J Supercomput (2019) 75:1298–1309
https://doi.org/10.1007/s11227-018-2423-5

Analysis of an efficient parallel implementation of active-set Newton algorithm

Pablo San Juan Sebastián[1] · **Tuomas Virtanen**[2] · **Victor M. Garcia-Molla**[1] · **Antonio M. Vidal**[1]

Published online: 19 May 2018

Abstract This paper presents an analysis of an efficient parallel implementation of the active-set Newton algorithm (ASNA), which is used to estimate the nonnegative weights of linear combinations of the atoms in a large-scale dictionary to approximate an observation vector by minimizing the Kullback–Leibler divergence between the observation vector and the approximation. The performance of ASNA has been proved in previous works against other state-of-the-art methods. The implementations analysed in this paper have been developed in C, using parallel programming techniques to obtain a better performance in multicore architectures than the original MATLAB implementation. Also a hardware analysis is performed to check the influence of CPU frequency and number of CPU cores in the different implementations proposed. The new implementations allow ASNA algorithm to tackle real-time problems due to the execution time reduction obtained.

Keywords Newton algorithm · Convex optimization · Sparse representation · Multicore · Parallel computing

✉ Pablo San Juan Sebastián
p.sanjuan@upv.es

Tuomas Virtanen
tuomas.virtanen@tut.fi

Victor M. Garcia-Molla
vmgarcia@dsic.upv.es

Antonio M. Vidal
avidal@dsic.upv.es

[1] Department of Information Systems and Computing, Universitat Politècnica de València, Valencia, Spain

[2] Department of Signal Processing, Tampere University of Technology, Tampere, Finland

1 Introduction and motivation

One of the most commonly used models in modern audio processing is the representation of an audio magnitude or power spectrum $x \in \Re_+^{1\times f}$ as a nonnegative linear combination of basis vectors belonging to a precomputed "dictionary". This model is used in different applications, such as source separation [1], automatic music transcription [2], and sound event detection [3].

Usually the n basis vectors in the dictionary are stored as a matrix $B \in \Re_+^{n\times f}$, where each signal of the dictionary is a row of B. The model of the problem can be written as $x \approx v = wB$ subject to $w \geq 0$. The simplest solution would be to find the vector of nonnegative weights $w \in \Re^{1\times n}$ such that $||wB - x||_2$ is minimized. This amounts to solving a nonnegative least squares problem, which is usually solved through active-set methods [4].

However, in audio applications (and in some other fields) better results are often obtained using different measures instead of the 2-norm, such as the Kullback–Leibler (KL) divergence [5].

The KL divergence between vectors x and $\acute{x}$ is defined as

$$KL(x||\acute{x}) = \sum_i d(x_i, \acute{x_i})$$

where function d is

$$d(p,q) = \begin{cases} p\,log(p/q) - p + q & p > 0 \text{ and } q > 0 \\ q & p = 0 \\ \infty & p > 0 \text{ and } q = 0 \end{cases}$$

In the problem of obtaining nonnegative representations of audio for overcomplete dictionaries approached by Virtanen et al. [6], for each input signal $x \in \Re_+^{1\times f}$ we want to find a nonnegative vector $w \in \Re_+^{1\times n}$ that minimizes the KL divergence with respect to the dictionary $B \in \Re_+^{n\times f}$:

$$\min_{w>0} KL(x||wB)$$

However, the KL divergence is a nonlinear function; therefore, the minimization of the KL divergence is a nonlinear optimization problem, with the additional restriction of nonnegativity. In [6], an active-set Newton algorithm (ASNA) was proposed to solve this problem. The algorithm was implemented in MATLAB, and the experiments showed its advantages against some state-of-the-art algorithms like the expectation-maximization update rules [7] and the projected gradient algorithm [8, pp. 267–268].

Due to the great performance of the ASNA algorithm but the lack of a computational efficient implementation of the algorithm, the authors decided to improve the existing MATLAB [9] implementation in order to obtain a lower execution time. This reduction in the execution time is necessary to approach real-time applications. The resulting

implementation is an efficient parallel version suitable for shared memory multicore machines.

The structure of the paper is as follows. In Sect. 2 ASNA algorithm and its existing implementation are explained. In Sect. 3 the developed implementations are presented, and in Sect. 4 the problem used for the experiments on this paper is explained. Then, Sect. 5 shows several experimental analysis performed with all the proposed implementations. Finally, in Sect. 6 the results are discussed.

2 ASNA algorithm

The ASNA algorithm falls into the category of active-set algorithms. These are a family of iterative matricial algorithms where in each iteration only some of the columns or rows are used to compute the iterative approximation of the algorithm. Those columns (or rows) are considered columns (or rows) in the active set, and usually there are steps in the algorithm where columns (or rows) are added or removed from the active set.

The main principle of the ASNA algorithm is that it estimates and updates a set of active atoms (which are the rows of the dictionary matrix) that have nonzero weights. The active set is initialized with a single atom which alone gives the smallest divergence. Then, it finds the most promising atom not in the active set by identifying the atom whose weight derivative is the smallest and adds it to the active set. The weights of the atoms in the active set are estimated using the Newton method where the step size is chosen to ensure nonnegativity of the weights. Atoms whose weights become zero or negative are removed from the active set. The algorithm iterates until a convergence criterion is achieved or a maximum number of iterations given by the user are reached. A detailed view of the algorithm can be found in [6, Sect. III].

The existing implementation programmed in MATLAB, that can be found in [10], uses a more general model than the one shown in the original algorithm [6, pp. 5]. The extended model can work with multiple observations at time, becoming $X \approx V = WB$ subject to $(W, V) \geq 0$ where the rows of $X, V \in \Re_+^{o \times f}$ are the observations and the rows of $W \in \Re_+^{o \times n}$ are the nonnegative weights corresponding to each observation. That model gets some advantages from the fact of computing multiple observations at a time because some matrix-vector operations are replaced by matrix–matrix operations which are more efficient. In this model the approximation matrix V in each iteration is defined as

$$V = W_A B_A \tag{1}$$

where A is the global active set composed of the union of the active sets of each observation X_m (a). In (1) W_A denotes a submatrix formed with the columns of W which are in the global active set A and B_A denotes a submatrix formed with the rows of B which are in the global active set.

A brief pseudocode of that implementation can be found in Algorithm 1. In that implementation the weights in the active set are represented by the nonzero elements in a sparse weight matrix W and the active atoms in the dictionary are represented by B_A.

2.1 Initialization

Under this model, after normalizing each dictionary atom to Euclidean length, the sets of active atoms are initialized with a single index n that alone minimizes the KL divergence for each observation X_m, which is defined as (2) where the weight of each atom $W_{m,n}$ is computed as (3) [11].

$$a = \underset{n}{\operatorname{argmin}} KL(X_m || W_{m,n} B_n) \tag{2}$$

$$W_{m,n} = \frac{X_m 1^T}{B_n 1^T} \tag{3}$$

Here 1 is an all-one vector of length f.

2.2 Adding atoms to the active set

Every K-th iteration with $K > 1$ the algorithm tries to add one new atom to the active set of each observation. The atom with the lowest gradient (the one which will decrease the KL divergence the most) is selected.

The gradients are computed with respect to all weights (the ones corresponding to the atoms in the active set and not in the active set), and then all the atoms not already in the active set are used as candidates to add new atoms to the active set. The gradients of the ones that are already in the active set will be used lately by the algorithm to update the weighs, so computing them together in this step saves computation time.

Taking advantage of the matricial model, the formula of this gradient computation is

$$\frac{d}{dW} KL(W) = \left(1 - \frac{X}{V}\right) B^T \tag{4}$$

here the division of matrices is computed entry-wise and V is computed according to (1). Note that $1B^T$ can be precomputed at the initialization to save computation time during the iterations.

2.3 Updating weights of active atoms

In the updating phase of the algorithm (which corresponds to the inner loop), all operations are performed for each observation X_m as in the original model with one observation vector. In this phase, the algorithm uses the Newton method to update the weights of the atoms on the active set, choosing an appropriate step size to ensure nonnegativity. Let us denote a dictionary matrix whose rows consist of atoms in the active set a of X_m as B_a and a weight row vector which consists of the weights of the active atoms of X_m as w_a. The model (1) can be written as $V_m = w_a B_a$, where V_m is a row of matrix V and corresponds to an approximated observation. The gradient of the KL divergence with respect to w_a is given as (5), and the Hessian matrix with respect to w_a computed at w_a is given by (6).

$$grad = \left(1 - \frac{X_m}{V_m}\right) B_a^T \tag{5}$$

$$H_{w_a} = B_a diag\left(\frac{X_m}{V_m^2}\right) B_a^T \tag{6}$$

Here, "diag" denotes a diagonal matrix whose entries consist on the elements of its argument vector, and V_m^2 denotes entry-wise squaring of vector V_m.

When the gradients have been computed in the atom addition steps of the algorithm, the algorithm uses that gradients instead of computing (5).

Finally the weights are updated as (7) where α is the step size and the search direction can be obtained by solving the system of Eq. (8).

$$w_a \leftarrow w_a - \alpha\, searchDir \tag{7}$$

$$(H_{w_a} + \varepsilon I) \times searchDir = grad \tag{8}$$

An identity matrix I multiplied by a small constant ε is added to ensure numerical stability.

The step size α is obtained by computing the ratio vector $r = w_a / searchDir$ element-wise and choosing the minimum positive element. If $\alpha > 1$ the step size $\alpha = 1$ is used, which corresponds to the standard Newton algorithm. This computation ensures that the weights computed in (7) are nonnegative.

3 Proposed algorithms

The first step was to improve the existing MATLAB implementation before tackling the reimplementation of the algorithm in a different programming language; then we implemented a version of the algorithm in C programming language using the HPC mathematical libraries BLAS and LAPACK. Finally, we implemented a parallel version of the algorithm using threading with OPENMP together with BLAS and LAPACK. A first approach to the proposed implementations were presented in [12]. The source code of all proposed implementations can be found in the repository [13]. All line numbers mentioned in the current section refer to Algorithm 1.

3.1 Improved MATLAB implementation

The improved MATLAB implementation has some modifications that affect positively to the performance of the algorithm.

The first change was transposing the problem. Most of the operations in the original implementation were made row-wise, while MATLAB uses a column-wise memory arrangement. Transposing the problem allows the algorithm to do its operations column-wise taking advantage of MATLAB's memory arrangement. The second modification was changing some conditionals that were checking the existence of a variable containing all gradients to boolean variables, what caused a surprising improve in the performance. Then the sparse product function in line 23 was reworked to use both

Algorithm 1 Original ASNA implementation algorithm

Require: $X \in \Re_+^{o \times f}$ $B \in \Re_+^{n \times f}$.
1: **return** $W \in \Re_+^{o \times n}$
2: Normalize each dictionary atom to unity norm
3: Pre compute $1B^T$ for the gradient computations
4: Initialize active set for each observation
(Active atoms have values in W_A and not active are 0)
5: **for** $i = 1$ **to** $maximum\ number\ of\ iterations$ **do**
6: Find global active atoms A
7: Compute $V = W_A B_A$ (1)
8: $R = X/V$ (element-wise)
9: **if** $i\ mod\ K = 0$ **then**
10: Compute gradient w.r.t all weights (4)
11: **if** $i\ mod\ 10 = 0$ **then**
12: Check convergence for non converged observations
13: Remove converged observations from the computations
14: **if** all observation have converged **then**
15: Scale back W and exit
16: **end if**
17: **end if**
18: Mark as 0 the gradient of the already active weights
19: Add the atom with the minimum gradient of each observation
to the active set, adding a small number to W_A
20: **end if**
21: Compute $R2 = X/V^2$ (element-wise)
22: Find the indexes of the active atoms
23: Compute sparse product $Rcov = RB^T$
24: **for** each observation not converged X_m **do**
25: Find the active atoms of X_m (a)
26: **if** all gradients computed **then**
27: Get $grad$ from the already computed gradients
28: **else**
29: Compute gradients w.r.t active atoms of X_m $(grad)$ (5)
30: **end if**
31: Compute Hessian H_{w_a} (6)
32: Compute the search direction (8)
33: Compute step size
34: Update weights in W_A (7). If a weight becomes negative is removed.
35: **end for**
36: **end for**

matrices in column-wise order, and the system of equations solving in line 32 was solved directly using the Cholesky decomposition instead of using the default MATLAB solver. Finally, some minor tweaks and structural changes were done to improve performance and code readability.

3.2 C implementation

The authors chose the C programming language because it is much more efficient than MATLAB. The C implementation uses the BLAS and LAPACK linear algebra interfaces through the Intel Math Kernel Library (MKL) which is a very efficient implementation for Intel architectures.

The implementation is based on the improved MATLAB implementation and uses all improvements explained in Sect. 3.1. In this implementation the weight matrix is stored in memory as a full matrix, and the atoms in the active set are controlled by a double linked list of "atoms" for each observation. Each "atom" contains a link to the adjacent active atoms and the index of that atom in the full matrix in memory. Using this strategy the algorithm still can compute the sparse products in lines 7 and 23 without the need of finding the active atoms each time (lines 6 and 22), reducing the computation time needed for the sparse products. When removing active atoms in line 34, the atom should be removed from the atom list of observation X_m.

The second main improvement is that the sparse product on line 23, the computation of $R2$ (line 21), the computation of the gradient (line 29) and the computation of the Hessian (line 31) have been combined. All these operations use the same data, so mixing the computations in the proper way instead of computing them one after the other diminishes the number of memory accesses and operations.

Finally, the system of linear equations in line 32 has been solved by means of the LAPACK functions DPOTRF and DPOTRS. The first function computes the Cholesky factorization of a symmetric and positive definite matrix, while the second function uses the factor computed by DPOTRF to solve a triangular system of linear equations. Note that the DPOTRF function is threaded inside the MKL library, which means that in a multicore architecture it will benefit from the multiple cores increasing the algorithm performance. This function is one of the most costly parts of the algorithm, and this is why we do not name sequential the non-parallel implementation.

3.3 Parallel C implementation

The parallel implementation of the ASNA algorithm takes advantage of the data independence between all the observations. Due to this, all observations can be processed in parallel. For the parallel implementation, we used the OpenMP [14] pragma "parallel for" for all loops which iterate along the observations. These loops correspond to lines 4, 7, 18, 19 and 24 . The schedule chosen is dynamic because during the iterative progression of the algorithm the already converged observations are removed from the computations, so the thread that tries to compute an already converged observation will skip it. The dynamic scheduling improves the performance for unbalanced load situations like that.

As said in Sect. 3.2 the DPOTRF functions is already threaded inside the library. But the parallel implementation is used sequentially for each observation because the threading is controlled at observation level. That fact will impact the speedup between both versions.

4 Analysed problem

The sound separation problem analysed in the original ASNA paper [6] was used again to test the proposed implementations with a real application. In this problem, the algorithm should compute the weights matrix W to approximate the mixture matrix X (created by mixing two speech signals) taking into account the dictionary matrix

B which contains dictionaries of both speakers from the original speech signals. The goal is to separate the mixed signal into two individual signals, one for each speaker. For those experiments, 100 signals were generated mixing 2 random speakers for each signal from a pool of 34 speakers . Each signal is represented by a magnitude spectrogram matrix X obtained by using the short-time Fourier transform with o columns (observations) and $f = 751$ rows (features). The number of observations o ranges between 94 and 177, with an average of 129.73. The dictionaries for each speaker were generated by k-means clustering and then combined to form the dictionary B of each test signal. Different dictionary sizes were evaluated: 100, 1000 and 10,000 atoms (50, 500 and 5000 atoms per speaker). In the present paper, a bigger dictionary size of 100,000 atoms per speaker will be evaluated. For more detailed information on the matrix generation process, check [6, Sect. V]. Once the weights are estimated using the ASNA, the models for each speaker in a mixture can be calculated separately, and signals corresponding to each speaker reconstructed as described in [6].

5 Experimental analysis

5.1 Evaluation of the proposed implementations

The experimental environment, from now on called *Server*, consists of a multicore machine with two Intel Xeon E5-2697 V2 (2,7 GHz) processors with 12 physical cores each and 128 GB RAM. By the software side, the machine has MATLAB R2016b and the Intel parallel studio 2017 (contains icc v17.0.1 and MKL v2017) installed. All the tests were executed using the 24 cores available. The development process was carried out in a multicore *Workstation* equipped with an Intel Core i7-3820 (3,6 GHz) processor with four physical cores and 16 GB RAM. By the software side, the machine has MATLAB R2016b and the Intel parallel studio 2017 installed. All the tests were executed using the four cores available. Note that the workstation has a lower number of cores than the server but with a higher CPU frequency.

In all proposed versions, the KL divergence value obtained is the same and equal to the KL divergence obtained by the original MATLAB implementation. Due to this, we are not going to evaluate the KL divergence value in this experiment.

To compare the results of the proposed implementations with the experiments in the original ASNA paper [6], all implementations were tested with three different dictionary sizes (100, 1000 and 10,000) until convergence was achieved. Furthermore, we tested a new bigger dictionary size of 100,000 atoms that we will discuss deeper in Sect. 5.3.

Table 1 shows the execution times in seconds of every proposed implementation for all the dictionary sizes tested. Each cell represents the averaged execution time of the 100 signals tested, and the execution time of each signal has been obtained by averaging 10 measurements to avoid system load effects on the measured times.

It is necessary to test all the signal database because the algorithm convergence criterion affects the execution time of each signal. On the other hand, the matrix X representing each signal has a different number of observations o and this will affect the proposed implementations execution time, especially the parallel C implementation.

Table 1 Execution times of each ASNA implementation for different dictionary sizes on *Server* (s)

	100	1000	10,000	100,000
Original MATLAB implementation	0.962	3.306	20.021	92.253
Improved MATLAB implementation	0.552	1.970	11.554	63.171
C implementation	0.212	0.925	6.588	31.514
Parallel C implementation	0.021	0.144	1.343	16.084

Table 2 Speedup respect to the original MATLAB implementation

	100	1000	10,000	100,000
Original MATLAB implementation	1.000	1.000	1.000	1.000
Improved MATLAB implementation	1.742	1.678	1.733	1.460
C implementation	4.544	3.574	3.039	2.927
Parallel C implementation	44.986	23.004	14.903	5.736

The signal duration in seconds range from 1.46 to 2.71, with an average of 1.99 seconds.

The results show that there is a huge improvement in the execution time of more than one order of magnitude. Comparing the three dictionary sizes from the original ASNA paper, computing the decomposition with the biggest dictionary (10,000 atoms) with the parallel C implementation is almost as fast as the original MATLAB implementation with the smallest (100 atoms) and needs less than half of the time of the medium size dictionary (1000 atoms).

The reduction in execution time obtained by the parallel C implementation makes possible to use the ASNA algorithm with 1000 and 10,000 atoms for real-time applications because the execution time is lower than the signal duration for almost all cases, with the exception of three signals with 10,000 atoms dictionaries. Furthermore, the execution time obtained by the improved MATLAB implementation for the 1000 atoms dictionary is good enough to tackle some real-time applications because it is lower than the signal duration for 60 of the 100 signals.

In order to clarify the improvement obtained, Table 2 shows the speedup obtained from the different implementations respect to the original MATLAB implementation.

5.2 Hardware comparison

To check the influence of the CPU frequency and the number of cores available, we repeated all the experiments in *Workstation* which has a higher CPU frequency but lower number of cores than *Server*.

Table 3 shows the execution times in seconds of the same experiments presented in the previous section but in the *Workstation* machine.

The comparison shows that the MATLAB implementations obtain a lower execution time in *Workstation* (Table 3) than in *Server* (Table 1). The lower execution time in

Table 3 Execution times of each ASNA implementation for different dictionary sizes on *Workstation* (s)

	100	1000	10,000	100,000
Original MATLAB implementation	0.6612	2.5758	14.5771	76.9163
Improved MATLAB implementation	0.3909	1.5792	9.2278	61.8698
C implementation	0.1423	0.7979	7.9010	60.3893
Parallel C implementation	0.0470	0.3354	4.5072	51.2103

the MATLAB version is due to the higher CPU frequency on *Workstation* and the poorer utilization of the multicore architecture of MATLAB in comparison with the C implementations. The C implementation still obtains a lower execution time in *Workstation* for the smaller dictionary sizes, again due to the higher CPU frequency. However, for the bigger dictionary sizes *Server* starts to achieve lower execution times than *Workstation* due to the higher number of cores. The parallel C implementation obtains always lower execution times in *Server* due to the higher number of CPU cores.

5.3 Bigger dictionaries evaluation

Due to the big performance obtained by the parallel C implementations with the original dictionary sizes, more tests with a bigger dictionary of 100,000 atoms were performed. As shown in Table 1, the execution time of the parallel C implementation for the 100,000 atom dictionary is lower than the original MATLAB implementation for the 10,000 atom dictionary.

The motivation of these experiments with bigger dictionaries was to check whether it was worth to use that big dictionaries for the sound separation problem, because the 100,000 atom dictionary is much bigger than the usual dictionaries used in the audio field. Some signal-to-distortion ratio (SDR) experiments were performed to measure the quality of the reconstructed signal with the 100,000 atom dictionaries. We use the signal-to-distortion ratio (SDR) as the metric to measure the separation quality. SDR calculates the ratio of energies of the target signal and the separation error [6, Sect. VI.C] and is a commonly used objective metric in audio source separation evaluations. Figure 1 shows the evolution of the SDR with the progression of the algorithm for different dictionary sizes. As shown in the Figure, the bigger dictionaries need more iterations to achieve convergence. Also, the execution time of each iteration increases with the dictionary size. On the other hand, bigger dictionaries are able to obtain asymptotically the best separation quality measured by the SDR. The results obtained with the new dictionary size (100 000 atoms) achieves the best separation quality among the tested methods. Table 4 shows the SDR achieved on convergence and the time needed to achieve it for the best proposed implementation.

6 Discussion

The experimental results show a big improvement in the performance of the algorithm by using the proposed versions. Especially the parallel C implementation obtains an

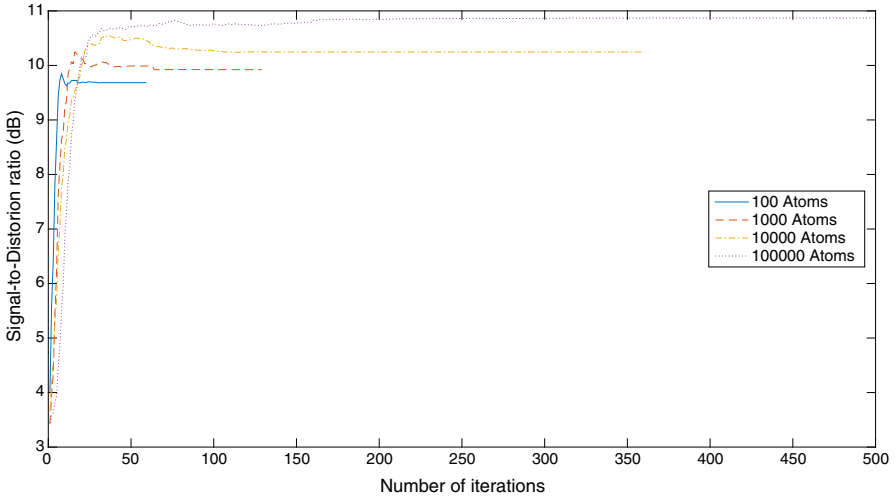

Fig. 1 Signal-to-distortion ratio (dB) per iteration for the different dictionary sizes

Table 4 Signal-to-distortion ratio comparison for the parallel C implementation

	100	1000	10,000	100,000
Signal-to-distortion ratio (dB)	9.684	9.923	10.246	10.869
Execution time (s)	0.021	0.144	1.343	16.084

improvement of more than one order of magnitude in multicore systems. Furthermore, if only one observation needs to be computed, due to the internal parallelism of the MKL library , the algorithm will still benefit from the multicore architecture with the C implementation.

Nonnegative sparse representations have recently been used in many audio processing problems. However, their use in practical applications has been so far limited because of their high computational complexity. In this paper, we show that the computational complexity of state-of-the-art ASNA algorithm, which itself is significantly faster than the established expectation-maximization update rules, can be reduced by more than 10 times. This makes the algorithm appealing for real-time applications such as speech enhancement.

The hardware experiments showed that when using the ASNA algorithm on MATLAB, a faster CPU frequency with a low number of cores will obtain better results than a multicore with more CPU cores but slower CPU frequency. On the other hand, the parallel C implementation will always benefit from a higher number of CPU cores.

To our knowledge, the 100,000-atom dictionaries used in this paper are the largest used for NMF-based sound source separation. The previously used dictionary sizes were typically significantly smaller, the largest used until so far being around 16000 [15] and 10,000 atoms [6]. We showed that increasing the dictionary size up to 100 000 atoms can still increase the source separation quality, and the large dictionary

still benefits significantly from the proposed efficient implementation. Such large dictionary sizes may not be feasible in real-time processing, but the methods will still benefit from the obtained computational savings even in offline processing, where high accuracy is needed requiring large dictionaries.

Due to the trivial parallelism of the multiple observations model, a GPU version of the algorithm can be implemented in future works to speed up the process even more.

Acknowledgements This work has been partially supported by Programa de FPU del MECD, by MINECO and FEDER from Spain, under the projects TEC2015-67387- C4-1-R, and by project PROMETEO FASE II 2014/003 of Generalitat Valenciana. The authors want to thank Dr. Konstantinos Drossos for some very useful mind changing discussions. This work has been conducted in Laboratory of Signal Processing, Tampere University of Technology.

References

1. Raj B, Smaragdis P (2005) Latent variable decomposition of spectrograms for single channel speaker separation. In: Proceedings of the IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA 2005), New Paltz, Ny
2. Bertin N, Badeau R, Vincent E (2010) Enforcing harmonicity and smoothness in Bayesian non-negative matrix factorization applied to polyphonic music transcription. IEEE Trans Audio Speech Lang Process 18(3):538–549
3. Dikmen O, Mesaros A (2013) Sound event detection using non-negative dictionaries learned from annotated overlapping events. In: IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA 2013). New Paltz, NY
4. Lawson CL, Hanson RJ (1995) Solving least squares problems. Society for Industrial and Applied Mathematics, Philadelphia
5. Virtanen T (2007) Monaural sound source separation by nonnegative matrix factorization with temporal continuity and sparseness criteria. IEEE Trans Audio Speech Lang Process 15(3):1066–1074
6. Virtanen T, Gemmeke J, Raj B (2013) Active-set Newton algorithm for overcomplete non-negative representations of audio. IEEE Trans Audio Speech Lang Process 21(11):2277–2289
7. Cemgil AT (2009) Bayesian inference for nonnegative matrix factorisation models. Comput Intell Neurosci 2009:785152
8. Cichocki A, Zdunek R, Phan AH, Amari S (2009) Nonnegative matrix and tensor factorizations. Wiley, New York
9. MATLAB (2014) The Mathworks Inc., MATLAB R2014B, Natnick MA
10. Tuomas Virtanen, Original MATLAB implementation of ASNA algorithm. http://www.cs.tut.fi/~tuomasv/software.html
11. Carabias-Orti J, Rodriguez-Serrano F, Vera-Candeas P, Canadas-Quesada F, Ruiz-Reyes N (2013) Constrained non-negative sparse coding using learnt instrument templates for realtime music transcription. Eng Appl Artif Intell 26:1671–1680
12. San Juan P, Virtanen T, Garcia-Molla Victor M, Vidal Antonio M (2016) Efficient parallel implementation of active-set newton algorithm for non-negative sparse representations. In: 16th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE 2016), Rota, Spain
13. Juan P San, Efficient implementations of ASNA algorithm. https://gitlab.com/P.SanJuan/ASNA
14. OpenMP v4.5 specification (2015). http://www.openmp.org/wpcontent/uploads/openmp-4.5.pdf
15. Gemmeke JF, Hurmalainen A, Virtanen T, Sun Y (2011) Toward a practical implementation of exemplar-based noise robust ASR. In: Signal Processing Conference, 19th European, IEEE, pp 1490–1494