Learning Based MPC

Xiaojing Zhang

Documentation for `LBmpcTP` template class - Version: PD IIPM
Implementation using primal-dual infeasible interior point method (PD IIPM)
Department of Electrical Engineering and Computer Sciences (EECS), UC Berkeley

5. November 2011

Introduction

The aim of this report is to give a brief introduction to the `LBmpcTP` template class. This documentation describes the primal-dual infeasible interior point method (**PD IIPM**) based on Mehrotra's predictor-corrector algorithm [1]. Our solver is taylored to the learning based MPC algorithm described in [2] with quadratic cost and affine oracle dynamics, (1).

There is another implementation of the `LBmpcTP` template class, that uses a primal barrier infeasible start interior point method (*PB IIPM*). In general, it is known that PD IIPM outperforms PB IIPM (for example [3, 4]). This also coincides with our experience.

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 [2]. In our particular framework, the cost is assumed to be quadratic and the oracle dynamics 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:

$$\min_{c[\cdot],\theta} \quad (\tilde{x}[m+N]-x^\star[m+N])^T\tilde{Q}_f(\tilde{x}[m+N]-x^\star[m+N])+ \tag{1}$$
$$\sum_{i=0}^{N-1}\Big\{(\tilde{x}[m+i]-x^\star[m+i])^T\tilde{Q}(\tilde{x}_[m+i]-x^\star[m+i])+(\check{u}[m+i]-u^\star[m+i])^T R(\check{u}[m+i]-u^\star[m+i])\Big\}$$

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

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

Tabelle 1: Assumptions on matrices.

$\tilde{Q}$	positive definite
$\tilde{Q}_f$	positive definite
R	positive definite
$F_{\bar{x}[m+i]}$	full rank
$F_{\check{u}[m+i]}$	full rank
F_θ	full rank
K	$A + BK$ Schur

To solve (1), the optimization problem (1) is casted into a standard convex quadratic program (QP) form:

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

where z is the stacked vector:

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

The above QP is convex. Hence, the KKT-conditions for optimality (necessary and sufficient) are that for every optimal z_{opt} there exists vectors λ_{opt}, ν_{opt} and t_{opt} such that at the optimal point $(z, \lambda, \nu, t) = (z_{opt}, \lambda_{opt}, \nu_{opt}, t_{opt})$ the following equations are satisfied ([5]):

$$\mathcal{F}(z, \lambda, \nu, t) \triangleq \begin{pmatrix} r_H \\ r_C \\ r_P \\ r_T \end{pmatrix} \triangleq \begin{pmatrix} 2Hz + g + P^T\lambda + C^T\nu \\ Cz - b \\ Pz - h + t \\ T\Lambda\mathbf{1} \end{pmatrix} = 0, \qquad (\lambda, t) \ge 0 \tag{3}$$

where t is the slack variable associated with the inequality, $T \triangleq \text{diag}(t)$, $\Lambda \triangleq \text{diag}(\lambda)$ and $\mathbf{1}$ is the all-one vector. Our PD IIPM algorithm generates sequences $(z^i, \lambda^i, \nu^i, t^i)$ with $(\lambda^i, t^i) > 0$ that approach the optimality condition (3).

Let us also define the complementary measure μ, which is a measure of optimality for the point (z, λ, ν, t):

$$\mu \triangleq \frac{\lambda^T t}{m_P}, \tag{4}$$

where m_P is the number of inequality equations, i.e. the number of rows in the Matrix P.

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, $\{F_{\bar{x}[m+i]}\}_i$, $\{f_{\bar{x}[m+i]}\}_i$, $\{F_{\check{u}[m+i]}\}_i$, $\{f_{\check{u}[m+i]}\}_i$, $F_{x\theta}$, F_θ, $f_{x\theta}$ and scalars such as n_{iter}, ϵ_{reg}, ϵ_H, ϵ_C, ϵ_P, ϵ_μ in MATLAB file `Init.m`. These values are written to a binary file, which is called `ConstrParam.bin` by default. The complete list of variables to be specified can be found in Tab. 2.

2. A C++-file (e.g. `mainLBmpcTP.cpp`) then reads the binary file `ConstrParam.bin`. `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: $L_m,\ M_m,\ t_m,\ \hat{x},\ \{x^\star[m+i]\}_i$.

The files 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, `Init.m` consists of two parts:

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

In appendix 4.1, a typical implementation of the `Init.m` file is shown.

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, and denoted by `_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, and denoted by `_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 4.2. 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 (2) by calling the step-function with the updated variables above, i.e. `myObj.step(.)`, and obtain the optimal input `u_opt`.

Since `LBmpcTP` is a template class and makes use of the linear algebra template class `Eigen`, some template parameters are of the form `const int`. 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` and `_pos_omega`.
2. `mainLBmpcTP.cpp` reads the parameters (with `_arg` appended) listed in Tab. 2 from the binary source file `ConstrParam.bin`.

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, $i = 1, \ldots, N$	
`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, $i = 0, \ldots, N-1$	
`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`	$F_{x\theta}\bar{x}[m+i] + F_\theta\theta \leq f_{x\theta}$	
`n_iter`	max. number of Newton steps to solve (2)	$[50, 200]$
`reg`	regularization coefficient to render Matrix H in (2) positive definite	$[0, 0.1]$, depends on H
`resNorm_H`	ϵ_H, necessary stopping criteria for $\|r_H\|$	0.01
`resNorm_C`	ϵ_C,necessary stopping criteria for $\|r_C\|$	0.01
`resNorm_P`	ϵ_P,necessary stopping criteria for $\|r_P\|$	0.01
`muNorm`	ϵ_μ, necessary stopping criteria for μ	0.01

3. It calls the constructor `LBmpcTP<...>::LBmpcTP(...)` and instantiates an object, e.g. `myObj`.
4. The oracle matrices `Lm_arg`, `Mm_arg`, `tm_arg`, `x_hat_arg`, `x_star_arg[_N]` are updated. Note: Computation of these values is not provided by the `LBmpcTP` template class.
5. The optimization problem (1) is then solved with the updated oracle matrices by evoking the member function `Matrix<Type,_m,1> LBmpcTP<...>::step(...)`, i.e. `myObj.step(...)`.

It should be noticed that the parameters in the MATLAB file `Init.m` and the C++ file `mainLBmpcTP.cpp` have to be consistent with each other.

Tabelle 3: Key parameters in `mainLBmpcTP.cpp`

variable	description	default
`Type`	only `double` is supported	`double`
`_N`	length of MPC horizon	
`_m`	number of inputs	
`_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[_N]`	array of desired states in cost function (1)	
`u_opt`	optimal input, return argument of `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 granted through two methods, the constructor and the `step(.)` method. Details on the underlying mathematics can be found in [1, 3, 5, 6]. The constructor initializes some of the private variables as discussed in the previous sections. The `step(.)`-method performs the following tasks:

- We recursively compute the sequence $\{u^\star[m+i]\}_i$ from the given desired state sequence $\{x^\star[m+i]\}_i$ by solving

$$x^\star[m+i] = (A + L_m)x^\star[m+i-1] + (B + M_m)u^\star[m+i-1] + (s + t_m)$$

  and taking the least-squared solution (SVD).
- Cast (1) into (2).
- Finally, it returns the computed optimal control input.

3 Tuning

Some of the parameters given in Tab. 2 can be used to tweak the `LBmpcTP` template class if the algorithm does not work as desired:

- The problem cannot be solved with the default parameters. It either does not converge (number of iterations exceeds `num_iter`) or the code returns `nan`.
- Convergence is too slow for the desired purpose, i.e. the optimization step needs too many Newton iterations.
- 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
`n_iter`	Can be used to obtain an inaccurate solution of (2) if for example computational time is limited	$[50, 200]$
`reg`	regularization coefficient to render Matrix H in (2) positive definite, see (6)	$[0, 0.1]$, depends on H
`resNorm_H`	ϵ_H, necessary stopping criteria for $\|r_H\|$, see (5)	0.01
`resNorm_C`	ϵ_C,necessary stopping criteria for $\|r_C\|$, see (5)	0.01
`resNorm_P`	ϵ_P,necessary stopping criteria for $\|r_P\|$, see (5)	0.01
`muNorm`	ϵ_μ, necessary stopping criteria for μ, see (5)	0.01

3.1 Compiling

Some compilers provide the option to generate optimized executable codes. 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 (more than 1 min) and more memory (more than 1.5 GB RAM) usage: `g++ -I /usr/local/include/eigen3/ -O3 mainLBmpcTP.cpp -o mainLBmpcTP`

3.2 Stopping Criteria and Regularization

This section addresses how and when the iterations are terminated. In this Algorithm, a simple stopping criteria suggested in [4, 7, 8] is used. It consists of the following four conditions:

$$\begin{aligned} \|r_H\| &\leq \epsilon_H \\ \|r_C\| &\leq \epsilon_C \\ \|r_P\| &\leq \epsilon_P \\ \mu &\leq \epsilon_\mu \end{aligned} \tag{5}$$

Our algorithm terminates if all four conditions are satisfied. It should be mentioned in this place that the smaller we choose $\epsilon_{\dots}$, the badly conditioned our problem becomes. Because the matrix H in (1) is not strictly convex, numerical issues arise as we try to push the residua in (3) towards zero: the normal equation becomes badly conditioned, posing challenges when computing the Cholesky decomposition numerically. For that reason, we have introduced a regularization parameter ϵ_{reg} (`reg`) that regularizes our problem to:

$$H_{reg} = H + \epsilon_{reg}\mathbb{I}, \tag{6}$$

where $\mathbb{I}$ is the identity matrix.

3.3 Troubleshooting

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

1. **Many Newton steps ($\gtrapprox 200$) are required to solve (2).**
   Solutions:

   - Many Newton steps with tiny step sizes are perfomed. Since this is waste of computational time, the number of Newton iterations can be upperbounded by choosing a smaller `n_iter`. Depending on the problem setup, numbers as few as 10 iterations might be enough to produce usable results.

2. **Obtained result is a `nan`-vector (not a number).**
   Solutions:

   - This problem typically shows up after the residua in (3) have become small, especially when z_{opt} lies on some face of the feasible set. Even though positive definiteness (and hence the existence 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 in (2) is regularized according to (6). Thus, choosing a larger `reg` usually avoids this problem, but may return inferior results. Alternatively, we may want to increase the different $\epsilon_{\dots}$ in (5).

3.4 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 how to choose the regularization and how to initialize the starting points $(z^0, \lambda^0, \nu^0, t^0)$. However, it usually suffices to tune 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 some preallocated arrays of the LLT-class must be increased to: `L_diag[2*_N]`, `LOmicron_diag[_N+1]`, `LPi_diag[_N]`, `LRho_diag[_N]`.
- When an `LBmpcTP` object is instantiated, the class variable `z` is initialized. Between the time steps, `z` can take the role of "warm start". However, when the `LBmpcTP` is instantiated, it set as the 0-vector. If a priori information is available, then a more suitable `z` can be chosen.

4 Example Files

4.1 Init.m

```
%% Init.m
% Writes relevant data to binary file.
% author: Xiaojing ZHANG
% date: October 28, 2011


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

%% MPC parameters:
N = 4;      % MPC horizon
m = 2;      % # input
n = 5;      % # states

%% Parameters for constructor

n_iter = 200; % maximum number of Newton iterations
reg = 1e-4;  % regularization Term
resNorm_H = 0.1;
resNorm_C = 0.1;
resNorm_P = 0.1;
muNorm = 0.1;

%% 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 = 1*eye(n);
Q_tilde_f = Q_tilde+1;

R = 1*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]';

%% write data for constructor arguments into file
% ConstrParam.bin

writeParam;      % call writeParam.m

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

4.2 mainLBmpcTP.cpp

```cpp
// mainLBmpcTP.cpp
// example file to test simple examples
// date: October 28, 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 -------
    const int _N = 4;       // MPC horizon
    const int _m = 2;       // #input
    const int _n = 5;       // #states
    const int _nSt = 10;    // # state constraints
    const int _nInp = 4;    // # state constraints
    const int _nF_xTheta = 10;  // # Omega constraints
    const int _pos_omega = 4;   // ≤ _N

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

    Matrix<double, _nF_xTheta, _n> F_xTheta_arg;    // _nF_xTheta x n
    Matrix<double, _nF_xTheta, _m> F_theta_arg; // _nF_xTheta x m
    Matrix<double, _nF_xTheta, 1> f_xTheta_arg; // _nF_xTheta x 1

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


    // ---------- no changes necessary ---------
    int n_iter_arg;
    double reg_arg;     // regularization term for PhaseII
    double resNorm_H_arg;
    double resNorm_C_arg;
    double resNorm_P_arg;
    double muNorm_arg;
```

```
    // ---------------- 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 *) &n_iter_arg, sizeof(int));
    fin.read((char *) &reg_arg, sizeof(double));
    fin.read((char *) &resNorm_H_arg, sizeof(double));
    fin.read((char *) &resNorm_C_arg, sizeof(double));
    fin.read((char *) &resNorm_P_arg, sizeof(double));
    fin.read((char *) &muNorm_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 << "n_iter_arg: " << n_iter_arg << endl << endl;
    cout << "reg_arg: " << reg_arg << endl << endl;
    cout << "resNorm_H_arg: " << resNorm_H_arg << endl << endl;
    cout << "resNorm_C_arg: " << resNorm_C_arg << endl << endl;
    cout << "resNorm_P_arg: " << resNorm_P_arg << endl << endl;
    cout << "muNorm_arg: " << muNorm_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;
    */

    // -------- object instantiation -----------
     LBmpcTP<double, _n, _m, _N, _nSt, _nInp, _nF_xTheta, _pos_omega> myObj(                // constructor
                    n_iter_arg, reg_arg, resNorm_H_arg, resNorm_C_arg, resNorm_P_arg, muNorm_arg,
                    A_arg, B_arg, Q_tilde_arg, Q_tilde_f_arg, R_arg, Fx_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;

    for(int i = 0; i <= _N-1; i++)
    {
        x_star_arg[i].setZero();
    }
    x_star_arg[0] << 29.1867 ,  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] << -1.4669*1000  ,  0.0849*1000 ,   0.1898*1000 ,   2.8506*1000 ,  -1.8977*1000;
    x_star_arg[3] << -0.8542*1000  ,  9.2976*1000   , 7.0920*1000 ,  -1.5346*1000 ,   4.6926*1000;

    u_opt = myObj.step( Lm_arg, Mm_arg, tm_arg, x_hat_arg, x_star_arg);
    cout << "optimal input:" << endl << u_opt << endl << endl;

    return 0;
}
```

Literatur

[1] S. Mehrotra, “On the implementation of a primal-dual interior point method,” *SIAM Journal on Optimization*, vol. 2, no. 4, pp. 575–601, 1992.

[2] A. Aswani, H. Gonzalez, S. S. Sastry, and C. Tomlin, “Provable safe and robust learning-based model predictive control,” Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Tech. Rep., 2011.

[3] F. A. Potra and S. J. Wright, “Interior-point methods,” *Journal of Computational and Applied Mathematics*, vol. 124, no. 1-2, pp. 281 – 302, 2000, ¡ce:title¿Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations¡/ce:title¿. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377042700004337

[4] S. Boyd and L. Vandenberghe, *Convex Optimization*, 1st ed. Cambridge University Press, 2004.

[5] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed., ser. Springer Series in Operation Research and Financial Engineering. Springer, April 2000.

[6] C. V. Rao, S. J. Wright, and J. B. Rawlings, “Application of interior-point methods to model predictive control,” *Journal of Optimization Theory and Applications*, vol. 99, pp. 723–757, 1998.

[7] J. B. Jørgensen, “Quadratic optimization, primal-dual interior-point algorithm,” Lecture notes for course 02611: Optimization Algorithms and Data-Fitting, November 2006.

[8] T. R. Krüth, “Interior-point algorithms for quadratic programming,” Master’s thesis, Technical University of Denmark, 2008.