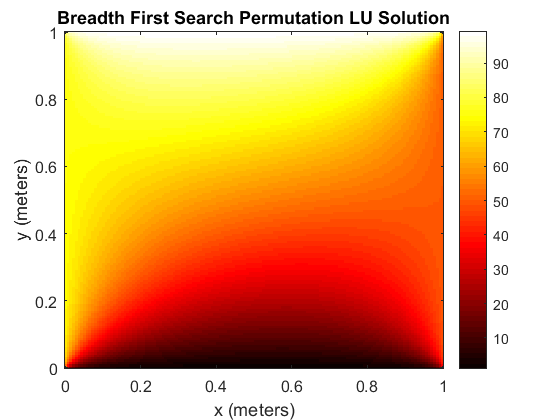
The table showing time taken in seconds for the four conditions tested

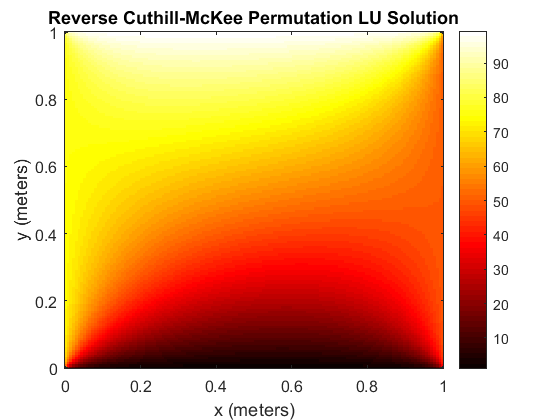
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dim (nxn) | Unpermuted (s) | Built-In Routine(s) | BFS (s) | RCM (s) |
| 100x100 | 0.001058 | 0.002111 | 0.000452 | 0.000504 |
| 400x400 | 0.002605 | 0.002769 | 0.001558 | 0.001309 |
| 900x900 | 0.008263 | 0.008247 | 0.002709 | 0.003878 |
| 1600x1600 | 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 |

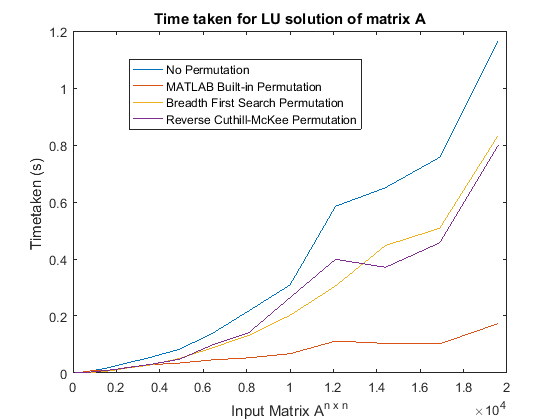
The plot of solutions obtained from the four (4) condition for dimension (19600x19600)

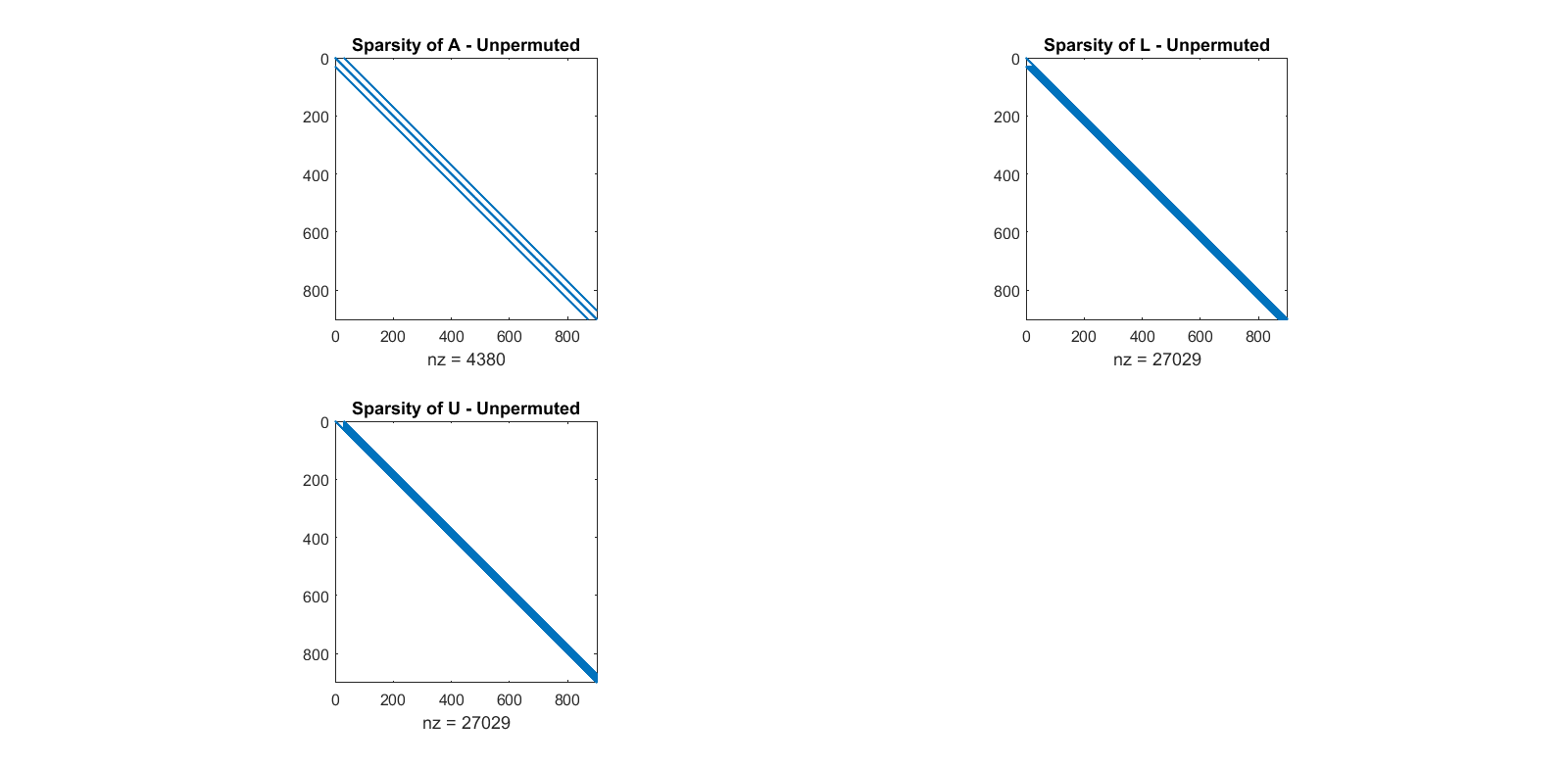


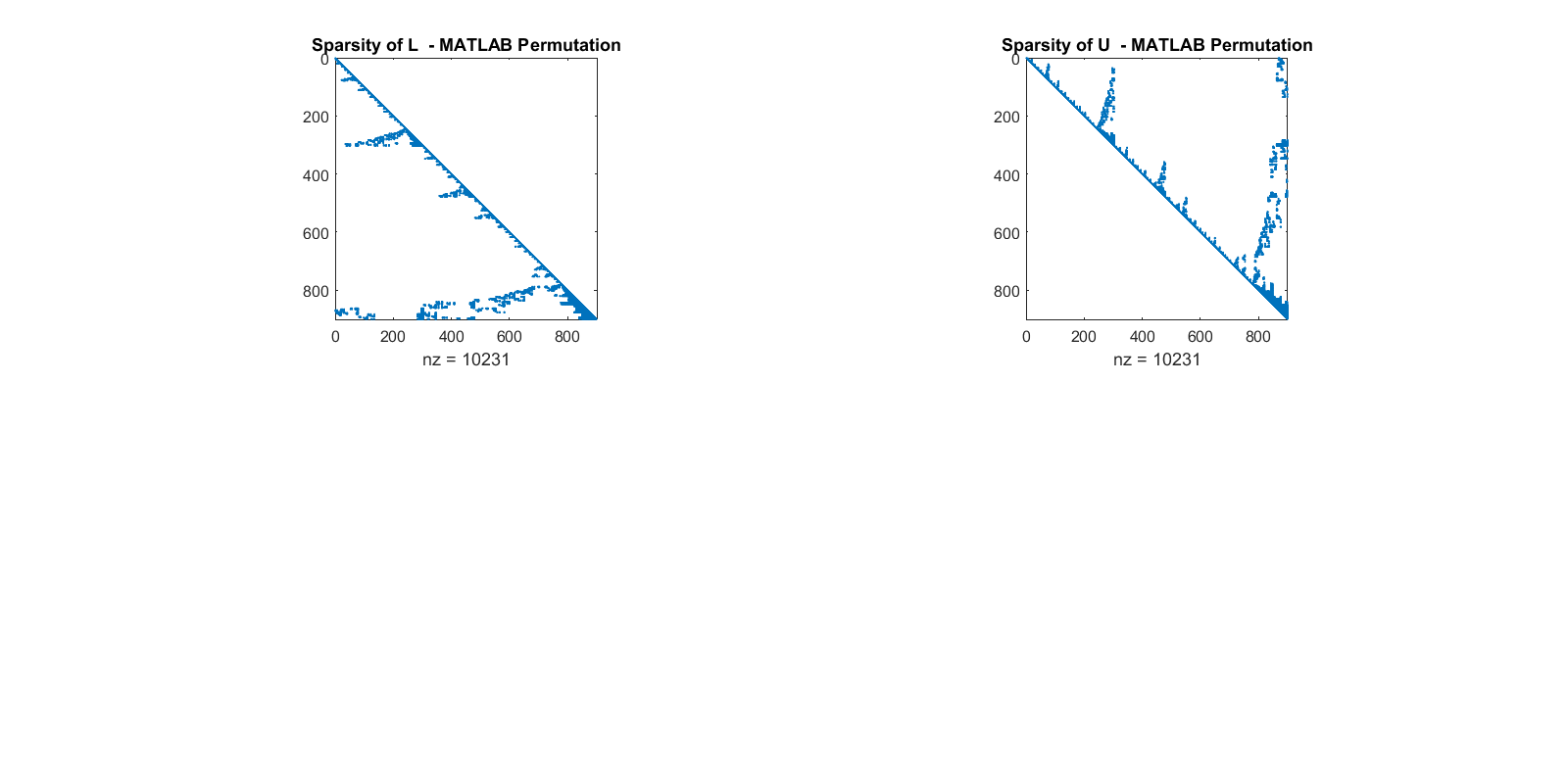
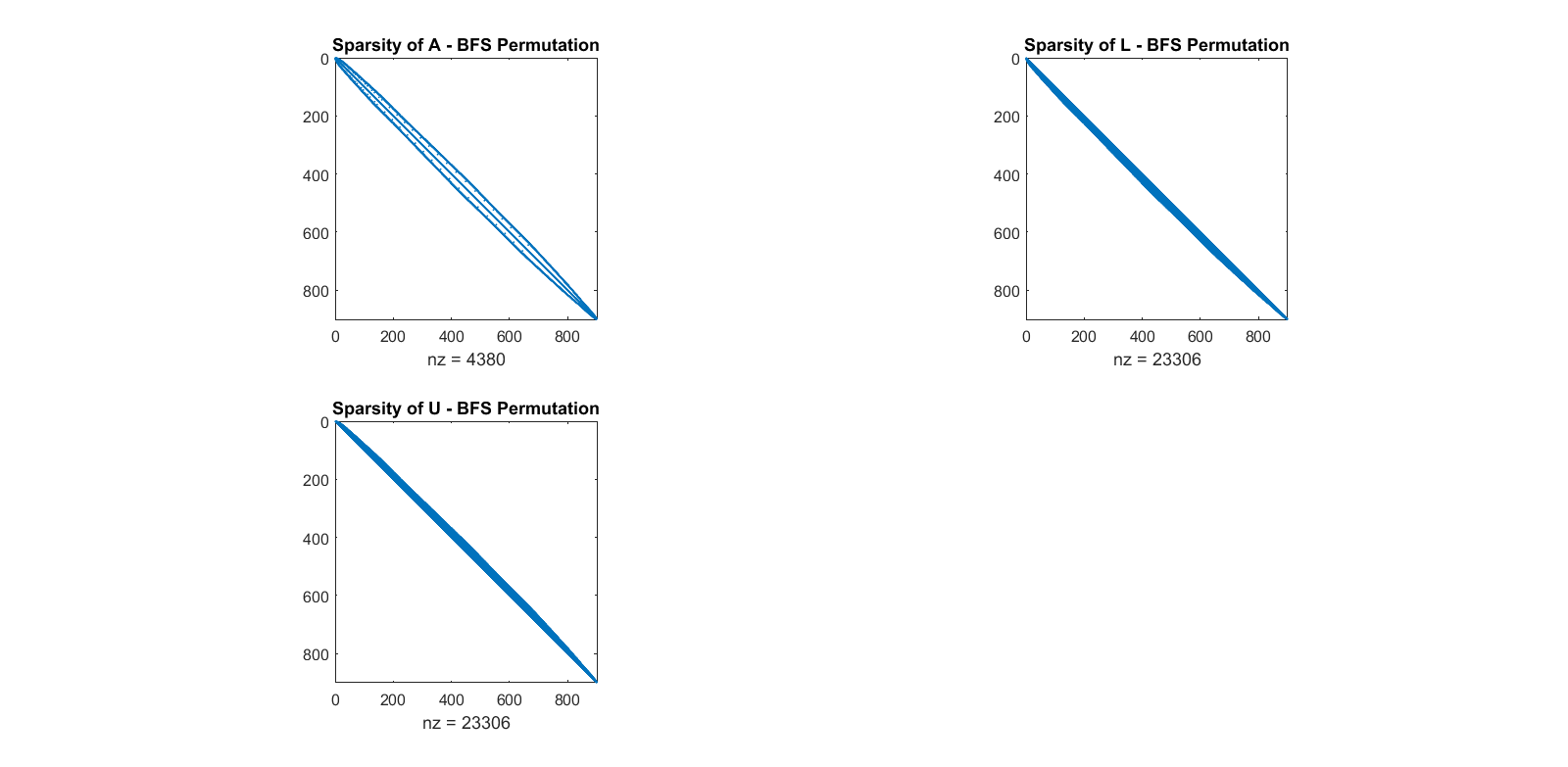


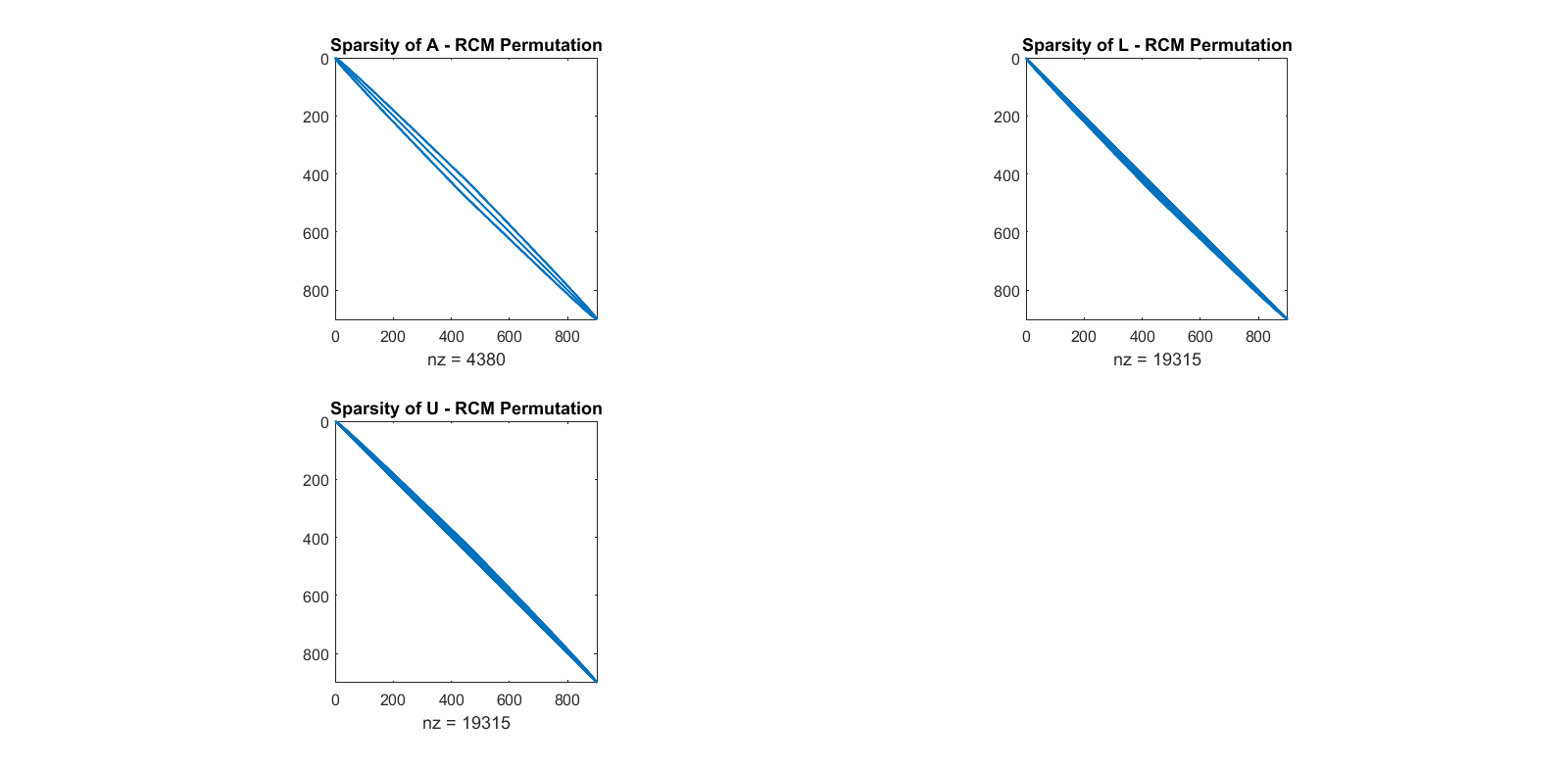






For visualizing the effect of three (3) applied permutation on the input matrix A, file named “**Q2visualize.m**” was used for this purpose and the following graphs were obtained.





**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 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 program for this question is “**Q3.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.