Papers Collection

Gianluca Frison

30th December 2015

Contents

1 Paper [32] **2**

2 Paper [34] **9**

3 Paper [33] **16**

4 Paper [31] **23**

5 Paper [38] **30**

6 Paper [37] **37**

7 Abstract [35] **44**

8 Paper [36] **47**

9 Paper [39] **56**

Chapter 1

Paper [32]

2013 IEEE Multi-conference on Systems and Control
(Conference on Control Applications)
Hyderabad, India, August 28-30, 2013

Efficient Implementation of the Riccati Recursion for Solving Linear-Quadratic Control Problems

Gianluca Frison, John Bagterp Jørgensen

***Abstract*— In both Active-Set (AS) and Interior-Point (IP) algorithms for Model Predictive Control (MPC), sub-problems in the form of linear-quadratic (LQ) control problems need to be solved at each iteration. The solution of these sub-problems is typically the main computational effort at each iteration. In this paper, we compare a number of solvers for an extended formulation of the LQ control problem: a Riccati recursion based solver can be considered the best choice for the general problem with dense matrices. Furthermore, we present a novel version of the Riccati solver, that makes use of the Cholesky factorization of the P_n matrices to reduce the number of flops. When combined with regularization and mixed precision, this algorithm can solve large instances of the LQ control problem up to 3 times faster than the classical Riccati solver.**

I. Introduction

The linear-quadratic (LQ) control problem can be considered the core problem in Model Predictive Control (MPC). It represents an unconstrained optimal control problem where the controlled system is linear and the cost function is quadratic. This problem formulation is especially important because it arises as sub-problem in Active-Set (AS) and Interior-Point (IP) algorithms for MPC, where a problem of this form has to be solved at each iteration [1], [2]. The solution of these sub-problems is typically the main computational effort at each iteration, and this explain the need for efficient solvers.

The LQ control problem is a special instance of equality constrained quadratic program. The related KKT system is sparse and structured, and this structure can be exploited to implement more efficient solvers. We can distinguish two main approaches to the solution of this KKT system, that differ on the choice of the optimization variables.

The first approach considers as optimization variables the sole controls: by exploiting the dynamic system linear equation, the large, sparse KKT system is rewritten into a smaller, dense form. The reduced KKT system is typically solved by using the Cholesky factorization of the (positive definite) Hessian. The cost of this approach is $\mathcal{O}(N^3 n_u^3)$ (plus the cost of the condensing phase), and then suitable for problems with small N and n_u [3].

The second approach considers as optimization variables also the states: larger systems where the sparsity is preserved are solved. Well known examples are general purpose sparse solvers, Riccati recursion based solver, Schur complement based solver, and sparse iterative methods: in case of dense matrices in the LQ control problem, the complexity is typically $\mathcal{O}(N(n_x+n_u)^3)$, and they are suitable for problems with long control horizon [3].

In this paper we consider only solvers in this second group. In particular, we will focus our attention on the Riccati solver, that is known to be an efficient method for the solution of the LQ control problem [1]. We present a novel implementation, where the recursion matrix is no longer P_n, but its Cholesky factor $\mathcal{L}_n$: this allows a reduction in the number of flops. Furthermore, we propose the use of regularization, iterative refinement and mixed precision in a Riccati solver able to solve large instances of the LQ control problem up to 3 times faster than the classical implementation.

The paper is organized as follows. Section II introduces the extended LQ control problem and states necessary and sufficient conditions for its solution. In section III we present and compare methods for the solution of the extended LQ control problem: direct sparse solvers, Schur complement solver and Riccati solver. In section IV we present our implementations of the Riccati solver, and analyze their theoretical complexity. In section V we compare each other the Riccati solvers presented in this paper. Finally, section VI contains the conclusion.

II. The extended LQ control problem

The extended LQ control problem is an generalization of the classical LQ control problem. The cost function has quadratic, linear and constant terms, and the constraints are affine. Furthermore, all matrices are time variant. Its structure is flexible enough to describe a wide range of problems [4]. In particular, it can be used as a routine in AS and IP methods [2].

Problem 1: The extended LQ control problem is the equality constrained quadratic program

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

where $n \in \{0, 1, \ldots, N-1\}$ and

$$l_n(x_n, u_n) = \frac{1}{2}\begin{bmatrix} x_n' & u_n' \end{bmatrix}\begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix}\begin{bmatrix} x_n \\ u_n \end{bmatrix} + \begin{bmatrix} q_n' & s_n' \end{bmatrix}\begin{bmatrix} x_n \\ u_n \end{bmatrix} + \rho_n$$

$$l_N(x_N) = \frac{1}{2} x_N' \hat{P} x_N + \hat{p}' x_N + \hat{\rho}_N$$

The state vector x_n has size n_x, the input vector u_n has size n_u, and N is the control horizon length.

G. Frison and J.B. Jørgensen are with Technical University of Denmark, DTU Compute - Department of Applied Mathematics and Computer Science, DK-2800 Kgs Lyngby, Denmark. {giaf , jbj} at imm.dtu.dk

Problem (1) can be rewritten in a more compact form as

$$\begin{aligned} \min_x \quad & \phi = \frac{1}{2} x' H x + g' x \\ s.t. \quad & Ax = b \end{aligned} \tag{2}$$

where (in the case of $N = 3$)

$$x = \begin{bmatrix} u_0 \\ x_1 \\ u_1 \\ x_2 \\ u_2 \\ x_3 \end{bmatrix}, H = \begin{bmatrix} R_0 & & & & & & \\ & Q_1 & S_1' & & & & \\ & S_1 & R_1 & & & & \\ & & & Q_2 & S_2' & & \\ & & & S_2 & R_2 & & \\ & & & & & \hat{P} \end{bmatrix}, g = \begin{bmatrix} \tilde{s}_0 \\ q_1 \\ s_1 \\ q_2 \\ s_2 \\ \hat{p} \end{bmatrix}$$

$$A = \begin{bmatrix} -B_0 & I & & & & \\ & -A_1 & -B_1 & I & & \\ & & & -A_2 & -B_2 & I \end{bmatrix}, b = \begin{bmatrix} \tilde{b}_0 \\ b_1 \\ b_2 \end{bmatrix}$$

where $\tilde{s}_0 = S_0 x_0 + s_0$ and $\tilde{b}_0 = A_0 x_0 + b_0$. The matrices H (of size $(n_x + n_u)N \times (n_x + n_u)N$) and A (of size $n_x N \times (n_x + n_u)N$) are large and sparse; furthermore, H is block diagonal.

The following theorem gives necessary conditions for the solution of problem (2).

Theorem 1 (KKT (necessary) conditions): If x^* is a solution of problem (2), then a vector π^* of size Nn_x exists such that

$$\begin{bmatrix} H & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} x^* \\ \pi^* \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix} \tag{3}$$

System (3) is the KKT system associated with problem (2), and in the case of the extended LQ control problem the KKT matrix is large (of size $(2n_x + n_u)N \times (2n_x + n_u)N$) and sparse.

Sufficient conditions for existence and uniqueness of the solution of problem (2) are given in the following theorem.

Theorem 2 (Sufficient conditions): Let the matrices P and $\begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix}$ be positive semi-definite, and the matrices R_n be positive definite for all $n \in \{0, 1, \ldots, N-1\}$, then problem (2) has one and only one solution, given by the solution of the KKT system (3).

The proof of both theorems can be found in [3].

If the hypothesis of theorem 2 are satisfied and if the matrices Q_n, R_n and P are symmetric, then the KKT system (3) is a symmetric indefinite system of linear equations.

The KKT system (3) can be rewritten in the band diagonal form [1]

$$\begin{bmatrix} R_0 & B_0' & & & & & & & \\ B_0 & & -I & & & & & & \\ & -I & Q_1 & S_1' & A_1' & & & & \\ & & S_1 & R_1 & B_1' & & & & \\ & & A_1 & B_1 & & -I & & & \\ & & & & -I & Q_2 & S_2' & A_2' & \\ & & & & & S_2 & R_2 & B_2' & \\ & & & & & A_2 & B_2 & & -I \\ & & & & & & & -I & \hat{P} \end{bmatrix} \begin{bmatrix} u_0 \\ \pi_1 \\ x_1 \\ u_1 \\ \pi_2 \\ x_2 \\ u_2 \\ \pi_3 \\ x_3 \end{bmatrix} = \begin{bmatrix} -\tilde{s}_0 \\ -\tilde{b}_0 \\ -q_1 \\ -s_1 \\ -b_1 \\ -q_2 \\ -s_2 \\ -b_2 \\ -\hat{p} \end{bmatrix} \tag{4}$$

and then it can be solved in time $\mathcal{O}(N(n_x + n_u)^3)$ using a generic band diagonal solver.

III. Solution methods for the KKT system

In this section we briefly introduce a number of well-known methods for the solution of the KKT system (3). The asymptotic complexity is $\mathcal{O}(N(n_x + n_u)^3)$ (linear in the horizon length N and cubic in the number of states and inputs) for all solvers, but in practice the difference in performance between them can be more than an order of magnitude.

A. Direct sparse solvers

One approach is the use of general purpose sparse solvers for the direct solution of the KKT system in the band diagonal form (4). We consider two of the best solvers for the solution of sparse symmetric systems of linear equations: MA57 and PARDISO.

MA57 is a direct solver for symmetric sparse systems of linear equations. It is part of HSL [5] software. The code consists in routines for initialization, analysis, factorization and solution of the linear system, in both single and double precision. We tested version 3.7.0 of the software.

PARDISO [6] is a software package for the solution of large sparse systems of linear equations, both symmetric and non-symmetric. The solution process is divided into 3 phases: analysis, numerical factorization and solution. We tested version 4.1.2 of the software.

Both solvers use of the same optimized BLAS implementation. Times are relative to factorization and solution phases, since in MPC analysis phase is performed off-line.

The main advantage in the use of direct sparse solvers is that sparsity in the problem matrices A_i, B_i, Q_i, S_i, R_i can be exploited. Anyway, if these matrices are dense (as in the case of our tests), other methods are more efficient. Our numerical tests show that MA57 is roughly 4 times faster than PARDISO in solving (4).

B. Schur complement based solver

The Schur complement method for solving the KKT system (3) requires the matrix H to be invertible (and then positive definite). The method has been recently used in [7] for the fast computation of the Newton step.

An analytic expression for the solution of (3) is

$$\begin{aligned} (AH^{-1}A')\pi &= b + AH^{-1}g \\ Hx &= A'\pi - g \end{aligned} \tag{5}$$

where the matrix $\Psi = AH^{-1}A'$ (the Schur complement of the matrix H in the KKT matrix) is positive definite, since H^{-1} is positive definite and the matrix A has full row rank. All matrices are large, but the sparsity is preserved: for example, the Ψ matrix is block tridiagonal, with dense blocks of size $n_x \times n_x$. It is possible to exploit the matrices structure and use routines for dense linear algebra (e.g. BLAS and LAPACK) on the blocks.

A detailed description of our implementation can be found in [3]. It has complexity $N(\frac{19}{3}n_x^3 + 8n_x^2 n_u + 3n_x n_u^2 + \frac{2}{3}n_u^3)$ flops, that is $\mathcal{O}(N(n_x + n_u)^3)$.

The main advantage of this method is that it can be simplified in case of diagonal H matrix: the complexity reduces to $N(\frac{10}{3}n_x^3 + n_x^2 n_u)$ flops, that is linear in n_u.

In the general case of dense H matrix, the Schur complement based method is faster than direct sparse solvers, but slower than the Riccati recursion based method.

C. Riccati recursion based solver

The Riccati recursion is a well known method for the solution of the classical LQ control problem, and can be easily adapted to the solution of the extended formulation (1) [3], [8]. Several derivations exist: particularly important for the following of the paper is the interpretation of the Riccati recursion as a factorization procedure for (4) [1].

The Riccati recursion method is not able to exploit special problem properties (such as diagonal H matrix or time-invariant system), but in the general case it is more efficient than all previously considered methods: it is then particularly suitable as general solver for problem (1).

Some variants of the algorithm are presented in details in the next section.

D. Comparison of solvers

All algorithms considered above have been implemented in C code and compared each other in the solution of a general instance of problem (1).

The tests have been preformed on a laptop equipped with an Intel Pentium Dual-Core T2390 @ 1.86 GHz processor. To perform linear algebra operations, we used the BLAS and LAPACK libraries are provided by Intel MKL.

The tests confirmed the theoretical complexity of $\mathcal{O}(N(n_x + n_u)^3)$ for all solvers. It should be noted that the cubic growth in n_x and n_u is observed only for respectively $n_x \gg n_u$ and $n_u \gg n_x$: in fact, as long as $n_x \gg n_u$, changes in the value of n_u do not affect much the computation time.

In figure 1 there is a comparison of the computation time in the case where only n_x is varied (i.e. for fixed N and n_u): as already said, the fastest method is the Riccati recursion, followed by Shur complement. Both method are tailored for the special form of problem (1), and outperform general-purpose direct sparse solvers.

IV. Efficient implementation of the Riccati recursion based solver

As seen in the previous section, the Riccati recursion based solver is particularly well suited as general solver for problem (1). In this section we present two versions of the algorithm, and show how to efficiently implement the second one. The result will be a solver up to 3 times faster (for systems with many states) than the classical implementation of the algorithm.

For our purposes, it is convenient to interpret the Riccati solver as a factorization method for (4): in particular, the method can be divided into a factorization phase (where the KKT matrix is factorized) and a solution phase (where the KKT system is solved). The factorization phase is the main computational effort of the algorithm: its complexity is

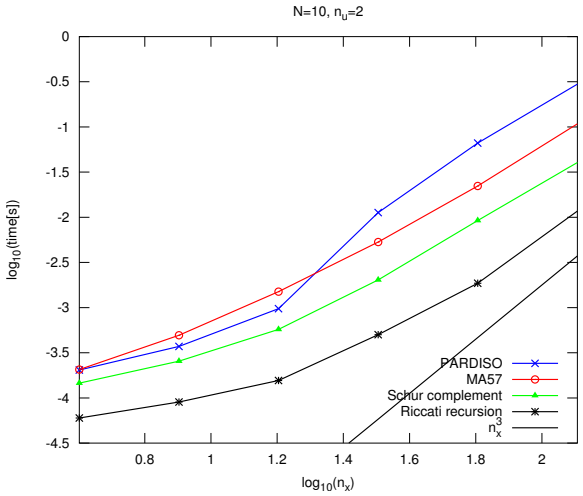


Fig. 1: Comparison of solvers PARDISO, MA57, Schur complement and Riccati recursion, for problem (1).

$\mathcal{O}(N(n_x+n_u)^3)$, while the solution phase is only $\mathcal{O}(N(n_x+n_u)^2)$, quadratic in n_x and n_u.

A. Classical version

What follows is a careful implementation of the classical Riccati solver. In this version we only use the BLAS routines `dgemm`, `dtrsm` (and vector counterparts), and `dpotrf`: the algorithm can thus be easily implemented also in Matlab.

Particular attention is given in accessing contiguous data in memory: since all matrices are stored in column-major (or Fortran-like) order, the better performance in matrix-matrix multiplications is obtained when the left matrix is transposed and the right one is not.

1) Factorization phase: The factorization phase is given by the classical Riccati (backward) recursion, the algorithm is presented in Algorithm 1. In the common case of $n_x > n_u$, the most expensive operation is the computation of the term $A_n' P_{n+1} A_n$, requiring $4n_x^3$ flops.

To improve performance, A_n and B_n are packed in the matrix $[A_n|B_n]$, of size $n_x \times (n_x + n_u)$: this reduces the number of calls to `dgemm` and improves data reuse. The three matrix-matrix multiplications in lines 3,4,5 totally require $4n_x^3 + 4x_x n_u^2 + 2n_x n_u^2$ flops. Notice that the left matrices are always transposed (by exploiting the symmetry of P_{n+1}) and the right ones are never.

The Cholesky factorization in line 6 is performed using the blocked LAPACK routine `dpotrf`, and requires $\frac{1}{3}n_u^3$ flops. The triangular system solution in line 7 is performed using the BLAS routine `dtrsm`, and requires $n_x n_u^2$ flops. The matrix-matrix product in line 8 is performed using the BLAS routine `dgemm`, requiring $2n_x^2 n_u$ flops.

Numerical evidence shows that line 9 may improve the stability of the algorithm, in case of unstable systems [8]. It is implemented as a blocked algorithm, to reuse data in cache.

The overall algorithm requires

$$N\left(4n_x^3 + 6n_x^2 n_u + 3n_x n_u^2 + \tfrac{1}{3}n_u^3\right) \text{ [flops].}$$

Algorithm 1 Factorization phase, classical version

1: $P_N \leftarrow \hat{P}$
2: **for** $n = N-1 \rightarrow 0$ **do**
3: $\quad [PA|PB] \leftarrow P'_{n+1} \cdot [A_n|B_n]$
4: $\quad [B'PA|B'PB] \leftarrow B'_n \cdot [PA|PB]$
5: $\quad A'PA \leftarrow A'_n \cdot PA$
6: $\quad \Lambda_n \leftarrow \mathrm{chol_L}(R_n + B'PB)$
7: $\quad L_n \leftarrow \Lambda_n^{-1}(S_n + B'PA)$
8: $\quad P_n \leftarrow Q_n + A'PA - L'_n \cdot L_n$
9: $\quad P_n \leftarrow 0.5(P_n + P'_n)$
10: **end for**

2) Solution phase: In the solution phase, we need the matrices sequences L_n, Λ_n and P_n computed in the previous factorization phase. The algorithm is presented in Algorithm 2. It consists of a backward loop and a forward loop. All matrix-vector multiplications are implemented using the BLAS routine `dgemv`, while the system solutions at lines 3 and 8 using the BLAS routine `dtrsv`. The cost of the algorithm is

$$N\big(8n_x^2 + 8n_x n_u + 2n_u^2\big) \text{ [flops]}.$$

Algorithm 2 Solution phase

1: $p_N \leftarrow \hat{p}$
2: **for** $n = N-1 \rightarrow 0$ **do**
3: $\quad l_n \leftarrow \Lambda_n^{-1}(s_n + B'_n \cdot (P'_{n+1} \cdot b_n + p_{n+1}))$
4: $\quad p_n \leftarrow q_n + A'_n \cdot (P'_{n+1} b_n + p_{n+1}) - L'_n \cdot l_n$
5: **end for**
6: $\pi_0 \leftarrow P_0 \cdot x_0 + p_0$
7: **for** $n = 0 \rightarrow N-1$ **do**
8: $\quad u_n \leftarrow -(\Lambda'_n)^{-1}(L_n \cdot x_n + l_n)$
9: $\quad x_{n+1} \leftarrow A_n \cdot x_n + B_n \cdot u_n + b_n$
10: $\quad \pi_{n+1} \leftarrow P'_{n+1} \cdot x_{n+1} + p_{n+1}$
11: **end for**

B. Factorized version

In this version we aim at reducing the theoretical number of flops as much as possible. The algorithm is presented in Algorithm 3.

This version requires that all matrices in the sequence P_n must be (strictly) positive definite: a sufficient condition for this is the further hypothesis that all matrices Q_n and P are positive definite [3]. This could restrict the applicability of the algorithm used in this form, or calls for some tricks to use it in case of rank-deficient matrices, as shown later.

1) Factorization phase: The key idea in this version is to write the recursion in terms of the Cholesky factor $\mathcal{L}_n$ in place of P_n: this allows a reduction in the number of flops. Furthermore, this permits to pack the matrices, reducing the number of function calls and improving the reuse of data in cache. The requirement about the positive definiteness of the matrices P_n is a technical condition needed for the use of the Cholesky factorization.

The matrices A_n and B_n are packed in the $n_x \times (n_x + n_u)$ matrix $[B_n|A_n]$. The matrix $\mathcal{L}_{n+1}$ is the lower triangular Cholesky factor of P_{n+1}, and then the product at line 3 is performed using the BLAS routine `dtrmm`, requiring $n_x^2(n_x + n_u)$ flops. The lower triangular part of the matrices $A'PA$ and $B'PB$ and the matrix $A'PB$ are built all together thank to the matrix-matrix product at line 4, performed using the BLAS routine `dsyrk`, requiring $n_x(n_x + n_u)^2$.

Finally, the matrices Λ_n, L_n and $\mathcal{L}_n$ are build all together thanks to a call to the Cholesky factorization routine `dpotrf`, requiring $\frac{1}{3}(n_x + n_u)^3$. In fact, if we perform a block Cholesky factorization on the right-hand-side matrix at line 5, we get (compare lines 6,7,8 of Algorithm 1)

$$\begin{aligned}
\Lambda_n &\leftarrow \mathrm{chol_L}(R_n + B'PB) \\
L'_n &\leftarrow (S'_n + A'PB)(\Lambda'_n)^{-1} \\
\mathcal{L}_n &\leftarrow \mathrm{chol_L}(Q_n + A'PA - L'_n \cdot L_n)
\end{aligned}$$

The total cost of the algorithm is

$$N\left(\tfrac{7}{3}n_x^3 + 4n_x^2 n_u + 2n_x n_u^2 + \tfrac{1}{3}n_u^3\right) \text{ [flops]},$$

lower than the cost of the classical version. In case of n_x large and $n_x \gg n_u$, the theoretical cost of the classical version is roughly $\frac{12}{7} = 1.71$ times the cost of the factorized version.

Algorithm 3 Factorization phase, factorized version

1: $\mathcal{L}_N \leftarrow \mathrm{chol_L}(\hat{P})$
2: **for** $n = N-1 \rightarrow 0$ **do**
3: $\quad [\mathcal{L}'B|\mathcal{L}'A] \leftarrow \mathcal{L}'_{n+1} \cdot_{dtrmm} [B_n|A_n]$
4: $\quad \begin{bmatrix} B'PB & \\ A'PB & A'PA \end{bmatrix} \leftarrow [\mathcal{L}'B|\mathcal{L}'A]' \cdot_{dsyrk} [\mathcal{L}'B|\mathcal{L}'A]$
5: $\quad \begin{bmatrix} \Lambda_n & \\ L'_n & \mathcal{L}_n \end{bmatrix} \leftarrow \mathrm{chol_L}\Big(\begin{bmatrix} R_n + B'PB & \\ S'_n + A'PB & Q_n + A'PA \end{bmatrix} \Big)$
6: **end for**

2) Solution phase: The algorithm is almost identical to the one presented in Algorithm 2. The only difference is that now we have the lower Cholesky factor $\mathcal{L}_{n+1}$ of P_{n+1}: the product in the innermost bracket at line 3 is computed as $\mathcal{L}_{n+1} \cdot (\mathcal{L}'_{n+1} \cdot b_n) + p_{n+1}$ (using the triangular matrix-vector product routine `dtrmv`), and similarly for the product at line 10. The cost of the algorithm is the same.

3) Static and dynamic regularization: The main disadvantage of the factorized version is the requirement for the positive definiteness of the sequence of matrices P_n: this may limit the applicability of the algorithm, and it may happen that, due to round off error, a theoretically positive definite matrix actually has a negative or null leading minor. To overcome this limitations, we use regularization. We present two different approaches.

The first one consists in a combination of dynamic regularization of the matrix $Q_n + A'PA$ and a modification of the Cholesky factorization routine. In details, the diagonal elements a_{jj} of $Q_n + A'PA$ are checked, and if $a_{jj} < \epsilon$, then we set $a_{jj} = \epsilon$, with $\epsilon = 10^{-14}$. This is justified by the fact that the strict positiveness of the diagonal elements is a

necessary (even if not sufficient) condition for the positive definiteness of a matrix. Furthermore, the LAPACK routine `dpotf2` is modified such that, if the next diagonal element to be processes a_{jj} is too small or non-positive, it is replaced by a small positive number:

```
...
if a_jj < 10^-14 then
   a_jj ← 10^-14
end if
a_jj ← sqrt(a_jj)
...
```

The combination of dynamic regularization and modification of the Cholesky factorization routine is an heuristic that gives a good trade-off between stability and performance: in all our tests, it allowes to successfully complete the factorization, and if the matrix is already positive definite, no regularization is performed.

The second approach is more conservative, and it consists in a static regularization of the (in general only positive semi-definite) Q_n matrix, that is replaced with $Q_n + \epsilon I$ with $\epsilon = 10^{-14}$. This should ensure that the resulting matrix is positive definite, but the matrix is modified also if it is already positive definite. For extra safety, the modified Cholesky factorization may be used.

If regularization is performed, the algorithm commits an approximation error in the solution of (4): anyway, our numerical tests show that typically the approximate solution is only one order of magnitude less accurate than the solution computed by means of the classical version.

4) Iterative refinement: The accuracy of the approximate solution computed in case of regularization can be improved by using iterative refinement [9].

The idea is the following: we look for the solution of the system $My = m$, but in the solution process we prefer to use the matrix $\widetilde{M}$, close to M but easier to factorize. This means that we actually solve the system $\widetilde{M}\tilde{y} = m$, where the solution $\tilde{y}$ is only an approximation of y: the residuals are $r_1 = m - M\tilde{y} \neq 0$.

We can look for a correction term Δy_1 such that $M\Delta y_1 = r_1$, that means $M(\tilde{y}+\Delta y_1) = m - r_1 + r_1 = m$. Again, in the solution process we prefer the use of $\widetilde{M}$, and then we solve the system $\widetilde{M}\Delta\tilde{y}_1 = r_1$, obtaining the new approximate solution $\tilde{y} + \Delta\tilde{y}_i$. The new residuals $r_2 = m - M(\tilde{y} + \Delta\tilde{y}_1)$ are smaller than r_1, and the procedure can be iterated until the desired accuracy is reached.

5) Residual computation: In an iterative refinement step, the two most expensive operations are the system solution and the computation of the residuals (since the matrix has already been factorized).

The residual computation in the case of problem (1) is presented in Algorithm 4. Matrix-vector products are performed by using the BLAS routines `dgemv` and `dsymv`. The cost of the algorithm is

$$N\left(6n_x^2 + 8n_x n_u + 2n_u^2\right) \text{ [flops]}.$$

Notice that the algorithm can be simplified in case of diagonal H.

Algorithm 4 Residual computation

1: $rs_0 \leftarrow -(S_0 x_0 + R_0 u_0 + B_0' \pi_1 + s_0)$
2: $rb_0 \leftarrow x_1 - (A_0 x_0 + B_0 u_0 + b_0)$
3: **for** $n = 1 \rightarrow N - 1$ **do**
4: $\quad rq_n \leftarrow \pi_n - (Q_n x_n + S_n' u_n + A_n' \pi_{n+1} + q_n)$
5: $\quad rs_n \leftarrow -(S_n x_n + R_n u_n + B_n' \pi_{n+1} + s_n)$
6: $\quad rb_n \leftarrow x_{n+1} - (A_n x_n + B_n u_n + b_n)$
7: **end for**
8: $rq_N \leftarrow \pi_N - (P x_N + p)$

6) Mixed precision: In most current computer architectures, there is significant performance advantage in using single instead of double precision floating point numbers. In particular, this is true for SIMD instructions of conventional processors, that can process twice as many floats as doubles per clock cycle. Hence the use of mixed precision techniques, to speed up the computation while maintaining the double precision of the resulting solution [10].

In our case, it is particularly advantageous to adopt this approach, and correct at the same time the errors due to the regularization and to the single precision. The overall algorithm is summarized in Algorithm 5. In single precision, we use as regularization parameter $\epsilon = 10^{-6}$ in static and dynamic regularization and modified Cholesky factorization.

Algorithm 5 Riccati recursion based solver, mixed precision factorized version with regularization

1: Factorize the KKT matrix in single precision using Algorithm 3 with regularization
2: Solve the KKT system in single precision using Algorithm 2, obtaining x, u, π
3: Compute the residuals in double precision using Algorithm 4
4: **while** the residuals are not small enough **do**
5: $\quad$ Solve the KKT system in single precision using Algorithm 2 and the residuals as right hand side, obtaining $\Delta x, \Delta u, \Delta\pi$
6: $\quad$ Update the solution $(x, u, \pi) \leftarrow (x, u, \pi) + (\Delta x, \Delta u, \Delta\pi)$
7: $\quad$ Compute the residuals in double precision using Algorithm 4
8: **end while**

V. Numerical Results

As test problem, we used a system of $q = \frac{n_x}{2}$ equal masses connected in a row by springs, and to walls at the ends. Each mass is 1 Kg, and the spring constant is 1 N/m. There are 4 actuators that can exert a force on the first 4 masses. A continuous-time state-space system is obtained by choosing the masses displacement as the first q states and the masses velocity as the remaining q states. This system is sampled with sampling time $T_s = 1$ sec to obtain a discrete-time space-state system. There are no constraints on the masses displacement or on the forces. The cost function is chosen

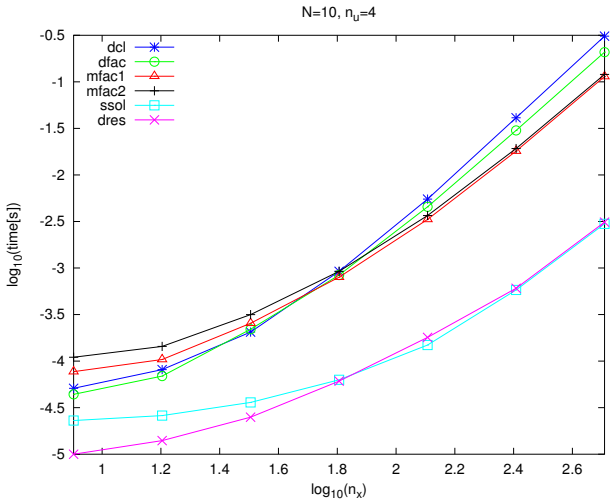


Fig. 2: Run-time. Proposed Algorithm 3 (`dfac`) is faster than classical Algorithm 1 (`dge`). Mixed precision versions `mfac1` and `mfac2` are faster than double precision version `dfac` for large n_x.

such that $P = Q_n = [I_q\ 0]'[I_q\ 0]$, $S_n = 0$, $R_n = I_4$, $p = q_n = 0$, $s_n = 0$. Notice that the sampling procedure will produce subnormal values for large n_x, and that they will heavily influence the performance if not flushed to zero.

The test machine is a laptop equipped with Intel i5-2410M CPU @ 2.30GHz, running Xubuntu 12.10. The processor supports the AVX instruction set, and can process a vector of 4 doubles or 8 floats per cycle. The code is written in C, and compiled with gcc 4.7.2. The flush-to-zero mode is activated by using the command `_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON)` in the main. The BLAS and LAPACK libraries are provided by OpenBLAS 0.2.6, an highly optimized implementation released with the BSD license [11]: this allowed us to modify the `dpotf2` routine and still have high performances. All tests are performed in single thread mode.

We performed a test varying the number of states n_x in a wide range. We tested the algorithms: double precision classical version (`dcl`), double precision factorized version (`dfac`), single precision factorized version (`sfac`), mixed precision factorized version with 1 (`mfac1`) and 2 (`mfac2`) refinement steps, single precision solution phase (`ssol`) and double precision residual computation (`dres`). Results are in Figure 2 and Table I. In double precision, the factorized version is almost always faster than the classical one (except for $n_x = 32$), with a speed-up of roughly 1.5 times for large n_x. About the mixed precision version, it is slow for small n_x, since the cost of even a single refinement step is of the same order of magnitude as the factorization. But for medium to large n_x it gets quickly faster, with a speed-up of up to 3 with respect to the classical version.

Table II shows an accuracy test: the mixed precision factorized version is already very accurate with 1 refinement step, and as accurate as the double precision classical version with 2 steps.

n_x	`dfac`	`sfac`	`mfac1`	`mfac2`
8	1.16	1.16	0.66	0.46
16	1.17	1.27	0.78	0.56
32	0.93	1.06	0.80	0.65
64	1.10	1.37	1.15	1.00
128	1.21	1.83	1.65	1.50
256	1.37	2.44	2.28	2.14
512	1.49	2.87	2.71	2.58
1024	1.56	3.00	2.87	2.75
2048	1.61	3.14	3.06	2.99

TABLE I: Speedup of double (`dfac`), single (`sfac`) and mixed (`mfac1`, `mfac2`) precision versions of proposed Algorithm 3 with respect to double precision version `dcl` of classical Algorithm 1.

n_x	`dcl`	`dfac`	`sfac`	`mfac1`	`mfac2`
32	3.55e-14	5.59e-14	1.78e-05	2.23e-11	3.02e-14

TABLE II: $||\cdot||_\infty$ of residuals. Mixed precision version is already very accurate with 1 refinement step (`mfac1`), as accurate as double precision with 2 steps (`mfac2`).

VI. Conclusion

In this paper we have seen that a Riccati recursion based solver is an efficient solver for LQ control problems in the general form (1), being faster than other widely-used solvers.

The main contribution of the paper is a novel implementation of the Riccati solver, that makes use of Cholesky factorization of the P_n matrices to reduce the number of flops. When combined with regularization, iterative refinement and mixed precision, the resulting algorithm can solve large instances of the LQ control problem up to 3 times faster than the classical version, and maintaining the same accuracy.

References

[1] C.V. Rao, S.J. Wright, and J.B. Rawlings (1998). Application of interior-point methods to model predictive control. *Journal of Optimization Theory and Applications*, 99(3), 723-757

[2] J.B. Jørgensen, J.B. Rawlings, and S.B. Jørgensen (2004). Numerical methods for large scale moving horizon estimation and control. In *DYCOPS 7*. IFAC, Cambridge, MA.

[3] G. Frison (2012). *Numerical Methods for Model Predictive Control*. M.Sc. thesis, Department of Informatics and Mathematical Modelling, Technical University of Denmark, Kgs. Lyngby, Denmark.

[4] J.B. Jørgensen, G. Frison, N.F. Gade-Nielsen, and B. Damman (2012). Numerical Methods for Solution of the Extended Linear Quadratic Control Problem. *Proc. IFAC NMPC'12*, 187-193.

[5] *http://www.hsl.rl.ac.uk/*

[6] *http://www.pardiso-project.org/*

[7] Y. Wang and S. Boyd (2010). Fast model predictive control using online optimization. *IEEE Transactions on Control Systems Technology*, 18(2), 267-278.

[8] J.B. Jørgensen (2005). *Moving Horizon Estimation and Control*. Ph.D. thesis, Department of Chemical Engineering, Technical University of Denmark, Kgs. Lyngby, Denmark.

[9] J. Mattingley, S. Boyd (2012). CVXGEN: a code generator for embedded convex optimization. *Optim. Eng.*, 12, 1-27.

[10] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak (2007). Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems. *The International Journal of High Performance Computing Applications*, 21(4), 457-466.

[11] *http://xianyi.github.io/OpenBLAS/*

Chapter 2

Paper [34]

Parallel Implementation of Riccati Recursion for Solving Linear-Quadratic Control Problems

Gianluca Frison * **John Bagterp Jørgensen** *

* *Technical University of Denmark, DTU Compute - Department of Applied Mathematics and Compute Science, DK-2800 Kgs Lyngby, Denmark. (e-mail: {giaf, jbj} at imm.dtu.dk).*

Abstract: In both Active-Set (AS) and Interior-Point (IP) algorithms for Model Predictive Control (MPC), sub-problems in the form of linear-quadratic (LQ) control problems need to be solved at each iteration. The solution of these sub-problems is usually the main computational effort. In this paper an alternative version of the Riccati recursion solver for LQ control problems is presented. The performance of both the classical and the alternative version is analyzed from a theoretical as well as a numerical point of view, and the alternative version is found to be approximately 50% faster than the classical one, for systems with many states. A number of parallel implementations of the alternative version has been proposed and tested.

Keywords: Riccati recursion, LQ control problem, parallel computation

1. INTRODUCTION

The linear-quadratic (LQ) control problem can be considered the core problem in Model Predictive Control (MPC). In its classical formulation, it represents an unconstrained optimal control problem where the controlled system is linear time-invariant and the cost function is quadratic. This problem formulation is especially important because it arises as a sub-problem in Active-Set (AS) and Interior-Point (IP) algorithms for MPC (Wright (1997); Rao et al. (1998); Jørgensen et al. (2004)). The solution of these sub-problems is typically the main computational effort at each iteration, and this explains the need for efficient solvers.

From a mathematical point of view, the LQ control problem is an equality constrained quadratic program, and it can be solved using general solvers for this class of problems. The cost of this approach is $\mathcal{O}(N^3(n_x + n_u)^3)$, where N is the control horizon length, n_x is the number of states and n_u is the number of controls (or inputs).

However, it is well known that the KKT system associated with the LQ control problem is sparse and highly structured, and this structure can be exploited to obtain more efficient solvers. In case of dense controlled systems, the Riccati recursion based solver is known to be the fastest among a large class of solvers (Frison et al. (2013)).

In this paper, we present two versions of the Riccati recursion based solver for an extended formulation of the LQ control problem. For both the classical and the alternative (called 'factorized' in Frison et al. (2013)) version, we state a detailed description of the algorithm, and we suggest and test the use of numerical libraries for their parallel implementation on shared memory machines. The implementation of the classical version scales quite well with the number of threads, since its key routine (the matrix-matrix multiplication routine) is particularly parallel friendly. On the contrary, the key routine of the factorized version (the Cholesky factorization routine) is not so parallel friendly, and this affects the scalability of the factorized version. Therefore, we tested a number of implementations of the factorized version, aiming at improving its scalability.

The paper is organized as follows. In section 2 we present an extended formulation of the LQ control problem, and we state conditions for its solution. In section 3 we present a general formulation of the Riccati recursion based solver for the extended LQ control problem. Efficient implementation of both the classical and the factorized version of this Riccati solver are presented in section 4. In section 5 we present the libraries used in our tests, and the result and the discussion of these tests are reported in section 6. Finally, section 7 contains the conclusion.

2. THE EXTENDED LQ CONTROL PROBLEM

In this paper we consider an extended version of the classical LQ control problem: in this formulation, the cost function has a quadratic, a linear and a constants term, and the constraint (given by the equation describing the dynamic system) is affine. Furthermore, all matrices are time variant. The classical and the extended LQ control problems can be solved by means of Riccati recursion based solvers at the same asymptotic cost: the cubic (dominant) terms in the respective cost functions are identical. The main advantage of the extended formulation is that it is flexible enough to describe a wide range of problems (Jørgensen et al. (2012)): in particular, it can be used as sub-routine in AS and IP methods.

Problem 1. The extended LQ control problem is the equality constrained quadratic program

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

where $n \in \{0, 1, \ldots, N-1\}$ and

$$l_n(x_n, u_n) = \frac{1}{2} \begin{bmatrix} x_n' & u_n' \end{bmatrix} \begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \begin{bmatrix} q_n' & s_n' \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \rho_n$$

$$l_N(x_N) = \frac{1}{2} x_N' P x_N + p' x_N + \rho_N$$

The state vector x_n has size n_x, the input vector u_n has size n_u, and N is the control horizon length.

Problem (1) can be rewritten in a more compact form as

$$\begin{aligned} \min_{x} \quad & \phi = \frac{1}{2} x' H x + g' x \\ s.t. \quad & Ax = b \end{aligned} \tag{2}$$

where (in the case of $N = 3$)

$$x = \begin{bmatrix} u_0 \\ x_1 \\ u_1 \\ x_2 \\ u_2 \\ x_3 \end{bmatrix}, H = \begin{bmatrix} R_0 & & & & & \\ & Q_1 & S_1' & & & \\ & S_1 & R_1 & & & \\ & & & Q_2 & S_2' & \\ & & & S_2 & R_2 & \\ & & & & & P \end{bmatrix}, g = \begin{bmatrix} \tilde{s}_0 \\ q_1 \\ s_1 \\ q_2 \\ s_2 \\ p \end{bmatrix}$$

$$A = \begin{bmatrix} -B_0 & I & & & & \\ & -A_1 & -B_1 & I & & \\ & & & -A_2 & -B_2 & I \end{bmatrix}, b = \begin{bmatrix} \tilde{b}_0 \\ b_1 \\ b_2 \end{bmatrix}$$

where $\tilde{s}_0 = S_0 x_0 + s_0$ and $\tilde{b}_0 = A_0 x_0 + b_0$. The matrices H and A are large and sparse; in particular, H is block diagonal.

Theorem 1. (KKT (necessary) conditions). If x^* is a solution of problem (2), then there exists a vector π^* of size $N \cdot n_x$ such that

$$\begin{bmatrix} H & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} x^* \\ \pi^* \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix} \tag{3}$$

System (3) is the KKT system associated with problem (2), and in the case of the extended LQ control problem the KKT matrix is large (of size $(2n_x + n_u)N \times (2n_x + n_u)N$) and sparse.

Sufficient conditions for existence and uniqueness of the solution of problem (2) are given in the following theorem.

Theorem 2. (Sufficient conditions). Let the matrices P and $\begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix}$ be positive semi-definite, and the matrices R_n be positive definite for all $n \in \{0, 1, \ldots, N-1\}$, then problem (2) has one and only one solution, given by the solution of the KKT system (3).

The proof of both theorems can be found in Frison (2012).

If the hypothesis of theorem 2 are satisfied and if the matrices Q_n, R_n and P are symmetric, then the KKT system (3) is a symmetric indefinite system of linear equations. In the following we assume that these hypothesis hold.

3. RICCATI RECURSION FOR SOLVING LQ CONTROL PROBLEMS

As shown in Wright (1997), the KKT system (3) can be rewritten in band diagonal form as

$$\begin{bmatrix} R_0 & B_0' & & & & & & & & \\ B_0 & -I & & & & & & & & \\ & -I & Q_1 & S_1' & A_1' & & & & & \\ & & S_1 & R_1 & B_1' & & & & & \\ & & A_1 & B_1 & -I & & & & & \\ & & & & -I & Q_2 & S_2' & A_2' & & \\ & & & & & S_2 & R_2 & B_2' & & \\ & & & & & A_2 & B_2 & -I & & \\ & & & & & & & -I & P & \end{bmatrix} \begin{bmatrix} u_0 \\ \pi_1 \\ x_1 \\ u_1 \\ \pi_2 \\ x_2 \\ u_2 \\ \pi_3 \\ x_3 \end{bmatrix} = \begin{bmatrix} -\tilde{s}_0 \\ -\tilde{b}_0 \\ -q_1 \\ -s_1 \\ -b_1 \\ -q_2 \\ -s_2 \\ -b_2 \\ -p \end{bmatrix} \tag{4}$$

and solved in time $\mathcal{O}(N(n_x + n_u)^3)$ by using the Riccati recursion to factorize the KKT system. A Riccati recursion based solver for problem (1) is summarized in Algorithm 1 (see Frison (2012); Jørgensen (2005)).

Algorithm 1 Riccati recursion based solver for the extended LQ control problem (1)

$P_N \leftarrow P$
$p_N \leftarrow p$
for $n = N-1 \rightarrow 0$ **do**
  $R_{e,n} \leftarrow R_n + B_n' P_{n+1} B_n$
  $K_n \leftarrow -R_{e,n}^{-1}(S_n + B_n' P_{n+1} A_n)$
  $P_n \leftarrow Q_n + A_n' P_{n+1} A_n - K_n' R_{e,n} K_n$
  $k_n \leftarrow -R_{e,n}^{-1}(s_n + B_n'(P_{n+1} b_n + p_{n+1}))$
  $p_n \leftarrow q_n + A_n'(P_{n+1} b_n + p_{n+1}) - K_n' R_{e,n} k_n$
end for
$\pi_0 \leftarrow P_0 x_0 + p_0$
for $n = 0 \rightarrow N-1$ **do**
  $u_n \leftarrow K_n x_n + k_n$
  $x_{n+1} \leftarrow A_n x_n + B_n u_n + b_n$
  $\pi_{n+1} \leftarrow P_{n+1} x_{n+1} + p_{n+1}$
end for

4. EFFICIENT IMPLEMENTATION OF THE RICCATI RECURSION BASED SOLVER

In this section we present a detailed description of two different implementations of the Riccati recursion based solver.

We will focus our attention to the case $n_x > n_u$: this means that, at each iteration, the most expensive part is the computation of the term $A_n' P_{n+1} A_n$. This expression has some structure: the left matrix A_n' is equal to the transpose of the right matrix A_n; furthermore, the central matrix P_{n+1} is symmetric and positive semi-definite (proof in Frison (2012)). It is possible to exploit this structure, as shown in the following.

We assume that all matrices are stored in memory in column-major (Fortran-like) order, and we make use of optimized BLAS and LAPACK routines for linear algebra operations. The following discussion can be easily adapted to the case of row-major (C-like) order, using C wrappers to BLAS and LAPACK.

4.1 Classical version

In this version, the term $A_n' P_{n+1} A_n$ is implemented as

$$A'_n \cdot (P'_{n+1} \cdot A_n)$$

by exploiting the symmetry of the P_{n+1} matrix: since the matrices are stored in column-major order, the best performance in the matrix-matrix multiplication is obtained if the left matrix is transposed and the right is not. The two matrix-matrix multiplications are performed using the BLAS general matrix-matrix multiplication routine `dgemm`. The computation of the expression requires roughly $4n_x^3$ flops.

The expressions $B'_n P_{n+1} B_n$ and $B'_n P_{n+1} A_n$ are computed in a similar way, as $B'_n \cdot (P'_{n+1} \cdot B'_n)$ (cost $2n_x^2 n_u + 2n_x n_u^2$ flops) and $(P'_{n+1} B_n)' \cdot A_n$ (cost $2n_x^2 n_u$ flops, re-using the already computed expression $P'_{n+1} B_n$).

The matrix $R_{e,n}$ is symmetric positive definite (since R_n is symmetric positive definite, and $B'_n P_{n+1} B_n$ is symmetric positive semi-definite): it can be factorized using the LAPACK Cholesky factorization routine `dpotrf`, obtaining the lower triangular factor Λ_n. This costs $\frac{1}{3} n_u^3$ flops.

About the computation of the term $K'_n R_{e,n} K_n$, we have

$$K'_n R_{e,n} K_n = M'_n R_{e,n}^{-1} R_{e,n} R_{e,n}^{-1} M_n = M'_n R_{e,n}^{-1} M_n = \\ = M'_n (\Lambda'_n)^{-1} \Lambda_n^{-1} M_n = (\Lambda_n^{-1} M_n)' (\Lambda_n^{-1} M_n) = L'_n L_n$$

where $M_n = S_n + B'_n P_{n+1} A_n$, and $L_n = \Lambda_n^{-1} M_n$. The operation $\Lambda_n^{-1} M_n$ is performed using the BLAS routine `dtrsm`, requiring $n_x n_u^2$ flops.

The equations for updating vectors are implemented in a similar way, even if their contribution to the total computation time is negligible.

In case of unstable systems, numerical evidence shows that the stability of the algorithm is improved by ensuring the symmetry of matrix P_n by means of the term $P_n \leftarrow 0.5(P_n + P'_n)$ (Jørgensen (2005)). There is not a BLAS or LAPACK routine implementing this operation, and we suggest to implement a blocked version in order to reduce cache misses, with block size equal to the cache line size.

The overall algorithm requires

$$N \left(4n_x^3 + 6n_x^2 n_u + 3n_x n_u^2 + \tfrac{1}{3} n_u^3 \right)$$

flops. The algorithm is summarized in Algorithm 2.

Algorithm 2 Efficient implementation of Riccati recursion based solver solver, classical version

$P_N \leftarrow P$
$p_N \leftarrow p$
for $n = N-1 \rightarrow 0$ **do**
  $R_{e,n} \leftarrow R_n + B'_n \cdot (P'_{n+1} \cdot B_n)$
  $\Lambda_n \leftarrow \text{chol}(R_{e,n}, \text{'lower'})$
  $L_n \leftarrow \Lambda_n^{-1} (S_n + (P'_{n+1} B_n)' \cdot A_n)$
  $P_n \leftarrow Q_n + A'_n \cdot (P'_{n+1} \cdot A_n) - L'_n \cdot L_n$
  $P_n \leftarrow 0.5(P_n + P'_n)$
  $l_n \leftarrow \Lambda_n^{-1} (s_n + B'_n \cdot (P_{n+1} \cdot b_n + p_{n+1}))$
  $p_n \leftarrow q_n + A'_n \cdot (P_{n+1} b_n + p_{n+1}) - L'_n \cdot l_n$
end for
$\pi_0 \leftarrow P_0 \cdot x_0 + p_0$
for $n = 0 \rightarrow N-1$ **do**
  $u_n \leftarrow -(\lambda'_n)^{-1} (L_n \cdot x_n + l_n)$
  $x_{n+1} \leftarrow A_n \cdot x_n + B_n \cdot u_n + b_n$
  $\pi_{n+1} \leftarrow P_{n+1} \cdot x_{n+1} + p_{n+1}$
end for

4.2 Factorized version

This version requires all matrices P_n to be positive definite: a sufficient condition for this is the further hypothesis that all matrices Q_n and P are positive definite Frison (2012).

The term $A'_n P_{n+1} A_n$ is implemented as

$$(\mathcal{L}'_n \cdot A_n)' \cdot (\mathcal{L}'_n A_n)$$

where $\mathcal{L}$ is the lower triangular factor of the Cholesky factorization of P_{n+1}. The advantage of this implementation is that the product $\mathcal{L}'_n \cdot A_n$ can be computed using the BLAS routine `dtrmm`, requiring n_x^3 flops, and the product $(\mathcal{L}'_n A_n)' \cdot (\mathcal{L}'_n A_n)$ of a matrix and its transposed can be computed using the BLAS routine `dsyrk`, requiring n_x^3 flops. Since the cost of the Cholesky factorization is roughly $\frac{1}{3} n_x^3$ flops, the total complexity is roughly $\frac{7}{3} n_x^3$ flops.

Using the LAPACK routine `dpotrf`, the computation of the lower factor is slightly less efficient than the computation of the upper factor; on the other hand, the lower factor gives the advantage that in each matrix-matrix multiplication the left matrix factor is transposed and the right matrix factor is not, exploiting the data order in memory.

In a similar way, the term $B'_n P_{n+1} B_n$ is computed as $(\mathcal{L}'_n \cdot B_n)' \cdot (\mathcal{L}'_n B_n)$, at the cost of $n_x^2 n_u + n_x n_u^2$ (re-using the factorization of P_{n+1}), and the term $B'_n P_{n+1} A_n$ is computed as $(\mathcal{L}'_n B_n)' \cdot (\mathcal{L}'_n A_n)$, at the cost of $2n_x^2 n_u$ flops (re-using the products $(\mathcal{L}'_n A_n)$ and $(\mathcal{L}'_n B_n)$).

The term $K'_n R_{e,n} K_n$ is computed again as in the classical version, except that the term $L'_n \cdot L_n$ is computed using the BLAS routine `dsyrk` instead of `dgemm`. The use of `dsyrk` implies that only the lower triangular part of P_{n+1} can be referenced: the terms $P_{n+1} \cdot b_n$ and $P_{n+1} \cdot x_{n+1}$ are then computed using the BLAS routine `dsymv` instead of `dgemv`.

The total cost of the algorithm is

$$N \left(\tfrac{7}{3} n_x^3 + 4n_x^2 n_u + 2n_x n_u^2 + \tfrac{1}{3} n_u^3 \right)$$

flops, lower than the cost of the classical version. In the case of n_x large and $n_x \gg n_u$, the theoretical cost of the classical version is approximately $\frac{12}{7} = 1.71$ times the cost of the factorized version. The algorithm is summarized in Algorithm 3.

5. LIBRARIES

In this section we want to briefly describe the libraries used in the code to perform linear algebra operations.

5.1 OpenBLAS

The BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage) libraries are provided by OpenBLAS [1], version 0.2.6. OpenBLAS is an open-source project (BSD license) that aims to extend GotoBLAS to the most recent architectures (e.g. Intel SandyBridge with AVX instruction set). It provides an optimized implementation of all BLAS and part of LAPACK routines: in particular, it provides an optimized implementation of the Cholesky factorization routine `dpotrf`. The

[1] see http://xianyi.github.com/OpenBLAS/

Algorithm 3 Efficient implementation of Riccati recursion based solver, factorized version

$P_N \leftarrow P$
$p_N \leftarrow p$
for $n = N-1 \rightarrow 0$ **do**
  $\mathcal{L} \leftarrow \text{chol}(P_{n+1}, '\text{lower}')$
  $R_{e,n} \leftarrow R_n + (\mathcal{L}' \cdot B_n)' \cdot (\mathcal{L}' B_n)$
  $\Lambda_n \leftarrow \text{chol}(R_{e,n}, '\text{lower}')$
  $L_n \leftarrow \Lambda_n^{-1}(S_n + (\mathcal{L}' B_n)' \cdot (\mathcal{L}' \cdot A_n))$
  $P_n \leftarrow Q_n + (\mathcal{L}' A_n)' \cdot (\mathcal{L}' A_n) - L_n' \cdot L_n$
  $l_n \leftarrow \Lambda_n^{-1}(s_n + B_n' \cdot (P_{n+1} \cdot b_n + p_{n+1}))$
  $p_n \leftarrow q_n + A_n' \cdot (P_{n+1} b_n + p_{n+1}) - L_n' \cdot l_n$
end for
$\pi_0 \leftarrow P_0 \cdot x_0 + p_0$
for $n = 0 \rightarrow N-1$ **do**
  $u_n \leftarrow -(\lambda_n')^{-1}(L_n \cdot x_n + l_n)$
  $x_{n+1} \leftarrow A_n \cdot x_n + B_n \cdot u_n + b_n$
  $\pi_{n+1} \leftarrow P_{n+1} \cdot x_{n+1} + p_{n+1}$
end for

remaining part of LAPACK is the library version 3.4.2 build using OpenBLAS as BLAS library.

OpenBLAS provides a parallel implementation of BLAS for shared memory machines, and makes use of Pthreads by default. The number of threads can be chosen by means of the environment variable `OPENBLAS_NUM_THREADS`, or at run time by using the function `openblas_set_num_threads()` in the code. This second option has the advantage to allow different number of threads in different parts of the code. Alternatively, it is possible to directly build a sequential library (without support for multi-threading): if possible, this second option should be preferred, since it avoids the overhead associated with the creation and destruction of threads at run-time.

LAPACK relies upon BLAS for parallelization: in fact, the LAPACK libraries has been written having in mind sequential machines, and can exploit parallelism only by calling parallel implementations of BLAS. This approach, however, limits the scalability of the code with the number of threads, especially for medium size problems.

5.2 PLASMA

PLASMA [2] (Parallel Linear Algebra for Scalable Multicore Architectures) is a project that aims to provide efficient parallel implementation of linear algebra routines on shared memory machines. It is released with BSD license. We tested the version 2.5.0 of the library. The approach is completely different compared to LAPACK's one: the parallelization is not hidden in the BLAS, but it is performed to an higher level. PLASMA needs a sequential implementation of BLAS, and explicitly takes care of parallelization, making use of Pthreads.

The main features are: tile matrix layout (the matrices are stored in memory in sub-matrices of contiguous elements), tile algorithms (exploiting the tale matrix layout, reducing the cache and TLB misses, and optimizing reuse of data in cache), dynamic scheduling (the assignment of the parallel tasks to the processors is made at run time) and asynchronous algorithms (returning before completion, and then allowing a routine to start on the idle processors even if the previous routine has not completed yet).

[2] see http://icl.cs.utk.edu/plasma/

PLASMA is under active development and currently provides many important LAPACK routines (and in particular the Cholesky factorization routine `dpotrf`) together with a tile and asynchronous version of all level 3 BLAS: this allows us to write the entire Riccati recursion algorithm in tile format.

6. NUMERICAL RESULTS

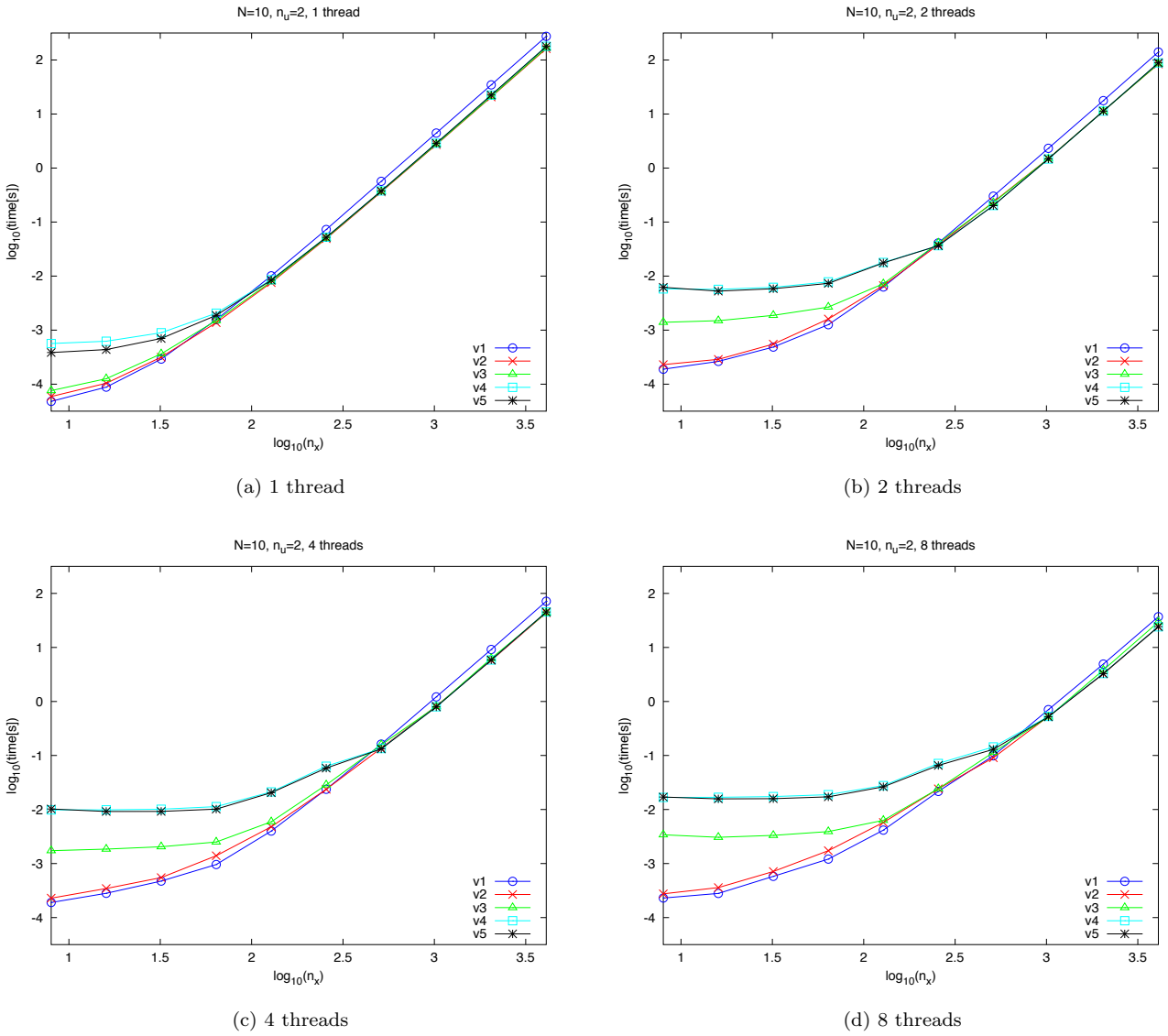
In this section we consider a number of parallel implementations of algorithms (2) and (3) on shared memory machines. We decided to test the following algorithms, that for simplicity we call `v1` to `v5`:

- **v1** implementation of algorithm (2), with BLAS and `dpotrf` provided by parallel OpenBLAS.
- **v2** implementation of algorithm (3), with BLAS and `dpotrf` provided by parallel OpenBLAS.
- **v3** implementation of algorithm (3), with BLAS provided by parallel OpenBLAS and `dpotrf` provided by PLASMA (that makes use of sequential OpenBLAS; in this case sequential and parallel OpenBLAS are given by the same library, and the switch between the two is made at run-time by means of `openblas_set_num_threads()`).
- **v4** implementation of algorithm (3), with level 3 BLAS and `dpotrf` provided by tile version of PLASMA (that makes use of sequential OpenBLAS).
- **v5** implementation of algorithm (3), with level 3 BLAS and `dpotrf` provided by tile and asynchronous version of PLASMA (that makes use of sequential OpenBLAS); routines working on independent data are gathered together into sets, and explicit barrier is used among sets.

The test machine is a HPC node equipped with dual Intel Xeon X5550 processor (in total 8 cores running at 2.66 GHz, 8 MB level 3 cache per socket) running Scientific Linux version 6.1. The processor supports the SSE, SSE2, SSSE3, SSE4_1, SSE4_2 instruction sets.

In figure (1) there are results of numerical tests. About the test problem, the linear system is a randomly-generated time-invariant asymptotically-stable one, while the cost function is strictly quadratic with identity as Hessian: anyway, the special structure of this test problem has not been exploited. In all tests only the number of states has been varied: we investigated the behavior of the proposed algorithms for $n_x \in \{4, 8, 16, 32, 64, 128, 256, 1024, 2048, 4096\}$. The number of inputs was fixed to $n_u = 2$ (its actual value does not influence the performance, as long as $n_u \ll n_x$), and the horizon length to $N = 10$ (its actual value does not influence the results of tests since Riccati recursion is linear in N).

The block size for the tile matrix layout in PLASMA has been chosen equal to $NB = 128$: this is a good trade off between fine-grid parallelism and performance of the sequential BLAS on matrices of size NB. For values of $n_x \leq NB$ the PLASMA routines clearly will reduce to a call to the sequential BLAS, with some overhead. Anyway the largest matrices, of size 4096, are decomposed into $16 \cdot 16 = 256$ blocks, enough to have a fine-grid parallelism.



(a) 1 thread

(b) 2 threads

(c) 4 threads

(d) 8 threads

Fig. 1. Comparison of the different implementations of the Riccati recursion based solver, for the solution of problem (1), for 1,2,4 or 8 threads. Problem size: $N = 10$, n_x varied, $n_u = 2$.

In the test in figure (1a) 1 thread was used. As expected, the implementations making use of PLASMA (i.e. `v3, v4, v5`) suffer a certain overhead for small matrices. For large matrices, all implementations of algorithm 3 (i.e. `v2, v3, v4, v5`) behave in a very similar way, and are faster than the implementation of algorithm 2 (i.e. `v1`), as expected from the theoretical complexity. Anyway, for very small problems, the latter is the fastest, due to the better performance of `dgemm` on small matrices compared to the others level 3 BLAS and `dpotrf` routines. The tile asynchronous implementation `v5` is always slightly faster than the tile synchronous one `v4`, and this is true also for a larger number of threads.

As the number of threads increases to 2, in figure (1b), the overhead associated with implementations making use of PLASMA (i.e. `v3, v4, v5`) increases of an order of magnitude, and it seems proportional to the number of PLASMA routines used per iteration (1 for `v3`, 9 for `v4` and `v5`). For $n_x \in \{256, 512\}$ the tile implementations `v4` and `v5` are slightly faster than the implementation `v2` making use of parallel OpenBLAS. Anyway for larger systems their performance is almost identical.

As the number of threads further increases to 4 (figure (1c)) and 8 (figure (1d)), the trend remains unchanged. In fact, the overhead associated with the use of PLASMA routines increases, and then they become competitive with respect to parallel OpenBLAS only for increasingly larger systems. For large n_x the performance of `v4` and `v5` is almost identical to the one of `v2`, while `v3` is slightly slower. Also the cross-over point between the parallel OpenBLAS implementations of algorithm 2 and algorithm 3 (respectively `v1` and `v2`) moves toward larger values of n_x, since `dgemm` (the key routine in `v1`) is particularly parallel friendly, while `dpotrf` (the key routine in `v2`) is not.

As a result, on the tested machine implementation `v2` making use of OpenBLAS and implementations `v4, v5` making use of PLASMA shows an almost identical per-

	number of threads			
n_x	1	2	4	8
4	0.89	0.58	0.58	0.56
8	0.81	0.82	0.83	0.84
16	0.85	0.92	0.81	0.78
32	0.92	0.88	0.86	0.81
64	1.13	0.78	0.69	0.70
128	1.34	0.94	0.83	0.72
256	1.48	1.08	1.00	0.90
512	1.58	1.28	1.18	1.09
1024	1.64	1.55	1.48	1.34
2048	1.68	1.55	1.60	1.52
4096	1.69	1.67	1.64	1.54

Fig. 2. Speed-up of `v2` with respect to `v1`, computed as $time_{v1}/time_{v2}$. Problem size: $N = 10$, $n_u = 2$.

formance. Anyway the result can be different on shared memory machines with more cores (e.g. PLASMA documentation reports test on machines with 16 or 32 cores). We also notice that, in case of loaded machine, PLASMA shows a smaller decrease in performance compared to OpenBLAS.

In the following we thus analyze more deeply the performance of implementations `v1` (implementing the classical version in algorithm 2) and `v2` (implementing the factorized version in algorithm 3), both making use of OpenBLAS.

In figure 2 there is a table showing the relative speed-up of implementation `v2` compared to `v1`, as function of the number of states n_x and the number of threads. For a given number of threads, implementation `v1` is more efficient for small n_x, while `v2` is more efficient for large n_x. The cross-over points moves toward lager values of n_x as the number of threads increases: this means that `v1` scales better with the number of threads compared to `v2`. Looking at the rows of the table, we can arrive at the same conclusion. In particular it is interesting to notice as, for $n_x = \{64, 128, 256\}$, implementation `v2` is faster in case of 1 thread, but slower in case of 8.

In figure 3 there is a table showing, for both `v1` and `v2`, the speedup obtained using more threads, with respect to the sequential code. The parallel code is faster than the sequential one for $n_x \geq 64$ for `v1`, and $n_x \geq 128$ for `v2`. The efficiency in the use of all available cores increases with the problem size, and again we notice as `v1` has a better scalability than `v2`.

7. CONCLUSION

In this paper we presented two version of Riccati recursion based solver for an extended formulation of the LQ control problems. Algorithm 2 has a worst theoretical complexity but it performs better for small instances; algorithm 3 has a better theoretical complexity, that gives it an advantage for large instances. As the number of threads increases, implementations of algorithm 2 scale better than implementations of algorithm 3. This is due to the fact that the key routine in algorithm 3, the Cholesky factorization, is not parallel friendly.

We tested a number of implementations of algorithm 3, one making use of OpenBLAS, 3 making use of PLASMA

	v1 number of threads			v2 number of threads		
n_x	2	4	8	2	4	8
4	0.61	0.59	0.61	0.40	0.39	0.39
8	0.25	0.25	0.21	0.26	0.26	0.21
16	0.33	0.31	0.31	0.36	0.30	0.29
32	0.60	0.61	0.50	0.57	0.57	0.44
64	1.24	1.63	1.29	0.86	0.99	0.80
128	1.61	2.53	2.43	1.13	1.58	1.31
256	1.77	3.07	3.36	1.30	2.08	2.04
512	1.88	3.50	5.75	1.53	2.63	3.97
1024	1.91	3.65	6.29	1.81	3.28	5.13
2048	1.96	3.78	7.03	1.81	3.60	6.38
4096	1.95	3.83	7.41	1.92	3.70	6.76

Fig. 3. Speed-up obtained using multiple threads, compared to sequential code. Problem size: $N = 10$, $n_u = 2$.

(in the combinations synchronous/asynchronous tile algorithms, and making use or not of the parallel level 3 BLAS provided by PLASMA). On the test machine (with 8 cores), the use of PLASMA does not give significant advantages with respect to OpenBLAS.

As future work, further tests may be performed on machines with a larger number of cores.

REFERENCES

Wright, S.J. (1997). Applying new optimization algorithms to model predictive control. *Fifth International Conference on Chemical Process Control CPC V*, 147-155. CACHE, Tahoe City, California.

Rao, C.V., Wright, S.J., and Rawlings, J.B. (1998). Application of interior-point methods to model predictive control. *Journal of Optimization Theory and Applications*, 99(3), 723-757.

Jørgensen, J.B., Rawlings, J.B., and Jørgensen, S.B. (2004). Numerical methods for large scale moving horizon estimation and control. In *DYCOPS 7*. IFAC, Cambridge, MA.

Frison, G. (2012). *Numerical Methods for Model Predictive Control*. M.Sc. thesis, Department of Informatics and Mathematical Modelling, Technical University of Denmark, Kgs. Lyngby, Denmark.

Jørgensen, J.B., Frison, G., Gade-Nielsen, N.F., Damman, B. (2012). Numerical Methods for Solution of the Extended Linear Quadratic Control Problem. *Proc. IFAC Conf. Nonlinear Model Predictive Control (NMPC'12)*. Noordwijkerhout, The Netherlands, 2012, pp. 187-193.

Jørgensen, J.B. (2005). *Moving Horizon Estimation and Control*. Ph.D. thesis, Department of Chemical Engineering, Technical University of Denmark, Kgs. Lyngby, Denmark.

Frison, G., Jørgensen, J.B. (2013). Efficient Implementation of the Riccati Recursion for Solving Linear-Quadratic Control Problems. Submitted to IEEE MSC 2013.

Chapter 3

Paper [33]

52nd IEEE Conference on Decision and Control
December 10-13, 2013. Florence, Italy

A Fast Condensing Method for Solution of Linear-Quadratic Control Problems

Gianluca Frison, John Bagterp Jørgensen

Abstract— **In both Active-Set (AS) and Interior-Point (IP) algorithms for Model Predictive Control (MPC), sub-problems in the form of linear-quadratic (LQ) control problems need to be solved at each iteration. The solution of these sub-problems is usually the main computational effort. In this paper we consider a condensing (or state elimination) method to solve an extended version of the LQ control problem, and we show how to exploit the structure of this problem to both factorize the dense Hessian matrix and solve the system. Furthermore, we present two efficient implementations. The first implementation is formally identical to the Riccati recursion based solver and has a computational complexity that is linear in the control horizon length and cubic in the number of states. The second implementation has a computational complexity that is quadratic in the control horizon length as well as the number of states. When the state dimension is high, this implementation is faster than the Riccati recursion based implementation.**

I. Introduction

The linear-quadratic (LQ) control problem can be considered the core problem in Model Predictive Control (MPC). In its classical form, it represents an unconstrained optimal control problem where the controlled system is linear time-invariant and the cost function is quadratic. This problem formulation is especially important because it arises as a sub-problem in Active-Set (AS) and Interior-Point (IP) algorithms for MPC [1]–[3]. The solution of these sub-problems is typically the main computational effort at each iteration, and this explains the need for efficient solvers.

From a mathematical point of view, the LQ control problem is an equality constrained quadratic program, and it can be solved by factorizing its KKT matrix with dense linear algebra. The cost of this approach is $\mathcal{O}(N^3(n_x + n_u)^3)$, where N is the control horizon length, n_x is the number of states and n_u is the number of controls (or inputs). However, the KKT system associated with the LQ control problem is sparse, and its special structured can be exploited to obtain more efficient solvers. Classical structure-exploiting solvers may be divided into two groups, depending on whether the states are considered as optimization variables or not.

In the first group, only the inputs are considered as optimization variables. The large, sparse KKT system is rewritten into a smaller, dense form, that can be solved using dense linear algebra. Since the Hessian of the dense formulation is positive definite, these solvers typically make use of dense Cholesky factorization to factorize the Hessian. The cost of this approach is $\mathcal{O}(N^3 n_u^3)$, plus the cost of the condensing phase. The solvers in this first group, that can be referenced as condensing (or state elimination) methods, can be used to solve problems with a short control horizon [5].

In the second group, also the states and co-states are considered as optimization variables, and larger systems where the sparsity is preserved are solved. Well known examples are general purpose sparse solvers, Riccati recursion based solver and Schur complement based solvers. The complexity of all solvers is $\mathcal{O}(N(n_x + n_u)^3)$, and they can be used to solve problems with a long control horizon [5].

A recent paper [6] presents a connection point between the two groups. The authors show that the Riccati recursion, traditionally used to efficiently factorize the large sparse KKT system, can also be used to exploit the remaining structure of the small dense Hessian of condensing methods, and to factorize it in time $\mathcal{O}(N^2)$ (instead of $\mathcal{O}(N^3)$ using the usual Cholesky factorization). However, this hybrid method will never be faster than the Riccati recursion, and it has a complexity that is quadratic in N and cubic in n_x.

In this paper, we consider a condensing method where the special structure of the LQ control problem is exploited not only in the factorization of the small dense Hessian matrix, but also in the solution of the system. Furthermore, we present two efficient implementations of this method: the one is formally identical (at least regarding the matrix-matrix operations) to the Riccati solver for the sparse KKT system, and then linear in N and cubic in n_x, while the other is quadratic in both N and n_x. This second implementation is faster than the Riccati solver in case of a large n_x and a moderate N.

The paper is organized as follows. Section II introduces an extension to the classical LQ control problem, and states necessary and sufficient conditions for its solution. In section III the small, dense formulation of the LQ control problem is computed by condensing the KKT system. Section IV presents a method for structure-exploiting factorization and system solution. In section V two efficient implementations of the method are presented, and results of numerical tests are presented in section VI. Finally, section VII contains the conclusion.

II. The extended LQ control problem

The extended LQ control problem is a generalization of the classical LQ control problem. The cost function has quadratic, linear and constant terms, and the constraints are affine. Furthermore, all matrices are time variant. Its structure is flexible enough to describe a wide range of problems [7].

G. Frison and J.B. Jørgensen are with Technical University of Denmark, DTU Compute - Department of Applied Mathematics and Computer Science, DK-2800 Kgs Lyngby, Denmark. {giaf , jbj} at imm.dtu.dk

In particular, it can be used as a sub-routine in AS and IP methods.

Problem 1: The extended LQ control problem is the equality constrained quadratic program

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

where $n \in \{0, 1, \ldots, N-1\}$ and

$$l_n(x_n, u_n) = \frac{1}{2} \begin{bmatrix} x_n' & u_n' \end{bmatrix} \begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \begin{bmatrix} q_n' & s_n' \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \rho_n$$

$$l_N(x_N) = \frac{1}{2} x_N' P_N x_N + p_N' x_N + \rho_N$$

The state vector x_n has size n_x, the input vector u_n has size n_u, and N is the control horizon length.

Problem (1) can be rewritten in a more compact form as

$$\begin{aligned} \min_{y} \quad & \phi = \frac{1}{2} y' \mathcal{H} y + g' y \\ s.t. \quad & \mathcal{A} y = b \end{aligned} \tag{2}$$

where (for $N = 3$)

$$y = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \hline x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} u \\ \hline x \end{bmatrix}, \quad g = \begin{bmatrix} \tilde{s}_0 \\ s_1 \\ s_2 \\ \hline q_1 \\ q_2 \\ p_3 \end{bmatrix} = \begin{bmatrix} s \\ \hline q \end{bmatrix}, \quad b = \begin{bmatrix} \tilde{b}_0 \\ b_1 \\ b_2 \end{bmatrix},$$

$$\mathcal{H} = \left[\begin{array}{ccc|ccc} R_0 & & & S_1 & & \\ & R_1 & & & S_2 & \\ & & R_2 & & & \\ \hline S_1' & & & Q_1 & & \\ & S_2' & & & Q_2 & \\ & & & & & P_3 \end{array}\right] = \left[\begin{array}{c|c} \bar{R}_N & \bar{S}_N \\ \hline \bar{S}_N' & \bar{Q}_N \end{array}\right],$$

$$\mathcal{A} = \left[\begin{array}{ccc|ccc} -B_0 & & & I & & \\ & -B_1 & & -A_1 & I & \\ & & -B_2 & & -A_2 & I \end{array}\right] = \left[\begin{array}{c|c} -\bar{B}_N & \bar{A}_N \end{array}\right],$$

and $\tilde{s}_0 = S_0 x_0 + s_0$ and $\tilde{b}_0 = A_0 x_0 + b_0$.

Theorem 1 (KKT (necessary) conditions): If y^* is a solution of problem (2), then there exists a vector π^* of size $N \cdot n_x$ such that

$$\begin{bmatrix} \mathcal{H} & -\mathcal{A}' \\ -\mathcal{A} & 0 \end{bmatrix} \begin{bmatrix} y^* \\ \pi^* \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix}. \tag{3}$$

Proof: See [4]. ∎

System (3) is the KKT system associated with problem (2), and in the case of the extended LQ control problem the KKT matrix is large (of size $(2n_x + n_u)N \times (2n_x + n_u)N$) and sparse.

Theorem 2 (Sufficient conditions): Let the matrices P_N and $\begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix}$ be positive semi-definite, and the matrices R_n be positive definite for all $n \in \{0, 1, \ldots, N-1\}$, then problem (2) has one and only one solution, given by the solution of the KKT system (3).

Proof: See [5]. ∎

In the following, we assume that conditions holds such that theorems 1 and 2 apply. We also assume that the matrices P_N, Q_n and R_n are symmetric.

III. Condensing of the KKT system

There exist several methods to solve the KKT system (3). In this paper we consider a condensing (or state elimination) method. It consists in the elimination of the explicit dependence of the KKT system (or equivalently of problem (1)) from x and π. The result is a smal dense system of linear equations.

By using the definition of y, the KKT system (3) can be re-written as

$$\begin{bmatrix} \bar{R}_N & \bar{S}_N & \bar{B}_N' \\ \bar{S}_N' & \bar{Q}_N & -\bar{A}_N' \\ \bar{B}_N & -\bar{A}_N & 0 \end{bmatrix} \begin{bmatrix} u \\ x \\ \pi \end{bmatrix} = - \begin{bmatrix} s \\ q \\ b \end{bmatrix}. \tag{4}$$

Since the squared matrix $\bar{A}_N$ has full rank, it is invertible, and then it is possible to eliminate π from (4), obtaining

$$\begin{bmatrix} \bar{R}_N + \Gamma_N' \bar{S}_N' & \bar{S}_N + \Gamma_N' \bar{Q}_N \\ \bar{B}_N & -\bar{A}_N \end{bmatrix} \begin{bmatrix} u \\ x \end{bmatrix} = - \begin{bmatrix} s + \Gamma_N' q \\ b \end{bmatrix} \tag{5}$$

where $\Gamma_N = \bar{A}_N^{-1} \bar{B}_N$ is lower block-triangular. Again, since $\bar{A}_N$ is invertible, it is possible to eliminate x from (5), obtaining

$$(H_N) u = -f \tag{6}$$

where

$$H_N = \bar{R}_N + \Gamma_N' \bar{S}_N' + \bar{S}_N \Gamma_N + \Gamma_N' \bar{Q}_N \Gamma_N \tag{7}$$

$$f = s + \Gamma_N' q + (\bar{S}_N + \Gamma_N' \bar{Q}_N) \bar{A}_N^{-1} b \tag{8}$$

The dense KKT matrix H_N (that is also the Hessian of the dense formulation of problem (1)) is small (of size $Nn_u \times Nn_u$) and dense. Notice that $\bar{S}_N \Gamma_N$ is lower block-triangular, with zero block-diagonal.

According to theorem 2, the matrix H_N is positive definite [5], and it can be factorized using Cholesky factorization.

IV. Derivation

In this section, we present a structure-exploiting procedure to factorize the Hessian matrix H_N and solve system (6). There are two equivalent approaches: a Cholesky-like factorization of the matrix H_N, or a standard Cholesky factorization of a properly permuted matrix $\hat{H}_N$ (see appendix A). We prefer the second approach for the reason that it requires only standard software (e.g. BLAS and LAPACK, of which there exist highly optimized implementations).

Let us consider the permutation matrix (for $N = 3$)

$$\mathcal{M}_x = \mathcal{M}_x' = \mathcal{M}_x^{-1} = \begin{bmatrix} & & I_x \\ & I_x & \\ I_x & & \end{bmatrix}$$

of size Nn_x (where I_x is the identity matrix of size n_x) and the permutation matrix $\mathcal{M}_u$ of size Nn_u (where the identity matrices I_u have size n_u). The permuted system is

$$\hat{H}_N \hat{u} = (\mathcal{M}_u H_N \mathcal{M}_u)(\mathcal{M}_u u) = -\mathcal{M}_u f = -\hat{f}.$$

In the following we will use the hat to indicate permuted matrices and vectors.

A. Structure-exploiting factorization of $\hat{H}_N$

In this part we describe a procedure equivalent to the Cholesky factorization of the matrix $\hat{H}_N$, but that requires less than $\mathcal{O}(N^3)$ flops. At each iteration, one block-row is factorized, and the correction of the part of the matrix that has not yet been factorized (see appendix A) is substituted with the update of one of the matrices Q_n.

The permuted problem matrices are (for $N = 3$)

$$\hat{Q}_N = \begin{bmatrix} P_3 & & \\ & Q_2 & \\ & & Q_1 \end{bmatrix}, \hat{S}_N = \begin{bmatrix} 0 & S_2 & \\ & 0 & S_1 \\ & & 0 \end{bmatrix},$$
$$\hat{R}_N = \begin{bmatrix} R_2 & & \\ & R_1 & \\ & & R_0 \end{bmatrix}, \hat{q} = \begin{bmatrix} p_3 \\ q_2 \\ q_1 \end{bmatrix}, \hat{s} = \begin{bmatrix} s_2 \\ s_1 \\ s_0 \end{bmatrix}, \hat{u} = \begin{bmatrix} u_2 \\ u_1 \\ u_0 \end{bmatrix},$$
$$\hat{A}_N = \begin{bmatrix} I & -A_2 & \\ & I & -A_1 \\ & & I \end{bmatrix}, \hat{B}_N = \begin{bmatrix} B_2 \\ B_1 \\ B_0 \end{bmatrix}, \hat{b} = \begin{bmatrix} b_2 \\ b_1 \\ b_0 \end{bmatrix}.$$

We define the matrix

$$\hat{Q}_N^* = \begin{bmatrix} Q_3^* & & \\ & Q_2 & \\ & & Q_1 \end{bmatrix}$$

where $Q_3^* = P_3$.

The permuted matrix $\hat{H}_N$ and vector $\hat{f}$ take the form

$$\hat{H}_N = \hat{R}_N + \hat{\Gamma}'_N \hat{S}'_N + \hat{S}_N \hat{\Gamma}_N + \hat{\Gamma}'_N \hat{Q}_N^* \hat{\Gamma}_N \quad (9)$$
$$\hat{f} = \hat{s} + \hat{\Gamma}'_N \hat{q} + (\hat{S}_N + \hat{\Gamma}'_N \hat{Q}_N^*) \hat{A}_N^{-1} \hat{b} \quad (10)$$

where $\hat{\Gamma}_N = \hat{A}_N^{-1} \hat{B}_N$ is block upper triangular.

We want to emphasize the structure of the first block-row by decomposing all matrices. In particular, the matrix $\hat{A}_N$ is

$$\hat{A}_N = \begin{bmatrix} I_x & -A_{N-1}\mathcal{E}_{N-1} \\ & \hat{A}_{N-1} \end{bmatrix}$$

where $\mathcal{E}_{N-1} = \begin{bmatrix} I_x & O \end{bmatrix}$, where O is a zero matrix of size $nx \times ((N-1)-1)n_x$ and I_x is an identity matrix of size n_x. Notice that $\hat{A}_1 = I_x$. The inverse $\hat{A}_N^{-1}$ is

$$\hat{A}_{N-1}^{-1} = \begin{bmatrix} I_x & A_{N-1}\mathcal{E}_{N-1}\hat{A}_{N-1}^{-1} \\ & \hat{A}_{N-1}^{-1} \end{bmatrix}$$

and then $\hat{\Gamma}_N = \hat{A}_N^{-1}\hat{B}_N$ is

$$\hat{\Gamma}_N = \begin{bmatrix} B_{N-1} & A_{N-1}\hat{\mathcal{E}}_{N-1}\hat{\Gamma}_{N-1} \\ & \hat{\Gamma}_{N-1} \end{bmatrix}.$$

Notice that $\hat{\Gamma}_1 = B_0$. Similarly for $\hat{Q}_N^*$, $\hat{R}_N$ and $\hat{S}_N$

$$\hat{Q}_N = \begin{bmatrix} Q_N^* & \\ & \hat{Q}_{N-1} \end{bmatrix}, \quad \hat{R}_N = \begin{bmatrix} R_{N-1} & \\ & \hat{R}_{N-1} \end{bmatrix}$$
$$\hat{S}_N = \begin{bmatrix} 0 & S_{N-1}\mathcal{E}_{N-1} \\ & \hat{S}_{N-1} \end{bmatrix}$$

The block upper triangular part of the matrix $\hat{\Gamma}'_N \hat{Q}_N^* \hat{\Gamma}_N$ is (using $3 = N$ and $2 = N-1$ for space issues)

$$\hat{\Gamma}'_3 \hat{Q}_3^* \hat{\Gamma} = \begin{bmatrix} B_2'Q_3^*B_2 & B_2'Q_3^*A_2\mathcal{E}_2\hat{\Gamma}_2 \\ * & \hat{\Gamma}'_2\hat{Q}_2\hat{\Gamma}_2 + \hat{\Gamma}'_2\mathcal{E}'_2A'_2Q_3^*A_2\mathcal{E}_2\Gamma_2 \end{bmatrix}.$$

Similarly, the matrix $\hat{S}_N\hat{\Gamma}_N$ is

$$\hat{S}_2\hat{\Gamma}_3 = \begin{bmatrix} 0 & S_2\mathcal{E}_2\hat{\Gamma}_2 \\ & \hat{S}_2\hat{\Gamma}_2 \end{bmatrix}$$

In the following we consider only the block upper triangular part of the matrix $\hat{H}_N$, that is

$$\hat{H}_3 = \begin{bmatrix} H_{11} & H_{12} \\ * & H_{22} \end{bmatrix} = \hat{R}_3 + \hat{S}_3\hat{\Gamma}_3 + \hat{\Gamma}'_3\hat{Q}_3^*\hat{\Gamma}_3 =$$
$$\begin{bmatrix} R_2 + B_2'Q_3^*B_2 & (S_2 + B_2'Q_3^*A_2)\hat{\mathcal{E}}_2\hat{\Gamma}_2 \\ * & \hat{R}_2 + \hat{S}_2\hat{\Gamma}_2 + \hat{\Gamma}'_2\hat{Q}_2\hat{\Gamma}_2 + \hat{\Gamma}'_2\hat{\mathcal{E}}'_2A'_2Q_3^*A_2\hat{\mathcal{E}}_2\hat{\Gamma}_2 \end{bmatrix}$$

The symmetric positive definite matrix $D_{N-1} = H_{11}$ can be factorized using the Cholesky factorization, as

$$D_{N-1} = R_{N-1} + B'_{N-1}Q_N^*B_{N-1} = U'_{N-1}U_{N-1}. \quad (11)$$

where U_{N-1} is the U_{11} matrix in (27). The rectangular matrix U_{12} in (27) is obtained as

$$U_{12} = (U'_{N-1})^{-1}H_{12} = L_{N-1}\mathcal{E}_{N-1}\hat{\Gamma}_{N-1}$$

where

$$L_{N-1} = (U'_{N-1})^{-1}M_{N-1}. \quad (12)$$

and

$$M_{N-1} = S_{N-1} + B'_{N-1}Q_N^*A_{N-1} \quad (13)$$

The correction term $-U'_{12}U_{12}$ is

$$-U'_{12}U_{12} = -\hat{\Gamma}'_{N-1}\mathcal{E}'_{N-1}L'_{N-1}L_{N-1}\mathcal{E}_{N-1}\hat{\Gamma}_{N-1}$$

and then the corrected bottom-right matrix is

$$H_{22} - U'_{12}U_{12} = \hat{R}_{N-1} + \hat{S}_{N-1}\hat{\Gamma}_{N-1} + \hat{\Gamma}'_{N-1}\hat{Q}_{N-1}^*\hat{\Gamma}_N + \\ + \hat{\Gamma}'_{N-1}\mathcal{E}'_{N-1}A'_{N-1}Q_N^*A_{N-1}\mathcal{E}_{N-1}\hat{\Gamma}_{N-1} \quad (14)$$

where

$$\hat{Q}_{N-1}^* = \begin{bmatrix} Q_{N-1}^* & \\ & \hat{Q}_{N-2} \end{bmatrix}$$

and

$$Q_{N-1}^* = Q_{N-1} - L'_{N-1}L_{N-1}. \quad (15)$$

Notice that the corrected term $H_{22}^* = H_{22} - U'_{12}U_{12}$, is exactly in the same form as H_{22}, with the corrected matrix Q_{N-1}^* in place of Q_{N-1}. The correction of the not-factorized-yet matrix H_{22} is equivalent to the correction of Q_{N-1}^*, and the use of the latter to compute H_{22}^*.

The procedure to factorize one row of the matrix $\hat{H}_N$ reduces to the factorization of the block-diagonal element $H_{11} = U'_{N-1}U_{N-1}$, the computation of the remaining of the row by solving the triangular system $U_{12} = (U'_{N-1})^{-1}H_{12}$, and the computation of $Q_{N-1}^* = Q_{N-1} - L'_{N-1}L_{N-1}$ in place of the correction term $-U'_{12}U_{12}$.

The upper Cholesky factor computed so far is

$$\begin{bmatrix} U_{11} & U_{12} \\ & H_{22} - U'_{12}U_{12} \end{bmatrix} = \begin{bmatrix} U_{N-1} & L_{N-1}\mathrm{row}_1(\hat{\Gamma}_{N-1}) \\ & H_{22}^* \end{bmatrix}$$

where $\mathrm{row}_1(\hat{\Gamma}_{N-1}) = \mathcal{E}_{N-1}\hat{\Gamma}_{N-1}$ is the first block row of $\hat{\Gamma}_{N-1}$). Notice that the first-block row is factorized, and the matrix H_{22}^* can be factorized by repeating the same procedure.

Let us define the matrix $\hat{H}_N^*$ as

$$\hat{H}_N^* = \hat{R}_N + \hat{S}_N\hat{\Gamma}_N + \hat{\Gamma}'_N\hat{S}'_N + \hat{\Gamma}'_N\hat{\mathcal{Q}}_N^*\hat{\Gamma}_N$$

where (for $N = 3$)

$$\hat{\mathcal{Q}}_N^* = \begin{bmatrix} Q_3^* & & \\ & Q_2^* & \\ & & Q_1^* \end{bmatrix}.$$

The Cholesky factorization of the matrix $\hat{H}_N$ is then equivalent to the following procedure on the matrix $\hat{H}_N^*$: for each block-row, Cholesky factorization of the block-diagonal element, and triangular system solution to compute the remaining of the block-row.

At the end of the factorization procedure, the upper block-triangular factor $\hat{\mathcal{U}}$ is (for $N = 3$)

$$\hat{\mathcal{U}} = \begin{bmatrix} U_2 & L_2B_1 & L_2A_1B_0 \\ & U_1 & L_1B_0 \\ & & U_0 \end{bmatrix} = \hat{U}_N + \hat{L}_N\hat{A}_N^{-1}\hat{B}_N \quad (16)$$

where

$$\hat{U}_N = \begin{bmatrix} U_2 & & \\ & U_1 & \\ & & U_0 \end{bmatrix}, \quad \hat{L}_N = \begin{bmatrix} 0 & L_2 & \\ & 0 & L_1 \\ & & 0 \end{bmatrix}. \quad (17)$$

Notice that the matrix $\hat{L}_N\hat{A}_N^{-1}\hat{B}_N$ is upper block-triangular, with zero block-diagonal.

B. Structure-exploiting system solution

The factorized system

$$\hat{\mathcal{U}}'\hat{\mathcal{U}}\hat{u} = -\hat{f} \quad (18)$$

may be solved using forward and backward substitutions. The input vector is then computed as $u = \mathcal{M}_u\hat{u}$. Anyway also in this case it is possible to exploit the form of the problem, and in particular of the upper factor $\hat{\mathcal{U}}$ in (16).

System (18) may be rewritten as

$$\hat{\mathcal{U}}'\hat{y} = -\hat{g} \quad (19)$$

where we define

$$\hat{\mathcal{U}}\hat{u} = \hat{y}. \quad (20)$$

The first step is then the solution of the lower block-triangular system (19) using forward substitution. Inserting (16) in (19) we have

$$\hat{y} = -(\hat{U}'_N)^{-1}\left(\hat{g} + \hat{B}'_N(\hat{A}'_N)^{-1}\hat{L}'_N\hat{y}\right) \quad (21)$$

that in the case $N = 3$ looks like

$$\begin{bmatrix} y_2 \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} -(U'_2)^{-1}(g_2) \\ -(U'_1)^{-1}(g_1 + B'_1L'_2y_2) \\ -(U'_0)^{-1}(g_0 + B'_0A'_1L'_2y_2 + B'_0L'_1y_1) \end{bmatrix}.$$

The second step is the solution of the upper block-tridiagonal system (20) using backward substitution. Inserting (16) in (20) we have

$$\hat{u} = \hat{U}_N^{-1}\left(\hat{y} - \hat{L}_N\hat{A}_N^{-1}\hat{B}_N\hat{u}\right) \quad (22)$$

that in the case $N = 3$ looks like

$$\begin{bmatrix} u_2 \\ u_1 \\ u_0 \end{bmatrix} = \begin{bmatrix} U_2^{-1}(y_2 - L_2B_1u_1 - L_2A_1B_0u_0) \\ U_1^{-1}(y_1 - L_1B_0u_0) \\ U_0^{-1}(y_0) \end{bmatrix}.$$

Equations (21) and (22) imply that it is not necessary to explicitly build the $\hat{\mathcal{U}}$ matrix (16). Solution of (18) only requires the computation of the matrices $\hat{U}_N$ and $\hat{L}_N$ in (17).

C. Computational cost

The cost of the matrix-matrix operations in the factorization and solution of system (18) is then

$$\tfrac{1}{3}Nn_u^3 + (N-1)n_xn_u^2 + (N-1)n_x^2n_u \quad (23)$$

flops, where the first term comes from the Cholesky factorization of the block-diagonal elements (11), the second term from the solution of system (12) and the third term from the computation of (15).

The cost to solve problem (1) is $\frac{1}{3}Nn_u^3 + (N-1)n_xn_u^2 + (N-1)n_x^2n_u$ plus the cost to build the matrix $\hat{H}_N^*$.

V. Implementation

In this section, we present two implementations of the procedure described in section IV, characterized by different asymptotic complexity. The basic difference between the two implementations is the procedure to build the matrix $\hat{H}_N^*$.

The first procedure is formally equivalent to the Riccati solver for the sparse KKT system, and then problem (1) can be solved in time $\mathcal{O}(N(n_x + n_u)^3)$.

The second procedure can be used to solve problem (1) in time $\mathcal{O}(N^2n_x^2n_u + Nn_xn_u^2 + Nn_u^3)$, and it is faster than Riccati recursion if roughly $n_x > Nn_u$.

The key to build the matrix $\hat{H}_N^*$ is the exploitation of the special structure of $\hat{A}_N^{-1}$ (see appendix B).

A. Riccati-like solver

The key idea of this implementation is to perform a number of operations linear in N. This solver is formally identical to the Riccati solver for the sparse KKT system: the only difference is in the matrix-vector operations.

Let us define $P_N = Q_N$. The upper block-triangular part of the matrix $\hat{H}_N^*$ is (for $N = 3$)

$$\hat{H}_3^* = \hat{R}_3 + \hat{S}_3\hat{\Gamma}_3 + \hat{\Gamma}'_3\hat{Q}_3^*\hat{\Gamma}_3 = \begin{bmatrix} H_{11} & H_{12} \\ * & H_{22} \end{bmatrix} =$$

$$\begin{bmatrix} R_2 + B'_2P_3B_2 & (S_2 + B'_2P_3A_2)\hat{\mathcal{E}}_2\hat{\Gamma}_2 \\ * & \hat{R}_2 + \hat{S}_2\hat{\Gamma}_2 + \hat{\Gamma}'_2\hat{Q}_2^*\hat{\Gamma}_2 + \hat{\Gamma}'_2\hat{\mathcal{E}}'_2A'_2P_3A_2\hat{\mathcal{E}}_2\hat{\Gamma}_2 \end{bmatrix} =$$

$$\begin{bmatrix} R_2 + B'_2P_3B_2 & (S_2 + B'_2P_3A_2)\hat{\mathcal{E}}_2\hat{\Gamma}_2 \\ * & \hat{R}_2 + \hat{S}_2\hat{\Gamma}_2 + \hat{\Gamma}'_2\hat{P}_2\hat{\Gamma}_2 \end{bmatrix}$$

where $\hat{P}_{N-1}$ is $\hat{Q}_{N-1}^*$ with P_{N-1} in place of Q_{N-1}^*, and

$$\begin{aligned} P_{N-1} &= Q_{N-1}^* + A'_{N-1}P_NA_{N-1} = \\ &= Q_{N-1} + A'_{N-1}P_NA_{N-1} - L'_{N-1}L_{N-1}. \end{aligned} \quad (24)$$

Notice that (24) is the well-known Riccati recursion, and that with this definition matrix H_{22} is in the exact same form as $\hat{H}_N^*$. As a consequence the procedure can be repeated

for the matrix $H_{22} \doteq \hat{H}^*_{N-1}$. The recursion ends for $\hat{H}_1 = R_0 + B_0' P_1 B_0$.

The matrix $\hat{H}^*_N$ can be build in $\frac{7}{3}n_x^3 + 3n_x^2 n_u + n_x n_u^2$ flops. In fact, as shown in [8], the term $A_n' P_{n+1} A_n$ can be computed in $\frac{7}{3}n_x^3$ flops, the matrix $D_n = R_n + B_n' P_{n+1} B_n$ in $n_x^2 n_u + n_x n_u^2$ flops, and the matrix $M_n = S_n + B_n' P_{n+1} A_n$ in $2n_x^2 n_u$ flops.

Adding the costs (23), problem (1) can be solved with a computational cost of roughly

$$N\left(\tfrac{7}{3}n_x^3 + 4n_x^2 n_u + 2n_x n_u^2 + \tfrac{1}{3}n_u^3\right) \tag{25}$$

flops. That is the exact same cost of the Riccati recursion implementation in [8].

B. Pure condensing solver

The key idea of this implementation is to avoid operations cubic in n_x, even at the cost of more operations in N. This solver is efficient for large values of n_x.

The matrix $\hat{\Gamma}_N$ is formed explicitly, and it can be computed efficiently by using a multiplication-cascade procedure, in $\frac{N(N-1)}{2} 2n_x^2 n_u \approx N^2 n_x^2 n_u$ flops. For $N = 3$, it is like

$$\hat{\Gamma}_N = \begin{bmatrix} B_2 & A_2 B_1 & A_2 A_1 B_0 \\ & B_1 & A_1 B_0 \\ & & B_0 \end{bmatrix}.$$

We do not have to explicitly compute $\hat{H}^*_N$, but we only need $\hat{D}_N$ and $\hat{M}_N$. $\hat{D}_N$ can be computed in time $\mathcal{O}(N^2)$ as

$$\hat{D}_N = \hat{R}_N + \hat{B}_N' \cdot \operatorname{diag}\left((\hat{A}_N')^{-1}(\hat{Q}_N^* \hat{\Gamma}_N)\right).$$

Notice that matrix $\hat{D}_N$ is build one row at a time, as soon as the updated matrices Q_n^* are computed. The procedure requires $\frac{N(N+1)}{2} 2n_x^2 n_u$ flops for the computation of $\hat{Q}_N^* \cdot \hat{\Gamma}_N$, $\frac{N(N-1)}{2} 2n_x^2 n_u$ flops for the computation of the upper block-triangular part of $(\hat{A}_N')^{-1} \cdot (\hat{Q}_N^* \hat{\Gamma}_N)$, and $N 2 n_x n_u^2$ for the computation of $\hat{B}_N' \cdot \operatorname{diag}(\dots)$. This procedure requires $\mathcal{O}(N)$ function calls to the BLAS routine `dgemm`. The overall cost is roughly $2N^2 n_x^2 n_u + 2N n_x n_u^2$ flops.

$\hat{M}_N$ can be computed as

$$\hat{M}_N = \hat{S}_N + \left(\operatorname{diag}\left((\hat{A}_N')^{-1}(\hat{Q}_N^* \hat{\Gamma}_N)\right)\right)' \cdot \hat{\mathcal{A}}_N$$

in $2(N-1)n_x^2 n_u$ flops, where for $N = 3$

$$\hat{\mathcal{A}}_N = \begin{bmatrix} 0 & A_2 & \\ & 0 & A_1 \\ & & 0 \end{bmatrix}.$$

The cost to solve problem (1) by using this solver is roughly

$$2N^2 n_x^2 n_u + 3N n_x n_u^2 + \tfrac{1}{3} N n_u^3 \tag{26}$$

flops, plus the computation of $\hat{\Gamma}_N$, requiring $N^2 n_x^2 n_u$ flops, but that can be performed off-line in an IP method.

VI. Numerical results

The two implementations presented in section V have been implemented in C code and compared to each other. The tests have been performed on a laptop equipped with Intel i5-2410M @ 2.30 GHz, running Ubuntu 12.04 version 64 bit. OpenBLAS version 0.2.4 provides the BLAS and LAPACK libraries. The number of threads is set to one.

As test problem, we used a randomly generated linear system, while in the cost function Q_n and R_n are identities and S_n, s_n and q_n are zero matrices.

Figure 1a shows the computation time as function of n_x. For large values of n_x, the pure condensing solver is faster than the Riccati-like one, since it is quadratic instead of cubic in n_x.

Figure 1b shows the computation time as function of n_u. The pure condensing solver is almost linear for a wide range of values of n_u: in fact, the dominant term in (26) is $2N^2 n_x^2 n_u$, unless n_u is really large. The Riccati-like solver is almost insensitive to the value of n_u as long as $n_u < n_x$, but it becomes quickly cubic in n_u as soon as $n_u > n_x$.

Figure 1c shows the computation time as function of N. As expected, the Riccati-like solver is linear in N, while the pure condensing solver is quadratic in N.

VII. Conclusion

In this paper we present a method for the solution of (1) that is based on condensing (i.e. state elimination). The method exploits the special form of problem (1) in both factorization of matrix (9) and solution of system (18). This method is interesting from a theoretical point of view, and furthermore it leads to two efficient implementations. The one is formally identical to the Riccati solver for the sparse KKT system (linear in N and cubic in n_x). The other (quadratic in both N and n_x) is faster than the Riccati solver for large n_x and moderate N.

APPENDIX

A. Cholesky factorization

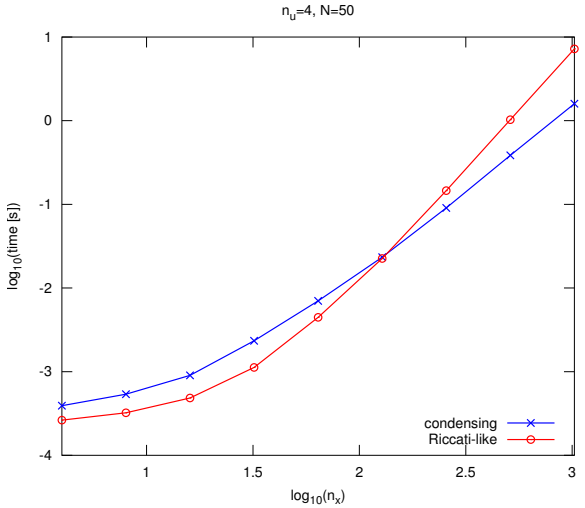
The standard Cholesky factorization is used to factorize a symmetric positive definite matrix H in the form $H = LU$, where the left matrix L is lower triangular, the right matrix U is upper triangular and $U = L'$ (or $L = U'$).

A basic algorithm to compute the upper triangular Cholesky factor is found by considering the expression

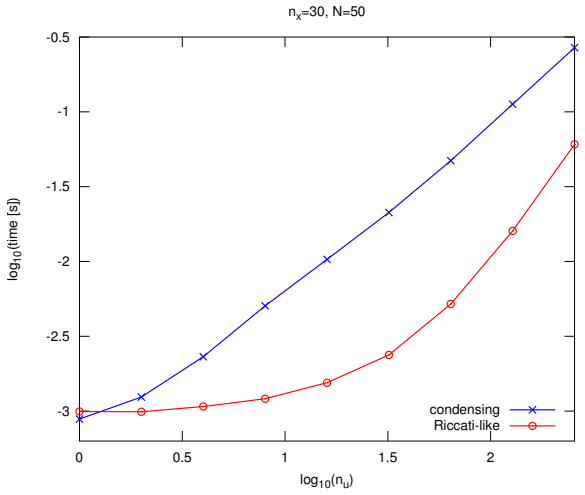
$$\begin{aligned} H &= \begin{bmatrix} H_{11} & H_{12} \\ H_{21} & H_{22} \end{bmatrix} = U'U = \begin{bmatrix} U_{11}' & \\ U_{12}' & U_{22}' \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ & U_{22} \end{bmatrix} \\ &= \begin{bmatrix} U_{11}' U_{11} & U_{11}' U_{12} \\ U_{12}' U_{11} & U_{22}' U_{22} + U_{12}' U_{12} \end{bmatrix} \end{aligned} \tag{27}$$

The factorization procedure consist in the factorization of H_{11} to obtain U_{11}, the solution of the triangular system $U_{12} = (U_{11}')^{-1} H_{12}$ to obtain U_{12}, the computation of the correction term $-U_{12}' U_{12}$, and the factorization of the corrected term $H_{22}^* = H_{22} - U_{12}' U_{12}$ to obtain U_{22}.

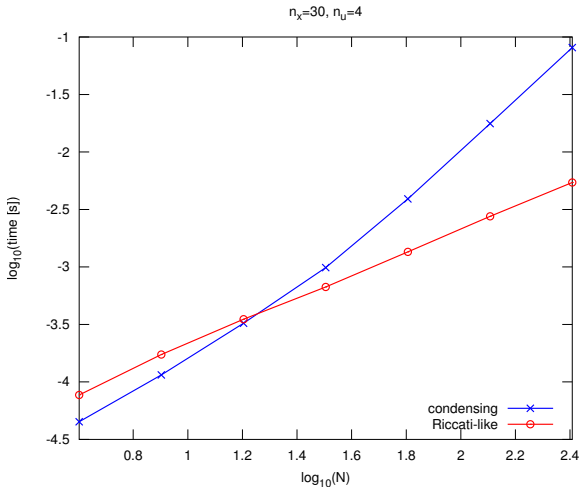
The 'Cholesky-like factorization' is a factorization analogue to Cholesky one, with the difference that the matrix



(a) n_x varied, $n_u = 4$, $N = 50$



(b) $n_x = 30$, n_u varied, $N = 50$



(c) $n_x = 30$, $n_u = 4$, N varied

Fig. 1: Time to solve problem (1) by using the Riccati-like (red) and condensing (blue) solvers described in this paper.

H is factorized in the form $H = UL$, where this time the left matrix U is upper triangular and the right matrix L is lower triangular, while as usual $U = L'$ (or $L = U'$).

A basic algorithm to compute the lower triangular Cholesky like factor is found by considering the expression

$$H = \begin{bmatrix} H_{11} & H_{21}' \\ H_{21} & H_{22} \end{bmatrix} = L'L = \begin{bmatrix} L_{11}' & L_{21}' \\ & L_{22}' \end{bmatrix} \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} = \\ = \begin{bmatrix} L_{11}'L_{11} + L_{21}'L_{21} & L_{21}'L_{22} \\ L_{22}'L_{21} & L_{22}'L_{22} \end{bmatrix}.$$

B. Structure of $\hat{A}_N^{-1}$

The shape of the matrix $\hat{A}_N^{-1}$ plays an important role in the method considered in this paper. For $N = 3$, it looks like

$$\hat{A}_N^{-1} = \begin{bmatrix} I & -A_2 & \\ & I & -A_1 \\ & & I \end{bmatrix}^{-1} = \begin{bmatrix} I & A_2 & A_2A_1 \\ & I & A_1 \\ & & I \end{bmatrix}.$$

As we can see, $\hat{A}_N$ is sparse (it has $2N-1$ block-elements), while $\hat{A}_N^{-1}$ is full (it has $\frac{N(N+1)}{2}$ block-elements).

Let us consider the product $\hat{\alpha} = \hat{A}_N\hat{\beta}$:

$$\begin{bmatrix} \alpha_2 \\ \alpha_1 \\ \alpha_0 \end{bmatrix} = \begin{bmatrix} I & -A_2 & \\ & I & -A_1 \\ & & I \end{bmatrix} \begin{bmatrix} \beta_2 \\ \beta_1 \\ \beta_0 \end{bmatrix} = \begin{bmatrix} \beta_2 - A_2\beta_1 \\ \beta_1 - A_1\beta_0 \\ \beta_0 \end{bmatrix}.$$

This means that a term in the form $\hat{\beta} = \hat{A}_N^{-1}\hat{\alpha}$ can be computed in N steps by using the backward recursion

$$\begin{bmatrix} \beta_2 \\ \beta_1 \\ \beta_0 \end{bmatrix} = \begin{bmatrix} \alpha_2 + A_2\beta_1 \\ \alpha_1 + A_1\beta_0 \\ \alpha_0 \end{bmatrix}. \tag{28}$$

Similarly, a term in the form $\hat{\beta} = (\hat{A}_N')^{-1}\hat{\alpha}$ can be computed in N steps by using the forward recursion

$$\begin{bmatrix} \beta_2 \\ \beta_1 \\ \beta_0 \end{bmatrix} = \begin{bmatrix} \alpha_2 \\ \alpha_1 + A_2'\beta_2 \\ \alpha_0 + A_1'\beta_1 \end{bmatrix}. \tag{29}$$

REFERENCES

[1] S.J. Wright (1997). Applying new optimization algorithms to model predictive control. *Proceedings of the Fifth International Conference on Chemical Process Control*, 147-155. CACHE Pubblications.

[2] C.V. Rao, S.J. Wright, and J.B. Rawlings (1998). Application of interior-point methods to model predictive control. *Journal of Optimization Theory and Applications*, 99(3), 723-757

[3] J.B. Jørgensen, J.B. Rawlings, and S.B. Jørgensen (2004). Numerical methods for large scale moving horizon estimation and control. In *DYCOPS 7*. IFAC, Cambridge, MA.

[4] J. Nocedal and S. J. Wright, *Numerical Optimization*. Springer, 1999.

[5] G. Frison (2012). *Numerical Methods for Model Predictive Control*. M.Sc. thesis, Department of Informatics and Mathematical Modelling, Technical University of Denmark, Kgs. Lyngby, Denmark.

[6] D. Axehill, M. Morari (2012). An alternative use of the Riccati recursion for efficient optimization. In *Systems & Control Letters*, 61, 37-40.

[7] J.B. Jørgensen, G. Frison, N.F. Gade-Nielsen, B. Damman (2012). Numerical Methods for Solution of the Extended Linear Quadratic Control Problem. *Proc. IFAC Conf. Nonlinear Model Predictive Control (NMPC'12)*. Noordwijkerhout, The Netherlands.

[8] G. Frison, J.B. Jørgensen (2013). Efficient Implementation of the Riccati Recursion for Solving Linear-Quadratic Control Problems. *Proc. of IEEE MSC 2013*, 1117-1122.

Chapter 4

Paper [31]

2014 European Control Conference (ECC)
June 24-27, 2014. Strasbourg, France

High-Performance Small-Scale Solvers for Linear Model Predictive Control

Gianluca Frison, Hans Henrik Brandenborg Sørensen, Bernd Dammann, John Bagterp Jørgensen

Abstract— **In Model Predictive Control (MPC), an optimization problem needs to be solved at each sampling time, and this has traditionally limited use of MPC to systems with slow dynamic. In recent years, there has been an increasing interest in the area of fast small-scale solvers for linear MPC, with the two main research areas of explicit MPC and tailored on-line MPC. State-of-the-art solvers in this second class can outperform optimized linear-algebra libraries (BLAS) only for very small problems, and do not explicitly exploit the hardware capabilities, relying on compilers for that. This approach can attain only a small fraction of the peak performance on modern processors. In our paper, we combine high-performance computing techniques with tailored solvers for MPC, and use the specific instruction sets of the target architectures. The resulting software (called HPMPC) can solve linear MPC problems 2 to 8 times faster than the current state-of-the-art solver for this class of problems, and the high-performance is maintained for MPC problems with up to a few hundred states.**

I. Introduction

In recent years, there has been an increasing interest in fast small-scale solvers for linear Model Predictive Control (MPC). This is due to both the need of extend the use of MPC to faster systems (with KHz sampling frequencies), and to the use of decomposition algorithms (where a large number of small problems has to be solved). The two main research areas in fast MPC are explicit MPC [1] and tailored solvers for on-line MPC [10]. In turn, solvers for on-line MPC can be divided into two classes: first order methods (e.g. gradient methods) and second order methods (e.g. interior-point methods). In our paper, we will focus on interior-point methods for on-line MPC, that have the useful property of converging in a number of iterations almost independent of the problems size and conditioning.

Second order methods make use of matrix-matrix linear-algebra operations (level 3 BLAS), that require $\mathcal{O}(n^3)$ floating-point operations (flops) while using $\mathcal{O}(n^2)$ storage space: thus each matrix element is accessed $\mathcal{O}(n)$ times. In modern architectures the cost of a memory operation (memop) is much higher that the cost of a floating-point operation (flop). Furthermore, most instructions are pipelined, and their latency can be effectively hidden if enough independent operations are present in the code. As a consequence, an implementation of a linear-algebra routine only concerned in reducing the number of flops would attain a low performance, since the processor would be idle most of the time, waiting for operands to be fetched from main memory or for dependent instructions to complete. A technique used to mitigate both issues is blocking for registers.

High-performance implementations of level 3 BLAS can attain performances very close to the theoretical peak for large-enough matrices [4]. This performance is obtained by employing different levels of blocking (e.g. for registers, level 2 cache, etc.), copying data in contiguous memory, and using assembly code for the innermost loops and architecture-specific SIMD (Single-Instruction Multiple-Data) instructions (e.g. SSE, AVX in Intel and AMD processors). However, compilers are not very good at producing blocked code, nor at using SIMD, so this is still something that should be done by the programmer.

The drawback of the approach employed in high-performance BLAS is that, for small matrices, the cost of all these memory copies and different levels of blocking would be totally dominant. Thus, in recent years there has been much research about the possibility of improving the speed of small-scale MPC solvers, studying alternatives to BLAS.

CVXGEN [5] is a well-known small-scale solver for convex optimization problems, that can solve many MPC problems. It employs a predictor-corrector Interior-Point (IP) method, and a sparse LDL factorization for the solution of the KKT system at each iteration of the IP method. The approach used to implement the linear algebra is code generation: a tailored solver is generated for the size and the form of each individual problem. The output of the code generation process is a set of C source files, where all the single operations are written down, without loops. In a following step, the compiler has the task to convert this C code into an executable. The main advantages of this approach are that there are no loops nor function calls (and then no associated overhead) and branches (and then no branch misprediction). The main disadvantages are that instruction cache is not exploited (since each instruction is executed only once), and that the code size grows with the cube of the matrices size, becoming quickly intractable.

FORCES [2] is a numerical optimization framework for convex multistage problems, that can solve a wide class of MPC problems. It employs a predictor-corrector IP method, and a tailored solver for the KKT system, based on a block Cholesky factorization of the Schur complement of the KKT matrix. This tailored solver has been previously employed in the Fast-MPC [10] solver. FORCES uses a different approach to code generation: instead of writing down all the single operations, it uses nested triple-loops, where the loops size is tailored for each individual problem and fixed at compile time. This enables the compiler to perform loop unrolling

Authors are with Technical University of Denmark, DTU Compute - Department of Applied Mathematics and Computer Science, DK-2800 Kgs Lyngby, Denmark. Email: giaf at imm.dtu.dk

when it is most profitable, while keeping the size of the executable approximately constant. The main advantage of this approach is that the performance scales much better with the problem size. The main disadvantage is that a triple-loop based approach can attain only a small fraction of the peak performance of the processor.

In this paper, we propose a novel approach to implement solvers for linear MPC problems, combining the implementation techniques of high-performance optimized BLAS libraries with the small-scale speed of code-generation and solvers specially tailored for MPC problems.

The proposed algorithm for the solution of the KKT system of MPC problems is similar to the one presented in [3], with the difference that it moves the integration process one step further: the factorization and the backward recursion of the solution are fused. This allows us to reduce the number of function calls to linear-algebra routines to 3 in the factorization and 3 in the solution, for each iteration of the Riccati-like recursion.

About the implementation, we use an approach somehow similar to the one proposed in the BLIS [9] framework, with the difference that we only block for registers and add code-generation. More specifically, we write the innermost loop as a separate (and optimized) micro-kernel, and block for the size of the registers. Furthermore, we employ a partial code-generation approach: only the two outermost loops around the micro-kernel are totally unrolled. This give a good balance between speed and code size, and in any case a library version of the code is available too. The performance for small-scale problems is up to one order of magnitude higher than the one obtained using optimized BLAS, and the cross-over point is for problems with several hundreds states and controls, large enough for most MPC applications. Furthermore, our tests in section VI show that our solver is from 2 (for the smaller problem) to 8 (for the larger) times faster than the current state-of-the-art solver for linear MPC.

The small number of function calls means that we need to write and optimize only 6 linear algebra routines. Actually, good performance can be obtained using the reference version of the code and optimizing only the matrix-matrix multiplication micro-kernel. This approach is portable, since the only code that needs to be optimized on a new architecture is this micro-kernel. The BLIS framework will make available highly optimized micro-kernels for a number of architectures [8], and at that time it will be possible to combine our solver with these micro-kernels, to obtain high performance on an even wider range of architectures.

We will publish the HPMPC code as open-source, so our optimized solver can be used out of the box on most Intel and AMD machines.

II. Problems

In this paper, we focus our attention on efficient solvers for the Linear-Quadratic (LQ) control problem, that can be considered the core problem in MPC. In fact, it is a rather general formulation that can represent a number of problems in optimal control and estimation, and in particular it arises as sub-problem in Interior-Point (IP) methods for MPC. High-performance of a solver for the LQ control problem immediately translates in high-performance for solvers for a wider class of problems.

A. LQ control problem

The LQ control problem (LQCP) is the equality constrained quadratic program

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

where $n \in \{0, 1, \ldots, N-1\}$ and

$$\begin{aligned} \varphi_n(x_n, u_n) &= \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}' \begin{bmatrix} R_n & S_n & s_n \\ S_n' & Q_n & q_n \\ s_n' & q_n' & \rho_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} = \mathcal{X}_n' \mathcal{Q}_n \mathcal{X}_n \\ \varphi_N(x_N) &= \begin{bmatrix} x_n \\ 1 \end{bmatrix}' \begin{bmatrix} P & p \\ p' & \pi \end{bmatrix} \begin{bmatrix} x_n \\ 1 \end{bmatrix} = \bar{\mathcal{X}}_n' \mathcal{P} \bar{\mathcal{X}}_n \end{aligned} \tag{2}$$

All matrices in this formulation can be dense and time variant. We assume that all matrices $\mathcal{Q}_n$ and $\mathcal{P}$ are symmetric positive definite.

B. Linear MPC problem

The linear MPC problem with linear constraints is the quadratic program

$$\begin{aligned} \min_{u_n, x_{n+1}} \quad & \phi = \sum_{n=0}^{N-1} \varphi_n(x_n, u_n) + \varphi_N(x_N) \\ s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \\ & x_0 = \bar{x}_0 \\ & C_n x_n + D_n u_n \geq d_n \\ & C_N x_N \geq d_N \end{aligned} \tag{3}$$

III. Solver for the LQ control problem

In this section we want to show a solution procedure for the LQ control problem (1) equivalent to the classical Riccati recursion, but with important advantages on the implementation side.

A. Derivation

It is well known from literature that the LQ control problem can be solved by using dynamic programming. Here we do not want to repeat the proof, but only to show a procedure to optimize the stage cost that leads to an efficient implementation in practice.

The optimal stage cost at the generic stage $n+1$ is

$$V_{n+1}^*(x_{n+1}) = \begin{bmatrix} x_{n+1}' & 1 \end{bmatrix} \begin{bmatrix} P_{n+1} & p_{n+1} \\ p_{n+1}' & \pi_{n+1} \end{bmatrix} \begin{bmatrix} x_{n+1} \\ 1 \end{bmatrix}.$$

Inserting the expression

$$x_{n+1} = \begin{bmatrix} B_n & A_n & b_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}$$

the optimal stage cost becomes

$$V^*_{n+1}(x_n,u_n) = \mathcal{X}'_n\mathcal{A}'_n\mathcal{P}_{n+1}\mathcal{A}_n\mathcal{X}_n = \\ = \begin{bmatrix} u_x \\ x_n \\ 1 \end{bmatrix}' \begin{bmatrix} B'_n & 0 \\ A'_n & 0 \\ b_n & 1 \end{bmatrix} \begin{bmatrix} P_{n+1} & p_{n+1} \\ p'_{n+1} & \pi_{n+1} \end{bmatrix} \begin{bmatrix} B_n & A_n & b_n \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}$$

If the matrix $\mathcal{P}_{n+1}$ is positive definite, it can be factorized using Cholesky factorization, as

$$\mathcal{P}_{n+1} = \mathcal{L}_{n+1}\mathcal{L}'_{n+1} = \\ = \begin{bmatrix} L_{n+1,22} & \\ L_{n+1,32} & L_{n+1,33} \end{bmatrix} \begin{bmatrix} L'_{n+1,22} & L'_{n+1,32} \\ & L'_{n+1,33} \end{bmatrix}$$

and then the optimal stage cost becomes

$$V^*_{n+1}(x_n,u_n) = \mathcal{X}'_n\mathcal{A}'_n\mathcal{L}_{n+1}\mathcal{L}'_{n+1}\mathcal{A}_n\mathcal{X}_n = \\ = \mathcal{X}'_n(\mathcal{L}'_{n+1}\mathcal{A}_n)'(\mathcal{L}'_{n+1}\mathcal{A}_n)\mathcal{X}_n$$

that can be build efficiently by exploiting the symmetry and the fact that $\mathcal{L}_{n+1}$ is a lower triangular matrix.

The stage cost at the stage n (dropping the indexes n and $n+1$ in the last equation)

$$V_n(x_n,u_n) = \varphi_n(x_n,u_n) + V^*_{n+1}(x_n,u_n) = \\ = \mathcal{X}'_n(\mathcal{Q}_n + (\mathcal{L}'_{n+1}\mathcal{A}_n)'(\mathcal{L}'_{n+1}\mathcal{A}_n))\mathcal{X}_n = \\ \begin{bmatrix} u \\ x \\ 1 \end{bmatrix}' \begin{bmatrix} R+B'PB & S+B'PA & s+B'(Pb+p) \\ S'+A'PB & Q+A'PA & q+A'(Pb+p) \\ s'+(Pb+p)'B & q'+(Pb+p)'A & \rho+b'Pb+2b'p+\pi \end{bmatrix} \begin{bmatrix} u \\ x \\ 1 \end{bmatrix}$$

is a function of x_n and u_n, and can be easily minimized with respect to u_n in the following way. The matrix is positive definite (since it is the sum of a positive definite matrix and a positive semi-definite matrix), and then the stage cost can be factorized by using the Cholesky factorization of the matrix,

$$\mathcal{M}_n = \mathcal{Q}_n + \mathcal{A}'_n\mathcal{P}_{n+1}\mathcal{A}_n = \\ = \begin{bmatrix} L_{n,11} & & \\ L_{n,21} & L_{n,22} & \\ L_{n,31} & L_{n,32} & L_{n,33} \end{bmatrix} \begin{bmatrix} L'_{n,11} & L'_{n,21} & L'_{n,31} \\ & L'_{n,22} & L'_{n,32} \\ & & L'_{n,33} \end{bmatrix}$$

obtaining the expression for the stage cost $V_n(x_n,u_n)$

$$\begin{bmatrix} L'_{n,11}u_n+L'_{n,21}x_n+L'_{n,31} \\ L'_{n,22}x_n+L'_{n,32} \\ L'_{n,33} \end{bmatrix}' \begin{bmatrix} L'_{n,11}u_n+L'_{n,21}x_n+L'_{n,31} \\ L'_{n,22}x_n+L'_{n,32} \\ L'_{n,33} \end{bmatrix} \\ = (L'_{n,11}u_n+L'_{n,21}x_n+L'_{n,31})'(L'_{n,11}u_n+L'_{n,21}x_n+L'_{n,31}) \\ + (L'_{n,22}x_n+L'_{n,32})'(L'_{n,22}x_n+L'_{n,32}) + L_{n,33}L'_{n,33}$$

Notice that u_n is present only in the first term of the sum: this term is a square, and then its minimum is zero, attained for the value of u_n

$$u_n = -(L'_{n,11})^{-1}(L'_{n,21}x_n + L'_{n,31}). \tag{4}$$

The corresponding optimal value $V^*_n(x_n)$ of the stage cost is given by the remaining two terms of the sum:

$$V^*_n(x_n) = (L'_{n,22}x_n+L'_{n,32})'(L'_{n,22}x_n+L'_{n,32}) + L_{n,33}L'_{n,33} = \\ = \begin{bmatrix} x'_n & 1 \end{bmatrix} \begin{bmatrix} P_n & p_n \\ p_n & \pi_n \end{bmatrix} \begin{bmatrix} x_n \\ 1 \end{bmatrix}$$

as in the classical formulation of the dynamic programming for the LQ control problem. Notice that the procedure gives a factorization of the matrix $\mathcal{P}_n$ that can be used at the following stage to efficiently compute $\mathcal{A}'_{n-1}\mathcal{P}_n\mathcal{A}_{n-1}$.

The value of u_n in (4) can be rewritten as

$$u_n = -(R_n+B'_nP_{n+1}B_n)^{-1}\big((S_n+B'_nP_{n+1}A_n)x_n + \\ + s_n + B'_n(P_{n+1}b_n+p_{n+1})\big) = K_nx_n + k_n$$

that is the usual expression of u_n as time varying affine state feedback given by the Riccati recursion. However, the procedure to compute u_n as in (4) is more efficient from a computational point of view. Also notice that the recursion matrix P_n of the Riccati recursion is never computed explicitly in the above solution procedure.

B. Algorithm

The Riccati-like procedure presented in the previous section leads to an efficient algorithm in practice, summarized in Algorithm 1.

The classical Riccati recursion for the solution of the LQ control problem consist of a backward recursion for the KKT matrix factorization (with cubic complexity in the matrices size) and backward and forward substitution for the KKT system solution (with quadratic complexity in the matrices size). For small systems, the cost for the factorization and the cost for the solution have the same order of magnitude.

In the proposed algorithm the factorization and the backward substitution are fused in a single backward loop. The number of function calls to BLAS per backward iteration is only 3, thanks to the packing of matrices. This reduces the function calls overhead and the data movement.

The proposed algorithm does not contain calls to level 2 BLAS functions in the backward loop, that have been replaced by packing the vectors s_n and q_n with the matrices R_n, S_n and Q_n, and the vector b_n with the matrices A_n and B_n to perform calls to level 3 BLAS on a single larger matrix. As a result, the calls to level 2 BLAS in the backward substitution of the the classical Riccati recursion come almost for free, since the matrix operands are already loaded in the registers.

Algorithm 1 Solution procedure for the LQ control problem

1: $\begin{bmatrix} L_{N+1,22} & \\ L_{N+1,32} & L_{N+1,33} \end{bmatrix} \leftarrow \mathcal{P}^{1/2}$ ▷ dpotrf
2: **for** $n \leftarrow N \rightarrow 0$ **do**
3: $\quad \mathcal{L}'_{n+1}\mathcal{A}_n \leftarrow L'_{n+1,22}\begin{bmatrix} B_n & A_n & b_n \end{bmatrix} + \begin{bmatrix} 0 & 0 & L'_{n+1,32} \end{bmatrix}$ ▷ dtrmm
4: $\quad \mathcal{M}_n \leftarrow \mathcal{Q}_n + (\mathcal{L}'_{n+1}\mathcal{A}_n)'(\mathcal{L}'_{n+1}\mathcal{A}_n)$ ▷ dsyrk
5: $\quad \begin{bmatrix} L_{n,11} & & \\ L_{n,21} & L_{n,22} & \\ L_{n,31} & L_{n,32} & L_{n,33} \end{bmatrix} \leftarrow \mathcal{M}_n^{1/2}$ ▷ dpotrf
6: **end for**
7: **for** $n \leftarrow 0 \rightarrow N$ **do**
8: $\quad u_n \leftarrow -(L'_{n,11})^{-1}(L'_{n,21}x_n + L'_{n,31})$ ▷ dgemv & dtrsv
9: $\quad x_{n+1} \leftarrow \begin{bmatrix} B_n & A_n \end{bmatrix}\begin{bmatrix} u'_n & x'_n \end{bmatrix}' + b_n$ ▷ dgemv
10: **end for**

The cost of the algorithm in flops is:

$$N\big((\tfrac{7}{3}n_x^3+4n_x^2n_u+2n_xn_u^2+\tfrac{1}{3}n_u^3)+(\tfrac{13}{2}n_x^2+9n_xn_u+\tfrac{5}{2}n_u^2)\big)$$

IV. Implementation details

In this section we present the techniques used in the implementation of our software.

A. Blocking for registers

The most important technique is certainly blocking for registers: this reduces the number of memops, and helps hiding the latency of operations. We will explain the idea with an example. Suppose that we want to compute the product of two squared matrices A and B of size n, and use the result to update the square matrix C of size n:

$$C = C + A \cdot B$$

If we use the definition of matrix-matrix product, we can compute each element c_{ij} of C as

$$c_{ij} = c_{ij} + \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}. \tag{5}$$

If we store the element c_{ij} in a register, the computation of one element of C requires $2n$ flops (n multiplications and n sums) and $2n + 2$ memops (1 load and 1 store of c_{ij}, n loads of both a_{ik} and b_{kj}). In total, the matrix-matrix product would require approximately $2n^3$ flops and $2n^3$ memops, with a ratio flops/memops of 1.

If we use more registers to store elements of C, we can improve this ratio. If for example we store a 2×2 sub-matrix of C, then, for each k, we can load 2 element of A and 2 elements of B, to update 4 elements of C, as

	b_0	b_1
a_0	$c_{00} + a_0 \cdot b_0$	$c_{01} + a_0 \cdot b_1$
a_1	$c_{10} + a_1 \cdot b_0$	$c_{11} + a_1 \cdot b_1$

(6)

Once loaded in the registers, each element of A and B is used twice: the ratio flops/memops is then about 2.

In general, if we can store a sub-matrix of C of size n_r, the ratio flops/memops is about n_r. In practice, the number of available registers is limited, and the size of the sub-matrix of C stored in the registers has to be chosen accordingly.

The same idea can be applied to other memory levels, for example blocking for level 2 or 3 cache. However, since our target are small-scale problems that can already fit in cache, we did not implement blocking for cache, but we store the elements of the matrices in the same order as they are accessed by the matrix-matrix multiplication micro-kernel.

Furthermore, notice that the 4 multiply-accumulate in (6) are totally independent, and could be performed in parallel, while this is not the case unrolling the loop in (5). Thus blocking for registers can be used to get enough independent operations to keep the execution units busy, since most floating-point instructions are pipelined, and their throughput is lower than the latency.

B. SIMD instructions

SIMD (Single-Instruction Multiple-Data) are instruction that perform the same operation in parallel on all elements of small vectors of data. In theory, an operation on a vector of size n_v can improve the performance up to n_v times.

In our implementation, we make use of SSE-SSE2-SSE3 instructions (that operates on 128-bit-wide vectors, storing 2 doubles) and AVX instructions (that operates on 256-bit-wide vectors, storing 4 doubles). We mainly use the intrinsics version of the instruction: this makes the programming much easier, since the compiler takes care of registers allocation and instruction scheduling.

If we want to implement (6) using SSE3 instructions, it is

	b_0	b_1
a_0	$\begin{bmatrix} c_{00} \\ c_{10} \end{bmatrix} + \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_0 \end{bmatrix}$	$\begin{bmatrix} c_{01} \\ c_{11} \end{bmatrix} + \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_1 \end{bmatrix}$
a_1		

where the squred brackets indicates the small vectors. As a result, the number of operations is halved.

SIMD instructions often have alignment requirements to obtain high performance: for example, SSE instructions can efficiently load and store data that is 128 bits (or 16 bytes) aligned, while for the AVX instructions the alignment requirement is 256 bits (or 32 bytes). In our LQ control problem solver, we require the data to be already aligned, and we deal with this in the IP method.

C. Customized BLAS

In order to obtain the highest performance for small problems, we implemented the few BLAS routines needed by our solver using the techniques presented above.

More in detail, we implemented a simplified version of the needed BLAS routines, with one only option per routine (e.g. we only work with lower triangular matrices). The innermost loop of each BLAS routine is implemented as a separate micro-kernel, coded using blocking for registers and SIMD.

We employ a partial code-generation approach, where the innermost loop is coded as a function (kernel), and the two outermost loops are totally unrolled. This gives a good trade-off between performance (fewer branches and indexes computation) and code size. A library version of the code is also available, and usually it is faster for system with more than about 30 states.

V. IP method for the linear MPC problem

The linear MPC problem in (3) can be solved using an IP method. In this paper, we employ a primal-dual IP method [6]. Let us consider the general quadratic program

$$\begin{aligned} \min_{y} \quad & \tfrac{1}{2} y' H y + g' y \\ s.t. \quad & Ay = b \\ & Cy \geq d \end{aligned}$$

then at each iteration k of the IP method it has to be solved a linear system of equations of the form

$$\begin{bmatrix} H + C'(T_k^{-1}\Lambda_k)C & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} y_k \\ \pi_k \end{bmatrix} = \\ = - \begin{bmatrix} g - C'(\Lambda_k e + T_k^{-1}\Lambda_k d + T_k^{-1}\sigma\mu_k e) \\ b \end{bmatrix} \tag{7}$$

where t_k are the slack variables, π_k and λ_k are the Lagrangian multipliers associated with the equality and inequality constraints, μ_k is the duality measure, σ is a centering parameter and e is a vector of ones. In the case of the linear MPC problem, it can be shown [7] that (7) is the KKT system of an instance of the LQ control problem (1). This means that at each iteration of the IP method we can use our solver for the LQ control problem to solve the linear system of equations (7), that is the main computational effort at each iteration.

VI. Results

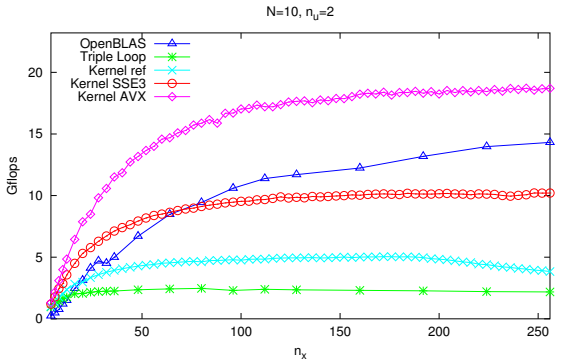
In this section we present the results of two series of test: in the one we compared the relative performance of different implementations of our solver for the LQ control problem; in the second one we compared our IP method for linear MPC with the current state-of-the-art solver for linear MPC. In case of multi-core machines, only one core is used.

A. LQ control problem

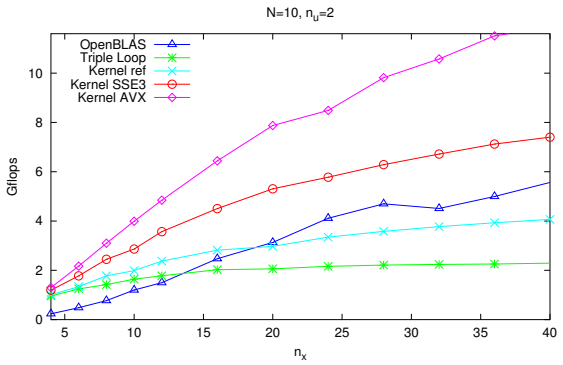
To assess the performance of the different implementations of our solver for the LQ control problem, we tested a version using for the linear-algebra BLAS and LAPACK provided by OpenBLAS 0.2.6 [11]; a version using tailored triple-loop based linear algebra and code generation; and three versions implemented using the techniques presented in this paper, and coded respectively in C code, SSE3 instructions and AVX instructions. All tests have been performed on a Laptop equipped with an Intel i5 2410M processor (2.3 GHz, up to 2.9 GHz in turbo mode), running Xubuntu 13.04; the compiler is gcc 4.7.3. In figure 1 we plot the performance in Gflops obtained using the different approaches, and compared with the theoretical peak performance of the processor (that has been computed assuming that it operates at the maximum turbo frequency, as 2.9 GHz * 2 floating-point instructions per clock (one add and one mult) * 4 flops per floating-point instruction (AVX) = 23.2 Gflops).

When our solver is linked to OpenBLAS, the performance is good for large problems (close to the theoretical peak), but it is poor for small problems. The approach making use of triple-loop and code generation is faster for small systems, but can only attain a small fraction of the theoretical peak: as a consequence, it can outperform OpenBLAS only for very small problems. The version written in C code and employing blocking for registers doubles the performance with respect to the triple-loop one.

The version using micro-kernels coded with SSE3 instructions and blocking for registers doubles again the performance. For very small problems, the performance is almost 10 times the one obtained using OpenBLAS. The version using micro-kernels coded with AVX instructions and blocking for registers almost doubles the performance again. In this test, the performance keeps increasing with n_x, and the maximum performance is 80.6% (18.7 Gflops) of the theoretical peak performance at turbo frequency. On the test machine, the `dgemm` micro-kernel has a steady performance above 90% of the peak for matrices of size up to about 340:



(a) Performance in Gflops.



(b) Performance in Gflops, small systems.

Fig. 1: Performance test of different implementations our LQCP solver. The test machine is the Intel Core i5 2410M. The x-axis is the number of states; $n_u = 2$ and $N = 10$. Figure (a) is scaled on the y-axis such that the top to the figure represents the turbo peak performance (23.2 Gflops).

for larger matrices, the memory footprint exceeds L3 cache, and the performance decreases. However, this matrix size is large enough for most MPC problems.

B. Linear MPC problem

In this section we compare the IP solver part of our HPMPC toolbox with FORCES (to our knowledge the state-of-the-art solver for linear MPC), running the mass-spring test in table VI in the paper [2]. The tests are performed on several x86 and x86_64 machines, all running different flavors of Linux (mainly Ubuntu-based) and using gcc as C compiler: results are in table I.

The versions using SSE3 and AVX could be compared each other on two laptops, one equipped with the CPU Intel Core i7 3520M (Ivy Bridge) at 2.9 GHz (3.6 GHz in turbo mode), the other with the CPU Intel Core i5 2410M (Sandy Bridge) at 2.3 GHz (2.9 GHz in turbo mode). On both machines, our IP solver is from about 2 (for the smaller tests) to 8 (for the larger test) times faster than FORCES. The better performance is obtained exploiting the wider AVX instruction set. The version using OpenBLAS is faster than FORCES starting from the third problem.

We tested our code also on a laptop equipped with the older Intel Core 2 Duo P8600 at 2.4 GHz. This architecture (named Penryn) features the SSE4.2 instruction set, but not AVX. This time the maximum speed-up with respect to FORCES is about 5, lower than using AVX.

Instead the embedded system equipped with of Intel Atom Z530 in [2], we used a netbook equipped with the equivalent Intel Atom N270 processor (1.6 GHz). This architecture is different compared to all the others considered in our tests. In fact, the processor is 32-bit (and then there are only 8 SSE registers, instead of 16), the cache is smaller (24 KB L1 data cache, 512 KB L2 cache, no L3 cache), and the processor performs in-order-execution (and then the order of the instructions matters): the resulting performance is thus quite low, and it is much more difficult to write fast code. The best code was obtained using scalar SSE2 instructions (no SIMD) in inline assembly. Also in this case the maximum speed-up with respect to FORCES is about 5.

We also tested our code on a machine equipped with the AMD processors Opteron 6168 (1.9 GHz, K10 architecture, SSE3 instruction set): again, the results are similar, with a speed-up of about 6 times with respect to FORCES.

The tests show as our code implementation combines and improves two approaches: the small-scale speed of tailored solvers and code-generation, with the large-scale high-performance of optimized BLAS libraries. Furthermore, our code is better than architecture-agnostic solvers as FORCES in exploiting the advanced features of recent hardware (e.g. AVX instructions).

M	n_x	n_u	N	OpenBLAS	HPMPC SSE	HPMPC AVX	FORCES
Intel Core i7 3520M (Ivy Bridge)							
2	4	1	10	0.25	0.04	0.03	0.08
4	8	3	10	0.44	0.10	0.09	0.29
6	12	5	30	1.59	0.71	0.53	1.67
11	22	10	10	1.63	0.88	0.61	2.90
15	30	14	10	2.63	1.82	1.18	6.70
30	60	29	30	28.02	31.20	19.01	153.02
Intel Core i5 2410M (Sandy Bridge)							
2	4	1	10	0.23	0.05	0.05	0.14
4	8	3	10	0.36	0.13	0.12	0.43
6	12	5	30	1.62	0.89	0.70	2.91
11	22	10	10	1.74	1.06	0.72	4.88
15	30	14	10	3.06	2.20	1.40	10.86
30	60	29	30	36.40	40.57	24.25	205.00
Intel Core 2 Duo P8600 (Penryn)							
2	4	1	10	0.27	0.07	*	0.21
4	8	3	10	0.51	0.20	*	0.70
6	12	5	30	2.58	1.27	*	4.17
11	22	10	10	2.61	1.42	*	6.35
15	30	14	10	4.96	2.85	*	13.88
30	60	29	30	72.02	49.12	*	264.12
Intel Atom N270 (Bonnell)							
2	4	1	10	1.14	0.48	*	1.21
4	8	3	10	2.61	1.30	*	3.94
6	12	5	30	14.94	7.58	*	26.32
11	22	10	10	15.61	9.18	*	36.79
15	30	14	10	30.04	18.30	*	75.46
30	60	29	30	498.53	332.72	*	1717.60
AMD Opteron 6168 (K10)							
2	4	1	10	0.30	0.10	*	0.26
4	8	3	10	0.58	0.26	*	0.98
6	12	5	30	2.92	1.70	*	6.19
11	22	10	10	3.07	1.91	*	12.52
15	30	14	10	5.49	3.90	*	26.81
30	60	29	30	80.37	70.40	*	470.03

TABLE I: Run times [in ms] for 10 interior point iterations, averaged over 100 random initial states. The tests are the same as in TABLE VI in [2]. * AVX instruct. not supported.

VII. Conclusion

In this paper, we reviewed current state-of-the-art solvers for linear MPC, and we proposed a novel approach for this class of problems. We presented a solver for the LQ control problem with good asymptotic complexity, implemented using a very small number of function calls to linear-algebra routines. This allows us to write and optimize only a small subset of BLAS, combining code-generation with high-performance techniques and exploiting the hardware specific instructions: the innermost loop of each linear-algebra routine is implemented as a separate micro-kernel, and coded using the available SIMDs. As a result, our solver outperforms both state-of-the-art linear MPC solvers and optimized BLAS libraries, attaining a performance close to the theoretical peak for a wide range of problem sizes.

Acknowledgement

We would like to thank Gabriele Frison for the help in testing the solvers on the Atom architecture. We would also like to thank the reviewers for the useful feedback.

References

[1] A. Bemporad, M. Morari, V. Dua, and E.N. Pistikopoulos. The Explicit Linear Quadratic Regulator for Constrained Systems. *Automatica*, 38(1):3–20, January 2002.

[2] A. Domahidi, A. Zgraggen, M.N. Zeilinger, M. Morari, and C.N. Jones. Efficient interior point methods for multistage problems arising in receding horizon control. In *IEEE Conference on Decision and Control (CDC)*, pages 668 – 674, Maui, HI, USA, December 2012.

[3] Gianluca Frison and John Bagterp Jørgensen. Efficient implementation of the riccati recursion for solving linear-quadratic control problems. In *2013 IEEE Multi-conference on Systems and Control*, pages 1117–1122. IEEE, 2013.

[4] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), 2008.

[5] Jacob Mattingley and Stephen Boyd. CVXGEN: a code generator for embedded convex optimization. *Optimization and Engineering*, 13(1):1–27, March 2012.

[6] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.

[7] Christopher V. Rao, Stephen J. Wright, and James B. Rawlings. Application of interior-point methods to model predictive control. *Journal of optimization theory and applications*, 99:723–757, 1998.

[8] Field G. Van Zee, Tyler Smith, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John Gunnels, Tze Meng Low, Bryan Marker, Lee Killough, and Robert A. van de Geijn. Implementing level-3 BLAS with BLIS: Early experience. FLAME Working Note #69. Technical Report TR-13-03, The University of Texas at Austin, Department of Computer Science, Apr. 2013. submitted to SC13.

[9] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for generating BLAS-like libraries. FLAME Working Note #66. Technical Report TR-12-30, The University of Texas at Austin, Department of Computer Science, Nov. 2012. submitted to ACM Transactions on Mathematical Software.

[10] Yang Wang and Stephen Boyd. Fast model predictive control using online optimization. In *IFAC World Congress*, pages 6974 – 6997, Seoul, July 2008.

[11] Xianyi Zhang. Openblas. http://www.openblas.net/.

Chapter 5

Paper [38]

Preprints of the 19th World Congress
The International Federation of Automatic Control
Cape Town, South Africa. August 24-29, 2014

A Family of High-Performance Solvers for Linear Model Predictive Control

Gianluca Frison * **Leo Emil Sokoler** *
John Bagterp Jørgensen *

* *Technical University of Denmark, DTU Compute - Department of Applied Mathematics and Computer Science, DK-2800 Kgs Lyngby, Denmark. (corresponding author: giaf at imm.dtu.dk).*

Abstract: In Model Predictive Control (MPC), an optimization problem has to be solved at each sampling time, and this has traditionally limited the use of MPC to systems with slow dynamic. In this paper, we propose an efficient solution strategy for the unconstrained sub-problems that give the search-direction in Interior-Point (IP) methods for MPC, and that usually are the computational bottle-neck. This strategy combines a Riccati-like solver with the use of high-performance computing techniques: in particular, in this paper we explore the performance boost given by the use of single precision computation, and techniques such as inexact search direction and mixed precision computation. Finally, we test our HPMPC toolbox, a family of high-performance solvers tailored for MPC and implemented using these techniques, that is shown to be several times faster than current state-of-the-art solvers for linear MPC.

1. INTRODUCTION

Model Predictive Control (MPC) has bee traditionally limited to systems with slow dynamic, with sampling times of seconds or minutes. This is due to the fact that an optimization problem needs to be solved at each sampling time. Nowadays, thanks to algorithmic as well as hardware improvements, this is no more the case, and recent works show that, in case of small systems, even control frequency of milliseconds are possible. The two main approaches for fast MPC are explicit (see Bemporad et al. [2002]) and structure-exploiting on-line MPC (see e.g. Rao et al. [1998],Wang et al. [2010]).

In recent years, several approaches have been proposed to the fast on-line solution of small-scale linear MPC problems, as flat code generation (CVXGEN, Mattingley et al. [2012]) and customized triple-loop based BLAS (FORCES, Domahidi et al. [2012]). However, these solvers do not fully exploit the hardware capabilities of modern architectures, and rely on compilers for the code optimization. As a result, typically they can attain only a small fraction of processor peak performance.

In this paper, we propose an efficient solver for the Linear-Quadratic Control Problem (LQCP), that is a common sub-problem in optimal control and estimation, and in particular it gives the search direction in Interior-Point (IP) methods for linear MPC. Our solver for LQCP only requires 3 calls to linear-algebra routines for the factorization of the KKT system: this decreases the data movement, and allows us to hand optimize these few routines. In particular, we make use of high-performance techniques such as blocking for registers, SIMD instructions, customized BLAS and mixed precision computation. The latter exploits the fact that on the target architecture (in this paper, an Intel's processor) the peak performance of single precision computation is twice as much as in double precision. The resulting solver for LQCP is shown to attain a large fraction of the peak performance for a wide range of problem sizes.

This high-performance solver is used as a routine in primal-dual and Mehrotra's predictor-corrector IP methods for linear MPC. Furthermore, we propose the use of inexact IP methods, where the search directions are found by solving the LQCP sub-problems in single precision in early iterations. These IP methods can produce a solution in double precision in a time that is only slightly larger than in single precision. The resulting solver is several times faster than state-of-the-art solvers for linear MPC (see Domahidi et al. [2012] as a reference), and the high-performance is attained for a wider range of problem sizes.

The paper is organized as follows: section 2 describes the LQCP and linear MPC problems. Section 3 presents high-performance solvers for the LQCP, in single, double and mixed precision. Section 4 briefly introduces primal-dual and Mehrotra's predictor-corrector IP methods, and proposes the inexact IP. Section 5 presents the results of some numerical test, and Section 6 contains the conclusion.

2. PROBLEMS DESCRIPTION

In this paper, we focus our attention on efficient solvers for the LQCP. This is a rather general formulation that can represent several problems in optimal control and estimation, and in particular it gives the search direction in Interior-Point (IP) methods for MPC. Thus an high-performance implementation of a solver for the LQCP can boost the performance of solvers for a wide class of problems.

2.1 Linear-quadratic control problem

The LQCP is the equality constrained quadratic program

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

where

$$\begin{aligned} \varphi_n(x_n, u_n) &= \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}' \begin{bmatrix} R_n & S_n & s_n \\ S_n' & Q_n & q_n \\ s_n' & q_n' & \rho_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} = \mathcal{X}_n' \mathcal{Q}_n \mathcal{X}_n \\ \varphi_N(x_N) &= \begin{bmatrix} u_N \\ x_N \\ 1 \end{bmatrix}' \begin{bmatrix} 0 & 0 & 0 \\ 0 & P & p \\ 0 & p' & \pi \end{bmatrix} \begin{bmatrix} u_N \\ x_N \\ 1 \end{bmatrix} = \mathcal{X}_N' \mathcal{P} \mathcal{X}_N \end{aligned} \tag{2}$$

All matrices can in general be dense and time variant. In this paper, we assume that the matrices $\mathcal{Q}_n$ are symmetric positive definite.

2.2 Linear MPC problem

The linear MPC problem with linear constraints is the quadratic program

$$\begin{aligned} \min_{u_n, x_{n+1}} \quad & \phi = \sum_{n=0}^{N-1} \varphi_n(x_n, u_n) + \varphi_N(x_N) \\ s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \\ & C_n x_n + D_n u_n \geq d_n \\ & C_N x_N \geq d_N \end{aligned} \tag{3}$$

where $\varphi_N(x_N)$ are defined as in (2). Again, all matrices can in general be dense and time variant.

3. SOLVERS FOR THE LQ CONTROL PROBLEM

In this section we present algorithms (and relative implementation) to efficiently solve LQCP. These algorithms can be used as building blocks in different IP methods.

3.1 Solution strategies

There exists several solution strategies for the LQCP (1). In the following of the paper we will consider two of them.

The first solution strategy is based on the fact that the LQCP (1) is an instance of the equality constrained quadratic program

$$\begin{aligned} \min_{x} \quad & \phi = \tfrac{1}{2} z' H z + g' z \\ s.t. \quad & Az = b \end{aligned} \tag{4}$$

The (in general only necessary) optimality conditions for (4) are the well-known KKT conditions, that can be written in matrix notation as

$$\begin{bmatrix} H & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} z^* \\ \pi^* \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix} \tag{5}$$

that is called the KKT system associated with (4). In the case of the LQCP, it can be proved that, if the matrices $\begin{bmatrix} R_n & S_n \\ S_n' & Q_n \end{bmatrix}$ and P are positive definite, then the KKT conditions are also sufficient and (5) has an unique solution. The KKT system of the LQCP is large and sparse, and has a special structure that can be exploited to obtain efficient solvers, see Rao et al. [1998]. The fact that the solution of the LQCP can be obtained by means of the solution of a system of linear equations (i.e. through factorization of the matrix and backward and forward substitutions) implies that we can use mixed precision to perform the computations, as shown later.

Another solution strategy is based on dynamic programming. We do not want to present the theory again (that can be found for example in Jørgensen [2005]), but only show that this leads to an efficient solver in practice, where the factorization and backward substitution in the solution of (5) are fused, as shown in section 3.2.

To implement the IP methods, we need routines to factorize and solve the KKT system, to solve an already factorized KKT system, and to compute the residuals. These routines can be seen as building blocks to implement a number of different IP methods.

3.2 Factorization and solution of the KKT system

The dynamic programming approach can be used to derive an efficient solver, analogue to the classical Riccati recursion but more efficient in practice, where the factorization and the backward substitution are fused together: see Frison et al. [2014] for the details of the derivation. The algorithm (together with the calls to BLAS functions) is presented in Algorithm 1 (where $\mathcal{M}_n^{1/2}$ is the lower triangular Cholesky factor of matrix $\mathcal{M}_n$, partitioned as $\mathcal{Q}_n$ in (2)), and only requires 3 function calls per backward iteration and 3 per forward iteration: this reduces the overhead associated with the function calls, as well as the data movement. The cost of the algorithm is $N\big((\frac{7}{3}n_x^3 + 4n_x^2 n_u + 2n_x n_u^2 + \frac{1}{3}n_u^3) + (\frac{13}{2}n_x^2 + 9n_x n_u + \frac{5}{2}n_u^2)\big)$ flops, plus eventually $N(2n_x^2)$ if π is needed (as e.g. in mixed precision).

Algorithm 1 Factorization and solution of LQCP

$\begin{bmatrix} L_{N+1,22} & \\ L_{N+1,32} & L_{N+1,33} \end{bmatrix} \leftarrow \mathcal{P}^{1/2}$ ▷ potrf
for $n \leftarrow N \rightarrow 0$ **do**
  $\mathcal{L}_{n+1}' \mathcal{A}_n \leftarrow L_{n+1,22}' \begin{bmatrix} B_n & A_n & b_n \end{bmatrix} + \begin{bmatrix} 0 & 0 & L_{n+1,32}' \end{bmatrix}$ ▷ trmm
  $\mathcal{M}_n \leftarrow \mathcal{Q}_n + (\mathcal{L}_{n+1}' \mathcal{A}_n)'(\mathcal{L}_{n+1}' \mathcal{A}_n)$ ▷ syrk
  $\begin{bmatrix} L_{n,11} & \\ L_{n,21} & L_{n,22} \\ L_{n,31} & L_{n,32} & L_{n,33} \end{bmatrix} \leftarrow \mathcal{M}_n^{1/2}$ ▷ potrf
end for
if $PI = 1$ **then**
  $\pi_0 \leftarrow L_{0,22}(L_{0,22}' x_0 + L_{0,32}')$ ▷ trmv
end if
for $n \leftarrow 0 \rightarrow N$ **do**
  $u_n \leftarrow -(L_{n,11}')^{-1}(L_{n,21}' x_n + L_{n,31}')$ ▷ gemv & trsv
  $x_{n+1} \leftarrow A_n x_n + B_n u_n + b_n$ ▷ gemv
  if $PI = 1$ **then**
    $\pi_{n+1} \leftarrow L_{n+1,22}(L_{n+1,22}' x_0 + L_{n+1,32}')$ ▷ trmv
  end if
end for

3.3 Solution of the (factorized) KKT system

In a predictor-corrector IP, the corrector step is computed by solving a system of linear equations (in the form (5)) that has the same left hand side as the system giving the predictor step, but a different right hand side. And similarly, in mixed precision we need to solve multiple systems with the same left hand side.

An efficient algorithm to solve (5) for the LQCP exploiting the already factorized l.h.s. matrix can be obtained by exploiting the analogy between Algorithm 1 and the classical Riccati recursion, i.e. that $L_{n,22}$ is the lower triangular Cholesky factor of the Riccati recursion matrix P_n. The algorithm is presented in Algorithm 2. The cost of the algorithm is $N(8n_x^2 + 8n_xn_u + 2n_u^2)$ flops, plus eventually $N(2n_x^2)$ if π is needed.

Algorithm 2 Solution of (factorized) LQCP

$p_N \leftarrow p$
for $n \leftarrow N \rightarrow 0$ **do**
  $P_{n+1}b_n \leftarrow L_{n+1,22}L'_{n+1,22}b_n + p_{n+1}$ ▷ `trmv`
  $\begin{bmatrix} l'_n & p'_n \end{bmatrix}' \leftarrow \begin{bmatrix} s'_n & q'_n \end{bmatrix}' + \begin{bmatrix} B'_n & A'_n \end{bmatrix} \cdot (P_{n+1}b_n)$ ▷ `gemv`
  $l_n \leftarrow L_{n,11}^{-1} l_n$ ▷ `trsv`
  $p_n \leftarrow p_n - L_{n,21}l_n$ ▷ `gemv`
end for
if $PI = 1$ **then**
  $\pi_0 \leftarrow L_{0,22}L'_{0,22}x_0 + p_0$ ▷ `trmv`
end if
for $n \leftarrow 0 \rightarrow N$ **do**
  $u_n \leftarrow -(L'_{n,11})^{-1}(L'_{n,21}x_n + l_n)$ ▷ `gemv & trsv`
  $x_{n+1} \leftarrow A_nx_n + B_nu_n + b_n$ ▷ `gemv`
  if $PI = 1$ **then**
    $\pi_{n+1} \leftarrow L_{n+1,22}L'_{n+1,22}x_0 + p_{n+1}$ ▷ `trmv`
  end if
end for

3.4 Residuals computation

To solve a system of linear equations using mixed precision, we need a routine for the computation of the residuals, that in the solution of (5) are defined as

$$\begin{bmatrix} r_g \\ r_b \end{bmatrix} = -\begin{bmatrix} H & -A' \\ -A & 0 \end{bmatrix}\begin{bmatrix} z^* \\ \pi^* \end{bmatrix} - \begin{bmatrix} g \\ b \end{bmatrix}.$$

If system (5) was solved exactly, the residuals would be zero. However, because of the finite precision of the computations, in practice residuals are generally not zero. An algorithm for the computation of the residuals for LQCP is presented in Algorithm 3. The cost of the algorithm is $N(6n_x^2 + 8n_xn_u + 2n_u^2)$ flops.

Algorithm 3 Residuals of LQCP

$r_{s,0} \leftarrow -(S_0x_0 + R_0u_0 + B'_0\pi_1 + s_0)$ ▷ `gemv & symv & gemv`
$r_{b,0} \leftarrow x_1 - \left(\begin{bmatrix} B_0 & A_0 \end{bmatrix}\begin{bmatrix} u_0 \\ x_0 \end{bmatrix} + b_0\right)$ ▷ `gemv`
for $n \leftarrow 1 \rightarrow N-1$ **do**
  $\begin{bmatrix} r_{s,n} \\ r_{q,n} \end{bmatrix} \leftarrow \begin{bmatrix} \pi_n \\ 0 \end{bmatrix} - \left(\begin{bmatrix} R_n & S_n \\ S'_n & Q_n \end{bmatrix}\begin{bmatrix} u_n \\ x_n \end{bmatrix} + \begin{bmatrix} B'_n \\ A'_n \end{bmatrix}\pi_{n+1} + \begin{bmatrix} s_n \\ q_n \end{bmatrix}\right)$ ▷ `symv & gemv`
  $r_{b,n} \leftarrow x_{n+1} - \left(\begin{bmatrix} B_n & A_n \end{bmatrix}\begin{bmatrix} u_n \\ x_n \end{bmatrix} + b_n\right)$ ▷ `gemv`
end for
$r_{q,N} \leftarrow \pi_N - (Px_N + p)$ ▷ `symv`

3.5 Implementation details

For each algorithm, we implemented two versions: one calling BLAS, and the other calling tailored linear algebra routines written in C using the following HPC techniques: see Frison et al. [2014] for more details.

Blocking for registers. This is the single most important technique, and can be used on all machines. It has a double aim: reduce the number of memory operations, and hide latency of floating-point operations. On modern architectures, the CPU is much faster than the main memory: as a consequence the cost of a memop is much higher that the cost of a flop. A hierarchy of smaller and faster memories (registers, caches) is used to mitigate this difference in speed, and the programmer should reuse data already present in faster memories. As an idea, blocking is a technique that consist of loading a sub-matrix in a certain memory level ($\mathcal{O}(n^2)$ memops), to perform the required operation on that sub-matrix ($\mathcal{O}(n^3)$ flops for level-3 BLAS). In this way, the ratio flops/memops is increased. In our implementation we only block for registers, since for matrices too large to fit in cache BLAS is high-performing, and thus we can switch to the version calling BLAS. Blocking for registers is also used to hide the latency of operations: for example, on most Intel machines floating-point add and mul are pipelined and can be issued every clock cycle, but their result is available after respectively 3 and 5 clock cycles.

SIMD instructions. SIMD (Single-Instruction Multiple-Data) are instructions that perform the same operation in parallel on all elements of small vectors of data: this reduces the number of operations, and can improve performance up to n_v times for small vectors of size n_v. Nowadays many architectures implement SIMD instructions: as an example, Intel and AMD have the SSE instructions (that operates on 2 doubles or 4 floats at a time) and AVX instructions (that operates on 4 doubles or 8 floats at a time). The size of the small vectors suggests that, using SIMD instructions, the theoretical performance in single precision is twice as much as the theoretical performance in double precision. The drawback is that usually SIMD are more difficult to program, and they have alignment requirements: SSE instructions can efficiently load and store data that is 16 bytes aligned, while for AVX instructions the alignment requirement is 32 bytes. The alignment requirements limit the possibility for a compiler to use SIMD. We explicitly deal with alignment requirements in the IP methods, such that the data passed to LQCP solvers is already aligned.

Customized BLAS. In our LQCP solvers, we need only a small subset of BLAS, and then there is no need to implement it all. The innermost loop of each linear-algebra routine is implemented as a separate micro-kernel, hand optimized using block for registers and SIMD intrinsics. Furthermore, in the code used for this paper the size of all matrices is fixed at compile time: this reduces the number of branches, and allows the compiler to further optimize.

Single/double/mixed precision. On the target architecture one SIMD instruction can operate on twice as many floats as doubles. This, together with the fact that floats occupy half the space in memory (including registers and caches) and use half the memory bandwidth, gives that the performance in single precision is about twice the performance in double precision. Hence the reason for using single precision whenever possible. Mixed precision iterative refinement is a technique that allows to solve a system of linear equations exploiting the higher performance of single precision in the most expensive parts while maintaining the double precision of the final result, see Buttari et al. [2007]. A mixed precision algorithm for the solution of LQCP is

presented in Algorithm 4. The algorithm can be seen as an iterative algorithm, where the l.h.s. factorized in single precision is used as a good preconditioner. Our numerical tests show that in most cases 1 iterative refinement step is enough to have almost double precision. For small systems, the mixed precision algorithm is slower than the double precision one, due to the cost of the additional solutions and residuals computations; anyway, for large systems the performance is close to the single precision one.

Algorithm 4 Factorization and solution of LQCP (mixed precision)

1: factorize and solve LQCP in single precision using Algorithm 1 with $PI = 1$, obtaining (x, u, π)
2: **for** $it\_ref \leftarrow 1 \rightarrow IT\_REF\_MAX$ **do**
3: compute the residuals in double precision using Algorithm 3, obtaining (r_s, r_q, r_b)
4: Solve LQCP in single precision using Algorithm 2 with $PI = 1$ and $(s, q, b) = (r_s, r_q, r_b)$ as r.h.s, obtaining $(\Delta x, \Delta u, \Delta\pi)$
5: update the solution in double precision $(x, u, \pi) \leftarrow (x, u, \pi) + (\Delta x, \Delta u, \Delta\pi)$
6: **end for**

4. IP METHODS FOR THE LINEAR MPC PROBLEM

The linear MPC problem (3) is an instance of the general quadratic program

$$\begin{aligned} \min_z \quad & \tfrac{1}{2} z' H z + g' z \\ s.t. \quad & Az = b \\ & Cz \geq d \end{aligned}$$

that can be solved by means of an interior-point (IP) method. In this paper, we consider the primal-dual IP and Mehrotra's predictor-corrector primal dual IP (in the following, predictor-corrector IP), see Nocedal et al. [2006] for details about the algorithms.

4.1 Primal-dual IP method

In the primal-dual IP, at each iteration k of the IP method it has to be solved a system of linear equations in the form

$$\begin{bmatrix} H + C'(T_k^{-1}\Lambda_k)C & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} y_k \\ \pi_k \end{bmatrix} = \\ = -\begin{bmatrix} g - C'(\Lambda_k e + T_k^{-1}\Lambda_k d + T_k^{-1}\sigma\mu_k e) \\ b \end{bmatrix} \quad (6)$$

where t_k are the slack variables, π_k and λ_k are the Lagrangian multipliers of the equality and inequality constraints, μ_k is the duality measure, σ is a centering parameter and e is a vector on ones. In case of the linear MPC problem, (6) is the KKT system of an instance of the LQCP (1), see Rao et al. [1998]. This means that (6) can be solved using Algorithm 1.

4.2 Predictor-corrector IP method

In case of the predictor-corrector method, at each iteration of the IP method two systems of linear equations have to be solved, respectively for the computation of the predictor and of the corrector search directions. These systems are similar to (6), and differ only for the right hand side: this means that the factorization has to be performed only once, and that they can be solved respectively using Algorithm 1 and Algorithm 2.

4.3 Inexact IP methods

In Fig. 1 we show the result of a convergence test for the duality measure in case of single, double and mixed precision used in the computation of the search direction, for both primal-dual and predictor-corrector IP methods. The fact that the single precision solution behaves as the higher precision ones till approximately 10^{-6} suggest that we can implement an inexact IP method (proposed for MPC problems by Shahzad et al [2010], with MINRES to compute the search direction), where the inexact search direction is computed by solving the LQCP subproblems using Algorithm 1 in single precision, exploiting the higher performance of single precision computation. Numerical tests shows that a value of the duality measure of 10^{-5} is a good threshold value between single and higher precision.

5. NUMERICAL RESULTS

In this section we test the HPMPC toolbox, that is our implementation of the solvers family presented in this paper. In a first part, we compare different implementations of the proposed solver for the LQCP; in the second part, we assess the performance of the proposed IP methods for the linear MPC problem. The test problem is the mass-spring system, see Domahidi et al. [2012].

The test machine is a laptop equipped with the processor Intel Core i7 3520M @ 2.9 GHz (up to 3.6 GHz in turbo mode), running Linux Xubuntu 13.04. The compiler is gcc 4.7.3. All tests are performed on one core.

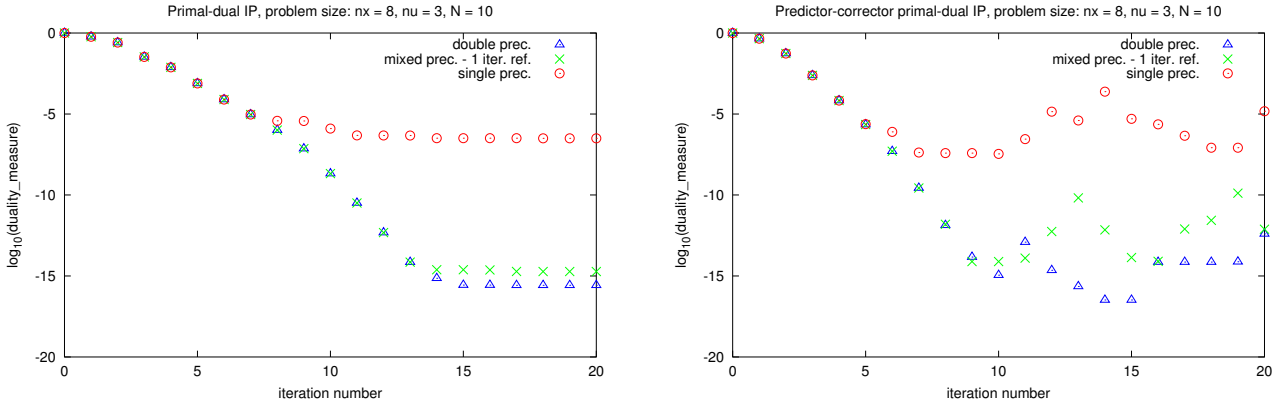
In Frison et al. [2014] we have already shown that the approach based on SSE and AVX micro-kernels gives high-performance on a number of Intel and AMD architectures.

5.1 LQ control problem

In this part we compare different approaches to implement the solver for LQCP proposed in Algorithm 1: a version making use of an highly-optimized BLAS (OpenBLAS version 0.2.6), a version using customized BLAS-like routines and AVX micro-kernels, and a version using customized BLAS-like routines and triple-loops, all of them in double, single and mixed precision with 1 iterative refinement step.

The results are in Fig. 2, where we assess the performance in Gflops of the different implementations. The processor theoretical peak performance in single precision is computed as 3.6 GHz * 2 floating-point instructions per clock (one add and one mul) * 8 flops per floating-point instruction (single precision, AVX instructions) = 57.6 Gflops. In double precision, AVX instructions can perform 4 flops per floating-point instruction, so the peak performance is the half, 28.8 Gflops.

The use of an highly-optimized BLAS library gives high-performance only for large systems, since it needs to perform a number of operations (e.g. copies of data in contiguous and aligned memory, blocking for different memory levels) that heavily impact performance for small matrices, while are well amortized for large ones. As a result, the performance is really poor for small systems. The code is implemented making explicit use of SIMD instructions, so the performance in single precision is



(a) Primal-dual IP.

(b) Predictor-corrector IP.

Fig. 1. Convergence test for the proposed IP methods, where the search direction is computed in single, double or mixed (with 1 iterative refinement step) precision. The test problem is the mass-spring system with $n_x = 8, n_u = 3, N = 10$. The combination of single precision above $\mu = 10^{-5}$ and higher (double or mixed) precision below is an inexact IP method that gives the same solution as an exact IP method, but requires less computation time.

higher that in double; the cross-over between mixed and double precision is around $n_x = 60$.

The triple-loop based approach can reach only a small fraction of the peak performance (even if the loops size is fixed at compile time), and the obtained performance is almost identical in single and in double precision. As a consequence, it can outperform BLAS only for very small systems. Furthermore, there is no advantage in using mixed precision, that would be always worse that double.

The proposed AVX micro-kernel based approach can attain a large fraction of the peak performance in both single and double precisions for a wide range of problem sizes. For small problems, this approach outperforms both optimized BLAS and triple-loop bases approach, and the performance increases quickly with the problem size. The maximum performance is attained at $n_x = 160$ in double precision (19.89 Gflops, 69% of peak) and $n_x = 128$ in single (40.19 Gflops, 70% of peak) and mixed (34.37 Gflops, 60% of peak) precisions. For larger problems, there is a certain degradation in performance, since memory footprint exceeds cache size, and our code does not perform blocking for cache. Anyway, for this size BLAS is high-performing, and the algorithm calling BLAS can be used instead.

5.2 Linear MPC problem

Here we assess the performance of the different IP methods for the linear MPC problem (3). We tested exact IP methods in single, double and mixed, and inexact ones in single+double and single+mixed precisions, where the threshold between single and higher precisions is $\tau = 10^{-5}$, for both primal-dual and predictor-corrector IP methods.

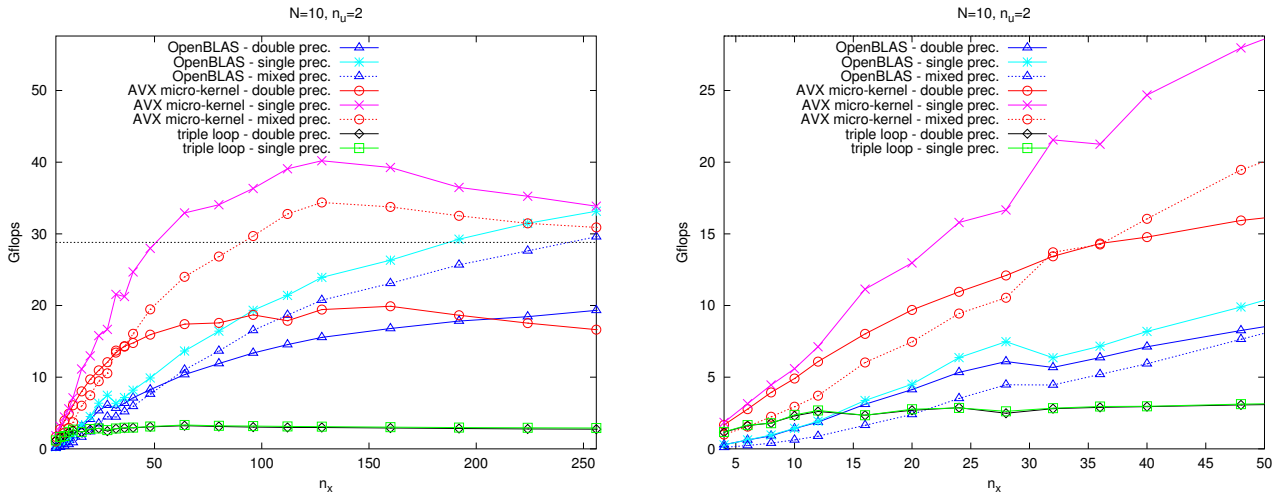
The results are in the table in Fig. 3. Since the factorization of the KKT matrix is the most expensive part in IP algorithms, the behavior of the IP methods closely resembles the behavior of Algorithm 1 in the different precisions. Single precision is always the fastest. Among higher precisions, the best results are usually obtained for inexact IP methods with the combination single+double for small problems, and single+mixed for large problems. For the largest problem, the use of inexact IP method and mixed precision requires a computational time slightly larger than the single precision, with the same accuracy as the double precision.

Whether primal-dual IP or predictor-corrector IP is the most efficient choice is problem dependent: the one has a lower cost per iteration, the other requires less iterations. Anyhow, in general primal-dual IP may be the best choice for small problems, and it takes more advantage of mixed precision computation.

Comparing the results in the table in Fig. 3 with the ones in Domahidi et al. [2012], we can see that the solvers of our HPMPC solvers family are several times faster than state-of-the-art solvers such as FORCES, CVXGEN and CPLEX, and that the performance gap increases with the problem size.

6. CONCLUSION

In this paper, we presented an efficient algorithm for the solution of the linear-quadratic control problem (LQCP). The fact that this algorithm performs only few function calls to linear-algebra routines was exploited to implement them using high-performance computing techniques, such as blocking for registers, SIMD instructions and customized BLAS. These high-performance routines were used as building blocks in solvers for the LQCP in single, double and mixed precision. In turn, the LQCP solvers were used as routines in IP methods, and in particular we proposed the use of inexact IPs where the inexact search direction is obtained solving the LQCP in single precision. This approach gives a solution in double precision, while exploiting the higher performance of single precision computation on modern architectures. An implementation of these solvers, HPMPC, is several times faster than state-of-the-art solvers for MPC. As future work, we plan to add multi-thread support, and optimize the code for embedded architectures (e.g. Intel Atom, ARM, PowerPC).



(a) Large scale.

(b) Small scale.

Fig. 2. Performance of different implementations of the proposed LQCP solver: triple-loop based, micro-kernel based and optimized BLAS based, in single, double and mixed precision. Figure (a) (respectively (b)) is scaled along the y axis to have theoretical single (respectively double) precision peak performance at turbo frequency at the top of the picture, 57.6 Gflops (respectively 28.8 Gflops).

			HPMPC: primal-dual IP					HPMPC: predictor-corrector IP					FORCES#	
n_x	n_u	N	s	d	m	s+d	s+m	s	d	m	s+d	s+m	s	d
4	1	10	0.04	0.04	0.06	**0.04**	0.04	0.05	**0.05**	0.10	0.05	0.08	0.08	0.11
8	3	10	0.09	0.09	0.16	**0.09**	0.11	0.14	**0.14**	0.26	0.14	0.20	0.29	0.33
12	5	30	0.41	0.51	0.71	**0.44**	0.48	0.60	0.71	1.13	**0.65**	0.84	1.67	2.00
22	10	10	0.41	0.55	0.62	**0.45**	0.47	0.53	0.70	0.91	**0.62**	0.73	2.90	3.25
30	14	10	0.80	1.09	1.16	**0.90**	0.92	0.99	1.34	1.63	**1.18**	1.32	6.70	7.23
60	29	30	10.05	18.23	14.11	12.45	**11.23**	11.58	20.35	18.42	16.16	**15.02**	153.02	143.80
90	44	30	29.80	60.04	42.23	38.53	**33.57**	32.67	65.59	52.67	50.54	**43.05**	*	*

Fig. 3. Proposed primal-dual and predictor-corrector IP methods (run time in [ms] for 10 iterations, averaged over 100 random initial states): exact IP methods in single (s), double (d) and mixed (with 1 iterative refinement step) (m) precision; inexact IP methods in double (s+d) and mixed (with 1 iterative refinement step) (s+m) precision, where the threshold between single and higher precisions is $\mu = 10^{-5}$. Bold represents the fastest high-precision solver for each problem size. Notice that the first 6 problems are taken from Domahidi et al. [2012]: the proposed predictor-corrector IP method is from 2 (1st problem) to about 10 (6th problem) times faster than FORCES, that in turns is faster that CPLEX and CVXGEN. # FORCES code is compiled using gcc 4.6.3, with optimization flags -O2 -mavx -funroll-loops. * The code for the larger problems could not be downloaded, since the connection to the server drops due to the long code generation time.

REFERENCES

Bemporad, A., Morari, M., Dua. V., Pistikopoulos, E.N. (2002). The Explicit Linear Quadratic Regulator for Constrained Systems. *Automatica*.

Buttari, A., Dongarra, J., Langou, J., Langou, J., Luszczek, P., Kurzak J. (2007). Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems. *The International Journal of High Performance Computing Applications*, 21(4), 457-466.

Domahidi, A., Zgraggen, A., Zeilinger, M.N., Morari, M., Jones, C.N. (2012). Efficient Interior Point Methods for Multistage Problems Arising in Receding Horizon Control. *proc. IEEE CDC 2012*

Frison, G., Jørgensen, J.B. (2013). Efficient Implementation of the Riccati Recursion for Solving Linear-Quadratic Control Problems. *proc. IEEE MSC 2013*.

Frison, G., Sørensen, H.H.B., Dammann, B, Jørgensen, J.B. (2014). High-Performance Small-Scale Solvers for Linear Model Predictive Control. *proc. IEEE ECC 2014*.

Jørgensen, J.B. (2005). *Moving Horizon Estimation and Control*. Ph.D. thesis, Department of Chemical Engineering, Technical University of Denmark, Denmark.

Mattingley, J, Boyd, S. (2012) CVXGEN: a Code Generator for Embedded Convex Optimization. *Optimization and Engineering*, 12(1):1-27.

Nocedal, J., Wright, S.J. (2006) *Numerical Optimization*. Springer, New York, 2nd edition.

Rao, C.V., Wright, S.J., and Rawlings, J.B. (1998). Application of interior-point methods to model predictive control. *Journal of Optimization Theory and Applications*, 99(3), 723-757.

Shahzad, A., Kerrigan, E.C., Constantinides, G.A. (2010). A Fast Well-conditioned Interior Point Method for Predictive Control. *proc. IEEE CDC 2010*.

Wang, Y. and Boyd, S. (2008). Fast model predictive control using online optimization. *proc. IFAC W.C.*

Chapter 6

Paper [37]

2014 IEEE International Conference on Control Applications (CCA)
Part of 2014 IEEE Multi-conference on Systems and Control
October 8-10, 2014. Antibes, France

Efficient Implementation of Solvers for Linear Model Predictive Control on Embedded Devices*

Gianluca Frison[1], D. Kwame Minde Kufoalor[2], Lars Imsland[2], and John Bagterp Jørgensen[1]

Abstract— **This paper proposes a novel approach for the efficient implementation of solvers for linear MPC on embedded devices. The main focus is to explain in detail the approach used to optimize the linear algebra for selected low-power embedded devices, and to show how the high-performance implementation of a single routine (the matrix-matrix multiplication `gemm`) can speed-up an interior-point method for linear MPC. The results show that the high-performance MPC obtained using the proposed approach is several times faster than the current state-of-the-art IP method for linear MPC on embedded devices.**

I. Introduction

Embedded Model Predictive Control (MPC) is about implementing MPC algorithms on embedded hardware. This is in contrast to the traditional approach where MPC is regarded as a high-level controller implemented in a PC or server-based technology. Due to the high computational demands of MPC and the comparably limited computational resources on embedded devices, obtaining a high performance MPC is not a trivial task. The key to overcome the challenges of embedded MPC on resource-limited devices is to employ efficient algorithms that exploit the computational performance capabilities of the target platform.

In the MPC literature, there are three approaches to obtain a fast online solution of linear MPC problems: explicit MPC, first-order methods, and second-order methods. Explicit MPC [1] exploits the fact that the solution of the MPC problem is piecewise affine over a polyhedral partition, and that it can be computed off-line for all possible initial states. As a drawback, the number of regions grows exponentially with the problem size, making this approach feasible only for problems with very few states and a short horizon.

First-order methods are variants of the gradient method [9]. The main advantages of this class of solvers are that each iteration is extremely cheap (the most expensive part is a matrix-vector multiplication), and they are easy to warm-start. First-order methods can also easily exploit sparsity of the QP problem arising from MPC and their computations are easy to parallelize. However, the number of iterations can vary orders of magnitude for different initial states. Also, the matrix-vector multiplication is memory-bounded (i.e. limited by the memory operations involved and thus the memory access speed) in modern computer architecture, and hence it can attain only a low fraction of peak performance. High-performance gradient methods have been implemented on FPGAs [7], and an efficient primal-dual first-order method for MPC is implemented on a PLC in [8].

Second-order methods (such as the interior point (IP) method proposed in this paper and active-set methods) are based on the Newton method. Making use of second-order information in the computation of the search direction, they usually need less iterations to converge, and the number of iterations does not change much with the initial state (especially for IP methods). On the other hand, each iteration requires a considerable amount of work compared to first-order methods: if BLAS is used, level 3 BLAS is required. This means that the linear algebra is more complex, but it can be optimized to attain a large fraction of peak performance on modern processors, and to take advantage of multiple cores. Implementation itself is thus very important for this class of solvers.

In this paper we explicitly target embedded devices. Thanks to the widespread diffusion of mobile computing, there is a race to build increasingly faster, low-power processors. In particular, the mass-market of smartphones has the potential to provide the scientific community with plenty of cheap and powerful embedded devices. In this paper, we consider three architectures: the low-power x86 Intel Atom (found in many netbooks), the ARM Cortex A9 (found in many smartphones and development boards), and the PowerPC 603e (an old architecture, but still present in many embedded devices for control). This paper has two key contributions: it explains how to optimize the linear algebra for these embedded devices; and it shows that a high-performance implementation of a single routine (the matrix-matrix multiplication `gemm`) can speed-up the entire IP method for the linear MPC. The resulting software (that we call HPMPC) is several times faster than the current state-of-the-art IP method for linear MPC on embedded devices, FORCES [2], for a suite of benchmark test problems.

The paper is organized as follows. Section II introduces the linear MPC problem and the unconstrained sub-problem. Section III summarizes the main implementation techniques employed to optimize the `gemm` micro-kernel. Section IV describes in detail the test architectures. Section V contains the results of comparison tests with FORCES, and finally section VI contains the conclusion.

[1] Technical University of Denmark, DTU Compute - Department of Applied Mathematics and Computer Science, DK-2800 Kgs Lyngby, Denmark. {giaf,jbjo}@dtu.dk

[2] Department of Engineering Cybernetics, Norwegian University of Science and Technology, O.S. Bragstads plass 2D N-7491 Trondheim, Norway. {kwame.kufoalor, lars.imsland}@itk.ntnu.no

* The research leading to these results has benefited from collaboration within the European Union's Seventh Framework Programme under ECGA No. 607957 TEMPO – Training in Embedded Predictive Control and Optimization. The financial support offered by the Research Council of Norway and Statoil for D. K. M. Kufoalor's research work is also gratefully acknowledged.

II. Problems

A. LQ control problem

The LQ control problem (LQCP) is formulated as

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

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

$$\begin{aligned} \varphi_n(x_n, u_n) &= \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}' \begin{bmatrix} R_n & S_n & s_n \\ S_n' & Q_n & q_n \\ s_n' & q_n' & \rho_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} = \mathcal{X}_n' \mathcal{Q}_n \mathcal{X}_n \\ \varphi_N(x_N) &= \begin{bmatrix} u_N \\ x_N \\ 1 \end{bmatrix}' \begin{bmatrix} 0 & 0 & 0 \\ 0 & P & p \\ 0 & p' & \pi \end{bmatrix} \begin{bmatrix} u_N \\ x_N \\ 1 \end{bmatrix} = \mathcal{X}_N' \mathcal{P} \mathcal{X}_N \end{aligned} \tag{2}$$

All matrices can be dense and time variant. We assume that the matrices $\mathcal{Q}_n$ and $\mathcal{P}$ are symmetric positive definite.

B. Linear MPC problem

Using the same definitions in (1) and (2), the linear MPC problem with linear constraints is the quadratic program

$$\begin{aligned} \min_{u_n, x_n} \quad & \phi = \sum_{n=0}^{N-1} \varphi_n(x_n, u_n) + \varphi_N(x_N) \\ s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \\ & x_0 = \bar{x}_0 \\ & C_n x_n + D_n u_n \geq d_n \\ & C_N x_N \geq d_N \end{aligned} \tag{3}$$

III. Implementation techniques

This section briefly describes the main optimization techniques used to implement our code: more details can be found in [3].

A. Computation bottleneck: `gemm` micro-kernel

In this paper, we employ an interior-point (IP) method for the solution of the linear MPC problem (3). The main computational bottleneck in an IP method is typically the computation of the search direction. In the linear MPC problem, this can be computed by solving LQCPs in the form (1). In turn, the LQCPs can be solved by means of the Riccati-like recursion presented in [4], where the most expensive linear algebra routines are part of level 3 BLAS.

All cubic operations in level 3 BLAS can be implemented as two loops around a `gemm` (general matrix-matrix multiplication) micro-kernel, following the approach proposed in [10]. The `gemm` computes $\mathcal{C} := \mathcal{A}\mathcal{B} + \mathcal{C}$, where $\mathcal{C} \in \mathbb{R}^{m \times n}$, $\mathcal{A} \in \mathbb{R}^{m \times k}$, and $\mathcal{B} \in \mathbb{R}^{k \times n}$. This micro-kernel corresponds to the innermost loop in the classical triple-loop implementation of level 3 BLAS, and it is the only part of the code that needs to be carefully optimized for the specific architecture. In our implementation, all the rest of the linear MPC solver is built around this assembly-coded micro-kernel, that is used for the most expensive computations.

B. Blocking for registers

This is the most important optimization and has a dual aim: reduce memory movements and hide operations latency.

In modern architectures, the cost (time) to move data (in the following memop, memory operation) from main memory to the CPU is much higher than the cost to perform a floating-point operation (flop). Level 3 BLAS performs $\mathcal{O}(n^3)$ flops on $\mathcal{O}(n^2)$ data, and thus every matrix element is accessed $\mathcal{O}(n)$ times: if each access needs a fetch from main memory, the implementation is memory-bounded. On the other hand, if faster memories such as registers and cache are exploited to reuse data, the flops/memops ratio increases.

In our implementation we block for registers, and an $m \times m$ sub-matrix stored in the registers increases the flops/memops ratio by a factor m. We do not block for cache, since this requires further information like Translation Look-aside Buffer (TLB) entry capacities [6], and improves the performance only for large matrix sizes, while embedded MPC problems are usually small/medium in size.

About hiding operations latency, in modern architectures floating-point operations are pipelined, and thus typically an instruction can be issued at every clock cycle (throughput), but the result is available only after a certain number of clocks (latency). As a result, instructions can be performed 'in parallel' keeping the pipeline busy only if there are no dependencies between them. Blocking for registers can be used to have enough independent instructions to hide latency and keep the floating-point units busy.

C. Use of contiguous memory

The use of contiguous memory helps exploiting the available memory bandwidth and improves cache reuse. When an element is fetched from memory, data is moved into cache in chunks (called cache lines) of typically 32 or 64 bytes. This means that the access to immediately following elements is faster, since the corresponding cache line is already in cache, and there is no need to fetch a new cache line for each element. A technique used to better exploit cache is packing of matrices such that elements are stored in memory in the same order as they are accessed by the `gemm` micro-kernel.

D. Use of SIMD

Many modern architectures have single-instruction multiple-data (SIMD) instruction sets: e.g. Intel and AMD have SSE and AVX instruction sets, while ARM has NEON. These are instructions that operate on small vectors of m elements, improving the performance up to m times (if the code is not memory-bounded). Regarding the machines considered in this paper, Intel Atom has the 4-wide SSE, and ARM Cortex A9 the 4-wide NEON, both capable of operating on small vectors of 4 floats, while doubles are processed as scalars. Thus the theoretical peak performance is higher in single than in double precision [5]. The AltiVec SIMD instruction set is available for some PowerPCs, but not for the PowerPC 603e. Compilers are often not very good in automatic vectorization, so there is an advantage in making explicit use of SIMD.

E. Target: embedded devices

The target processors are embedded devices. These are typically low cost and low power machines, that lack advanced features. This means that lower-level details of the architecture should be considered in the implementation. Tested processors lack out-of-order execution and register renaming, so instruction scheduling matters and should be carefully chosen by writing the micro-kernel code in inline assembly (or hope that the compiler makes a good job). They lack hardware prefetch as well, so software prefetch should be used to help hiding latency of L2 cache or main memory.

IV. Test architectures

A. Intel Atom

In this paper we consider the original Intel Atom processor (Bonnell micro-architecture). This is a relatively recent x86 architecture (2008), but has many features typical of older architectures, used to reduce the power consumption.

Our test machine is a netbook equipped with the popular N270 processor. It is a 32-bit processor with a thermal design power (TDP) of 2.5 W, and runs at 1.6 GHz; there are 24 KB L1 data cache, 32 KB L1 instruction cache and 512 KB L2 cache. The processor supports hyper-threading and has the SSE, SSE2 and SSE3 instruction sets. It is an in-order processor (i.e. instructions are performed in the same order as in the source code).

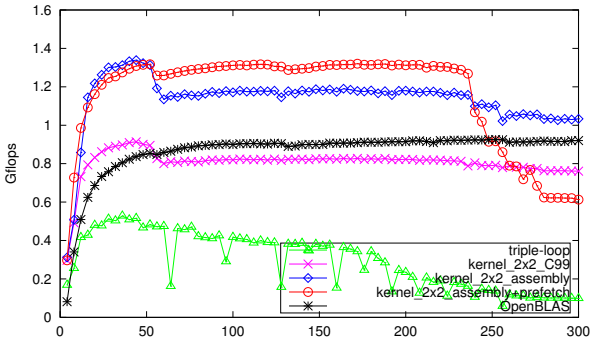
A 32-bit x86 processor has 8 floating-point registers: 4 of them are used to store a sub-matrix of $\mathcal{C}$, while the other 4 are to store elements from $\mathcal{A}$ and $\mathcal{B}$ and intermediate results.

The double precision SIMD are implemented in the SSE2 instruction set that can operate on small vectors of 2 doubles. However, in the Bonnell architecture the SSE2 vector multiply instruction is implemented as two sequential scalar multiplies. This means that SSE2 SIMD is actually slower than scalar code, since scalar instructions can be better reordered. Thus the best performance is obtained using scalar code: 4 registers are used to store a 2×2 sub-matrix of $\mathcal{C}$, and then a 2×2 micro-kernel is chosen.

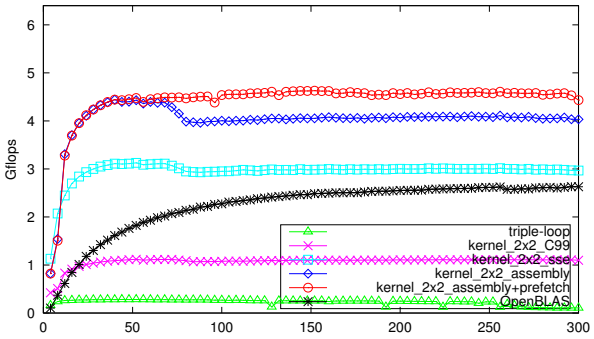
The single precision SIMD are implemented in the SSE instruction set, that is implemented properly and can operate on small vectors of 4 floats at a time. A 4×4 micro-kernel is chosen since 4 registers can hold a 4×4 sub-matrix of $\mathcal{C}$.

SSE or SSE2 instruction sets do not support fused multiply-add, adding further instruction dependences. The Atom architecture can issue an addition every cycle, but a multiplication only every other cycle. This limits the theoretical peak performance to 1 floating-point operation per cycle because in linear algebra routines there is an equal number of additions and multiplications. At 1.6 GHz it means 1.6 Gflops (where Gflops $=$ [flops/cycle] $\cdot$ [clock in GHz]) in double precision (scalar operations) and 6.4 Gflops in single precision (4-wide SIMD).

The achieved performance is limited by the very small number of registers. The processor does not support register renaming, and thus only 4 registers are actually used to store both intermediate results and elements of $\mathcal{A}$ and $\mathcal{B}$, limiting the possibility to effectively hide latency. This limitation is in part mitigated by the fact that the x86 architecture is CISC, and one of the operands of additions and multiplication can be in memory (but all instructions take two operands, and thus one of the two is overwritten with the result).



(a) Atom dgemm.



(b) Atom sgemm.

Fig. 1: Performance test of different implementations of `gemm` for squared $n \times n$ matrices, $n \in [4, 300]$, on an Intel Atom N270. Peak performance in double (single) precision is 1.6 (6.4) Gflops.

Fig. 1 reports the performance in Gflops for different single and double precision implementations of the routine for matrix-matrix multiplication. In general, triple-loop shows a poor performance (green), while blocking for registers and packing the matrices into contiguous memory improves notably the performance (magenta). In single precision, an important performance boost is achieved by the use of 4-ways SIMD (cyan). Since the processor is in-order, the maximum performance is obtained by carefully reordering instructions in inline assembly (blue). However, when memory footprint exceeds L1 cache (for size around 50 in double precision) there is a certain degradation of performance. Using software prefetch, it is possible to keep the same performance also for matrices fitting in L2 cache (red). For larger matrices, the performance drops significantly. Fortunately, this happens for matrices that are larger than the matrices in most embedded MPC applications.

Our best kernel reaches up to 83% (1.34 Gflops) of theoretical peak performance in double precision and 72% (4.63 Gflops) in single precision. It clearly outperforms optimized BLAS libraries such as OpenBLAS [11].

B. ARM Cortex A9

The ARM architecture is a quite different architecture compared to x86. It is a RISC architecture (and thus supports fewer addressing modes), but it has a rich set of instructions and many registers, making code optimization easy.

Our test machine is a development board called Wandboard Quad. It has a quad-core ARM Cortex A9 CPU running at 1 GHz. The Cortex A9 processor supports out-of-order execution and register-renaming only for general-purpose registers, while the floating-point (FP) units perform in-order execution without register-renaming. Each core has 32 KB L1 data and instruction caches, and all cores share 1 MB of L2 cache. In this paper, we address one CPU core.

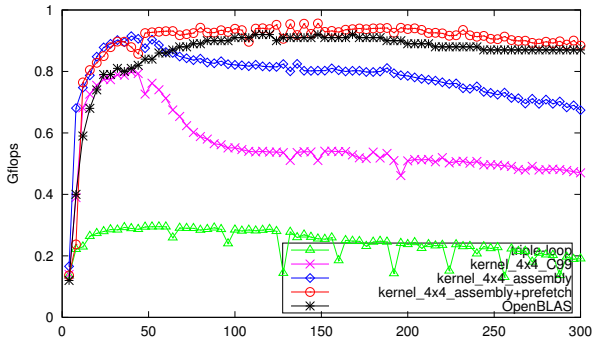
The Cortex A9 core has a scalar FP unit (VFPv3), and a SIMD unit (NEON) supporting only single-precision FP numbers (4-way SIMD). There are 32 double-word FP registers used by both VFP and NEON instructions. Each register can hold a double (registers d0-d31), while only the lower 16 registers can hold two scalar floats each (register s0-s31, where e.g. s0 and s1 are the lower and upper half of d0). Couples of consecutive d registers can be used to hold 128-bit wide vectors of 4 floats (registers q0-q15, where e.g. d0 and d1 are the lower and upper half of q0). As a result, there are 32 scalar registers (giving a 4×4 kernel for both scalar double and single precision), and 16 4-wide NEON registers (giving a 8×4 kernel for vector single precision).

The VFPv3 can operate on scalar doubles and floats. It supports fused multiply-add (FMA). In double precision, it can perform a FMA every other clock cycle, while in single precision it can perform a FMA every clock cycle. At 1 GHz, this gives for the VFP a theoretical peak performance of 1 Gflops in double and 2 Gflops in single precision.

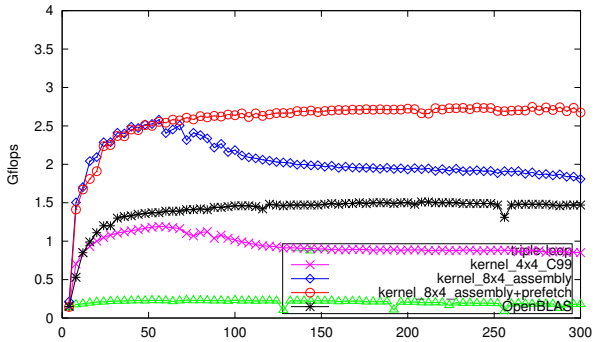
In single precision, the NEON co-processor can be used too, performing a 4-way FMA every other clock cycle, that at 1 GHz gives a theoretical peak performance of 4 Gflops.

In double precision (Fig. 2a), the large number of registers and the FMA instruction facilitate code optimization. The 4×4 kernel written in C can get about 80% if data fits in L1 cache (magenta). Reordering instructions such that memory loads are performed in idle cycles between FMAs, the performance arrives at 90%, if data fits in L1 cache (blue). The use of software prefetch gives a steady performance for data fitting in L2 cache, with maximum performance of 95%.

In single precision (Fig. 2b), if VFP is used, the performance improves with respect to double (since FMA can be performed every cycle), but does not double, since the processor seems to be unable to perform a FMA and a load in the same cycle (magenta). If NEON is used, the performance increases even more, but the best achieved performance is only about 68% of theoretical peak (blue): it looks like there is a performance penalty in mixing FMAs and loads (it is known that in the Cortex A9 there is a performance penalty mixing VFP and NEON instructions, and the two facts may be related). Again, the use of software prefetch gives a steady performance for data fitting in L2 cache (red). The achieved performance is competitive with respect to OpenBLAS (black) that does not make use of NEON.



(a) Cortex A9 dgemm.



(b) Cortex A9 sgemm.

Fig. 2: Performance test of different implementations of `gemm` for squared $n \times n$ matrices, $n \in [4, 300]$, on an ARM Cortex A9. Peak performance in double (single) precision is 1.0 (4.0) Gflops.

C. PowerPC 603e: G2_LE Core

The PowerPC target platform is the ABB AC500 PM592-ETH programmable logic controller (PLC), which has a Freescale MPC8247CVRTIEA microcontroller (SoC). The core is the G2_LE implementation of the MPC603e microprocessor. Our test PLC is equipped with 4MB RAM for user program memory and 4MB integrated user data memory.

In recent times, the increase in computational resources on PLCs and the emerging software support for the C/C++ programming language motivate the PLC implementation of optimization-based algorithms (e.g. MPC). However, the ABB PLC is a typical example of a target platform where the programmer's options for microprocessor performance exploitation are limited by a restricted list of system programming and runtime support libraries (implemented in a firmware API). The PLC software development tool, ABB PS501 Control Builder Plus 2.3 (based on the CoDeSys platform), provides programming and runtime support configurations for ANSI C89 and C99 code integrated into a PLC software/runtime architecture with a restricted set of C standard library functions. Therefore the PLC C code application consists of an IEC 61131-3 function or function block written in C. The GNU GCC 4.7.0 compiler toolchain is used to compile the C code part of the PLC application. Linking against external libraries (binaries) is not supported, implying that a library-free C code is required.

The G2_LE core is a low-power (1.5W) 32-bit RISC processor running at 400 MHz. It is equipped with independent on-chip 16 KB L1 caches for instructions and data, and on-chip memory management units (MMUs).

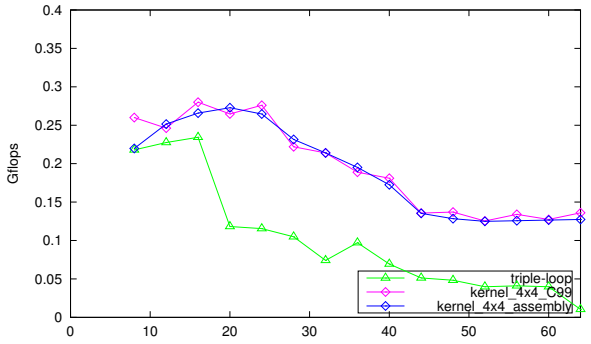
Despite the low-power design, the PowerPC G2_LE core can execute instructions out-of-order. Performance is further boosted by the superscalar architecture. A pipelined FP unit for all single-precision and most double-precision operations is also implemented, and there are 32 64-bit FP registers, each holding a single or double precision operand.

A single-precision FMA can be issued every clock cycle, whereas its double-precision counterpart every other cycle. Single-precision FMA instructions, therefore, operate faster than double-precision ones. Note that our PowerPC does not enjoy the luxury of having a floating-point SIMD instruction set, which was carefully exploited for performance boost in the Intel Atom and the ARM Cortex A9.

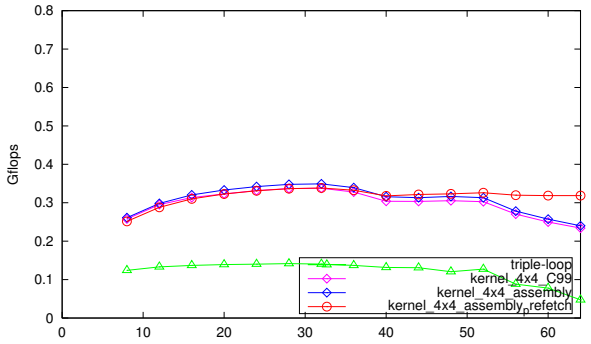
The features of the G2_LE core presented above suggest a theoretical peak of 0.4 Gflops in double and 0.8 Gflops in single-precision when the CPU is running at 400 MHz. We choose a 4×4 kernel for both double- and single-precision, and the tests results are presented in Fig. 3.

In double precision (Fig. 3a), the triple-loop version (green) can attain a good performance only for very small matrices, and performance drops significantly when the data has to be fetched from main memory (and especially for matrix size multiple of 32, due to the 4-way associativity of cache). The use of a 4×4 kernel gives a slight performance boost for matrices fitting into cache, but more importantly, it helps considerably when the memory footprint exceed cache size, since every element of $\mathcal{A}$ and $\mathcal{B}$ is used 4 times once in registers. Interestingly, the assembly coded kernel does not improve performance: FMA and the large number of registers make optimization easy, so `gcc` with -O2 already produces good code. Nevertheless, an optimized assembly code helps in case the overall code cannot be compiled with optimization flags. There are no advantages using prefetch. The maximum performance is 0.28 Gflops (70% of peak).

In single precision (Fig. 3b), our tests give a quite different picture compared to double precision. The first impression is that the performance graphs are much flatter, without the typical performance peak for data fitting in cache: the best attained performance is 0.349 Gflops (43.6% of theoretical peak). Our tests point toward the instruction fetching as the bottleneck: in fact, for $n = 32$, leaving only FMAs in the kernel loop coded in assembly, the performance ramps up to 0.45 Gflops, but leaving only memory operations the kernel execution time halves again, so memory movement is not the bottleneck either. The G2_LE core reference manual reports that the core can sustain 2 instruction fetches per clock cycle, and a memory load and a FMA can execute in parallel every clock cycle. In practice, however, the fact that the combination of loads and FMAs is slower than each of them alone is a strong argument that the core cannot co-issue load and FMA. In this framework, the performance gain of kernels compared to triple-loop is due to the lower number of memory instructions, rather than memory movements.



(a) PowerPC dgemm.



(b) PowerPC sgemm.

Fig. 3: Performance test of different implementations of `gemm` for squared $n \times n$ matrices, $n \in [8, 64]$, on an PowerPC 603e. Peak performance in double (single) precision is 0.4 (0.8) Gflops.

V. Results

In this paper we employ an interior-point (IP) method for the solution of the linear MPC problem. The current version of the software (that we call HPMPC, for High-Performance implementation of solvers for MPC) is a primal-dual IP, supporting box constraints on both inputs and states. The search direction is found by solving the LQCP (1) using the Riccati-like algorithm proposed in [4].

The key element of our implementation is that the linear-algebra routines in the LQCP solver are built around the optimized `gemm` micro-kernel. In this way, most of the computations are performed using a highly-optimized routine, tailored for the specific architecture, and the performance advantage with respect to triple-loop based implementations increases with the problems size.

In this section, we compare the performance of our software with the current (to our knowledge) state-of-the-art IP solver for linear MPC on embedded devices, FORCES [2]. FORCES makes use of code generation to build a solver tailored for the special problem instance: in particular, the linear-algebra routines are implemented as triple-loops, where the loop size is fixed, and thus the compiler can unroll the code where profitable, and perform other optimizations.

The constrained MPC problem for the comparison tests entails the control of a chain of M number of masses interconnected by springs.The problem size is defined by

TABLE I: Run times [in ms] for 10 IP iterations. The tests are the same as in TABLE VI in [2]. #: problem data too big to fit in RAM.

			Intel Atom N270 @ 1.6 GHz				ARM Cortex A9 @ 1.0 GHz				PowerPC 603e @ 0.4 GHz					
			HPMPC -O3		FORCES -O3		HPMPC -O3		FORCES -O3		HPMPC -O1		HPMPC no opt.		FORCES no opt.	
n_x	n_u	N	double	single	double	single	double	single	double	single	double	single	double	single	double	single
4	1	10	0.48	0.46	1.21	0.99	0.52	0.40	1.14	0.93	3.86	2.13	6.61	4.82	16.77	14.56
8	3	10	1.30	1.00	3.94	3.06	1.43	1.03	4.23	3.40	10.25	6.11	15.75	11.85	53.68	47.02
12	5	30	7.58	5.49	26.32	19.60	10.62	7.02	28.29	22.56	64.38	41.92	96.68	74.08	327.15	288.04
22	10	10	9.18	5.14	36.79	25.24	12.73	7.22	39.96	33.54	70.18	44.92	98.10	72.85	496.25	437.93
30	14	10	18.30	9.20	75.46	56.56	25.86	13.45	121.71	70.88	143.23	90.85	189.61	137.10	1120.54	988.38
60	29	30	332.72	130.59	1717.60	1468.38	531.34	225.07	1876.28	1373.46	#	#	#	#	#	#

$n_x = 2M$ states, $n_u = M-1$ control inputs, and the horizon N. The benchmark problem instances and the same test data used in [2] were prepared for our embedded platforms, table I summarizes the results. The same optimization flags have been used for both HPMPC and FORCES.

In the cases of Intel Atom N270 and ARM Cortex A9, the tests were easy: they run Ubuntu based operating systems (the compiler is gcc 4.6.3), and both solvers worked without any hacking. The optimization flags are `-O3 -msse3 -mfpmath=sse -march=atom` for the Atom, and `-O3 -marm -mfloat-abi=softfp -mfpu=neon -mcpu=cortex-a9` for the Cortex A9. For both architectures, the picture is alike: HPMPC is better in exploiting the processors capabilities, especially in single precision (where SIMD can be used), and the speed-up increases with the problem size, from 2x for the smaller problem up to 6x for the largest.

In case of the PowerPC 603e PLC, the tests were much harder. The first limitation is that the code has to be library-free and consist of a single C source file. Regarding HPMPC, the required hacking consisted of adding all needed files as 'headers' to the 'main'. Besides setting the compiler option `-mcpu=603e`, the maximum optimization level that could be used is `-O1`. In case of FORCES, many more hackings were needed to make the code work. The limitation that required most work in preparing the FORCES generated C code for the PLC is related to initialization of pointers. A pointer cannot be initialized by the address of another variable during declaration. That is, instructions like `float* myFloat_2 = &myFloat_1;` are not allowed. In the FORCES code, this means over 200 variable definitions must be modified, and a new function was written for initializing (setting up) all the pointers in the required format. Also, since the FORCES code generator is not open-source, the above modifications had to be done manually for each problem instance. At the end, FORCES could run on the PLC, but without using any optimization flag, not even `-O1`. Table I presents the results of 3 tests on the PLC: FORCES (without optimization), HPMPC with `-O1`, and for comparison HPMPC without optimization. The assembly coded `gemm` kernel gives HPMPC a good performance even without any optimization flag, and with `-O1` gives a speed-up from 7x to 10x compared to FORCES.

VI. Conclusion

In this paper, we proposed a novel approach to implement solvers for linear MPC on embedded devices.

We presented implementation techniques for level 3 BLAS linear algebra based on the use of optimized micro-kernels, and compared its performance to classical triple-loop based linear algebra. Furthermore, we described in detail how to optimize these micro-kernels for three different embedded architectures: Intel Atom, ARM Cortex A9 and PowerPC G2. We could get similarly high performance-per-GHz for the former two, despite being deeply different. For the latter, we could not get such a performing implementation, but we could point out the likely architectural bottleneck, and still improve the performance considerably compared to the triple-loop version.

The optimized micro-kernels have been used as the horse-power for an IP method for linear MPC. A comparison of our solver with the current state-of-the-art interior-point for MPC on embedded devices shows a considerable speed-up.

References

[1] A. Bemporad, M. Morari, V. Dua, and E.N. Pistikopoulos. The Explicit Linear Quadratic Regulator for Constrained Systems. *Automatica*, 38(1):3–20, January 2002.

[2] A. Domahidi, A. Zgraggen, M.N. Zeilinger, M. Morari, and C.N. Jones. Efficient interior point methods for multistage problems arising in receding horizon control. In *IEEE Conference on Decision and Control (CDC)*, pages 668 – 674, Maui, HI, USA, December 2012.

[3] G. Frison, H.H. B. Sørensen, B. Dammann, and J.B. Jørgensen. High-performance small-scale solvers for linear model predictive control. In *IEEE European Control Conference*, pages 128–133. IEEE, 2014.

[4] G. Frison and J.B. Jørgensen. Efficient implementation of the riccati recursion for solving linear-quadratic control problems. In *IEEE Multi-conference on Systems and Control*, pages 1117–1122. IEEE, 2013.

[5] G. Frison, L.E. Sokoler, and J.B. Jørgensen. A family of high-performance solvers for linear model predictive control. In *IFAC World Congress*, pages 3074–3079. IFAC, 2014.

[6] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), 2008.

[7] J.L. Jerez, P.J. Goulart, S. Richter, G. Constantinides, E.C. Kerrigan, and M. Morari. Embedded Predictive Control on an FPGA using the Fast Gradient Method. In *European Control Conference*, pages 3614 – 3620, Zurich, Switzerland, 2013.

[8] D. K. M. Kufoalor, S. Richter, L. Imsland, T. A. Johansen, M. Morari, and G. O. Eikrem. Embedded Model Predictive Control on a PLC using a Primal-Dual First-Order Method for a Subsea Separation Process. In *22nd IEEE Mediterranean Conference on Control and Automation*, Palermo, Italy, 2014.

[9] M. Morari S. Richter and C.N. Jones. Towards Computational Complexity Certification for Constrained MPC Based on Lagrange Relaxation and the Fast Gradient Method. In *IEEE Conference on Decision and Control*, pages 5223 – 5229, 2011.

[10] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for generating BLAS-like libraries. FLAME Working Note #66. Technical Report TR-12-30, The University of Texas at Austin, Department of Computer Science, Nov. 2012. submitted to ACM Transactions on Mathematical Software.

[11] Xianyi Zhang. Openblas. http://www.openblas.net/.

Chapter 7

Abstract [35]

Efficient solvers for soft-constrained MPC

Gianluca Frison, John Bagterp Jørgensen

Abstract— **The ability of easily and naturally handling constraints is certainly one of the winning features of Model Predictive Control (MPC). The use of hard output constraints, however, is often not physically necessary, and furthermore it can lead to unfeasible optimization problems. One way to avoid this issue is the use of soft-constraints on the outputs (and more in generals on the states). In the soft-constrained formulation, the constraint may be violated, but incurring in a penalty cost: the optimization procedure thus avoid the violation of these constraints whenever possible. Soft-constraints are traditionally handled by introducing a decision variable for each slack variable associated with the soft-constraints. This increases the size of the dynamic system variables, and therefore the size of the optimization problem, and it increases remarkably the solution time. In this paper, we want to show that IP and ADMM methods for box-constrained MPC can be modified to handle the case of soft-constraints on the states, and at a similar cost-per-iteration. This is obtained by exploit the special structure of the KKT system of the soft-constrained MPC problem, avoiding the introduction of additional control variables. As a consequence, each iteration of the IP or ADMM methods requires the solution of an unconstrained MPC subproblem with the same size as in the case of box-constrained MPC.**

I. Introduction

Model Predictive Control (MPC) is probably the most successful advance control technique in industry [6]. It makes use of a plant model to predict the future evolution of the plant dynamic and compute an input sequence optimal with respect to some cost function. At each sampling instant, only the first input of this optimal sequence is applied to the plant, before a new input sequence is computed using the latest measurements: thus, at each sampling instant an optimization problem has to be solved in real-time. This has traditionally limited the use of MPC to system with slow dynamic, as in process or chemical industry. In recent years MPC has been successfully applied to system with fast dynamic, with sampling times also in the micro-seconds range [4]: these improvements are due to both faster hardware as well as the use of structure-exploiting algorithms.

One of the winning features of MPC is certainly its ability of easily and naturally handling constraints [5]. However, the presence of constraints makes computationally-expensive the solution of optimization problems. Therefore, algorithms exploiting special constraints formulations (e.g. box constraints) have been proposed [1], [8]. One drawback of the use of hard-constraints is that they may make the optimization problem unfeasible: this is especially true in the case of output constraints. Furthermore, often the use of hard-constraints is not physically necessary.

One way to avoid this issue is the use of soft-constraints on the outputs (and more in general on the states). In this formulation, the constraint may be violated, but incurring on a penalty cost. This is usually obtained by introducing slack variables associated with the soft constrained, and heavily penalizing them: the optimization algorithm keeps these slack variables to zero whenever possible, and violates the constraints only if necessary. Soft-constraints are usually handled by introducing a decision variable for each slack variable associated with the soft-constraints. This approach has the advantage of formulating the optimization problem in the form of an hard-constrained one. However, this comes at a cost from a computational point of view: the simple constraint structure is lost (and thus algorithms for general constraints must be employed), and furthermore the extra decision variables enter in the optimization problem as dynamic system variables, that typically contribute with a cubic term in the flop count. Recently, a different formulation has been proposed [7], avoiding the introduction of extra optimization variables: however, this comes at the cost of approximating of the soft constraint penalty

In this paper, we propose a different approach. We want to show that both IP and ADMM methods for box-constrained MPC can be modified to handle the case of soft-constraints on the states, and that the flop count increases only by a linear term. This is obtained by exploit the special structure of the KKT system associated with the soft-constrained MPC problem: new optimization variables are introduced for the slack variables, but these are not additional control variables. As a consequence, each iteration of the IP and ADMM methods requires the solution of an unconstrained MPC subproblem (accounting for cubic and quadratic terms in the flop count) with the exact same structure and size as in the case of box-constrained MPC, and that can be solved efficiently [2], [3].

Authors are with Technical University of Denmark, DTU Compute - Department of Applied Mathematics and Computer Science, DK-2800 Kgs Lyngby, Denmark. Email: {giaf,jbjo} at dtu.dk

References

[1] A. Domahidi, A. Zgraggen, M.N. Zeilinger, M. Morari, and C.N. Jones. Efficient interior point methods for multistage problems arising in receding horizon control. In *IEEE Conference on Decision and Control (CDC)*, pages 668 – 674, Maui, HI, USA, December 2012.

[2] G. Frison, H.H. B. Sørensen, B. Dammann, and J.B. Jørgensen. High-performance small-scale solvers for linear model predictive control. In *IEEE European Control Conference*, pages 128–133. IEEE, 2014.

[3] G. Frison, L.E. Sokoler, and J.B. Jørgensen. A family of high-performance solvers for linear model predictive control. In *IFAC World Congress*, pages 3074–3079. IFAC, 2014.

[4] J. Jerez, P. Goulart, S. Richter, G. Constantinides, E. Kerrigan, and M. Morari. Embedded online optimization for model predictive control at megahertz rates. *IEEE Transactions on Automatic Control*, 2013.

[5] J.M. Maciejowski. *Predictive Control with Constraints*. Prentice Hall, 2002.
[6] S.J. Qin and T.A. Badgwell. A survey of industrial model predictive control technology. *Control Engineering Practice*, 11:733–764, 2003.
[7] A. Richards. Fast model predictive control with soft constraints. In *IEEE European Control Conference*, 2013.
[8] S. Richter, C.N. Jones, and M. Morari. Real-time input-constrained MPC using fast gradient methods. In *IEEE Conference on Decision and Control*, pages 7387 – 7393, 2009.

Chapter 8

Paper [36]

2015 European Control Conference (ECC)
July 15-17, 2015. Linz, Austria

MPC Related Computational Capabilities of ARMv7A Processors

Gianluca Frison, John Bagterp Jørgensen

Abstract— **In recent years, the mass market of mobile devices has pushed the demand for increasingly fast but cheap processors. ARM, the world leader in this sector, has developed the Cortex-A series of processors with focus on computationally intensive applications. If properly programmed, these processors are powerful enough to solve the complex optimization problems arising in MPC in real-time, while keeping the traditional low-cost and low-power consumption. This makes these processors ideal candidates for use in embedded MPC. In this paper, we investigate the floating-point capabilities of Cortex A7, A9 and A15 and show how to exploit the unique features of each processor to obtain the best performance, in the context of a novel implementation method for the linear-algebra routines used in MPC solvers. This method adapts high-performance computing techniques to the needs of embedded MPC. In particular, we investigate the performance of matrix-matrix and matrix-vector multiplications, which are the backbones of second- and first-order methods for convex optimization. Finally, we test the performance of MPC solvers implemented using these optimized linear-algebra routines.**

I. Introduction

The aim of embedded Model Predictive Control (MPC) is the implementation of MPC algorithms for embedded hardware. This is a non-trivial task, requiring the repeated solution in real-time of optimization algorithms on cheap and low-power hardware. To overcome these challenges, approaches focusing on the use of efficient algorithm [4], [5], or unconventional hardware [3], [12] have been proposed in recent years.

In this paper, we focus on the computational side of the problem rather than on the algorithmic side, and on the use of conventional and widespread hardware. The aim of our work is to investigate the most efficient techniques to implement the linear-algebra routines used in optimization algorithms on embedded CPUs, in order to fully exploit hardware computational capabilities. Therefore, our work is complementary to the research effort focusing on new and more efficient algorithms, since the combination of the two would produce even faster solvers.

In the first part of the paper, we review existing implementation techniques for the linear-algebra routines arising in first- and second-order optimization algorithms for MPC. In particular, our work is based on the observation that code-generated triple-loop linear-algebra routines currently employed in embedded MPC are unable to adequately exploit the computational capabilities of modern processors. Furthermore, highly-optimized BLAS libraries make use of implementation techniques that can attain close-to-peak performance, but they are optimized for large-scale problems, and performs poorly on the small-scale problems typical of embedded MPC applications. The implementation method we propose is an attempt to adapt advanced implementation techniques recently developed in the High-Performance Computing (HPC) community to the needs of embedded optimization. The focus is on obtaining the best performance for small-scale problems.

In this context, in the second part of the paper we investigate the computational capabilities of modern ARMv7A processors, that are powerful, cheap and with low power consumption. Therefore, they are widely employed in mobile computing, and ideal candidates for use in real-time embedded optimization. In particular, we present efficient implementations of the matrix-matrix and matrix-vector multiplications, and show that their performance directly affects the performance of second- and first-order optimization methods for the solution of constrained MPC problems.

Authors are with Technical University of Denmark, DTU Compute - Department of Applied Mathematics and Computer Science, DK-2800 Kgs Lyngby, Denmark. Email: {giaf,jbjo} at dtu.dk

II. Problems definitions

In this paper, we consider efficient solvers for the Linear-Quadratic Control Problem (in the following, LQCP). It is a rather general formulation, and it arises as a subproblem in many optimization algorithm used in MPC: in particular, we consider interior-point (IPM) and alternating direction method of multipliers (ADMM) methods for linear MPC with box constraints.

A. Linear-Quadratic Control Problem (LQCP)

The LQCP is the equality constrained quadratic program

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

where

$$\begin{aligned} \varphi_n(x_n, u_n) &= \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}' \begin{bmatrix} R_n & S_n & s_n \\ S_n' & Q_n & q_n \\ s_n' & q_n' & \rho_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} = \mathcal{X}_n' \mathcal{Q}_n \mathcal{X}_n \\ \varphi_N(x_N) &= \begin{bmatrix} x_N \\ 1 \end{bmatrix}' \begin{bmatrix} P & p \\ p' & \pi \end{bmatrix} \begin{bmatrix} x_N \\ 1 \end{bmatrix} = \mathcal{X}_N' \mathcal{P} \mathcal{X}_N \end{aligned} \tag{2}$$

All matrices can be dense and time variant. $\mathcal{Q}_n$ and $\mathcal{P}$ are symmetric positive semi-definite matrices, R_n are symmetric positive definite matrices. The state and input vector dimensions are n_x and n_u; the horizon length is N.

B. MPC problem

The linear MPC problem with box constraints is the QP

$$\begin{aligned} \min_{u_n, x_n} \quad & \phi = \sum_{n=0}^{N-1} \varphi_n(x_n, u_n) + \varphi_N(x_N) \\ s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \\ & x_0 = \overline{x}_0 \\ & \underline{u}_n \leq u_n \leq \overline{u}_n \quad , \quad n \in \{0, \ldots, N-1\} \\ & \underline{x}_n \leq x_n \leq \overline{x}_n \quad , \quad n \in \{1, \ldots, N\} \end{aligned} \tag{3}$$

where $\varphi_n(x_n, u_n)$ and $\varphi_N(x_N)$ are defined in (2).

III. General optimization techniques

In this section we present some general optimization techniques for the `gemm` and `gemv` routines. A good reference for matrix multiplication algorithms and definition of BLAS levels is [9].

A. Recent history of optimized `gemm`

The `gemm` routine is the general matrix-matrix multiplication routine, used to multiply two matrices A and B without assuming anything on their structure, i.e.

$$C \leftarrow \beta \cdot C + \alpha \cdot A \cdot B.$$

It is the most important level 3 BLAS routine, since all level 3 BLAS routines (and as a consequence factorizations in LAPACK) can be implemented calling `gemm` [11]. It is commonly used as a benchmark to evaluate the performance of linear-algebra libraries. Much work has been done in researching the most efficient way to implement `gemm`, and more in general linear-algebra routines. Here we review some recent work to put our approach into perspective.

The ATLAS (Automatically Tuned Linear Algebra Software) project [15] is an instantiation of the AEOS (Automated Empirical Optimization of Software) paradigm. It provides an optimized implementation of BLAS that typically is much faster than the reference BLAS from Netlib [1]. The main idea is that the library depends on a number of parameters to adapt to the different architecture features, such as number of registers and cache size. During installation, the performance is automatically and empirically tuned on the specific machine by performing an optimization over the parameter space. The code of the original library is written in C and depends on compilers to exploit different ISA (instruction set architecture); recent versions may use also hand-optimized kernels. It employs block for registers and for different levels of cache, copy of data into aligned memory, and a block-wise matrix format (if matrices are large enough to justify the copy). The original `gemm` kernel is used to multiply squared sub-matrices fitting in L1 cache, where the left operand is transposed and the right is not-transposed. This scheme optimizes the memory access of scalar instructions, but it is not effective in case of SIMD instructions (present nowadays on all architectures).

A rather different approach is used in the GotoBLAS library [10]. Here the focus in on using analytical insight to choose relevant architecture parameters, on minimizing TLB (Translation Lookaside Buffer) misses and streaming panels (i.e. sub-matrices where one dimension is big and the other is small) of data from L2 cache, instead of blocking for L1 cache. This is obtained by using registers to hold a sub-matrix of C (and thus reusing elements from A and B once on registers) such that the memory bandwidth between L2 cache and registers is large enough to hold the stream of data. Furthermore, registers and software prefetch are employed to hide latency of memory access from L2 cache. TLB misses are minimized by carefully rearranging data in memory such that elements are stored contiguously in the same order as they are accessed by the `gemm` kernel, and by considering TLB size in blocking for L2 cache. The computationally most expensive part of the code (the 'inner-kernel') is hand-written in optimized assembly for different architectures. The GotoBLAS inner-kernel consist of the three innermost loops of a layered approach, and it is therefore relatively big. GotoBLAS is typically faster than ATLAS and competitive with vendor's implementations: its performance is usually very close to the floating-point (FP) theoretical peak performance. GotoBLAS is no more under development, but a fork, OpenBLAS [16], provides optimized BLAS for recent architectures.

A recent effort to simplify the development of high-performance BLAS implementations is BLIS (BLAS-like Library Instantiation Software) [14]. It aims at providing a framework to quickly develop BLAS libraries for new architectures by focusing on code-reuse and portability. BLIS simplifies GotoBLAS's approach by splitting the inner-kernel in two: the micro-kernel (i.e. the innermost loop) and a portable macro-kernel (consisting of two loops around the micro-kernel). The micro-kernel computes a sub-matrix of C by using two panels from A and B, and it is the only part of the code that needs to be carefully hand-optimized. Only one `gemm` variant is covered by the micro-kernel, namely 'NT' (A not-transposed and B transposed): this is the optimal variant using SIMD instructions, since it avoids reductions and duplication operations in the innermost loop. This `gemm` micro-kernel is used to implement all level 3 BLAS by properly copying and transposing data matrices, and by using small routines for the corner cases.

In optimized BLAS implementations, the focus is usually on large-scale performance, and small-scale performance can be poor due to the overhead of memory copy and unnecessary blocking. Therefore, BLAS is not often used in embedded MPC, since most problems in this field are small to medium scale. An approach that has been widely used instead in embedded MPC is code generation of linear-algebra routines. It exploits knowledge from the MPC problem to generate a solver tailored to a special problem size. Since the size of each matrix is known, this knowledge can be used to perform optimizations at both generation and compilation time. Here we review two approaches that have been used recently. For the purposes of this review of optimized `gemm`, we are solely interested in the linear-algebra implementation techniques.

Code generation for embedded MPC gained widespread attention thanks to CVXGEN [13]. Knowledge about problem size is used to fully unroll all loops in a Netlib-style triple-loop implementation of linear-algebra routines. All indexes are precomputed at generation time, and there are no branches in the code. CVXGEN then relies on the compiler to optimize the generated code for the target hardware. The main disadvantage of this approach is that the code size grows with the cube of the problem size, since all triple loops are fully unrolled. This approach thus does not make use of instruction cache (since there is no code reuse), and compilation time can take a long time and possibly fail.

The FORCES [4] solver uses a different approach to code generation. For the solution of the unconstrained MPC subproblems, it makes use of a block-wise Cholesky factorization of a block tridiagonal matrix, where all blocks have equal size $n_x \times n_x$. Linear-algebra routines are implemented using Netlib-style triple-loops, but loop sizes are fixed and hard-coded at code-generation time. In this way, the compiler can decide to unroll where profitable. The main drawback of this approach is that it completely relies on the compiler for the code optimization: even if loop sizes are fixed, compilers are usually not able to properly optimize the code (e.g. `gcc` is unable to vectorize it), and thus this approach can typically attain only a small fraction of FP peak performance.

B. Implementing high-performance `gemm` for MPC

The approach we propose for implementation of the linear-algebra is novel and it is an attempt to adapt the advanced implementation techniques developed in the HPC community to the needs of embedded MPC applications.

Problems solved in embedded MPC are usually small to medium scale and need to be solved as fast as possible on cheap hardware. In most cases, MPC solvers are called at each sampling time for sequences of problems with constant structure. This has been exploited in the code generation framework by tailoring the solver to the specific problem. However, this requires a new solver to be generated for each problem instance, and this may be time-consuming.

Optimized BLAS implementations are typically libraries working for all matrix size, and they can attain a large fraction of the theoretical FP peak performance. However, they are optimized for large-scale problems, and thus their small-scale performance is usually poor. This is due to the overhead of memory copy and unnecessary blocking for caches.

We present an approach that is based on the following observations:

- The implementation technique commonly employed in embedded MPC (code-generated triple-loop based linear algebra) is unable to adequately exploit hardware capabilities, since this compiled code can attain only a small fraction of FP peak performance [6].
- Embedded MPC problems are small-medium scale, such that their overall data structure can often fit in LLC (last level cache), or at least each single data matrix can fit in LLC (matrix size up to a couple hundreds).
- GotoBLAS shows that, in the `gemm` kernel, data can be streamed from LLC fast enough to feed execution units, if it is properly arranged in memory.
- Solution methods for constrained optimization such as IPM or ADMM require many solutions to systems of linear equations, reusing the same data matrices many times. Therefore packing of matrices needs to be done only once, well amortizing its cost.

Previous papers [6]–[8] document some of the intermediate steps leading to the approach we propose. However, in the present paper we want to present the final result of this research, and discuss the intermediate steps from this perspective.

Due to the difficulty or inability of compilers to auto-vectorize triple-loop linear algebra, the first step (in the following, Step #1) has been to explicitly employ vectorization by means of instrinsics, together with a micro-kernel based approach similar to the one employed in BLIS (this approach is used for the tests in [8]). Common optimization techniques such as block for registers have been employed. This already requires detailed knowledge about the hardware (e.g. ISA, number of registers, alignment requirements). The performance shows a big improvement with respect to triple-loop based implementations, but suffers from some limitations. Vector instructions in many architectures require memory to be aligned to 128- or 256-bit boundaries to be efficiently loaded into registers. This requires data matrices to have a special structure (first element aligned, and leading dimension multiple of alignment requirement), or to copy them into this format. Tests show that performance obtained using this approach is good for data fitting in L1 cache, but already decrease for data fitting in LLC. Furthermore, performance can suddenly get very poor for some problem sizes, due to cache associativity: for these matrix sizes, elements in contiguous columns are mapped in the same cache set, effectively acting as a reduction in cache size. Performance is further affected by TLB misses, occurring since non-contiguous memory is accessed by micro-kernels.

To overcome these limitations, we considered to rearrange data in memory in a better way (Step #2). Since we assume that data can fit in LLC, we do not employ blocking for cache, but instead arrange matrix elements in the same order such as the `gemm` kernel access them. Our `gemm` kernel is analogous to BLIS's micro-kernel, but with some important differences. We decided to use a panel-wise matrix format as the default matrix layout in all our code (also at MPC solvers level), so the panel width (in the following b_s, for block size) has to be the same for all operand matrices. As a consequence, the kernel size $m_r \times n_r$ has the constraint that m_r has to be a multiple of n_r, or the other way around. The values of m_r and n_r are architecture-dependent and a function of the number of registers as well as the SIMD width. The value of b_s is usually chosen as the smaller of m_r and n_r, such that every time a cache line is accessed, it is fully utilized.

Fig. 1 shows the panel-wise matrix layout and the behavior of the `gemm` kernel. The `gemm` kernel computes the product

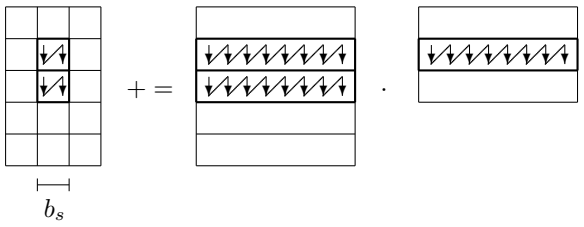


Fig. 1: Matrix layout in memory: matrix elements are stored in the same order such as the `gemm` kernel accesses them. The panels width b_s is the same for the left and the right matrix operand, as well as for the result matrix.

only in the variant 'NT' (i.e. $C = C + A \cdot B'$, where the left operand is not-transposed and the right operand is transposed). This is the optimal variant in a SIMD machine, and MPC algorithms are designed to naturally use this variant as much as possible, or to explicitly transpose matrices when strictly necessary. In the example in Fig. 1, $b_s = 2$ and the kernel is 4×2: this means that two panels of A and one panel of B are streamed to compute 4×2 elements of C. Notice that the result matrix C is automatically stored in panel-wise format at no extra cost. In the MPC framework, this means that only original data matrices eventually need to be converted into panel-wise format, while all internal matrices are automatically computed in this format.

Linear-algebra routines are implemented as two loops around this `gemm` kernel, with level-2 BLAS specialized routines handling corner cases and the solution phase in the Cholesky factorization. This means that the only routine that has to be hand-optimized to get good performance is the `gemm` kernel, that is explicitly called by all level-3 BLAS and LAPACK routines. The code used for numerical experiments in [7], [8] employs this implementation scheme. This approach gives steady and close-to-peak performance for medium-scale problems, but for small-scale problems it does not improve performance much compared to the approach in Step #1. In fact, an optimal streaming of data from LLC does not improve the performance for matrices already fitting in L1 cache. For such small matrices, the performance is influenced by the number of kernel calls, because of their constant cost before (e.g. zero the accumulation registers) and after (e.g. reductions or permutations of accumulation registers, update and store of final result) the main loop (that is the only part of the kernel accounting for flops).

This observation leads to the implementation scheme currently employed in our code (Step #3). The focus in on improving small-scale performance by merging linear-algebra routines and by designing specialized kernels for these merged linear-algebra routines, such that each element of the result matrices is accessed only once. As an example, let us consider the `syrk` and `potrf` operations in line 4 and 5 of Algorithm 1 in [6]. Using the approach presented in Step #2, the first operation computes a matrix as the product of a matrix and the transposed of the same matrix (one call to `gemm` kernel for each $\mathcal{L}$ sub-matrix), that is then Cholesky factorized (one call to `gemm` kernel and one call to either Cholesky factorization or matrix system solution routines for each $\mathcal{L}$ sub-matrix). In total, three routines are called for each $\mathcal{L}$ sub-matrix, and as a consequence the relative memory is loaded and stored three times. Instead, we can consider the new merged routine `syrk_potrf`, such that a single kernel is used to compute and factorize each $\mathcal{L}$ sub-matrix (and then the relative memory is loaded and stored only once). In this approach, the `gemm` kernel is not called explicitly, but it enters in the code of more complex kernels. This implementation of Algorithm 1 in [6] shows a speed-up of a factor 2 for small-scale problems, compared to the approach in Step #2, without compromising the performance for larger problems.

C. Implementing high-performance `gemv` for MPC

The `gemv` is the general matrix-vector multiplication routine, used to multiply a matrix and a vector without assuming anything about the structure of the matrix, i.e.

$$y \leftarrow \beta \cdot y + \alpha \cdot A \cdot x.$$

Two variants are considered, namely 'N' (the A matrix is not-transposed) and 'T' (the A matrix is transposed), and their implementation takes into account the panel-wise matrix format in Fig. 1. In the matrix-vector multiplication, there is no reuse of the elements of the matrix A, while the elements of the vector x are accessed several times. Since this routine is usually memory-bounded (i.e. the bottleneck is the memory bandwidth, and not the computation throughput), performance can be improved by reusing each element of x several times, once loaded into registers. This is achieved by employing blocking for registers to compute multiple elements of y at a time.

In the 'N' variant, the A matrix is accessed in panels (and thus the panel-wise format is optimal for this routine too). Each element of y is computed using the scalar-times-vector product: each x element has to be broadcast to an entire vector register and multiplied by a column-vector from A. No reduction is needed, but it may be necessary to use several accumulation registers to hide the latency of FP operations. Software prefetch is often beneficial, but processor with good hardware prefetch should detect the regular access pattern.

In the 'T' variant, the A matrix is accessed across panels (and thus the panel-wise format is not optimal for this routine), but (if several elements of y are computed at a time) several columns in the same panel are used contiguously, before moving to the next panel. Each element of y is computed using the dot-product: each column-vector from A is multiplied with a vector of consecutive elements from x, and the result is stored in a different accumulation register for each y element. At the end, reduction is needed to compute the final value for each y element. Software prefetch is fundamental, given the complex access pattern. The performance of this variant is usually lower than the 'N' variant, due to the need for reduction and the sub-optimal matrix format.

IV. Optimizing linear-algebra for the ARMv7A architecture

ARMv7 is the last 32-bit ARM architecture. It is divided into three profiles: A (application), R (real-time) and M (micro-controller). The A profile is intended for the most computationally intensive applications and provides features such as MMU (memory management unit) needed by modern operating systems (OS). ARMv7A is a RISC architecture, therefore it supports only simple addressing modes, and the operands of algebraic and FP operations must be in registers (and not memory). It has 16 32-bit GP (general purpose) registers and it supports two ISA: ARM (32-bit instructions encoding) and Thumb-2 (16-bit instruction encoding, allowing for higher compiled code density). ARM processors are well known for their low power consumption and widely used in embedded applications.

FP support is not mandatory, but de-facto it is present on all implementations for commercial devices such as smartphones, tablets and development boards. The FP unit can be considered a co-processor, having its own registers, pipelines and data paths. There are two FP instruction sets: VFP and NEON.

VFP give support to both double and single precision scalar operations. The most widely used variant has a set of 32 64-bit registers (`d0-d31`), each holding a double-precision FP number. The lower 16 d-registers can dually be seen as 32 32-bit registers (`s0-s31`).

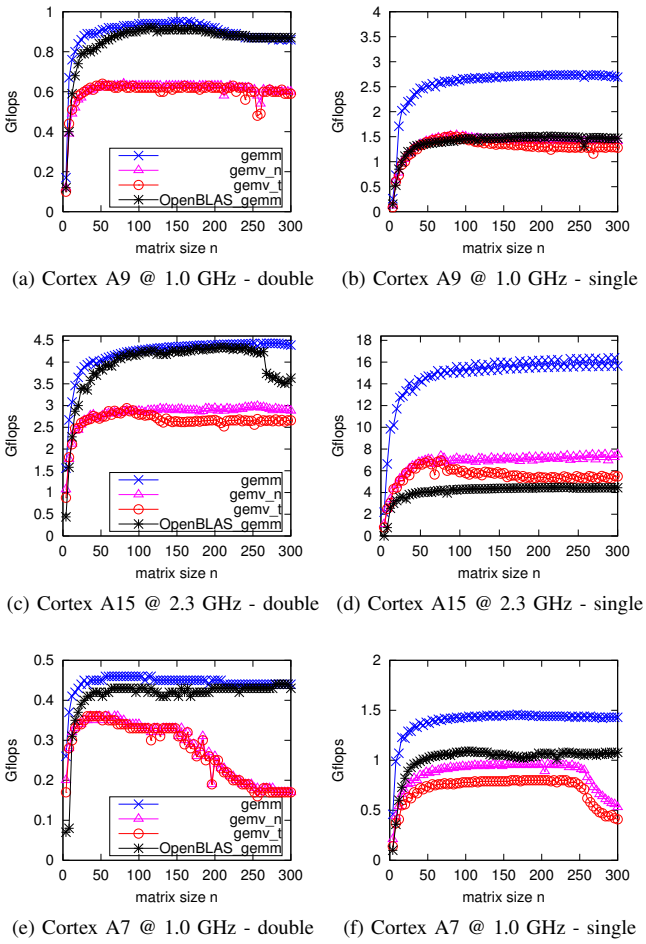
NEON is a SIMD instruction set, providing vectorization for 8-, 16-, 32- and 64-bit integers and for 32-bit FP numbers. It operates on a set of 16 128-bit registers (`q0-q15`), each consisting of a couple of consecutive d-registers. This dual view of s-, d- and q- registers makes coding of corner cases much easier and more natural than in x86, avoiding the need for shuffle instructions and using the available register space more effectively. NEON instructions can operate on small vectors of 4 or 2 single-precision FP numbers. It provides basic operations such as addition/subtraction, multiplication and multiply-accumulate (MLA), but lack complex operations such as division and square root (VFP has to be used instead for these operations). NEON implementation in ARMv7A in not considered fully IEEE 754 compilant since it only supports round-to-nearest mode and flushes all denormals to zero. This is not a concern in code for MPC, therefore we use NEON instructions where profitably. BLAS libraries often do not target NEON: therefore in single precision our code is much faster than OpenBLAS (Fig. 2).

In case of multicore CPU, only one core will be considered in our performance tests.

A. Cortex A9

The Cortex A9 was the higher performing ARM processor when introduced to the market (2010) as a replacement of the Cortex A8. It can be found in many SoC equipping smartphone, tablets and embedded devices.

It is a superscalar processor with an issue capability of 2 instructions per cycle (even if not all instruction combinations can be co-issued). It is the first multicore processor



(a) Cortex A9 @ 1.0 GHz - double (b) Cortex A9 @ 1.0 GHz - single

(c) Cortex A15 @ 2.3 GHz - double (d) Cortex A15 @ 2.3 GHz - single

(e) Cortex A7 @ 1.0 GHz - double (f) Cortex A7 @ 1.0 GHz - single

Fig. 2: Performance test for `gemm` (blue), and `gemv` in 'N' (magenta) and 'T' (red) variants, on different ARMv7A processors. The top of pictures is the theoretical peak performance.

from ARM, with up to 4 cache-coherent cores. There are 32 KB of both instruction and data L1 cache per core. There may be an external L2 cache shared among cores. The cache line size is 32 byte, corresponding to 4 doubles or 8 floats.

The processor supports speculative and out-of-order execution and register renaming in GP registers. However, these advanced features are not present in the FP pipeline and registers. As a consequence, we decided to code the micro-kernels using inline assembly to have full control over register allocation and instruction scheduling. VFP is present in the VFPv3 version, and FP datapath is 64-bit.

Our test machine is the development board Wandboard Quad, equipped with the i.MX6 Quad SoC from Freescale: the CPU is a quad-core Cortex A9 running at 1 GHz, and there is 1 MB of L2 cache.

In double precision, in the implementation of the `dgemm` kernel we use 16 out of 32 d-registers to hold a 4×4 sub-matrix of C. As a consequence, we choose $b_s = 4$, and one panel from both A and B has to be streamed. The other 16 registers are used to prefetch elements from A and B. The

Cortex A9 can issue a double-precision MLA every other cycle, giving a theoretical peak performance of 1 flop per cycle (1 Gflops at 1.0 GHz). Memory instructions are issued in the idle cycle between two MLAs. Software prefetch has to be used to hide L2 cache latency, since numerical tests show that there is no effective hardware prefetch. Fig. 2a shows that the performance of `dgemm` is as high as 95% of theoretical peak for matrices fitting in L2 cache, while performance starts to decrease for size about $n = 200$.

In the implementation of the `dgemv` kernels, for both the 'N' and 'T' versions we decided to use 8 registers to hold a 8×1 sub-vector of y. Given the large number of registers even larger values could be used, but with the chosen value each x element is already reused 8 times, effectively made the stream of A the bottleneck. Software prefetch is particularly important to have good performance, especially in the 'T' variant. Fig. 2a shows that the memory bandwidth from L2 cache is large enough to stream data, and that the peak performance is about 63% of theoretical peak. This value is rather large: the memory system is designed to feed the SIMD NEON units, while in double precision only scalar instructions can be used, and therefore consuming data at a lower rate.

In single precision, both the scalar VFP and the vector NEON units can be used. A single-precision VFP MLA can be issued every clock cycle (theoretical peak of 2 flops per cycle), while a 4-wide NEON MLA can be issued every other clock cycle (theoretical peak of 4 flops per cycle), as NEON can effectively execute 64-bit per cycle. We choose to use the vector NEON unit. Out ot 16 q-registers, we use 8 to hold a 8×4 sub-matrix of C. We choose again $b_s = 4$, meaning that two panels from A and one from B need to be streamed. It is well known that in the Cortex A9 there is performance penalty in mixing VFP and NEON instructions, since the FP pipeline need to be flushed when switching between the two instructions sets. However, our numerical tests show that there is a similar performance penalty in mixing memory loads and NEON instructions. As a consequence, the best performance in the `sgemm` kernel is obtained by loading all needed memory with consecutive load instructions, and then performing all MLAs on that data before loading new data. This limits the performance that can be attained in practice to about 68% (Fig. 2b), due to both the performance penalty and the inability to hide latency by loading data in the idle clock cycle between two MLAs.

In the `sgemv` implementation, we use again 8 registers to hold a 8×1 sub-vector of y in both the 'N' and 'T' variants. We employ software prefetch and we avoid mixing NEON and load instructions. In single precision, the best achieved performance is about 38% of the theoretical peak for both variants for matrices fitting in L1 cache, while there is a small performance decrease for matrices fitting in L2 cache.

B. Cortex A15

The Cortex A15 is the highest performing 32-bit processor designed by ARM. Originally designed for use in servers, it is present on the market since 2012. It can use up to 1 TB of memory thanks to the 40-bit Large Physical Address Extensions. Nowadays, it can be found in many high-end smartphones and tablets, either alone or in combination with lower-power Cortex A7.

It is a superscalar processor that can issue 3 instructions per cycle (and as an improvement over Cortex A9 can co-issue FP instructions and load instructions). It supports speculative and out-of-order execution, and register renaming. There can be up to 4 cores per cluster, with 32 KB of data L1 cache and 32 KB of instruction L1 cache per core, and up to 4 MB of integrated L2 cache per cluster. The cache line size is 64 byte, corresponding to 8 doubles or 16 floats.

The Cortex A15 has two FP units per core: VFPv4 and NEONv2. The main difference with respect to VFPv3 and NEON present in older processors like Cortex A9 is the presence of fused-multiply-accumulate instructions where the result is rounded only once, after the addition. However, this instruction does not support the vector-times-scalar format, and it has a lower throughput: we thus decide to use the MLA instruction. FP datapath is 128-bit, twice as much as Cortex A9.

Our test machine is the development board NVIDIA Jetson TK1, equipped with the 32-bit NVIDIA Tegra K1 SoC: the CPU is a quad-core Cortex A15 running at 2.3 GHz plus a low-power companion core, and there are 2 MB of L2 cache.

In double precision (Fig. 2c), the code for both `dgemm` and `dgemv` is the same as for Cortex A9, a part the fact that the half prefetch instruction are used, since the cache line size in Cortex A15 is twice the size in Cortex A9. However, since Cortex A15 can perform a double-precision MLA every cycle and can co-issue MLA and memory load, the performance is exactly twice as much as Cortex A9, i.e. 96% and 63% respectively of the theoretical peak performance of 2 flops per cycle (4.6 Gflops at 2.3 GHz).

In single precision (Fig. 2d), Cortex A15 shows even bigger improvements over Cortex A9. In fact, it can issued a 4-wide NEON MLA every cyle (NEON can effectively execute 128-bit per cycle), and furthermore there are no performance penalty in mixing VFP and NEON instructions, nor there are in mixing NEON and load instructions. Interestingly, Cortex A15 can not co-issue NEON MLA with NEON load, while it can co-issue NEON MLA with VFP load. So the best performance is obtained interleaving a NEON MLA with a VFP load, that can be both issued in the same clock cycle.

The fact that there are no performance penalty in mixing MLA and load instructions implies that it is possible to use even more registers to hold a sub-matrix of C, to reduce the memory operations further. Performance tests show that a 12×4 `sgemm` kernel performs better than a 8×4 kernel. This means that 12 out of 16 q-registers are used to hold a sub-matrix of C, while the other 4 are used for vectors from the A and B matrices. We still choose $b_s = 4$, and thus three panels from A and one panel from B are streamed. The best performance is 89% of the theoretical peak performance of 8 flops per cycle (18.4 Gflops at 2.3 GHz).

In the `sgemv` implementation, the code is analogous to the code for Cortex A9, with the differences that interleaving

MLAs and loads improves performance, and half the prefetch instructions are needed. The peak performance of 39% is less than half of the `sgemm` peak performance.

C. Cortex A7

The Cortex A7 is a low-power 32-bit processor designed by ARM. Present on the market since 2013, it can be found alone in low-end smartphones and tablets, or in combination with the Cortex A15 (big.LITTLE technology) in high-end devices. It is fully feature compatible with Cortex A15, but the design focus is on low-power consumption instead of high-performance.

It is partially superscalar, being able to double-issue only few combinations of instructions. It supports in-order execution, without any register renaming. There can be up to 8 cache-coherent cores per cluster, with 32 KB of both instruction and data L1 cache per core, and up to 1 MB integrated L2 cache. The cache line size is 64 byte for L1 data and L2 caches, and 32 byte for L1 instruction cache. It has the VFPv4 and NEONv2 FP units, and the FP datapath is 64 bit (same as Cortex A9).

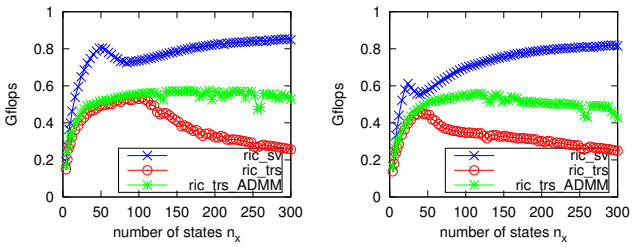
Our test machine is Cubieboard 2, a development board equipped with the Allwinner A20 SoC: the CPU is a dual-core Cortex A7 @ 1.0 GHz, with 512 KB of L2 cache.

In double precision (Fig. 2e), the code for both `dgemm` and `dgemv` is exactly the same as for Cortex A15. However, Cortex A7 can only perform a double-precision MLA every 4 cycles: as a consequence the performance-per-cycle is half of Cortex A9 and a quarter of Cortex A15, arriving at 92% and 72% respectively of a theoretical peak performance of 0.5 flops per cycle (0.5 Gflops at 1.0 GHz). Notice that the `dgemv` performance is very high for data fitting in cache: in fact, Cortex A7 has the same datapath as Cortex A9, but half the throughput.

In single precision (Fig. 2f), the Cortex A7 can perform a VFP MLA every cycle, or a NEON 4-wide MLA every 4 cycles (NEON can effectively execute 32-bit per cycle): so the theoretical peak performance is the same for VFP and NEON, namely 2 flops per cycle (2 Gflops at 1.0 GHz). However, the Cortex A7 shows the same performance penalties as the Cortex A9, and the penalty in mixing FP loads and MLA applies to both VFP and NEON. In practice, using NEON it is possible to have a slightly better performance, so the `sgemm` and `sgemv` kernels for Cortex A7 are the same as for Cortex A9, with the difference that it uses half the prefetch instructions (being the cache line twice as long). The best performance is 72% for `sgemm`, and 47% and 39% respectively for the 'N' and 'T' variants of `sgemv`.

V. LQCP SOLVERS

For the solution of the LQCP in (2), we consider the efficient Riccati recursion algorithm proposed in [6], [8]. Riccati recursion can be seen as a factorization procedure for the KKT system of (2), therefore two algorithms are presented in [8]: one to factorize the KKT matrix and solve the KKT system (Algorithtm 1), and one to solve the KKT system using an already factorized KKT matrix (Algorithm 2). These algorithms are implemented using the `gemm` and `gemv` kernels as their respective backbone. In this section, we want to show that the performance of the LQCP solvers strongly depends on the performance of these kernels.



(a) $N = 10$, $n_u = 2$ (b) $N = 50$, $n_u = 2$

Fig. 3: Performance test for the Riccati LQCP solvers: factorization & solution (blue), solution with varying b (red) and solution with fixed b (green), on an ARM Cortex A9. Tests in double precision (Peak performance is 1.0 Gflops).

In the LQCP (2), the data storage size grows approximately as $\mathcal{O}(N(n_x+n_u)^2)$: there are N stages, and at each stage the largest matrix has $\mathcal{O}((n_x+n_u)^2)$ elements. This stage-wise structure affects the performance of solvers.

Algorithm 1 makes use of level-3-BLAS-like routines (implemented using the `gemm` kernel), where each matrix element is used more than once. Fig. 3a and 3b show the performance of the double-precision version of the routine for a Cortex A9 (blue line): the performance of the Riccati recursion is higher than 80% of theoretical peak. There is a peak in performance for problems small enough to entirely fit in cache (meaning that all matrices for all N stages can fit in cache at once), followed by a small decrease in performance when data has to be fetch from main memory. Performance increases again for larger matrices, as long as each single matrix can individually fit in LLC, since each element is reused more times once moved from main menory to cache. The value and position of this peak is influenced by problem-dependent quantities such as the horizon length ($N = 10$ in Fig. 3a and $N = 50$ in Fig. 3b), as well as machine features such as memory hierarchy and FP computational capabilities.

Algorithm 2 makes use of level-2-BLAS-like routines (implemented using the `gemv` kernel), and each matrix element is used only once. As a consequence, once the problem memory footprint exceeds the LLC size, the performance keeps decreasing as the problem size increases (red line). However, if the Lagrangian multipliers are not needed (i.e. line 15 in Algorithm 2 is not executed), and if the value of $P_{n+1}b_n$ has already been computed in a previous solver call (i.e. line 3 in Algorithm 2 is not executed again), then many flops can be saved and much less memory needs to be accessed: this is the case in the ADMM algorithm. The accessed memory (but not the flops) can be further decreased if the dynamic system is time invariant, by reusing a single matrix at every stage: this keeps the performance high also for large values of n_x (green line).

TABLE I: Solution time [in ms] in double (single) precision for the MPC solvers: IPM in FORCES, IPM in HPMPC and ADMM in HPMPC. The code compiled using gcc. The OS is Ubuntu for Cortex A9 and Cortex A15, and Debian for Cortex A7.

			Cortex A7 @ 1.0 GHz			Cortex A9 @ 1.0 GHz			Cortex A15 @ 2.3 GHz		
MPC problem size			FORCES	HPMPC		FORCES	HPMPC		FORCES	HPMPC	
n_x	n_u	N	IPM	IPM	ADMM	IPM	IPM	ADMM	IPM	IPM	ADMM
4	1	10	2.24(1.90)	0.74(0.59)	0.87(0.97)	1.14(0.93)	0.53(0.47)	0.64(0.79)	0.36(0.30)	0.15(0.15)	0.16(0.25)
8	3	10	7.16(5.96)	2.14(1.38)	2.12(1.98)	4.23(3.40)	1.57(1.06)	1.49(1.54)	1.16(0.93)	0.38(0.30)	0.37(0.48)
12	5	30	47.6(36.4)	18.6(9.31)	13.2(10.4)	28.3(22.5)	10.6(6.90)	9.40(8.45)	7.21(5.46)	2.47(1.68)	2.22(2.38)
22	10	10	69.1(55.7)	21.1(9,29)	11.0(7.40)	40.0(33.5)	11.7(6.34)	7.43(5.63)	11.7(9.28)	2.69(1.43)	1.68(1.46)
30	14	10	156(126)	48.3(18.9)	19.2(16.0)	122(70.9)	23.9(11.7)	12.2(8.61)	25.1(20.9)	5.43(2.39)	2.82(2.12)
60	29	30	2256(1928)	906(340)	309(152)	1876(1373)	569(220)	173(74.8)	431(369)	116(39.1)	35.1(19.2)

VI. MPC SOLVERS

The Riccati solvers for the LQCP in (2) can be used as routines in solvers for MPC. In this paper, we consider two solvers: a Mehrotra's predictor-corrector IPM (calling both Algorithm 1 and 2 once per iteration), and an ADMM (calling Algorithm 1 only the first iteration, and then Algorithm 2 once per iteration). We use the mass-spring system as test problem and repeat the tests in [4] using the two solvers from our toolbox HPMPC [2] and the IPM FORCES. We exploit the fact that the dynamic system is linear time-invariant to reuse the same data matrices at each stage, and by computing the factorization of the KKT matrix off-line in the ADMM.

Table I provides the results. We decide to fix the number of iterations to 10 for the IPM and to 50 for the ADMM: for these values IPM and ADMM gives reasonably similar accuracy for this test problem. For very small problems, the different computational capabilities of processors are partially masked by fixed cost such as latency of memory access and pre- and post-loop operations in kernels. Similarly, there is not a big difference between single and double precision, and Algorithm 1 and Algorithm 2 have comparable costs. So ADMM is slower beacuse of the larger number of iterations. As the problem size increases, single precision becomes increasingly advantageous over double precision, and the same happens for Algorithm 2 (with a quadratic cost on state and input size) over Algorithm 1 (with cubic cost), even if the flops count is partially balanced by the higher performance of `gemm` over `gemv`. For both solvers in HPMPC the CPU times are well below 1 ms for the smallest problem, and below 1 s for the largest one, thanks to the high-performance of the `gemm` and `gemv` kernels. The generic code in FORCES can exploit hardware capabilities to a much smaller extent, especially in single precision.

Notice that the above analysis is based on the cost for a fixed number of iterations, while in practice number of iterations may vary considerably with the problem instance, especially in the case of ADMM.

VII. CONCLUSION

In this paper, we reviewed linear-algebra implementation techniques currently employed in the HPC community and in the embedded MPC community, and proposed a novel approach. This approach takes implementation techniques used in recent BLAS implementations and adapts them to the needs of embedded MPC. MPC solvers implemented using this approach can exploit hardware capabilities of processors well, and therefore attain a large fraction of theoretical peak performance. In this context, we investigated the computational capabilities of modern ARMv7A processors, and found that, regarding FP theoretical peak performance per cycle, Cortex A15 is 2x faster than Cortex A9, that in turn is 2x faster than Cortex A7. In practice, Cortex A15 appears to be of a different class (partially due to the higher clock frequency it can reach), being over 9x times faster than Cortex A7, and 6x times faster than Cortex A9 on compute-intensive workloads in single precision. The developed code is part of the open-source HPMPC toolbox [2].

REFERENCES

[1] http://www.netlib.org/blas/.

[2] HPMPC. http://www.github.com/giaf/hpmpc.

[3] G. Constantinides. Tutorial paper: Parallel architectures for model predictive control. In *IEEE European Control Conference*, 2009.

[4] A. Domahidi, A. Zgraggen, M.N. Zeilinger, M. Morari, and C.N. Jones. Efficient interior point methods for multistage problems arising in receding horizon control. In *IEEE CDC*, pages 668–674, 2012.

[5] H.J. Ferreau, H.G. Bock, and M. Diehl. An online active set strategy to overcome the limitations of explicit mpc. *International Journal of Robust and Nonlinear Control*, 18(8), 2008.

[6] G. Frison, H.H. B. Sørensen, B. Dammann, and J.B. Jørgensen. High-performance small-scale solvers for linear model predictive control. In *IEEE European Control Conference*, pages 128–133. IEEE, 2014.

[7] G. Frison, D.K.M. Kufualor, L. Imsland, and J.B. Jørgensen. Efficient implementation of solvers for linear model predictive control on embedded devices. In *IEEE Multi-conference on Systems and Control*, pages 1954–1959. IEEE, 2014.

[8] G. Frison, L.E. Sokoler, and J.B. Jørgensen. A family of high-performance solvers for linear model predictive control. In *IFAC World Congress*, pages 3074–3079. IFAC, 2014.

[9] G.H. Golub and C.F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1989.

[10] K. Goto and R.A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), 2008.

[11] K. Goto and R.A. van de Geijn. High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*, 35(1), 2008.

[12] J. Jerez, P. Goulart, S. Richter, G. Constantinides, E. Kerrigan, and M. Morari. Embedded online optimization for model predictive control at megahertz rates. *IEEE Transactions on Automatic Control*, 2013.

[13] J. Mattingley and S. Boyd. CVXGEN: a code generator for embedded convex optimization. *Optimization and Engineering*, 13(1):1–27, March 2012.

[14] F.G. Van Zee and R.A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 2013. Accepted.

[15] R.C. Whaley, A. Petitet, and J.J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27:2001, 2000.

[16] Xianyi Zhang. OpenBLAS. http://www.openblas.net/.

Chapter 9

Paper [39]

Preprints, 5th IFAC Conference on Nonlinear Model Predictive Control
September 17-20, 2015. Seville, Spain

High-Performance Small-Scale Solvers for Moving Horizon Estimation

Gianluca Frison * **Milan Vukov** ** **Niels Kjølstad Poulsen** *
Moritz Diehl **,*** **John Bagterp Jørgensen** *

* *Technical University of Denmark (email: {giaf, nkpo, jbjo}@dtu.dk)*
** *KU Leuven (email: milan.vukov@esat.kuleuven.be*
*** *University of Freiburg (email: moritz.diehl@imtek.uni-freiburg.de)*

Abstract:
In this paper we present a moving horizon estimation (MHE) formulation suitable to easily describe the quadratic programs (QPs) arising in constrained and nonlinear MHE. We propose algorithms for factorization and solution of the underlying Karush-Kuhn-Tucker (KKT) system, as well as the efficient implementation techniques focusing on small-scale problems. The proposed MHE solver is implemented using custom linear algebra routines and is compared against implementations using BLAS libraries. Additionally, the MHE solver is interfaced to a code generation tool for nonlinear model predictive control (NMPC) and nonlinear MHE (NMHE). On an example problem with 33 states, 6 inputs and 15 estimation intervals execution times below 500 microseconds are reported for the QP underlying the NMHE.

1. INTRODUCTION

Moving Horizon Estimation (MHE) has emerged as an effective option to state and parameter estimation of constrained or non-linear systems. It is found to give superior estimation performance with respect to the Extended Kalman Filter (EKF), at the cost of increased computational cost [11]: MHE requires the solution of an optimization problem at each sampling instant.

MHE can be seen as an extension of the Kalman Filter, where, beside the current measurement, a window of N past measurements is explicitly taken into account in the estimation. This makes the estimation less sensitive to the choice of the arrival cost, that rarely has an analytic expression in case of constrained or non-linear systems [18]. Furthermore, the MHE formulation can naturally and optimally take constraints into account.

From an algorithmic point of view, MHE is often considered the dual of Model Predictive Control (MPC), with the difference that the initial state is free. Therefore, algorithms for MPC have been used to solve MHE problems. In particular, a forward Riccati recursion (corresponding to a covariance Kalman filter recursion) has been proposed in [20; 14] for the solution of the unconstrained MHE subproblems. A QR factorization based, square-root forward Riccati is proposed as routine in an Interior-Point Method (IPM) for MHE in [12].

The focus of the current paper is on the computational performance of algorithms and implementations, rather than the control or estimation performance. More precisely, the focus is on the development of a fast solver for the equality-constrained linear MHE problem, specially tailored to small-scale problems. This solver is embedded in an algorithmic framework for non-linear MHE (presented in [15] and implemented using automatic code generation in [6]) and used to solve in real-time the QPs arising in equality-constrained non-linear MHE problems. The real-world test problem in Section 5.2 falls into this class of problems. Furthermore, the developed solver can be easily embedded as a routine into an IPM to solve inequality-constrained MHE problems, similarly to [9] for the MPC problem case. In an IPM, a solver for the equality-constrained MHE problem is used to compute the Newton direction, that is the most computationally expensive part of the algorithm. Hence the importance of a solver for this class of problems.

The focus on small-scale problems has important consequences on algorithmic and implementation choices. In case of small-scale dense problems (with dense meaning MPC and MHE problems where the dynamic system matrices are dense), solvers based on tailored recursions are much faster than general-purpose direct sparse solvers (see e.g. [7] for a comparison of a Riccati recursion based solver to PARDISO and MA57 direct sparse solves in the unconstrained MPC problem case). The performance gap suggests that direct sparse solvers may become competitive only for very sparse problems. In case of large-scale and sparse solvers, direct sparse solvers have been successfully applied to the MHE problem [23]. Furthermore, the focus on small-scale problems reduces the issues related to the numerical stability of the recursion schemes. It is well known that the Riccati recursion can be seen as a special stage-wise factorization of the KKT matrix of the unconstrained MPC problem. The factorization of different permutations of the KKT matrix can have better accuracy properties, especially in case of ill-conditioned problems.

In this paper, we study the applicability to the MHE problem of the efficient implementation techniques proposed in [9; 8] for the MPC problem, with special focus on small-scale performance. In particular, one of the key ingredients to obtain solvers giving high-performance for small matrices is the merging of linear algebra routines

[8]. In the case of the MHE problem, the efficient Riccati recursion implementation proposed in [9] can not be employed, since the covariance of the state estimates (needed in the computation of the state estimates in the forward-backward substitutions) is never computed explicitly, but instead only implicitly in a Cholesky factorization. Therefore, in this paper we consider a different recursion for MHE, corresponding to the information Kalman filter. This recursion can be effectively implemented using the proposed techniques, giving noticeable speedups with respect to the use of optimized BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage) libraries, as shown in Section 5.1. As a further advantage, the use of this recursion as routine in solvers for constrained MHE is straightforward, since the inversion of the information matrices (updated at each iteration of e.g. an IPM) is efficiently embedded in the recursion itself. Finally, this recursion can naturally handle state equality constraints at the last stage, that are useful to provide the controller with consistent state feedback.

Besides the use of the efficient implementation techniques proposed in [9; 8], the key difference between the solver presented in the current paper and the one presented in [12] regards the choices at the linear algebra level. Given the generic matrices A and B, the upper Cholesky factor R of the matrix $A + B' \cdot B$ can be computed efficiently using the BLAS rank-k symmetric update routine `syrk` and LAPACK Cholesky factorization routine `potrf` at a cost of $\frac{4}{3}n^3$ flops (if all matrices are of size n). Given the uniqueness of the Cholesky factorization, R can also be computed using the LAPACK QR factorization routine `geqrf`, as $\begin{bmatrix} B \\ A^{1/2} \end{bmatrix} = Q \cdot R$, at the larger cost of $\frac{10}{3}n^3$ flops, plus possibly $\frac{1}{3}n^3$ to compute $A^{1/2}$. On the other hand, the QR factorization based on Householder reflections is more accurate, since the worse-conditioned normal matrix $B' \cdot B$ is not computed explicitly. The choice between the two implementations therefore depends on accuracy and speed requirements.

2. PROBLEM FORMULATION

The aim of the MHE problem is the reconstruction of the state vectors x_k, process noise vectors w_k and measurement noise vectors v_k, given the plant model, the measurement vectors y_k for a window of past time instants $k = 0, 1, \ldots, N$ and an initial estimate of the state vector at time 0, $\bar{x}_0$, and relative covariance matrix $\widetilde{P}_0$, summarizing the contribution given by the measurements prior to time 0.

The (unconstrained) MHE problem is traditionally written as the Quadratic Program (QP)

$$\begin{aligned} \min_{x_k, w_k, v_k} \quad & \textstyle\sum_{k=0}^{N} \frac{1}{2}(v_k - \bar{v}_k)^T \widetilde{R}_k^{-1}(v_k - \bar{v}_k) + \\ & + \textstyle\sum_{k=0}^{N-1} \frac{1}{2}((w_k - \bar{w}_k)^T \widetilde{Q}_k^{-1}(w_k - \bar{w}_k) + \\ & + \tfrac{1}{2}(x_0 - \bar{x}_0)^T \widetilde{P}_0^{-1}(x_0 - \bar{x}_0) \\ s.t. \quad & x_{k+1} = A_k x_k + G_k w_k + f_k \\ & y_k = C_k x_k + v_k \end{aligned}$$

In this formulation, the inverse of the matrices in the cost function has a precise statistical interpretation: $\widetilde{R}_k$ is the covariance matrix of the measurement noise vector v_k, $\widetilde{Q}_k$ is the covariance matrix of the process noise vector w_k. The vectors $\bar{v}_k$ and $\bar{w}_k$ are the expected values of the measurement and process noises.

In this paper, we consider a different formulation of the MHE problem. Namely, we consider a QP in the form

$$\begin{aligned} \min_{x_k, w_k} \quad & \textstyle\sum_{k=0}^{N} \frac{1}{2} x_k^T Q_k x_k + q_k^T x_k + \\ & + \textstyle\sum_{k=0}^{N-1} \frac{1}{2} w_k^T R_k w_k + \\ & + r_k^T w_k + \tfrac{1}{2}(x_0 - \bar{x}_0)^T P_0 (x_0 - \bar{x}_0) \qquad (1a) \\ s.t. \quad & x_{k+1} = A_k x_k + G_k w_k + f_k \qquad (1b) \\ & y_k = C_k x_k + v_k \qquad (1c) \\ & D_N x_N = d_N \qquad (1d) \end{aligned}$$

The matrices Q_k, R_k and P_0 can be interpreted as information matrices. In general, the matrices R_k are assumed to be strictly positive definite, and the matrices Q_k and P_0 are assumed to be positive semi-definite, with the matrix $Q_0 + P_0$ strictly positive definite.

This formulation reflects the deterministic view of the MHE as the problem of finding the optimal x_k, w_k and v_k sequences in a least-square sense, with respect to some cost function. The penalization of x_k in place of v_k in the cost function (1a) is useful to account for QPs in non-linear MHE. The fact that the matrices in the cost function appear as not-inverted makes straightforward the use of a solver for this MHE formulation as a routine for constrained MHE (e.g., in an IPM these matrices are updated to take into account constraints). The inversion does not need to be performed explicitly, but instead implicitly and embedded in the solution algorithm, as shown in section 4.1.

In this formulation, we consider additional state equality constraints (1d) beside the dynamic system equations (1b). These equality constraints are used to provide consistent feedback signal to the controller. They are enforced only at the last stage to avoid Linear Independence Constraint Qualification (LICQ) problems.

The size of problem (1) is defined by the quantities: n_x (state vector size), n_w (process noise vector size), n_d (number of state equality constraints on the last stage), N (horizon lenght).

3. STAGE-WISE FACTORIZATION OF THE KKT MATRIX

The MHE problem (1) is an equality constrained QP with a special structure. For $N = 2$, the solution is obtained solving the KKT (Karush-Kuhn-Tucker) system

$$\left[\begin{array}{ccc|ccc|cc} E_0 & & A_0^T & & & & & \\ & R_0 & G_0^T & & & & & \\ A_0 & G_0 & & -I & & & & \\ \hline & & -I & Q_1 & & A_1^T & & \\ & & & & R_1 & G_1^T & & \\ & & & A_1 & G_1 & & -I & \\ \hline & & & & & -I & Q_2 & D_2^T \\ & & & & & & D_2 & \end{array}\right] \left[\begin{array}{c} x_0 \\ w_0 \\ \lambda_0 \\ \hline x_1 \\ w_1 \\ \lambda_1 \\ \hline x_2 \\ \lambda_2 \end{array}\right] = \left[\begin{array}{c} -e_0 \\ -r_0 \\ -f_0 \\ \hline -q_1 \\ -r_1 \\ -f_1 \\ \hline -q_2 \\ +d_2 \end{array}\right] \qquad (2)$$

where λ_k are the Lagrangian multipliers and

$$E_0 = Q_0 + P_0$$
$$e_0 = q_0 + P_0 \bar{x}_0.$$

The KKT matrix is symmetric, large and structured. If the structure is not exploited, it can be factorized using a dense

LDL factorization using $\frac{1}{3}((N-1)(2n_x+n_w)+n_x+n_d)^3$ flops. However, the problem structure can be exploited to greatly reduce this cost computational.

The stage-wise structure of the KKT matrix can be exploited to factorize it stage-by-stage using a forward recursion, starting from the first stage. This recursion is analogue to the Information Filter (IF) formulation of the Kalman filter proposed in [16]. The recursion can be easily generalized (at the cost of a modest increase in the solution time) to handle a cross-term S_k between x_k and w_k in the cost function. The top-left corner of the KKT matrix is

$$\left[\begin{array}{ccc|c} E_0 & & A_0^T & \\ & R_0 & G_0^T & \\ A_0 & G_0 & & -I \\ \hline & & -I & Q_1 \end{array}\right] \left[\begin{array}{c} x_0 \\ w_0 \\ \lambda_0 \\ \hline x_1 \end{array}\right] = \left[\begin{array}{c} -e_0 \\ -r_0 \\ -f_0 \\ \hline -q_1 \end{array}\right]. \tag{3}$$

If the matrix E_0 is invertible, the variable x_0 can be eliminated using the Schur complement of E_0, obtaining

$$\left[\begin{array}{cc|c} R_0 & G_0^T & \\ G_0 & -A_0E_0^{-1}A_0 & -I \\ \hline & -I & Q_1 \end{array}\right] \left[\begin{array}{c} w_0 \\ \lambda_0 \\ \hline x_1 \end{array}\right] = \left[\begin{array}{c} -r_0 \\ -f_0 + A_0E_0^{-1}e_0 \\ \hline -q_1 \end{array}\right].$$

Similarly, if the matrix R_0 is invertible, the variable w_0 can be eliminated, obtaining

$$\left[\begin{array}{c|c} -A_0E_0^{-1}A_0^T - G_0R_0^{-1}G_0 & -I \\ \hline -I & Q_1 \end{array}\right] \left[\begin{array}{c} \lambda_0 \\ \hline x_1 \end{array}\right] = \\ = \left[\begin{array}{c} -f_0 + A_0E_0^{-1}e_0 + G_0R_0^{-1}r_0 \\ \hline -q_1 \end{array}\right].$$

Finally, if the matrix $P_1^{-1} = A_0E_0^{-1}A_0^T + G_0R_0^{-1}G_0$ is invertible, the variable λ_0 can be eliminated, obtaining

$$(Q_1 + P_1)\, x_1 = -q_1 - P_1(-f_0 + A_0E_0^{-1}e_0 + G_0R_0^{-1}r_0),$$

that can be rewritten in the more compact form

$$E_1 x_1 = -e_1 \tag{4}$$

closing the recursion, since now the top-left corner of the KKT matrix is.

$$\left[\begin{array}{ccc|c} E_1 & & A_1^T & \\ & R_1 & G_1^T & \\ A_1 & G_1 & & -I \\ \hline & & -I & Q_2 \end{array}\right] \left[\begin{array}{c} x_1 \\ w_1 \\ \lambda_1 \\ \hline x_2 \end{array}\right] = \left[\begin{array}{c} -e_1 \\ -r_1 \\ -f_1 \\ \hline -q_2 \end{array}\right].$$

that is in the same form as (3). The recursion can therefore be repeated at the following stage. At the last stage, we can distinguish two cases, depending on the presence of equality constraints on the state vector at the last stage (1d).

If $n_d = 0$, the last stage looks like

$$E_2 x_2 = -e_2$$

that, if E_2 is invertible, can be easily solved to compute x_2. Notice that the information matrix E_2 of the estimate x_2 is available at no extra cost.

If $n_d > 0$, the last stage looks like

$$\begin{bmatrix} E_2 & D_2^T \\ D_2 & \end{bmatrix} \begin{bmatrix} x_2 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} -e_2 \\ +d_2 \end{bmatrix}.$$

If the matrix E_2 is invertible, the variable x_2 can be eliminated using the Schur complement of E_2, obtaining

$$(-D_2E_2^{-1}D_2^T)\lambda_2 = d_2 + D_2E_2^{-1}e_2.$$

If the matrix $D_2E_2^{-1}D_2^T$ is invertible, then the value of λ_2 can be computed, that in turn gives the value of x_2 as

$$E_2 x_2 = -e_2 - D_2^T\lambda_2.$$

The information matrix of the estimate in the null-space can be computed as

$$E_{Z,2} = Z' E_2 Z$$

where Z is a null-space matrix of D [17].

Notice that the proposed recursion requires the invertibility of the matrices R_k for $k = 0, \dots, N-1$, of the matrices $E_k = Q_k + P_k$ for $k = 0, \dots, N$, of the matrices P_k^{-1} (and then of the matrices P_k) for $k = 1, \dots, N$, and of the matrix $D_N E_N^{-1} D_N^T$. However, the matrix P_0 can be singular: in particular, it can be set to 0 if no prior information is available about the value of the estimate of x_0. Invertibility of Q_k for $k = 0, \dots, N$ and full row-rank of A_k for $k = 1, \dots, N-1$ and of D_N guarantees the invertibility of E_k for $k = 0, \dots, N$, of P_k for $k = 1, \dots, N$ and of $D_N E_N^{-1} D_N^T$.

4. IMPLEMENTATION

In this paper, the efficient implementation techniques proposed in [9; 8] for the Riccati-based solver for the unconstrained MPC problem are applied to the MHE problem (1).

4.1 Algorithm

In the MPC case, the backward Riccati recursion can be seen as a stage-wise factorization of the KKT matrix, with the recursion beginning at the last stage [19]. The key operation in the algorithm presented in [9] is the computation of $\mathcal{Q} + \mathcal{A}^T \cdot \mathcal{P} \cdot \mathcal{A}$, where $\mathcal{Q}$ is a positive semi-definite matrix. If all matrices $\mathcal{A}$, $\mathcal{P}$ and $\mathcal{Q}$ have size n, then the most efficient way to compute this operation is

$$\begin{aligned} \mathcal{Q} + \mathcal{A}^T \cdot \mathcal{P} \cdot \mathcal{A} &= \mathcal{Q} + \mathcal{A}^T \cdot (\mathcal{L} \cdot \mathcal{L}^T) \cdot \mathcal{A} = \\ &= \mathcal{Q} + (\mathcal{A}^T \cdot \mathcal{L}) \cdot (\mathcal{A}^T \cdot \mathcal{L})^T \end{aligned} \tag{5}$$

where $\mathcal{L}$ is the lower Cholesky factor of $\mathcal{P}$. Using specialized BLAS routines, the cost of this operation is $\frac{1}{3}n^3$ (`potrf`) + n^3 (`trmm`) + n^3 (`syrk`) = $\frac{7}{3}n^3$ flops.

In the MHE case, in the forward recursion presented in Section 3 the key operation is the computation of $\mathcal{Q} + \mathcal{A} \cdot \mathcal{P}^{-1} \cdot \mathcal{A}^T$, where $\mathcal{Q}$ is a positive definite matrix. Despite the presence of a matrix inversion, this operation can be computed in the exact same number of flops as the operation in (5). In fact, the matrix inversion is computed implicitly, as

$$\begin{aligned} \mathcal{Q} + \mathcal{A} \cdot \mathcal{P}^{-1} \cdot \mathcal{A}^T &= \mathcal{Q} + \mathcal{A} \cdot (\mathcal{L} \cdot \mathcal{L}^T)^{-1} \cdot \mathcal{A}^T = \\ &= \mathcal{Q} + (\mathcal{A} \cdot \mathcal{L}^{-T}) \cdot (\mathcal{A} \cdot \mathcal{L}^{-T})^T \end{aligned} \tag{6}$$

where again $\mathcal{L}$ is the lower Cholesky factor of $\mathcal{P}$. Since the matrix $\mathcal{L}$ is triangular, the operation $\mathcal{A} \cdot \mathcal{L}^{-T}$ can be computed efficiently using the routine `trsm` to solve a triangular system of linear equations with matrix RHS. Using specialized BLAS routines, the cost of this operation is $\frac{1}{3}n^3$ (`potrf`) + n^3 (`trsm`) + n^3 (`syrk`) = $\frac{7}{3}n^3$ flops. This makes the IF-like recursion in Section 3 competitive with respect to the forward Riccati recursion generally used to factorize the KKT matrix of the MHE problem.

The algorithm for the factorization of the KKT matrix (2) is presented in Algorithm 1. The algorithm can be implemented using standard BLAS and LAPACK routines: the name of the routines is in the comment to each

Algorithm 1 Factorization of the KKT matrix of the MHE problem (1)

Require:
$U_0 \quad s.t. \quad P_0 = U_0 \cdot U_0^T$

1: **for** $k \leftarrow 0, \dots, N-1$ **do**
2: $E_k \leftarrow Q_k + U_k \cdot U_k^T$ ▷ `lauum`
3: $L_{e,k} \leftarrow E_k^{1/2}$ ▷ `potrf`
4: $AL_{e,k} \leftarrow A_k \cdot L_{e,k}^{-T}$ ▷ `trsm`
5: $L_{r,k} \leftarrow R_k^{1/2}$ ▷ `potrf`
6: $GL_{r,k} \leftarrow G_k \cdot L_{r,k}^{-T}$ ▷ `trsm`
7: $P_{inv} \leftarrow AL_{e,k} \cdot AL_{e,k}^T + GL_{r,k} \cdot GL_{r,k}^T$ ▷ `syrk`
8: $L_p \leftarrow P_{inv}^{1/2}$ ▷ `potrf`
9: $U_{k+1} \leftarrow L_p^{-T}$ ▷ `trtri`
10: **end for**
11: $E_N \leftarrow Q_N + U_N \cdot U_N^T$ ▷ `lauum`
12: $L_{e,N} \leftarrow E_N^{1/2}$ ▷ `potrf`
13: **if** $n_d > 0$ **then**
14: $DL_e \leftarrow D_N \cdot L_{e,N}^{-T}$ ▷ `trsm`
15: $P_d \leftarrow DL_e \cdot DL_e^T$ ▷ `syrk`
16: $L_d \leftarrow P_d^{1/2}$ ▷ `potrf`
17: **end if**

line. The cost of the algorithm is of $N(\frac{10}{3}n_x^3 + n_x^2 n_w + n_x n_w^2 + \frac{1}{3}n_w^3) + \frac{2}{3}n_x^3 + n_d n_x^2 + n_d^2 n_x + \frac{1}{3}n_d^3$ flops. If the R_k matrices are diagonal, then operations in lines 5 and 6 can be performed in a linear and quadratic number of flops, respectively. This decreases $N(n_x n_w^2 + \frac{1}{3}n_w^3)$ flops from the complexity of the algorithm, making it linear in n_w. This is advantageous in typical situations with MHE formulations involving additive process noise.

The algorithm for the solution of the KKT system given the factorization of the KKT matrix is presented in Algorithm 2. It consists of forward and backward substitutions. Again, triangular matrices are exploited by means of specialized routines.

4.2 Merging of linear algebra routines

All linear-algebra routines are implemented using the implementation techniques presented in [9; 8]. In particular, high-performance kernels for the general matrix-matrix multiplication routine `gemm` are used as the backbone of kernels for all matrix-matrix operations and factorizations. These kernels are optimized for a number of architectures, and can attain a large fraction of the floating-point (FP) peak performance. The design focus is on performance for small-scale matrices, but the performance scales optimally for matrices of size up to a few hundreds, large enough for embedded MPC and MHE needs.

In the optimization of solvers for small scale problems, it is beneficial to merge linear algebra routines when possible, as shown in the Riccati recursion for unconstrained MPC problems in [9]. The main advantage is the reduction in the number of calls to linear algebra kernels. In fact, in our implementation linear algebra kernels are blocked for register size, and therefore they compute a sub-matrix of the result matrix with a single kernel call. If the size of the result matrix is not a multiple of the optimal kernel

Algorithm 2 Forward-backward substitution of the KKT system of the MHE problem (1)

Require:
$U_{k+1}, L_{e,k}, AL_{e,k}, L_{r,k}, GL_{r,k}, \quad k = 0, \dots, N-1$
$L_{e,N}, DL_e, L_d$

1: **for** $k \leftarrow 0, \dots, N-1$ **do**
2: $e_k \leftarrow q_k + U_k \cdot U_k^T \cdot \bar{x}_k$ ▷ `trmv`
3: $\bar{x}_{k+1} \leftarrow -f_0 + AL_{e,k} \cdot L_{e,k}^{-1} \cdot e_k$ ▷ `gemv & trsv`
4: $\bar{x}_{k+1} \leftarrow \bar{x}_{k+1} + GL_{r,k} \cdot L_{r,k}^{-1} \cdot r_k$ ▷ `gemv & trsv`
5: **end for**
6: $e_N \leftarrow q_N + U_N \cdot U_N^T \cdot \bar{x}_N$ ▷ `trmv`
7: **if** $n_d = 0$ **then**
8: $x_N \leftarrow -L_{e,N}^{-T} \cdot L_{e,N}^{-1} \cdot e_N$ ▷ `trsv`
9: **else**
10: $\lambda_N \leftarrow d_N + DL_e \cdot L_{e,N}^{-1} \cdot e_N$ ▷ `gemv & trsv`
11: $\lambda_N \leftarrow -L_d^{-T} \cdot L_d^{-1} \cdot \lambda_N$ ▷ `trsv`
12: $x_N \leftarrow -L_{e,N}^{-T} \cdot (e_N + DL_e^T \cdot \lambda_N)$ ▷ `gemv & trsv`
13: **end if**
14: **for** $k \leftarrow N-1, \dots, 0$ **do**
15: $\lambda_k \leftarrow U_k \cdot U_k^T \cdot (\bar{x}_{k+1} - x_{k+1})$ ▷ `trmv`
16: $x_k \leftarrow L_{e,k}^{-T} \cdot (-e_k - AL_{e,k}^T \cdot \lambda_k)$ ▷ `gemv & trsv`
17: $w_k \leftarrow L_{r,k}^{-T} \cdot (-r_k - GL_{r,k}^T \cdot \lambda_k)$ ▷ `gemv & trsv`
18: **end for**

size, there is a loss in performance: therefore merging small matrices into larger ones increases the likelihood of using the optimal kernel size. Furthermore, the reduction in the number of kernel calls reduces the corresponding overhead, and improves memory reuse. All these aspects are especially beneficial for small size problems.

As the problem size increases, however, the performance advantages of merging linear algebra routines become smaller, since the kernels call overhead gets amortized over a larger number of flops. On the contrary, numerical tests show that merging linear algebra routines often slightly decreases performance for large problems. This is due to the fact that merged routines operate on larger amounts of data than un-merged routines, and therefore cache size is exceeded for smaller problem sizes. The performance crossover point can be easily determined by numerical simulation, and it can be used as threshold to switch between merged and un-merged linear algebra routines.

In order to motivate the use of routine merging, let us consider a 3×3 blocked version of the operation $\mathcal{L} = (\mathcal{Q} + \mathcal{A} \cdot \mathcal{A}')^{1/2}$ in (7). The last line contains the explicit expression of the lower Cholesky factor $\mathcal{L}$: the expression for the $\mathcal{L}_{ij}$ block is in position ij in the matrix. We can see immmediately that the products $\mathcal{A}_i \cdot \mathcal{A}_j^T$ (used to compute the matrix to be factorized) are in the same form as the correction terms $-\mathcal{L}_{ik} \cdot \mathcal{L}_{jk}^T$ in the Cholesky factorization (a part the change of sign). This means that the $\mathcal{L}$ matrix can be computed sweeping it once block-wise: each block is initialized with $\mathcal{Q}_{ij}$, then updated with $\mathcal{A}_i \cdot \mathcal{A}_j^T$ and corrected with the products $-\mathcal{L}_{ik} \cdot \mathcal{L}_{jk}^T$, and finally Cholesky-factorized (diagonal blocks) or solved using a triangular matrix (off-diagonal blocks). So, diagonal blocks are computed using the merged kernel `syrk_potrf`, while the off-diagonal blocks are computed using the merged kernel `gemm_trsm`.

$$\begin{aligned}
\mathcal{L} &= \begin{bmatrix} \mathcal{L}_{00} & * & * \\ \mathcal{L}_{10} & \mathcal{L}_{11} & * \\ \mathcal{L}_{20} & \mathcal{L}_{21} & \mathcal{L}_{22} \end{bmatrix} = \left(\begin{bmatrix} \mathcal{Q}_{00} & * & * \\ \mathcal{Q}_{10} & \mathcal{Q}_{11} & * \\ \mathcal{Q}_{20} & \mathcal{Q}_{21} & \mathcal{Q}_{22} \end{bmatrix} + \begin{bmatrix} \mathcal{A}_0 \\ \mathcal{A}_1 \\ \mathcal{A}_2 \end{bmatrix} \cdot \begin{bmatrix} \mathcal{A}_0^T & \mathcal{A}_1^T & \mathcal{A}_2^T \end{bmatrix} \right)^{1/2} = \\
&= \left(\begin{bmatrix} \mathcal{Q}_{00} + \mathcal{A}_0 \cdot \mathcal{A}_0^T & * & * \\ \mathcal{Q}_{10} + \mathcal{A}_1 \cdot \mathcal{A}_0^T & \mathcal{Q}_{11} + \mathcal{A}_1 \cdot \mathcal{A}_1^T & * \\ \mathcal{Q}_{20} + \mathcal{A}_2 \cdot \mathcal{A}_0^T & \mathcal{Q}_{21} + \mathcal{A}_2 \cdot \mathcal{A}_1^T & \mathcal{Q}_{22} + \mathcal{A}_2 \cdot 2_0^T \end{bmatrix} \right)^{1/2} = \\
&= \begin{bmatrix} (\mathcal{Q}_{00} + \mathcal{A}_0 \cdot \mathcal{A}_0^T)^{1/2} & * & * \\ (\mathcal{Q}_{10} + \mathcal{A}_1 \cdot \mathcal{A}_0^T)\mathcal{L}_{00}^{-T} & (\mathcal{Q}_{11} + \mathcal{A}_1 \cdot \mathcal{A}_1^T - \mathcal{L}_{10} \cdot \mathcal{L}_{10}^T)^{1/2} & * \\ (\mathcal{Q}_{20} + \mathcal{A}_2 \cdot \mathcal{A}_0^T)\mathcal{L}_{00}^{-T} & (\mathcal{Q}_{21} + \mathcal{A}_2 \cdot \mathcal{A}_1^T - \mathcal{L}_{20} \cdot \mathcal{L}_{10}^T)\mathcal{L}_{11}^{-T} & (\mathcal{Q}_{22} + \mathcal{A}_2 \cdot \mathcal{A}_2^T - \mathcal{L}_{20} \cdot \mathcal{L}_{20}^T - \mathcal{L}_{21} \cdot \mathcal{L}_{21}^T)^{1/2} \end{bmatrix}
\end{aligned} \tag{7}$$

Having this in mind, lines 5, 6 of Algorithm 1 can be trivially merged: in fact, the `trsm` kernel is already used internally in the Cholesky factorization routine. This means that the operations in lines 5, 6 can be computed using a Cholesky-like factorizatin routine operating on rectangular matrices, as

$$\begin{bmatrix} L_{r,k} \\ GL_r \end{bmatrix} = \text{rect\_potrf} \left(\begin{bmatrix} R_k \\ G_k \end{bmatrix} \right).$$

Lines 2, 3, 4 of Algorithm 1 perform a similar operation to the one in (7), with the difference that the $\mathcal{A}$ matrix is upper triangular and the $\mathcal{Q}$ and $\mathcal{L}$ matrices are rectangular. This means that the operations in lineas 2, 3, 4 can be computed as

$$\begin{bmatrix} L_{e,k} \\ AL_e \end{bmatrix} = \text{rect\_potrf} \left(\begin{bmatrix} Q_k \\ A_k \end{bmatrix} + \begin{bmatrix} U_k \\ 0 \end{bmatrix} \cdot \begin{bmatrix} U_k^T & 0 \end{bmatrix} \right),$$

where the product $U_k \cdot U_k^T$ takes into account the fact that U_k is upper-triangular.

Notice that, if a cross term S_k is present in the cost function, then operations in lines 2, 3, 4, 5, 6, plus the additional operations related to S_k can be merged in the single routine

$$\begin{bmatrix} L_{e,k} & * \\ L_{s,k} & L_{r,k} \\ AL_e & GL_r \end{bmatrix} = \text{rect\_potrf} \left(\begin{bmatrix} Q_k & * \\ S_k & R_k \\ A_k & G_k \end{bmatrix} + \begin{bmatrix} U_k \\ 0 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} U_k^T & 0 \end{bmatrix} \right).$$

Lines 7, 8, 9 of Algorithm 1 can be merged as well. Lines 7, 8 implement the exact same operation in (7). The triangular matrix inversion and transposition in line 9 can be computed easily by considering the analogy of this operation with the `trsm` operation embedded in the Cholesky factorization. All operations in lines 7, 8, 9 can therefore be computed as

$$\begin{bmatrix} L_p \\ U_{k+1} \end{bmatrix} = \text{rect\_potrf} \left(\begin{bmatrix} 0 \\ I \end{bmatrix} + \begin{bmatrix} AL_e & GL_r \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} AL_e^T \\ GL_r^T \end{bmatrix} \right),$$

and taking into account the fact that U_{k+1} is upper triangular.

Similar arguments apply to the operations in the remaining lines 11, 12, 14, 15, 16 of Algorithm 1, and similarly the merged routine `gemv_trsv` can be used at lines 3, 4, 10, 12, 16, 17 of Algorithm 2.

5. NUMERICAL TESTS

5.1 Performance tests

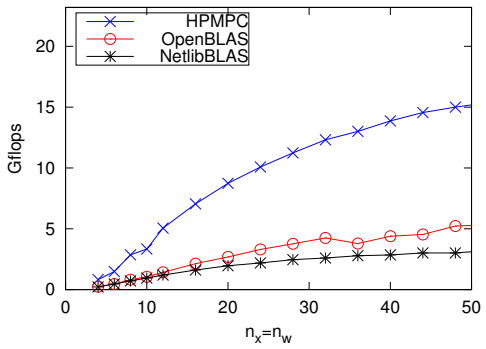
The results of the tests reported in this section assess the performance of the proposed MHE solver when implemented using different libraries for linear algebra. Namely, the implementation using the custom and merged linear algebra routines presented in section 4.2 (that is part of the HPMPC toolbox [1]) is compared against two open source BLAS libraries: OpenBLAS and the Netlib BLAS.

OpenBLAS [3] is an highly optimized BLAS implementation, providing code tuned for a number of architectures. It is a fork of the successful (and now unsupported) GotoBLAS [10], and it supports also the most recent architectures. It makes use of a complex blocking strategy to optimize the use of caches and TLBs (Translation Lookaside Buffer), and key routines are written in assembly using architecture-specific instructions. Its performance is competitive against vendor BLAS. The version tested in this paper is the 0.2.14.

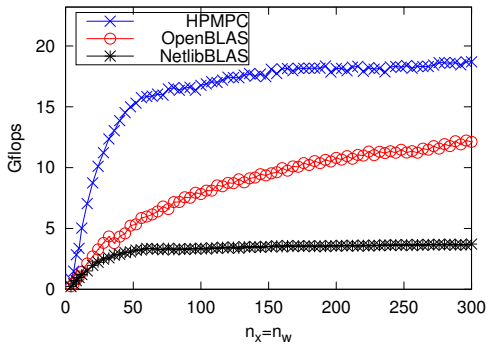
Netlib BLAS [2] is the reference BLAS. It is written in Fortran code and it is generic, not targeting any feature of specific architectures. It does not perform any blocking strategy, and level-3 routines are written as simple triple loops. The performance is usually poor for large matrices.

The test machine is a laptop equipped with the Intel Core i5 2410M processor, running at a maximum frequency of 2.9 GHz. The operating system is Linux Ubuntu 14.04, with `gcc` 4.8.2 compiler. The processor has 2 cores and 4 threads (however, only single-thread code is considered in our tests). The processor implements the Sandy Bridge architecture, supporting the AVX instruction set (that operates on 256-bit vector register, each holding 4 double or 8 single precision FP numbers). The Sandy Bridge core can perform one vector multiplication and one vector addition each clock cycle, and therefore in double precision it has a FP peak performance of 8 flops per cycle (that at 2.9 GHz gives 23.2 Gflops).

In Fig. 1 there is the result of a performance test. On the small scale (Fig. 1a), the performance of the HPMPC version is much better than both BLAS versions, and it can attain a large fraction of the FP peak performance for problems with tens of states. On the medium scale (Fig. 1b), the performance of HPMPC is steady at around 75-80% of FP peak, while the performance of the Netlib BLAS version is steady at around 15% of FP peak. On the other hand, the performance of OpenBLAS increases with the problem size. For even larger problems, the performance of unblocked implementations (HPMPC and Netlib BLAS) would decrease, while the performance of the OpenBLAS implementation would be steadily close to FP peak. Such large problem sizes are however of limited interest in embedded MHE, and therefore the HPMPC implementation gives the best performance for relevant problem sizes.

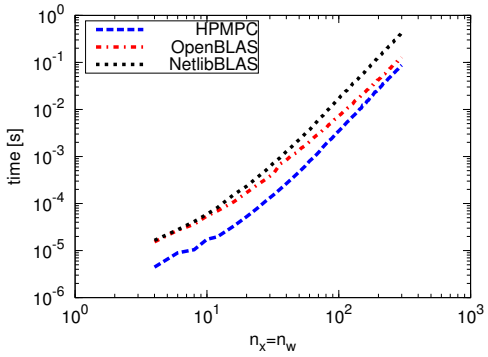


(a) Small scale.

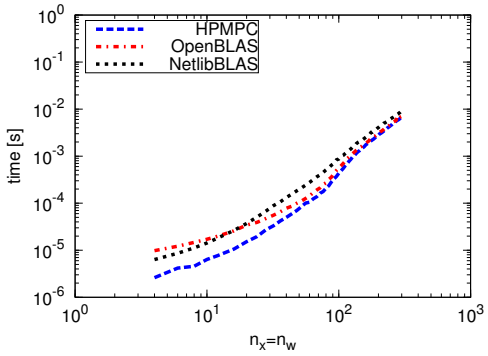


(b) Medium scale.

Fig. 1. Performance test for the proposed MHE KKT matrix factorization algorithm, assuming $S_k = 0$ and R_k dense. The performance in Gflops is represented as a function of $n_x = n_w$, while $N = 10$ and $n_d = 0$ are fixed. Top of the picture is the FP peak performance of the processor.



(a) Factorization time.



(b) Substitution time.

Fig. 2. Execution time for the proposed MHE KKT matrix factorization algorithm and forward-backward substitution algorithm, assuming $S_k = 0$ and R_k dense. The execution time in seconds is represented as a function of $n_x = n_w$, while $N = 10$ and $n_d = 0$ are fixed.

In Fig. 2 there are the running times for the factorization Algorithm 1 (Fig. 2a) and for the forward-backward substitution Algorithm 2 (Fig. 2b), in the three implementations using HPMPC, OpenBLAS and Netlib. In both the factorization and the substitution cases, the HPMPC implementation has a big advantage for small problems. In the factorization case, HPMPC retains the performance advantage over the Netlib BLAS version also for larger problems, while the the OpenBLAS version reduces the performance gap. In the substitution case, for larger problems the performance of the three implementations gets very similar. This is due to the fact that Algorithm 2 is implemented using level 2 BLAS, where matrices are streamed and there is no reuse in matrix elements. Therefore for large problems the substitution time is dominated by the cost of streaming matrices from main memory, that is the same for all implementations.

5.2 Nonlinear MHE and MPC in closed loop: real-time numerical simulations

In the following we present the strength of the presented solver for MHE for state estimation and control of a nonlinear system. Namely, we present results of closed-loop real-time simulations of rotational start-up for an airborne wind energy system [22]. The system is modeled as a differential-algebraic equation (DAE), with 27 differential states, 1 algebraic state and 4 control inputs. To solve the nonlinear MPC (NMPC) and nonlinear MHE (NMHE) formulations we use the ACADO Code Generation Tool (CGT) [13] that implements the real-time iteration (RTI) scheme [4; 15]. The QP underlying the NMHE solver is solved using the implementation presented in Section 4, while the QP underlying the NMPC solver is handled with an efficient implementation from [9].

An augmented model used for the NMHE, one that includes a disturbance model, has $n_x = 33$ states and $n_w = 6$ disturbance inputs. Consistency conditions of the DAE model yield $n_d = 9$ equality constraints, while the number of estimation intervals is $N = 15$. On the other hand, the NMPC formulation has $N = 50$ intervals. For more details, we refer to [21] and references therein.

The simulation results are reported in Figure 3. A control interval begins with a feedback step of the RTI scheme for the NMHE (MHE FBK), after which the current state estimate is obtained. Afterwards, the NMPC feedback step is triggered (MPC FBK) for calculation of optimal control inputs. In essence, the execution times of the feedback steps amount to solutions of underlying QPs. After each feedback step corresponding preparation step is executed (MHE PREP and MPC PREP), which includes model integration, sensitivity generation and linearization of the objective and the constraints. In this setting both NMHE and NMPC run on the separate CPU cores.

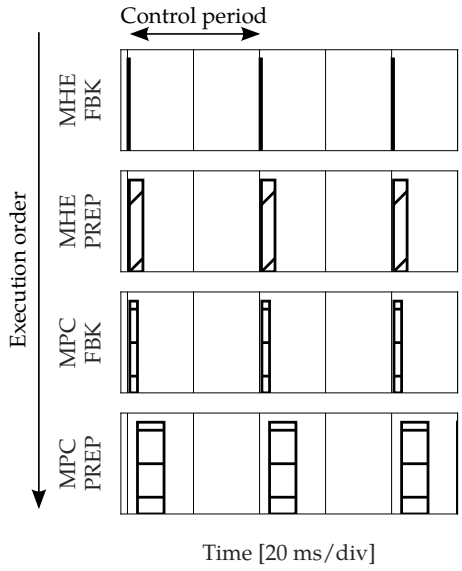


Fig. 3. Feedback and preparation step times for the MHE and MPC using the ACADO-HPMPC solver in the rotational start-up of an airbone wind energy system.

The solution times for the feedback step of the NMHE are always less than 500 μs, and the maximum feedback times for the NMPC are always less than 3 ms. In total, the maximum feedback delay is always less than 3.5 ms, far below the control period of 40 ms. Note that in [21] qpOASES [5] solver is used to solve the QPs underlying the same NMHE formulation. In that case the feedback step of the NMHE alone requires about 3.5 ms, i.e. nearly seven times more than with the MHE QP solver proposed in this paper.

6. CONCLUSION

In this paper, we presented an information Kalman filter recursion for the MHE problem, that can be easily used as routine in constrained and non-linear MHE. Furthermore we proposed efficient implementation techniques tailored to this recursion form, with special focus on small-scale performance. The resulting solver is shown to give noticeable performance improvements when compared to the same algorithm implemented using optimized BLAS and LAPACK libraries. Furthermore, the solver has been used to solve QPs underlying a nonlinear MHE formulation and provides state estimates necessary for control of a challenging non-linear system in less than 500 μs.

REFERENCES

[1] HPMPC. https://github.com/giaf/hpmpc.git.
[2] Netlib BLAS. http://www.netlib.org/blas/.
[3] Openblas. http://www.openblas.net/.
[4] M. Diehl. *Real-Time Optimization for Large Scale Nonlinear Process*. PhD thesis, Universität Heidelberg, 2001.
[5] H. J. Ferreau, H. G. Bock, and M. Diehl. An online active set strategy to overcome the limitations of explicit mpc. *International Journal of Robust and Nonlinear Control*, 18(8), 2008.
[6] H.J. Ferreau, T. Kraus, M. Vukov, W. Saeys, and M. Diehl. High-speed moving horison estimation based on automatic code generation. In *IEEE Conference on Decision and Control*, 2012.
[7] G. Frison and J. B. Jørgensen. Efficient implementation of the riccati recursion for solving linear-quadratic control problems. In *IEEE Multi-conference on Systems and Control*, pages 1117–1122. IEEE, 2013.
[8] G. Frison and J. B. Jørgensen. MPC related computational capabilities of ARMv7A processors. In *IEEE European Control Conference*. IEEE, 2015.
[9] G. Frison, H. H. B. Sørensen, B. Dammann, and J. B. Jørgensen. High-performance small-scale solvers for linear model predictive control. In *IEEE European Control Conference*, pages 128–133. IEEE, 2014.
[10] K. Goto and R. A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), 2008.
[11] E. L. Haseltine and J. B. Rawlings. Critical evaluation of extended kalman filtering and moving-horizon estimation. *Ind. Eng. Chem. Res.*, 8(44):2451–2460, 2005.
[12] N. Haverbeke, M. Diehl, and B. De Moor. A structure exploiting interior-point method for moving horizon estimation. In *IEEE Conference on Decision and Control*, pages 1273–1278. IEEE, 2009.
[13] B. Houska, H. J. Ferreau, and M. Diehl. An Auto-Generated Real-Time Iteration Algorithm for Nonlinear MPC in the Microsecond Range. *Automatica*, 47(10):2279–2285, 2011.
[14] J. B. Jørgensen, J. B. Rawlings, and S. B. Jørgensen. Numerical methods for large-scale moving horizon estimation and control. In *Int. Symposium on Dynamics and Control Process Systems*, 2004.
[15] P. Kühl, M. Diehl, T. Kraus, J. P. Schlöder, and H. G. Bock. A real-time algorithm for moving horizon state and parameter estimation. *Computers & Chemical Engineering*, 1(35), 2011.
[16] G. O. Mutambara. *Decentralized estimation and control for multi-sensor systems*. CRC press, 1998.
[17] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.
[18] C. V. Rao, J. B. Rawlings, and Q. Mayne. Constrained state estimation for nonlinear discrete-time systems: Stability and moving horizon approximations. *IEEE Transactions on Automatic Control*, 48(2), 2003.
[19] M. C. Steinbach. A Structured Interior-Point SQP Method for Nonlinear Optimal Control Problems. In *Computational Optimal Control*. Springer, 1994.
[20] J. Tenny, M. and J. B. Rawlings. Efficient moving horizon estimation and nonlinear model predictive control. In *American Control Conference*, 2002.
[21] M. Vukov. *Embedded Model Predictive Control and Moving Horizon Estimation for Mechatronics Applications*. PhD thesis, KU Leuven, April 2015.
[22] M. Zanon, S. Gros, and M. Diehl. Rotational Start-up of Tethered Airplanes Based on Nonlinear MPC and MHE. In *Proceedings of the European Control Conference*, 2013.
[23] V.M. Zavala and L.T. Biegler. Nonlinear programming strategies for state estimation and model predictive control. In *Nonlinear Model Predictive Control*, pages 419–432. Springer Berlin Heidelberg, 2009.