Smoothed Analysis Applied to Gaussian Elimination

Analyzing a Simple Algorithm Without Pivoting

Robert Moseley

Gordon and Jill Bourns College of Engineering, California Baptist University, Riverside, California, United States of America, robertwilliam.moseley@calbaptist.edu

ABSTRACT

Many current methods of evaluating Gaussian Elimination algorithms do not provide an accurate picture of in-practice time-complexity because they are evaluated based on their worst-case complexity. This paper will demonstrate the use of a technique known as smoothed analysis as applied to a common algorithm for computing Gaussian Elimination. One common algorithm will be shown, followed by an analysis of the strengths and weaknesses of that algorithm. All coding implementations have been done in Java for simplicity and clarity of algorithms. By applying the techniques of smoothed analysis to this algorithm used to compute Gaussian Elimination it can be demonstrated that 1) worst-case analysis of Gaussian Elimination does not yield results that reflect the time-complexity of the algorithm in practice, and 2) simple improvements to an existing algorithm can have a positive effect on overall time-complexity. Using these improvements, existing algorithms can be modified to improve performance of systems already using Gaussian Elimination techniques and methods. Additionally, the smoothed analysis of the algorithm can demonstrate that algorithms should not be judged on their appearance alone but should be given the opportunity to demonstrate their true properties in practice.

KEYWORDS

Gaussian Elimination, smoothed analysis, time-complexity, worst-case analysis, Cuthill McKee

1. Introduction:

There are several critical elements that need to be fully understood in order to perform a smoothed analysis of Gaussian Elimination. The first, and most important topic, is what Gaussian Elimination is, how it is calculated by hand, and what is it used for. Following an explanation of Gaussian Elimination and how it is computed, the theory and principles behind smoothed analysis also need to be fully understood. By combining these two topics we can gain a better understanding of how Gaussian Elimination works, and the various inefficiencies or problems that manifest themselves throughout the process. While current research on the topic of smoothed analysis, especially that of smoothed analysis applied to Gaussian Elimination, is limited in nature, enough research exists to form a solid foundation for the main points of this paper.

* 1. Gaussian Elimination

Gaussian Elimination, also known as row reduction, is a method in linear algebra that is used to solve systems of linear equations by performing a series of computations on a matrix of coefficients. These computations are performed until the lower left-hand corner has been populated with as many 0’s as possible. At this point the system will either have been solved, or it will be only a few steps away from being solved. There are three types of operations that are performed on the matrix to attain the desired result: 1) swap the position of two rows, 2) multiply a row by a non-zero scalar, and 3) add to one row a scalar multiple of another. When the matrix corresponds to a real system of equations, performing these row operations does not change the outcome or solutions that are found by performing Gaussian Elimination.

To begin explaining Gaussian Elimination we first need to start with a very simple 3-variable system of equations. This is given as follows:

While this is a very simple system, it is necessary to begin with a simple system to illustrate the process by which Gaussian Elimination can be performed. The above system can be translated into a matrix as shown below:

The first step of Gaussian Elimination is to take the above matrix and put it into row echelon form by continually performing row operations until the desired form has been reached. The following matrix is an example of a matrix in row echelon form:

One the matrix is in this form, it is a relatively quick process to perform back substitution and attain the final values for x, y, and z (x = 1, y =2, and z = 3 in the example equation above).

For more complicated systems, Gaussian Elimination is a straightforward, though arithmetically heavy, way to solve them and attain final coefficients for each variable. When systems start to become truly complicated in nature, or much larger, they become computationally expensive because of the number of operations required to solve them. This is why it is necessary to identify efficient algorithms for completing the process of Gaussian Elimination.

* 1. Smoothed Analysis

Smoothed analysis is a relatively new technique in computer science that has only been around for the past two decades. The purpose of smoothed analysis is to provide a more accurate look at algorithms that would normally be written off as being too slow to be useful. Smoothed analysis is a hybrid of average-case analysis and worst-case analysis that uses some of the advantages of both methods to provide a clear picture of an algorithm’s actual performance. Many algorithms that have polynomial, exponential, or factorial worst-case time-complexity are given early death sentences because they appear to be a poor choice of algorithm. In reality, many algorithms that are expected to perform poorly do not actually do so in practice. When using smoothed analysis, one of the key components to obtaining accurate results is to use random perturbations in algorithm inputs that are within a specific range of inputs that are expected to cause worst-case results or produce worst-case time complexity.

In addition to the random perturbations in algorithm inputs that are necessary to account for while performing a smoothed analysis, in the case of Gaussian Elimination it is also important to account for pivoting techniques that the algorithm takes advantage of. In the case of Gaussian Elimination, pivoting simply refers to the technique used to solve the system. With partial pivoting, an algorithm can modify the position of rows in the matrix to allow the system to be solved more efficiently. With full pivoting the algorithm can swap both rows and columns in a given matrix. To maintain the simplicity of the algorithm in section 2 of this paper, no pivoting will be used for solving.

* 1. Overview of Current Research

Performing a worst-case analysis of Gaussian Elimination typically yields less than desirable results–especially when evaluating the current selection of applied algorithms [5]. Though the worst-case analysis suggests incredibly poor time complexity, it is necessary to remember that this may not indicate actual in-practice time complexity. When using smoothed analysis, which is a hybrid between worst and average-case analysis, it is quickly revealed that the performance of some algorithms for solving linear systems with Gaussian Elimination are not as bad as they first appear [1]. Additionally, it is commonly thought that most algorithms, in a more general sense, have better performance than can actually be proved because of the lack of clear definitions in regard to how the algorithms are used and the inputs they are fed [6].

It is important that smoothed analysis is applied to Gaussian Elimination because, on paper, most algorithms that have been applied to Gaussian Elimination end up having either polynomial or exponential worst-case analysis–which is not an optimal result [5]. Especially in algorithms which do not use pivoting–when the first equation and first variable are chosen to be eliminated first­–it can be demonstrated that computational performance is not what the worst-case analysis would suggest [1]. It is worth noting, however, that “Gaussian Elimination with partial pivoting is unstable in the worst case: the ‘growth factor’ can be as large as , where n is the matrix dimension, resulting in a loss of bits of precisions” which is not an ideal result when efficiency and accuracy are the most considerable factors in a result [4]. Consequently, even though the algorithms demonstrated in this paper do not utilize pivoting, it is necessary to remember the drawbacks when attempting to scale results to different algorithms.

Smoothed analyses that have been performed already have shown smaller growth factors and a smaller chance of large condition numbers than expected on Gaussian Elimination without pivoting [1]. Traditionally, it is normal to use worst-case analysis because it is important to identify how poorly an algorithm might perform under specific circumstances, but for some algorithms it is rare for them to actually perform at their worst [9]. Smoothed analysis shows that in practice they rarely venture as far as their worst-case, with many algorithms even performing far better than anticipated [6]. Most current research on the topic suggests that this is also the case for Gaussian Elimination. While there are not a wide range of published algorithms on the subject, this paper will demonstrate 2 algorithms for Gaussian Elimination and show how they can be improved. After an algorithm has been identified, both worst-case and smoothed analysis will be applied to potentially demonstrate unique characteristics of the new algorithms and improved runtime capabilities.

* 1. Applications of Gaussian Elimination Algorithms

There are many important applications of Gaussian Elimination in computer science. While some uses are more on the theoretical side of computer science such as in machine learning, some algorithms have already been implemented and are an important component of existing infrastructures. One important application is seen in mesh-connected processors which can effectively utilize Gaussian algorithms to solve systems of linear equations efficiently [2]. In this scenario one of the processing structures is used for back substitution, while the other is used for the triangulation of the given matrix. Additionally, using improved Gaussian Elimination models can allow parallel solutions to systems to be more easily found [2]. While the applications for Gaussian Elimination algorithms are highly specified, they can be quite valuable in terms of reduced computational cost and time. Most Gaussian algorithms require a very large number of iterations to complete, which makes them very arithmetically costly for computers to regularly use. Despite Gaussian Elimination being computationally intensive, it is better than some of the alternatives that exist. One example of this is Gauss-Jordan Elimination, for which regular, Gaussian Elimination can be a much better alternative in terms of computational expense [2]. Essentially, any time it is necessary to compute a system of linear equations, Gaussian Elimination can be considered a potential solution. While Gaussian Elimination is not optimal for all systems, it can be modified to become more stable for certain types of systems, especially if pivoting is utilized. When pivoting is implemented as a component of the given algorithm, it becomes increasingly important to account for the drops in stability that can occur from the chosen pivoting method.

1. Analysis of A Gaussian Algorithm:

To begin the analysis of Gaussian Elimination algorithms, it is necessary to begin with an existing algorithm and analyze it for weaknesses or areas that it can be improved. Once a baseline algorithm is established, only then can an improved algorithm be constructed. In order to improve an algorithm, however, structural weaknesses must be found in that algorithm that can exemplify the necessity for improvement. Many Gaussian Elimination algorithms have poor worst-case analysis, which makes it easy to determine where the weaknesses are, but not necessarily easy to improve.

* 1. Algorithm 1: A Simple look at Gaussian Elimination

This algorithm comes from the book A First Course in Numerical Methods [10] and serves as an excellent starting point for the discussion of Gaussian Elimination algorithms. This algorithm is broken into two parts: 1) transforming the given matrix into upper triangular form, and 2) performing back substitution on the transformed matrix to complete the Gaussian Elimination. The algorithm is given below:

ALGORITHM 1: *A First Course in Numerical Methods* Algorithm

Transform into upper echelon form:

for k = 1 : n – 1

for i = k + 1 : n

for j = k + 1 : n

end

end

end

Back Substitution:

for k = n: -1:1

x(k) = b(k);

for j = k + 1 : n

x(k) = x(k) – A(k, j)\*x(j);

end

x(k) = x(k) / A(k, k);

end

As can be seen from the above algorithm, nothing overly complicated is going on throughout this process of Gaussian Elimination. The algorithm is essentially just looping through the indices of the matrix and slowly performing row operations until it is in upper triangular form. Once upper triangular form is achieved, the algorithm then performs back substitution by using the upper triangular form which gives the final answers to the system being solved.

To assess the operational cost of these algorithms it is necessary to approach them mathematically. Beginning with back substitution we attain operational values from the following equations [10]:

This equation is achieved by reducing the above algorithm for back substitution to its essential components [10]. It can also be achieved by evaluating the following representation of the back-substitution algorithm:

for k = n: -1:1

From these calculations it is shown that back substitution has a worst-case time complexity of [10], which is already not optimal. In addition to the time complexity for back substitution, however, it is also necessary to factor in the time complexity of the first part of the algorithm, which is putting the matrix into upper triangular form. The time complexity for upper triangular form is given below:

Given these solutions, the overall time complexity for this algorithm is which, for matrices of any large size, is a very poor time complexity. It will be shown later on using smoothed analysis that this case is not as bad as it appears, but for now this will be a starting place for creating an algorithm with an improved worst-case analysis and, hopefully, improved average-case analysis. Using Java this algorithm has been simply implemented, and the results of its implementation will be demonstrated in part 3 of this paper. The implementation is given below with each part of the above pseudo-code clearly labeled for ease of understanding:

public class StandardGaussianElimination {  
 */\*\*  
 \* Performs Gaussian Elimination on Matrix A and returns the result  
 \*****@param*** *n represents the size of the matrix (n x n)  
 \*****@param*** *A The matrix that is being transformed into upper triangular form  
 \*****@param*** *b a vector of size n  
 \** ***@return*** *returns the matrix A, after Gaussian Elimination has been performed  
 \*/* public static int[][] gaussianElimination(int n, int[][] A, int[] b) {  
 int[][] triangularForm = *upperTriangularTransform*(n, A, b);  
 int[][] finalMatrix = *backSubstitution*(n, triangularForm, b);  
 return finalMatrix;  
 }  
 */\*\*  
 \* Puts matrix A into upper triangular form as the first step of  
 \* Gaussian Elimination  
 \** ***@param*** *n represents the size of the matrix (n x n)  
 \** ***@param*** *A The matrix that is being transformed into upper triangular form  
 \** ***@param*** *b a vector of size n  
 \** ***@return*** *returns the newly changed matrix, A, in its upper triangular form  
 \*/* public static int[][] upperTriangularTransform(int n, int[][] A, int[] b) {  
 for (int k = 1; k <= n - 1; k++) {  
 for (int i = k + 1; i <= n; i++) {  
 int l = A[i][k] / A[k][k];  
 for (int j = k + 1; j <= n; j++) {  
 A[i][j] = A[i][j] - l\*A[k][j];  
 }  
 b[i] = b[i] - l\*b[k];  
 }  
 }  
 return A;  
 }  
  
 */\*\*  
 \* Performs the back substitution to finish the Gaussian Elimination  
 \** ***@param*** *n represents the size of the matrix (n x n)  
 \** ***@param*** *A The matrix that is being back substituted  
 \** ***@param*** *b The vector of coefficients from system A  
 \** ***@return*** *returns the matrix A after completing the Gaussian Elimination  
 \*/* public static int[][] backSubstitution(int n, int A[][], int[] b) {  
 int x[] = new int[b.length];  
 for (int row = n; row >= n; row--) {  
 x[row] = b[row];  
 for (int col = row + 1; col <= n; col++) {  
 x[row] = x[row] - A[row][col]\*x[col];  
 }  
 x[row] = x[row] / A[row][row];  
 }  
 return A;  
 }  
}

* 1. Additional Analysis

Additional analysis of the above algorithm can provide useful insight into the inner workings of the algorithms and where its strengths and weaknesses manifest themselves.

2.2.1 Space Complexity. Given the less than ideal worst-case time complexity of the current algorithm, the best improvement that can be made is to reduce the time complexity so something a little better. Especially in the case of the first portion of this algorithm which transforms the matrix into upper echelon form, is a truly awful time complexity and should be improved at any cost. One redeeming facet of the current algorithm is that it has constant, , space complexity which is something that would be beneficial to maintain in any new algorithms created to solve Gaussian Elimination. The current algorithm does not construct any new data structures of any kind. Instead, the existing data structure–a 2-dimensional array–is iterated over repeatedly until the desired result is attained, which is how constant space complexity is maintained throughout execution. While the space complexity of the algorithm is generally loosely connected to the time-complexity of the algorithm, it is necessary to student all facets of the algorithm to determine areas where the algorithm is failing, and how best to approach utilizing the algorithm.

2.2.2 Time Complexity. As stated previously, the overall worst-case time complexity of this algorithm is very poor. Any student of the field would recognize the performance of this algorithm as potentially problematic in scenarios with even slightly large inputs. Given inputs of a relatively small size, this algorithm will perform reasonably well, simply because the inputs are small, and computers can compute thousands of iterations very quickly without much trouble. Performance can become a significant problem, as stated above, when the inputs become larger. When it becomes necessary to perform millions, or billions of calculations in order to solve a problem, the true nature of the algorithm will become readily apparent. Additionally, not only does this algorithm utilize one exponential worst-case algorithm (), it actually uses two where the first is found in transforming the matrix into upper echelon form, and the second is found while performing back substitution. Clearly, then, even if the algorithm has good performance in practice, it would be good to guard against situations where the inputs become large. One solution to this would be to modify the algorithm in some way so that it could perform the necessary calculations in either polynomial or linear time, while still maintaining its overall stability and accuracy. Another solution might be to provide constraints to the algorithm in such a way that unnecessarily large inputs are ignored because of the immense amount of time required to solve them.

2.2.3 Arrays. The final component of this algorithm worth discussing is the data structure that it utilizes. Depending on the language and implementation chosen, the data structure used might be different, but in Java arrays provide an efficient way of performing many row operations. In addition, because access time of arrays is so quick in Java–­–arrays are an excellent choice for performing this amount of operations on a 2-dimensional data structure. For this algorithm, specifically, there is probably not a better structure to use, especially in Java. For implementation in other languages, it would be good to identify an equivalent data structure and then implement it as required by the algorithm. Python and C++, for example, both contain strong 2-dimensional array data structures, which could be a solid solution for implementing this algorithm in those respective languages.

1. Smoothed Analysis and Results:

While the above analysis and implementation clearly demonstrates the potential weaknesses of the algorithm, to truly understand the performance of the algorithm it is necessary to utilize the techniques of smoothed analysis. The results of the smoothed analysis will demonstrate the actual capabilities of the algorithm and whether or not it has practical uses. Worst-case analysis suggests that this algorithm’s performance could be very poor given worst-case inputs, but if we supply the algorithm with average-case inputs with random perturbations, it will provide a better indication how the algorithm can actually be expected to perform in practice. Combining techniques from both average-case analysis and worst-case analysis will provide a clearer picture of the true performance that can be expected from the algorithm. The smoothed analysis of this algorithm is demonstrated below, with results and additional analysis following.

* 1. Smoothed Analysis

In a typical worst-case analysis of or ), it would be normal to associate these run times with the following graph:

This graph clearly demonstrates the explosive growth properties of exponential time complexities, and the order of magnitude that they are capable of achieving with small inputs in good conditions. The above graph simply shows the differences between all 3 calculations. This, essentially, is why smooth analysis is necessitated by some algorithms. Many algorithms might have an important use case, but because of the large time complexity they are simply not implemented. With smoothed analysis we can better determine the use cases of an algorithm by testing it with random inputs that can demonstrate practical scenarios in which the algorithm could be utilized. Referring back to the original equation for calculating the time complexity of this algorithm we have:

Using this equation we can simply calculate the number of steps per matrix size ‘n’, which can provide useful information about the number of iterations required to solve a matrix of size “n x n”. To perform an accurate smoothed analysis, however, all of the inputs to the matrix need to be varied. One method for performing such a smoothed analysis involves implementing the Reverse Cuthill McKee algorithm, which is an algorithm that is used for reordering matrices that are symmetric and square. The Reverse Cuthill McKee algorithm is given as follows [11]:

ALGORITHM 2: *Reverse Cuthill McKee [11]*

vector<int> cuthill = new CuthillMckee

int n = cuthill.size

if n % 2 = 0

n = n – 1

n = n / 2

for i = 0; i < = n; i++

int j = cuthill[cuthill.size – 1 - i]

cuthill[cuthill.size – 1 – i] = cuthill[i]

cuthill[i] = j;

return cuthill

The first part of this algorithm instantiates a new Cuthill McKee object / structure, followed by the part of the algorithm that actually performs the reverse Cuthill McKee operations on the given structure. The code that implements this algorithm in Java is given below. This algorithm is adapted from an existing C++ algorithm [11].

public class CuthillMckee {  
 List<Double> list = new ArrayList<Double>();  
  
 public int findIndex(int[] a, int x) {  
  
 for (int i = 0; i < a.length; i++) {  
 if (a[i] == x) {  
 return i;  
 }  
 }  
 return -1;  
 }  
  
 private class reordering{  
  
 private double[][] matrix;  
  
 reordering(double[][] m) {  
 this.matrix = m;  
 }  
  
 public double[] degreeGenerator() {  
 double[] degrees = new double[matrix.length];  
  
 for (int i = 0; i < matrix.length; i++) {  
 double count = 0;  
  
 for (int j = 0; j < matrix[0].length; j++) {  
 count += matrix[i][j];  
 }  
 degrees[i] = count;  
 }  
 return degrees;  
 }  
  
 public List<Integer> CuthillMcKee() {  
 double[] degrees = degreeGenerator();  
  
 Queue<Integer> q = new LinkedList<Integer>();  
 List<Integer> R = new ArrayList<Integer>();  
 Map<Integer, Double> map = new HashMap<Integer, Double>();  
  
 for (int i = 0; i < degrees.length; i++) { map.put(i, degrees[i]); }  
  
 while (map.size() > 0) {  
 int minNodeIndex = 0;  
 for (int j = 0; j < map.size(); j++) {  
 if (map.get(j) < map.get(minNodeIndex)) {  
 minNodeIndex = j;  
 }  
 }  
 ((LinkedList<Integer>) q).push(minNodeIndex);  
 map.put(0, 0.0);  
 while (!q.isEmpty()) {  
 List<Integer> toSort = new ArrayList<Integer>();  
  
 for (int i = 0; i < matrix[0].length; i++) {  
 if (i != q.peek() && matrix[q.peek()][i] == 1) {  
 toSort.add(i);  
 map.put(i, 0.0);  
 }  
 }  
 Collections.*sort*(toSort);  
 for (int i = 0; i < toSort.size(); i++) {  
 ((LinkedList<Integer>) q).push(toSort.get(i));  
 }  
 R.add(q.remove());  
 }  
 }  
 return R;  
 }  
  
 public List<Integer> reverseCuthill() {  
 List<Integer> cuthill = new ArrayList<Integer>(CuthillMcKee());  
 int n = cuthill.size();  
 if (n % 2 == 0) { n -= 1; }  
 n /= 2;  
 for (int i = 0; i <= n; i++) {  
 int j = cuthill.get(cuthill.size() - 1 - i);  
 int temp = cuthill.get(cuthill.size() - 1 - i);  
 cuthill.add(i, temp);  
 cuthill.add(j, i);  
 }  
 return cuthill;  
 }  
 }  
}

Using all of the above algorithms and an analysis of Gaussian Elimination done at MIT [12], our final representation of the complexity of Gaussian Elimination can be given by the following equations sourced from the same analysis at MIT [12]:

Combining these equations with the Cuthill McKee algorithm we can obtain an interesting result from this analysis: a smoothed analysis that utilizes random inputs will perform better no matter the size of the inputs that we use. Larger inputs produce, as expected, larger time complexities, but by using random inputs the picture is different than it would be otherwise. Random inputs show that, while the algorithms certainly can require or produce extremely large run times, they do not always perform in this way. Additionally, when sparse matrices are used, there is an additional gain in performance over other matrices [12]. This is important because it indicates proper usage of matrices with these types of algorithms and the performance that can be anticipated when they are used. For the above algorithms, it is recommendable to not use extremely large inputs because they can be very inefficient to calculate, but for smaller inputs sizes the run time is not as extreme as it might first appear.

* 1. Conclusion

Smoothed analysis can be an incredibly powerful tool for gaining a better understanding of algorithms and how they perform in practice. Additionally, smoothed analysis can provide a means to properly judge an algorithm and its proper use cases. Relying solely on worst-case analysis, or average-case analysis, is not enough to understand the practical applications of complicated algorithms–especially algorithms that can lead people to a false understanding of their in-practice performance. From the above analysis it can be inferred that for Gaussian Elimination, specifically, the use of random inputs can help demonstrate the actual performance of Gaussian Elimination algorithms. The Cuthill McKee algorithm, in particular, benefits massively from sparse matrices as inputs. Moving forward, smoothed analysis should be a tool that is used to analyze algorithms and help researchers understand why algorithms behave as they do. Not all algorithms perform as poorly as they might first appear.

ACKNOWLEDGMENTS

I would like to thank Professor Benjamin Knisley at California Baptist University for encouraging me to pursue this topic for my paper. I still have a lot to learn, but Professor Knisley encouraging me to take on challenging topics and ask hard questions has been a huge blessing to me and my knowledge of computer science.

REFERENCES

[1] Arvind Sankar. 2004. Smoothed Analysis of Gaussian Elimination. Ph.D. Dissertation. Massachusetts Institute of Technology, Boston, Massachusetts.

[2] Maria Saeed, Sheza Nisar, Sundas Razzaq, Rabea Masood and Rabiya Imran. 2015. Gaussian Elimination Method–A Study of Applications. Global Journal of Science Frontier Research: For Mathematics and Decision Sciences 15, 5 (2015), 1-7.

[3] Jim Lambers. 2009. Gaussian Elimination and Back Substitution. University of Southern Mississippi, Hattiesburg, Mississippi.

[4] Lloyd N. Trefethen and Robert S. Schreiber. Average-case Stability of Gaussian Elimination. Society for Industrial and Applied Mathematics 11(3), (23 May 1988) 335-360. DOI: <https://doi.org/10.1137/0611023>

[5] Volker Strassen. 1969. Gaussian Elimination is Not Optimal. Numerische Methematik 13, (1969) 354-356. DOI: <https://doi.org/10.1007/BF02165411>

[6] Daniel A. Spielman and Shang-Hua Teng. (2003). Smoothed Analysis of Termination of Linear Programming Algorithms. Mathematical Programming 97, (July 2003).

[7] Daniel A. Spielman and Shang-Hua Teng. (2008). Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time. Massachusetts Institute of Technology, Boston, Massachusetts.

[8] Arvind Sankar and Daniel A. Spielman. (2005). Smoothed Analysis of the Condition Numbers and Growth Factors of Matrices. Massachusetts Institute of Technology, Boston, Massachusetts (December 8, 2005).

[9] Daniel A. Spielman and Shang-Hua Teng. (2009). Smoothed Analysis: An Attempt to Explain the Behavior of Algorithms in Practice. *Communcations of the ACM* 52, 10 (October 2009) 76-84. DOI: <https://doi.org/10.1145/1562764.1562785>

[10] Uri M. Ascher and Chen Greif. (2011). *A First Course in Numerical Methods*. Society for Industrial and Applied Mathematics, The University of British Columbia, Vancouver, Canada.

[11] Ronak Doshi. *Reverse Cuthill McKee Algorithm*. GeeksforGeeks: A Computer Science Portal for Geeks. <https://www.geeksforgeeks.org/reverse-cuthill-mckee-algorithm/>

[12] Dan Spielman. (2002). The Behavior of Algorithms in Practice: Smoothed Analysis of Gaussian Elimination. Massachusetts Institute of Technology, Boston, Massachusetts.

<https://ocw.mit.edu/courses/mathematics/18-409-behavior-of-algorithms-spring-2002/lecture-notes/lect5.pdf>