Tensorlab User Guide

Release 3.0

Nico Vervliet
Otto Debals
Laurent Sorber
Marc Van Barel
Lieven De Lathauwer

Tensorlab is a Matlab toolbox for tensor computations and complex optimization. It provides algorithms for (coupled) tensor decompositions of dense, sparse, incomplete and structured tensors with the possibility of imposing structure on the factors, as well as a tensorization framework and visualization methods.

March 28, 2016

CONTENTS

1 Getting started **2**
1.1 Installation . . . 2
1.2 Requirements . . . 2
1.3 Contents.m . . . 2

2 Datasets: dense, incomplete, sparse and structured **8**
2.1 Tensor representations . . . 8
2.2 Tensor operations . . . 13
2.3 Visualizing higher-order datasets . . . 16

3 Canonical polyadic decomposition **23**
3.1 Problem and tensor generation . . . 23
3.2 Computing the CPD . . . 24
3.3 Choosing the number of rank-one terms R . . . 29
3.4 Overview of algorithms . . . 30
3.5 Further reading . . . 32

4 Decomposition in multilinear rank-$(L_r, L_r, 1)$ terms **33**
4.1 Problem and tensor generation . . . 33
4.2 Computing the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms . . . 35
4.3 Overview of algorithms . . . 39
4.4 Further reading . . . 40

5 Multilinear singular value decomposition and low multilinear rank approximation **41**
5.1 Problem and tensor generation . . . 41
5.2 Computing an MLSVD . . . 43
5.3 Computing an LMLRA . . . 44
5.4 Choosing the size of the core tensor . . . 47
5.5 Overview of algorithms . . . 48
5.6 Further reading . . . 49

6 Block term decomposition **50**
6.1 Problem and tensor generation . . . 50
6.2 Computing a BTD . . . 52
6.3 Overview of algorithms . . . 53
6.4 Further reading . . . 54

7 Tensorization techniques **55**
7.1 Hankelization . . . 56
7.2 Löwnerization . . . 62
7.3 Segmentation . . . 64
7.4 Decimation . . . 68
7.5 Computing second-order statistics . . . 69
7.6 Computing higher-order statistics . . . 70

8 Structured data fusion **71**
8.1 Elements of an SDF model . . . 72
8.2 Checking and solving a model . . . 72
8.3 Imposing coupling and symmetry . . . 75
8.4 Imposing structure on factors . . . 78
8.5 Factorization and regularization types . . . 84
8.6 Further reading . . . 86

9 Structured data fusion: examples **87**
9.1 Example 1: nonnegative symmetric CPD . . . 87
9.2 Example 2: structured coupled matrix factorization . . . 89
9.3 Example 3: an orthogonal factor . . . 91
9.4 Example 4: decomposition in multilinear rank-$(L_r, L_r, 1)$ terms . . . 92
9.5 Example 5: constants . . . 95
9.6 Example 6: chaining factor structures . . . 97
9.7 Example 7: regularization . . . 98
9.8 Example 8: compression based initialization . . . 99
9.9 Example 9: advanced coupling . . . 100

10 Structured data fusion: advanced concepts **106**
10.1 Using indexed notations . . . 106
10.2 The `transform` field . . . 108
10.3 Using implicit factors . . . 110
10.4 High accuracy for NLS algorithms . . . 111

11 Structured data fusion: implementing a transformation **114**
11.1 Elements in a transformation . . . 114
11.2 Examples . . . 116
11.3 Storing intermediate results . . . 117
11.4 Non-numerical variables . . . 119

12 Structured data fusion: specification of domain specific language **121**
12.1 Definitions . . . 121
12.2 The model struct . . . 124
12.3 The variables field . . . 124
12.4 The factors field . . . 124
12.5 The factorizations field . . . 125
12.6 The transform field . . . 128
12.7 The options field . . . 130

13 Complex optimization **132**
13.1 Complex derivatives . . . 133
13.2 Nonlinear least squares . . . 138
13.3 Unconstrained nonlinear optimization . . . 142

14 Global minimization of bivariate functions **144**
14.1 Analytic bivariate polynomials . . . 144
14.2 Polyanalytic univariate polynomials . . . 144
14.3 Polynomials and rational functions . . . 145
14.4 Bivariate polynomials and rational functions . . . 145
14.5 Polyanalytic polynomials and rational functions . . . 147

15 Acknowledgements **149**

References **150**

Tensorlab provides various tools for tensor computations, (coupled) tensor decompositions and complex optimization. In Tensorlab, datasets are stored as (possibly incomplete, sparse or structured) vectors, matrices and higher-order tensors, possibly obtained after tensorizing lower-order data. By mixing different types of tensor decompositions and factor transformations, a vast amount of factorizations can be computed. Users can choose from a library of preimplemented transformations and structures, including nonnegativity, orthogonality, Toeplitz and Vandermonde matrices to name a few, or even define their own factor transformations.

Chapter 2 covers the representation of dense, incomplete, sparse and structured datasets in Tensorlab and the basic operations on such tensors. Tensor decompositions of both real and complex tensors such as the canonical polyadic decomposition (CPD), the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms, the low multilinear rank approximation (LMLRA), the multilinear singular value decomposition (MLSVD) and the block term decomposition (BTD) are discussed in Chapters 3, 4, 5 and 6, respectively. Chapter 7 discusses the topic and framework of tensorization, in which lower-order datasets can be transformed to higher-order data. A basic introduction in structured data fusion (SDF) is given in Chapter 8, with which multiple datasets can be jointly factorized while imposing structure on the factors using factor transformations. Examples of SDF are given in Chapter 9. Chapter 10 discusses the more advanced concepts within SDF, while Chapters 11 explains the implementation of a transformation. Chapter 12 gives a full specification of the domain specific language used by SDF. Many of the algorithms to accomplish the decomposition tasks are based on complex optimization, that is, optimization of functions in complex variables. Chapter 13 introduces the necessary concepts and shows how to solve different types of complex optimization problems. Finally, Chapter 14 treats global optimization of bivariate (and polyanalytic) polynomials and rational functions, which appears as a subproblem in tensor optimization.

The first version of Tensorlab was released in February 2013, comprising a wide range of algorithms for tensor decompositions. Tensorlab 2.0 was released in January 2014, introducing the structured data fusion framework. The current version of Tensorlab, version 3.0, is released in March 2016. Tensorlab 3.0 introduces a tensorization framework, offers more support for large-scale and structured datasets and enhances the performance and user-friendliness of the SDF framework.

More specifically in Tensorlab 3.0, different tensorization and detensorization techniques are introduced. Efficient representations of structured tensors are supported, which improve the speed of many operations and decompositions. A number of algorithms dealing with large-scale datasets are provided, as well as a family of algorithms for the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms. The SDF framework has been redesigned to allow greater flexibility in the model definition and to improve the performance. Further, a tool has been made available to verify and correct an SDF model, and a method is provided for visual exploration of high-dimensional data of arbitrary order. More details can be found in the release notes[1].

To illustrate the power of the toolbox in an accessible manner, a number of demos[2] have been developed accompanying this manual. These demos discuss different case studies using tensor decompositions such as multidimensional harmonic retrieval, independent component analysis (ICA) and the prediction of user involvement based on a GPS dataset.

A PDF version of the user guide can be found here[3]. This toolbox can be cited as in [24].

[1] http://www.tensorlab.net/versions.html#3.0
[2] http://www.tensorlab.net/demos
[3] http://www.tensorlab.net/userguide3.pdf

CHAPTER

ONE

GETTING STARTED

1.1 Installation

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

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

1.2 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 versions lower than 3.8 are 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. As Octave v3.8 added support for nested subfunctions, Tensorlab is fully functional now.

1.3 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 3.0, 2016-03-28
%
% BLOCK TERM DECOMPOSITION
% Algorithms
%    btd_core     - Computational core for block term decomposition.
%    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.
%    frobbtdres   - Frobenius norm of residual for a BTD.
%
% CANONICAL POLYADIC DECOMPOSITION
% Algorithms
%    cpd          - Canonical polyadic decomposition.
%    cpd_als      - CPD by alternating least squares.
%    cpd_core     - Computational routines for CPD decomposition.
%    cpd_minf     - CPD by unconstrained nonlinear optimization.
%    cpd_nls      - CPD by nonlinear least squares.
%    cpd_rbs      - CPD by randomized block sampling.
%    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
%    cpd_crb      - Diagonal Cramér-Rao bound approximation for CPD.
%    cpderr       - Errors between factor matrices in a CPD.
%    cpdgen       - Generate full tensor given a polyadic decomposition.
%    cpdres       - Residual of a polyadic decomposition.
%    frobcpdres   - Frobenius norm of residual for polyadic decomposition.
%    rankest      - Estimate rank.
%
% COUPLED/SYMMETRIC CANONICAL POLYADIC DECOMPOSITION
% Algorithms
%    ccpd_core   - Computational routines for coupled/symmetric CPD
%                  decomposition.
%    ccpd_minf   - Coupled CPD by unconstrained nonlinear optimization.
%    ccpd_nls    - Coupled/symmetric CPD by nonlinear least squares.
%
% 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
%                   trust region.
% 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_core   - Computational core for 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.
%    mlsvds       - Multilinear singular value decomposition for sparse tensors.
%    mlsvd_rsi    - Sequentially truncated MLSVD using randomized subspace
%                   iteration.
% 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.
%    froblmlrares - Frobenius norm of residual of a LMLRA.
%    mlrank       - Multilinear rank.
%    mlrankest    - Estimate multilinear rank.
%
% DECOMPOSITION IN MULTILINEAR RANK-(Lr,Lr,1) TERMS
% Algorithms
%    ll1          - Decomposition in LL1 terms.
%    ll1_core     - Computational routines for LL1 decomposition.
%    ll1_minf     - LL1 decomposition by nonlinear unconstrained
%                   optimization.
%    ll1_nls      - LL1 decomposition by nonlinear least squares.
% Initialization
%    ll1_gevd     - LL1 by generalized eigenvalue decomposition.
%    ll1_rnd      - Pseudorandom initialization for LL1 decomposition.
% Utilities
%    ll1convert   - Convert LL1 decomposition between CPD and BTD format.
%    ll1gen       - Generate full tensor given as a LL1 decomposition.
%    ll1res       - Residual for a LL1 decomposition.
%    frobll1res   - Frobenius norm of the residual of a LL1 decomposition.
%
% STRUCTURED DATA FUSION
% Language parser
```

```
%    sdf_check         - SDF language parser and syntax/consistency checker.
% Algorithms
%    ccpd_core         - Computational routines for coupled/symmetric CPD
%                        decomposition.
%    ccpd_minf         - Coupled CPD by unconstrained nonlinear optimization.
%    ccpd_nls          - Coupled/symmetric CPD using nonlinear least squares.
%    sdf_core          - Computational core for structured data fusion.
%    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_cauchy     - Cauchy matrix.
%    struct_const      - Keep parts of z constant.
%    struct_ctranspose - Complex conjugate transpose.
%    struct_diag       - Diagonal matrix.
%    struct_exp        - Matrix with columns as exponentials.
%    struct_fd         - Finite differences.
%    struct_gram       - Gramian matrix.
%    struct_hankel     - Hankel matrix.
%    struct_inv        - Matrix inverse.
%    struct_invsqrtm   - Matrix inverse square root.
%    struct_invtransp  - Matrix inverse transpose.
%    struct_kr         - Khatri-Rao-product of two or more matrices.
%    struct_kron       - Kronecker-product of two or more matrices.
%    struct_LL1        - Structure of third factor matrix in a LL1 decomposition.
%    struct_log        - Natural logarithm.
%    struct_matvec     - Matrix-vector and matrix-matrix product.
%    struct_nonneg     - Nonnegative array.
%    struct_nop        - No operation.
%    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_prod       - Hadamard product.
%    struct_rational   - Matrix with columns as rational functions.
%    struct_rbf        - Matrix with columns as sums of Gaussian RBF kernels.
%    struct_select     - Select entry from cell variable z.
%    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.
%
% TENSOR UTILITIES
% Structured tensors
%    detectstructure  - Detect structure in a tensor.
%    getstructure     - Determine the type of a tensor.
%    isvalidtensor    - Check if the representation of a tensor is correct.
% Products
%    contract         - Mode-n tensor vector contraction.
%    inprod           - Inner product of two tensors.
%    mtkronprod       - Matricized tensor Kronecker product.
%    mtkrprod         - Matricized tensor Khatri-Rao product.
%    outprod          - Outer vector/matrix/tensor product.
%    tmprod           - Mode-n tensor-matrix product.
% Utilities
%    fmt              - Format data set.
%    frob             - Frobenius norm.
%    ful              - Convert formatted data set to an array.
%    getorder         - Order of a tensor.
%    getsize          - Dimensions of a tensor.
%    noisy            - Generate a noisy version of a given array.
%    ttgen            - Generates full tensor from TT format.
%
% TENSORIZATION
% Deterministic tensorization
%    decimate         - Decimation of vectors, matrices or tensors.
%    hankelize        - Hankelization of vectors, matrices or tensors.
%    loewnerize       - Loewnerization of vectors, matrices or tensors.
%    mat2tens         - Tensorize a matrix.
%    segmentize       - Segmentation of vectors, matrices or tensors.
%    vec2tens         - Tensorize a vector.
% Deterministic detensorization
%    dedecimate       - Recover decimated signal(s).
%    dehankelize      - Recover the signal(s) from an (approximate) Hankel
%                       matrix/tensor.
%    deloewnerize     - Recover the signal(s) from an (approximate) Loewner
%                       matrix/tensor.
%    desegmentize     - Recover segmented signal(s).
%    tens2mat         - Matricize a tensor.
%    tens2vec         - Vectorize a tensor.
% Tensorization with statistics
%    dcov             - Covariance matrices along specific dimensions.
%    cum3             - Third-order cumulant tensor.
%    cum4             - Fourth-order cumulant tensor.
%    scov             - Shifted covariance matrices.
%    stcum4           - Fourth-order spatio-temporal cumulant tensor.
%    xcum4            - Fourth-order cross-cumulant tensor.
%
% UTILITIES
% Clustering
%    gap              - Optimal clustering based on the gap statistic.
```

```
%    kmeans           - Cluster multivariate data using the k-means++ algorithm.
% Polynomials
%    genpolybasis     - Polynomial basis.
%    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.
%    transform_poly   - Transform a polynomial.
% Various
%    crandn           - Complex normally distributed pseudorandom numbers.
%    dotk             - Dot product in K-fold precision.
%    fixedanglevect   - Vectors with fixed angle.
%    gevd_bal         - Generalized eigenvalue decomposition with balancing.
%    kr               - Khatri-Rao product.
%    kron             - Kronecker product.
%    sumk             - Summation in K-fold precision.
% 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.
%    visualize        - Visualize a higher-order tensor.
%    voxel3           - Visualize a third-order tensor with voxels.
```

CHAPTER

TWO

DATASETS: DENSE, INCOMPLETE, SPARSE AND STRUCTURED

A scalar is a tensor of order zero. Vectors and matrices are first- and second-order tensors, respectively. Arrays with three or more dimensions are called higher-order tensors.

Datasets can be *dense*, *sparse*, *incomplete* or *structured*. Tensorlab provides different representations for each of these cases, discussed in Section 2.1. These representations are developed in such a way that a number of operations can be computed efficiently, which is discussed in Section 2.2. In Section 2.3, different visualization techniques are illustrated.

2.1 Tensor representations

2.1.1 Dense tensors

A dense (or full) tensor is simply a MATLAB array, e.g., `A = randn(10,10)` or `T = randn(5,5,5)`. MATLAB supports a few basic operations on tensors, for example:

```
T = randn(10,20,30);
S = randn(10,20,30);
V = T + S;            % Elementwise addition
W = T.*S;             % Elementwise multiplication
```

2.1.2 Incomplete tensors

An incomplete tensor is a dataset in which some (or most) of the entries are unknown. An incomplete tensor is efficiently represented in Tensorlab by a structure that contains the necessary information. The efficient representation of an incomplete tensor of which the unknown entries are padded with `NaN` can be obtained by using the format command `fmt`. In the following example, a MATLAB array of size $9 \times 9 \times 9$ with only three known elements is replaced by its efficient representation:

```
T = NaN(9,9,9);
T(1,2,3) = -1;
T(4,5,6) = -2;
T(7,8,9) = -3;
T = fmt(T)
```

```
T =
    matrix: [9x81 double]
      size: [9 9 9]
```

```
        ind: [3x1 int64]
        val: [3x1 double]
incomplete: 1
    sparse: 0
        sub: {[3x1 int32]  [3x1 int32]  [3x1 int32]}
```

Note that if there are no entries equal to `NaN`, then `fmt(T)` returns `T` itself. On the other hand, from the set of known entries, their positions, the tensor's size and an incomplete flag, the efficient representation can be constructed directly such as in the following example:

```
T = struct;
T.val = [-1 -2 -3];                % The known elements
T.sub = [1 2 3; 4 5 6; 7 8 9];     % Their positions
T.size = [9 9 9];                  % The size of the tensor
T.incomplete = true;               % The incomplete flag

T = fmt(T);                        % Complete the representation
```

Note that for obtaining a valid efficient representation, `fmt` needs to be used such that additional fields needed for computational purposes are inserted, e.g., `T.matrix`. Alternatively, the user may supply linear indices with `T.ind` instead of subindices with `T.sub`. To convert linear indices to subindices and vice versa, see the MATLAB functions `ind2sub` and `sub2ind`, respectively.

Given the representation of an incomplete tensor, the MATLAB array with `NaN` values for the unknown entries can be obtained using the command `ful`:

```
T = struct;
T.val = [1 2 3];
T.ind = [1 5 9];
T.size = [3 3];
T.incomplete = true;
T = fmt(T);
S = ful(T)
```

```
S =
   1   NaN   NaN
 NaN     2   NaN
 NaN   NaN     3
```

2.1.3 Sparse tensors

A sparse tensor is a dataset in which most of the entries are zero, e.g., a large diagonal matrix. Tensorlab supports efficient representations of sparse tensors of which at least 95% of the entries are zero. The efficient representation can be obtained using the format command `fmt`:

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

```
T =
    matrix: [5x25 double]
      size: [5 5 5]
       ind: [5x1 int64]
```

```
        val: [5x1 double]
 incomplete: 0
     sparse: 1
        sub: {[5x1 int32]  [5x1 int32]  [5x1 int32]}
```

If less than 95% of the entries are zero, then `fmt` returns `T` itself. To construct an efficient representation of a sparse tensor directly, define a structure containing the tensor's nonzero elements in the same way as for incomplete tensors and set the sparse flag. Afterwards, `fmt` should be called to insert additional necessary fields for computational purposes and to obtain a valid efficient representation:

```
T.val = [-1 -2 -3];              % The non-zero elements
T.sub = [1 2 3; 4 5 6; 7 8 9];   % Their positions
T.size = [9 9 9];                % The size of the tensor
T.sparse = true;                 % The sparse flag

T = fmt(T);
```

The `ful` command can be used to convert the representation to the corresponding MATLAB array:

```
T = struct;
T.val = [1 2 3];
T.ind = [1 5 9];
T.size = [3 3];
T.sparse  = true;
T = fmt(T);
S = ful(T)
```

```
S =
  1     0     0
  0     2     0
  0     0     3
```

2.1.4 Structured tensors

Tensorlab 3.0 introduces additional support for structured tensors. Two groups of structured tensors can be handled: structured tensors resulting from the tensorization of data and tensors that are given under the form of an explicit factorization.

Structured tensors resulting from the tensorization of data

Consider a Hankel matrix, constructed from an arbitrary vector. Such a matrix has a special structure: each anti-diagonal (or skew-diagonal) is constant. For example, from the generating vector $\mathbf{v} = [1, 2, \ldots, 7]$, the following Hankel matrix $\mathbf{H}$ can be obtained:

$$\mathbf{H} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{bmatrix}.$$

One can exploit the structure of $\mathbf{H}$ when storing $\mathbf{H}$ and even when performing operations on $\mathbf{H}$. The former is done in this case by only storing the small generating vector $\mathbf{v}$, instead of the 16 elements of $\mathbf{H}$.

This Hankel-based mapping is a specific example of tensorization and the Hankel matrix is a specific example of a tensor obtained from a tensorization of some lower-order data. The concept of tensorization is covered in more

detail in Chapter 7. Tensorlab supports efficient representations of Hankel matrices and (block-)Hankel tensors of arbitrary order, as well as Löwner, segmentation-based and decimation-based structures obtained by tensorization.

Each efficient representation is a MATLAB structure array, analogous to the efficient representation for incomplete and sparse tensors. For example, the following is an efficient representation of the previously mentioned Hankel matrix $\mathbf{H}$:

```
H = hankelize(1:7,'full',false);
```

```
H =
       type: 'hankel'
        val: [7x1 double]
        dim: 1
      order: 2
        ind: 4
 ispermuted: 0
repermorder: [1 2]
       size: [4 4]
    subsize: [1x1 struct]
```

By calling `ful` on `H`, the dense matrix is obtained:

```
H = ful(H)
```

```
H =
 1     2     3     4
 2     3     4     5
 3     4     5     6
 4     5     6     7
```

Tensors given under the form of an explicit factorization

Consider a rank-1 tensor $\mathcal{T} \in \mathbb{C}^{100\times100\times100}$, which can be written as the outer product of three vectors $\mathbf{a} \in \mathbb{C}^{100}$, $\mathbf{b} \in \mathbb{C}^{100}$ and $\mathbf{c} \in \mathbb{C}^{100}$. Each element of $\mathcal{T}$ satisfies $t_{ijk} = a_i b_j c_k$. The 300 elements from $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ exactly determine all of the 1,000,000 elements of $\mathcal{T}$. Hence, instead of storing $\mathcal{T}$ and performing operations on $\mathcal{T}$, only the three vectors could be stored and used for calculations.

Tensorlab supports efficient operations on tensors represented by a factorization in rank-1 terms (CPD, see Chapter 3), with arbitrary rank, order and size. Tensorlab also supports efficient operations on tensors which are given under the form of a tensor train (TT) decomposition [23], a low multilinear rank approximation (LMLRA, see Chapter 5) or a block term decomposition (BTD, see Chapter 6, with as specific case the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms).

The efficient representations corresponding to each structure are a collection of factor matrices and core tensor(s). Consider an Nth-order tensor $\mathcal{T}$ of size $I_1 \times I_2 \times \cdots \times I_N$:

- For a CPD, the representation is a cell array containing the factor matrices of sizes $I_n \times R$ with R the number of rank-1 terms.
- For an LMLRA, the representation is a cell array containing (1) a cell array with the factor matrices of sizes $I_n \times R_n$, and (2) a core tensor of size $R_1 \times R_2 \times \cdots \times R_N$.
- For a BTD, the representation is a cell array with R entries, each representing a tensor with multilinear rank $(L_{r1}, L_{r2}, \ldots, L_{rN})$. The rth cell entry has $N+1$ elements: the first N elements represent the factor matrices of size $I_n \times L_{rn}$, while element $N+1$ represents the core tensor of size $L_{r1} \times L_{r2} \times \cdots \times L_{rN}$. Note that this is different from the LMLRA format as the factor matrices are not encapsulated in an additional cell.

- For a TT, the representation is a cell array with N entries. The first and last entry represent two factor matrices of sizes $I_1 \times R_1$ and $R_{N-1} \times I_N$. The other entries represent the cores of sizes $R_n \times I_{n+1} \times R_{n+1}$ for $n = 1, \ldots, N-2$.

While Tensorlab supports the efficient representation of TT decompositions, we refer to other toolboxes for the computation of such decompositions.

2.1.5 Useful commands for efficient representations

Besides generating the entire dense tensor given an efficient representation of an incomplete, sparse or structured tensor, the command `ful` provides an additional feature allowing the user to pass indices for specific elements of the tensor to be calculated or extracted. Given an efficient representation of the following Hankel matrix $\mathbf{H}$:

$$\mathbf{H} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{bmatrix},$$

the following returns the diagonal of $\mathbf{H}$ using linear indices:

```
ful(H,[1 6 11 16])
```

```
ans =
  1     3     5     7
```

Alternatively, subindexing can be used to extract fibers or slices. For example:

```
ful(H,2,':')
ful(H,2:3,':')
```

```
ans =
  2     3     4     5

ans =
  2     3     4     5
  3     4     5     6
```

Further, `getstructure` returns the type of efficient representation. The functions `getorder` and `getsize` return the order and the size of the tensor. Given a dense tensor, the function `detectstructure` will try to detect structure present in the dataset. Currently, only symmetry and Hankel structure is detected. If the detected structure corresponds to a supported efficient representation in Tensorlab, the representation is also returned. Finally, the function `isvalidtensor` determines if the given efficient representation is a valid tensor representation. In the following example, the tensor representation is not valid as the field `size` is not present:

```
T = struct;
T.val = [1 2 3];
T.ind = [1 2 3];
T.incomplete = true;
isvalidtensor(T)
```

2.2 Tensor operations

2.2.1 Matricization and tensorization

A dense tensor `T` can be flattened or unfolded 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)]`.

The commands `tens2mat` and `mat2tens` currently only support dense tensors. For other matricization and detensorization methods, we refer to Chapter 7.

2.2.2 Tensor-matrix product

In a mode-n tensor-matrix product, the tensor's mode-n fibers 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 $\mathtt{T} \cdot_{\mathtt{mode(1)}} \mathtt{U1} \cdot_{\mathtt{mode(2)}} \mathtt{U2} \cdots$. 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, a cell array is not necessary. 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);
```

Note that `tmprod` currently only supports dense tensors.

The `contract(T,v,n)` command computes contractions between the tensor `T` with the vectors `v` in the modes given by `n`.

2.2.3 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 $I \times J$ and $K \times L$, respectively, then the Kronecker product of A and B is the $IK \times JL$ matrix

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

More generally, `kron(A,B,C,...)` and `kron(U)` compute the Kronecker products $((\mathtt{A} \otimes \mathtt{B}) \otimes \mathtt{C}) \otimes \cdots$ and $((\mathtt{U\{1\}} \otimes \mathtt{U\{2\}}) \otimes \mathtt{U\{3\}}) \otimes \cdots$, respectively.

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

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

More generally, `kr(A,B,C,...)` and `kr(U)` compute the 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.

Note that `kr` and `kron` currently only support dense tensors.

2.2.4 Matricized-tensor times Kronecker and Khatri–Rao product

Matricized-tensor times Kronecker product

Often, algorithms for tensor decompositions do not explicitly need matricized tensors or Kronecker products, but rather need the following matricized tensor Kronecker product:

```
tens2mat(T,n)*conj(kron(U([end:-1:n+1 n-1:-1:1])))
```

where `kron(...)` 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.

Note that for `n=0`, the efficiently implemented product gives the same result as the following:

```
T(:).'*conj(kron(U(end:-1:1)))
```

For improved computational performance, it is recommended that the two largest dimensions of `T` are the first and last modes of `T`. The command `mtkronprod` supports dense, sparse, incomplete and structured tensors.

Matricized-tensor times Khatri–Rao product

Similarly, algorithms for tensor decompositions usually do not explicitly need matricized tensors or Khatri–Rao products, but rather the following matricized tensor Khatri–Rao product:

```
tens2mat(T,n)*conj(kron(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.

For `n=0`, the efficiently implemented product gives the same result as the following:

```
T(:).'*conj(kr(U(end:-1:1)))
```

For improved computational performance, it is recommended that the two largest dimensions of `T` are the first and last modes of `T`. The command `mtkrprod` supports dense, sparse, incomplete and structured tensors.

2.2.5 Frobenius norm

The Frobenius norm of a tensor is the square root of the sum of square moduli 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')`.

The command supports dense, sparse, incomplete and structured tensors. For structured tensors, the norm is computed in a very efficient way. Consider a third-order Hankel tensor H of size $334 \times 334 \times 334$, constructed using the `hankelize` method (see Chapter 7). We compare the time needed for the computation of the Frobenius norm given the dense tensor `H` with the time needed for the computation given its efficient representation `Heff`:

```
[H,Heff] = hankelize(linspace(0,1,1000),'order',3);
tic; disp(frob(H)); toc;        % Using the dense tensor H
tic; disp(frob(Heff)); toc;     % Using the efficient representation of H
```

```
3.2181e+03
Elapsed time is 0.026401 seconds.

3.2181e+03
Elapsed time is 0.000832 seconds.
```

2.2.6 Inner and outer product

Given two tensors `T1` and `T2`, the command `inprod(T1,T2)` computes the inner product `T2(:)'*T1(:)`. The result is a scalar. The command supports dense, sparse, incomplete and structured tensors for both `T1` and `T2`.

The command `outprod(T1,T2)` computes the outer product of two tensors `T1` and `T2`. If `T1` has size $I_1 \times \cdots \times I_M$ and `T2` has size $J_1 \times \cdots \times J_N$, then the resulting tensor has size $I_1 \times \cdots \times I_M \times J_1 \times \cdots \times J_N$. If `T1`, respectively `T2`, is a row or column vector, the singleton dimension is discarded. Currently, the command `outprod` supports only dense tensors.

2.2.7 Noisy versions of tensors and representations

The `noisy` command can be used to create a noisy version of dense, incomplete, sparse and structured tensors given a signal-to-noise ratio. For example,

```
T = randn(10,10,10);
SNR = 20;
That = noisy(U,SNR);
frob(That-T)/frob(T)
```

```
ans =
  0.1000
```

Note that for efficient representations of incomplete, sparse and structured tensors, noise is added to the variables of the representation rather than to the full tensor. For example, given the efficient representation `Heff` of a 3×3 Hankel matrix `H`, a Hankel matrix is returned with constant perturbations along the anti-diagonals:

```
[H,Heff] = hankelize(1:5);
SNR = 20;
Heffhat = noisy(Heff,SNR);
H
ful(Heffhat)
```

```
H =
  1     2     3
  2     3     4
  3     4     5


 ans =
  1.2691    2.2507    3.3571
  2.2507    3.3571    3.5825
  3.3571    3.5825    4.9737
```

The `noisy` command can be used for cell arrays as well. Noise is then added on every entry of the cell. 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`. This is equivalent to adding noise to the efficient representation of factorized tensors as discussed in the previous paragraph.

2.3 Visualizing higher-order datasets

Tensorlab offers four methods to visualize higher-order datasets. The methods `slice3`, `surf3` and `voxel3` accept third-order datasets, while the `visualize` method accepts tensors of any order. The following example demonstrates the first three methods on the amino acids dataset [16]:

```
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 plots are shown in Fig. 2.1, Fig. 2.2 and Fig. 2.3, respectively.

`visualize` is a more advanced method for visual exploration of tensors of arbitrary order. It extracts and plots slices of order one, two or three. The other dimensions can then be varied to 'walk through' the data. The following example illustrates `visualize` for a tensor with smooth slices:

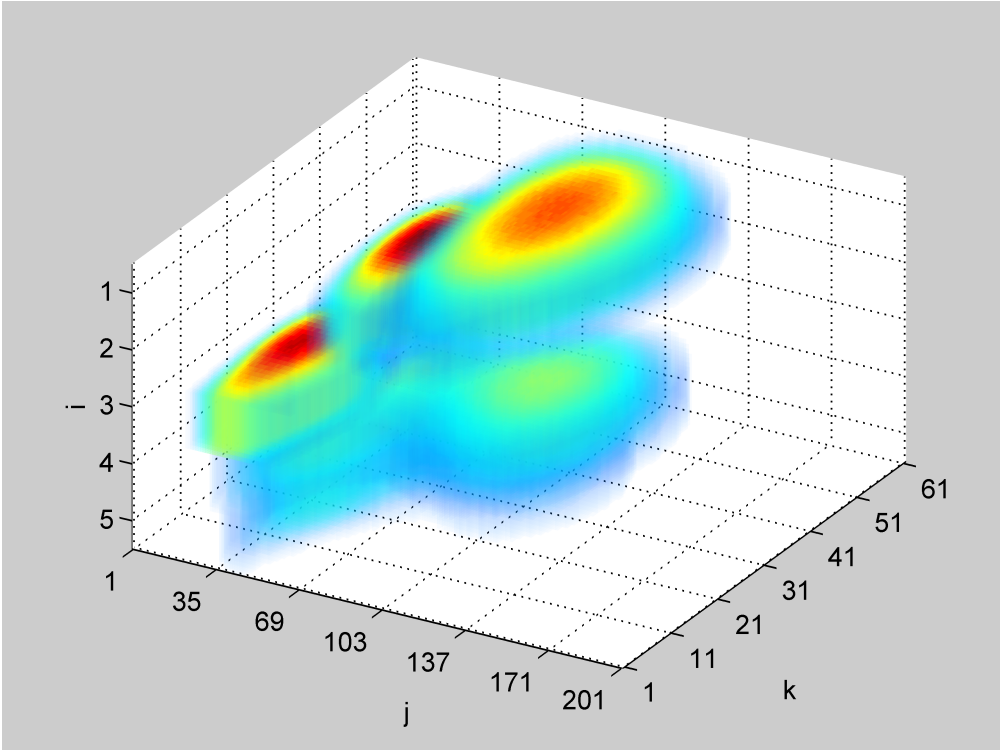

Fig. 2.1: Visualization of a third-order tensor using `voxel3`.

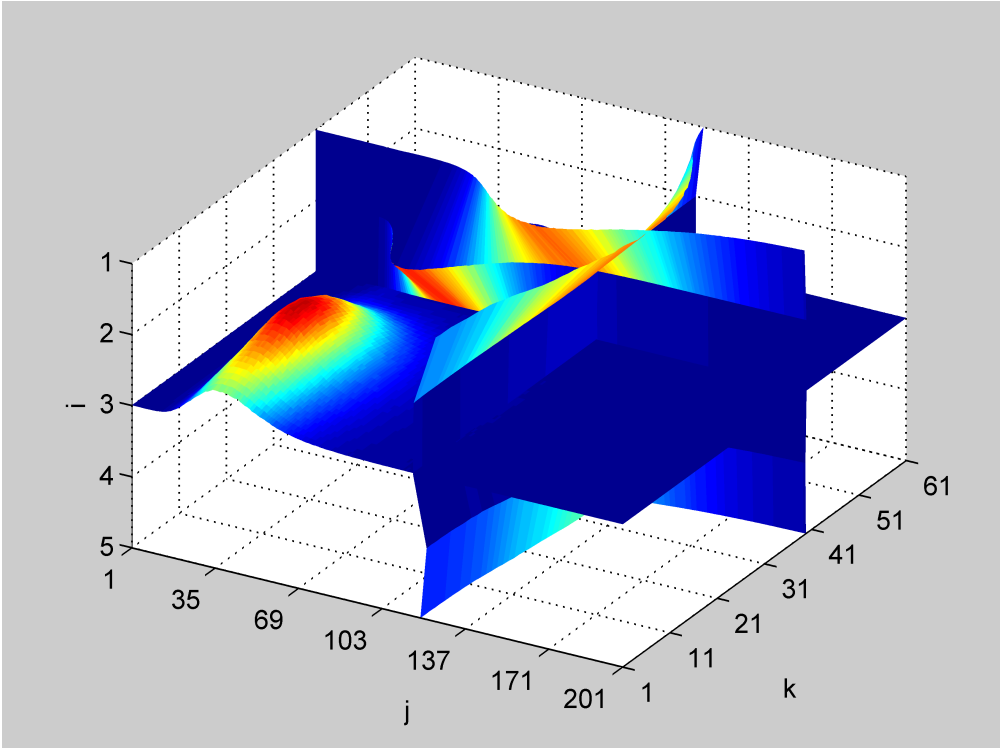

Fig. 2.2: Visualization of a third-order tensor using `surf3`.

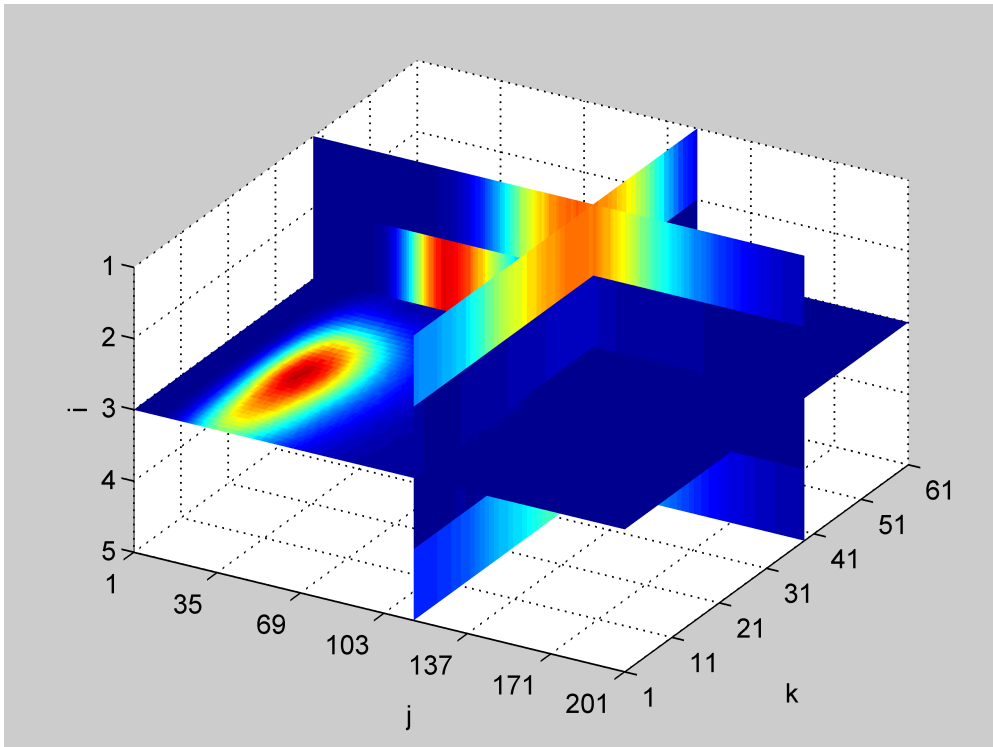

Fig. 2.3: Visualization of a third-order tensor using `slice3`.

```
R = 4; degree = 3;
sz = [10 15 20 25];
for n = 1:4
    % Use struct_poly to generate polynomial factor vectors
    U{n} = struct_poly(rand(R,degree+1), [], 1:sz(n));
end
T = cpdgen(U);

visualize(T)
```

The result is shown in Fig. 2.4. The check boxes can be used to select the dimensions that are plotted. By checking a single dimension, a mode-n vector is shown, checking two dimensions results in a surface plot, while checking three dimensions shows the `slice3` plot of the data. In Fig. 2.4 the slice $\mathcal{T}(:,:,1,1)$ is plotted, since $i_3 = i_4 = 1$. Changing the sliders or the text field for dimensions i_3 or i_4, allows the user to view other slices. When hovering a check box, label, slider or text field, a tooltip is shown with extra information.

The power of `visualize` lies in the ability to customize the plots and to view both the data tensor and its decomposition at the same time, as the following example demonstrates:

```
% Create dataset
R = 4; degree = 3;
sz = [10 15 20 25];
U = cell(1, 4); % Factor matrices
t = cell(1, 4); % Evaluation points
for n = 1:4
    % Use struct_poly to generate polynomial factor vectors in points t{n}
```

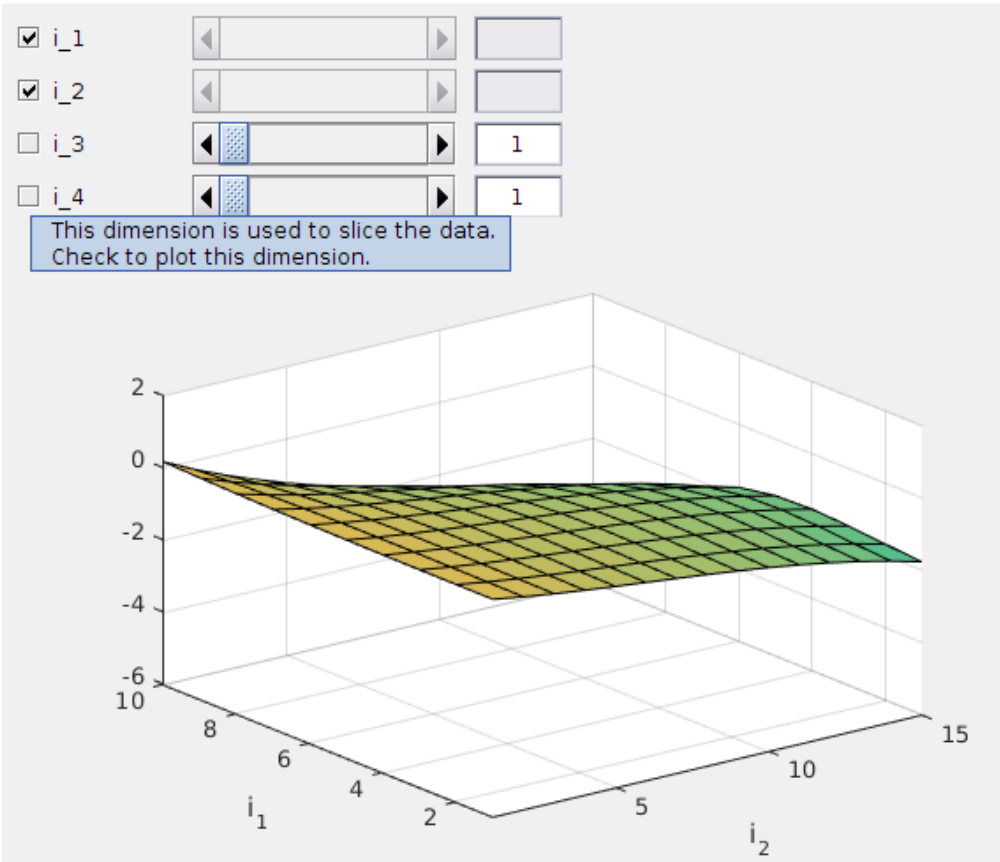

Fig. 2.4: Visualization of a fourth-order tensor using `visualize`.

```
    t{n} = linspace(-0.5, 0.5, sz(n));
    U{n} = struct_poly(rand(R,degree+1), [], t{n});
end
T = noisy(cpdgen(U), 30);

% Decompose tensor
Ures = cpd(T, R);

% Visualization
options = struct;
options.Original = T;             % Original data = T
options.Trace = true;             % Do not erase previous slices (only first-order)
options.DimensionTransform = t;   % Use points t for the axes
options.DimensionLabels = 't_%d'; % Use t_1, t_2 etc to label the dimensions

visualize(Ures, options)
```

The result after clicking a few sliders is shown in Fig. 2.5. Note that the dimensions have different labels and have the appropriate range as determined by the evaluation points. Going from a first-order to a second-order slice erases all traces again. By plotting the result and the data together, it is easier to see where the data is fitted well and where it is not. For an overview of the different options, see `help visualize`.

As a final example the `support` option is illustrated. Suppose we are only interested in the part of the data inside the cylinder $t_2^2 + t_4^2 \leq 0.5^2$, then we can prevent everything outside of this sphere to be plotted. To do this, an additional function indicating the region of interest is created:

```
function region = support(dim, ind)
% Compute support region for selected dimension and transformed indices
% dim is a logical array with a 1 if that dimension is plotted
% ind is a cell of (possibly transformed) indices
% region is vector or a matrix depending on sum(dim) == 1 or 2, respectively.

[t2,t4] = ndgrid(ind{2},ind{4});
region = t2.^2 + t4.^2 <= 0.5^2;
```

The following code block generates Fig. 2.6 for the same dataset as before:

```
options = struct;
options.DimensionTransform = t;   % Use points t for the axes
options.DimensionLabels = 't_%d'; % Use t_1, t_2 etc to label the dimensions
options.Support = @support;       % Use support region

visualize(Ures, options)
```

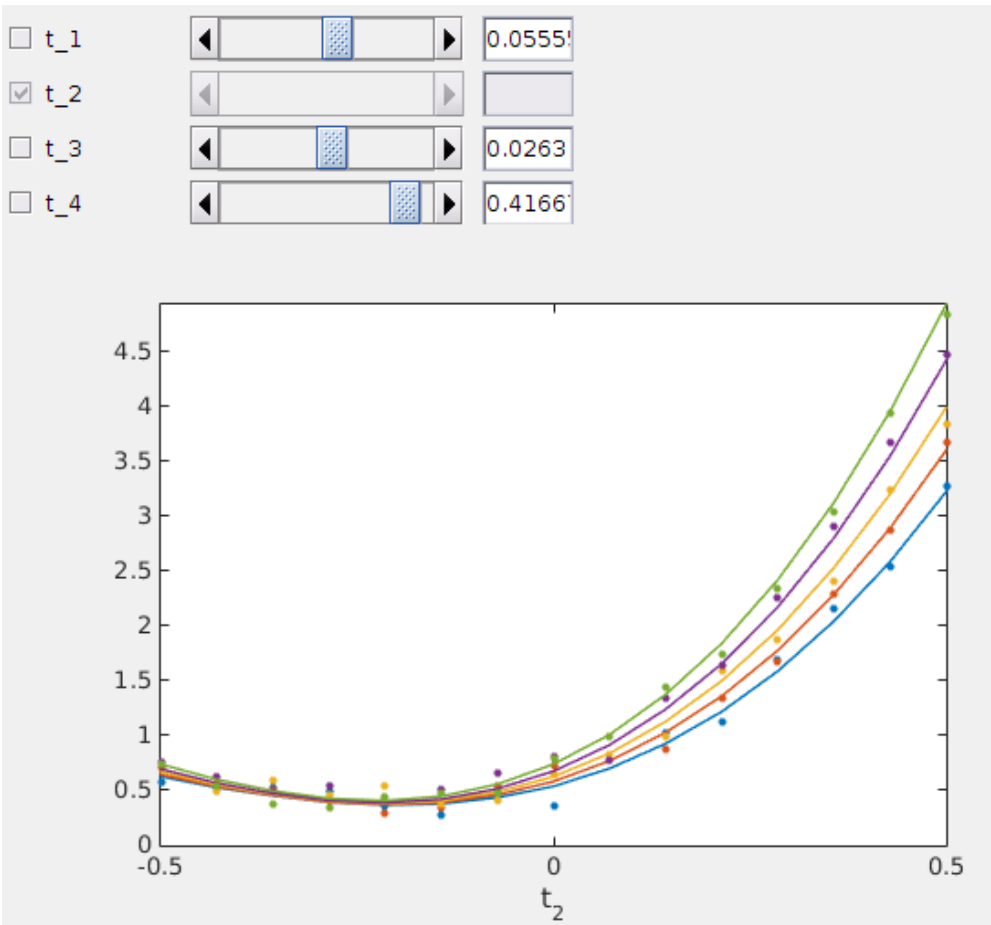

Fig. 2.5: Visualization of a fourth-order tensor using `visualize` with original data, tracing and dimension transforms after changing some sliders.

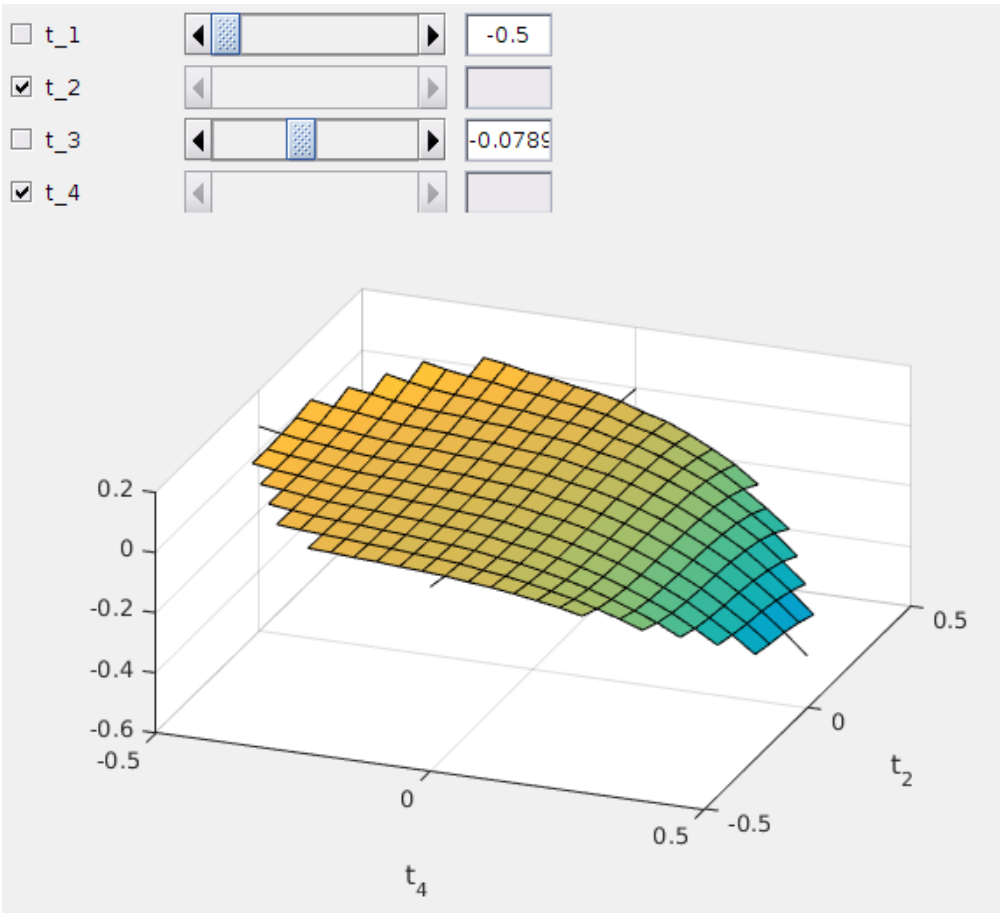

Fig. 2.6: Illustration of support regions for `visualize`.

CHAPTER

THREE

CANONICAL POLYADIC DECOMPOSITION

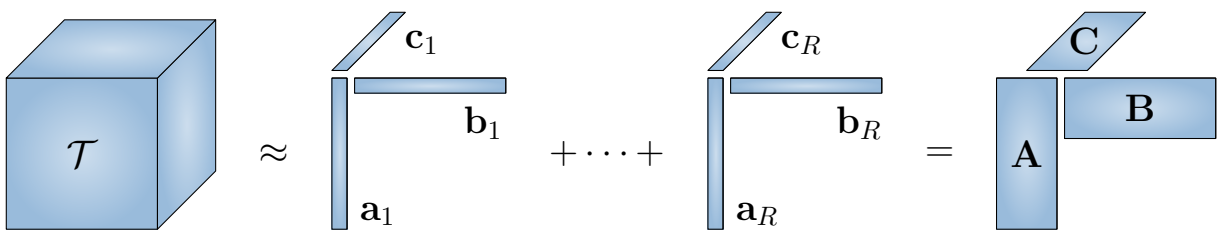

Fig. 3.1: A canonical polyadic decomposition of a third-order tensor.

The polyadic decomposition (PD) [5, 6, 10, 11, 12, 13] approximates a tensor with a sum of R rank-one tensors. If the number of rank-1 terms R is minimal, the decomposition is called canonical (CPD). Let $\mathcal{A} \otimes \mathcal{B}$ denote the outer product between an Nth-order tensor $\mathcal{A}$ and an Mth-order tensor $\mathcal{B}$, then $\mathcal{A} \otimes \mathcal{B}$ is the $(N+M)$th-order tensor defined by $(\mathcal{A} \otimes \mathcal{B})_{i_1 \cdots i_N\, j_1 \cdots j_M} = a_{i_1 \cdots i_N} \cdot b_{j_1 \cdots j_M}$. For example, let $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ be nonzero vectors in $\mathbb{R}^n$, then $\mathbf{a} \otimes \mathbf{b} \equiv \mathbf{a} \cdot \mathbf{b}^\mathsf{T}$ is a rank-one matrix and $\mathbf{a} \otimes \mathbf{b} \otimes \mathbf{c}$ is defined to be a rank-one tensor. Let $\mathcal{T}$ be a tensor of dimensions $I_1 \times I_2 \times \cdots \times I_N$, and let $\mathbf{U}^{(n)}$ be matrices of size $I_n \times R$ and $\mathbf{u}_r^{(n)}$ the rth column of $\mathbf{U}^{(n)}$, then

$$\mathcal{T} \approx \sum_{r=1}^{R} \mathbf{u}_r^{(1)} \otimes \mathbf{u}_r^{(1)} \otimes \cdots \otimes \mathbf{u}_r^{(N)}.$$

A visual representation of this decomposition in the third-order case is shown in Fig. 3.1.

3.1 Problem and tensor generation

3.1.1 Generating pseudorandom factor matrices

Tensorlab stores the factor matrices $\mathbf{U}^{(n)} \in \mathbb{C}^{I_n \times R}$, $n = 1, \ldots, N$, of an R-term PD of an Nth-order tensor as a cell array of length N, i.e., `U = {U{1},U{2},...,U{N}}`. To generate pseudorandom factor matrices $\mathbf{U}^{(n)}$, `cpd_rnd` can be used:

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

By default, `cpd_rnd` generates each factor `U{n}` as `randn(size_tens(n),R)`. Options can be given as a structure or as key-value pairs, e.g.:

```
% using a structure
options = struct;
options.Real = @rand;
options.Imag = @rand;
U = cpd_rnd(size_tens,R,options);

% using key-value pairs
U = cpd_rnd(size_tens, R, 'Real', @rand, 'Imag', @rand);
```

Each factor matrix `U{n}` is then generated as `rand(size_tens(n),R) + rand(size_tens(n),R)*1i`.

3.1.2 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);
```

The `ful` command can be used as well to generate the full tensor. Both `ful` and `cpdgen` also allow one to generate subtensors:

```
U = cpd_rnd([10 11 12], 5);
T = cpdgen(U);

t1 = cpdgen(U, [1 2 3 2 3]); % using linear indices
frob(t1 - T([1 2 3 2 3]))    % is zero

t2 = ful(U, ':', 2, ':');    % using subscripts (':' is a string!)
frob(t2 - T(:,2,:))          % is zero
```

3.2 Computing the CPD

A number of algorithms are available to compute the CPD of a tensor $\mathcal{T}$ of arbitrary order N. A deterministic algorithm using the generalized eigenvalue decomposition is implemented in `cpd_gevd` [14]. The `cpd_als`, `cpd_minf` and `cpd_nls` algorithms are optimization-based algorithms and use alternating least squares, nonlinear unconstrained optimization and nonlinear least squares, respectively. Finally, a high-level algorithm is implemented in `cpd`. This method automatically computes an initialization and uses this as a starting point for one or more optimization steps using an optimization-based algorithm. Because `cpd` is a higher-level algorithm, it is often the recommended method. All optimization-based algorithms accept dense, sparse, incomplete and structured tensors (see Chapter 2). If structured tensors are used, and a solution accurate up to machine precision is needed, make sure to read the note at the end of this chapter.

3.2.1 The principal method

The easiest way to compute the CPD of a dense, sparse, incomplete or structured tensor $\mathcal{T}$ in R rank-one terms, is to call `cpd(T,R)`. For example,

```
% Generate pseudorandom factor matrices U0 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 and decomposes a real rank-4 tensor. Internally, `cpd` performs a number of steps to provide a good initialization and to reduce the computational cost of the decomposition. First, `cpd` *compresses the tensor* using a low multilinear rank approximation, e.g., using `mlsvd_rsi` (see Chapter 5), if it is worthwhile. Then `cpd` chooses a method to *generate an initialization* `U0`, e.g., `cpd_gevd`, after which `cpd` *executes an algorithm* to compute the CPD given the initialization, e.g., `cpd_nls`. Finally `cpd` uncompresses the tensor and *refines the solution* (if compression was applied). Depending on the kind of input tensor `T`, additional substeps are performed. If `T` is a dense tensor, the command `detectstructure` is used to detect if there is any structure to be exploited during the decomposition (see `ExploitStructure` option). If structure is detected, the dense tensor is converted to its efficient representation. If `T` is the efficient representation of a sufficiently small structured tensor, `T` is expanded to allow the use of better compression and initialization algorithms (see `ExpandLimit` option).

The command `cpd(T, R)` automatically takes care of the initialization for the chosen optimization algorithm. Alternatively an initial solution can be provided using

```
U0   = cpd_rnd([10 11 12], R);
Uhat = cpd(T, U0);
```

Note that `cpd_gevd` is a deterministic algorithm, unless slices are combined randomly (see `Slices` option). Hence, if this algorithm is used for initialization in `cpd`, the result `Uhat` is always the same for multiple runs for the same tensor $\mathcal{T}$, rank R and options. Changing the initialization method to, e.g., `cpd_rnd`, is useful when different initializations are needed. The following example uses the `cpd_gevd` algorithm as initialization method for the first run, and uses a random initialization for the other runs:

```
init = @cpd_gevd;
results = cell(1, 5);
for n = 1:5
    results{n} = cpd(T, R, 'Initialization', init);
    init = @cpd_rnd;
end
```

The next subsection explains the different options for the `cpd` routine.

3.2.2 Setting the options

The different steps in `cpd` are customizable by using the different options (see `help cpd` for more information). Options can be provided using an `options` structure, or using key-value pairs, 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 function tolerance stop criterion
options.AlgorithmOptions.TolX   = 1e-12; % Set step size tolerance stop criterion
Uhat = cpd(T,R,options);
```

The structures `InitializationOptions`, `AlgorithmOptions` and `RefinementOptions` are passed as option structures to the algorithms corresponding to initialization, algorithm and refinement steps, respectively. In the example above `cpd` calls `cpd_als` as `cpd_als(T,options.Initialization(T,R),options.AlgorithmOptions)`.

Since Tensorlab 3.0 some frequently used options are automatically. For example `cpd(T, R, 'Display', 10)` sets `Display = true` for `cpd` and automatically passes the `Display = 10` option to `AlgorithmOptions` and `RefinementOptions`. Similarly, the `MaxIter`, `TolX`, `TolFun` and `CGMaxIter` options are passed automatically.

A full overview of the possible options is given in `help cpd`. The following options are worthwhile to be highlighted:

- `Complex`: If set to `'auto'`, a complex initialization is computed if the tensor `T` is complex, and a real initialization is computed otherwise. Set `Complex` to `true` if a complex solution is required, whether `T` is real or not.
- `ExpandLimit`: Allow a structured tensor `T` to be expanded for the compression and/or initialization steps if `prod(getsize(T)) <= ExpandLimit`. Set `ExpandLimit = 1` to make sure tensors are never expanded.
- `ExploitStructure`: If `'auto'` (the default) `cpd` tries to detect structure within the given dense tensor and uses its efficient representation for the decomposition of the uncompressed tensor if structure is found. Currently, only Hankel structure is detected. Set to `false` to disable structure detection. Set to `true` if structure detection should also be enabled for the compressed tensor.
- `UseCPDI`: If set to `true` specialized computational routines are used if `cpd_nls` is used to compute the CPD of an incomplete tensor. This option is useful if many entries are missing or the pattern of missing entries is not random [15].

3.2.3 Viewing the algorithm output

Each step may also output additional information specific to that step. For instance, most CPD algorithms such as `cpd_als` 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;
```

Setting the `Display` option to `true` or to a number greater than zero, prints additional information to the command window. For example:

```
U = cpd_rnd([10 11 12], 5);
T = cpdgen(U);
Tn = noisy(T, 20);
[Uhat, output] = cpd(Tn, 5, 'Display', 1);
```

```
Step 1: Compression is mlsvd_rsi (compression ratio 0.094697 < 0.5)...
        relative error = 0.0906931.
Step 2: Initialization is cpd_gevd (sum(R <= size(T)) >= 2)...
        relative error = 0.031175.
Step 3: Algorithm is cpd_nls on the compressed tensor... iterations = 6,
        relative error = 0.0207.
Step 4: Refinement is cpd_nls on the uncompressed tensor:
| Step 4a: Detecting structure... no structure detected.
| Step 4b: cpd_nls... iterations = 6, relative error = 0.092980
```

The four steps discussed in the previous section are clearly indicated. The `mlsvd_rsi` algorithm is used here to compress the tensor. Next the `cpd_gevd` method is used to compute an initialization for the compressed tensor. This initialization is used as starting point for the optimization method in step 3. Finally, the tensor is uncompressed again in step 4 and the solution is refined for the uncompressed tensor. In step 4a, `cpd` tries to detect structure to exploit in the computations, but no structure is found in this case.

In the next example, a sampled cosine function is Hankelized using the `hankelize` function (see Chapter 7). It can be shown that the rank of this real tensor is two in the complex field. A complex solution is computed by setting the `Complex` option to `true`.

```
T = hankelize(cos(2*pi*linspace(0,1,51)), 'order', 3);
options = struct;
options.Initialization = @cpd_rnd;
options.Algorithm      = @cpd_als;
options.Refinement     = @cpd_nls;
options.Complex        = true;

[Uhat, output] = cpd(T, 2, options, 'Display', 1);
```

The following output is printed:

```
Step 1: Compression is mlsvd_rsi (compression ratio 0.00145243 < 0.5)...
        relative error = 9.61532e-16.
Step 2: Initialization is cpd_rnd (requested by user)... relative error = 0.983011.
Step 3: Algorithm is cpd_als on the compressed tensor... iterations = 10,
        relative error = 2.30489e-05.
Step 4: Refinement is cpd_nls on the uncompressed tensor:
| Step 4a: Detecting structure... detected Hankel structure.
| Step 4b: Converted tensor to efficient representation.
| Step 4c: cpd_nls on efficient representation... iterations = 1,
           relative error = 6.47107e-10.
| Step 4d: cpd_nls on the original full tensor to improve accuracy...
           iterations = 1, relative error = 2.66907e-16.
```

The structure detection is shown in step 4 of the printed output. Although a full tensor is given, the algorithm automatically detected the Hankel structure (step 4a) and converted the tensor to the efficient representation

(step 4b). The refinement is then executed on this efficient representation (step 4c). This can be done more efficiently than when using the original full tensor. Finally, some additional refinement steps are applied to the original tensor to improve the accuracy (step 4d). These substeps are performed in step 3 if no compression is performed. Note that, for illustration purpose, the `cpd_als` and `cpd_minf` algorithms have been used instead of the default `cpd_nls` algorithm, as the latter converges too fast in this case. Due to this fast convergence, the structure detection is then no longer needed.

In the previous example, the tensor `T` is a full tensor. The next two examples show the behavior for efficient representations of structured tensors `T`. First, a rank-2 tensor $\mathcal{T}$ is created by Hankelizing a sampled cosine function. To obtain the efficient representation, the `hankelize` function with the `full` option set to `false` is used. (`getstructure(T)` can be used to verify that the efficient Hankel representation is used.) This efficient representation for the tensor $\mathcal{T}$ is used by `cpd`:

```
T = hankelize(cos(2*pi*linspace(0,1,51)), 'order', 3, 'full', false);
options = struct;
options.Algorithm = @cpd_minf;
options.Complex = true;
[Uhat, output] = cpd(T, 2, options, 'Display', 1);
```

```
Step 1: Compression is mlsvd_rsi (compression ratio 0.00145243 < 0.5):
| Step 1a: Structured tensor expanded to full tensor (prod(getsize(T)) <= 1e+06).
| Step 1b: mlsvd_rsi on full tensor... relative error = 1.06122e-15.
Step 2: Initialization is cpd_gevd (sum(R <= size(T)) >= 2)...
        relative error = 3.47269e-16.
Step 3: Algorithm is cpd_minf on the compressed tensor... iterations = 1,
        relative error = 3.47269e-16.
Step 4: Refinement skipped (relative error = 9.96998e-16 <= 1e-08)
```

From the printed output, we see that the number of elements in the full tensor is smaller than the `ExpandLimit = 1e6`. Therefore, the representation is expanded to a full tensor such that the `mlsvd_rsi` algorithm can be applied for compression. In case no compression is performed, the expansion is performed in step 2. Next, an initial guess is computed using the compressed tensor in step 2. This initial guess is then improved using `cpd_minf` in step 3. Finally, in step 4, a refinement step is performed on the original efficient representation instead of on the compressed tensor. However, in this example, the relative error is already smaller than what can be obtained using efficient representations. In case a high accuracy is required, please check the note at the end of this chapter. Note that the relative error in step 4 is higher than in step 3. This is due to the fact that the error is computed on the compressed tensor in step 3, while it is computed on the uncompressed tensor in step 4.

In the second example for structured tensors, we use `cpd_rnd` for the initialization instead of `cpd_gevd`. A different situation occurs: the relative error in the beginning of the refinement step is not smaller than `1e-08` and the refinement step is still performed on the efficient representation. As the tensor has Hankel structure by construction, no automatic detection is needed.

```
T = hankelize(cos(2*pi*linspace(0,1,51)), 'order', 3, 'full', false);
options = struct;
options.Algorithm = @cpd_minf;
options.Initialization = @cpd_rnd;
options.Complex = true;
[Uhat, output] = cpd(T, 2, options, 'Display', 1);
```

```
Step 1: Compression is mlsvd_rsi (compression ratio 0.00145243 < 0.5):
| Step 1a: Structured tensor expanded to full tensor (prod(getsize(T)) <= 1e+06).
| Step 1b: mlsvd_rsi on full tensor... relative error = 5.62399e-16.
Step 2: Initialization is cpd_rnd (requested by user)... relative error = 0.999409.
Step 3: Algorithm is cpd_minf on the compressed tensor... iterations = 73,
        relative error = 5.1671e-07.
Step 4: Refinement is cpd_minf on the uncompressed tensor:
| Step 4a: Detection of structure skipped (hankel structure already known).
| Step 4b: cpd_minf... iterations = 7, relative error = 1.69504e-08.
```

3.2.4 Computing the error

The residual between a tensor $\mathcal{T}$ and its CPD approximation defined by factor matrices $\hat{\mathbf{U}}^{(n)}$ can be computed with

```
U = cpd_rnd([10 11 12], 5);
T = cpdgen(U);
Uhat = cpd(noisy(T, 60), 5);

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); % or alternatively by
relerr = frobcpdres(T, Uhat)/frob(T);
```

The latter method also handles efficiently represented tensors.

One can also consider the relative error between the factor matrices $\mathbf{U}^{(n)}$ that generated $\mathcal{T}$ and their approximations $\hat{\mathbf{U}}^{(n)}$ computed by `cpd`. Due to the permutation and scaling indeterminacies of the CPD, the columns of $\hat{\mathbf{U}}^{(n)}$ need to be permuted and scaled to match the columns of $\mathbf{U}^{(n)}$ as well as possible. The function `cpderr` takes care of these indeterminacies and 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 $\mathbf{U}^{(n)}$ and $\hat{\mathbf{U}}^{(n)}$. This method is also applicable when $\hat{\mathbf{U}}^{(n)}$ is an under- or overfactoring of the solution, meaning $\hat{\mathbf{U}}^{(n)}$ comprises fewer or more rank-one terms than the tensor's rank, respectively. In the underfactoring case, $\hat{\mathbf{U}}^{(n)}$ is padded with zero-columns, and in the overfactoring case only the columns in $\hat{\mathbf{U}}^{(n)}$ that best match those in $\mathbf{U}^{(n)}$ are kept. The command also returns the permutation matrix, the N scaling matrices and the permuted and scaled factor matrices $\hat{\mathbf{U}}^{(n)}_{\mathsf{ps}}$:

```
[relerr,P,D,Uhatps] = cpderr(U,Uhat);
```

See the `help` information for more details.

3.3 Choosing the number of rank-one terms R

Determining the rank R of a tensor $\mathcal{T}$ is often a difficult problem. In many cases domain knowledge or trial-and-error are used to find the correct rank. A useful lower bound can be found using the multilinear singular values. A tool which can sometimes help choosing R is the `rankest` method. Running `rankest(T)` on a dense, sparse,

incomplete or structured tensor $\mathcal{T}$ 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 [16]:

```
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 Fig. 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 `MaxRelErr` option. The lower bound is based on the truncation error of the tensor's multilinear singular values [17]. 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 `MinRelErr` option. It is recommended to set `MaxRelErr` higher than the expected noise level and to set `MinRelErr` at a value near the noise level and smaller than `MaxRelErr`. 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.

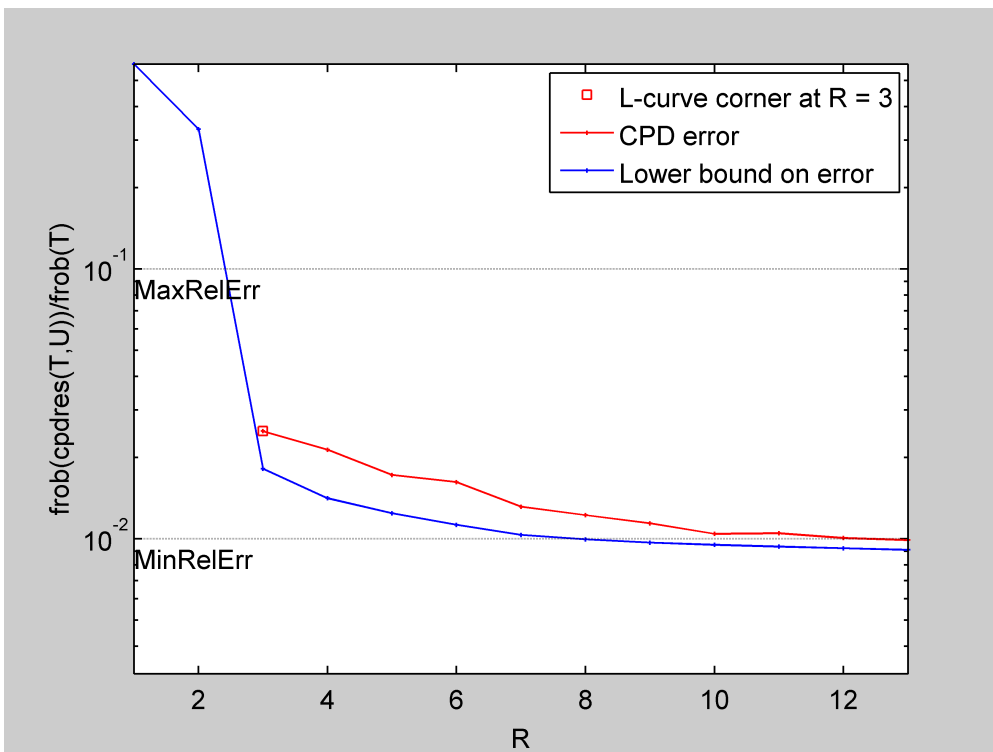

Fig. 3.2: The `rankest` tool for choosing the number of rank-one terms R in a CPD.

3.4 Overview of algorithms

The `cpd` method uses a variety of algorithms to compute the decomposition using a multi-step initialization sequence. These algorithms can be called separately as well. Below, an overview of the available algorithms is

given, as well as an overview of other convenient algorithms.

The following algorithms can be used to compute the canonical polyadic decomposition of dense, sparse, incomplete or efficiently represented tensors.

- `cpd(T, R)` computes the CPD of a tensor `T` using a multistep approach.
- `cpd_nls(T, U)` computes the CPD of a tensor `T` using an nonlinear least squares type algorithm with initial guess `U`.
- `cpd_minf(T, U)` computes the CPD of a tensor `T` using nonlinear unconstrained optimization with initial guess `U`.
- `cpd_als(T, U)` computes the CPD of a tensor `T` using alternating least squares with initial guess `U`.
- `cpd_rbs(T, U)` computes the CPD of a tensor `T` using randomized block sampling with initial guess `U` and is used for large-scale tensor decompositions [18].
- `cpd_gevd(T, R)` computes the CPD using the generalized eigenvalue decomposition [14]. `cpd_gevd` only accepts full and sparse tensors. For structured tensors `cpd_gevd(ful(T), R)` can be used. `cpd_gevd` often provides a good initialization for optimization-based algorithms.
- `cpd3_sd(T, U)` computes the CPD of a third-order, dense tensor `T` using simultaneous diagonalization with initial guess `U` [19].
- `cpd3_sgsd(T, U)` computes the CPD of a third-order, dense tensor `T` using simultaneous generalized Schur decomposition with initial guess `U` [20].

The following line and plane search methods can be used in combination with the optimization routines `cpd_nls`, `cpd_minf` and `cpd_als` (see `help cpd_xxx` for more information):

- `cpd_els` uses exact line search.
- `cpd_aels` uses approximated exact line search.
- `cpd_lsb` uses Bro's line search method [16].
- `cpd_eps` uses plane search.

The following routines can be used for problem generation, error analysis and rank estimation.

- `cpd_rnd(size_tens, R)` generates pseudorandom factors for a CPD.
- `cpdgen(U)` generates the full tensor from a given CPD `U`.
- `cpderr(U0, Uest)` computes the relative error for each factor matrix in `U0` and `Uest` after resolving the scaling and permutation indeterminacies.
- `cpdres(T, U)` computes the residual `cpdgen(U) - T`.
- `frobcpdres(T, U)` computes the Frobenius norm of the residual `cpdgen(U) - T`.
- `crb_cpd(U, variance)` computes the Cramér-Rao bound for a CPD `U` and a given noise variance [21, 22, 18].
- `rankest(T)` tries to estimate the rank of a tensor `T`.

Note: When using efficient representations of structured tensors, the expected numerical precision is only half the machine precision. This is due to the mathematical formulation implemented to exploit the structure. In most cases, this is not a problem as the maximal attainable precision is already limited due to noise and model errors. The effect of this reduced accuracy is only visible in the very specific case when a solution accurate up to machine precision can be found and is required. (This requires `TolFun` and `TolX` to be `eps^2` and `eps`, respectively.) An automatic solution has not been implemented yet. For now, a solution is to use the structured tensor `T` to compute a solution `Ures` accurate up to half the machine precision, and then refine this solution using `cpd_nls(ful(T), Ures, 'TolFun', eps^2,'TolX', eps)` for a few iterations. (It can be useful to set `MaxIter` to `2` or `3`.) This technique requires only very few expensive iterations involving the full tensor.

3.5 Further reading

The algorithms outlined in this chapter can be used to compute an unconstrained CPD. In the structured data fusion (SDF) framework constraints can be imposed and coupled decompositions can be defined. See Chapter 8 for more details.

A number of demos illustrating the CPD are available. In the demo "Basic Use of Tensorlab"[1] a beginner's introduction to Tensorlab and CPD is given. The demos "Multidimensional Harmonic Retrieval"[2], "Using Basic Tensorlab Features for ICA"[3] and "Using Advanced Tensorlab Features for ICA"[4] show more practical examples of the use of a CPD in a signal processing context.

[1] http://www.tensorlab.net/demos/basic.html
[2] http://www.tensorlab.net/demos/harmonic-retrieval.html
[3] http://www.tensorlab.net/demos/ica.html
[4] http://www.tensorlab.net/demos/sobi.html

CHAPTER

FOUR

DECOMPOSITION IN MULTILINEAR RANK-$(L_R, L_R, 1)$ TERMS

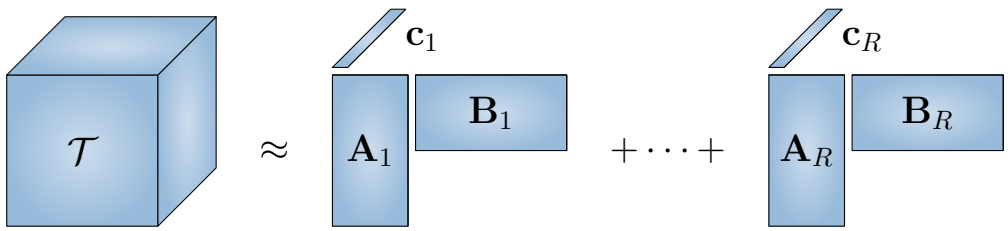

Fig. 4.1: A decomposition in multilinear rank-$(L_r, L_r, 1)$ terms of a third-order tensor.

The decomposition in multilinear rank-$(L_r, L_r, 1)$ terms writes a third-order tensor as a sum of R low multilinear rank terms [25, 26, 27, 28, 6]. Each of these low multilinear rank terms can be written as the outer product, denoted by $\otimes$, of a rank-L_r matrix and a vector. For a third-order tensor $\mathcal{T}$, the decomposition is given by

$$\mathcal{T} = \sum_{r=1}^{R} (\mathbf{A}_r \mathbf{B}_r^{\mathsf{T}}) \otimes \mathbf{c}_r. \tag{4.1}$$

The matrices $\mathbf{A}_r$ and $\mathbf{B}_r$ have L_r columns and have full rank. A visual representation of this decomposition is shown in Fig. 4.1. The decomposition in (4.1) is a special case of the polyadic decomposition (PD, see Chapter 3) and the block term decomposition (BTD, see Chapter 6). Indeed, (4.1) can be seen as a CPD by writing it as a sum of $\sum_r L_r$ rank-1 terms

$$\mathcal{T} = \sum_{r=1}^{R} \sum_{l=1}^{L_r} \mathbf{A}_r(:, l) \otimes \mathbf{B}_r(:, l) \otimes \mathbf{c}_r.$$

Alternatively, (4.1) can be formulated as a BTD by writing it as the sum of R block terms. The rth term is formed by the tensor-matrix products of a core tensor $\mathcal{S}_r$ of size $L_r \times L_r \times 1$ and matrices $\mathbf{A}_r$, $\mathbf{B}_r$ and vector $\mathbf{c}_r$:

$$\mathcal{T} = \sum_{r=1}^{R} \mathcal{S}_r \cdot_1 \mathbf{A}_r \cdot_2 \mathbf{B}_r \cdot_3 \mathbf{c}_r,$$

in which $\cdot_n$ denotes the tensor-matrix product in mode n.

Both formats are supported in Tensorlab, and all methods starting with `ll1` allow both the CPD format and the BTD format.

4.1 Problem and tensor generation

We first show how pseudorandom factors for the decomposition in rank-$(L_r, L_r, 1)$ terms can be generated in the two formats using `ll1_rnd` and conclude with the generation of the full tensors from both formats using `ll1gen` and `ful`.

Two different formats for the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms can be used: the CPD and the BTD formulation. The BTD formulation is the default format for most algorithms, and is represented by a cell with R entries, each of which contains a cell with exactly four entries: the matrices $\mathbf{A}_r$ and $\mathbf{B}_r$, the vector $\mathbf{c}_r$ and a core tensor $\mathcal{S}_r$. All algorithms starting with `ll1` normalize this core tensor to the identity matrix. The command `ll1_rnd` generates pseudorandom factors for a decomposition in multilinear rank-$(L_r, L_r, 1)$. For example, consider a tensor of size $10 \times 11 \times 12$ that can be written as a sum of three terms with multilinear ranks $(2, 2, 1)$, $(3, 3, 1)$ and $(4, 4, 1)$, hence `L = [2 3 4]`:

```
size_tens = [10 11 12];
L = [2 3 4];
U = ll1_rnd(size_tens, L);
U
U{1}
U{1}{4}
```

```
U =
  {1x4 cell}    {1x4 cell}    {1x4 cell}

ans =
  [10x2 double]    [11x2 double]    [12x1 double]    [2x2 double]

ans =
   1     0
   0     1
```

The CPD format can be requested using the `'OutputFormat'` option:

```
options = struct;
options.OutputFormat = 'CPD';
U = ll1_rnd(size_tens, L, options);

% alternatively, using the new options syntax
U = ll1_rnd(size_tens, L, 'OutputFormat', 'CPD');

U
```

```
U =
 [10x9 double]    [11x9 double]    [12x3 double]
```

The result `U` is a cell of length $R = 3$ with $\sum_r L_r$ columns in the first two factor matrices and R columns in the third factor matrix. When using the CPD format, the parameter `L` is required for all algorithms, as this parameter provides the necessary information on how the columns are grouped.

As for the corresponding CPD, BTD and LMLRA methods, additional options can be provided using the old syntax with an `options` structure, or as key-value pairs:

```
size_tens = [10 11 12];
L = [2 3 4];

% Use rand to generate factors
U = ll1_rnd(size_tens, L, 'Real', @rand);
% Create complex factors
U = ll1_rnd(size_tens, L, 'Imag', @randn);
```

Instead of `size_tens`, a full, sparse, incomplete or efficient representation of a structured tensor `T` can be given as input, i.e., `ll1_rnd(T,L)`. `size_tens` is then determined using `getsize(T)`. If the tensor `T` (or `ful(T)`) contains complex entries, complex factors are generated for the factors `U`, otherwise real factors are generated.

It is possible to convert a decomposition in multilinear rank-$(L_r, L_r, 1)$ terms from the BTD format to the CPD format and vice versa using the `ll1convert` command:

```
size_tens = [10 11 12];
L = [2 3 4];
Ubtd  = ll1_rnd(size_tens, L, 'OutputFormat', 'btd');
Ucpd  = ll1convert(Ubtd); % result in CPD format
Ubtd2 = ll1convert(Ucpd, L); % result again in BTD format (do not forget L)
Ucpd2 = ll1convert(Ucpd, L, 'OutputFormat', 'cpd') % explicitly request CPD format
```

To expand the factorized representation `U` of the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms to a full tensor, `ll1gen` can be used:

```
size_tens = [10 11 12];
L = [2 3 4];
Ubtd = ll1_rnd(size_tens, L, 'OutputFormat', 'btd');
T    = ll1gen(Ubtd);
T    = ll1gen(Ubtd, L); % L is optional for BTD format

Ucpd = ll1_rnd(size_tens, L, 'OutputFormat', 'cpd');
T    = ll1gen(U,L); % L is required for CPD format
```

The `ful` command can also be used, but it only accepts the BTD format. For example, to generate a vector from the resulting tensor, the following command can be used:

```
size_tens = [10 11 12];
L = [2 3 4];
Ubtd = ll1_rnd(size_tens, L);

T = ful(Ubtd); % generate full tensor
t = ful(Ubtd, 1, ':', 1); % generate first row vector

norm(t - T(1,:,1)); % is zero up to machine precision
```

If the factors `U` are in the CPD format, the factors have to be converted to the BTD format first:

```
Ucpd = ll1_rnd(size_tens, L, 'OutputFormat', 'cpd');
T = ful(ll1convert(Ucpd, L));
```

4.2 Computing the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms

To compute a decomposition in multilinear rank-$(L_r, L_r, 1)$ terms a number of algorithms are available: a deterministic algorithm based on the generalized eigenvalue decomposition `ll1_gevd`, optimization-based algorithms `ll1_minf` and `ll1_nls`, and the `ll1` method. This last method is the principal method which automatically handles the compression, expansion, initialization, optimization and refinement, in a similar way as the `cpd` method. The `ll1` method is a good algorithm choice for many problems. The optimization-based algorithms

`ll1_minf`, `ll1_nls` and `ll1` accept dense, sparse, incomplete and efficient representations of structured tensors. If efficient representations are used, and a solution accurate up to machine precision is needed, make sure to read the note at the end of this chapter.

4.2.1 The principal method

The easiest way to compute a decomposition in multilinear rank-$(L_r, L_r, 1)$ terms of a dense, sparse, incomplete or structured tensor $\mathcal{T}$ with $L = [L_1, L_2, \ldots, L_R]$ is to use the `ll1` method:

```
size_tens = [10 11 12];
L  = [2 3 4];
U = ll1_rnd(size_tens, L);
T = ll1gen(U);

Uhat = ll1(T, L);
```

Internally, `ll1` performs a number of steps to provide a good initialization to reduce the computational cost of the decomposition. First, the tensor is *compressed* using a low multilinear rank approximation (see Chapter 5) if it is worthwhile. Then an *initialization* `U0` is generated, e.g., using `ll1_gevd`. Next `ll1` *executes an algorithm* to compute the CPD given the initialization, e.g., `ll1_nls`. Finally the tensor is uncompressed and the solution is *refined* (if compression was applied). Depending on the kind of input tensor `T`, additional substeps are performed. If `T` is a dense tensor, the command `detectstructure` is used to detect if there is any structure to be exploited during the decomposition (see `ExploitStructure` option). If structure is detected, the dense tensor is converted to its efficient representation. If `T` is the efficient representation of a sufficiently small structured tensor, `T` is expanded to allow the use of better compression and initialization algorithms (see `ExpandLimit` option).

Instead of letting `ll1(T,L)` compute an initialization, the initialization can also be provided:

```
L = [2 3];
Ucpd = ll1_rnd([10 11 12], L, 'OutputFormat', 'cpd');
Ubtd = ll1_rnd([10 11 12], L, 'OutputFormat', 'btd');

Uhat = ll1(T, Ucpd, L);
Uhat = ll1(T, Ubtd, L);
Uhat = ll1(T, Ubtd);    % hence L is optional for the BTD format
```

Note that `ll1_gevd` is a deterministic algorithm, unless slices are mixed randomly (see `help ll1_gevd`). Hence, if this algorithm is used for initialization in `ll1`, the result is always the same for multiple runs for the same $\mathcal{T}$ and L, and for the same options. Changing the initialization method to, for instance, `ll1_rnd`, is useful when different initializations are needed. The following example uses the `ll1_gevd` algorithm as initialization method for the first run, and uses a random initialization (`ll1_rnd`) for the other runs:

```
init = @ll1_gevd;
results = cell(1, 5);
for n = 1:5
    results{n} = ll1(T, L, 'Initialization', init);
    init = @ll1_rnd;
end
```

The next subsection explains different options of the `ll1` routine.

4.2.2 Setting the options

The different steps in `ll1` are customizable by supplying the method with an options structure (see `help ll1` for more information). Options can be provided using an `options` structure, or using key-value pairs, e.g.,

```
% Using options struct
options = struct;
options.Display = true;            % Show progress on the command line
options.Initialization = @ll1_rnd; % Take ll1_rnd as initialization method
options.Complex = true;            % Use complex initialization even for real tensor
ll1(T, L, options);

% Using key-value pairs
ll1(T, L, 'Display', 10, 'Initialization', @ll1_rnd, 'complex', true);
```

When key-value pairs are used, the options are case insensitive. The options `InitializationOptions`, `AlgorithmOptions` and `RefinementOptions` can be used to pass options (as a structure) to the different algorithms. To set the maximum number of iterations for the refinement step, for example, we can use

```
refinementOptions = struct;
refinementOptions.MaxIter = 100;
Uhat = ll1(T, L, 'Refinement', @ll1_minf, 'RefinementOptions', refinementOptions);
```

These structures are passed to the algorithms corresponding to the initialization, algorithm or refinement step. For example, `ll1` calls `ll1_minf` in the example above as `ll1_minf(T, options.Initialization(T,L), L,options.RefinementOptions)` in which `options.Initialization` is the default initialization method.

Since Tensorlab 3.0 some frequently used options can be passed automatically. For example `ll1(T, L, 'Display', 10)` automatically passes the `Display = 10` option to `AlgorithmOptions` and `RefinementOptions`, unless these option structures define the `Display` option. Similarly, the `MaxIter`, `TolX`, `TolFun` and `CGMaxIter` options are passed automatically.

4.2.3 Viewing the algorithm output

Each step may also output additional information specific to that step. For instance, most algorithms to decompose a tensor into multilinear rank-$(L_r, L_r, 1)$ terms such as `ll1_nls` keep track of the number of iterations and objective function value. To obtain this information, capture the second output argument of `ll1`:

```
L = [2 3];
U = ll1_rnd([10 11 12], L);
T = ll1gen(U);
[Uhat,output] = ll1(T,L);
```

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(ll1res(T,U))');
grid on;
```

The `output` structure has at least four fields: `Compression`, `Initialization`, `Algorithm` and `Refinement`. Each field can be inspected to see which algorithm is called, e.g., in the example above,

`output.Compression.Name` is `mlsvd_rsi`, or what the result is, e.g., in the example above the compression ratio is `output.Compression.ratio = 0.0379`.

If the `Display` option is set to `true` or to a number greater than zero, a number of messages are printed, depending on the steps taken to provide the result. As an example, consider the following code:

```
U = ll1_rnd([10 11 12], [3 2]);
T = noisy(ll1gen(U), 40);
Uhat = ll1(T, [3 2], 'Display', 1);
```

prints the following messages

```
Step 1: Compression is mlsvd_rsi (compression ratio 0.0378788 < 0.5)...
        relative error = 0.00955528.
Step 2: Initialization is ll1_gevd (sum(sum(L) <= size(T)) >= 2)...
        relative error = 0.00160962.
Step 3: Algorithm is ll1_nls on the compressed tensor... iterations = 3,
        relative error = 0.000923906.
Step 4: Refinement is ll1_nls on the uncompressed tensor:
| Step 4a: Detecting structure... no structure detected.
| Step 4b: ll1_nls... iterations = 2, relative error = 0.00959971.
```

The output shows that the tensor is first compressed using `mlsvd_rsi`. The compression causes an approximation error. The `ll1_gevd` then computes a first factorization of the compressed tensor using the GEVD method. This result is used as initial solution for the `ll1_nls` optimization routine. As can be seen, the relative error is decreased in three iterations. In the refinement step, the original, uncompressed tensor `T` is decomposed, starting from the result of step 3. The refinement step consists of several substeps. In step 4a, the algorithm tries to detect if structure is present in the tensor. If structure is detected, this is exploited in the refinement algorithm. In this case, no structure is detected and the original tensor is used. Refinement is only performed if compression is used in step 1. If no compression is performed, the structure detection takes place in step 3.

If the efficient representation of a structured tensor is used, additional substeps can be performed. For example:

```
U = ll1_rnd([10 11 12], [3 2]);
Uhat = ll1(U, [3 2], 'Display', 1); % use U as structured tensor
```

```
Step 1: Compression is mlsvd_rsi (compression ratio 0.0378788 < 0.5):
| Step 1a: Structured tensor expanded to full tensor
           (prod(getsize(T)) <= 1e+06).
| Step 1b: mlsvd_rsi on full tensor... relative error = 8.65928e-16.
Step 2: Initialization is ll1_gevd (sum(sum(L) <= size(T)) >= 2)...
        relative error = 1.96691e-15.
Step 3: Algorithm is ll1_nls on the compressed tensor... iterations = 1,
        relative error = 1.73667e-16.
Step 4: Refinement skipped. (relative error <= 1e-08)
```

As can be seen from the output, an additional preprocessing step (step 1a) is introduced. If a structured tensor is small enough, i.e., if its number of elements is smaller than `ExpandLimit` (default: 10^6), the efficient representation is expanded to a full tensor for the compression and/or initialization steps as this allows the use of better algorithms. If the number of entries in the tensor is larger than `ExpandLimit`, expanding the tensor is too expensive, and other algorithms are selected.

4.2.4 Computing the error

The residual between a tensor `T` and its decomposition in multilinear rank-$(L_r, L_r, 1)$ terms given by `Uhat` can be computed with

```
res = ll1res(T, Uhat);     % Uhat in BTD format
res = ll1res(T, Uhat, L); % Uhat in CPD or BTD format
```

If the tensor is dense, then the result is equivalent to `ll1gen(Uhat)-T` or `ll1gen(Uhat,L)-T`. The relative error between the tensor and its CPD approximation can then be computed as

```
relerr = frob(ll1res(T,Uhat))/frob(T);    % Uhat in BTD format
relerr = frob(ll1res(T,Uhat,L))/frob(T); % Uhat in CPD or BTD format
```

In this approach, efficient representations of structured tensors are expanded to full tensors using `ful(T)`. The methods `frobll1res(T,Ubtd)` and `frobll1res(T,Ucpd,L)` compute the Frobenius norm of the residual without expanding representations of large structured tensors (if `prod(getsize(T)) > ExpandLimit`, see `help frobll1res` for more information). Note that a loss of accuracy can occur if the expanded tensor `That = ll1gen(Uhat,L)` is almost equal to `T` in relative sense (see note at the end of this chapter).

4.3 Overview of algorithms

The `ll1` method uses a variety of algorithms to compute the decomposition in a multi-step initialization sequence. These algorithms can be called separately as well. Below an overview of the different available algorithms is given, as well as an overview of other convenient algorithms.

The following algorithms can be used to compute the decomposition in rank-$(L_r, L_r, 1)$ terms of a full, sparse, incomplete or structured tensor.

- `ll1(T, L)` computes the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms of a tensor `T` using a multistep approach.
- `ll1_nls(T, Ubtd)` and `ll1_nls(T, Ucpd, L)` compute the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms of a tensor `T` using an NLS type algorithm with an initial guess in the BTD or CPD format.
- `ll1_minf(T, Ubtd)` and `ll1_minf(T, Ucpd, L)` compute the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms of a tensor `T` using a nonlinear unconstrained optimization type algorithm with an initial guess in the BTD or CPD format.
- `ll1_gevd(T, L)` computes the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms using the generalized eigenvalue decomposition [27]. `ll1_gevd` only accepts full and sparse tensors. For efficient representations of structured tensors `ll1_gevd(ful(T),L)` can be used. `ll1_gevd` often provides a good initialization for the optimization-based algorithms.

The following routines can be used to generate random tensors with structure, to generate full tensors and to compute (Frobenius norms of) residuals.

- `ll1_rnd(size_tens, L)` generates random factors `Ubtd` or `Ucpd` for a decomposition in multilinear rank-$(L_r, L_r, 1)$ terms.
- `ll1convert(U)` and `ll1convert(U, L)` convert BTD format to CPD format and vice versa.

- `ll1gen(U)` and `ll1gen(U,L)` generate the full tensor from the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms given in the BTD or CPD format.
- `ll1res(T, U)` and `ll1res(T, U, L)` compute the residual `ll1gen(U) - T`.
- `frobll1res(T, U)` and `frobll1res(T, U, L)` compute the Frobenius norm of the residual `ll1gen(U) - T`.

All algorithms that have factors `U` as an output argument, have an option `OutputFormat` which can be set to `btd` or `cpd` to get the desired output format.

Note: When using structured tensors, the expected numerical precision is only half the machine precision. This is due to the mathematical formulation implemented to exploit the structure. In most cases, this is not a problem as the maximal attainable precision is already limited due to noise and model errors. Only in the very specific case when a solution accurate up to machine precision can be found and is required, this can be an issue. (This requires `TolFun` and `TolX` to be `eps^2` and `eps`, respectively.) A solution is to use the structured tensor `T` to compute a solution `Ures` accurate up to half the machine precision, and then refine this solution using `ll1_nls(ful(T), Ures, 'TolFun', eps^2, 'TolX', eps)` for a few iterations. (It can be useful to set `MaxIter` to `2` or `3`.) This technique requires only very few expensive iterations involving the full tensor.

4.4 Further reading

The algorithms outlined in this chapter can be used to compute unconstrained decompositions in multilinear rank-$(L_r, L_r, 1)$ terms. In the structured data fusion (SDF) framework, constraints can be imposed as well, and coupled decompositions can be defined. See Chapter 8 for more details.

CHAPTER

FIVE

MULTILINEAR SINGULAR VALUE DECOMPOSITION AND LOW MULTILINEAR RANK APPROXIMATION

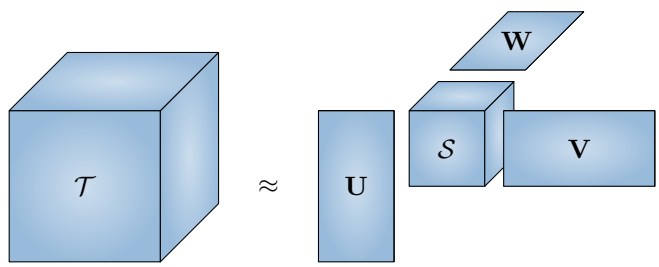

Fig. 5.1: A (truncated) multilinear singular value decomposition or a low multilinear rank approximation of a third-order tensor.

In both the multilinear singular value decomposition (MLSVD) [17] and the low multilinear rank approximation (LMLRA) [4, 5, 6] a tensor $\mathcal{T}$ of size $I_1 \times I_2 \times \cdots \times I_N$ is written as the multilinear tensor-matrix product of a core tensor $\mathcal{S}$ of size $R_1 \times R_2 \times \cdots \times R_N$ with N factors $\mathbf{U}^{(n)}$ of size $I_n \times R_n$:

$$\mathcal{T} \approx \mathcal{S} \cdot_1 \mathbf{U}^{(1)} \cdot_2 \mathbf{U}^{(2)} \cdot_3 \cdots \cdot_N \mathbf{U}^{(N)},$$

in which $\cdot_n$ is the tensor-matrix product in mode n (see Section 2.2). For all n, $R_n \leq I_n$. A visual representation of these decompositions in the third-order case is shown in Fig. 5.1. The relations between the two decompositions are explained in the demo on "Multilinear Singular Value Decomposition and Low Multilinear Rank Approximation"[1]. The differences between the MLSVD and the LMLRA lie in the computation and the optimality of the decomposition. As the decompositions are closely related and have the same format, both are discussed together here.

5.1 Problem and tensor generation

5.1.1 Generating pseudorandom factor matrices and core tensor

Tensorlab stores the core tensor $\mathcal{S} \in \mathbb{C}^{R_1 \times R_2 \times \cdots \times R_N}$ and the factor matrices $\mathbf{U}^{(n)} \in \mathbb{C}^{I_n \times R_n}$, $n = 1, \ldots, N$ corresponding to an MLSVD or LMLRA of a tensor of dimensions $I_1 \times I_2 \times \cdots \times I_N$ as a Matlab array `S` and a cell array `U = {U{1}, U{2}, ..., U{N}}`. The `lmlra_rnd` method can be used to generate a pseudorandom core tensor $\mathcal{S}$ and pseudorandom factor matrices $\mathbf{U}^{(n)}$ corresponding to an MLSVD or LMLRA. For example, to construct a tensor of dimensions `size_tens = [I1, I2, ..., IN]` and with core size `size_core = [R1, R2, ..., RN]` we can use:

[1] http://www.tensorlab.net/demos/mlsvd.html

```
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 the result is normalized such that the factors $\mathbf{U}^{(n)}$ have orthonormal columns and such that $\mathcal{S}$ is all-orthogonal and has ordered multilinear singular values. Other generators can also be specified by providing option as a structure or as key-value pairs, e.g.:

```
% using a structure
options = struct;
options.Real = @rand;
options.Imag = @rand;
[U,S] = lmlra_rnd(size_tens,size_core,options);

% using key-value pairs
[U,S] = lmlra_rnd(size_tens, size_core, 'Real', @rand, 'Imag', @rand);
```

The core tensor `S` is generated as `rand(size_core) + rand(size_core)*1i`, and is then normalized. Before the normalization, the factor matrices `U{n}` are generated as

```
rand(size_tens(n),size_core(n)) + rand(size_tens(n),size_core(n))*1i{}`{}`.
```

The normalization of the factors matrices and the core tensor can be disabled by setting the `Unitary` and `Normalize` options to `false`.

5.1.2 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); % or
T = ful({U,S});
```

Note that additional braces `{}` are necessary for the `ful` command. The tensor `T = lmlragen(U,S)` can be computed using elementary tensor operations as

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

Both the `ful` and the `lmlragen` method also allow the generation of subtensors:

```
[U, S] = lmlra_rnd([10 11 12], [3 4 2]);
T = lmlragen(U,S);

t1 = lmlragen(U, S, [1 1 2 3 5 8]); % using linear indices
frob(t1 - T([1 1 2 3 5 8]))         % is zero up to machine precision

t2 = ful({U,S}, ':', ':', 9);       % using subscripts (':' is a string!)
frob(t2 - T(:,:,9))                 % is zero up to machine precision
```

5.2 Computing an MLSVD

To compute an MLSVD of a tensor $\mathcal{T}$, three algorithms are available: `mlsvd` can be used for dense tensors, `mlsvds` can be used for sparse tensors and `mlsvd_rsi` can be used for large tensors, which can be both dense and sparse. Currently, there exist no MLSVD algorithms for incomplete or efficiently represented tensors in Tensorlab. In the latter case `ful(T)` can be used, in the former case LMLRA algorithms can be used.

For a dense tensor $\mathcal{T}$, the full MLSVD and the multilinear singular values can be computed using

```
[U, S] = lmlra_rnd([10 11 12], [3 4 5]);
T = lmlragen(U,S);
Tn = noisy(T, 40); % add Gaussian noise with an SNR of 40dB
[U, S, sv] = mlsvd(T);
```

It is often useful to inspect the multilinear singular values `sv` as they provide useful information on the multilinear rank of the tensor. As an example, the singular values can be plotted using:

```
for n = 1:3
    subplot(1,3,n);
    semilogy(sv{n});
end
```

For a sparse tensor $\mathcal{T}$, the core size needs to be provided; hence, to compute the full MLSVD the following command can be used:

```
% Create sparse tensor with 4% non-zeros
T = randn(10, 11, 12);
T(randperm(numel(T), ceil(0.96*numel(T)))) = 0;
T = fmt(T);
% Compute mlsvd
[U, S, sv] = mlsvds(T, [10 11 12]);
```

Often, the MLSVD is truncated, i.e., the size of the core is smaller than the size of the tensor. An MLSVD can be truncated manually: for the tensor above, we can use

```
new_core = [5 4 7];
Ut = arrayfun(@(n) U(:,1:new_core(n)), U, new_core, 'UniformOutput', false);
St = S(1:new_core(1),1:new_core(2),1:new_core(3));
```

Alternatively, the core size can be passed to the `mlsvd` and `mlsvds` algorithms as an additional parameter. For example, the following code can be used to generate a multilinear rank-$(2, 3, 4)$ approximation:

```
Tfull = randn(10, 11, 12);
[Ufull, Sfull] = mlsvd(Tfull, [2 3 4]);

Tsparse = randn(10, 11, 12);
Tsparse(randperm(numel(Tsparse), ceil(0.96*numel(Tsparse)))) = 0;
Tsparse = fmt(Tsparse);
[Usparse, Ssparse] = mlsvds(Tsparse, [2 3 4]);
```

The sizes of the core tensor and the factor matrices are now

```
Ufull
size(Sfull)
```

```
Ufull =
    [10x2 double]    [11x3 double]    [12x4 double]
ans =
     2     3     4
```

```
T = randn(10, 11, 12);
[U, S] = mlsvd(T, [2 2 5], [], 'FullSVD', true); % perms = [] uses default
```

The MLSVD relies on singular value decompositions (SVD) of matrix representations of the given tensor. These SVDs can be expensive for larger tensors. The `mlsvd_rsi` routine uses randomized SVDs to improve the computation speed for dense and sparse tensors:

```
size_core = [5 6 7];
[U, S] = lmlra_rnd([30,40,50], size_core);
T = lmlragen(U,S);

tic, [U1, S1] = mlsvd(T, size_core); toc
tic, [U2, S2] = mlsvd_rsi(T, size_core); toc
```

```
Elapsed time is 0.073221 seconds.
Elapsed time is 0.027600 seconds.
```

Major improvements can be obtained for larger tensors. The accuracy of the results computed by `mlsvd_rsi` is most often similar to the accuracy of the results computed by `mlsvd` or `mlsvds`:

```
froblmlrares(T, U1, S1)/frob(T)
froblmlrares(T, U2, S2)/frob(T)
```

```
ans =
   2.4422e-15
ans =
   1.4491e-15
```

5.3 Computing an LMLRA

Several algorithms are available to compute the LMLRA of a dense, sparse, incomplete or structured tensor $\mathcal{T}$ with a (fixed) core size $R_1 \times R_2 \times \cdots \times R_N$. The commands `lmlra_nls` and `lmlra_minf` are optimization-based algorithms using nonlinear unconstrained optimization and nonlinear least squares, respectively. `lmlra_aca` uses adaptive cross approximation. This method often provides a good initialization. Finally, the principal method is `lmlra`, which automatically computes an initialization and refines this initial guess using one of the optimization-based methods. All optimization-based algorithms accept dense, sparse, incomplete and efficient representations of structured tensors (see Chapter 2). If efficient representations are used, and a solution accurate up to machine precision is needed, make sure to read the note at the end of this chapter.

5.3.1 The principal method

The `lmlra` method is the easiest way to compute an LMLRA of a tensor $\mathcal{T}$. 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 an 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., `mlsvd_rsi` or `lmlra_aca`. Next it *executes an algorithm* to compute the LMLRA given the initialization, e.g., `lmlra_nls`.

The command `lmlra(T, size_core)` computes an initial solution with the desired multilinear rank. Alternatively an initial solution can be provided using

```
[Uinit, Sinit] = lmlra_rnd([10 11 12], size_core);
[Uhat, Shat]   = lmlra(T, Uinit, Sinit);
```

The default initialization methods `mlsvd_rsi` or `lmlra_aca` return nearly the same initialization for every run. Therefore, if multiple runs are performed for a specific tensor $\mathcal{T}$ and a specific multilinear rank $(R_1, R_2, \ldots, R_N)$ (for the same options), the initialization method should be changed, e.g., to `lmlra_rnd`. The following example uses the `mlsvd_rsi` algorithm as initialization method for the first run, and uses a random initialization (`lmlra_rnd`) for the other runs:

```
init = @mlsvd_rsi;
resultU = cell(1, 5);
resultS = cell(1, 5);
for n = 1:5
    [resultU{n}, resultS{n}]= lmlra(T, size_core, 'Initialization', init);
    init = @lmlra_rnd;
end
```

The next subsection explains different options for the `lmlra` routine.

5.3.2 Setting the options

Different options can be used to customize the two steps in `lmlra` (see `help lmlra` for more information). Options can be provided using an `options` structure, or using key-value pairs, e.g.,

```
options.Display = true;                  % Show progress on the command line.
options.Initialization = @lmlra_rnd;     % Select pseudorandom initialization.
options.Algorithm = @lmlra_minf;         % 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 `InitializationOptions` and `AlgorithmOptions` are passed as options structures to the algorithms corresponding to the initialization and algorithm steps, respectively.

See `help lmlra` for a full overview of the possible options. The following is worthwhile to be highlighted:

- `ExpandLimit`: allows a structured tensor `T` to be expanded for the compression and/or initialization steps if `prod(getsize(T)) <= ExpandLimit`. Set `ExpandLimit = 1` to make sure tensors are never expanded. The default is 10^6.

5.3.3 Viewing the algorithm output

Each step may 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;
```

Setting the `Display` option to `true` or to a number greater than zero, prints additional information to the command window. For example:

```
size_core = [5 6 3];
[U,S] = lmlra_rnd([10 11 12], size_core);
T  = lmlragen(U,S);
Tn = noisy(T, 20);
[Uhat, Shat, output] = lmlra(Tn, size_core, 'Display', 1);
```

```
Step 1: Initialization is mlsvd_rsi...relative error = 0.0930664.
Step 2: Algorithm is lmlra_nls... iterations = 5,
        relative error = 0.0930472.
```

Two steps are performed by `lmlra`. In step 1, the `mlsvd_rsi` algorithm is used to compute an initial guess, which is refined in step 2 using the `lmlra_nls` algorithm.

In the case the tensor `T` is given as the efficient representation of a structured tensor, step 1 can also contain an expansion step:

```
% Create a tensor with CPD structure
T    = cpd_rnd([10 11 12], 5);
T{1} = bsxfun(@times, T{1}, [1000 100 10 1 0.01]);

options = struct;
options.Algorithm = @lmlra_minf;  % use minf algorithm

[Uhat, Shat, output] = lmlra(T, [4 4 4], 'Display', 1, options);
```

```
Step 1: Initialization is mlsvd_rsi:
| Step 1a: Structured tensor expanded to full tensor (prod(getsize(T) <= 1e+06)
| Step 1b: mlsvd_rsi on full tensor... relative error = 5.32044e-06.
Step 2: Algorithm is lmlra_minf... iterations = 1, relative error = 5.32044e-06.
```

In step 1a, the tensor is expanded such that the `mlsvd_rsi` algorithm can be used as initialization. Without expansion, `lmlra_aca` is the default initialization algorithm for structured tensors.

Note that structure is not yet detected automatically in the `lmlra` command.

5.3.4 Computing the error

The residual between a tensor $\mathcal{T}$ and its computed LMLRA defined by the core tensor $\hat{\mathcal{S}}$ and factor matrices $\hat{\mathbf{U}}^{(n)}$ 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 original factor matrices $\mathbf{U}^{(n)}$ that generated $\mathcal{T}$ are known, the subspace angle between each of them and their corresponding approximations $\hat{\mathbf{U}}^{(n)}$ from `lmlra` can also be used as 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})`.

5.4 Choosing the size of the core tensor

For dense tensors, one can 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 [17] on the relative error of an LMLRA of `T` for different core tensor sizes, and the LMLRA's compression ratio. The following example applies `mlrankest` on the amino acids dataset [16]:

```
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 Fig. 5.2. 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, an 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.

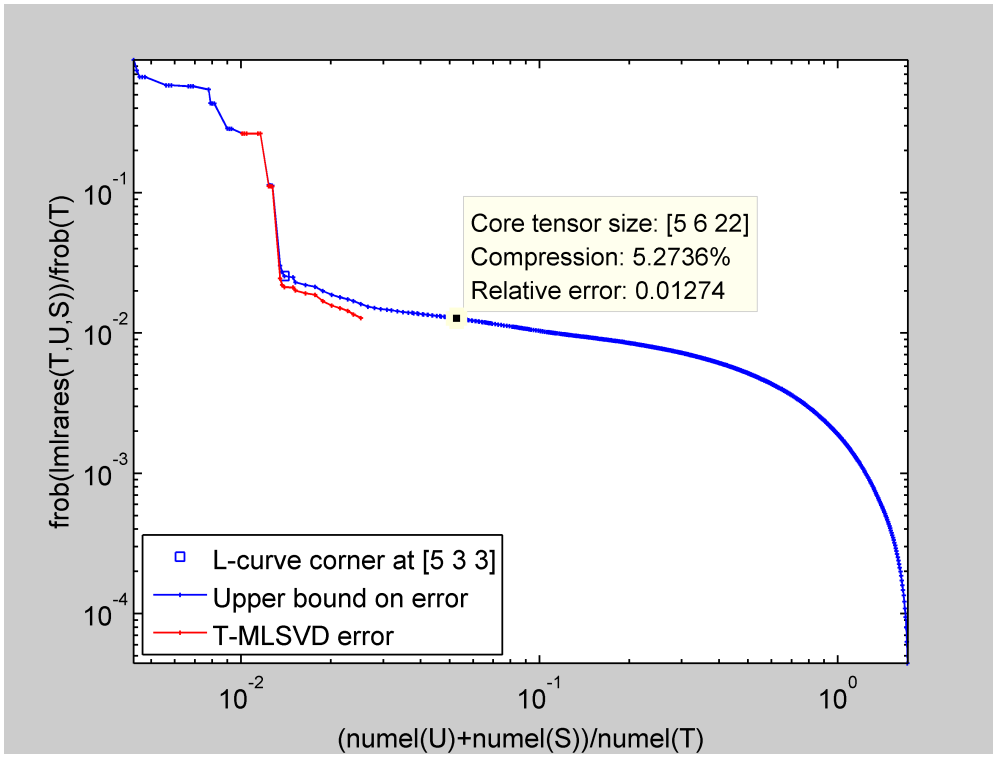

Fig. 5.2: The `mlrankest` tool for choosing the core tensor size `size_core` in an LMLRA.

5.5 Overview of algorithms

The `mlsvd` and `lmlra` methods are the principal methods for the MLSVD and the LMLRA. Below we give an overview of the available algorithms for the computation of the MLSVD and the LMLRA, as well as an overview of other convenient routines.

The following algorithms can be used to compute the low multilinear rank approximation or the multilinear singular value decomposition of a dense, sparse, incomplete or efficiently represented tensor, unless otherwise specified.

- `lmlra(T, size_core)` computes the LMLRA of a tensor `T` using a multistep approach [29].
- `lmlra_nls(T, U, S)` computes the LMRLA of a tensor `T` using an NLS type algorithm with as initial guess `U` for the factor matrices and `S` for the core tensor [29].
- `lmlra_minf(T, U, S)` computes the LMLRA of a tensor `T` using nonlinear unconstrained optimization with as initial guess `U` for the factor matrices and `S` for the core tensor [29].
- `lmlra_aca(T,size_core)` computes the LMLRA of a tensor `T` using adaptive cross approximation [30].
- `mlsvd(T)` computes the full MLSVD of a dense tensor `T`, including the multilinear singular values. `mlsvd(T, size_core)` truncates the MLSVD to the given core size and `mlsvd(T, tol)` truncates the MLSVD if the multilinear singular values are below a given tolerance `tol` [17, 31].
- `mlsvds(T,size_core)` computes the truncated MLSVD of a sparse tensor `T`. The same options as for `mlsvd` apply.
- `mlsvd_rsi(T, size_core)` computes the truncated MLSVD of a full or sparse tensor `T` using a ran-

domized SVD in combination with randomized subspace iteration. This algorithm is useful for larger tensors that have a low multilinear rank [32].

- `lmlra_hooi(T, U)` uses higher-order orthogonal iteration to compute the LMLRA of a dense tensor `T` [33].
- `lmlra3_dgn(T, U)` uses a differential-geometric Newton method to compute the LMLRA of a dense third-order tensor `T` [34].
- `lmlra3_rtr(T, U)` uses a Riemannian trust region method to compute the LMLRA of a dense third-order tensor `T` [35].

The following routines can be used for problem generation, error analysis and rank estimation.

- `lmlra_rnd(size_tens, size_core)` generates pseudorandom factors for an LMLRA.
- `lmlragen(U, S)` generates the full tensor from given factors `U` and core tensor `S`.
- `lmlrares(T, U, S)` computes the residual `lmlragen(U,S) - T`.
- `froblmlrares(T, U)` computes the Frobenius norm of the residual `lmlragen(U,S) - T`. This also works for structured tensors `T`.
- `lmlraerr(U, Uhat)` computes the angle between the subspaces of the original tensor and the approximation. The subspaces are given as the factor matrices `U` of the original tensor and the factor matrices `Uhat` of the approximation.
- `mlrank(T,tol)` computes the multilinear rank of a tensor `T` using a given tolerance `tol`.
- `mlrankest(T)` estimates the multilinear rank of a tensor `T` from an L-curve.

Note: When using structured tensors, the expected numerical precision is only half the machine precision. This is due to the mathematical formulation implemented to exploit the structure. In most cases, this is not a problem as the maximal attainable precision is already limited due to noise and model errors. Only in the very specific case when a solution accurate up to machine precision can be found and is required, this can be an issue. (This requires `TolFun` and `TolX` to be `eps^2` and `eps`, respectively.) A solution is to use the structured tensor `T` to compute a solution `Ures`, `Sres` accurate up to half the machine precision, and then refine this solution using `lmlra_nls(ful(T), Ures, Sres, 'TolFun', eps^2, 'TolX', eps)` for a few iterations. (It can be useful to set `MaxIter` to `2` or `3`.) This technique requires only very few expensive iterations involving the full tensor.

5.6 Further reading

The algorithms outlined in this chapter can be used to compute an unconstrained MLSVD or LMLRA. In the structured data fusion (SDF) framework constraints can be imposed as well, and coupled decompositions can be defined. See Chapter 8 for more details.

The demo on "Multilinear Singular Value Decomposition and Low Multilinear Rank Approximation"[2] gives some intuition behind the MLSVD and the LMLRA.

[2] http://www.tensorlab.net/demos/mlsvd.html

CHAPTER

SIX

BLOCK TERM DECOMPOSITION

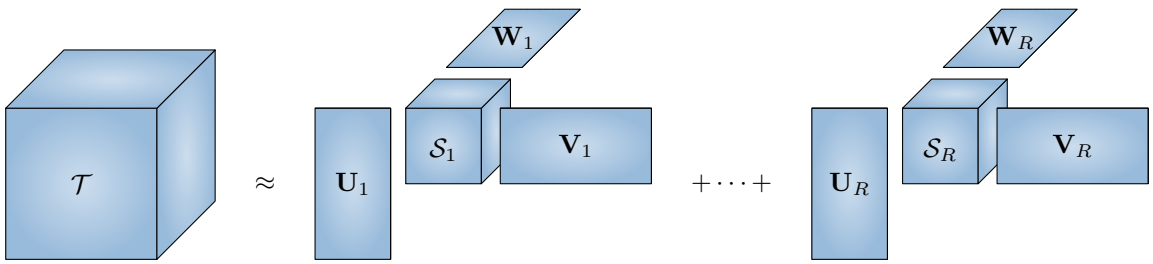

Fig. 6.1: A block term decomposition of a third-order tensor.

A block term decomposition (BTD) [4, 5, 6] approximates a tensor by a sum of low multilinear rank terms. Let $\mathcal{T}$ be a tensor of dimensions $I_1 \times I_2 \times \cdots \times I_N$. For $n = 1, \ldots, N$ and $r = 1, \ldots, R$, let $\mathbf{U}^{(r,n)}$ of size $I_n \times J_r^{(n)}$ represent the nth factor in the rth term in the BTD and $\mathcal{S}^{(r)}$ the core tensor in the rth term having $J_r^{(1)} \times J_r^{(1)} \times \cdots \times J_r^{(N)}$ as dimension. The BTD is then given as

$$\mathcal{T} \approx \sum_{r=1}^{R} \mathcal{S}^{(r)} \cdot_1 \mathbf{U}^{(r,1)} \cdot_2 \mathbf{U}^{(r,2)} \cdot_3 \cdots \cdot_N \mathbf{U}^{(r,N)}.$$

A visual representation of this decomposition for the third-order case is shown in Fig. 6.1.

6.1 Problem and tensor generation

6.1.1 Generating pseudorandom factor matrices and core tensor

In Tensorlab, a BTD with factor matrices $\mathbf{U}^{(r,n)}$ and core tensors $\mathcal{S}^{(r)}$ is stored as a cell of cells `U`. The rth entry `U{r}` contains the rth term in the BTD. Each term is a cell containing the N factor matrices and the core tensor for the rth term. This means `U{r}{n}` is the factor matrix $\mathbf{U}^{(r,n)}$ of size $I_n \times J_r^{(n)}$ for $n = 1, \ldots, N$, and `U{r}{N+1}` is the core tensor $\mathcal{S}^{(r)}$ of size $J_r^{(1)} \times J_r^{(2)} \times \cdots \times J_r^{(N)}$. Hence, `U = {U{1},U{2},...,U{R}}` is a cell with R cells, each having $N+1$ entries.

To generate pseudorandom factor matrices and core tensors corresponding to a BTD, `btd_rnd` can be used. The size of the tensor is given by `size_tens`, and the size of the R core tensors is given by the cell `size_core`, i.e., `size_core{r}` is the size of the rth core tensor. For example:

```
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 provided with options given as a structure or as key-value pairs, e.g.:

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

% using key-value pairs
[U,S] = btd_rnd(size_tens, size_core, 'Real', @rand, 'Imag', @rand);
```

Each term is normalized such that the factors $\mathbf{U}^{(r,n)}$ have orthonormal columns and such that $\mathcal{S}^{(r)}$ is all-orthogonal and has ordered multilinear singular values. The normalization can be disabled by setting the `Unitary` and `Normalize` options to `false`.

6.1.2 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);
```

Using basic tensor operations, the tensor `T` can also be computed as

```
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
```

The `ful` method can also be used to generate the full tensor. Both the `ful` and the `btdgen` method allow the generation of subtensors:

```
U = btd_rnd([10 11 12], {[3 4 2], [2,2,1]});
T = btdgen(U); % or, equivalently
T = ful(U);


t1 = btdgen(U, [1 1 3 2 1 5 8]); % using linear indices
frob(t1 - T([1 1 3 2 1 5 8]))    % is zero up to machine precision

t2 = ful(U, ':', 9, 3);          % using subscripts (':' is a string!)
frob(t2 - T(:,9,3))              % is zero up to machine precision
```

6.2 Computing a BTD

6.2.1 With a specific algorithm

In contrast to the CPD, the LMLRA and the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms, no specialized high-level algorithm is available for a general BTD. This means that the user has to 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, incomplete or structured tensor `T` given an initialization `U0`, the command `btd_nls(T,U0)` can be used. For example,

```
% Generate pseudorandom 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 Uinit 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.

The decomposition in multilinear rank-$(L_r, L_r, 1)$ terms is a special case of a BTD for third-order tensors, in which each term r has multilinear rank $(L_r, L_r, 1)$. For this kind of BTD, there exist efficient algorithms which are preferred over the generic BTD algorithms. See Chapter 4 for more information.

6.2.2 Setting the options

The selected algorithm may be customizable by supplying the method with additional options (see the relevant `help` for more information). These options can be given using an `options` structure, or using key-value pairs, 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);
```

6.2.3 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;
```

6.2.4 Computing the error

The residual between a tensor $\mathcal{T}$ and its BTD defined by the factors $\hat{\mathbf{U}}^{(r,n)}$ and the core tensors $\hat{\mathcal{S}}^{(r)}$ (stored in `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 be computed as

```
relerr = frob(btdres(T,Uhat))/frob(T); % or
relerr = frobbtdres(T,Uhat)/frob(T); % also works for structured tensors
```

6.3 Overview of algorithms

Below we give an overview of the available algorithms for the computation of a BTD, as well as an overview of other convenient algorithms.

The following algorithms can be used to compute the BTD of a dense, sparse, incomplete or efficiently represented tensor:

- `btd_nls(T, U)` computes the BTD of a tensor `T` using nonlinear least squares algorithm with initial guess `U`.
- `btd_minf(T, U)` computes the BTD of a tensor `T` using nonlinear unconstrained optimization with initial guess `U`.

The following routines can be used for problem generation and error analysis.

- `btd_rnd(size_tens, size_core)` generates pseudorandom factors for a BTD.
- `btdgen(U)` generates the full tensor from given factors `U`.
- `btdres(T, U)` computes the residual `btdgen(U,S) - T`.
- `frobbtdres(T, U)` computes the Frobenius norm of the residual `btdgen(U,S) - T`.

Note: When using structured tensors, the expected numerical precision is only half the machine precision. This is due to the mathematical formulation implemented to exploit the structure. In most cases, this is not a problem as the maximal attainable precision is already limited due to noise and model errors. Only in the very specific case when a solution accurate up to machine precision can be found and is required, this can be an issue. (This requires `TolFun` and `TolX` to be `eps^2` and `eps`, respectively.) A solution is to use the structured tensor `T` to compute a solution `Ures` accurate up to half the machine precision, and then refine this solution using `btd_nls(ful(T), Ures, 'TolFun', eps^2, 'TolX', eps)` for a few iterations. (It can be useful to set `MaxIter` to `2` or `3`.) This technique requires only very few expensive iterations involving the full tensor.

6.4 Further reading

Chapter 4 handles a specific type of BTD in which each term has multilinear rank $(L_r, L_r, 1)$. The algorithms outlined in this chapter can be used to compute an unconstrained BTD. In the structured data fusion (SDF) framework constraints can be added as well, and coupled decompositions can be defined. See Chapter 8 for more details.

CHAPTER

SEVEN

TENSORIZATION TECHNIQUES

Tensorization is defined as the transformation or mapping of lower-order data to higher-order data. For example, the low-order data can be a vector, and the tensorized result is a matrix, a third-order tensor or a higher-order tensor. The 'low-order' data can also be a matrix or a third-order tensor, for example. In the latter case, tensorization can take place along one or multiple modes.

> **Example:** A vector $\mathbf{v}$ can be reshaped into a matrix $\mathbf{V}$, or a tensor $\mathcal{V}$ of arbitrary order K as illustrated in Fig. 7.1.

> **Example:** A matrix $\mathbf{M}$ can be reshaped into a higher-order tensor $\mathcal{T}$ by reshaping each row or column of $\mathbf{M}$ to a tensor of order K and stacking the results along dimension $K+1$. This is also illustrated in Fig. 7.1.

In the preceding examples, the transformation or mapping consisted of a reshape, but other mappings can be considered as well. Tensorlab provides a variety of mappings and corresponding tensorization techniques. This chapter gives an overview of these techniques, which can be subdivided into deterministic techniques (Hankelization, Löwnerization, segmentation and decimation) and techniques based on statistics (lagged second-order statistics and higher-order statistics). A survey of these techniques is given in [40].

Given tensorized data, the original low-order data can be retrieved with a **detensorization** step. For example, one can fold the tensor $\mathcal{V}$ back to the vector $\mathbf{v}$. The result is unambiguous if the corresponding mapping is bijective. Tensorlab provides a detensorization routine for each deterministic tensorization method.

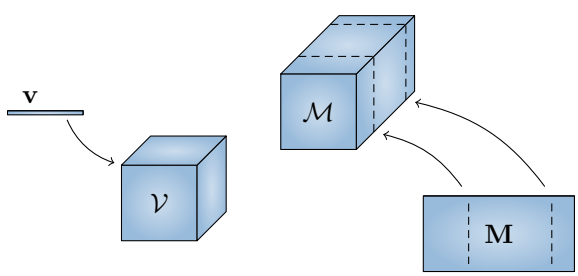

Fig. 7.1: Third-order tensorization of a vector $\mathbf{v}$ to a tensor $\mathcal{V}$ (left), and second-order tensorization of a matrix $\mathbf{M}$ to a tensor $\mathcal{M}$ (right). In the latter case, every column of $\mathbf{M}$ is mapped to a matrix, and the matrices are stacked along the third mode of $\mathcal{M}$.

In this chapter, we discuss the following tensorization and corresponding detensorization techniques:

- Hankelization makes use of a Hankel matrix/tensor mapping. Corresponding Tensorlab commands are `hankelize` and `dehankelize` (explained in Section 7.1).
- Löwnerization makes use of a Löwner matrix/tensor mapping. Corresponding Tensorlab commands are `loewnerize` and `deloewnerize` (explained in Section 7.2).

- Segmentation makes use of reshaping/folding. An advanced variant allows segments to overlap. Corresponding Tensorlab commands are `segmentize` and `desegmentize` (explained in Section 7.3).
- Decimation is similar to segmentation and uses a reshaping/folding mapping as well, but from the point of subsampling. Corresponding Tensorlab commands are `decimate` and `dedecimate` (explained in Section 7.4).
- The computation of second-order statistics with `scov` and `dcov` (explained in Section 7.5)
- The computation of higher-order statistics with `cum3`, `cum4`, `xcum4` and `stcum4` (explained in Section 7.6).

The first four tensorization techniques are deterministic, while the last two groups of techniques make use of statistics. The different commands for the deterministic techniques operate in a similar way. We explain the procedure in detail for the commands corresponding to Hankelization. For the other commands, we limit ourselves to the differences with the Hankelization commands.

7.1 Hankelization

7.1.1 Definition

Second-order Hankelization transforms a vector $\mathbf{v} \in \mathbb{C}^N$ into a Hankel matrix $\mathbf{H} \in \mathbb{C}^{I \times J}$ with $N = I + J - 1$, defined by:

$$(\mathbf{H})_{i,j} = \mathbf{v}_{i+j-1}, \quad \forall i = 1, \ldots, I; j = 1, \ldots, J.$$

Hence, each anti-diagonal (or skew-diagonal) of the matrix $\mathbf{H}$ is constant, as illustrated in the following example.

Example: Using $N = 5$ and $I = J = 3$, a vector $\mathbf{v} \in \mathbb{C}^5 = [1, 2, 3, 4, 5]^{\mathsf{T}}$ is mapped to a matrix $\mathbf{H} \in \mathbb{C}^{3 \times 3}$:

$$\mathbf{H} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}.$$

In general, Kth-order Hankelization transforms a vector $\mathbf{v} \in \mathbb{C}^N$ into a higher-order Hankel tensor $\mathbf{H} \in \mathbb{C}^{I_1 \times \cdots \times I_K}$ with $N = I_1 + \cdots + I_K - K + 1$, defined by:

$$(\mathcal{H})_{i_1, \ldots, i_K} = \mathbf{v}_{i_1 + \cdots + i_K - K + 1}, \quad \forall k = 1, \ldots, K : i_k = 1, \ldots, I_K.$$

A Hankel tensor has constant anti-hyperplanes.

Example: Using $N = 7$ and sizes $I_1 = 4$, $I_2 = 3$ and $I_3 = 2$, a vector $\mathbf{v} \in \mathbb{C}^7 = [1, 2, 3, 4, 5, 6, 7]^{\mathsf{T}}$ can be mapped to a Hankel tensor $\mathcal{H} \in \mathbb{C}^{4 \times 3 \times 2}$ with slices

$$\mathbf{H}_{::1} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{bmatrix}, \ \mathbf{H}_{::2} = \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \\ 5 & 6 & 7 \end{bmatrix}. \tag{7.1}$$

From the previous example, it is clear that the original vector $\mathbf{v}$ appears in the Hankel tensor as follows. The vector $\mathbf{v}$ is divided into K segments, of which consecutive segments overlap with one entry. The first segment appears as the first fiber along the first mode of the Hankel tensor, and the other segments appear as the last fibers along the other modes.

7.1.2 Hankelization in Tensorlab

Hankelization can be carried out with `hankelize`. We discuss some of its features. The `Order`, `Dim`, `Full`, `FullLimit` and `PermToFirst` options are used by the other tensorization commands as well. The `Sizes` and `Ind` options are only used by `hankelize`.

1. First, the user can specify the **order** of the Hankelization with the `Order` option. For a Hankelization of order K, a vector is tensorized to a Hankel tensor of order K. By default, a second-order Hankelization is carried out.

   Example: If `Order = 2`, then a vector $\mathbf{v} \in \mathbb{C}^N$ is mapped to a matrix $\mathbf{V} \in \mathbb{C}^{I \times J}$:

   ```
   V = hankelize(v);  % By default, 'Order' is 2
   ```

   If `Order = 4`, then a vector $\mathbf{v} \in \mathbb{C}^N$ is mapped to a fourth-order tensor $\mathcal{V} \in \mathbb{C}^{I_1 \times I_2 \times I_3 \times I_4}$:

   ```
   V = hankelize(v,'Order',4);
   ```

2. The `Sizes` option can be used to set the values $I_1, \ldots, I_{K-1}$. The size I_K is chosen as $I_K = N - (I_1 + \cdots + I_{K-1}) + K - 1$. By default, $I_1, \ldots, I_K$ are chosen as close to each other as possible.

   Instead of `Sizes`, the `Ind` option can be used as well. The vector $\mathbf{v}$ is partitioned into K segments $\mathbf{v}_{1:d_1}, \mathbf{v}_{d_1:d_2}, \ldots, \mathbf{v}_{d_{K-1}:N}$ indicating the first fiber along the first mode of the Hankel tensor and the last fibers along the other modes. The `Ind` option determines the indices $d_1, \ldots, d_{K-1}$. The `Ind` and `Sizes` options are related by `Ind = cumsum(Sizes)-(0:Order-1)`.

   Example: The third-order Hankel tensor $\mathcal{H} \in \mathbb{C}^{4 \times 3 \times 2}$ from (7.1) can be obtained from a vector $\mathbf{v} \in \mathbb{C}^7$ with

   ```
   H = hankelize(v,'Sizes',[4 3]);
   ```

   Equivalently, it can be obtained with

   ```
   H = hankelize(v,'Ind',[4 6]);
   ```

   as $\mathbf{v}_{1:4}$ is used for the first fiber along the first mode, and $\mathbf{v}_{4:6}$ and $\mathbf{v}_{6:7}$ are used for the last fibers along the second and third modes, respectively.

3. When matrices or tensors are Hankelized, the user can specify along which **mode** the tensorization is carried out with the `Dim` option. All fibers along this mode are Hankelized. Consider an Mth-order tensor $\mathcal{T} \in \mathbb{C}^{N_1 \times \cdots \times N_M}$. If $\mathcal{T}$ is tensorized along mode n with tensorization order K, `hankelize` returns a tensor $\mathcal{H}$ of size $N_1 \times \cdots \times N_{n-1} \times I_1 \times \cdots \times I_K \times N_{n+1} \times \cdots \times N_M$. By default, a tensorization is carried out along the first mode. Let us illustrate with some examples.

   Example: Let

   $$\mathbf{M} = \begin{bmatrix} 1 & 6 \\ 2 & 7 \\ 3 & 8 \\ 4 & 9 \\ 5 & 10 \end{bmatrix}.$$

   If `Dim = 1` and `Order = 2`, each column of $\mathbf{M} \in \mathbb{C}^{5 \times 2}$ is Hankelized. The different Hankel matrices are stacked along the third mode of a tensor $\mathcal{Z} \in \mathbb{C}^{3 \times 3 \times 2}$:

   ```
   Z = hankelize(M);  % By default, 'Dim' is 1 and 'Order' is 2
   ```

$\mathcal{Z}$ has slices

$$\mathbf{Z}_{::1} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}, \ \mathbf{Z}_{::2} = \begin{bmatrix} 6 & 7 & 8 \\ 7 & 8 & 9 \\ 8 & 9 & 10 \end{bmatrix}.$$

Example: Consider a matrix $\mathbf{M} \in \mathbb{C}^{21\times 3}$. If the tensorization order is 4, `hankelize` returns a fifth-order tensor $\mathcal{Z} \in \mathbb{C}^{6\times 6\times 6\times 6\times 3}$ containing Hankel tensors of size $6 \times 6 \times 6 \times 6$:

```
M = randn(21,3);
Z = hankelize(M,'Order',4);
size(Z)
```

```
ans =
  [6    6    6    6    3]
```

Example: Consider a tensor $\mathcal{T} \in \mathbb{C}^{10\times 20\times 10}$ which is Hankelized along the second mode (`Dim = 2`) using third-order Hankelization (`Order = 3`). The `hankelize` command returns a fifth-order tensor $\mathcal{Z} \in \mathbb{C}^{10\times 7\times 8\times 7\times 10}$:

```
T = randn(10,20,10);
Z = hankelize(T,'Dim',2,'Order',3);
size(Z)
```

```
ans =
  [10    7    8    7    10]
```

In this example, each mode-2 vector of length 20 is tensorized to a Hankel tensor of size $7\times 8\times 7$.

4. The Hankelized data can become very large due to the curse of dimensionality. Instead of working with the full Hankelized tensor, an efficient representation can be used for calculations by exploiting the Hankel structure. This efficient representation can be obtained from the second output argument of `hankelize`. Alternatively, it can be retrieved from the first output argument if the `Full` option is set to `false`. Note that the latter avoids the construction and storage of the full Hankelized tensor alltogether. The CPD and decomposition in multilinear rank-$(L_r, L_r, 1)$ terms are computed much faster when using efficient representations of Hankelized datasets than when using the full tensors.

Example: The third-order Hankel tensor of a vector $\mathbf{v}$ of size 1000×1 has size $334 \times 334 \times 334$ by default. Using double precision, this is a dataset of approximately 284 MB. Using

```
[~,H] = hankelize(v,'Order',3)
% or
H = hankelize(v,'Order',3,'full',false)
```

returns the following efficient representation of $\mathcal{H}$, which only requires 9.8 MB:

```
H =
      type: 'hankel'
       val: [1000x1 double]
       dim: 1
     order: 3
       ind: [334 667]
 ispermuted: 0
repermorder: [1 2 3]
      size: [334 334 334]
   subsize: [1x1 struct]
```

Note that `hankelize` automatically detects if the full tensor would become too large. When the storage limit `FullLimit` is exceeded, the full tensor is not constructed, but the efficient representation is returned. The `FullLimit` option is set to 1 (expressed in GB) by default, but can be adjusted by the user.

The full tensor can be obtained from the efficient representation with the `ful` command.

5. The `PermToFirst` option can be used to indicate whether the output modes of the tensorization should be the first modes of the resulting tensor. Consider an Mth-order tensor $\mathcal{T} \in \mathbb{C}^{N_1 \times \cdots \times N_M}$ which is tensorized along the nth mode with tensorization order K. By default, `PermToFirst` is `false`, and `hankelize` returns a tensor of size $N_1 \times \cdots \times N_{n-1} \times I_1 \times \cdots \times I_K \times N_{n+1} \times \cdots \times N_M$. If `PermToFirst` is set to `true`, `hankelize` returns a tensor of size $I_1 \times \cdots \times I_K \times N_1 \times \cdots \times N_{n-1} \times N_{n+1} \times \cdots \times N_M$.

   Example: Consider a tensor $\mathcal{T} \in \mathbb{C}^{10\times 20\times 10}$, which is Hankelized along the second mode (`Dim = 2`) using third-order tensorization (`Order = 3`). By default, a tensor $\mathcal{Z} \in \mathbb{C}^{10\times 7\times 8\times 7\times 10}$ is obtained. However, when setting `PermToFirst = true`, `hankelize` returns a tensor $\mathcal{Z} \in \mathbb{C}^{7\times 8\times 7\times 10\times 10}$:

```
Z = hankelize(T,'Dim',2,'Order',3,'PermToFirst',1);
```

   This option can be useful for decompositions such as the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms, where L_r may correspond to the rank of a Hankel representation.

7.1.3 Dehankelization in Tensorlab

The original data can be extracted from a Hankel matrix or Hankel tensor with the `dehankelize` command, which provides a number of options:

1. The `Order` option indicates the order of the dehankelization. By default, `Order = 2`, and each second-order slice in the first two modes is dehankelized. This is consistent with the default use of second-order tensorization in `hankelize`. If a vector should be extracted from a Kth-order Hankel tensor, `Order` should be set to K.

   Example: Consider a Hankel tensor $\mathcal{H} \in \mathbb{C}^{10\times 10\times 10}$. Dehankelization of $\mathcal{H}$ with

```
M = dehankelize(H);
```

   returns a matrix $\mathbf{M}$ of size 19×10, while setting `Order = 3` returns a vector of length 28:

```
v = dehankelize(H,'Order',3);
```

2. The `Dims` option can be used to indicate along which modes the dehankelization is carried out.

Example: Consider a tensor $\mathcal{H} \in \mathbb{C}^{5\times10\times5\times10\times5}$. The command

```
T = dehankelize(H,'Dims',[1 3 5]);
```

returns a tensor $\mathcal{T}$ of size $13 \times 10 \times 10$.

3. Instead of using the `Dims` option, a combination of the `Dim` options and `Order` can be used for consecutive modes along which the detensorization should take place. The result with `Dim = d` and `Order = K` is the same as with `Dims = d:d+K-1`.

4. The user can indicate how the data is extracted using the `Method` option. We discuss some possibilities:

   - The first column and the last row of a Hankel matrix are retained if `Method` is set to `fibers`. Considering a Kth-order Hankel tensor, the first fiber in the first mode, and the last fibers along the other modes are retained.

   - The data can also be estimated by averaging along the anti-diagonals if `Method` is set to the string `mean` or the function handle `@mean`. This can be useful if the data is noisy and/or if the data does not exactly have Hankel structure. For tensors, the average along the anti-hyperplanes is computed.

     Example: Let

     $$\mathbf{H} = \begin{bmatrix} 1 & 2.1 & 3.1 \\ 1.9 & 3 & 4.1 \\ 2.9 & 4 & 5.1 \\ 3.9 & 4.9 & 6 \end{bmatrix}.$$

     The command

     ```
     v = dehankelize(H,'Method','fibers');
     ```

     returns the concatenated first column and last row, $\mathbf{v} = [1, 1.9, 2.9, 3.9, 4.9, 6]^{\mathsf{T}}$, while

     ```
     v = dehankelize(H,'Method','mean');    % or @mean
     ```

     returns the averaged anti-diagonals $\mathbf{v} = [1, 2, 3, 4, 5, 6]^{\mathsf{T}}$.

   - The user can provide an alternative for the averaging along the anti-diagonals or anti-hyperplanes by passing a function handle to `Method`, such as `@median`:

     ```
     v = dehankelize(H,'Method',@median);
     ```

   By default, `Method` is set to `@mean` as the result is more robust than with the `fibers` method. Note that `@median` is computationally more demanding.

The `dehankelize` command works very efficiently on a polyadic representation of a Hankel tensor. Internally, it avoids the construction of the possibly large full tensor. Hence, while the results are the same,

```
v = dehankelize({U1,U2,U3});
```

is more time- and memory-efficient if the factor matrices are large than

```
T = cpdgen({U1,U2,U3});
v = dehankelize(T);
```

It is possible to dehankelize several polyadic representations at the same time by concatenating the factor matrices (provided that the dimensions agree) and indicating how the rank-1 terms are grouped with the `L` option.

Example: Consider two factor matrices $\mathbf{U}^{(1)} \in \mathbb{C}^{10\times 3}$ and $\mathbf{U}^{(2)} \in \mathbb{C}^{10\times 3}$ for a first representation in rank-1 terms and two factor matrices $\mathbf{V}^{(1)} \in \mathbb{C}^{10\times 2}$ and $\mathbf{V}^{(2)} \in \mathbb{C}^{10\times 2}$ for a second representation in rank-1 terms. The matrices $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ are the restuls of the concatenation of $\mathbf{U}^{(1)}$ and $\mathbf{V}^{(1)}$, and $\mathbf{U}^{(2)}$ and $\mathbf{V}^{(2)}$, respectively. Given $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$, `dehankelize` returns a vector $\mathbf{v} \in \mathbb{C}^{19}$ by default:

```
W = cpd_rnd([10 10],5);        % Concatenated U and V
v = dehankelize(W);
size(v)
```

```
ans =
   [19    1]
```

If `L` is set to `[3 2]`, the matrices $\mathbf{U}^{(1)}\mathbf{U}^{(2)^{\mathsf{T}}}$ and $\mathbf{V}^{(1)}\mathbf{V}^{(2)^{\mathsf{T}}}$ are dehankelized separately, and the computed vectors are stacked in a matrix $\mathbf{M} \in \mathbb{C}^{19\times 2}$:

```
W = cpd_rnd([10 10],5);        % Concatenated U and V
M = dehankelize(W,'L',[3 2]);
size(M)
```

```
ans =
   [19    2]
```

7.1.4 Application example

We consider blind signal separation for a mixture of exponentials to illustrate the use of Hankelization and dehankelization. Four exponential functions, sampled in 101 points $\mathbf{t} = [0, 0.01, 0.02, \ldots, 1]$, are mixed such that three observed signals are obtained. The goal is to determine the mixing coefficients and the exponentials using only the observed signals.

This can be achieved by Hankelizing the observed signals along the mode of the observations. The resulting tensor with Hankel slices is decomposed using a CPD. Estimates of the source signals can be obtained by dehankelizing the estimated source Hankel matrices. Note that, since there are more source signals than observed signals, the source signals cannot be estimated by inverting the estimated mixing matrix. More information on this application can for instance be found in [28].

```
% Signal construction
t = linspace(0,1,101).';
S = exp(t*[1 2 3 4]);
M = randn(4,3);
X = S*M;
% Separation
H = hankelize(X,'Dim',1);
U = cpd(H,4);
Sest = dehankelize(U(1:2),'L',[1 1 1 1]);
% Error calculation
cpderr(S,Sest)
```

```
ans =
   5.3055e-12
```

7.2 Löwnerization

7.2.1 Definition

Consider a vector $\mathbf{f} \in \mathbb{C}^N$ and a vector of evaluation points $\mathbf{t} \in \mathbb{C}^N$. Let us partition $[1, 2, \ldots, N]$ in two vectors $\mathbf{x} \in \mathbb{C}^I$ and $\mathbf{y} \in \mathbb{C}^J$ with $N = I + J$. For example, an interleaved partitioning ($\mathbf{x} = [1, 3, \ldots]$ and $\mathbf{y} = [2, 4, \ldots]$) or block partitioning ($\mathbf{x} = [1, 2, \ldots, I]$ and $\mathbf{y} = [I + 1, \ldots, N]$) can be used. The corresponding Löwner matrix $\mathbf{L} \in \mathbb{C}^{I \times J}$ is defined as follows:

$$(\mathbf{L})_{i,j} = \frac{f_{x(i)} - f_{y(j)}}{t_{x(i)} - t_{y(j)}}, \quad \forall i \in \{1, \ldots, I\}; j \in \{1, \ldots, J\}.$$

Example: Consider a vector $\mathbf{f} = [6, 7, 8, 9]$ and evaluation points $\mathbf{t} = [1, 2, 3, 4]$. The Löwner matrix using interleaved partitioning is given by

$$\mathbf{L} = \begin{bmatrix} \frac{6-7}{1-2} & \frac{6-9}{1-4} \\ \frac{8-7}{3-2} & \frac{8-9}{3-4} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}. \tag{7.2}$$

Löwner matrices can be useful for dealing with rational functions, as there is a close relationship between the degree of the rational function and the rank of the Löwner matrix [41]. We do not discuss higher-order Löwner tensors in this section.

7.2.2 Löwnerization in Tensorlab

Löwnerization can be carried out with the `loewnerize` command. It can be applied using similar options as in `hankelize`:

1. The `Order` option indicates the order of the Löwnerization and is set to two by default. For example, `L = loewnerize(f,'Order',3)`.
2. The `Dim` option indicates the mode along which the Löwnerization needs to be carried out. By default, `Dim = 1`. For example, `L = loewnerize(f,'Dim',2)`.
3. Setting the `Full` option to `false` avoids the construction of the full Löwner matrix or tensor. If the partitioned evaluation points in $\mathbf{x}$ and $\mathbf{y}$ are equidistant, time- and memory-efficient calculations can be performed using the efficient representation. The full Löwnerized tensor can be obtained from the efficient representation with the `ful` command.
4. The `PermToFirst` option can be used to indicate whether the output modes of the tensorization should be the first modes of the result. For example, `L = loewnerize(f,'Dim',2,'PermToFirst',true)` can be used to make the Löwner matrices appear in the first and second mode. By default, `PermToFirst = false`.

Two additional options are provided. These are illustrated in the examples below.

5. The `T` option can be used to set the evaluation points. By default, `T = 1:N`.
6. The `Ind` option can be used to set the partitioning. It is assumed to be a cell array with $K - 1$ distinct sets for Kth-order Löwnerization. The Kth set consists of the remaining indices. By default, an interleaved partitioning is used. If the cell contains only a single array, the cell brackets can be omitted.

Example: The Löwner matrix in (7.2) can be constructed as

```
f = [6 7 8 9];
% By default, 'Order' is 2, 'T' is [1 2 3 4] and 'Ind' is [1,3].
L = loewnerize(f)
```

```
L =

   1     1
   1     1
```

Example: Consider a vector $\mathbf{f} = [1, 4, 9, 16]$ and evaluation points $\mathbf{t} = [2, 4, 6, 8]$. The Löwner matrix with block partitioning ($\mathbf{x} = [1, 2]$ and $\mathbf{y} = [3, 4]$) can be computed as follows:

```
f = [1 4 9 16];
L = loewnerize(f,'T',[2 4 6 8],'Ind',[1 2])
```

```
L =

   2     2.5
   2.5   3
```

7.2.3 Delöwnerization in Tensorlab

Delöwnerization extracts the original data from a Löwner matrix or tensor. This can be done with `deloewnerize` which solves a linear system [41]. The command works in a similar way as `dehankelize`. The user can set the `Dims` option to indicate the modes along which the detensorization should be carried out (`Dims = [1 2]` by default). Alternatively, a combination of `Dim` and `Order` can be used for consecutive modes. In the case of delöwnerization, the `T` option can be used to specify the evaluation points, and the `Ind` option can be used to specify the partitioning (interleaved by default).

The `deloewnerize` command also works on polyadic representations using factor matrices. The `L` option can be used to indicate separate delöwnerizations, similar to `dehankelize`.

The result of the delöwnerization of a Löwner matrix is determined up to a constant offset. The `deloewnerize` command fixes this indeterminacy by returning a vector with zero mean.

7.2.4 Application example

We consider blind signal separation for a mixture of rational functions to illustrate the use of Löwnerization and delöwnerization. Four rational functions, sampled in 101 points $\mathbf{t} = [0, 0.01, 0.02, \ldots, 1]$, are mixed such that three observed signals are obtained. The goal is to determine the mixing coefficients and the rational functions using only the observed signals.

This can be achieved by Löwnerizing the observed signals along the mode of the observations. The resulting tensor with Löwner slices is decomposed using a CPD. Estimates of the source signals can be obtained by delöwnerizing the estimated source Löwner matrices. Note that, since there are more source signals than observed signals, the source signals cannot be estimated by inverting the estimated mixing matrix. More information on this application can for instance be found in [41].

```
% Signal construction
t = linspace(0,1,100).';
% Each column of S consists of a rational function of the form 1/(t-p)
```

```
S = 1./bsxfun(@minus,t,[0.2 0.4 0.6 0.8]);
M = randn(4,3);
X = S*M;
% Separation
H = loewnerize(X,'Dim',1,'T',t);
U = cpd(H,4);
Sest = deloewnerize(U(1:2),'L',[1 1 1 1]);
% Error calculation: compare Sest to S with zero-mean columns
cpderr(bsxfun(@minus,S,mean(S,1)),Sest)
```

```
ans =
   3.1050e-15
```

7.3 Segmentation

7.3.1 Definition

Given a vector $\mathbf{v} \in \mathbb{C}^N$, second-order segmentation without overlapping segments consists in reshaping $\mathbf{v}$ into a matrix $\mathbf{S} \in \mathbb{C}^{I \times J}$ with $N = IJ$. Each column of $\mathbf{S}$ is then called a segment of $\mathbf{v}$ of length I. For example, the segmentation of a vector $\mathbf{v} = [1, 2, \ldots, 6]$ with $I = 3$ and $J = 2$ results in a matrix $\mathbf{S}$ of size 3×2:

$$\mathbf{S} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}.$$

When the order of the segmentation is increased, each segment is segmented again, resulting in a tensor $\mathcal{S} \in \mathbb{C}^{I_1 \times I_2 \times \cdots \times I_K}$ for a Kth-order segmentation of a vector $\mathbf{v} \in \mathbb{C}^N$ with $N = I_1 I_2 \cdots I_K$.

Tensorlab also provides the possibility of using overlapping segments, as discussed below.

7.3.2 Segmentation in Tensorlab

Segmentation can be obtained with the `segmentize` command. Similar to `hankelize` and `loewnerize`, the `Order` option can be used to define the order of the segmentation and the `Dim` option can be used to indicate the mode along which the tensorization is carried out. Setting the `Full` option to `false` avoids the construction of the full tensor, while the `PermToFirst` option allows the user to indicate whether the output modes of the tensorization should be the first modes modes of the resulting tensor.

A number of additional options are available for `segmentize`:

1. The segment length can be set to I for second-order segmentation with the `Segsize` option. The number of segments J is automatically chosen as $J = \lfloor N/I \rfloor$.

   In general, for Kth-order segmentation, the `Segsize` option can be used to set the values $I_1, I_2, \ldots, I_{K-1}$. The value I_K is automatically chosen as $I_K = \lfloor N/(I_1 I_2 \cdots I_{K-1}) \rfloor$.

   > **Example:** Consider a matrix $\mathbf{M} \in \mathbb{C}^{60 \times 6}$ which is tensorized by third-order segmentation along the first mode. If $I_1 = 3$ and $I_2 = 4$, then the resulting tensor $\mathcal{S}$ has size $3 \times 4 \times 5 \times 6$:

```
M = randn(60,6);
S = segmentize(M,'Segsize',[3 4]);
size(S)
```

```
ans =
  [3   4   5   6]
```

2. Equivalently, for second-order segmentation, the `Nsegments` option can be used to set the number of segments J. The segment length I is then chosen as $I = \lfloor N/J \rfloor$.

   In general, for Kth-order segmentation, `Nsegments` can be used to set the values $I_2, I_3, \ldots, I_K$. The value I_1 is automatically chosen as $I_1 = \lfloor N/(I_2 I_3 \cdots I_K) \rfloor$.

   Example: The tensor $\mathcal{S}$ from the previous example can also be constructed as follows:

```
M = randn(60,6);
S = segmentize(M,'Nsegments',[4 5]);
size(S)
```

```
ans =
  [3   4   5   6]
```

3. The `UseAllSamples` option can be set to `true` if an error needs to be thrown if not all entries of the data are used (i.e., if $N \neq I_1 I_2 \cdots I_K$ for Kth-order segmentation). By default, `UseAllSamples` is set to `false` and no error is thrown.

4. The user can specify the shift between consecutive segments with the advanced `Shift` option. If the shift is smaller than the segment length for second-order segmentation, consecutive segments overlap. If the shift is larger than the segment length, some data entries are discarded. By default, `Shift` is equal to the segment length for second-order segmentation so that there is no overlap. The number of segments is chosen such that as many data values are used as possible.

   Example: Consider the segmentization of the vector $\mathbf{v} = [1, 2, \ldots, 9]$ with a shift of `2`:

```
v = 1:9;
S = segmentize(v,'Shift',2)
```

```
S =
   1     3     5     7
   2     4     6     8
   3     5     7     9
```

For general Kth-order segmentation, `Shift` is a vector with $K-1$ entries. Recall the process of Kth-order segmentation with recursive segmentation and without overlap: segments of length $I_1 I_2 \cdots I_{K-r}$ are constructed in the rth segmentation step which are then further segmented in the next step. The shift in step r is indicated by the uth entry of the `Shift` vector with $u = K - r$. Note that the segment length in segmentation step r depends not only on $I_1, I_2, \ldots, I_{K-r}$ but on the first $K - r$ shifts as well. By default, `Shift` is equal to the cumulative product of the values $I_1, I_2, \ldots, I_{K-1}$ so that there is no overlap.

Example: Consider the vector $\mathbf{v} = [1, 2, \ldots, 18]$. By default, using $I_1 = I_2 = 3$ and $I_3 = 2$, the vector $\mathbf{v}$ is first segmented into a matrix of size 9×2. Each column is then segmented to a 3×3 matrix. This results in a third-order tensor $\mathcal{S}^{(1)} \in \mathbb{C}^{3\times3\times2}$:

```
v = 1:18;
S1 = segmentize(v,'Segsize',[3 3]);
```

The tensor $\mathcal{S}^{(1)}$ has slices

$$\mathbf{S}^{(1)}_{::1} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}, \ \mathbf{S}^{(1)}_{::2} = \begin{bmatrix} 10 & 13 & 16 \\ 11 & 14 & 17 \\ 12 & 15 & 18 \end{bmatrix}.$$

In this case, the `Shift` option is set by default to `Shift = [3 9]`. When setting `Shift` to `[3 8]`, the segments of length 9 with starting indices 1 and 9 ($\mathbf{v}_{1:9}$ and $\mathbf{v}_{9:17}$) are created first. This results in a matrix of size 9×2. Each column is segmented again into segments of length 3 without overlap as the first element of `Shift` is 3. `segmentize` returns a tensor $\mathcal{S}^{(2)} \in \mathbb{C}^{3\times3\times2}$:

```
v = 1:18;
S2 = segmentize(v,'Segsize',[3 3],'Shift',[3 8])
```

```
S2(:,:,1) =

   1     4     7
   2     5     8
   3     6     9

S2(:,:,2) =

   9    12    15
  10    13    16
  11    14    17
```

Note that there is an overlapping element with value 9 along the third mode. Alternatively, with `Segsize = [3 3]` and `Shift = [2 7]`, one obtains a tensor $\mathcal{S}^{(3)}$ with slices

$$\mathbf{S}^{(3)}_{::1} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \\ 3 & 5 & 7 \end{bmatrix}, \ \mathbf{S}^{(3)}_{::2} = \begin{bmatrix} 8 & 10 & 12 \\ 9 & 11 & 13 \\ 10 & 12 & 14 \end{bmatrix}.$$

Note that there are overlapping elements along the second mode, but no overlap along the third mode. In this process, $\mathbf{v}$ is first segmented into consecutive segments of length 7 to obtain a matrix of size 7×2. Each column is then segmented again in segments of length 3 with an overlap of 1 as the first shift element is 2.

If `Segsize = [3 3]` and `Shift = [2 5]`, segments of length 7 are constructed first with an overlap of 2. Three such segments can be constructed from $\mathbf{v}$ to obtain a matrix of size 7×3. Each column is then segmented again to segments of length 3 with an overlap of 1, to obtain a tensor $\mathcal{S}^{(4)}$ of size $3 \times 3 \times 3$:

```
v = 1:18;
S4 = segmentize(v,'Segsize',[3 3],'Shift',[2 5]);
```

$\mathcal{S}^{(4)}$ has slices

$$\mathbf{S}^{(4)}_{::1} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \\ 3 & 5 & 7 \end{bmatrix}, \mathbf{S}^{(4)}_{::2} = \begin{bmatrix} 6 & 8 & 10 \\ 7 & 9 & 11 \\ 8 & 10 & 12 \end{bmatrix}, \mathbf{S}^{(4)}_{::3} = \begin{bmatrix} 11 & 13 & 15 \\ 12 & 14 & 16 \\ 13 & 15 & 17 \end{bmatrix}.$$

When setting `Shift = [1,...,1]` with $K-1$ entries, the segments constructed in each segmentation step overlap almost entirely. The result is the same as that of Kth-order Hankelization.

Example: Consider the vector $\mathbf{v} = [1, 2, \ldots, 6]$ which is segmentized with $I_1 = I_2 = 3$ and with `Shift = [1 1]` :

```
v = 1:6;
S = segmentize(v,'Segsize',[3 3],'Shift',[1 1])
```

```
S(:,:,1) =

   1     2     3
   2     3     4
   3     4     5

S(:,:,2) =

   2     3     4
   3     4     5
   4     5     6
```

The same result can be obtained with the `hankelize` command:

```
v = 1:6;
H = hankelize(v,'sizes',[3 3]);
```

Hence, Hankelization with `hankelize` is a special case of segmentation with `segmentize` .

Note that if the `Full` option in `segmentize` is set to `false` , an efficient representation of the segmented tensor is returned. This can be useful to avoid the redundancy due to overlapping segments. Tensorlab does not yet exploit this structure in tensor computations and decompositions, however. The full tensor can be obtained from the efficient representation with the `ful` command.

7.3.3 Desegmentation in Tensorlab

Desegmentation extracts the original data from the segmentized results. The `desegmentize` command works in a similar way as `dehankelize` . The user is able to indicate the modes along which the detensorization is carried out (with `Dims` , or with a combination of `Dim` and `Order`), and to set the `Shift` option. The `Method` option indicates the method to be used for the extraction of the data, such as single fibers, means, medians, ...

Example: Consider the vector $\mathbf{v} = [1, 2, \ldots, 7]$ which is segmentized with overlapping segments along different modes to a tensor $\mathcal{S}$. $\mathcal{S}$ is then desegmentized again. We verify that $\mathbf{v}$ is recovered correctly.

```
v = 1:7;
S = segmentize(v,'Segsize',[3 3],'Shift',[1 2])
```

```
S(:,:,1) =

     1     2     3
     2     3     4
     3     4     5

S(:,:,2) =

     3     4     5
     4     5     6
     5     6     7
```

```
vest = desegmentize(S,'Shift',[1 2])
```

```
vest =
     1
     2
     3
     4
     5
     6
     7
```

7.4 Decimation

7.4.1 Definition

Decimation is a tensorization technique similar to segmentation. Second-order decimation consists of the reshaping of a vector $\mathbf{v} \in \mathbb{C}^N$ into a matrix $\mathbf{D} \in \mathbb{C}^{I \times J}$ such that each column of $\mathbf{D}$ is a subsampled version of $\mathbf{v}$ with a subsampling factor J.

> **Example:** A vector $\mathbf{v} = [1, 2, \ldots, 6]$ can be decimated using a subsampling factor $J = 2$ in the following matrix $\mathbf{D} \in \mathbb{C}^{3 \times 2}$.
>
> $$\mathbf{D} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}.$$

Higher-order decimation consists in further reshaping the obtained vectors, resulting in a tensor $\mathcal{D} \in \mathbb{C}^{I_1 \times I_2 \times \cdots \times I_K}$ for the Kth-order decimation of a vector $\mathbf{v} \in \mathbb{C}^N$ with $N = I_1 I_2 \cdots I_K$.

7.4.2 Decimation in Tensorlab

Decimation can be obtained with the `decimate` command. The command works in a similar way as `segmentize`. The subsampling factor J in second-order segmentation can be set with the `subsample` option, while the `Nsamples` option can be used to set the number of samples I.

> **Example:** The matrix $\mathbf{D}^{(1)} \in \mathbb{C}^{6 \times 2}$ is obtained from the vector $\mathbf{v} = [1, 2, \ldots, 12]$ when decimating $\mathbf{v}$ with a subsampling factor of 2:

```
D1 = decimate(1:12,'subsample',2)
```

```
D1 =
   1     2
   3     4
   5     6
   7     8
   9    10
  11    12
```

while the matrix $\mathbf{D}^{(2)} \in \mathbb{C}^{3\times 4}$ is obtained when decimating $\mathbf{v}$ with a subsampling factor of 4:

```
D2 = decimate(1:12,'subsample',4)
```

```
D2 =
  1     2     3     4
  5     6     7     8
  9    10    11    12
```

For Kth-order decimation, the values $I_2, \ldots, I_K$ are set with `Subsample` while the values $I_1, \ldots, I_{K-1}$ are set with `Nsamples`.

The `decimate` command supports the other options of `segmentize` such as `Shift`, `Full` and `PermToFirst` as well. We refer the interested reader to Section 7.3.

7.4.3 Dedecimation in Tensorlab

The original data can be extracted from the decimated version with the `dedecimate` command. The command works in a similar way as `desegmentize`, supporting options such as `Dims`, `Dim`, `Order`, `Shift` and `Method`. We refer the interested reader to the discussion of desegmentation in Section 7.3.

7.5 Computing second-order statistics

7.5.1 Computing covariance matrices along different modes

The Matlab command `cov` calculates the variance of a data vector or the covariance matrix of a data matrix, by assuming that each row represents an observation and each column represents a variable. This procedure maps a matrix $\mathbf{X} \in \mathbb{C}^{N\times R}$ to a covariance matrix $\mathbf{C} \in \mathbb{C}^{R\times R}$.

The `dcov` command generalizes the functionality of `cov` to tensor data by allowing the user to indicate the modes along which the tensorization is carried out. The `Dims` option is an array with two elements, of which the first entry indicates the mode of the observations and of which the second entry indicates the mode of the variables.

When applying `dcov` on a data tensor $\mathcal{X} \in \mathbb{C}^{N\times I\times J}$ with `Dims = [1 2]`, a tensor $\mathcal{C} \in \mathbb{C}^{I\times I\times J}$ is returned. $\mathcal{C}$ contains covariance matrices along the third mode. When applying `dcov` on a data tensor $\mathcal{X} \in \mathbb{C}^{I\times J\times N\times R}$ with `Dims = [3 4]`, a tensor $\mathcal{C} \in \mathbb{C}^{I\times J\times R\times R}$ is returned.

7.5.2 Computing lagged covariance matrices

Lagged (or shifted) covariance matrices can be computed with `scov`. The first argument is the data matrix, and the second argument contains the lags. By definition, the lags should be positive. A third input argument indicates whether the data is prewhitened first. The lagged covariance matrices are stacked along the third mode of the resulting tensor.

Example: Given a data matrix $\mathbf{X} \in \mathbb{C}^{1000\times 10}$ and using the lags $[0, 1, 2]$, `scov` returns a tensor $\mathcal{C} \in \mathbb{C}^{10\times 10\times 3}$:

```
X = randn(1000,10);
C = scov(X,[0 1 2]);  % No prewhitening by default
```

7.6 Computing higher-order statistics

7.6.1 Computing third- and fourth-order cumulants

The `cum3` and `cum4` commands can be used to compute third-order and fourth-order cumulants, respectively. These commands also return the third-order and fourth-order moments as the second output arguments. The first input argument is the data matrix, in which each row represents an observation and in which each column represents a variable.

Example: Given a data matrix $\mathbf{X} \in \mathbb{C}^{1000\times 10}$, the third-order cumulant $\mathcal{C}^{(3)}$ has size $10 \times 10 \times 10$ and the fourth-order cumulant $\mathcal{C}^{(4)}$ has size $10 \times 10 \times 10 \times 10$:

```
X = randn(1000,10);
C3 = cum3(X);
size(C3)
```

```
ans =
   [10  10  10]
```

```
C4 = cum4(X);
size(C4)
```

```
ans =
   [10  10  10  10]
```

A second input argument can be used to indicate whether the data is prewhitened first:

```
C4 = cum4(X,'prewhitening');
```

7.6.2 Computing fourth-order cross- and spatio-temporal cumulant

Given four data matrices, the `xcum4` command computes the fourth-order cross-cumulant. For matrices $\mathbf{A} \in \mathbb{C}^{N\times I}$, $\mathbf{B} \in \mathbb{C}^{N\times J}$, $\mathbf{C} \in \mathbb{C}^{N\times K}$ and $\mathbf{D} \in \mathbb{C}^{N\times L}$, the obtained cumulant has size $I \times J \times K \times L$.

The `stcum4` command computes the fourth-order spatio-temporal cumulant. Consider a data matrix $\mathbf{X} \in \mathbb{C}^{N\times I}$ and lag parameter L. `stcum4` returns a seventh-order tensor of size $I \times I \times I \times I \times K \times K \times K$ with $K = 2L+1$.

For a discussion of higher-order statistics, we refer to [42].

CHAPTER

EIGHT

STRUCTURED DATA FUSION

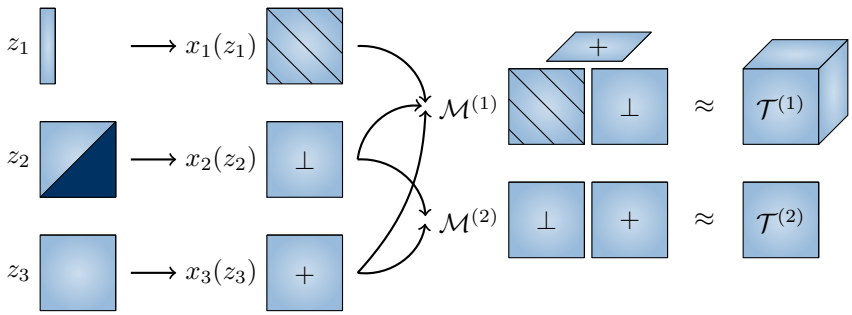

Fig. 8.1: Schematic of structured data fusion. The vector z_1, upper triangular matrix z_2 and full matrix z_3 are transformed into a Toeplitz, orthogonal and nonnegative matrix, respectively. The resulting factors are then used to jointly factorize two coupled data sets.

Structured data fusion or SDF [36] comprises a modeling language, a syntax checker and a number of solvers to jointly factorize one or more datasets, impose symmetry in the factorization, and optionally apply structure on the different factors. Each data set — stored as a dense, sparse, incomplete or structured tensor, cf. Section Chapter 2 — in a data fusion problem can be factorized with a different tensor decomposition. Currently, the user has the choice of the CPD, LL1, LMLRA and BTD models, as well as L0, L1 and L2 regularization terms. 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.

This section gives a basic introduction to SDF by explaining how it works and how different models can be constructed. In Chapter 9 specific examples are given. After that more advanced concepts are introduced in Chapter 10 which enable the creation of more dynamical models. Chapter 11 explains how a new transformation can be defined to impose a new kind of structure on a factor. Finally, a more technical specification of the modeling language is given in Chapter 12.

In this section the following topics are handled:

- Introducing SDF: elements of a data fusion model (Section 8.1)
- Using `sdf_check` to check the syntax (Section 8.2)
- Using different solvers (Section 8.2)
- Coupling datasets and setting weights (Section 8.3)
- Imposing symmetry (Section 8.3)
- Imposing structure on factors (Section 8.4)

- Using constant factors and subfactors (Section 8.4)
- The different factorization and regularization types (Section 8.5)
- The accepted data types including full, sparse, incomplete and structured tensors (Section 8.5)

8.1 Elements of an SDF model

The structured data fusion framework uses a domain specific language allowing one to express the coupling between datasets and the structure of different factors easily. Fig. 8.1 illustrates the rationale behind the language by showing how an SDF model is built. Structure can be imposed on factors by transforming variables z into factors x. The resulting factors can then be used in factorizations of different datasets. Coupling between the datasets is expressed by using one or more common factors in the different factorizations. Similarly, symmetry can be imposed by using a factor multiple times in the same factorization.

Hence, there are three important elements in an SDF model: `variables`, `factors` and `factorizations`. To compute a rank-R CPD of a third-order tensor $\mathcal{T}$ for example, the following code can be used:

```
R = 5;
T = ful(cpd_rnd([10 11 12], R));
model = struct;
model.variables.a = randn(size(T,1), R);
model.variables.b = randn(size(T,2), R);
model.variables.c = randn(size(T,3), R);
model.factors.A = 'a';
model.factors.B = 'b';
model.factors.C = 'c';
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};
```

It is advised to write `model = struct;` before a model is constructed in order to be sure that no elements from a previous model are used. In this model, the value provided to a variable is the initial guess for that particular variable. The line `model.factors.A` simply states that a factor with the name `A` is created, and it is equal to the variable `a`. The `'a'` is called a (named) reference. Similarly, a (named) factorization `tensor` is created, which factors the tensor `T` in the `data` field as a `cpd` with factors `A`, `B` and `C`. A factorization can only use references to factors (unless implicit factors are enabled, see Chapter 10). Apart from the CPD, other factorizations can be used as well, e.g., `btd`, `ll1`, `lmlra`, `regL0`, `regL1` and `regL2` (see Section 8.5).

Fig. 8.1 shows a typical case of structured data fusion. Two datasets $\mathcal{T}^{(1)}$ and $\mathcal{T}^{(2)}$ are jointly factorized. Both datasets share the factor matrices x_2 and x_3. Each factor matrix has a different structure. Factor matrix x_1 has a Toeplitz structure, x_2 has orthonormal columns and x_3 is nonnegative. In the SDF model, every structure is achieved by transforming some variables into factors. The first factor x_1 is created by transforming a vector z_1 into the Toeplitz matrix. A (vectorized) upper triangular matrix with the parameters for Householder reflectors z_2 can be transformed into factor x_2. Finally, squaring the variables in z_3 results in a nonnegative factor x_3.

For an overview of the different transformations or structures that can be applied, we refer to `help Contents.m`. All functions starting with `struct_` are transformations.

8.2 Checking and solving a model

Starting from Tensorlab 3.0 an overview of the factors and factorizations in an SDF model can be printed, after interpretation by the language parser. This way, the different transformations can be inspected, and the user can

see if the model is interpreted as he or she intended. The language parser, or syntax and consistency checker, is called `sdf_check`. This method interprets the model and converts it to the internal format. If a syntax error is found, e.g., because no variables are defined, or because a reference to a non-existing factor is made, an informative error is thrown. For example:

```
R = 5;
T = ful(cpd_rnd([10 11 12], R));
model = struct;
model.variables.a = randn(size(T,1), R);
model.variables.b = randn(size(T,2), R);
model.variables.c = randn(size(T,3), R);
model.factors.A = 'a';
model.factors.B = 'b';
model.factors.C = 'c';
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'D'};

sdf_check(model);
```

```
Error in line model.factorizations.tensor.cpd:

    Factorization tensor (id = 1) depends on the unknown factor D
```

In many cases, some parts of the error message are underlined, indicating that this is clickable (only available in the Matlab Command Window). For example, clicking on `model.factorizations.tensor.cpd` indeed gives the content of the line containing the error:

```
ans =
 'A'    'B'    'D'
```

If a consistency error is found, an error is thrown (using `sdf_check(model)`), or a message is printed in the model overview if the `'print'` option is provided (`sdf_check(model, 'print')`). Examples of consistency errors are that the number of columns in the factor matrices is not the same in the case of a CPD, or that the size of the tensor generated by the factors does not match the size of the data. Consider the following model:

```
R = 5;
T = ful(cpd_rnd([10 11 12], R));
model = struct;
model.variables.a = randn(11, R);
model.variables.b = randn(10, R);
model.variables.c = randn(12, R);
model.factors.A = 'a';
model.factors.B = 'b';
model.factors.C = 'c';
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};

sdf_check(model,'print');
```

```
Factor                Size                  Comment
-----------------------------------------------------------------------
A                     [11x5]
B                     [10x5]
```

```
C                    [12x5]

Factorization        Type       Factors              Comments
-----------------------------------------------------------------------
tensor (id = 1)      cpd        A,B,C                Dimension mismatch
```

The comment for the factorization states `Dimension mismatch`. In the Matlab Command Window, this is a link and when clicked, the following error is shown:

```
Error in line model.factorizations.tensor.cpd:

    Factorization tensor (id = 1): the size of the tensor generated by the
    factors [11x10x12] should match the size of the data [10x11x12]
```

This is indeed the problem. For more complicated models, this often helps locating possible errors. If `sdf_check(model, 'print')` does not give an error, the available solvers in Tensorlab should run without problems. If there is a syntax or consistency error, the solvers will not run, as the model is invalid (`sdf_check` is called internally as well).

When a model is valid, i.e., no syntax or consistency errors are found, it can be computed using one of the following solvers:

- `sdf_minf` or `sdf_nls`: a general purpose solver for SDF models.
- `ccpd_minf` or `ccpd_nls`: a specialized solver for symmetric and/or coupled CPDs only, in which no structure, constants or concatenations are allowed. These solvers are recommended when they can be used.

The solvers can be called as follows:

```
R = 5;
T = ful(cpd_rnd([10 11 12], R));
model = struct;
model.variables.a = randn(size(T,1), R);
model.variables.b = randn(size(T,2), R);
model.variables.c = randn(size(T,3), R);
model.factors.A = 'a';
model.factors.B = 'b';
model.factors.C = 'c';
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};

sdf_check(model, 'print'); % recommended

sol = sdf_nls(model);
```

The solution is a struct containing a `variables` field and a `factors` field with the results:

```
sol.variables
```

```
ans =
    a: [10x5 double]
    b: [11x5 double]
    c: [12x5 double]
```

```
sol.factors
```

```
ans =
    A: [10x5 double]
    B: [11x5 double]
    C: [12x5 double]
```

The `ccpd_minf` and `ccpd_nls` solvers use a different output format for the solution. As there are no transformations on the variables, only the factors are given as output as a cell, e.g., for the model above:

```
Ures = ccpd_nls(model)
```

```
Ures =
 [10x5 double]    [11x5 double]    [12x5 double]
```

Similar to the other optimization routines, extra options can be provided, and extra information on the convergence of the algorithm can be inspected using:

```
% Using key-value pairs
[sol, output] = sdf_nls(model, 'Display', 10, 'MaxIter', 200);

% Using an options structure
options = struct;
options.Display = 10;
options.MaxIter = 200;
[sol, output] = sdf_nls(model, options);
```

The `output` contains convergence information, how many iterations are carried out, etc. The `output` also contains the absolute and relative errors for each factorization in `output.abserr` and `output.relerr`. The errors are computed as the Frobenius norm of the difference between the input data and the tensor generated by the model, e.g., for the example above it is `output.abserr = frob(T - cpdgen(sol.factors.A, sol.factors.B, sol.factors.C))`.

8.3 Imposing coupling and symmetry

The data fusion part in Structured Data Fusion means that different datasets can be factorized or decomposed together while sharing factors or underlying variables. In this section a basic coupling between datasets is discussed, followed by a discussion on how different weights (or hyperparameters) can be set. Finally, symmetric decompositions are discussed.

> **Note:** The following examples are meant to explain the different modeling techniques. Actually solving the models may not yield a good solution, as the examples do not necessarily reflect good numerical choices and the initialization strategies are not discussed.

8.3.1 Coupling datasets

Consider the following coupled tensor matrix factorization problem

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C}} \frac{1}{2}\,||\mathcal{T} - \mathcal{M}_{\mathsf{CPD}}(\mathbf{A},\mathbf{B},\mathbf{C})||_{\mathsf{F}}^2 + \frac{1}{2}\,||\mathbf{M} - \mathcal{M}_{\mathsf{CPD}}(\mathbf{C},\mathbf{D})||_{\mathsf{F}}^2$$

Note that $\mathcal{M}_{\mathsf{CPD}}(\mathbf{C},\mathbf{D}) = \mathbf{C}\mathbf{D}^{\mathsf{T}}$. This can be modeled easily using SDF by defining two factorizations that have a factor in common:

```
T = randn(10,11,12);
M = randn(12,13);
R = 4;
model = struct;
model.variables.a = randn(size(T,1),R);
model.variables.b = randn(size(T,2),R);
model.variables.c = randn(size(T,3),R);
model.variables.d = randn(size(M,2),R);
model.factors.A = 'a';
model.factors.B = 'b';
model.factors.C = 'c';
model.factors.D = 'd';
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};
model.factorizations.matrix.data = M
model.factorizations.matrix.cpd  = {'C', 'D'};

sdf_check(model, 'print');
```

As can be seen below, the two factorizations indeed share the common factor matrix $\mathbf{C}$:

```
Factor                Size                  Comment
------------------------------------------------------------------------
A                     [10x4]
B                     [11x4]
C                     [12x4]
D                     [13x4]

Factorization         Type        Factors               Comments
------------------------------------------------------------------------
tensor (id = 1)       cpd         A,B,C
matrix (id = 2)       cpd         C,D
```

8.3.2 Absolute and relative weights

Often different weights are needed for the different factorization terms, i.e., the problem

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C}} \frac{\lambda_1}{2} ||\mathcal{T} - \mathcal{M}_{\text{CPD}}(\mathbf{A},\mathbf{B},\mathbf{C})||_{\text{F}}^2 + \frac{\lambda_2}{2} ||\mathbf{M} - \mathcal{M}_{\text{CPD}}(\mathbf{C},\mathbf{D})||_{\text{F}}^2$$

is solved instead. Using SDF the weights λ_1 and λ_2 can be given as absolute weights and as relative weights. Absolute weights can be provided using the `weight` field:

```
% variables and factors omitted
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};
model.factorizations.tensor.weight = lambda1;
model.factorizations.matrix.data = M
model.factorizations.matrix.cpd  = {'C', 'D'};
model.factorizations.matrix.weight = lambda2;
```

Note that the factor $\frac{1}{2}$ in the objective function is present for implementation purposes. Therefore `lambda1` is equal to λ_1 and not $\frac{\lambda_1}{2}$. To get a term $||\mathcal{T} - \mathcal{M}_{\text{CPD}}(\mathbf{A},\mathbf{B},\mathbf{C})||_{\text{F}}^2$, λ_1 should be set to 2.

Often, it is easier to use relative weights. The relative weights ρ_1 and ρ_2 can be given as

```
% variables and factors omitted
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};
model.factorizations.tensor.relweight = rho1;
model.factorizations.matrix.data = M
model.factorizations.matrix.cpd  = {'C', 'D'};
model.factorizations.matrix.relweight = rho2;
```

Relative weights ρ_i are automatically translated to absolute weights using the following expression:

$$\lambda_i = 2\frac{\rho_i}{\sum_j \rho_j}\frac{1}{N_i}$$

in which N_i is the number of known entries in tensor `Ti`, i.e., `Ni = prod(getsize(Ti))` unless `Ti` is incomplete, then `Ni = length(Ti.val)`. For regularization terms $N_i = \sum_f N_i^{(f)}$ in which $N_i^{(f)}$ is the number of entries in factor f after expansion. For example:

```
model = struct;
model.variables.a = rand(10, 5);
model.variables.b = rand(1, 5);
model.factors.A = 'a';
model.factors.B = {'b', @struct_diag};
model.factorizations.reg.regL1 = {'A', 'B'};
```

Factors `A` and `B` are regularized. Factor `A` has size 10×5. Therefore, $N_1^{(A)} = 50$. After expansion, factor `B` has size 5×5 (see `sdf_check(model, 'print')`). Hence, $N_1^{(B)} = 25$. Together we have $N_1 = 75$.

Absolute and relative weights cannot be mixed, i.e., either all factorizations define a `weight` field (or no field), or all factorizations define a `relweight` field (or no field). Omitted (relative) weights are assumed to be 1. If absolute nor relative weights are given, relative weights equal to 1 are assumed.

8.3.3 Imposing symmetry

Finally, some special attention is given to decompositions involving symmetry. Symmetry can easily be imposed by using the same factor multiple times in a factorization. For example, consider a fourth-order tensor $\mathcal{T}$ that is symmetric in the first two modes and in the last two modes:

$$\min_{\mathbf{A},\mathbf{B}} \frac{1}{2}||\mathcal{T} - \mathcal{M}_{\text{CPD}}(\mathbf{A},\mathbf{A},\mathbf{B},\mathbf{B})||_{\text{F}}^2$$

This can be implemented as:

```
R  = 5;
% create symmetric tensor
U0 = cpd_rnd([10 11], R);
U0 = U0([1 1 2 2]);
T  = cpdgen(U0);
% Create model
model = struct;
model.variables.a = randn(size(T,1), R);
model.variables.b = randn(size(T,3), R);
model.factors.A = 'a';
```

```
model.factors.B = 'b';
model.factorizations.symm.data = T;
model.factorizations.symm.cpd  = {'A', 'A', 'B', 'B'};
```

Often when imposing symmetry, especially symmetry in all modes, it is useful to give each rank-1 term a weight α_r:

$$\mathcal{T} \approx \sum_{r=1}^{R} \alpha_r \mathbf{u}_r^{(1)} \otimes \mathbf{u}_r^{(1)} \otimes \cdots \otimes \mathbf{u}_r^{(N)}.$$

This weight α_r allows some extra freedom. For example, in the case of a fourth-order tensor and a factorization $\mathcal{M}_{\text{CPD}}(\mathbf{A}, \mathbf{A}, \mathbf{A}, \mathbf{A})$, the diagonal entries of the result are always nonnegative, while it is perfectly possible to have a fully symmetric tensor with negative diagonal entries. Another example imposes orthogonality structure (`struct_orth`) on all factors $\mathbf{A}$ in $\mathcal{M}_{\text{CPD}}(\mathbf{A}, \mathbf{A}, \mathbf{A})$. As `struct_orth` normalizes each column to have norm one, each rank-1 term also has norm one, which can result in a bad fit. In these two examples, adding the weight α_r is important to obtain a meaningful fit. Adding the weights α_r can easily be done in Tensorlab by creating an additional factor matrix of size $1 \times R$, i.e., a row vector:

```
R  = 5;
% create symmetric tensor
U0 = cpd_rnd([10 11], R);
U0 = U0([1 1 2 2]);
T  = cpdgen(U0);
% Create model
model = struct;
model.variables.a = randn(size(T,1), R);
model.variables.b = randn(size(T,3), R);
model.variables.c = randn(1, R);
model.factors.A = 'a';
model.factors.B = 'b';
model.factors.C = 'c';
model.factorizations.symm.data = T;
model.factorizations.symm.cpd  = {'A', 'A', 'B', 'B', 'C'};
```

This is a valid model, as can be verified using `sdf_check(model, 'print')`. The reason is that $\mathcal{T}$ can be seen as a tensor of size $10 \times 10 \times 11 \times 11 \times 1$.

8.4 Imposing structure on factors

8.4.1 Structure by transformation of variables

Using the SDF language, it is easy to create structured factors. The way to do so, is by transforming a variable into a factor using some transformation. Some examples:

- Nonnegativity can be imposed on a factor by squaring underlying variables (`struct_nonneg`) [37].
- A factor matrix in which each column is a polynomial evaluated in some points can be modeled by the coefficients of these polynomials. These coefficients are the variables (`struct_poly`).
- Orthogonality can be imposed using Householder reflectors as underlying variables (`struct_orth`).

See `help Contents.m` to get an overview of all implemented transformations. All functions starting with `struct_` are transformations. New transformations can be defined as well (see Chapter 11).

In Tensorlab, structure can be imposed on factors as follows. For example, to impose nonnegativity on factors `A` and `B` the following model can be used:

```
M = rand(10,10);
model = struct;
model.variables.a = randn(10,3);
model.variables.b = randn(10,3);
model.factors.A = {'a', @struct_nonneg};
model.factors.B = {'b', @struct_nonneg};
model.factorizations.matrix.data = M;
model.factorizations.matrix.cpd  = {'A', 'B'};

sdf_check(model, 'print');
```

```
Factor                Size                  Comment
-------------------------------------------------------------------------
A                     [10x3]
B                     [10x3]

Factorization         Type        Factors               Comments
-------------------------------------------------------------------------
matrix (id = 1)       cpd         A,B
```

In the Matlab Command Window, `A` and `B` can be clicked to get the different transformations applied to the factors, e.g. for `A`:

```
Expansion of factor A (id = 1) (view):
  (1,1)  a                [10x3]     struct_nonneg        Factor          [10x3]
```

When clicking `a` and `Factor` we see indeed that every number in `a` is squared:

```
ans =
 0.1383    0.9000   -0.6235
-0.3784   -1.5312   -1.3501
 0.1431    0.5046   -1.1622
 1.6050   -0.8642   -0.9443
 1.3491   -0.3766   -0.6712
-0.4484    0.7880    0.5767
 0.1684    0.2982   -2.0858
-1.1205   -0.1637    0.2360
 0.4007    0.6067   -0.7784
 0.7391    1.6345    1.0996
```

```
ans =
 0.0191    0.8101    0.3887
 0.1432    2.3446    1.8228
 0.0205    0.2546    1.3508
 2.5761    0.7469    0.8917
 1.8201    0.1418    0.4505
 0.2011    0.6210    0.3326
 0.0284    0.0889    4.3504
 1.2555    0.0268    0.0557
 0.1606    0.3681    0.6059
```

```
  0.5462    2.6716    1.2090
```

For some transformations extra parameters are required. For example, to model a factor matrix in which each column is a polynomial, `struct_poly` can be used. `struct_poly` needs the evaluation points t_i for these polynomials as extra parameter. In the example below, each column of factor `A` is a polynomial of maximal degree d:

```
M = rand(11,10);
t = 0:0.1:1; % evaluation points
d = 4;       % degree
R = 3;       % rank
model = struct;
model.variables.c = randn(R,d+1); % note that c has R rows!
model.variables.b = randn(10,R);
model.factors.A = {'c', @(z,task) struct_poly(z,task,t)};
model.factors.B = {'b', @struct_nonneg};
model.factorizations.matrix.data = M;
model.factorizations.matrix.cpd  = {'A', 'B'};

sdf_check(model, 'print');
```

```
Factor                Size                 Comment
-----------------------------------------------------------------------
A                     [11x3]
B                     [10x3]

Factorization         Type       Factors              Comments
-----------------------------------------------------------------------
matrix (id = 1)       cpd        A,B
```

As can be seen, `A` is an 11×3 matrix. If `A` is clicked, the effect of the transformation can be seen:

```
Expansion of factor A (id = 1) (view):
  (1,1)  c               [3x5]      struct_poly          Factor          [11x3]
```

`Factor` can be clicked again to get the resulting factor. Note that when a link, e.g, `Factor` or `view`, is clicked, the result becomes available as `ans`. Hence, the resulting factor `A` can be plotted using

```
A = ans;
plot(t, A);
```

Finally, different transformations can be chained. For example:

```
M = rand(11,10);
t = 0:0.1:1; % evaluation points
d = 4;       % degree
R = 3;       % rank
model = struct;
model.variables.c = randn(R,d+1); % note that c has R rows!
model.variables.b = randn(10,R);
model.factors.A = {'c', @(z,task) struct_poly(z,task,t), @struct_nonneg};
model.factors.B = {'b', @struct_nonneg};
model.factorizations.matrix.data = M;
```

```
model.factorizations.matrix.cpd  = {'A', 'B'};

sdf_check(model, 'print');
```

Clicking `A` again, allows us to view the individual transformations:

```
Expansion of factor A (id = 1) (view):
  (1,1)  c               [3x5]      struct_poly          temp            [11x3]
         temp            [11x3]     struct_nonneg        Factor          [11x3]
```

Any number of transformations can be chained, as long as the final result is a valid factor:

```
M = rand(11,10);
t = 0:0.1:1; % evaluation points
d = 4;       % degree
R = 5;       % rank

diagonal = @struct_diag;
poly     = @(z,task) struct_poly(z,task,t);
nonneg   = @struct_nonneg;
negate   = @(z,task) struct_matvec(z, task, [], -eye(R));

model = struct;
model.variables.c = randn(1,d+1); % note that c has R rows!
model.variables.b = randn(10,R);
model.factors.A = {'c', diagonal, poly, nonneg, negate};
model.factors.B = {'b', nonneg};
model.factorizations.matrix.data = M;
model.factorizations.matrix.cpd  = {'A', 'B'};

sdf_check(model, 'print');
```

This example creates a factor `A` in which each column r is negative polynomial of the form $-(t^{r-1})^2$. For readability the shorter names for the transformations have been created first.

8.4.2 Constant factors

Constant factors can be defined as follows. Consider the following CPD in which the second factor matrix is kept constant:

```
T = randn(10, 11, 12);
R = 5;
model = struct;
model.variables.a = randn(size(T,1), R);
model.variables.c = randn(size(T,3), R);
model.factors.A = 'a';
model.factors.B = rand(size(T,2), R);
model.factors.C = 'c';
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};

sdf_check(model, 'print');
```

```
Factor                Size                  Comment
---------------------------------------------------------------------------
A                     [10x5]
B                     [11x5]                Constant
C                     [12x5]

Factorization         Type        Factors               Comments
---------------------------------------------------------------------------
tensor (id = 1)       cpd         A,B,C
```

The factor `B` is indeed recognized as constant. Constants follow the same rules as variables. This means that constants can be cells or numerical arrays of arbitrary dimensions (except a scalar, see Chapter 12), and that constants can be transformed.

Subsection 8.4.3 explains how partially constant factors can be constructed. Here, we only consider a special case in which specific entries are constant. As an example, let the strictly upper triangular part of one factor matrix, say `A`, be fixed to zero. The following initial value is created:

```
A = randn(10,5);
[i,j] = ndgrid(1:10,1:5);
A(j > i) = 0
```

```
A =
 2.0684         0         0         0         0
 0.1294   -0.1904         0         0         0
 0.5001   -0.1355   -0.3705         0         0
 1.9050    0.8140   -0.3259   -0.5112         0
 0.5555    1.0007    1.6666   -0.4472   -1.8578
-0.4579    0.2320    1.2689   -1.6985   -0.7134
-0.1454   -0.0285   -1.0390   -0.8552    0.7025
-0.2466   -0.9286   -0.8979    1.6053   -0.0543
-0.3119   -1.6001    0.7471    1.5946   -0.2169
 1.0205   -0.7063   -1.3862    1.1280   -1.6644
```

To keep the strictly upper triangular part constant, the transformation `struct_const` can be used using the additional `mask` argument (see `help struct_const`). This mask is true except for the entries in the strictly upper triangular part:

```
mask = true(size(A));
mask(i > j) = false;
```

The model can now be created as:

```
T = randn(10, 11, 12);
model = struct;
model.variables.a = A;
model.variables.b = randn(11, 5);
model.variables.c = randn(12, 5);
model.factors.A = {'a', @(z,task) struct_const(z,task,mask)};
model.factors.B = 'b';
model.factors.C = 'c';
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};
```

The tensor decomposition is now computed in the same way as in the unconstrained case, with the exception that the derivatives of the entries in the strictly upper triangular part of `A` are explicitly set to zero. The entries in the strictly upper triangular part therefore stay the same as in the initial value.

8.4.3 Subfactors

In the previous examples, we have assumed that every factor consists of a single subfactor. Factors may however consist of an arbitrary number of subfactors. Consider the following example in which one factor has a block-diagonal structure:

```
model = struct;
model.variables.a11 = randn(5,2);
model.variables.a22 = randn(5,3);
model.variables.b   = randn(11,5);
model.variables.c   = randn(12,5);
model.factors.A = {'a11', zeros(5,3); zeros(5,2), 'a22'};
model.factors.B = 'b';
model.factors.C = 'c';
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};

sdf_check(model, 'print');
```

```
Factor                Size                 Comment
-----------------------------------------------------------------------
A                     [10x5]               Partially constant
B                     [11x5]
C                     [12x5]

Factorization         Type       Factors              Comments
-----------------------------------------------------------------------
tensor (id = 1)       cpd        A,B,C
```

The factor `A` is partially constant as the off-diagonal blocks do not depend on any variable. When clicking on `A` in the Matlab Command Window, an overview for this factor matrix is printed:

```
Expansion of factor A (id = 1) with structure [2x2] (view):
  (1,1)  a11             [5x2]
  (2,1)  (Constant)      [5x2]
  (1,2)  (Constant)      [5x3]
  (2,2)  a22             [5x3]
```

This overview correctly states that `A` consists of `2x2` subfactors. Each entry in this 2×2 structure can be accessed individually as before. It is also possible to impose structure on a subfactor, e.g., nonnegativity can be imposed on block `a11` as follows:

```
model.factors.A = { {'a11', @struct_nonneg}, zeros(5,3);
                    zeros(5,2),              'a22'       };
```

Running `sdf_check(model, 'print')` again, and clicking on `A` gives the expected output:

```
Expansion of factor A (id = 1) with structure [2x2] (view):
  (1,1)  a11              [5x2]      struct_nonneg        Factor          [5x2]
  (2,1)  (Constant)       [5x2]
  (1,2)  (Constant)       [5x3]
  (2,2)  a22              [5x3]
```

In the previous examples, all factors are actually matrices or vectors. A factor can be a tensor as well, and it is also possible to concatenate subfactors along modes higher than two, for example:

```
model = struct;
model.variables.u  = randn(10,2);
model.variables.v  = randn(11,3);
model.variables.w  = randn(12,3);
model.variables.s1 = randn(2,3,2);
model.variables.s2 = randn(2,3,1);
model.factors.U = 'u';
model.factors.V = 'v';
model.factors.W = 'w';
model.factors.S = reshape({'s1', 's2'}, 1, 1, 2);
model.factorizations.tensor.data  = T;
model.factorizations.tensor.lmlra = {'U','V','W','S'};

sdf_check(model, 'print');
```

In the expansion of `S` (click on `S`), we see that the parts `s1` and `s2` are indeed concatenated in the third mode (recall that matrix `s2` is a $2 \times 3 \times 1$ tensor):

```
Expansion of factor S (id = 4) with structure [1x1x2] (view):
  (1,1,1)  s1              [2x3x2]
  (1,1,2)  s2              [2x3]
```

8.5 Factorization and regularization types

Currently, the SDF language supports four different factorization types (CPD, BTD, LMLRA and the decomposition in multilinear rank-$(L_r, L_r, 1)$ terms) and three regularization types (L0, L1 and L2 regularization). The different models and factorization types are explained in their respective sections: Chapter 3, Chapter 6, Chapter 5 and Chapter 4. In the remaining part of this section we specify the format for the different factorization types:

- CPD: a (canonical) polyadic decomposition of a tensor $\mathcal{T}$ with factor matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, ... can be computed using

  ```
  model.factorizations.mycpd.data = T;
  model.factorizations.mycpd.cpd  = {'A', 'B', 'C', ...};
  ```

  In the case $\mathcal{T}$ is an incomplete tensor, specialized routines can be used if the `sdf_nls` solver is used. To activate these specialized routines use

  ```
  model.factorizations.mycpd.data = T;
  model.factorizations.mycpd.cpdi = {'A', 'B', 'C', ...};
  ```

- LMLRA: a low multilinear rank approximation of a tensor $\mathcal{T}$ with factor matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, ... and core tensor $\mathcal{S}$ can be computed using

```
model.factorizations.mylmlra.data   = T;
model.factorizations.mylmlra.lmlra  = {'A', 'B', 'C', ..., 'S'};
% or
model.factorizations.mylmlra.lmlra  = {{'A', 'B', 'C', ..., 'S'}};
```

- BTD: an R-term block term decomposition of a tensor $\mathcal{T}$ with factor matrices $\mathbf{A}_r$, $\mathbf{B}_r$, $\mathbf{C}_r$, ... and core tensors $\mathcal{S}_r$ for $r = 1, \ldots, R$ can be computed using

```
model.factorizations.mybtd.data = T;
model.factorizations.mybtd.btd  = {{'A1', 'B1', 'C1', ..., 'S1'}, ...,
                                   {'AR', 'BR', 'CR', ..., 'SR'}};
```

- LL1: a decomposition in multilinear rank-$(L_r, L_r, 1)$ terms of a third-order tensor $\mathcal{T}$ with factor matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and `L = [L1 L2 ... LR]` can be computed using

```
model.factorizations.myll1.data = T;
model.factorizations.myll1.ll1  = {'A', 'B', 'C'};
model.factorizations.myll1.L    = L;
```

  Note that the `L` field is obligatory in this case. Only the CPD format for a decomposition in multilinear rank-$(L_r, L_r, 1)$ terms is accepted, see Chapter 4.

In all cases above, both the `data` and the field that specifies the factorization type are obligatory.

Three regularization types, namely, L1 and L2 regularization (available since Tensorlab 2.0) and L0 regularization (available since Tensorlab 3.0), are currently available:

- L1 regularization: a term $\frac{1}{2}\|\text{vec}\,(\mathbf{A})\,\|_1$ or $\frac{1}{2}(\|\text{vec}\,(\mathbf{A})\,\|_1 + \|\text{vec}\,(\mathcal{B})\,\|_1 + \ldots)$ is added to factors, which can be matrices or tensors, using

```
model.factorizations.myreg.regL1 = 'A';
% or
model.factorizations.myreg.regL1 = {'A', 'B', ...};
```

  All factors mentioned in a single regularization term get the same weight (see Section 8.3.2). If different factors should have different weights, multiple regularization terms have to be added, e.g.,

```
% A and B have the same weight
model.factorizations.myreg1.regL1 = {'A', 'B'};
model.factorizations.myreg1.relweight = 1;
% C gets a different weight
model.factorizations.myreg2.regL1 = 'C';
model.factorizations.myreg2.relweight = 10;
```

  By default, a term is regularized towards zero. A term $\frac{1}{2}(\|\mathbf{0} - \text{vec}\,(\mathbf{A})\,\|_1 + \|\mathbf{1} - \text{vec}\,(\mathbf{B})\,\|_1)$ can be added using

```
model.factorizations.nonzeroreg.data = {zeros(size(A)), ones(size(B))};
model.factorizations.nonzeroreg.regL1 = {'A', 'B'};
```

- L2 regularization: a term $\frac{1}{2}\|\mathbf{A}\|_{\mathsf{F}}^2$ or $\frac{1}{2}(\|\mathbf{A}\|_{\mathsf{F}}^2 + \|\mathcal{B}\|_{\mathsf{F}}^2 + \ldots)$ is added to factors, which can be matrices or tensors, using

```
model.factorizations.myreg.regL2 = 'A';
% or
model.factorizations.myreg.regL2 = {'A', 'B', ...};
```

  The same remarks as for L1 regularization hold for L2 regularization as well.

- L0 regularization: a term $\|\text{vec}\,(\mathbf{A})\,\|_0$ or $(\|\text{vec}\,(\mathbf{A})\,\|_0 + \|\text{vec}\,(\mathcal{B})\,\|_0 + ...)$ is added to factors, which can be matrices or tensors, using

  ```
  model.factorizations.myreg.regL0 = 'A';
  % or
  model.factorizations.myreg.regL0 = {'A', 'B', ...};
  ```

  For computational reasons, a smoothed variant of the L0 norm is used: $\|\text{vec}\,(\mathbf{A})\,\|_0$ is approximated by $\sum_{i=1}^{I} 1 - \exp(-\frac{a_i^2}{\sigma^2})$ with I the number of entries in $\mathbf{A}$. The parameter σ can be a scalar constant or a function of the iteration `k` and the cell of regularized factors `x`. The value can be set using the `sigma` field:

  ```
  model.factorizations.myreg.regL0 = 'A';
  model.factorizations.myreg.sigma = 0.01; % (the default), or
  model.factorizations.myreg.sigma = @(k,x) 0.01*frob(x{1}); % x is cell of factors
  ```

  The same remarks as for L1 regularization hold for L0 regularization as well.

Similar to the other optimization-based decomposition algorithms, the SDF algorithms `sdf_minf`, `sdf_nls`, `ccpd_minf` and `ccpd_nls` accept both full, sparse, incomplete and structured tensors as data in the `data` field. Structured tensors can, for example, be given in the LMLRA format, the tensor train format, the Hankel format or the Loewner format. See Chapter 2 for more information. (Structured tensors are not to be confused with structure related to factors.) For structured tensors the maximal achievable numerical accuracy is only half the machine precision due to the way the structure is exploited by the algorithm. This is usually not a problem as noise, inexact models, regularization or chosen tolerances limit the attainable accuracy. If accuracy up to machine precision is required, the accuracy of the result `sol` of the decomposition of the structured tensor(s) can be improved using an additional refinement step. This refinement step uses `sol.variables` as initial guess for the same problem, but all structured tensors are replaced by their expanded form. For example, if `T` is a structured tensor, we replace

```
model.factorization.fac.data = T;
```

by

```
model.variables = sol.variables;
model.factorization.fac.data = ful(T);
```

Then we run the solver again using the updated model. It is often useful to limit the maximum number of iterations, e.g., by setting `MaxIter` to `3` or `5`. In case a high accuracy is required, Section 10.4 may be of interest.

8.6 Further reading

The remaining sections on structured data fusion handle other, more advanced concepts of the SDF language. More concrete examples and useful tricks can be found in Chapter 9. One section in the advanced chapter (Chapter 10) is particularly useful: Section 10.1 explains how the indexed notation can be used to simplify models. This also allows the auto-wrapping feature to be used to its fullest potential, and it makes it easier to create more dynamical models. (Auto-wrapping basically means that most braces `{}` are added automatically in unambiguous cases.)

CHAPTER

NINE

STRUCTURED DATA FUSION: EXAMPLES

In Chapter 8 the elements in the domain specific language (DSL) used for modeling structured data fusion problems are explained. The three key ingredients of an SDF model are the definition of (1) the variables, (2) the factors as transformed variables and (3) the data sets and which factors to use in their factorizations. Here, different examples illustrate how the SDF framework can be used to model different types of constraints, structures and couplings between datasets. These examples should quickly give a good idea of how the DSL can be used to create new models. Reading Chapter 8 remains highly recommended even if a more example based approach is favored by the user.

9.1 Example 1: nonnegative symmetric CPD

As a first example, the nonnegative symmetric CPD of an incomplete tensor $\mathcal{T}$ is computed. 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 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. It is recommended to start a new model with

```
model = struct;
```

This clears previous definitions of variables, factors and factorizations. Forgetting to clear a previous model is a common source of errors.

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 that are optimized, and is also used to initialize 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 define as a transformation of the variable `u`. More specifically, we require the factor to be nonnegative:

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

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 (named) factorization called `myfac` and it contains two fields. The first is `data` and specifies the tensor to be factorized. The second should be either `cpd`, `cpdi`, `btd`, `lmlra` or `ll1` depending on which model to use, and define the factors to be used in the decomposition (see Section 8.5).

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};
```

Since Tensorlab 3.0, the language parser is a lot more forgiving w.r.t. braces `{}`. Missing braces are added automatically if this can be done unambiguously. This is called auto-wrapping. After removing superfluous braces, the model becomes:

```
% Equivalent SDF model without using names for variables and factors.
model.variables = randn(I,R); % only if there is a single variable
model.factors = {1,@struct_nonneg};
model.factorizations.myfac.data = T;
model.factorizations.myfac.cpd = ones(1,3); % arrays are also allowed
```

The resulting model can be checked using the `sdf_check(model, 'print')` function:

```
Factor                  Size                    Comment
-------------------------------------------------------------------------
1                       [15x4]

Factorization           Type        Factors                 Comments
-------------------------------------------------------------------------
myfac (id = 1)          cpd         1,1,1
```

By clicking on the `1` in the Matlab Command Window, it can be seen that nonnegativity has indeed been imposed (click the `1` and `Factor` links to investigate this):

```
Expansion of factor 1 (view):
  (1,1)  1                [15x4]      struct_nonneg        Factor          [15x4]
```

By clicking `1` or `Factor`, its value becomes available as `ans`, which can be useful, for example, if we wish to plot an intermediate result. For example to check if all entries in the factor are positive, after clicking `Factor`, we can use `all(ans(:)) >= 0`.

The model can now be solved with one of the two families of algorithms for SDF problems: `sdf_minf` and `sdf_nls`. 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
```

If no nonnegativity is required, and hence only symmetry is used, the `ccpd_nls` and `ccpd_minf` methods can be used. For unstructured coupled and/or symmetric problems, these dedicated methods often behave better numerically:

```
model = struct;
model.variables = rand(I,R);
model.factors = 1;
model.factorizations.unconst.data = T;
model.factorizations.unconst.cpd  = [1 1 1];

Ures = ccpd_nls(model, 'Display', 10);
```

In case a tensor is symmetric in all modes, it can be useful to introduce an additional dimension to take scaling and signs of the rank-1 terms into account. This extra dimension is a vector of size $1 \times R$. For example a fourth-order fully symmetric real tensor having negative entries on the diagonal cannot be modeled without the extra dimension. With the extra dimension, the problem can be modeled as

```
model = struct;
model.variables.A     = randn(I, R);
model.variables.extra = randn(1, R);
model.factors.A = 'A';
model.factors.E = 'extra';
model.factorizations.symm.data = T;
model.factorizations.symm.cpd  = {'A', 'A', 'A', 'A', 'E'};
```

9.2 Example 2: structured coupled matrix factorization

Let $\mathbf{X}$ and $\mathbf{Y}$ be square matrices of dimension $N \times N$ which we wish to jointly factorize as $\mathbf{U} \cdot \mathbf{\Lambda}_X \cdot \mathbf{V}^{\mathsf{T}}$ and $\mathbf{U} \cdot \mathbf{\Lambda}_Y \cdot \mathbf{W}^{\mathsf{T}}$, respectively. The factor $\mathbf{U}$ is common to both $\mathbf{X}$ and $\mathbf{Y}$, and the extent to which the columns are shared is given by the absolute values of the diagonal matrices $\mathbf{\Lambda}_X$ and $\mathbf{\Lambda}_Y$. Furthermore, we will require the entries of $\mathbf{W}$ to lie in the interval (-3,5) and impose a Vandermonde structure on $\mathbf{V}$, so that

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

with $d = R - 1$ and R the number of terms. Note that $\mathbf{V}$ depends only on a so-called generator vector $\mathbf{v} = \begin{bmatrix} v_1 & \cdots & v_N \end{bmatrix}^{\mathsf{T}}$.

First, the matrices $\mathbf{X}$ and $\mathbf{Y}$ are created:

```
% 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;               % Entries 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 variables

```
% Define model variables.
model = struct;
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 $\mathbf{U}$ and $\mathbf{W}$, the generator vector $\mathbf{v}$ for the Vandermonde matrix $\mathbf{V}$ and the diagonals $\boldsymbol{\lambda}_X$ and $\boldsymbol{\lambda}_Y$. The model assumes that $\mathbf{X}$ and $\mathbf{Y}$ share at most $R = 4$ vectors in $\mathbf{U}$. 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]; % Bounds for d: first column is z^0 and last column z^(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};
model.factors.W  = {'w',sigmoid};
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 $\mathbf{X}$ and $\mathbf{Y}$ 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 $\mathbf{X} = \mathbf{U} \cdot \boldsymbol{\Lambda}_X \cdot \mathbf{V}^{\mathsf{T}}$ and $\mathbf{Y} = \mathbf{U} \cdot \boldsymbol{\Lambda}_Y \cdot \mathbf{W}^{\mathsf{T}}$ by associating the factor matrices $\boldsymbol{\lambda}_X$ and $\boldsymbol{\lambda}_Y$, which are row vectors, with the third dimension of $\mathbf{X}$ and $\mathbf{Y}$. 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   = 10;   % View convergence progress every 10 iterations.
options.TolFun    = 1e-9; % Stop earlier.
options.CGMaxIter = 500;  % Recommended if structure/coupling is imposed
[sol, output] = sdf_nls(model,options);
sol.variables
sol.factors
```

Although the algorithm converges, it usually does not converge to the solution we used to generate the data. Nevertheless a good relative error is obtained (see `output.relerr`). This may indicate that the solution is not unique (in contrast to the example in Section 9.1).

9.3 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 $\mathbf{q}$ that parameterizes a matrix with orthonormal columns $\mathbf{Q}$. The structure `struct_orth` can then be used to transform the variable $\mathbf{q}$ into the factor $\mathbf{Q}$. First, we construct the data for the problem:

```
% Generate CPD, where 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 $\mathbf{q}$ should be a vector of length $IR - 0.5R(R-1)$. To transform $\mathbf{q}$ into a matrix with orthonormal columns, we need to pass the size of $\mathbf{Q}$ 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:

```
% Clear previous model
model = struct;

% 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 that transforms 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'};

sdf_check(model, 'print');
```

```
 Factor                Size                 Comment
 ------------------------------------------------------------------------
 Q                     [15x4]
 B                     [15x4]
 C                     [15x4]

 Factorization         Type       Factors              Comments
 ------------------------------------------------------------------------
 myfac (id = 1)        cpd        Q,B,C
```

The factor `Q` is indeed a 15×4 matrix. By clicking `Q` in the Matlab Command Window, and then `view`, we can indeed verify the orthogonality of the factors, as it becomes available as `ans`:

```
Q = ans;
Q'*Q
```

```
ans =
    1.0000    0.0000   -0.0000    0.0000
    0.0000    1.0000   -0.0000    0.0000
   -0.0000   -0.0000    1.0000   -0.0000
    0.0000    0.0000   -0.0000    1.0000
```

The model can now be computed, e.g., using the `sdf_nls` solver:

```
% Solve the SDF problem.
sol = sdf_nls(model, 'Display', 10);

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

9.4 Example 4: decomposition in multilinear rank-$(L_r, L_r, 1)$ terms

In this example, the decomposition of a tensor $\mathcal{T}$ into multilinear rank-$(L_r, L_r, 1)$ terms is illustrated. This decomposition can be seen as a special case of the BTD, i.e., having an identity matrix as core, or as a special case of the CPD, i.e., some rank-1 terms have an identical vector in the third mode. More concretely, we want to compute a multilinear rank-$(L_r, L_r, 1)$ BTD [25, 26, 27] of a tensor $\mathcal{T}$ in R terms, i.e.,

$$\mathcal{T} \approx \sum_{r=1}^{R} (\mathbf{A}_r \mathbf{B}_r^{\mathsf{T}}) \otimes \mathbf{c}_r$$

where $\mathbf{c}_r$ are vectors and $\mathbf{A}_r$ and $\mathbf{B}_r$ have L_r columns. In the unconstrained and uncoupled case, the decomposition into rank-$(L_r, L_r, 1)$ terms can be computed using the specialized `ll1` routines (see Chapter 4). In this example, three ways to compute this decomposition are shown:

- using the `ll1` factorization type,

- using the BTD formulation, and
- using the CPD formulation by imposing that some vectors are equal using `struct_LL1`.

The `ll1` factorization type has been added in Tensorlab 3.0, and is the recommended way to compute a decomposition in multilinear rank-$(L_r, L_r, 1)$ terms in the constrained and/or coupled case. When no constraints or coupling are needed, the `ll1` method can be used.

9.4.1 Example 4a: using the `ll1` factorization type

We use a decomposition into multilinear rank-$(L_r, L_r, 1)$ terms with $L = [3, 2]$ in this example. In the CPD formulation of this decomposition, the different factors $\mathbf{A}_1$ and $\mathbf{A}_2$ are concatenated along the second mode, i.e., in Matlab code we have `A = [A1 A2]`. The same holds for $\mathbf{B}$ and $\mathbf{C}$. Hence, factors `A` and `B` have `sum(L)` columns, while `C` has `length(L)` columns (for more information see Chapter 4). To generate the data the `ll1_rnd` routine can be used:

```
size_tens = [10 11 12];
L  = [3 2];
U0 = ll1_rnd(size_tens, L);
T  = ll1gen(U0);
```

For the unconstrained decomposition into multilinear rank-$(L_r, L_r, 1)$ terms, the `ll1`, `ll1_minf` and `ll1_nls` routines can be used. Here we illustrate the `ll1` factorization type in the SDF framework, as this can be useful if some factors have structure, or are shared between factorizations. The decomposition can then be modeled as follows:

```
L = [3 2];
model = struct;
model.variables.A = randn(size_tens(1), sum(L));
model.variables.B = randn(size_tens(2), sum(L));
model.variables.C = randn(size_tens(3), length(L));
model.factors.A = 'A';
model.factors.B = 'B';
model.factors.C = 'C';
model.factorizations.myll1.data = T;
model.factorizations.myll1.ll1 = {'A', 'B', 'C'};
model.factorizations.myll1.L   = L;
```

The `ll1` factorization type only accepts the CPD format for the decomposition into multilinear rank-$(L_r, L_r, 1)$ terms. The `L` field defines the different terms. The `model` can again be solved using the `sdf_nls` and `sdf_minf` solvers.

9.4.2 Example 4b: using the `btd` factorization type

The decomposition into rank-$(L_r, L_r, 1)$ terms is a special case of a block term decomposition (BTD), in which the core tensors $\mathcal{S}_r$ are matrices of size $L_r \times L_r \times 1$. The decomposition then takes the form:

$$\mathcal{T} \approx \sum_{r=1}^{R} \mathcal{S}_r \cdot_1 \mathbf{A}_r \cdot_2 \mathbf{B}_r \cdot_3 \mathbf{c}_r$$

By default, `ll1_rnd` returns a decomposition into multilinear rank-$(L_r, L_r, 1)$ terms in the BTD format. By setting the `OutputFormat` option to `'cpd'` the CPD format is obtained. The `btd` factorization type can be used as follows:

```
% Define rank-(Lr,Lr,1) BTD model using the BTD.
model = struct;
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 indexed factors. The model can be shortened even further by making use of indexed definitions and references (see Chapter 10):

```
model = struct;

U0 = ll1_rnd(size_tens, L);          % Initial guess in BTD format
% Convert {{U{1}{1},U{1}{2},...}, {U{2}{1},...},...}
% to {U{1}{1},U{1}{2},...,U{2}{1},...}
U0 = cat(2, U0{:});
model.variables = U0;
model.factors = 1:8;
model.factorizations.mybtd.data = T;
model.factorizations.mybtd.btd = {1:4, 5:8};
```

Note that in SDF the BTD is specified in the same format as for other BTD algorithms, e.g., `btd_nls` and `btd_rnd`, i.e., a nested cell of cells is used, but now references to factors are used instead of factor matrices and tensors. Thanks to auto-wrapping, many of the braces can be omitted, though. The BTD can then be computed by

```
% Solve the SDF problem.
options.Display = 10; % View convergence progress every 10 iterations.
sol = sdf_nls(model,options);
sol.variables
sol.factors
```

9.4.3 Example 4c: using the `cpd` factorization type

The third way to compute a decomposition into multilinear rank-$(L_r, L_r, 1)$ terms uses a CPD. In our example, the decomposition can be written as:

$$\mathcal{T} \approx \sum_{j=1}^{L_1} \mathbf{A}_1(:,j) \otimes \mathbf{B}_1(:,j) \otimes \mathbf{c}_1 + \sum_{j=1}^{L_2} \mathbf{A}_2(:,j) \otimes \mathbf{B}_2(:,j) \otimes \mathbf{c}_2$$

Here, the third CPD factor matrix `C` looks like `[repmat(c1,1,L1) repmat(c2,1,L2)]`, while the first two are `[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 = struct;
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', 'c1', 'c2', 'c2'};
model.factorizations.mycpd.data = T;
model.factorizations.mycpd.cpd = {'A','B','C'};
```

The line `model.factors.A = {'A1', 'A2'}` states that the factor `A` is a matrix consisting of two subfactors that are concatenated horizontally. Internally, the SDF algorithm first builds the subfactors, which are in this case simply references to the variables `A1` and `A2`, and then calls `cell2mat` on the cell array of subfactors to create the factor `A`. Besides on 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).

The explicit modeling in the previous code block can be avoided using the transformation `struct_LL1`:

```
L = [L1 L2];
LL1 = @(z,task)struct_LL1(z,task,L);

model = struct;
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.myll1.data = T;
model.factorizations.myll1.cpd = {'A','B','C'};
```

9.5 Example 5: constants

To indicate that a factor or subfactor is constant, it suffices to use that constant array in the factor instead of a reference to a variable.

9.5.1 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 C = U{3} is known.
model = struct;
model.variables.a = randn(size(U{1}));
model.variables.b = randn(size(U{1}));
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
```

When running the `sdf_check(model, 'print')` function, `C` is indeed recognized as a constant factor:

```
C                    [15x4]                constant
```

9.5.2 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 fixed 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 last columns of C are known
model = struct;
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
```

When running the `sdf_check(model, 'print')` function, `C` is indeed recognized as a partially constant factor:

```
C                    [15x4]              Partially constant
```

9.6 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 $\mathbf{A}$ 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 an unconstrained matrix `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 = struct;
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.
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'};

sdf_check(model, 'print');
```

The different steps used to construct factor matrix $\mathbf{A}$ can be investigated by clicking `A` in the Matlab Command Window:

```
Expansion of factor A (id = 1) with structure [2x1] (view):
  (1,1)  atop            [7x1]      struct_nonneg        temp            [7x1]
         temp            [7x1]      struct_toeplitz      Factor          [4x4]
  (2,1)  abtm            [11x4]
```

```
% Solve the SDF problem.
options.Display = 10; % View convergence progress every 10 iterations.
sol = sdf_nls(model,options);
```

9.7 Example 7: regularization

In addition to the CPD and BTD models, SDF also foresees three models that represent L0, 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';
```

in the previous example, the terms $\frac{1}{2}\| \left[\text{vec}\,(\mathbf{A})^{\mathsf{T}} \quad \text{vec}\,(\mathbf{B})^{\mathsf{T}}\right]^{\mathsf{T}} - \mathbf{0}\|_2^2 = \frac{1}{2}(\|\mathbf{A}\|_{\mathsf{F}}^2 + \|\mathbf{A}\|_{\mathsf{F}}^2)$ and $\frac{1}{2}\|\text{vec}\,(\mathbf{C}) - \mathbf{1}\|_1$ are added to the objective function. The `data` field defaults to all zeros, if omitted. In the case of L0 regularization, a smoothed variant is used, i.e., $\|\mathbf{x}\|_0$ is computed using $\sum_i 1 - \exp(-\frac{x_i^2}{\sigma^2})$. The parameter σ can be set using the `sigma` field and can be either a scalar constant, or a function in the iteration number `k` and the factors `x`. For example:

```
% Use L2-regularization on the factors A.
model.factorizations.myreg0.regL0 = 'A';
model.factorizations.myreg0.sigma = 0.01; % (the default) or
model.factorizations.myreg0.sigma = @(k,x) 0.01*frob(x{1}); % x is a cell of factors
```

When regularization is used, the importance or weight of the regularization is often an important hyperparameter. Consider the following problem:

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C}} \frac{\lambda_1}{2} ||\mathcal{T} - \mathcal{M}_{\text{CPD}}(\mathbf{A},\mathbf{B},\mathbf{C})||_{\mathsf{F}}^2 + \frac{\lambda_2}{2} ||\mathbf{A}||_{\mathsf{F}}^2 + \frac{\lambda_3}{2}(||\text{vec}\,(\mathbf{B})||_1 + ||\text{vec}\,(\mathbf{C})||_1).$$

This can be modeled as follows:

```
model = struct;
% definition of variables and factors omitted
model.factorizations.tensor.data   = T;
model.factorizations.tensor.cpd    = {'A', 'B', 'C'};
model.factorizations.tensor.weight = lambda1;
model.factorizations.regA.regL2    = 'A';
model.factorizations.regA.weight   = lambda2;
model.factorizations.regBC.regL1   = {'B', 'C'};
model.factorizations.regBC.weight  = lambda3;
```

The example above makes use of absolute weights. It is also possible to work with relative weights, by using `relweight` instead of `weight`. For example:

```
model = struct;
% definition of variables and factors omitted
model.factorizations.tensor.data      = T;
model.factorizations.tensor.cpd       = {'A', 'B', 'C'};
model.factorizations.tensor.relweight = 100;
model.factorizations.regA.regL2       = 'A';
model.factorizations.regA.relweight   = 1;
model.factorizations.regBC.regL1      = {'B', 'C'};
model.factorizations.regBC.relweight  = 2;
```

This example states that the tensor decomposition is 100 times more important than the L2 regularization term, and 50 times more important than the L1 regularization term, regardless of how many entries the different terms involve. Similarly, the L1 regularization term is more important than the L2 regularization term. The precise relation between the relative weights and the weights λ_i is explained in Chapter 8.

9.8 Example 8: compression based initialization

The `cpd` and `ll1` functions implement a full initialization strategy in order to reduce the computation time. A similar `sdf` method does not yet exist. Nevertheless, we show how compression can be used in SDF to reduce the computational cost. Consider the following nonnegative CPD of a rank-5 tensor $\mathcal{T}$ with dimensions $150 \times 160 \times 170$:

```
R = 5;
model = struct;
model.variables.a = rand(150,R);
model.variables.b = rand(160,R);
model.variables.c = rand(170,R);
model.factors.A   = {'a', @struct_nonneg};
model.factors.B   = {'b', @struct_nonneg};
model.factors.C   = {'c', @struct_nonneg};
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};

sol = sdf_nls(model);
```

In order to reduce the computational cost, the tensor $\mathcal{T}$ can be compressed, e.g., using the `mlsvd_rsi` method:

```
[U,S] = mlsvd_rsi(T, [R R R]);
```

In the unconstrained CPD case, the factor matrices `Ucomp{1}`, `Ucomp{2}`, and `Ucomp{3}` in the CPD of the compressed tensor $\mathcal{S}$ can be computed and expanded to the factors of the full tensor $\mathcal{T}$. This can be done by multiplying the factor matrices `Ucomp{1}`, `Ucomp{2}`, and `Ucomp{3}` with the compression matrices `U{1}`, `U{2}`, and `U{3}`, respectively.

```
Ucomp = cpd(S, R);
Uinit = cellfun(@(u,v) u*v, U, Ucomp, 'UniformOutput', false);
Ures  = cpd_nls(T, Uinit); % refinement
```

The last refinement step improves the accuracy in case information was lost in the compression.

In the constrained case, this strategy can no longer be used, as the compressed tensor $\mathcal{S}$ is not nonnegative. Tensorlab 3.0 foresees a different strategy, which can be followed in the constrained case as well. The idea is

to use the results of the compression for a compact representation as a structured tensor, and use the latter to efficiently obtain a first estimate of the desired factors.

```
R = 5;
model = struct;
model.variables.a = randn(150,R);
model.variables.b = randn(160,R);
model.variables.c = randn(170,R);
model.factors.A   = {'a', @struct_nonneg};
model.factors.B   = {'b', @struct_nonneg};
model.factors.C   = {'c', @struct_nonneg};
model.factorizations.tensor.data = {U,S};
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};

sol = sdf_nls(model);
```

The results in `sol` are next used in a refinement step:

```
R = 5;
model = struct;
model.variables.a = sol.variables.a; % initialization for refinement step
model.variables.b = sol.variables.b;
model.variables.c = sol.variables.c;
model.factors.A   = {'a', @struct_nonneg};
model.factors.B   = {'b', @struct_nonneg};
model.factors.C   = {'c', @struct_nonneg};
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};

sol = sdf_nls(model);
```

It can be useful to limit the number of iterations in the refinement step by setting the `MaxIter` option to avoid unneeded work:

```
sol = sdf_nls(model, 'MaxIter', 10);
```

9.9 Example 9: advanced coupling

The previous examples only considered coupled datasets in which the coupling took the form of a shared factor matrix. In this section, we consider joint tensor-matrix and joint tensor-tensor decompositions with a more complex relation between the datasets. Consider the following three datasets: a tensor $\mathcal{T} = \mathcal{M}_{\text{CPD}}(\mathbf{A}, \mathbf{B}, \mathbf{C})$, a tensor $\mathcal{S} = \mathcal{M}_{\text{CPD}}(\mathbf{D}, \mathbf{E}, \mathbf{F})$ and a matrix $\mathbf{M} = \mathcal{M}_{\text{CPD}}(\mathbf{D}, \mathbf{E}) = \mathbf{D}\mathbf{E}^{\mathsf{T}}$. Exact coupling, e.g., $\mathbf{A} = \mathbf{D}$, has already been discussed in Section 9.2. Here, the following relations are considered:

- $\mathcal{T}$ and $\mathbf{M}$ partially share a factor matrix, i.e., $\mathbf{A} = \mathbf{D}(:, 1 : R_1)$, in which $\mathcal{T}$ has rank R_1 and $\mathbf{M}$ has rank $R_2 > R_1$.
- $\mathcal{T}$ and $\mathcal{S}$ are approximately coupled using $\mathbf{D} = \mathbf{A} + \mathbf{N}$, where $\mathbf{N}$ is small compared to $\mathbf{A}$.
- $\mathcal{T}$ and $\mathbf{M}$ are coupled using $\mathbf{D} = \mathbf{X}\mathbf{A}\mathbf{Y}$, i.e., $\mathbf{D}$ is a linear transformation of $\mathbf{A}$ with known matrices $\mathbf{X}$ and $\mathbf{Y}$.
- $\mathcal{T}$ and $\mathcal{S}$ share an underlying variable z, i.e., $\mathbf{A} = f(z)$ and $\mathbf{D} = g(z)$.

9.9.1 Example 9a: partial coupling

In this example, we investigate the case in which only a part of a factor matrix is shared. Consider a tensor $\mathcal{T}$ and a matrix $\mathbf{M}$. The first $R_1 = \text{rank}(\mathcal{T})$ columns of $\mathbf{D}$ are shared. Let $\mathbf{D} = [\mathbf{D}_1, \mathbf{D}_2]$ then $\mathbf{D}_1 = \mathbf{A}$. The two datasets are modeled as follows:

```
% Tensor T
U1 = cpd_rnd([10, 11, 12], 3); % factors A, B, C
T = cpdgen(U1);
% matrix M
U2 = cpd_rnd([10, 13], 5);      % factors D, E
U2{1}(:,1:3) = U1{1};           % D(:,1:3) = A
M = cpdgen(U2);
```

In the SDF language, the partial coupling can be expressed using subfactors, i.e., factor $\mathbf{D}$ is the horizontal concatenation of variables `a` and `d2`: `D = {'a', 'd2'}`. The full model is given by

```
model = struct;
model.variables.a  = randn(10,3);
model.variables.b  = randn(11,3);
model.variables.c  = randn(12,3);
model.variables.d2 = randn(10,2); % remaining two vectors
model.variables.e  = randn(13,5);
model.factors.A = 'a';
model.factors.B = 'b';
model.factors.C = 'c';
model.factors.D = {'a', 'd2'};
model.factors.E = 'e';
model.factorizations.T.data = T;
model.factorizations.T.cpd  = {'A', 'B', 'C'};
model.factorizations.M.data = M;
model.factorizations.M.cpd  = {'D', 'E'};

[sol,output] = sdf_nls(model, 'Display', 10, 'MaxIter', 200, 'CGMaxIter', 500);
```

Some extra options are provided to the `sdf_nls` solver. The `Display` option prints an update every 10 iterations. The `MaxIter` option decreases the maximum number of iterations to limit the computation time, if no good solution is found. Finally, the `CGMaxIter` option increases the maximum number of CG iterations from 15 (the default) to 500. When structure is imposed on some factors or coupled datasets are used, it is recommended to increase `CGMaxIter` as it generally improves the convergence behavior at the cost of more expensive iterations. Often the reduction in the number of iterations makes up for the increased cost per iteration.

The algorithm may converge to a local optimum, depending on the initialization. It is therefore recommended to perform multiple runs with different initial values. Recall that the lines `model.variables.a = randn(10,3)` etc. provide the initial guess, hence simply executing the `sdf_nls(model)` multiple times does not change the result. The variables have to be reinitialized by reevaluating the lines `model.variables.a`, `model.variables.b`, etc. In the example above, a good solution can be be found using (multiple) random initializations. Other problems may require better initialization strategies in order to get a good solution. One can, for example, compute the decompositions of the uncoupled datasets, match the different terms and then use this result to initialize the coupled problem.

9.9.2 Example 9b: approximate coupling

The coupling between two or more tensors may not be exact. For example, the shared factors can be approximately the same (see, e.g., [38]). In this example, two tensors $\mathcal{T}$ and $\mathcal{S}$ are coupled in the first mode. To model the inexact coupling, an additional 'noise' term $\mathbf{N}$ is used: $\mathbf{D} = \mathbf{A} + \mathbf{N}$. If this noise term is large compared to $\mathbf{A}$, there is almost no coupling. If the noise term is small compared to $\mathbf{A}$, the coupling is stronger. The two tensors are constructed as follows:

```
% first tensor T
U1 = cpd_rnd([10, 11, 12], 3);     % factors A, B, C
T  = cpdgen(U1);

% second tensor S
U2 = cpd_rnd([10, 13, 14], 3);     % factors D, E, F
U2{1} = U1{1} + randn(10,3)*1e-3; % D = A + N;
S = cpdgen(U2);
```

To model the approximate coupling, two transformations are needed: `struct_select` and `struct_plus`. The former can be used to select one entry from a cell variable. The latter can be used to sum all entries in a cell variable. The cell variable `an` contains the initial guesses for $\mathbf{A}$ and $\mathbf{N}$: `an = {randn(10,3), randn(10,3)*1e-2}`. Factor `A` simply selects the first entry of `an`, while factor `D` sums both entries. The noise term $\mathbf{N}$ should be small, otherwise both tensors are uncoupled. Therefore, L2 regularization is applied to $\mathbf{N}$.

```
model = struct;
model.variables.an = {randn(10,3), randn(10,3)*1e-2};          % {A, N}
model.variables.b = randn(11,3);
model.variables.c = randn(12,3);
model.variables.e = randn(13,3);
model.variables.f = randn(14,3);
model.factors.A = {'an', @(z,task) struct_select(z,task,1)};  % A
model.factors.B = 'b';
model.factors.C = 'c';
model.factors.D = {'an', @struct_plus};                        % A + N
model.factors.E = 'e';
model.factors.F = 'f';
model.factors.N = {'an', @(z,task) struct_select(z,task,2)};  % N
model.factorizations.T.data = T;
model.factorizations.T.cpd  = {'A', 'B', 'C'};
model.factorizations.S.data = S;
model.factorizations.S.cpd  = {'D', 'E', 'F'};
model.factorizations.N.regL2  = {'N'};

sdf_check(model, 'print');
```

The overview of the model indeed shows that the right factors have been selected. (Click the different factors in the Matlab Command Window.) The model can be computed by:

```
[sol,output] = sdf_nls(model, 'Display', 10, 'CGMaxIter', 100, 'MaxIter', 200);
```

The output `output.relerr` gives the relative error between the data and the computed decomposition for each factorization. As can be seen, the tensors are approximated quite well:

```
relerr: [1.6933e-06 1.4664e-06 Inf]
```

The `Inf` entry indicates that the norm of the data for the regularization term `N` is zero. (Recall that the `data` field is zero if no data is specified.) The absolute error `output.abserr` is more useful for the regularization term.

The average difference between the computed factors $\mathbf{A}$ and $\mathbf{D}$ is given by `mean(abs(sol.factors.A(:) - sol.factors.D(:)))`:

```
  0.0016
```

which is indeed of the expected order of magnitude (recall that N is generated as `randn(10,3)*1e-3`).

9.9.3 Example 9c: linearly transformed coupling

In this example, two datasets are coupled using the relation $\mathbf{D} = \mathbf{X}\mathbf{A}\mathbf{Y}$, i.e., $\mathbf{D}$ is a linear transformation of $\mathbf{A}$ with two known matrices $\mathbf{X}$ and $\mathbf{Y}$. This relation is used to jointly decompose a tensor $\mathcal{T}$ and a matrix $\mathbf{M}$, which are generated as follows:

```
% tensor
U1 = cpd_rnd([10, 11, 12], 4); % factors A, B, C
T = cpdgen(U1);

% Matrix
U2 = cpd_rnd([10, 13], 4);      % factors D, E
X = orth(randn(20,10));         % orthogonality is used pinv(X) = X'
Y = randn(4,4);                 % random transformation
U2{1} = X*U1{1}*Y;              % D = X*A*Y
M = cpdgen(U2);
```

To model the linear transformation of the factor matrix $\mathbf{A}$, the `struct_matvec` transformation is used. This transformation can be used to model left and right multiplications with matrices. Indexed variables are used to shorten the code.

```
model = struct;
model.variables = cpd_rnd([10 11 12 13], 4);
model.factors.A = 1;
model.factors.B = 2;
model.factors.C = 3;
model.factors.D = {1, @(z,task) struct_matvec(z, task, X, Y)};
model.factors.E = 4;
model.factorizations.T.data = T;
model.factorizations.T.cpd  = {'A', 'B', 'C'};
model.factorizations.M.data = M;
model.factorizations.M.cpd  = {'D', 'E'};

[sol,output] = sdf_nls(model, 'Display', 10, 'CGMaxIter', 500, ...
                       'TolFun', eps^2, 'TolX', eps, 'MaxIter', 250);
```

The `output.relerr` field shows that for both datasets the relative error is very small (multiple initializations may be needed):

```
output.relerr =
   1.0e-14 *
    0.0653    0.1022
```

The same holds for `cpderr(U1, {sol.factors.A, sol.factors.B, sol.factors.C})`. By construction, we have `all(all(X*sol.factors.A*Y - sol.factors.D == 0))`. Note that `cpderr(U2, {sol.factors.D, sol.factors.E})` is not small in general, as a matrix decomposition is not unique.

Note that the left multiplication with $\mathbf{X}$ can be removed, by multiplying the data matrix $\mathbf{M}$ on the left with the (pseudo)inverse of $\mathbf{X}$. As $\mathbf{X}$ has orthonormal columns, $\mathbf{X}^{\mathsf{H}}$ can be used. To do the left multiplication, the `tmprod` command is used, as it works both for matrices and higher-order tensors:

```
% Only relevant lines shown:
model.factors.D = {1, @(z,task) struct_matvec(z, task, [], Y)};
model.factorizations.M.data = tmprod(M, X', 1);

[sol,output] = sdf_nls(model, 'Display', 10, 'CGMaxIter', 500, ...
                       'TolFun', eps^2, 'TolX', eps, 'MaxIter', 250);
```

Note that in the first line `X` is replaced by `[]` as the multiplication from the left is no longer necessary.

9.9.4 Example 9d: same underlying variable

Datasets may share an underlying variable that is transformed using different transformations. In this example, factor matrices $\mathbf{A}$ and $\mathbf{D}$ are constructed as a transposed Vandermonde matrix (see Section 9.2) and a squared transposed Vandermonde matrix. The shared underlying variables z are the poles or generators of the Vandermonde matrices. The other factor matrices for tensors $\mathcal{T}$ and $\mathcal{S}$ are generated using `cpd_rnd`.

```
% Shorthand function for transposed Vandermonde matrices
vander = @(z,range) struct_vander(z,[],range).';
% Poles = shared variable
poles = exp(2*pi*1i*randn(1,4));
% First tensor T
U1 = cpd_rnd([10, 11, 12], 4); % factors A, B, C
U1{1} = vander(poles, [1 10]);
T = cpdgen(U1);
% Second tensor S
U2 = cpd_rnd([7, 13, 14], 4); % factors D, E, F
U2{1} = vander(poles, [1 7]).^2;
S = cpdgen(U2);
```

The coupling can easily be expressed by generating two factor matrices $\mathbf{A}$ and $\mathbf{D}$ that use the same variable `poles`, but through different transformations:

```
% Predefine transformations to improve readability
vander1 = @(z,task) struct_vander(z, task, [1 10]);
vander2 = @(z,task) struct_vander(z, task, [1 7]);
transp  = @struct_transpose;
square  = @(z,task) struct_power(z, task, 2);
```

```
model = struct;
model.variables.poles = exp(2*pi*1i*rand(1,4));
model.variables.b = randn(11, 4);
model.variables.c = randn(12, 4);
model.variables.e = randn(13, 4);
model.variables.f = randn(14, 4);
model.factors.A = {'poles', vander1, transp};
model.factors.B = 'b';
model.factors.C = 'c';
model.factors.D = {'poles', vander2, transp, square};
model.factors.E = 'e';
model.factors.F = 'f';
model.factorizations.T.data = T;
model.factorizations.T.cpd  = {'A', 'B', 'C'};
model.factorizations.S.data = S;
model.factorizations.S.cpd  = {'D', 'E', 'F'};

sdf_check(model, 'print');
```

The overview created by `sdf_check` shows that the 1×4 vector of poles is transformed into the correct matrices. For example, by clicking factor `D` the different steps can be investigated:

```
Expansion of factor D (id = 4) (view):
  (1,1)  poles           [1x4]      struct_vander        temp            [4x7]
         temp            [4x7]      struct_transpose     temp            [7x4]
         temp            [7x4]      struct_power         Factor          [7x4]
```

Finally, the model can be computed using the `sdf_nls` solver:

```
[sol,output] = sdf_nls(model, 'Display', 10, 'CGMaxIter', 150, ...
                       'TolFun', eps^2, 'TolX', eps, 'MaxIter', 350);
```

Multiple initializations may be required to find an accurate solution.

CHAPTER

TEN

STRUCTURED DATA FUSION: ADVANCED CONCEPTS

A few more advanced modeling concepts are explained in this section. These concepts allow for more dynamical and/or more compact models. Three concepts are discussed: indexed references, the `transform` field which applies transformations to multiple variables in a single line, and the implicit factors options which tries to automatically create missing factors from variables. The details of these concepts can be found in the full language specification (Chapter 12).

10.1 Using indexed notations

In the basic introduction to SDF (Chapter 8) we always used named variables, factors and factorizations, e.g., `model.variables.a`, and named references to variables and factors, e.g., `model.factors.A = {'a'}`. While this practice makes the model easy to read, it requires many keystrokes and it complicates the adaption of models afterwards, e.g., if a third-order tensor model is expanded into a fourth-order tensor model.

Internally, all structs are converted to cells, which means that instead of names indices are used. This cell format is available to the user. For example:

```
model = struct;
model.variables.a = a;
model.variables.b = b;
model.factors.A = 'a';
model.factors.B = 'b';
model.factorizations.mat.data = T;
model.factorizations.mat.cpd = {'A','B'};
```

can be rewritten using indexed references as

```
model = struct;
model.variables.a = a;
model.variables.b = b;
model.factors.A = 1; % refers to variable a
model.factors.B = 2; % refers to variable b
model.factorizations.mat.data = T;
model.factorizations.mat.cpd = 1:2; % refers to factors A (1) and B (2)
```

We see that both the names and indices of variables and factors can be used to refer to them. The index of the variable or factor is determined by the order in which the different variables or factors are defined the first time:

```
model = struct;
model.variables.a = a;  % has index 1
model.variables.c = c;  % has index 2
```

```
model.variables.a = a2; % still has index 1 (defined a first time before)
model.variables.b = b;  % has index 3
```

The names of the variables, factors and factorizations can even be omitted. In this situation we use indexed variables, factors or factorizations instead.

```
model = struct;
model.variables{1} = a;
model.variables{2} = b;
model.factors{1} = 1;
model.factors{2} = 2;
model.factorizations{1}.data = T;
model.factorizations{1}.cpd = 1:2;
```

Named and indexed referencing can even be mixed, e.g.,

```
model = struct;
model.variables{1} = a;
model.variables{2} = b;
model.factors.A = 1;
model.factors.B = 2;
model.factorizations{1}.data = T; % a third-order tensor
model.factorizations{1}.cpd = {1, 'A', 'B'}; % == {1,1,2} or {'A','A','B'}
```

Named references can only be used if the variable or factor being referred to has been named. Within the same field (`variables` , `factors` or `factorizations`) all definitions are either named or indexed. The following block throws a MATLAB error:

```
model = struct;
model.variables.a = a;
model.variables{2} = b;
```

while the following block throws an error or gives a warning depending on the MATLAB version:

```
model = struct;
model.variables{1} = a;
model.variables.b  = b;
```

If no error is thrown, the model above is the same as

```
model = struct;
model.variables.b = b;
```

hence the definition of variable 1 is ignored.

The fact that all fields can be cells, can be used to shorten the code. Consider a regularized CPD of a third-order tensor $\mathcal{T}$ of rank 5. In the basic SDF syntax, we have

```
Uinit = cpd_rnd(size(T), 5);
model = struct;
model.variables.a = Uinit{1};
model.variables.b = Uinit{2};
model.variables.c = Uinit{3};
model.factors.A = 'a';
model.factors.B = 'b';
model.factors.C = 'c';
model.factorizations.tensor.data = T;
```

```
model.factorizations.tensor.cpd = {'A', 'B', 'C'};
model.factorizations.regul.regL2 = {'A', 'B', 'C'};
```

By exploiting the fact that all fields can be cells, the model above can be rewritten as:

```
model = struct;
model.variables = cpd_rnd(getsize(T), 5);
model.factors = 1:getorder(T);
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd = 1:getorder(T);
model.factorizations.regul.regL2 = 1:getorder(T);
```

Note that we used `getorder(T)` instead of `3`. If $\mathcal{T}$ is a fourth-order tensor, the code block above still works without the need to alter the code. Instead of `getorder(T)` the Matlab command `ndims(T)` can also be used. The advantage of the former is that it works on full, sparse, incomplete and structured tensors, while the latter only works for full tensors.

For the `factorizations` field, there exists an additional shortcut. Apart from a struct or cell, the `factorizations` field can also be a struct array. As an example, consider a coupled CPD of two third-order tensors $\mathcal{S}$ and $\mathcal{T}$ which are coupled in the first mode. The usual way of constructing this model is (only relevant fields are shown):

```
model.factorizations{1}.data = S;
model.factorizations{1}.cpd  = 1:3;
model.factorizations{2}.data = T;
model.factorizations{2}.cpd  = [1 4 5];
```

Using a struct array, the following equivalent specification can be used:

```
datasets = {S, T};
coupling = {[1 2 3], [1 4 5]};
model.factorizations = struct('data', datasets, 'cpd', coupling);
```

This syntax can be useful for coupled factorizations that involve many datasets.

10.2 The `transform` field

In the case of structured factorizations, it often occurs that some factors have the same or a highly similar structure, i.e., the same transformation is used for multiple factors. This transformation can have the same or different parameters for each factor. Consider the following nonnegative CPD of a third-order rank-5 tensor $\mathcal{T}$:

```
model = struct;
model.variables.a = rand(size(T, 1), 5);
model.variables.b = rand(size(T, 2), 5);
model.variables.c = rand(size(T, 3), 5);
model.factors.A = {'a', @struct_nonneg};
model.factors.B = {'b', @struct_nonneg};
model.factors.C = {'c', @struct_nonneg};
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};
```

This model can be written more compactly using a `transform` field:

```
model = struct;
model.variables.a = rand(size(T, 1), 5);
model.variables.b = rand(size(T, 2), 5);
model.variables.c = rand(size(T, 3), 5);
model.transform.nonneg = {{'a', 'b', 'c'}, {'A', 'B', 'C'}, @struct_nonneg};
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = {'A', 'B', 'C'};
```

The `nonneg` transform transforms the variables `{'a', 'b', 'c'}` into the factors `{'A', 'B', 'C'}` using the transformation `@struct_nonneg`. As can be seen using `sdf_check(model, 'print')`, three factors `A`, `B` and `C` are created. The factor names are often not needed as such: if they are omitted, the variable names are also used as factor names. Note that if `{'A', 'B', 'C'}` is omitted, the newly created factors have names `a`, `b` and `c`, and lowercase names should also be used in the last line of the previous model. This can again be checked using `sdf_check(model, 'print')`. Using indexing instead of explicit naming, the model can be simplified further and made more dynamic:

```
model = struct;
model.variables = cpd_rnd(T, 5, 'Real', @rand);
model.transform{1} = {1:getorder(T), @struct_nonneg};
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = 1:getorder(T);
```

Similar to variables, factors and factorizations, multiple transforms can be defined by using the struct (`model.transform.t1 = ...`) or the cell (`model.transform{1} = ..`) notation. Also, the `factors` and the `transform` field can be used together in a model, as shown in the following example in which the second and third factor matrix have polynomials as columns, while the first factor matrix is unconstrained. Without `transform` the code is:

```
model = struct;
model.variables.A     = randn(size(T,1), 5); % R = 5
model.variables.coefB = randn(5, d); % d-1 is the degree of the polynomial
model.variables.coefC = randn(5, d); % d-1 is the degree of the polynomial
model.factors{1}      = 1; % unconstrained factor A
model.factors{2}      = {2, @(z,task) struct_poly(z,task,tb)}; % poly in points tb
model.factors{3}      = {3, @(z,task) struct_poly(z,task,tc)}; % poly in points tc
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = 1:3;
```

Using `transform`, the code becomes:

```
model = struct;
model.variables.A     = randn(size(T,1), 5); % R = 5
model.variables.coefB = randn(5, d); % d-1 is the degree of the polynomial
model.variables.coefC = randn(5, d); % d-1 is the degree of the polynomial
model.factors{1}      = 1; % unconstrained factor A
model.transform{1}    = {2:3, @struct_poly, {tb, tc}}; % tb, tc are
                                                       % evaluation points
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd  = 1:3;
```

Note that there is no need to define an anonymous function `@(z,task) struct_poly(...)`. The second factor still has evaluation points `tb` as parameters while the third factor has evaluations points `tc`. This way many parameters can easily be passed to the transformations. Consider the following transform (see `help struct_poly`

for more information on the different parameters):

```
model.transform{1} = {1:3, @struct_poly, {t1,t2,t3}, 'chebyshev', {true}};
% which is equivalent to
model.factors{1} = {1, @(z,task) struct_poly(z,task,t1,'chebyshev',true)};
model.factors{2} = {2, @(z,task) struct_poly(z,task,t2,'chebyshev',true)};
model.factors{3} = {3, @(z,task) struct_poly(z,task,t3,'chebyshev',true)};
```

Hence, cell parameters are distributed if the length of the cell is equal to the number of factors to be created. If the cell parameter has length one, each factor gets the content of the cell as argument. (Cells with a different length are not allowed.) All other parameter types are copied for each transformation. It is also possible to chain transformations:

```
model.transform{1} = {1:3, @struct_vander, @struct_transpose};
```

Each transformation can have its own set of parameters.

As shown, multiple variables can easily be transformed into multiple factors using the `transform` field if the transformation is the same for all factors. There are some (minor) restrictions regarding naming, however. The `transform` field cannot overwrite factors with the same name or index (remember that if no factor names/indices are specified, the variable names/indices become factor names/indices); multiple `transform` entries cannot define the same factor names or indices; and if named (indexed) factors are used, named (indexed) factor references in the `transform` field should be used. More details on the syntax of the `transform` field can be found in the full language specification (Section 12.6).

10.3 Using implicit factors

It is quite common to have lines like

```
model.factors.A = 'a';
```

In these lines the variable is used directly as a factor as no transformation or concatenation of subfactors is performed. These spurious lines can be avoided by allowing factors to be defined implicitly. This can be done by setting the `model.options.ImplicitFactors = true`. When implicit factors are allowed, variables can be used directly in factorizations, as a factor with the same name is implicitly defined. Consider the following example:

```
model = struct;
model.variables.A = rand(10,5);
model.variables.B = rand(11,5);
model.variables.C = rand(12,5);
model.factors.A = 'A';
model.factors.B = 'B';
model.factors.C = 'C';
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd = {'A','B','C'};
```

This model can also be defined by

```
model = struct;
model.variables.A = rand(10,5);
model.variables.B = rand(11,5);
model.variables.C = rand(12,5);
model.factorizations.tensor.data = T;
```

```
model.factorizations.tensor.cpd = {'A','B','C'};
model.options.ImplicitFactors = true;
```

No factors with names `A`, `B` or `C` can be found, but variables with these names exist. Therefore, the factors `A`, `B` and `C` are automatically created. If, for example, a factor with `A` as name does exist, the latter is used. The result can be checked using `sdf_check(model,'print')`. Of course the model can also be defined as

```
model = struct;
model.variables.A = rand(10,5);
model.variables.B = rand(11,5);
model.variables.C = rand(12,5);
model.factors = 1:3;
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd = 1:3;
```

If some factors are structured, however, the simple line `model.factors = 1:3;` cannot be used. Implicit factors are more convenient in this case. As an example, consider a model that involves two unstructured factors `A` and `C` and a nonnegative factor `B`.

```
model = struct;
model.variables.A = rand(10,5);
model.variables.B = rand(11,5);
model.variables.C = rand(12,5);
model.factors{2} = {'B', @struct_nonneg};
model.factorizations.tensor.data = T;
model.factorizations.tensor.cpd = 1:3;
model.options.ImplicitFactors = true;
```

As implicit factors can be confusing for inexperienced users, this option is turned off by default. The full language specification (Section Section 12.7.1) explains a method to turn the option on by default. This way the line `model.options.ImplicitFactors` is no longer needed in every model definition.

10.4 High accuracy for NLS algorithms

Sometimes accuracy up to machine precision is required. The nonlinear least squares (NLS) type algorithms like `cpd_nls`, `ll1_nls`, `lmlra_nls` and `btd_nls` are often a good choice to achieve high accuracy thanks to their fast convergence in the neighborhood of optima. This means that once the algorithm has found an iterate close to an optimum, the optimum is found up to machine precision a few iterations later. To get high accuracy results, the relative function tolerance `TolFun` and the step size tolerance `TolX` should be small enough, e.g., `TolFun = eps^2` and `TolX = eps`.

10.4.1 Number of CG iterations

If structure is imposed on one or more factors in a decomposition, it may seem that `sdf_nls` does not always have this fast convergence property. The reason behind this is rather technical, but the main idea is that underlying linear systems are not solved accurately enough by the Conjugate Gradients (CG) algorithm, as not enough CG iterations are performed. By default the maximum number of iterations is limited by `CGMaxIter = 15`. Often the number of iterations for the optimization routine can be significantly reduced if the maximum number of CG iterations is increased, e.g., `CGMaxIter = 50` or `CGMaxIter = 100`. Exceptionally, `CGMaxIter` needs to be

increased to the number of underlying variables. Increasing `CGMaxIter` is likely to increase the cost per iteration, but often the number of iterations of the optimization routines decreases resulting in a net gain.

Instead of increasing `CGMaxIter` it is also possible to use the `sdf_minf` solver, as `sdf_minf` avoids solving this linear system. When to use `sdf_minf` or `sdf_nls` with increased `CGmaxIter` is problem specific.

As an example, consider the following nonnegative tensor factorization:

```
R = 5;
T = ful(cpd_rnd([40 50 60], R, 'Real', @rand));
model = struct;
model.variables = cpd_rnd([40 50 60], R);
model.transform = {1:3, @struct_nonneg};
model.factorizations{1}.data = T;
model.factorizations{1}.cpd  = 1:3;

[sol,output] = sdf_nls(model, 'Display', 100, 'TolX', eps, 'TolFun', eps^2);
```

The algorithm prints its progress every 100 iterations:

```
      fval              relfval           relstep           delta        rho
      =1/2*norm(F)^2    TolFun = 5e-32    TolX = 2e-16

  0:  1.05508511e+02 |
  1:  1.15611717e+01 | 8.90424274e-01 | 1.52025153e-01 | 8.0584e+00 | 8.9202e-01
100:  4.61577625e-11 | 2.11901351e-14 | 8.20225374e-06 | 1.7762e-01 | 1.0063e+00
200:  1.05870120e-12 | 3.18735863e-16 | 1.07081681e-06 | 1.7762e-01 | 9.9611e-01
300:  4.59820639e-14 | 1.39850518e-17 | 2.29709351e-07 | 1.7762e-01 | 1.0072e+00
400:  2.37198537e-15 | 6.68979657e-19 | 4.45819524e-08 | 1.7762e-01 | 9.8808e-01
500:  1.07873831e-16 | 3.13699148e-20 | 1.09605852e-08 | 1.7762e-01 | 9.9295e-01
```

As can be seen, machine precision is not attained within the maximum number of iterations. When looking at the number of CG iterations (`output.cgiterations`) it is clear that they are clipped against the maximum number of CG iterations:

```
[min(output.cgiterations) max(output.cgiterations)]
```

```
ans =
 15    15
```

When the maximum number of CG iterations is increased to 150, the number of iterations decreases significantly:

```
sol = sdf_nls(model, 'Display', 10, 'TolX', eps, 'TolFun', eps^2,...
              'CGMaxIter', 150);
```

```
     fval              relfval           relstep           delta        rho
     =1/2*norm(F)^2    TolFun = 5e-32    TolX = 2e-16

 0:  1.05508511e+02 |
 1:  1.18695988e+01 | 8.87501030e-01 | 2.14514488e-01 | 1.1371e+01 | 8.8881e-01
10:  8.24322930e-03 | 2.17476982e-05 | 4.20034477e-02 | 1.6564e+00 | 1.0810e+00
20:  1.82692225e-13 | 7.25666009e-13 | 1.03495202e-04 | 1.3658e+00 | 9.9992e-01
30:  1.75577946e-26 | 2.92600487e-27 | 1.38043361e-11 | 1.3658e+00 | 9.9998e-01
34:  1.69764270e-31 | 2.60423058e-32 | 4.11640579e-14 | 1.3658e+00 | 9.9753e-01
```

Now only 34 iterations are needed to obtain a solution that is accurate up to machine precision, which can be validated by computing `cpderr` .

10.4.2 Incomplete tensors

By default, `sdf_nls` uses an approximation for the Gramian of the Jacobian in the case of incomplete tensors. This approximation is fast and works well if the number of missing entries is low and the missing entries are spread out uniformly across the tensor. When very few entries are known or when the missing entries follow a pattern, this approximation is no longer valid. This can result in very slow convergence. An exact implementation of the Gramian of the Jacobian is available that allows fast convergence regardless of the number of missing entries. This implementation can be selected by using the `cpdi` factorization type, instead of `cpd` [15]:

```
model.factorizations{1}.data = T;
model.factorizations{1}.cpdi = {'A','B','C'};
```

While the number of iterations in the optimization algorithm usually decreases, each iteration is more expensive. However, when very few entries are known, there is often a gain.

The remarks about the CG iterations in the previous subsection also hold for the `cpdi` factorization type. `sdf_minf` does not use the Gramian of the Jacobian, and therefore it does not have a problem with incomplete tensors.

CHAPTER

ELEVEN

STRUCTURED DATA FUSION: IMPLEMENTING A TRANSFORMATION

In SDF, structure can be imposed on a factor by transforming variables z into a factor x, i.e., x is a function $x : z \to x(z)$. An example is `struct_nonneg` in which nonnegativity is imposed by using the transformation $x(z) = z^2$ for each entry. This chapter discusses the different elements needed to implement a transformation and illustrates how the user can implement his or her own transformations. This way new types of structure can be imposed on factors. All concepts are illustrated by two examples: the implementation of `struct_sqrt` which implements a square root constraint, and the implementation of a (simplified) version of `struct_poly` which imposes that a factor has polynomial factor vectors.

11.1 Elements in a transformation

Each transformation implements at least three operations or tasks: the function evaluation, the right Jacobian-vector product and the left Jacobian-vector product. In this section, the global structure of a transformation is explained, as well as the different tasks. As this is a rather technical section, it can be useful to first read the examples in Section 11.2.

Each transformation accepts at least two input arguments: the underlying variables `z` and the `task` to perform. Two output arguments are needed: the computed result `x` which depends on the task, and `state` (which should be set to `[]` if not used). The function definition of a transformation looks like:

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

There are three tasks to be implemented depending on the fields in the `task` struct:

- `~isempty(task.r)` : `x` is the the right Jacobian-vector product $\frac{\partial x}{\partial z} \cdot r$ in which r is `task.r`.
- `~isempty(task.l)` : `x` is the left Jacobian-vector product $(\frac{\partial x}{\partial z})^{\mathsf{H}} \cdot l + (\frac{\partial x}{\partial \bar{z}})^{\mathsf{T}} \cdot l$ in which l is `task.l`.
- Otherwise (`task` can be `[]`): `x` is the function evaluation $x(z)$.

The different tasks are now discussed separately.

11.1.1 Function evaluation

The transformation returns the function evaluation $x(z)$ if `task` is empty, or both `task.r` and `task.l` are empty. To increase the readability, we introduce the variables `right` and `left` :

```
if nargin < 2, task = []; end
right = ~isempty(task) && isfield(task, 'r') && ~isempty(task.r);
left  = ~isempty(task) && isfield(task, 'l') && ~isempty(task.l);

if ~right && ~left
    % function evaluation
```

The input argument `z` is the variable, in the same format as defined in the model. For example, if a variable `a` defined as

```
model.variables.a = randn(10,3);
```

and is transformed, then `z = a` is a matrix of size 10×3. `z` can also be the output of a previous transformation, e.g.,

```
t = 1:10;
model.variables.c = rand(5,3);
model.factors.A = {'c', @(z,task) struct_poly(z,task,t), @struct_nonneg};
```

For `struct_poly`, `z` is the variable `c` (a 5×3 matrix), while for `struct_nonneg`, `z` is the output of `struct_poly` (a 10×5 matrix).

Depending on the transformation, the output `x` can be a scalar, vector, matrix or tensor. `x` can also be a cell, which can be processed by subsequent transformations.

11.1.2 Right Jacobian-vector product

The transformation returns the right Jacobian-vector product if `task.r` is not empty. `task.l` is always empty in this case. This right Jacobian-vector product is only computed in NLS type algorithms. In this task, the derivative $\frac{\partial x}{\partial z} \cdot r$ is computed, in which r is given in `task.r`. Mathematically, x, z and r are vectors of length F, V and V respectively. Suppose $D = \frac{\partial x}{\partial z}$ is an $F \times V$ matrix, then we have $d_{ij} = \frac{\partial x_i}{\partial z_j}$. To simplify the code, `task.r` is not a vector, but has the same format as the input variable `z`, while the result `x` has the same format as the resulting factor would have if $x(z)$ was evaluated. In the code of the transformation, the following lines are added:

```
elseif right
    % compute right Jacobian-vector product
```

The right Jacobian-vector product can only be computed if the function x depends solely on z, and hence not on its conjugate $\bar{z}$. If x is a function of both z and $\bar{z}$, i.e., $x(z, \bar{z})$, an error should be thrown if the input data is complex:

```
elseif right
    if ~isreal(z) || isreal(task.r)
        error('struct_name:nonanalytic', ['Nonanalytic objective ' ...
              'functions are currently not supported in sdf_nls, please ' ...
              'use sdf_minf instead.']);
    end
```

11.1.3 Left Jacobian-vector product

The transformation returns the left Jacobian-vector product if `task.l` is not empty (`task.r` is always empty in this case). This left Jacobian-vector product is computed in all algorithms. In this task, the derivative

$(\frac{\partial x}{\partial z})^{\mathsf{H}} \cdot l + (\frac{\partial x}{\partial \bar{z}})^{\mathsf{T}} \cdot l$ is computed, in which l is given. Mathematically, x, z and l are vectors of length F, V and F respectively. Suppose $D = \frac{\partial x}{\partial z}$ is an $F \times V$ matrix, then each entry is given by $d_{ij} = \frac{\partial x_i}{\partial z_j}$. To simplify the code, `task.l` is not a vector, but has the same format as as the output `x` would have if $x(z)$ was evaluated. The result of the derivative `x` should have the same format as the input variable `z`. In the code of the transformation, the following lines are added:

```
elseif left
    % compute left Jacobian-vector product
end
```

To compute $\frac{\partial x}{\partial z}$, $\bar{z}$ is assumed to be constant, while z is kept constant when $\frac{\partial x}{\partial \bar{z}}$ is computed. Hence, if x is only a function of z, i.e., $x(z)$, the derivative w.r.t. $\bar{z}$ is zero. If we have $x(z, \bar{z})$ both derivatives need to be computed.

11.2 Examples

11.2.1 Implementing `struct_sqrt`

The transformation `struct_sqrt` implements the function $f : \mathbb{R}^{I \times R} \to \mathbb{R}^{I \times R} : x = \sqrt{z}$ in which the square root is applied entry-wise. Three elements are required for this transformation: the function evaluation, the right Jacobian-vector product and the left Jacobian-vector product.

The function evaluation is simply `x = sqrt(z)`. The right and left Jacobian-vector products are given by $\frac{\partial x}{\partial z} \cdot r$ and $(\frac{\partial x}{\partial z})^{\mathsf{H}} \cdot l + (\frac{\partial x}{\partial \bar{z}})^{\mathsf{T}} \cdot l$, respectively. The derivative $\frac{\partial x}{\partial z}$ is given by $\frac{1}{2\sqrt{z}}$. When computing the derivative w.r.t. z, $\bar{z}$ is kept constant. As x does not depend on $\bar{z}$, the derivative $\frac{\partial x}{\partial \bar{z}} = 0$. Hence, the complete code for the transformation is given by:

```
function [x,state] = struct_sqrt(z,task)
state = [];
if nargin < 2, task = []; end
right = ~isempty(task) && isfield(task, 'r') && ~isempty(task.r);
left  = ~isempty(task) && isfield(task, 'l') && ~isempty(task.l);
if ~right && ~left
     x = sqrt(z);
elseif right
     x = 1./(2*sqrt(z)).*task.r;
elseif left
     x = conj(1./(2*sqrt(z))).*task.l;
end
```

11.2.2 Implementing `struct_poly`

The transformation `struct_poly` imposes polynomial structure on each of the columns of a factor matrix, i.e., each entry of the factor x (an $I \times R$ matrix) is given by $x_{ir} = c_{r0} + c_{r1}t_i + c_{r2}t_i^2 + \cdots + c_{rd}t_i^d$. The points t_i, $i = 1, \ldots, I$ are fixed. The coefficients c (an $R \times (d+1)$ matrix) are the variables z (the highest-degree coefficients c_{rd} form the first column).

Again, the different tasks are implemented. The first lines of the transformation are similar to `struct_sqrt`:

```
function [x, state] = struct_poly(z, task, t)
```

```
    if nargin < 2, task = []; end
    right = ~isempty(task) && isfield(task, 'r') && ~isempty(task.r);
    left  = ~isempty(task) && isfield(task, 'l') && ~isempty(task.l);
    state = [];
```

The function evaluation can be written as a matrix multiplication by using a basis matrix `B` in which each column is a power of the point vector `t`, i.e., `x = B*z`:

```
if ~right && ~left
    B = bsxfun(@power, t(:), size(z, 2)-1:-1:0);
    x = B*z;
```

The derivative $\frac{\partial x}{\partial z}$ is simply given by `B`, hence the right and left Jacobian-vector products are given by

```
elseif right
    B = bsxfun(@power, t(:), size(z, 2)-1:-1:0);
    x = B*task.r.';
elseif left
    B = bsxfun(@power, t(:), size(z, 2)-1:-1:0);
    x = task.l.'*conj(B);
end
```

11.3 Storing intermediate results

Some transformations require expensive computations, which may depend on the current value of the variables, but also on extra input parameters that stay constant over all iterations. In the latter case, persistent storage can be used to store precomputed results, while the `state` field can be used in the former case. First, the use of the `state` field is illustrated for a transformation that implements the square root constraint: `struct_sqrt`. Next, the use of the `persistent` field is illustrated for the polynomial transformation `struct_poly`.

11.3.1 Using the `state` field

In the `struct_sqrt` example, the quantity `sqrt(z)` and the division `1./(2*sqrt(z))` are computed many times in each iteration, as the three tasks are performed multiple times per iteration depending on the chosen optimization algorithm. A property of the optimization-based algorithms used in Tensorlab is that every time a variable is changed, the function value is computed. In other words, the variable `z` does not change between function value computations. Therefore, the variable `z` seen by the right and left Jacobian-vector products, is the same as in the last function evaluation. The computation time can hence be reduced by performing computations depending on `z` in the function evaluation and by reusing these results in the right and left Jacobian-vector product computations. The `state` output field allows the temporary storage of such intermediary results.

The `state` output is a struct. Each field set during the function evaluation is available under the same name in the `task` struct. In this example, a field `dz` is constructed to store the result `1./(2*sqrt(z))`. To compute the right and left Jacobian-vector product, this precomputed value is used:

```
function [x,state] = struct_sqrt(z,task)
state = [];
if nargin < 2, task = []; end
right = ~isempty(task) && isfield(task, 'r') && ~isempty(task.r);
left  = ~isempty(task) && isfield(task, 'l') && ~isempty(task.l);
```

```
if ~left && ~right
    x = sqrt(z);
    state.dz = 1./(2*x);
elseif right
    x = task.dz.*task.r;
elseif left
    x = conj(task.dz).*task.l;
end
```

As can be seen, the value of `state.dz` is only computed when the function is evaluated. As the function evaluation is always called before the left and right Jacobian-vector products, `state` should be set in the function evaluation task. The value of `state` is ignored in the other calls, hence `state` can be left empty (`[]`) for the right and left Jacobian-vector products.

11.3.2 Persistent storage

In the previous example, the precomputed value depends on the variable z. As this variable changes every iteration, the `state` needs to be recomputed every iteration. In the `struct_poly` transformation, the matrix basis `B` is computed every function evaluation, but `B` is independent of the variables. Hence, setting `state.B = B` every iteration reduces the computational cost a little, but this still wastes a lot of computation time. Therefore, a special field is provided: the `persistent` field. `B` can be computed in the first call to `struct_poly` and stored in `state.persistent.B = B`. In all subsequent calls, the struct `task` will contain the field `persistent` which contains `B` as a field. Hence, the transformation can be rewritten as:

```
function [x, state] = struct_poly(z, task, t)

   if nargin < 2, task = []; end
   right = ~isempty(task) && isfield(task, 'r') && ~isempty(task.r);
   left  = ~isempty(task) && isfield(task, 'l') && ~isempty(task.l);

   if ~isempty(task) && isfield(task, 'persistent')
       B = task.persistent.B;
       state = [];
   else % not present, compute
       B = bsxfun(@power, t(:), size(z, 2)-1:-1:0);
       state.persistent.B = B;
   end

   if ~left && ~right
       x = B*z;
   elseif right
       x = B*task.r.';
   elseif left
       x = task.l.'*conj(B);
   end
end
```

`B` is now computed only once. In subsequent iterations `B` can be loaded from the `task.persistent` struct to be used in further computations. Note that `state = []` if `B` has already been computed: there is no need to pass persistent data as this is done in the background.

11.3.3 Comparing `state` and `persistent`

Setting the `state` field in the function value call is useful to store intermediate results needed for the right and left Jacobian-vector products. The `state` field can be used if the intermediate results depend on the variables `z` and possibly on extra input parameters.

The `state.persistent` field can be used to store precomputed data that depend solely on extra input parameters.

11.4 Non-numerical variables

In the previous examples, only numerical values are used as variable z and output x. Cells can be used as well, which is useful if several variables need to be combined. As an example, consider the `struct_poly` example again. Now, instead of taking fixed points `t`, the evaluation points are considered variables as well, i.e.:

```
model.variables.z = {rand(5,3), rand(10,1)}; % note this is a cell
model.factors.x = {'z', @struct_poly2};
```

We reimplement the three tasks. Note that this example is given for illustration purposes only, and that the code below may not be the best way to implement this kind of structure.

The function evaluation for `struct_poly2` is very similar as for `struct_poly`, only now `z` is a cell with the coefficient `c = z{1}` and the evaluation points `t = z{2}`:

```
function [x, state] = struct_poly(z, task, t)
   % trivial parts omitted
   c = z{1}; t = z{2};
   if ~left && ~right
       B = bsxfun(@power, t(:), size(c,2)-1:-1:0);
       x = B*c;
```

Note that the `persistent` field cannot be used here, as `t` changes every iteration.

For the right Jacobian-vector product, `task.r` has the same format as the input variable `z`, i.e., `task.r` is a cell of length two. The output `x` has the same format as the factor. The derivative $\frac{\partial x}{\partial z} = [\frac{\partial x}{\partial c}, \frac{\partial x}{\partial t}]$ is multiplied by $[\mathrm{vec}\,(r_c)^{\mathsf{T}}, \mathrm{vec}\,(r_t)^{\mathsf{T}}]^{\mathsf{T}}$ in which r_c and r_t are the parts of r corresponding to the variables c and t, respectively. The multiplication can be written as $\frac{\partial x}{\partial c}\mathrm{vec}\,(r_c) + \frac{\partial x}{\partial t}\mathrm{vec}\,(r_t)$. Therefore, the right Jacobian-vector product can be computed as

```
elseif right
    % derivative w.r.t. c
    B = bsxfun(@power, t(:), size(c,2)-1:-1:0);
    x = B*task.r{1}.';
    % derivative w.r.t. t
    cder = [zeros(size(c,1),1),
            bsxfun(@times, c(:,1:end-1),size(c,2)-1:-1:1)];
    x = x + (B*cder).*task.r{2};
```

Finally, the left Jacobian-vector product is computed. `task.l` has the same format as the resulting factor. The output `x` should have the same format as the variable `z` in this case. Following the same reasoning as for the right Jacobian-vector product, the following result is obtained:

```
elseif left
    % derivative w.r.t. c
    B = bsxfun(@power, t(:), size(c,2)-1:-1:0);
    cl = B'*task.l;
    % derivative w.r.t. t
    cder = [zeros(size(c,1),1),
            bsxfun(@times, c(:,1:end-1),size(c,2)-1:-1:1)];
    tl = conj(B*cder).*task.l;
    x = {cl.', tl};
end
```

As before, the repeated computations of `B` and `cder` can be avoided by computing both during the function evaluation and using `state.B` and `state.cder` to store the results.

The transformations `struct_poly` and `struct_poly2` can be merged. The merged transformation executes the code for `struct_poly2` if `z` is a cell. Otherwise, the results for `struct_poly` are returned.

CHAPTER

TWELVE

STRUCTURED DATA FUSION: SPECIFICATION OF DOMAIN SPECIFIC LANGUAGE

An extensive, more technical overview of the domain specific language (DSL) to express structured data fusion (SDF) models is given in this section. The specification allows the user to get insight in how the language works, enabling him or her to unlock the full potential of the SDF framework.

This chapter is structured as follows. First we give an overview of important definitions w.r.t. naming, referencing and auto-wrapping. Next the model and its different fields (variables, factors, factorizations, transform and options) are explained.

12.1 Definitions

To simplify and shorten the specification of the DSL, we first define some frequently used concepts. The following concepts are explained:

- Named and indexed definitions and references
- Valid references
- Auto-wrapping

12.1.1 Named and indexed definitions

The `variables`, `factors`, `transform` and `factorizations` fields can be defined in two ways: as a struct or as a cell. The following definitions are equivalent:

```
% named definition
model.variables.A = A;
model.variables.B = B;

% indexed definition
model.variables{1} = A;
model.variables{2} = B;
```

If the field is a struct, it is converted to a cell using `struct2cell`. This means that the order in which the variables appear first remains unchanged. Hence the following two blocks are equivalent:

```
% named definition
model.variables.B = B;
model.variables.A = A;
```

```
model.variables.B = B; % B is already defined, and therefore is still the
                       % first variable

% indexed definition
model.variables{1} = B;
model.variables{2} = A;
```

The fields are not sorted and redefining a field (see `B`) does not change the order of the fields.

Variables, factors, transforms and factorizations can be defined using names or indexes. References to variables and factors can also be named or indexed. For example:

```
% first define named variables
model.variables.A = A;
model.variables.B = B;

% named references
model.factors.f1 = {'A', 'A'};
model.factors.f2 = 'B';

% indexed references (equivalent to the above)
model.factors.f1 = {1, 1};
model.factors.f2 = 2;

% mixing named and index references (also equivalent)
model.factors.f1 = {1, 'A'};
model.factors.f2 = 2;
```

Named references can only be used if named definitions are used for the variable or factor that is being referred to. For example:

```
% indexed definition for variables
model.variables{1} = A;
model.variables{2} = B;
% named definitions for factors, only indexed references to variables can be
% used.
model.factors.A = 1;
model.factors.B = 2;
% indexed definition of factorizations (both named and indexed references)
model.factorizations{1}.regL1 = {'A', 2};
```

In the remaining part of this section, named and indexed definitions/references are used interchangeably.

12.1.2 Valid references

A valid reference is a named or indexed reference to a named or indexed variable or factor that can be resolved, i.e., a variable or factor with the given name or index exists, regardless of how it is constructed. Recall that factors can be constructed using a `factors` or a `transform` field, or can be defined implicitly. Named variables or factors also have indices. As an example, consider

```
model.factors.F = {'a', 2};
```

The factor `F` has two references `a` and 2. The named reference `a` is valid if the variables are defined using named definitions (`model.variables` is a struct) and `model.variables` contains a field with name `a`. Names are

case sensitive. The indexed reference `2` is valid if `model.variables{2}` exists or if `model.variables` is a struct and at least two variables are defined. Hence, `2` is a valid reference if `length(model.variables) >= 2`. The same holds for factor references in factorizations. Note that the `transform` field generates new factors as if they are defined using the `factors` field. Also note that the `ImplicitFactors` option (see Section 12.7.1) allows factors to be defined implicitly, i.e., if a factor is not defined and if there exists a variable with a matching name or index, a new factor is created automatically.

12.1.3 Auto-wrapping

The auto-wrapping function automatically adds missing braces `{}` around content, or calling `num2cell` if this can be done unambiguously. Some examples:

```
model.factors = 1:3;
% is equivalent to
model.factors{1} = {{1}};
model.factors{2} = {{2}};
model.factors{3} = {{3}};

model.factors.A = rand(2,2);
% is equivalent to
model.factors.A = {{rand(2,2)}};

model.factors = {1, @struct_nonneg};
% is equivalent to
model.factors{1} = {{1, @struct_nonneg}};

model.factors{1} = {1, 1; 'a', {'b', @struct_nonneg}};
% is equivalent to
model.factors{1} = {{1}, {1}; {'a'}, {'b', @struct_nonneg}};

model.factorizations{1}.cpd = 1:3;
% is equivalent to
model.factorizations{1}.cpd = {1,2,3};
% but the following is invalid
model.factorizations{1}.cpd = {1:3};

model.factorizations{1}.btd = 1:4;
% is equivalent to
model.factorizations{1}.btd = {1:4};
% is equivalent to
model.factorizations{1}.btd = {1,2,3,4};
% is equivalent to
model.factorizations{1}.btd = {{1,2,3,4}};
% but the following is invalid
model.factorizations{1}.btd = {{1:4}};
% mixing is allowed though:
model.factorizations{1}.btd = {{1,2,3,4}, 5:8};

model.transform = {1:3, 3:5, @struct_nonneg};
% is equivalent to
```

```
model.transform = {{1:3, 3:5, @struct_nonneg}};
% is equivalent to
model.transform{1} = {1:3, 3:5, @struct_nonneg};
% is equivalent to
model.transform{1} = {{1,2,3}, {3,4,5}, @struct_nonneg};
```

In the examples above, the last lines (excluding the invalid cases) are the internal representations used. All other formulations facilitate the definition of the models. If in doubt, `sdf_check(model, 'print')` can be used to see if the desired model has been constructed.

12.2 The model struct

Each SDF model is defined as a struct (in Matlab's definition). Five different fields can be defined, some of which are always obligatory, others are only in some cases obligatory, and some are optional. The five fields are:

- `model.variables` : always obligatory
- `model.factors` : obligatory unless the `model.transform` field is present or implicit factors are used
- `model.factorizations` : always obligatory
- `model.transform` : optional
- `model.options` : optional

In the remaining part of this section, the different fields are discussed.

12.3 The variables field

The `model.variables` field is a struct or cell defining named or indexed variables, respectively. Here, we only consider indexed variables as named variables are converted to indexed variables internally. (The names are kept to be able to resolve named references.) Each variable can be either an array of numerical values, or a cell of arrays of numerical values.

If named variables are used (and hence `model.variables` is a struct) the name `(constant)` cannot be used.

12.4 The factors field

The `model.factors` field is a struct or cell defining named or indexed factors, respectively. Here, we only consider indexed factors as named factors are converted to indexed factors internally (while keeping track of the names for references). Each entry in the cell defines a single factor, and has one of the following forms (after auto-wrapping):

```
% one subfactor with variable v1
model.factors{1} = {{v1}};
% two subfactors concatenated column-wise, no transformation
model.factors{1} = {{v1}, {v2}};
% two subfactors concatenated row-wise, with transformation t1
model.factors{1} = {{v1,t1}; {v2}};
% two subfactors concatenated in mode 3, with transformations t1, t2, t3
model.factors{1} = reshape({{v1, t1}, {v2,t1,t2,t3}}, 1,1,2);
```

```
% two subfactors concatenated in mode 3, with transformed constant c1
model.factors{1} = {{v1}, {c1,t1}};
```

Each factor can be a concatenation of an arbitrary number of subfactors, as long as the dimensions of the subfactors are compatible, i.e., if the dimensions of the expanded subfactors are compatible for concatenation using `cell2mat` (see `sdf_check(model, 'print')`). Each subfactor is of the form `{v1,t1,t2,...}` or `{c1,t1,t2,...}` in which `v1` is a valid variable reference and `c1` is a constant. Each variable reference or constant can be followed by an arbitrary number of transformations `t1`, `t2`, etc. Each transformation is a function handle that accepts two arguments: a variable `z` and a `task` struct. If extra parameters are to be defined, the following can be used:

```
t1 = @(z,task) struct_name(z,task,param1,param2,...);
```

All transformations are applied from left to right. For each subfactor, the result of the final transformation should be a scalar, vector, matrix or tensor.

Constants follow the same rules as variables, i.e., each constant is a scalar, vector, matrix, or tensor, or is a cell of scalars, vectors, matrices and/or tensors. All transformations defined for constants are applied before the actual computations start. An alternative syntax for constant subfactors uses the `(constant)` keyword:

```
model.factors{1} = {{c1, '(constant)', t1, t2, ..}};
```

The `(constant)` keyword is optional if `c1` is not a scalar. If `c1` is a scalar, it is recognized as a variable reference if the `(constant)` keyword is not given.

If implicit factors are allowed and if `model.factors` is a struct (and hence named factors are used), names of the form `IFx` in which `x` is a number are prohibited.

12.5 The factorizations field

The `model.factorizations` field defines the different parts of the objective function. These parts can be factorization or regularization terms. `model.transformations` is a struct or a cell, each entry being a single factorization or regularization term. Each such term is a struct with fields depending on the factorization or regularization type. In general, a factorization looks like

```
model.factorizations.fac1.data = T;
model.factorizations.fac1.cpd  = {f1,f2,f3};
```

in which `f1`, `f2` and `f3` should be valid named or indexed factor references.

As a special case, `model.factorizations` can be a struct array as well, e.g.,

```
T = {rand(10,11,12), randn(12,13,14)};
couplingind = {1:3, 3:5};
model.factorizations = struct('data', T, 'cpd', couplingind);

% is equivalent to
model.factorizations{1}.data = rand(10,11,12);
model.factorizations{1}.cpd  = 1:3;
model.factorizations{2}.data = rand(12,13,14);
model.factorizations{2}.cpd  = 3:5;
```

This only works if `numel(model.factorizations) > 1`.

12.5.1 Factorization types

Seven different factorization types are currently defined: four factorizations (`cpd` (and a `cpdi` variant), `btd`, `ll1` and `lmlra`) and three regularizations (`regL0`, `regL1` and `regL2`). Each factorization defines a field with the factorization or regularization type as name, and a (possibly nested) cell of factor references as values. Apart from the factorization type, a second field `data` is required for the factorizations, and optional for the regularizations. The `data` has as value the tensor to be decomposed or the value to be regularized to.

The different factorization and regularization types are now discussed. The shorthand notation $\mathcal{M}_{\text{type}}(\mathbf{A},\ldots)$ is used to define a factorization of type `type` involving factors `A`, etc.

- `cpd`: a term $\frac{1}{2}\,||\mathcal{T}-\mathcal{M}_{\text{CPD}}(\mathbf{A},\mathbf{B},\ldots)||_{\text{F}}^2$ is added to the objective function. The references in the `cpd` field indicate which factors are involved in this term. The `data` field defines the tensor $\mathcal{T}$. For example:

  ```
  model.factorizations{1}.cpd = {'A','B',...};
  ```

  In the case $\mathcal{T}$ is an incomplete tensor, a more accurate algorithm can be selected if an NLS based solver is used. For SDF models this algorithm is chosen by using the `cpdi` factorization instead of `cpd`, e.g.,

  ```
  model.factorizations{1}.cpdi = {'A','B',...};
  ```

- `btd`: a term $\frac{1}{2}\,||\mathcal{T}-\mathcal{M}_{\text{BTD}}(\mathbf{A}_1,\mathbf{B}_1,\mathbf{C}_1,\mathcal{S}_1,\mathbf{A}_2,\mathbf{B}_2,\mathbf{C}_2,\mathcal{S}_2,\ldots)||_{\text{F}}^2$ is added to the objective function. The references in the `btd` field indicate which factors are involved in this term. The `btd` field is a nested cell of factor references (after auto-wrapping), e.g.,

  ```
  model.factorizations{1}.btd = {{'A1','B1','C2','S1'},{'A2','B2','C2','S2'}};
  ```

  The `data` field defines the tensor $\mathcal{T}$.

- `ll1`: a term $\frac{1}{2}\,||\mathcal{T}-\mathcal{M}_{\text{LL1}}(\mathbf{A},\mathbf{B},\mathbf{C})||_{\text{F}}^2$ is added to the objective function. The references in the `ll1` field indicate which factors are involved in this term. The LL1 term only accepts factors in the CPD format, e.g.,

  ```
  model.factorizations{1}.ll1 = {'A', 'B', 'C'};
  model.factorizations{1}.L = L;
  ```

  The extra field `L = [L1,L2,...,LR]` is obligatory. `A` and `B` have `sum(L)` columns, and `C` has `length(L)` columns. The `data` field defines the tensor $\mathcal{T}$.

- `lmlra`: a term $\frac{1}{2}\,||\mathcal{T}-\mathcal{M}_{\text{LMLRA}}(\mathbf{A},\mathbf{B},\mathbf{C},\ldots,\mathcal{S})||_{\text{F}}^2$ is added to the objective function. The references in the `lmlra` field indicate which factors are involved in this term. The `lmlra` field is a cell, or a cell of cells (after auto-wrapping), e.g.,

  ```
  model.factorizations{1}.lmlra = {'A','B','C','S'}; % or
  model.factorizations{1}.lmlra = {{'A','B','C','S'}};
  ```

  Note that in contrast to other LMLRA routines such as `lmlra_rnd` and `lmlra` and in contrast to the structured tensor format, the correct format here is `{A,B,C,S}` and not `{{A,B,C},S}`. The `data` field defines the tensor $\mathcal{T}$.

In the case of regularization types, define $\mathcal{M}(\mathbf{A}) = ||\text{vec}\,(\mathbf{A})||_0$ for type `regL0`, $\mathcal{M}(\mathbf{A}) = \frac{1}{2}\,||\text{vec}\,(\mathbf{A})||_1$ for type `regL1` and $\frac{1}{2}\mathcal{M}(\mathbf{A}) = ||\mathbf{A}||_{\text{F}}^2$ for type `regL2`, in which $\mathbf{A}$ can be a scalar, vector, matrix or tensor. We then have (replace `reg` by `regL0`, `regL1` or `regL2`):

- $\mathcal{M}(\mathbf{A})$ can be modeled as

```
model.factorizations.regterm.reg = {'A'};
```

- $(\mathcal{M}(\mathbf{A}) + \mathcal{M}(\mathbf{B}))$ can be modeled as

```
model.factorizations.regterm.reg = {'A', 'B'};
```

- $(\mathcal{M}(\mathbf{1} - \mathbf{A}) + \mathcal{M}(\mathbf{2} - \mathbf{B}))$ can be modeled as

```
model.factorizations.regterm.data = {ones(size(A)), ones(size(B))*2};
model.factorizations.regterm.reg  = {'A', 'B'};
```

In the case of L0 regularization, a smoothed approximation of the L0 norm is used for computational reasons: $\|\text{vec}\,(\mathbf{A})\,\|_0$ is approximated by $\sum_{i=1}^{I} 1 - \exp(-\frac{a_i^2}{\sigma^2})$ with I the number of entries in $\mathbf{A}$ [39]. The parameter σ can be a scalar constant or a function of the iteration `k` and the cell of regularized factors `x`. The value can be set using the `sigma` field:

```
model.factorizations.myreg.regL0 = 'A';
model.factorizations.myreg.sigma = 0.01; % (the default), or
model.factorizations.myreg.sigma = @(k,x) 0.01*frob(x{1}); % x = cell of factors
```

For all the cases above, `sdf_check(model)` (which is automatically called before all computational routines) checks if the given data and factors are consistent, i.e., if the assumptions for the model or type hold, and match with the provided data. For example, for the CPD all factors should be matrices with exactly R columns, and the number of rows of each factor should match the dimension of the data in the corresponding mode. It is recommended to run `sdf_check(model, 'print')` as this signals potential syntax and consistency errors.

Internally, the different fields in a factorization are redefined. The `type` field contains a string with the factorization type, the `factors` field contains the factor references. The original field, e.g., `cpd = 1:3`, is removed. For example:

```
model.factorizations{1}.myfac.data = T;
model.factorizations{1}.myfac.cpd  = {1,2,3};
% becomes
model.factorizations{1}.myfac.data     = T;
model.factorizations{1}.myfac.type     = 'cpd';
model.factorizations{1}.myfac.factors  = {1,2,3};
```

12.5.2 Defining weights

When different datasets are coupled or regularization terms are added, it is often needed to give terms a weight. For example:

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C}} \frac{\lambda_1}{2} ||\mathcal{T} - \mathcal{M}_{\text{CPD}}(\mathbf{A},\mathbf{B},\mathbf{C})||_{\text{F}}^2 + \frac{\lambda_2}{2} ||A||_{\text{F}}^2 + \frac{\lambda_3}{2}(||B||_1 + ||C||_1)$$

The weights λ_i are often called hyperparameters. Here, the term weight is used. The weights can be set as follows

```
model = struct;
% definition of variables and factors omitted
model.factorizations.tensor.data   = T;
model.factorizations.tensor.cpd    = {'A', 'B', 'C'};
model.factorizations.tensor.weight = lambda1;
model.factorizations.regA.regL2    = 'A';
```

```
model.factorizations.regA.weight   = lambda2;
model.factorizations.regBC.regL1   = {'B', 'C'};
model.factorizations.regBC.weight  = lambda3;
```

or can be set as an option of the computational routine, e.g.:

```
sdf_nls(model, 'Weight', [lambda1, lambda2, lambda3]);
```

If weights are provided to both the model and the computational routine, the latter weights take precedence. If weights are provided solely through the model, and some factorizations do not define a weight, `weight` is assumed to be `1`.

Instead of absolute weights, it is possible to define relative weights using the `relweight` field of a factorization or the `RelWeights` option of the computational routine. Relative weights ρ_i are translated into absolute weights using the following formula:

$$\lambda_i = 2\frac{\rho_i}{\sum_j \rho_j}\frac{1}{N_i}$$

in which N_i is the number of entries in a tensor `Ti`, i.e., `Ni = prod(getsize(Ti))`. If `Ti` is incomplete, then `Ni = length(Ti.val)`. For regularization terms, `Ni = sum(cellfun(@numel, f))` in which `f` is the cell of all expanded factors to be regularized by regularization term `i`.

In the case weights are provided by the model, absolute weights (the `weight` field) and relative weights (the `relweight` field) cannot be combined, i.e., if one factorization defines the field `weight` it cannot define `relweight` and no other factorization can define `relweight`.

12.5.3 Providing extra information

Some computational routines accept extra information provided by the model to speed up computations. Currently, the `ccpd_nls` and `ccpd_minf` routines exploit symmetry if symmetry is detected or if a dataset is explicitly declared symmetric using the `issymmetric` field. The example below exploits the symmetry between the first and second mode of `T`.

```
model = struct;
model.variables = {A, B};
model.factors{1} = {{1}};
model.factors{2} = {{2}};
model.factorizations{1}.data = T;
model.factorizations{1}.cpd = {1,1,2};
model.factorizations{1}.issymmetric = true;

Ures = ccpd_nls(model);
```

The `cpd` field defines the type of symmetry.

12.6 The transform field

The `model.transform` field facilitates the creation of multiple factors having a similar structure. Using a single line of code, multiple variables can be transformed using the same transformation. The transformations do not necessarily have the same parameters.

Consider the following example. Each column in factor matrix 1 is a polynomial evaluated in points `t1`. Similarly each column in factor matrix 2 is a polynomial evaluated in points `t2`, and the same holds for factor matrix 3. Without the `transform` field, this is written as:

```
model.factors{1} = {1, @(z,task) struct_poly(z, task, t1)};
model.factors{2} = {2, @(z,task) struct_poly(z, task, t3)};
model.factors{3} = {3, @(z,task) struct_poly(z, task, t3)};
```

The `transform` field allows the code block above to be rewritten as

```
model.transform{1} = {1:3, @struct_poly, {t1, t2, t3}};
```

Both code fragments result in exactly the same model, as the `transform` fields are converted to factors by the DSL parser `sdf_check`. If all points sets are the same, i.e., if `t1 = t2 = t3 = t`, this can be further simplified as

```
model.transform{1} = {1:3, @struct_poly, t};
```

The `transform` field should be a struct or a cell with different transformations. (In contrast to variables and factors, the names of the transformations do not have a meaning.) A transform should have the following structure:

```
% as a struct
model.transform.a = {varrefs, factrefs, func1, args1, func2, args2, ...};
% or as a cell
model.transform{1} = {varrefs, factrefs, func1, args1, func2, args2, ...};
```

The entry `varrefs` contains references to variables, either indexed or named. The entry `factrefs` contains the indices or names of the factors to be created as a result of the transformations `func1`, `func2`, etc. This entry is optional. If omitted, `factrefs = varrefs`. An arbitrary number of transformations `func1`, `func2` can be defined (including zero transformations). Each transformation can have an arbitrary number of parameters (including zero), separated by commas. For example, if `func1` does not require any parameter, and `func2` requires two, the code is as follows:

```
model.transform{1} = {varrefs, factrefs, func1, func2, param1, param2};
```

In the remaining part of this subsection, different restrictions are outlined.

- `varrefs`: variable references should be valid references to defined variables. If `factrefs` is omitted, the rules for `factrefs` also apply for `varrefs`.
- `factrefs`: factor references should be valid references to undefined factors. A number of additional rules apply:
  - All entries of the cell `factrefs` (after auto-wrapping) should be either integers or strings. Mixing named and indexed references is not allowed.
  - If `model.factors` is defined, and named factors are used, then named references should be used for `factrefs` as well. If indexed factors are used, indexed references should be used for `factrefs`.
  - All references in `factrefs` should point to non-existing factors to prevent overwriting previously defined factors. Factors can be defined previously using `model.factors` (this field is always processed before `model.transform`), or by previous transforms. This also means that all references in `factrefs` should be unique.
  - The number of references in `varrefs` should be equal to the number of references in `factrefs`.

  - If named references are used in `factrefs` the list of factors is created as follows. First the factors defined using the `factors` field are added. Then the factors created by the first transform `model.transform{1}` are added in the order defined by `factrefs`. Next the factors created by the second transform `model.transform{2}` are added, and so on. This order is important if indexed references are used in the `factorizations` field.

- `func`: each transformation should be a function handle to a valid transformation, e.g., `struct_xxx` in which `xxx` can be any transformation defined in Tensorlab. `func` should accept two $+ N$ arguments in which N is the number of extra parameters.
- `arg`: arguments, or parameters, of the transformations. Each created factor can have a different parameter value for a transformation. How the parameters are distributed to the factors depends on the type of the parameter:
  - if `arg` is not a cell, every created factor gets the same value for the parameter.
  - if `arg` is a cell of length 1, every created factor gets the same value for the parameter, being `arg{1}`. (Additional braces may be necessary depending on the intended result. For example, to pass a cell `{1,2}` it has to be wrapped: `{{1,2}}`.)
  - if `arg` is a cell with length `length(factrefs)` (or `length(varrefs)` if `factrefs` is omitted), the nth factor gets as parameter the value `arg{n}`.
  - if `arg` is a cell of a different length, an error is thrown. If one intends to have this cell as a parameter for all factors, `arg` should be replaced by `{arg}` such that the length 1 case applies.

12.7 The options field

The `model.options` field can be used to set additional options to allow or disallow certain language constructs. The field `model.options` is a struct with case-sensitive fields defining the option names. Currently, one option can be defined:

- `model.options.ImplicitFactors` can be used to allow (`true`) or disallow (`false`) the use of implicit factors. The default is `false`.

The options can be permanently set to a value chosen by the user. If a file `tensorlabsettings.mat` is on the search path, this file is loaded by `sdf_check`. All fields in the `sdfoptions` struct (if present in the file) are used automatically. Hence, if `sdfoptions.ImplicitFactors = true`, the default value for `model.options.ImplicitFactors` is set to true. If an option is defined in `model.options` and `sdfoptions`, the value provided by `model.options` takes precedence.

12.7.1 Implicit factors

If implicit factors are allowed, variables can be used directly in factorizations. For example, consider the following:

```
model = struct;
model.variables = {rand(4,3), rand(5,3)};
model.factors{2} = 2;
model.factorizations{1}.data = rand(4,4,5);
model.factorizations{1}.cpd  = {1,1,2};
```

If `sdf_check(model)` is called, an error is thrown as factor 1 is not defined (`model.factors{1}` is empty). If implicit factors are allowed, the DSL parser `sdf_check` attempts to create all missing factors by using variables with the same reference. Hence, in the example above, `model.variables{1}` can be used as a replacement for `model.factors{1}` . Therefore, `sdf_check` creates the new factor matrix 1 as variable 1, i.e., as if the following had been written:

```
model = struct;
model.variables = {rand(4,3), rand(5,3)};
model.factors{1} = 1;
model.factors{2} = 2;
model.factorizations{1}.data = rand(4,4,5);
model.factorizations{1}.cpd  = {1,1,2};
```

Note that implicit factors are only created in the last phase of the DSL parsing, i.e., after `model.factors` has been checked, and after all transforms in `model.transform` have been converted to factors. If named factors are used, the new implicitly defined factors are added after the already defined factors, in the order in which they occur in `model.factorizations` . If indexed variables and named factors are used, the newly created factor has as name `IFx` with `x` the index of the variable to which is referred in the factorization. Note that `IFx` is only created when needed, so it cannot be referenced before it is created. For example, the following model is invalid:

```
model = struct;
model.variables = cpd_rnd([10 11], 5);
model.factors.A = 1;
model.factorizations{1}.data = randn(10,11,11);
model.factorizations{1}.cpd  = {1, 'IF2', 'IF2'};
model.options.ImplicitFactors = true;
sdf_check(model, 'print') % throws unknown factor error
```

The factors `IF2` cannot be resolved because no factor or variable with the name `IF2` is known. The following model is valid though:

```
model = struct;
model.variables = cpd_rnd([10 11], 5);
model.factors.A = 1;
model.factorizations{1}.data = randn(10,11,11);
model.factorizations{1}.cpd  = {1, 2, 'IF2'};
model.options.ImplicitFactors = true;
sdf_check(model, 'print')
```

Now `sdf_check` creates a factor `IF2` when the reference `2` is encountered. When the reference `IF2` is processed, this factor already exists and can therefore be resolved.

CHAPTER

THIRTEEN

COMPLEX OPTIMIZATION

Optimization problems

An integral part of Tensorlab comprises optimization of real functions in complex variables [7, 8]. Tensorlab offers algorithms for complex optimization that solve unconstrained nonlinear optimization problems of the form

$$\underset{\mathbf{z} \in \mathbb{C}^n}{\text{minimize}} \quad f(\mathbf{z}, \overline{\mathbf{z}}), \tag{13.1}$$

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

$$\underset{\mathbf{z} \in \mathbb{C}^n}{\text{minimize}} \quad \frac{1}{2}\|\mathcal{F}(\mathbf{z}, \overline{\mathbf{z}})\|_F^2, \tag{13.2}$$

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

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

are also supported (cf. `nlsb_gndl`). Furthermore, when a real solution $\mathbf{z} \in \mathbb{R}^n$ 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 (13.1) and (13.2), we define the objective function and residual function as

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

and

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

respectively.

Note: 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.

13.1 Complex derivatives

13.1.1 Pen & paper differentiation

Scalar functions

To solve optimization problems of the form (13.1), 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 $\frac{\partial f_R}{\partial \mathbf{x}}$. For example, let

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

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

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

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

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

where $\mathbf{z} \in \mathbb{C}^n$ and an overline denotes the complex conjugate of its argument. It is clear that $f(\mathbf{x}) \equiv f_R(\mathbf{x})$ for $\mathbf{x} \in \mathbb{R}^n$ and hence f is a generalization of f_R to the complex plane. Because f is now a function of both $\mathbf{z}$ and $\overline{\mathbf{z}}$, the limit $\lim_{\mathbf{h}\to\mathbf{0}} \frac{f(\mathbf{z}+\mathbf{h})-f(\mathbf{z})}{\mathbf{h}}$ 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 $\mathbf{z}$, i.e., its Taylor series in $\mathbf{z}$ alone does not exist. However, it can be shown that f is analytic in $\mathbf{z}$ and $\overline{\mathbf{z}}$ as a whole, meaning f has a Taylor series in the variables $\mathbf{z}_C := \begin{bmatrix}\mathbf{z}^{\mathsf{T}} & \overline{\mathbf{z}}^{\mathsf{T}}\end{bmatrix}^{\mathsf{T}}$ with a complex gradient

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

where $\frac{\partial f}{\partial \mathbf{z}}$ and $\frac{\partial f}{\partial \overline{\mathbf{z}}}$ 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 $\mathbf{z}$ ($\overline{\mathbf{z}}$), while treating the variables $\overline{\mathbf{z}}$ ($\mathbf{z}$) as constant. For the example above, we have

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

First, we notice that $\overline{\frac{\partial f}{\partial \mathbf{z}}} = \frac{\partial f}{\partial \overline{\mathbf{z}}}$, which holds for any real-valued function $f(\mathbf{z}, \overline{\mathbf{z}})$. 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 $\mathbf{z} \in \mathbb{R}^n$ are equal to the real gradient $\frac{df_R}{d\mathbf{x}}$ up to a factor 2. Taking these two observations into account, the unconstrained nonlinear optimization algorithms in Tensorlab require only the *scaled conjugate cogradient*

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

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

Vector-valued functions

To solve optimization problems of the form (13.2), 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 $\frac{\partial \mathsf{vec}(\mathcal{F})}{\partial \mathbf{x}^\mathsf{T}}$. For example, let

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

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

$$\frac{d\mathsf{vec}\left(\mathcal{F}_R\right)}{d\mathbf{x}^\mathsf{T}} = \begin{bmatrix} \cos(\mathbf{x}^\mathsf{T}\mathbf{x}) \cdot (2\mathbf{x}^\mathsf{T}) \\ \mathbf{a}^\mathsf{T} \\ \mathbf{b}^\mathsf{T} \\ -\sin(\mathbf{x}^\mathsf{T}\mathbf{x}) \cdot (2\mathbf{x}^\mathsf{T}) \end{bmatrix}.$$

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

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

where $\mathbf{z} \in \mathbb{C}^n$ 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 $\frac{\partial \mathsf{vec}(\mathcal{F})}{\partial \mathbf{z}^\mathsf{T}}$ and $\frac{\partial \mathsf{vec}(\mathcal{F})}{\partial \overline{\mathbf{z}}^\mathsf{T}}$, respectively. For the example above, we have

$$\frac{\partial \mathsf{vec}\left(\mathcal{F}\right)}{\partial \mathbf{z}^\mathsf{T}} = \begin{bmatrix} \cos(\mathbf{z}^\mathsf{T}\mathbf{z}) \cdot (2\mathbf{z}^\mathsf{T}) \\ \mathbf{0}^\mathsf{T} \\ \mathbf{b}^\mathsf{T} \\ -\sin(\overline{\mathbf{z}}^\mathsf{T}\mathbf{z}) \cdot \overline{\mathbf{z}}^\mathsf{T} \end{bmatrix} \quad \text{and} \quad \frac{\partial \mathsf{vec}\left(\mathcal{F}\right)}{\partial \overline{\mathbf{z}}^\mathsf{T}} = \begin{bmatrix} \mathbf{0}^\mathsf{T} \\ \mathbf{a}^\mathsf{T} \\ \mathbf{0}^\mathsf{T} \\ -\sin(\overline{\mathbf{z}}^\mathsf{T}\mathbf{z}) \cdot \mathbf{z}^\mathsf{T} \end{bmatrix}.$$

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 (13.2) 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 $\mathbf{z}$, which implies $\frac{\partial \mathsf{vec}(\mathcal{F})}{\partial \overline{\mathbf{z}}^\mathsf{T}} \equiv 0$. Tensorlab offers nonlinear least squares solvers for both the general nonanalytic case and the latter analytic case.

Numerical differentiation

Real scalar functions (with the i-trick)

The real gradient can be numerically approximated with `deriv` using the so-called i-trick [9]. For example, define the scalar functions

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

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

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

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

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

where $\mathbf{a} \in \mathbb{R}^n$ and $\mathbf{z} \in \mathbb{C}^n$. Since f is real-valued and $\mathbf{z}$ 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 $\mathbf{a}$ is chosen as `ones(size(z))` and the relative error of the finite difference approximation of the scaled conjugate cogradient $\mathbf{g}(\mathbf{z})$ is computed:

```
% Approximate the scaled conjugate cogradient with finite differences
% and compute its relative error.
f = @(z)ones(size(z)).'*(z+conj(z))+log(z'*z);
g = @(z)2*ones(size(z))+2/(z'*z)*z;
z = randn(10,1)+randn(10,1)*1i;
relerr = norm(g(z)-deriv(f,z))/norm(g(z))
```

Note: 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

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

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

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

Set $\mathbf{a}$ equal to `ones(size(x))`. Since each entry in $\mathcal{F}_1(\mathbf{x})$ is real-valued and not a function of $\overline{\mathbf{x}}$, we can approximate the real Jacobian $J_1(\mathbf{x})$ 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 $\mathcal{F}(\mathbf{z})$ 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, $\mathcal{F}(\mathbf{z}) := \begin{bmatrix}\mathbf{z}^{\mathsf{H}}\mathbf{z} & \mathrm{Re}\{\mathbf{z}\}^{\mathsf{T}}\mathbf{z}\end{bmatrix}$ is not analytic in $\mathbf{z} \in \mathbb{C}^n$ because it depends on $\overline{\mathbf{z}}$, but is analytic when $\mathbf{z} \in \mathbb{R}^n$. An example of a function that is analytic for both real and complex $\mathbf{z}$ is the function $\mathcal{F}_1(\mathbf{z})$. The following two examples compute the relative error of the finite differences approximation of the Jacobian $J_1(\mathbf{x})$ in a real vector $\mathbf{x}$:

```
% 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 $J_1(\mathbf{z})$ in a complex vector $\mathbf{z}$:

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

Nonanalytic vector-valued functions (with finite differences)

In general, a tensor-valued function may be function of its argument and its complex conjugate. The matrix-valued function

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

where $\mathbf{a} \in \mathbb{R}^n$ 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

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

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

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

$$\begin{bmatrix}\dfrac{\partial \mathrm{vec}\,(\mathcal{F})}{\partial \mathbf{z}^{\mathsf{T}}} & \dfrac{\partial \mathrm{vec}\,(\mathcal{F})}{\partial \overline{\mathbf{z}}^{\mathsf{T}}}\end{bmatrix},$$

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 $\mathbf{a}$ 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))
```

13.2 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 (13.2) require the Jacobian of the residual function $\mathcal{F}(\mathbf{z},\overline{\mathbf{z}})$, which will be $\mathcal{F}_{\text{lyap}}(U,V)$ for the remainder of this section (cf. Section 13). 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 $\frac{\partial \text{vec}(\mathcal{F})}{\partial \mathbf{z}^{\mathsf{T}}}$, assuming $\mathcal{F}$ is analytic in $\mathbf{z}$. The second case, `'Jacobian-C'`, corresponds to approximating the complex Jacobian consisting of the Jacobian $\frac{\partial \text{vec}(\mathcal{F})}{\partial \mathbf{z}^{\mathsf{T}}}$ and conjugate Jacobian $\frac{\partial \text{vec}(\mathcal{F})}{\partial \overline{\mathbf{z}}^{\mathsf{T}}}$, where $\mathbf{z} \in \mathbb{C}^n$. 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 $\text{vec}\,(A \cdot X \cdot B) \equiv (B^{\mathsf{T}} \otimes A) \cdot \text{vec}\,(X)$, it is easy to verify that the Jacobian of $\mathcal{F}_{\text{lyap}}(U,V)$ is given by

$$\frac{\partial \mathrm{vec}\left(\mathcal{F}_{\mathrm{lyap}}\right)}{\partial\left[\mathrm{vec}\left(U\right)^{\mathsf{T}} \quad \mathrm{vec}\left(V\right)^{\mathsf{T}}\right]} = \left[\left(V^{\mathsf{T}} \otimes A\right) + \left(\overline{A} \cdot V^{\mathsf{T}} \otimes \mathbf{I}\right) \quad \left(\mathbf{I} \otimes A \cdot U\right) + \left(\overline{A} \otimes U\right)\right]. \tag{13.3}$$

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 $\mathbf{z}$. For problems for which the residual function $\mathcal{F}$ depends on both $\mathbf{z}$ and $\overline{\mathbf{z}}$, the complex Jacobian can be supplied with the field `dF.dzc`. See Section 13.1 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 $\mathbf{z}$ (i.e., it is not a function of $\overline{\mathbf{z}}$) 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 $J := \frac{\partial \mathrm{vec}(\mathcal{F})}{\partial \mathbf{z}^{\mathsf{T}}}$ 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 $\frac{\partial f}{\partial \mathbf{z}} \equiv J^{\mathsf{H}} \cdot \mathrm{vec}\left(\mathcal{F}\right)$ 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)
```

[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};
        gU = AHA*(U*(V*V'))+A'*(U*(V*A'*V'))+A'*(Q*V')+ ...
             A*(U*(V*A*V'))+U*(V*AHA*V')+Q*(A*V');
        gV = (U'*AHA*U)*V+((U'*A'*U)*V)*A'+(U'*A')*Q+ ...
             ((U'*A*U)*V)*A+((U'*U)*V)*AHA+(U'*Q)*A;
        g = {gU,gV};
    end

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

end
```

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

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

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

With an inexact step

The most computationally intensive part of most nonlinear least squares problems is computing the next descent direction, which involves inverting either the Jacobian $J := \frac{\partial \mathsf{vec}(\mathcal{F})}{\partial \mathbf{z}^{\mathsf{T}}}$ 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 $J \cdot \mathbf{x}$ and $J^{\mathsf{H}} \cdot \mathbf{y}$, or the single matrix-vector product $(J^{\mathsf{H}}J) \cdot \mathbf{x}$. 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 (13.3) 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 $\mathcal{F}(\mathbf{z}, \overline{\mathbf{z}})$, computing an inexact step requires matrix-vector products $J \cdot \mathbf{x}$, $J^{\mathsf{H}} \cdot \mathbf{y}$, $J_c \cdot \mathbf{x}$ and $J_c^{\mathsf{H}} \cdot \mathbf{y}$, where $J := \frac{\partial \mathrm{vec}(\mathcal{F})}{\partial \mathbf{z}^{\mathsf{T}}}$ and $J_c := \frac{\partial \mathrm{vec}(\mathcal{F})}{\partial \overline{\mathbf{z}}^{\mathsf{T}}}$ 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 (13.2) 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);
```

> **Note:** 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.

13.3 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 (13.1) require the (scaled conjugate co-)gradient of the objective function $f(\mathbf{z}, \overline{\mathbf{z}})$, which will be $f_{\text{lyap}}(\mathbf{z}, \overline{\mathbf{z}})$ for the remainder of this section (cf. Section 13). 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 $2\frac{\partial f_{\text{lyap}}}{\partial \overline{\mathbf{z}}}$, which coincides with the real gradient for $\mathbf{z} \in \mathbb{R}^n$. See Section *Complex derivatives* for more information on complex derivatives.

Note: 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 $2\frac{\partial f_{\text{lyap}}}{\partial \overline{\mathbf{z}}}$ where $\mathbf{z} := \left[\text{vec}\,(U)^{\mathsf{T}} \quad \text{vec}\,(V)^{\mathsf{T}}\right]^{\mathsf{T}}$.

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.

CHAPTER

FOURTEEN

GLOBAL MINIMIZATION OF BIVARIATE FUNCTIONS

14.1 Analytic bivariate polynomials

Consider the problem of minimizing a bivariate polynomial

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

or more generally, a rational function

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

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

$$\begin{aligned}f(x,y) &= 0\\ g(x,y) &= 0\end{aligned} \tag{14.1}$$

as the eigenvalues of a generalized eigenvalue problem [1, 2]. Stationary points of bivariate polynomials and rational functions may be computed with `polymin2` and `ratmin2`, respectively.

14.2 Polyanalytic univariate polynomials

Closely related is the problem of minimizing a polyanalytic univariate polynomial

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

or more generally, a rational function

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

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 13.1). The method `polysol2` can also solve systems of polyanalytic polynomials

$$\begin{aligned} f(z,\overline{z}) &= 0 \\ g(z,\overline{z}) &= 0. \end{aligned} \tag{14.2}$$

In fact, given a system of bivariate polynomials (14.1), `polysol2` will first convert it to the form (14.2) 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.x

14.2.1 Stationary points of polynomials and rational functions

14.3 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)`.

14.4 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 [3]

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

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 Fig. 14.1. 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))`.

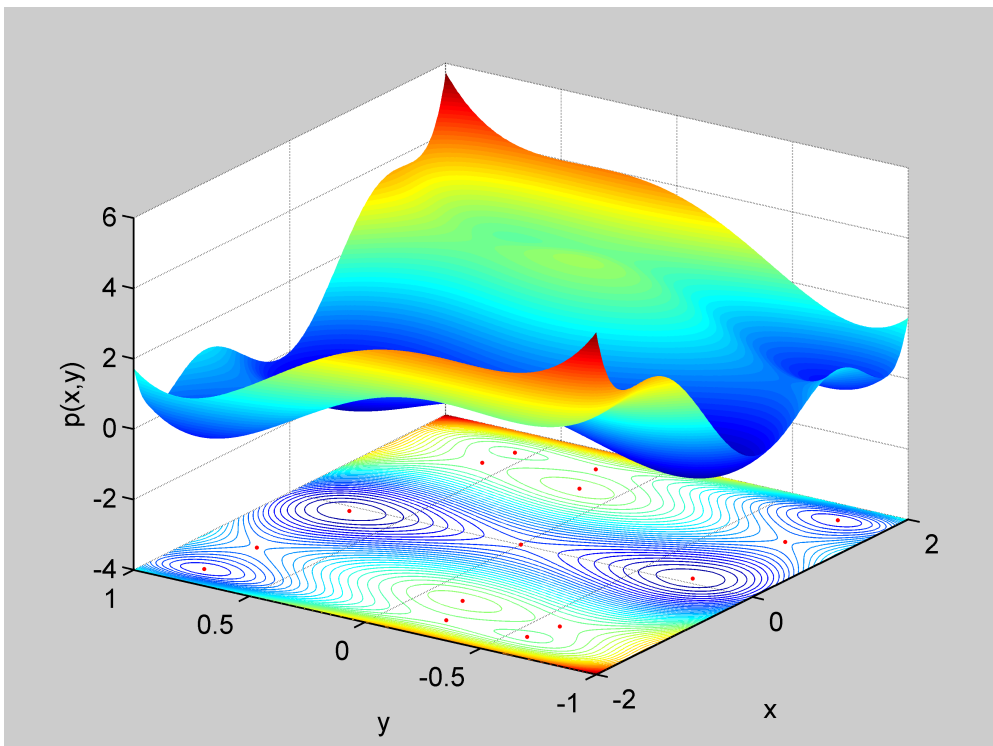

Fig. 14.1: The six-hump camel back function and its stationary points as red dots.

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))`.

14.5 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);
```

Note: 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`.

14.5.1 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 (14.1) or systems of polyanalytic univariate polynomials (14.2). In the case (14.1), `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 $\mathbb{C}^2$.

Note: 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 Fig. 14.2, 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.

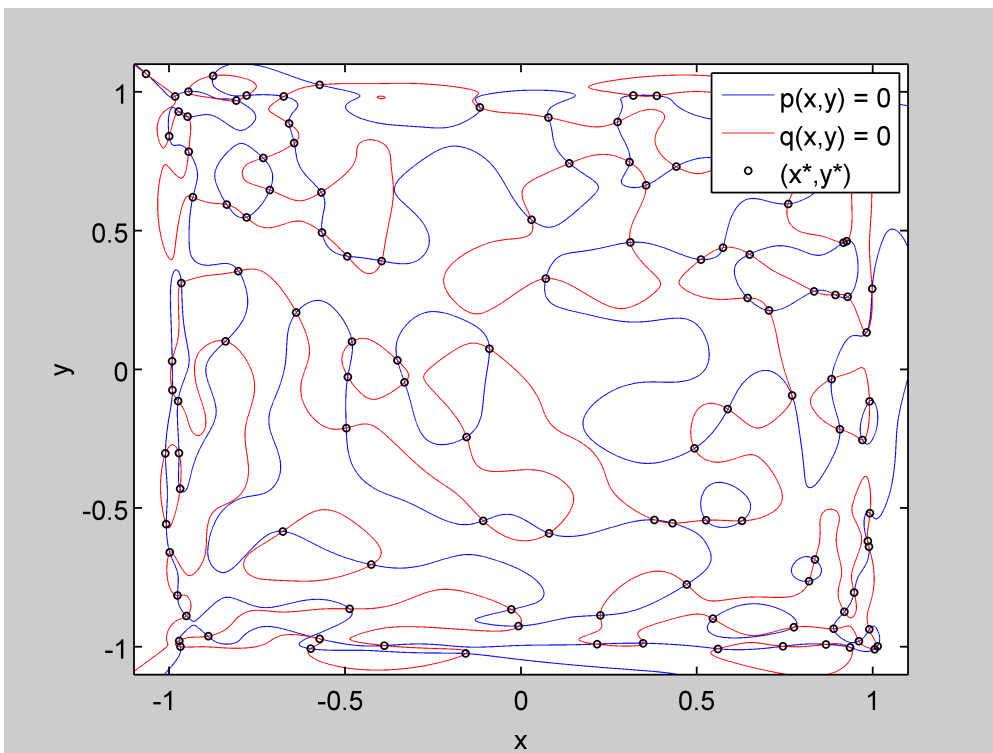

Fig. 14.2: A system of bivariate polynomials $p(x, y) = q(x, y) = 0$ of total degree 20.

CHAPTER

FIFTEEN

ACKNOWLEDGEMENTS

We would like to acknowledge Mariya Ishteva for contributing the `lmlra3_dgn` and `lmlra3_rtr` functions, and Nick Vannieuwenhoven for his contributions to the `mlsvd`, `tmprod` and `mtkrprod` functions. 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. We would like to thank Martijn Boussé for writing various unit tests and Frederik Van Eeghem for an update on the statistics routines, as well as for helping with the new website and for writing new demos.

Nico Vervliet is supported by an aspirant grant by the Research Foundation - Flanders (FWO).

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

Laurent Sorber was supported by a doctoral fellowship of the Flanders agency for Innovation by Science and Technology (IWT). Laurent was the first developer of Tensorlab and its main developer up to version 2.02.

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 in part by the Research Council KU Leuven under C1 projects c16/15/059-nD and CoE PFV/10/002 (OPTEC), in part by FWO under projects G.0830.14N and G.0881.14N, in part by the Belgian Federal Science Policy Office under Grant IUAP P7/19 (DYSCO II, Dynamical systems, control and optimization, 2012-2017), and in part by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Advanced Grant: BIOTENSORS (no. 339804). This work reflects only the authors' views and the Union is not liable for any use that may be made of the contained information.

REFERENCES

[1] L. Sorber, I. Domanov, M. Van Barel, and L. De Lathauwer. Exact line and plane search for tensor optimization. *Computational Optimization and Applications*, 63(1):121–142, 2015.

[2] L. Sorber, M. Van Barel, and L. De Lathauwer. Numerical solution of bivariate and polyanalytic polynomial systems. *SIAM Journal on Numerical Analysis*, 52(4):1551–1572, 2014.

[3] L. C. Dixon and G. P. Szegő. The global optimization problem: an introduction. *Towards global optimisation II*, pages 1–15, 1978.

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

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

[6] A. Cichocki, D. Mandic, A. H. Phan, C. Caiafa, G. Zhou, Q. Zhao, and L. De Lathauwer. Tensor decompositions for signal processing applications: From two-way to multiway component analysis. *IEEE Signal Processing Magazine*, 32(2):145–163, 2015.

[7] 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.

[8] T. Adali and P. J. Schreier. Optimization and estimation of complex-valued signals: theory and applications in filtering and blind source separation. *IEEE Signal Processing Magazine*, 5(31):112–128, 2014.

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

[10] 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.

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

[12] 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.

[13] 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.

[14] E. Sanchez and B. R. Kowalski. Tensorial resolution: a direct trilinear decomposition. *Journal of Chemometrics*, 4(1):29–45, 1990.

[15] N. Vervliet, O. Debals, L. Sorber, and L. De Lathauwer. Breaking the curse of dimensionality using decompositions of incomplete tensors: Tensor-based scientific computing in big data analysis. *Signal Processing Magazine, IEEE*, 31(5):71–79, September 2014.

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

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

[18] N. Vervliet and L. De Lathauwer. A randomized block sampling approach to canonical polyadic decomposition of large-scale tensors. *IEEE Journal of Selected Topics in Signal Processing*, 10(2):284–295, March 2016.

[19] L. De Lathauwer. A link between the canonical decomposition in multilinear algebra and simultaneous matrix diagonalization. *SIAM Journal on Matrix Analysis and Applications*, 28(3):642–666, 2006.

[20] L. De Lathauwer, B. De Moor, and J. Vandewalle. Computation of the canonical decomposition by means of a simultaneous generalized Schur decomposition. *SIAM Journal on Matrix Analysis and Applications*, 26(2):295–327, 2004.

[21] X. Liu and N. D. Sidiropoulos. Cramér–Rao lower bounds for low-rank decomposition of multidimensional arrays. *IEEE Transactions on Signal Processing*, 49(9):2074–2086, Sept. 2001.

[22] P. Tichavský, A.-H. Phan, and Z. Koldovský. Cramér–Rao-induced bounds for CANDECOMP/PARAFAC tensor decomposition. *IEEE Transactions on Signal Processing*, 61(8):1986–1997, Apr. 2013.

[23] I. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[24] N. Vervliet, O. Debals, L. Sorber, M. Van Barel, and L. De Lathauwer. Tensorlab 3.0. Mar. 2016. Available online. URL: http://www.tensorlab.net[1].

[25] 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.

[26] 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.

[27] 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.

[28] L. De Lathauwer. Blind separation of exponential polynomials and the decomposition of a tensor in rank-$(L_r, L_r, 1)$ terms. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1451–1474, 2011.

[29] L. De Lathauwer, B. De Moor, and J. Vandewalle. On the best rank-1 and rank-$(R_1, R_2, \ldots, R_n)$ approximation of higher-order tensors. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1324–1342, 2000.

[30] C. F. Caiafa and A. Cichocki. Generalizing the column–row matrix decomposition to multi-way arrays. *Linear Algebra and its Applications*, 433(3):557–573, 2010.

[31] N. Vannieuwenhoven, R. Vandebril, and K. Meerbergen. A new truncation strategy for the higher-order singular value decomposition. *SIAM Journal on Scientific Computing*, 34(2):A1027–A1052, 2012.

[32] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[33] P. M. Kroonenberg. *Applied multiway data analysis*. volume 702. Wiley-Interscience, 2008.

[34] M. Ishteva, L. De Lathauwer, P.-A. Absil, and S. Van Huffel. Differential-geometric Newton method for the best rank-(R_1, R_2, R_3) approximation of tensors. *Numerical Algorithms*, 51(2):179–194, June 2009.

[1] http://www.tensorlab.net

[35] M. Ishteva, P.-A. Absil, S. Van Huffel, and L. De Lathauwer. Best low multilinear rank approximation of higher-order tensors, based on the Riemannian trust-region scheme. *SIAM Journal on Matrix Analysis and Applications*, 32(1):115–135, 2011.

[36] L. Sorber, M. Van Barel, and L. De Lathauwer. Structured data fusion. *IEEE Journal of Selected Topics in Signal Processing*, 9(4):586–600, June 2015.

[37] J.-P. Royer, N. Thirion-Moreau, and P. Comon. Computing the polyadic decomposition of nonnegative third order tensors. *Signal Processing*, 91(9):2159–2171, 2011.

[38] R. Cabral Farias, J.E. Cohen, and P. Comon. Exploring multimodal data fusion through joint decompositions with flexible couplings. Working paper or preprint, May 2015. URL: https://hal.archives-ouvertes.fr/hal-01158082.

[39] H. Mohimani, M. Babaie-Zadeh, and C. Jutten. A fast approach for overcomplete sparse decomposition based on smoothed ℓ^0 norm. *IEEE Transactions on Signal Processing*, 57(1):289–301, Jan 2009.

[40] O. Debals and L. De Lathauwer. Stochastic and deterministic tensorization for blind signal separation. In *Latent Variable Analysis and Signal Separation*, volume 9237 of Lecture Notes in Computer Science, 3–13. Springer Berlin / Heidelberg, 2015.

[41] O. Debals, M. Van Barel, and L. De Lathauwer. Löwner-based blind signal separation of rational functions with applications. *IEEE Transactions on Signal Processing*, 64(8):1909–1918, April 2016.

[42] C. L. Nikias and A. P. Petropulu. Higher-order spectra analysis: A nonlinear signal processing framework. *PTR Prentice Hall, Englewood Cliffs, NJ*, 1993.