Auto-Code Generation for Fast Embedded Model Predictive Controllers

Jonathan Currie
Electrical and Electronic Engineering
AUT University
Auckland, New Zealand
Email: jocurrie@aut.ac.nz

Arrian Prince-Pike
Electrical and Electronic Engineering
AUT University
Auckland, New Zealand

David I. Wilson
Industrial Information and Control Centre
AUT University
Auckland, New Zealand

***Abstract*—The implementation of Model Predictive Controllers (MPC) on low-cast hardware such as micro-controllers has been traditionally hampered by the high computing and associated memory demands of the algorithm. This paper describes a completely automatic way to implement an MPC controller on embedded hardware starting from a dynamic model in MATLAB. The resultant controller runs stand-alone on the embedded hardware, is extremely fast, exhibits a modest memory footprint and best of all, requires no particular embedded programming experience from the user.**

I. Introduction

Model Predictive Control, or MPC, is a one of the most successful control advanced algorithms in common use today [1]. Due to its ability to handle multivariable systems, explicitly take into account equipment and state constraints, the algorithm is popular in applications where good control is economically vital (chemical plants, utility systems), or good control is needed for safe operations (aircraft systems). However, in many of these cases the process time constants are modest, and the support available in terms of people and resources is considerable.

Our specific research is in the area of embedded MPC, taking the benefits of constrained, multivariable control and applying it to small agile systems with fast dynamics, such as UAVs [2], aircraft [3], spacecraft [4], and a multitude of other systems. There is however a large hurdle to implementing MPC on an embedded platform; the significant computational requirements of the algorithm, as detailed in [5], [6], which require state-of-the-art algorithms and performance hardware.

In this paper we present our *practical* framework for automatically synthesizing MPC controllers that are fast (sampling rates in the kHz), that posses a small memory footprint (KB), and are modestly priced (≈$100) using MATLAB and a custom auto-coding toolset. Rather than target specialized hardware such as FPGAs [7] with their inherently long compilation times (making implementation tuning near impossible), we have opted for simple microcontrollers incorporating a floating point unit. We have also opted to avoid the explicit MPC formulation [8], which while far superior in speed (sample rates approaching MHz), requires prohibitively large memory requirements even for tiny problems. Therefore we do not purport to have the fastest embedded MPC, nor the smallest. However we have yet to see another implementation that is competitive to ours for applications that employ five to ten inputs and outputs using horizons from 10 to 20 operating at kHz sampling rates with hardware costing less than say $150.

II. Model Predictive Control

MPC is a multivariable control strategy that optimally computes control moves that not only aim to keep the process output at the setpoint, but also to keep it within predetermined system limits. This is achieved by solving a constrained optimization problem at each sample, which delivers a future sequence of control moves, $\mathbf{\Delta u}$, over the immediate future control horizon, N_c, such that the weighted sum of the squared deviations between the predicted output, $\hat{\mathbf{y}}$ and setpoint $\mathbf{y}^\star$ and control moves

$$J = \sum_{j=1}^{N_p} ||\gamma \hat{\mathbf{y}}_{k+j|k} - \mathbf{y}^\star_{k+j|k}||^2 + \sum_{j=1}^{N_c} ||\lambda \Delta \mathbf{u}_{k+j|k}||^2 \quad (1)$$

is minimised. Equation 1 is the standard quadratic cost function of a MPC controller, where the prediction, N_p, and control, N_c, horizons and weighting vectors γ and λ are important tuning parameters. The objective function is constrained by linear plant dynamics and possible linear output, input, and input rate constraints. Rearranging and substituting system matrices the objective function and constraints can be re-written as

$$\begin{aligned} \min_{\Delta \mathbf{u}} J &= \frac{1}{2}\Delta \mathbf{u}^T \mathbf{H} \Delta \mathbf{u} + \mathbf{f}^T \Delta \mathbf{u} \\ \text{subject to: } & \mathbf{A}\Delta \mathbf{u} \leq \mathbf{b} \end{aligned} \quad (2)$$

which is a standard Quadratic Program (QP) to be solved at each sample time [9].

A. Quadratic Programming Solvers

As part of our work in MPC we have also developed a number of algorithms for solving quadratic programs, with a focus on efficient memory use and high speed convergence. Our fastest algorithm is based on the work by Stephen Wright [10], which is a solver tailored for the quadratic problems arising from a MPC formulation. Specifically, the algorithm does not exploit sparsity, as the MPC problem is predominately dense, (as illustrated in Fig. 1 from [11]), as well as only

supporting inequality constraints, noting from Equation 2 that no equality constraints are required. We have further refined the algorithm by pre-factorizing where possible, using the most efficient factorizations (Cholesky Decomposition) when required, and heuristics for warm starting the algorithm. It has also been hand coded to optimize memory usage, allowing for a small and very fast QP solver. We have named this algorithm `quadwright` and have not found a faster QP solver for MPC.

To validate our claim, we benchmarked 500 quadratic programs resulting from 500 randomly generated MPC controllers across a range of competitive QP solvers. Each MPC controller was built with a random model with five states, two inputs, and two outputs, and with the prediction and control horizons set at 10 and 8 respectively. This configuration resulted in QPs with 16 decision variables and 104 constraints, which are typical sizes for the intended MPC implementations.

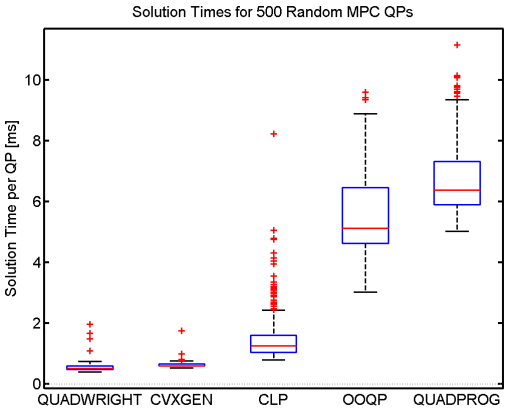

Fig. 1. Results of 5 QP solvers across 500 MPC QPs.

As can be seen in Figure 1, the `quadwright` algorithm out-performs all other solvers, including CVXGEN [12], a solver advertised as high speed and suitable for real time embedded applications. It is worth noting the CVXGEN solver required more than 6 minutes to generate, download and compile, will only work for a fixed problem size (given it is a completely unrolled implementation), and is much too large to fit in flash (greater than 1MB). In contrast the `quadwright` solver compiles in seconds, will work for any problem size, requires a few KB to store and yet is still faster.

The remaining solvers are not suitable for implementation on a microcontroller due to the large source libraries associated with them, as well as uncompetitive solve times. However they provide a clear comparison between fast solvers, and current state of the art for MPC.

B. The jMPC Toolbox

Our primary tool for MPC research is the jMPC Toolbox [11], [13], a MATLAB based toolbox our research team developed which allows a user to create, tune, and simulate linear model predictive controllers for both linear and nonlinear environments. By leveraging off the built-in functionality of MATLAB and the Control Systems Toolbox, jMPC allows an MPC controller to be created and simulated succinctly within a few lines of MATLAB code:

```
%Linearized Discrete Rotating Antenna Model
A = [1 0.1; 0 0.9];
B = [0; 0.0787];
C = [1 0];
D = zeros(1,1);
%Build jSS object
Plant = jSS(A,B,C,D,0.1);
Model = Plant; %no model-plant mismatch

%Horizons & Simulation Time
Np = 10; Nc = 3; T = 50;
%Setpoint & Constraints
setp = 2*ones(T,1);
con.u = [-2  2  1];
%Tuning Weights
uwt = 1; ywt = 3;

%Build MPC & Simulation
MPC1 = jMPC(Model,Np,Nc,uwt,ywt,con);
simopts = jSIM(MPC1,Plant,T,setp);
%Simulate & Plot Result
simresult = sim(MPC1,simopts);
plot(MPC1,simresult)
```

The above code snippet will generate Figure 2 which demonstrates MPC control of an unstable rotating antenna assembly from [14], which was built, simulated and plotted in less than a quarter of a second.

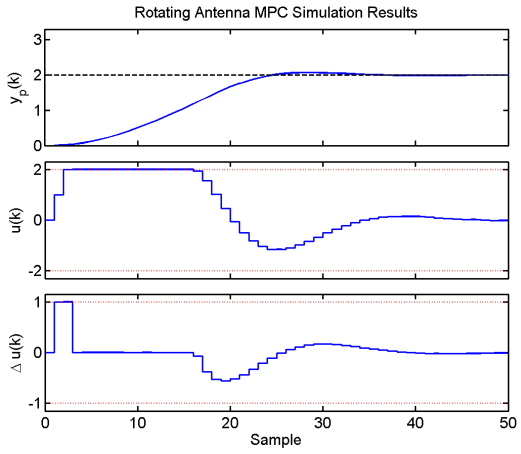

Fig. 2. MPC control of an unstable rotating antenna assembly.

As well as standard linear MPC simulation, the jMPC Toolbox provides advanced MPC functionality such as control move blocking, soft output constraints, state estimation to recover unmeasured outputs, measured disturbances for prediction modelling, setpoint look ahead as well as standard load and noise disturbance simulations. We have also developed high level routines for building and linearizing nonlinear models, so that the linearized model can be used as the controller model, and the nonlinear model as the plant, for more accurate simulations. The toolbox, including all functionality in this paper, is open-source and available free from our website, `www.i2c2.aut.ac.nz`.

III. Auto-Code Generation

Auto-coding tools for MATLAB are not new and there are several, now mature tools available, such as the MATLAB and SIMULINK Compiler. An auto-coded implementation for MPC was described in [6] where the authors used a Simulink model and the Real-Time Workshop to generate an embedded MPC controller. We experimented with this workflow but found it was too restrictive, primarily as the QP algorithm cannot be succinctly expressed in Simulink. Therefore we have opted to design our own toolset to enable hand optimized code to be generated, as well as enabling us to generate architecture specific code, customised memory management routines, and automatic testbenches.

As the target platforms for this research are microcontrollers and microprocessors, our auto coding tool generates ANSI C directly from the jMPC Toolbox. It does this in two ways: first by reading and copying a series of hand coded templates, containing the bulk of the mathematical and solver routines; and secondly by generating the required constants, variables, and application specific routines based on the controller being generated. The result is a combination of hand-optimized solvers written in C, auto-generated routines and auto-coded system matrices and vectors.

To generate the C code for an MPC controller designed in MATLAB, one simply calls `embed` on the jMPC object (noting the toolbox is written using the object orientated functionality of MATLAB). For the radar MPC controller presented in Figure 2, a C code MPC controller is generated in less than 50ms, together with an accurate estimate of the memory required by the controller. For this problem written in single precision, the utility routine `embed` estimates that 1KB of flash will be required by program constants, while 1.1KB of RAM will be required by program variables, which is suitable for implementation on a range of small microcontrollers.

The output of `embed` are five C source files and a common header file. The source files are divided as follows, a) constants declared for easy inspection, b) linear algebra routines, c) the quadratic program solver, d) the MPC engine and e) interface code for Processor In the Loop (PIL) simulation. Together, the 6 files contain all the algorithms and information required to implement high speed MPC on an embedded platform. It is worth noting that the code generated contains the full functionality of a jMPC MATLAB controller, including soft constraints, control move blocking and other functions previously described.

A. Code Verification

It is not sufficient to simply generate code without ensuring it performs exactly as the original algorithm. Therefore `embed` can optionally generate up to four testbench files to test both the QP solver and MPC engine, both in MATLAB and on the target microcontroller. By default, MATLAB Executable (MEX) interfaces to the auto-generated QP solver and the MPC engine will be automatically created and compiled, and a test run will be executed for each. This allows the code to be tested on the development computer and results inspected, prior to actual deployment on an embedded device. An example verification plot is shown in Figure 3 showing the difference between a MATLAB double precision simulation and a auto-coded, single precision simulation via a MEX library.

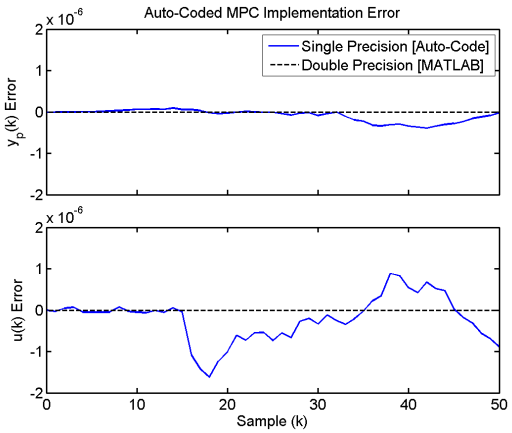

Fig. 3. Verification of the auto-coded MPC engine in MATLAB.

Once the code has been verified on the development computer, two automatically generated testbenches can be used to verify the code on the target device. The target QP testbench performs two functions, firstly running a test solution run of the first QP to be solved in the MPC simulation, then secondly re-running the solver to determine the worst case solution time. This is done by fixing the iterations run by the QP solver to the maximum (typically 30), as well as requiring a KKT step length calculation for every iteration. Using the maximum execution time of the QP, together with the execution of the MPC engine, we can determine the maximum achievable sampling rate for our embedded MPC implementation. The MPC testbench is run in a similar fashion, except that all simulation results are recorded so they can be inspected post execution.

B. Single Precision vs Double Precision

Hardware Floating Point Units (FPU) implementing IEEE 754 are not common on embedded devices such as small microcontrollers, thus typically computations are performed in fixed point (or integer) or by using software floating point libraries. There are however a few select small microcontrollers with a single precision hardware FPU, such as the Texas Instruments Delfino series and ATMEL UC3 series. These enable high performance computations by allowing instructions such as a floating point multiply in a single cycle. While the advantage of a hardware FPU is attractive, being limited to single precision can cause numerical issues, as noted in [15]. Other authors have opted for full double precision floating point, such as in [16] using a PowerPC, however we have found that a properly scaled system, with reasonable constraints (i.e. do not use 10^6 for unwanted constraints, but remove the constraint instead) can result in good control

without any significant numerical problems. By using the auto-coder, both single and double precision implementations can be automatically generated, allowing the user to exploit the hardware available on their chosen target.

IV. Embedded MPC

For this research we are implementing our embedded MPC algorithm on two hardware platforms; a Texas Instruments (TI) 32bit microcontroller and an ARM microprocessor. Both targets contain a single precision hardware FPU, however MPC on the TI microcontroller is run as a procedural application, while the ARM runs a Linux kernel and the algorithm is implemented as an application of the operating system.

A. Texas Instruments Delfino C28343

The TI C28343 is a 32bit, 200MHz microcontroller with a 32bit FPU and 260KB of on-chip RAM. The IC is targeted at real time control applications in the automotive and aerospace industries which require a small footprint, low power and high performance characteristics. The Integrated Circuit (IC) itself is available from US$12, while the control card pictured in Figure 4 only costs US$109 and contains an onboard power supply, external ADCs, and EEPROM, as well as breaking out all General Purpose Input / Output (GPIO) pins via a DIM100 connector. In summary, the control card measures 90mm x 30mm x 3mm, runs at 5V and draws typically less than 300mA. We consider this an 'off the shelf' product, and, due to its small form factor, believe it can be implemented as is in a range of embedded hardware applications.

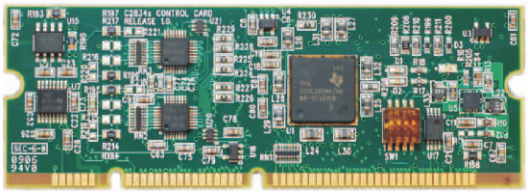

Fig. 4. Texas Instruments Delfino C28343 Control Card.

The auto-generated code is loaded into TI's Code Composer Studio (CCS), a mature Integrated Development Environment (IDE) based on Eclipse, together with an optimizing C/C++ compiler. Using a Blackhawk JTAG emulator, the IC can be programmed in just over a second, even with more than 80KB of program memory, because the program code is executed from RAM using this platform. Using this target, together with the software framework we developed, an embedded MPC controller can be generated, compiled and programmed to the target in less than 10 seconds. This is a substantial speed increase over work flows requiring compilation of a hardware based language such as Verilog, VHDL, or Handel-C, which are consistent with compilations times in the tens to hundreds of minutes for MPC problems, a factor of 100 or more.

B. PandaBoard ARM Cortex A9

The PandaBoard is a low cost single board computer based on a dual core ARM Cortex A9 processor running at 1 GHz. As well as a single precision hardware FPU, the board contains 1GB of off-chip RAM and numerous peripherals such as USB, ethernet and HDMI. The board is available for US$174 allowing for a fully featured embedded computer in a small, low power platform.

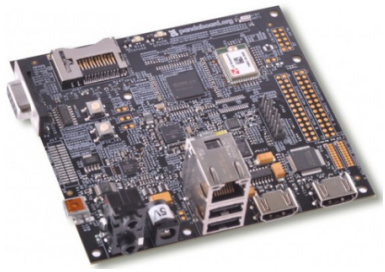

Fig. 5. PandaBoard with Dual Core ARM Cortex A9.

In contrast to the TI target, auto-generated code is compiled as a standalone executable and run as an application of the Operating System (OS), which in this case is Ubuntu 11.10. As this a processor in the loop implementation, as described in the next section, we are not concerned with regular timing of the OS. However for future work, we are investigating use of the real-time Linux kernel allowing for the deterministic sampling required of a control loop.

C. Processor In The Loop Implementation

In this work we will simulate the plant to be controlled using a Processor In the Loop (PIL) implementation. As shown in Figure 6, PIL uses the development computer to simulate the plant to be controlled, with the MPC controller implemented in hardware on the target device. Communication between the two pieces of hardware is done via a simple USB-serial link.

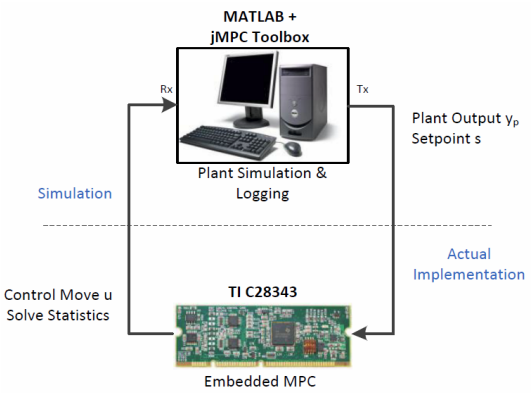

Fig. 6. Processor In the Loop MPC Implementation.

Using this approach we have not implemented regular sampling on the target, as the communications delay is much more significant than the computation time. Instead the controller waits until a sample is received via the serial interface, performs the control calculation, then immediately sends the control moves back via the serial interface. It is important to note this communications delay is only present with the PIL

implementation. In an industrial implementation the sample rates of typical Analog to Digital Convertors (ADCs) and Digital to Analog Convertors (DACs) are of orders of magnitude faster than the controller.

In order to assess how fast the controller is calculating control moves, a 1μs timer is used to measure the computation period required for each iteration of the implementation. Together with the control moves and model and QP information, the timing information is transmitted back for subsequent analysis. To initiate a PIL implementation using the jMPC Toolbox is as easy as running a MATLAB simulation. Instead of using the default MATLAB simulation target, PIL can be selected for the implementation and the same controller run on the target hardware:

```
%PIL Simulate & Plot Result
simresult = sim(MPC1,simopts,'PIL');
plot(MPC1,simresult)
```

Assuming the target has been programmed and is ready to go, the above commands will automatically begin the MPC implementation, transmitting and receiving data via a nominated serial port. To further ease this process, all required PIL code has been auto-generated by `embed`, meaning apart from setting a few include paths and compiler settings, the auto-generated code can be literally dropped in to the desired IDE, compiled, programmed, and a PIL embedded MPC implementation run.

V. Case Studies

To validate our algorithms and framework we present two case studies. The first is an example from one of the fastest reported embedded MPC applications, while the second is a rigorous nonlinear implementation.

A. Rotating Antenna

The following linearized discrete rotating antenna model is presented in [14] for their SoC implementation of real-time model predictive control.

$$\dot{x} = \begin{bmatrix} 1 & 0.1 \\ 0 & 0.9 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0.0787 \end{bmatrix} u \tag{3}$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} x \tag{4}$$

As in [14] the horizons are set as $N_p = 20$ and N_c varied from 3 to 10, with the sample time of 0.1. While not stated, the reference figure suggests that the control voltage is limited to $\pm$ 2V, at a maximum rate of change of $\pm$ 1V. Implemented using the jMPC Toolbox and our auto-code framework the PIL implementation results for $N_c = 3$ on the TI C28343 are shown in Figure 7.

Figure 7 shows the plant output (angle of the antenna), plant input (motor voltage) as well as the actual computation time and number of QP iterations required. Most noticeable is that solving the QP is only required when the constraints are active. Using our algorithm solving the QP can be skipped if it is determined that no constraints are active, thus saving a significant amount of power. To compare the speed of our

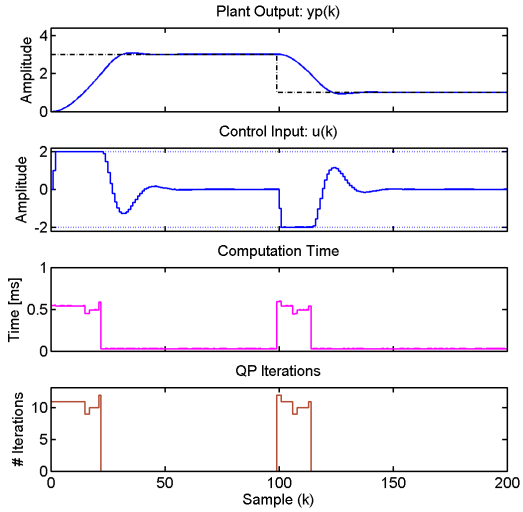

Fig. 7. Rotating antenna PIL implementation results.

implementation with those published, consult Table I. Note the timing information given for the TI and ARM column is based on the *worst case* sample rate (i.e. the the maximum execution time to calculate the MPC control moves), as measured on the device. The SoC column assumes a worst case of 15 optimization iterations, as the timing information in [14] is *per optimization iteration* only.

TABLE I
EMBEDDED MPC RESULTS FOR A ROTATING ANTENNA MODEL

N_c	RAM [KB]	Flash [KB]	TI [ms]	ARM [ms]	SoC [14] [ms]
3	1.18	1.52	0.60	0.15	6.75
4	1.58	1.80	0.93	0.27	8.66
5	2.02	2.14	1.36	0.34	10.98
6	2.49	2.52	1.91	0.49	13.65
7	3.01	2.94	2.58	0.76	16.76
8	3.56	3.42	3.65	1.01	20.78
9	4.16	3.94	4.72	1.25	24.83
10	4.80	4.52	5.51	1.71	29.29

Based on the results presented so far, it is evident our MPC algorithm is not only much faster (with sampling rates in the kHz range achievable), but also requires a very small memory footprint. There are however two problems with the simulation results presented so far, firstly the horizons are artificially large for the problem, and secondly the physical system itself has no need for such fast control, given the time basis is in minutes. In fact using tuning values of $N_p = 8$ and blocking N_c as $[4, 4]$, this results in near identical control, and at over a 3kHz sampling rate. Therefore for the next case study we will examine a system which will push the required sampling rate of our algorithm.

B. Inverted Pendulum on a Moving Cart

The inverted pendulum on a linear cart is a simple yet challenging control problem requiring high speed control to keep the pendulum balanced. For this case study we are considering

the fourth order single inverted pendulum manufactured by Quanser [17], described by the following nonlinear equations:

$$\ddot{y} = \frac{\frac{F}{m} + l\theta^2 \sin\theta - g\sin(\theta\cos\theta)}{\frac{M}{m} + \sin^2\theta} \quad (5)$$

$$\ddot{\theta} = \frac{\frac{-F}{m}\cos\theta + \frac{M+m}{mg\sin\theta} - l\theta^2 \sin\theta\cos\theta}{l\left(\frac{M}{m} + \sin^2\theta\right)} \quad (6)$$

where the system input is F, the force applied to the cart in newtons, and the system outputs are y, cart position in metres and θ, angle of the pendulum from vertical, in radians. M is the mass of the cart and m is the mass of the pendulum, specified as 0.455kg and 0.21kg, respectively. l is the distance to the centre of mass of the pendulum, specified as 0.305m, while g is the gravitational constant, 9.81m/s^2.

To design a jMPC linear controller the system must first be linearized about a suitable operating point. Using the jMPC `linearize` method, the system can be automatically linearized about an input point, operating point, or both. For this example we have linearized about the input point F = 0N:

```
%Create a jNL Nonlinear Plant
Plant = jNL(@nl_pend,C,param);
%Linearize the Plant to create the controller model
Model = linearize(Plant,0);
```

With a linear model, an MPC controller can be constructed and tested against the 'real' nonlinear plant. This allows a more realistic validation of both the controller, and controller tuning, increasing the chance of better control performance once implemented in a physical system. Given the fast dynamics of the system, together with a limit on the problem size that can be solved within a fixed time, a sampling rate of 20Hz was chosen and the model discretized at this rate. At this sample rate, the horizons were set as 25 and 5 for the prediction and control horizons, together with tuning weights of $u_{wt} = 2$ and $y_{wt} = [1.5, 0]$. Note that the second output weight is set as 0, indicating that we don't want to control the pendulum angle, however we do want to constrain it. This simple modification halves the size of the output prediction matrix, allowing faster computation times, as well reducing the need for a redundant setpoint. Finally the input is constrained to $\pm$5N, the cart is constrained to $\pm$2m from the starting point, and the pendulum constrained to $\pm 45^\circ$ from vertical.

Using the PIL framework described earlier, the linear inverted pendulum MPC controller can be generated and programmed to the target, whilst the full, nonlinear, differential equations can be simulated on the development computer. To further add realism to the implementation, artificial measurement noise is added to the sensor measurements to stress test the system. The implementation results are shown in Figure 8.

An interesting observation is that even with a small amount of noise, the MPC controller must now solve a QP at most of the simulation steps, due to perturbations pushing the open-loop unstable nature of the system. It can also be seen that the QP solver often reaches the maximum number of iterations (30), however the intermediate solution found keeps the system stable for this implementation. A comparison of memory requirements and maximum sampling rates versus control horizon are presented in Table II.

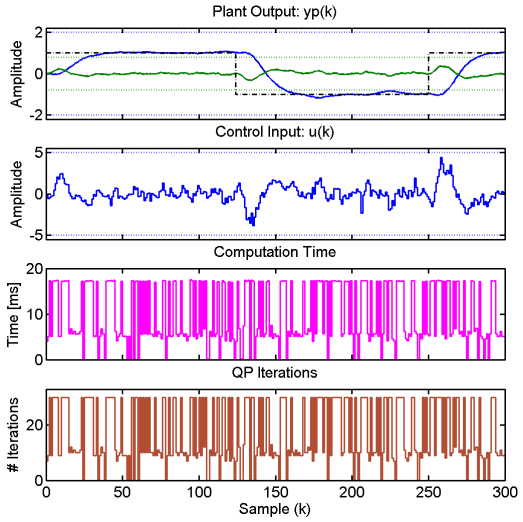

Fig. 8. Inverted Pendulum PIL implementation results.

TABLE II
EMBEDDED MPC RESULTS FOR A NONLINEAR INVERTED PENDULUM

N_c	RAM [KB]	Flash [KB]	TI [ms]	ARM [ms]
3	8.18	5.25	11.38	2.50
4	8.98	5.95	14.19	3.42
5	9.81	6.70	17.50	4.73
6	10.69	7.50	21.44	5.92
7	11.60	8.35	26.35	7.41
8	12.56	9.25	31.89	9.22
9	13.56	10.19	37.83	11.38
10	14.59	11.18	44.67	13.58

Given that the current implementation could run at up to 57Hz on the TI target it could be suggested that the sampling rate of the system be increased. It is easy to assume that a faster sample rate can improve the controller performance, given that it can react to disturbances faster. Up to a certain point this is valid, however a major problem with this approach concerns the prediction horizon of MPC. If we double the sample rate, the prediction horizon is now only half as long, which can substantially reduce the control performance. If we also double the prediction horizon, the QP constraint matrix A now roughly doubles in length, adding a significant amount more memory, as well as reducing the computational efficiency of the solver. We have also observed numerical errors in single precision with large prediction horizons, presumably due to the large number of times the model state matrix is squared to form the constraint Hankel matrix.

VI. CONCLUSION

This paper has presented a MATLAB framework for generating fast model predictive controllers suitable for implementation on a range of embedded targets. Moreover we demonstrated a processor in the loop implementation on two targets,

a Texas Instruments Delfino microcontroller and an ARM Cortex 9 microprocessor. Using the TI target we achieved sampling rates up to 3kHz and on the ARM up to 11kHz. The implementation was also shown to be highly memory efficient, with less than 20KB of RAM required for control of a nonlinear model using reasonable horizons of 25 and 5 for the prediction and control horizons. The framework has also demonstrated how an embedded model predictive controller can be automatically generated, compiled, and implemented in as little as 10 seconds on cost effective off-the-shelf hardware.

ACKNOWLEDGMENT

Financial support to this project from the Industrial Information and Control Centre, Faculty of Engineering, AUT University, New Zealand is gratefully acknowledged.

REFERENCES

[1] D. I. Wilson and B. R. Young, "The Seduction of Model Predictive Control," *Electrical & Automation Technology*, pp. 27–28, Dec/Jan 2006, ISSN: 1177-2123.

[2] M. Pachter and P. R. Cjhandler, "Challenges of autonomous control," *IEEE Control Systems Magazine*, vol. 18, no. 4, pp. 92–97, 1998.

[3] A. Hennig and G. J. Balas, "MPC supervisory flight controller: A case study to flight EL AL 1862," in *AIAA Guidance, Navigation and Control Conference*, 2008.

[4] V. Manikonda, P. O. Arambel, M. Gopinathan, R. K. Mehra, and F. Y. Hadaegh, "A model predictive control-based approach for spacecraft formation keeping and attitude control," in *American Control Conference*, 1999, p. 4258.

[5] L. G. Bleris, J. Garcia, M. V. Kothare, and M. Arnold, "Towards Embedded Model Predictive Control for System-on-a-Chip Applications," *Journal of Process Control*, vol. 16, no. 3, pp. 255–264, 2006.

[6] A. Richards, W. Stewart, and A. Wilkinson, "Auto-coding Implementation of Model Predictive Control with Application to Flight Control," in *European Control Conference*, 2009, pp. 150–155.

[7] K. V. Ling, B. F. Wu, and J. M. Maciejowski, "Embedded Model Predictive Control (MPC) using a FPGA," in *The 17th World Congress of the International Federation of Automatic Control (IFAC)*, 2008, pp. 15 250–15 255.

[8] T. A. Johansen, W. Jackson, R. Schreiber, and P. Tondel, "Hardware Synthesis of Explicit Model Predictive Controllers," *IEEE Transactions on Control Systems Technology*, vol. 15, no. 1, pp. 191–197, 2007.

[9] L. Wang, *Model Predictive Control System Design and Implementation using MATLAB*. Springer, 2009.

[10] S. J. Wright, "Applying New Optimization Algorithms to Model Predictive Control," in *Chemical Process Control-V, CACHE, AIChE Symposium*, vol. 93, 1997, pp. 147–155.

[11] J. Currie and D. I. Wilson, "A Model Predictive Control toolbox intended for rapid prototyping," in *16th Electronics New Zealand Conference (ENZCon 2009)*, T. Molteno, Ed., Dunedin, New Zealand, 18–20 November 2009, pp. 7–12.

[12] J. Mattingley and S. Boyd, "CVXGEN: A Code Generator for Embedded Convex Optimization," *Optimization and Engineering*, vol. 13, no. 1, pp. 1–27, 2012.

[13] J. Currie and D. I. Wilson, "Lightweight Model Predictive Control Intended for Embedded Applications," in *9th International Symposium on Dynamics and Control of Process Systems (DYCOPS)*, Leuven, Belgium, 5–7 July 2010, pp. 264–269.

[14] P. Vouzis, L. Bleris, M. Arnold, and M. Kothare, "A System-on-a-Chip Implementation for Embedded Real-Time Model Predictive Control," *IEEE Transactions on Control Systems Technology*, vol. 17, no. 5, pp. 1006–1017, 2009.

[15] M. S. K. Lau, S. P. Yue, K. V. Ling, and J. M. Maciejowski, "A Comparison of Interior Point and Active Set Methods for FPGA Implementation of Model Predictive Control," in *European Control Conference*, 2009, pp. 156–161.

[16] L. G. Bleris and M. V. Kothare, "Real-Time Implementation of Model Predictive Control," in *American Control Conference*, 2005, pp. 4166–4171.

[17] Quanser, "Linear Control Challenge: Inverted Pendulum," http://quanser.com/, 2012.