Real-time implementation of Model Predictive Control (MPC)

TU/e technische universiteit eindhoven

Faculty of Mechanical Engineering
Dynamics & Control Group

Real-time implementation of Model Predictive Control (MPC)

J.J.A.M. Keij BSc

DCT 2001.24

Supervisor: dr. ir. H.A. van Essen

Table of Contents

Introduction

1.1 Introduction

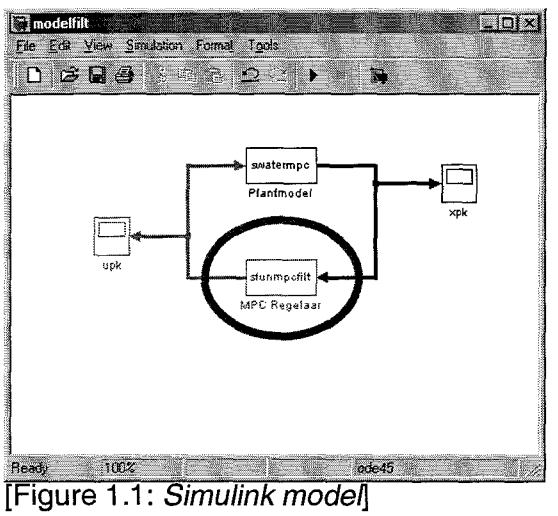
In this report the results and conclusions of my first internal practical training are presented. The used control strategy is a quite modern, still developing one. Starting in the early 1980's Model Predictive Controllers (or related strategies) has developed as a very powerful control strategy, among others due to the MPC ability of taking constraints of the system into account. Now computer capabilities are increasing fast, the computational demands of MPC controllers can be provided. A real-time implementation of a (state space) Model Predictive Controller can now be very interesting for a certain class of systems, which are not easy controllable with the conventional control strategies.
To introduce the principles of the class of controllers which are investigated a short overview will be given. This report does not give a complete overview of the whole class of Model Predictive Controllers, but just a brief introduction. In the next sections the problem and proposed approach will be stated.

1.2 Statement of the problem

The main objective of this traineeship is to design a real-time implementation of a model predictive controller. As a starting point a Matlab simulation script was available. This MPC controller was well implemented in a very clear structure. The separate elements of this controller are quite well distinguishable. Minor objectives of this traineeship were genericity (the controller should be model independent), and the design of a user-friendly interface for the user-inputs.

1.3 Approach

During this traineeship the first step was to understand the principles of MPC, in specific the MPC controller in state space approach which was available for simulation in Matlab.
Starting from a well-structured Matlab simulation the first step to a real-time implementation is the transcription to a Matlab/Simulink function. This function represents the controller. Simulations with this simulink element and a (non-linear) process model will increase knowledge of the closed loop behaviour of this MPC controller. Parameters can easily be adapted, such that this controller simulation can be very well used for demonstrating the working of MPC controllers.
In case of an experimental or industrial setup, this (non-linear) process model can be replaced with an interface to a real-time processor board, like dSpace. To



[Figure 1.1: *Simulink model*]

obtain a real-time implementation, the simulink system can be compiled with a tool like Real-Time Workshop (RTW). This method to obtain a real-time implementation is chosen over direct implementation in a low-level programming environment such as C/C++, because the intermediate stage in which the Simulink controller is obtained, is considered valuable.

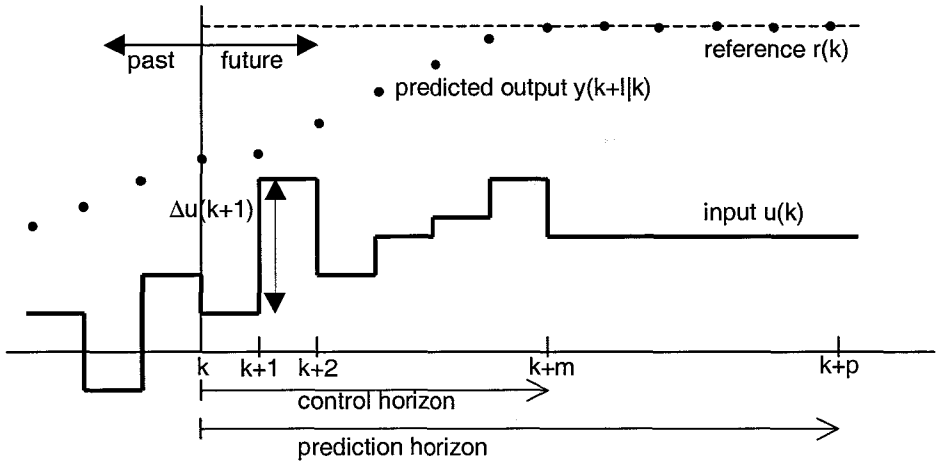
2. Model Predictive Control

2.1 Introduction of MPC

Model Predictive Control is a class of discrete time controllers, which base the input signal on a prediction of future outputs of the system (process). These predictions are based on a model of the system (process) that is to be controlled. The main technique behind this concept is the principle of receding or moving horizons. Due to the model-based approach the online optimisation can take constraints in to account with respect to input signals, controlled and uncontrolled states. Because of the large calculations due to this principle, MPC is most suitable and most applied for relatively 'slow' processes. Although due to fast increasing processor capabilities an increasing number of processes could be controlled with a Model Predictive Controller nowadays.
In the following sections a brief overview of the principles and features of MPC will be given. More information can be found in the lecture notes [Van Essen, 2000].

2.2 Concept of MPC

The scheme presented in figure 2.1 describes the principle of receding horizons that is applied to Model Predictive Controllers. For convenience only one input and one state is considered but for MIMO systems this principle holds just as well.



[Figure 2.1: *Concept of Model Predictive Control*]

At present time k, the response of the output is predicted over a prediction horizon with a length of p samples. The prediction is based on past inputs, current model states (or estimates of the states), latest process measurements, proposed future inputs, and if possible the predicted setpoint disturbances. The manipulated variables are allowed to vary over a control horizon with a length of m samples. The optimal input changes Δu are calculated by minimising a quadratic objective function in the tracking error (*y-r*) and the input changes Δu. Only the input(change) of the next sample is implemented. The next sample the whole procedure is repeated. This way the horizons are moving in time: receding horizons.
To minimise future deviations of the controlled variables from their reference values (setpoints or trajectories), while preventing the inputs from changing inadmissibly fast, the next (common for MPC) quadratic objective function (in y – r and Δu) is used.

$$\min_{\Delta u(k+1)\ldots\Delta u(k+m)} \sum_{l=2}^{p} [\mathbf{Q}(y(k+l\,|\,k) - r(k+l)]^2 + \sum_{l=1}^{m} [\mathbf{R}(\Delta u(k+l))]^2 \qquad (2)$$

Inspecting the quadratic objective function in the (filtered) process output, the reference signal and the input change Δu we see two weighting matrices. **Q** represents the weightings of the setpoints over the prediction horizon. The matrix **R** represents the weighting of the input changes Δu over the control horizon. These weightings may change over the horizons. Furthermore, $y(k+l\,|\,k)$ denotes the estimate of $y(k+l)$ obtained at sample k, taking into account all (available) information up to and including the current sample k.

2.3 Tuning parameters in MPC

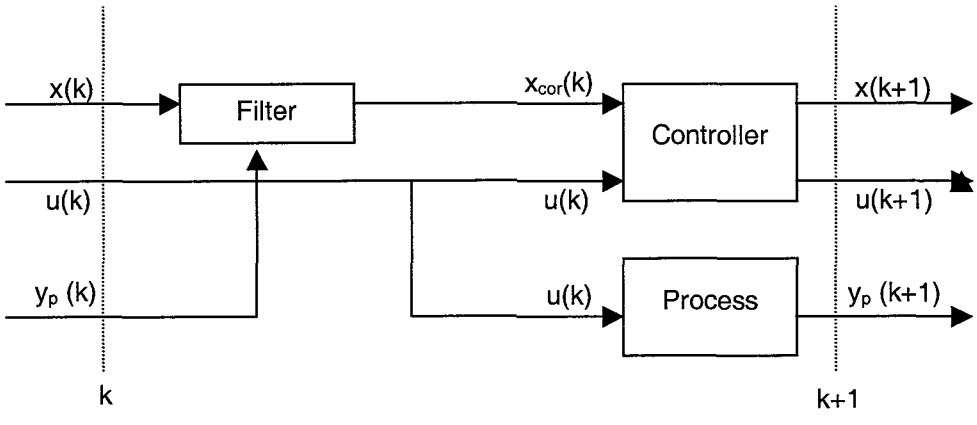
Summarising the tuning parameters from the previous sections, the next parameters are obtained:

- the number of samples in (or the length of) the prediction horizon
- the number of samples in (or the length of) the control horizon
- the sample interval
- the setpoint weighting factors
- the input weighting factors

The controller must be able to observe the consequences of its control actions. Therefore the prediction horizon should exceed the largest time constant of the controlled system, periods of dead-time and periods of inverse response.
System speed decreases when the prediction horizon is increased with respect to the control horizon, although the open-loop robustness of the system increases in this case. A good compromise should be chosen. Basic guidelines for tuning are formulated by [Morari, 1993] on basis of open-loop stable, minimum phase processes, that show responses corresponding to first-order responses.
Increasing lengths of prediction and control horizons, and decreasing the sample interval will increase computation times substantial as the number of d.o.f. in the optimisation rises exponential.

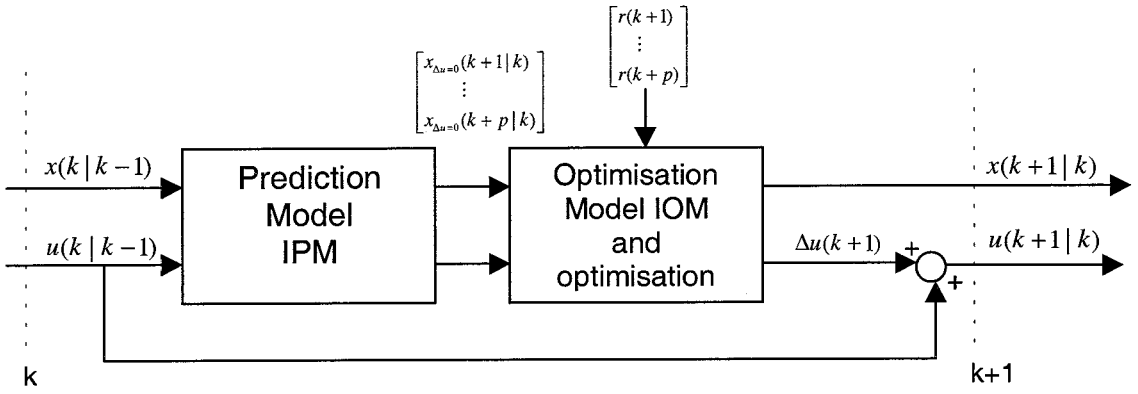
2.4 Implementation of Linear MPC

In case of a linear optimisation and a non-linear process model an error in introduced. This error can be corrected for by implementing a filter. This filter can be model-based (e.g. (Extended) Kalman filter) or non model-based like the implemented output disturbance filter.



[Figure 2.2: *Schematic view of MPC Implementation*]

The MPC controller is fed with the current input, the desired setpoint, previous prediction of the states and latest measurements of the process states (outputs). The controller calculates a prediction of the process outputs over the prediction horizon based on the current (unchanged) input and corrected state prediction by means of the (Internal) Prediction Model (IPM). This prediction vector and the current input vector are fed into the Internal Optimisation Model (IOM). The calculation in this optimisation algorithm results in an optimal input change. This input change is summarised with the current input resulting in a new optimal input for the process.



[Figure 2.2: *Schematic view of MPC Controller*]

Details and priciples of the optimisation algorithms can be found in the lecture notes [Van Essen, 2000]. There are roughly two classes of optimisation algorithms. The relatively simple unconstrained optimisation models (e.g. Least Squares LS), and the optimisations that can take constraints into account (e.g. Quadratic Programming QP). Constraints can be formulated with respect to the inputs or the states. Even uncontrolled states can be constrained. These constraints can be upper or lower limits due to safety of physical limitations of the system, or move constraints, like limitations to the actuator.
These QP problems are in most cases solved by standard routines, as they are available in NAG routines, Fortran, C/C++ and Matlab MEX files. In the implemented controller the Matlab routine QUADPROG is used, but other routines can easily be applied.

2.5 Filter and reconstruction

To correct the predicted states towards the actual measurements of the process a filter has been implemented. Besides differences due to errors in the used models and the linear predictions of non-linear systems, noise can introduce errors too, which are to be corrected.
The implemented first order non-model based filter corrects the current predicted states $\{x_m(k+1\,|\,k) \;\ldots\; x_m(k+p\,|\,k)\}$ with the difference between the previously predicted state $x_m(k\,|\,k-1)$ and the current process measurement $x_p(k\,|\,k)$. This class of filters is called output disturbance filters.

2.6 Non-linear predictive control

The previous principles all hold for the standard linear Model Predictive Control problem. The unconstrained (Least Squares) optimisation and the constrained

optimisation problem (Quadratic Programming) are proven algorithms. Linear approaches will not always be adequate, because most processes behave quite non-linear and can not always be linearised with satisfying results.
Non-linear MPC concepts are developed to deal with these cases. These MPC concepts calculate non-linear predictions over the horizon. Most of these algorithms are computational very demanding because of the increased complexity of the problems. A less demanding solution can be found in successive linearising. In that case the prediction model is linearised around the current setpoint each sample. This extension increases the computational demand, since it requires a sequential (SQP) optimisation problem instead of a single QP problem over the horizon.

3 Matlab/Simulink implementation

As an intermediate between MPC simulation and real-time implementation a Matlab/Simulink S-function was designed. This function can be applied in a Simulink-model in which simulations can be conducted and under some limitations the Real Time Workshop tool (RTW) can convert the simulated system from Simulink to executable code for a real-time (control-) processor like dSpace or the TUeDACS device.

3.1 Design criteria

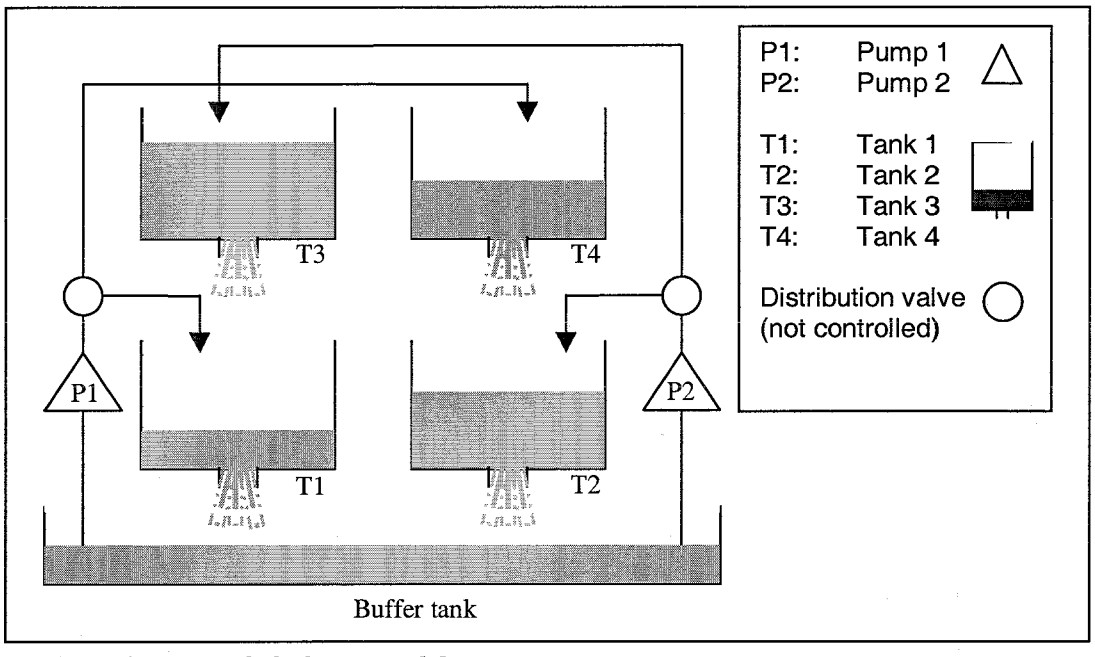
As stated earlier the main target of this practical training was to implement real-time version of a generic Linear Model Predictive Controller.
The first part of that goal consists of the word *real-time.* This implies that the generated code is able to measure the outputs at certain moments in time and apply inputs to the process, for example by means of the processor on the available dSpace systems. There are some tools available to create the necessary code such as the Real Time Workshop (RTW). Nevertheless this tool implies some limitations to the used functions and settings in the Simulink model. Not only the function itself has to be compiled but also the external functions or subroutines that are called by the main function are to be compiled.
The second part of the goal consists of the word generic. This means that the code can be easily be adapted to another model or controlled system. The controller itself should be *model independent.* The original MPC simulation was applied on a system of cross-linked water tanks (described in 3.*2 Verification systems*).
Besides these two hard design-criteria there are of course some criteria that are as important as the main target. These criteria are stated with respect to user-friendly interfaces, easy replaceable prediction models and extendable controller features.
Of course the S-function simulations should give exactly the same results as the original Matlab simulations in case the same systems, parameters and features are used.
Let's now recite all design criteria; The intermediate s-function first of all should give exactly the same output as the reference simulations in case same systems, parameters and features are used; The s-function should be RTW compatible, no variable step ODE-solvers may be used; The controller should be generic, no model dependent elements should exist within the controller; Parameters should be easily adaptable via a user-friendly interface (no messing around in the code).

3.2 Verification systems

Model Predictive Controllers are of course model-based controllers. The controller uses information of the model of the controlled system to predict its response and optimise the current input. In this practical training the main goal was to design a generic controller, one that could be applied on all systems of which a state space model is supplied. To verify that the controller works correct two systems have been chosen as verification systems. The first one was supplied in the original simulation code, the second one was chosen from [Franklin et al, 1994] as an arbitrary non-linear system.

3.2.1 Cross-linked water tanks

The first used verification system consists of 4 cross-linked water tanks, two pumps and a buffer-tank (figure 3.1: *Cross-linked water tanks*). The first pump (1) pumps up water from the buffer-tank and pumps the water into the upper left tank (3) and into the lower right tank (1). The second pump (2) pumps the water from the buffer into the lower left tank (2) and into upper right tank (4). The two upper tanks (3 and 4) are emptying into the lower tanks (respectively 1 and 2). The water levels in all tanks are supposed to be the (measured) states and the charge of the both pumps are supposed to be the two inputs of the system.
Of course the tanks have a finite height and the tanks cannot have a water level beneath zero.

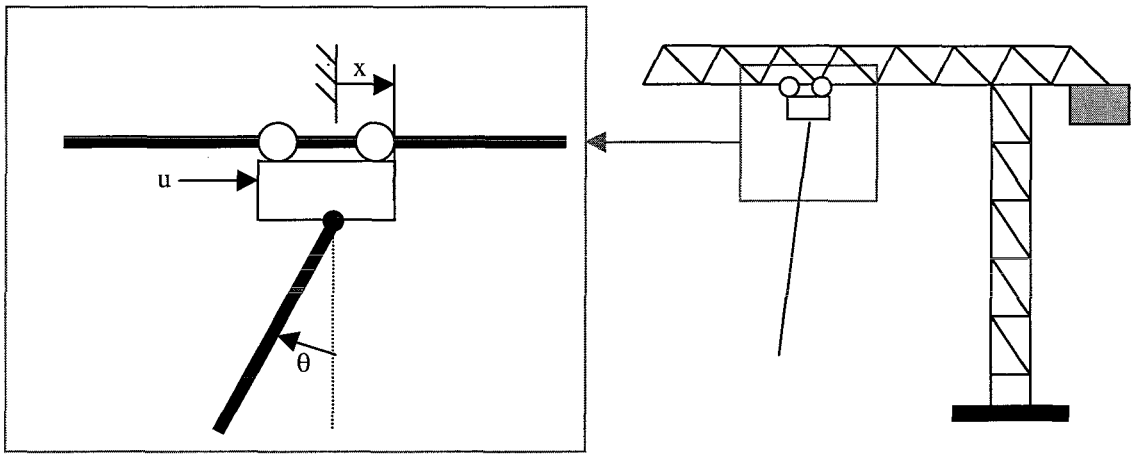


[figure 3.1: *Cross-linked water tanks*]

It is not easy to follow a certain setpoint reference on the both lower tanks, because of the emptying of the upper tanks without any knowledge of the system. If a pump pumps water into a low-levelled tank the other tank will be filled as well, direct through the pump or indirect via an upper tank. Nevertheless if the controller uses knowledge of the system this system can be controlled with quite satisfying results.
To illustrate the reference system some results of the original simulations will be presented. As can be seen the results with the Model Predictive Controller are quite acceptable. Because MPC is widely applied in process industries this example is a good illustration of the power of this control strategy.

3.2.2 Hanging crane

To be sure the controller does not contain any water tank dependent elements a second model was chosen to verify the correct performance of the designed s-function. This model describes a horizontally moving cart with a hanging pendulum beneath it (hanging crane) in a gravitational force field (figure 2.2: *Hanging crane*). This model describes some friction in de moving of the cart, but no damping in the pendulum. The (rotational-) position and speed of both cart and pendulum are measured.



[Figure 3.2: *Hanging crane*]

A crane-driver for example could want to move the crane from position A to B having the pendulum at least at the end of the trajectory in a vertical equilibrium position. MPC can take several constraints in account, such as maximum speeds and maximum positions of cart and pendulum and calculate optimal control inputs for this system with 4 measured states and 1 input.

3.3 Generic Multimask MPC

A MPC simulation is available as well as two test models. The intermediate Matlab/Simulink s-function is written in Matlab format, in stead of C/C++ format. This choice can still be made if this turns out to be better.
The controller could be constructed from 3 subsystems (controller, model and filter) in a simulink style, with basic simulink library elements. This does not appear to be the optimal choice with respect to the user-interface and replaceable models. In this case the filter is integrated within the controller in one single s-function, because of the easier calculation of the disturbance of the prediction with respect to the actual measurement. The prediction model is implemented in an external function, which can be called by the controller. The optimisation is done by means of the Matlab routine QUADPROG.

3.3.1 S-function

A Simulink s-function has a very stringent format. To illustrate this, let's now analyse the header to understand the working of s-functions better.

```
function [sys,x0,str,Ts] = sfunname(t,x,u,flag)
```

Input argument t (time) will be called by Simulink. The next three input arguments deserve some more attention. Simulink have defined s-functions in general, so x and u describing respectively the state and input of the function. In this special case of the MPC describing s-function the state of the controller describes the current input-signal of the process and the input of the controller are the current measured states of the process (See figure 3.3: *Simulink Model*). Current process inputs and states are denoted as $x_p(k)$ (state x of the process at time k), or xpk. The inputs are denoted analogue. So if x is replaced by upk, and u by xpk in the header of the s-function, variables are denoted intuitive.

```
function [sys,x0,str,Ts] = sfunmpc(t,upk,xpk,flag)
```

Next input argument is the *flag* argument. This is a variable that Simulink uses to describe the status of the function. Next a short table is included from a Matlab template in which the use of the flag argument is described.

```
%   FLAG    RESULT              DESCRIPTION
%   -----   ------              -------------------------------------------
%   0       [SIZES,X0,STR,TS]   Initialization, return system sizes in SYS,
%                               initial state in X0, state ordering strings
%                               in STR, and sample times in TS.
%   1       DX                  Return continuous state derivatives in SYS.
%   2       DS                  Update discrete states SYS = X(n+1)
%   3       Y                   Return outputs in SYS.
%   4       TNEXT               Return next time hit for variable step sample
%                               time in SYS.
%   5                           Reserved for future (root finding).
%   9       []                  Termination, perform any cleanup SYS=[].
%
```

[Table 3.1: *Use of the flag arguments (The Mathworks)*]

To illustrate the flag-call the outputs of the function at flag=0 are described as follows:

```
%   When SFUNC is called with FLAG = 0, the following information
%   should be returned:
%
%      SYS(1) = Number of continuous states.
%      SYS(2) = Number of discrete states.
%      SYS(3) = Number of outputs.
%      SYS(4) = Number of inputs.
%               Any of the first four elements in SYS can be specified
%               as -1 indicating that they are dynamically sized. The
%               actual length for all other flags will be equal to the
%               length of the input, U.
%      SYS(5) = Reserved for root finding. Must be zero.
%      SYS(6) = Direct feedthrough flag (1=yes, 0=no). The s-function
%               has direct feedthrough if U is used during the FLAG=3
%               call. Setting this to 0 is akin to making a promise that
%               U will not be used during FLAG=3. If you break the promise
%               then unpredictable results will occur.
%      SYS(7) = Number of sample times. This is the number of rows in TS.
%
%
%      X0     = Initial state conditions or [] if no states.
%
%      STR    = State ordering strings which is generally specified as [].
%
%      TS     = An m-by-2 matrix containing the sample time
%               (period, offset) information. Where m = number of sample
%               times.
```

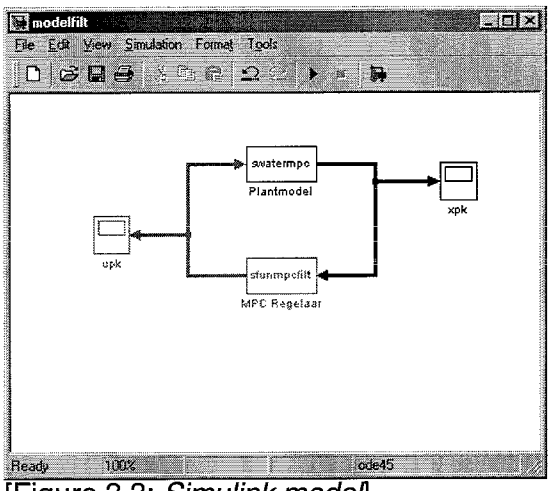
[Table 3.2: *Outputs at initialisation (The Mathworks)*]

In the initialisation stage (flag 0) the controller calculates initial parameter values, (linearised) system matrices and sets initial state and input values. The required parameters are save to disk. This is not very positive to minimise the used time per sample, because it implies disk activity.
During the control stage the controller can load the needed values from disk and use these parameters to calculate the next input signal for the process. Because in this case there are no continuous states, the only state update of the controller is implemented under flag 2 (update discrete states of the s-function) when all calculations that are needed that sample are finished, the controller saves all needed parameters to disk again. Flag 3 works as a 'C matrix' in control theory. It describes which (updated) states are used to describe the output of the function. In this case all states are used.
The flag 4 call calculates the 'time of next hit'. This is of course strongly dependent on the set sample time. By means of this calculation simulink is able to synchronise the timing of the elements (process model, controller) within the system In this special case, the process model does not calculate a 'time of next hit' but supplies a process state value whenever the controller *asks* for it. Within the controller a zero order hold is applied to feed the process continuously with the current calculated input signal. Finally the flag 9 call terminates the function.

3.3.2 Simulink model

To simulate the working of the implemented Model Predictive Controller a test case is needed. In this case the model of cross-linked water tanks described in 3.2.1 is used.

The (non-linear) equations of this system were implemented in a continuous s-function. This means that the states (water heights) are available 'whenever' they are needed in the controller and the discrete inputs are applied continuously.
The general structure of the designed simulink model is shown here. It may be clear that the s-function describing the non-linear *Plant model* will be replaced with the interface with the *real process.* In case a real-time processor is available (e.g. dSpace) the simulink block *Plant model* will be replaced with a communication block, which is supplied with the real-time processor. This block will cope with the communication to and from the process.



[Figure 3.3: *Simulink model*]

3.3.3 Graphical User Interface

To make an easy-to-use controller a GUI has been designed in which a user can make adaptations to the parameters of the controller, such as e.g. sample time, horizon lengths and filter configuration.
To prevent 'messing around in the code' and supply a clear overview of the chosen parameters a *Simulink 'mask'* has been designed. If a user now double clicks on the controller in the simulink model, this mask will be opened. It contains four masks (system, controller, constraints and filter) which lead the user through all needed parameters, such as prediction model, sample time, filter gain and constraints. Clicking System, Controller, Constraints or Filter buttons will open the different masks. In this case some parameters of the cross-linked water tanks are filled in.

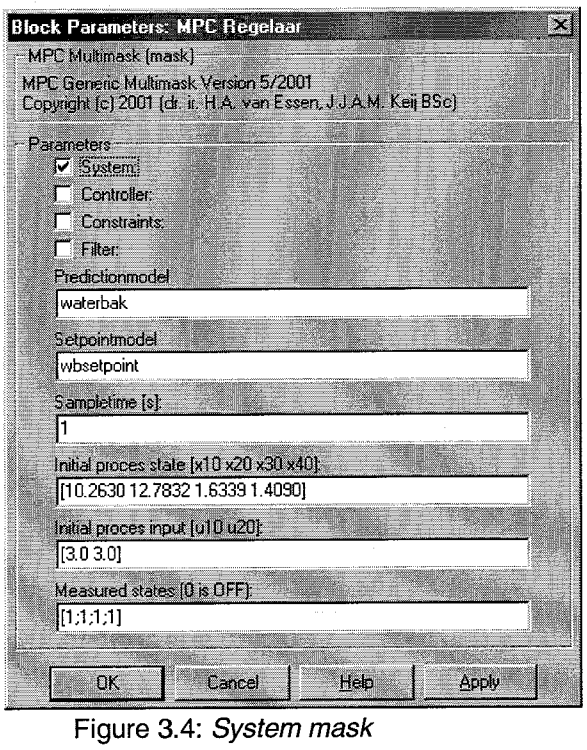


Figure 3.4: *System mask*

A short manual for extending the mask if the features of the controller are extended is supplied in appendix B.

If the controller has to be installed from scratch, all masks should be passed through. A manual for this procedure is supplied in Appendix A. Starting top-to-bottom (System mask to Filter). Some parameters are filenames (e.g. prediction model, setpoint model), some are scalars (e.g. sample time), and others are vectors or matrices (e.g. constraints, initial state approximations). The controller uses some external files and functions, which are

described in the next section. If all parameters are filled in values can be confirmed by pressing *Apply* or *Ok.* If everything is in place, the simulation can start.

3.3.4 Callback functions

Because of the vast amount of user input to configure the MPC controller, not all input can be put in one standard simulink mask. Therefore the parameters are distributed over four masks. This is no standard feature in simulink. To get the job done the advanced Matlab feature, Callback-functions has been used.
To do so, first the whole, undistributed mask is applied to the control block. After that, the command *set_param* is used to set the callbacks.

```
set_param(gcb,'MaskCallbacks',{'mpc_call1',…,'mpc_call4','',…,''});
```

This command sets the parameter *MaskCallbacks* of the current block (gcb = get current block). The Callbacks are defined in separate files called {mpc_call1...mpc_call4} and are called whenever the first four options in the mask are changed. Because these options are the check buttons of the four masks these files describe the commands that are to be executed when a button is pushed (set the check-mark to that option, remove the previous check-mark, hide all options that are not categorised into that mask, and fill in the known values for the visible options).

To demonstrate the callback function, an example is now presented.
First determine the visibilities, of course the first four mask options are 'on' (visible) in all four masks, there these options represent the buttons which should be visible at all time.

```
set_param(gcb,'MaskVisibilities',{'on','on','on',…,'off','off',…,'off'});
```

To '*remember*' and fill in the known values at the correct places of the visible options, all variables are called from the current block (*gcb*).

```
variable_name1=get_param(gcb,'variable_name1');
variable_name2=get_param(gcb,'variable_name2');
…
variable_namen=get_param(gcb,'variable_namen');
```

After recalling all variables, the known values are filled in at the correct places. Besides the recalled variables the first four options are set manual. Only the current called button is set to 'on' (checked) all other buttons are set to 'off' (not checked).

```
set_param(gcb,'MaskValues',{'on','off','off','off',variable_name1,...
                           ...variable_name2,…,variable_namen});
```

3.4 Simulation results

3.4.1 Reference results

To check the designed MPC controller s-function, the following experiment has been conducted. The (original) available simulation script is executed within Matlab with the model of the cross-linked water tanks with certain parameter settings. The same model and parameters are applied on the designed MPC s-function. If the s-function works correctly, both responses should be exactly the same.

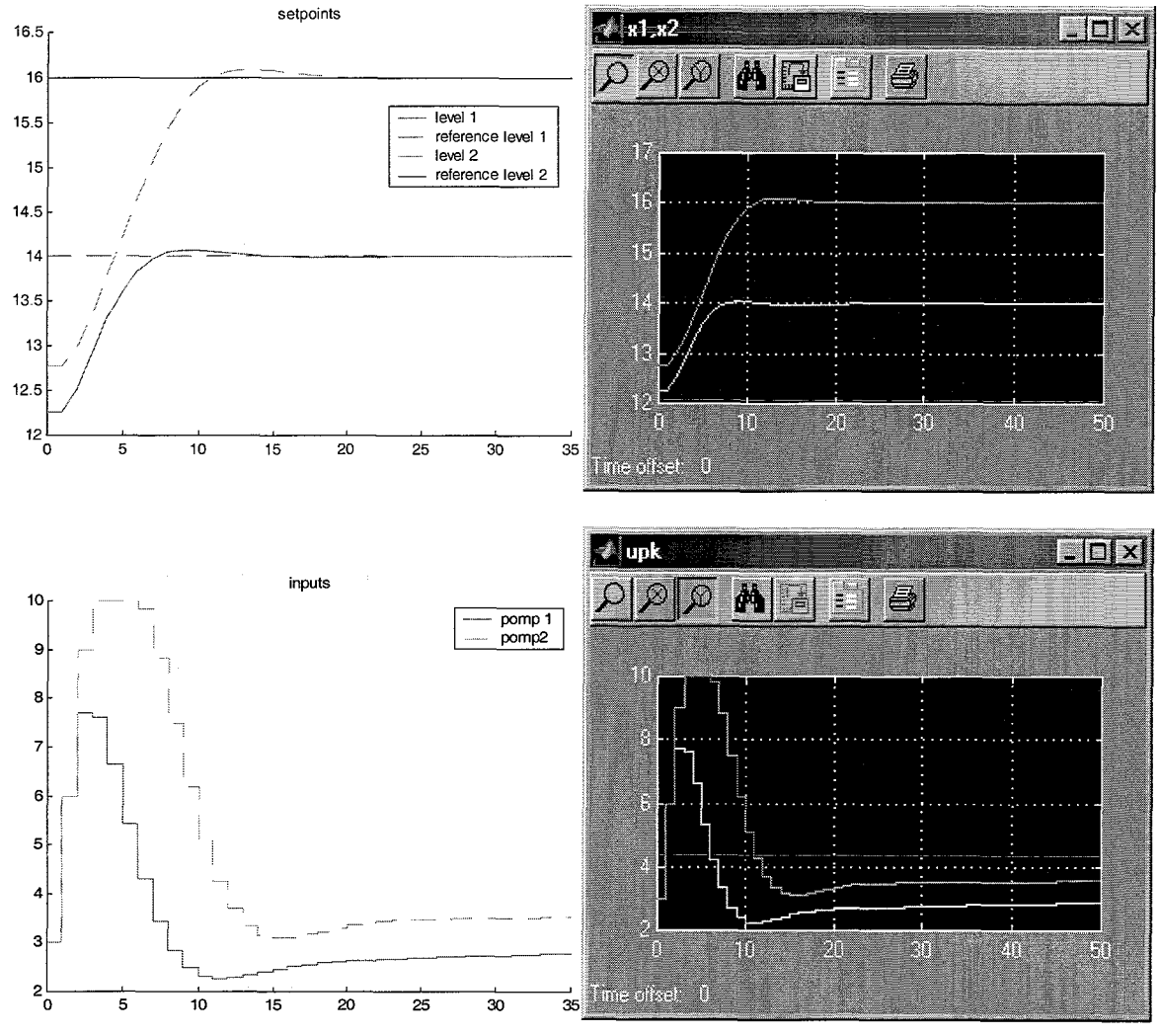


Figure 3.5: *Verification check (controlled states of simulation script (upper left) and s-function (upper right); calculated inputs of simulation script (lower left) and s-function (lower right)*

As can be seen in figure 3.5 (*Verification check)* the both responses are exactly the same. Numerical both the states as the inputs are identical. This is a confirmation of the correctness of the MPC controller s-function.

3.4.2 Verification results

At this point the genericity will be checked. The main goal, designing a real-time implementation of a generic Model Predictive controller, contains the word generic. This means the controller must not contain any model specific elements.
To check this a second model with a different number of inputs was implemented (for procedure to implement a new system, see Appendix A). This model of the hanging crane contains 4 states and one input (driving force of the crane), where the cross-linked water tanks contain 2 inputs (2 controlled pumps). Next the simulation results of this system are presented.

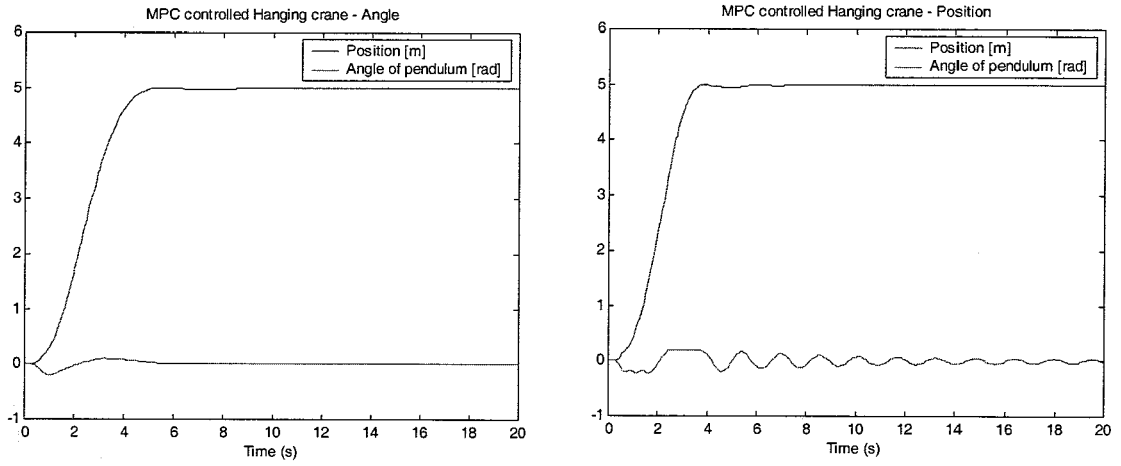


Figure 3.6: *MPC Controlled hanging crane with an output weighting of 100 on the angle and 10 on the position (left); output weighting of 10 on the angle and 100 on the position (right).*

Above the principle of output weighting is demonstrated. It's clear that in the left situation the angle is very well controlled but the set position is reached slower in comparison with the right situation.
To demonstrate another tuning parameter, take a look at the next example in which the prediction horizon is varied from 8 seconds (32 samples) to 1.25` second (5 samples) while the control horizon is kept at a constant 1 second (4 samples).
It might be clear that decreasing the prediction horizon with respect to the control horizon, results in a more aggressive controller, although as stated earlier also a less robust controller.

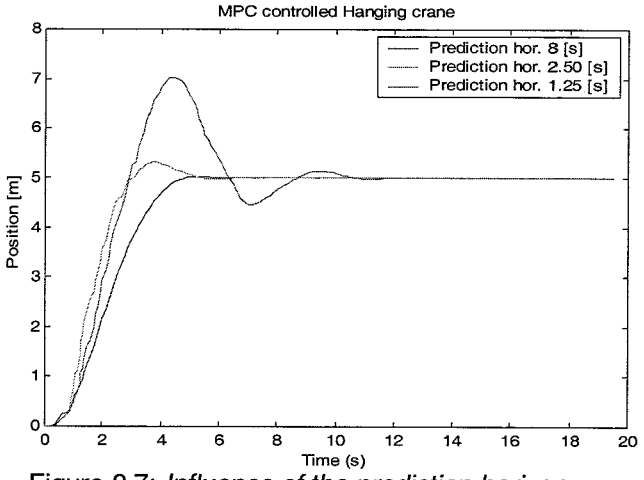


Figure 3.7: *Influence of the prediction horizon*

4 Real-time implementation

As stated before the main goal of this assignment was to design a real-time implementation of a generic MPC controller. In the previous section a description was given of the intermediate stage, a Matlab/Simulink s-function. This s-function should be compiled or rewritten to obtain a code that is executable on a real-time platform.

4.1 RTW model

A first test was conducted to see if the controller itself, including all called functions (e.g. prediction models, optimisation routine) were compilable with the Real-Time Workshop (RTW) tool within Simulink. To prevent fatal errors during the compilation, the combination of continuous and discrete systems in the same model was excluded. In practice the Process model would be replaced by a block set that is supplied with the real-time platform (e.g. dSpace). In this test case the MPC controller was used with a prediction model of the cross-linked water tanks (See figure 3.1: *Cross-linked water tanks*). In stead of the *difficult* implementation with the communication blocks, a fake system was set up. This means that the controller is fed with a constant *fake measurement* with appropriate sizes. Remember that this is just a test to see if the controller is compilable.

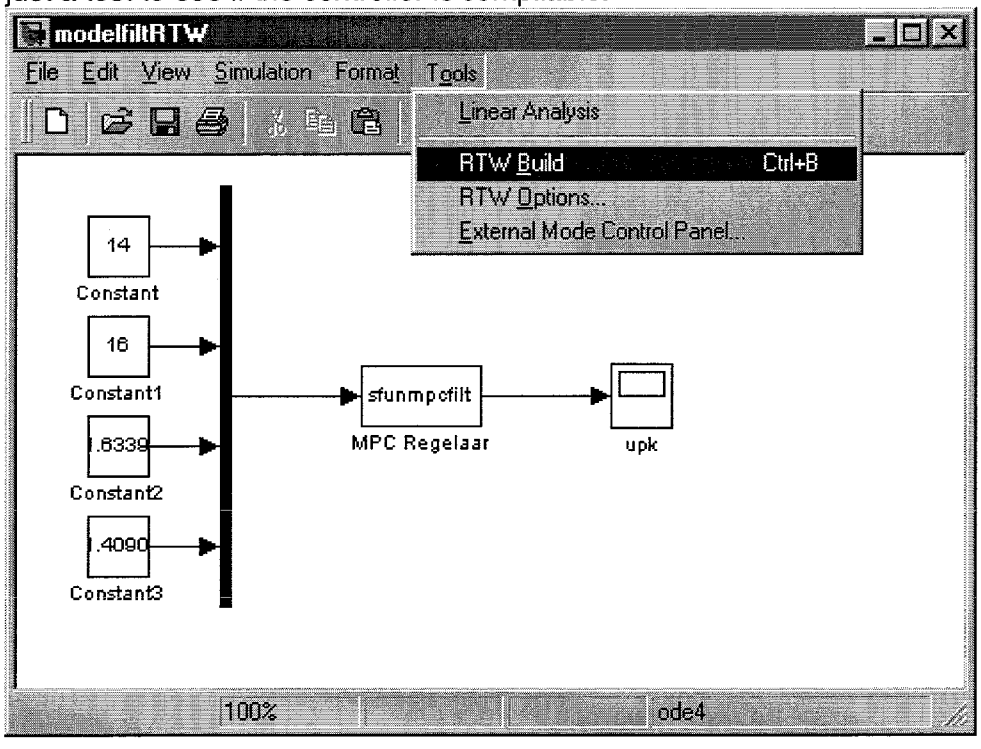


Figure 4.1: *Simulink RTW test model with fake system*

If this model is build by RTW an error prompts which says the s-function should be remapped from Matlab format into C format. No extended conclusions can be drawn from this. Due to time lack no further actions are taken to fulfil the remapping of the controller s-function and its subroutines from Matlab to C format.

5 Conclusions

Designing a real-time implementation of a generic Model Predictive Controller appeared to be a complex assignment with many aspects on several fields. First knowledge had to be obtained on the topic of MPC controllers and comparing the available MPC simulation script with other MPC controllers in literature. It seemed that the available controller was well structured and easy extendible with new features.
To get to the point where the controller could be executed on a real-time platform an intermediate stage was introduced. The available simulation script was implemented in a Matlab/Simulink s-function with a user-friendly (and extendible) interface. Future versions can be extended with a more sophisticated mask. For example a mask which checks the given inputs with the sizes of the used system. Such a s-function can be compiled under certain conditions with tools like Real-Time Workshop (RTW) into code for real-time processors, with respect to the used commands in the s-function. The controller was not only tested on the, in the simulation script supplied model of the cross-linked water tanks, but also on a second implemented model of the hanging crane. This model has a different number of states and inputs so the genericity of the controller could be tested.
At this point a correctly working s-function of the generic Linear Model Predictive Controller was available. Due to time lack in this traineeship the next step to real-time implementation was not completed. A test was done with a fake system to see if the RTW was able to compile the controller. It is recommendable for future developments that the available s-function in Matlab format is rewritten in C/C++ to be executable on a real-time platform. Besides that point of course new features should be tested on the MPC controller.
In my opinion the ever moving boundaries of MPC applications should be explored continuously. The increasing computational speed of computers implies an increasing number of applications for MPC controllers. It seems to be a very powerful class of controllers in which a lot of work still needs to be done.

References

Van Essen, H.A.; Advanced control (lecture notes), Eindhoven University of Technology, 2000.

Franklin, G.F.; Powell, J.D.; Emami-Naeini, A.; Feedback control of dynamic systems (3rd Edition), Addison Wesley, 1994.

Morari, M.; Garcia, C.E.; Lee, J.H.; Prett, D.M.; Model predictive control, Prentice Hall, pre-print, 1993a.

Morari, M.; Ricker, N.L.; Model Predictive Control toolbox, Matlab functions for the analysis and design of Model Predictive Control systems, The Mathworks, 1993b.

Oliveira, Numo M.C. de; Biegler, Lorenz T.; Constraint handling and stability properties of model predictive control, AIChE Journal, Vol. 40, p. 1138-1155, 1994.

Appendix A Procedure to configure the controller

To get the generic multimask MPC controller up-and-running, a few settings are to be configured. This procedure is based on the simulation case. To setup a working simulink model the active directory should contain the files

- *mpc_callX.m* (4 callback functions)
- *mpcset.m (callback activator function)*
- *qpmat.m (QP argument builder for linear case)*
- *qpmatnl.m (QP argument builder for non-linear case)*
- *sfunmpcX (Generic Multimask controller block)*

In addition to these supplied files, the user has to design the (non-linear) process model in s-function format. For example:

```
function [sys,x0,str,Ts] = sprocess(t,xpk,upk,flag)



% non-linear equations:
xdot1 = -a1/A1*sqrt(2*g*xpk(1)) + a3/A1*sqrt(2*g*xpk(3))
xdot2 = -a2/A2*sqrt(2*g*xpk(2)) + cos(xpk(1) + gam2*k2/A2*upk(2);
xdot3 = -sin(xpk(1)*xpk(3)) + (1-gam2)*k2/A3*upk(2);
xdot4 = -a4/A4*sqrt(2*g*xpk(4)) + (1-gam1)*k1/A4*upk(1);

sys=[xdot1;xdot2;xdot3;xdot4];

elseif flag == 0
   sys = [4 0 4 2 0 0 1 ];

% This means 4 continuous states, 0 discrete states, 4 output-channels, 2
input-channels, reserved parameter (always 0), no direct feedthrough, 1 row
of sample times

   x0=[12.2630;12.7832;1.6339;1.4090]; % initial states of the process

   str='';
   Ts= [0 0]; % continuous system


elseif flag == 3
       sys = [xpk(1);xpk(2);xpk(3);xpk(4)];



else
    sys = [];

end

% End of (non-linear) process s-function
```

This s-function can be executed in a simulink model by adding the standard simulink block called 's-function' from the library and set the filename of the designed s-function.

After this process model a prediction model has to be implemented in case a non-linear prediction is wanted. The prediction model is implemented in a separate file, like the example of the cross-linked water tanks on the next page. The filename has to be set in the system-mask.

```
function xdot=waterbak(t,xpk,opt,upk)

% non-linear prediction model of cross-linked water tanks

% parameters
A1=28;A2=32;A3=28;A4=32;
a1=0.071;a2=0.057;a3=0.071;a4=0.057;
g=981; gam1=0.7; gam2=0.6;
k1=3.33;k2=3.35;

for i=1:4
   if xpk(i)<0
   xpk(i)=0;
   end
end


% non-linear equations:
xdot(1) = -a1/A1*sqrt(2*g*xpk(1)) + a3/A1*sqrt(2*g*xpk(3)) +
gam1*k1/A1*upk(1);
xdot(2) = -a2/A2*sqrt(2*g*xpk(2)) + a4/A2*sqrt(2*g*xpk(4)) +
gam2*k2/A2*upk(2);
xdot(3) = -a3/A3*sqrt(2*g*xpk(3)) + (1-gam2)*k2/A3*upk(2);
xdot(4) = -a4/A4*sqrt(2*g*xpk(4)) + (1-gam1)*k1/A4*upk(1);

xdot=xdot';
```

To set the prediction matrices the supplied function *predmat.m* has to be adapted to the wanted prediction model. In the supplied file an example is already given. The example model has to be replaced with the wanted prediction model. The header and rest of the body must not be changed.

To operate the controller at a certain setpoint a setpoint generating function has to be defined. An example can be found in *wbsetpoint.m*. This file has to be saved in a separate file. The filename can be set in the system-mask.

If all files are in place, the controller is ready to receive its settings. The user has to open the multimask by clicking the controller block in the simulink model. If the correct parameters (with correct sizes) are filled in the masks from top to bottom, the controller is ready to run.

Appendix B Procedure to append arguments to controller

Although the mask has been designed to prevent messing around in the code, sometimes the arguments of the mask and controller have to be extended. For example if a new feature is appended.

1. Implement the new feature in the s-function script *sfunmpcX.m*
   Make sure all needed values of the arguments are present in the script

2. Add the new arguments in the header's list of arguments

3. Now select the controller element in the simulink model and press ctrl-m to open the mask dialog

4. Add all new elements in the mask (see Simulink manual for details)

5. Select the simulink option *Look under mask* to append new elements to the simulink argument array

6. Edit the callback functions *mpc_callX.m* and *mpcset.m* in the editor window and append the new arguments in the correct order of the mask. Notice that all arguments, both visible and invisible are to be added in all masks

7. Select the controller element in the Simulink system and execute the *mpcset.m* script to assign the correct callback functions to the mask arguments

8. Run the Simulink system and assign values of correct size to the new options in the multimask. Press *apply* and *ok* to check if values are set correctly

9. If all values are correct and the needed files (see appendix A) are in place the system is ready to run. Remove the initially set values of the parameters from the *sfunmpcX.m* controller function to make the controller calculate with the correct parameters.