User's Guide for LMaFit: Low-rank Matrix Fitting

Yin Zhang
Department of CAAM
Rice University, Houston, Texas, 77005

(CAAM Technical Report TR09-28)

(Versions beta-1: August 23, 2009)
(Versions beta-2: September 8, 2009)

Abstract

This User's Guide describes the functionality and basic usage of the Matlab package LMaFit for low-rank matrix optimization. It also briefly explains the formulations and algorithms used.

Version beta-2 change log: Only the matrix completion code is changed: (1) the field `Prob.Unknwn` has been eliminated; (2) a new option `opts.Zfull` is added (see page 6) to help complete large matrices with very low ranks.

1 Introduction

LMaFit contains a collection of solvers that can be used to solve three classes of low-rank matrix optimization problems: (1) Matrix Completion (MC), (2) Sparse Matrix Separation (SMS), and (3) Matrix Compressive Sensing (MCS). Although the first class is a subclass of the third, it is special enough to deserve a solver of its own to take advantages of its special structures.

The original, idealistic models for MC, SMS, and MCS are, respectively,

$$\min_{U\in\mathbb{R}^{m\times n}} \text{rank}(U) \text{ s.t. } U_{ij} = A_{ij},\ \forall (i,j)\in\mathcal{E}, \tag{1a}$$

$$\min_{U,Z\in\mathbb{R}^{m\times n}} \text{rank}(U) + \mu\|\text{vec}(Z)\|_1 \text{ s.t. } U + Z = A, \tag{1b}$$

$$\min_{U\in\mathbb{R}^{m\times n}} \text{rank}(U) \text{ s.t. } \mathcal{A}(U) = b, \tag{1c}$$

where $\mathcal{E} \subset \{(i,j) : 1 \le i \le m, 1 \le j \le n\}$, $\mu > 0$, vec(Z) is the vectorization of the matrix Z, and $\mathcal{A}(\cdot)$ is a linear map from $\mathbb{R}^{m\times n}$ to $\mathbb{R}^{n_b}$ for some $n_b < mn$.

Such rank minimization problems are generally hard. The standard approach is to replace the rank of U by the sum of its singular values, or the so-called nuclear norm of U. Such convex relaxations are known to produce the minimum rank solutions under suitable conditions (see, for example, [1, 2, 3]). Still, nuclear norm minimization problems can become excessively costly as problem sizes and ranks increase. As a result, algorithms based on nuclear norm minimization so far only have limited capacity for solving large-scale problems due to their slow speed when ranks are not extremely low. Further information on some references and algorithms for nuclear norm minimization can be found at the web sites: `http://www.dsp.ece.rice.edu/cs` and `http://www.stanford.edu/~raghuram/optspace/papers.html`.

To improve our problem-solving ability for large-scale problems (particularly when ranks are not extremely low), we consider slightly more restrictive versions of the three classes of problems. First let us denote the set of rank k matrices as

$$\mathbb{R}^{m\times n}(k) \triangleq \{U \in \mathbb{R}^{m\times n} : \text{rank}(U) = k\}. \tag{2}$$

Currently, LMaFit can be used to attack the following three problems:

$$\text{Find } U \in \mathbb{R}^{m\times n}(k) \text{ so that } U_{ij} \approx A_{ij},\ (i,j) \in \mathcal{E}, \tag{3a}$$

$$\text{Find } U \in \mathbb{R}^{m\times n}(k) \text{ and sparse } Z \text{ so that } U + Z \approx A, \tag{3b}$$

$$\text{Find } U \in \mathbb{R}^{m\times n}(k) \text{ so that } \mathcal{A}(U) \approx b \text{ where } \mathcal{A}\mathcal{A}^T = I. \tag{3c}$$

These three problems will be called, respectively, MC(k), SMS(k) and MCS(k). It is worth noting that, to accommodate noisy data, exact fidelity is not required in the above problem formulations.

Our specific formulations for solving problems (3a)-(3c) are, respectively,

$$\min_{X,Y,Z} \|XY - Z\|_F^2 \ \text{ s.t. } \ Z_{ij} = A_{ij}, \forall (i,j) \in \mathcal{E}, \tag{4a}$$

$$\min_{X,Y,Z} \|XY + Z - A\|_F^2/2 + \mu\|\text{vec}(Z)\|_1, \tag{4b}$$

$$\min_{X,Y,Z} \|XY - Z\|_F^2 + \rho\|\mathcal{A}(Z) - b\|_2^2, \tag{4c}$$

where $X \in \mathbb{R}^{m\times k}$, $Y \in \mathbb{R}^{k\times n}$, $Z \in \mathbb{R}^{m\times n}$, $\mu \in (0,\infty)$ and $\rho \in (0,\infty]$. When $\rho = \infty$ in the last problem, the fidelity term becomes a constraint. Straightforward alternating minimization

algorithms are applied to these problems. In the constrained case corresponding to $\rho = \infty$, an alternating direction method based on augmented Lagrangian is applied.

Using direct factorization is not a new idea in dealing with low-rank matrix optimization (see [4] for example). After an alpha version of our code was released, we found that very recently, this idea has also been applied to the matrix CS problem by Haldar and Hernando [5] using a different formulation, i.e., directly minimizing $\|\mathcal{A}(XY) - b\|_2^2$ alternatingly with respect to low-rank X and Y. The authors tested their algorithm on small problems with matrices up to 50 by 50. (Since the method in [5] requires solving linear least squares problems involving $n_b \times mk$ or $n_b \times nk$ coefficient matrices, it is questionable that this approach would be effective for large-scale problems.)

2 Quick Start

LMaFit can be downloaded from the website:

`http://www.caam.rice.edu/~optimization/L1/LMaFit/`

It has a simple interface with 2 inputs and 1 output:

```
Out = lmafit(Prob, opts);
```

where `Prob` is a structure for problem specification and `opts` is a structure carrying options such as the maximum number of iterations. The output `Out` is a structure for solutions. Further information for the output and inputs structures is given in the next section.

After downloading and unzipping the package, you will have a new directory: `LMaFit-xx` where "xx" will vary with LMaFit versions. Get into this directory and run the Matlab script `Run_Me_1st.m`, which sets necessary path and tries to compile a C++ code for fast Walsh-Hadamard transform into a Matlab `mex` file (as such you will need a relevant compiler installed on your system). Assuming that everything goes as it is supposed, then you will be able to run the `demo` files in the directory (as well as the `test` files in the `Tests` directory). These `demo` files demonstrate how to invoke the solvers.

Note: A function `setProb.m` is provided to help set up problem instances. Please see the `demo` files for its usage.

3 Output and Input Structures

3.1 Output

The output variable `Out` is a Matlab structure with two or three main fields:

- `Output.X` and `Output.Y`: The former holds an m by k matrix X and the latter a k by n matrix Y. The product $U = XY$ is supposed to be a rank-k solution to a relevant problem.

- `Output.Z`: This additional field exists in the case of (4b) representing a sparse matrix Z so that $XY + Z \approx A$.

Other fields exist that contain secondary output information. For example, `Output.iter` gives the number of iterations taken.

3.2 Input

An input structure `Porb` always has the following 3 fields for matrix variable size, rank and problem type, respectively,

- `Prob.size = [m,n];`

- `Prob.rank = k;`

- `Prob.type = 'MC'` or `'SMS'` or `'MCS';`

In addition, each problem type requires additional fields.

3.2.1 Matrix Completion

An MC instance requires 2 additional fields:

- `Prob.known`: a vector containing a subset of the index set $\{1, 2, \cdots, mn\}$,

- `Prob.data`: a column vector of the same length as `Prob.known`,

where `Prob.known` corresponds to the column-wise, linear arrangement of the index set $\mathcal{E}$ in (4a) and `Prob.data` contains the column-wise, linear arrangement of the known elements $\{A_{ij} : (i, j) \in \mathcal{E}\}$. (The field `Prob.unknown` has been eliminated from version beta-2.)

3.2.2 Sparse Matrix Separation

An SMS instance requires 2 additional fields:

- `Prob.A`: an m by n matrix,

- `Prob.mu`: a value for penalty parameter $\mu > 0$.

3.2.3 Matrix Compressive Sensing

An MCS instance requires 3 additional fields:

- `Prob.Amap`: a linear map from $\mathbb{R}^{m\times n}$ to $\mathbb{R}^{n_b}$,
- `Prob.b`: a column vector of the length n_b,
- `Prob.rho`: a value for penalty parameter $\rho > 0$ or $\rho =$ `inf`,

where the first two fields represent the left and right-hand sides of the equation $\mathcal{A}(U) = b$, respectively, which will be treated as a constraint when $\rho =$ `inf`.

Specifically, `Prob.Amap` can be (i) a Matlab structure, or (ii) a Matlab function handle. In case (i), the structure `Prob.Amap` should have two fields:

```
Prob.Amap.times -- a function handle for Amap(x),
Prob.Amap.trans -- a function handle for Amap'(y),
```

representing the action of the linear map `Prob.Amap` and that of its adjoint. That is, `Prob.Amap.times(·)` is from $\mathbb{R}^{m\times n}$ to $\mathbb{R}^{n_b}$ and `Prob.Amap.trans(·)` from $\mathbb{R}^{n_b}$ to $\mathbb{R}^{m\times n}$. For an example on how such a structure is defined, see the function `pdwht_operator.m` in the `Utilities` directory. In case (ii) the Matlab function, say `F`, represented by the handle `Prob.Amap` should have two modes:

```
F(x,1) represents Amap(x),
F(y,2) represents Amap'(y).
```

3.3 Options

The following are the input options that can be reset by the user. The values in the brackets "[ ]" are default values.

```
opts.tol = [set by user]      (stopping tolerance)
opts.maxit = [500];           (maximum iterations)
opts.print = 0 or [1];        (silence/verbal mode)
```

The option `opts.tol` specifies the stopping tolerance and must be set by the user. To get the best performance, its value should be set more or less consistent with the noise levels in the observed data. In most practical situations where noise inevitably exists, `opts.tol` values between 1E-3 to 1E-5 should generally be sufficient. Setting a tolerance value much

lower than noise level would only prolong computational time without the benefit of getting a higher accuracy.

In version beta-2, an additional option, `opts.Zfull`, has been added for solving the MC problem (4a):

```
opts.Zfull = 1 or 0;        (full Z stored or not)
```

The code uses some heuristics to set a default value for this option, but you can overwrite it by specifying a value. In general, for completing large matrices with very low ranks, this option should be set to 0 in which case the matrix variable Z is treated as a sum of a low-rank matrix and a sparse matrix. On the other hand, for smaller matrices with relatively high ranks, the MC code can be faster when Z is stored as a full matrix.

References

[1] Emmanuel Candes and Benjamin Recht. Exact matrix completion via convex optimization. Preprint, 2008.

[2] John Wright, Arvind Ganesh, Shankar Rao, and Yi Ma. Robust Principal Component Analysis: Exact Recovery of Corrupted Low-Rank Matrices by Convex Optimization. Preprint, 2009.

[3] Benjamin Recht, Maryam Fazel, and Pablo A. Parrilo. Guaranteed Minimum Rank Solutions to Linear Matrix Equations via Nuclear Norm Minimization. Preprint 2007.

[4] Samuel Burer and Renato D. C. Monteiro. A Nonlinear Programming Algorithm for Solving Semidefinite Programs via Low-rank Factorization. Mathematical Programming (Series B), 2003.

[5] Justin P. Haldar and Diego Hernando. Rank-Constrained Solutions to Linear Matrix Equations using PowerFactorization. IEEE Signal Processing Letters, 16:584-587, 2009.