Journal of the Serbian Society for Computational Mechanics / Vol. 15 / No. 1, 2021 / pp 81-97
(10.24874/jsscm.2021.15.01.06)

IMPROVEMENT OF THE SPARSE MATRICES STORAGE ROUTINES FOR LARGE FEM CALCULATIONS

Dragoljub Stevanović[1], Marko Topalović[2*], Miroslav Živković[1]

[1] Faculty of Engineering, University of Kragujevac, Serbia
e-mail: dragoljub.stevanovic.kg@gmail.com, zile@kg.ac.rs
[2] Institute for Information Technologies, University of Kragujevac, Serbia
e-mail: topalovic@kg.ac.rs
* *corresponding author*

Abstract

Efficient memory handling is one of the key issues that engineers and programmers face in developing software for numerical analysis such as the Finite Element Method. This method operates on huge matrices that have a large number of zero coefficients which waste memory, so it is necessary to save it and to work only with non-zero coefficients using so called "SPARSE" matrices. Analysis of two methods used for the improvement of "SPARSE" matrix creation is presented in this paper and their pseudo code is given. Comparison is made on a wide range of problem sizes. Results show that "indexing" method is superior to "dotting" method both in memory usage and in elapsed time.

Keywords: Memory usage optimization, Matrix handling, SPARSE matrix, Matrix notation, Finite Element Method

1. Introduction

The Numerical analysis of the ever increasing and more complex problems presents challenges for engineers and software developers primarily in the field of optimal utilization of the large number of processors and storage of required data in the memory. Increasing the size of the model leads to an exponential increase of the required memory, sometimes even beyond the available amount on brand new machines. Therefore, it is necessary to optimally store only the necessary data. This problem will be addressed in this paper. Solving the system of equations (for any kind of the problem) can be written in the matrix form $\mathbf{A}\cdot\mathbf{x}=\mathbf{r}$, where $\mathbf{A}$ represents the coefficient matrix, vector $\mathbf{r}$ is a vector of solutions and $\mathbf{x}$ is a vector of unknown values.

In real life problems there are a large number of coefficients that are equal to zero that unnecessarily occupy memory position according to Gilbert et al. (1992), and therefore engineers and programmers sought ways to avoid the storage of these coefficients using various approaches (Amestoy et al. 2002), (Armstrong et al. 2010) and (Mofrad et al. 2013). In this paper, we will focus on one of the most popular numerical methods - Finite Element Method (FEM), described in detail by Bathe (2007), and ways to improve its matrix handling in terms of memory requirements and computer processing time. The main motive to investigate this issue, was an analysis of dams, which required a large number of elements to accurately represent dam as well as surrounding area which resulted in a model which was too big to handle with the Finite Element solver. In this paper, we show two approaches that we tried out in FEM solver in order to make a large dam model more memory efficient, so it could be analyzed using the only computer RAM memory (64 GB) without a need for additional memory allocation on hard drive which would greatly increase analysis time.

According to Bathe (2007), in the FEM coefficient matrix $\mathbf{A}$ represents the relation between stress and strain and is called "stiffness matrix". This term is much older than FEM itself, with first mentioning in two papers by Duncan and Collar (1934), (1935) regarding a method called Matrix Structural Analysis (MSA), which, according to Felippa (2000) is a discrete, analog direct ancestor of FEM. This method was used in aero-elasticity, as the airplane maximum speeds increased, so did the wing flutter (Felippa 2000). FEM Analysis is characterized by problems which have symmetrical model, and very often symmetrical stiffness matrix. For symmetric stiffness matrices following statement is true $a_{ij} = a_{ji}, \forall i, j$. On the other hand, symmetric models are defined as those systems in which, if any element B affects element C then at the same time element C affects element B. Effect of B to C and effect of C to B can be with the same or different intensity $a_{ij} = a_{ji} \vee a_{ij} \neq a_{ji}$. According to Felippa (2000), aero-elastic analysis stiffness matrices are generally unsymmetrical, being the sum of a symmetric elastic stiffness and an unsymmetrical aerodynamic stiffness.

In the FEM evolution, according to Felippa (2000), the next big step is a Direct Stiffness Method introduced by Turner (1959) featuring the assembly procedure, in which the stiffness matrix for the whole model is generated by direct addition of element matrices.

Considering the symmetric matrix with large number of zero coefficients, there are several ways to save memory, out of which SPARSE matrices are the most effective and commonly implemented solution, which were developed by Wilson (1963) for his PhD. dissertation.

Algorithms and matrix operations developed for regular, dense matrices are slow and inefficient when applied to large SPARSE matrices, because a lot of processing time and memory are wasted on the zero elements; hence, over the years, specialized solvers/libraries were developed for these problems. In our PAK program for FEM analysis (Živković 2005), we use MUltifrontal Massively Parallel sparse direct Solver (MUMPS) which, according to Amestoy et al. (2001), represents the state of the art solution for solving SPARSE matrix problems. Also, according to Amestoy et al. (2003) using non-blocking communication primitives improves the performance and robustness in comparison to the simple Message Passing Interface (MPI) point-to-point communication primitives. MUMPS requires information where the non-zero members are located, a task that can be accomplished in several ways.

Traditional SPARSE matrix storage formats are: Coordinate Format (COO), Compressed Sparse Column (CSC), Compressed Sparse Row (CSR) and Blocked Compressed Sparse Row (BCSR). Armstrong et al. (2010) used these formats, and focused their research on development of algorithm for optimum format selection based on runtime parameters. More recently, Borštnik et al. (2014) developed Distributed Block-Compressed SPARSE row library.

Over the years, there were numerous approaches to the optimization of SPARSE matrix assembly and solving. For example, Gilbert et al. (1992) used pattern recognition for identification of recurring blocks of non-zero elements, while Romero and Zapata (1995) used storage by row of blocks implemented in their sparse matrix vector multiplication solution for distributed memory multiprocessors. Kaveh and Ghaderi (1997) dealt with ill-conditioned stiffness matrices that had large off-diagonal entries, which increased sparsity of the columns, while Adle et al. (2005) analyzed automatic parallelization of sparse matrix computations. Pichel et al. (2012) evaluated some of the most successful reordering techniques and tested a number of sparse matrix storage formats. Mofrad et al. (2013) used bit flipping algorithm to solve sparse systems of linear equations. As the hardware progress shifted more to the field of graphics cards, researchers also begun focusing more on GPU solvers rather than the CPU, like Oyarzun et al. (2014) for instance, who researched MPI-CUDA sparse matrix–vector multiplication using hybrid parallelization strategy, or Ashari et al. (2015) who developed a novel model-driven blocking strategy for load balanced sparse matrix–vector multiplication on GPUs, that reduces thread divergence and improves the load balance. However, since the PAK program (Živković 2005) solves balance equations on the CPU, we will focus on the state of the art for the traditional CPU implementation of SPARSE matrix computation.

According to Hiemstra et al. (2019), traditional FEM solvers have element subroutines that generate the specific element matrices, that are afterwards assembled in the system stiffness matrix, but, Hiemstra et al. (2019) have also shown that row-by-row or column-by-column assembly of the stiffness matrix is far superior to the element by element implementation. However, Hiemstra et al. (2019) also concluded that these new approaches would require a lot of FEM subroutines to be completely rewritten and that many quantities need to be precomputed for the use in the sum factorization which is the crucial part of their newly proposed algorithms. Implementation and testing of these new algorithms in PAK solver may be the topic of our further research, and comparison/combination with our own solutions could lead to further improvements of matrix assembly subroutines.

In this paper, we present two improved variants of element by element matrix assembly, "dotting" and "indexing", which can be selected for any particular problem or any SPARSE storage format. These enhanced methods main feature is storing of Boolean True/False values within the signed or unsigned integers, thus greatly reducing required memory for the matrix assembly.

Both methods are described and compared and pseudo codes which demonstrate the functionality of these methods in finding row indexes are given. Methods do not depend on whether the model is symmetric or has symmetric matrices because the arrays at the same time indicate the column of the lower triangular matrix. Examples of the "dotting" and "indexing" procedures used on a showcase matrix are given in appendices A, B and C.

2. Methods

The issue of SPARSE matrix implementation in numerical analysis of the large problems that we investigated in this paper is a problem of choosing appropriate method for determining the row index of the members, which would not take a lot of memory space. All standard (formats) notifications according to Armstrong et al. (2010) use 3 arrays – row Rk, column Ck and value Vk, while some like Kojić et al. (1998) use help auxiliary array Hn for faster search of elements in those arrays. The length of three standard arrays is equal to the number of nonzero elements in the matrix, while the length of the helper auxiliary array is equal to size of matrix+1. An example of the regular full matrix and its equivalent SPARSE matrix representation is given in Appendix A. In the following sections two methods for the SPARSE matrix generation that we used are described in detail.

2.1 Dotting method

This method is based on a concept that we can know positions of non-zero elements in stiffness matrix even before that matrix is created. This is achieved by representing every matrix member with True or False, with the value being evaluated based on the fact that a particular member holds non-zero value or not. Because the stiffness matrix is very large, we used shortened equivalents - Upper Triangular Matrix with diagonal elements (UTM) or contour matrix (all elements from the diagonal element to furthest nonzero element in row/column). Because many programming languages (FORTRAN, for example) use bytes for storing Boolean True/False values, we can reduce the memory requirements of True/False matrix storage if we use integers instead of Boolean values. This way every integer holds the values for 7 or 8 Boolean values, depending on chosen signed or unsigned integer type. This procedure (using signed and unsigned integers) is shown in Fig. 1.

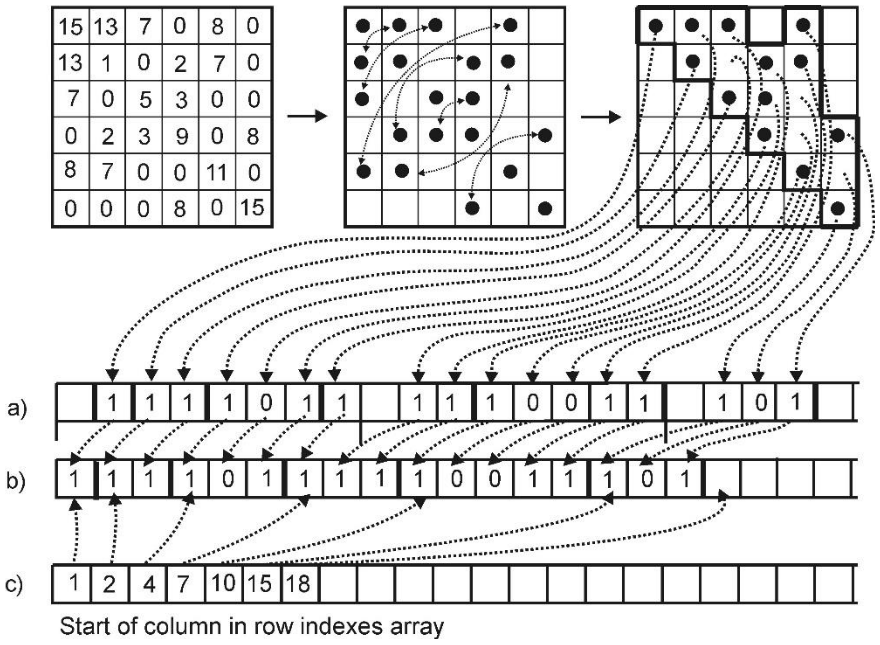

Fig. 1. Graphical representation of dotting process, for matrix given in Appendix A. (a) Storing True/False values in signed integers; (b) Storing True/False values in unsigned integers; (c) diagonal element indexes.

As can be seen from Fig. 1, Boolean arrays for signed (a), or unsigned integer type (b), are used to store information on whether the certain element of the matrix has a non-zero value or not. Array (c) contains the serial numbers of the elements that are the first in the appropriate column and which are the diagonal elements in UTM. Values in arrays (a) or (b) are saved and read using the Boolean algebra. At the beginning, all of the values in array (a) or (b) are 0. Then in order to enter the value TRUE (1) to the corresponding bit operation OR ($\vee$) is used on corresponding byte in (a) or (b) array and Boolean value of the position of the particular bit. For reading the value TRUE from the appropriate position in Boolean array operation AND ($\wedge$) is used in the same way as it is done for saving in the appropriate array position.

For the following pseudo code that describes the dotting method, a more detailed description of the process of forming Boolean (integer) array is given in Appendix B.

Dotting method pseudo code:

```
For k = 1 to number of elements
    Load equations array // element influence to equations
    For e = 1 to element size
        For m = e + 1 to element size
            I = equations(e) // column
            J = equations(m) // row
            If I >= J then // upper triangular matrix
                KK = c(I) + I - J
                Elem = (KK + D - 1) div D
                Pos = D - (Elem * D – KK)
                SET_DOT(Elem, Pos)
            End if
        Next m
    Next e
Next k
K = 0
E = 0
col = 1
S = 1
For I = 1 to size of TEMP array
    For J = 1 to D
        K = K+1
        If GET_DOT(I,J) = 1 or K = C(col) then
            E = E + 1
            If K = C(col) then
                Row(E) = col
                C(col) = E
                S = K
            Else
                Row(E) = col + S – K
            End if
        End if
    Next J
Next I
C(col+1) = E + 1
```

For example, the first bit in a byte is used to store information (in array (a) or (b) in Fig. 1.) if the corresponding element of the matrix is non-zero element using the number 128 in binary notation $(10000000)_{(2)}$ for (b) and 64 in binary notation $(01000000)_{(2)}$ for (a). For the second bit 64 is used for (b) and 32 in binary notation $(00100000)_{(2)}$ for (a) and so on. For 7th bit number 2 is used in binary notation $(00000010)_{(2)}$ for (b) or 1 in binary notation $(00000001)_{(2)}$ for array (a). For unsigned array (b) for 8th bit number 1 $(00000001)_{(2)}$ is used.

Dotting method requires the auxiliary array to be created with information, whether a particular place holds a non-zero element ((a) or (b) in Fig. 1). This is done in the first loop in pseudo code. The size of auxiliary array is determined, using the size of UTM or the size of contour matrix. In the case of UTM, it is relatively easy to determine the position of element $A_x = A_{i,j}$ when $x = i * (i - 1) + j$ is true. In the case of contour matrix, which is smaller than a UTM, there are some challenges in the determination of the correct position of element in that array, and because of these challenges we use a helper array of starting row/column index in auxiliary array. Also, there can be differences in size of contour matrix if elements of the matrix are packed row by row or column by column from stiffness matrix. Creation of a row index array from auxiliary array is shown in the second loop of pseudo code. Depending on which method is used in the subroutines for writing and reading value of 1 (logical true) into auxiliary array, significant difference in consumed time may occur. The row index array is formed by taking each member of the auxiliary array and calculating the row index if the member is equal to 1.

Element and a member of the auxiliary array are two different concepts. For example, if you access 893rd member, it means accessing 128th element and the 4th bit of that element (values of variables Elelm and Pos in pseudocode respectively). Or, in other words, 128th element carries information about the members 890, 891, 892, 893, 894, 895 and 896. After the auxiliary array is fully assigned, we can create an index array which is created dynamically by accessing each member of the auxiliary array continuously (Fig. 1).

2.1 Indexing method

This method as well as dotting method is based on a concept that we know in advance where the non-zero elements in the stiffness matrix are. Unlike dotting method that checks every position in the stiffness matrix, this method takes a different approach. First the possible number of non-zero members in each column/row is determined. Once this number is determined, this method goes through all the possible positions where non-zero elements are and places corresponding indexes into the auxiliary index array. Because the same non-zero member may appear multiple times for indexing, it means that auxiliary index array contains the number of extra indexes of non-existent members. In order to have a compact index array, the auxiliary index array is reduced by removing non-existent indexes. Since the non-existent members are located at the end of each block of the index in the same column/row this means that this array does not have to be destroyed, but it will become an array of row/column indexes. Hence, the total memory required for the determination of the index array is equal to the memory usage of the auxiliary index array.

Indexing method pseudo code:

```
For k=1 to number of equations + 1
C(k)=1
Next k
For k = 1 to number of elements
        Load equations array // element influence to equations
        For e = 1 to size of element
                I= equations(e) // column
                J= equations(k) // row
                If I>J then
                        C(I) = C(I)+1
                End if
        Next e
Next k
R(1) = 1
For k = 2 to number of equations + 1
        If C(k) > k then C(k) = k
        C(k) = C(k-1) + C(k)
        R(C(k)) = k
Next k
N = C(number of equations + 1) - 1
For k = 1 to number of elements
        Load equations array // element influence to equations
        For e = 1 to size of element
                I = equations(e) // column
                J = equations(k) // row
                If I>J then
                        P = C(I) + 1
                        While (R(P) <> J) and (R(P) > 0 ) P = P + 1
                        R(P) = J
                End if
        Next e

Next k
```

```
k = 2
While ((R(C(k+1)-1) > 0) and (k< number of equations)) do k = k + 1
p = C(k) + 1
while (R(p) > 0) do p = p + 1
while k < number of equations
        k = k + 1
        temp = p
        for i = C(k) to C(k+1) - 1 do
          if R(i) = 0 then exit loop
          R(p) = R(i)
          p = p + 1
        next i
        C(k) = temp
End while
```

Additional usage of memory by the auxiliary index array can be neglected for large examples, because the number of non-existing indexes will be less than the number of non-zero elements and hence the required memory to store the stiffness matrix will take the same amount of memory that is used for the storage of non-existing elements in the auxiliary index array. The indexing method, which is shown in the pseudo code, uses only addition and subtraction as logical operations in comparison to dotting method, which was discussed in the previous section.

The first loop counts diagonal elements of the matrix. In the second loop theoretical maximal number of non-zero elements above the main diagonal is determined. In the third loop the array that contains starting columns in row index array is created. N represents the number of non-zero elements in the matrix. In the fourth loop row index array is filled. After forth loop it is represented shortening of a row index array.

For the pseudo code that describes the indexing method, a more detailed description of the process of forming Boolean (integer) array is given in Appendix C.

3. Results and Discussion

Regardless of whether the stiffness matrix is symmetric or not, the amount of memory that is required for an array of row indexes is the same. On the other hand, memory requirements and time consumption for the search for the index array depend on the method used. In both methods, the three arrays (C, M, and R) are used.

Array R contains row indexes grouped by column, while array C contains starting indexes of the corresponding column blocks in array R. Array M contains stiffness matrix elements and depending on whether it is symmetrical or not its length is equal to the length of the array R or twice the length.

In the case when the stiffness matrix is non-symmetric, array M occupies twice the size of an array R because storage of another set of diagonal elements is redundant and hence the utilization of memory is reduced. This issue can be neglected because of the ease of access to members that are located below the main diagonal.

For each x belonging to $[C_i, C_{i+1}]$, it applies: $j = R_x$, $A_{i,j} = M_x$. In the case of the symmetrical model it applies $A_{j,i} = M_x$, while for the unsymmetrical model it applies $A_{j,i} = M_{x+N}$, where N is the total number of non-zero elements that are found in the stiffness matrix on and above the main diagonal. In order to have detailed, objective results to compare these two methods implemented in PAK solver over a wide range of model sizes, we created a simple 1m^3 cube that was constrained at the bottom and with prescribed displacement at the top. This cube is then divided into finite elements with increasing mesh density with number of elements along the edge ranging from 1 to 50 as it's shown in Table 1.

Model size	Number of equations	The number of "contour" elements	Number of non-zero elements in UTM	Number of elements in the auxiliary array needed for indexing	Number of SPARSE element in regards to the number of "contour" elements
Cube 1	12	78	78	78	100.000%
Cube 5	540	74,982	14,670	28,830	19.565%
Cube 10	3,630	2,669,717	120,390	254,310	4.509%
Cube 15	11,520	21,903,952	409,410	885,690	1.869%
Cube 20	26,460	96,760,437	973,980	2,132,220	1.007%
Cube 25	50,700	304,584,422	1,906,350	4,203,150	0.626%
Cube 30	86,490	774,583,657	3,298,770	7,307,730	0.426%
Cube 35	136,080	1,701,328,392	5,243,490	11,655,210	0.308%
Cube 40	201,720	3,358,251,377	7,832,760	17,454,840	0.233%
Cube 45	285,660	6,111,147,862	11,158,830	24,915,870	0.183%
Cube 50	390,150	10,431,675,597	15,313,950	34,247,550	0.147%

Table 1. Dependence between number of elements in stiffness matrix and number of elements in cube model.

As the size of the model increases, the number of contour elements also increases, as a power function with power in the range of 1.75 – 1.81. The number of non-zero elements also increases, but not as fast as a number of contour elements, so the share of non-zero elements in total number of contour elements decreases, which means that the matrix is getting more and more sparse, which is shown in Table 1.

The superiority of indexing method over dotting method is more and more evident as the size of the model increases, both in terms of the required memory and in terms of the elapsed time, which is shown in Table 2.

Model size	Required memory			Elapsed time [s]	
	Dotting	Indexing	Unit Size	Dotting	Indexing
Cube 1	0.37	0.36	kB	0.001	0.001
Cube 5	305.47	114.73	kB	0.005	0.009
Cube 10	10.56	0.98	MB	0.080	0.023
Cube 15	86.59	3.42	MB	0.415	0.080
Cube 20	382.40	8.23	MB	1.793	0.190
Cube 25	1,203.59	16.23	MB	5.586	0.372
Cube 30	3,060.66	28.21	MB	14.135	0.644
Cube 35	6,722.36	44.98	MB	39.954	1.007
Cube 40	13,269.01	67.35	MB	61.299	1.525
Cube 45	24,145.85	96.14	MB	111.377	2.168
Cube 50	41,216.38	132.13	MB	189.436	3.221

Table 2. Required memory and elapsed time for creating an index array.

Although the "dotting" method has only two steps, while "indexing" method has four, it can be seen that the "indexing" method is much faster, because in the "dotting" method we used multiplication, division and modulus which are the slowest basic mathematical operations that are executed by a computer, while "indexing" does not utilize these operations.

The advantages of "indexing" method can be clearly seen in Fig. 2. and 3. which show elapsed time and memory usage respectively.

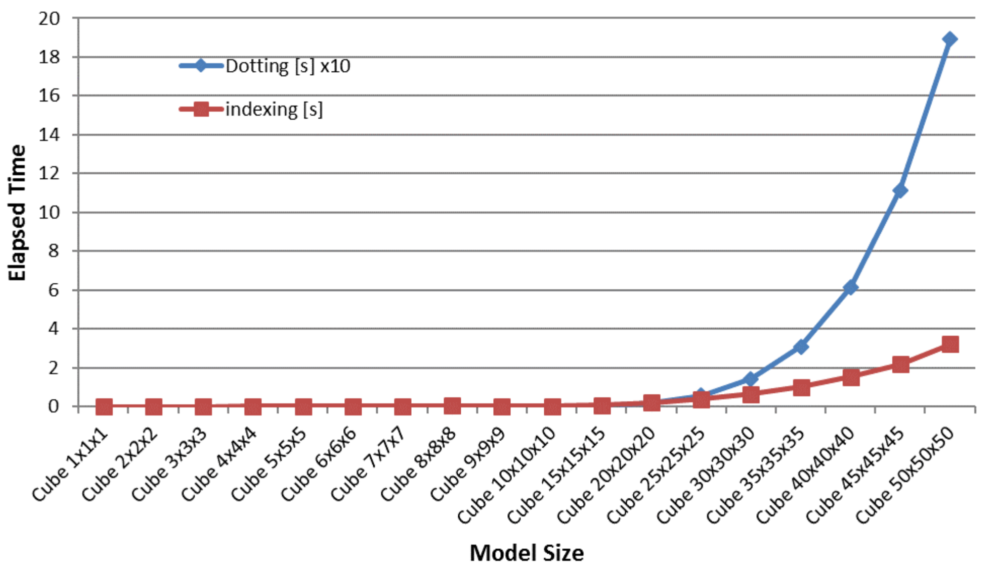

Fig. 2. Comparison of elapsed time of dotting and indexing methods.

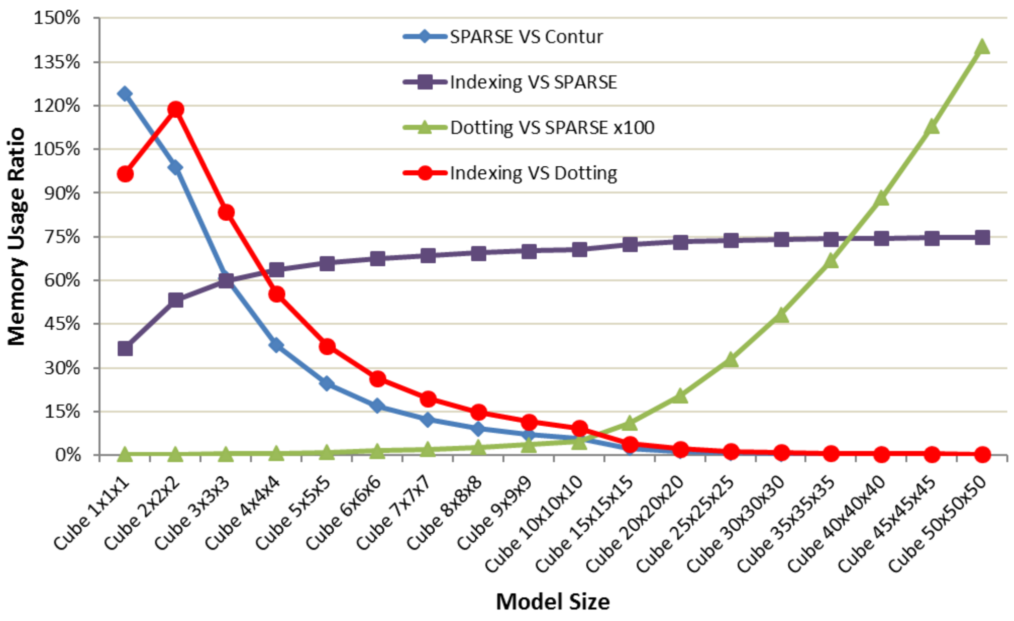

Fig. 3. Graphical Comparison of total memory required to store SPARSE matrix relative to the amount of memory required or the determination of the index array, for indexing and dotting methods.

4. Conclusions

Due to the large number of zero elements in the matrices in numerical methods such as the Finite Element Method, it is necessary to pay special attention to the processes of their storage and usage. The best solution is to save only non-zero elements in specialized SPARSE matrices using one of the several storage formats. This process must be efficient in terms of elapsed time and used memory. Two methods, "dotting" and "indexing" are explained and compared on a wide range of model sizes. The main idea behind "dotting" method was to save memory using bits (or so called dots) to mark positions of non-zero elements while determining arrays that store SPARSE matrix. This method needed to overcome imperfection of bit data saving in many programming languages using bytes which resulted in very inefficient memory usage. Another drawback of "dotting" method was the usage of multiplication, division and modulus which are the slowest basic mathematical operations.

The main disadvantage of the "indexing" method is that it has four steps, in comparison to the "dotting" method which has only two steps. But on the other hand, due to utilization of mentioned slow mathematical operations, "dotting" is inferior to "indexing" method regarding consumed time. As for the memory usage, "indexing" method also outperforms "dotting" method especially in large examples. The only advantage of "dotting" in terms of memory is with a model size 2 as it can be seen in Figure 2, but this can be neglected because of generally low memory usage.

This paper showed two diverse methods for SPARSE matrix creation and it showed the way to implement both methods. It is shown that indexing method performs better in large examples. This proposed approach can be very helpful in the implementation of software based on Finite Element Method, because it can help reduce the execution time and the utilized amount of memory. That way it would be possible to perform simulations with larger number of nodes and more complex geometries, that was either not possible or very time and memory consuming without the proposed approach.

Acknowledgements The part of this research is supported by the Ministry of Education and Science, Republic of Serbia, Grants TR32036 and 451-03-68/2020-14/200378.

References

Adle R, Aiguier M, Delaplace F (2005). Toward an automatic parallelization of sparse matrix computations, Journal of Parallel and Distributed Computing, 65, 313-330.

Amestoy P, Duff I, L'excellent J, Li X (2001). Analysis and comparison of two general sparse solvers for distributed memory computers, ACM Transactions on Mathematical Software (TOMS), 27, 388-421.

Amestoy P, Duff I, L’Excellent J, Li X (2003). Impact of the implementation of MPI point-to-point communications on the performance of two general sparse solvers, Parallel Computing, 29, 833-849.

Armstrong W, Rendell A (2010). Runtime sparse matrix format selection, Procedia Computer Science, 1, 135-144.

Ashari A, Sedaghati N, Eisenlohr J, P. Sadayappan P (2015). A model-driven blocking strategy for load balanced sparse matrix–vector multiplication on GPUs, Journal of Parallel and Distributed Computing, 76, 3-15.

Bathe K J (2007). Finite Element Procedures, Klaus-Jurgen Bathe

Borštnik U, Vondele J V, Weber V, Hutter J (2014). Sparse matrix multiplication: The distributed block-compressed sparse row library, Parallel Computing, 40, 47-58.

Duncan W, Collar A (1934). A method for the solution of oscillations problems by matrices, Philosophical Magazine, 7, 865.

Duncan W, Collar A (1935). Matrices applied to the motions of damped systems Philosophical Magazine, 7, 197.

Felippa C (2001). A historical outline of matrix structural analysis: a play in three acts, Computers & Structures, 79, 1313-1324.

Gilbert J, Moler C, Schreiber R (1992). Sparse matrices in matlab: design and implementation, SIAM Journal on Matrix Analysis and Applications, 13, 333-356.

Hiemstra R, Sangalli G, Tani M, Calabrò F, Hughes T (2019). Fast formation and assembly of finite element matrices with application to isogeometric linear elasticity, Computer Methods in Applied Mechanics and Engineering, 355, 234-260.

Kaveh A, Ghaderi I (1997). Conditioning of structural stiffness matrices, Computers & Structures, 63, 719-725.

Kojić M, Slavković R, Živković M, Grujović N (1998). Metod konačnih elemenata I (linearna analiza), Mašinski fakultet u Kragujevcu, Kragujevac, Serbia.

Mofrad A, Sadeghi M, Panario D (2013). Solving sparse linear systems of equations over finite fields using bit-flipping algorithm, Linear Algebra and its Applications, 439, 1818-1824.

Oyarzun G, Borrell R, Gorobets A, Oliva A (2014). MPI-CUDA sparse matrix–vector multiplication for the conjugate gradient method with an approximate inverse preconditioner, Computers & Fluids, 92, 244-252.

Pichel J, Rivera F, Fernández M, Rodríguez A (2012). Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs, Microprocessors and Microsystems, 36, 65-77.

Romero L, Zapata E (1995). Data distributions for sparse matrix vector multiplication, Parallel Computing, 21, 583-605.

Turner M (1959). The direct stiffness method of structural analysis, Structural and Materials Panel Paper, AGARD Meeting, Aachen, Germany.

Wilson E (1963). Finite element analysis of two-dimensional structures, Ph. D. Dissertation, Department of Civil Engineering, University of California at Berkeley, USA.

Živković M (2005). Nelinearna analiza konstrukcija, Mašinski fakultet u Kragujevcu, Kragujevac, Serbia.

Appendix A - saving notifications of sparse matrix

For example, matrix is defined as $A_{N,N}$ = [(15, 13, 7, 0, 8, 0), (13, 1, 0, 2, 7, 0), (7, 0, 5, 3, 0, 0), (0, 2, 3, 9, 0, 8), (8, 7, 0, 0, 11, 0), (0, 0, 0, 8, 0, 15)], N = 6.

For this matrix values of row **Rk**, column **Ck** and **Vk**, as well as auxiliary array Hn are:

R_K = (1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 6, 6), K = 20
C_K = (1, 2, 3, 5, 1, 2, 4, 5, 1, 3, 4, 2, 3, 4, 6, 1, 2, 5, 4, 6)
V_K = (15, 13, 7, 8, 13, 1, 2, 7, 7, 5, 3, 2, 3, 9, 8, 8, 7, 11, 8, 15)
H_{N+1} = (1, 5, 9, 12, 16, 19, 21), $H_{(N+1)}$ = K + 1

For all values for I=$H_{(P)}$ to $H_{(P+1)}$-1 $R_{(I)}$ = P

When there is symmetric matrix some notifications save only upper/lower triangular matrix. For this matrix those arrays have values:

RC_K = (1, 1, 1, 1, 2, 2, 2, 3, 3, 4, 4, 5, 6), K = 13
CR_K = (1, 2, 3, 5, 2, 4, 5, 3, 4, 4, 6, 5, 6)
V_K = (15, 13, 7, 8, 1, 2, 7, 5, 3, 9, 8, 11, 15)
H_{N+1} = (1, 5, 8, 10, 12, 13, 14), $H_{(N+1)}$ = K + 1

In PAK software we used short symmetric notation for both symmetric and non-symmetric matrices. When saving non-symmetric matrix values array is twice longer than rows / columns array. That way values array has N more values than standard notifications, but memory is saved because we do not save whole row /column arrays.

In our notation for this matrix we have these arrays:

H_{N+1} = (1, 2, 4, 6, 9, 12, 14), $H_{(N+1)} = K + 1$
CR_K = (1, 2, 1, 3, 1, 4, 3, 2, 5, 2, 1, 6, 4), K = 13
V_K = (15, 1, 13, 5, 7, 9, 3, 2, 11, 7, 8, 15, 8)

For the purpose of explaining non-symmetric matrix saving, we will assume that the values in the lower triangular matrix are different than in the upper triangular matrix (A_{ij} != A_{ji}) even they are same in this example.

H_K = (1, 2, 4, 6, 9, 12, 14), $H_{(N+1)} = K + 1$
CR_K = (1, 2, 1, 3, 1, 4, 3, 2, 5, 2, 1, 6, 4), K = 13
V_{2K} = (15, 1, 13, 5, 7, 9, 3, 2, 11, 7, 8, 15, 8, 0, 0, 13, 0, 7, 0, 3, 2, 0, 7, 8, 0, 8)

To know the position of element Vp, where p is greater than K column is the same as a row of element p-K ($RC_p = CR_{p-K}$) and row is the same as the column of element p-K (C, H_C < p-K and H_{C+1} > p-K). Zeroes in V_{2K} array are in positions of diagonal elements and these too can be omitted which would increase memory efficiency, but on the down side this would cause less efficiency in the solving of the problem because more mathematical operations would be needed to access elements below the diagonal elements.

Appendix B - forming of the auxiliary Boolean array

Let's assume that elements in the example matrix from Appendix A influence certain equations

Element 1 equations (1, 3)

Element 2 equations (3, 4)

Element 3 equations (2, 4)

Element 4 equations (1, 2, 5)

Element 5 equations (4, 6)

H_{N+1} = (1, 2, 4, 7, 10, 15, 18)

For dotting method, we will use the example using unsigned type to store the Boolean array.

Dotting method process:

B = (0, 0, 0) // array b on Fig 1. That contains 3 elements
Element 1: equations 1 and 3:
- I = 1, J = 1 => K = H(1) + J – I = 1 + 1 – 1 = 1

E = [(K + 7) / 8] = 1, P = 8 – (E * 8 – K) = 1
B_1 = B_1 or POS_1 = $(00000000)_{(2)}$ or $(10000000)_{(2)}$ = $(10000000)_{(2)}$ = 128
B = (128, 0, 0)
- I = 1, J = 3 => K = H(3) + J – I = 4 + 3 – 1 = 6

E = [(K + 7) / 8] = 1, P = 8 – (E * 8 – K) = 6
B_1 = B_1 or POS_6 = $(10000000)_{(2)}$ or $(00000100)_{(2)}$ = $(10000100)_{(2)}$ = 132
B = (132, 0, 0)
- I = 3, J = 3 => K = H(3) + J – I = 4 + 3 – 3 = 4

E = [(K + 7) / 8] = 1, P = 8 – (E * 8 – K) = 4
B_1 = B_1 or POS_4 = $(10000100)_{(2)}$ or $(00010000)_{(2)}$ = $(10010100)_{(2)}$ = 148
B = (148, 0, 0)

Element 2: equations 3 and 4

- $I = 3, J = 3 \Rightarrow K = H(3) + J - I = 4 + 3 - 3 = 4$

$E = [(K + 7) / 8] = 1, P = 8 - (E * 8 - K) = 4$

$B_1 = B_1$ or $POS_4 = (10010100)_{(2)}$ or $(00010000)_{(2)} = (10010100)_{(2)} = 148$

$B = (148, 0, 0)$

- $I = 3, J = 4 \Rightarrow K = H(4) + J - I = 7 + 4 - 3 = 8$

$E = [(K + 7) / 8] = 1, P = 8 - (E * 8 - K) = 8$

$B_1 = B_1$ or $POS_8 = (10010100)_{(2)}$ or $(00000001)_{(2)} = (10010101)_{(2)} = 149$

$B = (149, 0, 0)$

- $I = 4, J = 4 \Rightarrow K = H(4) + J - I = 7 + 4 - 4 = 7$

$E = [(K + 7) / 8] = 1, P = 8 - (E * 8 - K) = 7$

$B_1 = B_1$ or $POS_7 = (10010101)_{(2)}$ or $(00000010)_{(2)} = (10010111)_{(2)} = 151$

$B = (151, 0, 0)$

Element 3: equations 2 and 4

- $I = 2, J = 2 \Rightarrow K = H(2) + J - I = 2 + 2 - 2 = 2$

$E = [(K + 7) / 8] = 1, P = 8 - (E * 8 - K) = 2$

$B_1 = B_1$ or $POS_2 = (10010111)_{(2)}$ or $(01000000)_{(2)} = (11010111)_{(2)} = 215$

$B = (215, 0, 0)$

- $I = 2, J = 4 \Rightarrow K = H(4) + J - I = 7 + 4 - 2 = 9$

$E = [(K + 7) / 8] = 2, P = 8 - (E * 8 - K) = 1$

$B_2 = B_2$ or $POS_2 = (00000000)_{(2)}$ or $(10000000)_{(2)} = (10000000)_{(2)} = 128$

$B = (215, 128, 0)$

- $I = 4, J = 4 \Rightarrow K = H(4) + J - I = 7 + 4 - 4 = 7$

$E = [(K + 7) / 8] = 1, P = 8 - (E * 8 - K) = 7$

$B_1 = B_1$ or $POS_7 = (11010111)_{(2)}$ or $(00000010)_{(2)} = (11010111)_{(2)} = 215$

$B = (215, 128, 0)$

Element 4: equations 1, 2 and 5

- $I = 1, J = 1 \Rightarrow K = H(1) + J - I = 1 + 1 - 1 = 1$

$E = [(K + 7) / 8] = 1, P = 8 - (E * 8 - K) = 1$

$B_1 = B_1$ or $POS_1 = (10010111)_{(2)}$ or $(10000000)_{(2)} = (11110111)_{(2)} = 247$

$B = (247, 128, 0)$

- $I = 1, J = 2 \Rightarrow K = H(2) + J - I = 2 + 2 - 1 = 3$

$E = [(K + 7) / 8] = 1, P = 8 - (E * 8 - K) = 3$

$B_1 = B_1$ or $POS_3 = (10010111)_{(2)}$ or $(00100000)_{(2)} = (11110111)_{(2)} = 247$

$B = (247, 128, 0)$

- $I = 1, J = 5 \Rightarrow K = H(5) + J - I = 10 + 5 - 1 = 14$

$E = [(K + 7) / 8] = 2, P = 8 - (E * 8 - K) = 6$

$B_2 = B_2$ or $POS_6 = (10000000)_{(2)}$ or $(00000100)_{(2)} = (10000100)_{(2)} = 132$

$B = (247, 132, 0)$

- $I = 2, J = 2 \Rightarrow K = H(2) + J - I = 2 + 2 - 2 = 2$

$E = [(K + 7) / 8] = 1, P = 8 - (E * 8 - K) = 2$

$B_1 = B_1$ or $POS_2 = (10010111)_{(2)}$ or $(01000000)_{(2)} = (11110111)_{(2)} = 247$

$B = (247, 132, 0)$

- $I = 2, J = 5 \Rightarrow K = H(5) + J - I = 10 + 5 - 2 = 13$

$E = [(K + 7) / 8] = 2, P = 8 - (E * 8 - K) = 5$

$B_2 = B_2$ or $POS_5 = (10000100)_{(2)}$ or $(00001000)_{(2)} = (10001100)_{(2)} = 140$

$B = (247, 140, 0)$

- $I = 5, J = 5 \Rightarrow K = H(5) + J - I = 10 + 5 - 5 = 10$

$E = [(K + 7) / 8] = 2, P = 8 - (E * 8 - K) = 2$

$B_2 = B_2$ or $POS_2 = (10001100)_{(2)}$ or $(01000000)_{(2)} = (11001100)_{(2)} = 204$

$B = (247, 204, 0)$

Element 5: equations 4 and 6

- $I = 4, J = 4 \Rightarrow K = H(4) + J - I = 7 + 4 - 4 = 7$

$E = [(K + 7) / 8] = 1, P = 8 - (E * 8 - K) = 7$

$B_1 = B_1$ or $POS_7 = (11110111)_{(2)}$ or $(00000010)_{(2)} = (11110111)_{(2)} = 247$

$B = (247, 204, 0)$

- $I = 4, J = 6 \Rightarrow K = H(6) + J - I = 15 + 6 - 4 = 17$

$E = [(K + 7) / 8] = 3, P = 8 - (E * 8 - K) = 1$

$B_3 = B_3$ or $POS_1 = (00000000)_{(2)}$ or $(10000000)_{(2)} = (10000000)_{(2)} = 247$

$B = (247, 204, 128)$

- $I = 6, J = 6 \Rightarrow K = H(6) + J - I = 15 + 6 - 6 = 15$

$E = [(K + 7) / 8] = 2$, $P = 8 – (E * 8 – K) = 7$
$B_2 = B_2$ or $POS_7 = (11001100)_{(2)}$ or $(00000010)_{(2)} = (11001110)_{(2)} = 206$
B = (247, 206, 128)

B = (247, 206, 128) = (11110111, 11001110, 10000000)
$B_{(2)}$ = 1, 11, 101, 111, 10011, 101, 0000000 split by columns.

Now that we have fully created array B we can create array of row indexes. The row index array is of unknown size for now and is created dynamically.

- S = 0
- E = 1, P = 1, K = 1, C = 1

$U = B_K$ and $POS_P = (11110111)_{(2)}$ and $(10000000)_{(2)} = (10000000)_{(2)} = 128$
$S = S + 1 = 1$, $R_S = H(C) + C – K = 1 + 1 – 1 = 1$
R = (1)

- E = 1, P = 2, K = 2, C = 2

$U = B_K$ and $POS_P = (11110111)_{(2)}$ and $(01000000)_{(2)} = (01000000)_{(2)} = 64$
$S = S + 1 = 2$, $R_S = H(C) + C – K = 2 + 2 – 2 = 2$
R = (1, 2)

- E = 1, P = 3, K = 3, C = 2

$U = B_K$ and $POS_P = (11110111)_{(2)}$ and $(00100000)_{(2)} = (00100000)_{(2)} = 32$
$S = S + 1 = 3$, $R_S = H(C) + C – K = 2 + 2 – 3 = 1$
R = (1, 2, 1)

- E = 1, P = 4, K = 4, C = 3

$U = B_K$ and $POS_P = (11110111)_{(2)}$ and $(00010000)_{(2)} = (00010000)_{(2)} = 16$
$S = S + 1 = 4$, $R_S = H(C) + C – K = 4 + 3 – 4 = 3$
R = (1, 2, 1, 3)

- E = 1, P = 5, K = 5, C = 3

$U = B_K$ and $POS_P = (11110111)_{(2)}$ and $(00001000)_{(2)} = (00000000)_{(2)} = 0$
R = (1, 2, 1, 3)

- E = 1, P = 6, K = 6, C = 3

$U = B_K$ and $POS_P = (11110111)_{(2)}$ and $(00000100)_{(2)} = (00000100)_{(2)} = 4$
$S = S + 1 = 5$, $R_S = H(C) + C – K = 4 + 3 – 6 = 1$
R = (1, 2, 1, 3, 1)

- E = 1, P = 7, K = 7, C = 4

$U = B_K$ and $POS_P = (11110111)_{(2)}$ and $(00000010)_{(2)} = (00000010)_{(2)} = 2$
$S = S + 1 = 6$, $R_S = H(C) + C – K = 7 + 4 – 7 = 4$
R = (1, 2, 1, 3, 1, 4)

- E = 1, P = 8, K = 8, C = 4

$U = B_K$ and $POS_P = (11110111)_{(2)}$ and $(00000001)_{(2)} = (00000001)_{(2)} = 1$
$S = S + 1 = 7$, $R_S = H(C) + C – K = 7 + 4 – 8 = 3$
R = (1, 2, 1, 3, 1, 4, 3)

- E = 2, P = 1, K = 9, C = 4

$U = B_K$ and $POS_P = (11001110)_{(2)}$ and $(10000000)_{(2)} = (10000000)_{(2)} = 128$
$S = S + 1 = 8$, $R_S = H(C) + C – K = 7 + 4 – 9 = 2$
R = (1, 2, 1, 3, 1, 4, 3, 2)

- E = 2, P = 2, K = 10, C = 5

$U = B_K$ and $POS_P = (11001110)_{(2)}$ and $(01000000)_{(2)} = (01000000)_{(2)} = 64$
$S = S + 1 = 9$, $R_S = H(C) + C – K = 10 + 5 – 10 = 5$
R = (1, 2, 1, 3, 1, 4, 3, 2, 5)

- E = 2, P = 3, K = 11, C = 5

$U = B_K$ and $POS_P = (11001110)_{(2)}$ and $(00100000)_{(2)} = (00000000)_{(2)} = 0$
R = (1, 2, 1, 3, 1, 4, 3, 2, 5)

- E = 2, P = 4, K = 12, C = 5

$U = B_K$ and $POS_P = (11001110)_{(2)}$ and $(00010000)_{(2)} = (00000000)_{(2)} = 0$
R = (1, 2, 1, 3, 1, 4, 3, 2, 5)

- E = 2, P = 5, K = 13, C = 5

$U = B_K$ and $POS_P = (11001110)_{(2)}$ and $(00001000)_{(2)} = (00001000)_{(2)} = 8$
$S = S + 1 = 10$, $R_S = H(C) + C – K = 10 + 5 – 13 = 2$
$R = (1, 2, 1, 3, 1, 4, 3, 2, 5, 2)$
- $E = 2, P = 6, K = 14, C = 5$

$U = B_K$ and $POS_P = (11001110)_{(2)}$ and $(00000100)_{(2)} = (00000100)_{(2)} = 4$
$S = S + 1 = 11$, $R_S = H(C) + C – K = 10 + 5 – 14 = 1$
$R = (1, 2, 1, 3, 1, 4, 3, 2, 5, 2, 1)$
- $E = 2, P = 7, K = 15, C = 6$

$U = B_K$ and $POS_P = (11001110)_{(2)}$ and $(00000010)_{(2)} = (00000010)_{(2)} = 2$
$S = S + 1 = 12$, $R_S = H(C) + C – K = 15 + 6 – 15 = 6$
$R = (1, 2, 1, 3, 1, 4, 3, 2, 5, 2, 1, 6)$
- $E = 2, P = 8, K = 16, C = 6$

$U = B_K$ and $POS_P = (11001110)_{(2)}$ and $(00000001)_{(2)} = (00000000)_{(2)} = 0$
$R = (1, 2, 1, 3, 1, 4, 3, 2, 5, 2, 1, 6)$
- $E = 3, P = 1, K = 17, C = 6$

$U = B_K$ and $POS_P = (10000000)_{(2)}$ and $(10000000)_{(2)} = (10000000)_{(2)} = 128$
$S = S + 1 = 12$, $R_S = H(C) + C – K = 15 + 6 – 17 = 4$

$R = (1, 2, 1, 3, 1, 4, 3, 2, 5, 2, 1, 6, 4)$

New H row that contains information on starting index in the rows array can be created alongside a row index array R or with the help of it after its creation because when a new column starts relation $R_X > R_{X-1}$ is true and starts always with $R_1 = 1$.

NH7 = (1, 2, 4, 6, 9, 12, 14)

Appendix C - indexing method example

The first thing to do in this method is to calculate columns density – number of non-zero elements in each column above diagonal similar to how it was done with determination where non-zero element is in dotting method but instead of marking element as non-zero increase column density. For example:

1.
2.$CD_N = (0, 0, 0, 0, 0, 0)$
3.Element 1: eq 1 and 3
- $I = 1, J = 1, I \leq J \Rightarrow CD(J) = CD(J) + 1$

4.$CD_N = (1, 0, 0, 0, 0, 0)$
- $I = 1, J = 3, I \leq J \Rightarrow CD(J) = CD(J) + 1$

5.$CD_N = (1, 0, 1, 0, 0, 0)$
- $I = 3, J = 3, I \leq J \Rightarrow CD(J) = CD(J) + 1$

6.$CD_N = (1, 0, 2, 0, 0, 0)$

7. Element 2: eq 3 and 4
- $I = 3, J = 3, I \leq J \Rightarrow CD(J) = CD(J) + 1$

8.$CD_N = (1, 0, 3, 0, 0, 0)$
- $I = 3, J = 4, I \leq J \Rightarrow CD(J) = CD(J) + 1$

9.$CD_N = (1, 0, 3, 1, 0, 0)$
- $I = 4, J = 4, I \leq J \Rightarrow CD(J) = CD(J) + 1$

10.$CD_N = (1, 0, 3, 2, 0, 0)$

11.Element 3: eq 2 and 4
- $I = 2, J = 2, I \leq J \Rightarrow CD(J) = CD(J) + 1$

12.$CD_N = (1, 1, 3, 2, 0, 0)$
- $I = 2, J = 4, I \leq J \Rightarrow CD(J) = CD(J) + 1$

13.$CD_N = (1, 1, 3, 3, 0, 0)$
- $I = 4, J = 4, I \leq J \Rightarrow CD(J) = CD(J) + 1$

14.$CD_N = (1, 1, 3, 4, 0, 0)$

15.

6.

7.Element 4: eq 1, 2 and 5

- $I = 1, J = 1, I \le J \Rightarrow CD(J) = CD(J) + 1$

18.$CD_N = (2, 1, 3, 4, 0, 0)$

- $I = 1, J = 2, I \le J \Rightarrow CD(J) = CD(J) + 1$

19.$CD_N = (2, 2, 3, 4, 0, 0)$

- $I = 1, J = 5, I \le J \Rightarrow CD(J) = CD(J) + 1$

20.$CD_N = (2, 2, 3, 4, 1, 0)$

- $I = 2, J = 2, I \le J \Rightarrow CD(J) = CD(J) + 1$

21.$CD_N = (2, 3, 3, 4, 1, 0)$

- $I = 2, J = 5, I \le J \Rightarrow CD(J) = CD(J) + 1$

22.$CD_N = (2, 3, 3, 4, 2, 0)$

- $I = 5, J = 5, I \le J \Rightarrow CD(J) = CD(J) + 1$

23.$CD_N = (2, 3, 3, 4, 3, 0)$

4.Element 5: eq 4 and 6

- $I = 4, J = 4, I \le J \Rightarrow CD(J) = CD(J) + 1$

25.$CD_N = (2, 3, 3, 5, 3, 0)$

- $I = 4, J = 6, I \le J \Rightarrow CD(J) = CD(J) + 1$

26.$CD_N = (2, 3, 3, 5, 3, 1)$

- $I = 6, J = 6, I \le J \Rightarrow CD(J) = CD(J) + 1$

$CD_N = (2, 3, 3, 5, 3, 2)$

Now we have column density, but in this array there is some issue – CD(I) cannot be greater than I, so those elements will be decreased, and also some column density shows it is greater than actual density. This occurs due to two (2) reasons.

The first is that some equations are calculated several times because of its occurrence in several elements, this can be annulled by starting CD with 1 instead of 0 and increase only for I <> J.

The second reason is that in real examples some group of equations appear in several elements. This error cannot be annulled because in real problems we do not know how many of those groups occur and in what elements.

After reduction column density is: $CD_N = (1, 2, 3, 4, 3, 2)$.

Now we create a new temporary H_{N+1} array with following rules:

$H(1) = 1$ and $H(i+1) = H(i) + CD(i)$, for i from 1 to N.

Array H has the following values: $H_{N+1} = (1, 2, 4, 7, 11, 14, 16)$.

Now that we know the maximum possible length ($H(N + 1) - 1 = 15$) of non-zero elements in UTM we can create array R of row indexes. The first thing to do is to assign diagonal elements so those elements can be skipped while assigning values of row indexes. This is achieved with formula $R(H(I)) = I$, for each I between 1 and N. So at beginning we have:

$R_{15} = (1, 2, 0, 3, 0, 0, 4, 0, 0, 0, 5, 0, 0, 6, 0)$

With similar process as for creating column density we create row indexes:

7.Element 1: eq 1 and 3:

- I = 1 and J = 1, I=J – do nothing
- I = 1 and J = 3, I < J => insert row index I in column J

28.R_{15} = (1, 2, 0, 3, 1, 0, 4, 0, 0, 0, 5, 0, 0, 6, 0)

- I = 3 and J = 3, I=J – do nothing

29.

0.Element 2: eq 3 and 4:

- I = 3 and J = 3, I=J – do nothing
- I = 3 and J = 4, I < J => insert row index I in column J

31.R_{15} = (1, 2, 0, 3, 1, 0, 4, 3, 0, 0, 5, 0, 0, 6, 0)

- I = 4 and J = 4, I=J – do nothing

2.Element 3: eq 2 and 4:

- I = 2 and J = 2, I=J – do nothing
- I = 2 and J = 4, I < J => insert row index I in column J

33.R_{15} = (1, 2, 0, 3, 1, 0, 4, 3, 2, 0, 5, 0, 0, 6, 0)

- I = 4 and J = 4, I=J – do nothing

4.Element 4: eq 1, 2 and 5:

- I = 1 and J = 1, I=J – do nothing
- I = 1 and J = 2, I < J => insert row index I in column J

35.R_{15} = (1, 2, 1, 3, 1, 0, 4, 3, 2, 0, 5, 0, 0, 6, 0)

- I = 1 and J = 5, I < J => insert row index I in column J

36.R_{15} = (1, 2, 1, 3, 1, 0, 4, 3, 2, 0, 5, 1, 0, 6, 0)

- I = 2 and J = 2, I=J – do nothing
- I = 2 and J = 5, I < J => insert row index I in column J

37.R_{15} = (1, 2, 1, 3, 1, 0, 4, 3, 2, 0, 5, 1, 2, 6, 0)

- I = 5 and J = 5, I=J – do nothing

8.Element 3: eq 4 and 6:

- I = 4 and J = 4, I=J – do nothing
- I = 4 and J = 6, I < J => insert row index I in column J

39.R_{15} = (1, 2, 1, 3, 1, 0, 4, 3, 2, 0, 5, 1, 2, 6, 4)

I = 6 and J = 6, I=J – do nothing

Now we have completed array R, but it contains some 0 elements that should be removed, and it is not sorted as it was with dotting method. The fastest way to do this is with the counting sort, so after removing of 0 elements we have new values for array R and H:

R_{13} = (1, 2, 1, 3, 1, 4, 3, 2, 5, 1, 2, 6, 4)

H_{7} = (1, 2, 4, 6, 9, 12, 14)

If sorted:

R_{13} = (1, 2, 1, 3, 1, 4, 3, 2, 5, 2, 1, 6, 4)