A Practical Guide to Distributed Model Predictive Control

Author:
Oliver Gäfvert, oliverg@kth.se

August 28, 2014

Abstract

This paper aims to give an introduction to the *Distributed Model Predictive Control Toolbox* for Matlab. An introduction to Model Predictive Control (MPC) and Distributed Model Predictive Control (DMPC) and a description of how a distributed system of MPC controllers is represented in the toolbox is given. Three solution methods to the distributed MPC problem that are implemented in the toolbox are introduced and those methods are Nesterov's Accelerated Gradient Method [5], a Multi-step Gradient Method [3] and the Alternating Direction Method of Multipliers (ADMM) [7]. Also an attempt to implement a Hydro Power Valley Benchmark to evaluate and compare the algorithms implemented in the toolbox has been done but without success.

Contents

1 Introduction

1.1 Installation

To install the toolbox first add the path to the location of the toolbox

```
addPath('path to/Distributed Model Predictive Control Toolbox')
```

then run the command

```
dmpc_setup
```

this will permanently add the paths for all files in the toolbox to your Matlab environment. Use e.g the `pathtool` if you want to remove these paths from your Matlab environment.

2 Model Predictive Control

Given a discrete linear time-invariant state space model

$$x_{t+1} = Ax_t + Bu_t \tag{1}$$

$$y_t = Cx_t$$

where x_t is the state vector, u_t is the input vector, y_t is the output vector at time t and A and B are matrices. We want to define some performance measure, $J(k)$, and find a set of future input signals, $\{u_i\}_{i=k+1}^{k+N}$, to the system that are optimal in the sense that they minimize $J(k)$. N is called the prediction horizon and it is the number of time steps over which we want to minimize $J(k)$. See [1] for a detailed introduction to model predictive control.

We denote by $x(k+i|k)$ for $i, k \in \mathbb{N}$ the prediction of x_{k+i} provided the initial state $x_0 = x_k$. Then we can write (1) as

$$x(k+i+1|k) = Ax(k+i|k) + Bu(k+i|k) \tag{2}$$

We can now expand the $x(k+i|k)$ for $i = 0, 1, \ldots, N$ so that

$$\begin{aligned}
x(k+1|k) &= Ax(k|k) + Bu(k|k) \\
x(k+2|k) &= A^2x(k|k) + ABu(k|k) + Bu(k+1|k) \\
x(k+3|k) &= A^3x(k|k) + A^2Bu(k|k) + ABu(k+1|k) + Bu(k+1|k)
\end{aligned}$$

$$\vdots$$

$$x(k+N|k) = A^Nx(k|k) + A^{N-1}Bu(k|k) + \cdots + Bu(k+N-1|k)$$

and if we now form

$$\mathbf{u} = \begin{bmatrix} u(k|k) \\ u(k+1|k) \\ \vdots \\ u(k+N-1|k) \end{bmatrix},$$

we can form the matrices

$$\mathcal{C} = \begin{bmatrix} B & 0 & 0 & & 0 \\ AB & B & 0 & \dots & 0 \\ A^2B & AB & B & & 0 \\ \vdots & & & \ddots & \vdots \\ A^{N-1}B & & \dots & & B \end{bmatrix}$$

and

$$\mathcal{M} = \begin{bmatrix} A \\ A^2 \\ \vdots \\ A^N \end{bmatrix}$$

so that

$$x(k+i|k) = \mathcal{M}_i x(k|k) + \mathcal{C}_i \mathbf{u} \tag{3}$$

where $\mathcal{M}_i$ and $\mathcal{C}_i$ denotes the ith block row of $\mathcal{M}$ and $\mathcal{C}$ respectively.

Suppose we define our performance measure, $J(k)$, to be a quadratic cost function over the prediction horizon so that

$$J(k) = x(k+N|k)^T Q_f x(k+N|k) + \sum_{i=0}^{i=N-1} x(k+i|k)^T Q x(k+i|k) + u(k+i|k)^T R u(k+i|k) \tag{4}$$

where Q_f is called the terminal weight matrix. It is shown in [1] that if we choose Q_f so that it is the solution to the Lyapunov equation

$$Q_f - (A+BK)^T Q_f (A+BK) = Q + K^T R K$$

where K is the LQ-optimal gain, i.e the solution to the algebraic Riccati equation

$$Q + A^T K + KA - KBB^T K = 0 \tag{5}$$

then, given some additional constraints described in [1], the MPC controller will have closed-loop stability and optimal performance.

Now if we define

$$\tilde{Q} = \begin{bmatrix} Q & & & \\ & Q & & \\ & & \ddots & \\ & & & Q_f \end{bmatrix}, \; \tilde{R} = \begin{bmatrix} R & & & \\ & R & & \\ & & \ddots & \\ & & & R \end{bmatrix}$$

we can write $J(k)$ as

$$J(k) = (\mathcal{M}x(k|k) + \mathcal{C}\mathbf{u})^T \tilde{Q} (\mathcal{M}x(k|k) + \mathcal{C}\mathbf{u}) + \mathbf{u}^T \tilde{R} \mathbf{u} + x(k|k)^T Q x(k|k) =$$

$$\mathbf{u}^T (\mathcal{C}^T \tilde{Q} \mathcal{C} + \tilde{R}) \mathbf{u} + 2x(k|k)^T \mathcal{M}^T \tilde{Q} \mathcal{C} \mathbf{u} + x(k|k)^T (\mathcal{M}^T \tilde{Q} \mathcal{M} + Q) x(k|k)$$

Thus minimizing $J(k)$ results in the quadratic programming problem

$$\begin{aligned} \underset{\mathbf{u}}{\text{minimize}} \quad & \mathbf{u}^T H \mathbf{u} + 2x(k|k)^T F^T \mathbf{u} \\ \text{subject to} \quad & \mathbf{u}_{min} \leq \mathbf{u} \leq \mathbf{u}_{max} \\ & \mathbf{y}_{min} \leq \mathbf{y} \leq \mathbf{y}_{max} \end{aligned} \tag{6}$$

where

$$H = \mathcal{C}^T \tilde{Q} \mathcal{C} + \tilde{R}$$
$$F = \mathcal{C}^T \tilde{Q} \mathcal{M}$$

$$\mathbf{u}_{max} = \begin{bmatrix} u_{max} \\ u_{max} \\ \vdots \\ u_{max} \end{bmatrix}, \ \mathbf{u}_{min} = \begin{bmatrix} u_{min} \\ u_{min} \\ \vdots \\ u_{min} \end{bmatrix}$$

$$\mathbf{y}_{max} = \begin{bmatrix} y_{max} \\ y_{max} \\ \vdots \\ y_{max} \end{bmatrix}, \ \mathbf{y}_{min} = \begin{bmatrix} y_{min} \\ y_{min} \\ \vdots \\ y_{min} \end{bmatrix}$$

2.1 Formulating The Constraints

We want to express the constraints in (6) in matrix form

$$A_c \mathbf{u} \leq b + B_x x(k|k)$$

If the prediction horizon is N and $u(k+i|k) \in \mathbb{R}^n$ and we denote by I_n the identity matrix of degree n, we can write

$$\mathbf{u}_{min} \leq \mathbf{u} \leq \mathbf{u}_{max}$$

as

$$\begin{bmatrix} \tilde{I} \\ -\tilde{I} \end{bmatrix} \mathbf{u} \leq \begin{bmatrix} \mathbf{u}_{max} \\ -\mathbf{u}_{min} \end{bmatrix} \tag{7}$$

where

$$\tilde{I} = \begin{bmatrix} I_n & & \\ & \ddots & \\ & & I_n \end{bmatrix}$$

is a block diagonal matrix of N copies of I_n. Now using (3) we can write

$$\mathbf{y}_{min} \leq \mathbf{y} \leq \mathbf{y}_{max}$$

as

$$y_{min} - C\mathcal{M}_i x(k|k) \leq C\mathcal{C}_i \mathbf{u} \leq y_{max} - C\mathcal{M}_i x(k|k), \text{ for } i = 0, 1, \ldots, N-1$$

Hence

$$\begin{bmatrix} \tilde{\mathcal{C}} \\ -\tilde{\mathcal{C}} \end{bmatrix} \mathbf{u} \leq \begin{bmatrix} \mathbf{y}_{max} - \tilde{\mathcal{M}} x(k|k) \\ -\mathbf{y}_{min} + \tilde{\mathcal{M}} x(k|k) \end{bmatrix} \tag{8}$$

where

$$\tilde{\mathcal{C}} = \begin{bmatrix} C\mathcal{C}_1 + D \\ C\mathcal{C}_2 + D \\ \vdots \\ C\mathcal{C}_N + D \end{bmatrix}, \ \tilde{\mathcal{M}} = \begin{bmatrix} C\mathcal{M}_1 \\ C\mathcal{M}_2 \\ \vdots \\ C\mathcal{M}_N \end{bmatrix}$$

Collecting the above results we get

$$A_c\mathbf{u} \leq b + B_x x(k|k)$$

where

$$A_c = \begin{bmatrix} \tilde{I} \\ -\tilde{I} \\ \tilde{\mathcal{C}} \\ -\tilde{\mathcal{C}} \end{bmatrix},\ b = \begin{bmatrix} \mathbf{u}_{max} \\ -\mathbf{u}_{min} \\ \mathbf{y}_{max} \\ -\mathbf{y}_{min} \end{bmatrix},\ B_x = \begin{bmatrix} 0 \\ 0 \\ -\tilde{\mathcal{M}} \\ \tilde{\mathcal{M}} \end{bmatrix} \tag{9}$$

2.2 Prediction and Control Horizon

Suppose we want a prediction horizon N but we only want to optimize the first N_c steps. Then we can let $u(k+i|k)$ be free for $i < N_c$ and let the control law

$$u(k+i|k) = Kx(k+i|k) \tag{10}$$

hold for $i \geq N_c$, where K is the LQ-optimal gain, i.e the solution to (5).

From 2 and 10 we get for $i \geq N_c$

$$x(k+i+1|k) = Ax(k+i|k) + Bu(k+i|k) = (A+BK)x(k+i|k) \tag{11}$$

and the matrices $\mathcal{C}$ and $\mathcal{M}$ will take the form

$$\mathcal{C} = \begin{bmatrix} B & & & \\ AB & B & & \\ A^2B & AB & B & \\ \vdots & & \ddots & \\ A^{N_c-1}B & \cdots & & B \\ (A+BK)A^{N_c-1}B & \cdots & & (A+BK)B \\ \vdots & & & \vdots \\ (A+BK)^{N-N_c}A^{N_c-1}B & \cdots & & (A+BK)^{N-N_c}B \end{bmatrix}$$

$$\mathcal{M} = \begin{bmatrix} A \\ A^2 \\ \vdots \\ A^{N_c} \\ (A+BK)A^{N_c} \\ \vdots \\ (A+BK)^{N-N_c}A^{N_c} \end{bmatrix}$$

We now get the additional constraints for $i \geq N_c$

$$\mathbf{u}_{min} \leq [\mathbf{u} = Kx(k+i|k) = K(A+BK)^{i-N_c}x(k+N_c|k)] \leq \mathbf{u}_{max}$$

$$\Longleftrightarrow$$

$$\mathbf{u}_{min} - K\mathcal{M}_i x(k|k) \leq K\mathcal{C}_i\mathbf{u} \leq \mathbf{u}_{max} - K\mathcal{M}_i x(k|k)$$

We also get the corresponding constraints on $x(k+i|k)$ for $i \geq N_c$ but those are handled by the definition of $\mathcal{C}$ and $\mathcal{M}$.
If we now define

$$\bar{\mathcal{C}} = \begin{bmatrix} K\mathcal{C}_{N_c} \\ K\mathcal{C}_{N_c+1} \\ \vdots \\ K\mathcal{C}_N \end{bmatrix}, \ \bar{\mathcal{M}} = \begin{bmatrix} K\mathcal{M}_{N_c} \\ K\mathcal{M}_{N_c+1} \\ \vdots \\ K\mathcal{M}_N \end{bmatrix}$$

we have that

$$A_c = \begin{bmatrix} \tilde{I} \\ -\tilde{I} \\ \tilde{\mathcal{C}} \\ -\tilde{\mathcal{C}} \\ \bar{\mathcal{C}} \\ -\bar{\mathcal{C}} \end{bmatrix}, \ b = \begin{bmatrix} \mathbf{u}_{max} \\ -\mathbf{u}_{min} \\ \mathbf{y}_{max} \\ -\mathbf{y}_{min} \\ \mathbf{u}_{max} \\ \mathbf{u}_{min} \end{bmatrix}, \ B_x = \begin{bmatrix} 0 \\ 0 \\ -\tilde{\mathcal{M}} \\ \tilde{\mathcal{M}} \\ \bar{M} \\ -\bar{M} \end{bmatrix} \tag{12}$$

We now use these matrices to solve the quadratic programming problem in (6).

2.3 Soft Constraints

To ensure that there is always a feasible solution to the quadratic programming problem in (6) we can use soft constraints. We then have the quadratic programming problem

$$\begin{aligned} \underset{\mathbf{u},\xi,\eta}{\text{minimize}} \quad & \mathbf{u}^T H \mathbf{u} + 2x(k|k)^T F^T \mathbf{u} + \xi^T V^T V \xi + \eta^T W^T W \eta \\ \text{subject to} \quad & \mathbf{u}_{min} - \tilde{V}\xi \leq \mathbf{u} \leq \mathbf{u}_{max} + \tilde{V}\xi \\ & \mathbf{y}_{min} - \tilde{W}\eta \leq \mathbf{y} \leq \mathbf{y}_{max} + \tilde{W}\eta \\ & \xi \geq 0, \eta \geq 0 \end{aligned} \tag{13}$$

where

$$\tilde{V} = \begin{bmatrix} V \\ V \\ \vdots \\ V \end{bmatrix}, \ \tilde{W} = \begin{bmatrix} W \\ W \\ \vdots \\ W \end{bmatrix}$$

and also

$$dim(\xi) = dim(u(k+i|k)), \ dim(\eta) = dim(y)$$

Suppose $dim(u(k+i|k)) = n \times 1$ and $dim(y) = m \times 1$. We can now form

$$\mathbf{z} = \begin{bmatrix} \mathbf{u} \\ \xi \\ \eta \end{bmatrix}$$

so that (13) takes the form

$$\begin{aligned} \underset{\mathbf{z}}{\text{minimize}} \quad & \mathbf{z}^T H_s \mathbf{z} + 2x(k|k)^T F_s^T \mathbf{z} \\ \text{subject to} \quad & A_c \mathbf{z} \leq b + B_x x(k|k) \end{aligned} \tag{14}$$

where

$$H_s = \begin{bmatrix} H & & \\ & V^T V & \\ & & W^T W \end{bmatrix}, \ F_s = \begin{bmatrix} F \\ 0 \\ 0 \end{bmatrix}$$

$$A_c = \begin{bmatrix} \tilde{I} & -\tilde{V} & 0 \\ -\tilde{I} & -\tilde{V} & 0 \\ \tilde{\mathcal{C}} & 0 & -\tilde{W} \\ -\tilde{\mathcal{C}} & 0 & -\tilde{W} \\ 0 & -I_n & 0 \\ 0 & 0 & -I_m \end{bmatrix},\ b = \begin{bmatrix} \mathbf{u}_{max} \\ -\mathbf{u}_{min} \\ \mathbf{y}_{max} \\ -\mathbf{y}_{min} \\ 0 \\ 0 \end{bmatrix},\ B_x = \begin{bmatrix} 0 \\ 0 \\ -\tilde{\mathcal{M}} \\ \tilde{\mathcal{M}} \\ 0 \\ 0 \end{bmatrix}$$

2.4 Implementation in Matlab

The MPC controller is represented in Matlab as an object of the class `Mpc`. The constructor of the `Mpc` object takes as input an `LTI` object and a `params` struct specifying the parameters of the system and of the MPC controller. The `params` struct should contain the following fields

`params.`	`LTI`	- The `LTI` object
	`N`	- Prediction horizon of the MPC controller
	`Nc`	- Control horizon of the MPC controller
	`Q`	- Weight matrix in cost function (4)
	`R`	- Weight matrix in cost function (4)
	`x_0`	- Initial state of the system
	`umax`	- Maximum constraints on input signals
	`umin`	- Minimum constraints on input signals
	`ymax`	- Maximum constraints on output signals
	`ymin`	- Minimum constraints on output signals
	`soft` (optional)	= 0 - hard constraints = 1 - soft constraints on input signals = 2 - soft constraints on output signals = 3 - soft constraints on both input and output signals

Table 1: The fields that need to be set in the `params` struct in order to initiate an `Mpc` object

Methods of the `Mpc` class:

`Mpc`

```
obj :  Mpc(LTI, params)

 LTI      - an LTI object
 params   - the params struct described in Table 2.4
```

The class constructor. To construct a system object the syntax is

```
mpc = Mpc(LTI, params)
```

`initWeights`

```
obj :  initWeights(params)
```

Initiates the matrices for the quadratic programming problem in (13).

Mpc Properties	
LTI	The LTI object
N	Prediction horizon of the MPC controller
Nc	Control horizon of the MPC controller
Q	Weight matrix in cost function (4)
R	Weight matrix in cost function (4)
x_0	Initial state of the system
umax	Maximum constraints on input signals
umin	Minimum constraints on input signals
ymax	Maximum constraints on output signals
ymin	Minimum constraints on output signals
H	Weight matrix in (6) or (14)
F	Weight matrix in (6) or (14)
Ac	Constraint matrix in (9), (12) or (14)
b	Constraint matrix in (9), (12) or (14)
Bx	Constraint matrix in (9), (12) or (14)
Conv	The matrix $\mathcal{C}$
M	The matrix $\mathcal{M}$
soft	= 0 for hard constraints = 1, 2, 3 if MPC is configured for soft constraints (see Table 2.4)

Table 2: The properties of the Mpc class

reCompile
obj : reCompile()

Reinitiates the matrices initiated by initWeights(params). This method should be called if something is changed in the parameters of the Mpc object.

To solve the quadratic programming problem described in (13) for time step $k+1$ given the an initial state x_k the following matrices should be passed to the solver if using Matlabs quadprog solver

```
u = quadprog(mpc.H, mpc.F*mpc.x_k, mpc.Ac, mpc.b + mpc.Bx*mpc.x_k)
```

if using soft constraints the following matrices should be passed to the solver

```
u = quadprog(mpc.Hs, mpc.Fs*mpc.x_k, mpc.Acs, mpc.bs + mpc.Bxs*mpc.x_k)
```

3 Distributed Model Predictive Control

Consider a finite collection $\mathcal{T} \subset \mathbb{N}$ of indices describing discrete time-invariant linear systems such that system $i \in \mathcal{T}$ is described by

$$\begin{aligned} x_i(k+t+1|k) &= A_i x_i(k+t|k) + B_i u_i(k+t|k) \\ y_i(k+t|k) &= C_i x_i(k+t|k) + D_i u_i(k+t|k) \end{aligned} \tag{15}$$

where $x_i \in \mathbb{R}^{p_i}, u_i \in \mathbb{R}^{q_i}, y_i \in \mathbb{R}^{n_i}$, and system i is coupled to system $j \in \mathcal{T}$ with the constraints

$$E_{ij} y_i(k+t|k) = E_{ji} y_j(k+t|k)$$

for some $E_{ij} \in \mathbb{R}^{m_{ij} \times n_i}, E_{ji} \in \mathbb{R}^{m_{ij} \times n_j}$.
By formulating the MPC problem for each system $i \in \mathcal{T}$ we get the quadratic programming problem

$$\begin{aligned} \underset{\mathbf{u}_i \in \mathcal{U}_i}{\text{minimize}} \quad & J(\mathbf{u}_i) \\ \text{subject to} \quad & E_i y_i(k+t|k) = z_i(k+t|k), \forall t \leq N \end{aligned} \tag{16}$$

where $J(\mathbf{u}_i)$ is a quadratic positive definite cost function and

$$y_i(k+t|k) = C_i \mathcal{M}_t^{(i)} x_i(k|k) + C_i \mathcal{C}_t^{(i)} \mathbf{u}_i + \tilde{D}_t^{(i)} \mathbf{u}_i$$

$$\tilde{D}^{(i)} = \begin{bmatrix} D_i & & \\ & \ddots & \\ & & D_i \end{bmatrix}$$

$$\mathcal{U}_i = \{\mathbf{u}_i | A_i \mathbf{u}_i \leq b_i + B_x^{(i)} x_i(k|k)\}$$

where $\mathcal{C}^{(i)}$ and $\mathcal{M}^{(i)}$ are the convolution matrix and exponent matrix for each system $i \in \mathcal{T}$. $\mathcal{C}_t^{(i)}$ denotes the $t-$th block row of $\mathcal{C}^{(i)}$, and

$$z_i(k+t|k) = \begin{bmatrix} E_{ji} y_j(k+t|k) \\ \vdots \end{bmatrix}, \forall j \in \mathcal{N}_i$$

where $\mathcal{N}_i \subseteq \mathcal{T}$ is the set of all coupled neighbors to system i. Hence $z_i(k+t|k)$ is a vector of copies of the constrained variables from the neighboring systems.

3.1 The Global Problem

The global problem is the global quadratic programming problem for which we want to find an optimal solution. This problem is formulated by collecting all subproblems encoded in $\mathcal{T}$ in a block matrix such that we get

$$\begin{aligned} \underset{\mathbf{u} \in \mathcal{U}}{\text{minimize}} \quad & \mathbf{u}^T H \mathbf{u} + 2x(k|k)^T F^T \mathbf{u} \\ \text{subject to} \quad & E\mathbf{u} = c \end{aligned} \tag{17}$$

where

$$\mathbf{u} = \begin{bmatrix} \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_s \end{bmatrix}, \mathcal{U} = \{\mathbf{u} | \mathbf{u}_i \in \mathcal{U}_i, \forall i \in \mathcal{T}\}$$

$$E = \begin{bmatrix} \bar{E}_{1i} & 0 & \dots & 0 & 0 & -\bar{E}_{i1} & 0 \\ \bar{E}_{1j} & 0 & & 0 & -\bar{E}_{j1} & \dots & 0 \\ \vdots & & \ddots & & & & \vdots \\ 0 & 0 & 0 & \bar{E}_{ks} & 0 & \dots & -\bar{E}_{sk} \end{bmatrix},$$

$$\bar{E}_{ij} = E_{ij}(C_i \mathcal{C}^{(i)}_{[1,N_c]} + \tilde{D}^{(i)}_{[1,N_c]})$$

$$c = \begin{bmatrix} E_{i1}C_i\mathcal{M}^{(i)}_{[1,N_c]}x_i(k|k) - E_{1i}C_1\mathcal{M}^{(1)}_{[1,N_c]}x_1(k|k) \\ E_{j1}C_j\mathcal{M}^{(j)}_{[1,N_c]}x_j(k|k) - E_{1j}C_1\mathcal{M}^{(1)}_{[1,N_c]}x_1(k|k) \\ \vdots \\ E_{sk}C_s\mathcal{M}^{(s)}_{[1,N_c]}x_s(k|k) - E_{ks}C_k\mathcal{M}^{(k)}_{[1,N_c]}x_k(k|k) \end{bmatrix},$$

$$H = \begin{bmatrix} H_1 & & \\ & \ddots & \\ & & H_s \end{bmatrix}, F = \begin{bmatrix} F_1 & & \\ & \ddots & \\ & & F_s \end{bmatrix}, x(k|k) = \begin{bmatrix} x_1(k|k) \\ \vdots \\ x_s(k|k) \end{bmatrix}$$

and $s = |\mathcal{T}|$, H_i is the Hessian of $J_i(\mathbf{u}_i)$ for system i and $\mathcal{C}_{[1,N_c]}$ denotes the concatenation of block rows 1 to N_c of $\mathcal{C}$. Hence the matrix E encodes the couplings between the subsystems in $\mathcal{T}$.

3.2 Implementation in Matlab

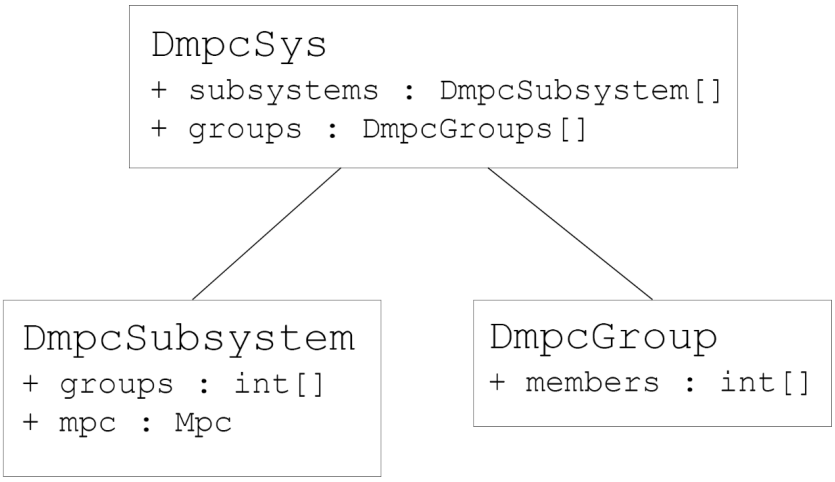

Figure 1: UML class diagram of the classes that represent the distributed system of MPC controllers.

A distributed system of coupled MPC controllers encoded in $\mathcal{T}$ is represented in Matlab by an object of the class `DmpcSys`. Each subsystem $i \in \mathcal{T}$ is represented by an object of the class `DmpcSubsystem` and each variable that is coupled between two (or more) subsystems is handled by an object of the class `DmpcGroup`.

3.2.1 The `DmpcSys` class

`Dmpc` properties	
`subsystems`	a cell array consisting of `DmpcSubsystem` objects representing the subsystems
`n_subsystems`	the number of subsystems
`groups`	a cell array consisting of `DmpcGroup` objects representing the couplings between the subsystems
`n_groups`	the number of groups
`subsID_subs`	a map that maps subsystem ids to positions in the `subsystems` cell array
`groupID_group`	a map that maps group ids to positions in the `groups` cell array
`N`	Prediction horizon of the system
`Nc`	Control horizon of the system
`c_type`	Connection type of the system. The supported connection types are the use of edge variables and node variables in the coupling between the subsystems. = 1 - says that the system is modeled by edge variables = 0 - says that the system is modeled by node variables

---

Methods of the `DmpcSys` class:

`DmpcSys`

```
obj :  DmpcSys(N, Nc)
obj :  DmpcSys(N, Nc, type)
```

`N` - The prediction horizon of the system
`Nc` - The control horizon of the system
`type` - Determines if the system should use edge/node variables. `type` can take the values
= `'Edge'` - which sets `c_type = 1`
= `'Node'` - which sets `c_type = 0`

The class constructor. To construct a `DmpcSys` object the syntax is

```
sys = Dmpc(10, 10)
```

for a system with `N = 10`, `Nc = 10` and which uses edge variables.

`addSubsystem`

```
[obj, id] :  addSubsystem(mpc)
[obj, id] :  addSubsystem(LTI, params)
```

`mpc`	- an `Mpc` object
`LTI`	- an `LTI` object
`params`	- the parameters described in Table 2.4 needed to initiate an `Mpc` object

to add a subsystem to a `Dmpc` object either an `Mpc` object is needed or an `LTI` object together with a `params` struct describing the initiation parameters. Suppose an `Mpc` object is defined in the variable `mpc` and the variable `sys` is a `Dmpc` object, the syntax is then

```
[sys, subsysID] = sys.addSubsystem(mpc)
```

`addGroup`

```
obj :  addGroup(subsIDs, y_index)
```

`subsIDs`	- a vector of all ids of the subsystems that should be added to the group object
`y_index`	- a vector of the coupled output signals in each subsystem

This method is used internally by the `connect` method and is not recommended for use elsewhere.
The method creates a `DmpcGroup` object - a group object - and adds the id of this object to all `DmpcSubsystem` objects that are coupled with the variable the group object represents. The group object is the added to the property `groups`.

`getGroupPosition`

```
position :  getGroupPosition(id)
```

Returns the position of the group with id 'id' in the cell array property `groups`.

`getSubsystemPosition`

```
position :  getSubsystemPosition(id)
```

Returns the position of the subsystem with id 'id' in the cell array property `subsystems`.

connect

```
obj :  connect(subs1, y_index, subs2, y_index)
obj :  connect(subs1ID, y_index, subs2ID, y_index)
obj :  connect(subs1, y_index, type, subs2, y_index, type)
```

`subs1`, `subs2`	- subsystem 1 and 2
`subs1ID`, `subs2ID`	- the id of subsystem 1 and 2
`type`	- the type of coupling. Possible types are `'input'` for coupling in the input signal and `'output'` for coupling in the output signal. Default is `type = 'output'`.
`y_index`	- The index of the coupled variable. If `type = 'output'` the index should be in the output signal and if `type = 'input'` the index should be in the input signal.

If the coupling between two systems is in the input signal, i.e if `type = 'input'`, a new output signal will be added to the subsystem in which the coupling is in the input signal. This new output signal encodes the coupling in the input signal as a coupling in the output signal, this allows the internal system to only consider couplings in the output signals.

The syntax to connect two subsystems that have been added to some `Dmpc` object defined in the variable `sys` is

```
sys = sys.connect(subs1ID, 1, subs2ID, 1)
```

this line connects the first output signal from the subsystem with id `subs1ID` to the first output signal of the subsystem with id `subs2ID`. The syntax to connect two input signals is

```
sys = sys.connect(subs1ID, 1, 'input', subs2ID, 1, 'input')
```

this line connects the first input signal from the subsystem with id `subs1ID` to the first input signal of the subsystem with id `subs2ID`.

print

```
print()
```

Prints some information about this system.

3.2.2 The `DmpcSubsystem` class

A `DmpcSubsystem` object is a representation of an `Mpc` object which is coupled to other `Mpc` objects within a distributed system. Each `DmpcSubsystem` object contains an `Mpc` object and information about how the `Mpc` object is coupled to other `DmpcSubsystem` objects. The coupling of a variable between two `DmpcSubsystem` objects is represented by that the variable in each subsystem is members of the same `DmpcGroup` object.

`DmpcSubsystem` properties	
`id`	the identifier for this subsystem within the global system
`mpc`	the `Mpc` object describing the MPC controller of this subsystem
`groups`	a vector of id's of `DmpcGroup` objects that this subsystem belong to (i.e
`n_groups`	the numb of groups that this subsystem belongs to
`coupled_variables`	a map mapping the index of the output variables of the subsystem to the group id's that the output variables belong to

Methods of the `DmpcSubsystem` class:

`DmpcSubsystem`

`obj : DmpcSubsystem(mpc)`

`mpc` - an `Mpc` object

The class constructor. To construct a system object the syntax is

```
subs = DmpcSubsystem(mpc)
```

`addGroup`

`obj : addGroup(groupID, y_index)`

`groupID` - the id of the group that the output signal with index `y_index` should belong to.
`y_index` - the index of the output signal whose coupling is encoded in the group with id `groupID`.

`addGroup(groupID, y_index)` adds the value of `groupID` to the property `groups` of the object and adds the value of `groupID` to the property `coupled_variables` at position `y_index` so that the output variable with index `y_index` is mapped to the group with group id `groupID`.

`getCoupledVar`

`y_index : getCoupledVar(groupID)`

The method returns the index of the output variable that belongs to the group with id `groupID`.

`isCoupled`

`ret : isCoupled(y_index)`

The method checks if the output variable with index `y_index` belongs to any group. If it belongs to some group the method returns `ret = 1`, else it returns `ret = 0`.

`clearGroups`

`clearGroups()`

Clears the `groups` and the `coupled_variables` properties.

`print`

`print()`
Prints some information about this subsystem.

3.2.3 The `DmpcGroup` class

A `DmpcGroup` object is a representation of a coupled variable between two `DmpcSubsystem` objects. If a variable in a `DmpcSubsystem` object is coupled to another `DmpcSubsystem` object this coupling is represented by the variable (in each subsystem) being a member of the same `DmpcGroup` object.

`DmpcGroup` properties	
`id`	the identifier for this group within the global system
`members`	the id's of the `DmpcSubsystem` objects that belong to this group
`n_members`	the number of members that belong to this group
`coupledVarIds`	a vector containing the index of the output variable that is coupled to this group for each member

Methods of the `DmpcGroup` class:

`DmpcGroup`

```
obj :  DmpcGroup(members, coupledVarIds)
```

`members`	- a vector containing the id's of the `DmpcSubsystem` objects that should belong to this group
`coupledVarIds`	- a vector containing the index of the output variable that is coupled to this group for each member

The class constructor. To construct a group object the syntax is

```
group = DmpcGroup([1 ; 2], [1 ; 1])
```

to create a group for the coupling of the first output variables of the subsystems with id 1 and 2.

`getMembers`

```
members :  getMembers()
```

Returns the `members` property of this object, i.e the id's of the `DmpcSubsystem` objects that belong to this group.

`print`

```
print()
```

Prints some information about this group.

4 Solution Methods

Using the above representation of a distributed system we can implement algorithms to solve the distributed MPC problem. The algorithms implemented in this toolbox are Nesterov's accelerated gradient method, see Section 4.1.1, a multi-step gradient method, see Section 4.1.2, and the Alternating Direction Method of Multipliers (ADMM), see Section 4.2.

4.1 Gradient Methods

In the dual formulation of the local quadratic programming problem (16) for each subsystem we get

$$\underset{\lambda_i}{\text{maximize}} \quad \underset{\mathbf{u}_i \in \mathcal{U}_i}{\text{minimize}} \quad J(\mathbf{u}_i) + \lambda_i^T(\tilde{E}_i \mathbf{y}_i - \mathbf{z}_i) \tag{18}$$

where

$$\tilde{E}_i = \begin{bmatrix} E_i & & \\ & \ddots & \\ & & E_i \end{bmatrix}$$

$$\mathbf{z}_i = \begin{bmatrix} z_i(k|k) \\ \vdots \\ z_i(k+N_c|k) \end{bmatrix}$$

where N_c is the control horizon. The outer problem, i.e the maximization problem, can be solved using gradient ascent. We then want to maximize

$$d(\lambda_i) = \min_{\mathbf{u}_i \in \mathcal{U}_i} J(\mathbf{u}_i) + \lambda_i^T(\tilde{E}_i \mathbf{y}_i - \mathbf{z}_i) \tag{19}$$

and this is done by sequentially moving in the direction of the gradient, $\nabla d(\lambda_i)$, where

$$\nabla d(\lambda_i) = \tilde{E}_i \mathbf{y}_i - \mathbf{z}_i \tag{20}$$

until $\nabla d(\lambda_i) = 0$. Hence we have the gradient ascent algorithm

---

$\forall i \in \mathcal{T}$ in parallell
repeat
  $\mathbf{u}_i^+ = \arg \min_{\mathbf{u}_i \in \mathcal{U}_i} J(\mathbf{u}_i) + \lambda_i^T(\tilde{E}_i \mathbf{y}_i - \mathbf{z}_i)$
  send $\tilde{E}_{ij}\mathbf{y}_i^+$ to all neighbors $j \in \mathcal{N}_i$ and receive $\mathbf{z}_i^+$
  $\lambda_i^+ = \lambda_i + \alpha \nabla d(\lambda_i)$
until *convergence*;

Algorithm 1: Gradient ascent algorithm for dual formulation of distributed MPC.

---

where α denotes the step size. The algorithm terminates when all $\nabla d(\lambda_i)$ are below some threshold.

4.1.1 Fast Gradient Method

The step size in Algorithm 5 is chosen arbitrarily and the convergence of the algorithm is highly dependent of the choice of step size. Hence a deterministic way to chose a step size that yields optimal convergence is desired. One way to deterministically chose the step size is to use the accelerated gradient method described in [5] and choose the step size as described in [6]. The algorithm for the accelerated gradient method is as follows

---

Initialize $\alpha = \frac{\sqrt{5}-1}{2}$, $\gamma = \lambda = 0$
$\forall i \in \mathcal{T}$ in parallell
repeat
  $\mathbf{u}_i^+ = \arg \min_{\mathbf{u}_i \in \mathcal{U}_i} J(\mathbf{u}_i) + \lambda_i^T(\tilde{E}_i \mathbf{y}_i - \mathbf{z}_i)$
  send $\tilde{E}_{ij}\mathbf{y}_i^+$ to all neighbors $j \in \mathcal{N}_i$ and receive $\mathbf{z}_i^+$
  $\gamma_i^+ = \lambda_i + \frac{1}{L}(\tilde{E}_i \mathbf{y}_i - \mathbf{z}_i)$
  $\alpha^+ = \frac{\alpha}{2}(\sqrt{\alpha^2 + 4} - \alpha)$
  $\beta = \frac{\alpha(1-\alpha)}{\alpha^2 + \alpha^+}$
  $\lambda_i^+ = \gamma_i + \beta(\gamma_i^+ - \gamma_i)$
until *convergence*;

Algorithm 2: Accelerated gradient ascent algorithm for dual formulation of distributed MPC.

---

here the step size is chosen as $\frac{1}{L}$ where L is the Lipschitz constant of the gradient $\nabla d(\lambda)$. We observe that the gradient, $\nabla d(\lambda)$, is the gradient of the global MPC problem. For the global problem (17) we have the following dual problem

$$\underset{\lambda}{\text{maximize}} \quad \underset{\mathbf{u} \in \mathcal{U}}{\text{minimize}} \quad J(\mathbf{u}) + \lambda^T(E\mathbf{u} - c) \tag{21}$$

and hence the gradient, $\nabla d(\lambda)$, takes the form

$$\nabla d(\lambda) = E\mathbf{u} - c$$

It is shown in [6] that the optimal choice for the Lipschitz constant L, using the above notation, is

$$L = ||EH^{-1}E^T|| \tag{22}$$

where $||\cdot||$ denotes the maximum singular value.

4.1.2 Multi-Step Gradient Method

A multi-step gradient method is an extension of the gradient ascent algorithm described in Algorithm 1 where previous iterates are used to achieve faster convergence, see e.g [3] for an introduction. The algorithm for the multi-step gradient method is

---

$\forall i \in \mathcal{T}$ in parallell
repeat
  $\mathbf{u}_i^+ = \arg\min_{\mathbf{u}_i \in \mathcal{U}_i} J(\mathbf{u}_i) + \lambda_i^T(\tilde{E}_i \mathbf{y}_i - \mathbf{z}_i)$
  send $\tilde{E}_{ij}\mathbf{y}_i^+$ to all neighbors $j \in \mathcal{N}_i$ and receive $\mathbf{z}_i^+$
  $\lambda_i^+ = \lambda_i + \alpha \nabla d(\lambda_i) + \beta(\lambda_i - \lambda_i^-)$
until *convergence*;

Algorithm 3: Multi-step gradient method for dual formulation of distributed MPC.

In the multi-step gradient method we need to choose two step sizes, α and β. For the global problem (17) it is shown in [3] that the optimal choice of α and β are

$$\alpha = \left(\frac{2}{\sqrt{\lambda_n(EH^{-1}E^T)} + \sqrt{\lambda_1(EH^{-1}E^T)}} \right)^2 \tag{23}$$

$$\beta = \left(\frac{\sqrt{\lambda_n(EH^{-1}E^T)} - \sqrt{\lambda_1(EH^{-1}E^T)}}{\sqrt{\lambda_n(EH^{-1}E^T)} + \sqrt{\lambda_1(EH^{-1}E^T)}} \right)^2 \tag{24}$$

where $\lambda_1(\cdot)$ denotes the smallest eigenvalue and $\lambda_n(\cdot)$ denotes the largest eigenvalue.

4.2 The Alternating Direction Method of Multipliers (ADMM)

The Alternating Direction Method of Multipliers (ADMM) is an extension of the Method of Multipliers which allow a distributed formulation. ADMM uses the augmented Lagrangian in the dual formulation to achieve better convergence properties. See [7] for an introduction to the Method of Multipliers and ADMM.

If we write the local quadratic programming problem (16) as

$$\begin{array}{ll} \underset{\mathbf{u}_i \in \mathcal{U}_i}{\text{minimize}} & \frac{1}{2}\mathbf{u}_i^T H_i \mathbf{u}_i + x_i(k|k)^T F_i^T \mathbf{u}_i \\ \text{subject to} & E_i \mathbf{y}_i = \mathbf{z}_i, \end{array} \tag{25}$$

the augmented Lagrangian is

$$\mathcal{L} = \frac{1}{2}\mathbf{u}_i^T H_i \mathbf{u}_i + x_i(k|k)^T F_i^T \mathbf{u}_i + \lambda_i(E_i \mathbf{y}_i - \mathbf{z}_i) +$$

$$\frac{\rho}{2}||E_i \mathbf{y}_i - \mathbf{z}_i||_2^2$$

where ρ is the step size of ADMM. The algorithm for ADMM is

$\forall i \in \mathcal{T}$ in parallell
repeat
$\mathbf{u}_i^+ = \arg \min_{\mathbf{u}_i \in \mathcal{U}_i} J(\mathbf{u}_i) + \lambda_i^T(\tilde{E}_i \mathbf{y}_i - \mathbf{z}_i) + \frac{\rho}{2}||E_i \mathbf{y}_i - \mathbf{z}_i||_2^2$
send $E_{ij}\mathbf{y}_i^+ + \lambda_i/\rho$ to all $j \in \mathcal{N}_i$ and receive $E_{ji}\mathbf{y}_j^+ + \lambda_j/\rho$
$\mathbf{z}_i^+ = \frac{\sum_{j \in \mathcal{N}_i} E_{ij}^T(E_{ij}\mathbf{y}_i^+ + \lambda_i/\rho + E_{ji}\mathbf{y}_j^+ + \lambda_j/\rho)}{2}$
$\lambda_i^+ = \lambda_i + \rho(E_i \mathbf{y}_i^+ - \mathbf{z}_i^+)$
until *convergence*;
Algorithm 4: ADMM for dual formulation of distributed MPC.

4.3 Implementation in Matlab

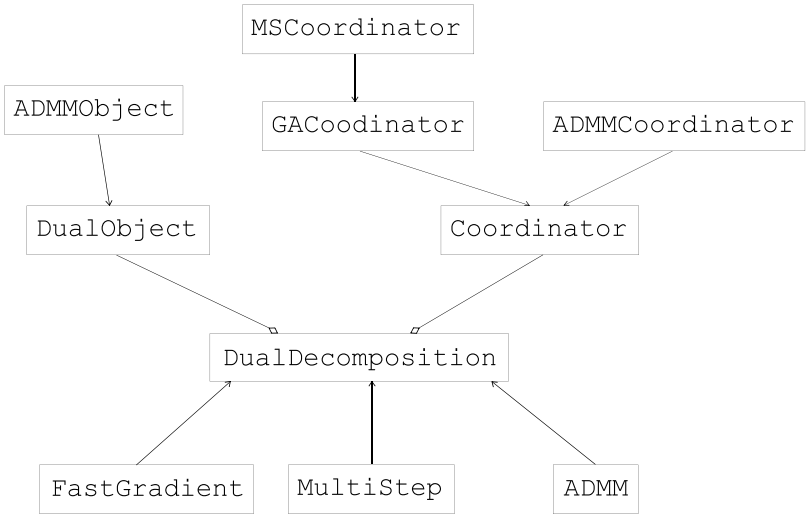

Figure 2: UML class diagram of solution method classes.

4.3.1 The `DualDecomposition` class

The `DualDecomposition` class solves the global MPC problem described by the `DmpcSys` object in a distributed way using dual decomposition. The `DualDecomposition` class is meant to serve as a super class for methods that use dual decomposition to solve the distributed MPC problem. It is in itself not a complete implementation of dual decomposition since it has no implemented dual variable update. The `DualDecomposition` class (or subclasses) uses the `DualObject` class (or subclasses) to represent subsystems and the `Coordinator` class (or subclasses) to manage the communication.
To simulate communication in the distributed system (i.e message passing between objects and coordinators) an object of the class `Communication` is used.

DualDecomposition properties	
objects	a cell array consisting of (subclasses of) DualObject objects representing the subsystems
n_objects	the number of objects (subsystems)
object_map	maps object id's to positions in the objects cell array
coordinators	a cell array consisting of (subclasses of) Coordinator objects representing the couplings between the objects (subsystems)
n_coordinators	the number of coordinators
coordinator_map	maps coordinator ids to positions in the coordinators cell array
com	the Communication object that simulates comminucation between subsystems
histStates	a matrix of previous states
histInputs	a matrix of previous inpus signals
histIter	a vector of the previous number of iterations until convergence
maxIter	the maximum number of iterations (dual variable updates) allowed
tol	the maximum allowed L_2 norm differance in the dual variables between the subsystems
local_qp_solver	the quadratic programming problem solver that should be used to solve the local quadratic programming problems (Default is Matlabs quadprog).
logger	a Log object for logging data.

Methods of the `DualDecomposition` class:

`DualDecomposition`

```
obj :  DualDecomposition(maxIter, tol, local_qp_solver, log)
obj :  DualDecomposition(maxIter, tol, local_qp_solver)
obj :  DualDecomposition(maxIter, tol)
```

`maxIter`	- the maximum number of iterations (dual variable updates) allowed
`tol`	- the maximum allowed L_2 norm differance in the dual variables between the subsystems
`local_qp_solver`	- the quadratic programming problem solver that should be used to solve the local quadratic programming problems (Default is Matlabs quadprog)
`log`	- determines what should be logged using a `Log` object. `log` can be either a `Log` object or one of the macros `Log.[NONE || ALL || SUBSYS || COORINATORS || SOLVER]`.

`mergeSys`

```
obj :  mergeSys(sys)
```

`sys`	- a `DmpcSys` object that will be used to solve the distributed MPC problem.

`mergeSys(sys)` merges the `DmpcSubsystem` objects into `DualObject` (or subclasses) objects and the `DmpcGroup` objects into `Coordinator` (or subclasses) objects.

`setCoordinator`

```
obj :  setCoordinator(index, group)
```

`index`	- position in the `coordinators` cell array
`group`	- a `DmpcGroup` object

Creates a `Coordinator` (or subclass) object using the `group` object and adds it at index `index` in the `coordinators` cell array.

`setObject`

```
obj :  setObject(index, object)
```

`index`	- position in the `objects` cell array
`object`	- a `DmpcSubsystem` object

Creates a `DualObject` (or subclass) object using the `object` object and adds it at index `index` in the `objects` cell array.

`getCoordinatorPosition`

```
position :  getCoordinatorPosition(id)
```

Returns the position of the coordinator with id `id` in the `coordinators` cell array.

`getObjectPosition`

```
position :  getObjectPosition(id)
```

Returns the position of the dual object with id `id` in the `objects` cell array.

`step`

```
obj :  step()
obj :  step(update)
```

`update` - determines if the state of system should be updated after the MPC problem is solved (Default is = 1)
= 1 - if the state should be updated
= 0 - if the state should not be updated

Solves the distributed MPC problem for one time step.

`sim`

```
obj :  sim(steps, hist)
```

`steps` - the number of time steps for which the distributed MPC problem should be solved
`hist` -

Solves the distributed MPC problem for `steps` number of time steps using the local LTI objects to simulate future states.

`getDualGradient`

```
[E, H] : getDualGradient()
```

`E` - the dual gradient of the global MPC problem
`H` - the Hessian of the global MPC problem

Returns the dual gradient (and the Hessian) of the global MPC problem.

`plotHistory`

```
plotHistory()
```

Plots historic information about the system.

`startHist`

```
obj :  startHist()
```

Initiates the `histStates`

`addHist`

`obj : addHist()`

Adds the current states of the subsystems to `histStates` and the current input signals of the subsystems to `histInputs`.

`getName`

`obj : getName()`

Returns the name of this class.

`initLog`

`obj : initLog(log)`

Initiates a `Log` object of type `log`. `log` can take the values `log = Log.[NONE || ALL || SUBSYS || COORINATORS || SOLVER]`.

`shouldLog`

`obj : shouldLog()`

Returns 1 if there is a `Log` object initiated in the property `logger`, else it returns 0.

`log`

`obj : log()`

Logs some data from this object.

`flushLog`

`obj : flushLog()`

Flushes the buffer of the logger.

4.3.2 The `FastGradient` class

The `FastGradient` class is a subclass of the `DualDecomposition` class which implements Nesterov's accelerated gradient method, Algorithm 2, to solve the distributed MPC problem.

`FastGradient` properties	
`L`	the Lipschitz constant of the dual gradient
`method`	sets the method to use for the dual variable update. The possible methods are Algorithm 2 (`'fastgradient'`) and Algorithm 1 (`'gradientascent'`).

Methods of the `FastGradient` class:

`FastGradient`

```
obj : FastGradient(sys, maxIter, tol, local_qp_solver, log,
method, stepsize)
obj : FastGradient(sys, maxIter, tol, local_qp_solver, log,
method)
obj : FastGradient(sys, maxIter, tol, local_qp_solver, log)
obj : FastGradient(sys, maxIter, tol, local_qp_solver)
obj : FastGradient(sys, maxIter, tol)
```

`sys`	- a `DmpcSys` object
`maxIter`	- the maximum number of iterations (dual variable updates) allowed
`tol`	- the maximum allowed L_2 norm differance in the dual variables between the subsystems
`local_qp_solver`	- the quadratic programming problem solver that should be used to solve the local quadratic programming problems (Default is Matlabs quadprog)
`log`	- determines what should be logged using a `Log` object. `log` can be either a `Log` object or one of the macros `Log.[NONE || ALL || SUBSYS || COORINATORS || SOLVER]`.
`method`	- sets the method to use for the dual variable update. = 1 or `'fast gradient'` sets Algorithm 2 = 0 or `'gradientascent'` sets Algorithm 1
`stepsize`	- sets the step size. If no step size is chosen the step size is chosen as 1/L where L is the Lipschitz constant of the dual gradient.

`calcLipschitzConstant`

```
L : calcLipschitzConstant()
```

Calculates the Lipschitz constant, `L`, of the dual gradient using (22).

4.3.3 The `MultiStep` class

The `MultiStep` class is a subclass of the `DualDecomposition` class which implements the multi-step gradient method described in Algorithm 3 to solve the distributed MPC problem.

`MultiStep` properties	
`alpha`	step size of the multi-step gradient method
`beta`	step size of the multi-step gradient method

Methods of the `MultiStep` class:

`MultiStep`

```
obj :  MultiStep(sys, maxIter, tol, local_qp_solver, log)
obj :  MultiStep(sys, maxIter, tol, local_qp_solver)
obj :  MultiStep(sys, maxIter, tol)
```

`sys`	- a `DmpcSys` object
`maxIter`	- the maximum number of iterations (dual variable updates) allowed
`tol`	- the maximum allowed L_2 norm differance in the dual variables between the subsystems
`local_qp_solver`	- the quadratic programming problem solver that should be used to solve the local quadratic programming problems (Default is Matlabs quadprog)
`log`	- determines what should be logged using a `Log` object. `log` can be either a `Log` object or one of the macros `Log.[NONE || ALL || SUBSYS || COORINATORS || SOLVER]`.

`calcStepSizes`

```
[alpha, beta] :  calcStepSizes()
```

Calculates the step sizes `alpha` and `beta` for the multi-step gradient method using (23) and (24).

4.3.4 The `ADMM` class

The `ADMM` class is a subclass of the `DualDecomposition` class which implements the ADMM described in Algorithm 4 to solve the distributed MPC problem. By default ADMM uses step size $\rho = 1$ and over-relaxation parameter $\alpha = 1.6$.

`FastGradient` properties	
`rho`	step size of the ADMM
`alpha`	over-relaxation parameter

Methods of the `ADMM` class:

`ADMM`

```
obj :  ADMM(sys, maxIter, tol)
obj :  ADMM(sys, maxIter, tol, local_qp_solver)
obj :  ADMM(sys, maxIter, tol, local_qp_solver, rho, alpha)
```

`sys`	- a `DmpcSys` object
`maxIter`	- the maximum number of iterations (dual variable updates) allowed
`tol`	- the maximum allowed L_2 norm differance in the dual variables between the subsystems
`local_qp_solver`	- the quadratic programming problem solver that should be used to solve the local quadratic programming problems (Default is Matlabs quadprog)
`rho`	- step size of the ADMM
`alpha`	- over-relaxation parameter

4.3.5 The `DualObject` class

A `DualObject` is a representation of a `DmpcSubsystem` object within the solution method object that is used to solve the distributed MPC problem.

`DualObject` properties	
`id`	the id of this object within the distributed system
`mpc`	the `Mpc` object belonging to this object
`k`	time-step variable
`iter`	the current number of iterations (i.e the number of dual variable updates that has been performed)
`coordinators`	id's of `Coordinator` (or subclasses of) objects that are coupled to this object (subsystem)
`n_coordinators`	the number of coordinators coupled with the object
`solver`	the solver for the local quadratic programming problem
`prev_u`	the current solution to the local qp-problem
`dual_variable`	the dual variable
`coord_dual`	a map mapping coordinator id's to positions in `dual_variable`
`dual_coord`	a map mapping `dual_variable` positions to coordinator id's
`Conv` `M`	The `Conv` and `M` matrices are the matrices such that `y` = `Conv`$*$`prev_u`+`M`$*$`mpc.x_k` where `y` is the output vector of the local system
`f`	modified linear term in the local QP
`ft`	temporary linear term in local QP
`logger`	a `Log` object to log data.
`u_format`	the format in which `prev_u` should be stored
`l_format`	the format in which `dual_variable` should be stored
`y_format`	the format in which the output signal should be stored

Methods of the `DualObject` class:

`DualObject`

```
obj :  DualObject(subsys)

 subsys   - a DmpcSubsystem object
```

`addCoordinator`

```
obj :  addCoordinator(group, y_index)

 y_index   - position in the coordinators cell array
 group     - a DmpcGroup object
```

`initSolver`

```
obj :  initSolver(solver, H, f, A, b, Bx, x, maxIter, tol)
obj :  initSolver(solver, H, f, A, b, Bx, x)
obj :  initSolver(solver)
obj :  initSolver()
```

`solver`	- the name of the solver that should be used (Default = `'quadprog'`)
`H`	- see Section 4.3.12
`f`	- see Section 4.3.12
`A`	- see Section 4.3.12
`b`	- see Section 4.3.12
`Bx`	- see Section 4.3.12
`x`	- initial value
`maxIter`	- the maximum number of iterations (Default = 10^6)
`top`	- the tolerance of the solution (Default = 10^{-6})

`getCouplingMatrix`

```
E : getCouplingMatrix(groupID)
```

Returns the coupling matrix for the group with group id `groupID`.

`initConvMatrix`

```
obj :  initConvMatrix()
```

Initializes the matrices `Conv` and `M`.

`step`

```
[obj, message] :  step()
```

Solves the local problem for one time step. No update is performed on the state of the system, use `updateState()` to perform this update.

`sendMessage`

```
message :  sendMessage()
```

Prepares a message to send to `Coordinator` object(s). The messages to each `Coordinator` contains the value subset of the output vector of this subsystem that is coupled to the `Coordinator`.

`setDualVariable`

```
[obj, epsilon] :  setDualVariable(message)
```

Sets the dual variable (`lambda`) for this object (subsystem). The message should contain subsets of the the updated dual variable from each `Coordinator`, i.e a message should look like

```
message =  [ id of Coordinator j, lambda_ij ;
     id of Coordinator k, lambda_ik ;
     ... ]
```

`getY`

`obj : getY()`

Returns the output vector `y`.

`getPrev_u`

`obj : getPrev_u()`

Returns the current solution the the local problem, `prev_u`.

`updateState`

`obj : updateState()`

Moves this subsystem to the next time step using the computed input signals and the LTI object of the system.

`updateSolver`

`obj : updateSolver()`

Updates values in the local solver of the subsystem to e.g warm start the solver.

`print`

`print()`

`initLog`

`obj : initLog(log)`

Initiates a `Log` object of type `log`. `log` can take the values `log = Log.[NONE || ALL || SUBSYS || COORINATORS || SOLVER]`.

`shouldLog`

`obj : shouldLog()`

Returns 1 if there is a `Log` object initiated in the property `logger`, else it returns 0.

`log`

`obj : log()`

Logs some data from this object.

4.3.6 The `ADMMObject` class

A `ADMMObject` is a subclass of the `DualObject` class that is used within the ADMM solution method to represent subsystems.

`ADMMObject` properties	
`z`	the `z` variable in ADMM
`rho`	step size of the ADMM
`H`	modified Hessian of the local QP
`alpha`	over-relaxation constant
`Ec` `Mc`	Coupling matrices so that `E_i*y_i = Ec*prev_u + Mc*mpc.x_k`

Methods of the `ADMMObject` class:

`ADMMObject`

```
obj :  ADMMObject(subsys, rho, alpha, solver, maxIter, tol)
obj :  ADMMObject(subsys, rho, alpha, solver, maxIter)
obj :  ADMMObject(subsys, rho, alpha, solver)
```

`subsys`	- a `DmpcSubsystem` object
`rho`	- step size of the ADMM
`alpha`	- over-relaxation constant of the ADMM
`solver`	- the solver that should be used for the local QP
`maxIter`	- the maximum number of iterations for the local QP
`top`	- tolerance that should be used for the local QP

`getCouplingMatrixAll`

```
[E, M] : getCouplingMatrixAll()
```

Returns the complete coupling matrices so that $E_i y_i$ =`E*prev_u + M*mpc.x_k` for subsystem i.

`getConstY`

```
y :  getConstY()
```

Returns $E_i y_i$ for subsystem i.

4.3.7 The `Coordinator` class

A `Coordinator` object is meant to serve as a super class for solution methods that use an edge variable formulation of the coupling constraints. A `Coordinator` object can have exactly two members and it can be used directly by solution methods that use a node formulation of the coupling constraints to simply pass messages between its two members without modifying the messages. A `Coordinator` object is an extension of a `DmpcGroup` object within a solution method object.

`Coordinator` properties	
`id`	the id of this coordinator within the distributed system
`members`	vector of `DualObject` (or subclass) object ids
`n_members`	the number of subsystems that are coupled with this coordinator
`coupled_variables`	a matrix consisting of the value of the coupled variable in each subsystem (that is coupled by this variable).
`members_coupled`	a map mapping member id's to positions in `coupled_variables`.
`k`	the current time step
`logger`	a `Log` object to log data.

Methods of the `Coordinator` class:

`Coordinator`

```
obj :  Coordinator(group)
```

`group` - a `DmpcGroup` object

`updateCoupledVariables`

```
[obj] :  updateCoupledVariables(id, val)
```

`id` - the id of a member
`val` - the value of this members dual variable

Stores the values of the coupled variables for this coordinators members.

`evalCoupledVariables`

```
[obj, message, epsilon] :  evalCoupledVariables(message)
```

`message` - a matrix in the format [member id, message to the other member ; ...]. The 'message to the other member' can be either a vector or a cell array.
`epsilon` - this value is not used in this coordinator

It swaps the values of the coupled variables returns this in `message`, the variable `epsilon` is not used and therefore set to zero.

`initLog`

`obj : initLog(log)`

Initiates a `Log` object of type `log`. `log` can take the values `log = Log.[NONE || ALL || SUBSYS || COORINATORS || SOLVER]`.

`shouldLog`

`obj : shouldLog()`

Returns 1 if there is a `Log` object initiated in the property `logger`, else it returns 0.

`log`

`obj : log()`

Logs some data from this object.

`update`

`[obj] : update()`

Moves this coordinator to the next time step by updating the property `k`.

4.3.8 The `GACoordinator` class

`GACoordinator` is a subclass of `Coordinator` and it is used as a coordinator for solution methods that use gradient ascent for the dual variable update. `GACoordinator` supports update rules corresponding to the gradient ascent method (Algorithm 1) and Nesterov's accelerated gradient method [5] (Algorithm 2).

`GACoordinator` properties	
`lambda`	the dual variable
`alpha`	the step size of the gradient method
`mu`	μ in Algorithm 2
`gamma`	γ in Algorithm 2
`method`	determines the gradient method that should be used for the dual variable update = 1 - Nesterov's accelerated gradient method, Algorithm 2 = 0 - the gradient ascent method described by Algorithm 1
`ctr`	counts the number coupled variables that have been received

Methods of the `GACoordinator` class:

`GACoordinator`

```
obj :  GACoordinator(group)
```

`group` - a `DmpcGroup` object

`setStepSize`

```
[obj] :  setStepSize(alpha)
```

`alpha` - the step size of the gradient method

Sets the step size of the gradient method.

`setMethod`

```
[obj] :  setMethod(method)
```

`method` = `'fastgradient'` or 1 sets the method to Algorithm 2
= 0 sets the method to Algorithm 1

Sets the method that should be used for the dual variable update.

`getMethod`

```
[method] :  getMethod()
```

`method` = `'fastgradient'` if method is Algorithm 2
= `'gradient'` if method is Algorithm 1

Returns the method that is used for the dual variable update.

`evalDualVariable`

`[obj] : evalDualVariable()`

Performs the dual variable update using the method determined by the property `method`.

`fastGradient`

`[obj] : fastGradient()`

Performs the dual variable update using Algorithm 2.

4.3.9 The `MSCoordinator` class

`MSCoordinator` is a subclass of `GACoordinator` and it is used as a coordinator for solution methods that use a multi-step gradient method for the dual variable update. `MSCoordinator` performs the dual variable update according to Algorithm 3.

`GACoordinator` properties	
`lambda_prev`	the value of dual variable in the previous iteration

Methods of the `MSCoordinator` class:

`MSCoordinator`

`obj : MSCoordinator(group)`

`group` - a `DmpcGroup` object

`setStepSize`

`[obj] : setStepSize(alpha, beta)`

`alpha` - the step size of the multi-step gradient method
`alpha` - the step size of the multi-step gradient method
Sets the step sizes of this multi-step gradient method.

4.3.10 The `ADMMCoordinator` class

`ADMMCoordinator` is a subclass of `Coordinator` and it is used as a coordinator for solution methods that use the ADMM. `ADMMCoordinator` performs the update of the z variable in Algorithm 4.

`ADMMCoordinator` properties	
`z`	the z variable in Algorithm 4
`ctr`	counts the number coupled variables that have been received

Methods of the `ADMMCoordinator` class:

`ADMMCoordinator`

```
obj :  ADMMCoordinator(group)

 group   - a DmpcGroup object
```

`evalDualVariable`

```
[obj] :  evalDualVariable()
```

Performs the dual variable update using Algorithm 4.

4.3.11 The `Solver` class

A wrapper around Matlabs `quadprog` function. Solves the quadratic programming problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x^T H x + f^T x \\ \text{subject to} \quad & Ax \leq b + B_x x_k \end{aligned} \tag{26}$$

`Solver` properties	
`H`	H in (26)
`f`	f in (26)
`A`	A in (26)
`b`	b in (26)
`Bx`	B_x in (26)
`x_k`	x_k in (26)
`c`	= `b + Bx*x_k`

Methods of the `Solver` class:

`Solver`

```
obj :  Solver(H, f, A, b, Bx, x_k)
```

`H` - H in (26)
`f` - f in (26)
`A` - A in (26)
`b` - b in (26)
`Bx` - B_x in (26)
`x_k` - x_k in (26)

`solve`

```
[obj, x, flag] :  solve(f, x)
```

Solves the quadratic programming problem specified by 26.

`update`

```
[obj] :  update(x_k)
```

Performs the update `c = b + Bx*x_k`.

4.3.12 The `SolverADMM` class

`SolverADMM` is a subclass of `Solver` that uses the ADMM to solve (26).

`Solver` properties	
`Q`	$= -(H + \rho A^T A)^{-1}$ where H and A are as in (26) and ρ is the step size of the ADMM
`rhoA`	ρA^T where A is as in (26) and ρ is the step size of the ADMM
`rho`	the step size of the ADMM
`alpha`	over-relaxation parameter
`z`	the z variable in the ADMM
`u`	$= \lambda/\rho$ where λ is the dual variable in the ADMM and ρ is the step size. `u` is called the scaled dual variable.
`maxIter`	the maximum number of iterations (Default = 10^6)
`tol`	the tolerance (Default = 10^{-6})

Methods of the `SolverADMM` class:

`SolverADMM`

```
obj :  SolverADMM(H, A, b, Bx, x_k)
obj :  SolverADMM(H, A, b, Bx, x_k, maxIter, tol)
```

`H`	- H in (26)
`A`	- A in (26)
`b`	- b in (26)
`Bx`	- B_x in (26)
`x_k`	- x_k in (26)
`maxIter`	- the maximum number of iterations (Default = 10^6)
`tol`	- the tolerance (Default = 10^{-6})

`computeStepSize`

```
[rho, alpha] :  computeStepSize(H, A)
```

`H`	- H in (26)
`A`	- A in (26)
`rho`	- step size in the ADMM
`alpha`	- over-relaxation parameter in the ADMM

Computes the optimal step size, `rho`, using

$$\rho = \frac{1}{\sqrt{\lambda_1(AH^{-1}A^T)\lambda_n(AH^{-1}A^T)}}$$

as described in [2] and sets the optimal over-relaxation parameter, `alpha`, according to [2].

`solve`

`[obj, x, flag] :  solve(f, x)`

Solves the quadratic programming problem specified by (26) using the ADMM.

`update`

`[obj] :  update(x_k)`

Performs the update `c` = `b + Bx*x_k` and warm starts the `z` and `u`.

5 Hydro Power Valley Benchmark

To test and compare the implemented solution methods in the *Distributed Model Predictive Control Toolbox* the Hydro Power Valley Benchmark, available at *http://www.ict-hd-mpc.eu/index.php?page=benchmarks*, was used. In the benchmark we are given system models for eight interconnected hydro power plants, but these system models are not suitable for distributed MPC as is since they are non-linear have non-linear constraints. Therefore the system models in the benchmark have been linearized and discretizised and a model reduction has been performed as described in [4] in order to be able to use distributed MPC to solve the control problem. The implementation of the benchmark closely follows the implementation in [4] and the same Matlab code for performing the linearization, discretization and model reduction of the system models were used.

The control problem that we aim to solve for the benchmark is a reference tracking problem where the problem is to track a power production profile. In the benchmark we are given the cost function

$$J(k) = \int_0^T \gamma \left| p^{tot}(t) - \sum_{i=1}^{8} p_i(x_i(t), u_i(t)) \right| \, dt \, +$$

$$\sum_{i=1}^{8} \int_0^T x_i(t)^T Q_i x_i(t) + u_i(t)^T R_i u_i(t) \, dt \tag{27}$$

where p^{tot} is the total power reference that we want to track and Q_i, γ and R_i are parameters of the benchmark. In order to be able to solve this power reference tracking problem in a distributed way we need to be able to separate (27) for each subsystem. This can be done by using static local power references as described in [4]. So if we use static local power references, a sampling interval T in the discretization of the system and a prediction horizon N we get that the local cost function for each subsystem is

$$J_i(k) = \sum_{t=0}^{N-1} x_i(k+t|k)^T Q_i x_i(k+t|k) + u_i(k+t|k)^T R_i u_i(k+t|k) +$$

$$\gamma \left| p_i^{loc}(t) - p_i(x_i(k+t|k), u_i(k+t|k)) \right|$$

and this can be written as

$$J_i(\mathbf{u}) = \mathbf{u}^T H \mathbf{u} + 2x(k|k)^T F^T \mathbf{u} + \gamma \left| p_i^{ref} - P\mathbf{u} \right| \tag{28}$$

where p_i^{ref} is a vector of the local static power references over the prediction horizon for subsystem i. We thus get that the local problem for each subsystem is a quadratic programming problem with an added ℓ_i-norm term. This problem can be solved using ADMM as described in the next section.

5.1 ADMM with ℓ_1-norm

In our formulation of a distributed system we can write the local optimization problem (28) in the following form.

$$\begin{aligned} \underset{\mathbf{u}}{\text{minimize}} \quad & J(\mathbf{u}) + \gamma|p^{ref} - P\mathbf{u}| \\ \text{subject to} \quad & A_c\mathbf{u} \leq b \end{aligned} \tag{29}$$

and this can be reformulated as

$$\begin{aligned} \underset{\mathbf{u}}{\text{minimize}} \quad & J(\mathbf{u}) + \gamma|\mathbf{v}| \\ \text{subject to} \quad & A_c\mathbf{u} \leq b \\ & P\mathbf{u} = p_{ref} - \mathbf{v} \end{aligned} \tag{30}$$

Then the augmented Lagrangian takes the form

$$\mathcal{L} = J(\mathbf{u}) + \gamma|\mathbf{v}| + \lambda^T(A_c\mathbf{u} - b + \mathbf{z}) + \mu^T(P\mathbf{u} - p^{ref} + \mathbf{v}) +$$

$$\frac{\rho}{2}||A_c\mathbf{u} - b + \mathbf{z}||_2^2 + \frac{\eta}{2}||P\mathbf{u} - p^{ref} + \mathbf{v}||_2^2$$

and assuming that $J(\mathbf{u})$ can be written in the form

$$J(\mathbf{u}) = \frac{1}{2}\mathbf{u}^T H\mathbf{u} + \mathbf{f}^T\mathbf{u}$$

the gradient of the Lagrangian is

$$\frac{\partial\mathcal{L}}{\partial\mathbf{u}} = H\mathbf{u}+\mathbf{f}+A_c^T\lambda+P^T\mu+\rho(A_c^TA_c\mathbf{u}-A_c^Tb+A_c^T\mathbf{z})+\eta(P^TP\mathbf{u}-P^Tp^{ref}+P^T\mathbf{v})$$

$$= (H + \rho A_c^TA_c + \eta P^TP)\mathbf{u} + \mathbf{f} + A_c^T(\lambda + \rho(\mathbf{z} - b)) + P^T(\mu + \eta(\mathbf{v} - p^{ref})) = 0$$

and hence

$$\mathbf{u} = -(H+\rho A_c^TA_c+\eta P^TP)^{-1}(\mathbf{f}+A_c^T(\lambda+\rho(\mathbf{z}-b))+P^T(\mu+\eta(\mathbf{v}-p^{ref}))) \tag{31}$$

$$\frac{\partial\mathcal{L}}{\partial\mathbf{z}} = \rho(\mathbf{z} + A_c\mathbf{u} - b) + \lambda = 0$$

$$\mathbf{z} = \max\{0, -A_c\mathbf{u} + b - \frac{\lambda}{\rho}\} \tag{32}$$

since we want to enforce $\mathbf{z} \geq 0$.

$$\frac{\partial\mathcal{L}}{\partial\mathbf{v}} = \eta(\mathbf{v} + P\mathbf{u} - p^{ref}) + \gamma(\text{sign}(\mathbf{v})) + \mu = 0$$

now using the soft thresholding described in [7] we get

$$\mathbf{v} = \begin{cases} p^{ref} - P\mathbf{u} - \frac{\mu}{\eta} - \frac{\gamma}{\eta} & \text{if } p^{ref} - P\mathbf{u} - \frac{\mu}{\eta} > \gamma/\eta \\ 0 & \text{if } |p^{ref} - P\mathbf{u} - \frac{\mu}{\eta}| \leq \gamma/\eta \\ p^{ref} - P\mathbf{u} - \frac{\mu}{\eta} + \frac{\gamma}{\eta} & \text{if } p^{ref} - P\mathbf{u} - \frac{\mu}{\eta} < \gamma/\eta \end{cases} \tag{33}$$

and thus we have the algorithm

Initialize $\lambda = \mu = \mathbf{z} = \mathbf{v} = 0$

repeat

$\mathbf{u}^+ =$

$-(H + \rho A_c^T A_c + \eta P^T P)^{-1}(\mathbf{f} + A_c^T(\lambda + \rho(\mathbf{z} - b)) + P^T(\mu + \eta(\mathbf{v} - p^{ref})))$

$\mathbf{z}^+ = \max\{0, -A_c\mathbf{u}^+ + b - \frac{\lambda}{\rho}\}$

$$\mathbf{v}^+ = \begin{cases} p^{ref} - P\mathbf{u}^+ - \frac{\mu}{\eta} - \frac{\gamma}{\eta} & \text{if } p^{ref} - P\mathbf{u}^+ - \frac{\mu}{\eta} > \gamma/\eta \\ 0 & \text{if } |p^{ref} - P\mathbf{u}^+ - \frac{\mu}{\eta}| \leq \gamma/\eta \\ p^{ref} - P\mathbf{u}^+ - \frac{\mu}{\eta} + \frac{\gamma}{\eta} & \text{if } p^{ref} - P\mathbf{u}^+ - \frac{\mu}{\eta} < \gamma/\eta \end{cases}$$

$\lambda^+ = \lambda + \rho(A_c\mathbf{u}^+ - b + \mathbf{z}^+)$

$\mu^+ = \mu + \eta(P\mathbf{u}^+ - p^{ref} + \mathbf{v}^+)$

until *convergence*;

Algorithm 5: ADMM with ℓ_1-norm.

5.2 Implementation in Matlab

The classes `ADMMHPV`, `FastGradientHPV`, `MultiStepHPV`, `DualObjectHPV`, `ADMMObjectHPV` and `HPVsolverADMM` have been created as subclasses of `ADMM`, `FastGradient`, `MultiStep`, `DualObject`, `ADMMObject` and `SolverADMM` respectively in order to cope with the ℓ_1-norm term in the cost function and the non-linear constraint in the local problems (as described in [4]). The implementation is although not complete since it does not seem to work properly.

References

[1] Mark Cannon. Lecture notes in Model Predictive Control. *www.eng.ox.ac.uk/ conmrc/ mpc*, 2013.

[2] Euhanna Ghadimi, Andre Teixeira, Iman Shames, Mikael Johansson. Optimal parameter selection for the alternating direction method of multipliers (admm): quadratic problems. *http://arxiv.org/abs/1306.2454*, 2013.

[3] Euhanna Ghadimi, Imam Shames, Mikael Johansson. Multi-step gradient methods for networked optimization. *http://people.kth.se/ euhanna/pub/tsp13.pdf*, 2013.

[4] Minh Dang Doan, Pontus Giselsson, Tamas Keviczky, Bart De Schutter and Anders Rantzer. A distributed accelerated gradient algorithm for distributed model predictive control of a hydro power valley. *http://www.control.lth.se/media/Staff/PontusGiselsson/publications/doanGisCEP2013.pdf*, 2013.

[5] Y. Nesterov. *Introductory lectures on convex optimization.* Springer, 2004.

[6] M. Morari S. Richter and C. N. Jones. Towards computational complexity certification for constrained mpc based on lagrange relaxation and the fast gradient method. *50th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC)*, 2011.

[7] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning Vol. 3 No. 1*, 2010.