Learning Based MPC

Xiaojing Zhang

Documentation for `LBmpcTP` template class - Version: PB IIPM

Implementation using primal barrier infeasible interior point method (PB 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, a particular implemention of the interior point method solver taylored to the learning based MPC algorithm described in [1] with quadratic cost and affine oracle dynamics.

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 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:

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

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

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

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

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:

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

To solve the QP, the 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 by a factor $\mu \in (0,1)$, i.e. $\{\kappa_0, \kappa_1, \kappa_2, \ldots\},\ \kappa_i \triangleq \mu^i \kappa_0$. 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 such as $A,\ B,\ s,\ \tilde{Q},\ \tilde{Q}_f,\ R,\ K,\ \{F_{\bar{x}[m+i]}\}_i,\ \{f_{\bar{x}[m+i]}\}_i,\ \{F_{\check{u}[m+i]}\}_i,\ \{f_{\check{u}[m+i]}\}_i$ and scalars such as $\epsilon_{barrier},\ \epsilon_{nt},\ \epsilon_{rp},\ \alpha,\ \beta,\ \kappa_{0,I},\ \kappa_{0,II},$ 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 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, `Init.m` consists of 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.

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 `LBmpcTP` is a template class and makes use of the linear algebra template class `Eigen`, some template parameters must be adjusted 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` and `_pos_omega`.
2. `mainLBmpcTP.cpp` reads the parameters (with `_arg` appended) listed in Tab. 2 from the binary source file `ConstrParam.bin`.
3. 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.
4. The optimization problem (1) is then solved with the updated oracle matrices by calling the member function `myObj.step(.)`.

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

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`	$F_{x\theta}\bar{x}[m+i] + F_\theta\theta \leq f_{x\theta}$	
`kappa_start_PhaseI`	Phase I starting value of $\kappa = \kappa_0$ to find $z_0 : \ Pz_0 < h$	100
`kappa_start_PhaseII`	Phase II starting value of $\kappa = \kappa_0$ 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. $z^+ = z + t \cdot \Delta z$	$1e-7$
`n_iter_PhaseI`	max. number of Newton steps allowed to solve (3) for fixed κ, (does not apply to the first step, i.e. $\kappa =$ `kappa_start_PhaseII`, tbd)	$[5, 50]$
`n_iter_PhaseII`	similar to `n_iter_PhaseI`, just for Phase I.	$[10, 50]$
`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	$[0.001, 1]$
`weight_PhaseI`	weight on the linear cost in PhaseI	$1e3$

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`	stores the optimal input from `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:

- 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).
- If the warm start variable `z_warm` does not satisfy $P \cdot \texttt{z\_warm} < h$, then the method `PhaseI()` computes a new `z_warm` which satisfies the above (strict) inequality, see App. A.
- Solve (2) by solving a sequence of (3). It can be shown that as κ approaches 0, the solution of (3) will converge to the solution of (2).
- Finally, it returns the control input based on the `eps_barrier`-suboptimal solution or if the number of Newton iterations exeeds `n_iter`.

3 Hidden Parameters, Tweaks

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 or the code returns `nan`.
- Convergence is too slow for the desired purpose, i.e. the optimization step needs too many Newton or backtracking 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.

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 2 min) and more memory (more than 2 GB) 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 requires too many Newton steps.**
   Solutions:
   - Many Newton steps may indicate that the matrices are badly conditioned. Try to increase `kappa_start_PhaseI`, i.e. start with a larger κ_0.
   - Try to decrease the value of `reg_PhaseI` to give the quadratic cost less weight.
   - Increase `weight_PhaseI` to give the linear cost (original cost) more weight.
   - If still a lot of Newton steps are required increase `n_iter_PhaseI`.
2. **PhaseII: many Newton steps ($\gtrapprox 100$) are required to solve** (3), i.e. κ fixed.
   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 choosing a smaller `n_iter_PhaseII`. Depending on the problem setup, numbers as few as 5 iterations might be enough to produce usable results.
3. **PhaseI/PhaseII: obtained result is a `nan`-vector (not a number).**
   Solutions:
   - This problem typically shows up during Phase II 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 function in (3) is regularized using the weight term `reg_PhaseI` or `reg_PhaseII`. Thus, choosing a larger`reg_PhaseI`/`reg_PhaseII` usually overcomes this problem, but may return inferior results.
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 known approximately, then `eps_barrier` can be chosen accordingly.

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 larger 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 a 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_PhaseII`	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.	$[5, 50]$
`n_iter_PhaseI`	Similar idea as for PhaseII, typically larger than in PhaseI	$[20, 50]$
`alpha_ls`	typically 0.01	
`beta_ls`	If set very small, t decreases rapidly, thus missing the optimal t by far. This might lead to the need to perfom more Newton steps, which are expensive. However, if `beta_ls` ≈ 1, 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 in (3). If the Phase I algorithm does not run as anticipated, decreasing `reg_PhaseI`, which influences the quadratic term, can help. Also see App. A	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`.	$[0.001, 1]$
`weight_PhaseI`	Weight on the linear cost during the PhaseI. See App. **??**	1000

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 exactly to regularize the cost functions in PhaseI and PhaseII. 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 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`.
- When an `LBmpcTP` object is instantiated, a class variable called `z_warm` representing the "warm start"from one time step to another is created. By default, it is the 0-vector. However, this can be changed easily to a more appropriate value.

A Phase I

This section roughly describes how the Phase I algorithm works. Parts of the ideas are taken from [3]. We will provide a variation of the Phase I algorithm proposed there. We start by explaining Phase I and formulating the original problem setup. In the second section, the original problem setup is casted to fit our PhaseII framework.

Original Problem Formulation

Phase I is needed if the provided vector z does not satisfy $Pz < h$, a requirement arising from the log-barrier term in (3). Hence, the goal of Phase I is to find a z that satisfies the (strict) inequality $Pz < h$. Subsequently, the existence of such a point is assumed to exist. One way to compute such a point is to introduce a variable $s \in \mathbb{R}$ and solve the following linear program (LP) [3]:

$$\begin{aligned} \min_{z,s} \quad & s \\ \text{s.t.} \quad & Cz = b \\ & (Pz - h)_i \le s \quad \forall i \end{aligned}$$

Assuming the polyhedron is not empty, then we can find a $s < 0$. The above problem can be solved using a LP-solver. However, we aspire to transform the equation above to our framework to exploit the structure of the problem. Simple augmentation of z and s to $\hat{z}$ (in this case, the $\hat{}$ refers to Phase I and not to the state estimate) will not work because it will destroy the structure specific to our problem.

Modified Problem Setup

We introduce $2N + 1$ new variables $s_0, \ldots, s_{2N}$, where N is the prediction horizon. So, the aim is to solve the following optimization problem:

$$\begin{aligned} \min_{z,s} \quad & s_0 + s_{2N} \\ \text{s.t.} \quad & Cz = b \\ & s_0 = s_1 = \ldots = s_{2N-1} \\ & (Pz - h)_j \le s_i, \qquad \forall j \in \mathcal{J}_i, \forall i, \end{aligned}$$

where $\mathcal{J}_i \subset \mathbb{N}$ is some set. We augment the vector z by $s \in \mathbb{R}^{2N+1}$ in a specific way and denote the augmented vector by $\hat{z}$. The new optimization problem is

$$\begin{aligned} \min_{\hat{z}} \quad & s_0 + s_{2N} \\ \text{s.t.} \quad & \hat{C}\hat{z} = \hat{b} \\ & \hat{P}\hat{z} - h \le \hat{s}. \end{aligned} \tag{4}$$

The $\hat{}$-matrices shall not be defined in more detail. To use our algorithm, we regularize (4) by adding a small quadratic term and get:

$$\begin{aligned} \min_{\hat{z}} \quad & \epsilon_{II} \hat{z}^T \hat{z} + w_{II} \cdot \hat{g}\hat{z} \\ \text{s.t.} \quad & \hat{C}\hat{z} = \hat{b} \\ & \hat{P}\hat{z} - h \le \hat{s}. \end{aligned} \tag{5}$$

The parameters ϵ_{II} and w_{II} correspond to `reg_PhaseI` and `weight_PhaseI` in Tab. 4, respectively. Since (5) corresponds to (2) and both problems have similar structures, the same interior point algorithm as in Phase II can be used.
Problems:

- Regularization does not guarantee that our the solution of the regularized QP will converge to the solution of original LP, hence making (5) a heuristic algorithm.

B Example Files

B.1 `Init.m`

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

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

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

%% Parameters for constructor
kappa_start_PhaseI = 100;  % barrier parameter for PhaseI - can be as high as 1e10
kappa_start_PhaseII = 1e9;  % 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_PhaseI = 30; % maximum number of Newton iterations for fixed kappa in PhaseI
n_iter_PhaseII = 10;    % maximum number of 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 = 1e-6; % regularization Term in PhaseI
reg_PhaseII = 1e-2;  % regularization Term in PhaseII
weight_PhaseI = 1e3;    % weight for linear cost (i.e. the original PhaseI problem)


%% 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']);
```

B.2 mainLBmpcTP.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 = 10;      // 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 = 10;  // ≤ _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 ---------
    double kappa_arg;   // for PhaseII
    double kappa_PhaseI_arg;    // for PhaseI
    int n_iter_arg;
    int n_iter_PhaseI_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
    double weight_PhaseI_arg;   // weight for linear cost in 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_PhaseI_arg, sizeof(int));
    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));
    fin.read((char *) &weight_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 << "n_iter_PhaseI_arg: " << n_iter_PhaseI_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 << "weight_PhaseI_arg: " << weight_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;
     */

    // -------- object instantiation -----------
     LBmpcTP<double, _n, _m, _N, _nSt, _nInp, _nF_xTheta, _pos_omega> myObj(                // constructor
                            kappa_arg, kappa_PhaseI_arg, n_iter_arg, n_iter_PhaseI_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, weight_PhaseI_arg, A_arg, B_arg, Q-
                            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] 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.

[3] S. Boyd and L. Vandenberghe, *Convex Optimization*, Cambridge University Press, March 2004.