An experience based guide to Tensorlab

Nico Vervliet

March 3, 2015

Disclaimer

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enimad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enimad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enimad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Optimization

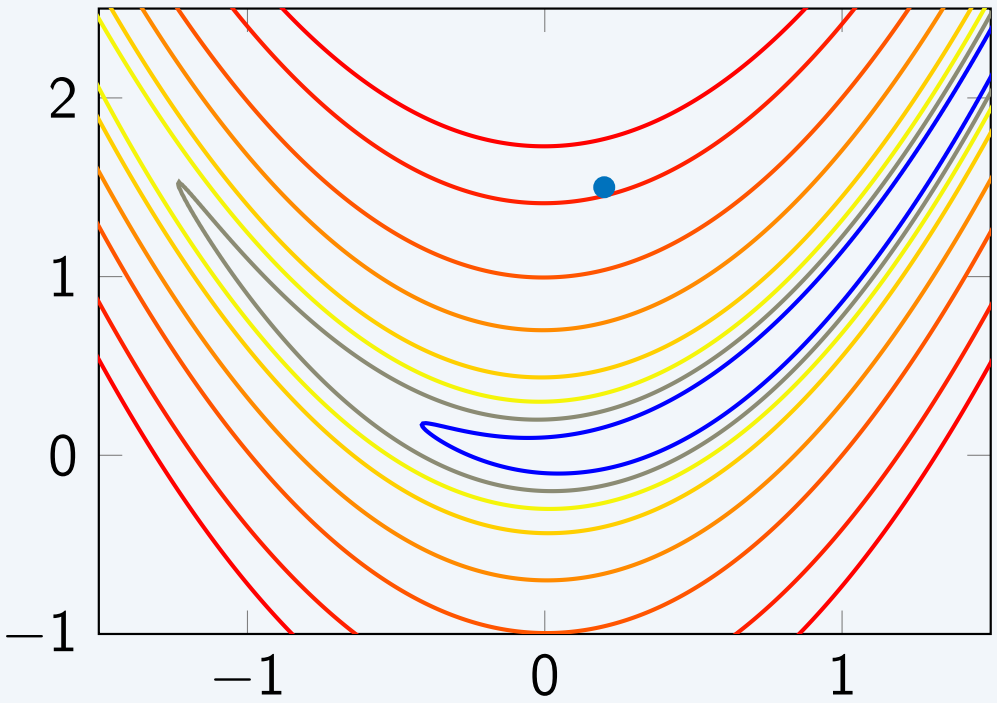
Basics unconstrained optimization

- Find $\mathbf{x}$ that minimizes objective function f:

$$\min_{\mathbf{x}} f(\mathbf{x})$$

- Iteratively update guess $\mathbf{x}_k$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$



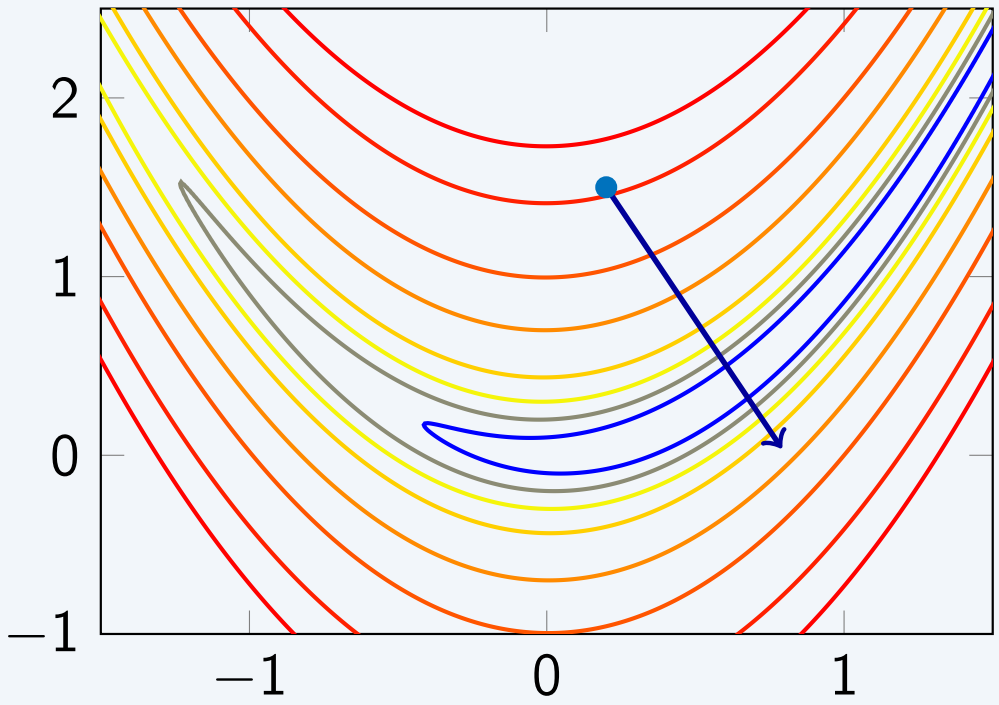
Basics unconstrained optimization

- Find $\mathbf{x}$ that minimizes objective function f:

$$\min_{\mathbf{x}} f(\mathbf{x})$$

- Iteratively update guess $\mathbf{x}_k$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$



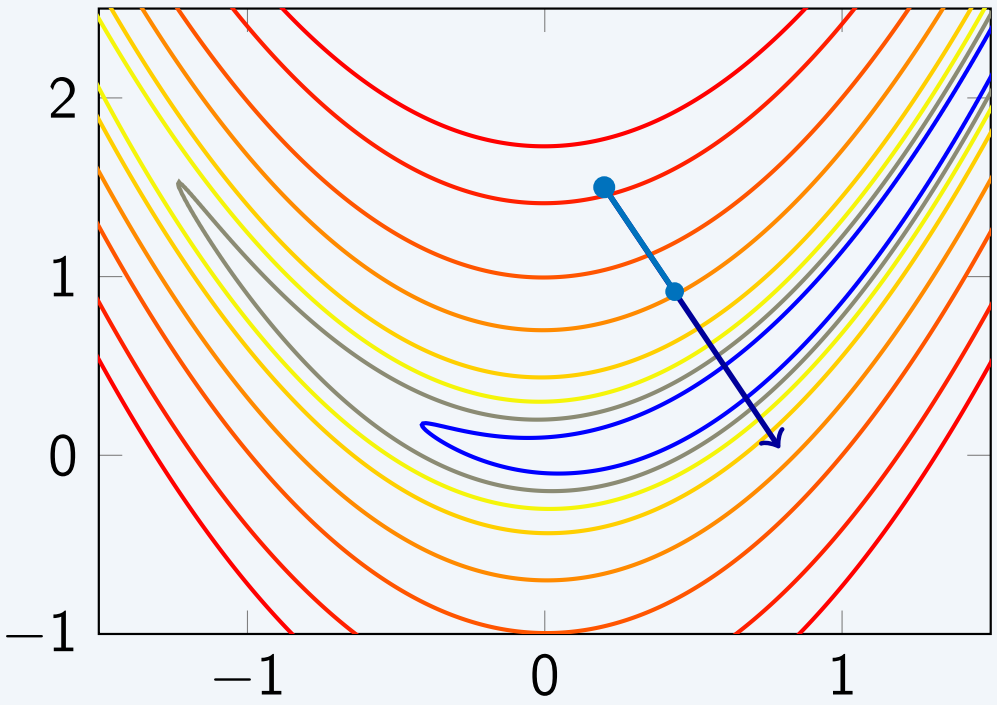
Basics unconstrained optimization

- Find $\mathbf{x}$ that minimizes objective function f:

$$\min_{\mathbf{x}} f(\mathbf{x})$$

- Iteratively update guess $\mathbf{x}_k$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$



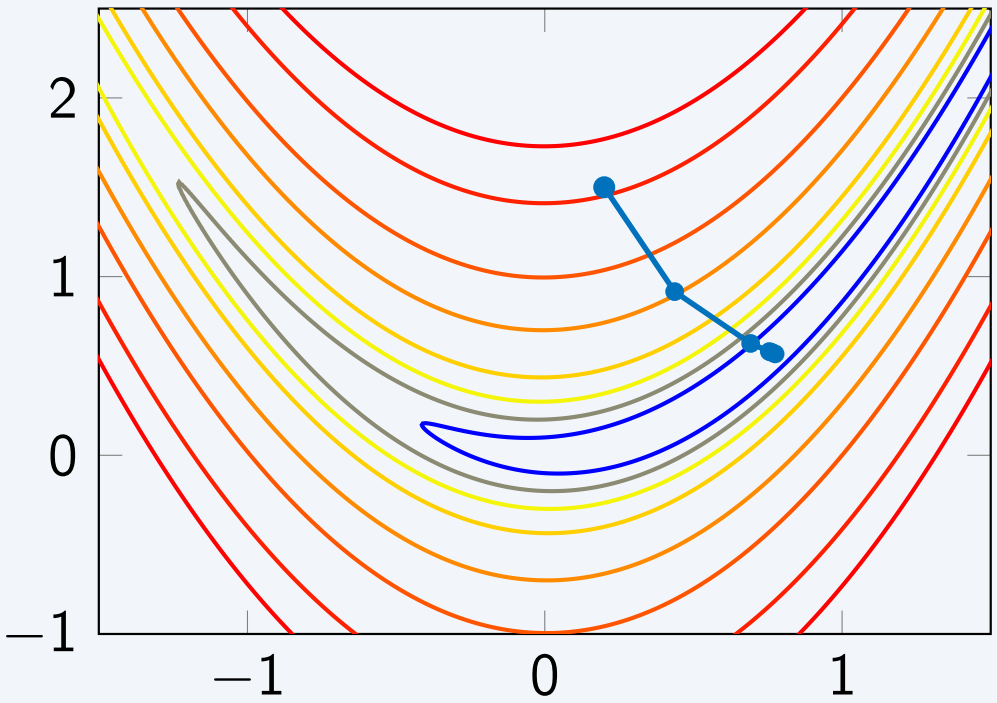
Basics unconstrained optimization

- Find $\mathbf{x}$ that minimizes objective function f:

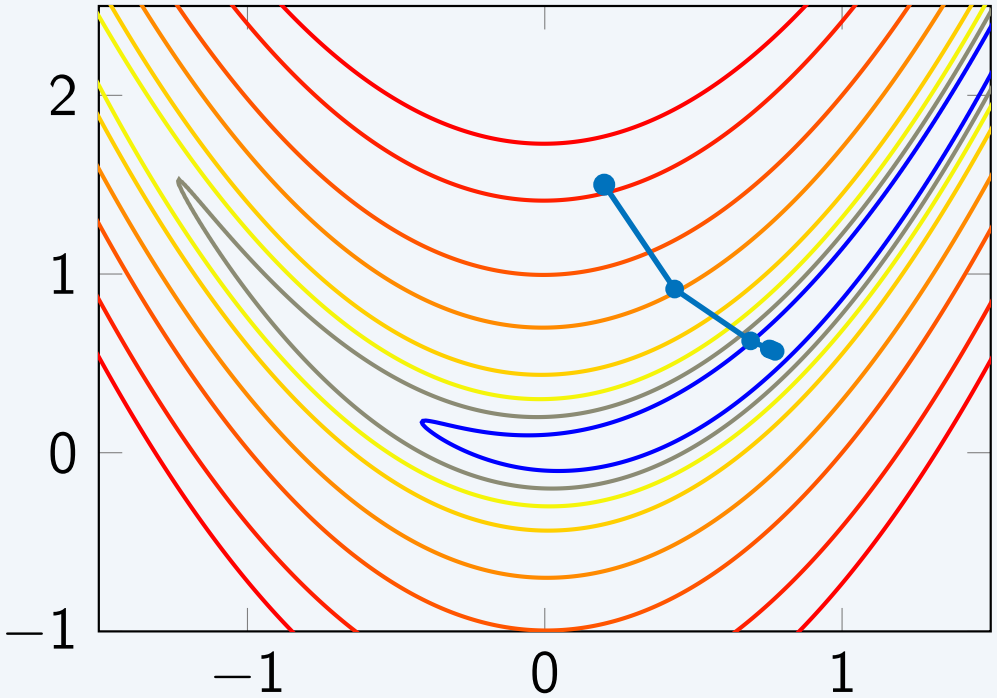
$$\min_{\mathbf{x}} f(\mathbf{x})$$

- Iteratively update guess $\mathbf{x}_k$:

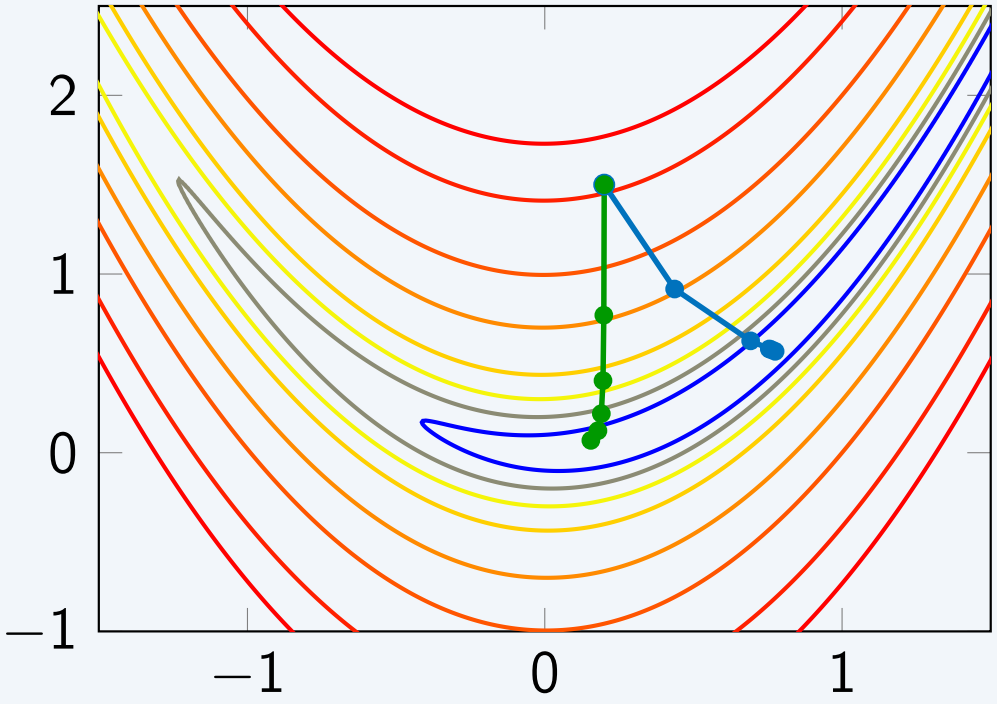
$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$



Choosing step direction



Choosing step direction



Choosing step direction

- Gradient descent

$$\mathbf{p}_k = -\mathbf{g}_k$$

  - Linear convergence

Choosing step direction

- Gradient descent

$$\mathbf{p}_k = -\mathbf{g}_k$$

  - Linear convergence

- Newton

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

  - Quadratic convergence

Choosing step direction

- Gradient descent

$$\mathbf{p}_k = -\mathbf{g}_k$$

  - Linear convergence
- Newton

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

  - Quadratic convergence
- Quasi-Newton

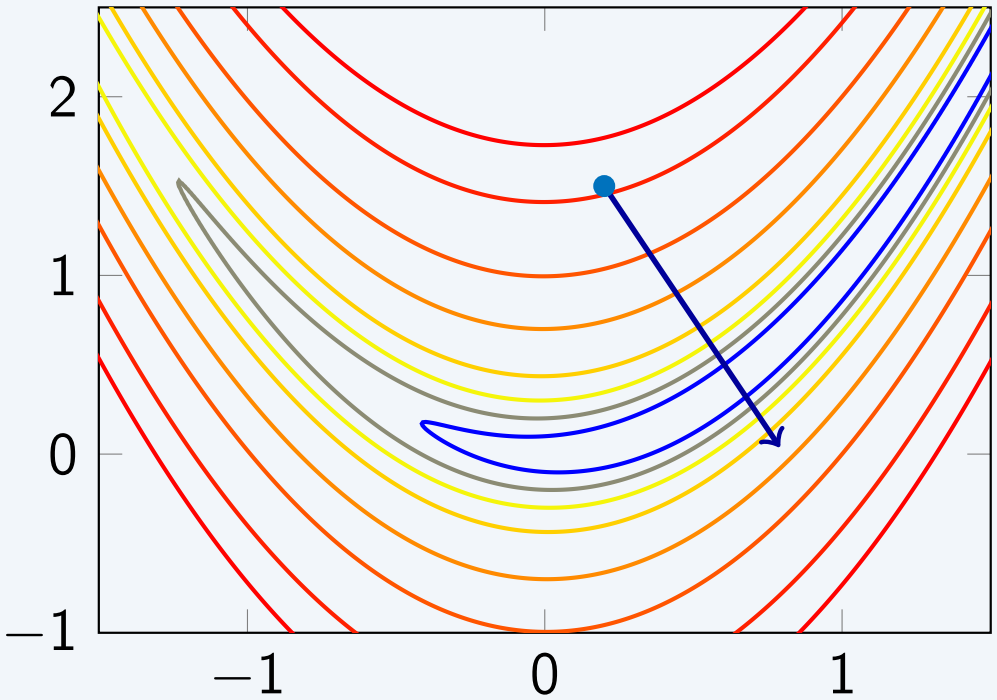
$$\mathbf{p}_k = -\mathbf{B}_k^{-1}\mathbf{g}_k$$

  - Superlinear to quadratic convergence
  - Examples: (L)BFGS, SR1, Gauss–Newton, Levenberg–Marquardt

Choosing step length

- Line search (`cpd_aels`, `cpd_els`, `cpd_lsb`, `ls_mt`)

$$\min_{\alpha_k} f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$$



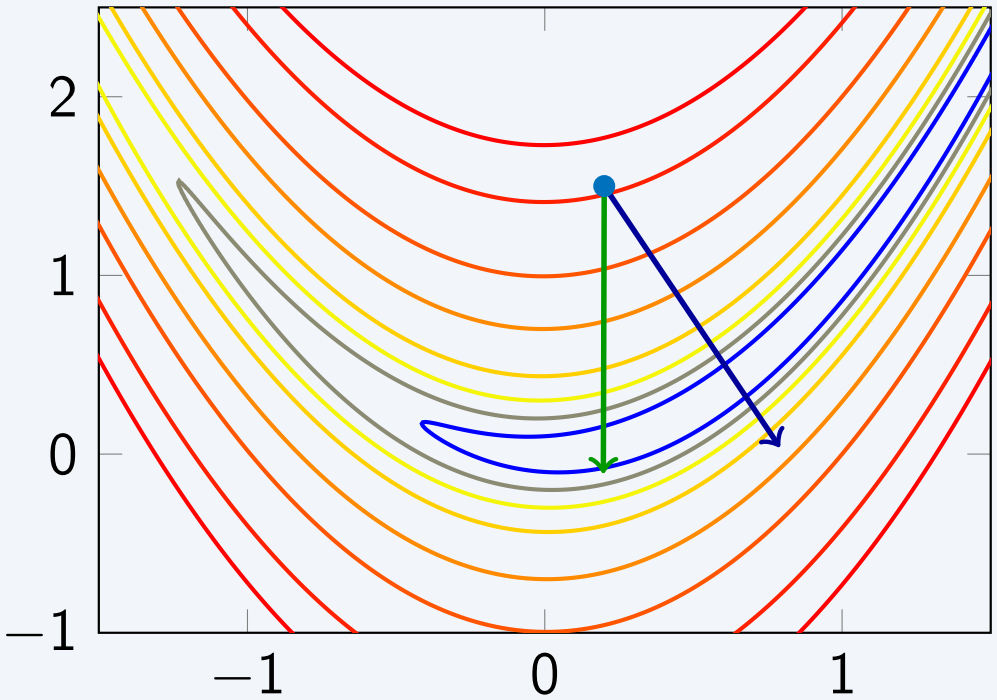
Choosing step length

- Line search (`cpd_aels`, `cpd_els`, `cpd_lsb`, `ls_mt`)

$$\min_{\alpha_k} f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$$

- Plane search (`cpd_eps`)

$$\min_{\alpha_k, \beta_k} f(\mathbf{x}_k + \alpha_k \mathbf{p}_k + \beta_k \bar{\mathbf{p}}_k)$$

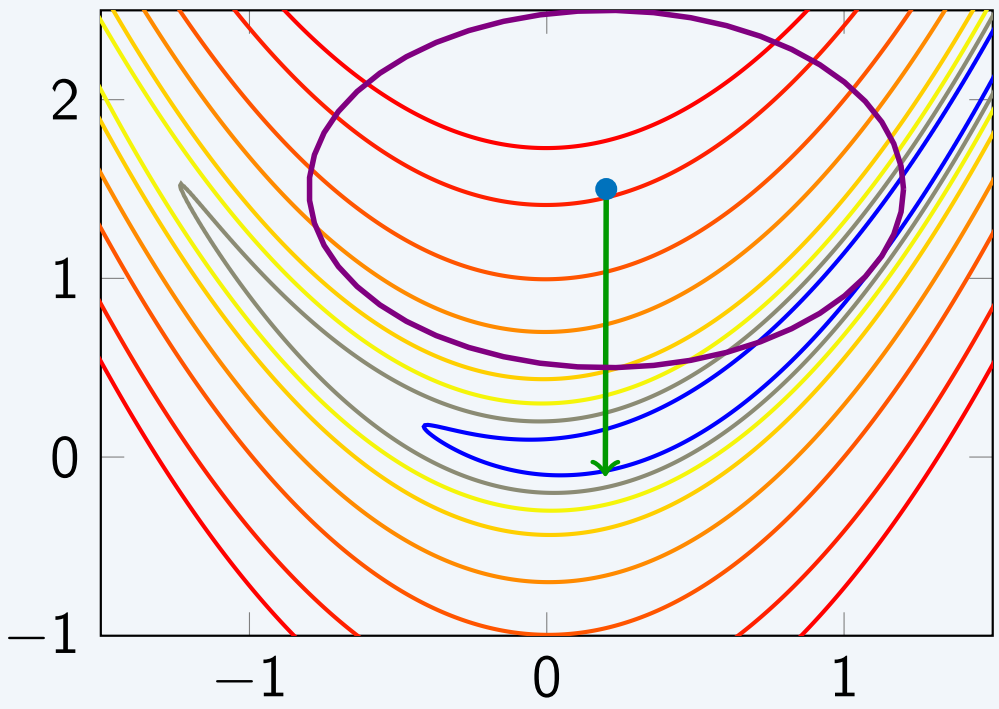


Globalization strategy

- Trust region

$$\min_{\mathbf{p}_k} f_k + \mathbf{g}_k^{\mathsf{H}}\mathbf{p}_k + \frac{1}{2}\mathbf{p}_k^{\mathsf{H}}\mathbf{B}_k\mathbf{p}_k \quad \text{s.t.} \quad ||\mathbf{p}_k|| \leq \Delta_k$$

- Trustworthiness $\rho = (f_k - f_{k+1})/(m_k - m_{k+1})$, m_k is value of quadratic model



Solving systems

Computing $\mathbf{p}_k$ often involves solving

$$\mathbf{B}_k \mathbf{p}_k = -\mathbf{g}_k.$$

The structure of $\mathbf{B}_k$ can often be exploited:

- CG uses only the product $\mathbf{B}_k \mathbf{p}_k$
- Improve convergence using preconditioner (e.g. Block-Jacobi)

$$\mathbf{M}^{-1} \mathbf{B}_k \mathbf{p}_k = -\mathbf{M}^{-1} \mathbf{g}_k.$$

Why?

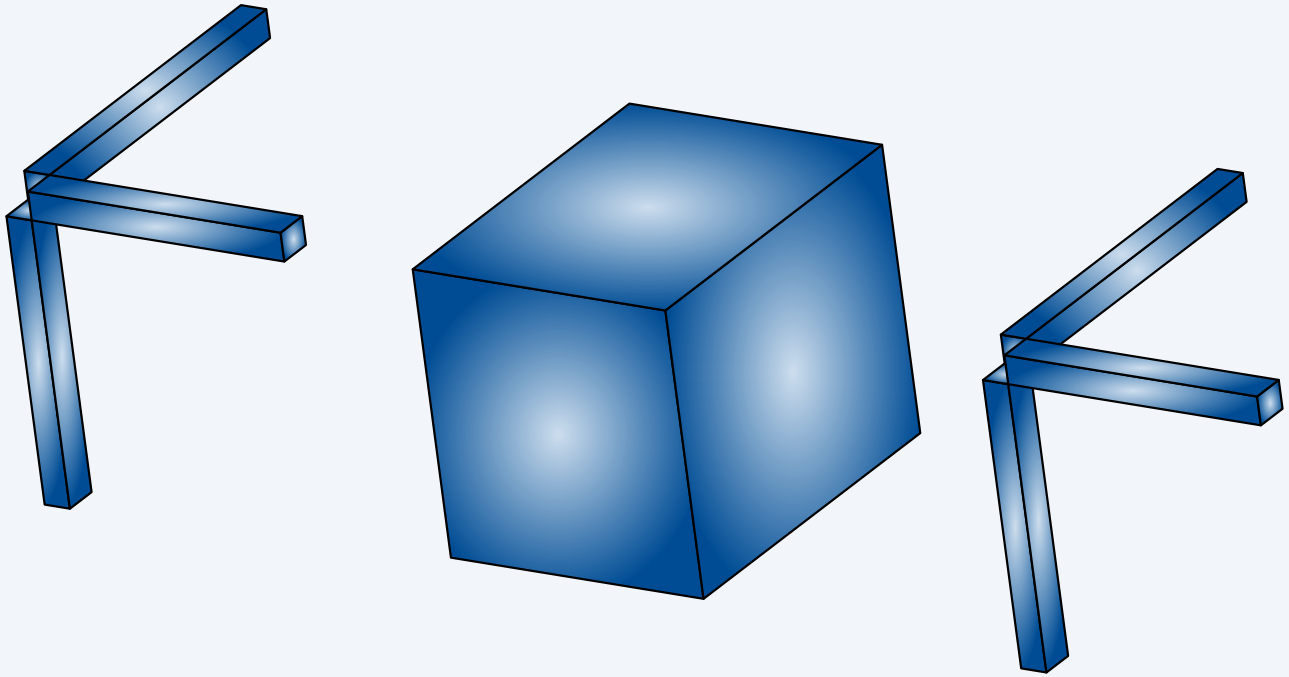
```
    fval                relfval            relstep
    =1/2*norm(F)^2      TolFun = 1e-12     TolX = 1e-06

0:  9.06912804e+05 |
1:  1.60066687e+05 | 8.23503774e-01 | 3.00000000e-01 |

    delta          rho

    1.9076e+01 | 1.0633e+00
```

Options

Control behavior using `options.<...> = ...`:

Parameter	Description
`TolX`	stop if `relstep < TolX`
`TolFun`	stop if `relfval < TolFun`
`MaxIter`	maximum number of iterations
`Display`	display progress every `display` iterations
`PlaneSearch`	plane search algorithm
`Algorithm`	optimization algorithm
`CGMaxIter`	maximum number of CG iterations
`CGTol`	stop if $\|\|\mathbf{B}_k\mathbf{p}_k + \mathbf{g}_k\|\| \,/\, \|\|\mathbf{g}_k\|\| <$ `CGTol`

Canonical Polyadic Decomposition



A nonlinear least squares problem

- A nonlinear, non-convex problem

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C}} \frac{1}{2} ||[\![\mathbf{A},\mathbf{B},\mathbf{C}]\!] - \mathcal{T}||^2$$

A nonlinear least squares problem

- A nonlinear, non-convex problem

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C}} \frac{1}{2} ||[\![\mathbf{A},\mathbf{B},\mathbf{C}]\!] - \mathcal{T}||^2$$

- How to solve?
  - Alternating least squares: `cpd_als`
  - Quasi-Newton, e.g. (L)BFGS: `cpd_minf`
  - Gauss–Newton or Levenberg–Marquardt: `cpd_nls`

NLS algorithms

The problem

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

is linearized around $\mathbf{x}_k$

$$\begin{aligned}\mathcal{F}(\mathbf{x}) &\approx \mathcal{F}(\mathbf{x}_k) + \nabla_{\mathbf{x}}\mathcal{F}(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k) \\ &= \mathcal{F}(\mathbf{x}_k) + \mathbf{J}_k\mathbf{p}_k.\end{aligned}$$

Find $\mathbf{p}_k$ that minimizes the quadratic model

$$\mathbf{J}_k^{\mathrm{H}}\mathbf{J}_k\mathbf{p}_k = -\mathbf{J}_k^{\mathrm{H}}\mathbf{g}_k$$

using CG.

Algorithms to choose from

Algorithm	Cost/Iteration	# Iterations	Overal
`cpd_gevd`	n.a.	1	fast
`cpd_als`	cheap	a lot	slow/fast
`cpd_minf`	moderate	not too many	fast
`cpd_nls`	expensive	a few	fastest

Algorithms to choose from

Algorithm	Cost/Iteration	# Iterations	Overal
`cpd_gevd`	n.a.	1	fast
`cpd_als`	cheap	a lot	slow/fast
`cpd_minf`	moderate	not too many	fast
`cpd_nls`	expensive	a few	fastest

- Which algorithm should you use?
  - In general: `cpd_nls`
  - Sometimes `cpd_minf` gives higher accuracy
  - For easy problems: `cpd_als`
  - In noiseless cases and as initialization: `cpd_gevd`

How to get the right solution fast?

How to get the right solution fast?

I don't know...

A strategy

1. Compression
2. Initialization
3. Decomposition
4. Refinement

Compression

- Compute CPD of compressed tensor instead of full tensor to
  - reduce computation time
  - reduce noise

Compression

- Compute CPD of compressed tensor instead of full tensor to
  - reduce computation time
  - reduce noise
- Algorithms:
  - `lmlra_aca`: fast, but not always very accurate (low SNR)
  - `mlsvd`: slow, but very accurate
  - `lmlra_rff`: fast and accurate experimental

Compression: how?

- When using `cpd`:

```
options.Compression = 'auto' % or true or false;
```

Currently only `lmlra_aca`.

Compression: how?

- When using cpd:

```
options.Compression = 'auto' % or true or false;
```

Currently only `lmlra_aca`.

- Manually:

```
[U, S] = lmlra_aca(T, core_size);
[U, S] = mlsvd(T, core_size);
[U, S] = lmlra_rff(T, core_size);
```

Choose `core_size>=[R R R]`.

```
Ures = cpd_nls(S, Uinit, options);
Ures = cellfun(@(u, v) v*u, Ures, U, 'uni', 0);
```

Initialization

A good initial guess is needed for:

- convergence to a (local) optimum
- convergence to a good optimum
- fast convergence

Algorithms:

- `cpd_rnd`
- `cpd_gevd` (when $R \leq I_n$ for at least two n)
- manual (using knowledge)

Initialization: how?

- When using cpd:

```
options.Initialization = 'auto'; % or @cpd_rnd or
                                 % @cpd_gevd
options.InitializationOptions = struct('Imag', @rand);
```

- Manually:

```
Uinit = cpd_rnd(size_tens, R, struct('Real', @rand));
Uinit = cpd_rnd(T, R, 'optimalScaling', true);
Uinit = cpd_rnd(T, R, 'Angle', pi/3);
Uinit = cpd_gevd(T, R);
```

Decomposition

Actually decompose the (compressed) tensor.
Algorithms:

- `cpd_als`
- `cpd_minf`
- `cpd_nls` (preferred)

Decomposition: how?

- When using cpd:

```
options.Algorithm = @cpd_nls; % default
options.AlgorithmOptions = struct(...);
```

Decomposition: how?

- When using cpd:

```
options.Algorithm = @cpd_nls; % default
options.AlgorithmOptions = struct(...);
```

- Manually:

```
Ures = cpd_nls(S, Uinit, options);
```

Decomposition: how?

- When using cpd:

```
options.Algorithm = @cpd_nls; % default
options.AlgorithmOptions = struct(...);
```

- Manually:

```
Ures = cpd_nls(S, Uinit, options);
```

- Remark: `options` in `cpd_nls` is `options.AlgorithmOptions` in `cpd`.

Decomposition: options

- Change solver

```
opt.Algorithm = @nls_gndl; % or @nls_lm or @nls_gncgs
opt.Algorithm = @(F, dF, z0, opt) ...
                nlsb_gndl(F, dF, LB, UB, z0, opt);
```

Decomposition: options

- Change solver

```
opt.Algorithm = @nls_gndl; % or @nls_lm or @nls_gncgs
opt.Algorithm = @(F, dF, z0, opt) ...
                nlsb_gndl(F, dF, LB, UB, z0, opt);
```

- Change termination criteria

```
opt.TolFun = eps^2; % Rule of thumb: TolX^2
opt.TolX = eps;
opt.MaxIter = 100;
```

Decomposition: options

- Change solver

```
opt.Algorithm = @nls_gndl; % or @nls_lm or @nls_gncgs
opt.Algorithm = @(F, dF, z0, opt) ...
                nlsb_gndl(F, dF, LB, UB, z0, opt);
```

- Change termination criteria

```
opt.TolFun = eps^2; % Rule of thumb: TolX^2
opt.TolX = eps;
opt.MaxIter = 100;
```

- Change advanced settings (only large scale)

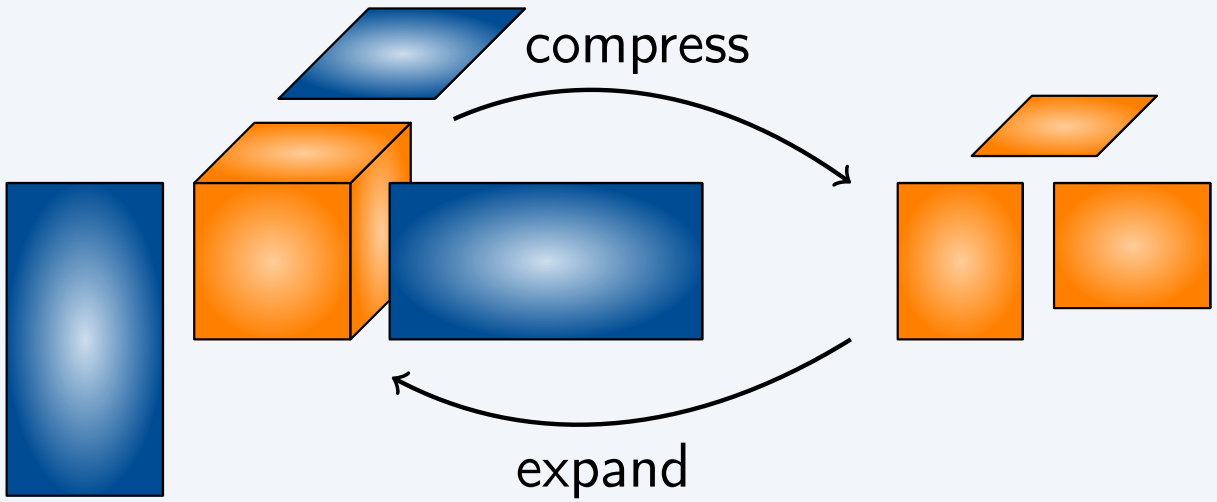
```
opt.CGMaxIter = 10;
opt.CGTol = 1e-6;
```

Higher number of iterations or lower tolerance, decreases # iterations, but increases time per iteration.

Refinement

Use the decompressed solution as initialization for the original tensor (only if compression is used.)

```
% U = factor matrices from compression
% Ures = factor matrices from CPD of core tensor S
Uinit = cellfun(@(u, v) v*u, Ures, U, 'uni', 0);
```



Refinement: how?

- When using cpd

```
options.Refinement = @cpd_nls;
options.RefinementOptions = struct(...);
```

- Manually:

```
Ures = cpd_nls(T, Uinit, struct(...));
```

Estimating the rank

- Use `rankest`
  - Fails on very noisy tensors
  - Try setting `options.MinRelErr = SNR`

Estimating the rank

- Use `rankest`
  - Fails on very noisy tensors
  - Try setting `options.MinRelErr = SNR`
- Compute decompose for different `R`
  - Check fit:

```
frob(cpderr(T, Ures))/frob(T)
```

  - Check factor matrices (if original are known):

```
cpderr(Ures, U0)
```

  - Look for degenerate terms (sums of terms that cancel each other out)
  - Use measures from your field

Estimating the rank

- Use `rankest`
  - Fails on very noisy tensors
  - Try setting `options.MinRelErr = SNR`
- Compute decompose for different `R`
  - Check fit:

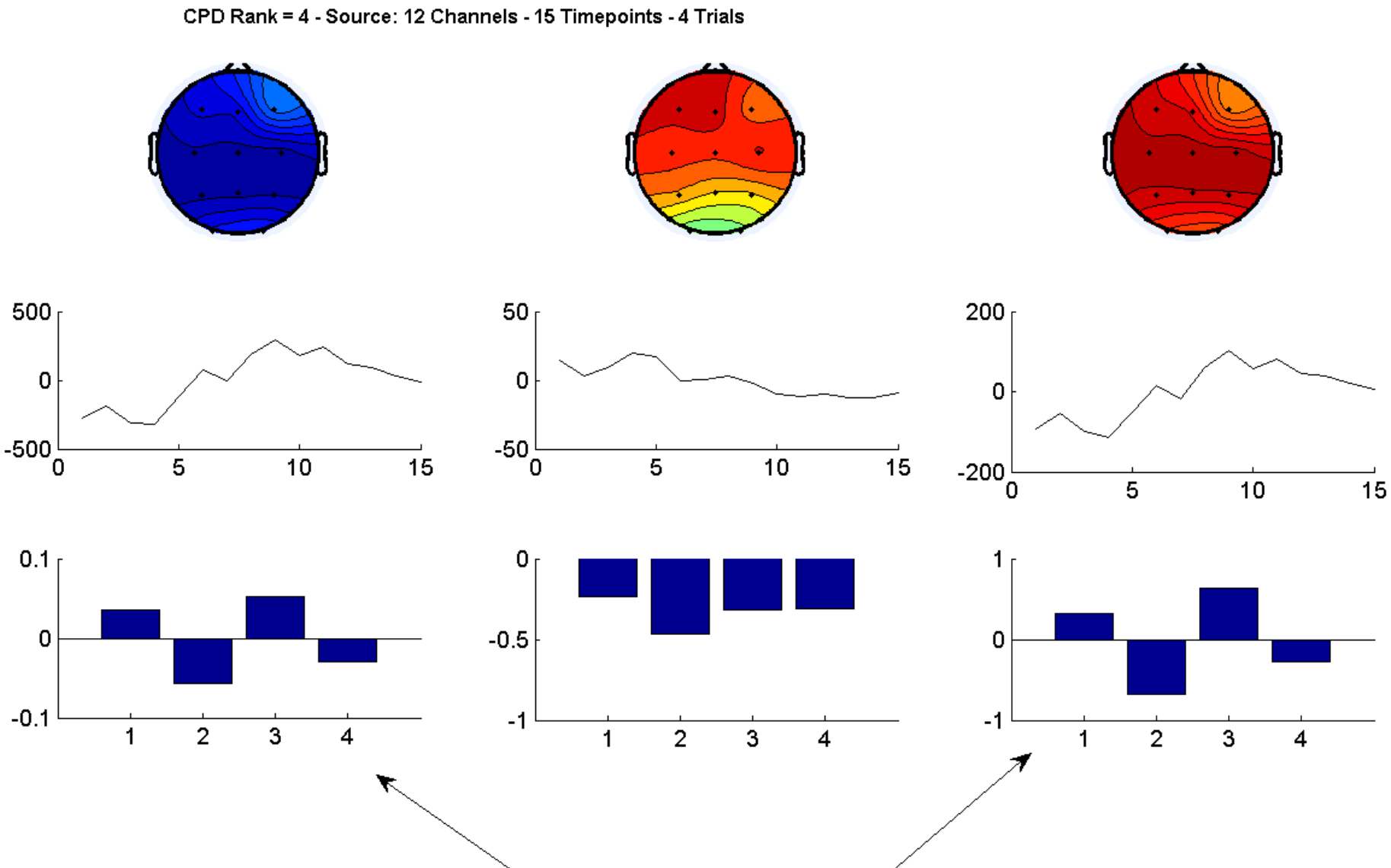
    ```
    frob(cpderr(T, Ures))/frob(T)
    ```

  - Check factor matrices (if original are known):

    ```
    cpderr(Ures, U0)
    ```

  - Look for degenerate terms (sums of terms that cancel each other out)
  - Use measures from your field
- Use knowledge

Degenerate terms



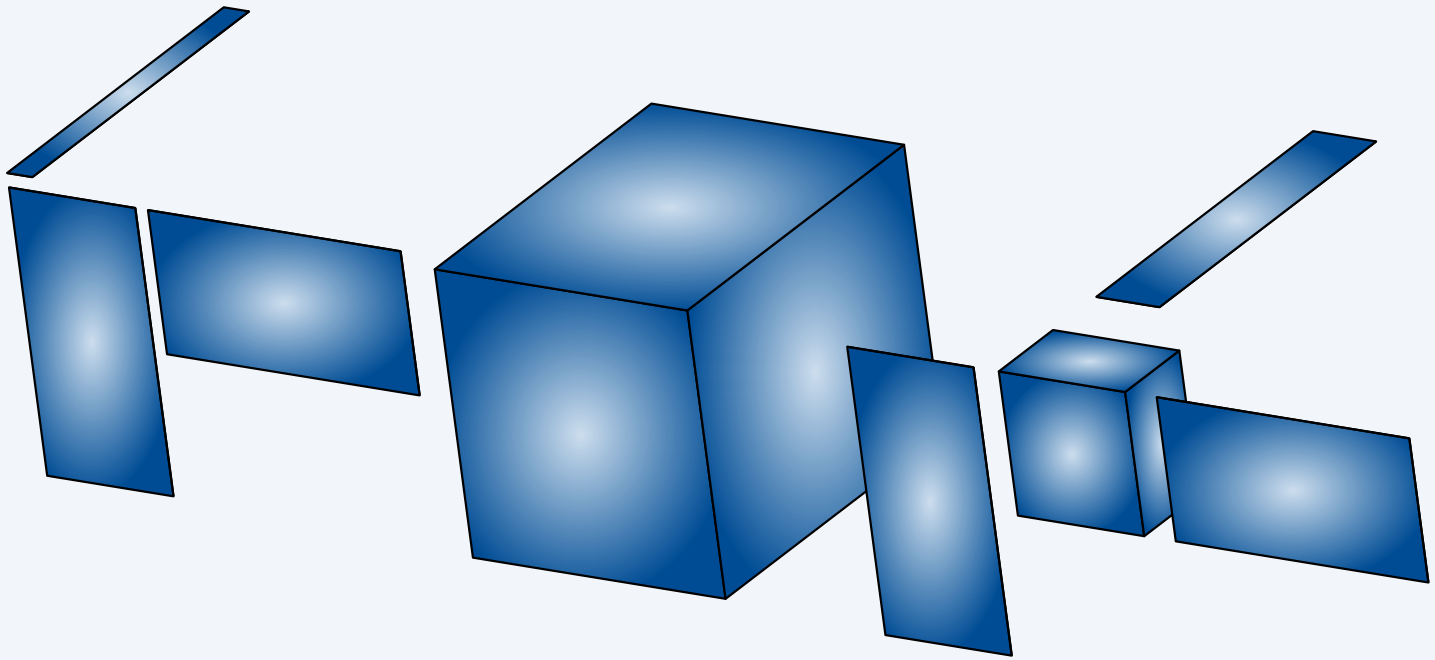
What to use?

- Algorithms
  - `cpd` if standard options
  - `cpd_nls` if non-standard options like other compression algorithm.
  - `cpd_minf` if non-standard options, and `cpd_nls` is too slow or is not accurate

What to use?

- Algorithms
  - `cpd` if standard options
  - `cpd_nls` if non-standard options like other compression algorithm.
  - `cpd_minf` if non-standard options, and `cpd_nls` is too slow or is not accurate
- Tips
  - Try denoising
    - `mlsvd`, `lmlra_rff`
  - Multiple initializations:
    - Set `options.Initialization = @cpd_rnd` in `cpd`
    - Use `cpd_gevd` in `cpd_nls`
  - Use multistage initialization
    - compression, initialization, decomposition, refinement

Block Term Decomposition



General BTD

- Objective function

$$\min \frac{1}{2} \left\| \mathcal{T} - \sum_{r=1}^{R} \left[\!\left[\mathcal{S}^{(r)}; \mathbf{U}^{(r,1)}, \ldots, \mathbf{U}^{(r,N)} \right]\!\right] \right\|^2$$

General BTD

- Objective function

$$\min \frac{1}{2} \left\| \mathcal{T} - \sum_{r=1}^{R} \left[\!\left[\mathcal{S}^{(r)}; \mathbf{U}^{(r,1)}, \ldots, \mathbf{U}^{(r,N)} \right]\!\right] \right\|^2$$

- Tips
  - Many initializations
  - Try denoising
  - Use a large number of iterations

Functions and algorithms

- BTD format

```
% U = { {U^(1,1), ..., U^(1,N), S^(1)}, ...,
%       {U^(R,1), ..., U^(R,N), S^(R)} }
```

Functions and algorithms

- BTD format

```
% U = { {U^(1,1), ..., U^(1,N), S^(1)}, ...,
%        {U^(R,1), ..., U^(R,N), S^(R)} }
```

- `btd_rnd`

```
U = btd_rnd([100, 100, 100], {[5,6,7], [3 2 1]});
```

Functions and algorithms

- BTD format

```
% U = { {U^(1,1), ..., U^(1,N), S^(1)}, ...,
%        {U^(R,1), ..., U^(R,N), S^(R)} }
```

- `btd_rnd`

```
U = btd_rnd([100, 100, 100], {[5,6,7], [3 2 1]});
```

- `btd_nls` and `btd_minf`

```
[U, output] = btd_nls(T, Uinit, options);
```

Block term decomposition

"... BTD is the most important decomposition in the future. I am going to work on it." – *Frederik*

$(L_r, L_r, 1)$ decomposition

- Objective function

$$\min \frac{1}{2} \left\| \mathcal{T} - \sum_{r=1}^{R} (\mathbf{A}_r \mathbf{B}_r^{\mathsf{T}}) \otimes \mathbf{c}_r \right\|^2$$

$(L_r, L_r, 1)$ decomposition

- Objective function

$$\min \frac{1}{2} \left\| \mathcal{T} - \sum_{r=1}^{R} (\mathbf{A}_r \mathbf{B}_r^{\mathsf{T}}) \otimes \mathbf{c}_r \right\|^2$$

- How to compute?
  - It is a BTD
    - `btd_nls`
  - It is a CPD (with structure)
    - `sdf_nls`
  - It is an LL1 decomposition
    - `ll1_gevd`, `ll1_nls`, `ll1_cpd`

How to compute: BTD

- Matlab

```
L = [3 3 2]
R = arrayfun(@(l) [l l 1], L, 'uni', 0);
Uinit = btd_rnd(size(T), R);

Ures = btd_nls(T, Uinit, options);
```

- Still a core tensor

How to compute: SDF

```
LL1 = @(z,state) struct_LL1(z,state,L);
size_tens = size(T)
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', LL1};
model.factorizations.cpd.data = T;
model.factorizations.cpd.cpd = {'A', 'B', 'C'};

[sol, output] = sdf_nls(model, options);
uvar = struct2cell(sol.variables);
usol = struct2cell(sol.factors);

Trec = cpdgen(usol);
```

How to compute: LL1

experimental

```
Uinit = ll1_rnd(size(T), L);
Ures = ll1_nls(T, Uinit, options);
Ures = ll1_cpd(T, Uinit, options);

Trec = btdgen(Ures);
```

Discussion

	Speed	Convergence	Coding	Maturity
`btd`			🥈	🥇
`sdf`	🥈			🥇
`ll1_nls`	🥈	🥇	🥇	
`ll1_cpd`	🥇	🥈	🥇	

Tips

- Use `lll_gevd`:

```
[U, output] = lll_gevd(T, L);
```

⚠ check if `output.L = L`

Tips

- Use `lll_gevd`:

```
[U, output] = lll_gevd(T, L);
```

⚠ check if `output.L = L`

- Use multiple initializations (not only `lll_gevd`)

Tips

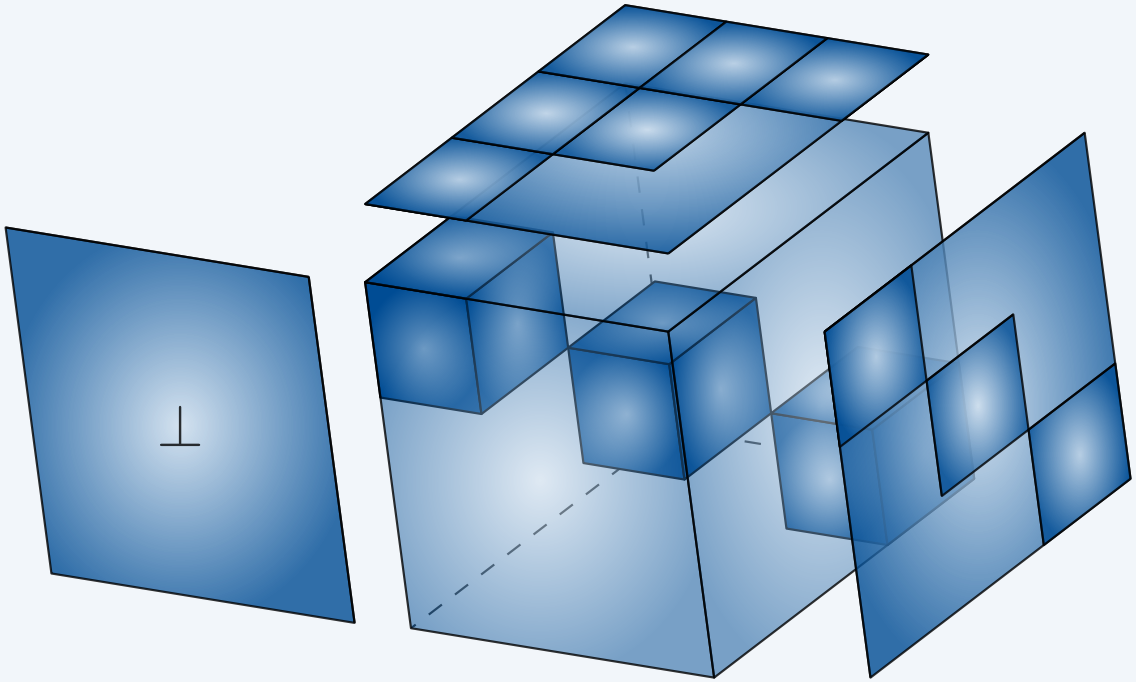
- Use `lll_gevd`:

```
[U, output] = lll_gevd(T, L);
```

⚠ check if `output.L = L`

- Use multiple initializations (not only `lll_gevd`)
- If `T` is large or noisy, compress first: `lmlra_aca`, `lmlra_rff`, `mlsvd`

Structured Data Fusion



Analysis of multiple datasets

- Suppose

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad \text{with} \quad f(\mathbf{x}) = \sum_{d=1}^{D} \omega_d f^{(d)}(\mathbf{x})$$

$$f^{(d)}(\mathbf{x}) = \sum_{d=1}^{D} \frac{1}{2} \left\| \mathcal{M}^{(d)}(\mathbf{x}) - \mathcal{T}^{(d)} \right\|^2$$

in which $\mathcal{M}^{(d)}(\mathbf{x})$ is a CPD, BTD, LMLRA, ...

Analysis of multiple datasets

- Suppose

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad \text{with} \quad f(\mathbf{x}) = \sum_{d=1}^{D} \omega_d f^{(d)}(\mathbf{x})$$

$$f^{(d)}(\mathbf{x}) = \sum_{d=1}^{D} \frac{1}{2} \left|\left| \mathcal{M}^{(d)}(\mathbf{x}) - \mathcal{T}^{(d)} \right|\right|^2$$

  in which $\mathcal{M}^{(d)}(\mathbf{x})$ is a CPD, BTD, LMLRA, ...

- The derivative of a sum is the sum of the derivatives:

$$\nabla_{\mathbf{x}} f = \sum_{d=1}^{D} \omega_d \nabla_{\mathbf{x}} f^{(d)}$$

Implementation of constraints

- Suppose $\mathbf{x}$ is a function of $\mathbf{z}$:

$$\min_{\mathbf{z}} \frac{1}{2} ||\mathcal{M}(\mathbf{x}(\mathbf{z})) - \mathcal{T}||^2$$

Implementation of constraints

- Suppose $\mathbf{x}$ is a function of $\mathbf{z}$:

$$\min_{\mathbf{z}} \frac{1}{2} ||\mathcal{M}(\mathbf{x}(\mathbf{z})) - \mathcal{T}||^2$$

- The chain rule applies

$$(\nabla_{\mathbf{z}} f)^{\mathsf{T}} = \nabla_{\mathbf{x}} f \cdot \nabla_{\mathbf{z}} \mathbf{x}$$
$$\mathbf{J}_{\mathbf{z}} = \mathbf{J}_{\mathbf{x}} \nabla_{\mathbf{z}} \mathbf{x}$$

Implementation of constraints

- Suppose $\mathbf{x}$ is a function of $\mathbf{z}$:

$$\min_{\mathbf{z}} \frac{1}{2} ||\mathcal{M}(\mathbf{x}(\mathbf{z})) - \mathcal{T}||^2$$

- The chain rule applies

$$(\nabla_{\mathbf{z}} f)^{\mathsf{T}} = \nabla_{\mathbf{x}} f \cdot \nabla_{\mathbf{z}} \mathbf{x}$$
$$\mathbf{J}_{\mathbf{z}} = \mathbf{J}_{\mathbf{x}} \nabla_{\mathbf{z}} \mathbf{x}$$

- The system of equations to compute the step becomes:

$$(\nabla_{\mathbf{z}} \mathbf{x})^{\mathsf{H}} \left(\mathbf{J}_{\mathbf{x}}^{\mathsf{H}} \mathbf{J}_{\mathbf{x}}\right) (\nabla_{\mathbf{z}} \mathbf{x}) \, \mathbf{p} = -\mathbf{g}$$

The structure of a structure

```
function [x, state] = struct_log(z, task)
if isempty(task) || (isempty(task.l) && isempty(task.r))
    % Evaluation of x(z)
    x = log(z);
    state.deriv = 1./z;
elseif ~isempty(task.r)
    % Computation of (∇_z x)*task.r, task.r = p
    x = task.deriv.*task.r;
elseif ~isempty(task.l)
    % Computation of (∇_z x)^H*state.l
    %  1. state.l = ∇_x f
    %  2. state.l = ((J_x^H J_x)(∇_z x) p)
    x = conj(task.deriv).*task.l;
end
```

The structure of a structure

```
function [x, state] = struct_log(z, task)
if isempty(task) || (isempty(task.l) && isempty(task.r))
    % Evaluation of x(z)
    x = log(z);
    state.deriv = 1./z;
elseif ~isempty(task.r)
    % Computation of (∇_z x)*task.r, task.r = p
    x = task.deriv.*task.r;
elseif ~isempty(task.l)
    % Computation of (∇_z x)^H*state.l
    %  1. state.l = ∇_x f
    %  2. state.l = ((J_x^H J_x)(∇_z x) p)
    x = conj(task.deriv).*task.l;
end
```

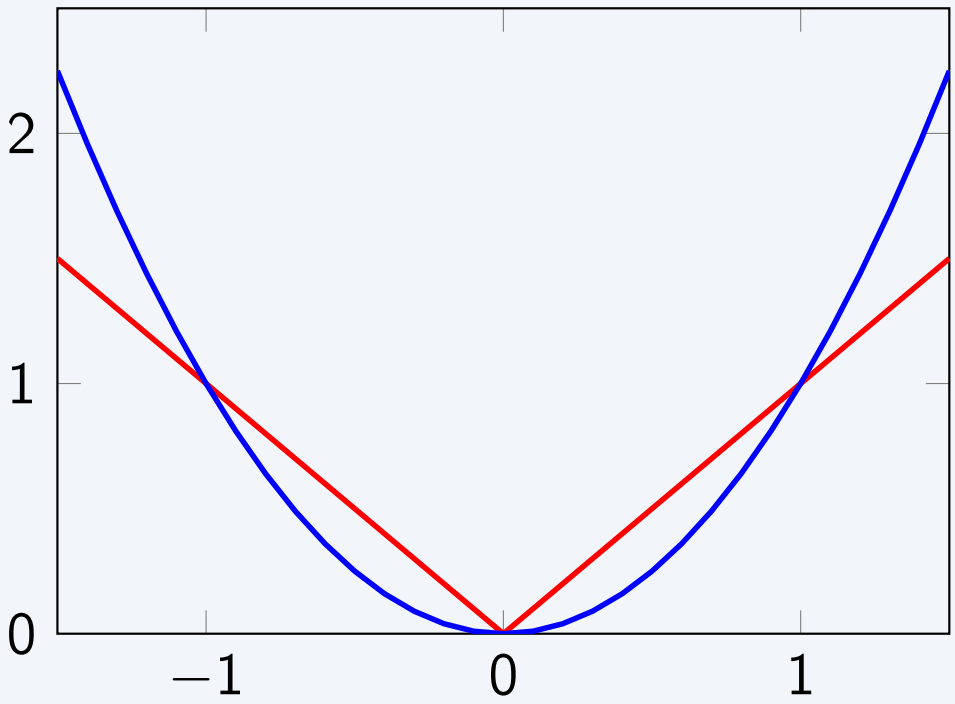
- `state.l/r` contains relevant part of factor matrix
- A little bit more complex for complex numbers

Using SDF

```
model.variables.a = randn(10, 5);
model.variables.b = randn(5, 5);
model.variables.c = randn(10, 1);
model.factors.A = {'a', rand(10,5)};
model.factors.B = {1, {2, @struct_nonneg}; rand(5,5)};
model.factors.B = {1, {1, @struct_nonneg}};
model.factors.C = {{'c', @struct_log, @struct_diag}};

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

[sol, output] = sdf_nls(model, options);
uvar = struct2cell(sol.variables);
usol = struct2cell(sol.factors);
```

Regularization

```
model.factorizations.myreg.regL1 = {'A'};
model.factorizations.yourreg.regL2 = {2};
model.factorizations.yourreg.data = ones(10,10);
```



Setting options

- Similar as before

Setting options

- Similar as before
- Choice of weights $\omega_d = \frac{\bar{\omega}_d}{2\text{numel}(\mathcal{T}^{(d)})}$

```
options.RelWeights = [10 100 1];
```

Workflow for SDF

Main idea: initialize using solution of relaxed problem

Workflow for SDF

Main idea: initialize using solution of relaxed problem

- Constraints
  1. Solve unconstrained problem
  2. Estimate parameters
  3. Add constraint and solve again

Workflow for SDF

Main idea: initialize using solution of relaxed problem

- Constraints
  1. Solve unconstrained problem
  2. Estimate parameters
  3. Add constraint and solve again
  4. Repeat until all constraints are added

Workflow for SDF

Main idea: initialize using solution of relaxed problem

- Constraints
  1. Solve unconstrained problem
  2. Estimate parameters
  3. Add constraint and solve again
  4. Repeat until all constraints are added
- Data fusion
  1. Solve problem for one dataset
  2. Estimate remaining factors keeping known factors fixed
  3. Re-estimate fixed factors
  4. Repeat

Workflow for SDF

Main idea: initialize using solution of relaxed problem

- Constraints
  1. Solve unconstrained problem
  2. Estimate parameters
  3. Add constraint and solve again
  4. Repeat until all constraints are added
- Data fusion
  1. Solve problem for one dataset
  2. Estimate remaining factors keeping known factors fixed
  3. Re-estimate fixed factors
  4. Repeat
- But only do this when no (good) direct solution can be found

Harmonic retrieval example

- Sequence of planar wave fronts measured on a $I \times J$ grid of sensors for K samples

$$t_{ijk} = \sum_{r=1}^{R} s_{kr} \mathrm{e}^{\jmath \alpha_r (i-1)} \mathrm{e}^{\jmath \beta_r (j-1)}$$

- Written as a CPD

$$\mathcal{T} = [\![\mathbf{A}, \mathbf{B}, \mathbf{S}]\!]$$

in which $\mathbf{A}$ and $\mathbf{B}$ have a Vandermonde structure

$$\mathbf{a}_r = \begin{bmatrix} 1 & \mathrm{e}^{\jmath \alpha_r} & \dots & \mathrm{e}^{\jmath \alpha_r (I-1)} \end{bmatrix}^{\mathsf{T}}$$

Harmonic retrieval: SDF

```
vandermonde = @(z, state) ...
              struct_vander(z, state, [0 sz(1)-1]);
model.variables.a = randn(1, R) + randn(1, R) * 1i;
model.variables.b = randn(1, R) + randn(1, R) * 1i;
model.variables.s = randn(sz(3), R) + randn(sz(3),R)*1i;
model.factors.A = {'a', vandermonde, @struct_transpose};
model.factors.B = {'b', vandermonde, @struct_transpose};
model.factors.S = {'s'};
model.factorizations.hr.data = Td;
model.factorizations.hr.cpd = {'A', 'B', 'S'};

[sol, output] = sdf_nls(model, options);
uvar = struct2cell(sol.variables);
usol = struct2cell(sol.factors);
```

Harmonic retrieval: SDF

```
        fval                relfval            relstep
        =1/2*norm(F)^2      TolFun = 1e-08     TolX = 1e-08

   0:   3.36998907e+04 |
   1:   3.33852257e+03 | 9.00933727e-01 | 3.000000e-01
  20:   1.38581644e+00 | 5.16779039e-07 | 1.637907e-02
  40:   1.27729165e+00 | 5.00268067e-08 | 8.947791e-03
  60:   1.25247376e+00 | 1.92818795e-08 | 3.601389e-03
  80:   1.23919216e+00 | 1.58284013e-08 | 3.639052e-03
 100:   1.23055086e+00 | 1.08583537e-08 | 2.393667e-03
 104:   1.22912806e+00 | 9.92786005e-09 | 2.178808e-03

err =
    0.8311    0.7563    0.5319
```

Harmonic retrieval: CPD + SDF

```
Uinit = cpd_rnd(Td, R, struct('Imag', @randn));
Ures = cpd_nls(Td, Uinit, options);

alpha_est = mean(Ures{1}(2:end,:)./Ures{1}(1:end-1,:));
beta_est = mean(Ures{2}(2:end,:)./Ures{2}(1:end-1,:));
S_est = Ures{3};

model.variables.a = alpha_est;
model.variables.b = beta_est;
model.variables.s = S_est;
```

Harmonic retrieval: CPD + SDF

```
        fval                relfval            relstep
        =1/2*norm(F)^2      TolFun = 1e-08     TolX = 1e-08

   0:   1.33180211e+03 |
   1:   1.30137568e+03 | 2.28460581e-02 | 3.000000e-01
  12:   9.42652003e-10 | 1.79934766e-10 | 8.970192e-05

err = [0.6386    0.6573    0.0000]

   0:   2.53718572e+00 |
   7:   7.60910370e-11 | 2.74361409e-09 | 1.175042e-04

err = 1.0e-05 * [0.1585    0.1230    0.2147]
```

Non-negativity constraints: SDF

```
Uinit = cpd_rnd(Tn, R, 'real', @rand, 'orth', false);

model.variables = cellfun(@(u) sqrt(u), Uinit, 'uni', 0);
model.factors.A = {1, @struct_nonneg};
model.factors.B = {2, @struct_nonneg};
model.factors.C = {3, @struct_nonneg};
model.factorizations.cpd.data = Tn;
model.factorizations.cpd.cpd = {'A', 'B', 'C'};

[sol, output] = sdf_nls(model);
Usdf = struct2cell(sol.factors);
```

Non-negativity constraints: CPD

```
LB = cellfun(@(u) zeros(size(u)), Uinit, 'uni', 0);
UB = cellfun(@(u) inf(size(u)), Uinit, 'uni', 0);

options = struct;
options.Algorithm = @(F,dF,z0,options) ...
    nlsb_gndl(F, dF, LB, UB, z0, options);

[Ures, output] = cpd_nls(Tn, Uinit, options);
```

Non-negativity constraints

A little experiment:

- $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathcal{U}(0,1)^{200\times 10}$
- $\mathcal{T} = [\![\mathbf{A}, \mathbf{B}, \mathbf{C}]\!]$ with rank 10

	`sdf`	`nlsb`	`nls`
Iterations	73	30	11
Time (s)	5.75	1.80	0.70
Time/it (ms)	79	60	63

Tips and tricks

- Use solutions to relaxed problems as initialization
- Be aware of alternatives if speed is important

Random remarks

- Be careful when using `tic` and `toc`

```
timer = tic; cpd(...); toc(timer)
```

Random remarks

- Be careful when using `tic` and `toc`

```
timer = tic; cpd(...); toc(timer)
```

- Use `noisy`, `cpdgen`, `cpderr`, ...

Random remarks

- Be careful when using `tic` and `toc`

```
timer = tic; cpd(...); toc(timer)
```

- Use `noisy`, `cpdgen`, `cpderr`, ...
- Master `cellfun`

```
U = cellfun(@(u) u(:,1), U, 'uni', 0);
```

Random remarks

- Be careful when using `tic` and `toc`

```
timer = tic; cpd(...); toc(timer)
```

- Use `noisy`, `cpdgen`, `cpderr`, ...
- Master `cellfun`

```
U = cellfun(@(u) u(:,1), U, 'uni', 0);
```

- Get the development version from git
  - Register at bitbucket.org
  - Send username to me
  - Clone tensorlab

Workflow

1. Be lazy
2. Try other parameters
3. Try other initialization techniques
4. Try other algorithms