Mathematical Programming (2020) 182:445–464
https://doi.org/10.1007/s10107-019-01401-3

SHORT COMMUNICATION

Series A

A filtered bucket-clustering method for projection onto the simplex and the ℓ_1 ball

Guillaume Perez[1] · Michel Barlaud[2] · Lionel Fillatre[2] · Jean-Charles Régin[2]

Received: 30 December 2017 / Accepted: 27 April 2019 / Published online: 15 May 2019

Abstract
We propose in this paper a new method processing the projection of an arbitrary size vector onto the probabilistic simplex or the ℓ_1 ball. Our method merges two principles. The first one is an original search of the projection using a bucket algorithm. The second one is a filtering, on the fly, of the values that cannot be part of the projection. The combination of these two principles offers a simple and efficient algorithm whose worst-case complexity is linear with respect to the vector size. Furthermore, the proposed algorithm exploits the representation of numeric values in digital computers to define the number of buckets and to accelerate the filtering.

Mathematics Subject Classification 49M30 · 65C60 · 65K05 · 90C25

1 Introduction

Machine learning techniques often involves to use a sparsity constraint which consists in minimizing the number of non-zero components of a given vector. This constraint acts as a regularization when the number of unknown parameters is too high with respect to the number of observations. It is also a mean to select the most relevant features. Looking for sparsity appears in many applications, including sparse representation [6], wavelets soft thresholding [5], Lasso regression [13], survival models

Part of this work has been already published [12].

✉ Guillaume Perez
guillaume.perez06@gmail.com

Michel Barlaud
barlaud@i3s.unice.fr

Lionel Fillatre
lionel.fillatre@i3s.unice.fr

1 Cornell University, Ithaca, NY 14850, USA

2 Université Côte d'Azur, CNRS, 06900 Sophia Antipolis, France

[14], compressed sensing [2], classification [1] and so on. Minimizing the number of non-zero components of a given vector, also called the ℓ_0 norm minimization, is generally a very difficult problem. The ℓ_0 minimization problem is known to be a non-convex NP-hard combinatorial problem [11]. Hence, the common solution is to minimize the ℓ_1 norm of the vector [6,13]. To this purpose, it is crucially important to have a simple algorithm to project a vector onto the ℓ_1 ball or, equivalently, onto the probabilistic simplex [3]. In many applications, for instance in machine learning, a large number of projections is needed and it is especially crucial to use an algorithm whose worst-case complexity is as small as possible.

The primary goal of this paper is to compute efficiently the projection of a vector onto the probabilistic simplex. It is well known that the projection consists to apply a thresholding operation on the vector. The main difficulty is to compute the threshold which depends itself on the vector. In fact, the threshold can be taken equal to one of the values of the input vector but the location of this value is unknown. An usual way to compute the threshold consists in sorting the vector and to look for the appropriate threshold. A common way to accelerate this search is to use a pivot which splits the sorted input vector into two sub-parts. A mathematical condition is then used to determine whether the sub-part with the highest values contains the threshold. From this way, it is then necessary to explore only one of the two sub-parts in order to identify the threshold. The same cutting principle is applied recursively in the selected sub-part until the process converges to the threshold. An other efficient mechanism to accelerate the search is the filtering. A lower bound of the threshold can be maintained during the whole process. The values of the input vector which are below this lower bound are ignored, which is called the filtering. Our paper proposes some novel extensions to the cutting principle and the filtering step.

The contribution of the paper is threefold. First, we propose to split the vectors into a larger number of sub-parts, called the buckets, in order to estimate more precisely and faster the threshold. We establish the convergence of this multiple splitting process. Second, the number of buckets is chosen in function of the number representation in computers. The computers represent the numbers as a string of digits and we exploit the length of this string to accelerate the sorting of the vector. Third, using many buckets allow us to maintain a tight lower bound of the threshold. Each bucket can be quickly filtered by using this lower bound. We show that the proposed algorithm, called the filtered bucketing algorithm $Bucket^F$, is faster than the other state-of-the-art algorithms. The main result of the paper is a projection algorithm whose complexity is $O((d + B) \times \log_B(M))$ where d is the size of the input vector, B is the base of the numeral system used to encode the vector values and M is the bound of the values which are processed. The values B and M are some a priori fixed constant values. A preliminary analysis of the proposed algorithm was published partly in [12].

Section 2 recalls the main technical aspects of the projection onto the ℓ_1 ball and the probabilistic simplex. Section 3 presents our main contribution; using a bucket partitioning to accelerate the calculation of the threshold. Some numerical results are presented in Sect. 4 which shows the efficiency of the filtered bucketing algorithm. The conclusion of this paper is given in Sect. 5.

2 Problem statement

Given a vector $y = (y_1, y_2, \dots, y_d) \in \mathbb{R}^d$ and a real $a > 0$, we aim at computing its projection $P_{\mathcal{B}_a}(y)$ onto the ℓ_1 ball $\mathcal{B}_a$ of radius a:

$$\mathcal{B}_a = \left\{ x \in \mathbb{R}^d : \quad \|x\|_1 \leq a \right\}, \tag{1}$$

where $\|x\|_1 = \sum_{i=1}^d |x_i|$. The projection $P_{\mathcal{B}_a}(y)$ is defined by

$$P_{\mathcal{B}_a}(y) = \arg \min_{x \in \mathcal{B}_a} \|x - y\|_2, \tag{2}$$

where $\|x\|_2$ is the Euclidean norm. as shown in [7] and revisited in [3], the projection onto the ℓ_1 ball can be derived from the projection onto the simplex Δ_a:

$$\Delta_a = \left\{ x \in \mathbb{R}^d : \sum_{i=1}^d x_i = a \text{ and } x_i \geq 0, \quad \forall i = 1, \dots, d \right\}. \tag{3}$$

Let the sign function sign(v) defined as sign$(v) = 1$ if $v > 0$, sign$(v) = -1$ if $v < 0$ and sign$(v) = 0$ otherwise, for any real value $v \in \mathbb{R}$. The projection of y onto the ℓ_1 ball is given by the following formula:

$$P_{\mathcal{B}_a}(y) = \begin{cases} y & \text{if } \ y \in \mathcal{B}_a, \\ (\text{sign}(y_1)z_1, \dots, \text{sign}(y_d)z_d) & \text{otherwise}, \end{cases} \tag{4}$$

where $z = P_{\Delta_a}(|y|)$ with $|y| = (|y_1|, |y_2|, \dots, |y_d|)$ is the projection of $|y|$ onto Δ_a. The fast computation of the projection $x = P_{\Delta_a}(y)$ for any vector y is of utmost importance to deal with sparse vector surrogates. An important property has been established to compute this projection. It was shown [3] that there exists a unique $\tau = \tau_y \in \mathbb{R}$ such that

$$x_i = \max\{y_i - \tau, 0\}, \quad \forall i = 1, \dots, d. \tag{5}$$

The projection is almost equivalent to a thresholding operation. The main difficulty is to compute quickly the threshold τ_y for any vector y. Let $y_{(i)}$ be the ith largest value of y such that $y_{(d)} \leq y_{(d-1)} \leq \cdots \leq y_{(1)}$. The vector y sorted in decreasing order is denoted $y_{(\downarrow)}$. It is interesting to note that (5) involves that $\sum_{i=i}^d \max\{y_i - \tau, 0\} = a$. Let S^* be the support of x, i.e., $S^* = \{i | x_i > 0\}$. Then,

$$a = \sum_{i=1}^d x_i = \sum_{i \in S^*} x_i = \sum_{i \in S^*} (y_i - \tau).$$

It follows that $\tau_y = (\sum_{i \in S^*} y_i - a)/|S^*|$ where $|S^*|$ is the number of elements of S^*. The following property allows us to compute the threshold τ_y. Let

$$\varrho_j(y) = \left(\sum_{i=1}^{j} y_{(i)} - a \right) \Big/ j \tag{6}$$

for any $j = 1, \ldots, d$. Then, it was shown that $\tau_y = \varrho_{K_y}(y)$ where

$$K_y = \max\{k \in \{1, \ldots, d\} \mid \varrho_k(y) < y_{(k)}\}. \tag{7}$$

Looking for K_y, or equivalently $y_{(\tau_y)}$, allows us to find immediately the threshold τ_y. The most famous algorithm to compute the projection, which has been presented in [8], is based on (7). It consists on sorting the values and then finding the first value satisfying (7). A possible implementation is given in Algorithm 1. The worst case and average case complexity of this algorithm are $O(d \log d)$. Such a complexity was the motivation for many works that have use different methods such as extraction of the K_y largest elements using a heap and reducing the complexity to $O(d + k \log d)$ [15].

An important turn on the history of these algorithms has been the use of pivot based search for K_y [9]. Such a method, instead of sorting the elements or extracting the largest one by one, randomly chooses a pivot p and splits the values into two vectors $y_>$ and $y_<$ containing respectively the elements smaller than p and the elements larger than p. The goal of such a split is that it indicates in which sub-part of the vector K_y is. If $\varrho_{|y_>|}(y) \geq p$, then we know that $\tau_y > p$, thus we can stop processing the elements of $y_<$ and continue to split the values of $y_>$. Otherwise if the value is $\leq p$, then $\tau_y < p$ and we now have to work with the values of $y_<$, and only keep in memory the sum of the values of $y_>$.

One particularity of this algorithm is that it starts using some information about the τ_y that we are looking for. After each iteration, we refine the bounds of its possible values. And working with the bounds of τ_y, especially the lower bound, is the main part of one of the best current algorithms for finding τ_y [3,10]. The idea of these algorithm is to no longer take a random pivot, but to use a lower bound of τ_y as a pivot. Such a pivot allows us to only consider the $y_>$ part of the vector after a splitting. Let V be any sub-sequence of y, if we set the pivot to be

$$p = \frac{\sum_{y_i \in V} y_i - a}{|V|}, \tag{8}$$

then we obtain $a = \sum_{y_i \in V}(y_i - p) \leq \sum_{y_i \in V} \max(y_i - p, 0) \leq \sum_{i=1}^{d} \max(y_i - p, 0)$. This implies that $p \leq \tau_y$ and can be used as a lower bound pivot. The particularity of such a lower bound pivot p is that $\forall y_i \in y, y_i < p \implies x_i = 0$, thus any element y_i lower or equal to p will be discarded during by the algorithm. How to efficiently process and maintain p is the main difference between these algorithms, they both have a worst-case complexity of $O(d^2)$, but show an observed practical complexity of $O(d)$ most of the time. A good review of the state of the art can be found in [3].

As stated, most of the state of the art algorithms are based either on sorting the data, or searching using pivot based search. In this paper we propose the following idea, instead of splitting the data into two sub-sequences, we propose to split it into B sub-sequences. This gives us a better granularity and allows us to find a smaller sub-sequence to split again at the next iteration, for the same cost. In addition, such a splitting is naturally compatible with a discarding techniques using lower bounds on τ_y, allowing our algorithm to achieve the best running time in most of our experiments.

Algorithm 1: Sort based algorithm [8]

```
Data: $y, a$
$u \leftarrow sort(y)$
$K \leftarrow \max_{1 \leq k \leq d}\{k : (\sum_{r=1}^{k} u_r - a)/k < u_k\}$
$\tau \leftarrow (\sum_{r=1}^{K} u_r - a)/K$
for $i \in 1 \ldots |y|$ do
    $x_i \leftarrow \max(y_i - \tau, 0)$
```

3 Bucket partitioning

This section describes the bucket-based algorithm which achieves a better worst case complexity than existing algorithms.

3.1 Principle and convergence of the bucket partitioning

The bucket-based method is based on the existence of K_y in (7). The main idea is to split recursively the vector y into a hierarchical family of $B \geq 2$ ordered sub-vectors $\tilde{y}_b^k$ with $b = 1, \ldots, B$ and $k = 1, \ldots, \bar{k}$. The number of recursive splitting, also called the number of levels, is $\bar{k}$. It may depend on y contrary to B which is constant. The sub-vectors are ordered in the sense that all elements of $\tilde{y}_b^k$ are smaller than the ones of $\tilde{y}_{b+1}^k$ for all $b = 1, \ldots, B - 1$. Each sub-vector $\tilde{y}_b^k$ is called a bucketed vector or simply a bucket. The goal is to find the bucket which contains $y_{(K_y)}$. We will show that only one bucket at level k is relevant because only this single bucket is necessary to identify the bucket which actually contains $y_{(K_y)}$ where K_y is defined in (7). Hence, all the buckets $\tilde{y}_b^{k+1}$ at level $k + 1$ are contained in a single bucket $\tilde{y}_{b_k}^k$ where b_k is the identification number of the bucket at level k which needs a deeper analysis. By convention, we assume that $\tilde{y}_{b_0}^0 = y$ and $b_0 = 1$.

Let us define the process to create and analyze the hierarchical family of buckets. For any level $k + 1 \geq 1$, let us consider the interval I^{k+1} defined by

$$I^{k+1} = \left[\min \tilde{y}_{b_k}^k, \max \tilde{y}_{b_k}^k\right] \tag{9}$$

where $\min \tilde{y}_b^k$, resp. $\max \tilde{y}_b^k$, denotes the minimum element, resp. maximum element, of bucket $\tilde{y}_b^k$. Let us consider a partition of I^{k+1} into B ordered sub-intervals $I_1^{k+1}, \ldots, I_B^{k+1}$. Let $h^{k+1} : I^{k+1} \mapsto \{1, \ldots, B\}$ be the bucketing function such that $h^{k+1}(v) = b$ when the real value v belongs to I_b^{k+1}. The bucket $\tilde{y}_{b_k}^k$ is then split into B ordered sub-vectors $\tilde{y}_b^{k+1}$ such that

(i) $\tilde{y}_b^{k+1} = (y_i)_{i \in S_b^{k+1}}$,
(ii) $S_b^{k+1} = \{i \in S_{b_k}^k \ : \ h^{k+1}(y_i) = b\}$,
(iii) $\max \tilde{y}_b^{k+1} < \min \tilde{y}_{b+1}^{k+1}$ for all $b = 1, \ldots, B-1$,

with the convention $S_{b_0}^0 = \{1, \ldots, d\}$. We get $\left|S_B^k\right| \geq 1$ at any level $k \geq 1$ because of the definition of I^{k+1}. The fact that $\max \tilde{y}_b^{k+1} < \min \tilde{y}_{b+1}^{k+1}$ follows from the fact that equal values of y necessarily belongs to the same bucket.

Let $C_{b_0+1}^0 = 0$ and, for any $k > 0$,

$$C_b^{k+1} = \begin{cases} C_{b_k+1}^k + \sum_{b' \geq b} \sum_{i \in S_{b'}^{k+1}} y_i & \text{if } b_k < B, \\ C_{b_{k-1}+1}^{k-1} + \sum_{b' \geq b} \sum_{i \in S_{b'}^{k+1}} y_i & \text{if } b_k = B, \end{cases} \tag{10}$$

be the cumulative sum of buckets $\tilde{y}_b^{k+1}$ to $\tilde{y}_B^{k+1}$, including also the cumulative sum of the buckets kept at previous levels 1, 2, …, k which have been not discarded. Thus, by definition, C_b^{k+1} is the cumulative sum of all $y_{(i)}$ for $i \leq N_b^{k+1}$ where

$$N_b^{k+1} = \begin{cases} N_{b_k+1}^k + \sum_{b' \geq b} \left|S_{b'}^{k+1}\right| & \text{if } b_k < B, \\ N_{b_{k-1}+1}^k + \sum_{b' \geq b} \left|S_{b'}^{k+1}\right| & \text{if } b_k = B, \end{cases} \tag{11}$$

is the number of elements in the family of buckets $\tilde{y}_b^{k+1}$, …, $\tilde{y}_B^{k+1}$, including also the number of elements not discarded at previous levels 1, 2, …, k. We adopt the convention $N_{b_0+1}^0 = 0$. It follows from (10) and (11) that

$$\varrho_{N_b^{k+1}}(y) = (C_b^{k+1} - a)/N_b^{k+1}. \tag{12}$$

By definition of the buckets, it follows that

$$\min \tilde{y}_{b:B}^{k+1} = y_{(N_b^{k+1})} \tag{13}$$

where $\tilde{y}_{b:B}^{k+1}$ is the vector obtained from the concatenation of buckets $\tilde{y}_b^{k+1} \ldots, \tilde{y}_B^{k+1}$. If $\varrho_{N_B^{k+1}}(y) \geq \min \tilde{y}_B^{k+1} = y_{(N_B^{k+1})}$, then $K_y < N_B^{k+1}$ according to (7). Hence, we can discard all the remaining buckets $\tilde{y}_b^{k+1}$ for $b < B$ and continue the bucket-based exploration of $\tilde{y}_B^{k+1}$ to approximate K_y more accurately. Otherwise, we know that $K_y \geq N_B^{k+1}$, thus we continue the analysis of the remaining buckets $\tilde{y}_b^{k+1}$. Let b_{k+1} be the largest b such that

$$\varrho_{N_b^{k+1}}(y) \geq \min \tilde{y}_{b:B}^{k+1}. \tag{14}$$

When $k = 0$, the index b_1 may not exist if $y \in \Delta_a$: the process is stopped. Otherwise, if b_{k+1} exists, then $K_y < N_{b_{k+1}}^{k+1}$. Hence, we can discard all the remaining buckets $\tilde{y}_b^{k+1}$ for $b < b_{k+1}$ and continue the analysis with $\tilde{y}_{b_{k+1}}^{k+1}$, …, $\tilde{y}_B^{k+1}$. However, by definition of b_{k+1}, we know that

$$\varrho_{N_{b_{k+1}+1}^{k+1}}(y) < \min \tilde{y}_{b_{k+1}+1:B}^{k+1}. \tag{15}$$

Hence, K_y necessarily satisfies $N^{k+1}_{b_{k+1}+1} \leq K_y < N^{k+1}_{b_{k+1}}$. To compute K_y, it is then sufficient to explore only $\tilde{y}^{k+1}_{b_{k+1}}$.

From the definition of I^{k+1}, it is easy to verify that the size of the bucket $\tilde{y}^k_{b_k}$ is strictly decreasing as a function of k since the boundaries of I^k are two elements of the previous bucket $\tilde{y}^{k-1}_{b_{k-1}}$ and $B \geq 2$. After a finite number $\bar{k}$ of iterations, $\tilde{y}^{\bar{k}}_{b_{\bar{k}}}$ contains only one value or some repetitions, say $t_0 \geq 1$, of the same value, say v_0. It is straightforward to verify that

$$\frac{C^{\bar{k}}_{b_{\bar{k}}} - a}{N^{\bar{k}}_{\bar{k}}} \geq v_0 \iff \frac{C^{\bar{k}}_{b_{\bar{k}}} - t_0 v_0 - a}{N^{\bar{k}}_{\bar{k}} - t_0} \geq v_0. \tag{16}$$

Hence, this last bucket $\tilde{y}^{\bar{k}}_{b_{\bar{k}}}$ can not contain $y_{(K_y)}$. We stop the exploration. It follows that τ_y satisfies

$$\tau_y = \begin{cases} \varrho_{N^{\bar{k}}_{b_{\bar{k}}+1}}(y) & \text{if } b_{\bar{k}} < B, \\ \varrho_{N^{\bar{k}-1}_{b_{\bar{k}-1}+1}}(y) & \text{if } b_{\bar{k}} = B. \end{cases} \tag{17}$$

The convergence of the algorithm is established. However, it should be noted that it can even be stopped sooner. When K_y is at the minimal extremity of a bucket $\tilde{y}^{\bar{k}}_i$. We obtain the following stop criteria:

$$\max \tilde{y}^{\bar{k}}_i < \varrho_{N^{\bar{k}}_{i+1}}(y) \wedge \min \tilde{y}^{\bar{k}}_{i+1} \geq \varrho_{N^{\bar{k}}_{i+1}}(y). \tag{18}$$

This implies that $\tau_y = \varrho^{\bar{k}}_{N_{i+1}}(y)$. Note that using only this criteria is enough since it generalizes the case of unicity of the value.

The complexity of this algorithm essentially depends on the choice of the bucketing functions h_k at any level k. This aspect is studied in the next subsection. A possible implementation is given in Algorithm 2.

Example A synthetic example is given in Fig. 1. The vector y is split using 6 buckets. Starting from the sub-sequence (bucket) of the largest values $\tilde{y}^1_6$, the algorithm checks if $\varrho_{N^1_6}(y) < \min \tilde{y}^1_6$. It is true, thus all the values in this bucket belong to S^*, the bucket is said *in* and is colored in green in the diagram. Then algorithm continues and checks $\tilde{y}^1_5$, which is also true, and thus *in*. Then for $\tilde{y}^1_4$, the assertion is false, this implies that it exists at least one value which does not belong to S^*, the bucket become the bucket to *split* and is colored in blue in the diagram. Note that all the remaining buckets in this layer of buckets are directly discarded, said *out* and are colored in red in the diagram. The algorithm continue by spliting the bucket $\tilde{y}^1_4$. The same process is made until the last layer $\bar{k}$, where the bucket $\tilde{y}^{\bar{k}}_5$ has the property to stop the algorithm: $\tau_y = \varrho_{N^{\bar{k}}_6}(y)$. The algorithm is finished and the node is colored in grey in the diagram.

In Fig. 2, we show how splitting in several buckets shrinks the number of values that we have to process at each iteration, and allows us to get closer to K_y. The color code is the same as in Fig. 1.

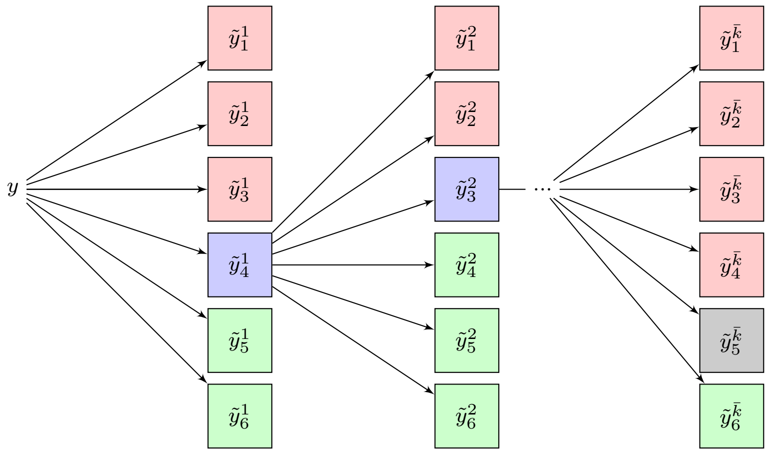


Fig. 1 Principle of "hierarchical bucket filtering". The vector y is split into 6 buckets. The buckets in green must be kept to compute the threshold τ_y. The buckets in red are not involved in the computation of τ_y. The bucket in blue must be explored in order to identify its elements which participate to tau_y. Hence, the splitting process is repeated recursively until all the values of y involved in the computation of tau_y are identified

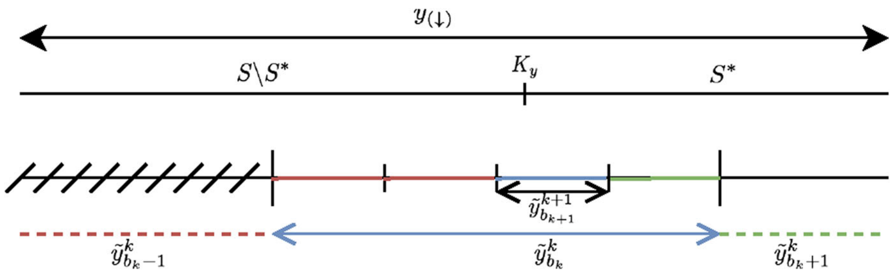


Fig. 2 This figure shows how the algorithm gets closer to K_y after each splitting step. Th goal is to identify the K_y largest elements of the sorted vector $y_{(\downarrow)}$ which are necessary to compute τ_y. The support of these K_y largest elements is S^*. Each splitting step removes the smallest elements not involved in τ_y and identifies the bucket which contains the K_y-th largest element. This bucket is then split again into buckets in order to identify progressively the number K_y. The size of the buckets becomes smaller as the number of splitting increases

3.2 Implementation and complexity

The efficiency of the bucket algorithm is strongly correlated to the h^k function used to do the partitioning, in this section, we propose an efficient implementation and highlight its complexity. Let $\lceil x \rceil$ be the function, defined from the real number to the integer number, returning the lowest integer greater than x. A simple possible implementation of function h^k, using the knowledge that the values of y are in the interval $[\alpha = \min \tilde{y}_{b_k}^k, \beta = \max \tilde{y}_{b_k}^k]$ is:

$$h^{k+1}(x) = \left\lceil \frac{x - \alpha}{\beta - \alpha} * B \right\rceil . \tag{19}$$

This function splits the interval $[\alpha, \beta]$ into B intervals of equal length. Using this function, at worst, at each iteration we split the values into two sets, one containing only one element (for example α) and another containing the remaining ones. Thus d iterations can be required to fully split the data and the worst-case complexity is $O(d^2)$.

That is why we propose another function, based on the numbers' encoding instead of their actual values. Let D be the number of digits used to encode the numbers of y. Assume that number u is greater than number v if the encoding of u is lexicographically greater than the one of v. This assumption is true in nowadays computers using positive double precision values. Consider for instance the numbers $u = 1.6250$ and $v = 0.9375$. If we look at the digits, u is lexicographically greater than v. Today's computers store the number by first storing the *exponent* of the first non zero bit of the number, often using a bias for reaching negative exponent. Then, starting from the next bit on the right of the first non zero bit, the k next digits are stored. These vector of k bits is usually called the *fraction*. For negative numbers, a bit is set to 1, but we consider here only positive numbers. Consider for instance an exponent given using 3 bits and 4 bits for the fraction. We use here a bias of 3. Consider the numbers:

– 011-1010: the exponent is equal to 3 minus the bias, thus 0; The represented number is $2^0 + 2^{-1} + 2^{-3} = 1 + 0.5 + 0.125 = 1.625$
– 010-1110: the exponent is equal to 2 minus the bias, thus -1; the represented number is $2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} = 0.5 + 0.25 + 0.125 + 0.0625 = 0.9375$

Using this binary encoding, the number are still lexicographically ordered. It must be noted that, using encoded numbers over D digits, the number of different numbers is finite. Moreover, let B be the base of the encoding, the maximum number of comparison needed to compare two numbers u and v is $\lceil \log_B(\max(u, v)) \rceil$. In other words, it is only necessary to consider the non-zero digits.

Implementation In order to define an efficient function h^k, we relax the Eq. (9) and thus the property ensuring that at each iteration, the number of elements of the bucket is decreasing. But we ensure that the maximal number of hierarchical iterations is finite. Here it will be D. Let the function E_B^k, defined for numbers encoded over D digits in base B, be the function returning the kth digit in the lexicographical order. The function h^k is now defined as follows:

$$h^k(x) = E_B^k(x). \tag{20}$$

In today's computers, double precision values are often encoded over 64 bits. Using a base $b = 2^8 = 256$, thus defined over 8 binary digits (a Byte), the maximum depth of the algorithm is $log_{2^8}(2^{64}) = 8$. Moreover, the h^k function is strongly computationally less expensive than ones using divide and multiply operations, making the implementation highly competitive. Note that the number of buckets is equal to the base too. Since the maximum number of operations at each iteration is bounded by the number of values of y. The complexity of applying this method in classical double precision implementation in computer is $O(d)$. Note that using a Byte for comparing numbers is often done while implementing efficient sorting method such as the Radix sort [4].

In the general case, using D digits in a base B, the complexity is $O((d+B)\times D)$. Another way to express this complexity is the following. Let M be the maximal value that a number can take. The worst-case complexity is then:

$$O\left((d+B)\times \log_B(M)\right). \tag{21}$$

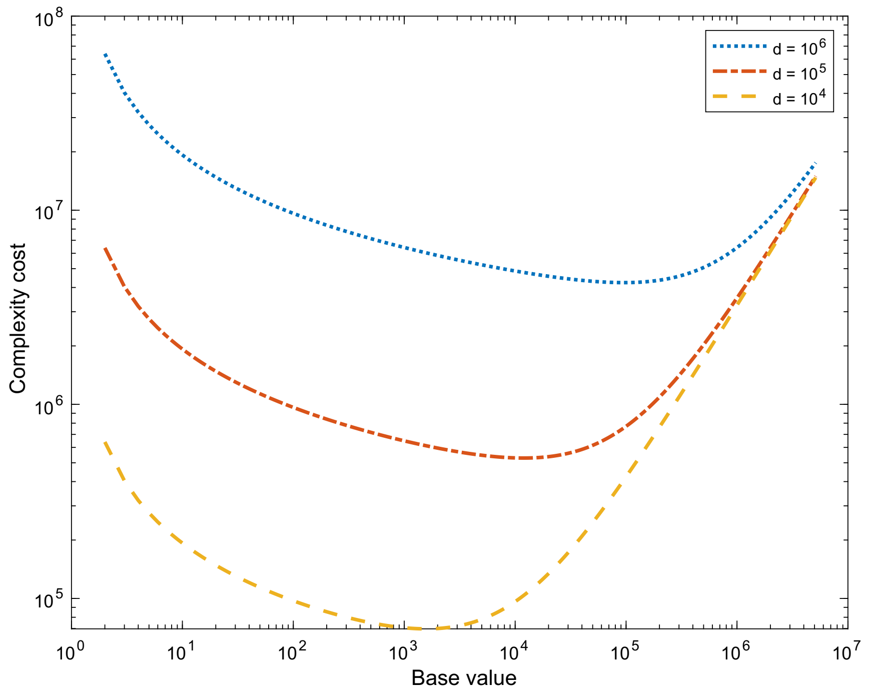
Complexity analysis Such a complexity is interesting in two points, both the base (B) and the number of elements (d) have an important impact on the complexity, while the largest element of the vector (M) only has a logarithmic impact on it. Even if numbers are often encoded using 8-bits unit words, it may be interesting to see if this $2^8 = 256$ base is worth it. Figure 3 shows the impact of these two main factors on the worst-case complexity. The top plot shows that the more we have elements, the larger the best base is. For example, for 10^6 elements, the best possible base seems to be 10^5. Of course, such a complexity does not take into account the fact that at each iteration the number of values change (Table 2 will later show it) and that the range of values is not necessarily $[0, 2^{64}-1]$. Moreover, The second plot shows that using such a bucket-based algorithm starts to payoff when the size of the vector d is larger than the base B. Also, this plot shows that using a base 256 seems to be a good fixed trade-off for the base value.

3.3 Acceleration with filtering

As previously stated, one of the main advantages of the method proposed in [3] is to filter, while iterating over the values, the values that are known to be zero in the projection (i.e. $x_i = 0$). This is done by maintaining a lower bound τ' of τ_y and removing values $y_i < \tau'$ according to (8).

The bucket algorithm is an iterative process of splitting the data. In order to incorporate the filtering by lower bound, we need to decompose its iterations by extracting the first one. This is done by splitting the loop from line 1 in Algorithm 2 into the loops lines 1 and 5 in Algorithm 3. For the first iteration, we have no prior information of what could be the lower bound, thus we apply a process close to the first part of the algorithm proposed in [3] (named $Condat$ in our experiments). We maintain the lower bound τ' of τ_y while iterating over the values, and discard any value smaller than this lower bound and use any larger value to update it. This process is shown in line 1 of the algorithm.

Once the first iteration done, we could directly map the filtering previously proposed in [3]. This implies that, at each new iteration, we use most of the remaining values as an initial guess for the lower bound. Then, we refine this loose bound by removing values that are dominated by it. We did implement this version of the filtering, and without too much surprise, we got the same performance as in [3]. However, the fine grain structure of the bucket algorithm allows us to use exact information collected on the fly during the iterations. Once the first iteration done, and then at each iteration over the buckets, we can extract a lower bound by using the $\varrho_{N^k_{b_k}}(y)$ of the current bucket (line 4). Thus instead of using an initial lower bound that sum all the elements



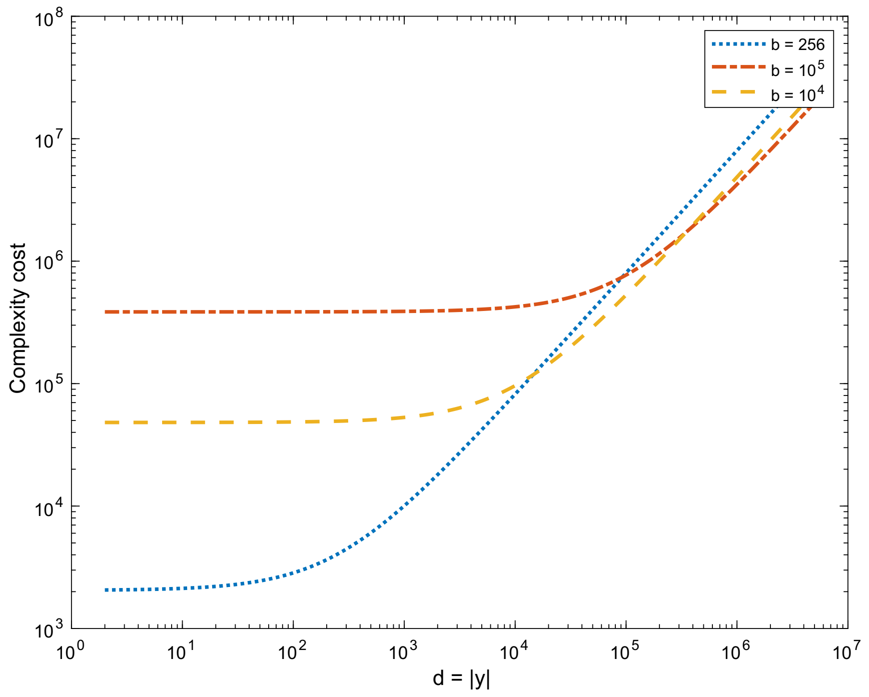


Fig. 3 Bucket-based algorithms Worst-case complexity plot with respect to: (top) the base value, using $M = 2^{64} - 1$ and different d. (Bottom) the number of values, using $M = 2^{64} - 1$ and different bases b

Algorithm 2: *Bucket*

```
  Data: $y, a$
  $S^0_{b_0} \leftarrow \{1, \ldots, d\}$
  $\tilde{y}^0_{b_0} \leftarrow y$
  $C^0_{b_0} \leftarrow -a$
  $N^0_{b_0} \leftarrow 0$
1 for $k \in 1 \ldots \lceil \log_b(D) \rceil$ do
      for $b \in 1 \ldots B$ do
          $S^k_b \leftarrow \{i \in S^{k-1}_{b_{k-1}} \ : \ h^{k-1}(y_i) = b\}$
          $\tilde{y}^k_b \leftarrow (y_i)_{i \in S^k_b}$
2     for $b \in B \ldots 1$ do
          $b_k \leftarrow b$
          if $\varrho_{N^k_{b+1}}(y) > max(\tilde{y}^k_b)$ then
              break loop 1
          if $\varrho_{N^k_b}(y) \geq min(\tilde{y}^k_b)$ then
              break loop 2
  $\tau \leftarrow \varrho_{N^k_{b_k}}(y)$
  for $i \in 1 \ldots |y|$ do
      $x_i \leftarrow \max(y_i - \tau, 0)$
```

as an initial guess, we use only the values that we already know that they belong to τ. Finally, at each pass onto the current bucket to split, we can discard the values that are lower than τ' and update it in an online fashion. A possible implementation is given in Algorithm 3. This novel implementation of the filtering, due to the bucket structure, is one of the key points of the better efficiency of our algorithm.

Remark Another interesting property of algorithms *Bucket* and $Bucket^F$ is that, using prior knowledge, such as the minimum and maximum values or some bounds on these values, we can avoid some of the first iterations of the algorithm. The exponent is coded over 11 bits and has a bit of sign before. Using this prior knowledge can prevent a useless iteration over the bit of sign and the 7 first bits of exponent if we know the interval of definition of these values. Note that only the *left* side of the bit encoding need to be hard-coded as a prior knowledge while the *right* side is automatically detected by our algorithms.

4 Experimental evaluation

In these experiments, we defined random vectors of size varying between 10^5 and 10^7, using either uniform or Gaussian distributions. We generated 500 vectors for each experiment and ran each algorithm independently, and extracted their mean times.

Algorithm 3: $Bucket^F$

```
    Data: $y, a$
    $S^0_{b_0} \leftarrow \{1, \dots, d\}$
    $\tilde{y}^0_{b_0} \leftarrow y$
    $C^0_{b_0} \leftarrow -a$
    $N^0_{b_0} \leftarrow 0$
    $\tau' \leftarrow -a$
    $n \leftarrow 0$
    for $i \in 1 \dots d$ do
1       if $y_i > \tau'$ then
            $n \leftarrow n + 1$
            $\tau' \leftarrow \tau' + (y_i - \tau')/n$
            if $y_n - a > \tau'$ then
                $n \leftarrow 1$
                $\tau' \leftarrow y_i - a$
            Put $y_i$ in bucket $h^1(y_i)$
2   for $k \in 1 \dots \lceil \log_b(D) \rceil$ do
3       for $b \in B \dots 1$ do
            $b_k \leftarrow b$
            if $\varrho_{N^k_{b+1}}(y) > max(\tilde{y}^k_b)$ then
                break loop 2
            if $\varrho_{N^k_b}(y) \geq min(\tilde{y}^k_b)$ then
                break loop 3
4           $n \leftarrow N^k_b; \tau' \leftarrow \varrho_{N^k_{b_k}}(y)$
        for $y_i \in b_k$ do
5           if $y_i > \tau'$ then
                $n \leftarrow n + 1$
                $\tau' \leftarrow \tau' + (y_1 - \tau')/n$
                put $y_i$ in bucket $h^k(y_i)$
    $\tau \leftarrow \varrho_{N^k_{b_k}}(y)$
    for $i \in 1 \dots |y|$ do
        $x_i \leftarrow \max(y_i - \tau, 0)$
```

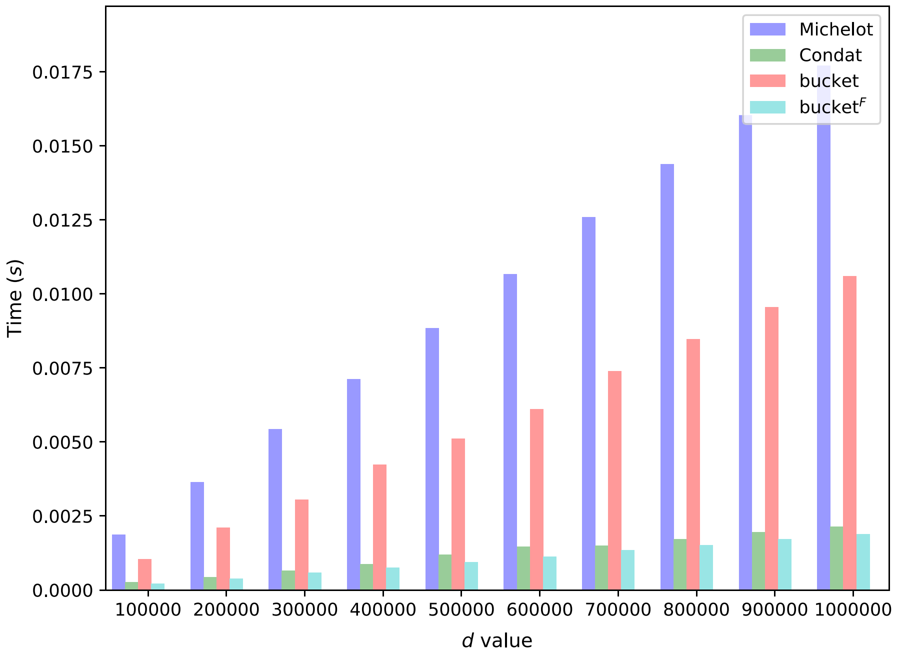


Fig. 4 Uniform distribution: Projection time comparison, while the d value (size of the vector y to project) changes from 10^5 to 10^6, with $a = 1$

We show here the performance of the proposed algorithms, $Bucket$[1] and $Bucket^F$ against the algorithm proposed by $Condat$ [3] which is the current best one, and the one proposed by $Michelot$ in [10], which can be seen as a non dynamic version of the $Condat$ algorithm (i.e. it updates the lower bound only after the iteration instead of during).

Uniform We start our experiments with Fig. 4. This figure shows that when a uniform random distribution is used for vector y, the time needed for projection the vector grows linearly for all three methods, as a function of the vector size. Moreover, the $Bucket^F$ algorithm seems to perform best on this kind of distribution. In this figure, it is hard to differentiate $Bucket^F$ and $Condat$, thus the next experiments are going to focus on this difference.

Gaussian Figure 5 shows that the same kind of results occur while we use a Gaussian distribution to generate the values of vector y. The time seems to grow linearly with d, the size of vector y. Moreover, once again, the $Bucket^F$ algorithm performs better. Figure 6 shows that first, when the radius of the simplex ball is small or unit, the filtering algorithms are the most efficient, but the more the radius grows, the less they are, and the classical bucket become the best one. Such a result may imply that in function of the radius size, one should choose to use the filtering or not in the $Bucket$

[1] The code used for these experiments is available at www.perezguillau.me/code/projection.zip. Note that part of this code is based on the generously provided code from https://www.gipsa-lab.grenoble-inp.fr/~laurent.condat/software.html.

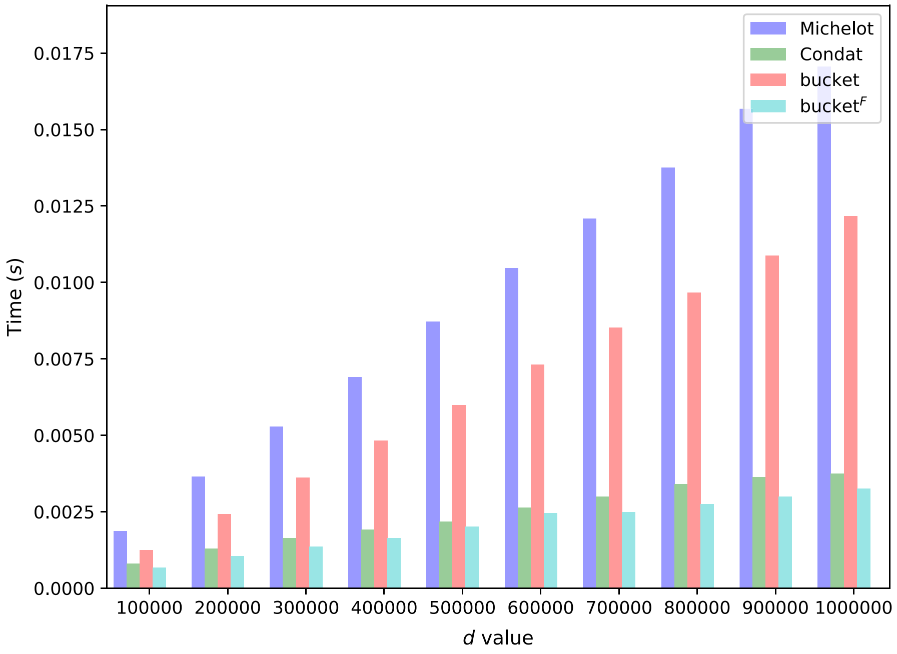


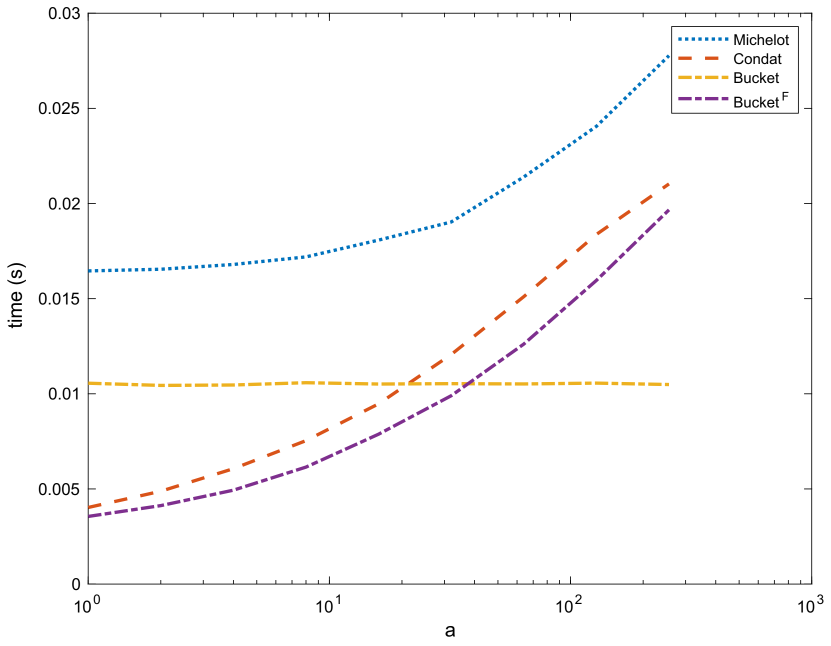
Fig. 5 Gaussian distribution: Projection time comparison, while the d value change from 10^5 to 10^6, with $a = 1$

algorithm. Lastly, when we bias the vector y by adding the value a randomly in one of its component, the size of K decreases, but the same kind of results seems to show up. This last result implies that the increasing of K_y is not the only reason of why the time grows for both filtering algorithms.

Iteration as shown in [3], the number of iterations required by an algorithm is a good information on its efficiency. Table 1 shows the average number of iterations required by each algorithm on different types of instances. As we can see, the $Bucket$ based algorithms are able to find the K_y in a relatively small amount of step, compared to other algorithms. More importantly, if we consider Table 2, we can remark that $Bucket^F$ successfully takes advantage of its bucket design, but also from its efficient fine grain filtering algorithm. Finally, as we can see, the filtering scheme used in $Condat$ improves when σ decreases. In contrary, for the bucket algorithms, when sigma increases, the range of values grows and it is easier to split the values into buckets, according to their encoding, that's why we have a number of iterations decreasing.

Cumulative Finally, Fig. 7 shows a cumulative plot, using all our instances, of the number of instances solved by each method as a function of time. As we can see, the introduction of the filtering for the bucket algorithm drastically changes its performance, allowing it to have the best cumulative plot. Moreover, the average percentage of improvement, in our experiments, of $bucket^F$ against its opponent is 14%.

Remarks When we profile the resolution time, we can remark that, for both filtering algorithms, more than half of the time is spent on the first iteration over y. Moreover,



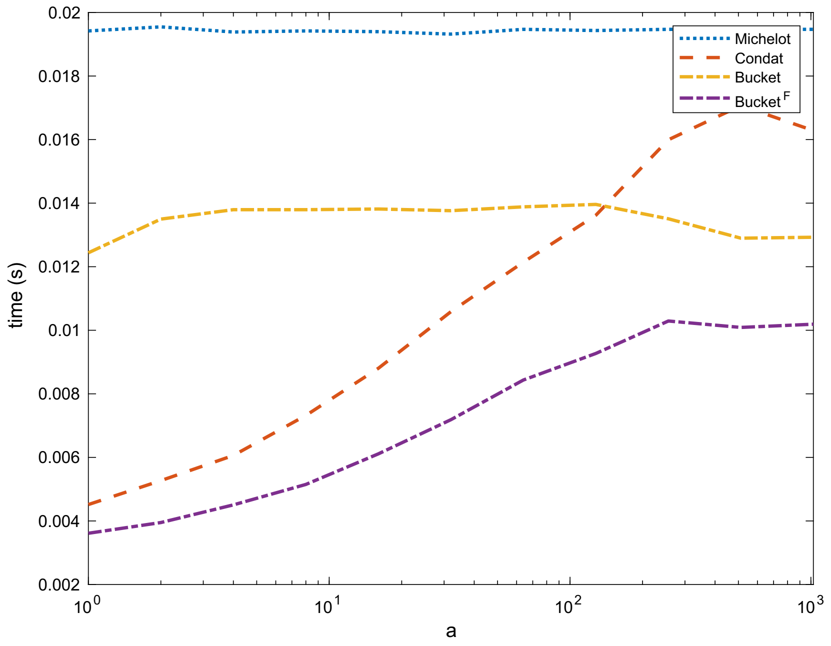


Fig. 6 Gaussian law: (top) projection time comparison, while the a value change from 1 to 1024, with $d = 10^6$ and std-dev $= 10^{-3}$ (when $a = 1$, $K \approx 3500$, when $a = 64$, $K = 14{,}500$). (bottom) Projection time comparison, while the a value change from 1 to 1024, with a vector generated using $d = 10^6$ and std-dev $= 10^{-3}$ and by adding a to one of the components of the vector selected randomly (when $a = 1$, $K \approx 18$, when $a = 64$, $K = 21$)

Table 1 Average number of iterations needed by the algorithm to project a vector of size 10^6

Law	*Michelot*	*Condat*	*Bucket^F*	*Bucket*
Uniform	13.2	6.2	4.0	4.0
Gaussian $\sigma = 0.001$	10.0	6.3	3.4	3.6
Gaussian $\sigma = 0.01$	12.1	6.6	3.1	3.1
Gaussian $\sigma = 0.1$	13.7	7	2.9	2.9

Table 2 Example of number of values at each iteration required by the algorithm to project the vector

Iteration	*Michelot*	*Condat*	*Bucket^F*	*Bucket*
0	10^6	10^6	10^6	10^6
1	500,403	65,656	65,655	999,999
2	212,510	13,315	796	500,628
3	86,122	2950	39	39
4	34,147	669	6	39
5	13,324	143	0	0
6	5107	46	0	0
7	1987	20	0	0
8	750	20	0	0
9	289	0	0	0
10	115	0	0	0
11	55	0	0	0
12	28	0	0	0
13	24	0	0	0
14	24	0	0	0

The vector size is $d = 10^6$ at iteration 0 and the values follow a Gaussian distribution with $\sigma = 0.01$

almost the other half is spent building the vector x (i.e., to project the vector y by applying the threshold), the remaining part of the algorithm is completely flooded in the global run time. The more we increase the number of elements, the more the time spent inside the algorithm is uncorrelated to the global time. Figure 8 shows this behaviour for the two fastest algorithms. Such a result also implies that the current cost of finding K_y is only twice the linear time required to pass through y, thus no more order of magnitude can be gained using this kind of method.

5 Conclusion

This paper proposes a new projection algorithm based on a bucket decomposition, that allows a finer grain splitting of the values, with a linear worst-case complexity. It exploits the representation of numerical values in digital computers to fix the number of buckets and to assign the values to the buckets. Moreover, an improvement based of

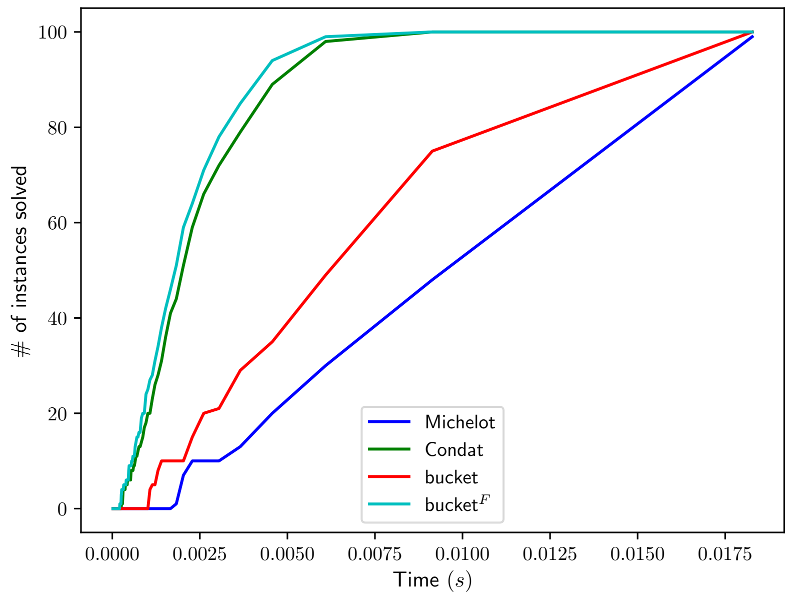


Fig. 7 Cumulative plot of the number of instances solved as a function of time

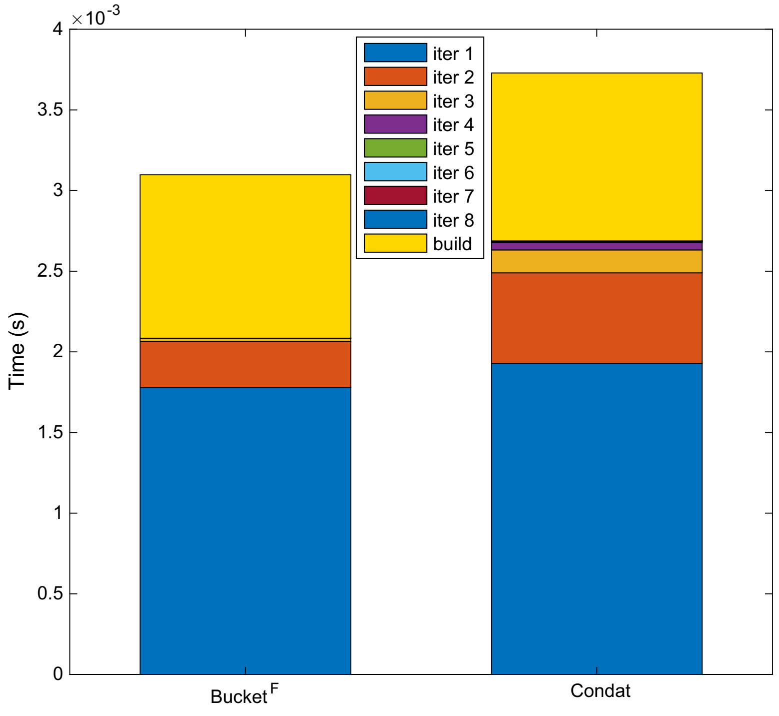


Fig. 8 Time spent in each iteration of the algorithm

the filtering principle is also given, increasing the efficiency of the algorithm. Finally, as shown in the experimental section, our algorithm performs better in practice on the tested instances, and has better guarantees thanks to its complexity.

Appendix A

See Table 3.

Table 3 (Top) Projection time using Gaussian generation with $\mu = a/d$, and $\sigma = 10^{-3}$, and $a = 1$. (Bottom) Projection time using Gaussian generation with $\mu = a/d$, and $\sigma = 10^{-3}$, and $a = 64$

size	Quicksort + simple [8]	Radix + simple	Heap based [15]	Pivot random [9]	Pivot median [9]
1E+05	9.61E-03 (1.0E-03)	1.03E-02 (1.2E-03)	1.30E-03 (9.4E-05)	1.11E-03 (5.9E-04)	2.29E-03 (1.9E-04)
2E+05	2.05E-02 (1.2E-03)	2.09E-02 (1.6E-03)	2.30E-03 (2.2E-04)	2.20E-03 (1.2E-03)	4.41E-03 (3.4E-04)
3E+05	3.05E-02 (1.7E-03)	3.22E-02 (2.2E-03)	3.31E-03 (3.6E-04)	3.22E-03 (1.7E-03)	6.58E-03 (5.3E-04)
4E+05	4.52E-02 (3.7E-03)	4.28E-02 (2.5E-03)	4.40E-03 (4.9E-04)	4.45E-03 (2.3E-03)	8.67E-03 (5.5E-04)
5E+05	5.71E-02 (2.8E-03)	5.72E-02 (3.1E-03)	5.53E-03 (5.2E-04)	5.30E-03 (2.7E-03)	1.09E-02 (7.5E-04)
6E+05	6.77E-02 (3.3E-03)	6.97E-02 (3.3E-03)	6.72E-03 (6.2E-04)	6.77E-03 (3.5E-03)	1.30E-02 (7.7E-04)
7E+05	8.14E-02 (3.8E-03)	8.22E-02 (3.5E-03)	7.79E-03 (6.8E-04)	7.84E-03 (4.2E-03)	1.52E-02 (9.9E-04)
8E+05	9.20E-02 (5.9E-03)	9.30E-02 (4.6E-03)	8.82E-03 (7.9E-04)	8.94E-03 (4.7E-03)	1.73E-02 (1.0E-03)
9E+05	9.61E-02 (3.7E-03)	1.05E-01 (4.8E-03)	9.63E-03 (7.5E-04)	9.87E-03 (4.9E-03)	1.97E-02 (1.2E-03)
1E+06	1.07E-01 (3.9E-03)	1.16E-01 (5.4E-03)	1.09E-02 (9.6E-04)	1.08E-02 (5.6E-03)	2.16E-02 (1.0E-03)
1E+07	1.27E+00 (6.5E-02)	1.34E+00 (2.1E-02)	1.09E-01 (3.1E-03)	1.24E-01 (5.5E-02)	2.04E-01 (4.7E-03)
size	Michelot [10]	Condat [3]	Bucket	Bucket^F	
1E+05	1.77E-03 (1.4E-04)	7.98E-04 (6.6E-05)	1.08E-03 (1.2E-04)	6.45E-04 (4.8E-05)	
2E+05	3.32E-03 (2.6E-04)	1.25E-03 (1.1E-04)	2.18E-03 (2.4E-04)	9.99E-04 (1.1E-04)	
3E+05	4.93E-03 (4.7E-04)	1.60E-03 (1.5E-04)	3.29E-03 (3.4E-04)	1.33E-03 (1.3E-04)	
4E+05	6.56E-03 (6.6E-04)	1.98E-03 (1.5E-04)	4.46E-03 (4.8E-04)	1.65E-03 (1.7E-04)	
5E+05	8.24E-03 (7.4E-04)	2.42E-03 (2.5E-04)	5.60E-03 (5.3E-04)	2.01E-03 (1.8E-04)	
6E+05	9.88E-03 (8.1E-04)	2.75E-03 (2.8E-04)	6.62E-03 (7.0E-04)	2.36E-03 (1.9E-04)	
7E+05	1.16E-02 (9.2E-04)	3.09E-03 (3.1E-04)	7.53E-03 (6.9E-04)	2.64E-03 (2.5E-04)	
8E+05	1.32E-02 (9.3E-04)	3.39E-03 (2.9E-04)	8.55E-03 (7.6E-04)	2.95E-03 (3.0E-04)	
9E+05	1.47E-02 (9.5E-04)	3.72E-03 (3.0E-04)	9.45E-03 (7.4E-04)	3.27E-03 (2.6E-04)	
1E+06	1.65E-02 (1.1E-03)	4.05E-03 (3.4E-04)	1.05E-02 (9.5E-04)	3.56E-03 (2.7E-04)	
1E+07	1.66E-01 (3.9E-03)	2.49E-02 (1.3E-03)	1.43E-01 (6.1E-03)	2.65E-02 (1.6E-03)	
size	Quicksort + simple	Radix + simple	Heap based	Pivot random	Pivot median
1E+05	1.12E-02 (1.3E-03)	1.12E-02 (9.8E-04)	9.26E-03 (8.1E-04)	1.55E-03 (4.3E-04)	2.31E-03 (2.0E-04)
2E+05	2.23E-02 (1.9E-03)	2.26E-02 (1.4E-03)	1.42E-02 (9.5E-04)	3.30E-03 (1.1E-03)	4.48E-03 (3.7E-04)
3E+05	3.16E-02 (2.3E-03)	3.41E-02 (2.0E-03)	1.75E-02 (1.2E-03)	4.84E-03 (1.8E-03)	6.44E-03 (4.3E-04)
4E+05	4.68E-02 (2.2E-03)	4.62E-02 (2.5E-03)	2.04E-02 (1.4E-03)	6.40E-03 (2.2E-03)	8.71E-03 (7.1E-04)
5E+05	5.92E-02 (2.6E-03)	5.73E-02 (3.4E-03)	2.30E-02 (1.5E-03)	7.84E-03 (3.0E-03)	1.09E-02 (7.6E-04)
6E+05	7.21E-02 (2.9E-03)	7.09E-02 (3.3E-03)	2.55E-02 (1.8E-03)	9.58E-03 (3.6E-03)	1.30E-02 (8.5E-04)
7E+05	8.45E-02 (3.2E-03)	8.27E-02 (3.4E-03)	2.77E-02 (1.7E-03)	1.06E-02 (4.1E-03)	1.52E-02 (1.0E-03)
8E+05	9.68E-02 (3.5E-03)	9.55E-02 (3.8E-03)	3.00E-02 (2.0E-03)	1.16E-02 (4.4E-03)	1.73E-02 (1.1E-03)
9E+05	1.09E-01 (3.8E-03)	1.06E-01 (5.0E-03)	3.26E-02 (1.9E-03)	1.25E-02 (4.9E-03)	1.94E-02 (9.9E-04)
1E+06	1.21E-01 (4.1E-03)	1.18E-01 (5.2E-03)	3.51E-02 (1.9E-03)	1.38E-02 (5.6E-03)	2.16E-02 (1.1E-03)
1E+07	1.32E+00 (3.6E-02)	1.28E+00 (4.3E-02)	1.55E-01 (4.7E-03)	1.38E-01 (6.2E-02)	2.04E-01 (4.9E-03)
size	Michelot	Condat	Bucket	Bucket^F	
1E+05	3.40E-03 (3.5E-04)	2.04E-03 (2.2E-04)	1.06E-03 (1.0E-04)	2.35E-03 (2.4E-04)	
2E+05	5.94E-03 (5.9E-04)	4.15E-03 (4.1E-04)	2.07E-03 (2.1E-04)	4.15E-03 (3.6E-04)	
3E+05	7.82E-03 (7.6E-04)	5.96E-03 (5.4E-04)	3.18E-03 (3.0E-04)	5.65E-03 (5.0E-04)	
4E+05	1.02E-02 (7.8E-04)	7.65E-03 (6.6E-04)	4.44E-03 (4.6E-04)	7.04E-03 (6.4E-04)	
5E+05	1.19E-02 (8.4E-04)	9.11E-03 (6.9E-04)	5.67E-03 (5.9E-04)	8.16E-03 (6.9E-04)	
6E+05	1.36E-02 (9.3E-04)	1.06E-02 (9.4E-04)	6.64E-03 (6.4E-04)	9.33E-03 (7.5E-04)	
7E+05	1.55E-02 (1.2E-03)	1.17E-02 (7.9E-04)	7.53E-03 (5.7E-04)	1.03E-02 (7.7E-04)	
8E+05	1.73E-02 (1.2E-03)	1.29E-02 (8.1E-04)	8.61E-03 (7.4E-04)	1.13E-02 (8.2E-04)	
9E+05	1.95E-02 (1.4E-03)	1.40E-02 (8.3E-04)	9.51E-03 (7.2E-04)	1.21E-02 (8.5E-04)	
1E+06	2.15E-02 (1.3E-03)	1.51E-02 (9.3E-04)	1.05E-02 (9.3E-04)	1.29E-02 (8.5E-04)	
1E+07	1.88E-01 (8.3E-03)	7.33E-02 (9.4E-03)	1.43E-01 (6.3E-03)	6.53E-02 (2.4E-03)	

In both tables, the color gradient from red to green shows the running time ordering from worst to best. Note that the difference in color does not take into account the value, only the position in the ordering

References

1. Barlaud, M., Belhajali, W., Combettes, P.L., Fillatre, L.: Classification and regression using an outer approximation projection-gradient method. IEEE Trans. Signal Process. **65**(17), 4635–4644 (2017)
2. Candès, E.J., Tao, T.: Decoding by linear programming. IEEE Trans. Inf. Theory **51**(12), 4203–4215 (2005)
3. Condat, L.: Fast projection onto the simplex and the ℓ_1 ball. Math. Program. Ser. A **158**(1), 575–585 (2016)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms., vol. 6. MIT Press, Cambridge (2001)
5. Donoho, D.L.: De-noising by soft-thresholding. IEEE Trans. Inf. Theory **41**(3), 613–627 (1995)
6. Donoho, D.L., Elad, M.: Optimally sparse representation in general (nonorthogonal) dictionaries via ℓ_1 minimization. Proc. Natl. Acad. Sci. **100**(5), 2197–2202 (2003)
7. Duchi, J., Shalev-Shwartz, S., Singer, Y., Chandra, T.: Efficient projections onto the ℓ_1-ball for learning in high dimensions. In: Proceedings of the 25th International Conference on Machine Learning. ACM (2008), pp. 272–279
8. Held, M., Wolfe, P., Crowder, H.P.: Validation of subgradient optimization. Math. Program. **6**(1), 62–88 (1974)
9. Kiwiel, K.C.: Breakpoint searching algorithms for the continuous quadratic knapsack problem. Math. Program. **112**(2), 473–491 (2008)
10. Michelot, C.: A finite algorithm for finding the projection of a point onto the canonical simplex of $\mathbb{R}^n$. J. Optim. Theory Appl. **50**(1), 195–200 (1986)
11. Natarajan, B.: Sparse approximate solutions to linear systems. SIAM J. Comput. **24**(2), 227–234 (1995)
12. Perez, G., Barlaud, M., Fillatre, l., Régin, J.-C.: A filtered bucket-clustering method for Projection onto the Simplex and the ℓ_1 ball. In: XXVIème Colloque Gretsi, Juan-Les-Pins (2017)
13. Tibshirani, R.: Regression shrinkage and selection via the lasso. J. R. Stat. Soc. Ser. B (Methodol.) 267–288 (1996)
14. Tibshirani, R.: The lasso method for variable selection in the COX model. Stat. Med. (1997)
15. Van Den Berg, E., Friedlander, M.P.: Probing the pareto frontier for basis pursuit solutions. SIAM J. Sci. Comput. **31**(2), 890–912 (2008)