ENEE 641

PROGRAMMING ASSIGNMENT 2

Algorithmic Aspects of Vertex Elimination on Graphs-2
(Using Gaussian elimination to get the solution.)

Abhay Raina
Date of Submission:
11-12-14

ABSTRACT

The problem is to use the use the lexp, lexm and original orderings to solve a given set of equations and get the corresponding solutions. Simple Gaussian elimination is used and we try to make the lower triangular matrix to be all zeros by eliminating the vertices in the given order. Finally backward substitution is used to get the solutions to the variables of the equations.

1. INTRODUCTION

Solving very large sets of simultaneous sparse linear equations has been one of the most fundamental problems in science and engineering. Gaussian elimination is an algorithm for solving systems of linear equations. We insert the coefficients of various equations in matrices. Gaussian elimination is the sequence of operations performed on these matrices of coefficients to get the correct solution. These operations can be performed in various sequences, but some sequences can be better than others to get the solutions. The objective of this project is to find a perfect sequence and if no such sequence exists then to find sequence with minimal number if fill-ins.

We consider the problem of solving such sets of linear equations using Gaussian elimination method. To take advantage of their sparseness, it is crucial to find a pivoting order that minimizes the number of new non-zero elements, called fill-ins, to be generated in the elimination process. A number of graph-theoretic approaches to finding such an optimal pivoting order have been introduced. Using the model proposed by Rose [1] and implement in C the perfect and optimal vertex elimination ordering algorithms developed by Rose, Tarjan, and Lueker [2]. Using such pivoting orders, we then develop an efficient C code to solve the corresponding set of linear equations using Gaussian elimination method.

We use the LexP followed by fill to check if the matrix has any perfect ordering if not then LexM is used to get a minimal order of elimination followed by fill in to get the corresponding fill in edges.

Then we use the found orderings to eliminate the vertices and get the corresponding lower triangular matrix. Finally backward substitution is used to get the solutions to the variables. The same is performed for all the three orderings and the derived solutions are compared to be the same.

We observe theoretical analysis of the code w.r.t the space and time as given in the paper and then we use our program to find the time required for the execution of the program and the space required for the program. Later we compare these two, and find that our algorithm is working as desired.

- Section 2 describes the problem definition, Solution and Bound Analysis.
- Section 3 presents the code.
- In Section 4 we discuss the experiment and its analysis with respect to the theoretical results.
- Section 5 contains the discussions about the process of development of the algorithm.
- Section 6 has the references followed by references.

2. PROBLEM DEFINITION, SOLUTION & BOUND ANALYSES

2.1 PROBLEM DEFINITION

Develop efficient C code to perform Gaussian Elimination Method (GEM) (forward elimination and backward substitution) to solve n linear equations. The GEM requires a pivoting order to determine the sequence of variables to be used for processing the equations. Run the C code with the LEX P, LEX M, and original orders, and keep newly produced nonzero elements in the process of GEM. Compare the locations of those new non-zero elements and their corresponding fill-in edges obtained in Part 1 of PA2.

2.2 SOLUTION

N vertices are taken in as an input from the Matrix market file. So we take following as the input to the LexP, LexM function and generate the ordering this ordering is used to calculate the fill in edges. Original ordering is also used. We insert the various elements in to the matrix into the respective locations using the selected ordering and the B matrix is added to the same to get in **Ax+B** form. Gaussian elimination is performed on them which involves row reduction and an upper triangular matrix is obtained. Finally backward substation is used to get back the solutions to the corresponding variables.

Matrixchange()

This function is used to rearrange the matrix in the order of elimination so that the changed matrix can be supplied to the guass function. We take in the alpha ordering as an input as and well as we take the an ordermap as an input. Then this alpha is used to update the matrix accordingly.

Guass()

The matrix is input to this function which is already in the elimination order. Now row reduction is used to get an upper triangular matrix by eliminating the elements below diagonal elements of the matrix one by one. This is followed by the backward substitution which involves putting the derived variable values as the coefficients of the previous row.

Fill()

Function fill has been modified to store the previously generated fill in edges which need to be compared with the non-zero elements that we get during the subsequent GEM. We use a linked list to store the previous values. And use a linked list again to compare with the new value.

Also, in order to improve the performance of the fill in function the array arrangement was replaced with a linked list arrangement to improve the space requirements under the given space constraints.

PSUEDOCODE(New Parts):

1.In main function we make a (NxN) matrix and with the given non zero elements and also add the B matrix (randomly generated to the same.)

2. In the guass() function.

```
for (j = 1; j <= n; j++)
        for (i = 1; i <= n; i++)
                if (i>j)
                        c = array[i][j] / array[j][j];
                        matrix[i][j] = 0;
                        for (k = j+1; k <= n + 1; k++)
                                matrix[i][k] = matrix[i][k] - c*matrix[j][k];

                if (matrix[n][n] != 0)
                        x[n] = matrix[n][n + 1] / matrix[n][n];

                else
                        printf("GEM has stopped here");
                        return x;
                for (i = n - 1; i >= 1; i--)
                        sum = 0;
                        for (j = i + 1; j <= n; j++)
                                sum = sum + matrix[i][j] * x[j];
                                 if (matrix[i][i] != 0)
                                x[i] = (matrix[i][n + 1] - sum) / matrix[i][i];
                        else
                                printf("GEM has stopped here");
```

Time analysis:

As described in the paper by Rose, Tarjan, and Lueker

As we need to traverse all the subsequent rows and each of its elements, hence we need $O(n^3)$ time.

Space Analysis:

As we need to store the whole matrix hence we need to $O(n^2)$ space.

3. CODE:

Here is the guass function code and matrixchange function code made for implementing the algorithm the whole code has been attached in the appendix 1. :

Program	Comments

```
double * matrixchange(double ** matrix, int* ordermap, int *alpha, int M)
{
	int l1,i,j;
	double tempaks;

	for (i = 1; i<M + 1; i++)
	{
		if (ordermap[i] != alpha[i])
		{
			//l=i;
			for (j = 1; j <= M; j++)
			{
				if (ordermap[j] == alpha[i])
				{
					tempaks = ordermap[j];
					ordermap[j] = ordermap[i];
					ordermap[i] = tempaks;
					break;
				}
			}
			l1 = j;
			tempaks = matrix[i][M+1];
			matrix[i][M + 1] = matrix[l1][M + 1];
			matrix[l1][M + 1] = tempaks;
			for (j = 1; j<M + 1; j++)
			{
				tempaks = matrix[i][j];
				matrix[i][j] = matrix[l1][j];
				matrix[l1][j] = tempaks;
			}
			for (j = 1; j<M + 1; j++)
			{
				tempaks = matrix[j][i];
				matrix[j][i] = matrix[j][l1];
				matrix[j][l1] = tempaks;
			}
		}
	}

	return matrix;
}
```

Pass the matrix to the function and rearrange the matrix according to the given order. Ordermap array is used to keep a track of the vertices when their order is changed.

```
double * guass(double** array, int size, int* alpha,int* alphainv)
{
fprintf(ofp, "Inside guass function\n");
int i, j, k, n,flag,count,error;
error = 0;
count = 0;
flag = 0;
```

```
float c;
float sum = 0.0;
n = size;

struct checkListNode* pivot = (struct checkListNode*) malloc(sizeof(struct
checkListNode));
pivot->i = 0;
pivot->j = 0;
pivot->next = NULL;
pivot2 = pivot;

double* x = (double*)calloc(size + 1, sizeof(double));

for (j = 1; j <= n; j++) /* loop for the generation of upper triangular
matrix*/
{
for (i = 1; i <= n; i++)
{
if (i>j)
{
//printf("\n");
//printmatrix(array, size);

if (array[j][j] == 0)
{
printf("The GEM stops here as element is zero");
error = 1;
goto here;
}

c = array[i][j] / array[j][j];

array[i][j] = 0;

for (k = j+1; k <= n + 1; k++)
{
if (array[i][k] == 0 & k!=n+1 )
{
flag = 1;
}
array[i][k] = array[i][k] - c*array[j][k];

if (flag == 1 & k != n + 1  )
{
if (array[i][k] !=0 & k>i)
{
count++;
fprintf(ofp, "(%d,%d %lg) ", alpha[i], alpha[k], array[i][k]);
head2 = newcheckListNode(alpha[i], alpha[k]);
temp2 = pivot2->next;
pivot2->next = head2;
head2->next = temp2;
/*                   }
else
{
fprintf(ofp, "(%d,%d %lf) ", alpha[k], alpha[i], array[i][k]);
head2 = newcheckListNode(alpha[k], alpha[i]);
temp2 = pivot2->next;
pivot2->next = head2;
head2->next = temp2;
}*/
}
```

Here we generate the upper triangular matrix in the echelon form. We use three for loops to interchange elements which takes $O(n^3)$ time.

Fill in edges are added to the linked list to compare them.

```
flag = 0;
}
}
}
}

}



if (array[n][n] != 0)
{
x[n] = array[n][n + 1] / array[n][n];
}
else
{
printf("GEM has stopped here");
}
/* this loop is for backward substitution*/

for (i = n - 1; i >= 1; i--)
{
sum = 0;

for (j = i + 1; j <= n; j++)
{
sum = sum + array[i][j] * x[j];
}

if (array[i][i] != 0)
{
x[i] = (array[i][n + 1] - sum) / array[i][i];

}

else
{
printf("GEM has stopped here");
}

}
here:
fprintf(ofp,"\nNon zero element count = %d", count);
printf("\nNon zero element count = %d \n", count);
fprintf(ofp,"\nThe solution is: \n");
if (error == 0)
{
for (i = 1; i <= size; i++)
{
fprintf(ofp, "x%d=%f\t", i, x[alphainv[i]]); /* x1, x2, x3 are the required
solutions*/
}

fprintf(ofp, "\n");
}
else
{
fprintf(ofp, "\nNo Solution exists");
error = 0;
}
int chkcnt=0;
```

Here we perform the backward substitution to get our solution.

Here the solution is printed.

```
struct checkListNode* tempchk1;
struct checkListNode* tempchk2;
tempchk1 = pivot1;
tempchk2 = pivot2;
while (pivot1 != NULL)
{
while (pivot2 != NULL)
{
if ((pivot1->i == pivot2->i & pivot1->j == pivot2->j )|| (pivot1->i ==
pivot2->j & pivot1->j == pivot2->i))
{
chkcnt++;
pivot1->i=0;
pivot1->j=0;
}
pivot2 = pivot2->next;
}
pivot2 = tempchk2;
pivot1 = pivot1->next;
}
pivot1 = tempchk1;

/*printf("\n");
while (pivot1 != NULL)
{
if (pivot1->i != 0 & pivot1->j != 0)
{
printf("%d %d ", pivot1->i, pivot1->j);
}
pivot1 = pivot1->next;
}*/

printf("\n");
chkcnt--;
fprintf(ofp, "Matched=%d Notmatched=%d\n", chkcnt, count-chkcnt);
//printmatrix(array, size);
return x;
}
```

Here we search for matching fill in edges and print the number of matching edges. We use a linked list arrangement which saves space when compared to a array arrangement .

Data Structure & Time analysis:

We use a NX(N+1) matrix to store the input values from the matrix market to the given matrix and also the corresponding B vector. Then we use three for loops to traverse each element of the subsequent rows to perform row reduction this takes $O(n^3)$ time.

Data Structure & Space Analysis:

As we use a matrix to store the elements we need space for each of the elements and this takes $O(n^2)$ space.

4. EXPERIMENTS AND ANALYSIS:

The experiment here was to calculate the time and space requirements theoretically and then to compare it with the practical values that we get from our program.

Table 1: File Number,File Id, File Name, Size, Non Zero Elements, Time taken and Memory used in KB.

File No.	File ID	File Name	Size	Non-zero elements	Time taken (in sec)	Memory Used (in Kbytes)	Premature Exit
1.	220	nos4.mtx	100 * 100	594	0.38	85	No
2.	141	fs_183_1.mtx	183*183	998	12.9	281	No
3.	876	mesh2em5.mtx	306 * 306	2,018	31.69	755	No
4.	344	bcsstm34.mtx	588 * 588	21,418	435.190002	2906	No
5.	240	saylr3.mtx	1000 * 1000	3,750	245	7869	Yes
6.	2401	netscience.mtx	1589 * 1589	5,484	15.33	19799	Yes
7.	68	bcsstm13.mtx	2003 * 2003	21,181	26.555	31750	Yes
8.	1239	laser.mtx	3002 * 3002	9,000	33.08002	70543	Yes

4.1 ANALYSIS OF NUMBER OF FILL-IN EDGES AND THE NON-ZERO ELEMENTS IN THE GEM.

In some rare cases we observed that Fill edges generated were not equal to the non zeros generated. After some analysis I found out that this is happening due to division approximations ex. $\left(\frac{2}{3}\right) * 3 = 2$ is the expected result but in the computer approximation leads to 2.00000001 and when subtracted we get 0.00000001 which is non-zero but should have been zero. This leads to wrong fill in edges being generated in some cases.

4.2 TIME CALCULATION EXPERIMENT:

For the time calculation a time counter was placed at the beginning and the end of the program to find the time required to go through the program. The same was repeated for the 10 files given and corresponding execution times were noted down. Table 1 contains the values of the same.

Also we plot the following graph using the values from Table 1.

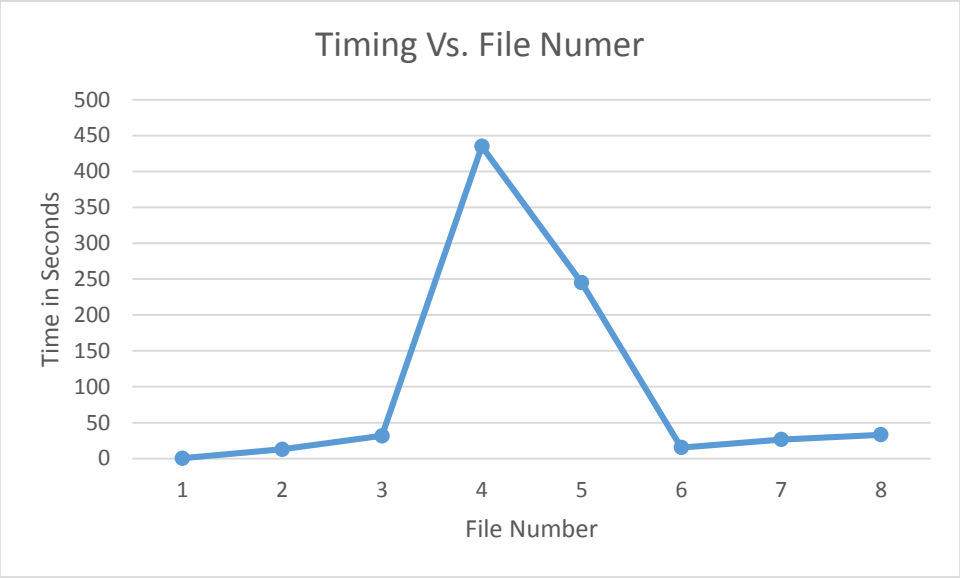


Figure 1: Graph of Execution Time vs. Input File Size

The above graph contains the experimental values obtained when the c program was run the columns contain the file id, then has the file name, then has the size and then has the memory requirements of the code in Kilobytes.

Analysis:

My theoretical analysis of the code showed that the runtime of the code increases as the number of vertices and the number of edges increased. When the experimental values were plotted in the graph above result was obtained.

As we can see that for first three files, as we are getting solutions as our timing graph increases exponentially. But subsequent files stop execution in between as there is no solution hence we see the sudden drop in the timing graph.

4.3 SPACE CALCULATION EXPERIMENT:

The space calculation was done using counting the number of calloc() assignments in the program and finding the amount of memory that the reserve in each case and then adding them all up to get an idea of how much memory is required in general. The same was repeated for the files given and corresponding space requirements were noted. Following table contains

the values of the same. As at a time we are using only 3 arrays in general, that amount has been added to the normal space requirements. The space requirement has only the analysis according to what maximum memory we can require at any give time.

Also we plot the following graph using the above values from Table 1.

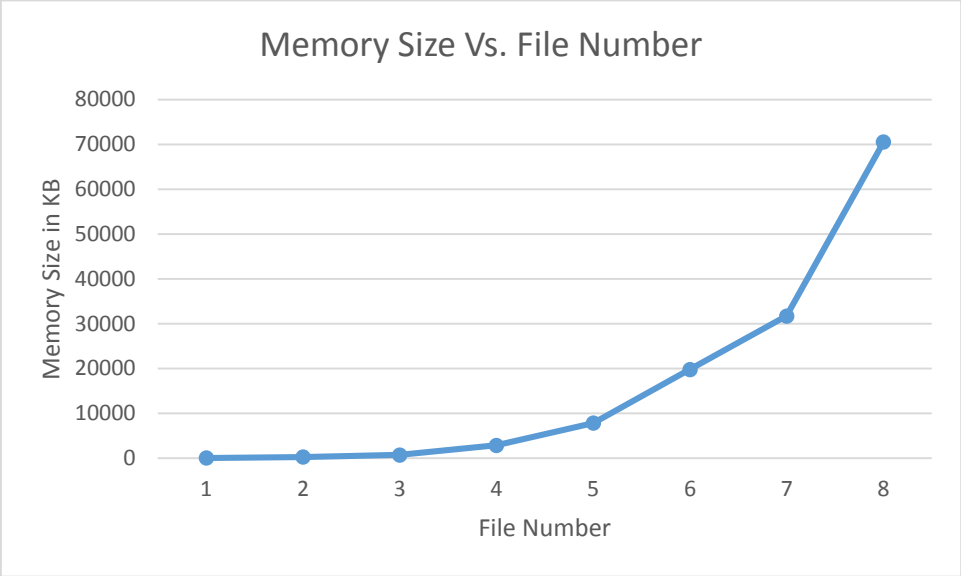


Figure 2: Graph of Memory Required Vs. Input File

Therefore from the above graph we can safely conclude that our code is working as desired from it i.e. the space requirements are as expected. As we can observe the Space is increasing $O(n^2)$ as our file size increases. As the input size becomes large the space requirement might serve as a constraint for the program to run and the program may need access to secondary memory.

5. DISSCUSSIONS

While designing the algorithm following problems were faced:

1. To interchange the matrix entries in correct order which corresponds to the given elimination order.
2. To ensure that asymmetric matrix is turned into a symmetric matrix.
3. Keep a count of the produced fill in edges.

Weaknesses of my approach would be as follows:

1. As the number of inputs increase the amount of memory required at that moment increases and sometimes the system stack won't be having the memory to hold such huge amount of numbers and tries to access the secondary memory this leads to slowing of the program.

Future improvements to my approach could be to improve the memory consumption of the program.

BONUS1 (TASK 2)

Objective: Apply LEX P and LEX M to more symmetric matrices with real/integer coefficients from the Florida Sparse Matrix Collection and generate fill-in edges with the LEX P, LEX M, and original orderings. Analyse the results and identify, if any, types of matrices for which any of these three orderings produces smaller numbers of fill-in edges than the others.

Analysis:

When analysis on various matrices was done following patterns were seen:

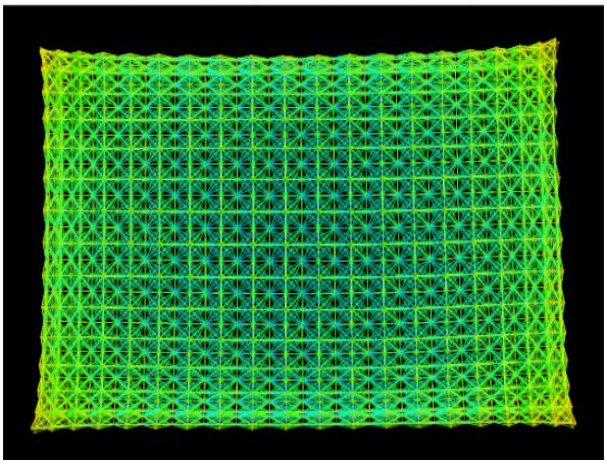
1. In some matrices original orderings gave lower fill-ins than LexP and LexM.
2. In some matrices fill ins of both LexP and LexM were the same.

We analyse the two cases below one by one and give the reasons for such behaviour.

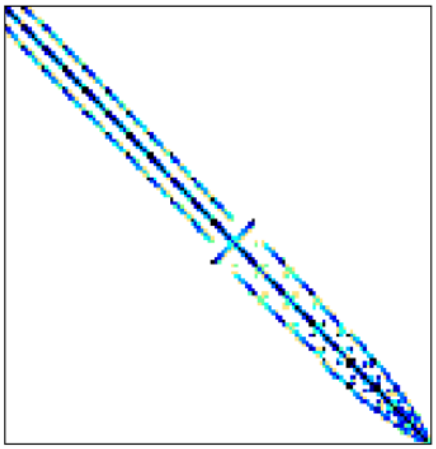
1. In some matrices original orderings gave lower fill-ins than LexP and LexM.

Following are the structures of matrices which gave lower number of fillins for original orderings as compared to the fill-in edges for LexP and LexM.

1. Filename-ex14.mtx

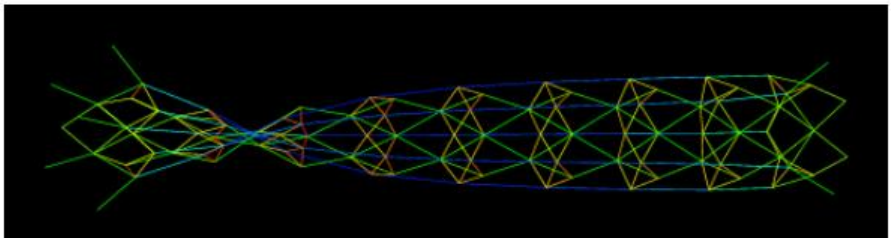


Graph Drawing

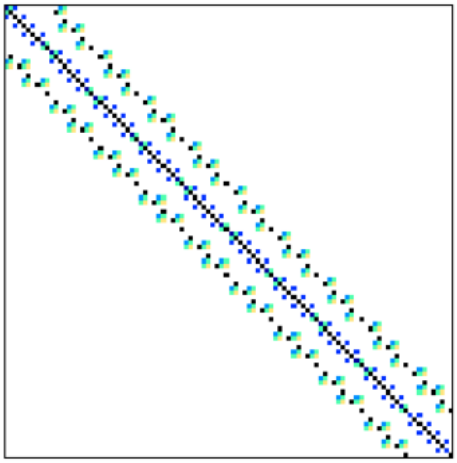


Matrix Drawing

2. Filename-nos4.mtx

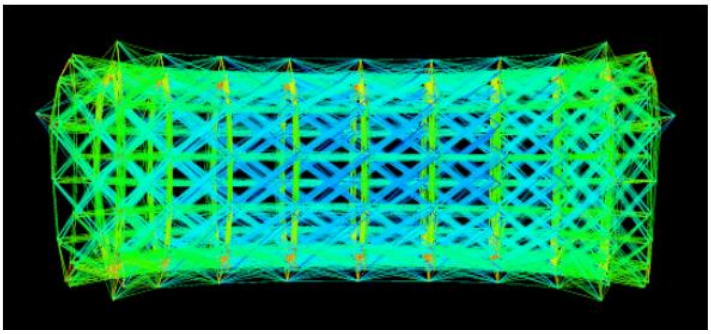


Graph Diagram

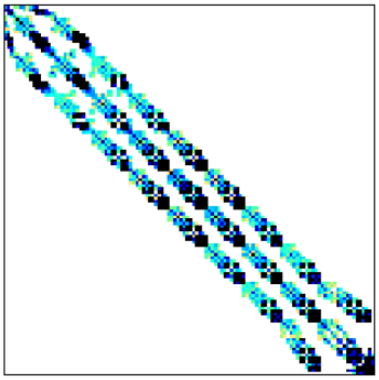


Matrix Diagram

3. Filename- bcsstm34.mtx



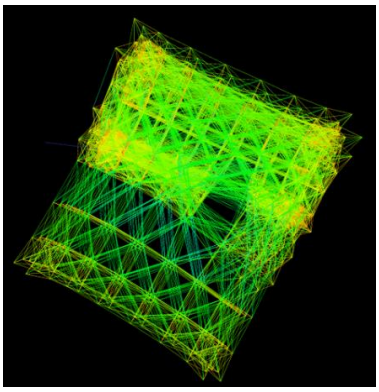
Graph Diagram



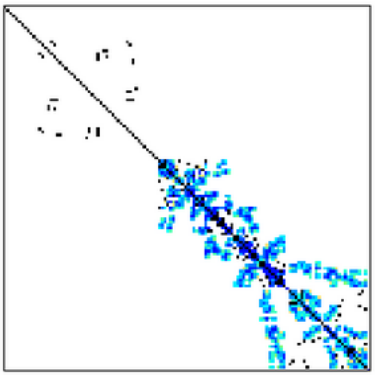
Matrix Diagram

As we can see above all the matrices above have the common property that their undirected graphs are symmetric in the 3 dimensions. It can also be seen that the matrix diagrams have lines running parallel to the diagonal. This is another property that hints towards the matrices that have lower original ordering fill in counts as compared to the LexP and LexM. As we have symmetric graphs therefore when the lexP and lexM functions eliminate in original ordering we are bound to get better results. On the similar lines another matrix was searched found and was observed to have lower original ordering fill in counts as compared to the LexP and Lex. Its details are as follows.

Filename- bcsstm13.mtx



Graph Diagram



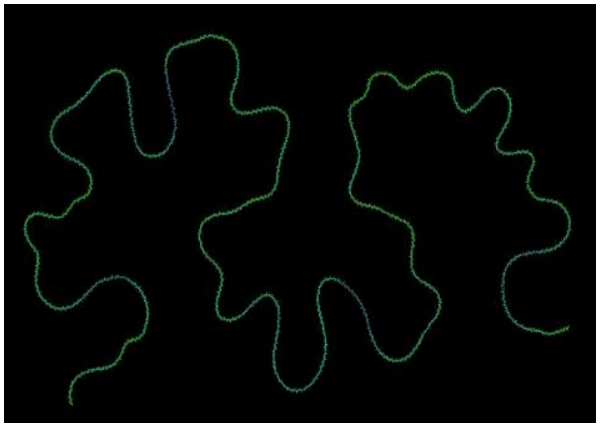
Matrix Diagram

Hence we can conclude that the matrices with the above mentioned properties do actually have lower original ordering fill in counts as compared to the LexP and LexM.

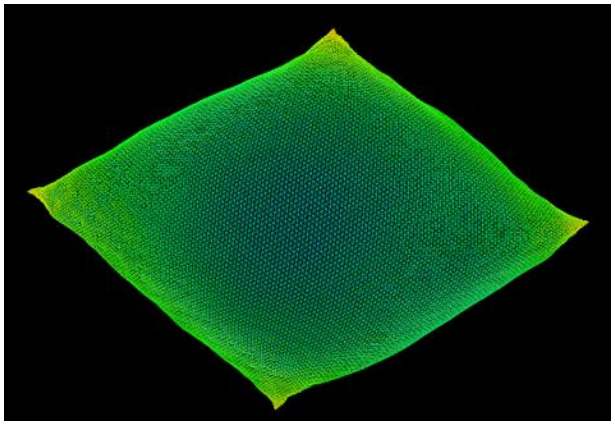
2. In some matrices fill ins of both LexP and LexM were the same.

Following are the graph diagrams of matrices which gave equal number of fillins for orderings of LexP and LexM.

File name-Laser.mtx



File name-fV3.mtx



If we observe carefully both the graph diagrams have all the points lying on the same plane and hence when we apply lexP or LexM the adjacent elements are in a line. Hence from both the algorithms we are bound to get the same number of fillin edges. Therefore if we look for other matrices of the same type they might also have equal lexp and lexM fillins.

BONUS1 (TASK 3)

Objective: Convert Asymmetrical Matrix to Symmetrical Matrix

Solution: Code snippet explaining the same.

```
for (i = 0; i < nz; i++)
	{
		if (val[i] == 0)
		{
			I[i] = 0;
			J[i] = 0;
		}
	}

	int j;
	for (i = 0; i < nz; i++)
	{
		for (j = i + 1; j < nz; j++)
		{
			if ((I[i] == J[j]) & (J[i] == I[j]) & I[i]!=0 & J[i]!=0 )
			{
				if (val[i] >= val[j])
				{
					val[i] = val[j];
				}

				I[j] = 0;
				J[j] = 0;
				val[j] = 0;


			}

		}
	}
```

We go through the given Matrix market format and look for any asymmetric elements. If any asymmetric elements are found the smaller value needs to be copied to the both. In case one of them is zero the non zero value is copied to both vertices in the adjacency list.

Bonus2 (Task 4)

Objective: (3% bonus) Add the fill-in edges stored in each of the output files of Part 1 with an ordering α (LEX P, LEX M, or original) to the original graph. This results in graph Gα*. Apply LEX P and LEX M to this graph and find their corresponding elimination orders. Then apply FILL with each of these orders to generate the corresponding fill-in edges. Finally analyse the results.

Solution:

Our fill in function was generating a graph with the fill in edges added already. What we did is to run the fill-in code back on it again in order to see if the fill-in gives a 0 output. Our results were satisfying as we got the following results for some of the files. The code has been supplied in the uploaded files.

File Name	LexP Fill in Count	LexM Fill in Count	Original Fill in Count
Before adding fillins to Nos4.mtx(220)	493	468	158
After adding fillins to Nos4.mtx(220)	0	0	0
Before adding fillins to bcsstm34.mtx (344)	29851	28006	26981
After adding fillins to bcsstm34.mtx (344)	0	0	0
Before adding fillins to saylr3.mtx(240)	21878	21377	22610
After adding fillins to saylr3.mtx(240)	0	0	0

In general following steps were followed:

Lexp(graph, alpha1, alpha2);
fillin(graph, alpha1, alpha2);
fillin(graph, alpha1, alpha2);

Lexm(graph, alpha1, alpha2);
fillin(graph, alpha1, alpha2);
fillin(graph, alpha1, alpha2);

fillin(graph, alpha1original, alpha2 original);
fillin(graph, alpha1original, alpha2original);

6. REFERENCES:

1. Paper on Algorithmic aspects of vertex elimination on graphs by Donald J. Rose, R. Endre Tarjan and George s. Lueker.
2. http://www.tutorialspoint.com/c_standard_library/
3. http://en.wikipedia.org/wiki/Gaussian_elimination
4. www.wikipedia.org/wiki/**Radix_sort**
5. http://www.geeksforgeeks.org/

Appendix 1.

```
#include <stdio.h>
#include"mmio.h"
#include"mmio.c"
#include<time.h>
FILE *f, *ofp;
long long int memcount = 0;
// A structure to represent an adjacency list node
struct AdjacencyListNode
{
	int dest;
	struct AdjacencyListNode* next;
	double weight;
};

struct checkListNode
{
	int i;
	int j;
	struct checkListNode* next;
};

struct checkListNode* newcheckListNode(int i, int j)
{
	struct checkListNode* newNode =
		(struct checkListNode*) malloc(sizeof(struct checkListNode));
	newNode->i = i;
	newNode->j = j;
	newNode->next = NULL;
	return newNode;
}

// A structure to represent an adjacency liat

struct AdjList
{
	struct AdjacencyListNode* head;  // pointer to head node of list
	struct LexpCellNode* cell;
};

struct Graph
{
	int V;
	struct AdjList* array;
};

struct AdjacencyListNode* newAdjacencyListNode(int dest)
{
	struct AdjacencyListNode* newNode =
		(struct AdjacencyListNode*) malloc(sizeof(struct AdjacencyListNode));
	newNode->dest = dest;
	newNode->next = NULL;
	return newNode;
}

struct Graph* CREATEGRAPH(int V)
{
	struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
	graph->V = V;

	graph->array = (struct AdjList*) calloc(V + 1, sizeof(struct AdjList));
```

```
	int i;
	for (i = 1; i <= V; i++)
		graph->array[i].head = NULL;

	return graph;
}

void EDGEADD(struct Graph* graph, int src, int dest, double weight)
{

	if (src != dest)
	{
		struct AdjacencyListNode* newNode = newAdjacencyListNode(dest);
		newNode->next = graph->array[src].head;
		newNode->weight = weight;
		graph->array[src].head = newNode;

		newNode = newAdjacencyListNode(src);
		newNode->weight = weight;
		newNode->next = graph->array[dest].head;
		graph->array[dest].head = newNode;
	}
}

void EDGEADD1(struct Graph* graph, int src, int dest, int weight)
{

	if (src != dest)
	{
		struct AdjacencyListNode* newNode = newAdjacencyListNode(dest);
		newNode->next = graph->array[src].head;
		newNode->weight = weight;
		graph->array[src].head = newNode;

		/*newNode = newAdjacencyListNode(src);
		newNode->weight = weight;
		newNode->next = graph->array[dest].head;
		graph->array[dest].head = newNode;*/
	}
}

void EDGEADD2(struct Graph* graph, int src, int dest, double weight)
{

		struct AdjacencyListNode* newNode = newAdjacencyListNode(dest);
		newNode->next = graph->array[src].head;
		newNode->weight = weight;
		graph->array[src].head = newNode;

		if (src != dest)
		{
			newNode = newAdjacencyListNode(src);
			newNode->weight = weight;
			newNode->next = graph->array[dest].head;
			graph->array[dest].head = newNode;
		}
}

void printGraph(struct Graph* graph)
{
	int v, d;
	for (v = 1; v <= graph->V; v++)
```

```
	{
		struct AdjacencyListNode* pCrawl = graph->array[v].head;
		printf("\n Adjacency list of vertex %d\n head ", v);
		while (pCrawl != 0)
		{

			printf("-> %d (wt=%lf)", pCrawl->dest, pCrawl->weight);
			pCrawl = pCrawl->next;
		}
		printf("\n");
		free(pCrawl);
	}
}

struct LexpHeadNode
{
	int flag;
	struct LexpHeadNode* head;
	struct LexpCellNode* next;  // pointer to head node of list
	struct LexpHeadNode* back;
};

struct LexpCellNode
{
	int head;
	struct LexpCellNode* next;
	struct LexpCellNode* back;
	struct LexpHeadNode* flag;
};
struct LexpCellNode* newAdjacencyListNode1(int vertex) //diff2
{
	struct LexpCellNode* newNode =
		(struct LexpCellNode*) calloc(1, sizeof(struct LexpCellNode));
	newNode->head = vertex;
	newNode->next = NULL;
	newNode->back = NULL;
	newNode->flag = NULL;
	return newNode;
}

struct LexpHeadNode* newAdjacencyListNode2()
{
	struct LexpHeadNode* newNode =
		(struct LexpHeadNode*) calloc(1, sizeof(struct LexpHeadNode));
	newNode->head = NULL;
	newNode->next = NULL;
	newNode->back = NULL;
	newNode->flag = 0;
	return newNode;
}

void Lexp(struct Graph* graph, int *alpha, int *alphainv)
{
	int i;
	int y = graph->V;
	struct AdjacencyListNode *w;
	struct LexpHeadNode* RootHeadNode = newAdjacencyListNode2();
	struct LexpHeadNode* FirstHeadNode = newAdjacencyListNode2();
	RootHeadNode->back = NULL;
	RootHeadNode->next = NULL;
	RootHeadNode->head = FirstHeadNode;
	RootHeadNode->flag = 0;
	FirstHeadNode->flag = 0;
```

```
FirstHeadNode->head = NULL;
FirstHeadNode->back = RootHeadNode;
struct LexpCellNode* nextptr;
struct LexpHeadNode* freevar;

//Making cell nodes and putting adjacent to the first head node.
for (i = graph->V; i > 0; i--)//diff1
{
	if (i == graph->V)
	{
		struct LexpCellNode* NewCellNode = newAdjacencyListNode1(i);
		NewCellNode->flag = FirstHeadNode;
		FirstHeadNode->next = NewCellNode;
		graph->array[i].cell = NewCellNode;
	}
	else
	{
		struct LexpCellNode* NewCellNode = newAdjacencyListNode1(i);
		NewCellNode->flag = FirstHeadNode;
		graph->array[i].cell = NewCellNode;
		NewCellNode->back = graph->array[i + 1].cell;
		nextptr = graph->array[i + 1].cell;
		nextptr->next = NewCellNode;
	}
}


for (i = graph->V; i > 0; i--)
{
	// skipping the vertices
	FirstHeadNode = RootHeadNode->head;
	while (FirstHeadNode->next == NULL)
	{
		RootHeadNode->head = FirstHeadNode->head;
		FirstHeadNode->head->back = RootHeadNode;
		freevar = FirstHeadNode;
		FirstHeadNode = FirstHeadNode->head;
		free(freevar); //freeing variables.
	}

	//Select a new vertex p to delete from the doubly data structure.

	struct LexpCellNode* p;
	struct LexpCellNode* q;
	struct LexpHeadNode* headptr;

	p = FirstHeadNode->next;

	if (p->next != NULL)
	{
		FirstHeadNode->next = p->next;
		p->next->back = NULL;//FirstHeadNode
	}
	else
	{
		FirstHeadNode->next = NULL;
	}

	alpha[i] = p->head;
	alphainv[p->head] = i;

	//update2
	for (w = graph->array[p->head].head; w != NULL; w = w->next)
```

```
	{
		if (alphainv[w->dest] == 0)  // check if the order has been assigned.
		{  // delete w from the list .

			q = graph->array[w->dest].cell;

			if (q->back == NULL)
			{
				q->flag->next = q->next;
			}
			else
			{
				q->back->next = q->next;
			}

			if (q->next != NULL)
			{
				q->next->back = q->back;
			}

			headptr = q->flag->back;

			if (headptr->flag == 0)
			{
				struct LexpHeadNode* NewHeadNode = newAdjacencyListNode2();
				NewHeadNode->head = headptr->head;
				headptr->head = NewHeadNode;
				NewHeadNode->head->back = NewHeadNode;
				NewHeadNode->back = headptr;
				NewHeadNode->flag = 1;
				NewHeadNode->next = 0;
				headptr = NewHeadNode;//??
				// NewHeadNode = NewHeadNode->head;//??
			}

			// add sselected cell to the new set.

			q->next = headptr->next;
			q->flag = headptr;
			q->back = NULL;


			if (headptr->next != NULL)
			{
				headptr->next->back = q;

			}
			headptr->next = q;
		}
	}

	free(p);

	headptr = RootHeadNode;

	while (headptr->head != NULL)
	{
		headptr->flag = 0;
		headptr = headptr->head;
	}
}
fprintf(ofp, "\nLexP:\n");
fprintf(ofp, "Alpha Ordering:\n");
```

```
	for (i = 1; i <= graph->V; i++)
	{
		fprintf(ofp, "%d->", alpha[i]);
		//fprintf("alphainv: %d\n", alphainv[i]);
	}
	free(RootHeadNode);
	free(FirstHeadNode);
}


struct checkListNode* head1;
struct checkListNode* temp1;
struct checkListNode* head2;
struct checkListNode* temp2;
struct checkListNode* pivot1;
struct checkListNode* pivot2;
int fillin(struct Graph* graph, int *alpha, int *alphainv)
{
	int m = 0;
	int x, j;
	int i = 0;
	int count = 0;
	struct AdjacencyListNode *w;
	struct AdjacencyListNode *u;
	struct AdjacencyListNode *z;
	int *test = (int *)calloc((graph->V) + 1, sizeof(int));
	int *y = (int *)calloc((graph->V) + 1, sizeof(int));
	int *elim = (int *)calloc((graph->V) + 1, sizeof(int));
	struct checkListNode* pivot = (struct checkListNode*) malloc(sizeof(struct checkListNode));
	pivot->i = 0;
	pivot->j = 0;
	pivot->next = NULL;
	pivot1 = pivot;
	/*struct checkListNode* head;
	struct checkListNode* temp;*/

	fprintf(ofp, "\nFill in Edges:\n");

	for (i = 1; i < graph->V; i++)
	{
		int k = graph->V;
		int v = alpha[i];
		//eliminate duplicates in Adj[v] and compute m[v]
		struct AdjacencyListNode *prev = graph->array[v].head;
		for (w = graph->array[v].head; w != NULL; w = w->next)
		{
			if (elim[w->dest] == 0)
			{
				if (test[alphainv[w->dest]])
				{
					//delete w from adj[v]
					y[w->dest] = 0;
					count--;
					prev->next = w->next;
					//graph->array[w->dest].head = graph->array[w->dest].head->next;
					//while ()
				}

				else
				{
					/*if ((y[w->dest]==1))
					{
					printf("Fill in edges: %d, %d\n", x, w->dest);
```

```
                                        }*/
                                        //chk here
                                        //k = min(k, alphainv[w->dest]);//define min function

                                        test[alphainv[w->dest]] = 1;

                                        if (k > alphainv[w->dest])
                                        {
                                                k = alphainv[w->dest];
                                        }
                                        prev = w;
                                }
                        }

                        else { prev = w; }

        }

        for (j = 1; j <= graph->V; j++)
        {
                        if (y[j] != 0)
                        {
                                //printf("Fill in edges: %d, %d\n", x, y[j]);
                                //fprintf(ofp, "Fillin Edges:");

                                if (x < y[j])
                                {
                                        //fprintf(ofp, "%d,%d  ", x, y[j]);
                                }
                                else
                                {
                                        //fprintf(ofp, "%d,%d  ", y[j], x);
                                }

                        }
                        y[j] = 0;
        }
        m = alpha[k];

        int flag = 0;
        //add reqn fill in edges and reset test
        for (w = graph->array[v].head; w != NULL; w = w->next)
        {
                        u = graph->array[m].head;
                        //neighb[w->dest] = 1;
                        test[alphainv[w->dest]] = 0;
                        if (w->dest != m & (elim[w->dest] == 0))
                        {
                                while (u != NULL)
                                {
                                        if (u->dest == w->dest)
                                        {
                                                flag = 1;
                                        }
                                        u = u->next;
                                }

                                if (flag == 0)
                                {
                                        x = m;
                                        count = count + 1;
                                        //add w to adj[m]
                                        EDGEADD1(graph, m, w->dest, 9);
```

```
                                        y[w->dest] = w->dest;
                                        if (m < w->dest)
                                        {
                                                fprintf(ofp, "%d,%d  ", m, w->dest);//print the edge
                                                head1=newcheckListNode(m, w->dest);
                                                temp1 = pivot1->next;
                                                pivot1->next = head1;
                                                head1->next = temp1;
                                        }
                                        else
                                        {
                                                fprintf(ofp, "%d,%d  ", w->dest, m);//print the edge
                                                head1 = newcheckListNode(m, w->dest);
                                                temp1 = pivot1->next;
                                                pivot1->next = head1;
                                                head1->next = temp1;
                                        }

                                }
                                else
                                {
                                        flag = 0;
                                }
                        }
                }
                //printGraph(graph);
                //printf("count is %d\n", count);
                elim[v] = 1;
        }
        printf("count is %d\n", count);
        fprintf(ofp, "\nNumber of Fill-ins: %d\n\n", count);
        free(test);
        free(elim);
        free(y);
        free(graph);
        return count;

}



struct Lexmcell
{
        int dest;
        struct Lexmcell* next;
};

// A structure to represent an adjacency list in the LexM

struct Lexmhead
{
        struct Lexmcell* head;  // pointer of head node in LexM
};



void LexM(struct Graph* graph, int *alpha, int *alphainv)
{

        int i, k, j, p, sel_vertex, m, temp, templ, count;
        float *L = (float *)calloc((graph->V) + 1, sizeof(float));
        int *vmap = (int *)calloc((graph->V) + 1, sizeof(int));
        int *Lint = (int *)calloc((graph->V) + 1, sizeof(int));
```

```
int *vreach = (int *)calloc((graph->V) + 1, sizeof(int));
struct Lexmhead* reach;
struct AdjacencyListNode *w;
struct AdjacencyListNode *z;
struct Lexmcell* d;


for (i = 1; i <= graph->V; i++)
{
        vmap[i] = i;

}

for (i = 1; i <= graph->V; i++)
{
        L[i] = 1;
        alphainv[i] = 0;
}

k = 1;

//loop

for (i = graph->V; i > 0; i--)
{
        //select

        sel_vertex = vmap[graph->V];
        vreach[sel_vertex] = 1;
        L[sel_vertex] = 0;
        alphainv[sel_vertex] = i;
        alpha[i] = sel_vertex;

        reach = (struct Lexmhead*)calloc(k + 1, sizeof(struct Lexmhead));

        for (j = 1; j <= k; j++)
        {
                reach[j].head = NULL;
        }

        //all vertices unnumbered

        for (j = 1; j <= graph->V; j++)
        {
                if (alphainv[j] == 0)
                {
                        vreach[j] = 0;
                }
        }

        for (w = graph->array[sel_vertex].head; w != NULL; w = w->next)
        {

                // add w to reach(l(w))
                if (alphainv[w->dest] == 0)
                {
                        struct Lexmcell* node = (struct Lexmcell*) malloc(sizeof(struct Lexmcell));
                        node->dest = w->dest;
                        node->next = reach[(int)(L[w->dest])].head;
                        reach[(int)(L[w->dest])].head = node;
                        vreach[w->dest] = 1;
                        L[w->dest] = L[w->dest] + 0.5;
                        // Mark {v,w} as an edge of G
```

```
		}
	}



	for (j = 1; j <= k; j++)
	{
		while (reach[j].head != NULL)
		{
			//delete a vertex w from reach(j)??
			d = reach[j].head;
			reach[j].head = reach[j].head->next;

			for (z = graph->array[d->dest].head; z != NULL; z = z->next)
			{

				if (vreach[z->dest] != 1)
				{
					vreach[z->dest] = 1;

					if (L[z->dest] > j)
					{
						struct Lexmcell* nodez = (struct Lexmcell*)
malloc(sizeof(struct Lexmcell));

						nodez->dest = z->dest;
						nodez->next = reach[(int)(L[z->dest])].head;
						reach[(int)(L[z->dest])].head = nodez;
						L[z->dest] = L[z->dest] + (0.5);
						// mark {v,z} as an edge of G
					}
					else
					{
						struct Lexmcell* nodez = (struct Lexmcell*)
malloc(sizeof(struct Lexmcell));

						nodez->dest = z->dest;
						nodez->next = reach[j].head;
						reach[j].head = nodez;
					}
				}
			}
			free(d);
		}
	}

	for (p = 1; p <= graph->V; p++)
	{
		if (L[p] != 0)
		{
			Lint[p] = ((10 * L[p]));
		}
	}


	for (m = 1; m <= graph->V; m++)
	{
		vmap[m] = m;
	}

	free(reach);
	int size = (graph->V) + 1;
	int large = (20 * k) + 1;
	int *semiSorted, *bsort;
	semiSorted = (int*)calloc(size, sizeof(int));
```

```
bsort = (int*)calloc(size, sizeof(int));
int significantDigit = 1;
int largestNum = large;

// Loop until we reach the largest significant digit
while (largestNum / significantDigit > 0)
{

	int bucket[10] = { 0 };

	// Counts the number of "keys" or digits that will go into each bucket
	for (p = 1; p < size; p++)
		bucket[(Lint[p] / significantDigit) % 10]++;

	/*
	* Add the count of the previous buckets,
	* Acquires the indexes after the end of each bucket location in the array
	* Works similar to the count sort algorithm
	*/
	for (p = 1; p < 10; p++)
		bucket[p] += bucket[p - 1];

	// Use the bucket to fill a "semiSorted" array
	for (p = size - 1; p > 0; p--)
	{
		int l = --bucket[(Lint[p] / significantDigit) % 10];
		semiSorted[l] = Lint[p];
		bsort[l] = vmap[p];

	}

	for (p = 0; p < size - 1; p++)
		Lint[p + 1] = semiSorted[p];

	for (p = 0; p < size - 1; p++)
		vmap[p + 1] = bsort[p];

	// Move to next significant digit
	significantDigit *= 10;


}

m = 1;

while (Lint[m] == 0)
{
	m = m + 1;
}

temp = 0;
templ = 0;
count = 0;

for (m; m <= graph->V; m++)
{
	if (Lint[m] == temp)
	{
		Lint[m] = count;
	}
	else
	{
		count = count + 1;
```

```
                                templ = Lint[m];
                                Lint[m] = count;
                                temp = templ;
                        }

                }

                for (m = 1; m <= graph->V; m++)
                {
                        L[vmap[m]] = Lint[m];
                }

                k = Lint[graph->V];
        }

        //printf("\n");
        fprintf(ofp, "\nLexM:\n");
        fprintf(ofp, "Alpha Ordering:\n");
        for (i = 1; i <= graph->V; i++)
        {

                fprintf(ofp, "%d->", alpha[i]);
                //fprintf(ofp, "%d->", alpha1[i]);
        }
        //printf("\n");
        free(vmap);
        free(Lint);
        free(vreach);
        free(L);

}

rowchange(double ** matrix,int* ordermap, int src, int dest,int size)
{
        double c;
        int i;
        for (i = 1; i <= size+1; i++)
        {
                c = matrix[ordermap[src]][i];
                matrix[ordermap[src]][i] = matrix[dest][i];
                matrix[dest][i] = c;

                //printmatrix(matrix, size);

                /*c = matrix[i][ordermap[src]];
                matrix[i][ordermap[src]] = matrix[i][dest];
                matrix[i][dest] = c;
         */
        }

        //printmatrix(matrix,size);
}

double ** matrixchange(double ** matrix, int* ordermap, int *alpha, int M)
{
        int l1,i,j;
        double tempaks;

        for (i = 1; i<M + 1; i++)
        {
                if (ordermap[i] != alpha[i])
                {
                        //l=i;
```

```
                        for (j = 1; j <= M; j++)
                        {
                                if (ordermap[j] == alpha[i])
                                {
                                        tempaks = ordermap[j];
                                        ordermap[j] = ordermap[i];
                                        ordermap[i] = tempaks;
                                        break;
                                }
                        }
                        l1 = j;
                        tempaks = matrix[i][M+1];
                        matrix[i][M + 1] = matrix[l1][M + 1];
                        matrix[l1][M + 1] = tempaks;
                        for (j = 1; j<M + 1; j++)
                        {
                                tempaks = matrix[i][j];
                                matrix[i][j] = matrix[l1][j];
                                matrix[l1][j] = tempaks;
                        }
                        for (j = 1; j<M + 1; j++)
                        {
                                tempaks = matrix[j][i];
                                matrix[j][i] = matrix[j][l1];
                                matrix[j][l1] = tempaks;
                        }
                }
        }

        return matrix;
}

colchange(double ** matrix, int* ordermap, int* ordermap1, int src, int dest, int size)
{
        double c;
        int i;
        for (i = 1; i <= size; i++)
        {
                c = matrix[i][ordermap[src]];
                matrix[i][ordermap[src]] = matrix[i][dest];
                matrix[i][dest] = c;


        }

        //c = ordermap[src];

        c = ordermap[src];
        ordermap[src] = dest;
        ordermap[ordermap1[dest]] = c;

        c=ordermap1[dest];
        ordermap1[dest]=src;
        ordermap1[src] = c;


        //printmatrix(matrix, size);

        //printf("\n\n");

        /*for (i = 1; i <= size; i++)
        {
                printf("%d ", ordermap[i]);
```

```
	}*/

}

double * guass(double** array, int size, int* alpha,int* alphainv)
{
	fprintf(ofp, "Inside guass function\n");
	//printmatrix(array, size);

		int i, j, k, n,flag,count,error;
		error = 0;
		count = 0;
		flag = 0;
		float c;
		float sum = 0.0;
		n = size;

		struct checkListNode* pivot = (struct checkListNode*) malloc(sizeof(struct checkListNode));
		pivot->i = 0;
		pivot->j = 0;
		pivot->next = NULL;
		pivot2 = pivot;

		double* x = (double*)calloc(size + 1, sizeof(double));

		for (j = 1; j <= n; j++) /* loop for the generation of upper triangular matrix*/
		{
			for (i = 1; i <= n; i++)
			{
				if (i>j)
				{
					//printf("\n");
					//printmatrix(array, size);

					if (array[j][j] == 0)
					{
						printf("The GEM stops here as element is zero");
						error = 1;
						goto here;
					}

					c = array[i][j] / array[j][j];

					array[i][j] = 0;

					for (k = j+1; k <= n + 1; k++)
		{
			if (array[i][k] == 0 & k!=n+1 )
				{
					flag = 1;
			}
					array[i][k] = array[i][k] - c*array[j][k];

				if (flag == 1 & k != n + 1  )
				{
					if (array[i][k] !=0 & k>i)
					{
						count++;

					//if (alpha[i] < alpha[k])
							//{
					fprintf(ofp, "(%d,%d %lg) ", alpha[i], alpha[k], array[i][k]);
					head2 = newcheckListNode(alpha[i], alpha[k]);
```

```
                                                                        temp2 = pivot2->next;
                                                                        pivot2->next = head2;
                                                                        head2->next = temp2;
                                        /*                              }
                                                                        else
                                                                        {
                                        fprintf(ofp, "(%d,%d %lf) ", alpha[k], alpha[i], array[i][k]);
                                        head2 = newcheckListNode(alpha[k], alpha[i]);
                                                                        temp2 = pivot2->next;
                                                                        pivot2->next = head2;
                                                                        head2->next = temp2;
                                                                        }*/
                                                                }

                                                                flag = 0;
                                                        }
                                        }
                                }
                        }

        }




        if (array[n][n] != 0)
        {
                x[n] = array[n][n + 1] / array[n][n];
        }
        else
        {
                printf("GEM has stopped here");
        }
        /* this loop is for backward substitution*/

        for (i = n - 1; i >= 1; i--)
        {
                sum = 0;

                for (j = i + 1; j <= n; j++)
                {
                        sum = sum + array[i][j] * x[j];
                }

                if (array[i][i] != 0)
                {
                        x[i] = (array[i][n + 1] - sum) / array[i][i];

                }

                else
                {
                        printf("GEM has stopped here");
                }

        }
here:
        fprintf(ofp,"\nNon zero element count = %d", count);
        printf("\nNon zero element count = %d \n", count);
        fprintf(ofp,"\nThe solution is: \n");
        if (error == 0)
        {
```

```
                        for (i = 1; i <= size; i++)
                        {
                                fprintf(ofp, "x%d=%f\t", i, x[alphainv[i]]); /* x1, x2, x3 are the required solutions*/
                        }

                        fprintf(ofp, "\n");
                }
                else
                {
                        fprintf(ofp, "\nNo Solution exists");
                        error = 0;
                }
                int chkcnt=0;
                struct checkListNode* tempchk1;
                struct checkListNode* tempchk2;
                tempchk1 = pivot1;
                tempchk2 = pivot2;
                while (pivot1 != NULL)
                {
                        while (pivot2 != NULL)
                        {
                                if ((pivot1->i == pivot2->i & pivot1->j == pivot2->j )|| (pivot1->i == pivot2->j & pivot1-
>j == pivot2->i))
                                {
                                        chkcnt++;
                                        pivot1->i=0;
                                        pivot1->j=0;
                        }
                                pivot2 = pivot2->next;
                        }
                        pivot2 = tempchk2;
                        pivot1 = pivot1->next;
                }
                pivot1 = tempchk1;

                /*printf("\n");
                while (pivot1 != NULL)
                {
                        if (pivot1->i != 0 & pivot1->j != 0)
                        {
                                printf("%d %d ", pivot1->i, pivot1->j);
                        }
                        pivot1 = pivot1->next;
                }*/

                printf("\n");
                chkcnt--;
                fprintf(ofp, " Matched=%d Notmatched=%d\n", chkcnt, count-chkcnt);
                //printmatrix(array, size);
 return x;
}

printmatrix(double ** array, int size)
{
        fprintf(ofp,"\n\n");
        int i, j,temp;
        for (i = 1; i <= size; i++)
        {
                for (j = 1; j <= size+1; j++)
                {
                        //temp = (int)(array[i][j]);
                        fprintf(ofp,"%lg  ", array[i][j]);
                }
```

```
                        fprintf(ofp,"\n");
                }
}
int main(int argc, char *argv[])
{
                clock_t t1, t2;
                t1 = clock();
                int ret_code;
                char *inFileString, *outFileString;
                MM_typecode matcode;
                inFileString = argv[1];
                outFileString = argv[2];
                //FILE *f, *ofp;
                int M, N, nz;
                int i, *I, *J;
                double *val;
                /*if (argc < 2)
                {
                fprintf(stderr, "Usage: %s [martix-market-filename]\n", argv[0]);
                exit(1);
                }
                else
                {
                if ((f = fopen(argv[1], "r")) == NULL)
                exit(1);
                }*/

                f = fopen(inFileString, "r");

                if (f == NULL)
                {
                        printf("File null\n");
                        exit(1);
                }
                if (f != NULL)
                {
                        printf("FIle is not null\n");
                }


                if (mm_read_banner(f, &matcode) != 0)
                {
                        printf("Could not process Matrix Market banner.\n");
                        exit(1);
                }

                /*  This is how one can screen matrix types if their application */
                /*  only supports a subset of the Matrix Market data types.     */

                if (mm_is_complex(matcode) && mm_is_matrix(matcode) &&
                        mm_is_sparse(matcode))
                {
                        printf("Sorry, this application does not support ");
                        printf("Market Market type: [%s]\n", mm_typecode_to_str(matcode));
                        exit(1);
                }

                /* find out size of sparse matrix .... */

                if ((ret_code = mm_read_mtx_crd_size(f, &M, &N, &nz)) != 0)
                        exit(1);
```

```
/* reseve memory for matrices */

I = (int *)malloc(nz * sizeof(int));
memcount = memcount + (nz * sizeof(int));
J = (int *)malloc(nz * sizeof(int));
memcount = memcount + (nz * sizeof(int));
val = (double *)malloc(nz * sizeof(double));
memcount = memcount + (nz * sizeof(double));


/* NOTE: when reading in doubles, ANSI C requires the use of the "l"  */
/*   specifier as in "%lg", "%lf", "%le", otherwise errors will occur */
/*  (ANSI C X3.159-1989, Sec. 4.9.6.2, p. 136 lines 13-15)          */

for (i = 0; i < nz; i++)
{
	fscanf(f, "%d %d %lg\n", &I[i], &J[i], &val[i]);
	//I[i]--;  /* adjust from 1-based to 0-based */
	//J[i]--;
}

for (i = 0; i < nz; i++)
{
	if (val[i] == 0)
	{
		I[i] = 0;
		J[i] = 0;
	}
}

int j;
for (i = 0; i < nz; i++)
{
	for (j = i + 1; j < nz; j++)
	{
		if ((I[i] == J[j]) & (J[i] == I[j]) & I[i]!=0 & J[i]!=0 )
		{
			if (val[i] >= val[j])
			{
				val[i] = val[j];
			}

			I[j] = 0;
			J[j] = 0;
			val[j] = 0;


		}

	}
}

if (f != stdin) fclose(f);

/************************/
/* now write out matrix */
/************************/

mm_write_banner(stdout, matcode);
mm_write_mtx_crd_size(stdout, M, N, nz);
struct Graph* graph = CREATEGRAPH(M);
//memcount = memcount + ((N + 1)*sizeof(struct AdjacencyListNode));
struct Graph* graph2 = CREATEGRAPH(M);
```

```
//memcount = memcount + ((N + 1)*sizeof(struct AdjacencyListNode));
struct Graph* graph3 = CREATEGRAPH(M);
//memcount = memcount + ((N + 1)*sizeof(struct AdjacencyListNode));


/*struct Graph* graph4 = CREATEGRAPH(M);
memcount = memcount + ((N + 1)*sizeof(struct AdjacencyListNode));*/

for (i = 0; i < nz; i++)
{
	EDGEADD(graph, I[i], J[i], val[i]);
	EDGEADD(graph2, I[i], J[i], val[i]);
	EDGEADD(graph3, I[i], J[i], val[i]);
	//fprintf(stdout, "%d %d %20.19g\n", I[i] + 1, J[i] + 1, val[i]);
}
//fprintf(stdout, "%d %d %20.19g\n", I[i] + 1, J[i] + 1, val[i]);
//printGraph(graph);

char outputFilename[] = "major.txt";
ofp = fopen(outFileString, "w");
if (ofp == NULL)
{
	printf("file is not read properly\n");
}
fprintf(ofp, "Name of the file %s \n" ,inFileString);
fprintf(ofp, "Size: %d * %d\nNonZero Elements: %d\n", M, N, nz);

double* b;
b = (double *)calloc(M + 1, sizeof(double));
memcount = memcount + ((M + 1)*sizeof(double));


fprintf(ofp, "\nB matrix values are \n");

for (i = 1; i <= M; i++)
{
	if (mm_is_integer(matcode))
	{
		b[i] = rand() % 10;
		fprintf(ofp, "%d ", b[i]);
	}
	else
	{
		b[i] = rand() % 200;
		b[i] = b[i] / 1.0000;
		b[i] = b[i] / RAND_MAX;
		fprintf(ofp, "%lg ", b[i]);
	}
	}

fprintf(ofp, "\n");


//////////////////////////////////////////////lexp////////////////////

int	*alpha1lexp;
int	*alpha2lexp;
alpha1lexp = (int *)calloc(M + 1, sizeof(int));
//memcount = memcount + ((N + 1)*sizeof(int));
alpha2lexp = (int *)calloc(M + 1, sizeof(int));
//memcount = memcount + ((N + 1)*sizeof(int));
int count = 0;
// ordering of elimination
```

```
//memcount = memcount + (3 * (N + 1)*sizeof(int));
Lexp(graph, alpha1lexp, alpha2lexp);
//memcount = memcount + ((N + 1)*sizeof(struct LexpCellNode));
printf("\n LexP number of Fillin ");
count = fillin(graph, alpha1lexp, alpha2lexp);

double **matrix;
// allocate the 2d array
matrix = (double **)calloc(M + 1, sizeof(double*));
for (i = 0; i < M + 1; i++)
        matrix[i] = (double*)calloc(M + 2, sizeof(double));
memcount = memcount + ((M*M)*sizeof(double));

//printmatrix(matrix, M);
for (i = 0; i < nz; i++)
{
        matrix[I[i]][J[i]] = val[i];
        matrix[J[i]][I[i]] = val[i];
}


for (i = 1; i <= M; i++)
{
        matrix[i][M + 1] = b[i];
}

//printmatrix(matrix, M);

int     *ordermap;


ordermap = (int *)calloc(M + 1, sizeof(int));
memcount = memcount + ((M + 1)*sizeof(int));

for (i = 1; i <= M; i++)
{
        ordermap[i] = i;
}

//do swapping

matrix=matrixchange(matrix, ordermap, alpha1lexp, M);

double* l;

//printf("\n");

l=guass(matrix, M,alpha1lexp,alpha2lexp);
memcount = memcount + ((M + 1)*sizeof(double));




free(matrix);
free(ordermap);
//free(ordermap1);
free(alpha1lexp);
free(alpha2lexp);
//free(pivot1);
//free(pivot2);
```

////////////////////////////////////lexm//////////////////////

```
int          *alpha1lexm;
int          *alpha2lexm;
alpha1lexm = (int *)calloc(M + 1, sizeof(int));
//memcount = memcount + ((N + 1)*sizeof(int));
alpha2lexm = (int *)calloc(M + 1, sizeof(int));
//memcount = memcount + ((N + 1)*sizeof(int));

LexM(graph2, alpha1lexm, alpha2lexm);
//memcount = memcount + ((N + 1)*sizeof(struct Lexmcell));
printf("\n LexM number of Fillin ");
count = fillin(graph2, alpha1lexm, alpha2lexm);
//}


// allocate the 2d array
matrix = (double **)calloc(M + 1, sizeof(double*));
for (i = 0; i < M + 1; i++)
             matrix[i] = (double*)calloc(M + 2, sizeof(double));
//printmatrix(matrix, M);
for (i = 0; i < nz; i++)
{
             matrix[I[i]][J[i]] = val[i];
             matrix[J[i]][I[i]] = val[i];
}

for (i = 1; i <= M; i++)
{
             matrix[i][M + 1] = b[i];
}

//printmatrix(matrix, M);

ordermap = (int *)calloc(M + 1, sizeof(int));

for (i = 1; i <= M; i++)
{
             ordermap[i] = i;
}


matrix=matrixchange(matrix, ordermap, alpha1lexm, M);
l=guass(matrix, M,alpha1lexm,alpha2lexm);
free(matrix);
free(ordermap);
free(alpha1lexm);
free(alpha2lexm);
//free(ordermap1);


////////////////////////////original////////////////////////////////

int          *alpha1ori;
int          *alpha2ori;
alpha1ori = (int *)calloc(M + 1, sizeof(int));
//memcount = memcount + ((N + 1)*sizeof(int));
alpha2ori = (int *)calloc(M + 1, sizeof(int));
//memcount = memcount + ((N + 1)*sizeof(int));
fprintf(ofp, "\nOriginal Ordering:\n");
for (i = 1; i <= graph3->V; i++)
{
```

```
                alpha1ori[i] = i;
                alpha2ori[i] = i;
                fprintf(ofp, "%d->", alpha1ori[i]);
        }
        printf("\n Original ordering number of Fillin ");
        count = fillin(graph3, alpha1ori, alpha2ori);

        // allocate the 2d array
        matrix = (double **)calloc(M + 1, sizeof(double*));
        for (i = 0; i < M + 1; i++)
                matrix[i] = (double*)calloc(M + 2, sizeof(double));
        //printmatrix(matrix, M);
        for (i = 0; i < nz; i++)
        {
                matrix[I[i]][J[i]] = val[i];
                matrix[J[i]][I[i]] = val[i];
        }

        for (i = 1; i <= M; i++)
        {
                matrix[i][M + 1] = b[i];
        }

        //printmatrix(matrix, M);

        ordermap = (int *)calloc(M + 1, sizeof(int));

        for (i = 1; i <= M; i++)
        {
                ordermap[i] = i;
        }

        matrix = matrixchange(matrix, ordermap, alpha1ori, M);
        l=guass(matrix, M,alpha1ori,alpha2ori);
        free(matrix);
        free(ordermap);

        fprintf(ofp, "\n");
        fclose(ofp);
        t2 = clock();
        float diff = (((float)t2 - (float)t1)*0.000001);
        printf("\n \n Total Execution time %f seconds \n", diff);
        printf("\n \n Total Memory used is %llu Kilobytes \n", memcount / 1024);
        return 0;
}
```