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 $N n_x$ 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

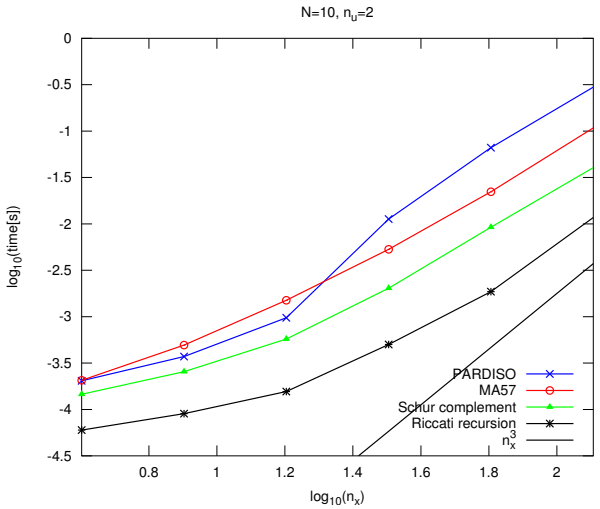

Fig. 1: Comparison of solvers PARDISO, MA57, Schur complement and Riccati recursion, for problem (1).

$\mathcal{O}(N(n_x+n_u)^3)$, 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

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

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: 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: Update the solution $(x, u, \pi) \leftarrow (x, u, \pi) + (\Delta x, \Delta u, \Delta\pi)$
7: 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

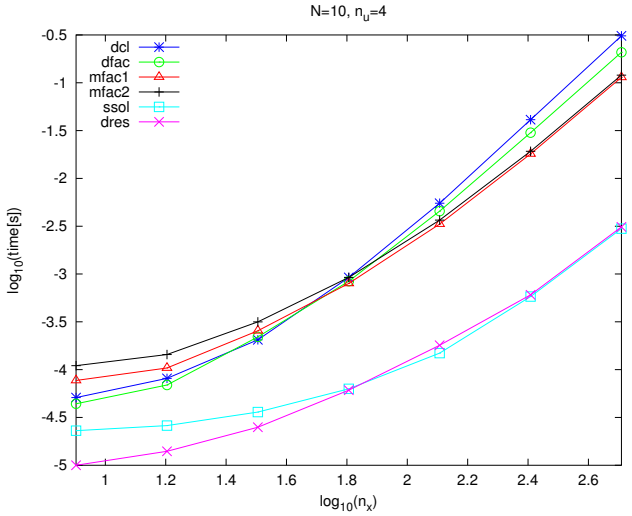

Fig. 2: Run-time. Proposed Algorithm 3 (`dfac`) is faster than classical Algorithm 1 (`dge`). Mixed precision versions `mfac1` and `mfac2` are faster than double precision version `dfac` for large n_x.

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/*