

# Computable Bitstring-compressed Matrices

Crysttian Arantes Paixão<sup>1\*</sup>, Flávio Codeço Coelho<sup>2</sup>,

**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 [crysttian.paixao@fgv.br](mailto:crysttian.paixao@fgv.br)**

## Abstract

The biggest cost of computing with large matrices in any modern computer is related to memory latency and bandwidth. The average latency of modern RAM reads is 150 times greater than a clock step of the processor [1]. Throughput is a little better but still 25 times slower than the CPU can consume. The application of bitstring compression allows for larger matrices to be moved entirely to the cache memory of the computer, which has much better latency and bandwidth (average latency of L1 cache is 3 to 4 clock steps). This allows for massive performance gains as well as the ability to simulate much larger models efficiently. In this work, we propose a methodology to compress matrices in such a way that they retain their mathematical properties. Considerable compression of the data is also achieved in the process. Thus allowing for the computation of much larger linear problems within the same memory constraints when compared with the traditional representation of matrices.

## Author Summary

## Introduction

Data compression is traditionally used to reduce storage resources usage and/or transmission costs [2]. 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 [3].

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*  $\rightarrow$  *compress*  $\rightarrow$  *store*  $\rightarrow$  *uncompress*  $\rightarrow$  *use*. The focus of the present work is to allow a slightly different usage: *produce*  $\rightarrow$  *compress*  $\rightarrow$  *perform mathematical manipulations* and *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 [4], 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 [5]. Given a sufficiently dense stream of data, compression and 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 [6], and it 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 [7], 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 [8,9]. 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 [10] 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. Also presented are some results in relation to the traditional method of matrix allocation. We also consider the arithmetic operations to measure the efficiency of the proposed methodology.

Inseri isso

## Methods

### Matrix compression

To maintain mathematical equivalence with the original data for any arithmetic operations, we need to maintain the structure of the matrix, ie, the ability to access 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 analyzes presented in this article, in the first moment, are related to arrays of positive integers, but can be applied to integers and real numbers with some adaptations, which are presented in due course.

adicionei aqui

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 memory block sequence of fixed size (power of 2 numbers of bit), 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 numbers). So for matrix  $M$ , the total memory allocated, assuming chunks of 64 bits, is given by  $\mathcal{B} = r \times c \times 64$ .

removi isso  
The examples in this paper will be restricted to matrices with positive integer elements, but

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.







$$\begin{aligned}\eta_1 &= \frac{64 \times rc - b(\max M) \times rc}{64 \times rc} \\ &= \frac{64 - b(\max M)}{64}\end{aligned}\quad (5)$$

As we see in 5,  $\eta_1$  does not depend on size of the matrix, only on the bit-length of  $\max M$ . If  $b(\max M) = 64$ ,  $\eta_1$  is 0, ie, no compression is possible. On the other extreme, if the matrix is composed exclusively of 0s and 1s, maximal compression is achievable,  $\eta_1 = 0.9843$ .

### VLB Method

For the VLB method, compression depends on the value of each element of the matrix. In this method bit-length 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  $k = b(b(\max M))$ , ie, the bit length of the bit-length of  $\max M$ . The efficiency  $\eta_2$  is shown below.

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

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

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

Knowing that

$$\sum_{i=1}^g f_i = rc,$$

we can simplify the equation above, obtaining (7).

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

For this method, the highest value obtained for the data compression is 0.9687, when  $b = 1$  and  $k = 1$ . However, the lower value is defined by a range of values, such that the value of  $k = \frac{\sum_{i=1}^g b_i \times f_i}{rc} - 64$ .

## Results

**Random Matrix Generation.** In both methods, compression efficiency 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.

Altereí a proporção para 0.98, pois o maior valor que pode ser obtido.

Adicionei esse trecho para ficar igual ao SM

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 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  $[1, 64] \subset \mathbb{Z}$ .

$$f(x) = w \text{Beta}(\alpha_1, \beta_1) + (1 - w) \text{Beta}(\alpha_2, \beta_2) \quad (8)$$

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 it will be used later to summarize our numerical results.

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

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 SM and VLB methods for random matrices generated as describe above.

With  $w = 0$ , a single Beta distribution is used. In Figure 3, we show some distributions of bit-lengths for some combinations of  $\alpha_1$  and  $\beta_1$ . From the figure it can be seen that a large variety of unimodal distributions can be generated in the interval  $[1, 64]$ .

As we are sampling from a large set 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 5 and 7, the expected efficiencies become:

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

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

where  $k$ , in (11), 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 SM method while a negative  $D$  favors VLB method.

The expected compression efficiency in the following numeric experiments, will be calculated from 3 matrices of dimension 10000, generated as described, and presented in tables and figures below.

In Figure 4, we can see the distribution of efficiencies and their difference for a sample generated from a single Beta distribution of bit-lengths. Note that both methods can achieve efficiencies greater than 80% for matrices with very small numbers. Also note that the VLB method is more efficient in the majority of cases.

Now let  $w = 0.5$ , ie, matrices will have elements with bit-lengths coming from a mixture of beta distributions,  $B_1 \sim \text{Beta}(\alpha_1, \beta_1)$  and  $B_2 \sim \text{Beta}(\alpha_2, \beta_2)$ . The expected value for this mixture is shown in Equation 12.

$$E(B) = 0.5E(B_1) + 0.5E(B_2) \quad (12)$$

For bit-lengths coming from a mixture ( $w > 0$ ), let the expected efficiencies for the SM and VLB methods be as given by Equations 13 and 14. So now, instead of having the efficiency be a function of greatest bit-length in the sample (denoted as  $k$  in 5 and 7), it will be a function of  $\max\{E(B_1), E(B_2)\}$ .

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

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

As before, we generate 3 matrices of dimension 10,000 for each parameterization, calculate the average efficiencies (Equations 13 and 14) and their difference  $D$ .

Before moving on to efficiency results and analyses, let's first inspect samples from the mixture of transformed Beta distributions. Figures 5 and 6, 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 ( $[1, 5, 9, \dots, 64]$ ,  $n = 65,536$ ) the bit-length 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 7), which clearly demonstrate the greater expected efficiency of method VLB (Figure 7(b)).

As we have shown, the VLB method is more effective compressing most integer datasets up to 64 bits 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, SM method will always be better, regardless of bit-length (Figures 8(a) and 8(b)). There are only exception, if for bit-length 64, where neither method is able to compress the data.

Altereí a  
ltima frase

## Discussion

### Calculating Efficiencies

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, the SM Method is better, otherwise, the VLB 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  $\eta_2$  can be calculated using Equation 7. After plugging in our numbers, we obtain a compression of 38.29%.

alterei de  
best para  
better

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

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

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

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



$$\eta_2 \approx 38.29\%$$

The efficiency of the VLB method 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.

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

with

$$p_i = \frac{f_i}{rc} \quad (16)$$

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.

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

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  $\eta_2$ , 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  $\eta_2$  to 1 and 0, respectively.

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

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%.

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

Thus, when  $p_1 = 0.1111$  and  $p_2 = 0.8889$  the efficiency is minimal for the VLB 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.

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 groups with bit-lengths 1, 32 and 64 bits, efficiency  $\eta_2 = 0.3854$ ,
- 5 groups with bit-lengths 1, 16, 32, 48 and 64 bits, efficiency  $\eta_2 = 0.3875$ ,
- 8 groups with bit-lengths 1, 8, 16, 24, 32, 40, 48, 56 and 64 bits, efficiency  $\eta_2 = 0.3888$

Acho que  
tinha que  
tirar esse  
Let's also,  
talvez colo-  
car Now, we  
are going to

When the distribution of the group probabilities is uniform, ie, 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

Let the bit-lengths of the matrices be distribute according to the Uniform distribution  $U(a = 1, b = 64)$ , which means bit-lengths may take values in the set  $\{1, 2, 3, \dots, 64\}$  with equal probability, ie,  $\frac{1}{64}$ .

**Theoretical Efficiency:** 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 VLB method (Equation 15), we have

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

assuming all bit-lengths are possible, i.e.,  $a = 1$  and  $b = 64$ , and hence  $k = 7$ , we can calculate  $\eta_2$ :

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

This result agrees with the numerical estimates presented in Table 4.

**Numerical Estimates:** To calculate the VLB efficiency, we generated a matrices with 100 ( $M_{10 \times 10}$ ), 10,000 ( $M_{100 \times 100}$ ) and 1,000,000 ( $M_{1,000 \times 1,000}$ ) elements with 1, 8, 16, 32 and 64 number of bits. The average efficiency (Table 3) is calculated from a 1,000 replicates of each matrix size. As expected the compression efficiency gets better with lower expected bit-length.

## Binomial Distribution

For the binomial distribution, we will use  $Bin(n, p)$ , with the number of trials  $n$  representing the greatest possible bit-length in the matrix, and  $np$  giving us the expected bit-length.

**Theoretical Efficiency:** Let bit-length ( $B$ ) be a random variable with Binomial distribution,  $B \sim B(n = 64, p = 0.5)$ ,  $E(b_i) = \sum_i b_i \times p(b_i) = n \times p$  and the efficiency becomes (with  $k = 7$ ):

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

Which again agrees with estimates in Table 5.

**Numerical Estimates:** 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 matrix 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 bit-lengths lead to higher compression efficiencies.



Observe que os bits posicionados mais esquerda devem ser utilizados para informar que o número é negativo. Para efeito de otimização da biblioteca, foi escolhido a conversão dos números negativos para positivos e a utilização de 1 bit para representar o sinal.

In turn, in order to represent real numbers, a conversion procedure of real numbers to integer numbers was used. The real number is multiplied by a power of ten and is then immediately truncated. The real numbers in a binary system of 64 bits are represented using a bit to represent the sign of the number, 11 bits to the exponent, and 52 bits to the mantissa. Note that 1 bit is still used implicitly, according to the IEEE-754 standard. In summary, the mantissa is represented in the base  $2^{-1}, 2^{-2}, 2^{-3}, \dots, 2^{-52}$ . Depending on the value to be represented, loss of information can take place, which is related to the accuracy of the type used. In this case, considering a real number with double precision, the demand of 64 bits to represent each one occurs, and in some cases, the number to be represented is an approximation. As an example, consider that the number to be stored is 1.109. Following the traditional system, the number 1,109 is represented by

0011111111110001101111100111011011001000101101000011100101011000

We consider that the power used is  $10^3$ , that is, the number which needs to be stored would be 1109. In binary, that number is represented by 10001010101. Note that the number of bits required is smaller, but depending on numerical accuracy, the compression methods suggested are ineffective. When the power of 10 is used, it must be applied across the matrix. This conversion allows the proposed methods to be applied to a set of real numbers. It should be taken into account that the greater the power of 10 used, which is associated with the conversion of the real number, the larger the set of bits will be required to represent the converted number. This affects the efficiency of data compression. The use of the library to represent real numbers is conditioned to a study of the necessary level of precision to solve the problem.

## Library Application

In order to create an application, some elementary matrix operations. Different operations have been performed by measuring the execution time of each. We basically, considered the attribution of a constant to all the elements of a matrix; the addition, subtraction and multiplication of the elements of a matrix by a constant, the addition, subtraction and multiplication of matrices. In addition to that, the calculation of the transposed matrix and the maximum an array were also considered. This measure was used to evaluate the efficiency of the method implemented when compared to traditional methods.

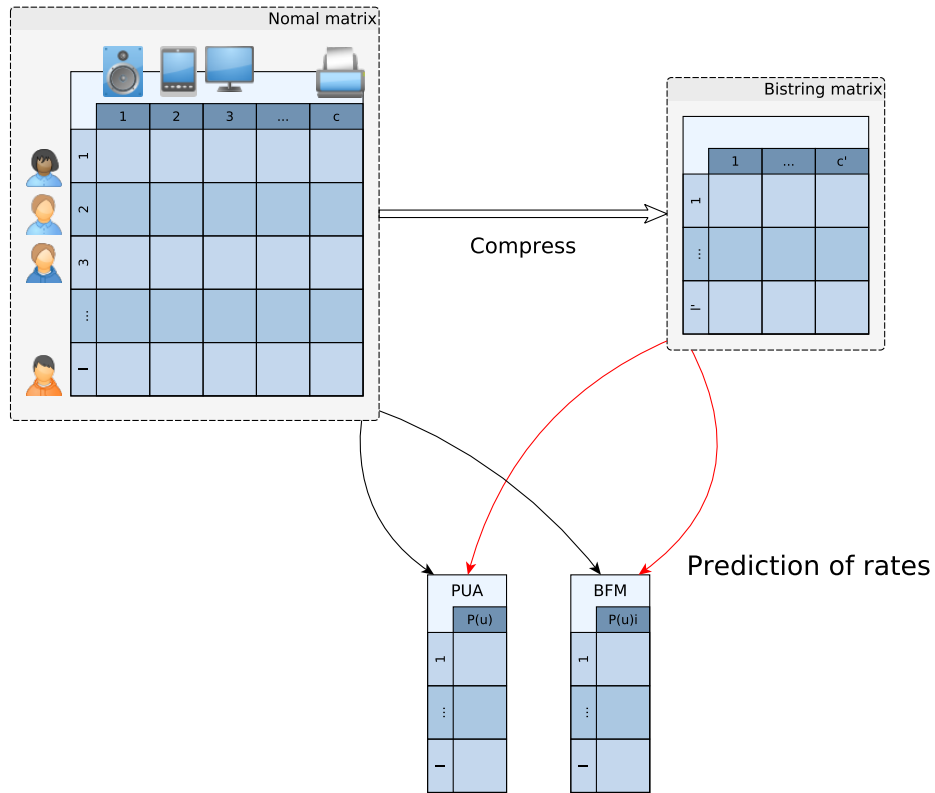
One of the characteristics of bitstring matrices is that the operations per column, in the case of implementation in Fortran, can be parallelized. This reduces the execution time, compared to traditional methods. Therefore, in order to achieve this optimization, threads implementations were used for the implemented operations.

As a test methodology, all the described operations were also compared to traditional methods. The matrix dimensions  $n \times n$  used were  $n = 1,000, 6,000, 11,000$  to  $20,000$ . We emphasize that the first matrix, created with a 1,000 dimension has  $10^6$  elements, while the one with a 20,000 dimension has  $4 \times 10^8$  elements. With the increase of the size of the matrices, a larger amount of memory must be reserved to allocate the numbers. The compression effect of the matrices can be verified in the results since by the proposed method the matrix is directly loaded into the main memory, avoiding the need for access to the secondary memory.

Another library application was the implementation of the algorithm in order to deal with the problem of arrays of collaborative filtering [11]. These arrays have large dimensions due to the number of users and products. Each element of this matrix stores the rating of a user with respect to a specific product. With this information, it is possible to specify a product for a determined customer.

The manipulation of these matrices, depending on the amount of memory to be allocated, are one of the possible applications of the methods proposed in this work, since it promotes the compression and data allocation in the main memory. The allocation of the matrix within the main memory makes the access time to the elements much smaller than when the matrix is stored in secondary memory. This latter type of storage is used in some already mentioned techniques. As an example, let us examine the manipulation of a CF matrix, evaluating the access time to the elements and especially the size of the memory required to run such operation. The algorithms applied to this matrix were the average per-use and the bias from mean [11], this last one of the simplest algorithms for predicting the rate of users.

An example of the application of the methodology presented is shown in Figure 1. In this figure, the matrix with normal and bitstring formats are displayed. In both matrices, the information is represented and can be accessed. Therefore, it becomes possible to apply the selected algorithms.



**Figure 1.** Illustration of the process of the bitstring model application to matrices of collaborative filtering of the algorithms per use average (PUA) and of the bias from mean (BFM) in a matrix represented in the traditional and bitstring formats.

Falar dos dados do GraphLab, os usados foram o do netflix, e medir o tempo mdio da executacao por umas 10 vezes.

Os dados utilizados nos testes realizados esto disponveis em XXX. Esses dados correspondem a ....

## Conclusion

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 [12].

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 [13].

For the compression calculations presented in this paper we limited bit-length of integers to 64 bits. However the compression would work in the same way as discussed for computer architectures with larger word sizes.

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 Codeço, Moacyr Silva and Paulo Cezar Carvalho for fruitful discussions and key ideas which helped improve the manuscript.

## References

1. Alted F (2010) Why modern cpus are starving and what can be done about it. *Computing in Science & Engineering* 12: 68–71.
2. Salomon D, Motta G, Bryant D (2009) *Handbook of Data Compression*. London: Springer, 1359 pp.
3. Salomon D (2007) *Data Compression: The Complete Reference*. Number v. 10 in *Data compression: the complete reference*. London: Springer-Verlag New York Incorporated.
4. Lynch C (2008) Big data: How do your data grow? *Nature* 455: 28–29.
5. Gaber M, Zaslavsky A, Krishnaswamy S (2005) Mining data streams: a review. *ACM Sigmod Record* 34: 18–26.
6. 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.
7. 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.
8. 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.
9. 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.

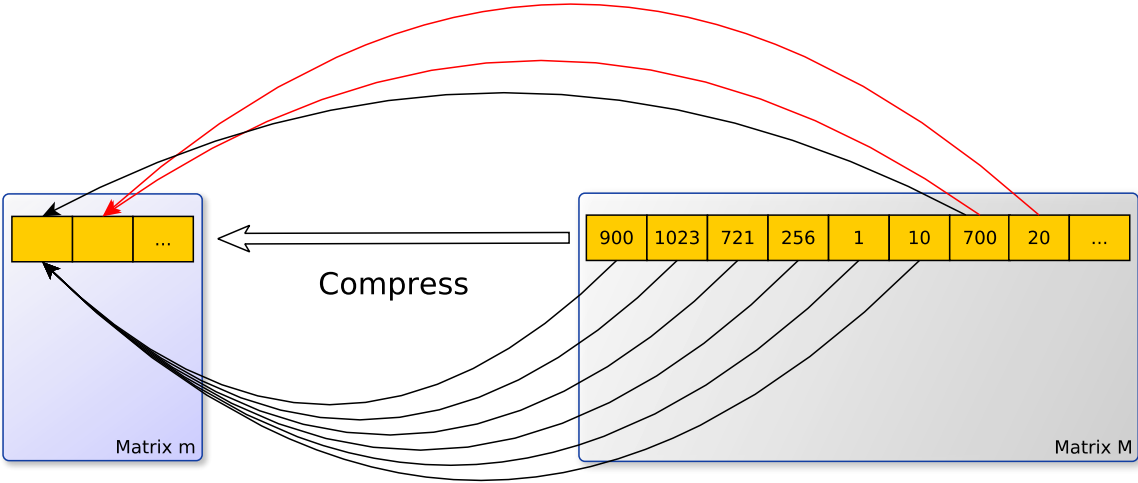
10. Bodden E, Clasen M, Kneis J (2007) Arithmetic coding revealed. In: Sable Technical Report 2007-5, Sable Research Group, School of Computer Science, (McGill University, Montréal.

11. Lemire D, Maclachlan A (2007) Slope one predictors for online rating-based collaborative filtering. CoRR abs/cs/0702144.

12. Paixão CA (2012) Bitstring model for study of the propagation Dengue fever. Phd thesis, Federal University of Lavras.

13. Dongarra JJ (1979) LINPACK users' guide. 8. Siam.

Figure Legends

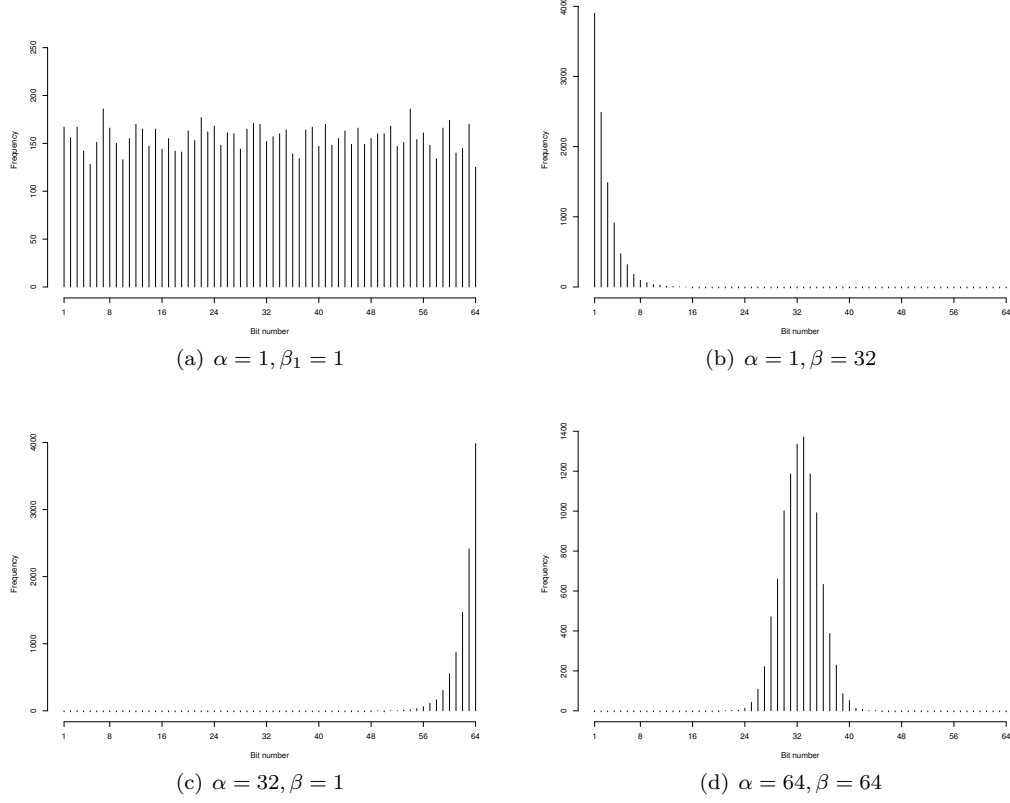


**Figure 2.** Representation of the compression process and allocation of elements of matrix M in the matrix m. The strip of value 700 is divided in two parts and stores in different strips in the matrix m.

Tables

Adicionei a  
ltima frase

A escala  
de cor a  
mesma em  
todas as  
figuras, no  
texto estava  
dizendo que  
era apenas  
na a, b e c.  
Eu alterei.

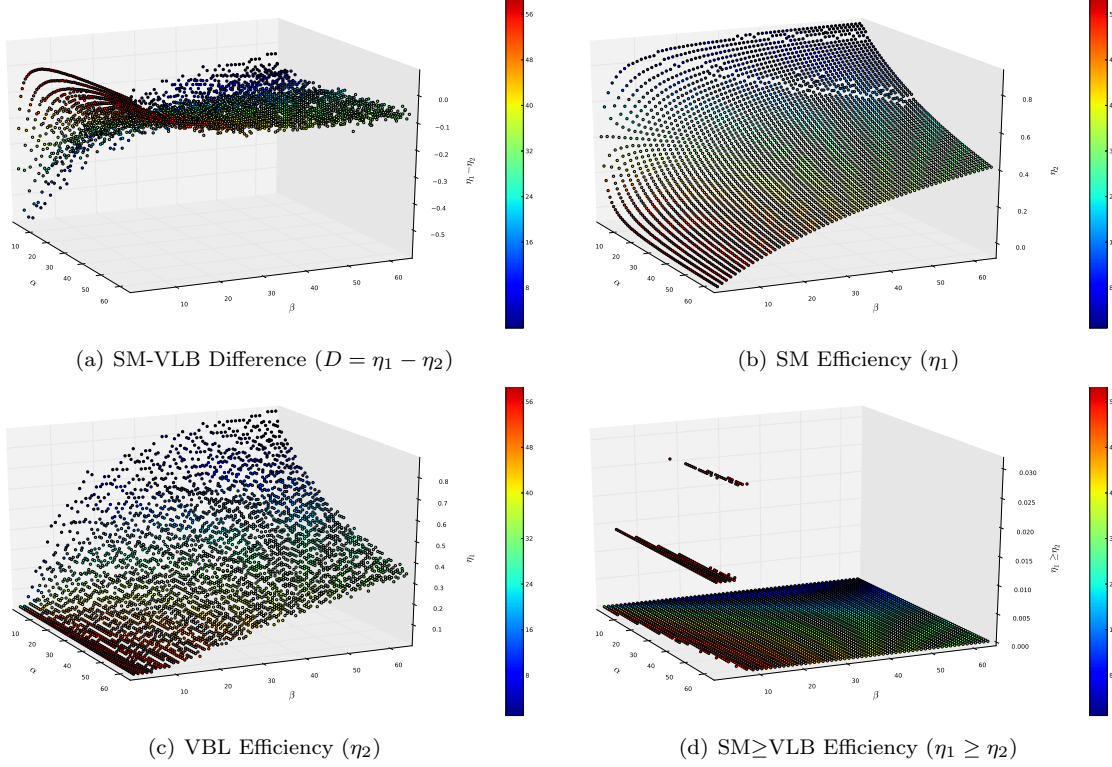


**Figure 3.** Histograms constructed from samples with 10,000 elements, generated from a Beta distribution. Below each histogram is possible to verify the parameters used. With these two parameters it is possible to perform different combinations of numbers to fill the arrays to be compressed.

**Table 1.** The first 8 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          |

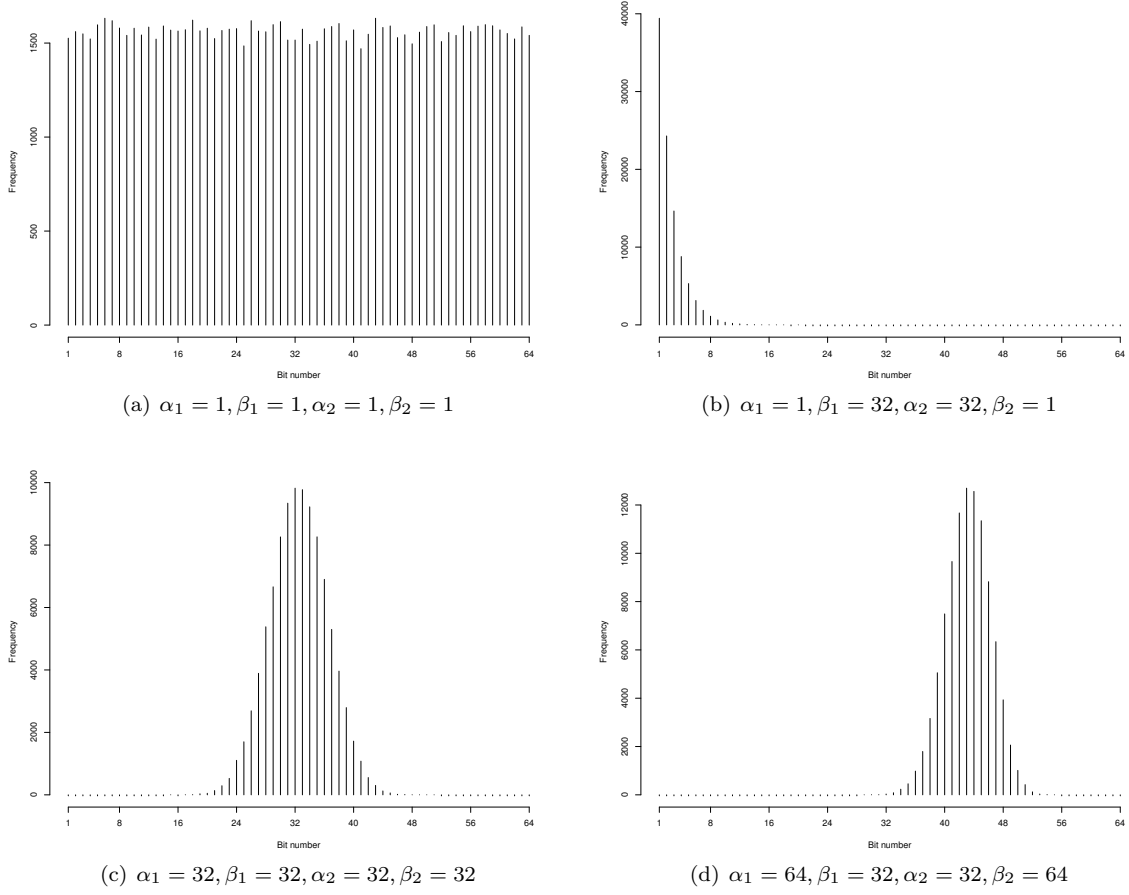




**Figure 4.** Comparing compression efficiency of methods 1 and 2. Color scale in represent average bit-length. In (a) we can see the difference  $D = \eta_1 - \eta_2$ . It can be seen that for most combination of  $\alpha$  and  $\beta$ ,  $D < 0$ , meaning the second method is more efficient to compress 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 SM 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 SM and VLB methods, respectively.

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

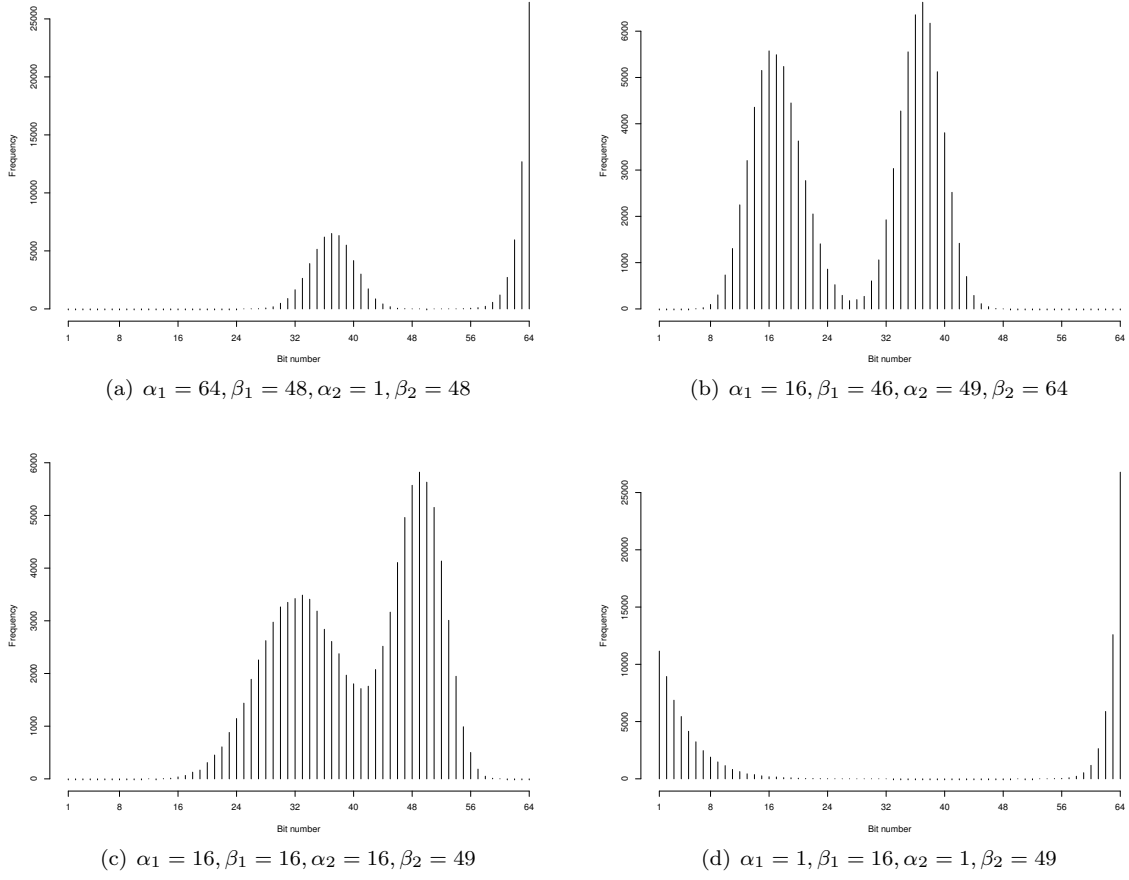
| Methods | n     | Percentage |
|---------|-------|------------|
| SM      | 592   | 0.9034%    |
| VLB     | 64944 | 99.0966%   |
| Total   | 65536 | 100%       |



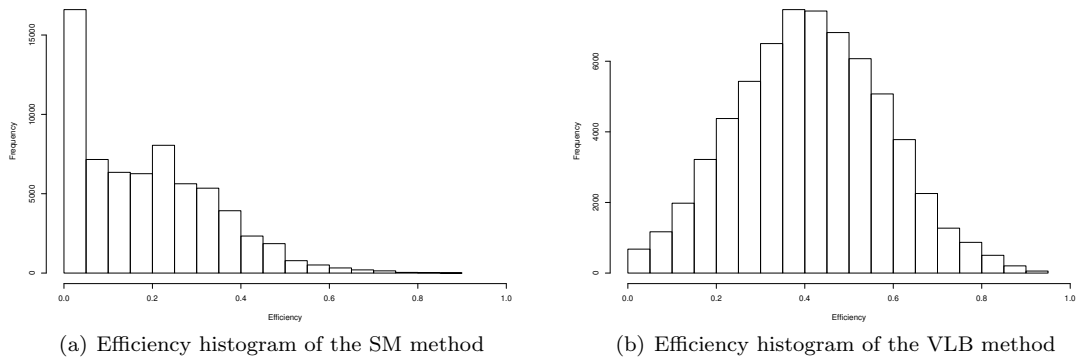
**Figure 5.** Histograms constructed from samples with 10,000 elements, generated from the mixture of two Beta distributions with  $w = 0.5$ . Below each histogram are the parameters of the mixture.

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

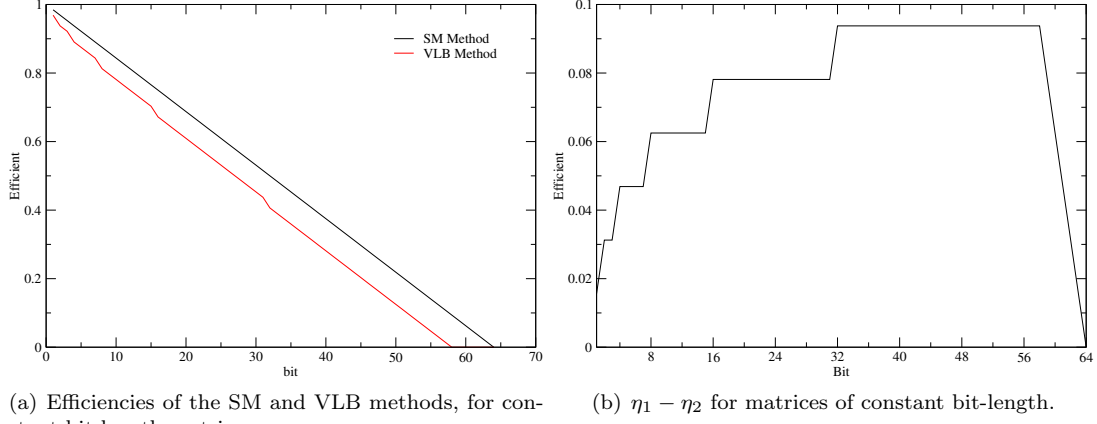
| $p_1$ | $p_2$ | $\eta_2$ |
|-------|-------|----------|
| 0.0   | 1.0   | -0.109   |
| 0.1   | 0.9   | -0.010   |
| 0.2   | 0.8   | 0.087    |
| 0.3   | 0.7   | 0.185    |
| 0.4   | 0.6   | 0.284    |
| 0.5   | 0.5   | 0.382    |
| 0.6   | 0.4   | 0.481    |
| 0.7   | 0.3   | 0.579    |
| 0.8   | 0.2   | 0.678    |
| 0.9   | 0.1   | 0.776    |
| 1.0   | 0.0   | 0.875    |



**Figure 6.** Histograms constructed from samples with 10,000 elements, generated from the mixture of two Beta distributions with  $w = 0.5$ . Below each histogram are the parameters of the mixture.



**Figure 7.** Efficiency histograms of the SM and VLB methods. Note that the VLB method has a greater average efficiency than SM method.



**Figure 8.** Compression efficiency of the SM and VLB methods for matrices of constant bit-length.

**Table 4.** Compression efficiency of VLB method of samples with bit-lengths coming from a Discrete Uniform distribution  $U(a = 1, b = 64)$ . Average efficiency ( $\bar{\eta}_2 \pm \text{SD}$ ) were calculated over a 1,000 replicates.

| Expected bit-length | Matrix sizes        |                     |                     |
|---------------------|---------------------|---------------------|---------------------|
|                     | Sample size         |                     |                     |
|                     | 100                 | 10,000              | 1,000,000           |
| 1                   | 0.8750 $\pm$ 0.0000 | 0.8750 $\pm$ 0.0000 | 0.8750 $\pm$ 0.0000 |
| 8                   | 0.8202 $\pm$ 0.0031 | 0.8203 $\pm$ 0.0003 | 0.8203 $\pm$ 0.0000 |
| 16                  | 0.7580 $\pm$ 0.0066 | 0.7578 $\pm$ 0.0007 | 0.7578 $\pm$ 0.0001 |
| 32                  | 0.6330 $\pm$ 0.0142 | 0.6329 $\pm$ 0.0014 | 0.6328 $\pm$ 0.0001 |
| 64                  | 0.3826 $\pm$ 0.0288 | 0.3828 $\pm$ 0.0028 | 0.3828 $\pm$ 0.0003 |

**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)

| Expected bit-length ( $np$ ) | Efficiency          |                     |                     |
|------------------------------|---------------------|---------------------|---------------------|
|                              | Sample size         |                     |                     |
|                              | 100                 | 1,000               | 1,000,000           |
| 1                            | 0.8828 $\pm$ 0.0004 | 0.8828 $\pm$ 0.0004 | 0.8828 $\pm$ 0.0000 |
| 8                            | 0.8283 $\pm$ 0.0022 | 0.8281 $\pm$ 0.0002 | 0.8281 $\pm$ 0.0000 |
| 16                           | 0.7656 $\pm$ 0.0032 | 0.7656 $\pm$ 0.0003 | 0.7656 $\pm$ 0.0000 |
| 32                           | 0.6406 $\pm$ 0.0045 | 0.6406 $\pm$ 0.0004 | 0.6406 $\pm$ 0.0000 |
| 64                           | 0.3910 $\pm$ 0.0065 | 0.3906 $\pm$ 0.0006 | 0.3906 $\pm$ 0.0001 |

**Table 6.** Compression efficiency with bit-lengths distributed according to a Poisson distribution ( $\lambda$ ), where  $\lambda$  represents the expected bit-length.

| Expected bit-length ( $\lambda$ ) | Efficiency $\eta_2$ |                     |                     |
|-----------------------------------|---------------------|---------------------|---------------------|
|                                   | Matrix Size         |                     |                     |
|                                   | 100                 | 1,000               | 1,000,000           |
| 1                                 | 0.8751 $\pm$ 0.0015 | 0.8750 $\pm$ 0.0004 | 0.8750 $\pm$ 0.0000 |
| 8                                 | 0.7654 $\pm$ 0.0046 | 0.7656 $\pm$ 0.0005 | 0.7656 $\pm$ 0.0000 |
| 16                                | 0.6405 $\pm$ 0.0064 | 0.6406 $\pm$ 0.0006 | 0.6406 $\pm$ 0.0001 |
| 32                                | 0.3908 $\pm$ 0.0087 | 0.3906 $\pm$ 0.0009 | 0.3906 $\pm$ 0.0001 |