Adaptive Dynamic Matrix Control for a Multivariable Training Plant

Isabel Remígio Ferrão Guiamba

Submitted in partial fulfillment of the academic requirements for the degree of Master of Science in Engineering in the School of Chemical Engineering at University of Natal, Durban

Durban
November 28, 2001

Abstract

Dynamic Matrix Control (DMC) has proven to be a powerful tool for optimal regulation of chemical processes under constrained conditions. The internal model of this predictive controller is based on step response measurements at an average operating point. As the process moves away from this point, however, control becomes sub-optimal due to process non-linearity. If DMC is made adaptive, it can be expected to perform well even in the presence of uncertainties, non-linearities and time-varying process parameters.

This project examines modelling and control issues for a complex multivariable industrial operator training plant, and develops and applies a method for adapting the controller on-line to account for non-linearity. A two-input/two-output sub-system of the Training Plant was considered. A special technique had to be developed to deal with the integrating nature of this system - that is, its production of ramp outputs for step inputs.

The project included the commissioning of the process equipment and the addition of instrumentation and interfacing to a SCADA system which has been developed in the School of Chemical Engineering.

To
Margareth, Fred and Joaquim

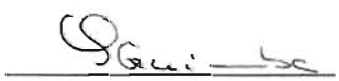
PREFACE

This dissertation is submitted as a component of a part-coursework MSc. Eng. in the School of Chemical Engineering, under the supervision of Professor Michael Mulholland.

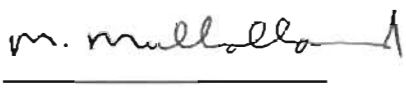
The coursework component was made up of the following subjects.

- Process Dynamics and Control [DNC4DC1]
- Real Time Process Data Analysis [DNC5RTM]
- Applied Control Theory [DNC5CTM]
- Linear Multivariable Control [DNL5NCM]

I certify that all of the work in this dissertation is my own work except where otherwise indicated and referenced.



Isabel Guiamba
November 28, 2001



Prof. Michael Mulholland
Supervisor
November 28, 2001

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude and appreciation to all people who directly or indirectly contributed to this research by giving some of their precious time and advice.

A special word is due to:

My supervisor Professor Michael Mulholland for his many contribution in this investigation, and for his encouragement and final proof-reading of this dissertation. The assistance provided in extending my understanding of this subject is much appreciated.

My children Margareth and Fred who patiently suffered during the past two years of my studies, being far away from their mother, and my son Joaquim for being wonderful company in this very difficult period.

My family, my husband and friends for their understanding, support and encouragement.

To the Chemical Engineering workshop staff for their assistance in the commissioning of the equipment and to Dudley Naidoo for fixing the electronic problems.

The financial support of this research from the Eduardo Mondlane University of Mozambique and from the Natal University Research Fund (NURF) is gratefully acknowledged.

Contents

List of Figures

Page

Chapter 6

Chapter 7

List of Symbols and Acronyms

$\mathbf{A}$	Matrix of model coefficients
A/D	Analogue to digital
ADMC	Adaptive Dynamic Matrix Control
A_{Hi}	Heat transfer area in heater Hi
A_i	Tank i area
a_{Ki}	Pump Ki coefficient
$\mathbf{B}$	Dynamic Matrix
$\mathbf{B}^*$	Reduced dynamic matrix (non-square $P \times N$ matrix)
$\mathbf{B}_{OL}$	Open loop matrix
$\mathbf{B}_0$	Offset matrix
b_{Ki}	Pump Ki coefficient
CL	Closed loop
Cp	Heat capacity at constant pressure
CV	Controlled variable
CVij	Control valve ij
CVi MVj	Controlled variable i and manipulated variable j
$\mathbf{C}_t$	Observation matrix from DAE system
C_{Cvij}	Control Valve ij size coefficient
c_{Ki}	Pump Ki coefficient
DAS	Differential-algebraic system
DAE	Differential and Algebraic Equation
DMC	Dynamic Matrix Control
D/A	Digital to analogue

EKF	Extended Kalman filter
e_{CL}	Closed loop error
e_M	Model error
e_{OL}	Open loop error
F_{ij}	Volumetric flow rate of stream ij
G	Observation matrix
h_{0i}	Height above ground level
hP_{ij}	Junction height above ground level and Pij pressure point
K	Kalman filter gain
kL_{ij}	Pipe resistance
Li	Tank i level
L_{ij}	Length of pipe ij
LDMC	Linear Dynamic Matrix Control
M	Steady state horizon (number of steps)
$\mathbf{M}_i$	Filter covariance matrix
MPC	Model Predictive Control
MRAC	Model reference adaptive control
MV	Manipulated variable
m	Number of inputs
N	Number of future control moves
n	Number of outputs
ODE	Ordinary differential equation
P	Optimisation horizon
P_{ij}	Pressure
p	Parameter vector
pr	Reduced parameter vector

$\mathbf{Q}$	Prediction error covariance matrix
QDMC	Quadratic Dynamic Matrix Control
q_{Hi}	Heat transfer rate for heater Hi
q_{HiHj}	Heat transfer rate for heaters Hi and Hj
$\mathbf{R}$	Observation error covariance matrix
Reg	Regularised model
RLS	Recursive least squares
SVij	Solenoid valve ij
SCADA	Supervisory Control and Data Acquisition
T_{ij}	Temperature
T_{amb}	Ambient temperature
t	Time
U	Overall heat transfer coefficient
$\mathbf{u}$	Input (or manipulated) variable
Vj	Valve j
V_i	Tank i volume
W	Process variable weighting matrix
$\hat{w}_t$	Observation vector from DAEs system
X_{ij}	Valve stem position (fraction of wide open)
$\mathbf{x}$	output vector
$\mathbf{x}_{CL}$	Closed-loop response
$\mathbf{x}_{MEAS}$	Measured output
$\mathbf{x}_{OL}$	"Open-loop" response
$\mathbf{x}_{PRED}$	Predicted output
$\mathbf{x}_{SP}$	Process variable set point
$\mathbf{y}$	State vector
$\hat{\mathbf{y}}$	Observation vector
$\mathbf{z}$	Associated variable

Greek letters

ΔP_{Ki}	Pressure drop over the pump
ΔP_{CVij}	Pressure drop over the valve ij
Δm^*	Limited sequence of N moves to be optimised
Δm_{PAST}	Vector of past inputs -M steps before present time
$\Delta m'_{PAST}$	Vector of past inputs between -2M and -M steps before present time
$\Delta m^{\cdot}_{PAST}$	Vector of past inputs with accumulated moves
Δt	Sampling time
α	Filter constant
β	Coefficient in gradient feedback
δ	Process noise
Λ	Move suppression
μ	Measurement noise
ρ	Density

Chapter 1

Introduction

1.1 Background to the work presented

Dynamic Matrix Control is one of the most successful model predictive schemes and quite popular within the process industry. Control design methods based on the Model Predictive Control concept have found wide acceptance in industrial applications because of their ability to handle process interactions, their handling of unusual dynamic responses and because they do not necessarily demand a rigorous model derived from first principles.

Dynamic Matrix Control is based on a linearised step response model called the *convolution model*. This dynamic model with its associated uncertainties is used to predict and optimise process performance. However, as the system dynamics change with time, a large mismatch can develop between the model and the process and it no longer reflects the actual system. Under these conditions the controller performance deteriorates, which may even destabilise the process. In these situations it becomes necessary to re-evaluate the process model. One way of doing this is to include an adaptive algorithm within the Dynamic Matrix Control algorithm. This strategy can allow adjustment of controller settings with changes in the operating point and better performance is expected even in the presence of time-varying process gain and process nonlinearity. Thus, the development of a method for closed-loop online identification for updating the dynamic matrix is proposed in this work.

Despite Dynamic Matrix Control presenting several novel features and advantages, it is not designed to deal with processes with an integrating nature. These are processes that produce a ramp change in the output for a step change in the input. The Training Plant currently under study exhibits this behaviour, and is also non-linear, multivariable and highly interactive. Thus, an approach for using Dynamic Matrix Control to control integrating processes was also considered. In this method the slope of the predicted response is determined from the slope of the output trajectory between the current and the previous control instants.

However, because of the complexity of the Training Plant, with this work being the first investigation with regard to it, only a 2-input / 2-output sub-system of the plant was studied in the present project. Thus, the proposed algorithm was applied only to this sub-system. For a better understanding of the software, preliminary simulation was in some cases done on a *second* pilot plant, a 2-input / 2-output, Pump-tank system, also available in the School of Chemical Engineering.

1.2 Thesis Layout

As an introduction, Chapter 1 is devoted to provide an overview of the thesis and aims of the present work.

Chapter 2 is designed to familiarise the reader with the Training Plant under study. With this goal in mind, attention has been given to a description of the system and presentation of the control problem emanating from this process.

Chapter 3 deals with the theoretical process model and provides a more comprehensive understanding of the Training Plant behaviour described by a non-linear system of mixed differential and algebraic equations. State-space step responses obtained from estimated data when running the extended Kalman filter simulation program written in Matlab[1], are also presented in this chapter.

The Dynamic Matrix Control technique and its applications are considered in Chapter 4. Adaptive Control principles are covered in Chapter 5.

Chapters 6 and 7 deal with the Adaptive Dynamic Matrix Control scheme with particular attention to an integrating system in Chapter 7. Results obtained from off-line tests, as well as actual application of the Integrating ADMC controller in the 2-input / 2-output sub-system of the Training Plant are also included.

1.3 The aim of the project

The aim of this study was to investigate different aspects of control of a Training Plant available in the School of Chemical Engineering laboratory at University of Natal. This multivariable system presents eight manipulated input variables and sixteen output variables of interest from the point of view of process control. A mathematical model derived from first principles revealed complex behaviour of the system with high order, non-linearity and a high level of interaction between variables. To overcome these difficulties, possibilities of applying a Model Predictive Control technique in this Training Plant had to be considered.

To fulfil the objective of this work, general tasks were proposed as follows:

- Recommissioning of the equipment,
- Addition of digital control to already existing analogue instrumentation,
- Interfacing of the system to a computer SCADA system, comprising software for real-time simulation and data acquisition developed in the School of Chemical Engineering,
- Development of the control algorithm
- Application of the control algorithm on the equipment

[1] Matlab is a registered trademark of The MathWorks, Inc.

Chapter 2

The Training Plant

2.1 Introduction

The Training Plant currently under study is a complex multivariable, non-linear and highly interactive industrial teaching facility comprising full-scale equipment. It is very useful for a diversity of studies in the field of mass and heat transfer as well as process control. As will be seen, a model will be developed for the entire plant, but only a sub-system of three tanks will be used in the application of the proposed adaptive control algorithm.

This Training Plant allows digital control of variables such as temperature, level, flow and pressure, since the old analogue instrumentation of the equipment was converted to a digital format in the course of this study, by introducing *analogue-to-digital* (A/D) and *digital-to-analogue* (D/A) converters. Thus, with digital processing the computer is able to receive the measurements directly from the process and based on the control law, resident in its memory, calculate the values of the manipulated variables. The decisions are then directly implemented in the process by the computer via the final control element.

Some advantages of digital controllers over their analogue are that high-speed digital systems have provided the means to produce low cost and accurate controllers. More complex algorithms can be implemented using digital controllers, and with greater accuracy than the corresponding analogue systems. Furthermore, there is greater flexibility in digital system since alterations to the control algorithm can be performed in software rather than through changing or tuning discrete components which are prone to drift.

The Training Plant is part of the available equipment in the Chemical Engineering Laboratory at the University of Natal and some pictures of it are presented in Figure 2.1.

Figure 2.1 Parts of the Training Plant rig

To fulfil some of the objectives presented in Chapter 1, tasks such as re-labeling, calibration, addition and repair of the equipment and control instrumentation, including improvement of piping / tubing were also taken into account. In a number of cases, new sensors were installed in parallel to the existing sensors (e.g. level transmitters and temperature transmitters).

A description of the equipment is presented in this chapter in Section 2.2. Section 2.3 provides information about data acquisition and control. The control problem emanating from this process is presented in section 2.4. Finally, a 2-input / 2-output sub-system from the Training Plant was considered in the present study; is presented in section 2.5.

2.2 System description

The experimental setup is schematically presented in Figure 2.2. The system can be divided in two parts involving mass and heat balancing as shown below. However, it is important to notice that sub-systems can be defined and simulated accordingly with fewer variables of interest, by manipulating limited manual and / or pneumatic control valves in the pipes connecting the equipment.

Mass balancing

In this part, the Training Plant consists of a mixing tank, Tank T102 (also called Tank 2), from which warm water is pumped by centrifugal pump, K2, to four different parts of the system as follows:

- Tank 1, T101, with its base at ground level, the same level as Tank 2. A butterfly pneumatic control valve, CV01, connects both tanks. The outlet stream from this tank is pumped by small centrifugal pump, K1 and returned to Tank T102 through pneumatic control valve CV11.

- Counter-current gas / liquid absorption column, C105, 0.5 m in diameter and 2.5 m in height at overhead level, filled with ceramic raschig rings. The column bottom is 2.7 m above the ground floor and is connected to Tank T102 by pneumatic control valve CV05. The output flow from the column is returned to Tank 2 by gravity.

- Upper storage Tank 3, T103, with base at 4.75 m above the ground floor. This stream is controlled by pneumatic control valve CV03, while pneumatic control valve CV13 controls the outlet stream from Tank T103 to Tank 2 which, is also returned by gravity.

- The flow is finally pumped to a co-current heat exchanger, H1, where it is heated and returned to Tank T102 through pneumatic control valve CV02.

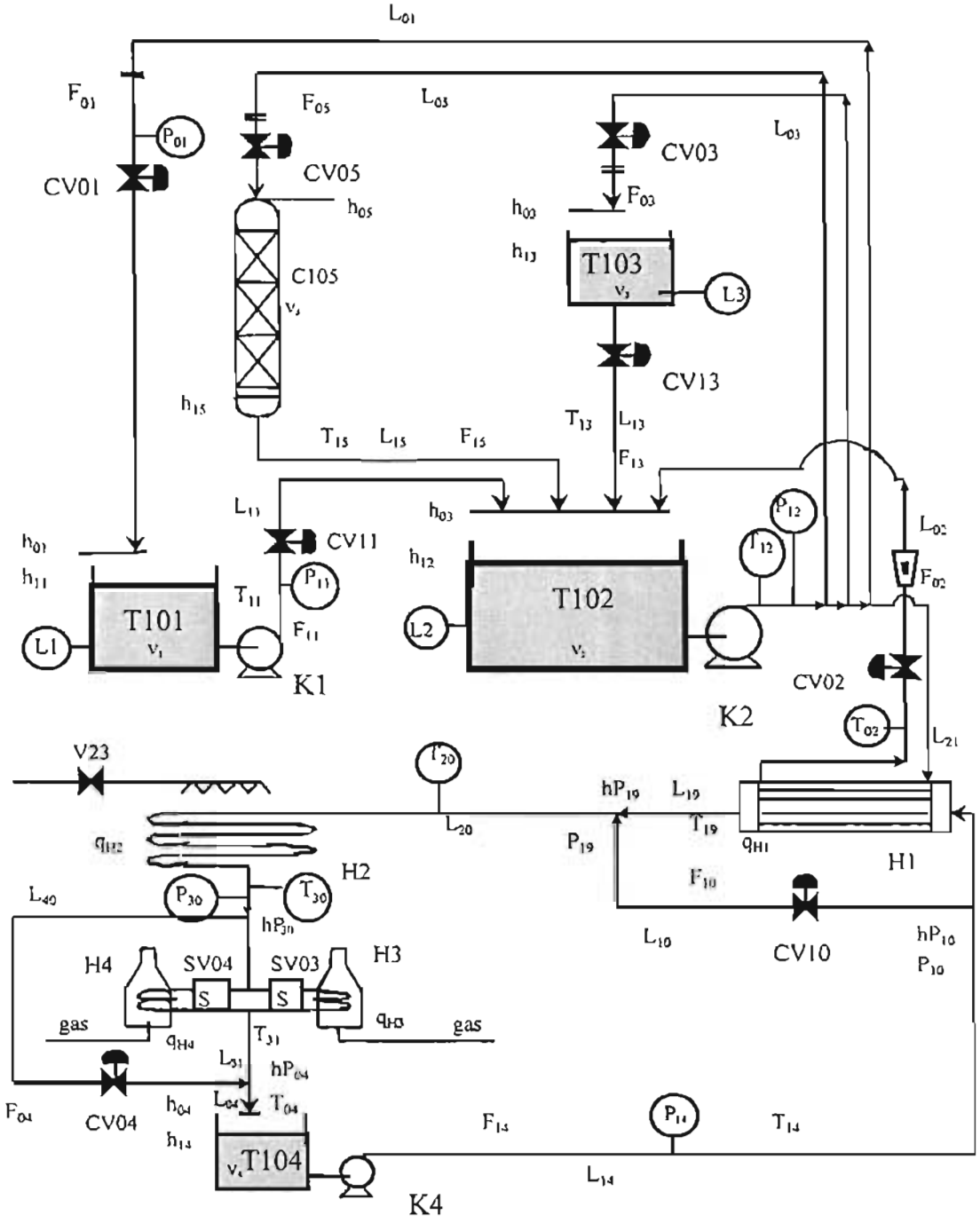


Figure 2.2 The Training Plant diagram

Heat balancing

Hot water is pumped by small centrifugal pump, K4, from Tank T104, and used in the above-mentioned co-current Heat Exchanger, H1, to heat the incoming flow from Tank T102. However, just before H1, the hot stream is by-passed, passing through pneumatic control valve CV10, and then mixed with the output flow from the heat exchanger, continuing then to a counter-current cooler, H2, where the temperature is dropped. Part of the outlet stream from the cooler enters the gas heaters, H3 and H4 where it is heated and then mixed with the by-passed cooler output flow and returned to Tank 4, T104, just after pneumatic control valve, CV04. Because the resistance of the piping through H3 and H4 is quite high, closing CV04 has the effect of increasing flow through H3 and H4 from virtually very low to full flow. Since the heater capacities are much larger than the cooler, shutting CV04 has the effect of passing from a net cooling mode to a net heating mode.

2.3 Data acquisition and Control

The rig is interfaced to a local PC via Eagle® PC30 comprising *analogue-to-digital* (A/D), and *digital-to-analogue* (D/A), input / output boards, with 16 12-bit inputs, 4 outputs and 24 programmable digital I/O. The experimental as well as simulation studies were carried out using a flexible Supervisory Control and Data Acquisition System (SCADA). Extensive configuration options are provided in this software, including Linear Dynamic Matrix Control (LDMC), constraints and tuning parameters, PID loop settings, measurement filtering, alarm levels and logging selections which may be changed on-line. In addition, an on-board *convolution model*[1] may be used to test algorithms. A version of this program with real-time graphing and plant mimic features currently exists in Microsoft® Visual C++ and was written for Windows 95. Previous research students, including Prosser [1998], have developed this software.

The evolution of this software has been fairly organic, comprising many modifications by various researchers as the need arose. Modifications were also made in the present work since an integrating adaptive algorithm was added to the existing Dynamic Matrix Control algorithm to overcome the problem of non-linearity and the integrating nature of the Training Plant (see Appendix E). Results of off-line and real-time simulations using a step-response model applied to an Integrating Adaptive Dynamic Matrix Control for a 2-input / 2-output sub-system defined on the Training Plant are presented in Chapter 7.

[1]The convolution model is a model based on linear combination of time-domain step response

2.4 The Control Problem

As already mentioned, this system is multivariable with a high level of interaction. A *multivariable process* is one with multiple inputs, u_1, u_2, ..., u_m, and / or multiple outputs, x_1, x_2, ..., x_n, where *m* is not necessarily equal to *n*.

One of the consequences of having several input and output variables is that such a control system can be configured in several different ways using traditional SISO (single input / single output) control loops depending on which input variable is used to control which output variable. This is referred to as the input / output pairing problem.

In order to be able to control the output variables arbitrarily, we need at least an equal number of input variables, and the choice of the best pairing to minimize the interaction and provide optimum control is very important. This does not mean that the control is "perfect" or even very good; it simply means that it is the best it can be in such a SISO format.

For non-square systems that is, a multivariable systems with unequal number of input and output variables, the most obvious problem is that after input / output pairing, there will always be a residual of unpaired input or output variables, depending on which of these are in excess.

For under-defined systems (with fewer input than output variables), not all the outputs can be controlled, since we do not have enough input variables. The strategy for loop pairing of such systems, is to choose a square sub-system by dropping off the excess number of output variables on the basis of economic importance [Ogunnaike and Ray, 1994]. As illustrated in Figure 2.3, the present Training Plant is an under-defined system from the point of view of control, with eight manipulated input variables (pneumatic control valves) and sixteen output variables of interest. The figure shows the anticipated main influences of the inputs on the outputs.

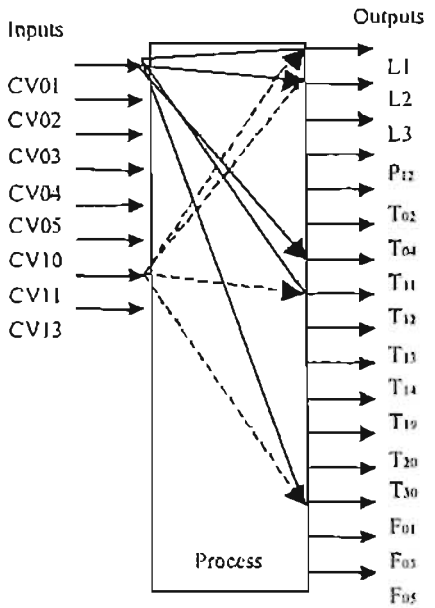


Figure 2.3 The Training Plant control problem structure

The real challenge in deciding on loop pairing for non-square systems is presented by over-defined systems (with more inputs than output variables), where arbitrary control of the fewer output variables can be achieved in more than one way. The Relative Gain Array (RGA) the most discussed index of loop interaction that suggests input / output pairings for which the interaction effects are minimized, can in this case also be used to define the best square sub-system. Further information about this issue can be found in Luyben, [1990] (pgs. 576 – 579) and Ogunnaike and Ray, [1994] (pgs. 728 –758).

According to Johnson, [1993], optimum control (control quality) can be defined in terms of the three effects resulting from a load or set point change as follows:

- Stability,
- Minimum deviation from set point and
- Minimum duration (time necessary to control)

In the present work the intention is to formulate a control strategy for the process viewed as a lumped multivariable system, in this way avoiding the pairing issue and allowing coordinated action to achieve control objectives within defined constraints. Thus, only a sub-system (described in the following section) with two inputs (CV01, CV03) and two outputs (L1, L2), will be considered.

2.5 The sub-system under study in the present work

As mentioned above in section 2.2, the Training Plant presented in Figure 2.2 allows the definition of the sub-system of interest, by manipulating some control valves in the equipment. Thus, to understand some aspects of the control of this system, a 2-input / 2-output sub-system was configured as follows:

Controlled variables were chosen as the levels in the tanks T101 and T102 while manipulated variables were valves CV01 (inflow from T102 to T101) and CV03 (inflow from T102 to T103). Constant positions as percentage open were set for valves CV11 (inflow from T101 to T102) and CV13 (inflow from T103 to T102). A simplified diagram is shown in Figure 2.4

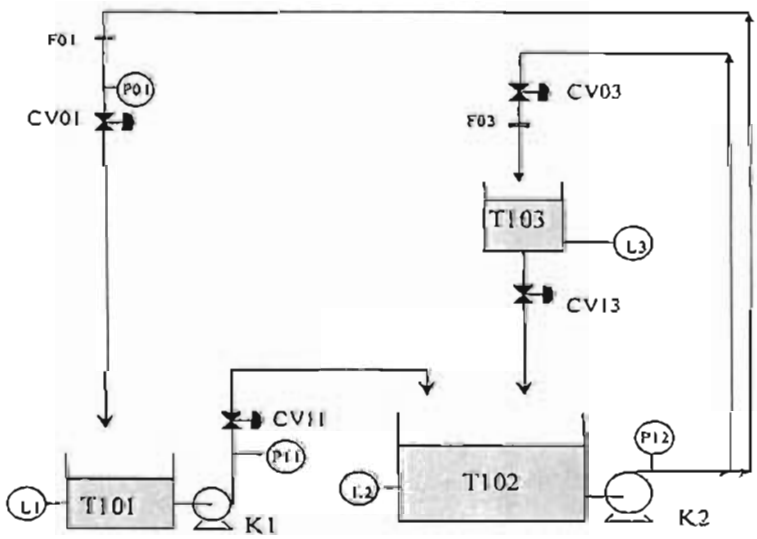


Figure 2.4 Sub-system of Training Plant for experiments

To understand this sub-system behavior, a 2x2 off-line step responses of it, were derived from the estimated outputs using the extended Kalman filter software applied to a state-space model discussed (Chapter 3). For process control purpose, the step responses were experimentally obtained and actually applied in the Dynamic Matrix Controller for off-line simulation as well as for real-time tests as presented in Chapter 7.

It is important to emphasize that, a *second* 2-input / 2-output Pump-tank system, described in section 4.4 and shown in Figure 4.4, was also used in the present work for on-line testing of Linear Dynamic Matrix Control and Adaptive Dynamic Matrix Control in Chapters 4 and 5 respectively.

Chapter 3

The Process Model

3.1 Introduction

A proper understanding of process dynamics significantly facilitates the design of effective controllers. This understanding has usually been shown to be dependent on the availability of a process model. A mathematical model representing a process consists of a system of equations, which represent the relationships between process variables, which purport to describe the behaviour of the physical system. The main use of the model is usually the possibility to investigate the system response under various input conditions rapidly, and inexpensively, without necessarily tampering with the actual physical entity.

With the exception of the most trivial process, it is impossible for a mathematical model to represent exactly all aspects of process behaviour [Ogunnaike and Ray, 1994]. This fact notwithstanding, however, the usefulness of the mathematical model should not be underestimated. We just need to keep its limitations in proper perspective. The effectiveness of any control system designed on the basis of a process model will, of course, depend on the integrity of such a model in representing the process.

To investigate the process behaviour of the present Training Plant, two process models based on different principles were developed. A state-space model formulated from first principles and a step-response model based on the experimental data from the equipment.

An extended Kalman filter method was used for state estimation of the non-linear system described by a class of differential-algebraic equations this was used to develop a state-space model for the entire system shown in section 2.2. In addition, a step response convolution model was obtained using process measurements.

The present chapter is structured as follows. Literature review is presented in section 3.2. Section 3.3 describes the Training Plant behaviour predicted using a state-space model. This section includes issues such as the theoretical model, applied technique for solution of the differential and algebraic equations describing the system, and finally, the step response resulting from the model simulation using a Matlab program.

3.2 Literature review of differential and algebraic equations systems

Differential-algebraic equations (DAEs) arise frequently in chemical engineering and may occur directly from the use of first principles to model a physical phenomenon. The differential equations represent the dynamics of the system and the algebraic equations usually describe the constraints among variables. These constraints may be linear or non-linear.

The DAEs have been studied by a number of investigators and are widely used for the dynamic modelling of chemical processes. Several approaches have been proposed to estimate a solution of DAEs. Cheng, Mongkhonsi and Kershenbaum [1997], employed the minimum least squares criterion, i.e., the minimisation of the integral of the weighted square residual errors in the process model and the measuring device, to develop a sequential algorithm for non-linear differential and algebraic systems with the use of variational calculus.

Byrne and Ponzi [1988] review the underlying methods in some differential-algebraic systems (DAS), software. A Newton-type method in which the computation of the Jacobian matrix is not performed during each iteration is described. Rather, it is computed at some point and retained until the convergence is deemed to be too slow.

Becerra, Roberts and Griffths [2001], explore the application of an extended Kalman filter (EKF), to systems described by non-linear DAE's. A time-varying linearisation has been derived for a DAE model and a simplified square root EKF algorithm has been described. They showed how the EKF can be used for noise filtering and estimation of unmeasured states, including algebraic states of a system described by a semi-explicit index one DAE, i.e., with similar behaviour to sets of ordinary differential equations (ODE), that can be solved using similar solution methods. Notice that the EKF technique has been traditionally applied to state and parameter estimation using models described by ordinary differential equations.

DASs solvers can play a significant role in chemical engineering since, the simultaneous treatment of both differential and algebraic equations in a mixed system eliminates the need for attaching together separate methods and software for ODEs and algebraic equations.

An extended Kalman filter was employed in the present work for state estimation of the system under study, since it provided a means to drive the solution towards convergence, and avoided problems of singularity in under or over-specification.

3.3 Modelling the Training Plant

3.3.1 The state-space model

The initial modelling approach of the existing Training Plant was based on a state-space model which, as described in section A.3.1 (Appendix A), is formulated from first principles. Fundamental laws including mass and energy balances were applied over the equipment and junction points and pressure drops over the pumps, pipes, and pneumatic control valves were also considered.

The main simplifying assumptions that were taken to derive this model were:

- The mixing is perfect
- Heat losses are negligible
- The valves have linear characteristics (but installed characteristics will be non-linear due to pressure variation)
- No mass transfer occurs in the absorption column
- The pump characteristics are described by quadratic functions
- Since only water is used in the system, the specific heats are omitted from energy balances.

The modelling revealed a complex non-linear, multivariable, and high order system with forty-seven equations. Nine were Ordinary Differential Equations (ODEs), describing the state variables and thirty-eight were algebraic equations representing the system constraints as shown below. The system has a total of nine states, ten inputs and thirty-eight associated variables. The equations below should be viewed in conjunction with Figure 2.2.

- **Model differential equations**

Volume balances in tanks and column

$$\frac{dh_{11}}{dt} = \frac{-F_{11} + F_{01}}{A_1} \tag{3.2}$$

$$\frac{dh_{12}}{dt} = \frac{-F_{12} + F_{02} + F_{11} + F_{13} + F_{15}}{A_2} \tag{3.3}$$

$$\frac{dh_{13}}{dt} = \frac{-F_{13} + F_{03}}{A_3} \tag{3.4}$$

$$\frac{dh_{14}}{dt} = \frac{-F_{14} + F_{14}}{A_4} = 0 \qquad \text{(level in Tank 4 is constant)} \tag{3.5}$$

$$\frac{dh_{15}}{dt} = \frac{-F_{15} + F_{05}}{A_5} \tag{3.6}$$

Energy balances in tanks

$$\frac{dT_{11}}{dt} = \frac{-F_{11}T_{11} + F_{01}T_{12} - \left(-F_{11} + F_{01}\right)T_{11}}{A_1 h_{11}} \tag{3.7}$$

$$\frac{dT_{12}}{dt} = \frac{-F_{12}T_{12} + F_{11}T_{11} + F_{13}T_{13} + F_{15}T_{15} + F_{02}T_{02} - \left(-F_{12} + F_{02} + F_{11} + F_{13} + F_{15}\right)T_{12}}{A_2 h_{12}} \tag{3.8}$$

$$\frac{dT_{13}}{dt} = \frac{-F_{13}T_{13} + F_{03}T_{12} - \left(-F_{13} + F_{03}\right)T_{13}}{A_3 h_{13}} \tag{3.9}$$

$$\frac{dT_{14}}{dt} = \frac{F_{14}\left(T_{04} - T_{14}\right)}{A_4 h_{14}} \tag{3.10}$$

$$\frac{dT_{15}}{dt} = \frac{-F_{15}T_{15} + F_{05}T_{12} - \left(-F_{15} + F_{05}\right)T_{15}}{A_5 h_{15}} \tag{3.11}$$

- **Model algebraic equations**

Overall energy balance in Heat Exchanger, H1

$$0 = -F_{02}\left(T_{02} - T_{12}\right) + \left(F_{14} - F_{10}\right)\left(T_{14} - T_{19}\right) \tag{3.12}$$

Heat balance at junction hP_{19}

$$0 = -F_{14}T_{20} + \left(F_{14} - F_{10}\right)T_{19} + F_{10}T_{14} \tag{3.13}$$

Heat transfer in Heat Exchanger H1

$$0 = -q_{H1} + UA_{H1}\left(\left(T_{14} - T_{02}\right) - \left(T_{19} - T_{12}\right)\right) / \ln\left(\left(T_{14} - T_{02}\right) / \left(T_{19} - T_{12}\right)\right) \quad (3.14)$$

Heat to output stream from the heat exchanger H1

$$0 = -q_{H1} + \rho Cp\left(T_{02} - T_{12}\right) F_{02} \quad (3.15)$$

Heat balance in gas heaters

$$0 = -q_{H3H4} + \rho cp\left(T_{31} - T_{30}\right) F_{31} \quad (3.16)$$

Heat balance in cooling coil

$$0 = -q_{H2} + UA_{H2}\left(\left(T_{20} - T_{amb}\right) - \left(T_{30} - T_{amb}\right)\right) / \ln\left(\left(T_{20} - T_{amb}\right) / \left(T_{30} - T_{amb}\right)\right) \quad (3.17)$$

Heat to output stream from cooling coil F_{14}

$$0 = -q_{H2} + \rho cp\left(T_{20} - T_{30}\right) F_{14} \quad (3.18)$$

Geyzer temperature setting

$$0 = -T_{31_{geyzer}} + T_{31} \quad (3.19)$$

Heat balance at junction hP_{04}

$$0 = F_{04} T_{30} + F_{31} T_{31} - F_{14} T_{04} \quad (3.20)$$

Mass balance at split hP_{12}

$$0 = -F_{12} + F_{01} + F_{02} + F_{03} + F_{05} \quad (3.21)$$

Mass balance at geyzer bypass hP_{30}

$$0 = -F_{31} + F_{14} - F_{04} \quad (3.22)$$

Pressure rise over the pumps

$$0 = -\Delta P_{K1} + a_{K1} F_{11}^{\ 2} + b_{K1} F_{11} + c_{K1} \quad (3.23)$$

$$0 = -\Delta P_{K2} + a_{K2} F_{12}^{\ 2} + b_{K2} F_{12} + c_{K2} \quad (3.24)$$

$$0 = -\Delta P_{K4} + a_{K4} F_{14}^{\ 2} + b_{K4} F_{14} + c_{K4} \quad (3.25)$$

Pressure drop over the valves

$$0 = -X_{01}\sqrt{\Delta P_{CV01}} + \frac{\rho^2 F_{01}}{C_{CV01}} \tag{3.26}$$

$$0 = -X_{02}\sqrt{\Delta P_{CV02}} + \frac{\rho^2 F_{02}}{C_{CV02}} \tag{3.27}$$

$$0 = -X_{03}\sqrt{\Delta P_{CV03}} + \frac{\rho^2 F_{03}}{C_{CV03}} \tag{3.28}$$

$$0 = -X_{04}\sqrt{\Delta P_{CV04}} + \frac{\rho^2 F_{04}}{C_{CV04}} \tag{3.29}$$

$$0 = -X_{05}\sqrt{\Delta P_{CV05}} + \frac{\rho^2 F_{05}}{C_{CV05}} \tag{3.30}$$

$$0 = -X_{10}\sqrt{\Delta P_{CV10}} + \frac{\rho^2 F_{10}}{C_{CV10}} \tag{3.31}$$

$$0 = -X_{11}\sqrt{\Delta P_{CV11}} + \frac{\rho^2 F_{11}}{C_{CV11}} \tag{3.32}$$

$$0 = -X_{13}\sqrt{\Delta P_{CV13}} + \frac{\rho^2 F_{13}}{C_{CV13}} \tag{3.33}$$

Pressure drop over the pipes

$$0 = -kL_{01}\ F_{01}^{\ 2} + P_{12} + \rho g\left(hP_{12} - h_{01}\right) - \Delta P_{CV01} \tag{3.34}$$

$$0 = -kL_{02}\ F_{02}^{\ 2} + P_{12} + \rho g\left(hP_{12} - h_{02}\right) - \Delta P_{CV02} \tag{3.35}$$

$$0 = -kL_{03}\ F_{03}^{\ 2} + P_{12} + \rho g\left(hP_{12} - h_{03}\right) - \Delta P_{CV03} \tag{3.36}$$

$$0 = -kL_{04}\ F_{14}^{\ 2} + P_{04} + \rho g\left(hP_{04} - h_{04}\right) \tag{3.37}$$

$$0 = -kL_{05}\ F_{05}^{\ 2} + P_{12} + \rho g\left(hP_{12} - h_{05}\right) - \Delta P_{CV05} \tag{3.38}$$

$$0 = -kL_{10}\ F_{10}^{\ 2} + \left(P_{10} - P_{19}\right) + \rho g\left(hP_{10} - hP_{19}\right) - \Delta P_{CV10} \tag{3.39}$$

$$0 = -kL_{11}\ F_{11}^{\ 2} + \rho g\left(h_{11} - h_{02}\right) - \Delta P_{CV11} + \Delta P_{K1} \tag{3.40}$$

$$0 = -kL_{12}\ F_{12}^{\ 2} - P_{12} + \rho g\left(h_{12} - hP_{12}\right) + \Delta P_{K2} \tag{3.41}$$

$$0 = -kL_{13}\ F_{13}^{\ 2} + \rho g\left(h_{13} - h_{02}\right) - \Delta P_{CV13} \tag{3.42}$$

$$0 = -kL_{14}\ F_{14}^{\ 2} - P_{10} + \rho g\left(h_{14} - hP_{10}\right) + \Delta P_{K4} \tag{3.43}$$

$$0 = -kL_{15}\ F_{15}^{\ 2} + \rho g\left(h_{15} - h_{02}\right) \tag{3.44}$$

$$0 = -kL_{19}\left(F_{14} - F_{10}\right)^2 + \left(P_{10} - P_{19}\right) + \rho g\left(hP_{10} - hP_{19}\right) \tag{3.45}$$

$$0 = -kL_{20}\ F_{14}{}^2 + \left(P_{19} - P_{30}\right) + \rho g\left(hP_{19} - hP_{30}\right) \tag{3.46}$$

$$0 = -kL_{21}\ F_{02}{}^2 + P_{01} + \rho g\left(hP_{01} - h_{02}\right) \tag{3.47}$$

$$0 = -kL_{31}\ F_{31}{}^2 + \left(P_{30} - P_{04}\right) + \rho g\left(hP_{30} - hP_{04}\right) \tag{3.48}$$

$$0 = -kL_{40}\ F_{04}{}^2 + \left(P_{30} - P_{04}\right) + \rho g\left(hP_{30} - hP_{04}\right) - \Delta P_{Cv04} \tag{3.49}$$

where:

A_i tank area, [m^2]

a_{Ki} pump coefficient, [h^2 /m]

b_{Ki} pump coefficient, [h]

c_{Ki} pump coefficient, [m]

Cp heat capacity at constant pressure, [kw/(kg °C)]

C_{Cvij} Control valve ij size coefficient

F_{ij} volumetric flow rate of stream ij, [m^3/h]

h_{1i} tank level, [m]

h_{0i} height level above floor, [m]

hP_{ij} junction height above floor to the Pij pressure point, [m]

kL_{ij} pipe ij resistance, [m water]

q_{Hi} heat transfer rate for heater Hi, [kw/h]

q_{HiHj} heat transfer rate for heaters Hi and Hj, [kw/h]

T_{ij} temperature, [°C]

T_{amb} ambient temperature, [°C]

t time [h]

X_{01} valve stem position (fraction of wide open)

$U\ A_{Hi}$ overall heat transfer coefficient x heat transfer area for heater Hi [kw/ (m^2 °C)][m^2]

ρ density, [kg /m^3]

ΔP_{Ki} pressure drop over the pump, [m water]

ΔP_{Cvij} pressure drop over the valve ij, [m water]

Numerical treatment of this *47 x 47* system of coupled differential and algebraic equations was a complicated task due to the high dimension and non-linearity. Therefore, for solution of this system, an extended Kalman filter algorithm for systems described by non-linear differential-algebraic equations developed by Mulholland [2001] (personal communication), was applied. A perturbation method is used in this technique for local linearisation, i.e., a fixed small fraction

(eg. 0.001) of the defined range for each variable is used as the perturbation. By avoiding the use of analytical derivatives, possible model / derivative mismatches in this large system were prevented. Admittedly, this risk could be reduced somewhat by using a symbolic mathematics package like *Mathematica* or *Maple*.

3.3.2 Comments on the extented Kalman filter

The Kalman filter is a stochastic filter that allows the estimation of the states of the system based on a linear state space model. The extended Kalman filter (EKF) uses local linearisation to extend the scope of the Kalman filter to systems described by non-linear ordinary differential equations, [Maybeck, 1982 in Becerra et al., 2001]. Thus, this scheme has been traditionally applied to state and parameter estimation using models described by ODEs.

As mentioned above, for state estimation of the Training Plant under study using the state-space model, an EKF technique was applied. A detailed description of this method is presented in the Appendix B.

Linearisation of DAEs

By applying a Taylor series expansions truncated after the first order term, the process model is linearised taking into account the DAE nature of the system, described by the following equation:

$$\begin{aligned} \frac{dy}{dt} &= f(y,z) \\ 0 &= g(y,z) \end{aligned} \tag{3.50}$$

where y is a vector of state variables, and z a vector of algebraic variables.

The Jacobian is calculated assuming that **f** and **g** functions are sufficiently differentiable in their arguments so that all needed differentiations are possible. Notice that for local linearisation a *perturbation* method is used in the EKF algorithm. The functions and the Jacobian matrices are re-evaluated at every iteration by perturbing each variable in turn, thus the values of the elements of the matrices change slowly as the process moves to a new operating point. Perturbation techniques find their most fruitful application in the class of non-linear systems [Rice and Do, 1995]. It can be applied to algebraic equations as well as differential equations. Analytical solutions requiring explicit formulas for the Jacobian terms are prone to error.

A good approximation of the initial operating point data is required to accelerate the convergence. The developed EKF algorithm, technique in this work has the advantage of reducing the problem of singularity since both excess equations and excess variables may be specified. The solution simply achieves the best least squares fit to this specification. Where there is no reason to change an excess variable, it is simply left at its original value.

The linear model obtained has the form given by equation (B.6) (see extended Kalman filter formulation in the Appendix B) as

$$\begin{pmatrix} \dot{y} \\ \dot{z} \end{pmatrix} = \begin{pmatrix} F_0 \\ H_0 \end{pmatrix} + \left[\begin{array}{c|c} A & B \\ \hline 0 & E \end{array}\right] \begin{pmatrix} y \\ z \end{pmatrix} \tag{3.51}$$

The discrete model

As mentioned in section A.3.1, state-space models are most useful for obtaining real-time behaviour of process systems, since they appear in continuous time. However, when the output variables are sampled, the control action is implemented only at discrete points in time. Because discrete-time formulations are most especially suited to computer simulation of process behaviour, a discrete-time model of the process based on the linear model and the relationship of model states to the measurements w_t are respectively given by equations (B.10) and (B.11) as follows

$$x_{t+\Delta t} = A_t x_t + B_t u_t \tag{3.52}$$

$$C_t x_t = w_t \tag{3.53}$$

Kalman filter

With equations (3.52) and (3.53), the transient responses of the state-space model can thus be founded using the Kalman filter (see equations (B.12) to (B.14)), taking into account the expected error covariances Q and R matrices for the model and the measurements respectively as described by equations (3.54) to (3.57). Note that the Kalman Filter has not been used to provide state estimates, but rather as a means to seek agreement between the differential and algebraic equations arising in this DAE model structure. Elements of the Q and R matrices were merely chosen to give satisfactory performance in this task. This technique proved useful to obtain the necessary convergence. For more details on the Kalman filter interpretation see section 5.5.2.1.

$$K_t = M_t\, C_t^T \left[C_t\, M_t\, C_t^T + R \right]^{-1} \tag{3.54}$$

$$x_{t+\Delta t} = A_t\, x_t + B_t\, u_t + K_t \left[\hat{w}_t - C_t\, x_t \right] \tag{3.55}$$

$$M_{t+\Delta t} = A_t \left[I - K_t\, C_t \right] M_t\, A_t^T + Q \tag{3.57}$$

Notice that for DAEs system the observation matrix and observation vector are defined as $\mathbf{C}_t$ and $\hat{\mathbf{w}}_t$ respectively.

3.3.3 Matlab software and results

The EKF algorithm was written in Matlab and is presented in Appendix C. The original Matlab version of the extended Kalman filter software was written by Professor M. Mulholland [2001] "Personal communication".

Using the model described by a set of equations (3.2) to (3.50), excluding equation (3.5) which is not relevant, it was possible to establish the open-loop performance. The variables were constrained in terms of physical reality. For example, the algorithm can handle emptying of tanks with pump pressures responding correctly and flow being lost.

The above theoretical model showed process complexity derived from being multivariable, non-linear, with a high level of interaction between the variables and complex model equations. Tuning of the EKF predictor to find good parameters of the expected observation error covariance, R and the expected prediction error, Q that gives satisfactory performance, was a very difficult task taking into account the large number of variables to deal with, and the little available information about the plant.

Recall that the EKF scheme has been traditionally applied to state and parameter estimation using models described by ODEs. Thus, the software was tested in order to infer the model, based on the estimated outputs for input step changes for the 2-input / 2-output sub-system defined on the Training Plant (see section 2.5). Notice that the output variables under study in the present work, are the levels in tanks T101 and T102, while the inputs are valves CV01 (inflow to T101 from T102) and CV03 (inflow to T102 from T103).

The state-space step responses obtained from the model are shown in Figure 3.1. These responses are equivalent to the step responses derived from experimental data, presented in Figure 7.2. Resulting responses showed similar integrating behaviour to the experimental data and same trend although slower. It took approximately ten minutes for all responses to become steady ramps. Thus, considering a steady state horizon, M = 5 steps, a time interval, Δt was defined to be two minutes (see section 4.3 for details of parameters).

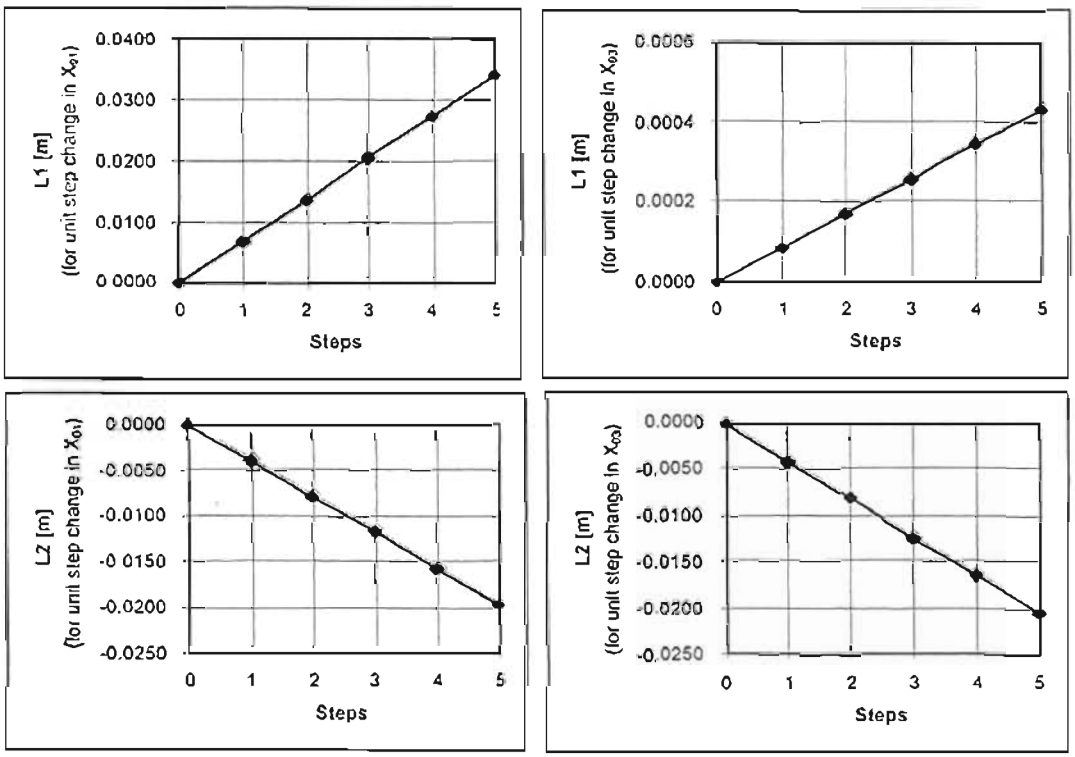


Figure 3.1 Model-predicted unit step responses for the 2-input / 2-output sub-system of the Training Plant (M = 5)

Taking into account that little information about physical coefficients was available, many parameters describing the equipment were guessed, (e.g. pipe friction coefficients) and will be erroneous. However, the results are promising considering that the proposed EKF technique can satisfactorily estimate the state variables describing this system. Further investigation to find actual process parameters is required to improve the model performance.

To overcome the difficulties presented by this theoretical model regarding its complexity and lack of information, an empirical model approach was further adopted (see section A.3.2 in Appendix A). Thus, the process model of the sub-system under study was built using experimental data and then applied to the Linear Dynamic Matrix Controller. Empirical models have the advantage that they are simply based on the arbitrary input functions u(t), and do not require any complex mathematical manipulations or any state transformations, only requiring a data record from well-designed experiments. This issue is discussed in the following chapters.

Chapter 4

Linear Dynamic Matrix Control

4.1 Introduction

The control of the existing Training Plant presents a difficult problem due to a number of factors described in Chapter 3. Among them are the multivariable, interactive nature of the system and the fact that the process is highly non-linear and constrained.

Since Model Predictive Control (MPC), is able to handle most of the difficulties mentioned above for the Training Plant, this technique was applied to design a controller for it, using Linear Dynamic Matrix Control (LDMC), one of the most popular applications of MPC based on the step response. In section 4.2, the general principles of MPC are presented. The design parameters of DMC are discussed in section 4.3 while a case study of DMC application is presented in section 4.4.

4.2 General Principles

Model Predictive Control refers to a class of control algorithms in which a dynamic model with its associated uncertainties is used to predict and optimise process performance. Control design methods based on the MPC concept have found wide acceptance in industrial applications because of their ability to handle process interactions and unusual dynamic responses, and because it does not necessarily demand a rigorous model derived from first principles. An explicit dynamic model of the plant is used to optimise the future actions of the manipulated variables on the output over a longer time period. This flexibility is helpful in modelling unusual process behaviour. On the other hand, MPC provides the only methodology able to handle constraints in a systematic way during the design and implementation of the control [Garcia, Prett and Morari, 1989]. These authors present a reviewed MPC methodology while de Vaal [1999] presents an overview of advanced control techniques.

Successful applications of DMC have been reported in the literature. Robertson, Watters, Desphande, Assef and Alatiqi [1996] compared a DMC method with standard PI control on reverse osmosis (RO) desalination processes to produce a constant quantity of product water with an acceptable purity. They reported that much more flexibility in the operation of an RO plant is available with DMC control. Linear Dynamic Matrix Control (LDMC), based on a linear programming solution for the optimal path to the control horizon has been developed by Mulholland and Prosser [1997] following the methods of Chang and Seborg [1983], as well as Morshedi *et al* [1985]. The combination of linear programming and DMC allows for the handling of explicit constraints on top of the standard DMC structure. The algorithm was applied to control the top and bottom temperature of a semi-industrial distillation column. Mulholland and Narotam [1996] and Prosser [1998] also considered multivariable control using LDMC.

Qualitative models based on fuzzy sets have also become a powerful tool for representing non-linear systems. Demircan, Çamurdan and Postlethwaite, [1999] demonstrated that besides quantitative fundamental and empirical models, a qualitative Fuzzy Relational Model (FRM), can also be used when implementing DMC.

Some design techniques emanating from MPC are Dynamic Matrix Control, discussed in the present chapter, Model Algorithmic Control (MAC) and Model Reference Adaptive Control (MRAC). The last one is discussed briefly in section 5.3.2. The fundamental framework of MPC consists of four elements shared in common by all schemes. What differentiates one specific scheme from the other is the strategy and philosophy underlying how each element is actually implemented. These elements (illustrated in Figure 4.1) may be defined as follows [Ogunnaike and Ray, 1994]:

1. **Reference trajectory specification**

   Desired target trajectory for the process output, **x**(i). This can simply be a step to the new set point value or more commonly, it can be a desired trajectory that is less abrupt than a step.

2. **Process output prediction**

   The appropriate model is used to predict the process output over a predetermined, extended time horizon (with the current time as the prediction origin) in the absence of further control action ("open-loop" response).

3. Control action sequence computation

The same model is used to calculate a sequence of future control moves of the manipulated variables that will satisfy some specified optimisation objective such as:

- Minimising the predicted deviation of the process output from target over the prediction horizon,
- Minimising the expenditure of control effort in driving the process output to target,

subject to prespecified operating constraints. Only the first calculated move is used, since the optimisation is repeated at each sampling time based on updated information (measurements) from the plant.

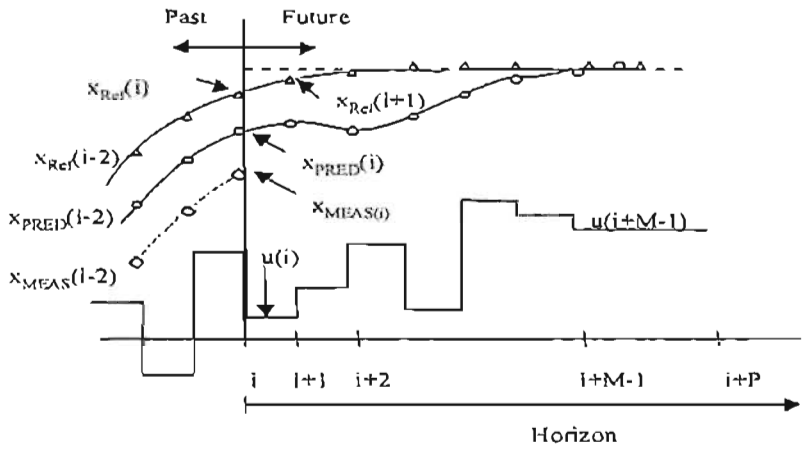


Figure 4.1 Example of elements in model predictive control: △ - △ : reference trajectory, o - o : predicted output, ◊ - ◊ : measured output

The DMC algorithm is currently one of the most popular and widely used MPC algorithms, because it is simple, intuitive and allows the formulation of the prediction vector in a natural way. It is based on a linearised step response model called the *convolution model* (Figure 4.2) to predict the effect of possible control actions. Such a strategy enables the model-based control to anticipate where the process is heading.

The elements of the step response represent the changes observed in the process output at M, consecutive equally spaced, discrete-time instants after implementing a unit change in the input variable.

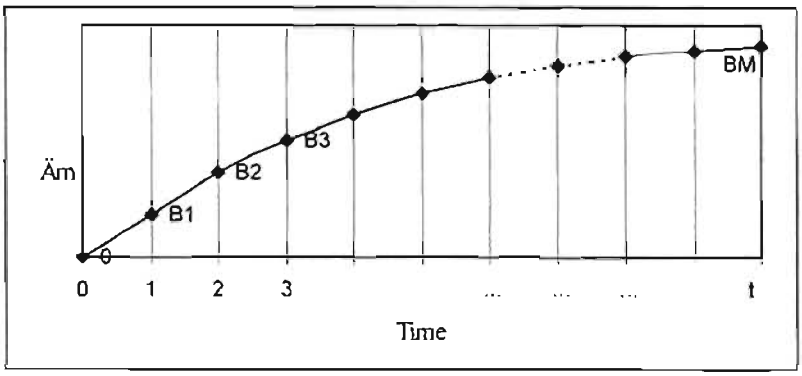


Figure 4.2 Typical output response to a unit step input

Step responses data can be represented in the *convolution model* for future outputs as

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_M \\ x_{M-1} \\ \vdots \\ x_P \end{pmatrix} = \begin{bmatrix} B_1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ B_2 & B_1 & 0 & 0 & 0 & \cdots & 0 \\ B_3 & B_2 & B_1 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\ B_M & B_{M-1} & \cdots & B_1 & 0 & \cdots & 0 \\ B_{M+1} & B_M & B_{M-1} & \cdots & B_1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\ B_N & B_{N-1} & B_{N-2} & B_{N-3} & B_{N-4} & \cdots & B_1 \end{bmatrix} \begin{pmatrix} \Delta m_1 \\ \Delta m_2 \\ \Delta m_3 \\ \vdots \\ \Delta m_M \\ \Delta m_{M-1} \\ \vdots \\ \Delta m_C \end{pmatrix} \tag{4.1}$$

or

$$\mathbf{x} = \mathbf{B}\,\Delta\mathbf{m} \tag{4.2}$$

To implement the predictive algorithm in accordance with the MPC configuration illustrated in Figure 4.3, it will be seen that other matrices are defined from the step response coefficients. $\mathbf{B_0}$ is the *Offset matrix* of the coefficients and $\mathbf{B_{OL}}$ a *matrix of the Open Loop* response coefficients.

An approach for on-line identification of the step response coefficients, in $\mathbf{B_0}$ matrix which, are used to fill $\mathbf{B_{OL}}$ and $\mathbf{B}$ matrices applied in the DMC algorithm, is illustrated in this study and is discussed in chapter 6.

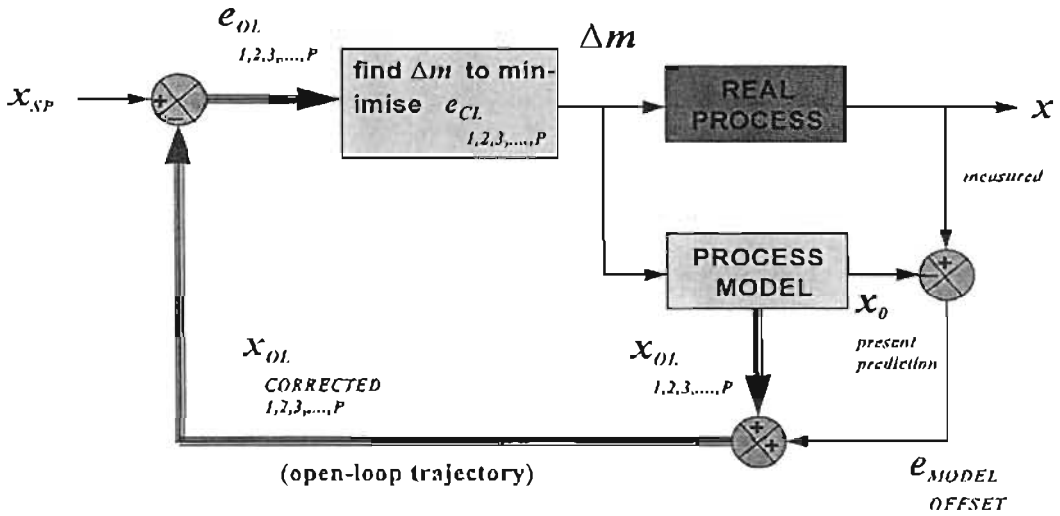


Figure 4.3 Model Predictive Control configuration [Mulholland and Prosser, 1997]

Thus, the step responses of an m-input / n-output system can be represented by a series of matrices B_1, B_2, B_3, ..., B_P. The position (i,j) in each matrix B_t is a point on the trajectory at time t for the (i th) output as it responds to the j th input. Now consider the following construction in which Δm_t is the vector of input moves (changes) made at time t, and x_t is the vector of outputs at time t. A moving frame of reference for time is used in which $t=0$ represents the present time.

$$\begin{pmatrix} x_{0PRED} \\ x_{PRED} \end{pmatrix} = \begin{pmatrix} x_{0PRED} \\ x_{0PRED} \\ x_{0PRED} \\ x_{0PRED} \\ \vdots \\ x_{0PRED} \\ \hline x_1 \\ x_2 \\ \vdots \\ x_M \\ x_{M+1} \\ \vdots \\ x_P \end{pmatrix} = \left[\begin{array}{ccccccc|ccccccc} B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \hline B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & B_1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & B_2 & B_1 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & B_{M-2} & B_{M-3} & \cdots & B_1 & 0 & \cdots & 0 \\ B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & B_{M-1} & B_{M-2} & \cdots & B_2 & B_1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & B_{M-1} & B_{M-2} & B_{M-3} & B_{M-4} & B_{M-5} & \cdots & B_1 \end{array}\right] \begin{pmatrix} \Delta m_{-M+1} \\ \Delta m_{-M+2} \\ \Delta m_{-M+3} \\ \Delta m_{-M+4} \\ \vdots \\ \Delta m_0 \\ \hline \Delta m_1 \\ \Delta m_2 \\ \vdots \\ \Delta m_M \\ \Delta m_{M+1} \\ \vdots \\ \Delta m_P \end{pmatrix} \tag{4.3}$$

The vector of vectors $x_{0PRED} = \mathbf{B}_0\,\Delta\mathbf{m}_{PAST}$ in equation (4.3) contains P identical predictions of the output vector at the present time $t=0$. The vector x_{PRED} contains predictions of the output vector at P points on the future trajectory, as contributed to by the past M control moves [$\Delta\mathbf{m}_{PAST} = (\Delta m_{-M+1}, \ldots, \Delta m_0)^T$], and a future P control moves. We choose M to be large enough to ensure that all step responses are at steady state at this point (B_M). Then we expect that our predictions of x_{0PRED} and x_{PRED} will only be in error by steady-state offsets emanating from earlier than M steps before the present time.

We make P copies of this offset error by comparison with a vector x_{0MEAS} containing P copies of the present output measurement

$$x_{0MEAS} = \begin{pmatrix} x_{0MEAS} \\ x_{0MEAS} \\ x_{0MEAS} \\ x_{0MEAS} \\ \vdots \\ x_{0MEAS} \end{pmatrix} \tag{4.4}$$

and use them to correct our future predictions as follows

$$\mathbf{x}_{CL} = \mathbf{x}_{0MEAS} + \left[\mathbf{B}_{OL} - \mathbf{B}_0\right]\Delta\mathbf{m}_{PAST} + \mathbf{B}\Delta\mathbf{m} \tag{4.5}$$

The matrices B_0, B_{OL} and B are clearly top-left, bottom-left and bottom-right respectively in the structure given by equation (4.3). In DMC we solve on each step for a limited sequence of N moves ($\Delta \mathbf{m}^*$) to be optimised ($N \leq P$). Actually, only one or two future moves are normally solved, allowing a reduction to one or two columns in B, resulting in the matrix $\mathbf{B}^*$ (non-square $P \times N$ matrix). Although only the first move is ever used, a solution for a second move allows stronger action on the first move, because the solution plans a correction with the second move. In this way higher gains are obtained.

A quadratic objective function J, dependent only on $\Delta \mathbf{m}^*$ can be defined as:

$$\begin{aligned} J(\Delta \mathbf{m}^*) &= (e_{CL})^T W (e_{CL}) + (\Delta m^*)^T \Lambda (\Delta m^*) \\ &= (\mathbf{x}_{OL} - \mathbf{x}_{SP} + \mathbf{B}^* \Delta \mathbf{m}^*)^T \mathbf{W} (\mathbf{x}_{OL} - \mathbf{x}_{SP} + \mathbf{B}^* \Delta \mathbf{m}^*) + (\Delta \mathbf{m}^*)^T \Lambda (\Delta \mathbf{m}^*) \end{aligned} \quad (4.6)$$

where $\mathbf{x}_{SP}$ is the process variable set point trajectory, while the closed loop error, e_{CL}, is defined as:

$$e_{CL} = e_{OL} + \mathbf{B} \Delta m^* \quad (4.7)$$

and the "open-loop" response, $\mathbf{x}_{OL}$, corrected for present model offset, is given by

$$\mathbf{x}_{OL} = \mathbf{x}_{0MEAS} + [\mathbf{B}_{OL} - \mathbf{B}_0] \Delta \mathbf{m}_{PAST} \quad (4.8)$$

Thus, an optimal sequence of control moves, $\Delta \mathbf{m}^*$, for minimum control move effort which achieves minimum deviation from the set point trajectory up to the time horizon P, is found by minimising J with respect to $\Delta \mathbf{m}^*$. This optimisation problem is solved using a least squares technique (see Section 5.5.2.2). The LDMC approach then finds the "closest" $\Delta \mathbf{m}^*$ to this point which satisfies the input and output constraints (see below).

A weighting factor matrix, $\mathbf{W}$, and a move suppression matrix, Λ, generally diagonal, (discussed in the next section), are incorporated to determine the extent to which deviations from setpoint, or control moves, are discouraged.

This model predictive control (MPC) format lends itself to dealing with constraints in both the input and output variables. The present work uses a combination of quadratic and linear objective functions in which linear programming (LP) is used to obtain a final solution within constraints (LDMC). This solution has the benefits of QDMC without being as computationally demanding (Chang and Seborg, 1983; Morshedi *et al*, 1985). It is based on the idea of obtaining the closest approach to the least squares optimum, should it lie outside of the constraints.

4.3 Design parameters for DMC

As mentioned above, within the DMC algorithm, the process dynamics are represented by a set of numerical coefficients determined from a process step response. While an extensive review of the DMC algorithm is not discussed in the above section (the interested reader should consult references [Ogunnaike and Ray, 1994; Mulholland and Prosser, 1997] for a detailed description), it is useful to discuss several design parameters which can be adjusted to give the desired response as well as an appropriate amount of controller effort. Bearing in mind that only the first N control moves of the possible P need be optimised (the rest can remain zero), the parameters include:

- sampling time, Δt
- steady state horizon, M
- optimisation horizon, P
- number of future control moves, N
- process variable weight, W
- move suppression, Λ

At discrete sampling instants, Δt, the step response coefficients may be determined from the step response model. The steady state horizon, M, is the time for the open-loop step response to reach e.g. 99 % completion. The choices of the sampling time and the steady state horizon are interrelated as the steady state corresponds to the settling time of the system expressed as a multiple of the sampling interval. The selection of these parameters should insure that no truncation problems arise in calculating the predicted values for the *convolution model*. The sampling time selected must be small enough to accurately represent the process dynamics. However, if chosen too small, it will require an extraordinarily large steady state horizon. Small values of M, with subsequent large values of Δt, are desired so as to reduce the computational requirement. In cases where a dead time is to be accounted for the sampling time must obviously be smaller than this delay.

Parameter P is defined as the optimisation horizon. It is equal to the number of predicted controlled variable response times that are used in the optimisation calculations.

N is used to indicate to the controller, the number of future control moves that are calculated in the optimisation step to reduce the predicted errors [Seborg, Edgar and Mellichamp, 1989]. The computational effort increases as N increases and a small value of N leads to a robust controller that is relatively insensitive to model errors. Thus, as the value of N is increased, more degrees of freedom are available for the controller optimisation step. This will result in tighter control, at least as far as minimising the objective function is concerned. While increasing N can result in better control system performance, the manipulated variable movements become larger and there is a reduction in the controller's robustness.

However, it has been noted that if a particularly fast response is necessary, and the reduced robustness can be tolerated, a controller with higher value of N can be more finely tuned using the continuous adjustment of Λ as opposed to discrete values of P or N [Prosser, 1998].

In practical situations, it is usually necessary to suppress the movement of the manipulated variables. This is achieved by incorporating move suppression values, Λ, into such variables. Increasing the Λ values will slow down the controller's closed-loop response and reduce the size of the changes that are generated. In addition, it is possible to achieve tighter control of some variables relative to others, by multiplying the controlled variables set point deviations (squared) by different weighting values, W, in the objective function (equation (4.6)).

The selection of such weight parameters for the controlled variables is generally a relative matter, depending on the need to control some outputs tighter than others. It is important to note that while the selection of individual weight parameters does indicate a designer's desire to control one output tighter than another, a comparison of the absolute values of the weighting parameters does not indicate the degree to which one variable is controlled relative to others [Robertson et al., 1996]. This comes about because variables may have different units, and some units, such as those of temperature and flow rate, cannot be directly compared.

4.4 Application of LDMC to a Pump-tank system – case study

The on-line testing of LDMC was done using a *second* plant considered in this research and mentioned in Section 1.1. This was the 2-input / 2-output pump-tank system used for simulation and laboratory tests in the School of Chemical Engineering at the University of Natal, represented in the Figure 4.4.

The system consists of a pair of interacting tanks receiving water from a reservoir by means of a centrifugal pump. The two input variables are control valves: valve 1 is on the return line to the reservoir and valve 2 connects the two tanks; the output variables are levels 1 and 2. At the bottom of each tank is a small pipe connecting the tanks to a shared line, which returns to the reservoir. The valves on these pipes remain at some fixed open position and allow water to flow out the tanks and back to the reservoir continually. V1, V2 indicate valve position as percentage open, and L1, L2 indicate tank level as percentage full.

The Pump-tank system was operated to get the open-loop step responses presented in Figure (4.5). These responses show stable process behaviour, i.e., "self-regulatory" (see section A.2 in Appendix A). The time taken for the system to reach a steady-state in all step responses after disturbances was approximately 10 minutes. Note that the response of L2 to V2 achieves the steady-state on the last point. A time interval, Δt, was chosen to be 60 seconds and giving a steady-state horizon, M of 10 steps.

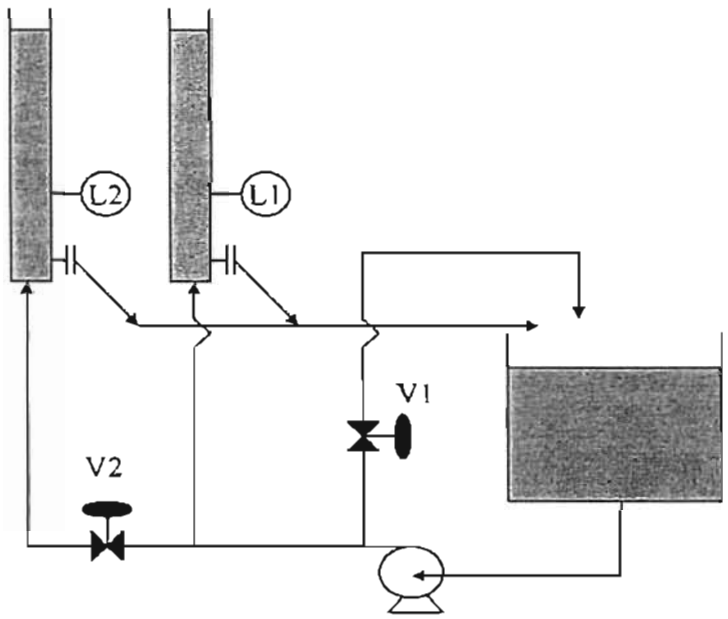


Figure 4.4 2-input / 2-output Pump-tank system for simulation and laboratory tests

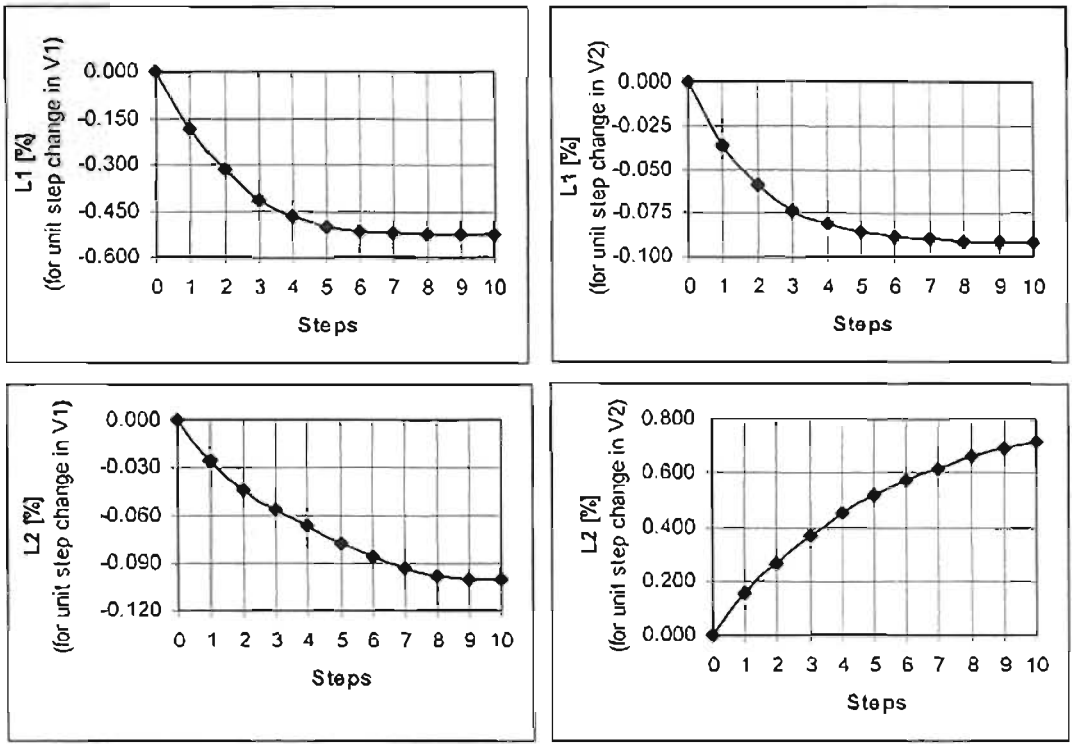


Figure 4.5 The unit step responses for the Pump-tank system (M = 10)

To illustrate some properties of LDMC, the controller design and the effect of constraints were considered. The data acquisition and control action implementation tasks were performed with the optimisation horizon, P = 10, the same as the steady state horizon, M. The move suppression factor, λ = 1 and weight factor, W = 100 were tuned on-line giving satisfactory performance with two optimised control moves as illustrated in Figures 4.6 and 4.7, for step changes in the set point made at 340 and 640 seconds for outputs 1 and 2 respectively. Note, as mentioned before, a controller update interval of 60 seconds was used in all of these tests.

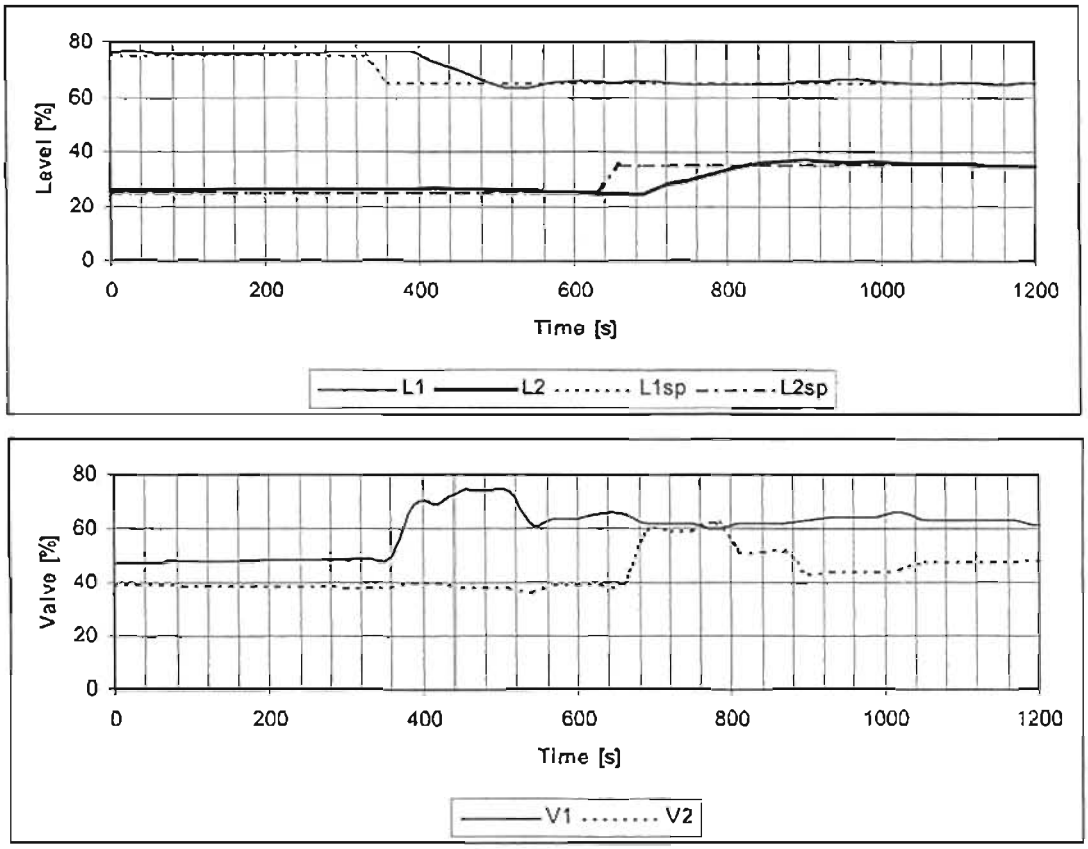


Figure 4.6 Unconstrained closed loop step response for N = 2, λ = 1 and W = 100

In the unconstrained run (Figure 4.6), the control was efficient in reducing set point deviations, while at the same time being severe in terms of control movements since the second optimised move makes a correction to the first move, which can therefore be an "overshoot" move.

The Linear Dynamic Matrix Control was also observed to take logical actions accounting for the system coupling and allowing for constraints as illustrated by Figure 4.7. In this case, upper constraints of 60 % for input 1 and 50 % for input 2 were imposed. As a result, an offset can be seen in the level 1 in the period where the constraints are active. This is as would be expected from reducing the degrees of freedom of the manipulated variables. However, level 2 presents a smooth response and attempts to obtain the best fit.

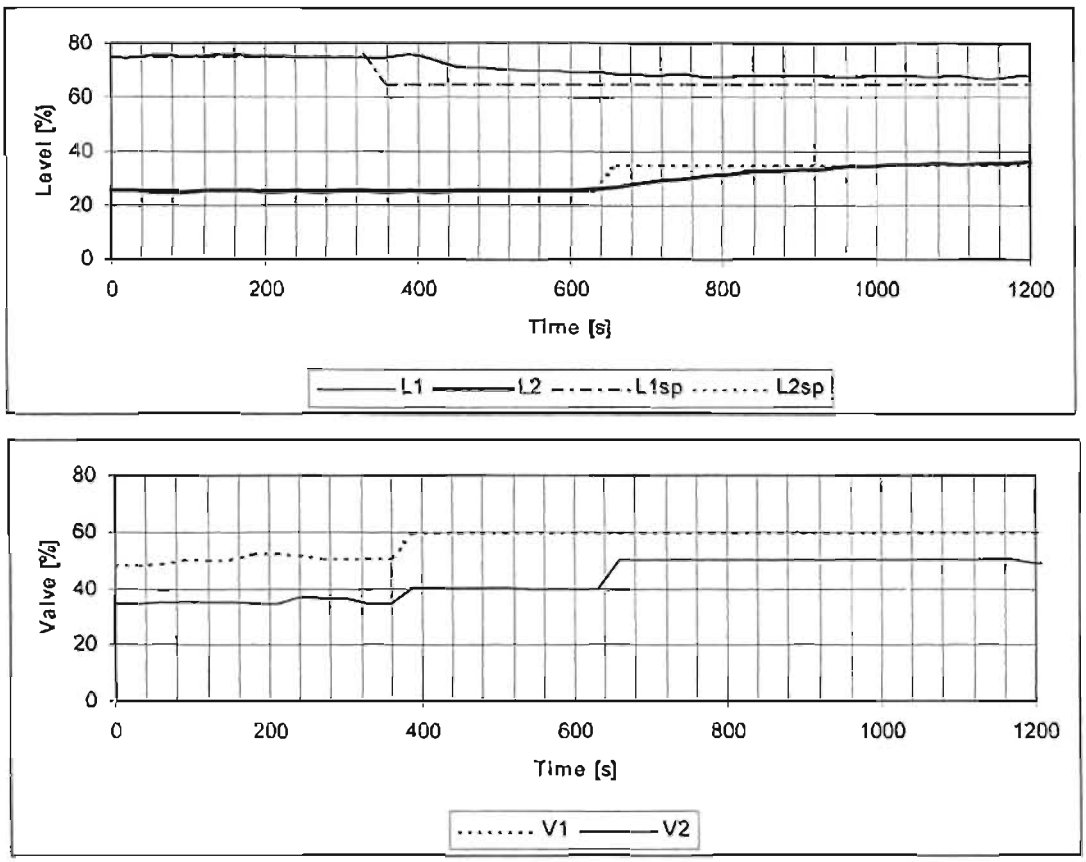


Figure 4.7 Closed loop step response for N = 2, Λ = 1 and W = 100 with upper constraints V1: 60 % and V2: 50 %

Notice that a large number of tuning parameters, including move suppression, set point deviation weighting factor and different numbers of control moves were tested for this system, subjected to a step change in the set point, and the LDMC algorithm gave acceptable performance for a wide range of these conditions. When the weighting factors are held constant, decreases in the move suppression values for each manipulated variable resulted in progressively faster responses. If the move suppression is held and the weighting factor increased, faster responses were again noted.

Chapter 5

Adaptive Control Principles

5.1 General

Traditionally, chemical process control has focused almost entirely on the analysis and control of linear systems. Therefore, most existing control systems design and analysis techniques are suitable only for linear systems. This is because many processes are in fact only mildly non-linear, and even strongly non-linear processes take on approximately linear behaviour as they approach steady state.

Nevertheless, for those non-linear process whose non-linearities are strong, linear controller design techniques often prove inadequate as the process moves further away from steady state and more effective alternatives must be considered. An adaptive control system is one of the advanced control strategies that can, in some cases, provide significantly improved process control beyond that which can be obtained with conventional controllers.

Despite DMC presenting several features and advantages as described in Chapter 4, being linear it introduces some limitations when it is applied to non-linear systems. To deal with the non-linearity nature of the present process (see section 3.3), an adaptive control scheme with recursive parameter estimation was proposed and implemented in the DMC algorithm. A detailed discussion about adaptive DMC is presented in Chapter 6.

Basic concepts of adaptive control and model parameter estimation using a recursive least squares technique are introduced in this chapter. An overview of adaptive control is given in the section 5.2, while several schemes of adaptive control are discussed in section 5.3. In section 5.4 suggestions of how to proceed in order to decide what type of controller to use are given and some aspects of the identification and parameter estimation are discussed in section 5.5.

5.2 Adaptive control overview

Modern industrial processes are getting more and more complex, and are inherently non-linear, respond slowly, and have large time-delays and so on. The quest for improvement in the performance of process plants and the availability of fast computing power has given rise to the development of a new generation of advanced control algorithms. These algorithms can identify the current optimal operating point of a process and effect the transition of the process to a new optimal point in an acceptable and safe manner.

According to Åström and Witternmark [1995], *adaptive control* is a technique that automatically adjusts the controller settings with process moves from one operating point to another, to accommodate changes in the process to be controlled or its environment.

An adaptive control system can be thought of as having two loops. One loop is a normal feedback with the process and the controller; the other loop is the parameter adjustment loop and this is often slower than the normal feedback. The general strategy for designing adaptive control systems is to estimate the model parameters on-line and then adjust the controller settings based on the current parameter estimates.

Adaptive control schemes provide systematic, flexible approaches for dealing with uncertainties, non-linearities, and time-varying process parameters. Consequently, adaptive control systems offer significant potential benefits for difficult process control problems such as non-linearity and / or high-order, where the process is poorly understood and / or changes in unpredictable ways. This control technique has been applied in chemical processes. Several theoretical and experimental studies have appeared in chemical engineering literature, [Seborg, Edgar and Shah, 1986; Åström and Witternmark, 1995], while the number of industrial adaptive control techniques available increases continuously. Most of the adaptive control systems require extensive computations for parameter estimation and optimal adjustment of controller settings.

Seborg et al [1986] have reviewed the adaptive control strategies from a process control perspective and describe leading design techniques. This survey paper is a good reference to get detailed information about this issue.

5.3 Adaptive schemes

Two general categories of adaptive control problems can be distinguished, differing mainly in the way the controller parameters are adjusted. The first category consists of problems where the process changes cannot be directly measured or anticipated. The most popular are model *reference adaptive control* and *self-tuning adaptive control*. The second category consists of control problems where process changes can be anticipated or inferred from process measurements. In these situations, if the process is reasonably well understood, it is feasible to adjust the controller settings in a predetermined manner as process conditions change, and this control strategy is referred to as *gain scheduling*. Most adaptive control literature has emphasised the first category [Maiti, Kapoor and Saraf, 1994; Maiti and Saraf, 1995a &1995b; Aitchison and Mulholland, 1997; Demircan, Çamurdan and Postlethwaite, 1999]. A brief description of these schemes is follows:

5.3.1 Scheduled Adaptive Control

As mentioned above, in some cases it is possible to find measurable variables that correlate well with changes in process dynamics. Such variables can then be used to change the controller parameters by monitoring the operating conditions of the process, to reduce the effects of parameter variations. This scheme is referred to as gain scheduling and is commonly used in industry to overcome the gain mismatch. A block diagram of this scheme is shown in the Figure 5.1.

A good knowledge of the process is required to apply this method and a great advantage of this scheme is that the controller adapts quickly to changing conditions. Since no estimation parameter occurs, the limiting factor depends on how quickly the auxiliary measurements respond to process changes.

Gain scheduling has been used in special cases, such as combustion control, pH control and other well-known control problems that present difficulties due to large variations in process dynamics [Ogunnaike and Ray, 1994].

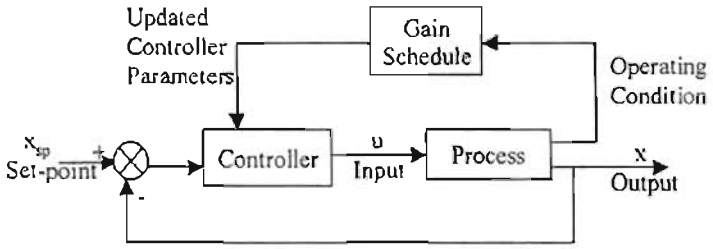


Figure 5.1 Block diagram of a system in which the influences of parameter variations are reduced by gain scheduling

5.3.2 Model Reference Adaptive Control

The key component of the Model Reference Adaptive Control scheme (MRAC), is the reference model that consists of a reasonable closed-loop model of how the process should respond to a set point change. The reference model output is compared with the actual process output and the observed error e_M is used to drive some adaptation scheme to cause the controller parameters to reduce e_M to zero. The adaptation scheme could be some control parameter optimisation algorithm that reduces the integral squared value of e_M, or some other procedure. This scheme is illustrated in the Figure 5.2.

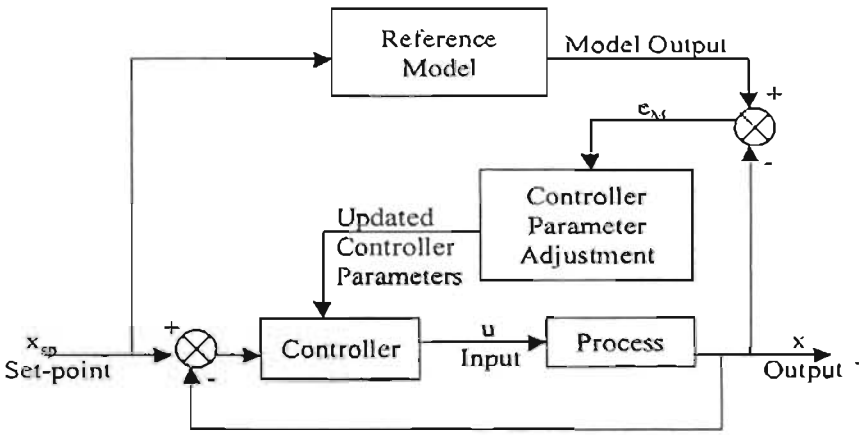


Figure 5.2 Model reference adaptive control structure

5.3.3 Self-tuning Adaptive Control

Self-tuning adaptive controllers differ from the model reference adaptive controller in basic principle. The self-tuning control configuration in Figure 5.3 is flexible enough to accommodate a wide variety of parameter estimation techniques and controller design strategies. It makes use of the process input and output to estimate recursively, on-line, the parameters of an approximate process model. Thus, as the actual non-linear process changes operating region or changes with time, an approximate model is continuously updated with new parameters. The updated model is then used in prespecified control system design procedures to generate updated controller parameters. The controller could be a PID controller or more complex control system structures such as a cascade controller, DMC, etc.

Since the estimated model determines the effectiveness of the controller, the most essential feature of the self-tuning controller is reliability and robust model identification. In the present work, a self-tuning adaptive control scheme is implemented in the Linear Dynamic Matrix Control algorithm, where the coefficients of the step response are updated recursively (Chapter 6).

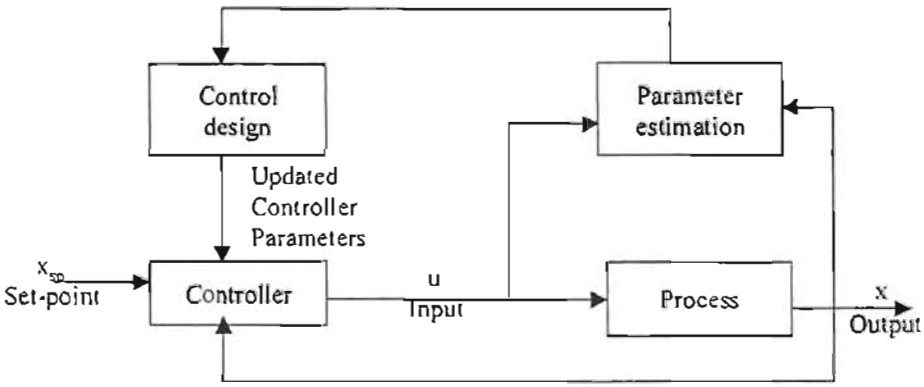


Figure 5.3 Self-tuning control system

5.4 Procedure to decide what type of controller to use

An adaptive controller, being inherently non-linear, is more complicated than a fixed gain controller. Before attempting to use adaptive control, it is therefore important to investigate whether the control problem might be solved by constant-gain feedback. Figure 5.4 shows how to decide what type of controller to use [Åström and Wittlermark, 1995].

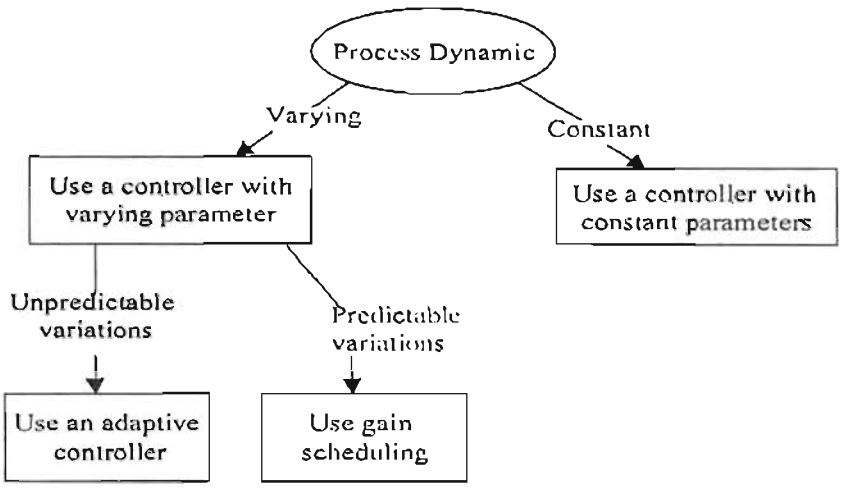


Figure 5.4 Procedure to decide what type of controller to use

5.5 Identification and parameter estimation

As mentioned above, on-line determination of the process parameters is a key element in adaptive control. A recursive parameter estimator appears explicitly as a component of a self-tuning regulator (see Figure 5.3). Parameter estimation also occurs implicitly in model reference adaptive control (see Figure 5.2). Since system identification is executed automatically in adaptive systems, some aspects of it are discussed in this section.

5.5.1 Comments on identification and parameter estimation

The non-linear and non-stationary nature of a typical chemical process leads to a change in its dynamic characteristics during operation. To cope with this situation, a controller should be able to adjust its parameters in an "optimum" manner. However, the complexity of these processes makes them difficult to understand, model, interpret and control. As a consequence engineers often try to develop empirical dynamic process models for these systems, directly from input / output data rather than attempting to develop time consuming, expensive fundamental, analytical models [Barnard and Aldrich, 2000].

System identification deals with the problem of building mathematical models of dynamical systems based on observed data from the system. This activity can be carried out in an *off-line* or *on-line* manner. In the *off-line* situation, the process data is first stored and later transferred to a computer and analysed. For this "batch" processing technique, a whole data set is evaluated at

once. In the *on-line* techniques, the identification is performed in on-line operation with the process and, two ways of processing data can be distinguished: real-time processing and batch processing. In real-time processing the data is evaluated immediately at each sample instant, while in batch processing, the data is evaluated periodically after periods of measurements have been made. Real-time processing in general needs no storage of data since each new point is used to update the model parameters.

Table 5.1 Some examples for the relationships between different final goals and some specifications of process identification [Isermann, 1980]

Final goal of model application	Type of process model	Required accuracy of model	Identification method
Verification theoretical models	linear/continuous-time non-parametric/ parametric	medium/high	off-line step response of frequency parameter-estimation
controller parameter tuning	linear, non-parametric, continuous-time	low for input/output	off-line step response
computer aided design of digital control discrete time	linear, parametric (non-parametric)	medium for input/output behaviour estimation	on-line off-line parameter algorithms
self-adaptive digital control	linear, parametric, discrete-time	medium for input/output behaviour	on-line parameter estimation in closed loop
process parameter monitoring and failure detection	linear, non-linear, continuous-time parameters	high for process	on-line parameter estimation

It is thus important to regard first the final goal for the application of the process model, since this determines the type of the model and its accuracy requirements and the identification method. Prior knowledge of the process is required. Table 5.1 shows some of the relationships between different final goals and some specifications of process identification [Isermann, 1980]. The key elements of system identification are the selection of model structure,

experiment design, parameter estimation and validation. The selection of model structure and parameterisation are fundamental issues.

A general procedure of process identification is illustrated in the Figure 5.5 and shows how the identification is an iterative procedure. Isermann [1980], presents a good description of practical aspects of process identification while Ljung [1999] describes a theory of system identification that has direct consequences for the understanding and practical use of available techniques.

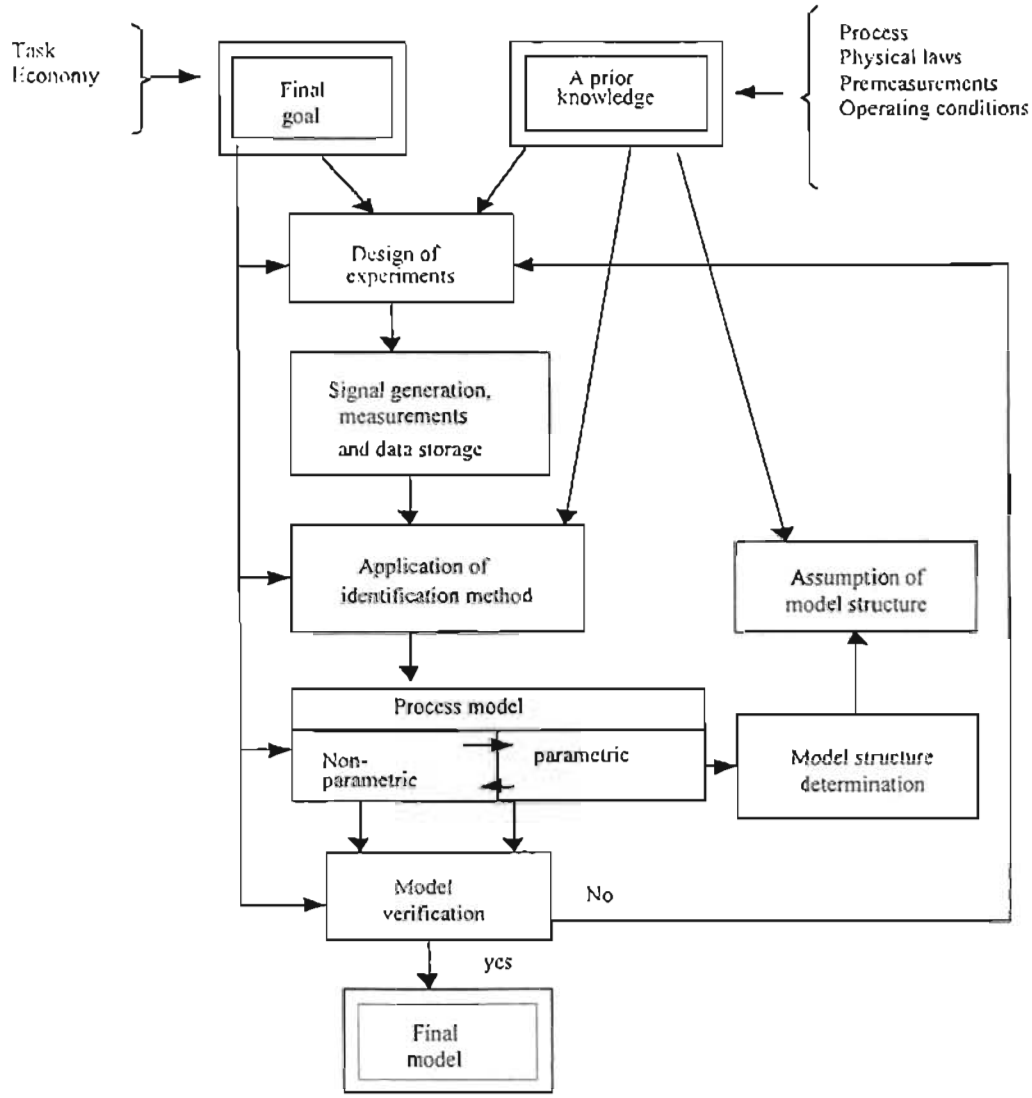


Figure 5.5 General procedure of process identification

Because system identification has been sufficiently formalised for linear systems, but not for the empirical identification of non-linear dynamic systems, many researchers have studied this problem. A method for establishing second order plus dead time model parameters under closed-loop PI control is proposed by Suganda, Krishnaswamy and Rangaiah [1998]. The

advantage of this technique is that the required closed-closed loop response data can be obtained while the process is in normal operation since many industrial controllers are of the PI-type.

Barnard and Aldrich [2000] propose a formal methodological framework for the empirical modelling of a non-linear dynamic system that can be parameterised as a state space system. The methodology involves classification of a time series and associating a suitable model with the prediction of the time series.

Some aspects of the real time parameter estimation are discussed below, followed by an on-line identification example using batch processing. A real-time processing example is presented in Chapter 6.

5.5.2 Recursive least square estimation of model parameters

In adaptive control, real-time (or sequential) updating of the model parameters is more appropriate than batchwise (nonsequential) processing of the input-output data. Algorithms that are suited to real-time usage and are based on successive updating of the model parameters are called *recursive*. There are a large number of recursive identification algorithms described in the literature. Treatments of these techniques are given in many textbooks, and Ogunnaike and Ray [1994]; Åström and Wittenmark [1995] and Ljung [1999] may be mentioned as suitable references for further study. The most popular technique is Recursive Least Squares (RLS), discussed in this section. It is assumed in this technique that the order and the form of the system are known. We start thus by giving an overview of Kalman filtering since it is used to estimate the parameters of the model recursively by using the least squares technique.

5.5.2.1 Kalman filter interpretation

Consider linear, discrete time dynamic systems where predictions and observations are subjected to random errors to account for the uncertainties:

$$\mathbf{x}_{j+1} = \mathbf{A}\mathbf{x}_j + \mathbf{B}\mathbf{u}_j + \boldsymbol{\delta}_{j+1} \tag{5.1}$$

and for the measurement:

$$\mathbf{y}_i = \mathbf{G}_i\mathbf{x}_i + \boldsymbol{\mu}_i \tag{5.2}$$

where **x** is the state vector, **u** the input vector of independent variables, i represents the time and **y** the system output. **G** is an observation matrix while **A** and **B** are matrices of appropriate dimensions (n x n and n x m, respectively, for an n-dimensional state and an m-dimensional input). These matrices (**A** and **B**) typically correspond to unknown values of physical coefficients, property constants and the like. δ and μ are process and measurement noise contributions respectively acting on the states. They result from both measurement imperfections and disturbances affecting the process. They are considered to be random variables with zero means, and with known covariances:

$$\mathbf{E}\left\{\delta\,\delta^T\right\} = \mathbf{R}$$
$$\mathbf{E}\left\{\mu\,\mu^T\right\} = \mathbf{Q} \tag{5.3}$$

and for uncorrelated δ and μ

$$\mathbf{E}\left\{\overline{\delta}\,\overline{\mu}^T\right\} = [0] \tag{5.4}$$

Linear, discrete-time models are preferred for adaptive control because they lead to algorithms that are readily implemented on a digital computer.

In a real time identification situation (Figure 5.6), $\mathbf{x}_{i+1}$, $\mathbf{x}_i$, and $\mathbf{u}_i$ are observed on-line and the elements inside A and B matrices that give a good description of the input / output behaviour of the system need to be found. The on-line computation of the model must be done in such a way that the processing of the measurements from one sample can, with certainty, be completed during one sampling interval. Otherwise the model building cannot keep up with the information flow.

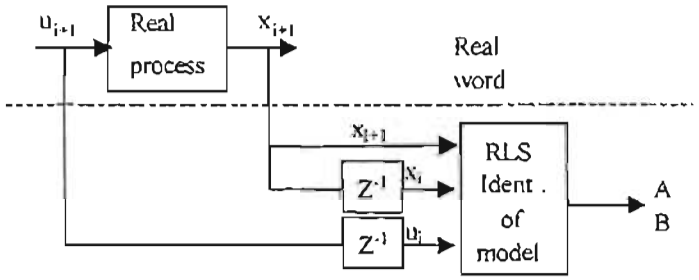


Figure 5.6 On-line identification configuration

The Kalman interpretation of this system, taking into account the expected errors, is that the filter gain K, is yielded on each time-step for adjusting x as follows:

$$K_l = M_l G_i^T \left[G_i M_l G_i^T + R \right]^{-1} \tag{5.5}$$

$$x_{l+1} = A_l x_l + B_l u_l + K_l \left[\hat{y}_l - G_i x_l \right] \tag{5.6}$$

$$M_{l+1} = A_i \left[I - K_l G_l \right] M_l A_i^T + Q \tag{5.7}$$

where $\mathbf{M}_i$ is the filter covariance matrix (diagonal and initially small to promote fast adjustment at the start), $\mathbf{R}$ the expected observation error covariance (usually diagonal) and $\mathbf{Q}$, the expected prediction error covariance (usually diagonal). Higher prediction errors $\mathbf{Q}$ relative to observation errors $\mathbf{R}$, force the filter to follow observations more closely, whilst specifying higher observation errors $\mathbf{R}$ makes the model less sensitive to observations.

The initial conditions can be set so that $\mathbf{x}_0$ is what we guess the parameter vector to be before we have seen the data, and $\mathbf{M}_0$ is the initial covariance matrix, which reflects the confidence in this guess.

Equation 5.6 shows that the estimate $\mathbf{x}_{i+1}$ is obtained by adding a correction to the prediction of $\mathbf{x}_{i+1}$ based on $\mathbf{x}_i$ according to the model. The correction term for the model parameter vector is thus proportional to the prediction error (difference between the measured value of $\mathbf{y}_i$ and the prediction of $\mathbf{y}_i$) based on the previous estimate. The components of the Kalman filter gain matrix, $\mathbf{K}_i$ are weighting factors that introduce an optimal correction into the integration cycle.

Notice that this form (equations (5.5) to (5.7)), allows A and B to vary in time. This provides a way to handle non-linearities, since, as the process moves to a new operating point, elements of these matrices will change. Recall that these equations were also applied in Chapter 3 for state estimation of the state-space model of the Training Plant, using an extended Kalman filter.

5.5.2.2 Least squares parameter estimation

Parameter estimation is concerned with the determination from experimental data of the best set of values for unknown parameters in a process of known form. The least squares estimation approach (also called linear regression) is a basic technique for parameter estimation. The method is particularly simple if the model has the property of being *linear in the parameters*.

Based on Ogunnaike and Ray [1994], Åström and Wittenmark [1995], we present some basic principles for parameter estimation. Consider a mathematical model subjected to an unavoidable measurement error, as well as the inaccuracies that can be written in the form:

$$\mathbf{y}_i = \mathbf{f}(\mathbf{g}_i, \mathbf{p}) + \mathbf{e}_{M_i} \qquad ; i = 1, 2, \ldots, t \tag{5.8}$$

where y is the observed variable, p are parameters of the model to be determined, g is a known function that may depend on other known variables and is called the *regression variable* or the *regressor* and e_{Mi} the vector of errors between the model prediction and the actual data.

The parameter estimation is now involved with finding a specific set of parameter values such that some scalar function J of the vector p, known as the objective function and usually represented as $J(p)$, since it depends on the parameter values, is minimised. Typically we use the quadratic form:

$$J(p) = \sum_{i=1}^{i=t} \left[e_{M\,i}\right]^T \left[e_{M\,i}\right] \tag{5.9}$$

or from equation (5.8)

$$J(\mathbf{p}) = \sum_{i=1}^{i=t} \left[\mathbf{y}_i - \mathbf{f}(\mathbf{g}_i, \mathbf{p})\right]^T \left[\mathbf{y}_i - \mathbf{f}(\mathbf{g}_i, \mathbf{p})\right] \tag{5.10}$$

where the summation is over all of the data points. Sometimes it might be necessary to assign more weight to more precise measurements, and less weight to others. This is accomplished by introducing weighting coefficients as follows:

$$J(\mathbf{p}) = \sum_{i=1}^{i=t} \left[\mathbf{y}_i - \mathbf{f}(\mathbf{g}_i, \mathbf{p})\right]^T \mathbf{W}_i \left[\mathbf{y}_i - f(\mathbf{g}_i, \mathbf{p})\right] \tag{5.11}$$

The coefficients W_i reflect the relative precision of the measurements (for more details about weight factors see section 4.3).

It can be seen from equation (5.10) that this parameter estimation is really a quadratic optimisation problem. Its objective is to estimate the parameters in p that minimise the model error.

Remember that this technique is also applied in the constrained Dynamic Matrix Control algorithm (Chapter 4) where J *is* a defined quadratic function depending on a limited sequence of N moves ($\Delta \mathbf{m}^*$) to be optimised.

If $\mathbf{f}(\mathbf{g}_i, \mathbf{p})$ is linear in the parameters $\mathbf{p}$, equation (5.8) can be represented by equation (5.2) as:

$$\mathbf{y} = \mathbf{G}\,\mathbf{p} + \mathbf{e}_{M_i} \tag{5.12}$$

where G is the complete matrix of independent variables compiled for each set; y is the entire collection of the experimentally obtained data; and $\mathbf{e}_{Mi}$ contains the prediction errors, and we wish to find a vector p of suitable parameters. If G were square and non-singular, we could get the constant coefficients p directly from

$$\mathbf{p} = \mathbf{G}^{-1}\,\mathbf{y} \tag{5.13}$$

However, in general, G will not be square and the linear (least squares) parameter estimation problem is now to find the vector p for which the squared deviation of the data set y from the model $\mathbf{G}p$ is minimised. The function is minimal for parameters p such that:

$$\mathbf{G}^T\,\mathbf{G}\,\mathbf{p} = \mathbf{G}^T\mathbf{y} \tag{5.14}$$

If the matrix $\mathbf{G}^T\mathbf{G}$ is non-singular, then the minimum is unique and given by:

$$\mathbf{p} = (\mathbf{G}^T\,\mathbf{G})^{-1}\,\mathbf{G}^T\,\mathbf{y} \tag{5.15}$$

5.5.2.3 The recursive least square parameter estimation

In the adaptive controllers, the observations are obtained sequentially in real time. It is then desirable to make the computation recursive to save computation time. Computation of least square estimates can be arranged in such a way that the results obtained at time i can be used to get the estimates at $i+1$. We can rewrite the solution of equation (5.15) in the recursive form by

using the Kalman filter. We also assume that in the equation (5.1), the matrix A is an identity matrix and, that matrix B is the null matrix:

$$\mathbf{p}_{t+1} = I\,\mathbf{p}_t + \mathbf{e}_{M_t} \tag{5.16}$$

Thus, following the Kalman filter interpretation (equations 5.5 to 5.7), the RLS fit is given on-line by:

$$\mathbf{K}_t = \mathbf{M}_t \mathbf{G}_t^T \left[\mathbf{G}_t \mathbf{M}_t \mathbf{G}_t^T + \mathbf{R}\right]^{-1} \tag{5.17}$$

$$\mathbf{p}_{t+1} = I\,\mathbf{p}_t + \mathbf{K}_t\left[\hat{\mathbf{y}} - \mathbf{G}_t \mathbf{p}_t\right] \tag{5.18}$$

$$\mathbf{M}_{t+1} = \left[\mathbf{I} - \mathbf{K}_t \mathbf{G}_t\right]\mathbf{M}_t + \mathbf{Q} \tag{5.19}$$

Least square parameter estimation can also be applied to certain non-linear models. The essential restriction is that the models be linear in the parameters so that they can be written as linear regression models [Åström and Wittenmark, 1995]. Notice that the regressors do not need to be linear in the inputs and outputs.

5.5.3 Application of the recursive least squares parameter estimation technique to a Pump-tank system - case study

The open loop on-line identification by recursive least squares parameter estimation technique is illustrated in this example using batch processing for step input changes, on a 2-input / 2-output Pump-tank system for simulation and laboratory tests presented in Figure 4.4. The model parameter estimation was constructed from observed data.

To evaluate the disturbance influence, a random-number generator was used to produce a sequence of input changes during the identification time that could be considered as a representation of white noise.

Determining the model from the data set

From the collected data for parameter estimation, a "best" set of the converged values of the parameters (Figure 5.7) was used to estimate the elements of A and B matrices of the model, by taking the average of that set. The resultant model for Δt = 200 seconds, is represented by equation (5.20).

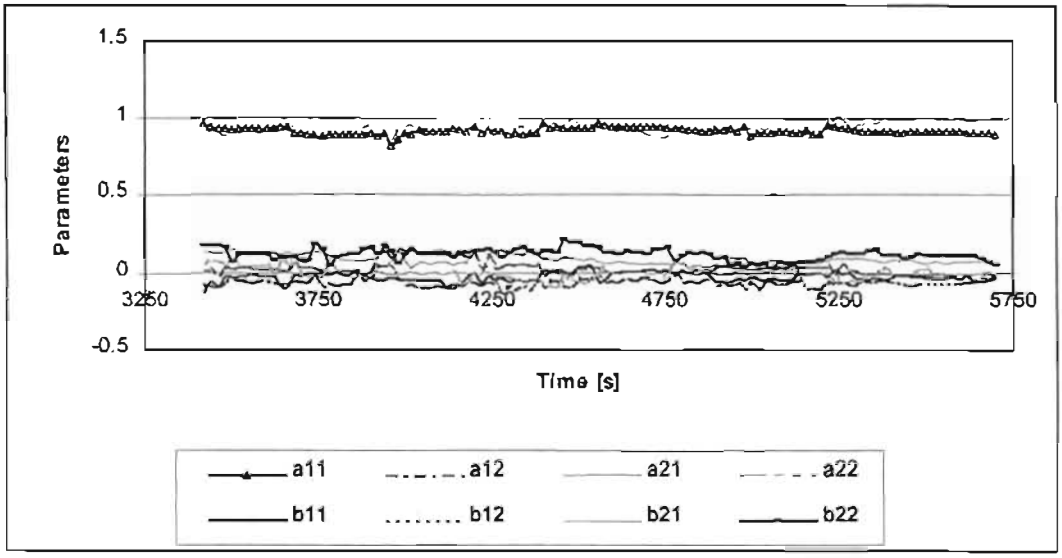


Figure 5.7 Data set for parameter estimation

$$x_{i+1} = \begin{bmatrix} 0.914 & 0.034 \\ 0.089 & 0.935 \end{bmatrix} x_i + \begin{bmatrix} -0.030 & -0.062 \\ 0.000 & 0.117 \end{bmatrix} u_i \tag{5.20}$$

Model validation

The actual situation is that a certain model structure to predict future outputs, was selected and therefore, a test to evaluate the model ability to describe the observed data is needed. A "good" model is one that is good at predicting that is, one that produces small prediction errors when compared to the observed data. Note that there is considerable flexibility in selecting various predictor functions, and this gives a corresponding freedom in defining "good" models in terms of prediction performance.

In the present case, the validation was based on integration using the dynamic matrix as in equations (5.21). The model initial value was attached to the measured sequence at the starting point of the sequence.

$$\begin{aligned} x_{1\,i+1} &= a_{11} * (x_{1\,i} - x_{1_{op}}) + a_{12} * (x_{2_i} - L_{2\,op}) + b_{11} * (u_{1\,i} - u_{1_{op}}) + b_{12} * (u_{2_i} - u_{2\,op}) + x_{1\,op} \\ x_{2\,i+1} &= a_{21} * (x_{1_i} - x_{1\,op}) + a_{22} * (x_{2\,i} - x_{2\,op}) + b_{21} * (u_{1\,i} - u_{1\,op}) + b_{22} * (u_{2\,i} - u_{2\,op}) + x_{2op} \end{aligned} \tag{5.21}$$

Where: a_{jk} and b_{jk} are the parameters of A and B matrices

x_{ji} output j at time i

u_{ji} input j at time i

x_{jop} output j at operating point

u_{jop} input j at operating point

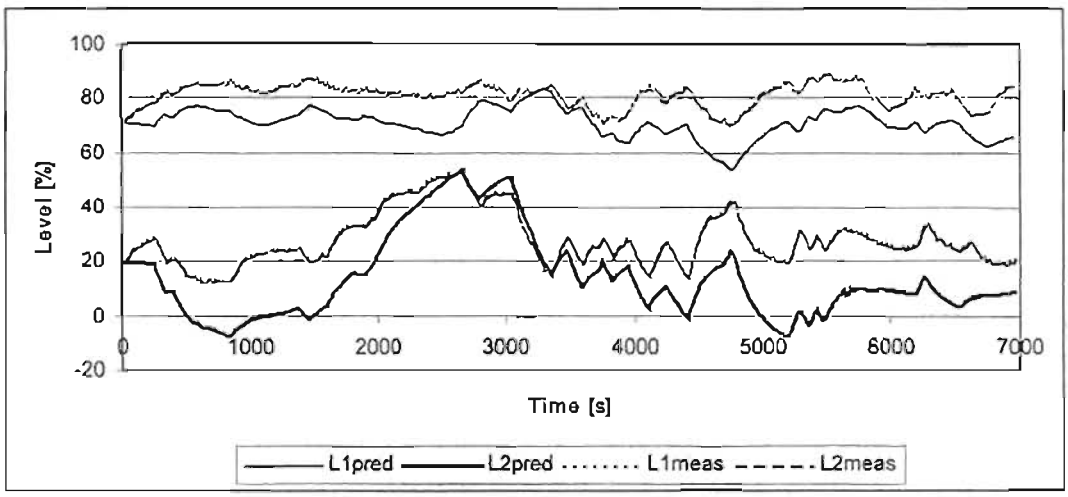


Figure 5.8 Comparison of measured and predicted outputs

Figure 5.8 shows how the predicted outputs track the observations in some cases, while in others we have an expected offset. This is because of the measurement errors, inaccuracies in the model formulation and the non-linearity. Recall that it is assumed that the process is linear in the parameters. However, it can be seen that the fit is acceptable.

Chapter 6

Adaptive Dynamic Matrix Control

6.1 Introduction

In this chapter, the implementation of the combined processes of Dynamic Matrix Control and Adaptive Control discussed in Chapters 4 and 5 respectively is presented. The performance of the resulting Adaptive Dynamic Matrix Control (ADMC) controller is expected to be significantly better than the conventional DMC. Because of the advantages of overcoming non-linearity problems and the avoidance of a rigorous model, its application in designing a controller for the Training Plant in this study was considered.

A Literature review of ADMC is presented in section 6.2. Formulation of the ADMC technique is shown in section 6.3. A regularisation approach to account for extensive computations required for parameter estimation when dealing with a high dimension system is considered in section 6.4. Finally, it is presented in section 6.5 a case study of ADMC algorithm implementation.

6.2 Literature review

As described in Chapter 4, Dynamic Matrix Control is a highly successful model predictive control scheme based on a step response model. However, being based on a linear model its performance begins to deteriorate as the process moves away from the nominal operating point. Thus, several researchers modified the original DMC algorithm making it adaptive to account for the various problems mentioned in previous chapters, since ADMC is expected to perform well even in the presence of time-varying process gain and process non-linearity. Successful applications of Adaptive DMC have been reported in the literature.

Maiti et al [1994] developed a closed-loop on-line scheme for identification of a process in terms of pseudo impulse response coefficients for a commonly encountered non-linear problem in the process industry, the control of pH. The model was subsequently used to update the dynamic matrix, making the DMC algorithm adaptive, and thus overcoming that difficulty.

Distillation is one of the most widely used separation processes in the Chemical / Petrochemical industry and its control is economically important. However, most of the industrially important distillation processes exhibit a large degree of non-linearity and the control system designed at one operating point may not perform well at another operating point. Thus updating of the process model is an important issue in distillation column control. Maiti and Saraf [1995a & 1995b] applied DMC adaptively to control the top product composition of a distillation column for both servo and regulatory problems. They also discussed the application of an adaptive DMC scheme to start-up and control of a distillation column, following a closed-loop on-line identification technique for single-input, single-output systems developed by Maiti et al [1994], to update the DMC controller model to accommodate process-model mismatch, and extended it for a multivariable system.

Zhu and Huang [1995] present an approach for self-tuning of DMC by on-line identification of the impulse model of the process based on the unit step set point change made for the closed-loop system. The proposed algorithm differs from that usually encountered in the ADMC literature since it is not implemented in the DMC algorithm. However, it has the advantage of being simple and requires little computing, and if added to conventional DMC algorithm, the resulting DMC algorithm is expected to find wider application in industry to control complicated high order processes with large time delays or varying dead time.

Aitchison and Mulholand [1997] applied an adaptive MPC to regulate the peak temperature in a heat exchanger. The system is highly distributed and conditions vary with both position and time, so control moves needed to account for the previous sequence of moves, and MPC's are well suited to these applications. Adaptation of the controller with flow changes, which have a severe effect on the system behaviour, showed some improvement in the controller performance when compared with a non-adapted controller.

As mentioned in the earlier chapters, the control of the present Training Plant is difficult because of its non-linearity, interactive nature, being multivariable and difficult to model.
A self-tuning adaptive control scheme described in section 5.3.3 was applied in parallel to DMC to generate updated controller parameters as the process moves from one operating point to another. This was done by adapting the internal model in the MPC structure (Figure 4.3), using

recursive identification of the step response coefficients following the technique developed by Deghaye, Guiamba and Mulholland [2000]. The implementation of the proposed adaptive control technique in the existing LDMC algorithm was done within a flexible SCADA system at the School of Chemical Engineering at this University, as developed by Mulholland and Prosser [1997]. This internal model control (IMC) design method is based on an assumed linear discrete process model and relates the controller settings to the model parameters in a straightforward manner. An advantage of this approach is that the initial model is easily specified, and deviations from it as the adaptation progresses have a significance, which any user is able to assess. A special technique to deal with the integrating nature of the process under study is also considered and is discussed in Chapter 7.

6.3 Formulation of an Adaptive DMC Algorithm

We desire to estimate in real-time by using a recursive least squares parameter estimation technique, the coefficients of the open loop step response, B_1, B_2, B_3, ..., B_M to be used to construct the matrices: *Open loop Matrix*, $\mathbf{B_{OL}}$, *Offset Matrix*, $\mathbf{B_0}$ and *Dynamic Matrix* **B** described in Chapter 4. The updated matrices are then applied in the DMC algorithm to compensate for non-linearity and changes in system behaviour.

Consider the process at a certain instant *M* steps *before the present*, where the contribution of the *M* past inputs *prior to that*, Δm^*_{PAST} and *M* subsequent control input steps, Δm_{PAST} up to present time will be considered as illustrated in Figure 6.1.

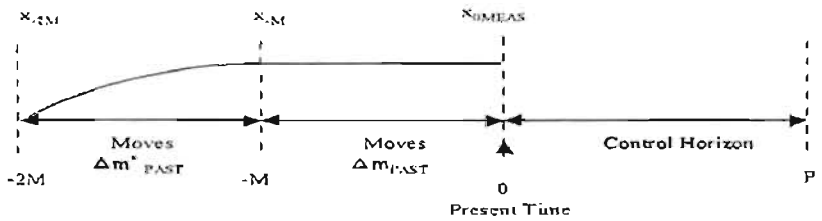


Figure 6.1 Predicted response from 2M past moves up to present time

A corrected prediction of the future trajectory from the present time up to *P* horizon is given by equation (4.5) as:

$$x = x_{0MEAS} + \left[B_{OL} - B_0\right]\Delta m_{PAST} + B\,\Delta m \tag{6.1}$$

In this formulation it is taken that $P = M$. For predictions of the outputs about the time $-M$, matrices $\mathbf{B}_0$, $\mathbf{B}_{OL}$ and $\mathbf{B}$ are defined from equation (4.3), with the bottom matrix-row of each of these as follows:

$$\begin{aligned} B'_0 &= \left[B_M \quad B_{M-1} \quad B_{M-2} \quad B_{M-3} \quad \cdots \quad B_2 \quad B_1\right] \\ B'_{OL} &= \left[B_M \quad B_M \quad B_M \quad B_M \quad \cdots \quad B_M \quad B_M\right] \\ B' &= \left[B_M \quad B_{M-1} \quad B_{M-2} \quad B_{M-3} \quad \cdots \quad B_2 \quad B_1\right] = B'_0 \end{aligned} \tag{6.2}$$

Moving the datum back to $-M$, ignoring the recent M moves, the predicted output at this time should be given by:

$$x_{-MPRED} = x_{-MMEAS} + \left[\mathbf{B}'_{OL} - \mathbf{B}'_0\right]\Delta \mathbf{m}^{\cdot}_{PAST} \tag{6.3}$$

So, it is now possible to predict the corrected output at present time as defined by equation (6.1), taking into account the most recent M input steps:

$$x_{0PRED} = x_{-MMEAS} + \left[B'_{OL} - B'_0\right]\Delta m^{\cdot}_{PAST} + B'_0\,\Delta m_{PAST} \tag{6.4}$$

Using $\mathbf{B}'_0 = \mathbf{B}'$ the predicted change of the state over the recent M steps is thus defined from equation (6.4) as:

$$\Delta x_{0PRED} = x_{0PRED} - x_{-MMEAS} = \left[B'_{OL} - B'_0\right]\Delta m^{\cdot}_{PAST} + B'_0\,\Delta m_{PAST} \tag{6.5}$$

The measured change is

$$\Delta x_{0MEAS} = x_{0MEAS} - x_{-MMEAS} \tag{6.6}$$

where x_{-MMEAS} is the measured outputs M steps ago.

Recall from the discussion of DMC in Section 4.2 that the model error e_M is given by the difference between the measured changes at present time and the predicted output changes, that is:

$$e_M = \Delta x_{0MEAS} - \Delta x_{0PRED} \tag{6.7}$$

Substituting above equation in the equation (6.5) and rearranging the measured changes are given by:

$$\left\{\Delta x_{MEAS} - \left[B'_{OL} - B'_0\right]\Delta m^{\cdot}_{PAST}\right\} = e_M + B'_0\,\Delta m_{PAST} \tag{6.8}$$

The left hand side of equation (6.8) defines corrected measured changes by $[\mathbf{B'_{OL}} - \mathbf{B'_0}]\,\Delta m^{\cdot}_{PAST}$, which is a correcting vector for the steady state and remaining transient at time M and it is expected that the true model error e_M will average **0**. Since the correction term does not contribute strongly here, we simplify the identification by using past values of B_1, B_2, B_3,...., B_M to construct $B^{\cdot}_0$ and $B^{\cdot}_{OL}$ on the left hand side. Thus the entire left-hand-side can be treated as a known "measurement".

Let us define the observations as:

$$\hat{y} = \left\{\Delta x_{MEAS} - \left[B'_{OL} - B'_0\right]\Delta m^{\cdot}_{PAST}\right\} \tag{6.9}$$

thus, equation (6.8) becomes

$$\hat{y} = e_M + B'_0\,\Delta m_{PAST} \tag{6.10}$$

where the matrix of matrices $\mathbf{B'_0}$ for m inputs, n outputs and M-step horizon is given by:

$$\mathbf{B}'_0 = \left[\begin{bmatrix} b_{M,11}\cdots b_{M,1m} \\ \vdots \quad\quad \vdots \\ b_{M,n1}\cdots b_{M,nm}\end{bmatrix}\begin{bmatrix} b_{M-1,11}\cdots b_{M-1,1m} \\ \vdots \quad\quad \vdots \\ b_{M-1,n1}\cdots b_{M-1,nm}\end{bmatrix}\cdots\begin{bmatrix} b_{1,11}\cdots b_{1,1m} \\ \vdots \quad\quad \vdots \\ b_{1,n1}\cdots b_{1,nm}\end{bmatrix}\right] \tag{6.11}$$

The n $\mathbf{b}_i$ vectors are defined as:

$$\begin{aligned} \mathbf{b}_1 &= \left[b_{M,11},\cdots,b_{M,1m},b_{M-1,11},\cdots,b_{M-1,1m},\cdots,b_{1,11},\cdots,b_{1,1m}\right]^T \\ &\vdots \\ \mathbf{b}_n &= \left[b_{M,n1},\cdots,b_{M,nm},b_{M-1,n1},\cdots,b_{M-1,nm},\cdots,b_{1,n1},\cdots,b_{1,nm}\right]^T \end{aligned} \tag{6.12}$$

Then, the observation vector (equation (6.10)) can be written as:

$$\hat{y} = \begin{bmatrix} 1 & 0 & 0 & 0 & \Delta m^T_{PAST} & 0^T & 0^T & 0^T \\ 0 & 1 & 0 & 0 & 0^T & \Delta m^T_{PAST} & 0^T & 0^T \\ \vdots & \vdots & \ddots & \vdots & 0^T & 0^T & \Delta m^T_{PAST} & 0^T \\ 0 & 0 & 0 & 1 & 0^T & 0^T & 0^T & \Delta m^T_{PAST} \end{bmatrix} \begin{bmatrix} e_{M1} \\ e_{M2} \\ \vdots \\ e_{Mn} \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix} \tag{6.13}$$

or

$$\hat{y} = G\, p \tag{6.14}$$

where

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & \Delta m^T_{PAST} & 0^T & 0^T & 0^T \\ 0 & 1 & 0 & 0 & 0^T & \Delta m^T_{PAST} & 0^T & 0^T \\ \vdots & \vdots & \ddots & \vdots & 0^T & 0^T & \Delta m^T_{PAST} & 0^T \\ 0 & 0 & 0 & 1 & 0^T & 0^T & 0^T & \Delta m^T_{PAST} \end{bmatrix}$$

and

$$p = \left[e_{M1}^T, e_{M2}^T, \ldots, e_{Mn}^T, b_1^T, b_2^T, \ldots, b_{n-1}^T, b_n^T \right]^T$$

G is an augmented matrix combining an identity matrix related to vector of model errors, e_{Mi} and a diagonal matrix with past input contribution, Δm_{PAST} in the recursive identification of the B'_0 matrix coefficients.

To estimate the contents of the parameter vector **p**, the Kalman filter is set up as (see section 5.5.2)

$$p_{t+1} = A\, p_t + K_t \left[\hat{y} - G_t\, p_t \right] \tag{6.15}$$

It is desired to force the offset errors e_{Mi} back to zero over a period, by adjusting the b_{kij} terms, but its must be allowed to move from zero temporarily whilst this happens. Thus **A** will be an identity matrix except that the first n elements on the diagonal will be zero, to give the e_{Mi} a zero base, i.e.

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.16}$$

When plant variations stop, there will be no more changes in the **p** vector.

For safety, the plant should be allowed to drift back slowly to the original measured step responses. The following model is used

$$\mathbf{p}_{t+1} = \mathbf{A}\,\mathbf{p}_t + \mathbf{B}\,\mathbf{p}_0 + \mathbf{K}_t\left[\hat{y} - \mathbf{G}_t\mathbf{p}_t\right] \tag{6.17}$$

where $\mathbf{p}_0$ is the initially proposed **p** vector and matrices **A** and **B** are defined as:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \alpha & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \alpha & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \alpha \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & (1-\alpha) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & (1-\alpha) & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & (1-\alpha) \end{bmatrix} \tag{6.18}$$

with $0 \le \alpha \le 1$ and we generally expect α to be close to 1 to find "constant" step response coefficients, but force the e_{Mi} toward zero by virtue of the first n diagonal elements being zero. Should the plant become quiescent, $\hat{y}_i$ and $\mathbf{G}_i$ move to **0** and the existing **p** may lose relevance. Then the **B** matrix slowly draws predictions towards the initial step-response $\mathbf{p}_0$.

The required recursion for the optimal Kalman gain $\mathbf{K}_i$ is thus given by equations (5.5 to 5.7) as:

$$\begin{aligned}
\mathbf{K}_i &= \mathbf{M}_i\mathbf{G}_i^T\left[\mathbf{G}_i\mathbf{M}_i\mathbf{G}_i^T + \mathbf{R}\right]^{-1} \\
\mathbf{p}_{i+1} &= \mathbf{A}_i\mathbf{p}_i + \mathbf{B}_i\mathbf{p}_0 + \mathbf{K}_i\left[\hat{\mathbf{y}} - \mathbf{G}_i\mathbf{p}_i\right] \\
\mathbf{M}_{i+1} &= \mathbf{A}_i\left[\mathbf{I} - \mathbf{K}_i\mathbf{G}_i\right]\mathbf{M}_i\mathbf{A}_i^T + \mathbf{Q}
\end{aligned} \tag{6.19}$$

As defined in Chapter 5, $\mathbf{M}_i$ is the filter covariance matrix (initially small and diagonal), $\mathbf{R}$ the expected observation error covariance (usually diagonal), $\mathbf{Q}$ the expected prediction error covariance (usually diagonal). For the present recursion, the first n elements on the diagonal matrix $\mathbf{Q}$, if small, will cause the errors e_{Mi} to reduce quickly with more rapid adjustment of the b_{kij} parameters.

At each sampling instant, the coefficients of the step response B_1, B_2, B_3, ..., B_n are updated, and then used to update $\mathbf{B'}_{OL}$, $\mathbf{B'}_0$ and $\mathbf{B'}$. In this way, the DMC controller continues to be constructed from a good local representation of the process.

6.4 Regularisation approach

If the $\mathbf{p}$ vector contains many parameters, the problem of minimising the error may be ill conditioned. Regularisation is really a general technique to solve ill-posed problems. It is of particular importance for non-linear black-box models, where many parameters are often used and it may not be possible to estimate several of them accurately [Ljung, 1999].

The identification problem as described above is of a high dimension ($M \times m \times n$ parameters). This will usually cause difficulty if the process variations are not rich in information. Thus an option for regularisation has also been provided. This is based on a weighted combination of the original step-response ($\times\ w_1$), and the same response delayed one step in time ($\times\ w_2$) (Figure 6.2). This reduces the search to $2 \times m \times n$ parameters.

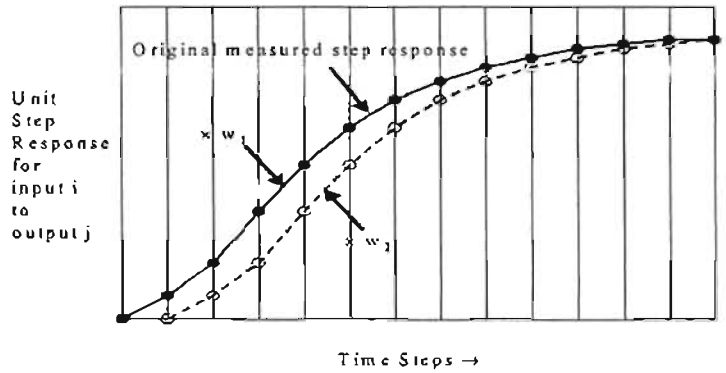


Figure 6.2: Basic functions for regularisation

The weight w_1 allows scaling of the original response, whilst the proportions of w_1 and w_2 give some ability to shift the step response in time. The original shape of the measured response is retained but moved in size and time by the proportions of w_1 and w_2. The adapted response thus becomes defined by only two parameters, rather than individual new values for every point on the response. The regularisation is easily handled in the above formulation by noting that

$$p = F\, p_r \tag{6.20}$$

where p_r is the reduced parameter vector, so that we only have to replace G with GF in the formulation. The construction of F is most easily illustrated using a 2-input / 2-output process:

$$\begin{pmatrix} e_{M1} \\ e_{M2} \\ b_{M11} \\ b_{M12} \\ b_{M-1,11} \\ b_{M-1,12} \\ \vdots \\ b_{122} \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & b^0_{M11} & b^{0SHIFT}_{M11} & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & b^0_{M12} & b^{0SHIFT}_{M12} & \cdots & 0 & 0 & 0 \\ 0 & 0 & b^0_{M-1,11} & b^{0SHIFT}_{M-1,11} & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & b^0_{M-1,12} & b^{0SHIFT}_{M-1,12} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & b^0_{122} & b^{0SHIFT}_{122} \end{bmatrix} \begin{pmatrix} e_{M1} \\ e_{M2} \\ w_{111} \\ w_{211} \\ w_{112} \\ w_{212} \\ w_{121} \\ w_{221} \\ w_{122} \\ w_{222} \end{pmatrix} \tag{6.21}$$

6. 5 Application of ADMC on Pump-tank system - case study

The simulation study was carried out on the 2-input / 2-output Pump-tank system described in Chapter 4 and shown in Figure 4.4. The conventional DMC and the ADMC algorithm were applied to control the levels L1 and L2 manipulating valves V1 and V2. Notice that the conventional DMC is referred to as LDMC.

6.5.1 Off-line simulation

The data acquisition and control action implementation tasks were performed with the on-board *convolution model*, loaded with the true measured step responses illustrated in the Figure 4.5, and the DMC algorithm with "faulty" initial step responses in which, the diagonal responses were both twice their correct magnitudes (see Table D.1 in the Appendix D). An initial operating point lying at 50 % for both manipulated variables and controlled variables was considered.

The controller design parameters M = 10, P = 10, and N = 2, were used for LDMC / ADMC algorithms. The move suppression, Λ and weight factors W were tuned and a value of 1 for both parameters proved acceptable. The performance of the DMC algorithm was compared for the case where the initial step-responses were not adapted, versus the case with adaptation. The identification rapidly converged to the true step responses through a wide range of Q and R tuning.

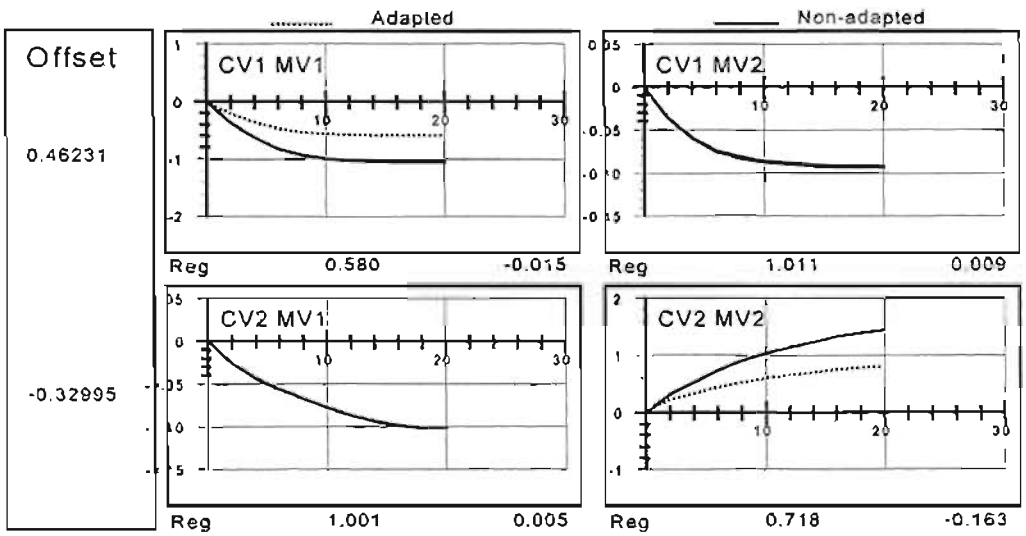


Figure 6.3 True step responses identified by regularised model 6 minutes and 49 seconds from the start of the run

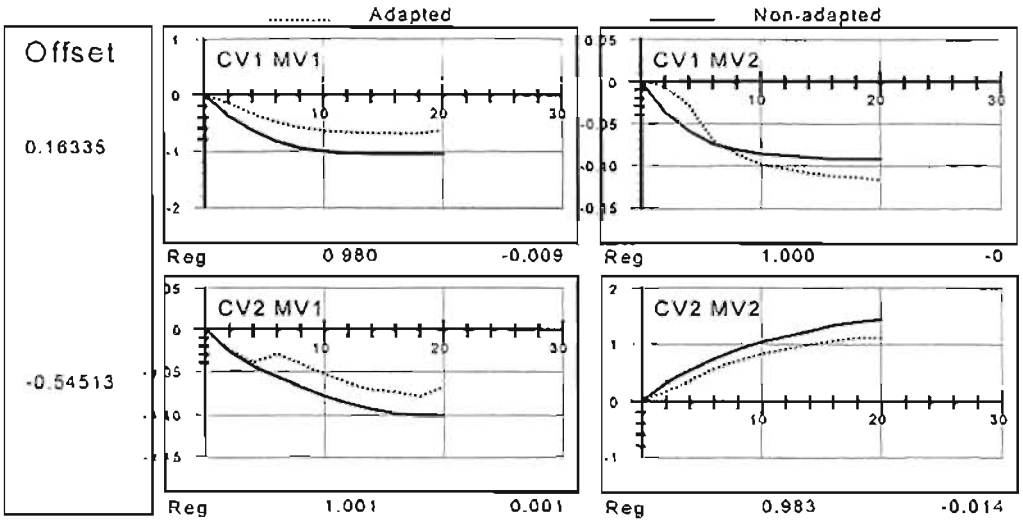


Figure 6.4 Attempt to find true step responses using unregularised model at same time as in Figure 6.3

In the tests considered here, the coefficient α (rate of going back to initial parameters vector, $\mathbf{p}_0$, discussed in section 6.3), was set at 0.9999, the model error, e_M terms at 1.0, and the expected observation error covariance, **R**, diagonal terms set at 1.0. Note that α = 0.9999 suggests an extremely slow movement towards the assumed starting forms of the responses. In practice this effect was faster than expected, possibly due to the structure of equation (6.17) used in the Kalman filter. Further, an identity matrix used for **R** was possible because of the similar input and output unit ranges (%). In Figure 6.3 the dark curve is the initial "faulty" step-response. The light dashed is the regularised result at a particular time. The equivalent result provided by the unregularised model, for the same tuning, at the same point in time (6'49" from start), is given in Figure 6.4. Because each point of the response is solved for individually, considerably more variability is encountered.

In Figure 6.5 a comparison is performed between the control performance in the non-adapted case (conventional DMC) (a), and the recursively adapted case (b) using the regularised model for changes in the set points. Although ADMC yielded improved performance over LDMC, LDMC was not totally unsatisfactory. In the adapted case there is a small overshoot and the process attains its new steady state quickly and smoothly. Thus, the controller performance is tighter and quite satisfactory with the valves working much harder. This is to be expected because the specified diagonal responses for the DMC were twice their correct sizes. Therefore control moves in the non-adapted case will be half what they should have been, giving a somewhat detuned control response in the top diagram.

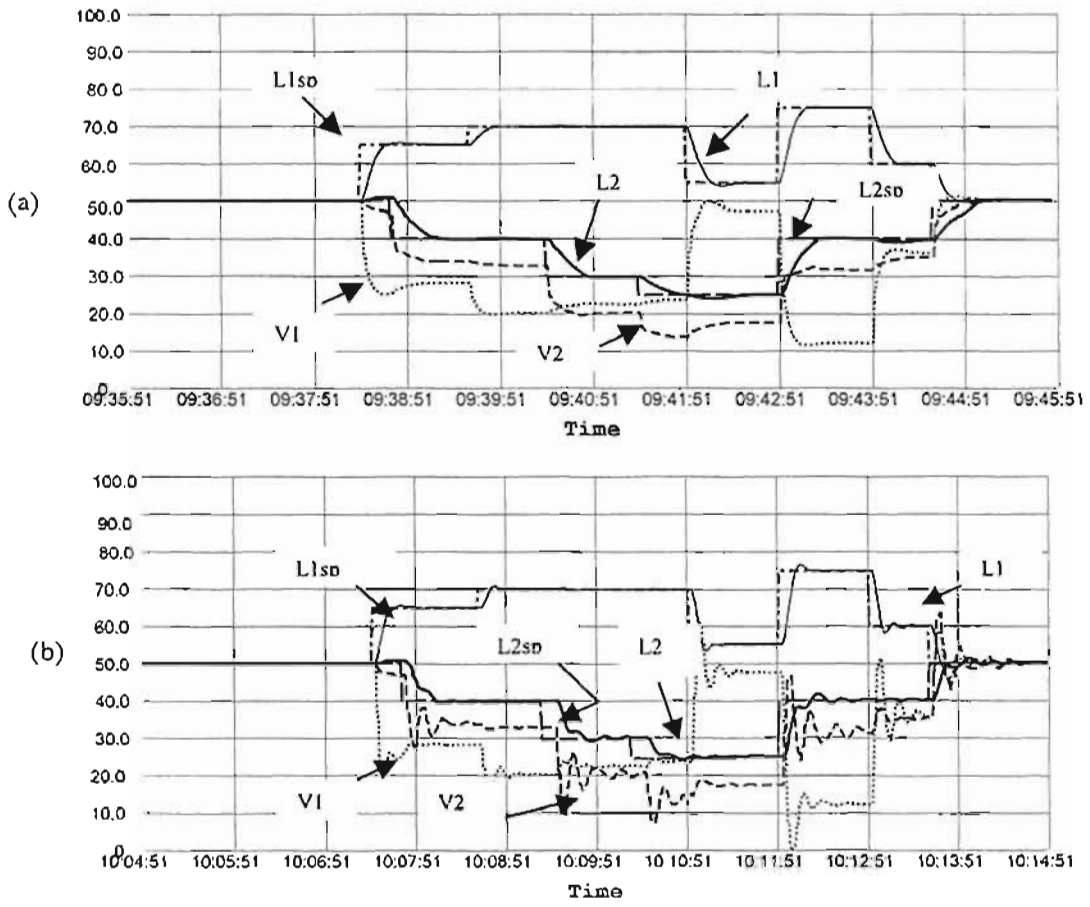


Figure 6.5 Closed loop DMC level response to set point variation for non-adapted (a) and adapted (b) cases (simulation model)

6.5.2 On-line application

So far, in Section 6.5.1 we have only considered application of the algorithm *off-line*, to a linear convolution model based on the measured process step responses. The real-time control study for the same input change sequence and same parameters as applied in the off-line simulation, was run with and without adaptation, with regularisation effective, on the Pump-tank system in the laboratory using a sampling interval of 60 seconds. However, a set point start and end of (50,30) was used instead, since the system shown in Figure 4.4 attests that (50,50) is not feasible!

Figures 6.6 and 6.7 shows the output responses to input step changes with the non-adaptation and adaptation cases respectively. The difference is not great, but careful inspection reveals the use of higher gain in the adapted case, as we would expect, because the DMC receives the updated smaller true step responses from the identifier.

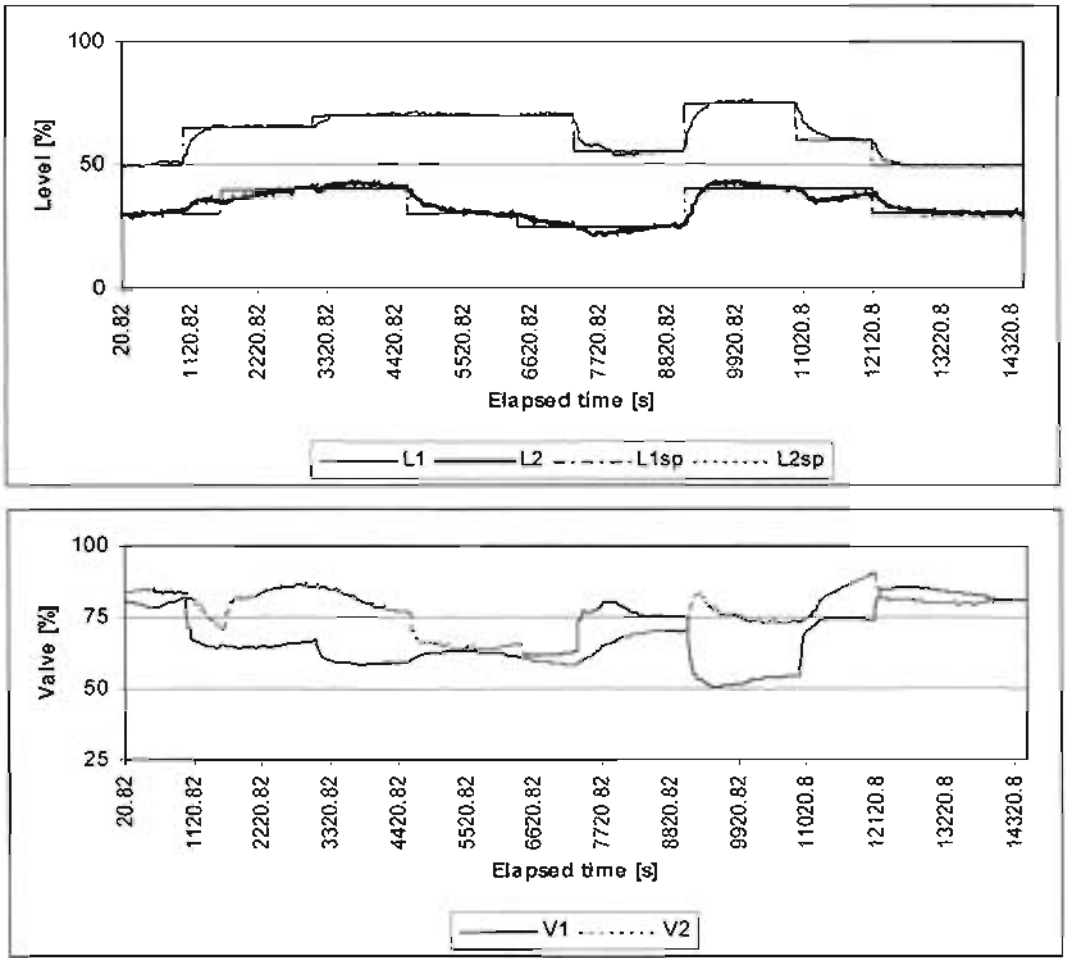


Figure 6.6 Closed loop on-line LDMC level responses to set point variation in the Pump-tank system

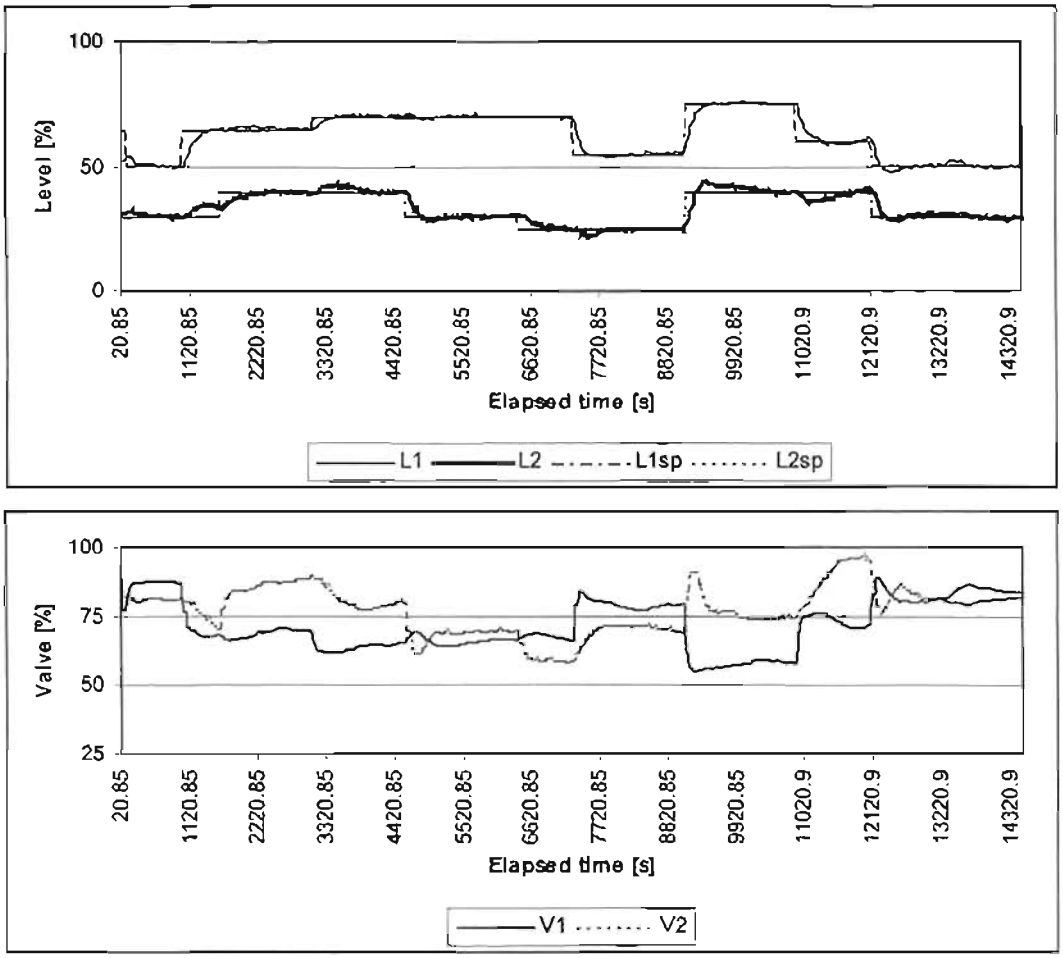


Figure 6.7 Closed loop on-line ADMC level responses to set point variation in the Pump-tank Plant

Since differences between the response curves cannot be easily distinguished in these figures, Quadratic Performance Indices (QPI) of the controlled variables, levels L1 and L2, were calculated from the logged data. These consist of the sum of the squares of the errors (difference between set point and process variable), and results are given in Table 6.1. As expected, the ADMC controller has a better performance for both levels when compared with the LDMC controller, since it presents smaller QPI values i.e. 9.24 and 6.07 for levels L1 and L2 respectively. Clearly this is only an approximate comparison, because the behaviour of each controller is determined by the definition of its objective function, which includes the control move penalisation. Thus if either of the adapted *vs* unadapted cases demanded a greater degree of movement, it would apportion terms differently with regard to the objective function.

Table 6.1 On-line LDMC and ADMC quadratic performance indices

	Conventional DMC	Adapted DMC
QPI - L1	10.92	9.24
QPI - L2	6.28	6.07

Note that the identified responses in Figures 6.8 and 6.9 move towards the true measured diagonal responses (non-adapted responses are 2x the initial responses) set in the DMC. Some spurious effect introduced by the plant causes a significant change to be registered in an off-diagonal term. This can be protected in practice by increasing the response time of the identified responses by lowering the prediction error, Q in the Kalman filter and by constraining the allowed range of variation.

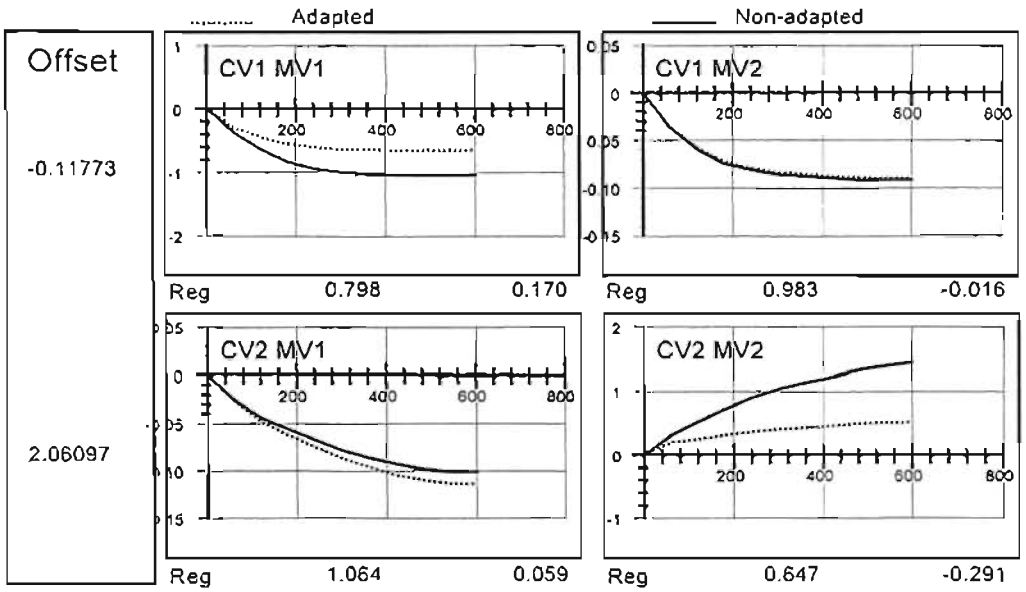


Figure 6.8 Identified step responses using regularised model in the Pump-tank Plant, 1 hour from the start of the run

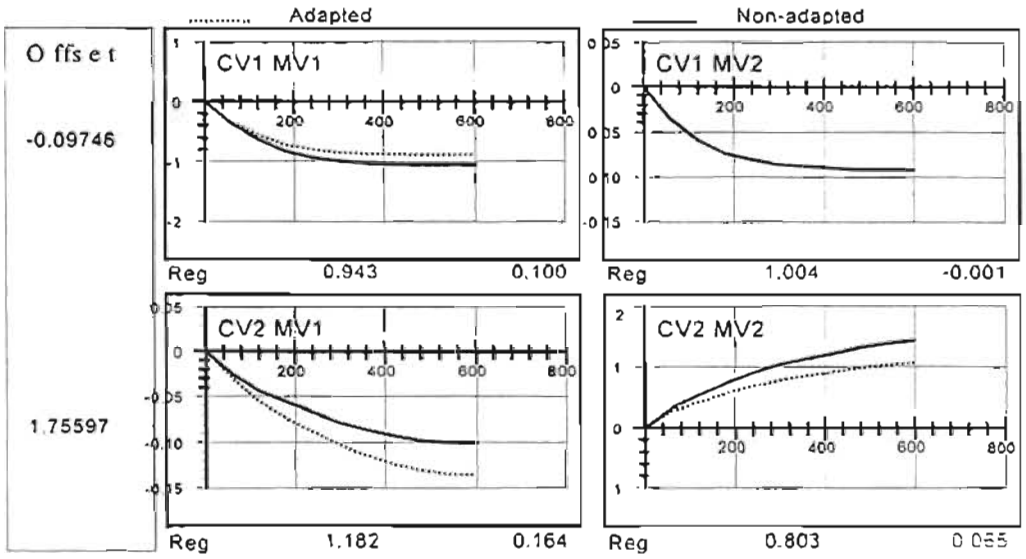


Figure 6.9 Identified step responses using regularised model in the Pump-tank Plant, 2 hours 37 minutes and 30 seconds from the start of the run

Chapter 7

Integrating Adaptive Dynamic Matrix Control

7.1 Introduction

The Adaptive DMC controller discussed in the above chapter proved adequate for multivariable and non-linear systems. However, when the LDMC algorithm is used for control of processes containing integrating process units, steady-state offsets occur for sustained changes. This offset it is not acceptable for most applications.

To overcome the problem of the integrating nature of the present Training Plant, an integrating ADMC approach was developed and is discussed in this chapter. A literature review covering integrating processes is presented in the section 7.2. Section 7.3 shows the formulation of integrating DMC while, Integrating ADMC is discussed in the section 7.4. Finally, application results of Integrating ADMC on the 2-input / 2-output sub-system defined on the Training Plant which include off-line and on-line simulation, are presented in section 7.5.

7.2 Integrating processes overview

Integrating processes are those that produce a ramp change in the output for a step change in the input (Figure 7.1). This results from the process unit's material or energy imbalance [Gupta, 1998], and they are commonly present in chemical industry. "Non-self regulatory" level processes are typical examples of integrating process units.

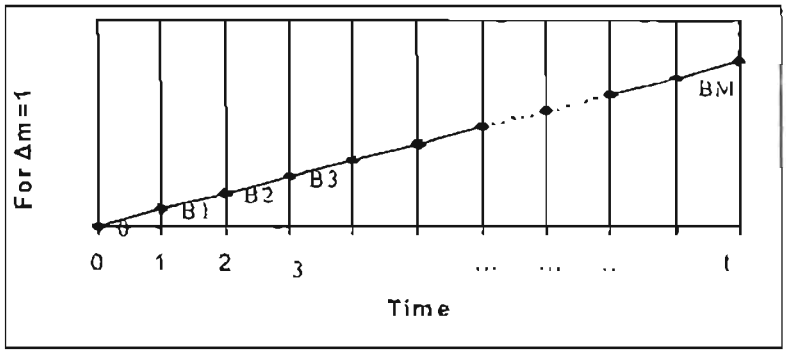


Figure 7.1 Typical integrating output response to a unit step input

Integrating processes produce a steady-state offset when controlled by a standard DMC algorithm, which assumes responses reach steady-state on the final point. For applications where the performance objective of the process control is to ensure that the controlled variables remain very close to their set points, these steady-state offsets are not acceptable. The control of the outputs of integrating process units needs to be considered along with the control of other process outputs and in the case of constrained variables, all of the constraints need to be considered simultaneously. This allows the determination of the true optimum of the optimisation problem that needs to be solved at every control instant.

Integrating process studies have appeared in the literature. Lee, Morari and Garcia [1994] describe a MPC technique based on step response parameters for systems of stable and / or integrating dynamics, developed using state-space estimation techniques. The standard step response model is extended, in this technique, so that integrating systems can be treated within the same framework. A ramp disturbance is eliminated by introducing a double integrator in the controller.They showed that the optimal observer can be calculated by solving a Riccati equation of significantly lower dimension.

Gupta [1998] presents an alternative approach to eliminate the steady-state offsets that are encountered when dealing with integrating process units. The proposed approach does not require the formulation of the MPC problem in the state-space form and because of this advantage, it can be implemented directly in the step response formulation of the DMC algorithm. This scheme takes advantage of the fact that the predicted response due to past inputs is a straight line passing through the output at the current control instant. Thus, the slope of the predicted response is determined from the slope of the output trajectory between the current and the previous control instants. Note that this slope includes the effect of unmeasured disturbances and any model mismatch that may be present. This approach allows the consideration of all inputs, outputs and constraints in the one optimisation problem.

As mentioned above, the Training Plant under study has integrating behaviour. Predicted responses of level due to past input steps resulted in ramp changes. Thus, to control this system, we follow the development made by Gupta [1998], for control of an integrating process using DMC by implementing the changes directly in the step response in the DMC algorithm, and this is described below.

7.3 Integrating LDMC formulation

As discussed in the Chapter 4, the step responses of an m-input / n-output system can be represented by a series of matrices B_1, B_2, B_3, ... B_P (scaled for a unit input step). The position (i,j) in each matrix B_t is a point on the trajectory at time t for the i th output as it responds to the j th input. It is possible to indicate an integrating relationship by unequal corresponding elements in the final two matrices, B_P and B_{P-1}, and consider that this final gradient continues indefinitely from this point onwards as a result of the integration. Define

$$\Delta B = B_P - B_{P-1} \tag{7.1}$$

Now let Δm_t be the vector of input moves (changes) made at time t, with x_t the vector of outputs at time t. We also make use of a moving frame of reference for time in which $t=0$ represents the present time. Thus, for integrating processes equation (4.3) can be written as equation (7.2).

Recall that the vector of vectors x_{0PRED} contains P identical predictions of the output vector at the present time $t=0$. The vector x_{PRED} contains predictions of the output vector at P points on the future trajectory, as contributed to by the past M control moves, and the future P control moves.

$$
\begin{pmatrix} x_{0PRED} \\ x_{PRED} \end{pmatrix} =
\begin{pmatrix} x_{0PRED} \\ x_{0PRED} \\ x_{0PRED} \\ x_{0PRED} \\ \vdots \\ x_{0PRED} \\ \hdashline x_1 \\ x_2 \\ \vdots \\ x_M \\ x_{M+1} \\ \vdots \\ x_P \end{pmatrix}
=
\left[\begin{array}{c:ccccccc:ccccccc}
0 & B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_2 & B_1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
\hdashline
\Delta B & B_M + \Delta B & B_M & B_{M-1} & B_{M-2} & \cdots & B_3 & B_2 & B_1 & 0 & 0 & 0 & 0 & \cdots & 0 \\
2\Delta B & B_M + 2\Delta B & B_M + \Delta B & B_M & B_{M-1} & \cdots & B_4 & B_3 & B_2 & B_1 & 0 & 0 & 0 & \cdots & 0 \\
3\Delta B & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
\ddots & \vdots & \vdots & \vdots & \vdots & \cdots & B_M & B_{M-1} & B_{M-2} & B_{M-3} & \cdots & B_1 & 0 & \cdots & 0 \\
\ddots & \ddots & \vdots & \vdots & \vdots & \cdots & B_M + \Delta B & B_M & B_{M-1} & B_{M-2} & \cdots & B_2 & B_1 & \cdots & 0 \\
(P-1)\Delta B & \ddots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
P\Delta B & B_M + P\Delta B & \ddots & \ddots & \vdots & \cdots & B_M + 2\Delta B & B_M + \Delta B & B_M & B_M & B_M & B_M & B_{M-1} & \cdots & B_1
\end{array}\right]
\begin{pmatrix} \sum_{i=-\infty}^{-M} \Delta m_i \\ \Delta m_{-M+1} \\ \Delta m_{-M+2} \\ \Delta m_{-M+3} \\ \Delta m_{-M+4} \\ \vdots \\ \Delta m_0 \\ \hdashline \Delta m_1 \\ \Delta m_2 \\ \vdots \\ \Delta m_M \\ \Delta m_{M+1} \\ \vdots \\ \Delta m_P \end{pmatrix}
\qquad (7.2)
$$

P copies of this offset error are made by comparison of x_{0PRED} with a vector x_{0MEAS} containing P copies of the present output measurement, and are used to correct our future predictions as defined by equation (4.4) as follows:

$$\mathbf{x}_{CL} = \mathbf{x}_{0MEAS} + \left[\mathbf{B}_{OL} - \mathbf{B}_0\right]\Delta\mathbf{m}_{PAST} + \mathbf{B}\Delta\mathbf{m} \tag{7.3}$$

where Δm_{PAST} is considered to include the summation term $\sum_{i=-\infty}^{-M}\Delta m_i$ which accumulates all moves older than M steps back from the present time. The matrices B_0, B_{OL} and B are clearly top-left, bottom-left and bottom-right in the above structure. Usually we choose M to be large enough to ensure that all step responses are at steady state at this point, but of course this is not possible for an integrating system, which gives a non-zero ΔB. For such a system we choose M where all slopes have become constant. The terms in the first column will be non-zero for an integrating system, and act on a non-zero absolute displacement of the control action (sum of all moves), to add a steady ramp to all integrating outputs of the system (see equation (7.2)).

It is assumed that our predictions of x_{0PRED} and x_{PRED} will only be in error by steady-state offsets emanating from earlier than M steps before the present time (see Figure 6.1). This is not really the case for integrating systems, where integration of an unobserved control action may have begun somewhat earlier, and may not be included in the compensatory term

$$\left(\sum_{i=-\infty}^{-M}\Delta m_i\right)$$

Moreover, any accumulated compensation will not be exact. These factors mean that an unaccounted ramp may already be in effect on the outputs. This has the potential to cause steady-state offset in the control. Thus the following technique has been developed to deal with this situation. It is based on a long-term identification of a consistent gradient error between predictions and observations. This gradient is then superimposed on the open-loop predictions. Note that the prediction of changes in the output over the last M steps is defined in the Chapter 6 by equation (6.5) as:

$$\Delta\mathbf{x}_{PRED} = \left[\mathbf{B}'_{OL} - \mathbf{B}'_0\right]\Delta\mathbf{m}^{\bullet}_{PAST} + \mathbf{B}'\,\Delta\mathbf{m}_{PAST} \tag{7.4}$$

The equivalent measurement is

$$\Delta\mathbf{x}_{MEAS} = \mathbf{x}_{0MEAS} - \mathbf{x}_{-MMEAS} \tag{7.5}$$

A simple filter then seeks a consistent error e_M

$$(\mathbf{e}_M)_0 = \beta(\Delta \mathbf{x}_{MEAS} - \Delta \mathbf{x}_{PRED})_0 + (1-\beta)(\mathbf{e}_M)_{-1} \tag{7.6}$$

and β=0.05 has proved adequate in this case. Construct a "gradient" correction vector for the future P predictions up to the horizon as follows

$$\Delta \mathbf{x}_{GRAD} = \begin{pmatrix} \frac{1}{M}(\mathbf{e}_M)_0 \\ \frac{2}{M}(\mathbf{e}_M)_0 \\ \frac{3}{M}(\mathbf{e}_M)_0 \\ \vdots \\ \frac{P-1}{M}(\mathbf{e}_M)_0 \\ \frac{P}{M}(\mathbf{e}_M)_0 \end{pmatrix} \tag{7.7}$$

Then the corrected prediction of the future output up to the horizon P becomes

$$x = x_{0MEAS} + [\boldsymbol{B}_{OL} - \boldsymbol{B}_0]\Delta m_{PAST} + \Delta x_{GRAD} + \boldsymbol{B}\Delta m \tag{7.8}$$

In DMC, the optimal set of future moves Δm is solved on each step.

7.4 Integrating Adaptive DMC formulation

A scheme for adapting the internal convolution model of a LDMC, by closed-loop recursive identification of the step response coefficients in real-time by following the methodology presented in the Chapter 6 is developed in this section for the integrating case.

Recall that in LDMC the optimal control move at any step is computed as dependent on the matrices $\boldsymbol{B}_{OL}$, $\boldsymbol{B}_0$ and $\boldsymbol{B}$. Moreover, these are constructed from the basic step-response matrices $B_1, B_2, B_3, \ldots, B_M$, which are thus the target of our real-time identification.

For an integrating process, the defined bottom matrix-row of each of the matrices $\boldsymbol{B}_{OL}$, $\boldsymbol{B}_0$ and $\boldsymbol{B}$ which are analogous to equation (6.2) are obtained from the structure in equation (7.2) are:

$$\begin{aligned}
\mathbf{B}_0' &= [\ 0 \quad B_M \quad B_{M-1} \quad B_{M-2} \quad B_{M-3} \quad \cdots \quad B_2 \quad B_1\ ] \\
\mathbf{B}_{OL}' &= [\ M\Delta B \quad B_M + M\Delta B \quad B_M + (M-1)\Delta B \quad B_M + (M-2)\Delta B \quad B_M + (M-3)\Delta B \quad \cdots \quad B_M + \Delta B \quad B_M\ ] \\
\mathbf{B}' &= [\ 0 \quad B_M \quad B_{M-1} \quad B_{M-2} \quad B_{M-3} \quad \cdots \quad B_2 \quad B_1\ ] = \mathbf{B}_0'
\end{aligned} \tag{7.9}$$

Moving the datum back to -*M*, the corrected prediction of the *present* state following equation (6.4) is thus given by

$$x_{0PRED} = x_{-MMEAS} + \left[\mathbf{B}'_{OL} - \mathbf{B}'_0\right]\Delta\mathbf{m}^{\bullet}_{PAST} + \mathbf{B}'_0\Delta\mathbf{m}_{PAST} \tag{7.10}$$

where $[\mathbf{B}'_{OL} - \mathbf{B}_0]\Delta\mathbf{m}^{\bullet}_{PAST}$ is a correction for the steady-state offset and the remaining transient at $-M$.

Notice that both $\Delta m^{\bullet}_{PAST}$ and Δm_{PAST} have the extra summation term shown in equation 7.2. In the case of $\Delta m^{\bullet}_{PAST}$ this term will clearly only sum moves until time $-2M$. Using $\boldsymbol{B}'_0 = \boldsymbol{B}'$, the predicted change of the state over the recent M steps is thus

$$\Delta x_{0PRED} = x_{0PRED} - x_{-MMEAS} = \left[\mathbf{B}'_{OL} - \mathbf{B}'_0\right]\Delta\mathbf{m}^{\bullet}_{PAST} + \mathbf{B}'_0\Delta\mathbf{m}_{PAST} \tag{7.11}$$

The measured change is $\Delta x_{0MEAS} = x_{0MEAS} - x_{-M\,MEAS}$ where $x_{-M\,MEAS}$ is the measured output M steps ago. The model error is $e_M = \Delta x_{0MEAS} - \Delta x_{0PRED}$ so

$$\Delta x_{0MEAS} = e_M + \left[\mathbf{B}'_{OL} - \mathbf{B}'_0\right]\Delta\mathbf{m}^{\bullet}_{PAST} + \mathbf{B}'_0\,\Delta\mathbf{m}_{PAST} \tag{7.12}$$

Any steady gradient error is not included in this "model error" for parameter estimation purposes, so we strip off the filtered version of e_M

$$\Delta x_{0MEAS} - \left(e_M\right)_0 = e_M + \left[\mathbf{B}'_{OL} - \mathbf{B}'_0\right]\Delta\mathbf{m}^{\bullet}_{PAST} + \mathbf{B}'_0\Delta\mathbf{m}_{PAST} \tag{7.13}$$

A non-zero $\left(e_M\right)_0$ term when the system is otherwise near steady-state (average condition) is the clearest indication of a steady gradient error due to unaccounted integration – ie. an error in the accumulation term $\sum_{i=-\infty}^{-M}\Delta m_i$.

The matrix of matrices $\boldsymbol{B}'_0$ for m inputs, n outputs and an M-step horizon is given by (similar to equations (6.11) and (6.12):

$$B_0' = \begin{bmatrix} \begin{bmatrix} b_{M,11} \cdots b_{M,1m} \\ \vdots \quad\quad \vdots \\ b_{M,m1} \cdots b_{M,mm} \end{bmatrix} \begin{bmatrix} b_{M-1,11} \cdots b_{M-1,1m} \\ \vdots \quad\quad \vdots \\ b_{M-1,m1} \cdots b_{M-1,mm} \end{bmatrix} \cdots \begin{bmatrix} b_{1,11} \cdots b_{1,1m} \\ \vdots \quad\quad \vdots \\ b_{1,m1} \cdots b_{1,mm} \end{bmatrix} \end{bmatrix}$$

we define

$$b_1 = \left[b_{M,11}, \cdots, b_{M,1m}, b_{M-1,11}, \cdots, b_{M-1,1m}, \cdots, b_{1,11}, \cdots, b_{1,1m} \right]^T$$
$$\vdots$$
$$b_n = \left[b_{M,m1}, \cdots, b_{M,mm}, b_{M-1,m1}, \cdots, b_{M-1,mm}, \cdots, b_{1,m1}, \cdots, b_{1,mm} \right]^T$$

Then,

$$e_M + B_0' \Delta m_{PAST} = \begin{bmatrix} 1 & 0 & 0 & 0 & \Delta m_{PAST}^T & 0^T & \cdots & 0^T \\ 0 & 1 & 0 & 0 & 0^T & \Delta m_{PAST}^T & \cdots & 0^T \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 1 & 0^T & 0^T & \cdots & \Delta m_{PAST}^T \end{bmatrix} \begin{bmatrix} e_{M1} \\ e_{M2} \\ \vdots \\ e_{Mn} \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix} \tag{7.14}$$

$$= G_1 \left(\Delta m_{PAST} \right) p$$

where

$$p = \begin{bmatrix} e_{M1} \\ e_{M2} \\ \vdots \\ e_{Mn} \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix} \tag{7.15}$$

Similarly other terms in (7.12) are rearranged to obtain

$$B_0' \Delta m^*_{PAST} = G_2 \left(\Delta m^*_{PAST} \right) p \tag{7.16}$$

and

$$\mathbf{B}'_{UL}\Delta\mathbf{m}^{*}_{PAST} = \mathbf{G}_3\left(\Delta\mathbf{m}^{*}_{PAST}\right)\mathbf{p} \tag{7.17}$$

Then (7.13) can be written as

$$\left(\Delta\mathbf{x}_{0MEAS} - \left(\mathbf{e}_M\right)_0\right) = \mathbf{G}\left(\Delta\mathbf{m}_{PAST}, \Delta\mathbf{m}^{*}_{PAST}\right)\mathbf{p} \tag{7.18}$$

where G is the observation matrix. Equation (7.18) shows that for integrating processes, $\mathbf{G}$ is a function of all past moves from $-2M$ up to present time, $\Delta\mathbf{m}_{PAST}$ as well as $\Delta\mathbf{m}^{*}_{PAST}$ which takes into account all accumulated moves older than $-2M$. For non-integrating processes $\mathbf{G}$ is only a function of the past moves from $-M$ up to present time, $\Delta\mathbf{m}_{PAST}$ (see equation 6.13). Thus

$$\mathbf{G}\left(\Delta\mathbf{m}_{PAST}, \Delta\mathbf{m}^{*}_{PAST}\right) = \mathbf{G}_1\left(\Delta\mathbf{m}_{PAST}\right) - \mathbf{G}_2\left(\Delta\mathbf{m}^{*}_{PAST}\right) + \mathbf{G}_3\left(\Delta\mathbf{m}^{*}_{PAST}\right) \tag{7.19}$$

For implementation of the Kalman filter interpretation, consider the A and B matrices defined by equation (6.18) and the parameter vector P is thus given by

$$\mathbf{p}_{i+1} = \mathbf{A}\,\mathbf{p}_i + \mathbf{B}\,\mathbf{p}_0 + \mathbf{K}_i\left[\left(\Delta x_{0MEAS}\left(e_M\right)_0\right)_i - \mathbf{G}_i\mathbf{p}_i\right] \tag{7.20}$$

such that, as defined in that chapter, $0 \le \alpha \le 1$, and α is set close to 1 to find "constant" step-response coefficients, but force the e_M, toward zero by virtue of the first n diagonal elements being zero. Should the plant become quiescent, $\left(\Delta x_{0MEAS} - \left(\mathbf{e}_M\right)_0\right)_i$ and $\mathbf{G}_i$ move to 0 and the existing $\mathbf{p}$ may lose relevance. Then the B matrix slowly draws predictions towards the initial step-responses p_0.

Notice that for integrating processes the vector of known measurements is defined by:

$$\hat{\mathbf{y}} = \left(\Delta\mathbf{x}_{0MEAS} - \left(\mathbf{e}_M\right)_0\right)_i \tag{7.21}$$

The required recursion, including solution for the optimal Kalman gain K_i is given by equation (6.19) and the regularisation approach described in section 6.4 is also applied to reduce the problem dimension.

Finally, since the identified B'_n matrix contains all of the fundamental step-response matrices, the various matrices used in the DMC can be updated in real-time – i.e., it is an adaptive control. Extracts of the Integrating ADMC algorithm from the SCADA system are presented in Appendix E.

7.5 Integrating ADMC application to a 2-input / 2-output sub-system of the Training Plant – case study

Application of the proposed Integrating LDMC / ADMC was carried out on the Training Plant described in Chapter 2 and schematically presented in Figure 2.2. However as mentioned in section 2.5, only a 2-input / 2-output sub-system of the Training Plant shown in Figure 2.4 was simulated.

Taking into account that Dynamic Matrix Control is based on the step response, the Training Plant was operated to get the open-loop step responses for the proposed 2-input / 2-output system in order to apply them in DMC algorithm. The time taken for the system to reach constant slopes in all step responses after disturbances was approximately 50 seconds. It was decided to use a time interval, Δt, of 10 seconds and thus have a steady-state horizon of 5 steps. The step responses that were used for the experiments are illustrated in Figure 7.2, which shows the integrating nature of this system since the responses become steady ramps.

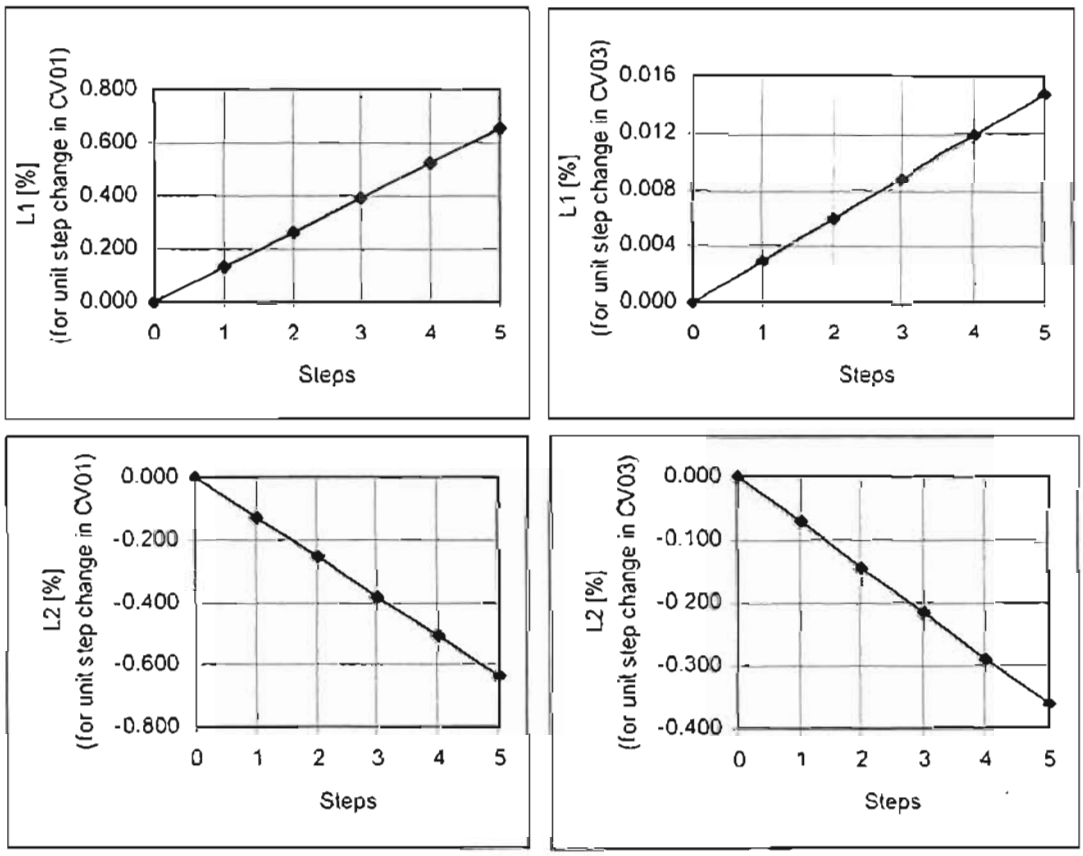


Figure 7.2 Unit step responses for the 2-input / 2-output sub-system of the Training Plant (M = 5)

Each controller test was run for the same defined sequence of set point changes. In off-line tests DMC was run with the true model in some cases and with "faulty" initial step responses in others. The "faulty" responses were set with diagonal response (1,1) at 2 times the correct responses, and (2,2) at ½ the correct response (see Table D.2 in the Appendix D). All tests on the plant were with an initial mismatched model in the DMC.

For the tests the following controller parameters were used: smoothing coefficient *in gradient Feedback*, $\beta = 0.05$ (see equation (7.6)), optimisation horizon, $P = 5$ and control moves, $N = 2$. The observation error covariance, R and predicted error covariance Q were equal to one, while for the B_i diagonal in equation (6.18), α was set to 0.9999. Tuned move suppression and weight factors that gave satisfactory controller performance, were $\Lambda = 1$ and $W = 100$ for both off-line and on-line simulations. Notice that all runs either off-line or on-line were with regularised identified responses (see section 6.4).

7.5.1 Off-line simulation

In preparation for the on-line controller software commissioning and to experiment with tuning parameters, an off-line test was designed for use with the on-board model of the Scad95 program that implemented the controller software. The program was written for Windows 95 using Microsoft's Visual C++ and is designed to work with on-line systems, but it also accepts inputs and outputs from the off-line models that are coded into the program. These are convolution models based on the step responses, with a fine-step integration (e.g. 1/10 DMC step) [Mulholand et al, 2001].

With the same parameters in both algorithms, LDMC and ADMC controller performances were compared taking into account the effect of several factors as follows, with initial and end operating points lying at (50,50).

Integrating compensation

As described in the above sections, the integrating compensation was included in the DMC software since conventional DMC it is not designed for dealing with integrating processes and as we know the present Training Plant presents such behaviour, beyond being non-linear. Recall that the integrating compensation includes factors like: *gradient feedback*, *accumulated moves* from past inputs earlier then M steps before present time, and *extended slopes* (see section 7.3). So, with the parameter values above, the integrating effect on the LDMC as well as Adaptive DMC were tested considering two distinct cases:

a) Simulation without integrating compensation and
b) Simulation with integrating compensation

We discuss each case as follows:

Case a: Simulation without integrating compensation

Figure 7.3 shows the control of the true integrating model using LDMC and ADMC controllers without integrating compensation. As expected, LDMC controller performance was bad, and led to excessive valve work and oscillations in the output responses due to the unaccounted integrating nature of the process (Figure 7.3 (a)). However, relatively better behaviour was shown by Adapted DMC, although the controller required lengthy periods to reduce oscillations apparently arising from the integrating nature of the process. This was reflected in the output responses as illustrated in Figure 7.3 (b).

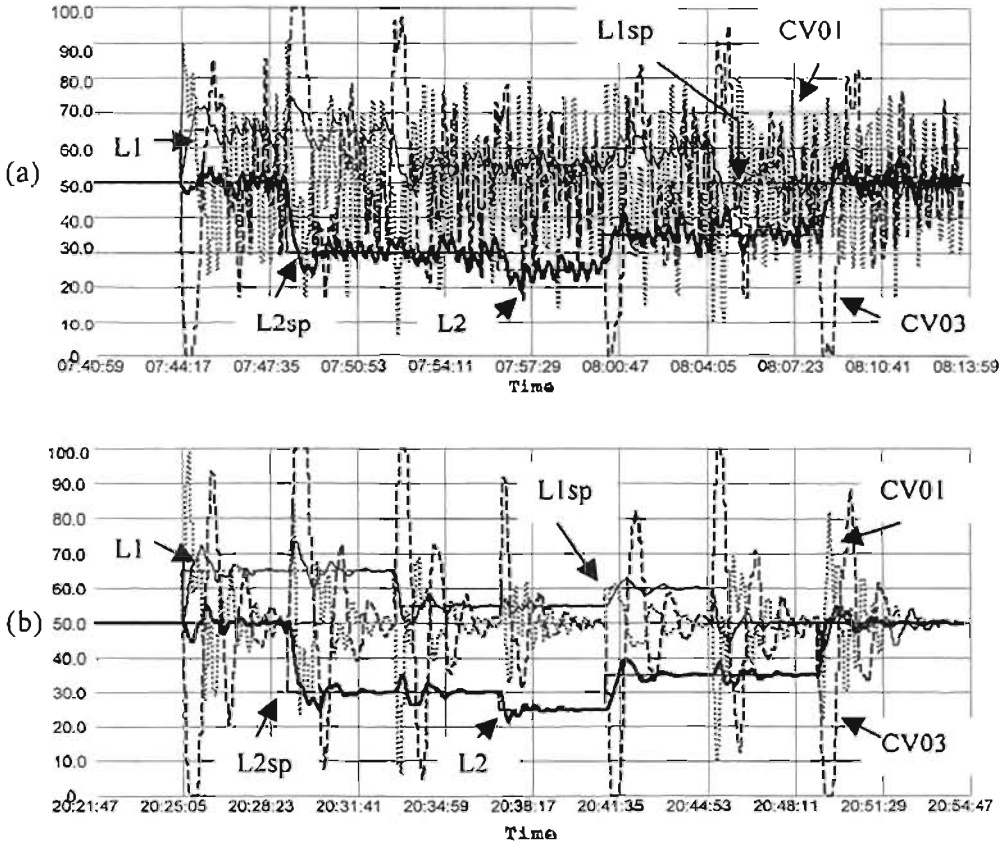


Figure 7.3 Closed loop LDMC (a) and ADMC (b) using true process model without integrating compensation, N = 2, Λ = 1, W = 100

Case b: Simulation with integrating compensation

The need for special DMC design when dealing with integrating processes is shown in this case, by running both algorithms with integrating compensation. Thus, runs with the true model in the DMC and ADMC under the influence of integrating compensation showed improved controller performances. Figure 7.4 shows the responses obtained by standard DMC. Clearly the integration in the process has been effectively dealt with.

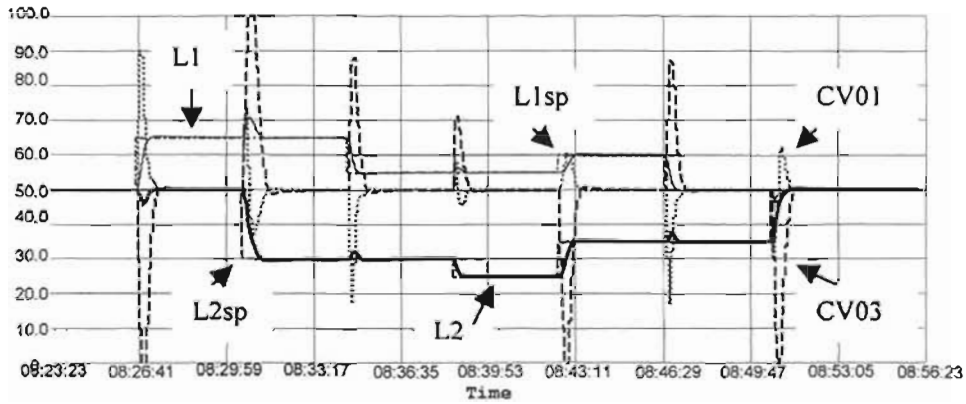


Figure 7.4 Closed loop LDMC using true process model with integrating compensation, N = 2, λ = 1, W = 100

The robustness of the DMC controller was further tested by incorporating an error factor in two of the four partitioned matrices of the system dynamic matrix, that is, diagonal terms were set with, the first one twice and the second half their correct values. The LDMC controller performance deteriorated as can be seen in Figure 7.5 (a), while the ADMC controller showed good behaviour (Figure 7.5 (b)). This can be explained as follows: with small or zero process model mismatches unadapted LDMC performs well as shown by Figure 7.4. When big process model errors occur, the unadapted linear controller becomes inadequate (Figure 7.5 (a) and updated parameters are necessary with changes in the operating point (compare 7.5 (b)).

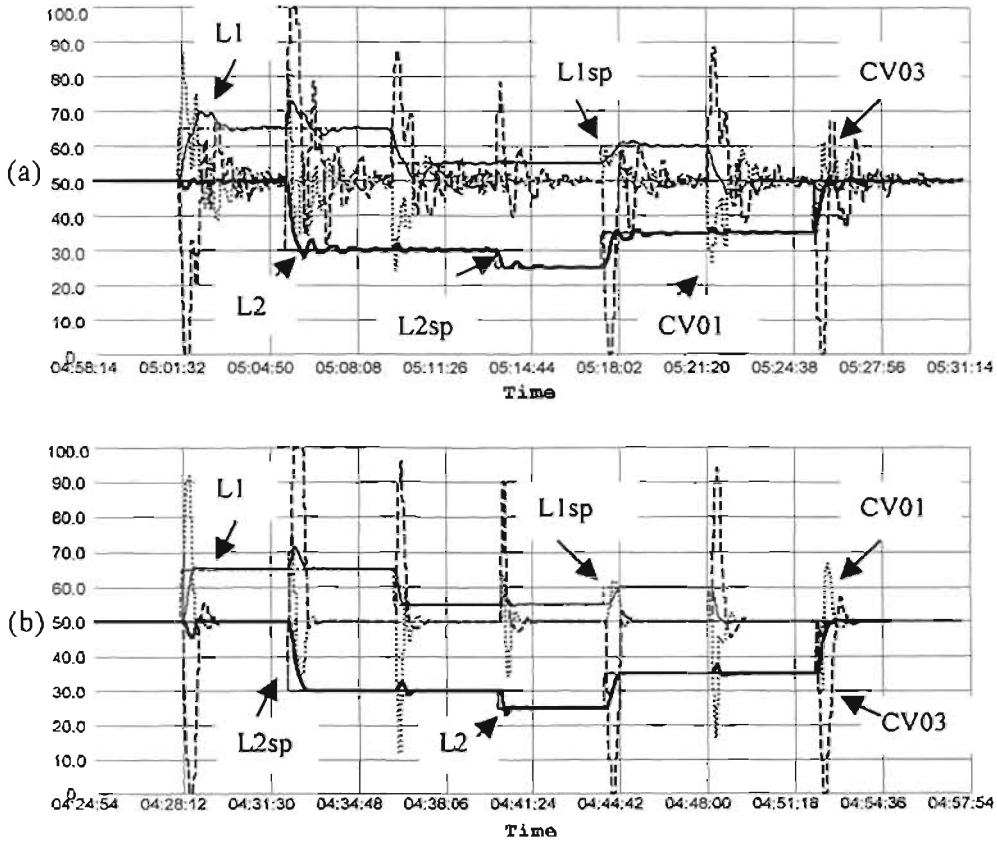


Figure 7.5 Closed loop LDMC (a) and ADMC (b) with integrating compensation and process-model mismatch. N = 2, λ = 1, W = 100

The identified responses in the ADMC run moved towards the true diagonal responses set in the model as illustrated in Figure 7.6 (dotted lines on the diagonal are closed to the true response curves).

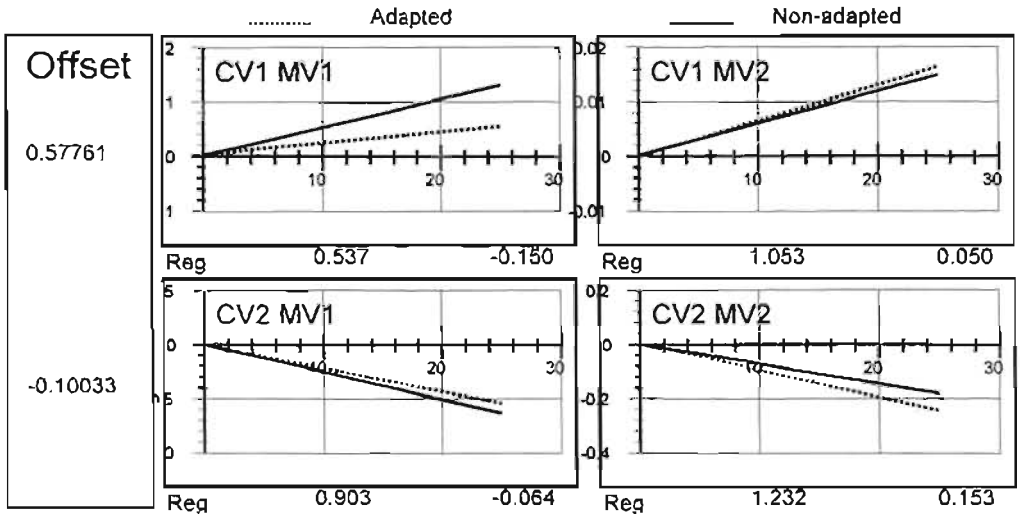


Figure 7.6 Identified step responses using simulation model, 7 minutes and 10 seconds from the start of the run

Gradient feedback and accumulated moves compensation

The influence of different factors used in the integrating compensation was also considered for both algorithms. It was found that runs with *gradient feedback* applied to *extended slopes* without accumulated moves resulted in the process instability, with severe oscillations resulting from the approach to bang-bang control in the movement of the valves. Recall that the accumulated moves represent the contribution of all moves older than M steps back from the present time as shown in equation (7.2). So, the unstable behaviour may be caused by an unobserved control action that may have begun somewhat earlier and may not be included in the *accumulated moves* term (see section 7.3).

However, runs with *accumulated moves* also applied to *extended slopes* without gradient feedback, showed distinct results with robust controller behaviour, since the lack of *gradient feedback* was not reflected in the controller performance. This was not expected since LDMC usually produces an offset error when applied to integrating processes (see on-line results in Figure 7.11). The additional compensation achieved by *gradient feedback* (equation (7.7)), did not seem significant in these tests, possibly because this system was steady at the start.

Move Suppression

The move suppression parameter, A, suppresses manipulated variable movements (see section 4.3). Its effect was tested holding constant set point deviation weight variables, W and changing A values. Reducing move suppression to values as small as 0.01 made little difference on controller performance, while increasing it to 100 led to a slow response and decreased performance.

Output Variable Weightings

As mentioned in section 4.3, these set point deviation-weighting parameters are used to set the relative ranges of variation of the outputs about set points. They have a direct effect on the dynamics of the closed loop system and thus can affect stability.

When holding the move suppression constant, decreasing the values of the weighting factor to 1 showed a slight effect on the output responses. On other side, it was found that increases in the weight factors are not reflected in the controller performance. These results as well as those found above in the move suppression tests, show that this controller is quite robust.

Constraints handling

Notice that the handling of constraints is the conventional LDMC feature and the proposed integrating modification only affects the prediction part of the algorithm. The optimisation part remains the same.

As would be expected, the presence of any active constraint reduces the available degrees of freedom and thus the controller performance. Figure 7.7 illustrates the handling of input constraints when valve CV01 is constrained between 45 - 55 % and CV03 between 30 - 70 %, values above and below the minimum and maximum levels reached by the unconstrained scenario (Figure 7.5 (b)). The effect of the constraints can be seen in the much slower set point tracking where the constraints are active, reducing the controller performance. When severe constraints are added, the control deteriorated and steady-state offsets were introduced.

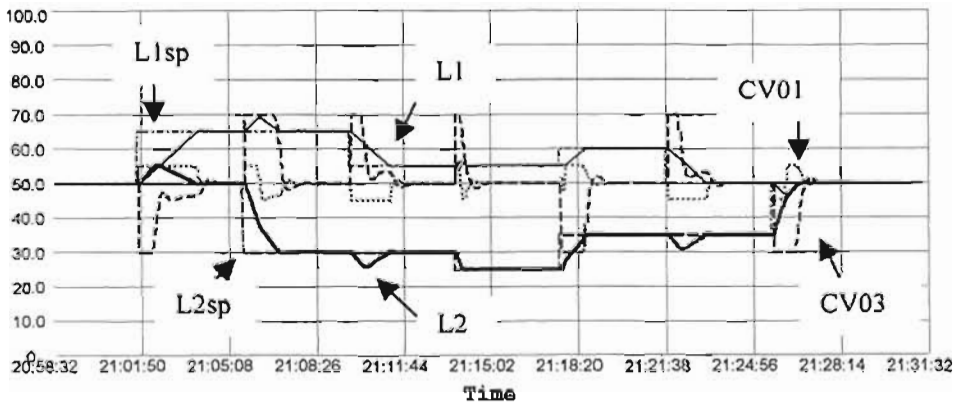


Figure 7.7 Closed loop Integrating ADMC with input constraints: CV01: 45 – 55 % and CV03: 30 – 70 %, N = 2, λ = 1, W = 100

7.5.2 Real-time application

As mentioned above, real-time tests were performed on the Training Plant illustrated in the Figure 2.2. However only a 2-input / 2-output sub-system was simulated (Figure 2.4).

Difficulties were experienced in obtaining accurate step responses due to integration, non-linearity, and the high level of interaction and noise in the system. These factors led to process-model mismatch and consequently a difficult control problem. Several tuning parameters resulted in outputs oscillating around set points even after a long time period.

On-line simulation was carried out with the same parameters used in the off-line simulations for β (0.05), predicted error, Q (1), observed error, R (1), move suppression, Λ (1), set point deviation weight factor, W (100) and also with α set to 0.9999. A sampling interval of 10 seconds and set point start and end of (40,40) were considered.

The same set point sequence change was applied, although on-line runs needed a long time for the controller to achieve the target. The applied results obtained for step changes in set points in LDMC and ADMC runs with integrating compensation are shown in Figures 7.8 and 7.9 respectively.

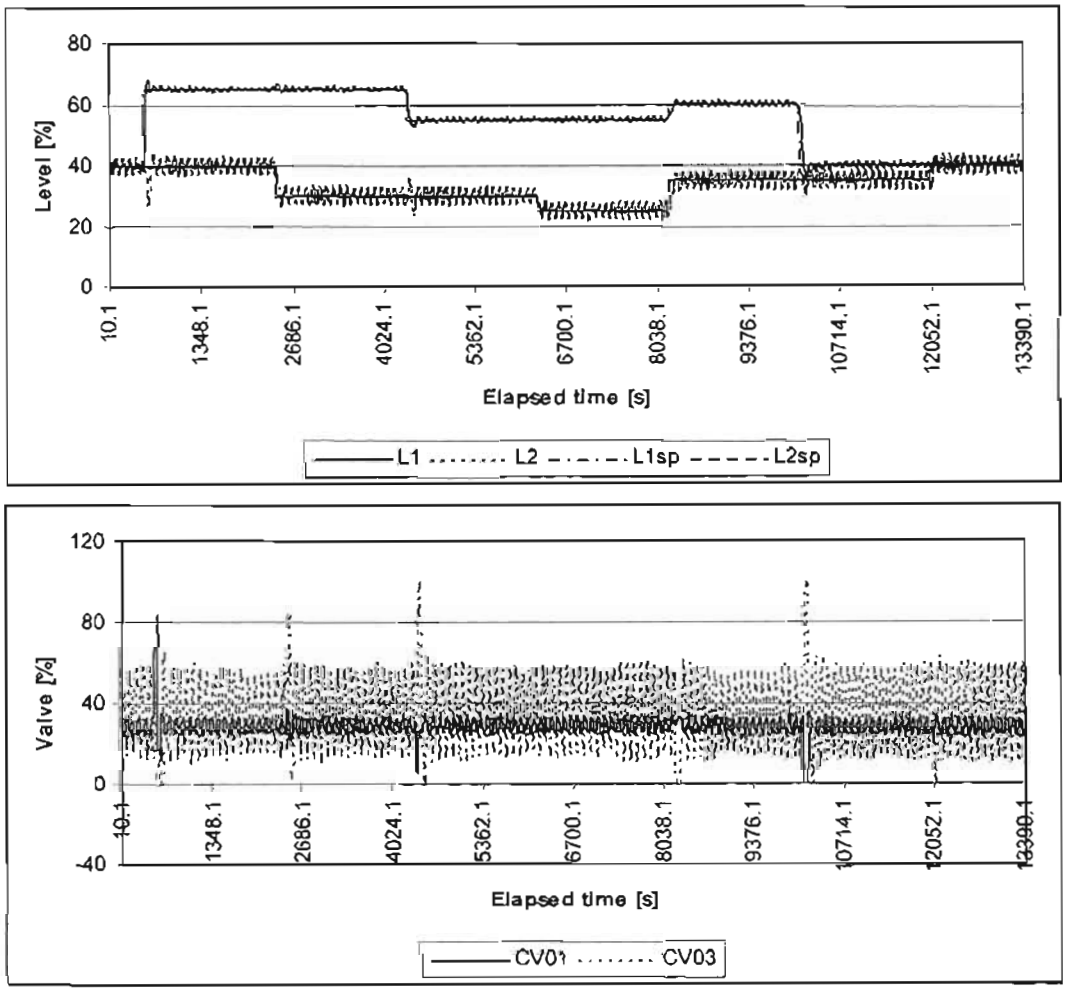


Figure 7.8 Closed loop on-line Integrating LDMC, N = 2, Λ = 1, W = 100

Bad performance, consisting of constant oscillation on the outputs with a bang-bang behaviour of the manipulated variables was showed by the LDMC controller (Figure 7.8). In contrast the ADMC controller showed better than the LDMC algorithm, and good tracking of set points was observed in the levels, although with slight oscillation in the second output (Figure 7.9).

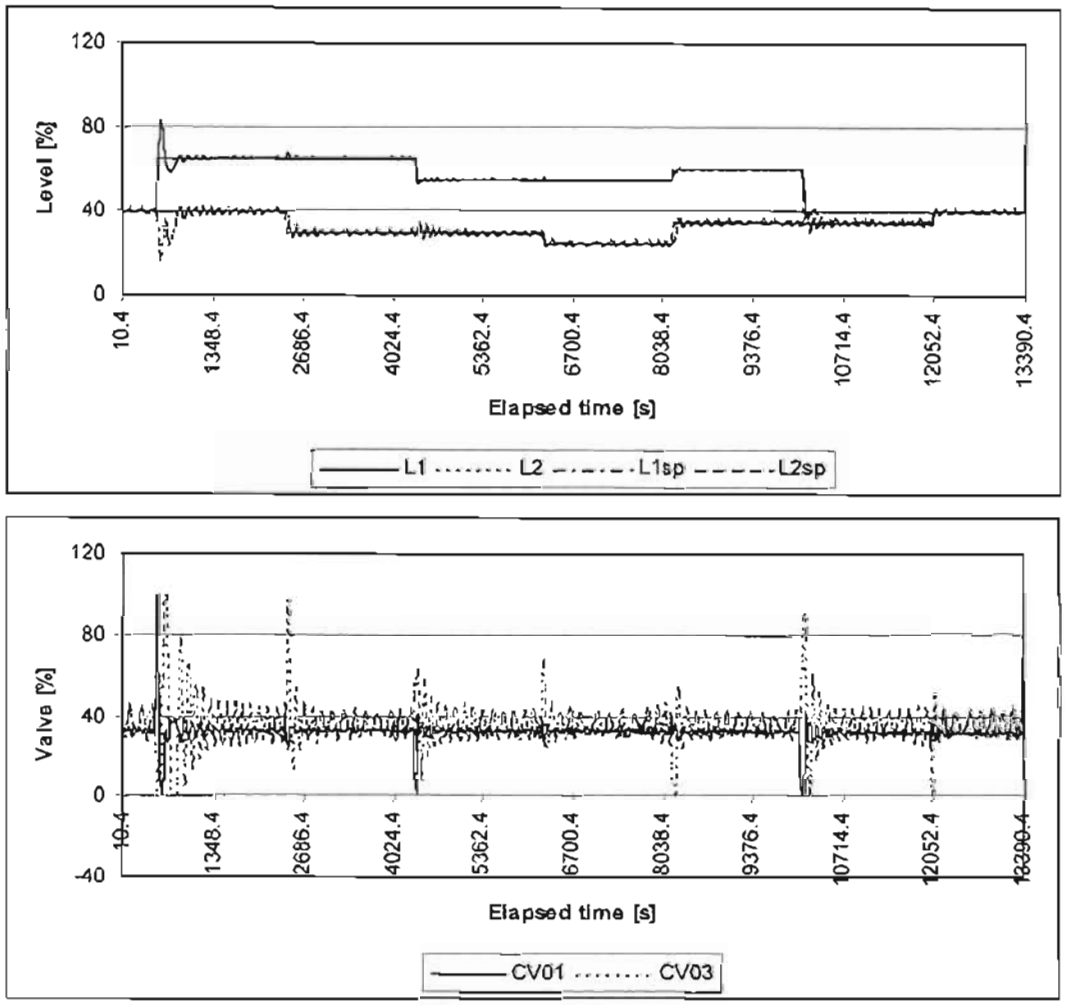


Figure 7.9 Closed loop on-line Integrating ADMC, N = 2, $\Lambda = 1$, $W = 100$

The adaptation plots presented in Figure 7.10 shows the effort of the Adaptive Dynamic Matrix Controller to identify correctly the coefficients of the step responses, although with noise and other disturbances affecting the process as can be seem in the off diagonal elements.

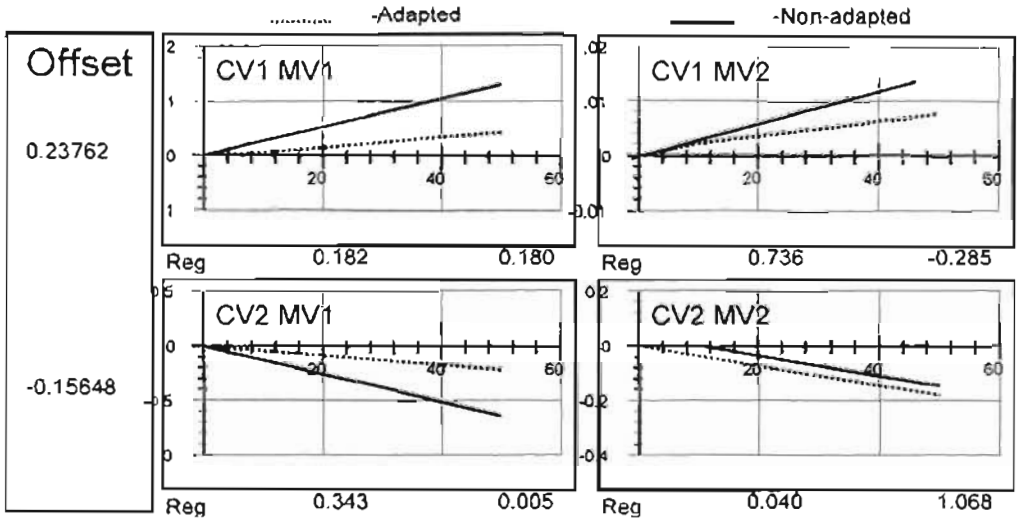


Figure 7.10 Identified responses in the Training Plant 43 minutes from the start of the run

Gradient feedback effect

The algorithms were also tested with all parameters set at the same values as above except for *gradient feedback*, which was not applied. Simulations resulted in deteriorated controller performance with steady-state offset in the LDMC run as illustrated in Figure 7.11 (compare with integrating LDMC in Figure 7.8). That was because, as mentioned in section 7.3, an unaccounted ramp may already be in effect on the outputs in the moves older than M steps back from the present time, which has the potential to cause steady-state offset in the control.

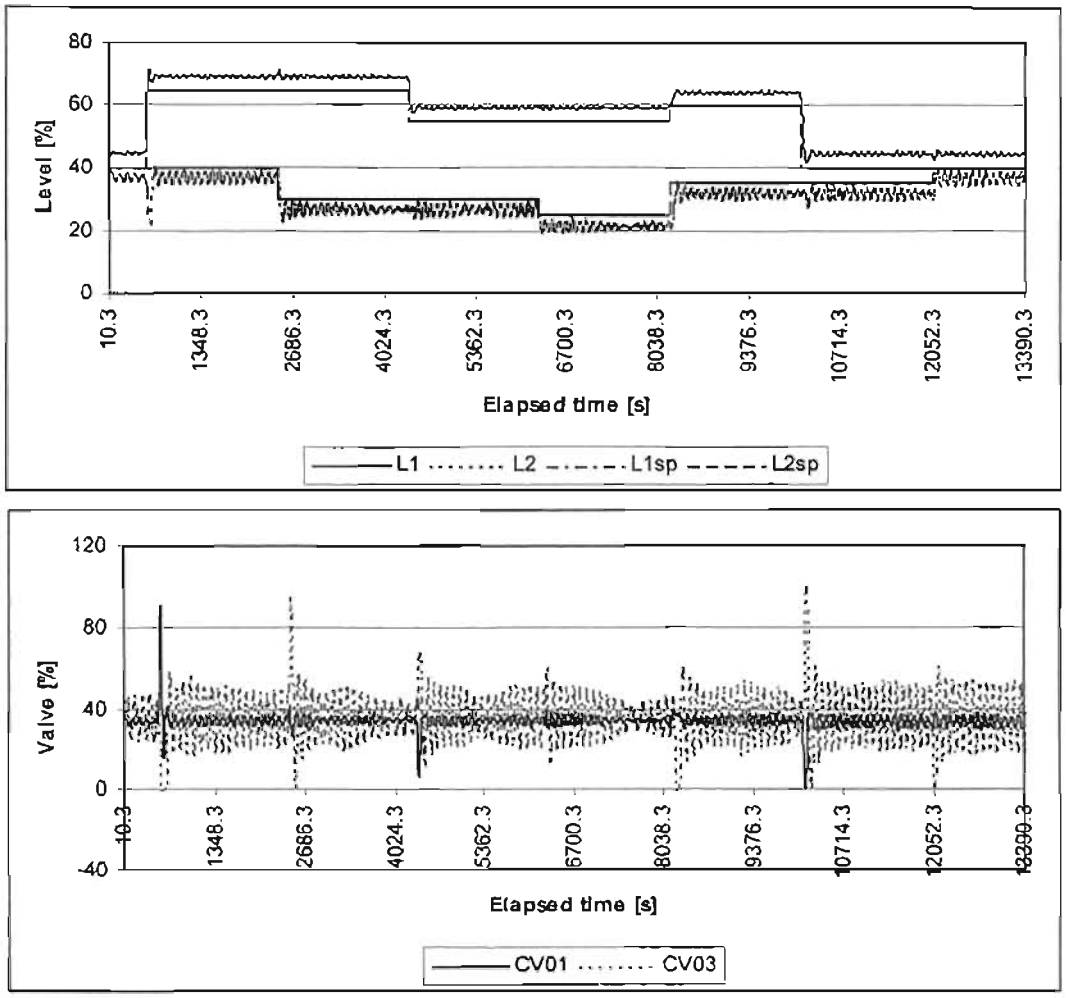


Figure 7.11 Closed loop on-line non-integrating LDMC, N = 2, Λ = 1, W = 100, β= 0

Although with relative improvement compared to Figure 7.11 algorithm with adapted parameters also showed decreased performance, with oscillations and offset in some cases, when *gradient feedback* was not applied as illustrated in Figure 7.12 (compare with the integrating ADMC in Figure 7.9). As also seen in the off-line simulation (section 7.5.1), with adaptation of the model parameters the output error is reduced. However due to the integrating nature of the process, the error is not completely eliminated and a slight offset is observed when integrating compensation is not applied.

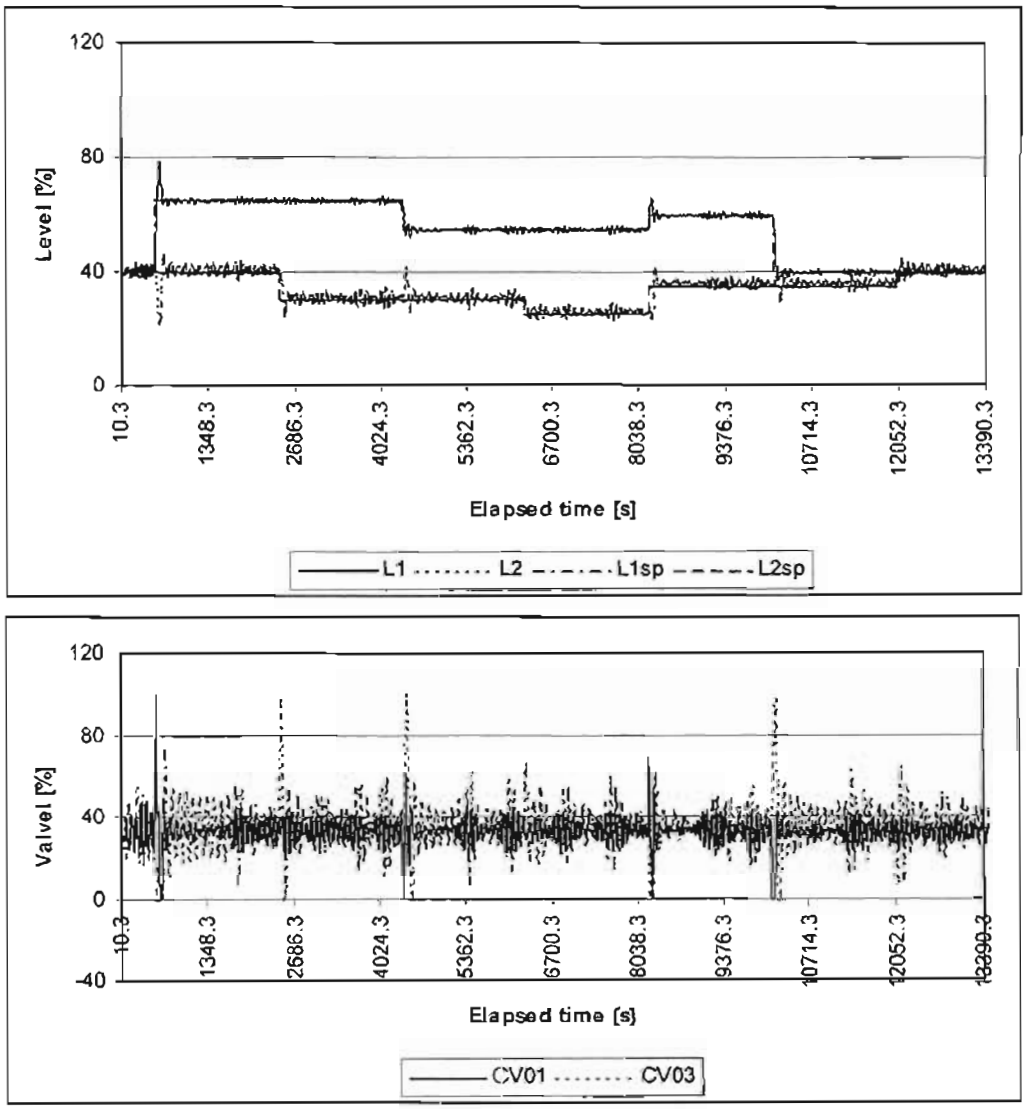


Figure 7.12 Closed loop on-line non-integrating ADMC, N = 2, Λ = 1, W = 100, $\beta = 0$

Thus the *gradient feedback* has proven effective in compensating for unknown ramp responses, perhaps from control moves not included in the summation terms (equation (7.2)). This is contrary to the simulation on-board model in section 7.5.1, where there presumably was little initial gradient error.

Chapter 8

Conclusions and Recommendations

One of the purposes of this study was to recommission the Training Plant, including the addition of digital control to the already existing analogue instrumentation. Other aspects related to software development, interfacing of the system to a computer SCADA system and application of novel algorithms to the equipment.

A major objective was to investigate different aspects of control of the multivariable, non-linear and highly interactive Training Plant. A 2-input / 2-output sub-system of the Training Plant was, defined as a case study.

Two different approaches were adopted for system investigation: a mathematical model derived from first principles, and a step response model based on experimental data.

The mathematical model revealed complex behaviour of the system described by the mixed non-linear differential and algebraic equations represented by a (*47 x 47*) Jacobian matrix. The system was solved through the application of an extended Kalman filter (EKF) technique. The algorithm written in Matlab was used for state estimation, and based on the estimated data, step responses for the 2-input / 2-output sub-system of the Training Plant were predicted. Promising results were obtained. Although the state-space step responses showed similar integrating behaviour to the step responses resulting from the experimental data, further investigation is required in order to find improved plant physical coefficients that give better description of the process since little information about the process is available. A trial and error method is suggested.

It was decided to use Dynamic Matrix Control (DMC), one of the most popular techniques of Model Predictive Control (MPC), to control the Training Plant. DMC is based on the linear *convolution model*, and therefore does not require a rigorous model derived from first principles as above. A fixed dynamic model is used to predict and optimise process performance for a

nominal operating point. However, as the process moves away from this point, control becomes sub-optimal due to process non-linearity. To deal with this problem, the use of a self-tuning adaptive DMC was proposed, since Adaptive Dynamic Matrix Control (ADMC) can be expected to perform well even in the presence of uncertainties, non-linearities and time-varying process parameters.

The conventional self-tuning regulator can be subdivided into two distinct steps: identification and control. The first is mainly concerned with updating an approximate process model so as to guarantee more reliable predictions of the system behaviour. It involves some form of on-line recursive parameter estimation, which also provides estimates of the system states. In the second step, i.e., in the control stage, it is usually necessary to recalculate the coefficients of the controller so that appropriate control action can be derived. However, the determination of new values for such parameters involves the solution of a set of recursive matrix equations. For a multivariable, system this becomes a very time-consuming procedure, and the problem of minimising the error may be ill conditioned if the process variations are not rich in information Therefore, some improvement is highly desirable, particularly when real-time control is to be applied.

One of the most attractive features of the ADMC control scheme developed in this work is that it does not require the solution of such complex matrix equations. Instead, by providing an option for a regularisation approach for a closed-loop configuration, it was possible to reduce the problem dimensions, thereby substantially reducing the computational burden characteristic of the optimal self-tuning regulator when applied to multivariable systems.

The algorithm was tested on the 2-input / 2-ouput Pump-tank system for simulation and laboratory tests. The real-time identification of step responses and adaptation of the DMC on this basis proved quite robust, particularly when the degrees of freedom of the identified responses were limited by regularisation. The off-line tests showed reliable identification over a wide range of RLS identifier tuning. As the changes became implemented in the DMC, expected changes were observed in the quality of control. In the on-line testing of this algorithm on the Pump-tank system, the results were not completely consistent, but nevertheless promising. Quadratic performance indices however showed better controller performance when the process parameters were updated with changes in the operating point. The errors introduced into the step responses supplied to the DMC were correctly identified most of the time, but occasionally a spurious variation would be introduced by the plant data. The best protection against this in practice will be to increase the response time of the identification, (lower Q values in the Kalman filter), and to constrain the allowed range of variation.

Despite Dynamic Matrix Control presenting several novel features and advantages, it is not designed to deal with processes with an integrating nature. These are processes that produce a ramp change in the output for a step change in the input, behaviour presented by the Training Plant currently under study. When a Linear Dynamic Matrix Controller is applied to processes with an integrating nature, a steady state offset is expected. This is caused by a ramp that may be in the effect on the outputs. Thus, an integrating compensation consisting of factors like *gradient feedback*, *accumulated moves* and *extended slopes* was included into the LDMC algorithm. *Gradient feedback* is a "gradient" correction vector for the P predictions up to the horizon. *Accumulated moves* represents a compensation that accumulates all moves older than M steps back from the present time, and *extended slopes* extends the integrating step response so that integrating processes can be treated within the same framework as "self-regulatory" processes.

The resulting Integrating Adaptive Dynamic Matrix Control algorithm was finally applied to the 2-input / 2-output sub-system of the Training Plant. In preparation for the on-line controller software commissioning, and to experiment with the tuning of parameters, preliminary closed loop off-line tests were designed to determine robustness and controller performance using a *convolution model* representation of the plant. Integrating DMC showed good behaviour as long as the process model mismatch was small. For large a mismatch, however, DMC was not satisfactory and model updating was required. The Integrating ADMC could successfully handle this problem, and was able to accurately control the outputs to their set points indicating that the algorithm is quite robust. Unexpected results were obtained when simulating off-line, using the DMC without gradient feedback. Good performance was noted instead of an offset. The absence of gradient feedback correction was not reflected in the controller possibly because the system was steady at the start.

In the real time tests, however, the gradient feedback compensation played a significant role, since as expected, steady state offset was observed when gradient-feedback was not applied in the controller, and integrating compensation was shown to be an effective tool in eliminating this steady state offset. Integrating ADMC showed better performance relative to Integrating DMC, with the outputs following the setpoint although with slight oscillations in the second output.

It is concluded that the tuning of a controller for this highly interacting and non-linear system is a very difficult task. Very slow or oscillating responses were often found when changing parameters. Therefore, many unexplored tuning values need to be tested in order to improve the

controller behaviour. The present work may be regarded as the first of many future studies related to this complex Training Plant, offering a diversity of possible fields of study, and since there are many unresolved research issues related to this system, further investigation is recommended.

REFERENCES

Aitchison, S. and Mulholland, M., "Adaptive Model Predictive Control of the Peak Temperature in a Heat Interchange", *SA Inst. Chem. Eng. 8th Nat. Meeting*, Cape Town, April 16-18, 1997.

Åström K. J. and Wittenmark B., "Adaptive Control" (2nd edition). *Addison-Wesley Publishing Company, Inc*, 1995.

Barnard, J. P. and Aldrich C., "Recent Advances in the Parametric Identification of Non-linear Dynamics Systems", SA Inst. *Chem. Eng. 9th Nat. Meeting*, Secunda, Mpumalanga, October 9-12, 2000.

Becerra, V. M., Roberts, P. D. and Griffiths, G. W., "Applying the Extended Kalman Filter to Systems Described by Non-linear Differential-algebraic Equations", *Control Engineering Practice*, 9, p267-281, 2001

Byrne, G. D. and Ponzi, P. R., "Differential – Algebraic Systems, Their Applications and Solutions", *Computers and Chemical Engineering*, **12** (5), p377-382, 1998.

Chang T. S. and Seborg, D. E., "A Linear Programming Approach for Multivariable Feedback Control With Inequality Constraints", *Int. J. Control*, 37, p583-597, 1983.

Cheng, Y. S., Mongkhonsi, T. and Kershenbaum, L. S., "Sequential Estimator For Non-linear Differential and Algebraic Systems – theoretical development and application", *Computers and Chemical Engineering*, 21 (9), p1051-1067, 1997.

Cutler, C. R. and Ramaker, B. L., "Dynamic Matrix Control – a Computer Control Algorithm", *Proceedings of the Joint Automatic Control Conference*, WP5-b, 1980.

Deghaye, S. W., Guiamba, I. and Mulholland, M., "Enhancements of Dynamic Matrix Control Applied to a Liquid-liquid Extractor", *SA Inst. Chem. Eng. 9th Nat. Meeting*, Secunda, Mpumalanga, October 9-12, 2000.

de Vaal, P. L., "An Overview of Advanced Control Techniques", *Chemical Technology*, Sept/Oct 1999, p13-19.

Dharaskar, K. P. and Gupta, Y. P., "Predictive Control of Non-linear Processes Using Interpolated Models", *Inst. of Chem. Eng., Trans IchemE*, **78**, Part A, p573-580, 2000.

Garcia C., Prett, D. and Morari, M., "Model Predictive Control: Theory and Practice – a Survey", *Automatica*, **25** (3), p335-348, 1989.

Gupta, Y. P., "Control of Integrating Processes Using Dynamic Matrix Control", *Inst. of Chem. Eng., Trans IchemE*, **76**, Part A, p465-470, 1998.

Henson, M. A., "Non-linear Model Predictive Control: current status and future directions", *Computers and Chemical Engineering*, **23**, p187-202, 1998.

Huang, D. and Zhu, X., "Auto-tuning of Dynamic Matrix Control" *Advances in Modelling & Analysis, C, AMSE Press*, 46 (1), p43-46, 1995.

Isermann, R., "Practical Aspects of Process Identification", *Automatica*, **16**, p575-587, 1980.

Johnson, C., "Process Control Instrumentation Technology", *Prentice-Hall International, Inc*, 1982.

Lee, J. H., Morari, M. and Garcia, C. E., "State Space Interpretation of Model Predictive Control", *Automatica*, **30** (4), p707-717, 1994.

Ljung, L., "System Identification. Theory for the User" (2nd edition), *Prentice Hall Inc*, 1999.

Luyben, W. L., "Process Modelling, Simulation and Control for Chemical Engineers" (2nd edition), *McGraw-Hill, Inc*, 1990.

Maiti, S. N., Kapoor N., and Saraf, N. D., "Adaptive Dynamic Matrix Control of pH", *Ind. Eng. Chem. Res.* **33**, p641-646, 1994.

Maiti, S. N. and Saraf, N. D., "Adaptive DMC of Distillation Column With Closed-loop On-line Identification", *J. Proc. Cont.*, **5**, p315-327, 1995a.

Maiti, S. N. and Saraf, N. D., "Start-up and Control of a Distillation Column Using Adaptive Dynamic Matrix Control: an experimental study", *Process Control and Quality*, 7, p143-156, 1995b.

Mulholland M. and Narotam, N. K., "Constrained Predictive Control of a Counter-current Extractor", System Modelling and Optimisation, edited by J. *Dolezal & J. Fidler*, ISBN 0 412 718804, Chapman & Hall, p251-258, 1996.

Mulholland, M., "Personal communication", 2001.

Mulholland, M., Le Lann, M. V., Chouai, A. and Prosser, J., "Visualisation of Constrained Predictive Control of a Liquid-liquid Extractor", in print, *Control Engineering Practice*, 2001.

Mulholland M. and Prosser, J. A., "Constrained Linear Dynamic Matrix Control of a Distillation Column", *SA Inst Ch. Eng. 8th Nat. Meeting*, Cape Town, South Africa, 16-18 April, 1997.

Morshedi A. M., Cutler, C. R. and Skrovanek, T. A., "Optimal Solution of Dynamic Matrix Control with Linear Programming Techniques (LDMC)", *Proc. Am. Control Conf.*, Boston, Massachusetts, 199-208, 1985.

Ogunnaike B. A. and Ray W. H., "Process Dynamics, Modelling and control", *Oxford University Press*, New York, 1994.

Patwardhan, S. C. and Madhavan, K. P., "Non-linear model predictive control using second-order model approximation", *Ind. Eng. Chem. Res.* 32, p334-344, 1993.

Prosser, J. A., "Variations of Linear Dynamic Matrix Control and its Applications", M.Sc. Eng. Dissertation, *University of Natal*, Durban, 1998.

Rice, R. G. and Do, D. D., "Applied Mathematics and Modelling for Chemical Engineers" *John Wiley & Sons, INC*, 1995.

Robertson, M. W., Watters, J. C., Desphande, P. B., Assef, J. Z. and Alatiqi, I. M., "Model Based Control for Reverse Osmosis Desalination Processes", *Desalination*, **104**, p59-68, 1996.

Suganda P., Krishnaswamy P. R. & Rangaiah G. P., "On-line Process Identification from Closed-loop Tests Under PI Control", *Institution of Chemical Engineers Trans IchemE.* 76 (part A). 1998.

Seborg, D. E., Edgar, T. F. & Shah S. L., "Adaptive Control Strategies for Process Control: a survey" *AIChE Journal*, **32**, (6), p881-913, 1986.

Appendix A

General Concepts about Process Models

This appendix presents the theory related to process models. It includes issues such as process variable definition, process characteristics and process models, as well as different forms of process model. Thus, the appendix structure is:

A.1 Variables of a process

The process model represents the relationship between process variables. Those variables can be classified as state, input and output variables.

Input variables are those that independently stimulate the system and can thereby induce change in the internal condition of the process. It is possible to classify these variables as manipulated (or control) variables and disturbance variables. *Manipulated variables* are those input variables, which are at our disposal to manipulate freely as we choose, and *disturbance variables* are those over which we have no control.

Output variables are those by which one obtains information about the internal state of the process. *State variables* are generally recognised as that minimum set of variables essential for completely describing the internal condition of a process, and can be used to predict future states provided future inputs are known. The state variables are, therefore, the true indicator of the internal state of the system. The actual manifestation of these internal states by measurement is what yields an output.

Some process variables (outputs as well as input variables), are directly available for measurement while some are not. Those process variables whose values are made available by direct on-line measurement are classified as measured variables; the others are called unmeasured variables.

Although output variables are defined as measurements, it is possible that some outputs are not measured on-line (no instrument is installed) on the process but require infrequent samples to be taken to the laboratory for analysis. Thus for control system design these are usually considered unmeasured outputs in the sense that the measurements are not available frequently enough for control purposes.

A.2 Process characteristics and process models

Chemical processes can be classified according to nature of the models used to describe their dynamics in several ways:

- Linearity – linear or non-linear

  Thus a process described by linear equations (i.e., equations containing only linear functions) is classified as *linear*, while the *non-linear* process is the one described by non-linear equations. A linear system satisfies the properties of superposition and homogeneity, while a non-linear system does not exhibit any of these properties.

- Number of independent variables – lumped or distributed

  A *lumped* parameter process (which may be linear or non-linear) is one in which the time is the only independent variable. It is described by ordinary differential equations. The process variables of a *distributed* parameter process on the other hand change with spatial as well as with time.

- Stability – stable or unstable

  The process is defined as *stable* if "self-regulatory", that is, the process variables converge to some steady state when disturbed and *unstable* if variables go to infinity (mathematically). Most processes are open-loop stable. However, the exothermic irreversible chemical reactor is a notable example of a process that can be open loop unstable. All real processes can be made closed-loop unstable (with a feedback controller in service), and therefore one of the principal objectives in feedback controller design is to avoid closed-loop instability.

- Order

  If a system is described by one ordinary differential equation with derivatives of order N, as shown in the bellow equation, the system is called the Nth order.

$$a_N \frac{d^N x}{dt^N} + a_{N-1} \frac{d^{N-1} x}{dt^{N-1}} + \cdots + a_1 \frac{dx}{dt} + a_0 x = f(t)$$

  where a_i are constants and f(t) is the forcing function or disturbance.

As we shall see in the next sections, we can classify the present Training Plant system as lumped, non-linear and high order.

A.3 The process model forms

According to [Ogunnaike and Ray, 1994], it is usual to cast the mathematical model for any particular process in one of four ways:

1. The state-space (differential equation) form
2. The transform-domain (Laplace or Z-transforms) form
3. The frequency-response (or complex variable) form
4. The impulse / step-response (or convolution) form.

The last three forms are called input / output models as the mathematical models strictly relate only the input and output variables entirely excluding the state variables. In general, input / output models occur as a result of appropriate transformations of the state-space form, but they can also be obtained directly from input / output data correlation. Because these model types are obviously interrelated, it is possible to convert from one form to another.

Recall that in attempt to control the Training Plant currently under study, a state-space model was developed (see Chapter 3). The system transpired to be multivariable, non-linear, of high order, described by a mixed ordinary differential and algebraic equations (DASs). On other hand, little information about the process parameters was available. Thus, to overcome this complexity, an experimental step response model was then obtained and used for process control (see Chapter 7).

The state-space and impulse / step-response forms are briefly described as follows

A.3.1 The state-space model

When the process model is formulated from first principles, it often naturally occurs in the state-space form in the time domain. The state variables occur explicitly along with the input and output variables. Since the modelling equations are formulated with time as an independent variable, state-space models are most useful for obtaining real-time behaviour of process systems. Discrete-time formulations are especially well suited to computer simulation of process behaviour. These models are also used, almost exclusively, for analysis of non-linear system behaviour, because most of the other model forms can represent only linear dynamic behaviour.

A.3.2 Impulse / step response models

It is typical to adopt the theoretical modelling approach when the underlying mechanisms by which a process operates are reasonably well understood. When the process is too complicated for the theoretical approach (usually because very little information about the fundamental nature of the process is available, or the theoretical model equations are enormously complex) the empirical approach is the appropriate choice.

Impulse / step-response models find their main application in dynamic analysis problems involving arbitrary input functions u(t). In building process models from experimental data sampled at an interval Δt, discrete impulse / step-response models are most useful because the model requires only a data record from well-designed experiments. The particular functional form that the input takes is immaterial. Observe that a very simple experiment of sending an impulse (or a unit step) function as input to the physical system will give this required impulse (or a unit step) response data record. This may then be used for dynamic analysis through the convolution model. A step function is easier to implement on a physical system than an impulse.

One application of impulse / step-response models outside of process dynamics and control is in study of residence time distributions in chemical reactors [Ogunnaike and Ray, 1994]. In the present work, convolution models of two processes with different behaviour, "self-regulatory" and integrating nature were obtained from the experimental data (see Figures 4.5 and 7.2) and then used in Dynamic Matrix Control algorithm for controller design.

Appendix B

Extended Kalman Filter Formulation

Presented in this appendix, is the extended Kalman filter formulation [Mulholand, 2001] for solution of differential and algebraic equation systems. The first step provides a linearisation of the system using a Taylor series expansion, and then the Kalman filter is used for state estimation.

Consider the system of first order differential and algebraic equations

$$\begin{aligned} \frac{dy}{dt} &= f(y,z) \\ 0 &= g(y,z) \end{aligned} \tag{B.1}$$

where y is a vector of state variables and z a vector of algebraic variables.

Defining the Jacobians:

$$A = J_{fy} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \cdots & \frac{\partial f_1}{\partial y_N} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \cdots & \frac{\partial f_2}{\partial y_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_N}{\partial y_1} & \frac{\partial f_N}{\partial y_2} & \cdots & \frac{\partial f_N}{\partial y_N} \end{bmatrix} \quad B = J_{fz} = \begin{bmatrix} \frac{\partial f_1}{\partial z_1} & \frac{\partial f_1}{\partial z_2} & \cdots & \frac{\partial f_1}{\partial z_M} \\ \frac{\partial f_2}{\partial z_1} & \frac{\partial f_2}{\partial z_2} & \cdots & \frac{\partial f_2}{\partial z_M} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_N}{\partial z_1} & \frac{\partial f_N}{\partial z_2} & \cdots & \frac{\partial f_N}{\partial z_M} \end{bmatrix}$$

$$C = J_{gy} = \begin{bmatrix} \frac{\partial g_1}{\partial y_1} & \frac{\partial g_1}{\partial y_2} & \cdots & \frac{\partial g_1}{\partial y_N} \\ \frac{\partial g_2}{\partial y_1} & \frac{\partial g_2}{\partial y_2} & \cdots & \frac{\partial g_2}{\partial y_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_N}{\partial y_1} & \frac{\partial g_N}{\partial y_2} & \cdots & \frac{\partial g_N}{\partial y_N} \end{bmatrix} \quad D = J_{gz} = \begin{bmatrix} \frac{\partial g_1}{\partial z_1} & \frac{\partial g_1}{\partial z_2} & \cdots & \frac{\partial g_1}{\partial z_M} \\ \frac{\partial g_2}{\partial z_1} & \frac{\partial g_2}{\partial z_2} & \cdots & \frac{\partial g_2}{\partial z_M} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_N}{\partial z_1} & \frac{\partial g_N}{\partial z_2} & \cdots & \frac{\partial g_N}{\partial z_M} \end{bmatrix} \tag{B.2}$$

linearise the right hand sides about (y_0, z_0) to obtain the augmented system

$$\begin{aligned}\begin{pmatrix}\dot{y}\\ \dot{0}\end{pmatrix} &= \begin{pmatrix}f(y_0,z_0)\\ g(y_0,z_0)\end{pmatrix} + \left[\begin{array}{c|c}A & B\\ \hline C & D\end{array}\right]\begin{pmatrix}(y-y_0)\\ (z-z_0)\end{pmatrix}\\ &= \begin{pmatrix}F_0\\ G_0\end{pmatrix} + \left[\begin{array}{c|c}A & B\\ \hline C & D\end{array}\right]\begin{pmatrix}y\\ z\end{pmatrix}\end{aligned} \tag{B.3}$$

where

$$\begin{pmatrix}F_0\\ G_0\end{pmatrix} = \begin{pmatrix}f(y_0,z_0) - Ay_0 - Bz_0\\ g(y_0,z_0) - Cy_0 - Dz_0\end{pmatrix} \tag{B.4}$$

To allow for the possibility that some of the z elements might be free, overspecify the behaviour by suggesting that z will move towards some observed value z_0

$$\dot{z} = \tfrac{1}{\tau}(z_0 - z) \tag{B.5}$$

so that equation (B.3) becomes

$$\begin{pmatrix}\dot{y}\\ \dot{z}\end{pmatrix} = \begin{pmatrix}F_0\\ H_0\end{pmatrix} + \left[\begin{array}{c|c}A & B\\ \hline 0 & E\end{array}\right]\begin{pmatrix}y\\ z\end{pmatrix} \tag{B.6}$$

with $H_0 = \tfrac{1}{\tau} z_0$

$E = -\tfrac{1}{\tau} I$

an additional requirement is also defined from equation (B.3) as

$$\left[C \mid D\right]\begin{pmatrix}y\\ z\end{pmatrix} = -G_0 \tag{B.7}$$

To handle the possibility that the states y may also be observed, augment the above equation as follows

$$\left[\begin{array}{c|c} I & 0 \\ \hline C & D \end{array}\right]\left(\begin{array}{c} y \\ \hline z \end{array}\right) = \left(\begin{array}{c} y_0 \\ \hline -G_0 \end{array}\right) \tag{B.8}$$

Letting $x = \left(\begin{array}{c} y \\ \hline z \end{array}\right)$, $\Phi = \left[\begin{array}{c|c} A & B \\ \hline 0 & E \end{array}\right]$ and $u_0 = \left(\begin{array}{c} F_0 \\ \hline H_0 \end{array}\right)$, integrate from 0 to t keeping u_0 fixed:

$$x_t = e^{\Phi t} x_0 + \left[e^{\Phi t} - I\right]\Phi^{-1} u_0 \tag{B.9}$$

The same integration can be applied to the interval $t \rightarrow t + \Delta t$ giving

$$x_{t+\Delta t} = A_t x_t + B_t u_t \tag{B.10}$$

with $A_t = e^{\Phi \Delta t}$

$$B_t = \left[e^{\Phi \Delta t} - I\right]\Phi^{-1}$$

and setting $C_t = \left[\begin{array}{c|c} L & 0 \\ \hline C & D \end{array}\right]$, $w_t = \left(\begin{array}{c} y_0 \\ \hline -G_t \end{array}\right)$, with L selecting observed states, it is also required

$$C_t x_t = w_t \tag{B.11}$$

Representing the equivalent set of measurements by $\hat{w}_t$, the Kalman filter is configured as follows

$$\begin{aligned} K_t &= M_t C_t^T \left[C_t M_t C_t^T + R\right]^{-1} \\ x_{t+\Delta t} &= A_t x_t + B_t u_t + K_t\left[\hat{w}_t - C_t x_t\right] \\ M_{t+\Delta t} &= A_t \left[I - K_t C_t\right] M_t A_t^T + Q \end{aligned} \tag{B.12}$$

where the covariance matrix is initialised with M_0 small and Q and R the expected error covariance matrices for the model and the measurements respectively.

Appendix C

Matlab Extended Kalman Filter Algorithm

The propose of this appendix is to provide the Matlab algorithm applied in Chapter 3, for state estimation of the theoretical process model of the Training Plant, using a extended Kalman filter. The main program was developed by Mulholland [2001]. All contributions are gratefully acknowledged.

```
% Differential Algebraic Equations - Extended Kalman Filter Solver (DAE-EKF)
% FOR AECI Training Plant Simulator

% M. Mulholland 2001.05.10  vers 11 : units to m3/h, m, kW
% B. Loveday 2001.04.02
% J-R. Vosloo 2001.04.02
% N. du Preez 2001.04.02
% I. Guiamba 2001.05.02

% The model is multivariable and nonlinear with a mixture of differential and
% algebraic equations, with state variable outputs, inputs, associated variables and
physical %parameters

% INPUT:  Initial approximation of the states, inputs, associated variables and physical
parameters
% Any of the above as measurements in real time

% OUTPUT: Fitted estimates in real time for states, inputs, associated variables and
physical parameters

% NOTATION:
% ny : total number of variables (including states, inputs, associated variables and
physical parameters)
% ns : number of states (first ns of the ny)
% nf : number of equations (first ns are the appropriate DE's for the ns states)

% Matrix dimensions

close all;
clear all;

% ----------------TO BE SET BY USER BELOW----------------

% Time interval
t = 0;
t_final = 1.5;  % h
dt = 0.002;     % h

% ----------------TO BE SET BY USER ABOVE----------------

% Maximum possible sizes
nfmax =100;  % No of equations
nymax =200;  % No of variables
f = zeros(nfmax,1);
ff = zeros(nfmax,3);
y=zeros(nymax,1);
y_lastJ=zeros(nymax,1);
yy=zeros(nymax,6);
yo = zeros(nymax,1);
ylimflag = zeros(nymax,3);      % flags (1:lower; 2:upper to indicate limiting; 3:limit
value)
for i=1:nymax
  ylimflag(i,1)=i;      % marker
end

pcmove_tol=5;  % max percentage move for any one variable before reevaluation of
Jacobians
minrange=0.001;  % minimum allowed ranging for automatic Q & R setting
tol = 1e-20;    % Tolerance for matrix exponential convergence
SM = 1e-10;     % Small value to protect against div-by-zero, etc
SMM = 1e-10;    % Small value to weed matrix Mt and matrix K
Kint = 5;       % No. of steps between re-evaluation of Kalman K
Kcomp = 10;     % No. of compulsory initial re-evaluations of Kalman K
fast_response_factor=0.5;
Tau = fast_response_factor*dt;

% PLOTTING INFORMATION
dtplot = 1*dt;
p=floor(t_final/dtplot);
tp=zeros(p,1);
tlastplot=0;
iplot=0;

% Main time loop
INIT = 1;  % Initialise on First Pass
nKint=0;
nKcomp=0;

while t<t_final

  t=t+dt

% ----------------TO BE SET BY USER BELOW----------------

% Set present observations on each step

  if INIT
      % plotting arrays
    h11P=zeros(p,1); h12P=zeros(p,1); h13P=zeros(p,1); h15P=zeros(p,1);
T11P=zeros(p,1); T12P=zeros(p,1);

    T13P=zeros(p,1); T14P=zeros(p,1); T15P=zeros(p,1); F01P=zeros(p,1);
F02P=zeros(p,1); F03P=zeros(p,1);

    F04P=zeros(p,1); F05P=zeros(p,1); F10P=zeros(p,1); F11P=zeros(p,1);
F12P=zeros(p,1); F13P=zeros(p,1);

    F14P=zeros(p,1); F15P=zeros(p,1); dPK1P=zeros(p,1); dPK2P=zeros(p,1);
dPK4P=zeros(p,1); dPCV01P=zeros(p,1);

    dPCV02P=zeros(p,1); dPCV03P=zeros(p,1); dPCV04P=zeros(p,1);
dPCV05P=zeros(p,1); dPCV10P=zeros(p,1);

    dPCV11P=zeros(p,1); dPCV13P=zeros(p,1); P03P=zeros(p,1); P04P=zeros(p,1);
P05P=zeros(p,1); P10P=zeros(p,1);

    P12P=zeros(p,1); P19P=zeros(p,1); P30P=zeros(p,1); T02P=zeros(p,1);
T04P=zeros(p,1); T19P=zeros(p,1);

    T20P=zeros(p,1); T30P=zeros(p,1); T31P=zeros(p,1); qH1P=zeros(p,1);
qH2P=zeros(p,1); qH3H4P=zeros(p,1);

      X01P=zeros(p,1); X02P=zeros(p,1); X03P=zeros(p,1); X04P=zeros(p,1);
X05P=zeros(p,1); X10P=zeros(p,1);
      X11P=zeros(p,1); X13P=zeros(p,1); wld_obs_err=zeros(p,1);
wld_deriv_err=zeros(p,1);
  end

    % Constants
  hSMALL = 0.05;                  % small level to protect derivatives for heat
balances in tanks
      LORheel= 0.01;              % heel of water left in tank
      LORflow= 0.0001;            % low flow for heat exchangers
      LORSMALL = 0.00001; % loss of static and pump heads for empty lines
      Tamb = 20;                  % ambient temperature (C)
      T31geyzer = 80;         % water temperature supplied by geyzer
      rog  = 1;          % densitP x gravitational constant  % All pressures in m of
water
      rocp = 4186/3600;       % densitP x specific heat       % Power in [kW/
(DC*(m3 water / h))
      hP03 = 8;       % Junctions height above Floor  [m]
      hP04 = 1.2;
      hP05 = 8;
      hP10 = 1;
      hP12 = 1;
      hP19 = 1;
      hP30 = 1;
      h01  = 1.5;
      h02  = 2.0;
      h03  = 8;
      h03BOT = h03- 1.5;   % tank with a datum above ground
      h04  = 1;
      h05  = 6;
      h05BOT = h05 - 4;                % tank with a datum above ground

      % tuning constants
      A1 =  0.1;                                % Tanks area  [m2]
      A2 =  0.1;
      A3 =  0.4;
      A4 =  0.4;
      A5 =  0.1;
      aK1 = 0;                                  % Pumps coefficient
      bK1 = -20/30;
      cK1 = 8;
      aK2 = 0;
      bK2 = -30/30;
      cK2 = 15;
      aK4 = 0;
      bK4 = -20/30;
      cK4 = 8;
      alpha01 = 30/30;                  % valves constants
      alpha02 = 20/30;
      alpha03 = 20/30;
      alpha04 = 10/30;
      alpha05 = 20/30;
      alpha10 = 10/30;
      alpha11 = 20/30;
      alpha13 = 20/30;
      kL01 = 400/(30*30);               % pipes constants
      kL02 = 500/(30*30);
      kL03 = 600/(30*30);
      kL04 = 500/(30*30);
      kL05 = 700/(30*30);
      kL10 = 50/(30*30);
      kL11 = 500/(30*30);
      kL12 = 50/(30*30);
      kL13 = 500/(30*30);
      kL14 = 100/(30*30);
      kL15 = 400/(30*30);       % lower this if the level builds up in the column
excessively
      kL19 = 300/(30*30);       % could drive out column using h15-h15BOT.
instead of h15-h02
      kL20 = 50/(30*30);
      kL31 = 4000/(30*30);
      kL40 = 1/(30*30);
      UAH1 = 1;                         % (overall heat transfer coefficient*heat
transfer area)
      UAH2 = 1;
      h14  = 1;                         % Tank 4 level (constant)

  % Set initial values
  h11o=1;
  h12o=1;
  h13o=7;
  h15o=2.1;
  T11o=25;
  T12o=25;
  T13o=25;
  T14o=80;
  T15o=25;
  F01o=3;
  F02o=3;
  F03o=3;
  F04o=1.5;
  F05o=1.5;
  F10o=1.5;
  F11o=3;
  F12o=3;
  F13o=3;
  F14o=1.5;
  F15o=0.3;
  dPK1o=7;
  dPK2o=12;
  dPK4o=7;
  dPCV01o=8;
  dPCV02o=6;
  dPCV03o=6;
  dPCV04o=6;
  dPCV05o=6;
  dPCV10o=6;
  dPCV11o=6;
  dPCV13o=6;
  P04o=3;
  P10o=5;
  P12o=8;
  P19o=5;
  P30o=4;
  T02o=50;
  T04o=70;
  T19o=60;
```

```
T20o=71;
T30o=64;
T31o=80;
qH1o=40;
qH2o=40;
qH3H4o=60;
X01o=0.4; if t>0.1 X01o=0.55; end; if t>1.0 X01o=0.5 ; end
X02o=0;
X03o=0.4;  if t>0.4 X03o=0.7; end; if t>0.7 X03o=0.55; end
X04o=0;
X05o=0;
X10o=0;
X11o=0.5;
X13o=0.5
F31o=3;

% Check percent of ranges moved since last step to tell if must re-evaluate Jacobians
if INIT
  REEVALUATE=1;
else
  REEVALUATE=0;
  for j=1:ny
    pcmove=100*abs((yy(j)-y_lastJ(j))/(yy(j,6)-yy(j,5)));
    if pcmove>pcmove_tol
      y_lastJ=y;
      REEVALUATE=1;  % if any one move greater than tolerance
      break;
    end
  end
end


% ----------------------TO BE SET BY USER ABOVE-----------------------------

      pertfr = 0.001;

n_evals = 1+REEVALUATE*nymax;   % breaks out of loop at 1+nys

      % EVALUATE FUNCTIONS & their JACOBIAN
      for ne=1:n_evals
      if (ne>1)
      % perturb
      jj=ne-1;
      dyjj = pertfr*(yy(jj,6)-yy(jj,5));
      y(jj)=y(jj)+ dyjj;
      end

      % ----------------------TO BE SET BY USER BELOW----

n=0;
   % Selection of Variables & Observations, Setting of Observation Errors and Ranges
   % (Observation Errors are as Standard Deviation (-ve=absolute; +ve=% of Initial))

   % STATES                 r s o  erx  min   rmax       init
            n=n+1; if INIT yy(n,:)=[ 1 1 0 1 0.0 2*h11o ]; y(n)=h11o; end; h11
=y(n); yo(n)=h11o;  % [m]
            n=n+1; if INIT yy(n,:)=[ 2 1 0 1 0.0 2*h12o ]; y(n)=h12o; end; h12
=y(n); yo(n)=h12o;  % [m]
            n=n+1; if INIT yy(n,:)=[ 3 1 0 1 h03BOT h03BOT+2 ]; y(n)=h13o; end;
h13  =y(n); yo(n)=h13o;  % [m]
            n=n+1; if INIT yy(n,:)=[ 4 1 0 1 h05BOT h05BOT+3 ]; y(n)=h15o; end;
h15  =y(n); yo(n)=h15o;  % [m]

            n=n+1; if INIT yy(n,:)=[ 5 1 0 1 0 15 85 ]; y(n)=T11o; end; T11  =y(n);
yo(n)=T11o;  % [C]
            n=n+1; if INIT yy(n,:)=[ 6 1 0 1 0 15 85 ]; y(n)=T12o; end; T12  =y(n);
yo(n)=T12o;  % [C]
            n=n+1; if INIT yy(n,:)=[ 7 1 0 1 0 15 85 ]; y(n)=T13o; end; T13  =y(n);
yo(n)=T13o;  % [C]
            n=n+1; if INIT yy(n,:)=[ 8 1 0 1 0 15 85 ]; y(n)=T14o; end; T14  =y(n);
yo(n)=T14o;  % [C]
            n=n+1; if INIT yy(n,:)=[ 9 1 0 1 0 15 85 ]; y(n)=T15o; end; T15  =y(n);
yo(n)=T15o;  % [C]
      ns=n;  % No of states


% OTHER VARIABLES

            n=n+1; if INIT yy(n,:)=[10 1 0 1 0.0 2*F01o ]; y(n)=F01o; end; F01
=y(n); yo(n)=F01o;  % [m3/h]
            n=n+1; if INIT yy(n,:)=[11 1 0 1 0.0 2*F02o ]; y(n)=F02o; end; F02
=y(n); yo(n)=F02o;  % [m3/h]
            n=n+1; if INIT yy(n,:)=[12 1 0 1 0.0 2*F03o ]; y(n)=F03o; end; F03
=y(n); yo(n)=F03o;  % [m3/h]
            n=n+1; if INIT yy(n,:)=[13 1 0 1 0.0 2*F04o ]; y(n)=F04o; end; F04
=y(n); yo(n)=F04o;  % [m3/h]
            n=n+1; if INIT yy(n,:)=[14 1 0 1 0.0 2*F05o ]; y(n)=F05o; end; F05
=y(n); yo(n)=F05o;  % [m3/h]
            n=n+1; if INIT yy(n,:)=[15 1 0 1 0.0 2*F10o ]; y(n)=F10o; end; F10
=y(n); yo(n)=F10o;  % [m3/h]
            n=n+1; if INIT yy(n,:)=[16 1 0 1 0.0 2*F11o ]; y(n)=F11o; end; F11
=y(n); yo(n)=F11o;  % [m3/h]
            n=n+1; if INIT yy(n,:)=[17 1 0 1 0.0 2*F12o ]; y(n)=F12o; end; F12
=y(n); yo(n)=F12o;  % [m3/h]
            n=n+1; if INIT yy(n,:)=[18 1 0 1 0.0 2*F13o ]; y(n)=F13o; end; F13
=y(n); yo(n)=F13o;  % [m3/h]
            n=n+1; if INIT yy(n,:)=[19 1 0 1 0.0 2*F14o ]; y(n)=F14o; end; F14
=y(n); yo(n)=F14o;  % [m3/h]
            n=n+1; if INIT yy(n,:)=[20 1 0 1 0.0 2*F15o ]; y(n)=F15o; end; F15
=y(n); yo(n)=F15o;  % [m3/h]

      n=n+1; if INIT yy(n,:)=[21 1 0 1 0.0 2*dPK1o ]; y(n)=dPK1o; end; dPK1  =y(n);
yo(n)=dPK1o;  % [m]
      n=n+1; if INIT yy(n,:)=[22 1 0 1 0.0 2*dPK2o ]; y(n)=dPK2o; end; dPK2  =y(n);
yo(n)=dPK2o;  % [m]
      n=n+1; if INIT yy(n,:)=[23 1 0 1 0.0 2*dPK4o ]; y(n)=dPK4o; end; dPK4  =y(n);
yo(n)=dPK4o;  % [m]

      n=n+1; if INIT yy(n,:)=[24 1 0 1 0.0 2*dPCV01o ]; y(n)=dPCV01o; end;
dPCV01 =y(n); yo(n)=dPCV01o;  % [m]
      n=n+1; if INIT yy(n,:)=[25 1 0 1 0.0 2*dPCV02o ]; y(n)=dPCV02o; end;
dPCV02 =y(n); yo(n)=dPCV02o;  % [m]
      n=n+1; if INIT yy(n,:)=[26 1 0 1 0.0 2*dPCV03o ]; y(n)=dPCV03o; end;
dPCV03 =y(n); yo(n)=dPCV03o;  % [m]
      n=n+1; if INIT yy(n,:)=[27 1 0 1 0.0 2*dPCV04o ]; y(n)=dPCV04o; end;
dPCV04 =y(n); yo(n)=dPCV04o;  % [m]
      n=n+1; if INIT yy(n,:)=[28 1 0 1 0.0 2*dPCV05o ]; y(n)=dPCV05o; end;
dPCV05 =y(n); yo(n)=dPCV05o;  % [m]
      n=n+1; if INIT yy(n,:)=[29 1 0 1 0.0 2*dPCV10o ]; y(n)=dPCV10o; end;
dPCV10 =y(n); yo(n)=dPCV10o;  % [m]
      n=n+1; if INIT yy(n,:)=[30 1 0 1 0.0 2*dPCV11o ]; y(n)=dPCV11o; end;
dPCV11 =y(n); yo(n)=dPCV11o;  % [m]
      n=n+1; if INIT yy(n,:)=[31 1 0 1 0.0 2*dPCV13o ]; y(n)=dPCV13o; end;
dPCV13 =y(n); yo(n)=dPCV13o;  % [m]

      n=n+1; if INIT yy(n,:)=[33 1 0 1 0.0 2*P04o ]; y(n)=P04o; end; P04 =y(n);
yo(n)=P04o;  % [m]
      n=n+1; if INIT yy(n,:)=[35 1 0 1 0.0 2*P10o ]; y(n)=P10o; end; P10 =y(n);
yo(n)=P10o;  % [m]
      n=n+1; if INIT yy(n,:)=[36 1 0 1 0.0 2*P12o ]; y(n)=P12o; end; P12 =y(n);
yo(n)=P12o;  % [m]
      n=n+1; if INIT yy(n,:)=[37 1 0 1 0.0 2*P19o ]; y(n)=P19o; end; P19 =y(n);
yo(n)=P19o;  % [m]
      n=n+1; if INIT yy(n,:)=[38 1 0 1 0.0 2*P30o ]; y(n)=P30o; end; P30 =y(n);
yo(n)=P30o;  % [m]

            n=n+1; if INIT yy(n,:)=[39 1 0 0.3 15 85 ]; y(n)=T02o; end; T02 =y(n);
yo(n)=T02o;  % [C]
            n=n+1; if INIT yy(n,:)=[40 1 0 1 15 85 ]; y(n)=T04o; end; T04 =y(n);
yo(n)=T04o;  % [C]
            n=n+1; if INIT yy(n,:)=[41 1 0 1 15 85 ]; y(n)=T19o; end; T19 =y(n);
yo(n)=T19o;  % [C]
            n=n+1; if INIT yy(n,:)=[42 1 0 1 15 85 ]; y(n)=T20o; end; T20 =y(n);
yo(n)=T20o;  % [C]
            n=n+1; if INIT yy(n,:)=[43 1 0 1 15 85 ]; y(n)=T30o; end; T30 =y(n);
yo(n)=T30o;  % [C]
            n=n+1; if INIT yy(n,:)=[44 1 0 1 15 85 ]; y(n)=T31o; end; T31 =y(n);
yo(n)=T31o;  % [C]

      n=n+1; if INIT yy(n,:)=[45 1 0 1 -2*qH1o 2*qH1o ]; y(n)=qH1o;  end; qH1
=y(n); yo(n)=qH1o;   % [kW]
      n=n+1; if INIT yy(n,:)=[46 1 0 1 0.0   2*qH2o  ]; y(n)=qH2o;  end; qH2
=y(n); yo(n)=qH2o;   % [kW]
      n=n+1; if INIT yy(n,:)=[47 1 0 1 0.0   2*qH3H4o ]; y(n)=qH3H4o; end; qH3H4
=y(n); yo(n)=qH3H4o;  % [kW]

   % Valves
   n=n+1; if INIT yy(n,:)=[48 1 1 1 0.0 1.0 ]; y(n)=X01o; end; X01 =y(n); yo(n)=X01o;
% [fraction open]
   n=n+1; if INIT yy(n,:)=[49 1 1 1 0.0 1.0 ]; y(n)=X02o; end; X02 =y(n); yo(n)=X02o;
% [fraction open]
   n=n+1; if INIT yy(n,:)=[50 1 1 1 0.0 1.0 ]; y(n)=X03o; end; X03 =y(n); yo(n)=X03o;
% [fraction open]
   n=n+1; if INIT yy(n,:)=[51 1 1 1 0.0 1.0 ]; y(n)=X04o; end; X04 =y(n); yo(n)=X04o;
% [fraction open]
   n=n+1; if INIT yy(n,:)=[52 1 0 1 0.0 1.0 ]; y(n)=X05o; end; X05 =y(n); yo(n)=X05o;
% [fraction open]
   n=n+1; if INIT yy(n,:)=[53 1 1 1 0.0 1.0 ]; y(n)=X10o; end; X10 =y(n); yo(n)=X10o;
% [fraction open]
   n=n+1; if INIT yy(n,:)=[54 1 1 1 0.0 1.0 ]; y(n)=X11o; end; X11 =y(n); yo(n)=X11o;
% [fraction open]
   n=n+1; if INIT yy(n,:)=[55 1 1 1 0.0 1.0 ]; y(n)=X13o; end; X13 =y(n); yo(n)=X13o;
% [fraction open]

   % New variables
            n=n+1; if INIT yy(n,:)=[56 1 0 1 0.0 2*F31o ]; y(n)=F31o; end; F31
=y(n); yo(n)=F31o;  % [m3/h]


   % STORE COUNT
   ny=n;  % No of variables

   % Common Error Factors for Tuning
   yo_err_factor = 0.1;  % multiplies by relative "yo" errors above to get observation
errors for observed variables only
   yu_err_factor =  1;  % multiplies by relative "y" errors above to set unknown 'z'
model errors
   f_err_factor  = 0.1;  % multiplies by relative "f" errors below to get equation errors

   % line overrides "LOR" = 0 for empty tanks, else 1
            if h11<=LORheel                          %h11
            LOR1=LORSMALL;
            else
            LOR1=1;
            end
            if h12<=LORheel                          %h12
            LOR2=LORSMALL;
            else
            LOR2=1;
            end
            if h13<=(h03BOT+LORheel)     %h13
            LOR3=LORSMALL;
            else
            LOR3=1;
            end
            if h15<=(h05BOT+LORheel)     %h15
            LOR5=LORSMALL;
            else
            LOR5=1;
      end
      if F02<=LORflow                            % F02
            LORH1=LORSMALL;
            else
            LORH1=1;
            end
      if F14<=LORflow                            % F14
            LORH2=LORSMALL;
            else
            LORH2=1;
            end

n=0;
            % Selection of Functions, Setting of Functions
            %          :sel err rel

   % VOLUME BALANCES
n=n+1; f(n,:)=[ 1  1  1];  f(n) = (- F11 + F01)/A1;
            % h11  Tank 1
n=n+1; f(n,:)=[ 1  1  2];  f(n) = (- F12 + F02 + F11 + F13 + F15)/A2;
   % h12  Tank 2
n=n+1; f(n,:)=[ 1  1  3];  f(n) = (- F13 + F03)/A3;
   % h13  Tank 3
```

```
    n=n+1; ff(n,:)=[ 1  1  4];  f(n) = (- F15 + F05)/A5
        % h15  column

        % HEAT BALANCES
    n=n+1; ff(n,:)=[ 1  1  5];  f(n) = (-F11*T11 + F01*T12)/(A1*(h11+hSMALL)) - ...
       (T11/(h11+hSMALL))*(- F11 + F01)/A1;
                % T11  Tank 1
    n=n+1; ff(n,:)=[ 1  1  6];  f(n) = (-F12*T12 + F11*T11 + F13*T13 + ...
       F15*T15 + F02*T02)/(A2*(h12+hSMALL)) - ...
       (T12/(h12+hSMALL))*(- F12 + F02 + F11 + F13 + F15)/A2;
                % T12  Tank 2
    n=n+1; ff(n,:)=[ 1  1  7];  f(n) = (-F13*T13 + F03*T12)/(A3*(h13-h03BOT+hSMALL))
...
       (T13/(h13-h03BOT+hSMALL))*(- F13 + F03)/A3;
                % T13  Tank 3

    n=n+1; ff(n,:)=[ 1  1  8];  f(n) = F14*(T04-T14)/(A4*(h14+hSMALL));
% T14  Tank 4
    n=n+1; ff(n,:)=[ 1  1  9];  f(n) = (-F15*T15 + F05*T12)/(A5*(h15-h05BOT+hSMALL))
...
       (T15/(h15-h05BOT+hSMALL))*(- F15 + F05)/A5;
% T15  Column

    n=n+1; ff(n,:)=[ 1 10*F14o  8];  f(n) = (F14-F10)*(T14 - T19) - F02*(T02-T12);
        % overal balance on H1
    n=n+1; ff(n,:)=[ 1 1*F14o  8];  f(n) = (F14-F10)*T19 + F10*T14 - F14*T20;
        % heat balance at junction hP19

                        dT1 = T14-T02;
                        dT2 = T19-T12;
       if (dT1*dT2 <= 0)
         q = 0;
       else
         if dT1~=dT2
           q = LORH1*UAH1*(dT1-dT2)/log(dT1/dT2);
       % will allow reverse heat flow too
         else
           q = LORH1*UAH1* dT1;
         end
       end
    n=n+1; ff(n,:)=[ 1 10 45];  f(n) = -qH1 + q;
    n=n+1; ff(n,:)=[ 1 10 45];  f(n) = -qH1 + (T02 - T12)*rocp*F02;
     % heat to F02 (kW)
    n=n+1; ff(n,:)=[ 1 1  47];  f(n) = -qH3H4 + (T31 - T30)*rocp*F31;
     % gas heaters (kW)
                dT1 = T20-Tamb;
                        dT2 = T30-Tamb;
       if (dT1*dT2 <= 0)
         q = 0;
       else
         if dT1~=dT2
           q = LORH2*UAH2*(dT1-dT2)/log(dT1/dT2);
     % will allow reverse heat flow too
         else
           q = LORH2*UAH2* dT1;
         end
       end
    n=n+1; ff(n,:)=[ 1 1 46];  f(n) = -qH2 + q;
    n=n+1; ff(n,:)=[ 1 1 46];  f(n) = -qH2 + (T20-T30)*rocp*F14;
       % Cooling coil
    n=n+1; ff(n,:)=[ 1 1 44];  f(n) = -T31+geyzer + T31;
       % geyzer temperature setting
    n=n+1; ff(n,:)=[ 1 1*F14o 40];  f(n) = -F14*T04 + F04*T30 + F31*T31;
        % heat balance at junction hP04


        % MASS BALANCE AT SPLIT hP12
    n=n+1; ff(n,:)=[ 1 1 17];  f(n) = -F12 + F01 + F02 + F03 + F05;
% tank 2 output flow


 % MASS BALANCE at geyzer bypass hP30
    n=n+1; ff(n,:)=[ 1 1 50];  f(n) = -F31 + F14 - F04;

                % PRESSURE DROP OVER THE PUMPS
    n=n+1; ff(n,:)=[ 1  1 21];  f(n) = -dPK1 + LOR1*(aK1*F11^2 + bK1*F11 + cK1);
% Pump K1 (from tank 1)
    n=n+1; ff(n,:)=[ 1  1 22];  f(n) = -dPK2 + LOR2*(aK2*F12^2 + bK2*F12 + cK2);
% Pump K2 (from tank 2)
    n=n+1; ff(n,:)=[ 1  1 23];  f(n) = -dPK4 + aK4*F14^2 + bK4*F14 + cK4;
 % Pump K4 (from tank 4)

        % PRESSURE DROP OVER THE VALVES
        % X01, X02, X03, X04, X05, X10, X11 and X13 are the fraction of the total flow
area
        % of the 01, 02, 03, 04, 05, 10, 11 and 13 valves respectively (inputs)
        % The alphai constant is equal to (ro2/Cvi), where ro is a density and Cvi the
valve
        % size coefficient

    n=n+1; ff(n,:)=[ 1  1 24];  f(n) = -sqrt(dPCV01)*X01 + alpha01*F01;
 % Input to tank 1
    n=n+1; ff(n,:)=[ 1  1 25];  f(n) = -sqrt(dPCV02)*X02 + alpha02*F02;
% Input to tank 2
    n=n+1; ff(n,:)=[ 1  1 26];  f(n) = -sqrt(dPCV03)*X03 + alpha03*F03;
% Input to tank 3
    n=n+1; ff(n,:)=[ 1  1 27];  f(n) = -sqrt(dPCV04)*X04 + alpha04*F04;
% Input to tank 4
    n=n+1; ff(n,:)=[ 1  1 28];  f(n) = -sqrt(dPCV05)*X05 + alpha05*F05;
 % Input to the column
    n=n+1; ff(n,:)=[ 1  1 29];  f(n) = -sqrt(dPCV10)*X10 + alpha10*F10;
% Heater exchange H1
    n=n+1; ff(n,:)=[ 1  1 30];  f(n) = -sqrt(dPCV11)*X11 + alpha11*F11;
 % ouput from tank 1
    n=n+1; ff(n,:)=[ 1  1 31];  f(n) = -sqrt(dPCV13)*X13 + alpha13*F13;
% output from tank 3

        % PRESSURE DROP OVER THE PIPES
    n=n+1; ff(n,:)=[ 1  1 34];  f(n) = - kL01*F01^2                 +
(P12 - 0 ) + rog*(hP12 - h01 )*LOR2  - dPCV01;

    n=n+1; ff(n,:)=[ 1  1 35];  f(n) = - kL02*F02^2                 + (P12 - 0 ) + rog*(hP12 -
h02 )*LOR2  - dPCV02;

    n=n+1; ff(n,:)=[ 1  1 32];  f(n) = - kL03*F03^2                 + (P12 - 0 ) + rog*(hP12 -
h03 )*LOR2  - dPCV03;

    n=n+1; ff(n,:)=[ 1  1 33];  f(n) = - kL04*F14^2                 + (P04 - 0 ) + rog*(hP04 -
h04 );

    n=n+1; ff(n,:)=[ 1  1 34];  f(n) = - kL05*F05^2                 + (P12 - 0 ) + rog*(hP12 -
h05 )*LOR2  - dPCV05;

    n=n+1; ff(n,:)=[ 1  1 35];  f(n) = - kL10*F10^2                 + (P10 - P19) + rog*(hP10 -
hP19)               - dPCV10;

    n=n+1; ff(n,:)=[ 1  1 30];  f(n) = - kL11*F11^2                 + (0  - 0 ) + rog*(h11  - h02
)*LOR1  - dPCV11  + dPK1;

    n=n+1; ff(n,:)=[ 1  1 36];  f(n) = - kL12*F12^2                 + (0  - P12) + rog*(h12  -
hP12)*LOR2          + dPK2;

    n=n+1; ff(n,:)=[ 1  1 31];  f(n) = - kL13*F13^2                 + (0  - 0 ) + rog*(h13  - h02
)*LOR3  - dPCV13;

    n=n+1; ff(n,:)=[ 1  1 33];  f(n) = - kL14*F14^2                 + (0  - P10) + rog*(h14  -
hP10)               + dPK4;

    n=n+1; ff(n,:)=[ 1  1 36];  f(n) = - kL15*F15^2                 +
- 0 ) + rog*(h15  - h02 )*LOR5;
 (0
    n=n+1; ff(n,:)=[ 1  1 35];  f(n) = - kL19*(F14-F10)^2
+ (P10 - P19) + rog*(hP10 - hP19);

    n=n+1; ff(n,:)=[ 1  1 37];  f(n) = - kL20*F14^2                 + (P10 - P30) + rog*(hP10 -
hP30);

    n=n+1; ff(n,:)=[ 1  1 38];  f(n) = - kL31*F31^2                 + (P30 - P04) + rog*(hP30 -
hP04);

    n=n+1; ff(n,:)=[ 1  1 38];  f(n) = - kL40*F04^2                 + (P30 - P04) + rog*(hP30 -
hP04)               - dPCV04;

% ----------------TO BE SET BY USER ABOVE---------

    % STORE COUNT
    nf=n;  % No of equations
                                        % UA.deltaT_LM (kW)
    if INIT

      INIT = 0;

% Selections to be included in solution
        nfs  = sum(ff(1:nf,1));          % Number of selected functions
                        nys  = sum(yy(1:ny,2));
% Number of selected variables
        nss  = sum(yy(1:ns,2));          % Number of selected states only
        if nf>ns
          nos  = sum(ff(ns+1:nf,1));   % Will include all selected equations plus states if
observed
        else
          nos = 0;
        end
        noss = 0;
        for i=1:ns                                  % UA.deltaT_LM
          if ((yy(i,2)==1) & (yy(i,3)==1))
            nos  = nos + 1;                         % Number of selected & observed
variables
            noss = noss + 1;        % Number of observed states
          end
        end

          R=sparse(nos,nos);
        Q=sparse(nys,nys);
        Jy=sparse(nys,nys);

        if noss>0
          Lt=sparse(noss,nss);
      % Selection matrix for observed states
        end              % flow through geyzers
        Mt=0.01*speye(nys,nys);    % Initialise filter covariance matrix (sparse)
        M0=sparse(nys,nys);        % zeros
        K=sparse(nys,nos);           % Set up Kalman gain matrix
        us=zeros(nys,1);

                        % make lookup tables
                        jy=zeros(nys,1);
        i=0;
        for j=1:ny
          if yy(j,2)==1
            i=i+1;
            jy(i)=j;
                        end
        end
                        jf=zeros(nfs,1);
        i=0;
        for j=1:nf
                        if ff(j,1)==1
          i=i+1;
          jf(i)=j;
                        end
        end
        % lookup table for yy parameters
        luyy=zeros(ny,4,6);
        ilast=0;
        for j=1:ny
          i=yy(j,1);
          if ilast~=i
            ilast=i;
            icounter=1;
          else
            icounter=icounter+1;
          end
          luyy(i,icounter,:)=yy(j,:);
        end

        % R Matrix
        i=0;
        for j=1:ns
          if ((yy(j,2)==1)&(yy(j,3)==1))
            i=i+1;
                  if yy(j,4)<=0
```

```
                R(i,i)=(yy(j,4)*yo_err_factor)^2;
              else
              R(i,i)= (yy(j,4)*yo_err_factor*max(yy(j,6)-yy(j,5),minrange)/100)^2;
            end
          end
        end
        if nf>ns
          kklast=0;
          for j=ns+1:nf
            if ff(j,1)==1
              i=i+1;
              kk=ff(j,3);  % lookup index
              if kk~=kklast
                kklast=kk;
                icounter=1;
              else
                icounter=icounter+1;
              end
              if ff(j,2)>0
                R(i,i)= (f_err_factor*ff(j,2)*max(luyy(kk,icounter,6)-
luyy(kk,icounter,5),minrange)/100)^2;
              else
                R(i,i)= (f_err_factor*ff(j,2))^2;
              end
            end
          end
        end

        % Q Matrix
        i=0;
        kklast=0;
                        for j=1:ns
                        if ff(j,1)==1
          i=i+1;
          kk=ff(j,3);  % lookup index
          if kk~=kklast
            kklast=kk;
            icounter=1;
          else
            icounter=icounter+1;
          end
          if ff(j,2)>0
            Q(i,i)= (f_err_factor*ff(j,2)*max(luyy(kk,icounter,6)-
luyy(kk,icounter,5),minrange)/100)^2;
          else
            Q(i,i)= (f_err_factor*ff(j,2))^2;
          end
                        end
                        end
        if ny>ns
          for j=ns+1:ny
            if yy(j,2)==1
              if yy(j,3)==1  % observed ?
                factor=yo_err_factor;
              else
                factor=yu_err_factor;
              end
                  i=i+1;
                  % Q Matrix
                        if yy(j,4)<=0
                                Q(i,i)=(yy(j,4)*factor)^2;
                        else
                        Q(i,i)= (yy(j,4)*factor*max(yy(j,6)-yy(j,5),minrange)/100)^2;
                                end
                                end
            end
          end

          % Check
          if i~=nys
            printf('\n\n #### ERROR #### Must Select Same Equations as States ! \n\n');
            halt;
          end

                        fs  = zeros(nfs,1);         % selected function values
          ys  = zeros(nys,1);   % selected variables
          ws  = zeros(nos,1);          % selected & observed variables
          zos = zeros(nys-nss,1); % partly observed, partly previous values

        end2

            if no==1
            f0=f;
                    else
            for i=1:nfs
            df=f(ff(i))-f0(ff(i));
            Jy(i,jj)=df/dyjj;
            end
            y(yjj)=y(yjj)- dyjj;
          end
          if ne==(1+nys)
            break;     % break out of the n_evals loop
          end
        end
              f = f0;  % back to original position
              % load fs selection from f
              for i=1:nfs
              fs(i)=f(ff(i));
        end

        % Fill out yo vector
              for i=1:ny
          if (yy(i,3)~=1)
            yo(i)=y(i);
          end
        end


        % SOLVE

        % Set zos,ys and first part of ws
        io=0;
              for i=1:nss
          ys(i)=y(yy(i));
          if yy(yy(i),3)==1
            io=io+1;
            ws(io)=yo(yy(i));
            Li(io,i)=1;  % Selection Matrix
          end
        end
        if nys>nss
          for i=nss+1:nys
            zos(i-nss)=yo(yy(i));
          end
        end

        if REEVALUATE
              AA = Jy(1:nss,1:nss);
              BB = Jy(1:nss,nss+1:nys);
              CC = Jy(nss+1:nfs,1:nss);
          DD = Jy(nss+1:nfs,nss+1:nys);

              Et = -speye(nys-nss,nys-nss)/Tau;
              P = sparse(nys,nys);
              P(1:nss,1:nss) = AA;
              P(1:nss,nss+1:nys) = BB;
              P(nss+1:nys,nss+1:nys) = Et;

              % for singular P use series to find "expmPdt_IdivP" = [expm(P*dt)-I]*P^-1
              change=99;
              expmPdt_IdivP=dt*speye(nys,nys);
              changemat=dt*speye(nys,nys);
              Pdt=P*dt;
          n=1;
              while change>tol
              n=n+1;
              changemat=(changemat*Pdt)/n;
              change=sum(sum(abs(changemat)));
        % makes a 1-by-n vector with the sum of the columns as its entries
              expmPdt_IdivP=expmPdt_IdivP+changemat;
              end;

              % Now integrate using matrix exponential
              At=expmPdt_IdivP*P+speye(nys,nys);
              Bt=expmPdt_IdivP;

              % Observation Matrix
              Ct=sparse(nos,nys);
              if noss>0
              Ct(1:noss,1:nss)=Li;
              end
              Ct(noss+1:nos,1:nss)=CC;
          Ct(noss+1:nos,nss+1:nys)=DD;
        end;

        % Working Vectors
        Ft = fs(1:nss) - AA*ys(1:nss) - BB*ys(nss+1:nys);
        Gt = fs(nss+1:nfs) - CC*ys(1:nss) - DD*ys(nss+1:nys);
        Ht = zos/Tau;

        % Augmented System
        us(1:nss) = Ft;
        us(nss+1:nys) = Ht;

        % Load rest of Observation Vector ws (first part loaded above)
        for i=noss+1:nos
          ws(i)=-Gt(i-noss);
        end

        % KALMAN FILTER
        nKint=nKint+1;
        nKcomp=nKcomp+1;
        if (nKint>Kint)|(nKcomp<=Kcomp)
          nKint=1;
        end
        if nKint==1    % re-evaluate Kalman "K" every "Kint" steps only
                       % K = Mt*Ct'*inv(Ct*Mt*Ct' + R);
          CMCR=Ct*Mt*Ct' + R;
          CM=Ct*Mt;
          K=(CMCR\CM)';
          Mt = At*(speye(nys,nys) - K*Ct)*Mt*At' + Q;
          MP=Mt-SMM*spones(Mt);
          MP=max(M0,MP);    % chop off low positives
          MN=Mt+SMM*spones(Mt);
          MN=min(MN,M0);    % chop off small negatives
          Mt=MP+MN;
        end
        % Actually operate the filter...
        ys = At*ys + Bt*us + K*(ws-Ct*ys);

        % clipping
        for j=1:nys
          if ys(j)<yy(yy(j),5)
            ys(j)=yy(yy(j),5);                    % low clip
            ylimflag(yy(j),2)=1;
            ylimflag(yy(j),3)=yy(yy(j),5);
          else
            if ys(j)>yy(yy(j),6)
              ys(j)=yy(yy(j),6);                  % high clip
              ylimflag(yy(j),2)=2;
              ylimflag(yy(j),3)=yy(yy(j),6);
            else
              ylimflag(yy(j),2)=0;
              ylimflag(yy(j),3)=-99;
            end
          end
        end

        err2=0;
        err2_deriv=0;
        ntotvar=0;
        ntotderiv=0;
        i=0;
        for j=1:nss
          if yy(yy(j),3)==1
            i=i+1;
            err2 = err2 + ((ys(j)-yo(yy(j)))*yo_err_factor)^2/R(i,i);
        % observed states
            ntotvar=ntotvar+1;
          end
        end
        for i=nss+1:nys
```

```
    if yy(jy(j),3)==1
      err2 = err2 + ((ys(j)-yc(jy(j)))*yo_err_factor)^2/Q(j,j);
% errors in observed 'z'
      ntotvar=ntotvar+1;
    end
  end
  i=noss;
  ii=0;
  for j=1:nf
    if jc==ns
      if ff(j,1)==1    % selected equation / state
        ii=ii+1;
        err2_deriv = err2_deriv + (f(j)*f_err_factor)^2/Q(ii,ii);  % derivatives
        ntotderiv=ntotderiv+1;
      end
    else
      if ff(j,1)==1    % selected equation
        i=i+1;
        err2 = err2 + (f(j)*f_err_factor)^2/R(i,i);
% compensate for the factor in RR
        ntotvar=ntotvar+1;
      end
    end
  end

  wt_ob_er = sqrt(err2/ntotvar);            % to see how well it is doing (maybe should
use (ntotvar-1) )
  wt_deriv_er = dt*sqrt(err2_deriv/ntotderiv);  % to see how unsteady the process
should be compared with actual


  % load back to full vector
        for j=1:nys
        y(jy(i))=ys(i);
        end

  % store for plotting
  if (t-tlastplot)>=(0.9*dtplot)
    iplot=iplot+1;
    tp(iplot)=t-dt; % NOTE: these are one step out, thus dt subtracted
    tlastplot = t;
                h11P(iplot)=h11;
                h12P(iplot)=h12;
    h13P(iplot)=h13-h03BOT;
    h15P(iplot)=h15-h05BOT;
    T11P(iplot)=T11;
    T12P(iplot)=T12;
    T13P(iplot)=T13;
    T14P(iplot)=T14;
    T15P(iplot)=T15;
    F01P(iplot)=F01;
    F02P(iplot)=F02;
    F03P(iplot)=F03;
    F04P(iplot)=F04;
    F05P(iplot)=F05;
    F10P(iplot)=F10;
    F11P(iplot)=F11;
    F12P(iplot)=F12;
    F13P(iplot)=F13;
    F14P(iplot)=F14;
    F15P(iplot)=F15;
    dPK1P(iplot)=dPK1;
    dPK2P(iplot)=dPK2;
    dPK4P(iplot)=dPK4;
    dPCV01P(iplot)=dPCV01;
    dPCV02P(iplot)=dPCV02;
    dPCV03P(iplot)=dPCV03;
    dPCV04P(iplot)=dPCV04;
    dPCV05P(iplot)=dPCV05;
    dPCV10P(iplot)=dPCV10;
    dPCV11P(iplot)=dPCV11;
    dPCV13P(iplot)=dPCV13;
    P04P(iplot)=P04;
    P10P(iplot)=P10;
    P12P(iplot)=P12;
    P19P(iplot)=P19;
    P30P(iplot)=P30;
    T02P(iplot)=T02;
    T04P(iplot)=T04;
    T19P(iplot)=T19;
    T20P(iplot)=T20;
    T30P(iplot)=T30;
    T31P(iplot)=T31;
    qH1P(iplot)=qH1;
    qH2P(iplot)=qH2;
    qH3H4P(iplot)=qH3H4;
    X01P(iplot)=X01;
    X02P(iplot)=X02;
    X03P(iplot)=X03;
    X04P(iplot)=X04;
    X05P(iplot)=X05;
    X10P(iplot)=X10;
    X11P(iplot)=X11;
    X13P(iplot)=X13;
    wtd_obs_err(iplot)=wt_ob_er;
    wtd_deriv_err(iplot)=wt_deriv_er;
  end

end

% ----------TO BE SET BY USER BELOW----------

% Draw graphs

figure(1);
plot(tp(:),h11P(:),'r',tp(:),h12P(:),'g',tp(:),h13P(:),'b',tp(:),h15P(:),'y');
legend('h11P','h12P','h13P','h15P');
axis([0 tp(iplot) 0 2.5]);

figure(2);
plot(tp(:),T02P(:),'m',tp(:),T04P(:),'c',tp(:),T19P(:),'r',...
    tp(:),T12P(:),'m',tp(:),T13P(:),'g',tp(:),T14P(:),'c',...
    tp(:),T15P(:),'b',tp(:),T19P(:),'r',tp(:),T20P(:),'g',...
    tp(:),T30P(:),'b',tp(:),T31P(:),'y');
legend('T02','T04','T11','T12','T13','T14','T15','T19','T20','T30','T31');
axis([0 tp(iplot) 15 85]);
```

```
figure(3);
plot(tp(:),F01P(:),'r',tp(:),F02P(:),'g',tp(:),F03P(:),'b',...
    tp(:),F04P(:),'y',tp(:),F05P(:),'c',tp(:),F10P(:),'m',...
    tp(:),F11P(:),'r',tp(:),F12P(:),'g',tp(:),F13P(:),'b',...
    tp(:),F14P(:),'y',tp(:),F15P(:),'c');
legend('F01','F02','F03','F04','F05','F10','F11','F12','F13','F14','F15');
axis([0 tp(iplot) 0 6]);

figure(4);
plot(tp(:),dPK1P(:),'r',tp(:),dPK2P(:),'g',tp(:),dPK4P(:),'b');
legend('dPK1','dPK2','dPK4');
axis([0 tp(iplot) 0 12]);

figure(5);
plot(tp(:),dPCV01P(:),'r',tp(:),dPCV02P(:),'g',tp(:),dPCV03P(:),'b',...
    tp(:),dPCV04P(:),'y',tp(:),dPCV05P(:),'c',tp(:),dPCV10P(:),'m',...
    tp(:),dPCV11P(:),'r',tp(:),dPCV13P(:),'g');
legend('dPCV01','dPCV02','dPCV03','dPCV04','dPCV05','dPCV10','dPCV11','dPCV13');
axis([0 tp(iplot) 0 12]);

figure(6);
plot(tp(:),P04P(:,1),'r',tp(:),P10P(:,1),'g',tp(:),P12P(:,1),'b',...
    tp(:),P19P(:,1),'c',tp(:),P30P(:,1),'m');
legend('P04','P10','P12','P19','P30');
axis([0 tp(iplot) 0 12]);

figure(7);
plot(tp(:),qH1P(:,1),'r',tp(:),qH2P(:,1),'g',tp(:),qH3H4P(:,1),'b');
legend('qH1','qH2','qH3H4');
axis([0 tp(iplot) -0.8 80]);

figure(8);
plot(tp(:),X01P(:),'r',tp(:),X02P(:),'g',tp(:),X03P(:),'b',...
    tp(:),X04P(:),'y',tp(:),X05P(:),'c',tp(:),X10P(:),'m',...
    tp(:),X11P(:),'r',tp(:),X13P(:),'g');
legend('X01','X02','X03','X04','X05','X10','X11','X13');
axis([0 tp(iplot) 0 1]);

figure(9);
plot(tp(:),wtd_obs_err(:),'r',tp(:),wtd_deriv_err(:),'b');
legend('wtd-obs-err','wtd-deriv-err');

% ----------TO BE SET BY USER ABOVE----------
```

Appendix D

Pump-tank and Training Plant Step Responses

True and mismatched unit step responses, obtained using experimental data from the Pump-tank system (Appendix D.1) and the 2-input / 2-output sub-system of the Training Plant (Appendix D.2) are provided. These step responses were used for off-line simulation and real time applications in Chapters 6 and 7 respectively.

Appendix D.1 Dynamic matrix coefficients of the true and mismatched process models for the 2-input / 2-output Pump-tank system (M = 10)

	True model		Mismatched model	
Variables	V1	V2	V1	V2
L1	-0.186	-0.036	-0.372	-0.036
	-0.317	-0.059	-0.634	-0.059
	-0.416	-0.074	-0.832	-0.074
	-0.470	-0.081	-0.940	-0.081
	-0.502	-0.086	-1.004	-0.086
	-0.516	-0.088	-1.032	-0.088
	-0.523	-0.090	-1.046	-0.090
	-0.525	-0.092	-1.050	-0.092
	-0526	-0.092	-1.052	-0.092
	-0.526	-0.092	-1.052	-0.092
L2	-0.026	0.157	-0.026	0.316
	-0.044	0.256	-0.044	0.532
	-0.056	0.366	-0.056	0.732
	-0.067	0.450	-0.067	0.900
	-0.078	0.519	-0.078	1.040
	-0.086	0.569	-0.086	1.140
	-0.093	0.614	-0.093	1.223
	-0.098	0.660	-0.098	1.320
	-0.101	0.694	-0.101	1.388
	-0.101	0.718	-0.101	1.436

The diagonal responses of the mismatched process model are both twice their correct magnitudes.

Appendix D.2 Dynamic matrix coefficients of the true and mismatched process models for the 2 inputs / 2 outputs system from the Plant Simulator (M = 5)

	True model		Mismatched model	
Variables	CV01	CV03	CV01	CV03
L1	0.131	0.003	0.261	0.003
	0.261	0.006	0.523	0.006
	0.392	0.009	0.784	0.009
	0.523	0.012	1.046	0.012
	0.654	0.015	1.307	0.015
L2	-0.127	-0.073	-0.127	-0.036
	-0.254	-0.145	-0.254	-0.073
	-0.382	-0.218	-0.382	-0.109
	-0.509	-0.290	-0.509	-0.145
	-0.636	-0.363	-0.636	-0.182

The diagonal responses of the mismatched process model are, the first one 2 times and the second ½ of their correct magnitudes.

Appendix E

Extracts of the Integrating Adaptive Dynamic Matrix Control Algorithm from the SCADA System

The Integrating Adaptive Dynamic Matrix Control technique, developed in the present work, was implemented in the existing Linear Dynamic Matrix Control (LDMC) algorithm developed by Mulholand and Prosser [1997], within a flexible SCADA system at the School of Chemical Engineering. The contributions, updates and programming expertise of Professor Mulholland must be fully acknowledged at this point. This program would never have been completed without his valuable input.

E.1 DMCObject

```
#ifndef _DMCObject_h_
#define _DMCObject_h_

// CDMCObject.h ; header file
//

// 20/10/97 Rev. 26: CASE 3 (same as previous code)

// prototypes
class CMatrix;
class CVector;
class CLinearProgram;


////////////////////////////////////////////////////
// CDMCObject class

#include "IOStream.h"
#include "DMCPlotWnd.h"
#include "vector.h"
#include "IODeviceModel.h"
#include "math.h"

class CDMCObject : public CScadObject
{
DECLARE_SERIAL (CDMCObject)

friend class CDlgDMCObject;
friend class CDMCStream;
friend class CIODeviceModel;

// private member variables
private:
  CDMCPlotWnd m_PlotWnd;
  CMatrix* m_pDynMatrix;        // the dynamic matrix
  CMatrix* m_pDynMatrix_Set;    // store for set dynamic matrix
  CMatrix* m_pOpenLoopMatrix;
  CMatrix* m_pOffsetMatrix;
  CMatrix* m_pOpenLoopMatrix_Set;
  CMatrix* m_pOffsetMatrix_Set;
  CMatrix* m_pWeightingMatrix;
  CMatrix* m_pMoveSupMatrix;
  CMatrix* m_pLeastSqrInvMatrix;
  CMatrix* m_pLowerTrilDMatrix;
  CVector* m_pPastOutputChanges;
  DWORD    m_dwPrevTime;        // last update time DMC
  DWORD    m_dwPrevTimeAdapt;   // last update time Adapter

  double           m_fTimeInterval;     // update interval
                   DWORD m_dwTimeTillUpdate; //###SD000815
// time till next update
  CString          m_cStepResponseFile;   // file containing step response
measurements
  int                              m_iSteadyStateHorizon;  // M
  int                              m_iNumberControlMoves;  // N
  int                              m_iOptimisationHorizon; // P
  int                              iNumInputs;
  int                              iNumOutputs;
  BOOL   m_bLogToFile;          // only used for serialisation and editing

  CObArray     m_DMCStreams;
  CStringArray m_csInputStreamNames;    // names of input streams
  CObArray     m_pInputStreams;         // pointers to input streams
               CDWordArray  m_fSetpoints;          // control-loop setpoints
               CDWordArray  m_fInputMins;          // lower limit on input
               CDWordArray  m_fInputMaxs;          // upper limit on input
  CDWordArray  m_fMoveSuppressionFactors; // lambda values

  CStringArray m_csOutputStreamNames;  // names of input streams
  CObArray     m_pOutputStreams;       // pointers to input streams
CDWordArray  m_fMaxChanges;          // maximum change allowed eg, valve
CDWordArray  m_fOutputMins;          // lower limit on output
CDWordArray  m_fOutputMaxs;          // upper limit on output
CDWordArray  m_fOutputWeights;       // weights on output var.
CDWordArray  m_fAdaptWtModErr;       // weights adaptation model errors
CDWordArray  m_fAdaptWtMeasErr;      // weights adaptation measurement
// errors
CDWordArray  m_fAdaptWtCurve;        // weights adaptation curves.
CDWordArray    m_fTimeFactors;       // factors on trajectory points
CDWordArray    m_fTimeFactorPositions;  // non-zero factor positions on tra
// jectories

  // variables for Linear Programming thread
  CLinearProgram* m_pLProg;
  CVector*      m_pResult;
  CVector*      m_pObjectiveFn;
  CVector*      m_pDeltaMUOO;
  CVector*      m_pPresentOutput;
  HANDLE        m_hThread;
  BOOL          m_bLPThreadFinished;
  BOOL          m_bLPThreadRunning;
  BOOL          m_bIgnoreLPThreadResult;  // for changes while LP thread is
// executing
  int           m_iLPError;

  // variables for Adaptive DMC - IG
  CMatrix* m_pB0ad;
  CMatrix* m_pBOLad;
  CMatrix* m_pBOL_INTEG;
  CMatrix* m_pB0ad_orig;
  CMatrix* m_pBOLad_orig;
  CMatrix* m_pBOL_INTEG_orig;
  CVector* m_pDoublePastOutputChangesad;
  CVector* m_pead;
  CVector* m_pPead;  // Present
  CVector* m_pLead;  // Last
  CVector* m_pSPead;  // Delta (change) - smoothed
  CVector* m_pDx0ad;
  CVector* m_ppad;
  CVector* m_ppad_orig;
  CMatrix* m_pGad;
  CMatrix* m_pGad1;
  CMatrix* m_pGad2;
  CMatrix* m_pGad3;
  CMatrix* m_pQad;
  CMatrix* m_pRad;
  CMatrix* m_plad;
  CMatrix* m_pMad;
  CMatrix* m_pKad;
  CMatrix* m_pAad;
  CMatrix* m_pBad;
  CVector* m_ppadw;
  CVector* m_ppadw_orig;
  CMatrix* m_pGadw;
  CMatrix* m_pQadw;
  CMatrix* m_pladw;
  CMatrix* m_pMadw;
  CMatrix* m_pKadw;
  CMatrix* m_pAadw;
  CMatrix* m_pBadw;
  CMatrix* m_pFadw;
  CMatrix* m_pxStackad;
  BOOL     m_bRestartad;
  BOOL             m_bEnableAdapt;
  BOOL             m_bUseAdapt;
  BOOL             m_bUseAdapt_last;
  BOOL             m_bRegularizeAdapt;
  BOOL             m_bExtSlope_Int;
  BOOL             m_bAccMoves_Int;
  BOOL             m_bGradFback_Int;
  double   m_fGradSmooth_Int;
  double   m_fQad;
  double   m_fABad;
  CVector* m_pPreviousOutput;
  CVector* m_pPreviousOutputChange;
  CVector* m_pPreviousInput;
  CMatrix* m_pTest;
  CVector* AccumulatedMoveINTEGCORR,
  CVector* AccumulatedMoveINTEGCORRad,
  BOOL             m_bSuppressINTEGad;

// public member variables
public:
  CString          cStepResponseFile;  // Variables for IODeviceModel
  int                                  nI;
  int                                  nO;
  int                                  nS;
  double                       modint;

// private member functions
private:
  void SetName (CString csName);
  void CalcDMCParameters (void);
  void SetTimeInterval (double fTimeInterval)
       {m_fTimeInterval = fTimeInterval;}
  void SetAdapterQ (double fQad)
       {m_fQad = fQad;}
  void SetAdapterAB (double fABad)
       {m_fABad = fABad;}
  void SetStepResponseFile (CString cStepResponseFile)
       {m_cStepResponseFile = cStepResponseFile;}
  void SetSteadyStateHorizon (int iSteadyStateHorizon);
  void SetNumberControlMoves (int iNumberControlMoves);
  void SetOptimisationHorizon (int iOptimisationHorizon);
  void SetLogging (BOOL bLogToFile);
  void SetInputStreamNames (const CStringArray& csInputStreamNames);
  void SetSetpoints (const CDWordArray& fSetpoints);
  void SetInputMins (const CDWordArray& fInputMins);
  void SetInputMaxs (const CDWordArray& fInputMaxs);
  void SetMoveSuppressionFactors (const CDWordArray& fMoveSuppressionFactors);
  void SetOutputStreamNames (const CStringArray& csOutputStreamNames);
  void SetMaxChanges (const CDWordArray& fMaxChanges);
  void SetOutputMins (const CDWordArray& fOutputMins);
  void SetOutputMaxs (const CDWordArray& fOutputMaxs);
  void SetOutputWeights (      const CDWordArray& fOutputWeights,

                               const CDWordArray& fAdaptWtModErr,

                               const CDWordArray& fAdaptWtCurve,

                               const CDWordArray& fAdaptWtMeasErr);
  void SetTimeFactors (const CDWordArray& fTimeFactors);
  void SetIODeviceModelData (void);
  void AdaptMatrices (void);

  void StreamWrite (int iIndex, double fValue);
  CString StreamGetVal (int iIndex);

  void LinearProgramThread (void);
  static DWORD __stdcall ThreadFunction (void* pThis)
    {((CDMCObject*)pThis)->LinearProgramThread (); return 0;}

// constructor and destructor
public:
  CDMCObject ();
  ~CDMCObject ();

// public member functions
public:
  void Serialize (CArchive& ar);
  void Initialise (CScadDoc* pScadDoc);
  void TimerUpdate (DWORD dwTime);
  void SetEnable (BOOL bEnabled);
  void SetEnableAdapt (BOOL bEnableAdapt);
  void SetUseAdapt (BOOL bUseAdapt);
  void SetRegularizeAdapt (BOOL bRegularizeAdapt);
  void SetExtSlope_Int (BOOL bExtSlope_Int);
  void SetAccMoves_Int (BOOL bAccMoves_Int);
  void SetGradFback_Int (BOOL bGradFback_Int);
  void SetGradSmooth_Int (double fGradSmooth_Int);
               //###SD000817
               DWORD WhatisTimeTillUpdate(void) {return m_dwTimeTillUpdate;}
               BOOL IsEnabled (void) {return m_bEnabled;}
               //###
};
#endif
```

E.2 DMCStream

```
// DMCStream.cpp : implementation File
//


#include "stdafx.h"
#include <math.h>
#include "resource.h"
#include "DMCObject.h"
#include "DlgDMCObject.h"
#include "MainWnd.h"
#include "ScadDoc.h"
#include "DMCStream.h"
#include "Matrix.h"
#include "LinearProgram.h"
#include "IODeviceModel.h"
#include "Converter.h"
#include "stdio.h"

IMPLEMENT_SERIAL (CDMCObject, CScadObject, 1)

CDMCObject::CDMCObject ()
{
 m_pDynMatrix = NULL;
 m_pDynMatrix_Set = NULL;
 m_pOpenLoopMatrix = NULL;
 m_pOffsetMatrix = NULL;
 m_pOpenLoopMatrix_Set = NULL;
 m_pOffsetMatrix_Set = NULL;
 m_pWeightingMatrix = NULL;
 m_pMoveSupMatrix = NULL;
 m_pLeastSqrInvMatrix = NULL;
 m_pLowerTrilDMatrix = NULL;
 m_pPastOutputChanges = NULL;

   ////##SD000815
 m_dwPrevTime = GetTickCount ();
 m_dwTimeTillUpdate = GetTickCount();
   ////##

 // DMC_IGadapt
 m_pB0ad = NULL;
 m_pBOLad = NULL;
 m_pBOL_INTEG = NULL;
 m_pB0ad_orig = NULL;
 m_pBOLad_orig = NULL;
 m_pBOL_INTEG_orig = NULL;
 m_pead = NULL ;
 m_pPead = NULL ;
 m_pLead = NULL ;
 m_pSPead = NULL ;
 m_pDx0ad = NULL ;
 m_ppad = NULL ;
 m_ppad_orig = NULL ;
 m_pGad = NULL ;
 m_pGad1 = NULL ;
 m_pGad2 = NULL ;
 m_pGad3 = NULL ;
 m_pQad = NULL ;
 m_pRad = NULL ;
 m_pKad = NULL ;
 m_pxStackad = NULL ;
 m_pAad = NULL ;
 m_pBad = NULL ;
 m_plad = NULL ;
 m_ppadw = NULL ;
 m_ppadw_orig = NULL ;
 m_pGadw = NULL ;
 m_pQadw = NULL ;
 m_pKadw = NULL ;
 m_pAadw = NULL ;
 m_pBadw = NULL ;
 m_pladw = NULL ;
 m_pFadw = NULL ;
 m_pPreviousOutput = NULL;
 m_pPreviousOutputChange = NULL;
 m_pDoublePastOutputChangesad = NULL;
 m_pPreviousInput = NULL;
 m_pTest = NULL ;
 AccumulatedMoveINTEGCORR = NULL;
 AccumulatedMoveINTEGCORRad = NULL;
 m_bSuppressINTEGad = FALSE;     // #### MM001002 To Suppress adapation in
Integrating Systems


 m_dwPrevTime = GetTickCount ();
 m_dwPrevTimeAdapt = GetTickCount ();

 // initialise DMC thread flags
 m_bLPThreadFinished = FALSE;
 m_bLPThreadRunning = FALSE;
 m_bIgnoreLPThreadResult = FALSE;
 m_cStepResponseFile = _T("");   // Initialise as signal not set yet
 SetIODeviceModelData ();

 // Global Initialisations for Adaptive DMC - IG
 m_bRestartad = TRUE;
 m_bEnableAdapt = FALSE;
 m_bUseAdapt = FALSE;
 m_bUseAdapt_last = FALSE;
 m_bRegularizeAdapt = FALSE;
 m_bExtSlope_Int = TRUE;
 m_bAccMoves_Int = TRUE;
 m_bGradFback_Int = TRUE;
 m_fGradSmooth_Int = 0.2;
 m_fQad = 1.0;        // Q diagonal value for model confidence
 m_fABad = 1.0;       // diagonal value for Offset model confidence

}

CDMCObject::~CDMCObject ()
{
 if (m_pDynMatrix) delete m_pDynMatrix;
 if (m_pDynMatrix_Set) delete m_pDynMatrix_Set;
 if (m_pOpenLoopMatrix) delete m_pOpenLoopMatrix;
 if (m_pOffsetMatrix) delete m_pOffsetMatrix;
 if (m_pOpenLoopMatrix_Set) delete m_pOpenLoopMatrix_Set;
 if (m_pOffsetMatrix_Set) delete m_pOffsetMatrix_Set;
 if (m_pWeightingMatrix) delete m_pWeightingMatrix;
 if (m_pMoveSupMatrix) delete m_pMoveSupMatrix;
 if (m_pLeastSqrInvMatrix) delete m_pLeastSqrInvMatrix;
 if (m_pLowerTrilDMatrix) delete m_pLowerTrilDMatrix;
 if (m_pPastOutputChanges) delete m_pPastOutputChanges;
 if (m_pB0ad) delete m_pB0ad;
 if (m_pBOLad) delete m_pBOLad;
 if (m_pBOL_INTEG) delete m_pBOL_INTEG;
 if (m_pB0ad_orig) delete m_pB0ad_orig;
 if (m_pBOLad_orig) delete m_pBOLad_orig;
 if (m_pBOL_INTEG_orig) delete m_pBOL_INTEG_orig;
 if (m_pead) delete m_pead;
 if (m_pPead) delete m_pPead;
 if (m_pLead) delete m_pLead;
 if (m_pSPead) delete m_pSPead;
 if (m_pDx0ad) delete m_pDx0ad;
 if (m_ppad) delete m_ppad;
 if (m_ppad_orig) delete m_ppad_orig;
 if (m_pGad) delete m_pGad;
 if (m_pGad1) delete m_pGad1;
 if (m_pGad2) delete m_pGad2;
 if (m_pGad3) delete m_pGad3;
 if (m_pRad) delete m_pRad;
 if (m_pKad) delete m_pKad;
 if (m_plad) delete m_plad;
 if (m_pQad) delete m_pQad;
 if (m_pMad) delete m_pMad;
 if (m_ppadw) delete m_ppadw;
 if (m_ppadw_orig) delete m_ppadw_orig;
 if (m_pGadw) delete m_pGadw;
 if (m_pKadw) delete m_pKadw;
 if (m_pladw) delete m_pladw;
 if (m_pQadw) delete m_pQadw;
 if (m_pMadw) delete m_pMadw;
 if (m_pFadw) delete m_pFadw;
 if (m_pPreviousOutput) delete m_pPreviousOutput;
 if (m_pPreviousOutputChange) delete m_pPreviousOutputChange;
 if (m_pDoublePastOutputChangesad) delete m_pDoublePastOutputChangesad;
 if (m_pPreviousInput) delete m_pPreviousInput;
 if (m_pTest) delete m_pTest;
 if (AccumulatedMoveINTEGCORR) delete AccumulatedMoveINTEGCORR;
 if (AccumulatedMoveINTEGCORRad) delete AccumulatedMoveINTEGCORRad;

 // if LP thread is running, wait for it to stop and delete temp variables
 if (m_bLPThreadRunning)
 {
  for (;!m_bLPThreadFinished;);
  delete m_pLProg;
  delete m_pResult;
  delete m_pObjectiveFn;
  delete m_pDeltaMUQO;
   delete m_pPresentOutput;
 }
}

void CDMCObject::Serialize (CArchive& ar)
{
 CScadObject::Serialize (ar);

 if (ar.IsStoring ())
 {
     ar << m_fTimeInterval;
     ar << m_iSteadyStateHorizon;
     ar << m_iNumberControlMoves;
     ar << m_iOptimisationHorizon;
     ar << m_bLogToFile;
     ar << m_cStepResponseFile;
     ar << m_fQad;
     ar << m_fABad;
     ar << m_bEnableAdapt;
     ar << m_bUseAdapt;
     ar << m_bRegularizeAdapt;
     ar << m_bExtSlope_Int;
     ar << m_bAccMoves_Int;
     ar << m_bGradFback_Int;
     ar << m_fGradSmooth_Int;
 }
 else
 {
     ar >> m_fTimeInterval;
     ar >> m_iSteadyStateHorizon;
     ar >> m_iNumberControlMoves;
     ar >> m_iOptimisationHorizon;
    ar >> m_bLogToFile;
     ar >> m_cStepResponseFile;
     ar >> m_fQad;
     ar >> m_fABad;
     ar >> m_bEnableAdapt;
     ar >> m_bUseAdapt;
     ar >> m_bRegularizeAdapt;
     ar >> m_bExtSlope_Int;
     ar >> m_bAccMoves_Int;
     ar >> m_bGradFback_Int;
     ar >> m_fGradSmooth_Int;
 }
 // built-in serialization for CArray derived classes
 m_csInputStreamNames.Serialize (ar);
   m_fSetpoints.Serialize (ar);
   m_fInputMins.Serialize (ar);
   m_fInputMaxs.Serialize (ar);
 m_csOutputStreamNames.Serialize (ar);
   m_fMaxChanges.Serialize (ar);
   m_fOutputMins.Serialize (ar);
   m_fOutputMaxs.Serialize (ar);
 m_fMoveSuppressionFactors.Serialize (ar);
 m_fOutputWeights.Serialize (ar);
 m_fTimeFactors.Serialize (ar);
 m_fAdaptWtModErr.Serialize (ar);
 m_fAdaptWtMeasErr.Serialize (ar);
 m_fAdaptWtCurve.Serialize (ar);
 m_fTimeFactorPositions.Serialize (ar);
}
```

```
void CDMCObject::Initialise (CScadDoc* pScadDoc)
{
 CScadObject::Initialise (pScadDoc);

 // resize input pointer and DMC stream arrays
 m_pInputStreams.SetSize (m_csInputStreamNames.GetSize (), 10);
 m_DMCStreams.SetSize (m_csInputStreamNames.GetSize (), 10);

 // set pointers to input streams (using names array to get them)
 for (int i = 0; i < m_csInputStreamNames.GetSize (); i++)
 {
  for (int j = 0; j < m_pScadDoc->m_DocObjectArray.GetSize (); j++)
  {
   CIOStream* pIOStream = (CIOStream*)m_pScadDoc->m_DocObjectArray[j];
   if (pIOStream->GetName () == m_csInputStreamNames[i])
    m_pInputStreams[i] = pIOStream;
  }
 }

 // connect to new input streams and find/create DMC streams
 for (i = 0; i < m_pInputStreams.GetSize (); i++)
 {
  if (m_bEnabled)
   ((CIOStream*)m_pInputStreams[i])->Connect (m_csName);

  CString csTemp = m_csName + " - " + m_csInputStreamNames[i];
  // search for DMC stream
  int iIndex = -1;
  for (int j = 0; j < m_pScadDoc->m_DocObjectArray.GetSize (); j++)
   if (((CScadObject*)m_pScadDoc->m_DocObjectArray[j])->GetName () == csTemp)
    iIndex = j;

  // if found, set m_pDMCObject in DMC stream, otherwise create
  if (iIndex != -1)
  {
   m_DMCStreams[i] = m_pScadDoc->m_DocObjectArray[iIndex];
   ((CDMCStream*)m_DMCStreams[i])->SetDMCObject (this, i);
  }
  else
  {
   m_DMCStreams[i] = new CDMCStream (this, &csTemp, i);
   m_pScadDoc->m_DocObjectArray.Add (m_DMCStreams[i]);
  }
  // set logging
  ((CDMCStream*)m_DMCStreams[i])->SetLogging (m_bLogToFile);
 }

 // resize output pointer array
 m_pOutputStreams.SetSize (m_csOutputStreamNames.GetSize (), 10);

 // set pointers to output streams (using names array to get them)
 for (i = 0; i < m_csOutputStreamNames.GetSize (); i++)
 {
  for (int j = 0; j < m_pScadDoc->m_DocObjectArray.GetSize (); j++)
  {
   CIOStream* pIOStream = (CIOStream*)m_pScadDoc->m_DocObjectArray[j];
   if (pIOStream->GetName () == m_csOutputStreamNames[i])
    m_pOutputStreams[i] = pIOStream;
  }
 }

 // connect to new output streams
 if (m_bEnabled)
  for (i = 0; i < m_pOutputStreams.GetSize (); i++)
   ((CIOStream*)m_pOutputStreams[i])->Connect (m_csName);

 // calc DMC parameters
 CalcDMCParameters ();
}

void CDMCObject::SetName (CString csName)
{
 // check for change
 if (csName == m_csName)
  return;

 // set name in controlled streams
 if (m_bEnabled)
 {
  for (int i = 0; i < m_pInputStreams.GetSize (); i++)
  {
   ((CIOStream*)m_pInputStreams[i])->DisConnect (m_csName);
   ((CIOStream*)m_pInputStreams[i])->Connect (csName);
  }
  for (i = 0; i < m_pOutputStreams.GetSize (); i++)
  {
   ((CIOStream*)m_pOutputStreams[i])->DisConnect (m_csName);
   ((CIOStream*)m_pOutputStreams[i])->Connect (csName);
  }
 }

 // change names of DMC streams
 for (int i = 0; i < m_DMCStreams.GetSize (); i++)
 {
  CString csTemp = csName + " - " + m_csInputStreamNames[i];
  ((CDMCStream*)m_DMCStreams[i])->SetName (csTemp);
 }

 // call base class
 CScadObject::SetName (csName);
}

void CDMCObject::SetSteadyStateHorizon (int iSteadyStateHorizon)
{
 if (m_iSteadyStateHorizon != iSteadyStateHorizon)
 {
  m_iSteadyStateHorizon = iSteadyStateHorizon;
  m_bRestartad = TRUE;  // Force complete re-initialisation of Adaptive DMC - IG
  CalcDMCParameters ();
 }
}

void CDMCObject::SetNumberControlMoves (int iNumberControlMoves)
{
 if (m_iNumberControlMoves != iNumberControlMoves)
 {
  m_iNumberControlMoves = iNumberControlMoves;
  CalcDMCParameters ();
 }
}

void CDMCObject::SetOptimisationHorizon (int iOptimisationHorizon)
{
 if (m_iOptimisationHorizon != iOptimisationHorizon)
 {
  m_iOptimisationHorizon = iOptimisationHorizon;
  CalcDMCParameters ();
 }
}

// crude but effective
void CDMCObject::SetInputStreamNames (const CStringArray& csInputStreamNames)
{
 // compare to see if any changes have occurred
 if (m_csInputStreamNames.GetSize () == csInputStreamNames.GetSize ())
 {
  BOOL bIdentical = TRUE;
  for (int i = 0; i < m_csInputStreamNames.GetSize (); i++)
   if (m_csInputStreamNames[i] != csInputStreamNames[i])
    bIdentical = FALSE;

  // if no changes, exit (our job is done)
  if (bIdentical)
   return;
 }

 // if number of input streams has changed, recalc parameters
 if (m_csInputStreamNames.GetSize () != csInputStreamNames.GetSize ())
 {
  m_bRestartad = TRUE;  // Force complete re-initialisation of Adaptive DMC - IG
  CalcDMCParameters ();
 }
 for (int i = 0; i < m_pInputStreams.GetSize (); i++)
 {
  // disconnect from all inputs and delete all DMC streams
  if (m_bEnabled)
   ((CIOStream*)m_pInputStreams[i])->DisConnect (m_csName);

  // find DMC stream in CScadDoc and remove it, then delete stream
  int iIndex = -1;
  for (int j = 0; j < m_pScadDoc->m_DocObjectArray.GetSize (); j++)
   if (m_pScadDoc->m_DocObjectArray[j] == m_DMCStreams[i])
    iIndex = j;
  ASSERT (iIndex != -1);  // should have found it
  m_pScadDoc->m_DocObjectArray.RemoveAt (iIndex);
  delete (CDMCStream*)m_DMCStreams[i];
 }

 // resize input name, pointer and DMC stream arrays
 m_pInputStreams.SetSize (csInputStreamNames.GetSize (), 10);
 m_csInputStreamNames.SetSize (csInputStreamNames.GetSize (), 10);
 m_DMCStreams.SetSize (csInputStreamNames.GetSize (), 10);

 // copy names across
 for (i = 0; i < csInputStreamNames.GetSize (); i++)
  m_csInputStreamNames[i] = csInputStreamNames[i];

 // set pointers to input streams (using names array to get them)
 for (i = 0; i < m_csInputStreamNames.GetSize (); i++)
 {
  for (int j = 0; j < m_pScadDoc->m_DocObjectArray.GetSize (); j++)
  {
   CIOStream* pIOStream = (CIOStream*)m_pScadDoc->m_DocObjectArray[j];
   if (pIOStream->GetName () == m_csInputStreamNames[i])
    m_pInputStreams[i] = pIOStream;
  }
 }

 // connect to new input streams and create DMC streams
 for (i = 0; i < m_pInputStreams.GetSize (); i++)
 {
  if (m_bEnabled)
   ((CIOStream*)m_pInputStreams[i])->Connect (m_csName);

  CString csTemp = m_csName + " - " + m_csInputStreamNames[i];
  m_DMCStreams[i] = new CDMCStream (this, &csTemp);
  m_pScadDoc->m_DocObjectArray.Add (m_DMCStreams[i]);
 }
}

void CDMCObject::SetLogging (BOOL bLogToFile)
{
 // set logging for each of the DMC streams
 for (int i = 0; i < m_DMCStreams.GetSize (); i++)
  ((CDMCStream*)m_DMCStreams[i])->SetLogging (bLogToFile);

 // make a record of value for serialisation and editing
 m_bLogToFile = bLogToFile;
}

void CDMCObject::SetSetpoints (const CDWordArray& fSetpoints)
{
 // set size
 m_fSetpoints.SetSize (fSetpoints.GetSize (), 10);
 // set values
 for (int i = 0; i < fSetpoints.GetSize (); i++)
  m_fSetpoints[i] = fSetpoints[i];
}

void CDMCObject::SetInputMins (const CDWordArray& finputMins)
{
 // set size
 m_finputMins.SetSize (finputMins.GetSize (), 10);
 // set values
 for (int i = 0; i < finputMins.GetSize (); i++)
  m_finputMins[i] = finputMins[i];
}

void CDMCObject::SetInputMaxs (const CDWordArray& finputMaxs)
{
 // set size
```

```
 m_fInputMaxs.SetSize (fInputMaxs.GetSize (), 10);
 // set values
 for (int i = 0; i < fInputMaxs.GetSize (); i++)
  m_fInputMaxs[i] = fInputMaxs[i];
}

void CDMCObject::SetMoveSuppressionFactors (const CDWordArray&
fMoveSuppressionFactors)
{
 // set size
 m_fMoveSuppressionFactors.SetSize (fMoveSuppressionFactors.GetSize (), 10);
 // set values
 for (int i = 0; i < fMoveSuppressionFactors.GetSize (); i++)
  m_fMoveSuppressionFactors [i] = fMoveSuppressionFactors [i];

 // recalculate DMC params
 CalcDMCParameters ();
}

void CDMCObject::SetOutputWeights (    const CDWordArray& fOutputWeights,
                                       const CDWordArray& fAdaptWtModErr,
                                       const CDWordArray& fAdaptWtCurve,
                                       const CDWordArray& fAdaptWtMeasErr)
{
 // set size
 m_fOutputWeights.SetSize (fOutputWeights.GetSize (), 10);
 m_fAdaptWtModErr.SetSize (fAdaptWtModErr.GetSize (), 10);
 m_fAdaptWtMeasErr.SetSize (fAdaptWtMeasErr.GetSize (), 10);
 m_fAdaptWtCurve.SetSize (fAdaptWtCurve.GetSize (), 100);
 // set values
 for (int i = 0; i < fOutputWeights.GetSize (); i++)
  m_fOutputWeights [i] = fOutputWeights [i];

 for (i = 0; i < fAdaptWtModErr.GetSize (); i++)
  m_fAdaptWtModErr [i] = fAdaptWtModErr [i];

 for (i = 0; i < fAdaptWtMeasErr.GetSize (); i++)
  m_fAdaptWtMeasErr [i] = fAdaptWtMeasErr [i];

 for (i = 0; i < fAdaptWtCurve.GetSize (); i++)
  m_fAdaptWtCurve [i] = fAdaptWtCurve [i];

 // recalculate DMC params
 CalcDMCParameters ();
}

void CDMCObject::SetTimeFactors (const CDWordArray& fTimeFactors)
{
 // set size
 m_fTimeFactors.SetSize (fTimeFactors.GetSize (), 10);
 m_fTimeFactorPositions.SetSize (fTimeFactors.GetSize (), 10);
 // set values
 for (int i = 0; i < fTimeFactors.GetSize (); i++)
  m_fTimeFactors [i] = fTimeFactors [i];

 // recalculate DMC params
 CalcDMCParameters ();
}


// crude but effective
void CDMCObject::SetOutputStreamNames (const CStringArray&
csOutputStreamNames)
{
 // compare to see if any changes have occurred
 if (m_csOutputStreamNames.GetSize () == csOutputStreamNames.GetSize ())
 {
  BOOL bIdentical = TRUE;
  for (int i = 0; i < m_csOutputStreamNames.GetSize (); i++)
   if (m_csOutputStreamNames[i] != csOutputStreamNames[i])
    bIdentical = FALSE;

  // if no changes, exit (our job is done)
  if (bIdentical)
   return;
 }

 // if number of output streams has changed, recalc parameters
 if (m_csOutputStreamNames.GetSize () != csOutputStreamNames.GetSize ())
 {
   m_bRestartad = TRUE;  // Force complete re-initialisation of Adaptive DMC - IG
  CalcDMCParameters ();
 }

 // disconnect from all outputs
 if (m_bEnabled)
  for (int i = 0; i < m_pOutputStreams.GetSize (); i++)
   ((CIOStream*)m_pOutputStreams[i])->DisConnect (m_csName);

 // resize output name and pointer arrays
 m_pOutputStreams.SetSize (csOutputStreamNames.GetSize (), 10);
 m_csOutputStreamNames.SetSize (csOutputStreamNames.GetSize (), 10);

 // copy names across
 for (int i = 0; i < csOutputStreamNames.GetSize (); i++)
  m_csOutputStreamNames[i] = csOutputStreamNames[i];

 // set pointers to output streams (using names array to get them)
 for (i = 0; i < m_csOutputStreamNames.GetSize (); i++)
 {
  for (int j = 0; j < m_pScadDoc->m_DocObjectArray.GetSize (); j++)
  {
   CIOStream* pIOStream = (CIOStream*)m_pScadDoc->m_DocObjectArray[j];
   if (pIOStream->GetName () == m_csOutputStreamNames[i])
    m_pOutputStreams[i] = pIOStream;
  }
 }

 // connect to new output streams
 if (m_bEnabled)
  for (i = 0; i < m_pOutputStreams.GetSize (); i++)
   ((CIOStream*)m_pOutputStreams[i])->Connect (m_csName);
}

void CDMCObject::SetMaxChanges (const CDWordArray& fMaxChanges)
{
 // set size
 m_fMaxChanges.SetSize (fMaxChanges.GetSize (), 10);
 // set values
 for (int i = 0; i < fMaxChanges.GetSize (); i++)
  m_fMaxChanges[i] = fMaxChanges[i];
}

void CDMCObject::SetOutputMins (const CDWordArray& fOutputMins)
{
 // set size
 m_fOutputMins.SetSize (fOutputMins.GetSize (), 10);
 // set values
 for (int i = 0; i < fOutputMins.GetSize (); i++)
  m_fOutputMins[i] = fOutputMins[i];
}

void CDMCObject::SetOutputMaxs (const CDWordArray& fOutputMaxs)
{
 // set size
 m_fOutputMaxs.SetSize (fOutputMaxs.GetSize (), 10);
 // set values
 for (int i = 0; i < fOutputMaxs.GetSize (); i++)
  m_fOutputMaxs[i] = fOutputMaxs[i];
}

CString CDMCObject::StreamGetVal (int iIndex)
{
 CString csTemp;
 csTemp.Format ("%8.4lf", d(m_fSetpoints[iIndex]));
 return csTemp;
}

void CDMCObject::CalcDMCParameters (void)
{
 // if LP thread is running, stop it and delete temp variables
 if (m_bLPThreadRunning)
  m_bIgnoreLPThreadResult = TRUE;

 // delete old values for matrices
 if (m_pDynMatrix) delete m_pDynMatrix;
 if (m_pDynMatrix_Set) delete m_pDynMatrix_Set;
 if (m_pOpenLoopMatrix) delete m_pOpenLoopMatrix;
 if (m_pOffsetMatrix) delete m_pOffsetMatrix;
 if (m_pOpenLoopMatrix_Set) delete m_pOpenLoopMatrix_Set;
 if (m_pOffsetMatrix_Set) delete m_pOffsetMatrix_Set;
 if (m_pWeightingMatrix) delete m_pWeightingMatrix;
 if (m_pMoveSupMatrix) delete m_pMoveSupMatrix;
 if (m_pLeastSqrInvMatrix) delete m_pLeastSqrInvMatrix;
 if (m_pLowerTrilDMatrix) delete m_pLowerTrilDMatrix;

 // note that inputs/outputs to the DMC controller are outputs/inputs,
 // resp. to the DMC controller's convolution model!
 iNumInputs = m_csInputStreamNames.GetSize (); // NB!!! inputs to DMCObject
 iNumOutputs = m_csOutputStreamNames.GetSize (); // NB!!! outputs from
DMCObject

 // create array of step response matrices
 CPtrArray StepRespMatArray;
 StepRespMatArray.SetSize (m_iSteadyStateHorizon);

 // create step response matrices
 for (int i = 0; i < m_iSteadyStateHorizon; i++)
  StepRespMatArray.SetAt (i, new CMatrix (iNumInputs, iNumOutputs));

 // open step response File
 if ( m_cStepResponseFile == _T(""))
     m_cStepResponseFile = _T("response.stp");
 m_pScadDoc->SetDMCStepResponseFile (m_cStepResponseFile);
 FILE* StepResponseFile = fopen (m_cStepResponseFile, "rt");

 // read step response data
 for (int iInput = 1; iInput <= iNumInputs; iInput++)
  for (int iTime = 0; iTime < m_iSteadyStateHorizon; iTime++)  // MM990429 read
order changed !
   for (int iOutput = 1; iOutput <= iNumOutputs; iOutput++)
   {
       double* pfTemp = &((*((CMatrix*)StepRespMatArray[iTime]))[iInput][iOutput]);
    fscanf (StepResponseFile, "%lf", pfTemp);
   }

 // close step response File
 fclose (StepResponseFile);

 // MM990422 find the size of m_pDynMatrix after stripping rows with ZERO
TimeFactor
 m_iOptimisationHorizon=0;
 for (i = 0; i < m_iSteadyStateHorizon; i++)
 {
   if (d(m_fTimeFactors[i]) > 2e-50)
    {
       m_iOptimisationHorizon++ ; // ie. non-zero
       m_fTimeFactorPositions[m_iOptimisationHorizon-1] = i+1;
    }
 }

 // create dynamic matrix
 m_pDynMatrix = new CMatrix (iNumInputs * m_iOptimisationHorizon,
                 iNumOutputs * m_iNumberControlMoves);
 m_pDynMatrix_Set = new CMatrix (iNumInputs * m_iOptimisationHorizon,
                 iNumOutputs * m_iNumberControlMoves);

 // initialise dynamic matrix values
 i=0;
 for (int iTime = 1; iTime <= m_iSteadyStateHorizon; iTime++)
   if (d(m_fTimeFactors[iTime-1]) > 2e-50)  // ie. not zero or at least one
   {
       i++ ;
       for (int k = 1; k <= m_iNumberControlMoves; k++)
       {
           // get step response number and limit it to m_iSteadyStateHorizon
           int iStepResp = iTime - k ;  // on 0 to "m_iSteadyStateHorizon - 1" scale
           iStepResp = min (iStepResp, m_iSteadyStateHorizon - 1);
           if (iStepResp >= 0)
           {
               // get step response matrix
               CMatrix* StepRespMat = (CMatrix*)StepRespMatArray[iStepResp];
```

```
            for (int iInput = 1; iInput <= iNumInputs; iInput++)
            {
                int ii= (i-1)*iNumInputs + iInput;
                for (int jOutput = 1; jOutput <= iNumOutputs; jOutput++)
                {
                    int jj= (k-1)*iNumOutputs + jOutput;
                    // set value for ii, jj
                    (*m_pDynMatrix)[ii][jj] = (*StepRespMat)[iInput][jOutput];
                    (*m_pDynMatrix_Set)[ii][jj] = (*StepRespMat)[iInput][jOutput];
//Also Store it
                }
            }
        }
        else
            for (int iInput = 1; iInput <= iNumInputs; iInput++)
            {
                int ii= (i-1)*iNumInputs + iInput;
                for (int j = 1; j <= iNumOutputs; j++)
                {
                    int jj= (k-1)*iNumOutputs + j;
                    (*m_pDynMatrix)[ii][jj] = 0.0;
                    (*m_pDynMatrix_Set)[ii][jj] = 0.0;
                }
            }
    }
}
```

```
m_PlotWnd.SetAMatrix (m_pDynMatrix, m_iOptimisationHorizon, iNumInputs);

/*
// testing code ****
FILE* File = fopen ("testA.dat", "wt");
for (i = 1; i <= iNumInputs * m_iOptimisationHorizon; i++)
{
  for (int j = 1; j <= iNumOutputs * m_iNumberControlMoves; j++)
  {
    if (j != iNumOutputs * m_iNumberControlMoves)
      fprintf (File, "%8.3lf  ", (*m_pDynMatrix)[i][j]);
    else
      fprintf (File, "%8.3lf\n", (*m_pDynMatrix)[i][j]);
  }
}
fprintf (File, "\n");
fprintf (File, "\n");
fclose (File);
// end of testing code ****
*/

// create open loop matrix
m_pOpenLoopMatrix = new CMatrix (iNumInputs * m_iOptimisationHorizon,
                    iNumOutputs * m_iSteadyStateHorizon);
m_pOpenLoopMatrix_Set = new CMatrix (iNumInputs * m_iOptimisationHorizon,
                    iNumOutputs * m_iSteadyStateHorizon);

// initialise open loop matrix values
CMatrix* StepRespMatLastINTEG =
(CMatrix*)StepRespMatArray[m_iSteadyStateHorizon - 1];
CMatrix* StepRespMatSecLastINTEG =
(CMatrix*)StepRespMatArray[m_iSteadyStateHorizon - 2];
CMatrix StepRespMatDeltaINTEG = (*StepRespMatLastINTEG) -
(*StepRespMatSecLastINTEG);
i=0;
for (iTime = 1; iTime <= m_iSteadyStateHorizon; iTime++)
  if (d(m_fTimeFactors[iTime-1]) > 2e-50)  // ie. non-zero
  {
    i++ ;
    for (int k = 1; k<=m_iSteadyStateHorizon; k++)
    {
      int iStepResp = iTime + (m_iSteadyStateHorizon - k ); // starts at 0
      int iStepRespINTEG = iStepResp;
      iStepResp = min (iStepResp, m_iSteadyStateHorizon - 1);
      // get step response matrix
      CMatrix* StepRespMat = (CMatrix*)StepRespMatArray[iStepResp];
      for (int iInput = 1; iInput <= iNumInputs; iInput++)
      {
        for (int jOutput = 1; jOutput <= iNumOutputs; jOutput++)
        {
          // set value for ii, jj
          int ii=(i-1)*iNumInputs + iInput;
          int jj=(k-1)*iNumOutputs + jOutput;
          if (iStepRespINTEG == iStepResp)
          {
            (*m_pOpenLoopMatrix)[ii][jj] = (*StepRespMat)[iInput][jOutput];
            (*m_pOpenLoopMatrix_Set)[ii][jj] =
(*StepRespMat)[iInput][jOutput];
          }
          else
          {
            // unequal previous two points indicates integration (extrapolate
B)
            (*m_pOpenLoopMatrix)[ii][jj] =
(*StepRespMatLastINTEG)[iInput][jOutput] +
              (double)m_bExtSlope_Int * (double)(iStepRespINTEG -
iStepResp) * StepRespMatDeltaINTEG[iInput][jOutput];
            (*m_pOpenLoopMatrix_Set)[ii][jj] =
(*StepRespMatLastINTEG)[iInput][jOutput] +
              (double)m_bExtSlope_Int * (double)(iStepRespINTEG -
iStepResp) * StepRespMatDeltaINTEG[iInput][jOutput];
          }
        }
      }
    }
  }

// create offset matrix
m_pOffsetMatrix = new CMatrix (iNumInputs * m_iOptimisationHorizon,
                    iNumOutputs * m_iSteadyStateHorizon);
m_pOffsetMatrix_Set = new CMatrix (iNumInputs * m_iOptimisationHorizon,
                    iNumOutputs * m_iSteadyStateHorizon);

// initialise offset matrix values
i=0;
for (iTime = 1; iTime <= m_iSteadyStateHorizon; iTime++)
```

```
  if (d(m_fTimeFactors[iTime-1]) > 2e-50)  // ie. non-zero
  {
    i++ ;
    for (int k = 1; k <= m_iSteadyStateHorizon; k++)
    {
      int iStepResp = (m_iSteadyStateHorizon - k ); // starts at 0
      // get step response matrix
      CMatrix* StepRespMat = (CMatrix*)StepRespMatArray[iStepResp];
      for (int iInput = 1; iInput <= iNumInputs; iInput++)
      {
        for (int jOutput = 1; jOutput <= iNumOutputs; jOutput++)
        {
          // set value for ii, jj
          int ii=(i-1)*iNumInputs + iInput;
          int jj=(k-1)*iNumOutputs + jOutput;
          (*m_pOffsetMatrix)[ii][jj] = (*StepRespMat)[iInput][jOutput];

          (*m_pOffsetMatrix_Set)[ii][jj] = (*StepRespMat)[iInput][jOutput];

        }
      }
    }
  }

// create optimisation trajectory weighting matrix
m_pWeightingMatrix = new CMatrix (m_iOptimisationHorizon * iNumInputs,
                    m_iOptimisationHorizon * iNumInputs);
// initialise weighting matrix
for (i = 1; i <= m_pWeightingMatrix->Height (); i++)
  for (int j = 1; j <= m_pWeightingMatrix->Width (); j++)
    (*m_pWeightingMatrix)[i][j] = 0.0;
// m_pWeightingMatrix->Identity ();  old initialisation

// set weighting matrix
i=0;
for (iTime = 1; iTime <= m_iSteadyStateHorizon; iTime++)
{
  if (d(m_fTimeFactors[iTime-1]) > 2e-50)  // ie. non-zero
  {
    i++ ;
    for (int iInput = 1; iInput <= iNumInputs; iInput++)
    {
      int index = (i-1) * iNumInputs + iInput;
      (*m_pWeightingMatrix)[index][index] =
        d(m_fOutputWeights[iInput-1]) * d(m_fTimeFactors[iTime-1]);
    }
  }
}

// create move suppression matrix
m_pMoveSupMatrix = new CMatrix (m_iNumberControlMoves * iNumOutputs,
                    m_iNumberControlMoves * iNumOutputs);
// initialise move suppression matrix **** better place to put this?
for (i = 1; i <= m_pMoveSupMatrix->Height (); i++)
  for (int j = 1; j <= m_pMoveSupMatrix->Width (); j++)
    (*m_pMoveSupMatrix)[i][j] = 0.0;

for (i = 1; i <= iNumOutputs; i++)
  for (int iTime = 0; iTime < m_iNumberControlMoves; iTime++)
  {
    int index = i + iTime * iNumOutputs;
    (*m_pMoveSupMatrix)[index][index] = d(m_fMoveSuppressionFactors[i-1]);
  }

// create least squares inverse matrix
m_pLeastSqrInvMatrix = new CMatrix (m_iNumberControlMoves * iNumOutputs,
                    m_iOptimisationHorizon * iNumInputs);

// set least squares inverse matrix
*m_pLeastSqrInvMatrix = (m_pDynMatrix->Transposed () * *m_pWeightingMatrix *
*m_pDynMatrix +
                *m_pMoveSupMatrix).Inverse () * m_pDynMatrix->Transposed () *
                *m_pWeightingMatrix;

m_pLowerTrilDMatrix = new CMatrix (iNumOutputs * m_iNumberControlMoves,
                    iNumOutputs * m_iNumberControlMoves);
for (i = 1; i <= m_iNumberControlMoves * iNumOutputs; i++)
{
  for (int j = 1; j <= m_iNumberControlMoves * iNumOutputs; j++)
  {
    if (i > j)
      (*m_pLowerTrilDMatrix)[i][j] = 0.0;  // zero in upper triangle
    else if ((i-j) % iNumOutputs == 0)
      (*m_pLowerTrilDMatrix)[i][j] = 1.0;
    else
      (*m_pLowerTrilDMatrix)[i][j] = 0.0;
  }
}

// Initialisations for Adaptive DMC - IG ----------

m_pScadDoc->SetDMCIntervalSec (m_fTimeInterval);

if (!m_bRestarted)
{
  // Only Update the Diagonals of Qad, Aad, Bad and Rad ...
  // Transfer Qad matrix weightings to the correct positions
  for (i=1; i<=iNumInputs; i++)
  {
    (*m_pQad)[i][i] = d(m_fAdaptWtModEn[i-1]);
    (*m_pQadw)[i][i] = d(m_fAdaptWtModEn[i-1]);
    for (int j=1; j<=iNumOutputs; j++)
    {
      for (int kkk=0; kkk<=m_iNumberControlMoves-1; kkk++)  // direct & delayed curves
      {
        int kk = iNumInputs + kkk*iNumInputs*iNumOutputs + (i-1)*iNumOutputs
+j;
        (*m_pQadw)[kk][kk] = m_fQad * d(m_fAdaptWtCurve[(i-
1)*iNumOutputs + j-1]); // NB Product
        (*m_pAadw)[kk][kk] = m_fAad;
        (*m_pBadw)[kk][kk] = 1.0-m_fAad;
      }
      for (int iT=1; iT<=m_iSteadyStateHorizon; iT++)
      {
```

```
                int k = iNumInputs * (i-1)*m_iSteadyStateHorizon*iNumOutputs
                    + (m_iSteadyStateHorizon-iT)*iNumOutputs+j;
                (*m_pQad)[k][k] = m_fQad * d(m_fAdaptWtCurve[i - 1)*iNumOutputs + j
- 1]); // NB Product
                (*m_pAad)[k][k] = m_fABad;
                (*m_pBad)[k][k] = 1.0-m_fABad;
            }
        }
    }
    for (i=1; i<=iNumInputs; i++)
    {
        (*m_pRad)[i][i] = d(m_fAdaptWtMeasErr[i-1]); // Meas Err Wt
    }

}
else
{
  // Only do a complete Restart if it is not a continuation

    // store data for plotting Adapted curves in ScadDoc
    m_pScadDoc->SetDMCNumOut (iNumOutputs);
    m_pScadDoc->SetDMCNumIn (iNumInputs);
    m_pScadDoc->SetDMCNumIn (iNumInputs);
    m_pScadDoc->SetDMCSteadyStateHorizon (m_iSteadyStateHorizon);

  // re-initialise past output changes vectors
  if (m_pPastOutputChanges) delete m_pPastOutputChanges;
  m_pPastOutputChanges = new CVector (iNumOutputs * m_iSteadyStateHorizon);
  if (m_pDoublePastOutputChangesad) delete m_pDoublePastOutputChangesad;
   m_pDoublePastOutputChangesad = new CVector (iNumOutputs *
m_iSteadyStateHorizon);
  for (i = 1; i <= iNumOutputs * m_iSteadyStateHorizon; i++)
   {
    (*m_pPastOutputChanges)[i] = 0.0;
    (*m_pDoublePastOutputChangesad)[i] = 0.0;  // a further block backwards in time
   }

   // delete old matrix structures
   if (m_pB0ad) delete m_pB0ad ;
   if (m_pBOLad) delete m_pBOLad;
   if (m_pBOL_INTEG) delete m_pBOL_INTEG;
   if (m_pB0ad_orig) delete m_pB0ad_orig ;
   if (m_pBOLad_orig) delete m_pBOLad_orig;
   if (m_pBOL_INTEG_orig) delete m_pBOL_INTEG_orig;
   if (m_pead) delete m_pead ;
   if (m_pPead) delete m_pPead ;
   if (m_pLead) delete m_pLead ;
   if (m_pSPead) delete m_pSPead ;
   if (m_pDx0ad) delete m_pDx0ad ;
   if (m_ppad) delete m_ppad ;
   if (m_ppad_orig) delete m_ppad_orig ;
   if (m_pGad) delete m_pGad ;
   if (m_pGad1) delete m_pGad1 ;
   if (m_pGad2) delete m_pGad2 ;
   if (m_pGad3) delete m_pGad3 ;
   if (m_pQad) delete m_pQad ;
   if (m_pRad) delete m_pRad ;
   if (m_pKad) delete m_pKad ;
   if (m_pAad) delete m_pAad ;
   if (m_pBad) delete m_pBad ;
   if (m_pxStackad) delete m_pxStackad ;
   if (m_plad) delete m_plad ;

   if (m_ppadw) delete m_ppadw ;
   if (m_ppadw_orig) delete m_ppadw_orig ;
   if (m_pGadw) delete m_pGadw ;
   if (m_pQadw) delete m_pQadw ;
   if (m_pKadw) delete m_pKadw ;
   if (m_pAadw) delete m_pAadw ;
   if (m_pBadw) delete m_pBadw ;
   if (m_pladw) delete m_pladw ;
   if (m_pFadw) delete m_pFadw ;

   if (m_pPreviousOutput) delete m_pPreviousOutput ;
   if (m_pPreviousOutputChange) delete m_pPreviousOutputChange ;
   if (m_pPreviousInput) delete m_pPreviousInput ;
   if (m_pTest) delete m_pTest ;
   if (AccumulatedMoveINTEGCORR) delete AccumulatedMoveINTEGCORR ;
   if (AccumulatedMoveINTEGCORRad) delete AccumulatedMoveINTEGCORRad ;

    m_pB0ad = new CMatrix (iNumInputs, iNumOutputs * m_iSteadyStateHorizon);
   m_pBOLad = new CMatrix (iNumInputs, iNumOutputs * m_iSteadyStateHorizon)
   m_pBOL_INTEG = new CMatrix (iNumInputs, iNumOutputs);
    m_pB0ad_orig = new CMatrix (iNumInputs, iNumOutputs *
m_iSteadyStateHorizon);
   m_pBOLad_orig = new CMatrix (iNumInputs, iNumOutputs *
m_iSteadyStateHorizon);
   m_pBOL_INTEG_orig = new CMatrix (iNumInputs, iNumOutputs);
    m_pead = new CVector (iNumInputs);
    m_pPead = new CVector (iNumInputs);
    m_pLead = new CVector (iNumInputs);
    m_pSPead = new CVector (iNumInputs);
    m_pDx0ad = new CVector (iNumInputs);
    m_ppad = new CVector (iNumInputs *
iNumOutputs*iNumInputs*m_iSteadyStateHorizon);
    m_ppad_orig = new CVector (iNumInputs *
iNumOutputs*iNumInputs*m_iSteadyStateHorizon);
    m_pGad = new CMatrix (iNumInputs,
iNumInputs+iNumOutputs*iNumInputs*m_iSteadyStateHorizon);
    m_pGad1 = new CMatrix (iNumInputs,
iNumInputs+iNumOutputs*iNumInputs*m_iSteadyStateHorizon);
    m_pGad2 = new CMatrix (iNumInputs,
iNumInputs+iNumOutputs*iNumInputs*m_iSteadyStateHorizon);
    m_pGad3 = new CMatrix (iNumInputs,
iNumInputs+iNumOutputs*iNumInputs*m_iSteadyStateHorizon);
    m_pRad = new CMatrix (iNumInputs, iNumInputs);
    m_pKad = new CMatrix
(iNumInputs+iNumOutputs*iNumInputs*m_iSteadyStateHorizon, iNumInputs);

    m_pAad = new CMatrix
(iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon,
                iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon);
    m_pBad = new CMatrix
(iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon,
                iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon);
    m_plad = new CMatrix
(iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon,
                iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon);
    m_pQad = new CMatrix
(iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon,
                iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon);
    m_pMad = new CMatrix
(iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon,
                iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon);

    m_ppadw = new CVector (iNumInputs + 2*iNumOutputs*iNumInputs);
    m_ppadw_orig = new CVector (iNumInputs + 2*iNumOutputs*iNumInputs);
    m_pGadw = new CMatrix (iNumInputs, iNumInputs+2*iNumOutputs*iNumInputs);
    m_pKadw = new CMatrix (iNumInputs+2*iNumOutputs*iNumInputs, iNumInputs);

    m_pAadw = new CMatrix (iNumInputs+2*iNumInputs*iNumOutputs,
                iNumInputs+2*iNumInputs*iNumOutputs);
    m_pBadw = new CMatrix (iNumInputs+2*iNumInputs*iNumOutputs,
                iNumInputs+2*iNumInputs*iNumOutputs);
    m_pladw = new CMatrix (iNumInputs+2*iNumInputs*iNumOutputs,
                iNumInputs+2*iNumInputs*iNumOutputs);
    m_pQadw = new CMatrix (iNumInputs+2*iNumInputs*iNumOutputs,
                iNumInputs+2*iNumInputs*iNumOutputs);
    m_pMadw = new CMatrix (iNumInputs+2*iNumInputs*iNumOutputs,
                iNumInputs+2*iNumInputs*iNumOutputs);
    m_pFadw = new CMatrix (iNumInputs +
iNumOutputs*iNumInputs*m_iSteadyStateHorizon,
                iNumInputs+2*iNumInputs*iNumOutputs);

    m_pxStackad = new CMatrix (iNumInputs, m_iSteadyStateHorizon+0),
   m_pPreviousOutput = new CVector (iNumOutputs);
   m_pPreviousOutputChange = new CVector (iNumOutputs);
    m_pPreviousInput = new CVector (iNumInputs);
    m_pTest = new CMatrix (iNuminputs, iNuminputs);
    AccumulatedMoveINTEGCORR = new CVector (iNumOutputs);
    AccumulatedMoveINTEGCORRad = new CVector (iNumOutputs);

    // Set up initial values for "previous input" and the delay stack for Outputs
   // If necessary, temporarily "Enabled" the DMCObject to get a Name
    BOOL m_bTempEnabled;
    if (!m_bEnabled)
    {
        m_bTempEnabled = TRUE;
        SetEnable (TRUE);
    }
    else
    {
        m_bTempEnabled = FALSE;
    }
    // outputs
   for (i = 1; i <= iNumOutputs; i++)
    {
        (*m_pPreviousOutput)[i] = ((CIOStream*)m_pOutputStreams[i-1])->Read
(m_csName);
        (*m_pPreviousOutputChange)[i] = 0.0;
    }
    // inputs
   for (i = 1; i <= iNuminputs; i++)
    {
        // set previous state vector to current state vector
        (*m_pPreviousInput)[i] = ((CIOStream*)m_pInputStreams[i-1])->Read
(m_csName);
        double x_Stackadi = ((CIOStream*)m_pInputStreams[i-1])->Read (m_csName);
        // fill the whole stack with present state vector
        for (int iTime = 1; iTime <= m_iSteadyStateHorizon+0; iTime++)
        {
            (*m_pxStackad)[i][iTime] = x_Stackadi;
        }
    }
    if (m_bTempEnabled) SetEnable(FALSE);

    // initial values for openloop INTEGRAL ERROR CORRECTIONS (if last two
positions on step-resp unequal)
    for (i = 1; i <= iNumOutputs; i++)
    {
        (*AccumulatedMoveINTEGCORR)[i] = 0.0;
        (*AccumulatedMoveINTEGCORRad)[i] = 0.0;
    }


    // Set up initial estimate of B0 as the values set in the Dynamic Matrix
    for (int iT = 1; iT <= m_iSteadyStateHorizon; iT++)
    {
        CMatrix* StepRespMat = (CMatrix*)StepRespMatArray[iT-1]; // starts at 0
        for (int i = 1; i <= iNuminputs; i++)
        {
            for (int j = 1; j <= iNumOutputs; j++)
            {
                int ij = (m_iSteadyStateHorizon-iT)*iNumOutputs+j;
                double B0adij = (*StepRespMat)[i][j];
                (*m_pB0ad)[i][ij] =  B0adij;
                (*m_pB0ad_orig)[i][ij] =  B0adij;
                m_pScadDoc->SetDMCB0initad( i, ij, B0adij);
                m_pScadDoc->SetDMCB0ad( i, ij, B0adij);
            }
        }
    }

    // Set up initial estimate of BOLad by extrapolating the final two values in B0ad
    for (iT = 1; iT <= m_iSteadyStateHorizon; iT++)
    {
        for (int i = 1; i <= iNuminputs; i++)
        {
            for (int j = 1; j <= iNumOutputs; j++)
            {
                int iTT;
                if ((m_bSuppressINTEGad)||(m_bExtSlope_Int))  // Suppress adaptation
for Integrating System ?
                {
                    iTT=1;  // Uniform matrix
                }
                else
                {
                    iTT=iT; // Ramping matrix
                }
                int ij=(m_iSteadyStateHorizon-iT)*iNumOutputs+j;
```

```
                (*m_pBOLad)[i][j] = (double)(iTT+1) * (*m_pB0ad)[i][j] -
                    (double)iTT * (*m_pB0ad)[i][j+iNumOutputs];
                (*m_pBOLad_orig)[i][j] = (*m_pBOLad)[i][j];
            } //Extrapolating in case of integration
        }
    }

    for (int i = 1; i <= iNumInputs; i++)
    {
        for (int j = 1; j <= iNumOutputs; j++)
        {
            (*m_pBOL_INTEG)[i][j] = (double)m_bAccMoves_Int *
                (double)m_iSteadyStateHorizon * ((*m_pB0ad)[i][j]-
(*m_pB0ad)[i][j+iNumOutputs]);
            (*m_pBOL_INTEG_orig)[i][j] = (*m_pBOL_INTEG)[i][j];
        } //A differential term for the tail slope for integrating systems control
    }

    // Set up initial estimate of IMC error
    for (i = 1; i <= iNumInputs; i++)
    {
        (*m_pead)[i] = 0.0;
        (*m_pPead)[i] = 0.0;
        (*m_pLead)[i] = 0.0;
        (*m_pSPead)[i] = 0.0;
        m_pScadDoc->SetDMCead( i, 0.0);
    }

    // Set up initial estimate of parameter vector xbar and original
    for (i=1; i<=iNumInputs; i++)
    {
        (*m_ppad)[i] = (*m_pead)[i];
        (*m_ppad_orig)[i] = (*m_pead)[i];
        (*m_ppadw)[i] = (*m_pead)[i];
        (*m_ppadw_orig)[i] = (*m_pead)[i];
        for (int j=1; j<=iNumOutputs; j++)
        {
            (*m_ppadw)[iNumInputs+(i-1)*iNumOutputs+j] = 1.0;   // on-time weight
            (*m_ppadw_orig)[iNumInputs+(i-1)*iNumOutputs+j] = 1.0;
            (*m_ppadw)[iNumInputs*(1+iNumOutputs)+(i-1)*iNumOutputs+j] = 0.0; //
delayed weight
            (*m_ppadw_orig)[iNumInputs*(1+iNumOutputs)+(i-1)*iNumOutputs+j] = 0.0;
        }
        for (int j=1; j<=m_iSteadyStateHorizon*iNumOutputs; j++)
        {
            (*m_ppad)[iNumInputs+(i-1)*iNumOutputs*m_iSteadyStateHorizon+j] =
(*m_pB0ad)[i][j];
            (*m_ppad_orig)[iNumInputs+(i-1)*iNumOutputs*m_iSteadyStateHorizon+j] =
(*m_pB0ad)[i][j];
        }
    }

    // Transformation Matrix F for regularization
    for (i=1; i<=iNumInputs + iNumOutputs*iNumInputs*m_iSteadyStateHorizon; i++)
    {
        for (int j=1; j<=iNumInputs+2*iNumInputs*iNumOutputs; j++)
        {
            (*m_pFadw)[i][j] = 0.0;   // first zero the whole thing
        }
    }
    for (i=1; i<=iNumInputs; i++)
    {
        (*m_pFadw)[i][i] = 1.0;   // the "e" error part
    }
    // direct curve scaling
    for (i=1; i<=iNumInputs; i++)
    {
        for (int j=1; j<=iNumOutputs; j++)
        {
            for (int k=1; k<=m_iSteadyStateHorizon; k++)
            {  // pluck out the ones required
                (*m_pFadw)[iNumInputs+(i-1)*iNumOutputs*m_iSteadyStateHorizon+
                    (m_iSteadyStateHorizon-k)*iNumOutputs+j][iNumInputs+(i-
1)*iNumOutputs+j] =
                    (*m_pB0ad)[i][(m_iSteadyStateHorizon-k)*iNumOutputs+j];
            }
        }
    }
    // delayed curve scaling for time-shift
    for (i=1; i<=iNumInputs; i++)
    {
        for (int j=1; j<=iNumOutputs; j++)
        {
            for (int k=1; k<=m_iSteadyStateHorizon; k++)
            {  // pluck out the ones required from time-shifted curves
                int kk = k-1;
                if (kk > 0)
                {
                    (*m_pFadw)[iNumInputs+(i-
1)*iNumOutputs*m_iSteadyStateHorizon+
                        (m_iSteadyStateHorizon-k)*iNumOutputs+j]
                        [iNumInputs*(1+iNumOutputs)+(i-1)*iNumOutputs+j]
                        = (*m_pB0ad)[i][(m_iSteadyStateHorizon-kk)*iNumOutputs+j];
                }
                else
                {
                    (*m_pFadw)[iNumInputs+(i-
1)*iNumOutputs*m_iSteadyStateHorizon+
                        (m_iSteadyStateHorizon-k)*iNumOutputs+j]
                        [iNumInputs*(1+iNumOutputs)+(i-1)*iNumOutputs+j] = 0.0;
                }
            }
        }
    }


    // Pad, Bad, Mad and Qad
    for (int i=1; i<=iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon;
i++)
    {
        for (int k=1;
k<=iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon; k++)
        {
            if(i==k)
            {
                (*m_pAad)[i][k] = 0.0;
                (*m_pBad)[i][k] = 0.0;
                (*m_pMad)[i][k] = 0.0;
                (*m_plad)[i][k] = 0.0;
                (*m_pQad)[i][k] = 0.0;
            }
            else
            {
                (*m_pMad)[i][k] = 0.0001;
                (*m_plad)[i][k] = 1.0;
                if (i>iNumInputs)
                {
                    (*m_pAad)[i][k] = m_fABad;
                    (*m_pBad)[i][k] = 1.0-m_fABad;
                    // See Qad diagonal set below
                }
                else
                {
                    (*m_pAad)[i][k] = 0.0;   // to force the eAll to zero
                    (*m_pBad)[i][k] = 0.0;
                    // See Qad diagonal set below
                }
            }
        }
    }

    // Transfer Qad matrix weightings to the correct positions
    for (i=1; i<=iNumInputs; i++)
    {
        (*m_pQad)[i][i] = d[m_fAdaptWtModErr[i-1]];
        for (int iT=1; iT<=m_iSteadyStateHorizon; iT++)
        {
            for (int j=1; j<=iNumOutputs; j++)
            {
                int k = iNumInputs + (i-1)*m_iSteadyStateHorizon*iNumOutputs
                    + (m_iSteadyStateHorizon-iT)*iNumOutputs+j;
                (*m_pQad)[k][k] =
                    m_fQad * d[m_fAdaptWtCurve[(i - 1)*iNumOutputs + j - 1]]; // NB
Product
            }
        }
    }

    // Aadw, Badw, Iadw, Madw and Qadw
    for (i=1; i<=iNumInputs+2*iNumInputs*iNumOutputs; i++)
    {
        for (int k=1; k<=iNumInputs+2*iNumInputs*iNumOutputs; k++)
        {
            if (i != k)
            {
                (*m_pAadw)[i][k] = 0.0;
                (*m_pBadw)[i][k] = 0.0;
                (*m_pMadw)[i][k] = 0.0;
                (*m_pIadw)[i][k] = 0.0;
                (*m_pQadw)[i][k] = 0.0;
            }
            else
            {
                (*m_pMadw)[i][k] = 0.0001;
                (*m_pIadw)[i][k] = 1.0;
                if (i>iNumInputs)
                {
                    (*m_pAadw)[i][k] = m_fABad;
                    (*m_pBadw)[i][k] = 1.0-m_fABad;
                    // See Qad diagonal set below
                }
                else
                {
                    (*m_pAadw)[i][k] = 0.0;   // to force the eAll to zero
                    (*m_pBadw)[i][k] = 0.0;
                    // See below Qad diagonal set below
                }
            }
        }
    }

    // Transfer Qadw matrix weightings to the correct positions
    for (i=1; i<=iNumInputs; i++)
    {
        (*m_pQadw)[i][i] = d[m_fAdaptWtModErr[i-1]];
        for (int j=1; j<=NumOutputs; j++)
        {
            int k = iNumInputs + (i-1)*iNumOutputs + j;
            (*m_pQadw)[k][k] =
                m_fQad * d[m_fAdaptWtCurve[(i - 1)*iNumOutputs + j - 1]]; // NB
Product
            int kk = iNumInputs*(1+iNumOutputs) + (i-1)*iNumOutputs + j;
            (*m_pQadw)[kk][kk] =
                m_fQad * d[m_fAdaptWtCurve[(i - 1)*iNumOutputs + j - 1]];// NB
Product
        }
    }


    // Rad
    for (i = 1; i <= iNumInputs; i++)
    {
        for (int k = 1; k <= iNumInputs; k++)
        {
            if (i != k)
            {
                (*m_pRad)[i][k] = 0.0;
            }
            else
            {
                (*m_pRad)[i][k] = d[m_fAdaptWtModErr[i-1]]; // Meas Err Wt
            }
        }
    }
    // MUST TOGGLE Restart OFF now that re-initialisation has been done
    m_bRestart = FALSE;

}
// ------------------------------------------------------------ IG

// delete step responses
```

```
for (i = 0; i < m_iSteadyStateHorizon; i++)
  if ((CMatrix*)StepRespMatArray[i])delete (CMatrix*)StepRespMatArray[i];
}

void CDMCObject::TimerUpdate (DWORD dwTime)
{
    iNumInputs = m_csInputStreamNames.GetSize (); // Inputs to DMCObject (PV's)
    iNumOutputs = m_csOutputStreamNames.GetSize (); // outputs from DMCObject
(MV's)

    //###SD000815
    m_dwTimeTillUpdate = DWORD(m_fTimeInterval*1000) - (dwTime -
m_dwPrevTime);
    //###

    // Use the adaptation timer to trigger normal past-output-update for
    // both adaptation & DMC
    double DeltaTimeAdapt = (double)((int)dwTime - (int)m_dwPrevTimeAdapt) /
1000.0;
    if (DeltaTimeAdapt >= m_fTimeInterval)
    {
        m_dwPrevTimeAdapt += (int)m_fTimeInterval * 1000;

        // accumulate the last of the double-delayed block into the adaptation
INTEGRAL ACCUMULATOR
        if (!m_bSuppressINTEGad)
        {
            for (int i = 1; i <= iNumOutputs; i++)(*AccumulatedMoveINTEGCORRad)[i] =
                (double)m_bAccMoves_Int * ((*m_pDoublePastOutputChangesad)[i] +
(*AccumulatedMoveINTEGCORRad)[i]);
        }

        // cascade m_pDoublePastOutputChangesad
        for (int i = 1; i <= iNumOutputs; i++)
            for (int iTime = 0; iTime < (m_iSteadyStateHorizon - 1); iTime++)
                (*m_pDoublePastOutputChangesad)[i + iTime * iNumOutputs] =
                        (*m_pDoublePastOutputChangesad)[i + (iTime+1) *
iNumOutputs];

        // pop the last of the next block into the end before it is overwritten below
        for (i = 1; i <= iNumOutputs; i++)
            (*m_pDoublePastOutputChangesad)[i + (m_iSteadyStateHorizon - 1)
                * iNumOutputs] = (*m_pPastOutputChanges)[i];

        // also accumulate the last of the next block into the INTEGRAL
ACCUMULATOR for DMC control
        for (i = 1; i <= iNumOutputs; i++)
        {
            (*AccumulatedMoveINTEGCORR)[i] =
            (double)m_bAccMoves_Int * ((*m_pPastOutputChanges)[i] +
(*AccumulatedMoveINTEGCORR)[i]);
        }

        // cascade m_pPastOutputChanges
        for (i = 1; i <= iNumOutputs; i++)
            for (int iTime = 0; iTime < (m_iSteadyStateHorizon - 1); iTime++)
                (*m_pPastOutputChanges)[i + iTime * iNumOutputs] =
                        (*m_pPastOutputChanges)[i + (iTime+1) * iNumOutputs];

        // cascade past input state vectors through stack (could just move pointer !)
        for (int iTime = 1; iTime < (m_iSteadyStateHorizon+0); iTime++)
        {
            for (i = 1; i <= iNumInputs; i++)
            {
                (*m_pxStackad)[i][iTime] = (*m_pxStackad)[i][iTime+1];
            }
        }

        // If necessary, temporarily "Enabled" the DMCObject to get a Name
        BOOL m_bTempEnabled;
        if (!m_bEnabled)
        {
            m_bTempEnabled = TRUE;
            SetEnable (TRUE);
        }
        else
        {
            m_bTempEnabled = FALSE;
        }

        // present Outputs
        for (i = 1; i <= iNumOutputs; i++)
        {
            (*m_pPastOutputChanges)[i + (m_iSteadyStateHorizon - 1) * iNumOutputs]
=
                    ((CIOStream*)m_pOutputStreams[i-1])->Read (m_csName) -
(*m_pPreviousOutput)[i];
            //(*m_pPreviousOutputChange)[i];  // DeltaM[i] one time step old
            //(*m_pPreviousOutputChange)[i] =
            //        ((CIOStream*)m_pOutputStreams[i-1])->Read (m_csName) -
(*m_pPreviousOutput)[i]; // DeltaM[i] new
            (*m_pPreviousOutput)[i] = ((CIOStream*)m_pOutputStreams[i-1])->Read
(m_csName);
        }

        // store previous and find current state vector
        for (i = 1; i <= iNumInputs; i++)
        {
            (*m_pPreviousInput)[i] =  (*m_pxStackad)[i][m_iSteadyStateHorizon+0];
            (*m_pxStackad)[i][m_iSteadyStateHorizon+0] =
((CIOStream*)m_pInputStreams[i-1])->Read(m_csName);
        }


        // Update Observation Matrix Gad.....

        // Gad for BOad * PastOutputChanges
        for (i=1; i<= iNumInputs; i++)
        {
            for (int j=1;
j<=iNumInputs*iNumOutputs*iNumInputs*m_iSteadyStateHorizon; j++)
            {
                (*m_pGad)[i][j] = 0.0;
            }
            (*m_pGad)[i][i] = 1.0;
        }

        for (i=1; i<= iNumInputs; i++)
        {
            for (int j=1; j<=iNumOutputs*m_iSteadyStateHorizon; j++)
            {
                (*m_pGad)[i][iNumInputs+(i-1)*iNumOutputs*m_iSteadyStateHorizon+j]
=
                                                    (*m_pPastOutputChanges)[j];
            }
        }

        // Gad1 for BOad * DoublePastOutputChanges
        for (i=1; i<= iNumInputs; i++)
        {
            for (int ijk=1;
ijk<=iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon; ijk++)
                (*m_pGad1)[i][ijk] = 0;
            for (int j=1; j<=iNumOutputs*m_iSteadyStateHorizon; j++)
            {
                (*m_pGad1)[i][iNumInputs+(i-
1)*iNumOutputs*m_iSteadyStateHorizon+j] =

(*m_pDoublePastOutputChangesad)[j];
            }
        }

        // Gad2 for BOLad * DoublePastOutputChanges
        for (i=1; i<= iNumInputs; i++)
        {
            for (int ijk=1;
ijk<=iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon; ijk++)
                (*m_pGad2)[i][ijk] = 0;
            for (int j=1; j<=iNumOutputs; j++)
            {
                for (int k=1; k<=m_iSteadyStateHorizon; k++)
                {
                    (*m_pGad2)[i][iNumInputs+(i-
1)*iNumOutputs*m_iSteadyStateHorizon+j] =
                        (*m_pGad2)[i][iNumInputs+(i-
1)*iNumOutputs*m_iSteadyStateHorizon+j]
                        + (double) (m_iSteadyStateHorizon+2-
k)*(*m_pDoublePastOutputChangesad)[(k-1)*iNumOutputs+j];

                    (*m_pGad2)[i][iNumInputs+(i-
1)*iNumOutputs*m_iSteadyStateHorizon+j+iNumOutputs] =
                        (*m_pGad2)[i][iNumInputs+(i-
1)*iNumOutputs*m_iSteadyStateHorizon+j+iNumOutputs]
                        - (double) (m_iSteadyStateHorizon+1-
k)*(*m_pDoublePastOutputChangesad)[(k-1)*iNumOutputs+j];
                }
            }
        }

        // Gad3 for BOL_INTEG * AccumulatedMoveINTEGCORRad
        for (i=1; i<= iNumInputs; i++)
        {
            for (int ijk=1;
ijk<=iNumInputs+iNumInputs*iNumOutputs*m_iSteadyStateHorizon; ijk++)
                (*m_pGad3)[i][ijk] = 0;
            for (int j=1; j<=iNumOutputs; j++)
            {
                (*m_pGad3)[i][iNumInputs+(i-
1)*iNumOutputs*m_iSteadyStateHorizon+j] =
                        (double)m_iSteadyStateHorizon *
(*AccumulatedMoveINTEGCORRad)[i];
                (*m_pGad3)[i][iNumInputs+(i-
1)*iNumOutputs*m_iSteadyStateHorizon+j+iNumOutputs] =
                    - (double)m_iSteadyStateHorizon *
(*AccumulatedMoveINTEGCORRad)[i];
            }
        }

        // Make combined observation matrix Gad
        (*m_pGad) = (*m_pGad) - (*m_pGad1) + (*m_pGad2) + (*m_pGad3);


        // Gad has been updated above .....

        // Measured change of state
        CVector m_pDmeas(iNumInputs);
        for (i=1; i<=iNumInputs; i++)
        {
            m_pDmeas[i] = ((CIOStream*)m_pInputStreams[i-1])->Read (m_csName) -
(*m_pxStackad)[i][1];
        }


        // error in prediction of change over last time-step
        if (m_bUseAdapt)
        {
            *m_pPead = m_pDmeas - (*m_pGad) * (*m_ppad);
        }
        else
        {
            *m_pPead = m_pDmeas - (*m_pGad) * (*m_ppad_orig);
        }

        //####MM010109 Test Output*****************************
        // char Values[100];
        // sprintf(Values, "e1=%9.6f e2=%9.6f amad1=%9.6f amad2=%9.6f
amcor1=%9.6f
amcor2=%9.6f",(*m_pPead)[1],(*m_pPead)[2],(*AccumulatedMoveINTEGCORRad)[1],(
*AccumulatedMoveINTEGCORRad)[2],(*AccumulatedMoveINTEGCORR)[1],(*Accumul
atedMoveINTEGCORR)[2]);
        // ((CMainWnd*)AfxGetMainWnd ())->SetStatusBarText (Values);
        //####*****************************************

        // smooth it
        for (i = 1; i <= iNumInputs; i++)
        {
            (*m_pSPead)[i] = (double) m_bGradFback_Int * (m_fGradSmooth_Int *
(*m_pSPead)[i] +
                        (1.0-m_fGradSmooth_Int) * (*m_pSPead)[i]);
        }
```

```
    if(m_bEnableAdapt)
    {
        // TimerUpdate for Adaptation of Model
        //------------------------------------------------------------ IG

        *m_pKad = (*m_pMad) * m_pGad->Transposed() *
            ((*m_pGad) * (*m_pMad) * m_pGad->Transposed() +
(*m_pRad)).Inverse();

        // Error covariance matrix
        *m_pMad = (*m_pAad) * ((*m_piad) - (*m_pKad) * (*m_pGad)) *
            (*m_pMad) * m_pAad->Transposed() + (*m_pQad);

        // Predicted output
        *m_ppad = (*m_pAad) * (*m_ppad) + (*m_pBad) * (*m_ppad_orig) +
            (*m_pKad) * (m_pDmeas - (*m_pSPead) - (*m_pGad) * (*m_ppad));
// Identifying Filter !!
        // NB : (*m_pSPead) : smoothed steady gradient from integration of
unknown moves stripped out - MM010103


        // REGULARISED SOLUTION OPTION

        *m_pGadw = (*m_pGad) * (*m_pFadw);  // transform

        *m_pKadw = (*m_pMadw) * m_pGadw->Transposed() *
            ((*m_pGadw) * (*m_pMadw) * m_pGadw->Transposed() +
(*m_pRadw)).Inverse();

        // Error covariance matrix
        *m_pMadw = (*m_pAadw) * ((*m_piadw) - (*m_pKadw) * (*m_pGadw)) *
            (*m_pMadw) * m_pAadw->Transposed() + (*m_pQadw);

        // Predicted output
        *m_ppadw = (*m_pAadw) * (*m_ppadw) + (*m_pBadw) * (*m_ppadw_orig)
+
            (*m_pKadw) * (m_pDmeas - (*m_pSPead) - (*m_pGadw) *
(*m_ppadw));  // Identifying Filter !!
        // NB : (*m_pSPead) : smoothed steady gradient from integration of
unknown moves stripped out - MM010103

        // Load back to original arrays
        if (!m_bRegularizeAdapt)
        {
            for (i=1; i<=iNumInputs; i++)
            {
                (*m_pead)[i] = (*m_ppad)[i] ;
                m_pScadDoc->SetDMCead( i, (*m_pead)[i]);
                for (int j=1; j<=iNumOutputs; j++)
                {
                    for (int k=1; k<=m_iSteadyStateHorizon; k++)
                    {
                        int jj = (k-1)*iNumOutputs + j;
                        double B0adijj = (*m_ppad)[iNumInputs+(i-
1)*iNumOutputs*m_iSteadyStateHorizon+jj];
                        (*m_pB0ad)[i][jj] = B0adijj;
                        m_pScadDoc->SetDMCB0ad( i, jj, B0adijj);
                    }
                }
            }
            for (int iT = 1; iT <= m_iSteadyStateHorizon; iT++)
            {
                for (int i = 1; i <= iNumInputs; i++)
                {
                    for (int j = 1; j <= iNumOutputs; j++)
                    {
                        int iTT;
                        if ((m_bSuppressINTEGad)||(!m_bExtSlope_Int))  // Suppress
adaptation for Integrating System ?
                        {
                            iTT=1;  // Uniform matrix
                        }
                        else
                        {
                            iTT=iT;  // Ramping matrix
                        }
                        int jj = (m_iSteadyStateHorizon-iT)*iNumOutputs+j;
                        (*m_pBOLad)[i][jj] =  (double)(iTT+1) * (*m_pB0ad)[i][j] -
                            (double)iTT * (*m_pB0ad)[i][j+iNumOutputs];
                    }  //Extrapolating in case of integration
                }
            }
        }
        else
        {
            // for regularised solution, must apply factors to get curves
            for (i=1; i<=iNumInputs; i++)
            {
                (*m_pead)[i] = (*m_ppadw)[i] ;
                m_pScadDoc->SetDMCead( i, (*m_pead)[i]);
                for (int j=1; j<=iNumOutputs; j++)
                {
                    // send regularisation factors for printing on graph
                    double regswij = (*m_ppadw)[iNumInputs +(i-1)*iNumOutputs
+j];
                    m_pScadDoc->SetDMCregsw( i,j, regswij);  // scaling factor
                    double regtwij = (*m_ppadw)[iNumInputs*(1+iNumOutputs)+(i-
1)*iNumOutputs +j];
                    m_pScadDoc->SetDMCregtw( i,j, regtwij);  // time-shift factor
                    for (int k=1; k<=m_iSteadyStateHorizon; k++)
                    {
                        int kk = k-1;
                        double B0adijk = (*m_ppad_orig)
                            [iNumInputs+(i-1)*iNumOutputs*m_iSteadyStateHorizon
*
                            (m_iSteadyStateHorizon-k)*iNumOutputs +j];
                        double B0adijkk;
                        if (kk>0)
                        {
                            B0adijkk = (*m_ppad_orig)
                            [iNumInputs+(i-1)*iNumOutputs*m_iSteadyStateHorizon
*
                            (m_iSteadyStateHorizon-kk)*iNumOutputs +j];
                        }
                        else
                        {
                            B0adijkk = 0.0;
                        }
                        double B0adreg = regswij*B0adijk + regtwij*B0adijkk;
                        (*m_pB0ad)[i][(m_iSteadyStateHorizon-k)*iNumOutputs +j] =
B0adreg;
                        m_pScadDoc->SetDMCB0ad( i,(m_iSteadyStateHorizon-
k)*iNumOutputs +j, B0adreg);
                    }
                }
            }

            // also reset the double delay prediction matrix
            for (int iT = 1; iT <= m_iSteadyStateHorizon; iT++)
            {
                for (int i = 1; i <= iNumInputs; i++)
                {
                    for (int j = 1; j <= iNumOutputs; j++)
                    {
                        int iTT;
                        if ((m_bSuppressINTEGad)||(!m_bExtSlope_Int))  // Suppress
adaptation for Integrating System ?
                        {
                            iTT=1;  // Uniform matrix
                        }
                        else
                        {
                            iTT=iT;  // Ramping matrix
                        }
                        int jj = (m_iSteadyStateHorizon-iT)*iNumOutputs+j;
                        (*m_pBOLad)[i][jj] =  (double)(iTT+1) * (*m_pB0ad)[i][j] -
                            (double)iTT * (*m_pB0ad)[i][j+iNumOutputs];
                    }  //Extrapolating in case of integration
                }
            }
        }

        // Notify RTCurveGraph via ScadDoc via RTCurvePage
        m_pScadDoc->SetAdaptation_updated (TRUE);
        //------------------------------------------------------------ IG
    }

    if (m_bTempEnabled) SetEnable(FALSE);
}


// DMC update code
double DeltaTime = (double)((int)dwTime - (int)m_dwPrevTime) / 1000.0;
if (DeltaTime >= m_fTimeInterval && m_bLPThreadRunning && m_bEnabled)
{
  // notify user of problem
  ((CMainWnd*)AfxGetMainWnd ())->SetStatusBarText ("DMC calcs take longer than
update interval");
  MessageBeep (MB_ICONASTERISK);
  // adjust previous time
  m_dwPrevTime = GetTickCount () - 1000 * (int)(m_fTimeInterval - 1);
}

if (DeltaTime >= m_fTimeInterval && !m_bLPThreadRunning && m_bEnabled)
{
  // update time
  m_dwPrevTime += (int)m_fTimeInterval * 1000;

  // update main matrices if necessary ..... NB!
  AdaptMatrices ();

  // calculate InputMIN, InputMAX, MeasuredInput and InputSetpoint
  // vectors
  CVector MeasuredInputs (iNumInputs * m_iOptimisationHorizon);
  CVector InputSetpoints (iNumInputs * m_iOptimisationHorizon);
  CVector InputMIN (iNumInputs * m_iOptimisationHorizon);
  CVector InputMAX (iNumInputs * m_iOptimisationHorizon);

  for (int i = 1; i <= iNumInputs; i++)
  {
    for (int iTime = 0; iTime < m_iOptimisationHorizon; iTime++)
    {
      // NOTE: DMC model input = DMC object output and vice versa
      MeasuredInputs[i + iTime * iNumInputs] = ((CIOStream*)m_pInputStreams[i-1])->
>Read (m_csName);
      InputSetpoints[i + iTime * iNumInputs] = d(m_fSetpoints[i-1]);
      InputMIN[i + iTime * iNumInputs] = d(m_fInputMins[i-1]);
      InputMAX[i + iTime * iNumInputs] = d(m_fInputMaxs[i-1]);
    }

    // Integration of moves that "fell off the back"  #### MM001001 for INTEGRATING
systems
    // ... and detected gradient correction
    CVector OpenLoopOutputAccumINTEG (iNumInputs * m_iOptimisationHorizon);
    CVector OutputAccumINTEG(iNumInputs);
    OutputAccumINTEG = (*m_pBOL_INTEG) * (*AccumulatedMoveINTEGCORR);

    CVector GradCorrection (iNumInputs * m_iOptimisationHorizon);
    for (int j=1; j<=m_iOptimisationHorizon; j++)
    {
      for (i=1; i<=iNumInputs; i++)
      {
        double OutputAccumINTEG_interp = OutputAccumINTEG[i] *
            (((double)m_fTimeFactorPositions[j-1]) /
((double)m_iSteadyStateHorizon));
        OpenLoopOutputAccumINTEG[(j-1)*iNumInputs+i] =
(double)m_bAccMoves_Int * OutputAccumINTEG_interp;
        GradCorrection [(j-1)*iNumInputs+i] = ((double) m_bGradFback_Int) *
(*m_pSPead)[i] *
            (((double)m_fTimeFactorPositions[j-1]) /
((double)m_iSteadyStateHorizon));
      }
    }

    // calculate open loop error vector
    CVector OpenLoopOutput (iNumInputs * m_iOptimisationHorizon);
    CVector OpenLoopError (iNumInputs * m_iOptimisationHorizon);
    OpenLoopOutput = MeasuredInputs + ((*m_pOpenLoopMatrix - *m_pOffsetMatrix) *
        *m_pPastOutputChanges) + OpenLoopOutputAccumINTEG + GradCorrection;
//
    // The GradCorrection is a feedback correction based on observed output, whilst the
```

```
    // OpenLoopOutputAccumINTEG is a feedforward correction based on accumulated
moves

   OpenLoopError = OpenLoopOutput - InputSetpoints;

   m_PlotWnd.SetOpenLoopOutput (&OpenLoopOutput);

   // calculate unconstrained quadratic optimal for deltam*
   m_pDeltaMUQO = new CVector (iNumOutputs * m_iNumberControlMoves);
   *m_pDeltaMUQO = -(*m_pLeastSqrInvMatrix) * OpenLoopError;

   // calculate OutputMIN, OutputMAX, PresentOutput and MAXOutputChange
   // vectors
   CVector OutputMIN (iNumOutputs * m_iNumberControlMoves);
   CVector OutputMAX (iNumOutputs * m_iNumberControlMoves);
   m_pPresentOutput = new CVector (iNumOutputs * m_iNumberControlMoves);
   CVector MAXOutputChange (iNumOutputs * m_iNumberControlMoves);

   for (i = 1; i <= iNumOutputs; i++)
    for (int iTime = 0; iTime < m_iNumberControlMoves; iTime++)
    {
     // NOTE: DMC model input = DMC object output and vice versa
     int index = i + iTime * iNumOutputs;
     OutputMIN[index] = d(m_fOutputMins[i-1]);
     OutputMAX[index] = d(m_fOutputMaxs[i-1]);
     MAXOutputChange[index] = d(m_fMaxChanges[i-1]);
     (*m_pPresentOutput)[index] = ((CIOStream*)m_pOutputStreams[i-1])->Read
(m_csName);
    }

   // create linear program object
   m_pLProg = new CLinearProgram (iNumOutputs * m_iNumberControlMoves * 2);

   // add output limit constraints
   CMatrix OLConstraints (iNumOutputs * m_iNumberControlMoves,
              iNumOutputs * m_iNumberControlMoves * 2);
   CVector OLValues (iNumOutputs * m_iNumberControlMoves);

   for (i = 1; i <= iNumOutputs * m_iNumberControlMoves; i++)
    for (int j = 1; j <= iNumOutputs * m_iNumberControlMoves; j++)
    {
     OLConstraints[i][j*2 - 1] = (*m_pLowerTrilDMatrix)[i][j];
     OLConstraints[i][j*2] = -(*m_pLowerTrilDMatrix)[i][j];
    }

   OLValues = OutputMIN - (*m_pPresentOutput) - (*m_pLowerTrilDMatrix) *
(*m_pDeltaMUQO);

   for (i = 1; i <= iNumOutputs * m_iNumberControlMoves; i++)
    m_pLProg->AddConstraint (CConstraint (OLConstraints[i], GreaterThan,
OLValues[i]));

   OLValues = OutputMAX - (*m_pPresentOutput) - (*m_pLowerTrilDMatrix) *
(*m_pDeltaMUQO);

   for (i = 1; i <= iNumOutputs * m_iNumberControlMoves; i++)
    m_pLProg->AddConstraint (CConstraint (OLConstraints[i], LessThan,
OLValues[i]));

   // add input ramp limit constraints
   CMatrix ORLConstraints (iNumOutputs * m_iNumberControlMoves,
              iNumOutputs * m_iNumberControlMoves * 2);
   CVector ORLValues (iNumOutputs * m_iNumberControlMoves);

   for (i = 1; i <= iNumOutputs * m_iNumberControlMoves; i++)
    for (int j = 1; j <= iNumOutputs * m_iNumberControlMoves; j++)
    {
     ORLConstraints[i][j*2 - 1] = (i==j);
     ORLConstraints[i][j*2] = -(i==j);
    }

   ORLValues = -MAXOutputChange - *m_pDeltaMUQO;

   for (i = 1; i <= iNumOutputs * m_iNumberControlMoves; i++)
    m_pLProg->AddConstraint (CConstraint (ORLConstraints[i], GreaterThan,
ORLValues[i]));

   ORLValues = MAXOutputChange - *m_pDeltaMUQO;

   for (i = 1; i <= iNumOutputs * m_iNumberControlMoves; i++)
    m_pLProg->AddConstraint (CConstraint (ORLConstraints[i], LessThan,
ORLValues[i]));

   // add input limit constraints
   CMatrix ILConstraints (iNumInputs * m_iOptimisationHorizon,
              iNumOutputs * m_iNumberControlMoves * 2);
   CVector ILValues (iNumInputs * m_iOptimisationHorizon);

   for (i = 1; i <= iNumInputs * m_iOptimisationHorizon; i++)
    for (int j = 1; j <= iNumOutputs * m_iNumberControlMoves; j++)
    {
     ILConstraints[i][j*2 - 1] = (*m_pDynMatrix)[i][j];
     ILConstraints[i][j*2] = -(*m_pDynMatrix)[i][j];
    }

   ILValues = InputMIN - OpenLoopOutput - (*m_pDynMatrix) * (*m_pDeltaMUQO);

   for (i = 1; i <= iNumInputs * m_iOptimisationHorizon; i++)
    m_pLProg->AddConstraint (CConstraint (ILConstraints[i], GreaterThan,
ILValues[i]));

   ILValues = InputMAX - OpenLoopOutput - (*m_pDynMatrix) * (*m_pDeltaMUQO);

   for (i = 1; i <= iNumInputs * m_iOptimisationHorizon; i++)
    m_pLProg->AddConstraint (CConstraint (ILConstraints[i], LessThan, ILValues[i]));

   // optimise
   m_pObjectiveFn = new CVector (iNumOutputs * m_iNumberControlMoves * 2);

   // new weighting
   CVector vTemp = (m_pDynMatrix->Transposed () * *m_pWeightingMatrix *
OpenLoopError);
   for (i = 1; i <= m_pObjectiveFn->Size (); i++)
    (*m_pObjectiveFn)[i] = fabs (vTemp[(i+1) / 2]);

   // old weighting
//   for (i = 1; i <= iNumOutputs * m_iNumberControlMoves * 2; i++)
//    (*m_pObjectiveFn)[i] = 1.0;

   m_pResult = new CVector (iNumOutputs * m_iNumberControlMoves * 2);

   // start LP thread
   m_bLPThreadRunning = TRUE;
   DWORD ThreadID;
   m_hThread = CreateThread (NULL, 0, ThreadFunction, this, NULL, &ThreadID);
  }

  if (!m_bLPThreadRunning && !m_bEnabled)
  {
   // adjust previous time
   m_dwPrevTime = GetTickCount () - 1000 * (int)(m_fTimeInterval - 1);
  }

  if (m_bLPThreadFinished && m_bIgnoreLPThreadResult)
  {
   m_bLPThreadRunning = FALSE;
   m_bLPThreadFinished = FALSE;
   m_bIgnoreLPThreadResult = FALSE;
   delete m_pLProg;
   delete m_pResult;
   delete m_pObjectiveFn;
   delete m_pDeltaMUQO;
   delete m_pPresentOutput;
  }

  if (m_bLPThreadFinished && !m_bIgnoreLPThreadResult)
  {
   m_bLPThreadRunning = FALSE;
   m_bLPThreadFinished = FALSE;

   if (m_iLPError == SIMPLEX_NOFEASIBLESOLUTION)
   {
    // notify user of problem
    ((CMainWnd*)AfxGetMainWnd ())->SetStatusBarText ("No solution in LP
constraints");
    MessageBeep (MB_ICONASTERISK);
   }

   if (m_iLPError == SIMPLEX_OBJECTIVEUNBOUNDED)
   {
    // notify user of problem
    ((CMainWnd*)AfxGetMainWnd ())->SetStatusBarText ("LP objective function
unbounded");
    MessageBeep (MB_ICONASTERISK);
   }

   if (m_iLPError == SIMPLEX_NOERROR)   // ==== MM000929
    {
        // get number of outputs
        int iNumOutputs = m_csOutputStreamNames.GetSize (); // outputs from
DMCObject

        // calculate delta outputs
        CVector DeltaM (*m_pDeltaMUQO);
        for (int i = 1; i <= iNumOutputs * m_iNumberControlMoves; i++)
            DeltaM[i] += (*m_pResult)[i*2-1] - (*m_pResult)[i*2];

        m_PlotWnd.Plot (&DeltaM);

        // set outputs of DMCObject
        for (i = 1; i <= iNumOutputs; i++)
        {
            double m_fOutput = (*m_pPresentOutput)[i] + DeltaM[i];
            m_fOutput = max (d(m_fOutputMins[i-1]) , m_fOutput);  // Clip in case went
out
            m_fOutput = min (d(m_fOutputMaxs[i-1]) , m_fOutput);
            ((CIOStream*)m_pOutputStreams[i-1])->Write (m_csName, m_fOutput);
        }
    }


   delete m_pLProg;
   delete m_pResult;
   delete m_pObjectiveFn;
   delete m_pDeltaMUQO;
   delete m_pPresentOutput;
  }
}

void CDMCObject::LinearProgramThread (void)
{
 double Minimum;
 m_iLPError = m_pLProg->Minimise (*m_pObjectiveFn, *m_pResult, Minimum);
 m_bLPThreadFinished = TRUE;
}

void CDMCObject::SetEnable (BOOL bEnabled)
{
 // connect/disconnect to controlled streams
 if (bEnabled && !m_bEnabled)
 {
  // scan to see whether any controlled streams have been usurped while
  // this object was disabled
  BOOL bTemp = TRUE;
  for (int i = 0; i < m_pInputStreams.GetSize (); i++)
   if (((CIOStream*)m_pInputStreams[i])->GetController () != "User")
    bTemp = FALSE;
  for (i = 0; i < m_pOutputStreams.GetSize (); i++)
   if (((CIOStream*)m_pOutputStreams[i])->GetController () != "User")
    bTemp = FALSE;

  if (!bTemp)
  {
   AfxGetMainWnd ()->MessageBox ("One or more controlled streams are being
controlled by something else.",
                 "Error");
   return;
  }

  // connect to controlled streams
  for (i = 0; i < m_pInputStreams.GetSize (); i++)
   ((CIOStream*)m_pInputStreams[i])->Connect (m_csName);
```

```
  for (i = 0; i < m_pOutputStreams.GetSize (); i++)
    ((CIOStream*)m_pOutputStreams[i])->Connect (m_csName);
}

if (!bEnabled && m_bEnabled)
{
  for (int i = 0; i < m_pInputStreams.GetSize (); i++)
    ((CIOStream*)m_pInputStreams[i])->DisConnect (m_csName);
  for (i = 0; i < m_pOutputStreams.GetSize (); i++)
    ((CIOStream*)m_pOutputStreams[i])->DisConnect (m_csName);
}

// call base class
CSteadObject::SetEnable (bEnabled);
}

void CDMCObject::SetEnableAdapt (BOOL bEnableAdapt)
{

 // Stop or Restart Adaptation of Model - IG
 if (bEnableAdapt && !m_bEnableAdapt)
 {
    m_bRestartad = TRUE;
 }
 m_bEnableAdapt = bEnableAdapt;
}

void CDMCObject::SetUseAdapt (BOOL bUseAdapt)
{

 // Use or Stop Using Adapted model (vs Set model) for DMC - IG
 m_bUseAdapt = bUseAdapt;
}

void CDMCObject::SetRegularizeAdapt (BOOL bRegularizeAdapt)
{

 // Regularize or Stop Regularizing Adapted model for DMC - IG
 m_bRegularizeAdapt = bRegularizeAdapt;
}

void CDMCObject::SetExtSlope_Int (BOOL bExtSlope_Int)
{
    // (m_bExtSlope_Int != bExtSlope_Int)
    {
        m_bRestartad = TRUE;  // Force complete re-initialisation of Adaptive DMC -
IG
        // Extend Integral Slope or Stop Extending Integral Slope for DMC Integral
Correction - IG
        m_bExtSlope_Int = bExtSlope_Int;
    }
}

void CDMCObject::SetAccMoves_Int (BOOL bAccMoves_Int)
{
    if (m_bAccMoves_Int != bAccMoves_Int)
    {
        m_bRestartad = TRUE;  // Force complete re-initialisation of Adaptive DMC -
IG
        // Use Accumulated Moves or Stop using Accumulated Moves for DMC Integral
Correction - IG
        m_bAccMoves_Int = bAccMoves_Int;
    }
}

void CDMCObject::SetGradFback_Int (BOOL bGradFback_Int)
{
    if (m_bGradFback_Int != bGradFback_Int)
    {
        m_bRestartad = TRUE;  // Force complete re-initialisation of Adaptive DMC -
IG
        // Observed Gradient Feedback Correction or Stop Observed Gradient
Feedback Correction for DMC Integral Correction - IG
        m_bGradFback_Int = bGradFback_Int;
    }
}

void CDMCObject::SetGradSmooth_Int (double fGradSmooth_Int)
{
  // Filter Smoothing Parameter for Observed Gradient Feedback Correction for DMC
Integral Correction - IG
  m_fGradSmooth_Int = fGradSmooth_Int;
}

void CDMCObject::SetIODeviceModelData (void)
{
  // set aside parameters to be used by IODeviceModel
    nI=m_pInputStreams.GetSize ();
    nO=m_pOutputStreams.GetSize ();
    nS=m_iSteadyStateHorizon;
    modint=m_fTimeInterval;
    cStepResponseFile = m_cStepResponseFile;
}

void CDMCObject::StreamWrite (int iIndex, double fValue)
{
    m_fSetpoints[iIndex] = w(fValue);
}

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

void CDMCObject::AdaptMatrices (void)

{
    // Adapt DMC Matrices if required

    if (!(m_bUseAdapt) && (m_bUseAdapt_last)) // change back to Set matrices
    {
        if(m_bUseAdapt) //back to Set dynamic matrix
        {
            //back to Set Matrices
            for (int i = 1; i <= m_iOptimisationHorizon*iNumInputs; i++)
                for (int j = 1; j <=  m_iNumberControlMoves*iNumOutputs; j++)
                {
                    (*m_pDynMatrix)[i][j] = (*m_pDynMatrix_Set)[i][j];  //Back to stored
sSet Matrix
                }
            for (i = 1; i <= m_iOptimisationHorizon*iNumInputs; i++)
                for (int j = 1; j <=  m_iSteadyStateHorizon*iNumOutputs; j++)
                {
                    (*m_pOpenLoopMatrix)[i][j] = (*m_pOpenLoopMatrix_Set)[i][j];
//Back to stored sSet Matrix
                }
            for (i = 1; i <= m_iOptimisationHorizon*iNumInputs; i++)
                for (int j = 1; j <=  m_iSteadyStateHorizon*iNumOutputs; j++)
                {
                    (*m_pOffsetMatrix)[i][j] = (*m_pOffsetMatrix_Set)[i][j];  //Back to
stored sSet Matrix
                }
        }
    }

    if (m_bUseAdapt) // using adapted matrices
    {
        // use current adapted model for dynamic matrix
        for (int k = 1; k <= m_iNumberControlMoves; k++)
        {
            for (int i = 1; i <= iNumInputs; i++)
            {
                int iT=0;
                for (int iTime = 1; iTime <= m_iSteadyStateHorizon; iTime++)
                {
                    if (fabs(m_fTimeFactors[iTime-1]) > 2e-50)  // ie. non-zero
                    {
                        iT++;
                        if ((iT-1+k)<=(m_iOptimisationHorizon))  // not off the end yet
                        {
                            for (int j = 1; j <= iNumOutputs; j++)
                            {
                                int ij = (m_iSteadyStateHorizon-iTime)*iNumOutputs+j;
                                (*m_pDynMatrix)[(iT-1+k-1)*iNumInputs+i][(k-
1)*iNumOutputs+j]
                                    = (*m_pB0ad)[ij][i];
                            }
                        }
                    }
                }
            }
        }

        // must recalculate dependent matrices
        // open loop matrix values
        int i=0;
        for (int iTime = 1; iTime <= m_iSteadyStateHorizon; iTime++)
        {
            if (fabs(m_fTimeFactors[iTime-1]) > 2e-50)  // ie. non-zero
            {
                i++;
                for (int j = 1; j <= m_iSteadyStateHorizon; j++)
                {
                    int kk = m_iSteadyStateHorizon-j+iTime+1;
                    int k = min(kk,m_iSteadyStateHorizon);
                    for (int iInput = 1; iInput <= iNumInputs; iInput++)
                    {
                        for (int jOutput = 1; jOutput <= iNumOutputs; jOutput++)
                        {
                            // set value for ii, jj
                            int ii=(i-1)*iNumInputs + iInput;
                            int jj=(j-1)*iNumOutputs + jOutput;
                            int jj = (m_iSteadyStateHorizon-k)*iNumOutputs+jOutput;
                            if ((k==kk)||(m_bExtSlope_Int))
                            {
                                (*m_pOpenLoopMatrix)[ii][jj] = (*m_pB0ad)[jj][ii];
                            }
                            else
                            {  // Extrapolation in case of INTEG slope at end of identified
response
                                (*m_pOpenLoopMatrix)[ii][jj] =
                                  (double)(kk-k+1)*(*m_pB0ad)[iInput][jj+iNumOutputs]
                                  (double)(kk-k)*(*m_pB0ad)[iInput][jj+iNumOutputs];
                            }
                        }
                    }
                }
            }
        }

        // offset matrix values
        i=0;
        for (iTime = 1; iTime <= m_iSteadyStateHorizon; iTime++)
        {
            if (fabs(m_fTimeFactors[iTime-1]) > 2e-50)  // ie. non-zero
            {
                i++;
                for (int j = 1; j <= m_iSteadyStateHorizon; j++)
                {
                    for (int iInput = 1; iInput <= iNumInputs; iInput++)
                    {
                        for (int jOutput = 1; jOutput <= iNumOutputs; jOutput++)
                        {
                            // set value for ii, jj
                            int ii=(i-1)*iNumInputs + iInput;
                            int jj=(j-1)*iNumOutputs + jOutput;
                            int jj = (j-1)*iNumOutputs+jOutput;
                            (*m_pOffsetMatrix)[ii][jj] = (*m_pB0ad)[iInput][jj];
                        }
                    }
                }
            }
        }

        // Also use the identified final slope for the INTEG system correction under
control
        for (j = 1; j <= iNumInputs; i++)
        {
            for (int j = 1; j <= iNumOutputs; j++)
            {
                (*m_pBDL_INTEG)[i][j] = (double)m_bAccMoves_Int *
(double)m_iSteadyStateHorizon *
                                        ((*m_pB0ad)[i][j]-(*m_pB0ad)[i][j+iNumOutputs]);
            }       //A differential term for the tail slope for integrating systems control
        }
```

```
    }

    if ((m_bUseAdapt) | (m_bUseAdapt_last))
    {
        // set least squares inverse matrix
        *m_pLeastSqrInvMatrix =
            (m_pDynMatrix->Transposed () * *m_pWeightingMatrix * *m_pDynMatrix *
             *m_pMoveSupMatrix).Inverse () * m_pDynMatrix->Transposed () *
             *m_pWeightingMatrix;
    }

    // Update
    m_bUseAdapt_last = m_bUseAdapt;

}

        // Test #####
        // CVector Test(iNumInputs); //#######
        // Test = (*m_pBOLad_orig) * (*m_pDoublePastOutputChangesad); //#######

        // for (i=1; i<= iNumInputs; i++)
        // {
        //    (*m_pGad)[i][i] = 0.0;
        // }
        // ###(*m_pPead) = (*m_pPead) - m_pDmeas + ((*m_pGad3) + (*m_pGad2) -
(*m_pGad1)) - (*m_pGad)) * (*m_ppad);
        // (*m_pPead) = Test - (*m_pGad2) * (*m_ppad_orig);
        // for (i=1; i<= iNumInputs; i++)
        // {
        //    (*m_pGad)[i][i] = 1.0;
        // }

        //####MM001216 Test Output**********
       // char Values[100];
        // Test = (*m_pBOL_INTEG) * (*AccumulatedMoveINTEGCORRad);
       // sprintf(Values, "Tst1=%9.6f Tst2=%9.6f FBk1=%9.6f FBk2=%9.6f Off1=%9.6f
Off2=%9.6f",Test[1],Test[2],(*m_pSPead)[1],(*m_pSPead)[2],(*m_pPead)[1],(*m_pPea
d)[2]);
        // sprintf(Values, "ediff1=%9.6f ediff2=%9.6f",(*m_pPead)[1],(*m_pPead)[2]);
        // ((CMainWnd*)AfxGetMainWnd ())->SetStatusBarText (Values);
        //####**********
```