Implementation Aspects of Model Predictive Control for Embedded Systems

Pablo Zometa, Markus Kögel, Timm Faulwasser, and Rolf Findeisen

Abstract— **In this paper we discuss implementation related aspects of model predictive control schemes on embedded platforms. Exemplarily, we focus on fast gradient methods and present results from an implementation on an embedded low-cost ARM processor. We show that input quantization happening in actuators should be taken into account in order to determine the suboptimality level of the online optimization. Furthermore, we present results which allow the off-line determination of the online memory demand of the fast-gradient MPC algorithm on the embedded system. As a case study we consider a *Segway*-like robot, modeled by an LTI-system with 8 states and 2 inputs subject to box input constraints. The test system runs with a sampling period of 4 ms and uses MPC horizons up to 20 steps in a hard real-time system with limited CPU time and memory.**

Index Terms— **model predictive control, embedded systems, real-time implementation, Fast Gradient method, LEGO NXT**

I. Introduction

Model Predictive Control (MPC) has a long and successful history in the chemical industry, where it is used to control high-performance processes with slow dynamics [1]. One of the main strengths of MPC is its ability to deal with input, output and state constraints as well as with multi-input multi-output systems in a systematic way. One of its main drawbacks is that high computational demands are typically associated with it, which makes real-time implementations challenging. The continuous development of more efficient MPC algorithms continues to reduce these demands. However, it is still challenging to implement MPC on embedded platforms that control fast mechatronic systems with sampling times in range of miliseconds. These challenges arise because embedded platforms are typically based on processors with clock frequencies in the MHz range and memories in the range of kB.

MPC is built upon the repetitive solution of an optimal control problem (OCP). In case of linear, time-invariant, discrete-time systems with affine input and state constraints and a quadratic cost function, the OCP is equivalent to a quadratic program (QP). The OCP arising in MPC can be solved using off-line or online methods. The former approach (known as explicit MPC [2]) is particularly fast for small systems, but has a much larger memory footprint than the latter (especially for medium and large size problems), since the explicit solution needs to be stored in a table or tree, which become very large for larger problems. An online MPC scheme with a good balance between computational speed and memory demand is the Fast Gradient method (FGM) which was presented in [3]. It is based on Nesterov's gradient method [4]. In [3] an upper bound for the maximum number of iterations needed to ensure a pre-defined accuracy is provided, and the computational complexity is shown to be quadratic with respect to the horizon length and the number of inputs. In [5], the method is implemented on a high performance digital signal processor to control a simulated plant, where FGM performs better than explicit MPC. In [6], the structure of the MPC problem is exploited to derive an algorithm with computational and memory demands linearly increasing with the horizon length and the number of inputs. A simulation example shows that it is of advantage for large horizon lengths.

In this work we focus on the implementation aspects of FGM-based MPC on embedded systems. In constrast to previous works on FGM-based MPC [3], [5], [6] we present results from an implementation on a real system, using a hard real-time operating system (RTOS), and a low-cost 32-bit 48-MHz embedded microcontroller. We stabilize a system with 8 states and 2 inputs using horizon lengths of up to 20 steps with sampling times of 4 ms. In addition, we derive a relation to determine off-line the memory demand of the algorithm. For the considered example the memory required is in the order of kilobytes even for long horizons. Moreover, we propose the use of the input quantization taking place in the system actuators to determine the suboptimality level of the approximate QP solution, instead of using an ad-hoc value based on the MPC cost function. Furthermore, we show with several Monte Carlo experiments that the theoretical upper bound for the number of iterations is a factor two to three higher than the number observed in practice. We implement the algorithm using fixed-point and floating-point arithmetics, and show that even in the former case the error is less than one quantization level of the inputs. Note that we focus on outlining the challenges and solutions for implementing MPC on limited embedded platforms. Explicit stability and robustness guarantees are beyond the scope of this paper.

The structure of this paper is as follows. Section II introduces the system to be controlled, and briefly describes the microcontroller and the RTOS used. In Section III we briefly recall the Fast Gradient method in the context of MPC, and we present an alternative way to select the maximum number of iterations based on the hardware. In Section IV we present the analysis of the computational requirements of the implementation. Finally, in Section V we state our conclusions.

P.Z., M.K., T.F. and R.F. are with the Institute for Automation Engineering, OvG University Magdeburg, Germany. M.K. and T.F. are also with the International Max Planck Research School for Analysis, Design and Optimization in Chemical and Biochemical Process Engineering, Magdeburg, Germany e-mail: {pablo.zometa, markus.koegel, timm.faulwasser, rolf.findeisen}@ovgu.de.

II. Test system

The system to be controlled is a *Segway*-like mobile robot. Our goal is to use MPC to stabilize a two-wheel vehicle at an unstable equilibrium point, see Figure 1.

We use a popular educational system (Lego mindstorm NXT) as test platform. The electronic system (called Brick) contains an ARM-based microcontroller. Two DC motors are used as actuators, one gyroscope and two incremental encoders (one for each motor) are used as sensors. Each wheel is described by three states: wheel rotation angle, wheel angular speed, and the integration of the wheel rotation angle. We add this last state to improve the performance of the system. Two additional states are the body tilt, and the body angular speed. Using this information, we model our system as linear time-invariant with 8 states, 2 inputs and 3 outputs. The system is shown in Figure 1. The gravity vector is represented by g. The input to the system is the pulse-width modulated (PWM) voltage applied to the motors in percentage ($-100 \leq u_{pwm} \leq 100$), and the measured values are the $\dot{\theta}_{body}$ and $\theta_{motor} = \theta_{wheel} - \theta_{body}$. See Appendix I for the continuous-time system matrices. We discretized our system using a zero-order hold discretization on the inputs.

Part of our work is based on [7]. The mechanical construction is identical. We rely on the same model structure, but use a different set of parameters. The states are estimated by discrete integration and differentiation of the measurements, as done in [7]. The fastest component in our system are the DC motors, with time constant of around 50 ms. The gyroscopic sensor delivers a new measurement about every 3 ms. We set 4 ms as our sampling period, and our goal is to solve the resulting OCP in MPC in hard real-time. A video of the controlled system can be found in [8].

Embedded hardware

The NXT Brick uses an Atmel AT91SAM7S256 microcontroller which includes an ARM7TDMI processor core, 64 kB RAM, and 256 kB flash memory. The clock frequency is 48 MHz. ARM-based processors are general purpose 32-bit reduced instruction set computer (RISC). An ARM7 processor has a 32-bit integer arithmetic logic unit, but lacks a floating-point unit (FPU). Floating point operations are therefore emulated by software. ARM7TDMI provides two instruction sets: ARM and Thumb. We only use the former (the standard 32-bit instruction set), as the latter did not bring any advandtage in this application.

Embedded software

To deploy the MPC algorithm (presented in Section III) in hard real-time, we use the open-source nxtOSEK real-time operating system (RTOS) [9]. nxtOSEK is based on the OSEK-VDX open specification for embedded RTOS (standard ISO 17356) [10]. nxtOSEK allows us to develop real-time applications written in C for the NXT, using different task scheduling policies with timer resolutions down to 1 ms. We use a rate-monotonic scheduler with 3 tasks: outputs, inputs, and controller. We can guarantee that all the tasks are executed within the specified deadline if CPU utilisation is kept under $\eta = m(2^{1/m} - 1) \approx 0.78$, with $m = 3$ tasks [11]. This is a worst case sufficient, but not necessary, condition. The whole RTOS and the application code are first cross-compiled in a standard PC, then downloaded into the flash memory. During runtime, they both (the RTOS and the user application) are copied from flash into RAM. This means, that although we have 256 kB of flash available, we are limited by RAM to 64 kB of memory. The MPC controller is implemented in plain C. We do not rely on external libraries (such as BLAS/ATLAS), nor include assembler code. We perform matrix-vector multiplication using a naive approach.

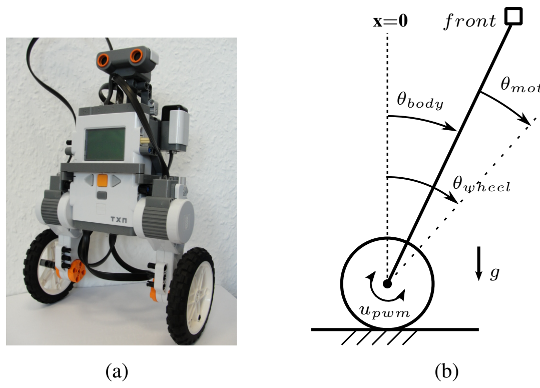

Fig. 1. The test platform: a two-wheeled vehicle constructed using the LEGO NXT hardware, see also [7].

III. MPC for embedded systems

The plant is a linear, time-invariant, discrete-time system with hard input constraints and no state constraints (see Appendix I). The system is described by

$$\begin{aligned} \mathbf{x}^+ &= A\mathbf{x} + B\mathbf{u}, \\ \mathbf{y} &= C\mathbf{x}, \end{aligned} \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{u} \in \mathbb{R}^p$, and $\mathbf{y} \in \mathbb{R}^q$ are the state, input, and output vector at the current sampling time, respectively. $\mathbf{x}^+$ is the successor state. The matrices have their typical meaning and the proper dimensions. The system is assumed to be stabilizable, which is easily verifiable for the considered test system.

The inputs are subject to the following box constraints

$$\mathcal{U} = \mathbf{l}[i] \leq \mathbf{u}[i] \leq \mathbf{r}[i], \forall\, i = 1, ..., p, \tag{2}$$

where $\mathcal{U}$ is a closed, convex set containing the origin. The performance index is a quadratic function of the form

$$V(\underline{\mathbf{x}}, \underline{\mathbf{u}}) = \frac{1}{2}\mathbf{x}_N^T P \mathbf{x}_N + \sum_{j=0}^{N-1} \left(\frac{1}{2}\mathbf{x}_j^T Q \mathbf{x}_j + \frac{1}{2}\mathbf{u}_j^T R \mathbf{u}_j\right) \tag{3}$$

where the integer $N \geq 1$ is the prediction horizon, the matrices $Q \in \mathbb{R}^{n\times n}$, $R \in \mathbb{R}^{p\times p}$, and $P \in \mathbb{R}^{n\times n}$ are the state, input, and final state weighting matrix, respectively. They are chosen such that $Q = Q^T \geq 0$, $R = R^T > 0$, and $P = P^T \geq 0$. The sequences $\underline{\mathbf{x}} = \{\mathbf{x}_0, ..., \mathbf{x}_N\}$ is the state and $\underline{\mathbf{u}} = \{\mathbf{u}_0, ..., \mathbf{u}_{N-1}\}$ is the control sequence. The elements of $\underline{\mathbf{x}}$ and $\underline{\mathbf{u}}$ need to satisfy the system dynamics (1), and the elements of satisfy (2), such that $\underline{\mathbf{u}} \in U = \{\mathbf{u}_j \in \mathcal{U},\ \forall j = 0, \dots, N-1\}$. The OCP that needs to be solved in MPC at each sampling time becomes

$$\min_{\underline{\mathbf{u}}} V(\mathbf{x}, \underline{\mathbf{u}}), \tag{4}$$

subject to (1), $\underline{\mathbf{u}} \in U$, and $\mathbf{x}_0 = \mathbf{x}$.

The MPC controller aims to bring the state of the system to the origin $\mathbf{x} = \mathbf{0}$ by penalising deviations through the control sequence $\underline{\mathbf{u}}$. Under the conditions that N is *large enough* [12], (A, B) is stabilizable, and $(A, Q^{\frac{1}{2}})$ is detectable, choosing the matrix P as the solution of the Ricatti algebraic equation, we obtain a controller that is stabilizing [13]. The optimal control sequence that minimizes V and satisfies the input constraints is denoted by $\underline{\mathbf{u}}^*$. Usually, only the first element from the sequence $\underline{\mathbf{u}}^*$ is used as input to the system at sampling time k (i.e. $\mathbf{u} = \mathbf{u}_0^*$). For an in-depth description of MPC see [1].

OCP (4) for system (1) with input constraints (2) can be expressed in condensed form as the following QP

$$\min_{\underline{\mathbf{u}} \in U} \; J(\mathbf{x}, \underline{\mathbf{u}}) = \min_{\underline{\mathbf{u}} \in U} \; \frac{1}{2} \underline{\mathbf{u}}^T H \underline{\mathbf{u}} + \underline{\mathbf{u}}^T F \mathbf{x}. \tag{5}$$

The constant matrices $F \in \mathbb{R}^{Np \times n}$ and $H = H^T \in \mathbb{R}^{Np \times Np}$ (the Hessian of J) depend on the system (1) and the performance index (3) (for a detailed description on how to compute these matrices see [14]). MPC solves the QP problem at every sampling time with a new $\mathbf{x}$.

Solution of the OCP using the Fast Gradient method

Next we briefly recall the FGM (which is also called Nesterov's gradient method [4], [15]) and its application to MPC [3], [6]. Our aim is to employ this method to solve (5) in hard realtime. To simplify notation, we denote $J(\mathbf{x}, \underline{\mathbf{u}})$ as $J(\underline{\mathbf{u}})$.

Generally speaking gradient methods are recursive algorithms that base their solution on the fact that, in the case of unconstrained minimization, $\nabla J(\underline{\mathbf{u}}^*) = 0$, i.e., the gradient of J is zero at the minimum of J. Convergence is then guaranteed if $J(\underline{\mathbf{u}})$ is strictly convex and the value of the cost function decreases in the next iteration, i.e. $J(\underline{\mathbf{u}}^{i+1}) < J(\underline{\mathbf{u}}^i)$. Often one starts an algorithm for a gradient method with an initial guess $\underline{\mathbf{u}}^0$, and stops it after i_{max} iterations, such that

$$\epsilon \geq J(\underline{\mathbf{u}}^{i_{max}}) - J(\underline{\mathbf{u}}^*), \tag{6}$$

where $\epsilon > 0$ is the suboptimality level, and $\underline{\mathbf{u}}^{i_{max}}$ is called an ϵ-suboptimal point. In general, i_{max} depends on the initial guess and is unknown when the algorithm starts. As it will become clear later, a theoretical bound for i_{max} can be found by performing a convergence analysis on the FGM.

The cost function in (5) has two important properties: it is strongly convex and its gradient is Lipschitz continuous. Strong convexity implies

$$J(\underline{\mathbf{u}}^{i_{max}}) - J(\underline{\mathbf{u}}^*) \geq \frac{1}{2}\mu \|\underline{\mathbf{u}}^{i_{max}} - \underline{\mathbf{u}}^*\|_2^2. \tag{7}$$

Furthermore, under the conditions $Q \geq 0$ and $R > 0$, the convexity parameter $\mu > 0$ and the Lipschitz constant $L > 0$ are given by the minimum and maximum eigenvalues of H, respectively. These two constants are key in finding the solution of the QP (5) via the FGM.

The algorithm of the FGM starts computing the gradient of J at a point $\underline{\mathbf{w}}$, which is the initial guess for $\underline{\mathbf{u}}^*$, as follows

$$\nabla J(\underline{\mathbf{w}}) = H\underline{\mathbf{w}} + F\mathbf{x}. \tag{8a}$$

Next an unconstrained gradient step is performed

$$\underline{\mathbf{v}}(\underline{\mathbf{w}}) = \underline{\mathbf{w}} - \frac{1}{L}\nabla J(\underline{\mathbf{w}}). \tag{8b}$$

Input constraints are considered by the projection of $\underline{\mathbf{v}}$ onto the set of admissible solutions

$$P_U\left(\underline{\mathbf{v}}(\underline{\mathbf{w}})\right) = \arg\min_{\underline{\mathbf{q}} \in U} \|\underline{\mathbf{q}} - \underline{\mathbf{v}}(\underline{\mathbf{w}})\|_2^2. \tag{8c}$$

$P_U(\cdot)$ is called the projected gradient step. If the input is subject to box constraints this projection is efficiently done by saturating $\underline{\mathbf{v}}$. In that case we have

$$\begin{aligned} P_U\left(\underline{\mathbf{v}}(\underline{\mathbf{w}})\right) =& \underline{\mathbf{z}}, \\ \mathbf{z}_j[i] =& \begin{cases} \mathbf{l}[i] & \text{if } \mathbf{v}_j[i] < \mathbf{l}[i] \\ \mathbf{r}[i] & \text{if } \mathbf{v}_j[i] > \mathbf{r}[i] \\ \mathbf{v}_j[i] & \text{otherwise} \end{cases} \\ & i = 1, \dots, p,\ j = 0, \dots, N-1. \end{aligned} \tag{8d}$$

The iterative process of the FGM to find an ϵ-suboptimal point $\underline{\mathbf{u}}^{i_{max}}$ is

$$\underline{\mathbf{u}}^i = P_U\left(\underline{\mathbf{v}}(\underline{\mathbf{w}}^{i-1})\right) \tag{8e}$$

$$\underline{\mathbf{w}}^i = \underline{\mathbf{u}}^i + c(\underline{\mathbf{u}}^i - \underline{\mathbf{u}}^{i-1}) \tag{8f}$$

$$i = 1, \dots, i_{max},$$

where $c \geq 0$ is a constant defined as

$$c = \frac{\sqrt{L} - \sqrt{\mu}}{\sqrt{L} + \sqrt{\mu}}, \tag{8g}$$

see [15], and $\underline{\mathbf{w}}^0 = \underline{\mathbf{u}}^0$ is our initial guess for $\underline{\mathbf{u}}^*$. The computation of $\underline{\mathbf{w}}^i$ in (8f) at iteration i can be understood as the computation of a suitable initial guess for next iteration. The FGM (8) is summarized in Algorithm 1.

An upper bound i_{max}, for the number of iterations that guarantee (6) is given in [15]. In the case of cold-starting i_{max} is given by

$$i_{max} = \min\left\{ \left\lceil \frac{\ln 2\epsilon - \ln(L+\mu)d^2}{\ln(1 - \sqrt{\frac{\mu}{L}})} \right\rceil, \left\lceil \sqrt{\frac{2(L+\mu)d^2}{\epsilon}} - 2 \right\rceil \right\}, \tag{9}$$

where $\lceil \cdot \rceil$ denotes rounding up to the next integer and the cold start initial guess is $\underline{\mathbf{u}}_j^0[i] = (\mathbf{r}[i] + \mathbf{l}[i])/2, i = 1, \dots, p, j = 0, \dots, N-1$. For box constraints the constant d is

$$d^2 = N \sum_{i=1}^{p} \left(\frac{\mathbf{r}[i] - \mathbf{l}[i]}{2} \right)^2.$$

Refer to [3] for details on computing an i_{max} for the warm-start strategy.

Algorithm 1 Fast Gradient Method

```
Require: state $\mathbf{x}$, initial guess $\underline{\mathbf{u}}^0$,
    and the scalar constants $i_{max}$, $L$, $c$, the constant vectors
    $\mathbf{l}$, $\mathbf{r}$, and the constant matrices $H$, $F$.
    set $\underline{\mathbf{w}} = \underline{\mathbf{u}}^0$
    for $i = 1 \to i_{max}$ do
        compute $\underline{\mathbf{u}}^i = P_U\left(\underline{\mathbf{v}}(\underline{\mathbf{w}})\right)$
        compute $\underline{\mathbf{w}} = \underline{\mathbf{u}}^i + c(\underline{\mathbf{u}}^i - \underline{\mathbf{u}}^{i-1})$
    end for
    return $\underline{\mathbf{u}}^i$
```

Numerical analysis

In the previous subsection we pointed out that for the FGM we can determine the maximum number of iterations i_{max} that guarantee an ϵ-suboptimal solution. It is important to note that i_{max} can be determined off-line. Next we discuss how to make a choice of ϵ that takes into account the numerical errors in the CPU computations and the quantization happening in the system actuators. In a practical environment, this choise might deliver a good trade-off between acceptable solutions and low number of iterations.

In Fig. 2, we represent the three main sources of numerical error in the generation of an optimal input to the system. Roughly speaking, first the FGM introduces a truncation error, as defined in (6). Afterwards, the central processing unit (CPU) of our digital computer calculates a numeric approximation of the FGM (round-off error). Finally, the digital-to-analogue converter (DAC) of the system actuator rounds the CPU solution (quantization).

At the mathematical level, where numbers are represented exactly, we first introduce a truncation error as defined in (6). This error is caused by the algorithm used to solve (5) and not by the computational platform. We define an upper bound on the error of the ϵ-suboptimal solutions as

$$\delta \geq \|\underline{\mathbf{u}}^{i_{max}} - \underline{\mathbf{u}}^*\|_\infty \geq \|\mathbf{u}_0^{i_{max}} - \mathbf{u}_0^*\|_\infty. \qquad (10)$$

From (6), (7), and $\|\underline{\mathbf{u}}^{i_{max}} - \underline{\mathbf{u}}^*\|_2 \geq \|\underline{\mathbf{u}}^{i_{max}} - \underline{\mathbf{u}}^*\|_\infty$, relation (10) holds if

$$\epsilon \leq \frac{1}{2}\mu\delta^2. \qquad (11)$$

This means that, if we can find a reasonable value for δ, we can use it to select a value for ϵ that might also be reasonable. We will see that finding such δ is for many practical cases straightforward.

The truncation error is introduced by the FGM in Fig. 2. We denote $\|\mathbf{u}_0^{i_{max}} - \mathbf{u}_0^*\|_\infty = \max|\mathbf{u}_0^{i_{max}}[i] - \mathbf{u}_0^*[i]|$, $i = 1, \ldots, p$, simply as $|u - u^*|$, and the corresponding lower and upper bounds as l and r, respectively. An additional numerical error stems from the execution of the algorithm in a machine with limited numerical precision (represented in Fig. 2 by the box CPU). The FGM might also increase the error if inexact gradient information is used (see [16]). Formally the CPU process is represented by a function $C : u \mapsto \overline{u}$, $\overline{u} \in \mathcal{C}$. The set of machine numbers $\mathcal{C}$ depends

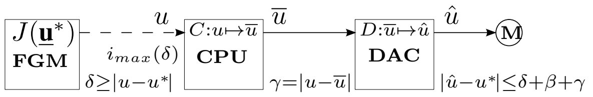
$J(\underline{\mathbf{u}}^*)$
FGM
u
$i_{max}(\delta)$
$\delta \geq |u - u^*|$
$C : u \mapsto \overline{u}$
CPU
$\overline{u}$
$\gamma = |u - \overline{u}|$
$D : \overline{u} \mapsto \hat{u}$
DAC
$\hat{u}$
M
$|\hat{u} - u^*| \leq \delta + \beta + \gamma$

Fig. 2. Propagation of numeric errors

on the numeric representation (fixed point, floating point, etc.) and the number of bits it uses. The accumulation of rounding errors after i_{max} iterations is represented by $\gamma = |u - \overline{u}|$, where $\gamma \in \mathbb{R}$. Finding an upper bound for γ requires an extensive analysis of the algorithm and the computing platform, which is beyond the scope of this work.

The next block in Fig. 2 is the DAC, placed between the CPU and the actuator M. The actual physical analogue output is not of interest here. We consider a linear DAC, which is commonly found in practice. We loosely use the term DAC to refer to other technologies which rely on quantization and for which this analysis holds (e.g. a pulse-width modulator). The DAC can be represented as a scalar quantisation function $D : \mathcal{C} \to \mathcal{D}$ that rounds a number $\overline{u}$ in the CPU representation, to the nearest number $\hat{u}$ in the DAC set $\mathcal{D} = \{\hat{u} \in \mathcal{U} \mid \hat{u} = s\rho(r - l) + l\}$, where $\rho = 2^{-B}$ is the DAC resolution, $B \in \mathbb{Z}$ is the number of bits of the DAC, $s \in \mathbb{Z}$, $0 \leq s < 2^B$ is the digital value to be converted, and $r - l$ is the analogue range of the converter. The absolute rounding error for the DAC is

$$\beta = \frac{1}{2}(r - l)\rho \geq |\overline{u} - \hat{u}|, \qquad (12)$$

which implies that even if $\overline{u} = u^*$, the input applied to the system will be $\hat{u} = u^* \pm \beta$. Equivalently, we can write $|\hat{u} - u^*| \leq |\overline{u} - u^*| + \beta$. On the one hand, if we consider the ideal case in which $\gamma = 0$, and we set $\delta = \beta$, the choice (11) guarantees that the difference between the value applied to the system $\hat{u}$ and the solution u^* is not greater than one quantization level 2β of the DAC. On the other hand, in a realistic scenario with a stable algorithm, a well-conditioned problem, and a DAC with much lower numeric precision than that of the CPU, we have $\beta \gg \gamma$. In that scenario, requiring $\delta = \beta$ and (11) the relation $|\hat{u} - u^*| \leq 2\beta$ provides a good approximation of the overall error. Clearly the overall tolerable error, and therefore the actual choice of ϵ, depends on several requirements and side constraints: stability of the system, performance, and available CPU time. Issues related to stability are subject of future work.

IV. Experimental data

So far we have discussed how to choose the maximum number of iterations i_{max} based on the numerical properties of the FGM, the CPU implementation of the algorithm and quantization at the input of the actuators. However, implementing an MPC scheme on an embedded platform usually also implies that one has to fulfill rather strict requirements in terms of computation time as well as in terms of memory demand.

Computational requirements and complexity

The online computation of the Fast Gradient algorithm does not require arithmetic division, nor matrix-matrix mul-

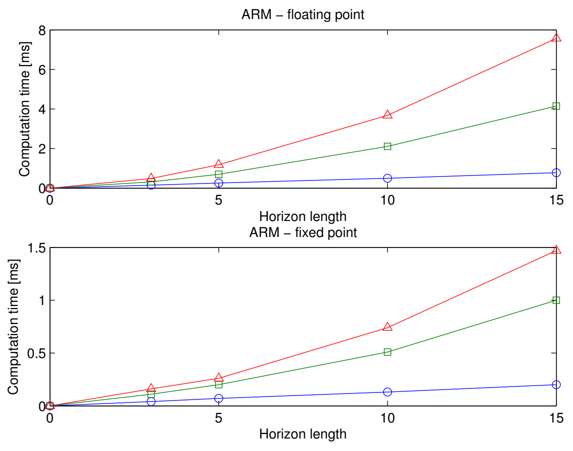

Fig. 3. Computation time vs. horizon length, for number of iterations $i_{max} = 0$ ($\circ$), $i_{max} = 1$ ($\square$), and $i_{max} = 2$ ($\triangle$). From top to bottom: floating-point arithmetic, fixed-point arithmetic using ARM instruction set.

tiplications. The term $F\mathbf{x}$ in (8a) is constant, so it needs to be computed only once per evaluation of the MPC. The computational complexity for this operation is $O(Npn)$. The main computational burden comes from the term $H\underline{\mathbf{w}}$, which has complexity $O((Np)^2)$ and needs to be computed i_{max} times per sample. Figure 3 shows the relation between the computation time required to solve the QP problem and the horizon length for different numeric representations. $i_{max} = 0$ represents the computation of $F\mathbf{x}$ alone. The depicted results correspond to the average of 100 computations.

The CPU utilization of the inputs and outputs RT tasks can be neglected. From our CPU utilization limit $\eta \approx 78\%$, we can safely assume that we can meet our RT deadlines if we compute the controller task (the FGM) in less than 3 ms. As illustrated in Fig. 3 the floating-point operations show a linear increase of computational time with respect to the number of iterations (i.e. $t_{i_{max}} \approx t_{i_{max}=0} + i_{max}(t_{i_{max}=1} - t_{i_{max}=0})$). We use this property to extrapolate the data for $i_{max} \geq 3$. The fixed-point computations do not follow this trend precisely, however we extrapolate in a similar way to cautiously make some remarks later in this section.

Memory requirements

As explained in Section II, we only have 64kB of memory available for the implementation of our MPC controller. In general, it is difficult to know beforehand how much memory is available for user data due to different factors (compiler flags, RTOS memory management, the code itself, etc.). We have empirically determined that around 16kB of RAM are available for user data (volatile and non-volatile). Therefore, we determine the amount of memory required for the variables of the FGM, which is a function of the number of states n, number of inputs p, horizon length N, and the number of bytes required to represent a number b. Algorithm 1 can be logically split in two memory blocks: the variables computed off-line (F and H), and the variables computed online (mainly the output of the algorithm $\underline{\mathbf{u}}^{i_{max}}$). To store the off-line variables $H \in \mathbb{R}^{Np \times Np}$ (not taking into account

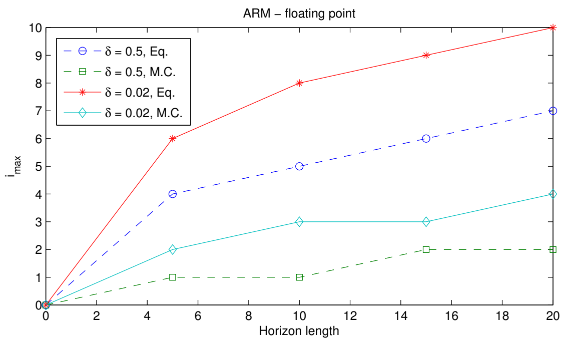

Fig. 4. Theoretical vs. Practical maximum number of iterations i_{max}. The theoretical bounds were computed using eq. (9) and are denoted in the legend as 'Eq.'. The practical ones were estimated with a Monte Carlo experiment, and are denoted as M.C.

the symmetry of H), and $F \in \mathbb{R}^{Np \times n}$ we need in bytes $b_A = (Np + n)Npb$.

The memory required for the online computations depends on how the program has been written. We introduce an integer a to represent the number of variables needed to store intermediate computations. In our particular implementation, we need to allocate space for $\underline{\mathbf{u}}^i$, $\underline{\mathbf{u}}^{i+1}$, $\underline{\mathbf{w}}$, $F\mathbf{x}$, $\underline{\mathbf{u}}^0$, and a auxiliary variables (all of them are arrays with Np elements of size b bytes). Therefore for the online computations we need in bytes $b_B = (5 + a)Npb$. We use single precision floating-point numbers or 32-bit fixed-point numbers, which implies $b = 4$ in both cases. To compute the algorithm in C we require $a = 2$ auxiliary variables to store intermediate computations. For our test system we empirically determined the FGM with a horizon lengths of at most $N = 28$ fits into the available free memory.

Numerical convergence

In Section III, we proposed using the numerical precision of the DAC to set the maximum number of iterations of the FGM. In our case we have a PWM with range 200% and $\beta = 0.5\%$ (which corresponds to $\rho = 2^{-8}$ with the binary range only partially used). We additionally consider a different scenario of practical relevance: we assume our system has $\rho = 2^{-12}$, which for the same range corresponds to $\beta \approx 0.02\%$. We computed i_{max} using (9) for the two values of ρ, with $\delta = \beta$ and $\epsilon = \frac{1}{2}\mu\delta^2$. Furthermore, we run an off-line Monte Carlo experiment for each ϵ that stops iterating once the required ϵ is reached. We use 1×10^6 uniformly distributed random states vectors $\mathbf{x}$ for different horizon lengths. In Fig. 4, we show the maximum i_{max} for each Monte Carlo experiment together with the values predicted by (9) for different horizons.

Table I is based on extrapolated data from Fig. 3 and Fig. 4. We see that a Monte-Carlo-based number of iterations increases the maximum real-time feasible horizon length by 50% (worst case). The use of warm-starting can further decrease i_{max} in many applications. Furthermore, for the case of $N = 20$ we required around 8 kB of memory. The main drawback of fixed-point arithmetic is lower numeric

TABLE I

REAL-TIME FEASIBLE HORIZON LENGTHS

Precision	Arithmetic	N_{max} Eq. (9)	N_{max} M.C.
$\delta = 0.5$	fixed point	12	20
$\delta = 0.5$	floating point	6	12
$\delta = 0.02$	fixed point	10	15
$\delta = 0.02$	floating point	4	7

precision compared to single or double precision floating-point arithmetic. The Hessian computed using the matrices in Appendix I has a small condition number, which allows the use of low precision numerics. The notation $Q15.16$ represents 32-bit binary fixed point arithmetic, with 15 integer bits plus 1 sign bit and 16-bit fractional bits. We computed the difference between the solution computed by the algorithm using $Q15.16$ arithmetic and the double precision arithmetic solution (computed in Matlab) ($\gamma \approx |\overline{u}_{double} - \overline{u}_{fixed}|$). Using 1×10^6 random state vectors for $N = 5, 10, 15, 20$ and the corresponding $i_{max} = 2, 3, 3, 4$ in Fig. 4, for a $\delta = \beta = 0.02$, γ was below 0.25β in all cases. The assumption $\beta \gg \gamma$ does not hold in this case, and implies $|\hat{u} - u^*| \leq \delta + \beta + \gamma \approx 2.25\beta$.

Results and Discussion

So far we have found the computational and memory demands as well as the numerical characteristics of an FGM based MPC implemtation on a low-cost embedded processor. As result we see that even for a problem with 8 states, 2 inputs, and a horizon length of 20 steps the solution is computed efficiently on a low-cost platform. We employed a hard real-time 4-milisecond deadline and an 48-MHz microcontroller based on the ARM7 architecture. Furthermore, the algorithm shows good numerical properties. It converges to a solution using 32-bit fixed-point arithmetic with 16 fractional bits with a small numeric error (below one quantization level of the DAC).

The theoretical upper bound for the number of iterations required to find a solution within a specified precision might be too high for very restricted embedded computers. In such cases, a heuristic upper bound should be used. We propose choosing the suboptimality level based on the hardware characteristics of the system actuators, instead of choosing it arbitrarily based on the value of the cost function or based on the performance of a simulation. The latter case can be misleading, and can predict an excessive number of iterations. We provided a relation to exactly compute the memory required for storing the FGM variables. The case with $N = 20$ required around 8 kB of memory.

V. CONCLUSIONS

We discussed the implementation related aspects of MPC on embedded systems relying on a case study of an MPC scheme based on a Fast Gradient method. We propose to not only account for the properties of the employed algorithm but also the for the CPU round-off error as well as for the quantization happening in the system actuators. Relying on this, a maximum number of iterations of practical use can be chosen for the optimization algorithm. Additionally, one can conclude from the considered case study that the memory as well as the computational demand of an MPC implementation are decisive for real-time use on low-cost embedded systems.

Furthermore we have shown by means of a real system implementation that the FGM is well suited for use on low-cost embedded platforms because it is both memory and computationally efficient even for long horizons. We used the fact that the FGM allows the off-line determination of the maximum number of iterations to run the MPC scheme in hard real time. Future work will focus on the stability issues related to numerical errors as well as on a comparison with other online MPC schemes based on different optimization algorithms.

APPENDIX I
SYSTEM MATRICES

We developed a continuous-time model for our system $\dot{\mathbf{x}}_c = A_c\mathbf{x}_c + B_c\mathbf{u}_c$, $\mathbf{y}_c = C\mathbf{x}_c$. A zero-order hold discretisation on the inputs was used to get the system in (1).

$$A_c = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 40 & -12 & 12 & 0 & 0 & 0 & 0 \\ 0 & 40 & 3.5 & -7 & 0 & 0 & 3.5 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 40 & 0 & 12 & 0 & 0 & -12 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix},$$

$$B_c^T = \begin{bmatrix} 0 & 0 & 1.6 & -0.45 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -0.45 & 0 & 0 & 1.6 & 0 \end{bmatrix}, C_c = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

$$\mathbf{x}_c^T = \begin{bmatrix} \theta_{w1} & \theta_b & \dot{\theta}_{w1} & \dot{\theta}_b & \theta_{iw1} & \theta_{w2} & \dot{\theta}_{w2} & \theta_{iw2} \end{bmatrix},$$

$$\mathbf{u}_c^T = \begin{bmatrix} u_{pwm1} & u_{pwm2} \end{bmatrix}, \mathbf{y}_c^T = \begin{bmatrix} \theta_{m1} & \dot{\theta}_b & \theta_{m2} \end{bmatrix},$$

with the notation w: wheel, b: body, i: integration, m: motor, 1: left, 2: right (refer to Fig. 1). For example, θ_{w1} refers to the rotation angle of the left wheel. The weighting matrices in (3) are

$$Q = \text{diag}(\begin{bmatrix} 1 & 6{\times}10^5 & 1 & 400 & 250 & 1 & 1 & 250 \end{bmatrix}), R = \text{diag}(\begin{bmatrix} 500 & 500 \end{bmatrix}),$$

P is selected as the discrete-time infinite horizon solution of the Ricatti equation associated to these matrices.

REFERENCES

[1] J. Rawlings and D. Mayne, *Model predictive control: Theory and design.* Nob Hill Pub., 2009.

[2] A. Bemporad, M. Morari, V. Dua, and E. Pistikopoulos, "The explicit linear quadratic regulator for constrained systems* 1," *Automatica*, vol. 38, no. 1, pp. 3–20, 2002.

[3] S. Richter, C. Jones, and M. Morari, "Real-time input-constrained MPC using fast gradient methods," in *Proceedings of the 48th IEEE Conference on Decision and Control and the 28th Chinese Control Conference, CDC/CCC.* IEEE, 2009, pp. 7387–7393.

[4] Y. Nesterov, "A method of solving a convex programming problem with convergence rate O (1/k2)," in *Soviet Mathematics Doklady*, vol. 27, no. 2, 1983, pp. 372–376.

[5] S. Richter, S. Mariethoz, and M. Morari, "High-speed online MPC based on a fast gradient method applied to power converter control," in *American Control Conference (ACC)*, 2010, pp. 4737–4743.

[6] M. Kögel and R. Findeisen, "A fast gradient method for embedded linear predictive control," in *Proceedings of the 18th IFAC World Congress*, 2011.

[7] Y. Yamamoto, "NXTway-GS Model-Based Design-Control of self-balancing two-wheeled robot built with LEGO Mindstorms NXT," www.pages.drexel.edu/.../NXTway-GS%20Model-Based_Design.pdf, 2008.

[8] "Fast embedded MPC on Lego NXT," http://ifatwww.et.uni-magdeburg.de/syst/about_us/people/zometa/, last accessed 09/2011.

[9] "nxtOSEK/JSP ANSI C/C++ with OSEK/ITRON RTOS for LEGO MINDSTORMS NXT," http://lejos-osek.sourceforge.net/, last accessed 09/2011.

[10] "OSEK VDX Portal," http://portal.osek-vdx.org/.

[11] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[12] J. Primbs and V. Nevistic, "Constrained finite receding horizon linear quadratic control," in *Proceedings of the 36th IEEE Conference on Decision and Control*, vol. 4, 1997, pp. 3196–3201.

[13] D. Mayne, J. Rawlings, C. Rao, and P. Scokaert, "Constrained model predictive control: Stability and optimality," *Automatica*, vol. 36, pp. 789–814, 2000.

[14] J. Maciejowski, *Predictive control: with constraints*. Pearson education, 2002.

[15] Y. Nesterov, *Introductory lectures on convex optimization: A basic course*. Kluwer Academic Publishers, 2004.

[16] O. Devolder, F. Glineur, and Y. Nesterov, "First-order Methods of Smooth Convex Optimization with Inexact Oracle," Available online at http://www.optimization-online.org.