QP-MPC solvers for real-time control of fast nonlinear systems using inexpensive microcontrollers

Lam, V.T.T.

Award date:
2020

Technische Universiteit Eindhoven (TU/e)

QP-MPC solvers for real-time control of fast nonlinear systems using inexpensive microcontrollers

Graduation MSc thesis

Name: V.T.T. (Victor Trường Thịnh) Lâm
ID: 0857216
S-number: s134462
Department: Electrical Engineering (EE)
Group: Control Systems (CS)
Email: v.t.t.lam@student.tue.nl
Supervisor: dr. M. (Mircea) Lazar

Presented on:
Wednesday July 1, 2020 9:00 – 9:40

Defended on:
Wednesday July 1, 2020 9:40 – 11:40

Graduation Committee:
dr. M. (Mircea) Lazar (supervisor)
prof. dr. P.M.J. (Paul) van den Hof (chairman)
dr. S. (Sofie) Haesaert
dr. D. (Dip) Goswami

Technische Universiteit Eindhoven University of Technology
Where innovation starts

QP-MPC solvers for real-time control of fast nonlinear systems using inexpensive microcontrollers

V.T.T. (Victor Trường Thịnh) Lâm

***Abstract*—In this thesis, we present a new design of a fast quadratic programming (QP)-solver for model predictive control (MPC) using inexpensive microcontrollers. The implementation of MPC is hampered by the need of expensive computational hardware, because high computational power for solving a QP within the sampling period T_s (which is usually small for fast nonlinear systems) as well as high memory capacity is necessary due to large memory footprints. This motivates us to design a new QP-solver that can be executed on an inexpensive microcontroller which is characterized by its limited computational power and memory capacity. The presented QP-solver based on Hildreth's algorithm is deployed on an Arduino Due microcontroller. For illustration purposes, MPC with integral action was used in a real-time reference tracking problem on the magnetic levitator setup, which has fast nonlinear dynamics and requires small sampling period $T_s = 5$ ms. Experimental results show that the designed QP-solver is suitable for real-time control of fast nonlinear systems (with small $T_s = 5$ ms) using inexpensive microcontrollers.**

Keywords: Model Predictive Control, Execution times, Quadratic Programming, Inexpensive Microcontrollers

I. Introduction

Model predictive control (MPC) is a control technique that is very attractive for high-tech applications. It has several advantages in comparison with classical control, such as PID. First, MPC can anticipate for future reference changes. Second, safety or actuator constraints can be handled a priori, by means of solving a quadratic program (QP). Finally, by means of solving a QP, it is doing optimal tracking and therefore has good performance. However, implementation of MPC is hampered by the need of expensive computational hardware. This is due to the need for high computational power for solving a QP within the sampling period T_s (which is small for fast nonlinear systems) as well as high memory capacity.

Recently, there were many efforts to develop MPC design and implementation methods, including optimization solvers for MPC, that can run on inexpensive microcontrollers. Inexpensive microcontrollers are platforms that are characterized by their low cost, which makes them very attractive to apply on a large scale. On the other hand, their relative low memory, low processor speed and limited library functions makes it challenging to apply MPC on fast nonlinear systems that require small sampling time T_s. The biggest challenge in MPC is to solve a QP fast enough, within a small sampling instance T_s.

In the past, MPC design already has been implemented on inexpensive microcontrollers. However, these implementations were very limited due to the limited computation power and memory capacity of inexpensive microcontrollers. First, MPC-QP with only box constraints have been imposed in [1], for which fast-gradient methods can be used. Box constraints are more easy to solve than affine constraints. Second, an Arduino Uno has been used for low level control (such as receiving output y and sending input u to the plant) only, while MPC-QP are solved in other hardware such as a PC with Matlab in [2] or even Matlab simulation on PC only in [3]. Third, offline MPC such as unconstrained MPC [4], [5] or explicit MPC [6] were implemented rather than online MPC which requires solving an MPC-QP and is more challenging. Finally, applications that do not require fast sampling time T_s, such as in process control [7] that has $T_s = 1$ s and $T_s = 3$ s.

Many algorithms have been developed for solving QPs, such as OSQP [8] and active-set QP-solvers qpOASES [9], [10], [11] and Hildreth [12]. A fast gradient QP-solver is qpdunes [13], [14] but affine constraints are not yet supported, only box constraints. In the literature, QP-solvers qpOASES, Hildreth and qpDUNES are well-known. However, it was the case that either these were slow applications (such as process control) that did not require a small T_s, the QP-solver had a big memory footprint, the platform used has high computational power and large memory capacity, or prediction and control horizons were small which makes the QPs to be less complex. In [15], qpOASES has been implemented on a PC, but its code size is 835 kB for prediction horizon $N_p = 12$ and control horizon $N_c = 5$. MPC was executed on programmable logic controllers (PLCs) using QP-solvers qpOASES [16], [17] and Hildreth [18], [19], [20]. In [21], [22] MPC has been conducted on a myRIO from National Instruments composed of both an FPGA and a microcontroller using QP-solver qpOASES. Hardware in the loop (HIL) simulations were performed on an Atmel ARM Cortex-M3 using qpOASES in [23], where in some cases the memory overflows due to a big memory footprint. Relevant details of these applications are summarized and provided in Appendix A. Several observations can be made. First, qpOASES usually demands a quite large memory footprint. Second, qpOASES has been used in slow applications which do not require a small T_s, such as in process control. Third, most of the platforms have high computational power and large memory capacity. Finally, some QPs were less complex due to small prediction or control horizon.

The main challenge in implementing online MPC (by solving a QP with affine constraints) on an Arduino, for example, is the lack of memory, libraries and computational power. This requires careful MPC problem formulation and customized QP solvers for MPC. This also motivates us to design a new QP-solver that can be executed on an inexpensive microcontroller which is characterized by its limited computational power and memory capacity. We choose to design a QP-solver based on Hildreth for a

number of reasons. First, it permits simple implementation in C/C++-code and it is not as memory demanding as qpOASES. Second, due to its simplicity of operations it can be executed fast and efficiently on inexpensive microcontrollers. Third, it is open-source MATLAB code and is publicly accessible [12]. Finally, even though qpOASES' code is also open-source in C++, Hildreth's algorithm is still less complicated then qpOASES', which means that Hildreth is less memory demanding than qpOASES and can be executed fast and efficiently. The designed QP-solver's CPU-time and accuracy will be compared with qpOASES for three reasons: qpOASES is fast, has a high accuracy and is quite popular in the literature as shown above. MATLAB also has default QP-solvers, such as quadprog and mpcqpsolver. Since quadprog's and mpcqpsolver's code are protected, we do not know their algorithms and base our designed QP-solver on them. Although quadprog is very slow, we consider its solution to be optimal and define the absolute value of the error w.r.t. quadprog's solution as a measure of accuracy of a QP-solver, i.e.

$$e_{\text{QP-solver}} = \left| u_{\text{quadprog}} - u_{\text{QP-solver}} \right|, \tag{1}$$

where u_{quadprog} and $u_{\text{QP-solver}}$ denote the MPC control input trajectory computed by quadprog or a QP-solver, respectively, that has been applied to the plant.

This thesis will focus on three main aspects. First, designing a new QP-solver based on Hildreth's QP-solver. The designed QP-solver should be simple enough to run on an inexpensive microcontroller, but fast enough for real-time control of a fast nonlinear system. The QP-solver should also be able to handle stalling situations. Second, formulating an MPC problem and deploying it on an inexpensive microcontroller. We will introduce the concept of constraint horizon N_q in order to reduce complexity of memory footprint and reduce the consumed CPU-time. For robustness, we design and use MPC with integral action as it removes constant offset in reference tracking. The Arduino Due, shown in Fig. 1, based on the Atmel ARM Cortex-M3, is chosen as the microcontroller target, because it has the most attractive specifications among all Arduino's [24], as summarized in Table I.

Table I: Arduino Due specifications

Price	Flash memory	SRAM	CPU speed	Library
€ 20,- – € 40,- (€ 35,- official)	512 kB	96 kB	84 MHz	simple C/C++-library

Finally, conducting experiments on a fast nonlinear system for validation. The magnetic levitator is chosen as the experimental setup, because its dynamics are nonlinear (as we will see later) and its sampling time is small, i.e. $T_s = 5$ ms.

The structure of this thesis is as follows. Brief preliminaries about MPC with integral action and MPC-QPs are discussed in Section II. The considered problem, active set QP-solver design for MPC, is formulated in Section III. In Section IV, we present two new QP-solvers based on Hildreth followed by some analysis. In Section V, we present some aspects about real-time implementation on

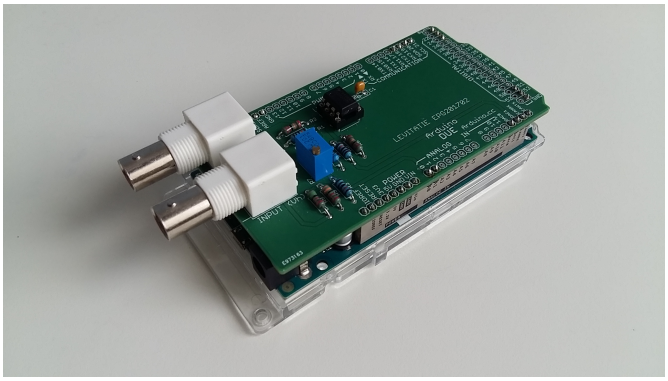


Figure 1: Arduino Due with shield only.

an Arduino. Simulation and experimental results are presented in Section VI. Conclusions and recommendations are summarized in Section VII.

II. Preliminaries

This section provides the necessary knowledge on integral action MPC and formulation of MPC as a QP.

A. Integral action linear MPC with affine term

The derivation of the discrete-time state-space model Σ_d

$$\Sigma_d := \begin{cases} x_p(k+1) & = A_d x_p(k) + B_d u(k) + w_d \\ y(k) & = C_d x_p(k) + D_d u(k) \end{cases}$$

can be found in Appendix D.

For MPC with integral action, we define new variables

$$\begin{aligned} \Delta x_p(k) &:= x_p(k) - x_p(k-1), \\ \Delta y(k+1) &:= y(k+1) - y(k), \\ \Delta u(k) &:= u(k) - u(k-1), \end{aligned}$$

where x_p is state variable of the original discrete-time state-space model. This gives the augmented discrete-time state-space model

$$\begin{aligned} x(k+1) &= \underbrace{\begin{bmatrix} A_d & 0 \\ C_d A_d & I \end{bmatrix}}_{A} x(k) + \underbrace{\begin{bmatrix} B_d \\ C_d B_d \end{bmatrix}}_{B} \Delta u(k) \\ y(k) &= \underbrace{\begin{bmatrix} 0 & I \end{bmatrix}}_{C} x(k), \text{where } x(k) = \begin{bmatrix} \Delta x_p(k) \\ y(k) \end{bmatrix}, \end{aligned}$$

where (A, B) should be controllable and (A, C) should be observable. Note that the affine term w_d cancels out in the augmented state-space model. Next, we define predicted inputs ΔU_k, predicted outputs Y_k and future reference R_k as

$$\begin{aligned} \Delta U_k &:= \begin{bmatrix} \Delta u_{0|k}^T & \cdots & \Delta u_{N-1|k}^T \end{bmatrix}^T, \\ Y_k &:= \begin{bmatrix} y_{1|k}^T & \cdots & y_{N|k}^T \end{bmatrix}^T, \\ \mathcal{R}_k &:= \begin{bmatrix} r_{1|k}^T & \cdots & r_{N|k}^T \end{bmatrix}^T \end{aligned}$$

with the following relation

$$Y_k = \Phi x(k) + \Gamma \Delta U_k, \tag{2}$$

where

$$\Phi = \begin{bmatrix} CA \\ \vdots \\ CA^N \end{bmatrix}, \Gamma = \begin{bmatrix} CB & \cdots & 0 \\ \vdots & \ddots & \vdots \\ CA^{N-1}B & \cdots & CB \end{bmatrix},$$

$x(k) = x_{0|k}$ and N is the prediction horizon, which indicates how many steps ahead we look in the future. In this thesis, the control horizon is equal to the prediction horizon, i.e. $N_c = N$. For some weighting matrices Q, R for stage cost, $P \succ Q$ for terminal cost and future tracking error $e_{i|k} = y_{i|k} - r_{i|k}$ for $i = 0, \dots, N$, we define the MPC cost function as

$$\begin{aligned} J(x(k), \Delta U_k) &= e_{N|k}^T P e_{N|k} + \\ &\sum_{i=0}^{N-1} \left[e_{i|k}^T Q e_{i|k} + \Delta u_{i|k}^T R \Delta u_{i|k} \right] \\ &= e_{0|k}^T Q e_{0|k} + E_k^T \Omega E_k + \Delta U_k^T \Psi \Delta U_k, \end{aligned}$$

where

$$E_k := \begin{bmatrix} e_{1|k}^T & \cdots & e_{N|k}^T \end{bmatrix}^T = Y_k - \mathcal{R}_k,$$

$$\Omega = \begin{bmatrix} Q & & & \\ & Q & & \\ & & \ddots & \\ & & & P \end{bmatrix}, \Psi = \begin{bmatrix} R & & & \\ & R & & \\ & & \ddots & \\ & & & R \end{bmatrix},$$

which drives $e_{i|k}$ and $\Delta u_{i|k}$ to 0 when the steady-state has been reached.

Substituting (2) and removing the constant terms gives

$$\bar{J}(x(k), \Delta U_k) = \frac{1}{2} \Delta U_k^T E \Delta U_k + \Delta U_k^T F, \tag{3}$$

where

$$E = 2\left(\Psi + \Gamma^T \Omega \Gamma\right) \succ 0, F = 2\Gamma^T \Omega \left(\Phi x(k) - \mathcal{R}_k\right).$$

$\bar{J}$ and J have the same unique global minimizer (both are convex since $E \succ 0$), so it suffices to minimize $\bar{J}$. In practice, there are always actuator constraints or safety constraints on the output. In general, constraints could also be imposed on the state vector x, but we consider constraints on Δu, u and y only.

$$\Delta u \in [\Delta u_{\min}, \Delta u_{\max}], u \in [u_{\min}, u_{\max}], y \in [y_{\min}, y_{\max}]$$

We can write these constraints into the form

$$M_i y_{i|k} + Z_i \Delta u_{i|k} \leq b_i, M_N y_{N|k} \leq b_N$$

for $i = 0, \dots, N-1$ and

$$M_i = \begin{bmatrix} 0 \\ 0 \\ -I_q \\ I_q \end{bmatrix}, Z_i = \begin{bmatrix} -I_m \\ I_m \\ 0 \\ 0 \end{bmatrix}, b_i = \begin{bmatrix} -\Delta u_{\min} \\ \Delta u_{\max} \\ -y_{\min} \\ y_{\max} \end{bmatrix},$$

$$M_N = \begin{bmatrix} -I_q \\ I_q \end{bmatrix}, b_N = \begin{bmatrix} -y_{\min} \\ y_{\max} \end{bmatrix},$$

where m and q are dimensions of $\Delta u(k)$ and $y(k)$, respectively. If we recall $y(k) = Cx(k)$, then we can rewrite constraints in compact form as

$$\mathcal{D} x(k) + \mathcal{M} Y_k + \mathcal{E} \Delta U_k \leq \mathcal{C}, \tag{4}$$

where

$$\mathcal{D} = \begin{bmatrix} M_0 C \\ \vdots \\ 0 \end{bmatrix}, \mathcal{C} = \begin{bmatrix} b_0^T \\ \vdots \\ b_N \end{bmatrix},$$

$$\mathcal{M} = \begin{bmatrix} 0 & \cdots & 0 \\ M_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & M_N \end{bmatrix}, \mathcal{E} = \begin{bmatrix} Z_0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & Z_{N-1} \\ 0 & \cdots & 0 \end{bmatrix}.$$

Substituting (2) and using the relation

$$U_k = \mathcal{S} u(k-1) + \mathcal{T} \Delta U_k, \tag{5}$$

where

$$U_k = \begin{bmatrix} u_{0|k} \\ \vdots \\ u_{N-1|k} \end{bmatrix}, \mathcal{S} = \begin{bmatrix} I_m \\ \vdots \\ I_m \end{bmatrix}, \mathcal{T} = \begin{bmatrix} I_m & \cdots & 0 \\ \vdots & \ddots & \vdots \\ I_m & \cdots & I_m \end{bmatrix},$$

and defining

$$U_{\min} = \begin{bmatrix} u_{\min}^T & \cdots & u_{\min}^T \end{bmatrix}^T \in \mathbb{R}^{mN},$$
$$U_{\max} = \begin{bmatrix} u_{\max}^T & \cdots & u_{\max}^T \end{bmatrix}^T \in \mathbb{R}^{mN},$$

we obtain the affine constraints

$$L \Delta U_k \leq c + W x(k) - V u(k-1)$$

with

$$L = \begin{bmatrix} \mathcal{M}\Gamma + \mathcal{E} \\ \mathcal{T} \\ -\mathcal{T} \end{bmatrix}, c = \begin{bmatrix} \mathcal{C} \\ U_{\max} \\ -U_{\min} \end{bmatrix},$$

$$W = \begin{bmatrix} -\mathcal{D} - \mathcal{M}\Phi \\ 0 \\ 0 \end{bmatrix}, V = \begin{bmatrix} 0 \\ \mathcal{S} \\ -\mathcal{S} \end{bmatrix},$$

where $\mathcal{C}$ is defined in (4) and $\mathcal{S}$, $\mathcal{T}$ are defined in (5).

The convex objective function and affine constraint result in a convex QP given as

$$\begin{aligned} \min_{\Delta U_k} \bar{J}(x(k), \Delta U_k) &= \frac{1}{2} \Delta U_k^T E \Delta U_k + \Delta U_k^T F, \\ \text{s.t. } L \Delta U_k &\leq c + W x(k) - V u(k-1) \end{aligned} \tag{6}$$

Finally, an observer is necessary for computing $x(k)$ from measured output $y(k)$ using the following observer model

$$\hat{x}(k+1) = (A - LC)\hat{x}(k) + B\Delta u(k) + L y(k), \tag{7}$$

where $\hat{x}(k)$ is the estimated state. Defining the estimation error

$$e(k) := x(k) - \hat{x}(k),$$

the estimation error dynamics is derived as

$$e(k+1) = (A - LC) e(k).$$

Observer gain matrix L is chosen in such a way that $|\lambda(A - LC)| < 1$ and $e(k)$ converges to 0 as fast as possible.

Finally, the receding horizon principle is applied as follows. First, output $y(k)$ is measured and state $\hat{x}(k)$ is estimated using the observer. Second, the MPC-QP (6) is solved to obtain ΔU_k^*. Finally, from ΔU_k^* the first element $\Delta u(k) = \Delta u_{0|k}^*$ is used for computing the next control input

$$u(k) = \Delta u(k) + u(k-1)$$

and repeat.

B. MPC-QP

In general, we can formulate the MPC-QP (6) in the following form

$$\min_{\theta} J(\theta) = \frac{1}{2}\theta^T E\theta + \theta^T F, \quad \text{s.t. } M\theta \leq \gamma, \tag{8}$$

where the decision variable is $\Delta U_k \rightarrow \theta$. For simplicity, we consider control input $u \in \mathbb{R}$ and measured output $y \in \mathbb{R}$ are scalars, i.e. $q = m = 1$. The objective function $J(\theta) \in \mathbb{R}$ is quadratic in the decision variable $\theta \in \mathbb{R}^N$ and the constraints are affine in θ. The matrices $E \in \mathbb{R}^{N\times N}$ and $M \in \mathbb{R}^{p\times N}$ are constant, while vectors $F \in \mathbb{R}^N$ and $\gamma \in \mathbb{R}^p$ are depending on state $x(k)$, future reference $\mathcal{R}_k$ or previous control input $u(k-1)$. The total number of constraints is denoted as p and the active set is denoted by $\mathbb{A}$ which consists of indices of active constraints. The number of active constraints is denoted as $c \leq \frac{p}{2}$, since the constraints equally consist of lower and upper bounds and the lower and upper bounds cannot be active simultaneously. It can be seen that complexity in terms of memory footprint (for storage) and CPU-time (number of computations) depends on prediction horizon N as a kind of a tuning knob.

III. Problem formulation

This section consists of two parts. First, we formulate the objectives of this thesis. Second, we describe the active-set solver Hildreth, on which we will base our new QP-solver. We also briefly describe the active-set QP-solver qpOASES.

A. Fast QP-solver design for Arduino

The problem formulation can be expressed in two objectives, each with its challenges. As a first objective, we want to design a QP-solver which should be simple enough to run on an Arduino, but fast enough for real-time control of a fast nonlinear system. Our new QP-solver will be based on the Hildreth QP-solver.

Our second objective is to make the MPC problem suitable for real-time control on an Arduino. This could be done by solving a few subproblems. First, complexity in terms of memory footprint and CPU-time, should be low and it should be possible influence it by using some additional parameter as a tuning knob. Second, an easy workflow from Matlab design to Arduino real-time deployment is necessary. Finally, the designed QP-solver should prevent stalling of the solver. Stalling cases are for example, that it takes too long to solve a QP, i.e. CPU-time $> T_s$ or there might be numerical problems such as `NaN` numbers.

B. Previous QP-solvers: Hildreth and qpOASES

Here, we will compare active-set QP-solvers Hildreth and qpOASES and motivate why we design a new QP-solver based on Hildreth. The designed QP-solver will be compared with qpOASES for reasons already mentioned in the introduction.

1) Hildreth: The Hildreth QP-solver takes as input the matrices E, M and vectors F, γ from QP (8) and outputs the constrained solution θ. Its algorithm is as follows (details can be found in [12]):

1. Unconstrained solution: $\theta = -E\backslash F$
2. Check $M\theta > \gamma$ in a scalar-based `for`-loop, `if` all constraints are satisfied `then` stop `else` continue `end`
3. Compute $H = M(E\backslash M^T)$, $K = M(E\backslash F + \gamma)$ and initialize Lagrange multiplier $\lambda^m = 0$ (cold-start). Set $m = 0$
4. Update λ^m in a scalar-based `for`-loop, with

$$w_i^m = -\frac{1}{h_{ii}}\left[k_i + \sum_{j=1}^{i-1} h_{ij}\lambda_j^m + \sum_{j=i+1}^{n} h_{ij}\lambda_j^{m-1}\right]$$
$$\lambda_i^m = \max(0, w_i^m)$$

   `if` $\left(\lambda^m - \lambda^{m-1}\right)^T\left(\lambda^m - \lambda^{m-1}\right) \leq \delta$ or $m \geq \bar{m}$ `then` stop `else` continue `end`
5. Return $\theta = -E\backslash F - E\backslash M^T\lambda^m$

where tolerance $\delta = 1 \cdot 10^{-7}$ and maximum number of iteration $\bar{m} = 38$. Note that the backslash operator ($\backslash$) is a way to solve a system of linear equations such as $Ax = b$, i.e. $x = A\backslash b$.

2) qpOASES: The qpOASES QP-solver parametrises vectors F and γ from QP (8) as affine functions of parameter w and outputs the constrained solution θ. Its algorithm is as follows (details can be found in [9], [10], [11]):

1. Compute updates Δw, ΔF, $\Delta\gamma$
2. Compute primal and dual step directions $\Delta\theta$ and $\Delta\lambda$
3. Maximum homotopy step length $\tau_{\max}$ is determined
4. Variables w, θ, λ are updated using $\tau_{\max}$
5. `if` $\tau_{\max} = 1$ `then` solution θ found and stop `else` based on $\tau_{\max} \in [0, 1)$ add or remove constraint from the active set $\mathbb{A}$ and continue with Step 1 `end`

3) Comparison: The main differences between Hildreth and qpOASES are summarized and provided in Table II

Table II: Differences between Hildreth and qpOASES

Hildreth	qpOASES
Scalar operations and divisions, no matrix inversion, little matrix-vector operations	Many matrix-vector operations
Simple and basic mathematical operations based on KKT-conditions	Based on KKT-conditions. Matrix updates using advanced mathematical operations such as: Cholesky factorization, TQ factorization, Givens plane rotation
Works with E, F, M, γ from (8) directly	Vectors F and γ from (8) are parametrised as affine functions of parameter w
Numerical convergence to λ	Based on current active set $\mathbb{A}$, λ is found exactly
Active set $\mathbb{A}$ is found as soon as λ converged	Converges to optimal active set $\mathbb{A}$ by adding and removing constraints to active set $\mathbb{A}$

The three main reasons why we will base our designed QP-solver on Hildreth are the following. First, Hildreth uses basic mathematical operations, while qpOASES uses advanced ones that result in a large memory footprint. Second, Hildreth involves little matrix-vector operations and performs scalar-based operations such as division. This avoids matrix inversions and thus avoids numerical degeneracy problems in case of singularity. Finally, Hildreth works directly with matrices and vectors from (8), while qpOASES parametrises the vectors from (8) as affine functions of parameter w.

IV. Designed QP-solvers

In this section, we present two designed Hildreth-based QP-solvers. Hildreth+' is presented briefly, while HildrethAct is discussed more in depth including convergence and complexity analysis in terms of memory footprint and computational complexity. For comparison, complexity of qpOASES will be analysed as well. The concept of constraint horizon N_q will also be introduced in this section.

A. Hildreth+'

When running Hildreth algorithm on the Arduino, we noticed it exceeded sampling period T_s severely, which makes it unsuitable for running on the Arduino. Hence, we made a few modifications to Hildreth's QP-solver which we will call Hildreth+'. It is still very similar to the original Hildreth algorithm. It takes as input the matrices $E^{-1}, M, H = ME^{-1}M^T$ and vectors F, γ from QP (8) and outputs the constrained solution θ. Constant matrices E^{-1} and H are precomputed offline. Its step-by-step algorithm is provided in Appendix B only, rather than its pseudo code, in order to give a clear overview of the mathematics (details can be found in [25].

Some key differences with respect to the original Hildreth are given in Table III.

Table III: Difference between Hildreth+' and Hildreth

Step	Hildreth+'	Hildreth
2	Vectorized constraint check	Scalar-based for-loop constraint check
3, 6	Warm-start λ	Cold-start λ
4	Better rate of convergence (ROC) using SOR [26], inspired by Jacobi and Gauss-Seidel method [27]	Usual ROC
1, 3, 5	Precomputing constants $H = L + D + U, E^{-1}$ offline, where L, U denote strict lower-, upper-triangular matrices and D denotes the diagonal matrix.	Computing H online and solves backslash operator on E online
4	Subtracts $\lambda^m - \lambda^{m-1}$ once and reuses it	Subtracts $\lambda^m - \lambda^{m-1}$ twice
3	Reduced dimensions $\tilde{\lambda}$, $\tilde{H}$ and $\tilde{K}$ by extracting indices of non-violated constraints that may become active	Full dimensions λ, H and K
5	Reuse of θ from Step 1	Computes $-E\backslash F$ again
5	Reduced dimension $\tilde{M}$	Full dimension M

B. HildrethAct

When running Hildreth+' on the Arduino, sampling period T_s was still exceeded but less severe than Hildreth. However, Hildreth+' is still unsuitable for running on the Arduino. Hence, we designed a new QP-solver which we will call HildrethAct. Although HildrethAct is very different from the original Hildreth algorithm, it does still rely on the key KKT-conditions. It takes a input the matrices $E^{-1}, M, H = ME^{-1}M^T$ and vectors F, γ from QP (8) and outputs the constrained solution θ. Constant matrices E^{-1} and H are precomputed offline. Its step-by-step algorithm is provided in Appendix C only, rather than its pseudo code, in order to give a clear overview of the mathematics. In what follows, we refer to the steps in this step-by-step algorithm of HildrethAct.

This QP-solver has numerous strengths with respect to the original Hildreth algorithm:

1. The QP-solver HildrethAct only terminates when optimality (condition Step 5.d.) and feasibility (condition Step 5) are reached or when `max_iter` is reached.
2. When the number of violated constraints is
   - 0, then no constraints are active. Unconstrained solution is used with $|\mathbb{A}| = 0$, $\lambda = \varnothing$ and $H_{\text{act}}^{-1} = \varnothing$,
   - 1, then this constraint must be active. Constrained solution is used with $|\mathbb{A}| = 1$ and $\lambda, H_{\text{act}}^{-1} \in \mathbb{R}$,
   - ≥ 2, then some of these constraints could be active. Constrained solution is used with $|\mathbb{A}| = c$, $\lambda \in \mathbb{R}^c$ and $H_{\text{act}}^{-1} \in \mathbb{R}^{c\times c}$, where c denotes the number of active constraints, $\mathbb{A}$ denotes the active set, $|\mathbb{A}| = c$ denotes the cardinality (number of elements) of active set $\mathbb{A}$, λ_{act} denotes the Lagrange multiplier corresponding to $\mathbb{A}$, H_{act}^{-1} denotes the matrix inverse of H corresponding to $\mathbb{A}$.
3. Ideally, we would like to know the active set $\mathbb{A}$ a priori. Then it is easy to compute
   $$\lambda_{\text{act}} = -H_{\text{act}}^{-1}K_{\text{act}}$$
   with matrix $H_{\text{act}} = M_{\text{act}}E^{-1}M_{\text{act}}^T$ and vector $K = \gamma_{\text{act}} + M_{\text{act}}E^{-1}F$. HildrethAct combines the KKT-conditions from Hildreth together with the constraint adding/removing principle from qpOASES which will converge to the optimal active set.
4. Using additive and subtractive matrix updates (its pseudo-code will be omitted due to space limitations, but it is a more efficient implementation of [28] optimized for Arduino Due) are used for updating H_{act}^{-1} and λ_{act}. This avoids direct matrix inversions and thus avoids numerical degeneracy problems in case of singularity.
5. Constant matrices H and E^{-1} are precomputed offline.
6. K_0 from Step 3 is reused in Step 5.g and θ_0 from Step 2 is reused in Step 6.
7. Let p denote the total number of constraints, then the number of active constraints is $c \leq \frac{p}{2}$, because upper and lower bounds cannot be active simultaneously.
8. Like Hildreth, we keep the cold-starting feature, because warm-starting is disadvantageous when the active set changes frequently due to disturbances in the experiment.

C. Convergence analysis HildrethAct

In this section, we will analyse the QP-solver HildrethAct and show it will always converge to a feasible and optimal solution without cycling. In what follows, we refer to the steps of the step-by-step HildrethAct algorithm given in Appendix C

First, suppose `max_iter = infinity`, i.e. we do not have time constraints (such as sampling time T_s) in the ideal case, then the algorithm only terminates when constrained solution θ is feasible, which is ensured by the termination condition of the `while`-loop in Step 5 to be $K_{\min} \geq 0$, i.e. K does not contain negative elements (no constraints are violated anymore). Likewise, the algorithm only terminates when constrained solution θ is optimal, which is ensured by the termination condition of the `while`-loop in Step 5.d to be $\lambda_{\min} \geq 0$, i.e. λ_{act} does not

contain negative elements (no active constraints became inactive).

Second, consider the following scenario. Constraint z is added to the active set $\mathbb{A}$ and $|\mathbb{A}| = c = 1$, because constraint z is violated the most (corresponding to $K_{\min} < 0$). Later, we also add constraint y to the active set $\mathbb{A}$ for the same reasoning and $|\mathbb{A}| = c = 2$. When we compute λ_{act}, we observe that there are negative elements in λ_{act} which indicate some constraints in the active set $\mathbb{A}$ became inactive. This is only possible when active set $\mathbb{A}$ contains 2 or more elements, i.e. $c \geq 2$. Suppose active constraint y covers the active constraint z (z is a subset of y, i.e. $z \subset y$), which makes constraint z the most redundant (corresponding to $\lambda_{\min} < 0$). Constraint z is then removed from the active set $\mathbb{A}$ and is inactive now. Since constraint y (which also covers constraint z) is still in active set $\mathbb{A}$, both constraints y and z can not be violated anymore and will not be added to the active set $\mathbb{A}$ again, thus preventing cycling. Thus, HildrethAct converges to the feasible and optimal solution in a finite number of iterations.

Finally, the QPs we consider are always bounded as $E \succ 0$ resulting in a convex objective function. Together with affine constraints, the QPs will be convex which guarantees a global unique minimum.

D. Constraint horizon N_q

We would like to introduce the concept of constraint horizon N_q, where we put only upper and lower bounds on stage variables $y_{i|k}$, $u_{i|k}$ and $\Delta u_{i|k}$ for $i = 0, \dots, N_q - 1$ (giving $6N_q$ constraints) and upper and lower bounds on terminal variable $y_{N_q|k}$ (giving 2 constraints), which gives us a total number of constraints

$$p = 6N_q + 2.$$

The constraint horizon N_q can be bounded in between $N_{\min} \leq N_q \leq N$, where N is the used prediction horizon and $N_{\min}$ is the lowest prediction horizon that resulted in a stable and feasible system. In this way complexity, in terms of memory footprint and computational complexity, of the QP can be reduced. This permits us to increase $\uparrow N$ and reduce $\downarrow T_s$ for better tracking performance. This alternative is more computationally efficient than move-blocking, where $N_q = N$ and additional equality constraints are introduced, which increases the QP's complexity.

E. Memory footprint analysis HildrethAct and qpOASES

In this section, we compare memory footprints of HildrethAct and qpOASES, in terms of how many numbers need to be stored. We will omit the minor contributions of scalar numbers.

1) HildrethAct: The memory footprint of HildrethAct is summarized in Table IV

Table IV: Memory footprint HildrethAct

	Input data	Internal data
Data	$E^{-1} \in \mathbb{R}^{N\times N}$, $F \in \mathbb{R}^N$, $M \in \mathbb{R}^{p\times N}$, $H \in \mathbb{R}^{p\times p}$, $\gamma \in \mathbb{R}^p$	$\theta, \theta_0 \in \mathbb{R}^N$, $K, K_0 \in \mathbb{R}^p$, $H^{-1}_{\text{act}} \in \mathbb{R}^{w\times w}$, $\lambda_{\text{act}}, \mathbb{A} \in \mathbb{R}^w$,
General	$N^2 + N + pN + p^2 + p$	$2N + 2p + w^2 + 2w$
In terms of N, N_q	$N^2 + N(6N_q + 3) + 36N_q^2 + 30N_q + 6$	$2N + 9N_q^2 + 24N_q + 7$

where N is the prediction horizon, N_q is the constraint horizon such that $N_{\min} \leq N_q \leq N$, the total number of constraints is denoted as $p = 6N_q + 2$ and $w = \frac{p}{2}$, i.e. at most half of the number of constraints are in the active set, because upper and lower bounds can not be active simultaneously.

Hence, a total of

$$N^2 + N(6N_q + 5) + 45N_q^2 + 54N_q + 13$$

numbers need to be stored. When $N_q \ll N$, we can see that the memory footprint is significantly reduced, as the terms

$$N^2 + N(6N_q + 5) + 13$$

dominate.

2) qpOASES: The memory footprint of qpOASES is summarized in Table V

Table V: Memory footprint qpOASES

	Input data	Internal data
Data	$E \in \mathbb{R}^{N\times N}$, $F \in \mathbb{R}^N$, $\gamma \in \mathbb{R}^p$, $M \in \mathbb{R}^{p\times N}$	$\theta, \Delta\theta \in \mathbb{R}^N$, $\lambda, \Delta\lambda, \mathbb{A} \in \mathbb{R}^p$, $R, T, Q \in \mathbb{R}^{N\times N}$
General	$N^2 + N + p + pN$	$3N^2 + 2N + 3p$
In terms of N, N_q	$N^2 + N(6N_q + 3) + 6N_q + 2$	$3N^2 + 2N + 18N_q + 6$

where R denotes the Cholesky factorization and T, Q denote TQ-factorization (modification of QR-factorization).

Hence, a total of

$$4N^2 + N(6N_q + 5) + 24N_q + 8$$

numbers need to be stored. When $N_q \ll N$, we can see that the memory footprint is significantly reduced, as the terms

$$4N^2 + N(6N_q + 5) + 8$$

dominate.

3) Comparison: We can conclude that when $N_q \ll N$, HildrethAct has a smaller memory footprint for data storage than qpOASES. It is of importance to note that N and N_q are the tuning knobs for memory footprint.

F. Computational complexity analysis HildrethAct and qpOASES

In this section, we compare computational complexity of HildrethAct and qpOASES, in terms of number of arithmetic operations, such as additions, subtractions and multiplications.

1) HildrethAct: The total number of arithmetic operations has been found as

$$\begin{aligned} &4N^2 + N(2c - 2) + p(2N + 1) + 2 + \\ &(\alpha - 1)\left[3(c-1)^2 + 6(c-1) + 5 + c + p(2c+1)\right] + \\ &\beta\left[2c^2 + 7c + 2\right] \end{aligned}$$

where N is the prediction horizon, N_q is the constraint horizon such that $N_{\min} \leq N_q \leq N$, the total number of constraints is denoted as $p = 6N_q + 2$ and $c \leq \frac{p}{2}$ denotes the number of active constraints (lower and upper bounds cannot be active simultaeously), $\alpha > 0$ denotes how many times constraints are added to active set $\mathbb{A}$ (Step 5) and $\beta \geq 0$ denotes how many times constraints are removed

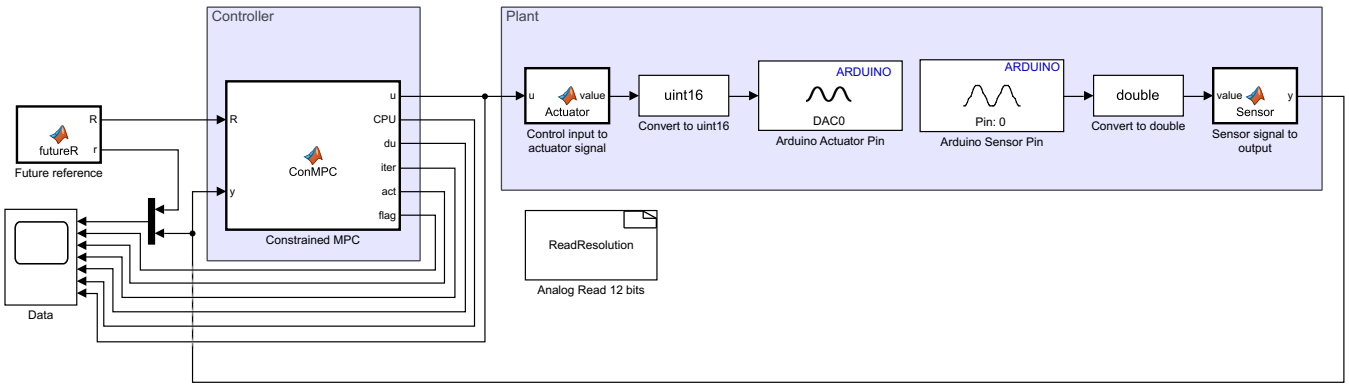


Figure 2: Simulink diagram for real-time experiments.

from active set $\mathbb{A}$ (Step 5.d). These steps refer to the step-by-step algorithm of HildrethAct in Appendix C.

It should be noted that N, p, α and β are fixed, while c is not. Furthermore, N, p are know a priori, while α, β, c are not. Since $N, c, p > 0$ and $c, N < p$, the following terms are dominating

$$4N^2 + p(2N+1) + (\alpha - 1)\left[3(c-1)^2\right] + 2\beta c^2.$$

2) qpOASES: The total number of arithmetic operations has been found as

$$\begin{aligned} &5N^2 - 2Nc + 2c^2 + N(p-c) + \\ &\alpha\left[10N^2 - 6Nc + 2c^2 + N(p-c)\right] + \\ &\beta\left[7\frac{1}{2}N^2 - 2\frac{1}{2}Nc + 2\frac{7}{8}c^2 + N(p-c)\right]. \end{aligned}$$

Since $N, c, p > 0$ and $c, N < p$, the following terms are dominating

$$5N^2 + 2c^2 + \alpha\left[10N^2 + 2c^2\right] + \beta\left[7\frac{1}{2}N^2 + 2\frac{7}{8}c^2\right].$$

3) Comparison: HildrethAct is less computational complex and has lower CPU-time than qpOASES, when we consider the N^2 in the dominating terms. However, since usually $p \gg c$, then HildrethAct becomes slower the more constraints are active as its computational complexity depends on

$$p(2N+1) + \alpha p(2c+1) + 2\beta c^2,$$

while qpOASES becomes faster the more constraints are active as its computational complexity depends on

$$N(p-c) + \alpha N(p-c) + \beta N(p-c).$$

Again, it is of importance to note that N and N_q are the tuning knobs for computational complexity.

V. Real-time implementation on Arduino

In this section, we discuss how to make real-time implementation of the MPC problem suitable for an Arduino. First, we present the workflow from designing in Matlab to running an experiment on the Arduino. Finally, we present a method to prevent the QP-solver to stall during real-time experiments, followed by other considerations.

A. Workflow

The Simulink diagram used in the performed experiments is shown in Fig. 2. The model must run in External Mode (`Monitor & Tune`) and the Arduino Due should be connected to a PC. In- and output pin blocks from the *Arduino Support Package for Simulink and Matlab* are necessary for assigning pins to variables such as control input u and measured output y. Data type conversion of type `uint16` and `double` are necessary for Arduino specific reasons.

The workflow that starts from Matlab-script and ends up in real-time experiment control is given as:

1. Write `m`-function in Matlab for
   - constrained MPC with integral action
   - future reference generator
   - mapping from control input u to actuator value
   - mapping from sensor value to output y
2. Connect Arduino Due to PC via USB-cable
3. Call the `m`-functions in *User-defined MATLAB-function blocks* in Simulink
4. Connect the blocks to in- and output pin blocks from the *Arduino Suppose Package for Simulink and Matlab*
5. Add an *S-function Builder Block* for reading 12 bits analog signals and select Arduino Due as hardware
6. Click `Monitor & Tune` (External Mode) to deploy generated C-code to Arduino board and the setup can be controlled in real-time standalone on the board. Sensor and actuator data can be viewed in real-time and logged via Scope.

Overrun is defined as

$$\text{CPU-time} > T_s$$

where T_s is the sampling period. In Simulink, the option `Overrun detection` can be enabled where a built-in LED (usually on pin 13) will light up when overrun occurs.

In Matlab, CPU-time can be measured as:

Script 1: CPU-time measurement in Matlab

```
startTime = tic;

% Code for measurement

elapsed = toc(startTime);
```

Similarly, we can measure CPU-time on an Arduino using the following code in a Matlab-function:

Script 2: CPU-time measurement in Arduino

```
startTime  = uint32(0);
finishTime = uint32(0);

if (coder.target('rtw'))
        startTime = coder.ceval('micros');
end

% Code for measurement

if (coder.target('rtw'))
        finishTime = coder.ceval('micros');
end

elapsed = finishTime - startTime;
```

which can be called in a Simulink-block. The Simulink-model can be converted into C-code and deployed on an Arduino.

After converting the Simulink-model into C-code and deploying to Arduino, the *Diagnostic Viewer*-window pops up which will display the memory footprint of the deployed C-code.

The reference trajectory is designed in advance and known a priori. However, we cannot store all samples as it would require a huge memory footprint or even overflow the memory. Rather, we store only the values of each piecewise constant part and compute $\mathcal{R}_k$ based on these values. Furthermore, constants such as $2\Gamma^T\Gamma\Phi$, $2\Gamma^T\Omega$ in (3) and $(A-LC)$ in (7) can be precomputed offline to save CPU-time.

Details about the blocks in Fig. 2 such as future reference generation (`futureR`), constrained MPC with integral action algorithm (`ConMPC`), sensor mapping (`Sensor`) and actuator mapping (`Actuator`) are as step-by-step algorithms provided in Appendix E, rather than pseudo-code in order to have a clear overview of the mathematics. These functions were implemented as efficiently as possible and optimized for the Arduino Due, but these details are omitted due to space limitations.

B. Stalling prevention

There are two causes why a QP-solver stalls. The first cause is constraint infeasibility, when the feasible set defined by affine constraints $M\theta \leq \gamma$ is empty. The second cause is time infeasibility, when the QP-solver has not finished within the sampling period T_s, i.e. CPU-time $> T_s$.

1) Constraint infeasibility: A QP-problem is constraint infeasible when the feasible set described by the polytope defined by the affine constraints $M\theta \leq \gamma$ is empty, i.e. it has zero vertices. There is a Matlab-function `plotregion` [29] which contains an algorithm for vertex enumeration and can compute the vertices of a feasible set, however it works only for 3D/2D feasible sets. For higher dimensions, this is a NP-hard problem [30]. Another reason why we do not compute these vertices, if possible, is that it may take a while to compute, which is challenging for small T_s.

2) Time infeasibility: Especially for small T_s, it is challenging to achieve CPU-time $< T_s$. The complexity of the QP could be reduced by removing redundant constraints. Redundant constraints are constraints that do not change the feasible set when they are removed. Note they are different than inactive constraints, because while they do not change the optimal solution when they are removed, but they may not be redundant. However, it still may take a while to compute the redundant constraints and again could take longer than the permitted sampling time T_s. Another reason why we do not compute the redundant constraints, is that we want to keep the affine constraints $M\theta \leq \gamma$ as generalized as possible.

Table VI: Flag-system: meanings, causes and actions

Flag	Meaning	Cause	Action
0	QP-solver fully converged to feasible and optimal solution	QP is feasible and CPU-time $< T_s$	Apply solution $u(k)$ to plant
1	λ_{act} in QP-solver is `NaN` or ∞	QP is infeasible	Compute unconstrained solution $\theta_0 = -E^{-1}F$ with saturation such that $\Delta u \in [\Delta u_{\min}, \Delta u_{\max}]$ and $u \in [u_{\min}, u_{\max}]$
2	Solution of QP-solver is sub-optimal and is infeasible (does not satisfy $\Delta u \in [\Delta u_{\min}, \Delta u_{\max}]$ and $u \in [u_{\min}, u_{\max}]$)	QP-solver terminated too early, because `max_iter` is reached	Saturate sub-optimal solution such that $\Delta u \in [\Delta u_{\min}, \Delta u_{\max}]$ and $u \in [u_{\min}, u_{\max}]$
3	Solution of QP-solver is sub-optimal but is feasible (satisfies $\Delta u \in [\Delta u_{\min}, \Delta u_{\max}]$ and $u \in [u_{\min}, u_{\max}]$)	QP-solver terminated too early, because `max_iter` is reached	Apply solution $u(k)$ to plant

3) Flag-system: Hence, as an alternative solution for handling stalling problem, we opted for using a flag system, which has been implemented for the HildrethAct step-by-step algorithm in Appendix C (Steps 5.d, 5.e and 5.f) and `ConMPC`-block step-by-step algorithm provided in Appendix E. The meanings, causes and actions for the flag-system are summarized in Table VI, where HildrethAct is meant by 'QP-solver' and constraint infeasibility is meant by 'infeasible'. Using the flag-system, both causes of QP-solver stalling are solved. The first cause, constraint infeasibility is prevented by checking whether λ_{act} in QP-solver is NaN or ∞. The second cause, time infeasibility (CPU-time $> T_s$) is handled by tuning `max_iter` such that CPU-time $< 0.9T_s$. But we have to make sure that computing the flags with `if`-, `else`-statements and computing the unconstrained saturated solution, as provided in Appendix C and Appendix E, must be done within $0.1T_s$, to make sure that the total CPU-time $< T_s$.

C. Other considerations

Computing the unconstrained solution and the constrained solution, in Step 1 and Step 5 respectively of the original Hildreth, are equivalent to solving $\frac{\partial \bar{J}(\theta)}{\partial \theta} = 0$ (with cost function $\bar{J}$) and the KKT equation

$$E\theta = -F, E\theta = -F - M^T\lambda^m$$

resepctively, for $\theta \in \mathbb{R}^N$, where N is the prediction horizon. Hence, we are solving a system of linear equations with N unknowns in N linear equations. To keep it as

a more general problem, suppose we want to solve the system of linear equations

$$Ax = b$$

for $x \in \mathbb{R}^n$, where $n \in \mathbb{N}$, which consists of n unknowns in n linear equations. There are three approaches for solving this problem, which are summarized in the following and we draw a few conclusions on these approaches.

1) Approach 1: Offline inverse, where we precompute offline `invA = inv(A)`, then `x = invA*b`.

This is the fastest approach as tested in Matlab and on the Arduino Due. It resulted in good tracking performance, because CPU-time $< T_s$, which as indicated by the built-in LED not lighting up. However, this approach is only suitable for when $A \in \mathbb{R}^{n\times n}$ is a constant. This is the case for linear MPC, where linearization is done once around an equilibrium point. The hessian $E \in \mathbb{R}^{N\times N}$ is therefore static, with prediction horizon N. This approach is therefore also applied to Hildreth+' (see Appendix B) and HildrethAct (see Appendix C).

2) Approach 2: Offline LU-factorization, where we precompute offline $LU = A$ as `[L,U,~] = lu(A,'vector')` and solve $L(Ux) = b$ using Ly (forward substitution) and $Ux = y$ (backward substitution). The substitutions are fast since L is lower-triangular with 1's in diagonal and U is upper-triangular.

This approach is bit slower than the first one as tested in Matlab and on the Arduino Due. It resulted in poor tracking performance, because CPU-time $> T_s$ as indicated by the built-in LED lighting up. However, this is the only option when A is varying. This is the case for MPC algorithms based on time-varying linear models, where hessian E is changing every time.

3) Approach 3: Solve system of equations online, where we solve $Ax = b$ online using `x = linsolve(A,b);`, `x = A\b;` or `x = mldivide(A,b);`.

This approach succeeded in Matlab, but failed during compiling to C/C++-code and uploading to the Arduino Due. This is due to `x = mldivide(A,b)` and `x = A\b` requiring the `mldivide`-library function and `x = linsolve(A,b)` requiring the `linsolve`-library function, which were both not found for the Arduino Due.

We can summarize the findings as follows. First, offline LU-factorization is slower than offline inverse computation as verified on Matlab and Arduno Due. Second, online LU-factorization, however, is faster than online inverse computation as verified on Matlab.

VI. Experimental validation on a magnetic levitator setup

Experimental validation will be done on a magnetic levitator setup as it has fast nonlinear dynamics, with sampling period chosen equal to $T_s = 5$ ms. Its dynamics are described by the continuous-time nonlinear differential equation given as

$$m\ddot{y} = \frac{c_m u + p_m}{(a-d)-y} - mg \tag{9}$$

with coil current as control input u and vertical ball position as measured output y. The nonlinear differential equation (9) can be linearized around an operating point $(y_0, u_0) = (0.0225, 1.23)$ and can be rewritten to the continuous-time state-space model Σ_c with affine term w_c, with state $x = \begin{bmatrix} y & v \end{bmatrix}^T$, where v is the vertical ball velocity. The derivation of the discrete-time state-space model Σ_d with affine term w_d follows the approach summarized in Appendix D. This discrete-time state-space model is then used for integral action MPC.

First, we present simulation results performed in Matlab on a PC. The results show accuracy with respect to quadprog among the QP-solvers qpOASES, Hildreth and HildrethAct. Simulation results of Hildreth+' on a different application can be found in [25]. QP-solvers Hildreth and HildrethAct were designed in Matlab `m`-scripts, but are converted into MEX-files for fair comparison with qpOASES, which also is a MEX-file. However, quadprog is not converted into a MEX-file, because its code is protected and we regard it as a black-box. MEX-files can link precompiled C/C++-code to a Matlab-function that can be called inside Matlab. Hence compiling time is excluded, because it does not need to recompile Matlab `m`-code to C/C++ every time. Since Arduino uses precompiled C/C++-code as well, this gives a realistic result, only the different clock frequencies of the PC and Arduino play a role.

Second, we will use the HildrethAct QP-solver on the magnetic levitator experimental setup using an Arduino Due microcontroller. Here, we present reference tracking, flag-system, computational complexity in terms of CPU-time and memory footprint of HildrethAct on the Arduino Due. The PC is only used for data logging and for compiling Simulink-code into C/C++-code that is deployed on the Arduino Due. The QP-solver HildrethAct runs standalone on the Arduino Due. We did not run qpOASES on the Arduino, because it had a big memory footprint (or even memory overflow) and required large sampling period T_s as seen in the literature study in the introduction, such as [21], [22], [23].

A. Simulations

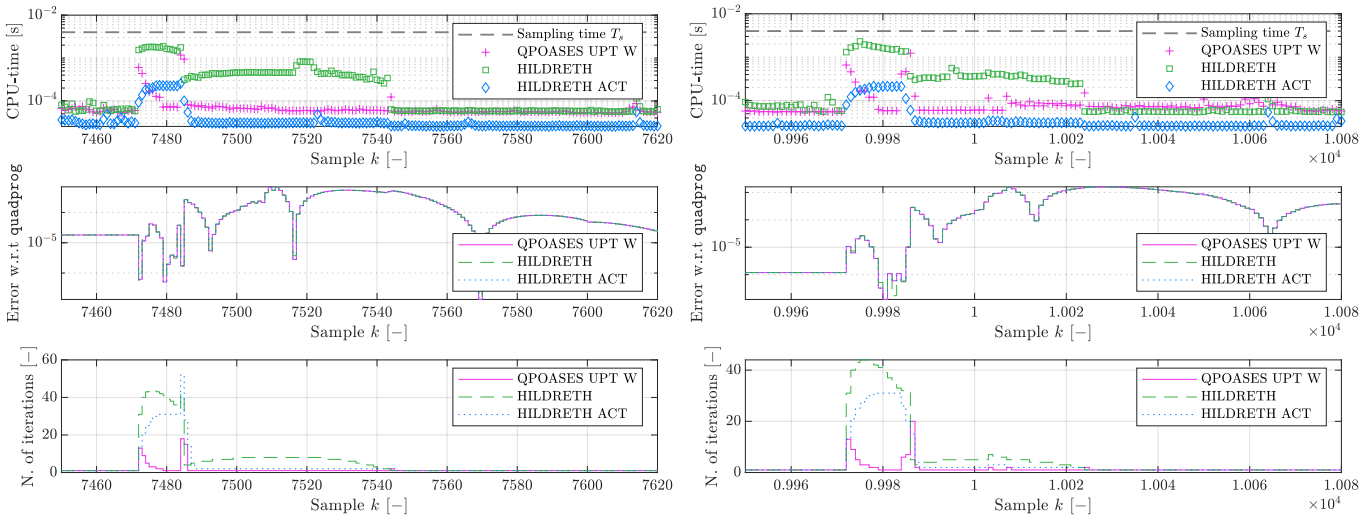
The simulations are performed in Matlab on an Acer Aspire 5 laptop with 8.00 GB RAM (7.66 GB available) and processor clock speed of 1.99 GHz (@1.80 GHz). Since the Arduino Due has clock frequency of 84 MHz, there is a factor difference of around 21× in the CPU-time of the laptop in simulation.

The simulation lasts for 70 s and with sampling time $T_s = 4$ ms this is equivalent to 17500 samples. The reference is a square wave with amplitude of 4 mm around $y_0 = 0.0225$ m and each level lasts for 10 seconds.

1) Results: The CPU-time, accuracy in terms of the error defined in (1) with respect to quadprog and number of iterations are presented in Fig. 3, zoomed in for

Table VII: MPC parameters for simulation

Parameter	Value	Unit
Stage cost weight Q	1	[–]
Stage cost weight R	0.02	[–]
Terminal cost weight P	$2Q$	[–]
Prediction horizon N	30	[–]
Constraint horizon N_q	30	[–]
Sampling period T_s	4	[ms]
$[\Delta u_{\min}, \Delta u_{\max}]$	$[-0.025, 0.025]$	[A]
$[u_{\min}, u_{\max}]$	$[0.73, 1.73]$	[A]
$[y_{\min}, y_{\max}]$	$[0.018, 0.028]$	[m]
`max_iter`	∞	[–]
Reference $[r_{\min}, r_{\max}]$	$[0.0185, 0.0265]$	[m]



(a) Sample interval $[7450, 7620]$ [–] **(b)** Sample interval $[9950, 10080]$ [–]

Figure 3: Simulation results.

Table VIII: Average, maximum, minimum CPU-time and accuracy of QP-solvers qpoases, Hildreth, HildrethAct

QP-solver	Avg. CPU-time	Max. CPU-time	Min. CPU-time	Accuracy: average error w.r.t. quadprog
quadprog	0.0031	0.0333	0.0022	–
qpOASES upt w	$5.0117 \cdot 10^{-5}$	0.0012	$4.1100 \cdot 10^{-5}$	$1.5463 \cdot 10^{-5}$
Hildreth	$6.2905 \cdot 10^{-5}$	0.0023	$4.8000 \cdot 10^{-5}$	$\mathbf{1.5460 \cdot 10^{-5}}$
HildrethAct	$\mathbf{2.8351 \cdot 10^{-5}}$	$\mathbf{3.7010 \cdot 10^{-4}}$	$\mathbf{2.4600 \cdot 10^{-5}}$	$1.5463 \cdot 10^{-5}$

two sample intervals. Similar results can be observed for other time windows, but are not reported due to space limitations. The average, maximum, minimum CPU-time and accuracy of the QP-solvers in terms of the error with respect to quadprog are summarized in Table VIII on the next page, where again the error is defined in (1). In each column the best numerical result is highlighted in blue with bold font. Here, the abbreviation 'qpOASES upt w' denotes the qpOASES QP-solver, but with 'warm-start'- and 'update'-features turned on. The 'warm-start'-feature permits active set $\mathbb{A}$ to be reused from the previous sample, which results in less number of iterations because only changes in $\mathbb{A}$ are updated. The 'update'-feature permits qpOASES to update F and γ in (8) only as they depend on estimated state $\hat{x}$ and future reference $\mathcal{R}_k$. Hildreth and HildrethAct are cold-started, which means that active set $\mathbb{A}$ is computed from scratch, which means more iterations are needed.

2) Observations: We can make four observations from Fig. 3 and Table VIII. First, we can see that HildrethAct has the same accuracy as qpOASES. This is also true for other prediction horizons N and constraint horizons N_q. Second, we can see that HildrethAct needs significantly less iterations than Hildreth, which therefore makes HildrethAct faster than Hildreth. Third, although qpOASES needs less iterations than HildrethAct due to warm-start, HildrethAct is actually faster if we divide the CPU-time by the number of iterations. Finally, HildrethAct has the lowest maximum, minimum and average CPU-time.

3) MPC parameters: All MPC parameters are summarized in Table VII on the previous page.

Note that for simulation we use $N_q = N$, which means the QP has full complexity. Furthermore, `max_iter` is set to ∞ for QP-solver HildrethAct to fully converge to an optimal and feasible solution with flag 0.

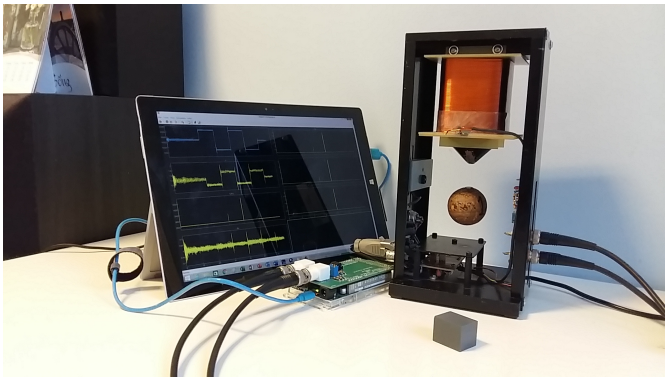


Figure 4: Experimental setup.

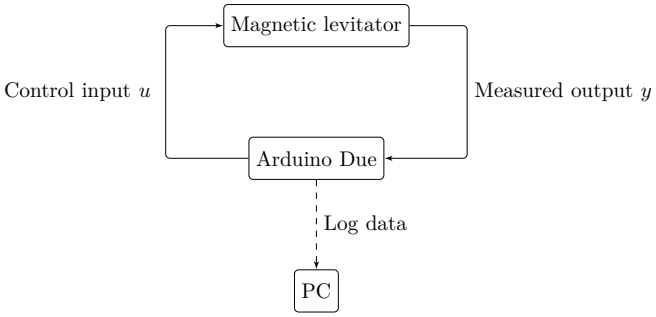
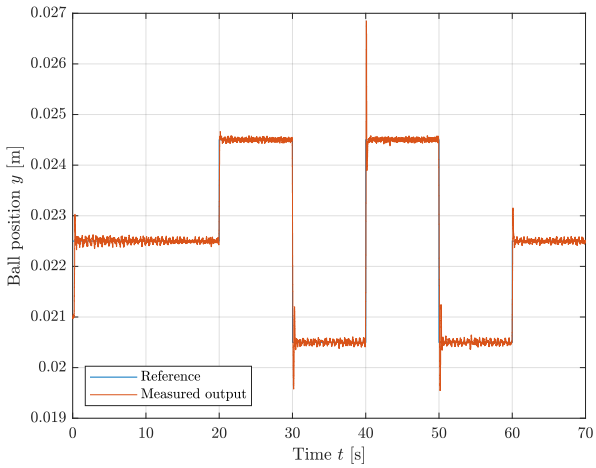
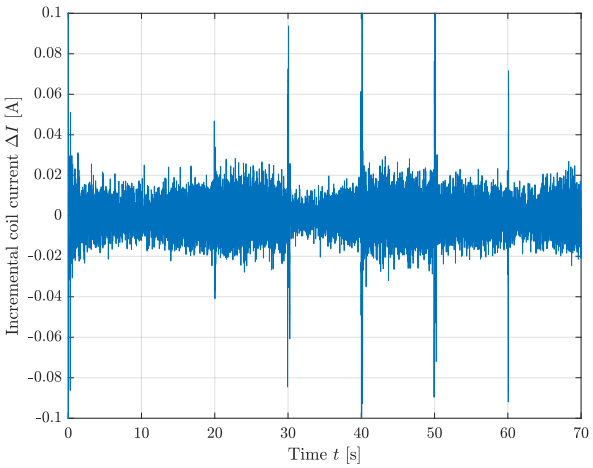


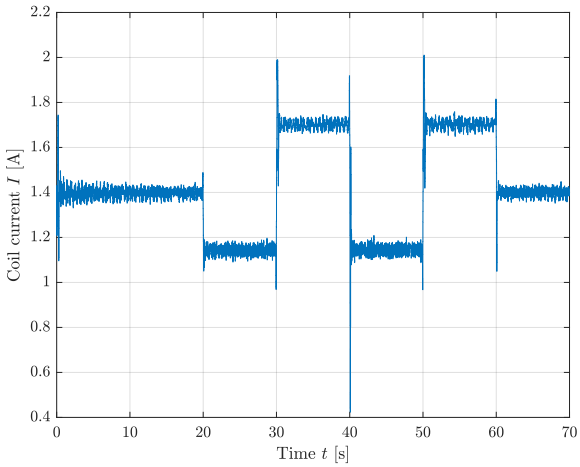
Figure 5: Schematic experimental setup (simplified).



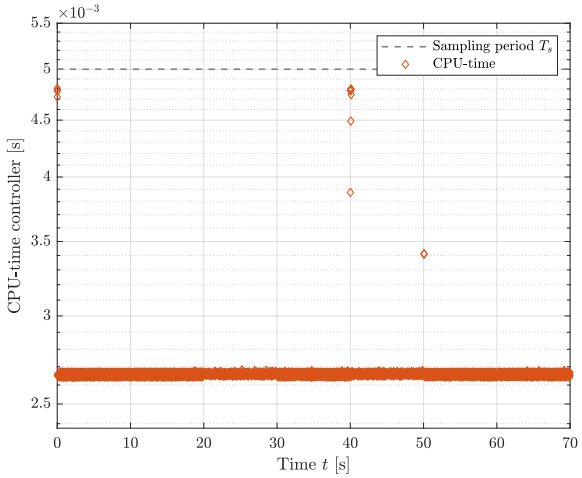
(a) Measured output y tracking reference r



(c) Incremental control input Δu



(b) Control input u



(d) CPU-time

B. Real-time experiments

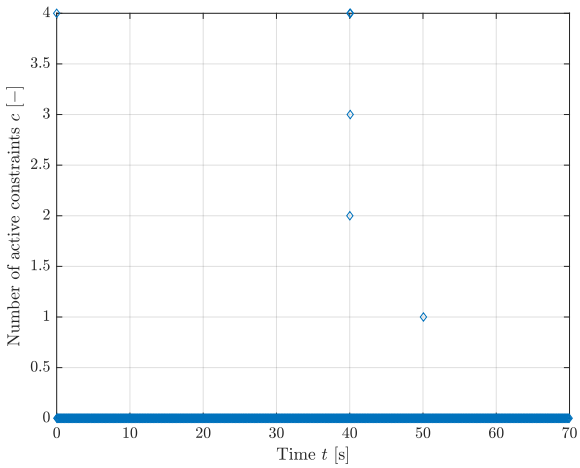
In Fig. 4, a picture of the magnetic levitator experimental setup is shown, where the ball is floating (magnetic force is equal to gravity) during experiment. In the figure can also be seen that power-LED turned on but built-in pin 13 (for overrun detection) is off because CPU-time $< T_s$.
The blue USB-cable in Fig. 4 connects the Arduino to the PC for data logging and uploading the C/C++-code. The PC is a Microsoft Surface Pro 3, but its specifications are not relevant. HildrethAct is run standalone on the Arduino Due. A simplified schematic representation of the experimental setup is sketched in Fig. 5.

A shield forms the bridge between the magnetic levitator setup and the Arduino Due. In our case, pin `DAC1` on the shield is connected to the actuator that provides the coil current as control input u and pin `0` on the shield is connected to the sensor for reading vertical ball position as output y.

The experiment lasts for 70 s and with sampling time $T_s = 5$ ms this is equivalent to 14000 samples. The reference is a square wave with amplitude of 2 mm around $y = 0.0225$ m and each level lasts for 10 seconds.

1) Results: Here, HildrethAct is cold-started, because the active set may be completely different from previous iterations due to disturbances on the experimental setup. Experimental data are presented in Fig. 6 on the next two pages. The plots showing the number of iterations `km` and flag-indicator were omitted due to space limitations. At most 4 iterations and on some occasions 3 and 2 iterations are performed, but mostly 1. On some occasions the QP-solver did not fully converge, as `max_iter` of 4 has been reached and the corresponding flags are 3 and 2 in order to respect the constraints on Δu, u and y. However, in most cases the QP-solver converged fully to an optimal and feasible solution as the flag is mostly 0.

2) Observations: We can make four observations about the experimental data. First, the output y tracks the reference r without offset due to the integral action MPC. There is some overshoot, but the output settles to the steady-state value pretty fast. There are also high frequency oscillation when the setpoint is reached in steady-state, this could be reduced by choosing a smaller T_s. We can also see that the controller anticipates for future reference changes by jumping early. PID cannot anticipate and therefore may have larger overshoots and higher settling time. Second, we can see that CPU-time $< T_s$, which is also indicated by the built-in LED on pin 13 which did not light up. Finally, we can see that constraints on Δu, u and y are respected and CPU-time is the highest when



(e) Number of active constraints c

Figure 6: Experimental data.

constraints are active, but it is useful to see that mostly constraints are not active.

3) MPC parameters: All MPC parameters are summarized in Table IX on the next page. The parameters for simulation were different with the idea to have as many constraints active as possible for better comparison between the QP-solvers. In simulation the constraints were more tight and the cost weights Q, R, P were tuned accordingly. Note that for experiment we use $N_q \ll N$, to reduce the complexity of the QP. Furthermore, `max_iter` is set to 4 as this value approximately corresponds to CPU-time $< 0.9T_s$, which results in flags 0, 2, or 3. The flag of 1 did not occur, because the QP was always feasible and λ was not NaN or ∞.

4) Memory footprint HildrethAct: The memory footprint of HildrethAct on the Arduino Due (Atmel ARM Cortex-M3) is presented in Table X on the next page. It is much lower than the ones for QP-solvers in the literature discussed in the introduction, such as [21], [22], [23].

Table IX: MPC parameters for experiment

Parameter	Value	Unit
Stage cost weight Q	1	[–]
State cost weight R	0.0005	[–]
Terminal cost weight P	$40Q$	[–]
Prediction horizon N	14	[–]
Constraint horizon N_q	4	[–]
Sampling period T_s	5	[ms]
$[\Delta u_{\min}, \Delta u_{\max}]$	$[-0.1, 0.1]$	[A]
$[u_{\min}, u_{\max}]$	$[0, 3]$	[A]
$[y_{\min}, y_{\max}]$	$[0.018, 0.028]$	[m]
`max_iter`	4	[–]
Baud rate	115200	[bits/s]
Reference $[r_{\min}, r_{\max}]$	$[0.0205, 0.0245]$	[m]

Table X: Memory footprint HildrethAct on Arduino Due (Atmel ARM Cortex-M3)

Memory type	Bytes	% full	Total
Program (`.text`) or Flash memory	60688	11.6	512 kB
Data (`.data + .bss`) or SRAM	15020	15.3	96 kB

VII. Conclusion

In this thesis, we designed two MPC-QP solvers Hildreth+' and HildrethAct, which are based on the Hildreth QP-solver algorithm. Hildreth's algorithm has been used as a starting point, due to its simple implementation of KKT-conditions. The HildrethAct QP-solver has been analysed for its memory footprint and computational complexity.

The MPC problem is made suitable for real-time control on an Arduino by reducing the complexity of a QP by introducing the concept of constraint horizon N_q. Workflow from design in MATLAB to real-time implementation on an Arduino has been described. The flag-system is chosen to prevent stalling of our QP-solver. Accuracy of HildrethAct has been compared with Hildreth and qpOASES in simulation on a PC. Finally, integral action MPC with affine term using HildrethAct as the QP-solver has been deployed standalone on an Arduino Due which succesfully tracked a reference on the magnetic levitator setup. This is a fast nonlinear system that requires a small sampling period of $T_s = 5$ ms. HildrethAct had a much smaller memory footprint than the QP-solvers in the literature and CPU-time $< T_s$ during experiment.

Recommendations for future research is twofold. First, formulating a quasi-LPV approach for the magnetic levitator setup, as this approach exploits the nonlinearity of the model. Second, conducting research on infeasibility detection by QP-solvers quadprog, qpOASES and mpcqpsolver.

VIII. Acknowledgements

First, I would like to thank my daily supervisor dr. Mircea Lazar for organizing this project. His advice and knowledge of MPC helped me a lot in designing the QP-solvers. His idea to start with the Hildreth's QP-solver was a good move and inspired the design of the new QP-solvers. He is always willing to help and discuss any problems I encountered. I learned a lot from his detailed feedback on my progress reports and final thesis.

I am thankful to ing. Will Hendrix for providing technical support and SIMULINK software for the experimental setup and Arduino.

Many thanks to the people from MathWorks for troubleshooting MATLAB and SIMULINK related problems and to MSc Shengling Shi for sharing ideas about the experimental setup.

I would also like to thank the graduation committee consisting of prof. Paul van den Hof, dr. Mircea Lazar, dr. Sophie Haesaert and dr. Dip Goswami for providing me detailed feedback during the halfway evaluation. It was very useful to reflect on my work and I learned a lot from it.

Last but not least, I would like to thank my parents Huỳnh Tuyết Nga and Lâm Thành Hồ and my sister Cindy Lâm Minh Kiều for their support during my thesis and making my study possible. During my studies, I learned many important things from them.

References

[1] P. Zometa, M. Kögel, T. Faulwasser, and R. Findeisen, "Implementation Aspects of Model Predictive Control for Embedded Systems," *Proceedings of the American Control Conference*, 2012.

[2] D. Piga, S. Formentin, and A. Bemporad, "Direct data-driven control of constrained linear parameter-varying systems: A hierarchical approach," *IEEE Transactions on Control Systems Technology*, vol. 26, pp. 1422–1429, 2018.
[3] P. Chalupa, J. Novák, and M. Malý, "Modelling and model predictive control of magnetic levitation laboratory plant," *Proceedings 31st European Conference on Modelling and Simulation*, 2017.
[4] R. Kouki, H. Salhi, and F. Bouani, "Application of Model Predictive Control for a thermal process using STM32 Microcontroller," *2017 International Conference on Control, Automation and Diagnosis (ICCAD'17), Hammamet - Tunisia*, 2017.
[5] pronenewbits, "Arduino Unconstrained MPC Library." [Online]. Available: https://github.com/pronenewbits/Arduino_Unconstrained_MPC_Library
[6] M. Gulan, G. Takács, N. A. Nguyen, S. Olaru, P. Rodríguez-Ayerbe, and B. Rohal'-Ilkiv, "Efficient Embedded Model Predictive Vibration Control via Convex Lifting," *IEEE Transactions on Control Systems Technology*, vol. 27, pp. 48 – 62, 2019.
[7] J. Hedengren, "Process Control Temperature Lab." [Online]. Available: https://github.com/APMonitor/arduino
[8] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "OSQP: An Operator Splitting Solver for Quadratic Programs," *arXiv: 1711.08013*, 2019.
[9] H. J. Ferreau, "Model Predictive Control Algorithms for Applications with Millisecond Timescales," *Universiteit Leuven, Department of Electrical Engineering, PhD thesis*, 2011.
[10] H. J. Ferreau, "An Online Active Set Strategy for Fast Solution of Parametric Quadratic Programs with Applications for Predictive Engine Control," *Ruprecht-Karls-Universität Heidelberg, MSc thesis*, 2006.
[11] H. J. Ferreau, C. Kirches, A. Potschka, H. G. Bock, and M. Diehl, "qpOASES: a parametric active-set algorithm for quadratic programming," *Mathematical Programming Computation*, vol. 6, pp. 327–363, 2014.
[12] L. Wang, "Model Predictive Control System, Design and Implementation using Matlab," *Springer*, 2009.
[13] J. V. Frasch, M. Vukov, H. J. Ferreau, and M. Diehl, "A dual Newton strategy for the efficient solution of sparse quadratic programs arising in SQP-based nonlinear MPC," *Optimization Online*, pp. 1–8, 2013. [Online]. Available: http://www.optimization-online.org/DB_FILE/2013/07/3972.pdf
[14] J. V. Frasch, M. Vukov, H. J. Ferreau, and M. Diehl, "A new quadratic programming strategy for efficient sparsity exploitation in SQP-based nonlinear MPC and MHE," *19th World Congress, IFAC*, 2014.
[15] S. Zavitsanou, A. Chakrabarty, E. Dassau, and F. J. Doyle III, "Embedded Control in Wearable Medical Devices Application to the Artificial Pancreas," *Processes*, vol. 4, 2016.
[16] B. J. T. Binder, D. K. M. Kufoalor, and T. A. Johansen, "Scalability of QP solvers for Embedded Model Predictive Control Applied to a Subsea Petroleum Production System," *2015 IEEE Conference on Control Applications (CCA)*, 2015.
[17] D. K. M. Kufoalor, B. J. T. Binder, H. J. Ferreau, L. Imsland, T. A. Johansen, and M. Diehl, "Automatic Deployment of Industrial Embedded Model Predictive Control using qpOASES," *2015 European Control Conference (ECC)*, 2015.
[18] B. Huyck, L. Callebaut, F. Logist, H. J. Ferreau, M. Diehl, J. De Brabanter, J. F. Van Impe, and B. De Moor, "Implementation and Experimental Validation of Classic MPC on Programmable Logic Controllers," *2012 20th Mediterranean Conference on Control & Automation (MED), Barcelona, Spain*, 2012.
[19] B. Huyck, H. J. Ferreau, M. Diehl, J. De Brabanter, J. F. Van Impe, B. De Moor, and F. Logist, "Towards Online Model Predictive Control on a Programmable Logic Controller: Practical Considerations," *Mathematical Problems in Engineering*, vol. 23, pp. –, 2012.
[20] B. Huyck, J. De Brabanter, B. De Moor, J. F. Van Impe, and F. Logist, "Online model predictive control of industrial processes using low level control hardware: a pilot-scale destillation column case study," *Control Engineering Practice*, vol. 28, pp. 34–48, 2014.
[21] C. Ibañez, C. Ocampo-Martinez, and B. Gonzalez, "Embedded optimization-based controllers for industrial processes," *2017 IEEE 3rd Colombian Conference on Automatic Control (CCAC)*, pp. 1–6, 2017.
[22] C. Ibañez and C. Ocampo-Martinez, "Implementation of optimization-based controllers for industrial processes," *Barcelona School of Industrial Engineering (ETSEIB), Barcelona, Spain, MSc thesis*, 2017.
[23] S. Adhau, S. Patil, D. Ingole, and D. Sonawane, "Implementation and Analysis of Nonlinear Model Predictive Controller on Embedded Systems for Real-time Applications," *2019 18th European Control Conference (ECC)*, 2019.
[24] Arduino, "Arduino Due." [Online]. Available: https://store.arduino.cc/arduino-due
[25] V. T. T. Lam, A. Sattar, L. Wang, and M. Lazar, "Fast Hildreth-based Model Predictive Control of Roll Angle fir a Fixed-Wing UAV," *Accepted for presentation at the 2020 IFAC World Congress, Berlin, Germany*, July 2020.
[26] S. Mittal, "A Study of Successive Over-relaxation (SOR) Method Parallelization Over Moderm MPC Languages," *International Journal of High Performance Computing and Networking, Inderscience*, vol. 7, pp. 292 – 298, 2014.
[27] S. Roberts, "Topic 3, Iterative methods for $Ax = b$, 3.2 Jacobi method, 3.3 Gauss-Seidel method," *University of Oxford, Machine Learning Research Group, Lecture notes*, 2010.
[28] Y. Cao, "Update Inverse Matrix," 2008. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/18063-update-inverse-matrix
[29] P. Bergström, "Plot 2d/3d region," 2010. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/9261-plot-2d-3d-region
[30] B. Grünbaum, "Convex Polytopes," *Springer*, 2003. [Online]. Available: https://www.springer.com/gp/book/9780387004242

Appendix A
State-of-the-art: memory footprint and CPU-time of various QP-solvers

The details of memory footprint and CPU-time for various QP-solvers found in the literature are summarized in Table XI on the next page.

Appendix B
Step-by-step Hildreth+' algorithm

0. The `persistent` variable $\tilde{\lambda}$ is defined for warm-start. Reuse solution from previous iteration $\tilde{\lambda} = \tilde{\lambda}_{\text{old}}$
1. Unconstrained solution: $\theta = -E^{-1}F$
2. Compute $K = \gamma - M\theta$, `if` constraints satisfied, i.e. `all(K >= 0)` `then` return `else` continue `end`
3. Compute indices that satisfy $K < \sigma$ and extract corresponding rows and columns of H and K to be $\tilde{H}$ and $\tilde{K}$ resp.
4. Using successive overrelaxation (SOR), update $\tilde{\lambda}^m$ in a scalar-based `for`-loop, with

$$w_i^m = (1-\omega)\,\tilde{\lambda}_i^{m-1} + \omega \cdot \underbrace{\frac{1}{D_{ii}}\left[-k_i - (L_i + U_i)\,\tilde{\lambda}^{m-1}\right]}_{\text{original}}$$

$$\tilde{\lambda}_i^m = \max\left(0, w_i^m\right)$$

`if` $e = \lambda^m - \lambda^{m-1}$, $e^T e \leq \delta$ or $m \geq \bar{m}$ `then` stop `else` continue `end`
5. Extract rows of M corresponding to indices from Step 4 and return $\theta = \theta - E^{-1}\tilde{M}^T\tilde{\lambda}^m$, where θ from Step 1 is reused.
6. Store $\tilde{\lambda}_{\text{old}} = \tilde{\lambda}$ for warm-start.

Appendix C
Step by step HildrethAct algorithm

1. Define maximum number of iterations `max_iter`, `flag = 0`
2. Compute unconstrained solution $\theta_0 = -E^{-1}F$
3. Compute $K_0 = \gamma - M\theta_0$
4.a. Set iteration counter to `km = 1`
b. Compute $K_{\text{min}} = \min(K_0)$
c. `if` $K_{\text{min}} \geq 0$ `then` stop `else` continue `end`
d. Initialize empty active set $\mathbb{A} = \varnothing$, empty Lagrange multiplier $\lambda_{\text{act}} = \varnothing$ and inverse matrix $H_{\text{act}}^{-1} = \varnothing$
5. `while` $K_{\text{min}} < 0$ (infeasible) `then`

Table XI: State-of-the-art: memory footprints and CPU-time of various QP-solvers

Application	**Platform**	**CPU-speed**	**Total memory**	**QP-solver**	**Sampling period T_s**	**Memory footprint**	**CPU-time**	**Prediction, control horizon N_p, N_c**
Process control of temperature [18]	Siemens CPU319-3DP/PN PLC	250 MHz	8 Mb	qpOASES	1 s	–	Max. 125 ms	$N_p = 22$, $N_c = 7$
				Hildreth			Max. 24 ms	
Process control of temperature [19]				qpOASES	–	112 kB	Max. 210 ms	
				Hildreth		7 kB	Max. 52 ms	
Process control of temperature [20]				qpOASES	–	51420 bytes	Max. 2026 ms, Avg. 1308.8 ms	$N_p = 50$, $N_c = 10$
				Hildreth		53124 bytes	Max. 556 ms, Avg. 212.4 ms	
Process control of water level in tank [21]	National Instruments NI myRIO-1900	667 MHz	nonvolatile 512 MB, DDR3 256 MB	qpOASES	1.430 s	–	1430 ms	$N_p = N_c = 10$
Process control of water level in tank [22]						707 KB	400 ms	$N_p = N_c = 5$
						756 KB	1430 ms	$N_p = N_c = 10$
DC motor control (2 states) [23]	Atmel ARM Cortex-M3	84 MHz	program memory 512 kB, SRAM 96 kB	qpOASES	–	352 kB	Avg. 242 μs	$N_p = N_c = 5$
				qpDUNES		298 kB	Avg. 305 μs	
Hovercraft control (6 states) [23]				qpOASES, qpDUNES	500 ms	Memory overflow	Not available	
Quadrotor control (9 states) [23]					–			
Process control of petroleum production [16]	ABB AC500 PM592-ETH PLC	500 MHz	User program 4 MB, user memory 4 MB	qpOASES	–	Program 372 kB, Data 119 kB	Max. 12 ms, Avg. 3 ms	17 variables, 83 inequalities
Process control of liquid-gas seperator [17]					1 s	Double precision floating point: PLC code size 1.59 MB, data size 0.29 MB	Warm-start, double precision: max. 13.1 ms, avg. 4.9 ms	24 variables, 96 inequalities, 24 bounds
Electric pump control [17]						Double precision floating point: PLC code size 0.37 MB, data size 0.11 MB	Warm-start, double precision: max. 7.3 ms, avg. 2.0 ms	17 variables, 78 inequalities, 17 bounds

a. Add constraint index corresponding to $K_{\min}$ to active set $\mathbb{A}$
b. Update H_{act}^{-1} and λ_{act} using additive matrix update
c. Compute $\lambda_{\min} = \min(\lambda_{\text{act}})$
d. `while` $\lambda_{\min} < 0$ (not optimal) `then`
  i. Remove constraint index corresponding to $\lambda_{\min}$ from active set $\mathbb{A}$
  ii. Update H_{act}^{-1} and λ_{act} using subtractive matrix update
  iii. Compute $\lambda_{\min} = \min(\lambda_{\text{act}})$
  iv. Set `km = km + 1`
  v. `if` $\lambda =$ NaN or $\lambda = \infty$ `then flag = 1` and stop `else` continue `end`
  vi. `if km >= max_iter then flag = 2` and stop `else` continue `end`
`end`
e. `if` $\lambda =$ NaN or $\lambda = \infty$ `then flag = 1` and stop `else` continue `end`
f. `if km >= max_iter then flag = 2` and stop `else` continue `end`
g. Compute $K = K_0 + H_{\text{act}}\lambda_{\text{act}}$
h. Compute $K_{\min} = \min(K)$
i. Set `km = km + 1`
`end`
6. Compute constrained solution $\theta = \theta_0 - E^{-1}M_{\text{act}}^T\lambda_{\text{act}}$

In Step 4.d, the symbol $\varnothing$ denotes an empty set. The min-function takes as input a vector and outputs the smallest number and the corresponding index.

Appendix D
Derivation of the state-space model with affine term

Suppose we have a continuous-time state-space model Σ_c given as

$$\Sigma_c := \begin{cases} \dot{x}_p &= A_c x_p + B_c u + w_c \\ &= A_c x_p + \begin{bmatrix} B_c & w_c \end{bmatrix} \begin{bmatrix} u \\ 1 \end{bmatrix} \\ y &= C_c x_p + D_c u. \end{cases}$$

Because of the affine term w_c, we need to augment the control input with an additional 1. Using MATLAB-function `c2d`, we can convert it to the discrete-time state-space model Σ_d with sampling period T_s.

$$\Sigma_d := \begin{cases} x_p(k+1) &= A_d x_p(k) + B_d u(k) + w_d \\ &= A_d x_p(k) + \begin{bmatrix} B_d & w_d \end{bmatrix} \begin{bmatrix} u(k) \\ 1 \end{bmatrix} \\ y(k) &= C_d x_p(k) + D_d u(k). \end{cases}$$

Appendix E
Simulink-blocks

A. *Sensor-block*

This block takes as input the sensor value s and outputs the measured output $y = \alpha s + \beta$, where $\alpha, \beta \in \mathbb{R}$ are constant.

B. *Actuator-block*

This block takes as input the control input u and outputs the actuator value $p = \psi u$, where $\psi \in \mathbb{R}$ is constant.

C. *Step-by-step algorithm futureR-block*

This algorithm outputs future reference $\mathcal{R}_k \in \mathbb{R}^N$ and the very next reference sample $\mathcal{R}_1 \in \mathbb{R}$, with prediction horizon N.

1. Define two `persistent` variables: sample counter i and future reference $\mathcal{R}_k$
2. `if` $\mathcal{R}_k$ is empty `then`
  a. Determine how many times q each piece-wise constant part fits within N and how many samples r remain using $[q, r] =$ quorem(N, samples per piece-wise constant part).
  b. Generate $\mathcal{R}_k$ based on q and r.
`end`
3. `if` i is empty `then` set $i = N - 1$ `end`
4. `if` $i > N - 1$ `then`
  a. Determine how many times q each piece-wise constant part fits within i and how many sample r remain using $[q, r] =$ quorem(i, samples per piece-wise constant part).
  b. Remove first element of $\mathcal{R}_k$ and append new value based on q and r.
`end`
5. Extract $\mathcal{R}_1$ as first element of $\mathcal{R}_k$
6. Increase sample counter $i = i + 1$

The quorem-function taks as input two scalars $a, b \in \mathbb{R}$ and outputs the quotient q and remainder r of the division $\frac{a}{b}$.

D. *Step-by-step algorithm ConMPC-block*

The inputs are future reference $\mathcal{R}_k \in \mathbb{R}^N$ and output $y \in \mathbb{R}$, where N is the prediction horizon. The outputs are scalars control input u, CPU-time, incremental control input Δu, number of iterations `km`, number of active constraints c and flag.

1. Start timer for CPU-time
2. Define `persistent` variables u_{old}, $\hat{x}_{\text{old}}$, Δu_{old}
3. Load constants such as equilibrium points (y_0, u_0), observer matrices etc.
4. `if` u_{old} is empty `then` $u_{\text{old}} = u_0$ `end`
5. `if` $\hat{x}_{\text{old}}$ is empty `then` $\hat{x}_{\text{old}} = \begin{bmatrix} 0 & 0 & y_0 \end{bmatrix}^T$ `end`
6. `if` Δu_{old} is empty `then` $\Delta u_{\text{old}} = 0$ `end`
7. State estimation with observer

$$\hat{x} = (A - LC)\hat{x}_{\text{old}} + B\Delta u_{\text{old}} + Ly$$

8. Compute

$$F = 2\Gamma^T\Omega\left(\Phi\hat{x} - \mathcal{R}_k\right), \gamma = c + W\hat{x} - Vu_{\text{old}}$$

9. Solve QP using the HildrethAct solver $[\Delta U_k,$ `km`, flag, c, $\theta_0] =$ HildrethAct(E^{-1}, F, M, γ, H)
10. Extract Δu as first element of ΔU_k
11. Compute control input $u = u_{\text{old}} + \Delta u$
12. Flag system:
`if` flag $= 2$ `then`
  a. `if` sub-optimal solution satisfies $\Delta u \in [\Delta u_{\min}, \Delta u_{\max}]$ and $u \in [u_{\min}, u_{\max}]$ `then` set flag $= 3$ `else` saturate the sub-optimal solution such that $\Delta u \in [\Delta u_{\min}, \Delta u_{\max}]$ and $u \in [u_{\min}, u_{\max}]$ `end`
`elseif` flag $= 1$ `then`
  a. Sature unconstrained solution θ_0 such that $\Delta u \in [\Delta u_{\min}, \Delta u_{\max}]$ and $u \in [u_{\min}, u_{\max}]$

end

13. Update $u_{\text{old}} = u$, $\hat{x}_{\text{old}} = \hat{x}$, $\Delta u_{\text{old}} = \Delta u$

14. Stop timer and compute CPU-time