Decentralized linear controller synthesis via LMI: decLMII Class

Contents

Definition

This class provide the user with a tool capable of synthetizing a decentralized linear controller that guarantees stability of the closed loop with the LTI discrete-time plant. Funcionalities are also given for achieve either robustness or stocastich convergence while guaranteeing also state and input constraints.

Class description

This work is based on the article, accepted for presentation at the 49th IEEE Control and Decision Coference 2010, "Synthesis of Networked Switching Linear Decentralized Controllers" by D. Barcelli, D. Bernardini and A. Bemporad.

The inputs for the class are

- the network state, i.e. the connection matrix of each node to each actuator;
- the plant A and B matrices;
- the Riccati equation weights;
- the state and input constraints;
- the initial state uncertainty polytope (set of vertices).
- the Markov chain (Needed for stochastic only)

All the methods provided build a SPD probelm which solution is composed by the feedbak gain K and the Riccati equation solution P. The LMI problem is dependent on the type of stability requested. Available methods cover:

- Centralized: the network is assumed fully connected;

- Decentralized Ideal: all links are reliable and fualtless;
- Decentralized Robust: the resulting controller gain is guranteed to be stable for any possible configuration of the packetdrops, while fulfilling the constraints;
- Decentralized Stochastic: stability is guaranteed in the mean square sense, as well as the constraint fulfillment, however the conservatism is expected to be minor then the robust case.

`classdef decLMI`

Output (read-only) properties

`Mc` is the Markov chain structure, which differs from the one requested by the construtor as it includes:

- T : transition matrix;
- E : emission matrix.

The other properties are the output of the SDP solving process, and are:

- P : Riccati equation solution;
- K : matrix gain (i.e. u=K*x);
- g : γ in $P = \gamma Q^{-1}$ that is the minimized parameter in the LMI optimization; where each of those is a structure with fileds:

- ci : centralized ideal;
- di : decentralized ideal
- dl : decentralized lossy, i.e. robust;
- ds : decentralized sotchastic. that are computed by corresponding methds.

```
properties (SetAccess=private)
    Mc
    P
    K
    g
end

% private properties
properties (Access=private)
    Net
    A
    B
    Qx
    Qu
    X0
```

```
        xmax
        umax
        n
        m
        nLi
        Nconf
        conf
        tLambda_i
        tLambda
        li
        L
        ell
    end

    methods
```

Class constructor

obj=decLMI(Net,A,B,Qx,Qu,X0,xmax,umax,Mc)

`Net` : matrix of row size equal to number of actuators and columns size equal to number of states, moreover the element (i,j) can be:

- 1 : if the state j is connected to actuator i;
- 0 : if previous condition doesn't hold;
- -1: if is a lossy link;

`A` : LTI state matrix of the plant;

`B` : LTI input matrix of the plant;

`Qx` : state weight matrix of the Riccati equation;

`Qu` : input weight matrix of the Riccati equation;

`X0` : set of vertices of the polytope that define the uncertainty of the initial state condition;

`xmax` : state infinity norm constraint;

`umax` : input infinity norm constraint;

`Mc` : two-staes Markov chain that model the probability of losing a packet. Must be a structor with fields:

- d : array, where d(i) is the probability of losing a packet being in the i-th state of the Markov chain;

- q : array, where q(i) is the probability of of remaining in the i-th state of the Markov chain.

```
function obj=decLMI(varargin)
    error(nargchk(8,9,nargin));

    Net=varargin{1};
    if 1%checkNet(Net)
        obj.Net=Net;
    else
        error('First argument must be a valit network topology');
    end
    clear Net;

    A=varargin{2};
    B=varargin{3};
    if (size(A,1)==size(A,2))&&(size(A,1)==size(B,1))
        obj.A=A;
        obj.B=B;
        [nx nu]=size(B);
    else
        error(['Second and Third arguments must be valid A and B '...
            'state space matrix, respectively']);
    end
    clear A B;

    Qx=varargin{4};
    if (size(Qx,1)==size(Qx,2)) & (size(Qx,1)==nx)
        obj.Qx=Qx;
    else
        error(['Fourth argument must be a square matrix of the '...
            'same size of A']);
    end
    clear Qx;

    Qu=varargin{5};
    if (size(Qu,1)==size(Qu,2)) & (size(Qu)==nu)
        obj.Qu=Qu;
    else
        error(['Fifth argument must be a square matrix of the '...
            'same size of B']);
    end
    clear Qu;

    X0=varargin{6};
    if (size(X0,1)==nx)
```

```
    obj.X0=X0;
else
    error(['Sixth argument must be an array of vetices of the '...
        'polytope containing all initial conditions, hence '...
        'each vertex must have same size as A']);
end
clear X0;

xmax=varargin{7};
if isnumeric(xmax)
    obj.xmax=xmax;
else
    error(['Seventh argument is the norm of the states, hence '...
        'must be a number']);
end
clear xmax;

umax=varargin{8};
if isnumeric(umax)
    obj.umax=umax;
else
    error(['Eigth argument is the norm of the inputs, hence '...
        'must be a number']);
end
clear umax;

if (nargin==9)%check is in order for Mc
    obj.Mc=varargin{9};
end

% Compuation of all possible Network configutations for all
% possible packet-loss configutation is performed by means of
% the following recursive function
obj=obj.set_up_lambda();
set(0,'RecursionLimit',1e4);
tLambda_i = [];
for i = 1:obj.m
    count=1;
    tLambda_i = define_tLambda_i(obj.Net,i,obj.nLi,tLambda_i,obj.conf{i},count)
end
tLambda=[];
tLambda = find_tilde_lambda(obj.Net,obj.conf,tLambda_i,tLambda);
obj.tLambda=tLambda;
obj.tLambda_i=tLambda_i;

obj.li = []; % no. of -1 in Net(i,:)
```

```
    for i=1:obj.m
        obj.li(i) = sum(obj.Net(i,:)==-1);
    end
    obj.L = sum(obj.li); % total no. of -1 in Net
    obj.ell = 2^obj.L;

    if (nargin==9)
        Mc=varargin{9};
        Mc.T = [Mc.q(1)   1-Mc.q(1); % transition matrix
        1-Mc.q(2) Mc.q(2)];
        obj.Mc=Mc;

        % create s0(i), s1(i)
        s0 = zeros(obj.ell,1);
        s1 = s0;
        for h=1:obj.ell
            for i=1:obj.m
                for j=1:obj.n
                    if obj.Net(i,j)==-1
                        s1(h) = s1(h) + obj.tLambda{h}(i,j);
                        s0(h) = s0(h) + (1-obj.tLambda{h}(i,j));
                    end
                end
            end
        end

        obj.Mc.E = zeros(2,obj.ell); % emission matrix
        for i=1:2
            for j=1:obj.ell
                obj.Mc.E(i,j)= obj.Mc.d(i)^(s0(j)) * (1-obj.Mc.d(i))^(s1(j));
            end
        end
    end

end
```

Centralized LMI computation

Computes the solution of the SDP problem assuming the network to be fully connected and each link completely reliable. The goal is to give an upper bound to the performance that can be achieved via the proposed method.

```
function obj = solve_centralized_lmi(obj)

    A=obj.A;
```

```
B=obj.B;
Qx=obj.Qx;
Qu=obj.Qu;
X0=obj.X0;
xmax=obj.xmax;
umax=obj.umax;

% Formulate and solve an SDP for a centralized problem with no packet loss

fprintf('\nSolving centralized SDP problem...\n');

n = obj.n;
m = obj.m;

Q = sdpvar(n,n,'symmetric');
Y = sdpvar(m,n);
g = sdpvar(1,1);

F = set(Q>0); % positive definiteness of Q

F = F + set([Q          (A*Q+B*Y)'      (Qx^.5*Q)'      (Qu^.5*Y)'; ...
             A*Q+B*Y    Q               zeros(n)        zeros(n,m); ...
             Qx^.5*Q    zeros(n)        g*eye(n)        zeros(n,m); ...
             Qu^.5*Y    zeros(m,n)      zeros(m,n)      g*eye(m)] >= 0); % stabi

F = F + set([umax^2*eye(m)      Y;
             Y'                 Q]>=0); % input constraints

F = F + set([Q          (A*Q+B*Y)';
             A*Q+B*Y    xmax^2*eye(n)]>=0); % state constraints

for i=1:size(X0,2)
    F = F + set([1          X0(:,i)'; ...
                 X0(:,i)    Q]>= 0); % ellipsoid definition for every vertex of
end

sdpopt = sdpsettings('solver','sedumi');
status_sol_cntr = solvesdp(F,g,sdpopt);

if status_sol_cntr.problem~=0 && status_sol_cntr.problem~=4
    fprintf('\n\nCentralized problem infeasible\n\n')
    status_sol_cntr.problem
    return
end

fprintf('\nCentralized SDP problem solved!\n');
```

```
    invQ = inv(double(Q));
    g = double(g);
    P = g*invQ;
    K = double(Y)*invQ;

    OLeig = eig(A);
    CLeig = eig(A+B*K);

    figure
    r = rectangle('Position',[-1 -1 2 2],'Curvature',[1 1],'EdgeColor',[.5 .5 .5]);
    hold on
    plot(real(OLeig),imag(OLeig),'*b')
    plot(real(CLeig),imag(CLeig),'or')
    axis equal
    xlim = get(gca,'XLim');
    ylim = get(gca,'Ylim');
    plot([floor(xlim(1)) ceil(xlim(2))],[0 0],'Color',[.5 .5 .5]);
    plot([0 0],[-1.55 1.55],'Color',[.5 .5 .5]);
    title 'Centralized feedback: Eigenvalues'
    box

    obj.P.ci=P;
    obj.K.ci=K;
    obj.g.ci=g;
end
```

Decentralized Ideal computation

Computes the solution considering only present links, but assuming that all of them are reliable. Basically does not account for dops.

```
function obj = solve_dec_ideal_lmi(obj)

    A=obj.A;
    B=obj.B;
    Qx=obj.Qx;
    Qu=obj.Qu;
    X0=obj.X0;
    xmax=obj.xmax;
    umax=obj.umax;
    Net=obj.Net;

    % Formulate and solve an SDP for a decentralization with no packet loss
```

```
fprintf('\nSolving decentralized SDP problem...\n');

n = size(Net,2);
m = size(Net,1);


% Definition of zero components in K
% K(i,j) = 0 if state j is not available to compute input i
K0 = Net;


% Definition of the structure of Y
Y0 = K0; % strucure of Y


% Definition of the variable Y
Y = sdpvar(m,n);
for i=1:m
    for j=1:n
        if Y0(i,j)==0
            Y(i,j) = 0;
        end
    end
end


% Definition of the structure of Q
Q0 = ones(n);
for i=1:m
    for j=1:n
        if K0(i,j)==0
            for h=1:n
                if K0(i,h)==1
                    Q0(h,j) = 0;
                    Q0(j,h) = 0;
                end
            end
        end
    end
end

% Definition of the variable Q
Q = sdpvar(n,n,'symmetric');
for i=1:n
    for j=1:n
        if Q0(i,j)==0
```

```
            Q(i,j) = 0;
        end
    end
end


g = sdpvar(1,1);

F = set(Q>0); % positive definiteness of Q

F = F + set([Q          (A*Q+B*Y)'      (Qx^.5*Q)'      (Qu^.5*Y)'; ...
    A*Q+B*Y    Q               zeros(n)        zeros(n,m); ...
    Qx^.5*Q    zeros(n)        g*eye(n)        zeros(n,m); ...
    Qu^.5*Y    zeros(m,n)      zeros(m,n)      g*eye(m)] >= 0); % stability cons

F = F + set([umax^2*eye(m)      Y;
    Y'                 Q]>=0); % input constraints

F = F + set([Q          (A*Q+B*Y)';
    A*Q+B*Y    xmax^2*eye(n)]>=0); % state constraints

for i=1:size(X0,2)
    F = F + set([1          X0(:,i)'; ...
        X0(:,i)     Q]>= 0); % ellipsoid definition for every vertex of X0
end

sdpopt = sdpsettings('solver','sedumi');
status_sol_dec = solvesdp(F,g,sdpopt);

if status_sol_dec.problem~=0 && status_sol_dec.problem~=4
    fprintf('\n\nDecentralized problem infeasible\n\n')
    return
end

fprintf('\nDecentralized SDP problem solved!\n');

invQ = inv(double(Q));
g = double(g);
P = g*invQ;
K = double(Y)*invQ;

OLeig = eig(A);
CLeig = eig(A+B*K);

figure
r = rectangle('Position',[-1 -1 2 2],'Curvature',[1 1],'EdgeColor',[.5 .5 .5]);
```

```
    hold on
    plot(real(OLeig),imag(OLeig),'*b')
    plot(real(CLeig),imag(CLeig),'or')
    axis equal
    xlim = get(gca,'XLim');
    ylim = get(gca,'Ylim');
    plot([floor(xlim(1)) ceil(xlim(2))],[0 0],'Color',[.5 .5 .5]);
    plot([0 0],[-1.55 1.55],'Color',[.5 .5 .5]);
    title 'Decentralized ideal feedback: Eigenvalues'
    box

    obj.P.di=P;
    obj.K.di=K;
    obj.g.di=g;

end
```

Decentralized Robust computation

Creates and solve the SDP problem guaranteeing stabilty and constraint sotisfaction for any possible occurrence of the losses. The most conservative solution.

```
function obj = solve_dec_lossy_lmi(obj)

    A=obj.A;
    B=obj.B;
    Qx=obj.Qx;
    Qu=obj.Qu;
    X0=obj.X0;
    xmax=obj.xmax;
    umax=obj.umax;
    Net=obj.Net;

    % Formulate and solve an SDP for a decentralization with packet loss

    fprintf('\nSolving dec. lossy SDP problem...\n');

    n = size(Net,2);
    m = size(Net,1);


    % In case there is no wireless links in a row, a fixed structure have to be
    % applied: K0fix. It should not be necessary to allocate all K0fix, but in
    % this way it's easier to manage indices
```

```
K0fix = sdpvar(m,n);

nLi = obj.nLi;
for i=1:m
    Nconf(i) = 2^(length(nLi{i}));
    if Nconf(i)>1
        conf{i} = dec2bin(Nconf(i)-1);
    else
        conf{i} = '-'; % there is no -1 in this line

        % set cooherently Kofix's line
        for j = 1:n
            if Net(i,j)==0
                K0fix(i,j) = 0;
            end
        end
    end

end


% define a memory structure that contains all possible configurations for
% each line of Net and its implications in K0{i,j}, where i means the i-th
% line of K0 and j means one of the 2^nLi{i} possible configurations

K0i = [];
set(0,'RecursionLimit',1e4);
for i = 1:m
    count=1;
    K0i = define_K0_i(Net,i,nLi,K0i,conf{i},count) ;
end
Y = [];

Y = def_lmis_K0i(K0i,nLi,K0fix,Y,conf,Net);

Q = sdpvar(n,n,'symmetric');
for w=1:length(Y)
    for i=1:m
        for j=1:n
            if double(Y{w}(i,j))==0
                for h=1:n
                    if isnan(double(Y{w}(i,h)))
                        Q(h,j)=0;
                        Q(j,h)=0;
                    end
                end
```

```
            end
        end
    end
end

g = sdpvar(1,1);

NN = length(Y);

F = set(Q > 0); % positive definiteness of Q

for j=1:size(X0,2)
    F = F + set([1          X0(:,j)'; ...
        X0(:,j)     Q       ]>= 0); % ellipsoid definition for every vertex of X
end

for i=1:NN

    F = F + set([Q          (A*Q+B*Y{i})' (Qx^.5*Q)'   (Qu^.5*Y{i})'; ...
        A*Q+B*Y{i} Q             zeros(n)     zeros(n,m); ...
        Qx^.5*Q    zeros(n)      g*eye(n)     zeros(n,m); ...
        Qu^.5*Y{i} zeros(m,n)    zeros(m,n)   g*eye(m)] >= 0); % stability const

    F = F + set([umax^2*eye(m)     Y{i};
        Y{i}'              Q  ]>=0); % input constraints

    F = F + set([Q              (A*Q+B*Y{i})';
        A*Q+B*Y{i}    xmax^2*eye(n)]>=0); % state constraints
end


sdpopt = sdpsettings('solver','sedumi');
sol = solvesdp(F,g,sdpopt);

if sol.problem~=0 && sol.problem~=4
    fprintf('\n\nDecentralized lossy problem infeasible\n\n')
    sol.problem
    return
end

g = double(g);
invQ = inv(double(Q));
P = g*invQ;
for i=1:NN
    K{i} = double(Y{i})*invQ;
end
```

```
    for i=1:NN
        disp(['Closed loop max eig: ' num2str(max(abs(eig(A+B*K{i}))))]);
    end

    OLeig = eig(A);
    CLeig=[];
    for i=1:NN
        CLeig = [CLeig;eig(A+B*K{i})];
    end

    figure
    r = rectangle('Position',[-1 -1 2 2],'Curvature',[1 1],'EdgeColor',[.5 .5 .5]);
    hold on
    plot(real(OLeig),imag(OLeig),'*b')
    plot(real(CLeig),imag(CLeig),'or')
    axis equal
    xlim = get(gca,'XLim');
    ylim = get(gca,'Ylim');
    plot([floor(xlim(1)) ceil(xlim(2))],[0 0],'Color',[.5 .5 .5]);
    plot([0 0],[-1.55 1.55],'Color',[.5 .5 .5]);
    title 'Decentralized lossy feedback: Eigenvalues'
    box

    obj.P.dl=P;
    obj.K.dl=K;
    obj.g.dl=g;
end
```

Decentralized Stochastic compuation

Solve the SDP problem exploiting the available knowledge of the loss probability. A two-states Markoc chain is used to describe the lossy behavior (which a commonly used choice). Stability and constraint satisfaction are guaranteed in the mean-square sense, meaning that in avarage hold but not point-wise.

```
function obj = solve_dec_stoch_lmi(obj)%v4
    if ~isempty(obj.Mc)

        Net=obj.Net;
        A=obj.A;
        B=obj.B;
        Qx=obj.Qx;
        Qu=obj.Qu;
```

```
X0=obj.X0;
xmax=obj.xmax;
umax=obj.umax;
Mc=obj.Mc;

% Formulate and solve an SDP for a decentralization with packet loss and
% convergence in mean-square

fprintf('\nSolving decentralized stoch. SDP problem...\n');

n = size(Net,2);
m = size(Net,1);

% In case there is no wireless links in a row, a fixed structure have to be
% applied: K0fix. It should not be necessary to allocate all K0fix, but in
% this way it's easier to manage indices
K0fix1 = sdpvar(m,n);
K0fix2 = sdpvar(m,n);


nLi = find_all_lossy(Net);
for i=1:m
    Nconf(i) = 2^(length(nLi{i}));
    if Nconf(i)>1
        conf{i} = dec2bin(Nconf(i)-1);
    else
        conf{i} = '-'; % there is no -1 in this line

        % set cooherently Kofix's line
        for j = 1:n
            if Net(i,j)==0
                K0fix1(i,j) = 0;
                K0fix2(i,j) = 0;
            end
        end
    end

end

% define a memory structure that contains all possible configurations for
% each line of Net and its implications in K0{i,j}, where i means the i-th
% line of K0 and j means one of the 2^nLi{i} possible configurations

K0i = [];
set(0,'RecursionLimit',1e4);
for i = 1:m
```

```
    count=1;
    K0i = define_K0_i(Net,i,nLi,K0i,conf{i},count);
end

Y = [];
Y1= [];
Y2= [];
Q=[];
Y1 = def_lmis_K0i(K0i,nLi,K0fix1,Y1,conf,Net);
for ww=1:2
    Q{ww} = sdpvar(n,n,'symmetric');
    for w=1:length(Y1)
        for i=1:m
            for j=1:n
                if double(Y1{w}(i,j))==0
                    for h=1:n
                        if isnan(double(Y1{w}(i,h)))
                            Q{ww}(h,j)=0;
                            Q{ww}(j,h)=0;
                        end
                    end
                end
            end
        end
    end
end

Y2 = def_lmis_K0i(K0i,nLi,K0fix2,Y2,conf,Net);

Y = {Y1' Y2'};

l = length(Y1);

% total numer of lossy links
L=log2(l);

conf=[];
for i=1:L
    conf=[conf '1'];
end

d=Mc.d;

e=Mc.E;

re=e;
```

```
for i=1:2
    for j=1:l
        re(i,j)=sqrt(e(i,j));
    end
end


g = sdpvar(1,1);

rhox = sdpvar(1,1);
rhou = sdpvar(1,1);

F = set(Q{1} > 0); % positive definiteness of Q
F = set(Q{2} > 0); % positive definiteness of Q

F = F + set(g > 0);
F = F + set(rhox > 0);
F = F + set(rhou > 0);

Qrho=1e3*[1 1];


for jj=1:size(X0,2)
    F = F + set([1          X0(:,jj)'; ...
        X0(:,jj)    Q{1}     ]>=0); % ellipsoid definition for every vertex

    F = F + set([1          X0(:,jj)'; ...
        X0(:,jj)    Q{2}     ]>=0); % ellipsoid definition for every vertex
end

for j=1:2

    % define the constraint as [Qj [CC' CC1'];[CC;CC1] DD]
    CC=[];
    for i=1:l
        CC=[CC ; sqrt(Mc.T(j,1))*re(j,i)*(A*Q{j}+B*Y{j}{i})];
    end
    for i=1:l
        CC=[CC ; sqrt(Mc.T(j,2))*re(j,i)*(A*Q{j}+B*Y{j}{i})];
    end
    CC=[CC ; Qx^.5*Q{j}];
    for i=1:l
        CC=[CC ; re(j,i)*Qu^.5*Y{j}{i}];
    end

    str=[];
```

```
    for i=1:l
        str=[str 'Q{1},'];
    end
    for i=1:l
        str=[str 'Q{2},'];
    end
    str=[str 'g*eye(n),'];
    for i=1:l
        str=[str 'g*eye(m),'];
    end
    str(end)=')';
    eval(['DD=blkdiag(' str ';']);

    F = F + set([Q{j} CC';CC DD]>=0);


end


% implement hard constraints (?)
F = F + set(rhox==0);
F = F + set(rhou==0);

sdpopt = sdpsettings('solver','sedumi');
%status_sol_dec = solvesdp(F,1e-3*g+Qrho*[rhox;rhou]+trace(Q{2})+trace(Q{1})
status_sol_dec = solvesdp(F,g+Qrho*[rhox;rhou],sdpopt);

if status_sol_dec.problem~=0 && status_sol_dec.problem~=4
    fprintf('\n\nDecentralized stoch. problem infeasible\n\n')
    return
end

fprintf('\nDecentralized stoch. SDP problem solved!\n');

g = double(g);
for j=1:2
    invQ = inv(double(Q{j}));
    P{j} = g*invQ;
    for i=1:l
        K{i,j} = double(Y{j}{i})*invQ;
    end
end


OLeig = eig(A);
CLeig=[];
```

```
            for j=1:2
                for i=1:l
                    CLeig = [CLeig;eig(A+B*K{i,j})];
                end
            end

            figure
            r = rectangle('Position',[-1 -1 2 2],'Curvature',[1 1],'EdgeColor',[.5 .5 .5
            hold on
            plot(real(OLeig),imag(OLeig),'*b')
            plot(real(CLeig),imag(CLeig),'or')
            axis equal
            xlim = get(gca,'XLim');
            ylim = get(gca,'Ylim');
            plot([floor(xlim(1)) ceil(xlim(2))],[0 0],'Color',[.5 .5 .5]);
            plot([0 0],[-1.55 1.55],'Color',[.5 .5 .5]);
            title 'Decentralized stochastic feedback: Eigenvalues'
            box

            obj.P.ds=P;
            obj.K.ds=K;
            obj.g.ds=g;
        else
            disp('Markov chain is needed. Computation is not performed.')
        end
    end

end

% private methods
methods (Access=private)

    function obj=set_up_lambda(obj)
        obj.n = size(obj.Net,2);
        obj.m = size(obj.Net,1);
        obj = obj.find_all_lossy();
        for i=1:obj.m
            obj.Nconf(i) = 2^(length(obj.nLi{i}));
            if obj.Nconf(i)>1
                obj.conf{i} = dec2bin(obj.Nconf(i)-1);
            else
                obj.conf{i} = '-'; % there is no -1 in this line
            end
        end
    end
```

```
        function obj = find_all_lossy(obj)
            obj.nLi=[];
            for i=1:size(obj.Net,1)
                obj.nLi{i} = find(obj.Net(i,:)==-1);
            end
        end

    end

end
```