HPMPC regerence guide

Gianluca Frison

April 4, 2015

Contents

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 (MPC). 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 Interior-Point Method (IPM) and Alternating Direction Methods of Multipliers (ADMM) solvers for the linear MPC (LMPC) problem with box constraints, and a Riccati-based solver for the unconstrained MPC 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 has been developed on this Linux machines and tested also on MAC machines. The code is intended to be compiled using gcc as a compiler (the code for some architecture makes use of extended asm inline assembly); it may be compiled using clang as well. 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 (e.g. 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 requiring extremely fast solvers for small-medium scale problems.

Chapter 2

Problems definition

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

2.1 Linear-Quadratic Control Problem

The Linear-Quadratic Control Problem (in the following, LQCP) is a general formulation of unconstrained MPC problem. It is the equality constrained quadratic program

$$\begin{aligned} \min_{u_n, x_n} \quad & \phi = \sum_{n=0}^{N-1} \varphi_n(x_n, u_n) + \varphi_N(x_N) \\ s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \\ & x_0 = \overline{x}_0 \end{aligned} \tag{2.1}$$

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

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

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

$$\begin{aligned} \min_{u_n, x_n} \quad & \phi = \sum_{n=0}^{N-1} \varphi_n(x_n, u_n) + \varphi_N(x_N) \\ s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \\ & x_0 = \overline{x}_0 \\ & \underline{u}_n \leq u_n \leq \bar{u}_n \\ & \underline{x}_n \leq x_n \leq \bar{x}_n \end{aligned} \tag{2.3}$$

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

Chapter 3

Compilation and installation of the library

The code has been developed on Linux machines, using the gcc compiler. It has been also tested on MAC machines using the clang compiler. The library is mainly written in C code, with time-critical parts (kernels) written using intrinsics or extended asm inline assembly: from here the need for the gcc or clang compiler. The use of commercial compilers would not improve performance.

Time-critical routines are carefully optimized by hand for a number of architectures, using assembly to explicitly exploit e.g. the number of registers and SIMD instruction sets specific to each architecture.

`gcc` is usually part of all Linux distributions.

Redarding utilities, `make` is used to automate the compilation of the code, and `ar` to build the static library: both of them are usually part of all Linux distributions. Alternatively, the library can be built using `CMake`.

Once the code has been downloaded, the first step is the editing of the configuration file `Makefile.rule`. In general, the only part that needs to be edited is the TARGET, used to choose architeture-specific code and flags.

Currently supported targets are

X64_AVX2 this a recent x86_64 processor supporting AVX2 and FMA3 instruction sets (e.g. Intel Haswell architecture). The 64-bit version of the OS is required. At the moment, this architecture is supported, but only the double-precision version of the library if fully-optimized.

X64_AVX this is a x86_64 processor supporting the AVX instruction set (e.g. Intel Sandy Bridge and Ivy Bridge, AMD Bulldozer or more recent architectures). The 64-bit version of the OS is required.

X64_SSE3 this is a x86_64 processor supporting the SSE3 instruction set (e.g. Intel Pentium 4 Prescott, AMD Athlon 64 revision E or more recent architectures). The 64-bit version of the OS is required. The code is not fully optimized yet.

CORTEX_A15 this is a processor implementing the ARMv7A architecture with VFPv4 and NEONv2 instruction sets, code optimized for ARM Cortex A15.

CORTEX_A9 this is a processor implementing the ARMv7A architecture with VFPv3 and NEON instruction sets, code optimized for ARM Cortex A9.

CORTEX_A7 this is a processor implementing the ARMv7A architecture with VFPv4 and NEONv2 instruction sets, code optimized for ARM Cortex A7.

C99_4X4 this version is written entirely in C code and works with all machines, even if performing worse that machine-specific code. The code works better on a machine with at least 32 scalar FP registers.

More architectures are supported in the older version 0.1 of the library, that can still be downloaded from github.

The supported instruction sets can be easily found by googling the processor name, or by typing

```
less \proc\cpuinfo
```

on a terminal, and looking among the flags. In any case, even if the code is compiled for the wrong architecutre, an `Illegal instruction` error will be raised at run time, so this can be immediately discovered running a test problem.

Once the architecture has been chosen, the static library can be build by typing

```
make static
```

on a terminal. The dynamic library can be build by typing

```
make shared
```

on a terminal.

The static library and the headers can be installed by typing

```
sudo make install_static
```

on a terminal. The dynamic library and the headers can be installed by typing

```
sudo make install_shared
```

on a terminal. The installation forder can be changed by editing the Makefile file.

A number of test problems is available in the folder test_problems, and it is possible to choose between them by editing the file `test_problems/Makefile`. Notice that test problems are not part of the library. The chosen test problem is compiled by typing

```
make test_problem
```

on a terminal opened on the main HPMPC folder, and it is run by typing

```
make run
```

on a terminal opened on the main HPMPC folder.

Chapter 4

High-level API

In this chapter the high-level API is presented. It is intended to give accesso to the higher level routines in the library, namely the Riccati solver for the unconstrained problem (2.1), and the IPM 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 high-level 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 (low-level API).

Chapter 5

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.
- G. Frison, D.K.M. Kufualor, L. Imsland, J.B. Jørgensen, *Efficient Implementation of Solvers for Linear Model Predictive Control on Embedded Devices*, in proceedings of IEEE Multi-conference on Systems and Control, Antibes (France), 2014.
- G. Frison, J.B. Jørgensen, *MPC Related Computational Capabilities of ARMv7A Processors*, in proceedings of 14th European Control Conference, Linz (Austria), 2015.