Solving equation systems associated with non-linear model predictive control

Comparison of two methods to solve equation systems which arise when solving optimization problems associated with non-linear model predictive control

Erik Ackzell

October 14, 2015

Abstract

When solving optimization problems associated with non-linear model predictive control, a linear equation system of a specific structure frequently arises. Two different software, CVXGEN and FORCES, developed to generate tailored solvers to specific optimization problems use two different methods when solving this equation system. In this paper, two different sets of software is presented which generate tailored solvers for this equation system using the two different methods. It is found that the solvers which implement the method used in FORCES are faster than the solvers which implement the method used in CVXGEN, as are the generations of the solvers using the FORCES method.

A third software which generate solvers to optimization problems is QPgen, which is now not able to generate solvers for non-linear model predictive control problems. The software presented in this paper is intended to be incorporated in the QPgen software in order to make it able to generate solvers for these problems as well.

0.1 Acknowledgements

I would like to thank my supervisor Pontus Giselsson for his encouraging and helpful comments throughout my work. I would also like to thank Claus Führer for always inspiring and guiding me through my studies. Lastly, I would like to thank my life partner Anna for all her wonderful help and support.

Contents

1 Introduction

In this work, the speed of two different methods of solving a linear system of equations which arise in non-linear model predictive control are compared. One of the methods appears in the CVXGEN software (Mattingley and Boyd, 2012) and the other method appears in the FORCES software (Domahidi et al., 2012). Software which generate tailored solvers to this equation system which implement the two different methods is presented. The software produced for this paper is intended to be incorporated in the QPgen software (Giselsson, 2015).

In Section 1.1, a short introduction to the problem formulation is given which is then expanded in Section 3.1. In Section 2, linear and a version of non-linear model predictive control is introduced and the Cholesky- and LDLT matrix factorization techniques are described. In Sections 3.2 and 3.3, the two methods are discussed in detail, the implementations of the methods in the software produced for this software are described and results on how the generated solvers performs are presented. In Section 4, the implementations and results are discussed and some conclusions are drawn, while Section 5 discusses what more work can be done.

1.1 Problem formulation and limitations

The linear equation which is solved in this paper is given by

$$Kv = w, \tag{1}$$

where K is a sparse matrix. Given the structure of K, the software created for this paper produces problem specific solvers written in C to solve (1) by exploiting the specific structure of K, but not the data, in order to create fast, efficient solvers.

The scope of this paper is limited to considering a structure of K which appears when solving optimization problems associated with model predictive control. This structure is described in Section 3.1.

1.2 CVXGEN, FORCES and QPgen

CVXGEN (Mattingley and Boyd, 2012), FORCES (Domahidi et al., 2012) and QPgen (Giselsson, 2015) are three software which all generate problem specific solvers to optimization problems, given certain parameters describing the problem.

Limited by size, the CVXGEN software generates solvers for general convex quadratic programming optimization problem, while the FORCES software is specifically developed to be used with model predictive control problems and can handle problems containing a bigger number of parameters than CVXGEN (Domahidi et al., 2012). The QPgen software generates solvers to convex optimization problems and while solvers generated by both CVXGEN and FORCES allows for all problem data to be updated in the finished solver (Mattingley and

Boyd (2012), Domahidi et al. (2012)), some data in the solvers generated by the QPgen software needs to be supplied in the generation stage (Giselsson, 2015). When using the QPgen software for model predictive control, it can only generate solvers for model predictive control for linear systems. The software produced for this paper is a step in making QPgen able to generate solvers for non-linear model predictive control.

2 Theory

2.1 Model Predictive Control

Model predictive control (MPC) is a method used in automatic control which requires a discrete time mathematical model of the process being controlled. At every time step a minimization problem is solved in order to calculate a control signal to be sent to the process. Early work on MPC includes Richalet et al. (1978) and Clarke (1987).

2.1.1 Linear MPC

Let $m, n \in \mathbb{N}^*$. At time step k, let $x_k \in \mathbb{R}^n$ be a vector describing some parameters of the process which is being controlled, $u_k \in \mathbb{R}^m$ be a vector of control signals which are sent to the process and let $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times m}$ be some matrices describing the dynamics of the process around some linearization point such that

$$x_k = Ax_{k-1} + Bu_{k-1}. \tag{2}$$

Furthermore, let some state- and control signal reference at time step k be given by $x_k^{\text{ref}} \in \mathbb{R}^n$ and $u_k^{\text{ref}} \in \mathbb{R}^m$, respectively, and let $Q \in \mathbb{R}^{n \times n}$ and $R \in \mathbb{R}^{m \times m}$ be symmetric positive definite weighing matrices. Set $\Delta x_k = x_k - x_k^{\text{ref}}$ and $\Delta u_k = u_k - u_k^{\text{ref}}$. Given some *prediction horizon* $N \in \mathbb{N}^*$, an initial state x_0, some limits $l_x, l_u, u_x, u_u \in \mathbb{R}$ and real, full-rank matrices $C_x \in \mathbb{R}^{n \times n}$ and $C_u \in \mathbb{R}^{m \times m}$, the minimization problem to be solved at time step $k = 0$ is given by

$$\min_{x_i, u_{i-1}, i=1,\ldots,N} \sum_{k=1}^{N} \|\Delta x_k\|_Q + \|\Delta u_{k-1}\|_R, \tag{3}$$

subject to (2) and

$$\begin{aligned} l_x &\leq C_x x_k \leq u_x \\ l_u &\leq C_u u_{k-1} \leq u_u, \end{aligned} \tag{4}$$

for $k = 1, 2, \ldots, N$. After solving (3) subject to constraints (2) and (4) at the current time step, the first control signal u_0 is sent to the process and the minimization problem is then solved again at the next time step.

2.1.2 Non-linear MPC

One version of non-linear MPC is described in this section. Instead of linearizing the dynamics of the process around a specific operating point, they can instead be linearized around some trajectory of operating points such that A and B are allowed to vary with time. If Q and R are also allowed to vary with time, the minimization problem (3) subject to (2) and (4) can be replaced with

$$\min_{x_i, u_{i-1}, i=1,\ldots,N} \sum_{k=1}^{N} \|\Delta x_k\|_{Q_k} + \|\Delta u_{k-1}\|_{R_{k-1}}, \tag{5}$$

subject to

$$\begin{aligned} x_k &= A_{k-1}x_{k-1} + B_{k-1}u_{k-1} \\ l_x &\leq C_x x_k \leq u_x \\ l_u &\leq C_u u_{k-1} \leq u_u. \end{aligned} \tag{6}$$

2.2 Cholesky factorization

The Cholesky factor of a matrix M is a lower triangular matrix L such that $LL^{\mathrm{T}} = M$. By Theorem 6.3.1 in Allaire and Kaber (2008), there exists a unique Cholesky factor for any real symmetric positive definite matrix M. An algorithm to obtain a Cholesky factor $L \in \mathbb{R}^{N \times N}$ of a matrix $M \in \mathbb{R}^{N \times N}$ can be observed in Algorithm 1.

Algorithm 1 Cholesky factorization

```
Initialize L=0 and set L_{1,1} = sqrt(M_{1,1})
for i = 2, 3, ..., N do
    for j = i, i+1, ..., N do
        L_{j,i-1} = M_{j,i-1}/L_{i-1,i-1}
    end for
    for j = 1, 2, ..., i do
        for k = i, i+1, ..., N do
            M_{k,i} = M_{k,i} - L_{k,j}L_{i,j}
        end for
    end for
    L_{i,i} = sqrt(M_{i,i})
end for
```

2.3 LDLT factorization

If M is a square matrix and the pair (L, D) is a unit lower triangular matrix and a diagonal matrix, respectively, such that $M = LDL^{\mathrm{T}}$, the pair (L, D) is denoted the LDLT factors of M.

Definition 2.3.1. *Let E, F be positive definite matrices and let G be any matrix and define*

$$M = \begin{bmatrix} -E & G^{\mathrm{T}} \\ G & F \end{bmatrix}. \tag{7}$$

Then M is quasi-definite.

By Theorem 2 in Vanderbei (1995), any symmetric quasi-definite matrix M is *strongly factorizable*. This means that regardless of the choice of permutation matrix P, there exists a unique pair of LDLT factors of PMP^{T}. An algorithm to obtain LDLT factors $L, D \in \mathbb{R}^{N \times N}$ of a matrix $M \in \mathbb{R}^{N \times N}$ is shown in Algorithm 2. In this algorithm, the strictly lower triangle of the original matrix M is overwritten by the strictly lower triangle of L, while the diagonal of M is overwritten by the diagonal elements in D.

Algorithm 2 LDLT factorization

```
for i = 1, 2, ..., N do
    for j = 1, 2, ..., i-1 such that M_{i,j} ≠ 0 do
        for k = i+1, i+2, ..., N such that M_{k,j} ≠ 0 do
            M_{k,i} = M_{k,i} - M_{k,j} M_{j,j} M_{i,j}
        end for
    end for
    for j = i+1, i+2, ..., N such that M_{j,i} ≠ 0 do
        M_{j,i} = M_{j,i} / M_{i,i}
        M_{j,j} = M_{j,j} - M_{j,j} M_{j,i}^2
    end for
end for
```

3 Analysis

3.1 Problem to be solved

This section uses the same naming as in Section 2.1.2. Since Q_i, R_{i-1} are all symmetric, Equation (5) can be written as

$$\min_{x_i, u_{i-1}, i=1,\ldots,N} \sum_{i=1}^{N} \|x_k\|_{Q_k} + \|u_{k-1}\|_{R_{k-1}} + q_k^x x_k + q_{k-1}^u u_{k-1}, \tag{8}$$

where

$$\begin{aligned} q_k^x &= -2(x_k^{\text{ref}})^{\mathrm{T}} Q_k \\ q_{k-1}^u &= -2(u_{k-1}^{\text{ref}})^{\mathrm{T}} R_{k-1}. \end{aligned}$$

Equation (8) is then equivalent to

$$\min z^{\mathrm{T}} H z + q z, \tag{9}$$

where

$$\begin{aligned} z &= [x_1, x_2, \ldots, x_N, u_0, u_1, \ldots, u_{N-1}] \\ q &= [q_1^x, q_2^x, \ldots, q_N^x, q_0^u, q_1^u, \ldots, q_{N-1}^u] \\ H &= \text{blkdiag}(Q_1, Q_2, \ldots, Q_N, R_0, R_1, \ldots, R_{N-1}). \end{aligned}$$

By setting

$$\begin{aligned} b &= [-A_0 x_0, 0, 0, \ldots, 0] \\ C &= \text{blkdiag}(C_x, C_x, \ldots, C_x, C_u, C_u, \ldots, C_u) \\ l &= [l_x, l_x, \ldots, l_x, l_u, l_u, \ldots, l_u] \\ u &= [u_x, u_x, \ldots, u_x, u_u, u_u, \ldots, u_u] \end{aligned}$$

$$A = \left[\begin{array}{ccccc|cccc} -I & & & & & B_0 & & & \\ A_1 & -I & & & & & B_1 & & \\ & A_2 & -I & & & & & B_2 & \\ & & \ddots & \ddots & & & & & \ddots \\ & & & A_{N-1} & -I & & & & B_{N-1} \end{array}\right],$$

solving the minimization problem (5) subject to the constraints (6) is equivalent to solving (9) subject to

$$\begin{aligned} Az &= b \\ l \leq Cz &\leq u. \end{aligned} \tag{10}$$

Let μ be some unknown vector of appropriate dimension, let λ, y be given vectors of appropriate dimensions and let $\rho \in \mathbb{R}$ be some given parameter. When using

any of the methods used in QPgen for solving (9) subject to (10), a system of linear equations which need to be solved is given by

$$\underbrace{\begin{bmatrix} \bar{H} & A^{\mathrm{T}} \\ A & 0 \end{bmatrix}}_{K} \underbrace{\begin{bmatrix} z \\ \mu \end{bmatrix}}_{v} = \underbrace{\begin{bmatrix} -\bar{q} \\ b \end{bmatrix}}_{w}, \tag{11}$$

where

$$\begin{aligned} \bar{H} &= H + \rho C^{\mathrm{T}} C \\ \bar{q} &= q + \lambda^{\mathrm{T}} C - \rho C y. \end{aligned} \tag{12}$$

Hence, the problem which is considered in this paper is Equation (11).

3.2 Method used in CVXGEN

3.2.1 Description

In the solvers generated by the CVXGEN software, the K matrix in Equation (11) is modified and LDLT factorized and this new system of linear equations is then solved using forward- and backward substitution. After this, the solution to the original system is determined in an iterative process. In Mattingley and Boyd (2012) it is assumed that the structure of the L factor in the LDLT factorization of a matrix A is dependent only on the structure of A and not on the data. This is exploited in the generated solver, since the structure of K is known in the generation state.

In the generation state of the CVXGEN software, a permutation matrix P is obtained such that the number of nonzero elements of the L factor in the LDLT factorization of the symmetrically permuted modified K matrix is small. The method used to determine this permutation is the *local minimum fill-in* method introduced in Duff et al. (1986) and is described in Algorithm 3.

As mentioned in Section 2.3, an LDLT factorization of a matrix exists if the matrix is quasi-definite. Since the K matrix is not quasi-definite, K is modified in the following way

$$\tilde{K} = K + \begin{bmatrix} \epsilon I & 0 \\ 0 & -\epsilon I \end{bmatrix} \tag{13}$$

for some $\epsilon > 0$. This new matrix $\tilde{K}$ is quasi-definite (Mattingley and Boyd, 2012). After solving the modified equation

$$\tilde{K}\tilde{v} = y, \tag{14}$$

the solution to the original system in Equation (11) can be obtained through *iterative refinement* (Mattingley and Boyd, 2012), a process described in Algorithm 4. The iterative refinement of the solution will converge to the solution of the original system (Mattingley and Boyd, 2012) when the $\tilde{K}$ is chosen in this way.

Algorithm 3 Calculate permutation

```
1: Initialize Kfinal = K
2: for i = 1, 2, ..., N-1 do
3:     Set K = Kfinal
4:     Set f = N^2
5:     for j = i+1, i+2, ..., N do
6:         Permute rows i and j in K
7:         Permute columns i and j in K
8:         Factorize K=LDL^T
9:         Set nnz = number of non-zero elements in L
10:        if nnz ≤ f then
11:            f = nnz
12:            p = j
13:        end if
14:    end for
15:    Save p for later use
16:    Permute rows i and p in Kfinal
17:    Permute columns i and p in Kfinal
18: end for
```

Algorithm 4 Iterative refinement

```
1: Set v^(0) = ṽ
2: Set maxit to be the maximum number of iterations
3: for i = 1, 2, ..., maxit do
4:     Obtain δv by solving K̃(δv) = w − Kv^(i−1)
5:     Update v^(i) = v^(i−1) + δv
6:     if ‖Kv^(i) − w‖ ≤ res then
7:         break
8:     end if
9: end for
```

3.2.2 Implementation in this paper

The programs which implement the method used in the CVXGEN software consists of a combination of C- and Python scripts. In the Python scripts, given the dimensions and structures of the matrices described in Section 2.1.2 a quasi-definite matrix $\tilde{K}$ with arbitrary data is set up using the NumPy library. A C-script which calculates the permutation P is then called from the Python script using the ctypes library. When this is finished, another Python script is called which generates the solver files, using the calculated permutations as well as the structure of the K matrix.

In the solver file, two different structs are introduced, the *square_mat*- and the *data* struct. The square_mat struct is used to store full, square matrices and has two members, an integer for the dimension and a pointer to the first element in the double array in which the data is stored. The data struct contains all data which is used by the solver, among which the K and $\tilde{K}$ matrices are kept as square_mats and the diagonal of the D matrix and all vectors are kept as pointers to the first elements in the array in which they are stored. The data struct is initialized on the first use of the script.

Most of the functions in the solver exploits the structure of the matrices involved. For example, matrix-vector multiplication, LDLT factorization, forward- and backward substitution and the function which performs the permutations are all loop free and hard coded in such a way that multiplication by zeros or permutation of two zeros are omitted when appropriate.

3.2.3 Results

In Figures 1 and 2, factorization-, solve- and generation times and residuals are shown for two different generated solvers. The results shown in Figure 1 were obtained when the $A_i, B_{i-1}, Q_i, R_{i-1}, C_x$ and C_u matrices introduced in Section 2.1.2 were all full. In Figure 2 however, the A_i and B_{i-1} matrices were only about half full and the Q_i, R_{i-1}, C_x and C_u were all diagonal. In all tests $A_i, Q_i, C_x \in \mathbb{R}^{4\times 4}$, $B_{i-1} \in \mathbb{R}^{4\times 2}$ and $R_{i-1}, C_u \in \mathbb{R}^{2\times 2}$, while the prediction horizon N varies with $N = 5, 6, \ldots, 50$. In the generation state, the ϵ in Equation (13) was chosen as $\epsilon = 1.0$, while it was chosen as $\epsilon = 10^{-6}$ in the tests of the generated solvers.

For every choice of N, 50 000 tests were run and the average factorization- and solve times as well as average residuals are shown, while the solver generation times shown are results of one test per choice of N. The average number of iterations in the solution refinement step was two for every choice of N. All tests were conducted on a Windows laptop with a 1.80GHz 64-bit dual core processor and 8Gb RAM.

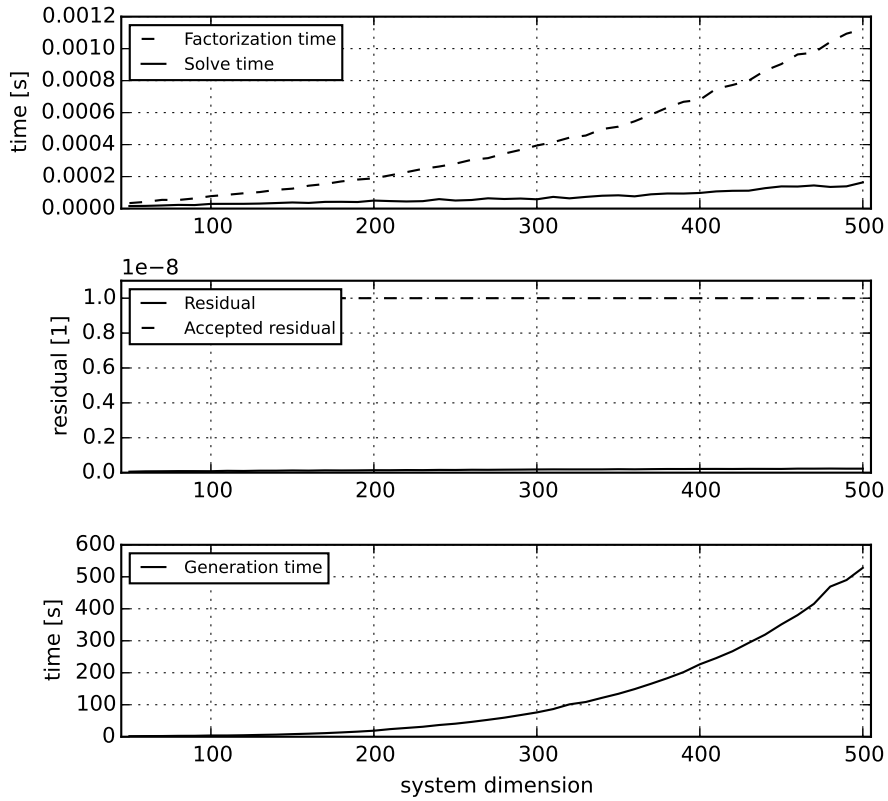

Figure 1: In this figure, the results when using the CVXGEN method for solving Equation (11) are shown. In the upper plot, the factorization- and solve times are shown with dashed and solid lines, respectively. In the middle plot, the accepted residual and the actual residuals, both measured in the two norm, are shown with dashed and solid lines, respectively, while the last plot shows the time taken to generate the solver. The first two plots shows average results when 50 000 tests were run, while the last plot only shows the results from one test. In these tests, using the same naming as in Section 3.1, the $A_i, B_{i-1}, Q_i, R_{i-1}, C_x$ and C_u matrices are full. As can be seen, the factorization times grows much more rapidly than the solve times. Furthermore, the actual residual is more than an order of magnitude smaller than what is accepted and the generation time grows exponentially with the size of the problem.

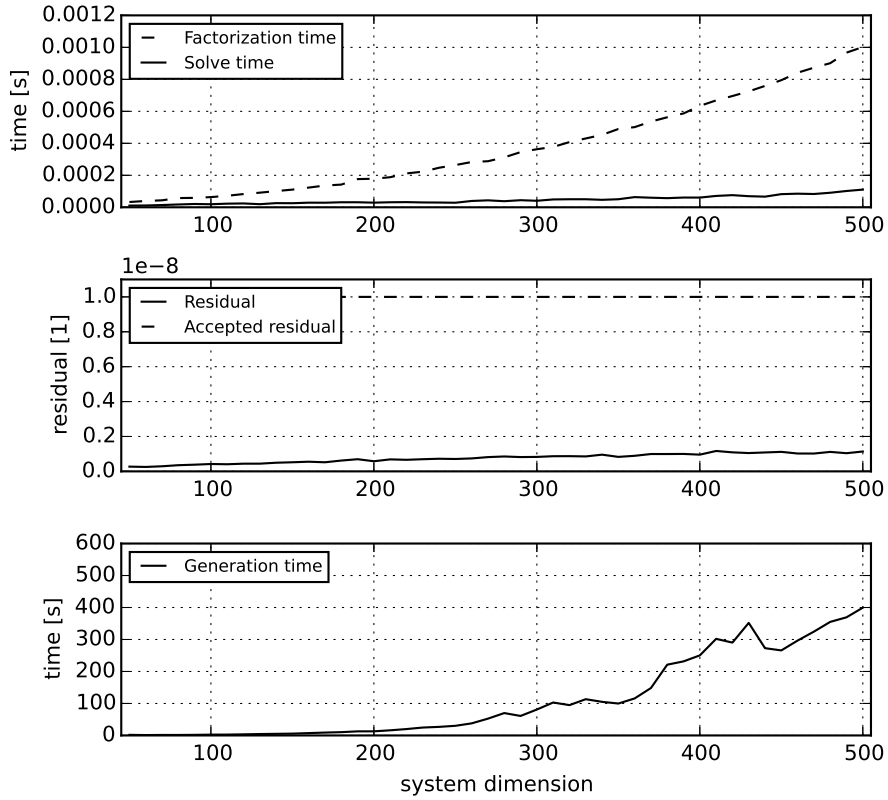

Figure 2: In this figure, the same number of tests are run and the same method is used as in Figure 1. However, this figure shows the result when using only half full matrices A_i, B_{i-1} and diagonal matrices Q_i, R_{i-1}, C_x and C_u. Comparing with Figure 1, both the factorization- and solve times are reduced slightly as is the generation time for almost all dimensions. However, the actual residual is slightly larger than in Figure 1.

3.3 Method used in FORCES

3.3.1 Description

In this section, the method used in Domahidi et al. (2012) to solve Equation (11) is described. The method exploits the banded structure of K. Using the same naming as in Section 2.1.2, for $i = 1, 2, \ldots, N$, set

$$\begin{aligned}
\hat{Q}_i &= Q_i + \rho C_x^{\mathrm{T}} C_x \\
\hat{R}_{i-1} &= R_{i-1} + \rho C_u^{\mathrm{T}} C_u \\
Y &= A \bar{H}^{-1} A^{\mathrm{T}} \\
&= \begin{bmatrix} Y_{1,1} & Y_{1,2} & & & \\ Y_{1,2}^{\mathrm{T}} & Y_{2,2} & \ddots & & \\ & \ddots & \ddots & & \\ & & & & Y_{N-1,N} \\ & & & Y_{N-1,N}^{\mathrm{T}} & Y_{N,N} \end{bmatrix},
\end{aligned}$$

where

$$\begin{cases}
Y_{1,1} & = \hat{Q}_1^{-1} + B_0 \hat{R}_0^{-1} B_0^{\mathrm{T}} \\
Y_{i,i+1} & = -\hat{Q}_i^{-1} A_i^{\mathrm{T}} \quad i = 1, 2, \ldots, N-1 \\
Y_{i,i} & = \hat{Q}_i^{-1} + B_{i-1} \hat{R}_{i-1}^{-1} B_{i-1}^{\mathrm{T}} + A_{i-1} \hat{Q}_{i-1}^{-1} A_{i-1}^{\mathrm{T}} \quad i = 2, 3, \ldots, N.
\end{cases}$$

Solving Equation (11) is then equivalent to solving

$$Y\mu = \beta \tag{15a}$$

$$\beta = -b - A\bar{H}^{-1}\bar{q} \tag{15b}$$

$$\bar{H}z = -\bar{q} - A^{\mathrm{T}}\mu. \tag{15c}$$

First determine the Cholesky factors of $\hat{Q}_i$ and $\hat{R}_{i-1}$ for $i = 1, 2, \ldots, N$,

$$\begin{aligned}
\hat{Q}_i &= L_{\hat{Q}_i} L_{\hat{Q}_i}^{\mathrm{T}} \\
\hat{R}_{i-1} &= L_{\hat{R}_{i-1}} L_{R_{i-1}}^{\mathrm{T}}.
\end{aligned}$$

Then solve

$$\begin{aligned}
V_i L_{Q_i}^{\mathrm{T}} &= I \\
W_{i-1} L_{R_{i-1}}^{\mathrm{T}} &= B_{i-1}
\end{aligned}$$

and set up

$$\begin{cases} Y_{1,1} & = V_1 V_1^{\mathrm{T}} + W_0 W_0^{\mathrm{T}} \\ Y_{i,i+1} & = -V_i V_i^{\mathrm{T}} A_i^{\mathrm{T}} \quad i = 1, 2, \ldots, N-1 \\ Y_{i,i} & = V_i V_i^{\mathrm{T}} + W_{i-1} W_{i-1}^{\mathrm{T}} + A_{i-1} V_{i-1} V_{i-1}^{\mathrm{T}} A_{i-1}^{\mathrm{T}} \quad i = 2, 3, \ldots, N. \end{cases}$$

Each block $Y_{i,j}$ is then factorized

$$\begin{cases} Y_{1,1} & = L_{1,1} L_{1,1}^{\mathrm{T}} & \\ Y_{i,i+1} & = L_{i,i} L_{i+1,i}^{\mathrm{T}} & i = 1, 2, \ldots, N \\ Y_{i+1,i+1} - L_{i+1,i} L_{i+1,i}^{\mathrm{T}} & = L_{i+1,i+1} L_{i+1,i+1}^{\mathrm{T}} & i = 1, 2, \ldots, N, \end{cases}$$

where $L_{i,i}$ is lower triangular and $L_{i,i+1}$ might be full. When this is done and β is set up, Equations (15a) and (15c) are solved using backward- and forward substitution.

3.3.2 Implementation in this paper

Just as with the implementation of the CVXGEN method, the software produced for this paper which implements the FORCES method consists of both C- and Python scripts. All needed dimensions are passed to a Python function, which then generates the solver which consists of one header file and one main file. In a similar way as in the implementation of the CVXGEN method, different structs are introduced to store matrices as well as all data needed to solve the problem.

In the generated solvers, only the blocks of the block- and tri-block diagonal matrices are stored and the lower triagonal matrices are stored as full matrices. The block- and tri-block diagonal structure is then exploited in the backward- and forward substitution steps.

3.3.3 Results

In Figure 3 results when testing generated solvers using the FORCES method are shown, along with solver generation times. With the same naming as in Section 2.1.2, the $A_i \in \mathbb{R}^{4\times 4}$ and $B_{i-1} \in \mathbb{R}^{4\times 2}$ matrices used are both full, as are the $\hat{Q}_i \in \mathbb{R}^{4\times 4}$ and $\hat{R}_{i-1} \in \mathbb{R}^{2\times 2}$ matrices mentioned in Section 3.3.1. For every choice of prediction horizon $N = 5, 6, \ldots, 50$, the average factorization times, solve times and residuals of 50 000 tests are shown. For every choice of N, one solver generation time was measured. All tests were conducted on a Windows laptop with a 1.80GHz 64-bit dual core processor and 8Gb RAM.

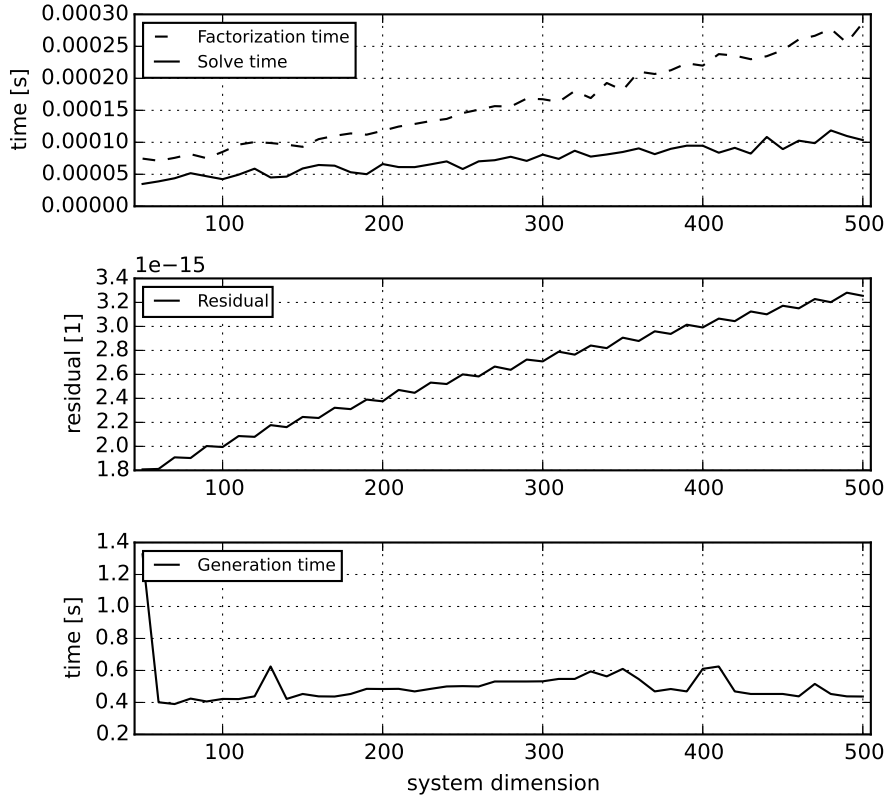

Figure 3: In this figure, the results obtained when using the FORCES method are shown. In the first plot, factorization- and solve times are shown with dashed and solid lines, respectively. With the same naming as in Section 2.1.2, the A_i and B_{i-1} matrices are full, as are the $\hat{Q}_i$ and $\hat{R}_{i-1}$ matrices introduced in Section 3.3. The first two plots shows average results when 50 000 test were conducted, while the last plot only shows results from one test. It can be seen that the factorization times increase more rapidly than the solve times as the size of the problem is increased. The residuals also increase with the size, while the generation times remain around 0.5s apart from the first generated solver.

4 Concluding discussion

4.1 Discussion of limitations

Since the solver generation time is quite long for big systems when producing solvers which use the CVXGEN method, only one generation time per prediction horizon was measured. Even though this might be enough to see that the these solvers take much more time to generate than the solvers using the FORCES method, the tests might not be enough to give an accurate estimate of how long time code generation actually takes. Since a goal of this paper is to compare the methods in an equal way, only one generation time per prediction horizon was measured for generation of the solvers using the FORCES method as well. This might indicate that the large difference in generation time for the solver for the smallest problem compared to all other problem sizes, as seen in Figure 3, is only a random occurrence.

When using solvers which use the FORCES method, no accepted residual can be specified. Since the tests of the solvers which use the CVXGEN method were all conducted with an accepted residual of 10^{-8}, the comparison of the efficiency of the two methods is only relevant for this specific choice of residual.

The relatively large choice of $\epsilon = 1.0$ in Equation (13) in the code generation step of the implementation of the CVXGEN method was made in order to minimize errors due to numerical instability. For this specific choice of ϵ, all LDLT factorizations in the generation step succeeded, while a smaller choice sometimes resulted in failed factorizations in the generation step. In all testing, all solvers of both methods always succeeded in solving the system of linear equations.

4.2 The produced software

The two considered methods are implemented in two different sets of software. The structures of these are quite similar but differ in some aspects. Different from the solvers implementing the CVXGEN method, the solvers implementing the FORCES does not consider the structure of the individual matrices $A_i, B_{i-1}, \hat{Q}_i$ and $\hat{R}_{i-1}$ apart from what dimensions they have. If for example the $\hat{Q}_i$ and $\hat{R}_{i-1}$ are diagonal, exploiting this might speed up the generated solvers. Furthermore, the solvers produced by the software implementing the CVXGEN method are free from for loops and are quite difficult to read, while the solvers generated using the FORCES method are far more readable.

4.3 Comparing the two methods

4.3.1 Factorization- and solve times

When comparing factorization times of the two methods, the FORCES method is much faster than the CVXGEN method. Since the FORCES software is specialized to solving only MPC problems, this is expected. However, the difference in solve time is much smaller, even though the FORCES method is slightly faster even here. In Figures 1 and 2 it can be observed that lower denseness of the individual matrices which made up the $\tilde{K}$ matrix in Equation (14) resulted in faster factorization times as well as faster solve times when using the CVXGEN method.

4.3.2 Code generation times

The local minimum fill-in method used in Mattingley and Boyd (2012) is very time consuming, while the solver generation time using the FORCES method is very fast. However, since the generation state is only done once, this is not necessarily an issue and might be negligible cost for a fast and efficient solver. Just as with factorization- and solve times, the generation time of the solvers which implemented the CVXGEN method was reduced as the denseness of the individual matrices making up the $\tilde{K}$ matrix was decreased.

5 Further research

When generating the solvers which use the CVXGEN method, the level of fill-in in the L matrix LDLT factorization of the $\tilde{K}$ matrix seemed to be related to the structure of the individual matrices which make up the matrix, and not only their denseness. This might be interesting to study further.

There are still several things in the software produced for this paper which can be improved, including making the generated solvers which use the FORCES method able to exploit the structures of the individual matrices which make up the system of linear equations to be solved. Furthermore, the way that matrices are stored in the solvers generated using the CVXGEN software can be made more efficient. Small modifications must be made to make the software produced for this paper compatible with the QPgen software.

A method presented in Patrinos and Bemporad (2014) to solve the equation system might also be of interest to compare with.

6 Notations

$S^* = S \backslash \{0\}$

$\|v\|_M = v^{\mathrm{T}} M v$

$$\text{If } v \in \mathbb{R}^m \text{ then } a \leq v \leq b \quad \Leftrightarrow \begin{cases} a & \leq v_i & \leq b & \forall i & \text{if} & a, b \in \mathbb{R} \\ a_i & \leq v_i & \leq b_i & \forall i & \text{if} & a, b \in \mathbb{R}^m \end{cases}$$

References

G. Allaire and S. M. Kaber. *Numerical linear algebra [Electronic resource].* Texts in applied mathematics: 55. New York, NY : Springer, c2008., 2008. ISBN 9780387689180.

D. Clarke. Generalized predictive control Part I. – The basic algorithm. *Automatica*, 23(2):137–148, 1987.

A. Domahidi, A. U. Zgraggen, M. N. Zeilinger, M. Morari, and C. N. Jones. Efficient interior point methods for multistage problems arising in receding horizon control. *51st IEEE Conference on Decision & Control (CDC)*, 2012. ISSN 9781467320658.

I. S. Duff, A. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Monographs on numerical analysis. Oxford: Clarendon, 1986. ISBN 0198534086.

P. Giselsson. QPgen, 2015.
http://www.control.lth.se/user/pontus.giselsson/qpgen/index.html.

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

P. Patrinos and A. Bemporad. An accelerated dual gradient-projection algorithm for embedded linear model predictive control. *IEEE Transactions on Automatic Control*, 59(1):18 – 33, 2014. ISSN 00189286.

J. Richalet, A. Rault, J.L Testud, and J. Papon. Model predictive heuristic control: applications to industrial processes. *Automatica*, 14(5):413–428, 1978.

R. J. Vanderbei. Symmetric quasi-definite matrices. *SIAM Journal on Optimization*, 5(1):100–113, 1995.