UNIVERSIDAD DE GRANADA

MÁSTER UNIVERSITARIO EN TECNOLOGÍAS PARA LA INVESTIGACIÓN DE MERCADOS Y MARKETING

TRABAJO DE FIN DE MÁSTER

Operations with Pivotal and Sparse matrices having large binary size and their implementation. Applications to image processing

Tutores:
Laiachi El Kaoutit Zerri
Carlos Rodríguez Domínguez

Autor:
Ștefan-Daniel Achirei

CEUTA – Junio 2019

Abstract

The first objective of this paper is to provide a mathematical definition and implement efficient algebraic operations with square sparse matrices of size 256. The operations needed the most in matrix algebra considered for implementation are: multiplication, transposing, summation and subtraction.

Another necessity has arisen while working with sparse matrices which have so little embedded information: mathematically defining and implementing an efficient representation and manipulation of data provided by sparse matrices. This new data format will be called *Sparse Matrix Format* (SMF) during this paper.

Taking notice of the sparsity attribute of the matrix will result in a more efficient approach which implies the development of specific applications that use a special data structure.

The necessity of more efficient operations has arisen while working with raw MRI Brain Scan images. Most of the times this data needs a lot of processing in order to make it easier to read and interpret by a medical doctor. One complete brain scan can contain up to 170 pictures, each one of them corresponding to a horizontal section of the human head. A doctor may recommend a MRI head scan if they suspect that a person has: blocked arteries, multiple sclerosis, eye or inner ear issues, epilepsy, a brain tumor or stroke.

The effectiveness of multiplication algorithm and storing format is analyzed on two working data sets: in the case of treatment imagines from *Anonymous MRI Brain Scan Images Database (The University of Granada)* and on data from *The University of Florida Sparse Matrix Collection*.

Contents

Chapter 1

Introduction

When solving different problems from economy, technical fields, social environment, optimization, as well as in modeling or simulating industrial and technological processes it is necessary to determine the mathematical model that describes the problem itself. The description of such systems leads to some mathematical systems which usually involves solving linear algebraic equation systems in which the associated coefficient matrix is sparse (most of the coefficients are zero). From the practical point of view the analysis of such systems produces very large mathematical models that may involve linear algebraic equation systems that can include thousands of equations. From a computational perspective, such systems require a lot of memory to represent them, and a lot of time to provide a solution to the equation system.

Consequently, the mathematical models of many real processes imply a large number of variables and constraints which contributes to the sparsity phenomenon of a matrix: Most of the entries are zero and are not connected between each other.

Taking notice of the sparsity property of the matrix can result in a more efficient approach, implying the development of specific applications that use a special data representation/structure. This will save memory and reduce the run time.

The main purpose of this work is to give a mathematical definition and implementation of an efficient computer representation of square sparse matrices. This format will be referred to as *Sparse Matrix Format* (SMF) from now on. The second objective is to mathematically define and implement efficient algorithms for the associated operations to this particular matrix format: multiplication, transposing, summation and subtraction.

Moreover, the SMF proposal is compared, in terms of memory use and run time, to the classical matrix representation and associated operations to ensure its efficiency and its practical applicability.

To sum up the achievements to be presented in this work, a square sparse matrix of dimensions *n*-by-*n*, with *NZ* non-zero elements, *SMF* data structure stores in memory *2*NZ+n* entries in comparison with the classical matrix format that stores *n*n* numbers which is greater than or equal to *NZ*. Thus for matrices with *NZ* smaller than or equal to *n*(n+1)/2* the SMF uses less or equal memory compared to the classical matrix format. In practice the big sparse matrices have less than 3% (see [1]) of the entries non-zero, so there is an obvious improvement in memory use. In terms of run time, in image processing we have been able to achieve up to 75 times better timings.

The remaining of this chapter will detail what is a sparse matrix, where is it mostly used, disadvantages of the standard format and what other formats have been proposed to represent sparse matrices.

1.1 Sparse matrix: definition, properties and application

In computer science having a matrix with a wide range of null values is different than having a matrix with all elements non-zero. If the most elements of a matrix are zero, the matrix is called *sparse matrix* in comparison with a *dense matrix* which has the majority of the elements different than zero.

In order to determine if a matrix is sparse or not, it is required to introduce the notion of *sparsity* of a matrix. The division result between the number of null-valued elements and the total number of elements is defined as *sparsity of the matrix* (see [1]). On the other hand the *density* is defined as the division between non-zero and total number of elements. In other words *sparsity* is equal to 1 minus *density* of a matrix. Using this definitions, a matrix is sparse if its sparsity is greater than 0.5.

In practical applications there are encountered large sparse matrices with non-zero entries between 0.15% and 3%, the sparsity varies form 0.97 to 0.9985.

$$A_{5*5} = \begin{bmatrix} 0 & 0 & A_{13} & 0 & 0 \\ 0 & 0 & A_{23} & A_{24} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & A_{42} & A_{43} & 0 & 0 \\ A_{51} & 0 & 0 & 0 & A_{55} \end{bmatrix} \tag{1.1}$$

$$sparsity(A) = \frac{(n^2) - NZ}{n^2} = \frac{(5^2) - 7}{5^2} = \frac{18}{25} = 0.72 > 0.5 \tag{1.2}$$

$$density(A) = \frac{NZ}{n^2} = \frac{7}{25} = 0.28 \tag{1.3}$$

A square matrix example of dimensions *5*-by-*5* and *7* non-zero elements out of *25* is that of (1.1). The *sparsity* respectively *density* are calculated as shown in equations (1.2) and (1.3). Because its *sparsity* value is greater than *0.5* the matrix is sparse.

Sparse matrices are encountered in modeling or simulating processes from different fields like: industry, economics, technology, social, etc (see [1]). Sparse matrices are the core of solving systems of linear equations. Some fields that widely use linear algebra represented by sparse matrices are:

- *modeling and simulation of large-scale systems*: described by thousands of linear algebraic equations in form of large sparse matrices
- *computer graphics*: adding and multiplying matrices is the most common operation in image processing
- *recommendation systems or search engines*: for instance links on the web are described in a sparse matrix, element *(i, j)* is non-zero if web page *i* has a link to web page *j*. Examples of this such implementations: *Google Ranking System* or *Facebook Friend Relations*.
- *machine learning*: in applied machine learning large sparse matrices are often used, for instance the correlation matrix or stochastic matrix whose edges define a relation between data points.

The last two examples are in fact based on the incidence matrix of a given directed graph.

1.2 Inefficiency of standard representation and operations with sparse matrices

In computers, the classical representation of a matrix (a bidimensional array) has been proven to be inefficient for the case of large sparse matrices. The first disadvantage of such structure is the uneconomical use of memory, the biggest part is allocated for useless data which is not carrying any information from the mathematical perspective. Using the largest part of the allocated memory for zero entries is not justified because they do not contribute to the result of the matrix operations such as multiplication, summation, etc. At the same time this large number of zero-values increase the run time of operations because of the repetitive summation and multiplication between zero values. Therefore if a matrix is sparse, but classically represented, more processing time and memory is inefficiently used on the zero-elements of the matrix than on the useful information from the matrix. This inconvenient is better observed as the size of the matrix increases.

Therefore large-scale systems demand for some alternative ways to represent sparse matrices in a more compact manner. The main objective here is to not represent the null-values of the matrix. Therefore, the storage format needs to incorporate both some way to identify the position in the original matrix and the non-zero values. It is an advantage and often essential to use specific algorithms and data structures that benefit from the structure of the matrix. Sparse data is easy to compress and thus require significantly less memory. Some enormous sparse matrices are almost impossible to manipulate using regular dense matrix algorithms.

1.3 Storing a sparse matrix

A matrix is typically stored in a two-dimensional array. Each entry in the array represents an element a_{ij} of the matrix. The element is accessed by the two indices i and j. Formally, the row is represented by i respectively the column by j. The numbering is done top to bottom for rows respectively left to right for columns. For an n-by-n square matrix the required memory to store the matrix in the conventional format is proportional to n.

When dealing with a sparse matrix, the memory requirement can be substantially reduced by storing only the elements different than zero. Taking into consideration the number and the distribution of non-zero elements, specific data structures can be used to save considerable memory when compared to the traditional approach. The compromise is that accessing the individual elements becomes more difficult and helping structures are needed to be able to get back to the original matrix.

As in [2], considering the accessibility of the compressed form the formats can be divided into two groups:

- easier value modification, harder element access: Dictionary of Keys (DOK) [3], List of Lists (LIL) [4], Coordinate List (COO) [4]
- easier element access: Compressed Sparse Column[5], Compressed Sparse Row or Yale format [5]. This formats support implementation of matrix operation.

Chapter 2

Particular types of sparse matrices

There are types of sparse matrices that present some characteristics which are helpful in defining new ways of storing the non-zero values of a matrix. This chapter will define a few particular types of sparse matrices and suggestions on how to store them efficiently in the memory. Following sub-chapters refer to these particular matrices: band matrix, diagonal matrix and permutation matrix.

2.1 Band matrix

One particular type of sparse matrix is the band matrix. In the case of band sparse matrix the non-zero values are grouped around the main or second diagonal (see [6]). For instance the matrix *A* of *5-by-8* dimensions:

$$A_{5*8} = \begin{bmatrix} 1 & 9 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 8 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 2 & 1 \end{bmatrix} \tag{2.1}$$

is a sparse matrix in which the useful elements are grouped on or near the main diagonal.

A group ia a non-zero consecutive entries in a given row. The goups of this matrix are written in the following table:

Table 2.1: Groups

1	9	2	8	3	4	5	6	7	3	2	1
1st row		2nd row			3rd row		4th row		5th row		

Using the information that the non-zero entries are grouped on the rows in a small range, it is possible to define a new method to store a band matrix. For each row it is stored the column index of the first non-zero entry and the column index of the last non-zero entry from that row (see Table 2.2). In a separate one-dimensional array the non-zero values (see [1]). Both Table 2.2 and Table 2.1 describe this format.

Table 2.2: Begin-end indexes for each row

group start index	0	1	2	3	5
group end index	1	3	3	4	7

It is observed that for each group of non-zero entries on a row it is stored in two one-dimensional array the column index of the first respectively the last non-zero value. Because of this

reason the values array stores the zero elements if there are in between non-zero entries. If on a row we have two groups separated by zero, then the matrix can be separated in two band matrix, therefore this method becomes.

For instance, for the band matrix A of example (2.1) with *NZ = 12* (non-zero entries) this storing method is more efficient than the coordinate format. If the matrix entries are stored on *4 byte* integers, then the coordinate format takes DIM_{coord} *= 3 * NZ * 4 bytes = 3 * 12 * 4 bytes = 144 bytes* (see equation (3.7) below) while the total needed memory for this format is *88 bytes* (see equation (2.2)). Here is the general formula:

$$DIM_{band} = (DIM_{start} + DIM_{end} + DIM_{val}) * 4bytes = (5+5+12) * 4bytes = 88 \quad bytes, \tag{2.2}$$

where:

- DIM_{start} = 5 = length of the group start index array,
- DIM_{end} = 5 = length of the group end index array,
- DIM_{val} = 12 = length of the group array.

2.2 Diagonal matrix

Another particular sparse matrix is the diagonal one. It is only the case of square matrices that have non-zero entries on the main or second diagonal. For the following matrix A:

$$A_{6*6} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 \end{bmatrix} \tag{2.3}$$

the representation method implies only storing the elements on the diagonal in an one-dimensional array A_{diag} (see [1]):

$$A_{diag} = [1, 9, 2, 8, 3, 7] \tag{2.4}$$

Total number of used bytes to store data is *24*:

$$DIM_{A_{diag}} = len(A_{diag}) * 4bytes = 6 * 4bytes = 24 \quad bytes \tag{2.5}$$

In the case of having non-zero entries on the second diagonal, all the above statements are still valid with one mention, the A_{diag} array will store the entries from the second diagonal.

2.3 Permutation matrix

The permutation matrix has on each row or column only a value of one, all the rest being zeros. This matrix is utilized in algebraic operations to permute the coordinates according to a previously established model. For instance, considering following matrix M:

$$M_{5*5} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

and the vector *X=[1, 2, 3, 4, 5]*, by multiplying the two, vector *XP=[3, 4, 2, 1, 5]* is obtained. The elements of X were rearranged according to the values of one from matrix M.

In order to store such a matrix it is only necessary to know the size of the matrix and the column index where the *1* is on each row (see [7]):

$$M = [3, 2, 0, 1, 4] \tag{2.7}$$

Considering the above examples of different types of sparse matrices it can be stated that a specific format for a specific type of matrix is far more efficient for storing it. In practice usually the working set of data is from the same class with the same particularities.

Chapter 3

Compact formats for sparse matrices

In order to reduce the storage space used for a sparse matrix, many data structures have been proposed. Specialized data structures reduce the required storage size by storing only the non-zero values of the matrix. Using specialized formats for sparse matrices can improve both the run time and storage space. This chapter describes different formats and the trade-off when choosing one over another.

3.1 Dictionary of keys

We start with *Dictionary of Keys (DOK)* format [3]. This is implemented using a dictionary that maps the (row, column) tuples to the value of the elements. The missing elements from the dictionary are considered to be zero. This format is mostly used to incrementally construct a sparse matrix in random order. It is not so efficient for iterating over non-zero elements in row/column order. Dictionary of keys format is typically used to construct the matrix and then, for processing, it is first converted to another more efficient format. For example, the matrix A of equation (1.1), has the *DOK* as follows:

$$\{(1,3) : A_{13}; (2,3) : A_{23}; (2,4) : A_{24}; (4,2) : A_{42}; (4,3) : A_{43}; (5,1) : A_{51}; (5,5) : A_{55}\} \quad (3.1)$$

This sequence is constructed assigning for each pair (row index, column index) the non-zero value of the matrix entry.

3.2 List of lists

The *List of lists* or *LIL* format (see [4]) consists of one list for each row of the matrix. Each element encapsulates the column index and the value. For easier iteration this lists are ordered by the column index. For example if A is the matrix of (1.3), then *LIL* format has the following structure:

Table 3.1: List of lists

$[1] \rightarrow$	$[3, A_{13}] \rightarrow$	$NULL$	
$[2] \rightarrow$	$[3, A_{23}] \rightarrow$	$[3, A_{24}] \rightarrow$	$NULL$
$[3] \rightarrow$	$NULL$		
$[4] \rightarrow$	$[2, A_{42}] \rightarrow$	$[3, A_{43}] \rightarrow$	$NULL$
$[5] \rightarrow$	$[1, A_{51}] \rightarrow$	$[3, A_{55}] \rightarrow$	$NULL$

Each row of the matrix is represented as a row in the above table. It contains multiple tuples, each consists of the column index and the non-zero value of the matrix entry.

3.3 Coordinate list

COO or *Coordinate list* (see [4]) is implemented using a list of *(row, column, value)* tuples. To improve access time the entries of the list can be kept sorted first by row, then by column. This format is also good for incremental construction of the matrix.

It is also called *aleatory sore format*. Because each non-zero entry of the matrix is individually identified it is also possible that the order could be aleatory. The advantage of an aleatory order is that the non-zero entries are added at the end without interfering with the rest or having to iterate over some elements.

If the matrix is symmetric this format is simplified by using two lists: one for the non-zero entries from the main diagonal and another for the ones above the diagonal (see [1]).

For example the matrix *A* of (1.3) has the following form:

$$\{(1,3,A_{13}) \to (2,3,A_{23}) \to (2,4,A_{24}) \to (4,2,A_{42}) \to (4,3,A_{43}) \to (5,1,A_{51}) \to (5,5,A_{55})\} \tag{3.2}$$

Therefore besides the *(row, column, value)* tuple each node of the list also encapsulates a pointer to the next node. Depending on the address length of the machine (*32/64 bits*) the pointer uses *4/8 bytes* [1]. Considering *4-byte* words on a *64-bit* machine, the total number of words needed to store this format as in (3.3).

$$DIM_{COO} = 3 \times NZ + 2 \times NZ \quad \textit{(data and pointer)}, \tag{3.3}$$

where as before *NZ* represents the non-zero entries of matrix *A*.

The division between this format's use of memory and the standard one as in (3.4). The upper limit of density so that this format is still more efficient than the classical one, as it is given in [1] is *0,2*.

$$r_{COO} = 5 \times density(A), \tag{3.4}$$

where *density(A)* is calculated using equation (1.3).

3.4 Coordinate format

The *Coordinate format* implies storing the sparse matrix entries as row index, column index and the non-zero value. Considering the following matrix *A*:

$$A_{5*5} = \begin{bmatrix} 0 & 0 & 0 & 9 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 \end{bmatrix} \tag{3.5}$$

The density of this *5*-by-*5* dimensions matrix *A* with *NZ=5* non-zero entries is:

$$density(A) = \frac{NZ}{n^2} = \frac{5}{5^2} = \frac{5}{25} = 0.25 \tag{3.6}$$

Following [1], in order to represent matrix *A* in the *coordinate format*, three one-dimensional array are defined as follows:

- row [ ] - stores the row number of the non-zero entry,
- col [ ] - stores the column number,

- val [ ] - stores the actual value.

The coordinate format memory use is defined by equation (3.7). If values respectively indexes are represented on *4 bytes* words, matrix *A* takes *15 bytes*.

$$DIM_{COO} = 3 \times NZ \quad words \tag{3.7}$$

Table 3.2: Matrix A represented in Coordinate Format

ROW	**COL**	**VAL**
0	3	9
1	1	2
3	0	1
3	1	5
4	4	7

In order to perform operations with sparse matrices defined in this way, a very important role is played by *row* and *col* arrays. For the result matrix array the three vectors must allocate enough for the new entries that may appear.

In this way for summation of the sparse matrices defined in Table 3.3 and Table 3.4 below is represented in Table 3.5.

Table 3.3: Matrix A represented in Coordinate Format

ROW_A	COL_A	VAL_A
0	0	-4
1	1	7
3	3	8

Table 3.4: Matrix B represented in Coordinate Format

ROW_B	COL_B	VAL_B
0	0	4
1	1	-7
2	1	8
3	1	5
3	2	6

Table 3.5: Result matrix C = A + B represented in Coordinate Format

ROW_C	COL_C	VAL_C
0	0	0
1	1	0
2	1	8
3	0	5
3	2	6
3	3	8
?	?	?
?	?	?

Because it is assumed that there is no match between indexes of the two matrices, the result vectors have a longer length than it is actually needed in this example. The unused cells are marked in the result with *?*.

Arrays ***ROW_C, COL_C*** and ***VAL_C*** have the length equal to *8* that is calculated using equation (3.8).

$$len(ROW\_C/COL\_C/VAL\_C) = len(ROW\_A) + len(ROW\_B), \tag{3.8}$$

where ***len*** is the function that returns the length of a one-dimensional array. For an array defined as (3.9):

$$int \quad a[n]; \tag{3.9}$$

where *n* is the vector dimension, ***Len*** function is given by (3.10):

$$len(a) = n + 1. \tag{3.10}$$

Because this problem is addressed to sparse matrices only, naturally the zero-value entries are eliminated, the final result being expressed in Table 3.6.

Table 3.6: Final result of summing sparse matrices A and B

ROW_C	COL_C	VAL_C
2	1	8
3	0	5
3	2	6
3	3	8
?	?	?
?	?	?
?	?	?
?	?	?

$$density(C) = \frac{4 \times 3}{8 \times 3} = \frac{12}{24} = 0.5 \tag{3.11}$$

The same as Table 3.5 there are cells in Table 3.6 which are not used to store any information, marked here with *?*. According to equation (3.11), Table 3.5 describes a sparse matrix with a density of *0.5*. Obviously there is an inefficient use of memory. For instance if arrays ***ROW_A*** and ***ROW_B*** have 3 respectively 5 used entries but they are defined of length 10, then the result array ***ROW_C*** will be defined of length 20.

According to the minimization criterion for used memory, this approach is not optimal because it requires more memory than used in the most of the cases. In order to achieve this goal, the implementation of a format for sparse matrices will use dynamically allocated arrays, the associated operation will generate arrays with a 100% use of memory.

In the case of multiplication or inverse it is possible that the result matrices will not respect the condition for a sparse matrix.

3.5 Compressed sparse row (or Yale format)

The *CSR* or *Compressed sparse row* format as described in [5], is implemented so that the matrix is represented by three one-dimensional arrays. This format has first been used in mid-1960s and has been fully described in 1977 (see [5]). Considering a matrix *M* of size *n*-by-*n*, with *NZ* non-zero elements, the three arrays *(A, IA, JA)* describing the matrix have the following meaning:

- *A* is of length *NZ* (number of non-zero elements) and stores the non-zero elements in row-major order
- *IA* is of length *n+1* and it is defined recursively:
  IA[0] = 0
  IA[i] = IA[i] + number of non-zero elements on the *i*-th row.
- *JA* is of length *NZ* and it contains the column index for all non-zero elements from the original matrix in a row-major order

For instance the non-zero elements form i^{th} row in the original matrix are stored form ***A[IA[i]]*** to ***A[IA[i+1]-1]***, including both ends. The column corresponding to each value in ***A*** is at the same index in ***JA***. This format uses less memory to store entries of a matrix only if condition (3.12) is respected (see [1]).

$$NZ < \frac{n \times (n-1) - 1}{2} \tag{3.12}$$

For example the square matrix *A* displayed in (3.13) of dimensions *4-by-4* and *4* non-zero elements:

$$A_{4*4} = \begin{bmatrix} 0 & 7 & 0 & 0 \\ 0 & 5 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 \end{bmatrix} \tag{3.13}$$

it has the associated **CSR format**:

$$\begin{aligned} &\text{A = [7, 5, 2, 8]} \\ &\text{IA = [0, 1, 3, 3, 4]} \\ &\text{JA = [1, 1, 2, 0]} \end{aligned} \tag{3.14}$$

Knowing the three arrays *A, IA* and *JA* the exact position in the original matrix can be easily calculated for every entry. For row number *i=1* (counting rows starts from 0, not 1), the non-zero elements are from *A[IA[1]]* to *A[IA[1+1]-1] = A[1]* to *A[2]*, which corresponds to *5* and *2*. For *A[1] = 5*, the column index is *JA[1] = 1* and for *A[2] = 2*, the column in the original matrix is *JA[2] = 2*.

Matrix *A* is sparse as it fulfills the constraint, *sparsity(A) = 0.75*. This *CSR* example contains *NZ*2+n+1 = 13* entries in comparison to the classical matrix format, which in this case has *16* entries.

3.6 Binary identified format

This method is based on the binary logic of the computing system. The non-zero entries are saved in a *primary memory zone (PZ)* in form of an one-dimensional array of length NZ. The structure of matrix is indicated using a binary sequence stored in a *secondary memory zone (SZ)*, see [1] for more details. For example, if we take the matrix (3.13), then:

$$PZ = [7, 5, 2, 8] \tag{3.15}$$

$$SZ = \begin{bmatrix} 0100 \\ 0110 \\ 0000 \\ 1000 \end{bmatrix} \tag{3.16}$$

Matrix A is stored in row-major order, another possibility would be column-major order. To reduce even more the memory needed to store the matrix, the secondary zone is bitwise.

If a matrix A with dimensions of *n-by-n*, *NZ* non-zero entries and the element data type of *b* bytes length, then the primary memory zone needs *n*n*density(A)* words of length *b* bytes. The secondary memory zone needs *(n*n)/(8*b)* words. The total number of words (of 4 bytes) needed to store matrix *A* using the two memory zones is *5*. The formula is given as follows:

$$DIM_{binary} = n^2 \times density(A) + \frac{n^2}{8 \times b} (\text{ in words of b bytes}) \tag{3.17}$$

Because the matrix classical format uses $DIM_{standard} = n*n$ words, the ratio between the binary identified format and the standard format is:

$$r = \frac{DIM_{binary}}{DIM_{standard}} = density(A) + \frac{1}{8 \times b} \tag{3.18}$$

Considering that entries in matrix *A* are unsigned integers stored on *4 bytes* each, then *r=0.28* is calculated in (3.19), indicating that using this format the matrix uses approximately a quarter of the original use of memory.

$$r_M = density(M) + \frac{1}{8 \times 4} = 0.25 + \frac{1}{32} \simeq 0.28 \tag{3.19}$$

The upper limit of the density at which the binary format uses the same memory as the standard format is 96% as described in [1] and can be calculated if *r=1* following the equation (3.20). For a matrix *A* if 15 out of 16 entries are non-zero, the binary format still uses less memory.

$$\begin{gathered} density(A)_{max} = 1 - \frac{1}{8 \times b} \\ for A : density(A)_{max} = 1 - \frac{1}{32} \simeq 0.96 \end{gathered} \tag{3.20}$$

This memory format is different than other because in the secondary zone there is allocated memory for the zero-values also. The binary identification format is less efficient for very large matrices with the sparsity value very big. Also the biggest challenge using this format is the complexity of the matrix algorithms as summation or multiplying. For instance a *100-by-100* sparse matrix with 3% non-zero entries (*NZ=300*) using words of *4* bytes the binary form uses *613* words, as explained in equation (3.21).

$$DIM_{binary} = 100^2 \times 0.03 + \frac{100^2}{8 \times 4} = 612.5 \tag{3.21}$$

3.7 Binary identified format improved

In order to make the method more efficient in practice when density of large sparse matrix is less or equal to 3% another method to store using binary identification has been developed. The difference from the first binary method is how data is organized in the secondary memory zone. The SZ has a half-word structure, containing the column indexes of non-zero entries as well as some control information for a faster identification of non-zero element position in the matrix [1]. The word structure of SZ is shown in the following table.

Table 3.7: SZ for the improved binary format

Word number	**Left half**	**Right half**
1	Row number	Row number
2	Number of non-zero element	
3	Non-zero entries on row no 1	Non-zero entries on row no 2
4	Non-zero entries on row no 3	Non-zero entries on row no 4
...	...	...
k	Non-zero entries on row no n-1	Non-zero entries on row no n
k+1	Column index of 1st non-zero	Column index of 2nd non-zero
k+2	Column index of 3rd non-zero	etc.
...	...	...
j	...	Column index of last non-zero

Table 3.8: Improved binary format SZ for matrix A

Word number	1	2	3		4		5		6	
Value	4	4	1	2	0	1	1	1	2	1

In Table 3.8 it is considered that the non-zero entries are represented on *4 bytes*, therefore a half-word in SZ it is represented on *2* bytes. Using the storing format presented above the maximum matrix size is *9999* rows or columns, with a maximum of $10^8 - 1$ non-zero entries. In case of a square matrix in the first word of SZ the matrix size will be stored.

The formula to calculate the number of words needed for the secondary memory zone is given in (3.22). The value is rounded up to the next integer value.

$$DIM_{SZ} = \frac{5 + n + n^2 \times density(A)}{2} \tag{3.22}$$

The total word count needed for both primary and secondary memory zone is given in (3.23).

$$DIM_{binary2} = \frac{5 + n + 3 \times n^2 \times density(A)}{2} \tag{3.23}$$

For a *100-by-100* sparse matrix with 3% non-zero entries (*NZ=300*) using words of 4 bytes the binary form uses *503* words (3.24), less memory than the first binary identified format which uses *613* words. It can be noticed a decrease of 18% in terms of used memory.

$$DIM_{binary2} = \frac{5 + 100 + 3 \times 100^2 \times 0.03}{2} = 503 \tag{3.24}$$

The division between the updated binary identified format and the standard square matrix format is given by:

$$r_{binary2} = \frac{3 \times density(A)}{2} + \frac{5 + n}{2 \times n^2} \tag{3.25}$$

For a *n-by-n* square sparse matrix, if $r_{binary2} = 1$ and $n \to \infty$, then the maximum value of *density(A)* for the binary identified format is *0.66*:

$$density(A)_{\lim n \to \infty} = \frac{2}{3}(1 - \frac{5 + n}{2 \times n^2}) = 0.666 = 66.6 \tag{3.26}$$

For a sparse square matrix of *100-by-100* dimensions with an average of *66* non-zero entries on each row, the above structure needs a total of *6600+(5+100+6600)/2 = 9952 words*, with *0,6%* less than the *10.000* words needed in the standard format. Because the density of a large sparse matrix varies between 1% and 3% this format is proven to be very efficient regarding the used memory.

3.8 Compact systematic format

This format assumes that the non-zero entries of a sparse matrix are stored in a certain row-major order (see [1]). In this case it is not necessary to store both indexes row and column. For a row-major order it is only needed the column indexes but also the beginning of each row needs to be signaled.

In this case more storing formats can be used, the one presented is characterized by only storing the column index of the needed element and a row separator, chosen arbitrarily to be *-1* as there cannot be such a column. The corresponded value for this inexistent column index represents the row number of the following values. The final value-index is *(0, -1)*.

For a matrix A as in (3.13), this format is described as follows:

Table 3.9: Compact systematic format for matrix A

Value	0	7	1	5	2	3	8	0
Column index	-1	1	-1	1	2	-1	0	-1

For this memory format the maximum words to store a sparse matrix of n-by-n dimensions is calculated with the formula below (3.27). This format is more efficient when compared to the classical format if the density is less than 0.5.

$$DIM_{compact} = 2 \times (n^2 \times density(A) + n + 1) \tag{3.27}$$

Chapter 4

Sparse Matrix Format

Considering all the attempts and approaches to find an efficient memory format for large sparse matrices this note proposes a different representation format, that combines more of the above formats. Next it is detailed a mathematical definition and then implementation of *Sparse Matrix Format*, regarding only square matrices, this will be explained in first part. In the second part we also treat the operations with this new format, also including the mathematics behind it and the implementation. Addressed operations are: multiplication, transposing, summation and difference.

4.1 Mathematical fundamentals

There are two approaches for to represent sparse matrices:

- static approach, in which the memory allocation is done in the compilation phase. This approach assumes that the programmer knows with a good precision the maximum number of non-zero entries
- dynamic approach, in which the memory allocation is done during the execution phase of program. In this case it is not necessary to know the number of non-zero elements. This approach is, consequently, the one that we have implemented.

Usually for identifying the non-zero entries two indices are used, for *row* and for *column*. Firstly, we propose a way to use only one index which contains information for both row and column number (we "linearize" the matrix, converting it to a one dimensional array). For each non-zero entry it is attached an integer number, an *aggregated* index, from which both the row and the column can be determined. For a square matrix A of *n-by-n* dimensions, if entry $A_{ij} \neq 0$, i counting rows and j columns both starting from *0* and ending at *n-1*, then the corresponding aggregated index is calculated using the following formula:

$$index_{Aij} = i \times n + j \tag{4.1}$$

For example *The Sparse Matrix Format* for matrix A as in (4.2) of size *4* is described in Table 4.1. This format stores only the non-zero entries from the vector-index-form of the matrix (see Table 4.2).

$$A_{4*4} = \begin{bmatrix} 0 & 0 & 5 & 0 \\ 0 & 4 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 \end{bmatrix} \; indexes(A) = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 12 & 14 & 15 \end{bmatrix} \tag{4.2}$$

Table 4.1: Sparse Matrix Format

Value	5	4	1	9
Cumulative index	2	5	8	15

Table 4.2: Vector form of Matrix A

Value	0	7	0	0	0	5	2	0	0	0	0	0	8	0	0	0
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The row index for any element of location *k* is calculated as the integer part (*floor* function) of the ratio between the $index_{Aij}$ and *n*, size of the matrix, as showed below:

$$row_{Aij} = \left\lfloor \frac{index_{Aij}}{n} \right\rfloor = floor(\frac{index_{Aij}}{n}). \tag{4.3}$$

Where the *floor function* takes as an input a real number *x* and gives as output the greatest integer less than or equal to *x*. For example *floor(3,4) = 3*.

The column index for any element of location *k* is calculated as the remainder of the division between the $index_{Aij}$ and *n*, also known as the *modulo* function as showed in (4.4).

$$col_{Aij} = index_{Aij} - \left\lfloor \frac{index_{Aij}}{n} \right\rfloor = index_{Aij} \text{ modulo } n \tag{4.4}$$

The advantage of this structure is that it uses only one index for each non-zero entry. On the other hand there are two operations executed in order to find the row and column indexes.

The total number of words needed for this format to be stored in the memory can be calculated using the formula (4.5), while the total number of words to store SMF on the disk is a little bit different and it is calculated using (4.6).

$$DIM_{SMFmem} = 2 \times n^2 \times density(A) + n \tag{4.5}$$

$$DIM_{SMFdisk} = 2 \times n^2 \times density(A) + 1 \tag{4.6}$$

The division between the memory requirements of SMF and the classical format is:

$$r_{SMF} = 2 \times density(A) + \frac{1}{n^2} \tag{4.7}$$

The upper limit for the density of a matrix for which this format is still memory efficient is *density(A) = 0.5*.

4.2 Implementation of Sparse Matrix Format

Considering a matrix M as in (4.8), the SMF implementation looks like (4.9). Therein, the notation *[p]* is for pointer to another memory zone. Each row is a dynamically adjustable array of tuples.

$$M_{4*4} = \begin{bmatrix} 1 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 8 & 0 & 0 & 4 \end{bmatrix} \tag{4.8}$$

$$\begin{aligned}(2, [p]) &\rightarrow [(1, 0), (5, 2)]\\ (0, [p]) &\rightarrow \text{NULL}\\ (1, [p]) &\rightarrow [(2, 2)]\\ (2, [p]) &\rightarrow [(8, 0), (4, 3)]\end{aligned} \tag{4.9}$$

For a more modular implementation there were defined classes used for storing data and operations on data:

- entry class
  - value: non-zero entry
  - column index
- SMF class
  - array of pointers to entry vectors
  - array of lengths of entry vectors

```
class entry {
    public:
        value_type value;
        size_type column;
    ...
}

class SMF {
    public:
        index_type matrix_size;
        entry** rows;
        size_type* row_len;
        index_type nnz = 0;
    ...
}
```

4.3 From Standard to Sparse Matrix Format

The function *StandardtoSMF* is used to transform the Standard format to SMF. It is a member function of the SMF class and it takes as an argument a matrix as a pointer to pointer to ***value_type***. In order to find the non-zero entries of the standard matrix it is necessary to iterate through all the elements. Once a non-zero entry is found it is inserted in the SMF format along with the aggregated index. Here is the implementation of such function:

```
// creating a SMF from standard
void StandardtoSMF(value_type** Standard){

    //iterate through rows
    for (value_type i = 0; i < matrix_size; ++i)

        //iterate through columns
        for (value_type j = 0; j < matrix_size; ++j)

            //if entry is non-zero
            if (Standard[i][j] != 0)

                //insert to SMF value and aggregated index
                this->insert(Standard[i][j], i*matrix_size + j);
}
```

4.4 From Sparse Matrix Format to Standard

The function *SMFtoStandard* is used to make the translation between the two formats. As SMF is implemented to only store each non-zero entry of the matrix and the composed index, for each SMF entry the column and row have to be determined. Once this operations are done the element is ready to take its position in the standard matrix. The implementation looks like:

```
1 // creating a square matrix_size matrix from SMF
2 value_type** SMFtoStandard(){
3     value_type **Standard, val;
4     index_type col;
5
6     //allocating dynamic array (of size=matrix_size)
7     //of pointers to element type (value_type)
8     //initializing all to 0
9     Standard = new value_type*[matrix_size]();
10
11    //allocate each row
12    for(index_type i = 0; i < matrix_size; ++i)
13        Standard[i] = new value_type[matrix_size];
14        // each i-th pointer is now pointing to dynamic array
15        // (size matrix_size) of actual value_type values
16
17    //iterating through rows
18    for(index_type i = 0; i < matrix_size; ++i)
19        //iterating through elements
20        for(index_type j = 0; j < row_len[i]; ++j){
21            //get col index
22            col = rows[i][j].getC();
23
24            //get value of entry
25            val = rows[i][j].getV();
26
27            //store the value at the precise indexes
28            Standard[i][col] = val;
29        }
30
31    return Standard;
32 }
```

Chapter 5

Operations with Sparse Matrix Format

In order to make the SMF a viable replacement for the standard matrix format, operations with it have to be defined. As stated before, the second objective here is to mathematically define and implement a solution for the associated operations of SMF. Operations considered to be implemented: multiplication, transposing, summation and subtraction.

5.1 Summation of two SMF matrices

The summation algorithm described below is implemented as a member function in the SMF class. It uses two position pointers to iterate through A's row respectively B's row called *apos* and *bpos*:

1. For each row of both matrices (first *while* loop).
2. Reinitialize *apos* and *bpos* to 0 and get row length in *len_rowA* and *len_rowB*.
3. While the pointers did not reach the end of the row (second *while* loop).
4. Get the 2nd element of tuple entry, the column index in *col_A* and *col_B*.
5. By comparing the column index it is decided if entries must be inserted individually in the result or added and then inserted in the result.
6. The remaining elements from A's or B's row are inserted (3rd and 4th *while* loops).

```
1  SMF add(SMF B){
2
3      SMF rez(matrix_size);
4
5      index_type apos, bpos;
6      index_type i = 0;
7
8      //same row for both matrices
9      while(i < matrix_size){
10         apos=0; bpos=0;
11
12         //get row A & B len
13         size_type len_rowA = row_len[i];
14         size_type len_rowB = B.row_len[i];
15
16         while (apos < len_rowA && bpos < len_rowB){
17
18             //get A's col
19             size_type A_col = rows[i][apos].getC();
20             //get B's col
```

```
21              size_type B_col = B.rows[i][bpos].getC();
22
23              //if B's col index is smaller than A's col index
24              if(B_col < A_col){
25
26                  //insert B's val & calc index
27                  rez.insert(B.rows[i][bpos].getV(),  i*matrix_size +
                        B_col);
28                  bpos++;
29
30              //else if A's col is smaller than B's col
31              } else if(B_col > A_col){
32
33                  //insert A's val & calc index
34                  rez.insert(rows[i][apos].getV(),
35                  i*matrix_size + A_col);
36                  apos++;
37
38              //else same col -> add them
39              } else {
40
41                  rez.insert(rows[i][apos].getV()+
42                  B.rows[i][bpos].getV(),
43                  i * matrix_size + A_col);
44                  apos++;
45                  bpos++;
46              }
47          }
48
49          //insert ramaining el from A
50          while(apos < len_rowA){
51
52              rez.insert(rows[i][apos].getV(),
53              i * matrix_size + rows[i][apos].getC());
54              apos++;
55          }
56
57          //insert ramaining el from B
58          while(bpos < len_rowB){
59
60              rez.insert(B.rows[i][bpos].getV(),
61              i * matrix_size + B.rows[i][bpos].getC());
62              bpos++;
63          }
64          i++;
65      }
66      return rez;
67  }
```

5.2 Subtraction of two SMF matrices

The subtraction algorithm described below is implemented as a member function in the SMF class. It is very similar with the summation algorithm with the difference that if the column indexes are different insert the inverse number (*0 - val*) in the result and if there are equal subtraction of the two is done instead of summation.

1. For each row of both matrices.
2. Reinitialize *apos* and *bpos* to 0 and get row length in *len_rowA* and *len_rowB*.
3. While the pointers did not reach the end of the row.

4. Get the 2nd element of tuple entry, the column index in *col_A* and *col_B*.
5. By comparing the column index it is decided if the inverse number (*0 - val*) must be inserted individually in the result or subtracted and then inserted in the result.
6. The remaining elements from A's or B's row are inverted and inserted.

5.3 Transposing the SMF

The transposing algorithm is simply iterating through all rows, then through entries and inserting to the result matrix the calculated aggregated index of the transpose:

1. For each row (the first *for* loop).
2. For each entry in the row (the 2nd *for* loop).
3. Get the value and column index of the entry.
4. Insert to the result matrix the transposed aggregated index.

```
SMF transpose(){

    SMF rez(matrix_size);
    value_type val;
    index_type col;

    //iterating through rows
    for(index_type i = 0; i < matrix_size; ++i)

        //iterating through elements
        for(index_type j = 0; j < row_len[i]; ++j){

            //get col index
            col = rows[i][j].getC();

            //get value of entry
            val = rows[i][j].getV();

            //insert to rez the new index for this value
            rez.insert(val, col * matrix_size + i);
    }

    return rez;
}
```

5.4 Multiplication of two SMF matrices

The multiplication algorithm is one key operation when dealing with matrices. Considering two matrices *A* and *B* that can be multiplied, a short description of the algorithm is below:

1. First transpose *B* for an easier iteration through columns, as SMF is row-major order, we need *B* in column-major order.
2. Iterate through rows of *A* (first *for* loop).
3. For each row in *A*, iterate through columns of *B*, actually rows of B transpose (second *for* loop).
4. Reinitialize the local pointer of A's row and B's column and the sum to 0.

5. While these pointers are within the range of A's row length respectively B's column length.
6. Compare A's column with B's row and skip elements in A's row respectively B's column until they are equal.
7. If A's column is equal to B's row do the multiplication and add to sum.
8. If sum is different than 0 add it to result matrix with the aggregated index.

```
SMF multiply(SMF B){

    SMF rez(matrix_size);

    index_type col_A, col_B, apos, bpos;
    value_type sum;

    SMF Bt = B.transpose();

    //iterating through rows of A
    for(index_type i = 0; i < matrix_size; ++i){

        //iterating through cols of B
        //as B is transpose iterate through rows
        for(index_type j = 0; j < matrix_size; ++j){

            //local pointer within A's row
            apos = 0;
            //local pointer within B's col
            bpos = 0;
            //sum of multiplication
            sum = 0;

            //iterating through elements of A's row & B's col
            while(apos < row_len[i] && bpos < Bt.row_len[j]){

                //get col index of A's entry
                col_A = rows[i][apos].getC();
                //get col index of B's entry
                col_B = Bt.rows[j][bpos].getC();

                //if A's col is smaller than B's row
                //skip entry in A
                if(col_A < col_B){
                    apos ++;

                //if B's row is smaller than A's col
                //skip entry in B
                } else if (col_A > col_B){
                    bpos++;

                //else both row and col are equal
                //multiply the entries and add to sum
                } else {
                    sum += rows[i][apos].getV() *
                    Bt.rows[j][bpos].getV();
                    apos++;
                    bpos++;
                }
            }

        //if the sum is non-zero add to rezult
        if (sum != 0)
            rez.insert(sum, i * matrix_size + j);
        }
    }
```

```
57      return rez;
58  }
```

Chapter 6

Efficiency of Sparse Matrix Format

Validation of the Sparse Matrix Format is done by calculating the memory and computational efficiency with respect to the classical format and algorithms. Test data that will be analyzed has two collection sources: *The University of Florida Sparse Matrix Set* taken from [8] and *Anonymous MRI Brain Scan Images Database (University of Granada)* taken from public sources.

This chapter will discuss the memory efficiency in two cases: when storing the matrix on disk and when manipulating the data in algorithms, as they are slightly different. In the second case a few more information are needed in order to easily access and manipulate the data. Besides the use of memory point of view, we also analyzed the computational efficiency of the *Sparse Matrix Format*.

6.1 The working sets

The University of Florida Sparse Matrix Collection

We will give a short description of this source. *The SuitSparse Matrix Collection* also known as *The University of Florida Sparse Matrix Collection* is a huge and continuously growing database of sparse matrices that are encountered in real applications. This collection is intensively used by the numerical linear algebra community for performance evaluation of sparse matrix algorithms. Following [8], the collection covers a large number of domains, divided in two classes, these are:

- matrices resulting from problems with a 2D or 3D geometrical representation:
  - structural engineering
  - computational fluid dynamics
  - model reduction
  - electromagnetics
  - semiconductor devices
  - thermodynamics
  - computer graphics/vision
  - robotics/kinematics
- matrices without geometrical source interpretation:

- optimization
- circuit simulation
- economic and financial modeling
- theoretical and quantum chemistry
- chemical process simulation
- mathematics and statistics
- power networks

From the *SuitSparse Matrix Collection* (see [8]) were chosen in a random manner 97 matrices. Size of this working set varies between *5* to *5000* (see Figures 6.2 and 6.1). It is worth mentioning that in practice the size can reach even 200.000 in which case *SMF* has even higher efficiency.

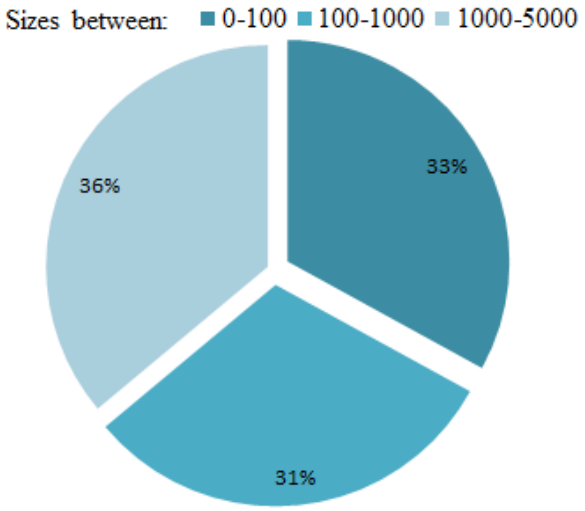


Figure 6.1: Intervals of sizes

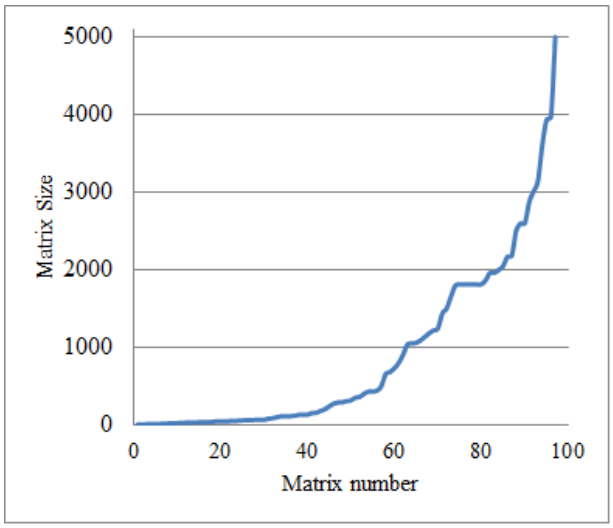


Figure 6.2: Matrix size range

Graph (6.2) plots the matrix size against the sample number, an easy way to visualize the number of matrices in each interval of size ranges.

From practical experiments the matrix density decreases as the matrix size increases. Figure 6.3 represents the size-density distribution. It can be easily notice how the left half of the chart with sizes smaller than *500* have a higher density in comparison with the right half in which, with a few exceptions, the density values are between 0 and 3%. Figure 6.3 also displays a trend line approximation of the data. It has been used a 5^{th} degree polynomial equation to approximate. For a graph with the matrices above the size of *500* with a linear trend see Figure 6.4.

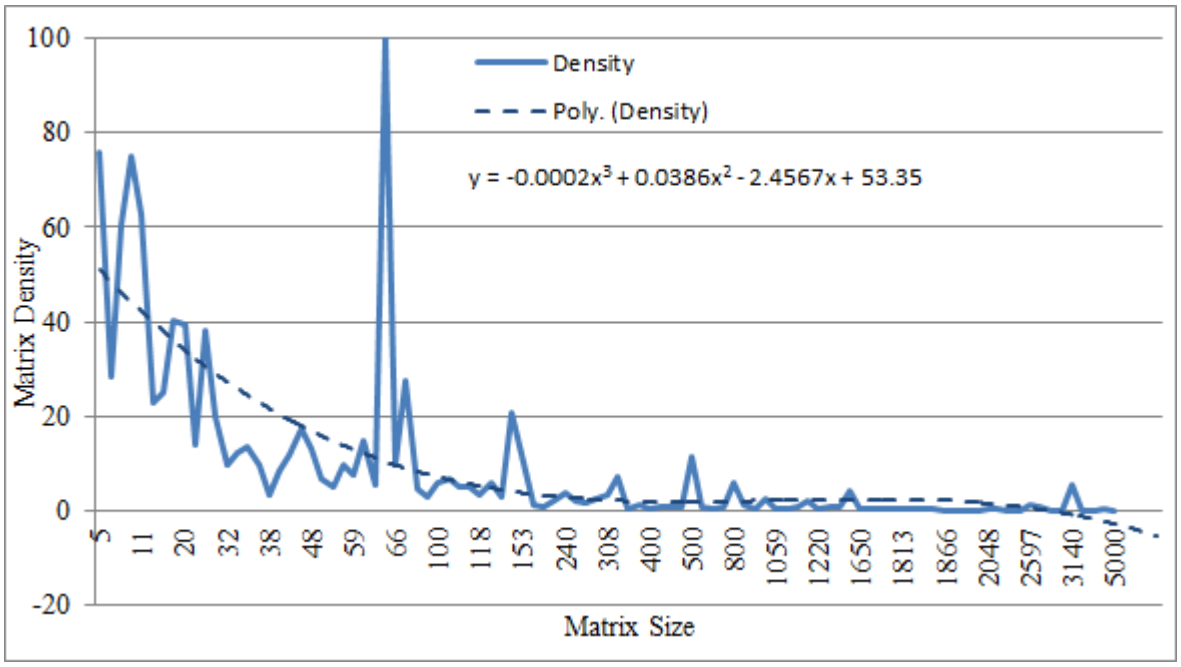


Figure 6.3: Matrix size-density distribution

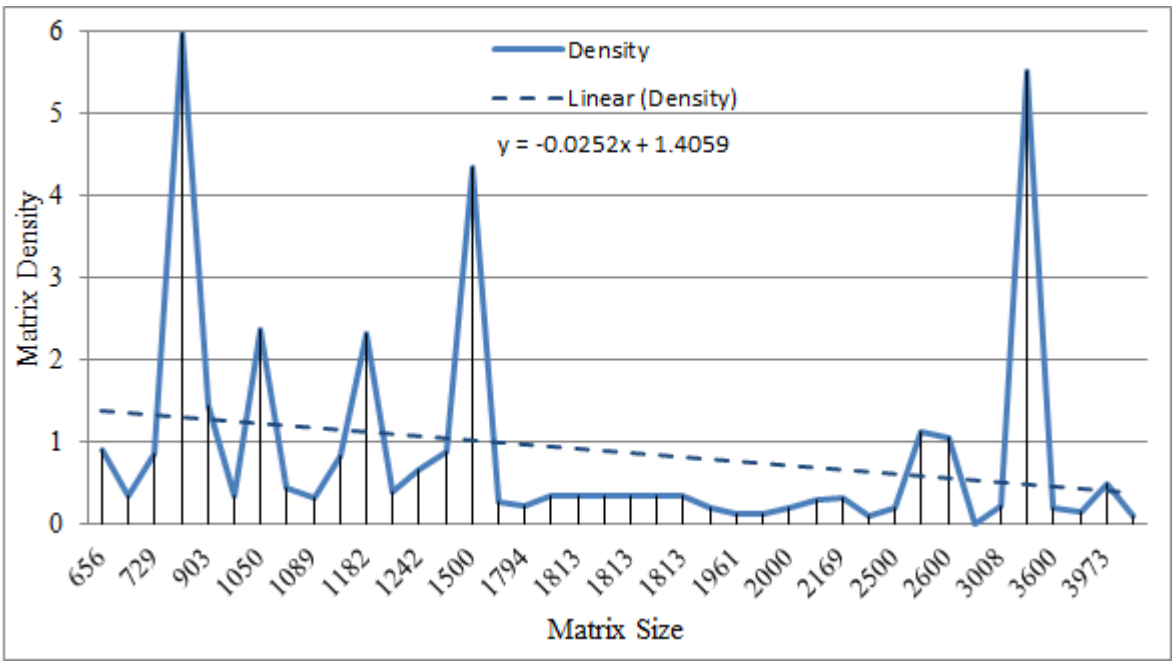


Figure 6.4: Zoomed matrix size-density distribution

Anonymous MRI Brain Scan Images Database

A second working set is *Anonymous MRI (Magnetic Resonance Imaging) Brain Scan Images Database*. But first it is needed to ensure that the representation of these images is a sparse matrix. For instance the image in Figure 6.5 can be loaded as a pixel-value 2D array. A gray-scale image has the pixel value represented on 8 bits:

- for white the pixel has a value of 255 (all bits are 1),
- for shades of gray the pixel has a value within the interval [254, 1],
- for black the value is 0.

As it can easily be noticed how the image below has most of the pixels black, so it can be considered a sparse matrix of pixels. The density of this image is among the higher that it can be in a MRI scan and it has a value of 25%. Therefore the compressed *SMF* brings a memory improvement.

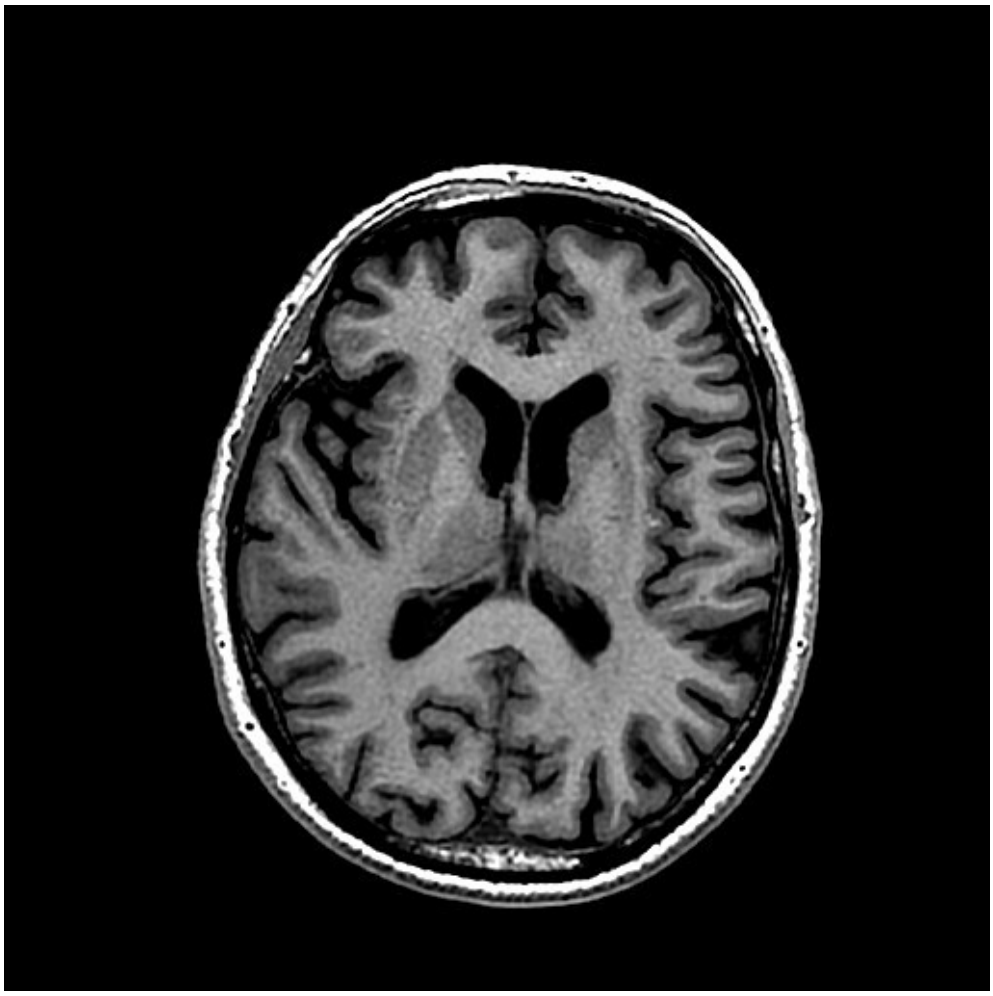
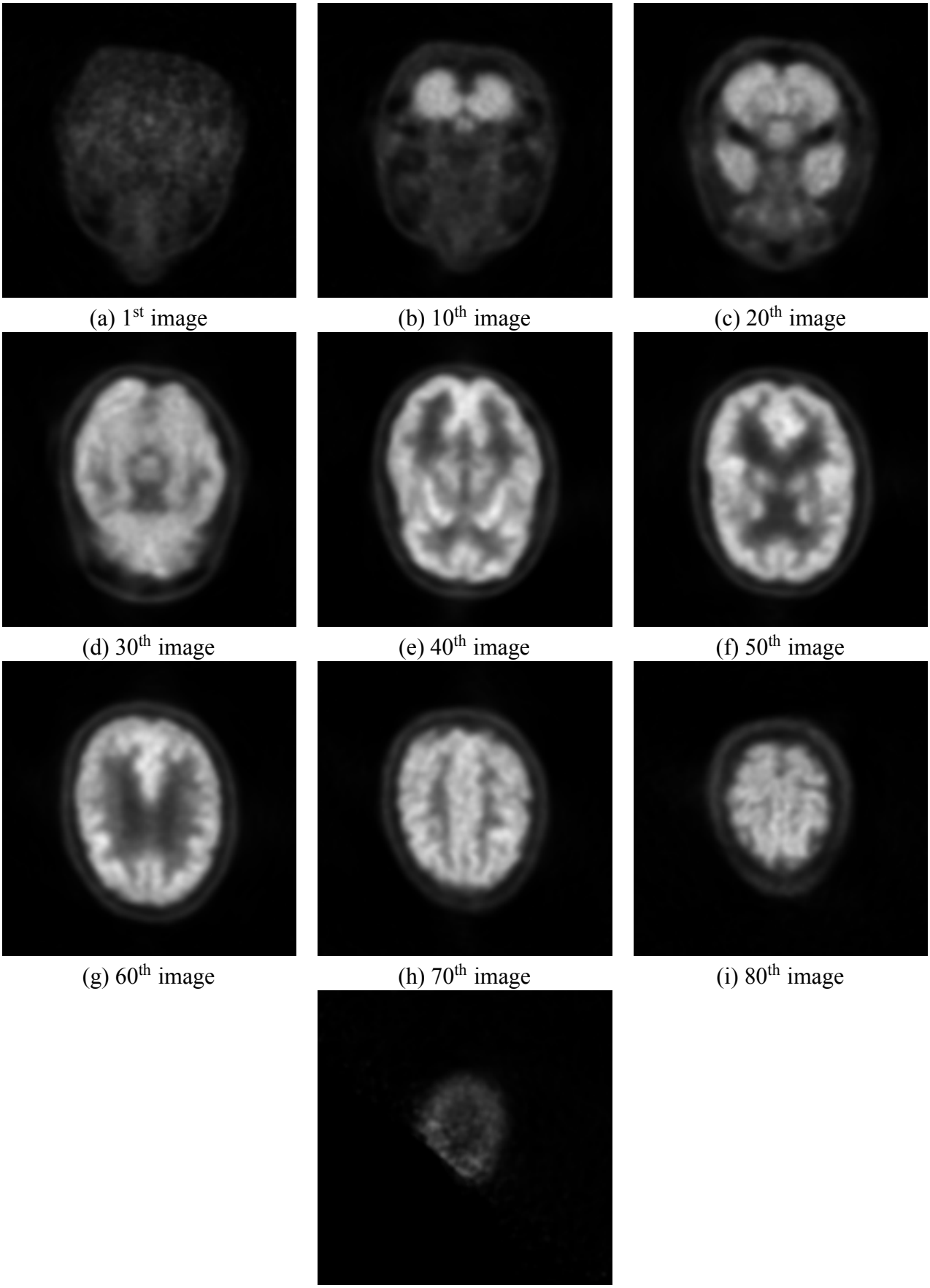


Figure 6.5: Brain MRI Example

Let us explain how a *MRI brain scan* is produced. A full MRI brain scan contains between 90 and 170 images. The scan is done as sections from one extremity of the head to the other, density of these images varies between 0% and 25% as it is illustrated in Figure 6.6. Below is showed every other 10^{th} image of a MRI scan containing 95 images.



(a) 1st image (b) 10th image (c) 20th image

(d) 30th image (e) 40th image (f) 50th image

(g) 60th image (h) 70th image (i) 80th image

(j) Last image (95th)

Figure 6.6: Samples from a full MRI scan

The best approximation of such evolution is a 2nd degree parabola as showed in Figure 6.7. The turning point of this equation is the image with the most information corresponding to the middle section of the head (Figure (6.6) between (e) and (f)). By plotting the sparsity - memory efficiency tuples it is observed in Figure 6.8 the linear growth of the memory efficiency proportional to the sparsity.

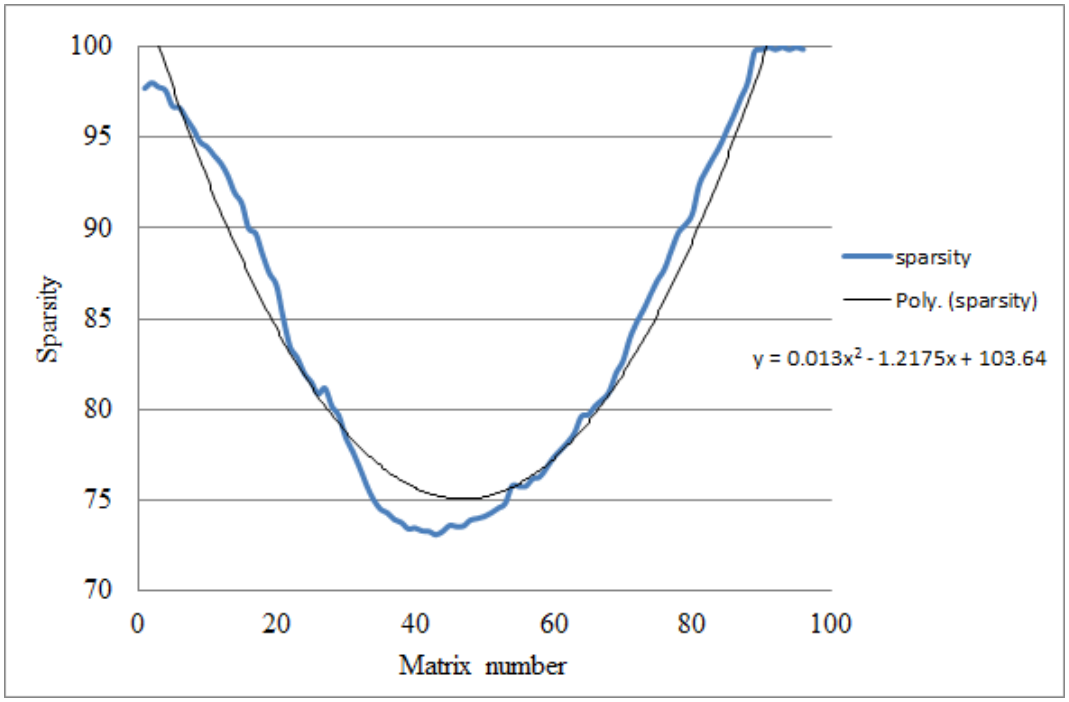


Figure 6.7: Sparsity of the MRI Brain Example

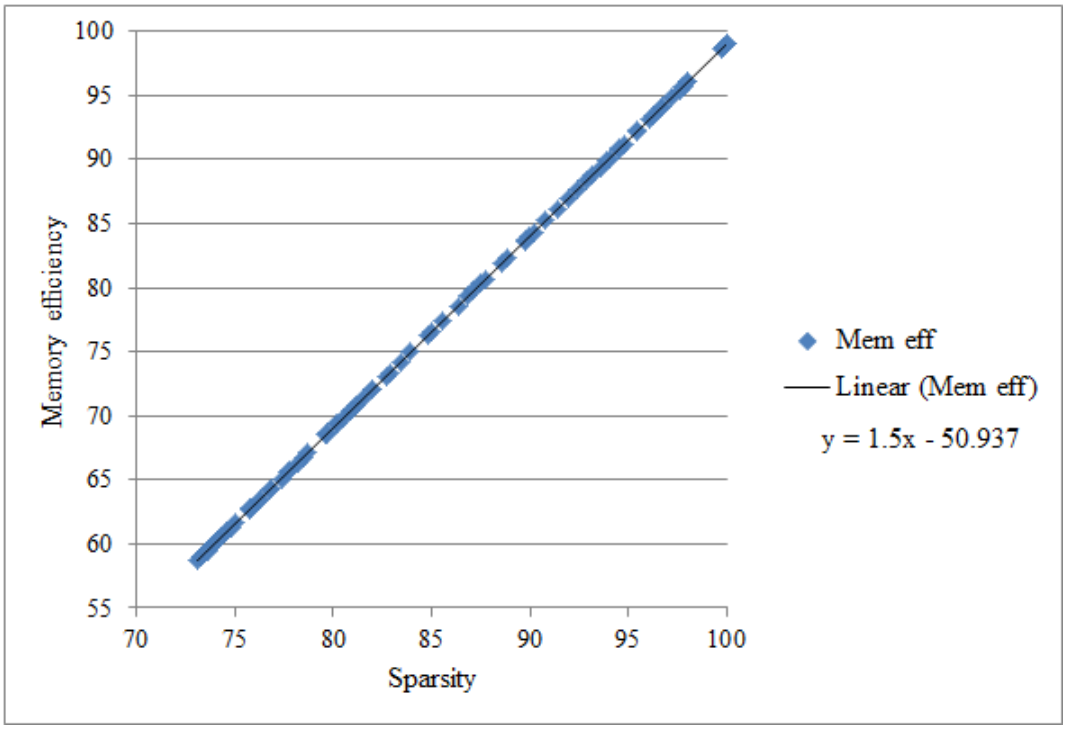


Figure 6.8: Memory efficiency of the MRI Eexample

6.2 Storing sparse matrices on the file system

As the sparse matrices can sometimes reach enormous sizes a method to store these matrices on the file system is needed. As described before, the SMF will be used, meaning that only the matrix size and the value-index tuples will be stored.

Table 6.1: Matrix in the standard format (left) and the Sparse Matrix Format (right)

10										10											
1	0	0	1	0	0	1	0	0	1	1	0	1	3	1	6	1	9				
1	0	0	1	0	0	1	0	0	1	1	10	1	13	1	16	1	19				
1	1	1	1	0	0	1	0	0	1	1	20	1	21	1	22	1	23	1	26	1	29
0	1	1	1	0	0	1	0	0	1	1	31	1	32	1	33	1	36	1	39		
0	0	1	1	0	0	1	0	0	1	1	42	1	43	1	46	1	49				
0	0	0	1	1	1	1	0	0	1	1	53	1	54	1	55	1	56	1	59		
0	0	0	0	1	1	1	0	0	1	1	64	1	65	1	66	1	69				
0	0	0	0	0	1	1	1	1	1	1	75	1	76	1	77	1	78	1	79		
0	0	0	0	0	0	0	1	1	1	1	87	1	88	1	89						
0	0	0	0	0	0	0	0	1	1	1	98	1	99								

Case study - large square matrix from the SuitSparse Matrix Collection

In order to study the efficiency of SMF when storing a big square sparse matrix on the disk, matrix A, with size of 20685 was chosen from the *SuitSparse Matrix Collection*. It describes a *Structural Problem* system and it was published by Christian Damhaug (Oslo, Norway) in 2004 [8]. The matrix contains only binary values and it has a perfect *pattern and value symmetry* (see Figure 6.9). It has *2.454.957* non-zero values and a density of *0.5738*% according to equation (6.1).

$$density(M) = \frac{NZ}{n^2} \times 100 = \frac{2.454.957}{20.685^2} \times 100 = 0.573763\% \tag{6.1}$$

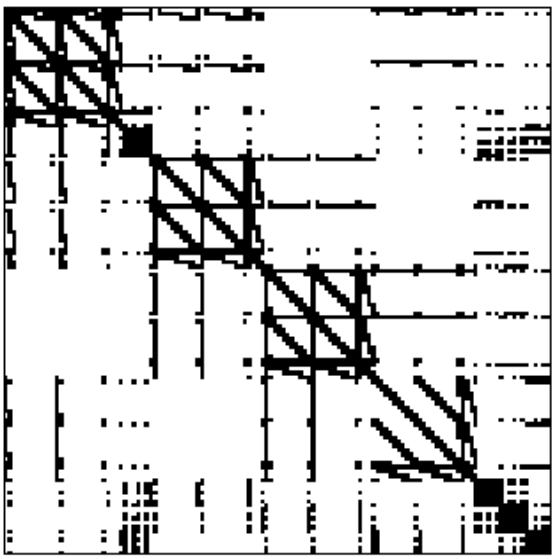


Figure 6.9: Pattern of the matrix A

The plain text file containing the above matrix on disk in the standard format (Table 6.1 - left) takes **816MB** while the plain text file with the SMF style (Table 6.1 - right) of the same matrix occupies only **27,4MB**. The second format represents an important improvement, taking 96.64% less space on the disk when compared to the standard storing format. The improvement has been determined according to equation (6.2).

$$improvement = 100 - \frac{27.4 \times 100}{816} = 96.64\% \tag{6.2}$$

6.3 SMF data structure implementation memory efficiency

Working set 1 - The University of Florida Collection

From the representation of the *SMF* memory efficiency against density (Figure 6.10) can easily be noticed that the smaller density has the higher memory efficiency. For densities between *100%* and *40%*, which belong to small size matrices, the memory efficiency figures are negative, meaning there is no improvement, on contrary the *SMF* representation for such matrices occupies more memory. However in practice the densities are well below the *40%* turning point that we can notice in the graph, therefore the efficiency varies between *30%* and *99.8%*.

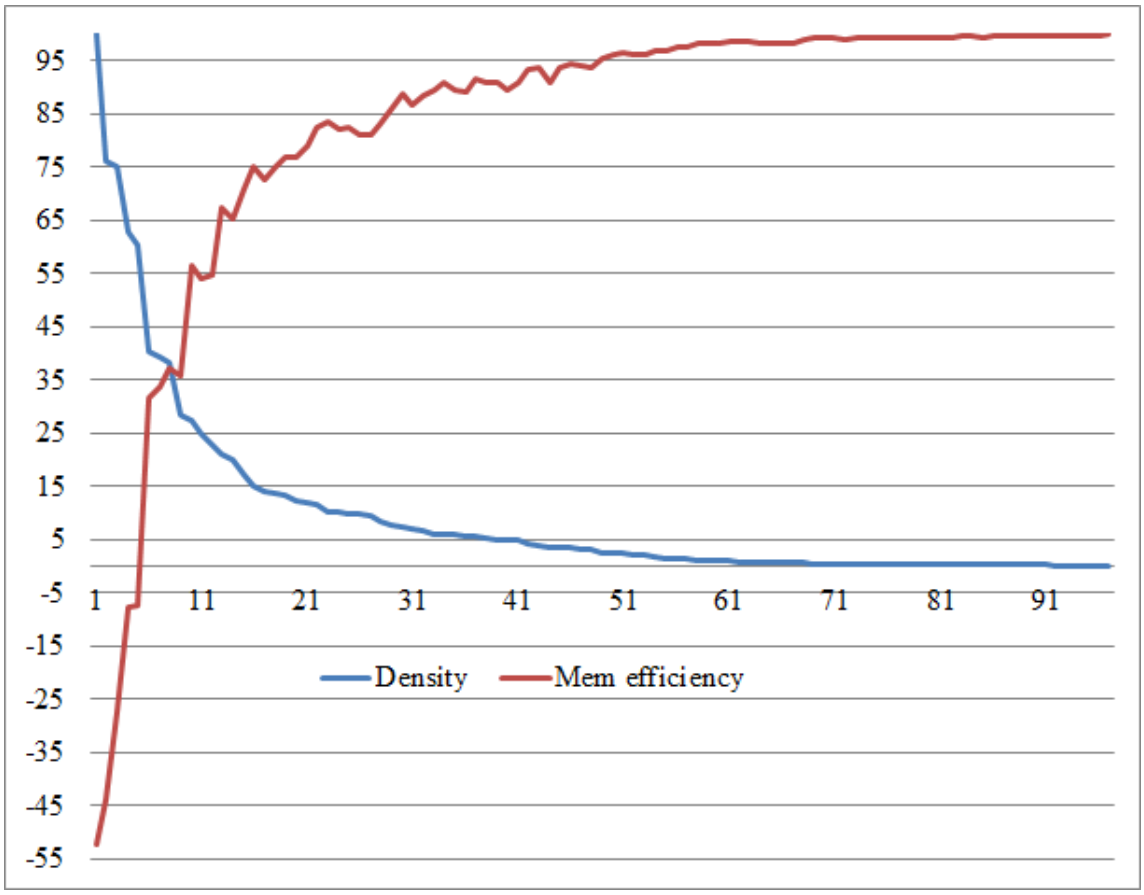


Figure 6.10: Memory efficiency against density

The total required memory for the new SMF data structure is calculated in several steps:

1. For each row a pointer to an array of entries and the length of the row are stored.
2. For each non-zero value it is used an entry structure (value and index), add to the total bytes number *sizeof(entry)*NZ*.

```
index_type bytes_no(){
    index_type rez = 0;

    //for each row pointer to row and length = NZ for the row
    rez += (sizeof(entry*) + sizeof(size_type)) * matrix_size;

    //for each non-zero it is used an entry
    rez += sizeof(entry) * NZ;

    return rez;
}
```

For the *SuitSparse* working set the disk occupancy efficiency improvement in terms of percentages calculated using (6.2) plotted against the matrix size clearly shows, with a very few exceptions of small size matrices, that *SMF* is very efficient compared to the standard format (Figure 6.11). Both *Sparse Matrix Format* and standard are storing data on the disk in plain text. Dividing the working set into three categories by size, a trend in improvement is also observed. All of the categories have some exceptions, the three cases are:

1. small size matrices: for n between *0* and *100* the improvement varies in the interval of *40%-80%*.

2. medium size matrices: for n between *100* and *1000* the improvement has higher percentages, it varies in the interval of *80%-95%*.

3. big size matrices: for n between *1000* and *5000* the improvement has the highest percentages, it varies in the interval of *95%-99.5%*.

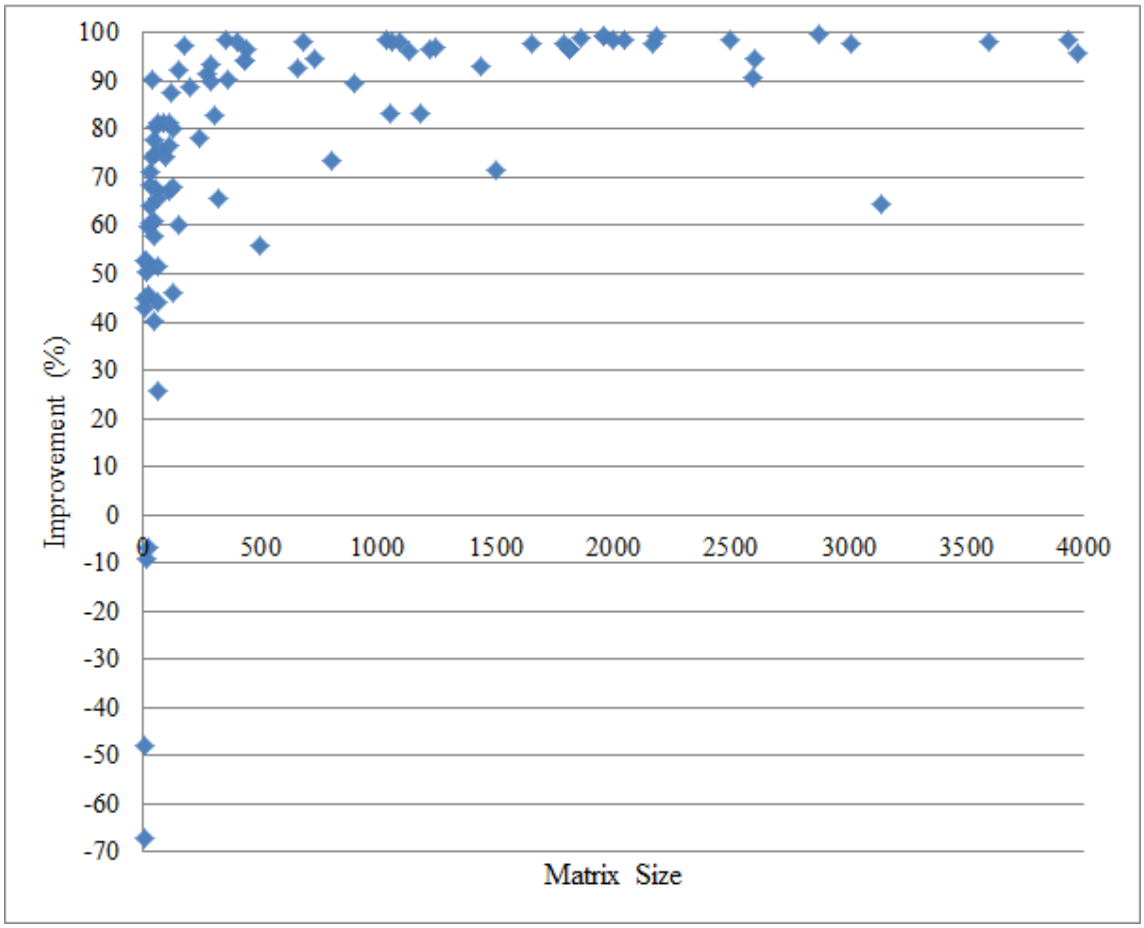


Figure 6.11: Memory improvement against matrix size

Working set 2 - Anonymous MRI Brain Scan Images Database

In Figure (6.12) it is compared the disk space needed for the standard way of storing matrices in plain text with the new *Sparse Matrix Format*. It can be easily observed how for the beginning and end image sections of the human head MRI scan the *SMF* is more efficient when compared with the middle section images where the information represents up to *25%* of the image so less sparse. For this full MRI scan example the total disk space taken by the 90 images in standard format is *8275 KB* compared with the *SMF* which takes *6596 KB*, representing *79%* of the original space on disk, so an improvement of *21% (1679KB)* for each scan. Scaling this percentage up to the whole database the disk savings are considerable.

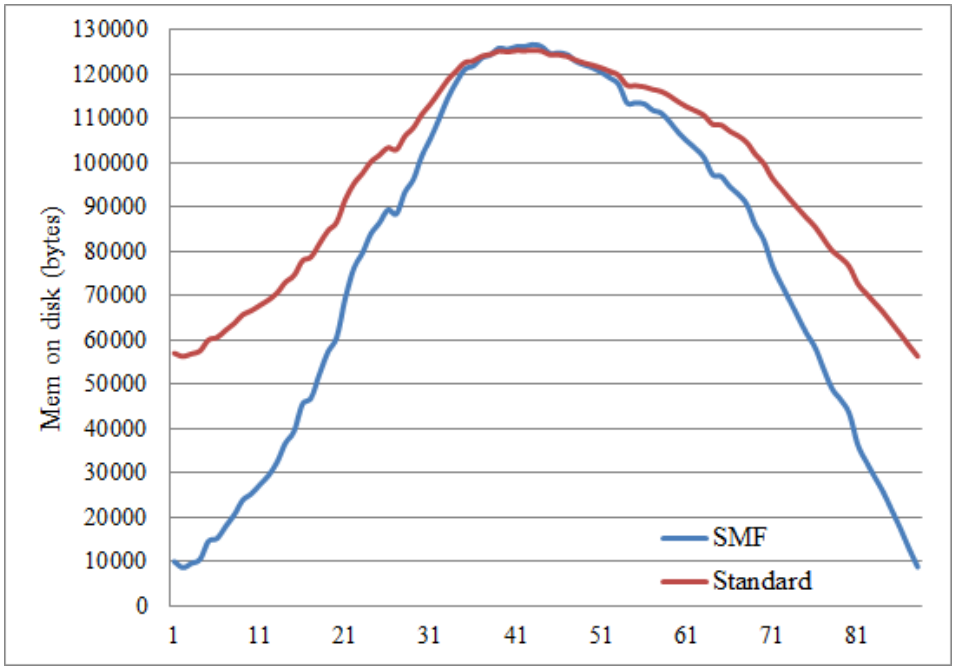


Figure 6.12: Memory on disk SMF and standard

Figure (6.13) represents the difference between *standard* and *SMF*. For instance, considering one of the images at the beginning of the MRI, say the 5th one, the difference is of *45.000 bytes*. In contrast, considering one of the images closer to the middle, the 37th one, the difference is of only *90 bytes*, which is insignificant. The difference is negative for a few matrices with a higher density for which the standard matrix arrangement is memory efficient.

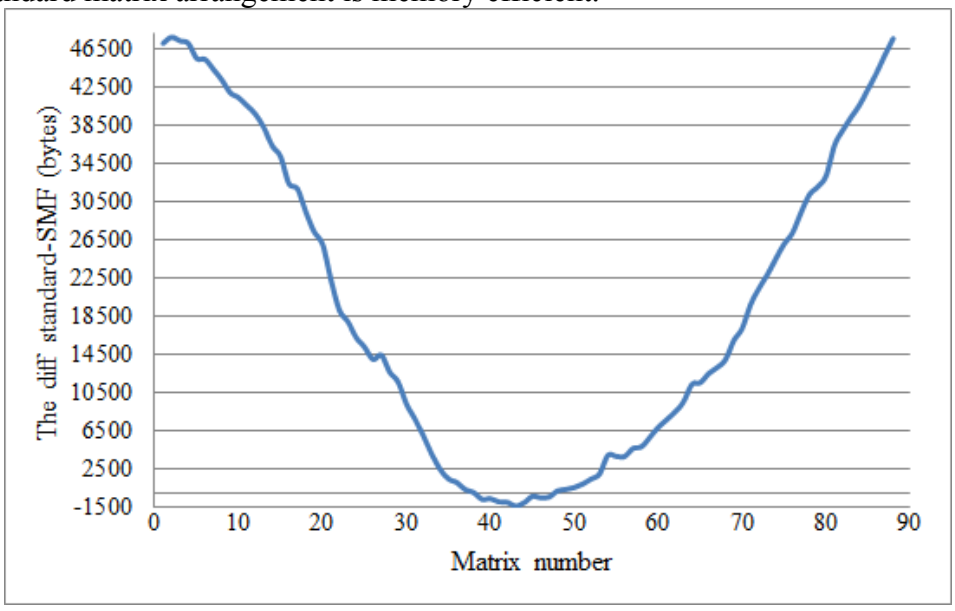


Figure 6.13: The difference between standard and SMF in Bytes

6.4 Computational efficiency

Computational efficiency is proven comparing the classical format multiplication algorithm with the *SMF* multiplication algorithm. For a better understanding of this operation let us consider two square matrices A and B of size 4 to be multiplied with the result matrix C, as in (6.3). These matrices are first written in the vector form (Table 6.2) and then put in a table in order to see the multiplications that have to be done between the matrices entries (Table (6.2)).

$$A_{4*4} = \begin{bmatrix} 2 & 0 & 0 & 3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 7 & 5 & 0 \end{bmatrix} B_{4*4} = \begin{bmatrix} 0 & 0 & 0 & 4 \\ 5 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 5 & 0 & 0 \end{bmatrix} C_{4*4} = A * B = \begin{bmatrix} 0 & 15 & 0 & 8 \\ 5 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \\ 35 & 10 & 7 & 0 \end{bmatrix} \tag{6.3}$$

Table 6.2: Vector format of matrices A and B

A $_{4x4}$ =	**Value**	2	0	0	3	0	1	0	0	0	0	0	0	0	7	5	0
	Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B $_{4x4}$ =	**Value**	0	0	0	4	5	0	1	0	0	2	0	0	0	5	0	0
	Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Table 6.3: Matrices A and B in the Sparse Matrix Format

A				B	
4				4	
2	0	3	3	4	3
1	5			5	4
				2	9
7	13	5	14	5	13

Table 6.4: Matrix multiplication in vector format

Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	**block**
	Value	0	0	0	4	5	0	1	0	0	2	0	0	0	5	0	0	
0	2	0	0	0	8													0
1	0					0	0	0	0									
2	0									0	0	0	0					
3	3													0	15	0	0	
4	0	0	0	0	0													1
5	1					5	0	1	0									
6	0									0	0	0	0					
7	0													0	0	0	0	
8	0	0	0	0	0													2
9	0					0	0	0	0									
10	0									0	0	0	0					
11	0													0	0	0	0	
12	0	0	0	0	0													3
13	7					35	0	7	0									
14	5									0	10	0	0					
15	0													0	0	0	0	

As the extended algorithm is very large when put in the table form of (6.4), it can be reduced to Table (6.5) and the content being unchanged. Furthermore, it is possible to write only the lines that have either the entry of matrix A or B non-zero like in Table (6.6) and still organizing it in blocks.

Table 6.5: Matrix multiplication in vector format

Index	**A**					**B**	**block**
0	2	0	0	0	8	0	
1	0	0	0	0	0	0	
2	0	0	0	0	0	0	0
3	3	0	15	0	0	4	
		0	**15**	**0**	**8**		
4	0	0	0	0	0	5	
5	1	5	0	0	0	0	
6	0	0	0	0	0	1	1
7	0	0	0	0	0	0	
		5	**0**	**1**	**0**		
8	0	0	0	0	0	0	
9	0	0	0	0	0	2	
10	0	0	0	0	0	0	2
11	0	0	0	0	0	0	
		0	**0**	**0**	**0**		
12	0	0	0	0	0	0	
13	7	35	0	7	0	5	
14	5	0	10	0	0	0	3
15	0	0	0	0	0	0	
		35	**10**	**7**	**0**		

Table 6.6: Compact matrix multiplication in vector format

Index	**A**					**B**	**block**
0	2	0	0	0	8	0	
3	3	0	15	0	0	4	0
		^{0}**0**	^{1}**15**	^{2}**0**	^{3}**8**		
4	0	0	0	0	0	5	
5	1	5	0	0	0	0	
6	0	0	0	0	0	1	1
		^{4}**5**	^{5}**0**	^{6}**1**	^{7}**3**		
9	0	0	0	0	0	2	
		^{8}**0**	^{9}**0**	^{10}**0**	^{11}**0**		2
13	7	35	0	7	0	5	
14	5	0	10	0	0	0	3
		^{12}**35**	^{13}**10**	^{14}**7**	^{15}**0**		

The result matrix is obtained by summing the blocks on columns, each row in bold is part of the result matrix C:

$$C_{4*4} = \begin{bmatrix} \mathbf{0} & \mathbf{15} & \mathbf{0} & \mathbf{8} \\ \mathbf{5} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{2} & \mathbf{0} & \mathbf{0} \\ \mathbf{35} & \mathbf{10} & \mathbf{7} & \mathbf{0} \end{bmatrix}, \tag{6.4}$$

with the *Sparse Matrix Format*:

Table 6.7: Result matrix C in the Sparse Matrix Format

C					
4					
15 5 2 35	1 4 9 12	8 1 10	3 6 13	 7	 14

A script written in Python was used to generate this simplified multiplication algorithm and compared with the classical matrix multiplication algorithm.

Listing 6.1: Script generator

```
def new_mult_generator(n):
    program = 'C = [0] * (%d*%d)' % (n, n);

    for i in range(0, n*n):
        block = int(i / n)
        program += '''
if A[%d] != 0: ''' % i
        for k in range(0, n):
        b_indx = int((i * n + k) % (n*n))
        program += '''
    if B[%d] != 0: C[%d] += A[%d] * B[%d]''' % (b_indx, ((b_indx % n) +
        (block * (n))), i, b_indx)

    return program
```

In Figure (6.14) it is presented the time evolution of the classical multiplication algorithm versus the new scripting method. The matrices used to make this comparison have a *20%* density and vary in size between *30* and *90*. For each matrix the operation was repeated *500* times in order to obtain a more precise average time. As it can be noticed, the standard multiplication algorithm has an exponential trend, while the new method can be better approximated with a linear growth. For Small size matrices the difference is insignificant while for bigger sizes the same multiplication repeated 500 times results in a huge difference. In practice this is the most encountered case, repeated operation with the same type of matrix and same particularities.

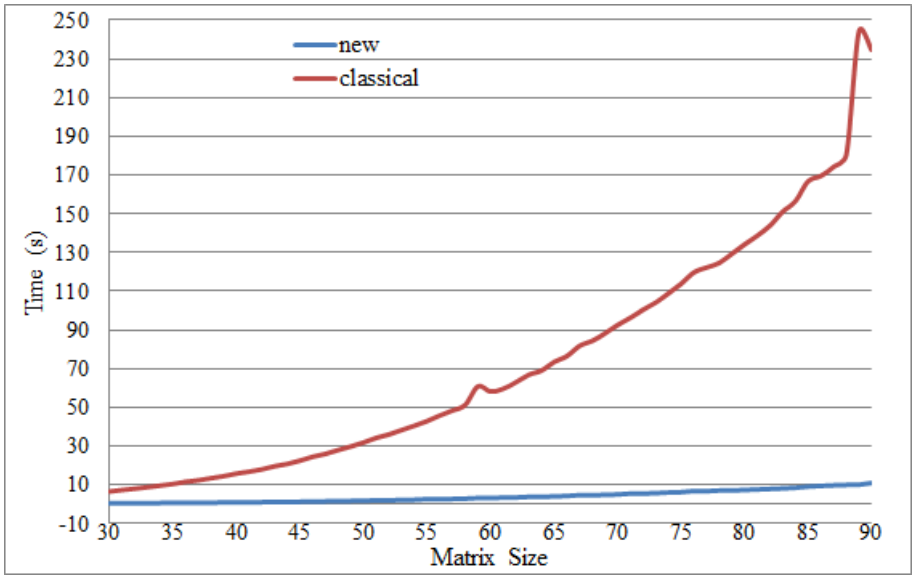


Figure 6.14: The time needed for 500 multiplications with the scripting vs classical method

The graph of Figure (6.15) represents the average time difference in seconds for the operation on matrices raging from 30 to 90 in size. Taking advantage of the particular type of matrix and of its sparsity, the new algorithm provides an improvement up to *0.45 seconds* for the same operation on the same data.

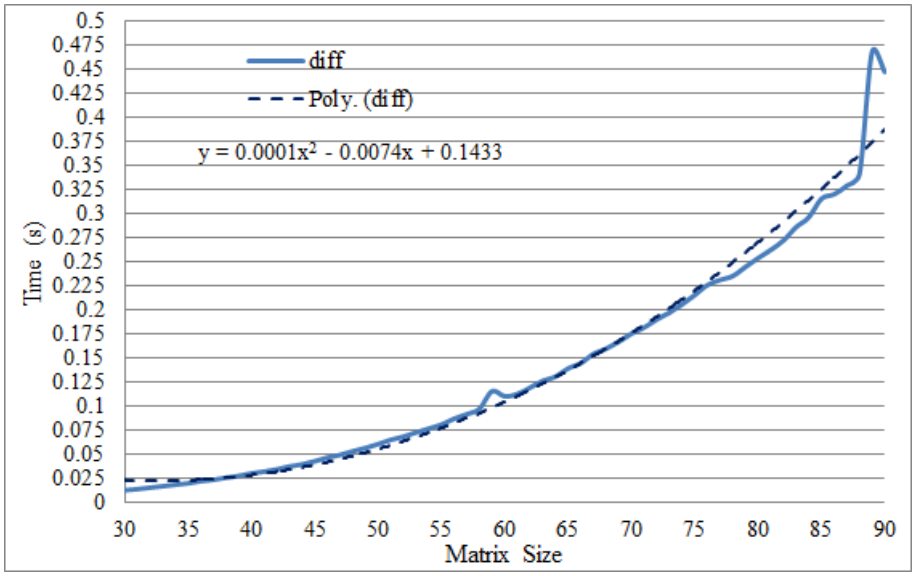


Figure 6.15: The time difference between average classical and scripting algorithm

The best approximation for such a trend is a 2nd degree equation. For this benchmark set of sparse matrices (sizes 30 to 90, 20% density) the time difference of 500 multiplication of each matrix is 70 minutes. Therefore the script as it was generated in listing (6.1) is proven to be more efficient for matrices with the presented constraints.

Chapter 7

Conclusions and Future Work

The objective of this memoir was to find a more efficient way of representing, manipulating and process the data from a particular type of matrix: the sparse matrix.

On one hand such data structures are encountered as the analysis output of problems from economy, technical fields, optimization, process functionality, under the form of a mathematical system. Solving or modeling this set of equations often implies using sparse matrices.

On the other hand some types of images represented as a matrix turn out to be sparse because of the high number of a constant color and very little information embedded, such as the MRI grayscale head images where most of the pixels are black (have the value equal to 0) and the rest carry the information.

Taking into consideration the sparsity character of the matrix will result in a more efficient approach which implies the development of specific applications that use a special data structure. This will save memory and reduce the run time needed for the operations with it.

The domains that produce sparse matrices and use them to solve, model or simulate a system:

- computational fluid dynamics
- model reduction
- thermodynamics
- computer graphics
- optimization
- circuit simulation
- economic and financial modeling
- mathematics and statistics
- power networks
- quantum computing simulation

The input data used for this paper was divided in two working sets with two different sources: *The University of Florida Sparse Matrix Collection* and *Anonymous MRI Brain Scan Images Database (The University of Granada)*.

The results of the work are: a definition and implementation of a new format for the square sparse matrices and the associated operations for it. The effectiveness of multiplication algorithm and storing format was analyzed on the two working sets and compared with the classical storing and operations of matrices. The new approach has been proven to bring a considerable improvement

when working with sparse matrices. Further research paths will consider the matrices in which the most encountered value is not zero.

As for future work, we plan to apply SMF matrices to simulate quantum computing algorithms in traditional computers, which is becoming an outstanding problem that requires optimal memory and run time huge-sized matrix representations of sparse matrices.

Appendix A

Sparse Matrix Format class

```
#include "pch.h"
#include <cmath>
#include <cstring>
#include <stdio.h>
#include <iostream>
#include <cstddef>
#include <fstream>
#include <chrono>
#include <string>
#include <windows.h>
#include <cstdlib>


#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

#define PACK( class_to_pack ) __pragma( pack(push, 1) ) class_to_pack
    __pragma( pack(pop) )

#define size_type unsigned
#define value_type double
#define index_type unsigned long

using namespace cv;
using namespace std;
using namespace std::chrono;

#pragma pack(push, 1)
class entry {
public:
    value_type value;
    size_type column;

    entry() :value(-1), column(-1) {};
    entry(value_type v, size_type c) : value(v), column(c) {};
    value_type getV() {
        return value;
    }

    size_type getC() {
        return column;
    }

    void setV(value_type v) {
        value = v;
    }

    void setC(size_type c) {
        column = c;
```

```
49      }
50
51      void print() {
52          cout << '(' << (int)this->getV() << ", " << (int)this->getC()
               << ')';
53      }
54  };
55  #pragma pack(pop)
56
57  #pragma pack(push, 1)
58  class SMF {
59  public:
60      index_type matrix_size;
61      entry** rows;
62      size_type* row_len;
63      index_type nnz = 0;
64
65      SMF(index_type s) {
66          rows = new entry*[s] {nullptr};
67          row_len = new size_type[s]{ 0 };
68          matrix_size = s;
69      }
70
71      ~SMF() {
72          for (index_type i = 0; i < matrix_size; i++)
73              delete[] rows[i];
74          delete [] row_len;
75      }
76
77      void insert(value_type val, index_type index) {
78          //insert only if val is non-zero
79          if (val != 0) {
80              //calculate row
81              size_type row_no = index / matrix_size;
82
83              //calculate col
84              size_type col_no = index % matrix_size;
85
86              nnz++;
87
88              //if it is the first elem in the row
89              if (row_len[row_no] == 0) {
90                  row_len[row_no]++;
91                  //allocate for one entry and set values
92                  entry* entr_vec = new entry;
93
94                  entr_vec[0].setC(col_no);
95                  entr_vec[0].setV(val);
96
97                  rows[row_no] = entr_vec;
98              }
99              else {
100                 //allocate +1 size and find position to insert the
                       entry
101                 size_type i = 0; //to iterate over col in row
102                 index_type aux_col = rows[row_no][i].column;
103                 while (col_no > aux_col && i < row_len[row_no]) {
104                     i++;
105                     aux_col = rows[row_no][i].getC();
106                 }
107
108                 //if same col do not allocate more, just replace value
109                 if (rows[row_no][i - 1].getC() == col_no) {
110                     rows[row_no][i - 1].setV(val);
111                 }
112                 else {
```

```
113                     //alocate +1 and increase row_size
114                     row_len[row_no]++;
115                     entry* new_r = new entry[row_len[row_no]];
116 
117                     //memcpy the values up to position i
118                     memcpy(new_r, rows[row_no], i * sizeof(entry));
119 
120                     //set values in of pos i, the new_el
121                     new_r[i].setC(col_no);
122                     new_r[i].setV(val);
123 
124                     //copy the rest
125                     memcpy(new_r + i + 1, rows[row_no] + i, (row_len[
                            row_no] - i - 1) * sizeof(entry));
126 
127                     delete[] rows[row_no];
128                     rows[row_no] = new_r;
129                 }
130             }
131         }
132     }
133 
134     void print() {
135 
136         for (index_type i = 0; i < matrix_size; i++) {
137             cout << "[" << (int)i << "]: ";
138             cout << "len=" << (int)row_len[i] << " ";
139             for (index_type j = 0; j < row_len[i]; j++) {
140                 rows[i][j].print();
141                 cout << ' ';
142             }
143             cout << endl;
144         }
145     }
146 
147     SMF add(SMF B) {
148         SMF rez(matrix_size);
149         //for each row
150 
151         index_type apos, bpos;
152         index_type i = 0;
153 
154         //same row for both matrices
155         while (i < matrix_size) {
156             apos = 0; bpos = 0;
157             //get row A & B len
158             size_type len_rowA = row_len[i];
159             size_type len_rowB = B.row_len[i];
160 
161             while (apos < len_rowA && bpos < len_rowB) {
162 
163                 //get A's col
164                 size_type A_col = rows[i][apos].getC();
165                 //get B's col
166                 size_type B_col = B.rows[i][bpos].getC();
167 
168                 //if B's col index is smaller than A's col index
169                 if (B_col < A_col) {
170                     //insert B's val & calc index
171                     rez.insert(B.rows[i][bpos].getV(), i*matrix_size +
                            B_col);
172                     bpos++;
173                     //else if A's col is smaller than B's col
174                 }
175                 else if (B_col > A_col) {
176                     //insert A's val & calc index
```

```
177                     rez.insert(rows[i][apos].getV(), i*matrix_size +
                            A_col);
178                     apos++;
179                     //else same col -> add them
180                 }
181                 else {
182                     rez.insert(rows[i][apos].getV() + B.rows[i][bpos].
                            getV(), i * matrix_size + A_col);
183                     apos++;
184                     bpos++;
185                 }
186             }
187             //insert ramaining el from A
188             while (apos < len_rowA) {
189                 rez.insert(rows[i][apos].getV(), i * matrix_size + rows
                        [i][apos].getC());
190                 apos++;
191             }
192 
193             //insert ramaining el from B
194             while (bpos < len_rowB) {
195                 rez.insert(B.rows[i][bpos].getV(), i * matrix_size + B.
                        rows[i][bpos].getC());
196                 bpos++;
197             }
198 
199             i++;
200         }
201 
202         return rez;
203     }
204 
205     SMF subtract(SMF B) {
206         SMF rez(matrix_size);
207         //for each row
208 
209         index_type apos, bpos;
210         index_type i = 0;
211 
212         //same row for both matrices
213         while (i < matrix_size) {
214             apos = 0; bpos = 0;
215             //get row A & B len
216             size_type len_rowA = row_len[i];
217             size_type len_rowB = B.row_len[i];
218 
219             while (apos < len_rowA && bpos < len_rowB) {
220 
221                 //get A's col
222                 size_type A_col = rows[i][apos].getC();
223                 //get B's col
224                 size_type B_col = B.rows[i][bpos].getC();
225 
226                 //if B's col index is smaller than A's col index
227                 if (B_col < A_col) {
228                     //insert 0 - B's val & calc index
229                     rez.insert(0 - B.rows[i][bpos].getV(), i*
                            matrix_size + B_col);
230                     bpos++;
231                     //else if A's col is smaller than B's col
232                 }
233                 else if (B_col > A_col) {
234                     //insert 0 - A's val & calc index
235                     rez.insert(0 - rows[i][apos].getV(), i*matrix_size
                            + A_col);
236                     apos++;
```

```
237                     //else same col -> subtract
238                 }
239                 else {
240                     rez.insert(rows[i][apos].getV() - B.rows[i][bpos].
                        getV(), i * matrix_size + A_col);
241                     apos++;
242                     bpos++;
243                 }
244             }
245             //insert ramaining el from A
246             while (apos < len_rowA) {
247                 rez.insert(0 - rows[i][apos].getV(), i * matrix_size +
                    rows[i][apos].getC());
248                 apos++;
249             }
250 
251             //insert ramaining el from B
252             while (bpos < len_rowB) {
253                 rez.insert(0 - B.rows[i][bpos].getV(), i * matrix_size
                    + B.rows[i][bpos].getC());
254                 bpos++;
255             }
256 
257             i++;
258         }
259 
260         return rez;
261     }
262 
263     SMF transpose() {
264         SMF rez(matrix_size);
265         value_type val;
266         index_type col;
267 
268         //iterating through rows
269         for (index_type i = 0; i < matrix_size; ++i)
270 
271             //iterating through elements
272             for (index_type j = 0; j < row_len[i]; ++j) {
273 
274                 //get col index
275                 col = rows[i][j].column;
276 
277                 //get value of entry
278                 val = rows[i][j].value;
279 
280                 //insert to rez the new index for this value
281                 rez.insert(val, i * matrix_size + col);
282             }
283 
284         return rez;
285     }
286 
287     index_type bytes_no() {
288         index_type rez = 0;
289 
290         //for each row pointer to entry and length=nnz for the row
291         rez += (sizeof(entry*) + sizeof(size_type)) * matrix_size;
292 
293         //for each non-zero it is used an entry
294         rez += sizeof(entry)*nnz;
295 
296         return rez;
297     }
298 
299     // creating a square matrix_size matrix from SMF
```

```
300     value_type** SMFtoStandard() {
301         value_type **Standard, val;
302         index_type col;
303 
304         //allocating dynamic array (of size=matrix_size)
305         //of pointers to element type (value_type)
306         //initializing all to 0
307         Standard = new value_type*[matrix_size]();
308 
309         //allocate each row
310         for (index_type i = 0; i < matrix_size; ++i)
311             Standard[i] = new value_type[matrix_size];
312         // each i-th pointer is now pointing to dynamic array (size
               matrix_size) of actual value_type values
313 
314       //iterating through rows
315         for (index_type i = 0; i < matrix_size; ++i)
316             //iterating through elements
317             for (index_type j = 0; j < row_len[i]; ++j) {
318                 //get col index
319                 col = rows[i][j].getC();
320 
321                 //get value of entry
322                 val = rows[i][j].getV();
323 
324                 //store the value at the precise indexes
325                 Standard[i][col] = val;
326             }
327 
328         return Standard;
329     }
330 
331     void StandardtoSMF(value_type** Standard) {
332 
333         //iterate through rows
334         for (index_type i = 0; i < matrix_size; ++i)
335 
336             //iterate through columns
337             for (index_type j = 0; j < matrix_size; ++j)
338 
339                 //if entry is non-zero
340                 if (Standard[i][j] != 0)
341 
342                     //insert to SMF value and aggregated index
343                     this->insert(Standard[i][j], i * matrix_size + j);
344     }
345 
346     SMF multiply(SMF B) {
347         SMF rez(matrix_size);
348 
349         index_type col_A, col_B, apos, bpos;
350         value_type sum;
351 
352         SMF Bt = B.transpose();
353 
354         //iterating through rows of A
355         for (index_type i = 0; i < matrix_size; ++i) {
356 
357             //iterating through cols of B
358             //as B is transpose iterate rows
359             for (index_type j = 0; j < matrix_size; ++j) {
360 
361                 //local pointer within A's row
362                 apos = 0;
363                 //local pointer within B's col
364                 bpos = 0;
```

```
365                 //sum of multiplication
366                 sum = 0;
367 
368                 //iterating through elements of A's row & B's col
369                 while (apos < row_len[i] && bpos < Bt.row_len[j]) {
370 
371                     //get col index of A's entry
372                     col_A = rows[i][apos].getC();
373                     //get col index of B's entry
374                     col_B = Bt.rows[j][bpos].getC();
375 
376                     //if A's col is smaller than B's row
377                     //skip entry in A
378                     if (col_A < col_B) {
379                         apos++;
380                         //if B's row is smaller than A's col
381                         //skip entry in B
382                     }
383                     else if (col_A > col_B) {
384                         bpos++;
385                         //else both row and col are equal
386                         //multiply the entries and add to sum
387                     }
388                     else {
389                         sum += rows[i][apos].getV() * Bt.rows[j][bpos].
                               getV();
390                         apos++;
391                         bpos++;
392                     }
393                 }
394                 //if the sum is non-zero add to rezult
395                 if (sum != 0)
396                     rez.insert(sum, i * matrix_size + j);
397             }
398         }
399         return rez;
400     }
401 
402     double density() {
403         double rez = 0;
404 
405         rez = ((double)((double)nnz / (double)(matrix_size*matrix_size)
               )) * 100;
406 
407         return rez;
408     }
409 };
410 #pragma pack(pop)
411 
412 void standardMultiply(value_type** mat1, value_type** mat2, value_type
       ** mat3, size_type m_size) {
413 
414     for (index_type r = 0; r < m_size; r++) {
415         for (index_type c = 0; c < m_size; c++) {
416             for (index_type in = 0; in < m_size; in++) {
417                 mat3[r][c] += mat1[r][in] * mat2[in][c];
418             }
419         }
420     }
421 };
422 
423 int main() {
424     Mat image;
425     image = imread("brain1.jpg", CV_LOAD_IMAGE_COLOR);   // Read the
           file
426 
```

```
427     if (!image.data)                              // Check for invalid
            input
428     {
429         cout << "Could not open or find the image" << std::endl;
430         return -1;
431     }
432
433     namedWindow("Display window", WINDOW_AUTOSIZE); // Create a window
            for display.
434     imshow("Display window", image);                // Show our image
            inside it.
435
436     value_type **im_mat;
437     im_mat = new value_type*[image.cols];
438     for (int i = 0; i < image.cols; i++)
439         im_mat[i] = new value_type[image.rows];
440
441     for (int i = 0; i < image.cols; i++) {
442         for (int j = 0; j < image.rows; j++) {
443             im_mat[i][j] = image.at<uchar>(i, j);
444         }
445     }
446
447     SMF brain(image.rows);
448     brain.StandardtoSMF(im_mat);
449     cout << brain.density() << "%" << endl;
450
451     size_type size = 0;
452     index_type count = 0;
453     value_type nr;
454
455     high_resolution_clock::time_point t1;
456     high_resolution_clock::time_point t2;
457
458         for (unsigned fil = 2; fil <= 111; fil++) {
459             string name = "in", name1 = "out";
460             name = name + to_string(fil) + string(".txt");
461             name1 = name1 + to_string(fil) + string(".txt");
462
463             cout << name << ' ' << name1 << endl;
464
465             fstream myinput(name, ios_base::in);
466
467             if (myinput) {
468                 fstream myoutput(name1, ios_base::out);
469                 while (myinput >> size) {
470
471                     myoutput << size << ' ' << '\n';
472                     nr_mat++;
473
474                     count = 0;
475                     total = (size * size) - 1;
476                     //then read elements
477                     while (myinput >> nr && count < total) {
478                         //standard[count / size][count % size] = nr;
479                         if (nr != 0) {
480                             //test.insert(nr, count);
481                             myoutput << nr << ' ' << count << ' ';
482                         }
483                         count++;
484                     }
485                     myoutput << '\n';
486                 }
487             } else {
488                 cout << "Problem opening file " << name << "\n";
489             }
```

```
490         }
491
492     fstream myoutput("res.txt", ios_base::out);
493     myoutput << "name | Size" << endl;
494     myoutput.flush();
495
496     index_type index = 0;
497     double time_imp = 0, mem_imp = 0;
498     string name;
499     for (unsigned fil = 1; fil <= 112; fil++) {
500         //name = "im_smf" + to_string(fil) + string(".txt");
501         name = "in" + to_string(fil) + string(".txt");
502
503         fstream myinput(name, ios_base::in);
504
505         if (myinput.is_open()) {
506             myinput >> size;
507             myinput.flush();
508
509             SMF test(size);
510             value_type** standard;
511             standard = new value_type*[size]();
512             for (unsigned i = 0; i < size; i++)
513                 standard[i] = new value_type[size]();
514
515             //then read elements
516             index = 0;
517             while (!myinput.eof()) {
518                 myinput >> nr >> index;
519                 standard[index / size][index % size] = nr;
520                 if (nr != 0)
521                     test.insert(nr, index);
522             }
523
524             mem_imp = (double)(100 - (double)((double)test.bytes_no()
525                 / (double)(size * size * sizeof(value_type))) * 100);
526             //test.print();
527             t1 = high_resolution_clock::now();
528             test.multiply(test);
529             t2 = high_resolution_clock::now();
530
531             auto duration = duration_cast<microseconds>(t2 - t1).count
                    ();
532             auto old = duration;
533
534             value_type** mat3 = new value_type*[size];
535             for (unsigned i = 0; i < size; i++)
536                 mat3[i] = new value_type[size];
537
538             t1 = high_resolution_clock::now();
539             standardMultiply(standard, standard, mat3, size);
540             t2 = high_resolution_clock::now();
541
542             for (unsigned i = 0; i < size; i++)
543                 delete[] mat3[i];
544             delete[] mat3;
545
546             duration = duration_cast<microseconds>(t2 - t1).count();
547             time_imp = (double)(100 - ((double)old / (double)duration)
                    * 100);
548
549             myoutput << size << ' ' << test.nnz << ' ' << test.density
                    () << ' ' << mem_imp << ' ' << time_imp << endl;
550             myoutput << name << ' ' << size << endl;
551             myoutput.flush();
552             myinput.close();
```

```
553             }
554             else {
555                 cout << "Problem opening file " << name << "\n";
556             }
557         }
558 
559         return 0;
560 }
```

Appendix A

New multiplying algorithm

```
import sys
import time
from random import randint
import itertools

def new_mult_generator(n):
    program = 'C = [0] * (%d*%d)' % (n, n)

    for i in range(0, n*n):
        block = i / n

        program += '''
if A[%d] != 0:''' % i

        for k in range(0, n):
            b_indx = (i * n + k) % (n*n)

            program += '''
    if B[%d] != 0:
        C[%d] += A[%d] * B[%d]''' % (b_indx, k + (block * n), i, b_indx
            )

    return program

def classical_mult(A, B, n):
    C = [0] * (n*n)
    for i in range(0, n):
        for j in range(0, n):
            for k in range(0, n):
                C[i*n + j] += A[i*n + k]*B[k*n + j]

if __name__ == '__main__':
    max_iterations = 2000
    print('n;new;classical')
    for n in range(10, 150):
        A = [0] * (n*n)
        B = [0] * (n*n)

        for k in range(0, n):
        A[k] = randint(0, 255)
        B[k] = randint(0, 255)

        program = compile(new_mult_generator(n), 'mult', 'exec')

        start = time.time()
        for _ in itertools.repeat(None, max_iterations):
            exec(program)

        new_time = (time.time() - start) / max_iterations
```

```
49
50          start = time.time()
51          for _ in itertools.repeat(None, max_iterations):
52              classical_mult(A, B, n)
53
54          classical_time = (time.time() - start) / max_iterations
55          print(str(n)+';'+str(new_time)+';'+str(classical_time))
```

Bibliography

[1] I. Ivan, M. Popa, and P. Pocatilu, *Structuri de date: Tipologii de structuri de date*, 2008. [Online]. Available: http://www.ionivan.ro/2015-PUBLICATII/CARTI.htm → pg. 1, 2, 5, 6, 10, 13, 14, 16

[2] S. Pissanetzky, *Sparse Matrix Technology*, 1984. [Online]. Available: http://sergio.pissanetzky.com/Publications/BookSampleSMT.pdf → pg. 3

[3] SciPy, *Dictionary of Keys Format (DOK)*. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.dok_matrix.html → pg. 3, 9

[4] Scipy, *List of Lists (LIL)*. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.lil_matrix.html → pg. 3, 9, 10

[5] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, *Yale Sparse Matrix Package II. The Nonsymmetric Codes*, 1977. [Online]. Available: https://apps.dtic.mil/dtic/tr/fulltext/u2/a047725.pdf → pg. 3, 12

[6] K. E. Atkinson, *An Introduction to Numerical Analysis*, 1989. [Online]. Available: http://www.math.science.cmu.ac.th/docs/qNA2556/ref_na/Katkinson.pdf → pg. 5

[7] R. A. Brualdi, *Combinatorial matrix classes*, 2006. [Online]. Available: https://www.researchgate.net/publication/267113963_Combinatorial_Matrix_Classes → pg. 7

[8] *The Suite Sparse Matrix Collection*. [Online]. Available: https://sparse.tamu.edu/ → pg. 27, 28, 33