UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

A Systematic Approach to Model Predictive Controller Constraint Handling: Rigorous Geometric Methods

by

André Herman Campher

A dissertation submitted in partial fulfillment
of the requirements for the degree

Master of Engineering (Control Engineering)

in the

Department of Chemical Engineering
Faculty of Engineering, the Built Environment and Information Technology

University of Pretoria
Pretoria

September 2011

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

A Systematic Approach to Model Predictive Controller Constraint Handling: Rigorous Geometric Methods

By: **André Herman Campher**
Study leader: **Carl Sandrock**
Department: **Chemical Engineering**
Degree: **MEng Control Engineering**

SYNOPSIS

The models used by model predictive controllers (MPCs) to predict future outcomes are usually unconstrained forms like impulse or step responses and discrete state space models. Certain MPC algorithms allow constraints to be imposed on the inputs or outputs of a system; but they may be infeasible as they are not checked for consistency via the process model. Consistent constraint handling methods – which account for their interdependence and disambiguate the language used to specify constraints – would therefore be an attractive aid when using any MPC package.

A rigorous and systematic approach to constraint management has been developed, building on the work of Vinson (2000), Lima (2007) and Georgakis *et al.* (2003) in interpreting constraint interactions. The method supports linear steady-state system models, and provides routines to obtain the following information:

- effects of constraint changes on the corresponding input and output constraints,
- feasibility checks for constraints,
- specification of constraint-set size and
- optimal fitting of constraints within the desirable input and output space.

Mathematical rigour and unambiguous language for identifying constraint types were key design criteria.

The outputs of the program provide guidance when handling constraints, as opposed to rules of thumb and experience, and promote understanding of the system and its constraints. The metrics presented are not specific to any commercial MPC and can be implemented in the user interfaces of such MPCs. The method was applied to laboratory-scale test rigs to illustrate the information obtained.

KEYWORDS: constraint handling, model predictive controllers, geometric methods, operability index

ACKNOWLEDGEMENTS

I would like to thank my study leader, Carl Sandrock, for the insights shared (and for his patience).

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

A	Coefficient matrix for constraints (used with subscript)
b	half-space offsets for constraints (used with subscript)
d	System disturbance
G	Process model
n	Number of dimensions (variables)
u	System inputs
x	System states
y	System outputs

Abbreviations

AIS	Available Input Space
AOIS	Achievable Output Interval Set
AOS	Available Output Space
AOS_I	Interval AOS
CV	Controlled variable
DIS	Desired Input Space
DMCPlus	Dynamic Matrix Control Plus
DOS	Desired Output Space
DV	Disturbance variable
EDS	Expected Disturbance Space

MPC Model Predictive Control(ler)

MV Manipulated variable

OI Operability Index

OOI Output Operability Index

RMPCT Robust Model Predictive Control Technology

Greek

μ Volume of a polytope

CHAPTER 1

INTRODUCTION

1.1 Background

Model predictive controllers (MPCs) use models to predict the future behaviour of a process. The ability of some MPC algorithms to impose constraints on the outputs or inputs of a system is their biggest selling point.

In the same way the process model predicts the effect of the inputs on the outputs, the process model dictates the interdependence of the constraints imposed on the system. At present, commercial MPC packages do not validate constraints based on the process model. This gives rise to the following problems:

- Specified constraints on an input or output may be infeasible due to their corresponding output or input requirements.
- Setpoint specification may be infeasible as the process has a limited output space (due to the input constraints).

A systematic approach to constraint management would therefore be an attractive addition to any MPC package. Such a method would help to avoid infeasible constraints and further the understanding of constraint interaction in MPCs. This follows from the use of the model to determine constraint interactions, and not rules of thumb or specific process experience.

1.2 Problem statement

The objective is therefore to develop a systematic method of managing constraints imposed on MPCs. Mathematical rigour and disambiguation of the language used to specify constraints were key design criteria.

1.3 Method

The method builds on the work by Vinson (2000), Lima (2007) and Georgakis *et al.* (2003) in interpreting constraint interactions.

The routines for this project were developed using the Python (Python Software Foundation, 2010) programming language. Version management of the source code was done using git (Git, 2010) and hosted on github (GitHub Inc., 2010; Campher, 2011b,a). This allowed for continuous and collaborative development and flagging of issues.

The routines were tested for functionality on the models and operating conditions of laboratory rigs.

1.4 Scope and deliverables

For this project, linear steady-state models are considered. The constraints imposed on the systems are also assumed to be linear. With these assumptions made, the project can now focus exclusively on convex sets in the input and output space.

This project does not attempt to reinvent or propose an alternative to the method of internal constraint handling of MPC algorithms. The method aims to provide a supportive framework in both the controller design and operation phases.

The deliverables of this project are;

- A framework for systematic constraint handling for MPCs.
- Routines to perform an analysis of constraints based on interaction via the process model.
- Results indicating the use of the aforementioned routines on practical systems.

CHAPTER 2

LITERATURE OVERVIEW

A summary of the mathematical techniques used in this dissertation is given in this chapter. Special attention is given to convex geometry and the properties of these sets.

To place the Operability Index in context, process operability and methods to quantify it are reviewed. Finally, the Operability Index is defined and its application to model predictive controllers and their constraints are discussed.

2.1 Mathematical Preliminaries

2.1.1 Convex Geometry

Nomenclature

The following nomenclature is reviewed and used throughout this document (Bayer and Lee, 1993: 487):

- polytope - the intersection of closed half-spaces in $\mathbb{R}^n$. A polytope of n dimensions will simply be referred to as an n-polytope.
- vertex - a 0-dimensional face of an n-polytope.
- facet - an $(n-1)$-dimensional face of an n-polytope.

Half-space geometry

A half-space is defined as the section of an n-dimensional space lying on one side of an $(n-1)$-dimensional hyperplane (Weisstein, 2003: 1282). Linear constraints are equivalent to half-spaces in $\mathbb{R}^n$ in both their mathematical description and meaning. From this we can conclude that all polytopes generated by linear constraints (half-spaces) are convex, as they have supporting hyperplanes at each boundary point (Mani-Levitska, 1993: 21).

From this it also follows that the intersection of such convex polytopes will be convex. This is due to the resulting intersection being merely a subset of the half-spaces defining the two intersecting polytopes.

The problem of degeneracy can occur when constructing polytopes from half-spaces. Degeneracy is defined when objects "change their nature to belong to another, usually simpler, class" (Weisstein, 2003: 688). The degeneracy of polytopes is important for the purposes of this dissertation. When the number of half-spaces used to construct a polytope is more than the final number of facets on the resulting closed polytope, the polytope is classified as degenerate.

Convex hull and vertex enumeration

The convex hull or facet enumeration is used extensively for calculations in this project. It is defined as the smallest convex set containing P if P is a set of points ($P \subseteq \mathbb{R}^n$) (Wenger, 2004: 74). In this project, P is generated from a set of half-spaces and the resulting vertices are the minimum vertex description of P. For this reason the convex hull of P and the half-spaces used to generate P are equivalent.

The opposite of facet enumeration is vertex enumeration, where a set of half-spaces are transformed to the vertices of P. A primal-dual method can be implemented to solve this problem. The full mathematical formulation of the primal-dual algorithm is omitted from this document. Gritzmann and Klee (1993: 636-639) and Bremner *et al.* (1998) can be consulted for the full formulation.

Linear transformations

Define X and Y as finite-dimensional sets with bases $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_m)$ respectively. For the linear transformation $\phi : X \to Y$, the following holds:

$$\phi(y_i) = \alpha_{i1}x_i + \cdots + \alpha_{in}x_n \quad \text{for } i = 1, \ldots, m \tag{2.1}$$

The scalars $(\alpha_{ij})_{i=1,\ldots,n;\ j=1,\ldots,m}$ can be written as

$$\begin{pmatrix} \alpha_{11} & \cdots & \cdots & \alpha_{1n} \\ \alpha_{21} & \cdots & \cdots & \alpha_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ \alpha_{m1} & \cdots & \cdots & \alpha_{mn} \end{pmatrix}$$

which is referred to as the matrix of ϕ (Leung, 1974: 48-49, 166).

Equation 2.1 is equivalent to the result of the following matrix multiplication

$$\Phi X = Y \tag{2.2}$$

where Φ is the matrix of ϕ. Equation 2.2 confirms that matrix multiplication is indeed a linear transformation.

2.1.2 Process models

The mathematical description of process models and specifically the space transformations those lead to are of importance.

Linear steady-state

The steady-state model of a linear system is the gain-matrix. This is simply a matrix of constants in $\mathbb{R}$. Models like these allow for matrix algebra when transforming between spaces, e.g. a matrix multiplication of the input space and the process model has the output space as result.

The transformation obtained for linear models are conformal mappings. This is defined as a mapping that preserves local angles and has a non-zero differential at every point (Weisstein, 2003: 514). Being conformal, the surface (or boundary) of the input space (a polytope) maps directly to the surface of the output space. For this reason, the constraints specifying a space can be used to generate the corresponding space with full mathematical rigour.

Non-linear models

The mapping of spaces via non-linear models is not guaranteed to be conformal. Therefore, the statement that boundaries map to boundaries in their corresponding spaces, cannot be made.

A practical example of this is illustrated by Subramanian and Georgakis (2001), for a vinyl acetate reactor where a set of inputs map to the same point in the output space. In this case, the phenomena of input multiplicities is ascribed to a rank deficiency of the Jacobian matrix of the process model.

2.2 Process Operability

The method of constraint handling presented in this document, is largely derived from work done by Georgakis and colleagues, in particular Vinson (2000), Vinson and Georgakis (2000), Lima (2007), Georgakis *et al.* (2003) and Subramanian and Georgakis (2001). To place the Operability Index of Vinson (2000) (defined in section 2.2.2) in context, a short overview of process operability and the most common process operability techniques is presented below.

2.2.1 Overview

Process operability is defined as "the ease with which a process can be operated and controlled" (Marlin, 2000: 778). Process operability is an ongoing field of study with many of the methods available being only qualitative (Skogestad and Postlethwaite, 2005: 164). A distinction is made between steady-state (input and output relationships at steady-state only) and dynamic operability methods.

Steady-state operability measures

The relative gain array (RGA) is one of the most used operability measures (Luyben, 1990: 576) and relates input and output interactivity. Each element in the RGA (β_{ij}) is defined as the ratio of the steady-state gain ratios between input, i, and output, j, when all other inputs and outputs are constant. Stanley *et al.* (1985) defined the RDG, which expanded on the RGA to include the effect of disturbances. Considering only the magnitudes of RGA elements can be misleading (Skogestad and Postlethwaite, 2005: 87), as seemingly favourable pairings can be unfavourable due to the phase. Therefore, both the magnitude and the phase need to be considered or, alternatively, norms such as the RGA-number can be used.

The Niederlinski index is another measure that only uses steady-state gains of the process model (Luyben, 1990: 572-573). Positive values of the Niederlinski index can correspond to unstable pairings and are therefore inconclusive. For this reason, the index is considered necessary but not sufficient for stability (Skogestad and Postlethwaite, 2005: 445).

Singular value decomposition stems from the use of eigenvalues as an operability measure. The use of singular values are preferred as eigenvalues are a poor measure of gain and can often be misleading (Skogestad and Postlethwaite, 2005: 75). Maximum and minimum singular values are mostly used to select controlled variables but can also be used as a performance measure along with the condition number (Luyben, 1990: 596; Skogestad and Postlethwaite, 2005: 80-82).

A competing aproach to the Operability Index (defined in the next section) is that of flexibility and the Flexibility Index defined by (Swaney and Grossmann, 1985). This approach comprises the definition of a feasible operating region (by way of fundamental equations as well as operating constraints) and then determining the maximum deviations on the combined process parameter set. The Flexibility Index corresponds to the largest allowed deviation (or flexibility) of the parameter set which still ensures feasible operation. (Grossmann and Floudas, 1987) expands on this index by reformulating the main optimisation problem to ensure solution convergence. An active constraint strategy is also applied to non-linear problems to reduce the computational effort required and ensure solution convergence (with some linearising assumptions).

Dynamic operability measures

The RGA can be expanded to contain frequency-dependant terms and is often referred to as the dynamic relative gain array (DRGA) (Marlin, 2000: 637). The RDG (Stanley *et al.*, 1985) can be expanded in a similar way to include frequency-dependant terms.

Nyquist array methods as defined by Rosenbrock, as quoted by Skogestad and Postlethwaite (2005: 92), are an extension of the SISO Nyquist stability criteria that include frequency dynamics. The two methods developed are the direct Nyquist array and the inverse Nyquist array. When used in conjunction with Gershgorin bands, conditions for overall stability can be derived (Skogestad and Postlethwaite, 2005: 440).

Vinson (2000) lists other dynamic operability measures which rely on the choice of a controller type. Further shortcomings of the above mentioned operability measures are also discussed in that text.

2.2.2 Operability Index

The Operability Index was defined by Vinson (2000) as a measure of steady-state process operability. Constraint interdependence is used as the basis of this index. Further work by Lima (2007) expanded this index to cover non-square systems and some application to dynamic systems.

Definitions

The Operability Index focuses on input-output relationships rather the system's dynamic states (Vinson, 2000). Input and output values are defined by spaces (in $\mathbb{R}^n$ and $\mathbb{R}^m$, with n the number of inputs and m the number of outputs). These spaces are the feasible regions which are bounded by the inequalities describing the ranges of the inputs or outputs. The following spaces are of interest:

Available Input Space (AIS); the set of attainable values of the process inputs. The limits on these values are based on both physical limits (e.g. valve openings) or process design values (e.g. flow-rates or temperatures). Figure 2.1 illustrates an AIS for two variables (u_1, u_2) bounded by simple high and low limits.

Achievable Output Space (AOS); this is the set of values which the process outputs can obtain, given the AIS. The AIS maps to the AOS by means of a process model, G. Therefore, a point u in the AIS corresponds to a point y in the AOS via $y = G(u)$. Examples of AOSs for linear and non-linear models are shown in figure 2.1.

Desired Output Space (DOS); this represents the desired output values of a process. These values are typically based on operational and financial parameters. Figure 2.2 shows a sample DOS for a two output system along with a sample AOS.

Expected Disturbance Space (EDS); all the values of the expected disturbances to the system. These values are translated back to the input space by means of the disturbance model, G_d.

Desired Input Space (DIS); in the same manner that the AIS maps to the AOS, the DIS represents combined reverse mappings of the DOS and EDS. The DIS is calculated as the inputs, u, that satisfy the model, i.e. $y = G(u, d)$ (with y and d from the DOS and EDS respectively). Figure 2.3 shows this combined translation.

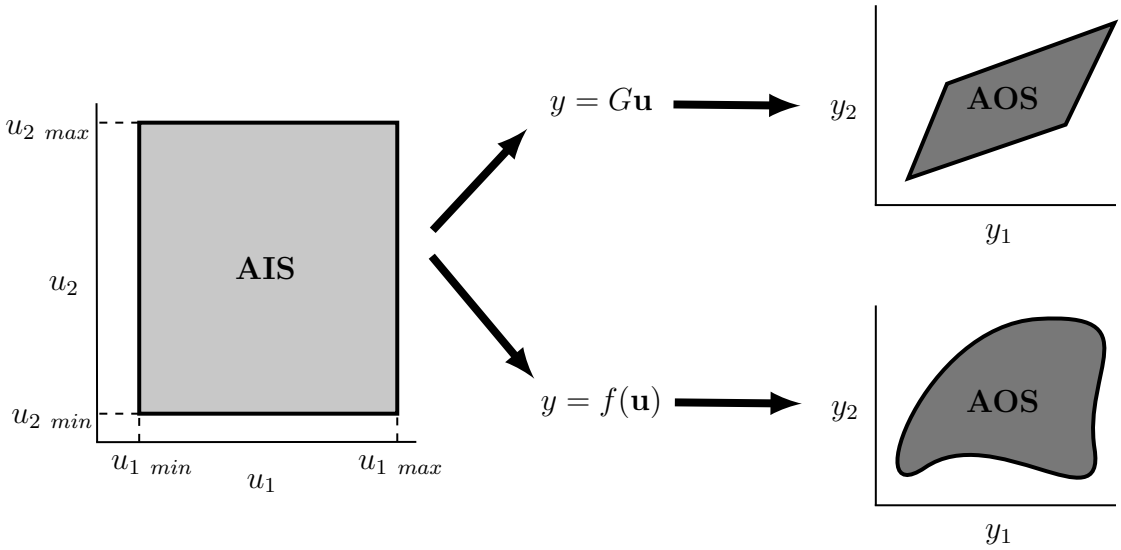

Figure 2.1: Sample Available Input Space (AIS) mapping via a linear model ($y = G\mathbf{u}$) and a non-linear model ($y = f(\mathbf{u})$) to the Achievable Output Space (AOS)

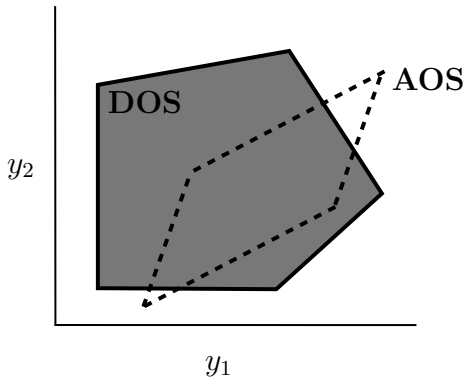

Figure 2.2: Sample Desired Output Space (DOS, with linear constraints on both outputs) along with a linear AOS

Vinson (2000) defines the generalised Output Operability Index (OOI) as shown in equation 2.3. From this definition it is clear that the operability of a process decreases if the AOS does not cover the whole DOS. Servo- and regulatory operability indices are also

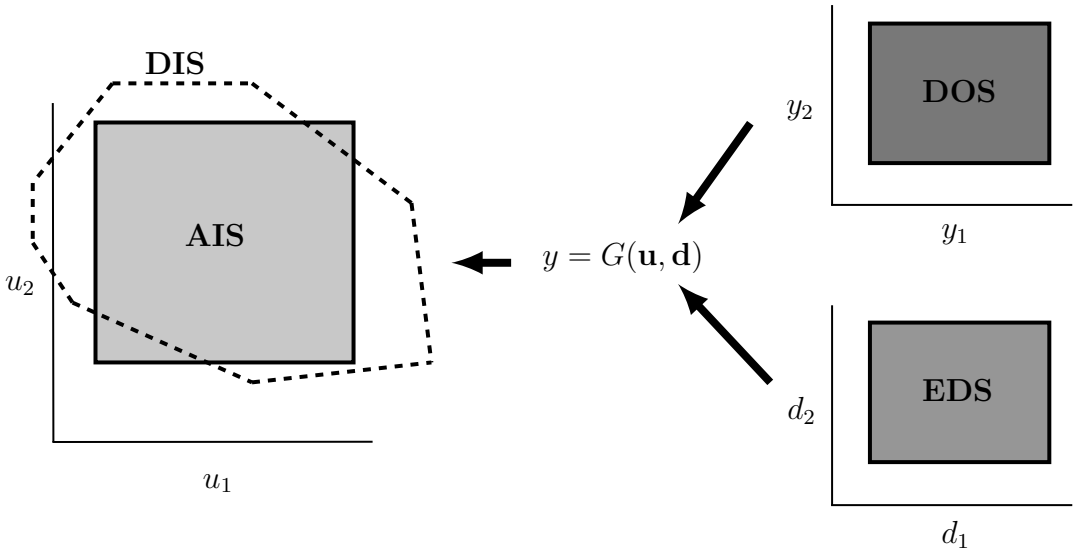

Figure 2.3: Sample Desired Input Space (DIS) for a linear model and rectangular DOS and EDS

defined in the text, but are not shown here.

$$OOI \triangleq \frac{\mu(AOS \cap DOS)}{\mu(DOS)} \tag{2.3}$$

where μ represents a function to calculate the volume of a space.

All the concepts above are directly applicable to square systems. For the case of non-square systems (more outputs than inputs) the Operability Index formulation is extended to interval operability. In this case, process outputs are split into two categories; setpoint controlled variables and range controlled variables. The first being controlled on a given setpoint, whereas the later is allowed to vary within its given maximum and minimum limits.

Lima (2007) elaborates on the interval operability of linear non-square systems and defines the following spaces:

The interval AOS (AOS_I); which is the union of all AOS(d) where d is within the EDS. This is typically generated by the shifting of AOS($d = 0$) in the directions determined by G_d (the disturbance model).

The Achievable Output Interval Set (AOIS); defined as a rectangle with the same aspect ratio as the DOS, bound by the lines (or hyperplanes) that correspond to the minimum and maximum values of d in the AOS. Since the issue of interval operability is being addressed, the AOIS does not need to be strictly contained within the AOS_I.

Lima (2007) proceeds to define a hyper-volume ratio (HVR) as shown in equation 2.4. The aim being here to define a measure for obtaining tighter control on CVs by reducing the

size of the DOS (without causing infeasibilities). The process remains interval controllable as long as the AOIS remains a subset of the DOS (AOIS⊆DOS).

$$HVR = \frac{\mu(DOS)}{\mu(AOIS)} \tag{2.4}$$

where μ represents a function to calculate the volume of a space.

To change the relative importance (tightness of control) on variables, weights are imposed on the AOIS. This in turn affects the aspect ratio of the AOIS.

Lima (2007) proposes two methods to calculate an AOIS for higher dimensional systems; an iterative method and a method based on linear programming (LP). In both cases, this corresponds to the tightest set of output constraints which still render the system controllable. Both methods are briefly discussed below:

- Iterative approach; this method relies on the use of computational geometry tools to calculate convex hulls and intersections. An AOIS is initially estimated and expanded until it intersects the extreme subsets of the AOS. The magnitude by which the AOIS is increased is calculated using the Secant and bisection method. The calculations contained in this method are computationally intensive and limit its use to systems of 8 dimensions and lower.

- LP approach; in this approach the constraints and objective function are reformulated as linear functions. The extreme subsets of the AOS are expressed as linear inequalities and objective function set up as to determine the common intersection point of the AOIS and a given extreme subset. The proposed LP approach overcomes the computational intensity of the iterative method and is suitable for on-line calculation of the AOIS.

Application

Numerous authors have illustrated the application of the Operability Index (OI). Georgakis *et al.* (2003) focus on a general application of the OI, emphasising its ability to identify inoperable processes without the need to specify a controller structure. Subramanian and Georgakis (2001) applies the OI to the design of ideal reactors.

2.2.3 Operability Index application to MPC

Although the Operability Index can be used with square or non-square, linear or non-linear models, its easiest application is to linear models. This is advantageous as most of the current MPCs use linear models (Vinson, 2000). Table 2.1 highlights some properties of MPCs which lend themselves to the application of the Operability Index.

Table 2.1: Application of the Operability Index to MPC (Vinson, 2000)

MPC property	Operability Index property
Linear process models	Calculation of the relevant spaces (AOS and DIS) is usually fast.
Constraints on MVs and CVs	In the case of fixed constraints; the input and output constraints result in polytopes in both the input and output spaces. From these descriptions the framework of Operability Index can be directly applied.
Solution of MV and CV steady state targets	Where an LP solver is used, the Operability Index can yield useful information about process characteristics.

Vinson (2000) presents a list of additional issues that limit the performance of MPCs (with specific attention to AspenTech's DMCplus©) and proposes methods (using the Operability Index) to solve these problems. These methods are indirectly applicable to the proposed constraint handling method of this document and are presented here for that reason.

Processes with considerable noise cause targets to move with each controller execution. Typical ways to avoid this are limiting MV stepsize or filtering CVs, both of which have a stabilizing effect but result in a more sluggish controller. These limits may also cause the controller to not reach optimum solutions. Using the OI and the associated output spaces, the possible outputs of a (newly) limited input space are clearly shown.

Disturbances can cause CVs to violate their constraints. In some cases, the controller might give up on a constraint and continue (now using a subset of the CVs). Possible solutions include; model re-identification (in the case of an inconsistency), fixing the source of the disturbance or adding a DV to account for the disturbance. Due to the fixed constraints on MVs and CVs the resulting AIS and DOS can be used to calculate the controller's ability to reject a disturbance. A measure of controller robustness can be derived from this.

Lastly, move suppression factors for MVs and give-ups for CVs are used to tune MPCs for dynamic performance. The choice of these factors are usually qualitative and according to guidelines (and validated via offline simulations). The OI (and its associated spaces) can provide guidance in determining move suppression factors and give-ups. The difficulty in controlling a CV at a given constraint can be derived from the OI. How far a CV constraint can be moved (given the MV constraints) can also be determined from the OI.

CHAPTER 3

MODEL PREDICTIVE CONTROL

With the application of the OI to MPCs shown in the previous chapter, this chapter gives an overview of MPCs in general. Attention is given to:

- general MPC theory,
- types of process models used and
- how constraints are implemented in MPCs.

Finally, background information is presented on two popular commercial MPC packages.

3.1 Model Predictive Control

Since its successful implementation in the petrochemical industry, model predictive control (MPC) has gained widespread acceptance in the processing sector (Maciejowski, 2001: 1). This has led to the development of many commercial MPC packages such as DMCplus© (Aspentech), RMPCT© (Honeywell), Connoisseur© (Invensys) and SMOC© (Shell Global Solutions) (Qin and Badgwell, 2003).

3.1.1 Nomenclature and notation

For the sake of coherence between sources a set nomenclature and notation scheme will be used. The following variables are defined; x refers to the state of the system, y to the outputs and u to the inputs. Where vectors are concerned (the MIMO case), a bold face character is used, e.g. $\mathbf{x}$ for x. Matrices are distinguished by the use of capitals.

The transpose of a matrix or a vector is indicated by a prime ($'$), e.g. $\mathbf{x}'$.

For discrete time forms, k is used to denote the sample number. For simplicity of notation, indices are used to refer to samples, i.e. x_{k+1} instead of $x(k+1)$.

3.1.2 Control theory

Model predictive control differs from other model based control techniques (such as Inverse Nyquist Array- and Internal Model Control) in its active use of predictions for future process outcomes (Maciejowski, 1989: 137). In this context, MPC further distinguishes itself from other predictive control techniques in its ability to accommodate constraints on inputs and outputs.

Maciejowski (2001: 8) summarises the control scheme of MPC in four steps; Measure, Predict, Optimise and Apply. These steps are summarized below:

1. Measure; the current outputs ($y(t)$) are measured and the error (deviation from the setpoint trajectory, $s(t)$) is calculated.

2. Predict; using the model, future outputs ($\hat{y}_k$) are calculated (over the prediction horizon, H_v).

3. Optimise (Calculate); control moves (over the control horizon, H_u) are now calculated to minimize the predicted deviation from the setpoint trajectory.

4. Apply; only the first control move is implemented, where-after this procedure is restarted.

Figure 3.1 illustrates the aforementioned steps along with some other responses. The reference trajectory, r_k, shows the ideal response a variable should follow to reach the setpoint trajectory, $s(t)$. The free response, $\hat{y}_{k\ f}$, shows the predicted values of the output if no change to the current control signal is made.

3.1.3 Objective functions

Optimal control moves are defined in terms of objective functions. These functions are often referred to as cost functions (Maciejowski, 2001: 41) as they can incorporate input and output weighting based on economic factors.

The general formulation of the unconstrained objective function is presented in equation 3.1. Modifications to this function (due to constraints) are discussed in section 3.2.3. From Rawlings and Mayne (2009: 17) and Maciejowski (2001: 41), the objective function which penalizes deviations from the setpoint trajectory as well as moves in the inputs is shown below;

$$V(\mathbf{x}_0, \mathbf{u}) = \frac{1}{2}\sum_{k=0}^{N-1}[\mathbf{x}_k' Q \mathbf{x}_k + \mathbf{u}_k' R \mathbf{u}_k] + \frac{1}{2}\mathbf{x}_N' P_f \mathbf{x}_N \tag{3.1}$$

It is clear that the objective function depends on both the state sequence and the input sequence. The current state, $\mathbf{x}_0$ is known (measured) and the subsequent states are

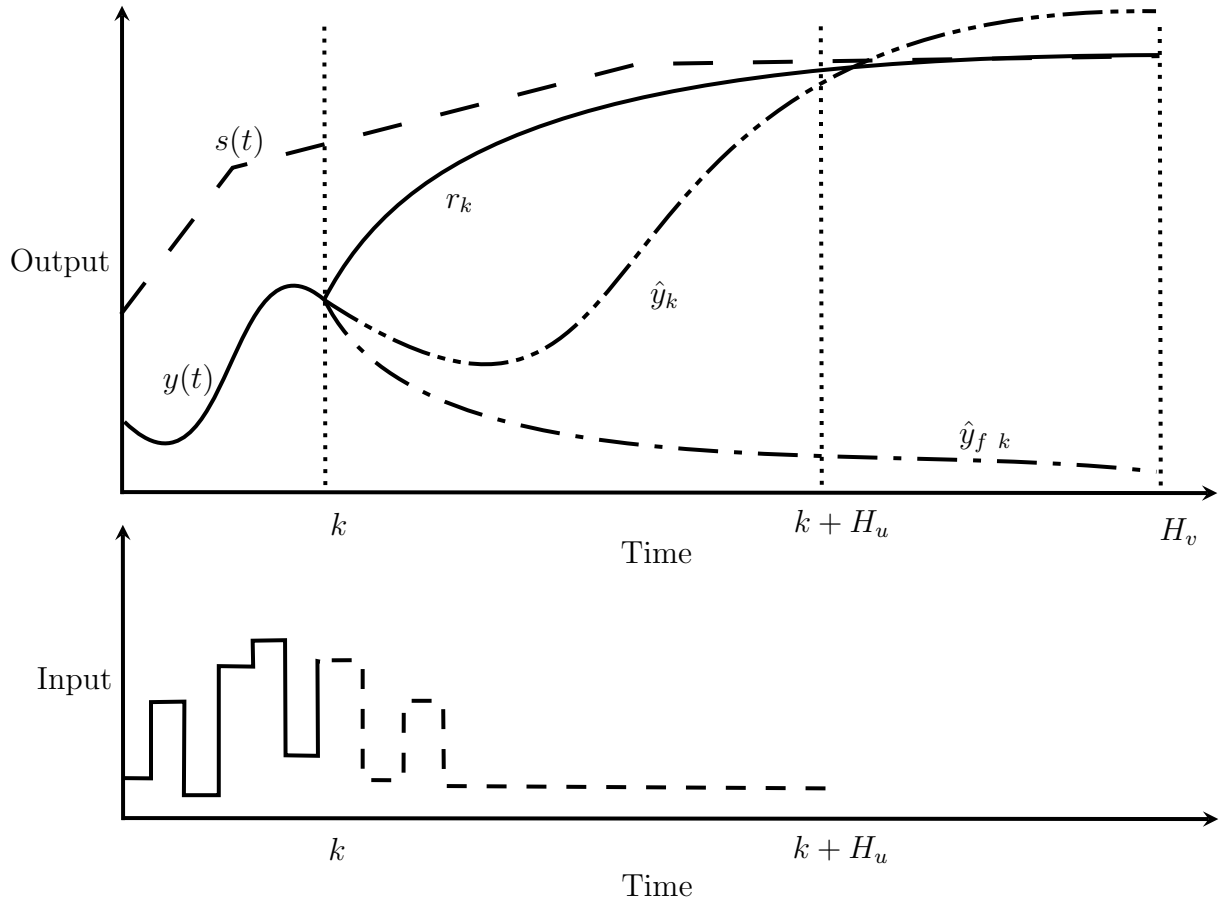

Figure 3.1: General MPC working, showing predictions on both outputs and inputs (Maciejowski, 2001: 8)

determined by the model and the input sequence. The optimal MPC control problem therefore becomes;

$$\min_{\mathbf{u}} V(\mathbf{x}_0, \mathbf{u}) \tag{3.2}$$

3.1.4 Models

The correct choice of model is one of the most important steps in the operation of MPCs (Rossiter, 2003: 17). This is due to the active use of the model in making predictions as the controller runs, and not serving merely as an analysis aid (Maciejowski, 2001: 37).

State space models

State space models are the most encountered type in literature. The general, linear, time-variant, state space model is presented in equation 3.3.

$$\begin{aligned} \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} &= A(t)\mathbf{x} + B(t)\mathbf{u} \\ \mathbf{y} &= C(t)\mathbf{x} + D(t)\mathbf{u} \\ \mathbf{x}(0) &= \mathbf{x}_0 \end{aligned} \tag{3.3}$$

Where $A(t)$, $B(t)$, $C(t)$ and $D(t)$ are appropriately sized matrices of the state space model, and $\mathbf{x}_0$ is the initial state of the system. For the time-invariant case, these matrices simply reduce to A, B, C and D. As MPC is mostly implemented at discrete time steps, the model in equation 3.3 (for the time-invariant case) translates to equation 3.4.

$$\begin{aligned} \mathbf{x}_{k+1} &= A\mathbf{x}_k + B\mathbf{u}_k \\ \mathbf{y}_k &= C\mathbf{x}_k + D\mathbf{u}_k \\ \mathbf{x}_0 &\text{ (given)} \end{aligned} \tag{3.4}$$

Step and pulse response models

Step and pulse response models, especially finite impulse response (FIR) models, were widely used in the original descriptions of MPC. There has, however, been a recent trend to move to other model types, such as transfer function models (Maciejowski, 2001: 113; Rossiter, 2003: 26).

The pulse response model can be defined using the common convolution model (Luyben, 1990: 284), as shown in equation 3.5. Where H is a matrix representing the response of an output to a unit pulse in the corresponding input.

$$\mathbf{y}(t) = \sum_{k=0}^{t} H(t-k)\mathbf{u}_k \tag{3.5}$$

Even though FIR models are intuitive they are not without shortcomings. The most prominent of these problems are that FIR models are only applicable to asymptotically stable plants, and they are only adequate if the CVs are measured outputs. Maciejowski (2001: 109) elaborates on this topic.

Transfer function models

In the Laplace domain, the input to output relationship of a generic process (as shown in figure 3.2) is represented by;

$$\bar{\mathbf{y}}(s) = G(s)\bar{\mathbf{u}}(s)$$

where G is the process matrix, and $\bar{\mathbf{u}}$ and $\bar{\mathbf{y}}$ are the process inputs and outputs (with the bars indicating deviation variables) respectively. The process matrix (G) can contain dynamic elements or, in the case of a steady-state model, only gains. Note that state ($\mathbf{x}$) information does not appear in the transfer function formulation.

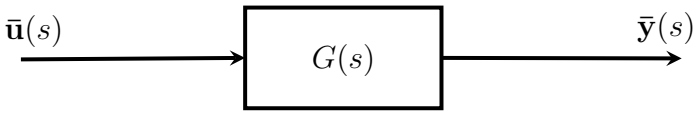

Figure 3.2: Generic process showing inputs ($\bar{\mathbf{y}}$), model (G) and outputs ($\bar{\mathbf{x}}$).

With the assumption of an initial zero state ($\mathbf{x}_0 = 0$), the state space description can be converted to a transfer function description. Taking the Laplace transform of equation 3.3 (for the time-invariant case), and rearranging, results in equation 3.6.

$$\begin{aligned} \bar{\mathbf{x}}(s) &= (sI - A)^{-1}B\bar{\mathbf{u}}(s) \\ \bar{\mathbf{y}}(s) &= (C(sI - A)^{-1}B + D)\bar{\mathbf{u}}(s) \end{aligned} \tag{3.6}$$

Other model types

Some other model types are also used in MPCs, most are however derived from or reformulations of the above mentioned types. Examples include, distributed models (Rawlings and Mayne, 2009: 4), CARIMA models and matrix fraction descriptions (Rossiter, 2003: 24,28) – with the later two stemming from transfer function descriptions.

Even though only linear models have been presented in this section, many MPCs are capable of using non-linear models. The use of linear models in MPC is, however, common practice. Important reasons for this are; no assurance of convergence of solutions and optimisation being non-trivial for non-linear models (Rossiter, 2003: 17).

3.1.5 Tuning

The tuning of MPCs is a complex task with many adjustable parameters, even in basic formulations. Most tuning is, however, based on experience or rules of thumb (Maciejowski, 2001: 188).

As far as the objective function (equation 3.1) is concerned, the most prominent tuning parameters for MPCs are the weighting matrices Q, R and P_f. These are used to enforce the relative importance of deviations in both the inputs and outputs.

The weighting matrices are by no means the only tuning parameters. Additional parameters include the horizons used for predictions, the reference trajectory and the auxiliary models used (e.g. disturbance models). Chapter 7 of Maciejowski (2001) covers the tuning of MPCs in some detail.

3.2 Constraints in MPC

The implementation of constraints is probably the most important selling point of MPCs. When used in conjunction with a steady-state optimiser, MPCs are able to operate with more CVs than MVs (Vinson, 2000). When this is the case, additional degrees of freedom are obtained by controlling CVs within ranges rather than on setpoints.

3.2.1 Constraint types

Different types of constraints are present in the MPC algorithm. The following section lists these constraints and give a short description of their function and properties.

Control constraints

Often called "hard" constraints, these constraints are used for control (as opposed to optimisation) and are never violated when control moves are calculated. Control constraints exist for both outputs and inputs, and typically sort into two categories:

- The constraints on inputs are typically physical constraints. Examples are valve saturation limits and maximum heat duties. Control constraints on outputs are usually concerned with safety and damage to equipment. Tank levels are an example of this.
- Another class of control constraints (usually on outputs) are concerned with product quality and are thus operational in nature.

To add robustness to the control solution, some commercial MPCs use constraints that regulate dynamic behaviour. These are typically a changing constraint at each solution interval. Examples of these so called "funnels" are given in section 3.3.1.

Optimisation constraints

The constraints implemented by the optimisers in MPC are often called "soft" constraints. These constraints are typically determined by operational and economical factors. These constraints are usually implemented as steady-state constraints which represent the optimal solutions of an external cost function. The limits of these constraints are contained within the control constraints mentioned above.

As these constraints are not determined by the control objective function, situations exist where they are violated in an attempt to ensure feasibility of the MPC solution. This is discussed further in section 3.2.4.

3.2.2 Constraint formulation

Constraints on the inputs, outputs or states of a system can be represented as sets of linear inequalities. For the most general case, they can be expressed as follows:

$$\begin{aligned} A_u \mathbf{u}_k &\leq \mathbf{b}_u \\ A_y \mathbf{y}_k &\leq \mathbf{b}_y \\ A_x \mathbf{x}_k &\leq \mathbf{b}_x \end{aligned} \tag{3.7}$$

where A_u, A_y and A_x are coefficient matrices, and $\mathbf{b}_u$, $\mathbf{b}_y$ and $\mathbf{b}_x$ the half-space offsets. For the case where only upper and lower bounds on variables exist, A_u and $\mathbf{b}_u$ reduce to:

$$A_u = \begin{bmatrix} I \\ -I \end{bmatrix} \qquad \mathbf{b}_u = \begin{bmatrix} \mathbf{u}_k^{max} \\ -\mathbf{u}_k^{min} \end{bmatrix}$$

where $\mathbf{u}_k^{max}$ and $\mathbf{u}_k^{min}$ correspond to the upper and lower limits of $\mathbf{u}_k$ respectively (Rawlings and Mayne, 2009: 6). The coefficient matrices and offsets are reduced similarly for the outputs and the states.

For constraints on rate of change of inputs, the formulation is similar to equation 3.7. The change in $\mathbf{u}_k$ during a sampling instance is now considered thus:

$$\begin{aligned} A_{\Delta u} \Delta \mathbf{u}_k &\leq \mathbf{b}_{\Delta u} \\ \text{with} \quad \Delta \mathbf{u}_k &= \mathbf{u}_{k+1} - \mathbf{u}_k \end{aligned}$$

3.2.3 Quadratic objective functions

When constraints are present, the general objective function (equation 3.1), now subject to equation 3.7 can be reformulated as a quadratic programming (QP) problem. This makes the problem convex and has favourable consequences for the solution of the optimisation.

The mathematical description of the QP problem is omitted from this dissertation. Maciejowski (2001: 81-83) as well as Rawlings and Mayne (2009: 489-490) can be consulted for the complete reformulation.

3.2.4 Effect on solutions

The reformulation of the objective function and constraints into a QP problem – and the subsequent convexity of the optimisation – affects the solution in numerous ways. Maciejowski (2001: 83) lists some of the solution properties obtained; namely that the termination of the QP problem is guaranteed and that computation time can be calculated.

Constrained optimisation does however suffer from the problem of possible infeasibilities. For commercial MPCs the solvers are usually custom built to provide a "back-up" solution in the case of infeasibilities. This back-up solution usually consists of modifying constraints (hence the term "constraint handling"). The most common methods of internal constraint handling are listed below:

- Constraint softening, where constraints are prioritized and then relaxed (or removed) until feasibility is obtained (Rossiter, 2003: 160).
- Using constraint windows which define a horizon over which constraints are enforced. Feasibility can now be obtained by changing the start and end-time of this window (Maciejowski, 2001: 281-282).
- Back- offs and borders can be defined, effectively changing all constraints into "soft" constraints. Due to the conservative approach, setting the back -off amounts therefore becomes a feasibility assurance and performance compromise (Rossiter, 2003: 161; Maciejowski, 2001: 282).

3.3 Commercial MPCs

This section gives a more detailed overview of two popular commercial MPC packages. Appendix A contains screenshots of the interfaces mentioned in the sections that follow.

3.3.1 Honeywell RMPCT

Honeywell's MPC controller, RMPCT (Robust Model Predictive Control Technology), forms part of their advanced process control suite, Profit Suite. A brief overview of the controller and optimiser is given below. Attention is given to the controller internals as well as the interface used to build such a controller. The information presented in this section is taken from Honeywell International Inc. (2007b), Honeywell International Inc. (2007c) and Honeywell International Inc. (2007a) unless stated otherwise.

Models

For the purpose of predictions, process models usually need to be identified first – typically via step testing. Honeywell International Inc. (2007a) lists the types of models which are supported by their Identifier and elaborates on their specific application. These model types include:

- FIR,
- PEM (the structure of which supports FIR, ARX, ARMA, ARMAX, ARIMA(X), ARARMAX, BJ and OE models) and
- Laplace domain parametric models.

The final model, however, is saved in the Laplace domain. Discrete models are converted to the s domain and has a final structure as shown in equation 3.8.

$$G(s) = \frac{k(b_{n-1}s^{n-1} + \cdots b_1 s + 1)e^{-ds}}{s(a_n s^n + a_{n-1}s^{n-1} + \cdots a_1 s + 1)} \tag{3.8}$$

The leading s in the denominator of equation 3.8 indicates an integrator in any of the sub-processes.

If a third-party model exists, then the model converter in Profit Suite can be used to convert it into a suitable form.

Constraints

For MVs, the implementation of control constraints are in the form of high and low limits, and movement limits. The high and low limits of MVs are never violated. Prioritisation and move suppression of MVs are done by means of weighting factors. The movement load (L_Δ) is spread across the MVs as per equation 3.9.

$$L_\Delta = \min \sum_j \Delta u_j^2 \times w_j^2 \tag{3.9}$$

where w is the weight and j the index of the MV.

CV control constraints are also specified as high and low limits. In the event of negative degrees of freedom, constraints are prioritised by means of a give-up factor called the engineering unit (EU) give up. Similar to equation 3.9 the cumulative weighted error on the CVs (ϵ) is minimised as per equation 3.10 with the weight (w) defined as shown.

$$\epsilon = \min \sum_i w_i^2 \times e_i^2 \tag{3.10}$$

$$w = \frac{1}{(\text{CV scaling factor})_i \sqrt{(\text{EU give up})_i}}$$

where e is the error and i the CV index

The optimisation constraints are defined as being a specified amount within the high or low control limits. These limits are not used for control.

For the constraining of dynamic response, funnels are implemented. Predefined funnel types can be chosen from:

- Type 0, which is an automatically generated funnel,
- Type 1, defined as a third of the Type 0 funnel pinched using the feedback performance ratio (FPR), and
- Type 2, which is similar to Type 1 but pinched with the decouple ratio (Type 2 is therefore FPR independent).

Figure 3.3 illustrates the concept of pinching for the RMPCT funnels. The definitions of the FPR and the decouple ratio are contained in Honeywell International Inc. (2007b) but are omitted from this dissertation.

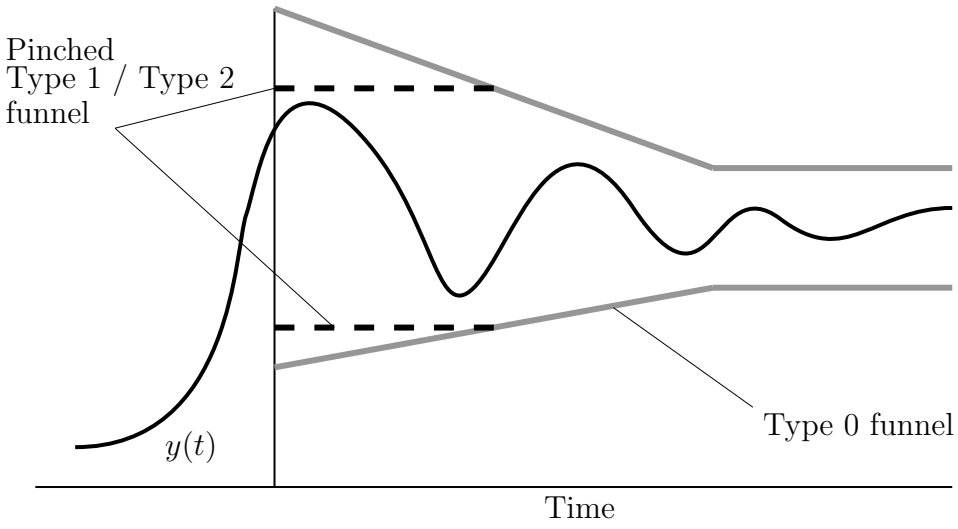

Figure 3.3: RMPCT funnel implementation showing pinched funnels with dashed lines.

Interfaces

Profit Design Studio (figure A.1) is used for model identification. From this interface the process model can be extracted. All the constraints (as discussed in the preceding section) can be added using this interface, where after the controller is built for on-line use from the Runtime Studio interface (figure A.2).

The operator interface, Profit Suite Operator Station (figure A.3), allows for constraint changes during plant operation. As far as feasibility of constraints are concerned, these constraints are only checked to be within the Engineering limits (an additional constraint present in this interface). A gain matrix (the steady-state model) can easily be obtained from this interface.

3.3.2 AspenTech DMCplus

Aspen Technologies' MPC controller, DMCplus (Dynamic Matrix Control Plus), forms part of their advanced process control suite, aspenONE. The overview presented in this section is taken from Aspen Technologies Inc. (2009) unless stated otherwise.

Models

Model identification is done with DMCplus Model or with SmartStep which enables model identification while still maintaining control of a plant. Models are stored as FIR models and conversion to and from other model types are not supported. Extracting a steady-state gain matrix is possible as the gains are displayed during the modelling phase.

Constraints

As with RMPCT, constraints on the CVs and MVs are specified as high and low limits. In addition to the constraints, equal concern errors (ECEs) are used to calculate the optimal move plan and emphasise the relative importance of CVs.

Three types of constraints are used:

- Validity limits which describe limits on measured variables and the values they can attain. These are the outermost limits and serve only to identify faulty measurements.
- Engineering limits are the hard limits in which the process is operated and are not violated in a solution. The engineering limits lie within the validity limits.
- Operator limits are used for optimisation, with successful solutions keeping within these limits. These are the innermost limits, contained within the engineering limits.

For constraint handling, constraints are ranked using rank groups. Lower ranked constraints are considered "hard" whereas higher ranking constraints are relaxed until steady-state feasibility is achieved. Optimisation constraints are ranked at 1000 and constraints that are ignored are ranked at 9999.

The constraint handling of integrating variables is done by utilising ramp rates. Constraints on the variable are generated using a factor of the time to steady-state and the upper and lower bound. This limits the rate of change of a variable. Figure 3.4 shows the constraints generated using ramp rates. The 'inverted funnel' becomes sharper when the full time to steady-state (t_{ss}) is used as opposed to a fraction thereof.

Interfaces

The DMCplus Model (figure A.4) interface is used to obtain models from data. Obtaining a gain matrix is possible from this interface. Additions to the model matrix are also made from this interface.

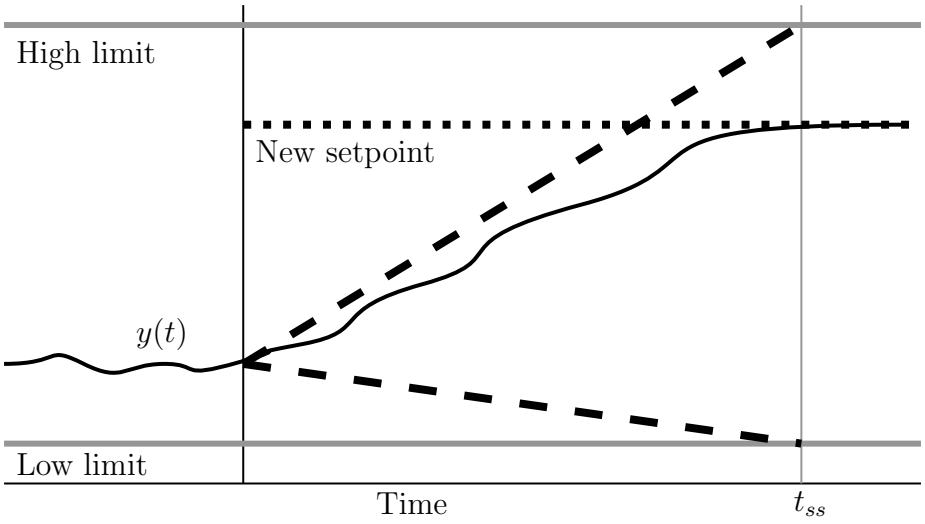

Figure 3.4: Ramp rates creating dynamic constraints on integrating variables after setpoint change.

The controller is built using the DMCplus Build interface (figure A.5). Models obtained in the previous step are implemented and a configuration file for the controller is generated.

The Production Control Web Interface (figure A.6) is used by operators to interact with the process. This interface is used to change constraints on inputs and outputs during process operation.

CHAPTER 4

SYSTEMATIC CONSTRAINT HANDLING

This chapter presents the proposed method of constraint handling based on the work of Vinson (2000). The method comprises:

- constraint checking for feasibility,
- quantification of operability clamping after constraint changes,
- the implementation of linear constraint in the commercial MPC structure, and
- constraint set fitting for reduction of the control problem.

A section is also devoted to disambiguating the language used to refer to constraints based on their sources.

4.1 Assumptions

Certain assumptions are made about the systems investigated. This serves to define the scope of the project and aids in the mathematical rigour of the methods.

- Steady-state models are used. These models result in real matrices and allow for strictly linear transformations of spaces.
- Only linear constraints are considered. This allows for half-space geometry to be used and ensures convexity of intersections (along with the assumption of steady-state gain models).
- The convexity of all spaces is assumed. This allows the use of minimum vertex descriptions.

Only a subset of the available model types are presented in this dissertation. It is by no means implied that rigorous mathematical methods are only possible for this given subset.

Note that when reference is made to a "constraint set", all the half-spaces comprising that set as well as the feasible region (of the set) are implied.

Finally, it should also be noted that the concepts presented in this document apply to higher order systems, even though only 2-dimensional examples are presented for ease of illustration. Programatically, the number of dimensions are only constrained by the algorithm used to calculate the convex hull (`qhull`). However, the design of the code allows for `qhull` to be replaced by another algorithm with only minor changes.

4.2 Constraint checking

Commercial MPCs use only their set engineering limits to check the validity of constraints. Along with the engineering constraints (which represent the ultimate bounds) the model should also be used to validate constraints. This is due to the constraints in the input and output space being interconnected via the process model.

4.2.1 Feasibility

Operating regions, such as the DOS, are usually specified using external requirements on the inputs or outputs. The validity of such a space should be checked against the attainable regions of the corresponding inputs or outputs. In this respect, determining the Operability Index provides a good measure of how valid a specified operating region is.

The same argument for feasible constraints apply to setpoints. The specification of setpoints should be checked to be within the attainable output space for a given available input space.

4.2.2 Constraint changes

The operator interfaces of commercial MPCs (figures A.3 and A.6) allow operators to change constraints during the operation of processes. The effects of constraint changes on their corresponding spaces are, however, neglected in commercial MPCs. The reduction or increase in size of the available input space for a change in output constraints (or vice versa) should be checked. Ideally a measure of clamping (or relaxing) of constraints should be supplied.

It is intuitive that constraint tightening in the output space has a corresponding tightening effect in the input space, but to which extent isn't immediately clear. To quantify the tightening of the inputs, the OI of Vinson (2000) can be used, although, this has the downside of remaining at a value of 1 when the DIS is already contained within the AIS, regardless of the size change of the DIS. Another possibility is a ratio of the hypervolumes of the original attainable part of the DIS and the attainable part of the

new DIS ($DISn$). This clamping factor (C) is shown in equation 4.1.

$$C = 1 - \frac{\mu(AIS \cap DISn)}{\mu(AIS \cap DIS)} \tag{4.1}$$

where μ is a function to calculate the hypervolume of a space.

4.3 Commercial MPC interfacing

The clamping factor (C) defined in the previous section serves to guide operators when changes to constraints are made. Implementation of this number into the user interface of a commercial MPC would therefore be beneficial.

4.3.1 Constraint types

When constraints are considered, and specifically changes to them, the types of constraints present are important. This is due to the direction of change allowed by each constraint type. The following set of constraint types are proposed to disambiguate the specification of constraints. These proposed types are based on the sources of constraints.

Physical constraints represent the physical limits of the system. Examples are tank levels, valve saturation limits and maximum heat duties. Allowed changes to these constraints are usually one-sided or to the inside of a range.

Quality constraints are concerned with product quality, these constraints are typically determined by external requirements or standards. Strictly speaking, changes to these constraints are allowed in any direction. However, due to the economic (e.g. product purity) and safety (e.g. emission limits) consequences of changes, these limits are only changed by those with granted access.

Operational constraints are the innermost constraint set used by the MPC controller. These constraints are typically changed using the operator interfaces (e.g. figures A.3 and A.6) and are contained within their respective variables' physical and quality constraints.

Optimisation constraints are those constraints that are not critical for the control of the plant, but rather the performance and efficiency of the plant as determined by external cost functions. Typically these external cost functions will represent a type of saving, e.g. energy savings or product give-away reduction.

The specification of spaces, in particular the AIS and the DOS, usually consists of a combination of the constraint types mentioned above. Classifying constraints in this

unambiguous manner will further the understanding of a system and help identify which constraints can be modified to change the size or shape of a space.

Commercial MPCs do not typically distinguish between different types of constraints. The constraint types present in RMPCT (section 3.3.1) and DMCplus (section 3.3.2) are listed below and their connection to the proposed constraint types discussed.

Engineering limits are typically based on a combination of physical constraints and quality constraints. The specification of the engineering limits should ideally be a subset of the physical constraints and the AOS and AIS of the process. One method would be to specify high and low limits on the outputs determined by their maximum values in the AOS. Along with a safety factor – dependant on model uncertainty – the intersection of these spaces would provide a better set of engineering limits.

Validity limits (DMCplus) are equal to or outside of the physical constraints of a variable.

4.3.2 Linear constraints

As discussed in section 3.3, most commercial MPC packages only allow for high and low limits to be imposed on variables. For the case where constraints that are linear combinations of variables are present (e.g. $\beta_1 y_1 + \beta_2 y_2 \leq b_1$), they need to be reformatted to conform to the limited structure provided by commercial MPCs. Adding an unmeasured variable to a system by adding rows to the process matrix is a common technique used with commercial MPCs. This same technique can be used to impose linear constraints on outputs or inputs. The unmeasured variables added to the process model will have gain constants as determined by the coefficients in the linear constraint.

Considering a model G for an $n \times m$ system (n inputs, m outputs) and the linear constraints

$$\alpha \mathbf{u} \leq b_u$$
$$\beta \mathbf{y} \leq b_y$$

where α and β are rows of A_u and A_y (equation 3.7) respectively, the following applies:

- For an input constraint a row is added to the process model and the new input, u_{n+1} is defined thus:

$$u_{n+1} = \alpha \tag{4.2}$$

  Even though u_{n+1} represents an input constraint, it is strictly speaking an additional output and the number of inputs to the system model remains unchanged.

- For an output constraint a row is added to the process model and the new output, y_{m+1} is defined thus:

$$y_{m+1} = (G' \times \mathbf{1}_{m\times 1})' \cdot \beta \tag{4.3}$$

  where $\mathbf{1}_{m\times 1}$ is a column vector of 1s of length m.

A high or low limit can now simply be applied to this newly added variable. For the examples above, this is simply $u_{n+1} \leq b_u$ and $y_{m+1} \leq b_y$

4.4 Constraint set fitting

As shown by Vinson (2000) – by means of the Operability Index – a larger operating region is advantageous for control. The largest operating region (when considering the output space) is represented by the AOS and DOS intersection. Ideally all the constraints specifying that intersection (the facets of the intersection) could be implemented in an MPC which would result in the maximum operating range for the controller.

Firstly, as the dimensions of the system increase the dimensions of the polytopes describing the input and output spaces increase as well. This results in a high number of facets present in the intersections of spaces. Furthermore, as the irregularity of the spaces increase the number of facets present in the intersections will also increase. Each of these facets represent a constraint that can be added to the controller.

To add to this, the structure available for constraints in commercial MPCs require linear constraints to be expressed as unmeasured variables with high and low limits. As mentioned in the preceding section, each linear constraint will add an additional variable along with its constraints to the system.

Therefore, although ideal, implementing the full description of a highly faceted AOS and DOS intersection will greatly increase the control problem. This will have an adverse effect on the computational time of the controller. The description of the controller will also suffer, as a large number of unmeasured variables (for the purpose of imposing constraints) can make housekeeping difficult.

The fitting of a constraint set (with fewer constraints) within the AOS and DOS intersection can be used to reduce the control problem. The operating region of the smaller (fitted) constraint set will inevitably be less than that of the full intersection. The fitting procedure therefore becomes a compromise between control problem size and the available operating region for the controller.

The sections that follow formally define the fitting problem and gives attention to the some important classes of set fitting and reduction. Although the output space (AOS and DOS intersection) is used for examples and explanations the same applies to the input space.

4.4.1 Problem formulation

In mathematical terms, the problem consists of fitting the largest volume polytope (with a specified number of facets) within another polytope (with more facets than the fitted polytope). Each half-space – or the facets of a shape – represents a constraint. The constraint set can therefore be described by equation 4.4, where A is the half-plane slope matrix and $\mathbf{b}$ the offsets. Although $\mathbf{x}$ is used in equation 4.4, substitution with $\mathbf{u}$ or $\mathbf{y}$ for input or output constraints can be done without any loss of generality.

$$A\mathbf{x} \leq \mathbf{b} \tag{4.4}$$

In n dimensions, the general problem now becomes:

- given a polytope, P_o, having l facets,
- fit a polytope, P_i, with k facets (where $n+1 \leq k \leq l$) within P_o, and
- maximise the volume of P_i.

With the specification of the number of facets, the dimensions of A and $\mathbf{b}$ for both P_i and P_o are fixed. For P_o, A is an $l \times n$ matrix and $\mathbf{b}$ is a $l \times 1$ vector. For P_i, A is an $k \times n$ matrix and $\mathbf{b}$ is a $k \times 1$ vector.

The fitting of an arbitrary shape can be expressed as a minimization problem, as shown in equation 4.5.

$$\begin{aligned} \min_{A,\ \mathbf{b}} \quad & \text{-volume}_{P_i} \\ \text{s.t.} \quad & P_i \subset P_o \\ & P_i \text{ and } P_o \text{ convex} \end{aligned} \tag{4.5}$$

This seemingly trivial optimisation problem is not covered in this form in available literature. A similar problem is that of optimised diamond cutting (Viswambharan, 1998) which maximises the volume of a diamond within a rough stone. This method allows for rotation, translation and scaling, but uses a fixed shape for the diamond (i.e. all aspect ratios of the fitted shape stays constant). Another well considered problem in literature is the packing problem, which involves packing the maximum number of shapes in a given shape (or container). Again, the shapes considered in this problem are of fixed dimensions (and aspect ratios), but the largest dissimilarity is that a number of shapes are fitted, not just a single one.

4.4.2 Set reduction

Implementing all the constraints of the full AOS and DOS intersection does not require any fitting to be done. The linear constraints only need to be converted to the high and low limit structure (section 4.3.2).

The fitting of constraint sets will decrease the operational space of the controller, but also decrease the size of the control problem. The extent to which the control problem will be decreased depends on the number of constraints fitted and the type of fit.

Intermediate fits

As per equation 4.5, a progressively lower number of constraints can be fitted within the intersection as arbitrary shapes. The closer the number of constraints in fitted shape is to that of the original intersection, the larger the operational space of the controller will be.

To reduce the number of additional variables that need to be added to implement linear constraints, parallel sets can be used. From equations 4.2 and 4.3 it is clear that the gains of variables added to the process model are only dependent on the half-space slope and the process model. Therefore, parallel constraints only require one variable to be added to the process model.

Sets of parallel constraints can therefore be fitted, which will require one additional variable to be added to the process model per pair. As the slopes of the parallel constraints are not fixed (for the minimisation problem) this will be referred to as free parallel fitting. Enforcing these parallel constraints on the minimisation problem will result in a lower operational space for the controller when compared to an arbitrary shape with the same number of constraints.

The fitting of free parallel sets is presented as an option here, but not covered in this project.

High and low limits

The most common method of specifying constraints for commercial MPCs is using high and low limits. Fitting the maximal volume set of high and low constraints within an intersection is a subset of free parallel fitting and has the following advantages: fitting results are directly applicable to commercial MPC packages, and no additional variables need to be defined.

The disadvantage of this method is a further decrease in the operational space of the controller when compared to free parallel fitting.

For the fitting of a rectangular constraint set (high and low limits), equation 4.5 reduces to equation 4.6. In this description $\mathbf{b}$ is the only input variable (of size $2n \times 1$)

and I is an $n \times n$ identity matrix.

$$\begin{aligned} \min_{\mathbf{b}} \quad & \text{-volume}_{P_i} \\ \text{s.t.} \quad & A = \begin{pmatrix} I \\ -I \end{pmatrix} \\ & P_i \subset P_o \\ & P_i \text{ and } P_o \text{ convex} \end{aligned} \tag{4.6}$$

where n is the number of dimensions (variables) present in the initial set.

Minimal fit

A minimum value exists for the number of constraints that can be fitted into an n-polytope. This is due to the condition that the resulting set needs to be fully bounded and have a finite, non-zero volume. For n dimensions, n points lie in a plane and result in a zero volume. Therefore, at least $n + 1$ points are needed to form a n-polytopewith a finite volume, with the resulting polytope being a simple polytope. With the number of vertices at $n + 1$, the minimum number of facets can be calculated as $n + 1$ (Barnette, 1971). Therefore, the smallest number of constraints that can be fitted into a set is $n + 1$, where n is the number of dimensions (variables) of the initial set.

4.4.3 Fitting implementation

The implementation of the fitting procedure is discussed in the code manual and program listing accompanying this document. Therein, attention is also given to different problem formulations and the advantages and disadvantages thereof.

CHAPTER 5

CASE STUDIES

The case studies investigated in this project are presented in this chapter. Details are given on design, models and operating conditions of the systems.

5.1 Case studies

To test the method of systematic constraint handling proposed in chapter 4 the following physical systems were investigated. Both rigs are from the Process Modelling and Control laboratory of the University of Pretoria.

5.1.1 Level and flow rig

System description

The level and flow rig consists of two control valves, a measured flow (to one control valve) and a measured level. Figure 5.1 shows a photograph of the rig along with its process flow diagram.

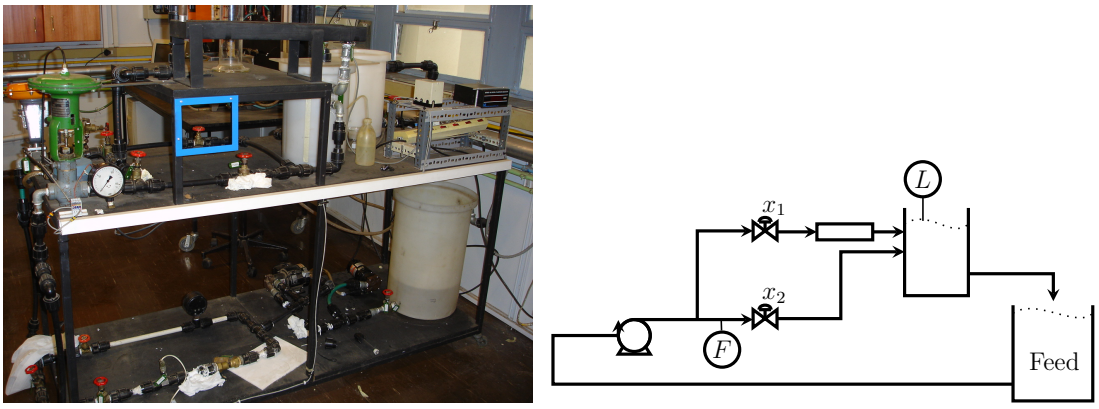

Figure 5.1: Level and flow rig photograph (left) and process flow diagram (right).

Process model

This rig is modelled as a 2×2 system. The MVs are the milliamp signals sent to the valves (x_1 & x_2) and correspond to the fractional openings of the two valves. The CVs are the flow (F) and the level (L). The steady-state gain matrix of this process (G_{fl}) is shown in equation 5.1. Note that action of valve 1 is air-to-open and valve 2 is air-to-close valve, which explains the signs of the gains in the first column of G_{fl}.

$$G_{fl} = \begin{pmatrix} -0.0476 & -0.0498 \\ 0.0111 & -0.0604 \end{pmatrix} \tag{5.1}$$

Operating conditions

The operating conditions for this rig are shown in table 5.1 The nominal operating point (output space) is; 1.322 gpm (F) and 16.4048 cm (L).

Variable		Operational constraints		Physical constraints	
		Low	High	Low	High
Inputs	x_1(mA)	4	20	4	20
	x_2(mA)	4	20	4	20
Outputs	F (gpm)	1.2	1.5	0	1.9
	L (cm)	15	18	6	25

Table 5.1: Operating conditions of level and flow rig.

5.1.2 Laboratory distillation column

System description

The second case study investigated is a 10-plate distillation column. The column is run as a closed system with bottoms and distillate being mixed and fed back into the column. Figure 5.2 shows a photograph of the column along with its process flow diagram.

System model

The process model has been reduced to a 2×2 matrix. Reflux flowrate (R, expressed as a milliamp value sent to the valve) and the setpoint of plate 10's temperature ($T_{10\ sp}$) are the MVs. The temperatures of plate 1 and 8 (T_1 & T_8) are the CVs. Figure 5.3 shows a graphic representation of the column model and equation 5.2 gives the values for the steady-state gain model. The nominal operating point (output space) is; 68 ℃ (T_1) and

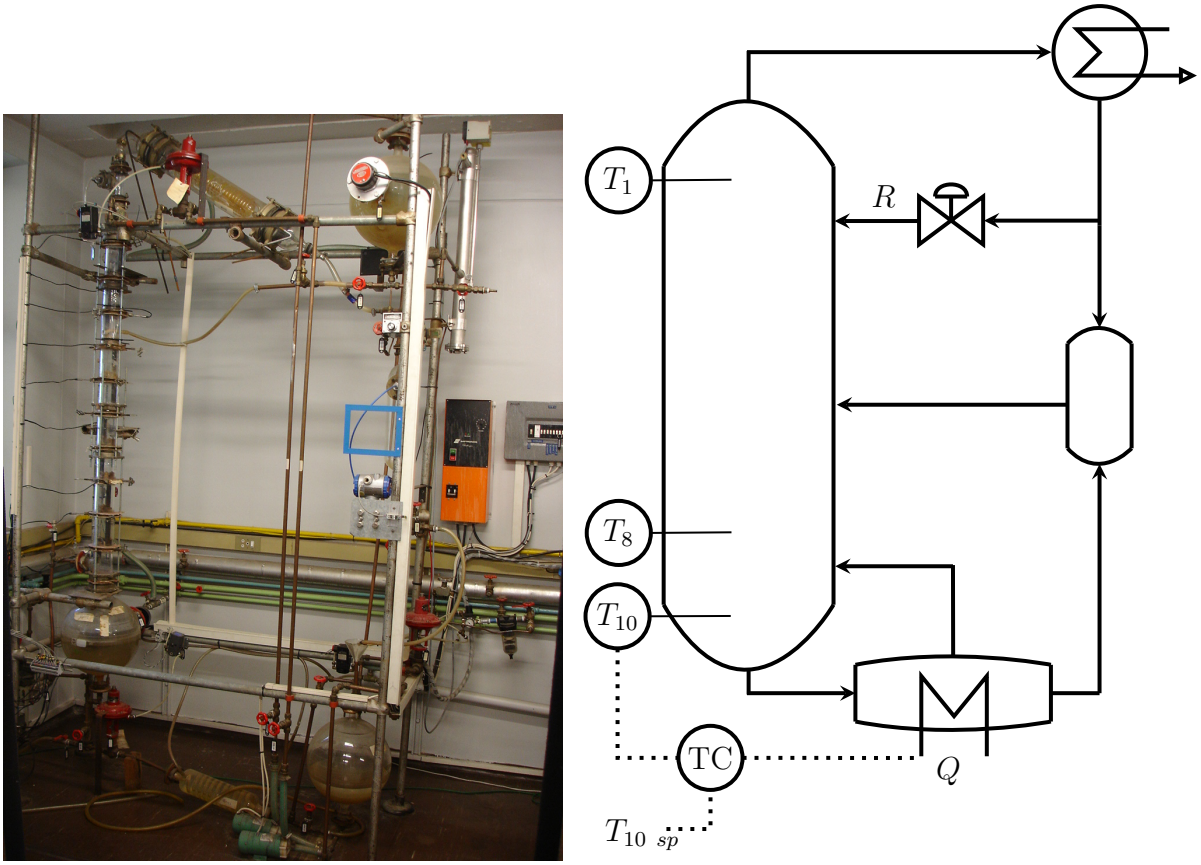

Figure 5.2: Laboratory distillation column photograph (left) and flow diagram (right).

78 ℃ (T_8).

$$G_{col} = \begin{pmatrix} -0.0575 & 0.96 \\ -0.146 & 0.518 \end{pmatrix} \tag{5.2}$$

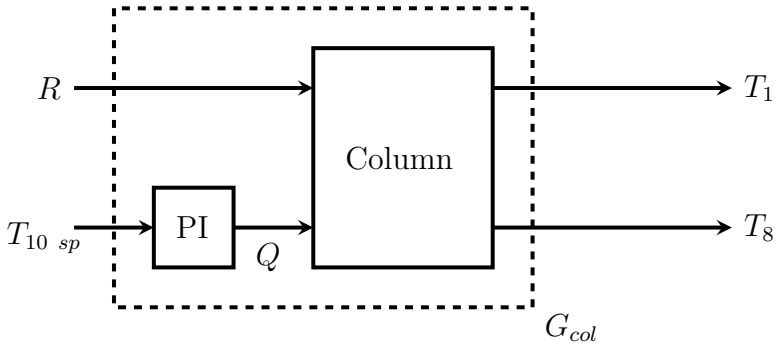

Figure 5.3: Column model showing internal PI controller.

The PI controller between $T_{10\ sp}$ and Q is contained in the process' baselayer. This was added as a safety measure, should the advanced control layer fail.

Operating conditions

Table 5.2 shows the operating conditions of the distillation column.

Variable		Operational constraints		Physical constraints	
		Low	High	Low	High
Inputs	R (mA)	11	15	4	20
	$T_{10\ sp}$ (℃)	78	82	0	90
Outputs	T_1 (℃)	66	68	15	68
	T_8 (℃)	78	82	25	90

Table 5.2: Operating conditions of distillation column.

CHAPTER 6

RESULTS AND DISCUSSION

The results of this project are summarized and discussed in this chapter. Implementation of the method to the case studies is illustrated. General results concerning the constraint set fitting are also discussed. A final section is devoted to discussing the rationale behind future expansions of constraint set fitting.

6.1 Case studies

The proposed method of systematic constraint handling is applied to the case studies. For ease of graphical representation, only 2×2 systems are evaluated. It should however be noted that the proposed method also applies to higher order systems – with the current exception of arbitrary constraint set fitting as discussed in section 6.2.2.

6.1.1 Level and flow rig

Input and Output spaces

The input and output spaces for the level and flow rig are shown in figure 6.1. These spaces are calculated from the system's operating conditions (column 5.1) and process model (equation 5.1), and shifted with the nominal operating point. From the intersection of the AOS and the DOS, the OI is calculated as 0.338.

Set fitting

It is clear that the level range expectations of this process are too ambitious. Decreasing these limits will not affect the control adversely, as the model suggests that these upper and lower level limits are not attainable. Figure 6.2 shows the fitting of upper and lower operational constraints within the intersection of the AOS and the DOS. The dark box represents the largest operating region (described only by high and low limits on the outputs) which are within the original DOS and the AOS. Compared to the DOS as

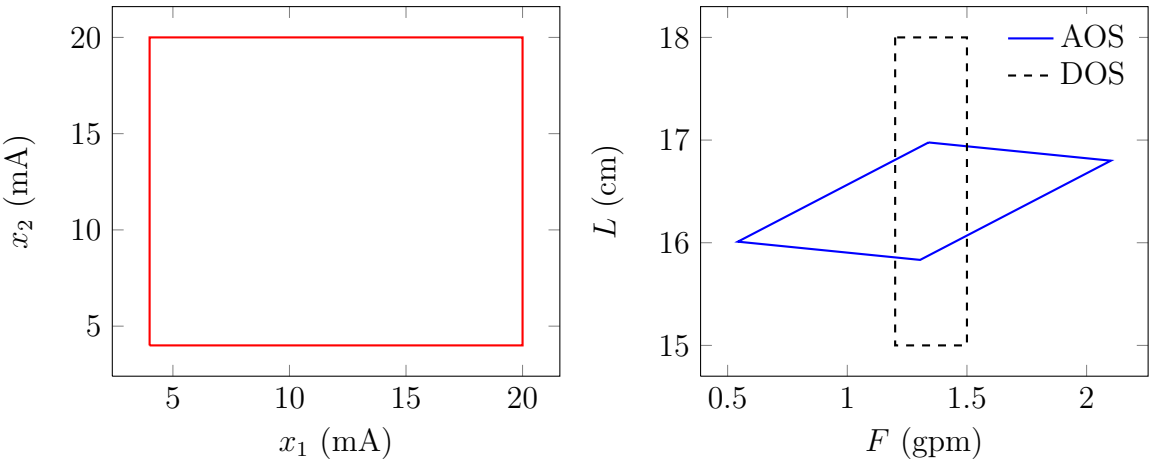

Figure 6.1: AIS (left), AOS and DOS (right) for the level and flow rig.

operational constraints, this procedure increases the OI to 1 and presents tighter bounds that are all feasible.

Constraint changes

As an example of constraint tightening, the newly fitted upper and lower operational constraints of figure 6.2 are considered and the effect of this constraint change in the input space investigated. Figure 6.3 shows the original AIS and DIS, along with the new DIS (the dark dashed box) that corresponds to the changed constraints. To quantify the tightening of the inputs, the proposed equation 4.1 is used. This indicates that the attainable movement on the inputs has effectively been tightened by 27%.

The unattainable limits of the DOS are just as clear in the input space (DIS). This figure also serves to confirm that the newly fitted constraints on the outputs correspond to a fully attainable region in the input space.

Constraint types

As mentioned in section 3.3, the only constraint validation done by both RMPCT and DMCplus is checking whether the operational constraints are within the engineering constraints. Figure 6.4 shows the AOS and the physical constraints of the system (which are typically used to specify the engineering constraints). It is clear that with only checking against physical constraints (and disregarding the system model), specification of operational constraints that are completely unattainable is possible.

Figure 6.4 also shows the fitting of a revised set of engineering limits with a safety factor of 20%. This new set of engineering limits represents the attainable region of the process as well as the physical constraints of the system. The risk of specifying operational constraints that are not attainable has also been reduced.

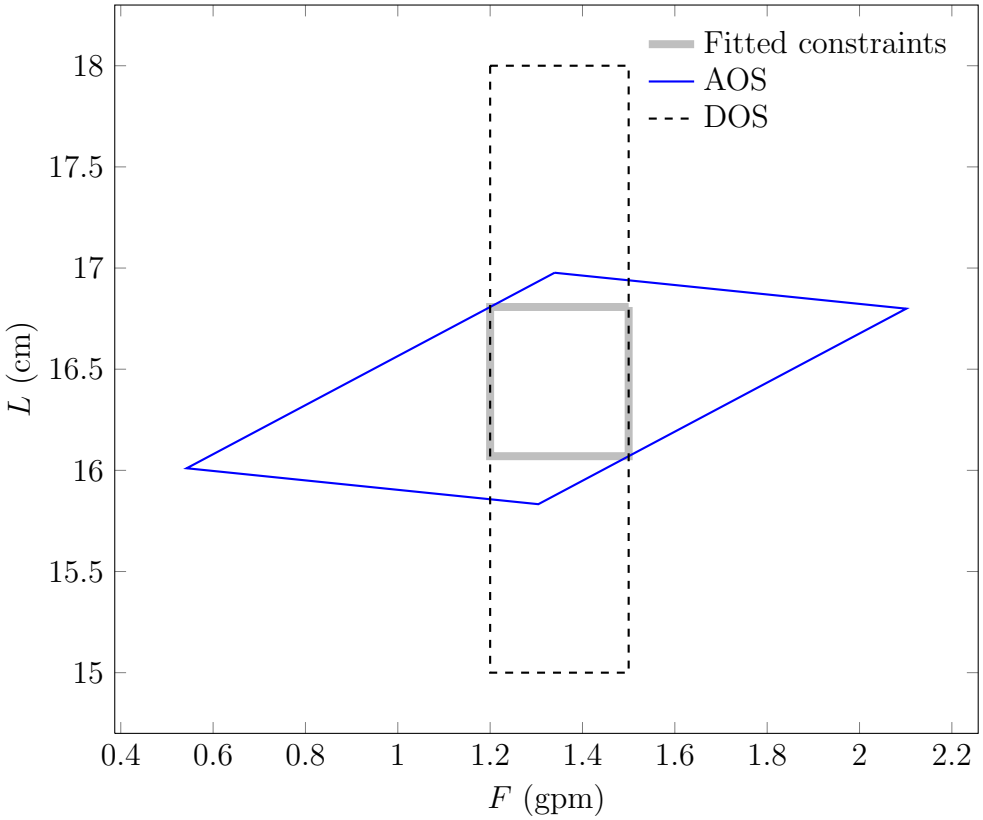

Figure 6.2: Fitted high and low constraints in the AOS and DOS intersection.

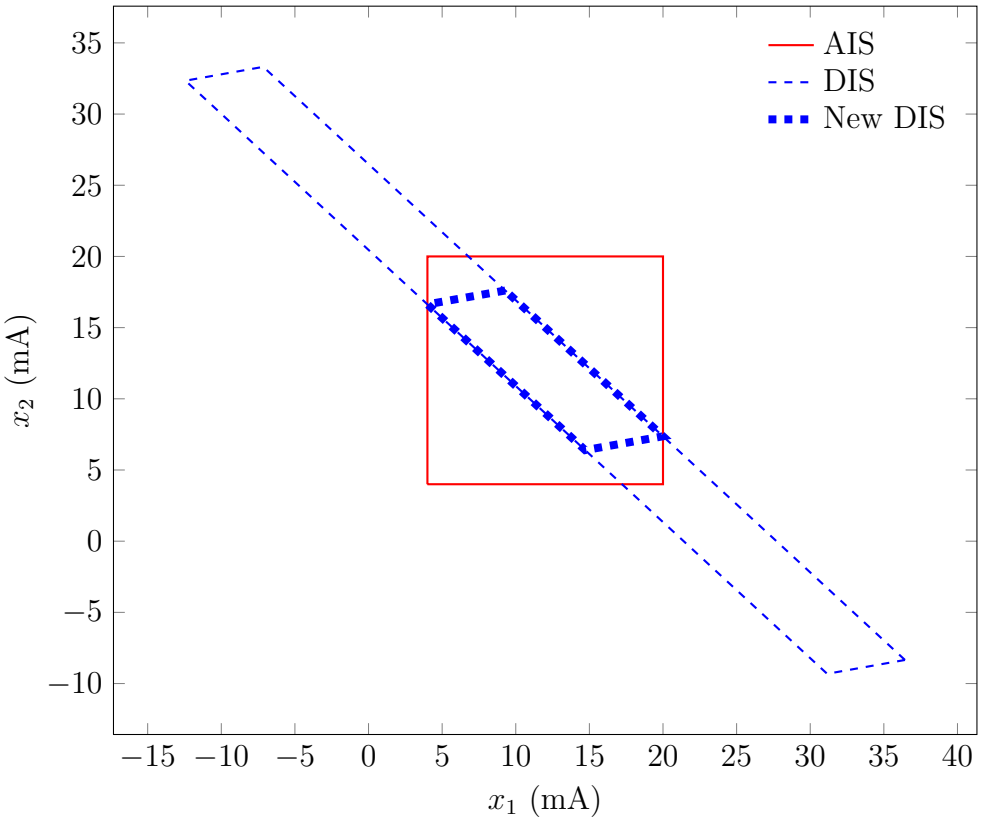

Figure 6.3: Newly fitted constraint set shows feasible input constraints as opposed to those of the original DOS.

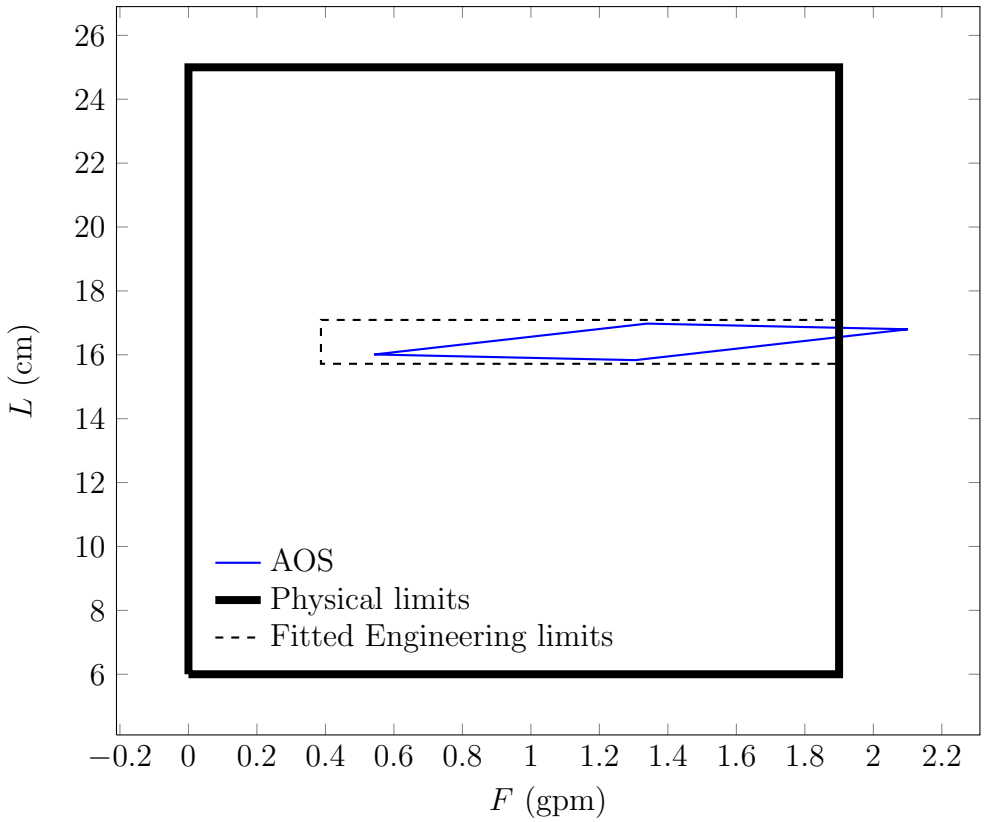

Figure 6.4: Physical constraint region, AOS and revised engineering limits (with a 20% safety factor) for the level and flow rig.

6.1.2 Laboratory distillation column

Input and Output spaces

From the data in table 5.2, equation 5.2 and the column's nominal operating point the AIS and AOS of the laboratory distillation column can be generated as shown in figure 6.5. The operating constraints for R are used to construct the AIS as values outside of this range (although possible) cause the validity of the linear model to diminish. The DOS, as described by the operating range in table 5.2, is also shown.

Figure 6.6 focuses on the intersection of the AOS and the DOS. The calculated OI is 0.006 which confirms that only a very small operating region within the DOS is attainable. It is also clear that the upper limit of 82 ℃ on T_8 is unrealistic as the maximum value of T_8 (in the operating region) is only 78.2 ℃.

Set fitting

Figure 6.7 focuses even closer on the AOS and DOS intersection. The constraints representing the intersection (dark triangle) and a fitted set of high and low limits (dark dashed box) are also shown. The fitted limits within the intersection shows that the operating region is essentially an operating point with temperatures only having a span of

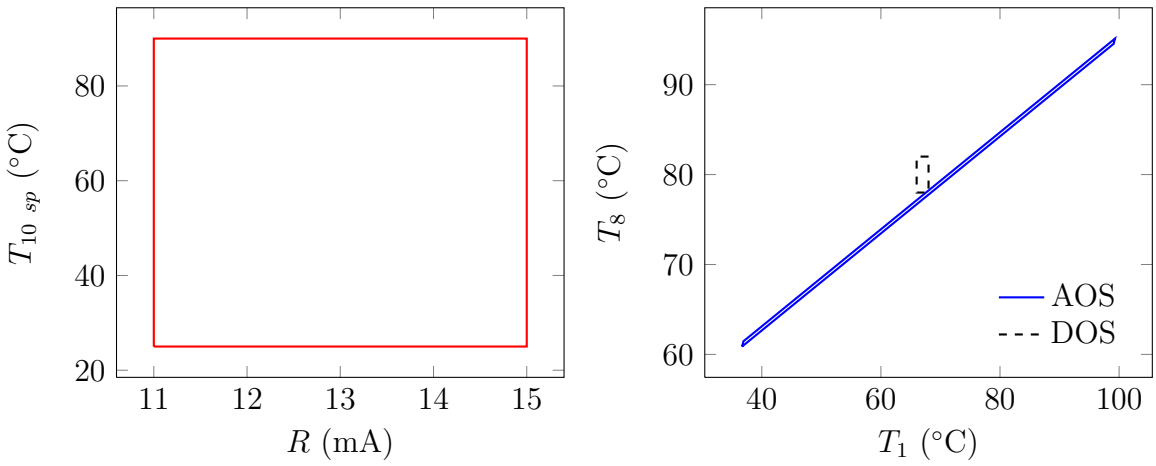

Figure 6.5: AIS (left), AOS and DOS (right) of the laboratory distillation column.

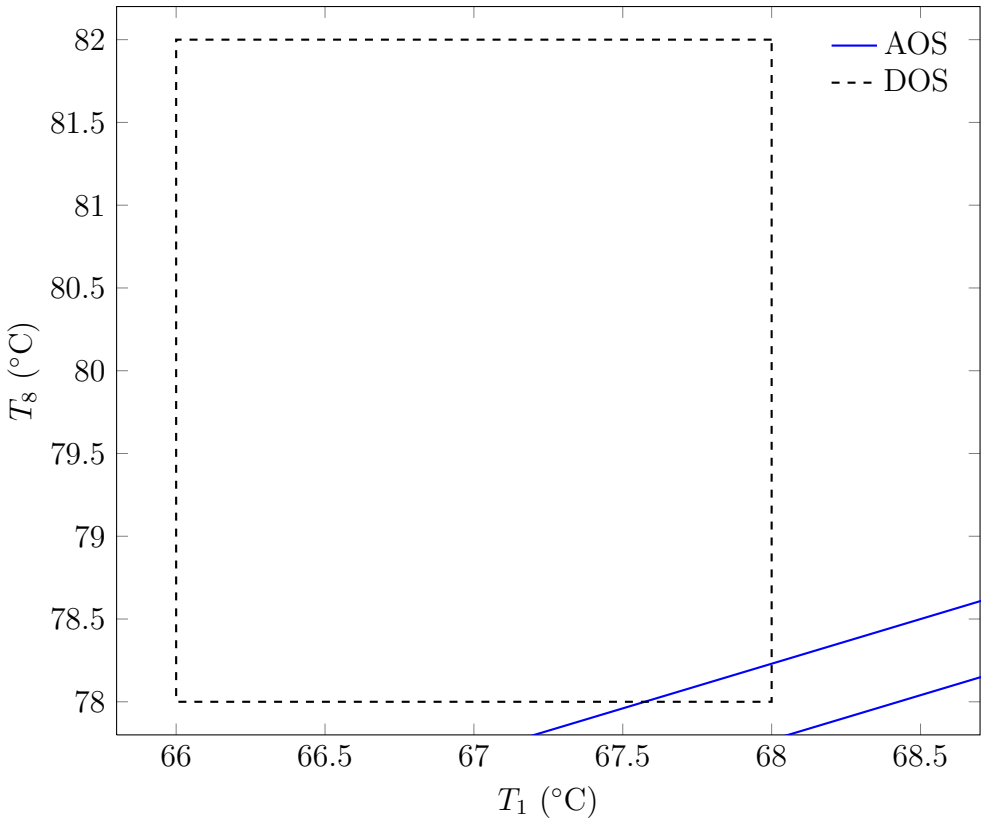

Figure 6.6: The AOS and DOS intersection shows a very small operating region.

about 0.1 ℃. In practice the DOS would be adjusted or the model re-evaluated, but for the sake of illustrating the outputs of the method, the DOS will be kept as is.

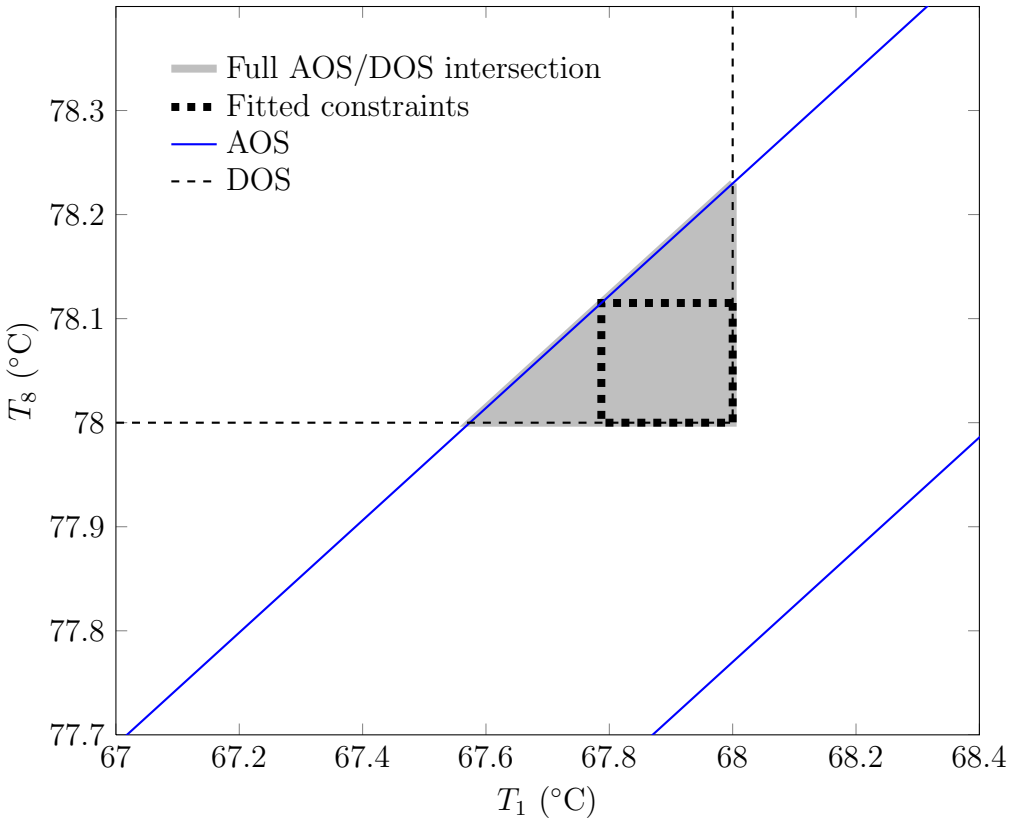

Figure 6.7: Intersection of AOS and DOS with fitted high and low limits for the column tray temperatures.

If constraints specifying the full intersection in figure 6.7 are used as operational constraints, an additional variable needs to be added to the system (along with its associated high and low limits). Using the fitted box constraints as operational constraints only require the high and low limits of the existing variables to be changed. The full intersection does, however, have the advantage of a 200% larger operating region at the cost of a larger control problem.

Constraint reformatting

To use the constraints describing the whole intersection in figure 6.7, the set needs to be reformatted to be compatible with commercial MPC packages, as they only accept high and low limits.

An additional variable needs to be added to the process model and the diagonal constraint ($0.88T_8 - 0.47T_1 \leq 36.56$ ℃) needs to be expressed as a high and low limit on this variable.

Naming the new output y_3 and augmenting the process model matrix as per equation 4.3 results in equation 6.1. The diagonal (linear) constraint can now be expressed as

$y_3 \leq 36.56$ ℃.

$$G_{col-new} = \begin{pmatrix} -0.0575 & 0.96 \\ -0.146 & 0.518 \\ 0.0956 & 1.30 \end{pmatrix} \tag{6.1}$$

6.1.3 MPC interfacing

The operator interfaces for RMPCT and DMCplus will be used to enter the operational constraints determined from the method. For RMPCT (figure A.3) the shaded columns labeled "Low Limit" and "High Limit" are used. For DMCplus (figure A.6) the column labeled "Lower Limit" and "Upper Limit" are used.

The revised engineering limits for the level and flow rig would be added to the controller via the building interfaces (figures A.2 and A.5).

The modelling interfaces (figures A.1 and A.4) are used to augment the model for the addition of unmeasured variables, as determined for the distillation column.

6.2 Constraint set fitting

The fitting of constraint sets, which form a large part of this dissertation, is discussed separately. Observations made while studying the fitting of constraint sets, which could lead to future research is discussed in section 6.2.3.

6.2.1 Solution times

The use of constrained, gradient-based solvers proved to be significantly faster than unconstrained solvers. Inconsistent gradient information does, however, hamper their implementation.

Figure 6.8 compares the solution times of the constrained, gradient based solver (henceforth referred to as SLSQP) and the unconstrained solver (henceforth referred to as simplex) for rectangular set fitting. Arbitrary sets with three constraints on two variables were generated. A high and low constraint set was fitted within the attainable region of the initial three constraints. The data in figure 6.8 is expressed as the fraction of tests that completed successfully within the given number of function evaluations.

Fitting a constraint set of an arbitrary size was unsuccessful using constrained, gradient-based solvers. Possible reasons for this are inconsistent gradient information regarding the position of vertices outside the initial set and the superfluous degrees of freedom present in the problem formulation. The sources of these problems are discussed in the accompanying code manual and program listing.

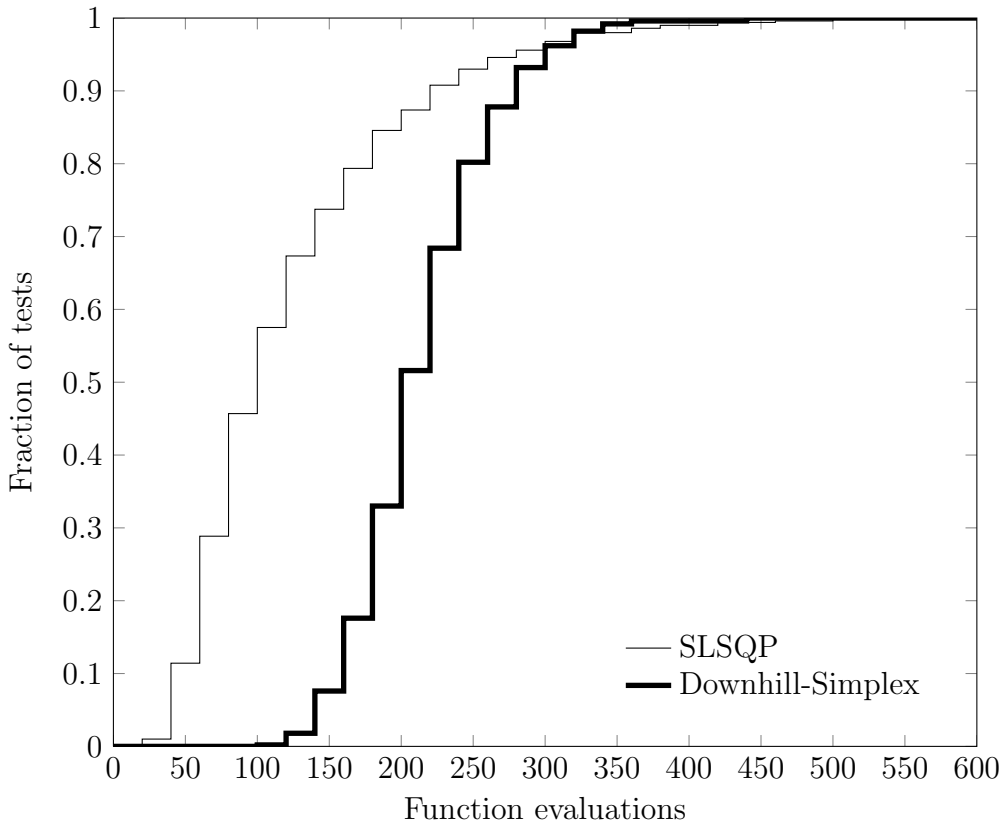

Figure 6.8: Cumulative plot of function evaluations showing faster execution of SLSQP when compared to simplex

6.2.2 Accuracy

The solutions for high and low constraint set fitting resulted in an optimal answer for each test. The fitting of arbitrary sized constraint sets suffered from accuracy issues due to the high dependence on the starting point. As mentioned in the preceding section, due to problems with the unconstrained, gradient-based solvers, only simplex was used for the fitting of arbitrary sets.

Two variable systems

For these tests, a 3-constraint set was fitted into a 4-constraint set. The initial set was chosen to be rectangular (high and low limits) to enable the algebraic calculation of the optimal fitted set. The optimal solution is 50% of the volume (area in 2 dimensions) of the initial constraint set. Figure 6.9 shows the accuracy of this fitting test. It can be seen that 65.8% of the solutions were within 10% of the optimal solution.

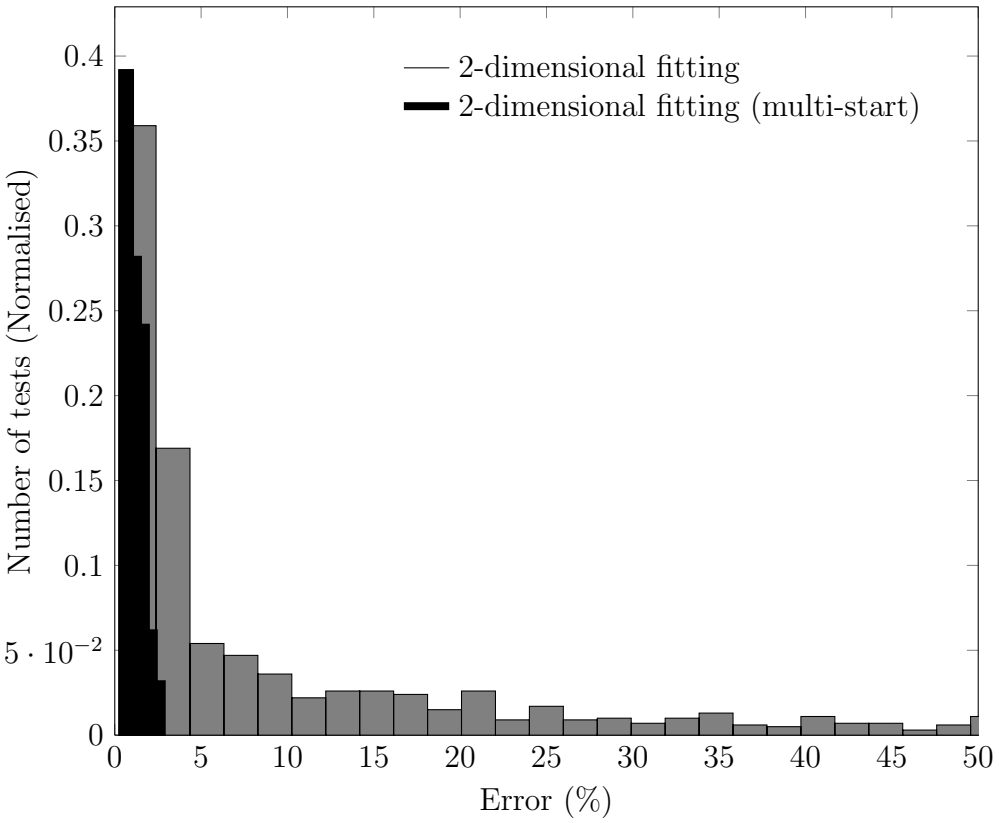

Figure 6.9: Accuracy of fitting results for a 3-constraint set into a 4-constraint set.

To increase the accuracy, a multi-start approach was taken where 25 random starting points are generated. The largest volume set (of the 25 generated) was then used for the fitting. Figure 6.9 shows the improvement in accuracy with this method.

Three variable systems

For the three variable tests, a 4-constraint set was fitted into a 6-constraint set. The initial constraint set was chosen to be a 3-dimensional cube – again to allow for calculation of the optimal solution. This test (in geometric terms) results in fitting a tetrahedron into a cube; the optimal solution being 33.33% of the volume of the cube. Figure 6.10 shows the accuracy of this test.

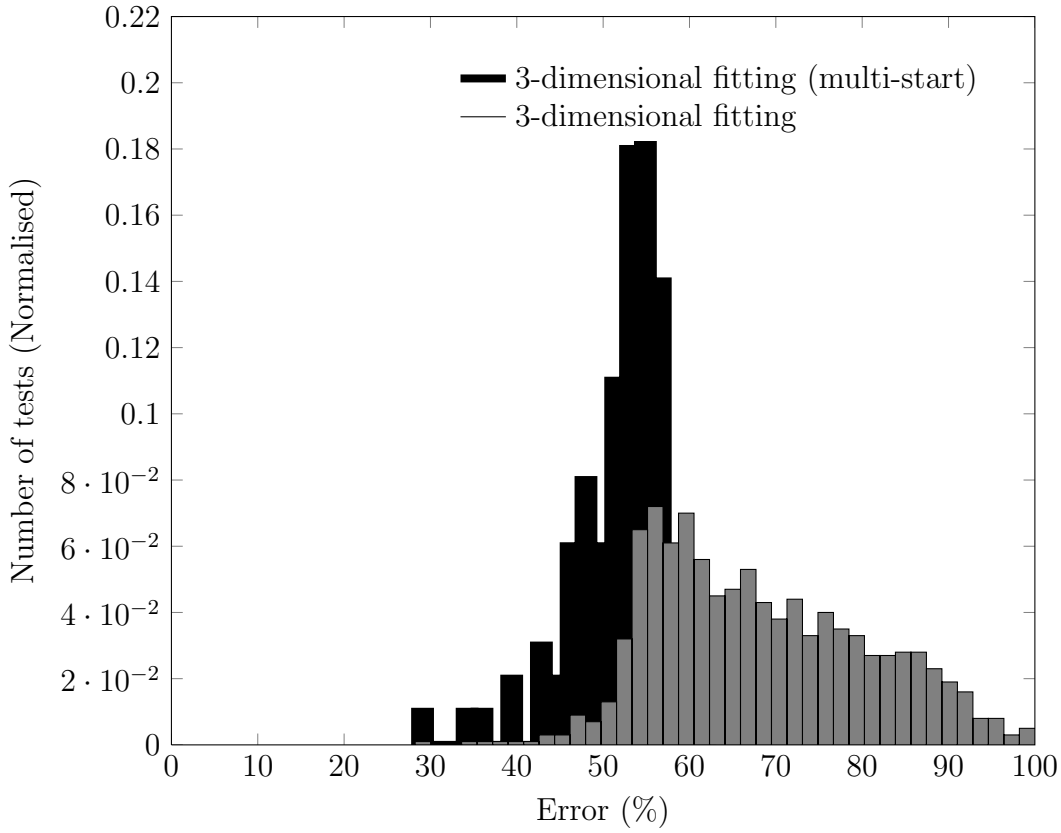

Figure 6.10: Low accuracy is obtained for higher dimensional fitting of constraint sets.

None of the tests resulted in an error below 10%. 22.8% was the lowest error obtained from all the tests. The low accuracy (even when using a multi-start approach) can be ascribed to the increased degrees of freedom and the ill conditioning of the problem.

6.2.3 Set fitting expansion

The graphs shown in figure 6.11 show a few optimal solutions obtained from the two variable tests of section 6.2.2. It can be seen that even though the sets are not equal, the output Operability Index (Vinson, 2000) calculated for all of them are equal.

From these results the following observations regarding the Operability Index of Vinson (2000) can be made:

- the Operability Index is a measure of 'mobility' and emphasises the need for inputs to have a good working range to achieve outputs.

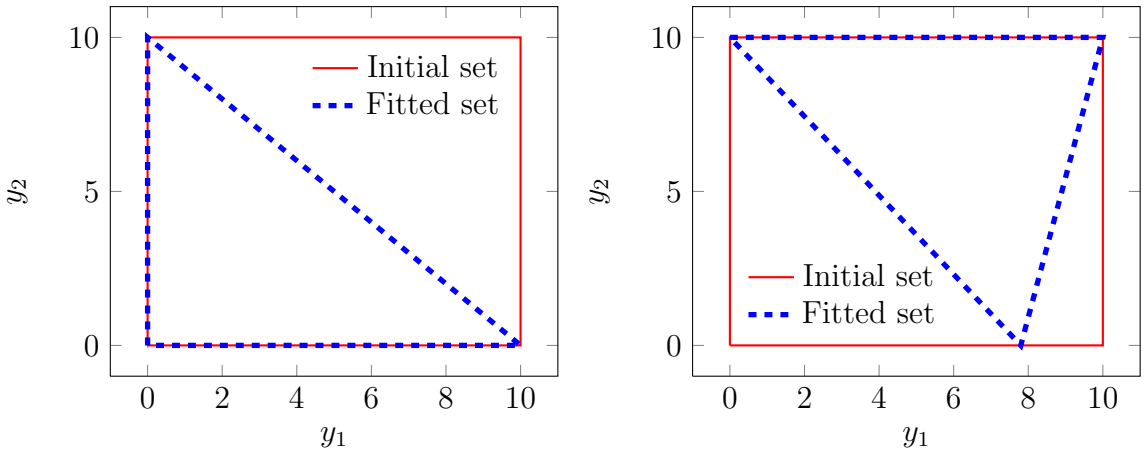

Figure 6.11: Different sets fitted with an equal size and equal calculated Operability Index.

- all of the input and output space are of equal importance in the Operability Index.

It is intuitive that certain sections of the input or output space are of greater importance when considering process economics, sensitivities, etc. It would therefore be a sensible approach to take into consideration an additional objective function when fitting constraint sets (or when making any changes to the input or output spaces). Fitting a set as the volume integral of the following possible objective functions would be favourable:

- Economic objective functions would generate an operating region that maximises profit or minimises cost.
- Sensitivity functions would identify regions that are problematic to control in and favour them less.
- Design cost functions would aid in process design when processes are being designed or modified – costs can be directly related to the improvement in control.

These improvements to the constraint set fitting are suggested as a topic for future research.

CHAPTER 7

CONCLUSIONS AND RECOMMENDATIONS

This chapter summarises the conclusions drawn from this project and makes recommendations for further study.

7.1 Conclusions

The following conclusions can be drawn based on the results of the proposed method of systematic constraint handling:

- Specification of unrealistic constraints leads to a low Operability Index and unattainable expectations of both the inputs and outputs of a process.
- If constraints are checked via the process model, constraint sets that are not attainable can be identified. Also, checking the changes made to constraints (via the process model) allow for a measure of clamping to be determined.
- Constraints that are linear combinations of variables can be implemented in commercial MPCs by adding unmeasured variables to the process and imposing high or low constraints thereon. This procedure improves the operating region of the controller but increases the size of the control problem.
- The results obtained by the method are readily applicable to commercial MPCs via user interfaces or as external programs.

From the process of fitting constraint sets, these additional conclusions can be drawn:

- The size of a constraint set can be reduced by fitting a smaller set within, while maximising the operating region of the fitted set.
- A more rigorous formulation is obtained when the optimisation problem is expressed in terms of facets rather than vertices, but this increases the solution complexity.

- The constrained problem formulation resulted in faster solution times when compared to the unconstrained case.
- The accuracy of fitting for arbitrary sets yielded the following results:
  - 65.8% of fitting results were within 10% of the optimum solution for set fitting in 2 dimensions. This increased to 100% within 3% of the optimum when using a multi-start approach.
  - 0% of fitting results were within 10% of the optimum solution for set fitting in 3 dimensions. 22.8% was the minimum error obtained for the tests. This low accuracy is ascribed to ill conditioning of the problem and increased superfluous degrees of freedom.

7.2 Recommended future research

The conclusions made and work presented in this dissertation opens a few avenues for further improvement and research.

7.2.1 Systematic constraint handling

At present the proposed method of systematic constraint handling consists only of routines to do calculations and support only a small subset of possible models. The following would be attractive additions to the method:

- Support for more model types.
- A graphical user interface to bind all the routines into an easily accessible tool.
- An improved method of data representation for higher order systems, as graphical representation becomes difficult in more than 3 dimensions.

7.2.2 Constraint set fitting

The process of constraint set fitting should be improved upon to increase its accuracy for higher dimensional systems.

Also, the incorporation of additional functions into the fitting procedure would improve the output of this process. Fitting done with respect to the volume integral of (for example) economic or sensitivity functions would highlight regions of economic or control concern.

7.2.3 The Operability Index

While not strictly speaking part of this project, an extension of the Operability Index and its associated spaces is proposed. The inclusion of dynamics in the models should produce a transient space around the original AOS. The information obtained from this space could be valuable in determining optimal and realistic dynamic constraints for MPC packages, e.g. the funnels of RMPCT. This transient space could also be used to help determine optimal move paths.

APPENDIX A

COMMERCIAL MPC SCREENSHOTS

Some screenshots of Honeywell's RMPCT and AspenTech's DMCplus are included in this appendix. The most important figures in this appendix are the operator interfaces of both MPCs (figures A.3 and A.6), as these interfaces are most frequently used to make changes to constraints.

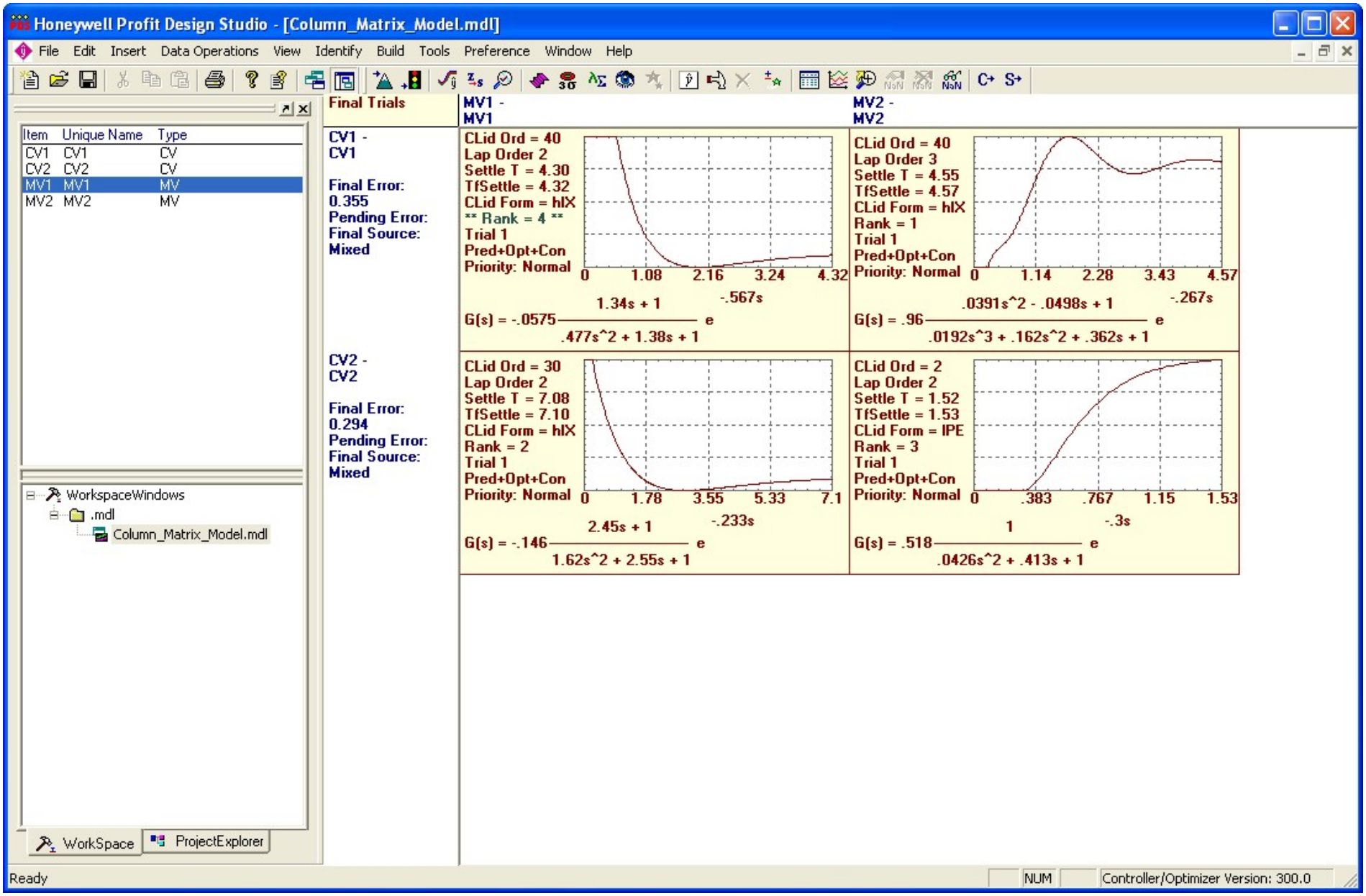

Figure A.1: RMPCT modelling interface – Profit Design Studio

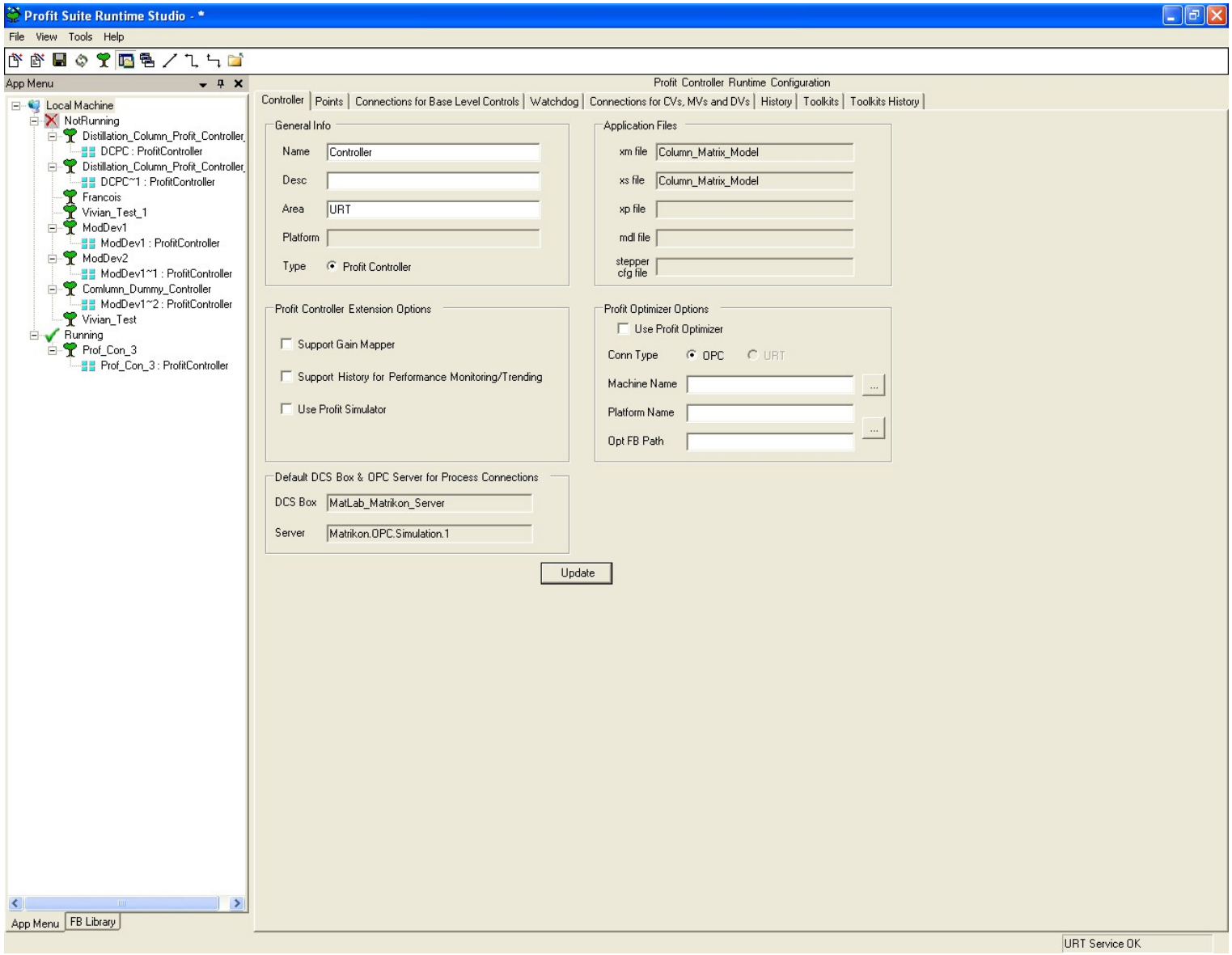

Figure A.2: RMPCT controller building interface – Runtime Studio

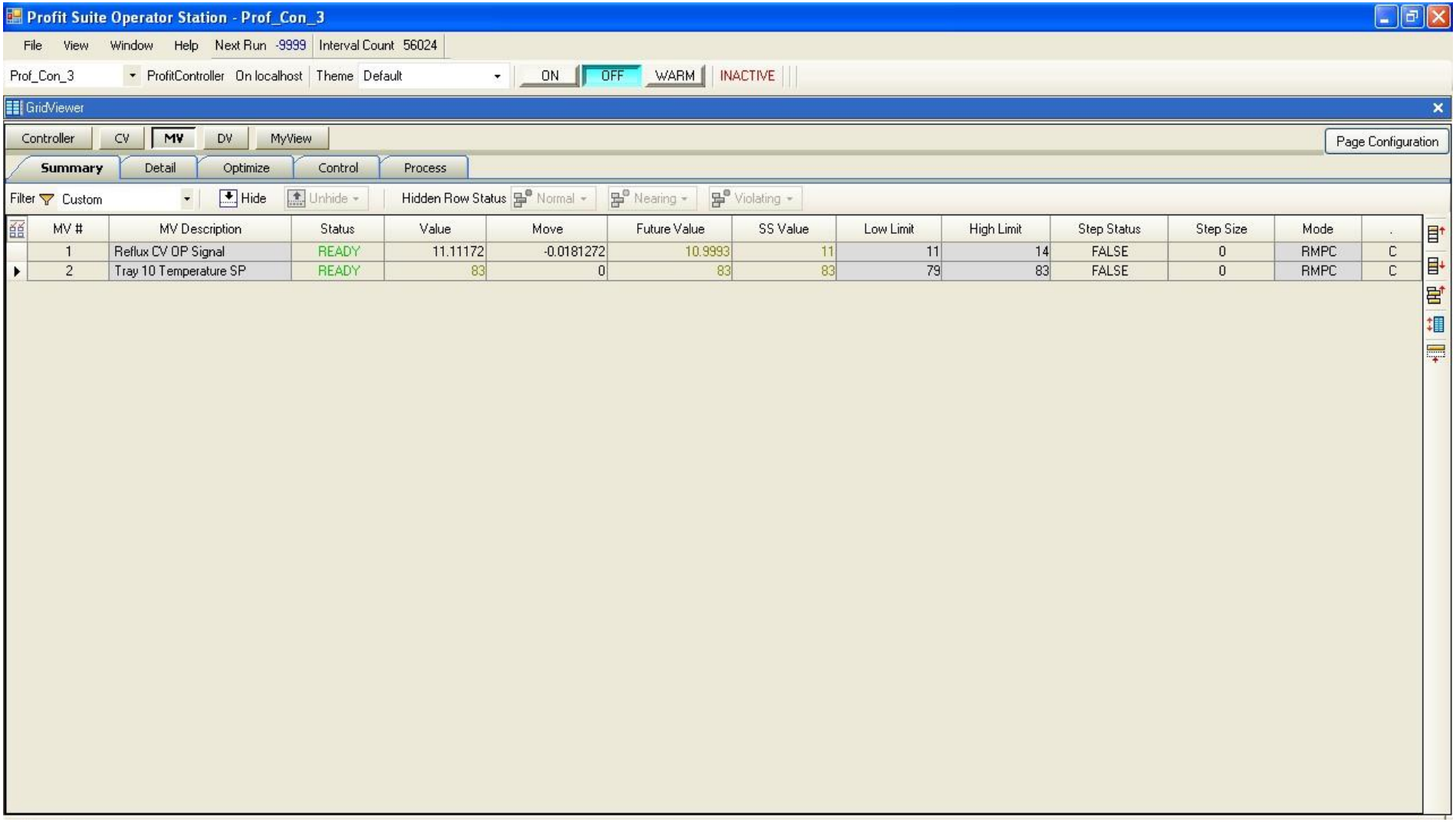

Figure A.3: RMPCT operator interface – Profit Suite Operator Station

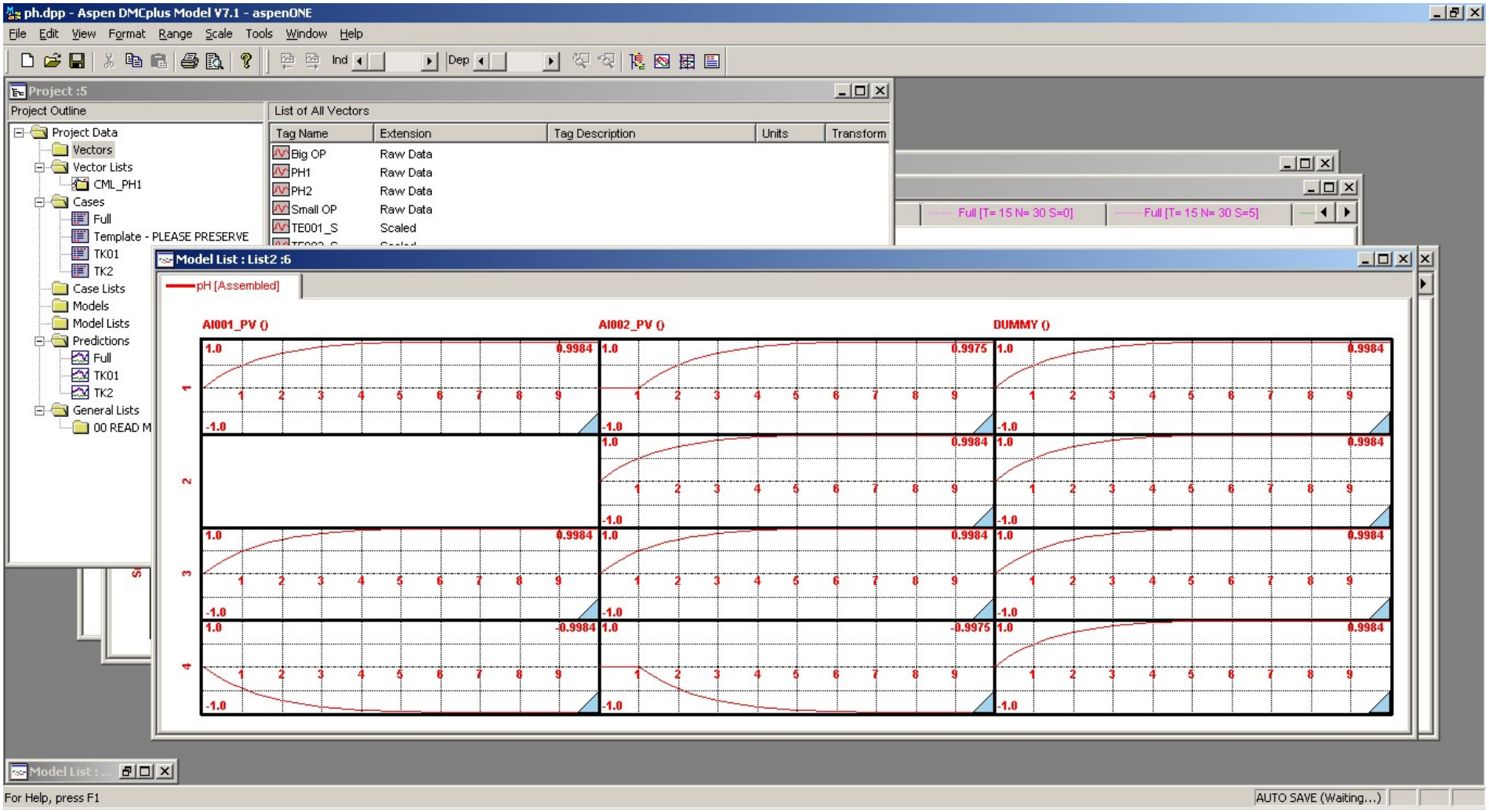

Figure A.4: DMCplus modelling interface – DMCplus Model

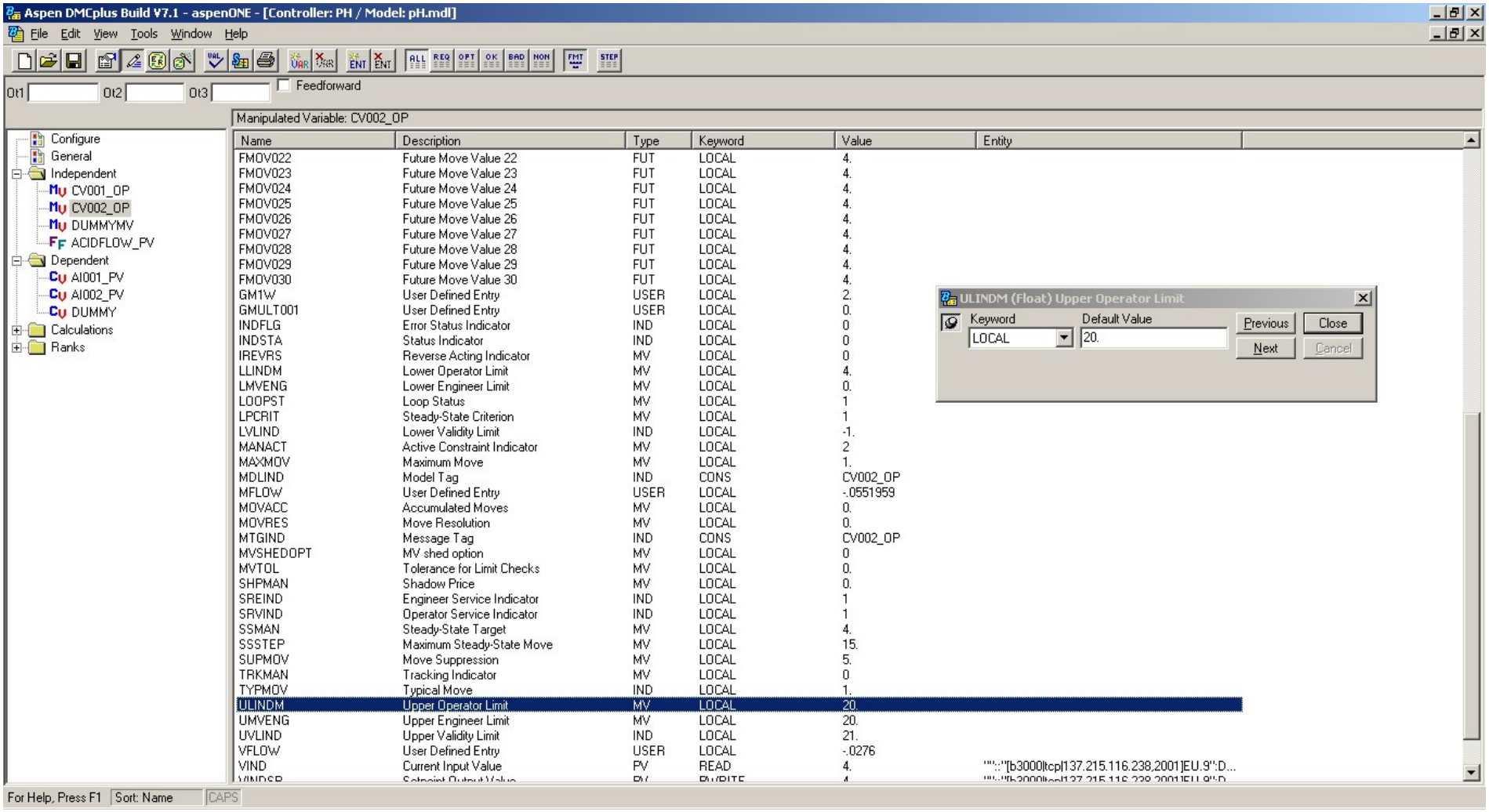

Figure A.5: DMCplus controller building interface – DMCplus Build

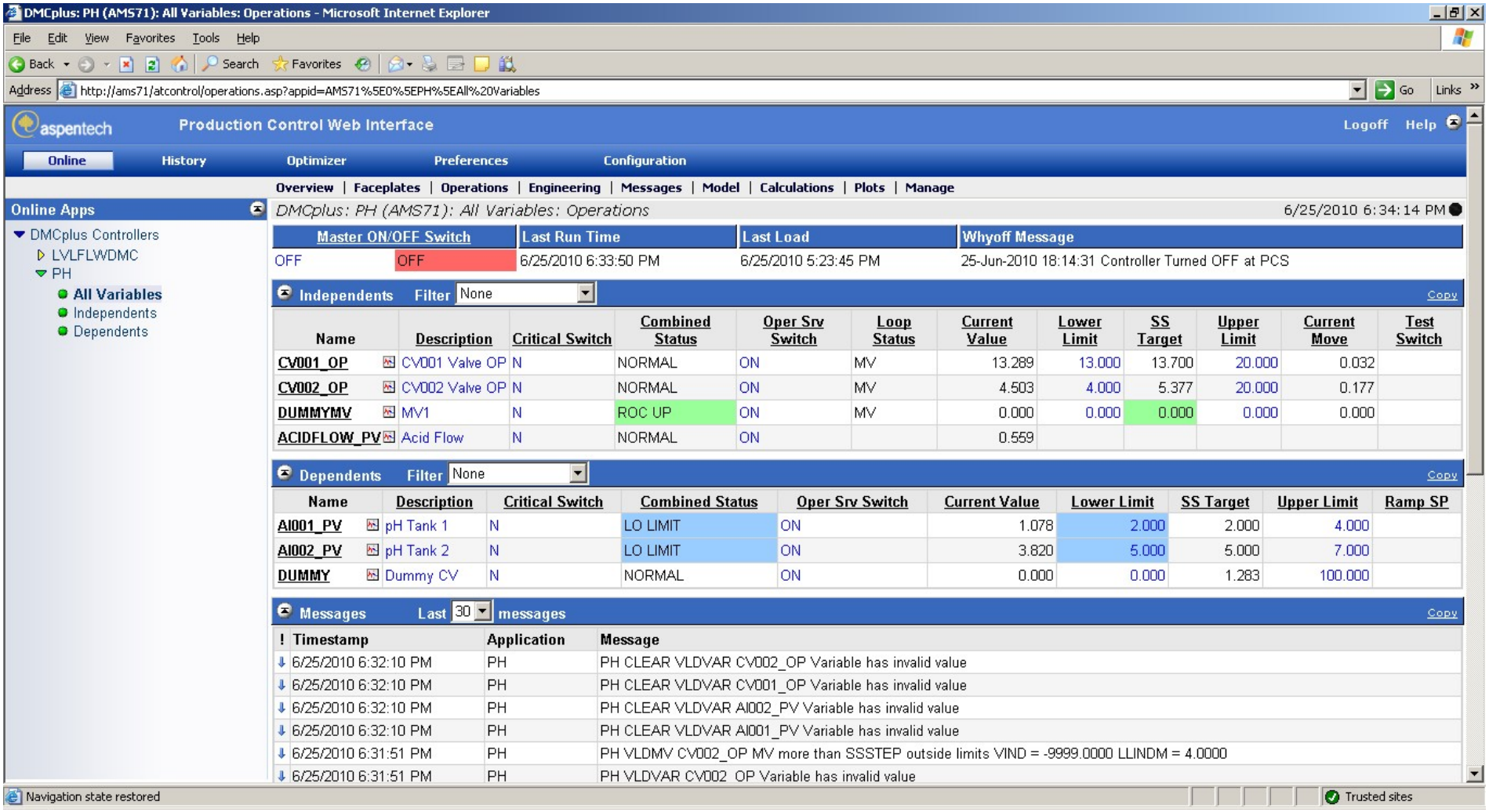

Figure A.6: DMCplus operator interface – Production Control Web Interface

BIBLIOGRAPHY

Aspen Technologies Inc. (2009), "Aspen Manufacturing Suite, Advanced Process Control, DMCplus Reference," DMCPlus technical documentation, Aspen Technologies Inc., Burlington, MA.

Barnette, DW (1971), "The minimum number of vertices of a simple polytope," *Israel Journal of Mathematics*, 10 (1), 121–125.

Bayer, MM and Lee, CW (1993), "Combinatorial aspects of convex polytopes," in PM Gruber and JM Wills, editors, "Handbook of Convex Geometry," volume A, pages 485–534, Elsevier Science Publishers B.V., Amsterdam.

Bremner, D, Fukuda, K and Marzetta, A (1998), "Primal-dual methods for vertex and facet enumeration," *Discrete and Computational Geometry*, 20, 333–357.

Campher, AH (2011a), "AHCampher Thesis – GitHub," URL `https://github.com/CR34M3/AHCampher_Thesis`, [2011, 14 March].

Campher, AH (2011b), "Optimised MPC constraints Code – GitHub," URL `https://github.com/CR34M3/Optimised_MPC_constraints__Code`, [2011, 14 March].

Georgakis, C, Uzturk, D, Subramanian, S and Vinson, DR (2003), "On the operability of continuous processes," *Control Engineering Practice*, 11, 859–869.

Git (2010), "Git – Fast Version Control System," URL `http://git-scm.com/`, [2010, 29 October].

GitHub Inc. (2010), "Secure source code hosting and collaborative development – GitHub," URL `https://github.com/`, [2010, 29 October].

Gritzmann, P and Klee, V (1993), "Mathematical programming and convex geometry," in PM Gruber and JM Wills, editors, "Handbook of Convex Geometry," volume A, pages 627–674, Elsevier Science Publishers B.V., Amsterdam.

Grossmann, IE and Floudas, CA (1987), "Active constraints strategy for flexibility analysis in chemical processes," *Computers & Chemical Engineering*, 11 (6), 675–693.

Honeywell International Inc. (2007a), "Advanced Process Control, Identifier, Users Guide (Release 310)," RMPCT technical documentation, Honeywell International Inc., Phoenix, AZ.

Honeywell International Inc. (2007b), "Advanced Process Control, Profit Controller, Concepts Reference Guide (Release 310)," RMPCT technical documentation, Honeywell International Inc., Phoenix, AZ.

Honeywell International Inc. (2007c), "Advanced Process Control, Profit Controller, Designers Guide (Release 310)," RMPCT technical documentation, Honeywell International Inc., Phoenix, AZ.

Leung, KT (1974), *Linear Algebra and Geometry*, Hong Kong University Press, Hong Kong.

Lima, FV (2007), *Interval Operability: A Tool To Design The Feasible Output Constraints For Non-Square Model Predictive Controllers*, Ph.D. thesis, Tufts University.

Luyben, WL (1990), *Process Modeling, Simulation and Control for Chemical Engineers*, 2nd edition, McGraw-Hill, New York.

Maciejowski, JM (1989), *Multivariable feedback design*, Addison-Wesley, Workingham.

Maciejowski, JM (2001), *Predictive Control with Constraints*, Prentice Hall, New York.

Mani-Levitska, P (1993), "Characterizations of convex sets," in PM Gruber and JM Wills, editors, "Handbook of Convex Geometry," volume A, pages 19–41, Elsevier Science Publishers B.V., Amsterdam.

Marlin, TE (2000), *Process Control, Designing Processes and Control Systems for Dynamic Performance*, 2nd edition, McGraw-Hill, Boston.

Python Software Foundation (2010), "Python programming language – official website," URL `http://www.python.org/`, [2010, 29 October].

Qin, SJ and Badgwell, TA (2003), "A survey of industrial model predictive control technology," *Control Engineering Practice*, 11, 733–764.

Rawlings, JB and Mayne, DQ (2009), *Model Predictive Control: Theory and Design*, Nob Hill Publishing, Madison.

Rossiter, JA (2003), *Model-based Predictive Control: A Practical Approach*, CRC Press, Boca Raton.

Skogestad, S and Postlethwaite, I (2005), *Multivariable Feedback Control, Analysis and Design*, 2nd edition, John Wiley & Sons Ltd., West Sussex.

Stanley, G, Marino-Galarraga, M and McAvoy, TJ (1985), "Shortcut Operability Analysis. 1. The Relative Disturbance Gain," *Industrial & Engineering Chemistry Process Design and Development*, 24 (4), 1181–1188.

Subramanian, S and Georgakis, C (2001), "Steady-state operability characteristics of idealized reactors," *Chemical Engineering Science*, 56, 5111–5130.

Swaney, RE and Grossmann, IE (1985), "An index for operational flexibility in chemical process design. part 1: Formulation and theory," *AIChE Journal*, 31 (4), 621–630.

Vinson, DR (2000), *A New Measure of Process Operability for Improved Steady-State Design of Chemical Processes*, Ph.D. thesis, Lehigh University.

Vinson, DR and Georgakis, C (2000), "A new measure for process output controllability," *Journal of Process Control*, 10, 185–194.

Viswambharan, A (1998), "Optimisation in diamond cutting," Technical report, University of Auckland, New Zealand.

Weisstein, EW (2003), *The CRC Concise Encyclopedia of Mathematics*, 2nd edition, Chapman & Hall, New York.

Wenger, R (2004), "Helly-type theorems and geometric transversals," in KH Rosen, editor, "Handbook of Discreet and Computational Geometry," 2nd edition, pages 71–94, Chapman & Hall, Boca Raton.

Code Manual and Program Listing for a Systematic Approach to Model Predictive Controller Constraint Handling: Rigorous Geometric Methods

André H. Campher

September 2011

Contents

1 Introduction

This document contains overviews of selected project code sections and gives details about design and implementation. The second part of this document gives the complete program listing for the project.

This document should accompany the project dissertation and the program (provided on the CD).

2 Installation notes

2.1 Operating systems

The routines for this project was developed in Linux (Ubuntu 10.04) and this is the only supported operating system. The code has been run on Apple Mac, but instructions for this is not included in this document.

2.2 Dependencies

The code was developed using Python 2.6 and should be compatible with subsequent versions. In addition to the base Python modles, NumPy and SciPy are required for calculations. Subprocess calls in conjunctions with the qhull binaries are utilised, which requires both the qhull libraries and binaries to be installed.

Table 1 lists the packages that need to be installed on Ubuntu. These can be installed from the command line using `sudo apt-get install` *[package name]*.

Package	Package name
Python 2.6	`python2.6`
SciPy	`python-scipy`
NumPy	`python-numpy`
qhull libraries	`libqhull5`
qhull binaries	`qhull-bin`

Table 1: Package installations required in Ubuntu 10.04.

3 Programming details

3.1 Programming language

The routines developed for this project was done using the Python programming language. The SciPy and NumPy modules provide the majority of the mathematical functions used.

3.2 Qhull

Extensive use is made of the `qhull` algorithm (Barberand *et al.*, 1996) for the calculation of normals, vertices and hypervolumes. This algorithm was selected on the basis of its multidimentional functionality and the availability of Unix binaries. Other Python libraries were considered but deemed insufficient, the most prominent routines being:

- The `spatial` module of SciPy contains a Python implementation of the `qhull` algorithm but does not currently allow for volume and normal calculation.
- The CGAL-Python project (Meskini, 2009) provides Python bindings to the Computational Geometry Algorithms Library (CGAL) but does not support more than 3 dimensions.

For this project, the `qhull` algorithm is directly implemented from its Unix binaries. A Python subprocess call is used and communication with `qhull` is done through standard input and standard output. The `qhull` algorithm works with vertices and conversion between vertices and half-spaces are done throughout the code.

3.3 Constraint set class

A Python class, `ConSet`, was created to handle constraint sets. Commonly used attributes of constraint sets were added as properties of the class and commonly used operations were added as methods.

3.3.1 Constructors

Constraint sets can be constructed from the set of constraints or from the vertices of their feasible region. For construction from vertices, the set is assumed to be closed and the corresponding constraints are generated accordingly. Open sets can occur when constructing from constraints as

the half-spaces comprising the set are predefined. Open sets are flagged as erroneous and construction does not proceed.

3.3.2 Methods

The methods of the `ConSet` class are briefly discussed below.

Volume
The `qhull` algorithm is used to calculate the hypervolume of the constraint set.

Space conversion
The process model is used when converting from the input space to the output space (or vice versa). When using a steady-state model (real matrix of gains) this results in a matrix multiplication. Therefore, the output of this operation is a linear transformation of the vertices of the initial constraint set.

Since the steady-state gain model of the system is used steady-state information is lost, and the result of the matrix multiplication is a change in outputs (e.g. $\Delta\mathbf{u}$). The nominal operating point (around which the model was linearised) is therefore specified and used to translate the change in the output space to absolute values. For the case where only sections of a space are converted (e.g. intersections), the deviation from the nominal operating point is transformed and added to the final translation.

Intersections
When the intersection of two constraint sets need to be calculated (as is the case with the AOS and the DOS), a method of superfluous constraints is used.

In an attempt to avoid the problem of degeneracy (dissertation section 2.1.1) the constraints that define the intersection are preprocessed. Redundant constraints are removed or retained according to the following criteria:

- Co-planar (duplicate) constraints will appear in the final solution and are retained.
- If all the vertices (of both intersecting sets) satisfy a given constraint, and a duplicate thereof does not exist, then that constraint is degenerate and excluded from the solution.
- If only a subset of the vertices violate a given constraint, then no verdict can be made about its degeneracy.

The final retained constraints of both sets are combined into a single set, the feasible region of this set represents the intersection of the two initial sets.

Constraint reformatting
The `ConSet` class allows for constraint sets to be expressed as a set of upper and lower limits. The output is a reformatted A matrix and $\mathbf{b}$ vector along with the corresponding transformations for the added variables.

Checks
A check to determine whether one constraint set is contained within another was added to the class. The `allinside` method checks whether all vertices of a constraint set satisfy the constraints of another set. With the enforcement of convexity, this is a sufficient test to ensure that the set is indeed contained.

When a set is not contained within another set, a measure to quantify the containment of the outer set is generated. For each constraint of the outer set, the perpendicular distance of all vertices violating that constraint is calculated and summed.

3.4 Constraint and vertex conversions

As mentioned in section 3.2, most calculations are done using the vertices of a constraint set. Methods to convert from vertices to constraints (and vice versa) are included in this program.

For conversion from vertices to constraints, the normals of the convex hull of the set of vertices are calculated. These normals satisfy $A\mathbf{x} \leq -\mathbf{b}$ (Barber, 2010) and from this the constraints can be calculated directly.

The conversion from constraints to vertices is slightly more complex. Matlab code by Kelder (2005) was converted to Python code. This code employs a primal-dual polytope method to calculate the resulting vertices. This method can produce duplicate vertices; the output is therefore checked and duplicates removed.

3.5 Constraint set fitting

3.5.1 Facet formulation

The minimization problem formulated in the dissertation is reviewed in equation 1 below.

$$\begin{aligned} \min_{A,\ \mathbf{b}} \quad & \text{-volume}_{P_i} \\ \text{s.t.} \quad & P_i \subset P_o \\ & P_i \text{ and } P_o \text{ convex} \end{aligned} \tag{1}$$

This formulation is based on the facets of P_i, with A and $\mathbf{b}$ as the input variables for the minimization. From these two variables the vertices of P_i are calculated and subsequently the volume.

The advantages of using this formulation are:

- It is a more rigorous approach, as the constraints that comprise the set are actively changed to find the optimum.
- By fixing the sizes of A and $\mathbf{b}$, the specified number of constraints to fit is ensured.
- With the use of only linear constraints, the convexity of P_i is ensured. This results from the feasible region of half-spaces being strictly convex.

This formulation, although rigorous, suffers from some problems in execution:

- Inconsistent gradient information is generated when checking if P_i is inside P_o. This arises from the fact that the concept of being 'inside' one another is not defined for two half-spaces (with the exception of parallel half-spaces).
- Superfluous degrees of freedom are present, as the complete A matrix and $\mathbf{b}$ are solved for. There is effectively 1 false degree of freedom for each facet of P_i. The rationale for still using this formulation is simple – full control of the A matrix allows for all orientations of hyperplanes to be expressed without the use of infinite slopes.
- Due to A and $\mathbf{b}$ begin fully changeable, shape inversion can occur. This occurs when constraints move in such a way that an open set is generated. Figure 1 illustrates this concept.
- A scaling-type problem presents itself with the calculation of the volume of P_i. This comes from the fact that the volume of P_i is the same for $[A,\ \mathbf{b}]$ and $[zA,\ z\mathbf{b}]$ where $z \in \mathbb{R}$. Very large elements in the rows of A (and their corresponding elements in $\mathbf{b}$) dramatically reduces the sensitivity of objective functions. This problem can, however, be overcome by adding a constraint to the solver used to keep the norm of $\mathbf{b}$ at a set value.

3.5.2 Vertex formulation

Another possibility was to express the problem in terms of the vertices of the convex hull of the constraint set. For this formulation, equation 1 needs to

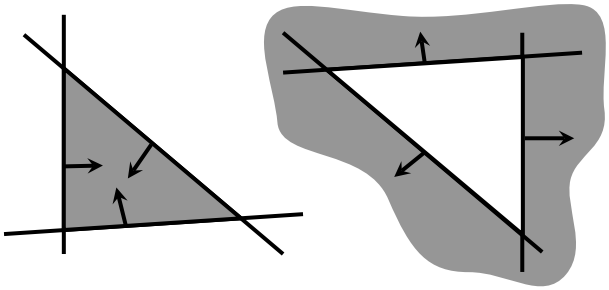

Figure 1: Shape inversion due to the movement of constraints (half-space normals arrows are shown with arrows).

be redefined in terms of the vertices, as shown in equation 2

$$\begin{aligned} \min_{\mathbf{v}} \quad & \text{-volume}_{P_i} \\ \text{s.t.} \quad & P_i \subset P_o \\ & P_i \text{ and } P_o \text{ convex} \end{aligned} \tag{2}$$

where $\mathbf{v}$ is all the vertices of P_i. A and $\mathbf{b}$ will now only result from the final optimised P_i.

The vertex based formulation has the following advantages:

- This approach is more intuitive as the fitting problem, conceptually, involves moving the vertices of P_i to the facets of P_o.
- Checking that P_i is inside P_o is now a direct operation, as vertices don't need to be calculated from A and $\mathbf{b}$ first.
- When a measure of containment needs to be generated, consistent gradient information can be expressed in terms of each vertex of P_i.

Most of the problems with the vertex based method stems from the movement of vertices to the inside of the convex hull of P_i during the solution of the problem. These internal points are now no longer vertices, but are still considered in the optimisation. This formulation was abandoned in this project due to the following disadvantages in its implementation:

- A clear correlation between the number of vertices and the number of facets on a shape does not exist (unless some properties of the shape are given). Figure 2 illustrates this concept for two 6-faceted, 3 dimensional shapes. This is a major problem when the number of facets (constraints) on P_i needs to be specified.

- A slightly different problem than the one mentioned above – the number of facets on P_i are determined by the final optimised vertices. Internal points and duplicate vertices in the solution will cause the number of facets to decrease. Therefore, specification of a set number of facets becomes nearly impossible.

- Enforcing convexity becomes a problem when vertices are used directly. This is caused by the movement of vertices inside the convex hull of P_i. If all points are to be vertices by definition, non-convex shapes are generated. The use of the `qhull` algorithm to calculate the volume now becomes invalid.

- Another problem with internal points occurs during the calculation of the volume of P_i. The perturbation of internal points have no effect on the volume (as only the convex hull is considered). This results in decreased sensitivity of objective functions.

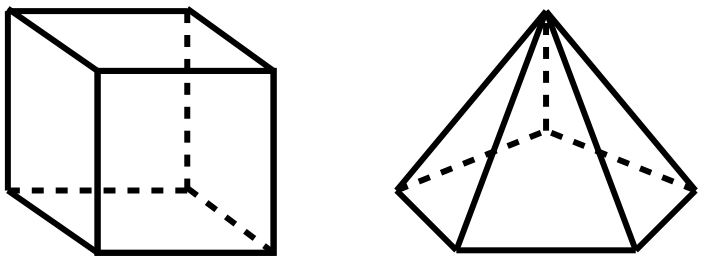

Figure 2: A cube and a pentagonal pyramid in 3 dimensions – both shapes have 6 facets, but differ in their number of vertices (8 and 6 respectively).

3.5.3 Optimisation

For the solution of the problem, two optimisers (solvers) were considered. Both are contained in the SciPy `optimize` module:

- A sequential least squares quadratic programming (SLSQP) algorithm, `fmin_slsqp`, henceforth simply referred to as SLSQP. This is a gradient based, constrained solver.

- A downhill-simplex algorithm, `fmin`, henceforth simply referred to as simplex. This is an unconstrained solver.

3.5.4 Objective functions

For SLSQP a combination of equality and inequality constraints were used. The constrained problem formulation for SLSQP is shown in equation 3 and follows from equation 1. The criteria that P_i should be convex is satisfied by using linear inequalities ($A\mathbf{x} \leq \mathbf{b}$) for its construction. The convexity of P_o is assumed to be a given.

$$\begin{aligned} \min_{A,\ \mathbf{b}} \quad & \text{-volume}_{P_i} \\ \text{s.t.} \quad & \text{norm}(\mathbf{b}) = 1 \\ & \text{vertices}_{P_i} \text{ satisfy } (A\mathbf{x} \leq \mathbf{b})_{P_o} \end{aligned} \tag{3}$$

The constraints as imposed on SLSQP are not possible for simplex. A penalty formulation for the objective function is therefore used. Equation 4 shows the objective function for use with simplex. The same argument for convexity holds as with SLSQP.

$$\min_{A,\ \mathbf{b}} \quad \text{-volume}_{P_i} + p_A(\text{norm}(\text{distance of vertices}_{P_i} \text{ outside } P_o)) + p_B(|\text{norm}(\mathbf{b}) - 1|) \tag{4}$$

where p_A and p_B are penalty weights.

The same objective functions (equations 3 and 4) hold for the fitting of rectangular constraint sets. Only difference begin that $\mathbf{b}$ is the only input variable and A has a fixed shape (as shown in equation 4.6 in the dissertation).

3.5.5 Shape exceptions

Two types of shape exceptions are handled during the optimisation. Shape inversion (as shown in figure 1) is handled by applying a sign to the volume of the shape. A sufficient criteria for an open, convex shape is when any one of the vertices violate any one of the constraints used to generate the shape. If a shape is open, the internal volume is calculated and given a negative sign.

The second class of shape exceptions is that of unbound shapes. These shapes are generated when enough constraints (used to generate the shape) are parallel, causing the shape to be unbound. The difference between these open shapes and those resulting from shape inversion is that there are no internal vertices. In this event, a large bounding n-cube is placed around P_o, closing the shape but generating vertices that are external to P_o. This method was chosen as opposed to returning an infinite volume.

3.5.6 Starting point generation

For the minimization problem, a starting point needs to be generated. A different method for starting point generation is utilized for arbitrary shape fitting and rectangular shape fitting respectively.

Both methods generate a starting shape around the centroid of P_o. With the assumption of convexity the centroid is simply calculated as the average of all the vertices of P_o.

For arbitrary shapes, $k-1$ random Gaussian vectors are generated and projected to points on an n-sphere (centered around P_o's centroid). A final vector is generated to ensure a closed shape. This vector is the negative of the resultant of the first $k-1$ vectors. Tangential planes to the n-sphere at each of the k points are generated. Finally, the sphere is scaled by a 1% scaling factor so that the resulting k-faceted shape is within P_o (to the largest extent).

For rectangular shapes a deterministic starting point is generated. An n-cube is generated around P_o's centroid. This cube is then scaled by a 1% scaling factor to be within P_o (to the largest extent).

4 Program Listing

auxfuns.py

```
#!/usr/bin/env python
"""Auxiliary functions to manipulate data-types and and
    change data-formats."""
from scipy import mat, array, ones
import numpy
import subprocess #to use qhull

def splitAb(inAb, nd):
    """Split input Ab array (1d) into separate A and b."""
    tmpAb = inAb.reshape(len(inAb)/(nd + 1), nd + 1)
    A = tmpAb[:, :-1]
    b = array([tmpAb[:, -1]]).T
    return A, b

def uniqm(A, t=1e-13):
    """
    Return input matrix with duplicate entries removed.
     A - [matrix] input matrix (possibly containing
         duplicates)
     t - [float]  tolerance (default=1e-13)
    """
    Nrows = A.shape[0]
    uniquerows = [r1 for r1 in range(Nrows)
                  if not any(numpy.all(abs(A[r1, :] - A[r2,
                      :]) < t)
                                for r2 in range(r1 + 1, Nrows)
                                    )]
    return A[uniquerows, :].copy()


def mat2ab(Asbmat):
    """
    Transform [A s b]-form matrix to standard Ax<b notation
        .
     Asbmat - [array] inequality matrix in the form Ax<b,
         matrix = [A s b]
                          with s the sign vector [1:>, -1:<]
    """
    stmp = array([Asbmat[:, -2]])
```

```
    b = Asbmat[:, -1]*-stmp
    A = Asbmat[:, :-2]*-stmp.T
    s = -ones(stmp.shape)
    return A, s.T, b.T

def qhullstr(V):
    """
    generate string qhull input format.

    yields a newline separated string of format:
        dimensions (columns of V)
        number of points (rows of V)
        one string for each row of V
    """
    V = numpy.array(V)
    return "%i\n%i\n" % (V.shape[1], V.shape[0]) \
           + "\n".join(" ".join(str(e) for e in row) for
               row in V)

def qhull(V, qstring):
    """
    Use qhull to determine convex hull / volume / normals.
     V - [matrix] vertices
     qstring - [string] arguments to pass to qhull
    """
    try:
        qhullp = subprocess.Popen(["qhull", qstring],
                                  stdin=subprocess.PIPE, stdout
                                      =subprocess.PIPE)
        Vc = qhullp.communicate(qhullstr(V))[0] #qhull
            output to Vc

        if qstring == "FS": #calc area and volume
            ks = Vc.split('\n')[-2]
            Vol = float(ks.split(' ')[-2]) #get volume of D
                -hull
            return Vol
        elif qstring == "Ft": #calc vertices and facets
            ks = Vc.split('\n')
            fms = int(ks[1].split(' ')[1]) #get size of
                facet matrix
            fmat = ks[-fms-1:-1]
            fmat = mat(';'.join(fmat)) #generate matrix
```

```
            fmatv = fmat[:, 1:] #vertices on facets
            return array(fmatv)
        elif qstring == "n": #calc convex hull and get
            normals
            ks = ';'.join(Vc.split('\n')[2:]) #remove
                leading dimension output
            k = mat(ks[:-1]) #convert to martrix with
                vertices
            return array(k)
        else:
            exit(1)
    except:
        raise NameError('QhullError')

if __name__ == "__main__":
    import doctest
    doctest.testfile("tests/auxfunstests.txt")

#TODO - auxfuns
# qhull error handling
# fix auxfuns tests
```

conclasses.py

```
#!/usr/bin/env python
"""
Class definitions for constraint set
Author: Andre Campher
"""
# Dependencies: - convertfuns
#               - auxfuns
#               - SciPy

from auxfuns import qhull, mat2ab, uniqm, splitAb
from convertfuns import vert2con, con2vert
from scipy import empty, vstack, dot, tile, all, zeros,
    sqrt, sum, c_, sign
from scipy import mat, ones, linalg, array
from scipy import all as sciall
import numpy

class ConSet:
    """
```

```
    Class for constraint sets. Generated by either a
       constraint set [A s b] or
    by a set of vertices [v].
    """
    def __init__(self, *inargs):
        if len(inargs) == 1:
            self.vert = inargs[0]
            self.A, self.s, self.b = vert2con(self.vert)
            self.closed = True
        elif len(inargs) == 3:
            if all(sign(inargs[1]) == -1):  # ensure an all
                -1 sign vector
                self.A, self.s, self.b = inargs
            else:
                self.A, self.s, self.b = mat2ab(c_[inargs])
            self.vert, self.closed = con2vert(self.A, self.
               b)
        else:
            exit(1)  # TODO: Raise exception
        self.nd = self.A.shape[1]
        self.cscent = sum(self.vert, axis=0)/len(self.vert)

    def vol(self):
        """Return 'volume' of feasible region."""
        return qhull(self.vert,"FS")

    def oi(self, conset2):
        """
        Return the Operabilty Index (Vinson, 2000) of the
           set, where conset2
        is equivalent to the DOS
        """
        Vint = ConSet(*self.intersect(conset2)).vol()
        return Vint/conset2.vol()

    def outconlin(self, model, iss, oss):
        """
        Convert constraints to another space using a linear
            model
        e.g. calc AOS (from G and AIS).
            iss [vector] - nominal steady state of current
               space ("from")
            oss [vector] - nominal steady state of
```

```
            transformed space ("to")
        """
        outverttemp = empty([1, self.vert.shape[1]])
        # handle shifting
        ishift = self.cscent - iss #input space dev
        oshift = model*ishift.T #output space dev
        fshift = oshift.T + oss #output space offset
        # center 'input'-space around [0]
        inverttemp = self.vert - self.cscent
        for v in inverttemp:
            x = model*v.transpose()
            outverttemp = vstack((outverttemp, x.transpose
                ()))
        # remove first line of junk data from outverttemp
            and shift
        outverttemp = outverttemp + fshift
        return vert2con(outverttemp[1:, :])

    def intersect(self, conset2):
        """Determine intersection between current
            constraint set and another"""
        def remredcons(A, b, verts):
            """Reduce a constraint set by removing
                unnecessary constraints."""
            eps = 10e-9
            #1 Co-planar constraints;
            #  Remove as not to affect 3rd check
            Ab = c_[A, b]
            Abnorms = ones((Ab.shape[0], 1))
            for i in range(Ab.shape[0]):
                Abnorms[i] = linalg.norm(Ab[i, :])
            Abn = Ab/Abnorms
            Abkeep = ones((0, Ab.shape[1]))
            Abtest = ones((0, Ab.shape[1]))
            for r1 in range(Abn.shape[0]):
                noocc = ones((1, 0))
                for r2 in range(Abn.shape[0]):
                    #print abs(Abn[r1, :] - Abn[r2, :])
                    if numpy.all(abs(Abn[r1, :] - Abn[r2,
                        :]) < eps):
                            noocc = c_[noocc, r2]
                if noocc.size == 1:
                    Abtest = vstack([Abtest, Ab[r1, :]])
```

```
                else:
                    Abkeep = vstack([Abkeep, Ab[r1, :]])
            if Abkeep.shape[0] > 1:
                Abkeep = uniqm(Abkeep, eps)
            #2 Vert subset satisfying; no action needed (
                redundancy uncertain)
            #3 All vert satisfying constraints;
            A, b = splitAb(array(Abtest).ravel(), verts.
                shape[1])
            keepA = ones((0, A.shape[1]))
            keepb = ones((0, 1))
            bt = tile(b, (1, verts.shape[0]))
            k = mat(A)*mat(verts.T) - bt
            kk = sum(k > eps, axis=1)
            for i in range(len(kk)):
                if kk[i] != 0:
                    keepA = vstack([keepA, A[i, :]])
                    keepb = vstack([keepb, b[i, :]])
            outAb = vstack([c_[keepA, keepb], Abkeep])
            return splitAb(outAb.ravel(), verts.shape[1])
        #Combine constraints and vertices
        combA = vstack((self.A, conset2.A))
        combb = vstack((self.b, conset2.b))
        combv = vstack((self.vert, conset2.vert))
        #Remove redundant constraints
        ncombA, ncombb = remredcons(combA, combb, combv)
        #Calc and return intersection
        intcombvert = con2vert(combA, combb)[0]
        return intcombvert

    def allinside(self, conset2):
        """
        Determine if all vertices of self is within conset2
            . allvinside merely
        returns True/False whereas insidenorm returns a
            measure of 'inside-ness'
        better suited for optimisers.
        """
        # Inside check
        Av = dot(conset2.A, self.vert.T)
        bv = tile(conset2.b, (1, self.vert.shape[0]))
        eps = 1e-13
        intmpvals = Av - bv
```

```
        intmp = intmpvals <= eps
        allvinside = sciall(intmp)
        # Inside norm
        insidenorm = zeros((Av.shape[0], 1))
        for cons in range(Av.shape[0]):
            for verts in range(Av.shape[1]):
                dist = abs(intmpvals[cons, verts])/sqrt(sum
                    (conset2.A[cons, :]**2))
                if not intmp[cons, verts]:  # outside
                    insidenorm[cons] = insidenorm[cons] -
                        dist
        # Outside volume
        return allvinside, insidenorm#, outsidevol
```

convertfuns.py

```
#!/usr/bin/env python
"""Functions to calculate vertices from constraints and
    vice versa."""
from scipy import ones, mat, vstack, zeros, hstack, eye,
    array, dot, tile, sum
from scipy import all as sciall
from numpy import linalg, matlib
from auxfuns import uniqm, qhull

def vert2con(V):
    """
    Convert sets of vertices to a list of constraints (of
        the feasible region).
    Return A, b and s of the set;  Ax < b   (s is the sign-
        vector [to be used later])
    vert2con always closes the shape and generates
        inequalities accordingly.
    """
    # Dependencies: * qhull (libqhull5, qhull-bin)
    #               * scipy
    k = qhull(V,"n") #convert to martrix with vertices
    # k is a (n+1)x(p) matrix in the form [A b] (from qhull
        doc: Ax < -b is
    # satisfied), thus;
    A = k[:, :-1]
    b = array([-k[:, -1]]).T
    s = -ones(b.shape)
```

```
    return A, s, b

def con2vert(A, b):
    """
    Convert sets of constraints to a list of vertices (of
        the feasible region).
    If the shape is open, con2vert returns False for the
        closed property.
    """
    # Python implementation of con2vert.m by Michael Kleder
        (July 2005),
    #  available: http://www.mathworks.com/matlabcentral/
        fileexchange/7894
    #  -con2vert-constraints-to-vertices
    # Author: Michael Kelder (Original)
    #         Andre Campher (Python implementation)
    c = linalg.lstsq(mat(A), mat(b))[0]
    btmp = mat(b)-mat(A)*c
    D = mat(A)/matlib.repmat(btmp, 1, A.shape[1])

    fmatv = qhull(D, "Ft") #vertices on facets

    G  = zeros((fmatv.shape[0], D.shape[1]))
    for ix in range(0, fmatv.shape[0]):
        F = D[fmatv[ix, :], :].squeeze()
        G[ix, :] = linalg.lstsq(F, ones((F.shape[0], 1)))
            [0].transpose()

    V = G + matlib.repmat(c.transpose(), G.shape[0], 1)
    ux = uniqm(V)

    eps = 1e-13
    Av = dot(A, ux.T)
    bv = tile(b, (1, ux.shape[0]))
    closed = sciall(Av - bv <= eps)

    return ux, closed

def con2pscon(cset, G, type):
    """
    Convert a constraint set to another constraint with
        only
    high/low limits.
```

```
        cset - constraint set to convert [ConSet]
        G - model of original set [array]
        type - type of model conversion to be done [string]
                'i'nput constraint set
                'o'utput constraint set
    """
    #Check for single variable entries
    checkmat = sum(zeros(cset.A.shape) == cset.A, axis=1)
    #Determine number of conversions
    nconv = sum(checkmat < cset.nd-1)
    if nconv:
        #keep original high/low limits (s remains unchanged
            )
        keepA = vstack([cset.A[x, :] for x in range(cset.A.
            shape[0])
                        if checkmat[x]])
        keepb = vstack([cset.b[x, :] for x in range(cset.b.
            shape[0])
                        if checkmat[x]])
        fixA = vstack([cset.A[x, :] for x in range(cset.A.
            shape[0])
                        if not checkmat[x]])
        fixb = vstack([cset.b[x, :] for x in range(cset.b.
            shape[0])
                        if not checkmat[x]])
        tempA = zeros((keepA.shape[0]+nconv, keepA.shape
            [1]+nconv))
        tempA[:keepA.shape[0], :keepA.shape[1]] = keepA
        tempA[-nconv:, -nconv:] = eye(nconv)
        tempb = vstack((keepb, fixb))
        if type in "iI":
            tempG = vstack((G, fixA))
        elif type in "oO":
            fixG = sum(G, axis=0)*fixA
            tempG = vstack((G, fixG))
        else:
            tempG = G
        return tempA, cset.s, tempb, tempG
    else:
        return cset.A, cset.s, cset.b, G

if __name__ == "__main__":
    import doctest
```

```
    doctest.testfile("tests/convertfunstests.txt")

#TODO con2vert =====
# error-checking
#    - fix volume check (for redundant constraints)
#TODO general
# check that floating-point math is used
# fix output of con2pscon
```

fitting.py

```
#!/usr/bin/env python
"""Functions to optimally fit one 'shape' into another."""
from scipy import array, optimize, eye, c_, r_, ones, sqrt
from scipy import linalg, sum, size
import numpy
import random
from conclasses import ConSet
from convertfuns import con2vert
from auxfuns import qhull, splitAb

def tryvol(A, b, cs):
    """
    Try to determine volume of feasible region, otherwise
       impose box
    constraint.
    """
    try:
        V = con2vert(A, b)[0] # get vertices for constraint
            set iteration
    except NameError: #catch unbound shapes
        #create large box around original shape
        fixA = r_[eye(cs.nd), -eye(cs.nd)]
        fixb = r_[[numpy.max(cs.vert[:, x]) for x in range(
           cs.vert.shape[1])],
                  [-numpy.min(cs.vert[:, x]) for x in range
                     (cs.vert.shape[1])]]
        fixb = array([fixb]).T
        fA = r_[fixA, A]
        fb = 2. * r_[fixb, b]  # double the box size
        V = con2vert(fA, fb)[0]
    #return vol (normal or fixed) and vertices (normal or
       fixed)
```

```
    return qhull(V, "FS"), V
    #TODO: check box fitting

def genstart(shapetype, *args):
    """
    Generate a starting shape (for optimisation) with given
        number of faces.
        shapetype : [string] (r)ectangle, (a)rbitrary
        args -> (initial constraint set, number of faces)
    """
    if shapetype in 'rR':  # Rectangle
        cs = args[0]
        ncon = cs.nd*2
    elif shapetype in 'aA':  # Arbitrary shape
        cs = args[0]
        ncon = args[1]

    #1. Determine centre of initial region, cscent
    cscent = cs.cscent  # assuming the region is convex
    def spherefn(srad, cent, svecs):
        """
        Function to create points on a sphere (of radius,
            srad), convert them to
        halfspaces and convert the halfspaces to a
            constraint set.
        """
        spherepts = svecs.T*(srad/sqrt(sum((svecs.T)**2)))
        spherepts = spherepts.T + cent  # move to center of
            constraint set
        #4. Generate tangent planes on sphere at points,
            convert to inequalities
        A = -(spherepts - cent)
        b = array([(sum((A*spherepts).T, axis=0))]).T
        s = array(ones(b.shape))
        return ConSet(A, s, b)  # constraints around sphere

    if shapetype in 'rR':  # Rectangle
        Astart = r_[eye(cs.nd), -eye(cs.nd)]
        sstart = -ones((cs.nd*2, 1))
        bstart = lambda k: array(r_[cscent.T + k, -(cscent.
            T - k)]).reshape(size(cscent)*2,1)
        kstart = 1.0
        if ConSet(Astart, sstart, bstart(kstart)).allinside
```

```
            (cs)[0]:
                while ConSet(Astart, sstart, bstart(kstart)).
                    allinside(cs)[0]:
                    kstart = kstart * 1.01
                kfin = kstart / 1.01
            elif not ConSet(Astart, sstart, bstart(kstart)).
                allinside(cs)[0]:
                while not ConSet(Astart, sstart, bstart(kstart)
                    ).allinside(cs)[0]:
                    kstart = kstart / 1.01
                kfin = kstart
            return ConSet(Astart, sstart, bstart(kfin))
        elif shapetype in 'aA':  # Arbitrary shape
            #2. Generate ncon-1 Gaussian vectors
            spherevecs = ones((ncon, cs.vert.shape[1]))
            for rows in range(spherevecs.shape[0] - 1):
                for cols in range(spherevecs.shape[1]):
                    spherevecs[rows, cols] = random.gauss(0,
                        0.33)  # TODO: check sigma
            #3. Determine resultant of vectors, add last vector
                as mirror of resultant
            spherevecs[-1, :] = -sum(spherevecs[:-1, :], axis
                =0)
            # points on sphere
            #6. Optimise sphere-radius, r, to have all points
                within initial shape
            startrad = cs.vol()**(1./cs.nd)
            if spherefn(startrad, cscent, spherevecs).allinside
                (cs)[0]:
                while spherefn(startrad, cscent, spherevecs).
                    allinside(cs)[0]:
                    startrad = startrad * 1.01
                finrad = startrad / 1.01
            elif not spherefn(startrad, cscent, spherevecs).
                allinside(cs)[0]:
                while not spherefn(startrad, cscent, spherevecs
                    ).allinside(cs)[0]:
                    startrad = startrad / 1.01
                finrad = startrad
            return spherefn(finrad, cscent, spherevecs)

def fitshape(cset, spset, solver):
    """
```

```
    Fit a constraint set (specified by the number of
        constraints) within an existing
    constraint set.
     cset - [ConSet] existing constraint set
     ncon - [int] number of constraints to fit
    """
    #### State problem
    # ncongiven > ncon >= nD+1
    # Constraints are bounding
    # All vertices within constraint set
    #### Define parameters
    snorm = linalg.norm(spset.b)
    sp = c_[spset.A, spset.b]/snorm # starting point -
        combined Ab matrix to optimise

    #### Objective fn (FMIN)
    def objfn(Ab, *args):
        """Volume objective function for fmin (Simplex)."""
        initcs = args[0]
        A, b = splitAb(Ab, initcs.nd)
        vol, V = tryvol(A, b, initcs)
        Pv = 200.
        Pn = 100.
        #Penalties
        # large b norm
        bnorm = abs(linalg.norm(b) - 1)
        # points outside of init space
        iterset = ConSet(V)
        outnorm = linalg.norm(iterset.allinside(initcs)[1])
        #outnorm = iterset.allinside(initcs)[2]
        # open shape
        cl = con2vert(A, b)[1]
        if cl:
            closed = 1
        else:
            closed = -1
        vol = tryvol(A, b, initcs)[0]
        return (-vol*closed) + Pn*(bnorm**initcs.nd) + Pv*(
            outnorm**(initcs.nd+3))

    #### Objective fn (SLSQP)
    def objfn2(Ab, *args):
        """Volume objective function for SLSQP."""
```

```
    initcs = args[0]
    A, b = splitAb(Ab, initcs.nd)
    cl = con2vert(A, b)[1]
    if cl:
        closed = 1
    else:
        closed = -1
    vol = tryvol(A, b, initcs)[0]
    return closed * -vol
def eqconsfn(Ab, *args):
    """Optimiser equality constraint function."""
    initcs = args[0]
    b = splitAb(Ab, initcs.nd)[1]
    # get vertices
    bn = linalg.norm(b) - 1
    return array([bn])
def ieqconsfn(Ab, *args):
    """Optimiser inequality constraint function."""
    initcs = args[0]
    A, b = splitAb(Ab, initcs.nd)
    # get vertices
    V = tryvol(A, b, initcs)[1]
    iterset = ConSet(V)
    # constraint checking
    if iterset.allinside(initcs):
        return 0
    else:
        return -1

#### Maximise volume
if solver in 'aA':
    optAb = optimize.fmin_slsqp(objfn2, sp, f_eqcons=
        eqconsfn,
                                f_ieqcons=ieqconsfn,
                                    args=[cset], iprint
                                    =0)
elif solver in 'bB':
    optAb = optimize.fmin(objfn, sp, args=[cset],
        maxiter=20000, disp=False)
elif solver in 'cC':
    optAb = optimize.fmin_cobyla(objfn2, sp, ieqconsfn,
        args=[cset], iprint=0)
    optAb = optAb.ravel()
```

```
        tA, tb = splitAb(optAb, cset.nd)
        ts = -ones(tb.shape)
        optsol = ConSet(tA, ts, tb)
        if solver in 'bB':
            optcent = sum(optsol.vert, axis=0)/len(optsol.vert)
            itersol = ConSet(optsol.vert)
            while not itersol.allinside(cset)[0]:
                vi = (itersol.vert - optcent)*0.9999 + optcent
                itersol = ConSet(vi)
            optsol = itersol
        return optsol

    def fitcube(cset, spset, solver):
        """
        Fit a rectangle (high/low limits on outputs) within an
           existing
        constraint set.
         cset - [ConSet] existing constraint set
        """
        #### State problem
        # nvar = cset.nd
        # Constraints are bounding
        # All vertices within constraint set
        #### Define parameters
        sp = spset.b.ravel() # starting point - b matrix to
           optimise
        #### Objective fn (SLSQP)
        def objfn(Ab, *args):
            """Volume objective function for SLSQP."""
            initcs = args[0]
            A = r_[eye(initcs.nd), -eye(initcs.nd)]
            b = array([Ab]).T
            cl = con2vert(A, b)[1]
            if cl:
                closed = 1
            else:
                closed = -1
            vol = tryvol(A, b, initcs)[0]
            return closed * -vol
        #### Constraints
        def ieqconsfn(Ab, *args):
            """Optimiser inequality constraint function."""
            initcs = args[0]
```

```
    A = r_[eye(initcs.nd), -eye(initcs.nd)]
    b = array([Ab]).T
    # get vertices for constraint set iteration
    V = tryvol(A, b, initcs)[1]
    iterset = ConSet(V)
    #constraint checking for vertices
    ineqs = iterset.allinside(initcs)[1]
    return array([-linalg.norm(ineqs)])

#### Objective fn (FMIN)
def objfn2(Ab, *args):
    """Volume objective function for fmin (Simplex)."""
    initcs = args[0]
    A = r_[eye(initcs.nd), -eye(initcs.nd)]
    b = array([Ab]).T
    vol, V = tryvol(A, b, initcs)
    Pv = 200.
    #Penalties
    # points outside of init space
    iterset = ConSet(V)
    outnorm = linalg.norm(iterset.allinside(initcs)[1])
    # open shape
    cl = con2vert(A, b)[1]
    if cl:
        closed = 1
    else:
        closed = -1
    vol = tryvol(A, b, initcs)[0]
    return (-vol*closed) + Pv*(outnorm**3)

#### Maximise volume
if solver in 'aA':
    optAb = optimize.fmin_slsqp(objfn, sp, f_ieqcons=
        ieqconsfn, args=[cset],
                                iprint=0)
elif solver in 'bB':
    optAb = optimize.fmin(objfn2, sp, args=[cset],
        maxiter=50000, disp=False)
if solver in 'cC':
    optAb = optimize.fmin_cobyla(objfn, sp, ieqconsfn,
        args=[cset],
                                 iprint=0)
tA = r_[eye(cset.nd), -eye(cset.nd)]
```

```
    tb = array([optAb]).T
    ts = -ones(tb.shape)
    optsol = ConSet(tA, ts, tb)
    if solver in 'bBcC':
        optcent = sum(optsol.vert, axis=0)/len(optsol.vert)
        itersol = ConSet(optsol.vert)
        while not itersol.allinside(cset)[0]:
            vi = (itersol.vert - optcent)*0.9999 + optcent
            itersol = ConSet(vi)
        optsol = itersol
    return optsol

def fitset(cset, *args):
    """
    Return a fitted constraint set within cset.
      cset - [ConSet] to fit within
      *args (in this order) - (stype)[string] 'a'rbitrary /
          'r'ectangular
                              - (solver)[string] 'a' SLSQP /
                                  'b' fmin / 'c' Cobyla
                              - (ncon)[integer] number of
                                  constraints to fit
                                    cset.nd + 1 < ncon < cset.A
                                        .shape[0]
    """
    #Get args
    stype = args[0]
    solver = args[1]
    if stype in 'aA':
        ncon = args[2]
        nostarts = 1
    else:
        ncon = 0
        nostarts = 1
    #Starting point
    refvol = 0
    for k in range(nostarts):
        spshape = genstart(stype, cset, ncon)
        if spshape.vol() > refvol:
            finalsp = spshape
            refvol = finalsp.vol()
    #Fit set
    if stype in 'aA':
```

```
        optsol = fitshape(cset, finalsp, solver)
    elif stype in 'rR':
        optsol = fitcube(cset, finalsp, solver)
    #Return
    return optsol

def fitmaxbox(cset, sf):
    """
    Return high/low constraint set around the given
        constraint set.
      cset - [ConSet] to fit over
      sf - [integer] safety factor (fraction 0-1)
    """
    #Get args
    dev = sf*((cset.vert.max(0))-cset.cscent)
    maxvals = cset.vert.max(0) + dev#-cset.cscent) + cset.
        cscent
    minvals = cset.vert.min(0) - dev#-cset.cscent) + cset.
        cscent
    Abox = r_[eye(cset.nd), -eye(cset.nd)]
    bbox = c_[maxvals, -minvals].T
    sbox = -ones((Abox.shape[0], 1))
    return ConSet(Abox, sbox, bbox)
```

maincol.py

```
#!/usr/bin/env python
"""
Main file for M Project (Optimised MPC Constraints).
Testing file for distillation column
Author: Andre Campher
"""
# Dependencies: - SciPy
#               - convertfuns
#               - auxfuns
#               - conclasses

from conclasses import ConSet
from scipy import array, linalg
from auxfuns import mat2ab
from fitting import fitset
from convertfuns import con2pscon
```

```
#MAIN START
================================================================================


# define AIS and DOS (equations : Ax<b)
# equations in the form Ax<b, matrix = [A s b]
# with s the sign vector [1:>, -1:<]
AISA, AISs, AISb = mat2ab(array([[1., 0., 1., 11],
                                 [1., 0., -1., 15],
                                 [0., 1., 1., 25],
                                 [0., 1., -1., 90.]]))
AIS = ConSet(AISA, AISs, AISb)

DOSA, DOSs, DOSb = mat2ab(array([[1., 0., 1., 66.],
                                 [1., 0., -1., 68.],
                                 [0., 1., 1., 78.],
                                 [0., 1., -1., 82.]]))
DOS = ConSet(DOSA, DOSs, DOSb)

# define G (steady-state model)
#                R      T10sp
G = array([[-0.0575, 0.96],     # T1
           [ -0.146, 0.518]])  # T8
Gi = linalg.inv(G)

# calc AOS (from G and AIS)
lss = array([[68., 78.]]) # nominal operating point (used
    for model generation)
AOSA, AOSs, AOSb = AIS.outconlin(G, AIS.cscent, lss)
AOS = ConSet(AOSA, AOSs, AOSb)
#TODO: check nominal op.point use

# Calc intersection of AOS|DOS
DOSi = ConSet(*mat2ab(array([[-0.47486499, 0.88005866, -1,
     36.55612415],
                             [ 1.,         0.,         -1,
                                68.],
                             [ 0.,         -1.,        -1,
                                -78.]])))


# Calc additional spaces
DIS = ConSet(*DOS.outconlin(Gi, lss, AIS.cscent))
```

```
DOSn = fitset(DOSi, 'r', 'a')
DISn = ConSet(*DOSn.outconlin(Gi, lss, AIS.cscent))
DISi = ConSet(*DOSi.outconlin(Gi, lss, AIS.cscent))
DIS2 = ConSet(AIS.intersect(DIS))
DOS2 = ConSet(*DIS2.outconlin(G, AIS.cscent, lss))

# Calc modified constraints and model for high/low limits
modA, mods, modb, modG = con2pscon(DOSi, G, 'o')
```

mainflow.py

```
#!/usr/bin/env python
"""
Main file for M Project (Optimised MPC Constraints).
Testing file for level and flow rig
Author: Andre Campher
"""
# Dependencies: - SciPy
#               - convertfuns
#               - auxfuns
#               - conclasses

from conclasses import ConSet
from scipy import array, linalg
from auxfuns import mat2ab
from fitting import fitset, fitmaxbox

#MAIN START
```

```
# define AIS and DOS (equations : Ax<b)
# equations in the form Ax<b, matrix = [A s b]
# with s the sign vector [1:>, -1:<]
AISA, AISs, AISb = mat2ab(array([[1., 0., 1.,  4],
                                 [1., 0., -1., 20],
                                 [0., 1.,  1., 4],
                                 [0., 1., -1., 20.]]))
AIS = ConSet(AISA, AISs, AISb)

DOSA, DOSs, DOSb = mat2ab(array([[1., 0.,  1., 1.2],
                                 [1., 0., -1., 1.5],
                                 [0., 1.,  1., 15.],
```

```
                                        [0., 1., -1., 18.]]))
DOS = ConSet(DOSA, DOSs, DOSb)

POS = ConSet(*mat2ab(array([[1., 0., 1., 0],
                            [1., 0., -1., 1.9],
                            [0., 1., 1., 6.],
                            [0., 1., -1., 25.]])))

# define G (steady-state model)
G = array([[-0.0476, -0.0498],
           [ 0.0111, -0.0604]])

Gi = linalg.inv(G)

# calc AOS (from G and AIS)
lss = array([[1.322, 16.4048]]) # nominal operating point (
    used for model generation)
AOS = ConSet(*AIS.outconlin(G, AIS.cscent, lss))
mbox = ConSet(fitmaxbox(AOS, 0.2).intersect(POS))

# -- AOS/DOS intersection
DOSi = ConSet(AOS.intersect(DOS))
DIS = ConSet(*DOS.outconlin(Gi, lss, AIS.cscent))

# -- Fitted constraints
DOSn = fitset(DOSi, 'r', 'a')
DISn = ConSet(*DOSn.outconlin(Gi, lss, AIS.cscent))
DISi = ConSet(DIS.intersect(AIS))

print DISi
```

main.py

```
#!/usr/bin/env python
"""
Main file for M Project (Optimised MPC Constraints).
Author: Andre Campher
"""
# Dependencies: - SciPy
#               - convertfuns
#               - auxfuns
#               - conclasses
```

```
from conclasses import ConSet
from scipy import array
from auxfuns import mat2ab

#MAIN START
=============================================================================


# define AIS and DOS (equations : Ax<b)
# equations in the form Ax<b, matrix = [A s b]
# with s the sign vector [1:>, -1:<]
AISA, AISs, AISb = mat2ab(array([[1., 0., 1., -0.0525],
                                 [1., 0., -1., 0.125],
                                 [0., 1., 1., -10],
                                 [0., 1., -1., 10.]]))
AIS = ConSet(AISA, AISs, AISb)

DOSA, DOSs, DOSb = mat2ab(array([[1., 0., 1., -1.],
                                 [1., 0., -1., 1.],
                                 [0., 1., 1., -1.],
                                 [0., 1., -1., 1.]]))
DOS = ConSet(DOSA, DOSs, DOSb)

# define G (steady-state model)
lss = array([[50., 50.]]) # nominal operating point (used
    for model generation)
G = array([[1, 0.0025],
           [2, 0.0025]])  # gain matrix - atm linear steady
               state matrix

# calc AOS (from G and AIS)
AOSA, AOSs, AOSb = AIS.outconlin(G, AIS.cscent, lss)
AOS = ConSet(AOSA, AOSs, AOSb)
print AOS.vert

# Calc intersection of AOS|DOS
#print AOS.intersect(DOS)
```

References

Barber, CB (2010), "Qhull output options," URL `http://www.qhull.org/html/qh-opto.htm`, [2010, 29 October].

Barberand, CB, Dobkin, DP and Huhdanpaa, HT (1996), "The quickhull algorithm for convex hulls," *ACM Trans. on Mathematical Software*, 22 (4), 469–483.

Kelder, M (2005), "CON2VERT - constraints to vertices," MATLAB program, URL `http://www.mathworks.com/matlabcentral/fileexchange/7894`, [2010, 29 October].

Meskini, N (2009), "CGAL Python Bindings," URL `http://cgal-python.gforge.inria.fr/`, [2010, 28 October].