

# HPMPC regerence guide

Gianluca Frison

September 4, 2014

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Problems definition</b>	<b>3</b>
2.1	Linear-Quadratic Control Problem . . . . .	3
2.2	Linear MPC problem . . . . .	3
<b>3</b>	<b>Basic API</b>	<b>5</b>
3.1	Solvers for problem (2.1) . . . . .	5
3.1.1	xxx_order_dynamic_mem_riccati_wrapper_init() . . . . .	5
3.1.2	xxx_order_dynamic_mem_riccati_wrapper_free() . . . . .	6
3.1.3	xxx_order_dynamic_mem_riccati_wrapper_fact_solve() . . . . .	6
3.1.4	xxx_order_dynamic_mem_riccati_wrapper_solve() . . . . .	7
<b>4</b>	<b>References</b>	<b>9</b>

# Chapter 1

## Introduction

HPMPC – A library for High-Performance implementation of solvers for MPC.

HPMPC has been developed with the aim of providing extremely fast building blocks to implement algorithms for Model Predictive Control. This has been achieved by carefully implementing the linear algebra routines using high-performance computing techniques with a focus on small-medium scale problems, typical of MPC applications.

The current version of the library contains an interior-point (IP) solver for the linear MPC (LMPC) problem with box constraints, and a Riccati solver for the linear-quadratic control problem, that is used as a routine in the IP method. A basic API is provided for these solvers, while lower level interfaces provide access to kernels, linear-algebra routines and MPC solvers. The library is self-contained, not requiring any other library beside the standard C library.

The code is highly-optimized for a number of common architectures, plus a reference version in plain C code. The code is intended to be used on a Linux machine (it has been developed on this OS) and using gcc as a compiler (the code for some architecture makes use of extended asm inline assembly). The time-critical routines have been carefully implemented in assembly exploiting architecture-specific features, so there would be no practical advantage in compiling the code with more performing (read commercial) compilers.

The library is released under the LGPL version 3 licence, and it can be used as a basis for the development of higher-level solvers for MPC or other applications extremely fast solvers for small-medium scale problems.

## Chapter 2

# Problems definition

In this chapter the constrained and unconstrained MPC problems are presented.

### 2.1 Linear-Quadratic Control Problem

The Linear-Quadratic Control Problem (in the following, LQCP) is the equality constrained quadratic program

$$\begin{aligned} \min_{u_n, x_{n+1}} \quad & \phi = \sum_{n=0}^{N-1} \varphi_n(x_n, u_n) + \varphi_N(x_N) \\ \text{s.t.} \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \end{aligned} \quad (2.1)$$

where  $n \in \{0, 1, \dots, N-1\}$  and  $\varphi_n(x_n, u_n)$  and

$$\begin{aligned} \varphi_n(x_n, u_n) &= \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}' \begin{bmatrix} R_n & S_n & s_n \\ S_n' & Q_n & q_n \\ s_n' & q_n' & \rho_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} = \mathcal{X}_n' \mathcal{Q}_n \mathcal{X}_n \\ \varphi_N(x_N) &= \begin{bmatrix} u_N \\ x_N \\ 1 \end{bmatrix}' \begin{bmatrix} 0 & 0 & 0 \\ 0 & P & p \\ 0 & p' & \pi \end{bmatrix} \begin{bmatrix} u_N \\ x_N \\ 1 \end{bmatrix} = \mathcal{X}_N' \mathcal{P} \mathcal{X}_N \end{aligned} \quad (2.2)$$

All matrices in this formulation can in general be dense and time variant. The matrices  $\mathcal{Q}_n$  and  $\mathcal{P}$  have to be symmetric and positive semi-definite, while the  $R_n$  matrices have to be symmetric and strictly positive definite.

### 2.2 Linear MPC problem

The linear MPC problem with box constraints is the quadratic program

$$\begin{aligned} \min_{u_n, x_{n+1}} \quad & \phi = \sum_{n=0}^{N-1} \varphi_n(x_n, u_n) + \varphi_N(x_N) \\ \text{s.t.} \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \\ & \underline{u}_n \leq u_n \leq \bar{u}_n \\ & \underline{x}_n \leq x_n \leq \bar{x}_n \end{aligned} \quad (2.3)$$

where  $n \in \{0, 1, \dots, N - 1\}$ ,  $\varphi_n(x_n, u_n)$  and  $\varphi_N(x_N)$  are defined as in (2.2). Again, all matrices can in general be dense and time variant.

## Chapter 3

# Basic API

In this chapter the basic API is presented. It is intended to give access to the higher level routines in the library, namely the Riccati solver for the unconstrained problem (2.1), and the IP solver for the MPC problem (2.3).

For performance purposes, internally the library makes use of a packed matrix format, and routines are provided to convert to and from this packed format and standard column- and row-major orders.

In this basic API, all matrices are passed using standard column- or row-major orders, and wrappers take care of the conversions. For the best performance, users may consider to skip the wrappers and directly call the solvers using the packed matrix format.

Furthermore, the library internally does not allocate static nor dynamic memory, since all work space has to be supplied from the extern. Again, wrappers take care of this, but at some performance expense, especially if dynamic memory has to be allocated and deallocated at each solver call. Notice that static memory allocation needs the knowledge of the problem instance size, so the wrapper has to be recompiled for different sizes.

As a consequence, each wrapper comes in 4 variants, for all combinations of static or dynamic memory allocation and column- or row major order.

### 3.1 Solvers for problem (2.1)

The linear-quadratic control problem (2.1) is solved using a modified version of Riccati recursion, improving small-scale performance with respect to the classical version.

#### 3.1.1 `xxx_order_dynamic_mem_riccati_wrapper_init()`

where 'xxx' can be either 'c' (i.e. row-major) or 'fortran' (i.e. column-major) order.

This routine allocates and returns dynamic memory, and converts all matrices into packed format. It returns 0 if execution went fine.

Input arguments are, in the order

**const int nx** number of states

**const int nu** number of inputs

**const int N** control horizon lenght

**double \*A** pointer to an array of doubles containing the  $N$  matrices  $A_n$  (each of size  $n_x \times n_x$ ) stored one after the other.

**double \*B** pointer to an array of doubles containing the  $N$  matrices  $B_n$  (each of size  $n_x \times n_u$ ) stored one after the other.

**double \*b** pointer to an array of doubles containing the  $N$  vectors  $b_n$  (each of size  $n_x$ ) stored one after the other.

**double \*Q** pointer to an array of doubles containing the  $N$  matrices  $Q_n$  (each of size  $n_x \times n_x$ ) stored one after the other.

**double \*Qf** pointer to an array of doubles containing the matrix  $Q_f$  (of size  $n_x \times n_x$ ).

**double \*S** pointer to an array of doubles containing the  $N$  matrices  $S_n$  (each of size  $n_u \times n_x$ ) stored one after the other.

**double \*R** pointer to an array of doubles containing the  $N$  matrices  $R_n$  (each of size  $n_u \times n_u$ ) stored one after the other.

**double \*q** pointer to an array of doubles containing the  $N$  vectors  $q_n$  (each of size  $n_x$ ) stored one after the other.

**double \*qf** pointer to an array of doubles containing the vector  $q_f$  (of size  $n_x$ ).

**double \*r** pointer to an array of doubles containing the  $N$  vectors  $r_n$  (each of size  $n_u$ ) stored one after the other.

**double \*\*ptr\_work** address of the pointer to an array of doubles, used to return the memory dynamically allocated by the routine, and containing the packed matrices.

### 3.1.2 xxx\_order\_dynamic\_mem\_riccati\_wrapper\_free()

where 'xxx' can be either 'c' (i.e. row-major) or 'fortran' (i.e. column-major) order.

This routine frees the dynamic memory allocated by the init routine. It returns 0 if execution went fine.

Input arguments are, in the order

**double \*work** pointer to the memory to be deallocated.

### 3.1.3 xxx\_order\_dynamic\_mem\_riccati\_wrapper\_fact\_solve()

where 'xxx' can be either 'c' (i.e. row-major) or 'fortran' (i.e. column-major) order.

This routine factorizes the KKT matrix and solves the KKT system using a modified Riccati recursion. The data in packed matrix format has to be initialized beforehand by the init routine, and it is passed in the work space.

After execution, the data in the factorized KKT matrix is passed in the work space. It returns 0 if execution went fine.

Input arguments are, in the order

**const int nx** number of states

**const int nu** number of inputs

**const int N** control horizon lenght

**double \*x** pointer to an array of doubles containing the  $N + 1$  states vectors  $x_n$  (each of size  $n_x$ ) stored one after the other. On input,  $x_0$  must contain the initial state. On output, it contains the optimal states sequence.

**double \*u** pointer to an array of doubles containing the  $N$  inputs vectors  $u_n$  (each of size  $n_u$ ) stored one after the other. On output, it contains the optimal inputs sequence.

**double \*pi** pointer to an array of doubles containing the  $N + 1$  equality constraints Lagrangian multipliers vectors  $\pi_n$  (each of size  $n_x$ ) stored one after the other. On output, it contains the optimal multipliers sequence.

**double \*work** pointer to an array of doubles, as previously allocated and initialized by the init routine. On output, it contains information about the factorized KKT matrix.

### 3.1.4 xxx\_order\_dynamic\_mem\_riccati\_wrapper\_solve()

where 'xxx' can be either 'c' (i.e. row-major) or 'fortran' (i.e. column-major) order.

This routine solves the KKT system using a modified Riccati recursion, reusing a previous factorization of the KKT system. It is used to solve several KKT systems with the same LHS matrix, but different RHS vector, without factorizing the KKT matrix again, as e.g. in predictor corrector interior-point methods, or in mixed precision algorithms. The data in packed matrix format has to be initialized beforehand by the init routine, and the KKT matrix factorized by the fact routine, and both are passed in the work space. It returns 0 if execution went fine.

Input arguments are, in the order

**const int nx** number of states

**const int nu** number of inputs

**const int N** control horizon lenght

**double \*b** pointer to an array of doubles containing the  $N$  vectors  $b_n$  (each of size  $n_x$ ) stored one after the other.

**double \*q** pointer to an array of doubles containing the  $N$  vectors  $q_n$  (each of size  $n_x$ ) stored one after the other.

**double \*qf** pointer to an array of doubles containing the vector  $q_f$  (of size  $n_x$ ).



**double \*r** pointer to an array of doubles containing the  $N$  vectors  $r_n$  (each of size  $n_u$ ) stored one after the other.

**double \*x** pointer to an array of doubles containing the  $N + 1$  states vectors  $x_n$  (each of size  $n_x$ ) stored one after the other. On input,  $x_0$  must contain the initial state. On output, it contains the optimal states sequence.

**double \*u** pointer to an array of doubles containing the  $N$  inputs vectors  $u_n$  (each of size  $n_u$ ) stored one after the other. On output, it contains the optimal inputs sequence.

**double \*pi** pointer to an array of doubles containing the  $N + 1$  equality constraints Lagrangian multipliers vectors  $\pi_n$  (each of size  $n_x$ ) stored one after the other. On output, it contains the optimal multipliers sequence.

**double \*work** pointer to an array of doubles, as previously allocated and initialized by the init routine. On input, it must contain information about the factorized KKT matrix as given by the fact routine.

## Chapter 4

# References

The HPMPC library is the result of a long research path that led to a novel way to implement solvers for MPC, specially tailored to small-medium scale problems. The library has undergone several revisions before being published, and some of the steps along this research path are documented in the following papers.

- G. Frison, H.H.B. Sørensen, B. Dammann, J.B. Jørgensen, *High-Performance Small-Scale Solvers for Linear Model Predictive Control*, in proceedings of 13th European Control Conference, Strasbourg (France), 2014.
- G. Frison, L.E. Sokoler, J.B. Jørgensen, *A Family of High-Performance Solvers for Linear Model Predictive Control*, in proceedings of 19th IFAC World Congress, Cape Town (South Africa), 2014.