PnPMPC Toolbox v. 1.0 - User manual[1]

Stefano Riverso, Alberto Battocchio, and Giancarlo Ferrari-Trecate

Dipartimento di Ingegneria Industriale e dell'Informazione
Università degli Studi di Pavia
via Ferrata, 1
27100 Pavia
Italy

April, 2014

[1]The research leading to these results has received funding from the European Union Seventh Framework Programme [FP7/2007-2013] under grant agreement n° 257462 HYCON2 Network of excellence.

Chapter 1

Introduction

The *PnPMPC toolbox* is a GNU-licensed MatLab toolbox for the modeling of constrained Large-Scale Systems (LSS) with linear time-invariant dynamics and for the implementation of the Plug-and-Play (PnP) Decentralized (De) and Distributed (Di) Model Predictive Control (MPC) schemes described in [1], [2], [3] and [4], and for the implementation of large-scale estimators and PnP state estimators [5], [6] and [4]. The PnPMPC toolbox offers also several functionalities for handling zonotopes set and for computing invariant sets.

The realization of the toolbox has received funding from the European Union Seventh Framework Programme [FP7/2007-2013] under grant agreement n° 257462 HYCON2 Network of excellence.

The last version of the PnPMPC toolbox can be downloaded at the webpage

`http://sisdin.unipv.it/pnpmpc/pnpmpc.php`

Please send bug reports, questions or comments to pnpmpc_toolbox@unipv.it

1.1 Notations

- $\mathbb{R}$ is the set of real number.
- $\mathbb{N}$ is the set of integers.
- $a : b$ is the set of integer $\{a,\ a+1, \ldots,\ b\}$
- $A \geq 0$ means that the matrix A is positive semi-definite. $A > 0$ means that matrix A is positive definite.
- $A = \text{diag}(A_{11}, \ldots, Ass)$ is the block diagonal matrix

$$\begin{bmatrix} A_{11} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & A_{ss} \end{bmatrix} \tag{1.1}$$

- $x_{[i]},\ u_{[i]},\ y_{[i]},\ d_{[i]}, u_{[cen_i]}$ are column vectors of suitable dimensions.

- $\mathbf{x} = (x_{[1]},\ x_{[2]}, \ldots,\ x_{[s]})$ means that $\mathbf{x}$ is composed by column vectors $x_{[i]}$, $i = 1 : s$ stacked in a single column, i.e.

$$\mathbf{x} = \begin{bmatrix} x_{[1]} \\ x_{[2]} \\ \vdots \\ x_{[s]} \end{bmatrix} \tag{1.2}$$

- The symbol $\oplus$ denotes the Minkowski sum, e.g. $A = B \oplus C$ if and only if $A = \{a : a = b + c, \forall b \in B, \forall c \in C\}$.
- $\bigoplus_{i=1}^{s} G_i = G_1 \oplus \ldots \oplus G_s$.
- $\times$ indicates the cartesian product and $\prod_{i=1}^{s} G_i = G_1 \times \ldots \times G_s$

Definition 1. *A polyhedron $\mathbb{X}$ is the intersection of a finite number of closed half spaces. Therefore we can represent a polyhedron either as*

$$\mathbb{X} = \{x \in \mathbb{R}^n :\ Hx \leq K\}$$

where $H \in \mathbb{R}^{v \times n}$ and $K \in \mathbb{R}^v$ (H-representation) or as a convex hull of vertices (V-representation).

Definition 2. *A zonotope is a centrally symmetric convex polytopes: given a vector $p \in \mathbb{R}^n$ and a matrix $\Xi \in \mathbb{R}^{n \times m}$, the zonotope $\mathbb{X} \subseteq \mathbb{R}^n$ is the set $\mathbb{X} = \{x \mid x = p + \Xi d,\ ||d||_\infty \leq 1\}$, with $d \in \mathbb{R}^m$. We will refer to this representation as G-representation.*

1.2 Required toolboxes

For a complete use, PnPMPC toolbox requires the following toolboxes to be installed.

- Control System Toolbox which implements the class `ss` (state-space). We also use the function `c2d` for time-discretization.
- Optimization Toolbox which implements the class `fmincon` used to solve nonlinear optimization problems.
- MPT3 [7] which implements the `Polyhedron` class.
- YALMIP [8] which include the optimization function `solvesdp` that is called for solving optimization problems.
- GraphViz4MatLab toolbox [9] that allows one to plot the graph of a large-scale system composed by interconnected systems.
- INTLAB [**?**] that allows one to manage interval sets.
- Symbolic Math Toolbox to manage symbolic variables.

Please note that GraphViz4MatLab get automatically installed when installing PnPMPC. One can check if all required toolboxes are correctly installed with the following instructions.

```
yalmiptest
mpt_init
```

Also note that part of the toolboxes are needed only for few functions, therefore you could not need them.

1.3 Installation of the PnPMPC toolbox

- **Step 1** Add the folder of the PnPMPC toolbox and its subfolders in the MatLab path.
- **Step 2** Run `pnpmpc_toolbox_init`

  Remark 1: we tested PnPMPC toolbox on MatLab R2009b and newer version for Windows, Linux and MacOS. For older versions of MatLab we cannot guarantee that everything works correctly.

1.4 Directories

The PnPMPC toolbox consists of the following directories

- **./pnpmpc_toolbox/@cenmpc** contains methods for designing MPC controllers
- **./pnpmpc_toolbox/@epsilon_mRPI** contains methods for computing outer-approximation of minimal robust positively invariant sets
- **./pnpmpc_toolbox/@localControlLyapunov** contains methods for computing control invariant sets
- **./pnpmpc_toolbox/@lse** contains methods for designing large-scale state estimators
- **./pnpmpc_toolbox/@lss** contains methods for modeling of LSS
- **./pnpmpc_toolbox/@parameterizedRCI** contains methods for computing robust control invariant sets
- **./pnpmpc_toolbox/@pnpctrl** contains methods for designing of unconstrained PnP controllers
- **./pnpmpc_toolbox/@pnpEstimator** contains methods for designing of PnP state estimators
- **./pnpmpc_toolbox/@pnpmpc** contains methods for designing of PnPMPC controllers
- **./pnpmpc_toolbox/@subss** contains methods for the modeling of a subsystem of an LSS
- **./pnpmpc_toolbox/@zonotope** contains methods for using zonotope sets
- **./pnpmpc_toolbox/@zonotopeRCI** contains methods for computing zonotopic robust control invariant sets
- **./pnpmpc_toolbox/examples** contains examples demonstrating the functionalities of PnPMPC toolbox

- **./pnpmpc_toolbox/extra** contains extra useful functions
- **./pnpmpc_toolbox/toolbox** contains additional toolboxes provided with PnPMPC toolbox

For familiarizing quickly with PnPMPC toolbox, examples are supplied in the directory **./pnpmpc_toolbox/examples**. These .m files can also be used as templates for your personal experiments.

Chapter 2

Modeling of interconnected systems

In this chapter we describe the class of systems considered in the PnPMPC toolbox. Features of subsystems and their interconnections will be discussed in details. The use of the PnPMPC toolbox for modeling purposes will be illustrate in Chapter 3.

2.1 Large-scale linear systems

In this toolbox, we consider Multi-Input Multi-Output (MIMO), Linear Time-Invariant (LTI) systems either in discrete or continuous time. An LSS is composed by s physically interconnected subsystems.
In the following, dependence of variables from time will be omitted, except where indicated.
The dynamics of subsystem $i \in \mathcal{M} = 1 : s$ is

$$x^+{}_{[i]} = A_{ii}x_{[i]} + B_{ii}u_{[i]} + \sum_{j \in \mathcal{N}_i} (A_{ij}x_{[j]} + B_{ij}u_{[j]}) \tag{2.1}$$

where $x_{[i]} \in \mathbb{R}^{n_i}$ is the state at time k, $x^+{}_{[i]}$ is the state at time $k+1$ (for continuous-time systems, $x^+_{[i]}$ stands for $\frac{\mathrm{d}x_{[i]}}{\mathrm{d}t}$) and $u_{[i]} \in \mathbb{R}^{m_i}$ is the local input at time k.
Moreover we have $A_{ii} \in \mathbb{R}^{n_i \times n_i}$, $A_{ij} \in \mathbb{R}^{n_i \times n_j}$, $B_{ii} \in \mathbb{R}^{n_i \times m_i}$ and $B_{ij} \in \mathbb{R}^{n_i \times m_j}$. Next, we define the index set $\mathcal{N}_i$ appearing in (2.1).

Definition 3. *Subsystem i is* dynamically coupled *to subsystem j if $A_{ij} \neq 0$.*

Definition 4. *Subsystem i is* input coupled *to subsystem j if $B_{ij} \neq 0$.*

It is possible to use graph theory to represent the coupling between the subsystems. The coupling graph of a system is directed graph where nodes are the subsystems, and the set of edges is given by $\xi = \{(i, j) : i \neq j, A_{ij} \neq 0 \text{ or } B_{ij} \neq 0\}$. An example is given in Figure 2.1.

Coupling allows us to define the predecessors of a subsystem.

Definition 5. *The set of parents to subsystem i is $\mathcal{N}_i = \{j : (i, j) \in \xi\}$.*

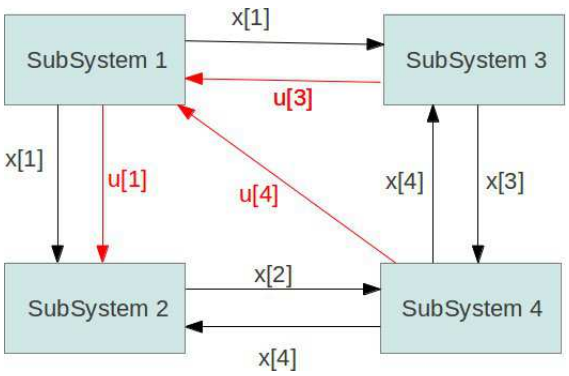

Figure 2.1: Example of a coupling graph of a system composed by four subsystems. A black arrow from system i to system j means that $A_{ij} \neq 0$; a red arrow means $B_{ij} \neq 0$.

For example, from Figure 2.1 one has $\mathcal{N}_2 = \{1, 4\}$ since A_{21}, B_{21} and A_{24} are not null matrices. We also define the set of children of subsystem i, i.e. the set of subsystems influenced by it.

Definition 6. *The set of children to subsystem i is $\mathcal{S}_i = \{j : (j, i) \in \xi\}$.*

For example, from Figure 2.1 one has $\mathcal{S}_2 = \{4\}$ since $A_{42} \neq 0$.
The output of system i is given by:

$$y_{[i]} = C_{ii}x_{[i]} + D_{ii}u_{[i]} + \sum_{j \in \mathcal{N}_i} (C_{ij}x_{[j]} + D_{ij}u_{[j]}) \tag{2.2}$$

where $y_{[i]} \in \mathbb{R}^{p_i}$, $C_{ii} \in \mathbb{R}^{p_i \times n_i}$, $D_{ii} \in \mathbb{R}^{p_i \times m_i}$, $C_{ij} \in \mathbb{R}^{p_i \times n_j}$ and $D_{ij} \in \mathbb{R}^{p_i \times m_j}$.

Definition 7. *Two subsystems i and j are* output coupled *if C_{ij} or D_{ij} are different from zero.*

It is also possible to include exogenous signals that act on system i, by replacing (2.1) and (2.2) with:

$$x^+{}_{[i]} = A_{ii}x_{[i]} + B_{ii}u_{[i]} + \sum_{j \in \mathcal{N}_i} (A_{ij}x_{[j]} + B_{ij}u_{[j]}) + \sum_{j \in \mathcal{N}_{d_i}} M_{ij}d_{[j]} \tag{2.3}$$

$$y_{[i]} = C_{ii}x_{[i]} + D_{ii}u_{[i]} + \sum_{j \in \mathcal{N}_i} (C_{ij}x_{[j]} + D_{ij}u_{[j]}) + \sum_{j \in \mathcal{N}_{d_i}} N_{ij}d_{[j]} \tag{2.4}$$

where $d_{[j]} \in \mathbb{R}$ is an exogenous input, $\mathcal{N}_{d,i} \subset \mathbb{N}$ is the set of exogenous inputs that act on subsystem i and $M_{ij} \in \mathbb{R}^{n_i}$, $N_{ij} \in \mathbb{R}^{p_i}$.
We further enhance our model by introducing centralized control inputs $u_{cen,[j]}$, so that the dynamics of subsystem i becomes:

$$x^+{}_{[i]} = A_{ii}x_{[i]} + B_{ii}u_{[i]} + \sum_{j \in \mathcal{N}_i} (A_{ij}x_{[j]} + B_{ij}u_{[j]}) + \sum_{j \in \mathcal{N}_{d,i}} M_{ij}d_{[j]} + \sum_{j \in \mathcal{N}_{u,i}} B_{cen,ij}u_{cen,[j]} \tag{2.5}$$

$$y_{[i]} = C_{ii}x_{[i]} + D_{ii}u_{[i]} + \sum_{j \in \mathcal{N}_i} (C_{ij}x_{[j]} + D_{ij}u_{[j]}) + \sum_{j \in \mathcal{N}_{d,i}} N_{ij}d_{[j]} + \sum_{j \in \mathcal{N}_{u,i}} D_{cen,ij}u_{cen,[j]} \tag{2.6}$$

where $\mathcal{N}_{u,i} \subset \mathbb{N}$ is the set of centralized control inputs $u_{cen,[j]}$, $j \in \mathbb{R}$ that act on system i and $B_{cen,ij} \in \mathbb{R}^{n_i}$ and $D_{cen,ij} \in \mathbb{R}^{p_i}$.

The difference between $u_{[i]}$ and $u_{cen[i]}$ is that $u_{[i]}$ is a local control input, i.e. the output of a local regulator for subsystem i while $u_{cen[j]}$ is a global input that cannot be computed locally.
From models (2.5) and (2.6), the collective dynamics of the resulting LSS is

$$\mathbf{x}^+ = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} + \mathbf{M}\mathbf{d} + \mathbf{B}_{\mathbf{cen}}\mathbf{u}_{\mathbf{cen}} \tag{2.7}$$

$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u} + \mathbf{N}\mathbf{d} + \mathbf{D}_{\mathbf{cen}}\mathbf{u}_{\mathbf{cen}} \tag{2.8}$$

where:

$$\mathbf{x} = (x_{[1]},\ x_{[2]},\ \ldots,\ x_{[s]}) \in \mathbb{R}^n \tag{2.9}$$

is the overall state and $n = \sum_{i\in\mathcal{M}} n_i$,

$$\mathbf{u} = (u_{[1]},\ u_{[2]},\ \ldots,\ u_{[s]}) \in \mathbb{R}^m \tag{2.10}$$

is the overall control input and $m = \sum_{i\in\mathcal{M}} m_i$,

$$\mathbf{y} = (y_{[1]},\ y_{[2]},\ \ldots,\ y_{[s]}) \in \mathbb{R}^p \tag{2.11}$$

is the overall output and $p = \sum_{i\in\mathcal{M}} p_i$,

$$\mathbf{d} \in \mathbb{R}^{n_d} \tag{2.12}$$

collects the exogenous inputs acting on the overall system and

$$\mathbf{u}_{\mathbf{cen}} \in \mathbb{R}^{n_u} \tag{2.13}$$

collects the centralized centralized inputs acting on the overall system.
Moreover one has:

$$\mathbf{A} = \begin{bmatrix} A_{11} & \ldots & A_{1s} \\ \vdots & \ddots & \vdots \\ A_{s1} & \ldots & A_{ss} \end{bmatrix} \in \mathbb{R}^{n\times n} \qquad \mathbf{B} = \begin{bmatrix} B_{11} & \ldots & B_{1s} \\ \vdots & \ddots & \vdots \\ B_{s1} & \ldots & B_{ss} \end{bmatrix} \in \mathbb{R}^{n\times m} \tag{2.14}$$

All other matrices in (2.7) and (2.8) have a similar block structure. Moreover has $\mathbf{C} \in \mathbb{R}^{p\times n}$, $\mathbf{D} \in \mathbb{R}^{p\times m}$, $\mathbf{M} \in \mathbb{R}^{n\times n_d}$, $\mathbf{N} \in \mathbb{R}^{p\times n_d}$, $\mathbf{B}_{cen} \in \mathbb{R}^{n\times n_u}$ and $\mathbf{D}_{cen} \in \mathbb{R}^{p\times n_u}$.
The matrix $\mathbf{A}$ can also be split into matrices $\mathbf{A}_d$ and $\mathbf{A}_c$, defined as:

$$\mathbf{A}_D = \begin{bmatrix} A_{11} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & A_{ss} \end{bmatrix} \qquad \mathbf{A}_C = \mathbf{A} - \mathbf{A}_D \tag{2.15}$$

Apparently $\mathbf{A}_D$ collects all the local dynamics of the subsystems, while $\mathbf{A}_C$ collects coupling terms between subsystems. The same decomposition can be done for the matrices $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$ appearing in (2.7) and (2.8).
Each subsystem can be equipped with the following constraints:

$$x_{[i]} \in \mathbb{X}_i \subseteq \mathbb{R}^{n_i} \tag{2.16a}$$

$$u_{[i]} \in \mathbb{U}_i \subseteq \mathbb{R}^{m_i} \tag{2.16b}$$

$$y_{[i]} \in \mathbb{Y}_i \subseteq \mathbb{R}^{p_i} \tag{2.16c}$$

We furthermore assume that $\mathbb{X}_i$, $\mathbb{U}_i$ and $\mathbb{Y}_i$ are polyhedra.

It is also possible to define coupling constraints between two or more subsystems. In the case of two subsystems (i and j) we have

$$(x_{[i]}, x_{[j]}) \in \mathbb{X}_{ij} \subseteq \mathbb{R}^{n_i+n_j} \tag{2.17a}$$

$$(u_{[i]}, u_{[j]}) \in \mathbb{U}_{ij} \subseteq \mathbb{R}^{m_i+m_j} \tag{2.17b}$$

$$(y_{[i]}, y_{[j]}) \in \mathbb{Y}_{ij} \subseteq \mathbb{R}^{p_i+p_j} \tag{2.17c}$$

Constraints for the global lss system can be defined as follow

$$\mathbf{x} \in \mathbb{X} \subseteq \mathbb{R}^n \tag{2.18a}$$

$$\mathbf{u} \in \mathbb{U} \subseteq \mathbb{R}^m \tag{2.18b}$$

$$\mathbf{y} \in \mathbb{Y} \subseteq \mathbb{R}^p \tag{2.18c}$$

where

$$\mathbb{X} = \Big(\prod_{i \in \mathcal{M}} \mathbb{X}_i \Big) \cap \Big(\bigcap_{i,j \in \mathcal{M}, i \neq j} \mathbb{X}_{ij} \Big) \tag{2.19a}$$

$$\mathbb{U} = \Big(\prod_{i \in \mathcal{M}} \mathbb{U}_i \Big) \cap \Big(\bigcap_{i,j \in \mathcal{M}, i \neq j} \mathbb{U}_{ij} \Big) \tag{2.19b}$$

$$\mathbb{Y} = \Big(\prod_{i \in \mathcal{M}} \mathbb{Y}_i \Big) \cap \Big(\bigcap_{i,j \in \mathcal{M}, i \neq j} \mathbb{Y}_{ij} \Big) \tag{2.19c}$$

Moreover we can define constraints for the exogenous inputs and for the centralized control inputs:

$$\mathbf{d} \in \mathbb{D} \subseteq \mathbb{R}^{n_d} \tag{2.20a}$$

$$\mathbf{u}_{cen} \in \mathbb{U}_{cen} \subseteq \mathbb{R}^{n_u} \tag{2.20b}$$

Recalling Definition 1 we have matrices $H_x, H_u, H_u, H_d, H_{cen}$ and vectors $K_x, K_u, K_u, K_d, K_{cen}$ of suitable dimensions such that

$$\mathbb{X} = \{\mathbf{x} \in \mathbb{R}^n : H_x \mathbf{x} \leq K_x\}. \tag{2.21a}$$

$$\mathbb{U} = \{\mathbf{u} \in \mathbb{R}^m : H_u \mathbf{u} \leq K_u\}. \tag{2.21b}$$

$$\mathbb{Y} = \{\mathbf{y} \in \mathbb{R}^p : H_y \mathbf{y} \leq K_y\}. \tag{2.21c}$$

$$\mathbb{D} = \{\mathbf{d} \in \mathbb{R}^{n_d} : H_d \mathbf{d} \leq K_d\}. \tag{2.21d}$$

$$\mathbb{U}_{cen} = \{\mathbf{u}_{cen} \in \mathbb{R}^{n_u} : H_{cen} \mathbf{u}_{cen} \leq K_{cen}\}. \tag{2.21e}$$

Moreover, for each subsystem $i \in \mathcal{M}$ there are matrices $H_{x_i}, H_{u_i}, H_{y_i}$ and vectors $K_{x_i}, K_{u_i}, K_{y_i}$ such that

$$\mathbb{X}_i = \{x_{[i]} \in \mathbb{R}^{n_i} : H_{x_i} x_{[i]} \leq K_{x_i}\}. \tag{2.22a}$$

$$\mathbb{U}_i = \{u_{[i]} \in \mathbb{R}^{m_i} : H_{u_i} u_{[i]} \leq K_{u_i}\}. \tag{2.22b}$$

$$\mathbb{Y}_i = \{y_{[i]} \in \mathbb{R}^{p_i} : H_{y_i} y_{[i]} \leq K_{y_i}\}. \tag{2.22c}$$

In a similar way, for $d_{[j]}$, $j \in 1 : n_d$ we have

$$\mathbb{D}_j = \{d_{[j]} \in \mathbb{R} : H_{d,j} d_{[j]} \leq K_{d,j}\} \tag{2.23}$$

and for all $u_{cen,[j]}$, $j \in 1 : n_u$ it holds

$$\mathbb{U}_{cen_j} = \{u_{cen,[j]} \in \mathbb{R}^{n_{u_j}} : H_{cen,j} u_{cen,[j]} \leq K_{cen,j}\} \tag{2.24}$$

Coupling constraints have a totally analogous representation.

Chapter 3

Modeling of LSS: the lss and subss classes

3.1 The lss class

There are many MatLab toolboxes that offer facilities for modeling MIMO and LTI systems. However, most of them have not been designed to handle the interconnection of several subsystems. Therefore it is not immediate to manage an LSS, that means to add/remove subsystems, to extract a subsystem, to set coupling terms, etc. In the PnPMPC toolbox we have implemented methods for these tasks so as to ease the modeling of systems in the form (2.7) and (2.8). Moreover, the system object stores the constraints and some other useful pieces of information.
The main properties of an `lss` object are:

- `numSys`: the number s of subsystems composing the LSS;
- `Ts`: the sampling time;
- all matrices appearing in (2.7) and (2.8);
- `ni, mi, pi`: vectors of s elements which collect the numbers of states, inputs and outputs of every subsystem as in (2.9), (2.10) and (2.11);
- `numExo, numICen` are respectively n_d in (2.12) and n_u (2.13);
- all constraint matrices `H` and `K` appearing in (2.21);
- `Hdeltau,Kdeltau` that will be explained in Section 3.2.4;
- `couplingmatrix, name`, that will be explained in Section 3.1;
- `namex, nameu, namey, named, nameucen` in order to assign names to variables (see Section 3.1).

Memory optimization

Taking into account that usually LSS have a sparse structure and therefore matrices in (2.7) and (2.8) have many zero elements, we decided to use the sparse matrices of MatLab, by default. This

usually allows one to save a considerable amount of memory. To visualize the matrices in the full format, the MatLab command `full` can be used.
To save further memory we also used the following trick. If in (2.8) one has $\mathbf{y} = \mathbf{x}$ then:

- the matrix $\mathbf{C}$ is the identity matrix of order n
- $\mathbf{D}$ is a matrix of zeros $\in \mathbb{R}^{n \times m}$
- there are no exogenous inputs or centralized control inputs acting on the output.

Therefore, we decided to not save these matrices. However if the user requires them with a `get` method, they will be generated upon demand.
In the following, this particular form of the output equation will be called the *standard setting*.

Model discretization

Continuous-time models can be discretized using the method `clss2dlss` (that means continuous LSS to discrete LSS), with this declaration

```
dobjlss = objlss.clss2dlss(Ts,method)
```

where `Ts` is the sampling time and `method` is the discretization technique. All the discretization techniques of the function `c2d` of the Control System Toolbox (which converts continuous-time systems to discrete-time models) are implemented, like zoh (zero order hold) or Tustin (see `help c2d` for more information). The `c2d` discretization methods are not always a good choice for LSS, because for exact discretization one has to compute the exponential of matrix $\mathbf{A}$. Since $\mathbf{A}$ is usually a large matrix with many zero elements, one has that

- computing the exponential is time-consuming,
- elements that are zeros in the matrix $\mathbf{A}$ of the continuous-time model can be different from zero in the exponential matrix. Therefore, exact discretization creates coupling between subsystems that were originally decoupled.

To avoid these problems we also implemented system-by-system discretization [10]. That means that each subsystem in (2.5) and (2.6) is discretized considering states $x_{[j]},\ j \neq i$ as exogenous inputs. We can save computational time because we discretize sequentially subsystems that usually have low dimension and we do not create new couplings among subsystems. For using this technique, set '`subsystem`' as `method`.

Other properties

There are some other properties of `lss` objects that we have not explained yet.

- `couplingmatrix` is a matrix with size $numSys \times numSys$ composed by boolean elements: element (i, j) is `true` only if subsystem i is dynamically coupled or input coupled to subsystem j, i.e. $(i, j) \in \xi$.
- `name` is a cell array of strings with size $1 \times numSys$, that associates a name to every subsystem in the lss object. It is possible to set name when calling the `addSystem` method.

If a name is not supplied to `addSystem`, a default name will be chosen. Every method that needs a subsystem as input, has been designed to use these names as an alternative to subsystem numbers. However, it is better to use numbers when possible, because MatLab is slower when working with strings.

- `namex, nameu, namey, named, nameucen` are cell arrays of strings, each with dimension $1 \times numSys$. The element in position i stores information related to i-th subsystem. For example `namexi` is a cell array of n_i strings which stores the name associated to every state of i-th subsystem. Similarly, remarks can be applied to properties `nameu`, `namey`, `named`, `nameucen` that are related to local control inputs, outputs, exogenous inputs and centralized control inputs. Names can be written with the LaTeXsyntax and are also used in plots of subsystems or time-evolution of individual variables.

3.2 How to use the lss class through examples

In the following sections, we provide several examples with comments that illustrate the modeling process with the PnPMPC toolbox. Section titles corresponds to m-files that can be found in the `./examples/lss` directory.

3.2.1 example1lss.m

We show how to model the two coupled mass-spring-damper systems represented in Figure 3.1.

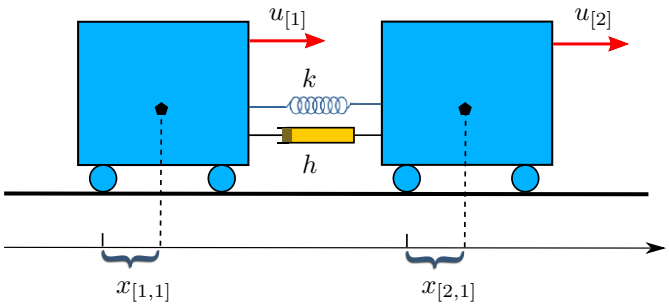

Figure 3.1: Array of 2 masses: details of interconnections.

The model is

$$\begin{bmatrix} \dot{x}_{[1]} \\ \dot{x}_{[2]} \end{bmatrix} = \mathbf{A} \begin{bmatrix} x_{[1]} \\ x_{[2]} \end{bmatrix} + \mathbf{B} \begin{bmatrix} u_{[1]} \\ u_{[2]} \end{bmatrix} \tag{3.1}$$

where $x_{[1]}$ is composed by $x_{[1,1]}$, the displacement of first mass with respect to a given equilibrium position, and $x_{[1,2]}$, the velocity of the first mass. Similarly $x_{[2]}$ is composed by $x_{[2,1]}$ and $x_{[2,2]}$. Moreover $u_{[1]}$ and $u_{[2]}$ are forces applied to the first and second mass, respectively. We also have

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad A_{11} = A_{22} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{M} & -\frac{h}{M} \end{bmatrix}, \quad A_{12} = A_{21} = \begin{bmatrix} 0 & 0 \\ \frac{k}{M} & \frac{h}{M} \end{bmatrix} \tag{3.2}$$

$$\mathbf{B} = \mathrm{diag}(B_{11}, B_{22}), \quad B_{11} = B_{22} = \begin{bmatrix} 0 \\ \frac{100}{M} \end{bmatrix} \tag{3.3}$$

First of all we present the methods that will be used.

The lss method

`lss` is the constructor of the homonymous class, with this declarations

```
objlss=lss(A,B,ni,mi,C,D,pi,name,Ts)
```

where

- `A` and `B` are the matrices in (2.7);
- `ni` and `mi` are vectors storing number of local states and local control inputs for each subsystem;
- `C` and `D` are the matrices in (2.8);
- `pi` is a vector storing number of outputs for each subsystem;
- `name` is an optional string defining the system name;
- `Ts` is the sampling time, that can be:
  - `Ts` = [ ] for continuous-time system;
  - `Ts` > 0 for discretized system with sampling time `Ts`;
  - `Ts` = -1 for system already in the discrete-time form.

It is possible to introduce directly the matrices of the collective system system (2.7) and (2.8) but this operation is usually time consuming and error-prone. Hence in the example we will start from an empty object and add subsystems with `addSystem` method.

The addSystem method

The `addSystem` method adds to an `lss` object a single subsystem. The method declaration is

```
objlss = objlss.addSystem(Ai,Bi,Ci,Di,name,Ts)
```

where the subsystem matrices are `Ai, Bi, Ci, Di` as defined in (2.5) and (2.6). The arguments `name` and `Ts` have the same meaning as in the `lss` class. We also highlight that `addSystem` could also receive one input argument only: it can be a `subss` object (see Section 3.3) or a `ss` object (state-space object of Control System Toolbox) or a `tf` object (transfer function object of Control System Toolbox).

The addCoupling method

By using `addSystem` method only the block diagonal part of the matrices **A**, **B**, **C** and **D** in (2.7) and (2.8) can be set. If subsystem i is coupled to subsystem j, one or more blocks among A_{ij}, B_{ij}, C_{ij} or D_{ij} are non zero. The method `addCoupling` allows one to set them.
The method declaration is

```
objlss = objlss.addCoupling(i,j,Aij,Bij,Cij,Dij)
```

where `i` and `j` represent indexes of the the coupled subsystems. It is possible to use subsystem names instead of numbers `i` and `j`.

Code for modeling system (3.1)

With the three methods presented above it is possible create the model of coupled masses.

```
M = 5;    % mass
k = 0.5; % elastic constant of the springs
h = 5;    % damping coefficient

% creation of an empty lss object
modelCart = lss;

A = [ 0 , 1; -k/M , -h/M ]; B=[ 0; 100/M];

% system 1
modelCart = modelCart.addSystem(A,B);

% system 2
modelCart = modelCart.addSystem (A,B);

% coupling among subsystems 1 and 2
Aij = [ 0 , 0 ; k/M , h/M ];
modelCart = modelCart.addCoupling(1,2,Aij);
modelCart = modelCart.addCoupling(2,1,Aij);

% plot graph of subsystems
modelCart.plot
```

In general, the declaration of the methods contains many optimal parameters, that are optional and the user can provide only meaningful quantity. In the example above, parameters `Ci,Di,name,Ts` are not supplied to `addSystem` method, because we are assuming $\mathbf{y} = \mathbf{x}$, so we are in the "standard setting" discussed in Section 3.1. Moreover, we do not want to associate a `name` to the model and we are defining a model in continuous time, so that `Ts` is empty.
Similar remarks apply to all methods of the toolbox.

3.2.2 example2lss.m

Now we will add an exogenous input and a centralized control input to the lss object modelCart through example1lss.m.
With reference to (2.7) and (2.8), we want to set

$$\mathbf{M} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 16 & 0 \\ 2 & 3 \end{bmatrix}, \quad \mathbf{N} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad \mathbf{B}_{cen} = \begin{bmatrix} 0 \\ 4 \\ 0 \\ 5 \end{bmatrix}, \tag{3.4}$$

Like in the previous section, we start with the description of the relevant methods.

The addExoInput method

The `addExoInput` method is used to set matrices $\mathbf{M}$ and $\mathbf{N}$.
The method declaration is

```
objlss = objlss.addExoInput(M,N,i,j)
```

and it can be used in the following two ways:

- `objlss = objlss.addExoInput(M,N)`, in this case we set the global `M` $\in \mathbb{R}^{n \times n_d}$ and `N` $\in \mathbb{R}^{p \times n_d}$ matrices of the LSS, according to (2.7) and (2.8).
- `objlss = objlss.addExoInput(M,N,i,j)`, in this case we set only blocks M_{ij} and N_{ij} of matrices $\mathbf{M}$ and $\mathbf{N}$ (2.5) and (2.6), i.e. the relation among subsystem `i` and the exogenous signal `j`, defined (2.5) and (2.6). The relation among all the other subsystems different from `i` and exogenous signal `j` is automatically set as zero if not defined yet.

The addCentralizedInput method

The `addCentralizedInput` method is used to set matrices $\mathbf{B}_{cen}$ and $\mathbf{D}_{cen}$.
The method declaration is:

```
objlss = objlss.addCentralizedInput(Bcen,Dcen,i,j)
```

and it can be used in the following two ways:

- `objlss = objlss.addCentralizedInput(Bcen,Dcen)`, in this case we set the global `Bcen` $\in \mathbb{R}^{n \times n_u}$ and `Dcen` $\in \mathbb{R}^{p \times n_u}$ matrices of the LSS, according to (2.7) and (2.8).
- `objlss = objlss.addCentralizedInput(Bcen,Dcen,i,j)`, in this case we set $B_{cen_{ij}}$ and $D_{cen_{ij}}$ of matrices $\mathbf{B}_{cen}$ and $\mathbf{D}_{cen}$ (2.5) and (2.6), i.e. the relation among subsystem `i` and the centralized input `j`, defined (2.5) and (2.6). The relation among all the other subsystems different from `i` and centralized control input `j` is automatically set as zero if not already defined.

```
example1lss;
% specify how exogenous input number 1 affects system 1 (creation of M)
modelCart = modelCart.addExoInput ([0;1],[], 1,1);

% specify how exogenous input number 1 affects system 2 (creation of M)
modelCart = modelCart.addExoInput ([16;2],[],2,1);

% specify how exogenous input number 2 affects system 2 (creation of M)
modelCart = modelCart.addExoInput ([0;3],[],2,2);

% instead of calling addExoInput three times we can use
% modelCart = modelCart.addExoInput ([0 0; 1 0; 16 0; 2 3],[]);

% specify how centralized input number 1 affects system 1 (creation of Bcen)
modelCart = modelCart.addCentralizedInput([0;4],[],1,1);

% specify how centralized input number 1 affects system 2 (creation of Bcen)
modelCart = modelCart.addCentralizedInput ([0;5],[],2,1);

% instead of the previous calls to addCentralizedInput we can use
% modelCart = modelCart.addCentralizedInput ([0;4;0;5],[]);
```

```
% plot all signals
modelCart.plot([],'all')
```

If, like in this case, the user does not need to set the `N` or `Dcen` matrix, it is possible to use empty matrices. Empty or not passed parameters are given default values. The same remark applies to all other methods in the toolbox.

3.2.3 example3lss.m

This example shows how to exploit modularity in the subsystem interconnection for quickly creating an LSS. The code below allows one to create a string of N mass-spring-damper systems (as shown in Figure 3.2) with different M(mass), different k (elastic constant of the springs) and different h (damping coefficient), verifying the bounds $minM \leq M \leq maxM$, $mink \leq k \leq maxk$ and $minh \leq h \leq maxh$.

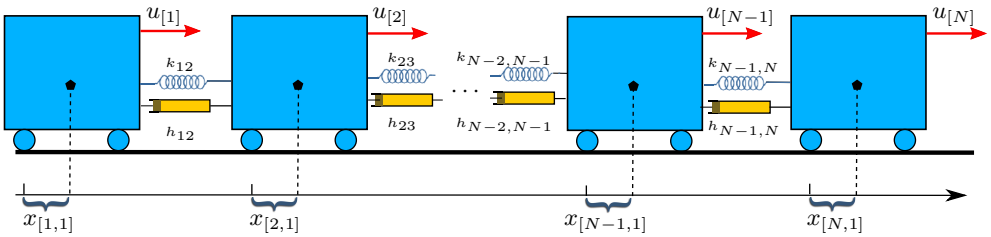

Figure 3.2: Large-scale system to model in example3lss.

The system is characterized by the matrices:

$$
\begin{aligned}
&A_{11} = \begin{bmatrix} 0 & 1 \\ -\frac{k(1)}{M(1)} & -\frac{h(1)}{M(1)} \end{bmatrix}, \quad A_{NN} = \begin{bmatrix} 0 & 1 \\ -\frac{k(N-1)}{M(N)} & -\frac{h(N-1)}{M(N)} \end{bmatrix} \\
&\forall i = 2 : N-1 \\
&A_{ii} = \begin{bmatrix} 0 & 1 \\ -\frac{k(i-1)+k(i)}{M(i)} & -\frac{h(i-1)+h(i)}{M(i)} \end{bmatrix}, \quad B_{ii} = \begin{bmatrix} 0 \\ \frac{100}{M(i)} \end{bmatrix} \\
&A_{(i-1),i} = \begin{bmatrix} 0 & 0 \\ -\frac{k(i-1)}{M(i-1)} & -\frac{h(i-1)}{M(i-1)} \end{bmatrix}, \quad A_{i,(i-1)} = \begin{bmatrix} 0 & 0 \\ -\frac{k(i-1)}{M(i)} & -\frac{h(i-1)}{M(i)} \end{bmatrix} \\
&C_{ii} = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad D_{ii} = 0;
\end{aligned}
\tag{3.5}
$$

The MatLab code, when $N = 1000$, is reported in the following script.

```
N     = 1000;
minM  = 1;
maxM  = 10;
mink  = 0.1;
maxk  = 0.9;
minh  = 0.1;
maxh  = 10;

M = minM+rand(1,N)*(maxM-minM);
```

```
k = mink+rand(1,N-1)*(maxk-mink);
h = minh+rand(1,N-1)*(maxh-minh);

modelCart = lss;
modelCart = modelCart.addSystem([0,1;-k(1)/M(1),-h(1)/M(1)],[0;100/M(1)],...
                    [1 0],0);
for i=2:N-1
    modelCart = modelCart.addSystem([0,1;-(k(i-1)+k(i))/M(i),...
                        -(h(i-1)+h(i))/M(i)],[0;100/M(i)],[1 0],0);
    modelCart = modelCart.addCoupling(i-1,i,[0,0;k(i-1)/M(i-1),...
                        h(i-1)/M(i-1)],zeros(2,1));
    modelCart = modelCart.addCoupling(i,i-1,[0,0;k(i-1)/M(i),h(i-1)/M(i)],...
                        zeros(2,1));
end
modelCart = modelCart.addSystem([0,1;-k(N-1)/M(N),-h(N-1)/M(N)],...
                    [0;100/M(N)],[1 0],0);
modelCart = modelCart.addCoupling(N,N-1,[0,0;k(N-1)/M(N),h(N-1)/M(N)],...
                    zeros(2,1));
modelCart = modelCart.addCoupling(N-1,N,[0,0;k(N-1)/M(N-1),h(N-1)/M(N-1)],...
                    zeros(2,1));
```

3.2.4 example4lss.m

This example shows how to add state and control input constraints to an lss model. The addition of constraints on output, exogenous input and centralized control input can be done in a similar way. Relevant methods are described next.

Methods for adding constraints

The `addStateConstraint` method sets the matrices H_{x_i} and K_{x_i} in (2.22). The method declaration is:

```
objlss = objlss.addStateConstraint(sysindex,H,K)
```

Where `sysindex` is the number or name of the subsystem to which constraints have to be added and `H` and `K` are the matrices H_{x_i} and K_{x_i} in (2.22). It is also possible to create constraints between two or more subsystems, in this case `sysindex` is a vector of scalars, or a cell array of names of the subsystems involved.
Methods that allow one to add constraints to control inputs, outputs, exogenous inputs and centralized control inputs.

- `objlss = objlss.addInputConstraint(sysindex,H,K)`
- `objlss = objlss.addOutConstraint(sysindex,H,K)`. Note that if the user tries to insert a constraint on the output, but the system is in standard setting ($\mathbf{y} = \mathbf{x}$), the constraint will be converted in a state constraint and a warning will be displayed.
- `objlss = objlss.addExoConstraint(exoindex,H,K)`
- `objlss = objlss.addCentralizedInputConstraint(ceninputindex,H,K)`

and works exactly as `addStateConstraint`.

The "double" methods

The *double* methods return matrices H and K related to constraints. The name "double" has been used for coherency with the MPT2 toolbox [11]. The method declaration is:

```
[Hx,Kx] = objlss.doubleStateConstraint(index,selection)
```

and it returns `Hx` and `Kx` matrix related to the states of the subsystem/subsystems expressed in index. `index` can be a scalar or a string, but also a vector of scalars or a cell array of names, if one is interested in coupling constraints. `selection` is a Boolean flag with the following meaning

- 0: exclusive (default). The state constraints only of the subsystem/subsystems indicated in `index` will be returned
- 1: all. All state constraints which includes the subsystem/subsystems indicated in `index` will be returned.

For example, let c_1 be a constraint related to subsystem 1, c_3 be a constraint related to subsystem 3 and c_{13} be a coupling constraint between subsystem 1 and 3. If `index` = 3 and `selection`=0 (or not passed) `H` and `K` contain only c_3 because this is the only constraint that involves subsystem 3 only. If `selection` = 1, `H` and `K` will contain c_3 and c_{13}, i.e. all constraints involving subsystem number 3. Another possible use of this method is:

```
X = objlss.doubleStateConstraint(index,selection)
```

with only one output argument. In this case, `X` is the Polyhedron (i.e. an object of Polyhedron class, see the documentation of the MPT3 for help) of subsystem specified in index giving rise to a state constraint. Also if `index` contains more than one index, a Polyhedron object will be returned.

Other double methods with similar meaning and use are

- `[Hu,Ku] = objlss.doubleInputConstraint(index,selection)`
- `[Hy,Ky] = objlss.doubleOutConstraint(index,selection)`
- `[Hd,Kd] = objlss.doubleExoConstraint(index,selection)`
- `[Hcen,Kcen]= objlss.doubleCenInputConstraint(index,selection)`

Moreover, the following methods return matrices `Hd, Kd` and `Hcen, Kcen` for exogenous inputs and centralized control inputs acting on subsystem `index`.

- `[Hd,Kd] = objlss.doubleExoConstraintOnSystem(index)`
- `[Hcen,Kcen]= objlss.doubleCenInputConstraintOnSystem(index)`

Next, we provide the code for adding constraints (described in the code comments) to the modelCart object.

```
example2lss;

% 0(x_3)+1(x_4)≤3
modelCart = modelCart.addStateConstraint (2,[0,1],3);

% 2(x_3)+1(x_4)≤0 and 1(x_3)+3(x_4)≤5
modelCart = modelCart.addStateConstraint (2,[2,1;1,3],[0;5]);

% 0(x_1)+1(x_2)+1(x_3)+2(x_4)≤0 & 3(x_1)+2(x_2)+0(x_3)+1(x_4)≤5
modelCart = modelCart.addStateConstraint ([1,2],[0,1,1,2;3,2,0,1],[0;5]);

% 2(u_2)≤3
modelCart = modelCart.addInputConstraint (2,2,3);

% 1(u_1)+2(u_2)≤3
modelCart = modelCart.addInputConstraint ([1,2],[1,2],3);

[Hx Kx] = modelCart.doubleStateConstraint(1)
disp('Hx and Kx are empty because there is no constraint affecting system 1
       only')

[Hu Ku] = modelCart.doubleInputConstraint(2,1)
disp('Hu and Ku contain 2(u_2)≤3 and 1(u_1) + 2(u_2)≤3 because selection is
        1 and we consider all the constraints related to system 2')
```

Input increment

In many systems, rather than constraining the input, it is preferable to bound the input increments $\delta u_{[i]} = u_{[i]}(k+1) - u_{[i]}(k)$. With `addDeltaUConstraint` is possible to define matrices $H_{\delta u_i}$ and $K_{\delta u_i}$ giving rise to the constraints $\{H_{\delta u_i}\delta u_{[i]} \leq K_{\delta u_i}\}$.
The method declaration is
`objlss = objlss.addDeltaUConstraint(sysindex,Hdeltau,Kdeltau)`
And the corresponding "double" method is
`[Hdeltau,Kdeltau] = objlss.doubleDeltaUConstraint(index,selection)`

3.2.5 example5lss.m

All the methods described above are useful to add subsystems, couplings and constraints to an lss object. But if the user inserts wrong parameters it can be useful to remove subsystems features in a selective way. This is the goal of the methods discussed next.

The removeCoupling method

`objlss = objlss.removeCoupling(i,j,how)`

This method allows one to remove coupling between subsystems `i` and `j`. The variable `how` can be `'a'` or `'b'` or `'c'` or `'d'` and specifies if we want to remove coupling terms in the matrices `A`,

B, C, D, respectively. If how is not passed, blocks (i, j) in all matrices A, B, C, D will be removed. See help removeCoupling for more information.

The removeExoSignal method

```
objlss = objlss.removeExoInput(i)
```

This method allows one to remove the exogenous input $d_{[i]}$. It deletes the column i of matrices M and N and also the constraints related to this exogenous inputs (if they exist). Note that a coupling constraint related to exogenous inputs i and j, after the removal of the i-th signal becomes a local constraint of the j-th signal. For example, let $d_{[1]} + 2d_{[2]} \leq 10$ be a constraint among exogenous signals $d_{[1]}$ and $d_{[2]}$; the execution of objlss = objlss.removeExoSignal(1) removes $d_{[1]}$ in the constraint, i.e. the new constraint is $2d_{[2]} \leq 10$. If one wants to remove all constraints where signal $d_{[i]}$ appears, the removeConstraint method can be used.

The removeCentralizedInput method

```
objlss = objlss.removeCentralizedInput(i)
```

This method allows one to deletes the i-th centralized control input. It eliminates the column i of matrices Bcen and Dcen and also the constraints related to this centralized control input (if they exist). Note that a coupling constraint related to centralized control inputs i and j, after the removal of i becomes a local constraint of j. If one wants to remove all constraints where centralized input $u_{cen,[i]}$ appears, the removeConstraint method can be used.

The removeConstraint method

```
objlss = objlss.removeConstraint(index,flag)
```

This method allows one to remove constraints. index can be, as usually, the subsystem number or name, if we want to remove local constraints. But it can also be a vector of scalars or a cell array of names, if we want to remove coupling constraints. flag can be 'state' or 'in' or 'out' or 'exo' or 'cen' or 'deltau'. In this way it is possible to remove, for instance, state constraints only. If flag is 'exo' or 'cen' we refer to the constraints of exogenous input $d_{[i]}$ or centralized control input $u_{cen,[i]}$, and in this case we have to specify the number of signal/signals interested. If flag is not specified all constraints on states, inputs and outputs of the system in index will be removed. Fore example if flag is empty and index is 1, then $\mathbb{X}_1$, $\mathbb{U}_1$ and $\mathbb{Y}_1$ will be removed. See help removeConstraint for more information.

The removeSystem method

```
objlss = objlss.removeSystem(i)
```

This method allows one to remove the subsystem number i from an lss object. Note that if subsystem i is removed, this triggers the following changes.

- if an exogenous signal acts only on subsystem i the method removeExoSignal will be automatically called for removing the signal from the lss model and a warning appears on the screen. The same remark applies to centralized inputs.

- if two subsystems i and j are involved in a coupling constraint, and i system is removed, the constraint becomes a local constraint of j. For example, let $u_{[1]} + u_{[2]} \leq 20$ be a constraint among subsystems 1 and 2; the execution of `objlss = objlss.removeSystem(1)` removes $u_{[1]}$ in the constraint, i.e. the new constraint is $u_{[2]} \leq 20$.

In this example we illustrate the use of the removal methods. We start from the model created in example4lss.

```
example4lss;

% remove exogenous signal number 2
modelCart = modelCart.removeExoSignal(2);

% remove centralized input number 1
modelCart = modelCart.removeCentralizedInput(1);

% remove coupling constraint on the state of system 1 and 2
modelCart = modelCart.removeConstraint([1,2],'state');

% remove dynamic coupling between 1 and 2 so A12 = zeros
modelCart = modelCart.removeCoupling(1,2,'a');

% remove subsystem number 2
modelCart = modelCart.removeSystem(2);
```

3.2.6 Performances of the methods dedicated to modeling

We discuss here performances, in terms of memory occupation and execution time of the methods of the PnPMPC toolbox dedicated to modeling. As a first case study, we use the string of N masses described in example1lss.m and example3lss.m with local constraints on states and inputs. In the left panel of Figure 3.3 one can see

- the execution time for creating the lss object (blue line),
- the execution time for discretizing the lss object using the system-by-system approach (red line)

as a function of the number of masses.

In the right panel of Figure 3.3 it is reported

- the memory occupation (in Kb) due to the continuous-time model (blue line),
- the memory occupation (in Kb) due to the discretized model (red line)

as a function of the number of masses.

These graphics have been obtained on a processor Intel Core i3 3.06 GHz, with 4GB of Ram 1.33 GHz running MatLab $r2011a$.

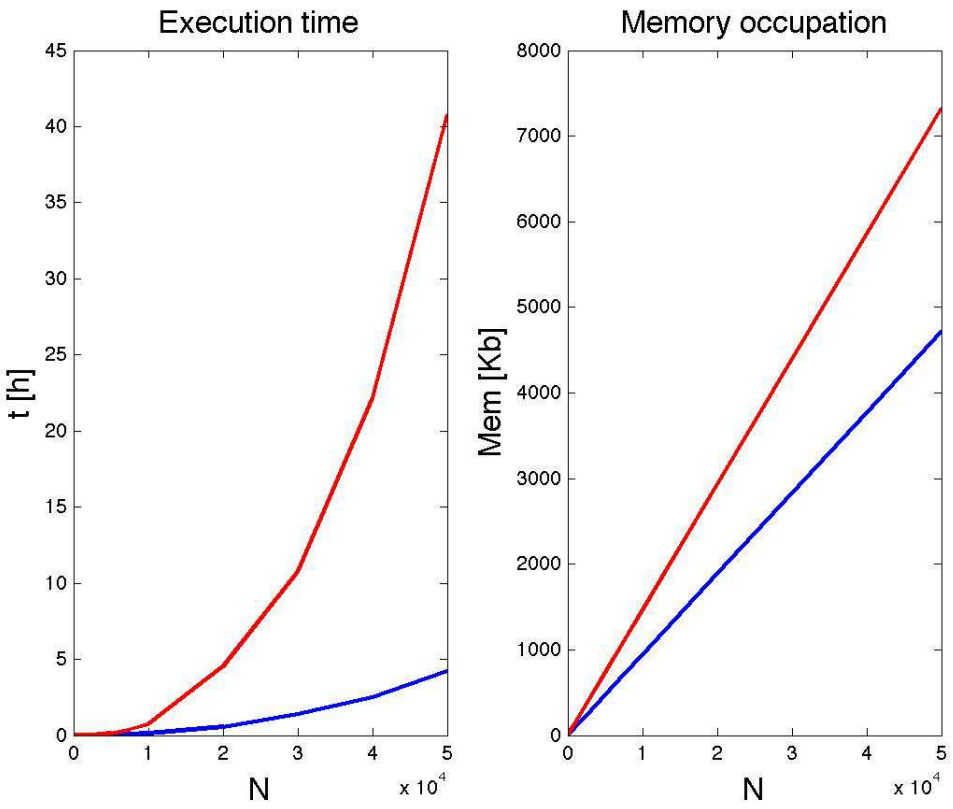

Figure 3.3: Performance of PnPMPC toolbox.

3.3 The subss class

This class allows one to model individual subsystems and it is useful for the following reasons.

1. Creating a `subss` object and pass it to an lss object with the `addSystem` method. In this way, dynamics and constraints of the subsystem will be added to the LSS.

2. Extracting a subsystem from an lss object through the `getSystem` method.

Many methods of this class have the same name of methods of the lss class and work in a similar way. In addition, many properties are the same as those of the lss class. Next we describe only the new ones referring the reader to the html documentation in the doc directory for a comprehensive description of all methods and properties. If we extract subsystem i from an lss object we lose coupling with other subsystems. Therefore we created the following properties to save

- `couplingA`, `couplingB`, `couplingC`, `couplingD` are scalar vectors containing the indices of the subsystems in $\mathcal{N}_i$ that are coupled with the extracted subsystem (i.e. subsystem i). `couplingA` will contain the indices of the subsystems coupled through the $\mathbf{A}$ matrix. `couplingB`, `couplingC` and `couplingD` have a similar meaning.

- `Aij`, `Bij`, `Cij`, `Dij`. `Aij` contains blocks A_{ij} where i is the number of the extracted subsystem and $j \in$ `couplingA`. Same remarks apply to the other properties.

3.3.1 example1subss.m

In this example we show how to add a subsystem to an lss object using a subss object.

```
example2lss;

A = [0,1;-k/M,-h/M];
B = [0;100/M];
sub = subss (A,B);
sub = sub.addStateConstraint([3,5;0,1],[1;2]);
modelCart = modelCart.addSystem(sub);
```

3.3.2 example2subss

After running example3lss, we can extract subsystem 237 as follows.

```
example3lss;

sub = modelCart.getSystem(237)
```

With the previous command, subsystem `sub` is an object of the class subss. In this case only local constraints are saved. For example constraints among subsystem 237 and 238 will not be saved. `sub` will have only the `couplingA` and `Aij` matrix, because the subsystem is dynamically coupled with 236 and 238, so
`sub.couplingA = [236 238]`
`sub.Aij` $=[A_{237,236}; A_{237,238}]$
where $A_{237,236}$ is the dynamic coupling among subsystems 237 and 236, and $A_{237,238}$ is the dynamic coupling among subsystems 237 and 238.

3.4 List of other useful methods of class `lss`

- `.addLocalInput` defines a new input for the i-th subsystem of an `lss` object.
- `.display` display properties of an `lss` object in the command window.
- `.eig` compute eigenvalues of an LSS.
- `.generateCouplingmatrix` recompute the property `.couplingmatrix` of an `lss` object.
- `get` methods, return properties of an `lss` object.
- `.isCoupled` check if two subsystems are coupled.
- `.join` join two `lss` objects.
- `.name2index` index of the subsystem called "name".
- `.order` order of an LSS.

- `.plot` plot graph of the network of subsystems.
- `.plotU`, `.plotUcen`, `.plotX`, `.plotY` using the plot function, plot signals that represent control inputs, centralized control inputs, states and outputs.
- `.stairsU`, `.stairsUcen`, `.stairsX`, `.stairsY` using the plot function, plot signals that represent control inputs, centralized control inputs, states and outputs.
- `set` methods, set properties of an `lss` object.
- `union` join two or more subsystems.

Chapter 4

The cenmpc class

4.1 Introduction

In this chapter, we introduce the `cenmpc` class in order to design an MPC controller for a LSS. The interested reader is refer to [12], [13], [14] and [15]. Details of the design procedures for robust tube MPC can be found in [16] and [17]. The model of the LSS is given by (2.7) and (2.8). Our control design procedure will hinge on the following assumptions.

Assumption 1. *The matrix pairs* $(\mathbf{A}, [\mathbf{B}\ \mathbf{B_{cen}}])$ *is controllable.*

Assumption 2. *Constraints* $\mathbb{X}$ *and* $\mathbb{U}$*,* $i \in \mathcal{M}$ *are compact and convex polytopes containing the origin in their nonempty interior.*

4.1.1 The cenmpc method

Assume `objlss` stores the model of an LSS given by (2.7) and (2.8). MPC controller is created through

```
objcenmpc = cenmpc(objlss,N,flag,options)
```

where `N` is the receding horizon,`options` is used to set the `sdpsettings` for YALMIP (see YALMIP manual for details [8]). The input argument `flag` is a structure with the following possible fields.

- `.ExoBounded`: indices of exogenous inputs acting on the LSS `objlss` that are disturbances, therefore `cenmpc` design a robust MPC controller based on tube as in [17].

`objcenmpc` is the MPC controller designed with options given in `flag`.

We propose three methods to compute the terminal region and the terminal penalty. We refer the reader to help of methods `XFqpmax`, `XFqpmaxDec` and `zeroTerminal`.

4.1.2 The uRH method

In order to compute the control action u we need to solve the MPC optimization problem. The control action is computed executing the function

```
[ u diagnostics ] = objcenmpc.uRH(x0,din,xrefin,urefin,options)
```

- `x0` is the initial local state $x(0)$, a vector of dimension $n \times 1$.
- `din` is the vector of exogenous inputs which act on the subsystem and they are not disturbances. Indices of these exogenous inputs are in `objcenmpc.Exo`.
- `xrefin` means the reference state of nominal system over the control horizon; it can be expressed as a vector ($n \times 1$, reference constant over the control horizon) or a matrix ($n \times N$, the columns 1 refer to the values at the time instant 1 within the control horizon, columns 2 refer to the values at time 2, etc.). If this parameter is empty or not passed, it is set to zero by default.
- `urefinin` means the reference input of nominal system over the control horizon; it can be expressed as a vector ($m \times 1$, reference constant over the control horizon) or a matrix ($m \times N$, the columns 1 refer to the values at the time instant 1 within the control horizon, columns 2 refer to the values at time 2, etc.). If this parameter is empty or not passed, it is setted to zero by default.
- `options` `sdpsettings` object for YALMIP options.

As output we have:

- u control input value;
- *diagnostics* information about the optimization problems.

4.2 Examples

Examples of `cenmpc` class can be found in the `./examples/cenmpc` directory.
Execute `example1cenmpc` for a deterministic MPC controller and `example2cenmpc` for tube-based MPC.

Chapter 5

The pnpmpc class

5.1 Introduction

In this chapter, we introduce the `pnpmpc` class in order to design a state-feedback PnPMPC controller for a subsystem. PnPMPC controllers are described in [1], [18], [3], [2], [19] and [4].

In those papers, the authors propose a PnP design procedure hinging on the notion of tube MPC [16] and [17] for handling coupling among subsystems, and aim at stabilizing the origin of the whole closed-loop system while guaranteeing satisfaction of constraints on local inputs and states. The model of subsystem i, $i \in \mathcal{M}$ is given by (2.3). Our control design procedure will hinge on the following assumptions.

Assumption 3. *The matrix pairs (A_{ii}, B_i) $\forall i \in \mathcal{M}$ are controllable.*

Assumption 4. *Constraints $\mathbb{X}_i$ and $\mathbb{U}_i$, $i \in \mathcal{M}$ are compact and convex polytopes containing the origin in their nonempty interior.*

Let $\mathbb{Z}_i$ be an RCI[1]set for the nominal[2] subsystem i where the coupling term $\bigoplus_{j\in\mathcal{N}_i} A_{ij}\mathbb{X}_j \oplus B_{ij}\mathbb{U}_j$ and bounded disturbances $M_i\mathbb{D}_i$ are treated as a disturbance. We design the following controller for subsystem i

$$\mathcal{C}_{[i]}: \quad u_{[i]} = v_{[i]} + \bar{\kappa}_i(x_{[i]} - \bar{x}_{[i]}) + \sum_{j\in\mathcal{N}_i} \delta_{ij} K_{ij} x_{[j]} \tag{5.1}$$

where $\bar{\kappa}_i : \mathbb{Z}_i \to \mathbb{U}_i$ is any feedback control law guaranteeing $x_{[i]} \in \mathbb{Z}_i \Rightarrow x^+{}_{[i]} \in \mathbb{Z}_i$, $\forall x_{[j]} \in \mathbb{X}_j$, $j \in \mathcal{N}_i$, $K_{ij} \in \mathbb{R}^{m_i \times n_j}$ and $\delta_{ij} \in \{0, 1\}$. Note that if $\delta_{ij} = 0$, $\forall i \in \mathcal{M}$, $\forall j \in \mathcal{N}_i$, the control scheme is completely decentralized, since $u_{[i]}$ depends on local quantities only.

Following [16] and [17], we set in (5.1)

$$\begin{aligned} v_{[i]}(t) &= v_{[i]}(0|t), \\ \bar{x}_{[i]}(t) &= \hat{x}_{[i]}(0|t) \end{aligned} \tag{5.2}$$

[1]**Robust Control Invariant (RCI) set** Consider the discrete-time Linear Time-Invariant (LTI) system $x(t+1) = Ax(t) + Bu(t) + w(t)$, with $x(t) \in \mathbb{R}^n$, $u(t) \in \mathbb{R}^m$, $w(t) \in \mathbb{W}^n$ and subject to constraints $u(t) \in \mathbb{U} \subseteq \mathbb{R}^m$ and $w(t) \in \mathbb{W} \subset \mathbb{R}^n$. The set $\mathbb{X} \subseteq \mathbb{R}^n$ is an RCI set with respect to $w(t) \in \mathbb{W}$, if $\forall x(t) \in \mathbb{X}$ then there exist $u(t) \in \mathbb{U}$ such that $x(t+1) \in \mathbb{X}$, $\forall w(t) \in \mathbb{W}$.

[2]Model of subsystem i without coupling terms.

where $v_{[i]}(0|t)$ and $\hat{x}_{[i]}(0|t)$ are optimal values of the variables $v_{[i]}(0)$ and $\hat{x}_{[i]}(0)$, respectively, appearing in the MPC-i problem

$$\mathbb{P}_i^N(x_{[i]}(t)) = \min_{\substack{v_{[i]}(0:N_i-1) \\ \hat{x}_{[i]}(0)}} \sum_{k=0}^{N_i-1} \ell_i(\hat{x}_{[i]}(k), v_{[i]}(k)) + V_{f_i}(\hat{x}_{[i]}(N_i)) \tag{5.3a}$$

$$x_{[i]}(t) - \hat{x}_{[i]}(0) \in \mathbb{Z}_i \tag{5.3b}$$

$$\hat{x}_{[i]}(k+1) = A_{ii}\hat{x}_{[i]}(k) + B_i v_{[i]}(k), \quad k \in 0 : N_i - 1 \tag{5.3c}$$

$$\hat{x}_{[i]}(k) \in \hat{\mathbb{X}}_i, \qquad k \in 0 : N_i - 1 \tag{5.3d}$$

$$v_{[i]}(k) \in \mathbb{V}_i, \qquad k \in 0 : N_i - 1 \tag{5.3e}$$

$$\hat{x}_{[i]}(N_i) \in \hat{\mathbb{X}}_{f_i} \tag{5.3f}$$

In (5.3), $N_i \in \mathbb{N}$ is the control horizon, $\ell_i : \mathbb{R}^{n_i \times m_i} \to \mathbb{R}_+$ is the stage cost, $V_{f_i} : \mathbb{R}^{n_i} \to \mathbb{R}_+$ is the final cost and $\hat{\mathbb{X}}_{f_i}$ is the terminal set. Moreover, from (5.3c) and (5.3e), the tightened constraints $\hat{\mathbb{X}}_i$ and $\mathbb{V}_i$ are defined respectively as

$$\hat{\mathbb{X}}_i = \mathbb{X}_i \ominus \mathbb{Z}_i, \qquad \mathbb{V}_i = \mathbb{U}_i \ominus \bar{\kappa}_i(\mathbb{Z}_i). \tag{5.4}$$

Therefore, in order to design a PnPMPC controller for subsystem i we need to solve the following problem.

Problem $\mathcal{P}_i$: Given δ_{ij}, compute matrices K_{ij} and nonempty RCI $\mathbb{Z}_i$ for the nominal subsystem i treating the coupling term as a disturbance. Compute sets $\hat{\mathbb{X}}_i$ and $\mathbb{V}_i$.

We highlight that Problem $\mathcal{P}_i$ can be solved using efficient procedure proposed in [20] that requires the solution of a suitable LP problem. Moreover $\hat{\mathbb{X}}_i$ and $\mathbb{V}_i$ can be computed using optimizers from the LP problem.

If Problem $\mathcal{P}_i$ cannot be solved, we declare that it is impossible to design a PnPMPC controller for subsystem i.

The interested reader is refer to [1], [18], [3], [2], [19] and [4].

5.1.1 The pnpmpc method

Assume `subss` stores the model of subsystem i given by (2.3) and (2.4). Controller $\mathcal{C}_{[i]}$ is created through

```
objpnpmpc = pnpmpc(subss,N,Xj,Uj,flag,options)
```

where `N` is the receding horizon, `Xj` is a cell matrix $\{H_{xj1}, K_{xj1}; H_{xj2}, K_{xj2}; \ldots\}$ such that parent states $x_{[j]}$, $j \in \mathcal{N}_i$ are in the set $Xj1 = \{xj1 | H_{xj1} xj1 \leq K_{xj1}\}$, $Xj2 = \{xj2 | H_{xj2} xj2 \leq K_{xj2}\}$, where $xj1$ and $xj2$ are states of two parent subsystems of the local subsystem `subss`. Similarly, `Uj` is a cell matrix collecting matrices describing input constraints of parent subsystems. `options` is used to set the `sdpsettings` for YALMIP (see YALMIP manual for details [8]). The input argument `flag` is a structure with the following possible fields.

- `.LPdesign`: `true` or `false`. If `true`, the design of local PnPMPC controllers is based on the use of LP procedure proposed in [2], [3] and Chapter 6 of [4], hence $\mathbb{Z}_i$ is an RCI set and function $\bar{\kappa}_i(\cdot)$ will be evaluated online. If `false`, the design of local PnPMPC controllers is based on the solution of a nonlinear optimization problem as proposed in [1] and Chapter 5 of [4]: in this case parent constraints must be zonotopes, $\mathbb{Z}_i$ is an RPI set and function $\bar{\kappa}_i(x_{[i]} - \bar{x}_{[i]})$ is a linear function $K_i(x_{[i]} - \bar{x}_{[i]})$.

- `.ExoBounded`: indices of exogenous inputs acting on subsystem `subss` that are disturbances, therefore `pnpmpc` design a robust PnPMPC controller as in Chapter 7 of [4].
- `.distributedParents`: indices of subsystems such that $\delta_{ij} = 1$ in (5.1). It must be a subset of the indices of parent subsystems.
- `.norm`: specify norm for the computation of matrices K_{ij} (see Chapter 7 in [4]).
- `.boundKij`: specify bound for the computation of matrices K_{ij} (see Chapter 7 in [4]).
- `.k`: if `.LPdesign=true`, `.k={kmin,kmax}` allows to try different `k` for the LP design procedure proposed in [2], [3] and Chapter 6 in [4].
- `.Qi` and `.Ri`: if `.LPdesign=false`, `Qi` and `Ri` are inputs for the nonlinear optimization problem as proposed in Chapter 5 in [4].

`objpnpmpc` is the PnPMPC controller designed with options given in `flag`.

We propose two methods to compute the terminal region and the terminal penalty.

5.1.2 The XFqpmax method

Using `XFqpmax` we compute a quadratic terminal region using procedures proposed in [21]. In particular we consider

$$\ell_i(\hat{x}_{[i]}(k), v_{[i]}(k)) = (\hat{x}_{[i]}(k) - \hat{x}^{ref}_{[i]}(k))' Q (\hat{x}_{[i]}(k) - \hat{x}^{ref}_{[i]}(k)) + (v_{[i]}(k) - v^{ref}_{[i]}(k)))' R (v_{[i]}(k) - v^{ref}_{[i]}(k)))$$

$$V_{f_i}(\hat{x}_{[i]}(N_i)) = (\hat{x}_{[i]}(N_i) - \hat{x}^{ref}_{[i]}(N_i))' S (\hat{x}_{[i]}(N_i) - \hat{x}^{ref}_{[i]}(N_i))$$

$$\hat{\mathbb{X}}_{f_i} = \{\beta \in \mathbb{R}^{n_i} : \beta' S \beta \leq 1\}$$

where $Q = Q' \geq 0 \in \mathbb{R}^{n_i \times n_i}$, $R = R' > 0 \in \mathbb{R}^{m_i \times m_i}$ and $S = S' \geq 0 \in \mathbb{R}^{n_i \times n_i}$. $\hat{x}^{ref}_{[i]}(k)$ and $v^{ref}_{[i]}(k)$ are the state and input reference trajectories for tracking capabilities.
We can design the terminal penalty S and the ellipsoidal terminal constraint executing the function

```
objpnpmpc = objpnpmpc.XFqpmax(Q,R,options)
```

5.1.3 The zeroTerminal method

Using `zeroTerminal` we compute a zero terminal constraint as proposed in Chapter 2 in [15]. In particular we consider

$$\ell_i(\hat{x}_{[i]}(k), v_{[i]}(k)) = (\hat{x}_{[i]}(k) - \hat{x}^{ref}_{[i]}(k))' Q (\hat{x}_{[i]}(k) - \hat{x}^{ref}_{[i]}(k)) + (v_{[i]}(k) - v^{ref}_{[i]}(k)))' R (v_{[i]}(k) - v^{ref}_{[i]}(k)))$$

$$V_{f_i}(\hat{x}_{[i]}(N_i)) = 0$$

$$\hat{\mathbb{X}}_{f_i} = \hat{x}^{ref}_{[i]}(N_i)$$

where $Q = Q' \geq 0 \in \mathbb{R}^{n_i \times n_i}$ and $R = R' > 0 \in \mathbb{R}^{m_i \times m_i}$. $\hat{x}^{ref}_{[i]}(k)$ and $v^{ref}_{[i]}(k)$ are the state and input reference trajectories for tracking capabilities.
We can design the terminal penalt and the ellipsoidal terminal constraint executing the function

```
objpnpmpc = objpnpmpc.zeroTerminal(Q,R)
```

5.1.4 The uRH method

In order to compute the control action $u_{[i]}$ we need to solve the optimization problem (5.3). The control action is computed executing the function

```
[ u diagnostics ] = objpnpmpc.uRH(x0,xj,xjinv,din,xcrefin,vrefin,options)
```

- `x0` is the initial local state $x_{[i]}(0)$, a vector of dimension $n_i \times 1$.
- `xj` is the vector of states from parent subsystems such that $K_{ij} \neq 0$. Indices of these parent subsystems are in `objpnpmpc.Nij`.
- `xjinv` is a cell array where each element is the state of a parent subsystem. Useful only if `.LPdesign=true` and there are no disturbances. This input will be used to evaluate function $\bar{\kappa}_i(x_{[i]} - \bar{x}_{[i]}, \{x_{[j]}\}_{j \in \mathcal{N}_i})$ (for more details see Chapter 6 of [4]).
- `din` is the vector of exogenous inputs which act on the subsystem and they are not disturbances. Indices of these exogenous inputs are in `objpnpmpc.Exo`.
- `xcrefin` means the reference state of nominal system ($\hat{x}_{ref}$) over the control horizon; it can be expressed as a vector ($n_i \times 1$, reference constant over the control horizon) or a matrix ($n_i \times N$, the columns 1 refer to the values at the time instant 1 within the control horizon, columns 2 refer to the values at time 2, etc.). If this parameter is empty or not passed, it is set to zero by default.
- `vrefinin` means the reference input of nominal system (v_{ref}) over the control horizon; it can be expressed as a vector ($m_i \times 1$, reference constant over the control horizon) or a matrix ($m_i \times N$, the columns 1 refer to the values at the time instant 1 within the control horizon, columns 2 refer to the values at time 2, etc.). If this parameter is empty or not passed, it is set to zero by default.
- `options` sdpsettings object for YALMIP options.

As output we have:

- u control input value;
- *diagnostics* information about the optimization problems.

5.2 Design of PnPMPC controllers for a large-scale system

PnPMPC controllers $\mathcal{C}_{[i]}$ for each subsystem in an `lss` object are created with the method `createCtrlPnPMPC` described next.

5.2.1 The createCtrlPnPMPC method

This method acts on an lss object, and it is called as follows:

```
ctrl = createCtrlPnPMPC(objlss,N,flag,option)
```

where N, flag, options have the same meaning as in the pnpmpc method. This method extracts a subsystem, calls the constructor of class pnpmpc and design the controller. The output ctrl is an array of dimension $1 \times$numSys and in the i-th position the controller for subsystem i is stored. After the creation of the controllers, we can run XFqpmax or zeroTerminal methods, for setting terminal constraints, and then the uRH method, to compute the control inputs.

5.3 Examples

As an example of real application, in Chapter 6, we describe PnPMPC controllers for power network systems and run simulations using methods proposed in this chapter. In this section we propose simpler example. Section titles corresponds to m-files that can be found in the ./examples/pnpmpc directory.

5.3.1 example1pnpmpc.m

We consider a large-scale system composed of N masses coupled as in Figure 5.1.

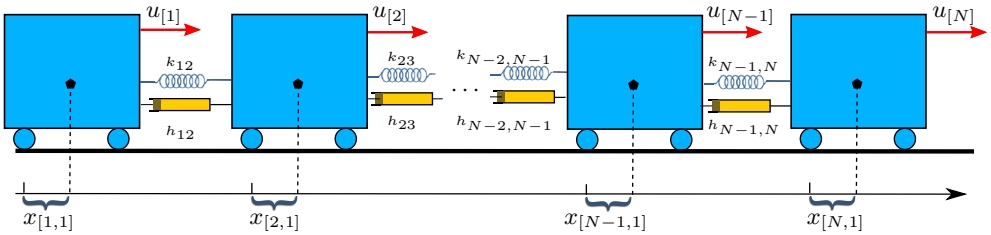

Figure 5.1: Array of masses: details of interconnections.

Each mass $i \in \mathcal{M} = 1 : N$, is a subsystem with state variables $x_{[i]} = (x_{[i,1]}, x_{[i,2]})$ and input $u_{[i]} = u_{[i,1]}$, where $x_{[i,1]}$ is the displacements of mass i with respect to a given equilibrium position, $x_{[i,2]}$ is the horizontal velocity of the mass i and $100u_{[i,1]}$ is the force applied to mass i in the horizontal direction. The values of m_i have been extracted randomly in the interval $[5, 10]$ while spring constants and damping coefficients are identical and equal to 0.5. Subsystems are equipped with the state constraints $||x_{[i,1]}||_\infty \leq 1.5$, $||x_{[i,2]}||_\infty \leq 0.8$, $i \in \mathcal{M}$ and with the input constraints $|u_{[i]}| \leq \beta_i$, where β_i have been randomly chosen in the interval $[1, 1.5]$. We obtain models $\Sigma_{[i]}$ by discretizing continuous-time models with 0.2 sec sampling time, using zero-order hold discretization for the local dynamics and treating $x_{[j]}$, $j \in \mathcal{N}_i$ as exogenous signals. At all time steps t, the control action $u_{[i]}(t)$ computed by the controller $\mathcal{C}_i$, for all $i \in \mathcal{M}$, is kept constant during the sampling interval and applied to the continuous-time system.

Executing file example1pnpmpc, the following functions will be executed. In order to create the large-scale system composed by N masses, we use the function makeModelTrucks1D as follows

```
N = 5;
[ modelTrucks1Dc modelTrucks1Dd ] = makeModelTrucks1D(N);
```

where N is the number of trucks, modelTrucks1Dc and modelTrucks1Dd are the lss objects of the continuous-time and discrete-time models, respectively in form of lss objects.

We can design local controllers executing the function

```
options = sdpsettings('verbose',0);
Trucks1Dpnp  = makePnpmpcTrucks1D(modelTrucks1Dd,options);
```

whose instructions are described next. More in detail, the controllers, $\mathcal{C}_{[i]}$, $i \in \mathcal{M}$ are generated as follows. `Trucks1Dpnp` is an array of N `pnpmpc` objects.

```
NMPC = 10; % prediction horizon
Trucks1Dpnp = modelTrucks1Dd.createCtrlPnPMPC(NMPC);
```

Then we add a zero terminal constraint to each controller using the following instructions.

```
Q = 10*eye(2); R = 1;
parfor i=1:modelTrucks1Dd.numSys
     Trucks1Dpnp(i) = Trucks1Dpnp(i).zeroTerminal(Q,R);
end
```

We note that the functions `makePnpmpcTrucks1D` or `modelTrucks1Dd.createCtrlPnPMPC` could take some minutes to be completely executed if `N` is large. If you want take less time for the execution, we recommend to install CPLEX optimizer [22] that is free available for academic use.

To start a simulation, we use the function `runSimTrucks1D` in the following way

```
x0       = repmat([1 0]',N,1);
Tsim     = 30;
[ x u ] = runSimTrucks1D(Tsim,x0,modelTrucks1Dc,modelTrucks1Dd,Trucks1Dpnp);
```

This function computes the control action using the method `uRH` of the pnpmpc class. We really recommend the use of CPLEX.

5.3.2 example2pnpmpc.m

In this example, differently from `example1pnpmpc.m`, we consider that each subsystem is affected by bounded disturbances. Therefore in order to design robust PnPMPC controllers, we select the bounded exogenous inputs, using the struct `flag` and its field `.ExoBounded`.

5.3.3 example3pnpmpc.m

We consider a large-scale system composed by N masses coupled as in Figure 5.3 (for the case of $N = 1024$) where the four edges connected to a point correspond to springs and dampers arranged as in Figure 5.2. Hereafter we assume that $N = z^2$ for some $z \in \mathbb{N}$, $z > 1$. Each mass $i \in \mathcal{M} = 1 : N$, is a subsystem with state variables $x_{[i]} = (x_{[i,1]}, x_{[i,2]}, x_{[i,3]}, x_{[i,4]})$ and input $u_{[i]} = (u_{[i,1]}, u_{[i,2]})$, where $x_{[i,1]}$ and $x_{[i,3]}$ are the displacements of mass i with respect to a given equilibrium position on the plane (equilibria lie on a regular grid), $x_{[i,2]}$ and $x_{[i,4]}$ are the horizontal and vertical velocity of the mass i, respectively, and $100u_{[i,1]}$ (respectively $100u_{[i,2]}$) is the force applied to mass i in the horizontal (respectively, vertical) direction. The values of m_i have been extracted randomly in the interval $[8, 10]$ while spring constants and damping coefficients are identical and equal to 0.5. Subsystems are equipped with the state constraints $||x_{[i,j]}||_\infty \leq 1.5$,

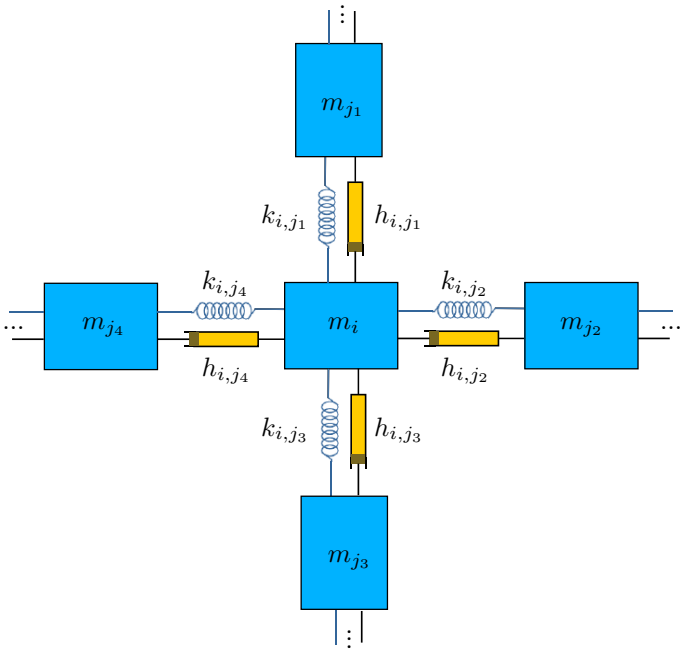

Figure 5.2: Array of masses: details of interconnections.

$j = 1, 3$, $||x_{[i,l]}||_\infty \leq 0.8$, $i \in \mathcal{M}$, $l = 2, 4$ and with the input constraints $||u_{[i]}||_\infty \leq \beta_i$, where β_i have been randomly chosen in the interval $[1, 1.5]$. We obtain models $\Sigma_{[i]}$ by discretizing continuous-time models with 0.2 sec sampling time, using zero-order hold discretization for the local dynamics and treating $x_{[j]}$, $j \in \mathcal{N}_i$ as exogenous signals [10]. At all time steps t, the control action $u_{[i]}(t)$ computed by the controller $\mathcal{C}_i$, for all $i \in \mathcal{M}$, is kept constant during the sampling interval and applied to the continuous-time system.

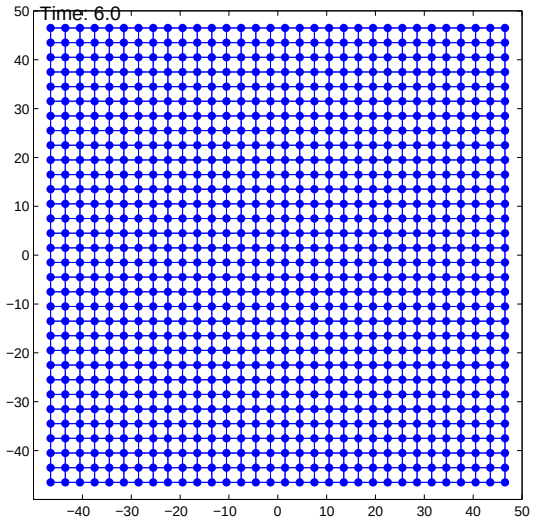

Figure 5.3: Position of the $N = 1024$ trucks on the plane.

In order to create the large-scale system composed by N masses, we can use the function

makeModelTrucks2D as follows

```
N = 16;
[ modelTrucks2Dc modelTrucks2Dd ] = makeModelTrucks2D(N);
```

where N is the number of trucks, modelTrucks2Dc and modelTrucks2Dd are the continuous-time and discrete-time large-scale models, respectively in form of lss objectes.

We can design local controllers executing the function

```
Trucks2Dpnp  = makePnpmpcTrucks2D(modelTrucks2Dd);
```

whose instructions are described next.

Controllers $\mathcal{C}_{[i]}$, $i \in \mathcal{M}$ are generated as follows. Trucks2Dpnp is an array of N pnpmpc objects.

```
NMPC = 10; % prediction horizon
Trucks2Dpnp = modelTrucks2Dd.createCtrlPnPMPC(NMPC);
```

Then, we add a zero terminal constraint to each controller using the following instructions.

```
Q = 10*eye(4); R = eye(2);
parfor i=1:modelTrucks2Dd.numSys
     Trucks2Dpnp(i) = Trucks2Dpnp(i).zeroTerminal(Q,R);
end
```

We note that the functions makePnpmpcTrucks2D or modelTrucks2Dd.createCtrlPnPMPC could take some minutes to be completely executed. If you want take less time for the execution, we recommend to install CPLEX optimizer [22] that is free available for academic use.

To start a simulation, we can use the function runSimTrucks2D in the following way

```
x0         = repmat([1 0 -1 0]',N,1);
Tsim     = 30;
[ x u ] = runSimTrucks2D(Tsim,x0,modelTrucks2Dc,modelTrucks2Dd,Trucks2Dpnp);
```

This function computes the control action using the method uRH of the pnpmpc class. We did not specify a solver in order to be as general as possible. We really recommend the use of CPLEX.

In [2], [3] and Chapter 6 of [4], we propose the case of $N = 1024$ trucks and we give the results in terms of computational times.

Chapter 6

Hycon2 Benchmark: Power Network System

Note: for MatLab simulations in this chapter, we recommend the use of CPLEX optimizer [22].

6.1 Introduction

An example of a real application that can benefit of decentralized and distributed control schemes is the regulation of a Power Network System (PNS). Here we describe the PNS proposed as a benchmark exercise [3] within the HYCON2 project [23].

We consider a PNS as composed by several power generation areas coupled through tie-lines [24]. The aim is to design the Automatic Generation Control (AGC) layer for frequency control with the goal of:

- keeping the frequency approximately at the nominal value;
- controlling the tie-line powers in order to reduce power exchanges between areas. In the asymptotic regime each area should compensate for local load steps and produce the required power.

We consider thermal power stations with single-stage turbines. The dynamics of an area equipped with primary control and linearized around equilibrium value for all variables can be described by the following continuous-time LTI model [24]

$$\Sigma^C_{[i]}: \quad \dot{x}_{[i]} = A_{ii}x_{[i]} + B_i u_{[i]} + L_i \Delta P_{L_i} + \sum_{j \in \mathcal{N}_i} A_{ij} x_{[j]} \tag{6.1}$$

where $x_{[i]} = (\Delta\theta_i,\ \Delta\omega_i,\ \Delta P_{m_i},\ \Delta P_{v_i})$ is the state, $u_{[i]} = \Delta P_{ref_i}$ is the control input of each area, ΔP_L is the local power load and $\mathcal{N}_i$ is the sets of neighbouring areas, i.e. areas directly connected

to $\Sigma^C_{[i]}$ through tie-lines. The matrices of system (6.1) are defined as

$$A_{ii}(\{P_{ij}\}_{j\in\mathcal{N}_i}) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\frac{\sum_{j\in\mathcal{N}_i} P_{ij}}{2H_i} & -\frac{D_i}{2H_i} & \frac{1}{2H_i} & 0 \\ 0 & 0 & -\frac{1}{T_{t_i}} & \frac{1}{T_{t_i}} \\ 0 & -\frac{1}{R_i T_{g_i}} & 0 & -\frac{1}{T_{g_i}} \end{bmatrix} \qquad B_i = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{1}{T_{g_i}} \end{bmatrix}$$

$$A_{ij} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{P_{ij}}{2H_i} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad L_i = \begin{bmatrix} 0 \\ -\frac{1}{2H_i} \\ 0 \\ 0 \end{bmatrix} \tag{6.2}$$

For the meaning of constants as well as some typical parameter values we refer the reader to Table 6.1.

$\Delta\theta_i$	Deviation of the angular displacement of the rotor with respect to the stationary reference axis on the stator
$\Delta\omega_i$	Speed deviation of rotating mass from nominal value
ΔP_{m_i}	Deviation of the mechanical power from nominal value (p.u.)
ΔP_{v_i}	Deviation of the steam valve position from nominal value (p.u.)
ΔP_{ref_i}	Deviation of the reference set power from nominal value (p.u.)
ΔP_{L_i}	Deviation of the nonfrequency-sensitive load change from nominal value (p.u.)
H_i	Inertia constant defined as $H_i = \frac{\text{kinetic energy at rated speed}}{\text{machine rating}}$ (typically values in range $[1-10]$ sec)
R_i	Speed regulation
D_i	Defined as $\frac{\text{percent change in load}}{\text{change in frequency}}$
T_{t_i}	Prime mover time constant (typically values in range $[0.2-2]$ sec)
T_{g_i}	Governor time constant (typically values in range $[0.1-0.6]$ sec)
P_{ij}	Slope of the power angle curve at the initial operating angle between area i and area j

Table 6.1: Variables of a generation area with typical value ranges [24]. (p.u.) stands for "per unit".

We note that model (6.1) is input decoupled since both ΔP_{ref_i} and ΔP_{L_i} act only on subsystem $\Sigma^C_{[i]}$. Moreover, subsystems $\Sigma^C_{[i]}$ are parameter dependent since the local dynamics depends on the quantities $-\frac{\sum_{j\in\mathcal{N}_i} P_{ij}}{2H_i}$.

In the following we introduce three scenarios corresponding to different interconnection topologies of generation areas. The model parameters and constraints on $\Delta\theta_i$ and on ΔP_{ref_i} for systems in all Scenarios are given in Table 6.2. We highlight that all parameter values are within the range of those used in Chapter 12 of [24]. We define M as the number of areas in the power network. For each scenario, discrete-time models $\Sigma_{[i]}$ with $T_s = 1$ sec sampling time are obtained from $\Sigma^C_{[i]}$ using two alternative discretization schemes.

- Exact discretization of the overall system (acronym D).
- Discretization system-by-system, i.e. exact discretization for each area treating $u_{[i]}$, ΔP_{L_i} and $x_{[j]}$, $j \in \mathcal{N}_i$ as exogenous inputs (acronym Dss) (see [10] for more information).

In particular, we note that Dss preserves the input-decoupled structure of $\Sigma^C_{[i]}$ while D does not.

6.1.1 Scenario 1

We consider four areas interconnected as in Figure 6.1. We will simulate Scenario 1 using the load steps specified in Table 6.3.

	Area 1	Area 2	Area 3	Area 4	Area 5
H_i	12	10	8	8	10
R_i	0.05	0.0625	0.08	0.08	0.05
D_i	0.7	0.9	0.9	0.7	0.86
T_{t_i}	0.65	0.4	0.3	0.6	0.8
T_{g_i}	0.1	0.1	0.1	0.1	0.15

	Area 1	Area 2	Area 3	Area 4	Area 5
$\Delta\theta_i$	$\|\|x_{[1,1]}\|\|_\infty \leq 0.1$	$\|\|x_{[2,1]}\|\|_\infty \leq 0.1$	$\|\|x_{[3,1]}\|\|_\infty \leq 0.1$	$\|\|x_{[4,1]}\|\|_\infty \leq 0.1$	$\|\|x_{[5,1]}\|\|_\infty \leq 0.1$
ΔP_{ref_i}	$\|\|u_{[1]}\|\|_\infty \leq 0.5$	$\|\|u_{[2]}\|\|_\infty \leq 0.65$	$\|\|u_{[3]}\|\|_\infty \leq 0.65$	$\|\|u_{[4]}\|\|_\infty \leq 0.55$	$\|\|u_{[5]}\|\|_\infty \leq 0.5$

$$P_{12} = 4 \qquad P_{23} = 2 \qquad P_{34} = 2 \qquad P_{45} = 3 \qquad P_{25} = 3$$

Table 6.2: Model parameters and constraints for systems $\Sigma_{[i]},\ i \in 1, \ldots, 5$.

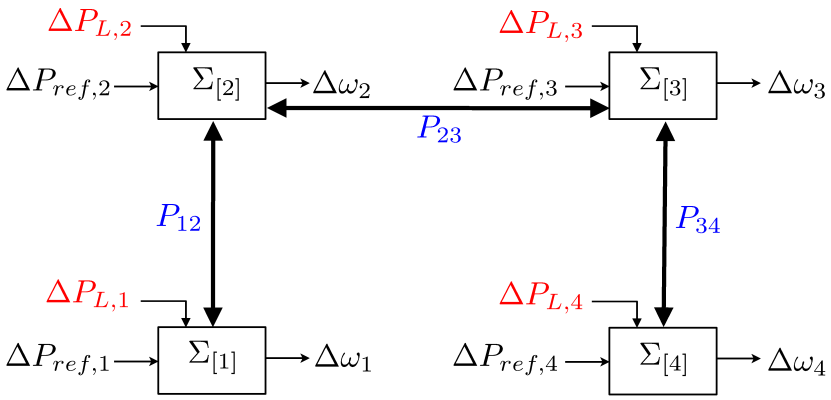

Figure 6.1: Power network system of Scenario 1

Step time	Area i	ΔP_{L_i}
5	1	+0.15
15	2	-0.15
20	3	+0.12
40	3	-0.12
40	4	+0.28

Table 6.3: Load of power ΔP_{L_i} (p.u.) for simulation in Scenario 1. $+\Delta P_{L_i}$ means a step of required power, hence a decrease of the frequency deviation $\Delta\omega_i$ and therefore an increase of the power reference ΔP_{ref_i}.

6.1.2 Scenario 2

We consider the power network proposed in Scenario 1 and add a fifth area connected as in Figure 6.2. We will simulate Scenario 2 using the load steps specified in Table 6.4.

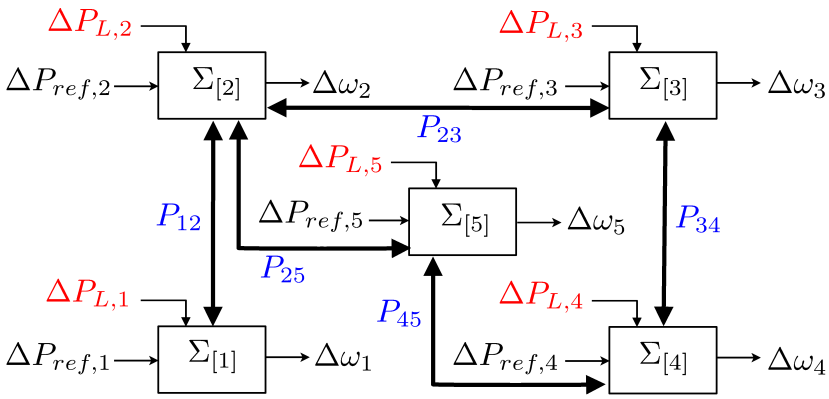

Figure 6.2: Power network system of Scenario 2

Step time	Area i	ΔP_{L_i}
5	1	+0.10
15	2	-0.16
20	1	-0.22
20	2	+0.12
20	3	-0.10
30	3	+0.10
40	4	+0.08
40	5	-0.10

Table 6.4: Load of power ΔP_{L_i} (p.u.) for simulation in Scenario 2. $+\Delta P_{L_i}$ means a step of required power, hence a decrease of the frequency deviation $\Delta\omega_i$ and therefore an increase of the power reference ΔP_{ref_i}.

6.1.3 Scenario 3

We consider the power network described in Scenario 2 and disconnect the area 4, hence obtaining the areas connected as in Figure 6.3. We will simulate Scenario 3 using load steps specified in Table 6.5.

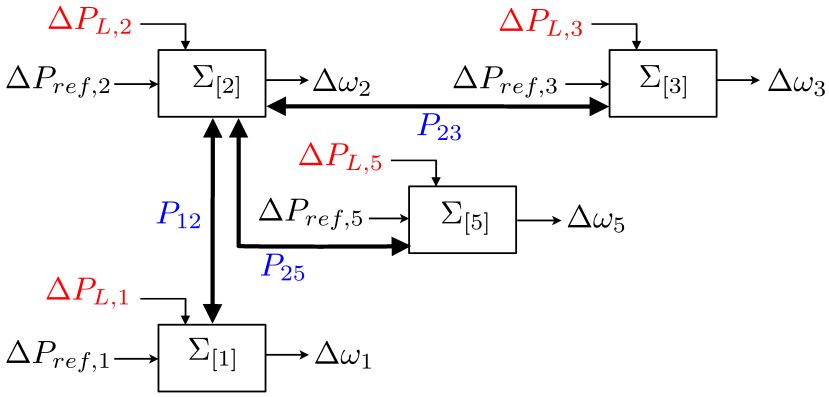

Figure 6.3: Power network system of Scenario 3

Step time	Area i	ΔP_{L_i}
5	1	+0.12
15	2	-0.15
20	5	+0.20
40	2	+0.15
40	3	+0.13
40	5	-0.20

Table 6.5: Load of power ΔP_{L_i} (p.u.) for simulation in Scenario 3. $+\Delta P_{L_i}$ means a step of required power, hence a decrease of the frequency deviation $\Delta\omega_i$ and therefore an increase of the power reference ΔP_{ref_i}.

We can create lss objects for each scenario, running `makeScenariosPNS`. For the first scenario we create 3 files: `pnsC1` (continuous-time lss object), `pnsD1` (discrete-time lss object), `pnsD1ss` (discrete-time system-by-system lss object). Similar files are created for scenario 2 and 3.

6.2 Design of the AGC layer for a power network using MPC

The goal of the Benchmark is to design the AGC layer for the scenarios introduced in Section 6.1. Different control schemes will be compared with the centralized MPC scheme described next. For a given Scenario, at time t we solve the centralized optimization problem

$$\mathbb{P}^N(\mathbf{x}(t)): \tag{6.3a}$$

$$\min_{\mathbf{u}(t:t+N-1)} \sum_{k=t}^{t+N-1} (||\mathbf{x}(k)-\mathbf{x}^{\mathbf{O}}||_{\mathbf{Q}}+||\mathbf{u}(k)-\mathbf{u}^{\mathbf{O}}||_{\mathbf{R}})+||\mathbf{x}(t+N)-\mathbf{x}^{\mathbf{O}}||_{\mathbf{S}}) \tag{6.3b}$$

$$\mathbf{x}(k+1)=\mathbf{A}\mathbf{x}(k)+\mathbf{B}\mathbf{u}(k)+\mathbf{L}\mathbf{\Delta}\mathbf{P}_{\mathbf{L}}(t) \qquad k\in 0:N-1 \tag{6.3c}$$

$$\mathbf{x}(k)\in\mathbb{X} \qquad k\in 0:N-1 \tag{6.3d}$$

$$\mathbf{u}(k)\in\mathbb{U} \qquad k\in 0:N-1 \tag{6.3e}$$

$$\mathbf{x}(N)\in\mathbb{X}_f \tag{6.3f}$$

and then apply $\Delta P_{ref}=\mathbf{u}(0)$. We note that the cost function depend upon $\mathbf{x}^{\mathbf{O}}$ and $\mathbf{u}^{\mathbf{O}}$ that are defined as $x^O_{[i]}=(0,\ 0,\ \Delta P_{L_i},\ \Delta P_{L_i})$ and $u^O_{[i]}=\Delta P_{L_i}$. The constraints $\mathbb{X}$ and $\mathbb{U}$ in (6.3d) and (6.3e) are obtained from constraints listed in Table 6.2. In the cost function (6.3b) we set $N=15$, $\mathbf{Q}=\text{diag}(Q_1,\ldots,Q_M)$ and $\mathbf{R}=\text{diag}(R_1,\ldots,R_M)$, where

$$Q_i=\begin{bmatrix}500 & 0 & 0 & 0\\ 0 & 0.01 & 0 & 0\\ 0 & 0 & 0.01 & 0\\ 0 & 0 & 0 & 10\end{bmatrix} \text{ and } R_i=10.$$

Weights Q_i and R_i have been chosen in order to penalize the angular displacement $\Delta\theta_i$ and to penalize slow reactions to power load steps. Since the power transfer between areas i and j is

given by

$$\Delta P_{tie_{ij}}(k) = P_{ij}(\Delta\theta_i(k) - \Delta\theta_j(k)) \tag{6.4}$$

the first requirement also penalizes huge power transfers.

In order to guarantee the stability of the closed loop system, we design the matrix S and the terminal constraint set $\mathbb{X}_f$ in three different ways.

- $\mathbf{S}$ *is full (MPCfull)*: we compute the symmetric positive-definite matrix $\mathbf{S}$ and the static state-feedback auxiliary control law $\mathbf{K_{aux}x}$, by maximizing the volume of the ellipsoid described by $\mathbf{S}$ inside the state constraints while fulfilling the matrix inequality

$$(\mathbf{A} + \mathbf{BK_{aux}})'\mathbf{S}(\mathbf{A} + \mathbf{BK_{aux}}) - \mathbf{S} \leq -\mathbf{Q} - \mathbf{K'_{aux}RK_{aux}}$$

  .

- $\mathbf{S}$ *is block diagonal (MPCdiag)*: we compute the decentralized symmetric positive-definite matrix $\mathbf{S}$ and the decentralized static state-feedback auxiliary control law $\mathbf{K_{aux}x}$, $\mathbf{K_{aux}} = \text{diag}(K_1, \ldots, K_M)$ by maximizing the volume of the ellipsoid described by $\mathbf{S}$ inside the state constraints while fulfilling the matrix inequality

$$(\mathbf{A} + \mathbf{BK_{aux}})'\mathbf{S}(\mathbf{A} + \mathbf{BK_{aux}}) - \mathbf{S} \leq -\mathbf{Q} - \mathbf{K'_{aux}RK_{aux}}$$

  .

- *Zero terminal constraint (MPCzero)*: we set $\mathbf{S} = 0$ and $\mathbb{X}_f = \mathbf{x^O}$.

6.2.1 Performance criteria

We propose the following performance criteria for evaluating different control schemes.

- η*-index*

$$\eta = \frac{1}{T_{sim}} \sum_{k=0}^{T_{sim}-1} \sum_{i=1}^{M} (||x_{[i]}(k) - x^O_{[i]}(k)||_{Q_i} + ||u_{[i]}(k) - u^O_{[i]}(k)||_{R_i}) \tag{6.5}$$

  where T_{sim} is the time of the simulation. From (6.5), η is a weighted average of the error between the real state and the equilibrium state and between the real input and the equilibrium input.

- Φ*-index*

$$\Phi = \frac{1}{T_{sim}} \sum_{k=0}^{T_{sim}-1} \sum_{i=1}^{M} \sum_{j \in \mathcal{N}_i} |\Delta P_{tie_{ij}}(k)| T_s \tag{6.6}$$

  where T_{sim} is the time of the simulation and $\Delta P_{tie_{ij}}$ is the power transfer between areas i and j defined in (6.4). This index gives the average power transferred between areas. In particular, if the η-index is equal for two regulators, the best controller is the one that has the lower value of Φ.

6.3 Control Experiments

We applied the centralized MPC schemes introduced in the previous section to scenarios 1, 2 and 3. Furthermore, for each scenario we discretized the continuos system with both discretization schemes D and Dss. At time t we solve the optimization problem (6.3) and then apply the control action to the continuos-time system, keeping the value constant between time t and $t+1$. If at time t the power load increases or decreases, we assume the controller can use this information at time t. This means at time t the controller knows exactly the value of ΔP_L hence can use it. We highlight that violation of this assumption can impact considerably on the index η. In all experiments we use $T_{sim} = 100$. In Table 6.6 and 6.7 the values of the performance parameters η and Φ, respectively, are reported for each control experiment.

	Scenario 1		Scenario 2		Scenario 3	
	D	Dss	D	Dss	D	Dss
$MPCfull$	0.0249	0.0249	0.0346	0.0347	0.0510	0.0511
$MPCdiag$	0.0249	0.0249	0.0346	0.0347	0.0510	0.0511
$MPCzero$	0.0249	0.0249	0.0346	0.0347	0.0510	0.0511

Table 6.6: Values of the performance parameter η using different centralized MPC schemes for the AGC layer.

	Scenario 1		Scenario 2		Scenario 3	
	D	Dss	D	Dss	D	Dss
$MPCfull$	0.0030	0.0029	0.0063	0.0060	0.0060	0.0058
$MPCdiag$	0.0030	0.0029	0.0063	0.0061	0.0060	0.0058
$MPCzero$	0.0030	0.0028	0.0063	0.0059	0.0059	0.0058

Table 6.7: Values of the performance parameter Φ using different centralized MPC schemes for the AGC layer.

6.4 Supporting MatLab files

In terms of PnPMPC controllers, running file

```
makePnpmpcControllersPNS( type )
```

we can create controllers with different terminal regions and features. The input `type` is a number from 1 to 6.

1. Create decentralized PnPMPC controllers based on LP design and equipping each subsystem with input constraints.
2. Create decentralized PnPMPC controllers based on LP design and without input constraints.
3. Create distributed PnPMPC controllers based on LP design and without input constraints.
4. Create decentralized PnPMPC controllers based on nonlinear design and equipping each subsystem with input constraints.

5. Create decentralized PnPMPC controllers based on nonlinear design and without input constraints.

6. Create distributed PnPMPC controllers based on nonlinear design and without input constraints.

Then we can run each simulation, executing file *scenario[number scenario][Full—Zero]*. For example if we want run the simulation for Scenario 2 using PnPMPC controllers with zero terminal constraints, we should run

```
scenario2Zero (options)
```

then a file with all simulation data is created. We note that using PnPMPC controllers we refer to $full$ when using a quadratic terminal region for controller $\mathcal{C}_{[i]}$, $i \in \mathcal{M}$. Moreover we consider discretization system-by-system only. `options` is a `sdpsettings` object for YALMIP. The data of a simulation are saved in a file *scenario[number scenario][Full—Zero]_sim[type simulation]PNS*, where *type simulation* depends on the PnP controller design. In particular *type simulation* could be a number from 1 to 4.

1. In the simulation, decentralized local controllers do not receive state of parent subsystems.

2. In the simulation, if decentralized local controllers have been designed with LP design, we receive state of parent subsystems (as in Section 6.5 in [4]).

3. In the simulation, distributed local controllers do not receive state of parent subsystems.

4. In the simulation, if distributed local controllers have been designed with LP design, we receive state of parent subsystems (as in Section 6.5 in [4]).

We highlight that different performances can be achieved using different solvers. We also highlight that each simulation could take some minutes to be completely executed. If you want take less time for the execution without numerical errors, we recommend to install CPLEX optimizer [22] that is free available for academic use. We note that most of the functions can be used in a parallel fashion, then the interested user can run `MatLabpool` before the simulations. For our simulations results, we refer the interested reader to [3].
One can also execute all files for modeling and designed PnPMPC controllers running

```
makeAllPNSfiles( type )
```

and can delete all files for PnPMPC controllers running `clearAllPNSfiles`.

For each control experiment we provide a file `.mat` of the simulation that contains

- lss object of the continuos linear system (`pnsCn`, where n is the number of the scenario);
- parameters of the control experiment `Tsim` and `▵PL`, where `▵PL` corresponds to ΔP_L;
- the results of the control experiment `x`, `▵Pref`, `eta` and `Phi`, where `▵Pref` corresponds to ΔP_{ref}.

For each Scenario we included also a Simulink model. In particular, one can load the file *.mat* of a control experiment and simulate the power network system given the power load steps and the power reference computed through centralized MPC or PnPMPC controllers.

6.4.1 Example of simulation

In the following we illustrate how to use the files *.mat* and the Simulink models for designing centralized and pnpmpc controllers.

- **Step 1** We can simulate different scenarios using the Simulink models present in the folder of each scenario. For Scenario 2 we then open the file *simulatorPNS_AGC_2.mdl*. This step is performed with the MatLab command:

```
open('simulatorPNS_AGC_2')
```

- **Step 2**

  Centralized case In the subfolders of each scenario there are MatLab files for centralized simulations as *scenario[number scenario][D—Dss][Diag—Full—Zero]Data*. Assume we want to simulate Scenario 2 using the discretization Dss and centralized MPC with zero terminal constraint ($MPCzero$). We need to load MatLab file *scenario2DssZeroData*. This operation can be performed with the MatLab command:

```
load scenario2DssZeroData
```

  PnPMPC case In the subfolders of each scenario there are MatLab files for PnPMPC simulations as *scenario[number scenario][Full—Zero]_sim[type simulation]PNS*. Assume we want to simulate Scenario 2, where decentralized controllers have been designed using nonlinear design, with zero terminal constraint and input constraints. We need to load MatLab file `scenario2ZeroData_sim1PNS`. This operation can be performed with the MatLab command:

```
load scenario2ZeroData_sim1PNS
```

- **Step 3** Start a simulation from Simulink will produce the results of the control experiments.

```
sim('simulatorPNS_AGC_2')
```

Simulation results are summarized in [3], [1], [25] and Chapters 5, 6 and 7 in [4].

Chapter 7

The pnpctrl class

7.1 Introduction

In this chapter, we introduce the pnpctrl class in order to design an unconstrained PnP controller for a single subsystem. Unconstrained PnP controllers are described in [19] and Chapter 4 in [4].

In those papers, the authors propose a PnP design procedure hinging on the notion of small-gain theorem for networks [26] for handling coupling among subsystems, and aim at stabilizing the origin of the whole closed-loop system. The model of subsystem i, $i \in \mathcal{M}$ is given by (2.3) and (2.4). We consider that subsystems are input decoupled. Our control design procedure will hinge on the following assumption.

Assumption 5. *The matrix pairs* (A_{ii}, B_i) $\forall i \in \mathcal{M}$ *are controllable.*

We design the following controller for subsystem i

$$\mathcal{C}_{[i]}: \quad u_{[i]} = v_{[i]} + K_i x_{[i]} + \sum_{j \in \mathcal{N}_i} \delta_{ij} K_{ij} x_{[j]} \tag{7.1}$$

where $K_{ij} \in \mathbb{R}^{m_i \times n_j}$ and $\delta_{ij} \in \{0, 1\}$. Note that if $\delta_{ij} = 0$, $\forall i \in \mathcal{M}$, $\forall j \in \mathcal{N}_i$, the control scheme is completely decentralized, since $u_{[i]}$ depends on local quantities only.

The key condition enabling PnP design is the following

$$\alpha_i = \sum_{j \in \mathcal{N}_i} \sum_{k=0}^{\infty} ||(A_{ii} + B_i K_i)^k (A_{ij} + \delta_{ij} B_i K_{ij})||_\infty < 1$$

Therefore, in order to design an unconstrained PnP controller for subsystem i we need to solve the following problem.

Problem $\mathcal{P}_i$: Given δ_{ij}, compute matrices K_{ii} and K_{ij}, $j \in \mathcal{N}_i$, such that $\alpha_i < 1$.

If Problem $\mathcal{P}_i$ cannot be solved, we declare that it is impossible to design an unconstrained PnP controller for subsystem i.

The interested reader is refer to [19] and Chapter 4 in [4].

7.1.1 The pnpctrl method

Assume `subss` stores the model of subsystem i given by (2.3) and (2.4). Controller $\mathcal{C}_{[i]}$ is created through

```
ctrl = pnpctrl(subss,Xj,flag)
```

The input argument `flag` is a structure with the following possible fields.

- `.distributedParents`: indices of subsystems such that $\delta_{ij} = 1$ in (7.1). It must be a subset of the indices of parent subsystems.
- `.norm`: specify norm for the computation of matrices K_{ij} (similar as in Chapter 7 in [4]).
- `.boundKij`: specify bound for the computation of matrices K_{ij} (similar as in Chapter 7 in [4]).
- `.Qi` and `.Ri`: `Qi` and `Ri` are inputs for the nonlinear optimization problem as proposed in Chapter 4 and 5 in [4].
- `.useConstraints`: if `false` the optimization problem does not take advantages of the knowledge of scaling factors for states of parent subsystems, `true` otherwise. Scaling factors are given in input using `Xj`, that is a cell matrix $\{H_{xj1}, K_{xj1}; H_{xj2}, K_{xj2}; \ldots\}$ such that parent states $x_{[j]}$, $j \in \mathcal{N}_i$ are in the set $Xj1 = \{xj1 | H_{xj1} xj1 \leq K_{xj1}\}$, $Xj2 = \{xj2 | H_{xj2} xj2 \leq K_{xj2}\}$, where $xj1$ and $xj2$ are two parent subsystems of the local subsystem `subss`.

`ctrl` is the PnPMPC controller designed with options given in `flag`.

7.1.2 The uk method

The control action $u_{[i]}$ is computed as in (7.1) executing the function

```
u = objCtrl.uRH(x0,xj)
```

- `x0` is the initial local state $x_{[i]}(0)$, a vector of dimension $n_i \times 1$.
- `xj` is the vector of states from parent subsystems such that $K_{ij} \neq 0$. Indices of these parent subsystems are in `objCtrl.Nij`.

As output we have:

- u control input value.

7.2 Design of unconstrained PnP controllers for a large-scale system

PnP controllers $\mathcal{C}_{[i]}$ for each subsystem in an lss object are created with the method `createCtrlPnP` described next.

7.2.1 The createCtrlPnP method

This method acts on an lss object, and it is called as follows:

```
[ ctrl , K ] = createCtrlPnP(objlss,flag)
```

where `flag` has the same meaning as in the pnpctrl method. This method extracts a subsystem, calls the constructor of class `pnpctrl` and design the controller. The output `ctrl` is an array of dimension $1 \times$`numSys` and in the i-th position the controller for subsystem i is stored. `K` is the overall matrix for the LSS, collecting matrices K_{ij}.

7.3 Examples

In this section we propose some examples. Section titles corresponds to m-files that can be found in the `./examples/pnpctrl` directory.

7.3.1 examplePnpCtrl.m

We consider a large-scale system composed of N masses coupled as in Figure 7.1.

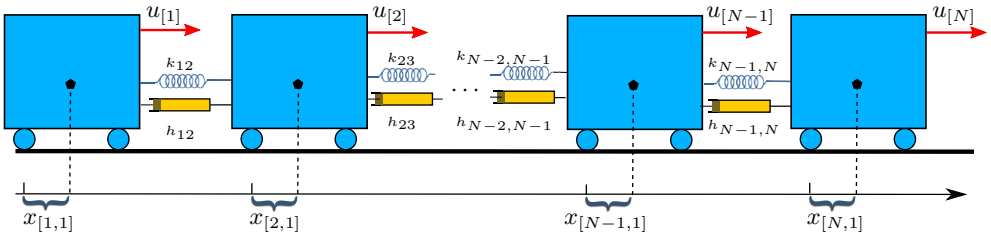

Figure 7.1: Array of masses: details of interconnections.

Each mass $i \in \mathcal{M} = 1 : M$, is a subsystem with input $u_{[i]}$ and state variables $x_{[i]} = (x_{[i,1]}, x_{[i,2]})$, where $x_{[i,1]}$ is the displacement with respect to a given equilibrium position, $x_{[i,2]}$ is the horizontal velocity and $100u_{[i]}$ is an external force in the horizontal direction. The values of $m_i = 10$, while spring and damping coefficients are identical and equal to 0.5. We obtain subsystems $\Sigma_{[i]}$ by discretizing continuous-time models with 0.2 sec sampling time, using zero-order-hold discretization for the local dynamics and treating $x_{[j]}$, $j \in \mathcal{N}_i$ as exogenous signals. We can successfully design PnP controllers. However, as the value of masses m_i, $i \in \mathcal{M}$ decreases, coupling terms increase, and, at the same point, it becomes impossible to design decentralized controllers $\mathcal{C}_{[i]}$. For example, if all masses $m_i = 0.01$ we cannot fulfill PnP conditions. However, by setting $\delta_{ij} = 1$, $j \in \mathcal{N}_i$ we can completely remove the coupling terms and therefore the synthesis of controllers $\mathcal{C}_{[i]}$ amounts to the synthesis of a state-feedback controller for each mass without coupling. This means that, for the system in Figure 7.1, PnP design of distributed controllers $\mathcal{C}_{[i]}$ is always possible.

Executing file `examplePnpCtrl`, all controllers will be designed.

Chapter 8

The pnpEstimator class

8.1 Introduction

In this chapter, we introduce the pnpEstimator class in order to design a PnP state estimator for a single subsystem. PnP state estimators are described in [6] and Chapter 8 in [4].

In those papers, the authors propose a distributed for (2.5) and (2.6). We assume that local outputs depend only on local states and disturbances. We define for $i \in \mathcal{M}$ the Local State Estimator (LSE) as

$$\tilde{\Sigma}_{[i]}: \quad \tilde{x}^{+}_{[i]} = A_{ii}\tilde{x}_{[i]} + B_{ii}u_{[i]} - L_{ii}(y_{[i]} - C_i\tilde{x}_{[i]}) + \sum_{j\in\mathcal{N}_i} A_{ij}\tilde{x}_{[j]} - \sum_{j\in\mathcal{N}_i} \tilde{\delta}_{ij}L_{ij}(y_{[j]} - C_j\tilde{x}_{[j]}) + \\ + \sum_{j\in\mathcal{N}_i} B_{ij}u_{[j]} + B_{cen}u_{cen} + \tilde{M}_i\tilde{d} \tag{8.1}$$

where $\tilde{x}_{[i]} \in \mathbb{R}^{n_i}$ is the state estimate, $L_{ij} \in \mathbb{R}^{n_i \times p_j}$ are gain matrices and $\tilde{\delta}_{ij} \in \{0, 1\}$. This implies that $\tilde{\Sigma}_{[i]}$ depends only on local variables ($\tilde{x}_{[i]}$, $u_{[i]}$ and $y_{[i]}$) and parents' variables ($\tilde{x}_{[j]}$, $u_{[j]}$, $y_{[j]}$, $j \in \mathcal{N}_i$), centralized inputs u_{cen} and exogenous inputs $\tilde{d}$. Binary parameters $\tilde{\delta}_{ij}$, $j \in \mathcal{N}_i$ can be chosen equal to one for exploiting the knowledge of parents' outputs, or equal to zero for reducing the number of transmitted output samples.

The key condition enabling PnP design is the following

$$\alpha_i = \sum_{j\in\mathcal{N}_i} \sum_{k=0}^{\infty} ||(A_{ii} + L_{ii}C_i)^k (A_{ij} + \tilde{\delta}_{ij}L_{ij}C_j)||_\infty < 1$$

Therefore, in order to design a PnP LSE for subsystem i we need to solve the following problem.

Problem $\mathcal{P}_i$: Given $\tilde{\delta}_{ij}$, compute matrices L_{ii} and L_{ij}, $j \in \mathcal{N}_i$, such that $\alpha_i < 1$.

If Problem $\mathcal{P}_i$ cannot be solved, we declare that it is impossible to design a PnP LSE for subsystem i.

The interested reader is refer to [6] and Chapter 8 in [4].

8.1.1 Bounded-error state estimation

In several applications, we need to guarantee that the state estimation error is bounded, i.e. $x_{[i]} - \tilde{x}_{[i]} \in \mathbb{E}_i$. Methods proposed in [6] and Chapter 8 in [4] allow one to guarantee prescribed bounds.

In the following, we introduce methods needed to equip an `lss` object with state estimation error constraints.

Methods for adding state estimation error constraints

The `addErrorEstimationConstraint` method sets the matrices H_{e_i} and K_{e_i}, such that $\mathbb{E}_i = \{x_{[i]} - \tilde{x}_{[i]} \in \mathbb{R}^{n_i} : H_{e_i}(x_{[i]} - \tilde{x}_{[i]}) \leq K_{e_i}\}$. The method declaration is:

```
objlss = objlss.addStateConstraint(sysindex,H,K)
```

Where `sysindex` is the number or name of the subsystem to which constraints have to be added and `H` and `K` are the matrices H_{e_i} and K_{e_i}. It is also possible to create constraints between two or more subsystems, in this case `sysindex` is a vector of scalars, or a cell array of names of the subsystems involved. In a similar way, as in Section 3.2.4, method

```
[He,Ke] = objlss.doubleErrorEstimationConstraint(index,selection)
```

returns matrices associated to sets $\mathbb{E}_i$.

8.1.2 The pnpEstimator method

Assume `subss` stores the model of subsystem i given by (2.5) and (2.6). State estimator $\tilde{\Sigma}_{[i]}$ is created through

```
estimator = pnpEstimator(subss,Ej,Cj,Oj,flag)
```

where

- `Ej` is a cell matrix $\{H_{ej1}, K_{ej1}; H_{ej2}, K_{ej2}; \ldots\}$ such that parent state estimation errors $e_{[j]}$, $j \in \mathcal{N}_i$ are in the set $Ej1 = \{ej1 | H_{ej1} ej1 \leq K_{ej1}\}$, $Ej2 = \{ej2 | H_{ej2} ej2 \leq K_{ej2}\}$, where $ej1$ and $ej2$ are state estimation errors of two parent subsystems of the local subsystem `subss`. These sets are needed if `flag.boundedEstimation=true`.

- `Cj` is a cell matrix $\{C_{j1}; C_{j2}; \ldots\}$, where each element is a the output linear transformation of parent subsystems. Matrix C_j is needed if $\tilde{\delta}_{ij} = 1$.

- `Oj` is a cell matrix $\{H_{oj1}, K_{oj1}; H_{oj2}, K_{oj2}; \ldots\}$ such that parent disturbances $d_{[j]}$, $j \in \mathcal{N}_i$ acting on outputs of parent subsystems are in the set $Oj1 = \{dj1 | H_{oj1} dj1 \leq K_{oj1}\}$, $Oj2 = \{dj2 | H_{oj2} dj2 \leq K_{oj2}\}$, where $dj1$ and $dj2$ are disturbances of two parent subsystems of the local subsystem `subss`. These sets are needed if `flag.boundedEstimation=true`.

The input argument `flag` is a structure with the following possible fields.

- `.distributedParents`: indices of subsystems such that $\tilde{\delta}_{ij} = 1$ in (8.1). It must be a subset of the indices of parent subsystems.

- `.norm`: specify norm for the computation of matrices L_{ij} (similar as in [6] and Chapter 8 in [4]).

- `.Qi` and `.Ri`: `Qi` and `Ri` are inputs for the nonlinear optimization problem as proposed in Chapter 8 in [4].

- `.boundedEstimation`: if `false` the optimization problem design a state estimator guaranteeing bounded-error state estimation, `true` otherwise.
- `.ExoBounded`: indices of exogenous inputs acting on subsystem `subss` that are disturbances, therefore, if `.boundedEstimation=true`, `pnpEstimator` design a robust state estimator as in [6] and Chapter 8 in [4].

`estimator` is the PnP state estimator designed with options given in `flag`.

8.1.3 The computeEstimation method

The state estimation $\tilde{x}_{[i]}$ is computed as in (8.1) executing the function

[xest, yest, yfilt] = estimator.computeEstimation(xt , yt , ut , xtj , ytj , utj , utcen , d)

- `xt` is $\tilde{x}_{[i]}$ at time t.
- `yt` is $y_{[i]}$ at time t.
- `ut` is $u_{[i]}$ at time t.
- `xtj` is a vector of $x_{[j]}$, $j \in \mathcal{N}_i$ at time t.
- `ytj` is a vector of $y_{[j]}$, $j \in \mathcal{N}_i$ at time t, needed if $\tilde{\delta}_{ij} = 1$.
- `utj` is a vector of $u_{[j]}$, $j \in \mathcal{N}_i$ at time t, needed if subsystems i and j are input decoupled.
- `ucen` is u_{cen} at time t.
- `d` is $\tilde{d}$ at time t, i.e. exogenous inputs that does not act on the subsystem as not measurable disturbances.

As output we have:

- `xest` is $\tilde{x}^+_{[i]}$ at time $t+1$.
- `yest` is $y_{[i]}$ estimated at time t.
- `yfilt` is $y_{[i]}$ estimated at time $t+1$.

8.1.4 The checkErrorEstimation method

If `flag.boundedEstimation=true`, this method checks if a given initial error $e_{[i]}(0) \in \mathbb{E}_i$, and hence if the state estimator $\tilde{\Sigma}_{[i]}$ can guarantee convergence of the state estimation and bounded-error at all time instants.

```
test = estimator.checkErrorEstimation(e)
```

8.1.5 The pnpEstimator2ss method

Given a PnP state estimator, returns a state space object.

```
sys = estimator.pnpEstimator2ss
```

`sys` is an `ss` object.

8.2 Design of PnP state estimators for a large-scale system

PnP state estimators $\tilde{\Sigma}_{[i]}$ for each subsystem in an lss object are created with the method `createPnPEstimators` described next.

8.2.1 The createPnPEstimators method

This method acts on an lss object, and it is called as follows:

```
[ estimators , L ] = createPnPEstimators(objlss,flag)
```

where `flag` has the same meaning as in the pnpEstimator method. This method extracts a subsystem, calls the constructor of class `pnpEstimator` and design the state estimator. The output `estimators` is an array of dimension $1 \times$`numSys` and in the i-th position the state estimator for subsystem i is stored. `L` is the overall matrix for the LSS, collecting matrices L_{ij}.

8.3 Examples

In this section we propose some examples. Section titles corresponds to m-files that can be found in the `./examples/pnpEstimator` directory.

8.3.1 example1pnpestimator.m

Executing file `example1pnpestimator`, the example proposed in [6] and Chapter 8 in [4] is generated, designing the LSS, the state estimators and simulating the convergence of the state estimation. Note that, since the LSS as several states for each subsystem, the generation of state estimators could take minutes.

Chapter 9

The lse class

9.1 Introduction

In this chapter, we introduce the lse class in order to design a large-scale state estimator for an LSS. PnP state estimators are described in [5] and Chapter 3 in [4].

In those papers, the authors propose a distributed for (2.5) and (2.6). We assume that local outputs depend only on local states and disturbances. Moreover the subsystems are input decoupled. We define for $i \in \mathcal{M}$ the Local State Estimator (LSE) as

$$\tilde{\Sigma}_{[i]}: \quad \tilde{x}_{[i]}^{+} = A_{ii}\tilde{x}_{[i]} + B_{ii}u_{[i]} - L_{ii}(y_{[i]} - C_i\tilde{x}_{[i]}) + \sum_{j\in\mathcal{N}_i} A_{ij}\tilde{x}_{[j]} - \sum_{j\in\mathcal{N}_i} \tilde{\delta}_{ij}L_{ij}(y_{[j]} - C_j\tilde{x}_{[j]}) + \\ + \sum_{j\in\mathcal{N}_i} B_{ij}u_{[j]} + B_{cen}u_{cen} + \tilde{M}_i\tilde{d} \tag{9.1}$$

where $\tilde{x}_{[i]} \in \mathbb{R}^{n_i}$ is the state estimate, $L_{ij} \in \mathbb{R}^{n_i \times p_j}$ are gain matrices and $\tilde{\delta}_{ij} \in \{0, 1\}$. This implies that $\tilde{\Sigma}_{[i]}$ depends only on local variables ($\tilde{x}_{[i]}$, $u_{[i]}$ and $y_{[i]}$) and parents' variables ($\tilde{x}_{[j]}$, $u_{[j]}$, $y_{[j]}$, $j \in \mathcal{N}_i$), centralized inputs u_{cen} and exogenous inputs $\tilde{d}$. Binary parameters $\tilde{\delta}_{ij}$, $j \in \mathcal{N}_i$ can be chosen equal to one for exploiting the knowledge of parents' outputs, or equal to zero for reducing the number of transmitted output samples.

The `lse` class design state estimators $\tilde{\Sigma}_{[i]}$, $i \in \mathcal{M}$, guaranteeing converge of state estimation and bounded-error at all time instants. The complete algorithm design can be found in Chapter 3 in [4].

9.1.1 The lse method

Assume `subss` stores the model of subsystem i given by (2.5) and (2.6). The collection of state estimators $\tilde{\Sigma}_{[i]}$ is created through

```
estimator = lse(objlss,flag,options)
```

where

- `objlss` is the large-scale system.
- `options` is the `sdpsettings` structure for `solvesdp` options in YALMIP.

- The input argument `flag` is a structure with the following possible fields.
  - `.Delta`: square matrix of dimension `objlss.numSys`. Element ij is `false` if subsystem j does not send outputs to subsystem i, i.e. $L_{ij} = 0$, `true` otherwise (default `true`).
  - `.norm`: specify norm for the computation of matrices L_{ij} (similar as in [5] and Chapter 3 in [4]).
  - `.ExoBounded`: indices of exogenous inputs acting on LSS `objlss` that are disturbances.

`estimator` is the large-scale state estimator designed with options given in `flag`.

9.1.2 The localEstimator method

The state estimation $\tilde{x}_{[i]}$ is computed as in (9.1) executing the function

```
[ xit, yit , yitf ] = localEstimator(obj,i,xit,xjt,yi,ui,yj,d,ucen)
```

- `i` is the index of the i-th subsystem.
- `xit` is $\tilde{x}_{[i]}$ at time t.
- `yi` is $y_{[i]}$ at time t.
- `ui` is $u_{[i]}$ at time t.
- `xjt` is a vector of $x_{[j]}$, $j \in \mathcal{N}_i$ at time t.
- `yj` is a vector of $y_{[j]}$, $j \in \mathcal{N}_i$ at time t, needed if $\tilde{\delta}_{ij} = 1$.
- `ucen` is u_{cen} at time t.
- `d` is $\tilde{d}$ at time t, i.e. exogenous inputs that does not act on the subsystem as not measurable disturbances.

As output we have:

- `xit` is $\tilde{x}^+_{[i]}$ at time $t+1$.
- `yit` is $y_{[i]}$ estimated at time t.
- `yitf` is $y_{[i]}$ estimated at time $t+1$.

9.1.3 The checkErrorEstimation method

This method checks if a given initial error $e_{[i]}(0) \in \mathbb{E}_i$, and hence if the state estimator $\tilde{\Sigma}_{[i]}$ can guarantee convergence of the state estimation and bounded-error at all time instants.

```
test = estimator.checkErrorEstimation(e,i,options)
```

9.1.4 The lse2ss method

Given a large-scale state estimator, returns a state space object.

```
sys = estimator.lse2ss( i )
```

`sys` is an `ss` object.

Note that several methods are available for different features of large-scale state estimators. All methods can be found in the `./@lse` directory and their utility is described in the help of each function.

9.2 Examples

In this section we propose some examples. Section titles corresponds to m-files that can be found in the `./examples/lse` directory.

9.2.1 example1lse.m

Executing file `example1lse`, an example is generated, designing the LSS, the large-scale state estimator and simulating the convergence of the state estimation.

Chapter 10

Invariant sets

In this chapter, we present methods to compute invariant sets. For more details we refer to the help of each function. First of all, we give some definitions.

Definition 8 (Robust Positively Invariant (RPI) set)**.** *The set* $\mathbb{X} \subseteq \mathbb{R}^n$ *is RPI for* $x(t+1) = f(x(t), w(t))$, $w(t) \in \mathbb{W} \subseteq \mathbb{R}^m$ *if* $x(t) \in \mathbb{X} \Rightarrow f(x(t), w(t)) \in \mathbb{X}$, $\forall w(t) \in \mathbb{W}$.

Definition 9 (Minimal Robust Positively Invariant (mRPI) set)**.** *The RPI set* $\underline{\mathbb{X}}$ *is minimal if every other RPI* $\mathbb{X}$ *verifies* $\underline{\mathbb{X}} \subseteq \mathbb{X}$. *The RPI set* $\mathbb{X}(\delta)$ *is a* δ*-outer approximation of the minimal RPI* $\underline{\mathbb{X}}$ *if*

$$x \in \mathbb{X}(\delta) \Rightarrow \exists\ \underline{x} \in \underline{\mathbb{X}} \text{ and } \tilde{x} \in B_\delta(0) : x = \underline{x} + \tilde{x}$$

where, for $\delta > 0$, $B_\delta(v) = \{x \in \mathbb{R}^n | \|x - v\| < \delta\}$.

Definition 10 (λ-contractive control invariant set)**.** *The set* $\mathbb{X} \subseteq \mathbb{R}^n$ *is a* λ*-contractive set for* $x(t+1) = f(x(t), u(t))$, $u(t) \in \mathbb{U} \subseteq \mathbb{R}^m$, *if* $\forall x(t) \in \mathbb{X}$ *there exist* $u(t) \in \mathbb{U}$ *such that* $x(t+1) \in \lambda\mathbb{X}$.

Definition 11 (Robust Control Invariant (RCI) set)**.** *The set* $\mathbb{X} \subseteq \mathbb{R}^n$ *is an RCI set for* $x(t+1) = f(x(t), u(t), w(t))$, $u(t) \in \mathbb{U} \subseteq \mathbb{R}^m$ *and* $w(t) \in \mathbb{W} \subseteq \mathbb{R}^p$, *if* $\forall x(t) \in \mathbb{X}$ *there exist* $u(t) \in \mathbb{U}$ *such that* $x(t+1) \in \mathbb{X}$, $\forall w(t) \in \mathbb{W}$.

10.1 ϵ-mRPI

In this section, we explain how to use the `epsilon_mRPI` class in order to compute an ϵ outer approximation of the mRPI set for a linear constrained systems.

Our implementation is based on the algorithm proposed in [27]. We propose two examples.

10.1.1 example1epsilon_mRPI.m

The code below shows how to compute an ϵ-mRPI set for the LTI system $x^+ = Ax + Bu + w$ where $x \in \mathbb{R}^2$, $u = Kx$ and w lies in the polytope $\mathbb{W}$. Non constraint on the state is assumed and hence $\mathbb{X} = \mathbb{R}^2$.

```
% matrices of the LTI system
A  = [ 1 1 ; 0 1 ];
```

```
B  = [ 1 ; 1 ];
K  = -[1.17 1.03];

% bounded disturbances as Polyhedron object
W = Polyhedron([ eye(2) ; -eye(2) ],[ 1 1 1 1 ]');

% approximation
epsilon = 5*10^-5;

% create an object F that is a epsilon-mRPI for the closed loop LTI system
% with dynamic A+BK and disturbances bounded in W (W is a Polyhedron object)
F  = epsilon_mRPI(A+B*K,W,epsilon)

% Otherwise one can create an object F that is a epsilon-mRPI for the
% closed loop LTI system with dynamic A+BK and disturbances bounded in W
% (W is a zonotope object)
% W = zonotope([-0.5 0.1]',eye(2));
% F = epsilon_mRPI(A+B*K,W,epsilon);

% plot the approximation of the mRPI
F.plot
```

Other useful methods fot the epsilon_mRPI class are

isinside Test if a point is inside the ϵ-mRPI set `F`.

```
x = [ 1;1 ];
test = F.isinside(x)
```

doubleHK If W is a Polyhedron object, the function returns the H-representation of the mRPI. Moreover a Polyhedron object of the mRPI is saved in `F_epsilon`.

```
[ H K F ] = F.doubleHK
FP        = F.F_epsilon
```

doublePG If W is a zonotope object, the function returns the G-representation of the mRPI. Moreover a zonotope object of the mRPI is saved in `F_epsilonZ`.

```
[ p G F ] = F.doublePG
FZ        = F.F_epsilonZ
```

10.1.2 example2epsilon_mRPI.m

This example shows how to compute an ϵ-mRPI set in the case of constraints on the state variables, i.e. $x \in \mathbb{X}$.

```
X = Polyhedron([ eye(2) ; -eye(2) ],[ 2 2 2 2 ]');
F = epsilon_mRPI(A+B*K,W,epsilon,X)
```

10.2 λ-contractive control invariant sets

In this section, we explain how to use the `localControlLyapunov` class in order to compute a λ-contractive control invariant set for a discrete-time LTI system. Our implementation is based on the algorithm proposed in [20].

We illustrate the use of the class localControlLyapunov in the following example.

10.2.1 example1localControlLyapunov.m

```
% matrices of the LTI system
A  = [ 1 1 ; 0 1 ];
B  = [ 0 ; 1 ];

% matrices for constraints on state and input variables
cx = [ eye(2) ; -eye(2) ];
dx = [ 2 2 1.5 2 ]';
cu = [ 1 ; -1 ];
du = [ 3 ; 3 ];

% parameter of the algorithm, ≥ controllability index
k  = 3;

% x0 is a parameter of the algorithm
% As x0 we consider the vertices of polytope = {x|cx*x≤dx}
x0 = [2 2;-1.5 2;-1.5 -2;2 -2];

% create the lambda contractive control invariant set
L  = localControlLyapunov(A,B,k,{cx,dx},{cu,du},x0)

% plot the lambda contractive control invariant set
L.plot

% Contractiveness
lambda = L.lambda
```

Other useful methods for the localControlLyapunov class are

isinside Test if a point is inside the λ-contractive control invariant set `L`.

```
x = [1;1]
test = L.isinside(x)
```

uInv Given $x \in \mathbb{X}$, compute the control action $u(x)$ such that $x(t+1) \in \lambda\mathbb{X}$.

```
x   = zeros(2,1)
u   = L.uInv(x)
```

double The function returns the H-representation of a λ-contractive control invariant set. Moreover a Polyhedron object of the set can be obtained through the attribute `LP`.

```
[ H K L ] = L.double
LP        = L.LP
```

10.3 Parameterized robust control invariant sets

In this section, we explain how to use the `parameterizedRCI` class in order to compute a parameterized robust control invariant set for a linear constrained systems. Our implementation is based on the algorithm proposed in [20].

We illustrate the use of the class parameterizedRCI in the following example.

10.3.1 example1parameterizedRCI.m

```
% matrices of the LTI system
A  = [ 1 1 ; 0 1 ];
B  = [ 0 ; 1 ];

% matrices for constraints on state and input variables
cx = [ eye(2) ; -eye(2) ];
dx = [ 2 2 1.5 2 ]';
cu = [ 1 ; -1 ];
du = [ 3 ; 3 ];

% parameter of the algorithm, greater than the controllability index of (A,B)
k  = 5;

% x0 is a parameter of the algorithm such that W \subseteq hull(x0)
x0 = [ 0.99 0.99 ; 0.99 -0.01 ; -0.01 -0.01 ; -0.01 0.99 ];

% create the parameterized robust control invariant set
R  = parameterizedRCI(A,B,k,{cx,dx},{cu,du},x0)

% plot the parameterized robust control invariant set
R.plot
```

Other useful methods for the parameterizedRCI class are

isinside Test if a point is inside the parameterized robust control invariant set `R`.

```
x = [1;1]
test = R.isinside(x)
```

uInv Given $x \in \mathbb{X}$, compute the control action $u(x)$ such that $x(t+1) \in \mathbb{X}$, $\forall w \in \mathbb{W}$.

```
x  = zeros(2,1)
u  = R.uInv(x)
```

double The function returns the H-representation of the parameterized robust control invariant set. Moreover a Polyhedron object of the set can be obtained through the attribute `RP`.

```
[ H K R ] = R.double
RP        = R.RP
```

10.4 Zonotopic robust control invariant sets

In this section, we explain how to use the `zonotopeRCI` class in order to compute a zonotopic robust control invariant set for a linear constrained systems. Our implementation is based on the algorithm proposed in [28].

We illustrate the use of the class zonotopeRCI in the following example.

10.4.1 example1zonotopeRCI.m

```
% matrices of the LTI system
A  = [ 1 1 ; 0 1 ];
B  = [ 0 ; 1 ];

% matrices for constraints on state and input variables
% X = { x | cx*x≤dx }
% U = { u | cu*u≤du }
cx = [ 1 0 ; -1 0 ; 0 -1 ; 0 1 ; -1 -1 ];
dx = [ 1.85 3 3 3 2.2 ]';
cu = [ 1 ; -1 ];
du = [ 3 ; 3 ];

% parameter of the algorithm, greater than the number of states
k  = 5;

% matrices of the zonotope for the description of set W = { w=f+Ed ,|d|inf≤1 }
E  = [ 1 0 ; 0 1 ];
```

```
f  = [ 0   ; 0 ];

% create the zonotopic robust control invariant set
qa = 1 ; qb = 1 ; qp = 1 ;
Z  = zonotopeRCI(A,B,{cx,dx},{cu,du},{f,E},k,[qa,qb,qp]);

% plot the zonotopic robust control invariant set
Z.plot
```

Other useful methods for the zonotopeRCI class are

isinside Test if a point is inside the zonotope robust control invariant set `Z`.

```
x = [1;1]
test = Z.isinside(x)
```

uInv Given $x \in \mathbb{X}$, compute the control action $u(x)$ such that $x(t+1) \in \mathbb{X}$, $\forall w \in \mathbb{W}$.

```
x  = zeros(2,1)
u  = Z.uInv(x)
```

Chapter 11

PnPMPC and others toolboxes

11.1 Integration with the toolbox WIDE [29]

11.1.1 The lsmodel2lss function

In WIDE a large-scale system is described by an `LSmodel` object. We use the function `lsmodel2lss` to convert an `LSmodel` object to an `lss` object. The function declaration is

```
objlss = lsmodel2lss( objlsmodel , m , p )
```

The arrays `m` and `p` are used to select the external inputs and external outputs in the `LSmodel` object. The k-th element of the array `m` is an integer number to classify the k-th external input of the `LSmodel` object: if `m(k)=0` the k-th external input is a centralized input for the `lss` object, if `m(k)=-1` the k-th external input is an exogenous input for the `lss` object and if `m(k)=i` the k-th external input is a local input for the i-th subsystem of the `lss` object. Similarly, the k-th element of the array `p` is an integer number to classify the k-th external output of the `LSmodel` object: if `p(k)=i` the k-th external input is a local input for the i-th subsystem of the `lss` object.

11.1.2 The createCtrlPnPMPC4lsmodel function

Given an `LSmodel` object, we can design PnPMPC controllers using the `createCtrlPnPMPC4lsmodel` function. This function calls the `lsmodel2lss` function and the `createCtrlPnPMPC` method of the `lss` object of the function and returns an array of `pnpmpc` objects. The function declaration is

```
[ ctrl objlss ]=createCtrlPnPMPC4lsmodel(objlsmodel,N,k,m,p,options)
```

For the meaning of the input arguments we defer the reader to Section 11.1.1 and Section 5.2.1.

11.1.3 Example

We create an `LSmodel` object of the power network system proposed in Scenario 1 in Chapter 6. We can create the `LSmodel` object executing the file `makePNSLSmodel.m`. Executing the

file `makePnpmpcControllersPNS4LSmodel` we can design PnPMPC controllers for the power network described by an `LSmodel`. We show below the essential part of the code.

```
Q    = diag([500 0.01 0.01 10]);
R    = 10;
N    = 15;
kmin = [ 8 8 8 6 ];
kmax = [ 20 20 20 20 ];

% m and p array for lsmodel2lss
% the odd elements of m are the power references of each area, therefore
% they are the local inputs of each subsystem; the even elements of m are
% the power loads of each area, therefore they are the exogenous signals
% of each subsystem
m = [ 1 -1 2 -1 3 -1 4 -1 ];
% the elements of p are the outputs of each subsystem, therefore they are
% the states of each area
p = [ 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 ];

% convert LSmodel to lss
objlss = lsmodel2lss(pnsD1ssLSmodel,m,p);
% design pnp controllers
for i =1:numel(kmin)
    flag(i).k = { kmin(i) kmax(i) };
end
ctrlPnpMpcFromLSmodel =
objlss.createCtrlPnPMPC(N,flag,sdpsettings('verbose',0));
% one can also use the following instruction
% [ ctrlPnpMpcFromLSmodel objlss ]=createCtrlPnPMPC4lsmodel(pnsD1ssLSmodel,
% N , {kmin,kmax} , m , p , sdpsettings('verbose',0) )

% zero terminal constraint for each pnpmpc controller
ctrlPnpMpcZeroFromLSmodel = ctrlPnpMpcFromLSmodel;
for i=1:size(ctrlPnpMpcFromLSmodel,2)
    ctrlPnpMpcZeroFromLSmodel(i)=ctrlPnpMpcFromLSmodel(i).zeroTerminal(Q,R);
end
```

Chapter 12

Zonotope class

In PnPMPC-toolbox the `zonotope` class has been developed as an inherited class of the `Polyhedron` class of MPT3 [7]. In the following we describe how to generate zonotope sets. Since several functions have the same meaning of functions of `Polyhedron` class, we defer the reader to the MPT3 manual. These functions implement the standard operations between zonotope sets: Minkowski sum ($\oplus$), Pontryagin difference ($\ominus$), intersection ($\cap$), union ($\cup$) and relational operators ($\subset$, $\subseteq$, $\supset$, $\supseteq$, $=$ and $\neq$). Zonotope arrays are managed as Polyhedron arrays in MPT3.

In the following sections, we introduce some useful operators for zonotope sets. Most of the definition are from [30].

12.1 Creating a zonotope

A zonotope in PnPMPC-toolbox is created by a call to the zonotope constructor

```
Z = zonotope(p,G)
```

This instruction creates a zonotope `Z` centered in p and described by generators G. Example

```
p = [ 1 ; 1 ];
G = [ 1 2 4 ; 3 1 4 ];
Z = zonotope(p,G);
Z.plotZ
```

The zonotope is shown in Figure 12.1.

In oder to access the G-representation of zonotope `Z`, one can access to properties `p` and `G`

```
p = Z.p
G = Z.G
```

In oder to access the H-representation of zonotope `Z`, one can use the command `computeHRep`

```
Zo = Z.computeHRep
```

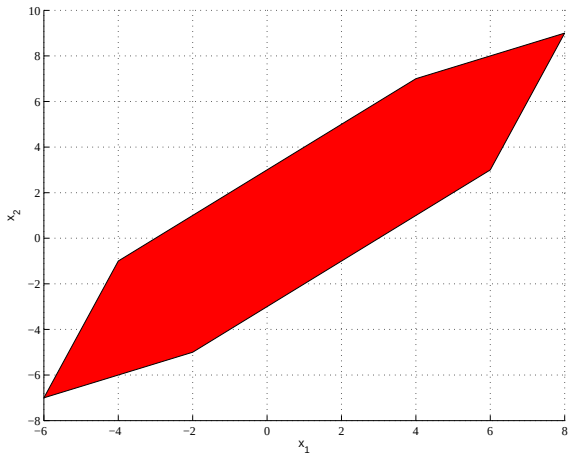

Figure 12.1: Zonotope $\mathbb{Z}$

```
H   = Zo.A
K   = Zo.b
```

Since the computation of the H-representation of zonotope `Z` requires the computation of the V-representation both representation are stored in the output variable `Zo` for future uses.

12.2 Functions for zonotope objects

12.2.1 Bisection and Split operators

The operator $Bisect_k(\cdot)$ generates two sub-zonotopes from one zonotope. In particular, given a zonotope $\mathbb{Z}$ the operator $Bisect_k(\mathbb{Z})$ generates the two sub-zonotopes

$$\mathbb{Z}^L = (p - \frac{g_k}{2}) \oplus [g_1 \dots \frac{g_k}{2} \dots g_m]d, \ ||d||_{inf} \leq 1$$

$$\mathbb{Z}^R = (p + \frac{g_k}{2}) \oplus [g_1 \dots \frac{g_k}{2} \dots g_m]d, \ ||d||_{inf} \leq 1$$

where g_k is the k-th column of G and is the biggest generator, i.e. we split the k-th column in the middle. With the operator $Split_k(\mathbb{Z}, \alpha)$ it is possible to split the k-th column of G in a desired position α, i.e., $Split_k(\mathbb{Z}, \alpha)$ generates two sub-zonotopes

$$\mathbb{Z}^L = (p - g_k(1-\alpha)) \oplus [g_1 \dots g_k\alpha \dots g_m]d, \ ||d||_{inf} \leq 1$$

$$\mathbb{Z}^R = (p + g_k\alpha) \oplus [g_1 \dots g_k(1-\alpha) \dots g_m]d, \ ||d||_{inf} \leq 1$$

where g_k is a column of G matrix and the parameter $\alpha \in [0, 1]$. Figure 12.2 shows an example of the operator $Bisect_k(\mathbb{Z})$ applied to the zonotope $\mathbb{Z}$ in Figure 12.1. This example shows that the operator $Bisect_k(\mathbb{Z})$ can generate two sub-zonotopes which intersect in their interior. The reason of the overlapping of $\mathbb{Z}^L$ and $\mathbb{Z}^R$ is that the line segment generators $g_1, \dots, g_m$ are not linearly independent. Given $G \in \mathbb{R}^{n \times m}$, with $Rank(G) = n$, then, the bisection is complete, i.e., $Bisect_k(\mathbb{Z})$ provides two sub-zonotopes that do not overlap. The operators $Bisect_k(\cdot)$ and $Split_k(\cdot)$ have been implemented in PnPMPC-toolbox in the following two instructions

```
Zn = Z.bisect
Zn = Z.split(gener,alpha)
```

The following code shows an example where zonotope Z is splitted.

```
p = [ 1 ; 1 ];
G = [ 1 2 4 ; 3 1 4 ];
alpha = 0.2;
k = 2;
Z = zonotope(p,G)
Znb = Z.bisect
Zns = Z.split(k,alpha)

figure(1)
Z.plotZ
figure(2)
Znb.plotZ(struct('shade',0.6))
figure(3)
Zns.plotZ(struct('shade',0.6))
```

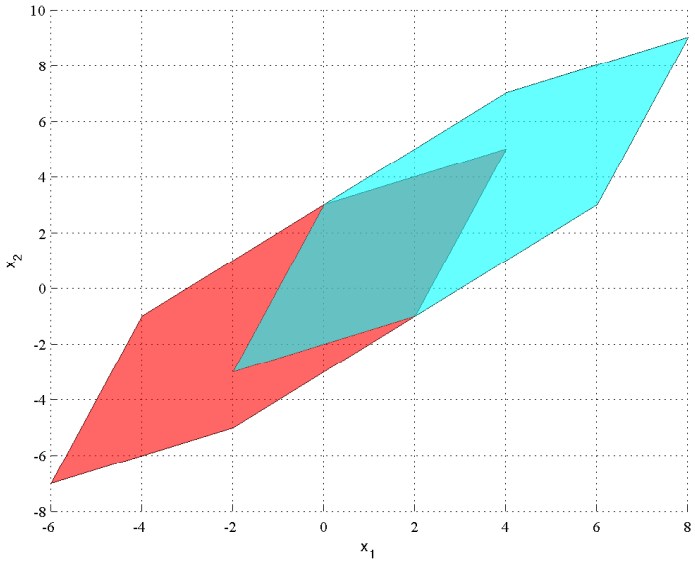

Figure 12.2: An example of split. $Bisect(\mathbb{Z})$.

12.2.2 Bounding box operator

The smallest interval vector containing $\mathbb{Z}$ and having its same center is given by the operator $rs(\cdot)$ (row sum). Given a matrix $G \in \mathbb{R}^{n\times m}$, $rs(G)$ is a $n \times n$ diagonal matrix

$$rs(G)_{ii} = \sum_{j=1}^{m} |G_{ij}|, \ i = 1, \ldots, n.$$

The operator $rs(\cdot)$ has been implemented in PnPMPC-toolbox in the following instruction

```
BB = Z.bounding_box
```

where `BB` is a zonotope object computed using operator $rs(\cdot)$. The following code shows an example where zonotope `Z` is bounded by the smallest interval vector.

```
p = [ 1 ; 1 ; -1 ];
G = [ 1 2 4 2.4 5 ; 3 1 4 5 6 ; 2 4 5 1 5];
Z = zonotope(p,G)
BB = Z.bounding_box
figure(1)
hold on
BB.plotZ
Z.plotZ('b')
```

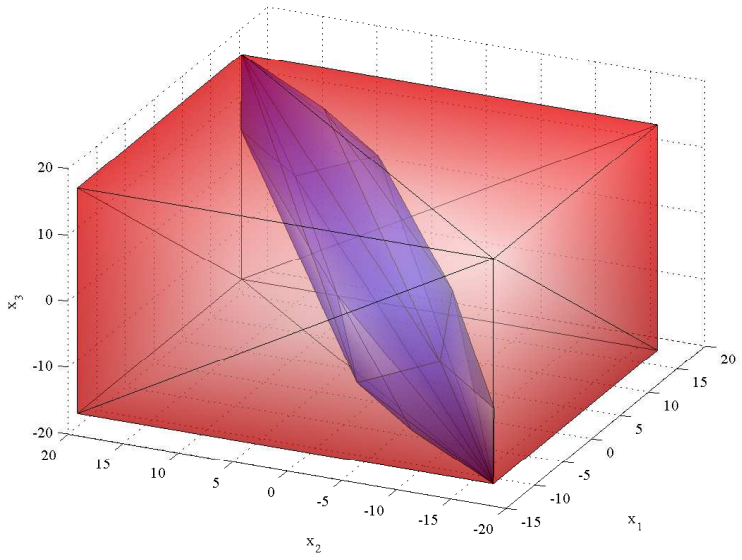

Figure 12.3: An example of bounding box of a set $\mathbb{Z}$.

12.2.3 Reduction operator

Another important operator is the reduction operator, whose purpose is to outer bound a given zonotope with a zonotope of reduced complexity, i.e., a reduced number of line segment generators. Given the zonotope $\mathbb{Z}$, the reduction operator $red_{ngen}(\mathbb{Z})$ produces a lower complexity zonotope generated by a maximum of $ngen$ line segment generators. The procedure consists of first sorting the columns of G with respect to decreasing Euclidean norm

$$G = [g_1, g_2, \ldots, g_m], \ ||g_i|| \geq ||g_{i+1}||, \ i = 1, \ldots, m-1.$$

Then, denoting by G_{ngen} the matrix describing $red_{ngen}(\mathbb{Z})$, we define

$$\begin{aligned} G_{gen} &= G, \text{ if } m \leq ngen \\ G_{ngen} &= [g_1, \ldots, g_{ngen-n}, g_r], \ G_r = rs([g_{ngen-n+1}, \ldots, g_m]), \text{ if } m > ngen. \end{aligned}$$

It is important to mention that $\mathbb{Z} \subseteq red_{ngen}(\mathbb{Z})$. Figure 12.4 shows the application of the reduction operator. As one can see, a reduction in the number of generators yields a more conservative zonotope. The operator $red_{ngen}(\cdot)$ has been implemented in PnPMPC-toolbox in the following instruction

```
Zo = Z.reduce(ngen)
```

The following code shows an example where the number of generators of zonotope `Z` is reduced.

```
p = [ 1 ; 1 ];
G = [ 1 2 4 2.4 5 ; 3 1 4 5 6 ];
Z = zonotope(p,G)
ngen = 3;
Zo = Z.reduce(ngen)
figure(1)
hold on
Zo.plotZ('r')
Z.plotZ('b')
```

12.2.4 The support function of polytope/zonotope sets

The support function of a polyhedral set is defined as

$$sup = \max_{x \in \mathbb{X}} c'x \tag{12.1}$$

One can compute sup using the following instruction

```
sup = Z.extreme(c)
```

where `X` is a `polytope` or `zonotope` object.

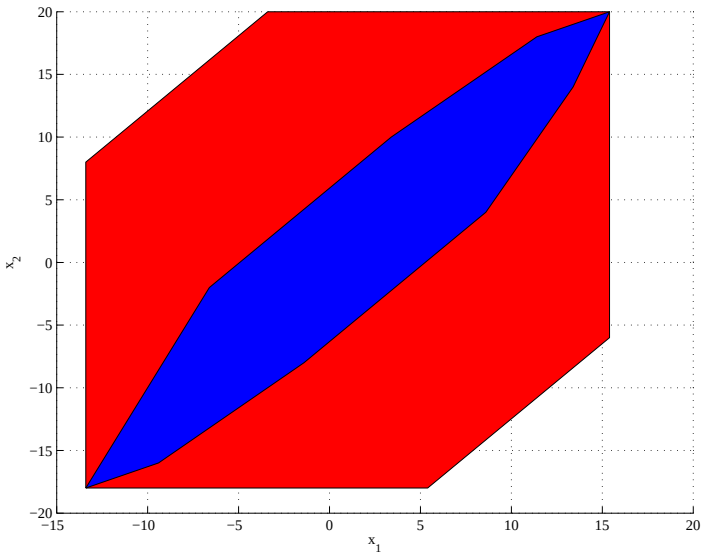

Figure 12.4: The reduction operator $red_{ngen}(\mathbb{Z})$ is applied to the green zonotope yielding a more conservative approximation (red zonotope).

12.2.5 Zonotope ↔ Polyhedron

- Polyhedron $\mathbb{P}$ is a zonotope?

```
bool = iszonotope(P)
```

- Polyhedron $\mathbb{P}$ to Zonotope $\mathbb{Z}$

```
Z = Polyhedron2zonotope(P)
```

- Zonotope $\mathbb{Z}$ to Polyhedron $\mathbb{P}$

```
P = zonotope2Polyhedron(Z)
```

12.3 Outer approximation of a nonlinear function

The function `outerApproximation`[1] allows one to compute an outer approximation of a nonlinear function evaluated on a zonotope $\mathbb{Z}$. We compute the outer approximation of the nonlinear function using the methods proposed in [31, 30]. In [31], an algorithm to compute zonotope outer approximations for nonlinear systems was proposed. The authors suggest to create an image of a zonotope through a nonlinear function using DC programming, which is based on DC functions.

[1] The function `outerApproximation` needs Symbolic Math Toolbox.

A DC function $f(x) : \mathbb{R}^n \to \mathbb{R}$ is a function that can be expressed as the difference of two convex functions, i.e. $f(x) = g(x) - h(x)$ where $g(x)$ and $h(x)$ are convex functions. In order to compute a tighter outer approximation, in [30] and [32] the authors proposed an algorithm in order to split the zonotope set $\mathbb{Z}$ where the function $f(x)$ is "more nonlinear" and then the outer approximation is obtained as the union of the outer approximations obtained using the DC programming on each zonotope of the splitting.

In the following, we propose three examples that can be found in the PnPMPC-toolbox.

12.3.1 example1outerApproximation.m

```
% definition of function f, set X and sampling of f(X)
npoints = 500;
X  = zonotope(zeros(2,1),[2 0;0 3]);
xf = zeros(dimension(X),npoints);
f  = @(x,w)([  1+0.1*x(1)+0.5*x(2)-exp(0.1*x(1)^2) ;
    0.1+0.9*x(1)-0.1*x(2)-0.1*cos(x(2))+0.05*x(2)^2 ]);
for i=1:size(xf,2)
    xx = randpoint(X);
    xf(:,i) = f(xx);
end

% options for outerApproximation function
options.split.ngenerators = 2;
options.split.alpha = 0.5;
options.split.max_zono = 5;

% compute zonotope approximation of f(X)
% Zu approximation using 1 zonotope
% ZuVec approximation using options.split.max_zono zonotopes
% Xs zonotope X splitted
% J jacobian and H hessian are outputs variables for future computations
% The third and fourth input argument are empty, that means the user does
% not know convex functions gf and hf, such that f = gf - hf. Therefore the
% function outerApproximation computes gf and hf. We highlight that the
% IntLab toolbox http://www.ti3.tuhh.de/rump/intlab/ is needed.
[ Zu ZuVec Xs J H ] = outerApproximation(X,f,[],[],[],options);

% plots
figh = figure(1);
subplot(1,2,2)
hold on
Zu.plotZ('y')
ZuVec.plotZ('r')
plot(xf(1,:),xf(2,:),'b.')
box on
subplot(1,2,1)
Xs.plotZ('g')
```

```
box on
```

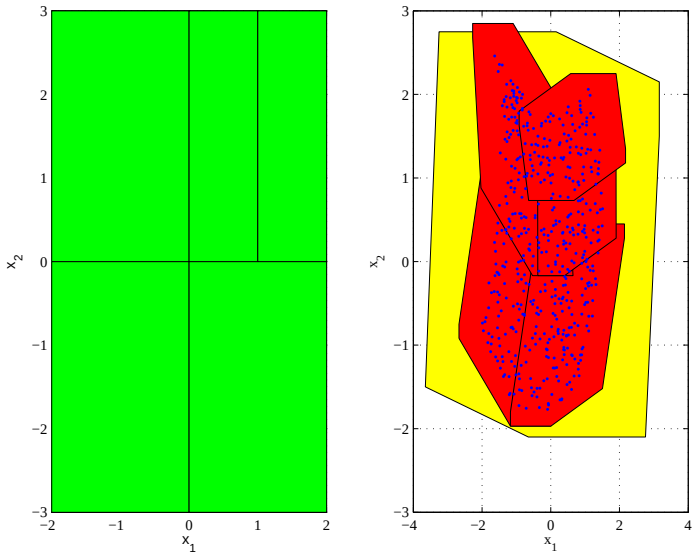

Figure 12.5: Results of example example1outerApproximation.m.

12.3.2 example2outerApproximation.m

```
% definition of function f, set X and sampling of f(X)
npoints = 500;
X  = zonotope(zeros(2,1),[2 0;0 3]);
xf = zeros(dimension(X),npoints);
f  = @(x,w)([  1+0.1*x(1)+0.5*x(2)-exp(0.1*x(1)^2)    ;
    0.1+0.9*x(1)-0.1*x(2)-0.1*cos(x(2))+0.05*x(2)^2     ]);
for i=1:size(xf,2)
    xx = randpoint(X);
    xf(:,i) = f(xx);
end

% definition of g and h such that f=g-h where g and h are convex functions
g  = @(x,w)([  0.5*x(2)    ;    0.1+0.9*x(1)-0.1*cos(x(2))+0.05*x(2)^2 ]);
h  = @(x,w)([  -1-0.1*x(1)+exp(0.1*x(1)^2)    ;    0.1*x(2)     ]);

% options for outerApproximation function
options.split.ngenerators = 2;
options.split.alpha = 0.5;
options.split.max_zono = 5;

% compute zonotope approximation of f(X)
```

```
% Zu approximation using 1 zonotope
% ZuVec approximation using options.split.max_zono zonotopes
% Xs zonotope X splitted
% J jacobian and H hessian are outputs variables for future computations
[ Zu ZuVec Xs J H ] = outerApproximation(X,f,[],g,h,options);

% plots
figh = figure(1);
subplot(1,2,2)
hold on
Zu.plotZ('y')
ZuVec.plotZ('r')
plot(xf(1,:),xf(2,:),'b.')
box on
subplot(1,2,1)
Xs.plot('g')
box on
```

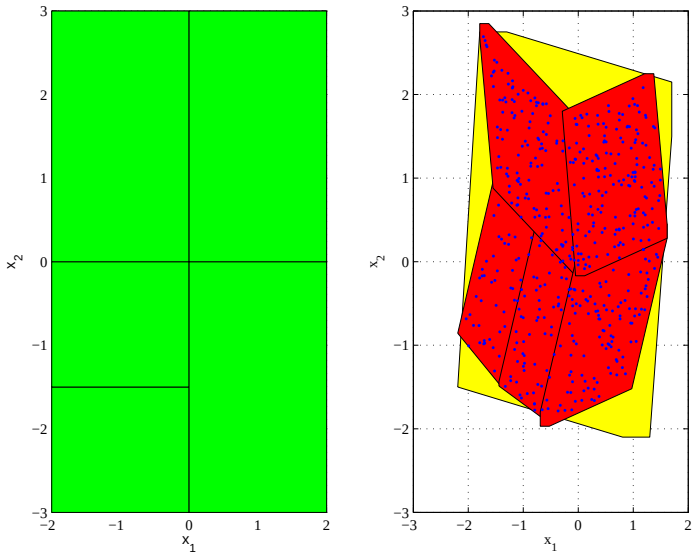

Figure 12.6: Results of example example2outerApproximation.m.

12.3.3 example3outerApproximation.m

```
% definition of function f, set X and sampling of f(X)
% differently from example2outerApproximation we want to split only along
% x(1), therefore x(2) is substituted by w(1)
npoints = 500;
X1 = zonotope(0,2);
```

```
W  = zonotope(0,3);
X  = X1*W;
xf = zeros(dimension(X),npoints);
f  = @(x,w)([  1+0.1*x(1)+0.5*w(1)-exp(0.1*x(1)^2)   ;
    0.1+0.9*x(1)-0.1*w(1)-0.1*cos(w(1))+0.05*w(1)^2     ]);
for i=1:size(xf,2)
    xx = randpoint(X);
    xf(:,i) = f(xx(1),xx(2));
end

% definition of g and h such that f=g-h where g and h are convex functions
g  = @(x,w)([  0.5*w(1)   ;    0.1+0.9*x(1)-0.1*cos(w(1))+0.05*w(1)^2   ]);
h  = @(x,w)([  -1-0.1*x(1)+exp(0.1*x(1)^2)   ;    0.1*w(1)     ]);

% options for outerApproximation function
options.split.ngenerators = 2;
options.split.alpha = 0.5;
options.split.max_zono = 5;

% compute zonotope approximation of f(X)
% Zu approximation using 1 zonotope
% ZuVec approximation using options.split.max_zono zonotopes
% Xs zonotope X splitted
% J jacobian and H hessian are outputs variables for future computations
[ Zu ZuVec Xs J H ] = outerApproximation(X1,f,W,g,h,options);

% plots
h=figure(1);
subplot(1,2,2)
hold on
Zu.plotZ('y')
ZuVec.plotZ('r')
plot(xf(1,:),xf(2,:),'b.')
box on
subplot(1,2,1)
Xs.plotZ('g')
box on
```

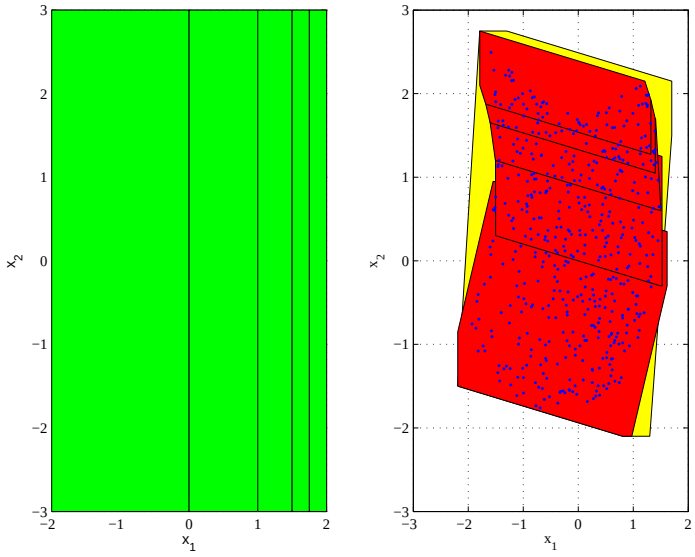

Figure 12.7: Results of example example3outerApproximation.m.

Bibliography

[1] S. Riverso, M. Farina, and G. Ferrari-Trecate, "Plug-and-Play Decentralized Model Predictive Control for Linear Systems," *IEEE Transactions on Automatic Control*, vol. 58, no. 10, pp. 2608–2614, 2013.

[2] ——, "Plug-and-Play Model Predictive Control based on robust control invariant sets," *Automatica*, p. Accepted, 2014.

[3] ——, "Plug-and-Play Model Predictive Control based on robust control invariant sets," Dipartimento di Ingegneria Industriale e dell'Informazione, Universita' degli Studi di Pavia, Pavia, Italy, Tech. Rep., 2012. [Online]. Available: arXiv:1210.6927

[4] S. Riverso, "Distributed and plug-and-play control for constrained systems," Ph.D. dissertation, Universita' degli studi di Pavia, 2014.

[5] S. Riverso, D. Rubini, and G. Ferrari-Trecate, "Distributed bounded-error state estimation for partitioned systems based on practical robust positive invariance," in *Proceedings of the 12th European Control Conference*, Zurich, Switzerland, July 17-19, 2013, pp. 2633–2638.

[6] S. Riverso, M. Farina, R. Scattolini, and G. Ferrari-Trecate, "Plug-and-play distributed state estimation for linear systems," in *Proceedings of the 52nd IEEE Conference on Decision and Control*, Florence, Italy, December 10-13, 2013, pp. 4889–4894.

[7] M. Herceg, M. Kvasnica, C. N. Jones, and M. Morari, "Multi-Parametric Toolbox 3.0," in *Proceedings of the 12th European Control Conference*, Zurich, Switzerland, July 17-19, Jul. 2013, pp. 502–510. [Online]. Available: http://control.ee.ethz.ch/~mpt

[8] J. Löfberg, "YALMIP: A toolbox for modeling and optimization in MATLAB," in *Proceedings of IEEE Symposium on Computer Aided Control Systems Design*, Taipei, Taiwan, September 2-4, 2004, pp. 284–289.

[9] M. Dunham, K. Murphy, L. Peshkin, and D. Eaton, "GraphViz4Matlab," 2004. [Online]. Available: https://github.com/graphviz4matlab/graphviz4matlab/

[10] M. Farina, P. Colaneri, and R. Scattolini, "Block-wise discretization accounting for structural constraints," *Automatica*, vol. 49, no. 11, pp. 3411–3417, 2013.

[11] M. Kvasnica, P. Grieder, and M. Baotić, "Multi-Parametric Toolbox (MPT)," 2004. [Online]. Available: http://control.ee.ethz.ch/~mpt/2/

[12] A. Bemporad and M. Morari, "Control of systems integrating logic, dynamics, and constraints," *Automatica*, vol. 35, no. 3, pp. 407–428, 1999.

[13] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert, "Constrained model predictive control: Stability and optimality," *Automatica*, vol. 36, no. 6, pp. 789–814, 2000.

[14] J. M. Maciejowski, *Predictive Control: with constraints*. Upper Saddle River, NJ, USA: Prentice Hall, 2002.

[15] J. B. Rawlings and D. Q. Mayne, *Model Predictive Control: Theory and Design*. Madison, WI, USA: Nob Hill Pub., 2009.

[16] D. Q. Mayne, M. M. Seron, and S. V. Raković, "Robust model predictive control of constrained linear systems with bounded disturbances," *Automatica*, vol. 41, no. 2, pp. 219–224, 2005.

[17] S. V. Raković and D. Q. Mayne, "A simple tube controller for efficient robust model predictive control of constrained linear discrete time systems subject to bounded disturbances," in *Proceedings of the 16th IFAC World Congress*, Prague, Czech Republic, July 4-8, 2005, pp. 241–246.

[18] S. Riverso, M. Farina, and G. Ferrari-Trecate, "Plug-and-Play decentralized Model Predictive Control," in *Proceedings of the 51st IEEE Conference on Decision and Control*, Maui, Hawaii, USA December 10-13, Dec. 2012, pp. 4193–4198.

[19] S. Riverso and G. Ferrari-Trecate, "Plug-and-Play distributed model predictive control with coupling attenuation," *Optimal Control Applications and Methods*, p. Submitted, 2013.

[20] S. V. Raković and M. Baric, "Parameterized Robust Control Invariant Sets for Linear Systems: Theoretical Advances and Computational Remarks," *IEEE Transactions on Automatic Control*, vol. 55, no. 7, pp. 1599–1614, 2010.

[21] S. Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan, *Linear matrix inequalities in system and control theory*. Philadelphia, Pennsylvania, USA: SIAM Studies in Applied Mathematics, vol. 15, 1994.

[22] IBM, "IBM ILOG CPLEX Optimization Studio 12.4," 2011.

[23] Hycon2, "Highly-complex and networked control systems (HYCON2 Network of excellence)," 2010. [Online]. Available: http://www.hycon2.eu

[24] H. Saadat, *Power System Analysis*, 2nd ed. New York. NY, USA: McGraw-Hill Series in Electrical and Computer Engineering, 2002.

[25] S. Riverso, M. Farina, and G. Ferrari-Trecate, "Design of plug-and-play model predictive control: an approach based on linear programming," in *Proceedings of the 52nd IEEE Conference on Decision and Control*, Florence, Italy, December 10-13, 2013, pp. 6530–6535.

[26] S. Dashkovskiy, B. S. Rüffer, and F. R. Wirth, "An ISS small gain theorem for general networks," *Mathematics of Control, Signals, and Systems*, vol. 19, no. 2, pp. 93–122, 2007.

[27] S. V. Raković, E. C. Kerrigan, K. I. Kouramas, and D. Q. Mayne, "Invariant approximations of the minimal robust positively invariant set," *IEEE Transactions on Automatic Control*, vol. 50, no. 3, pp. 406–410, 2005.

[28] S. V. Raković, "Robust Control of Constrained Discrete Time Systems: Characterization and Implementation," Ph.D. dissertation, Imperial College London, University of London, 2005.

[29] D. Barcelli, N. Bauer, and P. Trnka, "WIDE Toolbox," 2012. [Online]. Available: http://ist-wide.dii.unisi.it/index.php?p=toolboxsp

[30] D. M. Raimondo, S. Riverso, S. Summers, C. N. Jones, J. Lygeros, and M. Morari, "A set theoretic method for verifying feasibility of a fast explicit nonlinear Model Predictive Controller," in *Distributed Decision Making and Control*, R. Johansson and A. Rantzer, Eds. Springer, Lecture Notes in Control and Information Sciences vol. 417, 2012, ch. 13, pp. 289–311.

[31] T. Alamo, J. M. Bravo, M. J. Redondo, and E. F. Camacho, "A set-membership state estimation algorithm based on DC programming," *Automatica*, vol. 44, no. 1, pp. 216–224, 2008.

[32] D. M. Raimondo, S. Riverso, C. N. Jones, and M. Morari, "A Robust Explicit Nonlinear MPC Controller with Input-To-State Stability Guarantees," in *Proceedings of the 18th IFAC World Congress*, Milano, Italy, August 28 - September 2, Aug. 2011, pp. 9284–9289.