Sparse Matrix Ordering and Gaussian Elimination

ECE 3652 - Fundamentals of Computer Engineering
Term Paper

Bulut F. Ersavas
bersavas@coe.neu.edu
December 8, 2002

Abstract

Applying special techniques to solve large linear systems with sparse matrices (those with very large number of zero elements) have been gaining increasing popularity recently. By making use of the zero elements in a sparse matrix, significant gains can be achieved in space and performance. In this paper, I will describe one of these special techniques what is called sparse matrix reordering (specifically Cuthill-McKee Ordering) and show the improvements that such operation provides in solving the linear systems using Gaussian Elimination.

1. Introduction

In mathematics, science and engineering linear systems has very important place. Many applications involve solving linear systems. Therefore, there have been great efforts to solve or approximate solutions to such systems efficiently. Before proceeding, let us describe what a linear system is:

Given an *n x n* real matrix *A* and a real n-vector *b*, find *x* belonging to R^{nxn} such that:

$$Ax = b$$

This equation is a linear system in which *A* is the *coefficient matrix*, *b* is the *right-hand side vector* and *x* is the *vector of unknowns*.

The complexity of the linear systems varies with the size of matrix *A* and *b* (i.e. with n) and the content of matrix *A*. Depending on what is included in matrix *A*, several enhancements can be done in algorithms that solve such linear systems.

In applications of various disciplines of engineering, a very common matrix type is the sparse matrices. A sparse matrix is described as a matrix that has very few non-zero elements. The advantage of sparse matrices is that sophisticated techniques, which benefit from large number of zero elements and their locations, can be applied. [1]

One very frequently used enhancement on sparse matrices is *reordering* which is permuting its rows or columns or both rows and columns. By applying reordering algorithms, the zero and non-zero elements of a sparse matrix are rearranged such that the solver algorithm deals with it much more efficiently.

In this paper, I will implement the Cuthill-McKee Ordering, which is one of the most popular matrix reordering; show its effects on one of the direct linear system solving methods namely the Gaussian Elimination. The organization of the paper goes on as follows: In the next section, I will provide some background information on matrix reordering and describe Cuthill-McKee Algorithms. In section three, I will explain how Gaussian Elimination works and talk about a straightforward C++ implementation of it. In the later section, I will present the benchmark results of the described algorithms. Finally, I will give a brief conclusion of the topics covered in this paper.

2. Sparse Matrix Ordering

In this section, I will describe some important concepts in matrix ordering. Then, I will present and discuss the implementation of one of the most commonly used ordering algorithms known as Cuthill-McKee Ordering.

The sparse matrices used throughout this research (except when performing the Gaussian Elimination) are represented with graphs. Using graphs to represent sparse matrices helps in manipulating them and prevents wasting space to store zero elements of them. The non-zero pattern of a sparse matrix of a linear system is modeled with a graph G(V,E), whose n vertices in V corresponds to n unknowns and where there is an edge from vertex i to vertex j if the coefficient matrix A_{ij} is not zero. [1, 2]

2.1. Background

One very common operation in linear algebra is permuting the rows and/or columns of a sparse matrix. This introduces the terminology of permutation and permutation matrices, which are defined in [1] as follows:

Let A be a matrix and $\pi = \{i_1, i_2, i_3, \ldots, i_n\}$ a permutation of the set $\{1, 2, 3, \ldots, n\}$. Then the matrices;

$$A_{\pi,*} = \{a_{\pi(i),j}\}_{i=1,\ldots,n;\ j=1,\ldots,m}$$
$$A_{*,\pi} = \{a_{i,\pi(j)}\}_{i=1,\ldots,n;\ j=1,\ldots,m}$$

are called *row π-permutation* and *column π-permutation* of A, respectively. Furthermore,

$$A_{\pi,*} = P_\pi A, \quad A_{*,\pi} = A\, Q_\pi$$

where, P_π, and Q_π are called *permutation matrices*.

When the permutation is applied to both columns and rows of A, the operation is called *symmetric permutation* and it satisfies the following relation:

$$A_{\pi,\pi} = P_\pi^T A\, P_\pi$$

As an example, let's consider the linear system Ax = b where,

$$A = \begin{bmatrix} a11 & 0 & a13 & 0 \\ 0 & a22 & a23 & a24 \\ a31 & a32 & a33 & 0 \\ 0 & a42 & 0 & a44 \end{bmatrix}$$

using permutation π = {1, 3, 2, 4} we get the following linear system after symmetric permutation:

$$\begin{bmatrix} a11 & a13 & 0 & 0 \\ a31 & a33 & a32 & 0 \\ 0 & a23 & a22 & a24 \\ 0 & 0 & a42 & a44 \end{bmatrix} \begin{bmatrix} x1 \\ x3 \\ x2 \\ x4 \end{bmatrix} = \begin{bmatrix} b1 \\ b3 \\ b2 \\ b4 \end{bmatrix}$$

2.2. Cuthill-McKee Ordering

Matrix reordering is nothing but finding the permutation matrix that results in sparse matrix ordering, which relaxes the space and computational requirements of the linear system solvers.

In this sub-section I will describe one of the most commonly used ordering algorithms: Cuthill-McKee Ordering. The goal of this algorithm is to reduce the bandwidth (i.e. the maximum distance between two adjacent vertices) of a graph by reordering the indices assigned to each vertex. [1] Very broadly, the graph is traversed with Breadth First Search (BFS); unvisited adjacent vertices are placed in a production queue and the order each node visited is stored to form the permutation matrix.

Figure 2.2 shows the matrix generated by reordering the one in figure 2.1 using the Cuthill-McKee Ordering Algorithm.

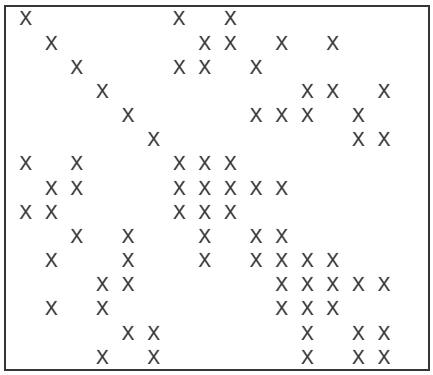

Figure 2.1 Placement of non-zero elements in a 15x15 sparse matrix.

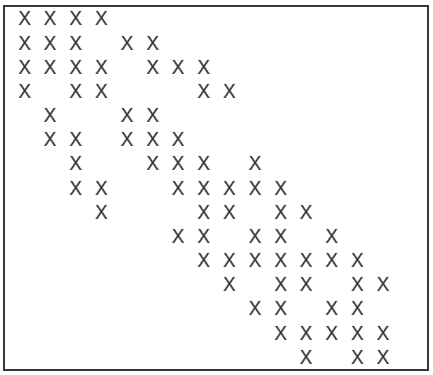

Figure 2.2 After Cuthill-McKee Ordering with starting node 3.

More detailed explanation of the algorithm follows:

- A production queue is defined and filled with a starting node initially,
- A one-dimensional matrix is generated to store the sequence each node is visited,
- Then, a node is pulled from the production queue to be visited (until all the nodes in the graph are visited)
    - All the unvisited neighboring nodes of the currently visited node are pushed to the production queue,
    - The current node is stored in the order-tracking matrix and marked as visited.
    - These steps repeated for every node in the graph.

One obvious variation of this algorithm is the reverse Cuthill-McKee Algorithm, which is observed to be yielding a better scheme for sparse Gaussian Elimination [1]. In the reverse Cuthill-McKee algorithm, the generated permutation matrix (including the order that each node is visited) is simply reversed.

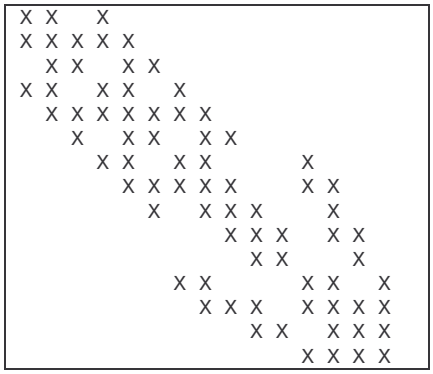

Figure 2.3 After reverse Cuthill-McKee Ordering with starting node 3.

The matrix generated from the matrix in figure 2.1 by this reverse algorithm is shown in figure 2.3. Both matrices generated in figure 2.2 and 2.3 consist of small arrow sub-matrices, but they differ in the direction that the arrows are pointing to. The matrix generated by reverse Cuthill-McKee algorithm includes the downward pointing arrow

sub-matrices which results in better performance in Gaussian Elimination because of the way it is performed (left to right and down to bottom, see next section for more information). This is also proven by the test results given in section 4.

C++ implementation of Cuthill-McKee Ordering Algorithm is given in figure 2.4. For the given function, user is asked to provide the direction (forward or reverse) of the ordering and the starting node of the graph traversal as well as the input, output and permutation matrices.

```
void cmkOrdering(const Graph &g, Graph &gn, bool reverse, int
start_node, int *iperm0)
{
  int *marker = new int[g.v()+1] = {0};
  int *iperm = new int[g.v()+1] = {0};    // Permutation array P
  int *riperm = new int[g.v()+1] = {0};   // Reverse permutation array
  int *ip_rev = new int[g.v()+1] = {0};
  Queue<int> levset;                      // Level Set
  int i = 0, j = 0, n = 0, next = 2;

  levset.put(start_node);  // i0 is the initial node of the level set
  marker[start_node] = 1;
  iperm[1] = start_node;

  while (!levset.empty())
    {
      int v = levset.get();
      Graph::adjIterator A(g,v);
      for (int w=A.beg(); !A.end(); w=A.nxt())
      {
        if (marker[w] == 0)
          {
            marker[w] = 1;
            levset.put(w);
            iperm[next] = w;
            next++;
          }
      }
    }
  n=g.v();
  if (reverse)
    {
      for (i=1; i<=n; i++)
      riperm[n-i+1] = iperm[i];
      cout << "Inverse permutation P of Above Matrix: " << endl;
      iperm = riperm;
    }
  else
    cout << "Permutation P of Above Matrix: " << endl;

  for (i=1; i<=g.v(); i++)
    {
      cout << iperm[i] << " ";
      ip_rev[iperm[i]] = i;
    }
  cout << endl;
```

```
  for (int v=1; v<=g.v(); v++)
    {
      Graph::adjIterator A(g,v);
      for (int w=A.beg(); !A.end(); w=A.nxt())
      {
        gn.insert(Edge(ip_rev[v],ip_rev[w],A.val()));
      }
    }

  // Update the input permutation matrix and convert its type
  for (i=1; i<=g.v(); i++)
    {
      iperm0[i-1] = iperm[i] - 1;
    }
}
```

Figure 2.4 C++ implementation of (reverse) Cuthill-McKee Ordering Algorithm

One important application step of the Cuthill-McKee Algorithm is the selection of the starting node. Different starting nodes (may) result in different orderings and the selection of that node depends on the preference of the later analysis or solver algorithm. For example, the selection of node 1 as the starting node (instead of 3) in our example case results in the matrix given in figure 2.5 which is clearly more suitable for Gaussian Elimination solver. In my implementation, the user is asked to pick the starting node to run the reordering algorithm. There have been several researches on the algorithms to find the best starting node. [2] describes and implements one of these algorithms called pseudo-peripheral pair heuristic. Such algorithms were left out of the scope of this paper.

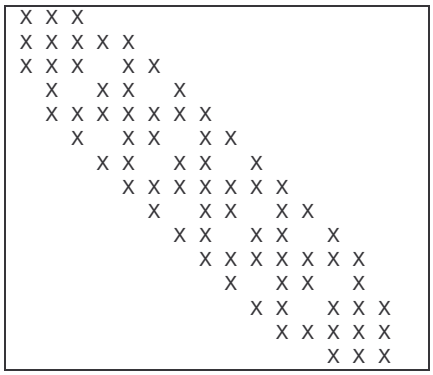

Figure 2.5 After reverse Cuthill-McKee Ordering with starting node 1.

3. Gaussian Elimination

In this section, the concept of Gaussian Elimination will be introduced and an example on how to use it will be given. Afterwards, a simple C++ implementation for applying it to n x n matrices will be described.

3.1 Background

Gaussian Elimination is a method for solving linear systems (or matrix equations) of the form: $Ax = b$, i.e.

$$\begin{bmatrix} a11 & a12 & \dots & a1n \\ a21 & a22 & \dots & a2n \\ \dots & \dots & \dots & \dots \\ an1 & an2 & \dots & ann \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ \dots \\ xn \end{bmatrix} = \begin{bmatrix} b1 \\ b2 \\ \dots \\ bn \end{bmatrix}$$

Starting with the system above, we form the augmented matrix of the form:

$$\left[\begin{array}{cccc|c} a11 & a12 & \dots & a1n & b1 \\ a21 & a22 & \dots & a2n & b2 \\ \dots & \dots & \dots & \dots & \dots \\ an1 & an2 & \dots & ann & bn \end{array}\right]$$

Afterwards, we apply a sequence of elementary row operations to the augmented matrix to transform the coefficient part to upper triangular form. The elementary operations to be applied are as follows:

- multiply a row by a non-zero real number c,
- swap two rows,
- add c times one row to another one.

The upper triangular form of the augmented matrix is as follows:

$$\left[\begin{array}{cccc|c} a'11 & a'12 & \dots & a'1n & b'1 \\ 0 & a'22 & \dots & a'2n & b'2 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & a'nn & b'n \end{array}\right]$$

Finally, we solve each row of the matrix in the following form to find the value of each unknown element of vector x.

$$x_i = \frac{1}{a'_{ii}}(b'_i - \sum_{j=i+1}^{n} a'_{ij}\, x_j) \qquad (3.1)$$

As an example, let’s consider the following linear system:

$$\begin{bmatrix} 1 & -3 & 1 \\ 2 & -8 & 8 \\ -6 & 3 & -15 \end{bmatrix}\begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} 4 \\ -2 \\ 9 \end{bmatrix} \Rightarrow \left[\begin{array}{ccc|c} 1 & -3 & 1 & 4 \\ 2 & -8 & 8 & -2 \\ -6 & 3 & -15 & 9 \end{array}\right]$$

First, in the augmented matrix given on the right, we multiply the first row with -2 and add it to the second row, and multiply it with 6 and add it to the third row. The resulting matrix is:

$$\left[\begin{array}{ccc|c} 1 & -3 & 1 & 4 \\ 0 & -2 & 6 & -10 \\ 0 & -15 & -9 & 33 \end{array}\right]$$

Then, to reduce the coefficients of second and third row, divide them by 2 and 3 respectively and finally multiply the second one by -5 and add to the last row to eliminate a'_{23} as shown below.

$$\left[\begin{array}{ccc|c} 1 & -3 & 1 & 4 \\ 0 & -1 & 3 & -5 \\ 0 & -5 & -3 & 11 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} 1 & -3 & 1 & 4 \\ 0 & -1 & 3 & -5 \\ 0 & 0 & -18 & 36 \end{array}\right]$$

The resulting augmented matrix is used in equation 3.1 and the values for elements of x is found as follows: $x_3 = -2$, $x_2 = -1$, $x_1 = 3$.

One last comment before we continue to the next section is about the singular matrices. If all the elements on a column are zero, this shows that the matrix is a singular one, that is, the n equations formed by the rows are not all independent and therefore there isn't a unique solution for the system.

3.2. C++ Implementation

The segment of the C++ source code, which performs the Gaussian Elimination calculations, is given in figure 3.1. The algorithm is described as follows:

- Given the augmented matrix, iterate through each column (foreach column i),
  - Starting from row i+1, find the row with largest element value in i^{th} column,
  - Swap the largest row with i^{th} row,
  - Check to see if this is a singular matrix (if a[i][i] = 0), exit if it is,
  - Starting from row i+1, eliminate the members of i^{th} column,
- Perform the back substitution and find the elements of unknown vector x.

```
for (i=0;i<mSize;i++) {

  // Find the largest value row
  max = i;
  for (j=i+1;j<mSize;j++) {
    if (abs(a[i][j]) > abs(a[i][max]))
    max = j;
  }

  // Swap the largest row with the ith row
  for (k=i;k<mSize+1;k++) {
    temp = a[k][i];
    a[k][i] = a[k][max];
    a[k][max] = temp;
  }

  // Check to see if this is a singular matrix
  // Return false if it is
  if (abs(a[i][i]) == 0.0)
    return false;

  // Starting from row i+1, eliminate the elements of the ith column
  for (j=i+1;j<mSize;j++) {
    if (a[i][j] != 0)
    {
      for (k=mSize;k>=i;k--) {
        a[k][j] -= a[k][i] * a[i][j] / a[i][i];
        nEleminations++;
      }
    }
  }
}

// Perform the calculations
for (j=mSize-1;j>=0;j--) {
  temp = 0;
  for (k=j+1;k<mSize;k++)
    temp += a[k][j] * x[k];
  x[j] = (a[mSize][j] - temp) / a[j][j];
}
```

Figure 3.1 C++ Implementation of Gaussian Elimination

4. Benchmark Results

As we have seen in the previous section, if we have an upper triangular coefficient matrix (i.e. A), we don't need to do any elimination and we can directly calculate the elements of the unknown matrix *x*. This is an extreme case and our goal is to approach to this case by reordering the sparse coefficient matrix we have. The closer we get to a single diagonal matrix, the more we save time from elimination.

The benchmark I've run on 10 instances with different sizes shows the improvement that the (reverse) Cuthill-McKee ordering achieves. In some cases the achievement is significant and in some cases it is negligible because of the placement of the zero elements in the original matrix. Please also note that the orderings performed on the coefficient matrices are not the optimum ones because of random selection of the starting node for Cuthill-McKee ordering algorithm. Several algorithms such as the pseudo-peripheral pair heuristic [2] can be applied to pick the best starting node.

Before presenting the results, let us investigate the best case on a simple coefficient matrix. The matrix given below (on the left) would take 8 eliminations before the algorithm starts calculating the unknowns. However, after reverse Cuthill-McKee ordering (on the right) the matrix can directly be solved without requiring any elimination to be done.

$$\begin{bmatrix} 3 & 0 & 5 \\ 9 & 0 & 0 \\ 2 & 1 & 0 \end{bmatrix} \xrightarrow{reverseCuthill-McKeeOrdering} \begin{bmatrix} 1 & 0 & 2 \\ 0 & 5 & 3 \\ 0 & 0 & 9 \end{bmatrix}$$

Table 4.1 summarizes the benchmarks run. As expected, reverse Cuthill-McKee algorithm provides the best ordering for Gaussian Elimination among others because of the downward arrow shape it tends to generate. When the input matrix is symmetric, reordered matrix becomes closer to a diagonal one (thus, upper triangular) and Gaussian Elimination algorithm needs less elimination and performs better.

Instance Number	Instance Matrix Size	Number of Eliminations		
		Original Matrix	After Cuthill-McKee Ordering	After Reverse Cuthill-McKee Ordering
1	3x3	8	8	0
2	5x5	30	30	18
3	7x7	98	94	63
4	9x9	180	170	118
5	11x11	324	312	238
6	13x13	387	384	310
7	15x15	457	368	312
8	17x17	1131	941	889
9	19x19	1714	1597	1402
10	21x21	2459	2374	1935

Table 4.1 Results of benchmark runs of Gaussian Elimination with various input matrices.

Figure 4.1 is the graphical representation of the above test results.

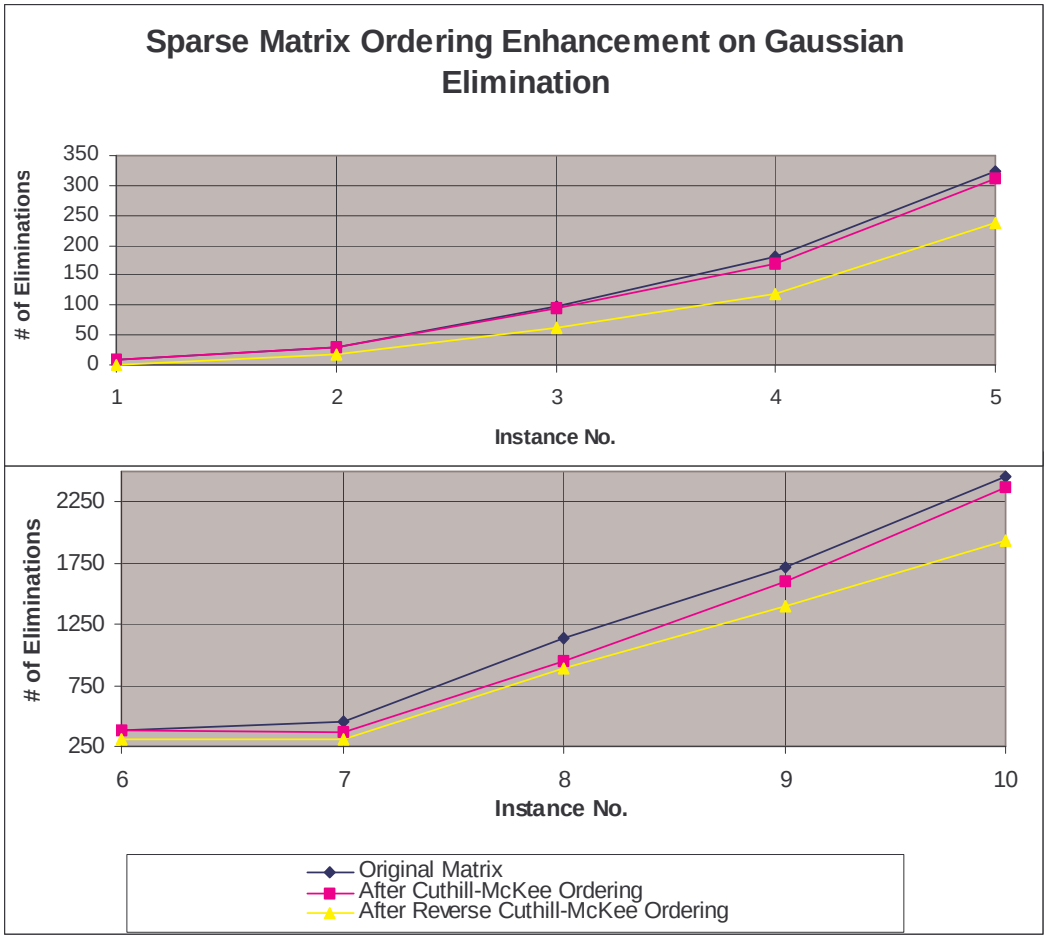

Figure 4.1 Charts generated for benchmark results.

Another benchmark I performed, which is a sort of derivation of the first one, is related to timing. The same instances fed to the solver before and after reordering and were solved 100K times and the elapsed time measured. There are two reasons for the selection of a large number such as 100K: first, some applications such as CAD design and analysis tools that deal with very large circuits (including millions of transistors) solves very large number of matrix equations; second, the time observed starts to be significant with such large number of computations. Table 4.2 gives these timing results. One good example from the table is with 15x15 matrix in which the time saved by reverse Cuthill-McKee Ordering is about 16% (~0.9 sec).

Instance Number	Instance Matrix Size	Time elapsed for 100K runs		
		Original Matrix	After Cuthill-McKee Ordering	After Reverse Cuthill-McKee Ordering
1	3x3	0.39	0.388	0.365
2	5x5	0.798	0.762	0.778
3	7x7	1.557	1.438	1.36
4	9x9	2.38	2.204	2.102
5	11x11	3.495	3.275	3.253
6	13x13	4.341	4.236	4.238
7	15x15	5.376	5.095	4.491
8	17x17	9.167	8.729	8.314
9	19x19	13.072	12.893	11.853
10	21x21	17.715	17.827	17.003

Table 4.2 Gaussian Elimination Timing analysis.

5. Conclusion

In this paper, I have shown how reordering operation over a sparse matrix can save time and space with linear system solving techniques such as Gaussian Elimination. Different type of solvers requires different matrix orderings for optimum performance; therefore, the selection of the ordering algorithm should be made accordingly. Independent set ordering algorithms and greedy multi-coloring algorithm [1] are some of the other ordering techniques that are widely used.

References:

[1] Saad, Yousef, *Iterative Methods for Sparse Linear Systems*, 2nd Edition, WPS, 2000

[2] Siek, Jeremy, *The Boost Graph Library (BGL) (Cuthill-McKee Ordering),* Addison Wesley, 2001

[3] Weisstein, Eric W., *Gaussian Elimination*, CRC Press LLC, 1999

[4] Bourke, Paul, *Solving Simultaneous Equations – Gaussian Elimination*, http://astronomy.swin.edu.au/~pbourke/analysis/gausselim/, 1997

[5] Khamsi, M.A., *Systems of Linear Equations: Gaussian Elimination*, http://www.sosmath.com/matrix/system1/system1.html

[6] Watkins David, *Fundamentals of Matrix Computations*, John Wiley & Sons, Second Edition (2002)