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

November 10, 2011

Introduction

This report introduces the `LBmpcTP` template class which implements the primal-dual infeasible interior point method (**PD IIPM**) based on Mehrotra's predictor-corrector algorithm [1]. Our solver is tailored to learning-based model predictive control (LBMPC) [2] for the special case where the costs are quadratic and all involved dynamics are linear.

The report is structured as follows: First, LBMPC for the case where all dynamics are linear is introduced. Second, the interface to the `LBmpcTP` class is presented. In the third section, advanced parameters for configuration and fine-tuning are described.

1 The Learning-Based MPC model

We consider a special case of LBMPC [2], in which all dynamics are linear and the cost quadratic. Furthermore, we consider the situation where the feasible sets are convex polyhedron.

This instance of LBMPC is given by the following optimization problem:

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

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

We assume that the conditions given in table 1 hold.

Table 1: We make the following assumptions for (1).

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

For the purpose of solving this LBMPC problem, it is useful to consider the optimization problem as belonging to a general class of problems, in this case a convex quadratic program (QP):

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

where z is the stacked vector:

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

It can be shown that the above QP is convex. Furthermore, we assume that Slater's condition [3, Ch. 2] holds. Hence, the KKT-conditions for optimality are necessary and sufficient and state 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 [4]:

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

where t is the slack variable associated with the inequality in (2), $T \triangleq \text{diag}(t)$, $\Lambda \triangleq \text{diag}(\lambda)$ 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).

Because the PD IIPM algorithm is not guaranteed to generate feasible iterates (except in the limit as the algorithm converges), a duality gap cannot be defined. Instead, the complementary measure μ is used to measure the optimality of 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 $A,\ B,\ s,\ \tilde{Q},\ \tilde{Q}_f,\ R,\ K,\ \{F_{\bar{x},i}\}_i,\ \{f_{\bar{x},i}\}_i,\ \{F_{\check{u},i}\}_i,\ \{f_{\check{u},i}\}_i,\ F_{x\theta},\ F_\theta,\ f_{x\theta}$, scalars $n_{iter},\ \epsilon_{reg},\ \epsilon_H,\ \epsilon_C,\ \epsilon_P,\ \epsilon_\mu$ and `fileName` for binary file in MATLAB file `Init.m`. The matrices and scalars are written to a binary file, whose name is specified by `fileName` (by default `ConstrParam.bin`). The complete list of variables to be specified can be found in table 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`. For example:
   `LBmpcTP<double, _n, _m, _N, _nSt, _nInp, _nF_xTheta, _pos_omega> myObj( fileName, verbose)`

   (b) It calls the step-function `myObj.step(.)` which computes the optimal input and returns a status flag. The optimal input is stored in the public variable `u_opt`. Each call of the step-function requires the following (updated) parameters: $L_m,\ M_m,\ t_m,\ \hat{x},\ \{x^\star[m+i]\}_i$.
   ```
   status = myObj.step( Lm, Mm, tm, x_hat, x_star );
   u_opt = myObj.u_opt;
   ```

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 table 2.
- A binary file (default: `ConstrParam.bin`) with the parameters in table 2 is created automatically 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. In the following, we outline the typical steps in `mainLBmpcTP.cpp`:

1. **SPECIFY parameters:** `_N`, `_m`, `_n`, `_nSt`, `_nInp`, `_nF_xTheta`, `_pos_omega` (see table 3).

2. **SPECIFY binary source file name and verbose-flag:** `fileName`, `verbose`:
   ```
   const char fileName[] = "ConstrParam.bin";
   bool verbose = 0;    // 0 = shut up
   ```

Table 2: Key parameters which are to be defined in `Init.m`

MATLAB variable	description	typical range/value
`N`	length of MPC horizon	
`m`	number of inputs	
`n`	number of states	
`fileName`	name of binary file that stores the matrices and scalars defined below.	
`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`$\{i\}$	$F_{\bar{x},i}\bar{x}[m+i\|m] \leq f_{\bar{x},i}$, full-rank, $i = 1, \ldots, N$	
`fx`$\{i\}$	$F_{\bar{x},i}\bar{x}[m+i\|m] \leq f_{\bar{x},i}$	
`Fu`$\{i\}$	$F_{\check{u},i}\check{u}[m+i\|m] \leq f_{\check{u},i}$, full-rank, $i = 0, \ldots, N-1$	
`fu`$\{i\}$	$F_{\check{u},i}\check{u}[m+i\|m] \leq f_{\check{u},i}$	
`F_xTheta`	$F_{x\theta}\bar{x}[m+j\|m] + F_\theta\theta \leq f_{x\theta}$	
`F_theta`	$F_{x\theta}\bar{x}[m+j\|m] + F_\theta\theta \leq f_{x\theta}$, full-rank	
`f_xTheta`	$F_{x\theta}\bar{x}[m+j\|m] + F_\theta\theta \leq f_{x\theta}$	
`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 $\\|r_H\\|$, (5)	0.01
`resNorm_C`	ϵ_C, necessary stopping criteria for $\\|r_C\\|$, (5)	0.01
`resNorm_P`	ϵ_P, necessary stopping criteria for $\\|r_P\\|$, (5)	0.01
`muNorm`	ϵ_μ, necessary stopping criteria for μ, (5)	0.01

Table 3: The following template parameters are required to instantiate a `LBmpcTP`-object.

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 j in $F_{x\theta}\bar{x}[m+j\|m] + F_\theta\theta \leq f_{x\theta}$	

3. Call the constructor and instantiate an object (e.g. `myObj`):
   `LBmpcTP<double,_n,_m,_N,_nSt,_nInp,_nF_xTheta,_pos_omega> myObj(fileName,verbose);`
4. Update the variables needed for step()-function: `Lm`, `Mm`, `tm`, `x_hat`, `x_star[_N]` (table 4). Note: these values are not provided in this framework.

Table 4: These arguments must be updated before each `step()`-call.

variable	description
`Lm`	oracle matrix, i.e. $\mathcal{O}_m(\tilde{x}[m+i\|m], \check{u}[m+i\|m]) = L_m\tilde{x}[m+i\|m] + M_m\check{u}[m+i\|m] + t_m$
`Mm`	oracle matrix, i.e. $\mathcal{O}_m(\tilde{x}[m+i\|m], \check{u}[m+i\|m]) = L_m\tilde{x}[m+i\|m] + M_m\check{u}[m+i\|m] + t_m$
`tm`	oracle matrix, i.e. $\mathcal{O}_m(\tilde{x}[m+i\|m], \check{u}[m+i\|m]) = L_m\tilde{x}[m+i\|m] + M_m\check{u}[m+i\|m] + t_m$
`x_hat`	current state estimate, i.e. $\tilde{x}[m] = \hat{x}[m], \quad \bar{x}[m] = \hat{x}[m]$
`x_star[_N]`	states our system wants to track

5. Call the step-function to solve the optimization problem:
   `status = myObj.step( Lm, Mm, tm, x_hat, x_star );`
   The meaning of the status-flag are given in table 5

Table 5: The meaning of the status-flags returned by `step()`-function.

status flag	meaning
0	success
1	too many iterations, stopping criterion $\|r_H\| \leq \epsilon_H$ not satisfied
2	too many iterations, stopping criterion $\|r_C\| \leq \epsilon_C$ not satisfied
3	too many iterations, stopping criterion $\|r_P\| \leq \epsilon_P$ not satisfied
4	too many iterations, stopping criterion $\mu \leq \epsilon_\mu$ not satisfied
5	`nan`
6	other error

6. The optimal input can be accessed by: `u_opt = myObj.u_opt;`

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.

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, 4–6]. 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 $\{u^\star[m+i|m]\}_i$ from the given desired state sequence $\{x^\star[m+i|m]\}_i$ by solving

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

  and taking the least-squared solution (SVD).
- Cast (1) into (2).
- Finally, it computes the optimal control input and stores it in the public member variable `myObj.u_opt`.

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

Table 6: Tuning parameters defined in `Init.m`

tuning variable	influence	typical range/value
`n_iter`	Can be used to limit the number of Newton iterations or for early termination to obtain an inaccurate solution of (2). This can be useful if the 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

Note that in order to use (and compile) the `LBmpcTP` template library, the EIGEN[*] library has to be installed.

Furthermore, 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 code:

```
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 [7–9] is used. It consists of the following four conditions:

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

[*]`http://eigen.tuxfamily.org/`

Our algorithm terminates if all four conditions are satisfied. It should be mentioned in this place that the smaller we choose the various ϵ to be, the badly conditioned our problem becomes. A forteriori, this is true for ϵ_{μ}. Because the matrix H in (1) is not strictly convex, numerical issues arise as we try to push the residuals (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 satisfying results.

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

   - This problem typically shows up after the residuals (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 ϵ 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 Table (6).
- 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

%% binary file name
fileName = 'ConstrParam.bin';

%% Parameters for constructor

n_iter = 200; % maximum number of Newton iterations
reg = 1e-3;  % 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

```
// mainLBmpcTP.cpp
// example file to test simple examples
// date: November 08, 2011
// author: Xiaojing ZHANG

// matrices are imported from binary file created by MATLAB


#include <iostream>     // I-O
#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
    const char fileName[] = "ConstrParam.bin";
    bool verbose = 0;   // '0' if it should shut up
    int status;     // stores status code

    // -------- object instantiation -----------
    // fileName contains name of file with parameters
    // bool verbose: 1 or 0
    LBmpcTP<double, _n, _m, _N, _nSt, _nInp, _nF_xTheta, _pos_omega> myObj( fileName, verbose);


    // ----------- SPECIFY arguments for step() ---------------
    // those matrices are computed externely
    // -----------  sizes of matrices ---------------
        Matrix<double, _n, _n> Lm;      // n x n
        Matrix<double, _n, _m> Mm;      // n x m
        Matrix<double, _n, 1> tm;       // n x 1
        Matrix<double, _n, 1> x_hat;        // n x 1, state estimate
        Matrix<double, _n, 1> x_star[_N];   // n x 1, [_N], tracking
        Matrix<double, _m, 1> u_opt;            // m x 1, optimal input is saved there

    // -------- they are updated at each time step ------------
    Lm << 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 << 1,  1.4,
       2,  -1,
       1,   2,
       0,   0,
       2, -1;

    tm << -1, 2, 1, 1, 2;
    x_hat << 3, 3, -2, 3, 4;

    for(int i = 0; i ≤ _N-1; i++)
    {
        x_star[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;

    status = myObj.step( Lm, Mm, tm, x_hat, x_star );
    if (!status)
    {
        u_opt = myObj.u_opt;
        cout << "optimal input:" << endl << u_opt << endl << endl;
    }

    return 0;
}
```

References

[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. Borrelli, A. Bemporad, and M. Morari, *Predictive Control for Linear and Hybrid Systems*. Cambridge University Press, 2011.

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

[5] 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.

[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.

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