

Computable Compressed Matrices

Crysttian Arantes Paixão^{1*}, Flávio Codeço Coelho²,

1 Crysttian Arantes Paixão Applied Mathematics School, Getulio Vargas Foundation, Rio de Janeiro, RJ, Brazil

2 Flávio Codeço Coelho Applied Mathematics School, Getulio Vargas Foundation, Rio de Janeiro, RJ, Brazil

*** E-mail: Corresponding author@institute.edu**

Abstract

Author Summary

Introduction

Data compression is traditionally used to reduce storage resources usage and/or transmission costs [1]. Compression techniques can be classified into lossy and lossless. Good examples of lossy data compression are sound (mp3), image (JPEG) and movie (mpeg) compression. In this paper we are concerned exclusively with lossless compression. In lossless compression, no information is lost, during the compression process, and they are the method of choice to compress numbers and text.

Any kind of compression incurs in some computational cost. Such costs often have to be paid twice since the data needs to be decompressed to be used for its original purpose. Sometimes computational costs are irrelevant, but the need to decompress for usage, can signify that savings in storage costs are applicable to storage situation since we still require the uncompressed space allocation for usage.

Lossless compression methods usually try to exploit redundancies present in the data in order to find a shorter form for of describe the same information content, for example a dictionary based compression, can only store the positions in which a given word occurs in a document, thus saving the space required to store all its repetitions [2].

Most if not all existing lossless compression methods were developed under the following paradigm: produce \rightarrow compress \rightarrow store \rightarrow uncompress \rightarrow use. However in the brave new world of big data, the need to analyze data immediately after its coming into existence became the norm. And this analysis must take place, efficiently, within the confines of (RAM) memory. Given a sufficiently big flow of data, compression and decompression costs may become prohibitive as it will slow down data transmission. Moreover, the need to decompress before analyzing removes any potential gains in memory usage.

With the growth of data and analytical demands, creative solutions are needed to not only efficiently store large data as consume it on demand. This phenomenon is present in many areas of application, ranging from business to science [3], and is being called the big data phenomenon.

One data type which is very common across areas of application is numerical data. Most common statistical and machine learning algorithms operate over large matrices, frequently making use of linear algebra methods. So it would be very desirable to have a compression method in which the compressed object retained the mathematical properties as the original, i.e. could be subject to the same mathematical operations. The idea of operating with compressed array is not new [4], but has yet to be applied consistently to the field of numerical computations. One application which employs a form compression is the sparse matrix linear algebra algorithms [5], in this case there is no encoding of the data, but only the non-zero elements of the matrices are stored and operated upon.

Currently, the technique most commonly used when dealing with large matrices for numerical computations, is memory mapping [6,7]. In memory mapping the matrix is allocated in a virtual contiguous address space which overflows from memory into disk, this way larger than memory datastructures can be manipulated as if they were in memory. This technique has a big performance penalty due to much

lower access speeds of disk when compared to RAM.

In this paper we present two methods that allow the lossless compression of (numerical) arrays, which do not require decompression to be manipulated mathematically. The method involves the encoding of the numbers as strings of bits of variable length. The methods resemble the arithmetic coding [8] algorithm, but is cheaper to compute and does not require decompression. We describe the process of compression and decompression, and how to operate with the arrays in compressed format. We discuss the efficiency of the compression as a function of the distribution of the elements of the matrix.

Methods

Matrix compression

The main goal of our algorithm in the context of this paper was to achieved the best compression ratio possible while maintaining mathematical equivalence with the original data regarding all arithmetic operations. In order to achieve this, we decided we needed to maintain the structure of the matrix, i.e., the ability to access any element given its row i and column j . In order to achieve compression we decided to exploit inefficiencies in the conventional way matrices are allocated in memory.

The compression method is as follows. Consider a matrix $M_{r \times c}$ in which r is the rows number and c the columns number. Each element of this matrix, called m_{ij} , is an integer. Since all information is stored as binary code in digital computers, the conventional way to store arrays of integers in most programming languages is to chose a maximum size for the elements (up to 64bits in a typical CPU), and then allocating a block of memory sufficient to store all the elements of M :

$$\mathcal{B} = r \times c \times 64 \quad (1)$$

The number of bits allocated \mathcal{B} , is larger than the absolute minimum number of bits required to represent all the elements of M . To help illustrate this fact let's consider as an extreme example: a matrix composed exclusively of 0s and 1s. If we set the element to be 64-bit integers we willbe wasting 63 bits per element of the matrix, since the number of bits really needed to store the this matrix, \mathfrak{b} is given by:

$$\mathfrak{b} = l \times c \times 1 \quad (2)$$

The potential economy of bits ξ can be represented by the difference between \mathcal{B} and \mathfrak{b} (Equations 3 to 4).

$$\xi = \mathcal{B} - \mathfrak{b} \quad (3)$$

$$\xi = r \times c \times 63 \quad (4)$$

From equations 1 to 4, it can be seen that for any matrix whose greatest element requires less than 64 bits (or the fixed type of the matrix) to be represented, memory savings ξ grow linearly with the size of the matrix.

Now we will introduce two simple methods to exploit these inefficiencies in conventional memory allocation, and save a lot of memory in the process. In this section we will use capital roman letters to denote the original uncompressed matrices and the corresponding lower case letter for the compressed version.

Method 1: The Supreme Minimum

The first methodology consists in determining the value of the greatest element of matrix M , which coincides with its supremum, $\max M == \sup M$ and determine the minimum number of bits required to store it (b) .

$$b(\sup M) \approx \begin{cases} 1, & \text{if } \sup M \in \{0, 1\} \\ \lfloor \log_2(\sup M) \rfloor + 1, & \text{if } \sup M > 0 \end{cases} \quad (5)$$

The allocation of memory still happens in the usual way, only that now, in the space required for a single 64 bit integer, we can store an entire 8×8 matrix of 0s and 1s.

To make it clearer, lets look at a concrete example: suppose that the greatest value to be stored in a matrix M is $\max M = 1023$. Therefore, the number of bits required to represent it is 10 (1111111111). Let the first 8 elements of M be:

$$M = \begin{bmatrix} 900 & 1023 & 721 & 256 & 1 & 10 & 700 & 20 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \quad (6)$$

These elements of M , in binary, are shown in Table 1. It is evident that the number of bits required to represent any other element must be ≤ 10 .

Table 1. Some elements of M represented in binary base.

Element	Value	Binary	Bit length
$M_{1,1}$	900	1110000100	10
$M_{1,2}$	1023	1111111111	10
$M_{1,3}$	721	1011010001	10
$M_{1,4}$	256	100000000	9
$M_{1,5}$	1	1	1
$M_{1,6}$	10	1010	4
$M_{1,7}$	700	1010111100	10
$M_{1,8}$	20	10100	5

Now consider a matrix m , which corresponds to M converted to base 2 (M_2). How do we store such matrix in memory? The conventional way to allocate matrices in memory is to unravel its structure by column (column major, e.g. in Fortran) or by row (row major, e.g. in C) and line them up in a contiguous block of memory. Each element of the matrix is allocated in turn according to the type associated with the matrix.

For our example, since we have determined that each element will require at most 10 bits, we can divide the memory block corresponding to a single 64 bit integer into 6 10-bit chunks which can each hold a single element of M . Each of these blocks we will call a bitstring. The remaining 4 bits will be used later. The number of bit strings needed will be $\lfloor \frac{\dim(M)*b}{64} \rfloor + 1$.

The final layout of the first 6 elements of m in the first bitstring can be seen in equation 7.

$$bitstring1 = 0000 \underbrace{0000001010}_{10} \underbrace{0000000001}_1 \underbrace{0100000000}_{256} \underbrace{1011010001}_{721} \underbrace{1111111111}_{1023} \underbrace{1110000100}_{900} \quad (7)$$

Here is a step-by-step description of the allocation process:

1. Element $M_{1,1} = 900 = m_{1,1} = 1110000100$ is stored in the first 10-bit chunk of the element strip $bitstring[1]$, which corresponds to bits 0 to 9.

We can further simplify equation 19 to get at shorter expression for the compression ratio.

$$\begin{aligned}
\eta_2 &= \frac{64 \times rc - \sum_{i=1}^g (b_i \times f_i + k \times f_i)}{64 \times rc} \\
&= \frac{64 \times rc - \sum_{i=1}^g b_i \times f_i - \sum_{i=1}^g k \times f_i}{64 \times rc} \\
&= \frac{64 \times rc - \sum_{i=1}^g b_i \times f_i - k \times \sum_{i=1}^g f_i}{64 \times rc} \\
&\quad \sum_{i=1}^g f_i = rc \\
\eta_2 &= \frac{64 \times rc - \sum_{i=1}^g b_i \times f_i - k \times rc}{64 \times rc} \\
&= 1 - \frac{\sum_{i=1}^g b_i \times f_i}{64 \times rc} - \frac{k}{64}
\end{aligned} \tag{20}$$

Results

Consider a matrix M such that $m_{i,j} \in \mathbb{Z}$. In this section we will examine and compare the efficiency of the two compression methods described.

Since the efficiency of both methods depend on the distribution of the bit lengths $b(m_{i,j})$, we will explore a variety of distributions and try to cover the most common cases.

For simplicity we will model the distribution of as a mixture X of two Beta distributions, $B_1 \sim \text{Beta}(\alpha_1, \beta_1)$ and $B_2 \sim \text{Beta}(\alpha_2, \beta_2)$, whose probability function is shown in equation (eq.21). Since the Beta distribution is defined only in the interval $[0, 1] \subset \mathbb{R}$, we applied a simple transformation (23) to the mixture in order to map it to the interval of $[1, 64] \subset \mathbb{Z}$

$$F(x) = w \text{Beta}(\alpha_1, \beta_1) + (1 - w) \text{Beta}(\alpha_2, \beta_2) \tag{21}$$

The intention of using this mixture was to find a simple way to represent a large variety of bit length distributions. The first two central moments of this mixture are given in 22 and will be used later to summarize our numerical results.

$$\begin{aligned}
E(X) &= wE(B_1) + (1 - w)E(B_2) \\
&= w \frac{\alpha_1}{\alpha_1 + \beta_1} + (1 - w) \frac{\alpha_2}{\alpha_2 + \beta_2} \\
\text{Var}(X) &= w\text{Var}(B_1) + (1 - w)\text{Var}(B_2) + w(1 - w)(E(B_1)^2 - E(B_2)^2)
\end{aligned} \tag{22}$$

$$\lfloor 64 \times x \rfloor + 1 \tag{23}$$

In order to explore the compression efficiency of both methods, we generated samples from the mixture defined above, varying its parameters. From now on, when we mention Beta distribution we will mean the transformed version defined above.

Given any matrix generated by our mixture distribution, we can use equation 16 and 20, to determine the compression efficiency of methods 1 and 2, respectively.

With $w = 0$, ie, our mixture is reduce to a single Beta distribution. In Figure 2, we show some distributions of bit-lengths for some combinations of α_1 and β_1 . From the figure it can be seen that we can generate just about any kind of unimodal distributions in the interval $[1, 64]$.

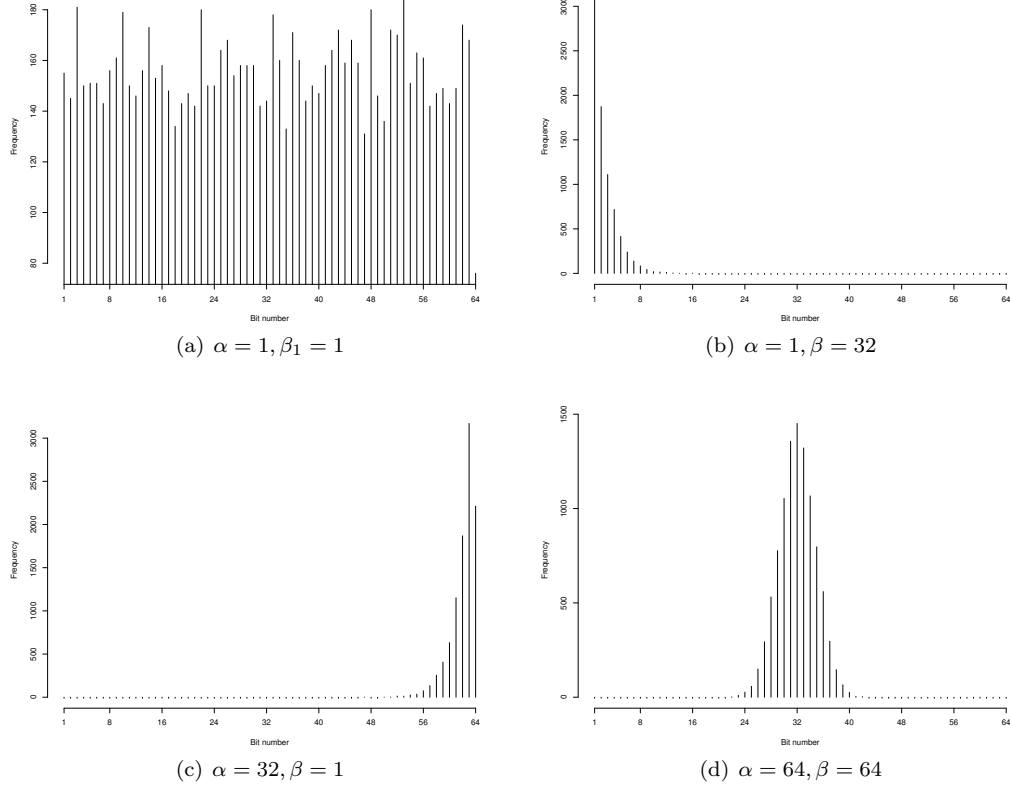


Figure 2. Histograms constructed from samples with 10,000 elements, generated from a Beta distribution. Below each histogram is possible to verify the parameters used.

As we are sampling from a set of all possible distributions of bit-length, represented by the mixture of betas presented above, in order to make our results more general, we will base our analysis on the expected bit-length of a sample, since the efficiency of both methods depends on it. So, from equations 16 and 20, the expected efficiencies become:

$$E(\eta_1) = 1 - \frac{k}{64} \quad (24)$$

$$E(\eta_2) = 1 - \frac{E(b)}{64} - \frac{k}{64} \quad (25)$$

where k in the expression of η_2 is set to 7 (the bit-length required to represent the largest possible bit-length: 64). In 24, k is the bit-length of the greatest element, or in the worst case, 64.

We will use the difference $D = E(\eta_1) - E(\eta_2)$ to compare the efficiency of the two methods. Thus a positive D will favor method 1 while a negative D favors method 2.

For the numeric experiments, we generate 3 samples of size 10000, from which we calculate the expected compression efficiency, using equations 24 and 25. The values presented in tables and figures below are averages of the efficiencies of the three samples.

In Figure 3, we can see the distribution of efficiencies and their difference for a sample generated from a single Beta distribution of bit-lengths. We can see that both methods can achieve efficiencies greater

than 80% for matrices with very small numbers. We also see in figure 3 that method 2 is more efficient in the majority of cases.

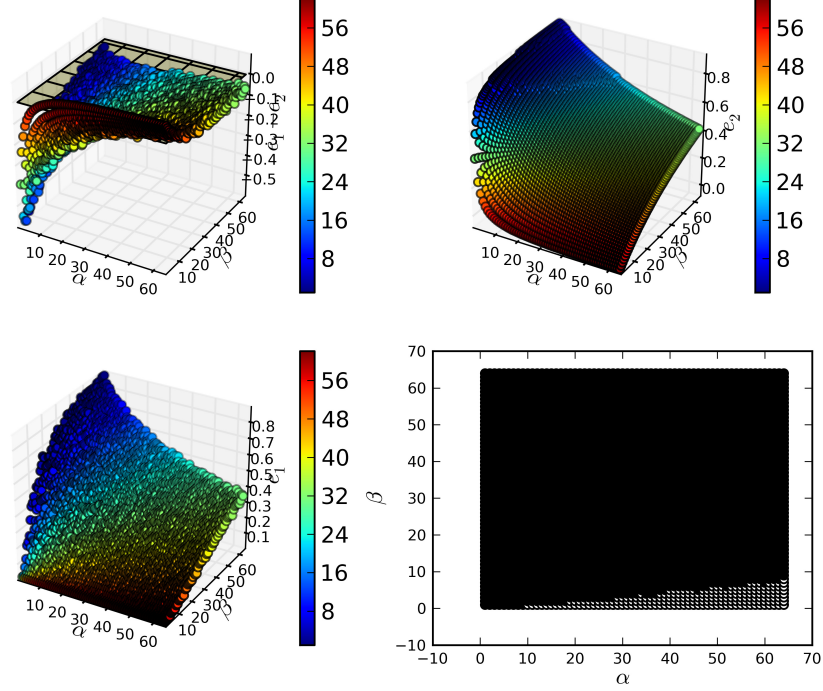


Figure 3. Comparing compression efficiency of methods 1 and 2. Color scale in (a), (b) and (c) represent average bit-length. In (a) we can see the difference $D = \eta_1 - \eta_2$. It can be seen that for most combination of α and β , $D < 0$, meaning the second method is more efficient to compress a sample of numbers with bit-lengths coming from a $Beta(\alpha, \beta)$ distribution. However, there is a small region in parameter space, which is shown in white on (d), where the first method is more efficient. This region corresponds to the dots in red in (a), where the average bit-length is higher. In panels (b) and (c), we can see the efficiencies of methods 1 and 2, respectively.

Now let's analyze matrices whose elements have bit lengths sampled from a mixture of beta distributions, B_1 and B_2 . For this mixture B we set $w = 0.5$, i.e., the mixture will have half the numbers generated from the distribution $B_1 \sim Beta(\alpha_1, \beta_1)$ and the other part distribution $B_2 \sim Beta(\alpha_2, \beta_2)$. The expected value for this mixture is shown in equation 26.

$$E(B) = 0.5E(B_1) + 0.5E(B_2) \quad (26)$$

We must also redefine the calculations of efficiency for the mixture samples. Equation 27 shows the expected efficiency for method 1. Now, instead of having the efficiency being a function of greatest bit-length in the sample (denoted as k in 24 and 25), it will be a function of $\max\{E(B_1), E(B_2)\}$. We do the same for method 2 (equation 28).

$$E(\eta_1) = 1 - \frac{\max\{E(B_1), E(B_2)\}}{64} \quad (27)$$

$$E(\eta_2) = 1 - 0.5 \frac{E(B_1)}{64} - 0.5 \frac{E(B_2)}{64} - \frac{\max\{E(B_1), E(B_2)\}}{64} \quad (28)$$

As before, we generate 3 samples of size 10000 for each parameterization, calculate the average efficiencies (equations 27 and 28) and their difference D .

Before we move on to the efficiency results and analyses, let's first inspect our samples from the mixture of transformed Beta distributions. Figures 4 and 5, show a few parameterizations and their resulting sample distributions. It is important to note that from the mixture we can now generate bimodal distributions as well as the unimodal types tested before. Since we are making statements about efficiency as a function of the expected bit-length, it is important to verify if these statements hold for bimodal distributions as well.

After sampling uniformly ($[1, 5, 9, \dots, 64]$, $n = 65536$) the parameter space and comparing efficiencies, we summarized the results on table 2. In it we see how many parameterizations (from our sample) favor each method. We can also look at the distribution of efficiencies on our samples for each method (figure 6), which clearly demonstrate the greater expected efficiency of method 2 (figure 6(b)).

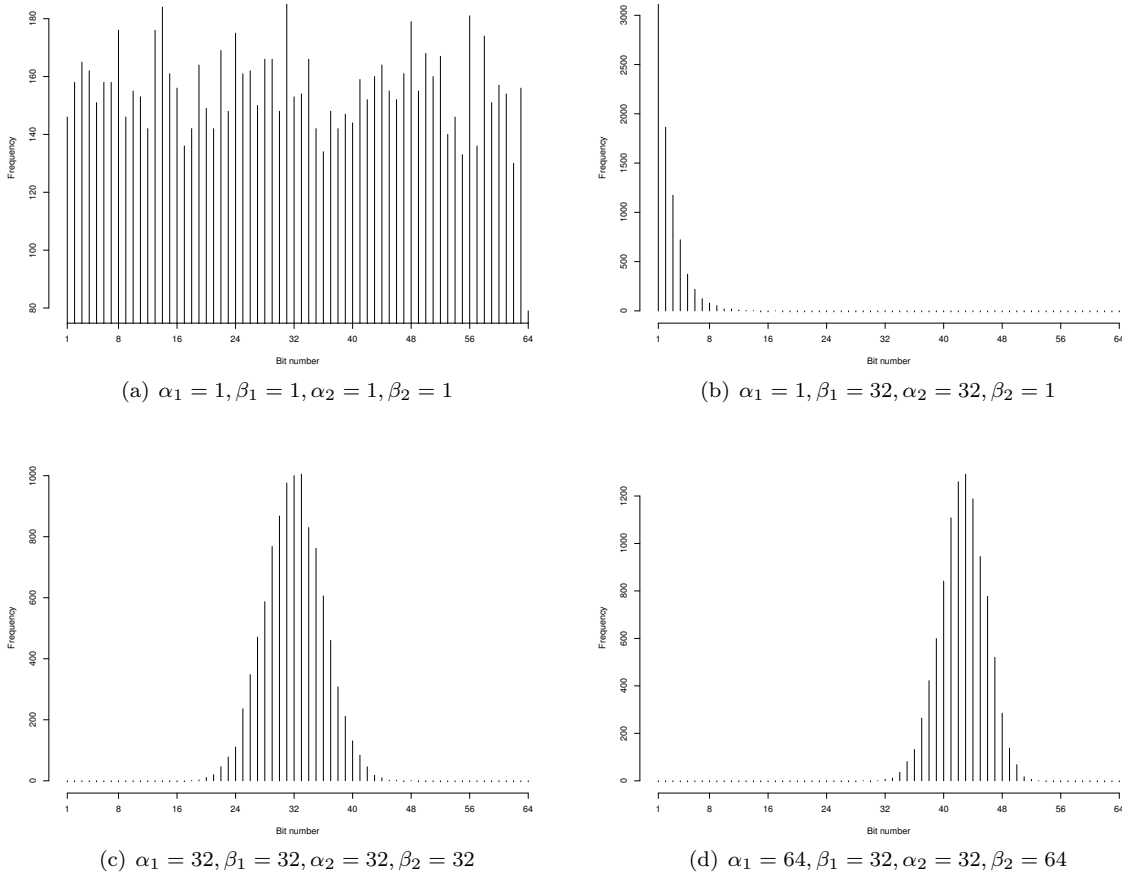


Figure 4. Histograms constructed from samples with 10,000 elements, generated from the combination of two distributions Betas. Below each histogram is possible to verify the parameters used..

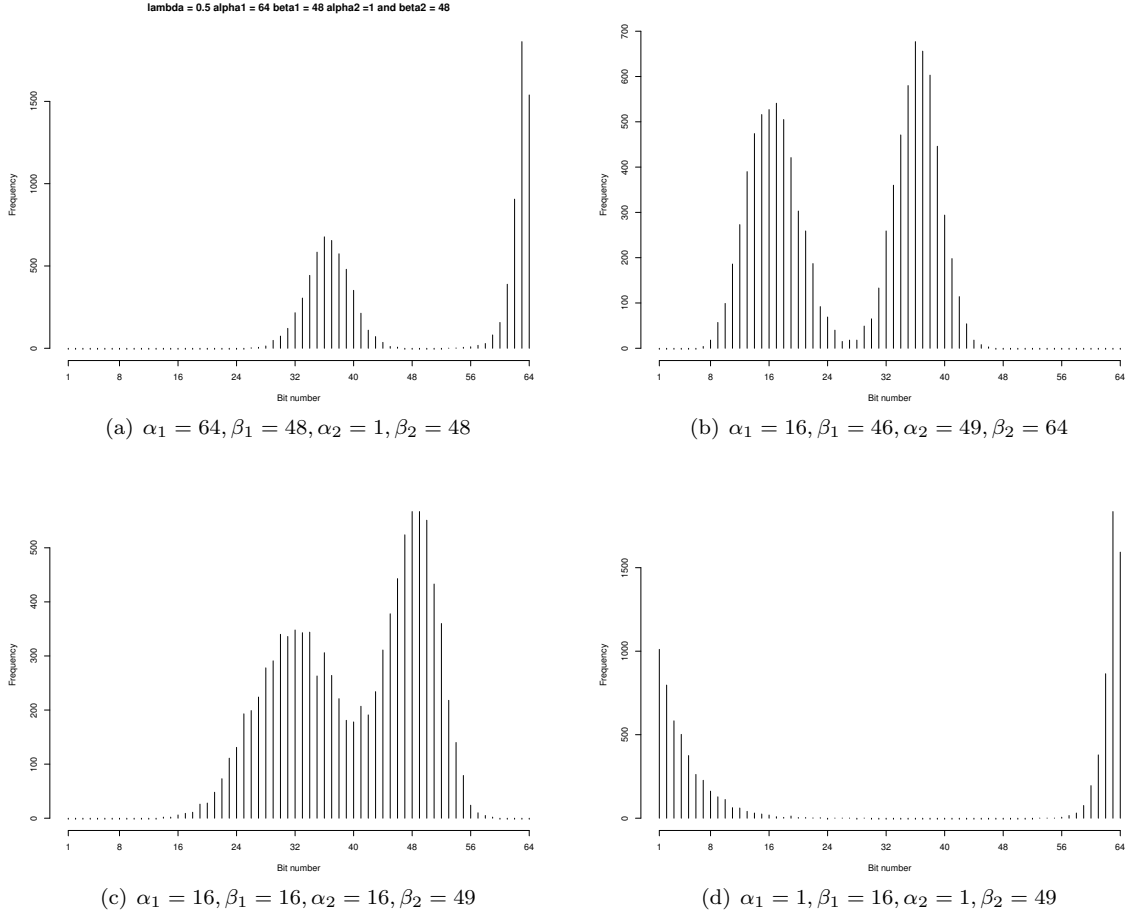


Figure 5. Histograms constructed from samples with 10,000 elements, generated from the combination of two distributions Betas. Below each histogram is possible to verify the parameters used.

As we have shown, method is more effective compressing most integer datasets up to 64bits in size. This is due to its ability to exploit the variance in the data set and reduce the waste of bits in the representation of some numbers. In specific cases where the variance in the data null or too small, method I will be more efficient. As a matter of fact, for matrices where all elements have the same bit-length, method I will always be better, regardless of bit-length (figures 7(a) and 7(b)), The only exception if for bit-length 64 where neither method is able to compress the data.

The two methods presented allow the storage and information manipulation in the compressed form, which provides an data managing large amounts optimization. The next step in the work development is the implementation of the proposed methods, as well as tests to be performed on data large volumes.

Table 2. Efficiency comparison of methods I and II for parameters covering uniformly the support of B . Column n shows the number of parameter combinations with which each method has superior compression.

Methods	n	Percentage
I	592	0.9034%
II	64944	99.0966%
Total	65536	100%

Discussion

Which method to use?

To determine the best compression method to apply, it's necessary to inspect the distribution of bit-lengths of matrix elements. When matrix elements are small or have nearly-constant bit-length, Method I is best, otherwise, the second method should be chosen.

As an example, let $M_{r \times c}$ be a integer matrix such that the half of its elements have bit-length 1 and the other half 64. Recalling equation 20, now we have two groups of elements (by bit-length), $b_1 = 1$, $b_2 = 64$ and $f_i = \frac{rc}{2}$ for $i = 1$ and 2. As the greatest bit-length is 64, then $k = 7$. Compression efficiency η_2 can be calculated using Equation 20. After plugging in our numbers, we obtain a compression of 38.29%.

$$\begin{aligned}\eta_2 &= 1 - \frac{\sum_{i=1}^2 b_i \times f_i}{64 \times rc} - \frac{7}{64} \\ \eta_2 &= 1 - \frac{1 \times \frac{rc}{2} + 64 \times \frac{rc}{2}}{64 \times rc} - \frac{7}{64} \\ \eta_2 &= 1 - \frac{32.5 \times rc}{64 \times rc} - \frac{7}{64} \\ \eta_2 &\approx 1 - 0.5078 - 0.1093 \\ \eta_2 &\approx 38.29\%\end{aligned}$$

The efficiency of method II is influenced by the relative size of the bit-length groups. In this first example we considered only two groups, each comprised of half the matrix elements. Let's now vary the relative frequency of the groups, $\frac{f_i}{rc}$, while sticking to two groups. Let's also assume that $\frac{f_i}{rc}$ is a good approximation to the probability of a given bit-length in a matrix, which we will denote by p_i .

With this definition we can rewrite the equation 20, which becomes 29. In Equation 29, the $\frac{f_i}{rc}$ is replaced by p_i , representing the probability of elements from group i in matrix M .

$$\eta_2 = 1 - \frac{\sum_{i=1}^g b_i \times p_i}{64} - \frac{k}{64} \quad (29)$$

with

$$p_i = \frac{f_i}{rc} \quad (30)$$

With the equation 29 can analyze the influence of bit-length probability in compression efficiency. In this example, p_1 and p_2 represent the probability of elements of bit-lengths 1 and 64, respectively. Thus, efficiency is defined in Equation 31.

$$\eta_2 = 1 - \frac{1 \times p_1 + 64 \times p_2}{64} - \frac{7}{64} \quad (31)$$

Now, we can determine which probabilities give us the best and worst compression levels. When $\eta_2 = 1$, then the efficiency is maximal and if $\eta_2 = 0$, a efficiency is minimal. To calculate the values of p_1 and p_2 for both extreme values of η_2 , we must solve the linear systems shown in equations 32 and 33. The first equation on both systems come from the law of total probability. The second comes from 31 after setting η_2 to 1 and 0, respectively.

$$\eta_2 = 1 : \begin{cases} p_1 + p_2 = 1 \\ p_1 + 64p_2 = 7 \end{cases} \quad (32)$$

Solving the system above, we find that when $p_1 = 0.9047$ and $p_2 = 0.0953$, efficiency is maximal, and in this particular case is equal to 87.5%.

$$\eta_2 = 0 : \begin{cases} p_1 + p_2 = 1 \\ p_1 + 64p_2 = 57 \end{cases} \quad (33)$$

Thus, when $p_1 = 0.1111$ and $p_2 = 0.8889$ the efficiency is minimal for the second method. For other combinations see table 3. Looking at this table, one can see two negative efficiencies, when (p_1, p_2) assume the values (0,1) and (0.1,0.9). This correspond th cases when the method increases the memory requirements instead of decreasing it.

Table 3. Combinations p_1 and p_2 to calculate the efficiency.

p_1	p_2	Efficiency
0.0	1.0	-0.109
0.1	0.9	-0.010
0.2	0.8	0.087
0.3	0.7	0.185
0.4	0.6	0.284
0.5	0.5	0.382
0.6	0.4	0.481
0.7	0.3	0.579
0.8	0.2	0.678
0.9	0.1	0.776
1.0	0.0	0.875

So far, we have examined only two groups (hence two probabilities) of bit-length for the sake of simplicity. Before we generalize to probability distributions let's take a quick look at the efficiencies for more groups, with uniform probability:

- 3 number groups that require numbers 1, 32 and 64 bits, efficiency $\zeta = 0.3854$,
- 5 number groups that require numbers 1, 16, 32, 48 and 64 bits, efficiency $\zeta = 0.3875$,
- 8 number groups that require numbers 1, 8, 16, 24, 32, 40, 48, 56 and 64 bits, efficiency $\zeta = 0.3888$

When the distribution of the group probabilities is uniform, i.e. the groups have approximately the same size, efficiency is basically the same, regardless of the number of groups.

Now we can leverage the notion of bit-length probabilities, and study efficiency when bit-lengths follow some commonly used discrete probability distributions: Discrete Uniform, Binomial and Poisson. For all the experiments, we assume $k = 7$, that is, the maximum possible bit-length is 64 bits. Thus, efficiency

obtained will not be the best possible, since for that we would need assume small values of k (equation 29).

Discrete Uniform

The discrete uniform distribution has two parameters, a and b representing the minimum and maximum value of the distribution. For our case we will use $U(a = 1, b = 64)$, which means bit-lengths may take values in the set $\{1, 2, 3, \dots, 64\}$ with equal probability, i.e., $\frac{1}{64}$.

For the Poisson distribution, we use $\text{Poisson}(\lambda)$ with λ being the expected bit-length in the sample.

To calculate the efficiency, we generated a sample with 100 ($M_{10 \times 10}$), 10,000 ($M_{100 \times 100}$) and 1,000,000 ($M_{1,000 \times 1,000}$) numbers. The average efficiency (table 4) is calculated from a 1000 replicates of each sample size.

Table 4. Compression efficiency of method II of samples with bit-lengths coming from a Discrete Uniform distribution $U(a = 1, b = 64)$. Average efficiency were calculated over a 1000 replicates.

Sample size	Avg. efficiency
100	0.3760938
10000	0.3795953
1000000	0.3826869

Binomial Distribution

For the binomial distribution, we will use $\text{Bin}(n, p = 0.5)$, with the number of trials n representing the greatest possible bit-length in the matrix, and $p = 0.5$ giving us an expected bit-length of 32.

For these experiments, the parameter n represents the maximum bit-length of matrix elements and takes values in $\{1, 8, 16, 32, 64\}$. In this case, we evaluate the efficiency as a function of the parameter n , and sample size. Even though efficiency does not depend on matrix size, we tried different sizes to test the stability of the compression algorithm. Results are shown in Table 5. As expected smaller expected bit lengths lead to higher compression efficiencies. However, for the case in which the elements require 64 bits, the efficiency is negative. [REMOVE AFTER FIXING TABLE] This result indicates that using the second method, when elements require 64 bit this becomes impractical, because it would be require more memory to allocate the compressed matrix.[REMOVE]

Table 5. Compression efficiency with bit-lengths distributed according to a binomial distribution $B(n, 0.5)$. Parameter $n \in \{1, 8, 16, 32, 64\}$ represents the maximum bit-length. Since $p = 0.5$ the expected bit-length is $n/2$ (first column). [REMOVE LAST LINE SINCE WE CANNOT HAVE AN EXPECTED VALUE OF 64 and check for correct values]

Expected bit-length ($n/2$)	Efficiency		
	Sample size		
	100	1,000	1,000,000
1	0.9760937	0.9764797	0.9765566
8	0.8109375	0.8128453	0.8125079
16	0.626875	0.6251859	0.6249333
32	0.2476562	0.2504953	0.2499712
64	-0.5004688	-0.4999828	-0.5000366

Poisson Distribution

With bit-length derived from a $\text{Poisson}(\lambda)$, the parameter λ corresponds to the expected bit-length. Values greater than 64 generated in this simulation were considered as being equal to 64. The results for this simulation can be seen in Table 6. Note that increasing the bit number to represent numbers increases, there is a loss of efficiency in the compression process. Another interesting fact is the emergence of a negative efficiency. In this case, for values generated with a $\lambda = 64$, the second method requires more memory to allocate the matrix formed by elements generated from this distribution.

Table 6. Compression efficiency with bit-lengths distributed according to a Poisson distribution (λ), where λ represents the expected bit-length. [REMOVE LAST LINE]

Expected bit-length (λ)	Efficiency		
	Size sample		
	100	1,000	1,000,000
1	0.8746875	0.8749859	0.8749834
8	0.7657812	0.7656016	0.7656138
16	0.6300000	0.6400594	0.6405952
32	0.3851562	0.3903531	0.3903477
64	-0.0528125	-0.05984375	-0.05953145

With bit-length as a random variable, we can generalize the results about the compression efficiency as a function of expected bit-length

Let the random variable $B \sim U(a = 1, b = 64)$ represent the bit-length of the elements of matrix M . Then $E(b_i) = \sum_i b_i \times p(b_i) = \frac{a+b}{2}$. Applying this result to the expected compression efficiency of method II (equation 29), we have

$$E(\eta_2) = 1 - \frac{E(B)}{64} - \frac{k}{64} \quad (34)$$

assuming all bit-lengths are possible, i.e., $a = 1$ and $b = 64$, and hence $k = 7$, we can calculate η_2 :

$$E(\eta_2) = 1 - \frac{\frac{1+64}{2}}{64} - \frac{7}{64} \approx 38.28\% \quad (35)$$

This result agrees with the numerical estimates presented in table 4. We can do the same for the case where bit-length (B) is a random variable with Binomial and Poisson distributions. In the case of Binomial, $B \sim B(n = 64, p = 0.5)$, $E(b_i) = \sum_i b_i \times p(b_i) = n \times p$ and the efficiency becomes (with $k = 7$):

$$\begin{aligned} E(\eta_2) &= 1 - \frac{64 \times p}{64} - \frac{k}{64} \\ &= 1 - \frac{64 \times 0.5}{64} - \frac{7}{64} \approx 39.05\% \end{aligned} \quad (36)$$

Which again agrees with estimates in Table 5.

Lastly, for B distributed as $\text{Poisson}(\lambda = 32)$ case, $E(b_i) = \lambda$, $k = 7$, and the efficiency becomes:

$$E(\eta_2) = 1 - \frac{\lambda}{64} - \frac{k}{64} \quad (37)$$

$$= 1 - \frac{32}{64} - \frac{7}{64} \approx 39.06\% \quad (38)$$

This result is again in accordance to Table 6. These results show that a good compression is guaranteed when bit-lengths are distributed according to the tested distributions regardless of sample size.

[devemos incluir um estudo sobre a variancia?]

In this paper we have focused in the compression of matrix data, since this is one of the most important application the authors foresee. However, the compression methodology presented can be applied to any numerical data structure, with gains to performance and memory footprint[citar tese Crysttian e possveis artigos derivados].

Further discussions about doing computation with such compressed data-structures will be the subject of another manuscript (in preparation) in which we will present details about the implementation of the compression algorithm, and benchmarks on classical linear algebra tasks such as those in Linpack[ref].

Representation of floating point numbers is also possible within the proposed compression framework, but at the expense of precision in their representation. Although this may sound like a limitation, when we take into consideration that most experimental data have fewer “significant” digits than the maximal precision available in modern computers, fairly good compression may still be achievable for floats.

Acknowledgments

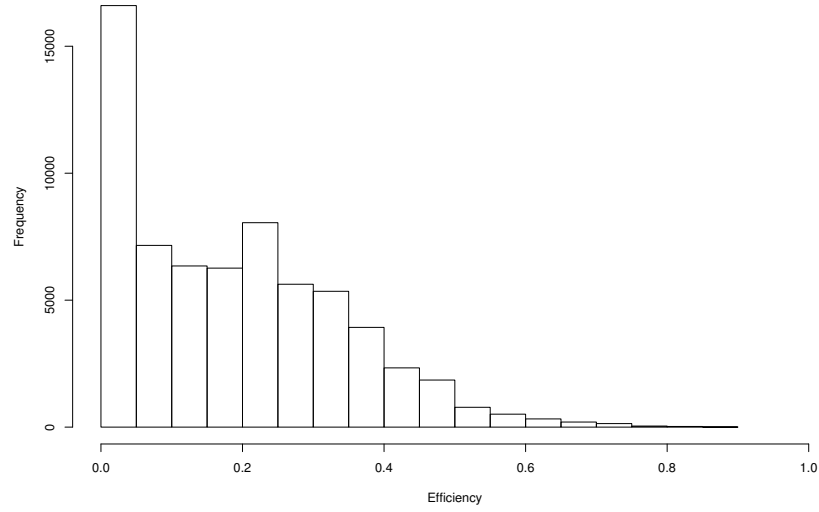
We would like to thank Paulo Cezar Carvalho and Moacyr silva for fruitful discussions and key ideas which helped to improve this paper.

References

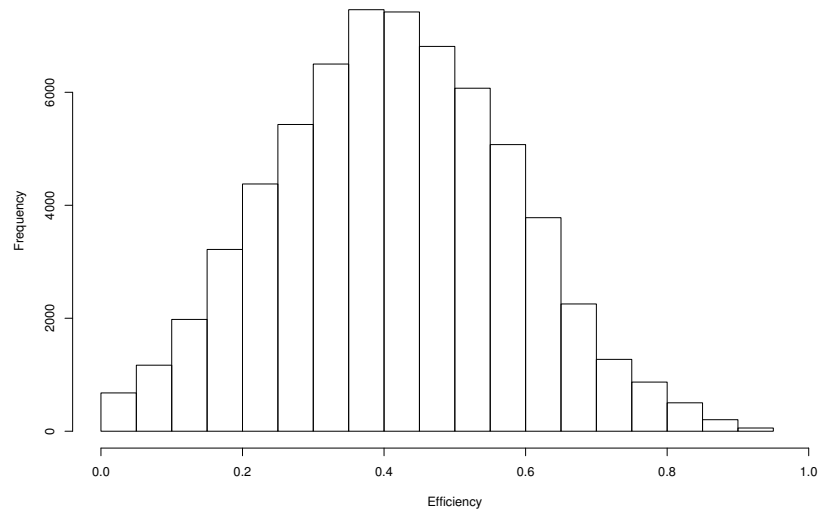
1. Salomon D, Motta G, Bryant D (2009) Handbook of Data Compression. London: Springer, 1359 pp.
2. Salomon D (2007) Data Compression: The Complete Reference. Number v. 10 in Data compression: the complete reference. London: Springer-Verlag New York Incorporated.
3. Lynch C (2008) Big data: How do your data grow? Nature 455: 28–29.
4. Yemliha T, Chen G, Ozturk O, Kandemir M, Degalahal V (2007) Compiler-directed code restructuring for operating with compressed arrays. In: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference: Embedded Systems. Citeseer, pp. 221–226.
5. Dodson D, Grimes R, Lewis J (1991) Sparse extensions to the fortran basic linear algebra subprograms. ACM Transactions on Mathematical Software (TOMS) 17: 253–263.
6. Van Der Walt S, Colbert S, Varoquaux G (2011) The numpy array: a structure for efficient numerical computation. Computing in Science & Engineering 13: 22–30.
7. Kane MJ, Emerson JW (2012) bigmemory: Manage massive matrices with shared memory and memory-mapped files. URL <http://CRAN.R-project.org/package=bigmemory>. R package version 4.3.0.
8. Bodden E, Clasen M, Kneis J (2007) Arithmetic coding revealed. In: Sable Technical Report 2007-5, Sable Research Group, School of Computer Science, (McGill University, Montréal.

Figure Legends

Tables

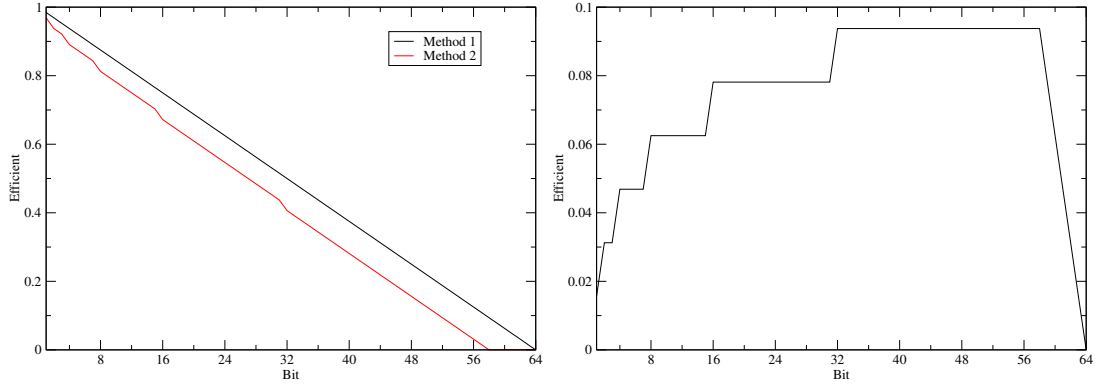


(a) Efficiency histogram of the method I



(b) Efficiency histogram of the method II

Figure 6. Efficiency histograms of the methods I and II. Note that the method II has a greater average efficiency than method I.



(a) Efficiencies of the methods I e II, for constant bit-length matrices.

(b) $\eta_1 - \eta_2$ for matrices of constant bit-length.

Figure 7. Compression efficiency of the methods I and II for matrices of constant bit-length.