Energy-Aware MPC: EAMPC class definition

This class provides an implementation of an explicit MPC controller, where communications between controller and sensor nodes are subject to an energy-aware policy intended to lower the number of transmissions and, utlimately, save sensor nodes battery.

by D. Bernardini, 2010.

Contents

Class description

This class is based on the papers:

- D. Bernardini and A. Bemporad, "Energy-aware robust model predictive control based on wireless sensor feedback," in Proc. 47th IEEE Conf. on Decision and Control, Cancun, Mexico, 2008, pp. 3342–3347.

- D. Bernardini and A. Bemporad, "Energy-aware robust model predictive control with feedback from multiple noisy wireless sensors," 10th European Control Conference, Budapest, Hungary, 2009, pp. 4308–4313.

The input arguments to create an EAMPC object are:

- the A and B matrices of the controlled plant model, which is a discrete-time LTI system.

- the WSN configuration, namely the number of sensor nodes to use, the thresholds for the transmission policy, and the length of the state predictions buffer.

- the desired constraints on inputs and outputs.

- the weight matrices and other parameters which define the MPC objective function.

The overall control system is assumed to have the following configuration. A number of wireless sensor nodes collect noisy state measurements for disturbance rejection, compute an estimated state value, and transmi this estimation to the controller according to a transmission policy. This policy is based on a threshold logic. Namely, at every time step the estimated state value `Ymean` is transmitted to the controller if and only if there exists `i` such that

$|$`Ymean(i) - Xhat(i)`$| \geq$ `th(i)`

where `th` is a vector of user-defined thresholds, and `Xhat` is the predicted states buffer, which is precomputed by the controller and transmitted beforehand to the sensor nodes. Updated state predictions are computed and transmitted to the sensors every time new measurements are received by the controller. It is assumed that every sensor node collects a (noisy) measurement of the complete state vector, and that the energy cost of a (short-range) transmission between sensors is negligible with respect to a (long-range) transmission between sensors and controller.

`classdef eampc`

Public properties

`Net` : specifies the network configuration. It is a structure with fields

- `nodes` : number of wireless sensor nodes used.
- `th` : threshold vector for measurement transmission policy. **Note:** if `th` is a vector with all zero entries, then a standard MPC control law is implemented, where at every time step measurements are sent to controller regardless to the threshold-based transmission policy.
- `Ne` : estimation horizon for state predictions computation.

```
properties
    Net
end
```

Output (read-only) properties

`nu` : number of inputs of the controlled system.

`ny` : number of outputs of the controlled system.

`Plant` : controlled plant, modeled as a discrete-time LTI system. It is a structure with fields

- `A` : state matrix (`ny` x `ny`).
- `B` : input matrix (`ny` x `nu`).

`Net` : network configuration. It is a structure with fields

- `nodes` : number of wireless sensor nodes used.
- `th` : threshold vector for measurement transmission policy. **Note:** if `th` is a vector with all zero entries, then a standard MPC control law is implemented, where at every time step measurements are sent to controller regardless to the threshold-based transmission policy.
- `Ne` : estimation horizon for state predictions computation.

`Limits` : element-wise constraints on outputs and inputs. It is a structure with fields

- `ymin` : lower bounds on output (`ny` x 1).
- `ymax` : upper bounds on output (`ny` x 1).
- `umin` : lower bounds on input (`nu` x 1).
- `umax` : upper bounds on input (`nu` x 1).
- `dumin` : lower bounds on input rate (`nu` x 1). Optional.
- `dumax` : upper bounds on input rate (`nu` x 1). Optional.

`Weights` : weight matrices to be used in the MPC objective function. It is a structure with fields

- `Qu` : input weight (`nu` x `nu`).
- `Qy` : output weight (`ny` x `ny`).
- `Qn` : terminal state weight (`ny` x `ny`). Optional.
- `rho` : positive weight for soft output constraints (1 x 1). Optional.

`Params` : other parameters for the MPC problem design. It is a structure with fields

- `pnorm` : norm used in the MPC objective function.
- `N` : prediction horizon.
- `Nc` : control horizon. Optional.

`Ctrl` : explicit MPC controller obtained with MPT Toolbox. See mpt_control for more details.

`Sim` : Data used for simulations. It is a structure with fields

- `Y` : set of current measurements.
- `Ymean` : estimated output value, taken as the mean value of Y.
- `Xhat` : buffer of predicted state values.
- `tx` : records transmissions from sensor nodes to controller.
- `rx` : records transmissions from controller to sensor nodes.

```
properties (SetAccess=private)
    nu
    ny
    Plant
    Limits
    Weights
    Params
    Ctrl
    Sim
end


methods
```

Class constructor

`obj = eampc(Plant,Net,Limits,Weights,Params)`

`Plant` : controlled plant, modeled as a discrete-time LTI system. It is a structure with fields

- `A` : state matrix (`ny` x `ny`). Mandatory.
- `B` : input matrix (`ny` x `nu`). Mandatory.

`Limits` : element-wise constraints on outputs and inputs. It is a structure with fields

- `ymin` : lower bounds on output (`ny` x 1). Mandatory.
- `ymax` : upper bounds on output (`ny` x 1). Mandatory.
- `umin` : lower bounds on input (`nu` x 1). Mandatory.
- `umax` : upper bounds on input (`nu` x 1). Mandatory.
- `dumin` : lower bounds on input rate (`nu` x 1). Optional (default: `-Inf`).
- `dumax` : upper bounds on input rate (`nu` x 1). Optional (default: `Inf`).

`Weights` : weight matrices to be used in the MPC objective function. It is a structure with fields

- `Qu` : input weight (`nu` x `nu`). Mandatory.
- `Qy` : output weight (`ny` x `ny`). Mandatory.
- `Qn` : terminal state weight (`ny` x `ny`). Optional (default: `Qy`).
- `rho` : positive weight for soft output constraints (1 x 1). Optional (default: `Inf`).

`Params` : other parameters for the MPC problem design. It is a structure with fields

- `pnorm` : norm used in the MPC objective function. It can be 1, 2, `Inf`. Mandatory.
- `N` : prediction horizon. Mandatory.
- `Nc` : control horizon. Optional (default: `N`).

```
function obj = eampc(varargin)

    % Check input arguments
    error(nargchk(5,5,nargin)); % min nargin, max nargin
    iarg = 0;

    % check PLANT
    iarg = iarg + 1;
    fields = {'A','B'};
    for i=1:length(fields)
        if ~isfield(varargin{iarg},fields{i})
            error(['Missing field ' fields{i} ...
                ' from input argument Plant']);
        end
    end
    obj.Plant = varargin{iarg};

    % check NET
    iarg = iarg + 1;
    fields = {'nodes','th','Ne'};
    for i=1:length(fields)
        if ~isfield(varargin{iarg},fields{i})
            error(['Missing field ' fields{i} ...
                ' from input argument Net']);
        end
    end
    obj.Net = varargin{iarg};

    % check LIMITS
    iarg = iarg + 1;
    fields = {'ymin','ymax','umin','umax'};
    for i=1:length(fields)
        if ~isfield(varargin{iarg},fields{i})
            error(['Missing field ' fields{i} ...
                ' from input argument Limits']);
        end
    end
    obj.Limits = varargin{iarg};
    % optional fields
    fields = {'dumin','dumax'};
    for i=1:length(fields)
```

```
    if ~isfield(varargin{iarg},fields{i})
        obj.Limits.(fields{i}) = [];
    end
end

% check WEIGHTS
iarg = iarg + 1;
fields = {'Qu','Qy'};
for i=1:length(fields)
    if ~isfield(varargin{iarg},fields{i})
        error(['Missing field ' fields{i} ...
            ' from input argument Weights']);
    end
end
obj.Weights = varargin{iarg};
% optional fields
fields = {'Qn','rho'};
for i=1:length(fields)
    if ~isfield(varargin{iarg},fields{i})
        obj.Weights.(fields{i}) = [];
    end
end
% defaults for optional fields
if isempty(obj.Weights.Qn)
    obj.Weights.Qn = obj.Weights.Qy;
end

% check PARAMS
iarg = iarg + 1;
fields = {'pnorm','N'};
for i=1:length(fields)
    if ~isfield(varargin{iarg},fields{i})
        error(['Missing field ' fields{i} ...
            ' from input argument Params']);
    end
end
obj.Params = varargin{iarg};
% optional fields
fields = {'Nc'};
for i=1:length(fields)
    if ~isfield(varargin{iarg},fields{i})
        obj.Params.(fields{i}) = [];
    end
end
```

```
        % Set aux vars and compute explicit MPC

        % plant dimension
        obj.nu = size(obj.Plant.B,2);
        obj.ny = size(obj.Plant.A,1);

        % compute MPC controller
        obj = build_MPC(obj);

    end
```

init_sim

Initializes EAMPC object for simulations. This method must be called before running a new simulation.

Usage:

```
obj = init_sim(obj)
```

where

`obj` : EAMPC object. Mandatory.

```
    function obj = init_sim(obj)

        % Check input arguments
         error(nargchk(1,1,nargin)); % min nargin, max nargin

        % Init variables used in simulation
        obj.Sim = struct();
        obj.Sim.Y = []; % measured outputs
        obj.Sim.Ymean = []; % estimated output value
        obj.Sim.Xhat = []; % predicted state buffer
        obj.Sim.tx = []; % records outgoing transmissions
        obj.Sim.rx = []; % records incoming transmissions

    end
```

send_predictions

Checks if an updated prediction buffer needs to be provided to the sensor nodes. In this case, a sequence of `obj.Net.Ne` future state predictions are computed and transmitted to the sensor nodes. **Note:** if all the components of `obj.Net.th` are zeros, no transmission takes place.

Usage:

`obj = send_predictions(obj,Xk,Uprev)`

where

`obj` : EAMPC object. Mandatory.

`Xk` : current estimated state value. Mandatory.

`Uprev` : previous input value. Mandatory if input rate constraints are imposed, otherwise ignored.

```
function obj = send_predictions(obj,Xk,Uprev)

    % Check input arguments
    if isfield(obj.Ctrl.sysStruct, 'dumode')
        % parameters are state and previous input
        error(nargchk(3,3,nargin)); % min nargin, max nargin
        np = obj.ny+obj.nu; % length of parameters vector
    else
        % parameters are only state
        error(nargchk(2,3,nargin)); % min nargin, max nargin
        if nargin<3
            Uprev = zeros(obj.nu,1); % dummy variable
        end
        np = obj.ny; % length of parameters vector
    end


    if any(obj.Net.th~=0)

        % Compute and transmit predictions
        if isempty(obj.Sim.Xhat) || obj.Sim.tx(end)==1

            % compute predictions up to Ne steps ahead,
            % using nominal plant model
            obj.Sim.Xhat = Xk; % predicted state values
            for i=1:obj.Net.Ne
                % get MPC move
                param = [obj.Sim.Xhat(:,i); Uprev];
                Uhat = mpt_getInput(obj.Ctrl,param(1:np));
                if isfield(obj.Ctrl.sysStruct, 'dumode')
                    Uhat = Uhat + Uprev;
                end
                obj.Sim.Xhat(:,i+1) = obj.Plant.A*obj.Sim.Xhat(:,i) ...
```

```
                    + obj.Plant.B*Uhat; % update system
                Uprev = Uhat; % update previous input move
            end
            if ~isempty(obj.Sim.Y)
                obj.Sim.Xhat = obj.Sim.Xhat(:,2:end);
            end

            % predictions are sent to sensors
            obj.Sim.rx(end+1) = 1; % record incoming transmissions

        else

            obj.Sim.rx(end+1) = 0; % predictions are not transmitted

        end

    else

        % standard MPC (no predictions are computed)
        obj.Sim.Xhat = zeros(obj.ny,1); % dummy values
        obj.Sim.rx(end+1) = 0; % predictions are not transmitted

    end

end
```

get_measurements

Given the actual state `X` and the output noise `V`, provides to the controller an estimation of the state vector. Measurements from all sensor nodes are gathered (assuming all the state components are measured by each node) and the average output `Ymean` is computed. Then, `Ymean` is transmitted to the controller if and only if there exists `i` such that

$|$`Ymean(i) - Xhat(i)`$| \geq$ `th(i)`

Usage:

```
[Xestim,obj] = get_measurements(obj,X,V)
```

where

`obj` : EAMPC object. Mandatory.

`X` : state value (`ny` x 1). Mandatory.

`V` : output noise matrix (`ny` x no. of nodes). Mandatory.

`Xestim` : estimated state value.

```
function [Xestim,obj] = get_measurements(obj,X,V)

    % get measured output from each sensor node
    if isempty(obj.Sim.Y)
        nc = 1;
    else
        nc = size(obj.Sim.Y,3) + 1;
    end
    obj.Sim.Y(:,:,nc) = zeros(obj.ny,obj.Net.nodes);
    for i=1:obj.Net.nodes
        obj.Sim.Y(:,i,end) = X + V(:,i); % measured output
    end

    % compute estimated output value
    obj.Sim.Ymean = mean(obj.Sim.Y(:,:,end),2);

    % check if data need to be transmitted
    delta = 0; % binary variable, 0 = do not transmit, 1 = transmit
    for i=1:obj.ny
        % evaluate threshold condition
        if abs(obj.Sim.Ymean(i)-obj.Sim.Xhat(:,1)) >= obj.Net.th(i)
            delta = 1;
            break
        end
    end

    % update estimated state value
    if delta==1
        Xestim = obj.Sim.Ymean; % data are transmitted
    else
        Xestim = obj.Sim.Xhat(:,1); % data are not transmitted
    end
    obj.Sim.tx(end+1) = delta; % record outgoing transmission
    obj.Sim.Xhat = obj.Sim.Xhat(:,2:end); % update Xhat buffer

end
```

get_input

Computes MPC control move given a controller and an initial state.

Usage:

```
Uopt = get_input(obj,X,Uprev)
```

where

`obj` : EAMPC object. Mandatory.

`X` : initial state. Mandatory.

`Uprev` : previous contrl move. Mandatory if input rate constraints are imposed, otherwise ignored.

```
        function Uopt = get_input(obj,X,Uprev)

            if isfield(obj.Ctrl.sysStruct, 'dumode')
                [Uopt,feasible,region] = mpt_getInput(obj.Ctrl,[X; Uprev]);
                Uopt = Uopt + Uprev;
            else
                [Uopt,feasible,region] = mpt_getInput(obj.Ctrl,X);
            end

        end

    end


    methods (Access=private)
```

build_mpc

Computes the explicit solution of the MPC problem:

$$J(x(0)) = \min_u \|Q_N x(N)\|_p + \sum_{j=0}^{N-1} \|Q_y y(j)\|_p + \|Q_u u(j)\|_p$$
$$\text{subject to } \; x(j) \in X, \; u(j) \in U$$

for every $x(0) \in X$. Assume full state-feedback.

Usage:

`obj = build_MPC(obj)`

where

`obj` : EAMPC object. Mandatory.

```
    function obj = build_MPC(obj)

        % set parameters
        sys = struct();
        sys.A = obj.Plant.A;
        sys.B = obj.Plant.B;
        sys.C = eye(obj.ny);
        sys.D = zeros(obj.ny,obj.nu);
        sys.umin = obj.Limits.umin;
        sys.umax = obj.Limits.umax;
        sys.xmin = obj.Limits.ymin;
        sys.xmax = obj.Limits.ymax;
        if ~isempty(obj.Limits.dumin)
            sys.dumin = obj.Limits.dumin;
        end
        if ~isempty(obj.Limits.dumax)
            sys.dumax = obj.Limits.dumax;
        end
        sys.Pbnd = polytope([eye(obj.ny);-eye(obj.ny)], ...
            [obj.Limits.ymax; -obj.Limits.ymin]);
        prob = struct();
        prob.N = obj.Params.N;
        prob.Q = obj.Weights.Qy;
        prob.R = obj.Weights.Qu;
        prob.P_N = obj.Weights.Qn;
        %prob.Qy = [];
        prob.norm = obj.Params.pnorm; % norm in the objective function
        prob.y0bounds = 0;
        prob.tracking = 0; % regulation problem
        prob.Tconstraint = 0; % no terminal constraint
        if ~isempty(obj.Params.Nc)
            prob.Nc = obj.Params.Nc; % control horizon
        else
            prob.Nc = obj.Params.N; % default value
        end
        prob.subopt_lev = 0;
        if ~isempty(obj.Weights.rho) && isfinite(obj.Weights.rho)
            % set penalty for soft constraints
            prob.Sx = obj.Weights.rho*eye(obj.ny);
        end
        % compute explicit MPC and try to join regions
        obj.Ctrl = mpt_simplify(mpt_control(sys,prob));

    end

end
```

```
end
```