UNIVERSITÀ DEGLI STUDI DI PADOVA
FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE

TESI DI LAUREA

NUMERICAL METHODS FOR MODEL PREDICTIVE CONTROL

Laureando: **GIANLUCA FRISON**

Matricola: **620826**

Relatore: Prof. **MATTEO FISCHETTI**

ANNO ACCADEMICO 2011-2012

Ringraziamenti

Vorrei ringraziare il mio relatore, Matteo Fischetti, per aver accettato di seguirmi, e per essere stato sempre estremamente disponibile nell'assecondare le mie limitate disponibilità di orari.

Vorrei anche ringraziare mio fratello Gabriele, per l'aiuto nella revisione linguistica, e la mia ragazza Marie e tutta la mia famiglia, per il loro supporto e incoraggiamento.

Summary

This thesis mainly deals with the extended linear quadratic control problem 2.1, that is a special case of equality constrained quadratic program. 2.1 is a very general problem formulation, and it is useful for itself, since a number of other problems in optimal control and estimation of linear systems can be reduced to this form. Furthermore, it arises as sub-problem in sequential quadratic programs methods and interior-point methods for the solution of optimal control and estimation in case of non-linear systems and in presence of inequality constraints.

This thesis can be divided into two parts. In the first part, we present and analyze a number of methods for the solution of problem 2.1. These methods have been implemented in efficient C code and compared each other.

In the second part, we define problem 8.1, that is an extension of problem 2.1 and takes into account also inequality constraints. Two interior-point methods for the solution of problem 8.1 are presented and analyzed. Both methods have been implemented in C code and compared each other.

The focus is on the first part: the main goal of this thesis is the efficient implementation and comparison of different methods for solution of 2.1.

Contents

Notation

In this thesis we use the following notation:

- matrices are written with capital letters, e.g. A; vectors and scalars are written in small letters, e.g. x, α.
- the transpose of A is indicated as A'.
- the gradient of $\phi(x,u)$ with respect to the vector u is indicated as $\nabla_u \phi$.
- the minimizer of the cost function $\phi(x)$ is indicated as x^*.
- the integer n is used as index for loops over the control horizon, $n \in \{0, 1, \ldots, N\}$; the integer k is used as index for iterations of interior-point methods. The same notation is used for the relative subscripts.

In the algorithms, we use the following notation:

- with $\{Q_n\}$ we mean the set of matrices $\{Q_0, Q_1, \ldots, Q_N\}$; similarly for vectors.
- the notation $\{x_n\}_k$ indicates the set of vectors $\{x_0, x_1, \ldots x_N\}$ at the k-th iteration of an interior-point method; in order to keep the notation easier, sometimes we use the notation x_k instead.
- the notation $x_{(0:n-1)}$ indicates the sub-vector of the vector x with indexes between 0 and $n-1$ both inclusive.
- the notation $A_{(n,0:n-1)}$ indicates the elements on the n-th row of the matrix A, with column indexes between 0 and $n-1$ both inclusive.
- in the algorithms for the solution of problem 2.1, a sub-script in roman style indicates the BLAS or LAPACK routine used to perform the operation, e.g. $C \leftarrow A \cdot_{\mathrm{dgemm}} B$ means that the matrix C gets the result of the product between the matrices A and B, computed using the BLAS routine `dgemm`.

CHAPTER 1

Introduction

This thesis deals mainly with a sub-problem often arising in model predictive control, namely the extended linear quadratic control problem

$$\begin{aligned} \min_{u_n,x_{n+1}} \quad & \phi = \sum_{n=0}^{N-1} l_n(x_n,u_n) + l_N(x_N) \\ s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \end{aligned} \tag{1.1}$$

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

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

From a mathematical point of view, problem 1.1 is an equality constrained quadratic program.

Our work consists in a theoretical study of a number of well known methods for the solution of 1.1 and in the following efficient implementation in C code.

The main contribution of our work consists in the systematic presentation of existing methods, and on the comparison of their efficient implementations in terms of equally optimized C code.

The structure of this thesis is the following:

- This chapter, chapter 1, introduces the central argument and the structure of this thesis.
- Chapter 2 introduces problem 1.1 and presents necessary and sufficient conditions for its solution. It is shown how the solution of 1.1 can be found solving a linear system of equations (the KKT system).
- Chapter 3 briefly presents and compares two direct sparse solvers for the solution of the KKT system associated with problem 1.1.

- Chapters 4 to 6 present a number of methods for the solution of 1.1, one for chapter. Each chapter contains the theoretical derivation of the relative method in at least one way, the efficient implementation, and the performance analysis in terms of number of floating-point operations and use of sub-routines.
- Chapter 7 states and analyzes a test problem, and compares the methods derived in the previous chapters in terms of execution time. Two series of tests are performed: in the first one the methods are compared in the solution of problem 1.1 on its own; in the second one the methods are compared in the solution of problem 1.1 as sub-problem in an interior-point method, avoiding the re-computation of quantities that are constant among iterations.
- Chapter 8 introduces problem 8.1, that is an extension of problem 1.1 including also inequality constraints (and thus deals with an general quadratic program, with both equality and inequality constraints). Both necessary and sufficient conditions for its solution are derived. It is shown how the solution of 8.1 can be found solving a system of non-linear equations (the KKT system for a general quadratic program).
- Chapter 9 presents two interior-point methods for the solution of 8.1: the methods are derived in the case of a general quadratic program, then the implementation is tailored in the case of problem 8.1. Methods performance is compared in terms of floating-point operations per iteration. An important result in this chapter is the fact that problem 1.1 arises as sub-problem in interior-point methods for the solution of problem 8.1.
- Chapter 10 states a test problem for problem 8.1 compares the behavior of the two interior-point methods.
- Chapter 11 presents a few problems related to 1.1 and 8.1, and shows how to tailor the methods presented in previous chapters to fully exploit the special structure of these problems.
- Chapter 12 contains some brief conclusions.
- Appendix A briefly introduces the hardware and the software used to write the code and test it.
- Appendix B presents some useful algorithms performing basic linear algebra operations, and states their complexity as number of floating-point operations. The BLAS or LAPACK routines implementing the different methods are briefly described.
- Appendix C presents and tests a number of implementations of the BLAS and LAPACK libraries.
- Finally, appendix D contains tables collecting data from numerical tests.

Chapter 2

Extended Linear Quadratic Control Problem

In this chapter we present the problem we mainly deal with, namely the extended linear quadratic control problem. We also derive necessary and sufficient conditions for its solution, and show that the solution can be found solving a system of linear equations (KKT system).

2.1 Definition

The extended linear quadratic control problem can be used to represent a number of problems in optimal control and estimation of linear systems. It is also a subproblem in sequential quadratic programming methods and interior-point methods for non-linear and constrained optimal control and estimation: examples can be found in [JGND12].

The extended linear quadratic control problem is defined as

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

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

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

$$\begin{aligned} l_n(x_n, u_n) =& \frac{1}{2} \begin{bmatrix} x'_n & u'_n \end{bmatrix} \begin{bmatrix} Q_n & S'_n \\ S_n & R_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \begin{bmatrix} q'_n & s'_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \rho_n \\ l_N(x_N) =& \frac{1}{2} x'_N Q_N x_N + q'_N x_N + \rho_N \end{aligned}$$

In the above problem definition, the dynamic system model is linear (or better affine) time variant, and that the cost function is quadratic, with weights matrices time variant as well.

Problem 2.1 is an extension of the standard linear quadratic control problem, defined as

Problem 2. *The standard linear quadratic control problem is the equality constrained quadratic program*

$$\begin{aligned} \min_{u_n, x_{n+1}} \quad & \phi = \frac{1}{2} \sum_{k=0}^{N-1} \begin{bmatrix} x_n' & u_n' \end{bmatrix} \begin{bmatrix} Q & S' \\ S & R \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \frac{1}{2} x_N' Q_N x_N \\ s.t. \quad & x_{n+1} = A x_n + B u_n \end{aligned} \tag{2.2}$$

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

In this problem definition the dynamic system model is strictly linear and time invariant, and the cost function is strictly quadratic and time invariant.

This work focuses on the extended linear quadratic control problem 2.1, and deals with the most efficient methods to solve it. Problem 2.2 is a special case of problem 2.1, and thus can be solved using the methods derived for the latter. It is also possible to tailor the algorithms in order to exploit the simpler structure and obtain better performances.

2.1.1 Matrix form

From a mathematical point of view, the extended linear quadratic control problem is an equality constrained quadratic program. It can be rewritten in a more compact form as

$$\begin{aligned} \min_{x} \quad & \phi = \frac{1}{2} x' H x + g' x \\ s.t. \quad & A x = b \end{aligned} \tag{2.3}$$

where the state vector is

$$x = \begin{bmatrix} u_0 \\ x_1 \\ u_1 \\ \vdots \\ x_{N-1} \\ u_{N-1} \\ x_N \end{bmatrix}$$

and the matrices relative to the cost function and the constraints are

$$H = \begin{bmatrix} R_0 & & & & & \\ & Q_1 & S_1' & & & & \\ & S_1 & R_1 & & & & \\ & & & \ddots & & & \\ & & & & Q_{N-1} & S_{N-1}' & \\ & & & & S_{N-1} & R_{N-1} & \\ & & & & & & Q_N \end{bmatrix}, \quad g = \begin{bmatrix} S_0 x_0 + s_0 \\ q_1 \\ s_1 \\ \vdots \\ q_{N-1} \\ s_{N-1} \\ q_N \end{bmatrix}$$

$$A = \begin{bmatrix} -B_0 & I & & & & & \\ & -A_1 & -B_1 & I & & & \\ & & & \ddots & \ddots & & \\ & & & & -A_{N-1} & -B_{N-1} & I \end{bmatrix}, \quad b = \begin{bmatrix} A_0 x_0 + b_0 \\ b_1 \\ \vdots \\ b_{N-1} \end{bmatrix}.$$

The matrices H and A are large, sparse and highly structured: this structure can be exploited to obtain efficient algorithms.

Note In 2.3, the cost function does not contain a constant term, even if 2.1 does. The constant term does not influence the position of the minimum but only its value, and we are interested only in the first: then we prefer to drop the constant term, to keep the notation easier.

2.2 Optimality conditions

In the first part of this section we present necessary and sufficient conditions for the solution of a general equality constrained quadratic program; later we derive conditions for the specific case of the extended linear quadratic control problem 2.1, exploiting the special form of this problem.

2.2.1 KKT conditions

Here we present first order necessary optimality condition, known as Karush-Kuhn-Tucker (briefly KKT) conditions, for x^* to be a solution of a general equality constrained quadratic program. For the general theory, see [NW06]

The Lagrangian function associated to a general equality constrained quadratic program in the form 2.3 takes the form

$$\mathcal{L} = \mathcal{L}(x, \pi) = \phi - \pi'(Ax - b) = \frac{1}{2}x'Hx + g'x - \pi'(Ax - b).$$

In the special case of problem 2.1 it takes the form

$$\begin{aligned} \mathcal{L}(x, \pi) =& \frac{1}{2}x'Hx + g'x - \pi'(Ax - b) = \\ =& \sum_{n=0}^{N-1} l_n(x_n, u_n) + l_N(x_N) - \sum_{n=0}^{N-1} \pi'_{n+1}(x_{n+1} - A_n x_n - B_n u_n - b_n) \end{aligned}$$

where

$$\pi = \begin{bmatrix} \pi_1 \\ \pi_2 \\ \vdots \\ \pi_N \end{bmatrix}$$

is the vector of the the Lagrangian multipliers associated to the N constraints $Ax - b = 0$.

The first order necessary conditions for optimality (known as KKT conditions) can be found setting to zero the gradients of the Lagrangian function with respect to the vectors x and π,

$$\nabla_x \phi(x, \pi) = Hx + g - A'\pi = 0 \tag{2.4a}$$

$$\nabla_\pi \phi(x, \pi) = Ax - b = 0. \tag{2.4b}$$

The conditions are stated in the proposition:

Proposition 1 (First order necessary conditions)**.** *If x^* is a solution of the equality constrained quadratic program, then there exists a vector π^* such that*

$$Hx^* + g - A'\pi^* = 0 \qquad \textit{and} \qquad Ax^* - b = 0.$$

Proof. We immediately notice that condition 2.4b is the equality constraints equation: if x^* does not satisfy this condition, it can not be a solution since it does not satisfy the constraints.

Thus, let x^* be a solution: x^* is a point satisfying the constraints $Ax^*+b=0$ and is a global minimizer for the cost function in the feasible region defined by the constraints, i.e. for each x satisfying the constraints $Ax+b=0$ we have $\phi(x^*) \leq \phi(x)$.

We notice that, if the point x^* is on the constraints, given any other point x on the constraint we have that the step $\Delta x = x - x^*$ satisfies $A\Delta x = Ax - Ax^* = b - b = 0$, and thus a general vector along the constraint is in the kernel (or null space) of the matrix A, and thus it is in the image of the null space matrix Z. The matrix Z is defined as the matrix with full column rank whose columns generate the null space of A; this implies $AZ = 0$.

The geometric interpretation of condition 2.4a is that the gradient of the cost function in x^* is orthogonal to the constraints. In fact the first condition can be rewritten as $Hx^* + g = A'\pi^*$, that means that the vector $Hx^* + g$ is in the image of the matrix A', $Hx^* + g \in Im(A')$. The relation[1] $Ker(A) = Im(A')^\perp$ implies that $(Hx^* + g) \perp Ker(A)$.

Now we want to show that, if condition 2.4a is false, then it is always possible to find a step Δx along the constraint and such that $\phi(x^* + \Delta x) < \phi(x^*)$: in particular, the step in the direction defined by the projection on the vector space generated by the matrix Z of the opposite of the gradient of the cost function computed in x^* makes the job.

Let us suppose that condition 2.4a is false, this means $(Hx^*+g) \not\perp Ker(A)$, and then there exist a vector $z \in Ker(A)$ such that $(Hx^* + g)'z \neq 0$. This implies that the projection of the gradient on the null space of the matrix A is not zero, $(Hx^* + g)'Z \neq 0$, and then also $y = -(Hx^* + g)'Z \neq 0$. The vector y is a vector in the null space of A: its expression in the larger space is Zy; this is not the zero vector since $y \neq 0$ and Z ha full column rank.

As step we choose the scaled vector

$$\Delta x = (Zy)\lambda = ZZ'(Hx^* + g)(-\lambda)$$

with the scalar $\lambda > 0$. The point $x = x^* + \Delta x$ satisfies the constraint, since $Ax = Ax^* + AZy\lambda = b + 0 = b$, and the value of the cost function in x is

$$\begin{aligned}\phi(x) =& \phi(x^* + \Delta x) = \frac{1}{2}(x^*)'Hx^* + g'x^* + \frac{1}{2}\Delta x'H\Delta x + (Hx^* + g)'\Delta x = \\ =& \phi(x^*) + \frac{1}{2}\lambda y'Z'HZy\lambda - (Hx^* + g)'ZZ'(Hx^* + g)\lambda = \\ =& \phi(x^*) + \frac{1}{2}\alpha\lambda^2 - \beta\lambda.\end{aligned}$$

[1]This relation can be proved in this way: if we use the notation $< v_i >$ to denote the vector space generated by the vectors v_i, we have that $Im(A') =< col_i(A') >$, where $col_i(A')$ is the i-th column of the matrix A'. We can now prove that $Ker(A) = \{x|Ax = 0\} = \{x|x \perp row_i(A)\} = \{x|x \perp col_i(A')\} =< col_i(A') >^\perp = Im(A')^\perp$.

where in general $\alpha \in \mathbb{R}$ and $\beta > 0$.

If $\alpha \leq 0$, we immediately have $\phi(x) \leq \phi(x^*) - \beta\lambda < \phi(x^*)$, and thus x^* is not a minimizer, and then neither a solution.

If $\alpha > 0$, we have that $\frac{1}{2}\alpha\lambda^2 - \beta\lambda < 0$ for $\lambda < \frac{2\beta}{\alpha}$: then for $\lambda < \frac{2\beta}{\alpha}$ we have again $\phi(x) < \phi(x^*)$, and thus x^* is not a minimizer, and then neither a solution.

This implies that condition 2.4a is a necessary condition for a point x^* to be a solution. □

Conditions 2.4 can be rewritten in matrix form as the system of linear equations

$$\begin{bmatrix} H & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} x \\ \pi \end{bmatrix} = \begin{bmatrix} -g \\ -b \end{bmatrix}, \tag{2.5}$$

that in the case of problem 2.1 takes the form (for $N = 3$)

$$\begin{bmatrix} R_0 & & & & & & B_0' & & \\ & Q_1 & S_1' & & & & -I & A_1' & \\ & S_1 & R_1 & & & & & B_1' & \\ & & & Q_2 & S_2' & & & -I & A_2' \\ & & & S_2 & R_2 & & & & B_2' \\ & & & & & Q_3 & & & -I \\ B_0 & -I & & & & & & & \\ & A_1 & B_1 & -I & & & & & \\ & & & A_2 & B_2 & -I & & & \end{bmatrix} \begin{bmatrix} u_0 \\ x_1 \\ u_1 \\ x_2 \\ u_2 \\ x_3 \\ \pi_1 \\ \pi_2 \\ \pi_3 \end{bmatrix} = \begin{bmatrix} -S_0x_0 - s_0 \\ -q_1 \\ -s_1 \\ -q_2 \\ -s_2 \\ -q_3 \\ -A_0x_0 - b_0 \\ -b_1 \\ -b_2 \end{bmatrix}$$

This system is known as KKT system, and its solutions are the points where the Lagrangian function is stationary: the solutions of problem 2.1 have to be searched among them.

The KKT conditions 2.4 are in general only necessary; they become also sufficient in case of convex cost function. In particular, in the case of equality constrained quadratic program, we have that a point x^* satisfying the KKT conditions is a solution (i.e. a global minimizer) if the matrix H is positive semi-definite, and it is the unique solution if the matrix H is positive definite.

Proposition 2 (First order sufficient conditions)**.** *If the point x^* satisfies the KKT conditions with the Lagrangian multiplier π^* and H is positive semi-definite, then x^* is a global minimizer for the cost function ϕ, i.e. a solution for the equality constrained quadratic program.*

Proof. The vectors x^* and π^* satisfy the first KKT condition, and so $Hx^* + g = A'\pi^*$. Let x be any point satisfying the constraints (i.e. $Ax + b = 0$), and let $\Delta x = x - x^*$, then $A\Delta x = A(x - x^*) = b - b = 0$.

The value of the cost function at the point x is thus

$$\begin{aligned} \phi(x) = \frac{1}{2}x'Hx + g'x = \frac{1}{2}(x^* + \Delta x)'H(x^* + \Delta x) + g'(x^* + \Delta x) = \\ = \left(\frac{1}{2}(x^*)'Hx^* + g'x^*\right) + \Delta x'(Hx^* + g) + \frac{1}{2}\Delta x'H\Delta x = \\ = \phi(x^*) + (A\Delta x)'\pi^* + \frac{1}{2}\Delta x'H\Delta x = \phi(x^*) + \frac{1}{2}\Delta x'H\Delta x \geq \phi(x^*) \end{aligned}$$

since the matrix H is positive semi-definite. Thus x^* is a solution for the equality constrained quadratic program. □

Proposition 3 (First order sufficient conditions for uniqueness). *If the point x^* satisfies the KKT conditions with the Lagrangian multiplier π^* and H is positive definite, then x^* is the unique global minimizer for the cost function ϕ, i.e. the unique solution for the equality constrained quadratic program.*

Proof. We already know from the previous proposition that $\phi(x^*) \leq \phi(x)$ for all x satisfying the constraints. If there exist an x such that the previous relation is an equation, we have

$$\phi(x) = \phi(x^*) + \frac{1}{2}\Delta x' H \Delta x,$$

that means $\frac{1}{2}\Delta x' H \Delta x = 0$, and since H is positive definite, we necessarily have $\Delta x = x - x^* = 0$ and thus $x = x^*$. Thus x^* is the unique solution for the equality constrained quadratic program. □

In the case of problem 2.1, since the matrix H is block diagonal, it is positive (semi)-definite if and only if each block is positive (semi)-definite.

A sufficient condition for problem 2.1 to have an unique solution is that the matrix H is positive definite; anyway, this is not necessary, since it holds the weaker result

Proposition 4 (Sufficient conditions for uniqueness of the solution of 2.5). *Let A have full row rank and Z be the matrix whose columns are a base for the kernel of the matrix A; if the reduced Hessian matrix $Z'HZ$ is positive definite, then the KKT matrix is non-singular, and then the KKT system 2.5 has an unique solution.*

Proof. By definition of Z, we have that $AZ = 0$ and Z has full column rank. We have to show that the KKT system is non-singular: thus we have to show that the vectors u and v in the expression

$$\begin{bmatrix} H & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = 0$$

are both zero.

The second equation gives $-Au = 0$, and thus u is in the null space of A, and thus there exists the vector $\tilde{u}$ such that $u = Z\tilde{u}$.

The first equation gives $Hu - A'v = 0$; if we multiply both side for u', we have

$$u'Hu - u'A'v = u'Hu - (Au)'v = u'Hu = 0 \qquad (2.6)$$

since from the second equation we know that $Au = 0$. Inserting the expression $u = Z\tilde{u}$ into 2.6, we have

$$u'Hu = \tilde{u}'Z'HZ\tilde{u} = 0$$

and since by hypothesis $Z'HZ$ is positive definite, we have that $u = 0$. Inserting this value in the first equation gives $-A'v = 0$, that implies $v = 0$ since A has full row rank, and thus A' has full column rank. Thus the vector

$$\begin{bmatrix} u \\ v \end{bmatrix} = 0$$

and thus the KKT matrix is non-singular. □

Proposition 5 (Second order sufficient conditions)**.** *Proposition 4, together with proposition 2, expresses the second order sufficient conditions for optimality.*

Proof. The proof is an immediate consequence of propositions 4 and 2.

It is also possible to write a direct proof. Let x^* be a point satisfying the KKT conditions together with the Lagrangian multiplier π^*, and let x be any other point satisfying the constraints. Defined the step $\Delta x = x - x^* \neq 0$, we have $A\Delta x = A(x - x^*) = b - b = 0$, and then Δx is in the null space of A, and can be written as $\Delta x = Zy$ for some vector $y \neq 0$. The value of the cost function in x is

$$\begin{aligned}\phi(x) =& \frac{1}{2}x'Hx + g'x = \phi(x^* + \Delta x) = \\ =& \left(\frac{1}{2}(x^*)'Hx^* + g'x^*\right) + (Hx^* + g)'\Delta x + \frac{1}{2}\Delta x' H \Delta x = \\ =& \phi(x^*) + (Hx^* + g)Zy + \frac{1}{2}y'Z'HZy = \phi(x^*) + \frac{1}{2}y'Z'HZy\end{aligned}$$

and, since $Z'HZ$ is positive definite by hypothesis, $\phi(x) > \phi(x^*)$: x^* is the unique global minimizer. □

We can now study the specific case of problem 2.1; we have that

Proposition 6 (Sufficient conditions for the existence and uniqueness of the solution of problem 2.1)**.** *Let the matrices*

$$\begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix}$$

and the matrix Q_N be positive semi-definite, and the matrices R_n be positive definite for all $n \in \{0, \dots, N-1\}$, then problem 2.1 has one and only one solution.

Proof. We are proving the proposition only in the case $N = 3$, since there are not conceptual difficulties to extend the proof to the general case, while there are notation problems.

The first step consist in finding the Z matrix; we notice that the matrix A has size $Nn_x \times N(n_x + n_u)$, and thus we are looking for a matrix of size $N(n_x + n_u) \times Nn_u$ with full column rank Nn_u and whose columns are in the kernel space of the matrix A. Since we have that

$$AZ = \begin{bmatrix} -B_0 & I & 0 & 0 & 0 & 0 \\ 0 & -A_1 & -B_1 & I & 0 & 0 \\ 0 & 0 & 0 & -A_2 & -B_1 & I \end{bmatrix} \begin{bmatrix} I & 0 & 0 \\ B_0 & 0 & 0 \\ 0 & I & 0 \\ A_1B_0 & B_1 & 0 \\ 0 & 0 & I \\ A_2A_1B_0 & A_2B_1 & B_2 \end{bmatrix} = 0$$

the above Z matrix does the work.

The $Nn_u \times n_u$ matrix $Z'HZ$ takes the form

$$Z'HZ = \left[\begin{array}{cc} R_0 + B_0'Q_1B_0 + B_0'A_1'Q_2A_1B_0 + B_0'A_1'A_2'Q_3A_2A_1B_0 & \ldots \\ S_1B_0 + B_1'Q_2A_1B_0 + B_1'A_2'Q_3A_2A_1B_0 & \ldots \\ S_2A_1B_0 + B_2'Q_3A_2A_1B_0 & \ldots \end{array}\right.$$

$$\left.\begin{array}{cc} B_0'S_1' + B_0'A_1'Q_2B_1 + B_0'A_1'A_2'Q_3A_2B_1 & B_0'A_1'S_2' + B_0'A_1'A_2'Q_3B_2 \\ R_1 + B_1'Q_2B_1 + B_1'A_2'Q_3A_2B_1 & B_1'S_2' + B_1'A_2'Q_3B_2 \\ S_2B_1 + B_2'Q_3A_2B_1 & R_2 + B_2'Q_3B_2 \end{array}\right]$$

and, since the H matrix is symmetric positive semi-definite, it is symmetric positive semi-definite as well. In order to prove that it is positive definite we have to show that, if $x'Z'HZx = 0$, the necessarily the vector $x = 0$.

We notice that we can write the matrix $Z'HZ$ as the sum of four symmetric positive semi-definite matrices each containing only matrices Q_n, S_n and R_n with the same index n:

$$\begin{aligned} Z'HZ &= \begin{bmatrix} R_0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} B_0'Q_1B_0 & B_0'S_1' & 0 \\ S_1B_0 & R_1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \\ &+ \begin{bmatrix} B_0'A_1'Q_2A_1B_0 & B_0'A_1'Q_2B_1 & B_0'A_1'S_2' \\ B_1'Q_2A_1B_0 & B_1'Q_2B_1 & B_1'S_1' \\ S_2A_1B_0 & S_2B_1 & R_2 \end{bmatrix} + \\ &+ \begin{bmatrix} B_0'A_1'A_2'Q_3A_2A_1B_0 & B_0'A_1'A_2'Q_3A_2B_1 & B_0'A_1'A_2'Q_3B_2 \\ B_1'A_2'Q_3A_2A_1B_0 & B_1'A_2'Q_3A_2B_1 & B_1'A_2'Q_3B_2 \\ B_2'Q_3A_2A_1B_0 & B_2'Q_3A_2B_1 & B_2'Q_3B_2 \end{bmatrix} = \\ &= \begin{bmatrix} I \\ 0 \\ 0 \end{bmatrix} R_0 \begin{bmatrix} I & 0 & 0 \end{bmatrix} + \begin{bmatrix} B_0' & 0 \\ 0 & I \\ 0 & 0 \end{bmatrix} \begin{bmatrix} Q_1 & S_1' \\ S_1 & R_2 \end{bmatrix} \begin{bmatrix} B_0 & 0 & 0 \\ 0 & I & 0 \end{bmatrix} + \\ &+ \begin{bmatrix} B_0'A_1' & 0 \\ B_1' & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} Q_2 & S_2' \\ S_2 & R_2 \end{bmatrix} \begin{bmatrix} A_1B_0 & B_1 & 0 \\ 0 & 0 & I \end{bmatrix} + \\ &+ \begin{bmatrix} B_0'A_1'A_2' \\ B_1'A_2' \\ B_2' \end{bmatrix} Q_3 \begin{bmatrix} A_2A_1B_0 & A_2B_1 & B_2 \end{bmatrix}. \end{aligned}$$

Let the vector x be

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}.$$

The scalar $x'Z'HZx = 0$ can be written as the sum of four terms bigger or equal to zero: this implies that, in order to have 0 as sum, they have to be all 0.

The first one is

$$\begin{bmatrix} x_0' & x_1' & x_2' \end{bmatrix} \begin{bmatrix} I \\ 0 \\ 0 \end{bmatrix} R_0 \begin{bmatrix} I & 0 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = x_0'R_0x_0 = 0$$

that implies $x_0 = 0$, since the matrix R_0 is positive definite.

The second one is

$$\begin{bmatrix} 0 & x_1' & x_2' \end{bmatrix} \begin{bmatrix} B_0' & 0 \\ 0 & I \\ 0 & 0 \end{bmatrix} \begin{bmatrix} Q_1 & S_1' \\ S_1 & R_2 \end{bmatrix} \begin{bmatrix} B_0 & 0 & 0 \\ 0 & I & 0 \end{bmatrix} \begin{bmatrix} 0 \\ x_1 \\ x_2 \end{bmatrix} = x_1' R_1 x_1 = 0$$

that implies $x_1 = 0$, since the matrix R_1 is positive definite.

The third one is

$$\begin{bmatrix} 0 & 0 & x_2' \end{bmatrix} \begin{bmatrix} B_0' A_1' & 0 \\ B_1' & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} Q_2 & S_2' \\ S_2 & R_2 \end{bmatrix} \begin{bmatrix} A_1 B_0 & B_1 & 0 \\ 0 & 0 & I \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ x_2 \end{bmatrix} = x_2' R_2 x_2 = 0$$

that implies $x_2 = 0$, since the matrix R_2 is positive definite.

In this way the whole vector $x = 0$, and so the matrix $Z'HZ$ is positive definite, and by proposition 5 this means that problem 2.1 has one and only one solution. □

Note (Symmetry) Even if it is not necessary for the proof of the previous proposition, it is usually requested to the matrices R_0, $\begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix}$ and Q_N to be also symmetric: this would lead to more efficient algorithms; on the other hand, this is not a limitation, since it is always possible to rewrite a positive semi-definite matrix in an equivalent symmetric form.

In what follows we assume that the above matrices are all symmetric.

2.2.2 Band diagonal representation of the KKT system

The KKT system of problem 2.1 can be written as

$$\begin{bmatrix} R_0 & B_0' & & & & & & & & \\ B_0 & & -I & & & & & & & \\ & -I & Q_1 & S_1' & A_1' & & & & & \\ & & S_1 & R_1 & B_1' & & & & & \\ & & A_1 & B_1 & & -I & & & & \\ & & & & -I & Q_2 & S_2' & A_2' & & \\ & & & & & S_2 & R_2 & B_2' & & \\ & & & & & A_2 & B_2 & & -I & \\ & & & & & & & -I & Q_3 \end{bmatrix} \begin{bmatrix} u_0 \\ \pi_1 \\ x_1 \\ u_1 \\ \pi_2 \\ x_2 \\ u_2 \\ \pi_3 \\ x_3 \end{bmatrix} = \begin{bmatrix} -S_0 x_0 - s_0 \\ -A_0 x_0 - b_0 \\ -q_1 \\ -s_1 \\ -b_1 \\ -q_2 \\ -s_2 \\ -b_2 \\ -q_3 \end{bmatrix} \tag{2.7}$$

The advantage of this form is that the KKT system is represented in band diagonal form, with band width depending on n_x and n_u but independent of N: this implies that the system can be solved in time $\mathcal{O}(N(n_x + n_u)^3)$ (where n_x is the length of the x vector and n_u is the length of the u vector), using a band diagonal generic solver. As a comparison, the solution of 2.7 by means of the use of a dense LDL' factorization requires roughly $\frac{1}{3}(N(2n_x + n_u))^3$ floating-point operations.

This representation will also be useful in chapter 5 about the Riccati, since it is possible to derive the Riccati recursion method as a block solution strategy for this system.

Chapter 3

Direct Sparse Solvers

In this chapter we directly solve the KKT system 2.7 using direct sparse solvers. The system is large and sparse, and can be written in a band diagonal form, with a bandwidth depending only on n_x and n_u and not on N: the solvers are able to exploit this structure, and find the solution in a time of the order of $N(n_x + n_u)^3$.

We investigate the solvers MA57 and PARDISO, two of the best solvers for the solution of sparse symmetric linear systems. We present the two solvers in brief; the reader can find more information consulting the relative web-sites or literature, for example [Duf04] and [SG11].

3.1 MA57

MA57 is a direct solver for symmetric sparse systems of linear equations; it is part of HSL[1] software.

It is designed to solve the linear system of equations

$$AX = B$$

where the $n \times n$ matrix A is large sparse and symmetric, and not necessarily definite, and the right-hand side B may have more that one column. The algorithm performs an LDL' factorization of the matrix A, implemented using a multi-frontal approach.

The solver uses the BLAS library to perform matrix-vector and matrix-matrix operations: this means it can exploit a BLAS implementation optimized for the specific architecture. It also uses (optionally) the MeTiS library, performing partitioning of graphs.

The code consists in a number of routines; the one of interest for us are:

[1]HSL, a collection of Fortran codes for large-scale scientific computation. See http://www.hsl.rl.ac.uk/

i	IRN(i)	JCN(i)	A(i)
1	1	1	2.0
2	2	1	3.0
3	3	2	4.0
4	3	3	1.0
5	4	3	5.0
6	5	2	6.0
7	5	5	1.0

Table 3.1: *A* matrix 3.1 stored in sparse format.

- `MA57I/ID` initializes with default values the arrays holding the control parameters.
- `MA57A/AD` analyzes the structure of the matrix A and chooses the pivot sequence; this routine makes use of MeTiS, if available.
- `MA57B/BD` factorizes the matrix A using the information from the previous call to the routine `MA57A/AD`.
- `MA57C/CD` solves the system $Ax = b$ (or $AX = B$ in case of multiple right hand sides) using the factorization computed by the previous call to the routine `MA57B/BD`.
- `MA57D/DD` solves the system $Ax = b$ using iterative refinement and the factorization computed by the previous call to the routine `MA57B/BD`.

The routine named MA57X is the single-precision version, while the routine named MA57XD is the double-precision version. In our tests we use only the double-precision version.

The matrix A has to be stored in sparse format, using two vectors of integers to store the row and column index of each entry, and a vector of reals to store the value of each entry. Since the matrix A is symmetric, only one of the entries a_{ij} and a_{ji} needs to be stored; eventual zero entries on the diagonal do not need to be stored. For example, the matrix

$$A = \begin{bmatrix} 2 & 3 & & & \\ 3 & 0 & 4 & & 6 \\ & 4 & 1 & 5 & \\ & & 5 & 0 & \\ & 6 & & & 1 \end{bmatrix} \tag{3.1}$$

may be stored as in table 3.1, where IRN is the vector of the row indexes JCN is the vector of the column indexes. We notice that the row and column indexes use the FORTRAN notation, starting from 1. The test problem is the mass-spring problem 7.2, with $n_x = 128$, $n_u = 8$ and $N = 50$.

3.2 PARDISO

PARDISO[2] is a software package for the solution of large sparse system of linear equations, both symmetric and non-symmetric. It is designed to solve a set of

[2]see also the reference web-site `http://www.pardiso-project.org`.

i	IA(i)	JA(i)	A(i)
1	1	1	2.0
2	3	2	3.0
3	6	2	0.0
4	8	3	4.0
5	9	5	6.0
6	10	3	1.0
7		4	5.0
8		4	0.0
9		5	1.0

Table 3.2: A matrix 3.1 stored in compressed sparse row format.

large sparse linear systems of equations

$$AX = B$$

where the $n \times n$ matrix A is a large sparse squared matrix, that may be symmetric definite or indefinite, or non-symmetric. The package contains direct sparse solvers and multi-recursive iterative sparse solvers, performing LU, LDL' or LL' factorizations of the A matrix.

In this work we test the software version 4.1.2 for Linux and X86 architecture; we only make use of the direct sparse solver for the solution of large sparse symmetric non-definite systems of linear equations.

The solvers use the BLAS and LAPACK libraries to perform basic linear algebra operations: thus they can exploit libraries optimized for the specific architecture, if available.

The solution process is divided into three phases, and it is possible to perform them in any of the combinations 11, 12, 13, 22, 23, 33, where the first digit indicates the starting phase and the second digit indicates the ending phase. The three phases are:

1. Analysis.

2. Numerical factorization.

3. Solution and iterative refinement.

There are also an initialization function, and an additional phase, called -1, used to release all the internal memory for the matrices.

The matrix A has to be stored in compressed sparse row format: the entries are stored one row at a time starting from the first one, and inside each row the entries are stored in increasing order of column. A real vector A contains the entries values, an integer vector JA contains the column indexes, and an integer vector IA (of size $n+1$) is such that $IA(i)$ is the index in the A and JA vectors of the first element of the row i, and $IA(n+1)$ is the index of the first element outside the table (and thus the number of elements in the A and JA vectors is $IA(n+1)-1$). Furthermore all the diagonal entries have to be stored explicitly. For example, the A matrix 3.1 has to be stored as in the table 3.2. We notice that the row and column indexes use the FORTRAN notation, starting from 1.

	MA57		PARDISO	
phase	time (s)	%	time (s)	%
create A and B	0.033544	4.7	0.026609	0.8
initialization	0.000012	0.0	0.001402	0.0
analysis	0.133541	18.7	2.081128	65.8
factorization	0.534067	74.6	0.975534	30.9
solution	0.014367	2.0	0.076888	2.4
total	0.715531	100.0	3.161561	99.9

Table 3.3: Comparison between the computational time needed by the different components of the solvers MA57 and PARDISO. In the table, 'time' means wall clock time, measured in seconds.

3.3 Performance analysis

In this section we analyze the performance of the two direct sparse solvers in the solution of the KKT system 2.7 associated with the extended linear quadratic control problem 2.1.

3.3.1 Cost of the different parts of the solvers

Here we compare the cost in term of computational time of the different parts of the two solvers.

We solve an instance of the mass-spring test problem 7.2 with $n_x = 128$, $n_u = 8$, $N = 50$. The results are in table 3.3.

There is some substantial difference between the solvers. First of all, the MA57 solver is more than 4 times faster than the PARDISO solver. Furthermore, the analysis phase in the PARDISO solver takes the best part of the computational time, while in the case of the MA57 solver, it is the factorization phase that takes the best part of the computational time.

3.3.2 Comparative test

In a model predictive control framework, a new input sequence has to be computed at each sampling time, solving a problem that has the same structure. This means that the initialization and the analysis phases can be computed just once, off-line.

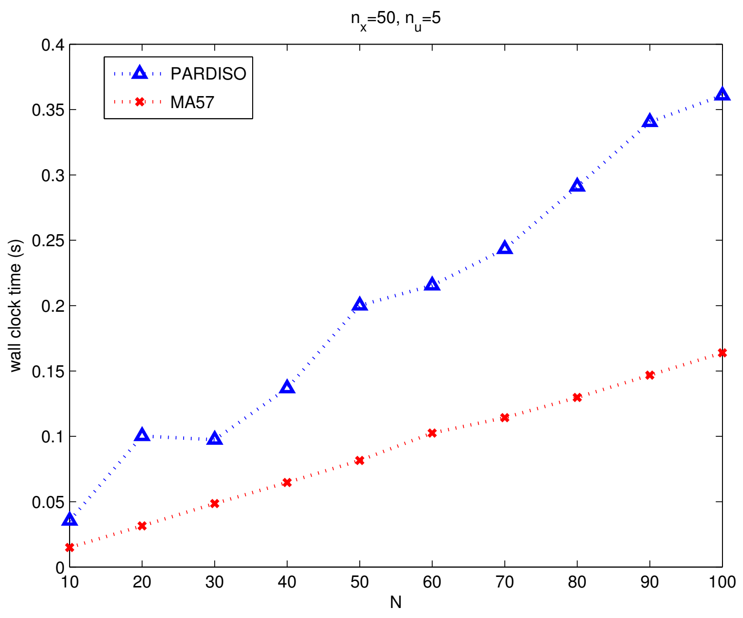
In the pictures 3.1a and 3.1b there are plots of the computational time needed by the two solvers, just considering the creation of A and B, the factorization and the solution phases (not the initialization and the analysis phases). The test problem is again the mass-spring problem 7.2. The tables with the numerical data are D.1 and D.2, in the appendix D.

In figure 3.1a n_x and n_u are fixed, and just N is varied. Apart from the numerical oscillations (larger in the case of PARDISO than in MA57 even averaging on the same number of tests), both solvers appear linear in N, even if MA57 is faster.

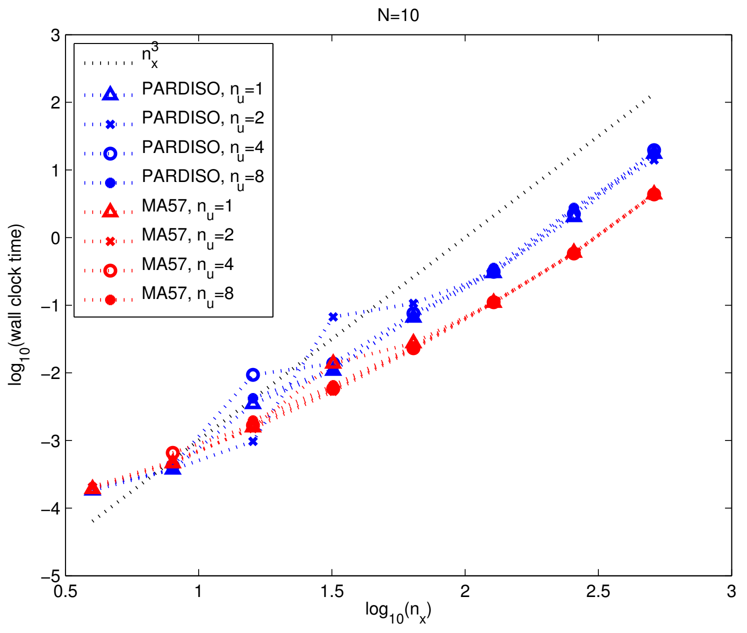
In figure 3.1b N is fixed, while n_x and n_u are varied. Our test problem requires $n_u \leq n_x/2$, and often in the tests it holds $n_u \ll n_x$: the computational time then is just slightly affected by the variation of n_u, as long as $n_u \ll n_x$.

On the other hand, the computational time appears to be cubic for large values of n_x, and slower for small values. The PARDISO solver seems to be slightly faster for small values of n_x, while it is quite slower (up to 4 times) for large values of n_x.

In conclusion, the MA57 solver shows best performances over the PARDISO solver in the solution of the extended linear quadratic control problem 2.1. Anyway, both methods are not competitive with respect to the tailored methods presented in following chapters.



(a) $n_x = 50$ and $n_u = 5$ fixed, N varied.



(b) $N = 10$ fixed, $n_x \in \{4, 8, 16, 32, 64, 128, 256, 512\}$ and $n_u \in \{1, 2, 4, 8\}$ varied.

Figure 3.1: Comparison of the performance of the solvers PARDISO and MA57 in the solution of the problem 2.1, using as test problem the mass-spring problem 7.2.

Chapter 4

Schur Complement Method

The KKT system 2.7 may be solved even by using the Schur complement of the KKT matrix. This approach is producing matrices whose non-zero elements are all around the diagonal, packed into sub-matrices. Thus the sparsity of the KKT matrix is preserved. Furthermore, it is possible to work just on the dense sub-matrices, using the standard BLAS and LAPACK routines. The drawback of this approach is the need for the Hessian matrix H to be positive definite instead of just positive semi-definite. The method has the same asymptotic complexity as the direct sparse solvers in chapter 3, namely $N(n_x + n_u)^3$, but it is faster in practice.

4.1 Derivation

This section is divided into two parts: in the first part we derive the Schur complement method for the solution of a general equality constrained quadratic program. In the second part we specialize the method in the case of problem 2.1.

General case

The use of the Schur complement method for the solution of the general equality constrained quadratic program is discussed in [NW06].

We consider the KKT system of the general equality constrained quadratic program 2.3, that is in the form

$$\begin{bmatrix} H & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} x \\ \pi \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix}. \tag{4.1}$$

We can rewrite 4.1 in the equivalent form

$$\begin{cases} Hx - A'\pi & = -g \\ -Ax & = -b \end{cases} \tag{4.2}$$

more convenient for our purposes. It is difficult to handle the system written in this form since the vector x in the second equation is already isolated but the matrix A is not invertible in general.

The method requires the matrix H to be invertible: since it is already positive semi-definite by previous hypothesis, this means that it is positive definite. Because of the block diagonal form, the matrix H is positive definite if and only if every single block is positive definite. In particular, this requires both Q_i and R_i to be positive definite (even if this is not sufficient in general).

Since the matrix H is invertible, we can isolate the vector π by multiplying the first equation in 4.2 by AH^{-1} and summing it with the second equation in 4.2, obtaining the equivalent system

$$\begin{cases} AH^{-1}A'\pi = b + AH^{-1}g \\ Hx = A'\pi - g \end{cases} \tag{4.3}$$

The matrix $\Psi = AH^{-1}A'$ is the Schur complement of the matrix H in the KKT matrix. It is positive definite since the matrix H^{-1} is positive definite, and the matrix A has full row rank. In this way we can solve the first equation in 4.3 for π, and afterward substitute its value in the second equation in 4.3 and solve it for x.

Problem 2.1 case

The use of Schur complement method for the solution of model predictive control problems can be found also in [WB10].

In the special case of problem 2.1, the matrix $AH^{-1}A$ takes the form (for $N = 3$)

$$AH^{-1}A' = \left[\begin{matrix} \tilde{Q}_1 + B_0\tilde{R}_1B_1' & -\tilde{Q}_1A_1' - \tilde{S}_1'B_1' & \cdots \\ -A_1\tilde{Q}_1 - B_1\tilde{S}_1 & \tilde{Q}_2 + A_1\tilde{Q}_1A_1' + B_1\tilde{S}_1A_1' + A_1\tilde{S}_1'B_1 + B_1\tilde{R}_1B_1' & \cdots \\ 0 & -A_2\tilde{Q}_2 - B_2\tilde{S}_2 & \cdots \end{matrix}\right.$$
$$\left.\begin{matrix} 0 \\ -\tilde{Q}_2A_2' - \tilde{S}_2'B_2' \\ \tilde{Q}_3 + A_2\tilde{Q}_2A_2' + B_2\tilde{S}_2A_2' + A_2\tilde{S}_2'B_2 + B_2\tilde{R}_2B_2' \end{matrix}\right]$$

where

$$\begin{bmatrix} \tilde{Q}_i & \tilde{S}_i' \\ \tilde{S}_i & \tilde{R}_i \end{bmatrix} = \begin{bmatrix} Q_i & S_i' \\ S_i & R_i \end{bmatrix}^{-1}.$$

The vector $b + AH^{-1}g$ takes the form

$$b + AH^{-1}g = \begin{bmatrix} A_0x_0 + b_0 - B_0\tilde{R}_0(S_0x_0 + s_0) + \tilde{Q}_1q_1 + \tilde{S}_1's_1 \\ b_1 - (A_1\tilde{Q}_1 + B_1\tilde{S}_1)q_1 - (A_1\tilde{S}_1' + B_1\tilde{R}_1)s_1 + \tilde{Q}_2q_2 + \tilde{S}_2's_2 \\ b_2 - (A_2\tilde{Q}_2 + B_2\tilde{S}_2)q_2 - (A_2\tilde{S}_2' + B_2\tilde{R}_2)s_2 + \tilde{Q}_3q_3 \end{bmatrix}$$

and the vector $A'\pi - g$ takes the form

$$A'\pi - g = \begin{bmatrix} -B_0'\pi_1 - (S_0x_0 + s_0) \\ \pi_1 - A_1'\pi_2 - q_1 \\ -B_1'\pi_2 - s_1 \\ \pi_2 - A_2'\pi_3 - q_2 \\ -B_2'\pi_3 - s_2 \\ \pi_3 - q_3 \end{bmatrix}.$$

4.2 Implemetation

In this section we show that the matrix Ψ preserves sparsity, and its non-zero entries are grouped in dense sub-matrices displaced around the diagonal. The implementation is exploiting this structure, obtaining better performances.

To start with, the matrix H has the block diagonal structure

$$H = \begin{bmatrix} R_0 & & & & & \\ & Q_1 & S_1' & & & \\ & S_1 & R_1 & & & \\ & & & Q_2 & S_2' & \\ & & & S_2 & R_2 & \\ & & & & & Q_3 \end{bmatrix}$$

where the first block is $n_u \times n_u$, the following $N-1$ blocks are $(n_u+n_x) \times (n_u + n_x)$, and the last block is $n_x \times n_x$.

As already stated before, in this chapter H is assumed to be invertible (and thus positive definite): each single block is also invertible (and thus positive definite). Instead of explicitly compute the inverse H^{-1}, we compute the Cholesky factorization of each single block and invert the resulting triangular matrix. We prefer to use these triangular matrices since we can exploit the useful triangular form in the following computations.

Instead of storing the mainly zero H matrix, we only store the diagonal blocks. We use a single array containing one after the other the single blocks, each saved in column-major order, with the upper triangular part of each block containing the matrix elements and the strictly lower triangular part containing just zeros. In this way the number of stored floating points is $n_u^2 + (N-1)(n_u + n_x)^2 + n_x^2$, while saving all the H matrix the number of stored floating points would be $N^2(n_u + n_x)^2$.

We compute the Cholesky factorization of each block, using the LAPACK routine `dpotrf`[1], and computing the upper factor: after the call to `dpotrf`, in the upper triangular part of each block there is the upper factor of the Cholesky factorization. The overall upper factor U (such that H is thus factorized as $H = U'U$) is thus

$$U = \begin{bmatrix} U_{0,22} & & & & & \\ & U_{1,11} & U_{1,12} & & & \\ & 0 & U_{1,22} & & & \\ & & & U_{2,11} & U_{2,12} & \\ & & & 0 & U_{2,22} & \\ & & & & & U_{3,11} \end{bmatrix}$$

where $U_{i,11}$ and $U_{i,22}$ are upper triangular matrices and $U_{i,12}$ is a full matrix.

We compute the inverse of each upper factor matrix using the LAPACK routine `dtrtri`[2]. Since the inverse of an upper triangular matrix is an upper

[1]See appendix B for a description of the BLAS and LAPACK routines, and B.4 for the description of the `dpotrf` routine.

[2]See appendix B.5.

triangular matrix, the inverse U^{-1} of the U matrix is

$$U^{-1} = \begin{bmatrix} \tilde{U}_{0,22} & & & & & \\ & \tilde{U}_{1,11} & \tilde{U}_{1,12} & & & \\ & 0 & \tilde{U}_{1,22} & & & \\ & & & \tilde{U}_{2,11} & \tilde{U}_{2,12} & \\ & & & 0 & \tilde{U}_{2,22} & \\ & & & & & \tilde{U}_{3,11} \end{bmatrix}$$

where $\tilde{U}_{i,11}$ and $\tilde{U}_{i,22}$ are upper triangular matrices and $\tilde{U}_{i,12}$ is a full matrix.

The Schur complement Ψ can be rewritten as

$$\begin{aligned} \Psi &= AH^{-1}A' = A(U'U)^{-1}A' = AU^{-1}(U')^{-1}A' = AU^{-1}(U^{-1})'A' \\ &= (AU^{-1})(AU^{-1})' = \Phi\Phi' \end{aligned}$$

that is the product of the matrix Φ and its transposed Φ'.

The next step is the computation of $\Phi = AU^{-1}$. We exploit the fact that $\tilde{U}_{i,11}$ and $\tilde{U}_{i,22}$ are upper triangular matrices, using the specialized routine `dtrmm`[3], requiring approximately half of the floating-point operations compared to the general matrix multiplication routine `dgemm`[4]. The matrix $\Phi = AU^{-1}$ takes the form

$$\Phi = AU^{-1} =$$

$$= \begin{bmatrix} -B_0 & I & & & & \\ & -A_1 & -B_1 & I & & \\ & & & -A_2 & -B_2 & I \end{bmatrix} \begin{bmatrix} \tilde{U}_{0,22} & & & & & \\ & \tilde{U}_{1,11} & \tilde{U}_{1,12} & & & \\ & 0 & \tilde{U}_{1,22} & & & \\ & & & \tilde{U}_{2,11} & \tilde{U}_{2,12} & \\ & & & 0 & \tilde{U}_{2,22} & \\ & & & & & \tilde{U}_{3,11} \end{bmatrix} =$$

$$= \begin{bmatrix} -B_0\tilde{U}_{0,22} & \tilde{U}_{1,11} & \tilde{U}_{1,12} & 0 & 0 & 0 \\ 0 & -A_1\tilde{U}_{1,11} & -A_1\tilde{U}_{1,12} - B_1\tilde{U}_{1,22} & \tilde{U}_{2,11} & \tilde{U}_{2,12} & 0 \\ 0 & 0 & 0 & -A_2\tilde{U}_{2,11} & -A_2\tilde{U}_{2,12} - B_2\tilde{U}_{2,22} & \tilde{U}_{3,11} \end{bmatrix} =$$

$$= \begin{bmatrix} \Phi_{0,22} & \Phi_{1,11} & \Phi_{1,12} & & & \\ & \Phi_{1,21} & \Phi_{1,22} & \Phi_{2,11} & \Phi_{2,12} & \\ & & & \Phi_{2,21} & \Phi_{2,22} & \Phi_{3,11} \end{bmatrix}$$

and has size $Nn_x \times N(nu + nx)$.

We compute only the upper triangular part of the matrix $\Psi = (AU^{-1})(AU^{-1})'$. We make use of the specialized routines `dtrmm`, `dsyrk`[5] and `dlauum`[6], the first two require approximately half and the latter a sixth of the floating-points operations compared to the general matrix multiplication routine `dgemm`. The Ψ matrix is a symmetric positive definite $Nn_x \times Nn_x$ matrix, and has a block tridiagonal form with blocks of size $n_x \times n_x$. It takes the form

$$\Psi = AH^{-1}A' = (AU^{-1})(AU^{-1})' = \begin{bmatrix} \Psi_{11} & \Psi_{12} & 0 \\ \Psi'_{12} & \Psi_{22} & \Psi_{23} \\ 0 & \Psi'_{23} & \Psi_{33} \end{bmatrix}$$

[3]see appendix B.1.2
[4]see appendix B.1.1
[5]see appendix B.1.3
[6]see appendix B.1.4

where in general

$$\begin{aligned}
\Psi_{11} =&\Phi_{0,22}\Phi'_{0,22} + \Phi_{1,11}\Phi'_{1,11} + \Phi_{1,12}\Phi'_{1,12} = \\
=&B_0\tilde{U}_{0,22}\tilde{U}'_{0,22}B'_0 + \tilde{U}_{1,11}\tilde{U}'_{1,11} + \tilde{U}_{1,12}\tilde{U}'_{1,12} = \\
=&B_0\tilde{R}_0B'_0 + \tilde{Q}_1 \\
\Psi_{nn} =&\Phi_{n-1,21}\Phi'_{n-1,21} + \Phi_{n-1,22}\Phi'_{n-1,22} + \Phi_{n,11}\Phi'_{n,11} + \Phi_{n,12}\Phi'_{n,12} = \\
=&(A_{n-1}\tilde{U}_{n-1,11})(A_{n-1}\tilde{U}_{n-1,11})' + \\
&+ (A_{n-1}\tilde{U}_{n-1,12} + B_{n-1}\tilde{U}_{n-1,22})(A_{n-1}\tilde{U}_{n-1,12} + B_{n-1}\tilde{U}_{n-1,22})' + \\
&+ \tilde{U}_{n,11}\tilde{U}'_{n,11} + \tilde{U}_{n,12}\tilde{U}'_{n,12} = \\
=&A_{n-1}\tilde{Q}_{n-1}A'_{n-1} + A_{n-1}\tilde{S}'_{n-1}B'_{n-1} + B_{n-1}\tilde{S}_{n-1}A'_{n-1} + \\
&+ B_{n-1}\tilde{R}_{n-1}B'_{n-1} + \tilde{Q}_n \\
\Psi_{n,n+1} =&\Phi_{n,11}\Phi'_{n,21} + \Phi_{n,12}\Phi'_{n,22} = \\
=&-\tilde{U}_{n,11}\tilde{U}'_{n,11}A'_n - \tilde{U}_{n,12}(A_n\tilde{U}_{n,12} + B_n\tilde{U}_{n,22})' = -\tilde{Q}_nA'_n - \tilde{S}'_nB'_n \\
\Psi_{N,N} =&\Phi_{N-1,21}\Phi'_{N-1,21} + \Phi_{N-1,22}\Phi'_{N-1,22} + \Phi_{N,11}\Phi'_{N,11} = \\
=&(A_{N-1}\tilde{U}_{N-1,11})(A_{N-1}\tilde{U}_{N-1,11})' + \\
&+ (A_{N-1}\tilde{U}_{N-1,12} + B_{N-1}\tilde{U}_{N-1,22})(A_{N-1}\tilde{U}_{N-1,12} + B_{N-1}\tilde{U}_{N-1,22})' + \\
&+ \tilde{U}_{N,11}\tilde{U}'_{N,11} = \\
=&A_{N-1}\tilde{Q}_{N-1}A'_{N-1} + A_{N-1}\tilde{S}'_{N-1}B'_{N-1} + B_{N-1}\tilde{S}_{N-1}A'_{N-1} + \\
&+ B_{N-1}\tilde{R}_{N-1}B'_{N-1} + \tilde{Q}_N
\end{aligned}$$

where

$$\tilde{R}_0 = R_0^{-1}, \qquad \begin{bmatrix} \tilde{Q}_n & \tilde{S}'_n \\ \tilde{S}_n & \tilde{R}_n \end{bmatrix} = \begin{bmatrix} Q_n & S'_n \\ S_n & R_n \end{bmatrix}^{-1}, \qquad \tilde{Q}_N = Q_N^{-1}.$$

The following step is the factorization of the symmetric positive definite matrix Ψ. It can be done efficiently exploiting the block tridiagonal structure of Ψ: the upper factor $\bar{U}$ will be block diagonal banded as well, with just the main and the immediately upper block diagonals non-zero. In fact we have that

$$\Psi = \bar{U}'\bar{U} = \begin{bmatrix} U'_{11} & 0 & 0 \\ U'_{12} & U'_{22} & 0 \\ 0 & U'_{23} & U_{33} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & 0 \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} =$$

$$= \begin{bmatrix} U'_{11}U_{11} & U'_{11}U_{12} & 0 \\ U'_{12}U_{11} & U'_{12}U_{12} + U'_{22}U_{22} & U'_{22}U_{23} \\ 0 & U'_{23}U_{22} & U'_{23}U_{23} + U'_{33}U_{33} \end{bmatrix} = \begin{bmatrix} \Psi_{11} & \Psi_{12} & 0 \\ \Psi'_{12} & \Psi_{22} & \Psi_{23} \\ 0 & \Psi'_{23} & \Psi_{33} \end{bmatrix}$$

and thus in general

$$\begin{aligned}
U'_{11}U_{11} &= \Psi_{11} \\
U'_{n-1,n-1}U_{n-1,n} &= \Psi_{n-1,n} \qquad && n = 2, \dots, N \\
U'_{n,n}U_{n,n} &= \Psi_{n,n} - U'_{n-1,n}U_{n-1,n} \qquad && n = 2, \dots, N
\end{aligned}$$

This leads to the following algorithm: the general block $U_{n,n}$ on the diagonal can be found subtracting to the corresponding block in Ψ the symmetric product of the block immediately above, $\Psi_{n,n} - U'_{n-1,n}U_{n-1,n}$, and then calling the LAPACK routine `dpotrf`, implemeting the Cholesky factorization. The block immediately on the right can be found as $U_{n,n+1} = (U'_{n,n})^{-1}\Psi_{n,n+1}$, using the BLAS routine `dtrsm`[7].

The following step is the solution of the system

$$AH^{-1}A'\pi = b + AH^{-1}g.$$

The right hand side can be build using only matrix-vector operations, as $b + (AU^{-1})((U^{-1})'g) \doteq \beta$. The system can be solved by forward and backward substitution,

$$\bar{U}'\bar{U}\pi = \bar{U}'\gamma = \beta \quad \Rightarrow \quad \bar{U}\pi = \gamma = (\bar{U}')^{-1}\beta \quad \Rightarrow \quad \pi = \bar{U}^{-1}\gamma = \bar{U}^{-1}(\bar{U}')^{-1}\beta$$

using tailored routines.

The first triangular system we have to solve is

$$\begin{bmatrix} U'_{11} & 0 & 0 \\ U'_{12} & U'_{22} & 0 \\ 0 & U'_{23} & U_{33} \end{bmatrix} \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \end{bmatrix} = \begin{bmatrix} U'_{11}\gamma_1 \\ U'_{12}\gamma_1 + U'_{22}\gamma_2 \\ U'_{23}\gamma_2 + U'_{33}\gamma_3 \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

leading to the forward substitution procedure

$$\begin{aligned} \gamma_1 &= (U'_{11})^{-1}\beta_1 \\ \gamma_2 &= (U'_{22})^{-1}(\beta_2 - U'_{12}\gamma_1) \\ \gamma_3 &= (U'_{33})^{-1}(\beta_3 - U'_{23}\gamma_2) \end{aligned}$$

where the required matrix-vector operations are implemented in the BLAS routines `dgemv`[8] and `dtrsv`[9].

The second triangular system we have to solve is

$$\begin{bmatrix} U_{11} & U_{12} & 0 \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} \begin{bmatrix} \pi_1 \\ \pi_2 \\ \pi_3 \end{bmatrix} = \begin{bmatrix} U_{11}\pi_1 + U_{12}\pi_2 \\ U_{22}\pi_2 + U_{23}\pi_3 \\ U_{33}\pi_3 \end{bmatrix} = \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \end{bmatrix}$$

leading to the backward substitution procedure

$$\begin{aligned} \pi_1 &= U_{11}^{-1}(\gamma_1 - U_{12}\pi_2) \\ \pi_2 &= U_{22}^{-1}(\gamma_2 - U_{23}\pi_3) \\ \pi_3 &= U_{33}^{-1}\gamma_3 \end{aligned}$$

The last step is the solution of the system

$$Hx = A'\pi - g.$$

The product at the right hand side can be computed as

$$\begin{bmatrix} -B'_0 & & \\ I & -A'_1 & \\ & -B'_1 & \\ & I & -A'_2 \\ & & -B'_2 \\ & & I \end{bmatrix} \begin{bmatrix} \pi_1 \\ \pi_2 \\ \pi_3 \end{bmatrix} = \begin{bmatrix} -B'_0\pi_1 \\ \pi_1 - A'_1\pi_2 \\ -B'_1\pi_2 \\ \pi_2 - A'_2\pi_3 \\ -B'_2\pi_3 \\ \pi_3 \end{bmatrix}.$$

[7]See appendix B.3.2.
[8]See appendix B.2.1.
[9]See appendix B.3.1.

We have already computed the Cholesky factorization of the block diagonal matrix H, and the inversion of each block factor, and so the system can be solved by two triangular matrix-vector multiplications in each single block,

$$u_0 = \tilde{U}_{0,22}\tilde{U}'_{0,22}(-B'_0\pi_1 - S_0x_0 - s_0)$$

$$\begin{bmatrix} x_n \\ u_n \end{bmatrix} = \begin{bmatrix} \tilde{U}_{n,11} & \tilde{U}_{n,12} \\ 0 & \tilde{U}_{n,22} \end{bmatrix} \begin{bmatrix} \tilde{U}'_{n,11} & 0 \\ \tilde{U}'_{n,12} & \tilde{U}'_{n,22} \end{bmatrix} \begin{bmatrix} \pi_n - A'_n\pi_{n+1} - q_n \\ -B'_n\pi_{n+1} - s_n \end{bmatrix}, \quad n = 1 \dots N-1$$

$$x_N = \tilde{U}_{N,11}\tilde{U}'_{N,11}(\pi_N - q_N)$$

obtaining the vector x.

In the following algorithm 1 we present the overall algorithm, and we compute the complexity of the algorithm as number of floating-point operations up to the quadratic terms. A subscript indicates each BLAS or LAPACK routine used to perform algebra operations.

The asymptotic cost of the algorithm is

$$N\left(\left(\frac{19}{3}n_x^3 + 8n_x^2n_u + 3n_xn_u^2 + \frac{2}{3}n_u^3\right) + \left(\frac{35}{2}n_x^2 + 15n_xn_u + \frac{7}{2}n_u^2\right)\right),$$

that is linear in N and cubic in both n_x and n_u. The above cost is pretty large compared to the cost of the Riccati recursion method, as we will see in the chapter 5.

4.3 Performance analysis

4.3.1 Cost of sub-routines

We can use profiling tool `gprof`[10], that is part of the `gcc` tools collection, to analyze the cost of the different routines. In order to obtain more complete information, in this section we use BLAS and LAPACK libraries obtained compiling source code[11] with the flag `-pg`.

In table 4.1 there is the cost in percentage of the most expensive routines, taken from the flat profile produced by `gprof`: this profile shows how much time is spent in each routine. The test problem is the mass-spring problem 7.2 with $n_x = 128$, $n_u = 1$ and $N = 10$. There are only 4 routines using at least the 5% of the computational time, and they all are part of BLAS, and the first 3 (covering together 84.04%) are part of the level-3 BLAS. This shows as the implementation of the BLAS library is a key factor to have high performances.

The program `gprof` produces also a call graph, giving useful information about the structure of the call graph between routines: this also shows the total time spent in a routine and its sub-routines. We notice that the Cholesky factorization routine `dpotrf` with its sub-routines accounts for the 39.3% of the total computational time, that is much more than the theoretical (in the approximation $n_x \gg n_u$) $\frac{2/3n_x^3+n_x^2n_u}{19/3n_x^2+8n_x^2n_u} = 10.6\%$. Analogously, the triangular matrix inversion routine `dtrtri` with its sub-routines accouts for the 19.3%, and

[10]See appendix A

[11]The standard BLAS and LAPACK implementations can be found at the web sites `www.netlib.org/blas` and `www.netlib.org/lapack`.

Algorithm 1 Schur complement method for the solution of problem 2.1

```
Require: $(x_0, \{Q_n\}, \{S_n\}, \{R_n\}, \{q_n\}, \{s_n\}, \{A_n\}, \{B_n\}, \{b_n\}, Q_N, q_N)$

  $U_{0,22} \leftarrow_{\text{dpotrf}} chol(R_0)$                    ▷ $\frac{1}{3}n_u^3 + \frac{1}{2}n_u^2$ flops
  for $n = 1 \to N-1$ do
      $\begin{bmatrix} U_{n,11} & U_{n,12} \\ 0 & U_{n,22} \end{bmatrix} \leftarrow_{\text{dpotrf}} chol\left(\begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix}\right)$
                    ▷ $(N-1)(\frac{1}{3}(n_x+n_u)^3 + \frac{1}{2}(n_x+n_u)^2)$ flops
  end for
  $U_{N,11} \leftarrow_{\text{dpotrf}} chol(Q_N)$                    ▷ $\frac{1}{3}n_x^3 + \frac{1}{2}n_x^2$ flops

  $\tilde{U}_{0,22} \leftarrow_{\text{dtrtri}} U_{0,22}^{-1}$                    ▷ $\frac{1}{3}n_u^3$ flops
  for $n = 1 \to N-1$ do
      $\begin{bmatrix} \tilde{U}_{n,11} & \tilde{U}_{n,12} \\ 0 & \tilde{U}_{n,22} \end{bmatrix} \leftarrow_{\text{dtrtri}} \begin{bmatrix} U_{n,11} & U_{n,12} \\ 0 & U_{n,22} \end{bmatrix}^{-1}$    ▷ $(N-1)\frac{1}{3}(n_x+n_u)^3$ flops
  end for
  $\tilde{U}_{N,11} \leftarrow_{\text{dtrtri}} U_{N,11}^{-1}$                    ▷ $\frac{1}{3}n_x^3$ flops

  $\Phi_{0,22} \leftarrow -B_0 \cdot_{\text{dtrmm}} \tilde{U}_{0,22}$                    ▷ $n_x n_u^2$ flops
  for $n = 1 \to N-1$ do
      $\begin{bmatrix} \Phi_{n,11} & \Phi_{n,12} \\ \Phi_{n,21} & \Phi_{n,22} \end{bmatrix} \leftarrow \begin{bmatrix} \tilde{U}_{n,11} & \tilde{U}_{n,12} \\ -A_n \cdot_{\text{dtrmm}} \tilde{U}_{n,11} & -A_n \cdot_{\text{dgemm}} \tilde{U}_{n,12} - B_n \cdot_{\text{dtrmm}} \tilde{U}_{n,22} \end{bmatrix}$
                    ▷ $(N-1)(n_x^3 + 2n_x^2 n_u + n_x n_u^2)$ flops
  end for
  $\Phi_{N,11} \leftarrow \tilde{U}_{N,11}$

  $\Psi_{11} = \Phi_{0,22} \cdot_{\text{dsyrk}} \Phi_{0,22}' + \Phi_{1,11} \cdot_{\text{dlauum}} \Phi_{1,11}' + \Phi_{1,12} \cdot_{\text{dsyrk}} \Phi_{1,12}'$
                    ▷ $\frac{1}{3}n_x^3 + 2n_x^2 n_u + \frac{1}{2}n_x^2 + 2n_x n_u$ flops
  $\Psi_{12} = \Phi_{1,11} \cdot_{\text{dtrmm}} \Phi_{1,21}' + \Phi_{1,12} \cdot_{\text{dgemm}} \Phi_{1,22}'$                    ▷ $n_x^3 + 2n_x^2 n_u$ flops

  for $n = 2 \to N-1$ do
      $\Psi_{nn} = \Phi_{n-1,21} \cdot_{\text{dsyrk}} \Phi_{n-1,21}' + \Phi_{n-1,22} \cdot_{\text{dsyrk}} \Phi_{n-1,22}' + \Phi_{n,11} \cdot_{\text{dlauum}} \Phi_{n,11}' +$
  $\Phi_{n,12} \cdot_{\text{dsyrk}} \Phi_{n,12}'$
                    ▷ $(N-2)(\frac{4}{3}n_x^3 + 2n_x^2 n_u + \frac{3}{2}n_x^2 + 2n_x n_u)$ flops
      $\Psi_{n,n+1} = \Phi_{n,11} \cdot_{\text{dtrmm}} \Phi_{n,21}' + \Phi_{n,12} \cdot_{\text{dgemm}} \Phi_{n,22}'$
                    ▷ $(N-2)(n_x^3 + 2n_x^2 n_u)$ flops
  end for
  $\Psi_{N,N} = \Phi_{N-1,21} \cdot_{\text{dsyrk}} \Phi_{N-1,21}' + \Phi_{N-1,22} \cdot_{\text{dsyrk}} \Phi_{N-1,22}' + \Phi_{N,11} \cdot_{\text{dlauum}} \Phi_{N,11}'$
                    ▷ $\frac{4}{3}n_x^3 + n_x^2 n_u + \frac{1}{2}n_x^2 + n_x n_u$ flops
```

```
$U_{11} \leftarrow_{\text{dpotrf}} \mathit{chol}(\Psi_{11})$    $\triangleright\ \frac{1}{3}n_x^3 + \frac{1}{2}n_x^2$ flops
**for** $n = 2 \to N$ **do**
    $U_{n-1,n} \leftarrow_{\text{dpotrf}} (U'_{n-1,n-1})^{-1}\Psi_{n-1,n}$    $\triangleright\ (N-1)n_x^3$ flops
    $U_{n,n} \leftarrow_{\text{dpotrf}} \mathit{chol}(\Psi_{n,1} - U'_{n-1,n} \cdot_{\text{dsyrk}} U_{n-1,n})$ $\triangleright\ (N-1)(\frac{4}{3}n_x^3 + \frac{3}{2}n_x^2)$ flops
**end for**

$\phi_{0,2} \leftarrow \tilde{U}_{0,22} \cdot_{\text{dtrmv}} (s_0 + S_0 \cdot_{\text{dgemv}} x_0)$    $\triangleright\ 2n_x n_u + n_u^2$ flops
**for** $N = 1 \to N-1$ **do**
    $\phi_{n,1} \leftarrow \tilde{U}_{n,11} \cdot_{\text{dtrmv}} q_n + \tilde{U}_{n,12} \cdot_{\text{dgemv}} s_n$    $\triangleright\ (N-1)(n_x^2 + 2n_x n_u)$ flops
    $\phi_{n,2} \leftarrow \tilde{U}_{n,22} \cdot_{\text{dtrmv}} s_n$    $\triangleright\ (N-1)n_u^2$ flops
**end for**
$\phi_{N,1} \leftarrow \tilde{U}_{N,11} \cdot_{\text{dtrmv}} q_N$    $\triangleright\ n_x^2$ flops

$\beta_1 \leftarrow b_0 + A_0 \cdot_{\text{dgemv}} x_0 + \Phi_{0,22} \cdot_{\text{dgemv}} \phi_{0,2} + \Phi_{1,11} \cdot_{\text{dtrmv}} \phi_{1,1} + \Phi_{1,12} \cdot_{\text{dgemv}} \phi_{1,2}$
    $\triangleright\ 3n_x^2 + 4n_x n_u$ flops
**for** $n = 2 \to N-1$ **do**
    $\beta_n \leftarrow b_{n-1} + \Phi_{n-1,21} \cdot_{\text{dgemv}} \phi_{n-1,1} + \Phi_{n-1,22} \cdot_{\text{dgemv}} \phi_{n-1,2} + \Phi_{n,11} \cdot_{\text{dtrmv}}$
$\phi_{n,1} + \Phi_{n,12} \cdot_{\text{dgemv}} \phi_{n,2}$
    $\triangleright\ (N-2)(3n_x^2 + 4n_x n_u)$ flops
**end for**
$\beta_N \leftarrow b_{N-1} + \Phi_{N-1,21} \cdot_{\text{dgemv}} \phi_{N-1,1} + \Phi_{N-1,22} \cdot_{\text{dgemv}} \phi_{N-1,2} + \Phi_{N,11} \cdot_{\text{dtrmv}} \phi_{N,1}$
    $\triangleright\ 3n_x^2 + 2n_x n_u$ flops

$\gamma_1 \leftarrow_{\text{dtrsv}} (U'_{11})^{-1}\beta_1$    $\triangleright\ n_x^2$ flops
**for** $n = 2 \to N$ **do**
    $\gamma_n \leftarrow_{\text{dtrsv}} (U'_{n,n})^{-1}(\beta_n - U'_{n-1,n} \cdot_{\text{dgemv}} \gamma_{n-1})$    $\triangleright\ (N-1)3n_x^2$ flops
**end for**

$\pi_N \leftarrow_{\text{dtrsv}} U_{n,n}^{-1}\gamma_N$    $\triangleright\ n_x^2$ flops
**for** $n = N-1 \to 1$ **do**
    $\pi_n \leftarrow_{\text{dtrsv}} U_{n,n}^{-1}(\gamma_n - U_{n,n+1} \cdot_{\text{dgemv}} \pi_{n+1})$    $\triangleright\ (N-1)3n_x^2$ flops
**end for**

$u_0 = \tilde{U}_{0,22} \cdot_{\text{dtrmv}} (\tilde{U}'_{0,22} \cdot_{\text{dtrmv}} (-B'_0 \cdot_{\text{dgemv}} \pi_1 - (s_0 + S_0 x_0)))$
    $\triangleright\ 2n_x n_u + 2n_u^2$ flops
**for** $n = 1 \to N-1$ **do**
    $\begin{bmatrix} x_n \\ u_n \end{bmatrix} = \begin{bmatrix} \tilde{U}_{n,11} & \tilde{U}_{n,12} \\ 0 & \tilde{U}_{n,22} \end{bmatrix} \cdot_{\text{dtrmv}} \left( \begin{bmatrix} \tilde{U}'_{n,11} & 0 \\ \tilde{U}'_{n,12} & \tilde{U}'_{n,22} \end{bmatrix} \cdot_{\text{dtrmv}} \begin{bmatrix} \pi_n - A'_n \cdot_{\text{dgemv}} \pi_{n+1} - q_n \\ -B'_n \cdot_{\text{dgemv}} \pi_{n+1} - s_n \end{bmatrix} \right)$
    $\triangleright\ (N-1)(2(n_x + n_u)^2 + 2n_x^2 + 2n_x n_u)$ flops
**end for**
$x_N = \tilde{U}_{N,11} \cdot_{\text{dtrmv}} \left( \tilde{U}'_{N,11} \cdot_{\text{dtrmv}} (\pi_N - q_N) \right)$    $\triangleright\ 2n_x^2$ flops

**return** $(\{x_n\}, \{u_n\})$
```

routine	percentage
dsyrk	40.31
dtrsm	26.74
dtrmm	16.99
dgemv	5.08
others	10.88

Table 4.1: Cost of the most expensive routines in the Schur complement method, using as test problem the mass-spring problem 7.2, with $n_x = 128$, $n_u = 1$ and $N = 10$. Only routines using at least 5% of the computational time are reported in the table.

the rank-k update with triangular matrices routine `dlauum` with its subroutines for the 11.2% instead of the theoretic $\frac{1/3n_x^3+n_x^2n_u}{19/3n_x^2+8n_x^2n_u} = 5.8\%$.

The efficient implementation of the above routines is the bottle neck in the implementation of the Schur method for the solution of the extended linear quadratic control problem.

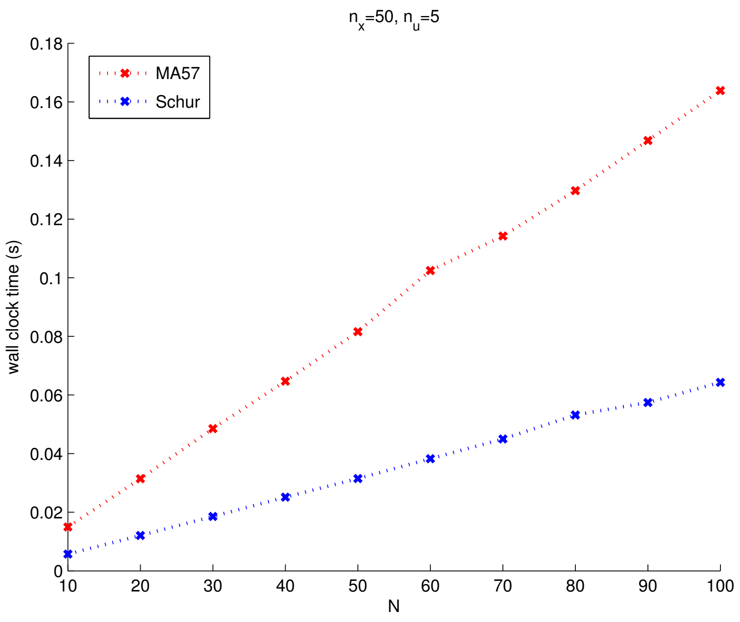
4.3.2 Comparative test

In figure 4.1a and 4.1b there are plots comparing the performance of the Schur complement method (the algorithm developed in this chapter) with the direct sparse solver MA57 (analyzed in chapter 3). MA57 shows a better performance compared to PARDISO, the other direct sparse solver considered. The test problem is the mass-spring problem 7.2, and as usual $n_u \leq n_x/2$. The tables containing all the data are D.3 and D.4.

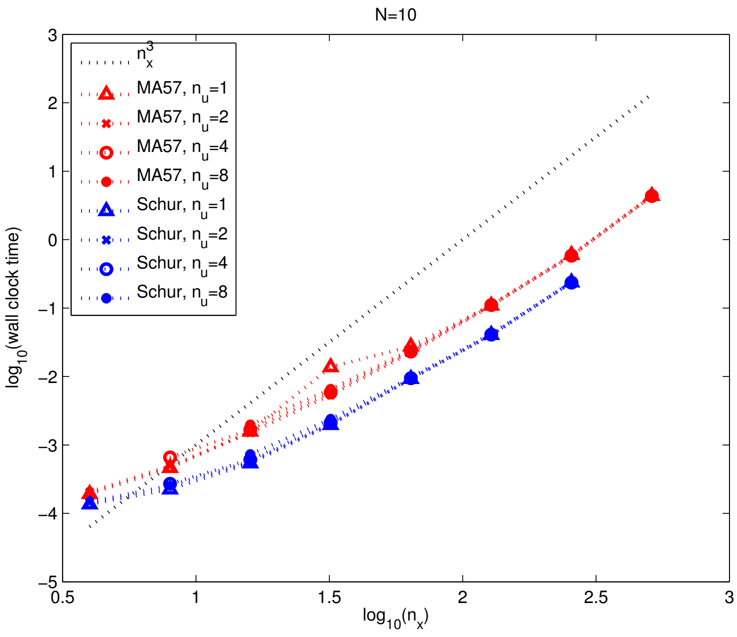
In figure 4.1a n_x and n_u are fixed, and the only N is varied. The behavior of the Schur complement method algorithm is clearly linear in N, and the algorithm is at least twice as fast as the MA57 solver.

In figure 4.1b N is fixed, while n_x and n_u are varied. The algorithm behaves approximately as the MA57 solver: the value of n_u influences just slightly the computational time, as long as $n_u \ll n_x$ holds. Anyway, the Schur complement method algorithm is approximately 2.5 times faster than the MA57 solver, at least for large systems.

However, our current version of the algorithm implementing the Schur complement method crashes for large systems: the problem seems to be connected with to the large quantity of memory needed to store the data within the algorithm, that is quadratic in n_x. In the case of $N = 10$, using the MKL BLAS the crash happens for $n_x > 340$. Further work is needed to investigate and fix the problem.



(a) $n_x = 50$ and $n_u = 5$ fixed, N varying.



(b) $N = 10$ fixed, $n_x \in \{4, 8, 16, 32, 64, 128, 256, 512\}$ and $n_u \in \{1, 2, 4, 8\}$ varying.

Figure 4.1: Comparison of the performances of the direct sparse solver MA57 and the Schur complement method algorithm in the solution of the problem 2.1, using as test problem the mass-spring problem 7.2.

Chapter 5

Riccati Recursion Method

The Riccati recursion is a well known method for the solution of the standard linear quadratic control problem 2.2, and the algorithm can be modified for the solution of problem 2.1. The Riccati recursion methods for the solution of 2.2 and 2.1 have asymptotic complexity $N(n_x + n_u)^3$ (the same as the methods considered in chapters 3 and 4) but in practice they are faster.

5.1 Derivation

There are several ways to derive the Riccati recursion methods for the solution of problems 2.2 and 2.1. In what follows we consider:

- direct derivation from the cost function expression
- derivation using dynamic programming
- derivation as block factorization procedure for the solution of the KKT system.

In this section we derive Riccati recursion methods for the solution of both problems 2.2 and 2.1, using the three considered methods.

5.1.1 Derivation from the Cost Function Expression

This derivation technique can be found also in [FM03].

Standard Linear Quadratic Control Problem

In this chapter we use a equivalent expression for the standard linear quadratic control problem 2.2:

$$\begin{aligned} \min_{u_n, x_{n+1}} \quad & \phi = \sum_{n=0}^{N-1} \begin{bmatrix} x'_n & u'_n \end{bmatrix} \begin{bmatrix} Q & S' \\ S & R \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + x'_N P x_N \\ s.t. \quad & x_{n+1} = Ax_n + Bu_n \end{aligned} \tag{5.1}$$

The difference is the absence of the factor $\frac{1}{2}$, and the use of P instead of Q_N.

In this section we show that the solution of the problem 5.1 is the input sequence u_n obtained as linear feedback from the state x_n with a time variant gain matrix K_n,

$$u_n = K_n x_n$$

computed using the Riccati recursion.

The matrix R is assumed to be symmetric positive definite, and the matrix $\begin{bmatrix} Q & S' \\ S & R \end{bmatrix}$ to be symmetric positive semi-definite.

The first step in solving 5.1 is to rewrite the cost function ϕ in a more useful form. We notice that for each sequence of generic squared matrices P_n, $n \in \{0, 1, \dots, N\}$ of size $n_x \times n_x$, we have

$$\begin{aligned} 0 &= x'_0 P_0 x_0 - x'_0 P_0 x_0 + x'_1 P_1 x_1 - x'_1 P_1 x_1 + \cdots + x'_N P_N x_N - x'_N P_N x_N = \\ &= x'_0 P_0 x_0 - x'_N P_N x_N + \sum_{n=0}^{N-1} (x'_{n+1} P_{n+1} x_{n+1} - x'_n P_n x_n) = \\ &= x'_0 P_0 x_0 - x'_N P_N x_N + \sum_{n=0}^{N-1} \left((Ax_n + Bu_n)' P_{n+1} (Ax_n + Bu_n) - x'_n P_n x_n \right) = \\ &= x'_0 P_0 x_0 - x'_N P_N x_N + \sum_{n=0}^{N-1} \left(\begin{bmatrix} x'_n & u'_n \end{bmatrix} \begin{bmatrix} A' \\ B' \end{bmatrix} P_{n+1} \begin{bmatrix} A & B \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} - x'_n P_n x_n \right) = \\ &= x'_0 P_0 x_0 - x'_N P_N x_N + \sum_{n=0}^{N-1} \begin{bmatrix} x'_n & u'_n \end{bmatrix} \begin{bmatrix} A' P_{n+1} A - P_n & A' P_{n+1} B \\ B' P_{n+1} A & B' P_{n+1} B \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix}. \end{aligned}$$

Adding this expression to the cost function expression in 5.1, we obtain

$$\begin{aligned} \phi =& x'_0 P_0 x_0 + x'_N (P - P_N) x_N + \\ &+ \sum_{n=0}^{N-1} \begin{bmatrix} x'_n & u'_n \end{bmatrix} \begin{bmatrix} Q + A' P_{n+1} A - P_n & S' + A' P_{n+1} B \\ S + B' P_{n+1} A & R + B' P_{n+1} B \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix}. \end{aligned}$$

Choosing the ending value of the sequence P_n as $P_N = P$ the term $x'_N(P - P_N)x_N$ vanishes. Choosing the remaining element of the sequence such that the expression of P_n is obtained from P_{n+1} as

$$P_n = Q + A' P_{n+1} A - (S + B' P_{n+1} A)' (R + B' P_{n+1} B)^{-1} (S + B' P_{n+1} A) \tag{5.2}$$

we have that the quadratic terms of the cost function factorize as

$$\begin{aligned}
\phi &= x_0'P_0x_0 + \\
&+ \sum_{n=0}^{N-1} \begin{bmatrix} x_n' & u_n' \end{bmatrix} \begin{bmatrix} (S'+A'P_{n+1}B)(R+B'P_{n+1}B)^{-1}(S+B'P_{n+1}A) & S'+A'P_{n+1}B \\ S+B'P_{n+1}A & R+B'P_{n+1}B \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} = \\
&= x_0'P_0x_0 + \\
&+ \sum_{n=0}^{N-1} \begin{bmatrix} x_n' & u_n' \end{bmatrix} \begin{bmatrix} (S+B'P_{n+1}A)' \\ R+B'P_{n+1}B \end{bmatrix} (R+B'P_{n+1}B)^{-1} \begin{bmatrix} S+B'P_{n+1}A & R+B'P_{n+1}B \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} = \\
&= x_0'P_0x_0 + \sum_{n=0}^{N-1} v_n'(R+B'P_{n+1}B)^{-1}v_n
\end{aligned}$$

where

$$v_n = (S+B'P_{n+1}A)x_n + (R+B'P_{n+1}B)u_n.$$

Equation 5.2 is the well known expression of the Riccati recursion.

In the last expression of the cost function the term $x_0'P_0x_0$ is a constant, and, since the matrix $(R+B'P_{n+1}B)$ is positive definite (as R is positive definite and P_{n+1} is at least positive semi-definite, as shown in 5.1.4), the sum at the second term is positive or zero, and the minimum 0 can only be obtained choosing $v_n = 0$ for each n. This implies

$$u_n = (R+B'P_{n+1}B)^{-1}(S+B'P_{n+1}A)x_n \doteq K_nx_n.$$

This proves that the optimal input sequence can be written as a linear feedback from the state, with a time variant gain matrix obtained using the Riccati recursion.

The optimal value of the cost function is

$$\phi^* = x_0'P_0x_0.$$

The Riccati recursion method for the solution of problem 2.2 is summarized in algorithm 1.

Extended Linear Quadratic Control Problem

Also in the case of problem 2.1, in this chapter we use an equivalent expression

$$\begin{aligned}
\min_{u_n, x_{k+1}} \quad & \phi = \sum_{n=0}^{N-1} \frac{1}{2} \begin{bmatrix} x_n' & u_n' \end{bmatrix} \begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \begin{bmatrix} q_n' & s_n' \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \rho_n + \\
& + \frac{1}{2}x_N'Px_N + p'x_N + \pi \\
s.t. \quad & x_{n+1} = A_nx_n + B_nu_n + b_n
\end{aligned} \tag{5.3}$$

where we use P, p and π instead of Q_N, q_N and ρ_N.

In this section we show that the solution of problem 5.3 is an input sequence u_n obtained as affine feedback from the state x_n with a time variant gain matrix K_n and constant k_n,

$$u_n = K_nx_n + k_n$$

Algorithm 1 Riccati recursion method for the solution of problem 2.2

```
$P_{n+1} \leftarrow P$
for $n = N-1 \rightarrow 0$ do
    $R_{e,n} \leftarrow R + B'P_{n+1}B$
    $K_n \leftarrow -R_{e,n}^{-1}(S + B'P_{n+1}A)$
    $P_n \leftarrow Q + A'P_{n+1}A - K_n'R_{e,n}K_n$
end for
for $n = 0 \rightarrow N-1$ do
    $u_n \leftarrow K_n x_n$
    $x_{n+1} \leftarrow Ax_n + Bu_n$
end for
```

obtianed using the Riccati recursion.

Again the matrices R_n are assumed to be symmetric positive definite, and the matrices $\begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix}$ to be symmetric positive semi-definite.

Also in this case the first step to solve 5.3 is to rewrite the cost function. Like we presented in the solution procedure of problem 5.1, for each sequence of general matrices P_n of size $n_x \times n_x$ holds the expression

$$\begin{aligned}
0 =& \frac{1}{2}x_0'P_0x_0 - \frac{1}{2}x_N'P_Nx_N + \frac{1}{2}\sum_{n=0}^{N-1}\left(x_{n+1}'P_{n+1}x_{n+1} - x_n'P_nx_x\right) = \\
=& \frac{1}{2}x_0'P_0x_0 - \frac{1}{2}x_N'P_Nx_N + \\
&+ \frac{1}{2}\sum_{n=0}^{N-1}((A_nx_n + B_nu_n + b_n)'P_{n+1}(A_nx_n + B_nu_n + b_n) - x_n'P_nx_x) = \\
=& \frac{1}{2}x_0'P_0x_0 - \frac{1}{2}x_N'P_Nx_N + \\
&+ \frac{1}{2}\sum_{n=0}^{N-1}\left(\begin{bmatrix} x_n' & u_n' \end{bmatrix}\begin{bmatrix} A_n'P_{n+1}A_n - P_n & A_n'P_{n+1}B_n \\ B_n'P_{n+1}A_n & B_n'P_{n+1}B_n \end{bmatrix}\begin{bmatrix} x_n \\ u_n \end{bmatrix} + \right. \\
&\left. +2\begin{bmatrix} b_n'P_{n+1}A_n & b_n'P_{n+1}B_n \end{bmatrix}\begin{bmatrix} x_n \\ u_n \end{bmatrix} + b_n'P_{n+1}b_n\right)
\end{aligned}$$

In a similar way, given any sequence of generic vectors p_n of length n_x, holds the expression

$$\begin{aligned}0 &= p_0'x_0 - p_N'x_N + \sum_{n=0}^{N-1}\left(p_{n+1}'x_{n+1} - p_n'x_n\right) = \\ &= p_0'x_0 - p_N'x_N + \sum_{n=0}^{N-1}\left(p_{n+1}'(A_nx_n + B_nu_n + b_n) - p_n'x_n\right) = \\ &= p_0'x_0 - p_N'x_N + \sum_{n=0}^{N-1}\left(\begin{bmatrix} p_{n+1}'A_n - p_n' & p_{n+1}'B_n \end{bmatrix}\begin{bmatrix} x_n \\ u_n \end{bmatrix} + p_{n+1}'b_n\right)\end{aligned}$$

The cost function can thus be rewritten as

$$\begin{aligned}\phi =& \frac{1}{2}x_0'P_0x_0 + p_0'x_0 + \frac{1}{2}x_N'(P - P_N)x_x + (p - p_N)'x_n + \pi + \\ &+ \sum_{n=0}^{N-1}\left(\frac{1}{2}\begin{bmatrix} x_n' & u_n' \end{bmatrix}\begin{bmatrix} Q_n + A_n'P_{n+1}A_n - P_n & S_n' + A_n'P_{n+1}B_n \\ S_n + B_n'P_{n+1}A_n & R_n + B_n'P_{n+1}B_n \end{bmatrix}\begin{bmatrix} x_n \\ u_n \end{bmatrix} + \right. \\ &+ \begin{bmatrix} q_n' + b_n'P_{n+1}A_n + p_{n+1}'A_n - p_n' & s_n' + b_n'P_{n+1}B_n + p_{n+1}'B_n \end{bmatrix}\begin{bmatrix} x_n \\ u_n \end{bmatrix} + \\ &\left. + \frac{1}{2}b_n'P_{n+1}b_n + p_{n+1}'b_n + \rho_n\right).\end{aligned}$$

Choosing again the sequence P_n such that $P_N = P$, the term $\frac{1}{2}x_N'(P - P_N)x_N$ is zero. Choosing the remaining elements of the sequence such that the expression of P_n is obtained from P_{n+1} satisfying the Riccati recursion

$$P_n = Q_n + A_n'P_{n+1}A_n - (S_n + B_n'P_{n+1}A_n)'(R_n + B_n'P_{n+1}B_n)^{-1}(S_n + B_n'P_{n+1}A_n)$$

we have that the term quadratic in $\begin{bmatrix} x_n' & u_n' \end{bmatrix}'$ in the cost function factorizes as

$$\begin{aligned}\frac{1}{2}\begin{bmatrix} x_n' & u_n' \end{bmatrix}\begin{bmatrix} (S_n + B_n'P_{n+1}A_n)' \\ R_n + B_n'P_{n+1}B_n \end{bmatrix}(R_n + B_n'P_{n+1}B_n)^{-1}\begin{bmatrix} S_n + B_n'P_{n+1}A_n & R_n + B_n'P_{n+1}B_n \end{bmatrix}\begin{bmatrix} x_n \\ u_n \end{bmatrix} = \\ = v_n'(R_n + B_n'P_{n+1}B_n)^{-1}v_n \doteq v_n'H_nv_n\end{aligned}$$

where again

$$v_n = (S_n + B_n'P_{n+1}A_n)x_n + (R_n + B_n'P_{n+1}B_n)u_n$$

and the matrix $H_n = (R_n + B_n'P_{n+1}B_n)^{-1}$ is positive definite.

The cost function expression becomes

$$\begin{aligned}\phi =& \frac{1}{2}x_0'P_0x_0 + p_0'x_0 + (p - p_N)'x_n + \pi + \\ &+ \sum_{n=0}^{N-1}\left(\frac{1}{2}v_n'H_nv_n + (b_n'P_{n+1}b_n + p_{n+1}'b_n + \rho_n) + \right. \\ &\left. + \begin{bmatrix} q_n' + b_n'P_{n+1}A_n + p_{n+1}'A_n - p_n' & s_n' + b_n'P_{n+1}B_n + p_{n+1}'B_n \end{bmatrix}\begin{bmatrix} x_n \\ u_n \end{bmatrix}\right).\end{aligned}$$

The aim at this point is to handle also the linear term in $\begin{bmatrix} x_n' & u_n' \end{bmatrix}'$ such that it can be rewritten as a term in v_n. Choosing the sequence p_n such that $p_N = p$

the term $(p - p_N)'x_N$ is zero. Choosing the remaining terms of the sequence such that the expression of p_n is obtained from p_{n+1} as

$$\begin{aligned} &q'_n + b'_n P_{n+1} A_n + p'_{n+1} A_n - p'_n = \\ &\quad = (s'_n + b'_n P_{n+1} B_n + p'_{n+1} B_n)(R_n + B'_n P_{n+1} B_n)^{-1}(S_n + B'_n P_{n+1} A_n), \end{aligned}$$

that means

$$\begin{aligned} p_n = {} & q_n + A'_n(P_{n+1}b_n + p_{n+1}) - \\ & - (S_n + B'_n P_{n+1} A_n)'(R_n + B'_n P_{n+1} B_n)^{-1}(s_n + B'_n(P_{n+1}b_n + p_{n+1})), \end{aligned}$$

the linear term becomes

$$(s_n + B'_n(P_{n+1}b_n + p_{n+1}))(R_n + B'_n P_{n+1} B_n)^{-1} \begin{bmatrix} S_n + B'_n P_{n+1} A_n & R_n + B'_n P_{n+1} B_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} =$$

$$= (s_n + B'_n(P_{n+1}b_n + p_{n+1}))(R_n + B'_n P_{n+1} B_n)^{-1} v_n \doteq g'_n v_n.$$

The cost function expression finally becomes

$$\phi = \frac{1}{2}x'_0 P_0 x_0 + p'_0 x_0 + \pi + \sum_{n=0}^{N-1} \left(\frac{1}{2} v'_n H_n v_n + g'_n v_n + (\frac{1}{2} b'_n P_{n+1} b_n + p'_{n+1} b_n + \rho_n) \right)$$

and it must be minimized as a function of v_n. We notice that the term outside the sum is a constant, while for each n the minimum of the positive definite quadratic function is obtained setting the gradient with respect to v_n to zero:

$$\nabla_{v_n} \left(\frac{1}{2} v'_n H_n v_n + g'_n v_n + (\frac{1}{2} b'_n P_{n+1} b_n + p'_{n+1} b_n + \rho_n) \right) = H_n v_n + g_n = 0.$$

This means $v_n = -H_n^{-1} g_n$, and then

$$\begin{aligned} &(S_n + B'_n P_{n+1} A_n)x_n + (R_n + B'_n P_{n+1} B_n)u_n = \\ &\quad = -(R_n + B'_n P_{n+1} B_n)(R_n + B'_n P_{n+1} B_n)^{-1}(s_n + B'_n(P_{n+1}b_n + p_{n+1})) \end{aligned}$$

and finally

$$\begin{aligned} u_n = & -(R_n + B'_n P_{n+1} B_n)^{-1}(S_n + B'_n P_{n+1} A_n)x_n - \\ & -(R_n + B'_n P_{n+1} B_n)^{-1}(s_n + B'_n(P_{n+1}b_n + p_{n+1})) = \\ = & K_n x_n + k_n. \end{aligned}$$

This shows that the optimal input sequence u_n can be obtained as a time variant affine feedback from the state.

The optimal value of the cost function is thus

$$\begin{aligned} \phi^* &= \frac{1}{2}x'_0 P_0 x_0 + p'_0 x_0 + \left(\pi + \sum_{n=0}^{N-1} \left(-\frac{1}{2} g'_n H_n^{-1} g_n + \frac{1}{2} b'_n P_{n+1} b_n + p'_{n+1} b_n + \rho_n \right) \right) \\ &= \frac{1}{2}x'_0 P_0 x_0 + p'_0 x_0 + \pi_0 \end{aligned}$$

where

$$\pi_0 = \pi + \sum_{n=0}^{N-1} \left(\frac{1}{2} b_n' P_{n+1} b_n + p_{n+1}' b_n + \rho_n - \right.$$
$$\left. - \frac{1}{2} (s_n + B_n'(P_{n+1}b_n + p_{n+1}))' (R_n + B_n' P_{n+1} B_n)^{-1} (s_n + B_n'(P_{n+1}b_n + p_{n+1})) \right)$$

The Riccati recursion method for the solution of problem 2.1 is stated in algorithm 2.

Algorithm 2 Riccati recursion method for the solution of problem 2.1

```
$P_{n+1} \leftarrow P$
$p_{n+1} \leftarrow p$
for $n = N-1 \to 0$ 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
for $n = 0 \to N-1$ do
    $u_n \leftarrow K_n x_n + k_n$
    $x_{n+1} \leftarrow A_n x_n + B_n u_n + b_n$
end for
```

5.1.2 Derivation using Dynamic Programming

In this section we derive the same Riccati recursion methods for the solution of both problems 2.2 and 2.1 using dynamic programming. This derivation technique can be found also in [Jør05].

Standard Linear Quadratic Control Problem

Using dynamic programming, it is possible to find the solution of problem 2.2 by solving a sequence of sub-problems of increasing size. The solution of the general sub-problem of size n is an input sequence of length n, obtained adding a new element to the sequence solution of length $n-1$ obtained solving the sub-problem of size $n-1$. The procedure is based on the fact that the optimal input sequence for the problem of size n contains as sub-sequence the optimal input sequence for the problem of size $n-1$. The solution can thus be build starting from the last stage $N-1$, solving a trivial problem of size 1. Then, at stage $N-2$, the solution is expanded with another term, computed solving a trivial problem of size 1, and so on. In what follows there are the details.

Let us define φ_n the element of the cost function at time n,

$$\varphi_n = \begin{bmatrix} x_n' & u_n' \end{bmatrix} \begin{bmatrix} Q & S' \\ S & R \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix},$$

and ϕ_n the cost function between time n and N,

$$\phi_n = \sum_{i=n}^{N-1} \begin{bmatrix} x_i' & u_i' \end{bmatrix} \begin{bmatrix} Q & S' \\ S & R \end{bmatrix} \begin{bmatrix} x_i \\ u_i \end{bmatrix} + x_N' P_N x_N = \sum_{i=n}^{N-1} \varphi_i + x_N' P_N x_N.$$

We notice that, given the system dynamic $x_{n+1} = Ax_n + Bu_n$ and the generic input sequence u_i, $i \in \{n, \dots, N-1\}$, the state sequence depends only on the initial state x_n. This means that the cost function value depends only on the input sequence u_i and initial state x_n. Furthermore, the optimal input sequence u_i^* (i.e. the input sequence minimizing the cost function ϕ_n) is in general a function of the initial state x_n.

Let us define the minimum value V_n^* of the cost function ϕ_n as

$$V_n^* = \min_{u_i, x_{i+1}} \phi_n.$$

The above discussion implies that also the minimum value V_n^* (obtained for the optimal input sequence $u_i^*(x_n)$, $i \in \{n, \dots, N-1\}$) is function only of the initial state x_n: we stress this using the notation $V_n^*(x_n)$.

The possibility to use the dynamic programming approach for the computation of the optimal input sequence is based on the following result, a particular instance of Bellman's principle of optimality:

Proposition 7. *If $u_i^*(x_n)$, $i \in \{n, \dots, N-1\}$ is the optimal input sequence for the cost function ϕ_n, then $u_i^*(x_{n+1})$, $i \in \{n+1, \dots, N-1\}$ is an optimal input sequence for the cost function ϕ_{n+1}.*

Proof. The proof is by contradiction. Let us assume that the optimal input sequence for the cost function ϕ_{n+1} is $\hat{u}_i(x_{n+1})$, $i \in \{n+1, \dots, N-1\}$. This in particular means that

$$\phi_{n+1}(x_{n+1}, \{\hat{u}_{n+1}, \dots, \hat{u}_{N-1}\}) < \phi_{n+1}(x_{n+1}, \{u_{n+1}^*, \dots, u_{N-1}^*\})$$

for all the possible values of the initial state x_{n+1}.

In this way it is possible to build the input sequence $\{u_n^*, \hat{u}_{n+1}, \dots, \hat{u}_{N-1}\}$ for the cost function ϕ_n such that

$$\begin{aligned} \phi_n(x_n, \{u_n^*, \hat{u}_{n+1}, \dots, \hat{u}_{N-1}\}) &= \varphi_n(x_n, u_i^*) + \phi_{n+1}(x_{n+1}, \{\hat{u}_{n+1}, \dots, \hat{u}_{N-1}\}) < \\ < \varphi_n(x_n, u_n^*) + \phi_{n+1}(x_{n+1}, \{u_{n+1}^*, \dots, u_{N-1}^*\}) &= \phi_n(x_n, \{u_n^*, u_{n+1}^*, \dots, u_{N-1}^*\}) \end{aligned}$$

and then u_i^* is not the optimal input sequence for the cost function ϕ_n, against the hypothesis. □

The previous proposition implies that it is possible to build an optimal input sequence for the cost function ϕ_n adding the proper element $u_n^*(x_n)$ to the input sequence $u_i^*(x_{n+1})$, $i \in \{n+1, \dots, N-1\}$, optimal for the reduced cost function ϕ_{n+1}. The problem of finding the optimal input sequence for the cost function ϕ_{n+1} is then a sub-problem (of size $N-1-(n+1)+1 = N-n-1$) of the problem of finding the optimal input sequence for the cost function ϕ_n (that is a problem of size $N-1-n+1 = N-n$).

This implies that the input sequence can be build step by step from the last stage. The problem of finding the optimal input $u_{N-1}^*(x_{N-1})$ for the cost function ϕ_{N-1} is a trivial problem of size 1. Once solved, the problem of finding the optimal input $u_{N-2}^*(x_{N-2})$ such that the sequence $\{u_{N-2}^*(x_{N-2}), u_{N-1}^*(x_{N-2})\}$ is optimal for the cost function ϕ_{N-2} is again a trivial problem of size 1. The procedure is iterated step by step till the initial stage, finding the input sequence $\{u_0^*(x_0), \dots, u_{N-1}^*(x_0)\}$ as function of x_0.

At the initial stage 0, the state x_0 is known, and then it is possible to compute the actual value of the input sequence.

In the following we explicitly compute the input sequence using the procedure described above. At each stage the minimum value of the cost function takes the form

$$V_n^*(x_n) = x_n' P_n x_n$$

where P_n is a symmetric positive semi-definite matrix.

At the last stage N, the cost function is a function only of x_N, and takes the form

$$V_N^*(x_N) = \phi_N = x_N' P_N x_N = x_N' P x_N,$$

where $P_N = P$ is a symmetric positive semi-definite matrix by hypothesis.

At the general stage n, we assume that the optimal value of the cost function at the stage $n+1$ is $V_{n+1}^*(x_{n+1}) = x_{n+1}' P_{n+1} x_{n+1}$ (with P_{n+1} symmetric and positive semi-definite), due to the optimal input sequence $\{u_{n+1}^*, \dots, u_{N-1}^*\}$. Using this input sequence, we have to minimize the cost function

$$\begin{aligned}\phi_n =& \varphi_n(x_n, u_n) + V_{n+1}^*(x_{n+1}) = \varphi_n(x_n, u_n) + V_{n+1}^*(Ax_n + Bu_n) = \\ =& \begin{bmatrix} x_n' & u_n' \end{bmatrix} \begin{bmatrix} Q & S' \\ S & R \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + (Ax_n + Bu_n)' P_{n+1} (Ax_n + Bu_n) = \\ =& \begin{bmatrix} x_n' & u_n' \end{bmatrix} \begin{bmatrix} Q + A'P_{n+1}A & S' + A'P_{n+1}B \\ S + B'P_{n+1}A & R + B'P_{n+1}B \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix}.\end{aligned} \tag{5.4}$$

only with respect to u_n.

We notice that $\phi_n(x_n)$ is a quadratic function of the input u_n, with an Hessian matrix $R + B'P_{n+1}B$ symmetric positive definite (since R is symmetric positive definite and P_{n+1} is symmetric positive semi-definite): this means that there is an unique minimizer. A necessary and sufficient condition to find the minimum is to set to zero the gradient of the cost function with respect to u_n,

$$\nabla_{u_n}(\varphi_n(x_n, u_n) + V_{n+1}^*(x_{n+1})) = 2(S + B'P_{n+1}A)x_n + 2(R + B'P_{n+1}B)u_n = 0$$

and so

$$u_n^*(x_n) = -(R + B'P_{n+1}B)^{-1}(S + B'P_{n+1}A)x_n = K_n x_n. \tag{5.5}$$

We thus found that the optimal input is a linear feedback from the state.

Using the expression 5.5 in 5.4, the optimal value of the cost function ϕ_n is

$$\begin{aligned}
&V_n^*(x_n) = \\
=&x_n' \left[Q + A'P_{n+1}A - (S + B'P_{n+1}A)'(R + B'P_{n+1}B)^{-1}(S + B'P_{n+1}A)\right] x_n = \\
=&x_n' \left[Q + A'P_{n+1}A - K_n'(R + B'P_{n+1}B)K_n)\right] x_n = \\
=&x_n' P_n x_n
\end{aligned}$$

where P_n is a symmetric positive semi-definite matrix. In fact it is symmetric since it is the sum of three symmetric matrices. And it is positive semi-definite: $V_n^*(x_n) = x_n'P_nx_n \geq 0$ for all possible x_n, since it is the minimum of the cost function $\phi_n \geq 0$, that is a sum of positive semi-definite quadratic forms.

The minimum value of the cost function $\phi_0 = \phi$ is obtained at the first stage $n = 0$, as

$$V_0^*(x_0) = x_0'P_0x_0.$$

Once the process is at the first stage $n = 0$, the value of x_0 is known, and used to compute the actual value of u_0^* with 5.5. In turns, the system dynamic equation gives $x_1 = Ax_0 + Bu_0$, and so it is possible to compute the actual value of the sequences of states and inputs iterating this procedure till the last stage $n = N$.

Extended Linear Quadratic Control Problem

We can repeat the procedure presented in the previous part also in the case of problem 2.1. Let us define φ_n the element of the cost function at time n,

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

and ϕ_n the cost function between n and N,

$$\phi_n = \sum_{i=n}^{N-1} \varphi_n + \frac{1}{2} x_N'Px_N + p'x_N + \pi.$$

The minimum of the cost function ϕ_n is defined again as

$$V_n^* = \min_{u_i, x_{i+1}} \phi_n.$$

The system dynamic equation is $x_{n+1} = A_nx_n + B_nu_n + b_n$.

In the case of 5.3, at each stage the minimum value of the cost function takes the form

$$V_n^*(x_n) = x_n'P_nx_n + p_n'x_n + \pi_n$$

where P_n is a symmetric positive semi-definite matrix.

At the final stage $n = N$, the cost function is function only of x_N, and takes the value

$$V_N^*(x_N) = \phi_N(x_N) = \frac{1}{2}x_N'P_Nx_N + p_N'x_N + \pi_N = \frac{1}{2}x_N'Px_N + p'x_N + \pi,$$

where the matrix $P_N = P$ is symmetric and positive semi-definite by hypothesis.

At the general stage n, assuming that the optimal value of the cost function at the stage $n+1$ is $V_{n+1}^*(x_{n+1}) = \frac{1}{2}x_{n+1}'P_{n+1}x_{n+1} + p_{n+1}'x_{n+1} + \pi_{n+1}$, with

P_{n+1} symmetric and positive semi-definite, we have to minimize with respect to u_n the cost function

$$\begin{aligned}
\phi_n(x_n, u_n) =& \varphi(x_n, u_n) + V^*_{n+1}(x_{n+1}) = \\
=& \varphi_n(x_n, u_n) + V^*_{n+1}(A_n x_n + B_n u_n + b_n) = \\
=& \frac{1}{2}\begin{bmatrix} x'_n & u'_n \end{bmatrix} \begin{bmatrix} Q_n & S'_n \\ S_n & R_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \begin{bmatrix} q'_n & s'_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \rho_n + \\
&+ \frac{1}{2}(A_n x_n + B_n u_n + b_n)' P_{n+1} (A_n x_n + B_n u_n + b_n) + \\
&+ p'_{n+1}(A_n x_n + B_n u_n + b_n) + \pi_n = \\
=& \frac{1}{2}\begin{bmatrix} x'_n & u'_n \end{bmatrix} \begin{bmatrix} Q_n + A'_n P_{n+1} A_n & S'_n + A'_n P_{n+1} B_n \\ S_n + B'_n P_{n+1} A_n & R_n + B'_n P_{n+1} B_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \\
&+ \begin{bmatrix} q'_n + (b'_n P_{n+1} + p'_{n+1}) A_n & s'_n + (b'_n P_{n+1} + p'_{n+1}) A_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \\
&+ \rho_n + \frac{1}{2} b'_n P_{n+1} b_n + p'_{n+1} b_n + \pi_n
\end{aligned}$$

We notice that this is a quadratic function of u_n, with the Hessian matrix $R_n + B'_n P_{n+1} B_n$ symmetric positive definite (since R_n is symmetric positive definite and P_{n+1} is symmetric positive semi-definite): this means that there is an unique minimizer. A necessary and sufficient condition to find the minimum is to set to zero the gradient of the cost function with respect to u_n,

$$\begin{aligned}
&\nabla_{u_n}(\varphi_n(x_n, u_n) + V^*_{n+1}(x_{n+1})) = \\
&= (S_n + B'_n P_{n+1} B_n) x_n + (R_n + B'_n P_{n+1} B_n) u_n + (s_n + B'_n (P_{n+1} b_n + p_{n+1})) = 0
\end{aligned}$$

and so

$$\begin{aligned}
u^*_n(x_n) =& -(R_n + B'_n P_{n+1} B_n)^{-1} (S_n + B'_n P_{n+1} A_n) x_n - \\
&- (R_n + B'_n P_{n+1} B_n)^{-1} (s_n + B'_n (P_{n+1} b_n + p_{n+1})) \\
=& K_n x_n + k_n.
\end{aligned}$$

We thus found that the optimal input is an affine feedback from the state.

The optimal value of the cost function $\phi_n(x_n, u_n)$ is

$$
\begin{aligned}
&V_n^*(x_n) = \phi_n^N(x_n, u_n^*(x_n)) = \\
=&\frac{1}{2}x_n' \big[Q_n + A_n'P_{n+1}A_n + \\
&+(S_n' + A_n'P_{n+1}B_n)K_n + K_n'(S_n + B_n'P_{n+1}A_n) + K_n'(R_n + B_n'P_{n+1}B_n)K_n\big]\, x_n + \\
&+ [q_n + A_n'(P_{n+1}b_n + p_{n+1}) + K_n'(s_n + B_n'(P_{n+1}b_n + p_{n+1})) + \\
&\qquad +(S_n + A_n'P_{n+1}B_n)k_n + k_n'(R_n + B_n'P_{n+1}B_B)K_n\big] + \\
&+ \bigg[\frac{1}{2}k_n'(R_n + B_n'P_{n+1}B_n)k_n + (s_n + B_n'(P_{n+1}b_n + p_{n+1}))'k_n + \rho + \\
&+\frac{1}{2}b_n'P'_{n+1}b_n + p_{n+1}'b_n + \pi_n\bigg] = \\
=&\frac{1}{2}x_n' \big[Q_n + A_n'P_{n+1}A_n - \\
&-\ (S_n + B_n'P_{n+1}A_n)'(R_n + B_n'P_{n+1}B_n)^{-1}(S_n + B_n'P_{n+1}A_n)\big]\, x_n + \\
&+ [q_n + A_n'(P_{n+1}b_n + p_{n+1}) - \\
&-(S_n + B_n'P_{n+1}A_n)'(R_n + B_n'P_{n+1}B_n)^{-1}(s_n + B_n'(P_{n+1}b_n + p_{n+1}))\big]'\, x_n + \\
&+ \bigg[-\frac{1}{2}(s_n + B_n'(P_{n+1}b_n + p_{n+1}))'(R_n + B_n'P_{n+1}B_n)^{-1}(s_n + B_n'(P_{n+1}b_n + p_{n+1})) + \\
&\qquad +\rho_n + \frac{1}{2}b_n'P_{n+1}b_n + p_{n+1}'b_n + \pi_n\bigg] \\
=&\frac{1}{2}x_n' \big[Q_n + A_n'P_{n+1}A_n - K_n'(R_n + B_n'P_{n+1}B_n)K_n\big]\, x_n + \\
&+ \big[q_n + A_n'(P_{n+1}b_n + p_{n+1}) - K_n'(R_n + B_n'P_{n+1}B_n)k_n\big]'\, x_n + \\
&+ \bigg[-\frac{1}{2}k_n'(R_n + B_n'P_{n+1}B_n)k_n + \rho_n + \frac{1}{2}b_n'P_{n+1}b_n + p_{n+1}'b_n + \pi_n\bigg] \\
=&x_n'P_nx_n + p_n'x_n + \pi_n
\end{aligned}
$$

where P_n is a symmetric positive semi-definite matrix. In fact it is symmetric since it is the sum of three symmetric matrices. And it is positive semi-definite, since its expression is the same (a part form the time variant matrices) as in the standard linear quadratic control problem case.

The minimum of the cost function $\phi_0^N = \phi$ is obtained at the first stage $n = 0$, as

$$V_0^*(x_0) = x_0'P_0x_0 + p_0'x_0 + \pi_0.$$

5.1.3 Derivation from the KKT System

It is possible to derive the Riccati recursion methods directly from the expression of the KKT system 2.7. The procedure is the same for both problems 2.2 and 2.1. This derivation technique can be found also in [RWR98].

We report the expression of the KKT system for 2.1 (in the case $N = 3$)

$$\begin{bmatrix} R_0 & B_0' & & & & & & & \\ B_0 & & -I & & & & & & \\ & -I & Q_1 & S_1' & A_1' & & & & \\ & & S_1 & R_1 & B_1' & & & & \\ & & A_1 & B_1 & & -I & & & \\ & & & & -I & Q_2 & S_2' & A_2' & \\ & & & & & S_2 & R_2 & B_2' & \\ & & & & & A_2 & B_2 & & -I \\ & & & & & & & -I & P \end{bmatrix} \begin{bmatrix} u_0 \\ \pi_1 \\ x_1 \\ u_1 \\ \pi_2 \\ x_2 \\ u_2 \\ \pi_3 \\ x_3 \end{bmatrix} = \begin{bmatrix} -S_0x_0 - s_0 \\ -A_0x_0 - b_0 \\ -q_1 \\ -s_1 \\ -b_1 \\ -q_2 \\ -s_2 \\ -b_2 \\ -p \end{bmatrix}$$

The derivation is based on the fact that, the last block of 4 rows

$$\begin{bmatrix} -I & Q_{N-1} & S_{N-1}' & A_{N-1}' & \\ & S_{N-1} & R_{N-1} & B_{N-1}' & \\ & A_{N-1} & B_{N-1} & & -I \\ & & & -I & P \end{bmatrix} \begin{bmatrix} \pi_{N-1} \\ x_{N-1} \\ u_{N-1} \\ \pi_N \\ x_N \end{bmatrix} = \begin{bmatrix} -q_{N-1} \\ -s_{N-1} \\ -b_{N-1} \\ -p \end{bmatrix} \tag{5.6}$$

(that is a linear system with 4 equations and 5 variables) can be reduced to the equivalent equation in two variables

$$-\pi_{N-1} + P_{N-1}x_{N-1} = -p_{N-1} \tag{5.7}$$

where P_{N-1} and p_{N-1} are matrices of a particular form. This equation has the exact same form of the last row in 5.6: we can thus substitute in the KKT matrix the block of 4 rows 5.6 with the single row 5.7, and so on. At the end there is a system of 3 equations in 3 variables, that can be immediately solved.

Here we show the computations in details. At the general step, we have that the last 4 rows in the modified KKT matrix are

$$\begin{bmatrix} -I & Q_n & S_n' & A_n' & \\ & S_n & R_n & B_n' & \\ & A_n & B_n & & -I \\ & & & -I & P_{n+1} \end{bmatrix} \begin{bmatrix} \pi_n \\ x_n \\ u_n \\ \pi_{n+1} \\ x_{n+1} \end{bmatrix} = \begin{bmatrix} -q_n \\ -s_n \\ -b_n \\ -p_{n+1} \end{bmatrix}. \tag{5.8}$$

We can eliminate x_{n+1} using the third equation in 5.8 $x_{n+1} = A_nx_n+B_nu_n+b_n$, obtaining

$$\begin{bmatrix} -I & Q_n & S_n' & A_n' \\ & S_n & R_n & B_n' \\ & P_{n+1}A_n & P_{n+1}B_n & -I \end{bmatrix} \begin{bmatrix} \pi_n \\ x_n \\ u_n \\ \pi_{n+1} \end{bmatrix} = \begin{bmatrix} -q_n \\ -s_n \\ -(P_{n+1}b_n + p_{n+1}) \end{bmatrix}. \tag{5.9}$$

We eliminate also π_{n+1} using third the equation in 5.9 $\pi_{n+1} = P_{n+1}(A_nx_n + B_nu_n + b_n) + p_{n+1}$, obtaining

$$\begin{bmatrix} -I & Q_n + A_n'P_{n+1}A_n & S_n' + A_n'P_{n+1}B_n \\ & S_n + B_n'P_{n+1}A_n & R_n + B_n'P_{n+1}B_n \end{bmatrix} \begin{bmatrix} \pi_n \\ x_n \\ u_n \end{bmatrix} =$$

$$= \begin{bmatrix} -(q_n + A_n'(P_{n+1}b_n + p_{n+1})) \\ -(s_n + B_n'(P_{n+1}b_n + p_{n+1})) \end{bmatrix} \tag{5.10}$$

Finally we can eliminate u_n using the second equation in 5.10

$$(S_n + B_n' P_{n+1} A_n) x_n + (R_n + B_n' P_{n+1} B_n) u_n = -(s_n + B_n'(P_{n+1} b_n + p_{n+1})),$$

obtaining

$$-\pi_n + P_n x_n = -p_n$$

where

$$P_n = Q_n + A_n' P_{n+1} A_n - (S_n' + A_n' P_{n+1} B_n)(R_n + B_n' P_{n+1} B_n)^{-1}(S_n + B_n' P_{n+1} A_n) \tag{5.11}$$

and

$$\begin{aligned} p_n = q_n + A_n'(P_{n+1} b_n + p_{n+1}) - \\ -(S_n' + A_n' P_{n+1} B_n)(R_n + B_n' P_{n+1} B_n)^{-1}(s_n + B_n'(P_{n+1} b_n + p_{n+1})). \end{aligned}$$

Equation 5.11 is again the Riccati recursion.

In the end we have to solve the system

$$\begin{bmatrix} R_0 & B_0' & \\ B_0 & & -I \\ & -I & P_1 \end{bmatrix} \begin{bmatrix} u_0 \\ \pi_1 \\ x_1 \end{bmatrix} = \begin{bmatrix} -(S_0 x_0 + s_0) \\ -(A_0 x_0 + b_0) \\ -p_1 \end{bmatrix}$$

giving

$$\begin{aligned} u_0 &= -(R_0 + B_0' P_1 B_0)^{-1}(S_0 x_0 + s_0 + B_0'(p_1 + P_1(A_0 x_0 + b_0))) = \\ &= -(R_0 + B_0' P_1 B_0)^{-1}(S_0 + B_0' P_1 A_0) x_0 - (R_0 + B_0' P_1 B_0)^{-1}(s_0 + B_0'(P_1 b_0 + p_1)) \end{aligned}$$

It is possible to build the entire sequences x_n and u_n using in an switching way one after the other the third equation of 5.8:

$$x_{n+1} = A_n x_n + B_n u_n + b_n,$$

(that is the equation describing the dynamic of the system), and the second equation of 5.10:

$$\begin{aligned} u_n = & \quad -(R_n + B_n' P_{n+1} B_n)^{-1}(S_n + B_n' P_{n+1} A_n) x_n - \\ & -(R_n + B_n' P_{n+1} B_n)^{-1}(s_n + B_n'(P_{n+1} b_n + p_{n+1})) \\ = & \qquad K_n x_n + k_n \end{aligned}$$

(that characterizes the optimal control law as an affine feedback from the state).

The sequence π_n can be obtained from the value of π_{n+1} and the first equation in 5.9,

$$\pi_n = q_n + Q_n x_n + S_n' u_n + A_n' \pi_{n+1},$$

with initial value

$$\pi_N = P x_N + p.$$

5.1.4 Properties of the Sequence P_n

In this section we present a few properties of the sequence P_n computed using the Riccati recursion

$$P_n = Q_n + A'_n P_{n+1} A_n - (S_n + B'_n P_{n+1} A_n)'(R_n + B'_n P_{n+1} B_n)^{-1}(S_n + B'_n P_{n+1} A_n)$$

in the time variant form: the sequence exists, and P_n is a symmetric and positive semi-definite matrix, in the hypothesis that the matrices

$$P \quad \text{and} \quad \begin{bmatrix} Q_n & S'_n \\ S_n & R_n \end{bmatrix}$$

are symmetric and positive semi-definite. Furthermore, if they are also positive definite, then also P_n is positive definite as well.

The proof can be found by induction. At the first step, the matrix $P_N = P$ exists and is symmetric positive semi-definite by hypothesis.

At the general step, provided that P_{n+1} exists and is symmetric and positive semi-definite, P_n exists, since the matrix $(R_n + B'_n P_{n+1} B_n)$ is positive definite and thus invertible (R_n is positive definite by hypothesis and P_{n+1} is positive semi-definite from the previous step).

Furthermore, P_n is symmetric, since it is the sum of three symmetric matrices, provided that P_{n+1} is symmetric from the previous step.

Finally, P_n is positive semi-definite. In order to prove this, we notice that the expression of the matrix P_n in the Riccati recursion

$$P_n = (Q_n + A'_n P_{n+1} A_n) - (S_n + B'_n P_{n+1} A_n)'(R_n + B'_n P_{n+1} B_n)^{-1}(S_n + B'_n P_{n+1} A_n)$$

is the Schur complement of the matrix $(R_n + B'_n P_{n+1} B_n)$ in the matrix

$$\begin{bmatrix} Q_n + A'_n P_{n+1} A_n & (S_n + B'_n P_{n+1} A_n)' \\ S_n + B'_n P_{n+1} A_n & R_n + B'_n P_{n+1} B_n \end{bmatrix} =$$
$$= \begin{bmatrix} Q_n & S'_n \\ S_n & R_n \end{bmatrix} + \begin{bmatrix} A'_n P_{n+1} A_n & A'_n P_{n+1} B_n \\ B'_n P_{n+1} A_n & B'_n P_{n+1} B_n \end{bmatrix} =$$
$$= \begin{bmatrix} Q_n & S'_n \\ S_n & R_n \end{bmatrix} + \begin{bmatrix} A'_n \\ B'_n \end{bmatrix} P_{n+1} \begin{bmatrix} A_n & B_n \end{bmatrix}$$

that is positive semi-definite, since it is the sum of two matrices positive semi-definite. This implies that also the matrix P_n is positive semi-definite.

In fact, given the symmetric positive semi-definite matrix M represented in 2×2 block form, where the Z block is positive definite (and then invertible), we can decompose it using a sort of block UDU' decomposition,

$$M = \begin{bmatrix} X & Y' \\ Y & Z \end{bmatrix} = \begin{bmatrix} I & Y'Z^{-1} \\ 0 & I \end{bmatrix} \begin{bmatrix} X - Y'Z^{-1}Y & 0 \\ Y & Z \end{bmatrix} =$$
$$= \begin{bmatrix} I & Y'Z^{-1} \\ 0 & I \end{bmatrix} \begin{bmatrix} X - Y'Z^{-1}Y & 0 \\ 0 & Z \end{bmatrix} \begin{bmatrix} I & 0 \\ Z^{-1}Y & I \end{bmatrix}.$$

We notice then that the matrices on the left and the right of the last expression are invertible, and the it is possible to isolate the matrix in the middle

$$\begin{bmatrix} X - Y'Z^{-1}Y & 0 \\ 0 & Z \end{bmatrix} = \begin{bmatrix} I & -Y'Z^{-1} \\ 0 & I \end{bmatrix} \begin{bmatrix} X & Y' \\ Y & Z \end{bmatrix} \begin{bmatrix} I & 0 \\ -Z^{-1}Y & I \end{bmatrix}.$$

Then, for each vector v of length equal to the size of the X matrix, we have that

$$v'(X - Y'Z^{-1}Y)v = \begin{bmatrix} v' & 0 \end{bmatrix} \begin{bmatrix} X - Y'Z^{-1}Y & 0 \\ 0 & Z \end{bmatrix} \begin{bmatrix} v \\ 0 \end{bmatrix} =$$
$$= \begin{bmatrix} v' & 0 \end{bmatrix} \begin{bmatrix} I & -Y'Z^{-1} \\ 0 & I \end{bmatrix} \begin{bmatrix} X & Y' \\ Y & Z \end{bmatrix} \begin{bmatrix} I & 0 \\ -Z^{-1}Y & I \end{bmatrix} \begin{bmatrix} v \\ 0 \end{bmatrix} \geq 0$$

since the M matrix is positive semi-definite by hypothesis. This implies that also the Schur complement of Z in M, $Z - Y'Z^{-1}Y$, is positive semi-definite.

Finally, assuming the further hypothesis that the matrices

$$P \quad \text{and} \quad \begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix}$$

are positive definite, it is possible to prove that also P_n is positive definite at each step. The proof is again by induction. At the first step it is true that $P_N = P$ is positive definite by hypothesis.

At the general step, the expression of the matrix P_n in the Riccati recursion

$$P_n = (Q_n + A_n'P_{n+1}A_n) - (S_n + B_n'P_{n+1}A_n)'(R_n + B_n'P_{n+1}B_n)^{-1}(S_n + B_n'P_{n+1}A_n)$$

is the Schur complement of the matrix $(R_n + B_n'P_{n+1}B_n)$ in the matrix

$$\begin{bmatrix} Q_n + A_n'P_{n+1}A_n & (S_n + B_n'P_{n+1}A_n)' \\ S_n + B_n'P_{n+1}A_n & R_n + B_n'P_{n+1}B_n \end{bmatrix} =$$
$$= \begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix} + \begin{bmatrix} A_n'P_{n+1}A_n & A_n'P_{n+1}B_n \\ B_n'P_{n+1}A_n & B_n'P_{n+1}B_n \end{bmatrix} =$$
$$= \begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix} + \begin{bmatrix} A_n' \\ B_n' \end{bmatrix} P_{n+1} \begin{bmatrix} A_n & B_n \end{bmatrix}$$

that this time is positive definite, since it is the sum of a matrix positive definite and a matrix at least positive semi-definite. This implies that also the matrix P_n is positive definite.

In fact, given a generic symmetric positive definite matrix M represented in 2×2 block form, we can decompose it using again a sort of block UDU' decomposition,

$$M = \begin{bmatrix} X & Y' \\ Y & Z \end{bmatrix} = \begin{bmatrix} I & Y'Z^{-1} \\ 0 & I \end{bmatrix} \begin{bmatrix} X - Y'Z^{-1}Y & 0 \\ 0 & Z \end{bmatrix} \begin{bmatrix} I & 0 \\ Z^{-1}Y & I \end{bmatrix}$$

since the matrix Z is positive definite and then invertible.

We can thus write the inverse of the matrix M as

$$M^{-1} = \begin{bmatrix} X & Y' \\ Y & Z \end{bmatrix}^{-1} = \begin{bmatrix} I & 0 \\ -Z^{-1}Y & I \end{bmatrix} \begin{bmatrix} (X - Y'Z^{-1}Y)^{-1} & 0 \\ 0 & Z^{-1} \end{bmatrix} \begin{bmatrix} I & -Y'Z^{-1} \\ 0 & I \end{bmatrix} =$$
$$= \begin{bmatrix} (X - Y'Z^{-1}Y)^{-1} & 0 \\ -Z^{-1}Y(X - Y'Z^{-1}Y)^{-1} & Z^{-1} \end{bmatrix} \begin{bmatrix} I & -Y'Z^{-1} \\ 0 & I \end{bmatrix} =$$
$$= \begin{bmatrix} (X - Y'Z^{-1}Y)^{-1} & -(X - Y'Z^{-1}Y)^{-1}Y'Z^{-1} \\ -Z^{-1}Y(X - Y'Z^{-1}Y)^{-1} & Z^{-1} + Z^{-1}Y(X - Y'Z^{-1}Y)^{-1}Y'Z^{-1} \end{bmatrix}$$

The matrix M^{-1} is positive definite because it is the inverse of the positive definite matrix M. This implies that also the sub-matrix on the top left corner $(X - Y'Z^{-1}Y)^{-1}$ is positive definite, and its inverse $X - Y'Z^{-1}Y$ (the Schur complement of the matrix Z in M) is positive definite too.

5.2 Implementation

In many applications, and in our test problem, n_x is (much) larger than n_u: we optimize our implementation for this case. Provided that, the most expensive part of the Riccati recursion is the computation of the matrix $A'P_nA$. A trivial implementation of the computation of $A'P_nA$ requires roughly $4n_x^3$ floating-point operations. Anyway it is possible to exploit the special structure of this expression to do better: P_n is a symmetric matrix (and in general just positive semi-definite), and it is multiplied on the right by A and on the left by the transpose A'. Exploiting this we can write an algorithm to compute of the expression $A'P_nA$ with an asymptotic cost of $3n_x^3$ floating-point operations. Making the further assumption that P_n is positive definite (as often in practice), we can write an algorithm with asymptotic cost of $\frac{7}{3}n_x^3$.

We wrote three implementations of the Riccati recursion method:

- the general form of the algorithm, using the general matrix-matrix multiplication routine for all operations;
- a form exploiting the symmetry of the P_n matrix;
- a form requiring P_n to be symmetric positive definite, in the two versions computing the upper or lower Cholesky factor.

5.2.1 Genaral form

In this first subsection we consider the general form of the algorithm, and use only the general matrix-matrix multiplication routine `dgemm`[1]. The asymptotic complexity of the algorithm is $N(4n_x^3 + 6n_x^2n_u + 3n_xn_u^2 + \frac{1}{3}n_u^3)$ floating-point operations, worst than the specialized forms of the algorithm.

Anyway, this algorithm can show some advantages for small systems, since the implementation of the routine `dgemm` is usually more efficient, and thus for small matrices it is usually faster than specialized routines like `dtrmm`[2] or `dsyrk`[3].

In the computation of the product $A'P_nA$, that is where the most part of the computational time is spent (at least for large systems), we are exploiting the symmetry of P_n to use the routine `dgemm` where it is more efficient: in fact, the best performance is obtained if, in the product, the first factor is transposed and the second is not. Since P_n is symmetric $P_n = P_n'$, and thus we have that

$$A'P_nA = A'P_n'A = A'(P_n'A)$$

and thus we can use the routine `dgemm` both time in the more efficient situation.

[1]See appendix B.1.1.
[2]See appendix B.1.2.
[3]See appendix B.1.3.

It is also possible to obtain a further slightly improvement, compared to the algorithm stated in the previous section, in the computation of the term $K_n'R_{e,n}K_n$ in the P_n update formula. In fact the matrix $R_{e,n}$ is symmetric positive definite, and thus it is possible to compute its Cholesky factorization,

$$R_{e,n} = \Lambda_n\Lambda_n'.$$

We thus have

$$\begin{aligned}
K_n'R_{e,n}K_n &= (S_n + B_n'P_{n+1}A_n)'R_{e,n}^{-1}R_{e,n}R_{e,n}^{-1}(S_n + B_n'P_{n+1}A_n) = \\
&= (S_n + B_n'P_{n+1}A_n)'R_{e,n}^{-1}(S_n + B_n'P_{n+1}A_n) \\
&= (S_n + B_n'P_{n+1}A_n)'(\Lambda_n^{-1})'\Lambda_n^{-1}(S_n + B_n'P_{n+1}A_n) \\
&= (\Lambda_n^{-1}(S_n + B_n'P_{n+1}A_n))'\Lambda_n^{-1}(S_n + B_n'P_{n+1}A_n) = L_n'L_n
\end{aligned}$$

where L_n is defined as

$$L_n \doteq \Lambda_n^{-1}(S_n + B_n'P_{n+1}A_n)$$

In this way a multiplication and a linear triangular system solution are avoided, saving $2n_xn_u^2$ and $n_xn_u^2$ floating-points operations.

It is also possible to obtain similar improvement in the computation of the term $K_n'R_{e,n}k_n$ in the p_n update formula, but this time the savings would be in the quadratic terms, and thus in general non influential. We have

$$K_n'R_{e,n}k_n = L_n'l_n$$

where l_n is defined as

$$l_n \doteq \Lambda_n^{-1}(r_n + B_n'(P_{n+1}b_n + p_{n+1})).$$

Also the computation of the input u_n is influenced, becoming

$$u_n = K_nx_n + k_n = (-\Lambda_n')^{-1}L_nx_n + (-\Lambda_n')^{-1}l_n = (-\Lambda_n')^{-1}(L_nx_n + l_n),$$

where again the difference in the computational time affects a quadratic term. Regarding the space needs, also the sequence of the $n_u \times n_u$ matrices Λ_i needs to be saved, in addition to the sequences of the $n_u \times n_x$ matrices L_i (of the same size of k_i) and the $n_u \times 1$ vectors l_i (of the same size of k_i).

In the following algorithm 3 we compute the complexity per iteration up to the quadratic terms, that is

$$\left(4n_x^3 + 6n_x^2n_u + 3n_xn_u + \frac{1}{3}n_u^3\right) + \left(5n_x^2 + 7n_xn_u^2 + \frac{5}{2}n_u^2\right)$$

floating-point operations. We denote with a subscript the name of the used BLAS or LAPACK routines.

5.2.2 Form exploiting the symmetry of P_n

The algorithm presented in this part exploits the symmetry of the matrices of the sequence P_n to obtain an algorithm with asymptotic complexity of $N(3n_x^3 + 5n_x^2n_u + 3n_xn_u^2 + \frac{1}{3}n_u^3)$ floating-point operations.

Algorithm 3 Riccati recursion method for the solution of 2.1, general form

```
Require: $(x_0, \{Q_n\}, \{S_n\}, \{R_n\}, \{q_n\}, \{s_n\}, \{A_n\}, \{B_n\}, \{b_n\}, P, p)$
  $P_{n+1} \leftarrow P$
  $p_{n+1} \leftarrow p$
  for $n = N-1 \to 0$ do
    $R_{e,n} \leftarrow R_n + B_n' \cdot_{\text{dgemm}} (P_{n+1}' \cdot_{\text{dgemm}} B_n)$   $\triangleright$ $2n_x^2 n_u + 2n_x n_u^2 - n_x n_u$ flops
    $\Lambda_n \leftarrow_{\text{dpotrf}} chol(R_{e,n})$   $\triangleright$ $\frac{1}{3}n_u^3 + \frac{1}{2}x_u^2$ flops
    $L_n \leftarrow_{\text{dtrsm}} \Lambda_n^{-1}(S_n + (P_{n+1}' B_n)' \cdot_{\text{dgemm}} A_n)$   $\triangleright$ $2n_x^2 n_u + n_x n_u^2$ flops
    $P_n \leftarrow Q_n + A_n' \cdot_{\text{dgemm}} (P_{n+1}' \cdot_{\text{dgemm}} A_n) - L_n' \cdot_{\text{dgemm}} L_n$
                                            $\triangleright$ $4n_x^3 + 2n_x^2 n_u - n_x^2$ flops
    $l_n \leftarrow_{\text{dtrsv}} \Lambda_n^{-1}(r_n + B_n' \cdot_{\text{dgemv}} (P_{n+1} \cdot_{\text{dgemv}} b_n + p_{n+1}))$
                                            $\triangleright$ $2n_x^2 + 2n_x n_u + n_u^2$ flops
    $p_n \leftarrow q_n + A_n' \cdot_{\text{dgemv}} (P_{n+1} b_n + p_{n+1}) - L_n' \cdot_{\text{dgemv}} l_n$   $\triangleright$ $2n_x^2 + 2n_x n_u$ flops
  end for

  for $n = 0 \to N-1$ do
    $u_n \leftarrow_{\text{dtrsv}} -(\Lambda_n')^{-1}(L_n \cdot_{\text{dgemv}} x_n + l_n)$   $\triangleright$ $2n_x n_u + n_u^2$ flops
    $x_{n+1} \leftarrow A_n \cdot_{\text{dgemv}} x_n + B_n \cdot_{\text{dgemv}} u_n + b_n$   $\triangleright$ $2n_x^2 + 2n_x n_u$ flops
  end for
  return $(\{x_n\}, \{u_n\})$
```

It is possible to write the symmetric matrix P_n as the sum of a lower triangular matrix and an upper triangular matrix in such a way that one is the transposed of the other: $P_n = \Pi + \Pi'$. The cost to obtain the lower triangular matrix Π is $\frac{n_x(n_x-1)}{2}$ copies of floating point numbers and n_x floating-point divisions. Substituting this expression into $A'P_nA$ we have

$$A'P_nA = A'(\Pi + \Pi')A = A'\Pi A + A'\Pi' A = (A'\Pi' A)' + A'\Pi' A$$

and thus also the matrix $A'P_nA$ can be written as the sum of a matrix and its transposed. The advantage of this procedure is that it is possible to compute $\Pi' \cdot A$ in n_x^3 floating-point operations (instead of roughly $2n_x^3$) using the specialized BLAS routine `dtrmm`. The choice of Π' instead of Π has the advantage of exploit the used column-major order of data in memory.

The following multiplication $A' \cdot (\Pi A)$ has no structure, and so requires $2n_x^3$ floating-point operations; again the fact that the first factor is transposed exploits the data order in memory. The final sum requires $\frac{n_x(n_x+1)}{2}$ floating-point sums and $\frac{n_x(n_x-1)}{2}$ copies of floating-point numbers.

The asymptotic complexity in the computation of the term $A'P_nA$ is thus of $3n_x^3$ instead of $4n_x^3$ floating-point operations.

All the remaining parts of the algorithm are as in the previous version, with the only exception of the use of the specialized routine `dsyrk` (approximately requiring the half of the floating-point operations compared to `dgemm`) in the computation of the product $L_n'L_n$.

In the following algorithm 4 we compute the complexity per iteration up to the quadratic terms, that is

$$\left(3n_x^3 + 5n_x^2 n_u + 3n_x n_u^2 + \frac{1}{3}n_u^3\right) + \left(7n_x^2 + 8n_x n_u + \frac{5}{2}n_u^2\right)$$

floating-point operations.

Algorithm 4 Riccati recursion method for the solution of 2.1, form exploiting the symmetry of the matrix P_n

```
Require: $(x_0, \{Q_n\}, \{S_n\}, \{R_n\}, \{q_n\}, \{s_n\}, \{A_n\}, \{B_n\}, \{b_n\}, P, p)$
  $P_{n+1} \leftarrow P$
  $p_{n+1} \leftarrow p$
  for $n = N-1 \to 0$ do
      $R_{e,n} \leftarrow R_n + B'_n \cdot_{\text{dgemm}} (P'_{n+1} \cdot_{\text{dgemm}} B_n)$   $\triangleright$ $2n_x^2 n_n + 2n_x n_u^2 - n_x n_u$ flops
      $\Lambda_n \leftarrow_{\text{dpotrf}} chol(R_{e,n})$   $\triangleright$ $\frac{1}{3}n_u^3 + \frac{1}{2}n_u^2$ flops
      $L_n \leftarrow_{\text{dtrsm}} \Lambda_n^{-1}(S_n + (P'_{n+1}B'_n)' \cdot_{\text{dgemm}} A_n)$   $\triangleright$ $2n_x^2 n_u + n_x n_u^2$ flops
      Compute $\Pi_n$
      Compute $A'_n \cdot_{\text{dgemm}} (\Pi'_n \cdot_{\text{dtrmm}} A_n)$   $\triangleright$ $3n_x^3 - n_x^2$ flops
      $P_n \leftarrow Q_n + (A'_n(\Pi'_n A_n))' + A'_n(\Pi'_n A_n) - L'_n \cdot_{\text{dsyrk}} L_n$
                                                  $\triangleright$ $n_x^2 n_u + 2n_x^2 + n_x n_u$ flops
      $l_n \leftarrow_{\text{dtrsv}} \Lambda_n^{-1}(r_n + B'_n \cdot_{\text{dgemv}} (P_{n+1} \cdot_{\text{dgemv}} b_n + p_{n+1}))$
                                                  $\triangleright$ $2n_x^2 + 2n_x n_u + n_u^2$ flops
      $p_n \leftarrow q_n + A'_n \cdot_{\text{dgemv}} (P_{n+1}b_n + p_{n+1}) - L'_n \cdot_{\text{dgemv}} l_n$   $\triangleright$ $2n_x^2 + 2n_x n_u$ flops
  end for

  for $n = 0 \to N-1$ do
      $u_n \leftarrow_{\text{dtrsv}} -(\Lambda'_n)^{-1}(L_n \cdot_{\text{dgemv}} x_n + l_n)$   $\triangleright$ $2n_x n_u + n_u^2$ flops
      $x_{n+1} \leftarrow A_n \cdot_{\text{dgemv}} x_n + B_n \cdot_{\text{dgemv}} u_n + b_n$   $\triangleright$ $2n_x^2 + 2n_x n_u$ flops
  end for
  return $(\{x_n\}, \{u_n\})$
```

5.2.3 Form requiring P_n to be positive definite

Assuming that P_n is symmetric positive definite (and thus invertible), we can obtain an algorithm with asymptotic complexity $N(\frac{7}{3}n_x^3+4n_x^2n_u+2n_xn_u^2+\frac{1}{3}n_u^3)$. As already seen, a sufficient condition for this is that $Q_n, n = 0, ..., N-1$ and P are positive definite.

As already said, the most expensive part of the Riccati recursion algorithm is the computation of the term $A'P_nA$ at each iteration. Exploiting the fact that P_n is symmetric positive definite, we can factorize it using the Cholesky factorization, obtaining the upper $P_n = U'U$ or lower $P_n = LL'$ factor; the cost is $\frac{1}{3}n_x^3 + \frac{1}{2}n_x^2 + \frac{1}{6}n_x$. Then the term $A'P_nA$ is computed as

$$A'P_nA = A'U'UA = (UA)'UA \quad \text{or} \quad A'P_nA = A'LL'A = (L'A)'(L'A).$$

The advantage is that U and L are respectively upper and lower triangular, and the computation of the product UA or $L'A$ requires only n_x^3 floating-point operations using the specialized routine `dtrmm`; and the product of the transpose of the matrix UA or $L'A$ by the matrix UA or $L'A$ itself requires only $n_x^3 + n_x^2$ using the specialized routine `dsyrk`. In total the algorithm requires $\frac{7}{3}n_x^3+\frac{3}{2}n_x^2+\frac{1}{6}n_x$ floating point operations.

The use of the upper U or lower L factor in the algorithm influences the performance. In fact, even if the cost is the same as number of floating-point operations, the computation of U is faster than the computation of L because of the data order in memory. On the other hand, the computation of the product UA is slower than the computation of the product $L'A$, again because of the data order in memory. Which of the two version is the more efficient is really case sensitive, and should be tested for the specific BLAS and LAPACK implementation, as well as for the problem and problem size.

In the following algorithm 5 we compute the cost per iteration up to the quadratic terms, that is

$$\left(\frac{7}{3}n_x^3 + 4n_x^2 n_u + 2n_x n_u^2 + \frac{1}{3}n_u^3\right) + \left(\frac{15}{2}n_x^2 + 10 n_x n_u + \frac{5}{2}n_u^2\right)$$

floating-point operations.

Algorithm 5 Riccati recursion method for the solution of 2.1, form requiring P_n to be symmetric positive definite, version computing the upper Cholesky factor of P_n

```
Require: $(x_0, \{Q_n\}, \{S_n\}, \{R_n\}, \{q_n\}, \{s_n\}, \{A_n\}, \{B_n\}, \{b_n\}, P, p)$
  $P_{n+1} \leftarrow P$
  $p_{n+1} \leftarrow P$
  for $n = N-1 \to 0$ do
    $U_n \leftarrow_{\text{dpotrf}} chol(P_{n+1})$    $\triangleright\ \frac{1}{3}n_x^3 + \frac{1}{2}n_x^2$ flops
    $R_{e,n} \leftarrow R_n + (U_n \cdot_{\text{dtrmm}} B_n)' \cdot_{\text{dsyrk}} (U_n B_n)$    $\triangleright\ n_x^2 n_u + n_x n_u^2 + n_x n_u$ flops
    $\Lambda_n \leftarrow_{\text{dpotrf}} chol(R_{e,n})$    $\triangleright\ \frac{1}{3}n_u^3 + \frac{1}{2}n_u^2$ flops
    $L_n \leftarrow_{\text{dtrsm}} \Lambda_n^{-1}(S_n + (U_n B_n)' \cdot_{\text{dgemm}} (U_n \cdot_{\text{dtrmm}} A_n))$
        $\triangleright\ n_x^3 + 2n_x^2 n_u + n_x n_u^2$ flops
    $P_n \leftarrow Q_n + (U_n A_n)' \cdot_{\text{dsyrk}} (U_n A_n) - L_n' \cdot_{\text{dsyrk}} L_n$
        $\triangleright\ n_x^3 + n_x^2 n_u + n_x^2 + n_x n_u$ flops
    $l_n \leftarrow_{\text{dtrsv}} \Lambda_n^{-1}(r_n + B_n' \cdot_{\text{dgemv}} (P_{n+1} \cdot_{\text{dsymv}} b_n + p_{n+1}))$
        $\triangleright\ 2n_x^2 + 2n_x n_u + n_u^2$ flops
    $p_n \leftarrow q_n + A_n' \cdot_{\text{dgemv}} (P_{n+1} b_n + p_{n+1}) - L_n' \cdot_{\text{dgemv}} l_n$    $\triangleright\ 2n_x^2 + 2n_x n_u$ flops
  end for

  for $n = 0 \to N-1$ do
    $u_n \leftarrow_{\text{dtrsv}} -(\Lambda_n')^{-1}(L_n \cdot_{\text{dgemv}} x_n + l_n)$    $\triangleright\ 2n_x n_u + n_u^2$ flops
    $x_{n+1} \leftarrow A_n \cdot_{\text{dgemv}} x_n + B_n \cdot_{\text{dgemv}} u_n + b_n$    $\triangleright\ 2n_x^2 + 2n_x n_u$ flops
  end for
  return $(\{x_n\}, \{u_n\})$
```

5.3 Performance analysis

5.3.1 Cost of sub-routines

We can use the profiler tool `gprof`[4] to analyze the cost of the different routines, by compiling the code with the optimization flag `-pg`. Also the BLAS and LAPACK libraries used in this part are compiled with the optimization flag `-pg`.

In table 5.1 there is the cost in percentage of the BLAS and LAPACK routines, taken from the flat profile produced by `pgrof`.

[4]See appendix A.

	algorithm 3	algorithm 4	algorithm 5
routine	percentage	percentage	percentage
dgemm	98.29	66.09	5.99
dtrmm	unused	29.40	42.16
dsyrk	unused	1.06	43.57
dtrsm	0.16	0.25	4.43
dgemv	0.64	1.09	1.23
dpotrf	0.00	0.00	0.00
others	0.91	2.11	2.62

Table 5.1: Cost of BLAS and LAPACK routines, using as test problem the mass-spring problem 7.2, with $n_x = 256$, $n_u = 8$ and $N = 10$.

In the case of algorithm 3, almost all the computational time is due to routine `dgemm`. In the case of algorithm 4, `dgemm` is the most expensive routine, but also `dtrmm` has an important percentage. The case of algorithm 5 is more complex, with routines `dtrmm` and `dsyrk` each covering more than 40%.

It is interesting that routine `dpotrf`[5] (performing Cholesky factorization) accounts for 0% in all cases: in fact it is a LAPACK routine performing a blocked version of the Cholesky factorization, and it makes use of other BLAS and LAPACK routines. Thus the time spent in the routine itself is almost null.

The profiler `gprof` also produces a call graph, giving useful information about the total time spent in a routine also taking into account the calls to subroutines. In the case of algorithm 5, it is interesting the fact that routine `dpotrf` and its subroutines accounts for 34.4%, that is much more the theoretical value 13.5% (computed in the approximation $n_x \gg n_u$).

5.3.2 Comparative test

In this part we compare the performance in terms of computational time of four implementation of the Riccati recursion method, namely algorithm 3, algorithm 4 and algorithm 5 in the two version where the upper or lower Cholesky factor of the P_n matrix is computed. The test problem is the mass-spring problem 7.2 with $n_u \leq n_x/2$. The tables containing all the data are D.5, D.6, D.7 and D.8.

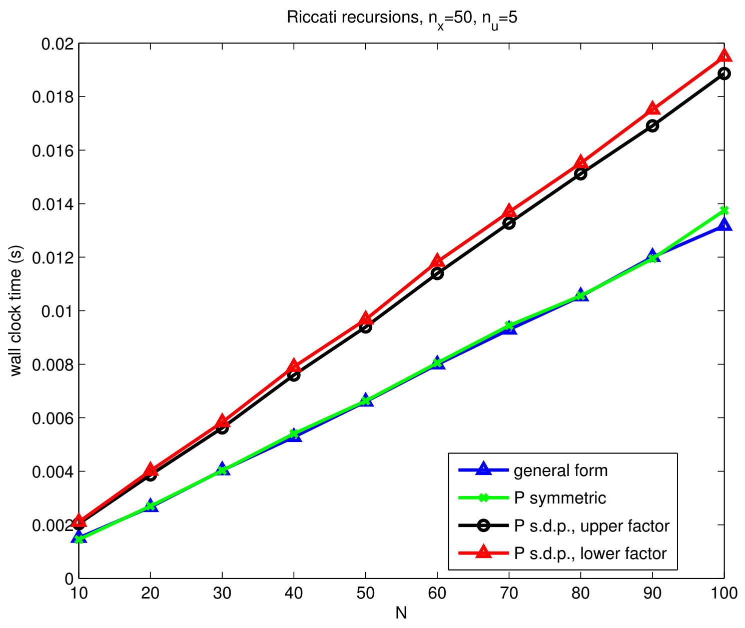
In figure 5.1a n_x and n_u are fixed and only N is varied. The behavior of the algorithm is clearly linear. With this particular choice of n_x and n_u algorithms 3 and 4 are actually faster than algorithm 5, even if the latter has a lower asymptotic complexity in terms of floating-point operations.

In figure 5.1b N and n_u are fixed and n_x is varied. The value n_u is kept fixed since it has very little influence on the computational time, holding the condition $n_x \gg n_u$. The computational time is clearly a cubic function of n_x for large systems. Algorithm 3 is the fastest for small and medium values of n_x, while algorithm 5 (in both versions) is faster for large values of n_x.

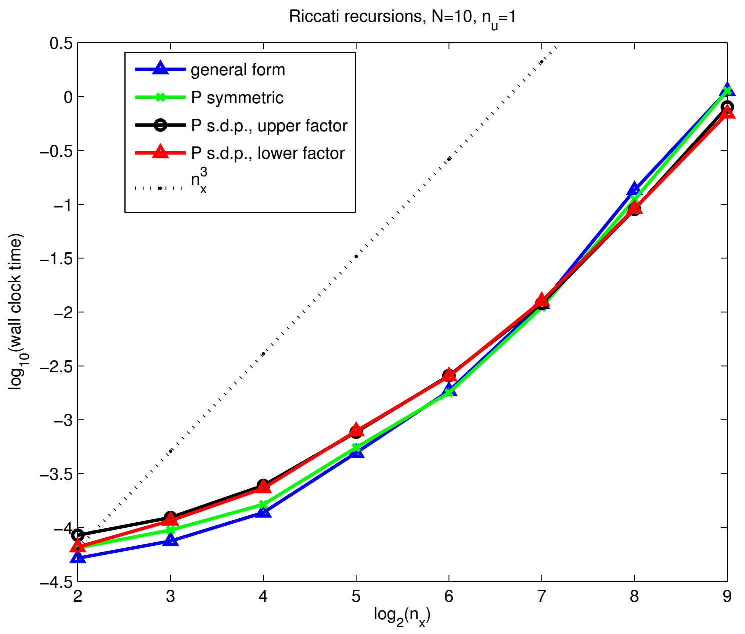
5.3.3 Conclusions

The asymptotic cost of the algorithms becomes important just for large systems. The use of specialized routines is useful just for large matrices, since in

[5]See appendix B.4.



(a) $n_x = 50$ and $n_u = 5$ fixed, N varying.



(b) $N = 10$ and $n_u = 1$ fixed, $n_x \in \{4, 8, 16, 32, 64, 128, 256, 512\}$ varying.

Figure 5.1: Comparison of the performance of four implementation of the Riccati recursion method for the solution of the problem 2.1, using as test problem the mass-spring problem 7.2: algorithm 3 (blue), algorithm 4 (green), algorithm 5, computing the upper Cholesky factor (black) or lower Cholesky factor (red).

general they are slower for small ones. For small systems, the best solution is thus the general form algorithm 3, using only routine `dgemm` for matrix-matrix multiplication.

For large systems, the best solution is algorithm 5 requiring P_n to be symmetric positive definite. Anyway, the performance is lower than the theoretical one due to the poor performance of the LAPACK routine `dpotrf` compared to the others BLAS routines. For small systems, this problem can partially be solved with the use of a recursive algorithm (as `ric2` presented in the appendix C) and efficient implementation in assembly of the base cases.

CHAPTER 6

Condensing Methods

Another approach for the solution of problem 2.1 consists in rewrite the problem from the form of a large sparse equality constrained quadratic program to a small dense system of linear equations, whose matrix is positive definite and has size $Nn_u \times Nn_u$. The solution of this system can be computed using the Cholesky factorization in time proportional to $(Nn_u)^3$, plus the time needed to build the system.

6.1 Derivation

We consider two different methods, and show how they lead to the same system of linear equaitons:

- the first one is the null space method for the solution of a generic equality constrained quadratic program
- the second one is the state elimination method: it explicitly uses the system dynamic equation to eliminate the states from the problem formulation and rewrite it as a small dense unconstrained quadratic program, whose solution is found setting its gradient to zero.

6.1.1 Null space method

This section is divided into two parts: in the first part we derive the null space method in the case of a generic equality constrained quadratic program. In the second one we use the method in the special case of problem 2.1. The use of null space method for the solution of the general equality constrained quadratic program can be found in [NW06].

General method

In this first part we present the null space method for the solution of the generic equality constrained quadratic program

$$\begin{aligned} \min_x \quad & \frac{1}{2}x'Hx + g'x \\ s.t. \quad & Ax = b \end{aligned} \tag{6.1}$$

In what follows, we suppose that the matrix A (of size $m \times n$) has full row rank m and that the matrix H (of size $n \times n$) is symmetric positive semi-definite. Furthermore, we suppose that the reduced Hessian matrix $Z'HZ$ (of size $(n-m) \times (n-m)$) is positive definite, where Z (of size $n \times (n-m)$) is a matrix with full column rank whose columns are a basis for the kernel (or null space) of the matrix A, $AZ = 0$ (this gives the name to the method). In chapter 2 we have already derived conditions on the matrices of problem 2.1 under which these hypotheses hold.

In the given hypothesis, we already know that the unique solution of 6.1 is given by the solution of the KKT system

$$\begin{bmatrix} H & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} x \\ \pi \end{bmatrix} = - \begin{bmatrix} g \\ b \end{bmatrix}. \tag{6.2}$$

Let Y be a $n \times m$ matrix such that the $n \times n$ matrix $[Y|Z]$ has full rank n (Y is not unique). Then we have that each vector x can be decomposed as

$$x = Yx_y + Zx_z.$$

Inserting this expression into the second equation in 6.2, we have that

$$Ax = A(Yx_y + Zx_z) = AYx_y + AZx_z = AYx_y = b$$

since $AZ = 0$ by definition of Z. This means that the vector Yx_y satisfies the constraints.

From the expression $A\,[Y|Z] = [AY|AZ] = [AY|0]$ we notice that AY is a squared matrix of size $m \times m$ with full rank, since A has full row rank m and $[Y|Z]$ has full rank n. This means that we can compute the value of x_y as

$$x_y = (AY)^{-1}b.$$

Inserting the decomposition $x = Yx_y + Zx_z$ into the first equation in 6.2, and using the above expression for x_y, we have

$$Hx - A'\pi = HYx_y + HZx_z - A'\pi = HY(AY)^{-1}b + HZx_z - A'\pi = -g.$$

Multiplying on the right both sides for Z' and rearranging the terms we have

$$Z'HZx_z = -Z'(g + HY(AY)^{-1}b)$$

since the term $Z'A'\pi = (AZ)'\pi = 0$.

Since the reduced Hessian matrix $Z'HZ$ is positive definite by hypothesis, we can finally solve the above equation for x_z:

$$x_z = -(Z'HZ)^{-1}Z'(g + HY(AY)^{-1}b).$$

Problem 2.1 case

We show the computations in the special case $N = 3$, since in this way the matrices are not excessively large, and it is easy to extend the procedure to the general N.

We have already computed a possible expression for the matrix Z in chapter 2. In this part we rearrange the components of the vector x in a more handy way: in what follows we assume

$$x = \left[\begin{array}{c} u_0 \\ u_1 \\ u_2 \\ \hline x_1 \\ x_2 \\ x_3 \end{array}\right].$$

The problem matrices thus take the form

$$H = \left[\begin{array}{ccc|ccc} R_0 & & & & & \\ & R_1 & & S_1 & & \\ & & R_2 & & S_2 & \\ \hline & S_1' & & Q_1 & & \\ & & S_2' & & Q_2 & \\ & & & & & Q_3 \end{array}\right] = \left[\begin{array}{c|c} \bar{R} & \bar{S} \\ \hline \bar{S}' & \bar{Q} \end{array}\right], \quad g = \left[\begin{array}{c} S_0x_0 + s_0 \\ s_1 \\ s_2 \\ \hline q_1 \\ q_2 \\ q_3 \end{array}\right] = \left[\begin{array}{c} \bar{s} \\ \hline \bar{q} \end{array}\right]$$

$$A = \left[\begin{array}{ccc|ccc} -B_0 & & & I & & \\ & -B_1 & & -A_1 & I & \\ & & -B_2 & & -A_2 & I \end{array}\right], \quad b = \left[\begin{array}{c} A_0x_0 + b_0 \\ b_1 \\ b_2 \end{array}\right]$$

and the Z matrix and a suitable Y matrix take the form

$$Z = \left[\begin{array}{ccc} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & I \\ \hline B_0 & 0 & 0 \\ A_1B_0 & B_1 & 0 \\ A_2A_1B_0 & A_2B_1 & B_2 \end{array}\right] = \left[\begin{array}{c} I \\ \hline \Gamma_u \end{array}\right], \quad Y = \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \hline I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{array}\right] = \left[\begin{array}{c} 0 \\ \hline I \end{array}\right]$$

The Z and Y matrices are not unique, and the actual value of the Y matrix influences the numerical properties of the method, but we do not care about this since we make all the computations analytically. We then choose the easier form for the matrix Y. We notice that, with these choices for the matrices Y and Z, the decomposition of the vector x is

$$x = \left[\begin{array}{c} u_0 \\ u_1 \\ u_2 \\ \hline x_1 \\ x_2 \\ x_3 \end{array}\right] = Yx_y + Zx_z = \left[\begin{array}{c} 0 \\ \hline I \end{array}\right] x_y + \left[\begin{array}{c} I \\ \hline \Gamma_u \end{array}\right] x_z = \left[\begin{array}{c} x_z \\ \hline x_y + \Gamma_u x_z \end{array}\right]$$

and then x_z is actually the input vector,

$$x_z = \left[\begin{array}{c} u_0 \\ u_1 \\ u_2 \end{array}\right].$$

With these definitions, we have that the vector x_y is

$$x_y = (AY)^{-1}b = \begin{bmatrix} I & 0 & 0 \\ -A_1 & I & 0 \\ 0 & -A_2 & I \end{bmatrix}^{-1} \begin{bmatrix} A_0x_0 + b_0 \\ b_1 \\ b_2 \end{bmatrix} \doteq \Gamma_b b =$$

$$= \begin{bmatrix} I & 0 & 0 \\ A_1 & I & 0 \\ A_2A_1 & A_2 & I \end{bmatrix} \begin{bmatrix} A_0x_0 + b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} A_0x_0 + b_0 \\ A_1(A_0x_0 + b_0) + b_1 \\ A_2(A_1(A_0x_0 + b_0) + b_1) + b_2 \end{bmatrix}$$

and finally x_z is

$$x_z = -(Z'HZ)^{-1}\big(Z'(g + HYx_y)\big) \doteq -\bar{H}^{-1}\bar{g}$$

where

$$\bar{H} = Z'HZ = \begin{bmatrix} I & \Gamma_u' \end{bmatrix} \begin{bmatrix} \bar{R} & \bar{S} \\ \bar{S}' & \bar{R} \end{bmatrix} \begin{bmatrix} I \\ \Gamma_u \end{bmatrix} = \bar{R} + \Gamma_u'\bar{S}' + \bar{S}\Gamma_u + \Gamma_u'\bar{Q}\Gamma_u =$$

$$= \begin{bmatrix} R_0 & 0 & 0 \\ 0 & R_1 & 0 \\ 0 & 0 & R_2 \end{bmatrix} + \begin{bmatrix} 0 & B_0'S_1' & B_0'A_1'S_2' \\ 0 & 0 & B_1'S_2' \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ S_1B_0 & 0 & 0 \\ S_2A_1B_0 & S_2B_1 & 0 \end{bmatrix} +$$

$$+ \left[\begin{matrix} B_0'Q_1B_0 + B_0'A_1'Q_2A_1B_0 + B_0'A_1'A_2'Q_3A_2A_1B_0 & \dots \\ S_1B_0 + B_1'Q_2A_1B_0 + B_1'A_2'Q_3A_2A_1B_0 & \dots \\ S_2A_1B_0 + B_2'Q_3A_2A_1B_0 & \dots \end{matrix}\right.$$

$$\left.\begin{matrix} B_0'S_1' + B_0'A_1'Q_2B_1 + B_0'A_1'A_2'Q_3A_2B_1 & B_0'A_1'A_2'Q_3B_2 \\ R_1 + B_1'Q_2B_1 + B_1'A_2'Q_3A_2B_1 & B_1'S_2' + B_1'A_2'Q_3B_2 \\ S_2B_1 + B_2'Q_3A_2B_1 & R_2 + B_2'Q_3B_2 \end{matrix}\right]$$

$$= \left[\begin{matrix} R_0 + B_0'Q_1B_0 + B_0'A_1'Q_2A_1B_0 + B_0'A_1'A_2'Q_3A_2A_1B_0 & \dots \\ B_1'Q_2A_1B_0 + B_1'A_2'Q_3A_2A_1B_0 & \dots \\ B_2'Q_3A_2A_1B_0 & \dots \end{matrix}\right.$$

$$\left.\begin{matrix} B_0'S_1' + B_0'A_1'Q_2B_1 + B_0'A_1'A_2'Q_3A_2B_1 & B_0'A_1'S_2' + B_0'A_1'A_2'Q_3B_2 \\ B_1'Q_2B_1 + B_1'A_2'Q_3A_2B_1 & B_1'A_2'Q_3B_2 \\ B_2'Q_3A_2B_1 & B_2'Q_3B_2 \end{matrix}\right]$$

and

$$\bar{g} = Z'(g + HYx_y) = \begin{bmatrix} I & \Gamma_u' \end{bmatrix} \left(\begin{bmatrix} \bar{s} \\ \bar{q} \end{bmatrix} + \begin{bmatrix} \bar{S} \\ \bar{Q} \end{bmatrix} \Gamma_b b \right) =$$

$$= \begin{bmatrix} I & \Gamma_u' \end{bmatrix} \begin{bmatrix} \bar{s} + \bar{S}\Gamma_b b \\ \bar{q} + \bar{Q}\Gamma_b b \end{bmatrix} = \bar{s} + \bar{S}\Gamma_b b + \Gamma_u'(\bar{q} + \bar{Q}\Gamma_b b) =$$

$$= \begin{bmatrix} s_0 + S_0x_0 + B_0'(q_1 + Q_1(A_0x_0 + b_0)) + B_0'A_1'(q_2 + Q_2(A_1(A_2x_0 + b_0) + b_1) + \\ \qquad + B_0'A_1'A_2'(q_3 + Q_3(A_2(A_1(A_0x_0 + b_0) + b_1) + b_2)) \\ s_1 + S_1(A_0x_0 + b_0) + B_1'(q_2 + Q_2(A_1(A_0x_0 + b_0) + b_1)) + \\ \qquad + B_1'A_2'(q_3 + Q_3(A_2(A_1(A_0x_0 + b_0) + b_1) + b_2)) \\ s_2 + S_2(A_1(A_0x_0 + b_0) + b_1)) + B_2'(q_3 + Q_3(A_2(A_1(A_0x_0 + b_0) + b_1) + b_2)) \end{bmatrix}$$

6.1.2 State elimination

A different approach, leading to the same system of linear equations, is the state elimination method: exploiting the fact that it is possible to write the future states $x_k, k \in \{1, \dots, N\}$ as function of the initial state x_0 and the (unknown) inputs $u_k, k \in \{0, 1, \dots, N-1\}$, it is possible to eliminate the states from the formulation of problem 2.1, dropping the direct dependence on the state vector.

The method consist of two phases: the preprocessing phase, rewriting the problem in the form of an unconstrained quadratic program, with a cost definite positive cost function in the only inputs u_k; and the solution phase, where the unconstrained quadratic program is solved by setting its gradient to zero.

We derive the method directly in the case of problem 2.1, for a general value of N.

Let us define the unknown input vector as

$$\bar{u} = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ n_{N-1} \end{bmatrix}$$

that is a vector of size $Nn_u \times 1$.

Our purpose is to rewrite the cost function ϕ as a quadratic function of the only $\bar{u}$, given the initial state vector x_0 of size $n_x \times 1$, and the system dynamic equation

$$x_{k+1} = A_k x_k + B_k u_k + b_k. \tag{6.3}$$

The cost function can be rewritten as

$$\begin{aligned}
\phi =& \sum_{n=0}^{N-1} \left(\frac{1}{2} x'_n Q_n x_n + x'_n S'_n u_n + \frac{1}{2} u'_n R_n u_n + q'_n x_n + s'_n u_n + \rho_n \right) + \\
&+ \frac{1}{2} x'_N Q_N x_N + q'_N x_N + \rho_N = \\
=& \frac{1}{2} \sum_{n=1}^{N} x'_n Q_n x_n + \sum_{n=1}^{N-1} x'_n S'_n u_n + \frac{1}{2} \sum_{n=0}^{N-1} u'_n R_n u_n + \sum_{n=1}^{N} q'_n x_n + \\
&+ \left(x'_0 S'_0 u_0 + \sum_{n=0}^{N-1} s'_n u_n \right) + \left(\frac{1}{2} x'_0 Q_0 x_0 + q'_0 x_0 + \sum_{n=0}^{N} \rho_n \right) = \\
=& \frac{1}{2} \bar{x}' \bar{Q} \bar{x} + \frac{1}{2} \bar{x}' \bar{S}' \bar{u} + \frac{1}{2} \bar{u}' \bar{S} \bar{x} + \frac{1}{2} \bar{u}' \bar{R} U + \bar{q}' \bar{x} + \bar{s}' \bar{u} + \bar{\rho},
\end{aligned}$$

where we define the scalar $\bar{\rho} = \frac{1}{2} x'_0 Q_0 x_0 + q'_0 x_0 + \sum_{n=0}^{N} \rho_n$ and the matrices

$$\bar{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}, \quad \bar{Q} = \begin{bmatrix} Q_1 & 0 & \dots & 0 \\ 0 & Q_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & Q_N \end{bmatrix}, \quad \bar{S} = \begin{bmatrix} 0 & \dots & \dots & 0 \\ S_1 & \ddots & & \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & S_{N-1} & 0 \end{bmatrix}$$

$$\bar{R} = \begin{bmatrix} R_0 & & 0 \\ & \ddots & \\ 0 & & R_{N-1} \end{bmatrix}, \quad \bar{q} = \begin{bmatrix} q_1 \\ \vdots \\ q_N \end{bmatrix}, \quad \bar{s} = \begin{bmatrix} s_0 + S_0 x_0 \\ \vdots \\ s_{N-1} \end{bmatrix}$$

of size: $\bar{x}$ is $Nn_x \times 1$, $\bar{Q}$ is $Nn_x \times Nn_x$, $\bar{S}$ is $Nn_u \times Nn_x$, $\bar{R}$ is $Nn_u \times Nn_u$, $\bar{q}$ is $Nn_x \times 1$ and $\bar{r}$ is $Nn_u \times 1$.

Using equation 6.3, the state vector can be written as

$$\bar{x} = \Phi_x x_0 + \Gamma_u \bar{u} + \Gamma_b \bar{b} \tag{6.4}$$

where

$$\Phi_x = \begin{bmatrix} A_0 \\ A_1 A_0 \\ A_{N-1} \dots A_1 A_0 \end{bmatrix}, \quad \Gamma_u = \begin{bmatrix} B_0 & 0 & \dots & 0 \\ A_1 B_0 & B_1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ A_{N-1} \dots A_1 B_0 & A_{N-1} \dots A_2 B_1 & \dots & B_{N-1} \end{bmatrix}$$

$$\Gamma_b = \begin{bmatrix} I & 0 & \dots & 0 \\ A_1 & I & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ A_{N-1} \dots A_1 & A_{N-1} \dots A_2 & \dots & I \end{bmatrix}, \quad \bar{b} = \begin{bmatrix} b_0 \\ \vdots \\ b_{N-1} \end{bmatrix}$$

Substituting 6.4 into the cost function expression, we obtain the formulation as a function of the only $\bar{u}$

$$\phi = \phi(\bar{u}) = \frac{1}{2} \bar{u}' \bar{H} \bar{u} + \bar{g}' \bar{u} + \bar{\rho}$$

where

$$\begin{aligned} \bar{H} &= \Gamma_u' \bar{Q} \Gamma_u + \Gamma_u' \bar{S}' + \bar{S} \Gamma_u + \bar{R} \\ \bar{g} &= \Gamma_u' \bar{Q} (\Phi_x x_0 + \Gamma_b \bar{b}) + \bar{S} (\Phi_x x_0 + \Gamma_b \bar{b}) + \Gamma_u' \bar{q} + \bar{s} \\ \bar{\rho} &= \frac{1}{2} (\Phi_x x_0 + \Gamma_b \bar{b})' \bar{Q} (\Phi_x x_0 + \Gamma_b \bar{b}) + \bar{q}' (\Phi_x x_0 + \Gamma_b \bar{b}) + \bar{\rho} \end{aligned}$$

The matrix $\bar{H}$ is $Nn_u \times Nn_u$, the vector $\hat{g}$ is $Nn_u \times 1$ and $\hat{\rho}$ is a scalar.

We notice that the $\bar{H}$ matrix is exactly the same as in the null space method, and that the matrices Φ_x and Γ_b are used only in the expression $\Phi_x x_0 + \Gamma_b \bar{b}$. Then defining

$$\Phi_x x_0 + \Gamma_b \bar{b} = \Gamma_b b$$

where

$$b = \begin{bmatrix} A_0 x_0 + b_0 \\ b_1 \\ \vdots \\ b_{N-1} \end{bmatrix}$$

we find again for $\bar{g}$ the expression

$$\bar{g} = \Gamma_u' \bar{Q} \Gamma_b b + \bar{S} \Gamma_b b + \Gamma_u' \bar{q} + \bar{s}$$

that is the exactly same as in the null space method.

The $\bar{H}$ matrix is positive definite, since it coincides with the reduced Hessian $Z'HZ$ matrix of the null space method, that is positive definite by hypothesis.

The $\bar{H}$ matrix is then invertible, and so the minimum of the cost function is unique, and can be found setting to zero the gradient of the cost function with respect to $\bar{u}$:

$$\nabla\phi(\bar{u}) = \bar{H}\bar{u} + \bar{g} = 0 \quad \Rightarrow \quad \bar{u}^* = -\bar{H}^{-1}\bar{g}.$$

The value of $\hat{\rho}$ does not influence the minimizer $\bar{u}^*$ but just the value of the minimum of the cost function: since we are only interested in the value of the minimizer and not on the value of the minimum, we avoid the computation of $\hat{\rho}$.

6.2 Implementation

In the case of the condensing methods, the implementation plays a key role in the performance of the algorithm: in fact, the matrices defined in the previous section have a lot of structure, and exploiting this fact it is possible to write algorithms using much less space and time.

As we have already seen, the algorithm consist in two parts: a first part (preprocessing phase), where the $\bar{H}$ matrix and the $\bar{g}$ vector are build; and a second part (solution phase) where the system of linear equations is solved. The $\hat{H}$ matrix is symmetric positive definite by hypothesis, and so we can solve the system of linear equations using the Cholesky factorization. The $\hat{H}$ matrix has size Nn_u, and then the cost for the factorization of the matrix is roughly $\frac{1}{3}(Nn_u)^3$, cubic in both the horizon length N and the input number n_u.

In the previous chapters we saw algorithms for the solution of 2.1 in time linear in N and cubic in both n_x and n_u: then the field of application of the condensing methods is necessary the case of problems with small values of the horizon length N and the number of inputs n_u.

In the condensing phase, it is usually possible to use two different approaches in the construction of matrices, leading to different complexities: an approach produces algorithm with higher complexity in N and lower in n_x, while on the opposite the other approach produces algorithms with lower complexity in N and higher in n_x. Since the solution phase is already cubic in N, we choose the first approach, in the hope to obtain an algorithm with good performances in the case of large value of n_x. This is true, since following this approach it is possible to obtain an algorithm quadratic in n_x, while the algorithms presented in the previous chapters are cubic in n_x.

We notice that the $\bar{Q}, \bar{S}$ and $\bar{R}$ matrices are mainly zeros, and can become very large (especially $\bar{Q}$, of size $Nn_x \times Nn_x$): the only function of these zeros is to keep the sub-matrices Q_k, S_k, R_k in the correct position during matrix multiplications. Then it is possible to avoid the construction of these matrices, and instead explicitly perform the correct matrix multiplications using the sub-matrices, at the cost of a larger number of calls to the matrix multiplication routine `dgemm`[1]. The number of calls to this routine is linear in N.

The largest matrix is Γ_b, of size $Nn_x \times Nn_x$; it is used only in the product $\Gamma_b b$, that is a vector of size $Nn_x \times 1$. In the case $N = 3$ this product is written

[1]See appendix B.1.1.

as

$$\Gamma_b b = \begin{bmatrix} I & 0 & 0 \\ A_1 & I & 0 \\ A_2A_1 & A_2 & I \end{bmatrix} \begin{bmatrix} A_0x_0 + b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} A_0x_0 + b_0 \\ A_1(A_0x_0 + b_0) + b_1 \\ A_2A_1(A_0x_0 + b_0) + A_2b_1 + b_2 \end{bmatrix} =$$

$$= \begin{bmatrix} A_0x_0 + b_0 \\ A_1(A_0x_0 + b_0) + b_1 \\ A_2(A_1(A_0x_0 + b_0) + b_1) + b_2 \end{bmatrix}$$

where in the last term we are stressing the fact that it is possible to exploit the structure of the matrix Γ_b and directly build $\Gamma_b\bar{b}$ in an efficient way.

The $\Gamma_b b$ vector is initialized computing $A_0x_0 + b_0$ in the block of index 0 and, for each i, copying b_i in the block of index i (where the indexes refer to the blocks of size $n_x \times 1$, starting from index 0). The $\Gamma_b b$ vector is built from the top to the bottom: the expression of the block of index i is obtained by multiplying A_i on the right by the block of index $i-1$, and adding the result to the already copied b_i. In this way the creation of $\Gamma_b b$ requires Nn_x^2 floating-point multiplications (and roughly the same number of floating-point sums), plus the initial copy of Nn_x floating-point numbers (not considered in the complexity computation).

After this, the most expensive part of the condensing phase is the computation of the $\Gamma_u'\bar{Q}\Gamma_u$ matrix. There are at least two different approaches: the first one consists on the computation the product using the `dgemm` routine, and working only on the sub-matrices to avoid zero multiplications. The second one is similar to the procedure used for the computation of the term $A'P_nA$ of the Riccati recursion in the case of P symmetric positive definite: namely, Cholesky factorization of $\bar{Q} = LL'$, computation of $L'\Gamma_u$ and of $\Gamma_u'\bar{Q}\Gamma_u = (L'\Gamma_u)'L'\Gamma_u$ using specialized routines. The problem with this second approach is that the Cholensky factorization of the matrix $\bar{Q}$ requires $N\frac{1}{3}n_x^3$ floating-point operations, while the first approach is quadratic in n_x. We choose the first one.

The first step in the computation of $\Gamma_u'\bar{Q}\Gamma_u$ is the creation of the matrix Γ_u, of size $Nn_x \times Nn_u$. In the case $N = 3$, it is

$$\Gamma_u = \begin{bmatrix} B_0 & 0 & 0 \\ A_1B_0 & B_1 & 0 \\ A_2A_1B_0 & A_2B_1 & B_2 \end{bmatrix}.$$

We noticed that an efficient procedure to build this matrix is in two steps, each consisting on a loops over the horizon. In the first loop, B_i is copyed in the block of indexes (i,i) (where the indexes refer to the blocks of size $n_x \times n_u$, starting from index 0). In the second loop, the matrix A_i is multiplied on the right by the block row with row index $i-1$ and column indexes from 0 to $i-1$ (both inclusive), and the result is written in the row block with row index i and column indexes from 0 to $i-1$ (again both inclusive). $N-1$ calls to `dgemm` are performed, and each time the first factor is a matrix of constant size, while the second factor is a matrix with constant number of rows and a number of columns equal to $i+1$. The total cost of this step is $\frac{N(N-1)}{2}n_x^2n_u$ floating-point multiplications (and roughly an equal number of floating-point sums), plus the initial copy of Nn_xn_u floating-point numbers (not considered).

The next step is the creation of the matrix $\bar{Q}\Gamma_u$, again of size $Nn_x \times Nn_u$.

In the case $N = 3$ it is

$$\bar{Q}\Gamma_u = \begin{bmatrix} Q_1B_0 & 0 & 0 \\ Q_2A_1B_0 & Q_2B_1 & 0 \\ Q_3A_2A_1B_0 & Q_3A_2B_1 & Q_3B_2 \end{bmatrix}.$$

For each i, the matrix Q_{i+1} is multiplied on the right by the row block from the Γ_u matrix with row index i and column indexes from 0 to i (both inclusive). The result is written in the row block from the $\bar{Q}\Gamma_u$ matrix with row index i and column indexes from 0 to i (both inclusive). N calls to `dgemm` are performed, and the cost of this step is $\frac{N(N+1)}{2}n_x^2n_u$ floating-point multiplications (and roughly the same number of floating-point sums).

The last step consists in the creation of the matrix $\Gamma_u'\bar{Q}\Gamma_u$, of size $Nn_u \times Nn_u$. There are at lest two approaches to perform this computation. One consist in the explicit performance of the multiplication $\Gamma_u' \cdot \bar{Q}\Gamma_u$, avoiding the multiplications when one of the factor is a zero block and computing the only lower (or upper factor). The complexity is roughly $\frac{1}{3}N^3n_xn_u^2$ floating point multiplications: the sum of the exponents of the terms N, n_x and n_u is 6.

Instead, it is possible to write an algorithm with sum of the exponents equal to 5 in the following way: we notice that for $N = 3$ the matrix $\Gamma_u'\bar{Q}\Gamma_u$ is

$$\Gamma_u'\bar{Q}\Gamma_u = \Gamma_u' \cdot \bar{Q}\Gamma_u = \begin{bmatrix} B_0' & B_0'A_1' & B_0'A_1'A_2' \\ 0 & B_1' & B_1'A_2' \\ 0 & 0 & B_2' \end{bmatrix} \begin{bmatrix} Q_1B_0 & 0 & 0 \\ Q_2A_1B_0 & Q_2B_1 & 0 \\ Q_3A_2A_1B_0 & Q_3A_2B_1 & Q_3B_2 \end{bmatrix} =$$

$$= \begin{bmatrix} B_0'(Q_1B_0 + A_1'(Q_2A_1B_0 + A_2'(Q_3A_2A_1B_0))) & \dots \\ B_1'(Q_2A_1B_0 + A_2'(Q_3A_2A_1B_0)) & \dots \\ B_2'(Q_3A_2A_1B_0) & \dots \end{bmatrix}$$

$$\begin{matrix} B_0'(0 + A_1'(Q_2B_1 + A_2'(Q_3A_2B_1))) & B_0'(0 + A_1'(0 + A_2'(Q_3B_2))) \\ B_1'(Q_2B_1 + A_2'(Q_3A_2B_1)) & B_1'(0 + A_2'(Q_3B_2))2 \\ B_2'(Q_3A_2B_1) & B_2'(Q_3B_2) \end{matrix} \Bigg]$$

In the last formulation we want to stress the fact that it is possible to build the matrix $\Gamma_u'\bar{Q}\Gamma_u$ in two steps: the first step is carried on in place, on the matrix $\bar{Q}\Gamma_u$, and consists in a loop on i from $N-1$ to 1 (both inclusive). For each i, the A_i' matrix is multiplied on the right by the row block from the $\bar{Q}\Gamma_u$ matrix with row index i, and the result is added to the row block from the $\bar{Q}\Gamma_u$ matrix with row index $i-1$. The second step consists in the creation of the matrix $\hat{H}$ of size $Nn_u \times Nn_u$, and is again a loop on i from $N-1$ to 0. For each i the matrix B_i' is multiplied on the right by the row block from the just modified matrix $\bar{Q}\Gamma_u$ with index i, and the result is written on the row block from the $\hat{H}$ matrix with index i.

The LAPACK routine `dpotrf`[2] for the computation of the Cholesky factorization requires only the lower or upper triangular matrix. Then we decide to build only the lower triangular part of $\Gamma_u'\bar{Q}\Gamma_u$. The resulting matrix $\hat{H}$ is

$$\hat{H} =$$

$$\begin{bmatrix} B_0'(Q_1B_0 + A_1'(Q_2A_1B_0 + A_2'(Q_3A_2A_1B_0))) & 0 & 0 \\ B_1'(Q_2A_1B_0 + A_2'(Q_3A_2A_1B_0)) & B_1'(Q_2B_1 + A_2'(Q_3A_2B_1)) & 0 \\ B_2'(Q_3A_2A_1B_0) & B_2'(Q_3A_2B_1) & B_2'(Q_3B_2) \end{bmatrix}$$

[2]See appendix B.4.

The first step requires $N-1$ calls to `dgemm` and $\frac{N(N-1)}{2}n_x^2 n_u$ floating-point multiplications (and roughly the same number of sums), the second N calls and $\frac{N(N+1)}{2}n_x n_u^2$ floating-point multiplications (and roughly the same number of sums).

Another optimization that can be performed in the computation of $\hat{H}$ is the following: it is possible to avoid the storing of all the zeros in the matrices Γ_u and $\bar{Q}\Gamma_u$, saving almost half of the space. Furthermore it is possible to choose a representation of data in memory that minimizes cache misses. In fact, in the construction of $\hat{H}$ operations are performed on sub-matrices that are block rows. Then it is convenient to save these blocks contiguously in memory. Both MATLAB and our C implementation are using the column-major order, and then it is convenient to save Γ_u and $\bar{Q}\Gamma_u$ as a long horizontal concatenation of these block rows: in this way, the data within each block is saved contiguously in memory, one column after the other, without jumps. For example, for $N=3$, Γ_u is actually saved in memory as the $n_x \times \frac{N(N+1)}{2}n_u$ matrix:

$$\widetilde{\Gamma}_u = \left[\begin{array}{c|c|c|c|c|c} B_0 & A_1B_0 & B_1 & A_2A_1B_0 & A_2B_1 & B_2 \end{array}\right],$$

and $\bar{Q}\Gamma_u$ as the $n_x \times \frac{N(N+1)}{2}n_u$ matrix:

$$\widetilde{\bar{Q}\Gamma_u} = \left[\begin{array}{c|c|c|c|c|c} Q_1B_0 & Q_2A_1B_0 & Q_2B_1 & Q_3A_2A_1B_0 & Q_3A_2B_1 & Q_3B_2 \end{array}\right].$$

In the computation of H, the two terms $\bar{S}\Gamma_u$ and $\Gamma_u'\bar{S}'$ need to be built and added. In the case $N=3$

$$\bar{S}\Gamma_u = \begin{bmatrix} 0 & 0 & 0 \\ S_1 & 0 & 0 \\ 0 & S_2 & 0 \end{bmatrix} \begin{bmatrix} B_0 & 0 & 0 \\ A_1B_0 & B_1 & 0 \\ A_2A_1B_0 & A_2B_1 & B_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ S_1B_0 & 0 & 0 \\ S_2A_1B_0 & S_2B_1 & 0 \end{bmatrix}$$

and then $\bar{S}\Gamma_u$ is a strictly lower triangular matrix. Since the matrix $\hat{H}$ contains only the lower triangular part of H, only the matrix $\bar{S}\Gamma_u$ needs to be computed, and not $\Gamma_u'\bar{S}'$.

Regarding the phase of the solution of the system $\bar{H}\bar{u}+\bar{g}=0$, the symmetric positive definite matrix $\bar{H}$ is factorized using the Cholesky factorization. The system is then solved with forward and backward substitutions for the solution of the two resulting triangular systems, obtaining the input vector $\bar{u}$. The state vector is then obtained as

$$\bar{x}_{n+1} = A_n\bar{x}_n + B_n\bar{u}_n + b_n.$$

In the following algorithm 6 we summarize the condensing algorithm and we compute its complexity as number of floating-point operations up to the quadratic terms in n_x and n_u.

It is interesting to analyze how the computational cost is distributed among the different parts of the overall algorithm. We choose to divide the algorithm into the following parts, with the asymptotic number of floating-point operations:

create Γ_u : $n_x^2 n_u N^2$ flops

create $\bar{Q}\Gamma_u$: $n_x^2 n_u N^2$ flops

Algorithm 6 Condensing method for the solution of the extended linear quadratic control problem

Require: $(x_0, \{Q_n\}, \{S_n\}, \{R_n\}, \{q_n\}, \{s_n\}, \{A_n\}, \{B_n\}, \{b_n\}, Q_N, q_N)$

```
  $\bar{s}_0 \leftarrow s_0 + S_0 \cdot_{\text{dgemv}} x_0$        ▷ $2n_x n_u$ flops
  $\bar{q}_0 \leftarrow q_1$
  for $n = 1 \to N-1$ do
      $\bar{s}_n \leftarrow s_n$
      $\bar{q}_n \leftarrow q_{n+1}$
  end for

  for $n = 0 \to N-1$ do
      $(\Gamma_u)_{(n,n)} \leftarrow B_n$
  end for
  for $n = 1 \to N-1$ do
      $(\Gamma_u)_{(n,0:n-1)} \leftarrow A_n \cdot_{\text{dgemm}} (\Gamma_u)_{(n-1,0:n-1)}$   ▷ $\frac{N(N-1)}{2}(2n_x^2 n_u - n_x n_u)$ flops
  end for

  for $n = 1 \to N$ do
      $(\bar{Q}\Gamma_u)_{(n-1,0:n-1)} \leftarrow Q_n \cdot_{\text{dgemm}} (\Gamma_u)_{(n-1,0:n-1)}$
                                                   ▷ $\frac{N(N+1)}{2}(2n_x^2 n_u - n_x n_u)$ flops
  end for

  $(\Gamma_b b)_0 \leftarrow A_0 \cdot_{\text{dgemv}} x_0 + b_0$        ▷ $2n_x^2$ flops
  for $n = 1 \to N-1$ do
      $(\Gamma_b b)_n \leftarrow A_n \cdot_{\text{dgemv}} (\Gamma_b b)_{n-1} + b_n$      ▷ $(N-1)2n_x^2$ flops
  end for

  $\bar{g}_0 \leftarrow \bar{s}_0 + (\Gamma_u')_{(0,0:N-1)} \cdot_{\text{dgemv}} \bar{q}_{(0:N-1)} + (\bar{Q}\Gamma_u')_{(0,0:N-1)} \cdot_{\text{dgemv}} (\Gamma_b b)_{(0:N-1)}$
                                                   ▷ $N4n_x n_u$ flops
  for $n = 1 \to N-1$ do
      $\bar{g}_n \leftarrow \bar{s}_n + S_n \cdot_{\text{dgemv}} (\Gamma_b b)_{n-1} + (\Gamma_u')_{(n,n:N-1)} \cdot_{\text{dgemv}} \bar{q}_{(n:N-1)} +$
  $(\bar{Q}\Gamma_u')_{(n,n:N-1)} \cdot_{\text{dgemv}} (\Gamma_b b)_{(n:N-1)}$
                                   ▷ $(N-1)2n_x n_u + \frac{N(N-1)}{2}4n_x n_u$ flops
  end for

  for $n = 0 \to N-1$ do
      $\bar{H}_{(n,n)} \leftarrow R_n$
  end for

  for $n = 1 \to N-1$ do
      $\bar{H}_{(n,0:n-1)} \leftarrow \bar{H}_{(n,0:n-1)} + S_n \cdot_{\text{dgemm}} (\Gamma_u)_{(n,0:n-1)}$   ▷ $\frac{N(N-1)}{2}2n_x n_u^2$ flops
  end for
```

```
for $n = N-1 \to 1$ do
    $(\bar{Q}\Gamma_u)_{(n-1,0:n-1)} \leftarrow A'_n \cdot_{\text{dgemm}} (\bar{Q}\Gamma_u)_{(n,0:n-1)}$
                                        $\triangleright \frac{N(N-1)}{2}(2n_x^2 n_u - n_x n_u)$ flops
end for
for $n = 0 \to N-1$ do
    $\bar{H}_{(n,0:n)} \leftarrow \bar{H}_{(n,0:n)} + B'_n \cdot_{\text{dgemm}} (\bar{Q}\Gamma_u)_{(n,0:n)}$    $\triangleright \frac{N(N+1)}{2}(2n_x n_u^2)$ flops
end for

$L \leftarrow_{\text{dpotrf}} chol(\bar{H})$                    $\triangleright \frac{1}{3}(Nn_u)^3 + \frac{1}{2}(Nn_u)^2$ flops
$\bar{u} \leftarrow_{\text{dtrsm}} -L^{-1}((L')^{-1}\bar{g})$                       $\triangleright 2(Nn_u)^2$ flops

$\bar{x}_1 = A_0 \cdot_{\text{dgemv}} x_0 + B_0 \cdot_{\text{dgemv}} \bar{u}_0 + b_0$              $\triangleright 2n_x^2 + 2n_x n_u$ flops
for $n = 1 \to N-1$ do
    $\bar{x}_{n+1} = A_n \cdot_{\text{dgemv}} \bar{x}_n + B_n \cdot_{\text{dgemv}} \bar{u}_n + b_n$   $\triangleright (N-1)(2n_x^2 + 2n_x n_u)$ flops
end for

return $(\{\bar{u}_n\}, \{\bar{x}_n\})$
```

create $\bar{g}$: $2n_x n_u N^2 + 2n_x^2 N$ flops

create $\bar{H}$: $n_x^2 n_u N^2 + 2n_x n_u^2 N^2$ flops

compute $\bar{u}$: $\frac{1}{3}n_u^3 N^3$ flops

compute $\bar{x}$: $n_x n_u N^2$ flops

The asymptotic complexity of the overall algorithm (condensing phase and system solution phase) is

$$3n_x^2 n_u N^2 + 2n_x n_u^2 N^2 + \frac{1}{3}n_u^3 N^3$$

floating-point operations, that is quadratic in n_x and cubic in both n_u and N: the algorithm can have some advantages over the methods presented in the previous chapters for large values of the number of states n_x.

6.3 Performance analysis

In this section we analyze the performance of our implementation of the algorithm 6. As test problem, we choose a medium size instance of the mass-spring problem 7.2, with $n_x = 40$, $n_u = 2$ and $N = 30$.

6.3.1 Cost of the different parts of the algorithm

In this part we analyze in practice the cost of the different parts of our implementation of the condensing method. In the test we used the MKL BLAS. The percentage of the total computational time needed by the different parts is in table 6.1. Most part of the time is used to compute Γ_u, $\bar{Q} \cdot \Gamma_u$ and finally the matrix $\bar{H}$ (out of which, especially the part $\Gamma'_u \cdot (\bar{Q}\Gamma_u)$), while for this problem size the time needed to actually solve the system is only 5.5%.

part	percentage
compute Γ_u	23.6
compute $\bar{Q} \cdot \Gamma_u$	25.7
compute $\bar{g}$	9.9
compute $\bar{H}$	32.9
compute $\bar{u}$	5.5
compute $\bar{x}$	2.3

Table 6.1: CPU time needed by the different parts of the condensing method algorithm, for a problem of size $n_x = 40$, $n_u = 2$ and $N = 30$.

routine	percentage
dgemm	94.77
dgemv	4.36
dtrsm	0.25
dpotrf	0.0
others	0.62

Table 6.2: Cost in percentage of the total computation time of the different BLAS and LAPACK routines, using as test problem the mass-spring problem 7.2 with $n_x = 40$, $n_u = 2$ and $N = 30$.

In an interior-point method framework, a sequence of equality constrained quadratic programs (all with the same A_n and B_n matrices and different Q_n, S_n and R_n matrices) needs to be solved, one problem at each iteration of the interior-point method. Then the matrix Γ_u can be computed just once for all problems in the sequence, saving a good percentage of the computation time. Even more, in case of diagonal Q_n and inequality constraints just in the form of box constraints on the states or inputs (i.e. in the form $x_{n,\min} \leq x_n \leq x_{n,\max}$ and $u_{n,\min} \leq u_n \leq u_{n,\max}$), the computation of the matrix $\bar{Q} \cdot \Gamma_u$ requires order of $N^2 n_x n_u$ instead of $N^2 n_x^2 n_u$, and then it is possible to save another important percentage of the computation time: in the case of the problem considered in table 6.1, the total saving is around 50%.

6.3.2 Cost of sub-routines

In this part we analyze how the computational cost is distributed among the different BLAS and LAPACK routines: this can be done easily using the profiler `gprof`[3] and compiling the solver code and the BLAS and LAPACK libraries using the flag `-pg`. The test problem is the mass-spring test problem 7.2, with $n_x = 40$, $n_u = 2$, $N = 30$. The results are presented in table 6.2.

The major part of the computation time is used by the BLAS matrix-matrix multiplication routine `dgemm` (94.77%), followed by the BLAS matrix-vector multiplication routine `dgemv` (4.36%).

The limiting factor in the performances of our implementation of the condensing method is then the efficient implementation of the level-3 BLAS `dgemm` routine.

[3]See appendix A.

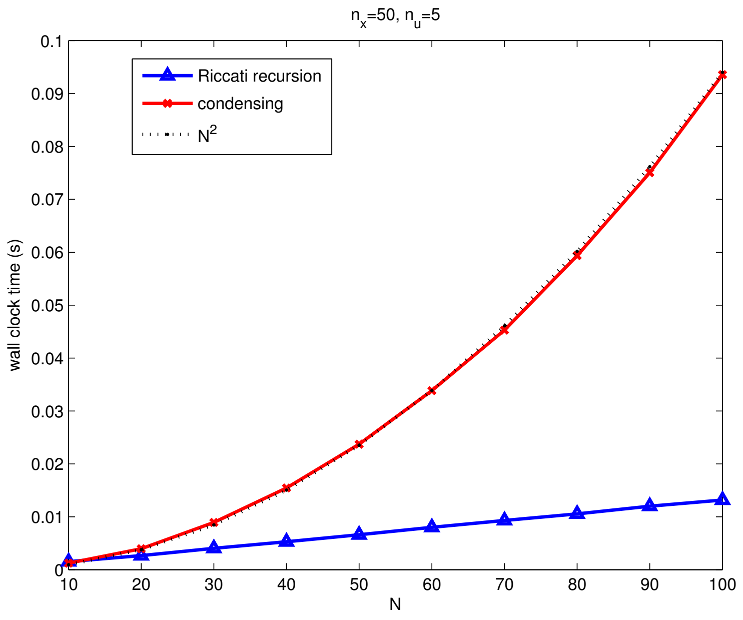


Figure 6.1: Comparison of the performances of our implementations of algorithm 3 (blue) and 6 (red). The test problem is the mass-spring problem 7.2. $n_x = 50$ and $n_u = 5$ fixed, N varying.

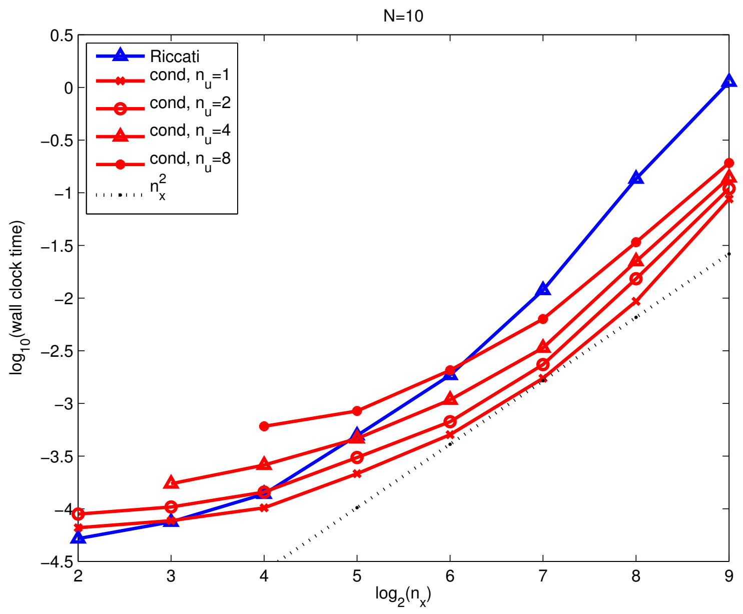
6.3.3 Comparative test

In this part we compare the performance of our implementation of algorithm 6 with our implementation of algorithm 3. The test problem is the mass-spring problem 7.2.

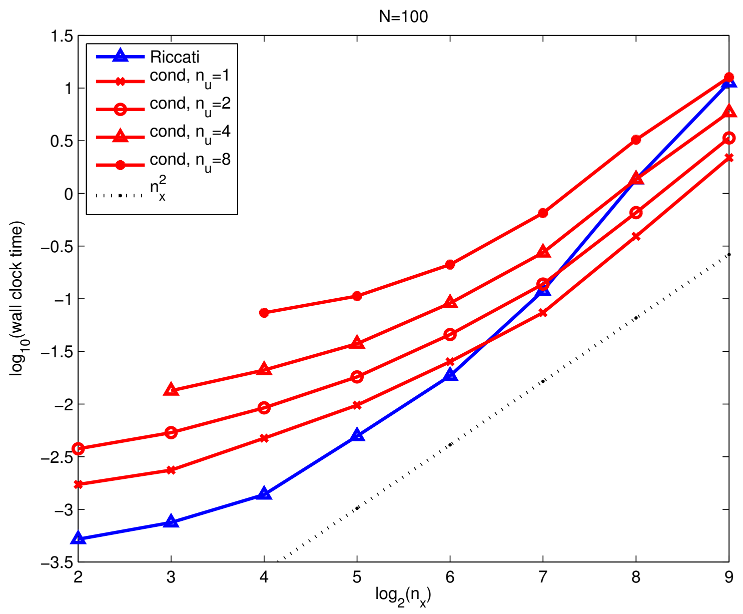
In figure 6.1 $n_x = 50$ and $n_u = 5$ are fixed, and N is varied. It is clear that the condensing method is not linear in N. It is interesting to see that it is almost quadratic, even if the cost function is cubic: in fact, for the size of the test problem, the quadratic cost for the construction of the matrix $\hat{H}$ is dominant with respect to the cubic cost for the factorization of the matrix $\hat{H}$.

In figure 6.2a and 6.2b N is fixed (to $N = 10$ and $N = 100$ respectively) while n_x and n_u are varied. The condensing method is roughly quadratic in n_x, and the value of n_u is influencing the computational time much more than in the case of the methods presented in the previous chapter. In fact, the condensing method seems to be linear in n_u for large values of n_x, and faster (in theory quadratic) for small values of n_x.

If the number of inputs n_u and the horizon length N have modest values, the condensing method is faster than Riccati recursion methods, especially for large values of the number of states n_x.



(a) $N = 10$ fixed, $n_x \in \{4, 8, 16, 32, 64, 128, 256, 512\}$ and $n_u \in \{1, 2, 4, 8\}$ varying.



(b) $N = 100$ fixed, $n_x \in \{4, 8, 16, 32, 64, 128, 256, 512\}$ and $n_u \in \{1, 2, 4, 8\}$ varying.

Figure 6.2: Comparison of the performances of our implementations of algorithm 3 (blue) and 6 (red). In the case of algorithm 3, the plots are only for $n_u = 1$, since the results are almost the same for the other values. The test problem is the mass-spring problem 7.2.

Chapter 7

Test problems as Extended Linear Quadratic Control Problem

In this chapter we present a test problem as an instance of the extended linear quadratic control problem, and compare the solvers derived in the previous chapters.

7.1 Mass-spring system

7.1.1 Problem definition

We use the mass-spring system as test problem for our implementations of the different methods for the solution of problem 2.1. The mass-spring system consists in a chain of p masses (all of value 1) connected by springs (all of constant 1); furthermore a spring connects the first mass to a wall, and another spring connects the last mass to another wall. m forces (inputs) acts on the first m masses. This test problem is actually an instance of 2.2, but the solvers developed for 2.1 will be used for its solution.

The system can be modeled as the system of p second order linear differential equations

$$\begin{array}{rcll} \ddot{q}_1 &=& -2q_1 + q_q + f_1 & \\ \ddot{q}_i &=& q_{i-1} - 2q_i + q_{i+1} + f_i, & i = 2, \dots, m \\ \ddot{q}_i &=& q_{i-1} - 2q_i + q_{i+1}, & i = m+1, \dots, p-1 \\ \ddot{q}_p &=& q_{p-1} - 2q_p & \end{array}$$

where q_i is the deviation from the equilibrium point of the i-th mass and f_i is the force acting on the i-th mass.

This system can be transformed into a system of $2p$ first order linear differential equations introducing the speed $v_i = \dot{q}_i$:

$$\begin{array}{rcll}
\dot{q}_i &=& v_i, & i = 1, \dots, p \\
\dot{v}_1 &=& -2q_1 + q_q + f_1 & \\
\dot{v}_i &=& q_{i-1} - 2q_i + q_{i+1} + f_i, & i = 2, \dots, m \\
\dot{v}_i &=& q_{i-1} - 2q_i + q_{i+1}, & i = m+1, \dots, p-1 \\
\dot{v}_p &=& q_{p-1} - 2q_p &
\end{array}$$

We choose as states vector the vector $x(t) \in \mathbb{R}^{2p}$, where the first p components are the deviations $q_i(t)$ from the equilibrium point of the p masses at time t, and the last p components are the speeds $v_i(t)$ of the masses at time t. We choose as inputs vector the vector $u(t) \in \mathbb{R}^m$ of the forces, and as output the vector $y(t) \in \mathbb{R}^p$ consisting in the first p components of $x(t)$. With these definitions, the system can be rewritten in more compact form as the linear continuous time invariant state space system

$$\begin{aligned} \dot{x}(t) &= A_c x(t) + B_c u(t) \\ y(t) &= Cx(t) \end{aligned} \tag{7.1}$$

where

$$x(t) = \begin{bmatrix} q_1 \\ \vdots \\ q_p \\ v_1 \\ \vdots \\ v_p \end{bmatrix}, \quad u(t) = \begin{bmatrix} f_1 \\ \vdots \\ f_m \end{bmatrix}, \quad y(t) = \begin{bmatrix} q_i \\ \vdots \\ q_p \end{bmatrix}$$

$$A_c = \begin{bmatrix} 0_{p\times p} & I_{p\times p} \\ T_{p\times p} & 0_{p\times p} \end{bmatrix}, \quad B_c = \begin{bmatrix} 0_{p\times m} \\ I_{m\times m} \\ 0_{(p-m)\times m} \end{bmatrix}, \quad C = \begin{bmatrix} I_{p\times p} & 0_{p\times p} \end{bmatrix}$$

and $T_{p\times p}$ is the $p \times p$ tridiagonal matrix

$$T_{p\times p} = \begin{bmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & \dots & 0 & 1 & -2 \end{bmatrix}.$$

We want to study a little more this system. The matrix A_c is invertible since it has full rank. Its eigenvalues are the zeros of the characteristic polynomial, that is

$$0 = \det(\lambda I - A_c) = \det\left(\begin{bmatrix} \lambda I & -I \\ -T & \lambda I \end{bmatrix}\right).$$

It is possible to write the above expression in a more useful way: first of all we notice that λI is invertible, since the matrix A_c is invertible and thus does not have 0 as eigenvalue. Then, since the determinant of the inverse matrix is the inverse of the determinant[1], given the matrix $M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ with A invertible, it holds the relation

$$\det(M) = \frac{1}{\det(M^{-1})} = \frac{1}{\det\left(\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1}\right)} =$$

$$= \frac{1}{\det\left(\begin{bmatrix} I & -A^{-1}B \\ 0 & I \end{bmatrix}\begin{bmatrix} A^{-1} & 0 \\ 0 & (D-CA^{-1}B)^{-1} \end{bmatrix}\begin{bmatrix} I & 0 \\ -CA^{-1} & I \end{bmatrix}\right)} =$$

$$= \frac{1}{1\cdot\det(A^{-1})\cdot\det((D-CA^{-1}B)^{-1}))\cdot 1} = \det(A)\det(D-CA^{-1}B).$$

Thus we have that the characteristic polynomial can be obtained as

$$\det\left(\begin{bmatrix} \lambda I & -I \\ -T & \lambda I \end{bmatrix}\right) = \det(\lambda I)\det(\lambda I - \lambda^{-1}T) = 0$$

that, since $\lambda \neq 0$, implies

$$\det(\lambda I - \lambda^{-1}T) = (\lambda^{-1})^p \det(\lambda^2 I - T) = 0 \quad \Rightarrow \quad \det(\lambda^2 I - T) = 0$$

where p is the size of T.

The matrix $\lambda^2 I - T$ has a lot of structure, and is (in the case $p = 3$)

$$\lambda^2 I - T = \begin{bmatrix} \lambda^2+2 & -1 & 0 \\ -1 & \lambda^2+2 & -1 \\ 0 & -1 & \lambda^2+2 \end{bmatrix}$$

and its determinant can be written recursively as

$$\det((\lambda^2 I - T)_p) = (\lambda^2+2)\det((\lambda^2 I - T)_{p-1}) - \det((\lambda^2 I - T)_{p-2})$$

with base cases

$$\det((\lambda^2 I - T)_1) = \lambda^2 + 2 \quad \text{and} \quad \det((\lambda^2 I - T)_2) = (\lambda^2+2)^2 - 1.$$

Furthermore, we notice that the system is controllable for each value of p. We show this fact in the case $p = 3$ (the generalization being trivial), and only one input $m = 1$. The first 5 powers of the matrix A_c take the form:

$$A_c = \begin{bmatrix} 0 & I \\ T & 0 \end{bmatrix},\ A_c^2 = \begin{bmatrix} T & 0 \\ 0 & T \end{bmatrix},\ A_c^3 = \begin{bmatrix} 0 & T \\ T^2 & 0 \end{bmatrix},\ A_c^4 = \begin{bmatrix} T^2 & 0 \\ 0 & T^2 \end{bmatrix},\ A_c^5 = \begin{bmatrix} 0 & T^2 \\ T^3 & 0 \end{bmatrix}$$

and so on. Because of the shape of the matrix B_c, the controllability matrix is

$$\begin{bmatrix} B_c & A_cB_c & A_c^2B_c & A_c^3B_c & A_c^4B_c & A_c^5B_c \end{bmatrix} =$$
$$= \begin{bmatrix} 0 & col_1(I) & 0 & col_1(T) & 0 & col_1(T^2) \\ col_1(I) & 0 & col_1(T) & 0 & col_1(T^2) & 0 \end{bmatrix}$$

[1]In fact, given M and its inverse M^{-1}, we have $1 = \det(I) = \det(MM^{-1}) = \det(M)\det(M^{-1})$.

and it has full rank $2p = 6$ since in general (for $k < p$)

$$col_1(T^k) = \begin{bmatrix} * \\ \vdots \\ * \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

with the 1 in position of index $k+1$ (where the index starts from 1).

System 7.1 can be discretized with sampling time $T_s = 1$, obtaining the linear discrete time invariant system

$$\begin{aligned} x_{k+1} &= A_d x_k + B_d u_k \\ y_k &= C_d x_k \end{aligned}$$

The problem can be written in the form of problem 2.1 as:

$$\begin{aligned} \min_{\{u_n, x_{n+1}\}} \quad & \phi = \sum_{n=0}^{N-1} \left(\frac{1}{2} \begin{bmatrix} x'_n & u'_n \end{bmatrix} \begin{bmatrix} Q_n & S'_n \\ S_n & R_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \begin{bmatrix} q'_n & s'_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \rho_n \right) + \\ & + \frac{1}{2} x'_N Q_N x_N + q'_N x_N + \rho_N \\ s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \end{aligned} \tag{7.2}$$

where

$$\begin{aligned} Q_n = I_{2p\times 2p}, \quad S_n = 0_{m\times 2p}, \quad R_n = I_{m\times m}, \\ q_n = 0_{2p\times 1}, \quad s_n = 0_{m\times 1}, \quad \rho_n = 0, \\ A_n = A_d, \quad B_n = B_d, \quad b_n = 0_{2p\times 1}, \end{aligned}$$

for each $n \in \{0, 1, \ldots, N-1\}$ (and $n = N$ for Q_n, q_n, ρ_n).

7.1.2 Small size example

In this part we want to study a small instance of problem 7.2. Choosing $p = 2$ masses (and thus $n_x = 2p = 4$), $m = 1$ forces (and thus $n_u = m = 1$), $N = 20$ as horizon length, and initial state vector $x_0 = \begin{bmatrix} 5 & 10 & 15 & 20 \end{bmatrix}$, we have the continuous time system

$$\dot{x}(t) = A_c x(t) + B_c u(t)$$

of matrices

$$A_c = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -2 & 1 & 0 & 0 \\ 1 & -2 & 0 & 0 \end{bmatrix}, \quad B_c = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}.$$

The eigenvalues of the matrix A_c are given by the zeros of the function

$$(\lambda^2 + 2)^2 - 1 = 0, \quad \Rightarrow \quad \lambda_{1,2} = \pm i, \lambda_{3,4} = \pm i\sqrt{3},$$

that are two couples of complex-conjugate numbers with zero real part. The two associate modes are persistent (not converging nor diverging) and precisely sinusoids of periods $T_1 = 2\pi$ and $t_2 = \frac{2\pi}{\sqrt{3}}$. Since the two periods are irrational, the states of the system are limited non-periodic functions of the time, and precisely sums of sinusoids: this is confirmed by the impulsive response of the system in figure 7.1a.

The discrete time system is obtained by sampling the continuous time system with sampling time $T_s = 1$. Remembering that the general solution at time t is

$$x(t) = e^{A_c(t-t_0)}x(t_0) + \int_{t_0}^{t} e^{A_c(t-\tau)}B_c u(\tau)d\tau,$$

and that the system matrices are constant, and assuming the input to be constant between two sampling times, we have the expression of the discrete time system

$$\begin{aligned} x_{n+1} = x(t_{n+1}) &= e^{A_cT_s}x(t_n) + \int_{t_n}^{t_{n+1}} e^{A_c(t_{n+1}-\tau)}B_c u(\tau)d\tau = \\ &= e^{A_cT_s}x_n + \int_{0}^{T_s} e^{A_ct}dtB_cu_n = A_dx_n + B_du_n. \end{aligned}$$

Since the matrix A_c is invertible, the matrices of the discrete time system can be found as

$$A_d = e^{A_cT_s} \quad \text{and} \quad B_d = A_c^{-1}(e^{A_cT_s} - I)B_c.$$

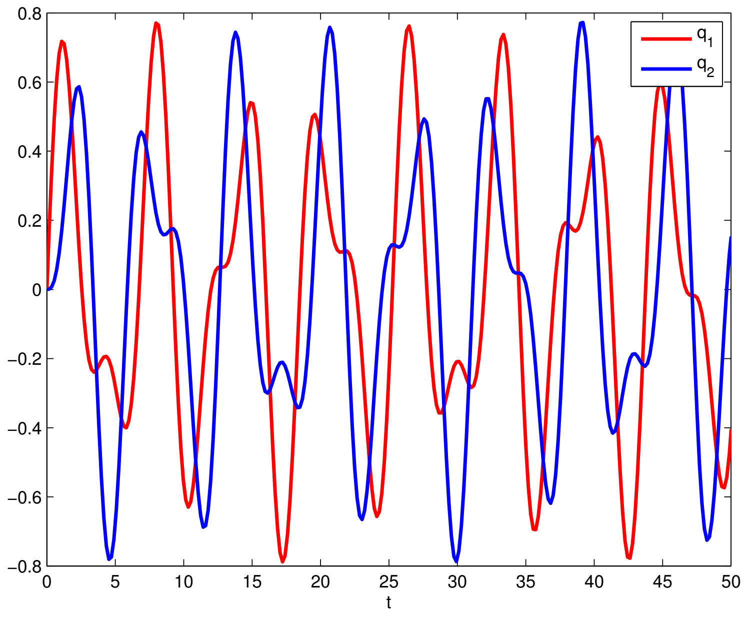
Otherwise, A_d and B_d can be found as the top left and top right sub-matrices of the matrix

$$E = \exp\left(\begin{bmatrix} A_cT_s & B_cT_s \\ 0_{1\times 2} & 0_{1\times 1} \end{bmatrix}\right).$$

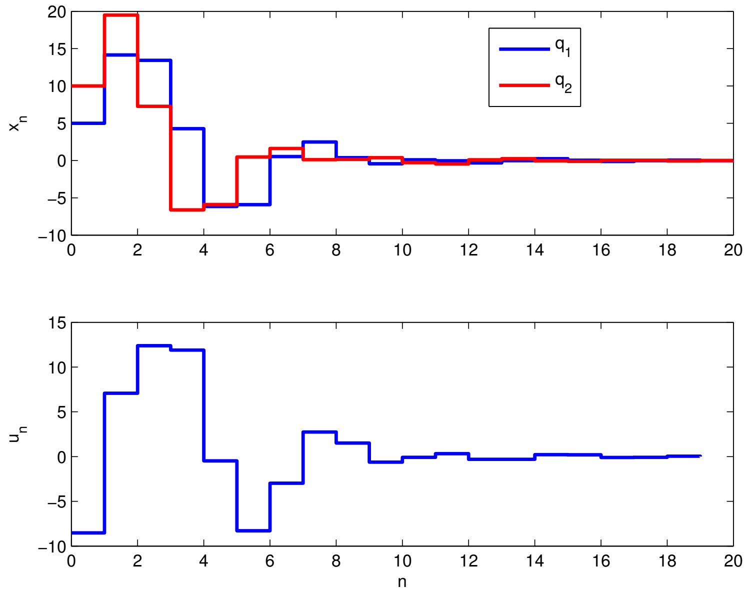
Anyway, the matrices of the discrete time system are

$$A_d = \begin{bmatrix} 0.1899 & 0.3504 & 0.7057 & 0.1358 \\ 0.3504 & 0.1899 & 0.1358 & 0.7057 \\ -1.2755 & 0.4341 & 0.1899 & 0.3504 \\ 0.4341 & -1.2755 & 0.3504 & 0.1899 \end{bmatrix}, \quad B_d = \begin{bmatrix} 0.4233 \\ 0.0364 \\ 0.7057 \\ 0.1358 \end{bmatrix}$$

In figure 7.1b there is a plot of the solution of problem 7.2 in the case $n_x = 4$, $n_u = 1$ and $N = 20$, computed using one of the developed methods. We can see that the states are correctly controlled to zero in the control horizon time.



(a) Impulsive response of continuous time system 7.1.



(b) States 1 and 2, and input computed using one of the developed methods. The test problem is 7.2 with $n_x = 4$, $n_u = 1$ and $N = 20$.

Figure 7.1

7.2 Comparison tests

In this section we perform a few tests to compare the performances of the solution methods derived in the previous chapters. In a first part we compare the methods in the solution of problem 2.1 on its own. In a second part we compare the methods in the solution of problem 2.1 as a subproblem in an interior-point method: in this case, some quantities needs to be computed only once for a sequence of problems.

7.2.1 Comparison in the solution of 2.1 on its own

In this part we compare the performances of our implementation of the methods presented in the previous chapters in the solution of test problem 7.2. Two series of tests are performed, one keeping $n = 10$ and the other keeping $N = 100$.

For all methods but condensing one, we choose to keep n_u fixed to 1, since all the tests performed in the previous chapters show that its value has very little influence on the computational time, as long as the condition $n_u \ll n_x$ holds. On the contrary, in the case of condensing method, the value of u_n greatly influences the computation time.

In figure 7.2a there is a plot of the tests for $N = 10$. It is clear that there are two classes of methods.

The first class contains direct sparse solvers, Schur complement method and Riccati recursion methods. All methods in this class have an asymptotic complexity of $N(n_x + n_u)^3$ and have a similar behavior: their curve is just shifted up or down. Direct sparse solvers have the worst performances, since they are not tailored for the specific problem 2.1. Riccati recursion method, in its different versions, has the best performance. Schur complement method is in the middle, but our current implementation crashes for large systems (in this case for $n_x = 512$ or larger).

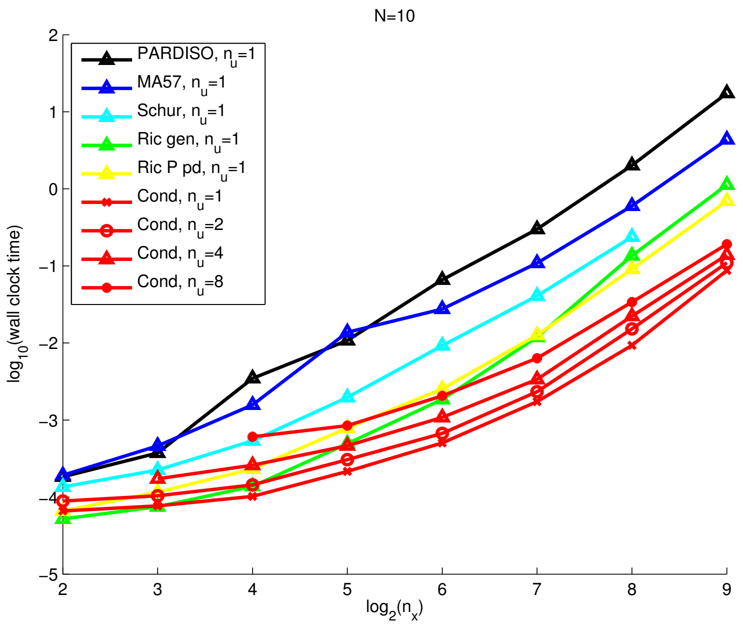
The second class contains only the condensing method, that has an asymptotic complexity of $N^2 n_x^2 n_u + N^2 n_x n_u^2 + (N n_u)^3$. The computation cost is quadratic in n_x, and then the method is suitable for large values of n_x: in this case, the first term in the complexity expression is usually dominant. In the approximation of $n_x \gg n_u$, the asymptotic complexity is $N^2 n_x^2 n_u$, quadratic in both N and n_x, and linear in n_u.

In conclusion, the best method has to be chosen between Riccati recursion methods and condensing method, depending on the problem size.

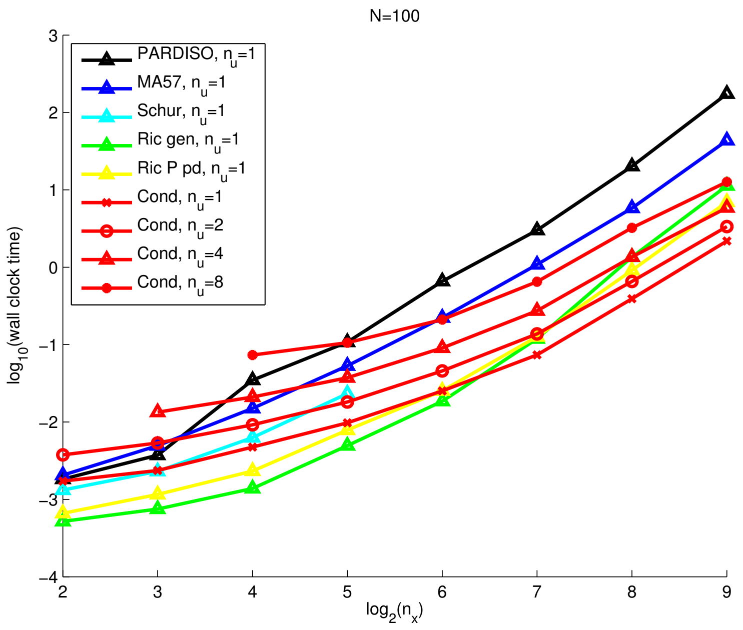
7.2.2 Comparison in the solution of 2.1 as subproblem in an interior-point method

In this case, a sequence of problems has to be solved, one for each iteration of the interior-point method: all of them have the same A_n, B_n and b_n matrices (the matrices describing the dynamic of the system), while the matrices of the cost function vary among iterations. This is showed in detail in chapter 9.

The condensing method presented in chapter 6 can exploit this: in fact, the Γ_u and $\Gamma_b b$ matrices are constant among the iterations of the interior-point method, and then can be computed only once for the entire sequence of problems, saving approximately $n_x^2 n_u N^2$ floating-point operations in the following iterations of the interior-point method.



(a) $N = 10$.



(b) $N = 100$.

Figure 7.2: Comparison of all methods presented in the previous chapters: PARDISO (black); MA57 (blue); Schur complement method (cyan); Riccati recursion methods, general form (green) and form assuming P_n positive definite (yellow); condensing method (red). Test problem is 7.2.

In this part we compare again the Riccati recursion method and the condensing method as sub-routines in an interior-point method: the computation time of the Riccati recursion method is exactly the same, while in the case of the condensing method we compute Γ_u and $\Gamma_b b$ off-line.

Comparing the plot in figure 7.3 with the one in figure 6.2, we notice that the trend of the computation time of the condensing method is the same in both pictures, but that it is slightly lower in figure 7.3: then, in case of use of the solver as a routine in an interior-point method, it is slightly wider the range of problem sizes for which the condensing method is the fastest.

7.2.3 Comparison between the C and MATLAB implementations of the same algorithm

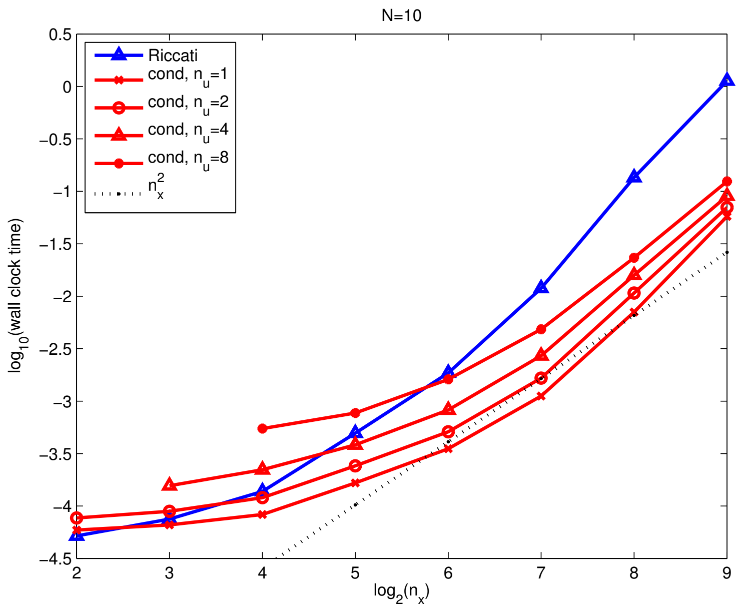
In this part we want to compare the performance of the same algorithm implemented in C and MATLAB: we use the Riccati recursion method presented in algorithm 3, since it can be implemented in MATLAB code without use of MEX files.

The comparison is justified by the fact that both our C code and MATLAB use the same BLAS version: then the test can show the efficiency of the two languages. The tables with the results are in appendix D.5; a picture of the computation time for $N = 10$, $n_u = 1$ and $n_x \in \{10, 20, 40, 60, 80, 100, 140, 200\}$ is in figure 7.4.

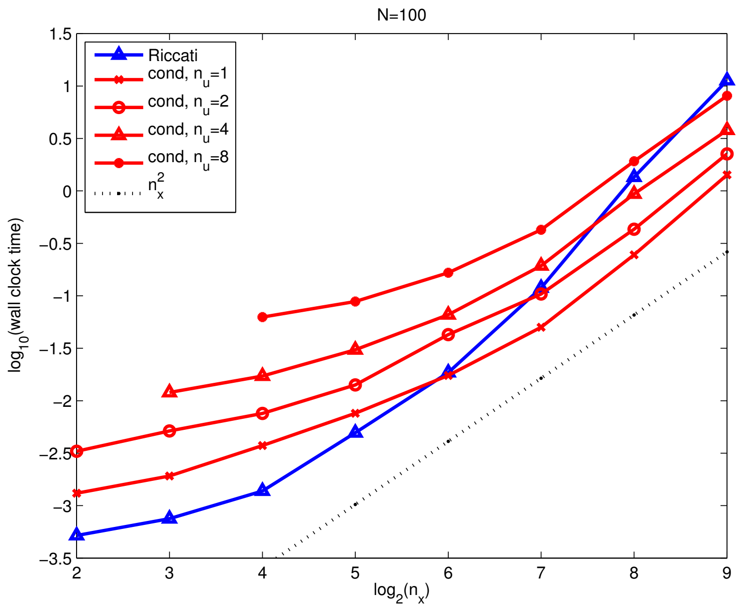
For small systems the MATLAB implementation is roughly an order of magnitude slower than the C one. As the size of the problem increases, the ratio between the MATLAB implementation time and the C implementation time gets closer to 1.

This shows that the code written in MATLAB is slow, and then programs working on small matrices are slow as well. Anyway, MATLAB uses the optimized MKL BLAS for the performance of matrix operations, and then programs working with large matrices are fast, since most of the computation time is spent in the optimized BLAS routines.

This test also shows the importance of the BLAS implementation used to perform matrix operations: a program written in C code and making use of a rather slow BLAS implementation may turn out to be slower than the same program written in MATLAB and making use of an efficient BLAS implementation.



(a) $N = 10$ fixed, $n_x \in \{4, 8, 16, 32, 64, 128, 256, 512\}$ and $n_u \in \{1, 2, 4, 8\}$ varying.



(b) $N = 100$ fixed, $n_x \in \{4, 8, 16, 32, 64, 128, 256, 512\}$ and $n_u \in \{1, 2, 4, 8\}$ varying.

Figure 7.3: Comparison of the performances of our implementations of algorithm 3 (blue) and 6 (red) as sub-routine in an interior-point method. In the case of algorithm 3, the plots are only for $n_u = 1$.

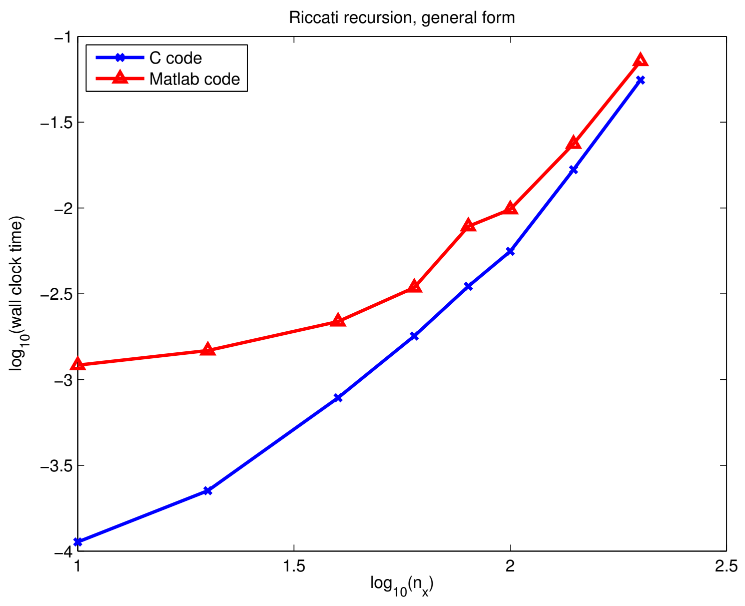


Figure 7.4: Comparison of the performances of our implementations of algorithm 3 in C code (blue) and MATLAB code (red). The test problem is problem 7.2, with $n_x \in \{10, 20, 40, 60, 80, 100, 140, 200\}$, $n_u = 1$ and $N = 10$.

Chapter 8

Extended Linear Quadratic Control Problem with Inequality Constraints

In this chapter we take into consideration an extension of problem 2.1, including also inequality constraints together with equality constraints.

8.1 Definition

In this section we define the extended linear quadratic control problem with inequality constraints, an extension of problem 2.1. The problem is defined as

Problem 3. *The extended linear quadratic control problem with inequality constraints is the quadratic program with equality and inequality constrains*

$$\begin{aligned}
\min_{u_n, x_{n+1}} \quad & \phi = \sum_{n=0}^{N-1} l_n(x_n, u_n) + l_N(x_N) \\
s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \\
& C_n u_n + D_n x_n \geq d_n \\
& D_N x_N \geq d_N
\end{aligned} \tag{8.1}$$

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

$$\begin{aligned}
l_n(x_n, u_n) &= \frac{1}{2} \begin{bmatrix} x'_n & u'_n \end{bmatrix} \begin{bmatrix} Q_n & S'_n \\ S_n & R_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \begin{bmatrix} q'_n & s'_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \rho_n \\
l_N(x_N) &= \frac{1}{2} x'_N Q_N x_N + q'_N x_N + \rho_N
\end{aligned}$$

The definition of this problem is exactly the same of problem 2.1, a part from the presence of the inequality constraints on the state and input.

8.1.1 Matrix From

Problem 8.1 can be seen as a particular case of the generic quadratic program with equality and inequality constraints,

$$\begin{aligned} \min_x \quad & \phi = \frac{1}{2}x'Hx + g'x \\ s.t. \quad & Ax = b \\ & Cx \geq d \end{aligned} \tag{8.2}$$

where the state vector is

$$x = \begin{bmatrix} u_0 \\ x_1 \\ u_1 \\ \vdots \\ x_{N-1} \\ u_{N-1} \\ x_N \end{bmatrix}$$

and the matrices relative to the cost function and the constraints are

$$H = \begin{bmatrix} R_0 & & & & & & \\ & Q_1 & S_1' & & & & \\ & S_1 & R_1 & & & & \\ & & & \ddots & & & \\ & & & & Q_{N-1} & S_{N-1}' & \\ & & & & S_{N-1} & R_{N-1} & \\ & & & & & & Q_N \end{bmatrix}, \quad g = \begin{bmatrix} S_0x_0 + s_0 \\ q_1 \\ s_1 \\ \vdots \\ q_{N-1} \\ s_{N-1} \\ q_N \end{bmatrix}$$

$$A = \begin{bmatrix} -B_0 & I & & & & & \\ & -A_1 & -B_1 & I & & & \\ & & & \ddots & \ddots & & \\ & & & & -A_{N-1} & -B_{N-1} & I \end{bmatrix}, \quad b = \begin{bmatrix} A_0x_0 + b_0 \\ b_1 \\ \vdots \\ b_{N-1} \end{bmatrix}$$

$$C = \begin{bmatrix} C_0 & & & & & \\ & D_1 & C_1 & & & \\ & & & \ddots & & \\ & & & & D_{N-1} & C_{N-1} & \\ & & & & & & D_N \end{bmatrix}, \quad d = \begin{bmatrix} d_0 - D_0x_0 \\ d_1 \\ \vdots \\ d_{N-1} \\ d_N \end{bmatrix}$$

8.2 Optimality Conditions

In this section we derive optimality conditions for the solution of a general quadratic program (meaning with this the quadratic program with both equality and inequality constraints). The general theory can be found in [NW06].

The first order necessary optimality conditions for x^* to be a solution of the quadratic program 8.2 are again the KKT conditions, usually derived from the Lagrangian function associated with the quadratic program.

The Lagrangian function of the quadratic program 8.2 takes the form

$$\mathcal{L} = \mathcal{L}(x, \pi, \lambda) = \frac{1}{2}x'Hx + g'x - \pi'(Ax - b) - \lambda'(Cx - d),$$

where π and λ are the vectors of the Lagrange multipliers relative to the equality and inequality constraints.

The first order necessary conditions for optimality are the KKT conditions

$$Hx^* + g - A'\pi^* - C'\lambda^* = 0 \tag{8.3a}$$
$$Ax^* - b = 0 \tag{8.3b}$$
$$Cx^* - d \geq 0 \tag{8.3c}$$
$$\lambda^* \geq 0 \tag{8.3d}$$
$$(\lambda^*)'(Cx^* - d) = 0 \tag{8.3e}$$

Because of the conditions on the sign of 8.3c and 8.3d, equation 8.3e implies that, if the i-th constraint is active, $(Cx^* - d)_i = 0$, then the relative Lagrange multiplier can be positive or null, $\lambda_i \geq 0$. But, if the i-th constraint is inactive, $(Cx^* - d)_i > 0$, then the relative Lagrange multiplier has to be null, $\lambda_i = 0$.

The KKT conditions in the form 8.3 are useful for the derivation of interior-point methods in chapter 9. Anyway, for the purposes of this chapter, we prefer another characterization.

Let us denote with $\mathcal{E}$ the set of the equality constraints, and with $\mathcal{I}$ the set of the inequality constraints. The constraints can be divided into two categories: the active set, containing all the equality constraints and the inequality constraints that are active in x^* (i.e. such that 8.3c is an equality); and the inactive set, containing all the others inequality constraints (i.e. the constraints that are inactive in x^*, such that 8.3c is strictly positive). We denote the active set at the point x^* as $\mathcal{A}(x^*)$.

Furthermore, let us consider the unified notation for both the equality and inequality constraints

$$Ax - b \geq 0$$

where the matrix A and the vector b contains the coefficient of both the equality and the inequality constraints. Using the definition of active set, we have that, at the point x,

$$\begin{aligned} (Ax - b)_i &= 0 \quad \text{for} \quad i \in \mathcal{A}(x) \\ (Ax - b)_i &> 0 \quad \text{for} \quad i \notin \mathcal{A}(x) \end{aligned}$$

The Lagrangian function can be rewritten as

$$\mathcal{L} = \mathcal{L}(x, \lambda) = \frac{1}{2}x'Hx + g'x - \lambda'(Ax - b)$$

and the KKT conditions can be rewritten as

$$Hx^* + g - \sum_{i \in \mathcal{A}(x^*)} A_i'\lambda_i^* = 0 \tag{8.4a}$$
$$(Ax^* - b)_i = 0 \quad \text{for } i \in \mathcal{A}(x^*) \tag{8.4b}$$
$$(Ax^* - b)_i > 0 \quad \text{for } i \notin \mathcal{A}(x^*) \tag{8.4c}$$
$$\lambda_i^* \geq 0 \quad \text{for } i \in \mathcal{I} \cap \mathcal{A}(x^*) \tag{8.4d}$$

where A_i is the i-th row of the matrix A. The meaning of 8.4a is that $Hx^* + g$, the gradient of the cost function in x^*, is orthogonal to the constraints in the active set. Conditions 8.4b and 8.4c imply that the point x^* is feasible. Condition 8.4d assure that the gradient of the cost function points towards the growing half-hyperplane of each inequality constraint in the active set.

Proposition 8 (First order necessary conditions)**.** *If x^* is a solution for the quadratic program 8.2, then there exists a vector λ^* such that the KKT conditions 8.4 hold.*

Proof. We show that, if one or more of the conditions 8.4 is not satisfied, then the point x^* is not a solution for 8.2.

If the point x^* does not satisfy 8.4b or 8.4c, it is not a feasible point, and then it is not a solution.

If condition 8.4a is not satisfied, we can repeat the proof of proposition 1 and find a step $\Delta x = x - x^*$ along the active constraints such that $\phi(x) < \phi(x^*)$. As matrices A and b in the proof of 1 we use the matrices of the constraints in the active set.

If condition 8.4d is not satisfied, then $\lambda_i^* < 0$ for at least one index $i \in \mathcal{A}(x^*)$. Let us assume that $\lambda_j^* < 0$. The idea of the proof is that it is possible to find a small step moving away from the j-th constraint while keeping on the other active constraints, and arriving in a feasible point. For a small length of the step, the cost function decreases.

Let us see the details. Let x be a feasible point such that $A_i x - b_i = 0$ for $i \in \mathcal{A}(x^*), i \neq j$, and $A_j x - b_j > 0$. Let us define the direction $\Delta x = x - x^*$, and $\delta x = \Delta x \epsilon$, $\epsilon > 0$ a small step in the direction Δx. Then $A_i \delta x = (A_i x - A_i x^*)\epsilon = (b_i - b_i)\epsilon = 0$ for $i \in \mathcal{A}(x^*), i \neq j$, and $A_j \delta x = (A_j x - A_j x^*)\epsilon > (b_i - b_i)\epsilon = 0$.

The projection of the gradient of the cost function along the step δx is

$$\delta x'(Hx^* + g) = \sum_{i \in \mathcal{A}(x^*), i \neq j} \delta x' A_i' \lambda_i^* + \delta x' A_j' \lambda_j^* = 0 + \epsilon (A_j \Delta x)' \lambda_j^* = -\epsilon\beta < 0$$

The value of the cost function in the point $\hat{x} = x^* + \delta x$ is

$$\phi(\hat{x}) = \phi(x^* + \delta x) = \phi(x^*) + \delta x'(Hx^* + g) + \frac{1}{2}\delta x' H \delta x = \phi(x^*) - \beta\epsilon + \frac{1}{2}\alpha\epsilon^2$$

where $\alpha = \Delta x' H \Delta x \in \mathbb{R}$ and $\beta > 0$.

If $\alpha \leq 0$, then $\phi(\hat{x}) \leq \phi(x^*) - \beta\epsilon < \phi(x^*)$, and then x^* is not a global minimizer.

If $\alpha > 0$, the quantity $\frac{1}{2}\alpha\epsilon^2 - \beta\epsilon < 0$ for $\epsilon < \frac{2\beta}{\alpha}$: then, for $\epsilon < \frac{2\beta}{\alpha}$, it holds $\phi(\hat{x}) < \phi(x^*)$, and then x^* is not a global minimizer. □

Proposition 9 (First order sufficient conditions)**.** *If the point x^* satisfies the KKT conditions 8.4 with the Lagrangian multiplier λ^*, and the matrix H is positive semi-definite, then x^* is a global minimizer for the cost function, i.e. a solution for the quadratic program.*

Proof. Let x be any feasible point other than x^*, then $A_i x = b_i$ for $i \in \mathcal{E}$ and $A_i x \geq b_i$ for $i \in \mathcal{I} \cap \mathcal{A}(x^*)$. Then the step $\Delta x = x - x^*$ satisfies $A_i \Delta x = b_i - b_i = 0$ for $i \in \mathcal{E}$ and $A_i \Delta x = A_i x - A_i x^* \geq b_i - b_i = 0$ for $i \in \mathcal{I} \cap \mathcal{A}(x^*)$.

Let us rewrite condition 8.4a as $Hx^* + g = \sum_{i \in \mathcal{E}} A_i' \lambda_i^* + \sum_{i \in \mathcal{I} \cap \mathcal{A}(x^*)} A_i' \lambda_i^*$. Then

$$\Delta x'(Hx^* + g) = \sum_{i \in \mathcal{E}} \Delta x' A_i' \lambda_i^* + \sum_{i \in \mathcal{I} \cap \mathcal{A}(x^*)} \Delta x' A_i' \lambda_i^* \geq 0 \qquad (8.5)$$

The value of the cost function in x is

$$\begin{aligned}\phi(x) =&\phi(x^* + \Delta x) = \frac{1}{2}(x^* + \Delta x)'H(x^* + \Delta x) + (x^* + \Delta x)'g = \\ =&\phi(x^*) + \Delta x'(Hx^* + g) + \frac{1}{2}\Delta x'H\Delta x \geq \phi(x^*)\end{aligned}$$

using the positive definiteness of H and expression 8.5. □

Note The geometrical interpretation of 8.5 is that the projection of the gradient of the cost function $Hx^* + g$ on a general feasible step Δx is positive or null, and then the cost function keeps steady or increases along the direction of Δx.

Proposition 10 (First order sufficient conditions for uniqueness). *If the point x^* satisfies the KKT conditions 8.4 with the Lagrangian multiplier λ^*, and the matrix H is positive definite, then x^* is the unique global minimizer for the cost function, i.e. the unique solution for the quadratic program.*

Proof. The beginning of the proof is as in the previous proposition. The value of the cost function in x is

$$\phi(x) = \phi(x^*) + \Delta x'(Hx^* + g) + \frac{1}{2}\Delta x'H\Delta x > \phi(x^*)$$

since H is positive definite and $\Delta x \neq 0$ (and then $\Delta x'H\Delta x{>}0$) and $\Delta x'(Hx^* + g) \geq 0$. □

Proposition 11 (Second order sufficient conditions for uniqueness). *Let the matrix $\hat{A}$ be the matrix of the constraints in the active set, and let it have full row rank. Let Z be a matrix whose columns are a base for the null space of $\hat{A}$. If the point x^* satisfies the KKT conditions 8.4 with the Lagrangian multiplier λ^*, the Hessian matrix H is positive semi-definite and the reduced Hessian matrix $Z'HZ$ is positive definite, then x^* is the unique global minimizer, i.e. the unique solution for the quadratic program.*

Proof. Let x^* be a point satisfying the KKT conditions 8.4 together with the Lagrangian multiplier λ^*, and let x be any other feasible point.

If x is on the active constraints, then $A_i x = b_i$ for $i \in \mathcal{A}(x^*)$. Let us define $\Delta x = x - x^* \neq 0$, then $A_i\Delta x = A_i x - A_i x^* = b_i - b_i = 0$ for $i \in \mathcal{A}(x^*)$, and then the vector Δx is in the null space of the matrix $\hat{A}$, and can be written as $\Delta x = Zy$, and $y \neq 0$ since Y has full column rank.

The projection of the gradient of the cost function on the vector Δx is

$$\Delta x'(Hx^* + g) = \sum_{i\in\mathcal{A}(x^*)} \Delta x' A_i' \lambda_i^* = 0.$$

The value of the cost function in x is

$$\begin{aligned}\phi(x) = \phi(x^* + \Delta x) = \phi(x^*) + \Delta x'(Hx^* + g) + \frac{1}{2}\Delta x'H\Delta x = \\ = \phi(x^*) + \frac{1}{2}\Delta x'H\Delta x > \phi(x^*)\end{aligned}$$

since H is positive definite and $y \neq 0$.

If x is not on the active constraints, let us assume that x is not on the j-th constraint. This means $A_j x > b_j$, while for the other constraints $A_i x \geq b_i$, $i \in \mathcal{A}(x^*), i \neq j$. Let us define the step $\Delta x = x - x^*$, then $A_j \Delta x > 0$ and $A_i \Delta x \geq 0$, $i \in \mathcal{A}(x^*)$.

The projection of the gradient of the cost function on the step Δx is

$$\delta x'(Hx^* + g) = \Delta x' A_j' \lambda_j^* + \sum_{i \in \mathcal{A}(x^*), i \neq j} \Delta x' A_i' \lambda_i^* \geq \Delta x_j' A_j' \lambda_j^* > 0$$

The value of the cost function in x is

$$\begin{aligned} \phi(x) = \phi(x^* + \Delta x) = \phi(x^*) + \Delta x'(Hx^* + g) + \frac{1}{2}\Delta x' H \Delta x \geq \\ \geq \phi(x^*) + \Delta x'(Hx^* + g) > \phi(x^*) \end{aligned}$$

This proves that x^* is the global minimizer. □

Chapter 9

Interior-point methods

In this chapter we present two interior-point methods for the solution of quadratic programs with both equality and inequality constraints: a basic primal-dual method, and one of the best interior-point methods in practice, namely the Mehrotra predictor-corrector method. One important result is that, thanks to the specific form of the inequality constraints, problem 2.1 arises as a subproblem in an interior-point method for the solution of 8.1.

9.1 Derivation

The presentation of interior-point methods for the solution of a general quadratic program can be found also in [NW06].

In this section we derive two interior-point methods for the solution of the quadratic program 8.2:

$$\begin{aligned} \min_x \quad & \frac{1}{2}x'Hx + g'x \\ & Ax = b \\ & Cx \geq d \end{aligned}$$

The Mehrotra predictor-corrector method is a development of the basic primal-dual method: the derivation process for the basic method then coincides with the first part of the derivation process for the Mehrotra's one.

9.1.1 Basic primal-dual method

The algorithm is a solution strategy for the KKT conditions 8.3. Introducing the vector of the slack variables $t = Cx - d \geq 0$ (of length l, the number of inequality constraints), they can be rewritten as

$$Hx + g - A'\pi - C'\lambda = 0 \tag{9.1a}$$
$$Ax - b = 0 \tag{9.1b}$$
$$Cx - d - t = 0 \tag{9.1c}$$
$$\lambda' t = 0 \tag{9.1d}$$
$$(\lambda, t) \geq 0 \tag{9.1e}$$

that is a non-linear system of equations, due to the 9.1d, with the further requirement on the sign of λ and t 9.1e. Because of 9.1e, equation 9.1d is equivalent to $\lambda_i t_i = 0$ for each i, and thus it can be rewritten as

$$\begin{bmatrix} \lambda_1 t_1 \\ \vdots \\ \lambda_l t_l \end{bmatrix} = \Lambda T e = 0$$

where Λ and T are the diagonal matrices whole diagonal element are the components of vectors λ and t, and e is a vector of length l of just ones,

$$\Lambda = \begin{bmatrix} \lambda_0 & & & \\ & \lambda_1 & & \\ & & \ddots & \\ & & & \lambda_{l-1} \end{bmatrix}, \quad T = \begin{bmatrix} t_0 & & & \\ & t_1 & & \\ & & \ddots & \\ & & & t_{l-1} \end{bmatrix}, \quad e = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}.$$

We can use the Newton method to find the solution of the system 9.1, that is an iterative method to find the zeros of the non-linear function $f(x)$, or equivalently the solutions of the system $f(x) = 0$. It can be derived easily as follows: if x^* is a solution (and so $f(x^*) = 0$) and x_k the current guess for the solution, and we suppose to reach the solution x^* in one step, the Taylor expansion of $f(x^*)$ around x_k gives:

$$0 = f(x^*) \simeq f(x_k) + \nabla f(x_k)(x^* - x) = f(x_k) + \nabla f(x_k)\Delta x$$

where $\nabla f(x_k)$ is the Jacobian matrix of the function $f(x)$ computed at the point $x = x_k$. The Newton step is then

$$\Delta x = -\nabla f(x_k)^{-1} f(x_k).$$

In our case, we have the non-linear function

$$f(x, \pi, \lambda, t) = \begin{bmatrix} Hx - A'\pi - C'\lambda + g \\ Ax - b \\ Cx - t - d \\ \Lambda T e \end{bmatrix},$$

whose Jacobian matrix is

$$\nabla f(x, \pi, \lambda, t) = \begin{bmatrix} H & -A' & -C' & 0 \\ A & 0 & 0 & 0 \\ C & 0 & 0 & -I \\ 0 & 0 & T & \Lambda \end{bmatrix}.$$

The pure Newton step is given by the solution of the linear system

$$\begin{bmatrix} H & -A' & -C' & 0 \\ A & 0 & 0 & 0 \\ C & 0 & 0 & -I \\ 0 & 0 & T_k & \Lambda_k \end{bmatrix} \begin{bmatrix} \Delta x_{\text{aff}} \\ \Delta \pi_{\text{aff}} \\ \Delta \lambda_{\text{aff}} \\ \Delta t_{\text{aff}} \end{bmatrix} = - \begin{bmatrix} H x_k - A'\pi_k - C'\lambda_k + g \\ A x_k - b \\ C x_k - t_k - d \\ \Lambda_k T_k e \end{bmatrix}, \tag{9.2}$$

where using the subscript k we want to stress the quantities depending on the current guess $(x_k, \pi_k, \lambda_k, t_k)$. This step is also the predictor (or affine-scaling) step of the Mehrotra predictor-corrector method.

The new iterate is computed with a linear search along the Newton step,

$$(x_{k+1}, \pi_{k+1}, \lambda_{k+1}, t_{k+1}) = (x_k, \pi_k, \lambda_k, t_k) + \gamma \alpha_{\text{aff}} (\Delta x_{\text{aff}}, \Delta \pi_{\text{aff}}, \Delta \lambda_{\text{aff}}, \Delta t_{\text{aff}})$$

where α_{aff} is chosen to take the longest step possible in the direction of the Newton step without violating the condition $(\lambda, t) \geq 0$,

$$\alpha_{\text{aff}} = \max\{\alpha \in [0,1] \mid (\lambda, t) + \alpha(\Delta \lambda_{\text{aff}}, \Delta t_{\text{aff}}) \geq 0\},$$

and $\gamma \in [0, 1)$ is a scalar close to one, ensuring that the conditions $(\lambda, t) > 0$ hold strictly.

The quantity

$$\mu_k = \frac{\lambda_k' t_k}{l},$$

that is the average value of the dot product between λ and t, is called the duality gap, and is often used as a measure of the optimality of the current guess $(x_k, \pi_k, \lambda_k, t_k)$.

In algorithm 6 it is summarized the basic interior-point method.

Algorithm 6 Basic primal-dual interior-point method

```
Initialize $(x_0, \pi_0, \lambda_0, t_0)$
Compute $\mu_0 = \frac{\lambda_0' t_0}{l}$
while $\mu_k > \mu_{\min}$ and $k < k_{\max}$ do
    Compute $(\Delta x_{\text{aff}}, \Delta \pi_{\text{aff}}, \Delta \lambda_{\text{aff}}, \Delta t_{\text{aff}})$ solving 9.2
    Compute $\alpha_{\text{aff}} = \max\{\alpha \in [0,1] | (\lambda, t) + \alpha(\Delta \lambda_{\text{aff}}, \Delta t_{\text{aff}}) \geq 0\}$
    Compute $(x_{k+1}, \pi_{k+1}, \lambda_{k+1}, t_{k+1}) = (x_k, \pi_k, \lambda_k, t_k)+$
                                    $+\gamma \alpha_{\text{aff}} (\Delta x_{\text{aff}}, \Delta \pi_{\text{aff}}, \Delta \lambda_{\text{aff}}, \Delta t_{\text{aff}})$
    Compute $\mu_k = \frac{\lambda_k' t_k}{l}$
end while
```

9.1.2 Mehrotra's predictor-corrector method

In this part we present a development of algorithm 6, leading to the Mehrotra's predictor-corrector method.

The Newton step computed in the basic algorithm is the predictor step in the Mehrotra's method. We notice that, using the affine step and the fact that $\Delta \Lambda_{\text{aff}} e = \lambda_{\text{aff}}$ and $\Delta T_{\text{aff}} e = t_{\text{aff}}$, the value of the equation 9.1d becomes

$$(\Lambda_k + \Delta \Lambda_{\text{aff}})(T_k + \Delta T_{\text{aff}})e = \Lambda_k T_k e + \Lambda_k t_{\text{aff}} + T_k \lambda_{\text{aff}} + \Delta \Lambda_{\text{aff}} \Delta T_{\text{aff}} e = \Delta \Lambda_{\text{aff}} \Delta T_{\text{aff}} e$$

instead of zero. We can thus take into account this second order term to compute a correction term.

It may also be an advantage to use a less aggressive approach and replace 0 with a positive quantity in the equation 9.1d. This may ensure the quantities λ_k and t_k to be strictly positive, and thus satisfy the constraint 9.1e strictly (this gives the name to the methods, since λ_k and t_k are in the interior of the quadrant $(\lambda, t) \geq 0$). The choice of the quantity $\mu_k e$ gives a vector with the same value of the measure parameter μ_k, but pointing toward the center of the so called central path (a region leading toward the solution). A centering parameter $\sigma \in [0, 1)$ is used to control the importance of this centering term.

For the choice of the parameter σ, it is usually used the following heuristic: first of all it is computed the maximum length of the step that can be taken along the affine-scaling direction without violating the constraints 9.1e as

$$\alpha_{\text{aff}} = \max\{\alpha \in [0,1] \mid (\lambda, t) + \alpha(\Delta\lambda_{\text{aff}}, \Delta t_{\text{aff}}) \geq 0\},$$

and it is then computed the duality gap of the relative point as

$$\mu_{\text{aff}} = \frac{(\lambda_k + \alpha_{\text{aff}}\Delta\lambda_{\text{aff}})'(t_k + \alpha_{\text{aff}}\Delta t_{\text{aff}})}{l}.$$

The value of the parameter σ is thus chosen as

$$\sigma = \left(\frac{\mu_{\text{aff}}}{\mu_k}\right)^3.$$

If the affine step leads to a great improvement toward the solution, the ratio μ_{aff}/μ_k is small, and then a small centering step is taken. On the contrary, if the affine step leads to a poor improvement toward the solution, the ratio is large, and thus a larger centering step is taken, with the hope that this will lead to larger steps toward the solution in the following iterations.

We can compute the centering-corrector part of the Mehrotra predictor-corrector method as the centering-corrector step

$$\begin{bmatrix} H & -A' & -C' & 0 \\ A & 0 & 0 & 0 \\ C & 0 & 0 & -I \\ 0 & 0 & T_k & \Lambda_k \end{bmatrix} \begin{bmatrix} \Delta x_{\text{cc}} \\ \Delta \pi_{\text{cc}} \\ \Delta \lambda_{\text{cc}} \\ \Delta t_{\text{cc}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -\Delta\Lambda_{\text{aff}}\Delta T_{\text{aff}} e + \sigma\mu_k e \end{bmatrix}, \tag{9.3}$$

where in the right hand side there are both the second order corrector and the centering terms.

The overall search direction is

$$\begin{bmatrix} \Delta x \\ \Delta \pi \\ \Delta \lambda \\ \Delta t \end{bmatrix} = \begin{bmatrix} \Delta x_{\text{aff}} \\ \Delta \pi_{\text{aff}} \\ \Delta \lambda_{\text{aff}} \\ \Delta t_{\text{aff}} \end{bmatrix} + \begin{bmatrix} \Delta x_{\text{cc}} \\ \Delta \pi_{\text{cc}} \\ \Delta \lambda_{\text{cc}} \\ \Delta t_{\text{cc}} \end{bmatrix}$$

and can be obtained directly by solving the system

$$\begin{bmatrix} H & -A' & -C' & 0 \\ A & 0 & 0 & 0 \\ C & 0 & 0 & -I \\ 0 & 0 & T_k & \Lambda_k \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \pi \\ \Delta \lambda \\ \Delta t \end{bmatrix} = - \begin{bmatrix} Hx_k - A'\pi_k - C'\lambda_k + g \\ Ax_k - b \\ Cx_k - t_k - d \\ \Lambda_k T_k e + \Delta\Lambda_{\text{aff}}\Delta T_{\text{aff}} e - \sigma\mu_k e \end{bmatrix}, \tag{9.4}$$

The length of the maximum step that can be taken in this direction without violating the constraint 9.1e is

$$\alpha_{\max} = \max\{\alpha \in [0,1] \mid (\lambda_k, t_k) + \alpha_{\max}(\Delta\lambda, \Delta t) \geq 0\}.$$

The actual step length is then chosen as

$$\alpha = \gamma\alpha_{\max}$$

with $\gamma \in (0,1)$ to ensure that the condition 9.1e holds strictly; in order to accelerate convergence, the value of γ may be chosen approaching 1 as the current iterate gets closer to the solution.

We notice that the point $(x_{k+1}, \pi_{k+1}, \lambda_{k+1}, t_{k+1}) = (x_k, \pi_k, \lambda_k, t_k) + \alpha(\Delta x_k, \Delta\pi_k, \Delta\lambda_k, \Delta t_k)$ is not feasible with respect to the equality constraints, while, on the contrary, the points $(x_{\text{aff}}, \pi_{\text{aff}}, \lambda_{\text{aff}}, t_{\text{aff}})$ and $(x_{\text{cc}}, \pi_{\text{cc}}, \lambda_{\text{cc}}, t_{\text{cc}})$ are.

The Mehrotra's predictor-corrector interior-point method is summarized in algorithm 7.

Algorithm 7 Mehrotra's predictor-corrector interior-point method

```
Initialize $(x_0, \pi_0, \lambda_0, t_0)$
Compute $\mu_0 \frac{\lambda_0' t_0}{l}$
**while** $\mu_k > \mu_{\min}$ and $k < k_{\max}$ **do**
    Compute $(\Delta x_{\text{aff}}, \Delta\pi_{\text{aff}}, \Delta\lambda_{\text{aff}}, \Delta t_{\text{aff}})$ solving 9.2
    Compute $\alpha_{\text{aff}} = \max\{\alpha \in [0,1] | (\lambda, t) + \alpha(\Delta\lambda_{\text{aff}}, \Delta t_{\text{aff}}) \geq 0\}$
    Compute $\mu_{\text{aff}} = \frac{(\lambda_k + \alpha_{\text{aff}}\Delta\lambda_{\text{aff}})'(t_k + \alpha_{\text{aff}}\Delta t_{\text{aff}})}{l}$
    Compute $\sigma = (\frac{\mu_{\text{aff}}}{\mu_k})^3$
    Compute $(\Delta x_{\text{cc}}, \Delta\pi_{\text{cc}}, \Delta\lambda_{\text{cc}}, \Delta t_{\text{cc}})$ solving 9.3
    Compute $(\Delta x_k, \Delta\pi_k, \Delta\lambda_k, \Delta t_k) = (\Delta x_{\text{aff}}, \Delta\pi_{\text{aff}}, \Delta\lambda_{\text{aff}}, \Delta t_{\text{aff}})+$
                                        $+(\Delta x_{\text{cc}}, \Delta\pi_{\text{cc}}, \Delta\lambda_{\text{cc}}, \Delta t_{\text{cc}})$
    Compute $\alpha_{\max} = \max\{\alpha \in [0,1] | (\lambda, t) + \alpha(\Delta\lambda_{\text{k}}, \Delta t_k) \geq 0\}$
    Compute $(x_{k+1}, \pi_{k+1}, \lambda_{k+1}, t_{k+1}) = (x_k, \pi_k, \lambda_k, t_k)+$
                                        $+\gamma\alpha_{\max}(\Delta x_k, \Delta\pi_k, \Delta\lambda_k, \Delta t_k)$
    Compute $\mu_{k+1} = \frac{\lambda_{k+1}' t_{k+1}}{l}$
**end while**
```

9.1.3 Computation of the steps as equality constrained quadratic programs

The most expensive part of the procedure is the computation of the predictor and centering-corrector steps. In this part we show that the computation of these two steps is equivalent to the solution of two equality constrained quadratic programs.

The two linear systems 9.2 and 9.3 have the same matrix, that can be factorized just once: then the total cost to compute the two steps is just slightly larger than the cost to compute one.

It is possible to rewrite the systems in a more useful form. Considering a system with a general right hand side in the form

$$\begin{bmatrix} H & -A' & -C' & 0 \\ A & 0 & 0 & 0 \\ C & 0 & 0 & -I \\ 0 & 0 & T_k & \Lambda_k \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\pi \\ \Delta\lambda \\ \Delta t \end{bmatrix} = \begin{bmatrix} r_H \\ r_A \\ r_C \\ r_T \end{bmatrix},$$

we can eliminate the variable Δt, obtaining

$$\begin{bmatrix} H & -A' & -C' \\ A & 0 & 0 \\ \Lambda_k C & 0 & T_k \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\pi \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} r_H \\ r_A \\ r_T + \Lambda_k r_C \end{bmatrix}.$$

Finally, by eliminating $\Delta\lambda$ and changing all the signs on the second equation, we get

$$\begin{bmatrix} H + C'T_k^{-1}\Lambda_k C & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\pi \end{bmatrix} = \begin{bmatrix} r_H + C'T_k^{-1}(r_T + \Lambda_k r_C) \\ -r_A \end{bmatrix}.$$

In particular, in order to find the predictor step we have to solve system 9.2, that can be rewritten as

$$\begin{bmatrix} H + C'T_k^{-1}\Lambda_k C & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} \Delta x_{\text{aff}} \\ \Delta\pi_{\text{aff}} \end{bmatrix} =$$
$$= \begin{bmatrix} -\big((H + C'T_k^{-1}\Lambda_k C)x_k - A'\pi_k + (g - C'(\lambda_k + T_k^{-1}\Lambda_k d))\big) \\ Ax_k - b \end{bmatrix},$$

that is the KKT system of the equality constrained quadratic program

$$\begin{aligned} \min_x \quad & \frac{1}{2}\Delta x'_{\text{aff}}(H + C'T_k^{-1}\Lambda_k C)\Delta x_{\text{aff}} + \\ & \qquad + \big(-(H + C'T_k^{-1}\Lambda_k C)\Delta x_k + g - C'(\lambda_k + T_k^{-1}\Lambda_k d)\big)'\Delta x_{\text{aff}} \\ s.t. \quad & A\Delta x_{\text{aff}} = 0 \end{aligned}$$

since the term $-A'\pi_k$ is not influencing the value of Δx_{aff}, but just the one of the Lagrangian multiplier $\Delta\pi_{\text{aff}}$.[1]

It may be convenient rewrite the KKT system to find the iterates instead of the steps, as

$$\begin{bmatrix} H + C'T_k^{-1}\Lambda_k C & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} x_{\text{aff}} \\ \pi_{\text{aff}} \end{bmatrix} = \begin{bmatrix} -\big(g - C'(\lambda_k + T_k^{-1}\Lambda_k d)\big) \\ -b \end{bmatrix},$$

(where $x_{\text{aff}} = x_k + \Delta x_{\text{aff}}$ and $\pi_{\text{aff}} = \pi_k + \Delta\pi_{\text{aff}}$) that is the KKT system of the equality constrained quadratic program

$$\begin{aligned} \min_x \quad & \frac{1}{2}x'_{\text{aff}}(H + C'T_k^{-1}\Lambda_k C)x_{\text{aff}} + (g - C'(\lambda_k + T_k^{-1}\Lambda_k d))'x_{\text{aff}} \\ s.t. \quad & Ax_{\text{aff}} = b \end{aligned} \qquad (9.5)$$

[1]In fact, the solution of the KKT system

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

can be written analiticly, using the Schur complement: π is $\pi = (AH^{-1}A')^{-1}(b + AH^{-1}g)$, and thus a term in the form $-A'y$ in g is equivalent to sum $-y$ to the right hand side; x is $Hx = A'(AH^{-1}A')^{-1}b + A'(AH^{-1}A')^{-1}AH^{-1}g - g$, and thus a term in the form $-A'y$ in g cancels itself.

Similarly, in order to find the centering-corrector step we have to solve the system 9.3, that can be rewritten as

$$\begin{bmatrix} H + C'T_k^{-1}\Lambda_k C & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} \Delta x_{\text{cc}} \\ \Delta \pi_{\text{cc}} \end{bmatrix} = \begin{bmatrix} C'T_k^{-1}(-\Delta\Lambda_{\text{aff}}\Delta T_{\text{aff}} + \sigma\mu_k e) \\ 0 \end{bmatrix},$$

that is the KKT system of the equality constrained quadratic program

$$\begin{aligned} \min_x \quad & \frac{1}{2}\Delta x_{\text{cc}}'(H + C'T_k^{-1}\Lambda_k C)\Delta x_{\text{cc}} + \big(C'T_k^{-1}(\Delta\Lambda_{\text{aff}}\Delta T_{\text{aff}} - \sigma\mu_k e)\big)'\Delta x_{\text{cc}} \\ s.t. \quad & A\Delta x_{\text{cc}} = 0 \end{aligned} \tag{9.6}$$

that, once again, has the same Hessian matrix as 9.5.

9.2 Implementation

In this section we specialize the interior-point methods presented in the previous section to the specific case of problem 8.1. The discussion of interior-point methods for the solution of model predictive control problems can be found also in [RWR98].

9.2.1 Basic primal-dual method

The first step in the application of algorithm 6 to the case of problem 8.1 is the computation of the modified Hessian matrix $\hat{H} = H + C'(T^{-1}\Lambda)C$ (where we are dropping the subscript k to keep the notation simpler). An important result is that, given the special shape of the C matrix, the modified Hessian matrix $\hat{H}$ has the exact same shape of the Hessian matrix H, namely block diagonal with block of the same size. For example, in the case $N = 3$, we have

$$C'T^{-1}\Lambda C =$$

$$= \begin{bmatrix} C_0' & & & \\ & D_1' & & \\ & C_1' & & \\ & & D_2' & \\ & & C_2' & \\ & & & D_3' \end{bmatrix} \begin{bmatrix} (T^{-1}\Lambda)_0 & & & \\ & (T^{-1}\Lambda)_1 & & \\ & & (T^{-1}\Lambda)_2 & \\ & & & (T^{-1}\Lambda)_3 \end{bmatrix} \begin{bmatrix} C_0 & & & & & \\ & D_1 & C_1 & & & \\ & & & D_2 & C_2 & \\ & & & & & D_3 \end{bmatrix} =$$

$$= \begin{bmatrix} C_0'(T^{-1}\Lambda)_0 C_0 & & & & & \\ & D_1'(T^{-1}\Lambda)_1 D_1 & D_1'(T^{-1}\Lambda)_1 C_1 & & & \\ & C_1'(T^{-1}\Lambda)_1 D_1 & C_1'(T^{-1}\Lambda)_1 C_1 & & & \\ & & & D_2'(T^{-1}\Lambda)_2 D_2 & D_2'(T^{-1}\Lambda)_2 C_2 & \\ & & & C_2'(T^{-1}\Lambda)_2 D_2 & C_2'(T^{-1}\Lambda)_2 C_2 & \\ & & & & & D_3'(T^{-1}\Lambda)_3 D_3 \end{bmatrix}.$$

Furthermore, the above matrix is symmetric positive semi-definite (since it holds $(\lambda, t) \geq 0$, and thus the matrices $(T^{-1}\Lambda)_n$ are diagonal positive semi-definite): this means that at each iteration of the interior-point method we have to solve two problems in the form 2.1, using one of the methods studied in the previous chapters.

We decide to test our implementations of the interior-point methods using algorithm 3 to solve problem 2.1, since this algorithm has shown itself to have good performances for a wide range of system size.

Algorithm 3 requires the data in the form of the matrices associated to the cost function formulation 2.1, instead of the matrix form 2.3. We thus have to compute how these matrices look like. We use for problem 2.1 the notation introduced in chapter 5.

From the previous discussion about the shape of the matrix $C'(T^{-1}\Lambda)C$, we already know that

$$\begin{aligned}
\hat{R}_0 &= R_0 + C_0'(T^{-1}\Lambda)_0 C_0 \\
\hat{R}_i &= R_i + C_i'(T^{-1}\Lambda)_i C_i && i = 1\dots, N-1 \\
\hat{S}_i &= S_i + C_i'(T^{-1}\Lambda)_i D_i && i = 1\dots, N-1 \\
\hat{Q}_i &= Q_i + D_i'(T^{-1}\Lambda)_i D_i && i = 1\dots, N-1 \\
\hat{P} &= P + D_N'(T^{-1}\Lambda)_N D_N
\end{aligned}$$

In the computation of the predictor step, we prefer to solve system 9.5 obtaining x_k instead of Δx_k, and to compute the predictor step as $\Delta x_k = x_{\text{aff}} - x_k$. The linear term of the cost function is

$$\begin{aligned}
\hat{g}_{\text{aff}} &= g - C'(\lambda + (T^{-1}\Lambda)d) = \\
&= \begin{bmatrix} S_0 x_0 + s_0 \\ q_i \\ s_1 \\ q_2 \\ s_2 \\ p_3 \end{bmatrix} - \begin{bmatrix} C_0' & & & \\ & D_1' & & \\ & C_1' & & \\ & & D_2' & \\ & & C_2' & \\ & & & D_3' \end{bmatrix} \begin{bmatrix} \lambda_0 + (T^{-1}\Lambda)_0(d_0 - D_0 x_0) \\ \lambda_1 + (T^{-1}\Lambda)_1 d_1 \\ \lambda_2 + (T^{-1}\Lambda)_2 d_2 \\ \lambda_3 + (T^{-1}\Lambda)_3 d_3 \end{bmatrix} \\
&= \begin{bmatrix} (S_0 + (T^{-1}\Lambda)_0 C_0' D_0)x_0 + s_0 - C_0'(\lambda_0 + (T^{-1}\Lambda)_0 d_0) \\ q_1 - D_1'(\lambda_1 + (T^{-1}\Lambda)_1 d_1) \\ s_1 - C_1'(\lambda_1 + (T^{-1}\Lambda)_1 d_1) \\ q_2 - D_2'(\lambda_2 + (T^{-1}\Lambda)_2 d_2) \\ s_2 - C_2'(\lambda_2 + (T^{-1}\Lambda)_2 d_2) \\ p_3 - D_3'(\lambda_3 + (T^{-1}\Lambda)_3 d_3) \end{bmatrix}
\end{aligned}$$

and thus

$$\hat{S}_0 = S_0 + (T^{-1}\Lambda)_0 C_0' D_0, \quad \hat{p}_{\text{aff}} = p_3 - D_3'(\lambda_3 + (T^{-1}\Lambda)_3 d_3)$$

and the matrices $\hat{s}_{\text{aff}}$ and $\hat{q}_{\text{aff}}$ are

$$\hat{s}_{\text{aff}} = \begin{bmatrix} s_0 - C_0'(\lambda_0 + (T^{-1}\Lambda)_0 d_0) \\ s_1 - C_1'(\lambda_1 + (T^{-1}\Lambda)_1 d_1) \\ s_2 - C_2'(\lambda_2 + (T^{-1}\Lambda)_2 d_2) \end{bmatrix}, \quad \hat{q}_{\text{aff}} = \begin{bmatrix} q_0 - D_0'(\lambda_0 + (T^{-1}\Lambda)_0 d_0) \\ q_1 - D_1'(\lambda_1 + (T^{-1}\Lambda)_1 d_1) \\ q_2 - D_2'(\lambda_2 + (T^{-1}\Lambda)_2 d_2) \end{bmatrix}.$$

We notice that from the form of $\hat{g}_{\text{aff}}$ we do not have any information about the value of $\hat{q}_{\text{aff},0}$, but this is not influencing the solution. The value of x_0 is the value of the system state at the time $n = 0$.

About the matrices relative to the equality constraints, we notice that these constraints are unchanged with respect to the problem formulation 2.1, and thus the matrices A_n, B_n, and b_n are unchanged.

In the following we show exactly how to compute the different components of the predictor step. At each iteration $k + 1$ of the interior point method, the Riccati recursion algorithm 3 is used to obtain the value of x_{aff}. The relative predictor step is obtained easily as

$$\Delta x_{\text{aff}} = x_{\text{aff}} - x_k.$$

The steps $\Delta\pi_k$ and $\Delta\lambda_k$ are obtained by computing the values π_{aff} and λ_{aff}, and by taking the differences with respect to the previous values π_k and λ_k. The value of the Lagrangian multiplier π_{aff} is obtained as

$$\pi_{\text{aff},n} = \hat{q}_{\text{aff},n} + \hat{Q}_n x_{\text{aff},n} + \hat{S}'_n u_{\text{aff},n} + A'_n \pi_{\text{aff},n+1}$$

with initial value

$$\pi_{\text{aff},N} = \hat{P} x_{\text{aff},N} + \hat{p}_{\text{aff}},$$

and the step as

$$\Delta\pi_{\text{aff}} = \pi_{\text{aff}} - \pi_k.$$

The value of t_{aff} is obtained from equation 9.1c, as

$$t_{\text{aff}} = Cx_{\text{aff}} - d = \begin{bmatrix} C_0 u_{\text{aff},0} + D_0 x_0 \quad - d_0 \\ C_1 u_{\text{aff},1} + D_1 x_{\text{aff},1} - d_1 \\ C_2 u_{\text{aff},2} + D_2 x_{\text{aff},2} - d_2 \\ D_3 x_{\text{aff},3} - d_3 \end{bmatrix},$$

and then

$$\Delta t_{\text{aff}} = t_{\text{aff}} - t_k.$$

Finally, the value of $\Delta\lambda_{\text{aff}}$ is directly obtained as

$$\Delta\lambda_{\text{aff}} = -T_k^{-1}\Lambda_k(\Delta t_{\text{aff}} + T_k e) = -T_k^{-1}\Lambda_k(\Delta t_{\text{aff}} + t_k) = -T_k^{-1}\Lambda_k t_{\text{aff}}.$$

In the following algorithm 8 we present the basic primal-dual interior-point method for the solution of problem 8.1, and we compute the complexity per iteration of the interior-point method up to quadratic terms.

The total cost per iteration, considering also the quadratic terms, is $N[(4n_x^3 + 6n_x^2 n_u + 3n_x n_u^2 + \frac{1}{3}n_u^3) + (9n_x^2 + 9n_x n_u + \frac{5}{2}n_u^2) + n_l(2n_x^2 + 2n_x n_u + 2n_u^2 + 5n_x + 5n_u)] + 2n_x^2 + n_l(2n_x^2 + 3n_x)$; the asymptotic complexity per iteration is thus

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

Using the version 5 of the Riccati recursion, and exploiting the symmetry of the matrices Q_n and R_n and the positive semi-definiteness of the diagonal matrix $T_k^{-1}\Lambda_k$, it can be lowered up to

$$N\left[\left(\frac{7}{3}n_x^3 + 4n_x^2 n_u + 2n_x n_u^2 + \frac{1}{3}n_u^3\right) + n_l\left(n_x^2 + 2n_x n_u + n_u^2\right)\right].$$

Since the number of iterations needed by the interior-point method to converge depends weakly on the problem size, we have that also the asymptotic complexity of the interior-point method is the same.

9.2.2 Mehrotra's predictor-corrector method

In the case of the Mehrotra's predictor-corrector method, we have to compute also the centering-corrector step Δx_{cc} by solving system 9.6. The linear term

Algorithm 8 Basic primal-dual interior-point method for the solution of problem 8.1

```
Require: $(\bar{x}_0, \{Q_n\}, \{S_n\}, \{R_n\}, \{q_n\}, \{s_n\}, \{A_n\}, \{B_n\}, \{b_n\}, \{C_n\}, \{D_n\}, \{d_n\}, P, p)$
  $((x_0, u_0), \pi_0, \lambda_0, t_0) \leftarrow ((0,0), 0, e, e)$
  $\mu_0 \leftarrow \frac{\lambda_0' t_0}{l}$

  while $\mu_k > \mu_{\min}$ and $k < k_{\max}$ do

    for $n = 0 \rightarrow N-1$ do
      $\hat{Q}_n \leftarrow Q_n + D_n'(T_k^{-1}\Lambda_k)_n D_n$                ▷ $N(2n_x^2 n_l + n_x n_l)$ flops
      $\hat{S}_n \leftarrow S_n + C_n'(T_k^{-1}\Lambda_k)_n D_n$                ▷ $N(2n_x n_u n_l)$ flops
      $\hat{R}_n \leftarrow R_n + C_n'(T_k^{-1}\Lambda_k)_n C_n$                ▷ $N(2n_u^2 n_l + n_u n_l)$ flops
      $\hat{q}_{\text{aff},n} \leftarrow q_n - D_n'(\lambda_n + (T_k^{-1}\Lambda_k)_n d_n)$       ▷ $N(2n_x n_l)$ flops
      $\hat{s}_{\text{aff},n} \leftarrow s_n - C_n'(\lambda_n + (T_k^{-1}\Lambda_k)_n d_n)$       ▷ $N(2n_u n_l)$ flops
    end for
    $\hat{P} \leftarrow P + D_N'(T_k^{-1}\Lambda_k) D_N$                          ▷ $2n_x^2 n_l + n_x n_l$ flops
    $\hat{p}_{\text{aff}} \leftarrow p - D_N'(\lambda_n + (T_k^{-1}\Lambda_k)_N d_N)$               ▷ $2n_x n_l$ flops

    $(x_{\text{aff}}, u_{\text{aff}}) \leftarrow \text{RiccatiSolver}(\bar{x}_0, \{\hat{Q}_n\}, \{\hat{S}_n\}, \{\hat{R}_n\}, \{\hat{q}_{\text{aff},n}\}, \{\hat{s}_{\text{aff},n}\}, \dots$
                                 $\dots \{A_n\}, \{B_n\}, \{b_n\}, \hat{P}, \hat{p}_{\text{aff}})$
        ▷ $N((4n_x^3 + 6n_x^2 n_u + 3n_x n_u^2 + \frac{1}{3}n_u^3) + (5n_x^2 + 7n_x n_u + \frac{5}{2}n_u^2))$ flops
    $\pi_{\text{aff},N} \leftarrow \hat{P}_N x_{\text{aff},N} + \hat{p}_{\text{aff},N}$                    ▷ $2n_x^2$ flops
    for $n = N-1 \rightarrow 0$ do
      $\pi_{\text{aff},n} \leftarrow \hat{q}_{\text{aff},n} + \hat{Q}_n x_{\text{aff},n} + \hat{S}_n' u_{\text{aff},n} + A_n' \pi_{\text{aff},n+1}$
                                                 ▷ $N(4n_x^2 + 2n_x n_u)$ flops
    end for

    $t_{\text{aff},0} \leftarrow C_0 u_{\text{aff},0} + D_0 x_0 - d_0$                       ▷ $2n_u n_l$ flops
    for $n = 1 \rightarrow N-1$ do
      $t_{\text{aff},n} \leftarrow C_n u_{\text{aff},n} + D_n x_{\text{aff},n} - d_n$       ▷ $(N-1)(2n_x n_l + 2n_u n_l)$ flops
    end for
    $t_{\text{aff},N} \leftarrow D_N x_{\text{aff},N} - d_N$                             ▷ $2n_x n_l$ flops

    $((\Delta x_{\text{aff}}, \Delta u_{\text{aff}}), \Delta\pi_{\text{aff}}, \Delta t_{\text{aff}}) = ((x_{\text{aff}}, u_{\text{aff}}), \pi_{\text{aff}}, t_{\text{aff}}) - ((x_k, u_k), \pi_k, t_k)$

    for $n = 0 \rightarrow N$ do
      $\Delta\lambda_{\text{aff},n} \leftarrow -(T_k^{-1}\Lambda_k)_n t_{\text{aff},n}$
    end for

    $\alpha_{\text{aff}} \leftarrow \max\{\alpha \in [0,1] | (\lambda, t) + \alpha(\Delta\lambda_{\text{aff}}, \Delta t_{\text{aff}}) \geq 0\}$
    $((x_{k+1}, u_{k+1}), \pi_{k+1}, \lambda_{k+1}, t_{k+1}) \leftarrow ((x_k, u_k), \pi_k, \lambda_k, t_k) +$
                    $+\gamma\alpha_{\text{aff}}((\Delta x_{\text{aff}}, \Delta u_{\text{aff}}), \Delta\pi_{\text{aff}}, \Delta\lambda_{\text{aff}}, \Delta t_{\text{aff}})$
    $\mu_{k+1} \leftarrow \frac{\lambda_{k+1}' t_{k+1}}{l}$

  end while
  return $(\{u_n\}_k)$
```

of the cost function is

$$\hat{g}_{\mathrm{cc}} = C'T^{-1}\big(\Delta\Lambda_{\mathrm{aff}}\Delta T_{\mathrm{aff}} - \sigma\mu_k e\big) = \begin{bmatrix} C_0'T_0^{-1}(\Delta\lambda_{\mathrm{aff},0}\Delta t_{\mathrm{aff},0} - \sigma\mu_k) \\ D_1'T_1^{-1}(\Delta\lambda_{\mathrm{aff},1}\Delta t_{\mathrm{aff},1} - \sigma\mu_k) \\ C_1'T_1^{-1}(\Delta\lambda_{\mathrm{aff},1}\Delta t_{\mathrm{aff},1} - \sigma\mu_k) \\ D_2'T_2^{-1}(\Delta\lambda_{\mathrm{aff},2}\Delta t_{\mathrm{aff},2} - \sigma\mu_k) \\ C_2'T_2^{-1}(\Delta\lambda_{\mathrm{aff},2}\Delta t_{\mathrm{aff},2} - \sigma\mu_k) \\ D_3'T_3^{-1}(\Delta\lambda_{\mathrm{aff},3}\Delta t_{\mathrm{aff},3} - \sigma\mu_k) \end{bmatrix}$$

and thus $\hat{p}_{\mathrm{cc}} = D_3'T_3^{-1}(\Delta\lambda_{\mathrm{aff},3}\Delta t_{\mathrm{aff},3} - \sigma\mu_k)$ and the matrices $\hat{s}_{\mathrm{cc}}$ and $\hat{q}_{\mathrm{cc}}$ are

$$\hat{s}_{\mathrm{cc}} = \begin{bmatrix} C_0'T_0^{-1}(\Delta\lambda_{\mathrm{aff},0}\Delta t_{\mathrm{aff},0} - \sigma\mu_k) \\ C_1'T_1^{-1}(\Delta\lambda_{\mathrm{aff},1}\Delta t_{\mathrm{aff},1} - \sigma\mu_k) \\ C_2'T_2^{-1}(\Delta\lambda_{\mathrm{aff},2}\Delta t_{\mathrm{aff},2} - \sigma\mu_k) \end{bmatrix}, \quad \hat{q}_{\mathrm{cc}} = \begin{bmatrix} D_0'T_0^{-1}(\Delta\lambda_{\mathrm{aff},0}\Delta t_{\mathrm{aff},0} - \sigma\mu_k) \\ D_1'T_1^{-1}(\Delta\lambda_{\mathrm{aff},1}\Delta t_{\mathrm{aff},1} - \sigma\mu_k) \\ D_2'T_2^{-1}(\Delta\lambda_{\mathrm{aff},2}\Delta t_{\mathrm{aff},2} - \sigma\mu_k) \end{bmatrix}$$

where again from the form of $\hat{g}_{\mathrm{cc}}$ we do not have any information about the actual value of $\hat{q}_{\mathrm{cc},0}$, but this is not affecting the solution. The initial state x_0 is this time set to the zero vector.

About the computation of the different components of the centering-corrector step, we use a modified version of the Riccati recursion algorithm for the computation of Δx_{cc}, exploiting the fact that in this case $b_n = 0$, and re-using the matrices L_n and Λ_n already computed (the notation is the same used in the chapter about the Riccati recursion, and thus in this context Λ_n is the lower factor in the Cholesky factorization of the matrix $R_{e,n}$).

The algorithm for the computation of the Δx_{cc} and Δu_{cc} is summarized in the following algorithm 9, with the quadratic terms of the computational complexity. The computational cost of the algorithm is quadratic, and counts for $N(4n_x^2 + 8n_xn_u + 2n_u^2)$ floating-point operations.

Algorithm 9 Modified Riccati recursion method for the computation of Δx_{cc} and Δu_{cc}

```
Require: $(\{q_n\}, \{s_n\}, \{A_n\}, \{B_n\}, p, \{\Lambda_n\}, \{L_n\})$
  $p_{n+1} \leftarrow p$
  for $n = N-1 \to 0$ do
    $l_n \leftarrow_{\mathrm{dtrsv}} -\Lambda_n^{-1}(s_n + B_n' \cdot_{\mathrm{dgemv}} p_{n+1})$          ▷ $2n_xn_u + n_u^2$ flops
    $p_n \leftarrow q_n + A_n' \cdot_{\mathrm{dgemv}} p_{n+1} - L_n' \cdot_{\mathrm{dgemv}} l_n$          ▷ $2n_x^2 + 2n_xn_u$ flops
  end for

  $x_0 \leftarrow 0$
  for $n = 0 \to N-1$ do
    $u_n \leftarrow_{\mathrm{dtrsv}} -(\Lambda_n')^{-1}(L_n \cdot_{\mathrm{dgemv}} x_n + l_n)$          ▷ $2n_xn_u + n_u^2$ flops
    $x_{n+1} \leftarrow A_n \cdot_{\mathrm{dgemv}} x_n + B_n \cdot_{\mathrm{dgemv}} n_u$          ▷ $2n_x^2 + 2n_xn_u$ flops
  end for
  return $(\{x_n\}, \{u_n\})$
```

The value of the increment in the Lagrange multiplier $\Delta\pi_{\mathrm{cc}}$ is obtained using the relation

$$\Delta\pi_{\mathrm{cc},n} = \hat{q}_{\mathrm{cc},n} + \hat{Q}_n\Delta x_{\mathrm{cc},n} + \hat{S}_n'\Delta u_{\mathrm{cc},n} + A_n'\Delta\pi_{\mathrm{cc},n+1}$$

with initial value

$$\Delta\pi_{\mathrm{cc},N} = \hat{P}\Delta x_{\mathrm{cc},N} + \hat{p}_{\mathrm{cc}}.$$

The value of Δt_{cc} is obtained from the equation

$$C\Delta x_{\text{cc}} - \Delta t_{\text{cc}} = 0$$

giving

$$\Delta t_{\text{cc}} = C\Delta x_{\text{cc}} = \begin{bmatrix} C_0\Delta u_{\text{cc},0} \\ C_1\Delta u_{\text{cc},1} + D_1\Delta x_{\text{cc},1} \\ C_2\Delta u_{\text{cc},2} + D_2\Delta x_{\text{cc},2} \\ D_3\Delta x_{\text{cc},3} \end{bmatrix}.$$

Finally, $\Delta\lambda_{\text{cc}}$ is obtained from the relation

$$T_k\Delta\lambda_{\text{cc}} + \Lambda_k\Delta t_{\text{cc}} = -\Delta\Lambda_{\text{aff}}\Delta T_{\text{aff}} + \sigma\mu_k e$$

giving

$$\Delta\lambda_{\text{cc}} = -T_k^{-1}\Lambda_k\Delta t_{\text{cc}} - T_k^{-1}(\Delta\Lambda_{\text{aff}}\Delta T_{\text{aff}} - \sigma\mu_k e).$$

In the following algorithm 10 we present the Mehrotra's predictor-corrector interior-point method for the solution of problem 8.1. We also compute the complexity per iteration up to quadratic terms, that is $N[(4n_x^3+6n_x^2n_u+3n_xn_u^2+\frac{1}{3}n_u^3) + (17n_x^2 + 19n_xn_u + \frac{9}{2}n_u^2) + n_l(2n_x^2 + 2n_xn_u + 2n_u^2 + 9n_x + 9n_u)] + 4n_x^2 + n_l(2n_x^2 + 5n_x)$. This means that each iteration of the Mehrotra's predictor-corrector interior-point method requires $N((8n_x^2 + 10n_xn_u + 2n_u^2) + n_l(4n_x + 4n_u)) + 2n_x^2 + 2n_xn_l$ floating-points operations more compared to the basic primal-dual interior-point method.

Anyway the asymptotic complexity per iteration is the same as the basic primal-dual interior-point method (algorithm 8), and is

$$N\left[\left(4n_x^3 + 6n_x^2n_u + 3n_xn_u^2 + \frac{1}{3}n_u^3\right) + n_l\left(2n_x^2 + 2n_xn_u + 2n_u^2\right)\right],$$

just using algorithm 3 for the computation of the affine step. This cost can be lowered up to

$$N\left[\left(\frac{7}{3}n_x^3 + 4n_x^2n_u + 2n_xn_u^2 + \frac{1}{3}n_u^3\right) + n_l\left(n_x^2 + 2n_xn_u + n_u^2\right)\right]$$

using algorithm 5.

Algorithm 10 Mehrotra's predictor-corrector interior-point method for the solution of problem 8.1

```
Require: $(\bar{x}_0, \{Q_n\}, \{S_n\}, \{R_n\}, \{q_n\}, \{s_n\}, \{A_n\}, \{B_n\}, \{b_n\}, \{C_n\}, \{D_n\}, \{d_n\}, P, p)$
  $((x_0, u_0), \pi_0, \lambda_0, t_0) \leftarrow ((0,0), 0, e, e)$
  $\mu_0 \leftarrow \frac{\lambda_0' t_0}{l}$

  while $\mu_k > \mu_{\min}$ and $k < k_{\max}$ do

    for $n = 0 \to N-1$ do
      $\hat{Q}_n \leftarrow Q_n + D_n'(T_k^{-1}\Lambda_k)_n D_n$                    $\triangleright$ $N(2n_x^2 n_l + n_x n_l)$ flops
      $\hat{S}_n \leftarrow S_n + C_n'(T_k^{-1}\Lambda_k)_n D_n$                    $\triangleright$ $N(2n_x n_u n_l)$ flops
      $\hat{R}_n \leftarrow R_n + C_n'(T_k^{-1}\Lambda_k)_n C_n$                    $\triangleright$ $N(2n_u^2 n_l + n_u n_l)$ flops
      $\hat{q}_{\mathrm{aff},n} \leftarrow q_n - D_n'(\lambda_n + (T_k^{-1}\Lambda_k)_n d_n)$      $\triangleright$ $N(2n_x n_l)$ flops
      $\hat{s}_{\mathrm{aff},n} \leftarrow s_n - C_n'(\lambda_n + (T_k^{-1}\Lambda_k)_n d_n)$      $\triangleright$ $N(2n_u n_l)$ flops
    end for
    $\hat{P} \leftarrow P + D_N'(T_k^{-1}\Lambda_k) D_N$                          $\triangleright$ $2n_x^2 n_l + n_x n_l$ flops
    $\hat{p}_{\mathrm{aff}} \leftarrow p - D_N'(\lambda_n + (T_k^{-1}\Lambda_k)_N d_N)$           $\triangleright$ $2n_x n_l$ flops

    $((x_{\mathrm{aff}}, u_{\mathrm{aff}}), \Lambda, L) \leftarrow \mathrm{RiccatiSolver}(\bar{x}_0, \{\hat{Q}_n\}, \{\hat{S}_n\}, \{\hat{R}_n\}, \{\hat{q}_{\mathrm{aff},n}\}, \{\hat{s}_{\mathrm{aff},n}\}, \dots$
                                        $\dots \{A_n\}, \{B_n\}, \{b_n\}, \hat{P}, \hat{p}_{\mathrm{aff}})$
          $\triangleright$ $N((4n_x^3 + 6n_x^2 n_u + 3n_x n_u^2 + \frac{1}{3}n_u^3) + (5n_x^2 + 7n_x n_u + \frac{5}{2}n_u^2))$ flops
    $\pi_{\mathrm{aff},N} \leftarrow \hat{P}_N x_{\mathrm{aff},N} + \hat{p}_{\mathrm{aff},N}$                         $\triangleright$ $2n_x^2$ flops
    for $n = N-1 \to 0$ do
      $\pi_{\mathrm{aff},n} \leftarrow \hat{q}_{\mathrm{aff},n} + \hat{Q}_n x_{\mathrm{aff},n} + \hat{S}_n' u_{\mathrm{aff},n} + A_n' \pi_{\mathrm{aff},n+1}$
                                                  $\triangleright$ $N(4n_x^2 + 2n_x n_u)$ flops
    end for

    $t_{\mathrm{aff},0} \leftarrow C_0 u_{\mathrm{aff},0} + D_0 x_0 - d_0$                          $\triangleright$ $2n_u n_l$ flops
    for $n = 1 \to N-1$ do
      $t_{\mathrm{aff},n} \leftarrow C_n u_{\mathrm{aff},n} + D_n x_{\mathrm{aff},n} - d_n$         $\triangleright$ $(N-1)(2n_x n_l + 2n_u n_l)$ flops
    end for
    $t_{\mathrm{aff},N} \leftarrow D_N x_{\mathrm{aff},N} - d_N$                                 $\triangleright$ $2n_x n_l$ flops

    $((\Delta x_{\mathrm{aff}}, \Delta u_{\mathrm{aff}}), \Delta\pi_{\mathrm{aff}}, \Delta t_{\mathrm{aff}}) = ((x_{\mathrm{aff}}, u_{\mathrm{aff}}), \pi_{\mathrm{aff}}, t_{\mathrm{aff}}) - ((x_k, u_k), \pi_k, t_k)$

    for $n = 0 \to N$ do
      $\Delta\lambda_{\mathrm{aff},n} \leftarrow -(T_k^{-1}\Lambda_k)_n t_{\mathrm{aff},n}$
    end for

    $\alpha_{\mathrm{aff}} \leftarrow \max\{\alpha \in [0,1] | (\lambda, t) + \alpha(\Delta\lambda_{\mathrm{aff}}, \Delta t_{\mathrm{aff}}) \geq 0\}$
    $\mu_{\mathrm{aff}} \leftarrow \frac{(\lambda_k + \alpha_{\mathrm{aff}}\Delta\lambda_{\mathrm{aff}})'(t_k + \alpha_{\mathrm{aff}}\Delta t_{\mathrm{aff}})}{l}$
    $\sigma \leftarrow (\frac{\mu_{\mathrm{aff}}}{\mu_k})^3$
    for $n = 0 \rightarrow N-1$ do
        $\hat{s}_{\mathrm{cc},n} \leftarrow C_n'(T_k^{-1}(\Delta\lambda_{\mathrm{aff}}\Delta t_{\mathrm{aff}} - \sigma\mu_k e))_n$        $\triangleright$ $N(2n_u n_l)$ flops
        $\hat{q}_{\mathrm{cc},n} \leftarrow D_n'(T_k^{-1}(\Delta\lambda_{\mathrm{aff}}\Delta t_{\mathrm{aff}} - \sigma\mu_k e))_n$        $\triangleright$ $N(2n_x n_l)$ flops
    end for
    $\hat{q}_{\mathrm{cc},N} \leftarrow D_N' T_N^{-1}(\Delta\lambda_{\mathrm{aff}}\Delta t_{\mathrm{aff}} - \sigma\mu_k e)_N$        $\triangleright$ $2n_x n_l$ flops

    $(x_{\mathrm{cc}}, u_{\mathrm{cc}}) \leftarrow \mathrm{ModifiedRiccatiSolver}(\{\hat{q}_{\mathrm{cc},n}\}, \{\hat{s}_{\mathrm{cc},n}\}, \{A_n\}, \{B_n\}, \hat{p}_{\mathrm{cc}}, \{\Lambda_n\}, \{L_n\})$
                                                        $\triangleright$ $N(4n_x^2 + 8n_x n_u + 2n_u^2)$ flops

    $\Delta\pi_{\mathrm{cc}} \leftarrow \hat{P}\Delta x_{\mathrm{cc},N} + \hat{p}_{\mathrm{cc}}$        $\triangleright$ $2n_x^2$ flops
    for $n = N-1 \rightarrow 0$ do
        $\Delta\pi_{\mathrm{cc},n} \leftarrow \hat{q}_{\mathrm{cc},n} + \hat{Q}_n\Delta x_{\mathrm{cc},n} + \hat{S}_n'\Delta u_{\mathrm{cc},n} + A_n'\Delta\pi_{\mathrm{cc},n+1}$
                                                        $\triangleright$ $N(4n_x^2 + 2n_x n_u)$ flops
    end for

    $\Delta t_{\mathrm{cc},0} \leftarrow C_0\Delta u_{\mathrm{cc},0}$        $\triangleright$ $2n_u n_l$ flops
    for $n = 1 \rightarrow N-1$ do
        $\Delta t_{\mathrm{cc},n} \leftarrow C_n\Delta u_{\mathrm{cc},n} + D_n\Delta x_{\mathrm{cc},n}$        $\triangleright$ $(N-1)(2n_x n_l + 2n_u n_l)$ flops
    end for
    $\Delta t_{\mathrm{cc},N} \leftarrow D_N\Delta x_{\mathrm{cc},N}$        $\triangleright$ $2n_x n_l$ flops

    for $n = 0 \rightarrow N$ do
        $\Delta\lambda_{\mathrm{cc},n} \leftarrow -(T_k^{-1}\Lambda_k)_n\Delta t_{\mathrm{cc},n} - (T_k^{-1}(\Delta\lambda_{\mathrm{aff}}\Delta t_{\mathrm{aff}} - \sigma\mu_k e))_n$
    end for
    $((\Delta x_k, \Delta u_k), \Delta\pi_k, \Delta\lambda_k, \Delta t_k) \leftarrow ((\Delta x_{\mathrm{aff}}, \Delta u_{\mathrm{aff}}), \Delta\pi_{\mathrm{aff}}, \Delta\lambda_{\mathrm{aff}}, \Delta t_{\mathrm{aff}}) +$
                                        $+((\Delta x_{\mathrm{cc}}, \Delta u_{\mathrm{cc}}), \Delta\pi_{\mathrm{cc}}, \Delta\lambda_{\mathrm{cc}}, \Delta t_{\mathrm{cc}})$
    $\alpha_{\max} \leftarrow \max\{\alpha \in [0,1] | (\lambda, t) + \alpha(\Delta\lambda_k, \Delta t_k) \geq 0\}$
    $((x_{k+1}, u_{k+1}), \pi_{k+1}, \lambda_{k+1}, t_{k+1}) \leftarrow ((x_k, u_k), \pi_k, \lambda_k, t_k) +$
                                        $+\gamma\alpha_{\max}((\Delta x_k, \Delta u_k), \Delta\pi_k, \Delta\lambda_k, \Delta t_k)$
    $\mu_{k+1} \leftarrow \frac{\lambda_{k+1}' t_{k+1}}{l}$

end while

return $(\{u_n\}_k)$
```

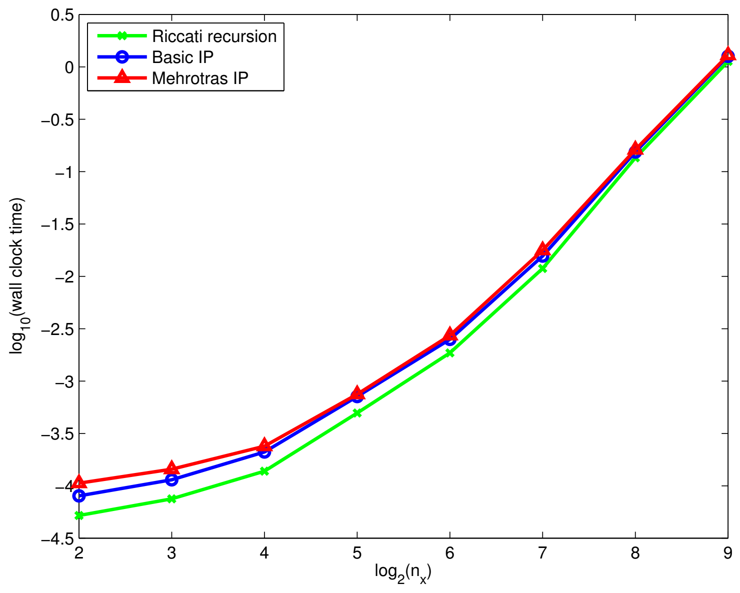


Figure 9.1: Comparison of the computation time per iteration for our implementations of interior-point methods 8 and 10, and the cost of the Riccati recursion method 3 for the solution of problem 2.1. The test problem is 10.1, with $n_x \in \{2, 4, 8, 16, 32, 64, 128, 256, 512\}$, $n_u = 1$ and $N = 10$. The time is wall clock time in seconds

9.3 Performance analysis

In this part we compare the performance of our implementations of the two interior-point methods 8 and 10. The test problem is 10.1.

In figure 9.1 there is a comparison of the wall clock time needed by the two implementations, together with the time needed by the Riccati recursion algorithm 3, used as sub-routine by both interior-point methods. The computation time is very similar for the two solvers, and the difference is relevant only in the case of very small systems. For medium and large values of n_x, it is almost impossible to distinguish the three curves.

In conclusion, the difference in complexity between the interior point methods 8 and 10 is quadratic in theory, and negligible in practice.

CHAPTER 10

Test Problems as Extended Linear Quadratic Control Problem with Inequality Constraints

In this chapter we define a test problem as an instance of problem 8.1, and we compare each other the performances of the two interior-point methods (algorithms 8 and 10) in the solution of this problem.

10.1 Problem definition

The test problem is the mass-spring problem with inequality constraints. The problem definition is the same that in the case of the mass-spring problem considered as an instance of problem 2.1, with the difference that there are also constraints on the maximum absolute value of the input action.

In fact, the input has to satisfy the constraints

$$-u_{\max} \leq u_n \leq u_{\max}$$

for $n \in \{0, 1, \ldots, N-1\}$, where $u_{\max}$ is the maximum absolute value of the input. The relation can be written as

$$\left\{ \begin{array}{r} u_n \geq -u_{\max} \\ -u_n \geq -u_{\max} \end{array} \right. \quad \Rightarrow \quad \left[\begin{array}{c} I \\ -I \end{array} \right] u_n \geq \left[\begin{array}{c} -I \\ -I \end{array} \right] u_{\max}.$$

The problem can be written in the form of problem 8.1 as:

$$\begin{aligned}\min_{\{u_n,x_{n+1}\}} \quad & \phi = \sum_{n=0}^{N-1} \left(\frac{1}{2} \begin{bmatrix} x_n' & u_n' \end{bmatrix} \begin{bmatrix} Q_n & S_n' \\ S_n & R_n \end{bmatrix} \begin{bmatrix} x_n \\ u_n \end{bmatrix} + \begin{bmatrix} q_n' & s_n' \end{bmatrix} \begin{bmatrix} cx_n \\ u_n \end{bmatrix} + \rho_n \right) + \\ & + \frac{1}{2} x_N' Q_N x_N + q_N' x_N + \rho_N \\ s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \\ & C_n u_n + D_n x_n \geq d_n \end{aligned} \tag{10.1}$$

where

$$\begin{gathered} Q_n = I_{2p\times 2p}, \quad S_n = 0_{m\times 2p}, \quad R_n = I_{m\times m}, \\ q_n = 0_{2p\times 1}, \quad s_n = 0_{m\times 1}, \quad \rho_n = 0, \\ A_n = A_d, \quad B_n = B_d, \quad b_n = 0_{2p\times 1}, \\ C_n = \begin{bmatrix} I_{m\times m} \\ -I_{m\times m} \end{bmatrix}, \quad D_n = 0_{2m\times 2p}, \quad d_n = -e_{2m\times 1} u_{\max} \end{gathered}$$

for each $n \in \mathcal{N} = \{0, 1, \ldots, N-1\}$ (and $n = N$ for Q_n, q_n, ρ_n), where $e_{2m\times 1}$ is a vector of $2m$ ones.

10.2 Small size example

We consider again the case of a system with $p = 2$ masses (and thus $n_x = 2p = 4$ states) and $m = 1$ forces (and thus $n_u = m = 1$ inputs), with horizon length $N = 20$ and initial state vector $x_0 = \begin{bmatrix} 5 & 10 & 15 & 20 \end{bmatrix}$; anyway this time the input is constrained between -5 and 5.

In figure 10.1 there is a comparison between the solution of test problem 10.1 (with inequality constraints) and the solution of test problem 7.2 (without inequality constraints). We notice that, when the input is constrained, it takes a longer time to control to zero the states, and that their oscillations are larger in absolute value. Anyway, in both cases the input is able to control to zero the states in the given control horizon of length 20.

10.3 Comparison between the two interior-point methods

In this section we compare the two interior-point methods in algorithms 8 and 10 for the solution of test problem 10.1.

We want to compare their performance in the solution of the small size problem presented in the previous section. We choose as stopping criteria a value of μ_k smaller than $\mu_{\min} = 10^{-5}$. The initial guess is a zero vector for x_0 and π_0, and a vector of just ones for λ_0 and t_0: it satisfies the constraint $(\lambda, t) \geq 0$ strictly, as requested by the interior-point methods. Anyway, this starting guess is infeasible with respect to the equality constraints and does not satisfies the other KKT equations.

In figure 10.2a there is a plot of the value of μ_k as function of the iteration number. It is interesting to notice that, even if the Mehrotra's method is converging in less iterations, in the first ones the value of μ_k is actually increasing

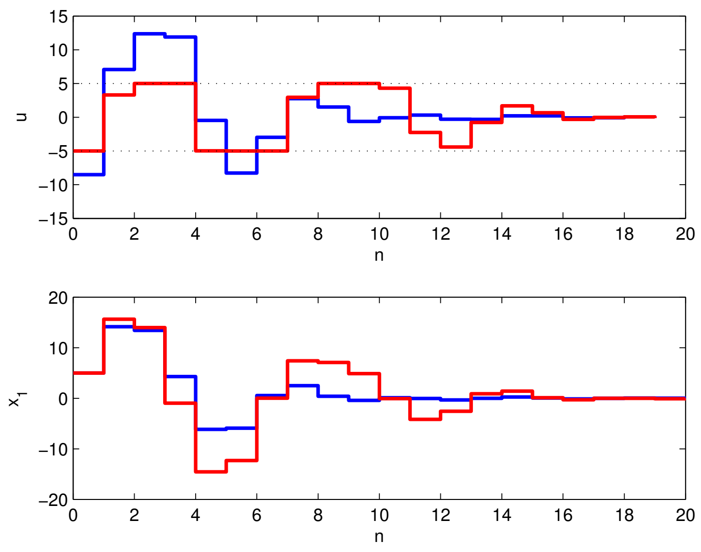
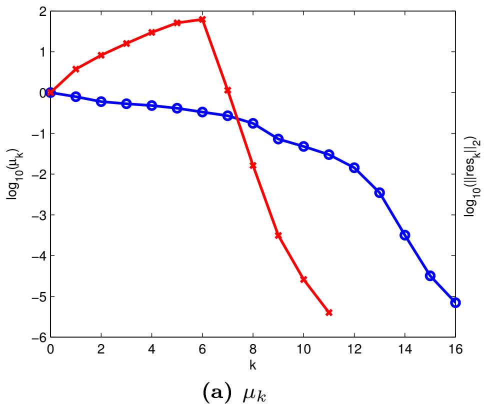
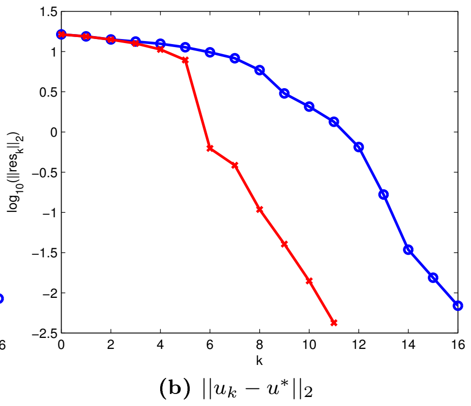


Figure 10.1: Comparison between the solution for the test problem with (red line) and without (blue line) the inequality constraints; in the top picture there is the input, in the bottom one there is the first component of the state vector.



(a) μ_k



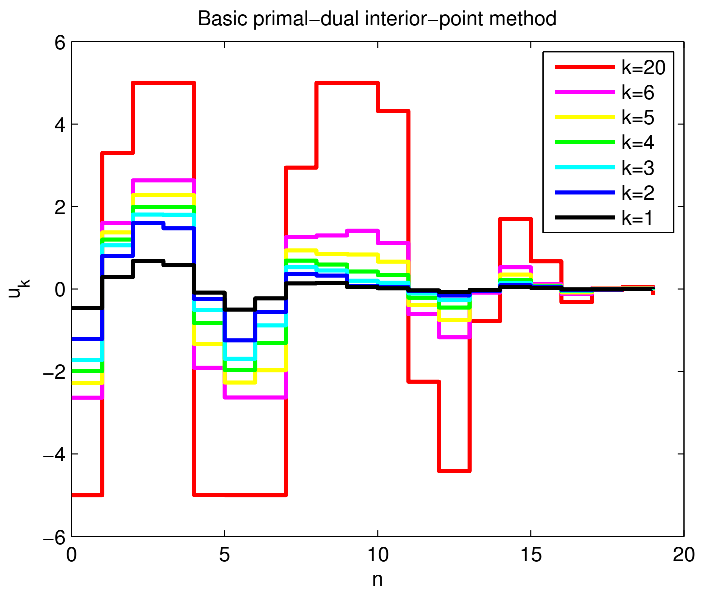
(b) $||u_k - u^*||_2$

Figure 10.2: Comparison between the value of μ_k (left) and the residuals of the input with respect to the optimal solution u^* (right) using the two interior-point methods, the basic primal-dual method (blue line) and the Mehrotra's predictor-corrector method (red line).

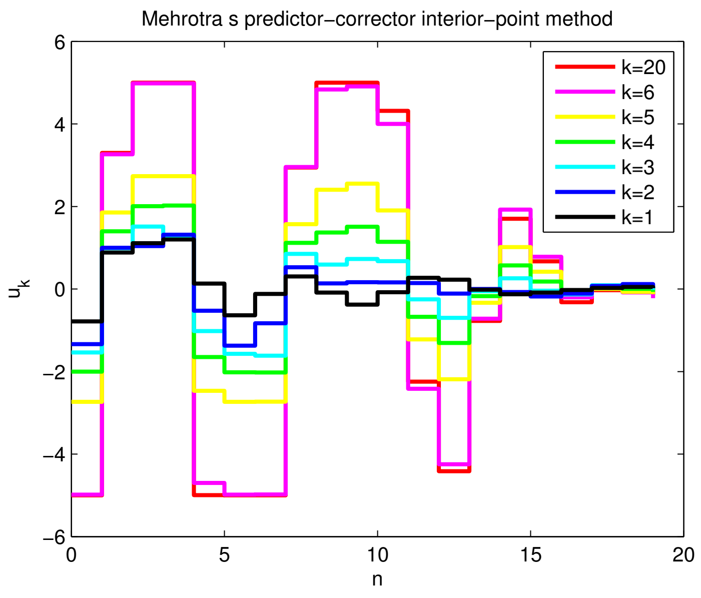
instead of decreasing. On the other hand, in the case of the basic primal-dual algorithm the value of μ_k decreases at each iteration. Both methods shows a faster convergence rate near the solution.

In figure 10.2b there is a plot of the 2-norm of the residuals of the input vector with respect to the 'optimal' solution (that is a solution obtained for a large number of iterations): it is interesting to notice that the norm of the residuals of the solution computed using the Mehtotra's method is always lower than the norm of the residuals computed using the basic primal-dual algorithm, even when the value of μ_k is larger. Furthermore, the value of the residuals is decreasing at each iteration for both methods, and in the case of the Mehrotra's method there is an important improvement for $k = 6$, when the iterate is almost the same as the optimal solution.

It is also interesting to study the shape of the control vector produced at different iterations of the two interior-point methods. Comparing the two methods in figure 10.3a and 10.3b, we notice that the basic interior-point method converges slower and the early iterations have the shape as the solution of problem 7.2 instead of the shape of the optimal solution. On the other hand, the shape of iterates produced the Mehrotra's predictor-corrector method is similar to the one of the optimal solution also in early iterations.



(a) Basic primal-dual interior-point method.



(b) Mehrotra's predictor-corrector interior-point method.

Figure 10.3: Comparison between the control laws produced by the interior-point methods after k iterations, $k \in \{1, 2, 3, 4, 5, 6, 20\}$.

CHAPTER 11

Problems variants

In this chapter we present a few variants of problems 2.2, 2.1 and 8.1, and we show how they can be rewritten in the same form as problems 2.2, 2.1 and 8.1: then they can be solved using the algorithms developed in previous chapters. Furthermore, we suggest how to tailor these algorithms to exploit the special structure of the problems.

11.1 Variants of problem 2.2

In this section we present a few problems that are variations of problem 2.2. Since problem 2.1 is an extension of problem 2.2, they can be seen as variants of problem 2.1 as well.

11.1.1 Output in the cost function

In this case, in the formulation of the cost function there is the output vector instead of the state vector. The output vector is defined as a linear function of the state vector, $y_n = Cx_n$. Then we consider the problem

$$\begin{aligned} \min_{\{y_{n+1},u_n\}} \quad & \phi = \frac{1}{2}\sum_{n=0}^{N-1}(y_n'Qy_n + u_n'Ru_n) + \frac{1}{2}y_N'Q_Ny_N \\ s.t. \quad & x_{n+1} = Ax_n + Bu_n \\ & y_n = Cx_n \end{aligned}$$

We can use the definition of the output vector to rewrite the cost function as

$$\begin{aligned}\phi &= \frac{1}{2}\sum_{n=0}^{N-1}(y_n' Q y_n + u_n' R u_n) + \frac{1}{2} y_N' Q_N y_N = \\ &= \frac{1}{2}\sum_{n=0}^{N-1}(x_n' C' Q C x_n + u_n' R u_n) + \frac{1}{2} x_N' C' Q_N C x_N = \\ &= \frac{1}{2}\sum_{n=0}^{N-1}\left(x_n' \bar{Q} x_n + u_n' R u_n\right) + \frac{1}{2} x_N' \bar{Q}_N x_N\end{aligned}$$

and the problem formulation as

$$\begin{aligned}\min_{\{y_{n+1},u_n\}} \quad & \phi = \frac{1}{2}\sum_{n=0}^{N-1}\left(x_n' \bar{Q} x_n + u_n' R u_n\right) + \frac{1}{2} x_N' \bar{Q}_N x_N \\ s.t. \quad & x_{n+1} = A x_n + B u_n\end{aligned}$$

that is an instance of problem 2.2. In general the matrix $\bar{Q} = C'QC$ is symmetric positive semi-definite, even if the matrix Q is positive definite: for example, in case of a smaller number of outputs than states, $\bar{Q}$ is rank deficient.

In general it is not possible to exploit the specific form of this problem to obtain better algorithms.

11.1.2 Input variation in the cost function

In this case, in the formulation of the cost function there is also a penalty term for the variation of the input vector $\Delta u_n = u_n - u_{n-1}$: then we consider the problem

$$\begin{aligned}\min_{\{x_{n+1},u_n\}} \quad & \phi = \frac{1}{2}\sum_{n=0}^{N-1}(x_n' Q x_n + u_n' R u_n + \Delta u_n' S \Delta u_n) + \frac{1}{2} x_N' Q_N x_N \\ s.t. \quad & x_{n+1} = A x_n + B u_n\end{aligned}$$

where S is a symmetric positive semi-definite matrix of size $n_u \times n_u$, and the other matrices are defined as usual.

Using the definition of the input variation, the cost function can be rewritten as

$$\begin{aligned}
\phi =& \frac{1}{2}\sum_{n=0}^{N-1}(x_n'Qx_n + u_n'Ru_n + \Delta u_n'S\Delta u_n) + \frac{1}{2}x_N'Q_Nx_N = \\
=& \frac{1}{2}\sum_{n=0}^{N-1}(x_n'Qx_n + u_n'Ru_n + (u_n-u_{n-1})'S(u_n-u_{n-1})) + \frac{1}{2}x_N'Q_Nx_N = \\
=& \frac{1}{2}\sum_{n=0}^{N-1}\Bigg(\begin{bmatrix} x_n' & u_{n-1}' \end{bmatrix}\begin{bmatrix} Q & 0 \\ 0 & S \end{bmatrix}\begin{bmatrix} x_n \\ u_{n-1} \end{bmatrix} + u_n'(R+S)u_n + \\
&+ u_n'\begin{bmatrix} 0 & -S \end{bmatrix}\begin{bmatrix} x_n \\ u_{n-1} \end{bmatrix} + \begin{bmatrix} x_n' & u_{n-1}' \end{bmatrix}\begin{bmatrix} 0 \\ -S \end{bmatrix}\Bigg) + \\
&+ \frac{1}{2}\begin{bmatrix} x_N & u_{N-1} \end{bmatrix}\begin{bmatrix} Q_N & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x_N \\ u_{N-1} \end{bmatrix} = \\
=& \frac{1}{2}\sum_{n=0}^{N-1}\left(\bar{x}_n'\bar{Q}\bar{x}_n + u_n'\bar{R}u_n + u_n'\bar{S}\bar{x}_n + \bar{x}_n\bar{S}'u_n\right) + \frac{1}{2}\bar{x}_N'\bar{Q}_N\bar{x}_N
\end{aligned}$$

where we define the augmented state vector as

$$\bar{x}_n = \begin{bmatrix} x_n \\ u_{n-1} \end{bmatrix}$$

and the matrices of the cost function as

$$\bar{Q} = \begin{bmatrix} Q & 0 \\ 0 & S \end{bmatrix}, \quad \bar{R} = R + S, \quad \bar{S} = \begin{bmatrix} 0 & -S \end{bmatrix}.$$

The constraints take the form

$$\bar{x}_{n+1} = \begin{bmatrix} x_{n+1} \\ u_n \end{bmatrix} = \begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x_n \\ u_{n-1} \end{bmatrix} + \begin{bmatrix} B \\ I \end{bmatrix}u_n = \bar{A}\bar{x}_n + \bar{B}u_n$$

and the problem is thus an instance of problem 2.2, with a state vector of size $n_{\bar{x}} = n_x + n_u$. This means that, for example, the cubic terms in the complexity of an iteration of algorithm 3 are

$$\begin{aligned}
4n_{\bar{x}}^3 + 6n_{\bar{x}}^2n_u + 3n_{\bar{x}}n_u^2 + \frac{1}{3}n_u^3 = \\
= 4(n_x+n_u)^3 + 6(n_x+n_u)^2n_u + 3(n_x+n_u)n_u^2 + \frac{1}{3}n_u^3 = \\
= 4n_x^3 + 18n_x^2n_u + 27n_xn_u^2 + \frac{40}{3}n_u^3.
\end{aligned}$$

Anyway, it is possible to write tailored algorithms to exploit the special structure of this problem, and obtain better performances. For example, the

Riccati recursion expression is

$$
\begin{aligned}
P_{n-1} &= \bar{Q} + \bar{A}'P_n\bar{A} - (\bar{S} + \bar{B}'P_n\bar{A})'(\bar{R} + \bar{B}'P_n\bar{B})^{-1}(\bar{S} + \bar{B}'P_n\bar{A}) = \\
&= \begin{bmatrix} Q & 0 \\ 0 & S \end{bmatrix} + \begin{bmatrix} A' & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} P_{n,11} & P_{n,12} \\ P_{n,21} & P_{n.22} \end{bmatrix} \begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix} - \\
&\quad - \left(\begin{bmatrix} 0 & -S \end{bmatrix} + \begin{bmatrix} B' & I \end{bmatrix} \begin{bmatrix} P_{n,11} & P_{n,12} \\ P_{n,21} & P_{n,22} \end{bmatrix} \begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix} \right)' \cdot \\
&\quad \cdot \left((R+S) + \begin{bmatrix} B' & I \end{bmatrix} \begin{bmatrix} P_{n,11} & P_{n,12} \\ P_{n,21} & P_{n,22} \end{bmatrix} \begin{bmatrix} B \\ I \end{bmatrix} \right) \cdot \\
&\quad \cdot \left(\begin{bmatrix} 0 & -S \end{bmatrix} + \begin{bmatrix} B' & I \end{bmatrix} \begin{bmatrix} P_{n,11} & P_{n,12} \\ P_{n,21} & P_{n,22} \end{bmatrix} \begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix} \right) = \\
&= \begin{bmatrix} Q & 0 \\ 0 & S \end{bmatrix} + \begin{bmatrix} A'P_{n,11}A & 0 \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} A'(P_{n,11}B + P'_{n,21}) \\ -S \end{bmatrix} \cdot \\
&\quad \cdot (R + S + B'P_{n,11}B + B'P_{n,12} + P_{n,21}B + P_{n,22})^{-1} \begin{bmatrix} (B'P_{n,11} + P_{n,21})A & -S \end{bmatrix}
\end{aligned}
$$

and then algorithm 3 can be modified to implement the above recursion, obtaining a complexity per iteration with cubic terms

$$
4n_x^3 + 6n_x^2 n_u + 9n_x n_u^2 + \frac{10}{3} n_u^3,
$$

that is just $6n_x n_u^2 + 3n_u^3$ larger than problem 2.2.

11.1.3 Diagonal Hessian in the cost function

This is a special case of problem 2.2, and some of the algorithms presented in the previous chapeters can take advantage of this.

Riccati recursion method In the case of this method, a diagonal Hessian in not influencing the cubic terms in the complexity expression.

Schur complement method This method can take advantage of a diagonal Hessian (that has to be positive definite). In fact the Cholesky factorization, the inversion of the factor U and the computation of $\Phi = AU^{-1}$ are trivial, and require at most a quadratic number of floating-point operations.

The matrix ϕ is in the form

$$
\Phi = \begin{bmatrix} \Phi_{0,22} & \Phi_{1,11} & & & & \\ & \Phi_{1,21} & \Phi_{1,22} & \Phi_{2,11} & & \\ & & & \Phi_{2,21} & \Phi_{2,22} & \Phi_{3,11} \end{bmatrix}
$$

where the $\Phi_{n,11}$ are diagonal $n_x \times n_x$ matrices, $\Phi_{n,21}$ are full $n_x \times n_x$ matrices and $\Phi_{n,22}$ are full $n_x \times n_u$ matrices.

Then the computation of the product $\Psi = \Phi\Phi'$ has an asymptotic cost of $N(n_x^3 + n_x^2 n_u)$ floating-point operations. The blocks $\Psi_{i,j}$ are all dense, and thus the Cholesky factorization of the matrix Ψ has an asymptotic cost of $N(\frac{7}{3}n_x^3)$ floating-point operations, as usual. The following operations (systems solution) have only a quadratic cost.

The asymptotic cost of the algorithm is then

$$N\left(\frac{10}{3}n_x^3+n_x^2n_u\right).$$

It is pretty smaller than the cost of the general Schur complement algorithm but still larger that the most efficient version of the Riccati recursion method in the case of $n_x \gg n_u$. Anyway, in case of large value of n_u, the algorithm is extremely convenient, being linear in the number of inputs.

Condensing methods Condensing methods can take advantage of a diagonal Hessian matrix: in this case the computation of the product $\bar{Q}\Gamma_u$ has an asymptotic cost of $\frac{1}{2}N^2n_un_x$ floating-point operations instead of $N^2n_x^2n_u$. The asymptotic cost of the algorithm becomes

$$2N^2n_xn_u(n_x+n_u)+\frac{1}{3}(Nn_u)^3$$

floating-point operations.

11.2 Variants of problem 2.1

In this section we present a problem that can be rewritten in the form of problem 2.1.

11.2.1 Control to a non-zero reference

In this part we consider the problem of control the states vector to a non-zero reference,

$$\begin{aligned}\min_{x_{n+1},u_n}\quad & \phi=\frac{1}{2}\sum_{n=0}^{N-1}(x_n-z_n)'Q(x_n-z_n)+u_n'Ru_n+\frac{1}{2}(x_N-z_N)'Q_N(x_N-z_N)\\ s.t.\quad & x_{n+1}=Ax_n+Bu_n\end{aligned}$$

The cost function can be rewritten as

$$\phi=\sum_{n=0}^{N-1}\left(\frac{1}{2}x_n'Qx_n+\frac{1}{2}u_n'Ru_n+(-z_n'Q)x_n+\frac{1}{2}z_n'Qz_n\right)+ \\ +\frac{1}{2}x_N'Q_Nx_N+(-z_N'Q_N)x_N+\frac{1}{2}z_N'Q_Nz_N$$

that can be seen as the cost function of problem 2.1 with $q_n=-Qz_n$ and $q_N=-Q_Nz_N$.

No special algorithm can be developed.

11.3 Variants of problem 8.1

11.3.1 Diagonal Hessian matrix in the cost function and box constraints

In this part we study a special case of problem 8.1, namely the case with diagonal Hessian in the cost function and box constraints.

Box constraints are in the form

$$\begin{aligned} u_{\min} &\leq u_n \leq u_{\max} \\ x_{\min} &\leq x_n \leq x_{\max} \end{aligned}$$

and can be rewritten as

$$C_n u_n + D_n x_n = \begin{bmatrix} I \\ -I \\ 0 \\ 0 \end{bmatrix} u_n + \begin{bmatrix} 0 \\ 0 \\ I \\ -I \end{bmatrix} x_n \geq \begin{bmatrix} u_{\min} \\ -u_{\max} \\ x_{\min} \\ -x_{\max} \end{bmatrix} = d_n.$$

The matrix $C'(T^{-1}\Lambda)C$, that is the update term of the Hessian matrix at each iteration of the interior-pint method, is diagonal. In fact, for each block i, we have

$$\begin{aligned} &D_i'(T^{-1}\Lambda)_i D_i = \\ &= \begin{bmatrix} 0 & 0 & I & -I \end{bmatrix} \begin{bmatrix} (T^{-1}\Lambda)_{i,1} & & & \\ & (T^{-1}\Lambda)_{i,2} & & \\ & & (T^{-1}\Lambda)_{i,3} & \\ & & & (T^{-1}\Lambda)_{i,4} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ I \\ -I \end{bmatrix} = \\ &= (T^{-1}\Lambda)_{i,3} + (T^{-1}\Lambda)_{i,4} \end{aligned}$$

$$\begin{aligned} &C_i'(T^{-1}\Lambda)_i C_i = \\ &= \begin{bmatrix} I & -I & 0 & 0 \end{bmatrix} \begin{bmatrix} (T^{-1}\Lambda)_{i,1} & & & \\ & (T^{-1}\Lambda)_{i,2} & & \\ & & (T^{-1}\Lambda)_{i,3} & \\ & & & (T^{-1}\Lambda)_{i,4} \end{bmatrix} \begin{bmatrix} I \\ -I \\ 0 \\ 0 \end{bmatrix} = \\ &= (T^{-1}\Lambda)_{i,1} + (T^{-1}\Lambda)_{i,2} \end{aligned}$$

$$\begin{aligned} &D_i'(T^{-1}\Lambda)_i C_i = \\ &= \begin{bmatrix} 0 & 0 & I & -I \end{bmatrix} \begin{bmatrix} (T^{-1}\Lambda)_{i,1} & & & \\ & (T^{-1}\Lambda)_{i,2} & & \\ & & (T^{-1}\Lambda)_{i,3} & \\ & & & (T^{-1}\Lambda)_{i,4} \end{bmatrix} \begin{bmatrix} I \\ -I \\ 0 \\ 0 \end{bmatrix} = 0 \end{aligned}$$

then the matrix $C'(T^{-1}\Lambda)C$ is diagonal, since the matrix $(T^{-1}\Lambda)$ is diagonal.

This means that, at each iteration of the interior-point method, the updated Hessian matrix $H+C'(T^{-1}\Lambda)C$ is diagonal, and then the asymptotic cost of one iteration of the interior-point method is the cost of the solution of a problem in the form 2.1 with diagonal Hessian, and has been computed in section 11.1.3.

Chapter 12

Conclusions

In this final chapter we want to briefly report some of the result obtained in this thesis, and that can be useful to others.

The main goal of this thesis was the comparison of some methods for the solution of problem 2.1: this have been made both in theory (as number of floating-point operations) and in practice (comparing the computation time of our C implementations).

Some of the results are that:

- In general, direct sparse solvers (chapter 3) are slow. They may be more attractive if the matrices of problem 2.1 are sparse (even if this has not been tested in practice).
- The method with the widest field of application is Riccati recursion method (chapter 5): then in general we suggest this method.
- In case of small values of the number of inputs and horizon length, and large values of the number of states, the condensing method (chapter 6) can be the best choice. Furthermore, it can take advantage of a diagonal Hessian in the cost function, and allows some savings if used to solve a set of problems with the same coefficient matrices in the system dynamic equation (for example if it is used as routine in an interior-point method).
- In the special case of large value of the number of inputs and diagonal Hessian in the cost function, Schur complement method (chapter 4) can be the best choice (although we had not tested this special case in practice).
- In some special problem formulation (chapter 11), it may be advantageous to specialize the algorithms presented in this thesis, and take advantage of the shape of the coefficient matrices (even if this has not been tested in practice).

Appendix A

Machines and programming languages

In this chapter we briefly describe the machine used to perform the tests, and briefly introduce the languages used to write the code.

A.1 Hardware and software

The software has been written and tested on a entry-level laptop, equipped with a rather slow processor and a small cache compared to other machines. Here can be found more details:

- processor: Intel Pentinum Dual-Core T2390 @ 1.86 GHz
- L1 cache: 2x32 KB data + 2x32 KB instructions
- L2 cache: 1 MB
- FSB speed: 533 MHz

The processor is based on the Intel IA32-64 architecture, and support the instruction sets x87 FPU, MMX, SSE, SSE2, SSE3, SSSE3. Since the algorithms in this thesis are based on floating-points operations, we are interested only on x87 FPU, SSE2 and SSE2 instructions sets; in particular there are 8 64-bit MMX registers (where x87 FPU and MMX operations are performed) and 16 128-bit XMM registers (where SSE, SSE2, SSE3, SSSE3 operations are performed).

Non-optimized code makes use only of general purpose and MMX registers, while highly optimized code uses in a smart way also the extended registers XMM.

The operative system is the Linux distribution Ubuntu, versions 12.04. The distribution provides, already installed, the compiler `gcc` (GNU Compiler Collection), that can be used to compile, among the others, C, C++, FORTRAN and Java code. Two `gcc` manuals are [Gou] and [Sta].

In the case of the program `myprog.c` written in C, the command

```
$ gcc myprog.c -o myprog.out
```

compiles and links the code, creating the executable `myprog.out`. The program can be executed with the command

```
$ ./myprog.out
```

The command

```
$ gcc -c myprog.c
```

compiles the code and creates the file in machine code `myprog.o`.

Multiple files .o can be linked together in the same executable with the command

```
$ gcc myprog1.o myprog2.o -o myprog.out
```

The program can be linked against libraries using the options `-L` and `-l`: for example the command

```
$ gcc myprog.c -o myprog.out -lblas -L. lapack.a
```

compiles the source code `myprog.c` in the machine file `myprog.o`, and links the latter with the libraries `libblas.a` (in the default folder) and `lapack.a` (in the current folder), creating the executable `myprog.out`.

The libraries `libblas.a` and `lapack.a` are called static libraries, and are a collection of files .o gathered together; their code is explicitly added to the executable during the linking phase. There exist also dynamic libraries, for example `blas.so`: their code is not included in the executable, but the linking is done at execution time. The environment variable `LD_LIBRARY_PATH` has to be set to the address of the folder containing the dynamic library, in case it is not in the standard folders.

The compiler `gcc` allows a number of compiling preferences; in particular, the optimization flags `-O1, -O2, -O3` and `-Ofast` can be used to set the level of optimization of the code; the option `-Ofast` (activating the flag `-ffast-math`) may produce code performing floating-point operations violating IEEE or ANSI standards; its use is then not recommended.

An useful tool in the `gcc` tools collection is the graph profiler `gprof`, used to determine how much time is spent in which sub-routine. The code has to be compiled with the flag `-pg`. The following execution of the program (in this case `myprog.out`) produces the file `gmon.out`, that can be analyzed with the command

```
$ gprof myprog.out gmon.out
```

producing both a flat profile and a call graph.

A.2 Programming languages

A.2.1 MATLAB

MATLAB (MATrix LABoratory) is a numeric computing environment and a fourth-generation programming language. It is mainly written in C; for the most intensive routines it uses optimized BLAS and LAPACK implementations.

The MATLAB version we used is MATLAB 7.10.0 (R2010a) for Linux, using the BLAS implementation MKL 10.0.3, by Intel.

In this work, we used MATLAB only on an early prototyping phase, since the code written in MATLAB language is rather slower than the code written in C. In section 7.4 there is a comparison of the same algorithm written in C and MATLAB.

A.2.2 C

C is an imperative, general-purpose programming language, appeared in 1972. It is one of the most used programming languages of all times. A C manual is [KR88].

The default matrix representation in C is row-major order (i.e. in memory a matrix is saved as a vector, storing the elements a row after the other; this means that elements on the same row are stored contiguously, while elements on the same column can be stored far away). Anyway, in our code we use column-major order (i.e. in memory a matrix is saved as a vector, storing the elements a column after the other; this means that elements on the same column are stored contiguously, while elements on the same row can be stored far away), since it is the default order required the BLAS and LAPACK. The storing order of data in memory is a factor affecting the performance of algorithms, since the access to data stored not contiguously may result in a cache miss, slowing down the execution.

For the most expensive parts, our C code calls algebra routine from optimized BLAS version, since hand written C (and FORTRAN) code is not competitive, even if optimized using the compilers flags. In particular, we use the Intel implementation MKL 10.0.3, the same as MATLAB, to have a direct comparison of the performance of our implementation of the algorithms in C and MATLAB. Other BLAS versions have been tested, see appendix C for details.

A.2.3 FORTRAN

FORTRAN is an imperative, general-purpose programming language, appeared in 1957. It is the most used language in scientific computing. A FORTRAN 77 manual is [Pag07].

The BLAS and LAPACK implementations are often written in this language, and thus their default data storing order is column-major, the FORTRAN order.

Appendix B

Basic Algorithms

In this section we present a number of algorithms performing basic linear algebra operations, used as building blocks in our code. Efficient implementation of these algorithms can be found in the BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage) libraries. We shortly describe the algorithms and compute their complexity as number of floating-point operations. Anyway, the exact number of floating-points operations depends on the specific implementation.

This section does not cover all the BLAS and LAPACK routines, but only the subset used in our implementations of the algorithms.

B.1 Matrix-matrix multiplication

The matrix-matrix multiplication routines are part of the level-3 BLAS (or LAPACK in some case), and they have a cubic complexity in the matrices size.

B.1.1 General matrices

The general matrix-matrix multiplication is implemented by the BLAS routine `dgemm`, performing $C \leftarrow \alpha \cdot op(A) \cdot op(B) + \beta C$, where $op(X) = X$ or $op(X) = X'$, C is a $m \times n$ matrix, and $op(A)$ and $op(B)$ are respectively $m \times k$ and $k \times n$ matrices. It is often used as benchmark to compare different BLAS implementations.

The algorithm is based on the definition of matrix product,

$$
\begin{aligned}
A \cdot B = \left[\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array}\right] \left[\begin{array}{cc} b_{11} & b_{12} \\ b_{21} & b_{22} \end{array}\right] = \left[\begin{array}{cc} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{array}\right] = \\
= \left[\begin{array}{cc} c_{11} & c_{12} \\ c_{21} & c_{22} \end{array}\right] = C,
\end{aligned}
$$

in case of a matrix 2×2. If a_{ij} and b_{ij} are scalars, we have an unblocked algorithm; if they are sub-matrices, we have a blocked algorithm; a blocked algorithm with a proper block size can make a better use of cache and avoid useless data movement and cache miss.

Complexity: $2mnk + 2mn$ (rectangular matrices); $2n^3 + 2n^2$ (squared matrices, $m = n = k$); it might be lowered up to $2mnk - mn$ and $2n^3 - n^2$ in the case of $|\alpha| = 1$ and $\beta = 0$.

B.1.2 Triangular matrix

If one of the factor matrices A is triangular, it is possible to save roughly half of the floating-points operations, since we can avoid to perform the multiplication if one of the factors is a zero element in A. The algorithm is implemented by the BLAS routine `dtrmm`, performing $B \leftarrow \alpha \cdot op(A) \cdot B$ or $B \leftarrow \alpha \cdot B \cdot op(A)$, where $op(A) = A$ or $op(A) = A'$, B is a $m \times n$ matrix and A is a $m \times m$ or $n \times n$ matrix.

Complexity: $2m\frac{n(n+1)}{2} - mn + mn = mn^2 + mn$ (A triangular matrix of size $n \times n$, B rectangular matrix of size $m \times n$); $2n\frac{m(m+1)}{2} - mn + mn = m^2n + mn$ (A triangular matrix of size $m \times m$, B rectangular matrix of size $m \times n$); $2n\frac{n(n+1)}{2} - n^2 + n^2 = n^3 + n^2$ (B squared matrix, $m = n$); it might be lowered to mn^2, m^2n and n^3 in the case $|\alpha = 1|$.

B.1.3 Rank-k update

If the two factor matrices are one the transposed of the other, the product matrix is symmetric (and positive semi-definite). Then it is possible to save roughly half of the floating-points operations, computing just the upper or the lower triangular part of the product.

The algorithm is implemented by the BLAS routine `dsyrk`, performing $C \leftarrow \alpha \cdot A \cdot A' + \beta C$ or $C \leftarrow \alpha \cdot A' \cdot A + \beta C$, where C is a $n \times n$ matrix and A is respectively a $n \times k$ and $k \times n$ matrix.

Complexity: $2k\frac{n(n+1)}{2} + 2\frac{n(n+1)}{2} = n^2k + nk + n^2 + n$ (A rectangular matrix); $n^3 + 2n^2 + n$ (A squared matrix, $k = n$); it might be lowered up to $n^2k + nk - n^2$ and n^3 in the case $|\alpha = 1|$ and $\beta = 0$.

B.1.4 Rank-n update with triangular matrices

If in the rank-k update the factor matrix is triangular, it is possible to save even more computations. In particular, the LAPACK routines `dlauum` (blocked version) and `dlauu2` (unblocked version) implement the algorithm in the case that the first factor is upper and the second lower triangular; the routines are computing $U \leftarrow U \cdot U'$ or $L \leftarrow L' \cdot L$, where U and L are respectively an upper and lower triangular matrix.

The routine may be used to efficiently invert a positive definite matrix, together with the Cholesky factorization and the triangular matrix inversion routines, all part of LAPACK.

The algorithms used to solve this problem are analogous to the ones for the Cholesky factorization, and in particular there are algorithms based on the representation of a matrix in 2×2 and 3×3 blocks form.

2×2 **algorithm** Given the general upper matrix in 2×2 block form

$$U = \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

the algorithm is based on the fact that

$$\begin{aligned} UU' &= \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} U'_{11} & 0 \\ U'_{12} & U'_{22} \end{bmatrix} = \begin{bmatrix} U_{11}U'_{11} + U_{12}U'_{12} & U_{12}U'_{22} \\ * & U_{22}U'_{22} \end{bmatrix} = \\ &= \begin{bmatrix} H_{11} & H_{12} \\ * & H_{22} \end{bmatrix} = H. \end{aligned}$$

Both iterative and recursive algorithms are possible. Regarding iterative algorithms, at the beginning of the general iteration $H_{22} = U_{22}U'_{22}$ contains the already computed part of the product matrix. In case of unblocked algorithm, U_{11} is a scalar and U_{12} a row vector, and the computation of $U_{11}U'_{11}$ is a trivial scalar multiplication, $U_{12}U'_{12}$ is the scalar product and $U_{12}U'_{22}$ is a triangular matrix-vector product. In case of a blocked algorithm, the computation of $U_{11}U'_{11}$ is made using the unblocked algorithm, $U_{12}U'_{12}$ is a rank-k update and $U_{12}U'_{22}$ is a triangular matrix-matrix product.

Regarding recursive algorithms, the matrix U is split such that the matrix U_{12} is roughly squared, and the algotithm is called on the submatrices U_{11} and U_{22}; as base case can be used an algorithm for matrices of small size, for example 1×1 and 2×2.

3×3 **algorithm** Given the general upper matrix in 3×3 block form

$$U = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

the algorithm is based on the fact that

$$\begin{aligned} UU' &= \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} \begin{bmatrix} U'_{11} & 0 & 0 \\ U'_{12} & U'_{22} & 0 \\ U'_{13} & U'_{23} & U'_{33} \end{bmatrix} = \\ &= \begin{bmatrix} U_{11}U'_{11} + U_{12}U'_{12} + U_{13}U'_{13} & U_{12}U'_{22} + U_{13}U'_{23} & U_{13}U'_{33} \\ * & U_{22}U'_{22} + U_{23}U'_{23} & U_{23}U'_{33} \\ * & * & U_{33}U'_{33} \end{bmatrix} \end{aligned}$$

Again both iterative (blocked and unblocked) and recursive algorithms are possible, in analogy with the Cholesky factorization algorithm (see B.4 for details).

The algorithms implemented by the LAPACK routines `dlauum` and `dlauu2` are respectively the blocked and unblocked iterative version of the 3×3 algorithm.

Complexity $\sum_{i=1}^{n} i(2(n+1-i)-1) = \frac{n(n+1)(2n+1)}{6} = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$ (exactly the same as the Cholesky factorization).

B.2 Matrix-vector multiplication

The matrix-vector multiplication routines are part of the level-2 BLAS, and have a quadratic complexity on the matrix size.

B.2.1 General matrix

The general case is implemented by the BLAS routine `dgemv`, performing $y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$ or $y \leftarrow \alpha \cdot A' \cdot x + \beta \cdot y$, where A is a $m \times n$ matrix.

Complexity: $2mn + 2sizeof(y)$ (rectangular matrix); $2n^2 + 2n$ (squared matrix), that can be reduced up to $2mn - sizeof(y)$ and $2n^2 - n$ if $|\alpha = 1|$ and $\beta = 0$.

B.2.2 Triangular matrix

In case of triangular matrix, we can save roughly half of the computations, and the algorithm can be carried out in place, overwriting the x vector; it is implemented by the BLAS routine `dtrmv`, performing $x \leftarrow Ax$ or $x \leftarrow A'x$.

For a generic lower triangular matrix of size 3×3 (the case of the upper triangular matrix being analogous),

$$Ax = \begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 \\ a_{21}x_1 + a_{22}x_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = b,$$

we can update the vector elements from the bottom to the top. In fact, since x_3 is present only in the computation of the last element, we can overwrite it with $x_3 \leftarrow a_{31}x_1 + a_{32}x_2 + a_{33}x_3$, and then we can update x_2 as $x_2 \leftarrow a_{21}x_1 + a_{22}x_2$, and finally $x_1 \leftarrow a_{11}x_1$.

Complexity: for the computation of the i-th row, we need i multiplications and $i - 1$ sums, and thus $\sum_{i=1}^{n} i + (i - 1) = n(n + 1) - n = n^2$ flops (exactly the same as the triangular solver with vector right hand side).

B.3 Triangular solver

B.3.1 Vector RHS

Implemented by the BLAS routine `dtrsv`, performing the system solution $x \leftarrow A^{-1}x$ or $x \leftarrow (A')^{-1}x$, where A is an upper or lower triangular matrix and x is a vector.

It is based on the fact that, for a generic lower triangular matrix of size 3×3 (the case of the upper triangular matrix being analogous),

$$Ax = \begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 \\ a_{21}x_1 + a_{22}x_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = b,$$

and thus the first element x_1 can be obtained directly as $x_1 = b_1/a_{11}$, the second x_2 substituting the value of x_1 and solving $x_2 = (b_2 - a_{21}x_1)/a_{22}$, the third x_3 substituting the value of x_1 and x_2 and solving $x_3 = (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}$, and so on for larger matrices.

Complexity: for the computation of the i-th row, we need $(i - 1)$ multiplications and subtractions, and 1 division, and thus $\sum_{i=1}^{n} 2(i-1)+1 = n(n-1)+n = n^2$ flops (exactly the same as the triangular matrix-vector multiplication).

B.3.2 Matrix RHS

Implemented by the BLAS routine `dtrsm`, performing the system solution $B \leftarrow \alpha \cdot op(A)^{-1} \cdot B$ or $B \leftarrow \alpha \cdot B \cdot op(A)^{-1}$, where $op(A) = A$ or $op(A) = A'$, A is $n \times n$ and B is respectively $m \times n$ or $n \times m$.

Complexity: $n^2m + nm$ (B rectangular); $n^3 + n^2$ (B squared) flops. It can be reduced up to n^2m and n^3 if $|\alpha = 1|$.

B.4 Cholesky factorization

Implemented by the LAPACK routines `dpotrf` (blocked version) and `dpotf2` (unblocked version), performing $H \leftarrow L$ or $H \leftarrow U$, where H is a symmetric positive definite matrix and L and U are the respectively the lower and upper triangular factors such that $L \cdot L' = H$ and $U' \cdot U = H$.

We can write two different algorithms, considering a matrix of 2×2 blocks or 3×3 blocks; in the following we only consider algorithms for the computation of the upper triangular factor, the algorithms for the lower triangular one being analogous.

2×2 **algorithm** The first algorithm is based on the fact that, given the general upper triangular matrix represented as the 2×2 block matrix

$$U = \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix},$$

then the matrix H can be written as the product

$$\begin{aligned} U'U &= \begin{bmatrix} U'_{11} & 0 \\ U'_{12} & U'_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} U'_{11}U_{11} & U'_{11}U_{12} \\ * & U'_{12}U_{12} + U'_{22}U_{22} \end{bmatrix} = \\ &= \begin{bmatrix} H_{11} & H_{12} \\ * & H_{22} \end{bmatrix} = H. \end{aligned}$$

The algorithm consist in the Cholesky factorization of H_{11} obtaining U_{11}, the solution of the triangular system of equations $U'_{11}U_{12} = H_{12}$ for U_{12}, the subtraction of the symmetric product $U'_{12}U_{12}$ to H_{22}, and finally the Cholesky factorization of the updated matrix $H_{22} - U'_{12}U_{12}$.

The algorithm can be carried on in place: if, at the beginning of a generic iteration, H contains the lower triangular part of the positive definite matrix, and U_{11} contains the already computed part of the lower factor, we are in the situation

$$\begin{bmatrix} U_{11} & H_{12} \\ * & H_{22} \end{bmatrix},$$

U_{12} can be found solving the triangular system $U'_{11}U_{12} = H_{12}$,

$$\begin{bmatrix} U_{11} & (U'_{11})^{-1}H_{12} \\ * & H_{22} \end{bmatrix} = \begin{bmatrix} U_{11} & U_{12} \\ * & H_{22} \end{bmatrix},$$

and finally U_{22} Cholesky factorizing $H_{22} - U'_{12}U_{12}$,

$$\begin{bmatrix} U_{11} & U_{12} \\ * & H_{22} - U'_{12}U_{12} \end{bmatrix} \Rightarrow \begin{bmatrix} U_{11} & U_{12} \\ * & U_{22} \end{bmatrix}.$$

It is possible to write iterative versions of this algorithm: in this case we compute the Cholesky factorization of squared matrices of increasing size, where the increase is 1 for the unblocked version and N_B for the blocked version. In any case, at the beginning of the general iteration U_{11} is the Cholesky factor already computed in the previous iterations. In the unblocked version, H_{12} is a column vector and H_{22} is a scalar, and U_{22} is computed as $U_{22} = \sqrt{H_{22} - U_{12}' U_{12}}$. In the blocked version, H_{12} is a rectangular matrix with N_B columns and H_{22} is a squared matrix of size N_B; U_{22} is computed calling the unblocked routine.

It is also possible to write a recursive algorithm: in this case the H matrix is divided into blocks such that H_{12} is roughly squared. The algorithm is then recursively called on the block H_{11} to obtain U_{11}, then U_{12} is computed as usual, as $U_{12} = (U_{11}')^{-1} H_{12}$, and then the algorithm is recursively called on the updated block $H_{22} - U_{12}' U_{12}$, to obtain U_{22}. As base case can be used a trivial algorithm on matrices of size 1×1 or 2×2 or larger, or a call to an iterative algorithm.

Complexity (unblocked version): at the i-th iteration, for the factorization of the matrix of size i given the factor of size $i-1$, we need $i-1$ flops to solve $U_{11}' U_{12} = H_{12}$, $2(i-1)$ to compute $H_{22} - U_{12}' U_{12}$ (one multiplication and a subtraction for each element of U_{12}), and finally one division to invert the scalar $H_{22} - U_{12}' U_{12}$ and obtain U_{22}. Thus we have $\sum_{i=1}^n (i-1)^2 + 2(i-1) + 1 = \sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{1}{3} n^3 + \frac{1}{2} n^2 + \frac{1}{6} n$ flops.

3×3 **algorithm** Another version of the algorithm (the one implemented by LAPACK) can be obtained considering the general upper triangular matrix represented as the 3×3 block matrix

$$U = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix},$$

the matrix H can be then written as the product

$$H = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ * & H_{22} & H_{23} \\ * & * & H_{33} \end{bmatrix} = U'U = \begin{bmatrix} U_{11}' & 0 & 0 \\ U_{12}' & U_{22}' & 0 \\ U_{13}' & U_{23}' & U_{33}' \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} =$$

$$= \begin{bmatrix} U_{11}' U_{11} & U_{11}' U_{12} & U_{11}' U_{13} \\ * & U_{12}' U_{12} + U_{22}' U_{22} & U_{12}' U_{13} + U_{22}' U_{23} \\ * & * & U_{13}' U_{13} + U_{23}' U_{23} + U_{33}' U_{33} \end{bmatrix}.$$

In an iterative version of the algorithm, the upper factor U is computed one block row at a time: at the beginning of the general iteration, we are in the situation

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ * & H_{22} & H_{23} \\ * & * & H_{33} \end{bmatrix},$$

where the first block row as already be computed in the previous iterations. The element U_{22} is found by factorizing the updated matrix $H_{22} - U_{12}' U_{12}$,

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ * & H_{22} - U_{12}' U_{12} & H_{23} \\ * & * & H_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ * & U_{22} & H_{23} \\ * & * & H_{33} \end{bmatrix},$$

and U_{23} is found by subtracting the product $U'_{12}U_{13}$ to H_{23}, and then by solving the triangular linear system $U_{23} = (U'_{22})^{-1}(H_{23} - U'_{12}U_{13})$,

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ * & U_{22} & (U'_{22})^{-1}(H_{23} - U'_{12}U_{13}) \\ * & * & H_{33} \end{bmatrix} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ * & U_{22} & U_{23} \\ * & * & H_{33} \end{bmatrix}.$$

Again we can consider blocked and unblocked versions of the algorithm.

It is also possible to write a recursive version of the algorithm: the routine calls itself 3 times on matrices of size approximately $n/3$. The first step is the computation of U_{11} calling the routine itself on the sub-matrix H_{11}.

$$\begin{bmatrix} H_{11} & H_{12} & H_{13} \\ * & H_{22} & H_{23} \\ * & * & H_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} U_{11} & H_{12} & H_{13} \\ * & H_{22} & H_{23} \\ * & * & H_{33} \end{bmatrix}.$$

Then the systems $U'_{11}U_{12} = H_{12}$ and $U'_{11}U_{13} = H_{13}$ are solved,

$$\begin{bmatrix} U_{11} & (U'_{11})^{-1}H_{12} & (U'_{11})^{-1}H_{13} \\ * & H_{22} & H_{23} \\ * & * & H_{33} \end{bmatrix} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ * & H_{22} & H_{23} \\ * & * & H_{33} \end{bmatrix}.$$

The element U_{22} is found calling the routine itself on the updated matrix $H_{22} - U'_{12}U_{12}$,

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ * & H_{22} - U'_{12}U_{12} & H_{23} \\ * & * & H_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ * & U_{22} & H_{23} \\ * & * & H_{33} \end{bmatrix},$$

and U_{23} is the found subtracting the product $U'_{12}U_{13}$ to H_{23}, and then solving the triangular linear system $U_{23} = (U'_{22})^{-1}(H_{23} - U'_{12}U_{13})$,

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ * & U_{22} & (U'_{22})^{-1}(H_{23} - U'_{12}U_{13}) \\ * & * & H_{33} \end{bmatrix} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ * & U_{22} & U_{23} \\ * & * & H_{33} \end{bmatrix}.$$

Finally U_{33} is found calling the routine itself on the updated matrix $H_{33} - U'_{13}U_{13} - U'_{23}U_{23}$,

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ * & U_{22} & U_{23} \\ * & * & H_{33} - U'_{13}U_{13} - U'_{23}U_{23} \end{bmatrix} \Rightarrow \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ * & U_{22} & U_{23} \\ * & * & U_{33} \end{bmatrix}.$$

Complexity (unblocked version): for the computation of the i-th row, U_{12} is $(i-1) \times 1$, U_{13} is $(i-1)(n-i)$, U_{22} is 1×1 and U_{23} is $1 \times (n-i)$. We thus have $i-1$ multiplications and subtractions for the computation of $H_{22} - U'_{12}U_{12}$, 1 square root, $(i-1)(n-i)$ multiplications and subtractions for the computation of $H_{23}U'_{12}U_{13}$ and finally $n-i$ divisions for the subsequent computation of $(U'_{22})^{-1}(H_{23}U'_{12}U_{13})$: thus $\sum_{i=1}^{n} 2(i-1) + 1 + 2(i-1)(n-1) + (n-i) = \sum_{i=1}^{n} -2i^2 + (2n+3)i - n - 1 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$ flops, the same as the previous version.

B.5 Triangular matrix inversion

It is implemented by the LAPACK routines `dtrtri` (blocked version) and `dtrti2` (unblocked version), performing $U \leftarrow U^{-1}$ or $L \leftarrow L^{-1}$, where U and L are respectively an upper and lower triangular matrix.

It is based on the fact that, in the example of a generic upper triangular matrix U represented into blocked form

$$U = \begin{bmatrix} A & B \\ 0 & C \end{bmatrix},$$

if A and C are invertible, then

$$\begin{bmatrix} A & B \\ 0 & C \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} & -A^{-1}BC^{-1} \\ 0 & C^{-1} \end{bmatrix}.$$

The algorithm consists in the inversion of A, inversion of C, and finally computation of the product $-A^{-1}BC^{-1}$. The algorithm can be carried on in place, using the standard BLAS routines. In fact, if at the beginning of the generic iteration A^{-1} contains the already computed part of the inverse matrix, we can use the routine `dtrmm` to compute the product $-A^{-1}B$ overwriting B, then compute the inverse of the matrix C, and finally use `dtrmm` to compute the product $(-A^{-1}B)C^{-1}$.

The algorithm type depends on how the blocks are chosen: in an iterative algorithm, A^{-1} is the already computed part of the invert matrix at the beginning of the generic iteration. Then the algorithm computes the inverse of C: in the unblocked version, C is a scalar and thus its inversion is trivial; in the blocked version, the inversion of C is computed calling an unblocked routine.

A recursive algorithm can be written using the routine itself to compute the inverse of the sub-matrices A and C: their size is chosen to be roughly the same. As base case can be used a trivial algorithm on matrices of size 1×1 or 2×2, or an iterative algorithm.

Complexity (unblocked version): for the computation of the i-th column, $(i-1)^2$ flops for the triangular matrix-vector multiplication $A^{-1}B$, 1 for the inversion of the scalar C, 1 to change the sign into $-C^{-1}$ and $i-1$ to scale the vector $A^{-1}B$ by the scalar $-C^{-1}$. In total $\sum_{i=1}^{n}(i-1)^2+(i-1)+2 = \frac{1}{3}n^3+\frac{5}{3}n$.

Appendix C

BLAS and LAPACK

BLAS (Basic Linear Algebra Subprograms) is a standard interface for basic linear algebra programs. LAPACK (Linear Algebra PACKage) is a collection of routines performing more advanced linear algebra, and it makes use of BLAS routines as sub-routines. The performances of the algorithm considered in this thesis strongly depend on the efficiency of the implementation of these routines. In this chapter we test a number of different implementations.

C.1 BLAS

BLAS is a standard interface containing a number of routines for the solution of basic linear algebra problems, and is used as building block of more advanced libraries, as LAPACK.

It is divided into 3 levels: level 1 contains routines implementing vector-vector operations (requiring $\mathcal{O}(n)$ space and $\mathcal{O}(n)$ time), level 2 contains matrix-vector operations (requiring $\mathcal{O}(n^2)$ space and $\mathcal{O}(n^2)$ time), and level 3 contains matrix-matrix operations (requiring $\mathcal{O}(n^2)$ space and $\mathcal{O}(n^3)$ time).

In the first two levels there is thus little space for optimization, since the computational time is dominated by the time needed to move data between the different memory levels. On the other hand, regarding the level 3 BLAS, it is possible to obtain important performance improvements exploiting data reuse: good implementations allow the processors to operate close to their peak performance.

C.1.1 Testing `dgemm`

The matrix-matrix multiplication algorithm, implemented by BLAS routine `dgemm`, is one of the most important algorithms, and often used as benchmark code to test different BLAS implementations. It performs the computation

$$C \leftarrow \alpha \cdot op(A) \cdot op(B) + \beta \cdot C,$$

where A, B and C are matrices of proper size, $op(A) = A$ or $op(A) = A'$, and α and β are scalars.

In this section we present a number of BLAS versions, and test many different implementation of `dgemm`.

Benchmark code

The performed test consist in the multiplication

$$C \leftarrow op(A) \cdot op(B),$$

for different size of the matrices A, B and C, all squared and of the same size. This means that we consider only the case $\alpha = 1$ and $\beta = 0$. This does not influence the results much, since the choice of a different values for α and β affects the computation time only with a quadratic term, negligible compared to the dominant cubic term.

This test shows that, to obtain top performances, it is necessary to take into account the specifics of the machine where the program runs (number of registers, caches size, bus speed). The high-level structure of the algorithm influences how the computation time scales with the size of the matrices, and the relative code can be written in an high-level language, as C.

Furthermore, the performances strongly depends also in the efficient implementation of the innermost loops. Top performances can be obtained only by the use of assembly code tailored on the specific architecture: in fact even the most efficient algorithm written in C code and optimized using the optimization flags of the `gcc` compiler much slower than the best BLAS implementations.

C code

In this part we test the performance of a simple algorithm for the matrix-matrix multiplication, that is a C translation of the reference NETLIB code of the `dgemm` routine found at `www.netlib.org/blas`.

We show that the performance of the code can improve using the compiler optimization flags, but that can not compete against assembly code. We also discuss some method to improve the performance of the code, for example the use of loop-unrolling and of a blocked version of the algorithm.

The compiler is the default LINUX compiler, `gcc`, and the resulting code is single thread.

Not optimized code The not-optimized C code is a C translation of the official FORTRAN implementation given in `www.netlib.org/blas`.

n_x	CPU time	wall clock time
10	0.01208	0.012
20	0.09088	0.092
30	0.29826	0.298
40	0.6986	0.70
50	1.3518	1.36

Table C.1: C code not optimized (single thread), time in milliseconds.

nx	`ta=n, tb=n`	`ta=t, tb=n`	`ta=t, tb=t`	`ta=n, tb=t`
4	0.001 (0.00099)			
8	0.0064 (0.0065)			
16	0.049 (0.0490)			
32	0.362 (0.367)			
64	2.87 (2.905)			
128	22.6 (22.64)			
256	179.7 (179.9)			
512	1459.0 (1462.0)			
1024	11640.0 (11668.5)			

Table C.2: `dgemm`, not optimized C code (single thread), CPU (wall clock) time in ms.

Optimization flags Using the optimization flags the code performance increases much. In particular, analyzing the assembly code generated using the compiler flag `-S` with and without the use of optimization flags, we notice that the best part of the performance increase comes from a better use of the CPU registers. In fact, the code produced without optimization flags moves the needed data from memory to the register, and then to memory again for each single operation, instead of keep it in the registers for consecutive operations. Other important improvements can come from the simplification of algebraic expressions, and the creation of variables saving the value of common sub-expressions, avoiding their computation multiple times.

n_x	CPU time	wall clock time
10	0.0016	0.002
20	0.0124	0.012
30	0.03886	0.040
40	0.09048	0.090
50	0.1741	0.176

Table C.3: C code optimized using the `-O3` option (single thread), time in milliseconds.

Hand optimized code In this part we test some code produced hand optimizing the code in C, and without the use of optimization flags. Among the performed optimizations, there is the use of the reserved word `register` in the definition of variables, used to indicate to the complier the most used variables: these will be saved in the registers instead of on the stack (if there are enough registers available).

Another optimization is the use of partial unrolling of the innermost loop, for example for this code we use unrolling of module 5.

Code optimized by hand and using optimization flags In this part we test the code optimized by hand, and further optimized using the compiler optimization flag `-O3`. The performance is slightly better than the code optimized

n_x	CPU time	wall clock time
10	0.0032	0.004
20	0.02008	0.020
30	0.05836	0.058
40	0.13018	0.130
50	0.24394	0.240

Table C.4: C code hand optimized (single thread), time in milliseconds.

using only the optimization flags: this shows the utility of loop-unrolling in this case.

n_x	CPU time	wall clock time
10	0.00156	0.002
20	0.0113	0.010
30	0.03392	0.034
40	0.07758	0.078
50	0.14724	0.148

Table C.5: C code hand optimized, further optimized using the `-O3` option (single thread), time in milliseconds.

FORTRAN

The FORTRAN code is the official implementation found in `www.netlib.org/blas`. For the compilation, we used the compiler `gfortran` to compile it: the resulting code is single thread.

Not optimazed code In this part there is a test of the code without any optimization flag. The performance is almost the same as in the case of the not optimized C code.

n_x	CPU time	wall clock time
10	0.0117	0.012
20	0.08864	0.088
30	0.29274	0.294
40	0.6878	0.696
50	1.3403	1.348

Table C.6: Fortran code not optimized (single thread), time in milliseconds.

Optimization flags Using the optimization flag `-O3` in the compilation of the library, it is obtained an important performance improvement. The computation time is roughly the same as the C code optimized using the same optimization flag: in fact `gcc` and `gfortran` are both part of the same GNU compiler collection.

nx	`ta=n, tb=n`	`ta=t, tb=n`	`ta=t, tb=t`	`ta=n, tb=t`
4	0.0002 (0.00024)	0.0002 (0.00027)	0.0002 (0.00026)	0.0002 (0.00024)
8	0.0011 (0.00105)	0.0011 (0.00105)	0.0013 (0.00128)	0.0011 (0.00108)
16	0.006 (0.0065)	0.007 (0.0068)	0.008 (0.0078)	0.007 (0.0074)
32	0.048 (0.048)	0.056 (0.0568)	0.058 (0.0578)	0.057 (0.0564)
64	0.37 (0.372)	0.4 (0.394)	0.45 (0.45)	0.43 (0.435)
128	3.1 (3.18)	3.7 (3.69)	7.4 (7.47)	3.3 (3.37)
256	27.4 (27.4)	32.2 (32.2)	89.5 (89.7)	36.9 (36.9)
512	354.0 (354.0)	325.0 (325.3)	2968.0 (2963.8)	356.0 (356.8)
1024	2750.0 (2756.0)	2630.0 (2633.0)	24930.0 (24975.0)	2790.0 (2790.4)

Table C.7: `dgemm`, NETLIB implementation, FORTRAN code optimized using the `-O3` option (single thread), CPU (wall clock) time in *ms*.

Ubuntu's default BLAS

Ubuntu has its own version of the BLAS library already installed: it is the static library `libblas.a`, and can be used linking with the flag `-lblas`. It is multi thread version, but despite of this, for small system the performance is worst than the one of the NETLIB version (that is single thread). This version shows an important performance drop increasing the matrix size between 36 and 37: this is probably due to an internal change of algorithm.

nx	`ta=n, tb=n`	`ta=t, tb=n`	`ta=t, tb=t`	`ta=n, tb=t`
4	0.0007 (0.00069)			
8	0.0013 (0.00131)			
16	0.007 (0.0066)			
32	0.040 (0.0402)			
64	0.43 (0.269)			
128	2.8 (1.69)			
256	21.0 (12.1)			
512	163.0 (92.9)			
1024	1290.0 (701.4)			

Table C.8: `dgemm`, Ubuntu standard BLAS implementation `libblas.a` library (multi thread), CPU (wall clock) time in *ms*.

ATLAS

ATLAS is an optimized library, implementing all the BLAS and part of the LAPACK routines. It is written in C, but can make use of assembly code for the innermost loops, if available. The optimization strategy is based on long series of test, used to empirically find the best values for the routines parameters, as block size in blocked algorithms.

Non optimized library A pre-build non-optimized version can be installed typing the command `sudo apt-get install libatlas-base-dev` on the shell. The library is the static library `libf77blas.a`: to use it, in linking phase the flag `-lf77blas` has to be used.

nx	`ta=n, tb=n`	`ta=t, tb=n`	`ta=t, tb=t`	`ta=n, tb=t`
4	0.0005 (0.00049)			
8	0.0011 (0.00111)			
16	0.007 (0.0063)			
32	0.040 (0.0401)			
64	0.31 (0.309)			
128	2.43 (2.43)			
256	20.2 (20.2)			
512	161.0 (160.6)			
1024	1251.0 (1253.9)			

Table C.9: `dgemm`, not optimized ATLAS `libf77blas.a` library (single thread), CPU (wall clock) time in *ms*.

Optimization Better performances should be obtained compiling the ATLAS library from source code. ATLAS installation guide is [Wha]. The process includes a long series of tests, used to tune the parameters to the specifications of the current machine. The CPU throttling must be switch off during the tests.

We have not build the optimized library yet, it is part of future work. Anyway, since we do not currently have an efficient assembly version of the innermost routines, the library is likely to be less efficient than the one presented in the following.

GotoBLAS2

GotoBLAS2 is an highly optimized implementation of BLAS written by Kazushige Goto, researcher at the Texas Advanced Computing Center of The University of Texas between 2003 and 2005. The developed software can be used free of charge for academic, research, experimental or personal use. The software can be downloaded from the link `www.tacc.utexas.edu/tacc-projects/gotoblas2/`. An article about efficient implementation of `dgemm` is [GVDG].

Once extracted, if the target architecture is supported, the libraries can be build just opening a shell and typing the command `make` from the `GotoBLAS2` directory. The installation process takes care of detect the machine features and tests the resulting libraries. In our case, the build library seems to have some problem operating on complex numbers.

In Goto's implementation, the A and B matrices are usually carefully packed into the smaller sub-matrices $\tilde{A}$ and $\tilde{B}$, respectively placed in L2 and L1 cache. The code performing this packing and unpacking is written in C. The inner kernels operating on these packed matrices are hand written by Goto in highly optimized assembly code: he wrote these kernels for a number of different architectures between 2003 and 2005. His approach is somehow the opposite compared to ATLAS's one: in fact, the optimization is based on theoretical considerations instead of on a series of empirical tests.

GotoBLAS makes extensive use of the SSE and SSE2 instruction sets, and thus to achieve the best performances it needs the data to be 16 bit aligned. In fact the floating points operations are performed on 128 bit long registers, that can hold a vector of two double-precision or a vector of 4 single-precision floating points. The same operation can be performed in parallel on all the elements of the vector at the same time. Using the most efficient instruction for data movement, the data are read from and written to memory as a contiguous block of 128 bits, whose starting address is multiple of 16 (i.e. 16 bit aligned).

The library is called `libgoto2.a`. For large matrices GotoBLAS is the most efficient version we tested, even more efficient than MKL (version 10.0.3).

nx	`ta=n, tb=n`	`ta=t, tb=n`	`ta=t, tb=t`	`ta=n, tb=t`
4	0.0012 (0.00059)	0.0011 (0.00056)	0.0012 (0.00061)	0.0012 (0.00061)
8	0.0069 (0.0035)	0.0067 (0.0033)	0.007 (0.0035)	0.0069 (0.00348)
16	0.011 (0.0058)	0.011 (0.00575)	0.011 (0.00611)	0.011 (0.00606)
32	0.027 (0.0137)	0.027 (0.0136)	0.027 (0.0145)	0.029 (0.0151)
64	0.130 (0.0667)	0.130 (0.0661)	0.150 (0.0763)	0.150 (0.0772)
128	1.1 (0.52)	0.9 (0.468)	1.1 (0.525)	1.0 (0.525)
256	9.4 (4.81)	8.6 (4.41)	10.9 (5.51)	10.8 (5.45)
512	77.0 (39.5)	64.0 (32.2)	84.0 (41.9)	80.0 (40.5)
1024	529.0 (286.0)	490.0 (248.0)	570.0 (287.1)	550.0 (276.3)

Table C.10: `dgemm`, GotoBLAS2 `ligoto2.a` library (multi thread), in *ms*.

MKL 10.0.3

The Matrix Kernel Library (MKL) is a proprietary implementation of BLAS by Intel. It is, together with Goto's BLAS, the most efficient implementation available for the architecture of out test machine.

The MKL library makes extensive use of SSE and SSE2 instruction sets, and thus to obtain the best performances needs the data to be 16 bit aligned.

nx	`ta=n, tb=n`	`ta=t, tb=n`	`ta=t, tb=t`	`ta=n, tb=t`
4	0.0005 (0.00053)	0 0005 (0.00054)	0.0006 (0.00062)	0.0006 (0.00059)
8	0.0011 (0.00115)	0.0011 (0.00115)	0.0012 (0.00115)	0.0012 (0.00116)
16	0.003 (0.00319)	0.003 (0.00322)	0.003 (0.00326)	0.003 (0.00324)
32	0.026 (0.0136)	0.027 (0.0136)	0.027 (0.0138)	0.027 (0.0139)
64	0.140 (0.077)	0.140 (0.0736)	0.150 (0.0752)	0.150 (0.0785)
128	1.1 (0.64)	1.2 (0.737)	1.1 (0.536)	1.3 (0.773)
256	12.8 (6.6)	12.1 (6.21)	12.2 (6.3)	12.5 (6.5)
512	107.0 (55.7)	102.0 (52.4)	105.0 (54.4)	108.0 (55.7)
1024	808.0 (410.7)	780.0 (400.4)	810.0 (421.8)	830.0 (427.4)

Table C.11: `dgemm`, MKL implemetation (multi thread), CPU (wall clock) time in *ms*.

Comparison

The best solution for matrices small enough to totally fit in cache together is the NETLIB implementation, since its code is straightforward. For large matrices the best solution is the Goto's implementation. If just one implementation has to be chosen, the best choice is probably MKL by Intel, since its performance is quite close to the best one for each matrix size: thus we test only this implementation in the next sections.

Regarding the transpose state of the factor matrices, the combination A transposed and B normal is the best one, making better use of contiguous data.

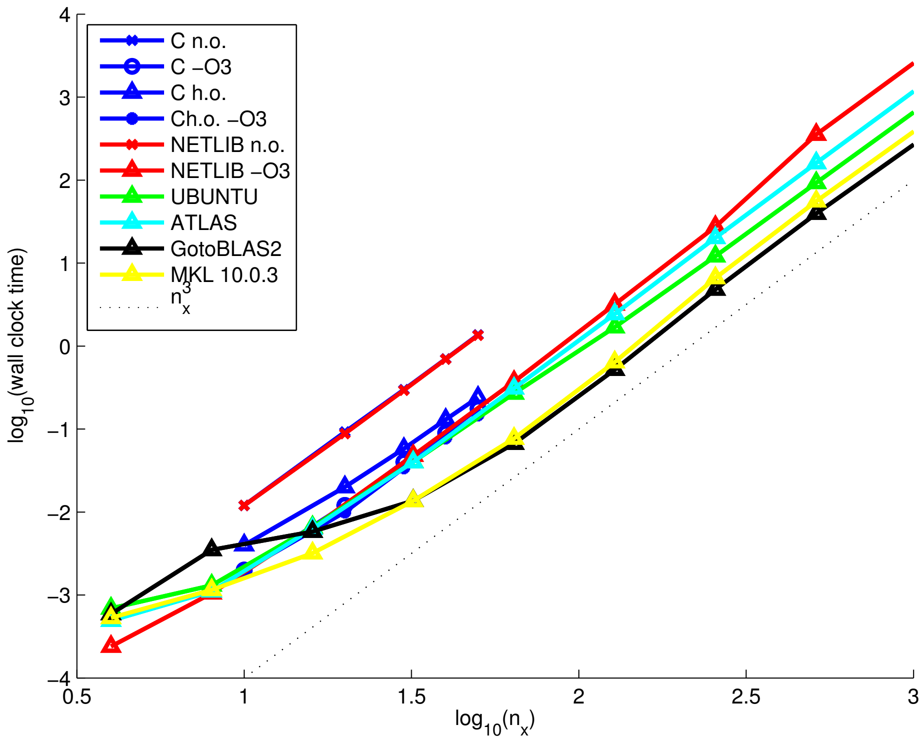


Figure C.1: Comparison of the different implementations of `dgemm`: C code not optimized (C n.o.), C code optimized with the flag `-O3` (C -O3), C code hand optimized (C h.o.), C code hand optimized and further optimized with the flag `-O3` (C h.o. -O3), NETLIB (FORTRAN) code not optimized (NETLIB n.o.), NETLIB (FORTRAN) code optimized with the flag `-O3` (NETLIB -O3), Ubuntu's BLAS (UBUNTU), non optimizes ATLAS (ATLAS), GotoBLAS2 (GotoBLAS2), MKL 10.0.3 (MKL 10.0.3). The time is wall clock time, in milliseconds.

C.1.2 Testing `dsyrk`

The BLAS routine `dsyrk` performs the rank-k update of a symmetric matrix:

$$C \leftarrow \alpha \cdot A \cdot A' + \beta C \quad \text{or} \quad C \leftarrow \alpha \cdot A' \cdot A + \beta C$$

where C is a symmetric $n \times n$ matrix and A is a $n \times k$ or $k \times n$ matrix, and α and β are scalars. In other words, it computes the upper or lower triangular part of the product of a matrix of size $n \times k$ by its transpose, and adds the result to the upper or lower triangular part of the $n \times n$ symmetric matrix.

As benchmark code, we choose to test the computations

$$C \leftarrow A \cdot A' \quad \text{and} \quad C \leftarrow A' \cdot A$$

for $\alpha = 1$ and $\beta = 0$, since other parameters values counts at most for a quadratic term in the computation time. As k size, we chose $k = n$.

nx	`uplo=u, ta=n`	`uplo=u, ta=t`	`uplo=l, ta=n`	`uplo=l, ta=t`
4	0.0003 (0.0003)	0.0003 (0.0003)	0.0003 (0.0003)	0.0003 (0.00032)
8	0.0006 (0.00062)	0.0006 (0.00061)	0.0007 (0.00062)	0.0006 (0.00063)
16	0.003 (0.00252)	0.002 (0.00231)	0.003 (0.00282)	0.003 (0.00233)
32	0.025 (0.0129)	0.021 (0.0112)	0.025 (0.0132)	0.021 (0.0113)
64	0.12 (0.0622)	0.1 (0.0533)	0.12 (0.0643)	0.1 (0.0542)
128	0.7 (0.396)	0.8 (0.477)	0.9 (0.493)	0.7 (0.35)
256	5.9 (2.97)	4.9 (2.49)	6.2 (3.19)	5.3 (2.72)
512	52.0 (26.7)	42.0 (21.1)	55.0 (27.6)	44.0 (22.4)
1024	420.0 (209.8)	340.0 (168.8)	430.0 (212.2)	390.0 (193.4)

Table C.12: `dsyrk`, MKL implemetation (multi thread), CPU (wall clock) time in ms.

C.1.3 Testing `dtrmm`

The BLAS routine `dtrmm` performs the product of a triangular matrix and a general matrix,

$$B \leftarrow \alpha \cdot op(A) \cdot B \quad \text{or} \quad B \leftarrow \alpha \cdot B \cdot op(A)$$

where $op(A)$ is A or A', B is a $m \times n$ matrix and A is a $m \times m$ or $n \times n$ upper or lower, unit diagonal or not, triangular matrix, and α is a scalar.

As benchmark code, we choose to test the operations

$$B \leftarrow A \cdot B \quad \text{and} \quad B \leftarrow A' \cdot B \quad \text{and} \quad B \leftarrow B \cdot A \quad \text{and} \quad B \leftarrow B \cdot A'$$

for both cases of B upper and lower, not unit diagonal triangular matrix.

nx	`uplo=u, ta=n`	`uplo=u, ta=t`	`uplo=l, ta=n`	`uplo=l, ta=t`
4	0.00014 (0.00014)	0.0006 (0.00065)	0.00014 (0.00014)	0.00012 (0.000126)
8	0.0018 (0.00185)	0.0010 (0.00103)	0.0018 (0.00185)	0.0019 (0.00194)
16	0.005 (0.00496)	0.004 (0.00398)	0.004 (0.0047)	0.006 (0.00559)
32	0.030 (0.0150)	0.028 (0.014)	0.029 (0.0144)	0.033 (0.0169)
64	0.14 (0.068)	0.14 (0.071)	0.13 (0.0659)	0.16 (0.0805)
128	0.8 (0.427)	0.8 (0.403)	0.8 (0.414)	0.8 (0.430)
256	6.2 (3.09)	5.5 (2.78)	6.1 (3.08)	5.4 (2.75)
512	49.0 (24.2)	46.0 (22.7)	51.0 (25.7)	40.0 (20.2)
1024	380.0 (192.4)	340.0 (171.4)	390.0 (193.9)	340.0 (169.7)

Table C.13: `dtrmm`, `side='l'`, MKL implemetation (multi thread), CPU (wall clock) time in *ms*.

nx	`uplo=u, ta=n`	`uplo=u, ta=t`	`uplo=l, ta=n`	`uplo=l, ta=t`
4	0 0014 (0.00141)	0 0015 (0.0015)	0.0014 (0.00143)	0.0014 (0.00143)
8	0.0023 (0.00233)	0.0021 (0.00211)	0.0021 (0.00211)	0.0024 (0.00238)
16	0.005 (0.00546)	0.005 (0.00459)	0.005 (0.00542)	0.006 (0.00638)
32	0.039 (0.0197)	0.048 (0.0241)	0.039 (0.0199)	0.051 (0.0261)
64	0.15 (0.0776)	0.16 (0.0809)	0.15 (0.0775)	0.17 (0.0864)
128	0.8 (0.428)	0.8 (0.399)	0.8 (0.42)	0.8 (0.403)
256	5.3 (2.69)	5.1 (2.59)	5.3 (2.67)	5.3 (2.65)
512	45.0 (22.7)	47.0 (23.5)	49.0 (24.5)	48.0 (24.4)
1024	480.0 (240.2)	480.0 (244.8)	470.0 (238.6)	480.0 (243.3)

Table C.14: `dtrmm`, `side='r'`, MKL implemetation (multi thread), CPU (wall clock) time in *ms*.

C.1.4 Testing `dtrsm`

The BLAS routine `dtrsm` performs the solution of a triangular system of linear equations, where the right hand side is a matrix,

$$B \leftarrow \alpha \cdot op(A)^{-1} \cdot B \quad \text{or} \quad B \leftarrow \alpha \cdot B \cdot op(A)^{-1}$$

where $op(A)$ is A or A', B is a matrix of size $n \times m$, and A is a $m \times m$ or $n \times n$, upper or lower, diagonal unit or not, triangular matrix.

As benchmark code, we choose to perform the operations

$$B \leftarrow A^{-1} \cdot B \quad \text{and} \quad B \leftarrow (A')^{-1} \cdot B \quad \text{and} \quad B \leftarrow B \cdot A^{-1} \quad \text{and} \quad B \leftarrow B \cdot (A')^{-1}$$

for both cases of B upper and lower, not unit diagonal triangular matrix.

nx	`uplo=u, ta=n`	`uplo=u, ta=t`	`uplo=l, ta=n`	`uplo=l, ta=t`
4	0 0009 (0.00091)	0 0009 (0.0009)	0.0009 (0.00091)	0.0009 (0.0009)
8	0.0018 (0.00184)	0.0019 (0.00192)	0.0019 (0.00189)	0.0019 (0.0019)
16	0.006 (0.00643)	0.007 (0.00647)	0.007 (0.00645)	0.007 (0.00656)
32	0.037 (0.0187)	0.038 (0.0191)	0.038 (0.0191)	0.038 (0.0191)
64	0.16 (0.0816)	0.17 (0.0851)	0.16 (0.082)	0.16 (0.0829)
128	0.8 (0.486)	0.9 (0.447)	0.8 (0.431)	0.9 (0.536)
256	5.4 (2.86)	5.6 (2.89)	6.6 (3.35)	5.8 (2.93)
512	41.0 (21.8)	45.0 (23.2)	67.0 (33.2)	44.0 (22.5)
1024	280.0 (144.7)	380.0 (198.6)	550.0 (276.6)	380.0 (192.7)

Table C.15: `dtrsm`, `side='l'`, MKL implemetation (multi thread), CPU (wall clock) time in ms.

nx	`uplo=u, ta=n`	`uplo=u, ta=t`	`uplo=l, ta=n`	`uplo=l, ta=t`
4	0 0013 (0.00131)	0 0013 (0.00133)	0.0013 (0.00135)	0.0013 (0.00132)
8	0.002 (0.00196)	0.0021 (0.00213)	0.0022 (0.00216)	0.0021 (0.00201)
16	0.004 (0.00439)	0.005 (0.00524)	0.006 (0.00592)	0.006 (0.00531)
32	0.033 (0.0167)	0.024 (0.024)	0.024 (0.0241)	0.024 (0.024)
64	0.21 (0.206)	0.36 (0.182)	0.25 (0.125)	0.37 (0.187)
128	1.1 (0.544)	1.2 (0.617)	1.1 (0.548)	1.2 (0.627)
256	5.8 (2.9)	5.9 (2.99)	5.8 (2.91)	6.0 (2.99)
512	45.0 (22.6)	49.0 (24.4)	45.0 (22.6)	48.0 (24.5)
1024	410.0 (205.5)	430.0 (214.6)	410.0 (206.1)	430.0 (214.)

Table C.16: `dtrsm`, `side='r'`, MKL implemetation (multi thread), CPU (wall clock) time in ms.

C.2 LAPACK

LAPACK is a library containing a number of routines for linear algebra problems, such as matrix factorizations, solution of linear systems and linear least squares, eigenvalue problems and singular value decomposition. The code is currently written in `Fortran 90`.

LAPACK depends upon BLAS to perform the basic linear algebra operations, such as matrix-matrix and matrix-vector multiplication: LAPACK performance thus strongly depends on the BLAS library it links against. LAPACK's routines are iterative algorithms, and usually there are two versions for each algorithm, an unblocked one (using the level 2 BLAS) and a blocked one (using the level 3 BLAS and the unblocked version as subroutine).

We teste a number of versions of the LAPACK library, using pre-built libraries or compiling from the source code, linking against different BLAS libraries.

By downloading ATLAS, we also download a pre-build sub-set of the LAPACK library linking against `libf77blas.a`. Another possibility is the installation of Ubuntu's default LAPACK, linking against `libblas.a`.

It is also possible to compile from source code, and choose the BLAS library to link against. The source code can be downloaded from the website `www.netlib.org/lapack`, in the compressed file `lapack-3.4.0.tgz` (the current latest version in 3.4.0). After extraction, we have to enter into the just created folder `lapack-3.4.0`, and rename the file `make.inc.example` into `make.inc` (it is an example file to install LAPACK on Linux). Then we have to edit the file `make.inc`, writing in the row `BLASLIB` the path of the BLAS library we want LAPACK to link against.[1] After this, the library can be build by opening a shell, entering in the folder `lapack-3.4.0` and typing the command `make all` (this is building the library and testing the BLAS and LAPACK implementations).

This produces a library called `liblapack.a`: it is possible to move the library where we want to place it. In order to use it, in linking phase we need to link our programs against the LAPACK library, the BLAS library and the `gfortran` library (with the command `-lgfortran`), written in this order.

We decided to test 4 LAPACK implementations: the pre-build Ubuntu's LAPACK, the pre-build ATLAS LAPACK, the compiled library linking against the standard FORTRAN BLAS and the compiled library linking against the MKL BLAS; for some routines we are also writing a recursive version.

C.2.1 Testing `dpotrf`

In this section we test different implementation of Cholesky factorization. We test the algorithm based on the 2×2 and 3×3 blocks matrix, in both iterative blocked and recursive versions (the LAPACK routine `dpotrf` is the iterative version of the algorithm based on the 3×3 blocks matrix) and the LAPACK routine `dpotf2` (an unblocked version of the algorithm based on the 3×3 blocks matrix).

Regarding the recursive algorithms, as base cases are use routines directly factorizing matrices of size 1×1, 2×2, 3×3 and 4×4, or the `dpotf2` routine.

[1]It is possible to customize many other options, like the compiler (the default is `gfortran`) and the optimization level (the default is `O2`).

Regarding the blocked iterative routines, we tested different block size, and in the following we present the best one.

All the routines are tested linking toward the standard BLAS implementation found at `www.netlib.org/blas`, and toward the optimized library MKL by Intel. A description of the algorithms is given in appendix B.

In the following, we consider the general symmetric positive definite matrix

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

There are two version of the algorithm: one returning the lower triangular matrix L such that

$$H = LL' = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}' & L_{21}' \\ 0 & L_{22}' \end{bmatrix}$$

and one returning the upper triangular matrix U such that

$$H = U'U = \begin{bmatrix} U_{11}' & 0 \\ U_{12}' & U_{22}' \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}.$$

Upper Factor

In this section we compare a number of algorithm for the computation of the upper factor.

In the following tables, `it2` and `it3` stand respectively for iterative blocked algorithm based on the 2×2 and 3×3 blocks matrix form, and `ric2` and `ric3` for the recursive algorithm based on the 2×2 and 3×3 blocks matrix form.

n	`dpotf2`	`it2` ($N_B = 64$)	`it3` ($N_B = 64$)	`ric2`	`ric3`
4	0.00091	0.00091	0.00091	0.00029	0.00029
8	0.0025	0.0025	0.0025	0.0019	0.0032
16	0.0072	0.0072	0.0072	0.0062	0.013
32	0.023	0.023	0.023	0.018	0.024
64	0.1	0.1	0.1	0.13	0.22
128	0.68	0.67	0.69	0.47	0.65
256	5.37	2.9	2.77	2.24	2.67
512	44.9	18.8	14.5	14.6	15.5
1024	567.0	167.0	102.0	120.0	127.0

Table C.17: Cholesky factorization computing the upper factor, MKL BLAS, CPU times in ms.

We notice that, for very small matrices, the NETLIB BLAS has an advantage over the MKL BLAS, since the matrices are small enougth to fit in cache, and the NETLIB implementation is straighforward. We also notice that recursive algorithms has an advantage over iterative ones, since matrices up to size 4×4 are solved directly.

For medium size matrices, the recursive methods show an advantage when linking toward the optimized MKL BLAS, while block iterative methods perform better if we just have a non-optimized BLAS like NETLIB. This is probably due to the fact that an optimized BLAS is already internally using a block version

n	`dpotf2`	`it2` ($N_B = 128$)	`it3` ($N_B = 64$)	`ric2`	`ric3`
4	0.00064	0.00064	0.00064	0.00029	0.00029
8	0.0016	0.0016	0.0016	0.0015	0.0019
16	0.0055	0.0055	0.0055	0.0069	0.0085
32	0.02	0.02	0.02	0.029	0.032
64	0.1	0.1	0.1	0.14	0.15
128	0.7	0.7	0.74	0.8	0.84
256	5.33	5.82	5.45	5.54	5.74
512	42.8	45.3	41.8	41.9	42.0
1024	564.0	496.0	319.0	521.0	519.0

Table C.18: Cholesky factorization computing the upper factor, NETLIB BLAS, CPU times in ms.

(with optimal block size) for its algorithm, and thus there is an advantage in using matrices as large as possible. On the contrary, the NETLIB implementation uses unblocked algorithm, and thus the choice the right block size in the blocked iterative version of the Cholesky factorization is an advantage, since the sub-matrices can fit better in the cache, and the performance is then quite sensitive in the block size.

Overall, the best solution is `ric2` and NETLIB BLAS for small matrices, `ric2` and MKL BLAS for medium matrices, `it3` and MKL BLAS for large matrices. If we just have the NETLIB BLAS, then the best solution for medium and large matrices is `it3`, implemented by the LAPACK routine `dpotrf`.

Lower Factor

In this section we compare a number of algorithm for the computation of the lower factor.

n	`dpotf2`	`it2` ($N_B = 128$)	`it3` ($N_B = 128$)	`ric2`	`ric3`
4	0.00089	0.00089	0.00089	0.00028	0.00028
8	0.0025	0.0025	0.0025	0.0023	0.0042
16	0.0066	0.0066	0.0066	0.0078	0.016
32	0.022	0.022	0.022	0.024	0.028
64	0.09	0.09	0.09	0.18	0.24
128	0.63	0.63	0.63	0.75	0.89
256	4.75	4.38	4.64	3.4	3.97
512	41.0	21.6	20.3	19.7	19.0
1024	510.0	155.0	173.0	150.0	192.0

Table C.19: Cholesky factorization computing the lower factor, MKL BLAS, CPU times in ms.

This time the results are quite different in the case of optimized and not-optimized BLAS implementation. In particular, using the MKL one, the iterative unblocked `dpotf2` routine is the fastest for medium matrices, while the recursive algorithm `ric2` is the fastest for small and large matrices.

Regarding the not-optimized NETLIB one, the best solution is the recursive

n	`dpotf2`	`it2` ($N_B = 64$)	`it3` ($N_B = 64$)	`ric2`	`ric3`
4	0.00061	0.00061	0.00061	0.00028	0.00028
8	0.0016	0.0016	0.0016	0.0011	0.0015
16	0.005	0.005	0.005	0.0047	0.0062
32	0.018	0.018	0.018	0.019	0.02
64	0.09	0.09	0.09	0.1	0.11
128	0.63	0.67	0.68	0.66	0.68
256	4.59	4.66	4.77	4.76	4.82
512	38.1	39.7	37.6	36.6	36.8
1024	501.0	1390.0	542.0	713.0	659.0

Table C.20: Cholesky factorization computing the lower factor, NETLIB BLAS, CPU times in ms.

`ric2` for small matrices and the unblocked iterative version implemented in the LAPACK routine `dpotf2` for medium and large matrices. It is notable the fact that none of the iterative blocked or the recursive can outperform the iterative unblocked for large matrices.

Appendix D

Time tables

D.1 Direct sparse solvers

			PARDISO		MA57	
n_x	n_u	N	CPU time	wall clock time	CPU time	wall clock time
50	5	10	0.068400	0.035516	0.029000	0.014991
50	5	20	0.196000	0.100256	0.060000	0.031467
50	5	30	0.190800	0.097424	0.093000	0.048538
50	5	40	0.266000	0.136725	0.121000	0.064692
50	5	50	0.390500	0.200064	0.153000	0.081592
50	5	60	0.421000	0.215411	0.191000	0.102485
50	5	70	0.475000	0.243368	0.209000	0.114247
50	5	80	0.568000	0.291098	0.241000	0.129714
50	5	90	0.663000	0.340451	0.256000	0.146826
50	5	100	0.705000	0.360866	0.291000	0.163874

Table D.1: PARDISO and MA57 solvers, mass-spring problem 7.2, $n_x = 50$, $n_u = 5$.

			PARDISO		MA57	
n_x	n_u	N	CPU time	wall clock time	CPU time	wall clock time
4	1	10	0.000358	0.000183	0.000170	0.000192
8	1	10	0.000700	0.000373	0.000460	0.000461
16	1	10	0.006750	0.003466	0.001550	0.001572
32	1	10	0.021100	0.010771	0.013500	0.013742
64	1	10	0.128600	0.065823	0.046000	0.027568
128	1	10	0.586000	0.299835	0.202500	0.108911
256	1	10	3.916000	2.016855	0.979000	0.599491
512	1	10	33.930000	17.404348	7.730000	4.362493
4	2	10	0.000400	0.000204	0.000200	0.000206
8	2	10	0.000720	0.000372	0.000500	0.000494
16	2	10	0.001800	0.000972	0.001500	0.001500
32	2	10	0.131400	0.067259	0.010300	0.005326
64	2	10	0.207800	0.106349	0.042200	0.022207
128	2	10	0.581000	0.297627	0.205000	0.108307
256	2	10	5.178000	2.652331	1.017000	0.580696
512	2	10	27.420000	14.078512	7.690000	4.338559
8	4	10	0.000800	0.000421	0.000580	0.000655
16	4	10	0.018300	0.009364	0.001700	0.001675
32	4	10	0.026800	0.013769	0.011100	0.005835
64	4	10	0.147600	0.075676	0.044200	0.023166
128	4	10	0.607000	0.311040	0.205000	0.110650
256	4	10	4.350000	2.229680	1.020000	0.580832
512	4	10	38.320000	19.640594	7.640000	4.342200
16	8	10	0.008200	0.004212	0.002000	0.001973
32	8	10	0.021700	0.011161	0.012500	0.006544
64	8	10	0.129800	0.066668	0.047400	0.024536
128	8	10	0.690000	0.354867	0.212000	0.111890
256	8	10	5.343000	2.750471	1.021000	0.591158
512	8	10	31.790000	16.393541	7.700000	4.395613

Table D.2: PARDISO and MA57 solvers, mass-spring problem 7.2, $N = 10$.

D.2 Schur complement method

			Schur complement method	
n_x	n_u	N	CPU time	wall clock time
50	5	10	0.011100	0.005724
50	5	20	0.023400	0.012084
50	5	30	0.036200	0.018556
50	5	40	0.048800	0.025174
50	5	50	0.060800	0.031487
50	5	60	0.073000	0.038248
50	5	70	0.086000	0.045024
50	5	80	0.101000	0.053198
50	5	90	0.112000	0.057464
50	5	100	0.124000	0.064299

Table D.3: Schur complement method, mass-spring problem 7.2, $n_x = 50$, $n_u = 5$.

			$N = 10$		$N = 100$	
n_x	n_u	N	CPU time	wall clock time	CPU time	wall clock time
4	1	10	0.000134	0.000136	0.001300	0.001320
8	1	10	0.000230	0.000225	0.002200	0.002308
16	1	10	0.000540	0.000539	0.006400	0.006285
32	1	10	0.003750	0.001966	0.046500	0.023270
64	1	10	0.017500	0.009240	crashed	crashed
128	1	10	0.079100	0.040777	crashed	crashed
256	1	10	0.461400	0.238059	crashed	crashed
512	1	10	crashed	crashed	crashed	crashed
4	2	10	0.000144	0.000146	0.001400	0.001402
8	2	10	0.000260	0.000255	0.002500	0.002549
16	2	10	0.000580	0.000574	0.006800	0.006725
32	2	10	0.003950	0.002024	0.047500	0.025309
64	2	10	0.017800	0.009206	0.194000	0.099484
128	2	10	0.078400	0.040389	crashed	crashed
256	2	10	0.458400	0.236535	crashed	crashed
512	2	10	crashed	crashed	crashed	crashed
8	4	10	0.000270	0.000270	0.002800	0.002765
16	4	10	0.000600	0.000608	0.007200	0.007196
32	4	10	0.004050	0.002100	0.051000	0.026064
64	4	10	0.018400	0.009454	0.196000	0.103613
128	4	10	0.079500	0.040824	crashed	crashed
256	4	10	0.457400	0.234945	crashed	crashed
512	4	10	crashed	crashed	crashed	crashed
16	8	10	0.000720	0.000725	0.008600	0.008672
32	8	10	0.004650	0.002392	0.057000	0.029795
64	8	10	0.018900	0.009742	0.207000	0.106149
128	8	10	0.081800	0.042090	crashed	crashed
256	8	10	0.466800	0.239919	crashed	crashed
512	8	10	crashed	crashed	crashed	crashed

Table D.4: Schur complement method, mass-spring problem, $N = 10$ and $N = 100$.

D.3 Riccati recursion

			P s.p.d. upper		P s.p.d. lower	
n_x	n_u	N	CPU time	wall clock time	CPU time	wall clock time
50	5	10	0.002800	0.001513	0.002800	0.001448
50	5	20	0.005200	0.002663	0.005400	0.002697
50	5	30	0.007800	0.004038	0.007900	0.004029
50	5	40	0.010200	0.005279	0.010600	0.005404
50	5	50	0.012800	0.006602	0.013000	0.006627
50	5	60	0.015600	0.007995	0.015800	0.008050
50	5	70	0.017800	0.009292	0.018800	0.009444
50	5	80	0.020600	0.010539	0.020800	0.010561
50	5	90	0.023200	0.012006	0.023800	0.011946
50	5	100	0.025600	0.013174	0.026400	0.013749

Table D.5: Riccati recursion, mass-spring problem, $n_x = 50$, $n_u = 5$.

			P s.p.d. upper		P s.p.d. lower	
n_x	n_u	N	CPU time	wall clock time	CPU time	wall clock time
50	5	10	0.003900	0.002039	0.003900	0.002110
50	5	20	0.007600	0.003864	0.007800	0.004021
50	5	30	0.011200	0.005618	0.011600	0.005837
50	5	40	0.014900	0.007597	0.015600	0.007903
50	5	50	0.018600	0.009392	0.019200	0.009671
50	5	60	0.022600	0.011389	0.023200	0.011827
50	5	70	0.026200	0.013270	0.027000	0.013689
50	5	80	0.029600	0.015116	0.030600	0.015514
50	5	90	0.033600	0.016908	0.034600	0.017517
50	5	100	0.037400	0.018864	0.038800	0.019491

Table D.6: Riccati recursion, mass-spring problem, $n_x = 50$, $n_u = 5$.

			general algorithm		P s.p.s.d.	
n_x	n_u	N	CPU time	wall clock time	CPU time	wall clock time
4	1	10	0.000052	0.000052	0.000064	0.000065
8	1	10	0.000070	0.000075	0.000100	0.000095
16	1	10	0.000140	0.000138	0.000160	0.000164
32	1	10	0.000950	0.000495	0.001000	0.000553
64	1	10	0.003600	0.001855	0.003500	0.001782
128	1	10	0.023300	0.011888	0.021800	0.011037
256	1	10	0.268200	0.135285	0.220000	0.110954
512	1	10	2.245000	1.127479	2.248000	1.130302
4	2	10	0.000060	0.000060	0.000074	0.000074
8	2	10	0.000090	0.000090	0.000110	0.000107
16	2	10	0.000140	0.000156	0.000180	0.000180
32	2	10	0.000950	0.000501	0.001050	0.000554
64	2	10	0.003700	0.001863	0.003700	0.001894
128	2	10	0.023000	0.011702	0.023300	0.011803
256	2	10	0.272000	0.137731	0.230200	0.116378
512	2	10	2.268000	1.139122	2.266000	1.139136
8	4	10	0.000110	0.000114	0.000130	0.000132
16	4	10	0.000180	0.000189	0.000220	0.000214
32	4	10	0.001100	0.000566	0.001200	0.000631
64	4	10	0.003900	0.002000	0.004000	0.002048
128	4	10	0.025300	0.012864	0.023600	0.011940
256	4	10	0.273200	0.137822	0.227000	0.114639
512	4	10	2.313000	1.161446	2.308000	1.160436
16	8	10	0.000260	0.000264	0.000300	0.000301
32	8	10	0.001300	0.000679	0.001400	0.000726
64	8	10	0.004300	0.002208	0.004500	0.002362
128	8	10	0.025700	0.012994	0.027500	0.013959
256	8	10	0.289800	0.146181	0.240400	0.121188
512	8	10	2.310000	1.160322	2.307000	1.160716

Table D.7: Riccati recursions, mass-spring problem, $N = 10$. On the left the general algorithm, on the right the algorithm fully exploiting the symmetry of the P matrix.

			P s.p.d. upper		P s.p.d. lower	
n_x	n_u	N	CPU time	wall clock time	CPU time	wall clock time
4	1	10	0.000084	0.000085	0.000066	0.000066
8	1	10	0.000130	0.000124	0.000120	0.000116
16	1	10	0.000240	0.000246	0.000220	0.000232
32	1	10	0.001500	0.000769	0.001450	0.000786
64	1	10	0.005100	0.002576	0.005000	0.002536
128	1	10	0.023600	0.011994	0.024900	0.012637
256	1	10	0.177800	0.089638	0.179800	0.090851
512	1	10	1.584000	0.797587	1.380000	0.695365
4	2	10	0.000088	0.000089	0.000074	0.000074
8	2	10	0.000140	0.000142	0.000130	0.000137
16	2	10	0.000280	0.000279	0.000260	0.000258
32	2	10	0.001600	0.000810	0.001500	0.000771
64	2	10	0.005000	0.002617	0.005000	0.002593
128	2	10	0.024200	0.012265	0.024700	0.012476
256	2	10	0.165800	0.083604	0.181600	0.091560
512	2	10	1.550000	0.780625	1.372000	0.696298
8	4	10	0.000170	0.000173	0.000150	0.000155
16	4	10	0.000320	0.000310	0.000300	0.000298
32	4	10	0.001650	0.000850	0.001600	0.000819
64	4	10	0.005100	0.002641	0.005300	0.002688
128	4	10	0.025000	0.012639	0.025200	0.012730
256	4	10	0.170800	0.086180	0.178000	0.089771
512	4	10	1.551000	0.781073	1.385000	0.697820
16	8	10	0.000380	0.000381	0.000360	0.000363
32	8	10	0.001850	0.000958	0.001800	0.000922
64	8	10	0.005800	0.002928	0.005600	0.002922
128	8	10	0.025300	0.012791	0.026700	0.013477
256	8	10	0.177400	0.089575	0.185200	0.093365
512	8	10	1.567000	0.789231	1.412000	0.711657

Table D.8: Riccati recursions, mass-spring problem, $N = 10$. Algorithms assuming P to be symmetric positive definite: on the left the version computing the upper factor of the Cholesky factorization, on the right the one computing the lower factor.

D.4 Condensing methods

n_x	n_u	N	CPU time	wall clock time
50	5	10	0.002200	0.001222
50	5	20	0.007600	0.003944
50	5	30	0.017300	0.008919
50	5	40	0.029900	0.015441
50	5	50	0.046000	0.023729
50	5	60	0.065500	0.033853
50	5	70	0.088000	0.045289
50	5	80	0.115100	0.059343
50	5	90	0.145500	0.075036
50	5	100	0.180100	0.093559

Table D.9: Condensing methods, mass-spring problem, $n_x = 50$, $n_u = 5$.

		$N = 10$		$N = 100$	
n_x	n_u	CPU time	wall clock time	CPU time	wall clock time
4	1	0.000066	0.000066	0.003200	0.001722
8	1	0.000077	0.000077	0.004600	0.002358
16	1	0.000102	0.000102	0.009300	0.004741
32	1	0.000425	0.000215	0.019300	0.009745
64	1	0.000980	0.000505	0.049000	0.025186
128	1	0.003420	0.001732	0.144000	0.073656
256	1	0.018500	0.009309	0.760000	0.391285
512	1	0.172200	0.087287	4.310000	2.177815
4	2	0.000089	0.000089	0.007300	0.003760
8	2	0.000103	0.000104	0.010500	0.005348
16	2	0.000145	0.000144	0.018200	0.009194
32	2	0.000609	0.000306	0.035800	0.018106
64	2	0.001310	0.000670	0.091000	0.045707
128	2	0.004610	0.002335	0.272000	0.137352
256	2	0.030300	0.015256	1.280000	0.655852
512	2	0.216800	0.109843	6.600000	3.355560
8	4	0.000342	0.000173	0.026500	0.013388
16	4	0.000518	0.000260	0.041600	0.020983
32	4	0.000915	0.000464	0.073700	0.037381
64	4	0.002110	0.001080	0.177000	0.090450
128	4	0.006680	0.003370	0.541000	0.273740
256	4	0.043400	0.022339	2.690000	1.358476
512	4	0.276500	0.139311	11.570000	5.869583
16	8	0.001201	0.000605	0.145100	0.073313
32	8	0.001687	0.000848	0.209100	0.105733
64	8	0.004090	0.002064	0.417000	0.210739
128	8	0.012560	0.006331	1.282000	0.648978
256	8	0.066400	0.033912	6.400000	3.229201
512	8	0.379600	0.191434	24.970000	12.680102

Table D.10: Condensing methods, mass-spring problem, $N = 10$ and $N = 100$.

		$N = 10$		$N = 100$	
n_x	n_u	CPU time	wall clock time	CPU time	wall clock time
4	1	0.000058	0.000059	0.002600	0.001312
8	1	0.000067	0.000066	0.003700	0.001914
16	1	0.000083	0.000083	0.007200	0.003742
32	1	0.000326	0.000166	0.014800	0.007596
64	1	0.000680	0.000352	0.034000	0.017429
128	1	0.002180	0.001118	0.098000	0.050246
256	1	0.013700	0.007059	0.460000	0.246363
512	1	0.111700	0.057584	2.770000	1.421578
4	2	0.000077	0.000077	0.006400	0.003296
8	2	0.000089	0.000089	0.009400	0.005154
16	2	0.000119	0.000120	0.014700	0.007571
32	2	0.000474	0.000242	0.027500	0.014155
64	2	0.000980	0.000511	0.070000	0.042709
128	2	0.003250	0.001658	0.202000	0.103970
256	2	0.020900	0.010743	0.840000	0.431169
512	2	0.136500	0.070237	4.390000	2.257383
8	4	0.000305	0.000156	0.023500	0.012006
16	4	0.000434	0.000222	0.033600	0.017193
32	4	0.000738	0.000383	0.059200	0.030572
64	4	0.001590	0.000823	0.129000	0.065949
128	4	0.005260	0.002709	0.376000	0.193193
256	4	0.030900	0.015875	1.810000	0.935091
512	4	0.174500	0.090218	7.350000	3.799661
16	8	0.001072	0.000548	0.122000	0.062759
32	8	0.001506	0.000770	0.172300	0.088271
64	8	0.003130	0.001614	0.322000	0.165869
128	8	0.009400	0.004832	0.824000	0.426235
256	8	0.045300	0.023296	3.740000	1.916088
512	8	0.241800	0.124526	15.600000	8.065975

Table D.11: Condensing method as sub-routine in interior-point method, mass-spring problem, $N = 10$ and $N = 100$.

D.5 Comparison between C and MATLAB code

n_x	n_u	C CPU	C w.c.	ML CPU	ML w.c.	ML/C CPU	ML/C w.c.
10	1	0.000100	0.000113	0.001300	0.001213	13.000	10.735
20	1	0.000200	0.000225	0.002600	0.001477	13.000	6.564
40	1	0.001400	0.000783	0.004300	0.002176	3.071	2.779
60	1	0.003400	0.001793	0.006800	0.003437	2.000	1.917
80	1	0.006800	0.003493	0.011900	0.007808	1.750	2.235
100	1	0.011000	0.005584	0.019500	0.009815	1.773	1.758
140	1	0.033100	0.016730	0.046900	0.023638	1.417	1.413
200	1	0.110200	0.055630	0.142100	0.071696	1.289	1.289
10	2	0.000100	0.000129	0.001100	0.001056	11.000	8.186
20	2	0.000200	0.000290	0.001500	0.001291	7.500	4.452
40	2	0.001500	0.000806	0.004000	0.002038	2.667	2.529
60	2	0.003400	0.001744	0.006900	0.003445	2.029	1.975
80	2	0.006800	0.003484	0.011400	0.005775	1.676	1.658
100	2	0.011600	0.005876	0.019000	0.009618	1.638	1.637
140	2	0.032600	0.016646	0.048200	0.024406	1.479	1.466
200	2	0.110900	0.056114	0.143600	0.072566	1.295	1.293
10	5	0.000200	0.000206	0.001500	0.001520	7.500	7.379
20	5	0.000300	0.000428	0.002000	0.001788	6.667	4.178
40	5	0.002000	0.001080	0.005500	0.002783	2.750	2.577
60	5	0.004000	0.002033	0.008400	0.004252	2.100	2.091
80	5	0.007900	0.004041	0.013600	0.006867	1.722	1.699
100	5	0.013600	0.006924	0.021900	0.011048	1.610	1.596
140	5	0.037300	0.019215	0.051200	0.025948	1.373	1.350
200	5	0.117500	0.059659	0.149700	0.075572	1.274	1.267

Table D.12: Mass-spring problem 7.2, $N = 10$. 'w.c.' stands for wall clock (time).

n_x	n_u	C CPU	C w.c.	ML CPU	ML w.c.	ML/C CPU	ML/C w.c.
10	1	0.001000	0.000863	0.021000	0.014293	21.000	16.562
20	1	0.002000	0.002147	0.031000	0.016101	15.500	7.499
40	1	0.015000	0.007882	0.046000	0.024146	3.067	3.063
60	1	0.035000	0.018307	0.070000	0.035293	2.000	1.928
80	1	0.067000	0.034888	0.114000	0.058447	1.701	1.675
100	1	0.112000	0.057212	0.196000	0.116709	1.750	2.040
140	1	0.321000	0.162945	0.458000	0.232451	1.427	1.427
200	1	1.123000	0.566469	1.376000	0.695039	1.225	1.227
10	2	0.001000	0.001292	0.010000	0.009969	10.000	7.716
20	2	0.003000	0.002661	0.027000	0.014737	9.000	5.538
40	2	0.015000	0.008049	0.040000	0.020681	2.667	2.569
60	2	0.036000	0.018279	0.070000	0.035729	1.944	1.955
80	2	0.066000	0.033761	0.114000	0.058436	1.727	1.731
100	2	0.111000	0.056542	0.184000	0.094759	1.658	1.676
140	2	0.330000	0.168015	0.462000	0.238610	1.400	1.420
200	2	1.142000	0.577790	1.381000	0.697683	1.209	1.208
10	5	0.002000	0.002032	0.017000	0.015135	8.500	7.448
20	5	0.004000	0.003929	0.035000	0.019857	8.750	5.054
40	5	0.021000	0.010964	0.059000	0.031030	2.810	2.830
60	5	0.040000	0.021524	0.087000	0.045186	2.175	2.099
80	5	0.079000	0.040335	0.135000	0.069729	1.709	1.729
100	5	0.137000	0.069816	0.224000	0.112631	1.635	1.613
140	5	0.390000	0.197508	0.508000	0.256430	1.303	1.298
200	5	1.170000	0.592969	1.446000	0.729695	1.236	1.231

Table D.13: Mass-spring problem 7.2, $N = 100$. 'w.c.' stands for wall clock (time).

D.6 Interior-point methods

			basic IP		Mehrotra's IP	
n_x	n_u	N	CPU time	wall clock time	CPU time	wall clock time
4	1	10	0.000080	0.000080	0.000104	0.000106
8	1	10	0.000120	0.000114	0.000150	0.000144
16	1	10	0.000200	0.000210	0.000240	0.000239
32	1	10	0.001250	0.000713	0.001400	0.000749
64	1	10	0.004700	0.002499	0.005100	0.002733
128	1	10	0.030300	0.015598	0.034500	0.017712
256	1	10	0.298800	0.153783	0.314000	0.161909
512	1	10	2.432000	1.258858	2.479000	1.292235

Table D.14: Comparison of the cost per iteration of the algorithms 8 and 10.

Bibliography

[Duf04] I.S. Duff. Ma57 - a new code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software*, 30:118–144, 2004.

[FM03] E. Fornasini and G. Marchesini. *Appunti di teoria dei sistemi.* Libreria Progetto, 2003.

[Gou] B. Gough. *An Introduction to GCC.*

[GVDG] K. Goto and R.A. Van De Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software.*

[JGND12] J.B. Jørgensen, N.F. Gade-Nielsen, and B. Damman. Numerical methods for solution of the extended linear quadratic control problem. (unpublished), 2012.

[Jør05] J.B. Jørgensen. *Moving Horizon Estimation and Control.* PhD thesis, Department of Chemical Engineering, Technical University of Denmark, Kgs Lyngby, Denmark, 2005.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C programming Language.* Prentice-Hall, 1988.

[NW06] J. Nocedal and S.J. Wright. *Numerical Optimization.* Springer, Kgs Lyngby, Denmark, 2nd edition, 2006.

[Pag07] C.G. Page. *Professional Programmer's Guide to Fortran77*, 2007.

[RWR98] C.V. Rao, S.J. Wright, and J.B. Rawlings. Application of interior-point methods to model predictive control. *Journal of Optimization Theory and Applications*, 99(3):723–757, 1998.

[SG11] O. Schenk and K. Gärtner. *PARDISO, User Guide Version 4.1.2*, 2011.

[Sta] R.M. Stallman. *Using the GNU Compiler Collection.*

[WB10] Y. Wang and S. Boyd. Fast model predictive control using online optimization. *IEEE Transactions on Control Systems Technology*, 18(2):267–278, 2010.

[Wha] R. C. Whaley. *ATLAS Installation Guide.*