

# Tensor Node Notation

## Alternating Optimization

### Solving a linear equation

In the following, we will look at the solution of linear systems given in tensor networks

$$\text{find } \min_N \|A[\times]N - T\|$$

The single networks  $A$ ,  $N$  and  $T$  will be assumed to be plain, and that no single node has duplicate mode names.

First we construct  $A = A(\omega, \alpha)$  (with inner mode names  $\gamma$ ) as finite differences Laplacian in the TT-format, which has rank 2, and then approximate it with an HT-representation.

```
d = 6;
n_discr = 30;

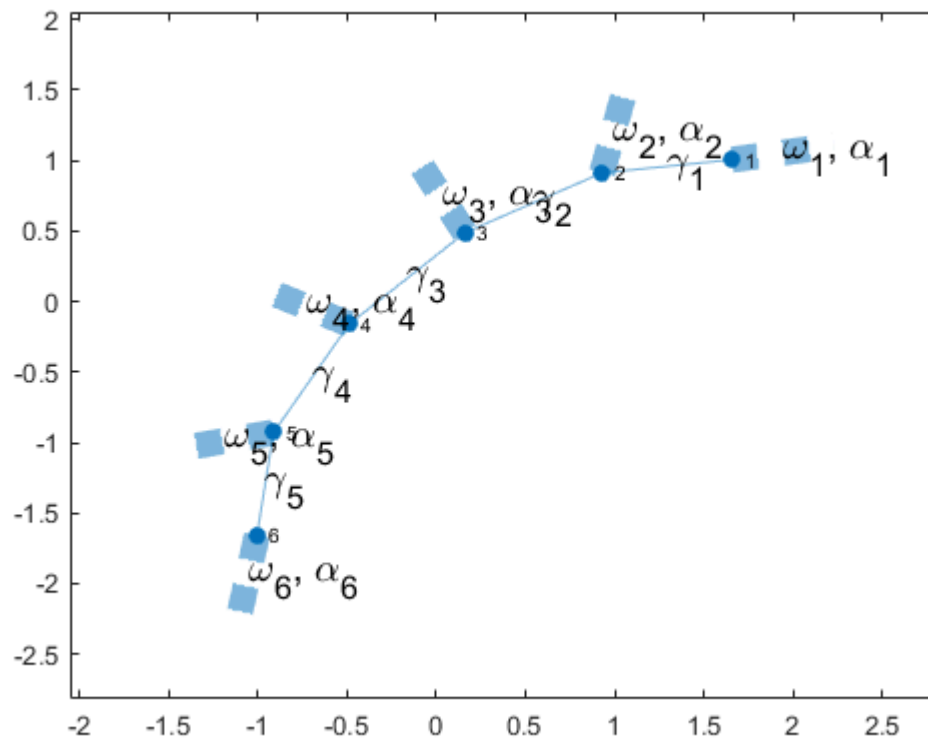
alpha = mna('alpha',1:d);
omega = mna('omega',1:d);
gamma = mna('gamma',1:d-1);
n_alpha = assign_mode_size(alpha,n_discr);
n_omega = assign_mode_size(omega,n_discr);
n_gamma = assign_mode_size(gamma,2);
n = merge_fields(n_alpha,n_omega,n_gamma);

% Parts in Laplacian:
D2 = gallery('tridiag',n_discr,-1,2,-1);
I = eye(n_discr);
O = zeros(n_discr);

mn = cell(1,d);
A = cell(1,d);

mn{1} = [omega(1),alpha(1),gamma(1)];
A{1} = fold([I,D2],mn{1},n);
for i = 2:d-1
    mn{i} = [omega(i),gamma(i-1),alpha(i),gamma(i)];
    A{i} = fold([I,D2;O,I],mn{i},n);
end
mn{d} = [omega(d),gamma(d-1),alpha(d)];
A{d} = fold([D2;I],mn{d},n);

net_view(A)
```



We can easily determine the TT-singular values of  $A$ :

```
[~,fA,sigmaA] = STAND(A);
get_data(sigmaA{1})
```

```
ans = 2x1
105 ×
    3.3667
    0.0936
```

```
get_data(sigmaA{2})
```

```
ans = 2x1
105 ×
    3.3659
    0.1184
```

```
get_data(sigmaA{3})
```

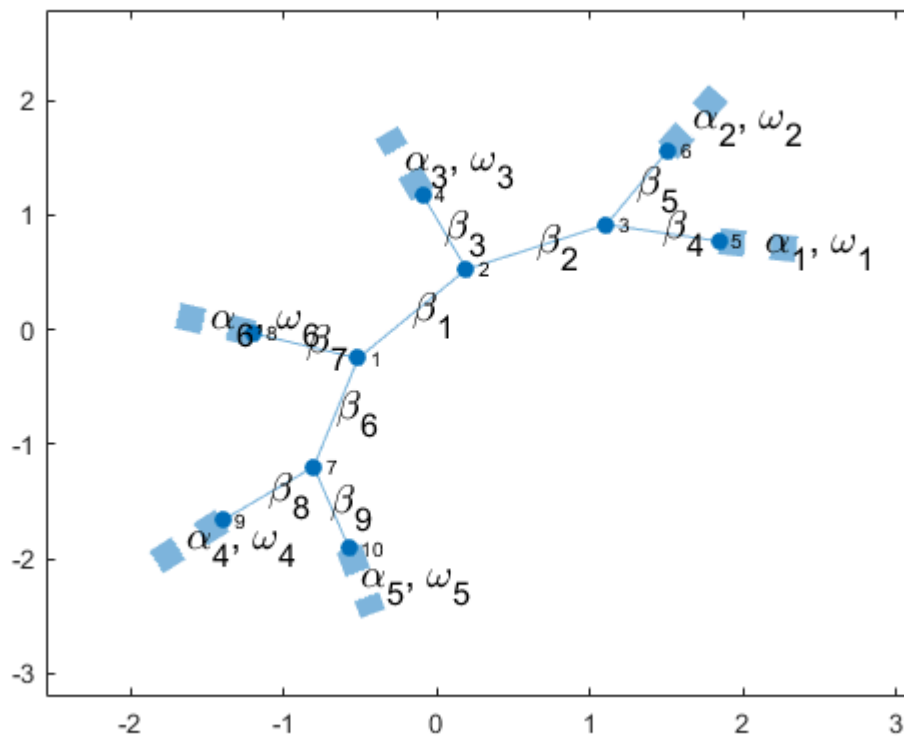
```
ans = 2x1
105 ×
    3.3656
    0.1256
```

**Approximation through an HT-representation**

A is given as TT-representation, but with a simple alternating least squares scheme, we may transfer it to any other tree tensor format of rank 2:

```
S = binary_tree([alpha;omega]);
[G,m,~,~,beta] = RTLGRAPH([alpha,omega],S,'beta');

n_beta = assign_mode_size(beta,2);
n = merge_fields(n,n_beta);
AHT = init_net(m,n);
AHT = randomize_net(AHT);
net_view(AHT)
```



The simple ALS algorithm performs `iter_max` steps in order to minimize  $\|N - T\|$ .

type `ALS.m`

```
function N = ALS(N,T,iter_max,r,G)
% ALS is a Vanilla implementation of alternating least squares
%
% N = ALS(N,T,iter_max) performs iter_max steps of alternating least
% squares in order to minimize \|N - T\|.
%
% It solely uses orthogonalizations in order to simplify the calculation.
%
% See also: TENSOR_NODE_NOTATION10_ALTERNATING_OPTIMIZATION.mlx, ORTHO,
% PATHQR, LINSOLVE

if nargin <= 3
```

```

    r = 1;
end

if nargin <= 4
    G = net_derive_G(N);
end

N = ORTHO(N,r);
h_crt = r;

for iter = 1:iter_max
    ALSREC(r,[]);
end

function ALSREC(b,P)
    for h = setdiff(neighbors(G,b)',P)
        ALSREC(h,b)
    end

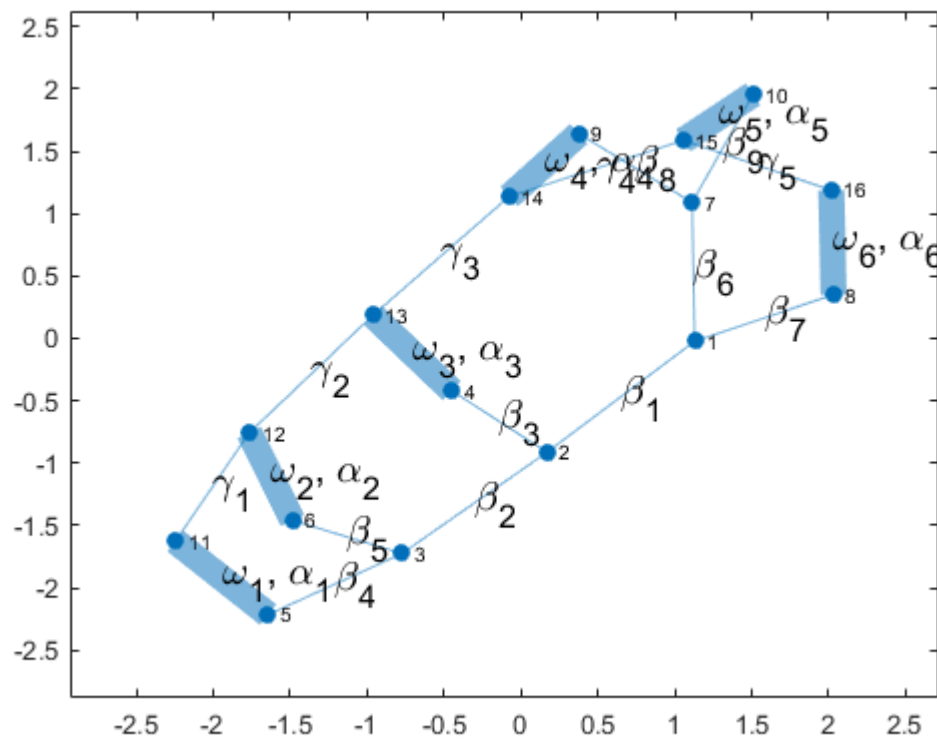
    [N,h_crt] = PATHQR(N,h_crt,b,G);

    all_but_b = true(size(N));
    all_but_b(b) = false;
    N{b} = boxtimes(N(all_but_b),T);
end

end

```

```
net_view(AHT,A)
```



```
AHT = ALS(AHT,A,1) % one iteration is already sufficient since the rank is exactly 2
```

```
AHT = 1×10 cell array
      {1×1 struct}      {1×1 struct}      {1×1 struct}      {1×1 struct}      {1×1 struct}      {1×1 struct}      {1×1 struct}
net_dist(AHT,A)/net_norm(A)
```

```
ans = 3.4795e-08
```

```
AHT = ALS(AHT,A,3) % does not help
```

```
AHT = 1×10 cell array
      {1×1 struct}      {1×1 struct}      {1×1 struct}      {1×1 struct}      {1×1 struct}      {1×1 struct}      {1×1 struct}
net_dist(AHT,A)/net_norm(A)
```

```
ans = 2.5934e-08
```

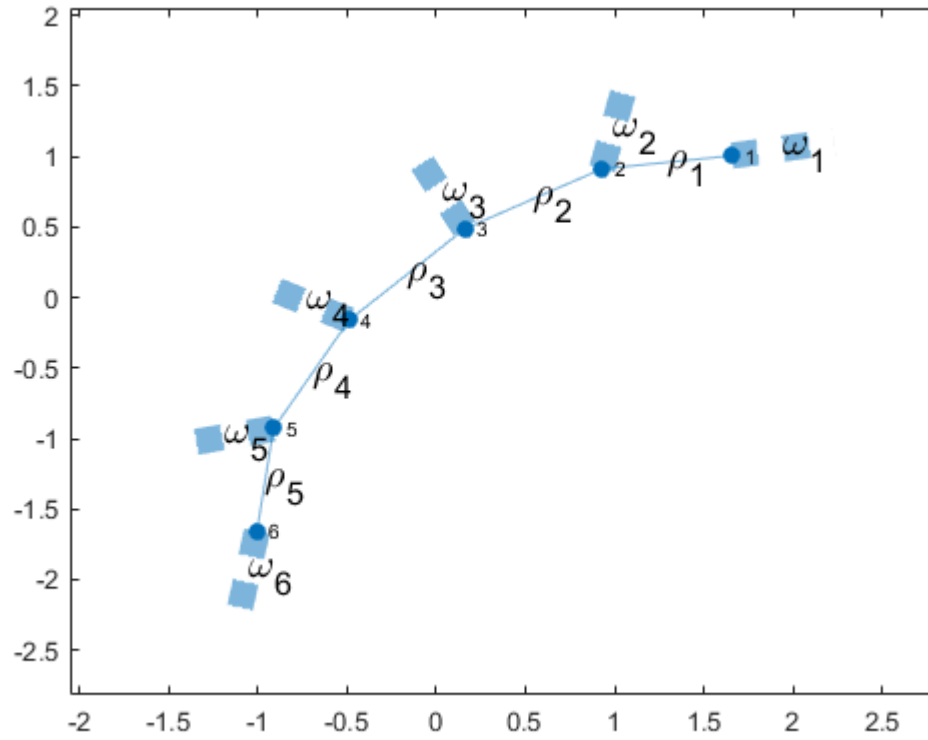
One might of course construct  $A$  as HT-representation directly in order to avoid round-off errors.

## Solving a linear system

We can solve any linear minimization problem  $\|A \begin{bmatrix} x \end{bmatrix} N - T\| \rightarrow \min$  using `LINSOLVE`, however under the slight mode name assumptions mentioned above. Here  $T$  will be rank one and would not need inner mode names. But we will assign such for a better overview. Each single micro step will make use of the normal equation. The correct structure of the network is further ensured using ghost nodes. `LINSOLVE` is however unreasonable for higher order tensors, since no computations are recycled (other than for `LINSOLVE_EQUALNET`).

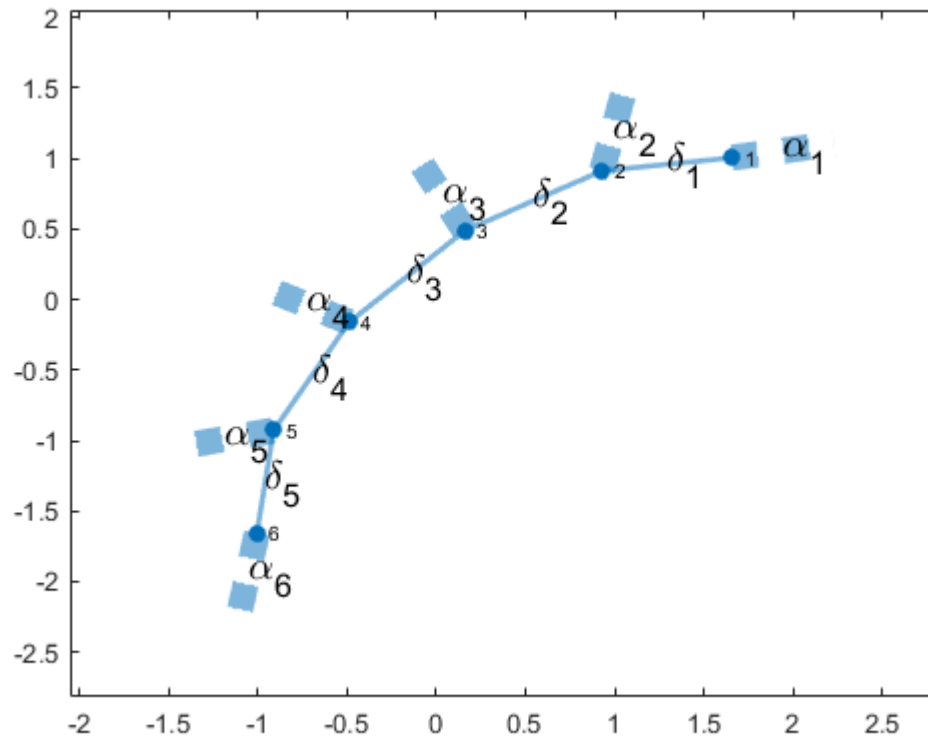
```
S = linear_tree(omega);
[G,m,~,~,rho] = RTLGRAPH(omega,S,'rho');
n_rho = assign_mode_size(rho,1);
n = merge_fields(n,n_rho);
T = init_net(m,n);
for i = 1:length(T)
    T{i}.data(:) = 1;
end

net_view(T)
```



For  $N = N(\alpha)$ , we use a TT-representation with rank 5 and inner mode names  $\delta$ :

```
S = linear_tree(alpha);
[G,m,~,~,beta] = RTLGRAPH(alpha,S,'delta');
n_beta = assign_mode_size(beta,6);
n = merge_fields(n,n_beta);
N = init_net(m,n);
N = randomize_net(N);
net_view(N)
```



type [LINSOLVE](#)

```
function N = LINSOLVE(A,N,T,iter_max,r,G)
% LINSOLVE_EQUALNET solves boxtimes(A,N) = T for arbitrary networks A,N,T
%
% LINSOLVE_EQUALNET(A,N,T,iter_max) performs alternating least squares
% for fixed rank to minimize ||boxtimes(A,N) - T||_F for N.
%
% Should not be used for too large networks.
%
% See also: TENSOR_NODE_NOTATION10_ALTERNATING_OPTIMIZATION.mlx,
% LINSOLVE_EQUALNET, ORTHO, PATHQR

if nargin <= 4
    r = 1;
end

if nargin <= 5
    G = net_derive_G(N);
end

N = ORTHO(N,r);
h_crt = r;

for iter = 1:iter_max
    ALSREC(r,[]);
end

function ALSREC(b,P)
    for h = setdiff(neighbors(G,b)',P)
        ALSREC(h,b)
    end
```

```

[N,h_crt] = PATHQR(N,h_crt,b,G); % for stability

% solve for N(b): boxtimes(A,[N(all_but_b),N(b)]) = T
all_but_b = ~(1:numel(N)==b);

H = boxtimes({A,N(all_but_b),ghost(N{b})},{A,N(all_but_b),ghost(N{b})});
B = boxtimes({A,N(all_but_b),ghost(N{b})},T);

N{b} = node_linsolve(H,B);
end

end

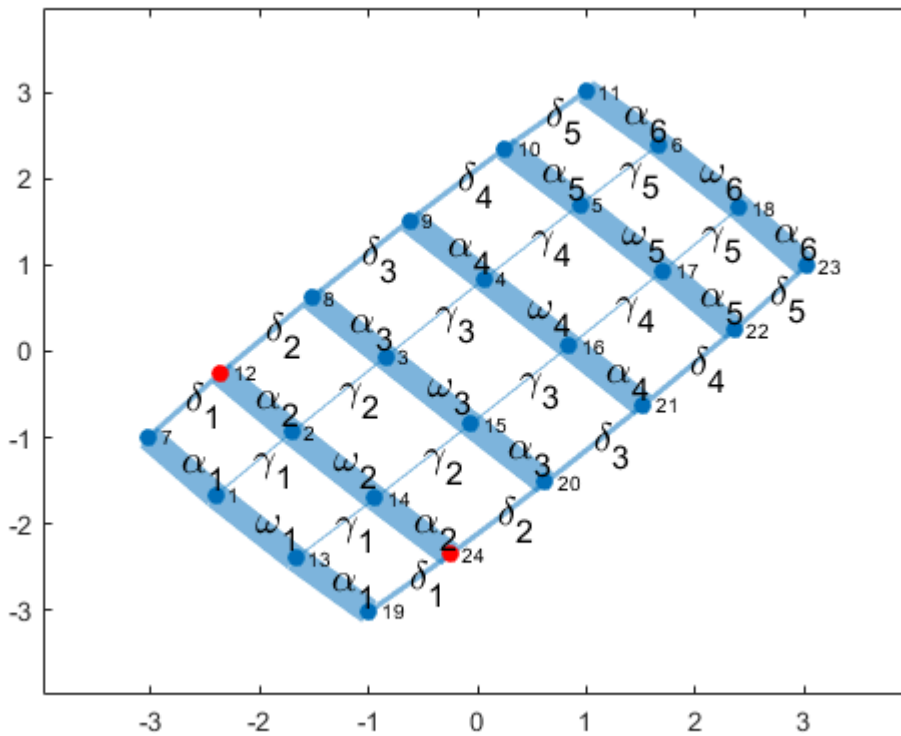
```

```

b = 2;
all_but_b = ~(1:numel(N)==b);

net_view({A,N(all_but_b),ghost(N{b})},{A,N(all_but_b),ghost(N{b})}); % here with A in a

```



```

boxtimes({A,N(all_but_b),ghost(N{b})},{A,N(all_but_b),ghost(N{b})})

```

```

ans = struct with fields:

```

```

    mode_names: {'alpha_2' 'delta_1' 'delta_2' 'alpha_2' 'delta_1' 'delta_2'}
           pos: [1x1 struct]
          data: [6-D double]

```

```

save('Laplacian','AHT','A','N','T')
deactivate_boxtimes_mem()
tic
N = LINSOLVE(AHT,N,T,20)

```



```

N = 1×6 cell array
    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1 struct}

```

```
seconds_without_memory = toc
```

```
seconds_without_memory = 19.4592
```

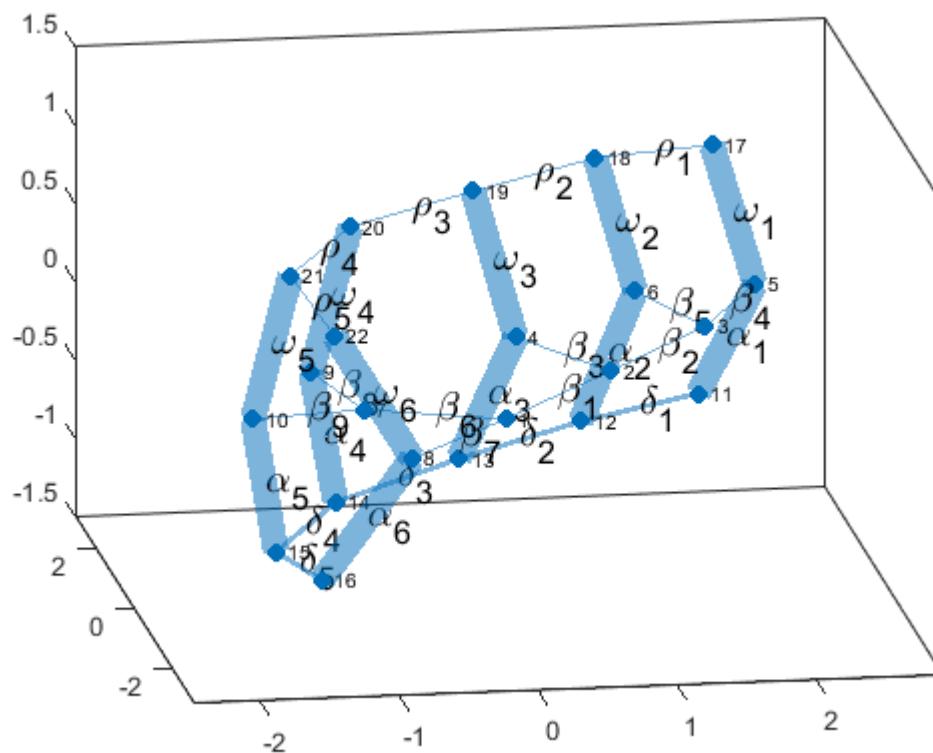
```
net_dist({AHT,N},T)/sqrt(get_data(boxtimes(T,T)))
```

```
ans = 1.6216e-05
```

```

net_view({AHT,N},T,'Layout','force3')
view([-4.70 2.00])
view([-9.10 22.00])

```



## Speed up with boxtimes memory:

```

load('Laplacian','AHT','A','N','T')
activate_boxtimes_mem()
tic
N = LINSOLVE(AHT,N,T,20)

```

```

N = 1×6 cell array
    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1 struct}

```

```
toc
```

```
Elapsed time is 6.797446 seconds.
```

```
seconds_without_memory
```

```
seconds_without_memory = 19.4592
```

```
net_dist({AHT,N},T)/sqrt(get_data(boxtimes(T,T))) % same result as above
```

```
ans = 1.6216e-05
```

## Linear equations in equal networks

If the involved networks are all of equal structure, then we can use the faster, more complicated `LINSOLVE_EQUALNET`, which makes use of the normal equation in each one of the nodes:

$$N_i^* := \operatorname{argmin}_{N_i} \|A \begin{bmatrix} \times \\ N \end{bmatrix} - T\| = \operatorname{argmin}_{N_i} \|A \begin{bmatrix} \times \\ N_{\neq i} \end{bmatrix} \begin{bmatrix} \times \\ N_i \end{bmatrix} - T\|$$

where

$$(A \begin{bmatrix} \times \\ N_{\neq i} \end{bmatrix})^T \begin{bmatrix} \times \\ N_i \end{bmatrix} = (A \begin{bmatrix} \times \\ N_{\neq i} \end{bmatrix} \begin{bmatrix} \times \\ N_i^* \end{bmatrix})^T \begin{bmatrix} \times \\ T \end{bmatrix}$$

Unfortunately, we can not resolve the brackets in the lower equation since some mode names will appear twice. This problem can be resolved by renaming mode names until each contraction is uniquely identified by one mode name. We therefore change

$$A(\omega, \alpha), N(\alpha), T(\omega)$$

to

$$A'(\omega, \alpha'), N'(\alpha'), A(\omega, \alpha), N(\alpha), T(\omega)$$

where  $A'$  and  $N'$  are defined to always have the same value as  $A$  and  $N$ , but just with different mode names. We can then restate

$$(A \begin{bmatrix} \times \\ N_{\neq i} \end{bmatrix})^T \begin{bmatrix} \times \\ N_i \end{bmatrix} = (A \begin{bmatrix} \times \\ N_{\neq i} \end{bmatrix} \begin{bmatrix} \times \\ N_i^* \end{bmatrix})^T \begin{bmatrix} \times \\ T \end{bmatrix} \Leftrightarrow N'_{\neq i} \begin{bmatrix} \times \\ A' \end{bmatrix} \begin{bmatrix} \times \\ A \end{bmatrix} \begin{bmatrix} \times \\ N_{\neq i} \end{bmatrix} \begin{bmatrix} \times \\ N_i^* \end{bmatrix} = N'_{\neq i} \begin{bmatrix} \times \\ A' \end{bmatrix} \begin{bmatrix} \times \\ T \end{bmatrix}$$

The mode names in both sides of the right-hand side equation then allow to arbitrarily disassemble each of the two products and evaluate parts of it branch-wise. In the code, the renaming works as follows. Instead of names such as  $\alpha'$ , the mode names in `LINSOLVE_EQUALNET` will be generic  $\mu_1, \mu_2, \dots$ . Sometimes one might therefore prefer to rename modes name manually.

```
type LINSOLVE_EQUALNET.m
```

```
function N = LINSOLVE_EQUALNET(A,N,T,iter_max,r,G)
% LINSOLVE_EQUALNET solves boxtimes(A,N) = T for equal networks A,N,T
%
% LINSOLVE_EQUALNET(A,N,T,iter_max) performs alternating least squares
% for fixed rank to minimize ||boxtimes(A,N) - T||_F for N.
%
% The algorithm makes use of the equal structure of networks and computes
% the normal equations branch-wise.
```

```

%
% See also: TENSOR_NODE_NOTATION10_ALTERNATING_OPTIMIZATION.mlx,
% LINSOLVE, ORTHO, PATHQR

%% input
if nargin <= 4
    r = 1;
end
if nargin <= 5
    G = net_derive_G(N);
end

%% apply unique identifiers (and later use [N_prime{:}] = derive_net(prime_rn,N); )
nF = length(T);
nA = length(A);
nN = length(N);

% rename mode names in main equation AN = M
Net = {T, {A,N}};
[Z,glob_repl,outer_mn_id,seed] = get_mn_replacements(Net);
[Z,~,renaming1] = rename_mn(Z,glob_repl,outer_mn_id);
T = Z(1:nF);
A = Z(nF+1:nF+nA);
N = Z(nF+nA+1:end);

% derive nodes in AN with different inner mode names
Net = {A,N};
[Z,glob_repl,outer_mn_id] = get_mn_replacements(Net,seed);
[Z,~,~,prime_rn] = rename_mn(Z,glob_repl,outer_mn_id);
A_prime = Z(1:nA);
N_prime = Z(nA+1:end);

%% branch-wise products
Br = branches(G,r);
Br_prod_ANTAN_towards_root = cell(1,numnodes(G));
Br_prod_ANTAN_away_from_root = cell(1,numnodes(G));
Br_prod_ANTM_towards_root = cell(1,numnodes(G));
Br_prod_ANTM_away_from_root = cell(1,numnodes(G));

Br_prod_ANTAN_towards_root{r} = [];
Br_prod_ANTM_towards_root{r} = [];

%% M^T M
MTM_node = boxtimes(T,T);
MTM = MTM_node.data;

%% Pre-computations
is_iter_zero = true; % ortho to r and record Br_prod 2, dont optimize
ALSREC(r,[]);
is_iter_zero = false;

for iter = 1:iter_max
    ALSREC(r,[]);

    ANTAN = Br_prod_ANTAN_away_from_root{r}.data;
    ANTM = Br_prod_ANTM_away_from_root{r}.data;
    relres = sqrt(ANTAN - 2*ANTM + MTM)/sqrt(MTM);
    fprintf('rel res: %.2e\n',relres);
end

[N{:}] = derive_net(renaming1,N{:});

function ALSREC(b,P)
    Neighb = neighbors(G,b)';

```

```

for h = setdiff(neighbors(G,b)',P)

    if ~is_iter_zero
        % root to leaf QR (N is then h-orthogonal)
        [N~,path] = PATHQR(N,b,h,G); % only changes b and h
        [N_prime{path}] = derive_net(prime_rn,N{path});

        % Br_prod_ANTAN_towards_root{b}
        U = int_setdiff(neighbors(G,b)',[P,h]);

        % combine branches
        MulNet = [{}, Br_prod_ANTAN_towards_root{b}, A{b}, A_prime{b}, N(b), N_prime(b), Br_prod_

        Br_prod_ANTAN_towards_root{h} = boxtimes(MulNet{:},'mode','optimal');

        MulNet = [{}, Br_prod_ANTM_towards_root{b}, T{b}, A_prime{b}, N_prime(b), Br_prod_ANTM_away

        Br_prod_ANTM_towards_root{h} = boxtimes(MulNet{:},'mode','optimal');
    end

    % step towards leaf
    ALSREC(h,b) % last updated N_h

end

H = int_setdiff(Neighb,P);

% solve least squares problem in node b via normal equation
if ~is_iter_zero
    % (AN w/o N_b)'*(AN w/o N_b):
    MulNet = [{}, A{b}, A_prime{b}, Br_prod_ANTAN_towards_root{b}, Br_prod_ANTAN_away_from_root{H}];
    ANwoN_bTANwoN_b = boxtimes(MulNet{:},'mode','optimal');

    % (AN w/o N_b)'*T
    MulNet = [{}, T{b}, A_prime{b}, Br_prod_ANTM_towards_root{b}, Br_prod_ANTM_away_from_root{H}];
    ANwoN_bTM = boxtimes(MulNet{:},'mode','optimal');

    % new N_b
    N{b} = node_linsolve(ANwoN_bTANwoN_b,ANwoN_bTM);
    N_prime{b} = derive_net(prime_rn,N{b});
end

if ~isempty(P)
    % leaf to root QR (N is then P-orthogonal)
    [N~,path] = PATHQR(N,b,P,G); % only changes b and P
    [N_prime{path}] = derive_net(prime_rn,N{path});
end

% ghost nodes are not necessary for plain networks
MulNet = [{}, A{b}, A_prime{b}, N(b), N_prime(b), Br_prod_ANTAN_away_from_root{H}];

Br_prod_ANTAN_away_from_root{b} = boxtimes(MulNet{:},'mode','optimal');

MulNet = [{}, A_prime{b}, N_prime(b), Br_prod_ANTM_away_from_root{H}, T{b}];

Br_prod_ANTM_away_from_root{b} = boxtimes(MulNet{:},'mode','optimal');

end

end

load('Laplacian','AHT','A','N','T')
N = randomize_net(N);
nF = length(T);

```

```

nA = length(A);
nN = length(N);
Net = {T,{A,N}};
[Z,glob_repl,outer_mn_id,seed] = get_mn_replacements(Net);
Z

```

```

Z = 1×18 cell array
    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1 struct}    {1×1

```

```

[Z,~,renaming1] = rename_mn(Z,glob_repl,outer_mn_id);
T = Z(1:nF);
A = Z(nF+1:nF+nA);
N = Z(nF+nA+1:end);
renaming1

```

```

renaming1 = struct with fields:

```

```

    mu_1: 'omega_1'
    mu_2: 'rho_1'
    mu_3: 'omega_2'
    mu_4: 'rho_2'
    mu_5: 'omega_3'
    mu_6: 'rho_3'
    mu_7: 'omega_4'
    mu_8: 'rho_4'
    mu_9: 'omega_5'
    mu_10: 'rho_5'
    mu_11: 'omega_6'
    mu_12: 'alpha_1'
    mu_13: 'gamma_1'
    mu_14: 'alpha_2'
    mu_15: 'gamma_2'
    mu_16: 'alpha_3'
    mu_17: 'gamma_3'
    mu_18: 'alpha_4'
    mu_19: 'gamma_4'
    mu_20: 'alpha_5'
    mu_21: 'gamma_5'
    mu_22: 'alpha_6'
    mu_23: 'delta_1'
    mu_24: 'delta_2'
    mu_25: 'delta_3'
    mu_26: 'delta_4'
    mu_27: 'delta_5'

```

```

Net = {A,N};
[Z,glob_repl,outer_mn_id] = get_mn_replacements(Net,seed);
[Z,~,~,prime_rn] = rename_mn(Z,glob_repl,outer_mn_id);
A_prime = Z(1:nA);
N_prime = Z(nA+1:end);
prime_rn

```

```

prime_rn = struct with fields:

```

```

    mu_12: 'mu_30'
    mu_13: 'mu_31'
    mu_14: 'mu_33'
    mu_15: 'mu_34'
    mu_16: 'mu_36'
    mu_17: 'mu_37'

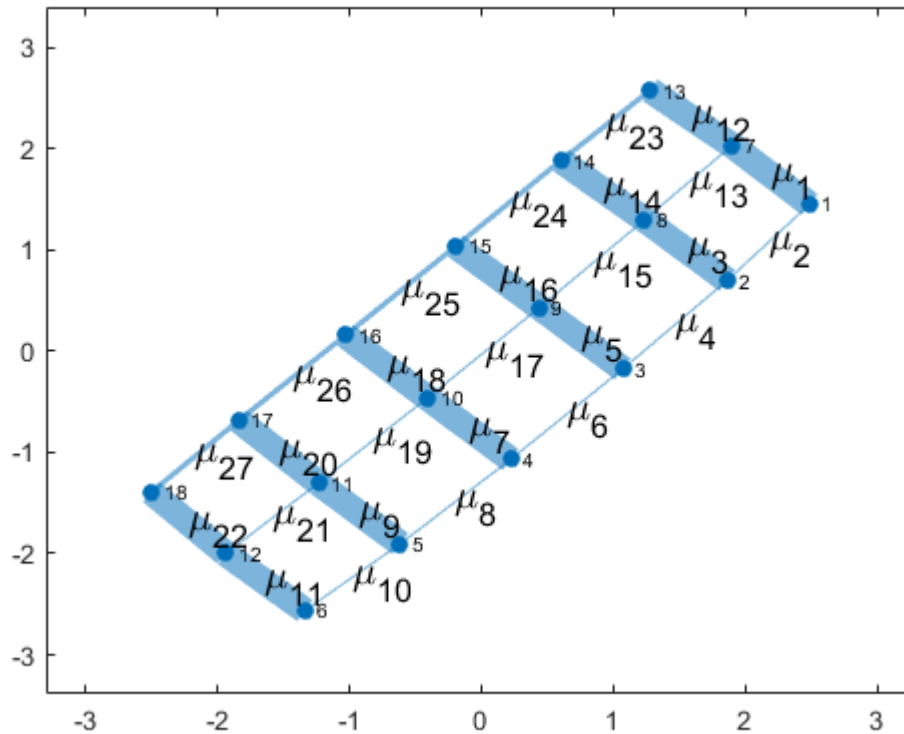
```

```

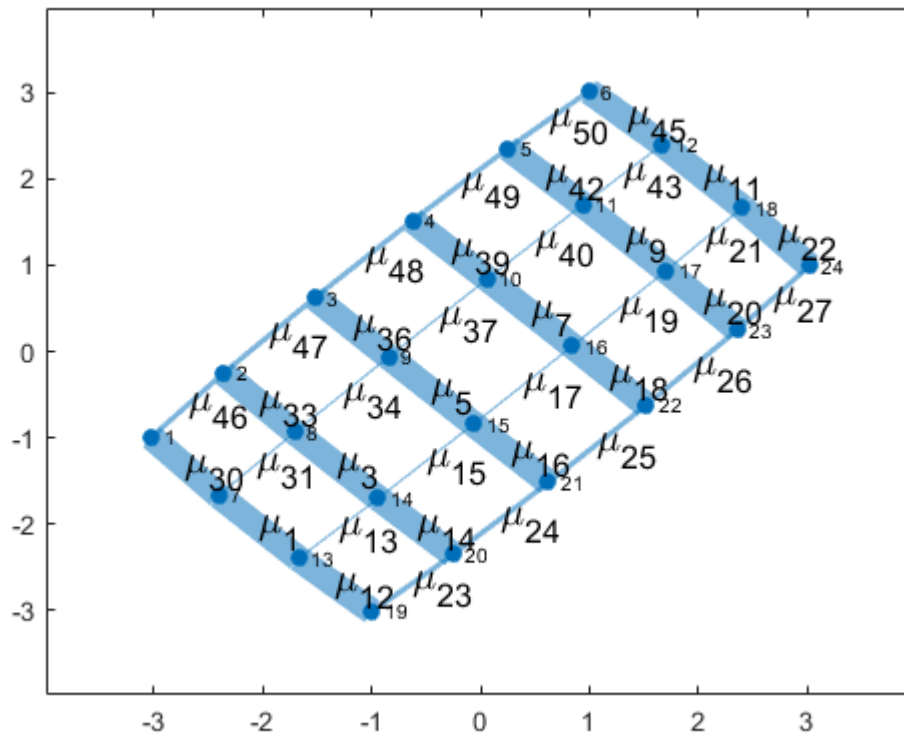
mu_18: 'mu_39'
mu_19: 'mu_40'
mu_20: 'mu_42'
mu_21: 'mu_43'
mu_22: 'mu_45'
mu_23: 'mu_46'
mu_24: 'mu_47'
mu_25: 'mu_48'
mu_26: 'mu_49'
mu_27: 'mu_50'

```

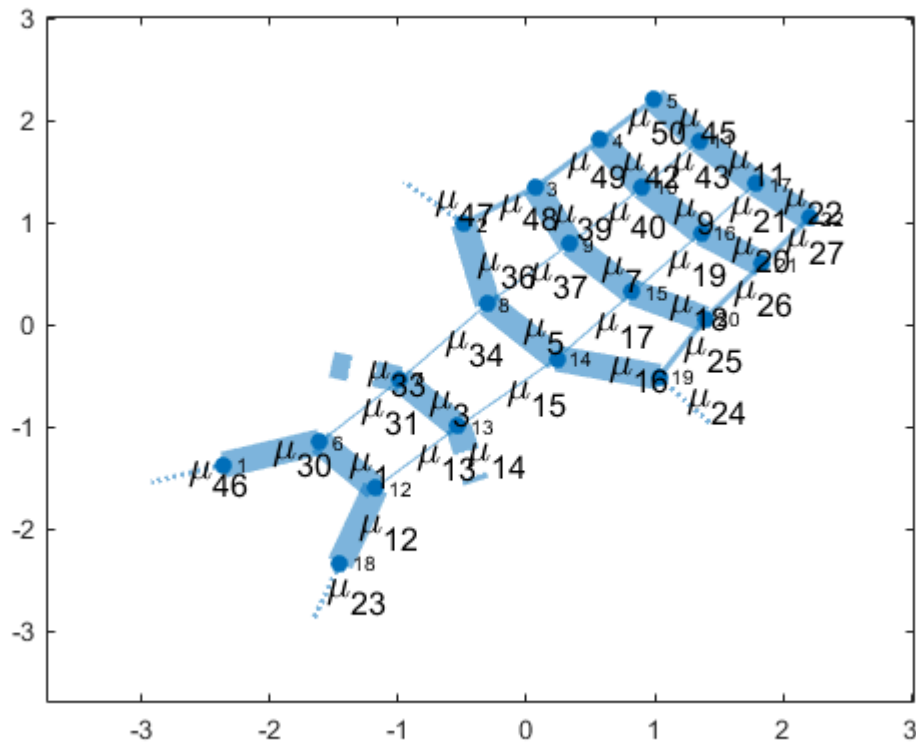
```
net_view(T,A,N) % no brackets necessary anymore
```



```
net_view(N_prime,A_prime,A,N) % no brackets necessary anymore
```



```
b = 2;
all_but_b = ~(1:numel(N)==b);
net_view(N_prime(all_but_b),A_prime,A,N(all_but_b)); % also no ghost nodes necessary and
```



## Running linsolve\_equalnet

The algorithm is not to be understood as a high performance tool. On the other hand, it is not terribly slow either:

```
data_volume(A)
```

```
ans = 18001
```

```
data_volume(N)
```

```
ans = 4681
```

```
activate_boxtimes_mem();
N = randomize_net(N);
tic
N = LINSOLVE_EQUALNET(A,N,T,5);
```

```
rel res: 2.57e-01
rel res: 1.92e-03
rel res: 3.97e-05
rel res: 1.65e-05
rel res: 1.63e-05
```

```
toc
```



Elapsed time is 1.655280 seconds.

```
net_dist({A,N},T)/net_norm(T)
```

```
ans = 1.6289e-05
```