Tensorlab

User Guide 2014-05-07

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).
[‡]STADIUS, 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

What is Tensorlab? In short, Tensorlab is a MATLAB toolbox for rapid prototyping of (coupled) tensor decompositions with structured factors. In Tensorlab, data sets are stored as (possibly incomplete) vectors, matrices and higher-order tensors. By mixing different types of decompositions and factor structures, a vast amount of factorizations can be computed. Users can define their own factor structures, or choose from the library of preimplemented structures, including nonnegativity, orthogonality, Toeplitz and Vandermonde matrices to name a few.

Section 2 covers how dense, incomplete and sparse data sets are represented as tensors in Tensorlab and the basic operations on such 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. Structured data fusion (SDF), with which multiple data sets can be jointly factorized while imposing structure on the factors is covered in Section 6. Many of the algorithms to accomplish these tasks are based on complex optimization, that is, optimization of functions in complex variables. Section 7 introduces the necessary concepts and shows how to solve different types of complex optimization problems. Finally, Section 8 treats global optimization of bivariate (and polyanalytic) polynomials and rational functions, which appear as subproblems in tensor optimization.

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 the tilde in `[~` and `~,` with `tmp`. To do so on Linux/OS X, browse to Tensorlab and run

```
sed -i "" '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.

Contents.m If you have installed Tensorlab to the directory `tensorlab`, run `doc tensorlab` from the command line for an overview of the toolboxes functionality (or, if that fails, try `help(pwd)`). Both commands display the file `Contents.m`, shown below. Although this user guide covers the most important aspects of Tensorlab, `Contents.m` shows a short one line description of all exported functions.

```
% TENSORLAB
% Version 2.02, 2014-05-07
%
% BLOCK TERM DECOMPOSITION
% Algorithms
%    btd_minf     - BTD by unconstrained nonlinear optimization.
%    btd_nls      - BTD by nonlinear least squares.
% Initialization
%    btd_rnd      - Pseudorandom initialization for BTD.
% Utilities
%    btdgen       - Generate full tensor given a BTD.
%    btdres       - Residual of a BTD.
%
% CANONICAL POLYADIC DECOMPOSITION
% Algorithms
%    cpd          - Canonical polyadic decomposition.
%    cpd_als      - CPD by alternating least squares.
%    cpd_minf     - CPD by unconstrained nonlinear optimization.
%    cpd_nls      - CPD by nonlinear least squares.
%    cpd3_sd      - CPD by simultaneous diagonalization.
%    cpd3_sgsd    - CPD by simultaneous generalized Schur decomposition.
% Initialization
%    cpd_gevd     - CPD by a generalized eigenvalue decomposition.
%    cpd_rnd      - Pseudorandom initialization for CPD.
% Line and plane search
%    cpd_aels     - CPD approximate enhanced line search.
%    cpd_els      - CPD exact line search.
%    cpd_eps      - CPD exact plane search.
%    cpd_lsb      - CPD line search by Bro.
% Utilities
%    cpderr       - Errors between factor matrices in a CPD.
%    cpdgen       - Generate full tensor given a polyadic decomposition.
%    cpdres       - Residual of a polyadic decomposition.
%    rankest      - Estimate rank.
%
% COMPLEX OPTIMIZATION
% Nonlinear least squares
%    nls_gncgs    - Nonlinear least squares by Gauss-Newton with CG-Steihaug.
%    nls_gndl     - Nonlinear least squares by Gauss-Newton with dogleg trust
    region.
%    nls_lm       - Nonlinear least squares by Levenberg-Marquardt.
%    nlsb_gndl    - Bound-constrained NLS by projected Gauss-Newton dogleg TR.
% Unconstrained nonlinear optimization
%    minf_lbfgs   - Minimize a function by L-BFGS with line search.
%    minf_lbfgsdl - Minimize a function by L-BFGS with dogleg trust region.
%    minf_ncg     - Minimize a function by nonlinear conjugate gradient.
%    minf_sr1cgs  - Minimize a function by SR1 with CG-Steihaug.
% Utilities
%    deriv        - Approximate gradient and Jacobian.
%    ls_mt        - Strong Wolfe line search by More-Thuente.
%    mpcg         - Modified preconditioned conjugate gradients method.
```

```
%
% LOW MULTILINEAR RANK APPROXIMATION
% Algorithms
%    lmlra        - Low multilinear rank approximation.
%    lmlra_hooi   - LMLRA by higher-order orthogonal iteration.
%    lmlra_minf   - LMLRA by unconstrained nonlinear optimization.
%    lmlra_nls    - LMLRA by nonlinear least squares.
%    lmlra3_dgn   - LMLRA by a differential-geometric Newton method.
%    lmlra3_rtr   - LMLRA by a Riemannian trust region method.
%    mlsvd        - (Truncated) multilinear singular value decomposition.
% Initialization
%    lmlra_aca    - LMLRA by adaptive cross-approximation.
%    lmlra_rnd    - Pseudorandom initialization for LMLRA.
% Utilities
%    lmlraerr     - Errors between factor matrices in a LMLRA.
%    lmlragen     - Generate full tensor given a core tensor and factor matrices.
%    lmlrares     - Residual of a LMLRA.
%    mlrank       - Multilinear rank.
%    mlrankest    - Estimate multilinear rank.
%
% STRUCTURED DATA FUSION
% Algorithms
%   sdf_minf      - Structured data fusion by unconstrained nonlinear
      optimization.
%   sdf_nls       - Structured data fusion by nonlinear least squares.
% Structure
%   struct_abs        - Absolute value.
%   struct_band       - Band matrix.
%   struct_cell2mat   - Convert the contents of a cell array into a matrix.
%   struct_conj       - Complex conjugate.
%   struct_ctranspose - Complex conjugate transpose.
%   struct_diag       - Diagonal matrix.
%   struct_gram       - Gramian matrix.
%   struct_hankel     - Hankel matrix.
%   struct_inv        - Matrix inverse.
%   struct_invsqrtm   - Matrix inverse square root.
%   struct_invtransp  - Matrix inverse transpose.
%   struct_LL1        - Structure of the third factor matrix in a rank-(Lr,Lr,1)
      BTD.
%   struct_log        - Natural logarithm.
%   struct_matvec     - Matrix-vector and matrix-matrix product.
%   struct_nonneg     - Nonnegative array.
%   struct_normalize  - Normalize columns to unit norm.
%   struct_orth       - Rectangular matrix with orthonormal columns.
%   struct_plus       - Plus.
%   struct_poly       - Matrix with columns as polynomials.
%   struct_power      - Array power.
%   struct_rational   - Matrix with columns as rational functions.
%   struct_rbf        - Matrix with columns as sums of Gaussian RBF kernels.
%   struct_sigmoid    - Constrain array elements to an interval.
%   struct_sqrt       - Square root.
%   struct_sum        - Sum of elements.
```

```
%   struct_times      - Times.
%   struct_toeplitz   - Toeplitz matrix.
%   struct_transpose  - Transpose.
%   struct_tridiag    - Tridiagonal matrix.
%   struct_tril       - Lower triangular matrix.
%   struct_triu       - Upper triangular matrix.
%   struct_vander     - Vandermonde matrix.
%
% UTILITIES
% Clustering
%    gap          - Optimal clustering based on the gap statistic.
%    kmeans       - Cluster multivariate data using the k-means++ algorithm.
% Polynomials
%    polymin      - Minimize a polynomial.
%    polymin2     - Minimize bivariate and real polyanalytic polynomials.
%    polyval2     - Evaluate bivariate and univariate polyanalytic polynomials.
%    polysol2     - Solve a system of two bivariate polynomials.
%    ratmin       - Minimize a rational function.
%    ratmin2      - Minimize bivariate and real polyanalytic rational functions.
% Statistics
%    cum3         - Third-order cumulant tensor.
%    cum4         - Fourth-order cumulant tensor.
%    scov         - Shifted covariance matrices.
% Tensors
%    dotk         - Dot product in K-fold precision.
%    fmt          - Format data set.
%    frob         - Frobenius norm.
%    ful          - Convert formatted data set to an array.
%    kr           - Khatri-Rao product.
%    kron         - Kronecker product.
%    mat2tens     - Tensorize a matrix.
%    mtkronprod   - Matricized tensor Kronecker product.
%    mtkrprod     - Matricized tensor Khatri-Rao product.
%    noisy        - Generate a noisy version of a given array.
%    sumk         - Summation in K-fold precision.
%    tens2mat     - Matricize a tensor.
%    tens2vec     - Vectorize a tensor.
%    tmprod       - Mode-n tensor-matrix product.
%    vec2tens     - Tensorize a vector.
% Visualization
%    slice3       - Visualize a third-order tensor with slices.
%    spy3         - Visualize a third-order tensor's sparsity pattern.
%    surf3        - Visualize a third-order tensor with surfaces.
%    voxel3       - Visualize a third-order tensor with voxels.
```

2 Data sets: dense, incomplete and sparse tensors

2.1 Representation

Dense tensors Scalars, vectors and matrices are zero-, one- and two-dimensional tensors, respectively. Arrays with three or more dimensions are called higher-order tensors. In Tensorlab, data sets are represented as dense, sparse or incomplete tensors. A dense tensor is simply a MATLAB array, e.g., `A = randn(10,10)` or `T = randn(5,5,5)`. All functions in Tensorlab accept dense tensors, and many (but not all) also accept incomplete and sparse tensors transparently.

Incomplete tensors An incomplete tensor is a data set in which some (or most) of the entries are unknown. To create an incomplete tensor, define a structure containing the tensor's known elements, their positions, the tensor's size and an incomplete flag as:

```
T.val = randn(3,1);
T.sub = [1 2 3; 4 5 6; 7 8 9];
T.size = [9 9 9];
T.incomplete = true;
```

and then call

```
T = fmt(T);
```

to convert the tensor to Tensorlab format. This creates a $9\times9\times9$ tensor `T` with three known elements at positions $(1,2,3)$, $(4,5,6)$ and $(7,8,9)$. Alternatively, the user may supply the linear indices in `T.ind` instead of the subindices `T.sub`. To convert linear indices to subindices or vice versa, see the MATLAB functions `ind2sub` and `sub2ind`, respectively.

Another way to create an incomplete tensor is to create a dense tensor in which the unknown values are `NaN`. For example

```
T = randn(5,5,5);
T(1:31:end) = NaN;
T = fmt(T);
```

creates a $5\times5\times5$ tensor `T` with diagonal elements equal to `NaN` and formats it as an incomplete tensor. If there are no entries equal to `NaN`, then `fmt(T)` returns `T` itself. To convert an incomplete tensor back to a MATLAB array, use `ful(T)`.

Sparse tensors A sparse tensor is a data set in which most of the entries are zero. To create a sparse tensor, define a structure containing the tensor's nonzero elements in the same way as for incomplete tensors and set the sparse flag:

```
T.val = randn(3,1);
T.sub = [1 2 3; 4 5 6; 7 8 9];
T.size = [9 9 9];
T.sparse = true;
```

before formatting the tensor with `T = fmt(T)`.

Another way to create a sparse tensor is to create a dense tensor in which at most 5% of the entries are nonzero. For example

```
T = zeros(5,5,5);
T(1:31:end) = 1;
T = fmt(T);
```

creates a $5 \times 5 \times 5$ diagonal tensor `T` with diagonal elements equal to 1 and formats it as a sparse tensor. If not enough entries are zero, then `fmt(T)` returns `T` itself. To convert a sparse tensor back to a MATLAB array, use `ful(T)`.

2.2 Tensor operations

2.2.1 For dense tensors

Matricization and tensorization A dense tensor `T` can be flattened into a matrix with `M = 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 `tens2mat`, 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 a 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 $\mathtt{T} \cdot_n \mathtt{U}$. The function `tmprod(T,U,mode)` computes the tensor-matrix product $\mathtt{T} \cdot_{\mathtt{mode(1)}} \mathtt{U\{1\}} \cdot_{\mathtt{mode(2)}} \mathtt{U\{2\}} \cdots$. 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]`. To compute a single tensor-matrix product, it is not necessary to use a cell array. The following is an example of a single mode-2 tensor-matrix product:

```
T = randn(3,5,7);
S = tmprod(T,randn(13,5),2);
```

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, then the Khatri–Rao product of A and B is the column-wise Kronecker product

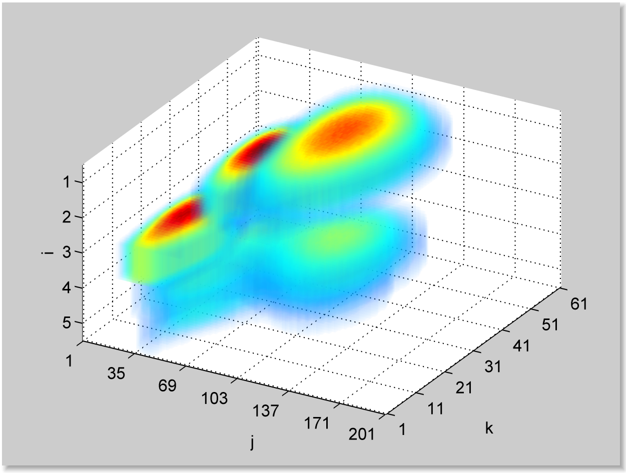
$$A \odot B := \begin{bmatrix} \boldsymbol{a}_1 \otimes \boldsymbol{b}_1 & \cdots & \boldsymbol{a}_N \otimes \boldsymbol{b}_N \end{bmatrix}.$$

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

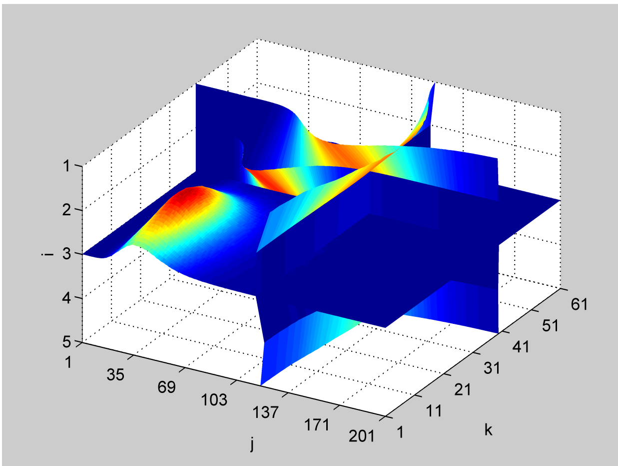
Visualization Tensorlab offers three methods to visualize dense third-order tensors: `slice3`, `surf3` and `voxel3`. The following example demonstrates these methods on the amino acids dataset [1]:

```
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);
```

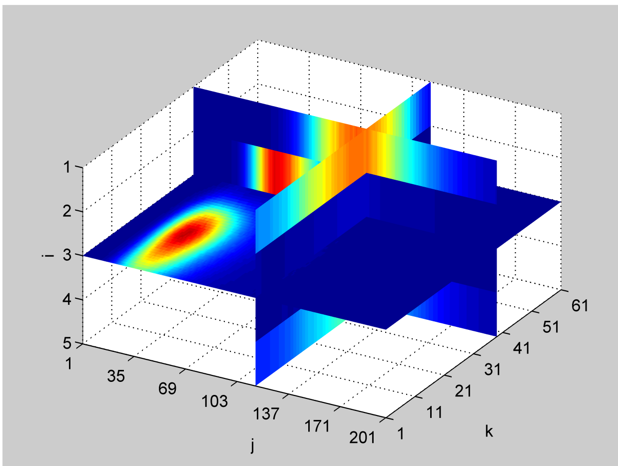
The resulting MATLAB figures can be seen in Figure 2.1. By default, the `voxel3` plot uses a fast but not so accurate renderer. To enable the slow but accurate renderer, use `voxel3(X,'fast',false)`. See the `help` information for more rendering options.



(a) `voxel3`



(b) `surf3`



(c) `slice3`

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

2.2.2 For dense, incomplete and sparse tensors

Frobenius norm The Frobenius norm of a tensor is the square root of the sum of square magnitudes of its (known) elements. Given a tensor `T`, its Frobenius norm can be computed with `frob(T)`. If the tensor is dense, this is equivalent with `norm(T(:))`, i.e., the two-norm of the vectorized tensor. The squared Frobenius norm can be computed with `frob(T,'squared')`.

Matricized tensor Kronecker product Often, algorithms for computing tensor decompositions do not explicitly need matricized tensors or Kronecker products, but rather the matricized tensor Kronecker product `tens2mat(T,n)*kron(U([end:-1:n+1 n-1:-1:1]))`, where the second operand represents the Kronecker product `U{end}` $\otimes \cdots \otimes$ `U{n+1}` $\otimes$ `U{n-1}` $\otimes \cdots \otimes$ `U{1}`. The function `mtkronprod(T,U,n)` computes the result of this operation without explicitly computing either of the operands and without permuting the elements of the tensor `T` in memory.

Another useful way of interpreting this product is as the result of

```
mode = [1:n-1 n+1:length(U)];
Ut = cellfun(@transpose,U,'UniformOutput',false);
tens2mat(tmprod(T,Ut(mode),mode),n)
```

Matricized tensor Khatri-Rao product Similarly, algorithms for computing tensor decompositions usually don't explicitly need matricized tensors or Khatri-Rao products, but rather the matricized tensor Khatri-Rao product `tens2mat(T,n)*kr(U([end:-1:n+1 n-1:-1:1]))`. The function `mtkrprod(T,U,n)` computes the result of this operation without explicitly computing either of the operands and without permuting the elements of the tensor `T` in memory.

Another useful way of interpreting the rth column of this product is as the result of

```
mode = [1:n-1 n+1:length(U)];
Urt = cellfun(@(u)u(:,r).',U,'UniformOutput',false);
tens2mat(tmprod(T,Urt(mode),mode),n)
```

Creating noisy versions of (cell) arrays The `noisy` function can be used to create a noisy version of a MATLAB array, of a (nested) cell array of arrays and of incomplete or sparse tensors. For example,

```
U = {randn(10,2),randn(15,2),randn(20,2)};
SNR = 20;
Uhat = noisy(U,SNR);
```

creates a cell array `U` and a noisy version of that cell array called `Uhat` with a signal-to-noise ratio of 20 dB.

3 Canonical polyadic decomposition

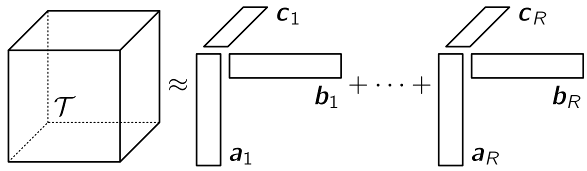


Figure 3.1: A canonical polyadic decomposition of a third-order tensor.

The canonical polyadic decomposition (CPD) [2, 7–10] approximates a tensor with a sum of R rank-one tensors. Let $\mathcal{A} \circ \mathcal{B}$ denote the outer product between an N-dimensional tensor $\mathcal{A}$ and an M-dimensional tensor $\mathcal{B}$, then $\mathcal{A} \circ \mathcal{B}$ is the $(N + M)$-dimensional 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 $\boldsymbol{a} \circ \boldsymbol{b} \equiv \boldsymbol{a} \cdot \boldsymbol{b}^\top$ is a rank-one matrix and $\boldsymbol{a} \circ \boldsymbol{b} \circ \boldsymbol{c}$ is defined to be a rank-one tensor. Let `T` be a tensor of dimensions $I_1 \times I_2 \times \cdots \times I_N$, and let `U{n}` be matrices of size $I_n \times R$, then

$$\mathtt{T} \approx \sum_{r=1}^{R} \mathtt{U\{1\}(:,r)} \circ \mathtt{U\{2\}(:,r)} \circ \cdots \circ \mathtt{U\{N\}(:,r)}$$

is a CPD of `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 = @rand;
U = cpd_rnd(size_tens,R,options);
```

and the inline equivalent

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

generate `U{n}` as `rand(size_tens(n),R)+rand(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 `T` can be computed with

```
T = cpdgen(U);
```

This is equivalent to

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

3.2 Computing the CPD

The principal method To compute the CPD of a dense, sparse or incomplete tensor `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.Display = true; % Show progress on the command line.
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 algorithm 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(cpdres(T,U))');
grid on;
```

Computing the error The residual between a tensor `T` and its CPD approximation defined by `Uhat` can be computed with

```
res = cpdres(T,Uhat);
```

If the tensor is dense, then the result is equivalent to `cpdgen(Uhat)-T`. The relative error between the tensor and its CPD approximation can then be computed as

```
relerr = frob(cpdres(T,Uhat))/frob(T);
```

One can also consider the relative error between the factor matrices `U{n}` that generated `T` and their approximations `Uhat{n}` computed by `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. See the `help` information for more details.

3.3 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)` on a dense, sparse or incomplete tensor `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.MinRelErr`. The lower bound is based on the truncation error of the tensor's multilinear singular values [5]. For incomplete and sparse tensors, this lower bound is not available and the first value to be tried for R is 1. 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.

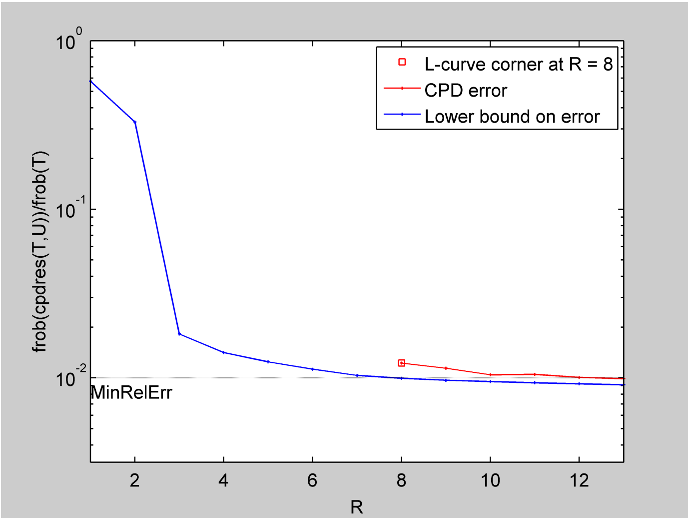


Figure 3.2: The `rankest` tool for choosing the number of rank-one terms R in a CPD.

4 Low multilinear rank approximation

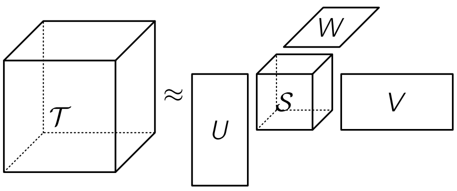


Figure 4.1: A low multilinear rank approximation of a third-order tensor.

A low multilinear rank approximation (LMLRA) [10, 17] approximates a tensor by a smaller (core) tensor and a set of factor matrices. Let `T` and `S` be 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 $\cdot_n$ denote the mode-n tensor-matrix product (see Section 2.2.1), then

$$\mathtt{T} \approx \mathtt{S} \cdot_1 \mathtt{U\{1\}} \cdot_2 \mathtt{U\{2\}} \cdot_3 \cdots \cdot_N \mathtt{U\{N\}}$$

is a LMLRA of the tensor `T` in the core tensor `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 factor matrices `U = {U{1},U{2},...}` with orthonormal columns and a core tensor `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 `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 `S`, its associated full tensor `T` can be computed with

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

This is equivalent to

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

4.2 Computing a LMLRA

The principal method To compute a LMLRA of a dense, sparse or incomplete tensor `T` with a core tensor 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 version of T with 20dB SNR.
[Uhat,Shat] = lmlra(noisy(T,20),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` and `S0` (e.g., `lmlra_aca`), after which it *executes an algorithm* to compute the LMLRA given the initialization (e.g., `lmlra_nls`).

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.Display = true; % Show progress on the command line.
options.Initialization = @lmlra_rnd; % Select pseudorandom initialization.
options.Algorithm = @lmlra_nls; % Select NLS 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 objective function value:

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

Computing the error The residual between a tensor `T` and its LMLRA defined by `Uhat` and `Shat` can be computed with

```
res = lmlrares(T,Uhat,Shat);
```

If the tensor is dense, then the result is equivalent to `lmlragen(Uhat,Shat)-T`. The relative error between the tensor and its LMLRA can then be computed as

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

If the factor matrices `U{n}` that generated `T` are known, the subspace angle between them and their approximations `Uhat{n}` computed by `lmlra` 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 nth entry is `subspace(U{n},Uhat{n})`.

4.3 Choosing the size of the core tensor

For dense tensors, use the `mlrankest` tool to help determine the size of the core tensor `size_core`. Running `mlrankest(T)` plots an L-curve which represents the balance between an upper bound [5] on the relative error of a LMLRA of `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 at 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.

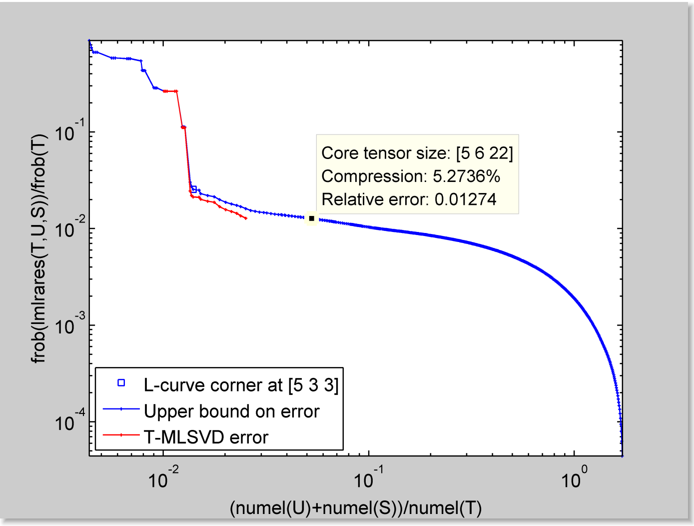


Figure 4.2: The `mlrankest` tool for choosing the core tensor size `size_core` in a LMLRA.

5 Block term decomposition

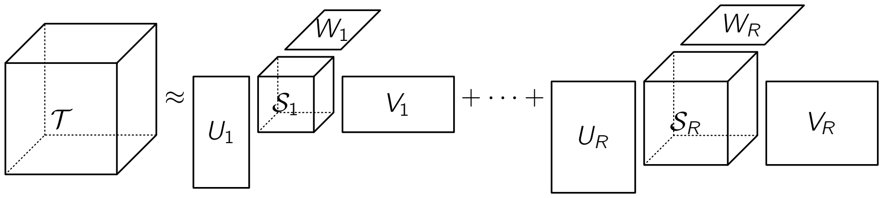


Figure 5.1: A block term decomposition of a third-order tensor.

A block term decomposition (BTD) [10,17] approximates a tensor by a sum of low multilinear rank terms. Let `T` be a tensor of dimensions $I_1 \times I_2 \times \cdots \times I_N$ and let `U{r}` represent the rth block term in the BTD. More specifically, let `U{r}{N+1}` be a core tensor of dimensions $J_1^{(r)} \times J_2^{(r)} \times \cdots \times J_N^{(r)}$ and let `U{r}{n}` be matrices of size $I_n \times J_n^{(r)}$ ($J_n^{(r)} \leq I_n$) for $1 \leq n \leq N$, then

$$\mathtt{T} \approx \sum_{r=1}^{R} \mathtt{U\{r\}\{N+1\}} \cdot_1 \mathtt{U\{r\}\{1\}} \cdot_2 \mathtt{U\{r\}\{2\}} \cdot_3 \cdots \cdot_{\mathtt{N}} \mathtt{U\{r\}\{N\}}$$

is a BTD of the tensor `T` in the block terms `U{r}`. A visual representation of this decomposition in the third-order case is shown in Figure 5.1.

5.1 Problem and tensor generation

Generating pseudorandom factor matrices and core tensor A cell array of block terms `U = {U{1},U{2},...}` in which the rth term has a core tensor `U{r}{end}` of size `size_core{r}` corresponding to a BTD of a tensor of dimensions `size_tens` can be generated with

```
size_tens = [17 19 21];
size_core = {[3 5 7],[6 3 5],[4 3 4]};
U = btd_rnd(size_tens,size_core);
```

By default `btd_rnd` generates `U{r}{n}` using `randn`, after which the factor matrices `U{r}{1:N}` are orthogonalized. Other generators can also be specified with an options structure, e.g.,

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

and its inline equivalent

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

Generating the associated full tensor Given a cell array of block terms `U = {U{1},U{2},...}`, its associated full tensor `T` can be computed with

```
T = btdgen(U);
```

This is equivalent to

```
R = length(U);
N = length(U{1})-1;
size_tens = cellfun('size',U{1}(1:N),1);
T = zeros(size_tens);
for r = 1:R
    T = T+tmprod(U{r}{N+1},U{r}(1:N),1:N);
end
```

5.2 Computing a BTD

With a specific algorithm Currently, the user must generate his or her own initialization and select a specific (family of) algorithm(s) such as `btd_nls` or `btd_minf` to compute the BTD given the initialization.

To compute a BTD of a dense, sparse or incomplete tensor `T` given an initialization `U0`, run `btd_nls(T,U0)`. For example,

```
% Generate pseudorandom BTD U and associated full tensor T.
size_tens = [17 19 21];
size_core = {[2 2 2],[2 2 2],[2 2 2]};
U = btd_rnd(size_tens,size_core);
T = btdgen(U);

% Generate an initialization U0 and compute the BTD with nonlinear least squares.
U0 = noisy(U,20);
Uhat = btd_nls(T,U0);
```

generates a tensor `T` as the sum of three real rank-(2,2,2) tensors and then computes its 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.Display = 5; % Show convergence progress every 5 iterations.
options.MaxIter = 200; % Set stop criteria.
options.TolFun = 1e-12;
options.TolX = 1e-12;
Uhat = btd_nls(T,U0,options);
```

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] = btd_nls(T,U0,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(btdres(T,U))');
grid on;
```

Computing the error The residual between a tensor `T` and its BTD defined by `Uhat` can be computed with

```
res = btdres(T,Uhat);
```

If the tensor is dense, then the result is equivalent to `btdgen(Uhat)-T`. The relative error between the tensor and its BTD can then be computed as

```
relerr = frob(btdres(T,Uhat))/frob(T);
```

6 Structured data fusion

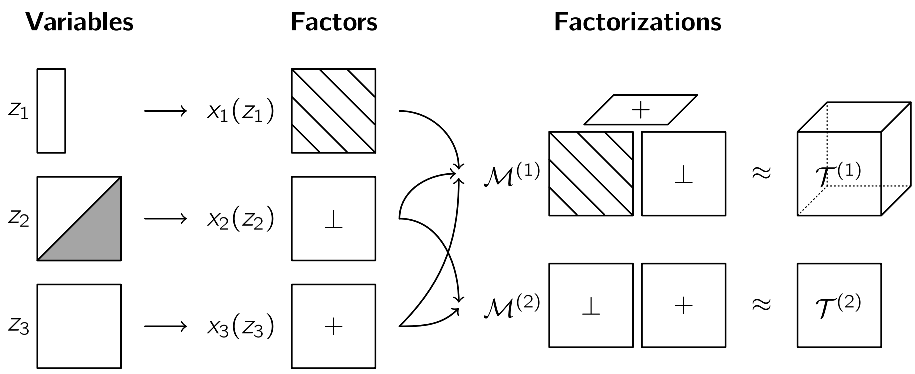


Figure 6.1: Schematic of structured data fusion. The vector z_1, upper triangular matrix z_2 and full matrix z_3 are transformed into a Toeplitz, orthogonal and nonnegative matrix, respectively. The resulting factors are then used to jointly factorize two coupled data sets.

Structured data fusion (SDF) [14] is the practice of jointly factorizing one or more coupled data sets while optionally imposing structure on the factors.

Each data set—stored as a dense, sparse or incomplete tensor, cf. Section 2—in a data fusion problem can be factorized with a different tensor decomposition. Currently, the user has the choice of the CPD and BTD models, and with a bit of effort it is easy to add new models as well.

Structure can be imposed on the factors in a modular way and the user can choose from a library of predefined structures such as nonnegativity, orthogonality, Hankel, Toeplitz, Vandermonde, matrix inverse, and many more. See `Contents.m` for a complete list. By selecting the right structures you can even compute classical matrix decompositions such as the QR factorization, eigenvalue decomposition and singular value decomposition.

6.1 Domain specific language for SDF

Tensorlab uses a domain specific language (DSL) for modelling structured data fusion problems. The three key ingredients of an SDF model are (1) defining variables, (2) defining factors as transformed variables and (3) defining the data sets and which factors to use in their factorizations (cf. Figure 6.1).

Example 1: nonnegative symmetric CPD. The DSL is best explained by example. To start, we'll compute the nonnegative symmetric CPD of an incomplete tensor `T`. First, generate the tensor:

```
% Generate a nonnegative symmetric CPD.
I = 15;
R = 4;
U = rand(I,R);
U = {U,U,U};
T = cpdgen(U);

% Remove 10% of its entries.
T(randperm(numel(T),round(0.1*numel(T)))) = NaN;

% Format as incomplete tensor.
T = fmt(T);
```

Next, create a structure `model` which defines the variables of the SDF problem. In this case, there is only one variable and its size is equal to that of the factor `U`. The `'variables'` field defines the parameters which are optimized, and is also used as initialization for the SDF algorithm.

```
% Define model variables.
model.variables.u = randn(I,R);
```

Here, the variable `u` is defined as a MATLAB array. It is also perfectly valid to define variables as (nested) cell arrays of arrays, if desired. Now we need to define the factors. There is only one factor in this CPD, which we will define as a transformation of the variable `u`. More specifically, we will require the factor to be nonnegative:

```
% Define model factors as transformed variables.
model.factors.U = {'u',@struct_nonneg}; % Create U as struct_nonneg(u).
```

The factor `U` is built taking the variable `u`, and applying the transformation `@struct_nonneg`. In fact, factors can be built with a much more complex structure using subfactors, see the following examples for details. Finally, we define the data set to be factorized and which factors to use:

```
% Define model factorizations.
model.factorizations.myfac.data = T;
model.factorizations.myfac.cpd = {'U','U','U'};
```

Each factorization in the SDF problem should be given a new name. In this case there is only one factorization `'myfac'` and it contains two fields. The first is `'data'` and contains the tensor to be factorized. The second should be either `'cpd'` or `'btd'`, depending on which model to use, and should define the factors to be used in the decomposition.

Note that it is not necessary to use fields to describe the names of the variables and factors. Instead, one may also create cell arrays of variables and factors and use indices to refer to them. In this format, the model would be written as

```
% Equivalent SDF model without using names for variables and factors.
model.variables = { randn(I,R) };
model.factors = { {1,@struct_nonneg} };
model.factorizations.myfac.data = T;
model.factorizations.myfac.cpd = {1,1,1};
```

The model can now be solved with one of the two families of algorithms for SDF problems: `sdf_minf` and `sdf_nls`. In the case of many missing entries, the `sdf_minf` family is likely to perform best. Their first output contains the optimized variables and factors in the fields `'variables'` and `'factors'`, respectively:

```
% Solve the SDF problem.
options.Display = 5; % View convergence progress every 5 iterations.
sol = sdf_nls(model,options);
sol.variables
sol.factors
```

Example 2: structured coupled matrix factorization. Let X and Y be square matrices of order N that we wish to jointly factorize as $U \cdot \Lambda_X \cdot V^T$ and $U \cdot \Lambda_Y \cdot W^T$, respectively. The factor U is common to both X and Y, and the extent to which the columns are shared is given by the absolute values of the diagonal matrices Λ_X and Λ_Y. Furthermore, we will require the elements of W to lie in the interval (-3,5) and impose a Vandermonde structure on V, so that

$$V = \begin{bmatrix} v_1^0 & v_1^1 & \cdots & v_1^d \\ \vdots & \vdots & \ddots & \vdots \\ v_N^0 & v_N^1 & \cdots & v_N^d \end{bmatrix}$$

for some positive integer d. Notice that V depends only on a so-called generator vector $\boldsymbol{v} = \begin{bmatrix} v_1 & \cdots & v_N \end{bmatrix}^\mathrm{T}$.

First, we'll create the matrices X and Y:

```
% Generate structured coupled matrices X and Y.
N = 10;
R = 4;
U = randn(N,R);
V = bsxfun(@power,randn(N,1),0:R-1); % Vandermonde factor.
W = rand(N,R)*(5+3)-3; % Elements in (-3,5).
lambdaX = 0:3; % X does not share first column of U.
lambdaY = 3:-1:0; % Y does not share last column of U.
X = U*diag(lambdaX)*V.';
Y = U*diag(lambdaY)*W.';
```

Then, we define a structure containing the model's variables

```
% Define model variables.
model.variables.u = randn(N,R);
model.variables.v = randn(N,1);
model.variables.w = randn(N,R);
model.variables.lambdaX = randn(1,R);
model.variables.lambdaY = randn(1,R);
```

Here, the variables are the matrices U and W, the generator vector $\boldsymbol{v}$ for the Vandermonde matrix V and the diagonals $\boldsymbol{\lambda}_X$ and $\boldsymbol{\lambda}_Y$. The model assumes that X and Y share at most $R = 4$ vectors in U. Next, we define the factors as transformed variables with

```
% Define the structure for V by creating an anonymous function which stores deg.
deg = [0 R-1];
vanderm = @(z,task)struct_vander(z,task,deg);

% Define the structure for W by creating an anonymous function which stores rng.
rng = [-3 5];
sigmoid = @(z,task)struct_sigmoid(z,task,rng);

% Define model factors as transformed variables.
model.factors.U = 'u';
model.factors.V = {'v',vanderm}; % Create V as vanderm(v).
model.factors.W = {'w',sigmoid}; % Create W as sigmoid(w).
model.factors.LX = 'lambdaX';
model.factors.LY = 'lambdaY';
```

In this example the transformations depend on parameters. To pass along these parameters, we encapsulate them inside anonymous functions. For example, the sigmoid transformation `struct_sigmoid` requires the interval `rng` to constrain its argument in.

Now we add the two factorizations of X and Y to the model with

```
% Define the joint factorization of the matrices X and Y.
model.factorizations.xfac.data = X;
model.factorizations.xfac.cpd = {'U','V','LX'};
model.factorizations.yfac.data = Y;
model.factorizations.yfac.cpd = {'U','W','LY'};
```

Here, we use the CPD to describe the factorizations $X = U \cdot \Lambda_X \cdot V^T$ and $Y = U \cdot \Lambda_Y \cdot W^T$ by associating the factor matrices $\boldsymbol{\lambda}_X$ and $\boldsymbol{\lambda}_Y$ with the third dimension of X and Y. Alternatively, we could describe the two factorizations with the BTD model by imposing the core tensor to be a diagonal matrix with `struct_diag`, but this is less efficient than using the CPD model.

The SDF problem can now be solved with

```
% Solve the SDF problem.
options.Display = 5; % View convergence progress every 5 iterations.
options.TolFun = 1e-9; % Stop earlier.
sol = sdf_nls(model,options);
sol.variables, sol.factors
```

Although the algorithm converges to a solution, it usually does not converge to the solution we used to generate the problem. This indicates that there are not enough constraints to make the solution to this SDF problem unique (in contrary to the first example).

Example 3: an orthogonal factor. We show how to compute a simple CPD in which one of the factors is constrained to have orthonormal colums. To this end, we will create a variable q that parameterizes a matrix with orthonormal columns Q. The structure `struct_orth` can then be used to transform the variable q into the factor Q (cf. Figure 6.1). First, we set up the problem:

```
% Generate CPD, wherein one factor has orthonormal columns.
I = 15;
R = 4;
U = cpd_rnd([I I I],R);
U{1} = orth(U{1}); % Enforce orthonormal columns.
T = cpdgen(U);
```

The help information provided by `struct_orth` tells us that the variable q should be a vector of length $IR - 0.5R(R-1)$. To transform q into a matrix with orthonormal columns, we need to pass the size of Q to `struct_orth`. One way to do this is with an anonymous function, as shown below. Now we are ready to define and solve the SDF problem:

```
% Define model variables.
model.variables.q = randn(I*R-0.5*R*(R-1),1);
model.variables.b = randn(I,R);
model.variables.c = randn(I,R);

% Define a function which will transform q into Q.
% This anonymous function stores the size of Q as its last argument.
orthq = @(z,task)struct_orth(z,task,[I R]);

% Define model factors.
model.factors.Q = {'q',orthq}; % Create Q as orthq(q).
model.factors.B = 'b';
model.factors.C = 'c';

% Define model factorizations.
model.factorizations.myfac.data = T;
model.factorizations.myfac.cpd = {'Q','B','C'};

% Solve the SDF problem.
options.Display = 5; % View convergence progress every 5 iterations.
sol = sdf_nls(model,options);

% Check that Q indeed has orthonormal columns
sol.factors.Q'*sol.factors.Q
```

Example 4: rank-$(L_r, L_r, 1)$ BTD. This example shows how to use the BTD model and how to use subfactors to build more complex factors in SDF models. We want to compute a rank-$(L_r, L_r, 1)$ BTD [3, 4, 13] of a tensor `T` in two terms, i.e.,

$$\mathsf{T} \approx (\mathsf{A1} \cdot \mathsf{B1}^{\mathsf{T}}) \circ \mathsf{c1} + (\mathsf{A2} \cdot \mathsf{B2}^{\mathsf{T}}) \circ \mathsf{c2}$$

where `c1` and `c2` are vectors and `size(A1,2)` equals `size(B1,2)` equals L_1 and analogously for L_2.

Example 4a: modelling a rank-$(L_r, L_r, 1)$ BTD with a BTD. We could compute this decomposition by formulating it as a more general BTD

$$\mathrm{T} \approx (\mathrm{A1} \cdot \mathrm{S1} \cdot \mathrm{B1}^{\mathsf{T}}) \circ \mathrm{c1} + (\mathrm{A2} \cdot \mathrm{S2} \cdot \mathrm{B2}^{\mathsf{T}}) \circ \mathrm{c2} = \mathrm{S1} \bullet_1 \mathrm{A1} \bullet_2 \mathrm{B1} \bullet_3 \mathrm{c1} + \mathrm{S2} \bullet_1 \mathrm{A2} \bullet_2 \mathrm{B2} \bullet_3 \mathrm{c2}$$

where we have introduced the core tensors `S1` and `S2`. First we create the tensor `T` with

```
% Generate rank-(Lr,Lr,1) BTD.
size_tens = [10 10 10];
L1 = 2;
L2 = 2;
U = btd_rnd(size_tens,{[L1 L1 1],[L2 L2 1]});
T = btdgen(U);
```

then we define the model as

```
% Define rank-(Lr,Lr,1) BTD model using the BTD.
model.variables.A1 = randn(size_tens(1),L1);
model.variables.B1 = randn(size_tens(2),L1);
model.variables.c1 = randn(size_tens(3),1);
model.variables.S1 = randn(L1,L1,1);
model.variables.A2 = randn(size_tens(1),L2);
model.variables.B2 = randn(size_tens(2),L2);
model.variables.c2 = randn(size_tens(3),1);
model.variables.S2 = randn(L2,L2,1);
model.factors = { 'A1','B1','c1','S1', 'A2','B2','c2','S2' };
model.factorizations.mybtd.data = T;
model.factorizations.mybtd.btd = {{1,2,3,4},{5,6,7,8}};
```

where, for the sake of brevity, we have used the index-notation for the `'factors'` field. Notice that the BTD is specified in the same format as for other BTD algorithms, except in place of factor matrices and core tensors, there are references to factors. The BTD can then be computed by

```
% Solve the SDF problem.
options.Display = 5; % View convergence progress every 5 iterations.
sol = sdf_nls(model,options);
sol.variables
sol.factors
```

Example 4b: modelling a rank-$(L_r, L_r, 1)$ BTD with a CPD. A different and more efficient approach to compute the rank-$(L_r, L_r, 1)$ BTD would be to cast it as a CPD as in

$$\mathrm{T} \approx (\mathrm{A1} \cdot \mathrm{B1}^{\mathsf{T}}) \circ \mathrm{c1} + (\mathrm{A2} \cdot \mathrm{B2}^{\mathsf{T}}) \circ \mathrm{c2} = \sum_{j=1}^{\mathrm{L1}} \mathrm{A1}_{:j} \circ \mathrm{B1}_{:j} \circ \mathrm{c1} + \sum_{j=1}^{\mathrm{L2}} \mathrm{A2}_{:j} \circ \mathrm{B2}_{:j} \circ \mathrm{c2}.$$

Here, the CPD's third factor matrix looks like `[repmat(c1,1,L1) repmat(c2,1,L2)]`, while the first two would be `[A1 A2]` and `[B1 B2]`, respectively. These factor matrices are all built out of subfactor matrices, which we can implement in the DSL as follows:

```
% Define rank-(Lr,Lr,1) BTD model using the CPD.
model.variables.A1 = randn(size_tens(1),L1);
model.variables.B1 = randn(size_tens(2),L1);
model.variables.c1 = randn(size_tens(3),1);
model.variables.A2 = randn(size_tens(1),L2);
model.variables.B2 = randn(size_tens(2),L2);
model.variables.c2 = randn(size_tens(3),1);
% The factors A, B and C are built out of subfactors.
% Structure may also be imposed on subfactors if desired.
model.factors.A = { {'A1'},{'A2'} };
model.factors.B = { {'B1'},{'B2'} };
model.factors.C = { {'c1'},{'c1'},{'c2'},{'c2'} };
model.factorizations.mybtd.data = T;
model.factorizations.mybtd.cpd = {'A','B','C'};
```

Let's deconstruct the meaning of this syntax: `model.factors.A = { {'A1'},{'A2'} }` says that the factor `A` is a matrix consisting of two subfactors placed horizontally next to each other. Subfactors may also be placed vertically by using a semicolon `;` instead of a comma `,` to separate them. The SDF algorithm will first build the subfactors, which are in this case simply references to the variables `A1` and `A2`, and then call `cell2mat` on the cell array of subfactors to create the factor `A`. Like factors, structure may also be imposed on subfactors. For instance, we could have written `model.factors.A = { {'A1',@struct_sigmoid},{'A2'} }` to constrain the elements of the first submatrix in the factor `A` to the interval $(-1, 1)$ (the elements of the variable `A1` itself are not restricted to this interval however).

Because computing the rank-$(L_r, L_r, 1)$ BTD is a relatively common task, we show a final compact and efficient way to do so using the predefined structure `struct_LL1`:

```
% Define rank-(Lr,Lr,1) BTD model using the CPD (more efficient).
L = [L1 L2];
LL1 = @(z,task)struct_LL1(z,task,L);
model.variables.A = randn(size_tens(1),L1+L2);
model.variables.B = randn(size_tens(2),L1+L2);
model.variables.c = randn(size_tens(3),2);
model.factors.A = 'A';
model.factors.B = 'B';
model.factors.C = {'c',LL1};
model.factorizations.mybtd.data = T;
model.factorizations.mybtd.cpd = {'A','B','C'};
```

Example 5: constants. When a factor or subfactor is intended to be constant, then instead of referring to a variable by its name or index, use the constant array itself.

Example 5a: known factor matrix. The following example computes a CPD in which one factor matrix is known:

```
% Generate a CPD.
I = 15;
R = 4;
U = cpd_rnd([I I I],R);
T = cpdgen(U);
```

```
% Model the CPD of T, where the second half of U{3} is known.
model.variables.a = randn(size(U{1}));
model.variables.b = randn(size(U{1}));
model.variables.c = randn(size(U{1},1),2);
model.factors.A = 'a';
model.factors.B = 'b';
model.factors.C = U{3}; % The third factor is constant.
model.factorizations.myfac.data = T;
model.factorizations.myfac.cpd = {'A','B','C'};

% Solve the SDF model.
options.Display = 5; % View convergence progress every 5 iterations.
sol = sdf_nls(model,options);
sol.variables
sol.factors
```

Example 5b: partially known factor matrix. To create a factor of which some of the columns are known, simply define the factor to consist of a number of subfactors where one of the subfactors is a numeric array. The following example computes a CPD in which part of one factor matrix is known:

```
% Generate a CPD.
I = 15;
R = 4;
U = cpd_rnd([I I I],R);
T = cpdgen(U);

% Model the CPD of T, where the second half of U{3} is known.
model.variables.a = randn(size(U{1}));
model.variables.b = randn(size(U{1}));
model.variables.c = randn(size(U{1},1),2);
model.factors.A = 'a';
model.factors.B = 'b';
model.factors.C = {'c',U{3}(:,1:2)}; % The third factor is partially known.
model.factorizations.myfac.data = T;
model.factorizations.myfac.cpd = {'A','B','C'};

% Solve the SDF model.
options.Display = 5; % View convergence progress every 5 iterations.
sol = sdf_nls(model,options);
sol.variables
sol.factors
```

Example 6: chaining factor structures. Factor structures can be chained so that a variable is transformed by a sequence of functions. In the following example, we compute a CPD in which a subfactor of the first factor matrix is a nonnegative Toeplitz matrix and the last factor is known. To create a nonnegative Toeplitz subfactor, we will create a generator vector for the Toeplitz matrix, transform it with `struct_nonneg` so that it is nonnegative, and then transform it with `struct_toeplitz` to create a nonnegative Toeplitz subfactor. The factor A is defined as a matrix consisting of two subfactors: the top subfactor transforms the variable `atop` with `struct_nonneg`, followed by `struct_toeplitz`, and the bottom subfactor is simply `abtm`.

```
% Generate CPD, wherein
% - the first factor has a nonnegative Toeplitz subfactor.
% - the last factor is known.
I = 15;
R = 4;
U = cpd_rnd([I I I],R);
U{1}(1:R,1:R) = toeplitz(rand(4,1));
T = cpdgen(U);

% Define model variables.
model.variables.atop = randn(2*R-1,1);
model.variables.abtm = randn(I-R,R);
model.variables.b = randn(I,R);

% Define model factors.
% The first factor vertically concatenates two subfactors.
% The top subfactor is generated as a nonnegative Toeplitz matrix,
% i.e., A = [ struct_toeplitz(struct_nonneg(atop)) ; abtm ].
model.factors.A = {{'atop',@struct_nonneg,@struct_toeplitz}; ...
                   {'abtm'}};
model.factors.B = 'b';
% Third factor matrix is known.
model.factors.C = U{3};

% Define model factorizations.
model.factorizations.myfac.data = T;
model.factorizations.myfac.cpd = {'A','B','C'};

% Solve the SDF problem.
options.Display = 5; % View convergence progress every 5 iterations.
sol = sdf_nls(model,options);
```

Example 7: regularization. Next to the CPD and BTD models, SDF also includes two models which represent L1- and L2-regularization terms. By including the factorizations

```
% Use L2-regularization on the factors A and B.
model.factorizations.myreg2.data = {zeros(size(U{1})),zeros(size(U{2}))};
model.factorizations.myreg2.regL2 = {'A','B'};

% Use L1-regularization on C-ones(size(C)).
model.factorizations.myreg1.data = ones(size(U{3}));
model.factorizations.myreg1.regL1 = {'C'};
```

to the previous example, the terms $\frac{1}{2}\left\| \begin{bmatrix} \operatorname{vec}(A)^{\mathsf{T}} & \operatorname{vec}(B)^{\mathsf{T}} \end{bmatrix}^{\mathsf{T}} - 0 \right\|_2^2$ and $\frac{1}{2}\| \operatorname{vec}(C) - 1 \|_1$ are added to the objective function. The `data` field defaults to all zeros, if omitted.

6.2 Implementing a new factor structure

Function signature To add your own structure to impose on factors in an SDF model, create a function with the following function signature

```
function [x,state] = struct_mystruct(z,task)
end
```

Function evaluation This function will be called in three different ways by SDF algorithms. The first is simply evaluating the structure, given the variable `z`:

```
[x,state] = struct_mystruct(z)
```

From here on, we will assume the transformation maps `z` to `x = sqrt(z)`. Thus, an early implementation could look like

```
function [x,state] = struct_mystruct(z,task)
if nargin < 2 || isempty(task), x = sqrt(z); state = []; end
end
```

Right Jacobian-vector product Once the function has been evaluated at a certain `z`, it will be called several times to evaluate the Jacobian-vector product

$$\frac{\mathrm{d}\,\mathrm{vec}(\sqrt{z})}{\mathrm{d}\,\mathrm{vec}(z)^{\mathsf{T}}} \cdot \boldsymbol{r}$$

at `z`, given a vector $\boldsymbol{r}$ stored in `task.r`. The latter will not actually be stored as a vector, but rather in the same format as the input variable `z`, be it an array or (nested) cell array of arrays. Likewise, the result of the right Jacobian-vector product `x` should be in the same format as the output of the evaluation `struct_mystruct(z)`. Moreover, the second output `state` from the function evaluation stage can be used to store computations in, which will then be made available as fields in `task`. Taking this into account, our transformation could look like

```
function [x,state] = struct_mystruct(z,task)
if nargin < 2 || isempty(task)
    x = sqrt(z);
    state.dz = 1./(2*x);
elseif ~isempty(task.r)
    % Here, task.dz is equal to 1/(2*sqrt(z)), which we stored earlier.
    x = task.dz.*task.r;
    state = [];
end
end
```

Left Jacobian-vector product Finally, the function will also be called to evaluate the left Jacobian-vector product

$$\left(\frac{\partial\,\mathrm{vec}(\sqrt{z})}{\partial\,\mathrm{vec}(z)^{\mathsf{T}}}\right)^{\mathsf{H}} \cdot \boldsymbol{\ell} + \overline{\left(\frac{\partial\,\mathrm{vec}(\sqrt{z})}{\partial\,\mathrm{vec}(\overline{z})^{\mathsf{T}}}\right)^{\mathsf{H}} \cdot \boldsymbol{\ell}}$$

where the partial derivative with respect to z ($\overline{z}$) treats $\overline{z}$ (z) as constant (cf. Section 7.1) and the vector ℓ is stored in `task.l`. The output `x` should be of the same format as the input `z`. Here, the second term in the left Jacobian-vector product is zero, and the full implementation of the structure becomes

```
function [x,state] = struct_mystruct(z,task)
if nargin < 2 || isempty(task)
    x = sqrt(z);
    state.dz = 1./(2*x);
elseif ~isempty(task.r)
    x = task.dz.*task.r;
    state = [];
elseif ~isempty(task.l)
    x = conj(task.dz).*task.l;
    state = [];
end
end
```

Remark Note that currently the `sdf_nls` family of algorithms does not accept nonanalytic structures, i.e., structures for which the second term in the left Jacobian-vector product is nonzero. The `sdf_minf` family does support nonanalytic structures, however. For example, `sdf_nls` currently does not support a structure of the form $z * \overline{z}$, but does support the structure $z * z$. Since `sdf_minf` does not use the right Jacobian-vector product, a partial workaround is to implement the structure $z * z$ for the right Jacobian-vector product so that `sdf_nls` can still use this structure if z is real-valued. At the same time, the left Jacobian-vector product can be based on the original structure $z * \overline{z}$ so that `sdf_minf` can be used for both real and complex z.

7 Complex optimization

Optimization problems An integral part of Tensorlab comprises optimization of real functions in complex variables [12]. 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`, `minf_ncg` and `minf_sr1cgs`) 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

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

are also supported (cf. `nlsb_gndl`). Furthermore, when a real solution $\boldsymbol{z} \in \mathbb{R}^n$ 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

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

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 $O(n^6)$ 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

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

respectively.

Remark 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.

7.1 Complex derivatives

7.1.1 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

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

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

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

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

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

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

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

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.

7.1.2 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

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

An advantage of using the *i*-trick is that it can compute first-order derivatives accurately 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 $\frac{\partial f_3}{\partial Y}$. Often, 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 $f(z)$ is a real-valued scalar function of complex variables, `deriv` can compute its scaled conjugate cogradient $\boldsymbol{g}(z)$. For example, let

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

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 $\boldsymbol{g}(z)$ 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))
```

Remark In case of doubt, use `deriv(f,z,[],'gradient')` to compute the scaled conjugate cogradient. Running `deriv(f,z)` will attempt to use the i-trick when `z` is real, which can be substantially more accurate, but should only be applied when `f` is analytic.

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

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

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

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

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

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 $\begin{bmatrix} J_2(X,Y) & J_2^c(X,Y) \end{bmatrix}$. 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))
```

7.2 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 $\mathcal{F}(z, \overline{z})$, which will be $\mathcal{F}_{\text{lyap}}(U, V)$ for the remainder of this section (cf. Section 7). 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 $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial z^{\mathsf{T}}}$, assuming $\mathcal{F}$ is analytic in z. The second case, `'Jacobian-C'`, corresponds to approximating the complex Jacobian consisting of the Jacobian $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial z^{\mathsf{T}}}$ and conjugate Jacobian $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \overline{z}^{\mathsf{T}}}$, where $z \in \mathbb{C}^n$. 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

$$\frac{\partial \operatorname{vec}(\mathcal{F}_{\text{lyap}})}{\partial \begin{bmatrix} \operatorname{vec}(U)^{\mathsf{T}} & \operatorname{vec}(V)^{\mathsf{T}} \end{bmatrix}} = \begin{bmatrix} (V^{\mathsf{T}} \otimes A) + (\overline{A} \cdot V^{\mathsf{T}} \otimes \mathbb{I}) & (\mathbb{I} \otimes A \cdot U) + (\overline{A} \otimes U) \end{bmatrix}. \qquad \text{(J-lyap)}$$

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 z. For problems for which the residual function $\mathcal{F}$ depends on both z and $\overline{z}$, the complex Jacobian can be supplied with the field `dF.dzc`. See Section 7.1 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[1] in z (i.e., it is not a function of $\overline{z}$) 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 $J := \frac{\partial \operatorname{vec}(\mathcal{F})}{\partial z^{\mathsf{T}}}$ 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 $\frac{\partial f}{\partial \overline{z}} \equiv J^{\mathsf{H}} \cdot \operatorname{vec}(\mathcal{F})$ are also necessary. Skipping the derivation of the gradient and Jacobian's Gramian, the implementation could look like

```
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)
```

[1]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.

```
        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 the option

```
% 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 $J \cdot \boldsymbol{x}$, $J^{\mathrm{H}} \cdot \boldsymbol{y}$, $J_c \cdot \boldsymbol{x}$ and $J_c^{\mathrm{H}} \cdot \boldsymbol{y}$, 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);
```

Remark 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.

7.3 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. Instead 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 7). 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 7.1 for more information on complex derivatives.

Remark 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.

8 Global minimization of bivariate 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 [11, 15]. 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 7.1). 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.

8.1 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

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

For example, the six-hump camel back function [6]

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

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 8.1. 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

```
[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))`.

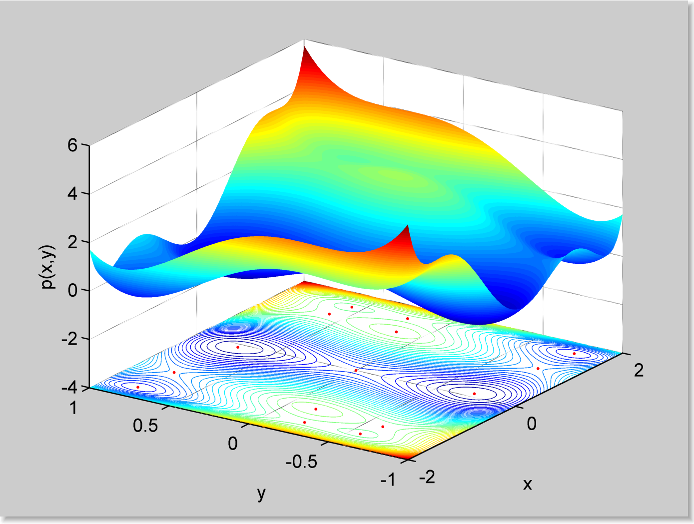


Figure 8.1: The six-hump camel back function and its stationary points as red dots.

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. The following example generates a random bivariate rational function and computes its stationary points:

```
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);
```

The inline equivalent of this example is `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

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

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);
```

Remark 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`.

8.2 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 $\mathbb{C}^2$ that contains no solution other than (x^*, y^*). Some systems may have solutions that are isolated in $\mathbb{R}^2$, but not in $\mathbb{C}^2$.

Remark 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 8.2, 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.

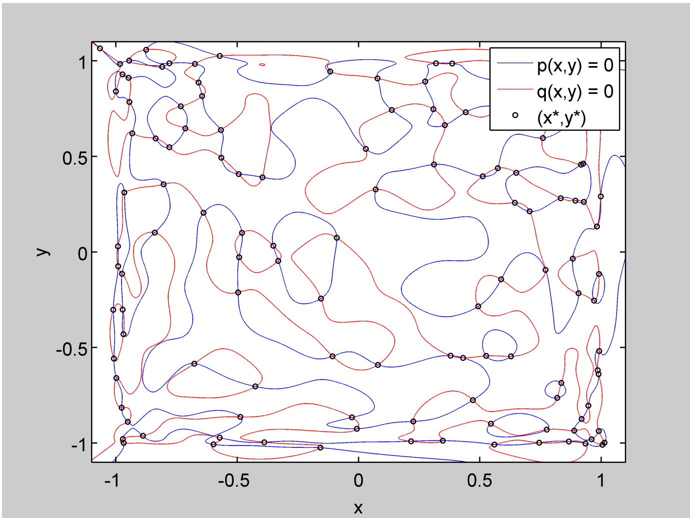


Figure 8.2: A system of bivariate polynomials $p(x,y) = q(x,y) = 0$ of total degree 20.

The following example generates a random bivariate system, computes its isolated solutions and plots the results:

```
p = tril(randn(5));
q = triu(randn(5));
options.Plot = true; % Note that plotting can take a while.
xy = polysol2(p,q,options);
```

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) OT/10/038, (b) PF/10/002 Optimization in Engineering (OPTEC), (2) the Research Foundation Flanders (FWO): G.0828.14N, and by (3) 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) G.0427.10N, (b) G.0830.14N, (c) G.0881.14N and by (3) the Belgian Network DYSCO: IUAP P7/19.

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 Journal on Matrix Analysis and Applications*, 30(3):1022–1032, 2008.

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

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

[6] 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.

[7] 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.

[8] F. L. Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematical Physics*, 6(1):164–189, 1927.

[9] F. L. Hitchcock. Multiple invariants and generalized rank of a p-way matrix or tensor. *Journal of Mathematical Physics*, 7(1):39–79, 1927.

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

[11] L. Sorber, I. Domanov, M. Van Barel, and L. De Lathauwer. Exact line and plane search for tensor optimization. ESAT-STADIUS Internal Report 13-02, Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium, 2013.

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

[13] 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 Journal on Optimization*, 23(2):695–720, 2013.

[14] L. Sorber, M. Van Barel, and L. De Lathauwer. Structured data fusion. ESAT-STADIUS Internal Report 13-177, Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium, 2013.

[15] L. Sorber, M. Van Barel, and L. De Lathauwer. Numerical solution of bivariate and polyanalytic polynomial systems. *SIAM Journal on Numerical Analysis*, 2014. Accepted.

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

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