Software for weighted structured low-rank approximation

Ivan Markovsky and Konstantin Usevich

Department ELEC
Vrije Universiteit Brussel
{imarkovs,kusevich}@vub.ac.be

Abstract

A software package is presented that computes locally optimal solutions to low-rank approximation problems with the following features:

- *mosaic Hankel structure* constraint on the approximating matrix,
- *weighted 2-norm* approximation criterion,
- *fixed elements* in the approximating matrix,
- *missing elements* in the data matrix, and
- *linear constraints* on an approximating matrix's left kernel basis.

It implements a variable projection type algorithm and allows the user to choose standard local optimization methods for the solution of the parameter optimization problem. For an $m \times n$ data matrix, with $n > m$, the computational complexity of the cost function and derivative evaluation is $O(m^2 n)$. The package is suitable for applications with $n \gg m$. In statistical estimation and data modeling—the main application areas of the package—$n \gg m$ corresponds to modeling of large amount of data by a low-complexity model. Performance results on benchmark system identification problems from the database DAISY and approximate common divisor problems are presented.

Keywords: mosaic Hankel matrix, low-rank approximation, total least squares, system identification, deconvolution, variable projection.

1 Introduction

Structured low-rank approximation is defined as low-rank approximation

$$\text{minimize} \quad \text{over } \widehat{D} \quad \|D - \widehat{D}\| \quad \text{subject to} \quad \operatorname{rank}(\widehat{D}) \leq r$$

with the additional constraint that the approximating matrix $\widehat{D}$ has the same structure as the data matrix D. A typical example where a rank deficient structured matrix arises is when a sequence $p = (p_1, \ldots, p_{n_p})$ satisfies a difference equation with lag $\ell < \lceil n_p/2 \rceil$, *i.e.*,

$$R_0 p_t + R_1 p_{t+1} + \cdots + R_\ell p_{t+\ell} = 0, \qquad \text{for } t = 1, \ldots, n_p - \ell. \tag{DE}$$

The system of equations (DE) is linear in the vector of parameters $R := \begin{bmatrix} R_0 & R_1 & \cdots & R_\ell \end{bmatrix}$, so that it can be written as $R\mathscr{H}_{\ell+1,n_p-\ell}(p) = 0$, where $\mathscr{H}_{\ell+1,n_p-\ell}(p)$ is a Hankel matrix constructed from p. This shows that, for $R \neq 0$, the fact that p satisfies a difference equation (DE) is equivalent to rank deficiency of a Hankel matrix $\mathscr{H}_{\ell+1,n_p-\ell}(p)$.

Many problems in machine learning, system theory, signal processing, and computer algebra can be posed and solved as structured low-rank approximation problem for different types of structures and different approximation criteria (see Section 5 and [21, 22]). In identification and model reduction of linear time-invariant dynamical systems, the structure is block-Hankel. In the computation of approximate greatest common divisor of two polynomials, the structure is Sylvester. In machine learning, the data matrix is often unstructured but the approximation criterion is a weighted 2-norm (or semi-norm, in the case of missing data).

Despite the academic popularity and numerous applications of the structured low-rank approximation problem, the only efficient publicly available software package for structured low-rank approximation is the one of [27]. The

Feature	Old version	New version
matrix structure	block-Hankel/Toeplitz	matrix $\times$ mosaic-Hankel
cost function	2-norm	weighted 2-norm
exact data	whole blocks	arbitrary elements
missing data	not allowed	arbitrary elements
constraints on the optimization variable	unconstrained	linear constraints
interface	Matlab	Matlab, Octave, R

Table 1: Comparison of the old and new versions of the software.

package presented in this paper is a significantly extended version of the software in [27]. The main extensions are summarized in Table 1 and are described in more details in Section 2.

The paper is organized as follows. Section 3 defines the considered weighted structured low-rank approximation and presents Matlab/Octave and R interfaces for calling the underlying C++ solver. Section 4 gives details about the solution method for solving the resulting parameter optimization problem. Implementation and software design issues are extracted in Appendix A. Section 5 lists applications of the package and describes in more details an application for solving scalar autonomous linear time-invariant identification and approximate common divisor problems. Appendix B lists extra options of the software for choosing the optimization method.

2 Main features of the software

1. *Matrix structure specification*

   The software package supports matrix structures of the form

   $$\mathscr{S}(p) := \Phi \mathscr{H}_{\mathbf{m},\mathbf{n}}(p), \tag{$\mathscr{S}$}$$

   where Φ is a full row rank matrix and $\mathscr{H}_{\mathbf{m},\mathbf{n}}$ is a *mosaic Hankel* structure [14], *i.e.*, a $q \times N$ block matrix

   $$\mathscr{H}_{[m_1\ \cdots\ m_q],[n_1\ \cdots\ n_N]}(p) = \begin{bmatrix} \mathscr{H}_{m_1,n_1}(p^{(11)}) & \cdots & \mathscr{H}_{m_1,n_N}(p^{(1N)}) \\ \vdots & & \vdots \\ \mathscr{H}_{m_q,n_1}(p^{(q1)}) & \cdots & \mathscr{H}_{m_q,n_N}(p^{(qN)}) \end{bmatrix}, \tag{$\mathscr{H}_{\mathbf{m},\mathbf{n}}$}$$

   with scalar *Hankel* blocks

   $$\mathscr{H}_{m,n}(p) := \begin{bmatrix} p_1 & p_2 & p_3 & \cdots & p_n \\ p_2 & p_3 & \iddots & & p_{n+1} \\ p_3 & \iddots & & & \vdots \\ \vdots & & & & \\ p_m & p_{m+1} & \cdots & & p_{m+n-1} \end{bmatrix} \in \mathbb{R}^{m\times n}. \tag{$\mathscr{H}_{m,n}$}$$

   ($\mathscr{H}_{\mathbf{m},\mathbf{n}}$) is more general than the block-Hankel and "flexible structure specification", used in the old version of the software (see Section 3 in [27]). In fact, the "flexible structure specification" is equivalent to $\mathscr{H}_{\mathbf{m},\mathbf{n}}$ with equal n_i's. Mosaic Hankel matrices with blocks of different column dimension allow us to solve, for example, system identification problems with multiple trajectories of different lengths [23].

   The matrix Φ further extends the class of mosaic Hankel matrices to (mosaic) Hankel-like matrices. A trivial example is the Toeplitz structure achieved by

   $$\Phi = J_m := \begin{bmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{bmatrix} \in \mathbb{R}^{m\times m} \quad \text{and} \quad \mathscr{H}_{\mathbf{m},\mathbf{n}}(p) = \mathscr{H}_{m,n}(p).$$

(Empty spaces in a matrix denote zeros.) A more interesting example is the *Toeplitz-plus-Hankel* structure, achieved by

$$\Phi = \begin{bmatrix} I_m & J_m \end{bmatrix} = \begin{bmatrix} 1 & & & & 1 \\ & \ddots & & \iddots & \\ & & 1 & 1 & \end{bmatrix} \in \mathbb{R}^{m \times 2m} \quad \text{and} \quad \mathscr{H}_{\mathbf{m},\mathbf{n}}(p) = \begin{bmatrix} \mathscr{H}_{m,n}(p^{(1)}) \\ \mathscr{H}_{m,n}(p^{(2)}) \end{bmatrix}. \qquad (\mathscr{T} + \mathscr{H})$$

2. *Cost function*

   In the old version of the package, the approximation criterion is the 2-norm (unweighted low-rank approximation). In the new version, the approximation criterion is the weighted 2-norm. Weights are needed for example when the accuracy of the elements of p vary, see [43]. Weights can be used also to replace the parameter 2-norm with the Frobenius matrix norm of the error matrix $\mathscr{S}(p - \widehat{p})$. As explained in Note 2, the weighted approximation criterion can be viewed alternatively as a modification of the data matrix structure.

3. *Exact data*

   Structure parameters can be fixed to predefined values. (In the old version of the package only subblocks of the structured matrix could be specified as fixed.) This feature allows us to solve problems, such as system identification under exactly known initial conditions, where the upper-left triangles of the Hankel blocks are fixed, and approximate common divisor computation, where the upper-left and lower-right triangles or the Hankel blocks are fixed.

4. *Missing data*[1]

   A new feature of the software is the ability to deal with missing data. Due to specific experiment design or data corruption, the data may be incomplete in real-life applications. A well known example from machine learning is prediction of user ratings (recommender system), see, *e.g.*, [3]. A possibility to specify missing values in Hankel low-rank approximation problems allows us to solve system identification problems with incomplete or irregularly sampled data.

5. *Total least squares and low-rank approximation*

   In the old version of the package, the optimization variable is the X matrix in the problem of solving approximately an overdetermined system of linear equations $AX \approx B$. This problem may be viewed as a restriction of the low-rank approximation problem to a subclass of matrices. To preserve backward compatibility, the new version of the software can also solve the structured total least-squares problems, see Note 6.

6. *Constraints on the optimization variables*

   The new version of the package uses as optimization variable a matrix R whose rows form a basis for the left kernel of the approximation $\widehat{D}$ and allows specification of linear structure constraints on R. An application where such a constraint is needed is multivariate system identification with fixed observability indices [12]. Other applications are given in Section 5.

7. *Interfaces to scientific computing environments*

   The package provides interface for calling the underlying C++ solver from Matlab, Octave, and R, which are among the most often used computing environments for scientists and engineers.

An advantage of unifying structured and weighted low-rank approximation with missing and fixed values and constraints on the optimization variable is that a single algorithm and a piece of software solves a large variety of problems. Section 5 lists special cases of the generic problem solved by the software package. Each special case is motivated by applications, which are considered in the literature and specialized algorithms are developed for their solution. Only a few of the algorithms reported in the literature, however, are implemented in publicly available software.

[1] Currently (September 26, 2013), this feature is implemented only in an experimental Matlab version of the software. It can be simulated in the C++ solver by assigning "small" weights to the missing parameters and replacing the missing values with zeros.

Fixed values in the approximating matrix can be viewed as an extreme case of weighted low-rank approximation with large weights corresponding to the fixed parameters. Similarly, the missing data case can be viewed as weighted low-rank approximation with small weights corresponding to the missing parameters. Both cases are equivalent to a single singular (zero or infinite weights) weighted low-rank approximation problem and can be solved with the same algorithm.

The generality of the algorithm is not compromised by its computational efficiency. Eliminating the approximation $\widehat{D}$ and exploiting the structure in the resulting nonlinear least squares problem, the cost function and derivative evaluation is done in $O(m^2 n)$ floating point operations [40] for weighted norm without missing values.

The software is written in C++ with Matlab/Octave and R interfaces and contains also the experimental Matlab code, presented in [24]. The latter supports general linear structure and missing values, but is inefficient and can be used only for small size matrices (say, $m \leq n < 100$). In Matlab, the C++ solver can be called directly via the mex function `slra_mex_obj` or via the wrapper function, presented in the paper. The wrapper function calls optionally the experimental version `slra_ext`. The source code of the package is hosted at the following address:

```
http://github.com/slra/slra
```

3 Problem formulation

We denote missing data values by the symbol `NaN` ("not a number") and define the extended set of real numbers $\mathbb{R}_{\mathrm{e}}$ as the union of the set of the real numbers $\mathbb{R}$ and the symbol `NaN`:

$$\mathbb{R}_{\mathrm{e}} := \mathbb{R} \cup \{\texttt{NaN}\}.$$

The considered structured low-rank approximation problem is defined as follows.

Problem 1. Given:

- structure specification $\mathscr{S}$,
- vector of structure parameters $p \in \mathbb{R}_{\mathrm{e}}^{n_p}$,
- nonnegative vector $w \in (\mathbb{R}_+ \cup \{\infty\})^{n_p}$, defining a weighted (semi-)norm

$$\|p\|_w^2 := \sum_{\{\, i \mid p_i \neq \texttt{NaN} \,\}} w_i p_i^2, \text{ and}$$

- desired rank r,

find a structure parameter vector $\widehat{p}$, such that the corresponding matrix $\mathscr{S}(\widehat{p})$ has rank at most r, and is as close as possible to p in the sense of the weighted semi-norm $\|\cdot\|_w$, *i.e.*,

$$\text{minimize} \quad \text{over } \widehat{p} \in \mathbb{R}^{n_p} \quad \|p - \widehat{p}\|_w^2 \quad \text{subject to} \quad \operatorname{rank}\left(\mathscr{S}(\widehat{p})\right) \leq r. \tag{SLRA}$$

Without loss of generality, it is assumed that $m \leq n$. An infinite weight $w_i = \infty$ imposes the equality constraint $\widehat{p}_i = p_i$ on the optimization problem (SLRA).

Problem (SLRA) is in general nonconvex. In the software package, it is solved numerically by local optimization methods. The Matlab wrapper function for low-rank approximation is defined as follows:

4 ⟨`slra` *function definition in Matlab/octave* 4⟩≡ (21b)

```
function [ph, info] = slra(p, s, r, opt)
```

The compulsory input arguments `p`, `s`, and `r` of the `slra` function correspond to the vector of parameters p, the problem structure, and the rank r, respectively, and are described in the subsequent sections. The optional parameter `opt` contains options for the optimization method and is described in Appendix B.

The output argument `ph` of the `slra` function is a locally optimal solution $\widehat{p}$ of (SLRA) and `info` is a structure containing additional information about the computed solution:

- `info.Rh` is an $(m-r)\times r$ full row rank matrix $\widehat{R}$, such that $\widehat{R}\mathscr{S}(\widehat{p})=0$ (low-rank certificate),
- `info.fmin` is the cost function value $\|p-\widehat{p}\|_w^2$ at the computed solution,
- `info.iter` is the number of iterations performed by the optimization solver,
- `info.time` is the execution time, and
- `info.Vh` covariance matrix of the optimization variables (see, $(\Theta \leftrightarrow X)$, Appendix A.1, and Section 5.1 for an example of its usage for computation of confidence ellipsoids).

Next we describe the required input parameters `p`, `s`, and `r`.

Matrix structure specification

The structure ($\mathscr{S}$) is specified by the two vectors

$$\mathbf{m} := \begin{bmatrix} m_1 & \cdots & m_q \end{bmatrix}^\top \in \mathbb{N}^q \qquad \text{and} \qquad \mathbf{n} := \begin{bmatrix} n_1 & \cdots & n_N \end{bmatrix}^\top \in \mathbb{N}^N \tag{$\mathbf{m},\mathbf{n}$}$$

and the matrix $\Phi \in \mathbb{R}^{m\times m'}$, where

$$m' := \sum_{i=1}^{q} m_i.$$

The number of columns n of the data matrix $\mathscr{S}(p)$ and the number of structure parameters n_p are computed from $(\mathbf{m},\mathbf{n})$

$$n = \sum_{i=1}^{N} n_i \qquad \text{and} \qquad n_p = Nm' + qn - qN.$$

5a ⟨*define constants* 5a⟩≡ (6b 21d)
```
q = length(s.m); ⟨default np 5b⟩, ⟨default s.n 5d⟩
N = length(s.n); n  = sum(s.n);
```

5b ⟨*default* np 5b⟩≡ (5)
```
if exist('p'),
  np = length(p);
else
  np = sum(s.m) * length(s.n) + length(s.m) * sum(s.n) ...
                              - length(s.m) * length(s.n);
end
```

5c ⟨*define* s2np 5c⟩≡
```
function np = s2np(s)
⟨default np 5b⟩
```

The parameter `s` should be a structure with a field `m` and, optionally, a field `n`, containing the vectors $(\mathbf{m},\mathbf{n})$. If `s.n` is skipped, by default $N=1$ and

$$\mathbf{n} = n = \frac{n_p - \sum_{i=1}^{q} m_i}{q} + 1.$$

This code chunk assumes already defined `np` variable.

5d ⟨*default* s.n 5d⟩≡ (5a)
```
if ~isfield(s, 'n'), s.n = np - sum(s.m) + 1; end
```

Optionally, `s` can have a field `phi`, containing the matrix $\Phi \in \mathbb{R}^{m\times m'}$. The default value for Φ is the identity matrix of size $m = m'$.

5e ⟨*default* s.phi 5e⟩≡ (21d)
```
if ~isfield(s, 'phi'), s.phi = eye(sum(s.m)); end
```

The vector p is composed of the structure parameter vectors of all scalar Hankel blocks in $(\mathscr{H}_{\mathbf{m},\mathbf{n}})$, ordered first top to bottom and then left to right, *i.e.*,

$$p = (p^{(11)}, \ldots, p^{(q1)}, \ldots, p^{(1N)}, \ldots, p^{(qN)}).$$

Weight specification, fixed and missing parameter values

The weight vector w is passed to the `slra` function by the field `w` of `s`. If w is not specified, its default value is the vector of all ones, corresponding to unweighted low-rank approximation. The parameter `s.w` can be

- n_p-dimensional nonnegative vector, in which case $w =$ `s.w`,
- $q \times N$ nonnegative matrix, in which case the weights corresponding to the structure parameters of the (i, j)th Hankel block in $(\mathscr{H}_{\mathbf{m},\mathbf{n}})$ are all equal to w_{ij}, or
- q-dimensional nonnegative vector, in which case the weights corresponding to the structure parameters of the (i, j)th Hankel block in $(\mathscr{H}_{\mathbf{m},\mathbf{n}})$ are all equal to w_i, for all j.

The second and third options of specifying `s.w` evoke a more efficient computational method but are supported only by the C++ solver.

Setting w_i to ∞ has the effect of specifying the parameter p_i as exact, *i.e.*, the approximation problem (SLRA) is solved with the additional constraint that $\widehat{p}_i$ is equal to the given parameter value p_i (and the weighted norm $\|\cdot\|_w$ is evaluated over the parameters with finite weights only). Setting w_i to 0 has the effect of ignoring the value of the parameter p_i in the solution of the approximation problem. Alternatively, ignored parameter values may be specified by the symbol `NaN` in p. From a practical point of view, ignored structure parameter values are "missing data".

`NaN` values of the parameters must correspond to zero weights and vice verse, parameters corresponding to zero weights are ignored. The following chunk of code ensures that there is conformity between `NaN`'s in p and zeros in w.

6a ⟨*convert* `NaN`*s in p to 0s and create or modify* `s.w` *accordingly* 6a⟩≡ (21b)
```
Im = find(isnan(p));
if ~isempty(Im)
  if ~isfield(s, 'w'), s.w = ones(size(p)); end
  ⟨expand q × 1 or q × N s.w to n_p × 1 vector 6b⟩
  p(Im) = 0; s.w(Im) = 0;
end
```

6b ⟨*expand* $q \times 1$ *or* $q \times N$ `s.w` *to* $n_p \times 1$ *vector* 6b⟩≡ (6a 21a)
```
⟨define constants 5a⟩
if length(s.w(:)) == q || all(size(s.w) == [q N])
  % convert q x 1 s.w to q x N
  if isvector(s.w), s.w = s.w(:); s.w = s.w(:, ones(1, N)); end
  % convert q x N s.w to np x 1
  w = [];
  for j = 1:N
    for i = 1:q
      wij = s.w(i, j) * ones(s.m(i) + s.n(j) - 1, 1); w = [w; wij];
    end
  end
  s.w = w;
end
```

Currently the C++ code does not support missing values. Missing values, however, can be *approximated* by assigning "small" weights and replacing the missing values with zeros. The closer the weights are to zeros, the better the approximation is, however, "very small" weights (say, less than 10^{-8}) causes ill-conditioning. In the extreme case of zero weights the problem, solved by the C++ code, is ill-posed.

6c ⟨*approximate zero weights with* `opt.tol_m` 6c⟩≡ (21a)
```
if isfield(s, 'w'), s.w(find(s.w == 0)) = opt.tol_m; end
```

Note 2 (Incorporating the weights w into the matrix structure $\mathscr{S}$). A weighted structured problem (SLRA) can be solved as an equivalent unweighted problem (SLRA), *i.e.*, with weights $w_i = 1$ for all i, and a modified structure specification $\mathscr{S}' : \mathbb{R}^{n_p} \to \mathbb{R}^{m \times n}$, defined by

$$\mathscr{S}'\big((p_1, \ldots, p_{n_p})\big) = \mathscr{S}\big((\sqrt{w_1^{-1}}p_1, \ldots, \sqrt{w_{n_p}^{-1}}p_{n_p})\big).$$

This fact shows that the weighted norm specification can be viewed equivalently as a modification of the *data* structure.

Note 3 (R interface). Calling the C++ solver from R is done by an R function `slra` with the same parameters as in the Matlab mex function, except that in R lists are used instead of structures, and the list elements are accessed by `<list>$<element>` instead of by `<structure>.<element>`. The computed result is returned in a list with fields `ph` and `info`, which contain, respectively, $\widehat{p}$ and the `info` variable defined above. Optional logical parameters `compute.ph` and `compute.Rh` can be used to disable the computation of $\widehat{p}$ and $\widehat{R}$. By default, $\widehat{p}$ is not computed. Examples of using the R interface can be found in the directory `test_r` of the package distribution.

4 Solution method

This section outlines the method used in the package for solving the structured low-rank approximation problem (SLRA). Let d be the rank reduction

$$d := m - r.$$

The rank constraint is represented in a kernel form

$$\operatorname{rank}\big(\mathscr{S}(\widehat{p})\big) \leq r \qquad \Longleftrightarrow \qquad \text{there is full row rank } R \in \mathbb{R}^{d \times m}, \text{ such that } R\mathscr{S}(\widehat{p}) = 0, \tag{KER}$$

i.e., the left kernel of $\mathscr{S}(\widehat{p})$ is parameterized by the d linearly independent rows of the matrix R. Then (SLRA) is equivalent to the following double minimization problem

$$\text{minimize} \quad \text{over full row rank } R \in \mathbb{R}^{d \times m} \quad f(R), \tag{SLRA$_R$}$$

where

$$f(R) := \left(\min_{\widehat{p} \in \mathbb{R}^{n_p}} \quad \|p - \widehat{p}\|_w^2 \quad \text{subject to} \quad R\mathscr{S}(\widehat{p}) = 0 \right). \tag{$f(R)$}$$

(SLRA$_R$) is refered to as the outer minimization and ($f(R)$) as the inner minimization. The inner minimization problem is a (generalized) linear least-norm problem and can be solved analytically for given R, resulting in a closed form expression of the cost function f in the outer minimization problem. The derivative of f with respect to the elements of R can also be computed analytically. Fast methods for evaluation of f and its derivatives are presented in [40] and implemented in C++. The general case is a generalized least norm problem and is solved in [24].

In data modeling—the main application area of the software—R can be interpreted as a model parameter. For example, in system identification, R is related to a difference equation (DE) representation of the system. Typically, a large amount of data is fitted by a simple model, which implies that the dimension of R is small compared to the dimension of $\widehat{p}$. Problem ($f(R)$) is bilinear in the variables R and $\widehat{p}$, which makes possible the elimination of $\widehat{p}$ for a fixed R. Many estimation problems have similar bilinear structure and their solution methods are based on elimination of the large dimensional optimization variable.

The elimination of $\widehat{p}$ in the analytic solution of ($f(R)$) is similar to the projection step in the variable projection method [10] for solving separable nonlinear least squares problems. Note, however, that the standard variable projection method was invented as a solution method for *unconstrained* optimization problems, where the cost function is explicitly given. The structured low-rank approximation problem (SLRA) in contrast is a constrained optimization problem and the parameterization (KER) is an implicit function of the optimization variables. As discussed next, (SLRA) is an optimization problem over a Grassmann manifold, which makes it rather different from the standard setup, for the variable projection method.

Note 4 (Feasibility of the minimization problem over $\widehat{p}$). In the analytic solution of ($f(R)$), it is assumed that the problem is feasible for any R in the search space. A necessary condition for this to hold is that the number of structure parameters is sufficiently large:

$$n_p \geq dn. \tag{C}$$

Condition (C) imposes restrictions on the class of problems that can be solved by the approach implemented in the package. For example, in scalar Hankel low-rank approximation problems the rank reduction d can be at most one. Interestingly, in applications of problem (SLRA) to approximate realization, system identification, and approximate common divisor computation constraint (C) can always be satisfied.

4.1 Parametrization of the search space and constraints

Problem (SLRA$_R$) is an optimization problem on a *Grassmann manifold* $\mathrm{Gr}(d,m)$ (set of d-dimensional subspaces of $\mathbb{R}^m$). Indeed, $f(R_1) = f(R_2)$ if the rows of R_1 and R_2 span the same subspace. In the package, we map the optimization on $\mathrm{Gr}(d,m)$ to an optimization problem on an Euclidean space, see [39]. The problem is turned into optimization of f over the set of full row rank matrices $R \in \mathbb{R}^{d\times m}$ that represent all (or a generic part) of the d-dimensional subspaces.

In some applications, it is necessary to impose linear constraints on R, which can also be incorporated into the parametrization of the search space.

- The experimental Matlab solver uses a general linear constraint on R

$$R = \mathscr{R}'(\theta) := \mathrm{vec}_d^{-1}(\theta\Psi), \qquad \text{where} \quad \theta \in \mathbb{R}^{n_p}, \tag{$\theta \mapsto R$}$$

  defined by a matrix $\Psi \in \mathbb{R}^{n_\theta \times dm}$. (Here $\mathrm{vec}(\cdot)$ is the column-wise vectorization operator and $\mathrm{vec}^{-1}(\cdot)$ is its inverse.) The rank constraint on R is imposed by

$$RR^\top = I_d. \tag{f.r.r. R}$$

- The C++ solver uses a matrix-product constraint on the matrix R

$$R = \mathscr{R}(\Theta) := \Theta\Psi, \qquad \text{where} \quad \Theta \in \mathbb{R}^{d\times m''}, \tag{$\Theta \mapsto R$}$$

  defined by a matrix $\Psi \in \mathbb{R}^{m''\times m}$. The rank constraint on R is imposed by

$$\Theta = \begin{bmatrix} X & -I_d \end{bmatrix}, \qquad \text{for some } X \in \mathbb{R}^{d\times(m''-d)}. \tag{$\Theta \leftrightarrow X$}$$

The Ψ matrix is passed to the `slra` function by an optimization option `opt.psi`. The default value of Ψ is I_m for the C++ version and I_{dm} for the Matlab version. The Ψ matrices—Ψ_{C} for the C++ versions and Ψ_{M} for the Matlab versions—are generally different. They coincide if and only if the rank reduction is one, *i.e.*, $d = 1$. The constraint $(\theta \mapsto R)$ is more general and includes the constraint $(\Theta \mapsto R)$ by choosing

$$\Psi_{\mathrm{M}} = \Psi_{\mathrm{C}} \otimes I_d \qquad \text{and} \qquad \theta = \mathrm{vec}(\Theta).$$

The rank constraint (f.r.r. R) parametrizes $\mathrm{Gr}(d,m)$, and turns the structured low-rank approximation problem into an optimization problem on $\mathbb{R}^{md}$ with a quadratic constraint. The rank constraint $(\Theta \leftrightarrow X)$ parametrizes a generic part of $\mathrm{Gr}(d,m)$. Its main advantage is that the structured low-rank approximation problem problem is turned into an unconstrained optimization problem on $\mathbb{R}^{(m''-d)d}$.

Note 5 (On parametrization of the whole $\mathrm{Gr}(d,m)$ in the C++ version). The whole $\mathrm{Gr}(d,m)$ can be covered by taking different Ψ (see [39]) or changing Ψ during the optimization, which is done in the optimization algorithms of [20]. Currently, in the C++ version, Ψ is a fixed parameter.

Note 6 (Solution of structured total least squares probelms [27]). Setting $\Psi = I_d$ in the C++ version is equivalent to solving (SLRA) with the constraint $\widehat{R} = \begin{bmatrix} X & -I_d \end{bmatrix}$, where $X \in \mathbb{R}^{d\times(m-d)}$ is the optimization variable. This problem is called a structured total least squares problem.

4.2 Local optimization methods

Due to the elimination of $\widehat{p}$, for $n \gg m$, the computation time of the search direction by the optimization method is negligible in comparison with the computation time for the cost function and derivative evaluations. Therefore, we use standard local optimization methods to minimize the cost function f.

The default optimization method for the C++ solver is the Levenberg-Marquardt algorithm [31, 6], implemented in the GSL library. Other options are to use the Nead-Melder method for minimization without derivatives, which is slow but robust, or the BFGS quasi-Newton method. The cost function gradient is computed analytically by

$$\mathrm{d}f(R,H) = 2y^\top \mathrm{d}s(R,H) - y^\top \mathrm{d}\Gamma(R,H)y, \qquad \text{where} \quad y := \Gamma^{-1}s(R).$$

It can be shown that the gradient evaluation has computational complexity of the same order as the one of the cost function evaluation. For details see [40].

In our experience the Levenberg-Marquardt algorithm is typically faster and more robust in practice than the alternative optimization methods. The Jacobian of $C^{-\top}s$, where C is the Cholesky factor of Γ is replaced by the pseudo-Jacobian [13], which computation is of the order of $O(m^2n)$.

The package provides an interface to the cost function and derivatives evaluation, so that any optimization method can be used (See Appendix A for details). More details about the C++ interface can be found in Appendix A. The experimental Matlab solver uses the function `fmincon` for constrained nonlinear minimization from the Optimization Toolbox of Matlab. In this case, the derivatives are approximated numerically.

All local optimization methods require an initial approximation. By default, unstructured rank-r approximation $\widehat{D}_{\text{lra}}$ of the data matrix $D = \mathscr{S}(p)$ is used, *i.e.*, the initial value for the parameter R is a full row rank matrix $R_{\text{lra}} \in \mathbb{R}^{d\times m}$, such that $R_{\text{lra}}\widehat{D}_{\text{lra}} = 0$. The default value can be overwritten by an optional argument `opt.Rini`.

In order to obtain the θ or Θ parameter from R, the equation $\mathscr{R}(\theta) \approx R_{\text{lra}}$ is solved in the least-squares sense. For the C++ solver, the obtained matrix Θ is then converted to X by

$$X := -P^{-1}Q, \qquad \text{where } \Theta =: \begin{bmatrix} Q & P \end{bmatrix}, \quad Q \in \mathbb{R}^{d\times(m''-d)}, \quad \text{and} \quad P \in \mathbb{R}^{d\times d}.$$

The result of the `slra` function—a locally optimal value of the parameter R—is returned in the variable `info.R`. The corresponding locally optimal approximation $\widehat{p}$ is computed by solving the inner minimization problem ($f(R)$). The solution is given by

$$\widehat{p}(R) = p + G(R)\big(\Gamma(R)\big)^{-1}s(R),$$

where $G(R)$ is a linear function in the elements of R, defined by

$$\operatorname{vec}\big(R\mathscr{S}(p)\big) = G(R)p.$$

5 Applications

We list below examples of low-rank approximation problems with different matrix structures $\Phi\mathscr{H}_{\mathbf{m},\mathbf{n}}$, approximation criteria $\|\cdot\|_w$, and constraints on a basis of the approximating matrix's left kernel. Each example is motivated by applications. (SYSID stands for system identification and GCD stands for greatest common divisor.)

#	example	application(s)	reference
1	unstructured	factor analysis	[11]
	uniform weights	subspace methods	[29, Ch. 7]
		latent semantic analysis	[7]
2	element-wise weights	chemometrics	[43]
3	scalar Hankel	model reduction	[2]
		autonomous SYSID	[29, Ch. 11]

		linear prediction	[19]
		shape from moments	[32, 9]
4	Hankel with structured kernel (palindromic)	spectral estimation	[38]
5	Toeplitz + Hankel	isospectral flow	[35]
6	Hankel with structured kernel (fixed poles)	SYSID with some predefined poles	[**?**]
7	$q \times 1$ block Hankel	multivariable SYSID	[30]
8	Hankel with structured kernel (fixed obsrv. indices)	SYSID with fixed obsrv. indices	[12]
9	$q \times N$ block Hankel	SYSID from multiple time series ($T_i = T$)	[30]
10	Hankel blocks next to each other	SYSID from multiple time series	[34]
11	Hankel with fixed variables	output error SYSID	[30]
12	Unstructured beneath Hankel block	deconvolution	[21]
13	Hankel with first ℓ elements fixed	SYSID with fixed initial conditions	[37]
14	Sylvester	approximate GCD of two polynomials	[16]
15	generalized Sylvester	approximate GCD of N polynomials	[15]
16	unstructured with missing elements	recommender systems	[36]
17	Hankel with missing elements	SYSID with missing data	[34]
18	missing and fixed elements	matrix completion	[4]

In the rest of the section, we illustrate the performance of the structured low-rank approximation package on a few examples from the list. Section 5.1 shows a deconvolution problem (example 12) in the errors-in-variables setting. Section 5.2 demonstrates the efficiency of the package on benchmark system identification problems (example 7) from the database DAISY [33]. Section 5.3 shows numerical examples of computing approximate common divisors of two polynomials (example 14), using Sylvester structured low-rank approximation.

5.1 Errors-in-variables deconvolution

The convolution $h \star u$ of the sequences

$$u = \big(u(1), \ldots, u(T)\big) \qquad \text{and} \qquad h = \big(h(0), h(1), \ldots, h(n-1)\big)$$

is a sequence y, defined by the convolution sum

$$y(t) := \sum_{\tau=0}^{n} h(\tau) u(t-\tau), \qquad \text{for} \quad t = n, n+1, \ldots, T. \tag{CONV}$$

With some abuse of notation, we denote by u, h, and y both the sequences and the corresponding vectors:

$$u := \begin{bmatrix} u(1) \\ \vdots \\ u(T) \end{bmatrix} \in \mathbb{R}^T, \qquad h := \begin{bmatrix} h(0) \\ \vdots \\ h(n-1) \end{bmatrix} \in \mathbb{R}^n, \qquad y := \begin{bmatrix} y(n) \\ \vdots \\ y(T) \end{bmatrix} \in \mathbb{R}^{T-n+1}.$$

Using the Toeplitz matrix

$$\mathscr{T}(u) = \begin{bmatrix} u(n) & u(n-1) & \cdots & u(2) & u(1) \\ u(n+1) & u(n) & \ddots & \ddots & u(2) \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ u(T) & u(T-1) & \cdots & u(T-n+2) & u(T-n+1) \end{bmatrix} \in \mathbb{R}^{(T-n+1)\times n},$$

the convolution sum (CONV) can be written as a matrix-vector product

$$y = \mathscr{T}(u)h. \tag{CONV'}$$

Consider the errors-in-variables data generating model [18]:

$$u = \bar{u} + \widetilde{u} \qquad \text{and} \qquad y = \bar{y} + \widetilde{y}, \tag{EIV}$$

where (u, y) is the measured data, $(\bar{u}, \bar{y})$ is its "true value", and $(\widetilde{u}, \widetilde{y})$ is the measurement noise. The true data satisfy the relation $\bar{y} = \mathscr{T}(\bar{u})\bar{h}$, for some "true parameter" vector $\bar{h}$. The goal in the errors-in-variables problem is to estimate consistently and efficiently $\bar{h}$ from the noisy data.

The true parameter vector, however, is not identifiable (a solution is not unique) unless there is prior knowledge about the measurement noise. We assume that the measurement noise elements $\widetilde{u}(t)$ and $\widetilde{y}(t)$ are zero mean, independent, and identically distributed, but the noise variance is unknown. It is proven in [18] that in this case the structured total least squares approximate solution of the overdetermined system of linear equations (CONV') yields a consistent estimator. If in addition, the noise distribution is normal, it is a maximum likelihood estimator and is asymptotically normal.

The structured total least squares problem is equivalent to low-rank approximation of the matrix $\begin{bmatrix} \mathscr{T}(u) & y \end{bmatrix}^\top$ (see Note 6). The structure is mosaic-Hankel-like

$$\mathscr{S}\left(\begin{bmatrix} u \\ y \end{bmatrix}\right) = \begin{bmatrix} \mathscr{T}^\top(u) \\ y^\top \end{bmatrix} = \underbrace{\begin{bmatrix} \begin{bmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{bmatrix} & \\ & 1 \end{bmatrix}}_{\Phi} \begin{bmatrix} \mathscr{H}_{n,T-n+1}(u) \\ \mathscr{H}_{1,T-n+1}(y) \end{bmatrix} = \Phi \mathscr{H}_{[n\ 1],T-n+1}\left(\begin{bmatrix} u \\ y \end{bmatrix}\right).$$

Therefore, the maximum likelihood estimator for $\bar{h}$ can be computed with the `slra` function

11a ⟨*call* `slra` 11a⟩≡ (12)

```
[uyh, info] = slra([u; y], s, n);
```

using the structure specification

11b ⟨*structure specification for deconvolution* 11b⟩≡ (12)

```
s.m = [n 1]; s.phi = blkdiag(fliplr(eye(n)), 1);
```

The estimate $\widehat{h}$ is obtained from the parameter vector $\widehat{R}$ (see (KER)) by normalization:

$$\widehat{R}' := -\widehat{R}/\widehat{R}_{n+1}, \qquad \widehat{h}(\tau) = \widehat{R}'_{\tau+1}, \quad \text{for } \tau = 0, 1, \ldots, n-1.$$

Indeed,

$$\widehat{R}' \begin{bmatrix} \mathscr{T}^\top(\widehat{u}) \\ \widehat{y}^\top \end{bmatrix} = 0 \qquad \Longleftrightarrow \qquad \mathscr{T}(\widehat{u})\widehat{h} = \widehat{y}.$$

11c ⟨$\widehat{R} \mapsto \widehat{h}$ 11c⟩≡ (12)

```
hh = - info.Rh(1:n)' / info.Rh(n + 1);
```

Note 7 (Identification of a finite impulse response system). In system theory and signal processing, (CONV) defines a finite impulse response linear time-invariant dynamical system. The sequence h is a parameter, u is the input, and y is the output of the system. The deconvolution problem is therefore a system identification problem: estimate the true data generating system from noisy data.

Numerical example

We illustrate empirically the consistency of the structured total least squares estimator. A true parameter vector $\bar{h}$ and a true input sequence $\bar{u}$ are randomly generated. The corresponding true output $\bar{y}$ is computed by convolution of $\bar{h}$ and $\bar{u}$. Zero mean independent and normally distributed noise is added to the true data according to the errors-in-variables model (EIV) and the `slra` function is evoked for the computation of the estimate.

The experiment is repeated $K = 500$ times with independent noise realizations (but fixed true values). Let $\widehat{h}^{(i)}$ be the total least squares solution obtained in the ith repetition. Figure 1, left, shows the root-mean-square error

$$e := \sqrt{\frac{1}{K}\sum_{i=1}^{K} \|h - \widehat{h}^{(i)}\|_2^2}$$

as a function of the sample size T. Theoretically, the maximum likelihood estimation error converges to zero at a rate that is proportional to the inverse square root of the sample size ($1/\sqrt{T}$ convergence). The simulation results confirm the theoretical convergence rate.

Figure 1, right, shows the true parameter value $\bar{h}$ (red cross), the 500 estimates $\widehat{h}^{(i)}$ (blue dots), and the 95% confidence ellipsoid, computed from the covariance matrix `info.Vh`, corresponding to $\widehat{h}^{(500)}$ and translated to $\bar{h}$. (Note that we do not plot the confidence ellipsoid around each estimate of the parameter in order to simplify the picture.) The fact that about 475 estimates have the true value of the parameter in the confidence region is an empirical confirmation that the confidence regions are correct.

12 ⟨*confidence bounds example* 12⟩≡

```
clear all; T = 2000; n = 2; sigma = 0.1;
N = 20; TT = round(linspace(100, T, N)); K = 500;
⟨structure specification for deconvolution 11b⟩
U0 = rand(T, 1); h0 = rand(n, 1); nh0 = norm(h0);

for i = 1:N
  t = TT(i); u0 = U0(1:t); y0 = conv(u0, h0, 'valid');
  for j = 1:K
    ut = randn(size(u0)); u = u0 + sigma * ut / std(ut);
    yt = randn(size(y0)); y = y0 + sigma * yt / std(yt);
    ⟨call slra 11a⟩, ⟨R̂ ↦ ĥ 11c⟩
    E(j) = norm(h0 - hh); Hh(:, j) = hh;
  end
  Em(:, i) = rms(E);
end

Emh = 1 ./ sqrt(TT); al = Em(1, :) / Emh;
figure(1), plot(TT, Em(1, :), '-k', TT, al * Emh, ':b');
ax = axis; axis([TT(1) TT(end) ax(3:4)]),
legend('empirical      .', 'theoretical'), print_fig('slra-err-bounds-f1')

c = chi2inv(0.95, n); vh = info.fmin / t; Vh = vh * info.Vh;
figure(2), ellipsoid(sqrtm(c * Vh), h0, 'g'), hold on
plot(h0(1), h0(2), 'rX', 'markersize', 10)
for i = 1:500, plot(Hh(1, i), Hh(2, i), 'b.', 'markersize', 10), end
print_fig('slra-err-bounds-f2')
```

5.2 Performance comparison with the old version of the package

In [25] and [28], the performance of the old version of the package is tested on benchmark system identification problems from the database DAISY [33]. The problems involve vector time-series and the model class consists of multiple-inputs multiple-outputs linear time-invariant systems. The identification problem in this case is equivalent to block-Hankel low-rank approximation, which is also a special case of the mosaic Hankel low-rank approximation problem (SLRA). For a description of the test examples, we refer the reader to [25].

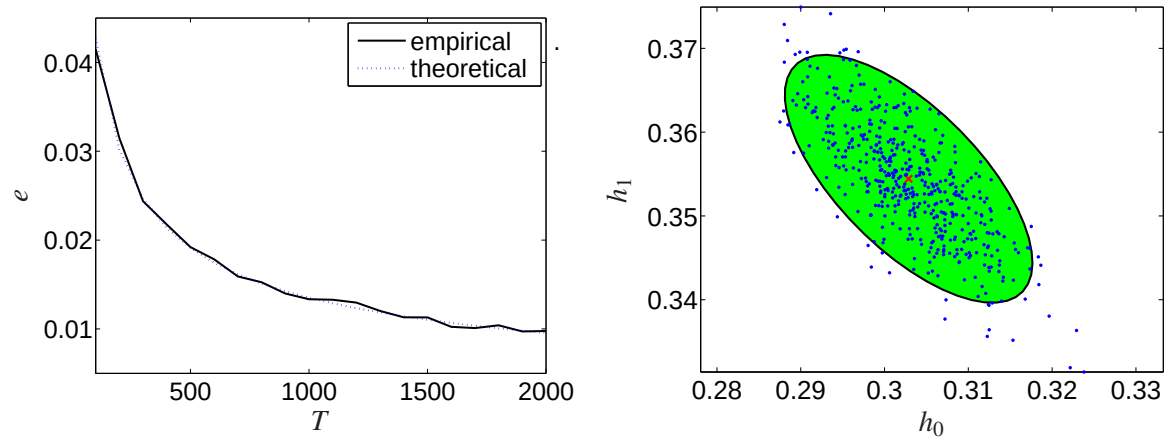

Figure 1: Left: empirical (solid line) and theoretical (dotted line) root-mean-square estimation error e as a function of the sample size T; Right: 95% confidence ellipsoid.

In Table 2, the execution time of the new and the old version of the package are compared for the same setting of the experiment as in [25]. In all examples, the approximate solutions computed by the two versions of the software are the same. The results show that the efficiency of the new version of the package is improved by an average of 40%. Note that the speedup is achieved by software improvements despite of the fact that the new version treats a more general problem.

#	Data set name	T	m	p	ℓ	t_{new} (sec)	t_{old} (sec)	Δt (%)
1	Data of a simulation of the western basin of Lake Erie	57	5	2	1	0.01	0.01	0
2	Data of ethane-ethylene distillation column	90	5	3	1	0.02	0.03	33
3	Heating system	801	1	1	2	0.01	0.02	50
4	Data from an industrial dryer (Cambridge Control Ltd)	867	3	3	1	0.16	0.25	36
5	Data of a laboratory setup acting like a hair dryer	1000	1	1	5	0.02	0.04	50
6	Data of the ball-and-beam setup in SISTA	1000	1	1	2	0.01	0.02	50
7	Wing flutter data	1024	1	1	5	0.02	0.04	50
8	Data from a flexible robot arm	1024	1	1	4	0.01	0.01	0
9	Data of a glass furnace (Philips)	1247	3	6	1	2.88	4.41	35
10	Heat flow density through a two layer wall	1680	2	1	2	0.09	0.22	59
11	Simulation data of a pH neutralization process	2001	2	1	6	0.03	0.06	50
12	Data of a CD-player arm	2048	2	2	1	0.09	0.23	61
13	Data from a test setup of an industrial winding process	2500	5	2	2	0.64	0.92	30
14	Liquid-saturated steam heat exchanger	4000	1	1	2	0.03	0.07	57
15	Data from an industrial evaporator	6305	3	3	1	1.36	2.22	39
16	Continuous stirred tank reactor	7500	1	2	1	0.24	0.75	68
17	Model of a steam generator at Abbott Power Plant	9600	4	4	1	13.10	15.77	17

Table 2: Performance test on examples from DAISY: T—number of data points, m—number of inputs, p—number of outputs, ℓ—lag of the identified model, t_{new} and t_{old} execution times in seconds for the new and old versions of the package, respectively, Δt—percentage speedup, achieved by the new version of the package.

5.3 Approximate greatest common divisor of two polynomials

Consider the polynomials p^1 and p^2 of degrees d_1 and d_2, respectively, and a positive integer $\ell < \min(d_1, d_2)$. The Sylvester matrix of p^1 and p^2 with parameter ℓ is a

$$(d_1 + d_2 - 2\ell + 2) \times (d_1 + d_2 - \ell + 1)$$

mosaic Hankel matrix with upper-left and lower-right triangles of the block-elements fixed to zero:

$$S_\ell(p^1,p^2) := \begin{bmatrix} S_{d_2-\ell}(p^1) \\ S_{d_1-\ell}(p^2) \end{bmatrix}, \qquad \text{where} \quad S_d(p) := \begin{bmatrix} & & p_0 & p_1 & \cdots & p_{d_p} \\ & \iddots & \iddots & & \iddots & \\ p_0 & p_1 & \cdots & p_{d_p} & & \end{bmatrix} \in \mathbb{R}^{(d+1)\times(d+d_p+1)}. \tag{SYLV}$$

The degree of the greatest common divisor of p^1 and p^2 is equal to

$$\text{degree}\left(\gcd(p^1,p^2)\right) = d_1 + d_2 - \text{rank}\left(S_0(p^1,p^2)\right). \tag{$*$}$$

The considered approximate common divisor problem is defined as follows: given polynomials p^1 and p^2 and a lower bound $\ell > 0$ on the degree of the common divisor, modify p^1 and p^2, as little as possible, so that the modified polynomials $\widehat{p}^1$ and $\widehat{p}^2$ have a greatest common divisor of degree at least ℓ, *i.e.*,

$$\begin{aligned} &\text{minimize} \quad \text{over } \widehat{p}^1 \in \mathbb{R}^{d_1+1} \text{ and } \widehat{p}^2 \in \mathbb{R}^{d_2+1} \quad \left\| \begin{bmatrix} p^1 \\ p^2 \end{bmatrix} - \begin{bmatrix} \widehat{p}^1 \\ \widehat{p}^2 \end{bmatrix} \right\|_2^2 \\ &\text{subject to} \quad \text{degree}\left(\gcd(\widehat{p}^1,\widehat{p}^2)\right) \geq \ell. \end{aligned} \tag{ACD}$$

The approximate common divisor for the polynomials p^1 and p^2 is the exact greatest common divisor of $\widehat{p}^1$ and $\widehat{p}^2$.

It can be shown [26] that the approximate common divisor problem (ACD) is equivalent to a Sylvester low-rank approximation problem

$$\begin{aligned} &\text{minimize} \quad \text{over } \widehat{p}^1 \in \mathbb{R}^{d_1+1} \text{ and } \widehat{p}^2 \in \mathbb{R}^{d_2+1} \quad \left\| \begin{bmatrix} p^1 \\ p^2 \end{bmatrix} - \begin{bmatrix} \widehat{p}^1 \\ \widehat{p}^2 \end{bmatrix} \right\|_2^2 \\ &\text{subject to} \quad \text{rank}\left(S_\ell(p^1,p^2)\right) \leq d_1 + d_2 - 2\ell + 1. \end{aligned}$$

14a ⟨`slra` *arguments for approximate GCD of 2 polynomials* 14a⟩≡ (14b 15a)

```
s.m = [d2 - ell + 1; d1 - ell + 1];
s.n = d1 + d2 - ell + 1;
z1 = zeros(d2 - ell, 1); z2 = zeros(d1 - ell, 1);
p = [z1; p1(:); z1; z2; p2(:); z2];
s.w = 1 ./ p; s.w(~isinf(s.w)) = 1;
r = d1 + d2 - 2 * ell + 1;
```

Numerical examples

Our first example is Example 4.1 from [44]. The given polynomials are

$$\begin{aligned} p^1(z) &= (4+2z+z^2)(5+2z) + 0.05 + 0.03z + 0.04z^2 \\ p^2(z) &= (4+2z+z^2)(5+\ z) + 0.04 + 0.02z + 0.01z^2 \end{aligned}$$

and an approximate common divisor of degree $\ell = 2$ is sought.

14b ⟨*example GCD* 14b⟩≡ 14c▷

```
clear all; d1 = 3; d2 = 3; ell = 2;
p1 = conv([4 2 1], [5 2]) + [0.05 0.03 0.04 0];
p2 = conv([4 2 1], [5 1]) + [0.04 0.02 0.01 0];
⟨slra arguments for approximate GCD of 2 polynomials 14a⟩
[ph, info] = slra(p, s, r);
```

The solution computed by the `slra` function

14c ⟨*example GCD* 14b⟩+≡ ◁14b 15a▷

```
ph1 = ph(2:5), ph2 = ph(8:11), r_ph1 = roots(ph1), r_ph2 = roots(ph2)
```

```
ph1 =                ph2 =                r_ph1 =                r_ph2 =
   20.0500              20.0392              -0.2510 + 0.4336i      -0.2510 + 0.4336i
   18.0332              14.0179              -0.2510 - 0.4336i      -0.2510 - 0.4336i
    9.0337               7.0176              -0.3973                -0.1974
    2.0000               0.9933
```

coincides (up to errors due to the numerical precision) with the one reported in [44].

The second example is Example 4.2, case 1, from [44] (originally given in [17]). The given polynomials are

$$p^1(\xi) = (1-\xi)(5-\xi) = 5 - 6\xi + \xi^2$$
$$p^2(\xi) = (1.1-\xi)(5.2-\xi) = 5.72 - 6.3\xi + \xi^2$$

and an approximate common divisor of degree $\ell = 1$ (a common root) is sought.

15a ⟨*example GCD* 14b⟩+≡ ◁14c 15b▷

```
d1 = 2; d2 = 2; ell = 1;
p1 = conv([1   -1], [5   -1]);
p2 = conv([1.1 -1], [5.2 -1]);
⟨slra arguments for approximate GCD of 2 polynomials 14a⟩
[ph, info] = slra(p, s, r);
```

Again, the solution computed by the `slra` function

15b ⟨*example GCD* 14b⟩+≡ ◁15a

```
ph1 = ph(2:4), ph2 = ph(7:9), r_ph1 = roots(ph1), r_ph2 = roots(ph2)
```

```
ph1 =                ph2 =                r_ph1 =                r_ph2 =
    4.9994               5.7206               1.0046                 0.9047
   -6.0029              -6.2971               0.1961                 0.1961
    0.9850               1.0150
```

coincides (up to numerical computation errors) with the one reported in the literature.

6 Conclusions

The developed software package is a generic tool for data modeling and has numerous applications in system theory and identification, signal processing, machine learning, chemometrics, and computer algebra. Its functionality generalized the one of the software of [27] by allowing specification of element-wise weights, arbitrary fixed and missing elements, linear constraints on the parameter matrix, and by generalizing the structure of the approximating matrix to the class of mosaic Hankel-like matrices. Planned extensions of the package are

- multiple rank constraints,
- mosaic Hankel matrices with repeated Hankel blocks (*i.e.*, structure parameters common to two or more blocks),
- linear equality constraints on the structure parameter vector,
- nonlinear structure of the kernel parameter R.

On the practical side, we plan to experiment with different optimization strategies, *e.g.*, methods for optimization on a Grassmann manifold [1, 5], and compare the performance of the package with state-of-the-art problem dependent methods for different applications. Preliminary results on using optimization methods on a Grassmann manifold are reported in [39].

Acknowledgements

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement number 258581 "Structured low-rank approximation: Theory, algorithms, and applications".

References

[1] P.-A. Absil, R. Mahony, and R. Sepulchre. *Optimization Algorithms on Matrix Manifolds*. Princeton University Press, Princeton, NJ, 2008.

[2] A. Antoulas. *Approximation of Large-Scale Dynamical Systems*. SIAM, 2005.

[3] J.-F. Cai, E. Candés, and Z. Shen. A singular value thresholding algorithm for matrix completion. *SIAM Journal on Optimization*, 20(4):1956–1982, January 2010.

[4] E. Candés and B. Recht. Exact matrix completion via convex optimization. *Found. of Comput. Math.*, 9:717–772, 2009.

[5] Y. Chikuse. *Statistics on Special Manifolds*. Lecture Notes in Statistics, 174. Springer, 2003.

[6] J. Dennis and R. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Society for Industrial Mathematics, 1987.

[7] S. Dumais. Latent semantic analysis. *Annual Review of Information Science and Technology*, 38:189–230, 2004.

[8] M. Galassi et al. GNU scientific library reference manual. `http://www.gnu.org/software/gsl/`.

[9] G. Golub, P. Milanfar, and J. Varah. A stable numerical method for inverting shape from moments. *SIAM J. Sci. Comput*, 21:1222–1243, 1999.

[10] G. Golub and V. Pereyra. Separable nonlinear least squares: the variable projection method and its applications. *Institute of Physics, Inverse Problems*, 19:1–26, 2003.

[11] R. Gorsuch. *Factor Analysis*. Hillsdale, 1983.

[12] R. Guidorzi. Invariants and canonical forms for systems structural and parametric identification. *Automatica*, 17(1):117–133, 1981.

[13] P. Guillaume and R. Pintelon. A Gauss–Newton-like optimization algorithm for "weighted" nonlinear least-squares problems. *IEEE Trans. Signal Process.*, 44(9):2222–2228, 1996.

[14] G. Heinig. Generalized inverses of Hankel and Toeplitz mosaic matrices. *Linear Algebra Appl.*, 216(0):43–59, February 1995.

[15] E. Kaltofen, Z. Yang, and L. Zhi. Approximate greatest common divisors of several polynomials with linearly constrained coefficients and singular polynomials. In *Proc. Int. Symp. Symbolic Algebraic Computation*, pages 169–176, 2006.

[16] N. Karmarkar and Y. Lakshman. On approximate GCDs of univariate polynomials. In S. Watt and H. Stetter, editors, *J. Symbolic Comput.*, volume 26, pages 653–666, 1998.

[17] N. Karmarkar and Y. Lakshman. On approximate GCDs of univariate polynomials. In S. Watt and H. Stetter, editors, *J. Symbolic Comput.*, volume 26, pages 653–666, 1998. Special issue on Symbolic Numeric Algebra for Polynomials.

[18] A. Kukush, I. Markovsky, and S. Van Huffel. Consistency of the structured total least squares estimator in a multivariate errors-in-variables model. *J. Statist. Plann. Inference*, 133(2):315–358, 2005.

[19] R. Kumaresan and D. Tufts. Estimating the parameters of exponentially damped sinusoids and pole-zero modeling in noise. *IEEE Trans. Acust., Speech, Signal Proc.*, 30(6):833–840, 1982.

[20] J. Manton, R. Mahony, and Y. Hua. The geometry of weighted low-rank approximations. *IEEE Trans. Signal Process.*, 51(2):500–514, 2003.

[21] I. Markovsky. Structured low-rank approximation and its applications. *Automatica*, 44(4):891–909, 2008.

[22] I. Markovsky. *Low Rank Approximation: Algorithms, Implementation, Applications*. Springer, 2012.

[23] I. Markovsky. A software package for system identification in the behavioral setting. *Control Engineering Practice*, 21:1422–1436, 2013.

[24] I. Markovsky and K. Usevich. Structured low-rank approximation with missing data. *SIAM J. Matrix Anal. Appl.*, pages 814–830, 2013.

[25] I. Markovsky and S. Van Huffel. High-performance numerical algorithms and software for structured total least squares. *J. Comput. Appl. Math.*, 180(2):311–331, 2005.

[26] I. Markovsky and S. Van Huffel. An algorithm for approximate common divisor computation. In *Proc. 17th Symp. on Math. Theory of Networks and Systems*, pages 274–279, Kyoto, Japan, 2006.

[27] I. Markovsky, S. Van Huffel, and R. Pintelon. Block-Toeplitz/Hankel structured total least squares. *SIAM J. Matrix Anal. Appl.*, 26(4):1083–1099, 2005.

[28] I. Markovsky, J. C. Willems, and B. De Moor. Comparison of identification algorithms on the database for system identification DAISY. In *Proc. 17th Symp. on Math. Theory of Networks and Systems*, pages 2858–2869, Kyoto, Japan, 2006.

[29] I. Markovsky, J. C. Willems, S. Van Huffel, and B. De Moor. *Exact and approximate modeling of linear systems: A Behavioral Approach*. SIAM, March 2006.

[30] I. Markovsky, J. C. Willems, S. Van Huffel, B. De Moor, and R. Pintelon. Application of structured total least squares for system identification and model reduction. *IEEE Trans. Automat. Control*, 50(10):1490–1500, 2005.

[31] D. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *SIAM J. Appl. Math.*, 11:431–441, 1963.

[32] P. Milanfar, G. Verghese, W. Karl, and A. Willsky. Reconstructing polygons from moments with connections to array processing. *IEEE Trans. Signal Proc.*, 43:432–443, 1995.

[33] B. De Moor, P. De Gersem, B. De Schutter, and W. Favoreel. DAISY: A database for identification of systems. *Journal A*, 38(3):4–5, 1997. Available from `http://homes.esat.kuleuven.be/~smc/daisy/`.

[34] R. Pintelon and J. Schoukens. Identification of continuous-time systems with missing data. *IEEE Trans. Instr. Meas.*, 48(3):736–740, 1999.

[35] O. Rojo. A new algebra of Toeplitz-plus-Hankel matrices and applications. *Computers & Mathematics with Appl.*, 55(12):2856–2869, 2008.

[36] T. Segaran. *Programming Collective Intelligence: Building Smart Web 2.0 Applications*. O'Reilly Media, 2007.

[37] T. Söderström and P. Stoica. *System Identification*. Prentice Hall, 1989.

[38] P. Stoica and R. Moses. *Spectral Analysis of Signals*. Prentice Hall, 2005.

[39] K. Usevich and I. Markovsky. Structured low-rank approximation as a rational function minimization. In *Proc. of the 16th IFAC Symposium on System Identification*, pages 722–727, Brussels, 2012.

[40] K. Usevich and I. Markovsky. Variable projection for affinely structured low-rank approximation in weighted 2-norms. *J. Comput. Appl. Math.*, 2013.

[41] D. van Heesch. Doxygen. `www.doxygen.org`.

[42] S. Van Huffel, V. Sima, A. Varga, S. Hammarling, and F. Delebecque. High-performance numerical software for control. *IEEE Control Systems Magazine*, 24:60–76, 2004.

[43] P. Wentzell, D. Andrews, D. Hamilton, K. Faber, and B. Kowalski. Maximum likelihood principal component analysis. *J. Chemometrics*, 11:339–366, 1997.

[44] L. Zhi and Z. Yang. Computing approximate GCD of univariate polynomials by structure total least norm. In *MM Research Preprints*, number 24, pages 375–387. Academia Sinica, 2004.

A Implementation details

The structured low-rank approximation solver is written in C++ language and the implementation uses object-oriented style. The software implementation in the C++ version is synchronized with the description of the algorithms in [40]. The LAPACK and BLAS libraries are used for the matrix computations and the GNU scientific library (GSL) [8] is used for vector-matrix multiplications and for nonlinear optimization. The key computational step—Cholesky factorization of the Γ matrix—is done by the LAPACK's function `DPBTRF`, which exploits the banded structure of the matrix. In case of block-wise weights (see Section 3), the package can optionally use the `MB02GD` function from the SLICOT library [42], which exploits both the banded and Toeplitz structure of Γ. (In our test examples `MB02GD` gives no advantage in terms of computation time.)

The package is divided into the following sub directories and files:

- `cpp`: C++ core classes and functions.[2]
- `mex`: source files for Matlab/Octave interface.
- `Rslra`: source files for R interface.
- `test_m`: demo files for Matlab/Octave interface.

[2] Type declarations and function prototypes are in `slra_xxx.h` files; the implementation is in `slra_xxx.cpp` files.

- `test_r`: demo files for R interface.
- `test_c`: demo files for C++ interface.
- `doc`: documentation and examples.
- `ident.m`: wrapper function for system identification [**?**].

A.1 Main function

The solver is called in C via the function `slra`, defined as follow:

18a ⟨`slra` *function definition in C* 18a⟩≡
```
int slra( CostFun *costFun, OptimizationOptions *opt,
          gsl_matrix *Rini, gsl_matrix *Psi,
          gsl_vector *p_out, gsl_matrix *r_out, gsl_matrix *v_out );
```

- `costFun`: object of type `CostFunction` containing all information about $f(R)$ (structure $\mathscr{S}$, input vector p and rank r).
- `opt`: optimization options and output information.
- `Rini`: matrix of initial approximation.
- `Psi`: $\Psi^\top$ matrix (see $(\Theta \mapsto R)$).
- `p_out`: approximation $\widehat{p}$.
- `R_out`: low-rank certificate $\widehat{R}^\top$.
- `v_out`: error covariance $(J^\top J)^{-1}$ of $\mathrm{vec}(X)$.

Note 8 (C row-major convention). In the C++ solver all matrices are transposed, due to the row-major order convention of C/GSL.

A.2 Structure specification and object-oriented paradigm

The `CostFunction` class represents the cost function $f(R)$ defined on a Grassmann manifold. It is constructed with the help of the C++ constructor

18b ⟨`CostFunction` *constructor* 18b⟩≡
```
CostFunction::CostFunction( const gsl_vector *p, Structure *s,
                            size_t d, gsl_matrix *Phi );
```

- `p_in`: input vector p.
- `s`: object of type `Structure`, see Section A.2.
- `d`: rank reduction.
- `Phi`: $\Phi^\top$ matrix (see $(\mathscr{S})$).

The `Structure` class represents the structure specification $\mathscr{S}$ and the weights vector w.

18c ⟨*definition of class* `Structure` 18c⟩≡
```
class Structure {
public:
   virtual ~Structure() {}
   virtual int getNp() const = 0;
   virtual int getM() const = 0;
   virtual int getN() const = 0;
   virtual void fillMatrixFromP( gsl_matrix* c, const gsl_vector* p )  = 0;
   virtual void correctP( gsl_vector* p, gsl_matrix *R, gsl_vector *yr,
                          long wdeg = 0 ) = 0;
   virtual Cholesky *createCholesky( int D ) const = 0;
   virtual DGamma *createDGamma( int D ) const = 0; };
```

The function `slra` can deal with an arbitrary `Structure` object using the following methods:

- `getNp()`, `getM()` and `getN()` return n_p, m and n, respectively.
- `fillMatrixFromP()` constructs the matrix $\mathscr{S}(p)^\top$ from the vector of structure parameters p.
- `correctP()` constructs the correction vector Δp from R and precomputed $y_r := \Gamma^{-1}(R)s(R)$.

- `createCholesky()` and `createDGamma()` create objects for Cholesky factorization and gradient/Jacobian computation respectively.

The currently implemented structures are:

- `LayeredHStructure`: class for layered Hankel structure with block-wise weights,
- `WLayeredHStructure`: class for layered Hankel structure with element-wise weights, and
- `MosaicHStructure` and `WMosaicHStructure`: classes for mosaic Hankel structure, that are implemented based on layered Hankel structure.

The object-oriented paradigm facilitates the memory management and software design. In particular, it allows the user to add a new problem specification (for example, a new type of structure) by implementing a new `Structure` subclass. Further details can be found in the manual for the C++ interface, which can be generated by running `doxygen` [41] in the `cpp` directory.

The object-oriented paradigm is also used in the interaction of the Matlab wrapper function and the C++ solver. This is performed through the mex-function `slra_mex_obj`, which constructs an object for operations with the cost function. This object is used for cost function and derivatives evaluation. Thus any optimization method implemented in matlab can be used for structured low-rank approximation.

A.3 Installation instructions

The package is distributed in the form of source code and precompiled mex files for different platforms. The mex files for Matlab/Octave and the R interface can be compiled by calling `make` in the root directory. In this case, the GSL, BLAS, and LAPACK libraries have to be installed in advance. The SLICOT library can be used optionally.

B Optional input arguments for the optimization method

The following optional input arguments of the `slra` function are related to the optimization solver, used for the solution of the parameter optimization problem. The options are specified as fields of the input argument `opt`.

- `solver` specifies the solver. The options are:
  - `c` — efficient C++ solver (default), and
  - `m` — general but inefficient solver, implemented in Matlab.
- `method` specifies the optimization method to be used with the C++ solver. The available options are all local optimization methods in the GSL library (see the GSL manual [8] for more details):
  - `l` — Levenberg–Marquardt methods for nonlinear least squares (default),
    - `ll` — method `lmder` (default),
    - `ls` — method `lmsder`,
  - `q` — Quasi-Newton and conjugate gradient methods for nonlinear minimization with derivatives,
    - `qb` — method `bfgs` (default),
    - `q2` — method `bfgs2`,
    - `qp` — method `conjugate_pr`,
    - `qf` — method `conjugate_fr`,
  - `n` — methods for nonlinear minimization without derivatives,
    - `nn` — method `nmsimplex` (default),
    - `n2` — method `nmsimplex2`, and
    - `nr` — method `nmsimplex2rand`.

- `disp` specifies the level of displayed information. The options are:
  - `iter` — print progress information per iteration,
  - `notify` — in case of lack of convergence only (default), or
  - `off` — no display.

20a ⟨*default options* 20a⟩≡ (21b)
```
if ~exist('opt'), opt = struct; end
if ~isfield(opt, 'solver'), opt.solver = 'c'; end
if ~isfield(opt, 'disp'), opt.disp = 'off'; end
if ~isfield(opt, 'tol_m'), opt.tol_m = 1e-6; end
```

- The following arguments control the termination of the optimization:
  - `maxiter` — maximum number of iterations,
  - `tol` — tolerance for the change of the cost function value,
  - `epsrel` and `epsabs` — relative and absolute tolerance for the element-wise change of the optimization variables,
  - `epsgrad` — tolerance for the norm of the gradient.

Inline help

20b ⟨`slra` *inline help* 20b⟩≡ (21b)
```
% SLRA - solves the structured low-rank approximation problem
%
% minimize over ph norm(w .* (p - ph)) ^ 2 subject to rank(S(ph)) <= r
% where S(ph) = Phi * H, with H a q x N block matrix with Hankel blocks
% H_ij(ph) = hankel(ph_ij(1:m_i, m_i:(m_i + n_j - 1)))
% and ph = [ph_11; ... ph_q1; ... ph_1N; ... ph_qN].
%
% [ph, info] = slra(p, s, r, opt)
%
% Input arguments:
% p - structure parameter vector
% s - problem structure specification:
% s.m = [m_1 ... m_q], s.n = [n_1 ... n_N],
% s.phi = Phi (default identity matrix)
% s.w = w (default ones(np, 1))
% (w(i) == inf <=> ph(i) == p(i), w(i) == 0 <=> p(i) not used)
% r - rank (default is rank reduction by 1)
% opt.psi - kernel basis constraint (default identity matrix)
% opt.Rini - initial approximation (default unstructured LRA)
% Rini is a basis for an approximate left kernel of S(p)
% opt.disp - information about progress of the optimization
% opt.solver - solver: 'c' -- efficient, 'm' -- general (default 'c')
% opt.method - optimization method
%
% Output arguments:
% ph - approximation structure parameter vector
% info.Rh - low-rank certificate: Rh * S(ph) = 0
% info.iter - number of iterations
% info.time - execution time
% info.fmin = norm(w .* (p - ph)) ^ 2
%
% Note: it is required that length(p) > n * (m - r).
```

`slra` wrapper function

21a ⟨*call the solver* 21a⟩≡ (21b)

```
if opt.solver == 'c'
  ⟨approximate zero weights with opt.tol_m 6c⟩
  opt = rmfield(opt, 'solver');
  obj = slra_mex_obj('new', p, s, r);
  [ph, info] = slra_mex_obj('optimize', obj, opt);
  slra_mex_obj('delete', obj);
elseif opt.solver == 'r'
  if isfield(s, 'w')
    ⟨expand q × 1 or q × N s.w to n_p × 1 vector 6b⟩
    opt.w = s.w;
  end
  if isfield(opt, 'Rini'), opt.P_init = null(opt.Rini); end
  np = length(p); s.tts = s2s(s, np);
  [ph, info] = reg_slra(p, s, r, opt);
  info.Rh = null(info.P')';
else
  if ~isfield(s, 'w'), s.w = []; end
  ⟨expand q × 1 or q × N s.w to n_p × 1 vector 6b⟩
  if ~all(size(s.w) == [length(p(:)) length(p(:))]), s.w = s.w(:); end
  if ~isfield(s,   'phi' ), s.phi = []; end
  if ~isfield(opt, 'Rini'), opt.Rini = []; end
  if ~isfield(opt, 'psi' ), opt.psi = []; end
  np = length(p); warning_state = warning; warning('off');
  [ph, info] = slra_ext(s2s(s, np), p, r, s.w, opt.Rini, s.phi, opt.psi, opt);
  warning('warning_state');
end
```

The complete function for structured low-rank approximation is:

21b ⟨`slra` 21b⟩≡

```
⟨slra inline help 20b⟩
⟨slra function definition in Matlab/octave 4⟩
⟨default options 20a⟩
⟨convert NaNs in p to 0s and create or modify s.w accordingly 6a⟩
⟨call the solver 21a⟩
⟨define s2s 21d⟩
```

The `slra` cost function is evaluated with `Mslra`:

21c ⟨`Mslra` 21c⟩≡

```
function [ph, info] = slra_misfit(p, s, R, psi)
opt.maxiter = 0; opt.Rini = R;
if nargin == 4, opt.psi = psi; end
r = size(R, 2) - size(R, 1);
[ph, info] = slra(p, s, r, opt);
```

Conversion from ($\mathscr{H}_{\mathbf{m},\mathbf{n}}$) to affine structure specification

21d ⟨*define* `s2s` 21d⟩≡ (21b)

```
function S = s2s(s, np)
⟨define constants 5a⟩, p = 1:np;
⟨default s.phi 5e⟩, [m, mp] = size(s.phi);
tmp = cumsum([1; s.m(:)]); Imb = tmp(1:end - 1); Ime = tmp(2:end) - 1;
tmp = cumsum([1; s.n(:)]); Inb = tmp(1:end - 1); Ine = tmp(2:end) - 1;
S = zeros(mp, n); ind = 1;
for j = 1:N
  for i = 1:q
    npij = s.m(i) + s.n(j) - 1;
```

```
    pij = p(ind:(ind + npij - 1)); ind = ind + npij;
    Hij = hankel(pij(1:s.m(i)), pij(s.m(i):end));
    S(Imb(i):Ime(i), Inb(j):Ine(j)) = Hij;
  end
end
```