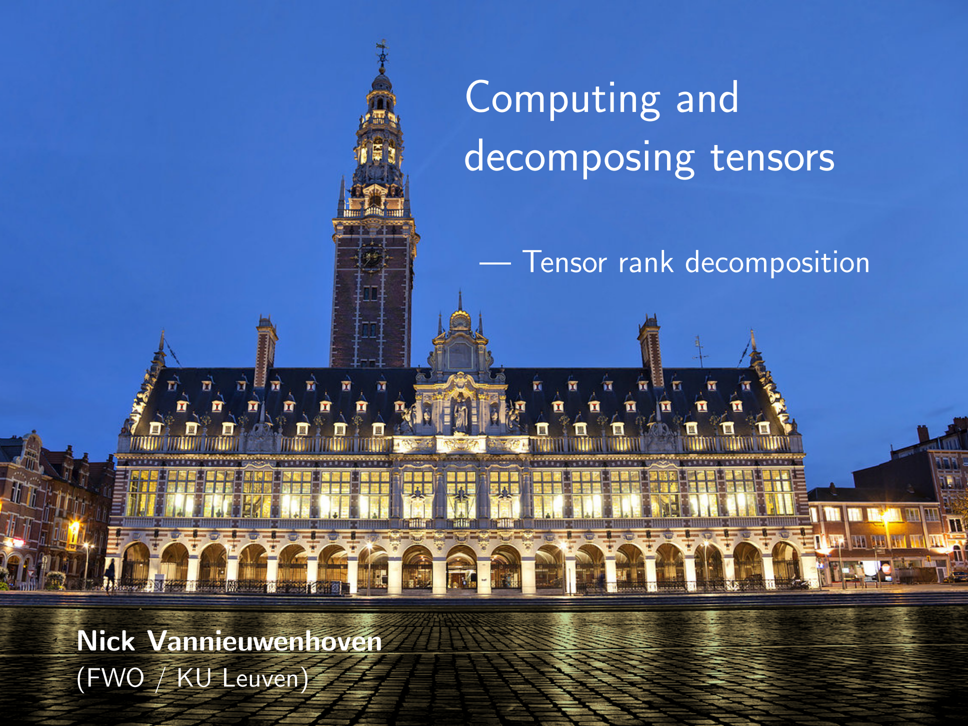


Computing and decomposing tensors

— Tensor rank decomposition

Nick Vannieuwenhoven
(FWO / KU Leuven)

Overview

Overview

Sensitivity

In numerical computations, the **sensitivity** of the output of a computation to **perturbations** at the input is extremely important.

We have already seen an example: computing singular values appears to behave nicely with respect to perturbations.

Consider the matrix

$$A = \frac{1}{177147} \begin{bmatrix} 88574 & 88574 & 2 \\ 88574 & 88574 & 2 \\ 2 & 2 & 177146 \end{bmatrix}$$

Computing the singular value decomposition $\widehat{U}\widehat{S}\widehat{V}^T$ of the floating-point representation $\widetilde{A}$ of A numerically using Matlab, we find $\|A - \widehat{U}\widehat{S}\widehat{V}^T\| \approx 5.66 \cdot 10^{-16}$.

The singular values are

numerical	exact
0.0000000000000000098..	0
0.9999830649121916	$0.99998306491219156971 3288... = 1 - 3^{-10}$
1.0000169350878808	$1.00001693508780843028 6711... = 1 + 3^{-10}$

In all cases, we found 16 correct digits of the exact solution.

However, when comparing the computed left singular vector corresponding to $\sigma_1 = 1 + 3^{-10}$ to the exact solution, we get

numerical	exact
0.577350269183747	$\frac{1}{\sqrt{3}}$
0.577350269183748	$\frac{1}{\sqrt{3}}$
0.577350269921281	$\frac{1}{\sqrt{3}}$

We have only recovered 11 digits correctly, even though the matrix $\widehat{U}\widehat{S}\widehat{V}^T$ contains at least 15 correct digits of each entry.

How is this possible?

We say that the problem of computing the singular values has a different sensitivity to perturbations than the computational problem of computing the left singular vectors.

Assuming the singular values are distinct, these problems can be modeled as functions

$$f_1 : \mathbb{F}^{m\times n} \to \mathbb{F}^{\min\{m,n\}}, \text{ respectively } f_2 : \mathbb{F}^{m\times n} \to \mathbb{F}^{m\times\min\{m,n\}}.$$

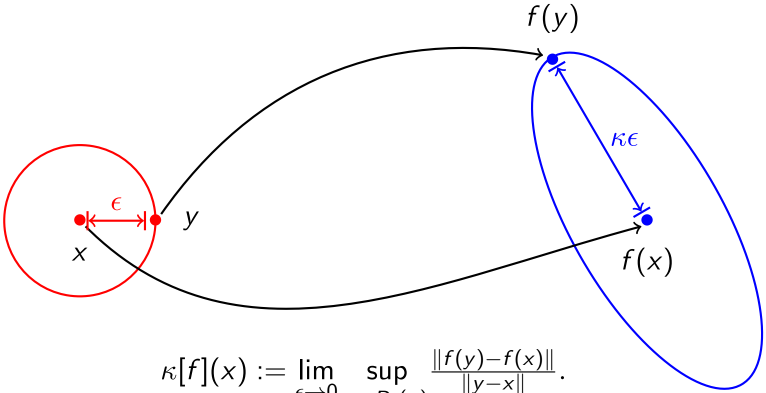
What we have observed above is that

$$0.4 \approx \frac{\|f_1(x) - f_1(x+\delta x)\|}{\|\delta x\|} \ll \frac{\|f_2(x) - f_2(x+\delta x)\|}{\|\delta x\|} \approx 800$$

at least $x = A$ and $\delta x = \widetilde{A} - A$ (with $\|\delta x\| \approx 5 \cdot 10^{-16}$).

Condition numbers

The **condition number** quantifies the **worst-case sensitivity** of f to perturbations of the input.



$$\kappa[f](x) := \lim_{\epsilon\to 0} \sup_{y\in B_\epsilon(x)} \frac{\|f(y)-f(x)\|}{\|y-x\|}.$$

If $f : \mathbb{F}^M \supset X \to Y \subset \mathbb{F}^N$ is a differentiable function, then the condition number is fully determined by the first-order approximation of f.

Indeed, in this case we have

$$f(\mathbf{x} + \boldsymbol{\Delta}) = f(\mathbf{x}) + J\boldsymbol{\Delta} + o(\|\boldsymbol{\Delta}\|),$$

where J is the Jacobian matrix containing all first-order partial derivatives. Then,

$$\begin{aligned}\kappa &= \lim_{\epsilon \to 0} \sup_{\|\boldsymbol{\Delta}\| \le \epsilon} \frac{\|f(\mathbf{x}) + J\boldsymbol{\Delta} + o(\|\boldsymbol{\Delta}\|) - f(\mathbf{x})\|}{\|\boldsymbol{\Delta}\|} \\ &= \max_{\|\boldsymbol{\Delta}\|=1} \frac{\|J\boldsymbol{\Delta}\|}{\|\boldsymbol{\Delta}\|} = \|J\|_2.\end{aligned}$$

More generally, for manifolds, we can apply Rice's (1966) **geometric framework of conditioning**:[1]

Proposition (Rice, 1966)

Let $\mathcal{X} \subset \mathbb{F}^m$ *be a manifold of inputs and* $\mathcal{Y} \subset \mathbb{F}^n$ *a manifold of outputs with* $\dim \mathcal{X} = \dim \mathcal{Y}$*. Then, the condition number of* $F : \mathcal{X} \to \mathcal{Y}$ *at* $x_0 \in \mathcal{X}$ *is*

$$\kappa[F](x_0) = \| \mathrm{d}_{x_0} F \| = \sup_{\|x\|=1} \| \mathrm{d}_{x_0} F(x) \|,$$

where $\mathrm{d}_{x_0} F : \mathrm{T}_{x_0}\mathcal{X} \to \mathrm{T}_{F(x_0)}\mathcal{Y}$ *is the* **derivative***.*

[1]See, e.g., Blum, Cucker, Shub, and Smale (1998) or Bürgisser and Cucker (2013) for a more modern treatment.

Overview

The tensor rank decomposition problem

In the remainder, $\mathcal{S}_1$ denotes the **smooth manifold** of rank-1 tensors in $\mathbb{R}^{n_1 \times \cdots \times n_d}$, called the **Segre manifold**.

To determine the condition number of computing a CPD, we analyze the **addition map**:

$$\begin{aligned}\Phi_r : \mathcal{S}_1 \times \cdots \times \mathcal{S}_1 &\to \mathbb{R}^{n_1 \times \cdots \times n_d}\\ (\mathcal{A}_1, \ldots, \mathcal{A}_r) &\mapsto \mathcal{A}_1 + \cdots + \mathcal{A}_r\end{aligned}$$

Note that the domain and codomain are **smooth manifolds**.

From differential geometry, we know that Φ_r is a **local diffeomorphism onto its image** if the **derivative** $\mathrm{d}_x \Phi_r$ is injective, i.e., has maximal rank equal to $r \dim \mathcal{S}_1$.

Let $\mathfrak{a} = (\mathcal{A}_1, \ldots, \mathcal{A}_r) \in (\mathcal{S}_1)^{\times r}$ and $\mathcal{A} = \Phi_r(\mathfrak{a})$. If $\mathrm{d}_{\Phi_r} \mathfrak{a}$ is injective, then by the **Inverse Function Theorem** there exists a local inverse function $\Phi_{\mathfrak{a}}^{-1} : \mathcal{E} \to \mathcal{F}$, where $\mathcal{E} \subset \mathrm{Im}(\Phi_r)$ is an open neighborhood of $\mathcal{A}$, $\mathcal{F} \subset (\mathcal{S}_1)^{\times r}$ is an open neighborhood of $\mathfrak{a}$, and

$$\Phi_r \circ \Phi_{\mathfrak{a}}^{-1} = \mathrm{Id}_{\mathcal{E}} \quad \text{and} \quad \Phi_{\mathfrak{a}}^{-1} \circ \Phi_r = \mathrm{Id}_{\mathcal{F}}\,.$$

In other words, $\Phi_{\mathfrak{a}}^{-1}$ computes one particular *ordered CPD* of $\mathcal{A}$.

This function has condition number

$$\kappa[\Phi_{\mathfrak{a}}^{-1}](\mathcal{A}) = \|\,\mathrm{d}_{\mathcal{A}} \Phi_{\mathfrak{a}}^{-1}\|_2 = \|(\mathrm{d}_{\mathfrak{a}} \Phi_r)^{-1}\|_2.$$

The derivative $\mathrm{d}_{\mathfrak{a}}\,\Phi$ is seen to be the map

$$\begin{aligned} \mathrm{d}_{\mathfrak{a}}\,\Phi : \mathrm{T}_{\mathcal{A}_1}\mathcal{S}_1 \times \cdots \times \mathrm{T}_{\mathcal{A}_r}\mathcal{S}_1 &\to \mathrm{T}_{\mathcal{A}}\mathbb{R}^{n_1\times\cdots\times n_d} \\ (\dot{\mathcal{A}}_1, \ldots, \dot{\mathcal{A}}_r) &\mapsto \dot{\mathcal{A}}_1 + \cdots + \dot{\mathcal{A}}_r. \end{aligned}$$

Hence, if U_i is an orthonormal basis of $\mathrm{T}_{\mathcal{A}_i}\mathcal{S}_1 \subset \mathrm{T}_{\mathcal{A}_i}\mathbb{R}^{n_1\times\cdots\times n_d}$, then the map is represented in coordinates as the matrix

$$U = \begin{bmatrix} U_1 & U_2 & \cdots & U_r \end{bmatrix} \in \mathbb{R}^{n_1\cdots n_d \times r \dim \mathcal{S}_1}$$

Summarizing, if we are given an ordered CPD $\mathfrak{a}$ of $\mathcal{A}$, then the condition number of computing this ordered CPD may be computed as the inverse of the smallest singular value of U.

From the above derivation, it also follows that

$$\kappa[\Phi_{\mathfrak{a}}^{-1}](\mathcal{A}) = \kappa[\Phi_{\mathfrak{a}'}^{-1}](\mathcal{A})$$

where

$$\mathfrak{a}' = (\mathcal{A}_{\pi_1}, \ldots, \mathcal{A}_{\pi_r}) \quad \text{for every permutation } \pi \in \mathfrak{S}_r.$$

Hence, the above gives us the condition number of the **tensor decomposition problem** at the CPD $\{\mathcal{A}_1, \ldots, \mathcal{A}_r\}$. We write

$$\kappa(\{\mathcal{A}_1, \ldots, \mathcal{A}_r\}) := \kappa[\Phi_{\mathfrak{a}}^{-1}](\mathcal{A}).$$

Interpretation

If

$$\mathcal{A} = \mathcal{A}_1 + \cdots + \mathcal{A}_r = \sum_{i=1}^{r} \mathbf{a}_i^1 \otimes \cdots \otimes \mathbf{a}_i^d$$

$$\mathcal{B} = \mathcal{B}_1 + \cdots + \mathcal{B}_r = \sum_{i=1}^{r} \mathbf{b}_i^1 \otimes \cdots \otimes \mathbf{b}_i^d$$

are tensors in $\mathbb{R}^{n_1 \times \cdots \times n_d}$, then for $\|\mathcal{A} - \mathcal{B}\|_F \approx 0$ we have the **asymptotically sharp bound**

$$\underbrace{\min_{\pi \in \mathfrak{S}_r} \sqrt{\sum_{i=1}^{r} \|\mathcal{A}_i - \mathcal{B}_{\pi_i}\|_F^2}}_{\text{forward error}} \lesssim \underbrace{\kappa(\{\mathcal{A}_1, \ldots, \mathcal{A}_r\})}_{\text{condition number}} \cdot \underbrace{\|\mathcal{A} - \mathcal{B}\|_F}_{\text{backward error}}$$

Overview

Overview

Pencil-based algorithms

A **pencil-based algorithm (PBA)** is an algorithm for computing the unique rank-r CPD of a tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ with $n_1 \geq n_2 \geq r$. Let

$$\mathcal{A} = \sum_{i=1}^{r} \mathbf{a}_i \otimes \mathbf{b}_i \otimes \mathbf{c}_i, \quad \text{where } \|\mathbf{a}_i\| = 1.$$

The matrices $A = [\mathbf{a}_i]$, $B = [\mathbf{b}_i]$ and $C = [\mathbf{c}_i]$ of $\mathcal{A}$ are called **factor matrices**.

Fix a matrix $Q \in \mathbb{R}^{n_3 \times 2}$ with two orthonormal columns. The key step in a PBA is the projection step

$$\mathcal{B} := (I, I, Q^T) \cdot \mathcal{A} = \sum_{i=1}^{r} \mathbf{a}_i \otimes \mathbf{b}_i \otimes \underbrace{Q^T \mathbf{c}_i}_{\mathbf{z}_i},$$

yielding an $n_1 \times n_2 \times 2$ tensor $\mathcal{B}$ whose factor matrices are (A, B, Z).

Thereafter, we compute the specific orthogonal Tucker decomposition

$$\mathcal{B} = (Q_1, Q_2, I) \cdot \mathcal{S} = \sum_{i=1}^{r} (Q_1 \mathbf{x}_i') \otimes (Q_2 \mathbf{y}_i') \otimes \mathbf{z}_i,$$

where $\mathbf{x}_i' = Q_1^T \mathbf{a}_i$ and $\mathbf{y}_i' = Q_2^T \mathbf{b}_i$.

Let $X = [\mathbf{x}_i'/\|\mathbf{x}_i'\|]$ and $Y = [\mathbf{y}_i'/\|\mathbf{y}_i'\|]$. Then, the core tensor $\mathcal{S} \in \mathbb{R}^{r\times r\times 2}$ has the following two 3-slices:

$$S_j = (I, I, \mathbf{e}_j^T) \cdot \mathcal{S} = \sum_{i=1}^{r} \lambda_{j,i} \mathbf{x}_i \otimes \mathbf{y}_i = X \operatorname{diag}(\boldsymbol{\lambda}_j) Y^T, \quad j = 1, 2,$$

where $\boldsymbol{\lambda}_j := [z_{j,i}\|\mathbf{x}_i'\|\|\mathbf{y}_i'\|]_{i=1}^r$ and $\mathbf{z}_i = [z_{j,i}]_{j=1}^2$.

If S_1 and S_2 are nonsingular, then we see that

$$S_1 S_2^{-1} = X \operatorname{diag}(\boldsymbol{\lambda}_1) \operatorname{diag}(\boldsymbol{\lambda}_2)^{-1} X^{-1}.$$

Thus X is the $r \times r$ matrix of eigenvectors of $S_1 S_2^{-1}$.

Recall that the factor matrices of $\mathcal{B}$ are (A, B, Z) and by definition $A = Q_1 X$. The rank-1 tensors can then be recovered by noting that

$$\mathcal{A}_{(1)} = \sum_{i=1}^{r} \mathbf{a}_i (\mathbf{b}_i \otimes \mathbf{c}_i)^T =: A(B \odot C)^T,$$

where $B \odot C := [\mathbf{b}_i \otimes \mathbf{c}_i]_{i=1}^{r} \in \mathbb{R}^{n_2 n_3 \times r}$.

Since A is left invertible, we get

$$A \odot (A^\dagger \mathcal{A}_{(1)})^T = A \odot (B \odot C) = [\mathbf{a}_i \otimes (\mathbf{b}_i \otimes \mathbf{c}_i)] \in \mathbb{R}^{n_1 n_2 n_3 \times r}$$

which is a matrix containing the rank-1 tensors of the CPD as columns.

```
**Algorithm 1:** Standard PBA Algorithm
input : A tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, $n_1 \geq n_2 \geq r$, of rank $r$.
output: Rank-1 tensors $\mathbf{a}_i \otimes \mathbf{b}_i \otimes \mathbf{c}_i$ of the CPD of $\mathcal{A}$.

Sample a random $n_3 \times 2$ matrix $Q$ with orthonormal columns.
$\mathcal{B} \leftarrow (I, I, Q^T) \cdot \mathcal{A}$;
Compute a rank-$(r, r, \min\{r, n_3\})$ HOSVD $(Q_1, Q_2, Q_3) \cdot \mathcal{S} = \mathcal{A}$;
$S_1 \leftarrow (I, I, \mathbf{e}_1^T) \cdot \mathcal{S}$;
$S_2 \leftarrow (I, I, \mathbf{e}_2^T) \cdot \mathcal{S}$;
Compute eigendecomposition $S_1 S_2^{-1} = XDX^{-1}$;
$A \leftarrow Q_1 X$;
$A \odot B \odot C \leftarrow A \odot (A^\dagger \mathcal{A}_{(1)})^T$;
```

Numerical issues

A variant of this algorithm is implemented in Tensorlab v3.0 as cpd_gevd. Let us perform an experiment with it.

We create the first tensor that comes to mind: a rank-25 random tensor $\mathcal{A}$ of size $25 \times 25 \times 25$:

```
>> Ut{1} = randn(25,25);
>> Ut{2} = randn(25,25);
>> Ut{3} = randn(25,25);
>> A = cpdgen(Ut);
```

Compute $\mathscr{A}$'s decomposition and compare its distance to the input decomposition, relative to the machine precision $\epsilon \approx 2 \cdot 10^{-16}$:

```
>> Ur = cpd_gevd(A, 25);
>> E = kr(Ut) - kr(Ur);
>> norm( E(:), 2 ) / eps
ans =
     8.6249e+04
```

This large number can arise because of a high condition number. However,

```
>> kappa = condition_number( Ut )
ans =
     2.134
```

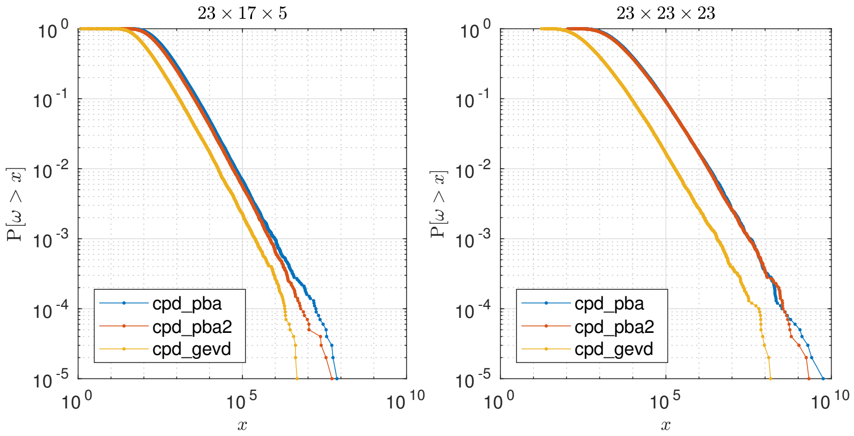
It thus appears that there is something wrong with the algorithm, even though it is a mathematically sound algorithm for computing low-rank CPDs. The reason is the following.

Theorem (Beltrán, Breiding, V, 2018)

Many algorithms based on a reduction to $\mathbb{R}^{n_1 \times n_2 \times 2}$ are **numerically unstable***: the forward error produced by the algorithm divided by the backward error is "much" larger than the condition number on an open set of inputs.*

These methods should be used with care as they do not necessarily yield the highest **attainable precision**.

The instability of the algorithm leads to an **excess factor** ω on top of the condition number of the computational problem:



Test with random rank-1 tuples.

Overview

Alternating least squares methods

The problems of computing a CPD of $\mathcal{A} \in \mathbb{F}^{n_1 \times \cdots \times n_d}$ and the problem of approximating $\mathcal{A}$ by a (low-rank) CPD can be formulated as an optimization problem. For example,

$$\min_{\substack{A_k \in \mathbb{F}^{n_k \times r} \\ k=1,\ldots,d}} \frac{1}{2} \| \mathcal{A} - (A_1 \odot A_2 \odot \cdots \odot A_d)\mathbf{1} \|_F^2.$$

One of the earliest methods devised specifically for this problem is the **alternating least squares** (ALS) scheme.

The ALS method is based on a reformulation of the **objective function**; it is equivalent to

$$\min_{\substack{A_j \in \mathbb{F}^{n_k \times r},\\ j=1,\ldots,d}} \frac{1}{2}\|\mathcal{A}_{(k)} - A_k(A_1 \odot \cdots \odot A_{k-1} \odot A_{k+1} \odot \cdots \odot A_d)^T\|^2.$$

for every $k = 1, 2, \ldots, d$.

The key observation is that if A_j, $j \neq k$, are fixed, then finding the optimal A_k is a linear problem! It is namely

$$\mathcal{A}_{(k)}((A_1 \odot \cdots \odot A_{k-1} \odot A_{k+1} \odot \cdots \odot A_d)^T)^\dagger.$$

The standard ALS method then solves the optimization problem by **cyclically fixing** all but one factor matrix A_k.

Algorithm 2: ALS method

```
input : A tensor $\mathcal{A} \in \mathbb{F}^{n_1 \times \cdots \times n_d}$.
input : A target rank $r$.
output: Factor matrices $(A_1, \ldots, A_d)$ of a CPD approximating $\mathcal{A}$.

Initialize factor matrices $A_k \in \mathbb{F}^{n_k \times r}$ (e.g., entries sampled i.i.d.
from $N(0,1)$, or truncated HOSVD);
while Not converged do
    for $k = 1, 2, \ldots, d$ do
        $\widehat{A}_k \leftarrow A_1 \odot \cdots \odot A_{k-1} \odot A_{k+1} \odot \cdots \odot A_d$;
        $A_k \leftarrow \mathcal{A}_{(k)} (\widehat{A}_k^\dagger)^T$;
    end
end
```

The ALS scheme produces a sequence of incrementally better approximations. However, the **convergence properties** of the ALS method are very poorly understood.

The scheme has accumulation points, but it is not known if they correspond to critical points of the objective function. That is, we do not know if the accumulation points of the ALS scheme correspond to points satisfying the first-order optimality conditions.

Uschmajew (2012) proved local convergence to critical points where the Hessian is positive semi-definite and of maximal rank.

Overview

Quasi-Newton optimization methods

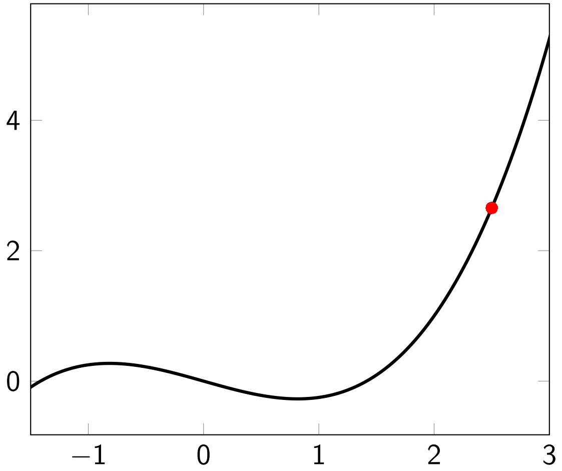
The optimization problem

$$\min_{\substack{A_k \in \mathbb{F}^{n_k \times r},\\ k=1,\ldots,d}} \frac{1}{2}\|\mathcal{A} - (A_1 \odot A_2 \odot \cdots \odot A_d)\mathbf{1}\|^2.$$

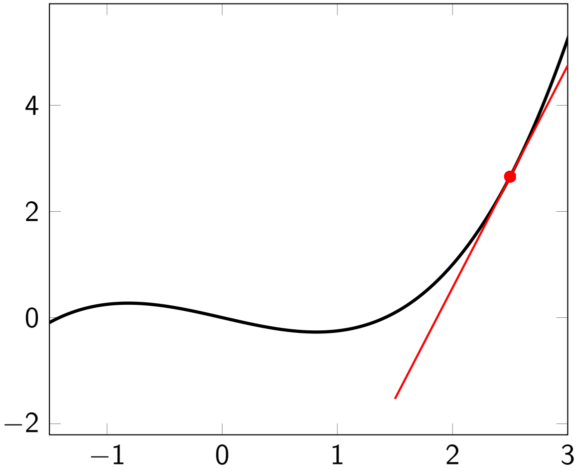
can be solved using any of the standard optimization methods, such as nonlinear conjugate gradient and **quasi–Newton** methods.

We discuss the **Gauss–Newton method** with **trust region** as globalization method.

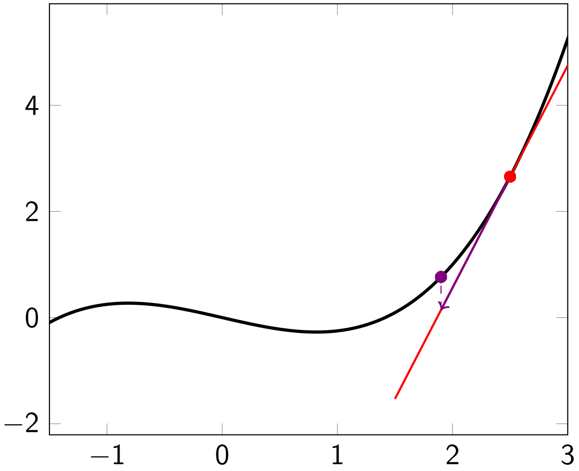
Recall that the univariate Newton method is based on the following idea.



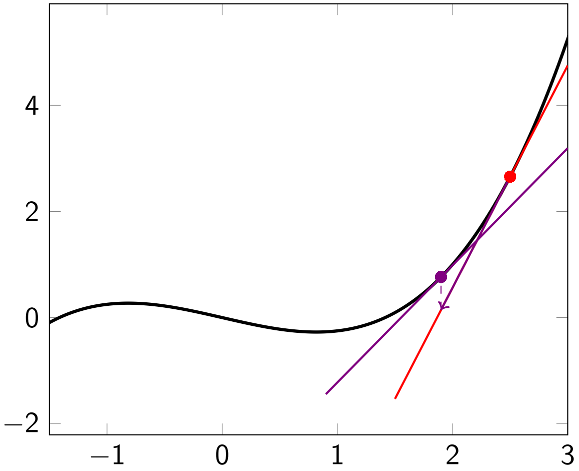
Recall that the univariate Newton method is based on the following idea.



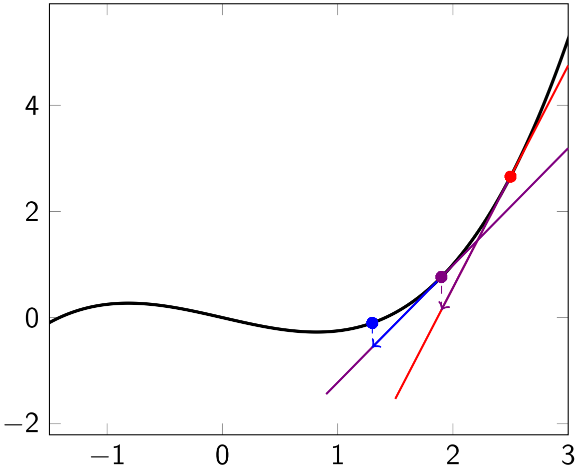
Recall that the univariate Newton method is based on the following idea.



Recall that the univariate Newton method is based on the following idea.



Recall that the univariate Newton method is based on the following idea.



Newton's method for minimizing a smooth real function $f(\mathbf{x})$ consists of generating a sequence of iterates $\mathbf{x}_0, \mathbf{x}_1, \ldots$ where $\mathbf{x}_{k+1}$ is the optimal solution of the local second-order Taylor series expansion of $f(x)$, namely

$$\begin{aligned} f(\mathbf{x}_k + \mathbf{p}) &\approx m_{\mathbf{x}_k}(\mathbf{p}) \\ &:= f(\mathbf{x}_k) + \mathbf{p}^T \mathbf{g}_k + \frac{1}{2} \mathbf{p}^T H_k \mathbf{p} \end{aligned}$$

where

- $\mathbf{g}_k := \nabla f(\mathbf{x}_k)$ is the **gradient** of f at $\mathbf{x}_k$, and
- $H_k := \nabla^2 f(\mathbf{x}_k)$ is the symmetric **Hessian** matrix of f at $\mathbf{x}_k$.

Recall from calculus that the gradient $f : \mathbb{R}^n \to \mathbb{R}$ with coordinates x_i on $\mathbb{R}^n$ is

$$\nabla f(\mathbf{y}) := \begin{bmatrix} \frac{\partial}{\partial x_1} f(\mathbf{y}) \\ \frac{\partial}{\partial x_2} f(\mathbf{y}) \\ \vdots \\ \frac{\partial}{\partial x_n} f(\mathbf{y}) \end{bmatrix}.$$

The Hessian matrix is

$$\nabla^2 f(\mathbf{y}) := \begin{bmatrix} \frac{\partial^2}{\partial x_1 \partial x_1} f(\mathbf{y}) & \frac{\partial^2}{\partial x_1 \partial x_2} f(\mathbf{y}) & \cdots & \frac{\partial^2}{\partial x_1 \partial x_n} f(\mathbf{y}) \\ \frac{\partial^2}{\partial x_2 \partial x_1} f(\mathbf{y}) & \frac{\partial^2}{\partial x_2 \partial x_2} f(\mathbf{y}) & \cdots & \frac{\partial^2}{\partial x_2 \partial x_n} f(\mathbf{y}) \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2}{\partial x_n \partial x_1} f(\mathbf{y}) & \frac{\partial^2}{\partial x_n \partial x_2} f(\mathbf{y}) & \cdots & \frac{\partial^2}{\partial x_n \partial x_n} f(\mathbf{y}) \end{bmatrix}.$$

The optimal search direction $\mathbf{p}$ according to the second-order model should satisfy the first-order optimality conditions. That is,

$$0 = \nabla m_{\mathbf{x}_k}(\mathbf{p}) = \mathbf{g}_k + H_k \mathbf{p}$$

Hence,

$$\mathbf{p} = -H_k^{-1} \mathbf{g}_k;$$

this is called the **Newton search direction**.

Hence, the basic Newton method is obtained.

```
Algorithm 3: Newton's method
input  : An objective function $f$.
input  : A starting point $\mathbf{x}_0 \in \mathbb{R}^n$.
output: A critical point $\mathbf{x}_\star$ of the objective function $f$

$k \leftarrow 0$;
while Not converged do
    Compute the gradient $\mathbf{g}_k = \nabla f(\mathbf{x}_k)$;
    Compute the Hessian $H_k = \nabla^2 f(\mathbf{x}_k)$;
    $\mathbf{p}_k \leftarrow -H_k^{-1}\mathbf{g}_k$;
    $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \mathbf{p}_k$;
    $k \leftarrow k + 1$;
end
```

Newton's method is *locally* quadratically convergent: if the initial iterate $\mathbf{x}_0$ is sufficiently close to the solution, then

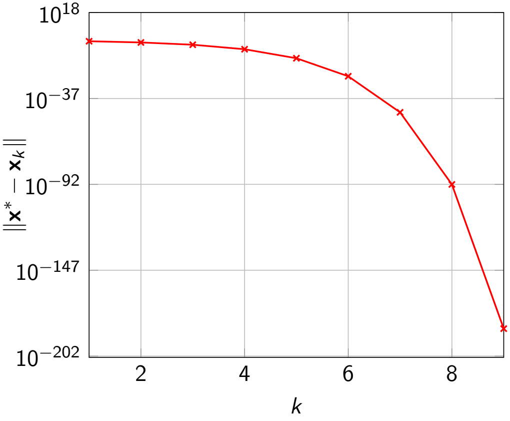
$$\|\mathbf{x}^* - \mathbf{x}_{k+1}\| = \mathcal{O}\big(\|\mathbf{x}^* - \mathbf{x}_k\|^2\big).$$

This means that close to the solution the number of correct digits **doubles** every step.

For example, Newton's method applied to

$$f(x,y) = \left\| \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} - \begin{bmatrix} 1 & y \\ x & xy \end{bmatrix} \right\|^2$$

converges to the root at $(2,2)$ starting from $\mathbf{x}_0 = (3,3)$.



While Newton's method has great local convergence properties, the plain version is not suitable because:

1. it has no guaranteed **global convergence**, and
2. the Hessian matrix can be difficult to compute.

These problems are addressed respectively by

1. incorporating a **trust region** scheme, and
2. using a cheap approximation of the Hessian matrix.

The Gauss–Newton Hessian approximation

A cheap approximation of the Hessian matrix is available for **nonlinear least squares** problems. In this case, the objective function takes the form

$$f(\mathbf{x}) = \frac{1}{2}\|F(\mathbf{x})\|^2 = \frac{1}{2}\langle F(\mathbf{x}), F(\mathbf{x})\rangle, \quad \text{where } F : \mathbb{R}^n \to \mathbb{R}^m.$$

The gradient of a least-squares objective function f at $\mathbf{x}$ is

$$\nabla f(\mathbf{x}) = \big(J_F(\mathbf{x})\big)^T F(\mathbf{x}),$$

where $J_F(\mathbf{x}) \in \mathbb{R}^{m\times n}$ is the **Jacobian matrix** of F. That is, if $F_k(\mathbf{x})$ denotes the kth component function of F, then

$$J_F(\mathbf{x}) := \begin{bmatrix} \frac{\partial}{\partial x_1} F_1(\mathbf{x}) & \frac{\partial}{\partial x_2} F_1(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} F_1(\mathbf{x}) \\ \frac{\partial}{\partial x_1} F_2(\mathbf{x}) & \frac{\partial}{\partial x_2} F_2(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} F_2(\mathbf{x}) \\ \vdots & \vdots & & \vdots \\ \frac{\partial}{\partial x_1} F_m(\mathbf{x}) & \frac{\partial}{\partial x_2} F_m(\mathbf{x}) & \cdots & \frac{\partial}{\partial x_n} F_m(\mathbf{x}) \end{bmatrix}.$$

The Hessian matrix of f is

$$\nabla^2 f(\mathbf{x}) = \big(J_F(\mathbf{x})\big)^T \big(J_F(\mathbf{x})\big) + \langle \mathrm{d}J_F(\mathbf{x}), F(\mathbf{x})\rangle.$$

Near a solution, we hope to have $F(\mathbf{x}^*) \approx 0$, so that the last term often has a negligible contribution.

This reasoning leads to the **Gauss–Newton** approximation

$$\big(J_F(\mathbf{x})\big)^T \big(J_F(\mathbf{x})\big) \approx \nabla^2 f(\mathbf{x}).$$

Replacing the Hessian with the Gauss–Newton approximation yields local linear convergence. If $f(\mathbf{x}^*) = 0$ at a solution, then the local convergence is quadratic.

Trust region globalization

The method sketched thus far has no global convergence guarantees. Furthermore, the Gauss–Newton approximation of the Hessian could be very ill-conditioned resulting in large updates.

The **trust region globalization** scheme can solve both of these problems. Let $J_k := J_F(\mathbf{x}_k)$. The idea is to trust the local second-order model at $\mathbf{x}_k$,

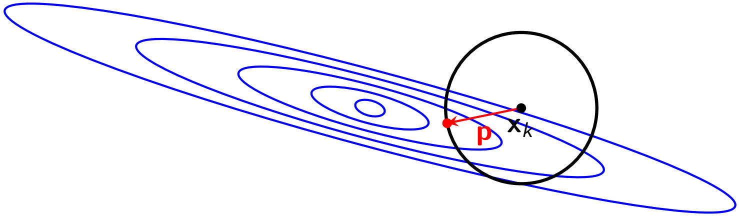
$$m_{\mathbf{x}_k}(\mathbf{p}) = f(\mathbf{x}_k) + \mathbf{p}^T \mathbf{g}_k + \frac{1}{2}\mathbf{p}^T J_k^T J_k \mathbf{p},$$

only in a small neighborhood around $\mathbf{x}_k$.

Instead of taking the unconstrained minimizer of $m(\mathbf{x}_k + \mathbf{p})$, a trust region method solves the **trust region subproblem**:

$$\min_{\mathbf{p}\in\mathbb{R}^n} m_{\mathbf{x}_k}(\mathbf{p}) \quad \text{subject to } \|\mathbf{p}\| \le \Delta_k,$$

where $\Delta_k > 0$ is the **trust region radius**.



The trust region radius is modified in every step according to the following scheme. Let the computed update direction be $\mathbf{p}_k$, with $\|\mathbf{p}_k\| \le \Delta_k$.

The **trustworthiness** of the second order model is defined as

$$\rho_k = \frac{f(\mathbf{x}_k) - f(\mathbf{x}_k + \mathbf{p}_k)}{m_{\mathbf{x}_k}(0) - m_{\mathbf{x}_k}(\mathbf{p}_k)}.$$

If the trustworthiness $\rho_k > 0$ is very high (e.g., $\rho_k \ge 0.75$) and if in addition $\|\mathbf{p}_k\| \approx \Delta_k$, then the trust region radius is increased (e.g, $\Delta_{k+1} = 2\Delta_k$). On the other hand, if $\rho_k \le \beta$ is very low (e.g., $\rho_k \le 0.25$), then the trust region radius is decreased (e.g., $\Delta_{k+1} = \Delta_k/4$).

The trustworthiness ρ_k is also used to decide whether or not to accept a step in the direction of the computed $\mathbf{p}_k$. If $\rho_k \le \gamma \le \beta$ is very small (e.g., $\rho_k \le 0.1$), then the search direction $\mathbf{p}_k$ is rejected. Otherwise, $\mathbf{p}_k$ is accepted as a good direction, and we set $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{p}_k$.

For (approximately) solving the trust region subproblem

$$\min_{\mathbf{p}\in\mathbb{R}^n} m_{\mathbf{x}_k}(\mathbf{p}) \quad \text{subject to } \|\mathbf{p}\| \leq \Delta_k,$$

one can exploit the following fact. If the unconstrained minimizer

$$\mathbf{p}_k^* = -(J_k^T J_k)^{-1}\mathbf{g}_k = (J_k^T J_k)^{-1} J_k^T \mathbf{r}_k = J_k^\dagger \mathbf{r}_k$$

where $\mathbf{r}_k := F(\mathbf{x}_k)$, falls within the trust region, $\|\mathbf{p}_k^*\| \leq \Delta_k$ then this is the optimal solution of the trust region subproblem.

Otherwise, there exists a $\lambda > 0$ such that the optimal solution $\mathbf{p}_k^*$ satisfies

$$(J_k^T J_k + \lambda I)\mathbf{p}_k^* = -J_k^T \mathbf{x}_r$$

with $\|\mathbf{p}_k^*\| = \Delta_k$. Nocedal and Wright (2006) discuss strategies for finding λ.

Several variations of this quasi–Newton method with trust region globalization are implemented in Tensorlab as `cpd_nls`. See Sorber, Van Barel, De Lathauwer (2013) for details.

Overview

Riemannian optimization

An alternative way to formulate the approximation of a tensor by a low-rank CPD consists of optimizing over a product of Segre manifolds:

$$\min_{(\mathcal{A}_1,\ldots,\mathcal{A}_r)\in(\mathcal{S}_1\times\cdots\times\mathcal{S}_1)} \|\Phi_r(\mathcal{A}_1,\ldots,\mathcal{A}_r) - \mathcal{A}\|_F.$$

This is an optimization of

1. a **differentiable** function,
2. over a **smooth manifold**.

These problems are studied in **Riemannian optimization**.

In general, if $\mathcal{M} \subset \mathbb{R}^N$ is an m-dimensional smooth manifold and $F : \mathcal{M} \to \mathbb{R}^n$ a smooth function, then

$$\min_{x\in\mathcal{M}} \frac{1}{2}\|F(x)\|^2$$

is a Riemannian optimization problem that can be solved by, e.g., a **Riemannian Gauss–Newton method**; see Absil, Mahoney, Sepulchre (2008).

In a RGN method, the objective function

$$f(x) = \frac{1}{2}\|\Phi_r(x) - \mathcal{A}\|^2$$

is locally approximated at $\mathfrak{a} \in \mathcal{S}_1^{\times r}$ by the quadratic model

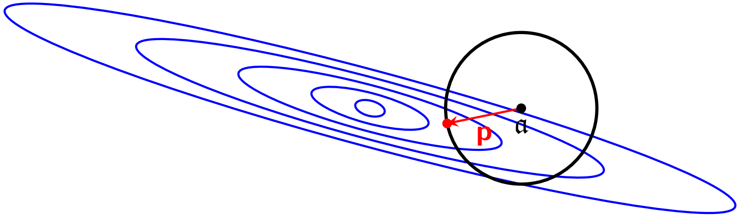
$$m_{\mathfrak{a}}(\mathbf{t}) := f(\mathfrak{a}) + \langle \mathrm{d}_{\mathfrak{a}} f, \mathbf{t}\rangle + \frac{1}{2}\langle \mathbf{t}, (\mathrm{d}_{\mathfrak{p}} \Phi_r{}^* \circ \mathrm{d}_{\mathfrak{a}} \Phi_r)(\mathbf{t})\rangle,$$

where

- $H_{\mathfrak{a}} := \mathrm{d}_{\mathfrak{a}} \Phi_r{}^* \circ \mathrm{d}_{\mathfrak{a}} \Phi_r$ is the **GN Hessian approximation**, and
- $\langle \cdot, \cdot \rangle$ is the inner product inherited from the ambient $\mathbb{R}^N$.

Note that the domain of $m_{\mathfrak{a}}$ is now $\mathrm{T}_{\mathfrak{a}}\mathcal{S}_1^{\times r}$.

As before, the RGN method with trust region considers the model to be accurate only in a radius Δ about $\mathfrak{a}$.



The **trust region subproblem** (TRS) is

$$\min_{\mathbf{t}\in \mathrm{T}_{\mathfrak{a}}\mathcal{S}_1^{\times r}} m_{\mathfrak{a}}(\mathbf{t}) \quad \text{subject to } \|\mathbf{t}\| \le \Delta,$$

whose solution $\mathbf{p} \in \mathrm{T}_{\mathfrak{a}}\mathcal{S}_1^{\times r}$ yields the next search direction.

In Breiding, V (2018), the TRS is solved by combining a standard **dogleg step** with a **hot restarting** scheme.

Let $\mathbf{g}_{\mathfrak{a}}$ be the coordinate representation of $\mathrm{d}_{\mathfrak{a}} f$, and let $H_{\mathfrak{a}}$ be the matrix of $\mathrm{d}_{\mathfrak{a}} \Phi_r{}^* \circ \mathrm{d}_{\mathfrak{a}} \Phi_r$. The **dogleg step** approximates the solution $\mathbf{p}$ of the TRS by

$$\widehat{\mathbf{p}} = \begin{cases} \mathbf{p}_{\mathrm{N}} = -H_{\mathfrak{a}}^{\dagger} \mathbf{g}_{\mathfrak{a}} & \text{if } \|\mathbf{p}_{\mathrm{N}}\| \leq \Delta \\ \mathbf{p}_{\mathrm{C}} = -\frac{\mathbf{g}_{\mathfrak{a}}^T H_{\mathfrak{a}} \mathbf{g}_{\mathfrak{a}}}{\mathbf{g}_{\mathfrak{a}}^T \mathbf{g}_{\mathfrak{a}}} \mathbf{g}_{\mathfrak{a}} & \text{if } \|\mathbf{p}_{\mathrm{N}}\| > \Delta \text{ and } \|\mathbf{p}_{\mathrm{C}}\| \geq \Delta \\ \mathbf{p}_{\mathrm{I}} := \mathbf{p}_{\mathrm{C}} + (\tau - 1)(\mathbf{p}_{\mathrm{N}} - \mathbf{p}_{\mathrm{C}}) & \text{s.t. } \|\mathbf{p}_{\mathrm{I}}\| = \Delta, \text{ otherwise} \end{cases}.$$

where $1 \leq \tau \leq 2$ is the solution of $\|\mathbf{p}_{\mathrm{C}} + (\tau - 1)(\mathbf{p}_{\mathrm{N}} - \mathbf{p}_{\mathrm{C}})\|^2 = \Delta^2$.

The Newton direction

$$\mathbf{p}_{\mathrm{N}} = -H_{\mathfrak{a}}^{\dagger}\mathbf{g}_{\mathfrak{a}}.$$

is vital to the dogleg step. Unfortunately, the $H_{\mathfrak{a}} = \mathrm{d}_{\mathfrak{a}}\,\Phi_r{}^{*} \circ \mathrm{d}_{\mathfrak{a}}\,\Phi_r$ can be close to a singular matrix. In fact,

$$\sqrt{\|H_{\mathfrak{a}}^{-1}\|_2} = \frac{1}{\varsigma_m(\mathrm{d}_{\mathfrak{a}}\,\Phi_r)} =: \kappa(\mathfrak{a}),$$

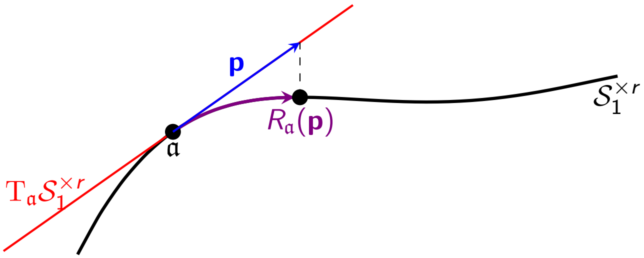
where $m = \dim \mathcal{S}_1^{\times r}$.

$H_{\mathfrak{a}}$ is ill-conditioned if and only if the CPD is ill-conditioned at $\mathfrak{a}$.

Whenever $H_{\mathfrak{a}}$ is close to a singular matrix we suggest to apply **random perturbations** to the current decomposition $\mathfrak{a}$ until $H_{\mathfrak{a}}$ is sufficiently well-behaved. We call this **hot restarting**.

Retraction

We need to advance from $\mathfrak{a} \in \mathcal{S}_1^{\times r}$ to $\mathfrak{a}' \in \mathcal{S}_1^{\times r}$, along the direction $\mathbf{p}$. However, while $\mathfrak{a} + \mathbf{p} \in \mathrm{T}_{\mathfrak{a}}\mathcal{S}_1^{\times r}$, this point does not lie in $\mathcal{S}_1^{\times r}$!


$\mathbf{p}$
$R_{\mathfrak{a}}(\mathbf{p})$
$\mathcal{S}_1^{\times r}$
$\mathfrak{a}$
$\mathrm{T}_{\mathfrak{a}}\mathcal{S}_1^{\times r}$

We need a **retraction operator** (Absil, Mahoney, Sepulchre, 2008) for smoothly mapping a neighborhood of $\mathbf{0} \in \mathrm{T}_{\mathfrak{a}}\mathcal{S}_1^{\times r}$ back to $\mathcal{S}_1^{\times r}$.

Given a retraction operator R' for $\mathcal{S}_1$, a retraction operator R for the product manifold $\mathcal{S}_1^{\times r} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_1$ at $\mathfrak{a} = (\mathcal{A}_1, \ldots, \mathcal{A}_r)$ is

$$R_{\mathfrak{a}}(\cdot) := (R'_{\mathcal{A}_1} \times R'_{\mathcal{A}_2} \times \cdots \times R'_{\mathcal{A}_r})(\cdot),$$

which is called the **product retraction**.

Some known retraction operators for $\mathcal{S}_1$ are

- the rank-$(1, \ldots, 1)$ T-HOSVD, and
- the rank-$(1, \ldots, 1)$ ST-HOSVD,

both essentially proved by (Kressner, Steinlechner, Vandereycken, 2014).

RGN with trust region method:

```
S1. Choose random initial points $\mathcal{A}_i \in \mathcal{S}_1$.
S2. Let $\mathfrak{a}^{(1)} \leftarrow (\mathcal{A}_1, \ldots, \mathcal{A}_r)$, and set $k \leftarrow 0$.
S3. Choose a trust region radius $\Delta > 0$.
S4. While not converged, do:
    S4.1. Solve the trust region subproblem, resulting in $\mathbf{p}_k \in \mathrm{T}_{\mathfrak{a}} \mathcal{S}_1^{\times r}$.
    S4.2. Compute the tentative next iterate $\mathfrak{a}^{(k+1)} \leftarrow R_{\mathfrak{a}^{(k)}}(\mathbf{p}_k)$ via a
          retraction in the direction of $\mathbf{p}_k$ from $\mathfrak{p}^{(k)}$.
    S4.3. Accept or reject the next iterate. If the former, increment $k$.
    S4.4. Update the trust region radius $\Delta$.
```

The details can be found in Breiding, V (2018).

Numerical experiments

We compare the RGN method of (Breiding, V, 2018) with some state-of-the-art nonlinear least squares solvers in Tensorlab v3.0 (Vervliet *et al.*, 2016), namely `nls_lm` and `nls_gndl`, both with the `LargeScale` option turned off and on.

We consider parameterized[2] tensors in $\mathbb{R}^{n_1 \times n_2 \times n_3}$ with varying condition numbers. There are three parameters:

1. $c \in [0, 1]$ regulates the "colinearity" of the factor matrices
2. $s \geq 1$ regulates the scaling, and
3. r is the rank.

Typically,

1. increasing c increases the geometric condition number.
2. increasing s increases the classic condition number.
3. increasing r decreases the probability of finding a decomposition.

[2]See the afternotes for the precise construction.

The true rank-r tensor is then

$$\mathcal{A} = \sum_{i=1}^{r} \mathbf{a}_i^1 \otimes \mathbf{a}_i^2 \otimes \mathbf{a}_i^3.$$

Finally, we normalize the tensor and add random Gaussian noise $\mathcal{E} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ of magnitude τ:

$$\mathcal{B} = \frac{\mathcal{A}}{\|\mathcal{A}\|_F} + \tau \frac{\mathcal{E}}{\|\mathcal{E}\|_F}.$$

The tensor $\mathcal{B}$ is the one we would like to approximate by a tensor of rank r.

We will choose k random starting points and then apply each of the methods to each of the starting points.

The key **performance criterion** (on a single processor) is the **expected time to success** (TTS).

Let

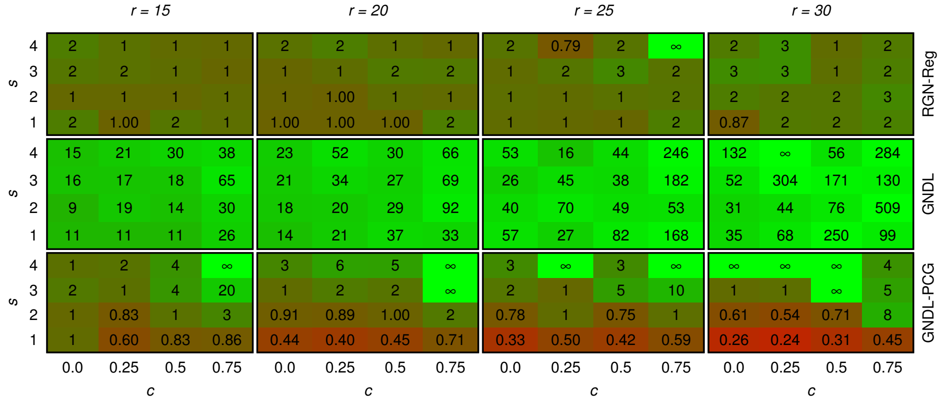
1. the probability of success be p_S,
2. the probability of failure be $p_F = 1 - p_S$,
3. a successful decomposition take m_S seconds, and
4. a failed decomposition take m_F seconds.

Then, the expected time to a first success is

$$\mathrm{E}[\mathrm{TTS}] = \sum_{k=0}^{\infty} p_F^{k-1} p_S (m_S + (k-1) m_F) = \frac{p_S m_S + p_F m_F}{p_S}.$$

Speedup of RGN-HR

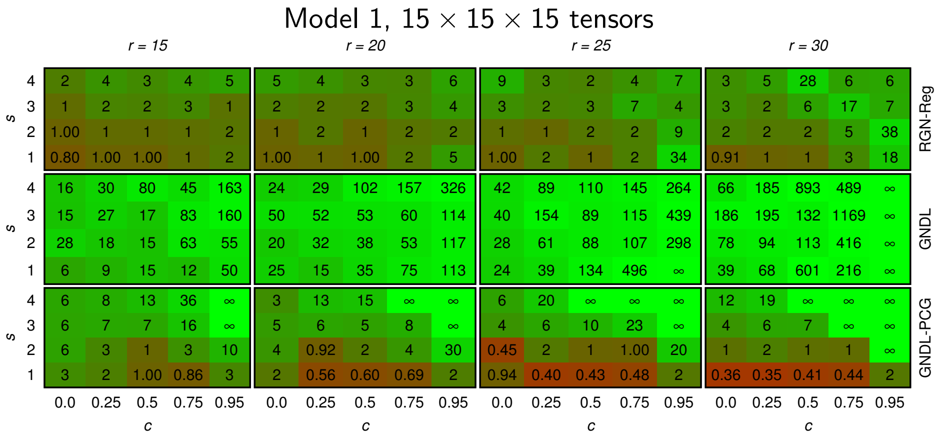
Model 1, $15 \times 15 \times 15$ tensors



		r = 15				r = 20				r = 25				r = 30			
	s \ c	0.0	0.25	0.5	0.75	0.0	0.25	0.5	0.75	0.0	0.25	0.5	0.75	0.0	0.25	0.5	0.75
RGN-Reg	4	2	1	1	1	2	2	1	1	2	0.79	2	∞	2	3	1	2
RGN-Reg	3	2	2	1	1	1	1	2	2	1	2	3	2	3	3	1	2
RGN-Reg	2	1	1	1	1	1	1.00	1	1	1	1	1	2	2	2	2	3
RGN-Reg	1	2	1.00	2	1	1.00	1.00	1.00	2	1	1	1	2	0.87	2	2	2
GNDL	4	15	21	30	38	23	52	30	66	53	16	44	246	132	∞	56	284
GNDL	3	16	17	18	65	21	34	27	69	26	45	38	182	52	304	171	130
GNDL	2	9	19	14	30	18	20	29	92	40	70	49	53	31	44	76	509
GNDL	1	11	11	11	26	14	21	37	33	57	27	82	168	35	68	250	99
GNDL-PCG	4	1	2	4	∞	3	6	5	∞	3	∞	3	∞	∞	∞	∞	4
GNDL-PCG	3	2	1	4	20	1	2	2	∞	2	1	5	10	1	1	∞	5
GNDL-PCG	2	1	0.83	1	3	0.91	0.89	1.00	2	0.78	1	0.75	1	0.61	0.54	0.71	8
GNDL-PCG	1	1	0.60	0.83	0.86	0.44	0.40	0.45	0.71	0.33	0.50	0.42	0.59	0.26	0.24	0.31	0.45

noise level $\tau = 10^{-3}$

Speedup of RGN-HR


Model 1, $15 \times 15 \times 15$ tensors

RGN-Reg

	r = 15					r = 20					r = 25					r = 30				
s \ c	0.0	0.25	0.5	0.75	0.95	0.0	0.25	0.5	0.75	0.95	0.0	0.25	0.5	0.75	0.95	0.0	0.25	0.5	0.75	0.95
4	2	4	3	4	5	5	4	3	3	6	9	3	2	4	7	3	5	28	6	6
3	1	2	2	3	1	2	2	2	3	4	3	2	3	7	4	3	2	6	17	7
2	1.00	1	1	1	2	1	2	1	2	2	1	1	2	2	9	2	2	2	5	38
1	0.80	1.00	1.00	1	2	1.00	1	1.00	2	5	1.00	2	1	2	34	0.91	1	1	3	18

GNDL

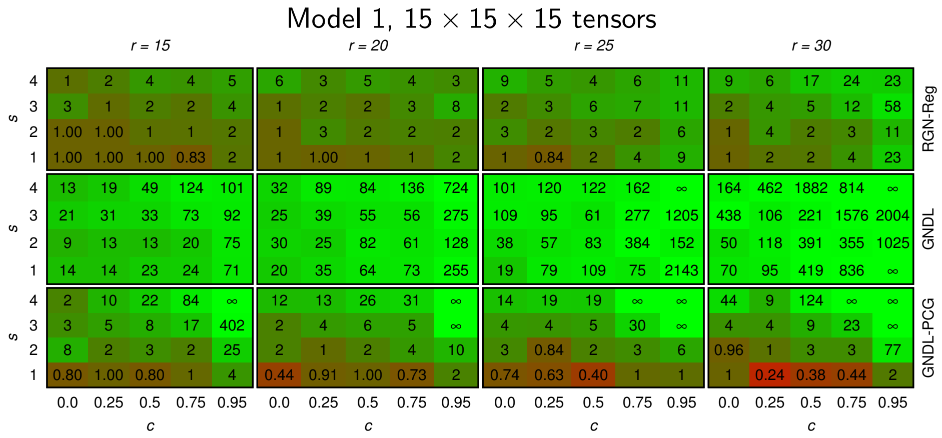
	r = 15					r = 20					r = 25					r = 30				
s \ c	0.0	0.25	0.5	0.75	0.95	0.0	0.25	0.5	0.75	0.95	0.0	0.25	0.5	0.75	0.95	0.0	0.25	0.5	0.75	0.95
4	16	30	80	45	163	24	29	102	157	326	42	89	110	145	264	66	185	893	489	∞
3	15	27	17	83	160	50	52	53	60	114	40	154	89	115	439	186	195	132	1169	∞
2	28	18	15	63	55	20	32	38	53	117	28	61	88	107	298	78	94	113	416	∞
1	6	9	15	12	50	25	15	35	75	113	24	39	134	496	∞	39	68	601	216	∞

GNDL-PCG

	r = 15					r = 20					r = 25					r = 30				
s \ c	0.0	0.25	0.5	0.75	0.95	0.0	0.25	0.5	0.75	0.95	0.0	0.25	0.5	0.75	0.95	0.0	0.25	0.5	0.75	0.95
4	6	8	13	36	∞	3	13	15	∞	∞	6	20	∞	∞	∞	12	19	∞	∞	∞
3	6	7	7	16	∞	5	6	5	8	∞	4	6	10	23	∞	4	6	7	∞	∞
2	6	3	1	3	10	4	0.92	2	4	30	0.45	2	1	1.00	20	1	2	1	1	∞
1	3	2	1.00	0.86	3	2	0.56	0.60	0.69	2	0.94	0.40	0.43	0.48	2	0.36	0.35	0.41	0.44	2

noise level $\tau = 10^{-5}$

Speedup of RGN-HR



Model 1, $15 \times 15 \times 15$ tensors

		r = 15					*r = 20*					*r = 25*					*r = 30*				
	s \ *c*	0.0	0.25	0.5	0.75	0.95	0.0	0.25	0.5	0.75	0.95	0.0	0.25	0.5	0.75	0.95	0.0	0.25	0.5	0.75	0.95
RGN-Reg	4	1	2	4	4	5	6	3	5	4	3	9	5	4	6	11	9	6	17	24	23
	3	3	1	2	2	4	1	2	2	3	8	2	3	6	7	11	2	4	5	12	58
	2	1.00	1.00	1	1	2	1	3	2	2	2	3	2	3	2	6	1	4	2	3	11
	1	1.00	1.00	1.00	0.83	2	1	1.00	1	1	2	1	0.84	2	4	9	1	2	2	4	23
GNDL	4	13	19	49	124	101	32	89	84	136	724	101	120	122	162	∞	164	462	1882	814	∞
	3	21	31	33	73	92	25	39	55	56	275	109	95	61	277	1205	438	106	221	1576	2004
	2	9	13	13	20	75	30	25	82	61	128	38	57	83	384	152	50	118	391	355	1025
	1	14	14	23	24	71	20	35	64	73	255	19	79	109	75	2143	70	95	419	836	∞
GNDL-PCG	4	2	10	22	84	∞	12	13	26	31	∞	14	19	19	∞	∞	44	9	124	∞	∞
	3	3	5	8	17	402	2	4	6	5	∞	4	4	5	30	∞	4	4	9	23	∞
	2	8	2	3	2	25	2	1	2	4	10	3	0.84	2	3	6	0.96	1	3	3	77
	1	0.80	1.00	0.80	1	4	0.44	0.91	1.00	0.73	2	0.74	0.63	0.40	1	1	1	0.24	0.38	0.44	2

noise level $\tau = 10^{-7}$

Speedup of RGN-HR

Model 2, $13 \times 11 \times 9$ tensors

GNDL

s \ r	5	7	9	11	13
4	4	28	40	59	failed
3	7	16	46	381	inf
2	4	17	31	168	452
1	4	5	20	31	166
0	3	4	10	16	58

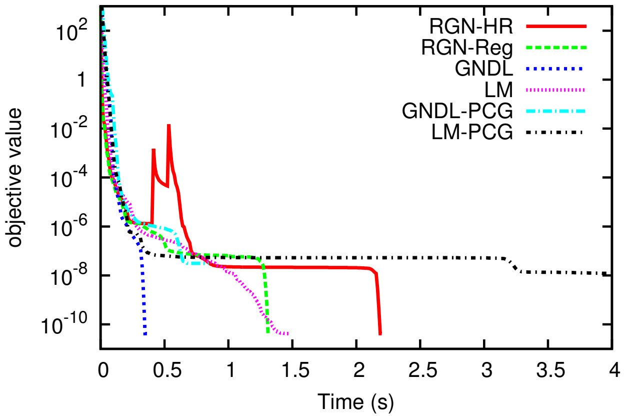
GNDL-PCG

s \ r	5	7	9	11	13
4	6	67	35	inf	failed
3	7	21	23	24	inf
2	13	18	22	27	24
1	17	7	2	16	24
0	7	8	6	4	10

noise level $\tau = 10^{-5}$

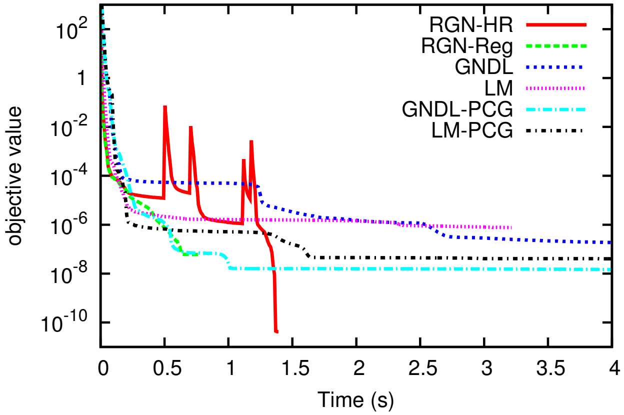
Convergence plots

Model 2, rank 7, scaling $s = 2$, noise level $\tau = 10^{-5}$



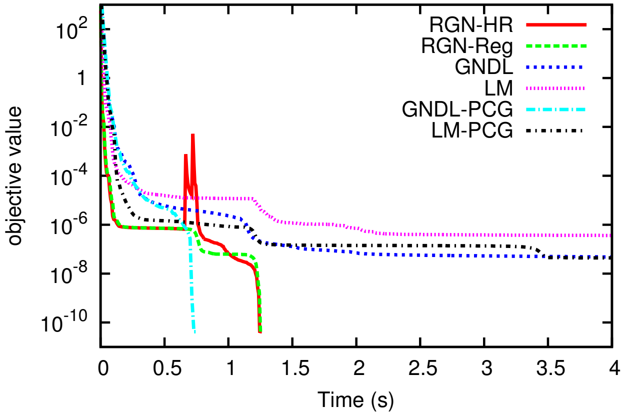
Convergence plots

Model 2, rank 7, scaling $s = 2$, noise level $\tau = 10^{-5}$



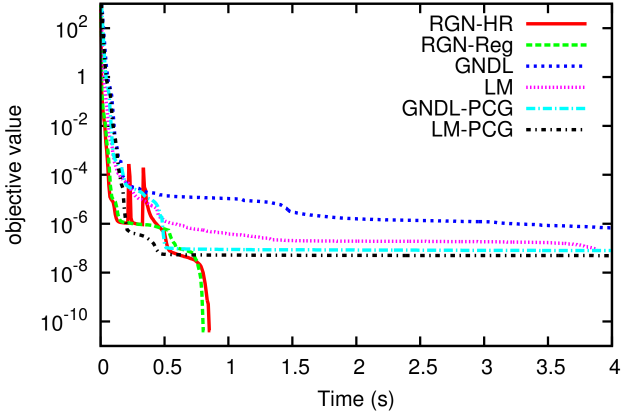
Convergence plots

Model 2, rank 7, scaling $s = 2$, noise level $\tau = 10^{-5}$



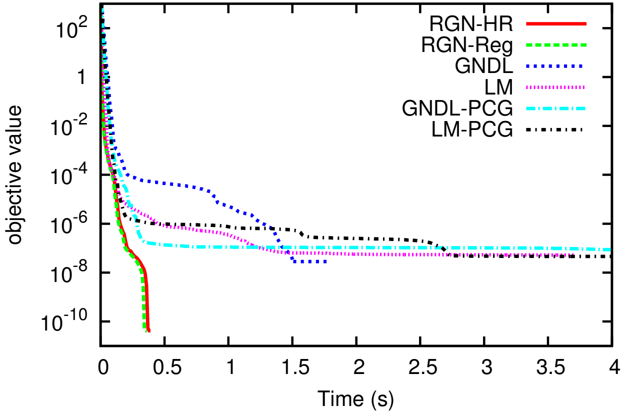
Convergence plots

Model 2, rank 7, scaling $s = 2$, noise level $\tau = 10^{-5}$



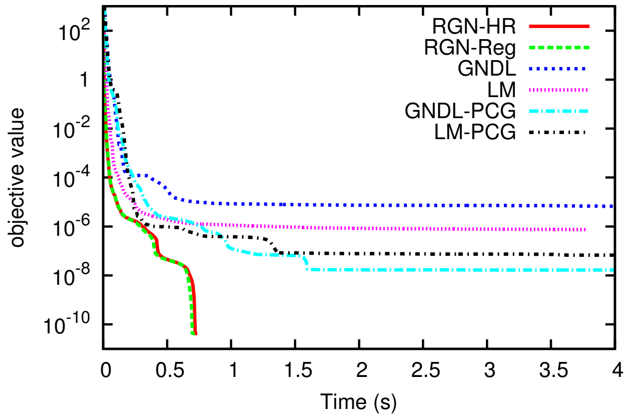
Convergence plots

Model 2, rank 7, scaling $s = 2$, noise level $\tau = 10^{-5}$



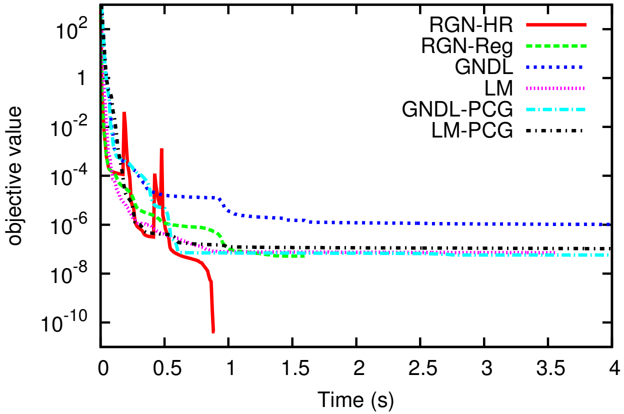
Convergence plots

Model 2, rank 7, scaling $s = 2$, noise level $\tau = 10^{-5}$



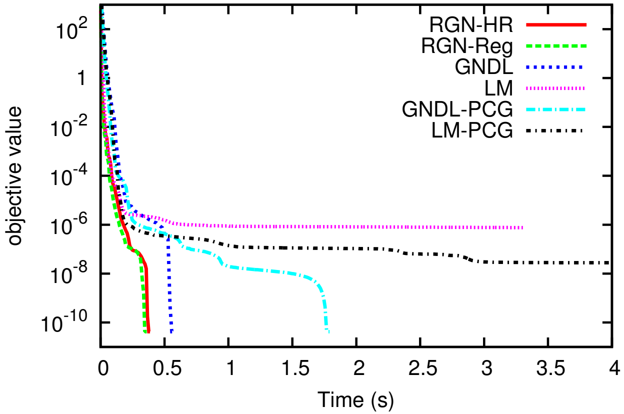
Convergence plots

Model 2, rank 7, scaling $s = 2$, noise level $\tau = 10^{-5}$



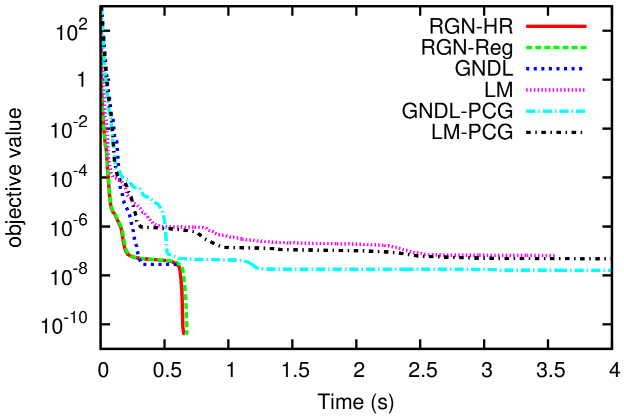
Convergence plots

Model 2, rank 7, scaling $s = 2$, noise level $\tau = 10^{-5}$



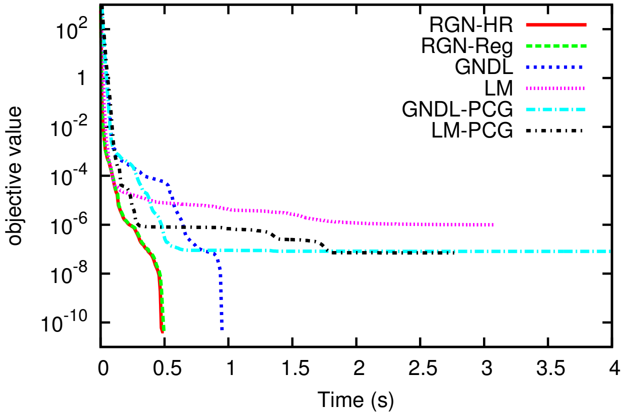
Convergence plots

Model 2, rank 7, scaling $s = 2$, noise level $\tau = 10^{-5}$



Convergence plots

Model 2, rank 7, scaling $s = 2$, noise level $\tau = 10^{-5}$



Overview

Tensorlab is a package facilitating computations with both structured and unstructured tensors in Matlab.

Development on Tensorlab v1.0 started in 2011 at KU Leuven by the research group of L. De Lathauwer. The current version, v3.0, was released in March 2016. Version 4.0 is anticipated **next week**!

Tensorlab focuses on interpretable models, offering algorithms for working with

- Tucker decompositions,
- tensor rank decompositions, and
- certain block term decompositions.

In addition, it offers a flexible framework called **structured data fusion** in which the various decompositions can be **coupled** or fused while imposing additional structural constraints, such as symmetry, sparsity and nonnegativity.

Overview

A **dense tensor** is represented as a plain Matlab array:

```
A = randn(3,5,7,11);

A(1:2,1:2,1:2,1:2)
ans(:,:,1,1) =                    ans(:,:,2,1) =
   2.123880   -0.071091              0.088988   -0.38890
   2.041218   -0.937120              1.129268   -1.10351
ans(:,:,1,2) =                    ans(:,:,2,2) =
   1.251419    0.451602              0.94860     0.55588
  -0.032686   -2.479674             -0.46106    -0.47215
```

The **Frobenius norm** of a tensor is computed as follows.

```
A = reshape(1:100,[2 5 10]);

frob(A)
ans =
   581.6786

frob(A,'squared') - 100*101*201/6
ans =
   0
```

Flattenings are computed as follows.

```
A = reshape(1:24,[2 3 4]);

% mode-1 flattening
tens2mat(A, [1], [2 3])
ans =
   1    3    5    7    9   11   13   15   17   19   21   23
   2    4    6    8   10   12   14   16   18   20   22   24

% mode-2 flattening
tens2mat(A, [2], [])
ans =
      1     2     7     8    13    14    19    20
      3     4     9    10    15    16    21    22
      5     6    11    12    17    18    23    24
```

Multilinear multiplications are computed as follows.

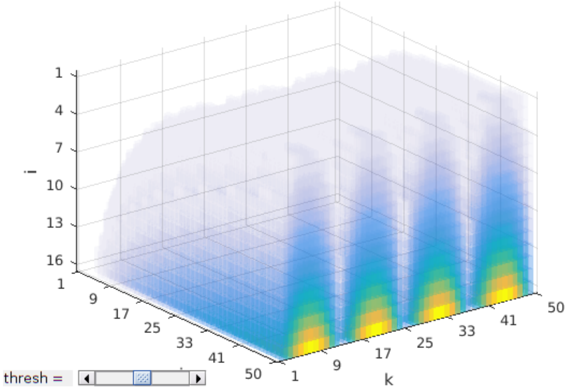
```
A = randn(5,5,5);
U1 = randn(5,5);
U2 = randn(5,5);
U3 = randn(5,5);

T = tmprod(A, {U1,U2,U3}, 1:3);
X1 = tmprod(A, {U2,U3}, 2:3, 'H');
X2 = tmprod(A, {U2',U3'}, 2:3);
frob(X1-X2)
ans =
     0
```

Tensorlab has a nice way for visualizing third-order tensors.

```
U = {1.25.^(-10:5)', linspace(0,1,50)',
        2*abs(sin(linspace(0,4*pi,50)))'};
A = cpdgen(U);
voxel3(A)
```

produces the following graphic:



Overview

Given a core tensor and the factor matrices, Tensorlab can generate the full tensor represented by this Tucker decomposition as follows.

```
S = randn(5,5,5);
U = {randn(4,5), randn(6,5), randn(7,5)};
T1 = lmlragen(U, S);
size(T1)
ans =
    4     6     8

% Compare with definition
T2 = tmprod(S, U, 1:3);
frob(T2-T1)
ans =
    2.4235e-15
```

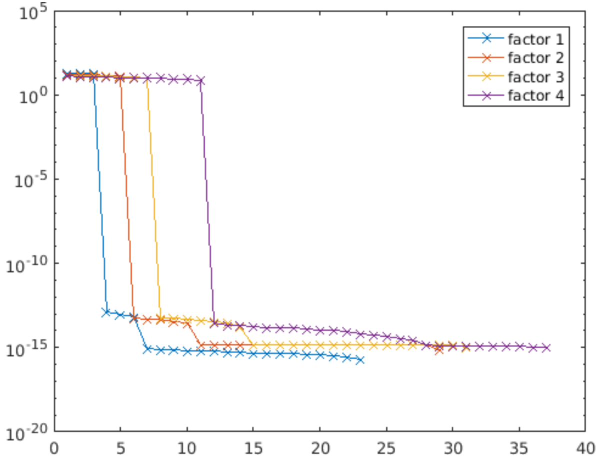
The HOSVD is called the **multilinear singular value decomposition** in Tensorlab and can be computed as follows.

```
[F, C] = lmlra_rnd([23 29 31 37],[3 5 7 11]);
A = lmlragen(F, C);
[U, S, sv] = mlsvd(A);  % 0.32 s
```

The factor-*k* singular values can be plotted by running

```
for k = 1 : 4, semilogy(sv{ k }, 'x-'), hold all, end
legend('factor␣1', 'factor␣2', 'factor␣3', 'factor␣4')
```

which produces the graphic



Of practical importance are truncated HOSVDs. The default strategy in Tensorlab is sequential truncation, while parallel truncation is provided as an option.

```
[F, C] = lmlra_rnd([100 100 10000],[25 25 25]);
A = lmlragen(F, C) + 1e-5*randn([100 100 10000]);

% Sequential truncation
[U1, S1] = mlsvd(A, [25 25 25]);     % 26.5 s
[U2, S2] = mlsvd(A, 5e-2, 1:3);      % 24.6 s

% Parallel truncation
[V1, T1] = mlsvd(A, [25 25 25], 0);  % 387.8 s
[V2, T2] = mlsvd(A, 5e-2, 0);        % 374.6 s
```

Tensorlab also offers optimization algorithms for seeking the **optimal low multilinear rank approximation** of a tensor. By default Tensorlab chooses an initial point by either a fast approximation to the ST-HOSVD via randomized SVDs (`mlsvd_rsi`), or by adaptive cross approximation (`lmlra_aca`).

The basic usage is as follows:

```
[F, C] = lmlra_rnd([100 100 10000],[25 25 25]);
A = lmlragen(F, C);

% Compute lmlra using default settings.
[U, S] = lmlra(A, [25 25 25]);
```

Overview

Tensorlab represents a CPD by a **cell array** containing its factor matrices. For example, random factor matrices can be generated as follows:

```
sizeTensor = [10 11 87];
rnk = 5;
F = cpd_rnd(sizeTensor, rnk)
F =
  1x3 cell array
    [10x5 double]    [11x5 double]    [87x5 double]
```

The tensor represented by these factor matrices can be generated as follows:

```
A = cpdgen(F);
```

Tensorlab offers several algorithms for computing CPDs. A deterministic but unstable PBA that can be applied if the rank is smaller than two of the dimensions is `cpd_gevd`.

```
tF = cpd_rnd([29 11 85], 5);
A = cpdgen(tF);
aF = cpd_gevd(A,5);
frobcpdres(A, aF)
ans =
   1.1966e-12
```

Warning

`cpd_gevd` returns random factor matrices if the assumptions of the method are not satisfied.

Several optimization methods are implemented in Tensorlab for approximating a tensor by a low-rank CPD.

The advised way for computing a CPD is via the **driver routine** `cpd`, which automatically performs several steps:

1. optional Tucker compression,
2. choice of initialization (`cpd_gevd` if possible, otherwise random),
3. optional decompression and refinement if compression was applied.

The basic usage is as follows:

```
tF = cpd_rnd([29 11 85], 5);
A = cpdgen(tF);
aF = cpd(A, 5);
frobcpdres(A, aF)
ans =
   5.0921e-14
```

Optionally, some algorithm options can be specified. The most important options are the following.

`Compression`: the string `'auto'`, the boolean `false`, or a function handle to the Tucker compression algorithm, for example `mlsvd`, `lmlra_aca` or `mlsvd_rsi`.

`Initialization`: the string `'auto'`, or a function handle to the initialization method that should be employed, for example `cpd_rnd` or `cpd_gevd`.

`Algorithm`: a function handle to the optimization method for computing the tensor rank decomposition, for example `cpd_als`, `cpd_nls` or `cpd_minf`.

`AlgorithmOptions`: a structure containing the options to for the optimization method.

The algorithm options are important in their own right, as they determine how much computational effort is spent and how accurate the returned solution will be. The main options are the following.

`TolAbs`: The tolerance for the squared error between the tensor represented by the current factor matrices and the given tensor. If it is less than this value, the optimization method halts successfully.

`TolFun`: The tolerance for the relative change in function value. If it is less than this value, the optimization method halts successfully.

`TolX`: The tolerance for the relative change in factor matrices. If it is less than this value, the optimization method halts successfully.

`MaxIter`: The maximum number of iterations the optimization method will perform before giving up.

For example,

```
algOptions = struct;
algOptions.TolAbs = 1e-12;
algOptions.TolFun = 1e-12;
algOptions.TolX    = 1e-8;
algOptions.MaxIter = 500;
algOptions.Algorithm = @nls_gndl;

options = struct;
options.Compression = false;
options.Initialization = @cpd_rnd;
options.Algorithm = @cpd_nls;
options.AlgorithmOptions = algOptions;
```

```
% Create an easy problem
tF = cpd_rnd([29 11 85], 5);
A = cpdgen(tF);

% Solve using options
[aF, out] = cpd(A, 5, options);
frobcpdres(A, aF)
ans =
    6.6368e-10
```

You can also ask for detailed information about the progression of the optimization method.

```
 out.Algorithm
ans =
   struct with fields:
          Name: 'cpd_nls'
         alpha: []
  cgiterations: [2 6 10 14 15 15 15 15 15 15 15 ... 15]
      cgrelres: [6.6613e-17 2.3642e-07 ... 7.5798e-07]
         delta: [1.1619 2.3238 4.6476 9.2952 ... 9.2973]
          fval: [8.8694e+04 8.8649e+04 ... 2.4837e-22]
          info: 4 % Absolute tolerance reached
        infops: []
    iterations: 15
        relerr: 5.2922e-14
       relfval: [4.9847e-04 0.0081 ... 1.7052e-15]
       relstep: [0.3000 0.6032 0.9451 ... 2.9188e-08]
           rho: [0.5589 3.0163 3.3691 ... 1.0000]
```

Overview

References for sensitivity

- Bürgisser, Cucker, *Condition: The Geometry of Numerical Algorithms*, Springer, 2013.
- Blum, Cucker, Shub, Smale, *Complexity and Real Computation*, Springer, 1998.
- Breiding, Vannieuwenhoven, *The condition number of join decompositions*, SIAM Journal on Matrix Analysis, 2017.
- Rice, *A theory of condition*, SIAM Journal on Numerical Analysis, 1966.

References for algorithms

- Absil, Mahony, Sepulchre, *Optimization Algorithms on Matrix Manifolds*, Princeton University Press, 2008.
- Beltrán, Breiding, Vannieuwenhoven, *Pencil-based algorithms for tensor decomposition are unstable*, arXiv:1807.04159, 2018.
- Breiding, Vannieuwenhoven, *A Riemannian trust region method for the canonical tensor rank approximation problem*, SIAM Journal on Optimization, 2018.
- Kolda, Bader, *Tensor decompositions and applications*, SIAM Review, 2008.
- Kressner, Steinlechner, Vandereycken, *Low-Rank tensor completion by Riemannian optimization*, BIT, 2014.
- Leurgans, Ross, Abel, *A decomposition for three-way arrays*, SIAM Journal on Matrix Analysis and Applications, 1993.
- Nocedal, Wright, *Numerical Optimization*, Springer, 2006.

References for algorithms

- Sorber, Van Barel, 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, 2013.
- Uschmajew, *Local convergence of the alternating least squares algorithm for canonical tensor approximation*, SIAM Journal on Matrix Analysis and Applications, 2012.

References for Tensorlab

- Sidiropoulos, De Lathauwer, Fu, Huang, Papalexakis, Faloutsos, *Tensor decompositions for signal processing and machine learning*, IEEE Transactions on Signal Processing, 2017.
- Sorber, Van Barel, 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, 2013.
- Vervliet, Debals, Sorber, Van Barel, De Lathauwer, *Tensorlab v3.0*, `www.tensorlab.net`.
- Vervliet, De Lathauwer, *Numerical optimization based algorithms for data fusion*, 2018.