Noname manuscript No.
(will be inserted by the editor)

qpBAMM: a parallelizable ADMM approach for block-structured quadratic programs

Michel Lahmann · Christian Kirches · Martin T. Köhler

Received: date / Accepted: date

Abstract Block-structured quadratic programs (QPs) frequently arise in the context of the direct approach to solving optimal control problems. For successful application of direct optimal control algorithms to many real-world problems it is paramount that these QPs can be solved efficiently and reliably. Besides interior-point methods and active-set methods, ADMM-based quadratic programming approaches have gained popularity. Existing ADMM-based QP solvers typically split the problem into an equality-constrained QP and a projection problem onto, for example, a box-shaped feasible region. While this is a proven and widely used method, splitting the optimal control QP in this way does not easily allow to utilize and exploit the block structure of optimal control QPs to the fullest extent. In this paper, we address this situation by splitting the QP into a box-constrained QP, and a projection onto the linearized dynamics. Restricting the QP constraint set to a box, the solution becomes possible through an embarrassingly parallel gradient projection method. The projection step can be parallelized only partially, but benefits from the invariance of the constraint set to be projected on. With the proposed splitting, we are hence able to utilize the optimal control block structure, and solve relevant parts of the problem in parallel. We offer an implementation in Julia, present numerical results on random problems as well as on a popular benchmark problem, and demonstrate significant advantages over four recent Augmented Lagrangian method based solvers when applied to instances of optimal control problems with many state variables.

Keywords Quadratic programming · parallelizable computation · Optimal Control · ADMM

M. Lahmann
Institute for Mathematical Optimization, Technische Universität Carolo-Wilhelmina zu Braunschweig, Universitätsplatz 2, 38106 Braunschweig, GERMANY
E-mail: m.lahmann@tu-braunschweig.de

C. Kirches · M.T. Köhler
Institute for Mathematical Optimization, Technische Universität Carolo-Wilhelmina zu Braunschweig, Universitätsplatz 2, 38106 Braunschweig, GERMANY

Mathematics Subject Classification (2020) 90C20 · 90C25 · 90C06 · 65K05 · 93B45

1 Introduction

1.1 QP Formulation

Convex quadratic programs (QPs) are generally one of the most important problem classes of mathematical optimization problems, with applications in fields such as control engineering, finance, or statistics. They also arise as subproblems in nonlinear programming methods such as Sequential Quadratic Programming. We consider quadratic programming problems of the form

$$\min_{x\in\mathbb{R}^n} \quad \tfrac{1}{2}x^T Hx + h^T x \tag{1a}$$
$$\text{s.t.} \quad Cx = c, \tag{1b}$$
$$\underline{x} \leq x \leq \overline{x} \tag{1c}$$

with a symmetric and positive definite Hessian matrix $H \in \mathbb{R}^{nxn}$, a linear objective $h \in \mathbb{R}^n$, a constraint matrix $C \in \mathbb{R}^{mxn}$ with full row rank, a constraint vector $c \in \mathbb{R}^m$, and with lower and upper bounds vectors $\underline{x}$, $\overline{x} \in \mathbb{R}^n \cup \{-\infty, \infty\}^n$ on the unknowns $x \in \mathbb{R}^n$. Note that this is a standard formulation of a QP from [30], and that every QP can be easily transformed into this form.

Since the Hessian matrix H is assumed to be positive definite throughout, problem (1) is convex and the necessary optimality conditions of first order are also sufficient:

Theorem 1 (Necessary and Sufficient Optimality Conditions) *A vector $x \in \mathbb{R}^n$ is the unique minimizer of problem* (1) *if and only if there are vectors $y \in \mathbb{R}^m$, $z \in \mathbb{R}^n$ such that the following holds:*

$$0 = Hx + h + C^T y + z, \tag{2a}$$
$$0 = Cx - c, \tag{2b}$$
$$\underline{x} \leq x \leq \overline{x} \tag{2c}$$
$$z_i \geq 0 \textit{ for all } i \in [n] \textit{ with } x_i = \overline{x}_i, \tag{2d}$$
$$z_i \leq 0 \textit{ for all } i \in [n] \textit{ with } x_i = \underline{x}_i, \tag{2e}$$
$$z_i = 0 \textit{ for all } i \in [n] \textit{ with } \underline{x}_i < x_i < \overline{x}_i. \tag{2f}$$

1.2 Solution methods

Convex quadratic programming has been studied as an application of the primal simplex method [15] in the mid-1950s, and a variety of solution approaches has emerged over the following decades. A rough classification might categorize them as active-set methods, first-order methods, interior point methods, semismooth Newton-type methods and Augmented Lagrangian methods.

- The first approach to be mentioned are interior point methods, which first became known for solving linear programs (LPs) in polynomial time [24]. Interior point methods do not work with the active set but operate within the feasible set. To do this, the constraints are drawn into the objective function with a penalty parameter. Subsequently, a series of the resulting unconstrained problems are solved iteratively with adapting penalty parameters until a sufficiently good solution is found. Examples for implementations of interior point methods include the well-known NLP solver IPOPT [39] and the open source solver OOQP [20].

- Another widely used approach are active set methods. In this approach, the active set is estimated first and the problem is solved with the proposed active constraints. The active set is then iteratively adjusted and the corresponding problems are solved until a satisfactory solution is achieved. In addition to implementations of the active set method in commercial solvers such as MOSEK and GUROBI, this method can also be found in open source software such as qpOASES [13].

- First-order methods only consider first order information of the cost function when calculating an optimal solution. This leads to inexpensive and simple steps and these methods are therefore well suited for applications with limited computing resources. The disadvantage of these methods is that many iterations are usually required until an accurate solution is found. In addition, these methods are strongly influenced by ill-conditioning of the problem data. To address this problem, however, there are several approaches to improve the behaviour of the first order methods, e.g. [35,37]. The well-known solver OSQP [36] relies on a first-order method.

- Semismooth Newton methods represent a further approach. This approach is based on applying a nonsmooth version of the Newton method to the KKT conditions of the original problem [33]. In the strictly convex case with a nonsingular linear system, this method performs very well. If this is not the case, regularized or stabilized semismooth Newton-type methods can handle this problem. Examples of implementations can be found in the solvers QPALM [23] and FBstab [27].

- Finally, we would like to mention Augmented Lagrangian methods. These methods extend the Lagrangian function with a quadratic penalty term for the violation of the constraints and then iteratively solve the resulting optimization problems. In each iteration, the primal variables are calculated, the Lagrange multipliers are updated and the penalty parameter for the quadratic penalty term is adjusted accordingly. Compared to other penalty methods this approach has the advantage, that the augmented lagrangian function largely preserves smoothness and the probability of ill conditioning is minimized [31]. Different implementations of this method with the purpose to solve QPs can be found, for example, in the solvers QPDO [10], ProxQP [2] and QPALM [23]. Furthermore, OSQP [36] and SCS [32] can be mentioned which uses an Alternating Direction Method of Multipliers (ADMM) approach to solve QPs. ADMM is an approach that also uses the augmented lagrangian and that we will discuss in more detail later.

1.3 QPs in Optimal Control

In addition to many other applications, QPs are often found in the context of optimal control. On the one hand, they arise in the field of model predictive control (MPC) when setting up a time discrete problem in order to calculate the next control step. On the other hand, they can also be of great importance in the area of nonlinear model predictive control (NMPC) when sequential quadratic programming (SQP) approaches are used to solve the resulting nonlinear programs [22]. In SQP, QPs occur as subproblems that are solved iteratively to obtain a solution. Due to the often fast dynamics in the systems to be controlled, a fast solution of the QPs is essential for the successful real-world application of MPC or NMPC.

In this work, we focus on a particular Augmented Lagrangian based method that is easily parallelizable if the problem's objective is linearly separable and the problem's constraints encode a one-step finite difference discretization. This structure is commonplace in the direct approach to optimal control. Especially for large systems, solving the QP from optimal control problems under real-time conditions can quickly become challenging. One approach to meet this challenge is to perform offline calculations in advance in order to speed up the solving of problems that occur online during the ongoing process. An example of this is [1], where the set of constraints is approximated offline by an ellipsoid in order to be able to solve an easier problem online. Furthermore, in [9] an offline trained and fully-connected neural network is used to generate initial values for an online primal active set solver. This warm start can drastically reduce the computation time of the active set solver.

Another approach is to solve the online QPs as efficiently as possible with the help of specialized structure-exploiting solvers. QPs that arise in the context of optimal control problems have a certain block structure due to the time discretization, which can be exploited, and this is the approach that will be discussed in more detail in this article. In the past, considerable effort has been made to utilize this structure, e.g. [38, 25, 3]. A common method in this context is condensing, as described in, for example, [6, 26] and accelerated variants due to [17, 18]. Condensing utilizes the block structure of the QP in the way that it transforms the sparse QP into a smaller dense one. This smaller QP can then be solved efficiently by a general purpose dense QP solver.

In recent years, several solvers have been published to solve block-structured QPs. In addition to the OSQP solver which does not specifically utilize the block structure in the QPs under consideration but generally makes use of the sparseness of the matrices, solvers such as HPIPM [16] or QPALM-OCP [29] have emerged. While HPIPM uses a condensing method and the interior point method, QPALM-QCP is based on the solver QPALM mentioned above. This article presents an ADMM-based solver written in Julia [5] that utilizes the block structure of the QP in such a way that parts of the problem can be solved in parallel and thus fast computation times can be achieved.

1.4 Structure of this article

Section 2 takes a closer look at ADMM and provides a brief introduction to the basics of this method. Section 3 will then present our ADMM approach. First, we consider general QPs and then block structured QPs with the corresponding adaptations of our algorithm for these problems. After some details on the Julia implementation of our solver qpBAMM in section 3.8, numerical results are considered in section 4. Here, we compare qpBAMM with other state-of-the-art QP solvers existing in Julia for solving block-structured QPs. Finally, section 5 draws a conclusion.

1.5 Notation

We denote by $[n] := \{0,\dots,n\}$ the index set of non-negative integers up to n. The $n \times n$ identity matrix is I_n, and superscripts (k), and (ℓ) denote ADMM iteration and subsolver iteration numbers, respectively. Subscript index i denotes a vector component, and subscript index j denotes a time discretization point.

2 Alternating Direction Method of Multipliers

To solve (1), we utilize the ADMM approach. ADMM is an approach to solve convex optimization problems which has become very popular in recent years, although the origins of this method can be traced back to the mid-1970s, see [19]. For a comprehensive survey, we refer to [8]. The idea of this method is to split a convex minimization problem into smaller pieces, both which must be designed such that they are easier to solve than the original problem. The standard ADMM problem formulation is

$$\min_{x\in\mathbb{R}^n, z\in\mathbb{R}^m} \quad f(x)+g(z) \tag{3a}$$

$$\text{s.t.} \quad Ax+Bz=d, \tag{3b}$$

with $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{m\times m}$ and $d \in \mathbb{R}^m$. Furthermore, it is assumed that functions f and g are proper and convex. With $\lambda \in \mathbb{R}^m$ as the Lagrange multiplier for the equality constraint, the Lagrangian function of (3) reads

$$L(x,z,\lambda) = f(x)+g(z)+\lambda^T(Ax+Bz-d). \tag{4}$$

By adding a quadratic penalty term with penalty factor $\rho > 0$, one obtains the Augmented Lagrangian function of (3),

$$L_\rho(x,z,\lambda) = L(x,z,\lambda)+\tfrac{\rho}{2}\|Ax+Bz-d\|^2. \tag{5}$$

It is finally convenient to introduce the scaled Lagrange multiplier $y = \lambda/\rho$ along with the scaled Augmented Lagrangian function

$$L_\rho(x,z,y) = f(x) + g(z) + \tfrac{\rho}{2}(||Ax + Bz - d + y||_2^2 - ||y||_2^2). \tag{6}$$

The ADMM identifies a saddle point of the augmented Lagrangian by iteratively alternating minimizing steps with respect to x and z, followed by a multiplier update derived from the quadratic penalty, as follows:

$$x^{(k+1)} := \arg\min_{x \in \mathbb{R}^n} L_\rho\left(x, z^{(k)}, y^{(k)}\right) \tag{7a}$$

$$z^{(k+1)} := \arg\min_{z \in \mathbb{R}^n} L_\rho\left(x^{(k+1)}, z, y^{(k)}\right) \tag{7b}$$

$$y^{(k+1)} := y^{(k)} + \rho\left(Ax^{(k+1)} + Bz^{(k+1)} - d\right). \tag{7c}$$

The fundamental incentive here is to decompose the problem into functions f and g such that the alternating minimization problems become significantly easier to solve than the joint one. Assuming that a unique minimizer exists, the ADMM iterations show primal-dual convergence under quite mild assumptions. We rely on the following theorem due to [8]:

Theorem 2 (cf. [8]) *Let f and g be proper, closed, and convex functions. Assume that the Lagrangian L has a saddle point, i.e., there exists $(x^*, z^*, y^*) \in \mathbb{R}^{2n+m}$ such that*

$$L_0(x^*, z^*, y) \leq L_0(x^*, z^*, y^*) \leq L_0(x, z, y^*)$$

holds for all $(x,z,y) \in \mathbb{R}^{2n+m}$. Then, as $k \to \infty$, the ADMM iteration (7) satisfies the following:

(i) The residual converges to zero,

$$r^{(k)} := Ax^{(k)} + Bz^{(k)} - d \to 0;$$

(ii) The objective function reaches the optimum value,

$$f(x^{(k)}) + g(z^{(k)}) \to f(x^*) + g(z^*);$$

(iii) The Lagrange multiplier converges to the saddle point value,

$$y^{(k)} \to y^*.$$

3 An ADMM Splitting Scheme for QP

We propose to solve problem (1) by using the ADMM applied to a splitting scheme that differs from previous work as, for example, the solver OSQP [36]. There, the ADMM splitting of (1) was designed such that variable x is associated with the quadratic objective function and the equality constraint, and the variable z was associated with the box constraint. The rationale was that equality constrained convex

QPs may be solved as a large but sparse linear KKT system of equations, and that the cost of projecting onto a box is negligible in comparison.

Introducing appropriate auxiliary variables, we may rewrite (1) as

$$\min_{x,z\in\mathbb{R}^n} \quad \tfrac{1}{2}x^T Hx + h^T x + \chi_{\underline{x}\le x\le\bar{x}}(x) + \chi_{Cz-c=0}(z) \tag{8a}$$

$$\text{s.t.} \quad x = z, \tag{8b}$$

wherein the characteristic function of the set $\mathscr{C}$ is denoted by

$$\chi_{\mathscr{C}}(x) := \begin{cases} 0 & x\in\mathscr{C}, \\ \infty & \text{otherwise.} \end{cases} \tag{9}$$

In doing so, we swap the assignment of the variables to the constraints. We assign the variable x to the quadratic objective and the box constraint, and the variable z to the linear equality constraint. This splitting of the problem now brings changes to the type of subproblems we need to solve: Instead of projecting onto a box, we need to solve box-constrained convex quadratic programs, which essentially amounts to a potentially long sequence of such box projections. Instead of solving large but sparse linear KKT system of equations, we need to solve a smaller sparse linear systems of equality constraints. Both changes bring advantages parallel solution of problems with a specific structure that will be discussed in more detail in §3.7.

To apply the ADMM iterations (7) to the splitting (8), we set

$$f(x) = \tfrac{1}{2}x^T Hx + h^T x + \chi_{\underline{x}\le x\le\bar{x}}(x), \quad g(z) = \chi_{Cz-c=0}(z), \tag{10a}$$

$$A = I,\ B = -I,\ d = 0 \tag{10b}$$

and obtain

$$x^{(k+1)} = \operatorname*{arg\,min}_{x\in\mathbb{R}^n} \left\{ \tfrac{1}{2}x^T Hx + h^T x + \chi_{\underline{x}\le x\le\bar{x}}(x) + \tfrac{\rho}{2}\left\| x - z^{(k)} + \tfrac{1}{\rho}y^{(k)} \right\|_2^2 \right\} \tag{11a}$$

$$z^{(k+1)} = \operatorname*{arg\,min}_{z\in\mathbb{R}^n} \left\{ \chi_{Cz-c=0}(z) + \tfrac{\rho}{2}\left\| x^{(k+1)} - z + \tfrac{1}{\rho}y^{(k)} \right\|_2^2 \right\} \tag{11b}$$

$$y^{(k+1)} = y^{(k)} + \rho(x^{(k+1)} - z^{(k+1)}). \tag{11c}$$

3.1 Update of the x-variables

As a foundation for an efficient implementation, it is essential to represent the individual steps of (11) as efficiently as possible. We may rewrite (11a) as

$$\frac{1}{2}x^T Hx + h^T x + \chi_{\underline{x}\le x\le\bar{x}}(x) + \tfrac{\rho}{2}\left\| x - z^{(k)} + \tfrac{1}{\rho}y^{(k)} \right\|_2^2 \tag{12a}$$

$$= \quad \frac{1}{2}x^T Hx + h^T x + \chi_{\underline{x}\le x\le\bar{x}}(x) + \tfrac{\rho}{2}x^T x - \rho {z^{(k)}}^T x + {y^{(k)}}^T x + \text{const.} \tag{12b}$$

$$= \quad \frac{1}{2}x^T (H + \rho I_n)x + (h + y^{(k)} - \rho z^{(k)})^T x + \chi_{\underline{x}\le x\le\bar{x}}(x) + \text{const.} \tag{12c}$$

The unconstrained minimization problem obtained from this reformulation is equivalent to the bound-constrained convex quadratic subproblem (BCQP)

$$\min_{x\in\mathbb{R}^n} \quad q(x) := \tfrac{1}{2}x^T\hat{H}x + \hat{h}^T x \tag{13a}$$

$$\text{s.t.} \quad \underline{x} \le x \le \overline{x}, \tag{13b}$$

in which $\hat{H} := H + \rho I$ is the Hessian of (1) regularized by the Augmented Lagrangian penalty ρ, $\hat{h} := h + y^{(k)} - \rho z^{(k)}$ is the objective linear part, and the box constraints on x are retained.

Problem (13) is one of the simplest types of quadratic programs. For a feasible point $x \in \mathbb{R}^n$, its necessary and sufficient optimality condition may be expressed by asking for a vanishing projected gradient,

$$\mathrm{P}(x - \nabla q(x); \underline{x}, \overline{x}) = x, \quad \text{with } \mathrm{P}(y;\underline{x},\overline{x})_i = \begin{cases} \underline{x}_i \text{ if } y_i \le \underline{x}_i \\ \overline{x}_i \text{ if } y_i \ge \overline{x}_i \\ y_i \text{ otherwise} \end{cases}, \quad i \in [n]. \tag{14}$$

Since this projection is cheap, problem (13) can be solved by, for example, a nonlinear trust region projected-Newton method as implemented by the solver TRON [28]. To additionally exploit the presence of a convex quadratic objective function, we make use of a textbook gradient projection method, cf. [31]. An iteration of the gradient projection method consists of determining the Cauchy point along the direction of steepest descent of (13a), which is projected onto active variable bounds. To accelerate the otherwise linear rate of convergence of this approach, we approximately solve an unconstrained convex quadratic subproblem on the null-space of the active bounds identified by the Cauchy point. This solution is computed by a truncated [projected conjugate-gradient (CG) method.
To determine the Cauchy point, we consider the piecewise-linear path

$$x(t) = P(x - tg, \underline{x}, \overline{x}), \tag{15}$$

starting from the current iterate $x = x^{(k)}$ and $g = \nabla q(x) = \hat{H}x + \hat{h}$. The path (15) is the projection of the steepest descent direction at x onto the box constraints. The Cauchy point x_C is given by the first local minimizer of the univariate, piecewise-quadratic function $q(x(t))$, for $t \ge 0$. This minimizer can be found by identifying the breakpoints of the piecewise affine-linear parts of $x(t)$ using the formula

$$\hat{t}_i = \begin{cases} (x_i - \overline{x}_i)/g_i & \text{if } g_i < 0 \text{ and } \overline{x}_i < \infty, \\ (x_i - \underline{x}_i)/g_i & \text{if } g_i > 0 \text{ and } \underline{x}_i > -\infty, \\ \infty & \text{otherwise.} \end{cases} \tag{16}$$

Duplicate and zero elements are then deleted from the set of breakpoints $\{\hat{t}_1, \hat{t}_2, ...\}$ and the remaining elements are sorted. We receive the reduced and sorted set $\{t_1, t_2, ..., t_r\}$ with $0 =: t_0 < t_1 < ... < t_r$. The path $x(t)$ is drawn in sequence on the resulting intervals $[t_0,t_1], [t_1,t_2], [t_2,t_3], ...$ until the Cauchy point x_C is found. The calculation of the Cauchy point has been summarized in alg. 1. Once the Cauchy point x_C has been de-

Algorithm 1 *cp*: Cauchy point computation, cf. [31]

```
1: Given: Starting point $x := x^{(k)}$, Set $\ell = 0$.
2: Compute piecewise linear path: Calculate the path $x(t)$ with (15) and the corresponding reduced
   ordered set $\{t_0, t_1, ..., t_r\}$.
3: while $\ell < r$ do
4:     if local minimizer $x_C$ is found on $x(t)$ with $t \in [t_\ell, t_{\ell+1}]$ then
5:         return $x_C$
6:     else
7:         $\ell \leftarrow \ell + 1$
8:     end if
9: end while
```

termined, the gradient projection method can be accelerated by solving the problem

$$\min_{x \in \mathbb{R}^n} \quad q(x) := \tfrac{1}{2} x^T \hat{H} x + \hat{h}^T x \tag{17a}$$

$$\text{s.t.} \quad x_i = x_{C_i}, \qquad \forall i = \mathscr{A}(x_C) \tag{17b}$$

$$\underline{x}_i \leq x_i \leq \overline{x}_i, \quad \forall i \neq \mathscr{A}(x_C), \tag{17c}$$

where

$$\mathscr{A}(x_C) := \{i \mid x_{C_i} = \underline{x}_i \text{ or } x_{C_i} = \overline{x}_i\} \tag{18}$$

defines the active set at the Cauchy point x_C. The problem (17) can be solved with a truncated CG method on the free variables. Note that the problem here does not necessarily have to be solved exactly to achieve an improvement in the convergence rate of the gradient projection method. This offers the option of not letting the CG method iterate to the end, but terminating it after a certain tolerance is reached. In alg. 2 the summarized gradient projection method can be found

Algorithm 2 *gpm*: Accelerated Gradient Projection Method for BCQPs

```
1: Given: Feasible point $x^{(k,0)}$ in ADMM iteration $k$. Set $\ell = 0$.
2: Check convergence. Stop if $x^{(k,\ell)}$ satisfies (14).
3: Compute new candidates.
   (a) Find the Cauchy point $x_C^{(k,\ell)} \leftarrow cp(x^{(k,\ell)})$.
   (b) Find the subspace minimizer $x_N^{(k,\ell)}$ by solving (17).
4: Next iteration. Set
   $$x^{(k,\ell+1)} = \begin{cases} x_N^{(k,\ell)} & \text{if } q\left(x_N^{(k,\ell)}\right) < q\left(x_C^{(k,\ell)}\right) \\ x_C^{(k,\ell)} & \text{otherwise} \end{cases}$$
   and $\ell \leftarrow \ell + 1$. Go to step 2.
```

3.2 Update of the z-variables

The second step of the ADMM iteration (11) consists of the update of the z variables. The z-update (11b) can be written as

$$\min_{z\in\mathbb{R}^n} \quad \tfrac{1}{2}\|v-z\|_2^2 \tag{19a}$$

$$\text{s.t.} \quad Cz = c, \tag{19b}$$

which computes the orthogonal projection of the point $v := x^{(k+1)} + \frac{1}{\rho}y^{(k)}$ onto the null space of the equality constraint $Cz = c$. Efficient and stable methods for computing this projection in the generic case make use of a (typically dense) QR decomposition or a sparse bordered LU decomposition, as presented by Fletcher and Johnson in [14], and solve the problem using the null-space method.

We pursue a Cholesky decomposition approach at the risk of losing some accuracy, since this allows to exploit the structure of the underlying matrix and maintain this structure in the resulting factor. We investigated all three approaches and find this approach to be significantly faster in our setting, while still providing adequate accuracy in our benchmarks. The projection (19) is defined by the normal equation, e.g. [4], through

$$z^* = v - C^T(CC^T)^{-1}(Cv-c), \tag{20}$$

wherein regularity is guaranteed by the full row rank assumption on C. In practice we solve this problem as

$$(CC^T + \mu I)\lambda = Cv + c, \quad z^* = v - C^T\lambda. \tag{21}$$

To solve (21) we compute a Cholesky factorization of $CC^T + \mu I$, and need to do so only once before the first ADMM iteration. The regulation term μI is added in the multiplication so that the positive definiteness of CC^T is not lost due to numerical errors. In the implementation of our method we choose $\mu = 10^{-14}$. Note that hence, the up-front computational cost of the z-variables update amounts to a matrix-matrix multiplication and a Cholesky decomposition of the resulting product, and the running cost comes from two matrix-vector multiplications and two substitution with the Cholesky factor per iteration of the method.

3.3 Preconditioning

Preconditioning a convex nonlinear problem is a well-known approach to potentially reduce the number of iterations of first-order methods such as ADMM. Finding a good preconditioner has been, and still is, an active field of research. It can be shown that, in order to obtain an optimal diagonal preconditioner with respect to the reduction of the condition number of a matrix, the solution of a semidefinite program is required [7]. Since this problem is generally more difficult to solve than (1) itself, it is customary to use a heuristics that finds, for example, a diagonal matrix D for which it can be hoped that it will improve the convergence behaviour. We use Ruiz matrix equilibration, which iteratively re-scales the rows and columns of a matrix such that their respective ℓ_∞-norms convergence to 1 [35]. This matrix equilibration for the Hessian H is easy to implement and requires only minuscule computational effort. We found that it typically helps to significantly reduce the number of iterations of our method. Overall, we calculate the scaling matrix with algorithm 3, wherein H_i denotes the i-th column of the matrix H.

Algorithm 3 *ruiz*: Ruiz Matrix Equilibrium

```
1: Given: Symmetric matrix $H \in \mathbb{R}^{n\times n}$ and tolerance $\varepsilon > 0$.
2: Initialization. Set $H^{(0)} \leftarrow H$, $D^{(0)} \leftarrow I_n$ and $\ell \leftarrow 0$.
3: Check convergence. Stop if $\max_{i=1,\dots,n} \left\{ \left| 1 - \|H_i^{(\ell))}\|_\infty \right| \right\} \le \varepsilon$
4: Compute next iterate.
       $(D_R)_{ii} \leftarrow \sqrt{\|H_i^{(\ell)}\|_\infty}, \quad H^{(\ell+1)} \leftarrow D_R^{-1} H^{(\ell)} D_R^{-1}, \quad D^{(\ell+1)} \leftarrow D^{(\ell)} D_R^{-1}$
5: Next iteration. Set $\ell \leftarrow \ell + 1$ and go to step 3.
```

With $D := D^{(k)}$ from algorithm 3 we get the scaled version of Problem (1)

$$\min_{\hat{x}\in\mathbb{R}^n} \quad \tfrac{1}{2}\hat{x}^T(DHD)\hat{x} + (h^T D)\hat{x} \tag{22a}$$

$$\text{s.t.} \quad (CD)\hat{x} = c, \tag{22b}$$

$$D^{-1}\underline{x} \le \hat{x} \le D^{-1}\overline{x}, \tag{22c}$$

and let $x = D\hat{x}$ after solution of the scaled problem.

Note that in qpBAMM, in addition to the option of scaling the problem using the Hessian matrix H, we also offer the option of scaling using the constraint matrix C or the KKT matrix. In this case, the Ruiz equilibrium from algorithm 3 is also used. The variant used to scale the problem can be selected via the *var_ruiz* option when setting up the problem in qpBAMM. Numerical experiments have shown that scaling with the Hessian matrix is the most useful in our case, which is why it is set as the default value.

3.4 Termination Criterion

To determine whether the current iteration is sufficiently close to the solution according to a specified primal-dual tolerance

$$\|r_p^{(k)}\|_\infty := \|x^{(k)} - z^{(k)}\|_\infty \le \varepsilon_p, \|r_d^{(k)}\|_\infty := \|z^{(k-1)} - z^{(k)}\|_\infty \le \varepsilon_d.$$

Here, $r_p^{(k)}$ is the primal and $r_d^{(k)}$ is the dual residuum of the necessary and sufficient optimality conditions. In order to take into account the relative size of the entries in the iterates, it is beneficial to compose the tolerances ε_p and ε_d of an absolute tolerance $\varepsilon_{abs} > 0$ and relative tolerance ε_{rel}, i.e.,

$$\|r_p^{(k)}\|_\infty \leq \varepsilon_{abs} + \varepsilon_{rel} \max\{\|x^{(k)}\|_\infty, \|z^{(k)}\|_\infty\}, \tag{23a}$$

$$\|r_d^{(k)}\|_\infty \leq \varepsilon_{abs} + \varepsilon_{rel} \max\{\|z^{(k-1)}\|_\infty, \|z^{(k)}\|_\infty\}. \tag{23b}$$

3.5 Penalty parameter and over-relaxation

In the practical application of ADMM, an appropriate choice of the penalty parameter ρ is essential to reduce the number of iterations required until convergence is reached. In [21] it was shown that the optimal fixed penalty parameter can be determined using the largest and smallest eigenvalues of the matrix $CH^{-1}C^T$. Calculating these eigenvalues however may take more time than solving the quadratic problem itself. In practice, heuristics that adjust ρ during the iterations are employed, and adaptive methods based on residual balancing are particularly popular. Residual balancing aims to keep the primal and the dual residuals approximately in the same order of magnitude during the ADMM iterations. We use the basic variant of residual balancing due to [8], which updates ρ according to the rule

$$\rho^{(k+1)} = \begin{cases} \rho^{(k)}/\tau & \text{if } \|r_d^{(k+1)}\|_\infty > \eta \|r_p^{(k+1)}\|_\infty, \\ \tau\rho^{(k)} & \text{if } \|r_p^{(k+1)}\|_\infty > \eta \|r_d^{(k+1)}\|_\infty, \\ \rho^{(k)} & \text{otherwise,} \end{cases} \tag{24}$$

where $\eta > 1$ indicates the difference between the primal and dual residual at which an update is performed and $\tau > 1$ indicates the factor by which ρ is reduced or increased in the case of an update. Note that the ADMM convergence theory is only applicable to a fixed choice of the penalty parameter. Without loss of generality, one would assume that a successful heuristics for the choice of ρ becomes stationary after a certain number of ADMM iterations and the case of a fixed penalty applies.
Another important point that typically leads to a reduction in the number of iterations is over-relaxation. With this technique, the previous iteration is also taken into account when calculating the new iteration. In our case, this results in the x-update $\hat{x}^{(k+1)}$ being calculated as usual, and the over-relaxed x variable becomes

$$x^{(k+1)} = \omega\hat{x}^{(k+1)} + (1-\omega)z^{(k)},$$

which is used in the subsequent ADMM steps. The parameter $\omega \in (0,2)$ is the relaxation parameter, with over-relaxation indicating the choice $\omega > 1$ while $\omega = 1$ is the original ADMM iteration. A choice of the over-relaxation parameter $\omega \in [1.5, 1.8]$ is suggested by, for examaple, [8,21,12].

3.6 qpBAMM Algorithm

We now summarize the generic qpBAMM algorithm that applies to convex QPs without assuming a particular structure of the Hessian or constraints matrix.

Algorithm 4 *qpBAMM*

```
1: Given: Problem of type (MPC) with corresponding block sizes. Initial primal values $x^{(0)} = z^{(0)}$ and
   dual values $y^{(0)}$. Parameters for the solver (see Table 1). Set $k = 0$.
2: Precompute: Transform the Problem with the Ruiz Equilibrium (see Alg. 3). Compute the Cholesky
   factorization of $CC^T$.
3: while $k < \max_{\text{iter}}$ do
4:     if (23) is fulfilled then
5:         return $x^{(k)}, y^{(k)}$
6:     else
7:         Compute the $x$-update: $\hat{x}^{(k+1)} \leftarrow gpm(x^{(k)})$
8:         Over-relaxation: $x^{(k+1)} \leftarrow \omega\hat{x}^{(k+1)} + (1-\omega)z^{(k)}$
9:         Compute the $z$-update: Project onto the equality constraint (see Section 3.2)
10:        Compute the dual update: $y^{(k+1)} \leftarrow y^{(k)} + \rho^k(x^{(k+1)} - z^{(k+1)})$
11:        Update the residuals: $r_p^{(k+1)} \leftarrow x^{(k+1)} - z^{(k+1)}$ and $r_d^{(k+1)} \leftarrow \rho^k(z^{(k)} - z^{(k+1)})$
12:        Update penalty parameter: Set $\rho^{(k+1)}$ with (24)
13:        $k \leftarrow k+1$
14:    end if
15: end while
```

Note that, compared to other ADMM-based QP solvers, we do not require a new factorization of a matrix when updating the penalty ρ. This allows to perform any number of updates of the penalty parameter without significantly affecting the computation time of a single iteration and sometimes helps to lower the iteration count. Furthermore, the cost for the evaluation of the termination criterion are low with our method, since no matrix vector multiplication is required. Thus, we may check the termination criterion in each iteration without significantly affecting the computation time.

3.7 Block-Structured QPs from Direct Optimal Control

The motivation for the proposed ADMM splitting of the QP is that we take interest in quadratic programs with a particular block structure that play a central role in the direct approach to optimal control. We strive to exploit this structure in the ADMM iterations of the splitting approach just presented. We consider a Linear-Quadratic Optimal Control Problem, which can easily be cast in the form (1) and arises, for example, in Linear MPC, cf. [34], or in a Sequential Quadratic Programming approach, e.g. [31], to Nonlinear Optimal Control and NMPC:

$$\begin{aligned}
\min_{\substack{x_0,\dots,x_M\in\mathbb{R}^{n_\mathrm{x}}\\ u_0,\dots,u_{M-1}\in\mathbb{R}^{n_\mathrm{u}}}} \quad & \left(\sum_{j=0}^{M-1} \frac{1}{2}\begin{pmatrix}x_j\\u_j\end{pmatrix}^T \begin{pmatrix}Q_j & S_j^T\\ S_j & R_j\end{pmatrix}\begin{pmatrix}x_j\\u_j\end{pmatrix} + q_j^T x_j + r_j^T u_j\right) + \frac{1}{2}x_M^T Q_M x_M + q_M^T x_M \\
\text{s.t.} \quad & x_0 = \hat{x}_0, \\
& x_{j+1} = E_j x_j + F_j u_j, && j \in [M-1] \quad \text{(MPC)} \\
& \underline{x}_j \le x_j \le \overline{x}_j, && j \in [M], M \ge 1 \\
& \underline{u}_j \le u_j \le \overline{u}_j, && j \in [M-1],
\end{aligned}$$

with state variables $x_j,\ j \in [M]$ and control variables $u_j,\ j \in [M-1]$ as degrees of freedom. The cost weight matrices are $Q_j \in \mathbb{R}^{n_x \times n_\mathrm{x}}$, $R_j \in \mathbb{R}^{n_\mathrm{u} \times n_\mathrm{u}}$ and $S_j \in \mathbb{R}^{n_\mathrm{u} \times n_\mathrm{x}}$ on the stage and control variables, the vectors $q_j \in \mathbb{R}^{n_\mathrm{x}}$ and $r_j \in \mathbb{R}^{n_\mathrm{u}}$ in the objective, the system dynamics matrices $E_j \in \mathbb{R}^{n_\mathrm{x} \times n_\mathrm{x}}$ and $F_j \in \mathbb{R}^{n_\mathrm{x} \times n_\mathrm{u}}$, the lower and upper bounds on the states $\underline{x}_j, \overline{x}_j \in \mathbb{R}^{n_\mathrm{x}}$ and controls $\underline{u}_j, \overline{u}_j \in \mathbb{R}^{n_\mathrm{u}}$. A fixed initial state $\hat{x}_0 \in \mathbb{R}^{n_\mathrm{x}}$ of the system may be represented by setting $\underline{x}_0$ and $\overline{x}_0$ appropriately. If we now define the matrices and vectors from 1 as follows

$$H = \begin{pmatrix} Q_0 & S_0^T & & & & \\ S_0 & R_0 & & & & \\ & & \ddots & & & \\ & & & Q_{M-1} & S_{M-1}^T & \\ & & & S_{M-1} & R_{M-1} & \\ & & & & & Q_M \end{pmatrix} \quad C = \begin{pmatrix} -I_{n_x} & & & & & \\ E_0 & F_0 & -I_{n_x} & & & \\ & & \ddots & \ddots & & \\ & & & E_{M-1} & F_{M-1} & -I_{n_x} \end{pmatrix}$$

$$x = \begin{pmatrix} x_0 \\ u_0 \\ \vdots \\ u_{M-1} \\ x_M \end{pmatrix} \quad h = \begin{pmatrix} q_0 \\ r_0 \\ q_1 \\ \vdots \\ r_{M-1} \\ q_M \end{pmatrix} \quad \underline{x} = \begin{pmatrix} \hat{x}_0 \\ \underline{u}_0 \\ \underline{x}_1 \\ \vdots \\ \underline{u}_{M-1} \\ \underline{x}_M \end{pmatrix} \quad \overline{x} = \begin{pmatrix} \hat{x}_0 \\ \overline{u}_0 \\ \overline{x}_1 \\ \vdots \\ \overline{u}_{M-1} \\ \overline{x}_M \end{pmatrix} \quad c = \begin{pmatrix} -\hat{x}_0 \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

we obtain a problem equivalent to (MPC).

This block structure now allows us to derive specializations of the updates of the ADMM iterates. First of all, because H is block diagonal, we can separate subproblem (13) into M smaller subproblems of dimensions $n_\mathrm{x} + n_\mathrm{u}$, except for the final one, which has dimension n_x. Moreover, these problems significantly smaller problems can be solved in an embarrassingly parallel manner,

$$\min_{\substack{x_j \in \mathbb{R}^{n_\mathrm{x}}, \\ u_j \in \mathbb{R}^{n_\mathrm{u}}}} \ \tfrac{1}{2} x_j^T Q_j x_j + \tfrac{1}{2} u_j^T R_j u_j + u_j^T S_j x_j + q_j^T x_j + r_j^T u_j \tag{26a}$$

$$\text{s.t.} \quad \underline{x}_j \le x_j \le \overline{x}_j, \tag{26b}$$

$$\underline{u}_j \le u_j \le \overline{u}_j. \tag{26c}$$

The z-update steps cannot be parallelized in the same way as the x-update steps due to the coupling conditions contained in C. Nevertheless, the structure of C can be utilized. The multiplications required in the formation of the normal equation matrix CC^T can be written as

$$CC^T = \begin{pmatrix} K_0 & L_0^T & & & & \\ L_0 & K_1 & L_1^T & & & \\ & L_1^T & K_2 & L_2^T & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & L_{M-1}^T \\ & & & & L_{M-1} & K_M \end{pmatrix}$$

with

$$\begin{aligned} K_0 &= I, & K_j &= \begin{pmatrix} E_{j-1} & F_{j-1} \end{pmatrix} \begin{pmatrix} E_{j-1}^T \\ F_{j-1}^T \end{pmatrix} + I, \\ L_0 &= -E_0, & L_j &= -E_j. \end{aligned}$$

The multiplications above can be carried out in parallel on dense matrix blocks, which leads to a substantial reduction in computation time compared to a generic multiplication of the whole sparse matrix for large C. The resulting $K_0, \ldots, K_M$ and $L_0, \ldots, L_{M-1}$ matrix blocks are stored as a list of dense blocks for further calculations. Note that when C is scaled, e.g., by preconditioning, the general structure of C remains unaffected as long as the identity blocks are replaced by diagonal matrices. Furthermore, as described in §3.2 we add the regulation term μI to each block multiplication.

With the list of blocks representing CC^T, the lower left triangular blocks V_j, $j \in [M]$ and the subdiagonal blocks Z_j, $j \in [M-1]$ of a Cholesky factor of matrix CC^T can now be calculated sequentially as follows:

$$\begin{aligned} V_0 V_0^T &= K_0, & & \\ Z_j &= L_j V_j^{-T} & & j \in [M-1], \\ V_{j+1} V_{j+1}^T &= K_{j+1} - Z_j Z_j^T & & j \in [M-1]. \end{aligned}$$

These Cholesky factor blocks can now be used in a sequential forward

$$\begin{aligned} y_0 &= V_0^{-1} \hat{c}_0, & & \\ y_{j+1} &= V_{j+1}^{-1} (\hat{c}_{j+1} - Z_j y_j) & & j \in [M-1] \end{aligned}$$

and backward substitution

$$\begin{aligned} \lambda_M &= V_M^{-T} y_M, & & \\ \lambda_j &= V_j^{-T} (y_j - Z_j^T \lambda_{j+1}) & & j \in [M-1, 0] \end{aligned}$$

in each ADMM iteration to find $\lambda = (\lambda_0^T, \ldots, \lambda_M^T)^T$ with $\hat{c} = (\hat{c}_0^T, \ldots, \hat{c}_M^T)^T = Cv + c$ in (21) .

3.8 Implementation

We have an implementation of qpBAMM in the mathematical programming language Julia. In addition to the ease of use when implementing the method, Julia offers high performance in terms of computing times. Furthermore, Julia enables a simple way of implementing parallelization, which is essential for the competitiveness of the proposed method.

Table 1: Solver parameters that can be set by the user and their default values.

Parameter	Description	Default value	Allowed values
ε_{abs}	Absolute termination tolerance	10^{-3}	$\varepsilon_{abs} \geq 0$
ε_{rel}	Relative termination tolerance	10^{-3}	$\varepsilon_{rel} \geq 0$
$\max_{\text{iter}}$	Maximum number of Iterations	10000	$\max_{\text{iter}} > 0$
$\max_{\text{time}}$	Runtime limit in seconds	3600	$\max_{\text{time}} > 0$
ρ_0	Initial value for the penalty parameter	10	$\rho_0 > 0$
τ	Update factor for ρ	2	$\tau > 1$
η	Factor among residuals for update	10	$\eta > 1$
ρ_{update}	Adaptive penalty parameter	True	True/False
ω	Relaxation parameter	1.8	$\omega \in (0,2)$
verbose	Print output	False	True/False
do_block_mul	Parallel blockwise Matrix-Vector mul.	False	True/False

3.9 Default values parameters

The method presented depends on a number of parameters to be defined by the user. Table 1 gives a brief overview of the default values of these parameters, which proved to work well during the numerical experiments.
The Solver can be found in `https://moto.math.nat.tu-bs.de/world/qpBAMM#`.

4 Numerical Results

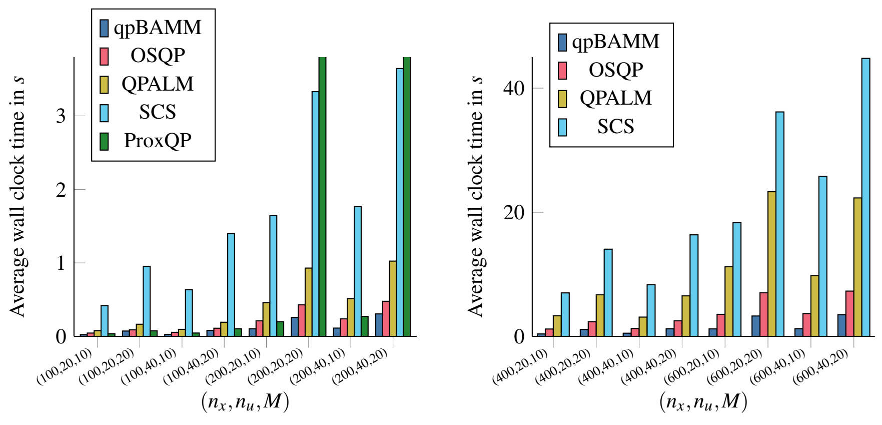
In this section we compare the speed of qpBAMM with other state-of-the-art solvers available in Julia. All experiments were conducted on a MacBook Pro from 2021, with an Apple M1 Pro CPU that contains 10 cores running at a maximum clock speed of 3.2GHz. All computations were performed with 8 parallel threads on the 8 performance cores of the CPU. We use the AppleAccelerate.jl package of Julia to further reduce the computing time of BLAS operations on dense matrix blocks. Besides the ease of parallelization, the fact that the proposed ADMM splitting allows to use accelerated BLAS on dense blocks is a major benefiting factor that help our method to outperform generic sparse solvers.

4.1 Random system

In this section we consider random convex quadratic problems conforming to the blocked form of (MPC). We create the positive definite matrix H by defining, for each block, $H_i := U_i \Lambda_i U_i^T$. Here, U_i is an orthogonal matrix, which we obtain as the result of a QR decomposition of a random matrix with normally distributed entries. Λ_i is a diagonal matrix with equally distributed entries between 10^{-1} and 10^4. Furthermore, we create the matrices E_i, F_i and the vectors q_i and r_i by choosing the entries to be normally distributed. We obtain the lower bounds $\underline{x}$ and $\underline{u}$ by choosing the values uniformly distributed between -5 and 0 and the upper bounds $\overline{x}$ and $\overline{u}$ by choosing the values uniformly distributed between 0 and 5. Finally, we define $\hat{x}_0 = 0 \in \mathbb{R}^n$.

This setup allows us to vary the size of the random QP depending on the size of the state vector n_x, the size of the control vector n_u, and the number of prediction steps M. In our experiments, we choose $n_x = 100, 200, 400, 600$, $n_u = 20, 40$ and $M = 10, 20$. In this way, we obtain 16 different possible problem sizes. For each of these problem variables, we generate 5 instances and thus obtain a total of 80 random instances. Then we solve the generated problems with qpBAMM as well as the general QP solvers OSQP, QPALM, SCS and ProxQP and compare the resulting wall-clock computing times.

While qpBAMM runs parallel computations on 8 threads, the other solvers considered benefit from parallelization only insofar as the linear subsystem solvers can make use of level 2 or level 3 BLAS operations accelerated by Apple's Accelerate framework.



(a) Medium sized instances **(b)** Large sized instances

Fig. 1 Comparison of the wall clock computation times of qpBAMM against OSQP, QPALM, SCS and ProxQP for a different size of random examples. qpBAMM was executed with 8 parallel threads on 8 performance cores. Note that ProxQP is not included in the large sized instances, as the run time limit is exceeded during the calculation. In addition, ProxQP needs much longer for the instances $(200, 20, 20)$ and $(200, 40, 20)$ than shown, as can be seen in the table 2. For display reasons, however, we limit the time in the plot to 4 seconds

We let all solvers solve the problems up to a tolerance of 10^{-3} and imposed a time limit of 100 seconds. Apart from that, we use the default values of all solvers. Furthermore, for each of the 16 different problem sizes, we consider the average solution time over the 5 generated instances, where we give each solver the opportunity to solve the instance twice and using the faster calculation time to reduce the effects of possible background computation of the computer during the calculation. As it is not possible to set an internal time limit in ProxQP, upstream calculations were used to determine which instances of ProxQP significantly exceed the time limit. ProxQP

Table 2: Average wall clock times per run in s for a different size of random problems. qpBAMM was executed with 8 parallel threads on 8 physical cores. Note that ProxQP exceeds the time limit for large systems, which is why no time factor was then determined.

n_x	n_u	M	Average wall clock time in s					Avg. time factor			
			OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP
100	20	10	0.046	0.079	0.420	0.037	**0.025**	1.86	3.21	17.11	1.51
100	20	20	0.090	0.165	0.954	0.076	**0.073**	**1.23**	**2.26**	13.06	**1.04**
100	40	10	0.055	0.096	0.637	0.046	**0.028**	2.00	3.48	**23.12**	1.68
100	40	20	0.111	0.192	1.399	0.105	**0.082**	1.36	2.36	17.15	1.28
200	20	10	0.213	0.460	1.647	0.200	**0.104**	2.04	4.41	15.80	1.92
200	20	20	0.430	0.929	3.330	45.496	**0.258**	1.67	3.60	12.90	176.28
200	40	10	0.239	0.514	1.766	0.272	**0.113**	2.11	4.55	15.62	2.40
200	40	20	0.478	1.024	3.645	50.680	**0.306**	1.56	3.35	11.91	165.65
400	20	10	1.191	3.329	7.010	100.000	**0.400**	2.98	8.33	17.54	
400	20	20	2.377	6.701	14.050	100.000	**1.126**	2.11	5.95	12.48	
400	40	10	1.268	3.110	8.338	100.000	**0.512**	2.48	6.07	16.28	
400	40	20	2.517	6.530	16.365	100.000	**1.240**	2.03	5.27	13.20	
600	20	10	3.553	11.228	18.336	100.000	**1.214**	2.93	**9.25**	15.10	
600	20	20	7.026	23.306	36.165	100.000	**3.288**	2.14	7.09	**11.00**	
600	40	10	3.680	9.803	25.800	100.000	**1.251**	**2.94**	7.83	20.62	
600	40	20	7.299	22.310	44.792	100.000	**3.509**	2.08	6.36	12.77	

was then excluded from the experiment for instances with $n_x = 400$ and $n_x = 600$. In figure 1, we can see that qpBAMM needs the shortest calculation times for all problem sizes considered. Especially for larger problems, qpBAMM is able to solve the problems much faster.

In Table 2, in addition to the average runtimes of the solvers, we show the wall-clock time factor of OSQP, QPALM, SCS and ProxQP in comparison to qpBAMM, which describes the factor by which the respective solver required more time than qpBAMM. The extreme cases are $n_x = 600$, $n_u = 40$ and $M = 10$ where we obtained an average speed-up of 2.94 over OSQP, $n_x = 600$, $n_u = 20$ and $M = 10$ where the speedup was 9.25 over QPALM and $n_x = 100$, $n_u = 40$ and $M = 10$ where a speedup of 23.12 was obtained over SCS. ProxQP also has short computing times for smaller instances. From a certain problem size, however, the solver becomes considerably slower and exceeds the time limit for large sized problems.

4.2 Linear mass spring system

The linear mass spring example is a commonly used benchmark in the field of large scale MPC [38,16,29,11]. Due to the ease of varying the number of masses and springs, corresponding to states and controls, this example is well suited for comparing the solution of block structured QPs of different sizes. For this we formulate the following MPC problem

$$\begin{aligned}
\min_{\substack{x_0,\dots,x_M\in\mathbb{R}^{n_x}\\ u_0,\dots,u_{M-1}\in\mathbb{R}^{n_u}}} \quad & \frac{1}{2}\left(\sum_{k=0}^{M-1} x_k^T Q x_k + u_k^T R u_k + x_M^T Q_M x_M\right) \\
\text{s.t.} \quad & x_0 = \hat{x}_0, \\
& x_{k+1} = E_k x_k + F_k u_k, \qquad k \in [M-1], \\
& -4\cdot \mathbf{1}_{n_x} \le x_k \le 4\cdot \mathbf{1}_{n_x}, \qquad k\in[M], k\ge 1 \\
& -1\cdot \mathbf{1}_{n_u} \le u_k \le 1\cdot \mathbf{1}_{n_u}, \qquad k\in[M-1],
\end{aligned} \tag{LMS}$$

with $Q = Q_M = 3I_{n_x}$ and $R = I_{n_u}$. The initial state of the system $\hat{x}_0$ is a standard normally distributed vector. The size of the problem is given by the number of prediction steps M and the number of masses N_m, where $n_x = 2N_m$ and $n_u = N_m - 1$. The dynamic matrices E_k and F_k are obtained from the linear ODE

$$\dot{x}(t) = A_{ms}x(t) + B_{ms}u(t), \tag{28}$$

where

$$A_{ms} = \left(\begin{array}{cccc|cccc}
\multicolumn{4}{c|}{\mathbf{0}_{\frac{n_x}{2}}} & \multicolumn{4}{c}{I_{\frac{n_x}{2}}} \\ \hline
a_{ms} & c_{ms} & & & b_{ms} & d_{ms} & & \\
c_{ms} & \ddots & \ddots & & d_{ms} & \ddots & \ddots & \\
 & \ddots & \ddots & c_{ms} & & \ddots & \ddots & d_{ms} \\
 & & c_{ms} & a_{ms} & & & d_{ms} & b_{ms}
\end{array}\right) \quad B_{ms} = \left(\begin{array}{cccc}
\multicolumn{4}{c}{\mathbf{0}_{\frac{n_x}{2}\times n_u}} \\ \hline
1 & & & \\
-1 & \ddots & & \\
 & \ddots & 1 & \\
 & & -1 &
\end{array}\right).$$

The parameters $a_{ms} = -2\kappa$, $b_{ms} = -2\delta$, $c_{ms} = \kappa$ and $d_{ms} = \delta$ are derived from the spring constant κ and damping constant δ, which we choose in our experiments as $\kappa = 1$ and $\delta = 0$. To determine the discrete time dynamics we calculate the analytical solution with time step $t_s = 0.5$ and get

$$E_k = \exp(t_s A_{ms}) \text{ and } F_k = A_{ms}^{-1}(\exp(t_s A_{ms}) - I_{n_x})B_{ms} \tag{29}$$

Note that the obtained matrices are dense due to the matrix exponential.

4.3 Wall-clock computation times

For the numerical experiments, we use different numbers of prediction steps $M \in \{5, 10, 20, 30\}$ and masses $N_m \in \{50, 100, 150, 200, 300\}$ to vary the size of the problem. By combining these quantities, we obtain 20 different configurations of the mass spring problem, for which we compare the computation time of qpBAMM again with those of OSQP, QPALM, SCS and ProxQP. For each of these configurations, we solve the problem again five times up to a tolerance of 10^{-3} and impose a time limit of 100 seconds. The remaining parameters are left in the default state for each solver. After solving the problems we compute the mean value of the calculation times for each configuration. It should be noted that in 6 of the total of 100 instances, infeasible problems arose due to the randomly selected initial state of the system. These instances were not taken into account. We also exclude ProxQP for some larger configurations because the time limit would be significantly exceeded, as in some random examples. We do the similar with some instances for SCS, because the building of the expression trees in Convex.jl, which is not part of the computation time, takes so much time that it would be impractical to run the experiments.

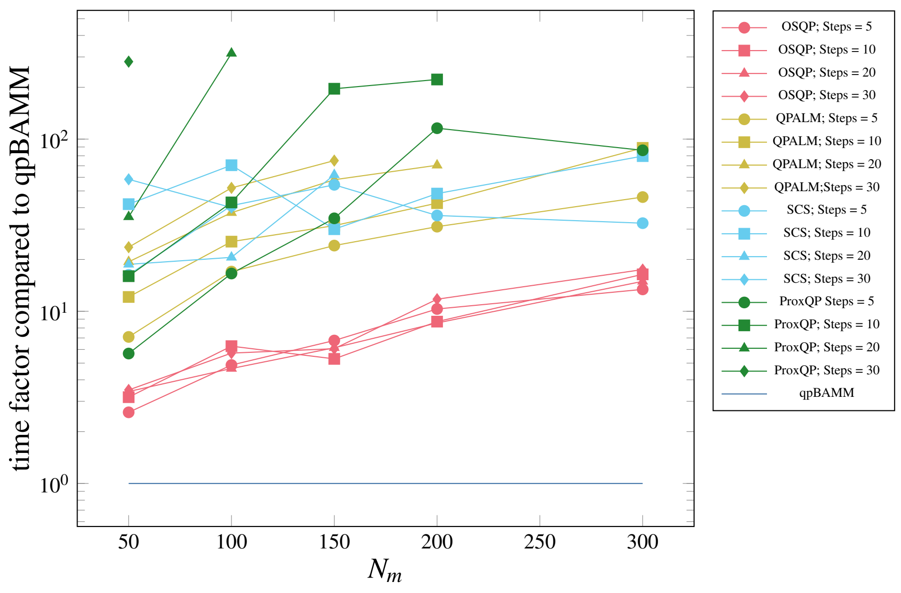
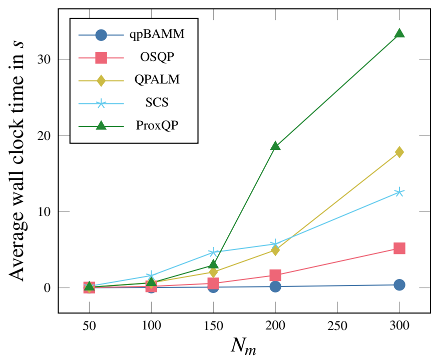
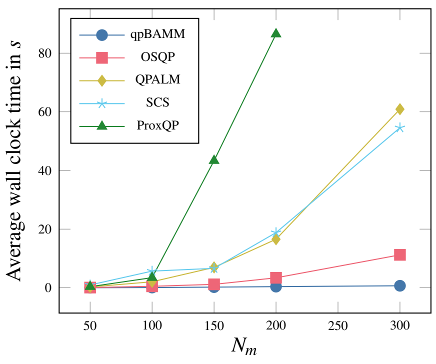


Fig. 2 Comparison of the relative wall clock computation times of qpBAMM against OSQP, QPALM, SCS and ProxQP for a different number of masses. qpBAMM was executed with 8 parallel threads on 8 physical cores. Note that QPALM, SCS and ProxQP exceeding the time limit for larger systems. These values are therefore not shown in the plot.

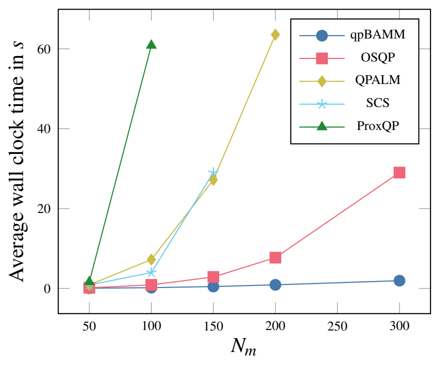
In Figure 2 we show that qpBAMM is able to find a solution in a short wall-clock time for all choices of the prediction horizon length. The effect is more pronounced for problems with many masses. For example, we obtain, on average, a speedup of more than 16 over OSQP, more than 88 over QPALM and of more than 79 over SCS for problems with 300 masses and 10 prediction steps. Missing values for QPALM indicate excess of the time limit of the solver for these problems, which happened for $(N_m, M) \in \{(200,30),(300,20),(300,30)\}$. The same holds for the missing values of SCS and ProxQP.
Figure 3 compares the absolute computation times of the different solvers. Here, it can be observed that the computing time for both OSQP and qpBAMM scales in the same proportion for each number of predictions horizon steps. This speaks in favour of good exploitation of the block structures by the generic sparse solver of OSQP. At the same time, the performance bottleneck for qpBAMM is the availability of only



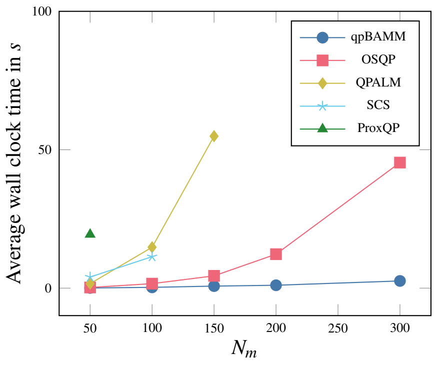
(a) $M = 5$



(b) $M = 10$



(c) $M = 20$



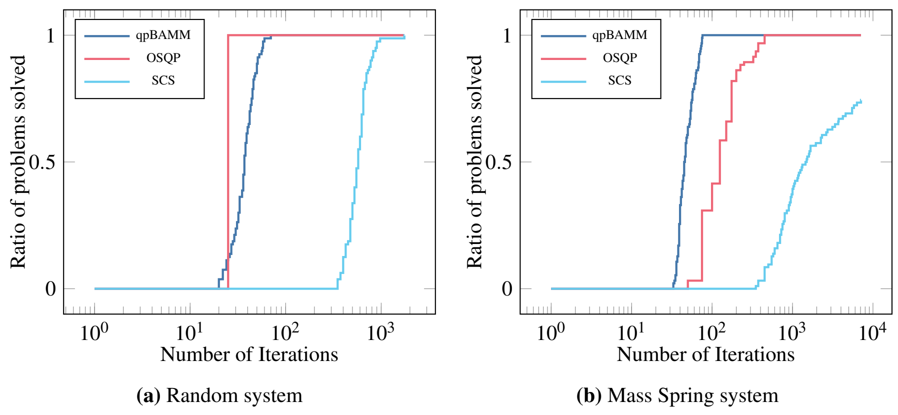
(d) $M = 30$

Fig. 3 Comparison of the absolute wall clock computation times of qpBAMM against OSQP, QPALM, SCS and ProxQP for a different number of masses. qpBAMM was executed with 8 parallel threads on 8 physical cores. Note that QPALM, SCS and ProxQP exceeding the time limit for the larger systems. These values are therefore not shown in the plots.

8 performance cores. QPALM and ProxQP requires proportionally more time for a higher number of steps in order to solve the problems up to the given tolerance. For SCS, no clear statements can be made here, as the number of iterations varies greatly within the instances, resulting in very different computation times.

4.4 Numbers of Iterations

In addition to the pure runtimes, the required iterations until convergence for the ADMM-based solvers are also taken of interest. When comparing qpBAMM with the ADMM based solvers OSQP or SCS, it should be noted that the method presented in this article does not require a new factorization when updating the penalty parameter, which leads to a much more flexible choice of the penalty parameters. It should also be mentioned that OSQP and SCS, when using the default settings, only checks whether the algorithm has converged every 25 iterations. In our case, since no matrix vector multiplication is required when checking the termination criterion, the cost is very low and we can perform this after each iteration. Since QPALM and ProxQP are based on the Augmented Lagrangian but not on an ADMM splitting they generally requires significantly fewer, but more expensive iterations. For this reason, they are not taken into account when comparing the iteration numbers.



(a) Random system **(b)** Mass Spring system

Fig. 4 Comparison of the number of iterations required until convergence is reached between the ADMM based solvers qpBAMM, OSQP and SCS. qpBAMM was executed with 8 parallel threads on 8 physical cores.

Figure 4 shows that OSQP always converged after 25 iterations for the randomly generated problems, which tend to be easier, whereby it should be noted that the termination criterion could be fulfilled after less than 25 iterations. qpBAMM also only needs a few dozen iterations to solve the randomly generated problems. Greater differences can be seen in the more realistic mass spring problems. Here again, qpBAMM only needs a few dozen iterations, while OSQP needs several hundred iterations to converge, especially for problems with a large number of masses. SCS, on the other hand, requires up to several thousand iterations for both problem types until convergence is achieved

5 Conclusion

In this paper we have presented the new ADMM based solver for block structured QPs qpBAMM. The presented method uses a splitting of the problem which leads to the fact that per iteration a box constrained QP has to be solved and a projection onto an equality constraint has to be performed. Given the existing structure of the problem, we have shown that we can greatly accelerate both the solution of the box constrained QPs and the projection onto the equality constraint. Finally, we were able to show based on numerical experiments that qpBAMM is competitive with state-of-the-art general-purpose QP solvers regarding block structures QPs.

Acknowledgements M. Lahmann and C. Kirches acknowledge funding by Deutsche Forschungsgemeinschaft (DFG) in the context of the SPP 2364 "Autonomous Processes in Particle Technology". C. Kirches and M.T. Köhler are funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work is supported by ERC grant SCARCE, 101087662.

6 Appendix

Table 3: Average wall clock computational times in s for a different size of mass spring problems. qpBAMM was executed with 8 parallel threads on 8 physical cores. Note that QPALM, SCS and ProxQP exceeding the time limit for large systems, which is why no time factor was then determined.

N_M	M	Avg. wall clock time in s					Avg. time factor			
		OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP
50	5	0.033	0.091	0.208	0.073	**0.013**	**2.59**	**7.10**	**16.28**	**5.68**
50	10	0.073	0.278	0.963	0.367	**0.023**	3.17	12.11	41.90	15.99
50	20	0.157	0.888	0.858	1.619	**0.046**	3.42	19.39	18.73	35.35
50	30	0.240	1.620	4.016	19.377	**0.069**	3.49	23.56	58.41	281.79
100	5	0.190	0.664	1.604	0.647	**0.039**	4.87	17.01	41.10	16.57
100	10	0.505	2.044	5.681	3.454	**0.081**	6.27	25.38	70.53	42.88
100	20	0.900	7.229	3.977	60.841	**0.193**	4.65	37.38	20.57	314.63
100	30	1.621	14.772	11.356	100.000	**0.284**	5.72	52.09	40.04	
150	5	0.580	2.066	4.656	2.974	**0.086**	6.76	24.08	54.27	34.67
150	10	1.169	6.950	6.624	43.332	**0.221**	5.29	31.45	29.98	196.10
150	20	2.874	27.189	29.019	100.000	**0.468**	6.14	58.09	62.00	
150	30	4.438	54.901	100.000	100.000	**0.732**	6.06	75.02		
200	5	1.649	4.958	5.756	18.497	**0.160**	10.31	31.01	36.00	115.70
200	10	3.394	16.551	18.807	86.489	**0.390**	8.71	42.47	48.26	221.94
200	20	7.711	63.549	100.000	100.000	**0.901**	8.55	70.50		
200	30	12.261	100.000	100.000	100.000	**1.043**	11.75			
300	5	5.183	17.813	12.564	33.294	**0.387**	13.40	46.07	32.49	86.10
300	10	11.223	60.892	54.567	100.000	**0.685**	16.39	88.91	79.68	
300	20	29.018	100.000	100.000	100.000	**1.944**	14.92			
300	30	45.356	100.000	100.000	100.000	**2.597**	**17.47**			

Table 4: Raw data for a different size of mass spring problems. qpBAMM was executed with 8 parallel threads on 8 physical cores. Note that QPALM, SCS and ProxQP exceeding the time limit for large systems, which is why no time factor was then determined.

N_M	M	Iterations					Wall clock time in s					Time factor			
		OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP
50	5	75	27	375	8	50	0.036	0.089	0.125	0.074	**0.014**	2.61	**6.49**	9.14	**5.40**
50	5	75	31	950	8	33	0.035	0.103	0.282	0.076	**0.012**	2.99	8.73	23.85	6.42
50	5	50	26	1025	7	37	0.030	0.092	0.302	0.073	**0.013**	**2.26**	6.96	22.80	5.50
50	5	50	26	625	5	35	0.030	0.087	0.196	0.072	**0.013**	2.29	6.72	15.16	5.59
50	5	75	23	425	6	34	0.035	0.083	0.136	0.069	**0.012**	2.85	6.75	11.07	5.59
50	10	100	44	5800	9	48	0.083	0.301	3.231	0.402	**0.026**	3.27	11.81	126.70	15.78
50	10	75	38	850	7	49	0.073	0.273	0.522	0.368	**0.022**	3.23	12.16	23.25	16.40
50	10	75	39	725	9	38	0.073	0.273	0.452	0.356	**0.019**	3.89	14.52	24.04	18.90
50	10	75	38	550	8	45	0.073	0.260	0.353	0.341	**0.022**	3.28	11.64	15.80	15.29
50	10	50	41	375	9	56	0.062	0.284	0.255	0.370	**0.026**	2.40	11.02	9.90	14.34
50	20	100	67	350	15	72	0.169	0.964	0.490	1.959	**0.059**	2.86	16.28	**8.28**	33.10
50	20	75	52	800	13	57	0.148	0.759	1.002	1.681	**0.048**	3.08	15.83	20.90	35.06
50	20	75	65	900	10	40	0.148	0.929	1.116	1.428	**0.036**	4.12	25.91	31.11	39.81
50	20	100	64	575	14	41	0.170	0.899	0.748	1.654	**0.038**	4.53	23.96	19.94	44.09
50	20	75	62	700	11	55	0.148	0.890	0.934	1.374	**0.048**	3.06	18.34	19.25	28.34
50	30	75	71	4225	12	67	0.226	1.575	7.384	17.183	**0.083**	2.72	18.95	88.83	206.70
50	30	75	64	600	10	41	0.226	1.484	1.207	13.284	**0.056**	4.04	26.61	21.63	238.14
50	30	100	82	425	15	54	0.262	1.822	0.875	23.456	**0.070**	3.72	25.89	12.43	333.26
50	30	100	72	2325	13	47	0.261	1.627	4.146	21.147	**0.063**	4.13	25.73	65.55	334.34
50	30	75	69	3700	16	54	0.226	1.592	6.471	21.817	**0.071**	3.18	22.33	90.78	306.07
100	5	75	33	675	7	36	0.189	0.701	0.813	0.508	**0.036**	5.22	19.29	22.37	13.99
100	5	75	32	1450	7	55	0.191	0.644	1.610	0.520	**0.051**	3.73	12.59	31.46	10.16
100	5	75	32	2250	9	36	0.190	0.644	2.383	0.840	**0.036**	5.30	18.01	66.61	23.48
100	5	75	30	2250	9	36	0.190	0.636	2.368	0.534	**0.036**	5.28	17.63	65.59	14.79
100	5	75	33	675	9	36	0.190	0.693	0.845	0.832	**0.036**	5.32	19.38	23.66	23.27
100	10	125	49	2725	10	39	0.587	2.097	5.896	3.646	**0.075**	7.83	27.99	78.69	48.66
100	10	75	47	1150	9	40	0.396	1.961	2.663	3.158	**0.077**	5.13	25.40	34.50	40.90
100	10	100	47	1600	9	50	0.438	1.974	3.586	3.325	**0.091**	4.80	21.63	39.30	36.44
100	10	150	54	7100	11	38	0.523	2.238	14.837	3.919	**0.074**	7.11	30.43	201.74	53.29

N_M	M	Iterations					Wall clock time in s					Time factor			
		OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP
100	10	125	47	550	10	47	0.583	1.951	1.423	3.220	**0.086**	6.79	22.75	16.58	37.54
100	20	125	78	400	17	42	0.982	6.782	2.455	73.152	**0.168**	5.84	40.29	14.59	434.63
100	20	100	89	1500	16	47	0.901	7.665	6.938	80.186	**0.200**	4.50	38.31	34.68	400.80
100	20	75	82	500	12	48	0.813	7.441	2.722	47.469	**0.185**	4.38	40.13	14.68	256.00
100	20	75	86	800	13	57	0.813	7.680	3.992	61.309	**0.212**	3.83	36.19	18.81	288.87
100	20	125	72	750	11	54	0.989	6.576	3.779	42.088	**0.201**	4.93	32.74	18.81	209.56
100	30	225	116	1625		47	2.023	15.680	11.530	100.000	**0.271**	7.45	57.77	42.48	
100	30	100	125	1675		69	1.356	16.862	11.854	100.000	**0.382**	3.55	44.13	31.02	
100	30	175	90	550		39	1.755	12.300	4.667	100.000	**0.235**	7.47	52.36	19.87	
100	30	125	80	3375		39	1.489	11.703	22.864	100.000	**0.227**	6.55	51.45	100.51	
100	30	125	128	775		53	1.484	17.314	5.867	100.000	**0.302**	4.91	57.30	19.42	
150	5	75	35	725	9	33	0.541	2.110	2.061	3.009	**0.088**	6.16	24.05	23.50	34.30
150	5	125	33	900	9	39	0.640	2.085	2.458	3.004	**0.084**	7.58	24.70	29.12	35.59
150	5	75	31	5500	7	36	0.540	1.935	12.744	2.909	**0.078**	6.94	24.87	163.79	37.39
150	5	75	34	1050	8	40	0.542	2.089	2.784	2.869	**0.084**	6.45	24.85	33.11	34.13
150	5	125	34	1250	8	46	0.639	2.113	3.235	3.081	**0.095**	6.73	22.24	34.05	32.43
150	10	100	56	450	11	63	1.235	7.001	2.972	45.236	**0.264**	4.68	26.55	11.27	171.55
150	10	75	54	975	10	53	1.129	6.647	5.409	41.552	**0.223**	5.05	29.75	24.21	186.00
150	10	75	58	3125	10	41	1.139	7.050	15.477	41.783	**0.182**	6.24	38.63	84.81	228.95
150	10	100	57	1200	10	60	1.218	7.226	6.410	41.354	**0.253**	4.82	28.57	25.34	163.49
150	10	75	51	425	11	42	1.126	6.824	2.851	46.735	**0.182**	6.17	37.43	15.64	256.36
150	20	125	104	6325		64	2.687	26.216	61.355	100.000	**0.563**	4.77	46.58	109.01	
150	20	225	102	1000		58	3.466	27.880	11.167	100.000	**0.492**	7.04	56.65	22.69	
150	20	125	94	5475		38	2.677	25.833	53.315	100.000	**0.335**	8.00	77.21	159.35	
150	20	125	105	600		69	2.671	27.475	7.413	100.000	**0.564**	4.74	48.75	13.15	
150	20	150	109	1075		45	2.867	28.541	11.847	100.000	**0.387**	7.41	73.73	30.60	
150	30	175	137			45	4.634	55.619	100.000	100.000	**0.589**	7.87	94.49		
150	30	150	126			52	4.320	52.945	100.000	100.000	**0.666**	6.49	79.49		
150	30	200	159			76	4.896	61.894	100.000	100.000	**0.946**	5.18	65.46		
150	30	125	119			75	4.024	48.999	100.000	100.000	**0.934**	4.31	52.49		
150	30	150	134			40	4.315	55.047	100.000	100.000	**0.525**	8.21	104.78		
200	5	175	38	1175	8	43	1.656	5.112	5.631	17.974	**0.159**	10.41	32.14	35.41	113.02

N_M	M	Iterations					Wall clock time in s					Time factor			
		OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP
200	5	175	34	1325	9	55	1.696	4.881	6.241	20.395	**0.197**	8.62	24.79	31.70	103.59
200	5	175	34	1675	9	37	1.657	4.739	7.519	18.161	**0.136**	12.18	34.85	55.29	133.55
200	5	125	37	650	8	40	1.586	5.101	3.634	17.459	**0.148**	10.75	34.56	24.61	118.27
200	10	125	55	750	10	39	3.297	15.605	8.553	74.833	**0.292**	11.28	53.39	29.27	256.04
200	10	125	65	1950	11	44	3.359	17.175	17.976	85.617	**0.318**	10.57	54.06	56.58	269.49
200	10	175	59	950	13	68	3.627	17.100	10.108	100.681	**0.474**	7.65	36.07	21.32	212.40
200	10	125	57	4575	11	68	3.294	16.324	38.589	84.823	**0.475**	6.94	34.38	81.27	178.65
200	20	175	118			57	7.598	65.640	100.000	100.000	**0.817**	9.30	80.38		
200	20	200	104			73	7.914	58.753	100.000	100.000	**1.019**	7.76	57.64		
200	20	250	137			69	8.662	77.835	100.000	100.000	**1.107**	7.83	70.34		
200	20	150	90			62	7.196	53.447	100.000	100.000	**0.880**	8.18	60.75		
200	20	150	114			47	7.185	62.071	100.000	100.000	**0.685**	10.49	90.60		
200	30	150				40	10.959	100.000	100.000	100.000	**0.888**	12.34			
200	30	175				40	11.473	100.000	100.000	100.000	**0.898**	12.77			
200	30	350				75	15.279	100.000	100.000	100.000	**1.575**	9.70			
200	30	175				44	11.526	100.000	100.000	100.000	**0.977**	11.80			
200	30	200				39	12.071	100.000	100.000	100.000	**0.878**	13.75			
300	5	175	38	1300	9	35	5.179	16.813	14.921	39.361	**0.340**	15.21	49.38	43.82	115.60
300	5	175	39	800	9	45	5.237	18.444	10.839	34.650	**0.361**	14.50	51.07	30.01	95.94
300	5	175	39	1000	7	45	5.241	17.999	12.638	27.240	**0.370**	14.18	48.71	34.20	73.72
300	5	175	38	925	8	63	5.076	17.998	11.858	31.927	**0.476**	10.67	37.84	24.93	67.12
300	10	175	67	2625		38	11.085	59.213	54.267	100.000	**0.639**	17.34	92.63	84.89	
300	10	175	68	1525		39	11.093	61.803	35.702	100.000	**0.645**	17.20	95.81	55.35	
300	10	200	67	3750		46	11.491	61.659	73.732	100.000	**0.770**	14.92	80.06	95.73	
300	20	325				39	27.947	100.000	100.000	100.000	**1.467**	19.05	68.43	68.17	68.17
300	20	375				74	29.391	100.000	100.000	100.000	**2.429**	12.10	41.29	41.16	41.16
300	20	325				71	27.853	100.000	100.000	100.000	**2.332**	11.94	42.90	42.88	42.88
300	20	425				42	30.880	100.000	100.000	100.000	**1.549**	**19.94**	64.70	64.56	64.56
300	30	350				43	43.365	100.000	100.000	100.000	**2.233**	19.42	46.52	44.77	44.77
300	30	400				62	45.626	100.000	100.000	100.000	**3.092**	14.76	33.85	32.34	32.34
300	30	375				50	44.953	100.000	100.000	100.000	**2.481**	18.12	42.09	40.31	40.31
300	30	450				60	48.277	100.000	100.000	100.000	**2.923**	16.52	35.73	34.21	34.21

N_M	M	Iterations					Wall clock time in s					Time factor			
		OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP
300	30	375				43	44.558	100.000	100.000	100.000	**2.254**	19.76	46.19	44.36	44.36

Table 5: Raw data for a different size of random problems. qpBAMM was executed with 8 parallel threads on 8 physical cores.Note that ProxQP exceeds the time limit for large systems, which is why no time factor was then determined.

n_x	n_u	M	Iterations					Wall clock time in s					Time factor			
			OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP
100	20	10	25	8	525	0	24	0.047	0.083	0.460	0.037	**0.022**	2.11	3.71	20.68	1.67
100	20	10	25	6	475	0	30	0.045	0.065	0.421	0.038	**0.025**	1.84	2.62	17.01	1.53
100	20	10	25	8	375	0	38	0.045	0.081	0.341	0.036	**0.029**	1.53	2.79	11.72	1.25
100	20	10	25	8	650	0	26	0.045	0.082	0.555	0.036	**0.024**	1.85	3.36	22.85	1.49
100	20	10	25	9	350	0	27	0.046	0.084	0.321	0.038	**0.022**	2.06	3.77	14.46	1.69
100	20	20	25	8	400	0	33	0.091	0.164	0.712	**0.077**	0.082	**1.11**	2.00	8.67	**0.94**
100	20	20	25	8	900	0	42	0.089	0.166	1.497	0.078	**0.071**	1.26	2.36	21.23	1.10
100	20	20	25	8	475	0	39	0.091	0.165	0.826	0.076	**0.065**	1.41	2.53	12.70	1.17
100	20	20	25	8	500	0	44	0.089	0.164	0.868	0.075	**0.072**	1.24	2.28	12.06	1.04
100	20	20	25	8	500	0	51	0.089	0.166	0.866	**0.075**	0.076	1.18	2.20	11.47	0.99
100	40	10	25	10	725	0	20	0.055	0.122	0.813	0.046	**0.025**	2.17	4.79	31.93	1.82
100	40	10	25	8	550	0	20	0.055	0.099	0.593	0.046	**0.024**	2.25	4.07	24.32	1.89
100	40	10	25	6	475	0	22	0.056	0.079	0.516	0.046	**0.026**	2.16	3.06	19.99	1.78
100	40	10	25	8	550	0	32	0.055	0.100	0.597	0.046	**0.030**	1.82	3.32	19.79	1.53
100	40	10	25	6	625	0	33	0.055	0.079	0.664	0.047	**0.032**	1.72	2.48	20.84	1.46
100	40	20	25	10	875	0	38	0.111	0.243	1.789	0.095	**0.083**	1.33	2.92	21.48	1.15
100	40	20	25	8	600	0	50	0.110	0.203	1.262	**0.095**	0.097	1.13	2.09	12.97	0.97
100	40	20	25	6	475	0	20	0.113	0.157	1.028	0.092	**0.065**	1.74	2.43	15.89	1.42
100	40	20	25	8	575	1	28	0.110	0.201	1.220	0.147	**0.077**	1.43	2.62	15.86	1.91
100	40	20	25	6	825	0	42	0.111	0.157	1.697	0.093	**0.086**	1.30	**1.83**	19.81	1.09
200	20	10	25	8	525	0	37	0.212	0.456	1.659	0.200	**0.110**	1.94	4.16	15.15	1.83
200	20	10	25	8	425	0	24	0.212	0.455	1.388	0.199	**0.096**	2.22	4.75	14.51	2.08

n_x	n_u	M	Iterations					Wall clock time in s					Time factor			
			OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP
200	20	10	25	8	525	0	29	0.212	0.461	1.657	0.201	**0.092**	2.30	4.99	17.97	2.18
200	20	10	25	10	700	0	42	0.214	0.467	2.146	0.201	**0.120**	1.78	3.87	17.81	1.66
200	20	10	25	8	425	0	32	0.213	0.463	1.386	0.200	**0.103**	2.06	4.47	13.39	1.93
200	20	20	25	8	400	0	43	0.428	0.919	2.551	38.435	**0.285**	1.50	3.23	8.97	135.09
200	20	20	25	8	600	0	47	0.428	0.914	3.608	36.745	**0.303**	1.41	3.01	11.89	121.08
200	20	20	25	8	625	0	43	0.426	0.922	3.750	37.889	**0.281**	1.52	3.28	13.37	135.05
200	20	20	25	8	450	0	22	0.434	0.927	2.834	35.691	**0.201**	2.16	4.62	14.12	177.80
200	20	20	25	10	650	1	27	0.433	0.965	3.906	78.722	**0.221**	1.96	4.36	17.66	355.91
200	40	10	25	8	500	0	39	0.240	0.517	1.812	0.227	**0.123**	1.95	4.21	14.74	1.84
200	40	10	25	8	575	0	37	0.240	0.515	2.028	0.225	**0.159**	1.51	3.25	12.79	1.42
200	40	10	25	8	625	1	29	0.239	0.514	2.178	0.454	**0.105**	2.27	4.90	20.73	4.32
200	40	10	25	8	400	0	22	0.237	0.511	1.483	0.227	**0.088**	2.71	5.83	16.92	2.59
200	40	10	25	8	350	0	24	0.239	0.513	1.330	0.225	**0.091**	2.63	5.63	14.61	2.48
200	40	20	25	8	525	0	25	0.479	1.024	3.616	36.517	**0.249**	1.92	4.11	14.52	146.66
200	40	20	25	9	650	1	45	0.482	1.033	4.352	96.791	**0.373**	1.29	2.77	11.67	259.46
200	40	20	25	8	425	0	38	0.476	1.021	3.016	41.068	**0.295**	1.61	3.46	10.22	139.10
200	40	20	25	8	575	0	40	0.474	1.022	3.924	38.952	**0.335**	1.41	3.05	11.72	116.32
200	40	20	25	8	475	0	37	0.478	1.021	3.314	40.072	**0.278**	1.72	3.68	11.94	144.36
400	20	10	25	8	350		30	1.193	2.910	4.926	100.000	**0.359**	3.33	8.11	13.73	
400	20	10	25	10	650		33	1.189	3.598	8.055	100.000	**0.372**	3.20	9.68	21.66	
400	20	10	25	10	625		56	1.193	3.603	7.787	100.000	**0.524**	2.28	6.88	14.86	
400	20	10	25	8	650		36	1.189	2.911	8.053	100.000	**0.407**	2.92	7.15	19.78	
400	20	10	25	11	475		27	1.190	3.623	6.226	100.000	**0.336**	3.54	10.77	18.51	
400	20	20	25	9	550		31	2.358	5.872	13.690	100.000	**1.059**	2.23	5.55	12.93	
400	20	20	25	9	700		33	2.390	5.884	16.781	100.000	**1.096**	2.18	5.37	15.31	
400	20	20	25	10	650		42	2.375	7.220	15.770	100.000	**1.216**	1.95	5.94	12.97	
400	20	20	25	11	550		36	2.382	7.301	13.770	100.000	**1.136**	2.10	6.43	12.13	
400	20	20	25	10	375		35	2.382	7.225	10.240	100.000	**1.122**	2.12	6.44	9.13	
400	40	10	25	8	775		50	1.271	3.109	10.028	100.000	**0.515**	2.47	6.04	19.47	
400	40	10	25	8	550		41	1.271	3.120	7.512	100.000	**0.447**	2.84	6.98	16.81	
400	40	10	25	8	525		58	1.268	3.108	7.218	100.000	**0.586**	2.16	5.30	12.31	
400	40	10	25	8	775		57	1.268	3.113	9.989	100.000	**0.574**	2.21	5.42	17.40	

n_x	n_u	M	Iterations					Wall clock time in s					Time factor			
			OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP	qpBAMM	OSQP	QPALM	SCS	ProxQP
400	40	10	25	8	500		38	1.264	3.102	6.941	100.000	**0.438**	2.89	7.08	15.85	
400	40	20	25	12	800		58	2.517	7.744	20.125	100.000	**1.506**	1.67	5.14	13.37	
400	40	20	25	10	500		33	2.514	6.286	13.649	100.000	**1.148**	2.19	5.47	11.89	
400	40	20	25	8	575		37	2.519	6.190	15.283	100.000	**1.206**	2.09	5.13	12.68	
400	40	20	25	8	575		32	2.518	6.197	15.299	100.000	**1.124**	2.24	5.51	13.61	
400	40	20	25	9	675		37	2.520	6.234	17.469	100.000	**1.215**	2.07	5.13	14.38	
600	20	10	25	10	650		40	3.540	9.520	19.687	100.000	**1.159**	3.05	8.22	16.99	
600	20	10	25	12	400		70	3.570	11.670	13.624	100.000	**1.673**	2.13	6.97	8.14	
600	20	10	25	12	650		39	3.561	11.720	19.679	100.000	**1.149**	3.10	10.20	17.13	
600	20	10	25	10	675		31	3.557	11.580	20.249	100.000	**1.010**	**3.52**	**11.47**	20.06	
600	20	10	25	10	600		36	3.536	11.650	18.443	100.000	**1.080**	3.27	10.79	17.08	
600	20	20	25	11	625		60	7.012	23.211	37.271	100.000	**3.781**	1.85	6.14	9.86	
600	20	20	25	10	575		46	7.030	18.902	34.983	100.000	**3.115**	2.26	6.07	11.23	
600	20	20	25	14	425		36	7.028	27.804	27.923	100.000	**2.980**	2.36	9.33	9.37	
600	20	20	25	13	625		51	7.028	23.468	37.368	100.000	**3.233**	2.17	7.26	11.56	
600	20	20	25	10	750		44	7.031	23.145	43.282	100.000	**3.332**	2.11	6.95	12.99	
600	40	10	25	8	550		35	3.696	9.767	18.101	100.000	**1.104**	3.35	8.85	16.40	
600	40	10	25	8	700		48	3.677	9.798	21.816	100.000	**1.331**	2.76	7.36	16.39	
600	40	10	25	8	475		45	3.669	9.783	16.163	100.000	**1.297**	2.83	7.54	12.46	
600	40	10	25	8	1750		53	3.675	9.767	53.676	100.000	**1.398**	2.63	6.99	**38.39**	
600	40	10	25	10	600		37	3.684	9.901	19.245	100.000	**1.127**	3.27	8.79	17.08	
600	40	20	25	11	400		45	7.298	24.332	27.835	100.000	**3.496**	2.09	6.96	**7.96**	
600	40	20	25	10	825		46	7.304	24.078	48.725	100.000	**3.549**	2.06	6.78	13.73	
600	40	20	25	9	625		36	7.293	19.608	38.891	100.000	**3.234**	2.26	6.06	12.02	
600	40	20	25	8	900		46	7.297	19.458	52.404	100.000	**3.531**	2.07	5.51	14.84	
600	40	20	25	10	975		50	7.304	24.075	56.106	100.000	**3.732**	1.96	6.45	15.03	

References

1. van Antwerp, J.G., Braatz, R.D.: Model predictive control of large scale processes. Journal of Process Control **10**(1), 1–8 (2000)
2. Bambade, A., Schramm, F., El-Kazdadi, S., Caron, S., Taylor, A., Carpentier, J.: Proxqp: an efficient and versatile quadratic programming solver for real-time robotics applications and beyond (2023)
3. Bartlett, R.A., Biegler, L.T., Backstrom, J., Gopal, V.: Quadratic programming algorithms for large-scale model predictive control. Journal of Process Control **12**(7), 775–795 (2002)
4. Beck, A.: First-Order Methods in Optimization. Society for Industrial and Applied Mathematics (2017)
5. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM Review **59**(1), 65–98 (2017)
6. Bock, H., Plitt, K.: A multiple shooting algorithm for direct solution of optimal control problems. IFAC Proceedings Volumes **17**(2), 1603–1608 (1984)
7. Boyd, S., El Ghaoui, L., Feron, E., Balakrishnan, V.: Linear Matrix Inequalities in System and Control Theory. Society for Industrial and Applied Mathematics (1994)
8. Boyd, S., Parikh, N., Chu, E., Peleato, B., Eckstein, J.: Distributed optimization and statistical learning via the alternating direction method of multipliers. Foundations and Trends® in Machine Learning **3**(1), 1–122 (2011)
9. Chen, S.W., Wang, T., Atanasov, N., Kumar, V., Morari, M.: Large scale model predictive control with neural networks and primal active sets. Automatica **135**, 109947 (2022)
10. De Marchi, A.: On a primal-dual Newton proximal method for convex quadratic programs. Computational Optimization and Applications **81**(2), 369–395 (2022)
11. Domahidi, A., Zgraggen, A., Zeilinger, M., Morari, M., Jones, C.: Efficient interior point methods for multistage problems arising in receding horizon control. pp. 668–674 (2012)
12. Eckstein, J.: Parallel alternating direction multiplier decomposition of convex programs. Journal of Optimization Theory and Applications, p. 39–62 (1994)
13. Ferreau, H.J., Kirches, C., Potschka, A., Bock, H.G., Diehl, M.: qpOASES: a parametric active-set algorithm for quadratic programming. Mathematical Programming Computation **6**, 327–363 (2014)
14. Fletcher, R., Johnson, T.: On the stability of null-space methods for kkt systems. SIAM Journal on Matrix Analysis and Applications **18**(4), 938–958 (1997)
15. Frank, M., Wolfe, P.: An algorithm for quadratic programming. Naval Research Logistics Quarterly **3**, 95–110 (1956)
16. Frison, G., Diehl, M.: Hpipm: a high-performance quadratic programming framework for model predictive control. IFAC-PapersOnLine **53**(2), 6563–6569 (2020). 21st IFAC World Congress
17. Frison, G., Jørgensen, J.B.: A fast condensing method for solution of linear-quadratic control problems. In: 52nd IEEE Conference on Decision and Control, pp. 7715–7720 (2013). DOI 10.1109/CDC.2013.6761114
18. Frison, G., Kouzupis, D., Jorgensen, J.B., Diehl, M.: An efficient implementation of partial condensing for nonlinear model predictive control. In: 2016 IEEE 55th Conference on Decision and Control, pp. 4457–4462 (2016)
19. Gabay, D., Mercier, B.: A dual algorithm for the solution of nonlinear variational problems via finite element approximation. Computers & Mathematics with Applications **2**(1), 17–40 (1976)
20. Gertz, E.M., Wright, S.J.: Object-oriented software for quadratic programming. ACM Transactions on Mathematical Software **29**(1), 58–81 (2003)
21. Ghadimi, E., Teixeira, A., Shames, I., Johansson, M.: Optimal parameter selection for the alternating direction method of multipliers (admm): Quadratic problems. IEEE Transactions on Automatic Control (2014)
22. Grüne, L., Pannek, J.: Nonlinear Model Predictive Control. Springer (2016)
23. Hermans, B., Themelis, A., Patrinos, P.: QPALM: A Newton-type Proximal Augmented Lagrangian Method for Quadratic Programs. In: 58th IEEE Conference on Decision and Control (2019)
24. Karmarkar, N.: A new polynomial-time algorithm for linear programming. Combinatorica **4**, 373–395 (1984)
25. Kirches, C., Bock, H., Schlöder, J., Sager, S.: Block-structured quadratic programming for the direct multiple shooting method for optimal control. Optimization Methods and Software (2013)
26. Kirches, C., Wirsching, L., Bock, H., Schlöder, J.: Efficient direct multiple shooting for nonlinear model predictive control on long horizons. Journal of Process Control **22**(3), 540–550 (2012)

27. Liao-McPherson, D., Kolmanovsky, I.: Fbstab: A proximally stabilized semismooth algorithm for convex quadratic programming. Automatica **113**, 108801 (2020)
28. Lin, C.J., Moré, J.J.: Newton's method for large bound-constrained optimization problems. SIAM Journal on Optimization **9**(4), 1100–1127 (1999)
29. Løwenstein, K.F., Bernardini, D., Patrinos, P.: Qpalm-ocp: A newton-type proximal augmented lagrangian solver tailored for quadratic programs arising in model predictive control. IEEE Control Systems Letters **8**, 1349–1354 (2024)
30. Maros, I., Mészáros, C.: A repository of convex quadratic programming problems. Optimization Methods and Software **11** (1999)
31. Nocedal, J., Wright, S.J.: Numerical Optimization, 2e edn. Springer, New York, NY, USA (2006)
32. O'Donoghue, B., Chu, E., Parikh, N., Boyd, S.: Conic optimization via operator splitting and homogeneous self-dual embedding. Journal of Optimization Theory and Applications **169**(3), 1042–1068 (2016)
33. Qi L., S.J.: A nonsmooth version of newton's method. Mathematical Programming **58**, 353–367 (1993)
34. Rawlings, J., Mayne, D.: Model Predictive Control: Theory and Design. Nob Hill Publishing, LLC (2009)
35. Ruiz, D.: A scaling algorithm to equilibrate both rows and columns norms in matrices. Technical Report RAL-TR-2001-034 (2001)
36. Stellato, B., Banjac, G., Goulart, P., Bemporad, A., Boyd, S.: OSQP: an operator splitting solver for quadratic programs. Mathematical Programming Computation **12**(4), 637–672 (2020)
37. Themelis, A., Patrinos, P.: Supermann: A superlinearly convergent algorithm for finding fixed points of nonexpansive operators. IEEE Transactions on Automatic Control **64**(12), 4875–4890 (2019)
38. Wang, Y., Boyd, S.: Fast model predictive control using online optimization. Control Systems Technology, IEEE Transactions on **18**, 267 – 278 (2010)
39. Wächter, A., Biegler, L.T.: On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Mathematical Programming **106**, 25–57 (2006)