Numerical algorithms for low-rank matrix completion problems

Marie Michenková
Seminar for Applied Mathematics,
Department of Mathematics,
Swiss Federal Institute of Technology Zurich,
Switzerland

May 30, 2011

We consider a problem of recovering low-rank data matrix from sampling of its entries. Suppose that we observe m entries selected uniformly at random from an $n_1 \times n_2$ matrix M. One can hope that when enough entries are revealed ($\mathcal{O}(nr\log n)$), it is possible to recover the matrix exactly. We downloaded eight solvers implemented in Matlab for low-rank matrix completion and tested them on different problems. The report includes brief description of the solvers used (based on the original papers) and results of the experiment carried out.

1 Definition of low - rank matrix completion problem

We will use the following definition of a matrix completion problem. Suppose that $M \in \mathbb{R}^{n_1 \times n_2}$ and let Ω be a subset of $[n_1] \times [n_2]$ of revealed entries of M. Then we want to find a solution of the following problem

$$\begin{array}{ll} \text{minimize} & \text{rank}(X) \\ \text{subject to} & X_{ij} = M_{ij} \quad (i,j) \in \Omega. \end{array} \tag{1}$$

In words, we are looking for a matrix of the lowest rank whose entries in set Ω correspond to the entries of M. Let us define a projector $\mathcal{P}_\Omega(\cdot) : \mathbb{R}^{n_1 \times n_2} \to \mathbb{R}^{n_1 \times n_2}$ as follows

$$(\mathcal{P}_\Omega(A))_{ij} = \begin{cases} A_{ij} & \text{if } (i,j) \in \Omega. \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Using projector (2), problem (1) can be rewritten as

$$(P_0): \quad \begin{array}{ll} \text{minimize} & \text{rank}(X) \\ \text{subject to} & \mathcal{P}_\Omega(X) = \mathcal{P}_\Omega(M). \end{array}$$

However, (P_0) is an NP-hard problem and has to be relaxed for real computations.

One can approximate (P_0) by the 'tightest' convex problem

$$(P_1): \quad \begin{array}{ll} \text{minimize} & ||X||_* \\ \text{subject to} & \mathcal{P}_\Omega(X) = \mathcal{P}_\Omega(M), \end{array}$$

where $\|\cdot\|_*$ denotes the nuclear norm (the sum of the singular values). The 'tightest' in this sense means that the unit nuclear ball $\{X : \|X\|_* \leq 1\}$ is the convex hull of the set of rank-one matrices with spectral norm bounded by one, i.e. $\{X : \text{rank}(X), \|X\| \leq 1\}$. Under slightly stronger assumptions than for the uniqueness of the matrix recovery, for random matrices, the problem (P_1) has the same solution as (P_0) with probability close to 1. See [4] for more details.

One can also approximate (P_0) by

$$(P_2): \quad \begin{array}{ll} \text{minimize} & \|\mathcal{P}_\Omega(X) - \mathcal{P}_\Omega(M)\|_F \\ \text{subject to} & \text{rank}(X) \leq r. \end{array}$$

This approximation is based on an a priori knowledge of the rank and already assumes that the revealed entries are corrupted by Gaussian noise.

The report is organized as follows. Algorithms based on the relaxation (P_1) are listed in section 2, algorithms based on (P_2) in section 3 and 4. Section 5 includes results of experiments and some remarks on the implementation.

2 Algorithms based on nuclear norm minimization

In this section, we will consider solvers for the constrained convex problem (P_1). We included a convex programming package and solvers based on Lagrangian, penalty, and augmented Lagrangian method.

2.1 Convex programming package *cvx* (cvx)

Problem (P_1) can be solved 'directly' using some convex programming package. We used *cvx* [7]. Using *cvx*, (P_1) can be easily implemented as

```
ind = find(M);
cvx_begin
    variable X(size(M));
    minimize( norm_nuc(X) );
    subject to
        M(ind) == X(ind);
cvx_end
```

where `M(ind) == X(ind)` means equality up to some tolerance, which can be set by `cvx_precision('level')`. The package is very reliable. The following table shows the computation time on author's desktop computer for different sizes of matrices (square) with sampling ratio 0.8, i.e. 80% of the entries were revealed.

Problem			cvx
n	r	SR	time
20	2	0.80	0.8 s
50	5	0.80	36.9 s
80	8	0.80	542.3 s

We see that if we want to solve the convex problem (P_1) using *cvx*, the computation is costly and we are only able to solve problems up to size of approximately 100×100. Since this is not sufficient for 'real problems', problem (P_1) is usually relaxed further in some sense.

2.2 Soft-thresholding operator

For the algorithms based on nuclear-norm minimization, it will be useful to define so-called soft-thresholding operator. Let $A \in \mathbb{R}^{n_1 \times n_2}$ be a matrix of rank r with the following SVD

$$A = U\Sigma V^T, U \in \mathbb{R}^{n_1 \times r}, \Sigma = \mathrm{diag}(\{\sigma_i\}_{i=1}^r) \in \mathbb{R}^{r \times r}, V \in \mathbb{R}^{n_2 \times r}.$$

For each $\tau > 0$ we define the *soft-thresholding operator* $\mathcal{D}_\tau(\cdot)$:

$$\mathcal{D}_\tau(A) \equiv U\mathcal{D}_\tau(\Sigma)V^T, \text{where } \mathcal{D}_\tau(\Sigma) = \mathrm{diag}(\{(\sigma_i - \tau)_+\}_{i=1}^r).$$

This operator is also often called *singular value shrinkage operator*. If many of the singular values of A are below the threshold τ, the rank of $\mathcal{D}_\tau(A)$ is considerably lower than that of A. Authors ins [3] proved that for all $\tau > 0$

$$\mathcal{D}_\tau(A) = \arg\min_B \left\{ \tau ||B||_* + \frac{1}{2}\|B - A\|_F^2 \right\}.$$

2.3 A Singular Value Thresholding Algorithm (SVT)

SVT algorithm introduced in [3] is based on the following iteration. Fix $\tau > 0$ and positive step sizes $\{\delta_k\}$. Starting with $Y^0 = 0$, repeat

$$\begin{cases} X^k = \mathcal{D}_\tau(Y^{k-1}) \\ Y^k = Y^{k-1} + \delta_k \mathcal{P}_\Omega(M - X^k) \end{cases} \tag{3}$$

until a stopping criterion is reached. Important is that $\{X^k\}$ have low rank (according to results presented in [3] empirically nondecreasing with k) and the auxiliary matrices $\{Y^k\}$ are sparse. Only few (depending on τ) singular values and corresponding singular values are needed in each iteration. These are computed iteratively using PROPACK [9]. However, PROPACK can compute only a given number of largest singular values. To use this package, we have to determine the number s_k of singular values of Y^{k-1} to be computed at the k-th iteration. Authors suggest the following procedure. Let $r_{k-1} = \mathrm{rank}(X^{k-1})$ Set $s_k = r_{k-1} + 1$ and compute the first s_k singular values of Y^{k-1}. If some of the computed singular values are already smaller than τ, then s_k is a right choice. Otherwise, increment s_k by a predefined integer l repeatedly until some of the singular values falls below τ. Authors choose $l = 5$.

It can be shown that for $0 < \inf \delta_k \leq \sup \delta_k < 2$, the sequence $\{X^k\}$ in (3) converges to the solution of the following problem

$$\begin{array}{ll} \text{minimize} & f_\tau(X) \equiv \tau||X||_* + \frac{1}{2}\|X\|_F^2 \\ \text{subject to} & \mathcal{P}_\Omega(M - X) = 0. \end{array} \tag{4}$$

For τ large enough, (4) is intuitively close to the problem (P_1). It can be shown, that the solution to (4) converges to the solution to (P_1) with minimal Frobenius norm as $\tau \to \infty$. But on the other side $\tau \to \infty$ slows down the convergence considerably.

The theory above can be extended to the problems with more general constraints, which could be useful for recovering matrices, where some of the entries are contaminated by noise. Algorithm (3) can be also viewed as a Lagrange multiplier algorithm (in this case known as Uzawa's algorithm) applied to the problem (4).

The following parameters can be adjusted: Shrinkage level τ, choose $\tau = 5n$, which should provide that on average, the value of $\tau||X||_*$ is about 10 times larger than the value of $\frac{1}{2}\|X\|_F^2$. Step sizes $\{\delta_k\}$, choose $\delta_k \equiv \delta = 1.2\frac{n_1 n_2}{m}$ instead of too conservative $0 < \delta < 2$. Stopping criterion is standard,

$$\frac{\|\mathcal{P}_\Omega(X^k - M)\|_F}{\|\mathcal{P}_\Omega(M)\|_F} \leq \text{Tol.} \tag{5}$$

Noise level Eps relaxes the constraints $\mathcal{P}_\Omega(M - X) = 0$, so that they are of the form $|X_{ij} - M_{ij}| \leq \text{Eps } \forall (i,j) \in \Omega$.

2.4 An Accelerated Proximal Gradient Algorithm (APGL)

APGL introduced in [14] solves the following nuclear norm regularized linear least squares problem

$$\min_{X \in \mathbb{R}^{n_1 \times n_2}} F_\lambda(X) \equiv \frac{1}{2}\|\mathcal{A}(X) - b\|_2^2 + \lambda||X||_*, \tag{6}$$

where λ is a given regularization parameter. For $\mathcal{A} \sim \mathcal{P}_\Omega$ and $b \sim \mathcal{P}_\Omega(M)$, problem (6) corresponds to the minimization problem

$$\min_{X \in \mathbb{R}^{n_1 \times n_2}} F_\lambda(X) \equiv \lambda||X||_* + \frac{1}{2}\|\mathcal{P}_\Omega(M - X)\|_F^2, \tag{7}$$

which is just a penalty method applied to our original constrained convex problem (P_1). $F_\lambda(Z)$ can be locally approximated as a quadratic function

$$\begin{aligned} Q_\tau(X, Z) &\equiv \frac{1}{2}\|\mathcal{P}_\Omega(Z - M)\|_F^2 + \langle \mathcal{P}_\Omega(Z - M), X - Z\rangle + \frac{\tau}{2}\|X - Z\|_F^2 + \lambda||X||_* \\ &= \frac{\tau}{2}\|X - Y\|_F^2 + \lambda||X||_* + \frac{1}{2}\|\mathcal{P}_\Omega(Z - M)\|_F^2 - \frac{1}{2\tau}\|\mathcal{P}_\Omega(Z - M)\|_F^2, \end{aligned} \tag{8}$$

where τ is the Lipschitz constant for the gradient of $\mathcal{P}_\Omega$ (i.e. $\tau = 1$), and $Y = Z + \frac{1}{\tau}\mathcal{P}_\Omega(M - Z)$. Algorithm suggested in [14] is then based on the minimization of function (8) over X:

$$X^k = \arg \min_{X \in \mathbb{R}^{n_1 \times n_2}} Q_\tau(X, Z^{k-1}) = \arg \min_{X \in \mathbb{R}^{n_1 \times n_2}} \frac{\tau}{2}\|X - Y^{k-1}\|_F^2 + \lambda||X||_*, \tag{9}$$

where $Y^{k-1} = Z^{k-1} + \frac{1}{\tau_k}\mathcal{P}_\Omega(M - Z^{k-1})$. There is a natural choice of $Z^k = X^k$ in (9) but the authors suggest taking $Z^k = X^k + \frac{t^{k-1}-1}{t^k}(X^k - X^{k-1})$ instead. The algorithm then looks as follows. Starting with $X^0 = X^{-1} \in \mathbb{R}^{n_1 \times n_2}$ and $t^0 = t^{-1} = 1$, repeat

$$\begin{cases} Z^k & = X^k + \frac{t^{k-1}-1}{t^k}(X^k - X^{k-1}) \\ Y^k & = Z^k + \frac{1}{\tau_k}\mathcal{P}_\Omega(M - Z^k), \text{where } \tau_k = 1 \\ X^{k+1} & = \mathcal{D}_{\frac{\lambda}{\tau_k}}(Y^k) \\ t^{k+1} & = \frac{1+\sqrt{1+4(t^k)^2}}{2} \end{cases} \tag{10}$$

until a stopping criterion is reached.
Authors suggest three techniques to accelerate the algorithm (10):

- It is too conservative to set $\tau_k \equiv 1$. To accelerate the convergence, it is desirable to take a smaller value. We can use linesearch-like technique to find a smaller τ_k that still satisfies inequality (14) in [14]. Second advantage of this approach is that the smaller is τ_k, the lower is rank(X^{k+1}).

- λ is usually chosen to a moderately small number, which means that we have to compute almost entire SVD in each step. Authors suggest computing a sequence of approximate solutions for decreasing sequence of $\{\lambda_0, \lambda_1, \ldots, \lambda_l = \lambda\}$, where we take the solution computed with λ_{i-1} as a starting value for the algorithm with λ_i.

- For APG without any acceleration technique, the iterates X^k are not low-rank, but the singular values will usually separate into two clusters with distant means. To achieve low-rank iterative in each step of APG, the authors suggest discarding singular values from the second cluster.

The only parameter that has to be specified is the constrain level λ. The authors suggest choosing $\lambda = 10^{-4}\lambda_0$, where $\lambda_0 = \|\mathcal{P}_\Omega(M)\|_F$. After each step of the algorithm (10) we evaluate two inequalities

$$\begin{aligned} \frac{\|X^k - X^{k-1}\|_F}{\max\{\|X^k\|_F, 1\}} &\leq \text{Tol} \\ \frac{|\|\mathcal{P}_\Omega(M - X^k)\|_F - \|\mathcal{P}_\Omega(M - X^{k-1})\|_F|}{\max\{\|M\|_F, 1\}} &\leq 5 \times \text{Tol}. \end{aligned}$$

If at least one of them is satisfied, we stop the computation. One can also adjust parameters used for $\mathcal{D}_{\frac{\lambda}{\tau_k}}(\cdot)$.

2.5 The Augmented Lagrange Multiplier Method (ALM)

To solve problem (P_1), ALM introduced in [11] uses method of augmented Lagrange multipliers for problems of the form of

$$\begin{array}{ll} \text{minimize} & f(X) \equiv ||X||_* \\ \text{subject to} & \mathcal{P}_\Omega(M) - X - E = 0. \end{array} \tag{11}$$

where $\mathcal{P}_\Omega(E) = 0$ is enforced directly in each step of the algorithm. The partial augmented Lagrangian of problem (11) then takes form of

$$\mathcal{L}(X, E, Y, \mu) = ||X||_* + \langle Y, \mathcal{P}_\Omega(M) - X - E\rangle + \frac{\mu}{2}\|\mathcal{P}_\Omega(M) - X - E\|_F^2 \qquad (12)$$

We can minimize directly (12) similarly as for SVT, i.e. minimize over (X, E) simultaneously. This will lead to so-called Exact ALM Method. But the authors suggest minimizing just once over X and then over E in each step. This method is called Inexact ALM Method:

$$\begin{cases} X^k &= \mathcal{D}_{\mu_{k-1}^{-1}}(\mathcal{P}_\Omega(M) - E^{k-1} + \mu_{k-1}^{-1}Y^{k-1}) \\ E^k &= \mathcal{P}_{\Omega^C}(X^k) \\ Y^k &= Y^{k-1} + \mu_{k-1}\mathcal{P}_\Omega(M - X^k) \\ \mu_k &= \rho\mu_{k-1}. \end{cases} \qquad (13)$$

Note that $\mathcal{P}_{\Omega^C}(Y^{k-1}) = 0$ throughout the iteration. The order of variables in the minimization procedure is not essential, but according to the authors provides numerically a small decrease of iterations. Implementation of this algorithm uses PROPACK, which means that the number of required singular values has to be provided in the first step of each iteration. This is a challenging task for algorithm (13), because the ranks of X^k may oscillate. Authors adopted the truncation strategy from APGL. The ranks of X^k then should become monotonically increasing, stable at the true rank.

Solver uses standard stopping criterion (5). Parameters (μ_0 and ρ) are set in the main function directly.

2.6 Other solvers

We add brief review of some other algorithms minimizing the nuclear norm.

Fixed point continuation with approximate SVD (FPCA). This method minimizes the same function (7) as APGL but in the approximate version, SVD is computed using a fast Monte Carlo algorithm. See [12] for more details.

The alternating splitting augmented Lagrangian method (ASALM). This method solves the following nuclear norm and l_1-norm minimization problem:

$$\min_{X,F} ||X||_* + \|F\|_1, \ \text{ subject to } \mathcal{P}_\Omega(M) - X - E - F = 0, \ \|\mathcal{P}_\Omega(E)\|_F \le \delta, \qquad (14)$$

where F corresponds to impulsive noise (sparse but large) and E corresponds to Gaussian noise. Problem (14) is solved by minimizing the following unconstrained problem

$$\mathcal{L}(X, F, E, Y, \mu) = ||X||_* + \langle Y, \mathcal{P}_\Omega(M) - X - F - E\rangle + \frac{\mu}{2}\|\mathcal{P}_\Omega(M) - X - F - E\|_F^2 \quad (15)$$

similarly as in Inexact ALM. Authors suggest minimizing first over E than over F and at last over X. See [13] for more details. Convergence of this algorithm is not proven. The authors suggested a variant of ASALM (VASALM), which minimizes over E and updates Y using the new E and then minimizes parallel extended functions over F and X (new parameter η is introduced, for $\eta = 1$ these functions correspond to the ones in ASALM) and than updates Y once again. Convergence of this algorithm was proven, but numerically is slower than the convergence of ASALM. Authors in [5] suggest using ASALM as a prediction step and a Gaussian back substitution procedure as a correction step. This should ensure convergence while preserving the advantages of ASALM.

3 Algorithms Based on the Minimization on Grassmann Manifold

In this section we will consider solvers solving (P_2) by minimizing a function F on a Grassmann manifold. We included three solvers minimizing different functions on $\mathrm{Gr}(r, n)$.

3.1 OPTSPACE

Algorithm known as OptSpace introduced in [8] is based on minimization of the following cost function

$$F(U, V) \equiv \min_{S \in \mathbb{R}^{r \times r}} \frac{1}{2} \|\mathcal{P}_\Omega(M - USV^T)\|_F^2, \tag{16}$$

where $U \in \mathbb{R}^{n_1 \times r}$ and $V \in \mathbb{R}^{n_2 \times r}$ are orthogonal matrices, normalized as $U^T U = n_2 I$ and $V^T V = n_1 I$, and $\lambda \in [0, 1]$. Obviously, $\mathrm{rank}(USV^T) \leq r$. Minimizing $F(U, V)$ is a difficult task, since it is a non-convex function, but according to the authors the SVD of $\mathcal{P}_\Omega(M)$ gives a good initial guess. Algorithm OptSpace consists of the following four main steps:

1. trimming (output $\tilde{M}$)
2. estimating the rank of M from the SVD of $\tilde{M}$ (output $\hat{r}$)
3. rank-$\hat{r}$ projection of $\tilde{M}$ (output U_0, S_0, V_0)
4. minimizing $F(\cdot, \cdot)$ through gradient descent algorithm

Trimming is a procedure that sets 0 to all overrepresented rows and columns (in the code only some entries are set to zero). Here overrepresented means that number of entries in the row/column is more than twice larger than the average value. Otherwise singular vectors can concentrate along the overrepresented rows and columns and will not provide any useful information about the unrevealed entries of M. Let us denote the trimmed matrix $\tilde{M}$.

Rank estimation is based on the SVD of $\tilde{M}$. Authors use two estimators and choose the maximum of both results.

Rank-$\hat{r}$ projection consists of rescaling of the singular values and singular vectors of $\tilde{M}$. Let $\tilde{M}$ have the following SVD $\tilde{M} = \sum_{i=1}^{\min(n_1, n_2)} u_i \sigma_i v_i^T$, $U_0 = \sqrt{n_1}[u_1, \ldots, u_{\hat{r}}]$, $V_0 = \sqrt{n_2}[v_1, \ldots, v_{\hat{r}}]$ and $S_0 = \frac{\sqrt{(n_1 n_2)}}{m} \mathrm{diag}(\sigma_1, \ldots, \sigma_{\hat{r}})$.

Then we start gradient descent algorithm on the Grassmann manifold for the function $F(U, V)$ with initial guess (U_0, V_0), where $X^k = U_k S_k V_k^T$. After each step we find optimal S_k for computed (U_k, V_k) solving a least squares problem. Standard stopping criterion (5) is used.

3.2 Space Evolution and Transfer (SET)

Assume that we are able to guess the rank r of our unknown matrix M. The aim of the algorithm called SET introduced in [6] is to find a matrix which satisfies (P_2). This algorithm is similar to algorithm OptSpace. However, in SET the rank has to be

defined a priori. Another difference is that OptSpace minimizes over columns and rows (U and V) simultaneously, while SET minimizes only over columns. SET is based on the minimization of the following function

$$F(U) \equiv \min_{W \in \mathbb{R}^{r \times n_2}} \|\mathcal{P}_\Omega(M - UW)\|_F^2, \tag{17}$$

where $U \in \mathbb{R}^{n_1 \times r}$ is an orthonormal matrix. Concerning that $\text{rank}(M) = r$ and no noise is added, the minimum value of $F(\cdot)$ is zero. Note that $F(U)$ can be defined column-wise as

$$F(U) = \sum_{j=1}^{n_2} \underbrace{\min_{w_j \in \mathbb{R}^r} \|\mathcal{P}_{\Omega,j}(M_j) - \mathcal{P}_{\Omega,j}(Uw_j)\|_F}_{F_j(U)}, \tag{18}$$

where $\mathcal{P}_{\Omega,j}$ is restriction of $\mathcal{P}_\Omega$ to the j-th column. The algorithm consists of the following three main steps. Starting with a random orthonormal matrix U repeat

1. subspace transfer
2. subspace evolution
3. find optimal W,

until a stopping criterion is reached.

Subspace evolution is based on the gradient descent algorithm on Grassmann manifold (or similar technique, where the direction H is rank-one matrix created from the first singular triplet of the gradient). Subspace evolution part is designed to search for the minimizer of function $F(\cdot)$ along the geodesic curve defined by starting point U_0 (output of previous iteration) and 'direction' $-H$. Subspace transfer was introduced because function $F(\cdot)$ is generally not convex and the subspace evolution may not converge to the global minimum of the function, when one encounters 'barriers'. These can appear, when the individual atomic functions F_j imply different directions. This can block the search procedure from reaching the global optimum. The authors devised heuristic procedure for detecting such barriers and transferring the current estimate U from one side of barrier to another. The technique is described in [6] section III.E and III.F. W is then the corresponding least squares solution. Standard stopping criterion (5) is used.

3.3 Grassmannian Rank-One Update Subspace Estimation (GROUSE)

Algorithm GROUSE introduced in [2] is based on the minimization of the cost function (17), but this cost function is optimized one column at a time. The authors use function $\bar{F}(\cdot,\cdot)$ defined as follows

$$\bar{F}(U, i) = \min_a \|\mathcal{P}_{\Omega,i}(M_i - Ua)\|^2. \tag{19}$$

Gradient of (19) can be expressed as

$$\nabla \bar{F} = -2rw^T,$$

where w is the corresponding least-squares solution, i.e. $w = \arg\min_a \|\mathcal{P}_{\Omega,i}(M_i - Ua)\|$, and r is residual vector, i.e. $r = \mathcal{P}_{\Omega,i}(M_i - Uw)$. The entire algorithm looks as follows. Given a set of step sizes $\{\eta_i\}$ and an orthogonal matrix U_0, for $i = 1, \ldots, n_2$, repeat

$$\begin{cases} w &= \arg\min_a \|\mathcal{P}_{\Omega,i}(M_i - Ua)\|^2 \\ p &= U_i w \\ r &= \mathcal{P}_{\Omega,i}(M_i - p) \\ U_{i+1} &= U_i + \left(\sin(\|r\|\|p\|\eta_i)\frac{r}{\|r\|} + (\cos(\|r\|\|p\|\eta_i) - 1)\frac{p}{\|p\|}\right)\frac{w^T}{\|w\|}, \end{cases} \tag{20}$$

where $p \equiv Uw$ and the last step of the algorithm corresponds to the step of length η in the direction $-\nabla\bar{F}$. The update rule consist only of a rank-one modification of the current subspace basis U. The authors do not suggest any particular choice of step size. Number of outer cycles (how many times is (20) repeated) can be set. One can also determine the initial matrix U_0.

3.4 Other solvers

Atomic decomposition for minimum rank approximation (ADMiRA) This method is based on atomic decomposition of the matrix M, i.e.

$$M = \sum_j \alpha_j \psi_j, \tag{21}$$

where ψ_j are rank-one matrices. The aim is then to maximize the norm of the projection $\mathcal{P}_\Psi(\cdot)$ over all subspaces spanned by a subset with at most r atoms ψ_j:

$$\hat{\Psi} \equiv \arg\max_\Psi \left\{ \|\mathcal{P}_\Psi(\mathcal{P}_\Omega(M - \hat{X}))\|_F; |\Psi| \leq r \right\}, \tag{22}$$

where $\hat{X}$ is our current approximation of X. The algorithm adjusts $\hat{X}$ and $\hat{\Psi}$ alternatively. See [10] for more details.

4 A Low-rank Matrix Fitting Algorithm (LMaFit)

In this section we will mention a special type of method for solving (P_2), which is not based on minimization on Grassmannian manifold. It is based on the minimization of the following type

$$\begin{array}{l} \min_{U,V,Z} \frac{1}{2}\|Z - UV\|_F^2 \\ \text{subject to } \mathcal{P}_\Omega(Z) = \mathcal{P}_\Omega(M), \end{array} \tag{23}$$

where $U \in \mathbb{R}^{n_1 \times r}$, $V \in \mathbb{R}^{r \times n_2}$ and $Z \in \mathbb{R}^{n_1 \times n_2}$. Rank approximation r is adjusted dynamically in every step. Z is introduced for computational purpose. Since this problem is not convex, we decided to include the algorithm in the same section as Grassmannian based methods.

For solving (23) authors in [15] suggest minimizing separately over X, Y and Z. Where for X and Y they use successive over-relaxation method applied to normal equations. Minimization over Z is trivial. This method may not converge to the global minimum but authors proved convergence to a stationary point.

The nonlinear SOR scheme reads (using $Z_\omega \equiv \omega Z + (1-\omega)UV$)

$$\begin{cases} U^{k+1} &= Z_\omega^k (V^k)^T \\ V^{k+1} &= ((U^{k+1})^T U^{k+1})^\dagger (U^{k+1})^T Z_\omega^k \\ Z^{k+1} &= \mathcal{P}_\Omega(M) + \mathcal{P}_{\Omega^C}(U^{k+1}V^{k+1}). \end{cases} \tag{24}$$

Algorithm (24) is implemented as follows

$$\begin{cases} U^{k+1} &= \mathrm{orth}(Z_\omega^k (V^k)^T) \\ V^{k+1} &= (U^{k+1})^T Z_\omega^k \\ Z^{k+1} &= \mathcal{P}_\Omega(M) + \mathcal{P}_{\Omega^C}(U^{k+1}(V^{k+1})), \end{cases} \tag{25}$$

where the first step is computed via QR-decomposition, which is more stable than using normal equations. Parameter ω is adjusted in every step using the following procedure. If the new residual is greater or equal than then previous one, compute $(U^{k+1}, V^{k+1}, Z^{k+1})$ once again from (U^k, V^k, Z^k) using $\omega = 1$. If not, accept the the new triplet. If the ratio between current and previous residual is small enough (i.e. smaller than some $\bar{\gamma} < 1$) than keep the ω, otherwise increase the ω. See chapter 2 of [15] for more details.

There are implemented two heuristics for rank adjustment: decreasing (recommended for well-conditioned problems) and increasing (recommended for problems where clear-cut rank is not available). Decreasing heuristics is based on comparison of diagonal entries of R-factor in QR decomposition of X. Increasing strategy increases rank by some integer κ whenever the procedure stagnates. As a default the decreasing is used. Starting rank has to be determined.

The standard stopping criterion (5) is used. One can use another linear solver (own or implemented) for normal equation instead of using QR decomposition (which is default). Initial rank has to be provided.

5 Numerical experiments

All the experiment were conducted and timed on the same workstation with an AMD Phenom II (2.8 GHz) processor that has 4 cores and 7.6 GB memory under Linux with Matlab 7.10.0 (R2010a).

5.1 Packages/Solvers

Codes for SVT, APGL, Inexact ALM (IALM), OptSpace, SET and GROUSE were downloaded from `http://perception.csl.uiuc.edu/matrix-rank/`. Code for LMaFit was downloaded from `http://lmafit.blogs.rice.edu/`. Code for Exact ALM (EALM) was implemented by the author using code of IALM as a template.

5.2 General setting

If not specified, we generated matrix M of rank r by sampling two matrices of sizes $n_1 \times r$ and $n_2 \times r$ with i.i.d. Gaussian entries and setting $M = M_L M_R^T$. We sampled m entries at random. 'SR' corresponds to the sampling ratio, i.e. $\frac{m}{n_1 n_2}$. Noise was drawn from normal distribution and level of noise corresponds to $\frac{\|\mathcal{P}_\Omega(\text{noise})\|_F}{\|\mathcal{P}_\Omega(M)\|_F}$. We averaged over 5

samples. The effectivity of the solver was measured by $\frac{\|M-X\|_F}{\|M\|_F}$. The value of stopping criterion was set to 10^{-4} and matrix was considered recovered if $\frac{\|M-X\|_F}{\|M\|_F} \leq 10^{-3}$.

5.3 Experiments

Experiment 1. In this experiment we want to demonstrate, how reliable the algorithms are for small matrices. We set maximum number of iterations to 50 and the size of the matrix to 25×25. For each pair (m, r) (number of revealed entries and rank) we repeated the following procedure 5 times. We constructed matrix of rank r with m entries revealed and solved each problem by cvx, IALM, SVT, OptSpace, LMaFit and APGL. We used rank estimator form OptSpace to determine the rank for OptSpace a LMaFit. We obtained figure 1. We use abbreviation 'DoF' for the number of degrees of freedom for each r. Shade corresponds to the percentage of successful recoveries. We can see that even for such a small number of iterations, IALM performs very well (very close to cvx). SVT would definitely need much larger number of iterations to perform reasonably. OptSpace has problems with almost full matrices - the rank is not determined correctly (the guess is high) and when we solve LS problem, corresponding matrix is extremely ill-conditioned. LMaFit faces similar problems as OptSpace because it uses the same rank estimator. At the moment we do not have explanation for the behavior of APGL, which performs comparably to cvx for higher ranks, but very poorly for low-rank matrices. The stopping criterion starts to stagnate after a few iterations. We decided to test OptSpace and LMaFit also for the case when the correct rank is given. Obviously, we get better results, see table 2. For OptSpace we observe better performance for matrices with lower sampling ratio and we get rid of the leak on the right-hand side, but the shape of the light part stays almost the same. For LMaFit we got a very considerable improvement, mainly for higher-rank matrices. To compare IALM, OptSpace and LMaFit, we decided to plot the difference between IALM, OptSpace and LMaFit, both for the estimated and the given rank. We observe, that generally it is more secure to use IALM, when the rank is not known. When the rank is known, it is in most cases advisable to use LMaFit instead of OptSpace or IALM. Results are shown in table 3.

Experiment 2. In this experiment we compared the solvers by recovering matrices of size 50 and 200. We set maximum number of iterations to 100 and maximum number of inner iterations for EALM to 50. We used different ranks, percentage of entries known and level of the noise. For the solvers based on the minimization on Grassmann manifold and for LMaFit we used the rank of the matrix as a direct input. We observed that the noise level generally does not have a big influence on the precision of the results. However, noise level equal to the required precision can slow down the computation, e.g. by OptSpace. IALM, EALM, OptSpace, and LMaFit preformed the best. OptSpace was faster than EALM and IALM in computation with small matrices. Sometimes, especially when only few entries are known, it might be useful to use EALM, which was the most reliable, instead of IALM. But in many cases, IALM is comparable to EALM and is faster. SVT would need more iterations to converge, but even the unsuccessful iterations took longer than the successful ones by IALM or Optspace. APGL performed considerably worse than IALM. Since SET is not meant for this type of tasks, it did not perform well. GROUSE was considerably faster than OptSpace for matrices with more entries reveled

when the noise approached the required precision. Results are shown in table 1.

Experiment 3. The third experiment compares IALM and EALM separately. Now we set the maximum number of inner iterations in EALM to 50 and the maximum number of outer iterations to 200. We wanted to demonstrate the dependence of the number of covered problems on the percentage of entries known. One can see, that EALM is more reliable for matrices with few entries revealed. Results are shown in table 2.

Experiment 4. This experiment compares IALM and EALM for different level of noise when enough entries are known. We observed that the computation time both for IALM and EALM remains the same for different values of noise. Results are shown in table 3.

Experiment 5. Now we compare IALM, LMaFit, OptSpace, the three solvers based on different relaxation that performed the best in experiment 2, for larger matrices. LMaFit and Optspace were provided the exact rank. LMaFit and IALM achieved the required precision. OptSpace was much slower and sometimes even less reliable. Results are shown in table 4.

Experiment 6. Sixth experiment uses IALM and LMaFit to solve matrix completion problem with large matrices. Since IALM uses very slow subroutine to project the factors U and V to Ω, we tried to substitute this one by the subroutine from LMaFit, this solver will be called IALM proj. LMaFit was tested both with exact rank provided and rank randomly adjusted up to $\pm 25\%$. Maximum number of iterations was set to 100. By changing the projection subroutine we saved up to 30% of the computation time of IALM. When LMaFit is not given the exact rank it becomes much slower and less reliable. Results are shown in table 5.

Experiment 7. In this experiment we tested all the solvers on matrices with decaying singular values. Singular vectors were drawn from normal distribution orthonormalized by built-in function `orth` The first singular value was set to one. We used different slopes for singular values (in semi-logarithmical scale). Maximum number of iterations was set to 500. We averaged over 3 samples. We see that cvx was able to recover all the matrices. Most of the solvers perform reasonably when the decay is not too steep. Only APGL was able to keep the close-to-required error for all slopes. Results are shown in table 6.

Experiment 8. We decided to test the efficiency of different implementations of the projection onto Ω. We took projections from IALM, SVT and LMaFit. All of them project product of two orthonormal matrices onto Ω without multiplying them. As mentioned already in experiment 6, projection used by IALM is extremely slow and makes the whole computation costly. Full results are shown in table 7.

Experiment 9. Since the only rank estimator for incomplete data is provided in OptSpace, we decided to use IALM, which is able to determine the rank very quickly, as the second estimator. If the rank estimation in IALM stagnates for 5 iterations, we take

this rank as our final rank estimation. We compared the estimators in the following way. We sampled different matrices with noise and let OptSpace and IALM estimate the rank of the complete data. The rank was considered as recovered if the estimate was within the range $(0.75r, 1.25r)$. IALM based estimator performed slightly better (for matrices with few entries). Results are shown in table 8.

Experiment 10. In this experiment we decided to test how trimming in OptSpace works. We constructed a matrix of size 500×500 of rank 3 with overdetermined last 20 columns and computed right singular vectors and singular values of both trimmed and untrimmed matrix. We got the plot 4. In this case trimming prevented the right singular vectors from being concentrated in last few entries. One can also easily estimate the rank from singular values computed after trimming. Unfortunately, it is not always the case.

Experiment 11. We wanted to improve the implementation of`getOptS` in `OptSpace`, i.e. the function that computes the LS solution S and which is the most costly operation in the computation. We avoided multiplying two rectangular matrices by using the projection onto Ω from LMaFit. The improvement is reasonable mainly for very sparse matrices, however the computation time is still not comparable to IALM or LMaFit. Full results are shown in table 9.

Experiment 12. In the following experiment we wanted to learn how fast the methods converge. We decided to plot a few convergence curves. Since the convergence is extremely problem dependent (especially for matrices with very low sampling ratio), this is not a rigorous way how to test the solvers. However, we decided to include the plots in this report for illustration. In the first part we tested LMaFit, IALM, APGL, OptSpace, EALM, SVT, and GROUSE for random matrix of size 100×100, rank 10, and sampling ratios 0.3, 0.4, and 0.5. The plots are produced in the following way. We set our step to 10 iterations, maximal number of iterations to 300 and maximal time to 5 s and stopping criterion to 10^{-16}. For each solver we compute 5 times the first 10 iterations of the algorithm and take the shortest computation time. Then we compute first 20 iterations in the same way, then 30, 40, etc. The computation is terminated when the maximal time or the maximal number of iterations is reached or when the relative error falls bellow 10^{-7}. Rank is given as an input to LMaFit, OptSpace, GROUSE. The curves are plotted in the upper row of figure 5. The dots correspond to one step of computation, i.e. 10 iterations. We see that for these matrices, LMaFit with exact rank is much faster than the other algorithms. For higher sampling ratio, it is advantageous to use IALM than EALM, but in the first case, IALM does not converge. APGL with default setting reaches relatively quickly a reasonable relative error, but then starts to stagnate. It is possible that this could be avoided using some other parameters for acceleration techniques. Concerning the time, OptSpace is not very fast (not optimal implementation in Matlab) but in some cases needs less iterations than LMaFit or IALM to achieve the same relative error. EALM needs only a few steps to achieve a good approximation but in every step needs to compute a SVD several times (maximal number of inner iterations was set to 10 to reduce the time demands of one step). It is a very reliable method but should be substituted by IALM for 'easy' problems. SVT both needs many iterations to converge (if it converges) and every iterations is time demanding. GROUSE in this

case achieves a good approximation in a few first steps but then starts to stagnate very quickly and do not reach a reasonable approximation of the original matrix

We carried out the same experiment for large matrices of size 1000×1000 with rank 10 and different sampling ratios. In this experiment we tested only LMaFit, IALM, APGL and OptSpace. Plots can be found in the lower row of figure 5. We observe a similar behavior for the problems with higher sampling ratio, but in the first case, where none of the methods converge we noticed a difference between nuclear norm based methods and determined-rank based methods.

Experiment 13. We decided to test LMaFit, APGL,IALM, and OptSpace also on real problems. We carried out the experiments with the same data sets as the authors of [14] in section 4.4. Detailed information about the data sets and setting can be found in [14] or on the corresponding web pages. In our case, entries were chosen at random. Maximal number of iterations was set to 100 and stopping criterion to 10^{-3}. As an rank estimator we used IALM. Estimated error was then computed using the originally revealed entries (see [14] for more details). Regarding the estimated error achieved, we do not observe any big difference among the solvers. Computation time corresponds to our expectations based on the results of previous experiments Results are shown in table 10.

Experiment 14. Since all of the nuclear-norm based solvers needs to compute a (partial) SVD in each iteration, we tried to compare two packages for partial SVD implemented in Matlab. All solvers which we included use PROPACK [9]. We compared PROPACK with another package based on Lanczos bidiagonalization implemented in Matlab - IRLBA [1]. We tested the two packages on different types of sparse matrices. We used matrices from previous experiment (Jester and MovieLens), random low-rank matrices (both from two factors and with decaying eigenvalues with slope 0.5) and then two matrices from Matrix Market. We computed first 3-times-more-than-required largest singular values of each matrix using Matlab built-in function `svds` and used these values as a reference. We set the tolerance to 10^{-16} for PROPACK and 10^{-8} to achieve (experimentally) approximately the same error in most cases. Than we ran 5 times both functions and took the shortest time from each five. Table 11 shows the times and euclidean norm of the difference between our reference singular values and computed ones. We see that IRLBA can be considerably faster when we want to compute only a few largest singular values but might become inefficient when more singular values are needed. IRLBA totally failed in computing the SVD of the two matrices from Matrix Market. For matrix `'west2021'` this can be easily treated using lower tolerance in IRLBA. For matrix `'west2021'` we will not get better results even if we set the tolerance to machine precision.The last few singular values will be still to far from the reference ones. Obtained results can be found in table 11.

References

[1] James Baglama, Lothar Reichel: *IRLBA*. Available from `http://www.math.uri.edu/~jbaglama/`.

[2] Laura Balzano, Robert Nowak and Benjamin Recht: *Online Identification and Tracking of Subspaces from Highly Incomplete Data*. `http://arxiv.org/abs/1006.4046v1`, 2010.

[3] Jian-Feng Cai, Emmanuel J. Candès, and Zuowei Shen: *A Singular Value Thresholding Algorithm for Matrix Completion*. `http://arxiv.org/abs/0810.3286`, 2008.

[4] Emmanuel J. Candès and Benjamin Recht: *Exact Matrix Completion via Convex Programming*. Foundations of Computational Mathematics, 2009.

[5] Bingsheng He, Min Tao, and Xiaoming Yuan: *Alternating Direction Method with Gaussian Back Substitution for Separable Convex Programming*. `http://www.optimization-online.org/DB_HTML/2010/12/2871.html`, 2011.

[6] Wei Dai, Olgica Milenkovic and Ely Kerman: Subspace Evolution and Transfer (SET) for Low-Rank Matrix Completion. `http://arxiv.org/abs/1006.2195`, 2010.

[7] Michael Grant and Stephen Boyd: *cvx Users' Guide for cvx version 1.21*. `http://cvxr.com/cvx/cvx_usrguide.pdf`, 2011.

[8] Raghunandan H. Keshavan and Sewoong Oh: *OPTSPACE: A gradient Descent Algorithm on the Grassmann Manifold fo Matrix Completion*. `http://arxiv.org/abs/0910.5260v2`, 2009.

[9] Rasmus Munk Larsen: *PROPACK - Software for large and sparse SVD calculations*. Available from `http://sun.stanford.edu/~rmunk/PROPACK/`.

[10] Kiryung Lee and Yoram Bresler: *ADMiRA: Atomic decomposition for minimum rank approximation*. IEEE Transactions on Information Theory, 2010.

[11] Zhouchen Lin, Minming Chen, and Lequin Wu: *The Augmented Lagrange Multiplier Method for Exact Recovery of Corrupted Low-Rank Matrices*. UIUC Technical Report UILU-ENG-09-2215, November 2009.

[12] Shiqian Ma, Donald Goldfarb, and Lifeng Chen: *Fixed Point and Bregman Iterative Methods for Matrix Rank Minimization*. `http://arxiv.org/abs/0905.1643v2`, 2009.

[13] Min Tao and Xiaoming Yuan: *Recovering Low-Rank and Sparse Components of Matrices from Incomplete and Noisy Observations*. SIAM J. Optim, 2011.

[14] Kim Chuan Toh and Sangwoon Yun: *An Accelerated Proximal Gradient Algorithm for Nuclear Norm Regularized Least Squares Problems*. Pacific J. Optimization, 6 (2010), pp. 615–640.

[15] Zaiwen Wen, Wotao Yin, and Yin Zhang: *Solving a Low-Rank Factorization Model for Matrix Completion by a Nonlinear Successive Over-Relaxation Algorithm*. `http://www.optimization-online.org/DB_HTML/2010/03/2581.html`, 2010.

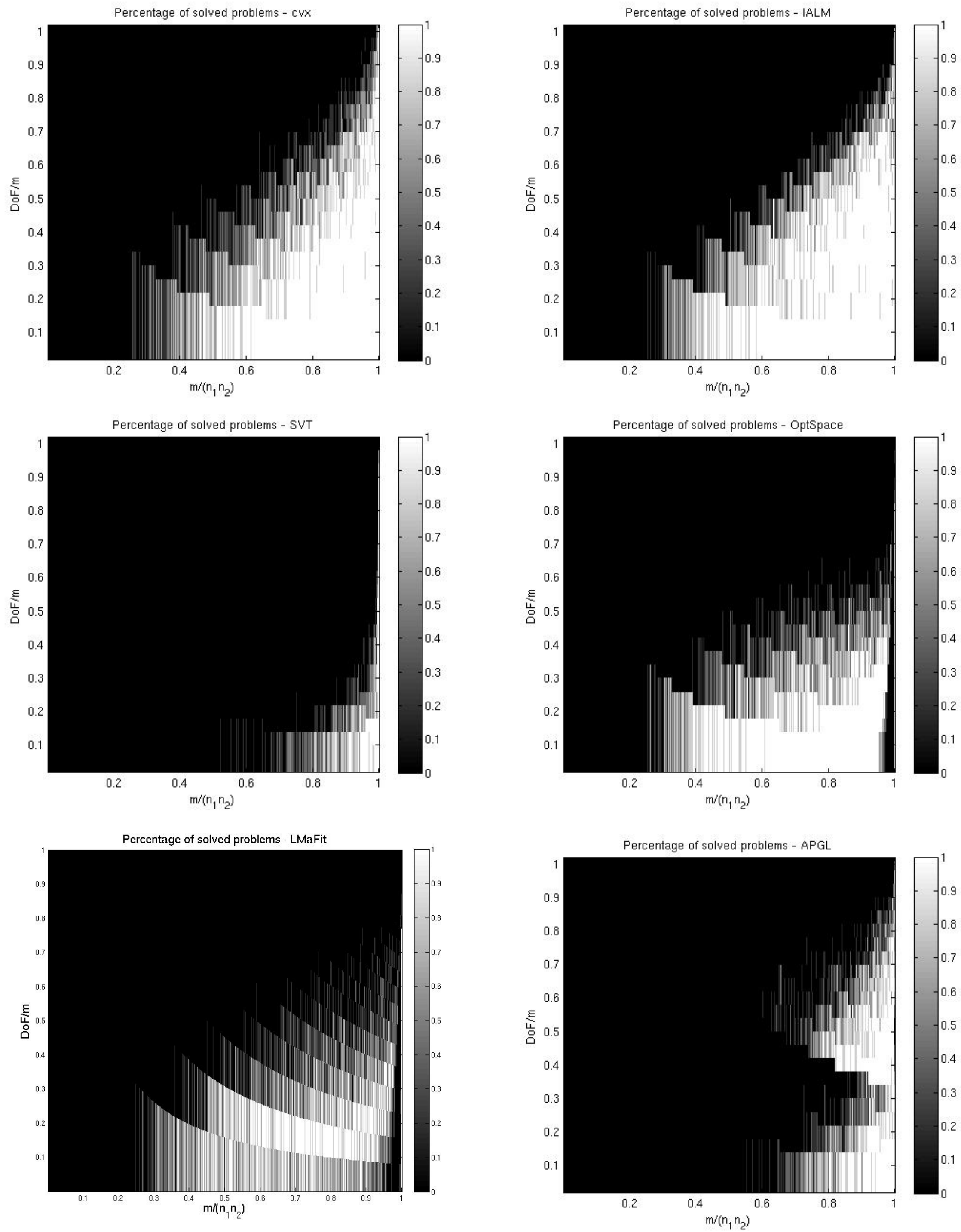

Figure 1: Recovery of full matrices of size 25×25 form sampling of their entries. Maximum number of iterations was set to 50. Code `experiment1.m`

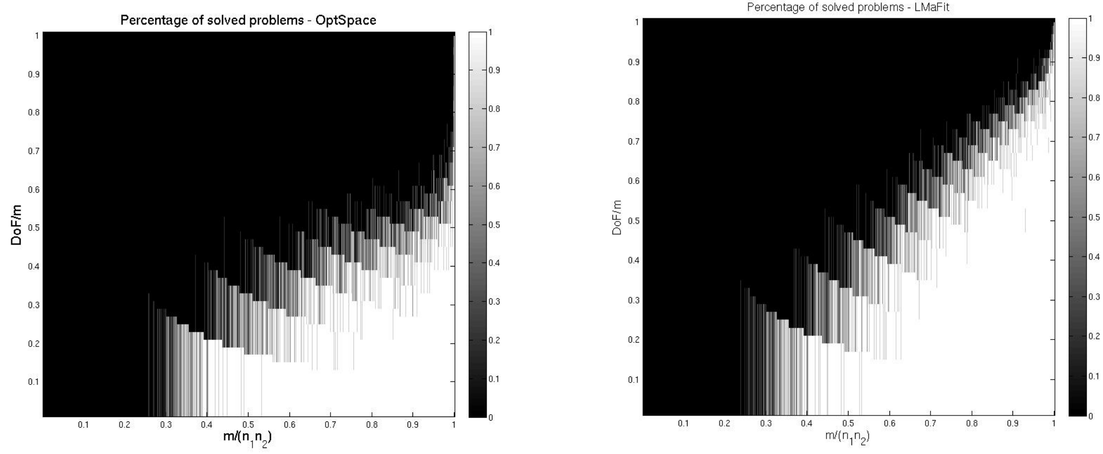

Figure 2: Recovery of full matrices of size 25×25 from sampling of their entries. Maximum number of iterations was set to 50. Exact rank was given as an input. Code `experiment1a.m`

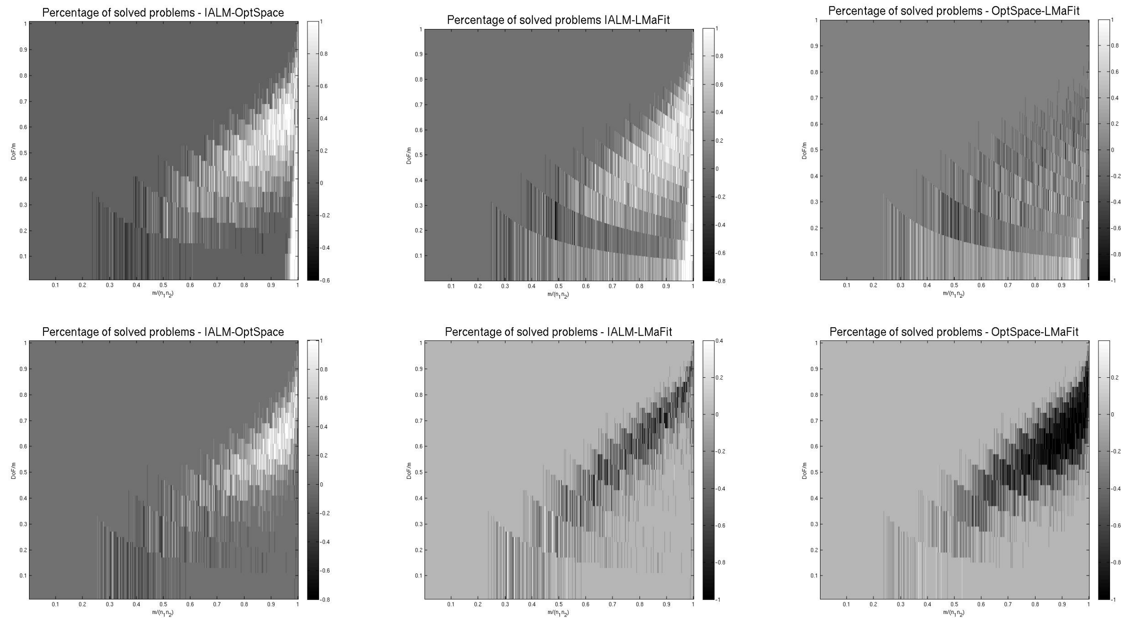

Figure 3: Comparison of IALM, OptSpace and LMaFit. First line corresponds to the rank estimator from OptSpace, second to the correct rank given directly. Lighter shade corresponds to advantage of the first mentioned solver, e.g. in the first picture, roughly speaking, IALM performs better than OptSpace.

Problem				IALM		SVT		OptSpace		APGL		EALM		SET		GROUSE		LMaFit	
n	r	SR	noise	solved	time	solved	time	solved	time	solved	time	solved	time	solved	time	solved	time	solved	time
50	1	0.20	0e+00	60 %	0.6 s	0 %	0.6 s	100 %	0.1 s	0 %	0.6 s	100 %	1.7 s	0 %	3.5 s	0 %	0.3 s	100 %	0.0 s
50	1	0.20	1e-07	60 %	0.5 s	0 %	0.6 s	100 %	0.1 s	0 %	0.6 s	100 %	1.7 s	0 %	3.5 s	0 %	0.3 s	100 %	0.0 s
50	1	0.20	1e-04	60 %	0.6 s	0 %	0.6 s	100 %	0.1 s	0 %	0.6 s	100 %	1.8 s	0 %	3.5 s	0 %	0.3 s	100 %	0.0 s
50	1	0.40	0e+00	100 %	0.1 s	100 %	0.7 s	100 %	0.0 s	0 %	0.4 s	100 %	0.8 s	80 %	1.9 s	40 %	0.3 s	100 %	0.0 s
50	1	0.40	1e-07	100 %	0.1 s	100 %	0.7 s	100 %	0.0 s	0 %	0.4 s	100 %	0.8 s	80 %	2.0 s	40 %	0.3 s	100 %	0.0 s
50	1	0.40	1e-04	100 %	0.2 s	100 %	0.8 s	100 %	0.0 s	0 %	0.4 s	100 %	0.9 s	80 %	1.9 s	40 %	0.3 s	100 %	0.0 s
50	1	0.70	0e+00	100 %	0.1 s	100 %	0.8 s	100 %	0.0 s	100 %	0.2 s	100 %	0.4 s	20 %	1.3 s	100 %	0.3 s	100 %	0.0 s
50	1	0.70	1e-07	100 %	0.1 s	100 %	0.8 s	100 %	0.0 s	100 %	0.2 s	100 %	0.4 s	20 %	1.3 s	100 %	0.3 s	100 %	0.0 s
50	1	0.70	1e-04	100 %	0.1 s	100 %	0.8 s	100 %	0.0 s	100 %	0.3 s	100 %	0.5 s	20 %	1.3 s	100 %	0.3 s	100 %	0.0 s
50	5	0.40	0e+00	0 %	0.3 s	0 %	0.9 s	0 %	0.4 s	0 %	0.8 s	100 %	2.2 s	0 %	18.7 s	0 %	0.3 s	60 %	0.1 s
50	5	0.40	1e-07	0 %	0.3 s	0 %	0.8 s	0 %	0.6 s	0 %	0.9 s	100 %	2.2 s	0 %	18.7 s	0 %	0.3 s	60 %	0.0 s
50	5	0.40	1e-04	0 %	0.3 s	0 %	0.8 s	0 %	0.4 s	0 %	0.8 s	100 %	2.3 s	0 %	18.5 s	0 %	0.3 s	60 %	0.0 s
50	5	0.70	0e+00	100 %	0.2 s	20 %	0.9 s	100 %	0.2 s	100 %	0.4 s	100 %	0.6 s	0 %	8.6 s	20 %	0.3 s	100 %	0.0 s
50	5	0.70	1e-07	100 %	0.2 s	20 %	0.9 s	100 %	0.2 s	100 %	0.3 s	100 %	0.6 s	0 %	8.7 s	20 %	0.3 s	100 %	0.0 s
50	5	0.70	1e-04	100 %	0.2 s	20 %	0.9 s	100 %	0.5 s	100 %	0.4 s	100 %	0.6 s	0 %	8.7 s	20 %	0.3 s	100 %	0.0 s
200	4	0.20	0e+00	100 %	0.5 s	0 %	3.1 s	100 %	0.4 s	0 %	1.0 s	100 %	2.1 s	0 %	49.5 s	80 %	1.2 s	100 %	0.1 s
200	4	0.20	1e-07	100 %	0.5 s	0 %	3.1 s	100 %	0.4 s	0 %	1.0 s	100 %	2.1 s	0 %	49.9 s	80 %	1.2 s	100 %	0.1 s
200	4	0.20	1e-04	100 %	0.6 s	0 %	3.0 s	100 %	1.4 s	0 %	0.9 s	100 %	2.2 s	0 %	49.4 s	80 %	1.2 s	100 %	0.1 s
200	4	0.40	0e+00	100 %	0.2 s	100 %	2.7 s	100 %	0.3 s	80 %	0.7 s	100 %	1.5 s	60 %	33.7 s	100 %	1.3 s	100 %	0.0 s
200	4	0.40	1e-07	100 %	0.2 s	100 %	2.7 s	100 %	0.3 s	80 %	0.7 s	100 %	1.5 s	60 %	33.8 s	100 %	1.3 s	100 %	0.1 s
200	4	0.40	1e-04	100 %	0.3 s	100 %	3.0 s	100 %	1.9 s	80 %	0.7 s	100 %	1.7 s	60 %	33.5 s	100 %	1.3 s	100 %	0.0 s
200	4	0.70	0e+00	100 %	0.2 s	100 %	2.3 s	100 %	0.3 s	100 %	0.6 s	100 %	1.0 s	40 %	23.3 s	100 %	1.5 s	100 %	0.0 s
200	4	0.70	1e-07	100 %	0.2 s	100 %	2.3 s	100 %	0.3 s	100 %	0.5 s	100 %	1.0 s	40 %	23.5 s	100 %	1.5 s	100 %	0.0 s
200	4	0.70	1e-04	100 %	0.2 s	100 %	2.6 s	100 %	2.4 s	100 %	0.6 s	100 %	1.1 s	40 %	22.8 s	100 %	1.5 s	100 %	0.0 s
200	20	0.40	0e+00	100 %	2.2 s	0 %	45.8 s	100 %	28.7 s	0 %	14.7 s	100 %	4.8 s	0 %	238.7 s	0 %	2.8 s	100 %	0.2 s
200	20	0.40	1e-07	100 %	2.2 s	0 %	45.7 s	100 %	27.6 s	0 %	14.3 s	100 %	4.8 s	0 %	239.1 s	0 %	2.8 s	100 %	0.1 s
200	20	0.40	1e-04	100 %	2.3 s	0 %	45.7 s	100 %	29.4 s	0 %	14.1 s	100 %	4.9 s	0 %	239.9 s	0 %	2.8 s	100 %	0.1 s
200	20	0.70	0e+00	100 %	0.5 s	100 %	5.8 s	100 %	10.0 s	100 %	1.1 s	100 %	2.4 s	0 %	84.7 s	100 %	3.6 s	100 %	0.1 s
200	20	0.70	1e-07	100 %	0.5 s	100 %	5.8 s	100 %	9.8 s	100 %	1.1 s	100 %	2.4 s	0 %	84.3 s	100 %	3.5 s	100 %	0.0 s
200	20	0.70	1e-04	100 %	0.5 s	100 %	5.8 s	100 %	35.5 s	100 %	1.1 s	100 %	2.5 s	0 %	84.1 s	100 %	3.6 s	100 %	0.1 s

Table 1: Comparison of the solvers for matrices of size 50×50 and 200×200 with different ranks, sampling ratio, and added noise. Maximum number of iterations was set to 100. Maximum number of inner iterations for EALM to 50. Code `experiment2.m`

Problem				EALM		IALM	
n	r	SR	noise	error	time	error	time
50	5	0.40	1e-07	4.8e-04	2.18 s	5.1e-01	0.30 s
50	5	0.50	1e-07	2.9e-04	1.10 s	7.3e-02	0.46 s
50	5	0.60	1e-07	2.0e-04	0.81 s	2.4e-04	0.32 s

Table 2: Comparison of EALM and IALM for matrices with different percentage of the entries known. Code `experiment3.m`

Problem				EALM		IALM	
n	r	SR	noise	error	time	error	time
50	5	0.60	0e+00	2.0e-04	0.81 s	2.4e-04	0.31 s
50	5	0.60	1e-07	2.0e-04	0.81 s	2.4e-04	0.30 s
50	5	0.60	1e-04	1.3e-04	0.84 s	1.4e-04	0.35 s

Table 3: Comparison of EALM and IALM for matrices with different noise level. Code `experiment4.m`

Problem				IALM		LMaFit		OptSpace	
n	r	SR	noise	error	time	error	time	error	time
1000	10	0.05	0e+00	3.9e-04	10.78 s	6.5e-04	0.69 s	1.5e-03	72.06 s
1000	10	0.05	1e-07	3.9e-04	10.84 s	6.5e-04	0.68 s	1.5e-03	67.83 s
1000	10	0.05	1e-04	2.7e-04	11.67 s	6.5e-04	0.64 s	1.5e-03	57.05 s
1000	20	0.10	0e+00	2.8e-04	9.32 s	2.6e-04	1.71 s	2.8e-04	247.35 s
1000	20	0.10	1e-07	2.8e-04	9.31 s	2.6e-04	1.68 s	2.8e-04	267.68 s
1000	20	0.10	1e-04	1.9e-04	9.98 s	1.8e-04	1.80 s	2.9e-04	266.67 s

Table 4: Comparison of IALM, LMaFit and OptSpace for larger matrices. Exact rank was provided to OptSpace and LMaFit. Code `experiment5.m`

Problem				IALM		IALM proj		LMaFit		LMaFit ex.rank	
n	r	SR	noise	error	time	error	time	error	time	error	time
1000	2	0.02	1e-04	1.8e-04	8.60 s	1.8e-04	5.50 s	5.0e-04	0.11 s	5.0e-04	0.09 s
1000	2	0.05	1e-04	4.7e-05	2.17 s	4.7e-05	1.54 s	5.8e-05	0.07 s	5.8e-05	0.09 s
1000	5	0.05	1e-04	1.2e-04	3.23 s	1.2e-04	2.27 s	1.1e-04	0.23 s	1.1e-04	0.22 s
5000	10	0.02	1e-04	1.0e-04	58.46 s	1.0e-04	28.22 s	9.3e-05	2.71 s	9.3e-05	2.70 s
5000	10	0.05	1e-04	4.3e-05	69.45 s	4.3e-05	51.22 s	3.6e-05	4.14 s	3.6e-05	4.16 s
5000	25	0.02	1e-04	6.9e-01	3741.77 s	6.9e-01	3055.67 s	4.6e-02	15.81 s	2.0e-03	15.42 s
5000	25	0.05	1e-04	7.8e-05	119.76 s	7.8e-05	83.26 s	5.1e-03	36.65 s	8.6e-05	12.56 s

Table 5: Comparison of IALM, LMaFit for large matrices. IALM with original projection onto Ω and with projection taken from LMaFit. Code `experiment6.m`

Problem				IALM	SVT	OptSpace	APGL	EALM	SET	GROUSE	LMaFit	cvx
n	r	SR	slope	error	error	error	error	error	error	error	error	error
30	3	0.80	-0.1	9.8e-05	4.5e-02	7.8e-05	2.3e-03	1.1e-04	3.9e-03	9.8e-01	1.4e-04	8.5e-10
30	3	0.80	-0.2	9.7e-05	6.1e-02	7.7e-05	2.3e-03	8.5e-05	4.9e-03	9.8e-01	1.2e-04	2.4e-09
30	3	0.80	-0.5	3.2e-01	1.1e-01	5.1e-02	2.2e-03	3.2e-01	7.7e-03	9.7e-01	1.5e-04	1.0e-08
30	3	0.80	-1.0	1.0e-01	1.0e-01	1.0e-02	1.5e-03	1.0e-01	9.4e-03	9.7e-01	2.5e-01	2.1e-08
30	3	0.80	-2.0	1.0e-02	2.3e-02	1.0e-02	8.3e-04	1.0e-02	1.1e-02	9.7e-01	2.7e-01	2.5e-09

Table 6: Comparison of the solvers for matrices with decaying singular values. No noise was added. 'slope' denotes the distance between two neighboring singular values in logarithmical scale. Code `experiment7.m`

Problem			IALM	SVT	LMaFit
n	r	SR	time	time	time
500	5	0.01	1.5e-02 s	1.6e-03 s	9.8e-05 s
2000	20	0.01	1.0e+00 s	7.0e-03 s	5.5e-03 s
10000	100	0.01	1.4e+03 s	3.3e+00 s	7.0e-01 s

Table 7: Efficiency of the different methods of projection onto set Ω. Code `test_project_omega.m`

Problem				OptSpace	IALM
n	r	SR	noise	recovered	recovered
200	4	0.10	1e-04	30 %	90 %
200	4	0.25	1e-04	100 %	100 %
200	4	0.40	1e-04	100 %	100 %
200	10	0.10	1e-04	0 %	0 %
200	10	0.25	1e-04	60 %	100 %
200	10	0.40	1e-04	100 %	100 %
1000	20	0.10	1e-04	100 %	100 %
1000	20	0.25	1e-04	100 %	100 %
1000	20	0.40	1e-04	100 %	100 %
1000	50	0.10	1e-04	0 %	0 %
1000	50	0.25	1e-04	100 %	100 %
1000	50	0.40	1e-04	100 %	100 %

Table 8: Efficiency of the rank estimators. The rank was considered as recovered if the estimate was within the range $(0.75r, 1.25r)$. Code `test_rank_est.m`

Problem			original	new	
n	r	SR	time	time	% saved
50	1	0.10	0.000 s	0.000 s	23 %
50	1	0.20	0.000 s	0.000 s	16 %
50	1	0.50	0.000 s	0.000 s	8 %
50	5	0.10	0.002 s	0.002 s	30 %
50	5	0.20	0.002 s	0.002 s	25 %
50	5	0.50	0.003 s	0.002 s	23 %
200	4	0.10	0.004 s	0.001 s	69 %
200	4	0.20	0.005 s	0.002 s	57 %
200	4	0.50	0.007 s	0.005 s	30 %
200	20	0.10	0.151 s	0.069 s	54 %
200	20	0.20	0.186 s	0.104 s	44 %
200	20	0.50	0.280 s	0.213 s	24 %
500	10	0.10	0.179 s	0.051 s	71 %
500	10	0.20	0.217 s	0.096 s	56 %
500	10	0.50	0.329 s	0.231 s	30 %
500	50	0.10	10.420 s	4.064 s	61 %
500	50	0.20	13.452 s	6.853 s	49 %
500	50	0.50	21.412 s	15.586 s	27 %

Table 9: Comparison of the computation time of the original `getoptS` (computes the least squares solution S in OptSpace) and `getoptS` with the projection taken from LMaFit. Code `test_getoptS.m`

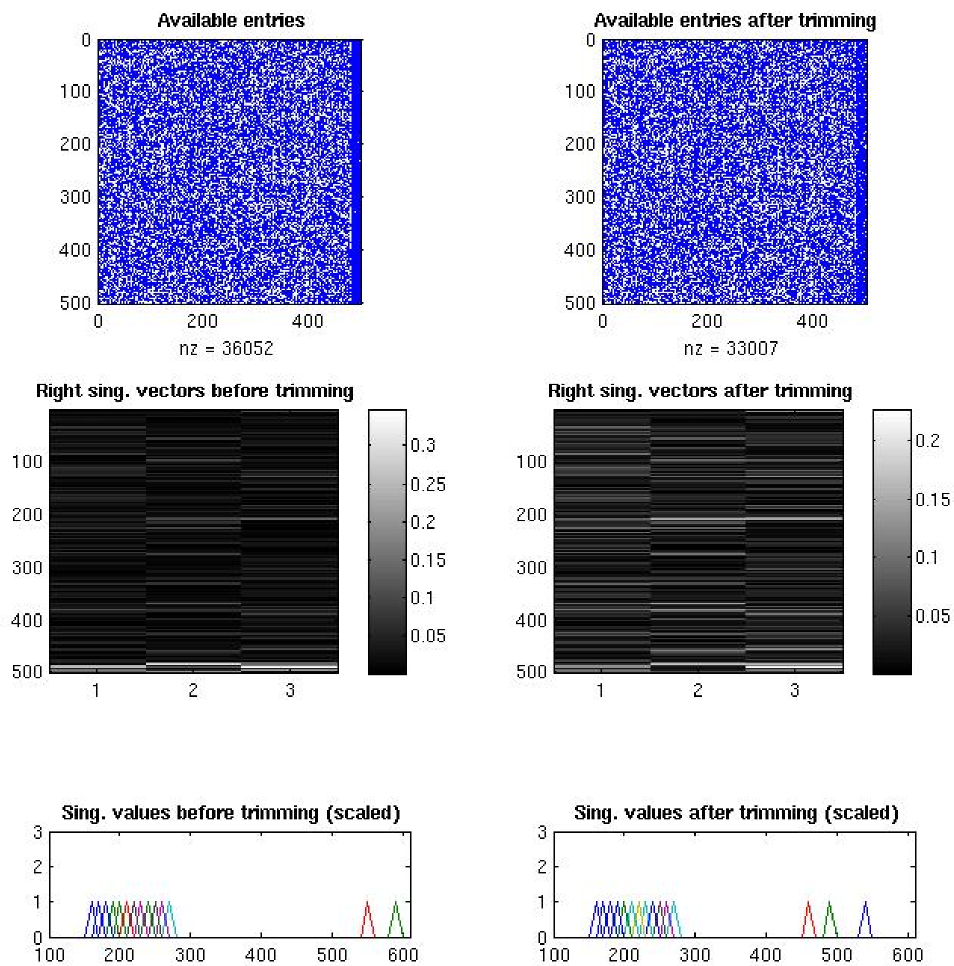

Figure 4: Trimming procedure implemented in OptSpace. Matrix 500×500 of rank 3 with 20 overdetermined columns. Code `OptSpace_test.m`

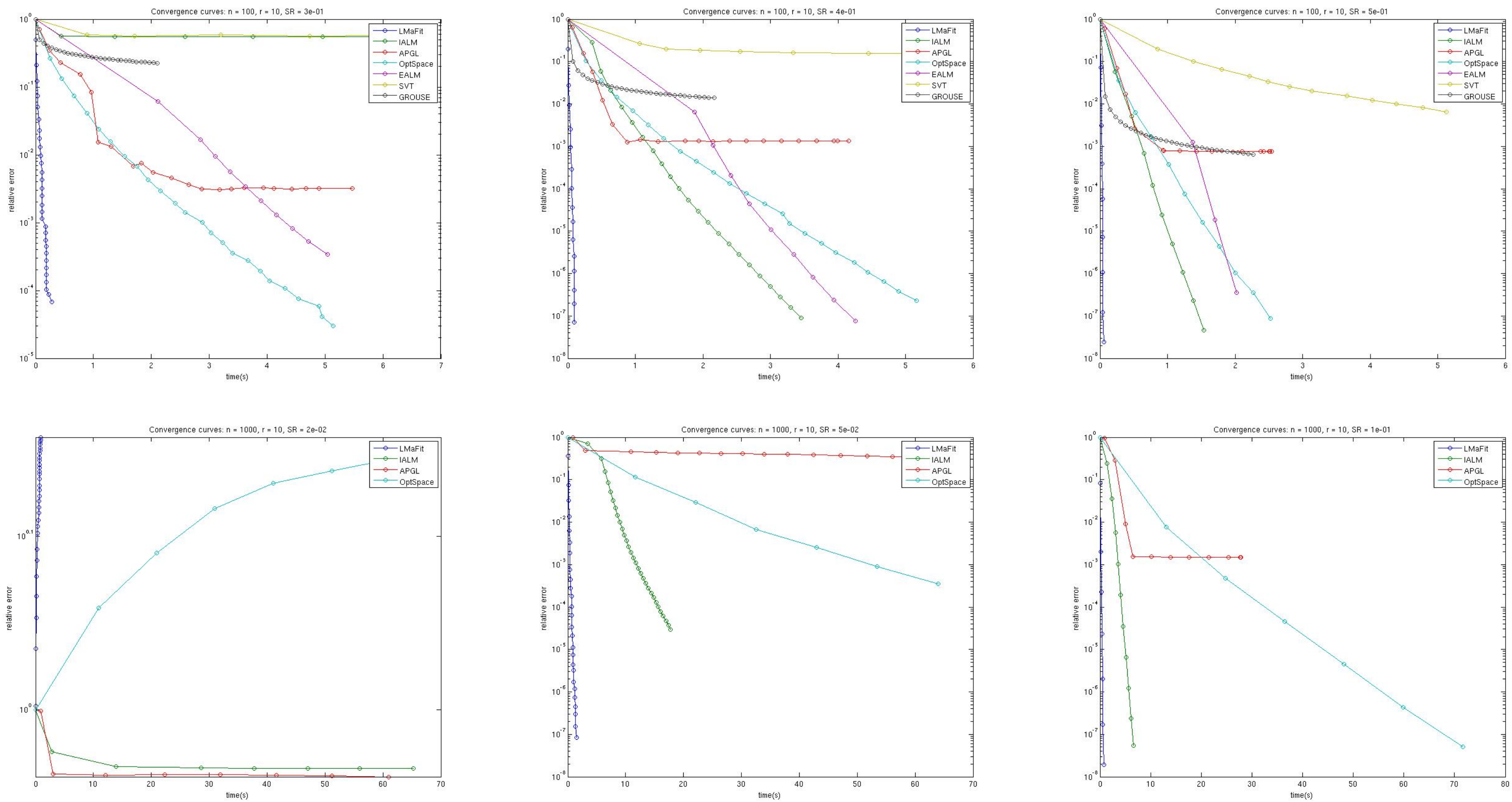

Figure 5: Convergence curves for matrices of different size and sampling ratio (only one matrix for each setting). Code `conv_curves_small.m` and `conv_curves_large.m`

Problem					APGL				IALM				EALM				LMaFit			
data	n_1	n_2	$\frac{\text{nnz}(A)}{n_1 n_2}$	$\frac{\lvert\Omega\rvert}{n_1 n_2}$	error	NMAE	rank	time	error	NMAE	rank	time	error	NMAE	rank	time	error	NMAE	rank	time
`jester-1`	24983	100	0.72	0.14	8.6e-01	1.6e-01	91	35.89 s	7.9e-01	1.5e-01	100	22.89 s	9.3e-01	1.9e-01	1	236.47 s	8.9e-01	1.8e-01	100	0.24 s
`jester-2`	23500	100	0.73	0.15	8.5e-01	1.6e-01	91	33.89 s	8.0e-01	1.5e-01	100	24.94 s	9.1e-01	1.9e-01	1	216.44 s	8.9e-01	1.8e-01	100	0.19 s
`jester-3`	24938	100	0.25	0.05	8.9e-01	1.7e-01	81	27.02 s	9.8e-01	2.0e-01	2	28.05 s	1.3e+00	2.6e-01	2	381.33 s	1.2e+00	2.3e-01	10	1.36 s
`MovieLens_small`	943	1682	0.06	0.03	2.5e-01	1.7e-01	6	10.38 s	2.6e-01	1.9e-01	1	3.53 s	2.5e-01	1.8e-01	1	23.59 s	2.7e-01	1.5e-01	36	1.13 s
`MovieLens_large`	6040	3706	0.04	0.02	2.4e-01	1.7e-01	6	161.45 s	2.4e-01	1.8e-01	1	20.89 s	2.4e-01	1.8e-01	1	143.06 s	2.3e-01	1.5e-01	71	13.50 s

Table 10: Comparison of the solvers for real problems - Jester (jokes ratings) and MovieLens (movie ratings). Here A corresponds to the matrix which is available (all rankings that are stored) and Ω to the set of entries that are used for computation. Code `experiment_real.m`

Problem		PROPACK			IRLBA		
name	size	error	time	multipl	error	time	multipl
`jester-1`	24983 × 100						
#sv=20		7.2e-12	0.80 s	138	1.7e-11	0.85 s	164
#sv=50		9.7e-12	1.29 s	200	1.9e-11	2.63 s	362
#sv=90		1.0e-11	1.30 s	200	1.3e-11	1.67 s	202
`MovieLens_small`	943 × 1682						
#sv=1		1.1e-13	0.01 s	20	2.3e-13	0.01 s	18
#sv=5		1.2e-12	0.02 s	50	9.8e-13	0.02 s	50
#sv=10		1.5e-12	0.04 s	68	1.1e-12	0.03 s	78
#sv=20		1.6e-12	0.08 s	130	1.5e-12	0.08 s	164
`MovieLens_large`	6040 × 3706						
#sv=1		1.4e-12	0.08 s	20	1.1e-12	0.07 s	18
#sv=5		1.7e-12	0.17 s	44	2.8e-12	0.22 s	56
#sv=10		4.0e-12	0.25 s	62	5.8e-12	0.29 s	72
#sv=20		4.4e-12	0.53 s	120	5.3e-12	0.63 s	146
`random sparse`	1000 × 1000						
#sv=1		6.4e-14	0.02 s	66	5.0e-14	0.01 s	66
#sv=3		1.4e-13	0.02 s	72	1.6e-13	0.02 s	88
#sv=10		1.3e-13	0.05 s	152	2.4e-13	0.05 s	234
`random sparse`	10000 × 10000						
#sv=10		5.1e-12	2.28 s	358	6.8e-12	2.79 s	528
#sv=30		9.8e-12	1.81 s	324	1.1e-11	1.80 s	280
#sv=100		1.1e-11	33.00 s	1324	1.3e-11	74.94 s	3858
`decaying sparse`	1000 × 1000						
#sv=1		1.0e-17	0.01 s	42	6.9e-18	0.01 s	30
#sv=3		2.9e-17	0.02 s	60	6.1e-17	0.01 s	76
#sv=10		7.5e-17	0.04 s	140	6.9e-17	0.04 s	180
`decaying sparse`	10000 × 10000						
#sv=10		1.1e-16	1.06 s	182	1.1e-16	1.21 s	228
#sv=30		1.7e-16	2.54 s	344	2.2e-16	3.76 s	574
#sv=100		2.0e-16	16.70 s	872	2.9e-16	29.09 s	1548
`abb313`	313 × 176						
#sv=10		4.1e-14	0.03 s	140	3.4e-14	0.01 s	108
#sv=50		8.5e-14	0.11 s	268	2.2e+00	0.05 s	170
#sv=100		7.9e-14	0.10 s	268	1.8e+00	0.17 s	252
`west2021`	2021 × 2021						
#sv=10		6.1e-09	0.32 s	304	1.8e-08	0.17 s	552
#sv=50		1.8e-08	0.13 s	202	1.8e-08	0.08 s	134
#sv=100		1.8e-08	0.56 s	402	5.9e-01	0.24 s	258

Table 11: Comparison of two Matlab packages for partial SVD. The first three matrices are taken from the previous experiment. Next four are low-rank random, of ranks 3, 30, 3, and 30 respectively, with sampling ratio 0.01. Last three are sparse matrices from Matrix Market. '#sv' is number of singular values. 'Error' corresponds to the euclidean norm of the difference between the reference singular values and the computed ones. 'Time' is the shortest time of five identical computations. 'Multipl' expresses the number of multiplications of kind '$y = Ax$' or '$y = A^T x$'. Code `IRLBA_PROPACK.m`