Diploma Thesis
Real-Time Model Predictive Control

Thomas Haugan

Automatic Control Laboratory
Swiss Federal Institute of Technology
ETH Zentrum - ETL I26, CH 8092 Zürich, Switzerland
tel. +41-1-632 6679, fax +41-1-632 1211

June 13, 2001

Student: Thomas Haugan
Supervisors: A. Bemporad, F. Borrelli, D. Mignone
Professors: M. Morari, S. Skogestad

Abstract

The primary objective of this project is to extend the Model Predictive Control (MPC) toolbox for real-time use. The basic MPC algorithm is coded in "C" and interfaced with Simulink by use of a S-function structure. Such extension of the toolbox enables the real-time implementation of MPC by using xPC Target, which is a host-target prototyping environment shipped with MATLAB R12. Additionally the xPC setup is compared against the Real-Time Toolbox. The Real-Time Toolbox is a package created by a third-party vendor of MATLAB for connecting MATLAB and Simulink to the real world.

The MPC is tested on the "Ball & Plate", available in the lab at Institut für Automatik (IFA) at ETH. The "Ball & Plate" is consisting of a ball rolling over a gimbal-suspended plate which is actuated by two independent motors. The results of this study show that the Real-Time Toolbox introduces a delay in the data acquisition card and has an excessive demand of CPU power. This causes a system overload and eventually a crash. The xPC Target constellation produces less noise, and has a smaller delay than the Real-Time Toolbox. As a result from this, the xPC Target is suggested used as standard hardware setup for real-time implementation of MPC. Simulation and testing show a successful extension of the MPC toolbox for real-time use. Further optimization of the code and QP-solver is recommended.

The result of this study will be used as part of the work in extending the MPC toolbox, currently licensed to over 1000 users, and to decide upon the best strategy for implementation of MPC for real-time use.

Contents

1 Introduction **6**

2 Theory **7**
2.1 Model Predictive Control (MPC) 7
2.1.1 Kalman filter 10

3 Experimental setup **13**
3.1 “Ball & Plate” system 13
3.1.1 Model of “Ball & Plate” 14
3.2 Control task 18
3.2.1 Real-Time Toolbox 18
3.2.2 xPC Target 19
3.3 S-function in Simulink 19
3.4 Programming in C 21

4 Real-time implementation of MPC **24**
4.1 Extension of the MPC toolbox for real-time use 24
4.1.1 Approach 1: MEX-file 24
4.1.2 Approach 2: S-function 25
4.2 Implementation of MPC using Real-Time Toolbox 25
4.3 Implementation of MPC by target-host constellation, xPC Target 25

5 Results from MPC implemented on “Ball & Plate” **26**
5.1 Simulation of M-file S-function and S-function in “C” 26
5.2 MPC implemented with Real-Time toolbox 28
5.2.1 Step response of “Ball & Plate” with Real-Time Toolbox 28
5.2.2 Anticipative versus non-Anticipative action 28
5.2.3 Calculation time in output stage 35
5.2.4 Call to C-function from original MPC algorithm 35
5.2.5 S-function written in C 38
5.3 MPC implemented with xPC Target 40
5.3.1 Step response of “Ball & Plate” with xPC Target 40
5.3.2 Tracking of reference 40
5.4 Calibration 46

5.5 Noise 48

6 Discussion **52**

6.1 Theoretical assumptions 52

6.2 Extension of MPC toolbox 52

6.3 MPC implemented with Real-Time Toolbox 53

6.3.1 Calculation time 54

6.3.2 Anticipative versus non-Anticipative action 54

6.3.3 Algorithms written in "C" 55

6.4 MPC implemented with xPC Target 55

6.5 Noise 57

7 Conclusion **58**

8 Recommendations **60**

Acknowledgements **61**

List of symbols **62**

Bibliography **63**

A MEX-function **65**

B C-function (MPC2) **72**

C DANTZIG-routine **80**

D S-function written in C **88**

E Photos of "Ball & Plate" system **114**

List of Figures

2.1 Receding horizon control . . . 8
2.2 Predictive control structure . . . 8

3.1 Ball & Plate system. . . . 14
3.2 TCP/IP network connection . . . 14
3.3 Block diagram of Motor, Plate and transmission . . . 15

5.1 Step response with original algorithm, x-direction. . . . 27
5.2 Step response with original algorithm, y-direction. . . . 27
5.3 Step response with S-function written in C, x-direction. . . . 27
5.4 Step response with S-function written in C, y-direction. . . . 28
5.5 Step response with S-function written in C, x-direction. . . . 28
5.6 Controller output compared to reference, anticipative action. . . . 30
5.7 Controller output compared to reference, non-anticipative action. . . . 30
5.8 Tracking of a square reference, anticipative action. . . . 30
5.9 Tracking of a square reference, non-anticipative action. . . . 31
5.10 Tracking of a circle reference, non-anticipative action. Angular frequency $\omega = \frac{\pi}{3}$ rad/s . . . 31
5.11 Tracking of a circle reference, anticipative action. Angular frequency $\omega = \frac{\pi}{3}$ rad/s . . . 31
5.12 Tracking of a circle reference, non-anticipative action. Angular frequency $\omega = \frac{\pi}{2}$ rad/s . . . 32
5.13 Tracking of a circle reference, anticipative action. Angular frequency $\omega = \frac{\pi}{2}$ rad/s . . . 32
5.14 Tracking of a circle reference, non-anticipative action. Angular frequency $\omega = \pi$ rad/s . . . 32
5.15 Tracking of a circle reference, anticipative action. Angular frequency $\omega = \pi$ rad/s . . . 33
5.16 Tracking of a circle reference, non-anticipative action. Angular frequency $\omega = \frac{5}{4}\pi$ rad/s . . . 33
5.17 Tracking of a circle reference, anticipative action. Angular frequency $\omega = \frac{5}{4}\pi$ rad/s . . . 33
5.18 Tracking of a circle reference, non-anticipative action. Angular frequency $\omega = \frac{6}{4}\pi$ rad/s . . . 34

5.19 Tracking of a circle reference, anticipative action. Angular frequency $\omega = \frac{6}{4}\pi$ rad/s . . . 34
5.20 Tracking of circle trajectory with M-file S-function calling C-algorithm . . . 36
5.21 Controller output in X- and Y-position when tracking a circle with angular velocity of $\frac{\pi}{2}$ rad/s. M-file S-function calling C-algorithm . . . 37
5.22 Tracking of circle trajectory with S-function written in C . . . 38
5.23 Controller output in X- and Y-position when tracking a circle with angular velocity of $\frac{\pi}{2}$ rad/s. S-function written in "C" . . . 39
5.24 Step response with xPC Target, x-direction. . . . 40
5.25 Tracking of square-reference with weighting = {0,1,[10 0]} . . . 41
5.26 Controller output with weighting = {0,1,[10 0]} . . . 41
5.27 Tracking of square-reference with weighting = {0,1,[1 0]} . . . 42
5.28 Controller output with weighting = {0,1,[1 0]} . . . 42
5.29 Tracking of square-reference with weighting = {0,10,[1 0]} . . . 42
5.30 Controller output with weighting = {0,10,[1 0]} . . . 43
5.31 Tracking of circle-reference with angular velocity = 1 rad/s . . . 44
5.32 Controller output with angular velocity = 1 rad/s . . . 44
5.33 Tracking of circle-reference with angular velocity = $\frac{\pi}{2}$ rad/s . . . 44
5.34 Controller output with angular velocity = $\frac{\pi}{2}$ rad/s . . . 45
5.35 Tracking of circle-reference with angular velocity = π rad/s . . . 45
5.36 Controller output with angular velocity = π rad/s . . . 45
5.37 Linearized plot of true angles and measured values, alpha-laser . . . 46
5.38 Error between the measured voltage and the linearized model, alpha-laser . 47
5.39 Error between the measured voltage and the linearized model, beta-laser . . 47
5.40 Error between the measured voltage and the linearized model, delta-laser . 47
5.41 Error between the measured voltage and the linearized model, gamma-laser 48
5.42 Ball positions during noise measurements. . . . 48

E.1 Photo of "Ball & Plate" system. . . . 114
E.2 Photo of "Ball & Plate" system. . . . 114
E.3 Photo of "Ball & Plate" system. . . . 115
E.4 Photo of ball tracking a circle reference. . . . 115

List of Tables

3.1 Physical constraints on Ball & Plate system 13
3.2 Physical reasons for model of Ball & Plate 15
3.3 Simulation stages 20
3.4 Functions called during simulation 21

5.1 Quality criteria for comparison of algorithms. 26
5.2 Parameters during anticipation experiments 29
5.3 Elapsed computation time in output stage of original algorithm 35
5.4 Elapsed computation time in output stage of M-file S-function calling C-function 35
5.5 Parameter-settings for MPC-controller during experiments 36
5.6 Statistics on ball position, tracking a circle with M-file S-function calling C-function 37
5.7 Parameter-settings for MPC-controller during experiments 38
5.8 Statistics on ball position, tracking a circle with C S-function 39
5.9 Parameters for tracking of a square with C S-function 40
5.10 Parameters for tracking of a circle with C S-function 43
5.11 Statistics on task execution time during tracking of circle 43
5.12 Parameters for Ball & Plate 46
5.13 Noise in plate angle. 49
5.14 Noise in ball positioning with Real-Time Toolbox. 49
5.15 Noise in ball positioning with xPC Target. 50
5.16 Covariance matrices for “Ball & Plate”. 50

Chapter 1

Introduction

This work was carried out at the Institut für Automatik (IFA), ETH Zürich. The primary purpose was to implement Model Predictive Control for real-time use. The Institut für Automatik is doing research in real-time implementation of Model Predictive Control. They developed the official MATLAB toolbox for Model Predictive Control (MPC), and contributed to spark a world-wide interest for this control strategy. MPC is an optimization-based strategy that uses a system model to predict the effect of a control action applied on the system, [14]. The advantage of MPC is the ability to handle multivariable constrained control problems in an optimal way. However, a practical disadvantage is the computational cost, which tends to limit MPC applications to linear processes with relatively slow dynamics. The available computational power increases at a still rising rate, which gives expectation of an implementation of the MPC algorithm on fast processes. In the lab at IFA, there exists a system called "Ball & Plate", consisting of a ball rolling over a gimbal-suspended plate which is actuated by two independent motors. It requires implementation of control action within few milliseconds, and is therefore suitable for testing the MPC algorithm with a short sampling time.

The main task in this work consisted of extending the MPC toolbox for real-time use. The basic algorithm needed to be coded in "C" and interfaced with Simulink. This consisted of coding the output sequence in "C", having M-file initialization files to do the necessary matrix buildup. The problem-areas in implementing MPC on a fast system is discussed. The computational cost of the algorithms in "C" and M-file were compared. Two available tools involving different platform configurations for implementation of MPC were tested and compared.

The thesis contains some theory on the MPC strategy, highlighting the basic concepts, and some general theory of Simulink. Also the idea behind the model of the system is shown. The results of the study and the discussion of these is thereafter presented.

Chapter 2

Theory

2.1 Model Predictive Control (MPC)

On-line optimization is a commonly used tool in the chemical process industry for operating plants at their maximum performance. Typically, this issue is addressed via a Model Predictive Control (MPC) framework where at regular time intervals the measurements from the plant are obtained and an optimization problem is solved to predict the optimal control actions, [2]. In [14], Morari and Lee states that even though the ideas of Model Predictive Control can be traced back to the 1960s, a real interest in this field started to surge only in the 1980s after publication of the first paper on Dynamic Matrix Control (DMC) by Cutler and Ramaker in 1979. The objective behind development of DMC was to tackle multivariable constrained control problems typical for oil and chemical industries. More than fifteen years after DMC appeared in industry, a theoretical basis for this technique has started to emerge.

The idea of redefining the control objective on-line, which is commonly referred to as Receding Horizon Control (RHC), forms the basis for the MPC strategy. RHC can be described by referring to figure 2.1. At present time $t = k$, the behavior of the process is considered over a horizon p, [13]. By using a model of the process, the response to changes in the manipulated variable is predicted. Current and future moves of the manipulated variables are selected such that the predicted response has certain desirable characteristics. For instance, a commonly used objective is to minimize the sum of squares of the future errors. An error is here defined as the deviation of the controlled variable from a desired setpoint. This minimization can also take into account constraints which may be present on the manipulated variables and the outputs. The idea is appealing, but would not work very well in practice if the moves of the manipulated variable determined at time k were applied blindly over the future horizon. The reason for this is that disturbances and modelling errors may lead to a deviation between the predicted behavior and the actual observed behavior. In this case, the computed manipulated variable moves are not appropriate anymore. Therefore, only the first computed move is actually implemented. At next time step, $t = k + 1$, a measurement $y(k + 1)$ is taken, the horizon is shifted

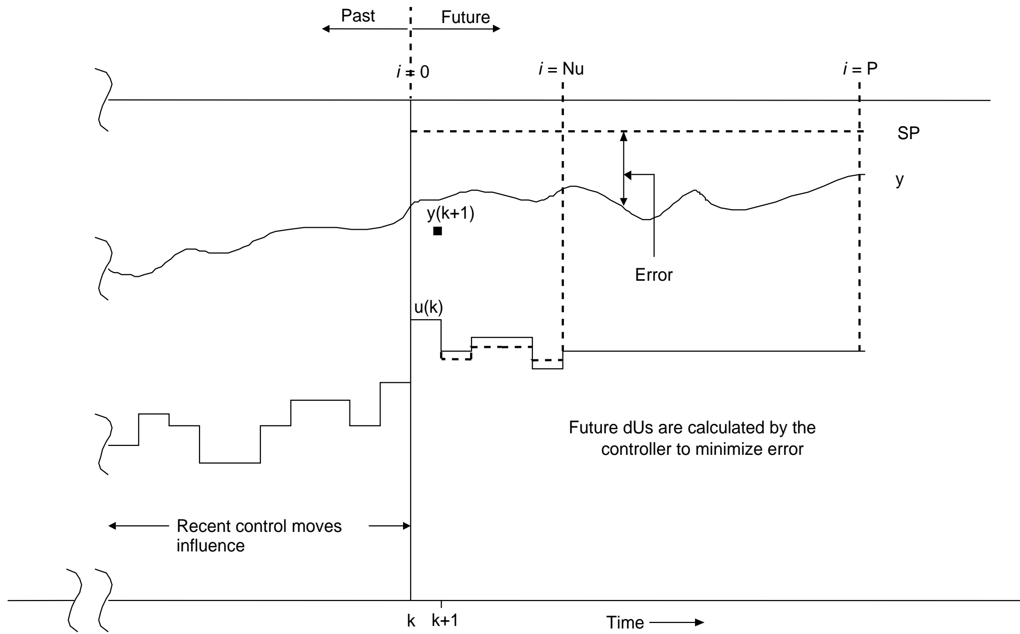

Figure 2.1: Receding horizon control

forward by one step, and the optimization is done again over this shifted horizon based on current system information.

Model-based predictive control algorithms are versatile and robust in applications, outperforming minimum variance and PID in challenging control situations. In [12], Marlin has given a general predictive control structure as shown in figure 2.2. Three transfer functions represent the true process with the final element and sensor, $G_p(s)$; the controller, $G_{cp}(s)$; and a dynamic model of the process, $G_m(s)$. The feedback signal E_m is the difference between the measured and predicted controlled variable values. The variable E_m is, however, equal to the effect of the disturbance, $G_d(s) * D(s)$, if the model is perfect. However, the model is never exact, and the feedback signal includes therefore the effect of

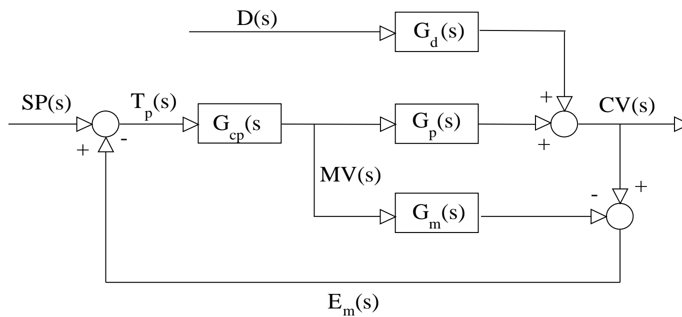

Figure 2.2: Predictive control structure

the disturbance together with model inaccuracy.

The MPC-block in the new MPC Simulink library is based on the linear state space model shown in (2.1) and (2.2).

$$x(k+1) = Ax(k) + B_u u(k) + B_v v(k) + B_d d(k) \tag{2.1}$$

$$y(k) = Cx(k) + D_v v(k) + D_d d(k) \tag{2.2}$$

Here $x(k) \in \mathbb{R}^n$ represents the state of the system, $u(k) \in \mathbb{R}^{n_u}$ are manipulated variables, $v(k) \in \mathbb{R}^{n_v}$ is a vector of measured disturbances, $d(k) \in \mathbb{R}^{n_d}$ are unmeasured disturbances and $y(k) \in \mathbb{R}^{n_y}$ is the output vector. The $A, B_{u,v,d}, C$ and $D_{v,d}$ matrices are state matrices describing the system, and the notation u, v and d is the distinction between manipulated variables, measured disturbance and unmeasured disturbance, respectively.

The MPC-block has a similar approach to the one referred to as Receding Horizon Control, outlined above. At time t, a measurement of the output y, along with reference r and disturbance v, is taken into the algorithm . The last control move and the last state is saved and used in a measurement update (or estimate) of the current state $x(t)$, shown in 2.3. As the true states $x(k), x_d(k)$ are not available to the controller, predictions are obtained from state estimates.

$$\begin{aligned} \hat{y} &= C_m * \hat{x}^- + D_{vm} \\ \hat{x} &= \hat{x}^- + L * (y - \hat{y}) \end{aligned} \tag{2.3}$$

where L denotes the linear filter gain, see section 2.1.1. y is the measurement at time t and $\hat{y}$ is the output estimated. The state estimate is here denoted as $\hat{x}$. C_m is the state matrix C, containing only the rows corresponding to the measured states. D_{vm} is containing the columns of measured disturbances of the state matrix D, and the rows of measured outputs. C_m, D_{vm} and L are created in the initialization section executed before the output sequence is computed. Thereafter the input sequence

$$\Delta U^* \triangleq \{\delta u^*(t), \delta u^*(t+1), \dots, \delta u^*(t+N_u)\} \tag{2.4}$$

is calculated up to N_u number of steps, minimizing the cost function outlined in equation (2.5). This takes into account the weighted difference between input u and u_{target} and introduces a slack variable ϵ. To explain the idea behind the introduction of a slack variable, there is a need for distinguish between *hard* and *soft* constraints. The constraints stipulated in (2.5) may render the optimization problem infeasible. Input saturation constraints cannot be exceeded, while constraints involving outputs can be violated, although with undesirable consequences for the controlled system. By treating the constraints involving state components as soft constraints, feasibility of the problem in (2.5) is ensured. As the inputs are generated by the optimization procedure, the input constraints can

always be considered as hard. This procedure prevents the control law from being infeasible. The cost function is minimized in a Quadratic Programming (QP) solver, the DANTZIG-routine. The DANTZIG-routine written by N. L. Ricker is used to find the optimal solution to the control problem, where the function to be optimized is shown in equation (2.5).

$$\min_{u(k|k),\dots,\Delta u(m-1+k|k)} \sum_{i=0}^{p-1} \left(\sum_{j=1}^{n_u} |w_{i,j}^{u}[u_j(k+i|k) - u_{target,j}(k)]|^2 + \sum_{j=1}^{n_u} |w_{i,j}^{\Delta u} \Delta u_j(k+i|k)|^2 + \sum_{j=1}^{n_y} |w_{i+1,j}^{y}[y_j(k+i+1|k) - r_j(k+i+1)]|^2 \right) + \rho_\epsilon \epsilon^2 \tag{2.5}$$

Subject to:

$$\begin{aligned} u_i^{min} \leq u(k+i|k) \leq u_i^{max} \\ \Delta u_i^{min} \leq \Delta u(k+i|k) \leq \Delta u_i^{max}, \qquad i = 0, \dots, p-1 \\ -\epsilon + y_i^{min} \leq y(k+i+1|k) \leq y_i^{max} + \epsilon \\ \Delta u(k+j|k) = 0, \qquad j = m, \dots, p \\ \epsilon \geq 0 \end{aligned}$$

In (2.5), $*(k+i|k)$ denotes the value predicted for time $(k+i)$ based on the information available at time k. $w_{i,j}^{u}$ is the input weight, $w_{i,j}^{\Delta u}$ is the input increment weight and $w_{i+1,j}^{y}$ is the output weight. $r(k)$ is the current sample of the output reference. When the future evolution of $r(k)$ is unknown, the current reference is extended over the whole prediction horizon p. If the future evolution of the reference is known, the reference is read from a file, where the time and reference at that time is stored. After calculation of the train of input increments, a time update of the state is made. This is straightforward by using (2.6).

$$\hat{x}_{k+1}^{-} = A\hat{x}_{k}^{-} + Bu_k \tag{2.6}$$

Then the first computed control move in the train is applied on the system. At next time-step, the procedure is repeated.

2.1.1 Kalman filter

In practice, it is usually not possible to measure all the disturbances and state variables, [13]. This is why estimation techniques are used to estimate the state from the measured input and output sequences. The most popular state estimation technique is the Kalman filter, first introduced by R. Kalman [8] in 1960 for discrete systems. A year after, the theory was applied on continous systems by Kalman & Bucy [9]. The technique of Kalman filters is a very general filtering technique which can be applied to the solution of such

problems as estimation, prediction, noise filtering and optimal control, [16]. Kalman filters can be applied to both stationary and non-stationary processes and can include initial conditions of processes in estimation, prediction, filtering or stochastic optimal control algorithms.

In [10], Lewis outlines the procedure of Kalman filtering. First step is to determine how the mean and the covariance of the state x_k propagate under the dynamics (2.7).

$$x(k+1) = Ax(k) + Bu(k) + Gw_k \tag{2.7}$$

The overbar will in the following be used to denote expected value. Since u_k is deterministic (it is applied to the system), and the noise w is assumed white, the expected mean of the state can be written as

$$\bar{x}_{k+1} = A\bar{x}_k + Bu_k \tag{2.8}$$

The state covariance is defined as

$$P_{x_k} \equiv E\{(x_k - \bar{x}_k)(x_k - \bar{x}_k)^T\} \tag{2.9}$$

Substituting (2.8) into (2.9), assuming white noise such that $E\{w_j w_k^T\} = 0$, the state covariance propagates to

$$P_{x_{k+1}} = AP_{x_k}A^T + GQG^T \tag{2.10}$$

where Q denotes the known covariance of the process noise. Let the output mean be given by

$$\bar{y}_k = C\bar{x}_k \tag{2.11}$$

Then the cross-covariance between state and output is given by(2.12)

$$P_{x_k y_k} = P_{x_k}C^T \tag{2.12}$$

In the same manner, it is straight forward algebra to show that the covariance of the output is given by (2.13)

$$P_{y_k} = CP_{x_k}C^T + R \tag{2.13}$$

where R denotes the covariance for the measurement noise.

At this stage, some new notation need to be introduced. Suppose that no measurements are taken. Let the priori estimation error be denoted as

$$\tilde{x}_k^- = x_k + \hat{x}_k^- \tag{2.14}$$

Moreover, let the estimate at time k *including* the output measurement be denoted as

$$\tilde{x}_k = x_k + \hat{x}_k \tag{2.15}$$

The idea is to find the best linear estimator for the state x that uses all the available information. The discrete Kalman filter will then consist of two steps, the first is the *time update*, by which $\hat{x}_{k-1}$ is updated to $\hat{x}_k^-$. The other is the *measurement update* by which the measurement y_k at time k is incorporated to provide the updated estimate $\hat{x}_k$. The time update is straightforward, using (2.16) and (2.17).

$$\hat{x}_k^- = A\bar{x}_{k-1} + Bu_{k-1} \tag{2.16}$$

$$P_k^- = AP_{k-1}A^T + GQG^T \tag{2.17}$$

Lewis minimizes the mean-square error $J = E\{\tilde{x}_k^T \tilde{x}_k\}$ in order to estimate x_k.

$$\hat{x}_k = \hat{x}_k^- + P_{x_k y_k} P_{y_k}^{-1} * (y_k - \bar{y}_k) \tag{2.18}$$

Substituting $P_{x_k y_k}$,P_{y_k} and $\bar{y}_k$ into (2.18) gives

$$\hat{x}_k = \hat{x}_k^- + P_k^- C^T (CP_k^- C^T + R)^{-1} (y_k - C\hat{x}_k^-) \tag{2.19}$$

which is the second and last step of the Kalman filtering, the *measurement update.*

From this it can be shown that the linear Kalman filter gain, L is

$$L = P_k^- C^T (CP_k^- C^T + R)^{-1} = P_k C^T R^{-1} \tag{2.20}$$

The Kalman filter is a low pass filter which possess both noise rejection and smoothing properties.

Chapter 3

Experimental setup

3.1 “Ball & Plate” system

The “Ball & Plate” system consists of a gimbal-suspended plate actuated by two independent motors. The ball position on the plate is detected by four lasers, reflected by mirrors such that the beams run parallel with the plate. The mirrors run clockwise on a constant speed, causing the laser beams to scan the plate one at a time. The laser beams pass through a calibration point once every scan, resetting an internal counter inside the motor driving the mirrors to zero. A detection is made when the laser beam gets reflected from the surface of the ball. The counter send the elapsed number of teeth to the converter, which converts this signal into a voltage spanning from ±1V. The angle-span measured by using the alpha-, beta-, delta- and gamma-laser between the wall and the ball can then be calculated, see figure 3.1. The system constraints are outlined in table 3.1.

Table 3.1: Physical constraints on Ball & Plate system

System constraints	**Value**
Angle(α, β)	$-17\,^{\circ}, \ldots, +17\,^{\circ}$
Position	$-30\ cm, \ldots, +30\ cm$
Input voltage	$-10\ V, \ldots, +10\ V$

The software I used during the whole project is listed below.

- Windows 2000 operating system
- MATLAB, Version 6.0.0.88, Release 12
- VISUAL C++, 6.0
- Real-Time Toolbox, Version 3.1
- xPC Target. Toolbox for MATLAB Release 12

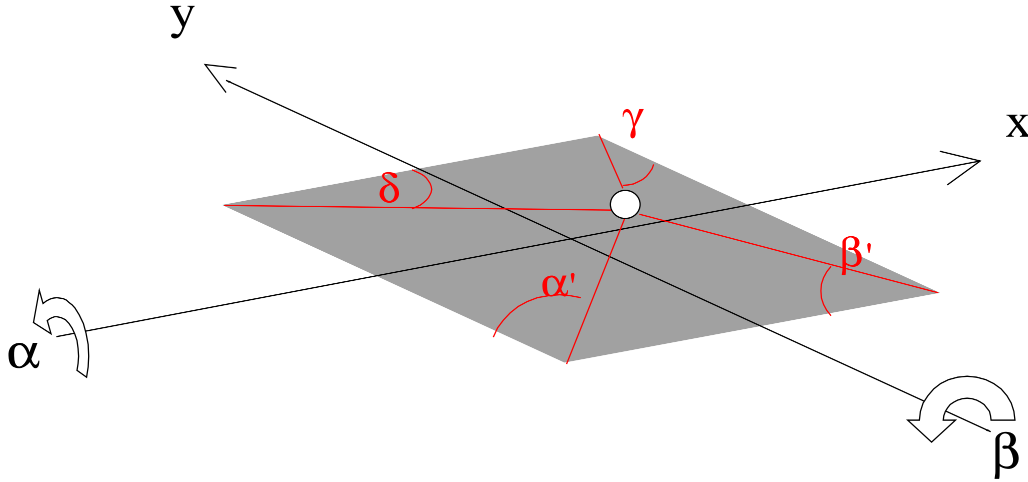

Figure 3.1: Ball & Plate system.

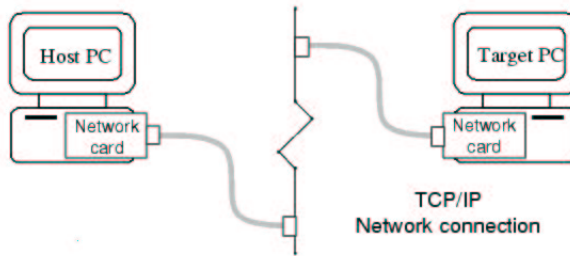

Figure 3.2: TCP/IP network connection

Two different setups for implementing the MPC controller on the system were tested, namely a target-host constellation and by use of Real-Time Toolbox. The host PC connected to the system was a Pentium 933 mHz with a 40Gb harddisk and 500 Mb RAM. The target PC was a Pentium 266 MMX, with 98 Mb RAM. Figure 3.2 shows the setup of the host-target using TCP/IP connection. Photos of the real system is shown in appendix **??**.

3.1.1 Model of "Ball & Plate"

The theory in this section is taken from Hermann [6]. In this model it is assumed that the ball does not slide on the plate, and a kinematic relation is proposed in (3.1).

$$\dot{x_s} - r * sin\psi * \dot{\theta} + r * cos\psi * sin\theta * \dot{\phi} = 0 \qquad (3.1)$$
$$\dot{y_s} + r * sin\psi * \dot{\theta} - r * cos\psi * sin\theta * \dot{\phi} = 0$$

In total, to describe the Ball & Plate system, there are seven degrees of freedom, namely:

- Angle of the plate: α, β
- Coordinates for ball-position on plate: x_s, y_s

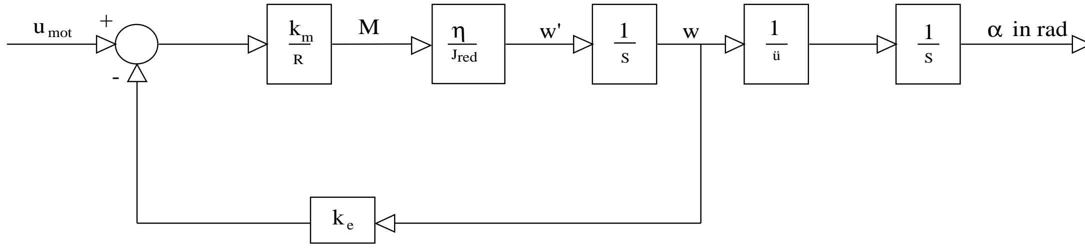

Figure 3.3: Block diagram of Motor, Plate and transmission

- Orientation angle of the ball: ϕ, ψ, θ

However, the system does not have any ability of measuring the orientation of the ball. In our system, we are interested in the x- and y-position, and the orientation of the ball is in the following assumed negligible. A simplified model was derived in [6], following some physical reasons summarized in table 3.2.

Table 3.2: Physical reasons for model of Ball & Plate

Current of motor	$i = \frac{u_{mot} - w^* k_e}{R}$	k_e : electrical constant of motor
Torque	$M = i * k_m$	k_m : magnetic constant of motor
Acceleration of motor axis	$\dot{w} = \frac{M*\eta}{J_{red}}$	J_{red} : moment of inertia η : efficiency
Angular velocity of plate angle	$\dot{\alpha} = \frac{w}{\ddot{u}}$	$\ddot{u}$: relation of reduction between axis of motor and axis of plate

Both the ball and the plate contributes to the moment of inertia J_{red}. However, the ball contribution is assumed negligible due to the following reasoning:

- The plate contribution is much larger than the ball due to the difference in mass.
- The motor is powerful.
- The ball contribution can be considered as disturbance on the positioning system.

From this reasoning, the moment of inertia is assumed constant.

Figure 3.3 illustrates the relation between u_{mot} and the plate angle α in radians, which gives two transfer functions, one for α-direction, and one for β-direction, outlined in (3.2) and (3.3)

$$H_1(s) = \frac{\beta(s)}{U_1(s)} = \frac{a_1}{s(s+b_1)} \tag{3.2}$$

$$H_2(s) = \frac{\alpha(s)}{U_2(s)} = \frac{a_2}{s(s+b_2)} \tag{3.3}$$

The model was identified in [10] and the corresponding state space representation for each axis is given below.

For alpha-direction:

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 700 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -34,69 \end{pmatrix}$$

$$\mathbf{B} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 3,1119 \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\mathbf{D} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

The four states are:

x_1: y position of the ball [cm]
x_2: y velocity of the ball [cm/s]
x_3: α angle of the plate [rad]
x_4: α angular velocity [rad/s]

The input:

U_1: input voltage[V]

The output:

y_1: y position of the ball [cm]
y_2: α angle of the plate [rad]

For beta-direction:

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & -700 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -33,18 \end{pmatrix}$$

$$\mathbf{B} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 3,7921 \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\mathbf{D} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

The four states are:

x_1: x position of the ball [cm]
x_2: x velocity of the ball [cm/s]
x_3: β angle of the plate [rad]
x_4: β angular velocity [rad/s]

The input:

U_2: input voltage[V]

The output:

y_1: y position of the ball [cm]
y_2: β angle of the plate [rad]

3.2 Control task

The "Ball & Plate" system is a fast process requiring the control action to be computed and implemented within few milliseconds. The system is constrained, and the control task is to let the ball follow given trajectories. In order to solve this task, MPC is one of the control strategies available. This means solving on-line, at each time step, the optimization problem outlined in (2.5). Two tools were available in the lab at IFA for implementation of real-time MPC. The first tool consisted of using the Real-Time Toolbox, provided by Humusoft. This toolbox is created for interface with Simulink, and consists of a data acquisition card and software which adds input- and output-blocks to the Simulink library, see section 3.2.1. The second option was implementation by means of the xPC toolbox, provided by MathWorks. The xPC toolbox uses a different hardware setup than the Real-Time Toolbox, and is based on a target-host constellation. xPC needs source code written in "C" to be able to download the model created in Simulink to the target, see section 3.2.2. The MPC algorithm existed only as M-file, and there was a need for coding the algorithm in "C". The implementation of the MPC algorithm real-time was tested out on the "Ball & Plate" system.

3.2.1 Real-Time Toolbox

Real-Time Toolbox is provided by Humusoft, a third-party vendor of MATLAB. This toolbox consists of a data-acquisition card and software supporting this card. The AD 512 data acquisition card is designed for IBM PC compatible computers, and makes it possible to receive signals, process them in the computer and send an output signal. The card contains a 100 kHz throughput 12 bit A/D converter with a sample hold circuit. There are four software selectable input ranges and eight channel input multiplexer, two independent double buffered twelve bit D/A converters and eight bit digital input/output port. The software selectable input ranges are $\pm 10V, \pm 5V, 0-10V, 0-5V$. The output ranges $\pm 10V, \pm 5V, 0-10V, 0-5V$ are set by jumpers on the card, for each analog output. The Real-Time Toolbox is a package for connecting Simulink to the real system. It adds the capability of acquiring data in real time, immediately processing them by Simulink model and sending them back to the real system. All features are implemented through the standard graphical user interface of Simulink. The conversion from simulation to real-time experiments is done by simple means, keeping the basic Simulink scheme made for simulation. The adapter and driver for the data acquisition card has to be specified before real-time implementation. In- and Out-blocks for the signals needs to be incorporated in the model, specifying the channel numbers connected to the system. The real-time experiment is carried out on the PC, which means that no further downloading of any code is necessary.

3.2.2 xPC Target

xPC Target is a host-target PC solution for prototyping, testing, and deploying real-time systems, [20]. It is an environment where the host and target computers are different computers. In this environment a desktop PC is the host computer with MATLAB, Simulink, and Stateflow (optional) to create models using Simulink blocks and Stateflow diagrams. After creating a model, simulations in nonreal-time can be run.

With Real-Time Workshop, Stateflow Coder (optional) and a C compiler on the host computer, the executable code is created. The xPC Target requires the source code of every Simulink block to be written in "C". The Simulink scheme to be downloaded must not contain any call to M-files. Parameters can be passed to a S-function in various ways, by making initialization files, loading parameters into workspace, or have prior blocks giving the correct parameters. After creating the executable code, the target application can be run in real time on a second PC compatible system. There are some special requirements on the hardware and software, listed below.

- The xPC Target software requires a host PC, target PC, and for I/O, the target PC must also have I/O boards supported by xPC Target.
- The xPC Target software requires either a Microsoft Visual C/C++ compiler (version 5.0 or 6.0) or a Watcom C/C++ compiler (version 10.6 or 11.0). In addition, xPC Target requires MATLAB, Simulink, and Real-Time Workshop.

The target PC does not require any operating system, like Windows, UNIX or even DOS, but is instead booted with a floppy. This floppy contains the highly optimized xPC Target kernel, created on the host computer by typing the command "xpcsetup" in the common MATLAB prompt. The kernel makes the target ready for downloading of applications from the host. This application need to have source code written in C. After downloading the application, the kernel makes the target ready for execution, and it is ready for real-time use. The application is written into the RAM at designated places by the kernel.

For communication between the target and the host, there are two options, serial connection or network connection. Network connection is in general faster, up 10Mbit/s instead of 100kBaud/s. More important, it does not restrict the distance between host and target to the length of the serial cable.

3.3 S-function in Simulink

Simulink is a program that runs as a companion to MATLAB, these programs are developed and marketed by the MathWorks, Inc. Simulink and MATLAB form a package that serves as a vehicle for modelling dynamic systems. Simulink provides a graphical user interface (GUI) that is used in building block diagrams, performing simulations, as well as analyzing results. In Simulink, models are hierarchical, giving the ability of viewing a system at a high level, and double-clicking on blocks by using the mouse to go down

through the design levels. Every block in the Simulink model is driven by a S-function. An S-function (system-function) is a computer language description of a Simulink block, [18]. S-functions can be written in MATLAB, C, C++, Ada or Fortran. The S-functions written in language other than MATLAB is compiled with the MEX (Matlab EXecutable) utility. This enables MATLAB to run the compiled file in the Simulink environment. The S-function enables you to interact with Simulink's equation solvers. The most common use of a S-function is in solving tasks like: adding new general purpose blocks to Simulink, adding blocks to represent hardware device drivers, incorporating C-code into simulation or describing systems as a set of equations.

The execution of a Simulink model proceeds in several stages. The first stage is the initialization phase. The library blocks are incorporated into the model, signal dimensions are propagated, sample times, block execution order and allocation of memory. After the initialization, a simulation loop is entered, where each scan through the loop is a simulation step. For a system with no discrete states, and a fixed sample time, the simulation loop consists of the calculation of outputs.

In M-file S-functions, the S-function routines are implemented as M-file subfunctions, [19] . For an M-file S-function, Simulink passes a flag parameter to the S-function. The flag indicates the current simulation stage. Table 3.3 lists the simulation stages and the associated flag value for M-file S-functions.

Table 3.3: Simulation stages

Simulation stage	**Flag (M-file S-function)**
Initialization	flag = 0
Calculation of next sample hit (optional)	flag = 4
Calculation of output	flag = 3
Update discrete states	flag = 2
Calculation of derivatives	flag = 1
End of simulation task	flag = 9

Unlike M-file S-functions, there is not an explicit flag parameter associated with each S-function routine, [19]. In a S-function written in C, Simulink controls execution of the different sections by calling each S-function routine at the appropriate time in the simulation stage. Table 3.4 describes the functions that Simulink calls during the simulation, in the order they are called.

One of the advantages of a MEX-file S-function is the direct access to internal data structure called SimStruct. This is a part of the memory created by Simulink specifically to each block. This makes it possible to use same initialization file for several blocks in

Table 3.4: Functions called during simulation

Simulation stage	S-Function routine
Initialization	mdlInitializeSizes mdlInitializeSampleTimes mdlInitializeConditions
Calculation of outputs	mdlOutputs
Update of discrete states	mdlUpdate
Calculation of next sample hit(optional)	mdlGetTimeofNextTimeHit
Calculation of derivatives	mdlDerivatives
End of simulation task	mdlTerminate

the same scheme, without having parameters mixed. Another advantage is the access to MATLAB's external interface, API, which gives array access and creation functions that can be used in C MEX-files to manipulate MATLAB arrays.

3.4 Programming in C

C came into being in the years 1969-1973, in parallel with the early development of the Unix operating system, [17]. The milestones in C's development as a language during this period are listed below:

- UNIX developed in 1969 – DEC PDP-7 Assembly Language BCPL – a user friendly OS providing powerful development tools developed from BCPL. Assembler tedious, long and error prone.
- A new language "B" as a second attempt. 1970.
- A totally new language "C" a successor to "B". 1971
- By 1973 UNIX OS almost totally written in "C".

The most creative period occurred during 1972, with the introduction of the preprocessor to incorporate macros with arguments and conditional compilation. Another spate of changes peaked between 1977 and 1979, when portability of the Unix system was being demonstrated. In the middle of this second period, the first widely available description of the language appeared: The C Programming Language, often called the "white book" or "K&R". Finally, in the middle 1980s, the language was officially standardized by the ANSI X3J11 committee, which made further changes. Until the early 1980s, although compilers existed for a variety of machine architectures and operating systems, the language was almost exclusively associated with Unix; more recently, its use has spread much more widely, and today it is among the languages most commonly used throughout the

computer industry.

This fact makes C one of the obligatory languages in the art of programming. In control purposes, like in PLC-systems, there is an increasing use of C as the basic language instead of ladder-logic. Programming in C is on a basic level, giving the programmer very much flexibility in creating algorithms and controlling the computers execution of the code, Horton [7]. It is a programmer language for general use, with simple ways of expression, a modern control and data-structure, and a big diversity of operators accessible for the programmer. The lack of constraints in use makes this language more efficient for many tasks compared to more heavy programming languages.

Some of C's characteristics that define the language and also have lead to its popularity as a programming language are small size, extensive use of function calls, loose typing – unlike PASCAL, structured language, low level (BitWise) programming readily available and pointer implementation - extensive use of pointers for memory, array, structures and functions. Some of the features of C is that it has high-level constructs, it can handle low-level activities, it produces efficient programs and it can be compiled on a variety of computers.

C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. C is sometimes referred to as a "high-level assembly language." C was designed this way so that seemingly-simple constructions expressed in C would not expand to arbitrarily expensive (in time or space) machine language constructions when compiled. A simpel and succinctly C program is likely to result in a succinct, efficient machine language executable. Some features like memory allocation and I/O are not included in the C language. However, the usual functions for doing such things are specified by the ANSI C Standard.

The benefits of C is that it has an extensive library for mathematical computations, character analysis, input and output functions, hardware structure, and graphics. While some functions are used more that others, they are all offered and are able to be used by the best programmers. Why write a code to find square roots or alter strings, when there are "header" files that can be used to call these special functions. C is also a relatively easy language to learn, so that you can also write some programs for your everyday life.

Its main drawback is that it has poor error detection which can make it off putting to the beginner. However diligence in this matter can pay off handsomely since having learned the rules of C the programmer can break them. Not many languages allow this. This, if done properly and carefully, leads to the power of C programming.

A C-code contains the normal loops and if-statements, and has a structure which makes it easy to follow. An extensive use of pointers prevents the code from allocating more memory than necessary. First all the variables need to be specified, thereafter

allocation of memory for the parameters on the left hand side of equations. Then the parameters needed is taken into the code, either by means of pointers, or by actually storing them in allocated memory. It is worthwhile to notice that a pointer points to the start of the address where the data is stored. It is straightforward to access the rest of an array when the starting point is known. After calculation and the output taken from the algorithm, there should always be a cleanup of memory allocated. This is necessary to prevent a memory buildup, when new memory is allocated every time the algorithm is run. The code for the C-functions called from Matlab and the DANTZIG-routine is given in appendix A to C, and the S-function written in C is given in appendix D.

Chapter 4

Real-time implementation of MPC

Real-world systems operate under real-time conditions, they are inherently distributed and dynamic. In order to reflect these properties the controlling hard- and software for computer-based systems which monitor, control, or simulate real world processes must in general provide adequate means to cope with time, concurrency, and decentralization. Real-time implementation of the MPC strategy comprises the idea that at each time step, an online optimization is carried out, and the computed optimal control move is applied to the system. The approaches used for implementation of MPC for real-time use is outlined below.

4.1 Extension of the MPC toolbox for real-time use

The main tasks of this project was extension of the MPC toolbox for real-time use, and implementation of the MPC on the "Ball & Plate". The extension of the toolbox consisted of writing the MPC algorithm in "C", for later to use the xPC Target package available with MATLAB R12. The task was limited to code the output sequence from the original M-file S-function in "C", having initialization files to load the variables into the common MATLAB workspace before execution of the code. The initialization M-files are making use of the structure in the original algorithm. By specifying the state space matrices, along with weighting, limits, sampling time, covariance and horizons, the matrices needed for MPC controller is built. Two different approaches were chosen to solve the task of extending the toolbox, briefly outlined below.

4.1.1 Approach 1: MEX-file

This approach consists of coding case 3 of the MPC-algorithm in "C" and keeping the M-file S-function as framework for the Simulink block. The calculation of the output corresponds to the core of the original MPC algorithm. This case 3 of the M-file S-function is replaced with a call to a function written in "C". This function is compiled with the mex utility in MATLAB, which creates an executable file able to be called from MATLAB. Since the original M-file S-function is kept as framework, the buildup of the matrices and their initialization and termination is taken care of by Simulink.

4.1.2 Approach 2: S-function

Approach 2 is similar to approach 1, but differs in the sense that the framework for driving the MPC block is a S-function in "C" instead of a M-file. For initialization and the matrix buildup, the initialization file-name was specified in the initialization pane of the mask. This initialization file is executed once every time the Simulink model is opened and when a new simulation is started. Simulink automatically stores the values in the work-space, locally to each block. This way the matrices resulting from the slightly different state matrices in the two directions x and y, do not interfere.

4.2 Implementation of MPC using Real-Time Toolbox

Early in the project it was decided to try out Real-Time Toolbox provided by Humusoft, a third-party provider of MathWorks. This toolbox was tested extensively on the "Ball & Plate" system, using original algorithm with and without anticipation from file, C-function called from M-file S-function as well as S-function written in "C". The results from these experiments were used to validate the C-code. Noise measurements were carried out in order to get correct values for the covariance matrices. Two different trajectories were given to the controller, namely a square and a circle. The limitations of the controller-performance were tested using various velocities of the ball tracking the reference.

4.3 Implementation of MPC by target-host constellation, xPC Target

Using the S-function written in "C", a new Simulink block was created. It consisted of a masked sub-system calling the MEX-compiled source code. Since only the core of the algorithm is coded in "C", two initialization files to be run prior to simulation were created. All the parameters needed by the S-function were loaded into the common workspace, and passed to the S-function by specifying them in the "parameters and dialog" box of the mask.

Chapter 5

Results from MPC implemented on "Ball & Plate"

5.1 Simulation of M-file S-function and S-function in "C"

Simulation to verify that the original algorithm and the S-function in "C" were identical was carried out. A step change in the input was introduced, having a state space block with the model of "Ball & Plate" to close the control loop. Rise time and overshoot was used as quality criteria for comparison of the simulation runs. Table 5.1 shows the criteria for x- and y-direction for both algorithms. Figure 5.1 and 5.2 shows the step response with the original algorithm after a step change in the reference has been introduced. Figure 5.3 and 5.4 shows the same response with the S-function in "C".

Table 5.1: Quality criteria for comparison of algorithms.

	Direction	**Rise time [s]**	**Overshoot**
Original algorithm	X-axes	1.25	0.221
	Y-axes	1.25	0.2201
S-function in "C"	X-axes	1.25	0.221
	Y-axes	1.25	0.2201

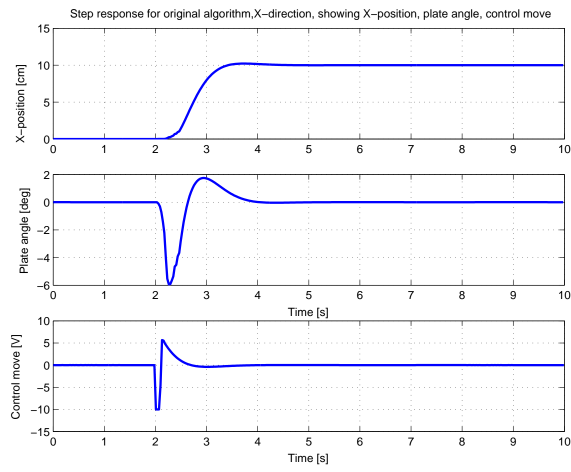

Figure 5.1: Step response with original algorithm, x-direction.

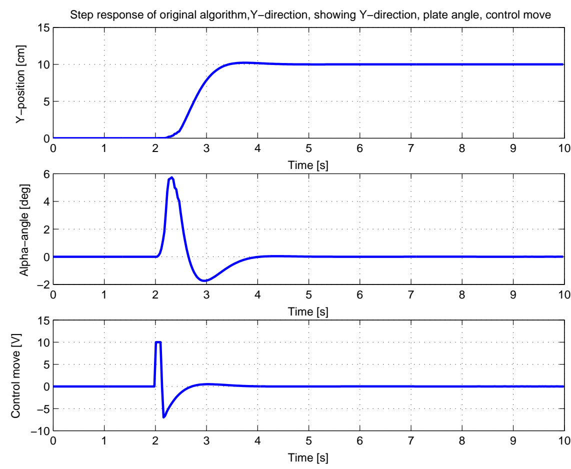

Figure 5.2: Step response with original algorithm, y-direction.

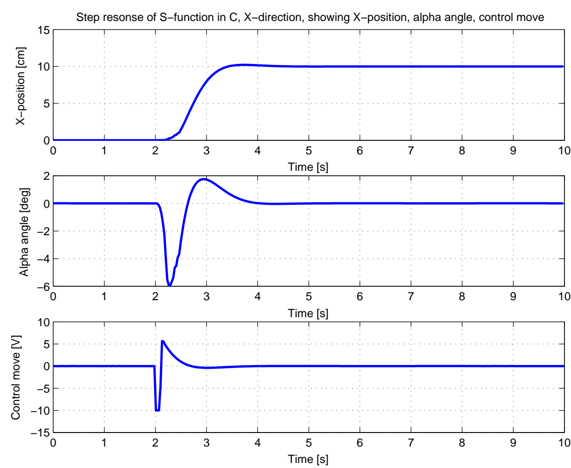

Figure 5.3: Step response with S-function written in C, x-direction.

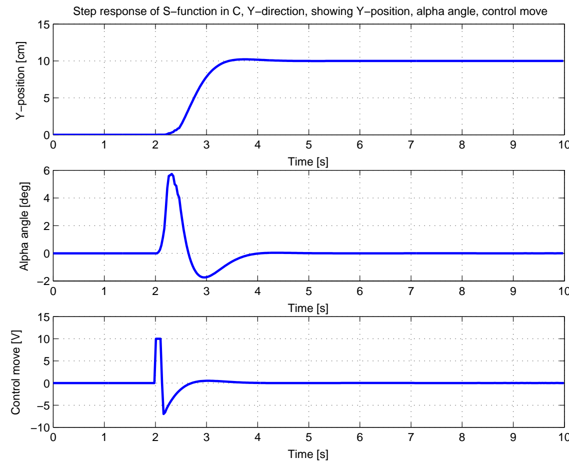

Figure 5.4: Step response with S-function written in C, y-direction.

5.2 MPC implemented with Real-Time toolbox

5.2.1 Step response of "Ball & Plate" with Real-Time Toolbox

In order to check the lag time of the system with Real-Time Toolbox implemented, a step input was introduced at time $t = 1$ seconds and the resulting step response in the plate angle was measured. The step change of magnitude 1, is converted into the highest voltage output in the adapter-card, giving full power to the motors. The sampling time was chosen to be 0.01 seconds. Figure 5.2.1 shows the step response in beta-angle. From the graph the delay was determined to be approximately 0.035 seconds.

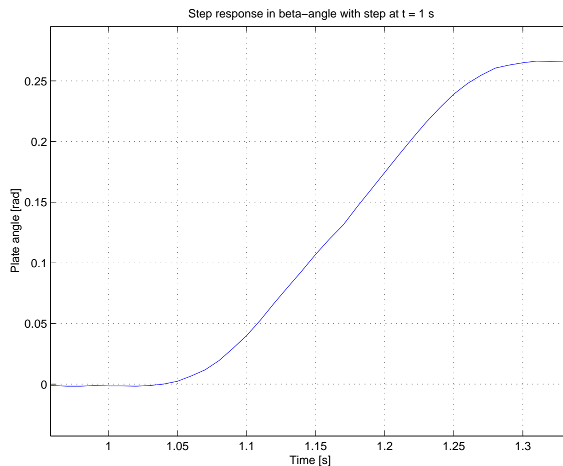

Figure 5.5: Step response with S-function written in C, x-direction.

5.2.2 Anticipative versus non-Anticipative action

Anticipative action can be implemented when the future reference trajectory is known. This reference trajectory is loaded into a file and can be read from the MPC-block at each time step. A filename in the GUI of the MPC-block specifies the MAT file where

the reference *r(t)* and the measured disturbance *v(t)* signals are stored, [3]. The format is the same as in the From File Simulink block. The first variable saved in the MAT-file is loaded from disk and is used to build the reference and measured disturbance signals. The variable is a matrix whose first row is a vector of time t, the following y rows are the reference *r(t)*, and the following v rows are the measured disturbance *v(t)*. Missing rows are treated as zeros. The signals are resampled with the MPC controller sampling time and stored in the MPC block memory. The first (last) sample is used for simulation time before (after) the specified range of t.

To test the effect of anticipation, experiments were run with both a square and a circle as the trajectory. Two M-files were created to calculate the reference trajectory and load the data into a MAT-file. The square was generated by four ramps, giving all the references along the lines describing the square. The circle was generated by a sinus and a cosinus term. In order to change the speed of the ball, different fractions were multiplied with the angular frequency. In the startup of experiments, the ball was placed in center of the plate with no offset in x- or y-direction. Table 5.2 shows weighting, prediction horizon, blocking moves and radius of the reference trajectory circle.

Table 5.2: Parameters during anticipation experiments

Parameters	**Value**
Radius of circle	15 cm
U-weight	0
dU-weight	1
Y-weight	[50 0]
Ts	0.03 s
Prediction horizon	40
Blocking moves	2

Figure 5.6 and 5.7 shows the controller output superposed on the reference with and without anticipation. The reference is here represented by a dotted line. Figure 5.8 and 5.9 shows the tracking of a square reference with and without anticipation, respectively.

Figure 5.10 to 5.19 shows the tracking of a circle reference with and without anticipation at different angular frequencies.

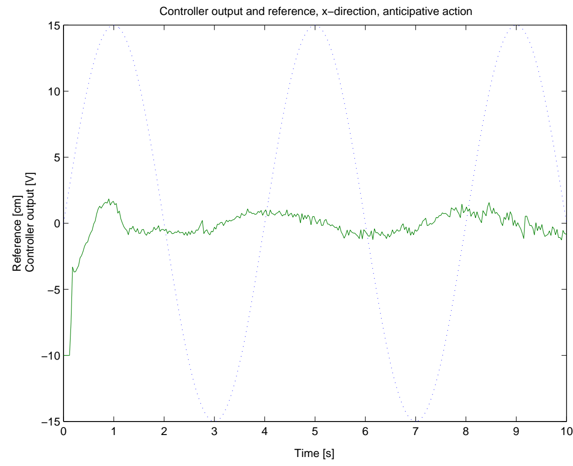

Figure 5.6: Controller output compared to reference, anticipative action.

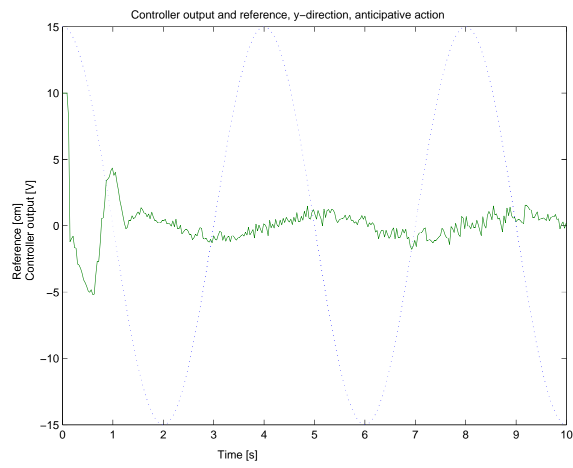

Figure 5.7: Controller output compared to reference, non-anticipative action.

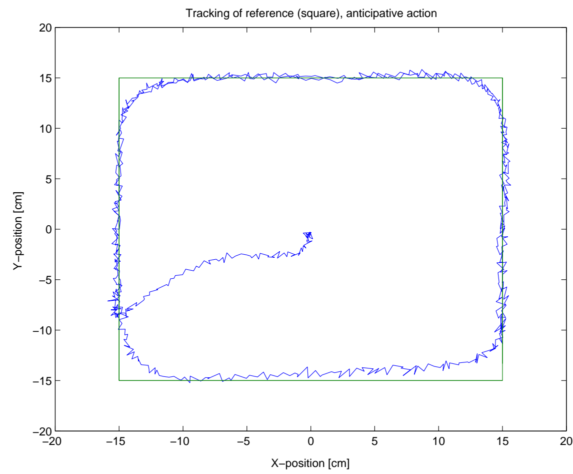

Figure 5.8: Tracking of a square reference, anticipative action.

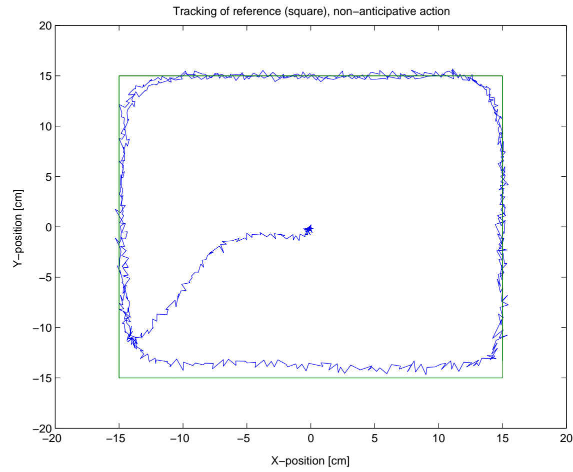

Figure 5.9: Tracking of a square reference, non-anticipative action.

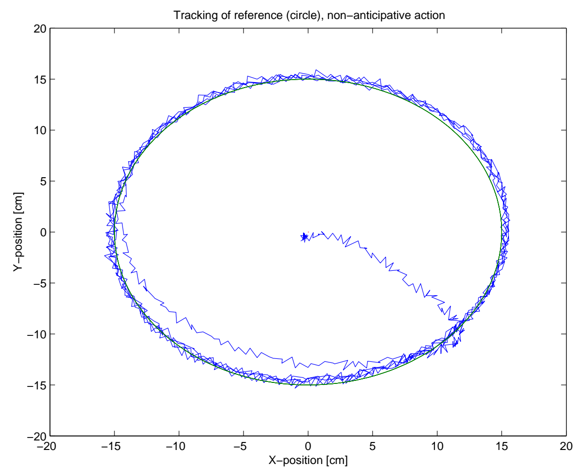

Figure 5.10: Tracking of a circle reference, non-anticipative action. Angular frequency $\omega = \frac{\pi}{3}$ rad/s

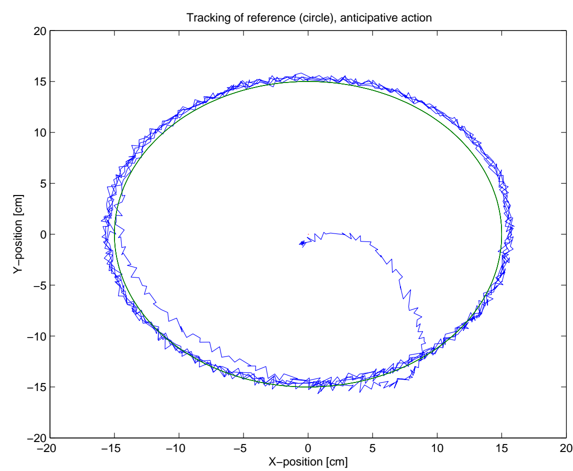

Figure 5.11: Tracking of a circle reference, anticipative action. Angular frequency $\omega = \frac{\pi}{3}$ rad/s

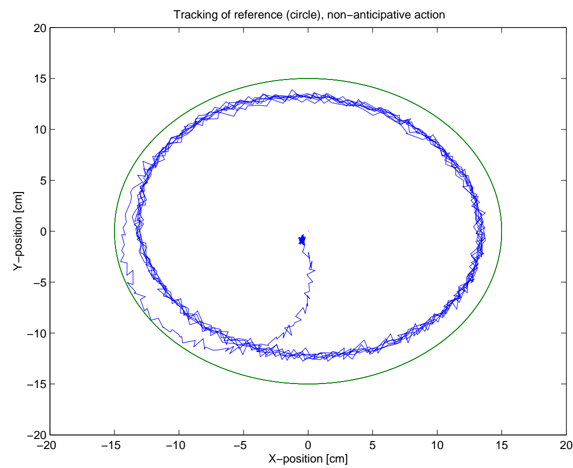

Figure 5.12: Tracking of a circle reference, non-anticipative action. Angular frequency $\omega = \frac{\pi}{2}$ rad/s

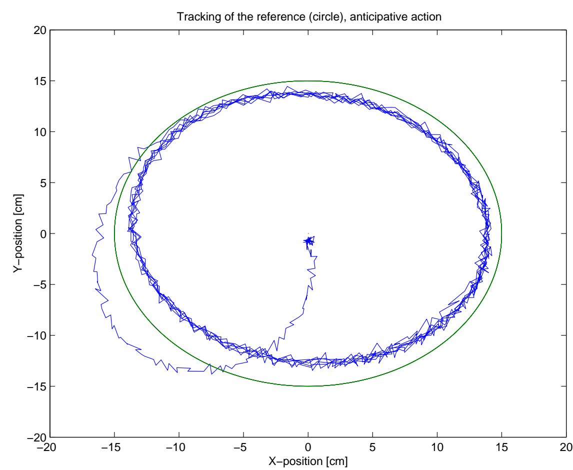

Figure 5.13: Tracking of a circle reference, anticipative action. Angular frequency $\omega = \frac{\pi}{2}$ rad/s

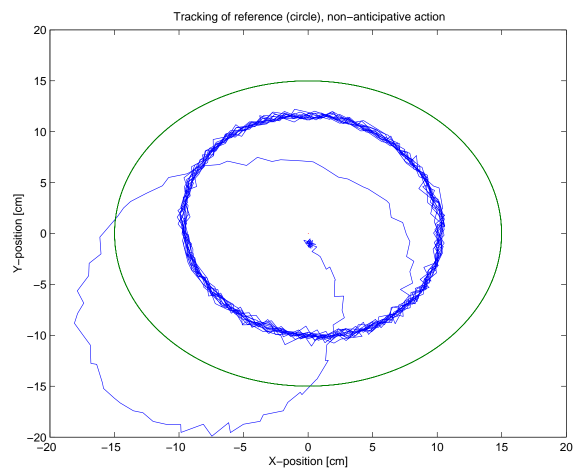

Figure 5.14: Tracking of a circle reference, non-anticipative action. Angular frequency $\omega = \pi$ rad/s

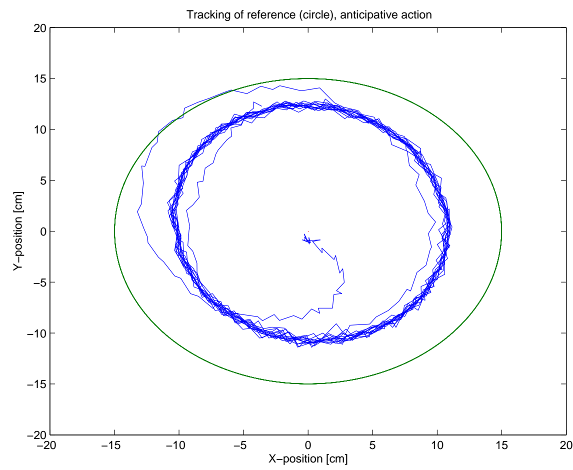

Figure 5.15: Tracking of a circle reference, anticipative action. Angular frequency $\omega = \pi$ rad/s

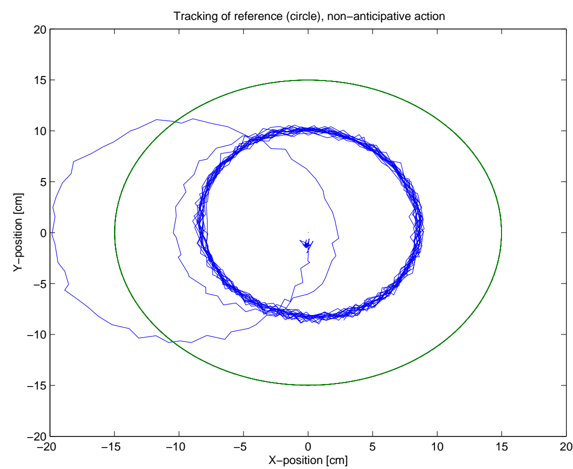

Figure 5.16: Tracking of a circle reference, non-anticipative action. Angular frequency $\omega = \frac{5}{4}\pi$ rad/s

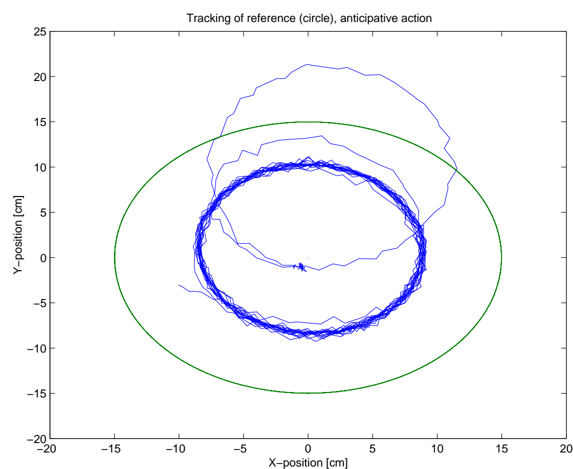

Figure 5.17: Tracking of a circle reference, anticipative action. Angular frequency $\omega = \frac{5}{4}\pi$ rad/s

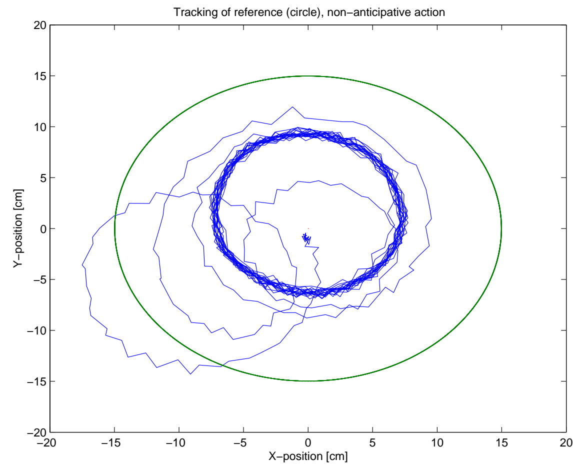

Figure 5.18: Tracking of a circle reference, non-anticipative action. Angular frequency $\omega = \frac{6}{4}\pi$ rad/s

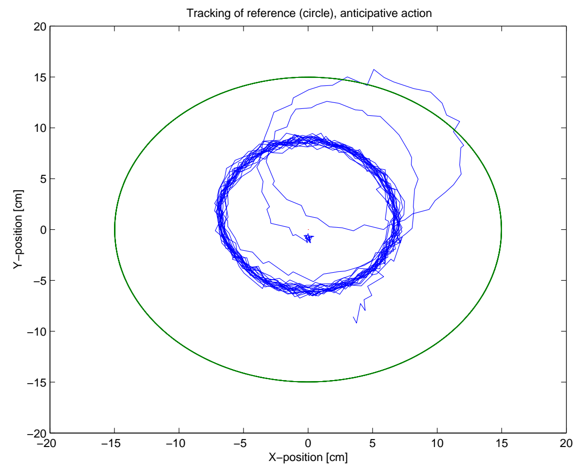

Figure 5.19: Tracking of a circle reference, anticipative action. Angular frequency $\omega = \frac{6}{4}\pi$ rad/s

5.2.3 Calculation time in output stage

When applying MPC on a fast system like "Ball & Plate", the calculation time in the output stage is an important factor for the controller performance. The computation time and statistics for the original algorithm is outlined in table 5.3. Similar is shown in table 5.4 for the algorithm which makes a call to the mex-compiled C-function. In order to strain the controller, simulations with different prediction horizons, giving more complex calculations, were carried out. The simulation time was 30 seconds, and statistics on the elapsed time was run in MATLAB.

Table 5.3: Elapsed computation time in output stage of original algorithm

	Mean[s]	**Min[s]**	**Max[s]**	**Median[s]**	**Std**	**Cov**
Pred. horizon=30	0.014735	0.000000	0.050000	0.010000	0.005276	$2.8 * 10^{-5}$
Pred. horizon=40	0.034555	0.010000	0.060000	0.030000	0.005212	$2.7 * 10^{-5}$
Pred. horizon=50	0.040400	0.010000	0.061000	0.040000	0.003113	$9.7 * 10^{-6}$
Pred. horizon=60	0.057230	0.020000	0.080000	0.060000	0.005007	$2.5 * 10^{-5}$

Table 5.4: Elapsed computation time in output stage of M-file S-function calling C-function

	Mean[s]	**Min[s]**	**Max[s]**	**Median[s]**	**Std**	**Cov**
Pred. horizon=30	0.003900	0.000000	0.020000	0.000000	0.004900	$2.4 * 10^{-5}$
Pred. horizon=40	0.006000	0.000000	0.020000	0.010000	0.004900	$2.4 * 10^{-5}$
Pred. horizon=50	0.009800	0.000000	0.041000	0.010000	0.002000	$4.1 * 10^{-5}$
Pred. horizon=60	0.013700	0.010000	0.040000	0.010000	0.004900	$2.3 * 10^{-5}$

5.2.4 Call to C-function from original MPC algorithm

In this section the M-file algorithm calls the C-function compiled as mex. The initialization is done in the MATLAB file in case 0. After execution of case 0, Simulink passes flag value 3 to the algorithm. This case contains a call to the mex-compiled C-code. A circle was given as reference trajectory for the controller. The offset is 0 (zero), and the amplitude is 10 (ten). The angular frequency, which determines the speed of the ball following the trajectory was chosen to be $\frac{\pi}{2}$ rad/s. The circle trajectory was made by two sinusoidal curves, one with a phase-shift of $\frac{\pi}{2}$. The experiment was run for a time period of 30 seconds, with the ball placed at origin at start of simulation. The settings of the parameters for the MPC-controller is given in table 5.5. Figure 5.20 shows the tracking of the circle with the M-file S-function calling the algorithm written in C.

Table 5.5: Parameter-settings for MPC-controller during experiments

Parameters	**Value**
Prediction horizon	30
Blocking moves	2
Limits(Umin,Umax,dUmin,dUmax,Ymin,Ymax)	-10,10,-inf,inf,[-30 -0.2618],[30 0.2618]
Weights(U,dU,Y)	0,1,[50 0]
Ts	0.03

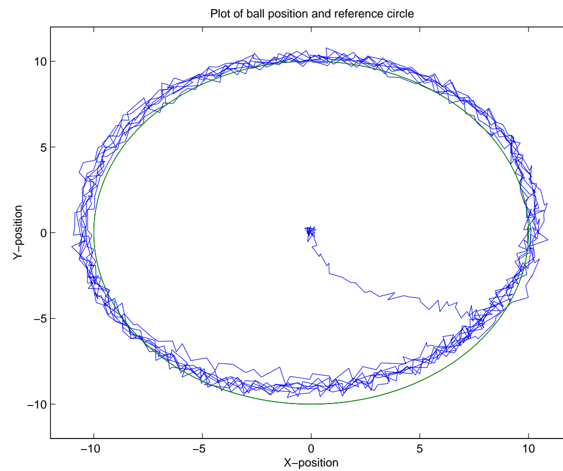

Figure 5.20: Tracking of circle trajectory with M-file S-function calling C-algorithm

From MATLAB, statistics was run on the plots, given in table 5.6 which compares the statistics for the tracking with the reference circle.

Table 5.6: Statistics on ball position, tracking a circle with M-file S-function calling C-function

	Tracking		Reference circle	
	X	Y	X	Y
min[cm]	-11.01	-9.619	-10	-10
max[cm]	10.9	10.78	10	10
median[cm]	0.1572	0.0012	0.03185	-0.9904
mean[cm]	0.0171	0.417	0.005	-0.5479
std	7.177	6.693	6.912	7.216
range[cm]	21.92	20.4	20	20

The controller output in X- and Y-position is plotted in figure 5.21, over the simulation period of 30 seconds.

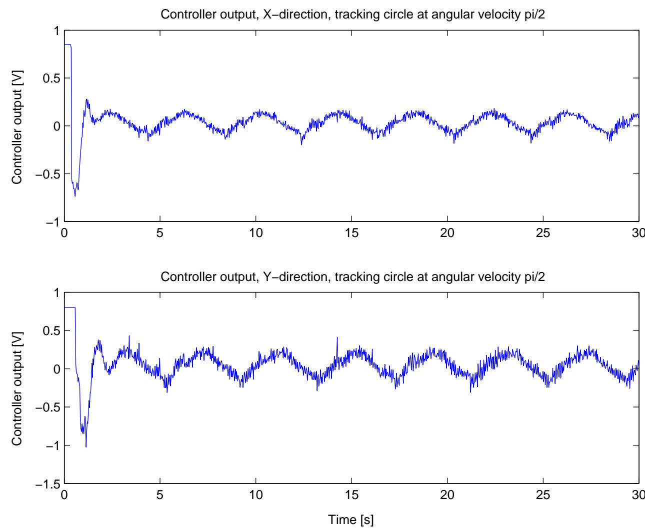

Figure 5.21: Controller output in X- and Y-position when tracking a circle with angular velocity of $\frac{\pi}{2}$ rad/s. M-file S-function calling C-algorithm

5.2.5 S-function written in C

In this section the original MPC-algorithm is removed, having a S-function written in C to drive the block. The initialization is specified in the Graphical User Interface (GUI) of the masked block, running an initialization file before every simulation or experiment is started. This initialization file does the matrix buildup needed for the controller. As in section 5.2.4 the reference is a circle, with the parameters for the MPC-controller given in table 5.7. The settings for the reference trajectory is the same as in section 5.2.4. Figure 5.22 shows the tracking of the circle with the S-function written in C.

Table 5.7: Parameter-settings for MPC-controller during experiments

Parameters	**Value**
Prediction horizon	30
Blocking moves	2
Limits(Umin,Umax,dUmin,dUmax,Ymin,Ymax)	-10,10,-inf,inf,[-30 -0.2618],[30 0.2618]
Weights(U,dU,Y)	0,1,[50 0]
Ts	0.03

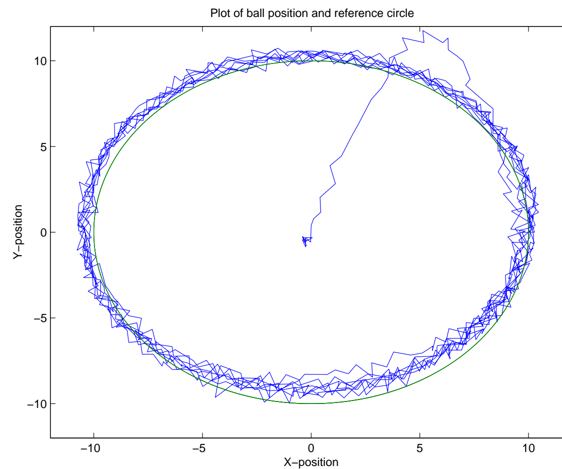

Figure 5.22: Tracking of circle trajectory with S-function written in C

As in the other cases, statistics were run on the plots, given in table 5.8 which compares the statistics for the tracking with the reference circle.

The controller output in X- and Y-position is plotted in figure 5.23, over the simulation period of 30 seconds.

Table 5.8: Statistics on ball position, tracking a circle with C S-function

	Tracking		Reference circle	
	X	Y	X	Y
min	-10.77	-9.969	-10	-10
max	10.47	11.76	10	10
median	0.4413	0.6626	0.03185	-0.9904
mean	0.1725	0.7247	0.005	-0.5479
std	7.218	6.819	6.912	7.216
range	21.24	21.73	20	20

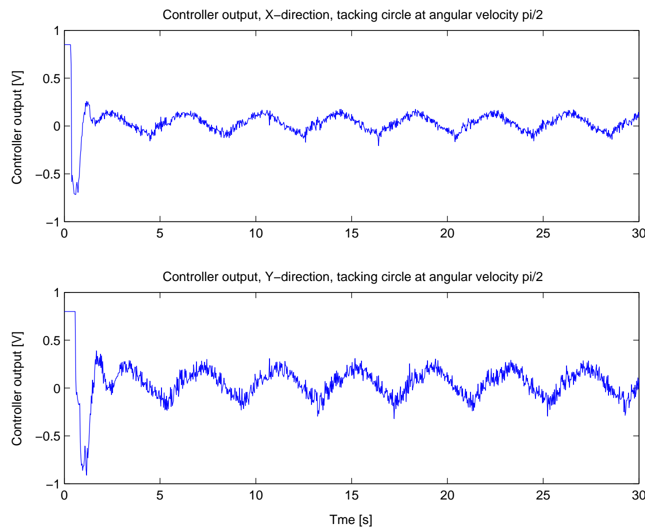

Figure 5.23: Controller output in X- and Y-position when tracking a circle with angular velocity of $\frac{\pi}{2}$ rad/s. S-function written in “C”

5.3 MPC implemented with xPC Target

5.3.1 Step response of "Ball & Plate" with xPC Target

In order to check the lag time of the system with xPC Target implemented, a step input at time $t = 1$seconds was introduced and the step response in plate angle measured. The step input was of magnitude 10, giving full power to the motors. Sampling time was chosen to be 0.01 seconds. Figure 5.24 shows the step response in beta-angle. From the graph the delay was determined to be approximately 0.02 seconds.

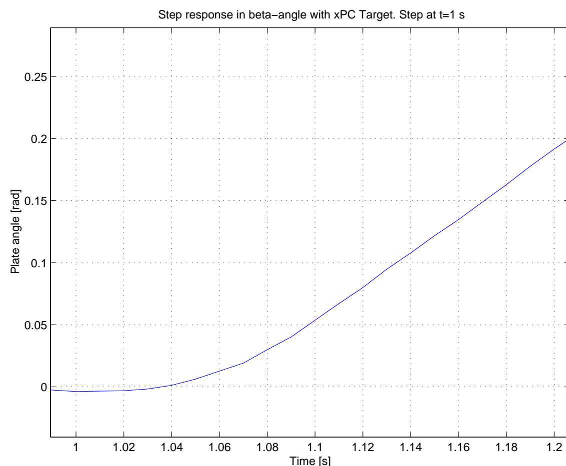

Figure 5.24: Step response with xPC Target, x-direction.

5.3.2 Tracking of reference

Two different trajectories were given to the controller, a square and a circle. The tracking of the square and the controller output is plotted in figure 5.25 to 5.30. The experiments were run for 50 seconds, using point references to determine the square. The parameters used in the experiments are shown in table 5.9. Different weights were tested in order to check the algorithm-performance.

Table 5.9: Parameters for tracking of a square with C S-function

Parameter	**Value**
Prediction horizon	35
Blocking moves	2
U-weight	0
δU-weight	1 & 10
y-weight	[10 0] & [1 0]
Sampling time	0.04 [s]

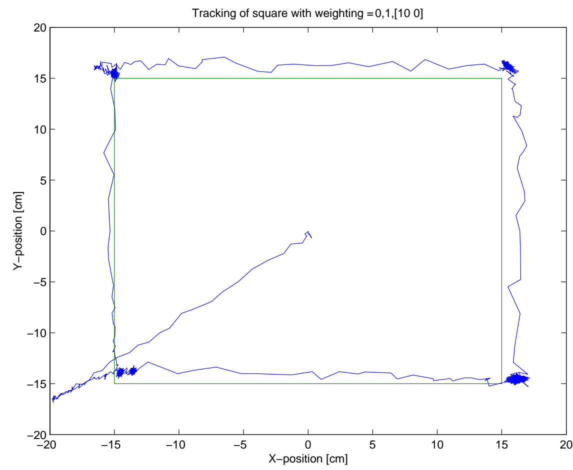

Figure 5.25: Tracking of square-reference with weighting = {0,1,[10 0]}

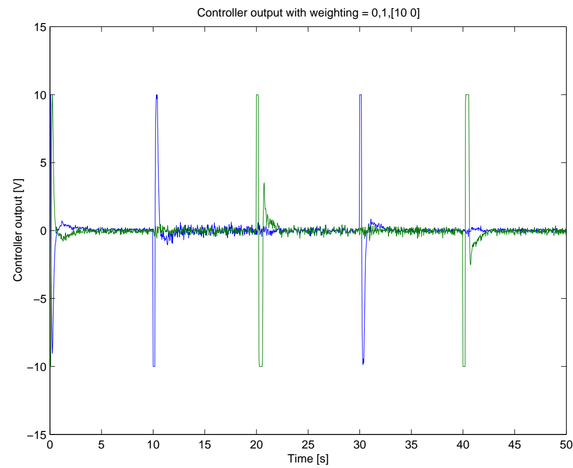

Figure 5.26: Controller output with weighting = {0,1,[10 0]}

A circle with radius 10 cm were given to the controller as reference, and experiments with different tracking velocities were carried out. Figure 5.31 to 5.36 shows tracking of the circle and controller outputs at different tracking velocities. The parameters used in the experiments are shown in table 5.10. The statistics on task execution time (TET) , is shown in table 5.11.

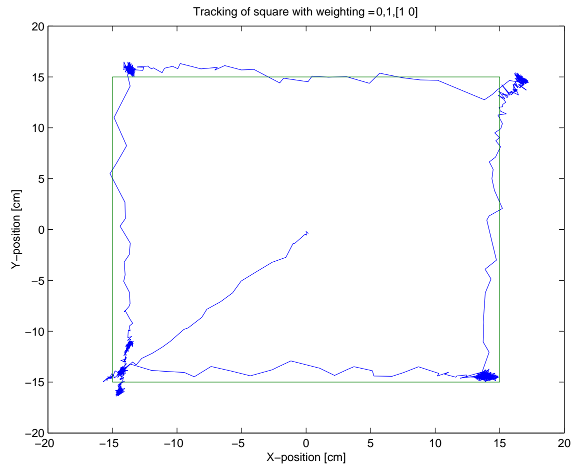

Figure 5.27: Tracking of square-reference with weighting = {0,1,[1 0]}

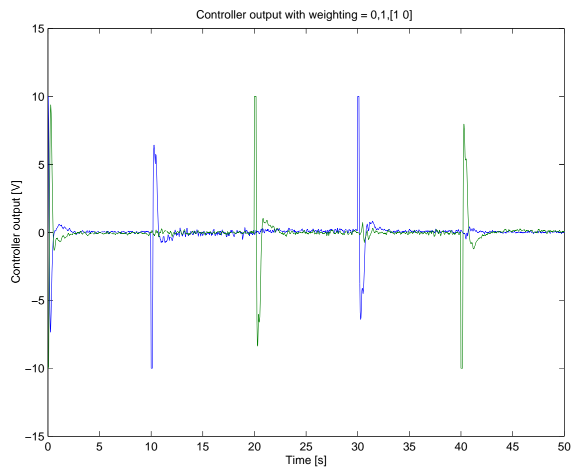

Figure 5.28: Controller output with weighting = {0,1,[1 0]}

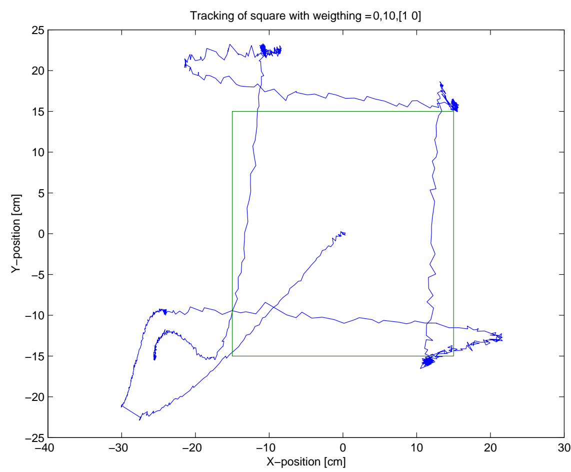

Figure 5.29: Tracking of square-reference with weighting = {0,10,[1 0]}

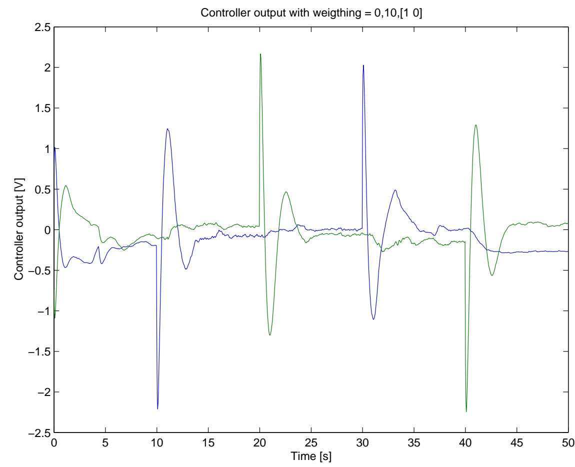

Figure 5.30: Controller output with weighting = {0,10,[1 0]}

Table 5.10: Parameters for tracking of a circle with C S-function

Parameter	**Value**
Prediction horizon	40
Blocking moves	2
U-weight	0
δU-weight	1
y-weight	[10 0]
Sampling time	0.03 [s]

Table 5.11: Statistics on task execution time during tracking of circle

Angular velocity [rad/s]	**Mean**	**Min**	**Max**	**Std**
1	0.0194	0.0193	0.0276	$9.1 * 10^{-4}$
$\frac{\pi}{2}$	0.0196	0.0194	0.0280	0.0010
π	0.0197	0.0194	0.0332	0.0017

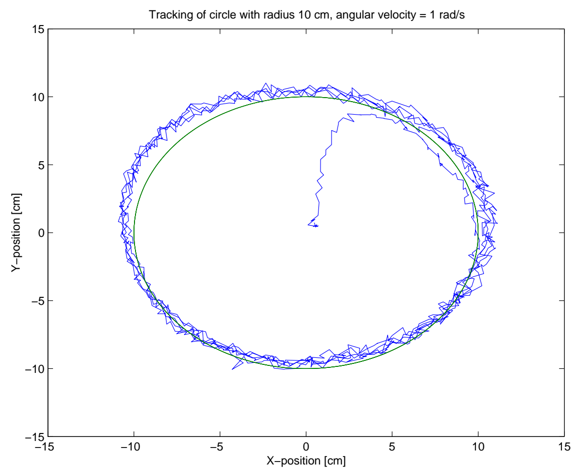

Figure 5.31: Tracking of circle-reference with angular velocity = 1 rad/s

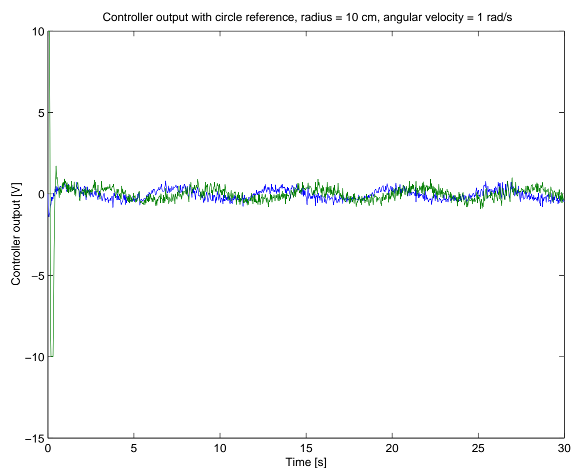

Figure 5.32: Controller output with angular velocity = 1 rad/s

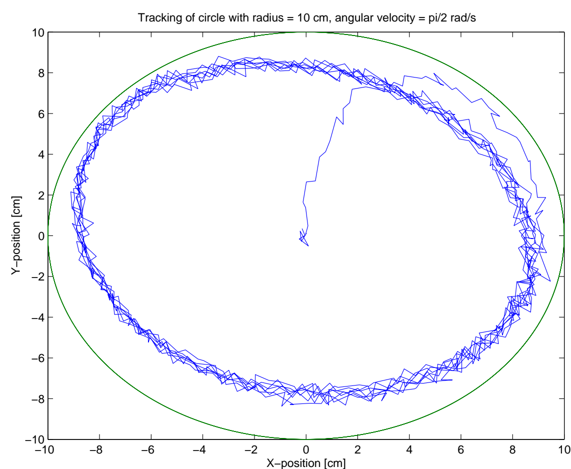

Figure 5.33: Tracking of circle-reference with angular velocity = $\frac{\pi}{2}$ rad/s

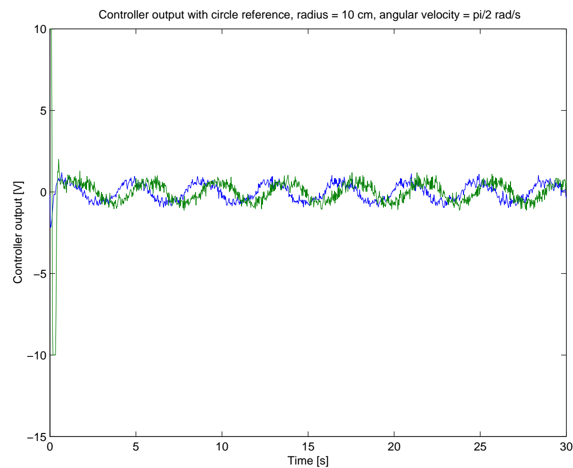

Figure 5.34: Controller output with angular velocity = $\frac{\pi}{2}$ rad/s

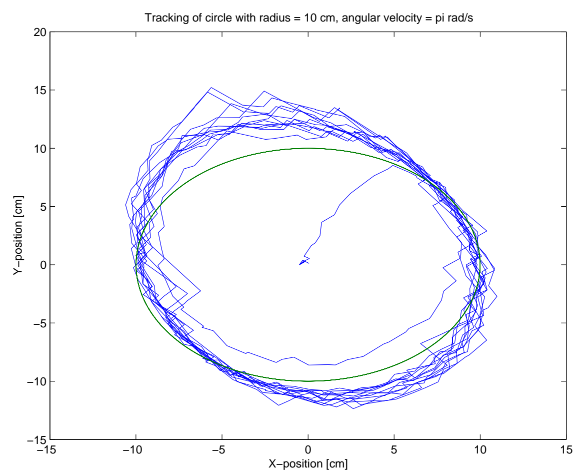

Figure 5.35: Tracking of circle-reference with angular velocity = π rad/s

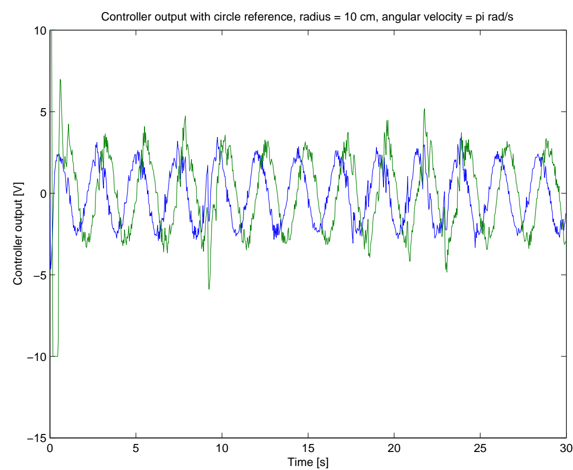

Figure 5.36: Controller output with angular velocity = π rad/s

5.4 Calibration

In order to handle an existing offset in the positioning of the ball, a calibration of the system was necessary. The voltage was measured with the ball fixed in different angles spanning from 10 to 80 degrees. The distance from the laser was chosen to be 0.5 meters. This procedure was done for all four lasers, namely alpha, beta, delta and gamma. The linearized plot for the alpha-laser of the measured angle and voltage against real angle is shown in figure 5.37. This plotting gives the multiplication factor and the summand factor used to set the relation between measured voltage and real angle. The multiplication factor is the fraction between the slope of the plot for true angles and the slope of the linearized plot for measured angles. The offset is the difference between value given by true angle plot and the linearized measured plot multiplied by the multiplication factor. Figure 5.38 shows the error between the measured voltage and the linearized model, for all the different angles of alpha. The same is shown for beta, delta and gamma in figure 5.39, 5.40 and 5.41, respectively. For the alpha-laser, the biggest error is approximately 0.155 degrees. For beta, delta and gamma laser the biggest error is 0.086 degrees, 0.0802 degrees and 0.120 degrees, respectively.

The results of the calibration are given in table 5.12.

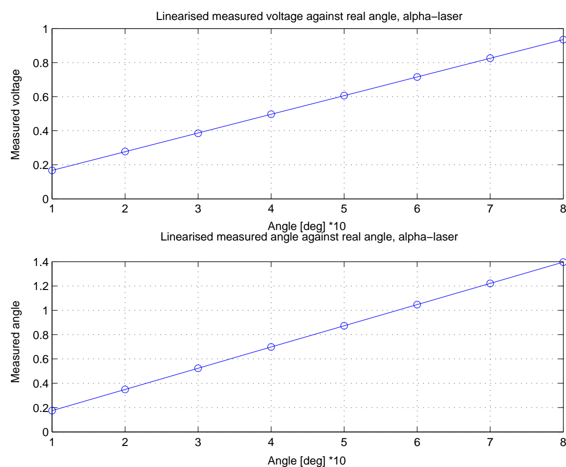

Figure 5.37: Linearized plot of true angles and measured values, alpha-laser

Table 5.12: Parameters for Ball & Plate

Angle	**Multiplication factor**	**Summand**
α	1.5894	-0.0908
β	1.6242	-0.0829
δ	1.5808	-0.0811
γ	1.6265	-0.0832

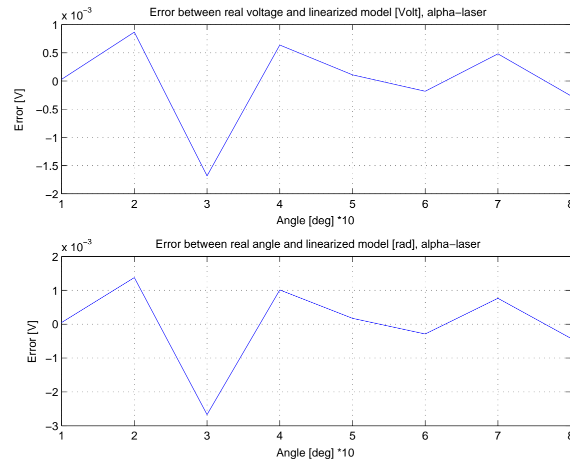

Figure 5.38: Error between the measured voltage and the linearized model, alpha-laser

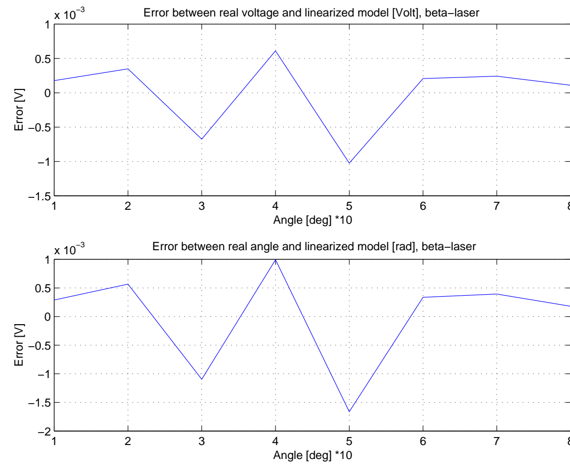

Figure 5.39: Error between the measured voltage and the linearized model, beta-laser

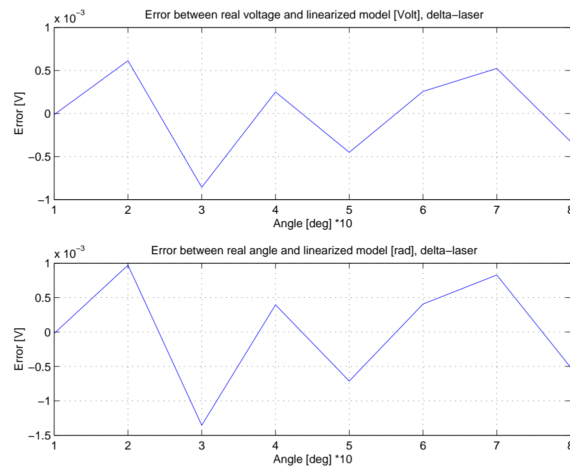

Figure 5.40: Error between the measured voltage and the linearized model, delta-laser

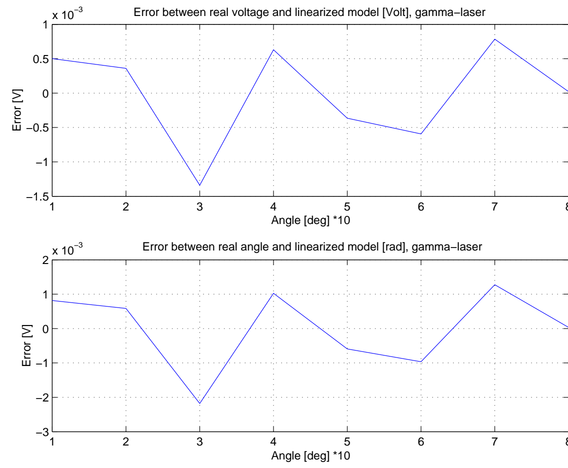

Figure 5.41: Error between the measured voltage and the linearized model, gamma-laser

5.5 Noise

In order to check the noise in the ball position, the ball was fixed at different positions on the plate, as shown in figure 5.42. The plate was kept in horizontal mode during the noise measurements. The measurements were run for 50 seconds, writing the data on x- and y-position to workspace in MATLAB for statistics. The result of these statistics are shown in table 5.14 where the position denoted as $2/2, 2/4, ..., 6/6$ refers to the relative position shown in figure 5.42.

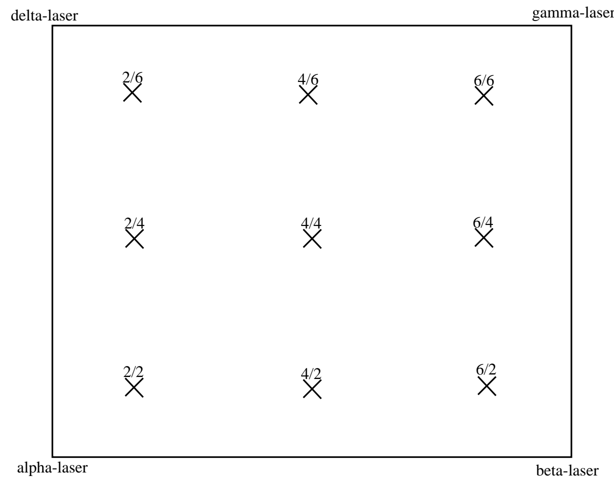

Figure 5.42: Ball positions during noise measurements.

The controller gets two parameters from the decoupled system, the ball position in cartesian coordinates and the plate angle in radians. The angle-span for the plate is 0.593 radians. The measurements were taken over a time period of 50 seconds, and statistics were run in MATLAB, shown in table 5.13

The worst-case values for the variance in the noise measurement is taken into the covariance used to calculate the linear filter gain. The magnitude of the process noise is

Table 5.13: Noise in plate angle.

	Mean[°]	Min[°]	Max[°]	Standarddev.	Variance
Alpha-angle=0	0.1210	$-9.8 * 10^{-4}$	-0.2074	0.0195	$3.8 * 10^{-4}$
Beta-angle=0	0.0012	$-8 * 10^{-4}$	-0.003	$4.10 * 10^{-4}$	$1.7 * 10^{-7}$
Alpha-angle=17	17.3424	17.2079	17.4708	0.0214	$4.6 * 10^{-4}$
Beta-angle=17	17.5614	17.4223	17.6972	0.0300	$9.0 * 10^{-4}$
Alpha-angle=-17	-17.9194	-18.1410	-17.5746	0.0547	0.0030
Beta-angle=-17	-17.9864	-18.2208	-17.8563	0.0325	0.0011

Table 5.14: Noise in ball positioning with Real-Time Toolbox.

		Mean[cm]	Min[cm]	Max[cm]	Median[cm]	Standarddev.	Covariance
2/2	X-pos.	-19.8657	-20.6199	-19.1197	-19.8548	0.2035	0.0414
	Y-pos.	-19.6616	-20.4003	-18.9761	-19.6096	0.2609	0.0680
4/2	X-pos.	-0.1287	-1.0733	0.8124	-0.1475	0.2978	0.0887
	Y-pos.	-19.6606	-20.4677	-19.0019	-19.6232	0.2938	0.0881
6/2	X-pos.	19.5201	18.1586	20.5224	19.4495	0.3692	0.1363
	Y-pos.	-19.7059	-21.1507	-18.7057	-19.6440	0.4050	0.1640
2/4	X-pos.	-19.9055	-20.7097	-19.3334	-19.8779	0.2548	0.0649
	Y-pos.	0.1024	-0.5868	0.7598	0.1305	0.2259	0.0510
4/4	X-pos.	-0.1409	-0.6975	0.2822	-0.1359	0.1643	0.0270
	Y-pos.	0.1101	-0.5604	0.7330	0.1408	0.2476	0.0613
6/4	X-pos.	19.5698	18.9523	20.1169	19.5795	0.1761	0.0310
	Y-pos.	0.0538	-0.9828	0.8957	0.0499	0.300	0.0900
2/6	X-pos.	-19.7863	-20.3790	-19.0811	-19.7962	0.2201	0.0484
	Y-pos.	19.7782	19.1718	20.4312	19.7758	0.1997	0.0399
4/6	X-pos.	-0.0601	-0.6234	0.5412	-0.0598	0.1717	0.0295
	Y-pos.	19.7265	18.8105	20.6175	19.6983	0.2820	0.0795
6/6	X-pos.	19.5163	18.9055	19.9825	19.5242	0.1465	0.0215
	Y-pos.	19.6917	18.7473	20.4924	19.6647	0.2781	0.0774

Table 5.15: Noise in ball positioning with xPC Target.

		Mean[cm]	**Min[cm]**	**Max[cm]**	**Median[cm]**	**Standarddev.**
2/2	X-pos.	-20.130	-20.5148	-19.7190	-20.1276	0.1292
	Y-pos.	-20.1578	-20.6600	-19.5246	-20.1481	0.1713
4/2	X-pos.	-0.6974	-1.8074	0.0014	-0.6655	0.3267
	Y-pos.	-20.0822	-20.6804	-19.4724	-20.1001	0.2259
6/2	X-pos.	19.2431	17.6810	20.2165	19.1932	0.3509
	Y-pos.	-20.0916	-20.9396	-19.2267	-20.1133	0.2406
2/4	X-pos.	-20.0937	-20.4515	-19.7239	-20.0980	0.1053
	Y-pos.	0.0758	-0.5176	0.6352	0.0666	0.2028
4/4	X-pos.	-0.2930	-0.8459	0.2578	-0.2977	0.1710
	Y-pos.	-0.2479	-0.9612	0.3534	-0.2565	0.2462
6/4	X-pos.	19.5844	18.5628	20.5816	19.5400	0.3238
	Y-pos.	-0.3700	-1.1611	0.3758	-0.4002	0.2744
2/6	X-pos.	-19.9353	-20.3518	-19.3873	-19.9441	0.1438
	Y-pos.	19.9258	18.9174	20.8791	19.9223	0.2876
4/6	X-pos.	-0.1150	-0.8824	0.5954	-0.1305	0.2098
	Y-pos.	19.6628	18.9892	20.3336	19.6252	0.2902
6/6	X-pos.	19.7016	18.8021	20.4324	19.7500	0.2883
	Y-pos.	19.4020	18.8066	20.1866	19.3626	0.3037

Table 5.16: Covariance matrices for "Ball & Plate".

Direction	**Covariance matrix**
X-axes	$\begin{bmatrix} 1 * 10^{-7} \end{bmatrix} \begin{bmatrix} 0.136 & 0 \\ 0 & 1.1 * 10^{-3} \end{bmatrix}$
Y-axes	$\begin{bmatrix} 1 * 10^{-7} \end{bmatrix} \begin{bmatrix} 0.164 & 0 \\ 0 & 3.0 * 10^{-3} \end{bmatrix}$

represented by the covariance matrix. The resulting covariance matrices for both x- and y-position are shown in table 5.16. The process noise, represented by the first element of the covariance matrix was set to a small value ($1 * 10^{-7}$). It was here assumed that the noise in measurements of position and plate angle is by far the strongest contributor to the noise. The validity of this is discussed in section 6.5.

Chapter 6

Discussion

6.1 Theoretical assumptions

As mentioned in section 3.1.1, some assumptions were made for the simplified model. This project task did not include the modelling of the system, and the model was therefore assumed correct throughout the project. However, a motor was changed after the model had been verified, and this might have caused an inaccuracy in the model of the real system. Compared to simulation runs, some tuning of weighting parameters was necessary in order to implement the MPC for real-time use without hurting the constraints. This fact leads to the recommendation that a verification of the model needs to be carried out.

6.2 Extension of MPC toolbox

Two possible solutions to the problem consisting of how to create, and then pass the necessary parameters to the S-function in "C", were considered. The first option consisted of making an initialization file, and specify this in the initialization pane of the masked S-function written in "C". Second option was splitting up the Simulink block in two, one driven by the M-file S-function, the other by the S-function coded in "C". The first block will in this constellation create the parameters needed for calculation of the output in the second block. However, this latter approach was abandoned because of the inefficiency of calculations when splitting up the Simulink block.

For the QP-solver, an DANTZIG-algorithm written by N.L. Ricker was used. At every scan through this algorithm, memory for the variables needed is allocated. There was no deallocation of used memory, which caused the target to crash after approximately 40 scans. It was a considerable memory buildup, which also slowed the algorithm down. The routine works when using a PC since the MATLAB memory management facility automatically frees all nonpersistent memory whenever a MEX-file completes. This does not happen on the target since it is only driven by the kernel made in Real-Time Workshop and has no operating system. Nevertheless, it is in general a good programming practice

to deallocate memory as soon as the algorithm is through using it. Doing so generally makes the entire system run more efficiently. In order to handle the memory buildup, work vectors needed to be created. The memory for these vectors is allocated in the initialization section of the S-function, and pointers to this memory are passed down in the function calls. The sub-functions are then able to access the allocated memory, and there is no need for further memory allocation. Deallocation of the memory is happening at the termination stage of the S-function. Simulations on PC show that the time spent in the calculation of output went considerably down when freeing memory, see section 5.2.3.

6.3 MPC implemented with Real-Time Toolbox

The Real-Time Toolbox has the advantage of easy conversion from a Simulink model to real-time experiments. It adds the capability of acquiring data in real time, and is supposed to immediately process them by MATLAB commands or a Simulink model in order to send them back to the outside world. However, it does require a lot of CPU-power in the data acquisition and processing. The PC was driven by a 933 MHz Pentium 3, and MATLAB had to be given top priority in the Windows Task Manager. By looking at the plot of the step response in section 5.2.1, it is clearly seen that the system has a delay of approximately 0.035 seconds. This corresponds to more than one sampling time, which contributes to question the toolbox's capability of implementing MPC real-time. During the experiments I also encountered an excessive loss of ticks, which limited the controller performance. Every block in the Real-Time Toolbox has its own timer, and the loss of ticks is a mechanism use to detect system overload. A tick is lost in the situation when a timer is due to execute and a request comes to execute the same timer again. Each timer counts its own lost ticks and when they reach the number specified in its "MaxLostTicks", the timer is stopped.

Another feature of the toolbox is that it continues to output the last computed increment even after experiments have been stopped. The system is driven to constraint, and there is a risk for stalling the motors. The problem was solved by having a manual switch in the scheme, giving opportunity to set the output to nil.

As shown in section 5.2.2 the tracking of trajectory for both square and circle is good when the speed of the ball is low. When slowly increase the speed of the ball, the error between reference and ball-position gets bigger. The main reason for the decrease in tracking radius is due to the fact that the plate deflection is increased at high tracking velocities. The component of gravity directed towards the center of the plate will increase with increasing plate-deflections, which causes a reduction in tracking radius. However, the MPC controller tended to be unstable during startup of experiments, which may be a result of the delay and/or the loss of ticks. This non-consistency happens when the MPC algorithm calculates a big increment in the control output, and there is a loss of ticks in

the next scan. The Real-Time Toolbox responds to this loss of ticks by implementing the last computed control increment, which results in a too big deflection of the plate. At next sampling time, the controller has to counter-act this behavior, and an even bigger increment is calculated. The resulting behavior is unstable.

6.3.1 Calculation time

The Real-Time Toolbox was tested extensively on the "Ball & Plate" system using three different algorithms, namely the original M-file S-function, call to the MEX-compiled C-function and the S-function written in C. The main problem with using this toolbox is concerning the calculation time spent in the algorithm. The loss of ticks, explained above, causes the toolbox to apply the last calculated control move for next sampling time. This results in a bad controller performance, and eventually a crash when the maximum number of ticks is reached. In order to prevent this, the prediction horizon had to be kept lower than 35 moves. The results in section 5.2.3 clearly shows that the original MPC algorithm has a too high computational cost with a prediction horizon more than 30. The sampling time used during the experiments was chosen to be 0.03 seconds. This sampling time was derived in Moschibroda & Muntwyler [15] from the fastest dynamics of the system, which determines the slowest possible sampling rate. However, these values give a time horizon of 0.9 seconds, which is too short in cases where big deflections in the plate angle are necessary. A too short time horizon prevents the controller from computing the error of a future overshoot, and the constraints will be violated. A sampling time bigger than 0.03 seconds tends to make the controller performance more sluggish, so a trade off needed to be carried out.

By comparing the elapsed computation time in table 5.3 with table 5.4, the algorithm written in "C" is approximately 4 times faster than the original M-file algorithm in solving identical problems. However, the first scan in the C-code implies memory allocation, and is the reason why this scan takes considerably more time than the following execution of the code.

6.3.2 Anticipative versus non-Anticipative action

Anticipative action was tested out solely on the original MPC algorithm, since this feature has still yet to be implemented in the S-function written in "C". The MAT-file specified in the GUI of the MPC-block contains information on how the reference is going to change and the time associated with the change. The effect of anticipative action is clearly shown when comparing figure 5.6 and figure 5.7. These figures show the control output compared to reference given to the controller. The reference is a sinusoid curve in each axes, with a phase-shift of $\frac{\pi}{2}$ rad in order to form a circle. With anticipative action, the control-action is shifted slightly in time compared to non-anticipative action. This is because the MPC controller has a *priori* knowledge on how the reference is going to change, and therefore incorporates this information in the optimization problem. The optimal solution is to

implement a control move slightly before the actual change in reference has taken place. This is why the chain of control moves is shifted approximately 0.5 seconds in time with a tracking speed of $\omega = \frac{\pi}{2}$ rad/s. The advantage of knowing the reference is seen when there is a fast tracking of the reference. Anticipation has an stabilizing ability on the system during fast tracking. With angular frequency more than π rad/s, anticipative action has a better performance with less overshoot and smoother behaviour. The system was clearly unstable at high tracking speeds, as expected. With an angular frequency greater than $\omega = \frac{6}{4}\pi$ rad/s, the controller was hurting the constraints constantly, both with and without anticipation. This is due to hardware constraints, the plate cannot move infinitely fast.

During the testing of the anticipation feature of MPC toolbox, some bugs in the algorithm were encountered. They were related to how the reference were read from the file and taken into the algorithm. These have now been fixed for future releases of the MPC toolbox.

6.3.3 Algorithms written in "C"

In section 5.2.4 and 5.2.5 the performance of the new algorithms is tested on tracking a circle reference with angular velocity $\frac{\pi}{2}$ rad/s. The plots shows the same performance as the original algorithm without anticipation at the same angular velocity. From these results and the simulation run in section 5.1 it is concluded that the algorithm is successfully translated from M-file to function in "C".

6.4 MPC implemented with xPC Target

During the setup of the target-host constellation, some problems were encountered regarding the linking between S-function and the functions to be called. In order to solve this, the two files to be called from within the S-function were included in the S-function file. The subfunctions are called with standard function calling routine. This way there was no need for the compiler to have any further linking specified. The compiler used in the xPC Target setup was specified to be "lcc", a standard compiler shipped by installation of MATLAB R12.

The xPC Target with Real-Time Workshop was tried out for real-time use on the "Ball & Plate" system. There are two ways of connecting the target and the host computer, either by means of serial cable or by TCP/IP. As outlined in section 5.3, the TCP/IP is in general preferable because of speed and no limitations on distance between computers. However, in this work a serial cable using the RS232 ports was used for connection. The reason for doing this was some problems with getting correct IP-addresses. Nevertheless, the TCP/IP connection should be set up for future work.

The controller parameters needed to be tuned on the target in order to get good per-

formance without hurting the constraints. As outlined in section 6.3.1, the time horizon for the MPC controller needs to exceed 0.9 seconds in order to ensure stability. The time horizon is influenced by two parameters, namely the prediction horizon and the sampling time. By increasing the prediction horizon, the complexity of the calculations increases, which slows the algorithm down and eventually gives an CPU overload on the target. The peak in task execution time is encountered during the first scan because of the memory allocation taking place. This scan requires approximately 40 percent more CPU power than a scan without any memory allocation. When increasing the sampling time, the controller gets more sluggish. After some testing, a prediction horizon of 35 and a sampling time of 0.04 seconds were chosen as appropriate parameters for having good controller performance. From this result, it is clear that the target is operating on its limits of performance. In order to improve this further, the execution of the code could be made faster, or the processor speed could be increased.

One of the advantages with xPC Target is that it ensures experiments to be run in real time. This because of the fact that it does not have any operating system, and it gets from this reason no interrupts during execution of the code. The controller behavior was clearly more smooth than when using the Real-Time Toolbox. Even though the speed of the host processor is 3.5 times faster than the target processor, the controller performance of the target was more stable than with the Real-Time Toolbox, given the same set of parameters.

From the results in section 5.3, it is clear that the weighting has the expected influence on the controller performance. In general, the controller gets more aggressive when input horizon is increased, output horizon is decreased, output weight is increased and input weight is decreased. Due to the fact that the controller was strained with respect to time horizon, only the ratio between input and output weight was changed. It is shown that the controller performance gets more sluggish when the ratio between output and input weight is decreased. The plots of controller output shows that the controller does not drive the motors as hard when the weighting ratio is deceased. This makes it slower and the settling time rises from 2 to 5 seconds. Comparison of the tracking of the square reference in section 5.3 compared to section 5.2.2 seem to show that the Real-Time Toolbox has a better performance in tracking the square reference. However, the reference in the two cases are slightly different. In section 5.2.2, the square is described by four ramps, producing reference positions for the controller to track at each sampling time. In section 5.3, the square is described as four points, namely the corners of the square. In this latter case, the controller does not have reference positions along the path between corners, but is getting step changes in the reference.

6.5 Noise

Having good information about the noise is important for mainly two reasons, it makes it easier and/or possible to find proper parameters for the MPC-controller, and secondly

the design of the linear filter gain makes use of the covariance matrix. The covariance matrix consists of measurement noise and process noise. Since the linear filter gain is strongly influencing the calculated control increments through the measurement update in the algorithm, the controller performance depends on correct values for the covariance matrix. I chose to measure the noise after the calculation of the cartesian coordinates instead of measuring angle positions either in radians or degrees. This due to the fact that the MPC-controller gets the signal in cartesian coordinates, and therefore will get the noise after the signal from the lasers has propagated through the conversion from radians to cartesian coordinates. The magnitude of the covariance matrix was increased with approximately 10 times compared to former groups. The reason for this deviation in noise compared to former work could originate from several sources. Firstly, the speed of the motor driving the mirrors is not constant. There was no possibility of measuring the magnitude of this variation, but during experiments the motor slowed down when the plate had big deflections. This will cause variations in the positioning of the ball, and thereby a noisy behavior of the controller. The wiring from the "Ball & Plate" to the PC is also producing noise. There was a problem with the connection between the system and the PC since a DB-37 connector needed to be mapped against a DB-25 connector. In order to solve this two external connectors was used, mapping the right pins. This solution is not optimal, and could easily be improved by creating a cable with one DB-37 and one DB-25 connector. The system is not very robust, and the mirrors were displaced several times, causing the bearing on the mirror-axes to be worn out. With a defect bearing, the mirror position fluctuates slightly, which results in measurement noise. The measuring of the position is sensitive to direct light and to the cleanliness of the ball. If the ball is cleaned thoroughly, and no direct sun-light affects the ball, the noise in positioning is reduced. As a result from this reasoning, the measurement noise was considered to be the by far biggest contributor to the total noise. Experiments show that the controller behavior was smooth when the new values for measurement noise were incorporated in the covariance matrix.

From comparing table 5.14 and 5.15, it is shown that the noise in the x- and y-positioning with xPC Target is less than with the Real-Time Toolbox. However, one laser was malfunctioning when implementation of MPC by xPC Target was carried out, which resulted in an increase of noise in position $6/2, 6/4$ *and* $6/6$, see figure 5.42. As a result from these noise-measurements and the update of the covariance matrix, the "nervous" behavior was considerably reduced for both MPC implemented with Real-Time Toolbox and xPC Target. The Kalman filter is functioning as a low-pass filter, smoothing the controller performance.

Chapter 7

Conclusion

The primary objective of this project was to extend the Model Predictive Control (MPC) toolbox for real-time use. The basic MPC algorithm was coded in "C" and interfaced with Simulink by use of a S-function structure. Two tools for implementation of MPC for real-time use was tested, namely the Real-Time Toolbox and xPC Target. The implementation of MPC by means of the toolbox encountered mainly two problems, a delay and a system overload. On a fast system like the "Ball & Plate", which requires the control action to be implemented within few milliseconds, a delay of 0.035 seconds corresponds to more than one sampling period and causes a "shaking" behavior. In addition, the toolbox caused a system overload when the execution of the algorithm required more than one sampling period. The implementation of MPC by this alternative is demanding on excessive amounts of CPU-power. In order to meet these demands, a PC with a 933 MHz processor was used. However, even though MATLAB was given top priority in the system manager, the complexity of calculations had to be limited by means of keeping the prediction horizon in the range of 30 steps. The problems with system overload and delay leads to the conclusion that the toolbox is not able to implement MPC real-time on a fast system with characteristics like "Ball & Plate".

The task of extending the MPC toolbox for real-time use, consisted of creating source code of the MPC algorithm in "C", and then interface this with Simulink. The task was confined to the output sequence of the algorithm, with the initialization done by two initialization M-files. The results show a successfully extended MPC algorithm for real-time use. In order to test the weights influence on the controller performance, different ratios between the output weight (y-weight) and the increment weight (δU-weight) were applied. The controller performance became clearly more sluggish when this ratio was decreased, which shows that the controller responds to the weighting in a correct manner. The variation in the calculation time of the algorithm indicates that the bottle-neck is the Quadratic Programming (QP) solver. This because of the fact that the only time-consuming *varying* factor in the algorithm is the number of iterations in the QP solver, indicating that an optimization with respect to time need to be carried out. The routine was slightly modified with respect to memory allocation in order to be run in the xPC setup. The calculation

time required for one scan was considerably reduced when the memory allocation was done in the initialization section of the S-function.

The xPC Target ensures real-time execution, with considerably less delay and noise than the Real-Time Toolbox. It is from this reason recommended as standard setup for implementation of MPC on the "Ball & Plate".

Zürich, June 13, 2001

Thomas Haugan

Chapter 8

Recommendations

From the experiments which show that there is reason for questioning Real-Time Toolbox's capability of applying MPC real-time, the xPC Target should be set as framework for implementing real-time MPC on the "Ball & Plate" system. However, at high computational loads, the target tended to crash because of a CPU overload. To prevent this, an optimization of the C-code is strongly suggested. Since an extensive use of loops for matrix calculation is used, different ways of storing the values should be investigated. The bottle-neck in the algorithm is the DANTZIG-routine. This can be seen from the variation in the task execution time. Since the size of the parameters are constant during simulation, the only time-consuming varying factor is the number of iterations. This routine should be optimized with respect to speed, including efficient ways of copying the parameters passed in the function call.

The initialization section should also be incorporated in the S-function in "C" to make the algorithm complete. This would make the MPC toolbox fully extended for use with xPC Target. A comparison with the already implemented piecewise linear MPC algorithm could also be carried out. When this initialization section is implemented it may be preferable to split the file into several files to ensure readability of the code. This requires a new build-file which specifies for the compiler in which order to link the files.

The opportunity to use anticipation from file should also be implemented in the new algorithm. This feature is stabilizing the system at high tracking velocities, and also tend to minimize the tracking error.

Acknowledgements

I would like to thank the whole IFA, Institut für Automatik at ETH, for the nice and friendly atmosphere and for providing me with such an interesting project. In particular I would like to thank Alberto Bemporad for his outstanding support on the MPC algorithm and in general MPC theory. I really much appreciated his positive and spontaneous way of supporting. I would also like to thank Francesco Borrelli and Domenico Mignone who have assisted me greatly during the project. Mato Baotic have been very helpful in issues regarding C-code, providing me with information on structure of S-functions and helped with critical debugging problems.

I would also like to thank my professors, Sigurd Skogestad and Manfred Morari for making it possible for me to do my diploma thesis at ETH. It has indeed been a great stay here at ETH, and in overall a very nice outcome for me.

Thank you!

List of symbols

p	Prediction horizon	no unit
N_u	Control horizon	no unit
A	State matrix	no unit
B_u	State matrix, w.r.t. manipulated variables	no unit
B_v	State matrix, w.r.t. measured disturbance	no unit
B_d	State matrix, w.r.t. unmeasured disturbance	no unit
C	State matrix	no unit
D_v	State matrix, w.r.t. measured disturbance	no unit
D_d	State matrix, w.r.t. unmeasured disturbance	no unit
D_{vm}	State matrix, w.r.t. measured disturbance and measured outputs	no unit
$\hat{y}$	Output estimate	no unit
$\hat{x}^-$	Priori state estimate	no unit
$\hat{x}$	State estimate	no unit
L	Linear filter gain	no unit
Δu	Input increments	no unit
ϵ	Slack variable	no unit
ρ_ϵ	Weighting for slack variable, by default $\rho_\epsilon = 10^5 max_{i=0}^{p-1}\{w_i^u, w_i^{\Delta u}, w_{i+1}^y\}$	no unit
r	Reference	no unit
w^u	Input weight	no unit
$w^{\Delta u}$	Input increment weight	no unit
w^y	Output weight	no unit
G	Plant model	no unit
w_k	Noise	no unit
$\bar{x_k}$	Expected mean value of state	no unit
$E\{zz^T\}$	Mean-square value	no unit
P_{x_k}	State covariance	no unit
$P_{x_k y_k}$	Cross covariance between state and output	no unit
R	Covariance for measurement noise	no unit
Q	Covariance for process noise	no unit
$\tilde{x}_k$	Estimation error	no unit

Bibliography

[1] Bemporad, A.,Boricelli, F. and Morari,M. 2001,*Handout in MPC-course - Optimization*, ETH - Institut für Automatik

[2] Bemporad A., Bozinis,N.A., Dua,V., Morari,M. and Pistikopoulos,E.N. 2000. Model Predictive Control: A multi-parametric programming approach, *European Symposium on Computer Aided Process Engineering-10, Florence, Italy*, 301-306

[3] Bemporad, A.,Morari,M. and Ricker,N.L., 2000.*The MPC Simulink Library - User Guide*, Version 1, USA:MathWorks Inc.

[4] Bemporad,A. & Morari,M., 1999. Robust model predictive control: a survey, Lecture Notes in Control and Information Sciences, vol. 245, Springer-Verlag, 207-226

[5] Forster, P. & Weber, M., 1999/00. *Regulation anf trajectory tracking of a ball on a plate.*, Semester thesis, ETH Zřich

[6] Hermann,O., 1996/97. *Regelung eines "Ball & Plate" Systems.*, Diploma thesis, ETH Zřich

[7] Horton, I., 1995. *Instant C Programming*, Chicago USA: Wrox Press Ltd.

[8] Kalman, R.E., 1960. A new approach to Linear Filtering and Prediction Problems, *Trans. ASME J. Basic Eng.*, 82, 34-35

[9] Kalman, R.E. and Bucy, R.S., 1961. New results in Linear filtering and prediction theory, *Trans. ASME J. Basic Eng.*, 83, 95-108

[10] Lewis, Frank L., 1992. *Applied optimal control & estimation*, New Jersey:Prentice Hall

[11] Levine, William S., 1996. *The Control Handbook*, New York: CRC Press Inc.

[12] Marlin, Thomas E., 1995. *PROCESS CONTROL. Designing processes and control systems for dynamic performance*, New York: McGraw-Hill Inc.

[13] Morari, M., Lee, Jay H. & Garcia, Carlos E. 2000, *Model Predictive Control*, under printing at present

[14] Morari, M. & Lee Jay, H., 1999. Model Predicitive control: past, present and futur. *Computers and Chemical Engineering*,23, 667-682.

[15] Moschibroda, S. & Muntwyler, U., 2000. *Regulation and trajectory tracking of a "Ball & Plate" using MPC*, Semester thesis, ETH Zřich

[16] Ogata, K., 1987. *Discrete time control systems*, New Jersey:Prentice Hall

[17] Ritchie, D.M. (1996). Bell Labs/Lucent Technologies [online]. Available from: http://cm.bell-labs.com/cm/cs/who/dmr/chist.html [1.june 2001]

[18] The MathWorks Inc., 2000. *Simulink : Writing S-Functions*, Version 4, USA:The MathWorks Inc.

[19] The MathWorks Inc., 1997. *Using Simulink*, Version 2, USA:The MathWorks Inc.

[20] The MathWorks Inc., 2000. *xPC Target - user guide*, Version 1.1, USA:The MathWorks Inc.

Appendix A

MEX-function

```
/*************************************************************************
Module:  MPC_controller Notices: Written 2001 Thomas Haugan
*************************************************************************/

/* Definition of Simstruct and its associated macros. Includes
mex.h and matrix.h */
 #include "mex.h"
 #include "matrix.h"
 #include <stdlib.h>

/* Gateway function starts here */
 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{

///////////////////////////////////////////////////////////////////////////
//***************   CONSTRUCTING TEMPORARY MXARRAYS    *******************//
mxArray *y_tmp,*r_tmp,*v_tmp,*xk1_tmp,*xk2_tmp,*uk1_tmp; mxArray
*yest_tmp,*accum_tmp,*xk_tmp,*zopt_tmp,*uk_tmp;

///////////////////////////////////////////////////////////////////////////
//********************           LOCALS           ***********************//
 real_T *y,*r,*v,*xk1,*xk2,*uk1,*yest,*accum,*xk,*zopt,*uk,t;
 real_T Ts,delay,*epsslack,*feasible;
 int size_yrd,size_lastx,size_lastu,nym,ny,nv,rows_Cm,size_A,degrees;
 int_T i, j, k;
 double *u_out,*xk_out,*dnym;
 real_T tr,tr_dec,tr_round;
 int tr_int,cols_A,cols_L,cols_Cm,rows_L,rows_A;

///////////////////////////////////////////////////////////////////////////
```

```
// ADDITIONAL PARAMETERS, ARRAY OF POINTERS IN FUNCTION CALL OF
// MPC2
    mxArray *plhs_MPC2[3];
    const mxArray *prhs_MPC2[22];
////////////////////////////////////////////////////////////////////////////
/******   ACCESS PARAMETERS USED IN MEMORY ALLOCATION **********/
    size_yrd = mxGetNumberOfElements(prhs[2]);
    nym = (int)mxGetPr(prhs[24])[0];
    ny  = (int)mxGetPr(prhs[25])[0];
    nv  = (int)mxGetPr(prhs[26])[0];
    size_lastx = mxGetNumberOfElements(prhs[21]);
    size_lastu = mxGetNumberOfElements(prhs[22]);
    rows_Cm = mxGetM(prhs[8]);
    cols_Cm = mxGetN(prhs[8]);
    rows_L = mxGetM(prhs[23]);
    cols_L = mxGetN(prhs[23]);
    size_A = mxGetM(prhs[4]);
    rows_A = mxGetM(prhs[4]);
    cols_A = mxGetN(prhs[4]);
    degrees = (int)mxGetPr(prhs[19])[0];

///////////////////////////////////////////////////////////////////////////
/***********   ALLOCATE DYNAMIC MEMORY FOR VARIABLES ***********/
    y_tmp       = mxCreateDoubleMatrix(nym,1,mxREAL);
    r_tmp       = mxCreateDoubleMatrix(ny,1,mxREAL);
    v_tmp       = mxCreateDoubleMatrix(nv,1,mxREAL);
    xk1_tmp     = mxCreateDoubleMatrix(size_lastx,1,mxREAL);
    xk2_tmp     = mxCreateDoubleMatrix(size_lastx,1,mxREAL);
    uk1_tmp     = mxCreateDoubleMatrix(size_lastu,1,mxREAL);
    yest_tmp    = mxCreateDoubleMatrix(rows_Cm,1,mxREAL);
    accum_tmp   = mxCreateDoubleMatrix(size_A,1,mxREAL);
    xk_tmp      = mxCreateDoubleMatrix(size_lastx,1,mxREAL);
    zopt_tmp    = mxCreateDoubleMatrix(degrees,1,mxREAL);
    uk_tmp      = mxCreateDoubleMatrix(size_lastu,1,mxREAL);

///////////////////////////////////////////////////////////////////////////
/************   ALLOCATE MEMORY FOR RETURN ARGUMENT *************/
    plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);
    plhs[1] = mxCreateDoubleMatrix(size_lastx,1, mxREAL);

///////////////////////////////////////////////////////////////////////////
/*******************   ASSIGN POINTERS TO EACH OUTPUT
```

```
*********************/
    u_out = mxGetPr(plhs[0]);
    xk_out = mxGetPr(plhs[1]);

///////////////////////////////////////////////////////////////////////////
/**********   ASSIGN POINTERS TO EACH ALLOCATED MEMORY **********/
    y   = mxGetPr(y_tmp);
    r   = mxGetPr(r_tmp);
    v   = mxGetPr(v_tmp);
    xk1 = mxGetPr(xk1_tmp);
    xk2 = mxGetPr(xk2_tmp);
    uk1 = mxGetPr(uk1_tmp);
    yest = mxGetPr(yest_tmp);
    accum = mxGetPr(accum_tmp);
    xk  = mxGetPr(xk_tmp);
    zopt = mxGetPr(zopt_tmp);
    uk  = mxGetPr(uk_tmp);

///////////////////////////////////////////////////////////////////////////
/***************    CHECK IF SAMPLE TIME IS UP *****************/
    t   = mxGetPr(prhs[0])[0];
    Ts  = mxGetPr(prhs[18])[0];
    delay = mxGetPr(prhs[17])[0];
/******** Rounding of tr ***********/
    tr=((t-delay)/(Ts));
    tr_int = tr;            // Storing integer part of tr in tr_int
    tr_dec = tr - tr_int;   // Storing decimal part of tr in tr_dec
    if (tr_dec >= 0.5)
        tr_dec = 1;
    else
        tr_dec = 0;
    tr_round = tr_int + tr_dec;
/**********************************/

    /*Check if t is integer multiple of Ts, and if one step has elapsed */
    if (((tr_round - (t-delay)/(Ts)) < 0.00000001) &&
         ((tr_round - (t-delay)/(Ts)) > -0.00000001)) {
        if (tr_round > mxGetPr(prhs[20])[0]) {

            if (size_yrd < nym){
                mexWarnMsgTxt("Wrong number of measured outputs");
            }
```

```
/* Assigning values to y, r and v */
if (nym > 0){
    for (i=0; i < nym; i++){
        y[i]= mxGetPr(prhs[2])[i];
    }
}

if (ny > 0){
    for (i=nym; i <= nym+ny; i++){
        r[i-nym]= mxGetPr(prhs[2])[i];
    }
}

if (nv > 0){
    for (i=nym+ny; i <= nym+ny+nv; i++){
        v[i-nym-ny]= mxGetPr(prhs[2])[i];
    }
}


/* Storing the last states in xk1*/
for (i=0; i < size_lastx; i++){
    xk1[i]= mxGetPr(prhs[21])[i];
}

/* Storing the last control moves in uk1*/
for (i=0; i < size_lastu; i++){
    uk1[i]= mxGetPr(prhs[22])[i];
}


/*Measurement update. (yest=Cm*xk1+Dvm*v)*/
for (i=0; i < rows_Cm; i++) {
    accum[0] = 0.0;

    for (j=0; j < cols_Cm; j++) {
        accum[0] += mxGetPr(prhs[8])[i+j*rows_Cm] * xk1[j];
    }

    for (k=0; k < nv; k++) {
```

```
            accum[0] += mxGetPr(prhs[10])[k] * v[k];
        }

        yest[i] = accum[0];
    }



    /*xk2=xk1+L(y-yest)*/
    for (i=0; i < rows_L; i++) {
        accum[0] = 0.0;
        accum[0] += xk1[i];
            for (j=0; j < cols_L; j++) {
                accum[0] += mxGetPr(prhs[23])[(i+j*(rows_L))]*y[j];//nym
                accum[0] += -mxGetPr(prhs[23])[(i+j*(rows_L))]*yest[j];
            }
        xk2[i] = accum[0];
    }

//////////////////////////////////////////////////////////////////////////
/********   CALCULATION OF CONTROL LAW. CALL TO MPC2_6 **********/

    prhs_MPC2[0]    = prhs[28];//MuKduINV;
    prhs_MPC2[1]    = prhs[29];//KduINV;[i]
    prhs_MPC2[2]    = prhs[30];//Kx;
    prhs_MPC2[3]    = prhs[31];//Ku1;
    prhs_MPC2[4]    = prhs[32];//Kut;
    prhs_MPC2[5]    = prhs[33];//Kr;
    prhs_MPC2[6]    = prhs[34];//zmin;
    prhs_MPC2[7]    = prhs[35];//rhsc0;
    prhs_MPC2[8]    = prhs[36];//Mlim;
    prhs_MPC2[9]    = prhs[37];//Mx;
    prhs_MPC2[10]   = prhs[38];//Mu1;
    prhs_MPC2[11]   = prhs[39];//Mvv;
    prhs_MPC2[12]   = prhs[40];//rhsa0;
    prhs_MPC2[13]   = prhs[41];//TAB;
    prhs_MPC2[14]   = xk2_tmp;
    prhs_MPC2[15]   = uk1_tmp;
    prhs_MPC2[16]   = prhs[42];//utarget;
    prhs_MPC2[17]   = r_tmp;
    prhs_MPC2[18]   = prhs[43];//vKv;
    prhs_MPC2[19]   = prhs[19];//degrees;
    prhs_MPC2[20]   = prhs[44];//isunconstr;
```

```
            prhs_MPC2[21]   = prhs[45];//TYPE;*/


            /* Function call of MPC2 */
            MPC2(3, plhs_MPC2, 22, prhs_MPC2);




            for (i=0; i < degrees; i++){
                zopt[i] = mxGetPr(plhs_MPC2[0])[i];
            }
////////////////////////////////////////////////////////////////////////////


            for (i=0; i < size_lastu; i++) {
                uk[i] = uk1[i] + zopt[i];   //The first delta move
                                            //have one degree of freedom
            }

            /*Time (or state) update. Kalman*/
            /* xk = A*xk2+Bu*uk+Bv*vk; */

            for (i=0; i < size_lastx; i++) {
                accum[0] = 0.0;

                for (j=0; j < cols_A; j++) {
                    accum[0] += mxGetPr(prhs[4])[i+j*rows_A]*xk2[j];
                }

                accum[0] += mxGetPr(prhs[11])[i]*uk[0];

                for (k=0; k < nv; k++) {
                    accum[0] += mxGetPr(prhs[12])[k]*v[k];
                }
                xk[i] = accum[0];
            }

/////////////////////////////////////////////////////////////////////////////
/**************   ASSIGN POINTERS TO EACH OUTPUT ****************/
            for (i=0; i < size_lastu; i++) {
                u_out[i] = uk[i];
            }
```

```
            for (i=0; i < size_lastx; i++) {
                xk_out[i] = xk[i];
            }
        }

        else {
            for (i=0; i < size_lastu; i++) {
                u_out[i] = uk1[i];
                xk_out[i] = uk1[i];
            }
        }
    }
}


/****************************************************************/
/********************** End Of File *****************************/
/****************************************************************/
```

Appendix B

C-function (MPC2)

```
/***********************************************************************
 Module:  MPC2
 Notices: Written 2001 Thomas Haugan
***********************************************************************/
////////////////////////////////////////////////////////////////////////
//                 This function returns the value zopt               //
////////////////////////////////////////////////////////////////////////

/* Includes API-functions */ #include "mex.h" #include "matrix.h"
#include <stdlib.h>


/* Declaring danzgmp as external algorithm */ extern void
dantzgmp_1(int nlhs,mxArray *plhs[],int nrhs,const mxArray
*prhs[]);

//////////////////////////////////////////////////////////////////////////
//*    Compute the optimal input sequence by solving a QP problem        *//
//////////////////////////////////////////////////////////////////////////
void MPC2(int nlhs,mxArray *plhs_MPC2[],int nrhs,const mxArray
*prhs_MPC2[]) {

//////////////////////////////////////////////////////////////////////////
//*****************   CONSTRUCTING TEMPORARY MXARRAYS    *****************//
 mxArray *zopt_tmp,*accum_tmp,*accum2_tmp,*rhsc_tmp,*rhsa_tmp,*basisi_tmp;
 mxArray *ili_tmp,*basis_tmp,*ib_tmp,*il_tmp,*iter_tmp,*epsslack_tmp;
 mxArray *feasible_tmp,*ibi_tmp*zopt1_tmp;

//////////////////////////////////////////////////////////////////////////
//*********************          LOCALS          **********************//
```

```
 real_T *zopt,*accum,*accum2,*rhsc,*rhsa,*basisi,*ibi,*ili,*basis,*ib;
 real_T *epsslack,*zopt1,*feasible,*il,*iter;
 int size_Kx,rows_Kx,cols_Ku1,rows_Kut,rows_utarget,rows_Kr,cols_Kr;
 int cols_KduINV,rows_rhsc0,cols_Mx,rows_Mu1,rows_rhsa0,rows_xk,nc,size_ibi;
 int cols_Kut,rows_KduINV,rows_MuKduINV,cols_MuKduINV,cols_Kx,cols_vKv;
 int_T i,j,k,buflen,status;

/////////////////////////////////////////////////////////////////////////////
// Additional parameters. Passing in-parameters to dantzgmp and
// get result
 mxArray *lhs_d[4]; const mxArray *rhs_d[4];

/////////////////////////////////////////////////////////////////////////////
/********   ACCESS PARAMETERS USED IN MEMORY ALLOCATION *********/
    size_Kx     = mxGetNumberOfElements(prhs_MPC2[2]);
    rows_Kx     = mxGetM(prhs_MPC2[2]);
    cols_Kx     = mxGetN(prhs_MPC2[2]);
    cols_Ku1    = mxGetN(prhs_MPC2[3]);
    rows_Kut    = mxGetM(prhs_MPC2[4]);
    cols_Kut    = mxGetN(prhs_MPC2[4]);
    rows_utarget = mxGetM(prhs_MPC2[16]);
    cols_Kr     = mxGetN(prhs_MPC2[5]);
    rows_Kr     = mxGetM(prhs_MPC2[5]);
    cols_vKv    = mxGetN(prhs_MPC2[18]);
    cols_KduINV = mxGetN(prhs_MPC2[1]);
    rows_rhsc0  = mxGetM(prhs_MPC2[7]);
    cols_Mx     = mxGetN(prhs_MPC2[9]);
    rows_Mu1    = mxGetM(prhs_MPC2[10]);
    rows_rhsa0  = mxGetM(prhs_MPC2[12]);
    rows_xk     = mxGetM(prhs_MPC2[14]);
    rows_KduINV = mxGetM(prhs_MPC2[1]);
    rows_MuKduINV   = mxGetM(prhs_MPC2[0]);
    cols_MuKduINV   = mxGetN(prhs_MPC2[0]);
////////////////////////////////////////////////////////////////////////////
/****************   ALLOCATE DYNAMIC MEMORY FOR VARIABLES
*****************/
    zopt_tmp    = mxCreateDoubleMatrix(rows_rhsa0,1,mxREAL);
    zopt1_tmp   = mxCreateDoubleMatrix(rows_rhsa0,1,mxREAL);
    accum_tmp   = mxCreateDoubleMatrix(1,1,mxREAL);
    accum2_tmp  = mxCreateDoubleMatrix(1,1,mxREAL);
    rhsc_tmp    = mxCreateDoubleMatrix(rows_rhsc0,1,mxREAL);
    rhsa_tmp    = mxCreateDoubleMatrix(rows_xk,1,mxREAL);
```

```
    basisi_tmp  = mxCreateDoubleMatrix(rows_rhsa0+rows_rhsc0,1,mxREAL);
    epsslack_tmp    = mxCreateDoubleMatrix(1,1,mxREAL);
    feasible_tmp    = mxCreateDoubleMatrix(1,1,mxREAL);
////////////////////////////////////////////////////////////////////////////
/**********   ASSIGN POINTERS TO EACH ALLOCATED MEMORY **********/
    zopt    = mxGetPr(zopt_tmp);
    zopt1   = mxGetPr(zopt1_tmp);
    accum   = mxGetPr(accum_tmp);
    accum2  = mxGetPr(accum2_tmp);
    rhsc    = mxGetPr(rhsc_tmp);
    rhsa    = mxGetPr(rhsa_tmp);
    basisi  = mxGetPr(basisi_tmp);
    epsslack    = mxGetPr(epsslack_tmp);
    feasible    = mxGetPr(feasible_tmp);
////////////////////////////////////////////////////////////////////////////


    if (mxGetPr(prhs_MPC2[20])[0]==1) {
    // Unconstrained MPC :
        /* Matrix multiplication and adding to accum */
        /* zopt = -KduINV*(Kx'*xk+Ku1'*uk1+Kut'*utarget+Kr'*r+vKv')*/
        for (i=0; i < cols_Ku1; i++) {
            accum[0] = 0.0;
            if (i == 0) {
                for (j=0; j < rows_Kx; j++) {
                    accum[0] += mxGetPr(prhs_MPC2[2])[j+cols_Kx]
                                *mxGetPr(prhs_MPC2[14])[j]; //Kx'*xk
                }
            }
            else {
                for (j=0; j < rows_Kx; j++) {
                    accum[0] += mxGetPr(prhs_MPC2[2])[j]
                                *mxGetPr(prhs_MPC2[14])[j]; //Kx'*xk
                }
            }

            for (j=0; j < cols_Ku1; j++) {
                accum[0] += mxGetPr(prhs_MPC2[3])[j]
                            *mxGetPr(prhs_MPC2[15])[0]; //Ku1*uk1
            }

            for (j=0; j < cols_Kut; j++) {
```

```
            accum[0] += mxGetPr(prhs_MPC2[4])[i+j*rows_Kut]
                        *mxGetPr(prhs_MPC2[16])[j];//Kut*utarget
        }
        for (j=0; j < rows_Kr; j++) {
            for (k=0; k < cols_Kr; k++) {
                accum[0] += mxGetPr(prhs_MPC2[5])[j+k*rows_Kr]
                            *mxGetPr(prhs_MPC2[17])[k];//Kr*r
            }
        }

        accum[0] += mxGetPr(prhs_MPC2[18])[i];//vKv

        for (j=0; j < cols_Ku1; j++) {
            accum[0] = -mxGetPr(prhs_MPC2[1])[j]*accum[0];//-(KduINV*accum)
        }

        zopt[i] = accum[0];
    }
    epsslack[0] = 0;
    feasible[0] = 1;

}
else {
// Constrained MPC:
    /* rhsc = rhsc0+Mlim+Mx*xk+Mu1*uk1+Mvv; */
    for (i = 0; i < rows_rhsc0; i++) {
        accum[0] = 0.0;

        accum[0] += mxGetPr(prhs_MPC2[7])[i]
                    +mxGetPr(prhs_MPC2[8])[i]; //*rhsc0+Mlim;

        for (j = 0; j < cols_Mx; j++) {
            accum[0] += mxGetPr(prhs_MPC2[9])[i+j*rows_rhsc0]
                        *mxGetPr(prhs_MPC2[14])[j];//Mx*xk;
        }

        accum[0] += mxGetPr(prhs_MPC2[10])[i]
                    *mxGetPr(prhs_MPC2[15])[0];// Mu1*uk1;

        accum[0] += mxGetPr(prhs_MPC2[11])[i];// Mvv[i];

        rhsc[i] = accum[0];
```

```
}

/* rhsa = rhsa0-[xk'*Kx+r'*Kr+uk1'*Ku1+vKv+utarget'*Kut,0]';*/
for (i = 0; i < rows_rhsa0-1; i++) {
    accum[0] = 0.0;

    for (j = 0; j < rows_xk; j++) {
        accum[0] += mxGetPr(prhs_MPC2[14])[j]
                    *mxGetPr(prhs_MPC2[2])[j+i*rows_Kx];//xk[i]*Kx[i];
    }

    for (j=0; j < rows_Kr; j=j+2) {
        for (k=0; k < cols_Kr; k++) {
            accum[0] += mxGetPr(prhs_MPC2[17])[k]
                        *mxGetPr(prhs_MPC2[5])[j+k+i*rows_Kr];//r*Kr
        }
    }

    accum[0] += mxGetPr(prhs_MPC2[3])[i]
                *mxGetPr(prhs_MPC2[15])[0];//uk1[i]*Ku1[i];

    for (j=0; j < rows_utarget; j++) {
        accum[0] += mxGetPr(prhs_MPC2[16])[j]
                    *mxGetPr(prhs_MPC2[4])[j+i*rows_Kut];//utarget*Kut;
    }

    accum[0] += mxGetPr(prhs_MPC2[18])[i];//vKv;

    accum2[0] = mxGetPr(prhs_MPC2[12])[i]-accum[0];//rhsa0-(accum);

    rhsa[i] = accum2[0];
}
    rhsa[rows_rhsa0] = mxGetPr(prhs_MPC2[12])[rows_rhsa0];//rhsa0;



/* basisi = [KduINV*rhsa;rhsc-MuKduINV*rhsa]; */
for (i = 0; i < rows_KduINV; i++) {
    accum[0] = 0.0;

    for (j = 0; j < cols_KduINV; j++) {
        accum[0] += mxGetPr(prhs_MPC2[1])[i+j*rows_KduINV]
```

```
                        *rhsa[j];//KduINV*rhsa;
            }
            basisi[i] = accum[0];
        }

        for (i = rows_rhsa0; i < rows_rhsc0+rows_rhsa0; i++) {
            accum2[0] = 0.0;
            accum2[0] += rhsc[i-rows_rhsa0];

            for (j = 0; j < cols_MuKduINV; j++) {
                accum2[0] -= mxGetPr(prhs_MPC2[0])[i-rows_rhsa0+j*rows_MuKduINV]
                        *rhsa[j];//MuKduINV*rhsa;
            }
            basisi[i] = accum2[0];
        }

        nc = rows_rhsc0;

        /* size_ibi = degrees + 1+ nc */
        size_ibi = mxGetPr(prhs_MPC2[19])[0] + 1 + nc;

        /* Allocates memory, initialising each element to zero */
        ibi_tmp = mxCreateDoubleMatrix(size_ibi,1,mxREAL);
        ili_tmp = mxCreateDoubleMatrix(size_ibi,1,mxREAL);

        ibi = mxGetPr(ibi_tmp);
        ili = mxGetPr(ili_tmp);

        for (i = 0; i < size_ibi; i++) {
            ibi[i] = -(1+i);
        }
        for (i = 0; i < size_ibi; i++) {
        ili[i] = -ibi[i];
        }

////////////////////////////////////////////////////////////////////////////
// OPTIMISATION WITH RESPECT TO COST-FUNCTION. CALL TO DANTZGMP, A
// QP-SOLVER

        rhs_d[0]    = prhs_MPC2[13];//TAB;
        rhs_d[1]    = basisi_tmp;
        rhs_d[2]    = ibi_tmp;
```

```
        rhs_d[3]    = ili_tmp;

        dantzgmp_1(4, lhs_d, 4, rhs_d);

        basis_tmp   = lhs_d[0];
        ib_tmp      = lhs_d[1];
        il_tmp      = lhs_d[2];
        iter_tmp    = lhs_d[3];
////////////////////////////////////////////////////////////////////////////
        /* Assign pointers to allocated memory */
        basis   = mxGetPr(basis_tmp);
        ib      = mxGetPr(ib_tmp);
        il      = mxGetPr(il_tmp);
        iter    = mxGetPr(iter_tmp);

        if (*iter >= 0) {
            feasible[0] = 1;
            for (i=0;i<mxGetPr(prhs_MPC2[19])[0]+1;i++) {   // degrees + 1
                if (il[i] <= 0)  {
                    zopt1[i] = mxGetPr(prhs_MPC2[6])[i];//zmin[i];
                }
                else  {
                    zopt1[i] = basis[i]+mxGetPr(prhs_MPC2[6])[i];//basis+zmin;
                }
            }
        }
        else {
            mexWarnMsgTxt("Warning: Constraints are overly stringent");
        }
        for (i=mxGetPr(prhs_MPC2[19])[0]+1;i<mxGetPr(prhs_MPC2[19])[0]+2;i++) {
            epsslack[0] = zopt1[i];//degrees+1
        }
        for (i=0;i<mxGetPr(prhs_MPC2[19])[0];i++) {
            zopt[i] = zopt1[i];
        }

    }

////////////////////////////////////////////////////////////////////////////
/***************   ASSIGN POINTERS TO EACH OUTPUT ***************/

    plhs_MPC2[0]=zopt_tmp;
```

```
    plhs_MPC2[1]=epsslack_tmp;
    plhs_MPC2[2]=feasible_tmp;
}

///////////////////////////////////////////////////////////////////////////
/***************     DESTROY ALLOCATED MEMORY ******************/
//mxDestroyArray(accum_tmp);


/****************************************************************/
/********************** End Of File *****************************/
/****************************************************************/
```

Appendix C

DANTZIG-routine

```
// Gateway to DANTZGMP c-mex routine.
// N. L. Ricker, 12/98

// MATLAB calling format:

// [bas,ib,il,iter,tab]=dantzgmp(tabi,basi,ibi,ili)

// Inputs:
// tabi : initial tableau
// basi : initial basis
// ibi : initial setting of ib
// ili : initial setting of il

// Outputs:
// bas : final basis vector
// ib : index vector for the variables -- see examples
// il : index vector for the lagrange multipliers -- see examples
// iter : iteration counter
// tab : final tableau

#include "math.h"

#include "mex.h" #include "dantzgmp.h"

void dantzgmp_1(int nlhs,mxArray *plhs[],int nrhs,const mxArray
*prhs[]) {
 double *tabi, *basi, *ibi, *ili;
 int M, N, rows, cols,iret;
 int nuc=0; int i, j;
 int MN = 0;
```

```
 long len;
 mxArray *ptrs[5];
 double *bas, *ib, *il, *iter, *tab;
 integer *ibint, *ilint;
 integer buflen;


// Verify correct number of input and output arguments.
if (nrhs != 4)
 mexErrMsgTxt("You must supply 4 input variables.\n");
 tabi = mxGetPr(prhs[0]);
 basi = mxGetPr(prhs[1]);
 ibi = mxGetPr(prhs[2]);
 ili = mxGetPr(prhs[3]);
 if (nlhs < 3)
 mexErrMsgTxt("You must supply at least 3 output variables.\n");

// Error checking on inputs
// Checking TABI
 M = mxGetM(prhs[0]);
 N = mxGetN(prhs[0]);
 if (M <= 0 || N <= 0)
 mexErrMsgTxt("TABI is empty.\n");
// Checking BASI
 rows = mxGetM(prhs[1]);
 cols = mxGetN(prhs[1]);
 len = max(rows,cols);
 if (min(rows,cols) != 1 || len != M)
 mexErrMsgTxt("BASI must be a vector, length = number of rows in TABI.\n");
// Checking IBI
 rows = mxGetM(prhs[2]);
 cols = mxGetN(prhs[2]);
 len = max(rows,cols);
 if (min(rows,cols) != 1 || len != M)
 mexErrMsgTxt("IBI must be a vector, length = number of rows in TABI.\n");
// Checking ILI
 rows = mxGetM(prhs[3]);
 cols = mxGetN(prhs[3]);
 len = max(rows,cols);
 if (min(rows,cols) != 1 || len != M)
 mexErrMsgTxt("ILI must be a vector, length = number of rows in TABI.\n");
```

```
// Allocate space for output variables and define corresponding C pointers
 ptrs[0] = mxCreateDoubleMatrix(M, 1, mxREAL);
 bas = mxGetPr(ptrs[0]);
 ptrs[1] = mxCreateDoubleMatrix(M, 1, mxREAL);
 ib = mxGetPr(ptrs[1]);
 ptrs[2] = mxCreateDoubleMatrix(M, 1, mxREAL);
 il = mxGetPr(ptrs[2]);
 ptrs[3] = mxCreateDoubleMatrix(1, 1, mxREAL);
 iter = mxGetPr(ptrs[3]);
 ptrs[4] = mxCreateDoubleMatrix(M, N, mxREAL);
 tab = mxGetPr(ptrs[4]);
// We have to convert ib and il from double to integer and vice-versa.
// Allocate arrays for storing the integer versions.
 buflen = M * sizeof(*ibint);
 ibint = (integer *) mxMalloc(buflen);
// Pointer to integer version of ib
 ilint = (integer *) mxMalloc(buflen);
// Pointer to integer version of il

// Initialization

for (i=0; i<M; i++) {
   bas[i] = basi[i];
   ibint[i] = (integer) ibi[i];
   ilint[i] = (integer) ili[i];
}

for (j=0; j<N; j++) {
  for (i=0; i<M; i++) {
    tab[MN] = tabi[MN];
    MN++;
  }
}

// Call DANTZG for the calculations

iret = dantzg(tab, &N, &N, &nuc, bas, ibint, ilint);
// Store number of iterations.
*iter = (double) iret;

// Return results to MATLAB. First convert integer versions
```

```
// of ib and il back to real, then set pointers to outputs.
 for (i=0; i<M; i++) {
  ib[i] = (double) ibint[i];
  il[i] = (double) ilint[i];
 }

 for (i=0; i<nlhs; i++) {
   plhs[i] = ptrs[i];
 }
}

/* Subroutine */ int dantzg(doublereal *a, int *ndim, int *n, int
*nuc, doublereal *bv, integer *ib, integer *il) {

 /* System generated locals */
 integer a_dim1, a_offset, i__1;

/* Local variables */
 integer ichk, iter;
 doublereal rmin, test;
 integer iout, i, ichki, ic, ir, nt, istand, irtest;
 extern /* Subroutine */ int trsimp_(doublereal *, int *, integer *, int *,
 doublereal *, integer *, integer *);
 integer iad;
 doublereal val, rat;
 int iret=-1;
/* ******************************************* */

/* VERSION MODIFIED 1/88 BY NL RICKER */ /* Modified 12/98 by NL
Ricker for use as MATLAB MEX file */

/* ******************************************* */

/* DANTZIG QUADRATIC PROGRAMMING ALGORITHM. */

/* N.L. RICKER 6/83 */

/* ASSUMES THAT THE INPUT VARIABLES REPRESENT A FEASIBLE INITIAL
*/ /* BASIS SET. */

/* N NUMBER OF CONSTRAINED VARIABLES (INCLUDING SLACK
VARIABLES).*/
```

```
/* NUC NUMBER OF UNCONSTRAINED VARIABLES, IF ANY */

/* BV VECTOR OF VALUES OF THE BASIS VARIABLES. THE LAST NUC */ /*
ELEMENTS WILL ALWAYS BE KEPT IN THE BASIS AND WILL NOT */ /* BE
CHECKED FOR FEASIBILITY. */

/* IB INDEX VECTOR, N ELEMENTS CORRESPONDING TO THE N VARIABLES.
*/ /* IF IB(I) IS POSITIVE, THE ITH */ /* VARIABLE IS BASIC AND
BV(IB(I)) IS ITS CURRENT VALUE. */ /* IF IB(I) IS NEGATIVE, THE
ITH VARIABLE IS NON-BASIC */ /* AND -IB(I) IS ITS COLUMN NUMBER IN
THE TABLEAU. */

/* IL VECTOR DEFINED AS FOR IB BUT FOR THE N LAGRANGE
MULTIPLIERS.*/

/* A THE TABLEAU -- SEE TRSIMP DESCRIPTION. */

/* IRET IF SUCCESSFUL, CONTAINS NUMBER OF ITERATIONS REQUIRED. */
/* OTHER POSSIBLE VALUES ARE: */ /* - I NON-FEASIBLE BV(I) */ /*
-2N NO WAY TO ADD A VARIABLE TO BASIS */ /* -3N NO WAY TO DELETE A
VARIABLE FROM BASIS */ /* NOTE: THE LAST TWO SHOULD NOT OCCUR AND
INDICATE BAD INPUT*/ /* OR A BUG IN THE PROGRAM. */



/* CHECK FEASIBILITY OF THE INITIAL BASIS. */

/* Parameter adjustments */ --il; --ib; --bv; a_dim1 = *ndim;
a_offset = a_dim1 + 1; a -= a_offset;

/* Function Body */ iter = 1; nt = *n + *nuc; i__1 = *n; for (i =
1; i <= i__1; ++i) { if (ib[i] < 0 || bv[ib[i]] >= 0.f) { goto
L50; } iret = -i; goto L900; L50: ; } istand = 0; L100:

/* SEE IF WE ARE AT THE SOLUTION. */

if (istand != 0) { goto L120; } val = 0.f; iret = iter;

i__1 = *n; for (i = 1; i <= i__1; ++i) { if (il[i] < 0) { goto
L110; }
```

```
/* PICK OUT LARGEST NEGATIVE LAGRANGE MULTIPLIER. */

test = bv[il[i]]; if (test >= val) { goto L110; } val = test; iad
= i; ichk = il[i]; ichki = i + *n; L110: ; }

/* IF ALL LAGRANGE MULTIPLIERS WERE NON-NEGATIVE, ALL DONE. */ /*
ELSE, SKIP TO MODIFICATION OF BASIS */

if (val >= 0.f) { iret=iter; goto L900; } ic = -ib[iad]; goto
L130;

/* PREVIOUS BASIS WAS NON-STANDARD. MUST MOVE LAGRANGE */ /*
MULTIPLIER ISTAND INTO BASIS. */

L120: iad = istand; ic = -il[istand - *n];

/* CHECK TO SEE WHAT VARIABLE SHOULD BE REMOVED FROM BASIS. */

L130: ir = 0;

/* FIND SMALLEST POSITIVE RATIO OF ELIGIBLE BASIS VARIABLE TO */
/* POTENTIAL PIVOT ELEMENT. FIRST TYPE OF ELIGIBLE BASIS VARIABLE
*/ /* ARE THE REGULAR N VARIABLES AND SLACK VARIABLES IN THE
BASIS. */

i__1 = *n; for (i = 1; i <= i__1; ++i) { irtest = ib[i];

/* NO GOOD IF THIS VARIABLE ISN'T IN BASIS OR RESULTING PIVOT
WOULD */ /* BE ZERO. */

if (irtest < 0 || a[irtest + ic * a_dim1] == 0.f) { goto L150; }
rat = bv[irtest] / a[irtest + ic * a_dim1];

/* THE FOLLOWING IF STATEMENT WAS MODIFIED 7/88 BY NL RICKER */ /*
TO CORRECT A BUG IN CASES WHERE RAT=0. */

if (rat < 0.f || rat == 0.f && a[irtest + ic * a_dim1] < 0.f) {
goto L150; } if (ir == 0) { goto L140; } if (rat > rmin) { goto
L150; } L140: rmin = rat; ir = irtest; iout = i; L150: ; }

/* SECOND POSSIBLITY IS THE LAGRANGE MULTIPLIER OF THE VARIABLE
ADDED*/ /* TO THE MOST RECENT STANDARD BASIS. */
```

```
if (a[ichk + ic * a_dim1] == 0.f) { goto L170; } rat = bv[ichk] /
a[ichk + ic * a_dim1]; if (rat < 0.f) { goto L170; } if (ir == 0)
{ goto L160; } if (rat > rmin) { goto L170; } L160: ir = ichk;
iout = ichki;

L170: if (ir != 0) { goto L200; } iret = *n * -3; goto L900;

L200:

/* SET INDICES AND POINTERS */

if (iout > *n) { goto L220; } ib[iout] = -ic; goto L230; L220:
il[iout - *n] = -ic; L230: if (iad > *n) { goto L240; } ib[iad] =
ir; goto L250; L240: il[iad - *n] = ir; L250:

/* TRANSFORM THE TABLEAU */

trsimp_(&a[a_offset], ndim, &nt, n, &bv[1], &ir, &ic); ++iter;

/* WILL NEXT TABLEAU BE STANDARD? */

istand = 0; i__1 = *n; for (i = 1; i <= i__1; ++i) { /* L260: */
if (ib[i] > 0 && il[i] > 0) { goto L270; } } goto L280; L270:
istand = iout + *n; L280: goto L100;

L900: return iret; } /* dantzg_ */

/* Subroutine */ int trsimp_(doublereal *a, int *ndim, integer *m,
int *n, doublereal *bv, integer *ir, integer *ic) { /* System
generated locals */ integer a_dim1, a_offset, i__1, i__2;

/* Local variables */ integer i, j; doublereal ap;


/* TRANSFORM SIMPLEX TABLEAU. SWITCH ONE BASIS VARIABLE FOR ONE */
/* NON-BASIC VARIABLE. */

/* N.L. RICKER 6/83 */

/* A SIMPLEX TABLEAU. ACTUALLY DIMENSIONED FOR NDIM ROWS IN */ /*
THE CALLING PROGRAM. IN THIS PROCEDURE, ONLY THE A(M,N) */ /*
```

```
SPACE IS USED. */

/* NDIM ACTUAL ROW DIMENSION OF A IN THE CALLING PROGRAM */

/* M NUMBER OF ROWS IN THE TABLEAU */

/* N NUMBER OF COLUMNS IN THE TABLEAU */

/* BV VECTOR OF M BASIS VARIABLE VALUES */

/* IR ROW IN TABLEAU CORRESPONDING TO THE BASIC VARIABLE THAT */
/* IS TO BECOME NON-BASIC */

/* IC COLUMN IN TABLEAU CORRESPONDING TO THE NON-BASIC VARIABLE */
/* THAT IS TO BECOME BASIC. */



/* FIRST CALCULATE NEW VALUES FOR THE NON-PIVOT ELEMENTS. */

/* Parameter adjustments */ --bv; a_dim1 = *ndim; a_offset =
a_dim1 + 1; a -= a_offset;

/* Function Body */ i__1 = *m; for (i = 1; i <= i__1; ++i) { if (i
== *ir) { goto L110; } ap = a[i + *ic * a_dim1] / a[*ir + *ic *
a_dim1]; bv[i] -= bv[*ir] * ap; i__2 = *n; for (j = 1; j <= i__2;
++j) { if (j == *ic) { goto L100; } a[i + j * a_dim1] -= a[*ir + j
* a_dim1] * ap; L100: ; } L110: ; }

/* NOW TRANSFORM THE PIVOT ROW AND PIVOT COLUMN. */

ap = a[*ir + *ic * a_dim1]; i__1 = *m; for (i = 1; i <= i__1; ++i)
{ a[i + *ic * a_dim1] = -a[i + *ic * a_dim1] / ap; /* L120: */ }

bv[*ir] /= ap; i__1 = *n; for (j = 1; j <= i__1; ++j) { a[*ir + j
* a_dim1] /= ap; /* L130: */ } a[*ir + *ic * a_dim1] = 1.f / ap;

return 0; } /* trsimp_ */
```

Appendix D

S-function written in C

```
/***************************************************************************
Module:  MPC_controller Notices: Written 2001 Thomas Haugan
***************************************************************************/

////////////////////////////////////////////////////////////////////////
//* This S-function is compiled as a wrapper s-function. Calls MPC2. *//
//* Uses state space representation of the system, and a QP solver to*//
//* solve the optimisation problem. Parameters needed are taken from *//
//* init.m which is run automatically before simulation              *//
////////////////////////////////////////////////////////////////////////


#define S_FUNCTION_NAME  MPC_S_controller_3    // Name of the
file

/* Level 2 is made for Simulink 2.2, takes advantage of new
features */ #define S_FUNCTION_LEVEL 2

/* Definition of Simstruct and its associated macros. Includes
mex.h and matrix.h */ #include "simstruc.h" #include "dantzgmp.h"

//#include <stdlib.h>

void MPC2(real_T *zopt,real_T *zopt1,real_T *accum,real_T
*accum2,real_T *rhsc,real_T *rhsa, real_T *epsslack,real_T
*feasible,real_T *bas,real_T *ib,real_T *il,real_T *iter,real_T
*tab, int nrhs,const mxArray *prhs_MPC2[21]);

void dantzgmp_1(real_T *bas,real_T *ib,real_T *il, real_T
*iter,real_T *tab, int nrhs,const mxArray *prhs[]);
```

```
////////////////////////////////////////////////////////////////

/* Number of expected parameters */
 #define NPARAMS 34

/* Defining the parameters needed in code execution */
 #define PAR_A(S)       ssGetSFcnParam(S,0)
 #define PAR_B(S)       ssGetSFcnParam(S,1)
 #define PAR_C(S)       ssGetSFcnParam(S,2)
 #define PAR_D(S)       ssGetSFcnParam(S,3)
 #define PAR_Cm(S)      ssGetSFcnParam(S,4)
 #define PAR_Dv(S)      ssGetSFcnParam(S,5)
 #define PAR_Dvm(S)     ssGetSFcnParam(S,6)
 #define PAR_Bu(S)      ssGetSFcnParam(S,7)
 #define PAR_Bv(S)      ssGetSFcnParam(S,8)
 #define PAR_myindex(S) ssGetSFcnParam(S,9)
 #define PAR_mdindex(S) ssGetSFcnParam(S,10)
 #define PAR_mvindex(S) ssGetSFcnParam(S,11)
 #define PAR_nu(S)      ssGetSFcnParam(S,12)
 #define PAR_delay(S)   ssGetSFcnParam(S,13)
 #define PAR_Ts(S)      ssGetSFcnParam(S,14)
 #define PAR_degrees(S) ssGetSFcnParam(S,15)
 #define PAR_L(S)       ssGetSFcnParam(S,16)

/* Parameters which are given to MPC2 */
 #define PAR_MuKduINV(S) ssGetSFcnParam(S,17)
 #define PAR_KduINV(S)   ssGetSFcnParam(S,18)
 #define PAR_Kx(S)       ssGetSFcnParam(S,19)
 #define PAR_Ku1(S)      ssGetSFcnParam(S,20)
 #define PAR_Kut(S)      ssGetSFcnParam(S,21)
 #define PAR_Kr(S)       ssGetSFcnParam(S,22)
 #define PAR_zmin(S)     ssGetSFcnParam(S,23)
 #define PAR_rhsc0(S)    ssGetSFcnParam(S,24)
 #define PAR_Mlim(S)     ssGetSFcnParam(S,25)
 #define PAR_Mx(S)       ssGetSFcnParam(S,26)
 #define PAR_Mu1(S)      ssGetSFcnParam(S,27)
 #define PAR_Mvv(S)      ssGetSFcnParam(S,28)
 #define PAR_rhsa0(S)    ssGetSFcnParam(S,29)
 #define PAR_TAB(S)      ssGetSFcnParam(S,30)
 #define PAR_utarget(S)  ssGetSFcnParam(S,31)
 #define PAR_vKv(S)      ssGetSFcnParam(S,32)
 #define PAR_isunconstr(S) ssGetSFcnParam(S,33)
```

```
////////////////////////////////////////////////////////////////////////////////
/* ADDITIONAL PARAMETERS, ARRAY OF POINTERS IN FUNCTION CALL OF
MPC2 */
    const mxArray *prhs_MPC2[21];

////////////////////////////////////////////////////////////////////////////////
/* Function: mdlInitializeSizes
===============================================
 * Abstract:
 *     Setup sizes of the various vectors. The sizes information is used
 *     by Simulink to determine the S-function block's characteristics
 *     (number of inputs, outputs, states etc.)
 */

static void mdlInitializeSizes (SimStruct *S)   /*Initialise the
sizes array*/ {

    ssSetNumContStates(S, 0);       /* Number of continous states */
    ssSetNumDiscStates(S, 0);       /* Number of discrete states */

    /* The input port should have width "dynamically sized" and direct feedthrough */
    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S,0,DYNAMICALLY_SIZED);
    ssSetInputPortDirectFeedThrough(S,0,1);

    /* The Simulink-Block should have ONE output port for the control signal u  */
    if (!ssSetNumOutputPorts(S,1)) return;

    /* The output port (u) should have width 1 */
    ssSetOutputPortWidth(S, 0, 1);

    /*  Two pointers, one to lastx, second to lastu. Values saved for next run */
    ssSetNumPWork(S,16);

    ssSetNumSampleTimes(S, 1);              /* Number of sample times */
    ssSetNumSFcnParams(S, NPARAMS);         /* Number of extra input parameters */

    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S))
    {
        return;         /* Parameter mismatch will be reported by Simulink */
```

```
    }
}

////////////////////////////////////////////////////////////////////////////////////
/* Function: mdlInitializeSampleTimes
=========================================
 * Abstract:
 *    S-function is continuous, fixed in minor time steps.
 */
static void mdlInitializeSampleTimes(SimStruct *S)  /*Initialise
sample times array*/ {
    real_T            *Ts     = mxGetPr(PAR_Ts(S));

    if (Ts[0] > 0) {
        ssSetSampleTime(S, 0, Ts[0]);   // If Ts is given a value,
                                        // then use this, otherwise inherited
    }
    else {
        ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    }
    ssSetOffsetTime(S, 0, 0.0);
}

///////////////////////////////////////////////////////////////////////////////////
/* Function: mdlInitializeConditions
==================================================
 * Abstract:
 *    The states are stored in the lastx vector, initialize it to zero
 *    The lastt stores t at last sample time, used to check if sample time is up
 *    The lastu stores the last computed control move, used in measurement update
 */
#define MDL_INITIALIZE_CONDITIONS   /* Change to #undef to remove
function */ #if defined(MDL_INITIALIZE_CONDITIONS)

    static void mdlInitializeConditions(SimStruct *S)
    {

        int_T           nStates     = mxGetN(PAR_A(S));
        int_T           rows_rhsc0  = mxGetM(PAR_rhsc0(S));
        int_T           rows_rhsa0  = mxGetM(PAR_rhsa0(S));
        int_T           rows_xk     = mxGetM(PAR_A(S));
        int_T           M           = mxGetM(PAR_TAB(S));
```

```
        int_T           N           = mxGetN(PAR_TAB(S));

//printf("\nsize_ibi er = %i ", size_ibi);

        ///////////////////////////////////////////////////////////////////////
        /***************   ALLOCATE DYNAMIC MEMORY FOR VARIABLES   **************/
        /*  Abstract:
        *       mxCalloc initializes each element in the allocated memory to zero.
        */
        ssGetPWork(S)[0] = calloc(nStates,sizeof(real_T));
        ssGetPWork(S)[1] = calloc(1,sizeof(real_T));

        ssGetPWork(S)[2] = calloc(rows_rhsa0,sizeof(real_T));
        ssGetPWork(S)[3] = calloc(rows_rhsa0,sizeof(real_T));
        ssGetPWork(S)[4] = calloc(1,sizeof(real_T));
        ssGetPWork(S)[5] = calloc(1,sizeof(real_T));
        ssGetPWork(S)[6] = calloc(rows_rhsc0,sizeof(real_T));
        ssGetPWork(S)[7] = calloc(rows_xk,sizeof(real_T));
        ssGetPWork(S)[8] = calloc(rows_rhsa0+rows_rhsc0,sizeof(real_T));
        ssGetPWork(S)[9] = calloc(1,sizeof(real_T));
        ssGetPWork(S)[10] = calloc(1,sizeof(real_T));
        ssGetPWork(S)[11] = calloc(M,sizeof(real_T));
        ssGetPWork(S)[12] = calloc(M,sizeof(real_T));
        ssGetPWork(S)[13] = calloc(M,sizeof(real_T));
        ssGetPWork(S)[14] = calloc(1,sizeof(real_T));
        ssGetPWork(S)[15] = calloc(M*N,sizeof(real_T));
    }
#endif /* MDL_INITIALIZE_CONDITIONS */


/////////////////////////////////////////////////////////////////////////////
/* Function: mdlOutputs
============================================================
 * Abstract:
 *    This part is executed when flag value past from Simulink is 3.
 *    The calculation of the control move, and call to MPC2, which again
 *    calls QP-solver Dantzgmp is done in this section.
 */
static void mdlOutputs(SimStruct *S, int_T tid) {

    /**** Parameters taken from workspace, pointers: ****/
    real_T          *A          = mxGetPr(PAR_A(S));
```

```
real_T          *B          = mxGetPr(PAR_B(S));
real_T          *C          = mxGetPr(PAR_C(S));
real_T          *D          = mxGetPr(PAR_D(S));
real_T          *Cm         = mxGetPr(PAR_Cm(S));
real_T          *Dv         = mxGetPr(PAR_Dv(S));
real_T          *Dvm        = mxGetPr(PAR_Dvm(S));
real_T          *Bu         = mxGetPr(PAR_Bu(S));
real_T          *Bv         = mxGetPr(PAR_Bv(S));
real_T          *myindex    = mxGetPr(PAR_myindex(S));
real_T          *mdindex    = mxGetPr(PAR_mdindex(S));
real_T          *mvindex    = mxGetPr(PAR_mvindex(S));
int             *nu         = (int *)mxGetPr(PAR_nu(S));
real_T          *delay      = mxGetPr(PAR_delay(S));
real_T          *Ts         = mxGetPr(PAR_Ts(S));
int_T           degrees     = (int)(mxGetPr(PAR_degrees(S))[0]);
real_T          *L          = mxGetPr(PAR_L(S));

/*************************   Locals   ***************************/
int_T           nStates     = mxGetN(PAR_A(S));
InputRealPtrsType   uPtrs   = ssGetInputPortRealSignalPtrs(S,0);
real_T              *u_out  = ssGetOutputPortRealSignal(S,0);


///////////////////////////////////////////////////////////////////////////
//*****************   CONSTRUCTING TEMPORARY MXARRAYS    *****************//
mxArray *y_tmp,*r_tmp,*v_tmp,*xk1_tmp,*xk2_tmp,*uk1_tmp;
mxArray *yest_tmp,*xk_tmp,*uk_tmp,*basisi_tmp;
real_T  *zopt1,*accum,*accum2,*rhsc,*rhsa,*basisi,*epsslack,*feasible;
real_T *bas, *ib, *il, *iter, *tab;
///////////////////////////////////////////////////////////////////////////
//**********************         LOCALS          **********************//
real_T *y,*r,*v,*xk1,*xk2,*uk1,*yest,*xk,*zopt,*uk,*lastx,*lastu;
int size_lastx,nym,ny,nv,rows_Cm,rows_L,rows_A;
int_T i, j, k;
int cols_A,cols_L,cols_Cm;
int_T size_yrd,size_lastu;

///////////////////////////////////////////////////////////////////////////
/***********   ACCESS PARAMETERS USED IN MEMORY ALLOCATION    *************/
size_yrd = ssGetInputPortWidth(S,0);
size_lastu = ssGetOutputPortWidth(S,0);
nym = mxGetN(PAR_myindex(S));   // # cols in myindex
```

```
    ny  = mxGetM(PAR_D(S));         // # rows in D
    nv  = mxGetN(PAR_mdindex(S));   // # cols in mdindex
    size_lastx = nStates;
    rows_Cm = mxGetM(PAR_Cm(S));
    cols_Cm = mxGetN(PAR_Cm(S));
    rows_L = mxGetM(PAR_L(S));
    cols_L = mxGetN(PAR_L(S));
    rows_A = mxGetM(PAR_A(S));
    cols_A = mxGetN(PAR_A(S));

////////////////////////////////////////////////////////////////////////////
/**********   ALLOCATE DYNAMIC MEMORY FOR VARIABLES ************/
    y_tmp       = mxCreateDoubleMatrix(nym,1,mxREAL);
    r_tmp       = mxCreateDoubleMatrix(ny,1,mxREAL);
    v_tmp       = mxCreateDoubleMatrix(nv,1,mxREAL);
    xk1_tmp     = mxCreateDoubleMatrix(size_lastx,1,mxREAL);
    xk2_tmp     = mxCreateDoubleMatrix(size_lastx,1,mxREAL);
    uk1_tmp     = mxCreateDoubleMatrix(size_lastu,1,mxREAL);
    yest_tmp    = mxCreateDoubleMatrix(rows_Cm,1,mxREAL);
    xk_tmp      = mxCreateDoubleMatrix(size_lastx,1,mxREAL);
    uk_tmp      = mxCreateDoubleMatrix(size_lastu,1,mxREAL);

////////////////////////////////////////////////////////////////////////////
/**********   ASSIGN POINTERS TO EACH ALLOCATED MEMORY **********/
    lastx = ssGetPWork(S)[0];
    lastu = ssGetPWork(S)[1];
    y   = mxGetPr(y_tmp);
    r   = mxGetPr(r_tmp);
    v   = mxGetPr(v_tmp);
    xk1 = mxGetPr(xk1_tmp);
    xk2 = mxGetPr(xk2_tmp);
    uk1 = mxGetPr(uk1_tmp);
    yest = mxGetPr(yest_tmp);
    xk  = mxGetPr(xk_tmp);
    uk  = mxGetPr(uk_tmp);
/////////////////////////////////////////////////////////////////////////////
    zopt    = ssGetPWork(S)[2];
    zopt1   = ssGetPWork(S)[3];
    accum   = ssGetPWork(S)[4];
    accum2  = ssGetPWork(S)[5];
    rhsc    = ssGetPWork(S)[6];
    rhsa    = ssGetPWork(S)[7];
```

```
    basisi  = ssGetPWork(S)[8];
    epsslack = ssGetPWork(S)[9];
    feasible = ssGetPWork(S)[10];

    // for dantzgmp
    bas = ssGetPWork(S)[11];
    ib = ssGetPWork(S)[12];
    il = ssGetPWork(S)[13];
    iter = ssGetPWork(S)[14];
    tab = ssGetPWork(S)[15];
////////////////////////////////////////////////////////////////////////////

    if (size_yrd < nym){
    //  mexWarnMsgTxt("Wrong number of measured outputs");
    }

    /* Assigning values to y, r and v */
    if (nym > 0){
        for (i=0; i < nym; i++){
            y[i]= *uPtrs[i];
        }
    }

    if (ny > 0){
        for (i=nym; i <= nym+ny; i++){
           r[i-nym]= *uPtrs[i];
        }
    }

    if (nv > 0){
        for (i=nym+ny; i <= nym+ny+nv; i++){
           v[i-nym-ny]= *uPtrs[i];
        }
    }

    /* Storing the last states in xk1*/
    for (i=0; i < size_lastx; i++){
        xk1[i]= lastx[i];//lastx[i];
    }

    /* Storing the last control moves in uk1*/
    for (i=0; i < size_lastu; i++){
```

```
        uk1[i]= lastu[i];//ssGetPWork(S)[1];//lastu[i];
    }

    /*Measurement update. (yest=Cm*xk1+Dvm*v)*/
    for (i=0; i < rows_Cm; i++) {
        accum[0] = 0.0;

        for (j=0; j < cols_Cm; j++) {
            accum[0] += Cm[i+j*rows_Cm] * xk1[j];
        }

        if (nv > 0) {
            for (k=0; k < nv; k++) {
                accum[0] += Dvm[k] * v[k];
            }
        }
        yest[i] = accum[0];
    }

    /*xk2=xk1+L(y-yest)*/
    for (i=0; i < rows_L; i++) {
        accum[0] = 0.0;
        accum[0] += xk1[i];
            for (j=0; j < cols_L; j++) {
                accum[0] += L[(i+j*(rows_L))]*y[j];//nym
                accum[0] += -L[(i+j*(rows_L))]*yest[j];
            }
        xk2[i] = accum[0];
    }
//////////////////////////////////////////////////////////////////////////
/********   CALCULATION OF CONTROL LAW. CALL TO MPC2  ********/

    prhs_MPC2[0]    = PAR_MuKduINV(S);
    prhs_MPC2[1]    = PAR_KduINV(S);
    prhs_MPC2[2]    = PAR_Kx(S);
    prhs_MPC2[3]    = PAR_Ku1(S);
    prhs_MPC2[4]    = PAR_Kut(S);
    prhs_MPC2[5]    = PAR_Kr(S);
    prhs_MPC2[6]    = PAR_zmin(S);
    prhs_MPC2[7]    = PAR_rhsc0(S);
    prhs_MPC2[8]    = PAR_Mlim(S);
    prhs_MPC2[9]    = PAR_Mx(S);
```

```
    prhs_MPC2[10]   = PAR_Mu1(S);
    prhs_MPC2[11]   = PAR_Mvv(S);
    prhs_MPC2[12]   = PAR_rhsa0(S);
    prhs_MPC2[13]   = PAR_TAB(S);
    prhs_MPC2[14]   = xk2_tmp;
    prhs_MPC2[15]   = uk1_tmp;
    prhs_MPC2[16]   = PAR_utarget(S);
    prhs_MPC2[17]   = r_tmp;
    prhs_MPC2[18]   = PAR_vKv(S);
    prhs_MPC2[19]   = PAR_degrees(S);
    prhs_MPC2[20]   = PAR_isunconstr(S);


    /* Function call of MPC2 */
    MPC2(zopt,zopt1,accum,accum2,rhsc,rhsa,
          epsslack,feasible,bas,ib,il,iter,tab, 21, prhs_MPC2);

////////////////////////////////////////////////////////////////////////////


    for (i=0; i < size_lastu; i++) {
        uk[i] = uk1[i] + zopt[i];
    }


    /*Time (or state) update. Kalman*/
    /* xk = A*xk2+Bu*uk+Bv*vk; */

    for (i=0; i < size_lastx; i++) {
        accum[0] = 0.0;

        for (j=0; j < cols_A; j++) {
            accum[0] += A[i+j*rows_A]*xk2[j];
        }

        accum[0] += Bu[i]*uk[0];

        for (k=0; k < nv; k++) {
            accum[0] += Bv[k]*v[k];
        }
        xk[i] = accum[0];
        /* Store states in RWork vector */
```

```
        lastx[i] = xk[i];
    }

//////////////////////////////////////////////////////////////////////////
/**************   ASSIGN POINTERS TO EACH OUTPUT ***************/
    for (i=0; i < size_lastu; i++) {
        u_out[i] = uk[i];
        /* Store control move in RWork vector */
        lastu[i] = uk[i];
    }
//////////////////////////////////////////////////////////////////////////
/***************     DESTROY ALLOCATED MEMORY  ******************/
    mxDestroyArray(y_tmp);
    mxDestroyArray(r_tmp);
    mxDestroyArray(v_tmp);
    mxDestroyArray(xk1_tmp);
    mxDestroyArray(xk2_tmp);
    mxDestroyArray(uk1_tmp);
    mxDestroyArray(yest_tmp);
    mxDestroyArray(xk_tmp);
    mxDestroyArray(uk_tmp);

}
//////////////////////////////////////////////////////////////////////////////
/* Function: mdlUpdate
========================================================
 * Abstract:
 *    This function is called once for every major integration time step.
 *    Discrete states are typically updated here, but this function is useful
 *    for performing any tasks that should only take place once per integration
 *    step.
 */
static void mdlUpdate(SimStruct *S) { }

//////////////////////////////////////////////////////////////////////////////
//////*                  Calculation of derivatives                    *//////
//////////////////////////////////////////////////////////////////////////////

static void mdlDerivatives(double *dx, const double *x, const
double *u, SimStruct *S, int tid) { }

///////////////////////////////////////////////////////////////////////////////
```

```
//////*             Performs tasks at end of simulation                 *//////
///////////////////////////////////////////////////////////////////////////////

static void mdlTerminate(SimStruct *S) {
    int i;
    for (i = 0; i<ssGetNumPWork(S); i++) {
        if (ssGetPWorkValue(S,i) != NULL) {
            free(ssGetPWorkValue(S,i));
        }
    }
}

///////////////////////////////////////////////////////////////////////////////

#ifdef  MATLAB_MEX_FILE  // Is this file being compiled as a
                         // MEX-file?
#include "simulink.c"    // MEX-file interface mechanism
#else
#include "cg_sfun.h"     // Code generation registration function
#endif



/****************************************************************************
Module:  MPC2 Notices: Written 2001 Thomas Haugan
****************************************************************************/
////////////////////////////////////////////////////////////////////////////
// This function returns the value zopt. Compute the optimal input        //
// sequence by solving a QP problem                                       //
////////////////////////////////////////////////////////////////////////////

void MPC2(real_T *zopt,real_T *zopt1,real_T *accum,real_T
*accum2,real_T *rhsc,real_T *rhsa, real_T *epsslack,real_T
*feasible,real_T *bas,real_T *ib,real_T *il,real_T *iter,real_T
*tab, int nrhs,const mxArray *prhs_MPC2[]) {

///////////////////////////////////////////////////////////////////////////
//***************   CONSTRUCTING TEMPORARY MXARRAYS    ******************//
mxArray *ibi_tmp,*basisi_tmp,*ili_tmp,*basis_tmp,
*ib_tmp,*il_tmp,*iter_tmp;

///////////////////////////////////////////////////////////////////////////
```

```
//**********************        LOCALS        ************************//
 real_T *ibi,*ili,*basis,*basisi;
 int size_Kx,rows_Kx,cols_Ku1,rows_Kut,rows_utarget,rows_Kr;
 int cols_Kr,cols_vKv,cols_KduINV,rows_rhsc0,cols_Mx,rows_Mu1;
 int rows_rhsa0,rows_xk,nc,size_ibi,cols_Kut,rows_KduINV;
 int rows_MuKduINV,cols_MuKduINV,cols_Kx,row_basisi,col_basisi;
 int_T i,j,k;

//////////////////////////////////////////////////////////////////////////
/* Additional parameters. */
 const mxArray *rhs_d[4];
//////////////////////////////////////////////////////////////////////////
/*********  ACCESS PARAMETERS USED IN MEMORY ALLOCATION *********/
    size_Kx      = mxGetNumberOfElements(prhs_MPC2[2]);
    rows_Kx      = mxGetM(prhs_MPC2[2]);
    cols_Kx      = mxGetN(prhs_MPC2[2]);
    cols_Ku1     = mxGetN(prhs_MPC2[3]);
    rows_Kut     = mxGetM(prhs_MPC2[4]);
    cols_Kut     = mxGetN(prhs_MPC2[4]);
    rows_utarget = mxGetM(prhs_MPC2[16]);
    cols_Kr      = mxGetN(prhs_MPC2[5]);
    rows_Kr      = mxGetM(prhs_MPC2[5]);
    cols_vKv     = mxGetN(prhs_MPC2[18]);
    cols_KduINV = mxGetN(prhs_MPC2[1]);
    rows_rhsc0  = mxGetM(prhs_MPC2[7]);
    cols_Mx      = mxGetN(prhs_MPC2[9]);
    rows_Mu1     = mxGetM(prhs_MPC2[10]);
    rows_rhsa0  = mxGetM(prhs_MPC2[12]);
    rows_xk      = mxGetM(prhs_MPC2[14]);
    rows_KduINV = mxGetM(prhs_MPC2[1]);
    rows_MuKduINV   = mxGetM(prhs_MPC2[0]);
    cols_MuKduINV   = mxGetN(prhs_MPC2[0]);

    basisi_tmp  = mxCreateDoubleMatrix(rows_rhsa0+rows_rhsc0,1,mxREAL);
    basisi=mxGetPr(basisi_tmp);
//////////////////////////////////////////////////////////////////////////

    if (mxGetPr(prhs_MPC2[20])[0]==1) {
    // Unconstrained MPC :
        /* Matrix multiplication and adding to accum */
        /* zopt = -KduINV*(Kx'*xk+Ku1'*uk1+Kut'*utarget+Kr'*r+vKv')*/
```

```
for (i=0; i < cols_Ku1; i++) {
    accum[0] = 0.0;

    if (i == 0) {
        for (j=0; j < rows_Kx; j++) {
            accum[0] += mxGetPr(prhs_MPC2[2])[j+cols_Kx]
                        *mxGetPr(prhs_MPC2[14])[j]; //Kx'*xk
        }
    }
    else {
        for (j=0; j < rows_Kx; j++) {
            accum[0] += mxGetPr(prhs_MPC2[2])[j]
                        *mxGetPr(prhs_MPC2[14])[j]; //Kx'*xk
        }
    }

    for (j=0; j < cols_Ku1; j++) {
        accum[0] += mxGetPr(prhs_MPC2[3])[j]
                    *mxGetPr(prhs_MPC2[15])[0]; //Ku1*uk1
    }

    for (j=0; j < cols_Kut; j++) {
        accum[0] += mxGetPr(prhs_MPC2[4])[i+j*rows_Kut]
                    *mxGetPr(prhs_MPC2[16])[j];//Kut*utarget
    }
    for (j=0; j < rows_Kr; j++) {
        for (k=0; k < cols_Kr; k++) {
            accum[0] += mxGetPr(prhs_MPC2[5])[j+k*rows_Kr]
                        *mxGetPr(prhs_MPC2[17])[k];//Kr*r
        }
    }

    accum[0] += mxGetPr(prhs_MPC2[18])[i];//vKv

    for (j=0; j < cols_Ku1; j++) {
        accum[0] = -mxGetPr(prhs_MPC2[1])[j]
                    *accum[0];//-(KduINV[i]*(accum[0])
    }

    zopt[i] = accum[0];
}
epsslack[0] = 0;
```

```
    feasible[0] = 1;
}
else {
// Constrained MPC:
    /* rhsc = rhsc0+Mlim+Mx*xk+Mu1*uk1+Mvv; */

    for (i = 0; i < rows_rhsc0; i++) {
        accum[0] = 0.0;

        accum[0] += mxGetPr(prhs_MPC2[7])[i]
                    + mxGetPr(prhs_MPC2[8])[i]; //*rhsc0 + Mlim;

        for (j = 0; j < cols_Mx; j++) {
            accum[0] += mxGetPr(prhs_MPC2[9])[i+j*rows_rhsc0]
                        *mxGetPr(prhs_MPC2[14])[j];//Mx*xk;
        }

        accum[0] += mxGetPr(prhs_MPC2[10])[i]
                    *mxGetPr(prhs_MPC2[15])[0];// Mu1*uk1;

        accum[0] += mxGetPr(prhs_MPC2[11])[i];// Mvv[i];

        rhsc[i] = accum[0];
    }

    /* rhsa = rhsa0-[xk'*Kx+r'*Kr+uk1'*Ku1+vKv+utarget'*Kut,0]';*/
    for (i = 0; i < rows_rhsa0-1; i++) {
        accum[0] = 0.0;

        for (j = 0; j < rows_xk; j++) {
            accum[0] += mxGetPr(prhs_MPC2[14])[j]
                        *mxGetPr(prhs_MPC2[2])[j+i*rows_Kx];//xk[i]*Kx[i];
        }

        for (j=0; j < rows_Kr; j=j+2) {
            for (k=0; k < cols_Kr; k++) {
                accum[0] += mxGetPr(prhs_MPC2[17])[k]
                            *mxGetPr(prhs_MPC2[5])[j+k+i*rows_Kr];//r*Kr
            }
        }

        accum[0] += mxGetPr(prhs_MPC2[3])[i]
```

```
            *mxGetPr(prhs_MPC2[15])[0];// uk1[i]*Ku1[i];

    for (j=0; j < rows_utarget; j++) {
        accum[0] += mxGetPr(prhs_MPC2[16])[j]
                *mxGetPr(prhs_MPC2[4])[j+i*rows_Kut];//utarget*Kut;
    }

    accum[0] += mxGetPr(prhs_MPC2[18])[i];//vKv[i];

    accum2[0] = mxGetPr(prhs_MPC2[12])[i] - accum[0];//rhsa0-(accum);

    rhsa[i] = accum2[0];
}
    rhsa[rows_rhsa0] = mxGetPr(prhs_MPC2[12])[rows_rhsa0];//rhsa0;


/* basisi = [KduINV*rhsa;rhsc-MuKduINV*rhsa]; */
for (i = 0; i < rows_KduINV; i++) {
    accum[0] = 0.0;

    for (j = 0; j < cols_KduINV; j++) {
        accum[0] += mxGetPr(prhs_MPC2[1])[i+j
                *rows_KduINV]*rhsa[j];//KduINV*rhsa[j];
    }
    basisi[i] = accum[0];
}

for (i = rows_rhsa0; i < rows_rhsc0+rows_rhsa0; i++) {
    accum2[0] = 0.0;
    accum2[0] += rhsc[i-rows_rhsa0];

    for (j = 0; j < cols_MuKduINV; j++) {
        accum2[0] -= mxGetPr(prhs_MPC2[0])[i-rows_rhsa0+j*rows_MuKduINV]
                *rhsa[j];//MuKduINV[i]*rhsa[i];
    }

    basisi[i] = accum2[0];
}

nc = rows_rhsc0;

/* size_ibi = degrees + 1+ nc */
```

```
        size_ibi = mxGetPr(prhs_MPC2[19])[0] + 1 + nc;

        /* Allocates memory, initialising each element to zero */
        ibi_tmp = mxCreateDoubleMatrix(size_ibi,1,mxREAL);
        ili_tmp = mxCreateDoubleMatrix(size_ibi,1,mxREAL);



        ibi = mxGetPr(ibi_tmp);
        ili = mxGetPr(ili_tmp);

        for (i = 0; i < size_ibi; i++) {
            ibi[i] = -(1+i);
        }
        for (i = 0; i < size_ibi; i++) {
        ili[i] = -ibi[i];
        }

///////////////////////////////////////////////////////////////////////////
/* OPTIMISATION WITH RESPECT TO COST-FUNCTION. CALL TO DANTZGMP, A
   QP-SOLVER */

        rhs_d[0]    = prhs_MPC2[13];//TAB;
        rhs_d[1]    = basisi_tmp;
        rhs_d[2]    = ibi_tmp;
        rhs_d[3]    = ili_tmp;

        dantzgmp_1(bas,ib,il,iter,tab, 4, rhs_d);
///////////////////////////////////////////////////////////////////////////

        if (iter[0] >= 0) {
            feasible[0] = 1;

            for (i=0;i<mxGetPr(prhs_MPC2[19])[0]+1;i++) {   //degrees + 1

                if (il[i] <= 0)  {
                    zopt1[i] = mxGetPr(prhs_MPC2[6])[i];//zmin[i];
                }
                else  {
                    zopt1[i] =bas[i]
                              +mxGetPr(prhs_MPC2[6])[i];//basis+zmin;
                }
            }
```

```
        }

        else {
          mexWarnMsgTxt("Warning: Constraints are overly stringent");
        }
        for (i=mxGetPr(prhs_MPC2[19])[0]+1;i<mxGetPr(prhs_MPC2[19])[0]+2;i++) {
            epsslack[0] = zopt1[i];//degrees+1
        }

        for (i=0;i<mxGetPr(prhs_MPC2[19])[0];i++) {
            zopt[i] = zopt1[i];
        }

    }

    ////////////////////////////////////////////////////////////////////////////
    /*****************     DESTROY ALLOCATED MEMORY       ******************/
    mxDestroyArray(ibi_tmp);
    mxDestroyArray(ili_tmp);
}




/****************************************************************************/
// Module:  dantzgmp
// Notices: N. L. Ricker, 12/98
// Modified 5/01 Thomas Haugan for use with xPC Target. MATLAB R12
/****************************************************************************/


// Gateway to DANTZGMP c-mex routine.
// N. L. Ricker, 12/98

// MATLAB calling format:

// [bas,ib,il,iter,tab]=dantzgmp(tabi,basi,ibi,ili)

// Inputs:
// tabi : initial tableau
// basi : initial basis
// ibi : initial setting of ib
```

```
// ili : initial setting of il

// Outputs:
// bas : final basis vector
// ib : index vector for the variables -- see examples
// il : index vector for the lagrange multipliers -- see examples
// iter : iteration counter
// tab : final tableau


//#include "dantzgmp.h"

void dantzgmp_1(real_T *bas,real_T *ib,real_T *il,real_T
*iter,real_T *tab,int nrhs,const mxArray *prhs[]) {

double *tabi, *basi, *ibi, *ili; int M, N, rows, cols, iret;
 int nuc=0;
 int i, j;
 int MN = 0;
 long len;
 mxArray *ptrs[5];
 integer *ibint, *ilint, buflen;


// Verify correct number of input and output arguments.
// if (nrhs != 4);
// mexErrMsgTxt("You must supply 4 input variables.\n");
 tabi = mxGetPr(prhs[0]);
 basi = mxGetPr(prhs[1]);
 ibi = mxGetPr(prhs[2]);
 ili = mxGetPr(prhs[3]);
//if (nlhs < 3);
// mexErrMsgTxt("You must supply at least 3 output variables.\n");

// Error checking on inputs
// Checking TABI
 M = mxGetM(prhs[0]);
 N = mxGetN(prhs[0]);
 if (M <= 0 || N <= 0) ;
//mexErrMsgTxt("TABI is empty.\n");
// Checking BASI
 rows = mxGetM(prhs[1]);
```

```
 cols = mxGetN(prhs[1]);
 len = max(rows,cols);
 if (min(rows,cols) != 1 || len != M) ;
//mexErrMsgTxt("BASI must be a vector, length = number of rows in TABI.\n");
// Checking IBI
 rows = mxGetM(prhs[2]);
 cols = mxGetN(prhs[2]);
 len = max(rows,cols);
 if (min(rows,cols) != 1 || len != M) ;
//mexErrMsgTxt("IBI must be a vector, length = number of rows in TABI.\n");
// Checking ILI
 rows = mxGetM(prhs[3]);
 cols = mxGetN(prhs[3]);
 len = max(rows,cols);
 if (min(rows,cols) != 1 || len != M) ;
//mexErrMsgTxt("ILI must be a vector, length = number of rows in TABI.\n");

// Allocate space for output variables and define corresponding C pointers
//** Changed 31/5-01 by Thomas Haugan.                                **//
//** Allocated work-vectors in S-function initialization. The work- **//
//** vectors are deallocated at termination of the simulation       **//


// We have to convert ib and il from double to integer and vice-versa.
// Allocate arrays for storing the integer versions.
 buflen = M * sizeof(*ibint);
 ibint = (integer *) mxMalloc(buflen);
// Pointer to integer version of ib
 ilint = (integer *)mxMalloc(buflen); // Pointer to integer version of il


// Initialization
for (i=0; i<M; i++) {
   bas[i] = basi[i];
   ibint[i] = (integer) ibi[i];
   ilint[i] = (integer) ili[i];
}


for (j=0; j<N; j++) {
    for (i=0; i<M; i++) {
        tab[MN] = tabi[MN];
```

```
        MN++;
    }
}


// Call DANTZG for the calculations

iret = dantzg(tab, &N, &N, &nuc, bas, ibint, ilint);
// Store number of iterations.
*iter = (double) iret;

// Return results to MATLAB. First convert integer versions
// of ib and il back to real, then set pointers to outputs.
for (i=0; i<M; i++) { ib[i] = (double) ibint[i]; il[i] = (double)
ilint[i]; }

////////////////////////////////////////////////////////////////////////////
/************     DESTROY LOCALLY ALLOCATED MEMORY *************/
 mxFree(ibint);
 mxFree(ilint);
////////////////////////////////////////////////////////////////////////////
}

/* Subroutine */
 int dantzg(doublereal *a, int *ndim, int *n, int
            *nuc, doublereal *bv, integer *ib, integer *il) {
 /* System generated locals */
  integer a_dim1, a_offset, i__1;

/* Local variables */
 integer ichk, iter;
 doublereal rmin, test;
 integer iout, i, ichki, ic, ir, nt, istand, irtest;
 extern /* Subroutine */ int trsimp_(doublereal *, int *, integer *, int *,
 doublereal *, integer *, integer *);
 integer iad; doublereal val, rat;
 int iret=-1;


/* ************************************************** */

// VERSION MODIFIED 1/88 BY NL RICKER
```

```
// Modified 12/98 by NL Ricker for use as MATLAB MEX file

/* ************************************************** */

/* DANTZIG QUADRATIC PROGRAMMING ALGORITHM. */

/* N.L. RICKER 6/83 */

/* ASSUMES THAT THE INPUT VARIABLES REPRESENT A FEASIBLE INITIAL
*/ /* BASIS SET. */

/* N NUMBER OF CONSTRAINED VARIABLES (INCLUDING SLACK
VARIABLES).*/

/* NUC NUMBER OF UNCONSTRAINED VARIABLES, IF ANY */

/* BV VECTOR OF VALUES OF THE BASIS VARIABLES. THE LAST NUC */ /*
ELEMENTS WILL ALWAYS BE KEPT IN THE BASIS AND WILL NOT */ /* BE
CHECKED FOR FEASIBILITY. */

/* IB INDEX VECTOR, N ELEMENTS CORRESPONDING TO THE N VARIABLES.
*/ /* IF IB(I) IS POSITIVE, THE ITH */ /* VARIABLE IS BASIC AND
BV(IB(I)) IS ITS CURRENT VALUE. */ /* IF IB(I) IS NEGATIVE, THE
ITH VARIABLE IS NON-BASIC */ /* AND -IB(I) IS ITS COLUMN NUMBER IN
THE TABLEAU. */

/* IL VECTOR DEFINED AS FOR IB BUT FOR THE N LAGRANGE
MULTIPLIERS.*/

/* A THE TABLEAU -- SEE TRSIMP DESCRIPTION. */

/* IRET IF SUCCESSFUL, CONTAINS NUMBER OF ITERATIONS REQUIRED. */
/* OTHER POSSIBLE VALUES ARE: */ /* - I NON-FEASIBLE BV(I) */ /*
-2N NO WAY TO ADD A VARIABLE TO BASIS */ /* -3N NO WAY TO DELETE A
VARIABLE FROM BASIS */ /* NOTE: THE LAST TWO SHOULD NOT OCCUR AND
INDICATE BAD INPUT*/ /* OR A BUG IN THE PROGRAM. */



/* CHECK FEASIBILITY OF THE INITIAL BASIS. */

/* Parameter adjustments */ --il; --ib; --bv; a_dim1 = *ndim;
```

```
a_offset = a_dim1 + 1; a -= a_offset;

/* Function Body */ iter = 1; nt = *n + *nuc; i__1 = *n; for (i =
1; i <= i__1; ++i) { if (ib[i] < 0 || bv[ib[i]] >= 0.f) { goto
L50; } iret = -i; goto L900; L50: ; } istand = 0; L100:

/* SEE IF WE ARE AT THE SOLUTION. */

if (istand != 0) { goto L120; } val = 0.f; iret = iter;

i__1 = *n; for (i = 1; i <= i__1; ++i) { if (il[i] < 0) { goto
L110; }

/* PICK OUT LARGEST NEGATIVE LAGRANGE MULTIPLIER. */

test = bv[il[i]]; if (test >= val) { goto L110; } val = test; iad
= i; ichk = il[i]; ichki = i + *n; L110: ; }

/* IF ALL LAGRANGE MULTIPLIERS WERE NON-NEGATIVE, ALL DONE. */ /*
ELSE, SKIP TO MODIFICATION OF BASIS */

if (val >= 0.f) { iret=iter; goto L900; } ic = -ib[iad]; goto
L130;

/* PREVIOUS BASIS WAS NON-STANDARD. MUST MOVE LAGRANGE */ /*
MULTIPLIER ISTAND INTO BASIS. */

L120: iad = istand; ic = -il[istand - *n];

/* CHECK TO SEE WHAT VARIABLE SHOULD BE REMOVED FROM BASIS. */

L130: ir = 0;

/* FIND SMALLEST POSITIVE RATIO OF ELIGIBLE BASIS VARIABLE TO */
/* POTENTIAL PIVOT ELEMENT. FIRST TYPE OF ELIGIBLE BASIS VARIABLE
*/ /* ARE THE REGULAR N VARIABLES AND SLACK VARIABLES IN THE
BASIS. */

i__1 = *n; for (i = 1; i <= i__1; ++i) { irtest = ib[i];

/* NO GOOD IF THIS VARIABLE ISN'T IN BASIS OR RESULTING PIVOT
WOULD */ /* BE ZERO. */
```

```
if (irtest < 0 || a[irtest + ic * a_dim1] == 0.f) { goto L150; }
rat = bv[irtest] / a[irtest + ic * a_dim1];

/* THE FOLLOWING IF STATEMENT WAS MODIFIED 7/88 BY NL RICKER */ /*
TO CORRECT A BUG IN CASES WHERE RAT=0. */

if (rat < 0.f || rat == 0.f && a[irtest + ic * a_dim1] < 0.f) {
goto L150; } if (ir == 0) { goto L140; } if (rat > rmin) { goto
L150; } L140: rmin = rat; ir = irtest; iout = i; L150: ; }

/* SECOND POSSIBLITY IS THE LAGRANGE MULTIPLIER OF THE VARIABLE
ADDED*/ /* TO THE MOST RECENT STANDARD BASIS. */

if (a[ichk + ic * a_dim1] == 0.f) { goto L170; } rat = bv[ichk] /
a[ichk + ic * a_dim1]; if (rat < 0.f) { goto L170; } if (ir == 0)
{ goto L160; } if (rat > rmin) { goto L170; } L160: ir = ichk;
iout = ichki;

L170: if (ir != 0) { goto L200; } iret = *n * -3; goto L900;

L200:

/* SET INDICES AND POINTERS */

if (iout > *n) { goto L220; } ib[iout] = -ic; goto L230; L220:
il[iout - *n] = -ic; L230: if (iad > *n) { goto L240; } ib[iad] =
ir; goto L250; L240: il[iad - *n] = ir; L250:

/* TRANSFORM THE TABLEAU */

trsimp_(&a[a_offset], ndim, &nt, n, &bv[1], &ir, &ic); ++iter;

/* WILL NEXT TABLEAU BE STANDARD? */

istand = 0; i__1 = *n; for (i = 1; i <= i__1; ++i) { /* L260: */
if (ib[i] > 0 && il[i] > 0) { goto L270; } } goto L280; L270:
istand = iout + *n; L280: goto L100;

L900: return iret; } /* dantzg_ */

/* Subroutine */ int trsimp_(doublereal *a, int *ndim, integer *m,
```

```
int *n, doublereal *bv, integer *ir, integer *ic) { /* System
generated locals */ integer a_dim1, a_offset, i__1, i__2;

/* Local variables */ integer i, j; doublereal ap;


/* TRANSFORM SIMPLEX TABLEAU. SWITCH ONE BASIS VARIABLE FOR ONE */
/* NON-BASIC VARIABLE. */

/* N.L. RICKER 6/83 */

/* A SIMPLEX TABLEAU. ACTUALLY DIMENSIONED FOR NDIM ROWS IN */ /*
THE CALLING PROGRAM. IN THIS PROCEDURE, ONLY THE A(M,N) */ /*
SPACE IS USED. */

/* NDIM ACTUAL ROW DIMENSION OF A IN THE CALLING PROGRAM */

/* M NUMBER OF ROWS IN THE TABLEAU */

/* N NUMBER OF COLUMNS IN THE TABLEAU */

/* BV VECTOR OF M BASIS VARIABLE VALUES */

/* IR ROW IN TABLEAU CORRESPONDING TO THE BASIC VARIABLE THAT */
/* IS TO BECOME NON-BASIC */

/* IC COLUMN IN TABLEAU CORRESPONDING TO THE NON-BASIC VARIABLE */
/* THAT IS TO BECOME BASIC. */



/* FIRST CALCULATE NEW VALUES FOR THE NON-PIVOT ELEMENTS. */

/* Parameter adjustments */ --bv; a_dim1 = *ndim; a_offset =
a_dim1 + 1; a -= a_offset;

/* Function Body */ i__1 = *m; for (i = 1; i <= i__1; ++i) { if (i
== *ir) { goto L110; } ap = a[i + *ic * a_dim1] / a[*ir + *ic *
a_dim1]; bv[i] -= bv[*ir] * ap; i__2 = *n; for (j = 1; j <= i__2;
++j) { if (j == *ic) { goto L100; } a[i + j * a_dim1] -= a[*ir + j
* a_dim1] * ap; L100: ; } L110: ; }
```

```
/* NOW TRANSFORM THE PIVOT ROW AND PIVOT COLUMN. */

ap = a[*ir + *ic * a_dim1]; i__1 = *m; for (i = 1; i <= i__1; ++i)
{ a[i + *ic * a_dim1] = -a[i + *ic * a_dim1] / ap; /* L120: */ }

bv[*ir] /= ap; i__1 = *n; for (j = 1; j <= i__1; ++j) { a[*ir + j
* a_dim1] /= ap; /* L130: */ } a[*ir + *ic * a_dim1] = 1.f / ap;

return 0; } /* trsimp_ */
```

Appendix E

Photos of “Ball & Plate” system

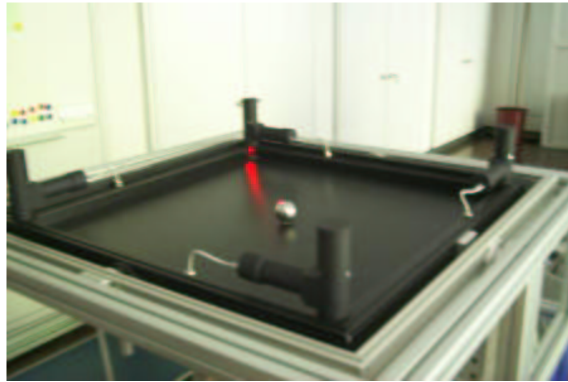

Figure E.1: Photo of “Ball & Plate” system.

Figure E.2: Photo of “Ball & Plate” system.

Figure E.3: Photo of “Ball & Plate” system.

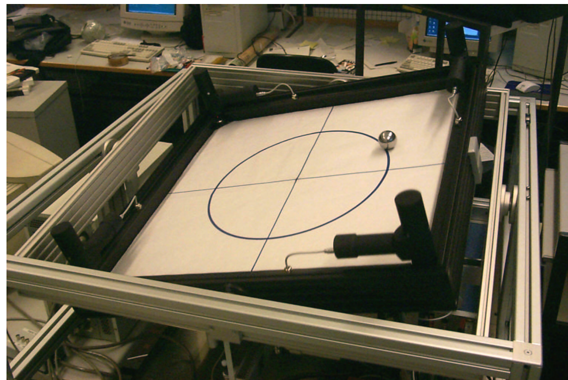

Figure E.4: Photo of ball tracking a circle reference.