Computable Compressed Matrices

Crysttian Arantes Paixão[1*], Flávio Codeço Coelho[2],
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. Examples of lossy data compression are MP3 (audio), JPEG (image) and MPEG (video). In this paper we discuss the use of lossless compression for numerical data structures such as numerical arrays to achieve compression without losing the mathematical properties of the original data.

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

Any kind of compression incurs 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 the space saved with compression must be available when data is decompressed for usage, thus partially negating the advantages of compression.

Most if not all existing lossless compression methods were developed under the following usage paradigm: *produce* → *compress* → *store* → *uncompress* → *use*. The focus of the present work is to allow a slightly different usage: *produce* → *compress* → *perform mathematical manipulations* → *decompress* (only for human reading).

With the growth of data volumes and analytical demands, creative solutions are needed to efficiently store as well as consume it on demand. This issue is present in many areas of application, ranging from business to science [3], and is being called the *Big Data* phenomenon. In the 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. This kind of analyses are what is now known as streaming data analysis [4]. Given a sufficiently dense stream of data, compression an decompression costs may become prohibitive. So having a way to compress data and keeping it compressed for the entire course of the analytical pipeline, is very desirable.

This paper will focus solely on numerical data which for the purpose of the applications is organized as matrices. This is a most common data structure found in computational data analysis environments. The matrices compressed according to the methodologies proposed here should be able to undergo the same mathematical operations as the original uncompressed matrices. e.g. linear algebra manipulations. This way, the cost of compression is reduced to a single event of compression and no need of decompression except when displaying the results for human reading. The idea of operating with compressed arrays is relatively new [5], and has yet to find mainstream applications to the field of numerical computations. One application which employs a form of compression is the sparse matrix linear algebra algorithms [6], in this case there is no alteration in the standard encoding of the data, but only the non-zero elements

of the matrices are stored and operated upon.

Larger than RAM data structures can render traditional analytical algorithms impracticable. Currently, the technique most commonly used when dealing with large matrices for numerical computations, is memory mapping [7, 8]. In memory mapping the matrix is allocated in a virtual contiguous address space which extends from memory into disk. Thus, larger than memory data structures can be manipulated as if they were in memory. This technique has a big performance penalty due to lower access speeds of disk when compared to RAM.

In this paper we present two methods for the lossless compression of (numerical) arrays. The methods involve the encoding of the numbers as strings of bits of variable length. The methods resemble the arithmetic coding [9] algorithm, but is cheaper to compute. We describe the process of compression and decompression, and study their efficiency under different applications. We also discuss the efficiency of the compression as a function of the distribution of the elements of the matrix.

Methods

Matrix compression

To maintain mathematical equivalence with the original data for any arithmetic operations, we need to maintain the structure of the matrix, i.e., the ability to acess any element given its row i and column j and also the numeric nature of its elements. In order to achieve compression we decided to exploit inefficiencies in the conventional way matrices are allocated in memory. The exemples in this paper will be restricted to matrices with positive integer elements.

The compression method is as follows. Let $M_{r\times c}$ be a matrix, in which r is the number of rows and c the number of columns. Each element of this matrix, called m_{ij}, is a positive integer. In digital computers, all information is stored as binary code (base 2 numbers). However the conventional way to store arrays of integers is on a sequence of memory blocks of fixed size (power of 2 number of bits), one for each element. The maximum size of a block is equal to the word size of the processor, which for most current CPUs is 64 bits. Some special number such as complex number may be encoded as two blocks instead of one. The size of the chunk of memory allocated to each number will determine their maximum size (for integers) or their precision (for floating-point numbes). So for matrix M, the total memory allocated, assuming chunks of 64 bits, is given by $\mathcal{B} = r \times c \times 64$.

The number of bits allocated $\mathcal{B}$, is larger than the absolute minimum number of bits required to represent all the elements of M, since smaller integers, when converted to base 2, require less digits. From now on, when the numerical base will be explicitly notated when necessary to avoid confusion between binary and decimal integers.

Let's consider an extreme example: a matrix composed exclusively of 0s and 1s (base 10). If the matrix type is set to 64-bit integers, 63 bits will be wasted per element of the matrix, since the minimum number of bits needed to store such a matrix is $\mathfrak{b} = r \times c \times 1$. The potential economy of bits ξ can be represented by $\xi = \mathcal{B} - \mathfrak{b} = r \times c \times 63$.

So it is evident that for any matrix whose greatest element requires less than 64 bits (or the fixed type of the matrix) to be represented, potential memory savings will grow linearly with the size of the matrix.

Method 1: The Supreme Minimum (SM)

The SM method consists in determining the value of the greatest element of matrix M, which coincides with its supremum, $maxM == \sup M$ and determine the minimum number of bits, $b(\sup M)$, required to store it. We will use capital roman letters to denote uncompressed matrices and the corresponding lower case letter for the compressed version.

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

The allocation of memory still happens in the usual way, i.e. in fixed size 64-bit chunks, only that now, in the space required for a single 64 bit integer, we can store for example, an entire 8×8 matrix of 0_{10} and 1_{10}.

Let's 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:

$$M = \begin{bmatrix} 900 & 1023 & 721 & 256 & 1 & 10 & 700 & 20 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{2}$$

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. From now on the minimum number of bits required to represent a base 10 integer will be refered to as its bit-length.

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

to store matrix M it first has to be converted to base 2 (M_2). Then it will be unraveled by column (column major, e.g. in Fortran) or by row (row major, e.g. in C) and its elements will be written as fixed size adjacent chunks of memory. The size of each chunk is determined by the type associated with the matrix (typically 64 bits, but always a power of 2).

According to the SM method, having 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. These 64-bit blocks will be called a bitstring. The remaining 4 bits will be used later. The number of bitstrings needed will be $\lfloor \frac{\dim(M) * b(\sup M)}{64} \rfloor + 1$.

The final layout of the first 6 elements of m in the first bitstring can be seen in 3.

$$bitstring_1 = 0000\underbrace{0000001010}_{10}\underbrace{0000000001}_{1}\underbrace{0100000000}_{256}\underbrace{1011010001}_{721}\underbrace{1111111111}_{1023}\underbrace{1110000100}_{900} \tag{3}$$

Here is a step-by-step description of the aplicationof the SM method to matrix M:

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.

2. Element $M_{1,2} = 1023$ is allocated in the second chunk, from bit 10 to bit 19.

3. Repeat for elements $M_{1,i}$ with $i = 1, \ldots, 6$ which are stored on the remaining chunks.

$$\underbrace{\underbrace{0000010100}_{20}\overbrace{101011}^{b}}_{bitstring_2} \mid \underbrace{\overbrace{1100}^{a}\underbrace{0000001010}_{10}\underbrace{0000000001}_{1}\ldots}_{bitstring_1}$$

4. Element 7 does not fit on the remaining 4 bits of the first bitstring. So it will straddle two bitstrings, i.e., it is divided in two segments a and b, a is written on the first bitstring b on the second.

$$\underbrace{\underbrace{0000010100}_{20}\overbrace{101011}^{b}}_{bitstring_2} \mid \underbrace{\overbrace{1100}^{a}\underbrace{0000001010}_{10}\underbrace{0000000001}_{1}\ldots}_{bitstring_1}$$

Please note that bitstrings are written from right to left.

$$\underbrace{0000010100}_{20}\underbrace{\overbrace{101011}^{a} \mid \overbrace{1100}^{b}}_{700}\underbrace{0000001010}_{10}\ldots$$

Thus the compressed matrix $m = M_2$ requires less memory than the conventional storage of M as a 64-bit integer array.

Method 2: Variable Length Blocks (VLB)

In the SM method, there is still waste of space since for elements smaller than the supremum, a number of bits remain unused.

In the VBL method, the absolute minimal number of bits are used to store each value. However, if we are going to divide the biststrings into variable length chunks, we also need to reserve some extra bits to represent the size of each chunk, otherwise the elements cannot be recovered once they are stored.

Lets use again the matrix described in equation 2, where the largest element is number 1023. Now instead of assigning one chunk of the bitstring to each element of m, we will assign two chunks: the first will store the number of bits required to store the element and the second will store the actual element. The first chunk will have a fixed size, in this case, 4 bits. These 4 bits are the required space to store the bit-length of $\sup M$, in this case, 10.

Lets go through VLB compression step-by-step. The largest element of M is 1023. Its bit-length is 10 which in turn is 4 bits long in base 2 (1010). Thus the fixed size chunk is 4 bits long for every element.

1. The first element $M_{1,1} = 900$ requires 10 bits to store, so we write 10 in the first chunk and 900 in the second.

$$bitstring_1 = 00000000000000000000000000000000000000000000000000\underbrace{\underbrace{1110000100}_{\text{element=900}}\ \underbrace{1010}_{\text{bit-length=10}}}_{M_{1,1}}$$

2. Do the same for the next element, $M_{1,2} = 1023$.

3. Element $M_{1,2} = 721$ is also added taking the bitstring to the state.

$$bitstring_1 = 00000000000000000000\underbrace{1011010001}_{721}\underbrace{1010}_{10}\underbrace{1111111111}_{1023}\underbrace{1010}_{10}\underbrace{1110000100}_{900}\underbrace{1010}_{10}$$

So far the VLB method is more wasteful than the SM, but when we add $M_{1,4} = 256$ we start to save some space.

4. Element $M_{1,4} = 256$ is added.
5. Elements $M_{1,5} = 1$ and $M_{1,6} = 10$ are added requiring a total of 13 bits instead of 20 with the SM method.

$$bitstring_1 = \underbrace{0100}_{4}\ \underbrace{1}_{1}\ \underbrace{0001}_{1}\ \underbrace{100000000}_{256}\ \underbrace{1001}_{9}\ \underbrace{1011010001}_{721}\ \underbrace{1010}_{10}\ \underbrace{1111111111}_{1023}\ \underbrace{1010}_{10}\ \underbrace{1110000100}_{900}\ \underbrace{1010}_{10}$$

$$bitstring_2 = 00000000000000000000000000000000000000000000000000000000000000\underbrace{1010}_{10}$$

6. The remaining two elements are added $M_{1,7} = 700$ and $M_{1,8} = 20$.

$$bitstring_2 = 00000000000000000000000000000000000\underbrace{10100}_{20}\underbrace{0101}_{5}\underbrace{1010111100}_{700}\underbrace{1010}_{10}\underbrace{1010}_{10}$$

We used a total of 87 bits to store matrix m with the VLB method instead of 80 bits using the SM method. However, as shall be seen later, the VLB method will be the most efficient for most matrices.

Compression Efficiency

Compression efficiency depends of the data being compressed. Below, a formula for calculating compression efficiency is derived for both methods. They will be based on the following ratio:

$$\eta = \frac{bits\,alocated - bits\,used}{bits\,alocated} \tag{4}$$

Where *bits alocated* above mean total bits required for standard storage of the matrix, without compression, while bits used mean total bits requires to store the matrix after compression.

SM Method

Let $M_{r\times c}$ be the matrix we wish to compress. In comparison with a conventional allocation (64-bit integers) we can apply equation 4 to caluculate the efficiency of the SM method:

$$\begin{aligned}\eta_1 &= \frac{64 \times rc - b(maxM) \times rc}{64 \times rc} \\ &= \frac{64 - b(maxM)}{64}\end{aligned} \tag{5}$$

As we see in 5, η_1 does not depend on size of the matrix, only on the bit-length of $\max M$. If $b(\max M) = 64$, η_1 is 0, i. e. no compression is possible. On the other extreme, if the matrix is composed exclusively of 0s and 1s, maximal compression is achievable, $\eta_1 = 1$.

VLB Method

For the VLB method, compression depends on the value of each element of the matrix. In this method bit-lentgh variability affects the compression ratio, so the formula will have to include this information.

Let the rc elements of the matrix $M_{r\times c}$ be divided into g groups, each with f_i numbers of bit-length $b_i = b(m_i)$. Thus f_i is the frequency of each bit-length present in M. Let $k = b(b(\max M))$, i.e., the bit length of the bit-length of $\max M$. The efficiency η_2 is shown below.

$$\eta_2 = \frac{64 \times rc - \sum_{i=1}^{g}(b_i + k) \times f_i}{64 \times rc} \tag{6}$$

We can further simplify equation 6 to get at shorter expression for the compression ratio.

$$\begin{aligned}
\eta_2 &= \frac{64 \times rc - \sum_{i=1}^{g}(b_i \times f_i + k \times f_i)}{64 \times rc} \\
&= \frac{64 \times rc - \sum_{i=1}^{g} b_i \times f_i - \sum_{i=1}^{g} k \times f_i}{64 \times rc} \\
&= \frac{64 \times rc - \sum_{i=1}^{g} b_i \times f_i - k \times \sum_{i=1}^{g} f_i}{64 \times rc} \\
&\sum_{i=1}^{g} f_i = rc \\
\eta_2 &= \frac{64 \times rc - \sum_{i=1}^{g} b_i \times f_i - k \times rc}{64 \times rc} \\
&= 1 - \frac{\sum_{i=1}^{g} b_i \times f_i}{64 \times rc} - \frac{k}{64}
\end{aligned} \tag{7}$$

Results

Random Matrix Generation. In both methods, compression eficiency depends on the distribution of the bit-lengths $b(m_{i,j})$. Thus, in this section, a method to generate a variety of random bit-length distributions is proposed.

For simplicity we will model the distribution of as a mixture X of two Beta distributions ,$B_1 \sim Beta(\alpha_1, \beta_1)$ and $B_2 \sim Beta(\alpha_2, \beta_2)$, whose probability function is shown in equation (eq.8). Since the Beta distribution is defined only in the interval $[0, 1] \subset \mathbb{R}$, we applied a simple transformation $(\lfloor 64 \times x \rfloor + 1)$ to the mixture in order to map it to the interval of $[1, 64] \subset \mathbb{Z}$

$$F(x) = w\, Beta(\alpha_1, \beta_1) + (1 - w)\, Beta(\alpha_2, \beta_2) \tag{8}$$

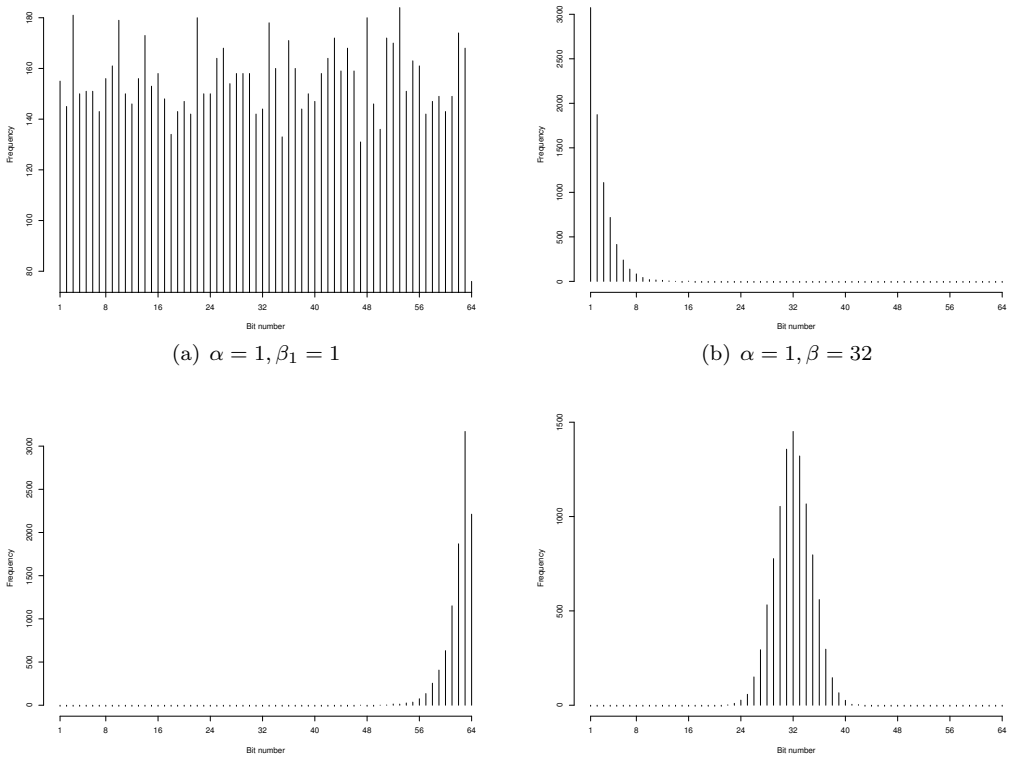
The intention of using this mixture was to find a simple way to represent a large variety of bit length distributions. The first two central moments of this mixture are given in 9 and will be used later to summarize our numerical results.

$$\begin{aligned}
E(X) &= wE(B_1) + (1 - w)E(B_2) \\
&= w\frac{\alpha_1}{\alpha_1 + \beta_1} + (1 - w)\frac{\alpha_2}{\alpha_2 + \beta_2} \\
Var(X) &= wVar(B_1) + (1 - w)Var(B_2) + w(1 - w)(E(B_1)^2 - E(B_2)^2)
\end{aligned} \tag{9}$$

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.

From now on we will apply equations 5 and 7, to determine the compression efficiency of methods SM and VLB for random matrices generated as describe above.

With $w = 0$, a single Beta distribution is used. In Figure 1, we show some distributions of bit-lengths for some combinations of α_1 and β_1. From the figure it can be seen that a large variety of unimodal distributions can be generated in the interval $[1, 64]$.



(a) $\alpha = 1, \beta_1 = 1$

(b) $\alpha = 1, \beta = 32$

(c) $\alpha = 32, \beta = 1$

(d) $\alpha = 64, \beta = 64$

Figure 1. 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 the from set of all possible distributions of bit-length, represented byt 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 depend on it. So, from equations 5 and 7, the expected efficiencies become:

$$E(\eta_1) = 1 - \frac{k}{64} \tag{10}$$

$$E(\eta_2) = 1 - \frac{E(b)}{64} - \frac{k}{64} \tag{11}$$

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 10, 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 10 and 11. the values presented in tables and figures below are averages of the efficiencies of the three samples.

In Figure 2, 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 2 that method 2 is more efficient in the majority of cases.

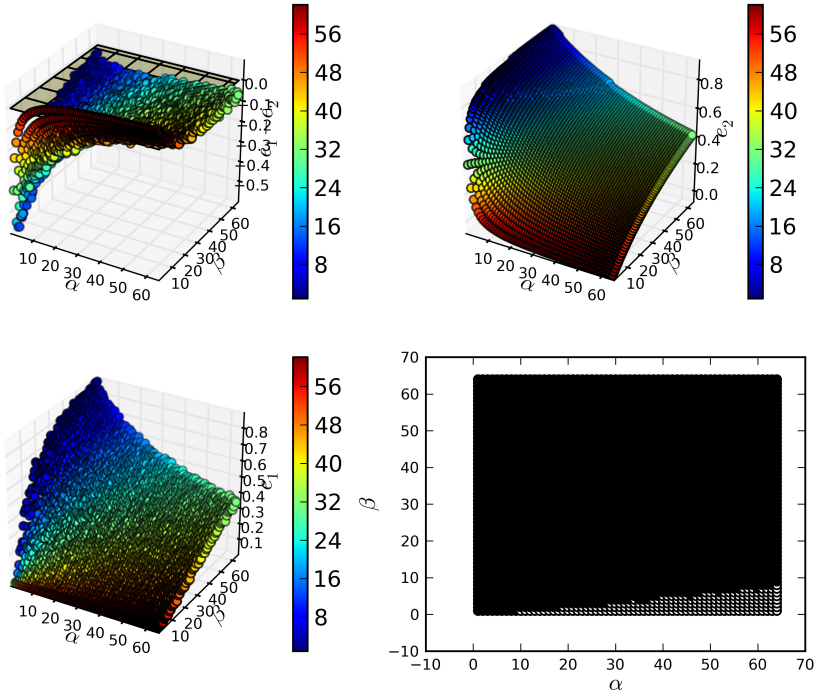


Figure 2. Comparing compression efficiency of methods 1 and 2. Color scale in (a), (b) and (c) represent average bit-length. In (a) we can see the difference $D = \eta_1 - \eta_2$. It can be seen that for most combination of α and β, $D < 0$, meaning the second method is more efficent to compress a sample 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 12.

$$E(B) = 0.5E(B_1) + 0.5E(B_2) \tag{12}$$

We must also redefine the calculations of efficiency for the mixture samples. Equation 13 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 10 and 11), it will be a function of $max\{E(B_1), E(B_2)\}$. We do the same for method 2 (equation 14).

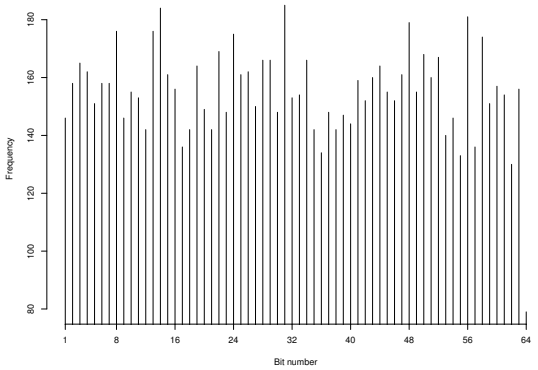
$$E(\eta_1) = 1 - \frac{max\{E(B_1), E(B_2)\}}{64} \tag{13}$$

$$E(\eta_2) = 1 - 0.5\frac{E(B_1)}{64} - 0.5\frac{E(B_2)}{64} - \frac{max\{E(B_1), E(B_2)\}}{64} \tag{14}$$

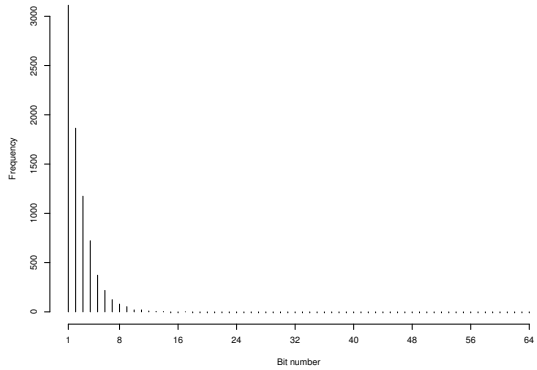
As before, we generate 3 samples of size 10000 for each parameterization, calculate the average efficiencies (equations 13 and 14) and their diference D.

Before we move on to the efficiency results and analyses, let's first inspect our samples from the mixture of transformed Beta distributions. Figures 3 and 4, 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.

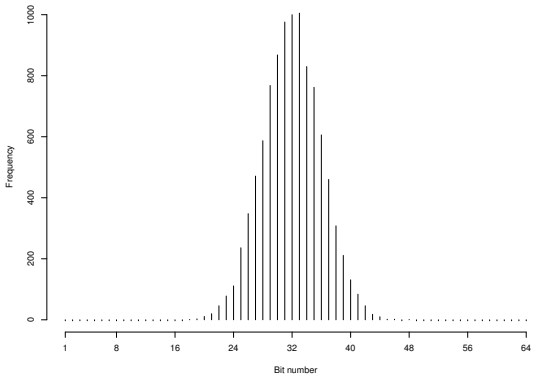
After sampling uniformly ($[1, 5, 9, \ldots, 64]$, $n = 65536$) the parameter space and comparing efficiencies, we summarized the results on table 2. In it we see how many parameterizations (from our sample) favor each method. We can also look at the distribution of efficiencies on our samples for each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



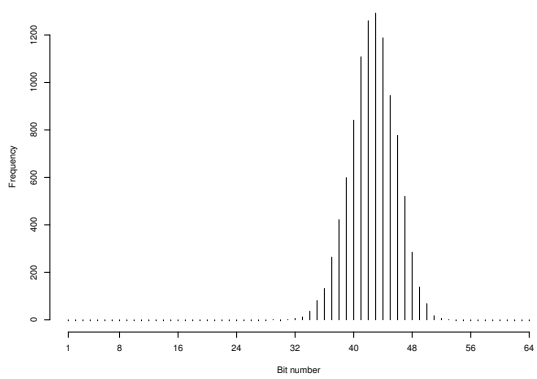
(a) $\alpha_1 = 1, \beta_1 = 1, \alpha_2 = 1, \beta_2 = 1$



(b) $\alpha_1 = 1, \beta_1 = 32, \alpha_2 = 32, \beta_2 = 1$

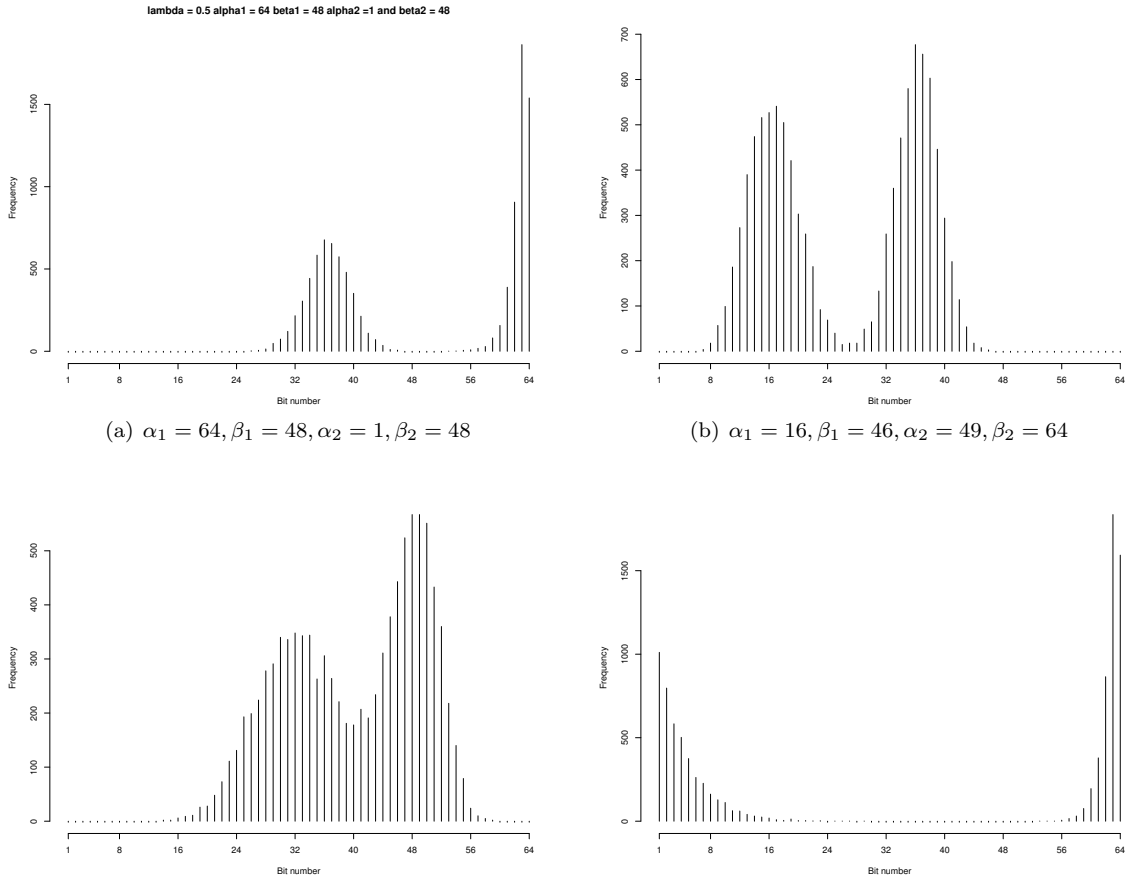


(c) $\alpha_1 = 32, \beta_1 = 32, \alpha_2 = 32, \beta_2 = 32$



(d) $\alpha_1 = 64, \beta_1 = 32, \alpha_2 = 32, \beta_2 = 64$

Figure 3. 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..



(a) $\alpha_1 = 64, \beta_1 = 48, \alpha_2 = 1, \beta_2 = 48$

(b) $\alpha_1 = 16, \beta_1 = 46, \alpha_2 = 49, \beta_2 = 64$

(c) $\alpha_1 = 16, \beta_1 = 16, \alpha_2 = 16, \beta_2 = 49$

(d) $\alpha_1 = 1, \beta_1 = 16, \alpha_2 = 1, \beta_2 = 49$

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.

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 6(a) and 6(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 7, 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 7. After plugging in our numbers, we obtain a compression of 38.29%.

$$\eta_2 = 1 - \frac{\sum_{i=1}^{2} b_i \times f_i}{64 \times rc} - \frac{7}{64}$$

$$\eta_2 = 1 - \frac{1 \times \frac{rc}{2} + 64 \times \frac{rc}{2}}{64 \times rc} - \frac{7}{64}$$

$$\eta_2 = 1 - \frac{32.5 \times rc}{64 \times rc} - \frac{7}{64}$$

$$\eta_2 \approx 1 - 0.5078 - 0.1093$$

$$\eta_2 \approx 38.29\%$$

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 7, which becomes 15. In Equation 15, the $\frac{f_i}{rc}$ is replaced by p_i, representing the probability of elements from group i in matrix M.

$$\eta_2 = 1 - \frac{\sum_{i=1}^{g} b_i \times p_i}{64} - \frac{k}{64} \tag{15}$$

with

$$p_i = \frac{f_i}{rc} \tag{16}$$

With the equation 15 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 17.

$$\eta_2 = 1 - \frac{1 \times p_1 + 64 \times p_2}{64} - \frac{7}{64} \tag{17}$$

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 18 and 19. The first equation on both systems come from the law of total probability. The second comes from 17 after setting η_2 to 1 and 0, respectively.

$$\eta_2 = 1 : \begin{cases} p_1 + p_2 = 1 \\ p_1 + 64p_2 = 7 \end{cases} \tag{18}$$

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%.

$$\eta_2 = 0 : \begin{cases} p_1 + p_2 = 1 \\ p_1 + 64p_2 = 57 \end{cases} \tag{19}$$

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 15).

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 $\{1, 2, 3, \ldots, 64\}$ with equal probability, i.e., $\frac{1}{64}$.

For the Poisson distribution, we use Poisson(λ) with λ being the expected bit-length in the sample.

To calculate the efficiency, we generated a sample with 100 ($M_{10\times10}$), 10,000 ($M_{100\times100}$) and 1,000,000 ($M_{1,000\times1.000}$) 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 comming 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 $Bin(n, p = 0.5)$, 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]

	Efficiency		
	Sample size		
Expected bit-length ($n/2$)	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 Poisson(λ), 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]

	Efficiency		
	Size sample		
Expected bit-length (λ)	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 15), we have

$$E(\eta_2) = 1 - \frac{E(B)}{64} - \frac{k}{64} \tag{20}$$

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

$$E(\eta_2) = 1 - \frac{\frac{1+64}{2}}{64} - \frac{7}{64} \approx 38.28\% \tag{21}$$

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 eficiency becomes (with $k = 7$):

$$\begin{aligned} E(\eta_2) &= 1 - \frac{64 \times p}{64} - \frac{k}{64} \\ &= 1 - \frac{64 \times 0.5}{64} - \frac{7}{64} \approx 39.05\% \end{aligned} \tag{22}$$

Which again agrees with estimates in Table 5.

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

$$E(\eta_2) = 1 - \frac{\lambda}{64} - \frac{k}{64} \tag{23}$$

$$= 1 - \frac{32}{64} - \frac{7}{64} \approx 39.06\% \tag{24}$$

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 varincia?]

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 Claudia Torres Codeo, 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. Gaber M, Zaslavsky A, Krishnaswamy S (2005) Mining data streams: a review. ACM Sigmod Record 34: 18–26.

5. 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.

6. 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.

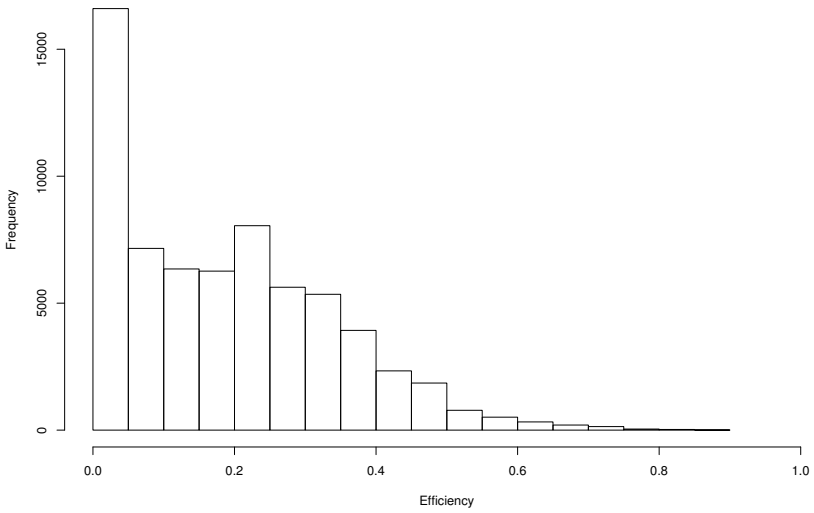
7. Van Der Walt S, Colbert S, Varoquaux G (2011) The numpy array: a structure for efficient numerical computation. Computing in Science & Engineering 13: 22–30.

8. 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.

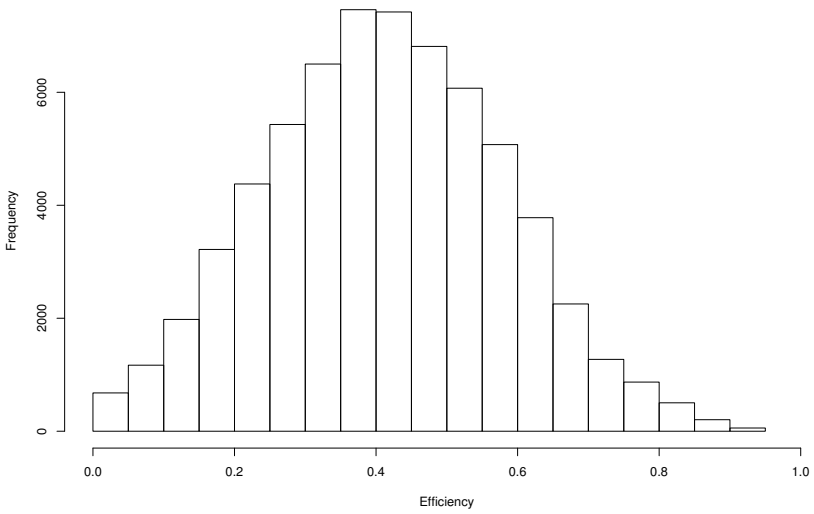
9. Bodden E, Clasen M, Kneis J (2007) Arithmetic coding revealed. In: Sable Technical Report 2007-5, Sable Research Group, School of Computer Science, (McGill University, Montréal.

Figure Legends

Tables

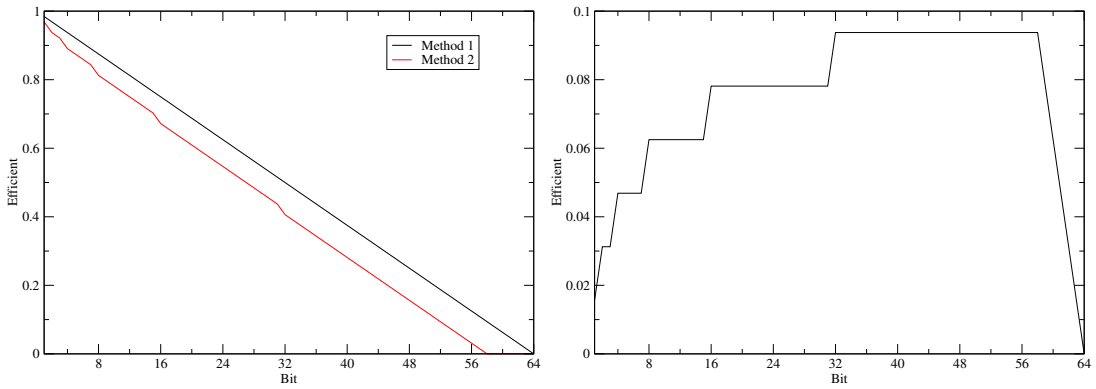


(a) Efficiency histogram of the method I



(b) Efficiency histogram of the method II

Figure 5. 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 6. Compression efficiency of the methods I and II for matrices of constant bit-length.