UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

HPC algorithms for nonnegative decompositions

Author: Pablo San Juan Sebastián

Director: Antonio M. Vidal Maciá
Victor M. Garcia Mollá

September 2018

Agradecimientos

Para empezar me gustaría dar las gracias a mis directores Prof. Antonio M. Vidal y Prof. Victor M. Garcia Mollá por brindarme la oportunidad de llevar a cabo mi formación doctoral bajo su tutela. Quiero agradecerles su apoyo durante estos cuatro años ayudándome a dirigir mi investigación y apoyándome con sus conocimientos y consejos.

Me gustaría agradecer también a la Universitat Politècnica de València y en particular al Departamento de Sistemas y Computación (DSIC) y al Instituto de Telecomunicaciones y Aplicaciones Multimedia (iTEAM) por darme el soporte institucional para llevar a cabo mi formación docente e investigadora. En particular al Prof. Pedro Alonso con el que colaboré en la docencia y a los grupos de investigación INCO2 y GTAC por los recursos utilizados en mi trabajo. Debo agradecer también su colaboración en torno a la librería NNMFPack al Prof. Jose Ranilla de la Universidad de Oviedo.

Estoy muy agradecido al Prof. Tuomas Virtanen del Audio Research Gruop (ARG) de la Universidad Tecnologica de Tampere (TUT) de Finlandia por recibirme y supervisar mi estancia en el ARG. Así como con los compañeros del ARG por recibirme con los brazos abiertos y colaborar en un ambiente de trabajo excepcional.

En el ámbito personal debo agradecer el apoyo de mis padres Rosa Sebastián y Fermin San Juan, no solo durante esta etapa formativa doctoral sino durante toda mi vida educativa que ha acabado conduciendo a este momento. Gracias por el apoyo, el ánimo y el cariño ofrecido en todo momento que me ha ayudado a perseguir fielmente mis objetivos y no venirme abajo en los malos momentos.

Quiero agradecer también estos años de buena convivencia y colaboración a mis compañeros de despacho a pesar de que no siempre nos pusieramos de acuerdo en la temperatura. También a mis compañeros de comida con los que siempre he tenido un respiro y buenos momentos a mitad de la jornada durante estos años.

Por ultimo, pero no menos importante, quiero agradecer a todos mis amigos por los buenos momentos que me han dado durante estos años y sobre todo los ánimos en los malos momentos. Siempre es más fácil llevar a cabo un duro trabajo cuando los que te rodean te apoyan y creen en ti.

Abstract

Many real world-problems can be modelled as mathematical problems with nonnegative magnitudes, and, therefore, the solutions of these problems are meaningful only if their values are nonnegative. Examples of these nonnegative magnitudes are the concentration of components in a chemical compound, frequencies in an audio signal, pixel intensities on an image, etc.

Some of these problems can be modelled to an overdetermined system of linear equations, that is, a system of equations with more equations than unknowns. When the solution of this system of equations should be constrained to nonnegative values, a new problem arises. This problem is called the Nonnegative Least Squares (NNLS) problem, and its solution has multiple applications in science and engineering, especially for solving optimization problems with nonnegative restrictions.

Another important nonnegativity constrained decomposition is the Nonnegative Matrix Factorization (NMF). The NMF is a very popular tool in many fields such as document clustering, data mining, machine learning, image analysis, chemical analysis, and audio source separation. This factorization tries to approximate a nonnegative data matrix with the product of two smaller nonnegative matrices. Furthermore, this matrix decomposition usually creates parts based representations of the data in the original matrix.

The algorithms that are designed to compute the solution of these two nonnegative problems have a high computational cost. Due to this high cost, these decompositions can benefit from the extra performance obtained using High Performance Computing (HPC) techniques. Nowadays, there are very powerful computational systems that offer high performance and can be used to solve extremely complex problems in science and engineering. From modern multicore CPUs to the newest computational accelerators (Graph-

ics Processing Units(GPU), Intel Many Integrated Core(MIC), etc.), the performance of these systems keeps increasing continuously. To make the most of the hardware capabilities of these HPC systems, developers should use software technologies such as parallel programming, vectorization, or high performance computing libraries.

While there are several algorithms for computing the NMF and for solving the NNLS problem, not all of them have an efficient parallel implementation available. Furthermore, it is very interesting to group several algorithms with different properties into a single computational library. This thesis presents a high-performance computational library with efficient parallel implementations of the best algorithms to compute the NMF in the current state of the art. In addition, an experimental comparison between the different implementations is presented. This library is focused on the computation of the NMF supporting multiple architectures like multicore CPUs, GPUs and Intel MIC. The goal of the library is to offer a full suit of algorithms to help researchers, engineers or professionals that need to use the NMF. As the NMF is a tool that is transversal to multiple disciplines, not all professionals that could benefit from this decomposition have the knowledge to take advantage of modern, high-performance computing systems. This library tries to solve that need by offering an easy-to-use library that includes top-tier high-performance algorithms that are designed to solve the NMF.

Another problem that is dealt with in this thesis is the updating of nonnegative decompositions. The updating problem has been studied for both the solution of the NNLS problem and the NMF. Sometimes there are nonnegative problems that are close to other nonnegative problems that have already been solved. The updating problem tries to take advantage of the solution of a problem A, that has already been solved in order to obtain a solution of a new problem B, which is closely related to problem A. With this approach, problem B can be solved faster than solving it from scratch and not taking advantage of the already known solution of problem A. In this thesis, an algorithmic scheme is proposed for both the updating of the solution of NNLS problems and the updating of the NMF. Empirical evaluations for both updating problems are also presented. The results show that the proposed algorithms are faster than solving the problems from scratch in all of the tested cases.

Resumen

Muchos problemas procedentes de aplicaciones del mundo real pueden ser modelados como problemas matemáticos con magnitudes no negativas, y por tanto, las soluciones de estos problemas matemáticos solo tienen sentido si son no negativas. Estas magnitudes no negativas pueden ser, por ejemplo, la concentración de los elementos en un compuesto químico, las frecuencias en una señal sonora, las intensidades de los pixeles de una imagen, etc.

Algunos de estos problemas pueden ser modelados utilizando un sistema de ecuaciones lineales sobredeterminado, es decir, un sistema de ecuaciones con más ecuaciones que incógnitas. Cuando la solución de dicho problema debe ser restringida a valores no negativos, aparece un problema llamado problema de mínimos cuadrados no negativos (NNLS por sus siglas en inglés). La solución de dicho problema tiene múltiples aplicaciones en ciencia e ingeniería, concretamente para resolver problemas de optimización con restricciones de no negatividad.

Otra descomposición no negativa importante es la Factorización de Matrices No negativas (NMF por sus siglas en inglés). La NMF es una herramienta muy popular utilizada en varios campos, como por ejemplo: clasificación de documentos, minado de datos, aprendizaje automático, análisis de imagen, análisis químicos o separación de señales sonoras. Esta factorización intenta aproximar una matriz no negativa con el producto de dos matrices no negativas de menor tamaño. Además, esta descomposición matricial suele crear representaciones por partes de los datos en la matriz original.

Los algoritmos diseñados para calcular la solución de estos dos problemas no negativos tienen un elevado coste computacional, y debido a ese elevado coste, estas descomposiciones pueden beneficiarse mucho del uso de técnicas de Computación de Altas Presta-

ciones (HPC por sus siglas en inglés). Hoy en día existen sistemas computacionales muy potentes capaces de ofrecer mucha potencia de computo que son utilizados para resolver problemas extremadamente complejos necesarios en diversos campos de la ciencia y la ingeniería. La potencia de estos sistemas continua aumentando continuamente desde los modernos computadores multinucleo a lo último en aceleradores de calculo (Unidades de Procesamiento Gráfico (GPU), Intel Many Integrated Core (MIC), etc.). Para obtener el máximo rendimiento de estos sistemas de computación de altas prestaciones, los desarrolladores deben utilizar tecnologías software tales como la programación paralela, la vectoración o el uso de librerías de computación altas prestaciones.

A pesar de que existen diversos algoritmos para calcular la NMF y resolver el problema NNLS, no todos ellos disponen de una implementación paralela y eficiente. Además, es muy interesante reunir diversos algoritmos con propiedades diferentes en una sola librería computacional. Esta tesis presenta una librería computacional de altas prestaciones que contiene implementaciones paralelas y eficientes de los mejores algoritmos existentes actualmente para calcular la NMF. Además la tesis también incluye una comparación experimental entre las diferentes implementaciones presentadas. Esta librería centrada en el cálculo de la NMF soporta múltiples arquitecturas tales como CPUs multinucleo, GPUs e Intel MIC. El objetivo de esta librería es ofrecer un abanico de algoritmos eficientes para ayudar a científicos, ingenieros o cualquier tipo de profesionales que necesitan hacer uso de la NMF. Debido a que la NMF es una herramienta transversal que puede ser utilizada en múltiples disciplinas, no todos los profesionales que puedan beneficiarse del uso de la NMF tienen los conocimientos para aprovechar al capacidad de cómputo de los sistemas de altas prestaciones actuales. Esta librería trata de resolver dicho problema ofreciendo un librería fácil de utilizar que incluye implementaciones de altas prestaciones de los mejores algoritmos diseñados para resolver la NMF.

Otro problema abordado en esta tesis es la actualización de las factorizaciones no negativas. El problema de la actualización se ha estudiado tanto para la solución del problema NNLS como para el calculo de la NMF. Existen problemas no negativos cuya solución es próxima a otros problemas que ya han sido resueltos, el problema de la actualización consiste en aprovechar la solución de un problema A que ya ha sido resuelto, para obtener la solución de un problema B cercano al problema A. Utilizando esta aproximación, el problema B puede ser resuelto más rápido que si se tuviera que resolver sin aprovechar la solución conocida del problema A. En esta tesis se presenta una metodología algorítmica para resolver ambos problemas de actualización: la actualización de la solución del problema NNLS y la actualización de la NMF. Además se presentan evaluaciones empíricas de las soluciones presentadas para ambos problemas. Los resultados de estas evaluaciones muestran que los algoritmos propuestos son más rápidos que resolver el problema desde el inicio en todos los casos examinados.

Resum

Molts problemes procedents de aplicacions del mon real poden ser modelats com problemes matemàtics en magnituts no negatives, i per tant, les solucions de estos problemes matemàtics només tenen sentit si son no negatives. Estes magnituts no negatives poden ser, per eixemple, la concentració dels elements en un compost químic, les freqüències en una senyal sonora, les intensitats dels pixels de una image, etc.

Alguns d'estos problemes poden ser modelats utilisant un sistema d'equacions llineals sobredeterminat, es a dir, un sistema d'equacions en mes equacions que incògnites. Quant la solució de este problema deu ser restringida a valors no negatius, apareix un problema nomenat problema de mínims quadrats no negatius (NNLS per les seues sigles en anglés). La solució de este problema te múltiples aplicacions en ciències i ingenieria, concretament per a resoldre problemes de optimisació en restriccions de no negativitat.

Un atra descomposició no negativa important es la Factorisació de Matrius No negatives(NMF per les seues sigles en anglés). La NMF es una ferramenta molt popular utilisada en diversos camps, com per eixemple: classificacio de documents, minat de dades, aprenentage automàtic, anàlisis de image, anàlisis químic o separació de senyals sonores. Esta factorisació intenta aproximar una matriu no negativa en el producte de dos matrius no negatives de menor tamany. Ames, esta descomposició matricial sol crear representacions a parts de les dades en la matriu original.

Els algoritmes dissenyats per a calcular la solució de estos dos problemes no negatius tenen un elevat cost computacional, i degut a este elevat cost, estes descomposicions poden beneficiar-se molt del us de tècniques de Computació de Altes Prestacions (HPC per les seues sigles en anglés). Hui en dia existixen sistemes computacionals molt potents capaços de oferir molta potencia de còmput que son utilisats per a resoldre problemes

extremadament complexos necessaris en diversos camps de la ciència i la ingenieria. La potencia de estos sistemes continua aumentant contínuament, des dels moderns computadors multinucli a lo últim en acceleradors de càlcul (Unitats de Processament Gràfic (GPU), Intel Many Core (MIC), etc.). Per a obtindre el màxim rendiment de estos sistemes de computació de altes prestacions, els desenrolladors deuen utilisar tecnologies software tals com la programació paralela, la vectorisació o el us de llibreries de computació de altes prestacions.

A pesar de que existixen diversos algoritmes per a calcular la NMF i resoldre el problema NNLS, no tots ells disponen de una implementació paralela i eficient. Además, es molt interessant reunir diversos algoritmes en propietats diferents en una sola llibreria computacional. Esta tesis presenta una llibreria computacional de altes prestacions que conté implementacions paraleles i eficients dels millors algoritmes existents per a calcular la NMF. Además, la tesis també inclou una comparació experimental entre les diferents implementacions presentades. Esta llibreria centrada en el càlcul de la NMF soporta diverses arquitectures tals com CPUs multinucli, GPUs i Intel MIC. El objectiu de esta llibreria es oferir una varietat de algoritmes eficients per a ajudar a científics, ingeniers o qualsevol tipo de professionals que necessiten utilisar la NMF. Degut a que la NMF es una ferramenta transversal que pot ser gastada en diverses disciplines, no tots els professionals que poden beneficiar-se del us de la NMF tenen els coneiximents necessaris per a aprofitar la capacitat de còmput dels sistemes de altes prestacions actuals. Esta llibreria tracta de resoldre eixe problema oferint una llibreria fàcil de utilisar que inclou implementacions de altes prestacions dels millors algoritmes dissenyats per a resoldre la NMF.

Un atre problema abordat en esta tesis es la actualisació de les factorisacions no negatives. El problema de la actualisació se ha estudiat tant per a la solució del problema NNLS com per a el càlcul de la NMF. Existixen problemes no negatius la solució dels quals es pròxima a atres problemes no negatius que ya han sigut resolts, el problema de la actualisació consistix en aprofitar la solució de un problema A que ya ha sigut resolt, per a obtindre la solució de un problema B pròxim al problema A. Utilisant esta aproximació, el problema B pot ser resolt molt mes ràpidament que si tinguera que ser resolt des de 0 sense aprofitar la solució coneguda del problema A. En esta tesis es presenta una metodologia algorítmica per a resoldre els dos problemes de actualisació: la actualisació de la solució del problema NNLS i la actualisació de la NMF. Además es presenten evaluacions empíriques de les solucions presentades per als dos problemes. Els resultats de estes evaluacions mostren que els algoritmes proposts son més ràpits que resoldre el problema des de 0 en tots els casos provats.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Background

The solution of the least squares problem is used to approximate the solution of overdetermined systems of equations, in other words, it is used to solve a system of linear equations where there are more equations than unknowns. There are lots of problems that can be modelled to an overdetermined system of linear equations and thus solved by solving the least squares problem [1]. The least squares problem (LSQ) can be defined as obtaining the solution $x \in \mathbb{R}^n$ which minimizes $||Ax - b||_2$ where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. There are several methods for obtaining the solution of the least squares problem (e.g., using the QR decomposition, the normal equations [2, Ch. 5], the SVD decomposition [1, Ch. 4], etc.).

There are many problems that arise from real-world applications that are represented by nonnegative magnitudes (e.g. chemical concentrations in a compound, pixel intensities on an image, different frequencies on an audio wave, etc.). Those problems only make sense with nonnegative solutions, so the methods used to solve them need to take care of that restriction in order to produce a meaningful solution once it is translated to the real world application.

When the LSQ problem to be solved has the nonnegativity restriction mentioned above, it is commonly called the nonnegative least squares (NNLS) problem. The NNLS problem

detailed in Section 2.1 is particularly interesting for those fields in which optimization techniques are used with nonnegativity restrictions. This problem adds the restriction $x \geq 0$ to the solution of the least squares minimization problem. A common approach for solving the NNLS problem is to solve the corresponding least squares problem and then project the solution to the nonnegative space by moving the negative entries of the solution vector to 0. This approach usually gives a solution that is not even close to the optimal solution, and a much better solution is obtained by using a properly constrained method. Numerous algorithms have been proposed to solve the NNLS problem [3, 4, 5, 6, 7, 8, 9, 10, 11, 1, 12, 13]. Despite the accuracy of the methods to solve the NNLS problem, they have a higher complexity. Also, from a computational point of view, they require more execution time to achieve a solution than the unconstrained version.

An extension of the NNLS problem is the NNLS problem with multiple right hand sides. When there are multiple NNLS problems that share the same coefficient matrix A, it is more efficient to tackle the problem as a whole instead of solving multiple NNLS problems. The NNLS problem with multiple right hand sides consists of obtaining the solution $X \in \mathbb{R}_+^{n\times k}$ which minimizes $||AX - B||_F$ where $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{m\times k}$.

The NNLS problem is also a key component of the algorithms that solve the Nonnegative Matrix Factorization (NMF), which is the core of this thesis and is detailed in Section 2.2. The NNLS problem needs to be solved in some of the methods that are used to calculate the NMF, especially in those methods that are based on the technique of alternately solving two convex optimization problems with nonnegativity constraints, which are, in fact, two NNLS with multiple right hand sides problems (see Section 2.4.4).

The NMF was first presented in [14] as Positive Matrix Factorization, but its use and popularity increased after the publication of [15]. It is a matrix factorization that approximates a nonnegative matrix $A \in \mathbb{R}_+^{m\times n}$ with the product of two matrices $W \in \mathbb{R}_+^{m\times k}$ and $H \in \mathbb{R}_+^{k\times n}$ where k its usually smaller than m and n.

One of the advantages of this factorization is that when $k << m,n$, a big reduction in the size of the represented data is achieved, and that is why the NMF is used in the field of data compression [16]. However, taking into account that the factorization is an approximation to a non unique solution, there will be some data loss in the process.

Another key point of the NMF is that it creates a parts based representation of the data on matrix A, which makes the factorization suitable to model any problem in which a parts based decomposition is needed(e.g., signal source separation, speech separation, etc.). This is easy to understand when looking at the example presented in [15], which is shown in Figure 1.1. In that example, a matrix A created from a face image database where each column represents a face is factorized with $k = 49$. In the resulting factorization, each column of W represents a basis image that contains parts of faces and each row of H

contains the coefficients of the linear superposition needed to create an approximate face image. The approximate image is created by superposing the basis images according to the linear coefficients. Each cell of the $7x7$ grid on the left side of Figure 1.1 represents one column of W, while the $7x7$ grid in the middle represents a row of H. The resulting face on the right represents a column of the approximated matrix WH, and the original face on top represents a column of matrix A. This small example intuitively shows how an approximate face is formed by adding parts of faces from the basis images learned by the NMF.

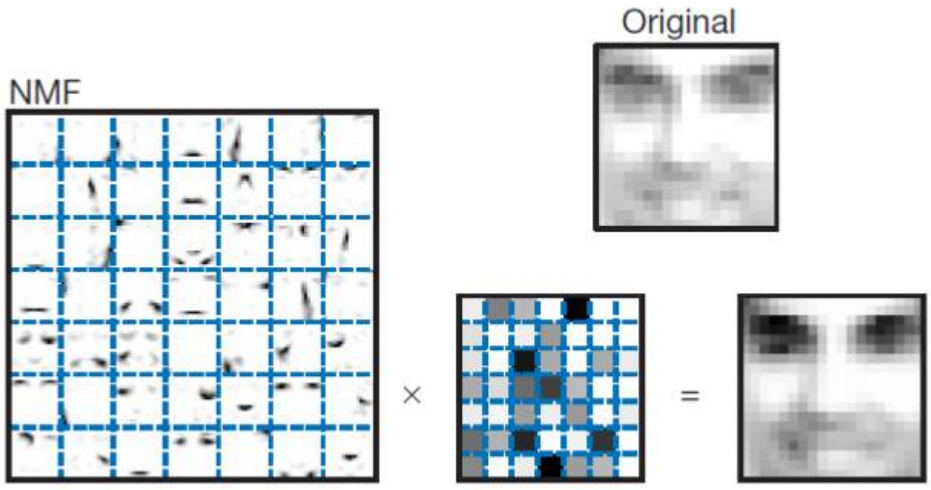


Figure 1.1: Parts based decomposition of a face image database obtained by means of NMF

These two properties of data compression and automatic learning of parts based representations have made the NMF a widely used tool in many science and engineering areas such as document clustering, data mining, machine learning, image analysis, audio source separation, and bioinformatics [17, 18, 19, 20, 21, 22].

There are several algorithms that are designed to compute the NMF, and the most relevant ones are presented in Section 2.4. Those algorithms, along with the ones for solving the NNLS problem presented on Section 2.3, are iterative methods with a high computational cost. The computational power needed to solve these problems can be achieved by means of High Performance Computing (HPC) techniques.

The evolution of computing systems and the increasing performance of their hardware, together with the increasing computational needs of science and engineering to solve complex problems, have caused the emergence of a discipline called High Performance Computing (HPC). HPC includes both the hardware and software technologies to aggregate high amounts of computing power and to use it efficiently to solve large problems with the highest performance possible. On the hardware side, multiple architectures that are detailed in Section 3.2 can be used depending on the problem to be solved and the resources that are available. Nowadays, multicore computers with their increasing number of cores, together with the General Purpose Graphics Processing Units (GPGPUs) deliver high performance with an affordable price. These systems can produce a large amount of computing power when used properly. On the software side, HPC offers developers parallel computing technologies and libraries to make the most of the hardware available and to be able to unveil the true performance of their computer systems.

By taking advantage of HPC technologies, efficient algorithms for solving nonnegative factorizations can be designed in order to decrease the execution time needed to solve the factorizations. If the execution time is reduced, the limiting high computational cost can be overcome, and the advantages of the factorizations can be exploited in practical applications.

1.2 Motivation

Nowadays, there are several implementations of different algorithms for computing the NMF in different programming languages and with different levels of efficiency. However, to our knowledge, there is no HPC package that contains a suite of different algorithms to compute the NMF. Usually, the authors of each algorithm share their source code and, at best, create a software library that includes their algorithms. However, if some scientists using the NMF as a tool in their work need to use different NMF algorithms for different problems, they must install different libraries in order to access the different algorithms. This represents a loss of time that could be better spent on improving their research. Moreover, in the case that only source codes are presented, the use of those algorithms may be not trivial for scientists that are not specialized in computer science. Also, some of those algorithms do not offer efficient or parallel versions.

This situation has motivated the creation of an HPC library containing different algorithms that have been chosen from the best ones to compute the NMF. The library should contain efficient parallel versions of each algorithm that can take advantage of the computational capabilities of modern HPC systems like multicores or GPUs. Furthermore, the library should be easy to use and include interfaces to the most commonly used lan-

guages in scientific computation, so that experts of different areas that are not related to computer science can use the library in their own computer programs.

The problem of updating the solution of NNLS problems and updating the NMF has not yet been studied. The idea behind the updating problem is the following: when the coefficient matrix of a NNLS problem is a slightly modified version of the coefficient matrix of another NNLS problem that has been already solved, the solution of the solved problem can be used to solve the modified problem with a lower computational cost. This has motivated our group to study the updating problem and to develop a methodology to solve it.

An equivalent updating problem arises from the NMF. The updating of the NMF consists of using the matrices obtained from a previous factorization to solve a new factorization where the matrix to be factorized is a modified version of the previous factorized matrix (for example, by adding or removing rows or columns). The use of this information can reduce the execution time needed to find the new factorization which is very convenient for real-time applications. Consider, for example, a problem of signal separation that is solved by means of the NMF where the whole signal is not available from the beginning and new columns are added to the matrix to be factorized when the next part of the signal is received. This problem can be solved by updating the factorization each time that new columns are added instead of recomputing the whole NMF from scratch. In this example, updating the NMF reduces the execution time needed to process each new part of the signal and makes the NMF suitable for real-time problems. The updating of the NMF allows the NMF factorization to be used for many real-time problems where the NMF was too slow to be used due to its complexity.

Finally, the Active Set Newton algorithm for nonnegative representations is a very good algorithm for solving the problem of signal separation that it was designed for, but it lacks of an efficient implementation. An efficient parallel implementation could increase the performance and make it usable in real-time speech separation problems. The algorithm solves a problem similar to the NNLS with multiple right hand sides but minimizing the Kullback-Leibler divergence instead of the Frobenius norm. Since the algorithm fits the nonnegative factorizations scope of this thesis and offers a great potential, the author of this work decided to implement an efficient parallel version of this particular algorithm, even thought the problem solved by the algorithm is not exactly an NMF problem.

1.3 Objectives

The main objective of this thesis is to develop an HPC computational library that contains efficient parallel implementations of the best algorithms from the current state of the art in order to solve the NMF. This library should include implementations for different HPC architectures so that the users can make the most of the hardware available. The main goal of this library is to offer the user a set of algorithms to compute the NMF with an easy-to-use interface and without the need for detailed knowledge of HPC hardware or programming. To achieve the main objective, other specific objectives need to be fulfilled. These necessary objectives are listed below:

- A study of the NMF, its properties, and the existing algorithms to compute the factorization. Research about existing implementations and libraries also needs to be performed. The algorithms need to be studied not only from a mathematical and algorithmical point of view, but also from a computational execution point of view. This is because an algorithm that has a lower theoretical cost does not always translate to a lower execution time. The study of algorithms for computing the NMF has shown the importance of the NNLS problem within the decomposition and the needs for its study.
- A study of the NNLS problem and the different algorithms used to solve it focusing on their computational capabilities.
- Development and testing of a methodology to update the solution of NNLS problems given the solution of a closely related NNLS problem.
- Development of an updating scheme for the NMF. Different algorithmic approaches for the updating of the NMF have been developed and tested.
- Development of efficient implementations of the studied algorithms for different HPC architectures in order to make the most of the available hardware.
- Comparison of the implementations developed to test the different algorithms with current hardware and software tools. This comparison has a double purpose: to test the performance of the implementations developed and to verify the accuracy and reliability of previous scientific works performed with those algorithms and the claims of the authors of the original algorithms.

1.4 Organization and key contributions of the Thesis

This document is structured in 11 chapters that cover most of the work performed. Chapters 2 to 4 include the known state of the art previous to this thesis. Chapters 5 and 6 describe algorithmic work and innovative solutions to problems, while Chapters 7 to 10 focus on the efficient implementation and evaluation of known algorithms using HPC technologies. Finally, chapter 11 presents some practical applications of the work performed during the development of this thesis and the final conclusions. A brief description of the chapters and the key contributions of this thesis are listed in the following:

- **Chapter 2:** This chapter contains a description and the mathematical definition of the Nonnegative Least Squares Problem and the Nonnegative Matrix Factorization. Furthermore, the chapter includes the main algorithms that are used nowadays to solve the NNLS problem and to compute the NMF. The chapter represents the state of the art of the methods that are used to compute the factorizations, which are the main topic of this PhD thesis. The algorithms introduced in this chapter are used in the work showed in later chapters.

- **Chapter 3:** This chapter introduces the basic high performance computing concepts and enumerates the different computing architectures that are currently most widely used in HPC environments. In addition, the software technologies needed to make the most of those architectures and the mathematical libraries that are key in the development of efficient HPC mathematical algorithms are described.

- **Chapter 4:** This chapter describes the existing NNMFPack library developed by the University of Oviedo and the Polytechnic University of Valencia. This library is the precursor of the library that has been developed during this thesis. There has been close collaboration with the developers of the NNMFPack library to continue the project and create the new NNMFPack library.

- **Chapter 5:** This chapter introduces the problem of updating the solution of NNLS problems. In the chapter, the different problems related to the updating of the NNLS problem are defined and an algorithmic methodology is proposed to solve them. Furthermore, some empirical analyses are performed using MATLAB coded algorithms to test the performance and accuracy of the proposed methodology. The key contributions of this chapter are the mathematical description of the updating problem and its different variations, and the algorithmic methodology developed to find the solution to the updating problem. The algorithms developed under that methodology proved to solve the updating problem faster than solving the related NNLS problem from scratch.

- **Chapter 6:** This chapter follows the ideas of Chapter 5 to tackle the updating of the NMF. In the chapter, an algorithmic methodology is presented and several solutions to the problem are proposed. Then an experimental analysis is performed using MATLAB coded algorithms to evaluate the proposed solutions and to show the advantages and disadvantages of each one. The key contributions of this chapter are the proposed algorithms that are designed to solve the updating of the NMF. Furthermore, a real application on automatic music transcription has been presented to show the potential of the updating of the NMF.

- **Chapter 7:** In this chapter, the efficient implementation of the ASNA algorithm is presented. The ASNA is an algorithm that was developed to solve a problem that is equivalent to a NNLS with multiple right hand sides but that minimizes the β-divergence instead of the Frobenius norm. The algorithm was developed to perform speech separation with overcomplete dictionaries. In the chapter, an efficient MATLAB version of the algorithm and two C versions are presented together with an experimental evaluation to compare all of the proposed implementations. Then the use of bigger dictionaries is tested motivated by the reduction of execution time achieved by the proposed versions with respect to the original implementation. The key contribution of this chapter is the reduction in execution time obtained by the implementations developed. That reduction in the execution time allowed to approach real-time applications such as speech enhancement with the ASNA algorithm.

- **Chapter 8:** This chapter shows the implementation details of all of the algorithms implemented during the development of this thesis, together with some discussion about the different implementations. The key contribution of this chapter is the presentation of a GPU implementation of the fHALS algorithm.

- **Chapter 9:** This chapter shows an experimental evaluation of the proposed implementations. It also shows a comparison between the different implementations and some discussion of the experimental results. Also, a comparison between CPU and GPU is performed for the algorithms that have been implemented on both architectures. Furthermore, the results of an evaluation of the influence of the β parameter performed with the multiplicative β-divergence algorithm for NMF are presented. This chapter contributes to deepening the knowledge about the studied algorithms from a computational point of view by means of empirical experimentation. In addition, it shows the difference between the theoretical performance of the algorithms and the performance obtained by implementing them in an efficient way using HPC technologies.

- **Chapter 10:** This chapter describes the HPC library for the NMF and NNLS developed during this thesis. The improvements with respect to the existing NNMFPack library (presented in Chapter 4) are discussed and the new implementations of the algorithms are explained. This chapter presents the key contribution and main objective of the thesis. The algorithms developed and tested in the previous chapters are finally included in an easy-to-use HPC computational library, which is the main contribution of this thesis to the scientific community.
- **Chapter 11:** This chapter presents the conclusions of this thesis and enumerates the scientific publications derived from the work performed during the development of the thesis.

Chapter 2

Nonnegative factorizations: definition and algorithms

2.1 Nonnegative least squares problem

There are many problems in several science and engineering fields whose data is nonnegative and therefore the least squares problem usually used to solve those problems needs to be adapted to this restriction [1]. Applying that restriction, the Nonnegative least squares problem (NNLS) problem is defined as follows:

Given a coefficient matrix $A \in \Re^{m \times n}$ and a right hand side vector $b \in \Re^m$, the NNLS problem associated to A, b consists of finding a vector, $w \in \Re^n_+$, with $w_i \geqslant 0, i = 1, \ldots, n$, such that

$$w = \underset{x \geqslant 0}{\operatorname{argmin}} \|Ax - b\|_2. \tag{2.1}$$

The NNLS problem (2.1) is a convex optimization problem for which an optimal solution can be found.

A natural extension of this problem occurs when several NNLS problems share the same coefficient matrix $A \in \Re^{m\times n}$. This problem is called NNLS with multiple right hand sides and can be defined in matrix form as

$$W = \underset{X\geqslant 0}{\operatorname{argmin}} \|AX - B\|_F. \tag{2.2}$$

where $B \in \Re^{m\times k}$ and $W \in \Re_+^{n\times k}$. This problem is named half NMF by some authors, because it is equivalent to find the NMF of B without updating matrix A.

2.2 Nonnegative Matrix factorization

The Nonnegative Matrix Factorization (NMF) is a very popular tool in fields such as document clustering, data mining, machine learning, image analysis, audio source separation or bioinformatics [17, 18, 19, 20, 21, 22]. The goal of the NMF 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 addressed as the computation of two matrices W_s, H_s such that

$$W_s, H_s = \underset{W,H\geq 0}{\operatorname{argmin}} \|WH - A\|_F \tag{2.3}$$

It is important to note that problem (2.3) is a nonconvex optimization problem, and therefore, there may not be not a unique optimal solution for the problem.

In certain science fields, the NMF is computed minimizing other target functions instead of the Frobenius norm, for example, alpha-divergence, beta-divergence, Kullback-Liebler divergence, etc. [23, 24, 25, 26, 27, 28, 29, 30, 31].

Many algorithms have been proposed for NMF calculation [20, 23, 14, 32, 33, 34], the current most relevant algorithms will be addressed on Section 2.4.

2.3 Algorithms to solve the Nonnegative Least Squares problem

Most algorithms to solve the NNLS problem (2.1) are based on the search for a pair of vectors x, y verifying the Karush-Kuhn-Tucker (KKT) conditions, which for this problem are

$$\begin{aligned} y &= A^T Ax - A^T b, \\ y &\geqslant 0, \\ x &\geqslant 0, \\ x_i y_i &= 0, i = 1 \ldots n. \end{aligned} \tag{2.4}$$

Given a NNLS problem (2.1), equations (2.4) form its linear complementary problem (LCP). If matrix A has full column rank, the product $A^T A$ is positive definite and the strictly monotone LCP (2.4) has unique solutions for each vector b. In this section the most used algorithms to find those vectors x and y will be outlined.

2.3.1 Active set algorithms

One of the main families of algorithms designed to solve problem (2.1) (and, more generally, linear least squares problems with linear restrictions) is the family of active set algorithms (also know as principal pivoting algorithms). See, for example, [8, 35, 12]. These algorithms divide the set of indexes $\{1,2,\ldots n\}$ into two sets, F and G, such that $F \cup G = \{1,2,\ldots n\}$ and $F \cap G = \emptyset$. Let x_F, x_G, y_F, and y_G denote the subsets of variables with corresponding indices in F or G, and let A_F and A_G denote the submatrices of A with corresponding column indices. A complementary basic solution is obtained by setting $x_G = 0$ and $y_F = 0$ in the conditions (2.4). The values of x_F and y_G may be computed as follows:

1. Solve the unconstrained linear least squares problem $\min \|A_F x_F - b\|^2$.
2. Set $y_G = (A_G)^T (A_F x_F - b)$.

If a complementary basic solution (x_F, y_G) satisfies $x_F \geq 0$ and $y_G \geq 0$, then it is called feasible. In this case current x is the optimal solution of (2.1), and the algorithm terminates. Otherwise, the solution (x_F, y_G) is infeasible. Then, the principal pivoting algorithms search the optimal sets F and G, by exchanging indexes between F and G. Once the optimal set is found, the solution is given by

$$x_i = \begin{cases} x_F & i \in F \\ 0 & i \in G \end{cases}$$

The Active Set method proposed in [1] is arguably the simplest principal pivoting method. As shown in Algorithm 1, the main distinguishing features of this algorithm are that only one index is exchanged between sets in each iteration, and that in the first iteration one set of indices (G) is initialized to $\{1,2,\ldots n\}$ and the other one (F) is taken as the empty set. Only the indices that violate the KKT conditions (2.4) can be exchanged. After some iterations, the optimal sets of indices will be found, as well as the optimal solution. This algorithm has been one of the most popular algorithm to solve the NNLS problem, and several improved algorithms based on it have been developed for certain scenarios [4, 5, 6, 12].

Block principal pivoting algorithm (BPP)

In many cases, it is more efficient to exchange several indices in each iteration of the algorithm. This idea is the basis of the Block Principal Pivoting algorithm (BPP) [32, 12]. This algorithm is similar to the Active Set method shown in Algorithm 1, but it allows any index partition to be used as initialization, and also allows to exchange multiple indices between the two sets at each iteration.

Algorithm 2 summarizes the BPP algorithm. Using the full exchange rule in Line 16 of Algorithm 2 when $V = \hat{V}$ may lead to an infinite cycle. To avoid that, in lines 6-15 the algorithm changes to a single index exchange after α iterations without decreasing the number of infeasibilities ($|V|$) , when $|V|$ starts to decrease again the algorithm returns to the full exchange rule. Parameter α can be chosen arbitrarily. For example, in [12] the authors proposed $\alpha \leqslant 10$; in [8], a value $\alpha = 3$ is proposed.

An extension of the BPP algorithm for the NNLS problem with multiple right hand sides was presented in [35, Alg. 2]. This algorithm (shown in Algorithm 3) is designed to solve problem (2.2) and involves interesting ideas to increase the performance of the computation. The column grouping strategy groups columns with the same F and G sets in order to avoid computing multiple times the Cholesky decomposition of $A_F^T A_F$. That decomposition is used to solve the equation on Line 21.

Algorithm 1 Lawson & Hanson Active Set algorithm

Input: $A \in \Re^{m \times n}, b \in \Re^m$
Output: $x \in \Re^n_+$ such that $x = NNLS(A,b)$
$F = \varnothing$; $G = [1:n]$; $x = 0$;
$y = A^T(b - Ax)$;
while $G \neq \varnothing$ **and** $y_j > 0$, for any $j \in G$ **do**
    Find $t \in G$ such that $y_t = max\{y_j : j \in G\}$;
    $F = F \cup \{t\}$; $G = G - \{t\}$;
    $\hat{z} = argmin_{z \in \Re^{|F|}} \|A_F z - b\|_2$, with $A_F = \left[A_{i_1}, A_{i_2}, ..., A_{i_k}\right]$, $i_j \in F$, $k = |F|$;
    $z = 0$;
    for $i = 1 : |F|$ **do**
        $z_{F(i)} = \hat{z}_i$;
    end for
    if $z_j > 0$, for all $j \in F$ **then**
        $x = z$;
    else
        while $z_j \leqslant 0$, for any $j \in F$ **do**
            Find $q \in F$ such that $x_q/(x_q - z_q)$=$min_{j \in F, z_j \leqslant 0}\{x_j/(x_j - z_j)\}$;
            $\alpha = x_q/(x_q - z_q)$;
            $x = x + \alpha(z - x)$;
            $F = F - \{j \in F \text{ such that } x_j = 0\}$;$G = G \bigcup \{j \in F \text{ such that } x_j = 0\}$;
            $\hat{z} = argmin_{z \in \Re^{|F|}} \|A_F z - b\|_2$ with $A_F = \left[A_{i_1}, A_{i_2}, ..., A_{i_k}\right]$, $i_j \in F$, $k = |F|$;
            $z = 0$;
            for $i = 1 : |F|$ **do**
                $z_{F(i)} = \hat{z}_i$;
            end for
        end while
        $x = z$;
    end if
    $y = A^T(b - Ax)$;
end while

Algorithm 2 Block Principal Pivoting algorithm (BPP)

Input: $A \in \Re^{m \times n}, b \in \Re^m, F, G$
Output: $w \in \Re^n_+$ such that $w = NNLS(A,b)$
$x = 0; y = A^T(b - Ax)$
$\beta = n+1$; $\alpha = 3$;
while $x_{F_i} < 0$, for some i **or** $y_{G_i} < 0$, for some i **do**
    $V = \{i \in F : x_i < 0\} \cup \{i \in G : y_i < 0\}$;
    if $|V| < \beta$ **then**
        $\beta = |V|$; $\alpha = 3$; $\hat{V} = V$;
    else if $|V| \geqslant \beta$ **and** $\alpha \geqslant 1$ **then**
        $\alpha = \alpha - 1$; $\hat{V} = V$;
    else if $|V| \geqslant \beta$ **and** $\alpha = 0$ **then**
        $\hat{V}$=$\{i : i = max\{i \in V\}\}$;
    end if
    $F = (F - \hat{V}) \cup \{\hat{V} \cap G\}$; $G = (G - \hat{V}) \cup \{\hat{V} \cap F\}$;
    $aux = \underset{z \in \Re^{|F|}}{\operatorname{argmin}} \|A_F z - b\|_2$;
    $y_G = A_G^T(A_F aux - b)$;
    $x = 0 \in \Re^n$;
    for $i = 1 : |F|$ **do**
        $x_{F(i)} = aux_i$;
    end for
end while
$w = x$

Algorithm 3 Block Principal Pivoting algorithm for the NNLS problem with multiple right hand sides

Input: $A \in \Re^{m \times n}, B \in \Re^{m \times r}, F, G$
Output: $W \in \Re_{+}^{n \times r}$ such that $W = NNLS(A, B)$
Compute $A^T A$ and $A^T B$
Initialize $X = 0; Y = -A^T B$
$F_j = \varnothing$ and $G_j = \{1, ..., q\}$ for all $j \in \{1, ..., r\}$
$\beta = n + 1$; $\alpha = 3$; $\alpha, \beta \in \Re^r$
Compute X_{Fj} solving $A_F^T A_F x_F = A_F^T b$ using column grouping
Compute Y_{Gj} solving $y_G = A_G^T A_F x_F - A_G^T b$ using column grouping
while any(X_{Fj}, Y_{Gj}) is infeasible **do**
    Find infeasible columns $I = \{j : (X_{Fj}, Y_{Gj})$`is infeasible`$\}$
    for all $j \in I$ **do**
        $V_j = \{i \in F_j : x_i < 0\} \cup \{i \in G_j : y_i < 0\}$;
        if $|V_j| < \beta_j$ **then**
            $\beta_j = |V_j|$; $\alpha_j = 3$; $\hat{V}_j = V_j$;
        else if $|V_j| \geqslant \beta_j$ **and** $\alpha_j \geqslant 1$ **then**
            $\alpha_j = \alpha_j - 1$; $\hat{V}_j = V_j$;
        else if $|V_j| \geqslant \beta_j$ **and** $\alpha_j = 0$ **then**
            $\hat{V}_j = \{i : i = max\{i \in V_j\}\}$;
        end if
        $F_j = (F_j - \hat{V}_j) \cup \{\hat{V}_j \cap G_j\}$; $G_j = (G_j - \hat{V}_j) \cup \{\hat{V}_j \cap F_j\}$;
        Compute X_{Fj} solving $A_F^T A_F x_F = A_F^T b$ using column grouping
        Compute Y_{Gj} solving $y_G = A_G^T A_F x_F - A_G^T b$ using column grouping
    end for
end while
$W = X$

2.4 Algorithms for the NonNegative Matrix Factorization

Nowadays, there are a wide range of known algorithms to compute the NMF and one of the goals of this thesis was to study them in order to achieve efficient implementations of the best algorithms to compute the factorization. In this chapter the most used algorithms will be summarized and commented from the computational point of view.

2.4.1 Multiplicative algorithms

One of the most used algorithms to compute the NMF are the multiplicative updates proposed by Lee and Seung in [23]. This well known algorithm uses a set of multiplicative rules to update the matrices W and H:

$$H \leftarrow H \cdot \frac{W^T A}{W^T (WH)} \quad W \leftarrow W \cdot \frac{AH^T}{WHH^T} \tag{2.5}$$

where the division and the $\cdot$ are taken entrywise.

The convergence of the algorithm to a solution of problem 2.3 was also proven in [23].

This multiplicative update rules are widely used due to its simplicity and good performance. The convergence per iteration is slow. But the low computational cost of each iteration allows the algorithm to be competitive against other NMF algorithms [36].

An efficient algorithm to compute the NMF based on the Lee and Seung algorithm was presented in [37]. This algorithm keeps the same cost per iteration but improves the convergence per iteration by updating the matrix W using the matrix H computed in the same iteration. The modified Lee and Seung algorithm (MLSA) is shown in Algorithm 4 and its computational cost per iteration is:

$$\mathcal{O}(4mnk + 4k^2(m+n)) \quad flops/iteration \tag{2.6}$$

Algorithm 4 Modified Lee and Seung Algorithm (MLSA)

Input: $A \in \mathbb{R}_+^{m\times n}$, $W0 \in \mathbb{R}_+^{m\times k}$, $H0 \in \mathbb{R}_+^{k\times n}$
$k < min(m,n), maxIter > 0$
Output: $W \in \mathbb{R}_+^{m\times k}, H \in \mathbb{R}_+^{k\times n}$
$W = W0; H = H0$
for $iter = 1,2,...,maxIter$ **do**
  $B = W^T W H; C = W^T A;$
  for $i = 1,2,...,k$ **do**
    for $j = 1,2,...,n$ **do**
      $H(i,j) = \frac{H(i,j)\cdot C(i,j)}{B(i,j)}$
    end for
  end for
  $D = WHH^T; E = AH^T;$
  for $i = 1,2,...,m$ **do**
    for $j = 1,2,...,k$ **do**
      $W(i,j) = \frac{W(i,j)\cdot E(i,j)}{D(i,j)}$
    end for
  end for
end for

2.4.2 Affine NMF algorithm

Sometimes, when there exists a common part in all the observations in the data matrix A, the standard NMF algorithm fails to obtain a proper parts decomposition due to that common part. The affine NMF model proposed in [38, Ch. 3.6] and shown in Algorithm 5 tries to solve that problem by introducing an offset w_0 to the general NMF model:

$$A \approx WH + w_0 \mathbb{1}^T \tag{2.7}$$

where $\mathbb{1}$ denotes an all one vector. The offset w_0 should absorb the common part offering a better parts decomposition than the obtained with the general multiplicative algorithm.

The advantages of this algorithm is clearly shown in the swimmer benchmark example in [38, Ch. 3.9]. The benchmark is included in Figure 2.1 for the sake of clarity. In that benchmark the difference between the multiplicative algorithm and the Affine NMF algorithm is clearly seen. With the Affine NMF algorithm, the "body" of the swimmer is absorbed by the offset w_0. While with the multiplicative algorithm, the "body" is split between all basis contaminating them.

Algorithm 5 Affine NMF algorithm

Input: $A \in \mathbb{R}_+^{m \times n}$ $W0 \in \mathbb{R}_+^{m \times k}, H0 \in \mathbb{R}_+^{k \times n}, w0 \in \mathbb{R}_+^m$

$k < min(m,n), maxIter > 0$

Output: $W \in \mathbb{R}_+^{m \times k}, H \in \mathbb{R}_+^{k \times n}, w \in \mathbb{R}_+^m$

$W = W0; H = H0; w = w0$

for $iter = 1, 2, ..., maxIter$ **do**

  $\hat{A} = WH + w1^T$

  $N = AH^T; D = \hat{A}H^T$

  $W = W \cdot N./D$

  Column normalization of W using 1-norm

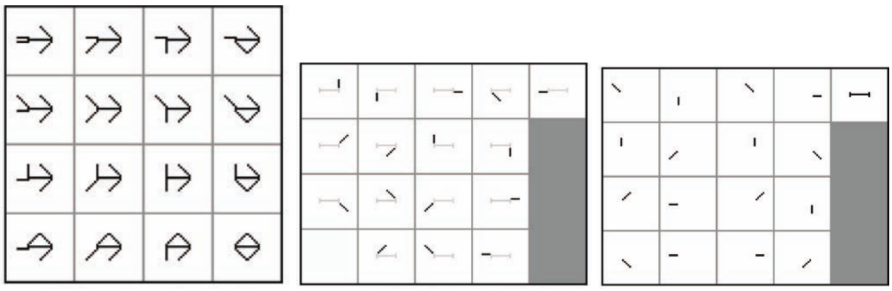
  $N = W^T A; D = W^T \hat{A}$

  $H = H \cdot N./D$

  $n = A1; d = \hat{A}1$

  $w = w \cdot n./d$

end for



(a) Original data matrix

(b) Basis components obtained by the Multiplicative algorithm

(c) Basis components obtained by the Affine NMF algorithm

Figure 2.1: Comparison of the basis components (matrix W) obtained by the multiplicative algorithm (b) and by the Affine NMF algorithm (c) for the swimmer benchmark (a). On (c) the upper rigth element corresponds to the offset w_0.

2.4.3 Beta divergence multiplicative algorithms

As stated in Section 2.2 different norms or divergences can be used to define the NMF minimization problem. Problem (2.3) can be redefined as:

$$W,H = \underset{W,H\geq 0}{\operatorname{argmin}} D_\beta(A,WH) \tag{2.8}$$

where D_β is the β-divergence between two matrices that can be expressed as:

$$D_\beta(X,Y) = \sum_i^m \sum_j^n d_\beta(X_{i,j},Y_{i,j}) \tag{2.9}$$

The β-divergence was introduced by Eguchi and Minami, see [39], as an error measure. It can be defined as (see, *e.g.* [40]):

$$d_\beta(x|y) := \begin{cases} \frac{1}{\beta(\beta-1)}(x^\beta + (\beta-1)y^\beta - \beta x y^{\beta-1}), & \text{if } \beta \in \mathbb{R}\setminus\{0,1\}, \\ x(\log x - \log y) + (y-x), & \text{if } \beta = 1 \\ \frac{x}{y} - \log\frac{x}{y} - 1, & \text{if } \beta = 0. \end{cases} \tag{2.10}$$

The previous cost function is defined for all real numbers, but values of β between 0 and 2 are usually considered on practical applications. Mathematically, this divergence is equal to other distances and divergences for some particular values of β: the Frobenius norm ($\beta = 2$), the Kullback-Leibler divergence ($\beta = 1$) and the Itakura-Saito divergence ($\beta = 0$).

As exposed in [40, sec. 2.1], by using the gradient of $D_\beta(X,Y)$ is possible to obtain the following rules to update matrices H and W:

$$H \leftarrow H \cdot \frac{W^T((WH)^{\cdot\beta-2}\cdot A)}{W^T(WH)^{\cdot\beta-1}}, \quad W \leftarrow W \cdot \frac{((WH)^{\cdot\beta-2}\cdot A)H^T}{(WH)^{\cdot\beta-1}H^T}, \tag{2.11}$$

where $X^{\cdot n}$ denotes the matrix with entries $([X]_{ij})^n$ and the division is taken entrywise.

This multiplicative β-divergence algorithm maintains the advantages of simplicity and low cost per iteration of the multiplicative update rules from Lee and Seung but with the flexibility offered by the β-divergence, which is the ability to adapt the divergence measure of the algorithm to the one that best suits the application approached [28, 39, 29]. Choosing the appropriate divergence can improve the quality of the decomposition.

2.4.4 Alternating Least Squares algorithms

Another well know scheme used to compute the NMF is the Alternating Least Squares (ALS) family of algorithms. This algorithms first presented in [14] perform a Least Square step to compute H keeping W fixed followed by another Least Square step to compute W keeping H fixed, there comes the Alternating Least Squares name. The main advantage of this scheme is that while problem 2.3 is a nonconvex problem, the subproblems 2.12 and 2.13 are convex, therefore an optimal solution can be found.

$$H = \underset{H \geq 0}{\operatorname{argmin}} ||WH - A|| \qquad (2.12) \qquad W = \underset{W \geq 0}{\operatorname{argmin}} ||H^T W^T - A^T|| \qquad (2.13)$$

The basic approach for this algorithm [14, 20] is to perform an unconstrained least squares step followed by a projection of the negative values to 0 to enforce nonnegativity. This approach produces Algorithm 6.

Algorithm 6 Alternating Least Squares Algorithm (ALS)

Input: $A \in \mathbb{R}_+^{m \times n}$
1: $k < min(m,n), maxIter > 0$
Output: $W \in \mathbb{R}_+^{m \times k}, H \in \mathbb{R}_+^{k \times n}$
2: $W = rand(m,k)$ %initialize positive W
3: **for** $iter = 1, 2, ..., maxIter$ **do**
4: $H = \operatorname{argmin} ||WH - A||$
5: $\forall i, j \quad H(i,j) = max(H(i,j), 0)$
6: $W = \operatorname{argmin} ||H^T W^T - A^T||$
7: $\forall i, j \quad W(i,j) = max(W(i,j), 0)$
8: **end for**

This ALS algorithm does not solve properly problems (2.12) and (2.13), and does not ensure convergence to an optimal solution. However, it is a widely used algorithm because it is faster than the algorithms that try to solve problems (2.12) and (2.13) exactly. There are various NMF algorithms based on this framework (e.g. [14, 20, 35, 32] or [38, Sec. 4.1]). The MATLAB function to solve the NMF uses this algorithm as its default algorithm.

Nevertheless, if a proper nonnegative least squares (NNLS) constrained algorithm (see Section 2.3) is used to solve the alternated problems, the convergence to a stationary point is proved in [41]. This framework is commonly named Alternating Nonnegative Least Squares (ANLS) and there are several implementations following it [35]. The ANLS framework has the advantage of guaranteeing convergence to a local minimum. But due to the greater computational cost of solving two NNLS problems in each iteration

(greater than the cost of solving unconstrained least squares steps), is much slower than the ALS framework.

In [42], the authors proposed an algorithm based on the ANLS framework focused on lowering the computational cost and using efficient operations to decrease the execution time. The algorithm called ALSA is shown in Algorithm 7.

Algorithm 7 Alternating Least Squares Algorithm (ALSA)

Input: $A \in \mathbb{R}^{m \times n}, A(i,j) \geq 0 \forall i,j$
1: $k < min(m,n)$
Output: $W \in \mathbb{R}^{m \times k}, H \in \mathbb{R}^{k \times n}, W(i,j), H(i,j) \geq 0 \forall i,j$
2: $W = rand(m,k)$ % initialize positive W
3: **for** $iter = 1,2,...,\rightarrow convergence$ **do**
4: $[Q,R] = qr(W)$;
5: $B = Q^T A$
6: **for** $j = 1,2,...,n$ **do**
7: $H(:,j) = \text{argmin}_{h \geq 0} ||Rh - B(:,j)||$
8: **end for**
9: $[Q,R] = qr(H^T)$;
10: $B = Q^T A^T$
11: **for** $j = 1,2,...,m$ **do**
12: $W(j,:) = \text{argmin}_{w \geq 0} ||Rw - B(:,j)||$
13: **end for**
14: **end for**

ANLS-BPP

An algorithm based on the ANLS framework using the Block Principal Pivoting (BPP) algorithm [8] is presented in [35]. In this algorithm a version for multiple right hand sides of the Block Principal Pivoting algorithm, which solves the NNLS problem, is used to solve problems (2.12) and (2.13) (see Section 2.3.1). It is also shown in [35] that the ANLS-BPP algorithm is faster than other algorithms using the ANLS framework. Furthermore ANLS-BPP can compete with the fast HALS algorithm (see Section 2.4.5) outperforming it in some datasets. ANLS-BPP is usually faster than fast HALS with sparser factors.

2.4.5 Hierarchical Alternating Least Squares (HALS) algorithms

The Hierarchical Alternating Least Squares (HALS) algorithm first presented in [34] and developed in [43] is an algorithm derived from the ALS algorithm. But due to its good performance it deserves its own section. The performance of the HALS algorithm has been proved against other state of the art algorithms in numerous studies [43, 35, 44, 45] and [38, Sec. 4.8]. The HALS algorithm shown in Algorithm 8 uses a set of alternating update rules until some convergence criterion (not stated by the author of the algorithm) is met.

Algorithm 8 Hierarchical Alternating Least Squares Algorithm (HALS)

Input: $A \in {\mathbb{R}_+}^{m \times n}$, $k < min(m,n)$
Output: $W \in {\mathbb{R}_+}^{m \times k}$, $H = B^T \in {\mathbb{R}_+}^{k \times n}$,

1: Initialize nonnegative matrix W and/or $X = B^T$ using ALS
2: Normalize the vectors w_j (or b_j) to unit $l_2 - norm$ length
3: $E = A - WB^T$
4: **while** convergence criterion not reached **do**
5: **for** $j = 1, 2, ..., k$ **do**
6: $A^{(j)} \Leftarrow E + w_j b_j^T$
7: $b_j \Leftarrow [A^{(j)T} w_j]_+$
8: $w_j \Leftarrow [A^{(j)} b_j]_+$
9: $w_j \Leftarrow w_j / ||w_j||_2$
10: $E \Leftarrow A^{(j)} - w_j b_j^T$
11: **end for**
12: **end while**

Beta divergence HALS

As seen with the multiplicative update algorithm in Section 2.4.1, there is a version of the HALS algorithm that minimizes β-divergence instead of Frobenius distance. The update rules needed to minimize the β-divergence in Algorithm 8 are 2.14 in place of line 7 and 2.15 in place of line 8.

$$b_j \Leftarrow \frac{([A^{(j)T}]_+) w_j^{[\beta-1]}}{w_j^T w_j^{[\beta-1]}} \qquad (2.14) \qquad\qquad w_j \Leftarrow \frac{([A^{(j)}]_+) b_j^{[\beta-1]}}{(b_j^T)^{[\beta-1]} b_j} \qquad (2.15)$$

These update rules are developed in [38, Sec. 4.7.9] but the β-divergence is defined in a different way in that book (e.g. the value of the β parameter which is equivalent to Frobenius distance is $\beta = 1$ instead of $\beta = 2$). Due to that, the update rules presented here are adapted to the β-divergence function defined on Section 2.4.3.

Fast HALS

In [43, Sec 3] an improved version of HALS algorithm called fast HALS (fHALS) was introduced. This version has a lower computational cost than the basic version and therefore is much faster. Fast HALS algorithm is shown in Algorithm 9. It can be seen that the most of the algorithm cost comes in form of matrix-matrix products, so implementing this algorithm with BLAS3 operations (see Chapter 3) can yield a tremendous performance in modern systems and even more in GPUs.

There is some confusing reference to HALS algorithm in the bibliography because most of the times the HALS algorithm is referenced, the fHALS version is in fact the one used (e.g. in [35] the HALS algorithm is compared with other NMF algorithms, but checking the results of the executions makes clear that fHALS is the algorithm used in the tests).

The fHALS algorithm is currently one of the fastest methods to compute the NMF and the ones that are faster or equivalent do not overcome fHALS for all cases. Furthermore, the fHALS algorithm is very easy to implement and usually easier than other fast algorithms for NMF. The fHALS algorithm is the fastest algorithm between all the algorithm tested during this tesis (see Chapter 9). The computational cost per iteration of the fHALS algorithm is:

$$\mathcal{O}(4mnk + 4k^2(m+n)) \quad flops/iteration \tag{2.16}$$

Despite having the same higher order computational cost per iteration than MLSA, fHALS algorithm has a better overall performance because it has a faster convergence per iteration than MLSA. Usually fHALS algorithm needs less iterations (around one order of magnitude) than MLSA to achieve the same approximation error.

A version of the fHALS algorithm for β-divergence can be derived from the β-divergence HALS algorithm, following the same procedures that led to the fHALS algorithm from the HALS algorithm.

Algorithm 9 Fast HALS Algorithm

Input: $A \in {\mathbb{R}_+}^{m \times n}$

$k < min(m,n)$

Output: $W \in {\mathbb{R}_+}^{m \times k}, H = B^T \in {\mathbb{R}_+}^{k \times n}$,

Initialize nonnegative matrix W and/or $X = B^T$ using ALS

Normalize the vectors w_j (or b_j) to unit $l_2 - norm$ length

while convergence criterion not reached **do**

    %Update B

    $X = A^T W$

    $V = W^T W$

    for $j = 1, 2, ..., k$ **do**

        $b_j \Leftarrow [b_j + x_j - B v_j]_+$

    end for

    %Update W

    $P = AB$

    $Q = B^T B$

    for $j = 1, 2, ..., k$ **do**

        $w_j \Leftarrow [w_j q_{jj} + p_j - W q_j]_+$

        $w_j \Leftarrow w_j / ||w_j||_2$

    end for

end while

2.4.6 Greedy Coordinate Descent algorithm (GCD)

The Greedy Coordinate Descent algorithm (GCD) was introduced in [44] with the goal of improving the fHALS algorithm using an individual variable selection approach. The authors of [44] claim that their algorithm is more efficient than the fHALS algorithm because fHALS uses a cyclic coordinate descent schema that may perform unneeded descent steps on unimportant variables. On the other hand, GCD algorithm performs a variable selection to decide which are the important variables in which to perform the gradient descents, so the number of variables updated is much lower (specially in the sparse case). Furthermore, theoretical convergence guaranties are presented.

On top of that, the authors claim that GCD is faster than fast HALS in practice, but experiments during the development of this thesis had shown that Fast HALS is faster than GCD in modern computing systems so that claim seems to be incorrect. This topic will be addressed in depth in the experiments shown in Chapter 9.

In Algorithm 10 an overview of the algorithm is presented. In the inner loop on line 11, the variable selection strategy and the one-variable update are performed. Before the inner loop, the gradient G^W and objective function decrease D^W matrices are initialized. Before the main loop (line 4), some matrix-matrix products are precomputed to decrease the cost during the iterations. The cost of each iteration of the algorithm (loop of line4) depends on the number of coordinate updates performed in each iteration of the inner loop (line 11), which depends on the data in matrix H^{new}.

2.4.7 Online NMF

There are scenarios where all the data matrix A is not available or is too large to be loaded into memory for its processing. To solve that problem, several authors developed algorithms capable of estimating a NMF for large or incomplete datasets with tractable memory complexity. These algorithms are usually named Online NMF, because they are designed to compute the NMF as soon as new data points are available. Although the basic approach computes a NMF using the Frobenius norm [46, 47, 48], there are several Online NMF algorithms designed to minimize other divergences [49, 48]. The online NMF is often used for dictionary generation. This is why most Online NMF algorithms emphasise in obtaining an accurate approximation of matrix W, sacrificing the accuracy of matrix H.

In [47] an easy to understand approach to the Online NMF problem is presented. Algorithm 11 shows the structure used by the algorithms presented in [47]. Matrix W is updated using the projected gradient methods presented in [50], specifically the second order projected gradient descent (PGD).

Algorithm 10 Greedy Coordinate Descent algorithm

Input: $A \in {\mathbb{R}_+}^{m \times n}, k < min(m,n)$
1: $\varepsilon(typically, \varepsilon = 0.001)$
Output: $W \in {\mathbb{R}_+}^{m \times k}, H \in {\mathbb{R}_+}^{k \times n}$,
2: Compute $P^{AH} = AH^T, P^{HH} = HH^T, P^{WA} = W^T A, P^{WW} = W^T W$
3: Initialize $H^{new} \leftarrow 0$
4: **while** not converged **do**
5: Compute $P^{AH} \leftarrow P^{AH} + A(H^{new})^T$ according to the sparsity of H^{new}
6: $W^{new} \leftarrow 0$
7: $G^W \leftarrow WP^{HH} - P^{AH}$
8: $S^W_{ir} \leftarrow max(W_{ir} - \frac{G^W_{ir}}{P^{HH}_{rr}}, 0) - W_{ir}$ for all i, r
9: $D^W_{ir} \leftarrow -G^W_{ir} S^W_{ir} - \frac{1}{2} P^{HH}_{rr} (S^W_{ir})^2$ for all i, r
10: $q_i \leftarrow argmax_j D^W_{ij}$ for all $i = 1, ..., m$ and $p^{init} \leftarrow \max_i D^W_{i,q_i}$
11: **for** $i = 1, 2, ..., m$ **do**
12: **while** $D^W_{i,q_i} < \varepsilon p^{init}$ **do**
13: $s^* \leftarrow S^W_{i,q_i}$
14: $P^{WW}_{q_i,:} \leftarrow P^{WW}_{q_i,:} + s^* W_{q_i,:}$(Also do a symmetric update for $P^{WW}_{:,q_i}$)
15: $W^{new}_{i,q_i} \leftarrow W^{new}_{i,q_i} + s^*$
16: $G^W_{i,:} \leftarrow G^W_{i,:} + s^* P^{HH}_{q_i,:}$
17: $S^W_{ir} \leftarrow max(W_{ir} - \frac{G^W_{ir}}{P^{HH}_{rr}}, 0) - W_{ir}$ for all $r = 1, ..., k$
18: $D^W_{ir} \leftarrow -G^W_{ir} S^W_{ir} - \frac{1}{2} P^{HH}_{rr} (S^W_{ir})^2$ for all $r = 1, ..., k$
19: $q_i \leftarrow argmax_j D^W_{ij}$
20: **end while**
21: **end for**
22: $W \leftarrow W + W^{new}$
23: For updates to H, repeat analogue steps to Step 5 through Step 22
24: **end while**

Algorithm 11 Online NMF structure

Input: $A \in \mathbb{R}^{m \times n}_+$, where not all n data points are available from the beginning, $W0 \in \mathbb{R}^{m \times k}_+$, $r \in \mathbb{N}$
1: $W = W0$
2: **for** $t = 1 : n/r$ **do**
3: $A^{(t)} =$ Draw r data points from A.
4: Compute $H^{(t)}$ by solving the NNLS problem $H^{(t)} = \text{argmin}_{H \geqslant 0} \|WH - A^{(t)}\|_F$.
5: Update matrix W using $H^{(t)}$ and $A^{(t)}$
6: **end for**
7:

Chapter 3

High performance computing technologies

3.1 High performance computing basics

High Performance Computing (HPC) can be defined as the use of high performance computational hardware in order to obtain the highest performance possible to solve complex problems. HPC computing is widely used in the scientific and engineering communities due to the high complexity and computational cost that arises from the problems tackled nowadays. For example, natural phenomena simulation, solving high dimensionality mathematical problems or data analysis coming from scientific experiments. At present time, HPC is gaining relevance in other fields like the financial sector.

Due to the increasing need of computing power, different hardware architectures have evolved to obtain systems with great compute power. Those architectures will be explained in Section 3.2.

Together with the development of high performance hardware, HPC includes the software technologies developed to obtain the maximum performance from that hardware. To take advantage of the multiple processors present in HPC systems, the developers should use *parallel programming*. Parallel programming is a programming paradigm in

which several computations are performed simultaneously. Using parallelism improves the performance of the applications while increasing the programming difficulty due to the need of handling concurrency problems. Furthermore, depending on the hardware architecture of the system, there are several programming models to take advantage of that architecture (shared memory, distributed memory, coprocessor offloading etc.). In section 3.3 some software technologies used in HPC are described.

3.2 Current HPC architectures

The Flynn taxonomy [51] is a good entry point to the HPC architectures section, Flynn made a logical classification of computer architectures into 4 groups:

1. **SISD:** Simple Instruction Simple Data category describes the basic sequential computer, where one instruction at a time interacts with one data stream.

2. **MISD:** In Multiple Instruction Single Data multiple instructions are executed over the same data stream. Nowadays there are not common computers based on this paradigm.

3. **SIMD:** In Single Instruction Multiple Data architectures the same instruction is executed synchronously over multiple data. Vectorial computers and Graphics Procesing Units (GPU) fall into this category.

4. **MIMD:** The Multiple Instruction Multiple Data architectures are the most extended architectures nowadays. Into this category multiprocessors and multicomputers can be clasified.

This simple classification gives an overview of the mayor computing architectures used in HPC and in general purpose computation. Traditionally the main architectures used for HPC were MIMD, but nowadays with the improvement of the GPUs, most of the high end HPC systems are combinations of MIMD and SIMD architectures.

From an evolution point of view, the sequential processors (SISD) improved during years by means of increasing the number of transistors on chip and the CPU frequency. But this improvement in performance was paid with an increment of the power consumption of the processors. Furthermore the increment of CPU frequency and the decrease on the lithography size to increase the number of transistors was getting close to the physical limits of the semiconductors, on which the current processors are based. This problems were addressed by the processor manufacturers decision of including multiple CPUs in a single chip, and that decision changed the trend of SISD sequential processors to MIMD multicore processors. The number of cores (independent CPUs) on a single processor increased up to 8 or 16 cores in public commercial versions and up to 24 in professional

versions. Furthermore, the manufacturers developed technologies to execute several execution streams (threads) into the same physical core, doubling the number of logical cores obtained and increasing even more the performance. These architectures started the need to develop parallel computing programming techniques and environments to make the most of the hardware available.

In HPC and professional environments, due to the high computational requirements, several processors were build into the same computer to increase the performance of the system giving us the multiprocessor computers that are also MIMD machines. These multicore and multiprocesor computers can be classified as shared memory architectures because all the processors have direct access to all the memory on the system. This architecture is detailed in Section 3.2.1.

Due to the cost of scaling up the performance of a system increasing its number of processors and memory, another approach with lower cost became popular. Connecting different computers with high bandwidth and low latency interconnection networks was cheaper than building a computer alone with the same computational power. Those systems called multicomputers (commonly known as computer clusters) can be easily scaled to achieve massive amounts of compute power with a reasonable economic cost. This systems are MIMD and are commonly classified as distributed memory architectures because each process only has access to its own local memory, communicating with other processes by means of message passing. More detailed information can be found in Section 3.2.2.

Another architecture currently being used in HPC are the Graphics Processing Units (GPUs). This SIMD architecture was developed for the videogame industry, but when its compute power started to grow, the interest on those architectures in the HPC community grew with it. But it was with the release of CUDA (see Section 3.3.3) when the GPUs started to take a very important position in HPC systems. CUDA is a programming environment for GPUs based on the C programming language developed by the GPU manufacturer NVIDIA. See Section 3.2.3 for more information about the GPU architecture.

Currently, most of the high end HPC systems are a combination of multiprocessor computers with GPUs interconnected in a cluster with huge combined computational power.

3.2.1 Shared memory

In shared memory architectures all processors can access the whole memory pool of the system (the processors share the same address space). The communication between processes is made by reading and writing positions of the memory. Another subclassification addresses the access time of the different processors to memory: if the memory access time is the same for all positions of memory in the address space, the system is classified as *Uniform Memory Access* (UMA), if the memory access time differs between different memory addresses, the system is classified as *Non Uniform Memory Access*(NUMA).

Developing software over shared memory architectures is usually easier because the developer does not need to handle the communications between processes as in distributed memory architectures. The only problem to take care in this architecture is to avoid concurrence problems, using synchronization methods when several threads are reading and writing over the same memory addresses. A thread is an execution stream that can be executed by a core of the processor, a multicore processor can execute concurrently as many threads as its number of physical cores. Nowadays, most multicore CPU manufacturers include technologies to overlap the execution of several (typically two) threads into the same physical core, but in HPC applications this core sharing technologies usually have a negative impact in the application performance. Thus, is usually advised to disable it in massively parallel applications.

There are many parallel programming utilities to program over shared memory machines like *POSIX threads*, *Intel TBB* or *OpenMP*. But due to its simplicity and performance *OpenMP* is the most extended paradigm for shared memory architectures. A brief description of *OpenMP* standard is given in Section 3.3.1.

In the past, shared memory machines were shadowed by distributed memory machines due to the high cost of scaling the computational power of shared memory machines, but nowadays with the improvement of the hardware capabilities there are very powerful shared memory machines within a reasonable price. That is why shared memory machines regained relevance nowadays in some research groups.

3.2.2 Distributed memory

The popularity of this architecture relies on its lower cost to achieve huge amounts of processing power and on its great scalability. In a computer cluster, each individual computer called node has one or more processors and its own memory pool. Then, several nodes are connected with high bandwidth and low latency interconnection networks to merge all of them into a single powerful system. From the scalability point of view, a new node can be easily added to the system if more computational power is needed, while scaling a shared memory machine is more difficult and expensive.

A computer cluster can range from a low cost system where the nodes are commodity workstations, to the most powerful supercomputers where each node is formed by high performance multiprocessors. Nowadays most of the high end supercomputers are made of multiprocessor nodes with accelerators (GPUs and/or manycores).

In distributed memory systems each processor can only access its local memory and has its own memory address space. When a process needs to access data in the local memory of a different processor, it will communicate with the process running in that processor via message passing to receive that data. Nowadays most of the nodes are shared memory multiprocessors themselves sharing a address space between the processors of the same node and communicating via message passing with the processors in different nodes.

Programming over distributed memory architectures is harder than over shared memory computers, because the developer should be aware of the node which contains the data needed by a process, and he should explicitly send or receive messages in order to move the data to the desired nodes. The communication time between the processes has a big impact in the overall performance of the parallel program executed on the cluster. Furthermore, the communication time between different nodes can be different, so the developer should acknowledge this to exploit the *data locality* to minimize the messages between the processors with higher communication times. There has been different message passing libraries to help the developers with higher level interfaces and tools to manage the process communications(e.g. Parallel Virtual Machine (PVM) or Message Passing Interface (MPI)). Nowadays the MPI standard is the most used paradigm to develop parallel applications for computer clusters. A brief description of the standard will be given in Section 3.3.2.

3.2.3 Graphical Processing Units

The Graphical Procesing Uints (GPUs) are SIMD coprocessors designed to accelerate computer graphics. The improvements in performance of the GPUs was motivated by the videogame industry, and the increasing needs of compute power to improve the graphical aspects of videogames. Due to the improvement of the overall performance of the GPUs the HPC and scientific communities started to use GPUs to solve problems non related to graphics, that trend was called General Purpose Computing on Graphics Preocesing Units (GPGPU).

GPUs have a higher number of cores than CPUs, but GPU cores are more limited and simple, with a smaller instruction set. Furthermore, due to its SIMD architecture all the cores are used to execute synchronously the same instruction, but nowadays, the modern GPU architectures are partitioned into multiprocessors that can execute different instruc-

tions concurrently. Each multiprocessor is a SIMD processor with a variable number of cores (depending on the architecture) executing the same instruction.

Nowadays the main GPU manufacturers AMD [52] and NVIDIA [53] have solutions for GPGPU but in the last years the dominating manufacturer in the HPC field has been NVIDIA. It was with the launch of the CUDA framework (see Section 3.3.3) and the Tesla architecture by NVIDIA when the GPUs presence in HPC systems started to grow. This proprietary framework together with the high performance of their GPGPU products keeps NVIDIA in a dominant position.

The latest computation accelerator, with the latest NVIDIA architecture called *Volta* [54], achieves up to 7.8 TFlops on double precision and 15.7 TFlops on single precession. But the research group in which this thesis has been carried out owns a machine with the previous generation of NVIDIA GPGPUs which architecture is called *Pascal* [55] and achieves upt to 5.3 TFlops on double precision. In the following lines the *Pascal* architecture will be briefly described.

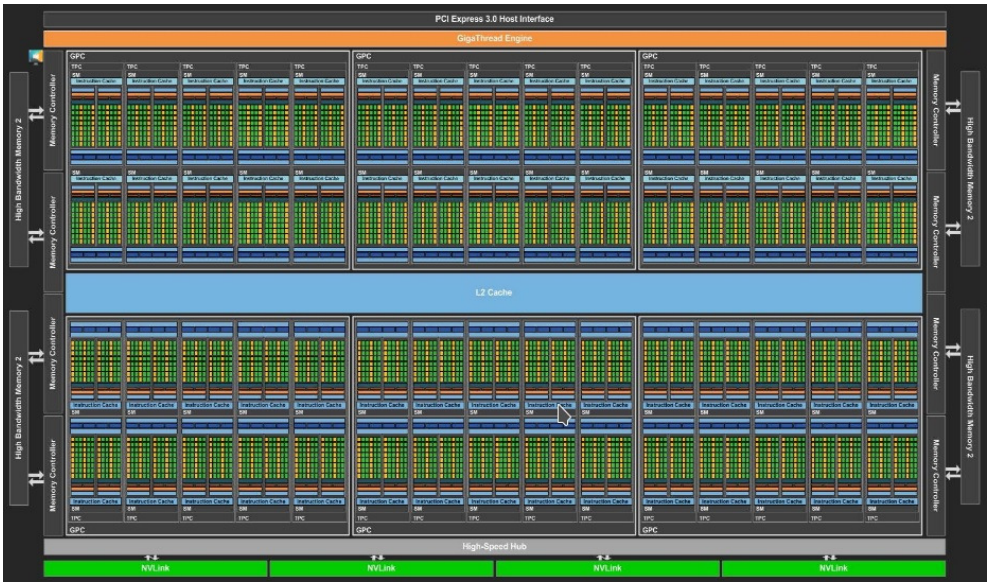


Figure 3.1: Overviwe of the NVIDIA Pascal architecture

The *Pascal* architecture has a hierarchical structure to manage its Multiprocessors which are called *Streaming Multiprocessors* (SM). The GPU has 6 Graphics Procesing Clusters (GPC), each one with 5 Texture Processing Clusters (TPCs) that contain 2 SMs. Figure 3.1 shows an overview of the Pascal GP100 GPU. Each SM has 64 single precision CUDA cores (32 when working with double precision) partitioned into two processing

blocks, each one with its own warp scheduler. A warp is a scheduling unit created by NVIDIA to gather the threads, each warp executes the same SIMD instruction and has 32 threads. Figure 3.2 shows a schematic view of a Pascal SM.

Figure 3.2: Schematic view of a Pascal GP100 SM Unit

From the memory point of view each SM has 64KB of shared memory and a L1 cache that can serve as texture cache too. The L2 cache of the full GP100 is a unified 4096 KB cache. Finally, the main device memory has changed from DDR5 to HBM2 which gives a better memory performance with higher bandwidth. The Tesla P100 accelerator has 16GB of main memory.

3.2.4 Intel Many Integrated Core processors

The Intel Many Integrated Core (MIC) is an architecture especially developed for parallel computing and HPC environments by the processor manufacturer Intel. That architecture was presented in the Intel Xeon Phi coprocessors (codename Knights Corner) [56] and was the strategy of Intel to compete with GPUs for massive parallel computing.

The initial Xeon Phi coprocessors followed the offload processing paradigm from the main host CPU to the coprocessor. The developer could send data from the host memory to the coprocessor and then process it in the coprocessor taking advantage of the highly

parallel architecture. The Knights corner coprocessors have more than 60 cores with Intel Architecture in a ring interconnect. Each core has 4 hardware threads adding more than 240 threads in the coprocessor. Each core has a scalar unit and a SIMD vector unit 512 bit wide, which allows to process 8 double precision or 16 single precision floating point numbers in a single operation. The 512KB coherent L2 cache of each core unifies in a fully coherent L2 cache of 30MB across the 60 cores (more with higher number of cores versions).

From the programming point of view, the Xeon Phi processors share the same software tools with the Intel Xeon architecture, so it is easier to adapt parallel codes developed for Intel Xeon architectures. Most of the Intel development tools like profilers and code optimizers can be used over the Intel MIC architecture. But in order to exploit all the benefits of the architecture, the developer should work at a lower level using all the hardware capabilities that the MIC architecture offers like the SIMD instructions. However some HPC libraries like Intel MKL (see Sec 3.3.5) have been ported to the MIC architecture so the developer can take advantage of the architecture performance without needing a fine grain knowledge of it.

One of the disadvantages of this coprocessor was the overhead produced by the data transfer from the host memory to the coprocessors memory and by the computation offloading. Its peak performance was very limited due to this factor. Thus with the next iteration of the MIC architecture called Knigths Landing, Intel presented two form factors for its architecture: they kept the old coprocessor form factor and they presented a new host processor form factor. This host processor is cabable to boot an OS and has access to the full memory of the system, avoiding the offloading problems of its predecessor. The newest Intel Xeon Phi host processors are used as independent node processors in some high end clusters of the TOP500 list [57] like *Trinity* and *Cory*. The top 2 cluster in the TOP500 list uses the old Knigths Corner MIC architecture that was able to achieve very good performance in highly parallel environments despite the offload limitations.

3.2.5 Heterogeneous computing

As seen in the previous Sections each of the architectures has its own advantages and disadvantages. From this mixture of architectures with different capabilities a new problem arises: a system can have an heterogeneous composition made of different architectures. To make the most of the performance of that systems, the heterogeneous computation programming models try to optimize the performance of the system by maximizing the benefits of each architecture, while minimizing its inconveniences. For example, an heterogeneous cluster with multiprocessors of different speed in different nodes needs to adapt the computational load distribution to the speed of each node of the cluster. If the

load is distributed homogeneously all the nodes of the cluster will work at the speed of the slowest node. Hence, wasting hardware on the most powerful nodes.

In modern HPC systems, the different architectures used into each node (multiprocessors, GPUs, coprocessors...) turn the cluster into an heterogeneous cluster even if all the nodes are homogeneous. In a cluster with multiprocessor nodes with GPUs an application should distribute the work between the nodes, then each node should use a shared memory approach to distribute its work between their processors/cores and it should offload computations to GPU when the computations can take benefit of the SIMD architecture.

3.3 HPC software technologies

To make the most of each HPC architecture described in Section 3.2, specific software is needed. This software can range from specific parallel programming models/languages to HPC libraries or complementary HPC tools. In this section the most used technologies in HPC will be briefly outlined.

3.3.1 OpenMP

OpenMP is an Application Program Interface (API) designed for shared memory architectures. The API supports *C/C++* and *fortran* programming languages.

OpenMP is based on compiler directives and its own library functions. Being based on compiler directives has two main advantages: first, the same code can be sequential or parallel depending on flags at compilation time; second, it can be applied to an existing sequential code to parallelize it without changing most of the code. This makes OpenMP a very powerful and simple tool to parallelize legacy codes without too much effort. This trivial parallelization can not always be done due to data dependences between the threads.

On OpenMP, the runtime manages all the aspects regarding the thread creation and scheduling transparently for the developer, while some optional parameters are available to fine tuning the management of the threads. Thus, the developer can focus on the parallelization of its application and the problems related to shared memory parallel programming: thread synchronization, critical multithread data accesses, etc.

After OpenMP version 4, the *target* and *simd* construct were added, enabling OpenMP code to offload execution to accelerators like GPUs or the Intel MIC described above.

For a full detail of the OpenMP API check the latest specification, currently version 4.5 [58].

3.3.2 MPI

The Message Passing Interface (MPI) [59] is a standard API designed for distributed memory architectures. There are several implementations of the standard like OpenMPI [60] or MPICH [61], each implementation should implement all the functions present on the MPI standard, but the internal aspects of each library are implementation dependent.

While designed for distributed memory, an MPI application can be executed in a shared memory system. However, the application will get some disadvantages of the distributed memory architecture without getting the advantages of a shared memory architecture.

An MPI application consists of a group of different processes which work together to solve a problem. Each process is independent from the other and has its own memory address space, sharing the necessary information to complete the job through message passing. The application starts when each process calls the *MPI_Init* function and finishes when all processes reach the end of the program calling the *MPI_Finalize* function.

The basic functionality of the standard is given by the *MPI_Send* and *MPI_Recv* functions. Those functions define a one to one message between two processes. One process uses the *MPI_Send* function to send a message to another process, who should use the *MPI_Recv* function to receive that message.

The standard defines more complex communication patterns between groups of processes called collective communications. These communications share a message between a group of processes defined by a *communicator* in a more efficient way than doing one to one message passings. A communicator is a way of grouping processes inside the standard, the communicator MPI_COMM_WORLD is the default communicator that groups all the processes in an MPI application. Some examples of these collective communications are *MPI_Broadcast* to send a message from one process to several processes, *MPI_Scather* to split some data contained in one process and send each part to a different process or *MPI_Gather* to collect data from different processes and mix it into one process.

All the functions named previously are blocking functions that enforce synchronous communications. However the standard includes equivalent functions for non-blocking communications that enable the application to communicate in an asynchronous manner. In than case, the developer should explicitly control the synchronization between the processes using the *MPI_Barrier* function. That function acts as a software wall stopping

the execution of all the processes of the communicator until all of them arrive to the barrier.

The MPI standard includes other general management functionalities like the topology definition or the datatype creation. For the entire standard definition check [62].

3.3.3 CUDA

The Computer Unified Device Architecture (CUDA) is a hardware-software platform presented in 2007 created to develop parallel applications over NVIDIA GPUs. The CUDA toolkit [63] contains the CUDA driver , the runtime library, a C/C++ compiler, GPU-Accelerated libraries and debugging and optimization tools. While there are some interfaces for *Fortran* and *Python*, CUDA devices are programmed with CUDA C. CUDA C is a C/C++ extension created to develop applications over CUDA capable GPUs. When the *nvcc* compiler compiles the CUDA C source files (.cu), the host code is compiled with the system compiler, then PTX code (an intermediate pseudo-assembly code) is generated for the CUDA *kernels* and finally the CUDA driver generates the machine code for the specific GPU on the system.

The *kernels* are functions designed to be executed by *N* threads in parallel by *N* CUDA threads in the GPU. Those kernels are the core of the CUDA C extension and are defined using the *__global__* declaration specifier together with the execution configuration enclosed between <<< ... >>>. That configuration has a particular syntax to define the number of threads that will execute the kernel and the thread hierarchy used. Each thread executing the kernel receives a unique thread ID within a block, contained in the built-in *threadIdx* variable.

The *threadIdx* variable is a 3-component vector, so threads can be grouped into one-dimensional, two-dimensional or three-dimensional blocks. All threads within a block will be executed by the same SM, and thus will share the resources of that SM. Then the blocks are organized into a one-dimensional, two-dimensional or three-dimensional grid of blocks. This hierarchy confers the developer the ability to map the threads to the problem at block level and to the hardware at grid level.

From the memory point of view, each thread has access to its own private memory, to a shared memory with all the threads in its thread block (only valid during the execution of that block) and to the global memory shared between all threads. There are also two read-only memories: the constant and the texture memories. The global, constant and texture memory spaces are persistent across kernel launches by the same application.

A basic execution of a CUDA *kernel* will involve the following steps:

1. Allocate memory on device (*cudaMalloc*).
2. Transfer data from host to device (*cudaMemcpy*)
3. Execute kernel ($\_\_global\_\_myKernel <<< N, M >>> ()$)
4. Transfer result from device to host (*cudaMemcpy*)

For more information about the CUDA programming model check the CUDA C programming guide [64].

3.3.4 MATLAB for HPC

MATLAB [65] is a mathematical software tool that offers a development environment together with its own programming language. It is a widely used tool used for development in various scientific fields. While MATLAB itself is not an HPC software and its language is focused in programmability instead of performance, it has very good performance when working with matrix operations. Furthermore, the MATLAB Parallel Computing Toolbox lets MATLAB developers create parallel applications within the MATLAB programming language syntax. This allows the developers to get more performance from HPC architectures (multicore processors, computer clusters and GPUs) without the need of learning new programming languages or technologies.

Another interesting feature is the MEX interface, which allows to call functions written in C/C++ or Fortran directly from MATLAB. This allows to reuse already programmed algorithms in those languages into larger MATLAB projects without the need of coding them again in MATLAB's programming language. Additionally, the developers can call more efficient code or efficiently developed libraries to improve the performance when MATLAB's performance is not enough. To use the MEX interface, an intermediate MEX file should be created containing a function callable from MATLAB that contains the C/C++/fortran function to call and the necessary steps to pre/post process the function arguments. Furthermore, CUDA code can be called from MATLAB using the MEX interface. For more information on the implementation of MEX files check [66].

3.3.5 HPC mathematical libraries

In the HPC community there are some very well known mathematical libraries optimized for different HPC architectures which can be used to improve the performance of mathematical algorithms. In this section some of them used during the development of this thesis will be introduced.

Basic Linear Algebra Subprograms (BLAS)

The Basic Linear Algebra Subprograms (BLAS) [67, 68] is a collection of computational routines designed to perform basic linear algebra operations like vector-matrix or matrix-matrix products. These routines originally designed in *Fortran*, were designed from a computational point of view to achieve the maximum performance of the available hardware. Nowadays, there is a C interface called CBLAS designed to make it easier to use the BLAS routines while developing in C. The BLAS routines are divided into 3 categories:

1. BLAS 1: Routines that perform vector-vector operations.
2. BLAS 2: Routines that perform vector-matrix operations.
3. BLAS 3: Routines that perform matrix-matrix operations.

Developers are encouraged to use higher level operations (BLAS 3) while designing their algorithms because they have a more efficient memory access pattern and carry out more flops per memory access, offering a better performance.

There are several implementations of the original routines adapted and optimized to the hardware characteristics of the system in where they are executing. These implementations achieve a great performance close to the peak performance of the hardware. Some of them are listed below.

Linear Algebra PACKage (LAPACK)

Linear Algebra PACKage (LAPACK) [69] is a set of routines for solving systems of linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. This routines are designed on top of the BLAS routines to improve the performance of existing packages on shared memory parallel processors. To get the most of the hardware performance, LAPACK is designed to use BLAS 3 operations in order to exploit the computational benefits of the operations in this level of BLAS. Designed originally in *Fortran*, there are *C* interfaces for the package. Those *C* interfaces (like LAPACKE from Intel and CLAPACK from *netlib*) were specific from each implementation of the routines, but nowadays the official *C* interface for LAPACK is LAPACKE (coming from a collaboration of the LAPACK team with the Intel Math Kernel Library Team).

The different LAPACK implementations which depend of a BLAS implementation installed on the system, they are commented below.

Implementations of BLAS and LAPACK

There are several implementations of BLAS and LAPACK packages, most times shipped together. First of all, there are the original implementations from the *Netlib* [70, 71]. These implementations are the core of all BLAS and LAPACK implementations but they are not tuned for every particular architecture or system. Most of the UNIX distributions include versions of BLAS and LAPACK libraries in its package managers.

The ATLAS (Automatically Tuned Linear Algebra Software) [72] is a software that automatically generates an optimized BLAS library for the architecture of the system in which is installed, with some LAPACK routines too. The software performs a series of tests during the installation process in order to optimize the library to the current system.

The GotoBLAS [73] was an open source implementation of the BLAS interface developed by *Kazushige Goto* with many manual low level optimizations for certain processors. Nowadays is a discontinued project, but an open source fork called OpenBLAS [74] continues its development. OpenBLAS includes the LAPACK interface along with threaded versions of the functions in the library.

The Intel Math Kernel Library [75] is a proprietary software library developed by Intel and optimized for Intel architecture processors it contains implementations of the BLAS and LAPACK interfaces with some other BLAS-like extensions that are not present in the BLAS interface. Additionaly, includes some other mathematical packages like vector functions, FFT or nonlinear optimization problem solvers. Most of the BLAS and some of the LAPACK routines are threaded inside MKL [76]. Furthermore, the ScaLAPACK and PBLAS interfaces for distributed memory systems are included in the library. There is a version of the Intel MKL for the Intel MIC architecture too.

There are some implementations of BLAS and LAPACK available for GPU architectures. For example, CULA [77] is a LAPACK implementation for CUDA capable GPUs developed by *EM Photonics* in partnership with NVIDIA and cuBLAS [78] is a BLAS implementation for CUDA capable GPUs developed by NVIDIA.

Finally the MAGMA project [79] develops a library similar to LAPACK for heterogeneous architectures (e.g. Multicore + GPU systems).

Chapter 4

Existing NNMFPack HPC library

4.1 Library overview

The NNMFPack library first presented in [36] and used in [80] was originally developed by researchers from the *Information Retrieval and Parallel Computing group* (IRPCG) from the *University of Oviedo* and from the *Interdisciplinary Computation and Communication Group* (INCO2) from the *Universitat Politecnica de Valencia*. The goal of this library was to provide a HPC framework to compute the NMF, in order to help researchers of different fields of science and engineering who use the NMF to take advantage of the newest technologies and computational architectures. Having a reliable and efficient HPC framework can free the researchers from the work of implementing already known methods to compute the NMF. Allowing them to focus in their applications while they make the most of their computational resources. The library source can be found in [81].

4.2 Technologies under the hood

At the beginning of the development of this thesis the library was at its version 2.1 which was released on April 2015 and in this section we are going to describe the supported technologies and architectures of that version. The library routines are developed in C programming language and the library has been designed for Linux-compatible operating systems. The library is not an heterogeneous-coprocessor library, the library supports several architectures but its not designed to work with different coprocessor architectures at the same time.

4.2.1 Supported architectures

The NNMFPack library v2.1 supports three architectures (see Section 3.2):

1. CPU: The library supports multicore and single-core CPUs. The multicore parallelization is performed through OpenMP and Intel development technologies. Originally compatible with x86_64 CPUs, the support of ARM CPU was added in version 2.2.

2. Graphic Processing Unites (GPU): The current version of the library only supports CUDA compatible GPUs.

3. Intel MIC: The library supports the Intel Many Core Integrated (MIC) architecture, present on the Intel Xeon Phi coprocessors.

4.2.2 Supported compilers

The library supports *gcc* and Intel C compiler (*icc*) which needs to be configured on installation. The Intel development technologies for parallelization are used only with the *icc* compiler. If the library is configured to use accelerators, it will need *nvcc* to compile for GPU architecture and *icc* for the Intel MIC architecture.

4.2.3 Required mathematical libraries

NNMFPack needs an implementation of BLAS and LAPACK linear algebra libraries installed in the system in order to work. By default, it uses the system implementation of the libraries, which may come from the package manager itself or from the ATLAS software (recommended). When using GPU, the library needs cuBLAS and MAGMA libraries and when using Intel MIC, the Intel Math Kernel library is required too. NNMFPack supports MKL for CPU computations too. The user should configure which implementation to use during the installation process.

4.2.4 Other technologies

Another interesting point from the usability part of the library is the integration with MATLAB/Octave through MEX interfaces. With these interfaces, developers that are used to work with MATLAB/Octave can benefit from the performance of the library without the need of learning to program in C language.

4.3 Implemented algorithms

The main algorithm currently available in the library is the multiplicative β-divergence algorithm (see Section 2.4.3). The main function prototype is as follows:

```
int <p>bdiv_<ARCH>(const int m, const int n, const int k,
    const double *A, double *W, double *H,
    const double beta, const int uType,
    const int nIter);
```

where $< p >$ represents the floating point precision (*s* for single precision or *d* for double precision) and $< ARCH >$ corresponds to the target architecture (cpu, gpu or mic).

The input parameters of the function are detailed bellow:

- m: Rows of matrix A.
- n: Columns of matrix A.
- k: Number of columns of matrix W and number of rows of matrix H.
- A: Contains the original matrix to factorize of size $m \times n$.
- W: Contains the $W0$ initialization matrix of size $m \times k$.
- H: Contains the $H0$ initialization matrix of size $k \times n$.
- $beta$: States which β-divergence to use.
- $uType$: Type of update. There are three update types available: *UpdateAll* which performs a complete NMF, *UpdateW* which only updates matrix W and *UpdateH* which only updates matrix H.
- $nIter$: Number of iterations to perform.

On output, matrices W and H contain the solution of the decomposition.

The mex interface for MATLAB/Octave follows a similar convention:

```
[W,H]=mex_bdiv<ARCH>(A, W0, H0, beta, uType, nIter);
```

There are only two differences on the parameters. The first one is that the matrix sizes are not necessary in MATLAB/Octave. The other one is that matrices $W0 \in \mathbb{R}^{m \times k}$ and $H0 \in \mathbb{R}^{k \times n}$ are the initialization matrices while $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{k \times n}$ contain the solution of the decomposition on exit.

As explained in Section 2.4.3 some β parameter values correspond to different known divergences, for example, the Frobenius norm with $\beta = 2$ and the Kullback-Leibler divergence with $\beta = 1$. For these cases, which are the most used, using a dedicated algorithm is more efficient than using the general β-divergence algorithm. That is why the library contains an efficient version of the mlsa algorithm (see Section 2.4.1) for the $\beta = 2$ case and an specific algorithm simplified from the general version for the $\beta = 1$ case. Internally, the library chooses which algorithm to use based on the β parameter selected by the user to achieve the best performance.

Chapter 5

Updating the solution of Nonnegative least squares problems

5.1 Introduction

While there are efficient algorithms for solving the NNLS problem introduced in Section 2.1, little attention has been paid to the problem of updating solutions of NNLS problems. There are well known solutions in the case of unconstrained Linear Least Squares problem (LSP), as can be seen, for example, in [2], where the updating of the QR decomposition is studied as a tool which allows the LSP to be efficiently solved for special cases where the coefficient matrix changes slightly.

The main objective of the work presented in this chapter, is to obtain efficient algorithms to solve the problem of updating a solution of the NNLS problem. In other words, given a matrix $A \in \Re^{m \times n}$ and a vector $b \in \Re^m$, and assuming we know the solution of the NNLS problem for this case ($w \in \Re^n_+, w = NNLS(A,b)$), find the solution of the NNLS problem for another matrix $\hat{A}$ (and sometimes for a new vector $\hat{b}$), $x = NNLS(\hat{A},\hat{b})$, which is obtained from A in a simple way, for example, by adding a row or column, or deleting a row or column. Obviously, it is always possible to compute the NNLS solution $x = NNLS(\hat{A},\hat{b})$ by tackling the problem from scratch, regardless of the information provided

by the NNLS solution w of the problem (A,b). However, the use of this information can decrease the computational cost of the solution of the new problem.

In the case of the updating of the QR decomposition, which can be used to solve the unconstrained least squares problem, the use of techniques that leverage the information about the initial problem allows the computational cost to be decreased by an order of magnitude. The cost decreases from $O(n^3)$ if the QR decomposition of $\hat{A}$ is done from scratch, to $O(n^2)$ if the information on the QR decomposition of A is used (e.g. see [2] Chap.6).

The abilities of the BPP algorithm, presented in section 2.3.1, to exchange multiple indices between the two index sets (F and G) at each iteration and to initialize the index sets to any index partition makes BPP algorithm suitable to be used as basic algorithm in updating problems. For convenience of notation, a function called IterateNNLS has been developed implementing the BPP algorithm. The function prototipe is as follows:

$$w = IterateNNLS(\hat{A}, b, x, y, F, G)$$

the user must provide an initial vector $x \in \Re^n$, with $x_i \geqslant 0, i = 1,\ldots,n$, and initial index sets F and G such that $F \cup G = \{1,2,\ldots n\}, F \cap G = \emptyset$, and $x_F = \underset{z \in \Re^{|F|}}{\operatorname{argmin}} \|A_F z - b\|_2, x_G = 0$. Then y is computed as $y_G = A_G^T(A_F x_F - b), y_F = 0$. Even though x and y can be computed directly, it is more convenient to use them as input arguments.

It is easy to verify experimentally that the computational cost of principal pivoting algorithms (and BPP in particular) depends directly on the number of exchanges of indices between the sets F and G. (This will become clear when experimental results are presented).

The idea behind the update algorithms is relatively simple. If the initial sets F and G are the optimal ones, the solution is obtained immediately. If the sets F and G are not optimal but are close to the optimal ones, then the optimal sets will be obtained in a few iterations. If the initial sets are very different from the optimal ones, then the algorithm must carry out a large number of index exchanges, thereby increasing the computational cost.

The fundamental idea in all cases is to use the vector w as an initial guess for the solution of the $NNLS(\hat{A}, b)$ problem in order to generate the initial sets F and G. Since the new problem is close to the original one, the optimal sets for the new problem are expected to be similar to the optimal sets of the original problem. Therefore, the number of exchanges needed should be small. Please note that the improvement of performance of the updating algorithms proposed, compared with solving the problems from scratch, can-

not be theoretically guaranteed. However, it will be empirically shown that the updating algorithms are indeed faster in most cases.

In sections 5.2, 5.3 and 5.4 we define more precisely different possible updates and solutions for column modifications, row modifications and low-rank modifications.

5.2 Column modifications of NNLS.

5.2.1 Appending a column.

The simplest case is to append a new column to the coefficient matrix; more precisely, the problem is stated as:

Case 1. *Given $A \in \Re^{m\times n}$, $b \in \Re^m$, $v \in \Re^m$, and $w \in \Re^n_+$, such that $w = NNLS(A,b)$, find a vector $x \in \Re_+^{(n+1)}$ such that $x = NNLS(\hat{A},b)$, with $\hat{A} = [A\ v]$.*

Since the vector w is the solution to the $NNLS(A,b)$ problem, it must verify the KKT conditions (2.4) for this case, i.e., $w \geqslant 0, y = A^T(Aw-b) \geqslant 0, w_i y_i = 0, \forall i$. To solve the $NNLS(\hat{A},b)$ problem, let us define F as the set of those indices i such that $w_i > 0$ and take $x = [w;0]$ as initial solution.

Note that

$$\hat{y} = \hat{A}^T(\hat{A}x-b) = \begin{bmatrix} A^T \\ v^T \end{bmatrix} (\begin{bmatrix} A & v \end{bmatrix} \begin{bmatrix} w \\ 0 \end{bmatrix} - b) = \begin{bmatrix} A^T \\ v^T \end{bmatrix} (Aw-b) = \begin{bmatrix} y \\ v^T(Aw-b) \end{bmatrix} \tag{5.1}$$

Besides,

$$\begin{bmatrix} w \\ 0 \end{bmatrix} = \underset{z\in\Re^{|F|}}{\operatorname{argmin}} \|[A\ v]_F z - b\|_2 = \underset{z\in\Re^{|F|}}{\operatorname{argmin}} \|A_F z - b\|_2$$

Since w and y meet the conditions $w \geqslant 0, y = A^T(Aw-b) \geqslant 0, w_i y_i = 0$, for all i, the pair $(x,\hat{y})$ is a solution of $NNLS(\hat{A},b)$ if $v^T(Aw-b)$ is greater than or equal to zero ($x \geqslant 0, \hat{y} \geqslant 0, x_i\hat{y}_i = 0$ for all i).

However, if $v^T(Aw-b) < 0$, the pair $(x,\hat{y})$ is not a solution for $NNLS(\hat{A},b)$ and new iterations should be carried out to find the solution for $NNLS(\hat{A},b)$. These ideas are summarized in the following algorithm:

Algorithm 12 Appending a column

1: **Input**: A, b, v, w such that $w = NNLS(A, b)$
2: **Output**: $x = NNLS([A\ v], b)$
3: $x = [w; 0]$;
4: $\hat{b} = Aw - b$;
5: $\alpha = v^T \hat{b}$;
6: **if** $\alpha < 0$ **then**
7: $\quad F = \{i : x_i > 0\}; G = \{1, 2, \cdots, n+1\} - F$; $\hat{y} = [A^T \hat{b}; \alpha]$; $\hat{A} = [A\ v]$;
8: $\quad x = IterateNNLS(\hat{A}, b, x, \hat{y}, F, G)$;
9: **end if**

If the new column must be added in an intermediate position between 1 and n, and not as column $n+1$, it suffices to calculate the permutation P that reorders the columns of the matrix $[A\ v]$ as desired:

$$[A_1, A_2, .., A_j, v, A_{j+1}, ..A_n] = [A\ v]P \tag{5.2}$$

and solve the problem

$$\underset{x \geqslant 0}{\operatorname{argmin}} \|[A_1, A_2, .., A_j, v, A_{j+1}, ..A_n]P^T Px - b\| = \underset{\hat{x} \geqslant 0}{\operatorname{argmin}} \|[A\ v]\hat{x} - b\|, \tag{5.3}$$

to obtain the solution of the initial problem as $x = P^T \hat{x}$.

5.2.2 Appending a block of columns.

The ideas of the previous section can be easily extended to the case when a column block must be added.

Case 2. *Given $A \in \Re^{m \times n}$, $b \in \Re^m$, $V \in \Re^{m \times r}$, and $w \in \Re^n_+$, such that $w = NNLS(A, b)$, find a vector $x \in \Re^{(n+r)}_+$ such that $x = NNLS(\hat{A}, b)$, with $\hat{A} = [A\ V]$.*

As in the case where a single column must be added, the vector w is the solution to the $NNLS(A, b)$ problem. Therefore the KKT conditions must be verified for this case, $w \geqslant 0, y = A^T(Aw - b) \geqslant 0, w_i y_i = 0$, for all i.

To solve the $NNLS(\hat{A}, b)$ problem, let us define now $F = \{i : w_i > 0\}$ and take $x = \left[w; 0; \overset{r}{\cdots}; 0\right] \in \Re^{n+r}$ as initial solution. Note that

$$\hat{y} = \hat{A}^T(\hat{A}x - b) = [A\ V]^T([A\ V][w;0;\overset{r}{\cdots};0] - b) = \begin{bmatrix} A^T(Aw-b) \\ V^T(Aw-b) \end{bmatrix} = \begin{bmatrix} y \\ V^T(Aw-b) \end{bmatrix}. \tag{5.4}$$

Besides,

$$\begin{bmatrix} w \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \underset{z\in\Re^{|F|}}{\operatorname{argmin}} \|[A\ V]_F z - b\|_2 = \underset{z\in\Re^{|F|}}{\operatorname{argmin}} \|A_F z - b\|_2$$

Since w and y meet the conditions $w \geqslant 0, y = A^T(Aw-b) \geqslant 0, w_i y_i = 0$, for all i, the pair $(x,\hat{y})$ is a solution to $NNLS(\hat{A},b)$ if $V^T(Aw-b)$ is greater than or equal to zero ($x \geqslant 0, \hat{y} \geqslant 0, x_i\hat{y}_i = 0$, for all i).

However, if a component of vector $V^T(Aw-b)$ is smaller than 0, the pair $(x,\hat{y})$ is not a solution to $NNLS(\hat{A},b)$ and some iterations of the algorithm $IterateNNLS(\hat{A},b\ ,x,\hat{y},F,G)$ must be applied to find the solution. This gives the following algorithm:

Algorithm 13 Appending a block of columns

1: **Input**: A,b,V,w such that $w = NNLS(A,b)$
2: **Output**: $x = NNLS([A\ V],b)$
3: $x = \left[w;0;\overset{r}{\cdots};0\right]$;
4: $\hat{b} = Aw - b$;
5: $aux = V^T\hat{b}$;
6: **if** $aux_i < 0$ for some i, $1 \leqslant i \leqslant r$ **then**
7: $\quad F = \{i : x_i > 0\}$; $G = \{1,2,\cdots,n+r\} - F$; $\hat{y} = [A^T\hat{b}; aux]$; $\hat{A} = [A\ V]$;
8: $\quad x = IterateNNLS(\hat{A},b,x,\hat{y},F,G)$;
9: **end if**

5.2.3 Deleting a column.

Case 3. *Given $A \in \Re^{m\times n}$, $b \in \Re^m$, and $w \in \Re^n_+$, such that $w = NNLS(A,b)$, find a vector $x \in \Re_+^{(n-1)}$ such that $x = NNLS(\hat{A},b)$, with $\hat{A} = A(:,1:n-1)$.*

Let us construct the vectors $x = w(1:n-1)$ and $\hat{y} = \hat{A}^T(\hat{A}x - b)$. Note that if $w_n = 0$, then $\hat{y} = y(1:n-1)$ and the set F corresponding to w would be the same than the set F corresponding to x ($x_F = w_F$), thus the pair $(x, \hat{y})$ satisfies the KKT conditions and would be the solution of the problem.

Otherwise, it is sufficient to take $F = \{i : x_i > 0\}$, $G = \{1,2,\cdots,n-1\} - F$, $x_F = \underset{z \in \Re^{|F|}}{\operatorname{argmin}} \|\hat{A}_F z - b\|_2$, $\hat{y}_G = \hat{A}_G^T(\hat{A}_F x_F - b)$, $x_G = 0, y_F = 0$, iterating afterwards if the KKT conditions are not verified for the pair x, y. The algorithm is summarized below:

Algorithm 14 Deleting a column

1: **Input**: A, b, w such that $w = NNLS(A, b)$
2: **Output**: $x = NNLS(A(:, 1:n-1), b)$
3: $x = w(1:n-1)$;
4: **if** $w_n \neq 0$ **then**
5: $\quad \hat{A} = A(:, 1:n-1)$;
6: $\quad F = \{i : x_i > 0\}$; $G = \{1,2,\cdots,n-1\} - F$;
7: $\quad x_F = \underset{z \in \Re^{|F|}}{\operatorname{argmin}} \|\hat{A}_F z - b\|_2$; $x_G = 0$; $\hat{y}_G = \hat{A}_G^T(\hat{A}_F x_F - b)$; $\hat{y}_F = 0$;
8: $\quad$ **if** $\hat{y}_i < 0$, for some $i \in G$ **or** $x_i < 0$, for some $i \in F$ **then**
9: $\quad\quad x = IterateNNLS(\hat{A}, b, x, \hat{y}, F, G)$;
10: $\quad$ **end if**
11: **end if**

If the column to be deleted is in an intermediate position j , $1 \leq j < n$ (i.e., not the last column) it is enough to calculate the permutation P which reorders the columns of the matrix $[A_1, A_2, ..., A_n]$ into the desired order:

$$\hat{A} = [A_1, A_2, .., A_{j-1}, A_n, A_{j+1}, ..A_j] = AP \tag{5.5}$$

and solve the problem

$$\underset{x \geqslant 0}{\operatorname{argmin}} \|APP^T x - b\| = \underset{\hat{x} \geqslant 0}{\operatorname{argmin}} \|\hat{A}\hat{x} - b\| \tag{5.6}$$

to obtain the solution of the original problem as $x = P\hat{x}$.

5.2.4 Deleting a block of columns.

Case 4. *Given $A \in \Re^{m\times n}$, $b \in \Re^m$, and $w \in \Re^n_+$, such that $w = NNLS(A,b)$, find a vector $x \in \Re_+^{(n-r)\times 1}$ such that $x = NNLS(\hat{A},b)$, with $\hat{A} = A(:,1:n-r)$.*

In this case, it is enough to take $F = \{1 <= i < n-r : x_i > 0\}$, $G = \{1,2,\cdots,n-r\} - F$, $x_F = \underset{z\in\Re^{|F|}}{\operatorname{argmin}} \|\hat{A}_F z - b\|_2$; $x_G = 0$; $\hat{y}_G = \hat{A}_G^T(\hat{A}_F x_F - b); \hat{y}_F = 0$, iterating afterwards if the KKT conditions for the x chosen and for the complementary vector $y = \hat{A}^T(\hat{A}x - b)$ are not verified. Note that if $w(n-r+1:n) = 0$, the iteration stage can be avoided. Therefore, the following algorithm is obtained:

Algorithm 15 Deleting a block of columns

Input: A, b, w such that $w = NNLS(A,b)$
Output: $x = NNLS(A(:,1:n-r),b)$
$x = w(1:n-r)$;
if $w_j \neq 0$ for some $j \in [n-r+1, n+r+2, \cdots, n]$ **then**
    $\hat{A} = A(:,1:n-r)$;
    $F = \{i : x_i > 0\}$; $G = \{1,2,\cdots,n-r\} - F$;
    $x_F = \underset{z\in\Re^{|F|}}{\operatorname{argmin}} \|\hat{A}_F z - b\|_2$; $x_G = 0$; $\hat{y}_G = \hat{A}_G^T(\hat{A}_F x_F - b), \hat{y}_F = 0$;
    if $\hat{y}_i < 0$, for some $i \in G$ **or** $x_i < 0$, for some $i \in F$ **then**
        $x = IterateNNLS(\hat{A}, b, x, \hat{y}, F, G)$;
    end if
end if

5.3 Row modifications of NNLS.

5.3.1 Appending a row.

Case 5. *Given $A \in \Re^{m\times n}$, $b \in \Re^m$, $v \in \Re^n$, $t \in \Re$ and $w \in \Re^n_+$, such that $w = NNLS(A,b)$, find a vector $x \in \Re^n_+$ such that $x = NNLS(\hat{A},\hat{b})$, with $\hat{A} = [A;\, v^T]$ and $\hat{b} = [b;\, t]$.*

Note that the pair (x,y), with $y = A^T(Ax - b)$ and $x = w$, fulfils the KKT conditions. Constructing the new complementary vector $\hat{y} = \hat{A}^T(\hat{A}x - \hat{b})$:

$$\begin{aligned}\hat{y} &= \begin{bmatrix} A^T & v \end{bmatrix} \left(\begin{bmatrix} A \\ v^T \end{bmatrix} x - \begin{bmatrix} b \\ t \end{bmatrix} \right) \\ &= A^T(Ax-b) + (v^T x - t)v = y + z,\end{aligned} \tag{5.7}$$

with $z = (v^T x - t)v$.

If the new pair $(x, \hat{y})$ verifies the KKT conditions, then x is the solution of the $NNLS(\hat{A}, \hat{b})$ problem. Given the sets $F = \{i : x_i > 0\}$ and $G = \{1, \ldots, n\} - F$, let us construct the components of $\hat{y}$ in G: $\hat{y}_G = A_G^T(A_F x_F - b) + v_G(v_F x_F - B) = y_G + z_G$. If $y_i + z_i < 0$ for some $i \in G$, a new stage of iteration must be carried out with initial values x, $F = \{i : x_i > 0\}$; $G = \{1, \cdots, n\} - F$; $\hat{y} = y_G + z_G$. The following algorithm summarizes these ideas.

Algorithm 16 Appending a row

1: **Input**: A, b, w, v, t such that $w = NNLS(A, b)$
2: **Output**: $x = NNLS([A; v^T], [b; t])$
3: $x = w$; $z = (v^T x - t)v$; $y = A^T(Ax - b)$;
4: **if** $z_i + y_i < 0$, for some i OR $x_i z_i \neq 0$ for some i **then**
5: $\quad \hat{A} = [A;\ v^T]$; $\hat{b} = [b; t]$;
6: $\quad \hat{y} = y + z$; $F = \{i : x_i > 0\}$; $G = \{1, \cdots, n\} - F$;
7: $\quad x = IterateNNLS(\hat{A}, \hat{b}, x, \hat{y}, F, G)$;
8: **end if**

Note that this algorithm also solves the problem $x = NNLS(\hat{A}, \hat{b})$, with $\hat{A} = [A(1 : j-1, :);$ $v^T; A(j : m, :)]$ and $\hat{b} = [b(1 : j-1);\ t; b(j : m)]$. This is the case in which the row must be appended in an intermediate position and not at the bottom of the matrix. Indeed, if P is the permutation matrix such that

$$\begin{aligned} P[A(1 : j-1, :); v^T; A(j : m, :)] &= [A; v^T]; \\ P[b(1 : j-1); t; b(j : m)] &= [b; t], \end{aligned}$$

then the following holds:

$$\min_{x\geqslant 0} \|[A(1:j-1,:);v^T;A(j:m,:)]x-[b(1:j-1);t;b(j:m)]\| = \tag{5.8}$$
$$\min_{x\geqslant 0} \|P[A(1:j-1,:);v^T;A(j:m,:)]x-P[b(1:j-1);t;b(j:m)]\| =$$
$$\min_{x\geqslant 0} \|[A;v^T]x-[b;t]\|.$$

5.3.2 *Appending a block of rows.*

Applying the ideas of the previous section to the case in which a row block must be added.

Case 6. *Given $A \in \Re^{m\times n}$, $b \in \Re^m$, $V \in \Re^{n\times r}$, $t \in \Re^{r\times 1}$, and $w \in \Re_+^{n\times 1}$, such that $w = NNLS(A,b)$, find a vector $x \in \Re_+^n$ such that $x = NNLS(\hat{A},\hat{b})$, with $\hat{A} = [A;\ V^T]$ and $\hat{b} = [b;\ t]$.*

For this case, a vector z, which is similar to the previous one, must be calculated in the form $z = V(V^T x - t)$.

If the new pair $(x,\hat{y})$ verifies the KKT conditions, then x is the solution of the $NNLS(\hat{A},\hat{b})$ problem. Otherwise, a new stage of iteration is needed to find the solution.

Algorithm 17 Appending a block of rows

1: **Input**: A,b,w,V,t such that $w = NNLS(A,b)$
2: **Output**: $x = NNLS([A;V^T],[b;t])$
3: $x = w$; $z = V(V^T x - t)$;
4: $y = A^T(Ax-b)$;
5: **if** $z+y_i < 0$, for some i OR $x_i z_i \neq 0$ for some i **then**
6: $\quad \hat{A} = [A;\ V^T]$; $\hat{b} = [b;\ t]$;
7: $\quad \hat{y} = y+z$; $F = \{i : x_i > 0\}$; $G = \{1,\cdots,n\} - F$;
8: $\quad x = IterateNNLS(\hat{A},\hat{b},x,\hat{y},F,G)$;
9: **end if**

5.3.3 Deleting a row

Case 7. *Given $A \in \Re^{m\times n}$, $b \in \Re^m$, and $w \in \Re^n_+$, such that $w = NNLS(A,b)$, find a vector $x \in \Re^n_+$ such that $x = NNLS(\hat{A},\hat{b})$, with $\hat{A} = A(1:m-1,:)$ and $\hat{b} = b(1:m-1)$.*

Let us choose $x = w$ and observe that

$$y = A^T(Ax-b) = \hat{y} + A(m,:)^T(A(m,:)x - b(m)) \tag{5.9}$$

with $\hat{y} = \hat{A}^T(\hat{A}x - \hat{b})$. Hence, $\hat{y} = y - A(m,:)^T(A(m,:)x - b(m))$.Therefore, if the pair $(x,\hat{y})$ fulfills the KKT conditions, $x = NNLS(\hat{A},\hat{b})$. Otherwise, a new iteration stage is needed to find the solution.

Algorithm 18 Deleting a row

1: **Input**: A, b, w such that $w = NNLS(A,b)$
2: **Output**: $x = NNLS(A(1:m-1,:), b(1:m-1)$
3: $x = w$; $y = A^T(Ax-b)$; $\alpha = (A(m,:)x - b(m))$;
4: $Aux = \alpha A(m,:)^T$;
5: $\hat{y} = y - Aux$;
6: **if** $\hat{y}_i < 0$, for some i OR $x_i\hat{y}_i \neq 0$ for some i **then**
7: $\quad \hat{A} = A(1:m-1,:)$; $\hat{b} = b(1:m-1)$;
8: $\quad F = \{i : x_i > 0\}$; $G = \{1,\cdots,n\} - F$;
9: $\quad x = IterateNNLS(\hat{A},\hat{b},x,\hat{y},F,G)$;
10: **end if**

Note that, as in Case 5, this algorithm also solves the problem when the row to be eliminated is in an intermediate position and not at the bottom of the matrix.

5.3.4 Deleting a row block.

Case 8. *Given $A \in \Re^{m\times n}$, $b \in \Re^m$, and $w \in \Re^n_+$, such that $w = NNLS(A,b)$, find a vector $x \in \Re^n_+$ such that $x = NNLS(\hat{A},\hat{b})$, with $\hat{A} = A(1:m-r,:)$ and $\hat{b} = b(1:m-r)$.*

As in the previous case, if we choose $x = w$, then $\hat{y}$ can be expressed as $\hat{y} = y - z$, with

$$y = A^T(Ax-b);$$
$$z = A(m-r+1:m,:)^T(A(m-r+1:m,:)x - b(m-r+1:m)).$$

If the pair $(x,\hat{y})$ fulfills the KKT conditions, $x = NNLS(\hat{A},\hat{b})$. Otherwise, a new iteration stage is needed to find the solution.

Algorithm 19 Deleting a row block

1: **Input**: A,b,w such that $w = NNLS(A,b)$
2: **Output**: $x = NNLS(A(1:m-r,:),b(1:m-r))$
3: $x = w$; $y = A^T(Ax-b)$;
4: $Aux = A(m-r+1:m,:)^T((A(m-r+1:m,:)x - b(m-r+1:m)))$;
5: $\hat{y} = y - Aux$;
6: **if** $\hat{y}_i < 0$, for some i OR $x_i\hat{y}_i \neq 0$ for some i **then**
7: $\quad \hat{A} = A(1:m-r,:)$;$\hat{b} = b(1:m-r)$; $F = \{i : x_i > 0\}$;$G = \{1,\cdots,n\} - F$;
8: $\quad x = IterateNNLS(\hat{A},\hat{b},x,\hat{y},F,G)$;
9: **end if**

5.4 Adding low-rank matrices to NNLS.

5.4.1 Adding a rank-one matrix.

Case 9. *Given $A \in \Re^{m\times n}$, $b \in \Re^m$, $v \in \Re^m$, $z \in \Re^n$, and $w \in \Re^n_+$, such that $w = NNLS(A,b)$, find a vector $x \in \Re^{(n+1)}_+$ such that $x = NNLS(\hat{A},b)$, with $\hat{A} = A + vz^T$.*

To deal with this case, let us choose $x = w$ and construct $\hat{y} = (A+vz^T)^T((A+vz^T)x - b)$. It holds $\hat{y} = A^T(Ax-b) + (z^Tx)A^Tv + (v^T(Ax-b) + (z^Tx)(v^Tv))z$.

Therefore $\hat{y} = y + aux$, with $aux = \gamma(A^T v) + \delta z$, and $\gamma = z^T x$, $\delta = v^T(Ax-b) + \gamma(v^T v)$. If the pair $(x, \hat{y})$ fulfills the KKT conditions, $x = NNLS(\hat{A}, b)$. Otherwise, a new iteration stage is needed to find the solution. The following algorithm summarizes these ideas:

Algorithm 20 Adding a rank-one matrix

1: **Input**: A, b, v, z, w such that $w = NNLS(A, b)$
2: **Output**: $x = NNLS(A + vz^T, b)$
3: $x = w$; $c = (Ax - b)$; $y = A^T c$;
4: $\gamma = z^T x$;
5: $\delta = v^T c + \gamma(v^T v)$;
6: $Aux = \gamma(A^T v) + \delta z$;
7: $\hat{y} = y + Aux$;
8: **if** $\hat{y}_i < 0$, for some i OR $x_i \hat{y}_i \neq 0$ for some i **then**
9: $\quad \hat{A} = A + vz^T$; $F = \{i : x_i > 0\}$;$G = \{1, \cdots, n\} - F$;
10: $\quad x = IterateNNLS(\hat{A}, b, x, \hat{y}, F, G)$;
11: **end if**

5.4.2 Adding a low-rank matrix (rank > 1).

Case 10. *Given $A \in \Re^{m \times n}$, $b \in \Re^m$, $V \in \Re^{m \times r}$, $Z \in \Re^{n \times r}$, and $w \in \Re^n_+$, such that $w = NNLS(A, b)$, find a vector $x \in \Re^n_+$ such that $x = NNLS(\hat{A}, b)$, with $\hat{A} = A + VZ^T$.*

As in the previous case, let us choose $x = w$ and construct $\hat{y} = (A + VZ^T)^T((A + VZ^T)x - b)$. It holds that $\hat{y} = y + aux$ with $aux = A^T p + Z(V^T(c + p))$ where $c = Ax - b$ and $p = V(Z^T x)$. If the pair $(x, \hat{y})$ fulfills the KKT conditions, $x = NNLS(\hat{A}, b)$. Otherwise, a new iteration stage is needed to find the solution. The following algorithm summarizes these ideas:

Algorithm 21 Adding a low-rank matrix

Input: A, b, V, Z, w such that $w = NNLS(A, b)$
Output: $x = NNLS(A + VZ^T, b)$
$x = w$;$c = (Ax - b)$; $y = A^T c$;
$p = V(Z^T x)$;
$Aux = A^T p + Z(V^T(c + p))$;
$\hat{y} = y + Aux$;
if $\hat{y}_i < 0$, for some i OR $x_i \hat{y}_i \neq 0$ for some i **then**
    $\hat{A} = A + VZ^T$; $F = \{i : x_i > 0\}$;$G = \{1, \cdots, n\} - F$;
    $x = IterateNNLS(\hat{A}, b, x, \hat{y}, F, G)$;
end if

5.5 Empirical analysis of the proposed algorithms

In this section, empirical evidence about the performance of the proposed algorithms is offered. The experiments have been divided into two groups: single algorithms and block algorithms. For each single updating algorithm, a row dimension m for matrix A with values $m = \{$ 2000, 4000, 6000, 8000, 10000$\}$ and a number of columns n with values $n = \{m/2, 3m/4, m\}$ have been chosen. On the other hand, for the block algorithms a problem size of $m = 10000$ and $n = 7500$ has been established and the number of columns (or rows) r removed (or added) has been incremented to see its effect in the performance.

For each case, the execution time and the number of exchanges of indices between sets F and G has been analysed, and a comparison between the solution of the NNLS problem carried out from scratch and the updating approaches has been performed. The block algorithms are extensions of their simpler versions that depend on the number r of columns (or rows) added (or deleted), that is why the results are shown for a determined problem size and relative to the parameter r.

The actual performance of the BPP algorithm for the NNLS problem depends on the number of exchanges needed. Because of this, the execution time is strongly correlated with the number of exchanges. A lower execution time is achieved with fewer exchanges (see Section 5.5.1). The proposed algorithms focus on reducing the number of exchanges by taking advantage of the initial problem solution in order to reduce the execution time.

Each experimental value shown in this section is the average of 10 executions with different random instances. The random matrices have been generated using the *rand* command of *MATLAB* [65], that is, matrices with random numbers between 0 and 1 with a uniform probability distribution. However, the results can change for any other type of matrix. Therefore, these results should only be interpreted as just a hint of the performance that the updating algorithms can obtain. All of the experiments were performed using MATLAB R2016b in the machine described in Appendix A.2.

To evaluate the performance of the proposed algorithms, an implementation of the Block Principal Pivoting algorithm developed by the author and its collaborators has been taken as the basic reference. Therefore any other NNLS algorithm can be used to obtain the basic reference solution.

The following routines have been developed and evaluated:

Case 1. Appending a column: `NNLS_UP_C1.m` (Algorithm 12)

Case 2. Appending a column block: `NNLS_UP_Cr.m` (Algorithm 13)

Case 3. Deleting a column: `NNLS_DOWN_C1.m` (Algorithm 14)

Case 4. Deleting a column block: `NNLS_DOWN_Cr.m` (Algorithm 15)

Case 5. Appending a row: `NNLS_UP_R1.m` (Algorithm 16)

Case 6. Appending a row block: `NNLS_UP_Rr.m` (Algorithm179)

Case 7. Deleting a row: `NNLS_DOWN_R1.m` (Algorithm 18)

Case 8. Deleting a row block: `NNLS_DOWN_Rr.m` (Algorithm 19)

Case 9. Adding a rank-one matrix: `NNLS_UP_ro.m` (Algorithm 20)

Case 10. Adding a low-rank matrix: `NNLS_UP_lr.m` (Algorithm 21)

Implementations in Matlab code of these routines can be found in the following GIT repository: https://gitlab.com/P.SanJuan/UpdatingNNLS

First, Case 1 will be discussed. The conclusions in this case are the same as for Cases 2-8; therefore Cases 2-8 are discussed together. Finally, we will comment on Cases 9 and 10, which present some special features.

5.5.1 Case 1

This case consists of updating the solution when one column is appended to the problem matrix. In the first experiment shown in Table 5.1, there is a comparison of execution times between our Matlab implementation of the BPP algorithm (**NNLS BPP**), and the implementation of Algorithm 12 **NNLS_UP_C1.m**. The first algorithm computed the solution of the extended problem from scratch, and Algorithm 12 computed the solution using the solution of the base problem as input data. As shown in Table 5.1 the implementation **NNLS_UP_C1.m** outperforms the **NNLS BPP** algorithm for all matrix sizes.

Table 5.1: Execution times (seconds) comparison when **one column** is **appended** to the problem matrix.

	NNLS BPP			**NNLS_UP_C1**		
Size	m/2	3m/4	m	m/2	3m/4	m
2000	0.5583	1.0566	1.6815	0.0211	0.0065	0.0053
4000	2.8797	5.0255	9.3407	0.0339	0.0079	0.0145
6000	8.7455	17.5797	35.6248	0.0606	0.0326	0.1749
8000	22.0365	45.6946	78.5715	0.1454	0.2273	0.2830
10000	36.5626	74.7961	135.8823	0.0266	0.0399	0.0412

As mentioned at the beginning of Section 5.5, the execution time is strongly correlated with the number of exchanges performed by the algorithm. In Table 5.2, a comparison between the number of exchanges executed in both algorithms shows that the updating algorithm performs many fewer exchanges than the BPP algorithm. This fact explains the large difference in execution times between those algorithms shown in Table 5.1.

Once the correlation between exchanges and execution times has been shown, the author does not consider necessary to show the number of exchanges in the rest of the experiments, because the execution times offer similar information. Some number of index exchanges are lower than one because the results have been averaged from the 10 executions performed for each problem size and not all the problem sizes performed exchanges to obtain the solution.

Table 5.2: Comparison of index exchanges when **one column** is **appended** to the problem matrix.

	NNLS BPP			**NNLS_UP_C1**		
Size	m/2	3m/4	m	m/2	3m/4	m
2000	2.000e+03	3.290e+03	5.297e+03	7.000e-01	0.000e+00	1.000e-01
4000	4.049e+03	6.654e+03	1.066e+04	2.000e-01	0.000e+00	0.000e+00
6000	6.116e+03	9.969e+03	1.603e+04	4.000e-01	0.000e+00	3.000e-01
8000	8.181e+03	1.328e+04	2.142e+04	2.000e-01	1.000e-01	4.000e-01
10000	1.023e+04	1.662e+04	2.671e+04	0.000e+00	0.000e+00	0.000e+00

5.5.2 Cases 2-8

In Cases 2-8, the results are similar; in all of them except Case 4, the updating algorithm is clearly faster than solving the modified problem from scratch. Case 4 does not show any improvement by using the updating algorithm. The results for these cases are presented as follows: Case 2 in Figure 5.1; Case 3 in Table 5.3; Case 4 in Figure 5.2; Case 5 in Table 5.4; Case 6 in Figure 5.3; Case 7 in Table 5.5; and Case 8 in Figure 5.4.

When compared with the performance of solving the problem from scratch, the experiments show that the performance of the block deletion algorithms (cases 4 and 8, Figures 5.2 and 5.4), may suffer if the size of the block (parameter r) becomes too large. This is quite logical because, if r is too large, the sets F and G of the original problem may be very different from the optimal ones. Furthermore, when many columns/rows are deleted, the cost of solving the problem from scratch decreases. Figures 5.2 and 5.4 show that the updating algorithms are faster than **NNLS BPP**, when the number of columns/rows to be deleted is small.

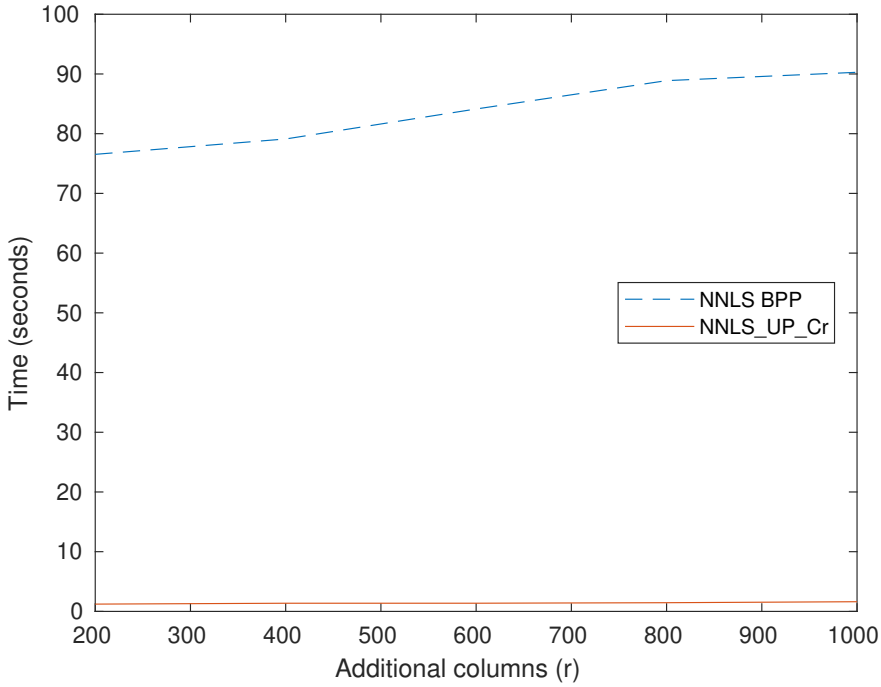


Figure 5.1: Evolution of execution times (seconds) from **appending** a **block of columns** of size r with a problem size of m=10000, n=7500.

Table 5.3: Comparison of execution times (seconds) when **one column** is **deleted** from the problem matrix.

	NNLS BPP			**NNLS_DOWN_C1.m**		
Size	m/2	3m/4	m	m/2	3m/4	m
2000	0.532	0.917	1.489	0.000	0.000	0.000
4000	2.168	4.101	7.910	0.000	0.000	0.105
6000	7.123	15.192	30.121	0.219	0.230	0.000
8000	18.464	36.837	69.731	0.000	0.000	0.000
10000	35.509	71.184	128.247	0.000	0.000	0.000

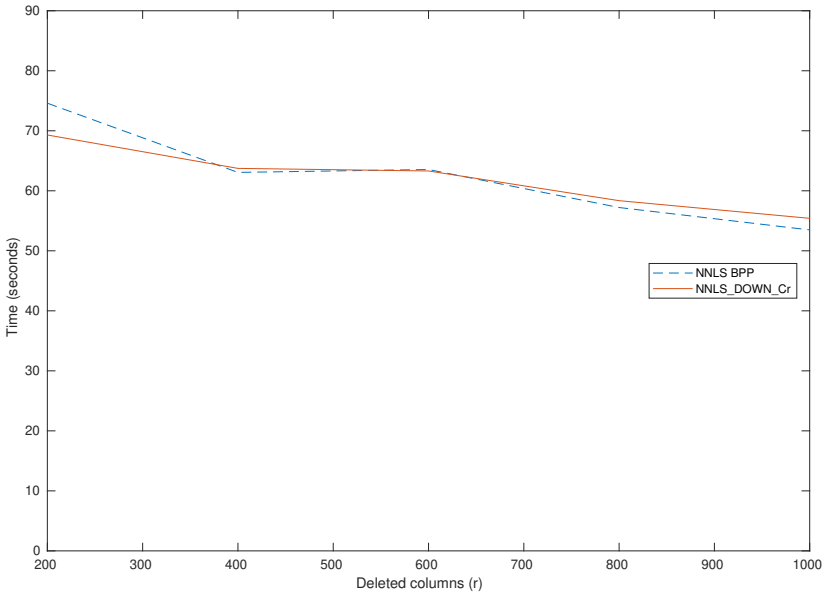


Figure 5.2: Evolution of execution times (seconds) when **deleting** a **block of columns** of size r with a problem size of m=10000, n=7500.

Table 5.4: Comparison of execution times (seconds) when **one row** is **appended** to the problem matrix.

	NNLS BPP			**NNLS_UP_R1.m**		
Size	m/2	3m/4	m	m/2	3m/4	m
2000	0.543	0.936	1.493	0.044	0.052	0.060
4000	2.271	4.322	8.276	0.236	0.291	0.415
6000	7.276	16.056	30.440	0.391	0.544	0.687
8000	18.007	37.685	70.795	0.633	0.758	1.132
10000	34.645	74.845	131.440	0.816	1.074	1.491

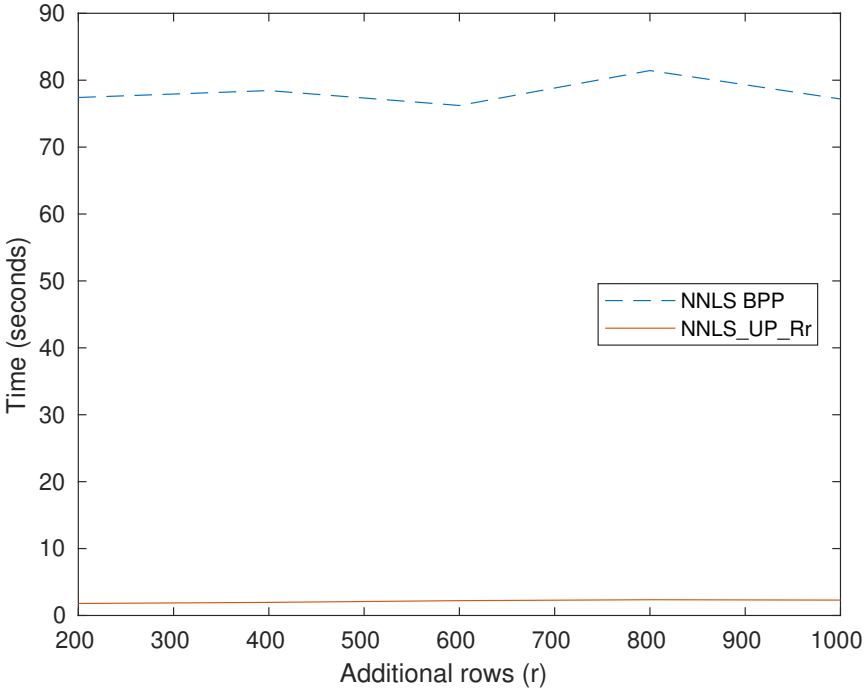


Figure 5.3: Evolution of execution times (seconds) when **appending** a **block of rows** of size r with a problem size of m=10000, n=7500.

Table 5.5: Execution time comparison(seconds) when **one row** is **deleted** from the problem matrix.

	NNLS BPP			**NNLS_DOWN_R1.m**		
Size	m/2	3m/4	m	m/2	3m/4	m
2000	0.551	0.944	1.487	0.047	0.048	0.058
4000	2.217	4.229	8.050	0.233	0.286	0.360
6000	7.243	15.962	30.649	0.377	0.558	0.654
8000	17.825	39.112	74.540	0.569	0.813	0.996
10000	34.498	72.285	134.046	0.805	1.293	1.757

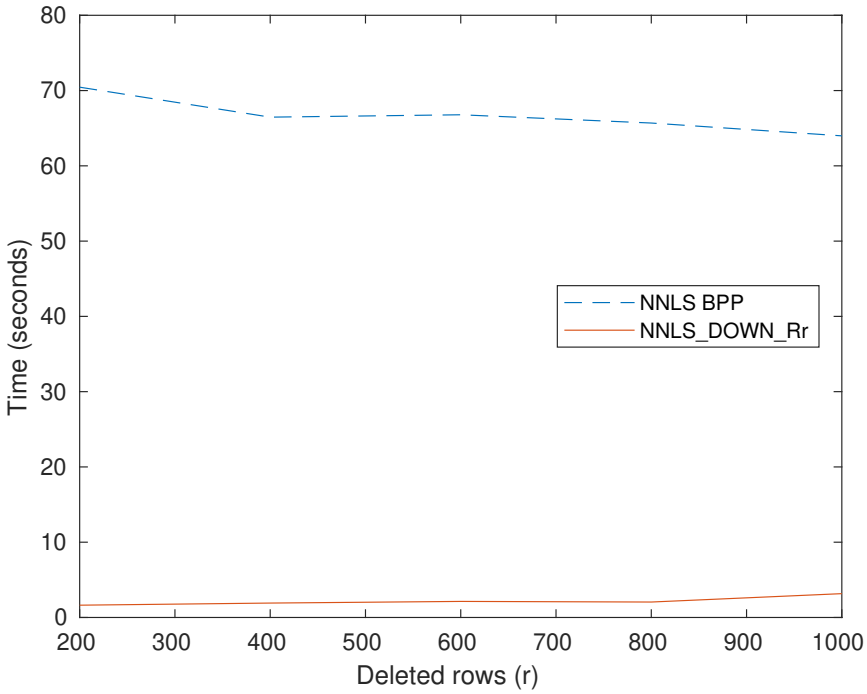


Figure 5.4: Evolution of execution times (seconds) when **deleting** a **block of rows** of size r with a problem size of m=10000, n=7500.

5.5.3 Cases 9-10

The cases of adding a rank-one/low-rank matrix to the original problem matrix have been included for reasons of completeness. However, these two algorithms seem to be more sensitive to the data in the target problem. When the matrices are generated using random numbers between 0 and 1, the performance of these algorithms is poor. Here some results where the random numbers of the matrix A and vector b are in the interval [0, 100] and the numbers of the added matrix in the interval [0,1] are presented. Table 5.6 shows the results for the rank-one case. In the low-rank case shown in Table 5.7, $m/50$ was used as the second dimension of vectors V and Z to make the rank of the added matrix proportional to the problem size.

Table 5.6: Comparison of execution times (seconds) when **a rank-one matrix** is **appended** to the problem matrix.

	NNLS BPP			**NNLS_UP_ro.m**		
Size	m/2	3m/4	m	m/2	3m/4	m
2000	0.540	0.910	1.230	0.031	0.043	0.052
4000	2.224	4.184	4.965	0.173	0.257	0.353
6000	7.134	15.873	15.596	0.311	0.415	0.551
8000	18.053	40.504	43.683	0.482	0.660	0.854
10000	35.818	74.228	75.785	0.712	1.209	1.238

Table 5.7: Comparison of execution times (seconds) when **a low-rank matrix** is **appended** to the problem matrix.

	NNLS BPP			**NNLS_UP_lr.m**		
Size	m/2	3m/4	m	m/2	3m/4	m
2000	0.540	0.961	1.243	0.053	0.061	0.076
4000	2.186	4.273	5.059	0.303	0.402	0.527
6000	7.086	15.667	15.809	0.529	0.723	0.886
8000	17.948	37.456	39.695	0.855	1.192	1.550
10000	36.500	74.066	83.306	1.054	1.628	2.238

5.6 Conclusion

The algorithms proposed in this paper allow for a faster solution of NNLS problems in a number of possible situations, where the problem to be solved is a slight modification of a NNLS problem that has already been solved. The efficiency of these algorithms has been compared for randomly generated matrices. Obviously, the relative efficiency may change in other problems, where the coefficient matrices have different properties. However, given that the idea behind these algorithms is simple and sound (i.e., if the modification of the problem is small, the sets of indices F and G of the original and of the modified problem should be similar), then the updating algorithms proposed should be reasonably efficient for any case.

Chapter 6

Updating the Non-negative matrix factorization

6.1 Introduction

There are situations where the matrix whose NMF 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 NMF factorization, and the data matrix receives new data items (sound recordings in a concrete time) very fast.

If the NMF 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 NMF: the updating of the NMF (recalculating the NMF when a new column or row is added to the data matrix) and the downdating of the NMF (recalculating the NMF when a column or row of the data matrix is discarded). This work is focused on dense matrices.

The idea of updating numerical factorizations is not new; indeed, the different updatings of the QR factorization [2] are routinely used in many fields, most often in Signal Processing. There are also other factorizations whose update has been proposed and studied. However, the study of the update of the NMF was proposed first in [82] and extended in [83]. This work will be presented in the current Chapter.

The most popular method to compute the NMF is the Multiplicative Algorithm of Lee & Seung [23, 84, 85, 86]. This algorithm has many advantages: its simplicity, it is easily parallelizable and there are many implementations readily available (there is an implementation in MATLAB [65]). 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 [35, 87]. One of the most successful seems to be the Hierarchical Alternating Least Squares method (HALS), proposed in [85, 86, 43], and more precisely, the fast HALS (fHALS) implementation proposed in the Algorithm 2 of [43]. This method obtains an important reduction of the factorization error, with a small computational cost. In [44] 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 work the fHALS method and the GCD method are used in the experiments. These methods can be combined to construct some updating algorithms.

6.2 Description of the NMF updating problem

The first problem to solve is the updating of the NMF. Given two matrices $W \in \mathbb{R}_+^{m\times k}$ and $H \in \mathbb{R}_+^{k\times n}$ which are a solution for the NMF problem $A \approx WH$, and a new column $b \in \mathbb{R}_+^{m\times 1}$, the updating needs 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 NMF problem:

$$V = [A\ b] \approx W_1 H_1 \tag{6.1}$$

Note that Matlab notation [88] will be used throughout this chapter to describe matrices.

This problem can be solved from scratch factorizing matrix V, but the proposed algorithms take advantage of the knowledge of W and H (such that $A \approx WH$) to solve problem (6.1) with a lower computational cost, and thus in less execution 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. The block problem can be rewritten as:

$$V = [A\ B] \approx W_1 H_1 \tag{6.2}$$

The second problem is the downdating of the NMF. For that, the following problem needs to be solved:

$$V_2 = A(:, r+1:n) \approx W_2 H_2 \tag{6.3}$$

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 to compute the NMF of V_2 from scratch.

Both problems are presented 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; from now on named "window" problem. This problem needs an update and a downdate, but both operations can be processed at the same time to save computational resources.

6.3 Algorithmic approach

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 to perform both stages. Some possibilities are analysed in Section 6.4.

As stated in the introduction, there are several algorithms to compute the NMF. In this section a modified version of the multiplicative Lee & Seung algorithm (from now on MLSA), the fast HALS algorithm and the GCD algorithm are compared by analysing 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 [23] and Algorithm 2 in [43] respectively. A similar counting of flops can be found in [87]. 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 [26, 27, 28, 89].

A similar cost in terms of superior order, can be found in [44] 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 a 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 [44]. 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 algorithm in [82]. fHALS algorithm has a faster convergence, for this reason, in this Chapter we use the fHALS algorithm as base algorithm. However, GCD method will be used in the preprocessing stage.

The convergence of the updating algorithms presented in this Chapter is a direct consequence of the convergence of the base algorithms, that is proved in [23] for MLSA, in [85] for fHALS algorithm and in [44] for GCD algorithm.

All algorithms presented contain 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 threaded LAPACK implementation. In addition, implementations of block algorithms are also presented whose runtime can be decreased by using parallel programming techniques(e.g. the OpenMP programming environment).

6.4 Proposed solutions

In the experiments, the initial factorization was 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.

At the postprocessing stage, 2 iterations of fHALS were 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 6.3 will be referred to as $cIterHals$, so the base factorizations will have a cost of $10 * cIterHals$.

6.4.1 Updating problem

Four algorithms to solve the updating problem are proposed in this Section:

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 22 Update rand fHALS

```
1: X = rand(k,r)
2: [W1,H1] = 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:

$$x = \max(0, \underset{x \in \mathbb{R}^k}{\operatorname{argmin}} \|Wx - b\|_2). \tag{6.4}$$

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 23 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 (6.2). 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 24 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 25 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.

6.4.2 Downdating problem

Following the same conventions of the updating problem, 2 iterations of the fHALS algorithm are computed for V_2 (6.3) 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.

6.4.3 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 NMF to the combination of both updating and downdating in one problem:

$$V_3 = [A(:, r+1:n)\ B] \approx W_3 H_3 \tag{6.5}$$

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 four algorithms are obtained:

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 is 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 6.1 and Table 6.2 the theoretical costs of the algorithms presented in this section are shown.

Table 6.1: 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 6.2: 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	$2 * cIterHalsDown$

6.5 Experimental analysis

In this section the results obtained from the experiments are shown and explained. Both the error obtained and the computation time needed to solve each problem are compared.

In the experiments the following error measure was used:

$$Err = \|WH - A\|_F / \sqrt{m * n} \tag{6.6}$$

Regarding computational time, each algorithm was executed 10 times registering the time needed to solve the problem. Then the times measured were averaged 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, two types of matrices were used. One type of matrices correspond to a real case of Automatic Music Transcription. The other matrices were generated using Algorithm 26, they are random matrices but incorporating some relationship between data to simulate a more realistic behaviour.

Algorithm 26 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 described in the Appendix A.2. The experiments were performed using MATLAB 2014b.

6.5.1 Updating experiments

The evaluation of algorithms related to NMF is never easy because the performance of the algorithms depends 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 26, 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).

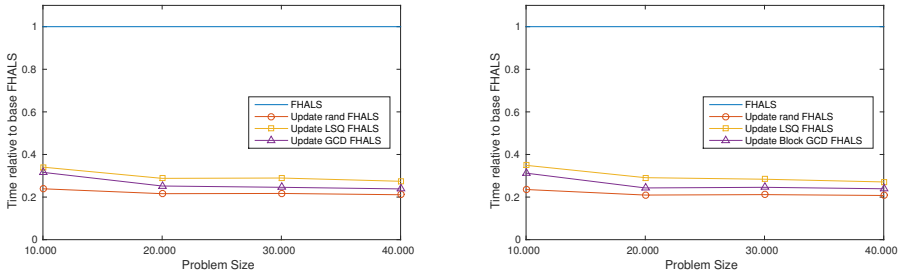
The execution times of the base fHALS algorithm for a single column update are shown in Table 6.3.

Table 6.3: 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 6.1 have been divided by execution times of the corresponding base fHALS experiment.

In Figure 6.1(a) we show the computation time needed to update the data matrix adding one column, with the proposed algorithms and with the NMF from scratch. The errors obtained in these experiments are shown in Table 6.4.



(a) Execution time, r=1

(b) Execution time, r=200

Figure 6.1: Execution times of updating experiments with synthetic matrices relative to fHALS base algorithm

Table 6.4: 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 NMF, and rand fHALS (Alg. 22) is the fastest. Its simpler initialization renders this algorithm faster, but also less accurate. The error of the proposed algorithms is slightly lower than the error of the complete NMF and all proposed algorithms have almost the same error.

The experiment was executed again adding 200 columns and using the Block LSQ fHALS (Alg. 24). Execution times and errors are shown in Figure 6.1(b) and Table 6.6. The execution times of the base fHALS algorithm for a 200 columns block update are shown in Table 6.5.

Table 6.5: Base execution times with r=200

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

Table 6.6: 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. 22) offers greater errors than the full factorization that grow with the problem size and the block size. The GCD fHALS algorithm (Alg. 25) keeps showing a higher performance than the LSQ fHALS algorithm (Alg. 23) 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. 23) and in the other case we updated the initial matrix in groups of 200 columns using the Block LSQ fHALS algorithm (Alg. 24).

Results in Table 6.7 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.

Table 6.7: 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

6.5.2 Downdating experiments

The first downdating experiment was to delete a column from the initial matrix and to recompute the NMF, either by applying a full NMF or by applying the downdating algorithm. The results are summarized in Table 6.8. Table 6.9 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 6.8: 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 6.9: 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.

6.5.3 Window experiments

To check the expected lower cost of the window algorithm, it was tested against an update followed by a downdate. The results are shown in Table 6.10. 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 6.10: 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

6.5.4 Real Application: Automatic Music Transcription

Automatic Music Transcription is an active area of research [90]. 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 NMF [89, 26, 27]. The main idea is that a NMF 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 [91, 28]. However, the computational cost of the NMF 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 NMF is considered not suitable for real time [91].

Nevertheless, the updating/downdating techniques described above may change this view. Without updating techniques, a NMF 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 matrix or spectrogram V and the H matrix would increase their size continuously, therefore the computational cost of each new NMF would increase as well.

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

A simple experiment was designed to illustrate the improvement that can be achieved. A relatively long piano part ([92], that lasts 624 seconds) was chosen and its spectrogram matrix was computed. (The data matrix can be downloaded from [93]; the procedure to obtain the spectrogram from an audio file is the described in [94]). 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 NMF of the data matrix, selecting k as 88 (number of piano keys, [89]). 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 only be done if the whole matrix is available, which is not true in a real time environment.

This large matrix was used to simulate real time processing. In such environment, the NMF of the part of the music already played in each time instant needs to be obtained. The computations were started with the matrix formed with the first 436 columns of V. The NMF of this submatrix was computed, and then one column (or several columns) at a time was added, performing the needed computations.

To use a full NMF 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, updating/downdating can be used 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, this updating procedure was chosen for the experiments.

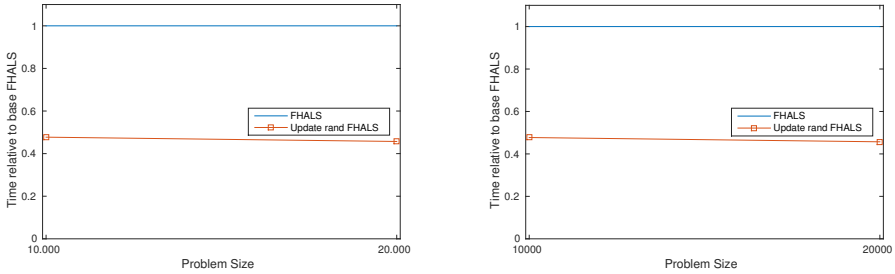
As a first approach the single column window procedure was used, that is, in each time instant a new column is added (at the right side of the data matrix) and an old column is discarded from the left side. Therefore the size of the matrix is fixed to 436 columns. The results are good from the point 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 NMF.

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, similar experiments to those shown in Section 6.5.1 were performed using matrices obtained from songs; the experiments were repeated 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 6.2 the execution times of those experiments are shown, while in Table 6.11 the corresponding error measures are shown. As in 6.5.1 the execution times are relative to execution times of the base fHALS algorithm, the base fHALS execution times are shown in Table 6.12.



(a) Relative execution time r=1

(b) Relative execution time r=200

Figure 6.2: Execution times of updating experiments with real application music matrices relative to fHALS base algorithm

Table 6.11: 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

Table 6.12: 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

6.5.5 Comparison with Online NMF

The updating algorithms proposed in this chapter can be used to tackle problems previously solved by means of the Online NMF (see Section 2.4.7). For example, when all the data points of a dataset are not available from the beginning and more data points (columns of the data matrix) are received over time. Furthermore, for the cases where the dataset does not fit entirely into memory, the window algorithm can be used to keep a constant memory cost.

For the sake of completeness, an experimental comparison between some of the updating algorithms proposed in this chapter(LSQ-fHALS updating and window algorithms) and an existing Online NMF algorithm is presented in this section.

The algorithm chosen is the ONMF with mini batch mode presented in [47], whose structure is shown in Algorithm 11. The matrix W update of line 5 is performed by using the second order PGD algorithm with diagonal approximation proposed in [47]. The NNLS problem of line 4 is solved using the BPP algorithm with multiple rigth hand sides (see Algorithm 3).

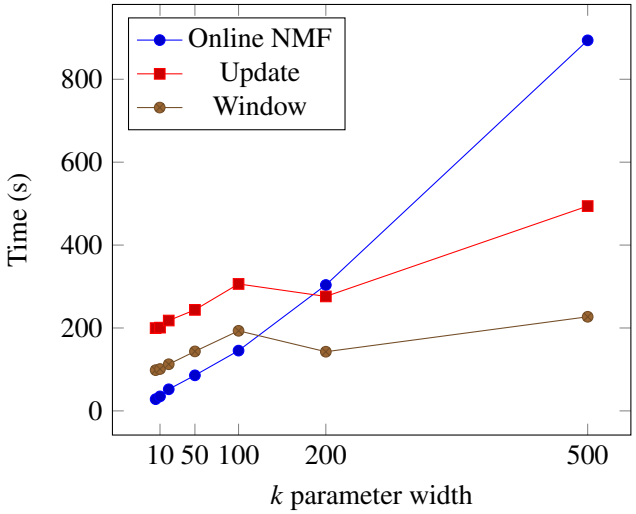
The updating algorithm used is the LSQ-fHALS due to its good stability and performance for both single and group updates. The window algorithm used in the test is the one based on the LSQ-fHALS too.

The main difference between the proposed updating algorithms and the Online NMF is that the updating algorithms keep updating matrices W and H incrementally with the new data points, while the Online NMF only updates matrix W. For each new data point (or data block), the Online NMF computes a new H losing any influence of the previously computed H matrices. The algorithm loses any information from the previous data points except the information contained on W. Even when the window algorithm is used, the proposed updating algorithms keep updating both matrices and keeping the history of all previous data points into its decomposition. This behaviour is more desirable when the data points of the data set are related between them and there is a continuity in the dataset(e.g. music transcription, sound source separation, etc.).

For the experimental comparison a data matrix of $m = 10000, n = 50000$ was generated using Algorithm 26 and the block size to simulate the new incoming data is $r = 1000$. The updating algorithms will use a base factorization with the first 11000 columns of the data matrix. The k parameter of the decomposition is changed during the experiments because it has a strong effect in the performance of the Online NMF algorithm.

Figure 6.3 shows the execution time of the entire simulation for each one of the three algorithms compared for different values of the internal size of the NMF k.

Figure 6.3: Execution time comparison between the Online NMF and the updating algorithms for different k values



As shown in Figure 6.3, the Online NMF algorithm is faster than the updating algorithms for small values of k but its execution time grows rapidly when k is increased, making it much slower for the biggest values of k tested. The window algorithm is faster than the update algorithm. This behaviour is meaningful because the window algorithm has the same cost for each update. On the other side, the cost of one update of the updating algorithm increases during the simulation because the matrices A and H grow in each update.

Table 6.13 shows the error obtained by the Online NMF, the updating algorithm and the window algorithm at the end of the simulation for different values of k. The error is computed using the same conventions as in Section 6.5.4. The results show that the Online NMF obtains the worst errors for all the experiments and the error gets worse when the k parameter increases. Furthermore, the update algorithm obtains a lower error than the window algorithm as expected, because the updating algorithm is not discarding any data and updating the factorization with all the available data at every moment.

Finally, despite the Online NMF is faster for small values of k, its error is very large compared with the updating versions. Furthermore, for bigger values of k it is slower than the updating algorithms and has a huge error.

Table 6.13: Error comparison between the Online NMF and the updating algorithms for different k values

Algorithm	Approximation errors					
	5	10	20	50	100	500
Online NMF	0.943	3.797	12.186	66.559	253.504	6031.6
Updating LSQ-fHALS	0.036	0.101	0.172	0.297	0.455	1.163
Window LSQ-fHALS	0.138	0.205	0.339	0.671	1.146	2.387

The window algorithm is faster than the updating algorithm at the expense of a slightly higher error. So, the window algorithm is preferable for memory or execution time bounded problems, while the update algorithm is preferable if the accuracy is the main goal.

6.6 Conclusions

We can safely conclude that the proposed algorithms solve the problems in less time than the NMF from scratch in all cases; and that LSQ versions (Alg. 23 and 24) lower the error measurements in addition to decrease the execution time.

GCD fHALS algorithm (Alg. 25) performs very well with single column updates being faster and slightly more accurate than LSQ algorithms (Algs. 23 and 24) 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. 22) 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. 24) 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.

The updating and window algorithms are suitable for large and incomplete datasets used in Online NMF models. In addition, the window algorithm can estimate the NMF of those large datasets with a constant memory cost. Both algorithms outperform the compared Online NMF algorithm.

Chapter 7

Efficient implementation of Active-Set Newton Algorithm for NonNegative representations

7.1 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 non-negative linear combination of basis vectors belonging to a precomputed "dictionary". This model is used in different applications, such as source separation [95], automatic music transcription [96], and sound event detection [97].

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 [98].

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 [99].

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

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

Note that the KL divergence is a particular case of the β-divergence (2.10) presented in Section 2.4 when $\beta = 1$. In the problem of obtaining nonnegative representations of audio for overcomplete dictionaries approached by [100], for each input signal $x \in \Re_+^{1\times f}$ a nonnegative vector $w \in \Re_+^{1\times n}$ that minimizes the KL divergence with respect to the dictionary $B \in \Re_+^{n\times f}$ should be found:

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

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 [100], 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 [101] and the projected gradient algorithm [38, pp. 267-268].

Due to the great performance of the ASNA algorithm but the lack of a computational efficient implementation of the algorithm, the author decided to improve the existing MATLAB implementation in order to obtain a lower execution time. This reduction of the execution time is necessary to approach real-time applications. The resulting implementation is an efficient parallel version suitable for shared memory multicore machines.

This work was considered within the scope of these thesis because the model used by the ASNA algorithm is a partial NMF where only one of the matrices of the decomposition is updated. Furthermore, NMF and the model used by the ASNA algorithm share some practical applications that make it worth to work on the algorithm in order to compare both approaches.

7.2 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 non-zero 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 non-negativity 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 [100, Sec. III].

The existing implementation programmed in MATLAB, that can be found in [102], uses a more general model than the one shown in the original algorithm [100, pp. 5]. The extended model can work with multiple observations at a 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 non-negative 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

$$V = W_A B_A \tag{7.1}$$

where A is the global active set composed of the union of the active sets of each observation X_m (a). In (7.1) 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 27. 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.

7.2.1 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 (7.2) where the weight of each atom $W_{m,n}$ is computed as (7.3) [91].

$$a = \underset{n}{\operatorname{argmin}} KL(X_m || W_{m,n} B_n) \qquad (7.2)$$

$$W_{m,n} = \frac{X_m 1^T}{B_n 1^T} \qquad (7.3)$$

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

7.2.2 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

$$\frac{d}{dW} KL(W) = (1 - \frac{X}{V}) B^T \qquad (7.4)$$

here the division of matrices is computed entry-wise and V is computed according to (7.1). Note that $1B^T$ can be precomputed at the initialization to save computation time during the iterations.

7.2.3 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 non-negativity. Let us denote a dictionary matrix whose rows consists 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 (7.1) 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 (7.5) and the Hessian matrix with respect to w_a computed at w_a is given by (7.6).

$$grad = (1 - \frac{X_m}{V_m})B_a^T \qquad (7.5) \qquad H_{w_a} = B_a diag(\frac{X_m}{V_m^2})B_a^T \qquad (7.6)$$

Here, "diag" denotes a diagonal matrix whose entries consists 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 (7.5).

Finally the weights are updated as (7.7) where α is the step size and the search direction can be obtained by solving the system of equations (7.8).

$$w_a \leftarrow w_a - \alpha\ searchDir \qquad (7.7) \qquad (H_{w_a} + \varepsilon I) \times searchDir = grad \qquad (7.8)$$

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 (7.7) are non-negative.

Algorithm 27 Original ASNA implementation algorithm

Input: $X \in \Re_+^{o \times f}$ $B \in \Re_+^{n \times f}$. **return** $W \in \Re_+^{o \times n}$

```
Normalize each dictionary atom to unity norm
Pre compute 1B^T for the gradient computations

Initialize active set for each observation
(Active atoms have values in W_A and not active are 0)
for i = 1 to maximum number of iterations do
    Find global active atoms A
    Compute V = W_A B_A (7.1)
    R = X/V (element-wise)
    if i mod K = 0 then
        Compute gradient w.r.t all weights (7.4)
        if i mod 10 = 0 then
            Check convergence for non converged observations
            Remove converged observations from the computations
            if all observation have converged then
                Scale back W and exit
            end if
        end if
        Mark as 0 the gradient of the already active weights
        Add the atom with the minimum gradient of each observation
to the active set, adding a small number to W_A
    end if
    Compute R2 = X/V^2 (element-wise)
    Find the indexes of the active atoms
    Compute sparse product Rcov = RB^T
    for each observation not converged X_m do
        Find the active atoms of X_m (a)
        if all gradients computed then
            Get grad from the already computed gradients
        else
            Compute gradients w.r.t active atoms of X_m (grad) (7.5)
        end if
        Compute Hessian H_{w_a} (7.6)
        Compute the search direction (7.8)
        Compute step size
        Update weights in W_A (7.7). If a weight becomes negative is removed.
    end for
end for
```

7.3 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 a version of the algorithm in C programming language was implemented using the HPC mathematical libraries BLAS and LAPACK. Finally, a parallel version of the algorithm was implemented using threading with OPENMP together with BLAS and LAPACK. A first approach to the proposed implementations was presented in [103]. The source code of all proposed implementations can be found in the repository [104]. All line numbers mentioned in the current section refer to Algorithm 27.

7.3.1 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 25 was reworked to use both matrices in column-wise order and the system of equations solving in line 34 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.

7.3.2 C implementation

The author 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 Section 7.3.1. 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 8 and 25 without the

need of finding the active atoms each time (lines 7 and 24), reducing the computation time needed for the sparse products. When removing active atoms in line 36 the atom should be removed from the atom list of observation X_m.

The second main improvement is that the sparse product on line 25, the computation of $R2$ (line 23), the computation of the gradient (line 31) and the computation of the Hessian (line 33) 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 34 has been solved by mean 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.

7.3.3 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 pragma *parallel for* for all loops which iterate along the observations. These loops correspond to lines 5, 8, 19, 20 and 26. 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 Section 7.3.2 the DPOTRF functions is already threaded inside the library. But in 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.

7.4 Analysed problem

The sound separation problem analysed in the original ASNA paper [100] 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 10000 atoms (50, 500 and 5000 atoms per speaker). In the present Chapter an evaluation with a bigger dictionary size of 50000 atoms per speaker will be presented. For more detailed information on the matrix generation process check [100, Sec. 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 [100].

7.5 Experimental analysis

7.5.1 Evaluation of the proposed implementations

The experimental environment, from now on called *Server*, consists of a multicore machine whose especifications are described in Appendix A.3. All the tests were executed using the 24 cores available. The development process was carried out in a multicore *Workstation* described in Apprendix A.5 All the tests were executed using the 4 cores available. Note that the workstation has a lower number of cores that 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, the KL divergence value was not evaluated in this experiment.

To compare the results of the proposed implementations with the experiments in the original ASNA paper [100], all implementations were tested with three different dictionary sizes (100, 1000 and 10000) until convergence was achieved. Furthermore, a new bigger dictionary with a size of 100000 atoms was tested. We will discuss that experiment deeper in section 7.5.3.

Table 7.1 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. The signal duration in seconds range from 1.46 to 2.71, with an average of 1.99 seconds.

	100	1000	10000	100000
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 7.1: Execution times of each ASNA implementation for different dictionary sizes on *Server* (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 (10000 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 of execution time obtained by the Parallel C implementation makes possible to use the ASNA algorithm with 1000 and 10000 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 10000 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 7.2 shows the speedup obtained from the different implementations respect the original MATLAB implementation.

	100	1000	10000	100000
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

Table 7.2: Speedup respect to the original MATLAB implementation

7.5.2 Hardware comparison

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

Table 7.3 shows the execution times in seconds of the same experiments presented in the previous section but in the *Workstation* machine.

	100	1000	10000	100000
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

Table 7.3: Execution times of each ASNA implementation for different dictionary sizes on *Workstation* (seconds)

The comparison shows that the MATLAB implementations obtain a lower execution time in *Workstation* (Table 7.3) than in *Server* (Table 7.1). The lower execution time in the MATLAB version is due to the higher CPU frequency on *Workstation* and the poorer utilisation of the multicore architecture of MATLAB (compared to 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.

7.5.3 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 100000 atoms were performed. As shown in Table 7.1 the execution time of the parallel C implementation for the 100000 atom dictionary is lower than the original MATLAB implementation for the 10000 atom dictionary.

The motivation of these experiments with bigger dictionaries was to check if it was worth to use that big dictionaries for the sound separation problem, because the 100000 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 100000 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 [100, Sec. VI.C], and is a commonly used objective metric in audio source separation evaluations. Figure 7.1 shows the evolution of the SDR with the progression of the algorithm for different dictionary sizes. As shown in Figure 7.1 , 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 7.4 shows the SDR achieved on convergence and the time needed to achieve it for the best proposed implementation.

	100	1000	10000	100000
Signal to distortion ratio (dB)	9.684	9.923	10.246	10.869
Execution time (s)	0.021	0.144	1.343	16.084

Table 7.4: Signal to distortion ratio comparison for the parallel C implementation

7.6 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 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.

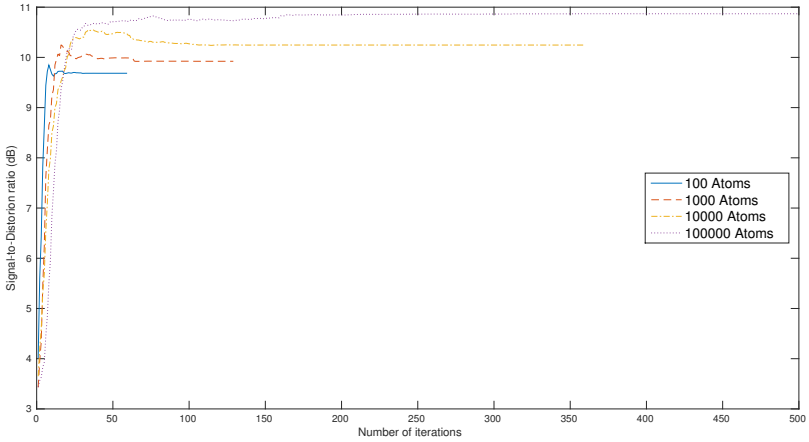


Figure 7.1: Signal to distortion ratio (dB) per iteration for the different dictionary sizes.

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 Chapter the reduction by more than 10 times of the execution time of state-of-the-art ASNA algorithm (which itself is significantly faster than the established expectation-maximization update rules) is shown. 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 my knowledge, the 100000-atom dictionaries used in this work 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 [105] and 10 000 atoms [100]. This Chapter shows 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.

Chapter 8

Implementations developed

During the development of this thesis different algorithms to compute the NMF or modified versions of the NMF problem have been implemented. While not all the algorithms have produced successful results, it is interesting to detail the implementations within the scope of this thesis. The ASNA algorithm implementation is excluded from this chapter because the implementations are detailed in its own chapter (see Chapter 7).

8.1 Sparsity and smoothness constrained multiplicative algorithms

These algorithms are based on the constrained multiplicative algorithms presented in [38, Ch. 3.4], which were designed to control the sparsity and smoothness of the factored matrices. The algorithms have been implemented on top of the β-divergence multiplicative algorithm implementations present on the NNMFPack library (see Chapter 4).

There are two restrictions implemented: smoothness enforcement and sparsity enforcement. For each one a restriction is applied to the numerator of the update rules (2.11).

In order to impose sparsity, problem (2.8) is modified with a regularization term based on the $\ell_1 - norm$:

$$W,H = \underset{W,H\geq 0}{\operatorname{argmin}}(D_\beta(A,WH) + \alpha_W||W||_1 + \alpha_H||H||_1) \tag{8.1}$$

Problem 8.1 leads to the following update rules:

$$H \leftarrow H \cdot \frac{(W^T((WH)^{\cdot\beta-2} \cdot A)) - \alpha_H}{W^T(WH)^{\cdot\beta-1}}, \quad W \leftarrow W \cdot \frac{(((WH)^{\cdot\beta-2} \cdot A)H^T) - \alpha_W}{(WH)^{\cdot\beta-1}H^T}, \tag{8.2}$$

where the operation $-\alpha$ is an elementwise subtraction applied to all the elements of the resulting numerator matrix. In the same way, problem (2.8) can be modified using the $\ell_2 - norm$ in order to impose smoothness to the solution:

$$W,H = \underset{W,H\geq 0}{\operatorname{argmin}}(D_\beta(A,WH) + \alpha_W||W||_F + \alpha_H||H||_F) \tag{8.3}$$

what leads to the following update rules:

$$H \leftarrow H \cdot \frac{(W^T((WH)^{\cdot\beta-2} \cdot A)) - \alpha_H \cdot H}{W^T(WH)^{\cdot\beta-1}}, \quad W \leftarrow W \cdot \frac{(((WH)^{\cdot\beta-2} \cdot A)H^T) - \alpha_W \cdot W}{(WH)^{\cdot\beta-1}H^T}, \tag{8.4}$$

here the operation $-\alpha_H \cdot H$ is an elementwise subtraction between the numerator and the scalar-matrix product $\alpha_H \cdot H$ (the same applies for matrix W). In both constrained updates, the α parameter is a user defined parameter to control the desired restriction for each matrix.

In the implemented source code, the restrictions are applied after computing the numerator of each update rule, before updating matrix H or W. The computations needed to apply these restrictions are trivially paralellizable. The constrained algorithms have been implemented on top of NNMFPack β-divergence multiplicative algorithms, so there are also specific restricted versions for the special cases of $\beta = 2$ and $\beta = 1$. The function prototype of the constrained algorithms has 3 parameters in addition to the parameters present in the non constrained function:

```
int <p>bdiv_<ARCH>(const int m, const int n, const int k,
    const double *A, double *W, double *H,
    const double beta, const int uType, const int nIter,
    const double alphaW, const double alphaH,
    const int restr);
```

First *alphaW* and *alphaH* are the α scalar to apply on the restriction for each matrix. Then the parameter *restr* indicates which restriction to apply and has the following valid values:

0. No restrictions applied.
1. Smoothness imposed to both matrices H and W.
2. Sparsity imposed to both matrices H and W.
3. Smoothness imposed to matrix W and sparsity imposed to matrix H.
4. Sparsity imposed to matrix W and smoothness imposed to matrix H.

when *restr* value is 0 or both $alphaW = 0$ and $alphaH = 0$, the non constrained version of the algorithm is called internally.

8.2 Affine NMF

The algorithm for affine NMF shown in Algorithm 5, has been implemented using BLAS operations which run in parallel if the BLAS implementation used is threaded. The four matrix multiplications of lines 6 and 9 have been implemented using the BLAS3 *dgemm* function and the matrix-vector multiplications of line 11 have been implemented using the BLAS2 *dgemv* function. Then line 5 has been implemented using two operations:

- *dgemm* (BLAS3): $\hat{A} = WH$
- *dger* (BLAS2): $\hat{A} = w1^T + \hat{A}$.

The column normalization of line 8 has been implemented by using two BLAS operations for each column w_j of W :

- *dasum* (BLAS1): $norm = ||w_j||_1$
- *dscal* (BLAS1): $w_j = w_j./norm$

Note that the operator $./$ represents an elementwise division of all elements on vector w_j by the scalar *norm*.

The elementwise products and divisions on lines 7, 10 and 12 have been implemented using OpenMP to paralellize the loop over all the elements on the matrix/vector.

8.3 Greedy Coordinate Descent (GCD) algorithm

Our implementation is based on the GCD algorithm presented in [44] and shown in Algorithm 10 of Section 2.4.6. However, the source code presented in that paper has significant differences with the algorithm. Some of them improved the performance of the theoretical algorithm in our tests and were used.

The improvement used from the original source code is that the initial matrix-matrix products of line 2 are performed inside the iteration loop before each update process. The products of $P^{AH} = AH^T$ and $P^{HH} = HH^T$ are performed in each iteration before the update of G^W in line 7. In the same way, P^{WA} and P^{WW} are computed at the beginning of the updating step for H. All this products are implemented with the BLAS3 operation *dgemm*. Doing the products inside the iteration process avoids the need of updating that products on line 5 and line 14. While theoretically it should be better to update only the modified elements, in practice, its is faster to perform the products.

Another improvement carried out by our implementation is to perform all the operations over matrix W and H, instead of over the partial matrices W^{new} and H^{new}. This speeds up the convergence and decreases the cost of the algorithm (by avoiding the matrix-matrix addition on line 22, the initialization of line 6 and its analogue steps in the H updating step).

The computation of G^W on line 7 is performed with the following operations:

- *dgemm* (BLAS3): $G^W = WP^{HH}$
- *daxpy* (BLAS1): $G^W = G^W - P^{AH}$

and the update of G^W on line 16 is performed by:

- *daxpy* (BLAS1): $G_i^W = G_i^W + s^* P_{q_i}^{HH}$

where G_i^W is a row of G^W and $P_{q_i}^{HH}$ a row of P^{HH}. Equivalent operations are performed for G^H on the updating step for H.

The computation of S^W and D^W of lines 8, 9, 17 and 18 has been performed with a custom function. This function computes each row of S^W and D^W in parallel using an OpenMP *parallel for* pragma, because there is no data dependency between the rows. When the *icc* compiler is used, the *simd* pragma from Intel parallelization technologies is used.

Then the BLAS1 operation *idamax* is used to compute q_i for each row of D^W on lines 10 and 19. The *idamax* function is used to compute p^{init} on line 10 too.

Again, an analogue function is designed to compute S^H and D^H in the updating step of H. Note that in this case the parallelization is made column-wise.

8.4 ANLS-BPP

The ANLS-BPP algorithm has been implemented with two alternate calls to the NNLS-BPP function, transposing the problem in the second function call. Due to this, in this section details on the implementation of the NNLS-BPP function are presented.

The NNLS-BPP function implements the BPP algorithm for the NNLS problem with multiple right hand sides shown in Algorithm 3 of Section 2.3.1. The most time consuming parts of the algorithm have been implemented using parallel efficient BLAS and LAPACK operations.

First, the initial computation of A^TA and A^TB in line 3 is performed using BLAS 3 *dgemm* function. Then, the computation of X_{Fj} on line 21 is performed with the following operations for each group of columns(see the column grouping explanation below):

- *dpotrf* (LAPACK): $L = cholesky(A_F^TA_F)$
- *dpotrs* (LAPACK): solve X_F in $LX_F = A_F^TB$

where A^TB_F is a matrix containing all columns of A^TB that share the same F and G sets, and so is the corresponding X_F. With this approach, we only compute one Cholesky factorization for all columns that share the same sets, saving computational time and resources. Furthermore, more resources are saved by computing the solution of all columns in A^TB_F with a single call to *dpotrs* with multiple right hand sides.

The computation of Y_{Gj} on line 22 is performed with the following operations:

- *dgemm* (BLAS3): $Y_G = A_G^TA_FX_F$
- *dmatSub* (custom op.): $Y_G = Y_G - A_G^TB$

again following the same grouping convention of the previous operation.

The sets F and G are disjoint sets. Thus, some computations can be avoided by updating and storing only set F and computing the elements on G on-line when needed. The F sets for all columns of X are stored as an integer matrix where each column stores the set of the corresponding column of X. Each column is an integer vector containing the indexes on F, hence it follows that the indexes not in the column vector are indexes in G.

Instead of using the set V in lines 12 to 20, the original sets used in [12] H_1 and H_2 are used, where $H_1 = \{i \in F_j : x_i < 0\}$, $H_2 = \{i \in G_j : y_i < 0\}$ and $|V| = |H_1| + |H_2|$. Using this sets the updating of the set F_J on line 20 simplifies to:

- Remove indexes on $H_{1\,j}$ from F_j
- Add indexes on $H_{2\,j}$ to F_j

which is performed by adding and removing indexes to each column j of the sets matrix containing F_j. Sets H_1 and H_2 are stored in matrices analogous to the integer matrix that stores the sets F_j. As said before, the updating of G is omitted.

For each $j \in I$ there are no dependencies between the columns of F, H_1 and H_2, therefore all operations in lines 12 to 20 are parallelized with an OpenMP *parallel for* pragma.

The column grouping is performed by comparing all F_j with $j \in I$ and grouping all the indexes j_1, j_2 where $F_{j_1} = F_{j_2}$. The column grouping is performed before the main computations of lines 21 and 22 in order to save resources during the computation. For each group, the necessary matrices are extracted from A^TA and A^TB and then the computations are performed.

For each group with $nElems$ columns sharing set F_j, the matrices $A_F^TB \in \Re^{sizeF \times nElems}$ and $A_G^TB \in \Re^{sizeG \times nElems}$ are extracted from the matrix A^TB computed in the initial computations of line 3. The matrix extraction is performed by iterating over the rows of matrix A^TB, copying all columns on the column group to the corresponding matrix A_F^TB if the row index is in F and to A_G^TB otherwise. Analogously, the matrices $A_F^TA_F \in \Re^{sizeF \times sizeF}$ and $A_G^TA_F \in \Re^{sizeG \times sizeF}$ are extracted from the matrix A^TA. In this case, the iteration over the rows of A^TA is performed for each column of A^TA whose index is in F.

The feasibility check and the search of the infeasible columns I of lines 9 and 10 are performed altogether for each group after computing $X_{F\,j}$ and $Y_{G\,j}$. During that check the feasibility sets H_1 and H_2 are created too. Each time an element is added to H_1 or H_2 for any column j, j is added to I and the general solution is set to infeasible. This feasibility check operations are independent for all columns of each group, because of this the computation is performed in parallel using an OpenMP *parallel for* pragma.

Due to the initialization, the operations in line 7 and line 8 can be omitted, and the feasibility check can be simplified to check if $Y_{i,j} > 0 \forall i \in \{1,...,q\}$., adding to $H_{2\,j}$ all indexes that do not suit the condition and j to the infeasible columns set I.

8.5 HALS algorithm

The implementation of the HALS algorithm shown in Algorithm 8 needs to receive initialized W and H matrices, and a number of iterations to perform in the outer loop (because no convergence criterion has been implemented). A parallel algorithm has been implemented using BLAS operations when possible, and implementing custom parallel functions otherwise.

The nonnegative enforcement operator $[\quad]_+$ of lines 7 and 8 has been implemented using the *parallel for* construct of OpenMP [58]. This was done to speed up the operation because it is a strictly parallelizable operation with no dependencies. When using the Intel Compiler (*icc*) the *simd* construct from Intel paralellization technologies is used instead of the OpenMP construct.

Due to the possibility to receive initialization matrices (W and H) of the proposed implementation, matrix H is transposed on entry into matrix B and transposed again on exit from B to H. This matrix transposition is implemented using the parallelization technologies mentioned above. Each column of matrix H is processed by a single thread.

In the same way, each column of the normalization on line 2 is processed by a thread.

The initial error estimation from line 3 is performed with two operations:

- *dgemm* (BLAS3): $E = WB^T$
- *dsub* (custom op.): $E = A - E$

lines 6 and 10 have been optimized by doing in-place rank one updates to matrix E, instead of using a different matrix to contain $A^{(j)}$:

line 6:

- *dger* (BLAS2): $E = w_j b_j^T + E$

line 10:

- *dger* (BLAS2): $E = -w_j b_j^T + E$

Then each column update of b_j and w_j is performed with another two operations:

Update of b_j (line 7):

- *dgemv* (BLAS2): $b_j = E^T w_j$
- *halfwave* (custom op.): $b_j = [b_j]_+$

Update of w_j (line 8):

- *dgemv* (BLAS2): $w_j = E b_j$
- *halfwave* (custom op.): $w_j = [w_j]_+$

Finally, the normalization of each column of W performed in lines 2 and 9 is implemented with the following operations:

- *dnrm2* (BLAS1): $norm = ||w_j||_2$
- *dscal* (BLAS1): $w_j = w_j./norm$

Note that the operator $./$ represents an elementwise division of all elements on vector w_j by the scalar *norm*.

8.6 fHALS algorithm: CPU and GPU implementations

This algorithm is one of the best algorithms to compute the NMF. Its structure with a high amount of matrix-matrix and matrix-vector multiplications makes it suitable for building a high performance parallel implementation of the algorithm for both multicore and GPU architectures.

In order to maximize the performance of the algorithm, the BLAS package was used for both CPU and GPU implementations. Furthermore, most of the CPU implementations of BLAS include threaded implementations for multicore CPUs of some functions. The functions needed to implement the fHALS algorithm are usually threaded. On the other side, the cuBLAS library for GPU is implemented to make the most of each NVIDIA GPU architecture. Therefore cuBLAS performance is hardly achievable by user kernels.

First of all, the details that are common to both CPU and GPU implementations are exposed, and then the singularities of the GPU implementation are explained. In the CPU version, all the non BLAS operations are performed as described in Section 8.5. The line numbers in the following subsections make reference to Algorithm 9.

8.6.1 Common Implementation Details

The matrix-matrix products on lines 6,7,12 and 13 of algorithm 9 have been implemented using the BLAS3 *dgemm* function. This is the most costly part of the algorithm and at the same time, the *dgemm* function is the most optimized function of the BLAS interface. This will have a huge impact in the high performance of the proposed implementations.

Then line 9 is implemented with two operations:

- *dgemv* (BLAS2): $x_j = -Bv_j + x_j$
- *daxpy* (BLAS1): $b_j = x_j + b_j$

In a similar way line 15 is implemented with 3 operations:

- *dgemv* (BLAS2): $p_j = -Wq_j + p_j$
- *daxpy* (BLAS1): $p_j = q_{jj}w_j + p_j$
- *dcopy* (BLAS1): $w_j = p_j$

The normalization of each column of W performed in lines 3 and 16 is implemented with the following operations:

- *dnrm2* (BLAS1): $norm = ||w_j||_2$
- *dscal* (BLAS1): $w_j = w_j./norm$

Note that the operator $./$ represents an elementwise division of all elements on vector w_j by the scalar *norm*.

8.6.2 GPU Implementation Details

The nonnegative enforcement operator $[\ \ ]_+$ of lines 9 and 15 has been implemented using the CUDA kernel listed below. The kernel is executed with a one dimensional grid of threads, which launches a thread per element of the enforced vector (b_j **or** w_j respectively). There are no dependencies in this operator so it is a perfect scenario for a SIMT kernel.

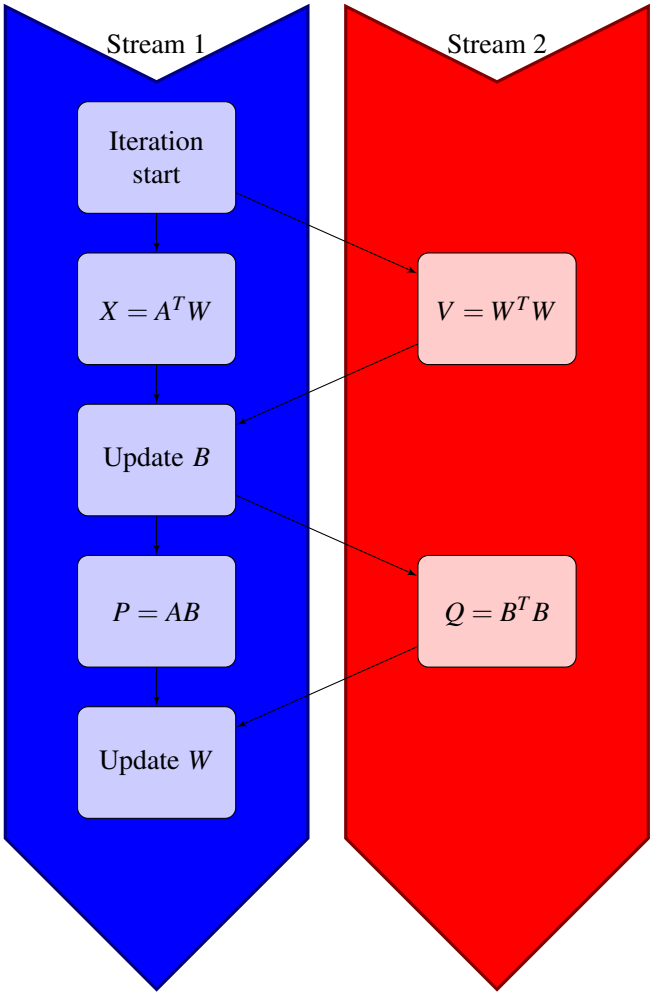
```
__global__ void vdhalfwave_cuda(const int n, double* __restrict__ x)
{
    unsigned int pos = blockDim.x * blockIdx.x + threadIdx.x;

    if (pos < n)
        if(x[pos] < EPS)
            x[pos] = EPS;
}
```

The matrix transposition needed to change the entry matrix H into the matrix $B = H^T$ used in the algorithm is performed using the BLAS-like extension function of cuBLAS *dgeam*.

The other peculiarity is the creation of two CUDA streams so the matrix-matrix operations on lines 6, 7 and lines 12, 13 can be executed concurrently. Each two multiplications are independent between them, and can be executed concurrently if the GPU has enough cores to perform both operations simultaneously. This concurrent execution will depend on the problem size and the hardware resources of the GPU. The planner of cuBLAS is in charge of that decision. The algorithm flow for a parallel stream execution is shown on Figure 8.1.

Figure 8.1: Execution flow of one iteration of the fHALS GPU algorithm with two streams in parallel.



Chapter 9

Experimental Evaluation

In this chapter, some experimental evaluations performed during the development of this thesis are presented. First, an evaluation of the β parameter of the multiplicative β-divergence algorithm for solving the NMF is presented. Then, the best implementations developed are compared in terms of computational performance and execution time. In section 9.3, an evaluation of the fHLAS GPU implementation is presented and the fHALS GPU algorithm is compared with the MLSA algorithm present in the existing NNMFPack library. Finally, some conclusions about the experimental results are discussed.

9.1 Beta parameter evaluation

In real applications the choice of the β parameter presents a problem for the users of the NMF when the Beta divergence multiplicative algorithms are used (see Section 2.4.3. The users must decide what value to use to obtain the best approximation. Initially, there is no best β parameter value for the NMF that ensures the minimal error among different values of β. In practice, the accuracy of the solutions given by the β-divergence algorithms seems to be related to the target problem. Due to this ambiguity an evaluation of the influence of the parameter β in the approximation error for different problems was performed.

The easiest way to evaluate this relation may be to use artificial datasets, but these experiments may not show the relations among data. That relations can probably influence the behaviour of the errors in the approximation of A by WH. Therefore to test Machine description properly the experiments must be performed with real datasets from practical problems, because these data may contain latent variables that affect to the final result of the factorization.

Exploiting that feature, this evaluation computed which value of the β parameter is the best to obtain the best approximation for each type of problem. Obviously, different matrices of each type were used to generalize the analysis for a given subset of problems.

9.1.1 Algorithms and data

The computational library NNMFPACK (see Chapter 4) was used in the experiments. NNMFPACK has an implementation of the multiplicative algorithm (2.11) called *dbdiv_cpu*. This routine has eight parameters: *dbdiv_cpu(m, n, k, A, W, H, beta, iters)*. The first three are the dimensions of the problem (m,n,k). Then, matrix A is the problem matrix. Matrices W and H, on input, are the initialization matrices W_0 and H_0; on output they are the solution matrices W and H. Last, $beta$ is the β parameter and $iters$ is the number of iterations.

For the evaluation process, the algorithm was executed several times varying some of the parameters. The β parameter values were selected between 0 and 2 because these are the values usually used in practical applications [29]. Derived from observations during the tests, different β values were used in some experiments as it will be explained in the proper section. The iteration number (*iters*) was set to 100, 200 and 400 iterations. Other NMF packages (*e.g.* MATLAB) use 100 as their default number of iterations that the algorithm must perform, this number was increased to evaluate the effect on the error measures. Finally for the k parameter the size selected was $\min(m,n)$ divided by 2, 4 and 8. Each k gives us a problem which will be tackled by varying the other two parameters.

In order to assess correctly the influence of the β parameter, the algorithm was tested over data from different types of problems. So we can look deep through the relation of the problem data and the β parameter.

Despite the algorithm tries to minimize the β-divergence error the Frobenius norm of $A-WH$ was used as another measure of quality of the solutions. Both error measures were computed as:

$$err_F = \frac{\|A-WH\|_F}{\sqrt{mn}},\ err_\beta = \frac{\sqrt{2D_\beta(A|WH)}}{\sqrt{mn}}. \tag{9.1}$$

Observe that *dbdiv_cpu* tries to minimize err_β but for $\beta = 2$ we get $err_F = err_\beta$. Obviously the highest decrease in err_F is expected to be reached when $\beta = 2$.

9.1.2 Test matrices

The matrices used in the experiments are described below.

1. **Random matrices:** The problem matrix A was generated randomly and so were the initialization matrices W_0 and H_0. The matrices were generated using an uniform distribution. To avoid side effects coming from the random number generation method, some tests were performed with normal distribution generation too. Motivated by the results of the first experiments the tests were repeated with several ranges of the elements of the matrices:

   (a) *[0-1]:* The basic approach was to generate random matrices with entries between 0 and 1.

   (b) *[0-255]:* Grey scale images give rise to matrices with entries between 0 and 255. In this case random matrices in the same range of the images (0-255) were tested.

   (c) *[x-y]:* The results of the experiments with the previous two types showed that the range of the matrix values is related with the β parameter influence. Some more tests were executed with different ranges of values to prove this relation between β and the range of the matrix values.

2. **Synthetic matrices:** In this experiment two random matrices W and H with a fixed k were used to create the matrix A. The matrix was created as: $A = WH + \varepsilon$, being ε a little constant error. The initial matrices W_0 and H_0 were generated randomly as in the experiments with type 1 matrices.

3. **Images:** This experiment was developed using grey-scale images as data matrices. For each image processed, a set of initial matrices (W_0, H_0) with certain k were created. Each set of 3 matrices (A, W_0, H_0) gives us a complete problem to test the β parameter and the number of iterations.

   (a) *[0-255]:* The original images were equivalent to matrices valued between 0 and 255.

   (b) *[0-1]:* Using the information of the random matrices experiment, some tests were done scaling the matrices to 0-1 range.

4. **Audio matrices:** The last experiment was to decompose matrices in the frequency domain [106] extracted from music tracks. These matrices were a great candidate to obtain information of the relation between the data of the matrix and the best β value because the matrices are extracted from a real world problem. The initial matrices W_0 and H_0 were generated following a uniform 0-1 distribution and a folded normal distribution in different tests.

   (a) *without scaling:* The elements of the original audio matrices in the spectrum domain had different ranges of values. The test were performed with 5 audio matrices called from this point forward: audio matrix 1-5.

   (b) *scaled:* The audio matrices of (a) were scaled down to values between 0 and 1.

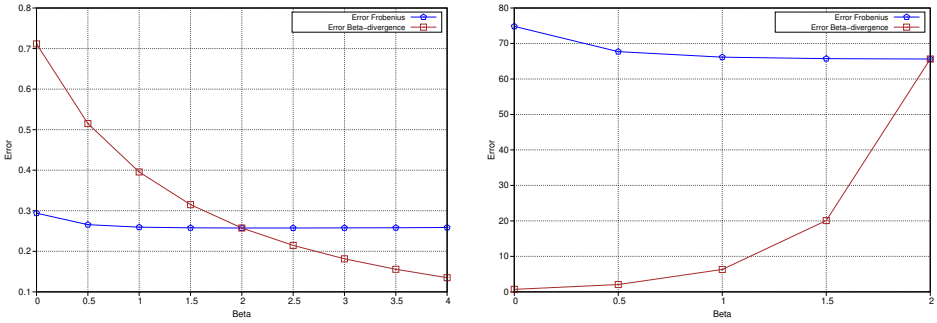
9.1.3 Experimental results

All the experiments were performed in Server1, which is described in Appendix A.1.

The NNMFPACK library works over several architectures and can be installed with different software configurations. The NNMFPACK has parallel implementations of the evaluated algorithm for GPU and many-core (Intel MIC), but this work does not focus on execution speed so the test were performed only on CPU. The software configuration used in this installation of the library was: icc as compiler and MKL as mathematical library.

Random matrices

Tests proved that there was no difference between both types of random number generated matrices (normal distribution or uniform distribution), when the *dbdiv_cpu* algorithm is applied. The experiments with the matrices of type *1.(a)* [0-1] showed that the Frobenius error and the β-divergence error always decrease when the number of iterations are increased as expected. With regard to variation of β parameter, both error measures have their maximum value in $\beta = 0$ and decrease continuously as β increases up to $\beta = 2$. In the case of $\beta > 2$ a difference between both error measures appears. The Frobenius error has its minimum in $\beta = 2$. On the other hand, err_β decreases while β increases. In the multiplicative updates (2.11) the β is an exponent and this causes an overflow when big values of β (*e.g.* $\beta > 128$) are reached. Therefore the best β-divergence error that can be achieved is the one with the biggest β executable without overflowing. This behaviour is represented in Figure 9.1(a).



(a) Random 0-1 matrices

(b) Random 0-255 matrices

Figure 9.1: Evolution of Frobenius error and beta-divergence error

In the matrix type *1.(b)* [0-255] the Frobenius error keeps the same behaviour, with its best value in $\beta = 2$. Nevertheless the β-divergence error has its minimum value in $\beta = 0$ and increases with the increment of β. This behaviour is represented in Figure 9.1(b). Note the different value of the Frobenius error, which is 20 times lower in the [0-1] case than in the [0-255] case.

Finally for the matrix type *1.(c)* the behaviour was the same as for the type *1.(b)* tests. That is, Frobenius error having its minimum in $\beta = 2$ and β-divergence having it in $\beta = 0$. Furthermore, the errors were bigger for bigger values of the upper limit of the range (y) and for wider range ($y - x$).

Synthetic matrices

The goal of this experiment was to check the accuracy of the NNMF when an exact solution exists for a given k. The error added to the matrix creation is to simulate some noise in the data matrix.

The trends of both error measures were the same as in the case of the random matrices, in the [0-1] case as well as in the [x-y] case and both errors increase as the noise introduced in the generation process (ε) does. If $\varepsilon = 0$ the algorithm returned the exact matrices for all β values.

Images

For the type *3.(a)* [0-255] matrices, the same behaviour is observed for both measures: With more iterations the error decreases. The lowest error value is obtained always with $\beta = 0$, increasing the error while β is increased (see 9.1(b)).

For the type *3.(b)* [0-1] matrices, both error measures decrease when the number of iterations increase. The err_β starts from the same value than in the matrices of type *3.(a)* but now decrease while β is increased. In addition, the Frobenius error has a different behaviour, keeping its lower value in $\beta = 0$ and increasing it as β grows. Despite maintaining its tendency, the scaling decreases notoriously the Frobenius error for all β values. Table 9.1 shows an example of these results.

Table 9.1: Image value range comparison 0-255 vs 0-1 with 100 iterations

Case	β **value**				
	0	0,5	1	1,5	2
err_β 0-255	0.142010	0.465982	1.576253	5.424420	19.030323
err_β 0-1	0.142010	0.116609	0.098708	0.085005	0.0746287
err_F 0-255	15.296724	15.663154	16.539595	17.644217	19.030323
err_F 0-1	0.059987	0.061424	0.064861	0.069193	0.0746287

Note that this experiments and the corresponding results have been carried out by iterating up to 400 iterations. This is the reason why it seems to have a different behaviour compared to the random and synthetic matrices experiments. That experiments showed that the Frobenius error always had its minimum value for $\beta = 2$, contrary to the results of that images test which gives us a minimum value for $\beta = 0$. But, increasing the number of iterations in several tests showed that the minimum Frobenius error value tends to happen for $\beta = 2$. In the image whose data is in the Table 9.1 we achieve the point where the minimum Frobenius error matches $\beta = 2$ with 13200 iterations. This effect will be detailed in Section 9.1.3. Once achieved that point, the matrix types *3.(a)* and *3.(b)* behave like the matrix types *1.(b)* and *1.(a)*. This behaviour is represented in Figures 9.1(b) and 9.1(a).

Audio processing matrices

For the matrix type *4.(a)* as the number of iteration increases both error measures decrease as expected. The β parameter shows the influence of the relation between the data into the matrices giving us two different behaviours: the Frobenius error has its minimum for $\beta = 1.5$ for all audio matrices and the β-divergence error has its minimum in a different value depending on the matrix as shown in Table 9.2. The Frobenius error

being minimal in $\beta = 1.5$ implies that each problem has a different optimum value for β. Furthermore, having β-divergence minimum values for different β values implies that each matrix can behave differently for the β-divergence error measure.All these measures were carried out with up to 400 iterations.

Table 9.2: Value of β parameter for the minimum β-divergence error

audio matrix #	1	2	3	4	5
β	0.5	0	0	0.5	0.5

For the audio matrices type *4.(b)* [0-1] both error measures decrease notoriously and behave in the same way as in the type *3.(b)* [0-1] (see Figure 9.1(a)). The Frobenius error decreases when β increases keeping its behaviour and the β-divergence error begins decreasing with the increment of β instead of increasing.

To ensure the Frobenius error will reach the best value in $\beta = 2$, as said in subsection 9.1.3, we tested all audio matrices (*4.(a)* and *4.(b)*) increasing the number of iterations. The Figure 9.2 shows the evolution of the Frobenius error for each β value. They always reach the point were the Frobenius error is smaller for $\beta = 2$ than for $\beta = 1.5$. But there is no fixed number of iterations where that will occur (*e.g.* audio matrix 1 arrives to the point around 1200 iterations, audio matrix 2 around 6200 and audio matrix 3 around 3300).

9.1.4 Result analysis

The main results obtained during the experiments are outlined here:

1. As the experiments showed, increasing the number of iterations always decreases the error obtained, with independence on the β parameter. This is the expected behaviour because the multiplicative algorithm used is based on a gradient descent minimization. Is the user choice to increase the number of iterations to get a better result. There is a trade-off between computing time and accuracy that the user must evaluate.

2. The experiments also showed that scaling the matrix to the range 0-1 both error measures decrease, independently of what is the chosen β value. Also, when the matrices are scaled to this range, β-divergence error decreases with the increment of β indefinitely, as seen in Section 9.1.3.

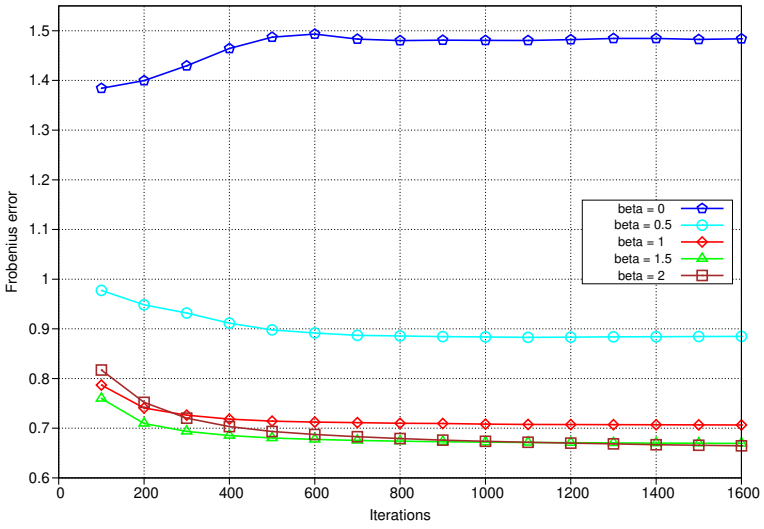


Figure 9.2: Evolution of the Frobenius error when the iteration number is increased in audio matrix 1.

3. The Frobenius error always has its best value in $\beta = 2$ if enough iterations are done. That is completely logical because for $\beta = 2$ the β-divergence is mathematically equivalent to the Frobenius error. So for $\beta = 2$ we are minimizing the Frobenius error.

4. Taking only the iteration number as measure of algorithm progression, it seems not worth to increase the number of iterations so much to achieve the minimum value of the Frobenius error at $\beta = 2$. But if total execution time is considered it is worth. The NNMFPACK library has been optimized for certain values of β. In the case of $\beta = 2$ the library uses an efficient implementation of the MLSA algorithm [37] which is considerably faster than the generic β-divergence algorithm. Figure 9.3 shows the variation of execution times when iterations are increased. Generic cases ($\beta = 0$, $\beta = 0.5$ and $\beta = 1.5$) have the same computational cost. As can be seen, a big amount of iterations for $\beta = 2$ are still faster than a few iterations for $\beta = 1.5$ achieving a smaller Frobenius error.

5. Matrices from real applications (image and audio) always reach better error with the same number of iterations than its random equivalent. This shows that the NNMF exploits the relation between data as expected.

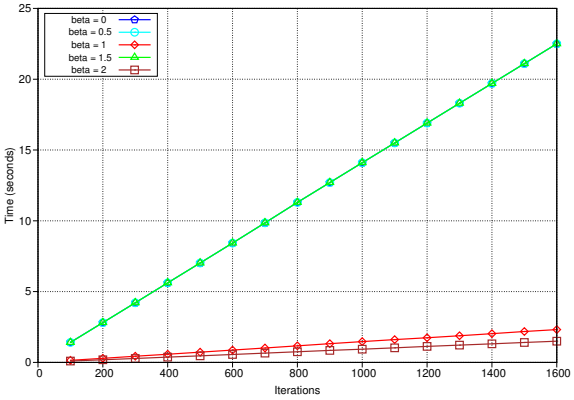


Figure 9.3: Execution times.

9.2 Experimental comparison of the proposed implementations

The purpose of the experiments presented in this section is to compare the performance of the implementations developed. The tests are designed to test the performance of the proposed implementations (see Chapter 8) and not the performance of the algorithms themselves. The performance of the algorithms was evaluated by the original authors of each algorithm when the algorithms were presented [43, 44, 35]. The experiments into this section have been executed in the machine described in Appendix A.4 using only 40 physical cores to avoid core overlapping. In order to avoid system load effects during the timing of the algorithms, each test was executed 10 times and the execution time obtained was averaged.

The experiments show the results obtained by testing the best implementations developed to solve the general NMF (2.3): fHALS and GCD algorithms. These implementations are tested against the MLSA algorithm present in the existing NNMFPack library (this algorithm is hidden under the β-divergence multiplicative algorithms with $\beta = 2$, check Chapter 4 for additional information). The HALS algorithm is also tested for reference against its fast version.

9.2.1 General implementation comparison

The first tests which results are presented in Table 9.3 were performed using random matrices with values between 0 and 1 for all three A, $W0$ and $H0$ matrices. The random numbers were generated using a uniform distribution. As each algorithm has a different error reduction per iteration, all algorithms were executed until they achieved a given error bound. This is the proper way to test the performance of the implemented algorithms, because the time needed to achieve the desired error is compared. The error measure used in all the experiments of this section is the following:

$$err_F = \frac{\|A - WH\|_F}{\sqrt{mn}} \tag{9.2}$$

The selected error for the tests was the one given by the MLSA algorithm with 100 iterations. Usually, less than 100 iterations are used when solving problems with the multiplicative algorithm from Lee and Seung [15, 94, 106]. In some cases the GCD algorithm obtained a error lower than the MLSA algorithm with only one iteration and the desired error was adjusted to that one, thus increasing the number of iterations of the MLSA algorithm.

Table 9.3 shows the results of the experiments for two matrix sizes of $m,n,k = (1000,800,200)$ and $m,n,k = (10000,8000,200)$. The k dimension is fixed to 200 because k is usually smaller than m and n and related to the problem solved by the factorization. Together with the execution time obtained by each algorithm and matrix size, the number of iterations needed too achieve the error bound and the error obtained are shown.

Table 9.3: Experimental comparison of the selected implementations to compute the NMF with random (0,1) matrices

	(1000,800,200)			**(10000,8000,200)**		
algorithm	Iters	Time (s)	Error	Iters	Time (s)	Error
MLSA	100	0.199	2.384e-01	200	17.722	2.830e-01
HALS	24	2.216	2.402e-01	10	63.339	2.817e-01
fHALS	10	0.066	2.387e-01	6	0.758	2.826e-01
GCD	3	0.978	2.400e-01	2	7.835	2.825e-01

The results of Table 9.3 show clearly that the proposed fHALS implementation is the fastest, while the HALS algorithm has a very poor performance being even slower than the MLSA algorithm. The GCD algorithm obtains good results, being faster than MLSA for the bigger size but much slower than fHALS algorithm. However, the performance obtained is far from the claims of the original paper, which claimed that GCD was faster than fHALS algorithm.

The NMF has a strong dependence on the data into the matrix to factorize, so the performance of the implementations may change depending on the data matrix. Furthermore, the inner loop of the GCD algorithm has a convergence criterion which may affect the results obtained in a significant way. Figure 9.4 shows the execution time of the tested algorithms for three different matrix types of size $m,n,k = (1000,800,200)$: random matrices with values in the range $0-1$, random matrices with values in the range $0-1$, and synthetic matrices. The synthetic matrices were generated by creating random $0-1$ W and H matrices, multiplying them and adding an small error proportional to the norm of the matrix. In this case the error used was 0.001.

Figure 9.4: Execution time of the tested implementations for three different matrix types of size $m,n,k = (1000,800,200)$.

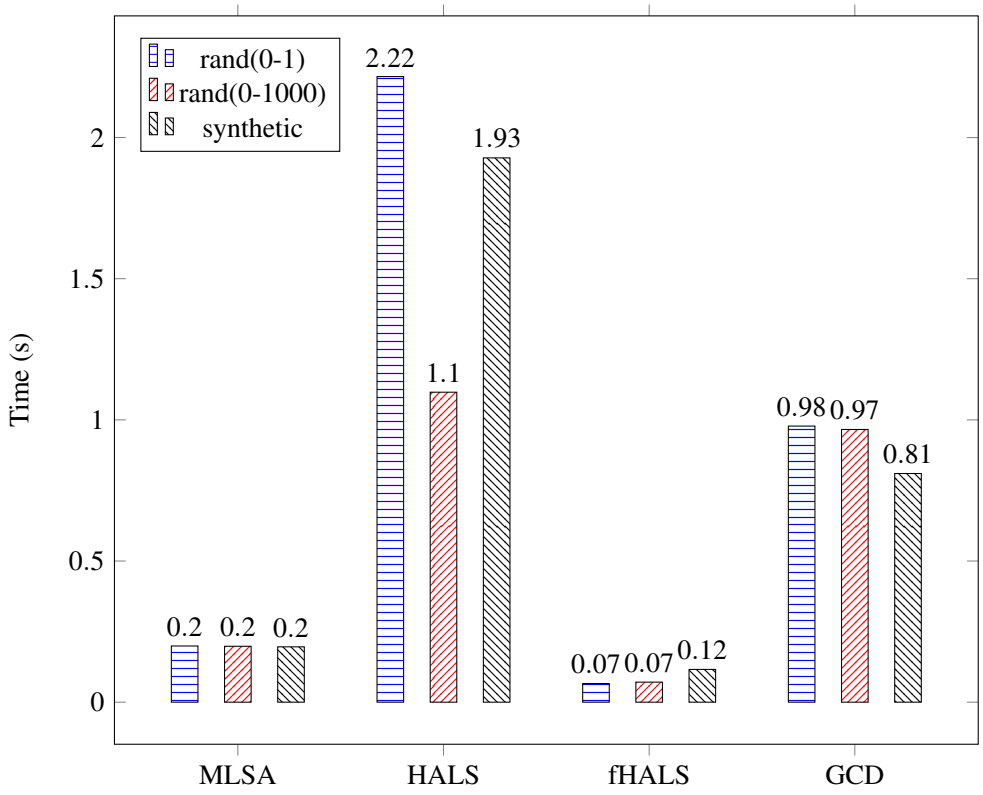
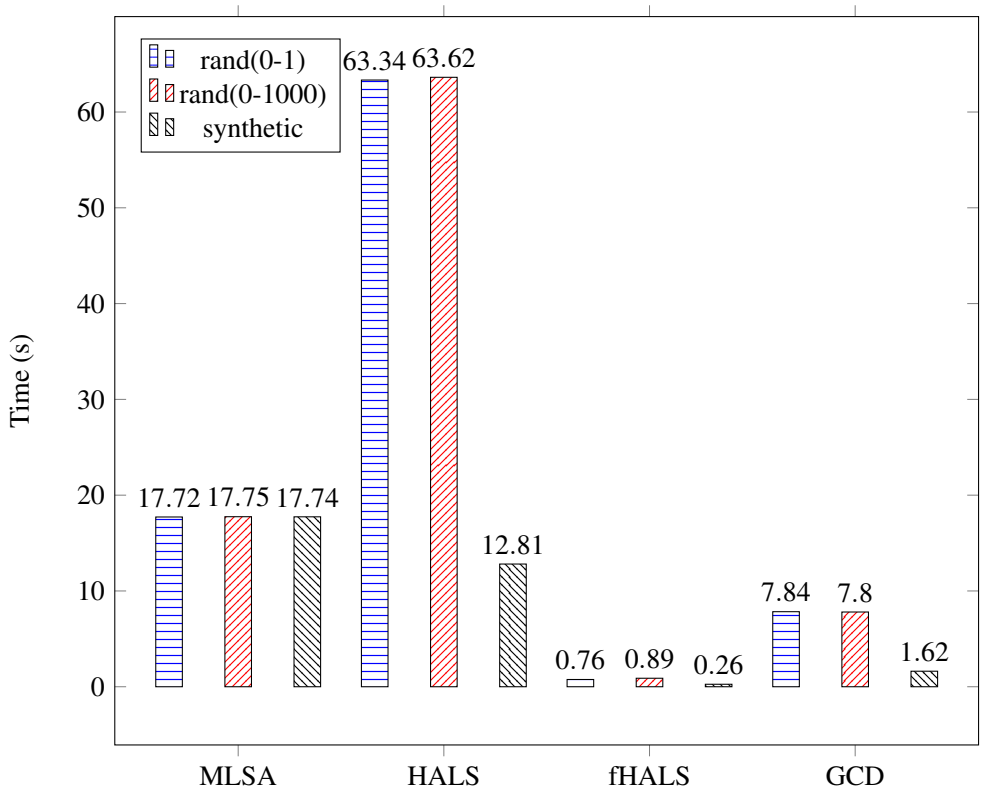


Figure 9.4 shows that MLSA is very stable, offering the same execution time for all matrices tested. On the other hand, the HALS algorithm seems to be sensitive to the values of the matrix, offering better results with a wider range of values into the matrix. fHALS

algorithm does not seem to suffer that influence from the data values. Furthermore, fHALS algorithm keeps being the fastest algorithm for all matrix types. GCD algorithm is still slower than MLSA but its performance increases for synthetic matrices.

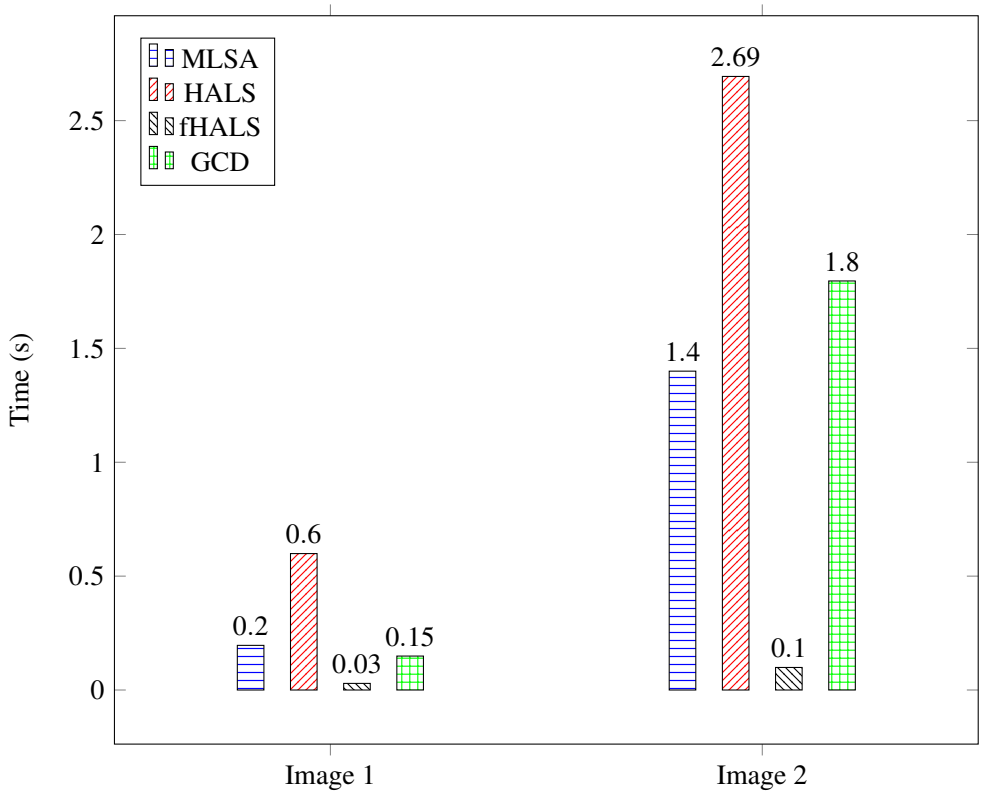
On Figure 9.5 the same matrix type comparison is performed with matrices of size $m,n,k = (10000, 8000, 200)$. In this case, MLSA is stable with the matrix type again but HALS algorithm has a significant performance improvement for the synthetic matrices. This performance improvement is because the algorithm needs only a few iterations to achieve the error bound selected. Again, fHALS is the fastest algorithm, but GCD is faster than MLSA as seen in Table 9.3. Furthermore, the performance of GCD is greatly increased for the synthetic matrices.

Figure 9.5: Execution time of the tested implementations for three different matrix types of size $m,n,k = (10000, 8000, 200)$.



In order to test the performance of the proposed implementations with real application matrices, the implemented algorithms were tested with two image matrices used in image compression applications. Figure 9.6 shows the performance of each implemented algorithm for each image. Again fHALS is the fastest algorithm with a notable difference for both images. The GCD algorithm implementation is faster than MLSA for the first image and slower for the second image. This may be explained due to the inner convergence condition of the GCD algorithm, which as said before, may be sensitive to the data in the matrix.

Figure 9.6: Execution time of the tested implementations for two different image matrices. Image 1 has a size of $m,n = (950,950)$ and the k used in this test is 95. Image 2 has a size of $m,n = (1536,2304)$ and the k used in this test is 154.



9.2.2 *Influence of matrix dimensions on the performance*

In the general implementation comparison the GCD was faster than MLSA on the bigger size and slower in the smaller. In order to clarify that apparent relation between the matrix size and the performance of the algorithms, Figure 9.7 shows the execution time of the four tested algorithms for different matrix sizes. The experiments shown in the figure were performed using random matrices with values in the range $0-1$ and increasing the matrix size proportionally while keeping the k fixed $(m,n,k = (2500,2000,200), (5000,4000,200), (10000,8000,200), (15000,12000,200))$.

Figure 9.7: Evaluation of the influence of the matrix size in the performance of the proposed implementations

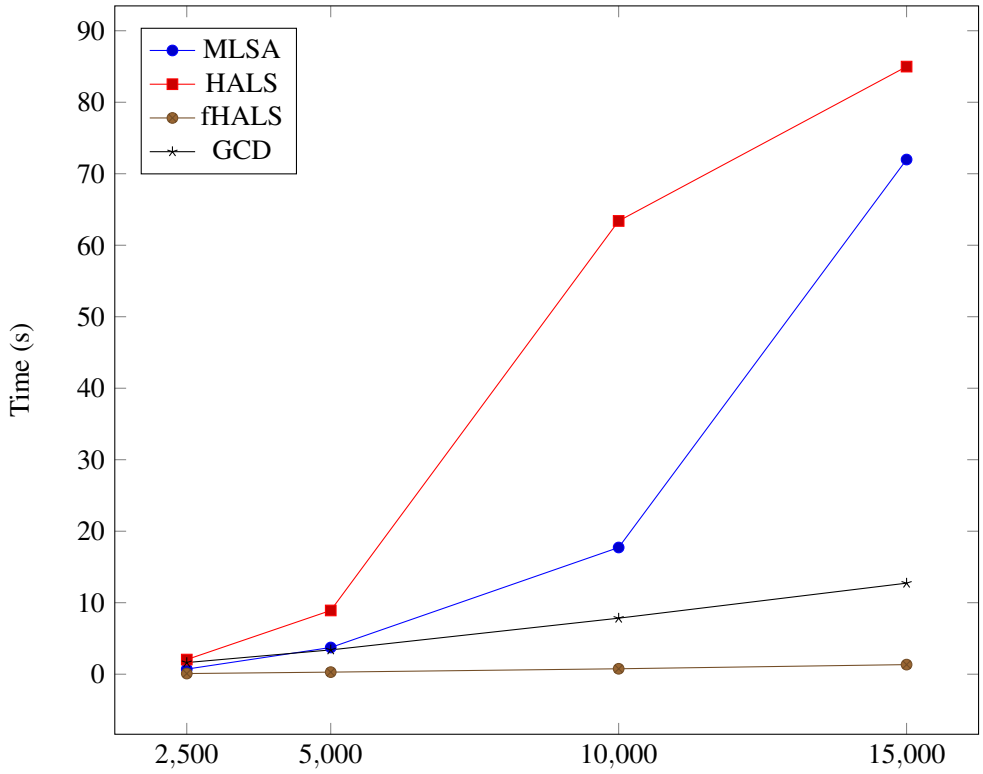


Figure 9.7 shows that the execution time of MLSA and HALS algorithm is greatly increased with the problem size. Furthermore, while MLSA algorithm is faster than GCD algorithm for the smallest sizes, it becomes slower for sizes bigger than $m = 5000$. The

execution time of fHALS algorithm only increases slightly with the problem size, which shows that fHALS algorithm is even better for big problem sizes.

The GCD and fHALS algorithms internal loops iterate over the k dimension of the decomposition, so the better performance of these algorithms in the bigger matrix sizes may be due to $k << m,n$. This may affect specially the GCD algorithm because the convergence criterion of the inner loop impacts highly the overall performance of the algorithm. To test that, Figure 9.8 shows an evaluation of the influence of the k dimension on the performance of the proposed implementations. For that tests, a matrix size of $m,n = (10000, 8000)$ have been chosen and the k dimension have been incremented from 200 to 3200.

Figure 9.8: Evaluation of the influence of the k dimension in the performance of the proposed implementations

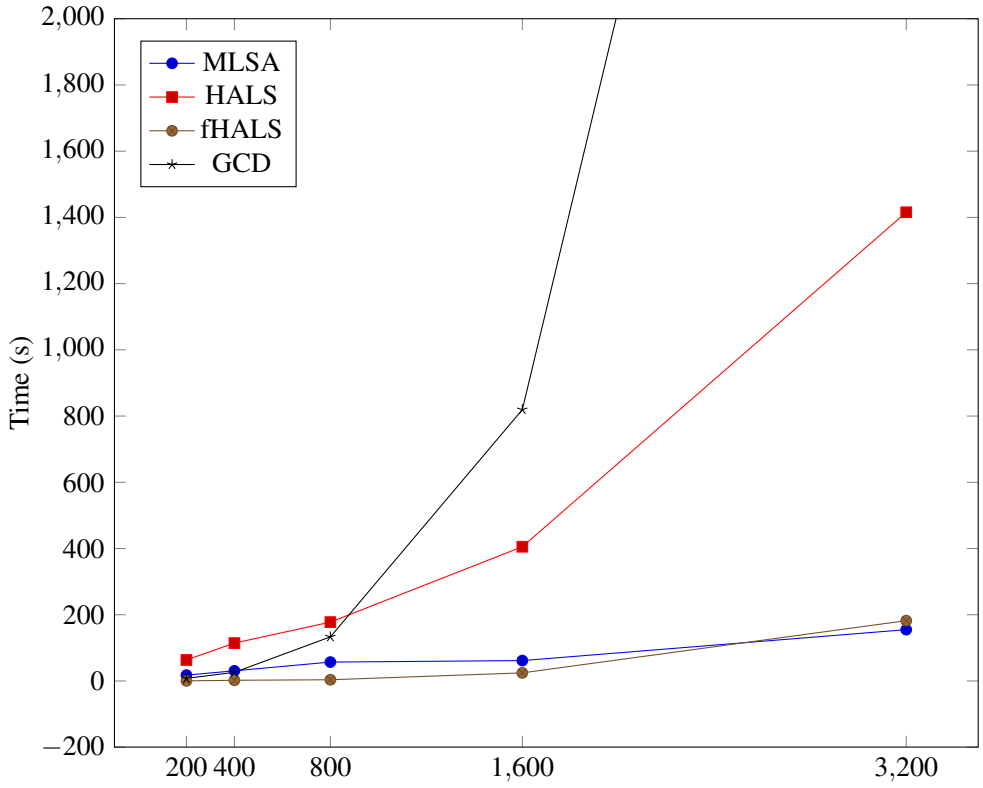


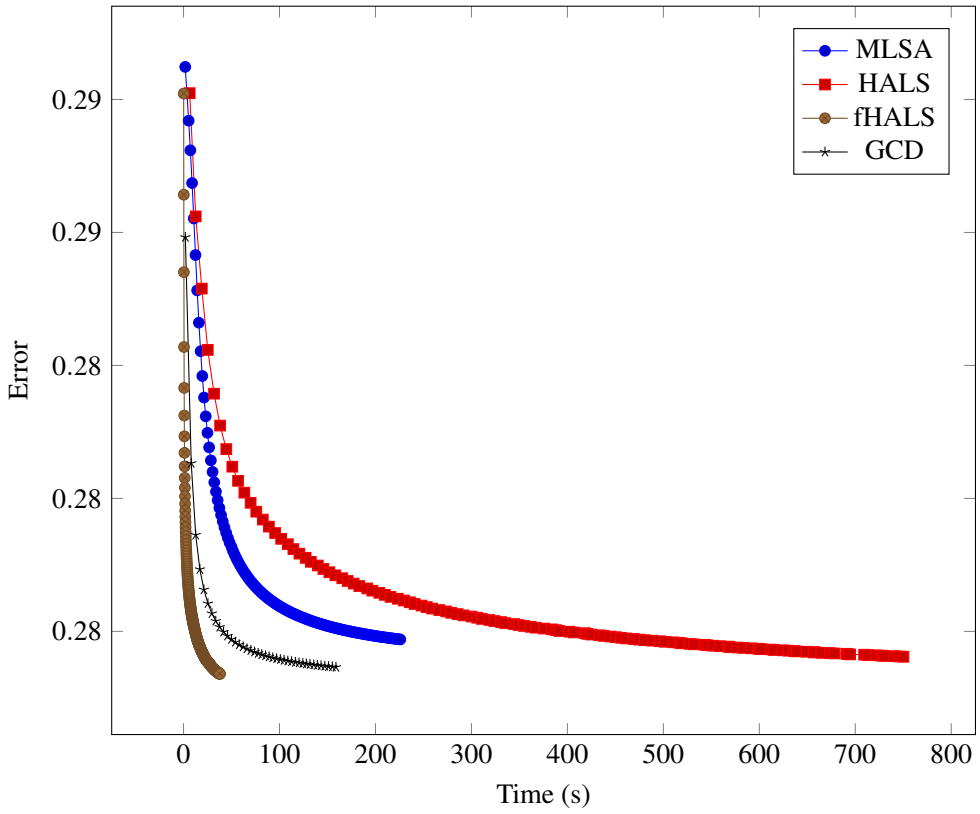
Figure 9.8 shows that the performance of GCD, HALS and fHALS (in a smaller proportion) decreases when the k dimension of the problem increases. Furthermore, the execution time of the GCD implementation was cropped form the figure for $k = 3200$ because it was too high. With that value of GCD the proportions of the figure did not allowed to see the differences between MLSA and fHALS and those differences are a key point of Figure 9.8.

9.2.3 Convergence evaluation of the implemented algorithms

At the beginning of Section 9.2.1 it was stated that the algorithms have different error reduction per iteration. This may cause that in the previous experiments some algorithms may benefit of the error bound selected. Those algorithms that decrease the error greatly in the first iterations but converge to a bigger error value are benefited if the error bound is bigger than the error achieved after convergence. On the other hand, some algorithms may converge slower but achieve a lower error bound.

Figure 9.9 shows the convergence properties of the proposed implementations by plotting the error value as the execution time goes by. The results show that both GCD and fHALS implementations obtain a lower error value when the convergence is achieved and their errors decrease faster than the error of the MLSA implementation. More precisely, the fHALS algorithm has the fastest error decrease and achieves the lower error once convergence is achieved.

Figure 9.9: Convergence comparison of the implemented algorithms



9.3 fHALS GPU evaluation

The performance evaluation was performed in a the machine described in Appendix A.4. All CPU tests were performed with 40 threads to avoid physical core overlapping and only one GPU was used.

Like in the general experiments, 10 different executions have been measured and averaged for each test case. This was done to avoid system load side effects in the time measurement.

Table 9.4 shows the execution time and error result of both algorithms MLSA and fHALS executed over both architectures GPU and CPU. The evaluation shown in Table 9.4 has been performed with three different test cases of different size. As the size of

k is usually much smaller than m and n, we have kept a fixed $k = 2000$ while increasing the size of m and n in order to test the influence of the problem size on the performance of each implementation. The three test cases evaluated are $(m,n,k) = (5000,4000,2000), (10000,8000,2000)$ and $(20000,16000,2000)$.

Taking into account that the fHALS algorithm has a larger error reduction per iteration than MLSA, the number of iterations of each algorithm has been fixed to a certain value to achieve a similar error in both algorithms. That way a fair execution time comparison can be performed. As stated before, comparing both algorithms with the same number of iterations would not be right. All experiments in this section have been performed with 165 iterations for MLSA algorithm and 3 iterations for fHALS algorithm, except the smaller problem $(5000,4000,2000)$ which needed 4 fHALS iterations.

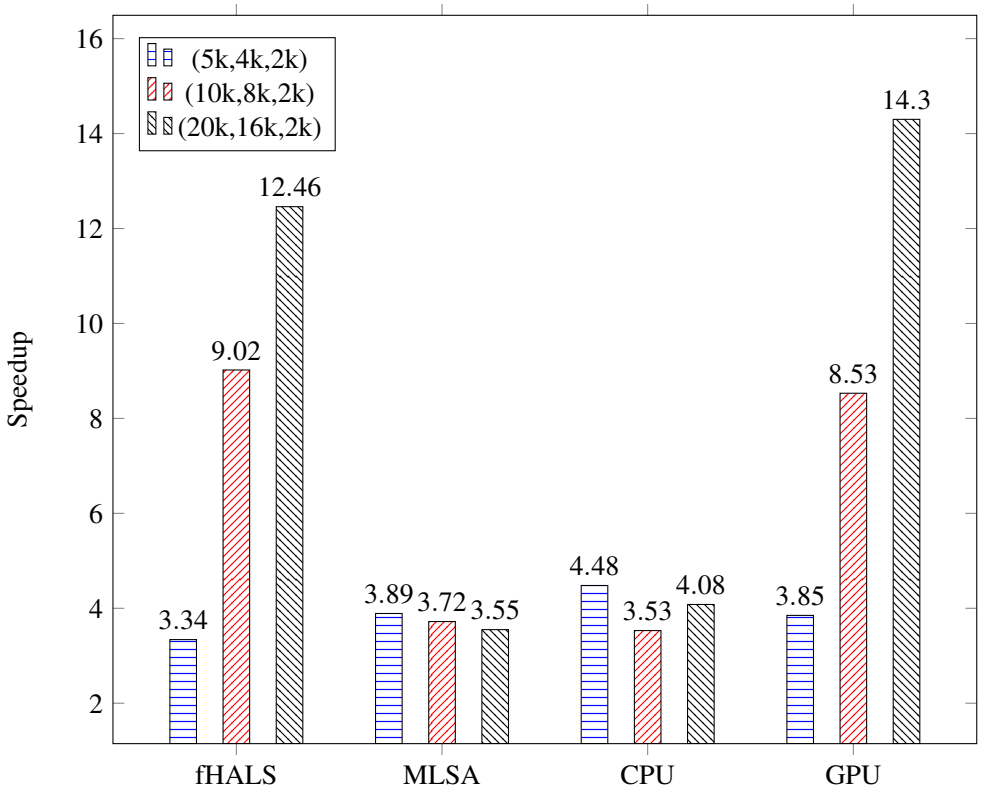
Table 9.4: Performance comparison between HALS and MLSA algorithms using both CPU and GPU architectures for different problem sizes

		(5k,4k,2k)		**(10k,8k,2k)**		**(20k,16k,2k)**	
arch.	algorithm	Time	Error	Time	Error	Time	Error
CPU	MLSA	45.75	2.518e-01	136.33	2.767e-01	464.68	2.854e-01
	fHALS	10.20	2.513e-01	38.61	2.850e-01	113.89	2.863e-01
GPU	MLSA	11.75	2.519e-01	36.55	2.767e-01	130.79	2.854e-01
	fHALS	3.05	2.4928e-01	4.28	2.768e-01	9.14	2.833e-01

The results presented in Table 9.4 show that the fHALS algorithm is faster than MLSA in both architectures tested. Furthermore, the GPU implementation of fHALS algorithm improves its performance with the increase of the problem size. This is easier to observe in the speedup comparison presented in Figure 9.10.

Figure 9.10 shows 4 speedup comparisons for the 3 problem sizes evaluated in Table 9.4. The first one labelled as *fHals* shows the speedup of the GPU implementation of the fHALS algorithm with regard to the CPU implementation. The second comparison labelled as *MLSA* shows the speedup of the GPU implementation of the MLSA algorithm with regard to the CPU implementation (implementations present in the NNMFPack library). Then, the third block labelled as *CPU* represents the speedup between the CPU versions, showing the speedup of fHALS algorithm with regard to MLSA algorithm. Finally, the last 3 bars labelled as *GPU* show the speedup between the GPU versions, again the speedup of fHals algorithm with regard to MLSA algorithm.

Figure 9.10: Speedup comparison



The first block of Figure 9.10 shows clearly that our GPU implementation of the fHALS algorithm improves its performance with the increase of the problem size. This is probably due to the great performance given by the GPU for matrix-matrix multiplications that represent the most of the cost of the fHALS algorithm. Furthermore, due to the k being fixed, the inner loops of Algorithm 9 which are less efficient represent a smaller part of the cost in the bigger problems.

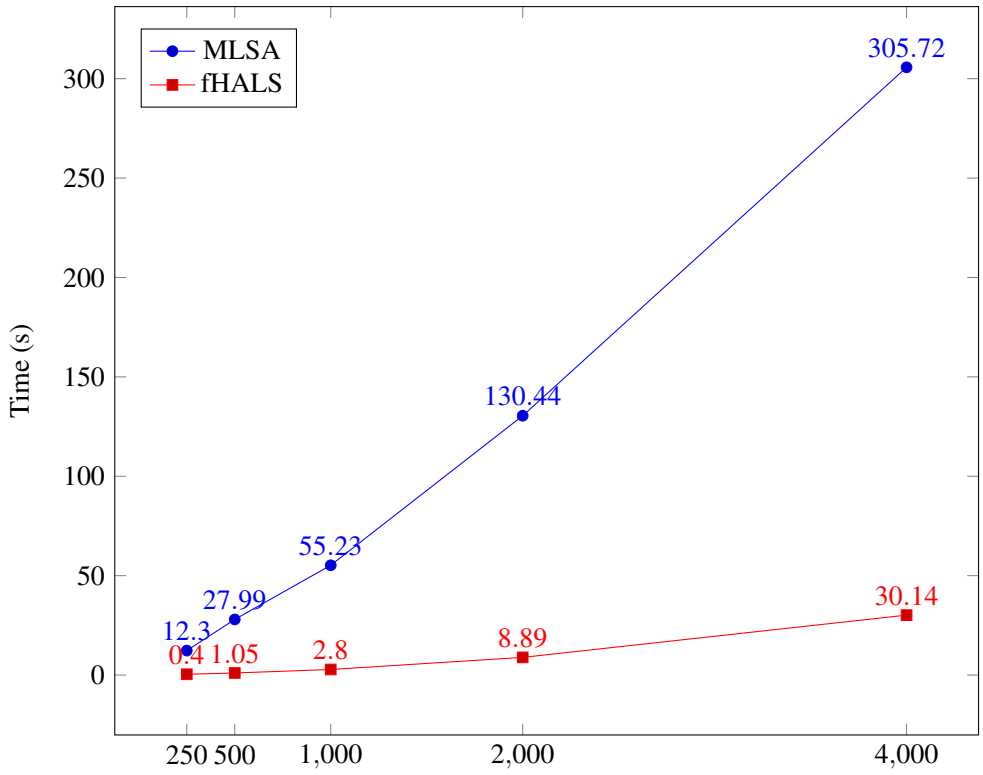
The second block surprisingly shows that the performance of the MLSA algorithm implementation for GPU decreases slightly respect the CPU implementation when the problem size grows.

The comparison between the CPU version of both algorithms shows a constant speedup around 4 independent of the problem size. This is consistent with previous literature which showed that fHALS algorithm is faster than MLSA algorithm.

Finally, the comparison between the GPU version of both algorithms again show that our GPU implementation of the fHALS algorithm has a great performance that increases with problem size.

Figure 9.11 shows the execution time obtained by the GPU versions of both algorithms with problems of $m = 20000$ and $n = 16000$ with variable k. The figure shows that the GPU implementation of the MLSA algorithm is more sensitive to the k dimension of the problem than the fHALS GPU implementation.

Figure 9.11: Evaluation of the influence of k dimension in GPU implementations



9.4 Conclusions

First of all the β parameter evaluation offered interesting information about the influence of the β parameter on the NMF when using the β-divergence multiplicative algorithms. Furthermore, the experiments helped to prove empirically some claims about the NMF like the exploit of the data relationships within the matrix.

The experimental comparison showed that the proposed fHALS implementation is the fastest for all cases among all of the implemented algorithms. However, the evaluation of the influence of the k dimension showed that the proposed fHALS implementation may be slower than the MLSA implementation for big values of k. Furthermore, the convergence evaluation shows that fHALS achieves the lowest error with the lowest execution time.

Finally, the GPU evaluation shows the great performance obtained by the GPU implementation of the fHALS algorithm. Furthermore, the performance of the GPU implementation presented increases with the problem size, making the GPU fHALS implementation a very powerful tool for those scientists and engineers that use the NMF in high dimensionality problems. These experiments showed again that both proposed implementations (GPU and CPU) of fHALS algorithm have better performance than its counterparts for the MLSA algorithm.

Chapter 10

Improved NNMFPack library

During the development of this thesis many efficient parallel algorithms have been implemented to compute the NMF and to find the solution of NNLS problems. This implementations are meant to be used by other scientists, engineers or whoever can benefit from the properties of the NMF in their work. As the potential users of the proposed implementations may not have computer programming knowledge, it is necessary to build a computational library containing the implementations developed. The development of an easy to use NMF computational library is one of the main goals and accomplishments of this thesis. Because it is the way to share the improvements obtained in the computation of the NMF with the scientific community and the society.

10.1 Library overview

Instead of developing a new computational library from scratch with the proposed implementations, the author decided to continue the NNMFPack project that was originally developed by the INCO2 research group in which the author have worked during the development of this thesis. For more information about the existing NNMFPack check Chapter 4.

Integrating the proposed implementations into NNMFPack allowed to take profit of the already developed multi-architecture installation process and some common functions already integrated into the library. On the other hand, the integration of new algorithms

improves the overall quality of the library increasing the possibilities offered by the library. Furthermore, some of the new implemented algorithms outperform the equivalent algorithms existing into the library.

Due to the introduction of a wide range of algorithms with different purposes the library has been structured to make it easier to understand for the users. This new structure will be detailed in Section 10.2.

In addition, a new *Python* interface have been added to the library. Nowadays there are many scientist moving from the traditional solutions of MATLAB/Octave towards the *Python* programming language for scientific computation. This is due to *Python* usability and the availability of powerful scientific packages like *numpy* or *scipy*. Including a *Python* interface for the library increases the number of potential users of the library.

Together with the addition of the new algorithms, some code maintenance and improvements have been performed to the library. Furthermore, the source code is now hosted in a public GIT repository[107].

10.2 Implemented algorithms

The new implemented algorithms together with the existing algorithms have been divided into three main categories:

- **NMF General:** In this category are included all the algorithms that perform the computation of the NMF minimizing the Frobenius norm. The algorithms into this category receive on entry the matrix to decompose A, the initialization matrices $W0$ and $H0$, the factorization dimensions (m,n,k) and the number of iterations to perform. On output the factorized matrices W and H are obtained.
- **NMF Specific:** This category groups the algorithms that have special properties or minimize a different metric to compute the NMF.
- **NNLS:** In this category are included algorithms to solve the NNLS problem with multiple right hand sides.

First of all we are going to list the implemented algorithms under the *NMF General* category and justify its inclusion into the library. For all of them, matrices W and H are on input the initialization matrices $(W0, H0)$ and on output the result of the factorization. In the function prototype definition $< p >$ represents the floating point precision (s for single precision or d for double precision) and $< ARCH >$ corresponds to the target architecture (cpu, gpu or mic).

- **MLSA:** The MLSA algorithm was already available into NNMFPack under the $\beta = 2$ case of the β-divergence multiplicative algorithms. With the new structure of the library it is better to expose the function directly to the user under the General NMF category. The function prototype of the algorithm is:

```
int <p>mlsa_<ARCH>(const int m, const int n,
        const int k, const double *A, double *W,
        double *H, const int nIter);
```

- **fHALS:** The fHALS implementation is the fastest implementation of the HALS algorithm and the one that proved to be the best algorithm to compute the NMF between the tested algorithms. The author decided to include only the fHALS algorithm into the library because the HALS algorithm yields the same results but with a higher execution time. The function prototype of this algorithm is:

```
int <p>fhals_<ARCH>(const int m, const int n,
    const int k, const double *A, double *W,
    double *H, const int nIter);
```

- **ANLS-BPP:** The ANLS-BPP algorithm has proved to be the only algorithm whose performance is comparable to fHALS algorithm and it can be faster than fHALS algorithm for certain datasets. This is a strong enough reason to include the algorithm into the library. The function prototype of this algorithm is:

```
int <p>anlsbpp_cpu(const int m, const int n,
    const int k, const double *A, double *W,
    double *H, const int nIter);
```

- **GCD:** While the GCD algorithm is slower than the fHALS algorithm, its variable selection strategy that decreases the number of variable updates may enable the CGD algorithms to obtain good resultas with certain datasets, specially sparse and large datasets. The function prototype of this algorithm has an extra parameter *epCD* that represents the accuracy used to solve the internal subproblems (typically $epCD = 0.001$):

```
int <p>GCD_cpu(const int m, const int n, const int k,
    const double *A, double *W, double *H,
    const int nIter, const double epCD);
```

Next, the following algorithms are implemented under the *NMF Specific* category:

- **β-divergence multiplicative algorithms:** The original algorithms present in NNMFPack have classified this category. Furthermore, the option of computing a partial NMF have been moved to the *NNLS* category for code clarity and because it fits better the type of problem under that category. The new function prototype is:

```
int <p>bdiv_<ARCH>(const int m, const int n,
    const int k,const double *A, double *W,
    double *H, const double beta, const int nIter);
```

- **Constrained β-divergence multiplicative algorithms:** The sparsity and smoothness constrained β-divergence multiplicative algorithms have been added.With this algorithms the user has the possibility to enforce the desired level of sparsity or smoothness into the computed NMF. The function prototype adds three new parameters respect to the unconstrained β multiplicative algorithms. The scalars *alphaW* and *alphaH* to control the strength of the restriction into the correspondent matrix, and the *restr* parameter to choose which restriction to apply:

```
int <p>bdivRestrict_cpu(const int m, const int n,
    const int k,const double *A, double *W,
    double *H, const double beta, const int nIter,
    const double alphaW, const double alphaH,
    const unsigned short restr);
```

- **NMF affine:** The affine NMF algorithm is useful to improve the quality of the NMF decompositions for datasets with constant parts within the observations. While it computes the NMF minimizing the Frobenius norm, it is classified under the *NMF Specific* category because its benefits are limited to certain datasets or problems. The function prototype includes the vector *w* which is the offset to absorb the common parts of the entries of the dataset:

```
int <p>affine_cpu(const int m, const int n,
        const int k, const double *A, double *W,
        double *H, double *w, const int nIter);
```

Finally, under the *NNLS* category are grouped several algorithms that compute a partial NMF decomposition where only one of the factorized matrices is computed. Which in the case of frobenius norm minimization corresponds to a multiple right hand sides NNLS problem.

- **BPP:** The BPP NNLS algorithm is the building block of the ANLS-BPP algorithm to compute the NMF. But on its own, is an excellent algorithm to solve NNLS problems with multiple right hand sides. The function prototype is as follows:

```
int <p>bpp_cpu(const int q, const int r,
        const int p, const double* A,
        const double *B , double *X);
```

  where matrix $A \in \mathbb{R}^{p \times q}$ is the coeficient matrix, $B \in \mathbb{R}^{p \times r}$ is the matrix whose columns are the independent vectors of each NNLS problem and $X \in \mathbb{R}_{+}^{q \times r}$ is the solution matrix whose columns are the solutions to each NNLS problem.

- **Partial β-divergence multiplicative algorithms:** The option to perform a partial decomposition with the β-divergence multiplicative algorithms is the equivalent to an NNLS with multiple right hand sides but minimizing the β divergence instead of the frobenius norm. The function prototype is the following:

```
int <p>bdiv_<ARCH>(const int m, const int n,
    const int k, const double *A, double *W,
    double *H, const double beta, const int uType,
    const int nIter);
```

  when *uType* is set to *UpdateAll* the corresponding β-divergence multiplicative algorithm is called internally.

- **ASNA:** The ASNA algorithm algorithm performs a equivalent operation to solving a NNLS problem with multiple right hand sides but minimizing the Kullback-Leibler divergence. The parallel version of the ASNA algorithm has been included with the following function prototype:

```
int <p>asna_cpu(const int f, const int n,
    const int o, const double* X,
    const double* B,
    double* W ,const int iter, const int nnz);
```

  where $X \in \mathbb{R}_{+}^{f \times o}$ is the matrix to factorize, $B \in \mathbb{R}_{+}^{f \times n}$ is the dictionary matrix, $W \in \mathbb{R}_{+}^{n \times o}$ is the weights matrix to compute and *nnz* is a parameter to restrict the memory need of the algorithm representing the maximum number of active observations.

Unlike the existing algorithms of the library, the new implemented algorithms are not available for all the architectures supported by the library. Only the fHALS algorithm has been implemented for all the supported architectures, due to its good performance shown in Chapter 9 and its good properties to be implemented into accelerator architectures.

Chapter 11

Conclusions

The research performed during the development of this thesis had two main branches: the updating of nonnegative factorizations and the efficient parallel implementation of algorithms to solve nonnegative factorizations. In this chapter, the main contributions arising from both research branches are discussed. The published results are listed in Section 11.4 and some future work is outlined in Section 11.5. Finally, some acknowledgements are presented in Section 11.6 in thanks for the support and funding received from several institutions to complete this thesis.

11.1 Updating of Nonnegative factorizations

This branch of the thesis focused on the algorithmic part of the problems, in contrast with the other branch of the thesis which focused on the computational implementation of the algorithms. In this branch, the problem of updating nonnegative factorizations has been studied for the Nonnegative Least Squares problem (NNLS) and for the Nonnegative factorization (NMF). An algorithmic scheme was developed for each problem. Several updating algorithms were proposed following the developed scheme, and the experimental tests offered promising results. In addition, the updating of the NMF was tested with data that came from a real-world application.

The key contributions in the updating branch are:

- the idea of using an updating scheme to solve the updating of nonnegative decompositions, decreasing the time needed to compute the solution of the updating problem.
- the mathematical description of the different variants of the updating problem applied to the NNLS problem.
- the proposed algorithms for solving the updating problem for the Nonnegative Least Squares problem and for the Nonnegative Matrix Factorization.

11.2 Implementation of algorithms to solve nonnegative factorizations

The main contribution of this branch is the development of an efficient computational library to compute Nonnegative decompositions, which evolved from the existing NNMFPack project. The new library supports several new algorithms to compute the Nonnegative Matrix Factorization, as well as some algorithms to solve the Nonnegative Least Squares problem. The new algorithms included were selected from the best existing algorithms for solving the NMF. This library allows high performance implementations of the supported algorithms to be executed in order to compute the NMF with an easy-to-use interface. Furthermore, the user does not need to know the details of each algorithm in order to take advantage of its properties. In addition, a new *python* interface has been developed to increase the number of potential users of the library. This interface has great potential due to the increase in popularity in the last few years of *python* for scientific computation programming.

During the development of the efficient parallel implementations of the new algorithms that are included in the library, an experimental evaluation has been carried out to compare the performance of the implemented algorithms. This evaluation was necessary because the theoretical performance of the algorithms may change once they are implemented using High Performance Computing techniques. The experimental evaluation showed the difference in performance among the algorithms implemented for a set of test cases. This difference is shown by comparing the execution time needed to achieve a given error bound by each algorithm implementation. Furthermore, an empirical convergence comparison was performed to check the convergence over time for the implemented algorithms.

The fHALS algorithm was implemented for GPU architectures because of the good performance shown in the experimental evaluation and its good algorithmic properties. Most of the computational cost of the fHALS algorithm comes in the form of matrix-vector and matrix-matrix products, which are extremely efficient on GPU architectures. This implementation proved to obtain greater performance in the experiments than the existing GPU implementation of the MLSA algorithm included in the existing NNMFPack. Furthermore, the performance of the fHALS algorithm presented increases with the problem size. These results make the GPU fHALS implementation a very powerful tool for those scientists or engineers who use the NMF in high dimensionality problems.

The efficient parallel implementation of the fHALS algorithm in both CPU and GPU architectures is a key contribution of this thesis. On the one hand, the fHALS implementations offer great performance and they were the fastest in the experimental evaluation. On the other hand, the GPU implementation of the fHALS algorithm is a novelty itself.

Closer to the practical application, an efficient parallel implementation of the ASNA algorithm was developed to improve the existing MATLAB implementation. With the reduction of the execution time obtained by this implementation, real-time problems could be dealt with. This shows that High Performance Computing techniques can improve algorithms that are already very good and can open new research lines due to the execution time reductions achieved. The ASNA algorithm is included in the new NNMFPack library as a special case.

11.3 Application of the developed solutions

Even though the main focus of this thesis is on the algorithms and their efficient implementation and not on specific applications, during the development the algorithms implemented were used in some real applications. With these applications the developed implementations were tested under conditions that are closer to the real ones in order to validate the results of these algorithms.

Section 6.5.4 presents an updating model for the problem of On-line Automatic Music Transcription. This application was used to test the algorithms that were developed to solve the updating of the NMF problem. The NMF was used in a an audio spectrogram matrix to determine the pitches that are active in a certain time instant, but the high cost of the factorization does not allow the use of these techniques in real time. By using the updating algorithms to solve the NMF developed in this thesis, the factorization can be updated each time a new sound or group of sounds (represented by new columns added to the data matrix) are received. With the updating model, the new factorizations can be computed fast enough to achieve real-time music transcription. This updating model was

tested in a simulation with a piano song and not only achieved real-time factorizations but also obtained a better transcription.

In Chapter 7, all of the ASNA algorithm implementations developed were tested on a speech separation problem. The original ASNA algorithm was designed to perform speech separation, but it can also be used for other related problems like sound source separation. The implementations presented in Chapter 7 were tested using the original speech separation test case. In that test case, mixture signals (formed by mixing two speech signals from different speakers) are separated using the ASNA algorithm to obtain the original speech signals. The proposed implementations reduced the execution time needed to perform the speech separation by up to 97.8%, allowing the separation to be performed in real-time for most of the signals tested.

Among the data matrices that were used to perform the experimental evaluation of the proposed implementations in Chapter 9, there are two image matrices that were used in a lossy image compression application. The goal of that application was to store the data contained in the images in less space by storing the factorized matrix and reconstructing the image when necessary. These matrices were used to test the influence of real data that may contain hidden data relationships and the effect that latent dependencies have on the proposed implementations.

Finally, the developed algorithms were used to improve the works from other researchers that use the NMF. Some examples of those applications are a percussive-harmonic separation model or a piano denoising model. In the percussive-harmonic separation, the NMF is used to separate the harmonic instruments from the percussive instruments in a musical piece. The result are two audio tracks that contain the separated instruments from the original song. On the other hand, the piano denoising model uses a multiple right hand sides NNLS but with Kullback-Leibler divergence minimization (like the one performed by the ASNA algorithm). The model tries to clean the noise from piano songs by performing a separation step on the spectrogram matrix with a precomputed noise database, obtaining a matrix that should contain the denoised piano music.

Furthermore, contacts have been made with other researchers to use the algorithms developed during this thesis in future applications.

11.4 Publications and conferences

During the development of this thesis several results have been published as journal papers and conference papers. Furthermore, the conference papers have been presented by the author in several conference sessions. In the present section those results are enumerated.

11.4.1 Journal papers

- P. San Juan, Antonio M. Vidal, and V.M. Garcia-Molla. *Updating/Downdating the NonNegative Matrix Factorization.* Journal of Computational and Applied Mathematics. Volume 318. Year 2017. pp. 59–68
- P. San Juan, T. Virtanen, V.M. Garcia-Molla, and A.M.Vidal. *Analysis of an Efficient Parallel Implementation of Active-Set Newton Algorithm*, Journal of Supercomputing. Year 2018. pp. 1–12 doi: https://doi.org/10.1007/s11227-018-2423-5
- P. San Juan, A. M. Vidal, and V.M. Garcia-Molla *Updating the Solution of Nonnegative Least Squares Problems*, Computational Optimization and Applications [UNDER REVIEW]

11.4.2 Conference papers

- P. San Juan et al. *Experiments with the NNMFPACK library: influence of β parameter in the NNMF approximation error*. In: Proceedings of the 15th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2015. 2015, pp. 1023–1034
- P. San Juan , A.M. Vidal, and V.M. Garcia-Molla. *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, pp 1125–1131
- P. San Juan, T. Virtanen, V.M. Garcia-Molla and A.M.Vidal. *Efficient Parallel Implementation of Active-Set Newton Algorithm for Non-Negative Sparse Representations*. In: Proceedings of the 17th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2017. 2017, pp. 1023–1034

11.5 Future work

Following the work performed during the development of this thesis, there are some open research and development topics that can be studied in the future:

- The updating algorithms designed to solve the updating problem for both the NNLS problem and the NMF can be implemented efficiently by following the conventions of the other implementations presented in this thesis. Furthermore, it would be interesting to pack the updating algorithms into a computational library in order to encourage the use of these methods into practical applications. The updating implementations can be integrated in NNMFPack or bundled in a standalone library focused in the updating problems.

- While NNMFPack is a heterogeneous library that supports multiple architectures, their algorithms only use one of the architectures at a time. The implemented algorithms do not take advantage of the different architectures when multiple coprocessors are available in the system, nor do they use the coprocessors and the CPU at the same time. Designing heterogeneous implementations of the algorithms that are present in the library is a very interesting future line of research. That way the library can benefit from all of the available hardware in a system. However, not all of the algorithms can be implemented following a heterogeneous computing paradigm because the performance obtained may be lower than using a single architecture alone.

- Currently, of all the new implemented algorithms during this thesis, only the fHALS algorithm has been implemented for GPU between. A natural extension of the implementation work is to develop GPU implementations of all of the new implemented algorithms. Note that not all of the algorithms will benefit from the GPU architectures and some may have poor performance in those architectures.

- As stated in Section 2.2, there are different target functions that can be minimized when computing the NMF. Since NMFPack has a multiplicative algorithm to compute the NMF by minimizing the β-divergence and there is a version of the HALS algorithm using β-divergence, it is reasonable to include a HALS algorithm implementation. Thus, the development of multicore and GPU implementations of the HALS algorithm for *beta*-divergence would be an interesting addition to the new NNMFPack.

- In order to benefit from the performance increase produced by the use of multiple GPUs on systems with several GPUs installed, a multi-GPU version of the fHALS algorithm could be developed. The cuBLASXt API supports the use of multi-GPU and is a strong candidate for developing a multi-GPU version of the fHALS algorithm.

- Another general continuation of the work done in this thesis is to develop efficient implementations for similar decompositions like the K-SVD or the multidimensional NMF. Those new implementations could then be integrated into the library.

- The NNMFPack library has been developed to work with dense matrices, but it is common to obtain sparse factors with the NMF [11, 99, 44, 35] and even to enforce sparseness in the solution [11][38, Ch. 3.4]. Furthermore, some of the developed algorithms such as the GCD and the ANLS-BPP claim to obtain better results with high dimensionality sparse matrices. An important topic for future development is to develop a library to compute the NMF of sparse matrices. This problem, while being very useful, has extreme complexity and could be the subject of an entire PhD thesis in its own. First of all, to work efficiently with sparse matrices, an efficient sparse matrix storage scheme should be selected. That storage scheme should minimize the memory used to store the matrix, but at the same time it should maximize the performance of the sparse algorithms. Then, the algorithms must be adapted to work with sparse operations with sparse stored matrices.

11.6 Institutional support

Several institutions have supported the development of this PhD thesis by means of institutional or financial support. The institutions and projects that supported this work are listed below:

- **FPU:** "Ayudas para la Formación de Profesorado Universitario" grant funded by "Ministerio de Educación, Cultura y Deporte" from Spain, under the reference "FPU13/03828".

- **Discosound:** "Procesado distribuido y colaborativo de señales sonoras: control activo" project funded by "Ministerio de Economía, Industria y Competitividad" from Spain, under the reference "TEC2012-38142-C04-01".

- **SSPressing:** "Smart Sound Processing for the digital Living" project funded by "Ministerio de Economía, Industria y Competitividad" from Spain, under the reference "TEC2015-67387-C4-1-R (MINECO/FEDER)".

- **PROMETEO II:** "Computación y comunicaciones de altas prestaciones y aplicaciones en ingeniería" project funded by "Generalitat Valenciana" from Spain, under the reference "Proyecto Prometeo II/2014/003"

- **CAPAP-H:** "Red de Computación de Altas Prestaciones sobre Arquitecturas Paralelas Heterogeneas" research network funded by "Ministerio de Economía y Competitividad" from Spain.

- **ARG:** The Audio Research group at the Laboratory of Signal Processing of the Tampere University of Technology(Tampere,Finland) hosted the author during a 6 month research intern-ship where some of the works of this thesis were performed.

- **DSIC:** Most of the work of this thesis was performed in the "Departamento de Sistemas Informáticos y Computación" of the Polytechnic University of Valencia (UPV).

Appendix A

Execution environments

In this appendix the hardware and software configurations of the different machines used in the experiments carried out while developing this thesis are listed.

A.1 Server 1

1. **Hardware:**
   a) **CPU:** 2x Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz
   b) **CPU physical cores:** 2x 12 = 24
   c) **RAM:** 128 GB
2. **Software:**
   a) **Compilers:** gcc, icc.
   b) **Parallelism libraries:** OpenMP, Intel OpenMP
   c) **Generic Mathematical Libraries:** Intel MKL, Atlas (BLAS, LAPACK)
   d) **Specific libraries:** NNMFPACK v2.0

A.2 Server 2

1. **Hardware:**
    a) **CPU:** 2x Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.70GHz
    b) **CPU physical cores:** 2x 13 = 26
    c) **RAM:** 128 GB
2. **Software:**
    a) **S.O:** Ubuntu 14.04
    b) **MATLAB:** MATLAB 2014b
    c) **MATLAB:** MATLAB 2016b

A.3 Server 3

1. **Hardware:**
    a) **CPU:** 2x Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz
    b) **CPU physical cores:** 2x 12 = 24
    c) **RAM:** 128 GB
2. **Software:**
    a) **S.O:** Ubuntu 16.04
    b) **MATLAB:** MATLAB 2016b
    c) **Compilers:** icc v17.0.1.
    d) **Parallelism libraries:** OpenMP, Intel OpenMP
    e) **Generic Mathematical Libraries:** Intel MKL v2017

A.4 Server 4

1. **Hardware:**
    a) **CPU:** 2x Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz
    b) **CPU physical cores:** 2x 20 = 40 (80 cores logical)
    c) **GPU:** 4x NVIDIA Tesla P100-SXM2
    d) **GPU memory:** 4 x 16GB = 64GB
    e) **RAM:** 512 GB
2. **Software:**
    a) **S.O:** Ubuntu 16.04
    b) **Compilers:** icc v17.0.4, nvcc
    c) **Parallelism libraries:** OpenMP, Intel OpenMP
    d) **Generic Mathematical Libraries:** Intel MKL v2017, cuBLAS
    e) **CUDA Runtime:** 9.1

A.5 Workstation

1. **Hardware:**
    a) **CPU:** Intel Core i7-4790 @ 3,6 GHz
    b) **CPU physical cores:** 4
    c) **RAM:** 16 GB
2. **Software:**
    a) **S.O:** Redhat
    b) **MATLAB:** MATLAB 2016b
    c) **Compilers:** icc v17.0.1.

d) **Parallelism libraries:** OpenMP, Intel OpenMP

e) **Generic Mathematical Libraries:** Intel MKL v2017

Bibliography

[1] Charles L Lawson and Richard J Hanson. “Solving least squares problems”. In: vol. 161. SIAM, 1974. Chap. 23 Linear Least Squares with linear inequality constraints (cit. on pp. 1, 2, 11, 14).

[2] Gene H Golub and Charles F Van Loan. *Matrix computations*. Vol. 3. JHU Press, 2012 (cit. on pp. 1, 51, 52, 74).

[3] Stefania Bellavia, Maria Macconi, and Benedetta Morini. “An interior point Newton-like method for non-negative least-squares problems with degenerate solution”. In: *Numerical Linear Algebra with Applications* 13.10 (2006), pp. 825–846 (cit. on p. 2).

[4] Alberto Bemporad. “A quadratic programming algorithm based on nonnegative least squares with applications to embedded model predictive control”. In: *IEEE Transactions on Automatic Control* 61.4 (2016), pp. 1111–1116 (cit. on pp. 2, 14).

[5] Åke Björck. “A direct method for sparse least squares problems with lower and upper bounds”. In: *Numerische Mathematik* 54.1 (1988), pp. 19–32 (cit. on pp. 2, 14).

[6] Rasmus Bro and Sijmen De Jong. “A fast non-negativity-constrained least squares algorithm”. In: *Journal of chemometrics* 11.5 (1997), pp. 393–401 (cit. on pp. 2, 14).

[7] Jason Cantarella and Michael Piatek. “Tsnnls: A solver for large sparse least squares problems with non-negative variables”. In: *arXiv preprint cs/0408029* (2004) (cit. on p. 2).

[8] Joaquim J Júdice and Fernanda M Pires. “A block principal pivoting algorithm for large-scale strictly monotone linear complementarity problems”. In: *Computers & operations research* 21.5 (1994), pp. 587–596 (cit. on pp. 2, 13, 23).

[9] Dongmin Kim, Suvrit Sra, and Inderjit S Dhillon. “A non-monotonic method for large-scale non-negative least squares”. In: *Optimization Methods and Software* 28.5 (2013), pp. 1012–1039 (cit. on p. 2).

[10] Dongmin Kim, Suvrit Sra, and Inderjit S Dhillon. “Fast Newton-type methods for the least squares nonnegative matrix approximation problem”. In: *Proceedings of the 2007 SIAM International Conference on Data Mining*. SIAM. 2007, pp. 343–354 (cit. on p. 2).

[11] Hyunsoo Kim and Haesun Park. “Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis”. In: *Bioinformatics* 23.12 (2007), pp. 1495–1502 (cit. on pp. 2, 151).

[12] Luís F Portugal, Joaquim J Judice, and Luís N Vicente. “A comparison of block pivoting and interior-point algorithms for linear least squares problems with nonnegative variables”. In: *Mathematics of Computation* 63.208 (1994), pp. 625–643 (cit. on pp. 2, 13, 14, 110).

[13] Mark H Van Benthem and Michael R Keenan. “Fast algorithm for the solution of large-scale non-negativity-constrained least squares problems”. In: *Journal of chemometrics* 18.10 (2004), pp. 441–450 (cit. on p. 2).

[14] Pentti Paatero and Unto Tapper. “Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values”. In: *Environmetrics* 5.2 (1994), pp. 111–126 (cit. on pp. 2, 12, 22).

[15] Daniel D Lee and H Sebastian Seung. “Learning the parts of objects by non-negative matrix factorization”. In: *Nature* 401.6755 (1999), p. 788 (cit. on pp. 2, 126).

[16] Zhijian Yuan and Erkki Oja. “Projective nonnegative matrix factorization for image compression and feature extraction”. In: *Scandinavian Conference on Image Analysis*. Springer. 2005, pp. 333–342 (cit. on p. 2).

[17] E. Battenberg, A. Freed, and 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 (cit. on pp. 3, 12).

[18] J. Wang, W. Zhong, and 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, 2006, pp. 513–517 (cit. on pp. 3, 12).

[19] Francisco J Rodriguez-Serrano et al. “Multiple instrument mixtures source separation evaluation using instrument-dependent NMF models”. In: *International Conference on Latent Variable Analysis and Signal Separation*. Springer. 2012, pp. 380–387 (cit. on pp. 3, 12).

[20] Michael W Berry et al. “Algorithms and applications for approximate nonnegative matrix factorization”. In: *Computational statistics & data analysis* 52.1 (2007), pp. 155–173 (cit. on pp. 3, 12, 22).

[21] Devarajan and Karthik. “Nonnegative matrix factorization: an analytical and interpretive tool in computational biology”. In: *PLoS Comput Biol* 4.7 (2008), e1000029 (cit. on pp. 3, 12).

[22] Derry Fitzgerald, Matt Cranitch, and Eugene Coyle. “Shifted non-negative matrix factorisation for sound source separation”. In: *Statistical Signal Processing, 2005 IEEE/SP 13th Workshop on*. IEEE. 2005, pp. 1132–1137 (cit. on pp. 3, 12).

[23] D.D. Lee and H.S. Seung. “Algorithms for non-negative matrix factorization”. In: *Advances in Neural Information Processing Systems* (2001), pp. 556–562 (cit. on pp. 12, 18, 74, 76).

[24] Andrzej Cichocki et al. “Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation”. In: John Wiley & Sons, 2009. Chap. 2 (cit. on p. 12).

[25] P. San Juan Sebastián et al. “Experiments with the NNMFPACK library: influence of β parameter in the NNMF approximation error”. In: *Proceedings of the 15th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2015*. 2015, pp. 1023–1034 (cit. on p. 12).

[26] Emmanuel Vincent, Nancy Bertin, and Roland Badeau. “Adaptive harmonic spectral decomposition for multiple pitch estimation”. In: *IEEE Trans. on Audio, Speech and Language Processing* 18.3 (2010), pp. 528–537 (cit. on pp. 12, 76, 85).

[27] Anis Khlif and Vidhyasaharan Sethu. “Multi Range Non-Negative Matrix Factorization Algorithm for Polyphonic Music Transcription”. In: *The 16th International Society for Music Information Retrieval Conference ISMIR*. 2015 (cit. on pp. 12, 76, 85).

[28] Arnaud Dessein, Arshia Cont, and Guillaume 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*. 2010, pp. 489–494 (cit. on pp. 12, 21, 76, 85).

[29] Derry FitzGerald, Matt Cranitch, and Eugene Coyle. “On the use of the beta divergence for musical source separation”. In: *Irish Signals and Systems Conference* (2009) (cit. on pp. 12, 21, 118).

[30] Antoine Liutkus, Derry Fitzgerald, and Roland Badeau. “Cauchy nonnegative matrix factorization”. In: *Applications of Signal Processing to Audio and Acoustics (WASPAA), 2015 IEEE Workshop on*. IEEE. 2015, pp. 1–5 (cit. on p. 12).

[31] Andrzej Cichocki, Sergio Cruces, and Shun-ichi Amari. “Generalized Alpha-Beta Divergences and Their Application to Robust Nonnegative Matrix Factorization”. In: *Entropy* 13.1 (2011), pp. 134–170. ISSN: 1099-4300. DOI: 10.3390/e13010134 (cit. on p. 12).

[32] Hyunsoo Kim and Haesun Park. “Nonnegative matrix factorization based on alternating nonnegativity constrained least squares and active set method”. In: *SIAM journal on matrix analysis and applications* 30.2 (2008), pp. 713–730 (cit. on pp. 12, 14, 22).

[33] Naiyang Guan et al. “NeNMF: An optimal gradient method for nonnegative matrix factorization”. In: *IEEE Transactions on Signal Processing* 60.6 (2012), pp. 2882–2898 (cit. on p. 12).

[34] Andrzej Cichocki, Rafal Zdunek, and Shun-ichi Amari. “Hierarchical ALS algorithms for nonnegative matrix and 3D tensor factorization”. In: *International Conference on Independent Component Analysis and Signal Separation*. Springer. 2007, pp. 169–176 (cit. on pp. 12, 24).

[35] Jingu Kim and Haesun Park. “Fast nonnegative matrix factorization: An active-set-like method and comparisons”. In: *SIAM Journal on Scientific Computing* 33.6 (2011), pp. 3261–3281. ISSN: 1064-8275 (cit. on pp. 13, 14, 22–24, 26, 74, 125, 151).

[36] N. Díaz-Gracia et al. “NNMFPACK: a versatile approach to an NNMF parallel library”. In: *Proceedings of the 14th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2014*. Vol. II. 2014, pp. 456–465 (cit. on pp. 18, 47).

[37] P. Alonso et al. “Accelerating the computation of nonnegative matrix factorization in multi-core and many-core architectures”. In: *Proceedings of the 13th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2013*. Vol. I. 2013, pp. 71–78 (cit. on pp. 18, 124).

[38] Andrzej Cichocki et al. *Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation*. John Wiley & Sons, 2009 (cit. on pp. 19, 22, 24, 25, 92, 105, 151).

[39] Minami Mihoko and Shinto Eguchi. “Robust blind source separation by beta divergence”. In: *Neural computation* 14.8 (2002), pp. 1859–1886 (cit. on p. 21).

[40] Cédric Févotte, Nancy Bertin, and Jean-Louis Durrieu. “Nonnegative matrix factorization with the Itakura-Saito divergence: With application to music analysis”. In: *Neural computation* 21.3 (2009), pp. 793–830 (cit. on p. 21).

[41] Luigi Grippo and Marco Sciandrone. “On the convergence of the block nonlinear Gauss–Seidel method under convex constraints”. In: *Operations research letters* 26.3 (2000), pp. 127–136 (cit. on p. 22).

[42] Pedro Alonso et al. “Parallel approach to NNMF on multicore architecture”. In: *The Journal of Supercomputing* 70.2 (2014), pp. 564–576 (cit. on p. 23).

[43] Andrzej Cichocki and PHAN Anh-Huy. “Fast local algorithms for large scale nonnegative matrix and tensor factorizations”. In: *IEICE transactions on fundamentals of electronics, communications and computer sciences* 92.3 (2009), pp. 708–721. ISSN: 1745-1337 (cit. on pp. 24, 26, 74, 76, 125).

[44] Cho-Jui Hsieh and Inderjit 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. 2011, pp. 1064–1072 (cit. on pp. 24, 28, 74, 76, 108, 125, 151).

[45] Kejun Huang, Nicholas D Sidiropoulos, and Ananthram Swami. “Non-negative matrix factorization revisited: Uniqueness and algorithm for symmetric decomposition”. In: *IEEE Transactions on Signal Processing* 62.1 (2014), pp. 211–224 (cit. on p. 24).

[46] Bin Cao et al. “Detect and Track Latent Factors with Online Nonnegative Matrix Factorization.” In: *IJCAI*. Vol. 7. 2007, pp. 2689–2694 (cit. on p. 28).

[47] Fei Wang et al. “Efficient document clustering via online nonnegative matrix factorizations”. In: *Proceedings of the 2011 SIAM International Conference on Data Mining*. SIAM. 2011, pp. 908–919 (cit. on pp. 28, 88).

[48] Renbo Zhao, Vincent YF Tan, and Huan Xu. “Online nonnegative matrix factorization with general divergences”. In: *arXiv preprint arXiv:1608.00075* (2016) (cit. on p. 28).

[49] Augustin Lefevre, Francis Bach, and Cédric Févotte. “Online algorithms for nonnegative matrix factorization with the Itakura-Saito divergence”. In: *Applications of Signal Processing to Audio and Acoustics (WASPAA), 2011 IEEE Workshop on*. IEEE. 2011, pp. 313–316 (cit. on p. 28).

[50] Chih-Jen Lin. “Projected gradient methods for nonnegative matrix factorization”. In: *Neural computation* 19.10 (2007), pp. 2756–2779 (cit. on p. 28).

[51] Michael J Flynn. “Very high-speed computing systems”. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909 (cit. on p. 32).

[52] *AMD GPUs for HPC*. URL: https://instinct.radeon.com/en/product/mi/ (cit. on p. 36).

[53] *NVIDIA GPUs for HPC*. URL: https://www.nvidia.com/en-us/data-center/tesla/ (cit. on p. 36).

[54] NVIDIA Corporation. *Whitepaper describing the Volta GPU architecture*. 2017. URL: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (cit. on p. 36).

[55] NVIDIA Corporation. *Whitepaper describing the pascal GPU architecture*. 2016. URL: http://www.nvidia.com/object/pascal-architecture-whitepaper.html (cit. on p. 36).

[56] Taylor IoT Kidd. *Resource Guide for People Investigating the Intel Xeon Phi Co-processor*. 2014. URL: https://software.intel.com/en-us/articles/resource-guide-for-people-investigating-the-intel-xeon-phi-coprocessor (cit. on p. 37).

[57] *TOP 500 HPC list*. URL: https://www.top500.org/lists/top500/ (cit. on p. 38).

[58] *OpenMP v 4.5 specification*. 2015. URL: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (cit. on pp. 40, 111).

[59] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. University of Tennessee, Knoxville, Tennessee (cit. on p. 40).

[60] *OpenMPI implementation of the MPI standard*. URL: https://www.open-mpi.org/ (cit. on p. 40).

[61] *MPICH implementation of the MPI standard*. URL: https://www.mpich.org/ (cit. on p. 40).

[62] *MPI v 3.1 specification*. 2015. URL: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (cit. on p. 41).

[63] *CUDA toolkit documentation*. URL: http://docs.nvidia.com/cuda/index.html (cit. on p. 41).

[64] *CUDA C programming guide*. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (cit. on p. 42).

[65] *MATLAB*. The Mathworks Inc. MATLAB R2014B, Natnick MA. 2014 (cit. on pp. 42, 64, 74).

[66] *MATLAB MEX File documentation*. The Mathworks Inc. MATLAB R2018B, Natnick MA. 2018. URL: https://es.mathworks.com/help/matlab/matlab_external/introducing-mex-files.html (cit. on p. 42).

[67] Chuck L Lawson et al. "Basic linear algebra subprograms for Fortran usage". In: *ACM Transactions on Mathematical Software (TOMS)* 5.3 (1979), pp. 308–323 (cit. on p. 43).

[68] Jack J Dongarra et al. "A set of level 3 basic linear algebra subprograms". In: *ACM Transactions on Mathematical Software (TOMS)* 16.1 (1990), pp. 1–17 (cit. on p. 43).

[69] Edward Anderson et al. "LAPACK: A portable linear algebra library for high-performance computers". In: *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press. 1990, pp. 2–11 (cit. on p. 44).

[70] *Netlib BLAS implementation*. URL: http://www.netlib.org/blas/ (cit. on p. 44).

[71] *Netlib LAPACK implementation*. URL: http://www.netlib.org/lapack/ (cit. on p. 44).

[72] *ATLAS project homepage*. URL: http://math-atlas.sourceforge.net/ (cit. on p. 44).

[73] *GotoBLAS2 legacy package site*. URL: https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2 (cit. on p. 44).

[74] *OpenBLAS homepage*. URL: http://www.openblas.net/ (cit. on p. 44).

[75] *Intel Math Kernel Library (MKL) developer reference (C)*. URL: https://software.intel.com/en-us/mkl-developer-reference-c (cit. on p. 45).

[76] *List of Intel MKL Threaded functions*. URL: https://software.intel.com/en-us/articles/intel-mkl-threaded-functions (cit. on p. 45).

[77] *CULA homepage*. URL: http://www.culatools.com/ (cit. on p. 45).

[78] *cuBLAS developer guide*. URL: http://docs.nvidia.com/cuda/cublas/index.html (cit. on p. 45).

[79] *MAGAMA project homepage*. URL: http://icl.cs.utk.edu/magma/ (cit. on p. 45).

[80] Noelia Díaz-Gracia et al. "Improving NNMFPACK with heterogeneous and efficient kernels for\ beta-divergence metrics". In: *The Journal of Supercomputing* 71.5 (2015), pp. 1846–1856. ISSN: 0920-8542 (cit. on p. 47).

[81] *NNMFPACK library*. http://pirserver.edv.uniovi.es/ Last access 19/06. 2016 (cit. on p. 47).

[82] P. San Juan Sebastián, A.M. Vidal, and V.M. 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 (cit. on pp. 74, 76).

[83] P San Juan, Antonio M Vidal, and VM Garcia-Molla. "Updating/downdating the NonNegative Matrix Factorization". In: *Journal of Computational and Applied Mathematics* 318 (2017), pp. 59–68 (cit. on p. 74).

[84] Andri Mirzal. "A convergent algorithm for orthogonal nonnegative matrix factorization". In: *Journal of Computational and Applied Mathematics* 260 (2014), pp. 149–166. ISSN: 0377-0427. DOI: http://dx.doi.org/10.1016/j.cam.2013.09.022 (cit. on p. 74).

[85] Nicolas Gillis et al. "Nonnegative matrix factorization: Complexity, algorithms and applications". PhD thesis. UCL, 2011. Chap. 4 (cit. on pp. 74, 76).

[86] Nicolas Gillis and François Glineur. "A multilevel approach for nonnegative matrix factorization". In: *Journal of Computational and Applied Mathematics* 236.7 (2012), pp. 1708–1723. ISSN: 0377-0427. DOI: http://dx.doi.org/10.1016/j.cam.2011.10.002 (cit. on p. 74).

[87] Nicolas Gillis and François Glineur. “Accelerated multiplicative updates and hierarchical ALS algorithms for nonnegative matrix factorization”. In: *Neural Computation* 24.4 (2012), pp. 1085–1105 (cit. on pp. 74, 76).

[88] Steve Eddins and Loren Shure. “Matrix indexing in Matlab”. In: *Mathworks Newsletter, http://www.mathworks.com/company/newsletters/articles/matrix-indexing-in-matlab.html(Sep. 13, 2014)* (2001) (cit. on p. 74).

[89] T. Fukuda et al. “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 (cit. on pp. 76, 85, 86).

[90] Emmanouil Benetos et al. “Automatic music transcription: Breaking the glass ceiling”. In: *13th International Conference on Music Information Retrieval (ISMIR)*. 2012 (cit. on p. 85).

[91] J.J. Carabias-Orti et al. “Constrained non-negative sparse coding using learnt instrument templates for realtime music transcription”. In: *Engineering Applications of Artificial Intelligence* 26.7 (2013), pp. 1671–1680. ISSN: 0952-1976. DOI: http://dx.doi.org/10.1016/j.engappai.2013.03.010 (cit. on pp. 85, 94).

[92] S Cherkassky and F. Chopin. *Polonaise Op. 44, Deutsche Grammophon*. 1973 (cit. on p. 85).

[93] *Audio file used in the automatic music transcription experiments*. 2017. URL: http://www.inco2.upv.es/software.php (cit. on p. 85).

[94] F.J. Canadas-Quesada et al. “Constrained non-negative matrix factorization for score-informed piano music restoration”. In: *Digital Signal Processing* 50 (2016), pp. 240–257. ISSN: 1051-2004 (cit. on pp. 85, 126).

[95] Bhiksha Raj and Paris Smaragdis. “Latent variable decomposition of spectrograms for single channel speaker separation”. In: *Applications of Signal Processing to Audio and Acoustics, 2005. IEEE Workshop on*. IEEE. 2005, pp. 17–20 (cit. on p. 91).

[96] Nancy Bertin, Roland Badeau, and Emmanuel Vincent. "Enforcing harmonicity and smoothness in Bayesian non-negative matrix factorization applied to polyphonic music transcription". In: *IEEE Transactions on Audio, Speech, and Language Processing* 18.3 (2010), pp. 538–549 (cit. on p. 91).

[97] Onur Dikmen and Annamaria Mesaros. "Sound event detection using non-negative dictionaries learned from annotated overlapping events". In: *Applications of Signal Processing to Audio and Acoustics (WASPAA), 2013 IEEE Workshop on*. IEEE. 2013, pp. 1–4 (cit. on p. 91).

[98] Charles L Lawson and Richard J Hanson. *Solving least squares problems*. SIAM, 1995 (cit. on p. 91).

[99] Tuomas Virtanen. "Monaural sound source separation by nonnegative matrix factorization with temporal continuity and sparseness criteria". In: *IEEE transactions on audio, speech, and language processing* 15.3 (2007), pp. 1066–1074 (cit. on pp. 92, 151).

[100] Tuomas Virtanen, Jort Florent Gemmeke, and Bhiksha Raj. "Active-set Newton algorithm for overcomplete non-negative representations of audio". In: *IEEE Transactions on Audio, Speech, and Language Processing* 21.11 (2013), pp. 2277–2289 (cit. on pp. 92, 93, 99, 100, 102, 103).

[101] Ali Taylan Cemgil. "Bayesian inference for nonnegative matrix factorisation models". In: *Computational Intelligence and Neuroscience* 2009 (2009) (cit. on p. 92).

[102] Tuomas Virtanen. *Original MATLAB implementation of ASNA algorithm*. 2013. URL: `http://www.cs.tut.fi/~tuomasv/software.html` (cit. on p. 93).

[103] Pablo San Juan et al. "Efficient Parallel Implementation of Active-Set Newton Algorithm for Non-Negative Sparse Representations". In: *Proceedings of the 17th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2017*. 2017, pp. 1023–1034 (cit. on p. 97).

[104] Pablo San Juan. *Efficient implementations of ASNA algorithm*. 2017. URL: `https://gitlab.com/P.SanJuan/ASNA` (cit. on p. 97).

[105] Jort F Gemmeke et al. “Toward a practical implementation of exemplar-based noise robust ASR”. In: *Signal Processing Conference, 2011 19th European*. IEEE. 2011, pp. 1490–1494 (cit. on p. 103).

[106] Francisco Jesus Canadas-Quesada et al. “Percussive/harmonic sound separation by non-negative matrix factorization with smoothness/sparseness constraints”. In: *EURASIP Journal on Audio, Speech, and Music Processing* 2014.1 (2014), p. 26 (cit. on pp. 120, 126).

[107] Pablo San Juan and Jose Ranilla. *Improved NNMFPACK library*. 2018. URL: `https://gitlab.com/SSPressing/NnmfPack` (cit. on p. 140).