Tensorlab

User Guide 2013-02-13

Laurent Sorber[*‡§] Marc Van Barel[*] Lieven De Lathauwer[†‡§]

Contents

[*]NALAG, Department of Computer Science, KU Leuven, Celestijnenlaan 200A, BE-3001 Leuven, Belgium (Laurent.Sorber@cs.kuleuven.be, Marc.VanBarel@cs.kuleuven.be).

[†]Group Science, Engineering and Technology, KU Leuven Kulak, E. Sabbelaan 53, BE-8500 Kortrijk, Belgium (Lieven.DeLathauwer@kuleuven-kulak.be).

[‡]SCD-SISTA, Department of Electrical Engineering (ESAT), KU Leuven, Kasteelpark Arenberg 10, BE-3001 Leuven, Belgium (Lieven.DeLathauwer@esat.kuleuven.be).

[§]iMinds Future Health Department, Kasteelpark Arenberg 10, BE-3001 Leuven, Belgium.

1 Getting started

Installation Unzip Tensorlab to any directory, browse to that location in MATLAB and run

```
addpath(pwd); % Add the current directory to the MATLAB search path.
savepath;     % Save the search path for future sessions.
```

Requirements Tensorlab requires MATLAB 7.9 (R2009b) or higher because of its dependency on the tilde operator `~`. If necessary, older versions of MATLAB can use Tensorlab by replacing `[~` and `~,` with `tmp`. To do so on Linux/OS X, browse to Tensorlab and run

```
sed -i -e 's/\[~/[tmp/g;s/~,/tmp,/g' *.m
```

in your system's terminal. However, most of the functionality in Tensorlab requires at the very minimum MATLAB 7.4 (R2007a) because of its extensive use of `bsxfun`.

Octave is only partially supported, mainly because it coerces nested functions into subfunctions. The latter do not share the workspace of their parent function, which is a feature used by Tensorlab in certain algorithms.

Tensorlab at a glance If you installed Tensorlab to the directory `tensorlab/`, run `doc tensorlab` from the command line for an overview of the toolboxes functionality. If Tensorlab is not in MATLAB's search path, another possibility is to browse to Tensorlab and run `help(pwd)`. Both commands display the file `Contents.m`. Although this user guide covers the most important aspects of Tensorlab, `Contents.m` serves as an important secondary resource for the functionality available in this toolbox.

Section 2 covers the basic operations on tensors. Tensor decompositions such as the canonical polyadic decomposition (CPD), low multilinear rank approximation (LMLRA) and block term decompositions (BTD) are discussed in Sections 3, 4 and 5, respectively. Many of these decompositions can be computed with complex optimization, that is, optimization of functions in complex variables. Section 6 introduces the necessary concepts and shows how to solve different types of complex optimization problems. Finally, Section 7 treats global optimization of bivariate (and polyanalytic) polynomials and rational functions, which appear as subproblems in tensor optimization.

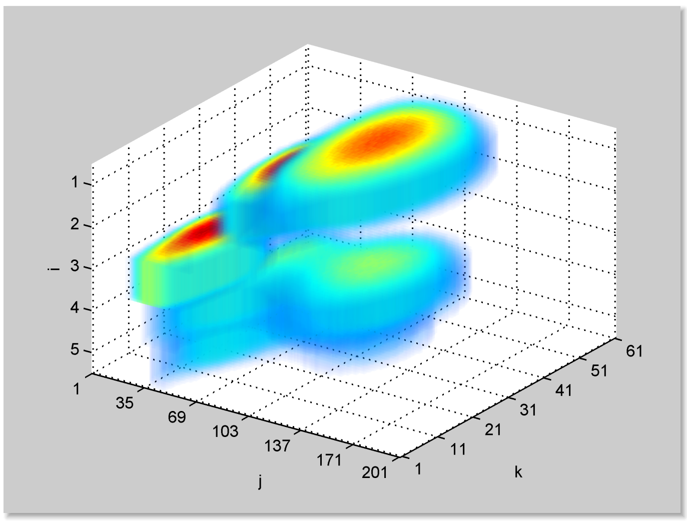
2 Tensor basics

2.1 Visualization

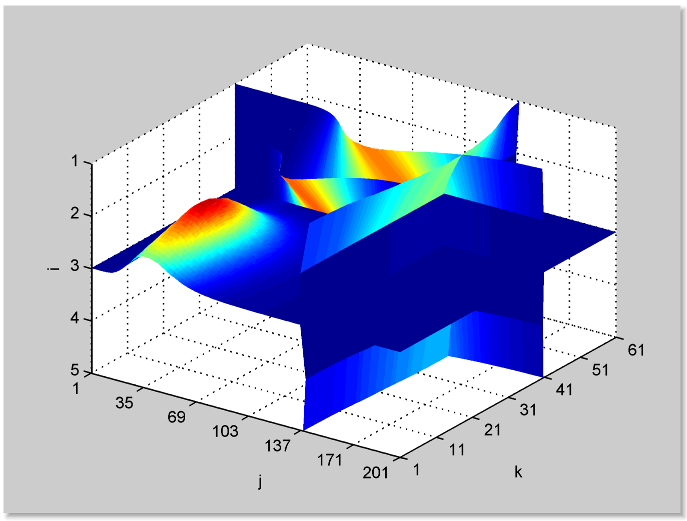
In Tensorlab, a tensor is represented as a dense N-dimensional MATLAB array, e.g., `T = rand(10,10,10);` creates a third-order tensor $\mathcal{T}$ with uniformly distributed elements. Tensorlab offers three methods to visualize third-order tensors: `slice3`, `surf3` and `voxel3`. The following example demonstrates these methods on the amino acids dataset [1]:

```
url = 'http://www.models.life.ku.dk/go-engine?filename=amino.mat';
urlwrite(url,'amino.mat'); % Download amino.mat in this directory.
load amino X;
figure(1); voxel3(X);
figure(2); surf3(X);
figure(3); slice3(X); colorbar;
```

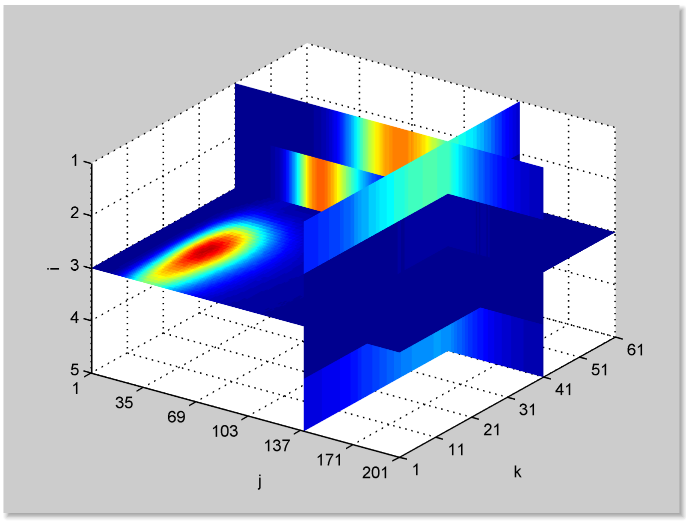
The resulting MATLAB figures are displayed in Figure 2.1.



(a) `voxel3`



(b) `surf3`



(c) `slice3`

Figure 2.1: Three functions for visualizing third-order tensors in Tensorlab.

2.2 Tensor operations

Frobenius norm The Frobenius norm of a tensor is the square root of the sum of square magnitudes of its elements. Given a tensor $\mathcal{T}$, its Frobenius norm can be computed by `norm(T(:))`, or by the Tensorlab function `frob(T)`.

Matricization and tensorization A tensor $\mathcal{T}$ can be matricized by `tens2mat(T,mode_row,mode_col)`. Let `size_tens = size(T)`, then the resulting matrix M is of size `prod(size_tens(mode_row))` by `prod(size_tens(mode_col))`. For example,

```
T = randn(3,5,7,9);
M = tens2mat(T,[1 3],[4 2]);
size(M)
```

outputs `[21 45]`. In a matricization, a given column (row) of M is generated by fixing the indices corresponding to `mode_col` (`mode_row`) and then looping over the remaining indices in the order `mode_row` (`mode_col`). To transform a matricized tensor M back into its original size `size_tens`, use `mat2tens(M,size_tens,mode_row,mode_col)`.

The most common use case is to matricize a tensor by placing its mode-n vectors as columns in a matrix, also called the mode-n matricization. This can be achieved by

```
T = randn(3,5,7);
n = 2;
M = tens2mat(T,n);
```

where the optional argument `mode_col` is implicitly equal to `[1:n-1 n+1:ndims(T)]`.

Tensor-matrix product In a mode-n tensor-matrix product, the tensor's mode-n vectors are premultiplied by a given matrix. In other words, `U*tens2mat(T,n)` is a mode-n matricization of the mode-n tensor-matrix product $\mathcal{T} \cdot_n U$. The function `tmprod(T,U,mode)` computes the tensor-matrix product $\mathcal{T} \cdot_{\texttt{mode(1)}} \texttt{U\{1\}} \cdot_{\texttt{mode(2)}} \texttt{U\{2\}} \cdots \cdot_{\texttt{mode(end)}} \texttt{U\{end\}}$. For example,

```
T = randn(3,5,7);
U = {randn(11,3),randn(13,5),randn(15,7)};
S = tmprod(T,U,1:3);
size(S)
```

outputs `[11 13 15]`.

Kronecker and Khatri–Rao product Tensorlab includes a fast implementation of the Kronecker product $A \otimes B$ with the function `kron(A,B)`, which overrides MATLAB's built-in implementation. Let A and B be matrices of size I by J and K by L, respectively, then the Kronecker product of A and B is the IK by JL matrix

$$A \otimes B := \begin{bmatrix} a_{11}B & \cdots & a_{1J}B \\ \vdots & \ddots & \vdots \\ a_{I1}B & \cdots & a_{IJ}B \end{bmatrix}.$$

The Khatri–Rao product $A \odot B$ can be computed by `kr(A,B)`. Let A and B both be matrices with N columns, respectively, then the Khatri–Rao product of A and B is the column-wise Kronecker product

$$A \odot B := \begin{bmatrix} \boldsymbol{a}_1 \otimes \boldsymbol{b}_1 & \cdots & \boldsymbol{a}_N \otimes \boldsymbol{b}_N \end{bmatrix}.$$

More generally, `kr(A,B,C,...)` and `kr(U)` compute the string of Khatri–Rao products $((A \odot B) \odot C) \odot \cdots$ and $((\texttt{U\{1\}} \odot \texttt{U\{2\}}) \odot \texttt{U\{3\}}) \odot \cdots$, respectively.

3 Canonical polyadic decomposition

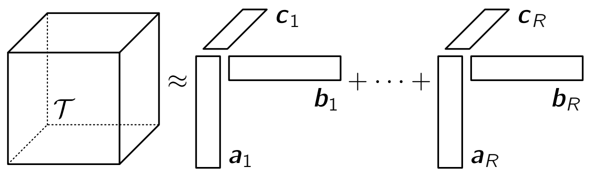


Figure 3.1: A canonical polyadic decomposition of a third-order tensor.

The canonical polyadic decomposition (CPD) [2, 8–11] approximates a tensor with a sum of R rank-one tensors. Let $\mathcal{A} \circ \mathcal{B}$ denote the outer product between an Nth-order tensor $\mathcal{A}$ and an Mth-order tensor $\mathcal{B}$, then $\mathcal{A} \circ \mathcal{B}$ is the $(N+M)$th-order tensor defined by $(\mathcal{A} \circ \mathcal{B})_{i_1 \cdots i_N j_1 \cdots j_M} = a_{i_1 \cdots i_N} \cdot b_{j_1 \cdots j_M}$. For example, let $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ be nonzero vectors in $\mathbb{R}^n$, then $\boldsymbol{a} \circ \boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{b}^{\mathrm{T}}$ is a rank-one matrix and $\boldsymbol{a} \circ \boldsymbol{b} \circ \boldsymbol{c}$ is defined to be a rank-one tensor.

Let $\mathcal{T}$ be an Nth-order tensor of dimensions $I_1 \times I_2 \times \cdots \times I_N$, and let $U^{(n)}$ be matrices of size $I_n \times R$ for $1 \leq n \leq N$, then

$$\mathcal{T} \approx \sum_{r=1}^{R} \boldsymbol{u}_r^{(1)} \circ \boldsymbol{u}_r^{(2)} \circ \cdots \circ \boldsymbol{u}_r^{(N)}$$

is a canonical polyadic decomposition of $\mathcal{T}$ in R rank-one terms. A visual representation of this decomposition in the third-order case is shown in Figure 3.1.

3.1 Problem and tensor generation

Generating pseudorandom factor matrices A cell array of pseudorandom factor matrices `U = {U{1},U{2},...}` corresponding to a CPD of a tensor of dimensions `size_tens` in R rank-one terms can be generated with

```
size_tens = [7 8 9]; R = 4;
U = cpd_rnd(size_tens,R);
```

By default `cpd_rnd` generates `U{n}` as `randn(size_tens(n),R)`. Other generators can also be specified with an options structure, e.g.,

```
options.Real = @rand;
options.Imag = @randn;
U = cpd_rnd(size_tens,R,options);
```

and the inline equivalent

```
U = cpd_rnd(size_tens,R,struct('Real',@rand,'Imag',@randn));
```

generate `U{n}` as `rand(size_tens(n),R)+randn(size_tens(n),R)*1i`.

Generating the associated full tensor Given a cell array of factor matrices `U = {U{1},U{2},...}`, its associated full tensor $\mathcal{T}$ can be computed by tensorizing its mode-1 unfolding $M = U^{(1)} \cdot (U^{(N)} \odot U^{(N-1)} \odot \cdots \odot U^{(2)})^{\mathrm{T}}$ as follows

```
M = U{1}*kr(U(end:-1:2)).';
size_tens = cellfun('size',U,1);
T = mat2tens(M,size_tens,1);
```

or simply with `T = cpdgen(U);`.

3.2 Computing the CPD

The basic method To compute the CPD of a tensor $\mathcal{T}$ in R rank-one terms, call `cpd(T,R)`. For example,

```
% Generate pseudorandom factor matrices U and their associated full tensor T.
size_tens = [7 8 9]; R = 4;
U = cpd_rnd(size_tens,R);
T = cpdgen(U);

% Compute the CPD of the full tensor T.
Uhat = cpd(T,R);
```

generates a real rank-4 tensor and decomposes it. Internally, `cpd` first *compresses the tensor* using a low multilinear rank approximation (see Section 4) if it is worthwhile, then chooses a method to *generate an initialization* `U0` (e.g., `cpd_gevd`), after which it *executes an algorithm* to compute the CPD given the initialization (e.g., `cpd_nls`) and finally decompresses the tensor and *refines the solution* (if compression was applied).

Setting the options The different steps in `cpd` are customizable by supplying the method with an options structure (see `help cpd` for more information), e.g.,

```
options.Compression = false;                     % Disable compression step.
options.Initialization = @cpd_rnd;               % Select pseudorandom initialization.
options.Algorithm = @cpd_als;                    % Select ALS as the main algorithm.
options.AlgorithmOptions.LineSearch = @cpd_els; % Add exact line search.
options.AlgorithmOptions.TolFun = 1e-12;         % Set stop criteria.
options.AlgorithmOptions.TolX = 1e-12;
Uhat = cpd(T,R,options);
```

The structures `options.InitializationOptions`, `options.AlgorithmOptions` and `options.RefinementOptions` will be passed as options structures to the algorithms corresponding to initialization, algorithm and refinement steps, respectively. For example, in the example above `cpd` will call `cpd_als` as `cpd_als(T,options.Initialization(T,R),options.AlgorithmOptions)`.

Viewing the algorithm output Each step may also output additional information specific to that step. For instance, most CPD algorithms such as `cpd_als` will keep track of the number of iterations and objective function value. To obtain this information, capture the second output:

```
[Uhat,output] = cpd(T,R,options);
```

and inspect its fields, for example by plotting the objective function value:

```
semilogy(0:output.Algorithm.iterations,sqrt(2*output.Algorithm.fval));
xlabel('iteration');
ylabel('frob(T-cpdgen(U))');
grid on;
```

Computing the error One way of computing the relative error between a tensor $\mathcal{T}$ and its CPD approximation defined by `Uhat` is using the Frobenius norm as in

```
relerr = frob(T-cpdgen(Uhat))/frob(T);
```

Another is to compute the relative error between the factor matrices $U^{(n)}$ that generated $\mathcal{T}$ and their approximations `Uhat{n}` resulting from `cpd`. Due to the permutation and scaling indeterminacies of the CPD, the columns of `Uhat` may need to be permuted and scaled to match those of `U` before comparing them to each other. The function `cpderr` takes care of these indeterminacies and then computes the relative error between the given two sets of factor matrices. I.e.,

```
relerr = cpderr(U,Uhat);
```

returns a vector in which the nth entry is the relative error between `U{n}` and `Uhat{n}`. This method is also applicable when `Uhat{n}` is an under- or overfactoring of the solution, meaning `Uhat{n}` comprises fewer or more rank-one terms than the tensor's rank, respectively. In the underfactoring case, `Uhat{n}` is padded with zero-columns, and in the overfactoring case only the columns in `Uhat{n}` that best match those in `U{n}` are kept.

3.3 Structure, (partial) symmetry and nonnegativity constraints

Most CPD algorithms in Tensorlab are implemented for tensors of arbitrary order and for both real and complex decompositions and data. Some are restricted to third-order tensors by design and this is indicated by the prefix `cpd3_` (e.g., `cpd3_sgsd` and `cpd3_sd`). Furthermore, the prefixes `cpdnn_` and `cpds_` represent solvers for nonnegative and structured or (partially) symmetric decompositions, respectively, and are examined in the following subsections.

Nonnegative decompositions

If some or all of the factor matrices are to be nonnegative, select an algorithm with the prefix `cpdnn_`. Currently, the only available (family of) algorithm(s) is `cpdnn_nlsb`, which solves the decomposition as a bound-constrained nonlinear least squares problem.

```
% Generate problem with two nonnegative factor matrices.
size_tens = [7 8 9]; R = 4;
U = cpd_rnd(size_tens,R);
U(1:2) = {abs(U{1}),abs(U{2})};
T = cpdgen(U);

% Solve problem as a nonnegative decomposition.
% The initialization itself does not need to be nonnegative.
U0 = cpd_rnd(size_tens,R);
options.Nonnegative = [1 2];
[Uest,output] = cpdnn_nlsb(T,U0,options);
```

Structured and (partially) symmetric decompositions

Algorithms with the prefix `cpds_` are capable of computing (partially) symmetric decompositions (e.g., $U^{(1)} \equiv U^{(2)}$) and decompositions with structured factor matrices (e.g., $U^{(n)}$ is a (block) Toeplitz, Hankel, Vandermonde, symmetric or Hermitian matrix) [14]. Moreover, these methods also support tensors with missing elements. Currently, two families of algorithms are offered: `cpds_minf` and `cpds_nls` solve structured decompositions with (complex) unconstrained nonlinear optimization and nonlinear least squares methods, respectively.

Missing elements in the tensor To compute a CPD of a tensor with missing elements, simply store the missing elements as `NaN` and call `cpds_minf` or `cpds_nls` with the tensor and a cell array of initial factor matrices:

```
% Generate problem with two nonnegative factor matrices.
size_tens = [7 8 9]; R = 4;
```

```
U = cpd_rnd(size_tens,R);
T = cpdgen(U);

% Set about 50% of the tensor's elements as missing.
idx = randi(numel(T),round(0.50*numel(T)),1);
T(idx) = NaN;

% Solve problem as a nonnegative decomposition.
% The initialization itself does not need to be nonnegative.
U0 = cpd_rnd(size_tens,R);
[Uest,output] = cpds_nls(T,U0);
```

(Partially) symmetric decompositions One or more factor matrices can be constrained to be equal to each other by defining `options.Symmetry` as a cell array that partitions the modes 1 to `ndims(T)`. For example, to impose the first and second factor matrix of a fourth-order tensor to be equal to each other, set `options.Symmetry = {[1 2],3,4}`. A fully symmetric CPD can be computed by setting `options.Symmetry = {[1 2 3 4]}`. When symmetry is imposed, the initialization `U0` requires only as many factor matrices as there are partitions in `options.Symmetry`. Hence, to compute a fully symmetric CPD, the initialization `U0` consists of only one factor matrix. The following example computes a partially symmetric CPD of a fourth-order tensor:

```
% Generate partially symmetric problem of the form U = {A,B,A,C}.
size_tens = [8 7 8 9]; R = 4;
U = cpd_rnd(size_tens,R);
U{3} = U{1};
T = cpdgen(U);

% Solve problem as a partially symmetric decomposition.
% The solution is of the form Uest = {A,B,C}.
U0 = cpd_rnd(size_tens,R);
U0 = U0([1 2 4]);
options.Symmetry = {[1 3],2,4};
[Uest,output] = cpds_nls(T,U0,options);
```

Structured factor matrices: structured variables If the nth factor matrix $U^{(n)}$ is structured and each of its elements is either a variable or a constant, the matrix is said to have structured variables. Let $\boldsymbol{z}$ be the column vector of variables that defines $U^{(n)}$, then $\boldsymbol{z}$ is said to generate $U^{(n)}$. The structure of the nth factor matrix is defined by the nth element of the cell array `options.Structure`. If $U^{(n)}$ is a matrix of structured variables, `options.Structure{n}` should be a matrix of the same size as $U^{(n)}$ and its elements indices that correspond to variables in $\boldsymbol{z}$. For example, if $U^{(n)}$ is a Toeplitz matrix of the form

$$\begin{bmatrix} a & b & c & d \\ 3 & a & b & c \\ 2 & 3 & a & b \\ 1 & 2 & 3 & a \\ 0 & 1 & 2 & 3 \end{bmatrix},$$

generated by the vector $\boldsymbol{z} = \begin{bmatrix} a & b & c & d \end{bmatrix}^{\mathrm{T}}$, then set

```
options.Structure = cell(1,ndims(T));
options.Structure{n} = [1 2 3 4; ...
                        0 1 2 3; ...
                        0 0 1 2; ...
                        0 0 0 1; ...
                        0 0 0 0];
```

or inline as `options.Structure{n} = toeplitz([1 0 0 0 0],1:4);`. In this example, the indices 1 to 4 refer to the first to fourth element in the generator z, and a 0 indicates that position in $U^{(n)}$ is a constant. The initialization `U0{n}` for $U^{(n)}$ may be given in the same format as $U^{(n)}$ itself (making sure that any constants in $U^{(n)}$ are also present in `U0{n}`). If there are no constants in $U^{(n)}$, the initialization `U0{n}` may also be given in the form of a generator z. In the following example, a CPD with a Toeplitz factor matrix with constants is computed:

```
% Generate problem where U{1} has a Toeplitz structure with constants.
size_tens = [7 8 9]; R = 4;
U = cpd_rnd(size_tens,R);
z = randn(4,1); % Generator.
c = [z(1) 5:-1:0]; % Constants.
U{1} = toeplitz(c,z);
T = cpdgen(U);

% Solve problem as a structured decomposition.
% The solution is of the form Uest = {z,U{2},U{3}}.
U0 = cpd_rnd(size_tens,R);
z0 = randn(4,1);
U0{1} = toeplitz([z0(1) c(2:end)],z0); % Generate U0{1} with z0 and fill in constants.
options = struct('Structure',{cell(1,ndims(T))});
options.Structure{1} = toeplitz([1; zeros(6,1)],1:4);
[Uest,output] = cpds_nls(T,U0,options);
```

The `cpds_` solvers return structured factor matrices in their generator format. In the example above, `Uest{1}` is returned as a vector of the same size as `z`. As a last example, a CPD with a Hankel factor matrix without constants is computed:

```
% Generate problem where U{1} has a Hankel structure without constants.
size_tens = [7 8 9]; R = 4;
U = cpd_rnd(size_tens,R);
z = randn(10,1); % Generator.
U{1} = hankel(z(1:7),z(7:10));
T = cpdgen(U);

% Solve problem as a structured decomposition.
% The solution is of the form Uest = {z,U{2},U{3}}.
U0 = cellfun(@(u)u+0.1*randn(size(u)),U,'UniformOutput',false);
z0 = randn(10,1);
U0{1} = z0; % U0{1} is initialized as a generator.
options = struct('Structure',{cell(1,ndims(T))});
options.Structure{1} = hankel(1:7,7:10);
[Uest,output] = cpds_nls(T,U0,options);
```

Structured factor matrices: structured analytic expressions If the nth factor matrix $U^{(n)}$ is a matrix-valued function of a vector z, the matrix is said to have structured expressions. Furthermore, if these expressions are analytic in the variables of z, and hence do not depend on the complex conjugates of the variables in z, the matrix is said to have structured analytic expressions. For such matrices, set `options.Structure{n} = {S,J}`, where `S(z)` is a function that generates $U^{(n)}$ given the generator z and `J(z)` computes the Jacobian $\frac{\partial \operatorname{vec}(U^{(n)})}{\partial z^{\mathsf{T}}}$ at `z`. Alternatively, set `J = 'Jacobian'` to approximate the Jacobian with numerical differentiation. For example, if $U^{(n)}$ is a Vandermonde matrix of the form

$$\begin{bmatrix} a^3 & a^2 & a & 1 \\ b^3 & b^2 & b & 1 \\ c^3 & c^2 & c & 1 \\ d^3 & d^2 & d & 1 \end{bmatrix},$$

generated by the vector $\boldsymbol{z} = \begin{bmatrix} a & b & c & d \end{bmatrix}^{\mathsf{T}}$, then set

```
options.Structure = cell(1,ndims(T));
options.Structure{n} = {@(z)vander(z), ...
                        @(z)[diag(3*z.^2); diag(2*z); eye(4); zeros(4)]};
```

or use `options.Structure{n} = {@(z)vander(z),'Jacobian'};` to approximate `J(z)` with finite differences using `deriv` (cf. Section 6). The initialization `U0{n}` must be given in the form of a generator $\boldsymbol{z}$. Likewise, structured factor matrices in the solution `Uest` are returned in generator format. The following example computes a CPD with a Vandermonde factor matrix:

```
% Generate problem where U{1} has a Vandermonde structure.
size_tens = [4 8 9]; R = 4;
U = cpd_rnd(size_tens,R);
z = randn(4,1); % Generator.
U{1} = vander(z);
T = cpdgen(U);

% Solve problem as a structured decomposition.
% The solution is of the form Uest = {z,U{2},U{3}}.
U0 = cpd_rnd(size_tens,R);
z0 = randn(4,1);
U0{1} = z0; % U0{1} is initialized as a generator.
options = struct('Structure',{cell(1,ndims(T))});
options.Structure{1} = {@(z)vander(z),'Jacobian'};
[Uest,output] = cpds_nls(T,U0,options);
```

Structured factor matrices: structured nonanalytic expressions Factor matrices $U^{(n)}$ which are matrix-valued functions of a vector $\boldsymbol{z}$ and its complex conjugate $\overline{\boldsymbol{z}}$ are said to have structured nonanalytic expressions. Currently, only the family of unconstrained nonlinear optimization solvers `cpds_minf` can be applied to this type of problem. Similar to the analytic case, set `options.Structure{n} = {S,J}`, where `S(z)` is a function that generates $U^{(n)}$ given the generator $\boldsymbol{z}$ and `J(z)` computes the complex Jacobian $\begin{bmatrix} \frac{\partial \operatorname{vec}(U^{(n)})}{\partial z^{\mathsf{T}}} & \frac{\partial \operatorname{vec}(U^{(n)})}{\partial \overline{z}^{\mathsf{T}}} \end{bmatrix}$ at `z` (cf. Section 6.1). Alternatively, set `J = 'Jacobian-C'` to approximate the complex Jacobian with numerical differentiation. For example, if $U^{(n)}$ is a Hermitian Toeplitz matrix of the form

$$\begin{bmatrix} a+\overline{a} & b & c & d \\ \overline{b} & a+\overline{a} & b & c \\ \overline{c} & \overline{b} & a+\overline{a} & b \\ \overline{d} & \overline{c} & \overline{b} & a+\overline{a} \end{bmatrix},$$

generated by the vector $\boldsymbol{z} = \begin{bmatrix} a & b & c & d \end{bmatrix}^{\mathsf{T}}$, then set

```
N = zeros(4); I = eye(4);
dUdz = sparse([I(:) [N(1:4) I(1:12)].' [N(1:8) I(1:8)].' [N(1:12) I(1:4)].']);
S = @(z)toeplitz([z(1);conj(z(2:4))],z)+conj(z(1))*eye(4);
J = @(z)[dUdz flipud(dUdz)];
options.Structure = cell(1,ndims(T));
options.Structure{n} = {S,J};
```

or use `options.Structure{n} = {S,'Jacobian-C'};` to approximate `J(z)` with finite differences using `deriv` (cf. Section 6). Notice that in the example above, the complex Jacobian is constant. In this case, it is preferrable to set `J = [dUdz flipud(dUdz)];`. The following example computes a CPD with a Hermitian Toeplitz factor matrix:

```
% Generate problem where U{1} has a Hermitian Toeplitz structure.
size_tens = [4 8 9]; R = 4;
```

```
U = cpd_rnd(size_tens,R,struct('Imag',@randn));
S = @(z)toeplitz([z(1);conj(z(2:4))],z)+conj(z(1))*eye(4);
z = randn(4,1)+randn(4,1)*1i; % Generator.
U{1} = S(z);
T = cpdgen(U);

% Solve problem as a structured decomposition.
% The solution is of the form Uest = {z,U{2},U{3}}.
U0 = cellfun(@(u)u+0.1*(randn(size(u))+randn(size(u))*1i),U,'UniformOutput',false);
z0 = randn(4,1)+randn(4,1)*1i;
U0{1} = z0; % U0{1} is initialized as a generator.
options = struct('Structure',{cell(1,ndims(T))});
options.Structure{1} = {S,'Jacobian-C'};
[Uest,output] = cpds_minf(T,U0,options);
```

3.4 Choosing the number of rank-one terms R

To help choose the number of rank-one terms R, use the `rankest` tool. Running `rankest(T)` plots an L-curve which represents the balance between the relative error of the CPD and the number of rank-one terms R. The following example applies `rankest` on the amino acids dataset [1]:

```
url = 'http://www.models.life.ku.dk/go-engine?filename=amino.mat';
urlwrite(url,'amino.mat'); % Download amino.mat in this directory.
load amino X;
rankest(X);
```

The resulting figure is shown in Figure 3.2. The algorithm computes the CPD of the given tensor for various values of R, starting at the smallest integer for which the lower bound on the relative error (displayed as a solid blue line) is smaller than the specified `options.MaxRelErr`. The lower bound is based on the truncation error of the tensor's multilinear singular values [5]. The number of rank-one terms is increased until the relative error of the approximation is less than `options.MinRelErr`. In a sense, the corner of the resulting L-curve makes an optimal trade-off between accuracy and compression. The `rankest` tool computes the number of rank-one terms R corresponding to the L-curve corner and marks it on the plot with a square. This optimal number of rank-one terms is also `rankest`'s first output. By capturing it as `R = rankest(X)`, the L-curve is no longer plotted.

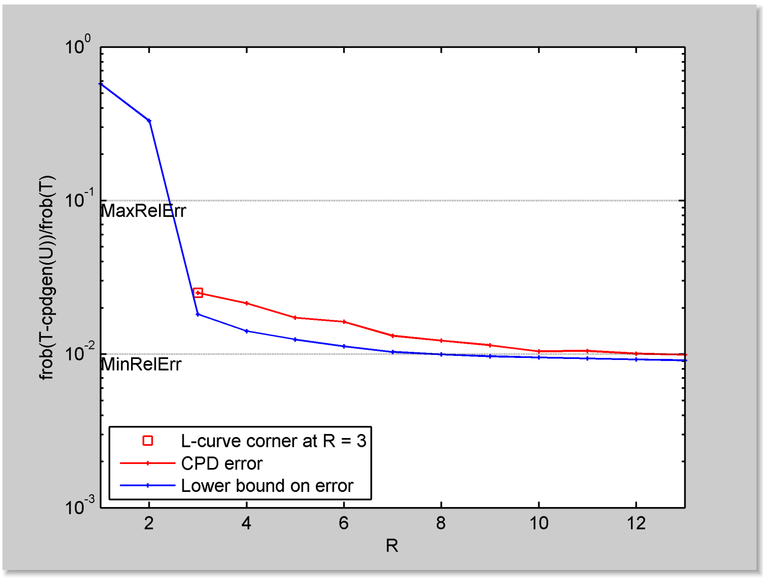


Figure 3.2: The `rankest` tool for choosing the number of rank-one terms R in a CPD.

4 Low multilinear rank approximation

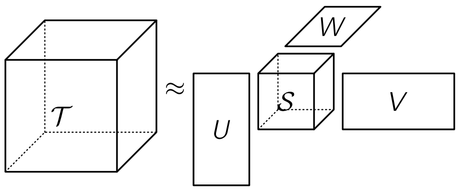


Figure 4.1: A low multilinear rank approximation of a third-order tensor.

A low multilinear rank approximation (LMLRA) [11, 17] approximates a tensor by a smaller (core) tensor and a set of factor matrices. Let $\mathcal{T}$ and $\mathcal{S}$ be Nth-order tensors of dimensions $I_1 \times I_2 \times \cdots \times I_N$ and $J_1 \times J_2 \times \cdots \times J_N$, respectively, let $U^{(n)}$ be matrices of size $I_n \times J_n$ ($J_n \leq I_n$) for $1 \leq n \leq N$, and let $\cdot \bullet_n \cdot$ denote the mode-n tensor-matrix product (see Section 2.2), then

$$\mathcal{T} \approx \mathcal{S} \bullet_1 U^{(1)} \bullet_2 U^{(2)} \bullet_3 \cdots \bullet_N U^{(N)}$$

is a low multilinear rank approximation of $\mathcal{T}$ in the core tensor $\mathcal{S}$ and factor matrices $U^{(n)}$. A visual representation of this decomposition in the third-order case is shown in Figure 4.1.

4.1 Problem and tensor generation

Generating pseudorandom factor matrices and core tensor A cell array of pseudorandom unitary factor matrices `U = {U{1},U{2},...}` and a core tensor $\mathcal{S}$ of dimensions `size_core` corresponding to a LMLRA of a tensor of dimensions `size_tens` can be generated with

```
size_tens = [17 19 21];
size_core = [3 5 7];
[U,S] = lmlra_rnd(size_tens,size_core);
```

By default `lmlra_rnd` generates `U{n}` and $\mathcal{S}$ using `randn`, after which `U{n}` is orthogonalized. Other generators can also be specified with an options structure, e.g.,

```
options.Real = @rand;
options.Imag = @rand;
[U,S] = lmlra_rnd(size_tens,size_core,options);
```

and its inline equivalent

```
[U,S] = lmlra_rnd(size_tens,size_core,struct('Real',@rand,'Imag',@rand));
```

Generating the associated full tensor Given a cell array of factor matrices `U = {U{1},U{2},...}` and core tensor $\mathcal{S}$, its associated full tensor $\mathcal{T}$ can be computed with the tensor-matrix product as

```
T = tmprod(S,U,1:length(U));
```

or simply by

```
T = lmlragen(U,S);
```

4.2 Computing a LMLRA

The basic method To compute a LMLRA of a tensor $\mathcal{T}$ so that the core tensor is of size `size_core`, call `lmlra(T,size_core)`. For example,

```
% Generate pseudorandom LMLRA (U,S) and associated full tensor T.
size_tens = [17 19 21];
size_core = [3 5 7];
[U,S] = lmlra_rnd(size_tens,size_core);
T = lmlragen(U,S);

% Compute a LMLRA of a noisy full tensor T.
T = T+0.01*randn(size_tens);
[Uhat,Shat] = lmlra(T,size_core);
```

generates a real rank-(3,5,7) tensor and computes its low multilinear rank approximation. Internally, `lmlra` chooses a method to *generate an initialization* `U0` (e.g., `mlsvd`), after which it *executes an algorithm* to compute the LMLRA given the initialization (e.g., `lmlra_hooi`).

In many cases, computing a best low multilinear rank approximation may not be necessary and a (sequentially) truncated multilinear SVD (T-MLSVD) may be sufficiently accurate. To compute a T-MLSVD where the core tensor is of size `size_core`, call `mlsvd(T,size_core)`.

Additionally, both `lmlra` and `mlsvd` can heuristically determine a `size_core` so that the relative error of the approximation in Frobenius norm is no larger than a specified tolerance `tol`. To use this heuristic, call `lmlra(T,tol)` or `mlsvd(T,tol)`.

Setting the options The two steps in `lmlra` are customizable by supplying the method with an options structure (see `help lmlra` for more information), e.g.,

```
options.Initialization = @lmlra_rnd;     % Select pseudorandom initialization.
options.Algorithm = @lmlra_hooi;         % Select HOOI as the main algorithm.
options.AlgorithmOptions.TolFun = 1e-12; % Set stop criteria.
options.AlgorithmOptions.TolX = 1e-12;
[Uhat,Shat] = lmlra(T,size_core,options);
```

The structures `options.InitializationOptions` and `options.AlgorithmOptions` will be passed as options structures to the algorithms corresponding to initialization and algorithm steps, respectively.

Viewing the algorithm output Each step may also output additional information specific to that step. For instance, most LMLRA algorithms such as `lmlra_hooi` will keep track of the number of iterations and the difference in subspace angle of every two successive iterates `sangle`. To obtain this information, capture the third output:

```
[Uhat,Shat,output] = lmlra(T,size_core,options);
```

and inspect its fields, for example by plotting the difference in subspace angle:

```
semilogy(0:output.Algorithm.iterations,output.Algorithm.sangle);
xlabel('iteration');
ylabel('subspace(U\{1\},U\{1\}_{prev})');
grid on;
```

Higher-order and complex decompositions Currently, only `mlsvd` and `lmlra_hooi` are implemented for tensors of arbitrary order and for both real and complex decompositions and data. Other algorithms are restricted to third-order tensors by design and this is indicated by the prefix `lmlra3_` (e.g., `lmlra3_dgn` and `lmlra3_rtr`). The latter methods are currently also only applicable to real data, although this may change in future versions of Tensorlab.

Computing the error One way of computing the relative error between a tensor $\mathcal{T}$ and its LMLRA defined by `Uhat` and `Shat` is using the Frobenius norm as in

```
relerr = frob(T-lmlragen(Uhat,Shat))/frob(T);
```

If the factor matrices $U^{(n)}$ that generated $\mathcal{T}$ are known, the subspace angle between them and their approximations `Uhat{n}` resulting from a `lmlra` or `mlsvd` of the noisy tensor is also a measure of the approximation error. The function `lmlraerr` computes the subspace angle between the given two sets of factor matrices. In other words,

```
sangle = lmlraerr(U,Uhat);
```

returns a vector in which the *n*th entry is `subspace(U{n},Uhat{n})`.

4.3 Choosing the size of the core tensor

To help choose the size of the core tensor `size_core`, use the `mlrankest` tool. Running `mlrankest(T)` plots an L-curve which represents the balance between an upper bound [5] on the relative error of a LMLRA of $\mathcal{T}$ for different core tensor sizes, and the LMLRA's compression ratio. The following example applies `mlrankest` on the amino acids dataset [1]:

```
url = 'http://www.models.life.ku.dk/go-engine?filename=amino.mat';
urlwrite(url,'amino.mat'); % Download amino.mat in this directory.
load amino X;
mlrankest(X);
```

The resulting figure is shown in Figure 4.2. The corner of this L-curve is often a good estimate of the optimal trade-off between accuracy and compression. The core tensor size `size_core` corresponding to the L-curve corner is marked on the plot with a square and is also `mlrankest`'s first output. By capturing it as in `size_core = mlrankest(X)`, the L-curve is no longer plotted. All together, a LMLRA of the amino acids dataset can be computed in only a few lines:

```
url = 'http://www.models.life.ku.dk/go-engine?filename=amino.mat';
urlwrite(url,'amino.mat'); % Download amino.mat in this directory.
load amino X;
size_core = mlrankest(X); % Optimal core tensor size corresponding to L-curve corner.
[U,S] = lmlra(X,size_core);
```

Additionally, the compression and relative error of other choices of `size_core` can be viewed by using the figure's Data Cursor tool.

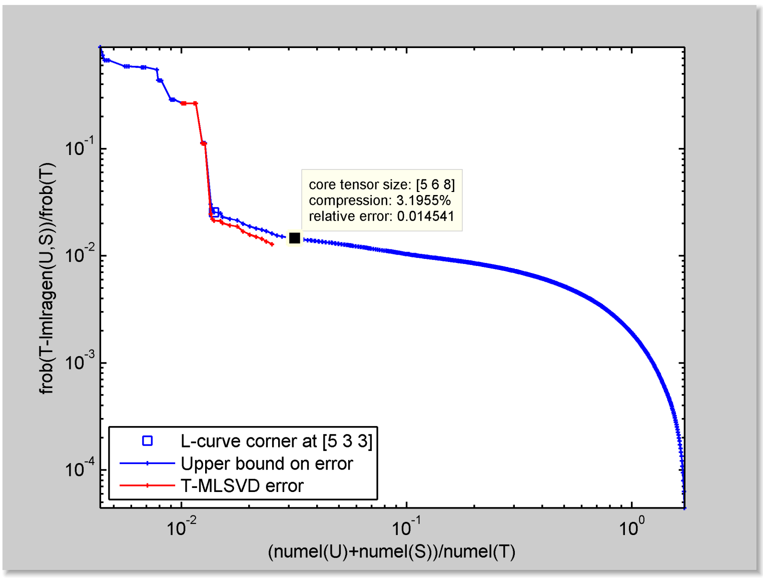


Figure 4.2: The `mlrankest` tool for choosing the core tensor size `size_core` in a LMLRA.

5 Block term decompositions

5.1 (rank-$L_r \circ$ rank-1) and rank-$(L_r, L_r, 1)$ BTD

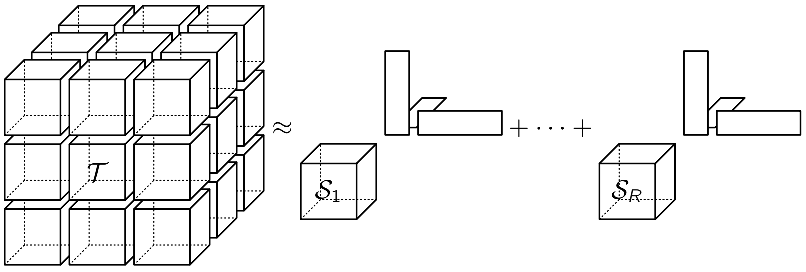


Figure 5.1: A (rank-$L_r \circ$ rank-1) block term decomposition of a sixth-order tensor.

The (rank-$L_r \circ$ rank-1) BTD The (rank-$L_r \circ$ rank-1) block term decomposition (BTD) [15] approximates a tensor with a sum of R terms, each of which is an outer product of a rank-L_r tensor and a rank-one tensor.

Let $\mathcal{T}$ be an Nth-order tensor of dimensions $I_1 \times I_2 \times \cdots \times I_N$, and let $U^{(n)}$ be matrices of size $I_n \times \sum_{r=1}^{R} L_r$ when $1 \leq n \leq P$ and of size $I_n \times R$ when $P < n \leq N$. Furthermore, define $\mathcal{S}_r$ as the rth rank-L_r tensor

$$\mathcal{S}_r := \sum_{v=1+\sum_{w=1}^{r-1} L_w}^{\sum_{w=1}^{r} L_w} \boldsymbol{u}_v^{(1)} \circ \boldsymbol{u}_v^{(2)} \circ \cdots \circ \boldsymbol{u}_v^{(P)}$$

then

$$\mathcal{T} \approx \sum_{r=1}^{R} \mathcal{S}_r \circ \left(\boldsymbol{u}_r^{(P+1)} \circ \boldsymbol{u}_r^{(P+2)} \circ \cdots \circ \boldsymbol{u}_r^{(N)} \right)$$

is a (rank-$L_r \circ$ rank-1) block term decomposition of $\mathcal{T}$ in R terms. A visual representation of this decomposition in the sixth-order case is shown in Figure 5.1.

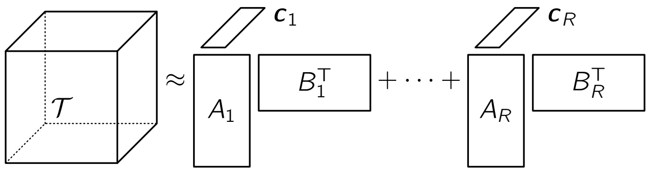


Figure 5.2: A rank-$(L_r, L_r, 1)$ block term decomposition of a third-order tensor.

The rank-$(L_r, L_r, 1)$ BTD Two important special cases of the (rank-L_r $\circ$ rank-1) BTD can be distinguished. First, the CPD (cf. Section 3) is a special case where the tensors $\mathcal{S}_r$ are rank-one, i.e., $L_r \equiv 1$. Second, the so-called rank-$(L_r, L_r, 1)$ block term decomposition [3, 4, 6] is a special case for third-order tensors where $\mathcal{S}_r \equiv A_r \cdot B_r^\mathsf{T}$ are rank-L_r matrices. In other words, when $P = 2$,

$$\mathcal{T} \approx \sum_{r=1}^{R} \mathcal{S}_r \circ \boldsymbol{u}_r^{(3)}$$

is a rank-$(L_r, L_r, 1)$ block term decomposition of $\mathcal{T}$ in R terms. A visual representation of this decomposition is shown in Figure 5.2. Herein, $U^{(1)} \equiv \begin{bmatrix} A_1 & \cdots & A_R \end{bmatrix}$, $U^{(2)} \equiv \begin{bmatrix} B_1 & \cdots & B_R \end{bmatrix}$ and $U^{(3)} \equiv \begin{bmatrix} \boldsymbol{c}_1 & \cdots & \boldsymbol{c}_R \end{bmatrix}$.

5.2 Problem and tensor generation

Generating pseudorandom factor matrices Let `L = [L(1),L(2),...]` be a vector of rank parameters corresponding to the ranks L_r of the tensors $\mathcal{S}_r$ and let `Q = [Q(1),Q(2),...]` be the index set of modes $\{P + 1, \ldots, N\}$ corresponding to the rank-one part of each term in a (rank-L_r $\circ$ rank-1) BTD. For example, in a rank-$(L_r, L_r, 1)$ BTD, `Q = 3`. Then, a cell array of pseudorandom factor matrices `U = {U{1},U{2},...}` corresponding to a (rank-L_r $\circ$ rank-1) BTD of a tensor of dimensions `size_tens` in R terms defined by `L` and `Q` can be generated with

```
size_tens = [7 8 9]; L = [2 2]; Q = 3;
U = btdLx1_rnd(size_tens,L,Q);
```

By default `btdLx1_rnd` generates `U{n}` as `randn(size_tens(n),sum(L))` when $1 \leq n \leq P$ or as `randn(size_tens(n),length(L))` when $P < n \leq N$. Other generators can also be specified with an options structure, e.g.,

```
options.Real = @rand;
options.Imag = @randn;
U = btdLx1_rnd(size_tens,L,Q,options);
```

and the inline equivalent

```
U = btdLx1_rnd(size_tens,L,Q,struct('Real',@rand,'Imag',@randn));
```

generate `U{n}` using a uniformly distributed real part and a normally distributed imaginary part.

Generating the associated full tensor Given a cell array of factor matrices `U = {U{1},U{2},...}` and a vector of ranks `L = [L(1),L(2),...]`, its associated full tensor $\mathcal{T}$ can be computed with

```
T = btdLx1gen(U,L);
```

5.3 Computing the BTD

With a specific algorithm Currently, Tensorlab does not offer a high-level function for computing block term decompositions in an automatic way. Instead, the user must generate his or her own initialization and select a specific (family of) algorithm(s), e.g., `btdLx1_als` or `btdLx1_nls`, to compute the BTD given the initialization. For example,

```
% Generate pseudorandom factor matrices U and associated full tensor T.
size_tens = [7 8 9]; L = [2 2]; Q = 3;
U = btdLx1_rnd(size_tens,L,Q);
T = btdLx1gen(U,L);

% Generate initialization U0 and compute the BTD with nonlinear least squares.
U0 = btdLx1_rnd(size_tens,L,Q);
Uhat = btdLx1_nls(T,U0,L);
```

generates and decomposes a rank-$(L_r, L_r, 1)$ BTD.

Setting the options The selected algorithm may be customizable by supplying the method with an options structure (see the relevant `help` for more information), e.g.,

```
options.LineSearch = @cpd_els;      % Add CPD-like exact line search.
options.TolFun = 1e-12;             % Set stop criteria.
options.TolX = 1e-12;
Uhat = btdLx1_als(T,U0,L,options); % Decompose using alternating least squares.
```

Viewing the algorithm output The selected method also returns output specific to the algorithm, such as the number of iterations and the algorithm's objective function value. To obtain this information, capture the second output:

```
[Uhat,output] = btdLx1_als(T,U0,L,options);
```

and inspect its fields, for example by plotting the objective function value:

```
semilogy(0:output.iterations,sqrt(2*output.fval));
xlabel('iteration');
ylabel('frob(T-btdLx1gen(U,L))');
grid on;
```

Computing the error A measure of how well the BTD approximates a tensor $\mathcal{T}$ is the relative error in Frobenius norm between $\mathcal{T}$ and its BTD approximation defined by `Uhat` and `L`, which can be computed as `relerr = frob(T-btdLx1gen(Uhat,L))/frob(T);`. Currently, Tensorlab does not offer a method to compare factor matrices between block term decompositions.

5.4 Structure, (partial) symmetry and nonnegativity constraints

Higher-order, complex and nonnegative decompositions Most BTD algorithms in Tensorlab are implemented for tensors of arbitrary order and for both real and complex decompositions and data. Nonnegative block term decompositions can be computed with algorithms that have the prefix `btdLx1nn_` (e.g., `btdLx1nn_nlsb` for bound-constrained nonlinear least squares). See Section 3.3 for details and examples.

Structured and (partially) symmetric decompositions It is possible to compute a structured or (partially) symmetric rank-$(L_r, L_r, 1)$ BTD with the structured CPD algorithms, which have the prefix `cpds_`, by noting that a rank-$(L_r, L_r, 1)$ BTD is nothing more than a CPD in which the third factor matrix $U^{(3)}$ has a Kronecker product structure. See Section 3.3 for details and examples.

6 Complex optimization

Optimization problems An integral part of Tensorlab comprises optimization of real functions in complex variables [13]. Tensorlab offers algorithms for complex optimization that solve unconstrained nonlinear optimization problems of the form

$$\underset{z \in \mathbb{C}^n}{\text{minimize}} \quad f(z, \overline{z}), \tag{minf}$$

where $f : \mathbb{C}^n \to \mathbb{R}$ is a smooth function (cf. `minf_lbfgs`, `minf_lbfgsdl` and `minf_ncg`) and nonlinear least squares problems of the form

$$\underset{z \in \mathbb{C}^n}{\text{minimize}} \quad \frac{1}{2}\|\mathcal{F}(z, \overline{z})\|_F^2, \tag{nls}$$

where $\|\cdot\|_F$ is the Frobenius norm and $\mathcal{F} : \mathbb{C}^n \to \mathbb{C}^{I_1 \times \cdots \times I_N}$ is a smooth function that maps n complex variables to $\prod I_n$ complex residuals (cf. `nls_gndl`, `nls_gncgs` and `nls_lm`). For nonlinear least squares problems, simple bound constraints of the form

$$\begin{aligned} \underset{z \in \mathbb{C}^n}{\text{minimize}} \quad & \frac{1}{2}\|\mathcal{F}(z, \overline{z})\|_F^2 \\ \text{subject to} \quad & \text{Re}\{l\} \leq \text{Re}\{z\} \leq \text{Re}\{u\} \\ & \text{Im}\{l\} \leq \text{Im}\{z\} \leq \text{Im}\{u\} \end{aligned} \tag{nlsb}$$

are also supported (cf. `nlsb_gndl`). Furthermore, when a real solution $z \in \mathbb{R}^n$ is sought, complex optimization reduces to real optimization and the algorithms are computationally equivalent to their real counterparts.

Prototypical example Throughout this section, we will use the Lyapunov equation

$$A \cdot X + X \cdot A^{\text{H}} + Q = 0,$$

which has important applications in control theory and model order reduction, as a prototypical example. In this matrix equation, the matrices $A, Q \in \mathbb{C}^{n \times n}$ are given and the objective is to compute the matrix $X \in \mathbb{C}^{n \times n}$. Since the equation is linear in X, there exist direct methods to compute X. However, these are relatively expensive, requiring $O(n^3)$ floating point operations (flop) to compute the solution. Instead, we will focus on a nonlinear extension of this equation to low-rank solutions X, which enables us to solve large-scale Lyapunov equations.

From here on, X is represented as the matrix product $U \cdot V$, where $U \in \mathbb{C}^{n \times k}$, $V \in \mathbb{C}^{k \times n}$ $(k < n)$. In the framework of (minf) and (nls), we define the objective function and residual function as

$$f_{\text{lyap}}(U, V) := \frac{1}{2}\|\mathcal{F}_{\text{lyap}}(U, V)\|_F^2 \tag{f-lyap}$$

and

$$\mathcal{F}_{\text{lyap}}(U, V) := A \cdot (U \cdot V) + (U \cdot V) \cdot A^{\text{H}} + Q, \tag{F-lyap}$$

respectively. Please note that this example serves mainly as an illustration and that computing a good low-rank solution to a Lyapunov equation proves to be quite difficult in practice due to an increasingly large amount of local minima as k increases[1].

[1] A technique that helps avoid many of these local minima is by first searching for the best rank-1 solution and using this solution as an initialization for the best rank-2 solution et cetera.

6.1 Complex derivatives

6.1.1 Pen & paper differentiation

Scalar functions To solve optimization problems of the form (minf), many algorithms require first-order derivatives of the real-valued objective function f. For a function of real variables $f_R : \mathbb{R}^n \to \mathbb{R}$, these derivatives can be captured in the gradient $\frac{\partial f_R}{\partial \boldsymbol{x}}$. For example, let

$$f_R(\boldsymbol{x}) := \sin(\boldsymbol{x}^\top \boldsymbol{x} + 2\boldsymbol{x}^\top \boldsymbol{a})$$

for $\boldsymbol{a}, \boldsymbol{x} \in \mathbb{R}^n$, then its gradient is given by

$$\frac{df_R}{d\boldsymbol{x}} = \cos(\boldsymbol{x}^\top \boldsymbol{x} + 2\boldsymbol{x}^\top \boldsymbol{a}) \cdot (2\boldsymbol{x} + 2\boldsymbol{a}).$$

Things get more interesting for real-valued functions of complex variables. Let

$$f(\boldsymbol{z}) := \sin(\overline{\boldsymbol{z}}^\top \boldsymbol{z} + (\overline{\boldsymbol{z}} + \boldsymbol{z})^\top \boldsymbol{a}),$$

where $\boldsymbol{z} \in \mathbb{C}^n$ and an overline denotes the complex conjugate of its argument. It is clear that $f(\boldsymbol{x}) \equiv f_R(\boldsymbol{x})$ for $\boldsymbol{x} \in \mathbb{R}^n$ and hence f is a generalization of f_R to the complex plane. Because f is now a function of both $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$, the limit $\lim_{\boldsymbol{h}\to 0} \frac{f(\boldsymbol{z}+\boldsymbol{h})-f(\boldsymbol{z})}{\boldsymbol{h}}$ no longer exists in general and so it would seem a complex gradient does not exist either. In fact, this only tells us the function f is not analytic in $\boldsymbol{z}$, i.e., its Taylor series in $\boldsymbol{z}$ alone does not exist. However, it can be shown that f is analytic in $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$ as a whole, meaning f has a Taylor series in the variables $\boldsymbol{z}_C := \begin{bmatrix}\boldsymbol{z}^\top & \overline{\boldsymbol{z}}^\top\end{bmatrix}^\top$ with a complex gradient

$$\frac{df}{d\boldsymbol{z}_C} = \begin{bmatrix} \dfrac{\partial f}{\partial \boldsymbol{z}} \\ \dfrac{\partial f}{\partial \overline{\boldsymbol{z}}} \end{bmatrix},$$

where $\frac{\partial f}{\partial \boldsymbol{z}}$ and $\frac{\partial f}{\partial \overline{\boldsymbol{z}}}$ are the cogradient and conjugate cogradient, respectively. The (conjugate) cogradient is a Wirtinger derivative and is to be interpreted as a partial derivative of f with respect to the variables $\boldsymbol{z}$ ($\overline{\boldsymbol{z}}$), while treating the variables $\overline{\boldsymbol{z}}$ ($\boldsymbol{z}$) as constant. For the example above, we have

$$\begin{aligned}\frac{\partial f}{\partial \boldsymbol{z}} &= \cos(\overline{\boldsymbol{z}}^\top \boldsymbol{z} + (\overline{\boldsymbol{z}} + \boldsymbol{z})^\top \boldsymbol{a}) \cdot (\overline{\boldsymbol{z}} + \boldsymbol{a}) \\ \frac{\partial f}{\partial \overline{\boldsymbol{z}}} &= \cos(\overline{\boldsymbol{z}}^\top \boldsymbol{z} + (\overline{\boldsymbol{z}} + \boldsymbol{z})^\top \boldsymbol{a}) \cdot (\boldsymbol{z} + \boldsymbol{a}).\end{aligned}$$

First, we notice that $\overline{\frac{\partial f}{\partial \boldsymbol{z}}} = \frac{\partial f}{\partial \overline{\boldsymbol{z}}}$, which holds for any real-valued function $f(\boldsymbol{z}, \overline{\boldsymbol{z}})$. A consequence is that any algorithm that optimizes f will only need one of the two cogradients, since the other is just its complex conjugate. Second, we notice that the cogradients evaluated in real variables $\boldsymbol{z} \in \mathbb{R}^n$ are equal to the real gradient $\frac{df_R}{d\boldsymbol{x}}$ up to a factor 2. Taking these two observations into account, the unconstrained nonlinear optimization algorithms in Tensorlab require only the *scaled conjugate cogradient*

$$\boldsymbol{g}(\boldsymbol{z}) := 2\frac{\partial f}{\partial \overline{\boldsymbol{z}}} \equiv 2\overline{\frac{\partial f}{\partial \boldsymbol{z}}}$$

and can optimize f over both $\boldsymbol{z} \in \mathbb{C}^n$ and $\boldsymbol{z} \in \mathbb{R}^n$.

Vector-valued functions To solve optimization problems of the form (nls), first-order derivatives of the vector-valued, or more generally tensor-valued, residual function $\mathcal{F}$ are often required. For a tensor-valued function $\mathcal{F} : \mathbb{R}^n \to \mathbb{R}^{I_1 \times \cdots \times I_N}$, these derivatives can be captured in the Jacobian $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{x}^\top}$. For example, let

$$\mathcal{F}_R(\boldsymbol{x}) := \begin{bmatrix} \sin(\boldsymbol{x}^\top \boldsymbol{x}) & \boldsymbol{x}^\top \boldsymbol{b} \\ \boldsymbol{x}^\top \boldsymbol{a} & \cos(\boldsymbol{x}^\top \boldsymbol{x}) \end{bmatrix}$$

for $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{x} \in \mathbb{R}^n$, then its Jacobian is given by

$$\frac{d \operatorname{vec}(\mathcal{F}_R)}{d\boldsymbol{x}^\top} = \begin{bmatrix} \cos(\boldsymbol{x}^\top \boldsymbol{x}) \cdot (2\boldsymbol{x}^\top) \\ \boldsymbol{a}^\top \\ \boldsymbol{b}^\top \\ -\sin(\boldsymbol{x}^\top \boldsymbol{x}) \cdot (2\boldsymbol{x}^\top) \end{bmatrix}.$$

But what happens when we allow $\mathcal{F} : \mathbb{C}^n \to \mathbb{C}^{I_1 \times \cdots \times I_N}$? For example,

$$\mathcal{F}(\boldsymbol{z}) := \begin{bmatrix} \sin(\boldsymbol{z}^\top \boldsymbol{z}) & \boldsymbol{z}^\top \boldsymbol{b} \\ \overline{\boldsymbol{z}}^\top \boldsymbol{a} & \cos(\overline{\boldsymbol{z}}^\top \boldsymbol{z}) \end{bmatrix},$$

where $\boldsymbol{z} \in \mathbb{C}^n$ could be a generalization of $\mathcal{F}_R$ to the complex plane. Following a similar reasoning as for scalar functions f, we can define a *Jacobian* and *conjugate Jacobian* as $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^\top}$ and $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \overline{\boldsymbol{z}}^\top}$, respectively. For the example above, we have

$$\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^\top} = \begin{bmatrix} \cos(\boldsymbol{z}^\top \boldsymbol{z}) \cdot (2\boldsymbol{z}^\top) \\ 0^\top \\ \boldsymbol{b}^\top \\ -\sin(\overline{\boldsymbol{z}}^\top \boldsymbol{z}) \cdot \overline{\boldsymbol{z}}^\top \end{bmatrix} \quad \text{and} \quad \frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \overline{\boldsymbol{z}}^\top} = \begin{bmatrix} 0^\top \\ \boldsymbol{a}^\top \\ 0^\top \\ -\sin(\overline{\boldsymbol{z}}^\top \boldsymbol{z}) \cdot \boldsymbol{z}^\top \end{bmatrix}.$$

Because $\mathcal{F}$ maps to the complex numbers, it is no longer true that the conjugate Jacobian is the complex conjugate of the Jacobian. In general, algorithms that solve (nls) require both the Jacobian and conjugate Jacobian. In some cases only one of the two Jacobians is required, e.g., when $\mathcal{F}$ is analytic in $\boldsymbol{z}$, which implies $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \overline{\boldsymbol{z}}^\top} \equiv 0$. Tensorlab offers nonlinear least squares solvers for both the general nonanalytic case and the latter analytic case.

6.1.2 Numerical differentiation

Real scalar functions (with the *i*-trick) The real gradient can be numerically approximated with `deriv` using the so-called *i*-trick [16]. For example, define the scalar functions

$$f_1(x) := \frac{10^{-20}}{1 - 10^3 x} \qquad f_2(\boldsymbol{x}) := \sin(\boldsymbol{x}^\top \boldsymbol{a})^3 \qquad f_3(X, Y) := \arctan(\operatorname{trace}(X^\top \cdot Y)),$$

where $x \in \mathbb{R}$, $\boldsymbol{a}, \boldsymbol{x} \in \mathbb{R}^n$ and $X, Y \in \mathbb{R}^{n \times n}$. Their first-order derivatives are

$$\frac{df_1}{dx} = \frac{10^{-17}}{(1 - 10^3 x)^2} \qquad \frac{df_2}{dx} = 3\sin(\boldsymbol{x}^\top \boldsymbol{a})^2 \cos(\boldsymbol{x}^\top \boldsymbol{a}) \cdot \boldsymbol{a} \qquad \begin{cases} \frac{\partial f_3}{\partial X} & = \frac{1}{1+\operatorname{trace}(X^\top \cdot Y)^2} \cdot Y \\ \frac{\partial f_3}{\partial Y} & = \frac{1}{1+\operatorname{trace}(X^\top \cdot Y)^2} \cdot X \end{cases}.$$

An advantage of using the *i*-trick is that it can compute first-order derivatives up to the order of machine precision. The disadvantages are that this requires an equivalent of about 4 (real) function evaluations per variable (compared to 2 for finite differences) and that certain requirements must be met. First, only the real gradient can be computed, meaning the gradient can only be computed where the variables are real. Second, the function must be real-valued when evaluated in real variables. Third, the function must be analytic on the complex plane. In other words, the function may not be a function of the complex conjugate of its argument. For example, the *i*-trick can be used to compute the gradient of the function `@(x)x.'*x`, but not of the function `@(x)x'*x` because the latter depends on $\overline{\boldsymbol{x}}$. As a last example, note that `@(x)real(x)` is not analytic in $x \in \mathbb{C}$ because it can be written as `@(x)(x+conj(x))/2`.

Choosing $\boldsymbol{a}$ as `ones(size(x))` in the example functions above, the following example uses `deriv` to compute the real gradient of these functions using the i-trick:

```
% Three test functions.
f1 = @(x)1e-20/(1-1e3*x);
f2 = @(x)sin(x.'*ones(size(x)))^3;
f3 = @(XY)atan(trace(XY{1}.'*XY{2}));

% Their first-order derivatives.
g1 = @(x)1e-17/(1-1e3*x)^2;
g2 = @(x)3*sin(x.'*ones(size(x)))^2*cos(x.'*ones(size(x)))*ones(size(x));
g3 = @(XY){1/(1+trace(XY{1}.'*XY{2})^2)*XY{2},1/(1+trace(XY{1}.'*XY{2})^2)*XY{1}};

% Approximate the real gradient with the i-trick and compute its relative error.
x = randn;
relerr1 = abs(g1(x)-deriv(f1,x))/abs(g1(x))
x = randn(10,1);
relerr2 = norm(g2(x)-deriv(f2,x))/norm(g2(x))
XY = {randn(10),randn(10)};
relerr3 = cellfun(@(a,b)frob(a-b)/frob(a),g3(XY),deriv(f3,XY))
```

In Tensorlab, derivatives of scalar functions are returned in the same format as the function's argument. Notice that `f3` is function of a cell array `XY`, containing the matrix X in `XY{1}` and the matrix Y in `XY{2}`. In similar vein, the output of `deriv(f3,XY)` is a cell array containing the matrices $\frac{\partial f_3}{\partial X}$ and $\frac{\partial f_3}{\partial Y}$. Oftentimes, this allows the user to conveniently write functions as a function of a cell array of variables (containing vectors, matrices or tensors) instead of coercing all variables into one long vector which must then be disassembled in the respective variables.

Scalar functions (with finite differences) If the conditions for the i-trick are not satisfied, or if a scaled conjugate cogradient is required, an alternative is using finite differences to approximate first-order derivatives. In both cases, the finite difference approximation can be computed using `deriv(f,x,[],'gradient')`. As a first example, we compute the relative error of the finite difference approximation of the real gradient of f_1, f_2 and f_3:

```
% Approximate the real gradient with finite differences and compute its relative error.
x = randn;
relerr1 = abs(g1(x)-deriv(f1,x,[],'gradient'))/abs(g1(x))
x = randn(10,1);
relerr2 = norm(g2(x)-deriv(f2,x,[],'gradient'))/norm(g2(x))
XY = {randn(10),randn(10)};
relerr3 = cellfun(@(a,b)frob(a-b)/frob(a),g3(XY),deriv(f3,XY,[],'gradient'))
```

If $f(\boldsymbol{z})$ is a real-valued scalar function of complex variables, `deriv` can compute its scaled conjugate cogradient $\boldsymbol{g}(\boldsymbol{z})$. For example, let

$$f(\boldsymbol{z}) := \boldsymbol{a}^{\mathsf{T}}(\boldsymbol{z}+\overline{\boldsymbol{z}}) + \log(\boldsymbol{z}^{\mathsf{H}}\boldsymbol{z}) \qquad \boldsymbol{g}(\boldsymbol{z}) := 2\frac{\partial f}{\partial \overline{\boldsymbol{z}}} \equiv 2\overline{\frac{\partial f}{\partial \boldsymbol{z}}} = 2\cdot\boldsymbol{a} + \frac{2}{\boldsymbol{z}^{\mathsf{H}}\boldsymbol{z}}\cdot\boldsymbol{z},$$

where $\boldsymbol{a} \in \mathbb{R}^n$ and $\boldsymbol{z} \in \mathbb{C}^n$. Since f is real-valued and $\boldsymbol{z}$ is complex, calling `deriv(f,z)` is equivalent to `deriv(f,z,[],'gradient')` and uses finite differences to approximate the scaled conjugate cogradient. In the following example $\boldsymbol{a}$ is chosen as `ones(size(z))` and the relative error of the finite difference approximation of the scaled conjugate cogradient $\boldsymbol{g}(\boldsymbol{z})$ is computed:

```
% Approximate the scaled conjugate cogradient with finite differences
% and compute its relative error.
f = @(z)ones(size(z)).'*(z+conj(z))+log(z'*z);
```

```
g = @(z)2*ones(size(z))+2/(z'*z)*z;
z = randn(10,1)+randn(10,1)*1i;
relerr = norm(g(z)-deriv(f,z))/norm(g(z))
```

Note that it is, in general, safer to use `deriv(f,z,[],'gradient')` to compute the scaled conjugate cogradient, as `deriv(f,z)` will attempt to use the i-trick when `z` is real.

Real vector-valued functions (with the i-trick) Analogously to scalar functions, the real Jacobian of tensor-valued functions can also be numerically approximated with `deriv` using the i-trick. Take for example the following matrix-valued function

$$\mathcal{F}_1(\boldsymbol{x}) := \begin{bmatrix} \log(\boldsymbol{x}^\mathsf{T}\boldsymbol{x}) & 0 \\ 2\boldsymbol{a}^\mathsf{T}\boldsymbol{x} & \sin(\boldsymbol{x}^\mathsf{T}\boldsymbol{x}) \end{bmatrix},$$

where $\boldsymbol{a}, \boldsymbol{x} \in \mathbb{R}^n$, and its real Jacobian

$$J_1(\boldsymbol{x}) := \frac{d\,\mathrm{vec}(\mathcal{F}_1)}{d\boldsymbol{x}^\mathsf{T}} = 2\begin{bmatrix} \frac{1}{\boldsymbol{x}^\mathsf{T}\boldsymbol{x}} \cdot \boldsymbol{x}^\mathsf{T} \\ \boldsymbol{a}^\mathsf{T} \\ 0 \\ \cos(\boldsymbol{x}^\mathsf{T}\boldsymbol{x}) \cdot \boldsymbol{x}^\mathsf{T} \end{bmatrix}.$$

Set $\boldsymbol{a}$ equal to `ones(size(x))`. Since each entry in $\mathcal{F}_1(\boldsymbol{x})$ is real-valued and not a function of $\overline{\boldsymbol{x}}$, we can approximate the real Jacobian $J_1(\boldsymbol{x})$ with the i-trick:

```
% Approximate the Jacobian of a tensor-valued function with the i-trick
% and compute its relative error.
F1 = @(x)[log(x.'*x) 0; 2*ones(size(x)).'*x sin(x.'*x)];
J1 = @(x)2*[1/(x.'*x)*x.'; ones(size(x)).'; zeros(size(x)).'; cos(x.'*x)*x.'];
x = randn(10,1);
relerr1 = frob(J1(x)-deriv(F1,x))/frob(J1(x))
```

Analytic vector-valued functions (with finite differences) The Jacobian of an analytic tensor-valued function $\mathcal{F}(\boldsymbol{z})$ can be approximated with finite differences by calling `deriv(F,z,[],'Jacobian')`. Functions that are analytic when their argument is real, may no longer be analytic when their argument is complex. For example, $\mathcal{F}(\boldsymbol{z}) := \begin{bmatrix} \boldsymbol{z}^\mathsf{H}\boldsymbol{z} & \mathrm{Re}\{\boldsymbol{z}\}^\mathsf{T}\boldsymbol{z} \end{bmatrix}$ is not analytic in $\boldsymbol{z} \in \mathbb{C}^n$ because it depends on $\overline{\boldsymbol{z}}$, but is analytic when $\boldsymbol{z} \in \mathbb{R}^n$. An example of a function that is analytic for both real and complex $\boldsymbol{z}$ is the function $\mathcal{F}_1(\boldsymbol{z})$. The following two examples compute the relative error of the finite differences approximation of the Jacobian $J_1(\boldsymbol{x})$ in a real vector $\boldsymbol{x}$:

```
% Approximate the Jacobian of an analytic tensor-valued function
% with finite differences and compute its relative error.
x = randn(10,1);
relerr1 = frob(J1(x)-deriv(F1,x,[],'Jacobian'))/frob(J1(x))
```

and the relative error of the finite differences approximation of $J_1(\boldsymbol{z})$ in a complex vector $\boldsymbol{z}$:

```
% Approximate the Jacobian of an analytic tensor-valued function
% with finite differences and compute its relative error.
z = randn(10,1)+randn(10,1)*1i;
relerr1 = frob(J1(z)-deriv(F1,z,[],'Jacobian'))/frob(J1(z))
```

Nonanalytic vector-valued functions (with finite differences) In general, a tensor-valued function may be function of its argument and its complex conjugate. The matrix-valued function

$$\mathcal{F}_2(X, Y) := \begin{bmatrix} \log(\mathrm{trace}(X^\mathsf{H} \cdot Y)) & 0 \\ \boldsymbol{a}^\mathsf{T}(X + \overline{X})\boldsymbol{a} & \boldsymbol{a}^\mathsf{T}(Y - \overline{Y})\boldsymbol{a} \end{bmatrix},$$

where $\boldsymbol{a} \in \mathbb{R}^n$ and $X, Y \in \mathbb{C}^{n\times n}$ is an example of such a nonanalytic function because it depends on X, Y and $\overline{X}$, $\overline{Y}$. Its Jacobian and conjugate Jacobian are given by

$$J_2(X,Y) := \frac{\partial \operatorname{vec}(\mathcal{F}_2)}{\partial \left[\operatorname{vec}(X)^\mathsf{T} \quad \operatorname{vec}(Y)^\mathsf{T}\right]} = \begin{bmatrix} 0 & \frac{1}{\operatorname{trace}(X^\mathsf{H}\cdot Y)} \cdot \operatorname{vec}(\overline{X})^\mathsf{T} \\ (\boldsymbol{a}\otimes\boldsymbol{a})^\mathsf{T} & 0 \\ 0 & 0 \\ 0 & (\boldsymbol{a}\otimes\boldsymbol{a})^\mathsf{T} \end{bmatrix}$$

$$J_2^c(X,Y) := \frac{\partial \operatorname{vec}(\mathcal{F}_2)}{\partial \left[\operatorname{vec}(\overline{X})^\mathsf{T} \quad \operatorname{vec}(\overline{Y})^\mathsf{T}\right]} = \begin{bmatrix} \frac{1}{\operatorname{trace}(X^\mathsf{H}\cdot Y)} \cdot \operatorname{vec}(Y)^\mathsf{T} & 0 \\ (\boldsymbol{a}\otimes\boldsymbol{a})^\mathsf{T} & 0 \\ 0 & 0 \\ 0 & -(\boldsymbol{a}\otimes\boldsymbol{a})^\mathsf{T} \end{bmatrix},$$

respectively. For a nonanalytic tensor-valued function $\mathcal{F}(\boldsymbol{z})$, `deriv(F,z,[],'Jacobian-C')` computes a finite differences approximation of the *complex Jacobian*

$$\begin{bmatrix} \frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^\mathsf{T}} & \frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \overline{\boldsymbol{z}}^\mathsf{T}} \end{bmatrix},$$

comprising both the Jacobian and conjugate Jacobian. The complex Jacobian of $\mathcal{F}_2(X,Y)$ is the matrix $\left[J_2(X,Y) \quad J_2^c(X,Y)\right]$. In the following example $\boldsymbol{a}$ is equal to `ones(length(z{1}))` and the relative error of the complex Jacobian's finite differences approximation is computed:

```
% Approximate the complex Jacobian of a nonanalytic tensor-valued function
% with finite differences and compute its relative error.
F2 = @(z)[log(trace(z{1}'*z{2})) 0; ...
          sum(sum(z{1}+conj(z{1}))) sum(sum(z{2}-conj(z{2})))];
J2 = @(z)[zeros(1,numel(z{1})) 1/trace(z{1}'*z{2})*reshape(conj(z{1}),1,[]) ...
          1/trace(z{1}'*z{2})*reshape(z{2},1,[]) zeros(1,numel(z{2})); ...
          ones(1,numel(z{1})) zeros(1,numel(z{1})) ...
          ones(1,numel(z{2})) zeros(1,numel(z{2})); ...
          zeros(1,2*numel(z{1})) zeros(1,2*numel(z{2})); ...
          zeros(1,numel(z{1})) ones(1,numel(z{1})) ...
          zeros(1,numel(z{1})) -ones(1,numel(z{1}))];
z = {randn(10)+randn(10)*1i,randn(10)+randn(10)*1i};
relerr2 = frob(J2(z)-deriv(F2,z,[],'Jacobian-C'))/frob(J2(z))
```

6.2 Nonlinear least squares

Algorithms Tensorlab offers three algorithms for unconstrained nonlinear least squares: `nls_gndl`, `nls_gncgs` and `nls_lm`. The first is Gauss–Newton with a dogleg trust region strategy, the second is Gauss–Newton with CG-Steihaug for solving the trust region subproblem and the last is Levenberg–Marquardt. A bound-constrained method, `nlsb_gndl`, is also included and is a projected Gauss–Newton method with dogleg trust region. All algorithms are applicable to both analytic and nonanalytic residual functions and offer various ways of exploiting the structure available in its (complex) Jacobian.

With numerical differentiation The complex optimization algorithms that solve (nls) require the Jacobian of the residual function $\mathcal{F}(\boldsymbol{z}, \overline{\boldsymbol{z}})$, which will be $\mathcal{F}_{\text{lyap}}(U, V)$ for the remainder of this section (cf. Section 6). The second argument `dF` of the nonlinear least squares optimization algorithms `nls_gndl`, `nls_gncgs` and `nls_lm` specifies how the Jacobian should be computed. To approximate the Jacobian with finite differences, set `dF` equal to `'Jacobian'` or `'Jacobian-C'`.

The first case, `'Jacobian'`, corresponds to approximating the Jacobian $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^\mathsf{T}}$, assuming $\mathcal{F}$ is analytic in $\boldsymbol{z}$. The second case, `'Jacobian-C'`, corresponds to approximating the complex Jacobian consisting of the Jacobian $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^{\mathsf{T}}}$ and conjugate Jacobian $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \overline{\boldsymbol{z}}^{\mathsf{T}}}$, where $\boldsymbol{z} \in \mathbb{C}^n$. Since $\mathcal{F}_{\text{lyap}}$ does not depend on $\overline{U}$ or $\overline{V}$, we may implement a nonlinear least squares solver for the low-rank solution of the Lyapunov equation as

```
function z = lyap_nls_deriv(A,Q,z0)

F = @(z)(A*z{1})*z{2}+z{1}*(z{2}*A')+Q;
z = nls_gndl(F,'Jacobian',z0);

end
```

The residual function `F(z)` is matrix-valued and its argument is a cell array `z`, consisting of the two matrices U and V. The output of the optimization algorithm, in this case `nls_gndl`, is a cell array of the same format as the argument of the residual function `F(z)`.

With the Jacobian Using the property $\operatorname{vec}(A \cdot X \cdot B) \equiv (B^{\mathsf{T}} \otimes A) \cdot \operatorname{vec}(X)$, it is easy to verify that the Jacobian of $\mathcal{F}_{\text{lyap}}(U, V)$ is given by

$$\frac{\partial \operatorname{vec}(\mathcal{F}_{\text{lyap}})}{\partial \left[\operatorname{vec}(U)^{\mathsf{T}} \quad \operatorname{vec}(V)^{\mathsf{T}}\right]} = \left[(V^{\mathsf{T}} \otimes A) + (\overline{A} \cdot V^{\mathsf{T}} \otimes \mathbb{I}) \quad (\mathbb{I} \otimes A \cdot U) + (\overline{A} \otimes U)\right]. \tag{J-lyap}$$

To supply the Jacobian to the optimization algorithm, set the field `dF.dz` as the function handle of the function that computes the Jacobian at a given point $\boldsymbol{z}$. For problems for which the residual function $\mathcal{F}$ depends on both $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$, the complex Jacobian can be supplied with the field `dF.dzc`. See Section 6.1 or the `help` page of the selected algorithm for more information. Applied to the Lyapunov equation, we have

```
function z = lyap_nls_J(A,Q,z0)

dF.dz = @J;
z = nls_gndl(@F,dF,z0);

    function F = F(z)
        U = z{1}; V = z{2};
        F = (A*U)*V+U*(V*A')+Q;
    end

    function J = J(z)
        U = z{1}; V = z{2}; I = eye(size(A));
        J = [kron(V.',A)+kron(conj(A)*V.',I) kron(I,A*U)+kron(conj(A),U)];
    end

end
```

With the Jacobian's Gramian When the residual function $\mathcal{F} : \mathbb{C}^n \to \mathbb{C}^{I_1 \times \cdots \times I_N}$ is analytic[2] in $\boldsymbol{z}$ (i.e., it is not a function of $\overline{\boldsymbol{z}}$) and the number of residuals $\prod I_n$ is large compared to the number of variables n, it may be beneficial to compute the Jacobian's Gramian $J^{\mathsf{H}} J$ instead of the Jacobian $J := \frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^{\mathsf{T}}}$ itself. This way, each iteration of the nonlinear least squares algorithm no longer requires computing the (pseudo-)inverse $J^{\dagger}$, but rather the less expensive (pseudo-)inverse $(J^{\mathsf{H}} J)^{\dagger}$. In the case of the Lyapunov equation, this can lead to a significantly more efficient method if the rank k of the solution is small with respect to its order n. Along with the Jacobian's Gramian, the objective function $f := \frac{1}{2}\|\mathcal{F}\|_F^2$ and its gradient $\frac{\partial f}{\partial \overline{\boldsymbol{z}}} \equiv J^{\mathsf{H}} \cdot \operatorname{vec}(\mathcal{F})$ are also necessary. Skipping the derivation of the gradient and Jacobian's Gramian, the implementation could look like

[2]In the more general case of a nonanalytic residual function, the structure in its complex Jacobian can be exploited by computing an inexact step. See the following paragraph for more details.

```
function z = lyap_nls_JHJ(A,Q,z0)

AHA = A'*A;
dF.JHJ = @JHJ;
dF.JHF = @grad;
z = nls_gndl(@f,dF,z0);

    function f = f(z)
        U = z{1}; V = z{2};
        F = (A*U)*V+U*(V*A')+Q;
        f = 0.5*(F(:)'*F(:));
    end

    function g = grad(z)
        U = z{1}; V = z{2};
        gU = AHA*(U*(V*V'))+A'*(U*(V*A'*V'))+A'*(Q*V')+ ...
             A*(U*(V*A*V'))+U*(V*AHA*V')+Q*(A*V');
        gV = (U'*AHA*U)*V+((U'*A'*U)*V)*A'+(U'*A')*Q+ ...
             ((U'*A*U)*V)*A+((U'*U)*V)*AHA+(U'*Q)*A;
        g = {gU,gV};
    end

    function JHJ = JHJ(z)
        U = z{1}; V = z{2}; I = eye(size(A));
        tmpa = kron(conj(V*A'*V'),A); tmpb = kron(conj(A),U'*A'*U);
        JHJ11 = kron(conj(V*V'),AHA)+kron(conj(V*AHA*V'),I)+tmpa+tmpa';
        JHJ22 = kron(I,U'*AHA*U)+kron(conj(AHA),U'*U)+tmpb+tmpb';
        JHJ12 = kron(conj(V),AHA*U)+kron(conj(V*A),A'*U)+ ...
                kron(conj(V*A'),A*U)+kron(conj(V*AHA),U);
        JHJ = [JHJ11 JHJ12; JHJ12' JHJ22];
    end

end
```

By default, the algorithm `nls_gndl` uses the Moore–Penrose pseudo-inverse of either J or $J^{\mathsf{H}}J$ to compute a new descent direction. However, if it is known that these matrices always have full rank, a more efficient least squares inverse can be computed. To do so, use

```
% Compute a more efficient least squares step instead of using the pseudoinverse.
options.JHasFullRank = true;
z = nls_gndl(@f,dF,z0,options);
```

The other nonlinear least squares algorithms `nls_gncgs` and `nls_lm` use a different approach for calculating the descent direction and do not have such an option.

With an inexact step The most computationally intensive part of most nonlinear least squares problems is computing the next descent direction, which involves inverting either the Jacobian $J := \frac{\partial \operatorname{vec}(\mathcal{F})}{\partial z^{\mathsf{T}}}$ or its Gramian $J^{\mathsf{H}}J$ in the case of an analytic residual function. With iterative solvers such as preconditioned conjugate gradient (PCG) and LSQR, the descent direction can be approximated using only matrix-vector products. The resulting descent directions are said to be inexact. Many problems exhibit some structure in the Jacobian which can be exploited in its matrix-vector product, allowing for an efficient computation of an inexact step. Concretely, the user has the choice of supplying the matrix vector products $J \cdot \boldsymbol{x}$ and $J^{\mathsf{H}} \cdot \boldsymbol{y}$, or the single matrix-vector product $(J^{\mathsf{H}}J) \cdot \boldsymbol{x}$. An implementation of an inexact nonlinear least squares solver using the former method can be

```
function z = lyap_nls_Jx(A,Q,z0)
```

```
dF.dzx = @Jx;
z = nls_gndl(@F,dF,z0);

    function F = F(z)
        U = z{1}; V = z{2};
        F = (A*U)*V+U*(V*A')+Q;
    end

    function b = Jx(z,x,transp)
        U = z{1}; V = z{2};
        switch transp
            case 'notransp' % b = J*x
                Xu = reshape(x(1:numel(U)),size(U));
                Xv = reshape(x(numel(U)+1:end),size(V));
                b = (A*Xu)*V+Xu*(V*A')+(A*U)*Xv+U*(Xv*A');
                b = b(:);
            case 'transp'   % b = J'*x
                X = reshape(x,size(A));
                Bu = A'*(X*V')+X*(A*V');
                Bv = (U'*A')*X+(U'*X)*A;
                b = [Bu(:); Bv(:)];
        end
    end

end
```

where the Kronecker structure of the Jacobian (J-lyap) is exploited by reducing the computations to matrix-matrix products. Under suitable conditions on A and Q, this implementation can achieve a complexity of $O(nk^2)$, where n is the order of the solution $X = U \cdot V$ and k is its rank.

In the case of a nonanalytic residual function $\mathcal{F}(\boldsymbol{z}, \overline{\boldsymbol{z}})$, computing an inexact step requires matrix-vector products $J \cdot \boldsymbol{x}$, $J^{\mathsf{H}} \cdot \boldsymbol{y}$, $J_c \cdot \boldsymbol{x}$ and $J_c^{\mathsf{H}} \cdot \boldsymbol{y}$, where $J := \frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^{\mathsf{T}}}$ and $J_c := \frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \overline{\boldsymbol{z}}^{\mathsf{T}}}$ are the residual function's Jacobian and conjugate Jacobian, respectively. For more information on how to implement these matrix-vector products, read the `help` information of the desired (nls) solver.

Setting the options The parameters of the selected optimization algorithm can be set by supplying the method with an options structure, e.g.,

```
% Set algorithm tolerances.
options.TolFun = 1e-12;
options.TolX = 1e-6;
options.MaxIter = 100;
dF.dz = @J;
z = nls_gndl(@F,dF,z0,options);
```

It is important to note that since the objective function is the square of a residual norm, the objective function tolerance `options.TolFun` can be set as small as 10^{-32} for a given machine epsilon of 10^{-16}. See the `help` information on the selected algorithm for more details.

Viewing the algorithm output The second output of the optimization algorithms returns additional information pertaining to the algorithm. For example, the algorithms keep track of the objective function value in `output.fval` and also output the circumstances under which the algorithm terminated in `output.info`. As an example, the norm of the residual function of each iterate can be plotted with

```
% Plot each iterate's objective function value.
dF.dz = @J;
```

```
[z,output] = nls_gndl(@F,dF,z0);
semilogy(0:output.iterations,sqrt(2*output.fval));
```

Since the objective function is $\frac{1}{2}\|\mathcal{F}\|_F^2$, we plot `sqrt(2*output.fval)` to see the norm $\|\mathcal{F}\|_F$. See the `help` information on the selected algorithm for more details.

6.3 Unconstrained nonlinear optimization

Algorithms Tensorlab offers three algorithms for unconstrained complex optimization: `minf_lbfgs`, `minf_lbfgsdl` and `minf_ncg`. The first two are limited-memory BFGS with Moré–Thuente line search and dogleg trust region, respectively, and the last is nonlinear conjugate gradient with Moré–Thuente line search. In stead of the supplied Moré–Thuente line search, the user may optionally supply a custom line search method. See the `help` information for details.

With numerical differentiation The complex optimization algorithms that solve (minf) require the (scaled conjugate co-)gradient of the objective function $f(z,\overline{z})$, which will be $f_{\text{lyap}}(z,\overline{z})$ for the remainder of this section (cf. Section 6). The second argument of the unconstrained nonlinear minimization algorithms `minf_lbfgs`, `minf_lbfgsdl` and `minf_ncg` specifies how the gradient should be computed. To approximate the (scaled conjugate co-)gradient with finite differences, set the second argument equal to the empty matrix `[]`. An implementation for the Lyapunov equation could look like

```
function z = lyap_minf_deriv(A,Q,z0)

f = @(z)frob((A*z{1})*z{2}+z{1}*(z{2}*A')+Q);
z = minf_lbfgs(f,[],z0);

end
```

As with the nonlinear least squares algorithms, the argument of the objective function is a cell array `z`, consisting of the two matrices U and V. Likewise, the output of the optimization algorithm, in this case `minf_lbfgs`, is a cell array of the same format as the argument of the objective function `f(z)`.

With the gradient If an expression for the (scaled conjugate co-)gradient is available, it can be supplied to the optimization algorithm in the second argument. For the Lyapunov equation, the implementation could look like

```
function z = lyap_minf_grad(A,Q,z0)

AHA = A'*A;
z = minf_lbfgs(@f,@grad,z0);

    function f = f(z)
        U = z{1}; V = z{2};
        F = (A*U)*V+U*(V*A')+Q;
        f = 0.5*(F(:)'*F(:));
    end

    function g = grad(z)
        U = z{1}; V = z{2};
        gU = AHA*(U*(V*V'))+A'*(U*(V*A'*V'))+A'*(Q*V')+ ...
             A*(U*(V*A*V'))+U*(V*AHA*V')+Q*(A*V');
        gV = (U'*AHA*U)*V+((U'*A'*U)*V)*A'+(U'*A')*Q+ ...
             ((U'*A*U)*V)*A+((U'*U)*V)*AHA+(U'*Q)*A;
        g = {gU,gV};
    end

end
```

The function `grad(z)` computes the scaled conjugate cogradient $2\frac{\partial f_{\text{lyap}}}{\partial \overline{z}}$, which coincides with the real gradient for $z \in \mathbb{R}^n$. See Section 6.1 for more information on complex derivatives.

Note that the gradient `grad(z)` is returned in the same format as the solution `z`, i.e., as a cell array containing matrices of the same size as U and V. However, the gradient may also be returned as a vector if this is more convenient for the user. In that case, the scaled conjugate cogradient should be of the format $2\frac{\partial f_{\text{lyap}}}{\partial \overline{z}}$ where $z := \begin{bmatrix}\text{vec}(U)^{\mathsf{T}} & \text{vec}(V)^{\mathsf{T}}\end{bmatrix}^{\mathsf{T}}$.

Setting the options The parameters of the selected optimization algorithm can be set by supplying the method with an options structure, e.g.,

```
% Set algorithm tolerances.
options.TolFun = 1e-6;
options.TolX = 1e-6;
options.MaxIter = 100;
z = minf_lbfgs(@f,@grad,z0,options);
```

See the `help` information on the selected algorithm for more details.

Viewing the algorithm output The second output of the optimization algorithms returns additional information pertaining to the algorithm. For example, the algorithms keep track of the objective function value in `output.fval` and also output the circumstances under which the algorithm terminated in `output.info`. As an example, the objective function value of each iterate can be plotted with

```
% Plot each iterate's objective function value.
[z,output] = minf_lbfgs(@f,@grad,z0);
semilogy(0:output.iterations,output.fval);
```

See the `help` information on the selected algorithm for more details.

7 Global minimization of bivariate polynomials and rational functions

Analytic bivariate polynomials Consider the problem of minimizing a bivariate polynomial

$$\underset{x,y\in\mathbb{R}}{\text{minimize}} \quad p(x,y), \tag{bipol}$$

or more generally, a rational function

$$\underset{x,y\in\mathbb{R}}{\text{minimize}} \quad \frac{p(x,y)}{q(x,y)}. \tag{birat}$$

Since all local minimizers (x^*, y^*) are stationary points, they are roots of the system of bivariate polynomials

$$\begin{aligned}\frac{\partial p}{\partial x}q - p\frac{\partial q}{\partial x} &= 0\\ \frac{\partial p}{\partial y}q - p\frac{\partial q}{\partial y} &= 0,\end{aligned}$$

where $q(x,y) \equiv 1$ in the case of minimizing a bivariate polynomial. With `polysol2`, Tensorlab offers a numerically robust way of computing isolated real roots of a system of bivariate polynomials

$$\begin{aligned} f(x,y) &= 0 \\ g(x,y) &= 0 \end{aligned} \tag{bisys}$$

as the eigenvalues of a generalized eigenvalue problem [12]. Stationary points of bivariate polynomials and rational functions may be computed with `polymin2` and `ratmin2`, respectively.

Polyanalytic univariate polynomials Closely related is the problem of minimizing a polyanalytic univariate polynomial

$$\underset{z \in \mathbb{C}}{\text{minimize}} \quad p(z,\overline{z}), \tag{unipol}$$

or more generally, a rational function

$$\underset{z \in \mathbb{C}}{\text{minimize}} \quad \frac{p(z,\overline{z})}{q(z,\overline{z})}. \tag{unirat}$$

Analogously to the analytic bivariate case, all local minimizers are roots of the system

$$\begin{aligned} \frac{\partial p}{\partial z} q - p \frac{\partial q}{\partial z} &= 0 \\ \frac{\partial p}{\partial \overline{z}} q - p \frac{\partial q}{\partial \overline{z}} &= 0, \end{aligned}$$

where the derivatives are Wirtinger derivatives (cf. Section 6.1). The method `polysol2` can also solve systems of polyanalytic polynomials

$$\begin{aligned} f(z,\overline{z}) &= 0 \\ g(z,\overline{z}) &= 0. \end{aligned} \tag{unisys}$$

In fact, given a system of bivariate polynomials (bisys), `polysol2` will first convert it to the form (unisys) before computing the roots as the eigenvalues of a (complex) generalized eigenvalue problem. Stationary points of real-valued polyanalytic polynomials and rational functions may be computed with `polymin2` and `ratmin2`, respectively.

7.1 Stationary points of polynomials and rational functions

Polynomials and rational functions In MATLAB, a polynomial $p(x)$ is represented by a row vector `p = [ad ... a2 a1 a0]` as

$$p(x) = \begin{bmatrix} a_d & \cdots & a_2 & a_1 & a_0 \end{bmatrix} \cdot \begin{bmatrix} x^d & \cdots & x^2 & x & 1 \end{bmatrix}^{\mathsf{T}}.$$

For example, the polynomial $p(x) = x^3 + 2x^2 + 3x + 4$ is represented by the row vector `p = [1 2 3 4]`. Its derivative $\frac{dp}{dx}$ can be computed with `polyder(p)` and its zeros can be computed with `roots(p)`.

The stationary points of $p(x)$, i.e., all x^* which satisfy $\frac{dp}{dx}(x^*) = 0$, are the output of `roots(polyder(p))`. However, some of these solutions may have a small imaginary part which correspond to a numerical zero. The stationary points can also be computed as roots of the polynomial's derivative with `polymin(p)`, which deals with solutions with small imaginary part and returns only real solutions.

Analogously, the stationary points of a a rational function $\frac{p(x)}{q(x)}$ are given by

```
roots(conv(polyder(p),q)-conv(p,polyder(q)))
```

where `conv(p,q)` is the convolution of the two row vectors `p` and `q` and is equivalent to computing the coefficients of the polynomial $p(x) \cdot q(x)$. As in the polynomial case, there are a few numerical issues which can be dealt with by computing the stationary points of $\frac{p(x)}{q(x)}$ with `ratmin(p,q)`.

Bivariate polynomials and rational functions In Tensorlab, a bivariate polynomial $p(x, y)$ is represented by a matrix `p` as

$$p(x,y) = \begin{bmatrix} 1 & y & \cdots & y^{d_y} \end{bmatrix} \cdot \begin{bmatrix} a_{00} & \cdots & a_{0d_x} \\ \vdots & \ddots & \vdots \\ a_{d_y0} & \cdots & a_{d_yd_x} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x \\ \vdots \\ x^{d_x} \end{bmatrix}.$$

For example, the six-hump camel back function [7]

$$p(x,y) = 4x^2 - 2.1x^4 + \frac{1}{3}x^6 + xy - 4y^2 + 4y^4$$

is represented by the matrix

```
p = [ 0    0    4    0   -2.1 0    1/3; ...
      0    1    0    0    0    0    0;   ...
     -4    0    0    0    0    0    0;   ...
      0    0    0    0    0    0    0;   ...
      4    0    0    0    0    0    0];
```

and can be seen in Figure 7.1. The stationary points of the polynomial $p(x, y)$ can be computed as the solutions of the system $\frac{\partial p}{\partial x} = \frac{\partial p}{\partial y} = 0$ with `polymin2(p)`. To obtain a global minimum, select the solution with smallest function value using something along the lines of

```
[xy,v] = polymin2(p);
[vmin,idx] = min(v);
xymin = xy(idx,:);
```

To visualize the level zero contour lines of $\frac{\partial p}{\partial x}$ and $\frac{\partial p}{\partial y}$ in the neighbourhood of the stationary points, set `options.Plot = true` as follows

```
p = randn(6); % Generate random bivariate polynomial of coordinate degree 5.
options.Plot = true;
xy = polymin2(p,options);
```

or inline as `polymin2(randn(6),struct('Plot',true))`.

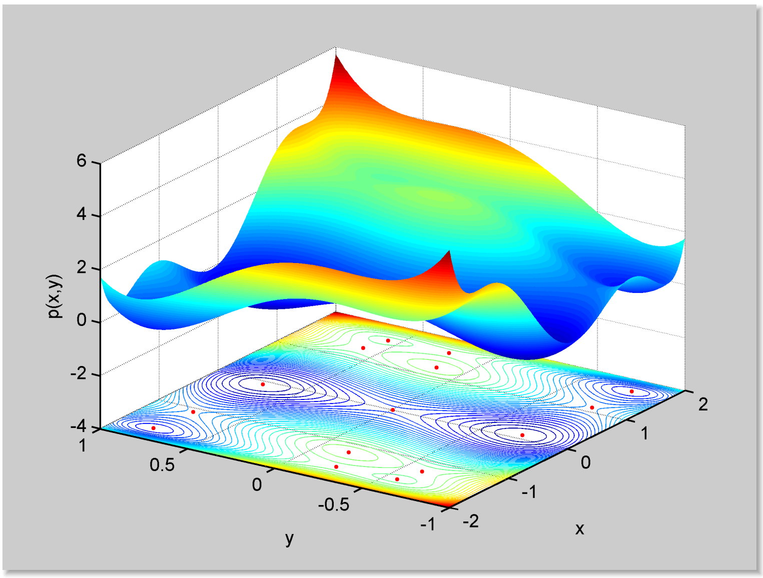


Figure 7.1: The six-hump camel back function and its stationary points as red dots.

The corresponding system of polynomials is solved by `polysol2`. The latter includes several balancing steps to improve the accuracy of the solution. For some poorly scaled problems, the method may fail to find all solutions of the system. In that case, try decreasing the `polysol2` balancing tolerance `options.TolBal` to a smaller value, e.g.,

```
options.TolBal = 1e-4;
xy = polymin2(p,options);
```

Computing the stationary points of a bivariate rational function $\frac{p(x,y)}{q(x,y)}$ is completely analogous to the polynomial case. For example,

```
p = randn(6); % Random bivariate polynomial of coordinate degree 5.
q = randn(4); % Random bivariate polynomial of coordinate degree 3.
options.Plot = true;
xy = ratmin2(p,q,options);
```

or its inline equivalent `ratmin2(randn(6),randn(4),struct('Plot',true))`.

Polyanalytic polynomials and rational functions Polyanalytic polynomials $p(z,\overline{z})$ are represented by a matrix `p` as

$$p(z,\overline{z}) = \begin{bmatrix}1 & \overline{z} & \cdots & \overline{z}^{d_y}\end{bmatrix} \cdot \begin{bmatrix} a_{00} & \cdots & a_{0d_x} \\ \vdots & \ddots & \vdots \\ a_{d_y0} & \cdots & a_{d_yd_x}\end{bmatrix} \cdot \begin{bmatrix} 1 \\ z \\ \vdots \\ z^{d_x}\end{bmatrix}.$$

For example, the polynomial $p(z,\overline{z}) = 1 + 2z + 3z^2 + 4\overline{z} + 5z\overline{z} + 6z^2\overline{z} + 7\overline{z}^2$ is represented by the matrix

```
p = [1 2 3; ...
     4 5 6; ...
     7 0 0];
```

However, minimizing a polyanalytic polynomial $p(z,\overline{z})$ only makes sense if $p(z,\overline{z})$ is real-valued for all $z \in \mathbb{C}$. A polyanalytic polynomial is real-valued if and only if its matrix representation is Hermitian, i.e., `p == p'`. As with bivariate polynomials, the stationary points of a real-valued polyanalytic polynomial $p(z,\overline{z})$ can be computed with `polymin2(p)`.

As an example, the stationary points of a pseudorandom real-valued polyanalytic polynomial can be computed with

```
p = rand(6)+rand(6)*1i;
p = p*p';
options.Plot = true;
xy = polymin2(p,options);
```

Computing the stationary points of a polyanalytic rational function $\frac{p(z,\overline{z})}{q(z,\overline{z})}$ is completely analogous to the polynomial case. For example,

```
p = rand(6)+rand(6)*1i; p = p*p';
q = rand(4)+rand(4)*1i; q = q*q';
options.Plot = true;
xy = ratmin2(p,q,options);
```

If the matrix `p` contains complex coefficients and is Hermitian, `polymin2` will treat `p` as a real-valued polyanalytic polynomial. Otherwise, it will be treated as a bivariate polynomial. In some cases it may be necessary to specify what type of polynomial `p` is. In that case, set `options.Univariate` to `true` if `p` is a real-valued polyanalytic polynomial and `false` otherwise. The same option also applies to `ratmin2`.

7.2 Isolated solutions of a system of two bivariate polynomials

The functions polymin2 and ratmin2 depend on the lower level function polysol2 to compute the isolated solutions of systems of bivariate polynomials (bisys) or systems of polyanalytic univariate polynomials (unisys). In the case (bisys), polysol2(p,q) computes the isolated real solutions of the system $p(x, y) = q(x, y) = 0$. A solution (x^*, y^*) is said to be isolated if there exists a neighbourhood of (x^*, y^*) in $\mathbb{C}$ that contains no solution other than (x^*, y^*). Note that some systems may have solutions that are isolated in $\mathbb{R}$, but not in $\mathbb{C}$.

The function polysol2 can also solve systems of polyanalytic univariate polynomials (unisys). By default, polysol2(p,q) assumes p and q are polyanalytic univariate if at least one of p and q contains complex coefficients. If this is not the case, the user can specify the type of polynomial by setting options.Univariate to true if both polynomials are polyanalytic univariate or false otherwise.

The algorithm in polysol2 applies several balancing steps to the problem in order to improve the accuracy of the computed roots, before refining them with Newton-Raphson. If the system is poorly scaled, it may be necessary to decrease the balancing tolerance options.TolBal to a smaller value. In Figure 7.2, the solutions of a relatively difficult bivariate system $p(x, y) = q(x, y) = 0$ are plotted. The polynomials $p(x, y)$ and $q(x, y)$ are both of total degree 20 and have coefficients of which the exponents (in base 10) range between 1 and 7.

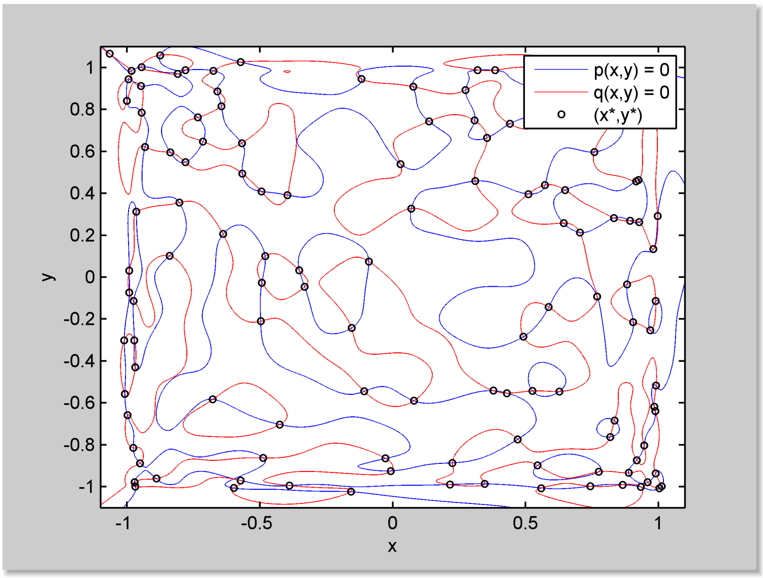


Figure 7.2: A system of bivariate polynomials $p(x, y) = q(x, y) = 0$ of total degree 20.

Acknowledgements

We would like to acknowledge Mariya Ishteva and Nick Vannieuwenhoven for their important contributions to Tensorlab. Additionally, some code in Tensorlab is based on a modified version of GenRTR (Generic Riemannian Trust Region Package), by Pierre-Antoine Absil, Christopher G. Baker, and Kyle A. Gallivan. The Moré–Thuente line search implemented in ls_mt was translated and adapted for complex optimization from the original Fortran MINPACK-2 implementation by Brett M. Averick, Richard G. Carter and Jorge J. Moré.

Laurent Sorber is supported by a doctoral fellowship of the Flanders agency for Innovation by Science and Technology (IWT).

Marc Van Barel is supported by (1) the Research Council KU Leuven: (a) project OT/10/038, (b) CoE EF/05/006 Optimization in Engineering (OPTEC), and by (2) the Belgian Network DYSCO (Dynamical Systems, Control, and Optimization), funded by the Interuniversity Attraction Poles Programme, initiated by the Belgian State, Science Policy Office.

Lieven De Lathauwer is supported by (1) the Research Council KU Leuven: (a) GOA-MaNet, (b) CoE EF/05/006 Optimization in Engineering (OPTEC), (c) CIF1, (d) STRT1/08/023, (2) the Research Foundation Flanders (FWO): (a) project G.0427.10N, (b) Research Communities ICCoS, ANMMM, and MLDM, (3) the Belgian Network DYSCO, and by (4) EU: ERNSI.

References

[1] R. Bro. *Multi-way analysis in the food industry: models, algorithms, and applications*. PhD thesis, University of Amsterdam, 1998.

[2] J. D. Carroll and J.-J. Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of “Eckart–Young” decomposition. *Psychometrika*, 35(3):283–319, 1970.

[3] L. De Lathauwer. Decompositions of a higher-order tensor in block terms — Part I: Lemmas for partitioned matrices. *SIAM J. Matrix Anal. Appl.*, 30(3):1022–1032, 2008.

[4] L. De Lathauwer. Decompositions of a higher-order tensor in block terms — Part II: Definitions and uniqueness. *SIAM J. Matrix Anal. Appl.*, 30(3):1033–1066, 2008.

[5] L. De Lathauwer, B. L. R. De Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl.*, 21(4):1253–1278, 2000.

[6] L. De Lathauwer and D. Nion. Decompositions of a higher-order tensor in block terms — Part III: Alternating least squares algorithms. *SIAM J. Matrix Anal. Appl.*, 30(3):1067–1083, 2008.

[7] L. C. Dixon and G. P. Szegő. *The global optimization problem: an introduction*, pages 1–15. Towards global optimisation II. North-Holland Pub. Co., 1978.

[8] R. A. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an “explanatory” multi-modal factor analysis. *UCLA Working Papers in Phonetics*, 16(1):84–84, 1970.

[9] F. L. Hitchcock. The expression of a tensor or a polyadic as a sum of products. *J. Math. Phys.*, 6(1):164–189, 1927.

[10] F. L. Hitchcock. Multiple invariants and generalized rank of a p-way matrix or tensor. *J. Math. Phys.*, 7(1):39–79, 1927.

[11] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Rev.*, 51(3):455–500, Sept. 2009.

[12] L. Sorber, I. Domanov, M. Van Barel, and L. De Lathauwer. Exact line and plane search for tensor optimization by global minimization of bivariate polynomials and rational functions. ESAT-SISTA Internal Report 13-02, Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium, 2013.

[13] L. Sorber, M. Van Barel, and L. De Lathauwer. Unconstrained optimization of real functions in complex variables. *SIAM J. Optim.*, 22(3):879–898, 2012.

[14] L. Sorber, M. Van Barel, and L. De Lathauwer. A framework for decoupling structure from parameters in tensor decompositions. ESAT-SISTA Internal Report 13-12, Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium, 2013.

[15] L. Sorber, M. Van Barel, and L. De Lathauwer. Optimization-based algorithms for tensor decompositions: canonical polyadic decomposition, decomposition in rank-$(L_r, L_r, 1)$ terms and a new generalization. *SIAM J. Optim.*, 2013. Accepted.

[16] W. Squire and G. Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Rev.*, 10(1):110–112, Mar. 1998.

[17] L. R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.