ÉCOLE POLYTECHNIQUE DE LOUVAIN

UCL Université catholique de Louvain

Master's Thesis
LINMA2990

Robust Low-Rank Matrix Completion

Léopold Cambier

Dissertation committee:

Prof. Pierre-Antoine Absil	(UCL, advisor)
Dr. Nicolas Boumal	(ENS de Paris, France, advisor)
Prof. Yurii Nesterov	(UCL)

Academic year 2014-2015
Louvain-la-Neuve, June 1, 2015

Acknowledgments

As a first real experience of academic work, this master's thesis was for me a great experience, that I certainly intent to continue in the upcoming years. This has been the confirmation that academia is that in which I want to work in the future.

I sincerely want to thank my two advisors, Pierre-Antoine Absil and Nicolas Boumal. Prof. Absil, thank you for all the insightful discussions, ideas and your unswerving support throughout the year. I also really appreciated your invitation to the RANSO meeting, as well as your proposition to submit a paper: this was a great first experience of academia. Finally, thanks for the careful reading of the paper and of parts of this thesis, and for some great paragraphs suggestions. Nicolas, thanks for all your precious advice for both this work and my future career. You have no idea how useful the few discussions we had were.

I am also grateful to Prof. Yurii Nesterov for the numerous fruitful ideas.

A special thank to my fellow students, Damien Scieur and Mathieu Dath, for accepting to review this document and to give me precious comments and ideas. Last but not least, many thanks to my family and other friends for their help and support.

Contents

Notations

$\mathbf{A}$	A matrix, usually of size $m \times n$ unless stated otherwise.
$\mathbf{A}^\top$	The transpose of the matrix $\mathbf{A}$.
A_{ij}	The (i, j) entry of the matrix $\mathbf{A}$.
$[m]$	The set of integers from 1 to m: $[m] = 1, 2, \ldots, m$.
$\mathcal{M}_r$	The set of $m \times n$ matrices of rank r.
Ω	The set of known or observed entries. $\Omega \subseteq [m] \times [n]$.
$\bar{\Omega}$	The set of unknown entries: $\bar{\Omega} = ([m] \times [n]) \backslash \Omega$.
$\|\cdot\|_{\ell_0}$	The ℓ_0 quasi-norm[1]: $\|\mathbf{X}\|_{\ell_0} = \#\{(i, j) \subseteq [m] \times [n] : X_{ij} \neq 0\}$.
$\|\cdot\|_{\ell_1} = \|\cdot\|_1$	The ℓ_1 norm: $\|\mathbf{X}\|_{\ell_1} = \sum_{i=1,j=1}^{m,n} \|X_{ij}\|$.
$\|\cdot\|_{\ell_2} = \|\cdot\|_F$	The ℓ_2 or Frobenius norm: $\|\mathbf{X}\|_{\ell_2} = \left(\sum_{i=1,j=1}^{m,n} \|X_{ij}\|^2\right)^{1/2}$.
$\|\cdot\|_{\ell_p}$	The ℓ_p norm: $\|\mathbf{X}\|_{\ell_p} = \left(\sum_{i=1,j=1}^{m,n} \|X_{ij}\|^p\right)^{1/p}$.
$\|\cdot\|_{S_p}$	The Schatten p-norm: $\|\mathbf{X}\|_{S_p} = \left(\sum_{i=1}^{\min(m,n)} \sigma_i^p\right)^{1/p}$ where $\sigma_1, \ldots, \sigma_{\min(m,n)}$ are the singular values of $\mathbf{X}$.
$\mathcal{P}_\Omega(\mathbf{X})$	The orthogonal projector of $\mathbf{X}$ onto the space of $m \times n$ matrices with 0 on $\bar{\Omega}$, i.e., $\mathcal{P}_\Omega(X)_{ij} = X_{ij}$ if $(i, j) \in \Omega$ and $\mathcal{P}_\Omega(X)_{ij} = 0$ if $(i, j) \notin \Omega$.
$\mathcal{P}_{\bar{\Omega}}(\mathbf{X})$	The orthogonal projector of $\mathbf{X}$ onto the space of $m \times n$ matrices with 0 on Ω, i.e., $\mathcal{P}_{\bar{\Omega}}(X)_{ij} = X_{ij}$ if $(i, j) \in \bar{\Omega}$ and $\mathcal{P}_{\bar{\Omega}}(X)_{ij} = 0$ if $(i, j) \notin \bar{\Omega}$.

[1]Since it does not respect the condition $\|\lambda \cdot \mathbf{X}\| = |\lambda| \cdot \|\mathbf{X}\| \; \forall \lambda \in \mathbb{R}$.

Introduction

"How to predict missing values in a database ?"

This question has grown in importance for the last ten years.

The growing quantity of information that can be collected and stored by modern computers makes the use of *data mining* tools more and more critical. Among the new challenges we are facing, the problem of inferring information from partial data is one of the most important. This thesis deals with one of the solutions to this problem. The key concept we will use can be summarized as

"Often, data live in low dimensional spaces".

The idea behind this sentence is that, even though this huge amount of data carries a lot of information, one can summarize this information using only some criteria.

For instance, consider the problem of choosing a book to read. There is approximately 130 millions books on Earth. Yet, one does not probably has in mind a preference score for each of the books. On the other hand, there is only a limited amount of different genres, and we all have our preferences for some genres in particular. Thus, we are able to choose a book by, basically, weighting each of the genres present in the book with our preferences. This notion of genre is what is modeled by the *low-rank* hypothesis.

Often, data can be stored as matrices. For instance, when someone read a book, it could rate it by filling an entry in a matrix, where each row correspond to a user and each column to a book. Naturally, this matrix would be really *incomplete* since no one has the time to read every book. The aim of a *Low-Rank Matrix Completion* method is to complete this huge incomplete matrix. Using the low-rank hypothesis, this can be done when observing only a small fraction of the entries.

In this thesis, we thus study the problem of *Low-Rank Matrix Completion*. We first begin to motivate the problem, we formally state our goal and we summarize the existing algorithms. We then explain the basics of Riemannian optimization which is a tool we will use afterwards.

In the first part, we study and improve the RCGMC algorithm (Boumal & Absil, 2015) that aims at solving this problem of low-rank matrix completion. In the first chapter, we study a modification of the original algorithm; in the second chapter, we build a parallel version of both the original and the modified algorithm. Both modifications are efficient: they allow to significantly speed up the resolution of the problem.

The second part of this thesis is dedicated to the problem of *Robust* Low-Rank Matrix Completion. This part is clearly different from the first one. But because it is of significant importance and because we introduce a more innovative idea, we decided to name this thesis

Robust Low-Rank Matrix Completion.

This part is thus dedicated to the problem of low-rank matrix completion where the data contain strong outliers. It uses significantly different tools as the one from the first part that can only handle the case where the data are corrupted by some small additive noise. To solve this problem, we developed three different techniques that we compare to other existing algorithms.

Finally, we end this thesis by applying one of our robust algorithm on some real datasets to show its effectiveness. We then briefly summarize our results and conclude on some possible future considerations.

1 | Low-Rank Matrix Completion

1.1 Motivations

Low-Rank Matrix Completion can be used as a building block to solve many problems in engineering and scientific computing, for instance in image processing, machine learning or data mining. Basically its role is to predict the missing values of a matrix using only a fraction of observed entries and with the assumption that the matrix is low-rank (or at least close to low-rank).

For instance, solving the problem of *recommender systems*, where one wants to predict users' ratings of some item, can be achieved using low-rank matrix completion. In this case, the data are ratings given by users to some items (movies, books, etc.), and our goal is to guess the remaining ratings. The low-rank assumption makes sense, since the "choice space" of each user (i.e., the space of different criteria—for instance the genres, in the case of movies or books—a user takes into account in order to choose an item) is often of low dimension. The most famous example is definitively the Netflix prize (Bennett & Lanning, 2007), where a component of the winning algorithm was a low-rank matrix completion method. In section 6.1, we present an application of one of our methods to the Netflix dataset.

Low-rank matrix completion can also be used in image processing and computer vision. For instance it can reconstruct the paths of 3D points in space from only observations of parts of their trajectory using a fixed camera. This is the problem of *structure from motion* (Kennedy *et al.*, 2014). The low-rank assumption comes from the fact that the matrix containing the coordinates of the points *in the camera plane* at each frame is known to be low-rank. These techniques can also be applied to reconstruct *structured* damaged images (Peng *et al.*, 2012) (where the data are the known pixels and the unknowns of the problem are the remaining ones), but the low-rank assumption does not always makes sense: one has to be careful to use the right "low-rank image". In section 6.2 we apply one of our algorithms to successfully reconstruct highly damaged structured images.

Finally low-rank matrix completion can be used in the *sensor network localization* problem where one wants to infer sensors positions in space using only a small fraction of pairwise distances between some points (Drineas *et al.*, 2006; So & Ye, 2007; Oh *et al.*, 2010). This may be useful for instance when one has a large amount of low-power wireless routers in a 2 or 3D space that are only able to compute their distances to a few neighbors, and one wants to create a map of their positions.

1.2 The Problem

The Low Rank Matrix Completion (LRMC) problem is the problem of recovering an unknown low-rank matrix by knowing only a (usually very) small subset of its entries, possibly corrupted by noise or containing outliers.

Formally, in the perfect noiseless case, the problem could be stated as recovering a matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$, observed only on a subset Ω of its entries, by finding a matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$ such that

$$X_{ij} = M_{ij} \quad \forall (i,j) \in \Omega$$

where $\mathbf{X} \in \mathbb{R}^{m \times n}$ is low-rank[1]. We emphasize the fact that $\mathbf{M}$ is given only on Ω: M_{ij} is known for $(i,j) \in \Omega$ but not for $(i,j) \notin \Omega$.

The equality constraint

$$X_{ij} = M_{ij} \quad \forall (i,j) \in \Omega$$

is often also expressed as

$$\mathcal{P}_\Omega(\mathbf{X}) = \mathcal{P}_\Omega(\mathbf{M})$$

where $\mathcal{P}_\Omega : \mathbb{R}^{m \times n} \to \mathbb{R}^{m \times n}$ is such that

$$\mathcal{P}_\Omega(X)_{ij} = \begin{cases} X_{ij} & \text{if } (i,j) \in \Omega, \\ 0 & \text{otherwise.} \end{cases}$$

1.3 Previous Work

To get the matrix with the lowest possible rank, one would want to solve the following program

$$\begin{aligned} \min_{\mathbf{X} \in \mathbb{R}^{m \times n}} \quad & \operatorname{rank}(\mathbf{X}) \\ \text{subject to} \quad & \mathcal{P}_\Omega(\mathbf{X}) = \mathcal{P}_\Omega(\mathbf{M}). \end{aligned} \tag{1.1}$$

This problem, however, is intractable in practice, because it is both NP-hard and the only known algorithm to solve it has time doubly exponential in the size of $\mathbf{X}$ (Chistov & Grigor'ev, 1984).

The now well known method (Candès & Recht, 2009) to solve such a problem is to relax the objective function by the nuclear norm:

$$\|\mathbf{X}\|_* = \sum_{i=1}^{\min(m,n)} \sigma_i(\mathbf{X}).$$

This leads to the following *convex* optimization problem

$$\begin{aligned} \min_{\mathbf{X} \in \mathbb{R}^{m \times n}} \quad & \|\mathbf{X}\|_* \\ \text{subject to} \quad & \mathcal{P}_\Omega(\mathbf{X}) = \mathcal{P}_\Omega(\mathbf{M}). \end{aligned}$$

The advantage with this formulation is that it is convex and can be thoroughly analyzed. Candès & Recht (2009) proved that under a few suitable assumptions, solving this convex program

[1] In our case, the rank r of $\mathbf{X}$ will be fixed in advance; in some other cases, the rank is not fixed in advance, and one only aims for the lowest-rank matrix $\mathbf{X}$.

leads to the solution of the original problem (1.1). Basically, these assumptions require the number of observed entries (i.e., the size of Ω) to be high enough (at least $\mathcal{O}\left(n^{5/4} r \log n\right)$ for an $n \times n$ matrix) and sampled uniformly at random. They also require the low-rank underlying matrix to come from the "random orthogonal model", meaning it needs to have "spread enough" singular vectors. More details can be found in (Candès & Recht, 2009).

This work has been the foundation of the Low-Rank Matrix Completion field, that has drawn much attention in the scientific community for the past 6 years. Starting in 2009, numerous algorithms have been developed to solve the matrix completion problem, in various settings.

An natural extension of such a theory would be the problem of *noisy* low-rank matrix completion where $\mathbf{M}$ is the sum of a low-rank matrix and a small Gaussian perturbation. Hence, $\mathbf{M}$ is not (at all) low-rank, but close to be low-rank. In this case it has been proved (Candes & Plan, 2010) that, if the few observed entries are corrupted by a small Gaussian noise, one can hope to recover the underlying low-rank matrix with an error proportional to the noise level.

To solve such a problem, there exist globally four main types of algorithms, relying on very different techniques.

First, many algorithms rely on the *nuclear norm* heuristic, as explained before. For instance, a natural formulation of the noisy low-rank matrix completion problem is (Candes & Plan, 2010)

$$\begin{aligned} \min_{\mathbf{X} \in \mathbb{R}^{m \times n}} \quad & \|\mathbf{X}\|_* \\ \text{subject to} \quad & \|\mathcal{P}_\Omega(\mathbf{X} - \mathbf{M})\|_{\ell_2} \leq \delta \end{aligned}$$

for some value[2] of δ.

Other methods handle the low-rank "constraint" differently. For instance, assume the target rank of $\mathbf{X}$ is known in advance (which actually makes sense in a lot of applications, especially in computer-vision). Hence if $\mathbf{X}$ is of rank (at most) r, it can be written as the product of two thin rectangular matrices, $\mathbf{U} \in \mathbb{R}^{m \times r}$ and $\mathbf{V} \in \mathbb{R}^{r \times n}$, such that $\mathbf{X} = \mathbf{U} \cdot \mathbf{V}$. It is already useful to note that this representation is not unique: for all $\mathbf{M} \in \mathbb{R}^{r \times r}$ invertible, $\mathbf{X} = \mathbf{U} \cdot \mathbf{V} = \mathbf{U} \cdot \mathbf{M} \cdot \mathbf{M}^{-1} \cdot \mathbf{V} = (\mathbf{U} \cdot \mathbf{M}) \cdot (\mathbf{M}^{-1} \cdot \mathbf{V}) = \tilde{\mathbf{U}} \cdot \tilde{\mathbf{V}}$, and $\tilde{\mathbf{U}} \cdot \tilde{\mathbf{V}}$ is another factorization of $\mathbf{X}$. Using this formulation, the optimization problem can be stated as finding $\mathbf{U}$ and $\mathbf{V}$ such that their product better fits $\mathbf{M}$ on the mask Ω:

$$\min_{\mathbf{U},\mathbf{V}} \|\mathcal{P}_\Omega(\mathbf{U}\mathbf{V} - \mathbf{M})\|_{\ell_2}.$$

A way to solve such a problem is to alternatively fix one factor and to optimize with respect to the other. This leads to intermediate easy least-squares problems that sometimes even have closed-form solutions. Using a slightly more complex nonlinear successive over-relaxation algorithm, Wen *et al.* (2012) proposed LMaFit, an algorithm to solve the low-rank matrix completion problem based on alternating minimization.

Another way to handle the low-rank constraint is to "notice" that the set $\mathcal{M}_r$ of matrices of rank r,

$$\mathcal{M}_r = \{\mathbf{X} \in \mathbb{R}^{m \times n} : \operatorname{rank}(\mathbf{X}) = r\},$$

[2]We use the notation $\|\cdot\|_{\ell_2}$ for the Frobenius norm $\|\cdot\|_F$ to emphasize the difference with the ℓ_1 norm to be used in the second part of this document.

is a smooth Riemannian manifold of dimension $r(m+n-r)$ (Lee, 2003). Then, using now well developed optimization methods on manifolds (see (Absil *et al.*, 2008) for a comprehensive analysis of such methods), we can solve smooth optimization problems using the search space $\mathcal{M}_r$. For instance, this problem

$$\min_{\mathbf{X}\in\mathcal{M}_r} \|\mathcal{P}_\Omega(\mathbf{X}-\mathbf{M})\|_{\ell_2}$$

can be solved using techniques very close to the well known conjugate gradient method. This led to LRGeomCG (Vandereycken, 2013).

Finally some algorithms rely on the fact that the data explicitly live in a low dimensional linear subspace. Taking account of this, we can formulate the low-rank matrix completion problem as the recovery of this linear r-dimensional subspace $\mathcal{U}$ (represented using an orthogonal matrix $\mathbf{U}$ which columns span $\mathcal{U}$, i.e., $\mathcal{U} = \text{col}(\mathbf{U})$, and slightly abusing notations):

$$\min_{\mathbf{U}:\mathbf{U}^\top\mathbf{U}=\mathbf{I}_r} \|\mathcal{P}_\Omega(\mathbf{U}\mathbf{W}_\mathbf{U}-\mathbf{M})\|_{\ell_2}$$

where $\mathbf{W}_\mathbf{U}$ is the solution of the following least-square problem

$$\mathbf{W}_\mathbf{U} = \text{argmin}_{\mathbf{W}\in\mathbb{R}^{r\times n}} \|\mathcal{P}_\Omega(\mathbf{U}\mathbf{W}-\mathbf{M})\|_{\ell_2}.$$

Such idea led to the RTRMC (Riemannian Trust-Region Matrix Completion) and RCGMC (Riemannian Conjugate Gradient Matrix Completion) algorithms (see (Boumal & Absil, 2015) and more details in section 3.1), which also include a regularization term to make the solution of the least-square problem unique. Another algorithm, GROUSE (Balzano *et al.*, 2010), relies on the same idea but is designed to perform *online* matrix completion, i.e., when we observe one column of $\mathbf{M}$ at a time.

It is interesting to note that in the first formulation, it is the rank (or some relaxation of the rank function) that is minimized, while the constraint enforces a certain level of "fit" between the data and the recovered matrix. On the other hand, the other ones basically do the opposite: the rank is fixed *a priori*, and it is the level of "fit" between $\mathbf{X}$ and $\mathbf{M}$ that is minimized. In this work, we will focus on this second formulation where the rank is fixed *a priori*.

2 | Essential Tools of Riemannian Optimization

This chapter describes the essential tools of Riemannian optimization. The goal is *not* to have formal and accurate formulations, but rather to get some insight about the main differences between Riemannian optimization and classical Euclidian optimization and to introduce the tools needed to extend the algorithms from $\mathbb{R}^n$ to arbitrary manifolds. Most of the concepts in this chapter are precisely defined in (Absil *et al.*, 2008). To keep things simple, we will only present how to generalize the steepest descent and the (non-linear) conjugate gradient algorithms.

Our topic of interest is the optimization of a (one or twice) differentiable function $f : \mathcal{M} \to \mathbb{R}$ defined from a manifold $\mathcal{M}$ to $\mathbb{R}$:

$$\min_{x \in \mathcal{M}} f(x).$$

Optimization on manifolds can simply be seen as the extension of classical Euclidian optimization where the search space is a general manifold instead of simply $\mathbb{R}^n$.

2.1 The Manifold Structure

Informally, a d-dimensional manifold $\mathcal{M}$ can be seen as a set that can be identified (through some bijection) to $\mathbb{R}^d$. The idea is that, locally, the manifold can be identified with $\mathbb{R}^d$.

Formally, we need to define what is called charts and atlases first. Then, the general definition will follow.

Definition 1 (Chart)**.** *A d-dimensional* chart φ *from $\mathcal{U} \subseteq \mathcal{M}$ to $\mathbb{R}^d$ is a bijection from $\mathcal{U}$ to $\mathbb{R}^d$. It is denoted by $(\mathcal{U}, \varphi)$*

Intuitively, a chart locally assign to a point of $\mathcal{U}$ some coordinates in $\mathbb{R}^d$.

The problem is that, to have suitable algorithms, we need to have charts that "match" correctly, i.e., that smoothly overlap at their boundaries. To do so, we define an atlas, a collection of charts that overlap smoothly:

Definition 2 (Atlas)**.** *An* atlas $\mathcal{A}$ *of $\mathcal{M}$ into $\mathbb{R}^d$ is a collection of charts $(\mathcal{U}_\alpha, \varphi_\alpha)$ such that*

- $\bigcup_\alpha \mathcal{U}_\alpha = \mathcal{M}$
- *for any pair α, β with $\mathcal{U}_\alpha \cap \mathcal{U}_\beta \neq \emptyset$, the sets $\varphi_\alpha(\mathcal{U}_\alpha \cap \mathcal{U}_\beta)$ and $\varphi_\beta(\mathcal{U}_\alpha \cap \mathcal{U}_\beta)$ are open sets in $\mathbb{R}^d$ and the change of coordinates $\varphi_\beta \circ \varphi_\alpha^{-1}$ is smooth.*

The last difficult point is that a given set $\mathcal{M}$ can have many atlases. To circumvent this difficulty, we define $\mathcal{A}^+$ as the maximal atlas of $\mathcal{M}$ such that it contains all possible charts of $\mathcal{M}$.

We now have the required tools to define a manifold.
Definition 3 (Manifold)**.** *A d-dimensional manifold is a couple* $(\mathcal{M}, \mathcal{A}^+)$ *where* $\mathcal{M}$ *is a set and* $\mathcal{A}^+$ *is a maximum atlas from* $\mathcal{M}$ *to* $\mathbb{R}^d$*.*

These definitions allow to formalize the idea that a manifold $(\mathcal{M}, \mathcal{A})$ (or simply a manifold $\mathcal{M}$ if the atlas is obvious from the context) is locally equivalent to $\mathbb{R}^d$.

2.1.1 Embedded and Quotient Manifolds

There exist two very different manifold structures of interest, namely embedded and quotient manifolds. Both are manifolds in the sense of the previous section, but they are conceptually quite different.

Embedded manifolds are manifolds that can be easily described by constraints in the ambient $\mathbb{R}^{m\times n}$ space. For instance, the sphere

$$\mathbb{S}_{n-1} = \{\mathbf{X} \in \mathbb{R}^n : \|\mathbf{X}\|_2 = 1\}$$

is an embedded manifold of $\mathbb{R}^n$ (Absil *et al.*, 2008), as it is simply defined by constraints in the $\mathbb{R}^n$ ambient space (and it matches the previous definition of a manifold). The set of low-rank matrices

$$\mathcal{M}_r = \{\mathbf{X} \in \mathbb{R}^{m\times n} : \text{rank}(\mathbf{X}) = r\}$$

can also be seen as an embedded manifold of $\mathbb{R}^{m\times n}$ (Lee, 2003). Theses manifolds are easy to handle in the sense that they are easy to visualize as curved shapes in the ambient space.

On the other hand, quotient manifolds are manifolds described by the mean of equivalence classes. A point on the manifold will be a class, and we will represent it in the computer by the mean of one element of the class. Even though they are quite useful (the RTRMC algorithm of Boumal & Absil (2011) is based on optimization on the Grassman manifold which is a quotient manifold) this is out of the scope of this very basic introduction.

2.2 Steepest Descent Algorithm

We first begin to introduce the tools needed to generalize the steepest descent algorithm, from its Euclidian version

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k)$$

to the Riemannian one. To do so, we need the notions of *tangent space*, *gradient* and *retraction.*

2.2.1 Tangent Space and Inner Product

A way to visualize smooth Riemannian manifolds is to view them as curved shapes in the ambient space. Because they are smooth, each point $x \in \mathcal{M}$ can be associated to a tangent plane $T_x\mathcal{M}$ which is a local first-order approximation of the manifold. This tangent plane or space is a vector space.

Given a point $x \in \mathcal{M}$ and a tangent space $T_x\mathcal{M}$, one can define an inner product between two vectors $\xi_x \in T_x\mathcal{M}$ and $\eta_x \in T_x\mathcal{M}$ (where the subscript x denotes the *foot* of the tangent vectors):

$$\langle \cdot, \cdot \rangle_x : T_x\mathcal{M} \times T_x\mathcal{M} \to \mathbb{R} : (\xi_x, \eta_x) \to \langle \xi_x, \eta_x \rangle_x$$

with the usual properties of the inner product. Note that, in general, this inner product depends on the point x, since the tangent space depends on x as well. The norm of a vector $\xi_x \in T_x\mathcal{M}$ is naturally defined as

$$\|\xi_x\|_x = \langle \xi_x, \xi_x \rangle_x.$$

These definitions allow us to define a metric on a manifold and the essential tools of optimization start to appear: scalar product and norms will allow us to control the behavior of algorithms and to quantify their convergence.

2.2.2 Gradient

Now, we can define the most essential tool of Riemannian optimization, the gradient. The gradient of a function $f : \mathcal{M} \to \mathbb{R}$ will be used to build an equivalent of the steepest descent algorithm.

When working with embedded manifolds of $\mathbb{R}^{m \times n}$, the computation of the gradient is quite easy: to compute the gradient of f at x, $\operatorname{grad} f(x)$, we simply need to compute the classical (Euclidian) gradient of the function $\bar{f}$ (the extension of f to $\mathbb{R}^{m \times n}$), at x, $\nabla_x \bar{f}(x)$, and then project it onto the tangent plane at x:

$$\operatorname{grad} f(x) = P_x \nabla_x \bar{f}(x),$$

where P_x is the orthogonal projector from $\mathbb{R}^{m \times n}$ onto $T_x\mathcal{M}$.

The gradient is a tangent vector at x and belongs to $T_x\mathcal{M}$. Note that this gradient, like the Euclidian gradient, is the steepest *ascent* direction (and his opposite is the steepest descent direction).

2.2.3 Retraction

To decrease a cost function, we need to be able to move in the direction of the gradient, but staying on the manifold. In the Euclidian space, to move into the direction $-\operatorname{grad} f(x) = -\nabla_x f(x)$, we would simply do something like

$$x_{k+1} = x_k - \alpha \operatorname{grad} f(x_k)$$

for a given $\alpha > 0$. But in general this is not well defined on a manifold, since a manifold does not have a vector space structure in general (simply consider the sphere for instance: following a tangent vector ξ_x at x using the "+" gives a point $x + \xi_x$ which does not belong to the sphere).

To address this issue, the notion of *retraction* has been introduced. The idea is to have a cheap way of moving on the manifold in the direction of a tangent vector with just enough properties so that the usual convergence results of the algorithms in $\mathbb{R}^{m \times n}$ will be preserved on $\mathcal{M}$.

Definition 4 (Retraction)**.** *A retraction on $\mathcal{M}$ is a smooth function*

$$R_x : T_x\mathcal{M} \to \mathcal{M}$$

such that

- *$R_x(0_x) = x$, where 0_x is the zero vector of $T_x\mathcal{M}$;*
- *$DR_x(0_x) = I_{T_xM}$, where D denotes the directional derivative and I_{T_xM} the identity mapping from $T_x\mathcal{M}$ to $T_x\mathcal{M}$.*

The idea is that the retraction from x in a direction $\xi_x \in \mathcal{T}_x\mathcal{M}$, denoted as $R_x(\xi_x)$, is a point on the manifold $\mathcal{M}$ obtained by coming from x in the direction ξ_x.

2.2.4 Steepest Descent

Now that we have this tool, we can define the equivalent of the steepest descent algorithm on a manifold $\mathcal{M}$. Basically, the idea is to start from a point $x_0 \in \mathcal{M}$ and then to successively update x_k by the following rule

$$x_{k+1} = R_{x_k}(-\alpha_k \operatorname{grad} f(x_k)).$$

We can see that everything is well defined: the gradient $\operatorname{grad} f(x_k)$ is computable, $-\alpha_k \operatorname{grad} f(x_k)$ is well defined since $\operatorname{grad} f(x_k) \in T_x\mathcal{M}$ (which is a vector space), and $R_{x_k}(-\alpha_k \operatorname{grad} f(x_k))$ allows us to "move" in the direction of the gradient, starting at x_k and staying on the manifold.

If we are working with a manifold which is identified with the full Euclidian space, that is $\mathcal{M} = \mathbb{R}^{m\times n}$, the retraction is simply the sum $R_x(\xi_x) = x + \xi_x$ and the steepest descent algorithm translates into $x_{k+1} = x_k - \alpha_k \operatorname{grad} f(x_k)$ which is exactly the steepest descent algorithm in $\mathbb{R}^n$.

2.3 Conjugate Gradient Algorithm

Now that the steepest descent algorithm has been generalized to manifolds, we would like to generalize the conjugate gradient algorithm. In $\mathbb{R}^{m\times n}$, the iteration scheme reads

$$s_k = -\operatorname{grad} f(x_k) + \beta_k s_{k-1},$$

$$x_{k+1} = x_k - \alpha_k s_k,$$

with $\operatorname{grad} f(x_k) = \nabla f(x_k)$ and for some β_k and $\alpha_k \in \mathbb{R}$.

To generalize this to an abstract manifold $\mathcal{M}$, the problem lies in "$-\operatorname{grad} f(x_k) + \beta_k s_{k-1}$". Indeed, $-\operatorname{grad} f(x_k)$ is a vector belonging to the tangent space at x_k while $\beta_k s_{k-1}$ belongs to the tangent space at x_{k-1}; because of this, the sum "+" does not make sense in general (since these are two different vector spaces). To resolve this issue, we need to introduce the notion of *vector transport*.

2.3.1 Vector Transport

The role of the vector transport is to be able to deal with tangent vectors in different tangent spaces, i.e., tangent spaces at different points on the manifold.

Definition 5 (Vector Transport)**.** *A vector transport on a manifold $\mathcal{M}$ is a smooth function*

$$\mathcal{T} : T_x\mathcal{M} \times T_x\mathcal{M} \to T\mathcal{M} : (\eta_x, \xi_x) \to \mathcal{T}_{\eta_x}(\xi_x)$$

satisfying a few properties:

- *$\mathcal{T}_{\eta_x}(\xi_x)$ is a tangent vector in $T_{R_x(\eta_x)}\mathcal{M}$ (meaning that the transport should be consistent with the retraction);*
- *$\mathcal{T}_{0_x}(\xi_x) = \xi_x$;*
- *$\mathcal{T}_{\eta_x}(a\xi_x + b\zeta_x) = a\mathcal{T}_{\eta_x}(\xi_x) + b\mathcal{T}_{\eta_x}(\zeta_x)$.*

This definition should be understood to be some "parallel vector transport", i.e., ξ_x is transported in parallel at the tangent space of the point $R_x(\eta_x)$. This definition allows us to work with tangent vectors at different points on the manifold.

2.3.2 Conjugate Gradient

Now, thanks to the tools of Riemannian optimization—like retractions and vector transports—it is possible to generalize the conjugate gradient algorithm scheme to an abstract manifold:

$$\eta_k = -\operatorname{grad} f(x_k) + \beta_k \mathcal{T}_{\alpha_{k-1}\eta_{k-1}}(\eta_{k-1}),$$

$$x_{k+1} = R_{x_k}(-\alpha_k \eta_k),$$

where we use the vector transport to give a meaning to "$-\operatorname{grad} f(x_k) + \beta_k \eta_{k-1}$". Otherwise, $\operatorname{grad} f(x_k)$ and $\beta_k \eta_{k-1}$ would belong to different vector spaces.

Again, should we work with $\mathcal{M} = \mathbb{R}^{m\times n}$, the retraction is the sum ($R_x(\xi_x) = x + \xi_x$), the vector transport is the identity ($\mathcal{T}_{\eta_x}(\xi_x) = \xi_x$), and this algorithm will give us the exact classical (Euclidian) conjugate gradient algorithm (with $\eta_k = s_k$).

Part I

Improvements of the RCGMC Algorithm

3 | A Dai-Yuan Conjugate Gradient Algorithm for RCGMC

This chapter is dedicated to the study of a particular type of conjugate gradient algorithm which has the remarkable property of being globally convergent (towards a critical point). Unlike other conjugate gradient algorithms, it only uses the *weak* Wolfe conditions, as opposed—for instance—to the Fletcher-Reeves conjugate gradient algorithm which is globally convergent using the *strong* Wolfe conditions (Sato, 2014). This algorithm is a Riemannian version of the Dai-Yuan conjugate gradient algorithm (Dai & Yuan, 1999) but extended by Sato to the Riemannian case (Sato, 2014).

In this chapter, we first briefly state the RCGMC algorithm (Boumal & Absil, 2015), we then explain this particular conjugate gradient and then perform some numerical comparisons.

3.1 RCGMC

Before getting into the specifics of the new conjugate gradient algorithm, let us begin by introducing RCGMC.

RCGMC (and RTRMC, see (Boumal & Absil, 2015)) is a low-rank matrix completion algorithm developed by Boumal & Absil. In this algorithm, we look for the low-rank matrix $\mathbf{X}$ that minimizes the following cost function

$$f(\mathbf{X}) = \sum_{(i,j)\in\Omega} C_{ij}^2 (X_{ij} - M_{ij}) + \lambda^2 \sum_{(i,j)\in\bar{\Omega}} X_{ij}^2 \tag{3.1}$$

where $\mathbf{X} \in \mathcal{M}_r$, for a given value of λ and $\mathbf{C}$. Typically, we will have $C_{ij} = 1$ for all $(i,j) \in \Omega$ and 0 otherwise, while λ will be some small value.

The trick relies on reformulating this problem as an optimization problem on the Grassman manifold $\mathrm{Gr}(m,r)$ of the $r-$dimensional linear subspaces of $\mathbb{R}^m$. To begin with, let us define $\mathbf{W_U}$ as

$$\mathbf{W_U} = \mathrm{argmin}_{\mathbf{W}\in\mathbb{R}^{r\times n}} \sum_{(i,j)\in\Omega} C_{ij}^2 ((UW)_{ij} - M_{ij}) + \lambda^2 \sum_{(i,j)\in\bar{\Omega}} (UW)_{ij}^2.$$

It can be shown that for a given $\mathbf{U}$ and if $\lambda \neq 0$, the solution to this problem is unique, can be found efficiently and smoothly depends on $\mathbf{U}$. Hence, the problem can be replaced by the problem of finding $\mathbf{U}$ that minimizes

$$f(\mathbf{U}) = \sum_{(i,j)\in\Omega} C_{ij}^2 ((UW_U)_{ij} - M_{ij}) + \lambda^2 \sum_{(i,j)\in\bar{\Omega}} (UW_U)_{ij}^2. \tag{3.2}$$

Now, it is useful to notice that every matrix $\mathbf{U}'$ that shares the same *column space* as $\mathbf{U}$ gives the same cost. Indeed, since only $\mathbf{UW}$ products appear in equation (3.2), every $\mathbf{U}'$ such that it exists $\mathbf{V} \in \mathbb{R}^{r\times r}$ invertible so that $\mathbf{U}' = \mathbf{UV}$ leads to the same cost, with $\mathbf{W_U}$ replaced by $\mathbf{W_{U'}} = \mathbf{W_{UV}} = \mathbf{V}^{-1}\mathbf{W_U}$. So the cost only really depends on col($\mathbf{U}$), the column space of $\mathbf{U}$.

This is by taking into account this invariance that the RTRMC/RCGMC algorithm eventually minimizes a function defined over the set of r-dimensional linear subspaces (the "col($\mathbf{U}$)") included in $\mathbb{R}^m$, i.e., the Grassman manifold $\mathrm{Gr}(m, r)$. A point on this manifold (i.e., a subspace, say $\mathcal{U}$) can be easily represented by a $m \times r$ matrix ($\mathbf{U}$) such that $\mathcal{U} = \mathrm{col}(\mathbf{U})$. Numerically, it is suitable to have $\mathbf{U}$ orthogonal (such that $\mathbf{U}^\top\mathbf{U} = \mathbf{I}_r$). We then eventually minimize

$$f : \mathrm{Gr}(m, r) \to \mathbb{R} : \mathrm{col}(\mathbf{U}) \to \sum_{(i,j)\in\Omega} C_{ij}^2((UW_U)_{ij} - M_{ij}) + \lambda^2 \sum_{(i,j)\in\bar{\Omega}} (UW_U)_{ij}^2.$$

This can be done efficiently using the now well-developed theory of Riemannian optimization (see (Absil *et al.*, 2008) for instance) since it can be shown that $\mathrm{Gr}(m, n)$ is a smooth Riemannian (quotient) manifold. RTRMC minimizes this function using a trust-region algorithm while RCGMC uses a conjugate gradient method. Both can be preconditioned to deal with ill-conditioned problems, and the algorithm is particularly efficient when the matrix $\mathbf{M}$ is rectangular since the search-space $\mathrm{Gr}(m, n)$ is of dimension $r(m-r) \approx mr$ whereas the dimension of $\mathcal{M}_r$ is $r(m + n - r) \approx r(m + n)$. So the dimension of the search space is proportional to $\min(m, n)$ (if $m > n$, we can simply transpose the matrix), while most algorithms have a search-space dimension proportional to $m + n$.

3.2 Conjugate Gradient

As explained in chapter 2, the Riemannian conjugate gradient algorithm updates x_k in the following way

$$\eta_k = -\operatorname{grad} f(x_k) + \beta_k \mathcal{T}_{\alpha_{k-1}\eta_{k-1}}(\eta_{k-1})$$

$$x_{k+1} = R_{x_k}(\alpha_k\eta_k)$$

where $\mathcal{T}$ is a vector transport (i.e., a mapping from $T_x\mathcal{M}$ to $T_y\mathcal{M}$ for $x, y \in \mathcal{M}$, see section 2.3.1) and R_x a retraction (see section 2.2.3) where β_k is directly computable and α_k is obtained from a linesearch algorithm.

There exist multiple options for the β_k parameter and the linesearch algorithm; this chapter deals with one particular choice and its consequences on the RCGMC algorithm.

3.3 Dai-Yuan Conjugate Gradient

This algorithm uses one ingredient of importance, a *scaled* vector transport, defined as

$$\mathcal{T}_\eta^0(\xi) = \frac{\|\xi\|_x}{\|\mathcal{T}_\eta(\xi)\|_{R_x(\xi)}}\mathcal{T}_\eta(\xi),$$

where $\mathcal{T}$ is the vector transport associated with the retraction R to be used in the algorithm. Typically, the vector transport will derive from the retraction R directly by

$$\mathcal{T}_\eta(\xi) = DR_x(\eta)[\xi]$$

where x is the foot of η and ξ.

Using this ingredient, we are able to prove the global convergence of the conjugate gradient algorithm (Sato, 2014) if the parameter β_k is computed such as

$$\beta_{k+1} = \frac{\| \operatorname{grad} f(x_{k+1})\|^2_{x_{k+1}}}{\langle \operatorname{grad} f(x_{k+1}), \mathcal{T}^{(k)}_{\alpha_k\eta_k}(\eta_k)\rangle_{x_{k+1}} - \langle \operatorname{grad} f(x_k), \eta_k\rangle_{x_k}}$$

where

$$\mathcal{T}^{(k)}_{\alpha_k\eta_k}(\eta_k) = \begin{cases} \mathcal{T}_{\alpha_k\eta_k}(\eta_k) & \text{if } \|\mathcal{T}_{\alpha_k\eta_k}(\eta_k)\|_{x_{k+1}} \leq \|\eta_k\|_{x_k} \\ \mathcal{T}^0_{\alpha_k\eta_k}(\eta_k) & \text{otherwise} \end{cases}$$

and where the search direction is computed using this vector transport:

$$\eta_{k+1} = -\operatorname{grad} f(x_{k+1}) + \beta_{k+1}\mathcal{T}^{(k)}_{\alpha_k\eta_k}(\eta_k).$$

This is one of the Riemannian versions of the β-rule proposed by Dai & Yuan (1999) but only this version is known to have a global convergence property in the Riemannian setup. Regarding the function f, a sufficient condition (Sato & Iwai, 2013) to ensure global convergence is that f is smooth and defined on a compact set. In our problem, f is indeed smooth (see (Boumal & Absil, 2015)) and the Grassman manifold is compact (Milnor & Stasheff, 1974). Hence, global convergence (towards a critical point) is guaranteed.

Using such definition, the conjugate gradient algorithm can be applied (see for instance Absil *et al.* 2008) and one needs to only meet the *weak* Wolfe conditions for the linesearch algorithm to guarantee global convergence.

Note that this algorithm can also be preconditioned; see appendix A for further details.

3.4 Line-Search Algorithm for the Weak Wolfe Conditions

The weak Wolfe conditions need to be met when the linesearch is performed, meaning that the α_k such that

$$\alpha_k \approx \min_{\alpha>0} f(R_{x_k}(\alpha\eta_k))$$

needs to satisfy the following two properties

$$f(R_{x_k}(\alpha_k\eta_k)) \leq f(x_k) + c_1\alpha_k\langle \operatorname{grad} f(x_k), \eta_k\rangle_{x_k}, \tag{3.3}$$

$$\langle \operatorname{grad} f(R_{x_k}(\alpha_k\eta_k)), DR_{x_k}(\alpha_k\eta_k)[\eta_k]\rangle_{R_{x_k}(\alpha_k\eta_k)} \geq c_2\langle \operatorname{grad} f(x_k), \eta_k\rangle_{x_k}, \tag{3.4}$$

with $0 < c_1 < c_2 < 1$.

To find α_k that satisfies these two conditions, we must develop a linesearch algorithm. Algorithm 1 describes the algorithm used to meet these weak Wolfe conditions. It is directly derived form its Euclidian version (see (Burke, 2014) for instance). Note that this is actually a heuristic, since we are not aware of any proof of convergence for this algorithm. But it

is clear that the algorithm terminates only if the two conditions are met. Note that it performs extremely well in practice, but it must be used with caution. In practice, note that the maximum number of iterations of this algorithm is set to 25.

Algorithm 1 A Weak-Wolfe Linesearch bisection algorithm that returns α matching the weak Wolfe conditions (3.3) and (3.4) (with $\alpha_k = \alpha$, $x_k = x$ and $\eta_k = \eta$).

procedure WEAK-WOLFE CONDITIONS LINESEARCH ($0 < c_1 < c_2 < 1$, $\beta = 0$, $\alpha = 1$, $\gamma = \infty$, η, x)
  while True **do**
    if $f(R_x(\alpha\eta)) > f(x) + c_1\alpha\langle \operatorname{grad} f(x), \eta\rangle_x$ **then**
      $\gamma \leftarrow \alpha$
      $\alpha \leftarrow \frac{1}{2}(\beta + \gamma)$
    else
      if $\langle \operatorname{grad} f(R_x(\alpha\eta)), \mathcal{T}_{\alpha\eta}(\eta)\rangle_{R_x(\alpha\eta)} < c_2\langle \operatorname{grad} f(x), \eta\rangle_x$ **then**
        $\beta \leftarrow \alpha$
        $\alpha \leftarrow \begin{cases} 2\beta & \text{if } \gamma = \infty \\ \frac{1}{2}(\beta+\gamma) & \text{otherwise} \end{cases}$
      else
        Break and return α.
      end if
    end if
  end while
end procedure

3.5 Numerical Results

We now ask ourselves if this modification can speedup the solution of the low-rank matrix completion problem. This turns out, quite surprisingly, to be often true: using this new conjugate gradient scheme combined with algorithm 1 seems to give better performances than any other CG scheme and it is often faster than RTRMC (but this depends on the run and this is not always the case).

We decided to compare this Dai-Yuan conjugate gradient algorithm (referred to later as CGDY) to the very simple steepest descent (SD), to RCGMC (conjugate gradient with the Hestenes-Stiefel β-rule, i.e., the default one for RCGMC) and to RTRMC (trust-region). After investigations, others β-rules give performances very similar to those of RCGMC. In the following numerical experiments, gradient tolerance is set to 10^{-8} and matrices are always observed with an oversampling of 4 (meaning that the ratio $\frac{r(m+n-r)}{mn} = 4$, where $r(m+n-r)$ is the dimension of $\mathcal{M}_r$). SD and RCGMC use an Armijo backtracking procedure as a linesearch, while CGDY uses the weak Wolfe algorithm described before. For the weak Wolfe linesearch algorithm, we use $c_1 = 10^{-4}$ and $c_2 = 0.9$. The value of c_1 corresponds to the sufficient decrease parameter of the Armijo backtracking used in SD and RCGMC (Boumal & Absil, 2015). Typical values for c_2 range from 0.1 to 0.9 (see (Wright & Nocedal, 1999) in the Euclidian case, for instance). We will use the later (even though the choice $c_2 = 0.1$ is quite common for non-linear Euclidian CG algorithms (Wright & Nocedal, 1999)), as it seems that lower values tend to give worse results (regarding the time the algorithm takes to converge; this is simply due to the fact that the linesearch algorithm spends more time in the first iterations, trying to meet stronger conditions). We will not study these two parameters

in more details, but an extension of the present work could be to precisely study and tune the choice of c_1 and c_2.

Low-rank matrices are created by first computing two thin rectangular factors $\mathbf{U}$ and $\mathbf{V}$ of sizes respectively $m \times r$ and $r \times n$ and filled with i.i.d. Gaussian random variables such as the product $\mathbf{UV}$ is filled with zero-mean and unit-variance Gaussian variables. We then record the RMSE between $\mathbf{X}$ and the original matrix $\mathbf{UV}$ as a function of time which will be depicted in the forthcoming figures. This can be done efficiently since we have a low-rank factorization of both the target and the "current" approximation in memory. Indeed, if $\mathbf{X} = \mathbf{AB}$ and $\mathbf{M} = \mathbf{UV}$, we have (Boumal & Absil, 2015)

$$\|\mathbf{X} - \mathbf{M}\|_{\ell_2}^2 = \left\| \begin{bmatrix} \mathbf{A} & \mathbf{U} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{B} \\ -\mathbf{V} \end{bmatrix} \right\|_{\ell_2}^2$$

and since the ℓ_2 or Frobenius norm is invariant to unitary matrices, we can compute the thin QR-factorization of both factors $\left(\begin{bmatrix} \mathbf{A} & \mathbf{U} \end{bmatrix} = \mathbf{Q}_1\mathbf{R}_1 \text{ and } \begin{bmatrix} \mathbf{B}^\top & -\mathbf{V}^\top \end{bmatrix} = \mathbf{Q}_2\mathbf{R}_2\right)$ and then simply compute the Frobenius norm $\|\mathbf{M}-\mathbf{X}\|_{\ell_2} = \|\mathbf{Q}_1\mathbf{R}_1\mathbf{R}_2^\top\mathbf{Q}_2^\top\|_{\ell_2} = \|\mathbf{R}_1\mathbf{R}_2^\top\|_{\ell_2}$. The RMSE is then defined by

$$\text{RMSE}(\mathbf{M}, \mathbf{X}) = \sqrt{\frac{\|\mathbf{X} - \mathbf{M}\|_{\ell_2}^2}{mn}}.$$

Numerical experiments are performed sequentially on a desktop computer with a 2.8 GHz dual core Intel Core i7 processor, 8 Gb of RAM and Matlab R2014a using Mac OS X 10.10.3.

Medium Scale Matrix Completion In this first very simple experiment, we sample a $5\,000 \times 5\,000$ rank-10 matrix uniformly at random. Figure 3.1 depicts the performances differences between the four algorithms. It seems from this experiment that CGDY performs quite well compared to the other ones. As it will be the case in the following, we can see that the convergence of the algorithm is very "stable" and linear. This is due to the linesearch which always needs one single iteration to converge after the first run (if using the previous linesearch solution as the initial guess). This is quite remarkable and leads to good performances.

Large Scale Matrix Completion This second experiment is the same as the first one but on larger matrices. Figure 3.2 presents the convergence of the algorithms on a $50\,000 \times 50\,000$ matrix. Clearly, CGDY seems to outperform all other algorithms.

Rectangular Matrix Completion This third experiment (figure 3.3) presents the performances of the four algorithms on the completion of a very *rectangular* matrix of size $10\,000 \times 100\,000$ and of rank 10. In this case, it seems that CGDY does not perform as well as before and has performances very similar to the ones of the other algorithms. Still, it outperforms RCGMC but it is slower than the steepest descent algorithm. It seems hard to find an explanation for the difference between this result and the previous ones where CGDY was always fastest. Indeed, the difference cannot come from a large computational difference: the main difference between the Armijo linesearch and the weak Wolfe one is the computation of the gradient, and it only requires a few operations (of complexity $\mathcal{O}(|\Omega|r)$) additional to the one needed for the cost. The main difference with the previous experiments is the reduction of the size of the search-space, which is approximately proportional to mr. It seems that the weak Wolfe linesearch combined with the Dai-Yuan β-rule is more useful and performs better

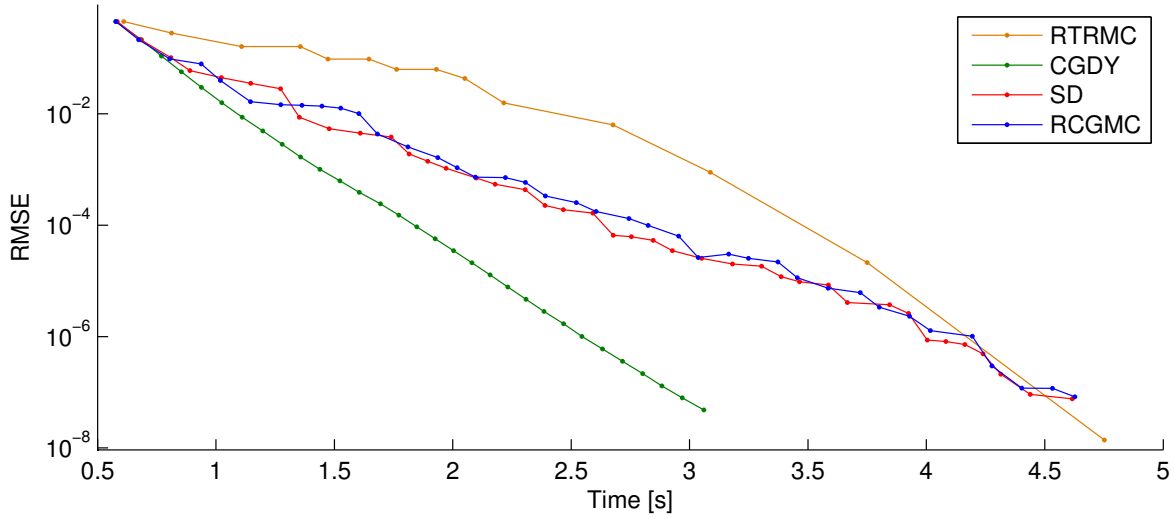


Figure 3.1: Medium scale matrix completion: $5\,000 \times 5\,000$ rank-10 matrix completion problem.

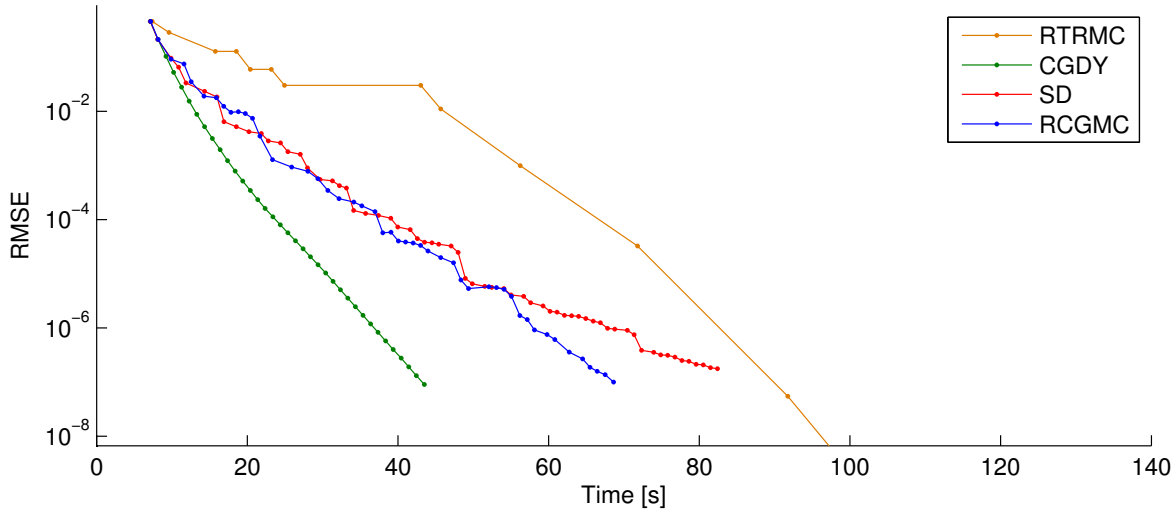


Figure 3.2: Large scale matrix completion: $50\,000 \times 50\,000$ rank-10 matrix completion problem.

when the search-space is larger while an Armijo backtracking and a simple steepest-descent performs better on smaller search-spaces.

Note that these first three experiments are very "stable" in a sense that running the experiments again with other random matrices leads to qualitatively similar results.

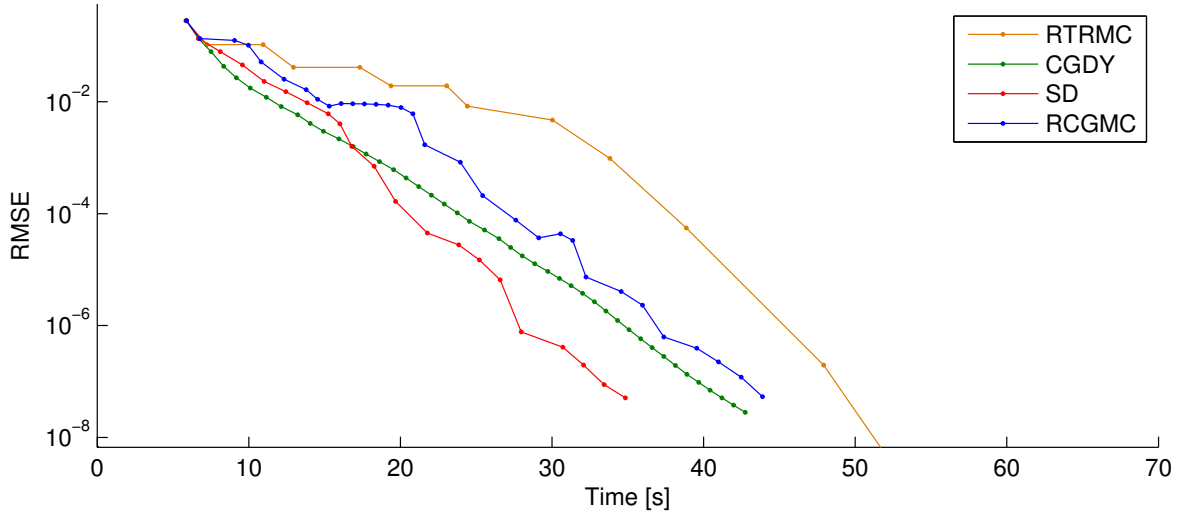


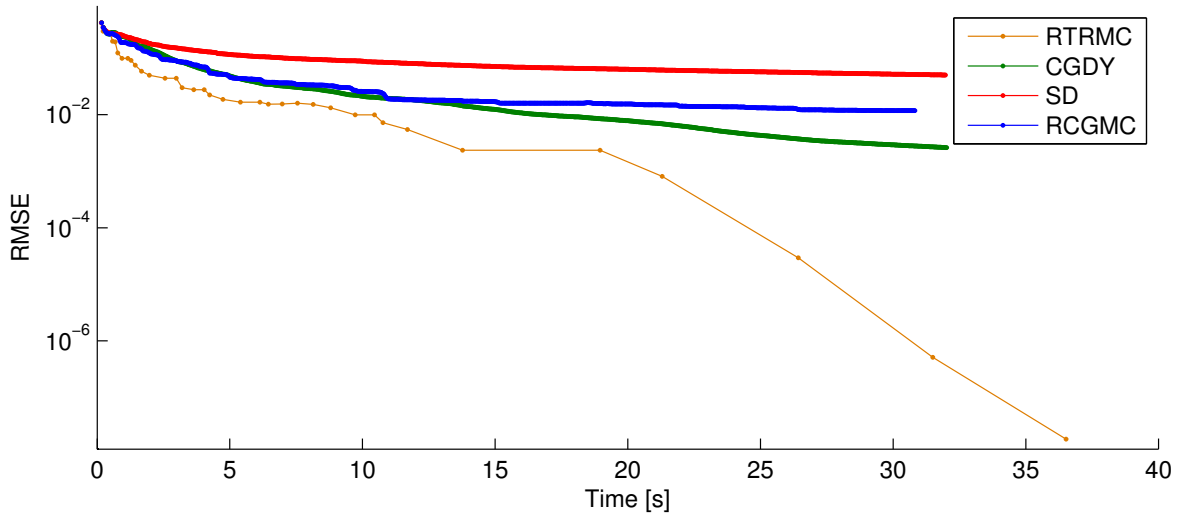
Figure 3.3: Rectangular matrix completion: $10\,000 \times 100\,000$ rank-10 matrix completion problem.

Ill-Conditioned Matrix Completion This experiment (greatly inspired from (Boumal & Absil, 2015)), aims to assess performances of the algorithm when the underlying low-rank matrix is ill-conditioned. The problem is that the ill-conditioning of $\mathbf{UV}$ translates into an (even worse) ill-conditioned hessian at the optimal value, slowing down convergence. Under such scenario, the preconditioner allows the algorithm to keep a good convergence rate while non-preconditionned methods almost always fail to converge in a reasonable amount of time. In this experiment (fig. 3.4), we create a $1\,000 \times 1\,000$ matrix of rank 10 with singular values decaying exponentially from $\sqrt{mn}$ to $\sqrt{mn}e^{-5}$. The matrix is observed with an oversampling of 5. Figure 3.4(a) depicts convergence of the algorithm without the preconditionner while figure 3.4(b) allows us to see the very strong effect of the preconditionner. It seems clear from this experiment that the preconditionner is efficient and that the CGDY algorithm takes great advantage of using it.

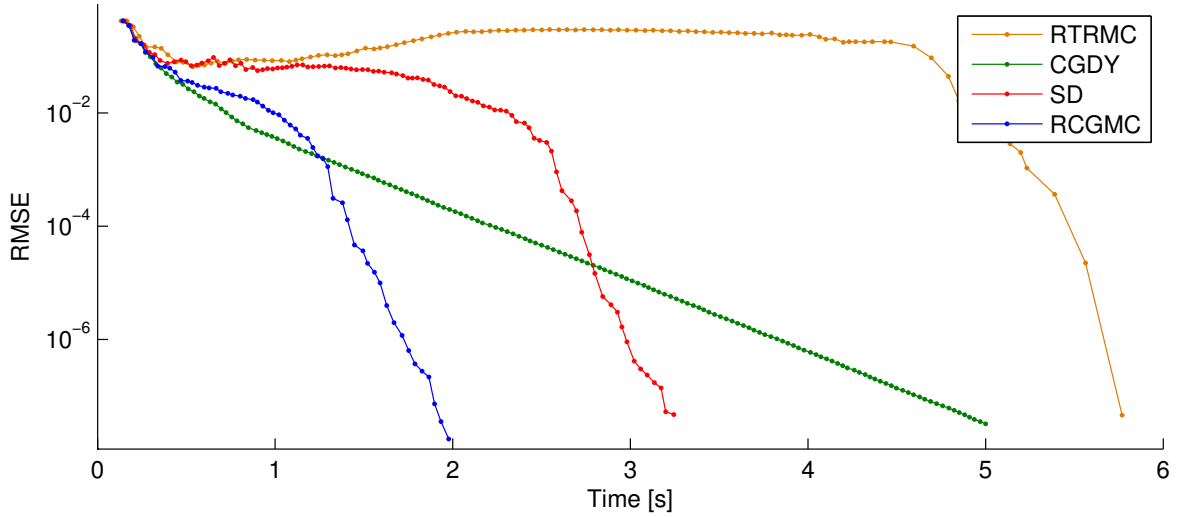
One thing worth mentioning is the qualitative difference of the asymptotic rate of convergence of the CGDY algorithm compared to RCGMC (or even to SD). It seems that they exhibit a much better rate of convergence than CGDY. We were unfortunately not able to find a simple explanation to this. After a lot of investigations, the only conclusion is that it does not depend on the linesearch algorithm (a simple backtracking algorithm giving—qualitatively—the same convergence rate) but only on the difference in the search direction in CGDY.

Also, note that this experiment is much less "stable" than the previous ones, in the sense that running the experiment again with other random matrices often leads to different results: sometimes one of the algorithm does not converge, sometimes most of them do not. We tried to pick the result that best represents relative differences between the algorithms when they

all converge. Still, when it converges, CGDY always pictures this very "linear" convergence rate.



(a) Without the preconditionner



(b) With the preconditionner

Figure 3.4: Ill-conditionned matrix completion: $1\,000 \times 1\,000$ rank-10 matrix completion problem of a matrix observed with an oversampling of 5 and with a condition number of e^5.

3.6 Conclusions

Even though the last experiments do not allow us to formally conclude about the performances of the CGDY scheme, the first ones are a lot more clear, and we can conclude that CGDY has in average better performances than RCGMC and RTRMC, at least in favorable situations. In the worst cases, their performances will be similar (but in this case, it clearly depends on the run).

4 | High Performance Computing

In this chapter, we study the problem of the parallelization of the RTRMC/RCGMC algorithm. For the sake of simplicity, we will focus ourselves on the parallelization of RCGMC. The parallelization of RTRMC would require the parallelization of the hessian, which appears to be quite tedious.

In this document, we will also focus on the parallelization of the cost function and the gradient. Since other operations (retractions, transports, etc.) are mostly basic matrix operations, it is assumed that MATLAB can natively efficiently take care of it.

4.1 The Main Parts of RCGMC

The execution of the RCGMC algorithm requires the computation of the cost and the gradient. Assume we are at point $\mathbf{U}$, an orthogonal basis representing a linear subspace of dimension r of $\mathbb{R}^m$; we are given a mask Ω, $\mathbf{C}$ is the weight matrix (usually uniform on Ω), $\mathbf{M}$ the data (i.e., the known entries on the mask), and λ is the regularization parameter.

Without going into too much details, all the operations required to compute the cost and the gradient can be summarized as follows:

1. Build and solve the n linear systems of size $r \times r$

$$\left[\mathbf{U}^\top \operatorname{diag}(\hat{\mathbf{C}}_i)\mathbf{U} + \lambda^2 \mathbf{I}_r\right] \mathbf{W}_{\mathbf{U},i} = (\mathbf{U}^\top[\mathbf{C}^{(2)} \odot \mathbf{M}])_i, \tag{4.1}$$

   build $\mathbf{W}_\mathbf{U}$ and compute the norm $\|\mathbf{W}_\mathbf{U}\|^2_{\ell_2}$. Subscripts i indicate the i^{th} column of the corresponding matrix, $\odot$ indicates entry-wise product and $\mathbf{C}^{(2)}$ is the entry-wise square of $\mathbf{C}$. We also define $\hat{\mathbf{C}}$ with $\hat{C}_{ij} = C^2_{ij} - \lambda^2 \ \forall (i,j) \in \Omega$ and 0 otherwise ;

2. Compute the *sparse* product $\mathbf{U}\mathbf{W}_\mathbf{U}$ on the mask Ω and compute its Ω-Frobenius norm $\|\mathbf{U}\mathbf{W}_\mathbf{U}\|^2_\Omega = \sum_{(i,j)\in\Omega} (UW_U)^2_{ij}$;

3. Compute the *sparse* residual matrix $\mathbf{R}_\mathbf{U}$ on the mask Ω

$$\mathbf{R}_\mathbf{U} = \hat{\mathbf{C}} \odot (\mathbf{U}\mathbf{W}_\mathbf{U} - \mathbf{M}) - \lambda^2 \mathbf{M}$$

   as well as the Ω-Frobenius norm $\|\mathbf{C} \odot (\mathbf{U}\mathbf{W}_\mathbf{U} - \mathbf{M})\|^2_\Omega$;

4. Compute the *full* product $\mathbf{R}_\mathbf{U}\mathbf{W}_\mathbf{U}^\top$ (where $\mathbf{R}_\mathbf{U}$ is sparse while $\mathbf{W}_\mathbf{U}^\top$ is full);

5. Compute the *full and symmetric* product $\mathbf{W}_\mathbf{U}\mathbf{W}_\mathbf{U}^\top$;

6. Compute the *full* product $\mathbf{U}(\mathbf{W}_\mathbf{U}\mathbf{W}_\mathbf{U}^\top)$.

The cost and the gradient can then be easily computed using

$$f(\mathbf{U}) = \frac{1}{2}\|\mathbf{C} \odot (\mathbf{U}\mathbf{W}_\mathbf{U} - \mathbf{M})\|_{\ell_2}^2 + \frac{\lambda^2}{2}\left(\|\mathbf{W}_\mathbf{U}\|_{\ell_2}^2 - \|\mathbf{U}\mathbf{W}_\mathbf{U}\|_\Omega^2\right)$$

and

$$\operatorname{grad} f(\mathbf{U}) = \mathbf{R}_\mathbf{U}\mathbf{W}_\mathbf{U}^\top + \lambda^2\mathbf{U}(\mathbf{W}_\mathbf{U}\mathbf{W}_\mathbf{U}^\top).$$

Note that in practice, they are both scaled by $\frac{1}{|\Omega|}$ for numerical convenience.

4.2 Parallelization Strategy

To better take advantage of the sparsity and to be able to fine tune every part of the algorithm, the idea is to perform all the heavy computations using compiled C-mex code and to use OpenMP (The OpenMP ARB, 2015) for the parallelization of the algorithm.

A note about how to store the matrices In general, matrices are stored by *columns* first: we store the matrix of size $m \times n$ as a long array of length mn beginning by the first column, followed by the second one, etc.

When dealing with sparse matrices, there are typically three arrays describing a matrix :

- the $\mathbf{V}$ array simply stores each non-zero element of the matrix, stored by column first. $\mathbf{V}[k]$ is the k^{th} element of the sparse matrix;
- the $\mathbf{J_c}$ array is such that the j^{th} column begins at the index $\mathbf{J_c}[j]$ and ends at index $\mathbf{J_c}[j+1]-1$ of the vector $\mathbf{V}$;
- the $\mathbf{I_r}$ array is such that $\mathbf{I_r}[k]$ is simply the row of the k^{th} element of $\mathbf{V}$, $\mathbf{V}[k]$.

This is the format MATLAB uses to store matrices, called the CSC format, and it has the advantage of allowing $\mathcal{O}(1)$ access to every column of the matrix.

But when computing $\mathbf{R}_\mathbf{U}\mathbf{W}_\mathbf{U}^\top$, the optimal storage would be exactly the opposite: $\mathbf{R}_\mathbf{U}$ should be stored by rows first (the CSR format) where each *row* can be accessed in $\mathcal{O}(1)$ and $\mathbf{W}_\mathbf{U}^\top$ should be stored by columns, meaning $\mathbf{W}_\mathbf{U}$ should be stored by rows. This is optimal since it allows to compute each element of resulting matrix as a dot product between two vectors (one row of $\mathbf{R}_\mathbf{U}$ and one column of $\mathbf{W}_\mathbf{U}^\top$) with all elements stored consecutively in memory. Because of that, in the following, we will do the necessary so that $\mathbf{R}_\mathbf{U}$ and $\mathbf{W}_\mathbf{U}$ are both stored by rows first.

The linear systems We need to solve equation (4.1) for each column of $\mathbf{W}_\mathbf{U}$.

To do so[1], we first build the (symmetric) coefficients matrix. Taking advantage that

$$\mathbf{U}^\top \operatorname{diag}\hat{\mathbf{C}}_i\mathbf{U} = \left(\sqrt{\operatorname{diag}\hat{\mathbf{C}}_i}\mathbf{U}\right)^\top\left(\sqrt{\operatorname{diag}\hat{\mathbf{C}}_i}\mathbf{U}\right),$$

we can efficiently build $\mathbf{U}^\top \operatorname{diag}\hat{\mathbf{C}}_i\mathbf{U}$ using the BLAS function `dgemm`. Note that this requires $\hat{C}_i \geq 0$. Then, the term $\lambda^2\mathbf{I}_r$ is added. If $\lambda > 0$, this matrix is always symmetric and positive

[1] This paragraph describes the method already used in RTRMC. See `http://perso.uclouvain.be/nicolas.boumal/RTRMC/`.

definite, and we can compute its Cholesky factorization using the `dpotrf` function. In the mean time, the right-hand-side of the system is computed efficiently thanks to the sparsity and finally the solution is computed using the BLAS `dpotrs` function.

The previous procedure has been incorporated into a single function which is then called in parallel to solve all linear systems efficiently. Note that this requires multiple temporary arrays to store intermediate results (one array per thread, otherwise collisions and data-race can occur) but the time required to allocate these arrays is almost negligible.

$\mathbf{UW_U}$ Computing $\mathbf{UW_U}$ on the mask Ω can be done efficiently thanks to the sparsity. But, because this is used to compute $\mathbf{R_U}$ afterwards, we want the resulting product to be stored by rows first, so that it will be easy to compute $\mathbf{R_U}$. To do so efficiently, we need to have access to each of the rows of $\mathbf{R_U}$ in $\mathcal{O}(1)$ which basically requires us to have access to the sparsity of "$\Omega^\top$" (i.e., the mask we would have if we were working with $\mathbf{M}^\top$ for instance) and the corresponding index-vectors $\mathbf{I_r}$ and $\mathbf{J_c}$. But the advantage of that is that we can precompute it "offline" before the parallel computation, once and for all, since this simply requires transposing the sparse data matrix $\mathbf{M}$. Using this sparsity, the strategy is simply to parallelize over the *rows* of $\mathbf{UW_U}$. The norm $\|\mathbf{UW_U}\|_\Omega^2$ is also computed at the same time since it simply requires summing the squares of the elements of $\mathbf{UW_U}$.

$\mathbf{R_U}$ Computing $\mathbf{R_U}$ is done using parallelization over the *rows* of $\mathbf{R_U}$. Since we have access to $\mathbf{UW_U}$ stored by rows first, and to $\mathbf{C}$ stored by rows first too (we simply need to compute it once and for all and then feed it to the algorithm), computing this component-wise sparse product is easy and fast. The norm $\|\mathbf{C} \odot (\mathbf{UW_U} - \mathbf{M})\|_{\ell_2}^2$ is also computed in the meantime.

$\mathbf{R_U W_U^\top}$ Computing $\mathbf{R_U W_U^\top}$ efficiently in parallel appears to be the most difficult task. As explained earlier, we have to work with $\mathbf{R_U}$ stored by rows first and $\mathbf{W_U^\top}$ stored by columns first. $\mathbf{R_U}$ is already stored as it should be; regarding $\mathbf{W_U}$, we simply have to transpose the full $r \times n$ matrix, which is quite easy and can be done very quickly.

Once this is done, the product $\mathbf{R_U W_U^\top}$ is basically based on 2 nested loops: one over the columns followed by one over the rows. For each row and column, the dot product can easily be computed since the 2 matrices are stored in the right way. To efficiently parallelize the algorithm, we have to parallelize it over the columns of $\mathbf{R_U W_U^\top}$, since parallelization over the rows would have lead to a very tight index set (from 1 to r instead of from 1 to m) and would give to a much less parallelizable algorithm.

$\mathbf{W_U W_U^\top}$ The product $\mathbf{W_U W_U^\top}$ is cheap to compute, thanks to the BLAS `dsyrk` function. Parallelization is possible, but not particularly useful, since this is one of the cheapest operation of the algorithm.

$\mathbf{U(W_U W_U^\top)}$ Finally, the last operation is the $\mathbf{U(W_U W_U^\top)}$ product. This is done simply using the BLAS `dsymm` subroutine. This is also a very cheap operation that does not require any parallelization.

More details on the implementation can be found in appendix B.

4.3 Numerical Results

We now analyze the performances of the parallel implementation. All the following tests were done on a 6 core Intel Xeon CPU E5-1650 v2 @ 3.50GHz with 64 Go of ram using Matlab R2014a and a 64 bit Linux machine.

Figure 4.1 depicts the time taken by all operations of the algorithm. Clearly, the following 3 operations are the most expensive ones:

- Building and solving the n linear systems: about 50% of the execution time;
- Computing $\mathbf{U}\mathbf{W}_\mathbf{U}$ on Ω: about 25% of the time;
- Computing $\mathbf{R}_\mathbf{U}\mathbf{W}_\mathbf{U}^\top$: about 15-20% of the time.

Other operations are very cheap and can be mostly ignored.

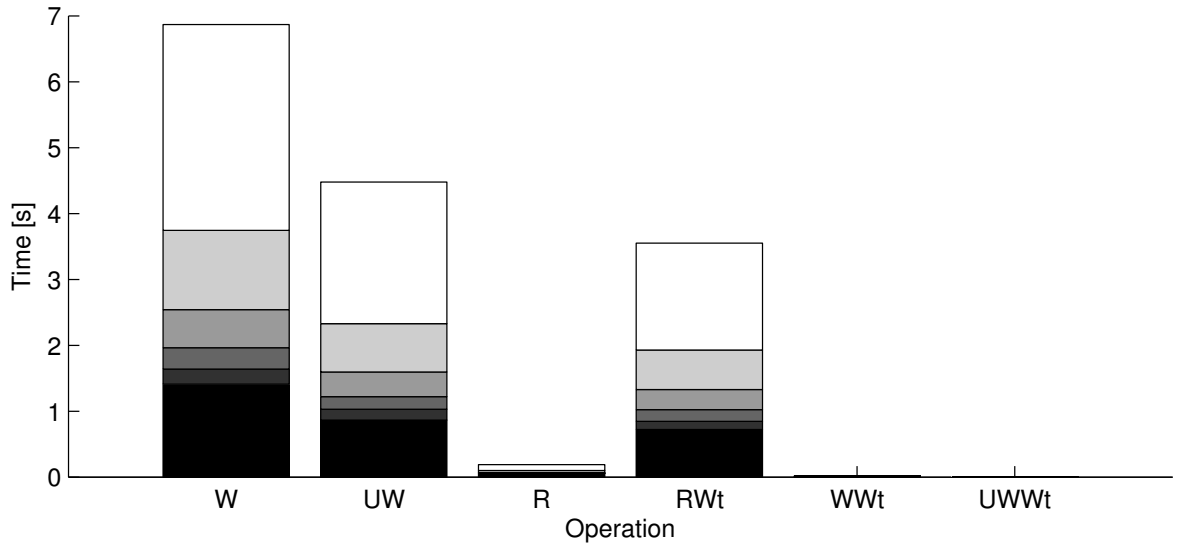


Figure 4.1: The time taken by all operations of the cost and gradient computation on a $300\,000 \times 1\,000\,000$ rank-10 matrix completion problem observed with an oversampling of 4. The colors indicate the number of core, from 1 (at the top in white) to 6 (at the bottom in black). Heights give the time taken by each part of the algorithm. Bars are *not* stacked; for instance, regarding the computation of $\mathbf{W}$, one should read that the time taken with 1 core is approximately 7 seconds while the time taken with 6 cores is approximately 1.5 seconds. $\mathbf{W}$ includes the computation of $\mathbf{W}_\mathbf{U}$ itself, the computation of $\|\mathbf{W}_\mathbf{U}\|_\Omega^2$ and the transposition of $\mathbf{W}_\mathbf{U}$. Note that in practice, these 2 last operations account for less than 2% of the total when using 6 cores. Also note that the time of the $\mathbf{U}(\mathbf{W}_\mathbf{U}\mathbf{W}_\mathbf{U}^\top)$ computation is not zero but indistinguishable at this scale. These timings are the average of 5 successive runs.

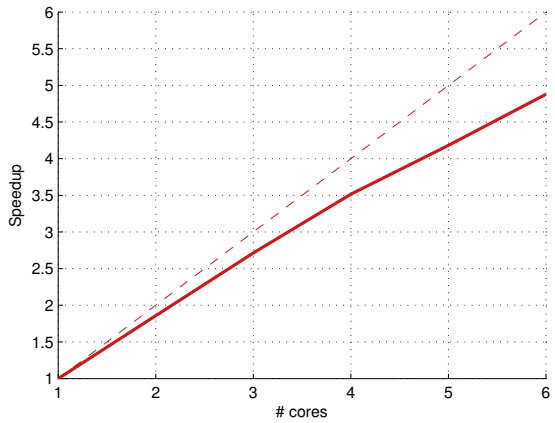
Because of this, we will analyze the speedup of both the full algorithm and of these three operations in particular.

The speedup of an algorithm (or some part of it) is defined as the ratio between the execution time with 1 core and the execution time with n cores, as a function of n:

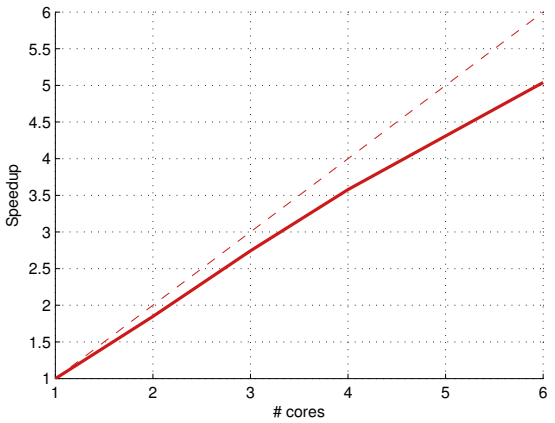
$$\text{speedup}(n) = \frac{\text{time}(1)}{\text{time}(n)}.$$

A very well parallelized algorithm would have an optimal speedup$(n) = n$: each addition of core leads to an optimal performances increase.

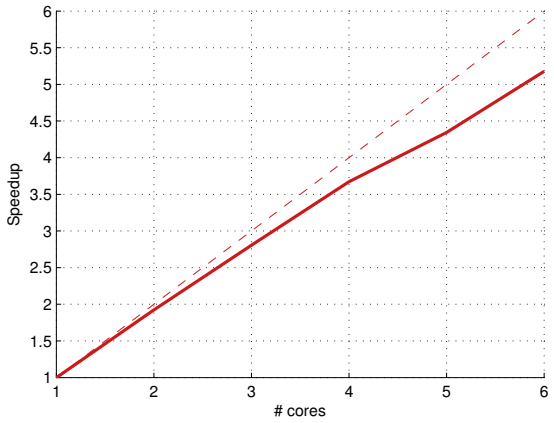
Using the approach described before, we obtain the speedups depicted at figure 4.2, with the execution timings presented in table 4.1 for the computation of the cost and the gradient. These timings are computed as the mean of 5 successive runs (on the same matrix and in the same conditions). Note that the actual values of $\mathbf{U}$, $\mathbf{C}$, λ, $\mathbf{M}$ etc. do not have a real impact on the performances and only the size of the matrices really matters. We can conclude from this graph that the parallelization is very good in general, but slightly not optimal. Note that the fact that the global speedup is slightly below the 3 others is that a few operations are still not parallelized (like the transposition of $\mathbf{W_U}$, the $\mathbf{W_U}\mathbf{W_U}^\top$ product and some other non significant operations). Reasons to this "sub-optimality" can be multiple, but it is likely due to some overhead and false cache-sharing.



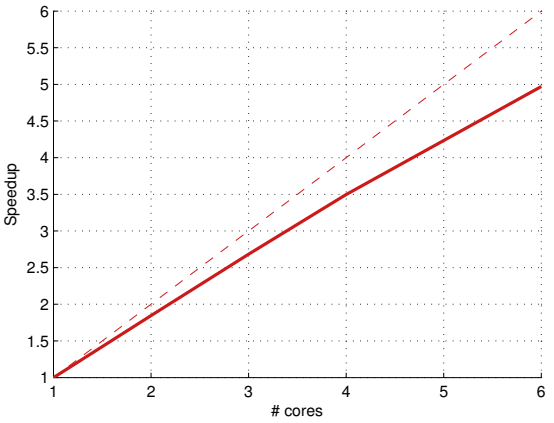
(a) The speedup of the algorithm



(b) The speedup of the computation of $\mathbf{W_U}$ (including only the solutions of the n linear systems, but not the norm $\|\mathbf{W_U}\|^2_\Omega$ nor the transposition of $\mathbf{W_U}$.).



(c) The speedup of the product $\mathbf{U}\mathbf{W_U}$ over Ω.



(d) The speedup of the $\mathbf{R_U}\mathbf{W_U}^\top$ product.

Figure 4.2: Speedup of the parallel implementation of the cost and gradient on a $300\,000 \times 1\,000\,000$ matrix of rank 10, with an oversampling of 4. We can observe that the speedup is almost optimal.

Overhead can occur when multiple threads try to access the same part of the (shared) memory at the same time, resulting in what is called "data-race". This do not lead to errors, but it might slow down a lot the algorithm.

	Full MATLAB	1 core	2 cores	3 cores	4 cores	5 cores	6 cores
Timing [s]	14.06	15.10	8.13	5.56	4.29	3.61	3.09
Speedup		1	1.85	2.71	3.51	4.18	4.87

Table 4.1: Timings for the parallel experiment on a $300\,000 \times 1\,000\,000$ matrix of rank 10 observed with an oversampling of 4. The full MATLAB implementation uses MATLAB for most of the algorithm, but also some C-mex compiled functions for the resolution of the linear systems as well as the computations involving sparse matrices.

False cache-sharing is a more insidious problem. Cache memory is an intermediary storage level between the usual memory (RAM) and the register where computations are performed. When a thread tries to access some memory, it usually loads in the cache memory (close to the corresponding processor) the adjacent elements in the array in order to make (potential) further operations faster. But when multiple threads work on data that are close in memory, this may lead to non-useful data being copied in cache, which can cause significant bandwidth saturation.

Still, the results are quite good in overall, since we almost reach a perfect speed-up. Note that using such an implementation with 1 or 2 cores often leads to better performances that the full MATLAB implementation (which can automatically use all the available cores on the computer). This is important, since it demonstrates that our implementation does not only scale well but is also efficient from the beginning (i.e., with 1 core).

4.4 A Parallel Dai-Yuan Conjugate Gradient

The advantage with this parallel version of RCGMC is that it is well suited for the CGDY algorithm using the weak Wolfe linesearch. Indeed, this linesearch requires multiple computations of *both* the cost and the gradient and the conjugate gradient also require both the cost and the gradient. So this parallel version should be particularly well suited for this method.

Using this parallel implementation, we run the full RCGMC algorithm (still using MANOPT (Boumal *et al.*, 2014) with a full MATLAB implementation for all operations other than the cost and the gradient) to see the global effect of the parallel implementation on the convergence curves of the algorithm. Convergence of the algorithm using a number of cores from 1 to 6 is depicted on figure 4.3 and compared to the full MATLAB implementation[2] (denoted by CGDY).

We can clearly observe that the parallelization has a significant impact on the performances of the algorithm but, as expected, increasing the number of cores does not allow us to drive the execution time towards zero. This is due to the remaining operations (transports, retractions, scalar products, etc.) that are done in full MATLAB and on which we do not really have any control. Still, this implementation is significantly better than the original one and the speedup with 6 cores is slightly above 4, which is already quite good.

An extension of this work would be to do the same for the hessian, in order to have an efficient parallelizable RCGMC algorithm.

[2] Available at `http://perso.uclouvain.be/nicolas.boumal/RTRMC/`

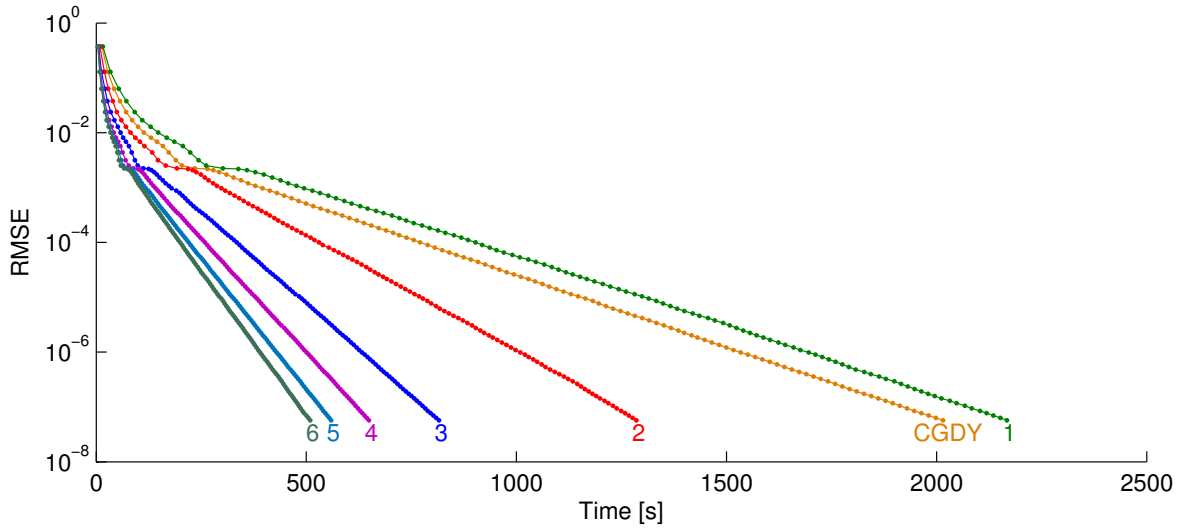


Figure 4.3: 300 000 × 1 000 000 rank-10 matrix completion observed with an oversampling of 4. CGDY is the regular implementation of RCGMC, combined with the Dai-Yuan conjugates gradient algorithm (without any restriction on the number of cores used, so MATLAB may use multiple cores in some part of the algorithm). Other lines are the parallel C-mex implementation, where the numbers indicate the number of cores used by the algorithm. It is not surprising that CGDY is slightly better than the 1-core implementation since MATLAB natively uses multiples cores on some operations.

Part II

Robust Low-Rank Matrix Completion

5 | Robust Low-Rank Matrix Completion

As explained in section 1.3, low-rank matrix completion is often solved by minimizing the ℓ_2 error between the data and a low-rank matrix $\mathbf{X}$. For instance, using Riemannian optimization, one can solve

$$\min_{\mathbf{X} \in \mathcal{M}_r} \|\mathcal{P}_\Omega(\mathbf{X} - \mathbf{M})\|_{\ell_2}$$

(see for instance Vandereycken (2013)).

All formulations that rely on the Frobenius norm however suffer from one drawback: even though they are robust to additive Gaussian noise, they are not well suited to recover the underlying low-rank matrix when the noise becomes non-Gaussian. Here we focus on the situation where only a few of the observed entries, termed *outliers*, are perturbed; that is,

$$\mathbf{M} = \mathbf{M}_0 + \mathbf{S}, \tag{5.1}$$

where $\mathbf{M}_0$ is the unperturbed data matrix of rank r and $\mathbf{S}$ is a sparse matrix. For instance, consider recovering the best rank-1 approximation of the following matrix

$$\mathbf{M}_x = \begin{pmatrix} 2 & -1+x \\ 4 & -2 \end{pmatrix}.$$

If $x = 0$, this matrix is rank-1 since, for instance,

$$\mathbf{M}_0 = \begin{pmatrix} 2 & -1 \\ 4 & -2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \cdot \begin{pmatrix} 2 & -1 \end{pmatrix}.$$

But when $x \neq 0$, this is not the case, and finding the rank-1 matrix that minimizes the ℓ_2 error leads to fundamentaly different solutions.

To observe that, we can simply compute, for each x, the rank-1 SVD of $\mathbf{M}_x$ (which is the solution an ℓ_2 method minimizing $\|\mathbf{M}_x - \mathbf{X}\|_{\ell_2}$ would return) and then compute the RMSE with respect to the original matrix $\mathbf{M}_0$. This is depicted in figure 5.1, and we can see that the error starts to grow as soon as $x \neq 0$.

However, if the method was able to find out that only the top right entry of $\mathbf{M}_x$ is corrupted by noise, then it would be able to recover $\mathbf{M}_0$ exactly by removing the top right entry from the mask Ω and performing low-rank matrix completion. More generally, in the problem of completing from its know entries Ω a matrix $\mathbf{M}$ generated as in equation (5.1), the ability of detecting the outliers in $\mathcal{P}_\Omega(\mathbf{M})$ (i.e., the entries affected by the sparse matrix $\mathcal{P}_\Omega(\mathbf{S})$) and removing those entries from the mask Ω would open the way for an exact recovery of the rank-r matrix $\mathbf{M}_0$.

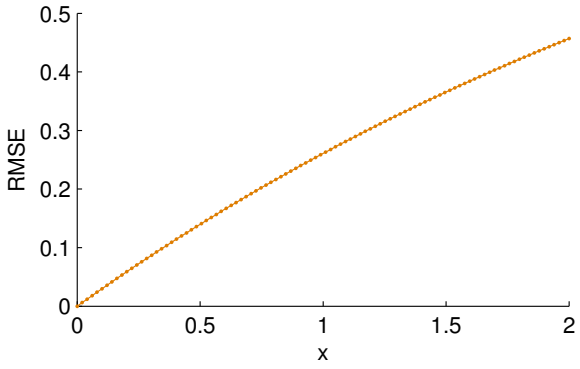


Figure 5.1: The error between the recovered matrix and the original one when minimizing the ℓ_2 norm, as a function of x. We can clearly observe that when using an ℓ_2 loss function, even a small perturbation on a single entry of the matrix can lead to a large error with respect to the original matrix.

One solution to solve this problem in general would be to minimize the number of different entries between the data and the recovered matrix. In this context, one would like to solve

$$\min_{\mathbf{X} \in M_r} \|\mathcal{P}_\Omega(\mathbf{X} - \mathbf{M})\|_{\ell_0} \tag{5.2}$$

where the ℓ_0 "norm" is defined as

$$\|X\|_{\ell_0} = \#\{(i,j) : X_{ij} \neq 0\}.$$

But this problem seems hard, since the objective function is not even continuous in $\mathbf{X}$.

It is well known in the compressed sensing community that the tightest convex relaxation of the ℓ_0 norm is the ℓ_1 norm: assume we work in $\mathbb{R}$ with $x \in [-1, 1]$. Hence we simply minimize $|x|_{\ell_0}$. In this case, it is easy to see that the ℓ_1 norm, $|x|$, is the tightest convex relaxation of the ℓ_0 norm, $|x|_{\ell_0}$. Even though this does not prove anything, it allows to give some insight of why minimizing the ℓ_1 norm often gives the same solution as minimizing the ℓ_0 norm and why this is the right relaxation.

The problem we want to solve can then be stated as.

$$\min_{\mathbf{X} \in M_r} \|\mathcal{P}_\Omega(\mathbf{X} - \mathbf{M})\|_{\ell_1}. \tag{5.3}$$

5.1 Previous work

Matrix completion in the presence of outliers has been considered in several papers.

Chen *et al.* (2011) studied the problem of low-rank matrix completion where a large number of columns are arbitrarily corrupted. They showed that only a small fraction of the entries are needed in order to recover the low-rank matrix with high probability, without any assumptions on the location nor the amplitude of the corrupted entries.

Both Li (2013) and Chen *et al.* (2013) studied a harder problem, when a constant fraction of the entries (not the columns) of the matrix are outliers. They studied what conditions need

to be imposed in order for the following convex optimization problem

$$\begin{aligned} \min \quad & \gamma\|\mathbf{X}\|_* + \|\mathbf{E}\|_{\ell_1} \\ \text{subject to} \quad & \mathcal{P}_\Omega(\mathbf{X}+\mathbf{E}) = \mathcal{P}_\Omega(\mathbf{M}) \end{aligned}$$

to exactly recover the underlying low-rank matrix (with $\|\cdot\|_*$ the nuclear norm). Basically, they showed that there exist universal constants such that with overwhelming probability the solution of the problem is equal to M on the mask Ω. In the close context of low-rank PCA, Candès *et al.* (2011) was also able to solve the same problem. The advantage of such an algorithm is that it is convex and can then be analyzed thoroughly.

This robust formulation has been improved to deal with Gaussian noise (Hastie, 2012), leading to the following convex optimization problem

$$\begin{aligned} \min \quad & \lambda\|\mathbf{X}\|_* + \|\mathbf{E}_1\|_{\ell_1} + \|\mathbf{E}_2\|_{\ell_2} \\ \text{subject to} \quad & \mathcal{P}_\Omega(\mathbf{X}+\mathbf{E}_1+\mathbf{E}_2) = \mathcal{P}_\Omega(\mathbf{M}) \end{aligned}.$$

He *et al.* (2011) developed a robust version of the GROUSE algorithm (Balzano *et al.*, 2010), named GRASTA, which aims at solving the problem of *robust* subspace tracking. Their algorithm can be casted to solve problems formulated as

$$\begin{aligned} \min \quad & \|\mathcal{P}_\Omega(\mathbf{S})\|_{\ell_1} \\ \text{subject to} \quad & \mathcal{P}_\Omega(\mathbf{UV}+\mathbf{S}) = \mathcal{P}_\Omega(\mathbf{M}) \\ & \mathbf{U} \in \mathrm{Gr}(m,r) \\ & \mathbf{V} \in \mathbb{R}^{r\times n} \end{aligned}$$

where $\mathrm{Gr}(m,r)$ is the Grassman manifold, i.e., the set of linear r-dimensional subspaces of $\mathbb{R}^m$. GRASTA solves this problem by first building the augmented Lagrangian problem, and it then solves it by alternating between $\mathbf{V}$, his dual variables and $\mathbf{U}$ and by performing steepest descent on the Grassman manifold. The advantage of their algorithm is that it is designed to tackle the problem of *online* subspace estimation from incomplete data, hence it can also be casted to solve online low-rank matrix completion where we observe one column of the matrix $\mathbf{M}$ at a time.

Nie *et al.* (2012a,b) solved a slightly more general problem where all norms become arbitrary p-norms

$$\min \lambda\|\mathbf{X}\|_{S_p}^p + \|\mathcal{P}_\Omega(\mathbf{X}-\mathbf{D})\|_{\ell_p}^p,$$

where $\|\mathbf{X}\|_{S_p}^p = \sum_{i=1}^{\min(m,n)} \sigma_i^p(\mathbf{X})$ and $\|\mathbf{X}\|_{\ell_p}^p = \sum_{i=1,j=1}^{m,n} |X_{ij}|^p$. The algorithm used to solve this non-convex program (when $p<1$) is, again, an augmented Lagrangian method. We were unfortunately unable to obtain or write an efficient implementation of this algorithm since it requires the storage of the full $m\times n$ matrix, as well as SVD of full matrices of this size. This formulation, however, is efficient for moderate size problems.

Yan *et al.* (2013) solved ℓ_2 problems of the form

$$\min_{X\in\mathcal{M}_r} \|\mathcal{P}_\Omega(\mathbf{X}-\mathbf{M})\|_{\ell_2}$$

where the mask Ω is adapted at each iteration to remove the suspected outliers. The idea is to first solve the problem with the original mask Ω, detect outliers, adapt the mask, and then solve the problem again until convergence. Intermediate problems are handled using RTRMC (Boumal & Absil, 2011).

Yang *et al.* (2014) studied the problem of robust low-rank matrix completion using a non-convex loss-function. They solve the following problem

$$\min_{X\in\mathbb{R}^{m\times n}:\operatorname{rank}(X)\leq r} \frac{\sigma^2}{2}\sum_{(i,j)\in\Omega}\left(1-\exp\left(-(X_{ij}-B_{ij})^2/\sigma^2\right)\right),$$

where the rank-constraint is relaxed using the now standard nuclear-norm heuristic.

Finally, Klopp *et al.* (2014) studied the optimal reconstruction error in the case of matrix completion, where the observations are noisy and column-wise or element-wise corrupted and where the only piece of information needed is a bound on the matrix entries. They provided a range of (optimal) estimators to solve such problems with guarantees.

5.2 Our Contribution

In this chapter, we propose and try to analyze a few methods to solve this problem.

The first method uses the RTRMC algorithm and the weights present in the objective function to emulate an ℓ_1 norm. Even though we have good theoretical convergence guarantees, the intermediate ℓ_2 problems appear to be challenging to solve accurately.

The second one is a very simple alternating linear minimization method. The advantage is that the method is very easy to implement, but it suffers from a very slow convergence due to the time one iteration takes.

Finally, the last method is based on Riemannian optimization and smoothing techniques. The idea is to smooth the ℓ_1 norm and to solve the resulting problem using optimization on manifolds. Clearly, this method seems the most "scalable" and robust one, as it is able to successfully solve large scale problems where the data contain very strong outliers.

5.3 Iteratively Reweighted Least-Squares Method

This method, called *Iteratively Reweighted Least-Squares* (IRLS), is an easy way to solve problems by successively solving multiple least-squares problems.

5.3.1 Convergence of the IRLS

Let us assume we are interested in solving the following problem

$$\min_{\mathbf{X}\in\mathcal{M}_r} \underbrace{\sum_{(i,j)\in\Omega} h\left(|X_{ij}-M_{ij}|\right)}_{C_h(\mathbf{X})}, \tag{5.4}$$

where $h:\mathbb{R}^+\to\mathbb{R}$ is some function. In the ℓ_2 case we have $h(x)=x^2$, while in the ℓ_1 case we have $h(x)=x$. In this section, we will keep things general and make as few assumptions as possible on h.

The IRLS method attempts to solve this problem by successively solving multiple weighted least-squares problems like

$$\min_{\mathbf{X}\in\mathcal{M}_r} \sum_{(i,j)\in\Omega} w_{ij} \left(X_{ij} - M_{ij}\right)^2.$$

This method has then the advantage of only requiring to solve (weighted) least-squares problems, which is something many algorithms can do.

In short, the algorithm will create a sequence of iterates $\{\mathbf{X}^{(k)}\}_{k=1}^{K}$ by solving

$$\mathbf{X}^{(k+1)} = \text{argmin}_{\mathbf{X}\in\mathcal{M}_r} \sum_{(i,j)\in\Omega} w_{ij}\left(\mathbf{X}^{(k)}\right)\left(X_{ij} - M_{ij}\right)^2.$$

Because we want this method to have the same optimal value as the original problem, we need to choose (Aftab & Hartley, 2015)

$$w_{ij}(\mathbf{X}) = \frac{h'(|X_{ij} - M_{ij}|)}{2|X_{ij} - M_{ij}|}.$$

Still, this does not prove that the algorithm will converge towards a critical point of (5.4).

The most important ingredient to ensure convergence of the algorithm is that h has to be such that $x \to h(\sqrt{x})$ is concave. Indeed, in this case, finding $\mathbf{X}^{(k+1)}$ such that

$$\sum_{(i,j)\in\Omega} w_{ij}\left(\mathbf{X}^{(k)}\right)\left(X_{ij}^{(k+1)} - M_{ij}\right)^2 < \sum_{(i,j)\in\Omega} w_{ij}\left(\mathbf{X}^{(k)}\right)\left(X_{ij}^{(k)} - M_{ij}\right)^2$$

induces a decrease in (5.4) (Aftab & Hartley, 2015), i.e.,

$$\sum_{(i,j)\in\Omega} h\left(|X_{ij}^{(k+1)} - M_{ij}|\right) < \sum_{(i,j)\in\Omega} h\left(|X_{ij}^{(k)} - M_{ij}|\right).$$

Aftab & Hartley (2015) formally proved that under the following conditions on h, the IRLS algorithm converges towards a critical point of the original problem (5.4):

1. h is continuous;
2. $x \to h(\sqrt{x})$ is concave and differentiable for $x \geq 0$;
3. the function
$$\mathbf{w} \to W(\mathbf{w}) = \text{argmin}_{\mathbf{X}\in\mathcal{M}_r} \sum_{(i,j)\in\Omega} w_{ij}(X_{ij} - M_{ij})^2$$
is a continuous function of the weights $\mathbf{w}$
4. the sublevel set $\{\mathbf{X} : C_h(\mathbf{X}) \leq C_h(\mathbf{X}^{(0)})\}$ is bounded.

5.3.2 The Choice of the h Function

In order to have a convergent method, we need to carefully choose the $h : \mathbb{R}^+ \to \mathbb{R}$ function. Aftab & Hartley give a wide range of interesting function. Unfortunately, the h function corresponding to the absolute value, $h(x) = x$, is not among these functions, since $x \to$

$h(\sqrt{x}) = \sqrt{x}$ is not differentiable at 0. In this document, we will thus focus on the Pseudo-Huber function defined by[1]

$$h : \mathbb{R}^+ \to \mathbb{R} : x \to h(x) = \sqrt{\delta^2 + x^2}.$$

His derivative is given by h' such that

$$h'(x) = \frac{x}{\sqrt{\delta^2 + x^2}}$$

and the weights are thus given by

$$w_{ij}(\mathbf{X}) = \frac{2}{\sqrt{\delta^2 + |M_{ij} - X_{ij}|}}$$

for a given δ which will be a parameter of the algorithm. The value δ is actually some regularization, since it controls the non-differentiability of h around 0. The smaller the δ, the closer h is to $x \to |x|$.

We can then easily check which conditions are met in our problem:

1. h is continuous;
2. $x \to h(\sqrt{x})$ is concave;
3. $W(\mathbf{w})$ is continuous since this is a least-squares problem and the weights are strictly positive;
4. the fourth condition is not garanteed to be met in all cases. This typically depends on the data. There exist many pathological cases where this will not be met; consider for instance the 2×2 rank-2 matrix completion where we only observe one single entry. This obviously gives us a non-bounded set of rank-2 solution (with a cost $C_h(\mathbf{X}) = 0$). In the following, we will then assume that this fourth condition is met. *Intuitively*, if the mask is uniform with enough entries and if the rank r is at most equal to the rank of $\mathbf{M}$, it seems that the sublevel set $\{\mathbf{X} : C_h(\mathbf{X}) \leq C_h(\mathbf{X}^{(0)})\}$ should be bounded. The addition of some noise (like a uniform Gaussian noise) can also help to have a bounded sublevel set.

Using these weights and the IRLS algorithm, we can conclude that our algorithm should converge towards critical points of (5.4), *as long as* the intermediate weighted least-squares problems are solved accurately (which is not so easy in pratice) and that condition 4 is met.

5.3.3 The Algorithm

The algorithm is given at algorithm 2. The stopping criterion comes from the fact that, in the limit when $k \to \infty$, the solution is in principle equal to the solution of the original problem. Hence, their costs should be close, and we will then stop the algorithm if $|f^{(k+1)} - f^{(k)}|$ is below some threshold.

The initial point can be any low-rank matrix. In practice, using the sparse rank-r SVD of $\mathcal{P}_\Omega(\mathbf{M})$ often gives a good starting point.

[1] The function used here is slightly different from the one in (Aftab & Hartley, 2015) but makes more sense in the limit $\delta \to 0$.

Algorithm 2 Iterated Reweighted Least-Squares

procedure IRLS($\mathbf{X}^{(0)}, \delta, \epsilon$)
  $f^{(0)} = \frac{1}{\Omega} \sum_{(i,j)\in\Omega} \sqrt{\delta^2 + (X_{ij} - M_{ij})^2}$
  $k \leftarrow 0$
  $e \leftarrow \infty$
  while $e > \epsilon$ **do**
    $w_{ij}^{(k+1)} \leftarrow \frac{1}{2\sqrt{\delta^2 + |M_{ij} - X_{ij}^{(k)}|}}$
    $\mathbf{X}^{(k+1)} \leftarrow \operatorname{argmin}_{\mathbf{X}\in\mathcal{M}_r} \sum_{(i,j)\in\Omega} w_{ij}^{(k+1)} (M_{ij} - X_{ij})^2$
    $f^{(k+1)} = \frac{1}{\Omega} \sum_{(i,j)\in\Omega} \sqrt{\delta^2 + (X_{ij}^{(k+1)} - M_{ij})^2}$
    $e \leftarrow |f^{(k+1)} - f^{(k)}|$
    $k \leftarrow k + 1$
  end while
  return $\mathbf{X}^{(k)}$
end procedure

5.4 Alternating Linear Matrix Completion

Now let us look at the problem from another angle. As already discussed, the problem can be stated as

$$\min_{\mathbf{U}\in\mathbb{R}^{m\times r}, \mathbf{V}\in\mathbb{R}^{r\times n}} \sum_{(i,j)\in\Omega} |M_{ij} - (UV)_{ij}|.$$

The algorithm we will present and analyse in this section is an heuristic method where we fix one factor, optimize with respect to the other and then repeat this procedure for each factor consecutively. This procedure should hopefully converge towards a local minimum of the cost function.

To begin with, let us suppose $\mathbf{V}$ is fixed. In this case, $\mathbf{U}$ can be found by solving the following linear program:

$$\min_{\mathbf{U}} \sum_{(i,j)\in\Omega} |M_{ij} - \sum_k U_{ik} V_{kj}|$$

which can be decomposed by the lines of U: for each line i^*, we solve (using MATLAB's notation where ":" denotes the extraction of the corresponding component)

$$\min_{\mathbf{U}_{i^*,:}} \sum_{(i,j)\in\Omega : i=i^*} |M_{i^*j} - \sum_k U_{i^*k} V_{kj}|$$

to recover the $i^{*\text{th}}$ line of U.

In a similar way, should $\mathbf{U}$ be fixed, we can recover $\mathbf{V}$ by solving

$$\min_{\mathbf{V}} \sum_{(i,j)\in\Omega} |M_{ij} - \sum_k U_{ik} V_{kj}|.$$

To recover each column j^* of V, we can solve the following linear program

$$\min_{\mathbf{V}_{:,j^*}} \sum_{(i,j)\in\Omega : j=j^*} |M_{ij^*} - \sum_k U_{ik} V_{kj^*}|.$$

The advantages of this procedure are multiple: first, the implementation is *very* easy; then, even if this is only an heuristic, this procedure produces in general good results (in the sense that the original underlying low-rank matrix is effectively recovered); finally, it seems *very* robust to outliers, in the sense that the strengh of the outliers does not seem to affect the convergence of the algorihtm at all. This can be explained easily using the fact that the LP's are very robust to outliers.

5.4.1 Solving the LP's

It is not completely obvious at first glance that this problem, when $\mathbf{U}$ or $\mathbf{V}$ is fixed, is an LP. But it is actually the case.

Again, assume $\mathbf{V}$ is fixed. By introducing an extra variable $\mathbf{t}$, defined on Ω, we can reformulate the problem as a linear program. If we define $t_{ij} \geq |M_{ij} - (UV)_{ij}|$, the problem can be formulated as

$$\begin{aligned}
&\min_{\mathbf{U}\in\mathbb{R}^{m\times r}} \|\mathcal{P}_\Omega(\mathbf{M} - \mathbf{U}\cdot\mathbf{V})\|_{\ell_1} \\
&= \min_{\mathbf{U}\in\mathbb{R}^{m\times r}} \sum_{(ij)\in\Omega} |M_{ij} - (UV)_{ij}| \\
&= \min_{\mathbf{U}\in\mathbb{R}^{m\times r}, t_{ij}\forall(ij)\in\Omega} \sum_{(ij)\in\Omega} t_{ij} \text{ s.t. } \begin{cases} t_{ij} \geq M_{ij} - \sum_{k=1}^r U_{ik}V_{kj}, \\ t_{ij} \geq -M_{ij} + \sum_{k=1}^r U_{ik}V_{kj}, \end{cases} \\
&= \sum_{i^*=1}^m \min_{\mathbf{U}_{i^*,:}} \sum_{(i,j)\in\Omega:i=i^*} t_{i^*j} \text{ s.t. } \begin{cases} t_{i^*j} \geq M_{i^*j} - \sum_{k=1}^r U_{i^*k}V_{kj}, \\ t_{i^*j} \geq -M_{i^*j} + \sum_{k=1}^r U_{i^*k}V_{kj}. \end{cases}
\end{aligned}$$

One advantage is that these are m *uncoupled* LP's: it can then be solved much more efficiently than the corresponding full LP, *and* it can be done in parallel. The exact same procedure can be done for the case where $\mathbf{U}$ is fixed.

Let us now briefly analyze the size of each of the LP's. For simplicity, assume $m = n$. When $\mathbf{V}$ is fixed, each row of $\mathbf{U}$ is associated to an LP containing

- r variables (for the length of the row of $\mathbf{U}$) ;
- $\frac{|\Omega|}{m} = \frac{fr(m+n-r)}{m} \approx 2fr$ (assuming $r \ll m$) variables, one for each of the variables of $\mathbf{t}$ that are to be taken into account, where f is the oversampling factor.

So the number of variables associated to each row of $\mathbf{U}$ is approximately $r + 2fr = (2f+1)r$. The number of constraints is twice the number of variables of $\mathbf{t}$ that are to be taken into account, that is $4fr$. If $f = 4$ for instance, the number of variables in each LP is $9r$ and the number of constraints is $16r$.

5.4.2 The Algorithm

The algorithm named Alternating Linear Matrix Completion (ALMC) is stated on algorithm 3. The stopping criterion is discussed in the next section.

This algorithm appears amazingly simple but still provides good results in practice.

Algorithm 3 Alternating Linear Matrix Completion

procedure ALMC($\mathbf{U}^{(0)}, \mathbf{V}^{(0)}, \epsilon$)
  $k \leftarrow 0$
  $e \leftarrow \infty$
  while $e > \epsilon$ **do**
    $\mathbf{U}^{(k+1)} \leftarrow \text{argmin}_{\mathbf{U}\in\mathbb{R}^{m\times r}} \|\mathcal{P}_\Omega(\mathbf{M} - \mathbf{U}\cdot\mathbf{V}^{(k)})\|_{\ell_1}$
    $\mathbf{V}^{(k+1)} \leftarrow \text{argmin}_{\mathbf{V}\in\mathbb{R}^{r\times n}} \|\mathcal{P}_\Omega(\mathbf{M} - \mathbf{U}^{(k+1)}\cdot\mathbf{V})\|_{\ell_1}$
    $e \leftarrow \|\mathcal{P}_\Omega(\mathbf{M} - \mathbf{U}^{(k)}\cdot\mathbf{V}^{(k)})\|_{\ell_1} - \|\mathcal{P}_\Omega(\mathbf{M} - \mathbf{U}^{(k+1)}\cdot\mathbf{V}^{(k+1)})\|_{\ell_1}$
    $k \leftarrow k+1$
  end while
  return $\mathbf{U}^{(k)}$, $\mathbf{V}^{(k)}$
end procedure

5.4.3 Convergence

It is actually quite easy to prove convergence of the algorithm using a result found in (Razaviyayn *et al.*, 2013, theorem 3). They proved the following.

Theorem 1 (Convergence of ALMC (Razaviyayn *et al.*, 2013))**.** *If iterates* $(\mathbf{U}^{(k)}, \mathbf{V}^{(k)})$ *are generated by algorithm 3, every limit point is a coordinate-wise minimizer of the function*

$$f : \mathbb{R}^{m\times r} \times \mathbb{R}^{r\times n} : (\mathbf{U}, \mathbf{V}) \to \|\mathcal{P}_\Omega(\mathbf{M} - \mathbf{U}\mathbf{V})\|_{\ell_1}$$

Proof. [2] Let us define $\mathbf{X}^{(r)} = (\mathbf{U}^{(\lceil r/2\rceil)}, \mathbf{V}^{(\lfloor r/2\rfloor)})$. Hence, r increases by one when we update either $\mathbf{U}$ or $\mathbf{V}$.

We trivially have, $\forall r \geq 0$:

$$f(\mathbf{X}^{(r)}) \geq f(\mathbf{X}^{(r+1)}) \geq \cdots \geq 0.$$

Note that this gives sense to the stopping criterion of the algorithm[3].

Let us extract from $\{\mathbf{X}^{(r)}\}_{r\in\mathbb{N}}$ a subsequence converging towards a limit point $\mathbf{X}^* = (\mathbf{U}^*, \mathbf{V}^*)$: $\{\mathbf{X}^{(r_j)}\}_{j\in\mathbb{N}}$. Let us also focus on the $\mathbf{U}$ iterates. For all $\mathbf{U} \in \mathbb{R}^{m\times r}$, and for all $j \in \mathbb{N}$ we have

$$\begin{aligned}
f(\mathbf{U}, \mathbf{V}^{(\lfloor r_j/2\rfloor)}) &\geq \min_{\mathbf{U}\in\mathbb{R}^{m\times r}} f(\mathbf{U}, \mathbf{V}^{(\lfloor r_j/2\rfloor)}) \\
&= f(\mathbf{U}^{(\lfloor r_j/2\rfloor+1)}, \mathbf{V}^{(\lfloor r_j/2\rfloor)}) \\
&\geq f(\mathbf{X}^{2(\lfloor r_j/2\rfloor+1)}) \\
&\geq f(\mathbf{X}^{(r_j+2)}) \\
&\geq f(\mathbf{X}^{(r_{j+2})}) \\
&= f(\mathbf{U}^{(\lceil r_{j+2}/2\rceil)}, \mathbf{V}^{(\lfloor r_{j+2}/2\rfloor)}).
\end{aligned}$$

By letting $j \to \infty$, we have $\forall \mathbf{U} \in \mathbb{R}^{m\times r}$

$$f(\mathbf{U}, \mathbf{V}^*) \geq f(\mathbf{U}^*, \mathbf{V}^*).$$

This is sufficient to conclude that $\mathbf{U}^*$ is a coordinate-wise minimizer of f.

[2] We emphasize the fact that this proof is nothing more than a rewriting of the proof of (Razaviyayn *et al.*, 2013, theorem 3) using our algorithm and notations.

[3] Meaning the algorithm will terminate at some point.

The exact same proof can be used to note that $\mathbf{V}^*$ is a coordinate-wise minimizer of f.

□

The problem with this proof is that it does not prove the existence of any limit point of the algorithm.

To *try* to ensure this, we can modify the algorithm by keeping the $\mathbf{U}^{(k)}$ orthogonal (such that $\mathbf{U}^{(k)\top}\mathbf{U}^{(k)} = \mathbf{I}_r$) using a thin QR-factorization $\mathbf{U}^{(k)} = \mathbf{QR}$ and by "putting" the triangular factor $\mathbf{R}$ into $\mathbf{V}^{(k)}$.

Intuitively, this *heuristic* prevents both the $\mathbf{U}$ and $\mathbf{V}$ factors to grow indefinitely. Indeed, without it, $\mathbf{U}$ can be arbitrarily small. For instance, since $\forall \mathbf{M} \in \mathbb{R}^{r\times r}$ invertible we have $\mathbf{UV} = (\mathbf{UM})(\mathbf{M}^{-1}\mathbf{V}) = \tilde{\mathbf{U}}\tilde{\mathbf{V}}$, we can then take

$$\mathbf{M} = \epsilon\mathbf{I}$$

so that $\tilde{\mathbf{U}} \to 0$ and $\tilde{\mathbf{V}} \to \infty$ if $\epsilon \to 0$. Note that this exact procedure cannot happen during the execution of the algorithm (since we optimize each factor consecutively, hence we cannot modify $\mathbf{U}$ and $\mathbf{V}$ at the same time). Still, it gives some insight on why this problem might happen. This is not a problem *per se* (since the product of $\mathbf{U}$ and $\mathbf{V}$ stays the same, and the objective function keeps decreasing) but it prevents us to prove convergence towards a coordinate-wise minimum. More details can be found in appendix C.

5.5 Smoothing Techniques

This section is mainly extracted from a paper in preparation: "Robust Low-Rank Matrix Completion by Riemannian Optimization" by Cambier & Absil.

The idea of this algorithm is to handle the low-rank constraint using the fact that $\mathcal{M}_r$ is a smooth Riemannian manifold, and to handle the non-differentiability of the objective function by smoothing techniques, relying on the fact that for small δ,

$$|x| \approx \sqrt{x^2 + \delta^2}.$$

Using these 2 ideas, we will be able to solve

$$\min_{\mathbf{X}\in\mathcal{M}_r} \|\mathcal{P}_\Omega(\mathbf{X} - \mathbf{M})\|_{\ell_1} + \lambda\|\mathcal{P}_{\bar{\Omega}}(\mathbf{X})\|_{\ell_2} \tag{5.5}$$

where λ is a regularization parameter, useful in applications. In synthetic artificial experiments, it can usually be set to zero.

5.5.1 The Low-Rank Matrices Manifold

Definition

The *low-rank matrix manifold*

$$\mathcal{M}_r = \{\mathbf{X} \in \mathbb{R}^{m\times n} : \operatorname{rank}(\mathbf{X}) = r\},$$

where $r \leq \min(m, n)$, is known to be a smooth manifold embedded in $\mathbb{R}^{m\times n}$ of dimension $r(m+n-r)$ (Lee, 2003; Vandereycken *et al.*, 2009). Hence, optimization techniques presented in (Absil *et al.*, 2008) can be applied to solve smooth optimization problems where constraints are formulated using $\mathcal{M}_r$.

Storing the Low-Rank Matrix

There are several ways of describing a matrix $\mathbf{X} \in \mathcal{M}_r$ (Absil & Oseledets, 2014). The natural method would be to store the entire matrix; but it comes with a storage cost of $mn \gg r(m+n-r)$. This is due to the fact that this representation does not take advantage of the low-rank underlying structure. In this document, we will use the very natural SVD-like representation

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top \tag{5.6}$$

where $\mathbf{U} \in \mathbb{R}^{m\times r}$ is an orthogonal matrix ($\mathbf{U}^\top\mathbf{U} = \mathbf{I}_r$), $\mathbf{V} \in \mathbb{R}^{n\times r}$ is an orthogonal matrix and $\mathbf{\Sigma} \in \mathbb{R}^{r\times r}$ is a diagonal full-rank matrix. This formulation requires $r(m+n+1) \approx r(m+n-r)$ storage capacity and has the advantage of having two orthogonal matrices.

The Tangent Space

Each vector $\dot{\mathbf{X}}$ belonging to the tangent space $T_{\mathbf{X}}\mathcal{M}_r$ of $\mathcal{M}_r$ at $\mathbf{X}$ has a unique representation $(\dot{\mathbf{U}}, \dot{\mathbf{\Sigma}}, \dot{\mathbf{V}})$ such that (Vandereycken, 2013)

$$\begin{gathered}\dot{\mathbf{X}} = \mathbf{U}\dot{\mathbf{\Sigma}}\mathbf{V}^\top + \dot{\mathbf{U}}\mathbf{V}^\top + \mathbf{U}\dot{\mathbf{V}}^\top, \\ \mathbf{U}^\top\dot{\mathbf{U}} = \mathbf{0} \text{ and } \mathbf{V}^\top\dot{\mathbf{V}} = \mathbf{0}.\end{gathered} \tag{5.7}$$

This formulation also requires $r(m+n+1) \approx r(m+n-r)$ storage capacity.

Given a vector $\mathbf{Z}$ in the ambient space $\mathbb{R}^{m\times n}$, its projection on the tangent space $T_{\mathbf{X}}\mathcal{M}_r$ can be computed (Vandereycken, 2013) and is given by $\mathbf{P}_{T_{\mathbf{X}}\mathcal{M}_r}(Z)$, defined as

$$\mathbf{P}_{T_{\mathbf{X}}\mathcal{M}_r} : \mathbb{R}^{m\times n} \to T_{\mathbf{X}}\mathcal{M}_r : \mathbf{Z} \to \mathbf{P}_{\mathbf{U}}\mathbf{Z}\mathbf{P}_{\mathbf{V}} + \mathbf{P}_{\mathbf{U}}^{\perp}\mathbf{Z}\mathbf{P}_{\mathbf{V}} + \mathbf{P}_{\mathbf{U}}\mathbf{Z}\mathbf{P}_{\mathbf{V}}^{\perp},$$

with $\mathbf{P}_{\mathbf{U}} = \mathbf{U}\mathbf{U}^\top$ and $\mathbf{P}_{\mathbf{U}}^{\perp} = \mathbf{I} - \mathbf{U}\mathbf{U}^\top$, $\mathbf{P}_{\mathbf{V}}$ and $\mathbf{P}_{\mathbf{V}}^{\perp}$ being defined in the same way.

Using the tangent space representation, a basic identification yields the following representation for the (orthogonal) projection of an ambient vector $\mathbf{Z}$ onto $T_{\mathbf{X}}\mathcal{M}_r$:

$$\dot{\mathbf{\Sigma}} = \mathbf{U}^\top\mathbf{Z}\mathbf{V} \qquad \dot{\mathbf{U}} = (\mathbf{I} - \mathbf{U}\mathbf{U}^\top)\mathbf{Z}\mathbf{V} \qquad \dot{\mathbf{V}} = (\mathbf{I} - \mathbf{V}\mathbf{V}^\top)\mathbf{Z}^\top\mathbf{U}. \tag{5.8}$$

Retraction

The algorithm we will describe requires to be able to move along directions on the manifold. It is now well established (e.g. in Absil *et al.* 2008) that this can be cheaply achieved using a *retraction* instead of the expensive exponential map for instance, while keeping all convergence guarantees. We decided to use the projective retraction (Absil & Oseledets, 2014): given a vector $\dot{\mathbf{X}} \in T_{\mathbf{X}}\mathcal{M}_r$, it finds $\mathbf{Y} \in \mathcal{M}_r$ such that

$$\mathbf{Y} = R_{\mathbf{X}}(\dot{\mathbf{X}}) = \operatorname{argmin}_{\mathbf{Y}\in\mathcal{M}_r} \|\mathbf{X} + \dot{\mathbf{X}} - \mathbf{Y}\|_F.$$

The solution of this minimization problem is known to be the rank-r SVD of $\mathbf{X}+\dot{\mathbf{X}}$ (Eckart–Young theorem). Assuming $\mathbf{X}$ is given as (5.6) and $\dot{\mathbf{X}}$ as (5.7), it is possible to compute it efficiently (Vandereycken, 2013; Absil & Oseledets, 2014).

Vector Transport

Because $\mathcal{M}_r$ is embedded in $\mathbb{R}^{m\times n}$, a suitable vector transport (i.e., a mapping from the tangent space at some point to the tangent space at another point) is simply the projection of the ambient version of the original vector in the tangent space at the new point (Vandereycken, 2013).

5.5.2 Smoothing Techniques

The Main Idea

As explained earlier, the problem we aim to solve is the following

$$\min_{\mathbf{X}\in\mathcal{M}_r} \|\mathcal{P}_\Omega(\mathbf{X}-\mathbf{M})\|_{\ell_1} + \lambda\|\mathcal{P}_{\bar{\Omega}}(\mathbf{X})\|_{\ell_2}^2, \tag{5.9}$$

with the following interpretation: we ought to find a low-rank matrix $\mathbf{X}$ that fits the data $\mathbf{M}$ in the ℓ_1 sense on the mask Ω. On the remaining entries in $\bar{\Omega}$, we have a small confidence λ (typically between 0 and 10^{-5}-10^{-3}, even though this is application-dependent) that the value should be zero, hence we minimize the ℓ_2 error between $\mathbf{X}$ and 0 on $\bar{\Omega}$ [4]. This is motivated by previous studies of (Boumal & Absil, 2015) were regularization was especially useful to deal with real datasets. Note that the reason for the use of the ℓ_2 norm in the regularization term is twofold: first, the ℓ_2 norm will allow significant simplifications to be detailed in the forthcoming sections. Secondly, we can observe outliers on Ω, so the ℓ_1 norm makes sense there; but we obviously cannot observe outliers on $\bar{\Omega}$, so it does not seem necessary to use an ℓ_1 norm there, and an ℓ_2 norm should be more suitable.

The obvious main drawback of using (5.9) is that it is non differentiable. To remedy this problem, we decided to use smoothing techniques in order to make the objective differentiable. The idea is that, for a small $\delta > 0$, the following function

$$\sum_{(i,j)\in\Omega} \sqrt{\delta^2 + (X_{ij}-M_{ij})^2}$$

is a smooth approximation of

$$\|\mathcal{P}_\Omega(X-M)\|_{\ell_1},$$

as depicted on figure 5.2. The idea is thus to solve the following optimization problem

$$\min_{X\in\mathcal{M}_r} \sum_{(i,j)\in\Omega} \sqrt{\delta^2 + (X_{ij}-M_{ij})^2} + \lambda \sum_{(i,j)\in\bar{\Omega}} X_{ij}^2 \tag{5.10}$$

for decreasing values of δ.

To solve a problem in the form

$$\min_{x\in\mathcal{M}} f(x)$$

where $\mathcal{M}$ is a smooth Riemannian manifold and where f is smooth, there exist now many different techniques such as Riemannian conjugate gradient or trust-region algorithms (Absil *et al.*, 2008). We have opted for the conjugate gradient approach, as it appears to be more precise and efficient when δ becomes small.

[4] Note that this assumes that the entries in the matrix have a mean equal to zero.

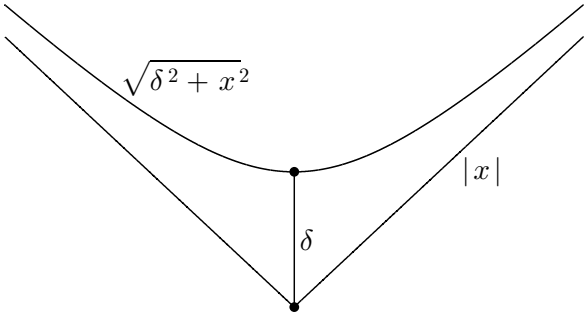


Figure 5.2: Illustration of both $|x|$ and $\sqrt{\delta^2 + x^2}$.

The Objective Function and its Gradient

Using a first-order algorithm like conjugate gradient requires the computation of the cost function and the (Riemannian) gradient.

Taking a look at (5.10), one may think that just evaluating the cost would require $\mathcal{O}(mn)$ operations, since we need to evaluate $\mathbf{X}$ over *both* Ω and $\bar{\Omega}$. Actually, as pointed out in (Boumal & Absil, 2011) this is not the case, since

$$\|\mathcal{P}_{\bar{\Omega}}(\mathbf{X})\|_{\ell_2}^2 = \|\mathbf{X}\|_{\ell_2}^2 - \|\mathcal{P}_{\Omega}(\mathbf{X})\|_{\ell_2}^2,$$

and, because we store $\mathbf{X}$ using the factorization $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, we can compute $\|\mathbf{X}\|_{\ell_2}^2$ easily thanks to the invariance of the Frobenius norm to orthogonal changes of basis :

$$\|\mathbf{X}\|_{\ell_2}^2 = \|\mathbf{\Sigma}\|_{\ell_2}^2,$$

which requires $\mathcal{O}(r)$ operations. We can then rewrite the cost function of (5.10) as

$$f_\delta(\mathbf{X}) = \sum_{(i,j)\in\Omega} \left(\sqrt{\delta^2 + (X_{ij} - M_{ij})^2} - \lambda X_{ij}^2 \right) + \lambda \|\mathbf{X}\|_{\ell_2}^2. \tag{5.11}$$

Hence, computing the cost function requires $\mathcal{O}(|\Omega| + r)$ operations, after having evaluated the product $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ on the mask Ω.

We can clearly see here the advantage of having added an ℓ_2 regularization term: the fact that $\|\mathbf{X}\|_{\ell_2}^2 = \|\mathbf{\Sigma}\|_{\ell_2}^2$ is crucial to avoid an $\mathcal{O}(mn)$ complexity in the computation of the objective function.

To compute the gradient, because $\mathcal{M}_r$ is embedded in $\mathbb{R}^{m\times n}$, we first need to compute the Euclidian gradient of f at $\mathbf{X}$ and then project it onto $T_X\mathcal{M}_r$. The Euclidian gradient is

$$\nabla f_\delta(\mathbf{X}) = \mathbf{S} + 2\lambda\mathbf{X}$$

where $\mathbf{S}$ is a *sparse matrix* defined as

$$S_{ij} = \begin{cases} \frac{X_{ij} - M_{ij}}{\sqrt{\delta^2 + (X_{ij} - M_{ij})^2}} - 2\lambda X_{ij} & \text{if } (i,j) \in \Omega, \\ 0 & \text{otherwise.} \end{cases}$$

The gradient is then the sum of a sparse component ($\mathbf{S}$) and a low-rank one ($2\lambda\mathbf{X}$). Then, projecting it onto $T_{\mathbf{X}}M_r$ can be done efficiently thanks to equation (5.8) and to the factorization

$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$. Indeed, we have

$$\begin{aligned}
\dot{\mathbf{\Sigma}} &= \mathbf{U}^\top(\mathbf{S} + 2\lambda\mathbf{X})\mathbf{V} \\
&= \mathbf{U}^\top\mathbf{S}\mathbf{V} + 2\lambda\mathbf{\Sigma}, \\
\dot{\mathbf{U}} &= (\mathbf{I} - \mathbf{U}\mathbf{U}^\top)(\mathbf{S} + 2\lambda\mathbf{X})\mathbf{V} \\
&= \mathbf{S}\mathbf{V} + 2\lambda\mathbf{U}\mathbf{\Sigma} - \mathbf{U}\mathbf{U}^\top\mathbf{S}\mathbf{V} - 2\lambda\mathbf{U}\mathbf{\Sigma} \\
&= \mathbf{S}\mathbf{V} - \mathbf{U}\mathbf{U}^\top\mathbf{S}\mathbf{V}, \\
\dot{\mathbf{V}} &= (\mathbf{I} - \mathbf{V}\mathbf{V}^\top)(\mathbf{S} + 2\lambda\mathbf{X})^\top\mathbf{U} \\
&= \mathbf{S}^\top\mathbf{U} + 2\lambda\mathbf{V}\mathbf{\Sigma} - \mathbf{V}\mathbf{V}^\top\mathbf{S}^\top\mathbf{U} - 2\lambda\mathbf{V}\mathbf{\Sigma} \\
&= \mathbf{S}^\top\mathbf{U} - \mathbf{V}\mathbf{V}^\top\mathbf{S}^\top\mathbf{U}.
\end{aligned}$$

Given the fact that $\mathbf{S}$ is sparse, these 3 terms can be computed efficiently. Note that the addition of the regularization parameter is cheap, since it only requires to modify the $\mathbf{S}$ matrix, and to add a small $r \times r$ matrix to $\dot{\Sigma}$.

The Algorithm

The full algorithm is stated in algorithm 4; the name RMC stands for "Robust Matrix Completion". Note that in the following, by outer and inner iteration we mean the δ and the CG loop, respectively.

We use a conjugate-gradient algorithm with a Hestenes-Stiefel modified rule (even though, after several experiments, we found that this choice does not really impact the algorithm) and an Armijo backtracking linesearch.

The starting point of the algorithm, $\mathbf{X}^{(0)}$, can be chosen simply using the rank-r SVD of $\mathcal{P}_\Omega(\mathbf{M})$. Suitable values for $\delta^{(0)}$ (the initial value for the smoothing parameter δ) are application-dependent, but for data $\mathbf{M}$ with values around unity, we use $\delta^{(0)} = 1$. Note that this value can have a significant impact on the solution final quality. The smoothing parameter is then updated using a geometric rule $\delta^{(k+1)} = \theta \cdot \delta^{(k)}$. A quite "aggressive" value of $\theta = 0.05$ gives good results in our synthetic experiments. In real applications this parameter has to be tuned to find the right value. For all experiments, ϵ is set to 10^{-8} (but again, this is application-dependent). The stopping criterion of the conjugate-gradient algorithm is set to a maximum of 40 iterations or a gradient norm of 10^{-8}, whichever is reached first.

5.5.3 Convergence Analysis

This section provides a basic convergence analysis of the RMC algorithm. Its goal is mostly to give sense to the stopping criterion of the outer loop, i.e., that the algorithm will terminate at some point (assuming exact arithmetic at least).

Theorem 2 (Strict decrease)**.** *If the sequence of iterates* $\{\mathbf{X}^{(0)}, \mathbf{X}^{(1)}, \mathbf{X}^{(2)}, \dots\}$ *is produced by algorithm 4 with* $\theta < 1$*, defining* $f^{(k)} = f_{\delta^{(k)}}(\mathbf{X}^{(k)})$*, we have*

$$f^{(0)} > f^{(1)} > \cdots > f^{(k)} > \cdots$$

Proof. At iteration k, the CG algorithm returns a feasible solution $\mathbf{X}^{(k)} \in \mathcal{M}_r$, associated with $\delta^{(k)}$. It is easy to see that $\mathbf{X}^{(k)}$ stays a feasible point for the next step (since it belongs

Algorithm 4 RMC

procedure RMC($\mathbf{X}^{(0)}, \delta^{(0)}, \theta, \epsilon, \lambda$)
  $f^{(0)} \leftarrow \infty$
  $k \leftarrow 0$
  $\delta^{(1)} \leftarrow \delta^{(0)}$
  $e \leftarrow \infty$
  while $e \geq \epsilon$ **do**
    Solve

$$\mathbf{X}^{(k+1)} = \text{argmin}_{\mathbf{X} \in \mathcal{M}_r} \sum_{(i,j) \in \Omega} \sqrt{(\delta^{(k+1)})^2 + (X_{ij} - M_{ij})^2} + \lambda \|\mathcal{P}_{\bar{\Omega}}(\mathbf{X})\|_{\ell_2}^2 \tag{5.12}$$

    using Riemannian Conjugate Gradient algorithm and $\mathbf{X}^{(k)}$ as a starting point.
    $f^{(k+1)} \leftarrow f_{\delta^{(k+1)}}(\mathbf{X}^{(k+1)})$
    $e \leftarrow f^{(k)} - f^{(k+1)}$
    $k \leftarrow k + 1$
    $\delta^{(k+1)} \leftarrow \delta^{(k)} \cdot \theta$
  end while
end procedure

to $\mathcal{M}_r$), and that

$$f_{\delta^{(k)}}(\mathbf{X}^{(k)}) > f_{\delta^{(k+1)}}(\mathbf{X}^{(k)}).$$

This follows from $\delta^{(k+1)} = \theta \cdot \delta^{(k)} < \delta^{(k)}$ and the expression (5.11) of $f_\delta(\mathbf{X})$. Then, because CG is a descent direction,

$$f_{\delta^{(k+1)}}(\mathbf{X}^{(k)}) \geq f_{\delta^{(k+1)}}(\mathbf{X}^{(k+1)}).$$

The claim follows from these two inequalities. □

Now, it is also easy to notice that, $\forall \delta \geq 0$ and $\forall \mathbf{X} \in \mathcal{M}_r$,

$$f_\delta(\mathbf{X}) \geq f(\mathbf{X}).$$

Hence, defining

$$f^* = \inf_{\mathbf{X} \in \mathcal{M}_r} f(\mathbf{X}) \geq 0,$$

we observe that the sequence of iterates $\{f_k\}_{k=1}^K$ is monotonically decreasing and bounded below by f^*.

This conclusion gives sense to the stopping criterion of the algorithm saying that it stops after iteration k if the difference between $f^{(k)}$ and $f^{(k+1)}$ is below some threshold ϵ: the algorithm will terminate at some point. We emphasize the fact that this does not prove that the algorithm converges towards a global minimum.

5.6 Numerical Results and Comparison of the Algorithms

In this section, we compare our three algorithms in term of convergence speed, solution quality and robustness.

5.6.1 Synthetic Experiments

On all experiments, synthetic artificial data are created in the following way: we build $\mathbf{U} \in \mathbb{R}^{m \times r}$ and $\mathbf{V} \in \mathbb{R}^{r \times n}$ with i.i.d. Gaussian-entries such that their product $\mathbf{M} = \mathbf{UV}$ is filled with zero-mean and unit-variance non independent Gaussian entries. We then sample $k = fr(m + n - r)$ entries uniformly at random, where f is the oversampling factor.

In some cases, we will add some non-Gaussian noise on part the observed entries to create outliers. To do so, we add one realization of the following random variable

$$\mathcal{O} = \mathcal{S}_{\pm 1} \cdot \mathcal{N}(\mu, \sigma^2)$$

where $\mathcal{S}_{\pm 1}$ is a random variable with equal probability to be equal to $+1$ or -1, while $N(\mu, \sigma^2)$ is a Gaussian random variable of mean μ and variance σ^2. Unless stated otherwise, outliers are created uniformly at random with some probability, usually 5%.

Different factors affect the task of matrix completion. Obviously, the size of the problem matters, and we will try to tackle large enough problems to show that our algorithm scales well. The oversampling factor f should also be greater than 1, and in the following experiments it will be fixed to either 4 or 5.

Let us denote by $\mathbf{M}_0$ the original low-rank matrix, without any outliers. We will monitor how the root mean square error (RMSE), defined as the error on *all* the entries between $\mathbf{X}$ and the *original* matrix $\mathbf{M}_0$

$$RMSE(\mathbf{X}, \mathbf{M}_0) = \sqrt{\frac{\sum_{i=1,j=1}^{m,n} (X_{ij} - M_{0,ij})^2}{mn}}$$

decreases. Since we have access to the factorization of both $\mathbf{X}$ and $\mathbf{M}_0$, this can be computed efficiently (see section 3.5 and Boumal & Absil, 2015). A decrease towards zero is what we expect from a robust matrix completion method, since our goal is to recover exactly (up to numerical errors) the original low-rank matrix, even in the presence of outliers.

We decided to compare our three algorithms (denoted by IRLS, ALMC and RMC) to both AOPMC (Yan *et al.*, 2013) and GRASTA (He *et al.*, 2011). We were unfortunately unable to find an efficient implementation of the algorithm described in (Nie *et al.*, 2012b), the implementation available requiring SVD of full $m \times n$ matrices.

For RMC, the maximum number of CG iterations (the inner loop) is set to 40 with a gradient tolerance of 10^{-8}. We use $\delta_0 = 1$, as well as $\theta = 0.05$.

For IRLS, the maximum number of iterations for each least-square problem, solved using RTRMC, is set to a quite "aggressive" value of only 5 (with an exception at the first iteration where this parameter is set to 20, to allows good convergence in problems without outliers), while the maximum number of iteration of the "outer loop" is set to 50. We use a value of $\delta = 10^{-5}$, which seems like a good compromise between precision and stability. Gradient tolerance is set to 10^{-8} throughout the execution of the algorithm, and we use $\epsilon = 10^{-8}$. We tried using the conjugate gradient method, but it seems to be slower and even less robust than trust-region. Note that, as we will see in the following, the parameters of IRLS are quite hard to determine accurately. In some experiments, these parameters work well, but on some other they do not.

ALMC uses the "orthogonalization" option, even though this does not seems necessary in these experiments. Yet, because the time spent in the ortogonalization is almost negligible,

we decided to use it anyway. The algorithm stops if the difference between two successive iterations is below 10^{-8}.

For AOPMC, we use the default settings with a maximum of 20 trust-region iterations at each outer iteration (i.e., when the mask Ω is fixed, with potentially some outliers removed) but a maximum of 20 iterations for the tCG algorithm (see (Boumal & Absil, 2011) for further information). Note that we use the code of the authors[5], but because we know the number of outliers, we decided to provide it to AOPMC. Otherwise, we would need to run the algorithm several times to guess the number of outliers. Also note that AOPMC automatically adjusts the number of iterations if it seems that the convergence is too bad. We did not change this option, so this may explain why the algorithm sometimes does more than 20 iteration between each update of the mask Ω.

Regarding GRASTA, we also use the implementation provided by the authors[6], with the default settings, but we had some troubles to run the algorithm on large $50\,000 \times 50\,000$ problems, since it was not even able to perform two complete sweeps over the data in less than 10 minutes (while RMC terminates in about 5 minutes in this situation). It seems to be due to the fact that the algorithm operates one column at a time, and since each rank-1 update of the $\mathbf{U}$ matrix takes about 0.01-0.02 second (using a standard but efficient MATLAB implementation), one loop over the 50 000 columns takes more than 10 minutes. Note that this can be easily understood, since the original goal of GRASTA is not to perform "batch" matrix completion, but rather *online* subspace tracking. For the 500×500 case, everything goes well and the algorithm converges in an amount of time comparable to RMC. The only modification made to the algorithm was to change the initial point that was set, like the other algorithms, to the left matrix of the rank-r SVD of the matrix $\mathcal{P}_\Omega(\mathbf{M})$ (instead of some complete random initialization). But it does not have a significant influence on the convergence of the algorithm.

On all figures, the large dots indicate a change in the outer-iteration: in RMC it indicates a decrease in δ, in IRLS it indicates an update of the weights while it indicates an update of the mask Ω in AOPMC.

Note that we tried, without success, to speed-up the RMC and AOPMC algorithms by terminating the inner loop sooner, when the decrease from iteration to iteration was small enough. Yet, it appears that solving the inner problems with a good enough quality (i.e., a small enough gradient norm) is crucial for the convergence of both algorithms towards the exact underlying low-rank matrix. For this reason, we left the gradient tolerance set to 10^{-8}.

Note that the experiments where run on quite large $50\,000 \times 50\,000$ matrices, in order to show that RMC scales well on large matrices. Some experiments were also run on $500\,000 \times 500\,000$ matrices of rank 10: the behavior of RMC was very stable and the running time appeared to be linear in the size of the matrix, i.e., in $\mathcal{O}(m+n)$. They are omitted for simplicity. Results on smaller matrices are qualitatively the same, except of course for the time the algorithm takes to reach the optimal value. Regarding IRLS and ALMC, things were more complicated, since they are already quite slow and/or unstable on $50\,000 \times 50\,000$ matrices. We did not try these methods on $500\,000 \times 500\,000$ problems.

All experiments where run on a 6 core Intel Xeon CPU E5-1650 v2 at 3.50GHz with 64 Go of ram using MATLAB R2014a on a 64 bit Linux machine. We used the MANOPT toolbox (Boumal *et al.*, 2014) (version 1.0.7) to handle the optimization part of the RMC algorithm

[5] Available at `https://binary-matching-pursuit.googlecode.com/files/AOPMCv1.zip`.

[6] Available at `https://sites.google.com/site/hejunzz/grasta`.

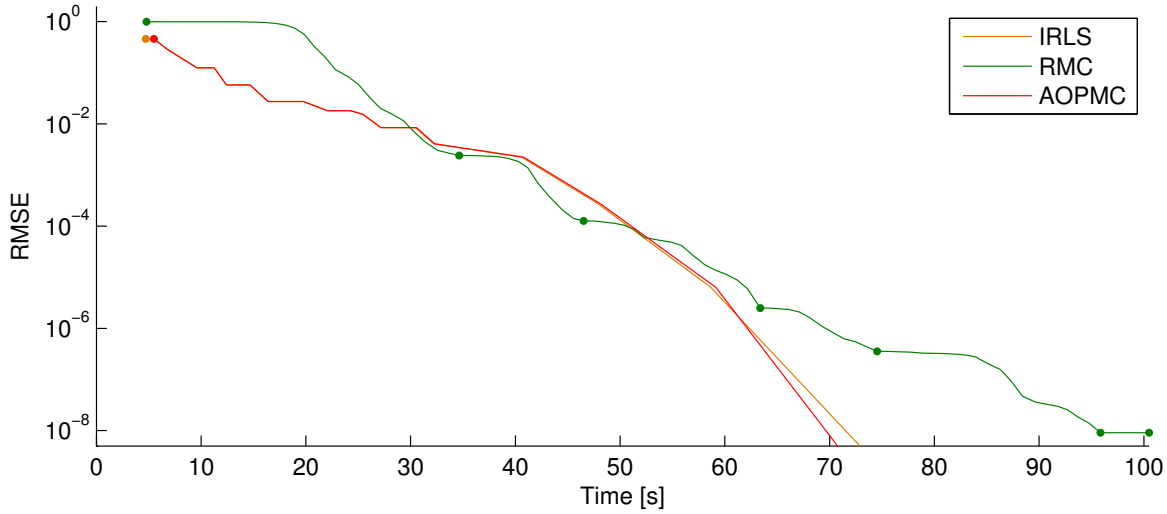


Figure 5.3: Perfect low-rank matrix completion: low-rank matrix completion of a rank-10 $50\,000 \times 50\,000$ matrix observed with an oversampling of 4. The decrease in the objective function of RMC from iteration to iteration is clear. In this example, it struggles to significantly decrease the RMSE at the end since the function becomes less and less differentiable near the "kinks" of the absolute values, where the solution takes place. AOPMC and IRLS do not have this problem, and converges well towards 0.

with the `fixedrankembeddedfactory` manifold factory and the default conjugate gradient method. IRLS uses RTRMC (version 3.1) and the same version of MANOPT ; ALMC uses GUROBI (`www.gurobi.com`) 5.6.3 to solve the LP's.

Perfect Low-Rank Matrix Completion As a sanity check, we test our algorithm on the very simple *perfect matrix completion* (matrix completion without any noise nor outliers) problem using a $50\,000 \times 50\,000$ matrix of rank 10. Results are depicted on figure 5.3. Exception made of ALMC and GRASTA, all algorithms successfully recover the underlying low-rank matrix. Note that the convergence of IRLS and AOPMC are almost the same since in their first iteration they both use RTRMC with a uniform weight matrix $\mathbf{C}$. ALMC does not even make one iteration. This is the main drawback of this algorithm: it is significantly slower than the other. As explained in section 5.4.1, we need to solve, at each iteration, thousands of LP's with, roughly $(2f+1)r \approx 100$ variables and $4fr = 160$ constraints. Even though this may seem easy because they all quite small, a careful analyze of the algorithm shows that most of the time is consumed inside the `gurobi` function at solving the LP's. GRASTA is too slow as well, and does not make one sweep over the data before 10 minutes. As explained earlier, this is due to the time each rank one update takes.

Low-Rank Matrix Completion with Outliers Given a 500×500 matrix for which we observe the entries uniformly at random with an oversampling of 4, we perturbed 5% of the observed entries by adding to them some non-Gaussian noise to create outliers. This problem would be cumbersome to solve with an ℓ_2 method because of the high weights the outliers would have in the objective function.

When running our algorithms, we obtain the results depicted on figure 5.4(a) for outliers created using $\mu = \sigma = 0.1$ and on figure 5.4(b) using $\mu = \sigma = 1$.

All the algorithms successfully solve the first problem, but we can see that GRASTA starts to have difficulties solving the second one. We suspect that this is due to the fact that GRASTA still uses an ℓ_2 norm in the algorithm (because of the augmented Lagrangian method), so the strength of the outliers still have some impact on the convergence. AOPMC seems to handle better the outliers, perhaps because the adaptive mask allows it to completely remove their impact from the objective function at the end. IRLS solves both problems well, but we can see that it seems more sensitive to the outliers strengh than AOPMC and RMC. It is interesting to note that ALMC does not seem to be slowed down by the fact that outliers are stronger in the second experiment. The convergence curve is almost indistinguishable from the first one. We believe this is due to the fact that solving the LP's is very stable numerically and is not affected at all by the outlier's strenght.

We then run the same experiment on larger $50\,000 \times 50\,000$ matrices, with still 5% of outliers. Figure 5.5(a) and 5.5(b) illustrate the results of these experiments, with $\mu = \sigma = 1$ and $\mu = \sigma = 5$ respectively, using an oversampling of 5.

We can see that both AOPMC and RMC solve the first problem well. GRASTA, on the other hand, does not converge in a decent amount of time. We also observe that RMC stays very robust when the strength of the outliers increases, while AOPMC starts to have important difficulties in the second experiment. The robustness of RMC may be due to the asymptotic linear behavior of the cost function, even in the first iterations. IRLS seems quite "unstable", in a sense that the matrix, his size, etc. seem to have some influence on his convergence. As expected, ALMC is too slow and has no chance of solving this problem in a decent amount of time compared to the other methods.

Note that this is a quite extreme experiment, in a sense that the outliers have a mean (absolute) amplitude of 5, while the entries have mean (absolute) value of 1. Yet, it demonstrates the robustness of RMC. Also note that the oversampling has a significant importance in this experiment, as an oversampling of 4 seems to make things harder for RMC: in this case, the RMSE stagnates around $10^{-4} - 10^{-5}$.

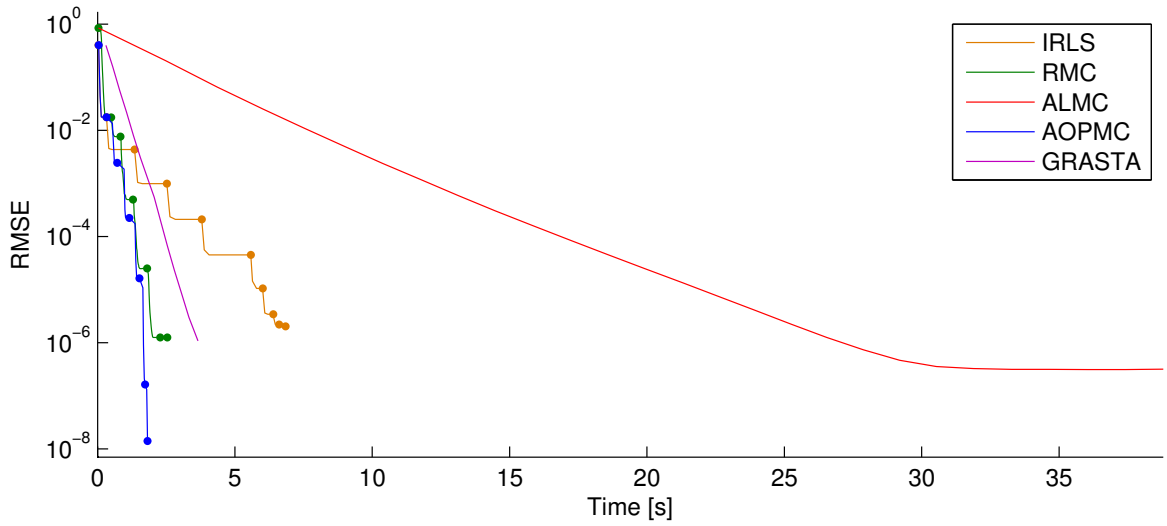
In the following experiments, we present some results using RMC only, since it clearly appears to be the best method among the one we compared.

Noisy Low-Rank Matrix Completion with Outliers In this experiment, we try to tackle the important problem of matrix completion in the presence of *both* (dense) noise and (sparse) outliers. Outliers are defined as previously, while noise is the addition, at each observed entry, of a zero-mean Gaussian random variable with variance σ_N^2.

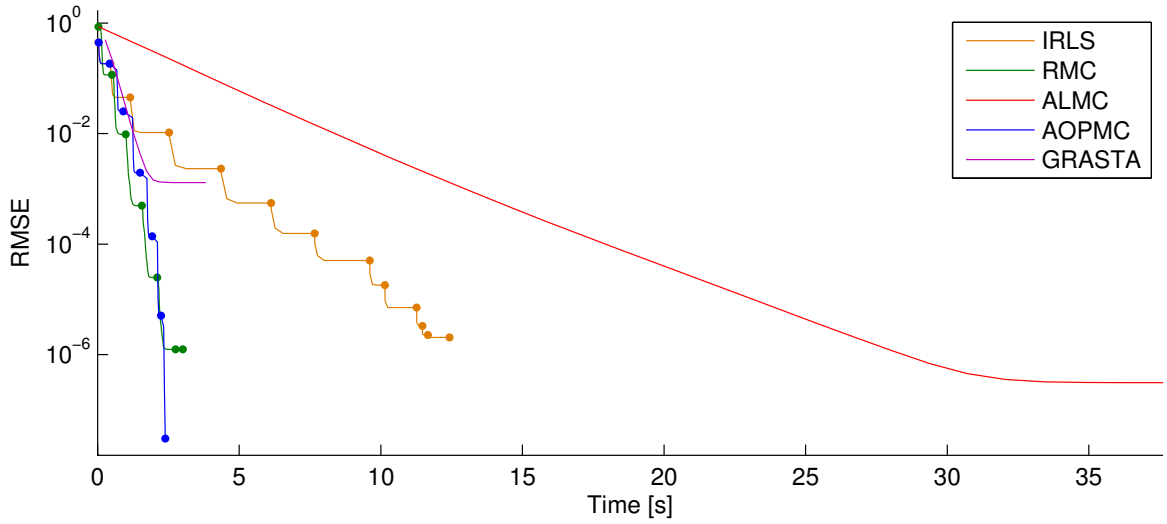
To have a point of comparison, we compare the results using RMC to the performances an oracle knowing the row and column space of $\mathbf{M}_0$, and returning the best matrix using this information, would give. If the entries are perturbed by Gaussian noise (without outliers) with variance σ_N^2, the best RMSE is equal (in expectation) to (Candes & Plan, 2010)

$$\mathrm{RMSE}_{\mathrm{Oracle}} = \sigma_N \sqrt{\frac{2nr - r^2}{|\Omega|}}$$

for the low-rank completion of an $n \times n$ matrix *with an ℓ_2 method.*

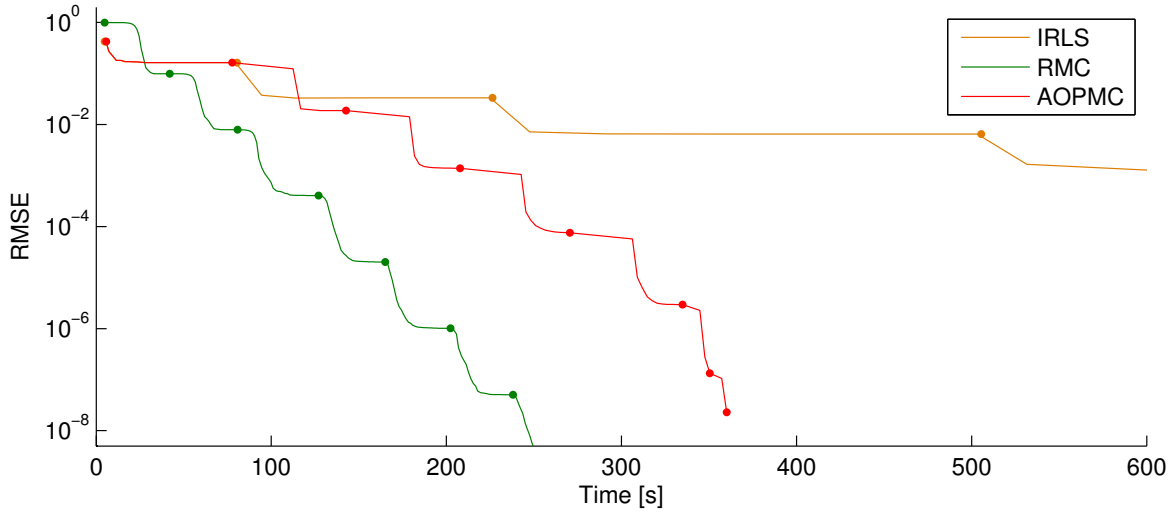


(a) $\mu = \sigma = 0.1$

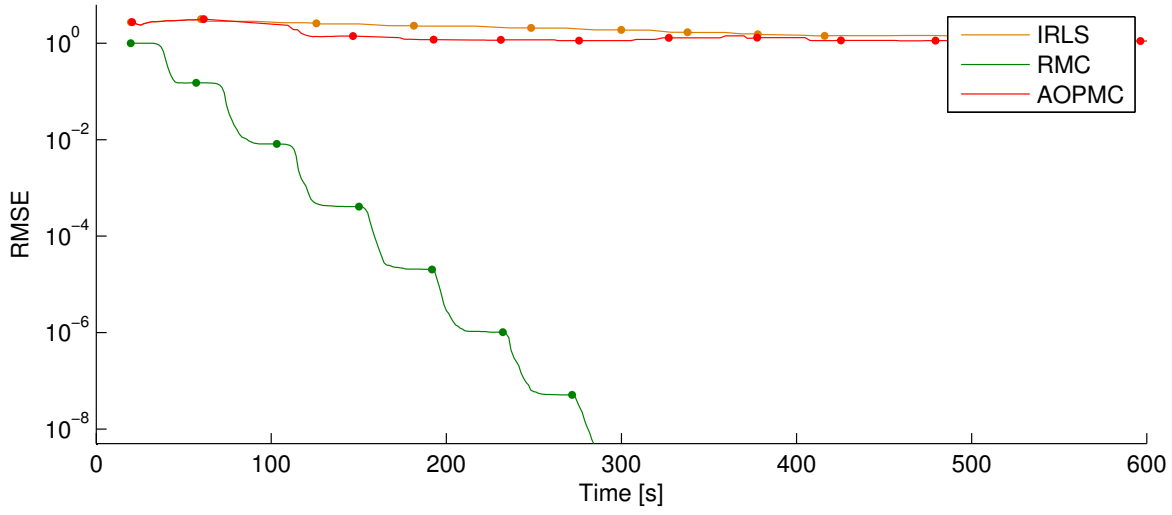


(b) $\mu = \sigma = 1$

Figure 5.4: Low-rank matrix completion with outliers: robust low-rank matrix completion of rank-10 500×500 matrices observed with an oversampling of 4 and with 5% outliers in the observed entries.



(a) $\mu = \sigma = 1$



(b) $\mu = \sigma = 5$

Figure 5.5: Low-rank matrix completion with outliers: robust low-rank matrix completion on $50\,000 \times 50\,000$ matrices of rank 10 with an oversampling of 5 and 5% outliers in the observed entries.

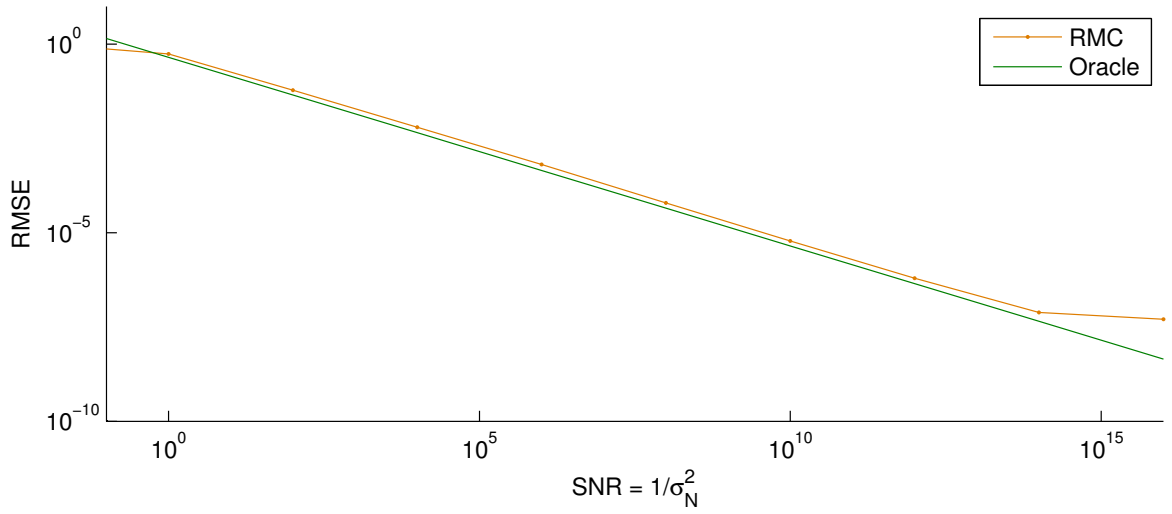


Figure 5.6: Noisy low-rank matrix completion with outliers: Evolution of the RMSE with respect to the signal-to-noise ratio SNR $= \frac{1}{\sigma_N^2}$ on a $5\,000 \times 5\,000$ matrix of rank 10 with an oversampling of 5 and 5% outliers created using $\mu = 1$ and $\sigma = 1$. Noise is the addition of i.i.d. Gaussian variables $\mathcal{N}(0, \sigma_N^2)$.

Figure 5.6 depicts the RMSE at termination of RMC with respect to the signal-to-noise (SNR) ratio. Results are the average of 3 successive experiments. Entries in the matrix $\mathbf{M}$ are such that the matrix has unit-variance Gaussian entries. We thus have SNR $= \frac{1}{\sigma_N^2}$.

We clearly see that—even in the presence of 5% of outliers—the algorithm successfully recovers the original low-rank matrix with an error proportional to the noise level. As long are the noise level is not too high, we have performances very similar to those of the oracle bound. For a high level of noise (SNR < 1), we see that the algorithm is slightly better than the oracle bound. This can be due to the ℓ_1 objective function which helps reduce the effect of the high variance. For a low level of noise, with a SNR greater than 10^{14}, the algorithm begins to have numerical difficulties to drive the RMSE towards zero because the objective function becomes less and less differentiable near the "kinks" of the absolute values. This is the same effect as in the previous experiments where the RMSE begins to stagnate around $10^{-8} - 10^{-6}$ due to the non-differentiability of the objective function near the solution, which happens to be exactly at the non-differentiable part of the objective function. A moderate and high level of noise has the effect of "smoothing" the objective function since the solution starts to deviate from the kinks of the ℓ_1 norm.

Evolution with the Number of Outliers It is now interesting to ask the following: is all of this true for all percentages of outliers ? Does our algorithm recover the original matrix for all levels of outliers ? Of course this is not the case but figure 5.7 shows how the RMSE at termination evolves when either the percentage of outliers added or the outliers strength increases. This figure is obtained after the average (at each point on the plot) of 3 experiments, aiming at the completion of a $5\,000 \times 5\,000$ rank-10 matrix observed with an oversampling of 4 (note the triple-log scale).

Three things are worth noting on this figure. First, the abrupt change in the RMSE around

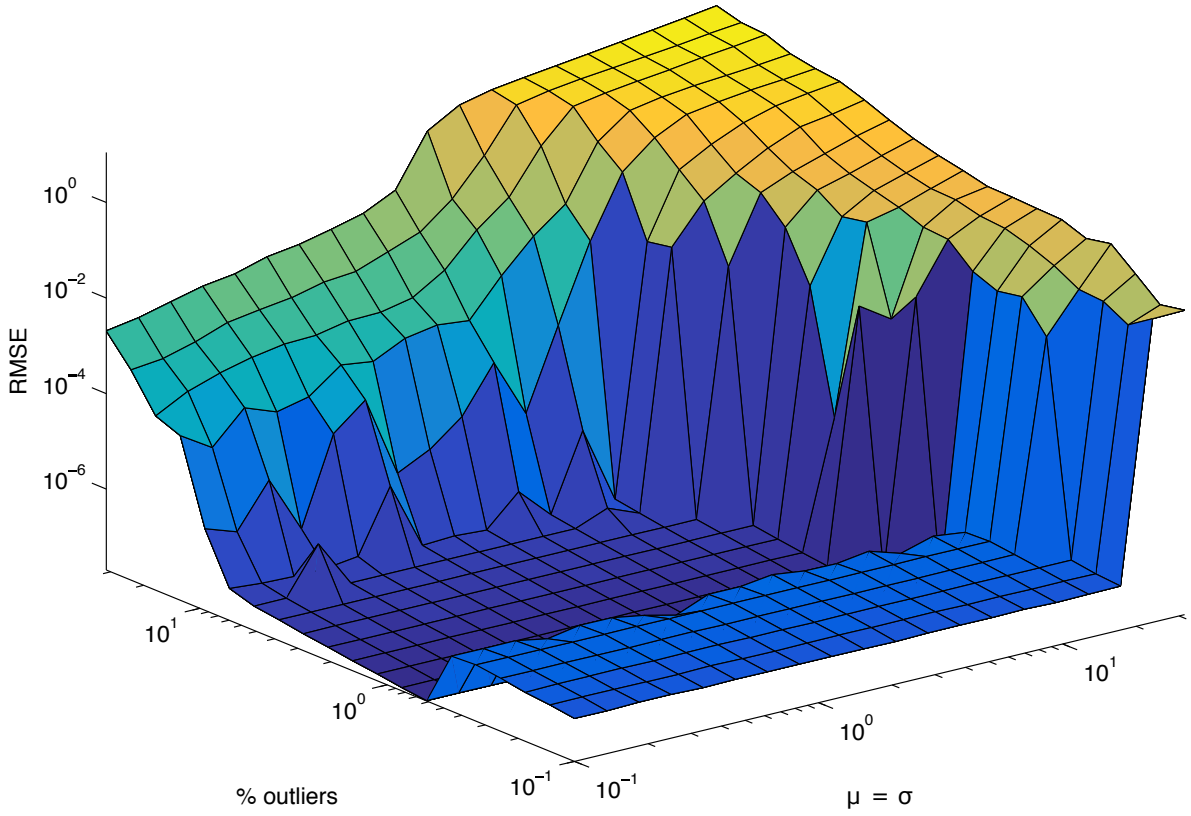


Figure 5.7: Evolution with the Number of Outliers: evolution of the RMSE with respect to the percentage of outliers and their strength on a $5\,000 \times 5\,000$ matrix of rank 10, observed with an oversampling of 4 and using $\mu = \sigma$.

0.5% of outliers is purely numeric. The reason is that the slight increase of the percent of outliers allows the method to converge better: the algorithm succeeds in decreasing $\delta^{(k)}$ one more time (without the conjugate gradient stalling) and can then decrease the RMSE even more.

Secondly, we notice a strong increase in the RMSE around 6-8% of outliers. This is quite low but can be explained by the small oversampling ratio used: a low percentage of outliers can reduce the number of healthy entries below the minimum required number. Still, this is interesting, as it shows that as long as the number of outliers is small enough, the recovered matrix stays almost exactly the same as the original unperturbed matrix.

Thirdly, it is interesting to note that the strength of the outliers, from $\mu = \sigma = 0.1$ to $\mu = \sigma \approx 10$, has a negligible impact on the quality of the final solution (remember entries have values around unity) as long as it remains low enough. This is in complete opposition with the previous experiment on noisy matrix completion, where the RMSE is clearly proportional to the level of the dense Gaussian noise

5.6.2 Conclusions

From these experiments, it seems clear that the method that combines both robustness (to strong outliers) and speed is RMC. By combining elementary smoothing techniques and Riemannian optimization, we built a method that can handle large-scale robust matrix completion tasks, even in the presence of both noise and outliers, which has a significant importance in practice.

ALMC gives quite good solutions and seems very robust to the outliers strength. But it has the major drawback of being extremely slow. In our opinion, this is simply due to the fact that it still need to solve *many* small LP's. Even though this is much easier than one large LP, it is still not efficient enough to compete with RMC for instance.

IRLS sometimes converges well but seems quite unstable. We did our best to find good parameters but it does not seem very robust (for instance, to the changes in the problem size, the strength of the outliers, etc.). In particular, the maximum number of iterations of the trust region algorithm is a critical parameter that is not that easy to tune. It should be high enough so that the trust-region algorithm can start to decrease the cost, but not too high, since in this case the algorithm can easily spend a lot of time in one iteration without updating the weights.

When comparing our algorithms to AOPMC and GRASTA, we found that AOPMC was often a quite good competitor, even though it clearly seems less robust to the strength of the outliers. GRASTA gives good results on small problems, with small outliers, but when the problem size increases, it clearly becomes too slow to compete with RMC or AOPMC on *batch* matrix completion problems.

In conclusion, our best method so far clearly is RMC. For this reason, this is the method we will use in the next chapter when we tackle low-rank matrix completion on real datasets.

6 | Applications

6.1 Recommender Systems

One of the most emblematic uses of Low-Rank Matrix Completion is in recommender systems and in particular in the Netflix Prize (Bennett & Lanning, 2007). In this section, we propose to test our algorithm RMC on this real-world dataset.

The problem is the following: Netflix, a movie rental company, wants to recommend movies to its users. To do so, they have a limited database of known ratings provided by some users themselves. In practice, this translates into the completion of a large $480\,189 \times 17\,770$ matrix, where columns correspond to movies, rows to user, and each entry is the rating of the movie represented by an integer from 1 to 5. To train the algorithm, we have 99 072 112 entries revealed; that is, approximately 1% of the entries are known. We then test our model on another set of 1 408 395 entries. Note that all values are shifted towards zero by subtracting the mean of the revealed entries (3.604), since our (regularized) model assumes a mean value of 0.

To assess the results, we use the root mean square criterion, i.e.,

$$\mathrm{RMSE}_{\mathrm{test}} = \sqrt{\frac{\sum_{(i,j)\in \mathrm{TestSet}} (X_{ij} - M_{ij})^2}{|\mathrm{TestSet}|}},$$

on the *test set* (i.e., the unrevealed entries). Returning the mean ratings as a prediction leads to a test RMSE of 1.13, while the winner of the Netflix prize, the BellKor's Pragmatic Chaos algorithm (Netflix, 2009), reached an RMSE of 0.8567, using a combination of many different techniques. Boumal & Absil (2015) performed extensive comparisons between different low-rank matrix completion algorithms (all minimizing the ℓ_2 norm on the training set). We can observe that the best result was obtained using LMaFit (Balzano *et al.*, 2010), reaching a test RMSE of 0.955.

Two parameters have a significant influence on the results. The rank "dictates" the complexity of the solution; the regularization parameter λ is used to limit overfitting, i.e., to avoid fitting too well the known entries while deviating too much from the mean on the unknown entries. To study the impact of both parameters, we picked a reference point with $r = 10$ and $\lambda = 8 \cdot 10^{-4}$ and we then changed the two parameters around these two values, one at the time. Note that the rank was chosen arbitrarily, while the value of λ is the one that gives the best results for a rank of 10. Also note that, after a few trials, we found that in this context, iteratively decreasing the δ parameter was not particularly useful. We then decided to fix it to 1. This value might seems high, but the smooth version is already a quite good approximation of the ℓ_1 norm. The number of iterations was limited to 100, and the gradient tolerance was set

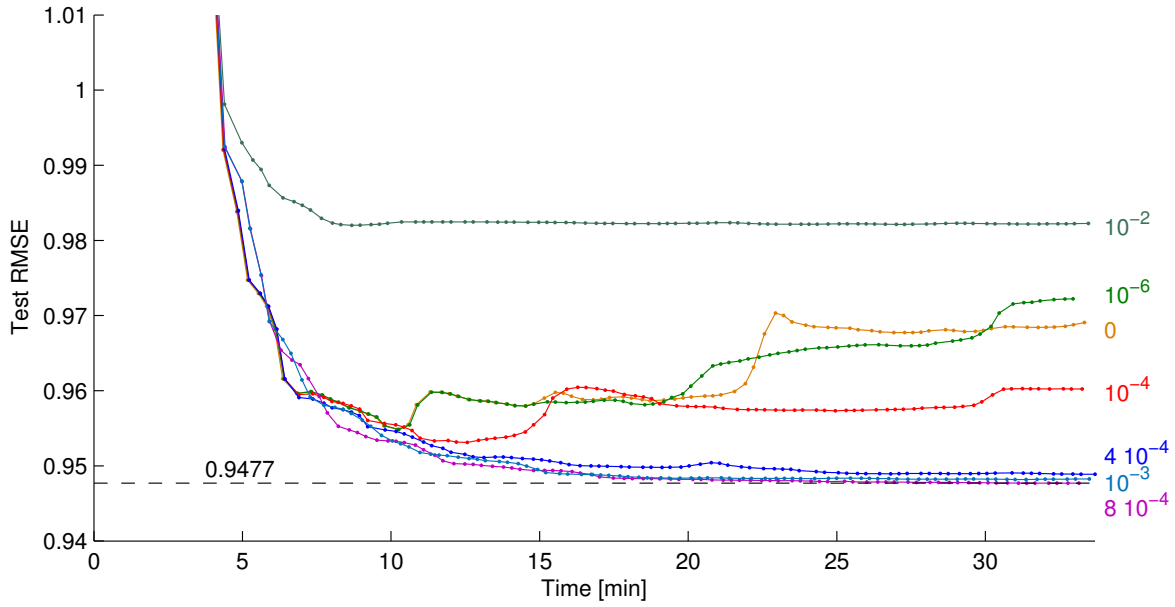


Figure 6.1: Evolution of the test RMSE on the NETFLIX dataset for different values of λ (displayed on the right) using a rank $r = 10$. It is clear from this experiment that the regularization λ play an important role. By tuning it to the right value, we obtain a quite good test RMSE of 0.9477

to 10^{-8}. In practice, most of the runs did not reach a gradient tolerance of 10^{-8} and were interrupted after 100 iterations.

Figure 6.1 depicts the evolution of the test RMSE for different values of λ. We can easily see that the non-regularized algorithm reaches a reasonable test RMSE but then tends to overfit. Increasing the regularization parameter λ around 10^{-3}-10^{-4} allows to find a good compromise between training error and overfitting. By increasing it even more, the test RMSE eventually stagnates, i.e., the regularization is so strong that it does not really fit anything except the mean value (see the $\lambda = 10^{-2}$ curve for instance).

Figure 6.2 illustrates how the rank plays a significant role in the solution quality. From this plot, it seems that the choice $r = 10$ was the right one, since other values for r give higher results. As underlined in (Boumal & Absil, 2015), choosing a high rank from the beginning does not seem to be the best choice and rank increasing methods may be worth investigating.

We can conclude from these experiments that our algorithm performs well on the NETFLIX dataset since it slightly outperforms the low-rank matrix completion algorithms that use the ℓ_2 norm. This may be due to the fact that our method is robust to outliers: this can help to reduce overfitting (even with no regularization), hence leading to a better overall model that better fits the test set.

Still, it is known that to reach better test RMSE, it is useful to combine this idea of low-rank matrix completion with other techniques. For instance, temporal effects have a large impact, as explained in (Bennett & Lanning, 2007): movies become more or less popular over time, user tastes and ratings can change over time, and so on. Neighborhood models, where users and movies are aggregated into groups of similar profiles, also help in decreasing the test RMSE (Bennett & Lanning, 2007). Yet, our algorithm proved itself to be quite efficient and

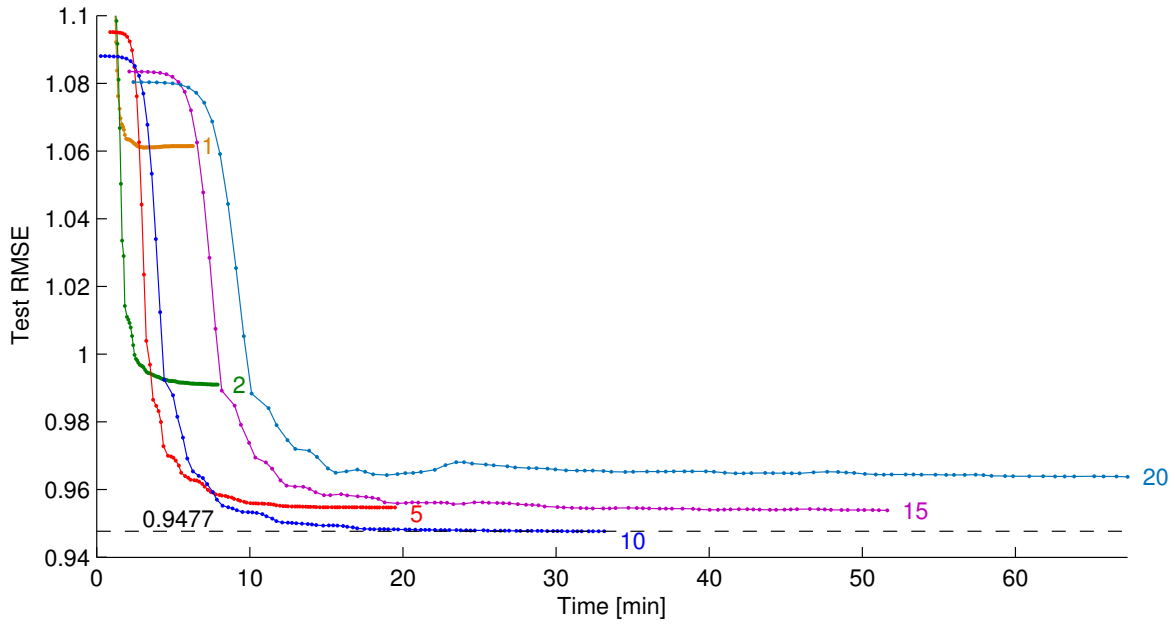


Figure 6.2: Evolution of the test RMSE on the NETFLIX dataset for different values of the rank r using $\lambda = 8 \cdot 10^{-4}$. It seems that the choice $r = 10$ is the right one, and that increasing the rank leads to overfitting.

may certainly be used as a good building block for a more complex algorithm. It should also be possible to better fine tune each parameter of the algorithm to find an even better optimal configuration.

6.2 Robust Structured Image Inpainting

We here try to tackle the problem of *robust image inpainting*. This problem is the following: given an image, with a lot of pixels missing, can we recover the original picture ? In general, the answer would be no: we at least need some hypothesis on the underlying original picture to recover it.

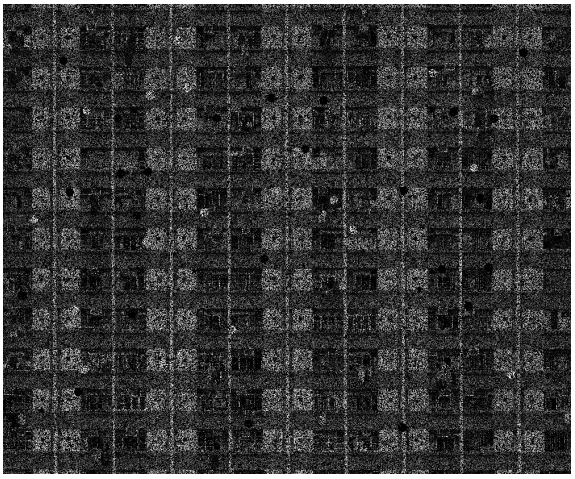
In this section, we assume that the underlying picture is low-rank, or has at least an underlying low-rank structure. Of course, this does not always make sense in applications, since a lot of pictures are *not* low-rank. Still, it makes sense if the picture presents some structure with repeating rectangular patterns for instance.

The 827×1000 picture to be used in this section (fig. 6.3(a)) depicts a crowded Asian building, where the image suits well the low-rank property: the repetition of the windows pattern makes the building low-rank. But this low-rank structure is clearly perturbed by a lot of details like towels and clothes hanging from the windows, A/C units, etc.

Because of that, this picture is actually "low-rank plus sparse", in the sense that the building is the low-rank component while the multiple details create a sparse perturbation. In this context, a robust low-rank matrix completion algorithm should be able to recover the low-rank background, i.e., the building. Note that we also added, by hand, around 35 large white and

black dots to make the low-rank matrix completion tasks a little bit harder.

(a) Original picture, perturbed by hand with 35 large white and black dots.



(b) Perturbed picture, with 45% observed pixels, including 10% of strong outliers.

(c) Recovered rank-30 picture using RMC

(d) Recovered rank-30 picture using the ℓ_2 algorithm RTRMC ($\lambda = 5 \cdot 10^{-2}$, with data shifted by substrating the mean of 107)

Figure 6.3: Robust image inpainting of a low-rank plus sparse picture, where the background building is low-rank and the multiple details create a "sparse" perturbation component.

Figure 6.3(b) illustrates the same picture with around 55% of unknown pixels (in black on the image), as well as 10% of outliers (created using $\mu = 75$ and $\sigma = 50$ — remember the entries of the image have values from 0 to 255).

After running the RMC algorithm (with $\theta = 0.1$, $\delta^{(0)} = 1$, $\epsilon = 10^{-6}$ and a maximum of 40 CG iterations for each value of δ), we obtain the result depicted in figure 6.3(c), using a rank of 30 and no regularization ($\lambda = 0$). In this case, it is clear that a low-rank matrix completion algorithm successfully extracts the underlying low-rank structure of the image, leaving aside all small local perturbations as well as (most) large white and black dots. The few remaining ones appear in area where there are not enough data available to properly reconstruct the image.

Figure 6.3(d) on the other hand presents the result of the RTRMC algorithm on the same

(a) Zoom on the original picture (with artificial black and white dots).

(b) Zoom on the recovered picture using RMC

Figure 6.4: Two thumbnails of the original and recovered image, clearly showing that RMC successfully recovers the low-rank component, leaving aside all sparse details.

picture, with a rank of 30. We did our best to find the optimal value of λ, and a value around $\lambda = 5 \cdot 10^{-2}$ seems to give the best result. Note that the data were shifted towards zero first.

This experiment tends to show that an ℓ_1 method seems more robust in this particular problem where the goal really is to extract the low-rank "background" from the image, for which the ℓ_1 norm minimization is a natural model. This is particularly clear when comparing figure 6.4(a) and figure 6.4(b), two thumbnails of respectively the original and the recovered image. The ℓ_2 norm model, on the other hand, seems to really fit *all* the details of the image including both the low-rank component as well as the sparse details. This leads to a quite bad reconstruction with a lot of noise.

Conclusions

In this thesis, we tackled the problem of low-rank matrix completion.

After having stated the problem, as well as having introduced the tools of Riemannian optimization, we successively studied two quite different problems.

In the first part, the goal was to improve the existing RCGMC algorithm. In chapter 3, we introduced a special type of conjugate gradient algorithm. This method, as opposed to other conjugate gradient algorithms, is known to have global convergence properties. In addition, experiments show that this new method is quite efficient, at least on well conditioned problems. In this situation, it can significantly speed up the existing algorithm.

We then developed, in chapter 4, a parallel version of RCGMC. This task required us to, first, precisely understand what needs to be computed and how to do it efficiently. For instance, we had to be careful to store the results correctly in the computer in order to allow for efficient computations. The second task was to properly implement the different parts of the algorithm. To be able to precisely parallelize the most expensive operations, the idea was to perform everything using compiled C-mex files. Doing so, we reached, using 6 cores and very large matrices, a quite good speedup of almost 5. In future work, one could attempt to build a parallel version of RTRMC. This would require parallelization of the Hessian.

The second part of this thesis was dedicated to the significant problem of *robust* low-rank matrix completion. This is an important problem since real datasets often contain outliers. To tackle this problem, we developed three different methods that we compared to two existing algorithms. These experiments showed that one of our algorithm that combines both Riemannian optimization and smoothing techniques appears to be an excellent competitor of existing algorithm: it both scales well when the size of the matrix increases and can easily solve the low-rank matrix completion problem in the presence of both dense additive noise and strong sparse outliers.

The part on smoothing techniques seems to be quite promising for the future: this work is one of the first in the area of *non-smooth* Riemannian optimization. Indeed, our method really is a non-smooth optimization method on manifold. Different research directions can be investigated. One could try to find better smoothing techniques; it could also be interesting to investigate the decrease of δ: is it possible to find a way such that, for instance, the $\mathbf{X}^{(k)}$ stays in the quadratic convergence area, so that an algorithm like trust-regions or the Newton's method would converge faster ? It could also be interesting to apply this method to other manifolds, to see if the good convergence results are conserved.

Finally, we used our best algorithm, RMC, on two real datasets. On the Netflix dataset, our method appeared to be slightly better than other low-rank matrix completion algorithms. We then used RMC to successfully recover badly damaged images. Using a suitable "low-

rank plus sparse" image representing a background building with a lot of sparse details, we recovered well the low-rank part of the image, i.e., the building.

In conclusion, low-rank matrix completion is a challenging task. But there exist nowadays suitable, fast and robust algorithms to solve this problem.

The codes used to create all the figures of this thesis can be downloaded from `http://baemerick.be/lcambier`.

Bibliography

Absil, P-A, & Oseledets, Ivan V. 2014. Low-rank retractions: a survey and new results. *Computational Optimization and Applications*, 1–25.

Absil, P-A, Mahony, Robert, & Sepulchre, Rodolphe. 2008. *Optimization algorithms on matrix manifolds*. Princeton University Press.

Aftab, Khurrum, & Hartley, Richard. 2015. *Convergence of Iteratively Re-weighted Least Squares to Robust M-estimators*.

Balzano, Laura, Nowak, Robert, & Recht, Benjamin. 2010. Online identification and tracking of subspaces from highly incomplete information. *Pages 704–711 of: Communication, Control, and Computing (Allerton), 2010 48th Annual Allerton Conference on*. IEEE.

Bennett, James, & Lanning, Stan. 2007. The netflix prize. *Page 35 of: Proceedings of KDD cup and workshop*, vol. 2007.

Boumal, Nicolas, & Absil, P.-A. 2015. Low-rank matrix completion via preconditioned optimization on the Grassmann manifold. *Linear Algebra and its Applications*, **475**(0), 200 – 239.

Boumal, Nicolas, & Absil, Pierre-antoine. 2011. RTRMC: A Riemannian trust-region method for low-rank matrix completion. *Pages 406–414 of: Advances in neural information processing systems*.

Boumal, Nicolas, Mishra, Bamdev, Absil, P.-A., & Sepulchre, Rodolphe. 2014. Manopt, a Matlab Toolbox for Optimization on Manifolds. *Journal of Machine Learning Research*, **15**, 1455–1459.

Burke, James V. 2014. *Class notes for MATH 408, Linesearch Methods*.

Candes, Emmanuel J, & Plan, Yaniv. 2010. Matrix completion with noise. *Proceedings of the IEEE*, **98**(6), 925–936.

Candès, Emmanuel J, & Recht, Benjamin. 2009. Exact matrix completion via convex optimization. *Foundations of Computational mathematics*, **9**(6), 717–772.

Candès, Emmanuel J, Li, Xiaodong, Ma, Yi, & Wright, John. 2011. Robust principal component analysis? *Journal of the ACM (JACM)*, **58**(3), 11.

Chen, Yudong, Xu, Huan, Caramanis, Constantine, & Sanghavi, Sujay. 2011. Robust matrix completion with corrupted columns. *arXiv preprint arXiv:1102.2254*.

Chen, Yudong, Jalali, Ali, Sanghavi, Sujay, & Caramanis, Constantine. 2013. Low-rank

matrix recovery from errors and erasures. *Information Theory, IEEE Transactions on*, **59**(7), 4324–4337.

Chistov, Alexander L, & Grigor'ev, D Yu. 1984. Complexity of quantifier elimination in the theory of algebraically closed fields. *Pages 17–31 of: Mathematical Foundations of Computer Science 1984*. Springer.

Dai, Yu-Hong, & Yuan, Yaxiang. 1999. A nonlinear conjugate gradient method with a strong global convergence property. *SIAM Journal on Optimization*, **10**(1), 177–182.

Drineas, Petros, Javed, Asif, Magdon-Ismail, Malik, Pandurangan, Gopal, Virrankoski, Reino, & Savvides, Andreas. 2006. Distance matrix reconstruction from incomplete distance information for sensor network localization. *Pages 536–544 of: Sensor and Ad Hoc Communications and Networks, 2006. SECON'06. 2006 3rd Annual IEEE Communications Society on*, vol. 2. IEEE.

Hager, William W, & Zhang, Hongchao. 2006. A survey of nonlinear conjugate gradient methods. *Pacific journal of Optimization*, **2**(1), 35–58.

Hastie, Trevor. 2012. Matrix Completion and Large-scale SVD Computations.

He, Jun, Balzano, Laura, & Lui, John. 2011. Online robust subspace tracking from partial information. *arXiv preprint arXiv:1109.3827*.

Kennedy, Ryan, Balzano, Laura, Wright, Stephen J, & Taylor, Camillo J. 2014. Online algorithms for factorization-based structure from motion. *Pages 37–44 of: Applications of Computer Vision (WACV), 2014 IEEE Winter Conference on*. IEEE.

Klopp, Olga, Lounici, Karim, & Tsybakov, Alexandre B. 2014. Robust Matrix Completion. *arXiv preprint arXiv:1412.8132*.

Lee, John. 2003. *Introduction to smooth manifolds*. Vol. 218. Springer Gradate Texts in Mathematics.

Li, Xiaodong. 2013. Compressed sensing and matrix completion with constant proportion of corruptions. *Constructive Approximation*, **37**(1), 73–99.

Milnor, John W, & Stasheff, James D. 1974. *Characteristic classes, volume 76 of Annals of Mathematics Studies*.

Netflix. 2009. *Netflix Prize Leaderboard*. Available at http://www.netflixprize.com/leaderboard.

Nie, Feiping, Huang, Heng, & Ding, Chris. 2012a. Low-Rank Matrix Recovery via Efficient Schatten p-Norm Minimization. *Pages 655–661 of: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. AAAI.

Nie, Feiping, Wang, Hua, Cai, Xiao, Huang, Heng, & Ding, Chris. 2012b. Robust matrix completion via joint schatten p-norm and lp-norm minimization. *Pages 566–574 of: Data Mining (ICDM), 2012 IEEE 12th International Conference on*. IEEE.

Oh, Sewoong, Montanari, Andrea, & Karbasi, Amin. 2010. Sensor network localization from local connectivity: Performance analysis for the mds-map algorithm. *Pages 1–5 of: Information Theory Workshop (ITW), 2010 IEEE*. IEEE.

Peng, Yigang, Ganesh, Arvind, Wright, John, Xu, Wenli, & Ma, Yi. 2012. RASL: Robust

alignment by sparse and low-rank decomposition for linearly correlated images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **34**(11), 2233–2246.

Razaviyayn, Meisam, Hong, Mingyi, & Luo, Zhi-Quan. 2013. A unified convergence analysis of block successive minimization methods for nonsmooth optimization. *SIAM Journal on Optimization*, **23**(2), 1126–1153.

Sato, H. 2014. A Dai-Yuan-type Riemannian conjugate gradient method with the weak Wolfe conditions. *ArXiv e-prints*, May.

Sato, Hiroyuki, & Iwai, Toshihiro. 2013. A new, globally convergent Riemannian conjugate gradient method. *Optimization*, 1–21.

So, Anthony Man-Cho, & Ye, Yinyu. 2007. Theory of semidefinite programming for sensor network localization. *Mathematical Programming*, **109**(2-3), 367–384.

The OpenMP ARB. 2015. *The OpenMP® API specification for parallel programming.* Available at http://openmp.org.

Vandereycken, Bart. 2013. Low-rank matrix completion by Riemannian optimization. *SIAM Journal on Optimization*, **23**(2), 1214–1236.

Vandereycken, Bart, Absil, Pierre-Antoine, Vandewalle, Stefan, *et al.* . 2009. Embedded geometry of the set of symmetric positive semidefinite matrices of fixed rank. *Pages 389–392 of: Proceedings of the IEEE 15th Workshop on Statistical Signal Processing.*

Wen, Zaiwen, Yin, Wotao, & Zhang, Yin. 2012. Solving a low-rank factorization model for matrix completion by a nonlinear successive over-relaxation algorithm. *Mathematical Programming Computation*, **4**(4), 333–361.

Wright, SJ, & Nocedal, J. 1999. *Numerical optimization.* Vol. 2. Springer New York.

Yan, Ming, Yang, Yi, & Osher, Stanley. 2013. Exact low-rank matrix completion from sparsely corrupted entries via adaptive outlier pursuit. *Journal of Scientific Computing*, **56**(3), 433–449.

Yang, Yuning, Fend, Yunlong, & Suykens, Johan A.K. 2014. *A Nonconvex Approach to Robust Matrix Completion.* Available at ftp://ftp.esat.kuleuven.be/stadius/yyang/yfs2014rmc.pdf.

A | Preconditioned Dai-Yuan Conjugate Gradient

Since we have access to a good preconditioner (Boumal & Absil, 2015), we need to develop the preconditioned version of the new Dai-Yuan conjugate gradient algorithm.

The idea behind preconditioning is to do a change of variable $x = Sy$ (at least in the Euclidian case) where S is chosen to speedup convergence of the algorithm (Hager & Zhang, 2006). If we begin by writing the conjugate gradient algorithm in y, changing the variables back to x leads

$$\eta_k = -P_{x_k} \operatorname{grad} f(x_k) + \bar{\beta}_{k-1}\eta_{k-1}$$

$$x_{k+1} = x_k + \alpha_k \eta_k$$

where $P_{x_k} = S_{x_k}^\top S_{x_k}$. The computation of $\bar{\beta}_k$ is the same as β_k but with $\operatorname{grad} f(x_k)$ replaced by $S_{x_k}^\top \operatorname{grad} f(x_k)$ and η_k replaced by $S_{x_k}^{-1}\eta_k$ (Hager & Zhang, 2006).

Using this rule, we get

$$\begin{aligned}\bar{\beta}_{k+1} &= \frac{\langle S_{x_{k+1}}^\top \operatorname{grad} f(x_{k+1}), S_{x_{k+1}}^\top \operatorname{grad} f(x_{k+1})\rangle_{x_{k+1}}}{\langle S_{x_{k+1}}^\top \operatorname{grad} f(x_{k+1}), \mathcal{T}^{(k)}_{\alpha_k\eta_k}(S_{x_k}^{-1}\eta_k)\rangle_{x_{k+1}} - \langle S_{x_k}^\top \operatorname{grad} f(x_k), S_{x_k}^{-1}\eta_k\rangle_{x_k}} \\ &\approx \frac{\langle \operatorname{grad} f(x_{k+1}), P_{x_{k+1}}(\operatorname{grad} f(x_{k+1}))\rangle_{x_{k+1}}}{\langle \operatorname{grad} f(x_{k+1}), \mathcal{T}^{(k)}_{\alpha_k\eta_k}(\eta_k)\rangle_{x_{k+1}} - \langle \operatorname{grad} f(x_k), \eta_k\rangle_{x_k}}\end{aligned}$$

with $P_x : T_x\mathcal{M} \to T_x\mathcal{M}$ the preconditioner, i.e., a "good" approximation of $\operatorname{Hess} f(x)^{-1}$. The $\approx$ comes from the fact that in general,

$$\langle S_{x_{k+1}}^\top \operatorname{grad} f(x_{k+1}), \mathcal{T}^{(k)}_{\alpha_k\eta_k}(S_{x_k}^{-1}\eta_k)\rangle_{x_{k+1}} \neq \langle \operatorname{grad} f(x_{k+1}), \mathcal{T}^{(k)}_{\alpha_k\eta_k}(\eta_k)\rangle_{x_{k+1}}$$

since S_x depends on x. Yet, since we do not have access in general to the factorization of $P_x = S_x^\top S_x$, we will use this approximation.

B | Implementation Details for the Parallel RCGMC

C-mex files are, basically, compiled C code that can be easily used with MATLAB. OpenMP is an interface compatible with a lot of processors and different kinds of hardware that allow programmers to easily write parallel code in C and in other programming languages.

In this work, we built a function `function [f, grad, WWt, stats] = costgradparallel(U, C, M, Ct, Mt, lambda, nThreads);` that takes the following inputs

1. `U`: the matrix representing the current point on $\mathrm{Gr}(m, r)$, i.e., a $m \times r$ real orthogonal matrix;
2. `C`: the weights $\mathbf{C}$ given as a $m \times n$ real *sparse matrix* (i.e., *not* a vector);
3. `M`: the data $\mathbf{M}$ given as a $m \times n$ real sparse matrix;
4. `Ct`: the transpose of the weights $\mathbf{C}$, $\mathbf{C}^\top$, given as a $n \times m$ real sparse matrix;
5. `Mt`: the transpose of the data $\mathbf{M}$, $\mathbf{M}^\top$, given as a $n \times m$ real sparse matrix;
6. `lambda`: the regularization parameter, a real positive scalar;
7. `nThreads`: the number of threads to be used, a real positive integer scalar.

and returns

1. `f`: the cost $f(\mathbf{U})$;
2. `grad`: the gradient $\operatorname{grad} f(\mathbf{U})$;
3. `WWt`: the matrix $\mathbf{W}\mathbf{W}^\top$ (useful in the preconditioner);
4. `stats`: some statistics to analyze the algorithm execution. This structure is made of 16 fields, containing the execution time of each part of the algorithm: inputs checks, allocations and then each different portion of the algorithm.

To write such a function, there is a few things required:

- The name of the file should be `costgradparallel.c`;
- The file should contain the following function

```
1 void mexFunction(
2           int nlhs, mxArray *plhs[],
3         int nrhs, const mxArray* prhs[] )
4 {
```

that will be called when the function is called from the MATLAB environment. The `nlhs` variable count the number of outputs that will all be stored in the `plhs` array, while `nrhs` count the inputs, stored in `prhs`;

- The file (and other depend files) should be compiled from the MATLAB environment. In our case, the dependent files[1] are compiled using

```
mex file.c -c -lmwlapack -lmwblas -largeArrayDims CFLAGS="\$CFLAGS -fopenmp -O3 -
Wall" LDFLAGS="\$LDFLAGS -O3 -fopenmp -Wall"
```

where `file` is one of the following `accsABtFast.c sUW.c sRW.c UtCX.c` . The `costgradparallel.c` on the other hand is compiled with

```
mex costgradparallel.c accsABtFast.o sUW.o sRW.o UtCX.o buildSolveChol.o -
lmwlapack -lmwblas -largeArrayDims CFLAGS="\$CFLAGS -O3 -fopenmp -Wall" LDFLAGS
="\$LDFLAGS -O3 -fopenmp -Wall"
```

where we can see the depend files `accsABtFast.o sUW.o sRW.o UtCX.o`. In the command, we see the `-fopenmp` flag which is used to activate the OpenMP commands. Note that we successfully compiled the files on both a OS X and a Linux computer, but there might be additional change to do in your system configuration to compile the files.

To parallelize portions of the code, we thus used the OpenMP interface. The use of such tool is very easy. For instance, to parallelize this loop

```
for(colW = 0; colW < n; colW++) {
    buildSolveChol(colW, m, n, r, U, Csparse, Xsparse, sIr, sJc, lambdaval, R, U, W+r*
    colW) ;
}
```

used to find each column of $\mathbf{W}$ sequentially, we simply need to add the following command `#pragma omp parallel for` in front of it and allocate a few arrays for each thread (in order for each of them to write intermediatary results in their own array, to avoid "collisions"). This leads to the following code

```
#pragma omp parallel for
for(colW = 0; colW < n; colW++) {
    int tidLoc = omp_get_thread_num();
    buildSolveChol(colW, m, n, r, U, Csparse, Xsparse, sIr, sJc, lambdaval, Rk[tidLoc
    ], Ui[tidLoc], W+r*colW) ;
}
```

where the `tidLoc` variable is used to retreive the "thread number", in order for each thread to write temporary results in his own array. All arrays are then merged at the end. This is the main and only strategy used to parallelize operations in this implementation.

[1] Regular C files, created in order to avoid having all functions in one single file.

C | Orthogonalized Alternating Linear Matrix Completion

Adding the orthogonalization *heuristic*[1] to algorithm 3, we obtain algorithm 5, the *orthogonalized* Alternating Linear Matrix Completion, oALMC.

Algorithm 5 Orthogonalized Alternating Linear Matrix Completion

procedure oALMC($\mathbf{U}^{(0)}, \mathbf{V}^{(0)}, \epsilon$)
  $k \leftarrow 0$
  $e \leftarrow \infty$
  while $e > \epsilon$ **do**
    $\mathbf{U}^{(k+1)} \leftarrow \operatorname{argmin}_{\mathbf{U}\in\mathbb{R}^{m\times r}} \|\mathcal{P}_\Omega(\mathbf{M} - \mathbf{U}\cdot\mathbf{V}^{(k)})\|_1$
    $\mathbf{V}^{(k+1)} \leftarrow \operatorname{argmin}_{\mathbf{V}\in\mathbb{R}^{r\times n}} \|\mathcal{P}_\Omega(\mathbf{M} - \mathbf{U}^{(k+1)}\cdot\mathbf{V})\|_1$
    $\mathbf{QR} \leftarrow \mathbf{U}^{(k+1)}$
    $\mathbf{U}^{(k+1)} \leftarrow \mathbf{Q}$
    $\mathbf{V}^{(k)} \leftarrow \mathbf{R}\mathbf{V}^{(k)}$
    $e \leftarrow f(\mathbf{U}^{(k)}, \mathbf{V}^{(k)}) - f(\mathbf{U}^{(k+1)}, \mathbf{V}^{(k+1)})$
    $k \leftarrow k + 1$
  end while
  return $\mathbf{U}^{(k)}$, $\mathbf{V}^{(k)}$
end procedure

We can easily prove that, if it converges, this algorithm converges to a coordinate-wise minimizer of

$$\min_{\mathbf{U}\in\mathbb{R}^{m\times r},\mathbf{V}\in\mathbb{R}^{r\times n},\mathbf{U}^\top\mathbf{U}=\mathbf{I}_r} \sum_{(i,j)\in\Omega} |M_{ij} - (UV)_{ij}|,$$

the orthogonality constraint on $\mathbf{U}$ making it probable. The proof is mostly the same as for ALMC, but we need to keep track of the orthogonality constraint.

Theorem 3 (Convergence of oALMC)**.** *If iterates* $(\mathbf{U}^{(k)}, \mathbf{V}^{(k)})$ *are generated by algorithm 5, then every limit point is a coordinate-wise minimizer of the function*

$$f : \mathbb{R}^{m\times r} \times \mathbb{R}^{r\times n} : (\mathbf{U}, \mathbf{V}) \to \|\mathcal{P}_\Omega(\mathbf{M} - \mathbf{U}\mathbf{V})\|_{\ell_1}$$

Proof. Let us define $\mathbf{X}^{(r)} = (\mathbf{U}^{(\lceil r/2\rceil)}, \mathbf{V}^{(\lfloor r/2\rfloor)})$. Hence, r increases by one when we either update $\mathbf{U}$ or $\mathbf{V}$.

[1] As it does not prove that the $(\mathbf{U}^{(k)}, \mathbf{V}^{(k)})$ remain bounded since it depends on multiple parameters like the mask Ω and the underlying low-rank matrix.

We trivially have, $\forall r \geq 0$:

$$f(\mathbf{X}^{(r)}) \geq f(\mathbf{X}^{(r+1)}) \geq \cdots \geq 0$$

Let us extract from $\{\mathbf{X}^{(r)}\}_{r\in\mathbb{N}}$ a subsequence converging towards $\mathbf{X}^* = (\mathbf{U}^*, \mathbf{V}^*)$: $\{\mathbf{X}^{(r_j)}\}_{j\in\mathbb{N}}$. Let us also denote by $\tilde{\mathbf{U}}^{(k)}$ and $\tilde{\mathbf{V}}^{(k)}$ the intermediate matrices (i.e., after optimization but before orthogonalization). Using this notation, note that

$$f(\tilde{\mathbf{U}}^{(k)}, \tilde{\mathbf{V}}^{(k)}) = f(\mathbf{U}^{(k)}, \mathbf{V}^{(k)}).$$

Let us focus on the $\mathbf{U}$ iterates. For all $\mathbf{U} \in \mathbb{R}^{m\times r}$ such that $\mathbf{U}^\top\mathbf{U} = \mathbf{I}_r$, and for all $j \in \mathbb{N}$ we have

$$\begin{aligned}
f(\mathbf{U}, \mathbf{V}^{(\lfloor r_j/2 \rfloor)}) &\geq \min_{\mathbf{U}\in\mathbb{R}^{m\times r}:\mathbf{U}^\top\mathbf{U}=\mathbf{I}_r} f(\mathbf{U}, \mathbf{V}^{(\lfloor r_j/2 \rfloor)}) \\
&\geq \min_{\mathbf{U}\in\mathbb{R}^{m\times r}} f(\mathbf{U}, \mathbf{V}^{(\lfloor r_j/2 \rfloor)}) \\
&= f(\tilde{\mathbf{U}}^{(\lfloor r_j/2 \rfloor+1)}, \mathbf{V}^{(\lfloor r_j/2 \rfloor)}) \\
&\geq f(\tilde{\mathbf{U}}^{(\lfloor r_j/2 \rfloor+1)}, \tilde{\mathbf{V}}^{(\lfloor r_j/2 \rfloor+1)}) \\
&\geq f(\mathbf{U}^{(\lfloor r_j/2 \rfloor+1)}, \mathbf{V}^{(\lfloor r_j/2 \rfloor+1)}) \\
&\geq f(\mathbf{X}^{2(\lfloor r_j/2 \rfloor+1)}) \\
&\geq f(\mathbf{X}^{(r_j+2)}) \\
&\geq f(\mathbf{X}^{(r_{j+2})}) \\
&= f(\mathbf{U}^{(\lceil r_{j+2}/2 \rceil)}, \mathbf{V}^{(\lfloor r_{j+2}/2 \rfloor)}).
\end{aligned}$$

By letting $j \to \infty$, we have $\forall \mathbf{U} \in \mathbb{R}^{m\times r}$ such that $\mathbf{U}^\top\mathbf{U} = \mathbf{I}_r$,

$$f(\mathbf{U}, \mathbf{V}^*) \geq f(\mathbf{U}^*, \mathbf{V}^*).$$

This is sufficient to conclude that $\mathbf{U}^*$ is a coordinate-wise minimizer of f.

The exact same proof can be used to note that $\mathbf{V}^*$ is a coordinate-wise minimizer of f.

□