Learning Based MPC

Xiaojing Zhang

Documentation for `LBmpcTP` template class
Department of Electrical Engineering and Computer Sciences, UC Berkeley

October 16, 2011

Introduction

The aim of this report is to given a brief introduction to the `LBmpcTP` template class, a particular implemention of the interior point method solver for the learning based MPC algorithm described in [1].

The report is structured as follows: First, the Learning Based MPC model is introduced. Second, the interface to the *LBmpcTP* class is presented. In the third section, some tweaks and hidden parameters are described.

1 The Learning Based MPC model

The learning based MPC model is taken from [1]. In this project, the cost is assumed to be quadratic and the oracle affine in the oracle state $\tilde{x}$ and input $\check{u}$. Furthermore, the feasible sets are assumed to be convex polyhedron.

Hence, we consider the following optimization problem:

$$\min_{c[\cdot],\theta} \quad (\tilde{x}[m+N] - x^\star[m+N])^T \tilde{Q}_f (\tilde{x}[m+N] - x^\star[m+N]) + \tag{1}$$

$$\sum_{i=0}^{N-1} \{ (\tilde{x}_[m+i] - x^\star[m+i])^T \tilde{Q} (\tilde{x}_[m+i] - x^\star[m+i]) + (\check{u}[m+i] - u^\star[m+i])^T R (\check{u}[m+i] - u^\star[m+i]) \}$$

$$\begin{aligned}
\text{s.t.} \quad & \tilde{x}[m] = \hat{x}[m], \quad \bar{x}[m] = \hat{x}[m] \\
& \tilde{x}[m+i] = A\tilde{x}[m+i-1] + B\check{u}[m+i-1] + s + \mathcal{O}_m(\tilde{x}[m+i-1], \check{u}[m+i-1]), \quad \forall i \\
& \mathcal{O}_m(\tilde{x}[m+i-1], \check{u}[m+i-1]) = L_m \tilde{x}[m+i-1] + M_m \check{u}[m+i-1] + t_m, \quad \forall i \\
& \bar{x}[m+i] = A\bar{x}[m+i-1] + B\check{u}[m+i-1], \quad \forall i \\
& \check{u}[m+i-1] = K\bar{x}[m+i-1] + c[m+i-1], \quad \forall i \\
& F_{\bar{x}[m+i]} \bar{x}[m+i] \le f_{\bar{x}[m+i]}, \quad F_{\check{u}[m+i]} \check{u}[m+i] \le f_{\check{u}[m+i]}, \quad \forall i \\
& F_{x\theta} \bar{x}[m+i] + F_\theta \theta \le f_{x\theta}, \quad i \in \{1, \dots, N\}
\end{aligned}$$

The conditions on the matrices above are given in Tab. 1.

To solve (1), the optimization problem (1) is casted into the following quadratic program form:

Tabelle 1: Assumptions on matrices.

$\tilde{Q}$	positive definite
$\tilde{Q}_f$	positive definite
R	positive semidefinite
$F_{\bar{x}[m+i]}$	full rank
$F_{\check{u}[m+i]}$	full rank
F_{θ}	full rank

$$\begin{aligned} \min_z \quad & z^T H z + g^T z \\ \text{s.t.} \quad & Cz = b \\ & Pz \leq b, \end{aligned} \tag{2}$$

where z is the stacked vector:

$$z = \begin{pmatrix} c[m]^T & \bar{x}[m+1]^T & \tilde{x}[m+1]^T & \cdots & c[m+N-1]^T & \bar{x}[m+N]^T & \tilde{x}[m+N]^T & \theta^T \end{pmatrix}^T$$

Next, LBmpcTP template class applies a primal barrier interior-point method. To this end, we add a log-barrier term to the cost function in (2), and get:

$$\begin{aligned} \min_z \quad & z^T H z + g^T z - \kappa \sum_i \log(h - Pz)_i \\ \text{s.t.} \quad & Cz = b. \end{aligned} \tag{3}$$

A sequence of such problems is solved where at each step the cost penalty factor κ is reduced. For each κ, (3) is solved using an infeasible Newton start method.

2 Using the LBmpcTP template class

The LBmpcTP template class is typically called in two seperate steps:

1. Definition of matrices A, B, s, $\tilde{Q}$, $\tilde{Q}_f$, R, K, $\{F_{\bar{x}[m+i]}\}_i$, $\{f_{\bar{x}[m+i]}\}_i$, $\{F_{\check{u}[m+i]}\}_i$, $\{f_{\check{u}[m+i]}\}_i$ and scalars $\epsilon_{barrier}$, ϵ_{nt}, ϵ_{rp}, α, β in MATLAB file `Init.m`. These values are written to a binary file, which is called `ConstrParam.bin` by default.
2. The binary file (e.g. `ConstrParam.bin`) is then read by a C++-file (e.g. `mainLBmpcTP.cpp`). `mainLBmpcTP.cpp` is the main function file and performs two tasks:
   (a) It calls the constructor of the template class in `LBmpcTP.h` and instantiates an object of this template class, e.g. `myObj`.
   (b) It performs the step-function `myObj.step(.)` which returns the optimal input `u_opt`. At each call of the step-function, the following parameters are needed: $\{\tilde{q}[m+i]\}_i$, $\tilde{q}_f$, $\{r[m+i]\}_i$, L_m, M_m, t_m, $\hat{x}$.

The file can be compiled using the gcc-compiler and the following command: `g++ -I /usr/local/include/eigen3/ -O3 mainLBmpcTP.cpp -o mainLBmpcTP`.

In the following sections, both files and the variables are described in more detail.

2.1 MATLAB: `Init.m`

In this MATLAB-file, the parameters required for the instantiation of the `LBmpcTP` object are defined. More specifically, there are two parts:

- User has to manually specify the parameters given in Tab. 2. See also (1).
- A binary file (default: `ConstrParam.bin`) containing the parameters in Tab. 2 is written by calling the m-file `writeParam.m`.

In appendix B.1, a typical implementation of the `Init.m` file is shown.

Tabelle 2: Key parameters in `Init.m`

MATLAB variable	description	typical range/value
`N`	length of MPC horizon	
`m`	number of inputs	
`n`	number of states	
`A`	linear dynamics matrix: $\bar{x}^+ = A\bar{x} + B\check{u} + s$	
`B`	input-state dynamics matrix: $\bar{x}^+ = A\bar{x} + B\check{u} + s$	
`s`	affine offset in state dynamics: $\bar{x}^+ = A\bar{x} + B\check{u} + s$	
`K`	feedback gain matrix, $\check{u} = K\bar{x} + c$, $A + BK$ is stable	
`Q_tilde`	p.d. weight matrix for state	
`Q_tilde_f`	p.d. weight matrix for final state	
`R`	p.s.d. weight matrix on input	
`Fx`$\{i\}$	$F_{\bar{x}[m+i]}\bar{x}[m+i] \leq f_{\bar{x}[m+i]}$, full-rank, $i = 1, \dots, N$	
`fx`$\{i\}$	$F_{\bar{x}[m+i]}\bar{x}[m+i] \leq f_{\bar{x}[m+i]}$	
`Fu`$\{i\}$	$F_{\check{u}[m+i]}\check{u}[m+i] \leq f_{\check{u}[m+i]}$, full-rank, $i = 0, \dots, N-1$	
`fu`$\{i\}$	$F_{\check{u}[m+i]}\check{u}[m+i] \leq f_{\check{u}[m+i]}$	
`F_xTheta`	$F_{x\theta}\bar{x}[m+i] + F_\theta\theta \leq f_{x\theta}$	
`F_theta`	$F_{x\theta}\bar{x}[m+i] + F_\theta\theta \leq f_{x\theta}$, full-rank,	
`f_xTheta`	$F_{x\theta}\bar{x}[m+i] + F_\theta\theta \leq f_{x\theta}$	
`kappa_start_PhaseI`	Phase I starting value of κ to find $z_0 : \; Pz_0 < h$	100
`kappa_start_PhaseII`	Phase II starting value of κ for which (3) is solved	$[1e2, 1e9]$
`mu`	rate at which κ is decreased in (3)	[1/50, 1/2]
`eps_nt`	threshold at which the Newton steps are stopped, i.e. $\| \begin{pmatrix} r_d^T & r_p^T \end{pmatrix} \| \leq$ `eps_nt`	0.1
`eps_normRp`	threshold primal feasibility, i.e. $\|r_p\| \leq$ `eps_normRp`	0.1
`eps_barrier`	bound on suboptimality of solution, i.e. $\kappa\cdot$(# constraints)$\leq$ `eps_barrier`	0.1, depends on J_{opt}
`eps_ls`	lower threshold on step size t in backtracking line search, e.g. $z^+ = z + t * \Delta z$	$1e-7$
`n_iter`	max. number of Newton steps allowed to solve (3) for fixed κ, does not apply for the first step, i.e. $\kappa =$ `kappa_start_PhaseII`	$[5, 50]$
`alpha_ls`	parameter in line search, i.e. $\|r(t)\| \leq (1 - t\cdot$`alpha_ls`$)\|r\|$, where $r = \begin{pmatrix} r_d^T & r_p^T \end{pmatrix}^T$	
`beta_ls`	decrease parameter of t in line search, i.e. $t := \beta t$	[0.1, 0.8]
`reg_PhaseI`	regularization coefficient for PhaseI	0.1
`reg_PhaseII`	regularization coefficient for PhaseII	$[0.001, 1]$

Remarks:

- The number of state constraints is assumed to be constant, i.e. the number of rows in `Fx`$\{i\}$ is constant for all i. Subsequently, the number of state constraints is denoted as `_nSt`.
- The number of input constraints is assumed to be constant, i.e. the number of rows in `Fu`$\{i\}$ is constant for all i. Similarly, the number of input constraints is denoted as `_nInp`.
- The number of constraints involving θ in (1) is assumed to be `_nF_xTheta`.

2.2 C++: mainLBmpcTP.cpp

This file contains the main control routine which interacts with the `LBmpcTP` template class. An example file is provided in appendix B.2. The tasks of `mainLBmpcTP.cpp` include:

- Read values from `ConstrParam.bin`.
- Instantiate an object from the template class `LBmpcTP`, e.g. `myObj`.
- Update the oracle dynamics and retrieve $L_m,\ M_m,\ t_m,\ \hat{x}[m],\ \{x^\star[m+i]\}_i$ from an external source (not provided in this framework).
- Solve the optimization problem (3) by calling the step-function with the updated variables above, i.e. `myObj.step(.)`, and obtain the optimal input `u_opt`.

Since the `LBmpcTP` is a template class and in addition uses the linear algebra template class `Eigen`, slightly more work is needed. A few template parameters must be changed manually. More specifically, the following steps must be completed (a description of the individual parameters can be found in Tab. 3):

1. **SPECIFY parameters:** `N`, `m`, `n`, `_nSt`, `_nInp`, `_nF_xTheta`.
2. **SPECIFY sizes of matrices:** For each of the matrices, their respective dimensions have must be specified *manually*, the the matrices are instances of the EIGEN template library. Note that the purpose of the suffixes `_arg` at the end of each variable is simply to remind the user that those matrices are passed on as arguments to the constructor.
3. `mainLBmpcTP.cpp` reads the parameters (with `_arg` appended) listed in Tab. 2 from the binary source file `ConstrParam.bin`.
4. **SPECIFY template parameters for object instantiation:** To instantiate a `LBmpcTP` template class object (e.g. `myObj`), the first set of parameters in Tab. 3 must be inserted *explicitly* as template parameters.
5. The oracle matrices `Lm_arg`, `Mm_arg`, `tm_arg`, `x_hat_arg`, `x_star_arg[.]` are updated. Note: Computation of these values are not provided by the `LBmpcTP` template class and can be computed in a separate file.
6. The optimization problem (1) is then solved with the updated oracle matrices by calling the member function `myObj.step(.)`.

2.3 C++ template class: LBmpcTP.h

This section gives a rough overview of what happens inside the `LBmpcTP` class. Access to the class is possible through two methods, the constructor and the `step(.)` method. The main idea of the algorithm is based on [2]. The constructor initializes some of the private variables as discussed in the previous sections. The `step(.)`-method performs the following tasks:

Tabelle 3: Key parameters in `mainLBmpcTP.cpp`

variable	description	default
`Type`	only "double" is supported	double
`N` and `_N`	length of MPC horizon	
`m` and `_m`	number of inputs	
`n` and `_n`	number of states	
`_nSt`	number of state constraints (constant over the horizon)	
`_nInp`	number of input constraints (constant over the horizon)	
`_nF_xTheta`	number of constraints involving θ in (1)	
`_pos_Omega`	index i in $F_{x\theta}\bar{x}[m+i] + F_\theta\theta \leq f_{x\theta}$	
`Lm_arg`	oracle matrix, i.e. $\mathcal{O}_m(\tilde{x}[m+i], \check{u}[m+i]) = L_m\tilde{x}[m+i] + M_m\check{u}[m+i] + t_m$	
`Mm_arg`	oracle matrix, i.e. $\mathcal{O}_m(\tilde{x}[m+i], \check{u}[m+i]) = L_m\tilde{x}[m+i] + M_m\check{u}[m+i] + t_m$	
`tm_arg`	oracle matrix, i.e. $\mathcal{O}_m(\tilde{x}[m+i], \check{u}[m+i]) = L_m\tilde{x}[m+i] + M_m\check{u}[m+i] + t_m$	
`x_hat_arg`	current state estimate, i.e. $\tilde{x}[m] = \hat{x}[m], \quad \bar{x}[m] = \hat{x}[m]$	
`x_star_arg[.]`	array of desired states in cost function (1)	
`u_opt`	stores the optimal input from `myObj.step(.)`	

- We recursively compute the sequence $\{u^\star[m+i]\}_i$ from the given desired state sequence $\{x^\star[m+i]\}_i$ by solving
$$x^\star[m+i] = (A + L_m)x^\star[m+i-1] + (B + M_m)u^\star[m+i-1] + (s + t_m)$$
using SVD.
- Cast (1) into (2).
- If the warm start variable `z_warm` does not satisfy P`z_warm` $< h$, then the method `PhaseI()` computes a new `z_warm` which satisfies the above (strict) inequality.
- Solve (2) by solving a sequence of (3). It can be shown that as κ approaches 0, the solution of (3) will approach the solution of (2).
- Finally, it returns the control input based on the `epsilon_barrier`-suboptimal solution or if the number of Newton iterations exeeds `n_iter`.

3 Hidden Parameters, Tweaks

Some of the parameters given in 2 can be used to tweak the `LBmpcTP` template class for the following purposes:

- The problem cannot be solved with the default parameters. It either does not converge or the code returns `nan`.
- Convergence is too slow for the desired purpose, i.e. optimization step need too many Newton or back-tracking line search steps.
- The exact solution is not desired and an approximate solution suffices to speed up algorithm.

The goal of this section is to share some experience of how to react to certain situations and give some general advice on how to choose the parameters.

Tab. 4 lists the tuning parameters from Tab. 2 and describes their role and influence in greater detail.

Tabelle 4: Tuning parameters defined in `Init.m`

tuning variable	influence	typical range/value
`kappa_start_PhaseI`	If Phase I needs a lot of Newton iterations to find a suitable `z_warm`, increasing `kappa_start_PhaseI` might reduce this number.	100
`kappa_start_PhaseII`	Small κ can lead to poorly conditioned matrices. If many iterations ($\gtrapprox 100$) are needed to solve (3) for $\kappa =$ `kappa_start_PhaseII`, starting with a larger `kappa_start_PhaseII` might help. Downside: a bigger sequence of problems of form (3) must be solved.	$[1e2,\ 1e9]$
`mu`	The smaller `mu`, the fewer problems of form (3) has to be solved. However, if `mu` is too small, i.e. $\mu\kappa \ll \kappa$, the problems might become difficult to solve, resulting in `nan`. This is especially the case if the minimizer lies on a face of the feasible set.	$[1/50,\ 1/2]$
`eps_nt`	The smaller this value, the more accurately are the intermediate problems of form (3) solved. However, there is usually no need to exactly solve the intermediate problems.	0.1
`eps_normRp`	Should not be set too large, because primal feasibility ought to be achieved accurately.	0.1
`eps_barrier`	If the optimal solution $J^\star$ was known, then one could simply allow an deviation of several percentages. However, $J^\star$ is usually not known apriori, making it difficult to choose.	0.1
`eps_ls`	The lower bound is motivated by the fact that for very small step sizes, $z^+ = z+t\Delta z \approx z$. Hence, it is better to stop the backtracking line search and perform the next Newton step.	$1e-7$
`n_iter`	This parameter bounds the max. number of Newton steps. It is often observed that, especially for small κ, only the first few Newton steps have significant step size t. The following Newton steps often have step sizes of order 10^{-17}, having no effect on the updates of z. However, if `n_iter` is set to be too small, the problems might be solved inaccurately, leading to poor, even no, solutions.	$[5, 50]$
`alpha_ls`	typically 0.01	
`beta_ls`	If set very small, we t decreases rapidly, thus missing the optimal t by far. This might lead to having to perfom more Newton steps, which are expensive. However, with a large `beta_ls`, too many $t := \beta t$ steps are needed, slowing down the algorithm.	$[0.1,\ 0.8]$
`reg_PhaseI`	This regularization term is needed to cast the Phase I formulation (which is a linear program) into a quadratic program, which we can solve using the same tools as for (3). If the Phase I algorithm does not run as anticipated, decreasing `reg_PhaseI`, which influences the quadratic term, can help.	0.1
`reg_PhaseII`	It can be often observed that if the minimzer z_{opt} lies on the face of the polyhedron, the algorithm runs into numerical problem when computing the Cholesky decomposition. To avoid this, the quadratic cost H in (3) is regularized using the coefficient `reg_PhaseII`. Note that if many Newton steps are performed, the algorithm more likely runs into numerical problems, requiring a larger `reg_PhaseII`.	$[0.001, 1]$

3.1 Compiling

Some compilers provide the option to optimize the code. For example, the gcc compiler allows the user to add the `-O3` option which reduces the size of the executable file and increases the performance of the generated at the expense of longer compiling time and more memory usage: `g++ -I /usr/local/include/eigen3/ -O3 mainLBmpcTP.cpp -o mainLBmpcTP`

3.2 Troubleshooting

In this section, some common errors are described. Possible sources for these errors are given and solutions are proposed.

1. **PhaseI: does not converge or required too many Newton steps.**
   Solutions:
   - Many Newton steps may indicate that the matrices are badly conditioned. Try to increase the value `kappa_start_PhaseI`.
   - Try to decrease the value of `reg_PhaseI` to give the quadratic cost less weight.
   - If still a lot of Newton steps are required, i.e. more than the predefined limit of 200 steps, this number may be increased. This can be done by accessing the `LBmpcTP.h` class and jump to the `PhaseI()` job.
2. **PhaseII: many Newton steps ($\gtrapprox 100$) are required to solve** (3).
   Solutions:
   - If for $\kappa =$ `KAPPA_START_PHASEII`: Increase `kappa_start_PhaseII`. For example, increasing the value of `kappa_start_PhaseI` from $\kappa = 100$ to $\kappa = 1e9$ reduced the number of Newton iterations from 170 to 5 for a particular problem. However, the larger `kappa_start_PhaseI`, the more problems of form (3) must be solved, until κ is sufficiently small.
   - If for $\kappa \neq$ `KAPPA_START_PHASEII`: This can be typically observed for small κ ($\kappa \lessapprox 0.1$), where many Newton steps with small step sizes ($t \approx 10^{-17}$) are performed. Since this is waste of computational time, the number of Newton iterations can be upperbounded by setting **n_iter**. Depending on the problem setup, numbers as few as 5 iterations might be enough to produce usable results.
3. **PhaseI/PhaseII: returned result is a vector of `nan` (not a number).** *Solutions:*
   - This problem typically emerges shows up during Phase II when z_{opt} lies on some face of the feasible set. Even though positive definiteness (and hence the existens of Cholesky decomposition) is theoretically guaranteed, this might not be true from a numerical point of view. Indeed, a `nan` often suggests that some of the eigenvalues numerically approach zero, ending up dividing by zero, leading to `nan`. To solve this, the cost function inf (3) is regularized using the weight term `reg_PhaseI` or `reg_PhaseII`. Thus, choosing a larger`reg_PhaseI`/`reg_PhaseII` usually helps.
4. **Choosing `eps_barrier`:** Difficult, since choosing it small when the optimal value is large is only a waste of computational power. However, if the order of the optimal value is know approximately, then `eps_barrier` can be chosen accordingly.

3.3 Additional Remarks

- So far, the algorithm only works for a minimum prediction horizon of 3.
- There are more parameters in the `LBmpcTP.h` file which can be used to improve the performance of the solver, such as the `difference` variable in PhaseI or how to regularize the cost functions. However, it usually suffices to work with the parameters given in Tab. (4).

- If the prediction horizon $N \geq 50$, then some variable definitions in the class file have to be changed. More precisely, the size of the preallocated array of the `LLT`-class must be increased. This is done easiest by searching for the term `LLT` in the variable definition in `LBmpcTP.h`.

A Figures

B Example Files

B.1 Init.m

```
%% Init.m
% Writes relevant data to binary file.
% author: Xiaojing ZHANG
% date: October 17, 2011


% clc; clear all;
format('short');

%% MPC parameters:
N = 10;      % MPC horizon
m = 2;      % # input
n = 5;      % # states

%% System dynamic parameters

A = [1 0 1.2 1.3 1
    0.5 2.1 1 1 -0.3
    1 1 .2 1 -2
    0 1 0.3 1.4 -2
    0.4 -0.9 2 1.2 -.4];

B = [1 0
    1.3 1
    0 1.2
    -0.1 1
    0.2 -1];

s = [0 ; 2 ; 1.4 ; 2 ; 1];

K = -[  -0.687725010189527   1.970370349984470  -0.865901978685416  -3.069636538756281  2.096473307971948
   0.181027584433678   1.040671203681152  -0.344287251091615   0.362844179335401  -1.109614558033092];

%% cost and constraint matrices
Q_tilde = eye(n);
Q_tilde_f = Q_tilde+1;

R = eye(m);

% constraint matrices: constrained on
H = eye(n); k = 1000*ones(n,1);
Fx{1} = [H ; -H];
Fx{2} = [H ; -H];
Fx{3} = [H ; -H];
Fx{4} = [H ; -H];
Fx{5} = [H ; -H];
Fx{6} = [H ; -H];
Fx{7} = [H ; -H];
Fx{8} = [H ; -H];
Fx{9} = [H ; -H];
Fx{10} = [H ; -H];
fx{1} = [k ; k]-3;
fx{2} = [k ; k]-0;
fx{3} = [k ; k];
fx{4} = [k ; k];
fx{5} = [k ; k]-2;
fx{6} = [k ; k]+3;
fx{7} = [k ; k]+4;
fx{8} = [k ; k]+2;
fx{9} = [k ; k]-10;
```

```
fx{10} = [k ; k]+10;


H = eye(m); k = 100*ones(m,1);
Fu{1} = [H ; -H];
Fu{2} = [H ; -H];
Fu{3} = [H ; -H];
Fu{4} = [H ; -H];
Fu{5} = [H ; -H];
Fu{6} = [H ; -H];
Fu{7} = [H ; -H];
Fu{8} = [H ; -H];
Fu{9} = [H ; -H];
Fu{10} = [H ; -H];
fu{1} = [k ; k+20]+1;
fu{2} = [k+4 ; k]-0;
fu{3} = [k ; k]+5;
fu{4} = [k ; k]+10;
fu{5} = [k ; k-10];
fu{6} = [k ; k]+2;
fu{7} = [k ; k]-7;
fu{8} = [k ; k]+10;
fu{9} = [k ; k]-10;
fu{10} = [k+10 ; k];


F_xTheta = [ 1  1  1  0  0
            -1 -1 -1  0  0
             0  0  0  1  1
             0  0  0 -1 -1
             1  0  1  0  0
            -1  0 -1  0  0
             0  0  0  1  0
             0  0  0 -1  0
             0 0   -1  1  1
             0  0  1 -1  -1];
F_theta = [ 1  0
           -1  0
            0  1
            0 -1
            0  1
            0 -1
            1  0
           -1  0
            0  1
            0 -1];

f_xTheta = 100*[20 20 20 20 30 30 40 40 50 50]';


%% Parameters for constructor
kappa_start_PhaseI = 100;  % barrier parameter for PhaseI - can be as high as 1e10
kappa_start_PhaseII = 100;  % barrier parameter for PhaseII -
mu = 1/10;   % decrease parameter of kappa, i.e. kappa := mu*kappa
eps_nt = 0.1;   % tolerance for residua norm([r_p ; r_d]) in Newton method
eps_normRp = 0.1;   % tolerance for primal residua norm(r_p)
eps_barrier = 0.1;  % barrier parameter, suboptimality of the solution
eps_ls = 1e-7;     % smallest t, s.t. z+ = z + t*dz, nu+ = nu + t*dnu
n_iter = 10;    % number of maximum Newton iterations for a fixed kappa in PhaseII
alpha_ls = 0.01; % alpha parameter in line search, (0.01,0.3)
beta_ls = 0.5;  % 0.1 < beta_ls < 0.8
reg_PhaseI = 0.0001; % regularization Term in PhaseI
reg_PhaseII = 0;  % regularization Term in PhaseII


%% write data for constructor arguments into file
```

```
% ConstrParam.bin

writeParam      % call writeParam.m

disp(['new parameters written to binary file']);
```

B.2 mainLBmpcTP.cpp

```cpp
// mainLBmpcTP.cpp
// example file to test simple examples
// date: October 17, 2011
// author: Xiaojing ZHANG
//
// horizon: N = 4
// states: n = 5
// input: m = 2;

// matrices are imported from binary file created by MATLAB


#include <iostream>     // I-O
#include <fstream>      // read binary data
#include <Eigen/Dense>  // matrix computation
#include "LBmpcTP.h"    // class template

using namespace Eigen;
using namespace std;

int main()
{
    // ------ SPECIFY parameters -------
    int N = 10;     // MPC horizon
    int m = 2;      // #input
    int n = 5;      // #states
    int _nSt = 10;  // # state constraints
    int _nInp = 4;  // # state constraints
    int _nF_xTheta = 10;    // # Omega constraints

    // ----------- SPECIFY sizes of matrices ---------------
    Matrix<double, 5, 5> A_arg;     // n x n
    Matrix<double, 5, 2> B_arg;     // n x m; resizng for non-square matrices doesn't work
    Matrix<double, 5, 1> s_arg;     // n x 1
    Matrix<double, 5, 5> Q_tilde_arg;   // n x n
    Matrix<double, 5, 5> Q_tilde_f_arg; // n x n
    Matrix<double, 2, 2> R_arg;         // m x m
    Matrix<double, 2, 5> K_arg;         // m x n
    Matrix<double, 10, 5> Fx_arg[10];       // _nSt x n, [_N]
    Matrix<double, 10, 1> fx_arg[10];       // _nSt x 1, [_N]
    Matrix<double, 4, 2> Fu_arg[10];        // _nInp x m, [_N]
    Matrix<double, 4, 1> fu_arg[10];        // _nInp x 1, [_N]

    Matrix<double,10,5> F_xTheta_arg;   // _nF_xTheta x n
    Matrix<double,10,2> F_theta_arg;    // _nF_xTheta x m
    Matrix<double,10,1> f_xTheta_arg;   // _nF_xTheta x 1

    Matrix<double, 5, 5> Lm_arg;        // n x n
    Matrix<double, 5, 2> Mm_arg;        // n x m
    Matrix<double, 5, 1> tm_arg;        // n x 1
    Matrix<double, 5, 1> x_hat_arg;     // n x 1, state estimate
    Matrix<double, 5, 1> x_star_arg[10];    // n x 1, [_N], tracking
    Matrix<double, 2, 1> u_opt;         // m x 1, optimal input is saved there


    // ---------- no changes necessary ---------
    double kappa_arg;   // for PhaseII
    double kappa_PhaseI_arg;    // for PhaseI
    int n_iter_arg;
    double mu_arg;
    double eps_barrier_arg;
    double eps_nt_arg;
    double eps_normRp_arg;
    double eps_ls_arg;
    double alpha_ls_arg;
    double beta_ls_arg;
    double reg_arg;     // regularization term for PhaseII
    double reg_PhaseI_arg;  // regularization term for PhaseI

    // ---------------- read from binary file ------------------
    ifstream fin;               // Definition input file object
    fin.open("ConstrParam.bin", ios::binary);   //  open file
    if (!fin.is_open())
    {
        cout << "File open error \n";
        return 1;
    }

    // read
    fin.read((char *) &kappa_arg, sizeof(double));
    fin.read((char *) &kappa_PhaseI_arg, sizeof(double));
    fin.read((char *) &n_iter_arg, sizeof(int));
    fin.read((char *) &mu_arg, sizeof(double));
    fin.read((char *) &eps_barrier_arg, sizeof(double));
    fin.read((char *) &eps_nt_arg, sizeof(double));
    fin.read((char *) &eps_normRp_arg, sizeof(double));
    fin.read((char *) &eps_ls_arg, sizeof(double));
    fin.read((char *) &alpha_ls_arg, sizeof(double));
    fin.read((char *) &beta_ls_arg, sizeof(double));
    fin.read((char *) &reg_arg, sizeof(double));
    fin.read((char *) &reg_PhaseI_arg, sizeof(double));


    // read A_arg
    for (int i = 0; i <= n-1; i++)
    {
        for (int j = 0; j <= n-1; j++)
        {
            fin.read((char *) &A_arg(j,i), sizeof(double));
        }
    }

    // read B_arg
    for (int i = 0; i <= m-1; i++)   // #columns
    {
        for (int j = 0; j <= n-1; j++) // #rows
        {
            fin.read((char *) &B_arg(j,i), sizeof(double));
        }
    }

    // read s_arg
    for (int i = 0; i <= n-1; i++)   // #columns
    {
        fin.read((char *) &s_arg(i,0), sizeof(double));
    }

    // read Q_tilde_arg
    for (int i = 0; i <= n-1; i++)   // #columns
    {
        for (int j = 0; j <= n-1; j++) // #rows
        {
            fin.read((char *) &Q_tilde_arg(j,i), sizeof(double));
        }
    }

    // read Q_tilde_f_arg
    for (int i = 0; i <= n-1; i++)   // #columns
    {
        for (int j = 0; j ≤ n-1; j++) // #rows
        {
            fin.read((char *) &Q_tilde_f_arg(j,i), sizeof(double));
        }
    }

    // read R_arg
    for (int i = 0; i ≤ m-1; i++)   // #columns
    {
        for (int j = 0; j ≤ m-1; j++) // #rows
        {
            fin.read((char *) &R_arg(j,i), sizeof(double));
        }
    }

    // read Fx_arg[]
    for (int k = 0; k ≤ N-1; k++)
    {
        for (int i = 0; i ≤ n-1; i++)   // #columns
        {
            for (int j = 0; j ≤ _nSt-1; j++) // #rows
            {
                fin.read((char *) &Fx_arg[k](j,i), sizeof(double));
            }
        }
    }

    // read fx_arg[]
    for (int k = 0; k ≤ N-1; k++)
    {
        for (int i = 0; i ≤ _nSt-1; i++)    // #columns
        {
                fin.read((char *) &fx_arg[k](i,0), sizeof(double));
        }
    }

    // read Fu_arg[]
    for (int k = 0; k ≤ N-1; k++)
    {
        for (int i = 0; i ≤ m-1; i++)   // #columns
        {
            for (int j = 0; j ≤ _nInp-1; j++) // #rows
            {
                fin.read((char *) &Fu_arg[k](j,i), sizeof(double));
            }
        }
    }

    // read fu_arg[]
    for (int k = 0; k ≤ N-1; k++)
    {
        for (int i = 0; i ≤ _nInp-1; i++)   // #columns
        {
                fin.read((char *) &fu_arg[k](i,0), sizeof(double));
        }
    }

    // read F_xTheta_arg
    for (int i = 0; i ≤ n-1; i++)   // #columns
    {
        for (int j = 0; j ≤ _nF_xTheta-1; j++) // #rows
        {
            fin.read((char *) &F_xTheta_arg(j,i), sizeof(double));
        }
    }

```

```
// read F_theta_arg
for (int i = 0; i <= m-1; i++)   // #columns
{
    for (int j = 0; j <= _nF_xTheta-1; j++) // #rows
    {
        fin.read((char *) &F_theta_arg(j,i), sizeof(double));
    }
}

// read f_xTheta_arg
for (int i = 0; i <= _nF_xTheta-1; i++)  // #columns
{
    fin.read((char *) &f_xTheta_arg(i,0), sizeof(double));
}

// read K_arg
for (int i = 0; i <= n-1; i++)   // #columns
{
    for (int j = 0; j <= m-1; j++) // #rows
    {
        fin.read((char *) &K_arg(j,i), sizeof(double));
    }
}

fin.close();                // close file

/*
cout << "kappa_arg: " << kappa_arg << endl << endl;
cout << "kappa_PhaseI_arg " << kappa_PhaseI_arg << endl << endl;
cout << "n_iter_arg: " << n_iter_arg << endl << endl;
cout << "mu_arg: " << mu_arg << endl << endl;
cout << "eps_barrier_arg: " << eps_barrier_arg  << endl << endl;
cout << "eps_nt_arg: " << eps_nt_arg << endl << endl;
cout << "eps_normRp_arg: " << eps_normRp_arg << endl << endl;
cout << "eps_ls_arg: " << eps_ls_arg << endl << endl;
cout << "alpha_ls_arg: " << alpha_ls_arg << endl << endl;
cout << "beta_ls_arg: " << beta_ls_arg << endl << endl;
cout << "n_iter_arg: " << n_iter_arg << endl << endl;
cout << "reg_arg: " << reg_arg << endl << endl;
cout << "reg_PhaseI_arg: " << reg_PhaseI_arg << endl << endl;
cout << "A_arg: " << endl << A_arg << endl << endl;
cout << "B_arg: " << endl << B_arg << endl << endl;
cout << "s_arg:" << endl << s_arg << endl << endl;
cout << "Q_tilde_arg: " << endl << Q_tilde_arg << endl << endl;
cout << "Q_tilde_f_arg: " << endl << Q_tilde_f_arg << endl << endl;
cout << "R_arg: " << endl << R_arg << endl << endl;

for (int i = 0; i<= N-1; i++)
{
    cout << "Fx_arg[" << i << "]: " << endl << Fx_arg[i] << endl << endl;
}
for (int i = 0; i <= N-1; i++)
{
    cout << "fx_arg[" << i << "]: " << endl << fx_arg[i] << endl << endl;
}
for (int i = 0; i <= N-1; i++)
{
    cout << "Fu_arg[" << i << "]: " << endl << Fu_arg[0] << endl << endl;
}
for (int i = 0; i <= N-1; i++)
{
    cout << "fu_arg[" << i << "]: " << endl << fu_arg[i] << endl << endl;
}
cout << "F_xTheta_arg: " << endl << F_xTheta_arg << endl << endl;
cout << "F_theta_arg: " << endl << F_theta_arg << endl << endl;
cout << "f_xTheta_arg: " << endl << f_xTheta_arg << endl << endl;
```

```
    cout << "K_arg:" << endl << K_arg << endl << endl;
    */


    // -------- SPECIFY template parameters for object instantiation -----------
    // template <class Type, int _n, int _m, int _N, int _nSt, int _nInp, int _nF_xTheta, int _pos_Omega>
     LBmpcTP<double, 5, 2, 10, 10, 4, 10, 10> myObj(                // constructor
                                kappa_arg, kappa_PhaseI_arg, n_iter_arg,  mu_arg,  eps_barrier_arg,
eps_nt_arg, eps_normRp_arg, eps_ls_arg,
                                alpha_ls_arg, beta_ls_arg, reg_arg, reg_PhaseI_arg, A_arg, B_arg, Q_tilde_arg,
                                fx_arg, Fu_arg, fu_arg, F_xTheta_arg, F_theta_arg, f_xTheta_arg, K_arg, s_arg);


    // ----------- SPECIFY arguments for step() ---------------
    // -------- they are updated at each time step ------------

    Lm_arg << 1,  2,  0,  1,  2,
          -2,  1.3,  1,  2,  2.3,
           1,  2, -1,  0,  -1,
           1,  2,  2,  -2.3,  1,
          0, 0,  2,  1.4, -2;

    Mm_arg << 1,  1.4,
       2,  -1,
       1,   2,
       0,   0,
       2, -1;

    tm_arg << -1, 2, 1, 1, 2;
    x_hat_arg << 3, 3, -2, 3, 4;

    srand((unsigned)time(0));

    // x_star_arg[0] << 0,   20.3181,   36.6838,   10.5584,  -24.6923;
    //  x_star_arg[1] << 401.2845,  262.2318,  -73.8211, -285.3312,  391.4877;
    //  x_star_arg[2] << -1466.9,    84.9,    189.8,    2850.6,   -1897.7;
    //  x_star_arg[3] << -854.2,    9297.6,    7092.0,   -1534.6,    4692.6;

    for(int i = 0; i ≤ N-1; i++)
    {
        x_star_arg[i].setZero();
        // x_star_arg[i] = 100 * x_star_arg[i];
    }

    // the following matrices are only needed for test purposes
    Matrix<double, 10*(2+5+5)+2,1> z_warm_arg;
    // requires phaseI

    /*
    z_warm_arg << 100, 100, 34, -98, 56, 45, 76, -30, 100, 65,
            1, 1, 0, 3, 1, 1, 1, 109, 12, 109,
            -89, 2, 3, 4, 0, 4, 1, 143, 2, 3, 1, 99,
            1, -45, 1, 1, 0, 4, 76, 45,
            1, 1, -54, 1, 1, 2, 1, 2, 6, 65;
    */

    z_warm_arg.setZero();
    z_warm_arg.setRandom();
    z_warm_arg = 100 * z_warm_arg;

    z_warm_arg <<

       0.001148698843107,
      -0.003290273495052,
       0.014272324367042,
       0.010793518327422,
      -0.007348884485118,
      -0.000233216192524,
      -0.007649617113494,
       0.004891975250732,
       0.000615548481374,
      -0.003152822400944,
      -0.003333216869256,
       0.001072413040433,
       0.003833892146796,
      -0.019961724439209,
      -0.029865210528105,
      -0.041848746265699,
       0.018138543769365,
       0.021358464663653,
       0.007405491880131,
      -0.011376898359488,
      -0.033235220205083,
      -0.023409841130554,
      -0.024393119983617,
       0.009997345729258,
       0.045594578261493,
       0.013524104753542,
       0.009852805310769,
      -0.020735116004147,
      -0.012521330238939,
      -0.073980771383218,
      -0.013352914756635,
      -0.074034130703376,
      -0.078936838025629,
      -0.017815939109988,
      -0.029063094849403,
      -0.118354733656974,
      -0.155384644866402,
       0.038789055733375,
      -0.052168759279625,
      -0.048984884002702,
       0.052468308976440,
      -0.000066243160052,
       0.004360060377921,
      -0.158827429659145,
      -0.084195671184460,
       0.061418626029024,
       0.111889264622883,
       0.030386312443187,
      -0.398719143216451,
      -0.030636540041834,
      -0.005646089547023,
      -0.000914622900134,
       0.006872441384278,
       0.021164679572702,
       0.005404322059116,
       0.081602987806510,
      -0.001881213023226,
      -0.058222744768716,
       0.133013683766810,
       0.156372126707285,
      -0.115960910186667,
      -0.085285586842392,
       0.016812108094816,
       0.011358414146085,
      -0.015524990879240,
      -0.004324400552158,
      -0.000461315326008,
       0.314950288436634,
       0.039936273225487,
      -0.101728610682870,
      -0.134161656271067,
       0.003637359135334,
       0.377382040475463,
       0.053192528545818,
       0.002117648325789,
      -0.001964333259952,
      -0.001279454128877,
      -0.005932747923164,
      -0.000298529665461,
       0.029854992692259,
      -0.011253489242809,
       0.161275610582213,
      -0.212149242219061,
      -0.245627604335095,
      -0.039162879020905,
       0.133718465030881,
      -0.002959704010831,
      -0.003966126683653,
       0.003572057150086,
       0.001203044284504,
       0.001066612801698,
      -0.291113682472646,
       0.025084051149883,
       0.331333022510823,
       0.232658601025521,
       0.190702393675488,
      -0.354137060423643,
      -0.260774482348422,
       0.000333755007832,
      -0.000380113321504,
      -0.000101316085314,
       0.001539368589362,
       0.000597144738049,
       0.597153215441143,
       0.419743818653965,
      -0.338610881138880,
       0.078132285658655,
       0.719974086113835,
       1.976779842160321,
      -0.108882097559082,
      -0.000382698488347,
       0.000007961296669,
      -0.000079369520373,
       0.000380307450810,
       0.000101962535505,
       1.009999919736511,
       0.702657486242329,
      -0.411041989290833,
      -1.009999922315370,
      -1.009999929359784,
      -0.778591128700403,
    1.143068816052561;

    z_warm_arg = 100*z_warm_arg;
    z_warm_arg.setConstant(-100);
    // cout << "z_warm_arg: " << endl << z_warm_arg << endl << endl;

    /*
     // no phaseI required
    z_warm_arg << -3, -3, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,
            3, 1, 1, 1, 1, 1, 1, 1, 2, 3, 4, 0, 4,
            1, 1, 2, 3, 1, 1, 1, 2, 1, 1, 0, 4, 1,
            1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1;
    */

    u_opt = myObj.step( Lm_arg, Mm_arg, tm_arg, x_hat_arg, z_warm_arg, x_star_arg);
```

```
    cout << "optimal input:" << endl << u_opt << endl << endl;

    return 0;

}
```

Literatur

[1] A. Aswani, H. Gonzalez, S.S. Sastry and C. Tomlin, *Provable Safe and Robust Learning-Based Model Predictive Control*, Submitted, 2011.

[2] Y. Wang and S. Boyd, *Fast Model Predictive Control Using Online Optimization*, IEEE Transactions on Control Systems Technology, Vol. 18, No. 2, March 2010.