Abstract—This paper presents a method to generate C code from MATLAB code applied to a nonlinear model predictive control (NMPC) algorithm. The C code generation uses the MATLAB Coder Toolbox. It can drastically reduce the time required for development compared to a manual porting of code from MATLAB to C, while ensuring a reliable and fairly optimized code. We present an application of code generation to the numerical solution of nonlinear optimal control problems (OCP). The OCP uses a sequential quadratic programming algorithm with multiple shooting and sensitivity computation. We consider the problem of glucose regulation for people with type 1 diabetes as a case study. The average computation time when using generated C code is 0.21 s (MATLAB: 1.5 s), and the maximum computation time when using generated C code is 0.97 s (MATLAB: 5.7 s). Compared to the MATLAB implementation, generated C code can run in average more than 7 times faster.

I. INTRODUCTION

In optimal control, the underlying optimization problem is usually constrained by the dynamics of a system described by nonlinear ordinary differential equations (ODEs), differential algebraic equations (DAEs), or even partial differential equations (PDEs). Nonlinear model predictive control (NMPC) is a receding horizon control technology that repeatedly solves open-loop nonlinear optimal control problems (OCPs). At every sample, it implements the computed optimal input associated to the current time period. Research groups are investigating a large number of potential applications of NMPC, spanning from real-time applications that need to be implemented on an embedded device where the OCP has to be solved within milliseconds [1]–[3], to very large-scale applications requiring high performance computing, for instance for oil recovery [4]–[6]. Numerous software tools using code generation for solving nonlinear OCPs have been considered. The ACADO toolkit is an open-source software for control and nonlinear optimization [7]. CasADi is a symbolic package capable of C code generation [8]. JModelica.org is a tool for large-scale dynamic optimization problems [9]. CVXGEN generates a custom C code to solve convex optimization problems [10].

MATLAB provides a user-friendly and simple environment useful for prototyping and software development. However, its code usually runs slower than compiled code and cannot be ported to all architectures. Conversely, C is a programming language used in a large number of applications, but the development of optimized and reliable C code is a time-consuming task.

In this paper, we present a new method for solving nonlinear OCPs arising in NMPC using the MATLAB Coder toolbox. The MATLAB Coder toolbox does not require any knowledge in C programming and only requires minor adaptations of the MATLAB code. The generated code is tailored to a specific OCP, such that the routines and the memory allocation are optimized. We provide comprehensive guidelines to generate a C code from MATLAB using the MATLAB Coder toolbox.

We present the problem of closed-loop control of blood glucose in people with type 1 diabetes, also referred as the artificial pancreas (AP), as a case study. The AP comprises a glucose sensor measuring glucose levels frequently, a control algorithm, and an insulin pump. It has the potential to improve the quality of life and reduce the burden of insulin therapy management. MPC and NMPC-based control algorithms are among the most popular for the design of the AP [11]–[14]. For this application, the control algorithm has to be implemented on a mobile platform, such as a smartphone, or even on a chip. Fig. 1 illustrates the AP.

The paper is structured as follows. Section II states the continuous-time OCP and presents a numerically tractable discrete-time approximation. Section III outlines the sequential quadratic program (SQP) algorithm. Section IV demonstrates the capabilities of the MATLAB Coder toolbox.
and provides the required steps to obtained a compiled C code from MATLAB. Section V presents the application to the diabetes problem and highlights the difference in running time between MATLAB and the generated C code. A summary of the main contributions of the paper is provided in Section VI.

II. PROBLEM FORMULATION

We consider the bound constrained continuous-time OCP

\[
\min_{\{x(t),u(t)\}_{t_0}^{t_f}} \phi = \int_{t_0}^{t_f} g(x(t), u(t)) dt + h(x(t_f)),
\]

s.t.

\[
x(t_0) = x_0, \quad \dot{x}(t) = f(x(t), u(t), d(t)), \quad t \in [t_0, t_f],
\]

\[
u_{\min} \leq u(t) \leq u_{\max}, \quad t \in [t_0, t_f],
\]

where \(x(t) \in \mathbb{R}^{nx}\) is the state vector, \(u(t) \in \mathbb{R}^{nu}\) are the manipulated inputs, and \(d(t) \in \mathbb{R}^{nd}\) are known disturbances. \(\dot{x}(t) = f(x(t), u(t), d(t))\) represents the model equations. The initial time, \(t_0\), and the final time, \(t_f\), are fixed parameters. The initial state, \(x_0\), is a known parameter in (1). The inputs are bound constrained and must be in the interval \(u(t) \in [u_{\min}, u_{\max}]\). The objective function is stated with a stage cost term, \(g(x(t), u(t))\), and a cost-to-go term, \(h(x(t_f))\).

A. Zero-order hold parametrization

In general, the continuous-time bound constrained problem (1) is not tractable and is solved numerically by discretization using a zero-order hold (ZOH) parametrization of the manipulated variables, \(u(t)\), and the known disturbance variables, \(d(t)\). We sample the time interval, \([t_0, t_f]\), into \(N\) equidistant intervals each of length \(T_n\). Let \(\mathcal{N} = \{0, 1, \ldots, N-1\}\) and \(t_k = t_0 + kT_n\) for \(k \in \mathcal{N}\). The ZOH parametrization on \(u(t)\) and \(d(t)\) yields

\[
u(t) = u_k, \quad t_k \leq t < t_{k+1}, \quad k \in \mathcal{N},
\]

\[
d(t) = d_k, \quad t_k \leq t < t_{k+1}, \quad k \in \mathcal{N}.
\]

Using this ZOH restriction on the inputs, the bound constrained continuous-time Bolza problem (1) may be expressed as

\[
\min_{\{x_{k+1},u_{k+1}\}_{k=0}^{N-1}} \phi = \sum_{k=0}^{N-1} G_k(x_k, u_k, d_k) + h(x_N),
\]

s.t.

\[
b_k := F_k(x_k, u_k, d_k) - x_{k+1} = 0, \quad k \in \mathcal{N},
\]

\[
u_{\min} \leq u_k \leq u_{\max}, \quad k \in \mathcal{N}.
\]

The discrete-time state transition function is

\[
F_k(x_k, u_k, d_k) = (x(t_{k+1}) : \dot{x}(t) = f(x(t), u_k, d_k), \quad x(t_k) = x_k),
\]

and the discrete time stage cost is

\[
G_k(x_k, u_k, d_k) = \left\{ \int_{t_k}^{t_{k+1}} g(x(t), u_k) dt : \dot{x}(t) = f(x(t), u_k, d_k), \quad x(t_k) = x_k \right\}.
\]

III. THE SEQUENTIAL QUADRATIC PROGRAM ALGORITHM

In this section, we describe a multiple-shooting based SQP algorithm [15]–[18]. The SQP algorithm is used for the numerical solution of (1). The quadratic sub-problems arising in the SQP algorithm are efficiently solved using Riccati iterations [19]–[22]. We use a fourth order Runge-Kutta scheme with fixed stepsize for numerical solution of the differential equation model and for computation of the sensitivities.

A. SQP algorithm

We define the parameter vector, \(p\), as

\[
p = [u_0, x_1', u_1', x_2', \ldots, x_{N-1}', u_{N-1}', x_N']',
\]

and the disturbance vector as

\[
d = [d_0', d_1', \ldots, d_{N-1}'].
\]

We can formulate the discrete-time dynamics as

\[
b(p) = b(p, x_0, d) =
\]

\[
\begin{bmatrix}
F_0(x_0, u_0, d_0) - x_1 \\
F_1(x_1, u_1, d_1) - x_2 \\
\vdots \\
F_{N-1}(x_{N-1}, u_{N-1}, d_{N-1}) - x_N
\end{bmatrix}.
\]

The objective function is

\[
\phi(p) = \phi(p, x_0, d) = \sum_{k=0}^{N-1} G_k(x_k, u_{k:d}) + h(x_N).
\]

Let \(c(p)\) denote the bound constraints, i.e.

\[
c(p) =
\]

\[
\begin{bmatrix}
 u_0 - u_{\min} \\
u_1 - u_{\min} \\
\vdots \\
u_{N-1} - u_{\min} \\
u_{\max} - u_0 \\
u_{\max} - u_1 \\
\vdots \\
u_{\max} - u_{N-1}
\end{bmatrix}.
\]

Using these notations, we can reformulate the discrete-time Bolza problem (3) as a constrained optimization problem in standard form

\[
\min_p \phi = \phi(p),
\]

s.t.

\[
b(p) = 0,
\]

\[
c(p) \geq 0.
\]

The Lagrangian of (11) is

\[
\mathcal{L}(p, y, z) = \phi(p) - y' b(p) - z' c(p).
\]
The nonlinear problem (11) can be solved iteratively using a SQP algorithm. In each iteration, (11) is locally approximated by the quadratic program (QP)

$$\min_{\Delta p} \frac{1}{2} \Delta p' W^k \Delta p + \nabla_p \phi(p^k) \Delta p,$$

s.t. 

$$[\nabla_p b(p^k)]' \Delta p = -b(p^k),$$

$$[\nabla_p c(p^k)]' \Delta p \geq -c(p^k),$$

where $W^k$ is an approximation of the Hessian of the Lagrangian. It is obtained by the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm [23].

The first order Karush-Kuhn-Tucker (KKT) conditions of the constrained nonlinear optimization problem (11) are

$$\nabla_p L(p, y, z) = \nabla_p \phi(p) - \nabla_p b(p)y - \nabla_p c(p)z = 0,$$

(14a)

$$b(p) = 0,$$

(14b)

$$c(p) \geq 0,$$

(14c)

$$z \geq 0,$$

(14d)

$$c_i(p) = 0 \vee z_i = 0 \ \forall i.$$  

(14e)

These conditions are used to test the convergence of the SQP algorithm.

B. Gradient computation

The most time consuming computations in the SQP algorithm are the computation of the objective function $\phi(p)$, the derivatives of the objective function $\nabla_p \phi(p)$, the dynamics $b(p)$, and the sensitivities, $\nabla_p b(p)$. $b(p)$ and $\phi(p)$ are computed by evaluation of (4) and (5), respectively. Consequently, the equality constraints and their sensitivities with respect to states and inputs are

$$b_k = F_k(x_k, u_k, d_k) - x_{k+1},$$

(15a)

$$\nabla_{x_k} b_k = \nabla_{x_k} F_k(x_k, u_k, d_k) = S_{x_k}(t_{k+1})' = A_k',$$

(15b)

$$\nabla_{u_k} b_k = \nabla_{u_k} F_k(x_k, u_k, d_k) = S_{u_k}(t_{k+1})' = B_k',$$

(15c)

$$\nabla_{x u_k} b_k = -I,$$

(15d)

where $x(t_{k+1}) = F(x_k, u_k, d_k)$, and the sensitivities (15b-(15c)) follow the systems of ordinary differential equations

$$\dot{x}(t) = f(x(t), u_k, d_k),$$

(16a)

$$\dot{S}_{x_k}(t) = \left( \frac{\partial f}{\partial x} (x(t), u_k, d_k) \right) S_{x_k}(t),$$

(16b)

$$\dot{S}_{u_k}(t) = \left( \frac{\partial f}{\partial u} (x(t), u_k, d_k) \right) S_{u_k}(t) + \left( \frac{\partial f}{\partial u} (x(t), u_k, d_k) \right),$$

(16c)

with the initial conditions $x(t_k) = x_k$, $S_{x_k}(t_k) = I$, and $S_{u_k}(t_k) = 0$. The stage cost and the associated derivatives are computed as

$$G_k = G_k(x_k, u_k, d_k) = \int_{t_k}^{t_{k+1}} g(x(t), u_k, d_k) dt,$$

(17a)

$$q_k = \nabla_{x_k} G_k = \int_{t_k}^{t_{k+1}} \left( \frac{\partial g}{\partial x} (x(t), u_k, d_k) \right) S_{x_k}(t) dt,$$

(17b)

$$r_k = \nabla_{u_k} G_k = \int_{t_k}^{t_{k+1}} \left[ \left( \frac{\partial g}{\partial u} (x(t), u_k, d_k) \right) S_{u_k}(t) \right. + \left. \left( \frac{\partial g}{\partial u} (x(t), u_k, d_k) \right) dt.$$  

(17c)

The derivatives $\nabla_{x_k} b_k$ and $\nabla_{u_k} G_k$ are computed for $\{x_k\}_{k=1}^{N-1}$ and $k \in N$. These derivatives are not computed for $x_0$ as $x_0 \notin p$, i.e. $x_0$ is a fixed parameter of the optimization problem but not a decision variable. The derivatives $\nabla_{u_k} b_k$ and $\nabla_{u_k} G_k$ are computed for $k \in N$. The derivatives with respect to $x_N$ are

$$\nabla_{x_N} b_{N-1} = -I,$$

(18a)

$$p_N = \nabla_{x_N} \phi = \nabla_{x_N} h(x_N).$$

(18b)

Therefore, the gradients of the equality constraints $b_k$ with respect to the parameter vector $p$ can be written as

$$\nabla_p b = \begin{bmatrix} B_0 & A_1 \\ -I & B_1 \\ -I & A_2 \\ -I & B_2 \\ \vdots & \vdots \\ -I & A_{N-1} \\ -I & B_{N-1} \end{bmatrix}.$$

(19)

C. Numerical integration

In this paper, we use the classical explicit Runge Kutta solver of order 4 to numerically compute the system state transition (4), the stage cost (5), the sensitivities (15b-15c), as well as the stage cost derivatives (17b-17c). Given the state values $x_n$ at time $t_n$, the solution at the next step $x_{n+1}$ is given by

$$X_1 = x_n,$$

(20a)

$$X_2 = x_n + \frac{1}{2} hf(t_1, X_1),$$

(20b)

$$X_3 = x_n + \frac{1}{2} hf(t_2, X_2),$$

(20c)

$$X_4 = x_n + hf(t_3, X_3),$$

(20d)

$$x_{n+1} = x_n + \frac{1}{3} h \left( \frac{1}{2} f_1 + f_2 + f_3 + \frac{1}{2} f_4 \right).$$

(20e)

where $h = t_{n+1} - t_n$ is a fixed step size. The times where internal stages are computed are $t_1 = t_n, t_2 = t_3 = t_n + \frac{1}{2} h$ and $t_4 = t_n + h = t_{n+1}$. Let $f_i = f(t_i, X_i), i = 1, 2, 3, 4$, be the function evaluations at times $t_i$. 

329
D. Interior point algorithm

We use a structured primal-dual interior point algorithm for the solution of the constrained QP (13). We implement the centering step correction proposed by Mehrotra [24]. We use a Riccati recursion to compute the Newton iterations in the primal-dual interior point algorithm [19], [20], [25]. This factorization can be used to compute the optimal variation in the manipulated variables Δu_k, the optimal change in states variables Δx_{k+1}, and the Lagrange multipliers y_{k−1}. For most problems, Riccati recursion-based solvers are considered as the most computationally efficient method to solve the linear quadratic (LQ) sub-problem arising in the interior point method [26].

IV. C CODE GENERATION

Since version R2011a, MATLAB contains a toolbox for C code generation [27], [28]. Before that date, several code generation tools in C or Fortran have been developed for control applications [29], [30]. The C code generation creates either a MATLAB executable (mex) version of the code, a dynamic library (dll), a stand-alone C library or an executable. The C code can be ported to a number of platforms, such as embedded systems, smartphones, or can be used for high performance computing. The source code is also editable and can for instance be integrated in an existing code. The C code generation consists of the four following steps

1) Implement the SQP-based control algorithm as described in Section III.
2) Include the model and the sensitivity functions also detailed in Section III.
3) Generate the C code from MATLAB command line or via the dedicated graphical user interface.
4) Compile the C code.

We configured the coder for further optimization by deactivating the responsiveness to CTRL+C and graphics refreshing, removing runtime checks and disabling dynamic memory sizing, since these features are not required in our case [31, Section 8.2.4].

A. Example of generated C code

Listing 1 shows an example of generated C code for the matrix-matrix multiplication 𝐶 = 𝐴𝐵, assuming that 𝐴 and 𝐵 are 5 × 5 matrices. Setting the type (double precision floating point numbers) and the sizes of 𝐴 and 𝐵 (5 times 5) is done by adding the following four lines

```c
assert(isa(A, 'double'));
assert(isa(B, 'double'));
assert(all(size(A)==[5 5]));
assert(all(size(B)==[5 5]));
```

at the beginning of the MATLAB function. In this case, the C code generator chooses to use nested for loops.

Listing 2 shows an example of generated C code for the matrix-matrix multiplication 𝐶 = 𝐴𝐵, assuming that 𝐴 and 𝐵 are 1000 × 1000 matrices. Similarly, we declare the types and sizes of 𝐴 and 𝐵 in the MATLAB function by adding the lines

```c
assert(isa(A, 'double'));
assert(isa(B, 'double'));
assert(all(size(A)==[1000 1000]));
assert(all(size(B)==[1000 1000]));
```

at the beginning of the MATLAB function. Since it is now a large-scale problem, the BLAS level 3 routine dgemm is more efficient to perform the matrix-matrix multiplication, and therefore is preferred to the nested for loops.

Since the generated code is tailored for the specific OCP to solve, the MATLAB Coder Toolbox optimizes the choice of the routines and the memory usage to a specific problem.
V. NUMERICAL RESULTS

The simulations are performed using MATLAB R2016a installed on a Dell Latitude E6540 (Intel Core i7-4800MQ processor, 16 Gb RAM). We use the compiler gcc version 6.2.0 on Linux Mint 17.3 to compile the generated C code.

The sampling time is $T_s = 5$ minutes. We use a continuous-discrete extended Kalman filter for state estimation at each sample [32], [33].

The objective of the insulin administration is to compensate glucose excursions caused by meals and variations in endogenous glucose production and utilization. We use a penalty function defined as

$$g(G(t)) = \frac{1}{2} (G(t) - \bar{G})^2 + \frac{\kappa}{2} \max \{0, G_L - G(t)\}^2. \tag{21}$$

$G(t)$ is the blood glucose concentration, $\bar{G} = 5 \text{ mmol/L}$ is the target value for the blood glucose concentration, $G_L = 4 \text{ mmol/L}$ is a lower acceptable limit on the glucose concentration. The weight $\kappa$ is used to heavily penalize hypoglycemia. Fig. 2 illustrates the penalty function used in the simulations.

The objective function used in the simulations is

$$\bar{\phi} = \int_{t_0}^{t_f} g(x(t), u(t)) dt + \frac{\eta}{2} \sum_{k=0}^{N-1} \|\Delta u_k\|^2, \tag{22}$$

where $\Delta u_k = u_k - u_{k-1}$. This objective function has no cost-to-go function, i.e. $h(x(t_f)) = 0$, and contains a regularization term. $\frac{\eta}{2} \sum_{k=0}^{N-1} \|\Delta u_k\|^2$. The objective function (22) can be brought into the standard form (3a) using state augmentation formulated by [34].

A. Simulation results

We use the Medtronic Virtual Patient (MVP) model [35] and the developed multiple shooting SQP algorithm for (1) to compute the optimal insulin administration profiles for people with type 1 diabetes. We run a MATLAB and a generated C version of the multiple shooting SQP algorithm.

Fig. 3 shows the glucose, insulin and meal profiles, as well as the CPU time for each time sample. We assume that the patient has three meals: a 75g carbohydrates (CHO) breakfast at 6AM, a 100g CHO lunch at 12PM and a 75g CHO dinner at 6PM. The meals are not anticipated, i.e. they are announced to the MPC only at mealtimes. The average computation time when using generated C code is 0.21 s, versus 1.5 s for the MATLAB code, and the maximum computation time when using generated C code is 0.97 s, versus 5.7 s for the MATLAB code. Thus, the generated C code provides in average a speedup of more than 7 times compared to the MATLAB implementation.

B. Discussion

MATLAB uses a number of optimized built-in functions, so it is not guaranteed that the generated code will provide any speedup if the initial code heavily relies on these built-in functions. In our OCP, the numerical integration routine for computation of the model dynamics, the objective function and the sensitivities represent the main computational workload in the local SQP algorithm, which accounts for approximately 70% of the total CPU time. Since it has a large number of function evaluations, it will benefit the most from code generation. Conversely, the interior point algorithm represents uses almost all the remaining CPU time.

The purpose of this paper is to compare the MATLAB implementation with the generated C code. Further optimization of the code can be obtained by using warm starts [1], [36]. Nevertheless, these optimizations would not benefit during
mealtimes due to the high disturbance level caused by the meal.

VI. CONCLUSION

In this paper, we presented a simple way to generate C code from Matlab with application to MPC. The generated C code combines the convenient development of MATLAB and the efficiency and portability of C code. It provides a convenient and efficient solution for the design and the implementation of optimal control algorithms, for instance to embedded systems. Moreover, the code is tailored to the problem size and can be used to solve small scale as well as large scale OCPs. An application to a case study (the blood glucose regulation in people with T1D) shows a significant speedup between the Matlab implementation and the generated C code.

REFERENCES


