Updating/Downdating the NonNegative Matrix Factorization

P. San Juan[a,*], A.M. Vidal[a], V.M. García-Mollá[a]

[a]*Department of Information Systems and Computing, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, SPAIN.*

Abstract

The Non-Negative Matrix Factorization (NNMF) is a recent numerical tool that, given a non-negative data matrix, tries to obtain its factorization as the approximate product of two nonnegative matrices. Nowadays, this factorization is being used in many science fields; in some of these fields, real-time computation of the NNMF is required. In some scenarios, all data is not initially available and when new data (as new rows or columns) becomes available the NNMF must be recomputed. Recomputing the whole factorization every time is very costly and not suitable for real time applications. In this paper we propose several algorithms to update the NNMF factorization taking advantage of the previously computed factorizations, with similar error and lower computational cost.

Keywords: NNMF, updating, downdating

1. Introduction

The Non-Negative Matrix Factorization (NNMF) is a very popular tool in fields such as document clustering, data mining, machine learning, image analysis, audio source separation or bioinformatics [1, 2, 3, 4, 5]. The goal of the NNMF of a nonnegative data matrix $A \in \mathbb{R}^{m \times n}, (a_{i,j} \geq 0 \; \forall i, j)$ is to obtain two nonnegative matrices $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{k \times n}$ with $k \leq \min(m, n)$, such that $A \approx WH$. The problem can be described as the minimization of the Frobenius norm based target function: $\|WH - A\|_F$ subject to $W, H \geq 0$.

In certain science fields, the NNMF is computed minimizing other target functions, based on alpha-divergence, beta-divergence, Kullback-Liebler divergence, etc. [6, 7, 8, 9, 10]. However, in this paper we will use as target function the Frobenius norm.

*Corresponding author.
Email addresses: `p.sanjuan@upv.es` (P. San Juan), `avidal@dsic.upv.es` (A.M. Vidal), `vmgarcia@dsic.upv.es` (V.M. García-Mollá)

There are situations where the matrix whose NNMF is needed, changes slightly but very often. As an example, this may happen in real time Automatic Music Transcription (a technique that obtains the music score of a piece that is being played) or in real time automatic source separation. The different notes in the incoming sound recordings can be detected through a NNMF factorization, and the data matrix receives new data items (sound recordings in a concrete time) very fast.

If the NNMF is recomputed from scratch after each new data item is received, the computational cost becomes excessive. On the other hand, since new data items are being added, the data matrix grows with time (and so does the cost of any computation associated with the matrix). Therefore, it makes sense that the oldest data items may be discarded, so that the computational cost remains under control.

This example shows the need of studying two problems related to NNMF: the updating of the NNMF (recalculating the NNMF when a new column or row is added to the data matrix) and the downdating of the NNMF (recalculating the NNMF when a column or row of the data matrix is discarded).The paper is focused on dense matrices.

The idea of updating numerical factorizations is not new; indeed, the different updatings of the QR factorization [11] are routinely used in many fields, most often in Signal Processing. There are also other factorizations whose update has been proposed and studied. However, as far as we know, the study of the update of the NNMF has been proposed first by the authors [12]. In this paper we extend and complete the study started in there.

The most popular method of computing the NNMF is the Multiplicative Algorithm of Lee & Seung (MLSA) [13, 14, 15, 16]. This algorithm has many advantages: its simplicity, it is easily parallelizable and there are many implementations readily available (there is an implementation in MATLAB [17]). However, there are other algorithms for this problem, mainly based on the Alternating Least Squares technique (ALS). There are several variations of the ALS technique [18, 19]. One of the most successful seems to be the Hierarchical Alternating Least Squares method (HALS), proposed in [15, 16, 20], and more precisely, the fast HALS (FHALS) implementation proposed in the Algorithm 2 of [20]. This method obtains an important reduction of the factorization error, with a small computational cost. In [21] a modification of HALS method, denoted Greedy Coordinate Descent (GCD) is presented. GCD method is based on a variable selection scheme that uses the gradient of the objective function to arrive at a new coordinate descent method. In this paper we include the FHALS method and the GCD method in the experiments. It will be shown that these methods can be combined to construct some updating algorithms.

The structure of the paper is as follows: first we will describe in detail the mathematical problem, in its different forms, and the basic algorithms for NNMF that are used also for the updating. Then we describe the basic idea to obtain an updated factorization, along with different possibilities of implementation. Last, we evaluate our proposals empirically, and discuss the results. As anticipated, the results show clearly that an updated factorization can be obtained in a small fraction of the time required for a full factorization.

2. Problem description

The first problem to solve is the updating of the NNMF. Given two matrices $W \in \mathbb{R}^{m\times k}$ and $H \in \mathbb{R}^{k\times n}$ which are a solution for the NNMF problem $A \approx WH$, and a new column $b \in \mathbb{R}^{m\times 1}$, we want to compute two new matrices $W_1 \in \mathbb{R}^{m\times k}$ and $H_1 \in \mathbb{R}^{k\times(n+1)}$ which are a solution for the following NNMF problem:

$$V = [A\ b] \approx W_1 H_1 \tag{1}$$

Note that Matlab notation [22] will be used throughout the paper to describe matrices.

This problem can be solved from scratch factorizing matrix V, but our algorithms take advantage of the knowledge of W and H (such that $A \approx WH$) to solve problem (1) with a lower computational cost, and thus in less time.

A generalization of this problem can be obtained adding, instead of a single column, a group of new data columns $B \in \mathbb{R}^{m\times r}$ where r is the number of new columns. We can rewrite the block problem as:

$$V = [A\ B] \approx W_1 H_1 \tag{2}$$

The second problem is the downdating of the NNMF. In this case we need to solve the following problem:

$$V_2 = A(:, r+1:n) \approx W_2 H_2 \tag{3}$$

where $V_2 \in \mathbb{R}^{m\times(n-r)}$, $W_2 \in \mathbb{R}^{m\times k}$ and $H_2 \in \mathbb{R}^{k\times(n-r)}$. Again, this problem can be addressed without the need of compute the NNMF of V_2 from scratch.

We present both problems in terms of adding and removing columns because is the most natural form of update. The problem of adding/deleting rows can be addressed with the same techniques, just by transposing the data matrix.

A natural extension of this problem happens when new columns are added and old columns are removed from the data matrix; we named this a ”window” problem. This problem needs an update and a downdate, but both operations can be processed at the same time to save computational resources.

3. Organizing the updating / downdating algorithms

The main idea behind the different update algorithms is to process the resultant matrices of the initial factorization and then use these matrices as initialization of a low iteration factorization. These algorithms aim to obtain the minimum error with the lowest computational cost. Thus, two stages can be established in every updating algorithm. In a first preprocessing stage some operations are carried out on the W and H input matrices. Then, a few iterations of a base algorithm are performed to compute the new factorization in the postprocessing stage. There are several options for perform both stages. We analyse some possibilities.

As stated in the introduction, there are several algorithms to compute the NNMF. In this section we compare the multiplicative Lee & Seung algorithm (MLSA), the fast HALS algorithm and the GCD algorithm, by analyzing the theoretical costs of all three algorithms.

The MLSA has a cost of $4mnk + 4k^2(m + n)$ flops per iteration, and FHALS algorithm has a cost of $4mnk+4k^2(m+n)+\mathcal{O}(k(m+k))$ flops per iteration.These costs are theoretically calculated from the implementations found in equations (4) and (5) from [13] and Algorithm 2 in [20] respectively. A similar counting of flops can be found in [19]. Despite the fact that FHALS has a greater cost per iteration, it has a faster convergence than MLSA. In comparison, 10 iterations of FHALS algorithm obtain lower error than 100 iterations of MLSA. That is why FHALS is much faster that MLSA in practice.

However, MLSA remains one of the most widely used algorithms in many applications, for example in the field of music processing. See references [8, 9, 10, 23].

A similar cost in terms of superior order, can be found in [21] for GCD Algorithm, although it uses a different strategy. FHALS conducts a cyclic coordinate descent, and it first updates all variables in W in a cyclic order, and then updates variables in H, and so on. Thus, the number of updates performed for each variable is exactly the same. However in GCD, variables are updated with frequency proportional to their importance, choosing to update the coordinate that can reduce more significantly the objective function value. A convenient election of stop criterion can decrease the number of times (denoted as inner iterations) that the updating of variables is carried out [21]. Thus, GCD algorithm has a cost of $4mnk + 4k^2(m + n) + \mathcal{O}(kt)$ flops per iteration, where t represents the average number of inner updates.

All algorithms exposed in the previous paragraph (and several more) are suitable to be used in both stages of the proposed methods. For example, MLSA algorithm was used as base in [12]. As FHALS algorithm has a faster convergence, in this paper we use the FHALS algorithm as base algorithm. GCD method will be used in the preprocessing stage.

The convergence of the updating algorithms presented in this paper is a direct consequence of the convergence of the base algorithms, that is proved in [13] for MLSA, in [15] for FHALS algorithm and in [21] for GCD algorithm.

All algorithms presented in this article have parts that could be implemented efficiently on multicore computers using high performance libraries or parallel multithread programming environments. In particular, the most costly operations in these algorithms are matrix-matrix products, easily parallelizable using, for example, a LAPACK with trheads. In addition, also are presented in this article implementations of block algorithms, in which by using the OpenMP programming environment, the runtime can be decrease with parallel programming techniques.

4. Proposed solutions

In the experiments, the initial factorization will be computed using the FHALS algorithm with 10 iterations over matrix A. This algorithm prototype is $[W, H] = FHALS(M, k, maxIters, W0, H0)$ where M is the matrix to factorize, k is the inner dimension of the factorization, $maxIters$ is the number of iterations to be computed and $W0, H0$ are the initialization matrices.

In the postprocessing stage, 2 iterations of FHALS will be executed because it is the minimum number of iterations to get a better error than the initial 10 iteration factorization.

The base problem against which we compare our solutions, is the factorization of V (or V_2) using the FHALS algorithm with 10 iterations too and the solution of the initial factorization as initialization matrices ($W_0 = W$, $H_0 = H$). The cost of one iteration of FHALS algorithm shown in section 3 will be referred to as $cIterHals$, so the base factorizations will have a cost of $10 * cIterHals$.

4.1. Updating problem

We propose 4 algorithms to solve the updating problem:

1. rand FHALS: The first approach is to add a randomly generated column to matrix H and use that new matrix $H_0 = [H\ x]$ ($x \in \mathbb{R}^k$) as initialization matrix of a 2 iteration FHALS of V. The cost of this method is $2 * cIterHals$ which is clearly lower than the cost of the base factorization. If we discard the cost of generating the random columns, the block version of this algorithm ($H_0 = [H\ X]$ ($X \in \mathbb{R}^{k\times r}$)) keeps the same cost, depending only on the size of the column block r.

Algorithm 1 Update rand FHALS

```
1: X = rand(k,r)
2: $[W_1, H_1] = FHALS(V, k, 2, W, [H\ X])$
```

2. LSQ FHALS: The second algorithm seeks to start the FHALS iterations with a better initial approximation for the new column added to H. So, instead of using a random column, we obtain the new column x as the solution of an unconstrained Linear Least squares problem, where in order to preserve the non-negativity, the negative components of x are set to zero:

$$x = \max(0, \operatorname*{argmin}_{x\in\mathbb{R}^k} \|Wx - b\|_2). \tag{4}$$

Then it uses $H_0 = [H\ x]$ and W as initialization matrices of a 2 iteration FHALS of V. The cost of this algorithm is $2k^2(m-k/3)+k^2+2*cIterHals$ flops which is the highest of the proposed algorithms but still lower than the base cost of $10 * cIterHals$.

Algorithm 2 Update LSQ FHALS

```
1: $[Q, R] = qr(W)$                                  ▷ Economy size QR
2: $c = Q' * b$
3: $x = R(1:k, 1:k) \backslash c(1:k)$   ▷ Solve a triangular system of linear equations
4: $[W_1, H_1] = FHALS(V, k, 2, W, [H\ x])$
```

3. Block LSQ FHALS: The third algorithm solves problem (2). This algorithm is useful when more than one column is available in each update. The cost of this algorithm is $2(k+r)^2(m-k/3)+rk^2+2*cIterHals$ which is more efficient than doing the updates one by one. But the main advantage of this algorithm is that the solving of the system of linear equations with multiple right had sides (line 3) and its previous matrix-matrix product (line 2) can be computed in parallel. Solving this in parallel, the time needed to compute the LSQ part of the algorithm decreases, in a ideal scenario, to the time needed to compute the LSQ for a single column.

Algorithm 3 Update Block LSQ FHALS

```
1: $[Q, R] = qr(W)$
2: $C = Q' * B$
3: $X = R(1:k, 1:k) \backslash C(1:k, :)$          ▷ Solve a triangular system of linear
   equations with multiple right hand sides
4: $[W_1, H_1] = FHALS(V, k, 2, W, [H\ X])$
```

4. GCD FHALS: In this case, GCD algorithm is used in the preprocessing stage, only over one column of H without updating W. Then, 2 iterations of FHALS are performed in post processing stage. The GCD algorithm is very efficient when used on a single column of the matrix H, because it gets the maximum possible reduction in the objective function with few iterations.
In this case, the cost of preprocessing is $2k^2n + 2mnk + \mathcal{O}(kt)$ flops, where t represents the average number of inner updates on matrix H. Hence, the total cost of the algorithm will be $2k^2n + 2mnk + \mathcal{O}(kt) + 2 * cIterHals$. Note that $B \in \mathbb{R}^{m \times r}$, when $r > 1$ the algorithm performs one GCD iteration for each new column added.

Algorithm 4 Update GCD FHALS

```
1: $Hg = upGCD(V, k, B, W, H)$
2: $[W_1, H_1] = FHALS(V, k, 2, W, Hg)$
```

All these algorithms can be easily modified to admit the addition of the column or group of columns not only at the right side of the matrix, but also at any position in the matrix.

4.2. Downdating problem

Following the same conventions of the updating problem, we compute 2 iterations of the FHALS algorithm for V_2 (3) using W and $H(:, r+1:n)$ as initialization matrices. This approach has a cost of $2 * cIterHals$.

This algorithm can be easily modified to admit the removal of the column or group of columns not only at the left side of the matrix, but also at any position in the matrix.

4.3. Window problem

As stated in the introduction, the main use of the downdating problem is to keep the problem size fixed as we increase the initial problem with new columns. Solving an updating and then a downdating is a waste of computational resources, since both can be done at the same time. We named window NNMF to the combination of both updating and downdating in one problem:

$$V_3 = [A(:, r+1:n)\ B] \approx W_3 H_3 \tag{5}$$

where $V_3 \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{m\times r}$,$W_3 \in \mathbb{R}^{m\times k}$ and $H_3 \in \mathbb{R}^{k\times n}$.

Combining the downdating with each updating approach we obtain four algorithms:

1. Window rand FHALS: In this algorithm the initialization matrices for the V_3 FHALS iteration are W and $H_0 = [H(:, r+1:n)\ X]$ where $X \in \mathbb{R}^{k\times r}$ is a set of randomly generated columns. The cost of this algorithm is $2 * cIterHals$, the same cost of a single downdate or update.
2. Window LSQ FHALS: In this algorithm the initialization matrices are W and $H_0 = [H(:, r+1:n)\ x]$ where $x \in \mathbb{R}^k$ is the solution of the LSQ problem. Here we keep the same cost of second update algorithm, avoiding again the cost of the downdate.
3. Window Block LSQ FHALS: Same as in the update case, if more than one column are added we use the Block LSQ algorithm to compute $X = W\backslash B$. So the initialization matrices for this case are W and $H_0 = [H(:, r+1:n)\ X]$ where $X \in \mathbb{R}^{k\times r}$.
4. Window Block GCD FHALS: In this algorithm we use the matrix $Hg(:, r+1:n)$ as initialization matrix, where Hg is the result of our GCD preprocesing algorithm.

To sum up, in Table 1 and Table 2 the theoretical costs of the algorithms presented in this section are shown.

5. Result analysis

In this section we show the results obtained from our experiments and explain them. We compare both the error obtained and the computation time needed to solve them.

Table 1: Theoretical cost summary (flops)

Algorithm	1 column
cIterHals update	$4mnk + 4k^2(m+n) + \mathcal{O}(k(m+n))$
update form scratch	$10 * cIterHalsUp$
update rand FHALS	$2 * cIterHalsUp$
update LSQ FHALS	$2k^2(m-k/3) + k^2 + 2 * cIterHalsUp$
cIterHals downdate	$4m(n-1)k + 4k^2(m+(n-1)) + \mathcal{O}(k(m+(n-1)))$
downdate from scratch	$10 * cIterHalsDown$
downdate FHALS	$2 * cIterHalsDown$

Table 2: Theoretical cost summary (flops)

Algorithm	r columns
cIterHals update	$4m(n+r)k + 4k^2(m+n+r) + \mathcal{O}(k(m+n+r)$
update form scratch	$10 * cIterHalsUp$
update rand FHALS	$2 * cIterHalsUp$
update LSQ FHALS	$2(k+r)^2(m-k/3) + rk^2 + 2 * cIterHalsUp$
cIterHals downdate	$4m(n-r)k + 4k^2(m+n-r) + \mathcal{O}(k(m+n-r)$
downdate from scratch	$10 * cIterHalsDown$
downdate FHALS	$2 * cIterHalsDown$

In the experiments we used the following error measure:

$$Err = \|WH - A\|_F / \sqrt{m * n} \tag{6}$$

Regarding computational time, we executed each algorithm 10 times registering the time needed to solve the problem. Then we averaged the times obtained in order to avoid outliers and obtain a more accurate measurement. Each execution was computed with different initial matrices but the same initial matrices were used in all the algorithms tested.

To evaluate the performance of the algorithms presented, we use two types of matrices. One type of matrices correspond to a real case of Automatic Music Transcription. The other matrices were generated using Algorithm 5, they are random matrices but incorporating some relationship between data to simulate a more realistic behaviour.

Algorithm 5 Matrix generation

```
1: $W = rand(m, k)$
2: $H = rand(k, n + r)$
3: $V = WH + 0.01 * rand(m, n + r)$
4: $A = V(:, 1 : n)$
5: $B = V(:, n + 1 : n + r);$
```

The specifications of the machine where the experiments were executed are:

1. **Hardware:**

   a) **CPU:** Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.70GHz
   b) **CPU physical cores:** 26
   c) **RAM:** 128 GB
2. **Software:**
   a) **S.O:** Ubuntu 14.04
   b) **MATLAB:** MATLAB 2014b

5.1. Updating experiments

The evaluation of algorithms related to NNMF is never easy because the performance of the algorithms depend strongly on the size of the matrices and, most importantly, on the parameter k. For the evaluation of the updating problem with a single column, we have chosen 4 experiments where the matrices have been generated using algorithm 5, and the size of the matrices and the parameter k grows proportionally. The chosen sets of values of m, n, k are as follows: 1) (m=10000, n=4000, k=1000); 2) (m=20000, n=8000, k=2000) ; 3) (m=30000, n=12000, k=3000); 4) (m=40000, n=12000, k=4000).

The execution times of the base FHALS algorithm for a single column update are shown in Table 3.

Table 3: Base execution times with r=1

Base times (s)	10000	20000	30000	40000
FHALS	21.772	295.646	1101.299	2967.827

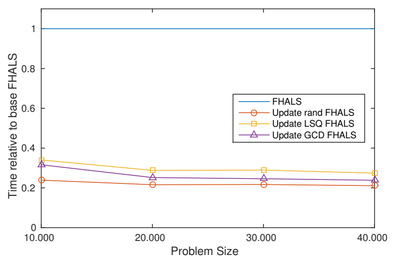
We want to compare visually the execution times of the updating algorithm against those of the base FHALS algorithm; however, since there are figures of very different magnitude, it is not appropriate to display them as raw data. Instead, we will present the results as relative to the base FHALS execution times; that is, the execution times of the 4 experiments displayed in Figure 1 have been divided by execution times of the corresponding base FHALS experiment.

In Figure 1(a) we show the computation time needed to update the data matrix adding one column, with the proposed algorithms and with the NNMF from scratch. The errors obtained in these experiments are shown in Table 4.

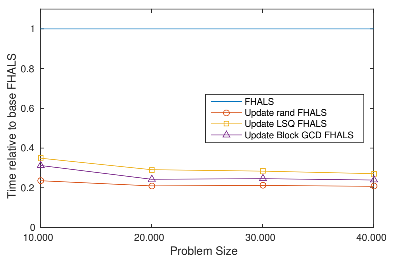
Table 4: Approximation errors of updating algorithms with r=1

Algorithm	**Approximation errors**			
	10000	20000	30000	40000
FHALS	2.488	6.102	10.562	15.722
update rand FHALS	2.459	6.053	10.509	15.651
update LSQ FHALS	2.450	6.032	10.473	15.599
update GCD FHALS	2.448	6.032	10.470	15.593

It can be seen that the proposed algorithms require around 30% of the computational time of the complete NNMF, and rand FHALS (Alg. 1) is the fastest. Its simpler initialization renders this algorithm faster, but also less



(a) Execution time, r=1



(b) Execution time, r=200

Figure 1: Execution times of updating experiments with synthetic matrices relative to FHALS base algorithm

accurate. The error of the proposed algorithms is slightly lower than the error of the complete NNMF and all proposed algorithms have almost the same error.

We repeated the experiment but now adding 200 columns and using the Block LSQ FHALS (Alg. 3). Execution times and errors are shown in Figure 1(b) and Table 6.The execution times of the base FHALS algorithm for a 200 columns block update are shown in Table 5.

Table 5: Base execution times with r=200

Base times (s)	10000	20000	30000	40000
FHALS	22.440	310.253	1127.891	3020.869

Table 6: Approximation errors of updating algorithms with r=200

Algorithm	**Approximation errors**			
	10000	20000	30000	40000
FHALS	2.497	6.111	10.563	15.730
update rand FHALS	3.601	8.863	15.357	22.895
update LSQ FHALS	2.464	6.060	10.480	15.611
update GCD FHALS	2.695	6.429	11.229	16.689

As shown in the figures, in this experiment computation times keep the same ratio as in the experiment with one column, but the error measures indicate that rand FHALS algorithm (Alg. 1) offers greater errors than the full factorization that grow with the problem size and the block size. The GCD FHALS algorithm (Alg. 4) keeps showing a higher performance than the LSQ FHALS algorithm (Alg. 2) but now its error is greater than the base FHALS algorithm error.

An interesting experiment is to test empirically what is the difference between single column updates and block updates. In this experiment we added 2000 columns to the initial matrix (m = 10000, n = 4000 and k=1000). In one case we updated the initial matrix column by column using the LSQ FHALS algorithm (Alg. 2) and in the other case we updated the initial matrix in groups of 200 columns using the Block LSQ FHALS algorithm (Alg. 3).

Table 7: Comparison of single column update and block updates

Algorithm	Time (s)	Error
single column update	12371.793	0.781
block update	85.481	2.019

Results in Table 7 show that block updates are much faster than single column updates. But due to the single column updates involves more FHALS iterations, it achieves a smaller error.

5.2. *Downdating experiments*

The first downdating experiment is to delete a column from the initial matrix and to recompute the NNMF, either by applying a full NNMF or by applying the downdating algorithm. The results are summarized in Table 8. Table 9 shows the results of a similar experiment, where 200 columns are removed. The matrix dimensions maintain the same evolution than in the update experiments.

Table 8: Execution time and approximation error of downdating algorithm with r=1

	Algorithm	10000	20000	30000	40000
Times (s)	FHALS	22.260	277.759	1085.258	2943.336
	downdate FHALS	5.145	61.082	236.412	625.144
Error	FHALS	12.479	6.101	10.551	15.720
	downdate FHALS	2.443	6.040	10.457	15.598

Table 9: Execution time and approximation error of downdating algorithm with r=200

	Algorithm	10000	20000	30000	40000
Times (s)	FHALS	21.374	305.390	1122.552	2945.313
	downdate FHALS	5.119	64.026	237.727	608.619
Error	FHALS	2.493	6.111	10.561	15.724
	downdate FHALS	2.459	6.040	10.465	15.591

As shown by the results, the proposed algorithm obtain a slightly lower error in a much lower amount of time than the factorization from scratch.

5.3. *Window experiments*

To check the expected lower cost of the window algorithm we tested it against an update followed by a downdate. The results are shown in Table 10. Due to the difference in FHALS iterations between both approaches, the window algorithm has a little bit more error than the updating + downdating approach. But its notorious speed advantage compensates it.

Table 10: Execution time and approximation error of window algorithm

	Algorithm	10000	20000	30000	40000
Times (s)	updating + downdating	12.475	151.725	566.963	1451.319
	window algorithm	7.303	87.008	312.521	806.234
Error	updating + downdating	2.433	5.983	10.384	15.489
	window algorithm	2.468	6.049	10.479	15.618

5.4. Real Application: Automatic Music Transcription

Automatic Music Transcription is an active area of research [24]. Usually, the audio file to be studied is transformed into an spectrogram (data matrix)$V \in \mathbb{R}^{m\times n}$ where m is the number of possible frequencies and n is the number of frames or time instants. Some of the methods of determination of pitches in V are based on different forms of the NNMF [23, 8, 9]. The main idea is that a NNMF of X is computed $V = W * H$, $W \in \mathbb{R}^{m\times k}$, $H \in \mathbb{R}^{k\times n}$ where each column of W must contain the frequency information of a concrete pitch, and the i-th column of H contains the information about what pitches (columns of W) are active in the i-th time instant. The parameter k must be selected as larger than the possible number of pitches.

One of the main areas in this field is real time music transcription, where the pitches in the music part must be determined in real time [25, 10]. However, the computational cost of the NNMF has limited its use as a real time tool. This has sometimes been tackled through different simplifications, such as using predetermined sounds for the W matrix, but in general NNMF is considered not suitable for real time [25].

Nevertheless, the updating/downdating techniques described above may change this view. Without updating techniques, a NNMF should be computed in each time instant, or maybe after a few new sounds (columns) have been received and appended to the spectrogram V. Furthermore, the data matriz or spectrogram V and the H matrix would increase their size continuously, therefore the computational cost of each new NNMF would increase as well.

Through updating we can process the new columns with small cost, and the computational cost per time step can be kept fixed by downdating the data matrix, possibly discarding the older columns.

We have designed a simple experiment to illustrate the improvement that can be achieved. We have chosen a relatively long piano part ([26], that lasts 624 seconds) and obtained its spectrogram. (The data matrix can be downloaded from *http://www.inco2.upv.es/software.php*; the procedure to obtain the spectrogram from an audio file is the described in [27]). The procedure used to obtain the spectrogram causes that each second corresponds to 43 new columns. The full spectrogram V has 401 rows and 26836 columns. As a reference, we have computed the full NNMF of the data matrix, selecting k as 88 (number of piano keys, [23]). It must be noted that this computation is quite fast, lasting only 8 seconds (150 FHALS iterations, obtaining an error of 0.67). However, this can be done if the whole matrix is available, which is not true in a real time environment.

We have used this large matrix to simulate real time processing. In such environment, we would need to obtain the NNMF of the part of the music already played in each time instant. We started the computations with the matrix formed with the first 436 columns of V. We compute the NNMF of this submatrix, and then add one column (or several columns) at a time, performing the needed computations.

To use a full NNMF for each new column (using 10 FHALS iterations) takes 10036 seconds, and therefore is obviously not appropriate for real time processing. The error in this case is 1.41.

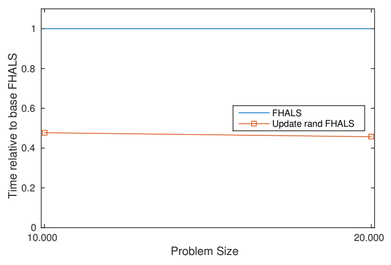
On the other hand, we can use updating/downdating in different ways to try to decrease the computational time. Given that the LSQ-FHALS has a good and stable performance either in single column or in block form, we have chosen this updating procedure for the experiments.

As a first approach we have used the single column window procedure, that is, in each time instant a new column is added (at the right side of the data matrix) and an old column is discarded from the left side. Therefore the size of the matrix is fixed to 436 columns. The results are good from the point of view of time; the whole computation took 432 seconds, carrying out two FHALS iterations per column, since the song lasts 624 seconds, this would mean that real time processing is possible. The evaluation of the error in this case must be done carefully, because the W matrix varies along the process. We have evaluated the error column by column; when a column of H, $H(:,J)$, is discarded (it will not be modified any more), the column vector $aux = V(:,j) - WH(:,j)$ is computed, using the "present" W matrix. Then the sum of the squares of the components of the aux vector is computed. This value will be accumulated to compute the Frobenius norm of the overall error. The overall error computed in this way was 0.54, even better than computing the full NNMF.

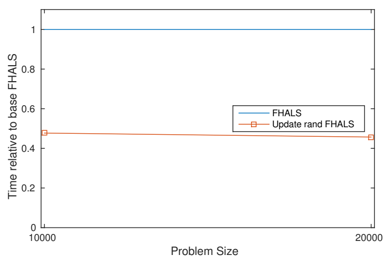
A block window procedure including/removing blocks of 10 columns is clearly faster (42.26 seconds), and still it gives a similar error (0.53). Of course, using a block procedure can create some sort of delay (the first column of the block will not be processed until 9 more columns have arrived). An appropriate block size should be determined for each practical application.

Next, we perform similar experiments to those shown in Section 5.1, using matrices obtained from songs; we repeat the experiments using the base FHALS method and LSQ-FHALS with two experiments: (m=10000, n=401, k=88) and (m=20000, n=401, k=88). The results show that the proposed algorithm perform the same with synthetic and real matrices.

In Figure 2 the execution times of those experiments are shown, while in Table 11 the corresponding error measures are shown. As in 5.1 the execution times are relative to execution times of the base FHALS algorithm, the base FHALS execution times are shown in Table 12.



(a) Relative execution time r=1



(b) Relative execution time r=200

Figure 2: Execution times of updating experiments with real application music matrices relative to FHALS base algorithm

Table 11: Approximation error of updating algorithm with r=1 and r=200 using application matrices

	Algorithm	**Approximation errors** 10000	20000
$r = 1$	FHALS	0.540	1.083
	update LSQ FHALS	0.523	1.051
$r = 200$	FHALS	0.513	1.027
	update LSQ FHALS	0.497	0.997

6. Conclusions and future work

We can safely conclude that the proposed algorithms solve the problems in less time that the NNMF from scratch in all cases; and that LSQ versions (Alg. 2 and 3) lower the error measurements in addition to decrease the execution time.

GCD FHALS algorithm (Alg. 4) performs very well with single column updates being faster and slightly more accurate than LSQ algorithms (Algs. 2 and 3) but its error increases too much for block column updates. That compromises its usefulness for practical applications.

If the reduction of computation time is a priority and the update is done column by column, the rand FHALS update algorithm (Alg. 1) is the fastest and it obtains a good error value. However it should not be used with block column updates because its error increases with the number of columns.

The block LSQ FHALS algorithm (Alg. 3) is a good approach when more than one column is available, and it can benefit of a multicore processor due to its good parallelism properties.

The window algorithms improve the execution time comparing with a full update and downdate.

In the future, this algorithms will be implemented in C and optimized for multicore processors, manycore procesors and GPU. This work will be integrated into the high performance library NNMFPACK [28].

Table 12: Base execution times using application matrices

Base times (s)	10000	20000
FHALS $r = 1$	0.326	0.842
FHALS $r = 200$	0.245	0.678

Acknowledgements

This work has been partially supported by EU together with Spanish Government through TEC2015-67387-C4-1-R (MINECO/FEDER), by Generalitat Valenciana through PROMETEOII/2014/003 and by Programa de FPU del Ministerio de Educación, Cultura y Deporte(Spain).

[1] E. Battenberg, A. Freed, D.Wessel, Advances in the parallelization of music and audio applications, in: Proceedings of the International Computer Music Conference, New York City/Stony Brook, New York, 2010.

[2] J. Wang, W. Zhong, J. Zhang, NNMF-Based Factorization Techniques for High-Accuracy Privacy Protection on Non-negative-valued Datasets, in: Proceedings of the Sixth IEEE International Conference on Computing and Processing, Data Mining Workshops ICDM Workshops, 513–517, 2006.

[3] F. J. Rodriguez-Serrano, J. J. Carabias-Orti, P. Vera-Candeas, T. Virtanen, N. Ruiz-Reyes, Multiple instrument mixtures source separation evaluation using instrument-dependent NMF models, in: International Conference on Latent Variable Analysis and Signal Separation, Springer, 380–387, 2012.

[4] M. Berry, M. Browne, A. Langville, V. Pauca, R. Plemmons, Algorithms and applications for approximate nonnegative matrix factorization, Computational statistics & data analysis 52 (1) (2007) 155–173.

[5] Devarajan, Karthik, Nonnegative matrix factorization: an analytical and interpretive tool in computational biology, PLoS Comput Biol 4 (7) (2008) e1000029.

[6] A. Cichocki, R. Zdunek, A. H. Phan, S.-i. Amari, Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation, chap. 2, John Wiley & Sons, 2009.

[7] P. S. J. Sebastián, A. Vidal, V. García-Mollá, F. Martínez-Zaldívar, J. Ranilla, P. Alonso, M. Alonso-González, R. Cortina., NNMF-Based Factorization Techniques for High-Accuracy Privacy Protection on Non-negative-valued Datasets, in: Proceedings of the 15th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2015, 1023–1034, 2015.

[8] E. Vincent, N. Bertin, R. Badeau, Adaptive harmonic spectral decomposition for multiple pitch estimation, IEEE Trans. on Audio, Speech and Language Processing 18 (3) (2010) 528–537.

[9] A. Khlif, V. Sethu, Multi Range Non-Negative Matrix Factorization Algorithm for Polyphonic Music Transcription, in: The 16th International Society for Music Information Retrieval Conference ISMIR, 2015.

[10] A. Dessein, A. Cont, G. Lemaitre, Real-time polyphonic music transcription with non-negative matrix factorization and beta-divergence, in: The 11th International Society for Music Information Retrieval Conference ISMIR, 489–494, 2010.

[11] G. H. Golub, C. F. Van Loan, Matrix computations, vol. 3, JHU Press, 2012.

[12] P. San Juan Sebastián, A. Vidal, V. García-Mollá, A first approach to column updating of NonNegative Matrix Factorization, in: Proceedings of the 16th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2016, 2016.

[13] D. Lee, H. Seung, Algorithms for non-negative matrix factorization, Advances in Neural Information Processing Systems (2001) 556–562.

[14] A. Mirzal, A convergent algorithm for orthogonal nonnegative matrix factorization, Journal of Computational and Applied Mathematics 260 (2014) 149 – 166, ISSN 0377-0427, doi: http://dx.doi.org/10.1016/j.cam.2013.09.022.

[15] N. Gillis, et al., Nonnegative matrix factorization: Complexity, algorithms and applications, Ph.D. thesis, UCL, 2011.

[16] N. Gillis, F. Glineur, A multilevel approach for nonnegative matrix factorization, Journal of Computational and Applied Mathematics 236 (7) (2012) 1708 – 1723, ISSN 0377-0427, doi: http://dx.doi.org/10.1016/j.cam.2011.10.002.

[17] MATLAB, the Mathworks Inc. MATLAB R2014B, Natnick MA, 2014.

[18] J. Kim, H. Park, Fast nonnegative matrix factorization: An active-set-like method and comparisons, SIAM Journal on Scientific Computing 33 (6) (2011) 3261–3281, ISSN 1064-8275.

[19] N. Gillis, F. Glineur, Accelerated multiplicative updates and hierarchical ALS algorithms for nonnegative matrix factorization, Neural Computation 24 (4) (2012) 1085–1105.

[20] A. Cichocki, P. Anh-Huy, Fast local algorithms for large scale nonnegative matrix and tensor factorizations, IEICE transactions on fundamentals of electronics, communications and computer sciences 92 (3) (2009) 708–721, ISSN 1745-1337.

[21] C.-J. Hsieh, I. S. Dhillon, Fast coordinate descent methods with variable selection for non-negative matrix factorization, in: Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 1064–1072, 2011.

[22] S. Eddins, L. Shure, Matrix indexing in Matlab, Mathworks Newsletter, http://www.mathworks.com/company/newsletters/articles/matrix-indexing-in-matlab.html(Sep. 13, 2014) .

[23] T. Fukuda, Y. Ikemiya, K. Itoyama, K. Yoshii, Score-informed Piano Tutoring System With Mistake Detection And Score Simplification, in: The 12th Sound and Music Computing Conference, Music Technology Research Group, Dept. of Computer Science, Maynooth University, Maynooth, Co. Kildare, Ireland, 2015.

[24] E. Benetos, S. Dixon, D. Giannoulis, H. Kirchhoff, A. Klapuri, Automatic music transcription: Breaking the glass ceiling, in: 13th International Conference on Music Information Retrieval (ISMIR), 2012.

[25] J. Carabias-Orti, F. Rodriguez-Serrano, P. Vera-Candeas, F. Canadas-Quesada, N. Ruiz-Reyes, Constrained non-negative sparse coding using learnt instrument templates for realtime music transcription, Engineering Applications of Artificial Intelligence 26 (7) (2013) 1671–1680, ISSN 0952-1976, doi:http://dx.doi.org/10.1016/j.engappai.2013.03.010.

[26] S. Cherkassky, F. Chopin, Polonaise Op. 44, Deutsche Grammophon, 1973.

[27] F. Canadas-Quesada, P. Vera-Candeas, D. Martinez-Munoz, N. Ruiz-Reyesa, J. Carabias-Orti, P. Cabanas-Molero, Constrained non-negative matrix factorization for score-informed piano music restoration, Digital Signal Processing 50 (2016) 240–257.

[28] NNMFPACK library, http://pirserver.edv.uniovi.es/ Last access 19/06, 2016.