NTNU – Trondheim
Norwegian University of
Science and Technology

Economic Plantwide Control for a Methanol Plant using Commercial Process Simulation Software

Adriana Reyes Lua

Chemical Engineering
Submission date: June 2014
Supervisor: Sigurd Skogestad, IKP
Co-supervisor: Vladimiros Minasidis, IKP

Norwegian University of Science and Technology
Department of Chemical Engineering

NTNU – Trondheim
Norwegian University of
Science and Technology

Economic Plantwide Control for a Methanol Plant using Commercial Process Simulation Software

Adriana Reyes Lúa

Submission date:	June 2014
Responsible professor:	Prof. Sigurd Skogestad
Co-Supervisor:	Vladimiros Minasidis

Norwegian University of Science and Technology
Department of Chemical Engineering

Abstract

It is desired to make the Economic Plantwide Control design procedure proposed by Skogestad (2000, 2004, 2012) available for engineers without deep knowledge of process control or optimization. The integration of the use of commercial process simulators to obtain the process model could be a useful tool for the automation of the economic plantwide design procedure. However, process simulators are set up in "design mode" and often work poorly in "operation mode". In this thesis, the use of commercial process simulators to generate process models suitable for an automated economic plantwide control procedure is explored. The analyzed process is methanol production, as it consists of: a reactor, a separator, and a recycle stream with purge. Simulations were made in `UniSim R400 Design Suite`.

The optimization for nominal conditions and disturbed process was done using a gradient-free algorithm, implemented in Python. As the active constraint area was scanned using different tolerances, more than 1000 optimization procedures and 60000 function evaluations (simulations) were performed. Four active constraint regions were found and a self-optimizing control structure was designed for one of them. The results of the resulting control structure were satisfying, with a consistently small loss.

It was shown that the tolerance for the optimizer is an important parameter in terms of finding consistent solutions. As a matter of comparison, the optimization at nominal conditions was also performed using Matlab gradient-based NLP `fmincon` algorithm. It was demonstrated that the gradient-free solver required less function evaluations than the gradient-based algorithm. This has a positive effect on reducing the time for evaluation and for performing step 2 in the plantwide design procedure.

Preface

This thesis was written as the final work of the master program in Chemical Engineering at the Norwegian University of Science and Technology.

I would like to thank my supervisor Sigurd Skogestad for allowing me to be part of the Process Systems Engineering group and letting me work on this project, which is directly linked to one of his dearest research lines.

I would like to thank my co-supervisor Vladimiros Minasidis for his time, patience, guidance, and feedback; both during the specialization project and my master thesis. In particular I would like to recognize his enormous help with the code for the gradient-free optimization. I would also like to thank him for encouraging me to learn a very useful programming language such as Python, which I had never used before, and trying to improve my programming abilities.

I will always be grateful to my family; they have always trusted on my capacity and have always done what is in their hands to help me reach my goals.

I am also extremely grateful to Mario, who has listened to me on the hard days and supported me every day that I have worked on this thesis.

I personally acknowledge the financial support from the Mexican National Council of Science and Technology (CONACYT) and the Mexican Secretariat of Public Education (SEP) to pursue my master studies.

Declaration of Compliance:
I declare that this is an independent work according to the exam regulations of the Norwegian University of Science and Technology (NTNU).

Trondheim, Norway;
16th of June, 2014.

[illegible]

Contents

Appendices

List of Figures

List of Tables

Chapter 1 Introduction

Plantwide control refers to a control structure design for complete chemical plants. Each chemical plant is unique and there are multiple control layers and potentially an enormous number of variables involved. Moreover, it is of the interest of the industry to operate chemical plants with the best possible economical performance. Therefore, it is a challenge to develop a systematic procedure for designing control structures that achieve safe and close to optimal economic performance.

A systematic procedure for the design of an economic plantwide control system was proposed by Skogestad (2000). The main goal of this procedure is to design an optimal control structure for a complete chemical plant based on steady state plant economics. It is a stepwise procedure clearly separated into a top-down part, concerned with the steady-state economics, and a bottom-up part, mainly concerned with stabilization and pairing of loops (Skogestad, 2012).

1.1 Motivation

Downs and Skogestad (2011) mention that techniques for plantwide process control require some characteristics in order to be accepted by process engineers in the industry. These techniques must result in processes with near optimal operation but at the same time should not employ complex control technology. Moreover, the control structure design procedure should not require "the care and feeding of control experts". Minasidis et al. (2013) observed that an important step would be to develop an automated procedure, hiding unnecessary complexities.

Commercial process simulators are a standard tool for process engineers in industry. They are convenient in terms of generating the process model because the engineer can set it in a rather intuitive manner, without the express mathematical model. Besides, this type of software includes a considerable amount of useful information such as kinetic and thermodynamic data. Previously, process simulators have been used to generate the steady state process model for the plantwide control procedure, obtaining good results and insight (Brandao de Araujo, 2007; Panahi, 2011; Jacobsen, 2011).

However, the use of process simulators to generate the process model still has some pending issues regarding the implementation of the plantwide process control procedure. One drawback is that process simulators are set up in "design mode", and are used extensively for these purposes (Steimel et al., 2013). However, they often work poorly in "operation mode". Another disadvantage is that the model usually results in a large non-linear equation set with poor numerical properties for optimization (Skogestad, 2012). Furthermore, the way that the model is commonly set and optimized ends up being useful to analyze a specific case for which it was designed, and makes it difficult to extend the analysis to other models or other optimization methods. This said, the implementation of a plantwide control procedure could be simpler with the integration of the automatic design in the major process simulators (Minasidis et al., 2013).

1.2 Objective

The overall objective is to investigate if commercial process simulators could be effectively used for the automation of the plantwide control procedure. The application of this analysis is mainly on the top-down part of the plantwide control procedure.

For the case study, the methanol plant is considered to incorporate the basic structure of most chemical plants: a reactor, a separator, and a recycle stream with purge.

1.3 Previous work

For the Specialization Project, the possibility of using commercial process simulators as a tool for Skogestad's procedure was explored. A stable model for a methanol production plant was developed in `UniSim Design R400`, a commercial process simulator. This model has the features the elements of a typical chemical plant, consisting of a reactor, a separator and a recycle stream with purge. The first two steps of Skogestad's procedure were applied.

The optimization at nominal conditions and with disturbances to find active constraint regions was done using the `Matlab` gradient-based (NLP) algorithm `fmincon`. Using a central differences gradient based optimization method can prove to be costly, as the number of evaluations required to estimate the gradient is $2n + 1$. When using a commercial process simulator this means running the simulation $2n + 1$ times and if the simulation itself needs to converge on recirculations and equilibria, the overall convergence time grows. Further analysis can still be done regarding the use of different optimization methods to improve reliability, and performance in terms of calculation cost.

Derivative free optimizers have the potential to reduce computation cost. They are designed to solve problems in which the derivatives are not available or costly to calculate. However, derivative free optimization methods have some issues that need to be addressed during this work, such as effective handling of constraints, which is relevant when using process simulators.

1.4 Thesis structure

This thesis is divided in 10 chapters. Chapter 1 gives a brief introduction to the motivation and objectives of the thesis and gives a brief summary of the work done during the specialization project.

Chapter 2 includes a summary of Skogestad's plantwide control procedure, with main focus on the first three steps, which are applied in the thesis.

Chapter 3 gives an introduction of derivative-free optimization methods, focusing on (local) trust-region methods. A summary of the limitations and options to overcome them is also given.

The description of the methanol plant is included in Chapter 4. The process flow diagram, reaction scheme, and information regarding the kinetic and thermodynamic models are also included in this chapter. The second part of Chapter 4 is the description of the `UniSim` process simulation.

Chapter 5 describes the definition of the optimization problem, including the objective function terms and costs, the input constraints and the output constraints.

Chapter 6 describes the use of the optimization method described in Chapter 3 to optimize the simulation described in Chapter 4, based on the problem defined in Chapter 5.

Chapter 7 describes the first three steps of the plantwide control structure, using the results of the previous chapter.

In Chapter 8, the performance of the gradient-free solver used for the optimization is briefly evaluated and compared to the performance of the gradient-based solver.

Chapter 9 includes a final discussion of the main findings of this thesis.

Conclusions and recommendations for further work are given in Chapter 10.

Chapter 2 Overview of Plantwide Control

This section describes briefly the plantwide control procedure proposed by Skogestad (2000), as presented in Skogestad (2004) and Skogestad (2012). As short overview of the complete procedure is included, but the description will be focused on the first steps, in which is the main application of the analysis in this work.

2.1 Basic concepts

Plantwide control refers to a control structure design applied to chemical plants; specifically, to the control philosophy for the whole plant. It might be thinkable to try to formulate the mathematical problem to describe and control the whole plant. However, it would be expensive and unpractical for normal-sized chemical plants, as an acceptable control can be achieved with simpler structures.

Figure 2.1 shows the typical control hierarchy in a chemical plant. It decomposes the overall control problem on a time scale basis. The upper layers are explicitly related to economic optimization. The presented procedure deals with the two lower layers.

Basically, the control system should: stabilize the plant and implement a near-optimal operation. Stabilization occurs in the regulatory control layer, in a fast time scale and is usually done with PID controllers. It does not use degrees of freedom because its setpoints come from the supervisory (upper) layer. Supervisory control, which sends the set points to the regulatory control can be achieved with PID controllers, but MPC is currently a widespread tool. The setpoints for the supervisory control come from the optimization layer.

In the end, after the plantwide control procedure has been followed, the following decisions will be made:

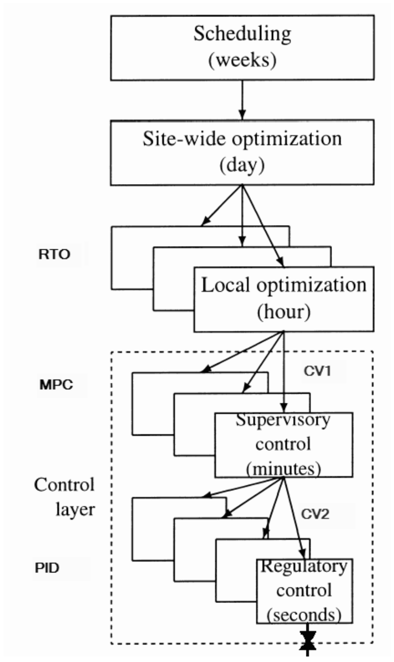

Scheduling (weeks)
Site-wide optimization (day)
RTO
Local optimization (hour)
MPC
CV1
Supervisory control (minutes)
Control layer
CV2
PID
Regulatory control (seconds)

Figure 2.1: Typical control hierarchy in a chemical plant (Skogestad, 2004).

- **Decision 1:** selection of primary controlled variables (CV_1).
- **Decision 2:** selection of secondary controlled variables (CV_2).
- **Decision 3:** location of the throughput manipulator (TPM).
- **Decision 4:** pair valves to controlled variables (CV_2).

Primary controlled variables are also called economic variables; while secondary controlled variables are also called stabilizing variables. These are sub-sets or combinations of the measured variables. The selection or combination is done using matrices H and H_2.

As Skogestad (2004) explains, to achieve a truly optimal operation, the model would need to be perfect and all the measurements should be available and reliable; which is unrealistic. Then, the concept of loss (L) is introduced as "the difference between the actual value of the cost function obtained with a specific control strategy, and the truly optimal value of the cost function". This concept brings the idea to find controlled variables such that, when keeping the setpoints constant we get an acceptable loss; even with disturbances.

This way, we would not need to be constantly reoptimizing every time that disturbances occur. This is called *"self-optimizing control"*.

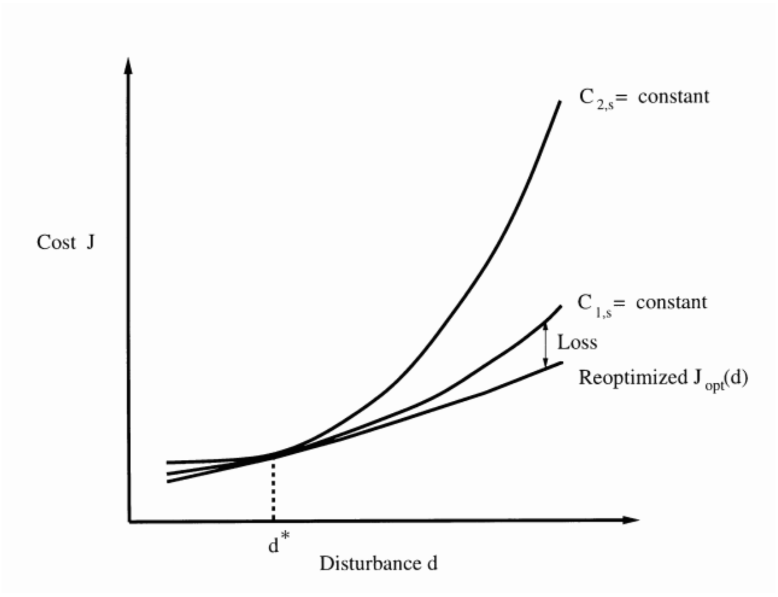


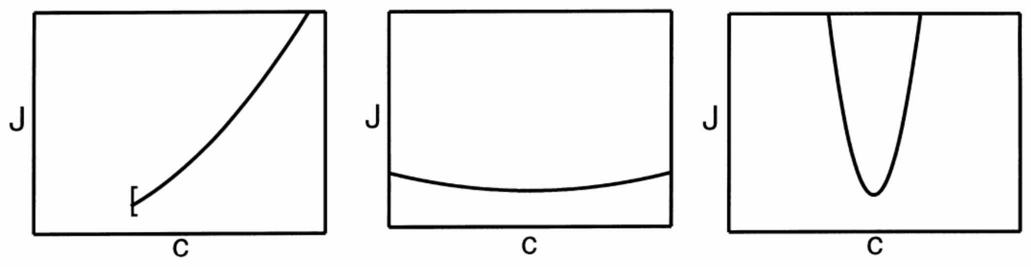
Figure 2.2: Loss imposed by keeping constant setpoint for the controlled variable (Skogestad, 2004).

Figure 2.2 illustrates the concept of loss. It can be seen that when disturbances occur, the optimum cost is also modified and if we wanted to stay at the optimum we clearly would need to reoptimize. It can also be seen that by keeping the setpoints of the controlled variables constant, there is a loss. In the figure, by keeping the setpoints of CV_1 constant, it is possible to achieve a smaller loss than by keeping constant the setpoints of CV_2.

Another factor that affects the optimum is the measurement error. The most convenient variables to keep constant are those that are active constraints or those controlled variables for which the cost is insensitive; case (a) and (b) in Figure 2.3. If the cost is very sensitive to the value of the controlled variable, implementation is harder and it may not work with a large measurement error.

2.2 Plantwide control procedure

The procedure is separated in two main parts: top-down and bottom-up. The top-down part focuses on steady-state optimal operation and economics. The bottom-up



(a) Implementation easy: Active constraint control

(b) Implementation easy: Insensitive to error in c

(c) Implementation difficult: Look for another controlled variable

Figure 2.3: Implementing the controlled variable (Skogestad, 2004).

part focuses on the control layer structure and, while the steady-state considerations are still relevant, a dynamic model is required.

The procedure is as follows:

1. **Top-down**

   – **Step 1:** Define operational objectives (economics) and constraints.

   – **Step 2:** Identify steady state degrees of freedom and determine steady state operation conditions.

   – **Step 3:** Identify candidate measurements y and select CV_1 =H y .

   – **Step 4:** Select the location of the throughput manipulator.

2. **Bottom-up**

   – **Step 5:** Select the structure of the regulatory control layer: $CV_2 = H_2$y and pairings for CV_2.

   – **Step 6:** Select the structure of the supervisory control layer.

   – **Step 7:** Select the structure for the optimization layer (RTO) - if required.

2.2.1 Procedure steps

Step 1: Define operational objectives

The operational objective is defined as a scalar cost function J (\$/s) that should be minimized. Typically:

$$J = \text{cost of feed} + \text{cost of utilities} - \text{value of products} \tag{2.1}$$

Constraints are operational constraints such as minimum and maximum flows, temperatures and pressures. Quality specifications, safety and environmental requirements should also be included here. Then, the optimization problem looks like:

$$\begin{aligned}
&\min_{u} && J(u, x, d) \\
&\text{s.t.} && \\
&\text{model equations} && f(u, x, d) = 0 \\
&\text{operational constraints} && g(u, x, d) \leq 0
\end{aligned} \tag{2.2}$$

Where u are the degrees of freedom for operation; they are "for operation" because the equipment is fixed. It is the number of *u's* that is important because it does not really matter which variables we include in u as long as they make up an independent set. The disturbances d could be changes in feed rate and composition, changes in specifications, changes in prices, among others. The internal variables (states) are denoted by x.

Step 2: Determine steady-state optimal operation

The steady-state optimization problem was defined in Step 1. The process model can be developed explicitly or it could be indirectly provided by some process simulator. The challenge of using process simulators is that the resulting model is a non-linear equation set with poor numerical properties for optimization (Skogestad, 2012).

The operational mode is chosen in this step. It can be either:

– *Mode 1:* Maximize efficiency given throughput - this results in a trade-off between valuable product recovery and energy usage.

– *Mode 2:* Maximize production - when product prices are high compared to energy and raw material's prices. In this case, the feed rate is a degree of freedom.

In order to determine the steady-state optimal operation: the degrees of freedom must be identified, the important disturbances should be identified, and the operation should be optimized, also for the disturbances. In the end, constraint regions (regions of operation with the same active constraints) should be found. In summary, in Step 2 the following should be done:

– **Identify steady state degrees of freedom**
  Here is important to differentiate between the physical degrees of freedom and the steady state degrees of freedom u. Physical degrees of freedom correspond to valves while steady state degrees of freedom are those that affect the cost function J. The former are the ones that are required for the optimization. They can be identified either by valve counting or using the potential degrees of freedom method, as described in Skogestad (2012).

– **Identify important disturbances and their expected range**
  The "importance" of a disturbance is proportional to the sensitivity of the cost function to that disturbance. Common important disturbances are the feed rate and feed composition. Other disturbances could be changes in product specification and active constraints, changes in parameters (equilibrium constants, efficiencies), and price variations.

– **Identify active constraints regions**
  Once that the disturbances and their range is specified, the function is optimized along the disturbance space. Finally, the active constraints regions are found.

Step 3: Select primary (economic) controlled variables

Each steady state degree of freedom needs to be paired with a primary controlled variable. Skogestad (2012) names two rules to select primary controlled variables:

1. **Rule 1:** control active constraints.
2. **Rule 2:** for the remaining unconstrained degrees of freedom control self-optimizing variables.

In other words, for each active constraint region, active constraints could be seen as self-optimizing variables because at the optimum they are constant. They can either be inputs or outputs. Active input constraints would mean to fully close or open a valve. Active output constraints would require a controller. As the controller would require some time to adjust after a disturbance and there will be some measurement error, the setpoint should not be exactly at the constrained value; a *back-off* is required.

After pairing the active constraints, self-optimizing controlled variables for the remaining degrees of freedom should be identified. First, candidate measurements ($\mathbf{y}$) and their expected error should be identified. Then, the primary controlled variables are selected. What is desired is to find variables for which constant setpoints give small (economic) loss when disturbances occur, and despite implementation errors. The selection of primary CVs ($\mathbf{c}$) is done using a selection or combination matrix $\mathbf{H}$, where $\mathbf{H}$=$\mathbf{c}$ $\mathbf{y}$. Some qualitative requirements mentined by Skogestad (2000) for the selection of variables are:

- The optimal value of the CV should be insensitive to disturbances.
- The CV should be easy to measure and control.
- The CV should be insensitive to manipulated variable variations.
- If there are two or more CVs, they should not be closely related.

Downs and Skogestad (2011); Skogestad (2012); Minasidis et al. (2013) discuss some quantitative approaches to define this variables; broadly classified as the "brute force approach", and "local approaches". A simple local approach is the null space method, explained by Alstad and Skogestad (2007). This method assumes that there is no noise, and the optimal constant set point can be defined as in equation 2.3.

$$\Delta c^{opt} = \boldsymbol{H}\Delta y^{opt} \tag{2.3}$$

The sensitivity $\mathbf{F}$ matrix can be defined as in equation 2.4.

$$\Delta y^{opt} = \boldsymbol{F}\Delta d \tag{2.4}$$

By combining equations 2.3 and 2.4, we get that:

$$\Delta c^{opt} = \boldsymbol{HF}\Delta d \tag{2.5}$$

If the number of measurements (n_y) is equal or larger than the sum of the number of inputs (n_u) and the number of disturbances (n_d), and $\mathbf{F}$ is evaluated with a constant active constraint set, meaning in an active constraint region, then $\mathbf{H}$ can be selected in the left null space of $\mathbf{F}$; $\mathbf{H} \in \mathcal{N}(F^T)$. Then we get equation 2.6. In other words, if we have enough measurements, we have enough information to define $\mathbf{H}$ as the null space of $\mathbf{F}$, for an active constraint region and assuming that there is no noise.

$$\boldsymbol{HF} = 0 \tag{2.6}$$

If we look at equation 2.5, it becomes evident that if $\mathbf{HF}$ is always zero, regardless the disturbances (Δd), Δc^{opt} will always be zero, meaning that the optimum value of the controlled variables will remain constant.

After the self-optimizing control structure is defined, it is possible to estimate the *Loss*, illustrated in Figure 2.2, using equation 2.7). As explained earlier, it could be seen as a penalty on the profit, when not optimizing every time that a disturbance occurs and using constant setpoints for the self-optimizing variables instead.

$$Loss = J(u; d) - J^{opt}(d) \tag{2.7}$$

Step 4: Select the location of throughput manipulator

One degree of freedom is specified as the throughput manipulator (TPM); it defines the mass moved through the plant. It could be situated anywhere in the plant, but the location will affect the economics and the structure of the regulatory system. This is the decision that links the top-down and the bottom-up part of the procedure. The TPM is further discussed by Aske and Skogestad (2009) and Skogestad (2012).

Step 5: Select the structure of the regulatory control layer

In this step the regulatory control variables CV_2=H_2y are selected; then, inputs and pairings for the CV_2 are selected. No degrees of freedom are actually used here because the setpoints for CV_2 are actually the the manipulated variables of the supervisory control layer. The selection should be done taking into account that by controlling CV_2, the effect of disturbances on CV_1 should be small (local disturbance rejection) and that the effect of disturbances on internal variables x should also be small.

Step 6: Select the structure of the supervisory control layer

There are two alternatives for this layer: advanced single loop control (PID control with some additions) or MPC. This layer must not only control the primary controlled variables CV_1 but also supervise the performance of the regulatory layer and switch the controlled variables if necessary.

Step 7: Select the structure of the optimization layer

The real time optimization layer sends the setpoints for the primary controlled variables and updates them if there are changes of the active constraint region. If self-optimizing variables are chosen, it is probable that the benefit of RTO is not so high.

Chapter 3 Overview of Derivative-Free Optimization

As the first order necessary conditions establish that for continuously differentiable functions, at a local minimum the first-order derivatives are zero, derivative information is useful information when optimizing (Nocedal and Wright, 2006). However, a very common case is that derivative information is not available, it is noisy, hence, unreliable or it is too expensive to estimate. This can be the case when the model is not a set of explicit functions, but a black-box simulation, a noisy process, or a process defined by a set of very complex equations expensive to differentiate (Conn et al., 2009). Despite these inconveniences, for a number of purposes, including the design of process control structures it is still required to carry out optimization and it is desired to do it in the most efficient manner.

As stated before, when using process simulators to build the process model, there is no set of equations to algebraically get the derivative. If used for optimizing a model without explicit equations, derivative-based algorithms require $2n+1$ function evaluations at each iteration step to approximate the derivative. When using process simulators, each function evaluation implies running the simulation. This requirement makes the optimization in step 2 of the Plantwide Control Procedure proposed by Skogestad (2000) expensive, when using process simulators.

If he model is available in the form of a simulation, it is a "black box" that returns the value of the objective function for any feasible set of inputs. For this reason, derivative-free optimization algorithms become an interesting option to try to reduce the number of function evaluations and consequently the cost of finding constraint regions.

As their name states, derivative-free algorithms do not rely on derivative information of the objective function or constraints, but exploit sample function values to build a model that is useful for the purpose. Derivative-free algorithms may or may not involve the computation of derivatives for functions other than the objective function (Ríos and Sahinidis, 2013).

3.1 Classification of derivative free optimization-algorithms

On their review of algorithms and comparison of software implementations, Ríos and Sahinidis (2013) classify this type of algorithms as:

– **Direct:** determine search directions by computing values of the objective function.

– **Model-based:** construct and utilize a surrogate model of the objective function to guide the search process.

Ríos and Sahinidis (2013) and Johnson (2008) also classify algorithms as:

– **Local:** find the local minimum, considering nearby feasible points.

– **Global:** minimize the objective over the entire feasible region; with the ability to refine the search domain arbitrarily.

A third classification given by Ríos and Sahinidis (2013) is:

– **Stochastic:** requiring random search steps.

– **Deterministic:** not requiring random search steps.

The well-known Nelder-Mead simplex algorithm is a direct local search algorithm, which remains popular due to its simplicity, flexibility and reliability (Ríos and Sahinidis, 2013). Trust-region methods, which are described in the next section, are local model-based search algorithms.

3.2 Trust-region methods

The main idea of trust-region methods is to use a model for the objective function which can be trusted in the neighborhood of the current point. This neighborhood is called the *trust region* and Δ represents the *trust region radius.* Typically, the model is quadratic, written as in equation 3.1 because it captures the curvature of functions better than linear models, as shown in Figure 3.1, and is still convenient for solving. Solving the model must be easier than solving the original problem.

$$m_k(p) = f(x_k) + p^T g_k + \frac{1}{2} p^T B_k p \tag{3.1}$$

At each iteration point x_k, p is the step, $\|p\| \leq \Delta_k$ The objective function f is approximated by a quadratic model (m_k) obtained by a second order Taylor approximation around x_k and $g_k = \nabla f(x_k)$. B_k is a symmetric approximation to the Hessian $\nabla^2 f(x_k)$. Therefore, to obtain each step, the subproblem in equation 3.2 should be solved.

$$\begin{aligned} \min_{p \in R^n} \quad & m_k(p) = f(x_k) + p^T g_k + \frac{1}{2} p^T B_k p \\ \text{s.t.} \quad & \|p\| \leq \Delta_k \end{aligned} \tag{3.2}$$

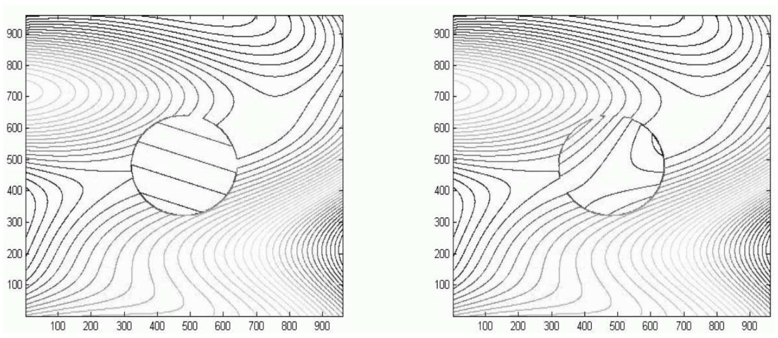


Figure 3.1: Contours of a linear model (left) and a quadratic model (right) of a nonlinear function in a trust region (Conn et al., 2009).

While line-search methods use the model to generate a search direction and then find a suitable step length, trust-region methods define a region around the current iterate within the model is trusted to be good enough and then choose a step (direction and length) to approximate the minimizer of the model in this region, as shown in Figure 3.2. In general, trust-region methods are available for derivative-based and derivative-free optimization.

Nocedal and Wright (2006) outline some advantages of trust region methods:

- These methods limit the step to a region within which the model is considered reliable.
- SQP methods do not require the Hessian matrix to be positive definite. This can be extended to methods using quadratic approximations, as in some derivative-free algorithms.
- By controlling the quality of the steps even with Hessian and Jacobian singularities, they provide a mechanism for global convergence.

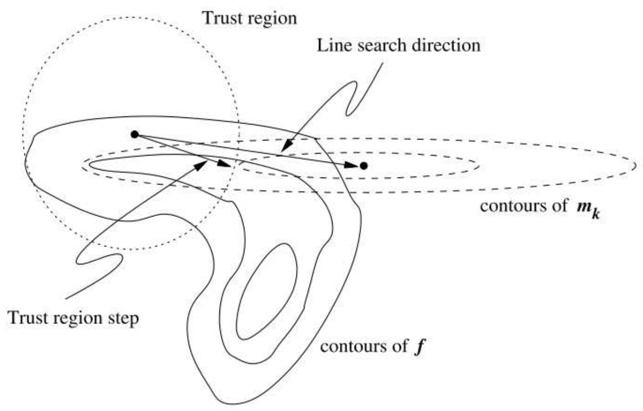


Figure 3.2: Trust-region and line search steps m_k (Nocedal and Wright, 2006).

The size of the trust region can vary, as shown in Figure 3.3. In most algorithms it is updated at each step, beginning with a bigger trust region and reducing it as the algorithm approaches the solution.

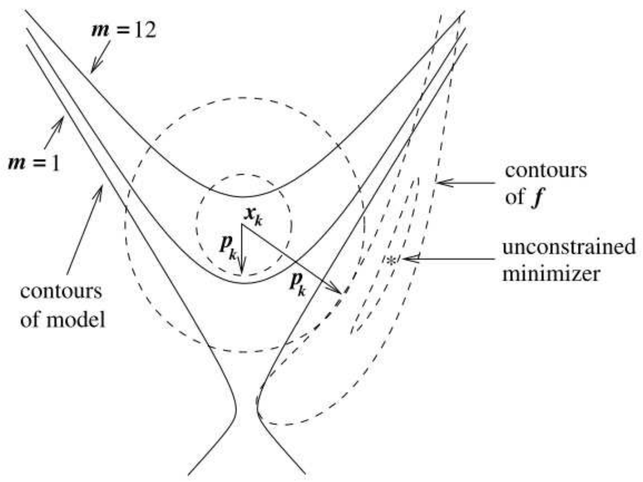


Figure 3.3: Two possible trust regions (circles) and their corresponding steps p_k . The solid lines are contours of the model function m_k (Nocedal and Wright, 2006).

As these methods add an additional constraint to the problem, the subproblems might become unfeasible and additional procedures to handle this situation need to be considered (Nocedal and Wright, 2006).

3.2.1 Derivative-free trust-region methods

Trust-region algorithms are local model-based methods. In this type of algorithms, properties such as Jacobian and Hessian information of the surrogate model are used. Typically, a high fidelity surrogate model is not initially available; therefore, these methods start by sampling the search space and building a surrogate model. Then, iteratively, the model is optimized (minimized to obtain a new trial point), the objective function is evaluated to find a solution, and the surrogate model is updated (Ríos and Sahinidis, 2013). A useful feature of some of these algorithms, in the sense of reducing function evaluations is that each iteration changes only one of the interpolation points.

If f is smooth, algorithms for unconstrained optimization are hardly ever efficient unless some attention is given to the curvature of f (Powell, 2007a). For this reason it is useful to use a quadratic approximation instead of a linear approximation, as has also been shown in 3.1. In any case, the parameters of the quadratic approximation are still obtained by interpolation.

When using a quadratic approximation the number of degrees of freedom increases to $\frac{1}{2}(n+1)(n+2)$. This would imply an important increase in the number of required function evaluations, at least to initialize the optimization. Therefore, among others, Powell (2007a, 2009) has worked on the idea of reducing the required interpolation points to $2n+1$. According to Powell (2009), this idea has given very promising results because, even with a reduced number of function evaluations, the result is not affected drastically.

Derivative-free trust-region algorithm

The algorithm that was used for this thesis is the BOBYQA (Bound Optimization BY Quadratic Approximation) algorithm proposed by Powell (2009), which is an extension of a previous algorithm, NEWUOA, to include bounds on inputs (Powell, 2003, 2007a, 2008).

This algorithm has been used previously to analyze the conceptual design and optimization of chemical processes under uncertainty (Steimel et al., 2013) and for the analysis of optimization of production of oil fields (Asadollahi et al., 2014) to try to reduce function evaluations.

BOBYQA is a model-based derivative-free algorithm that seeks the minimum of an objective function $f(x)$ with n variables as in equation 3.3.

$$\begin{aligned} \min_{x \in R^n} \quad & f(x) \\ \text{s.t.} \quad & a_i \leq x_i \leq bi,\ i = 1, 2, \ldots, n \end{aligned} \tag{3.3}$$

To achieve this, it requires m iteration points to construct the quadratic surrogate model. The number of iteration points is an integer in the range $[n+2, \frac{1}{2}(n+1)(n+2)$, being $2n+1$ a typical value.

In general, Arouxét et al. (2011) summarize the algorithm in the following steps:

Step 0: Initialization. Set the number of interpolation points, evaluate the function in these points and build an initial quadratic model.

Step 1: Solve the quadratic trust-region problem. Minimize the quadratic model, subject to the bounds, within the trust region. The solution is expected to be in the intersection of the trust-region with the bound constraints.

Step 2: Acceptance test. Update the trust region radius.

Step 3: Alternative iteration. If required, recalculate the step in order to improve the geometry of the interpolation set.

Step 4: Update the interpolation set and the quadratic model. Evaluate a new point and update model by eliminating one point and introducing the new one. The least Frobenius norm is used to update the interpolation model (Powell, 2003, 2009).

Step 5: Update the approximation. Update iteration number and return to step 1.

Step 6: Stopping criterion. Criterion can be number of iterations, or decrease of solution.

The complete algorithm and details can be consulted in the report by Powell (2009).

Besides solving bound constrained optimization problems, Powell (2009) and Arouxét et al. (2011) describe some features of this algorithm:

- Considers bounds when calculating the trust region step.
- Obtains an approximate solution for the approximation $m_k(x_k + s)$.
- The solution is in the intersection of the trust-region with the bound constraints.
- In each iteration, only one interpolation point is replaced. This means that the objective function is evaluated once per iteration.

As mentioned above, the evaluation of all the interpolation points, typically $2n+1$, is only required for initialization and afterwards only one point is updated at each iteration. Moreover, Powell (2009) suggests that for some problems this number can be decreased to $n + 2$. The model is updated by minimizing the Frobenius norm of the second derivative matrix.

Ríos and Sahinidis (2013) evaluated and compared several derivative-free optimization algorithms. Among the algorithms that were tested, BOBYQA and its unbounded version NEWUOA were the only local solvers that performed well for convex smooth problems, as global approaches performed generally better than local algorithms. Ríos and Sahinidis (2013) also found that BOBYQA could solve more than 90% of the test cases within the optimality tolerance, when the problem had up to nine variables. With more than 31 variables, this dropped to about 40%. However, a similar tendency was observed for all the tested algorithms. On the other hand, regardless the size of the problem, it solved more of 90% of non-convex smooth problems, performing better than most of the tested algorithms. In the case of non-convex non-smooth problems, it solved at least 90% of all problems when the size was nine or smaller, and performed similar or better than other algorithms with bigger sizes.

BOBYQA is a relatively young algorithm and there is ongoing work on improving its robustness and efficiency. For example, Arouxét et al. (2011) have modified Powell's algorithm to use an active set strategy to reduce function evaluations. On the other hand, some analysis of the geometry of the interpolation points of the NEWUOA algorithm were done by Fasano et al. (2009), who concluded that at some steps, the system could become ill conditioned, but overall, it performed better than methods without a geometry phase.

3.3 Limitations of derivative-free methods

Despite this type of methods has a long story, with algorithms such as the Neadler-Mead method available in 1965, most of the development has been in the last years and it is still not mature. Development of derivative-free methods is still ongoing and there are still some issues to solve. Nocedal and Wright (2006), Conn et al. (2009), and Ríos and Sahinidis (2013) mention the following areas of opportunity for derivative-free methods:

– Most algorithms can only handle "a few tens" of variables. The fraction of problems solved is reduced when the size of the problem increases.

– Accuracy of solutions is not as good as for derivative-based methods.

– Convergence is not always fast.

– There is still work to be done for the constrained problems.

– Stopping criteria is not always well-defined (if there is no derivative information, KKT conditions cannot be evaluated).

– Some algorithms have scaling problems.

After analyzing 22 derivative-free software implementations, Ríos and Sahinidis (2013) concluded that there is no single software whose performance dominates over the others. When derivative information is available and inexpensive, derivative-based methods still are preferred.

On the other hand, recent works have led to significant progress for derivative-free algorithms (Ríos and Sahinidis, 2013). For example, regarding convergence, Powell (2004, 2007b, 2009) has worked on developing algorithms to reduce the number of interpolation conditions to values as low as $n + 2$.

3.4 Constraint handling using penalty methods

Constraint handling may become an issue, not only for derivative-free optimization, but also for derivative based-optimization. One approach is to replace the original constrained problem by an unconstrained problem or a sequence of subproblems in which the constraints are represented by terms added to the objective (Nocedal and Wright, 2006).

Nocedal and Wright (2006) explain this type of methods and name the following approaches:

– *Quadratic penalty function:* adds a multiple of the square of the violation of each constraint to the objective.

– *Nonsmooth exact penalty methods:* the problem is replaced by a single unconstrained problem. The augmented Lagrangian method is one approach.

– *Log-barrier method:* logarithmic terms prevent feasible iterates from moving too close to the boundary of the feasible region. This approach is applied in interior-point methods.

As explained above, in the quadratic penalty function the penalty terms are the weighted squares of the constraint violations. As for any penalty function method, the weight given to each constraint violation is important in the sense that if it is too high, it might turn the problem an unbounded problem because the weight given to the penalty is to high compared to the objective function. On the other hand, the penalty must be large enough to avoid the iterates from going from the feasible region.

If there is a feasible solution, at the solution, equality constraints will be satisfied (equal to 0) and will not affect the value of the objective function. A more general case, considering also inequality constraints is:

$$Q(x;\mu) = f(x) + \frac{\mu}{2}\sum_{i\in\mathcal{E}} c_i^2(x) + \frac{\mu}{2}\sum_{i\in\mathcal{I}}([c_i(x)]^-)^2 \tag{3.4}$$

Where $[c_i(x)]^-$ denotes $max(-c_i(x), 0)$.

When introducing equality constraints this way, smooth problems remain smooth; and convergence is usually reached within few iterations. This allows to use unconstrained optimization methods and start with a low value for μ_k and increase it at each step. However, when introducing the term for inequality constraints, the function looses smoothness, as it has a discontinuous second derivative. In the case of derivative-free optimization, this might not be an important issue, depending on the specific algorithm.

The ℓ_1 penalty function a non-smooth penalty function also for equality constraints and the penalty term is μ times the ℓ_1 norm of the constraint violation.

$$Q(x;\mu) = f(x) + \sum_{i\in\mathcal{E}} |c_i| + \sum_{i\in\mathcal{I}}([c_i(x)]^-) \tag{3.5}$$

Other norms such as the ℓ_∞ norm or the ℓ_2 norm can be used instead of the ℓ_1 norm. When using derivative-free algorithms, the performance is not necessarily dependent of the smoothness of the problem (Ríos and Sahinidis, 2013); which makes sense if we think that derivatives are not calculated directly. Therefore, preserving smoothness could not be as important as it is when using derivative-based methods. Then, using non-smooth penalty functions can be a simple alternative to introduce constraints.

The augmented Lagrangian method estimates the Lagrangian multipliers λ to weight each constraint violation and it helps to preserve smoothness, and avoids ill conditioning (Nocedal and Wright, 2006). It can have very good convergence properties when the derivative matrices are available (Birgin and Martínez, 2007). However, it was decided to not use this method for this thesis because it was not desired to require additional calculations of derivatives during the optimization phase.

Chapter 4 Process Description

As a typical chemical plant, we consider a process consisting of a reactor, a separator and a recycle stream with purge. The methanol plant was selected because it incorporates this basic structure. In this section, the main elements of the plant as well as the simulation in `UniSim` are described.

4.1 Methanol Process Description

Synthesis gas (*syngas*) composed by hydrogen and carbon monoxide is the raw material for methanol production. Typically, syngas is produced onsite from natural gas, as shown in Figure 4.1. Syngas with some carbon dioxide is fed to the methanol production section. Crude methanol (containing water) is sent to a purifying section to produce high purity methanol ($\geq$ 99.5%) (Zhang et al., 2013).

For the purpose of this analysis, syngas production and methanol purification are not included and only the delimited section in Figure 4.1 will be considered. Syngas is considered as the feed (and disturbance) to the process and crude methanol is considered to be the product. Typical plant capacities range from 150 to 6000 t/d (Moulijn et al., 2013).

From ca. 1830 - 1923, methanol was produced by dry distillation of wood. It was first synthesized industrially in 1923 from syngas. To reach acceptable conversions, high pressure (250-350 bar) and temperatures of 320-450 °C were required. In the 1960's, the ability to produce sulfur-free syngas and new catalysts (Cu/ZnO) allowed the production of methanol at milder conditions, especially regarding pressure. "Low-pressure plants" operate at 50-100 bar and 200-300 °C. The upper temperature bound is because at higher temperatures sintering occurs (Lange, 2001; Speight, 2002; Fiedler et al., 2005; Moulijn et al., 2013).

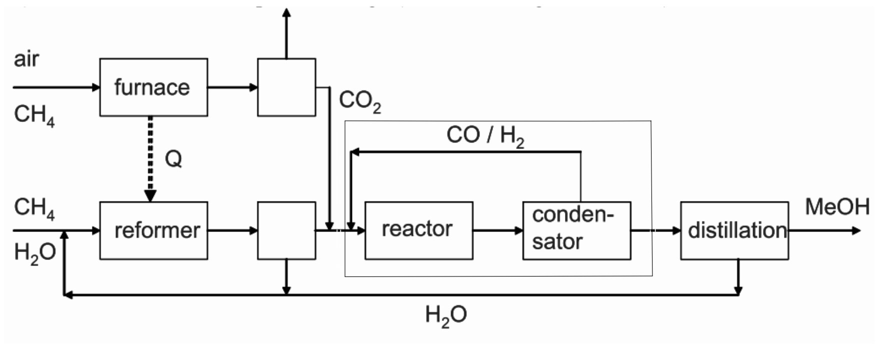


Figure 4.1: Conventional methanol process scheme, modified from Braunschweig et al. (2008).

4.1.1 Process flow diagram

Process flow diagrams for the industrial production of methanol are similar and the most important difference is the reactor. A general scheme is presented in Figure 4.2. Fresh syngas is mixed with recycled syngas and the mixture is pre-heated before entering the reactor. There is some kind of temperature control in the reactor, either by quenching or water cooling. When quenching is used as a means of cooling, as in the ICI process, the quenching flows are not pre-heated.

The reactor outlet is cooled by heat exchange with the feed and cooled further to separate methanol and water from the unreacted gas. The gas is recycled after purging a small part to keep the concentration of inert components within limits. The product of this section is called *crude methanol.* Crude methanol is purified in a distillation section, which is not considered in this analysis.

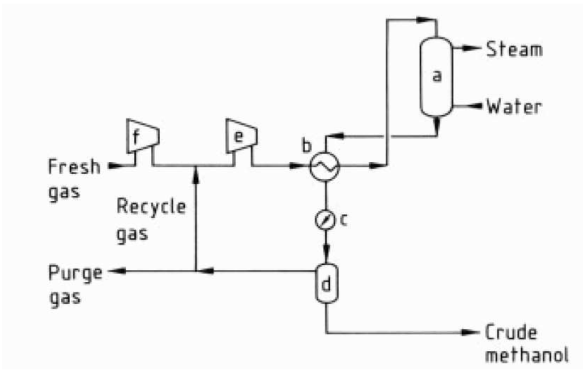


Figure 4.2: Methanol synthesis: a) reactor; b) heat exchanger; c) cooler; d) separator; e) recycle compressor; f) fresh gas compressor (Fiedler et al., 2005).

4.1.2 Reaction scheme

The main reactions for the formation of methanol are presented in equation set (4.1). Actually, only two of these three reactions are independent, and the reverse water-gas-shift reaction (4.1c) can be obtained by coupling reactions (4.1a) and (4.1b).

$$CO + 2H_2 \rightleftarrows CH_3OH \qquad \Delta H_{298\mathrm{K}} = -91 \text{ kJ/mol} \tag{4.1a}$$

$$CO_2 + 3H_2 \rightleftarrows CH_3OH + H_2O \qquad \Delta H_{298\mathrm{K}} = -50 \text{ kJ/mol} \tag{4.1b}$$

$$CO_2 + H_2 \rightleftarrows CO + H_2O \qquad \Delta H_{298\mathrm{K}} = +41 \text{ kJ/mol} \tag{4.1c}$$

The ideal syngas for methanol production has a stochiometric number (SN) of 2.05. By looking only at reaction (4.1a), a required H_2/CO ratio of 2 mol/mol can be deduced. However, the concept of stochiometric number is required because methanol is also produced through reaction (4.1b). A lower stochiometric number increases the formation of side products such as higher boiling alcohols and dimethyl ether; a lower ratio implies unreacting hydrogen.

Moreover, a small concentration of CO_2 (about 5%) increases catalyst activity (Løvik, 2001; Moulijn et al., 2013). Klier et al. (1982) found that at lower concentration of CO_2 the catalyst is deactivated by overreduction and at higher concentrations of CO_2, the synthesis is retarded by a strong adsorption of this gas. The low reaction temperatures possible with Cu/ZnO catalysts allow the high selectivity of current processes (Lange, 2001).

$$SN = \frac{H_2 - CO_2}{CO + CO_2} \tag{4.2}$$

Kinetics

A number of kinetic rate equations have been proposed in literature. Riaz et al. (2013) presented an updated summary of kinetic models for methanol synthesis. Most of the models are based on Langmuir-Hinshelwood kinetics or power law kinetics. Many models are also based on different variations of a $Cu/ZnO/Al_2O_3$ catalyst. A model that has been referred on several studies is the one proposed by Graaf et al. (1988) because it considered the three main reactions. Vanden Bussche and Froment (1996) presented a steady state kinetic model considering also the three reactions, and providing information about the catalyst.

A drawback of most published models is that the experiments on which they are based, in other words the range of validity, is up to about 50 bar (Graaf et al., 1988; Vanden Bussche and Froment, 1996; Lim et al., 2009; Riaz et al., 2013), while the operating pressure of industrial reactors is commonly around 80 bar.

4.1.3 Types of reactor

Commercial processes use a reactor with a circulation loop. Recirculation is standard because of the low single-pass conversion[1]. As the overall reaction is exothermic, quench reactors and cooled multitubular reactors, depicted in Figure 4.3, are applied.

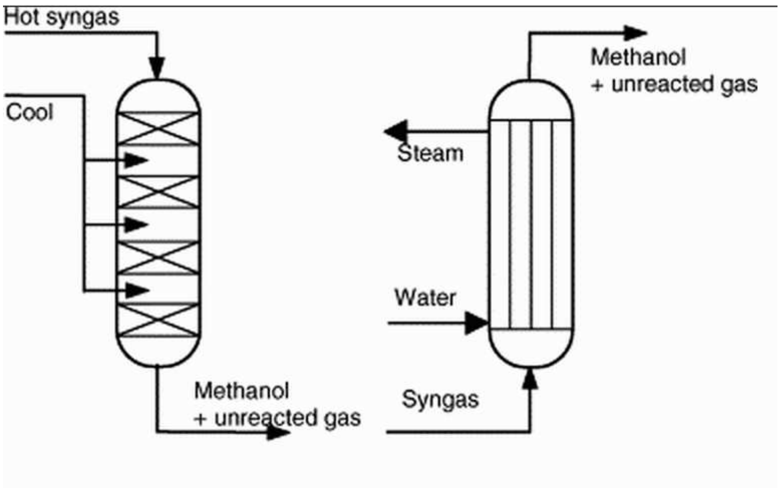


Figure 4.3: Methanol reactor types: quench (left) and multitubular (right) (Hamelinck and Faaij, 2002).

The ICI process is the most representative for the quench scheme. An adiabatic reactor is used; the reactor is a single catalyst bed and cold reactant gas at different heights of the bed is used to quench the reactor. The Lurgi process is the most representative for the multitubular scheme. Catalyst particles are located in the tubes and boiler feed water (BFW) cools the reaction, which is nearly isothermal. Although most commercial processes are two-phase processes, recently a slurry process has been developed (Lange, 2001; Moulijn et al., 2013).

4.1.4 Thermodynamic model

Modifications of Soave-Redlich-Kwong (SRK) and Peng-Robinson (PR) are widely used to calculate thermodynamic properties of hydrocarbon mixtures. SRK equation of state is reported to have a good fit for the methanol-water-carbon monoxide system. SRK or some modified version has been used for several previous studies (Løvik, 2001; Arthur, 2010; Van-Dal and Bouallou, 2013).

[1]There are some papers that discuss single pass configurations (Hamelinck and Faaij, 2002; Pellegrini et al., 2011).

While SRK is recommended for the methanol-water system, the rigorous SRK model does not predict phase equilibrium accurately enough due to the presence of the methanol-carbon dioxide system and due to the presence of hydrogen. The three compounds interact and the addition of water to methanol reduces the solubility of carbon dioxide. An extended SRK equation of state, such as the Mathias' polar correction factor, gives better results than the original SRK equation of state (Chang and Rousseau, 1985; Graaf et al., 1986; Løvik, 2001; Rostrup-Nielsen and Christiansen, 2011).

Peng-Robinson intends to improve SRK ability to predict hydrocarbons properties in the vicinity of the critical region (Ahmed, 1989). It has been proven to fit systems where low molecular weight alcohols, water and CO_2 are present (Breman and Beenackers, 1996; Joung et al., 2001; Shahrokhi and Baghmisheh, 2005; Zhang et al., 2013; Kim et al., 2013). PRSV gives a good description of nonideal systems by both enhancing pure compound vapor pressure prediction, specifically water-alcohol systems (Pellegrini et al., 2011).

Due to the non-linear increase in the solubility of carbon dioxide, most models report some deviations when approaching the critical point, as reported by Yoon et al. (1993) and Chang et al. (1997). The critical point of methanol is very close to the process conditions[2]. Chang et al. (1998) and Joung et al. (2001) analyzed the performance of Peng-Robinson to model the methanol-CO_2 system in the vicinity of the critical region for CO_2-H_2O system and concluded that performed well.

4.1.5 Circulation compressor

The circulation compressor in a methanol plant is a single-stage centrifugal compressor, with a low pressure ratio, commonly driven by a steam turbine (Lüdtke, 2004; Ohsaki et al., 2004), as depicted in the compressor in Figure 4.4. In the case of the simulation, the energy for the compressor is considered to be electrical energy.

A limitation on plant capacity (recirculated mass) is the compressor capacity. A minimum flow is required to avoid surge; a high flow would imply loss of compression capacity and probably undesired peaks in power consumption[3]. It is common that recirculation compressors in the methanol process have IGVs (inlet guide vanes), which allow a wider operating window and allow power savings (Bloch, 2006; Atlas Copco, 2011; Hitachi Plant Systems, 2012).

[2]The critical point is at 239.45°C and 80.9 bar (Chang et al., 1997).
[3]A high load might cause the compressor to trip

4.2 Process Simulation

The "operation mode" process simulation is the model. It was mostly developed during the Specialization Project, but it was modified during this thesis. The simulation was done in `UniSim R400`. It is important to keep in mind that when using this type of model the interactions among the variables are not explicit because we do not have equations to describe our model. Therefore, some process intuition and knowledge becomes important for the analysis of the interactions.

4.2.1 Process flow diagram

This analysis considers the methanol production section, with syngas as the raw material and crude methanol as the main product. After an initial analysis, the simplified process flowsheet proposed by Løvik (2001) and shown in Figure 4.4 was considered to best serve the purpose of the present study. This process considers a Lurgi reactor, in which the reaction heat is transferred to boiling water and the reactor temperature is actually controlled by the pressure of the boiling water to produce medium pressure steam. Fresh syngas is considered to have the required conditions to enter the loop.

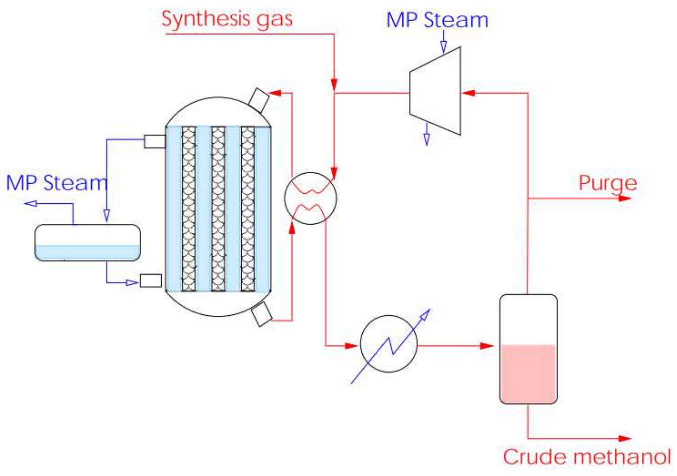


Figure 4.4: Methanol synthesis loop with Lurgi reactor (Løvik, 2001).

The simulation flow diagram is shown in Figure 4.5 and is based in Figure 4.4. Fresh syngas is fed to the loop at 140°C and the same pressure as the output of the recirculation compressor (Arthur, 2010). As the electrical energy consumption to compress the inlet (make-up) is constant and would therefore not affect the optimization, it was decided to leave it out of the simulation.

The inlet stream is mixed with the outlet of the recirculation compressor and the mixed stream is pre-heated before entering the reactor. Pressure is set upstream the reactor. The outlet stream of the reactor serves as pre-heating medium for the inlet. The outlet is further cooled to allow the separation of unreacted gas and crude methanol (methanol and water). A fraction of the gaseous stream is purged. The recirculated gas is then compressed.

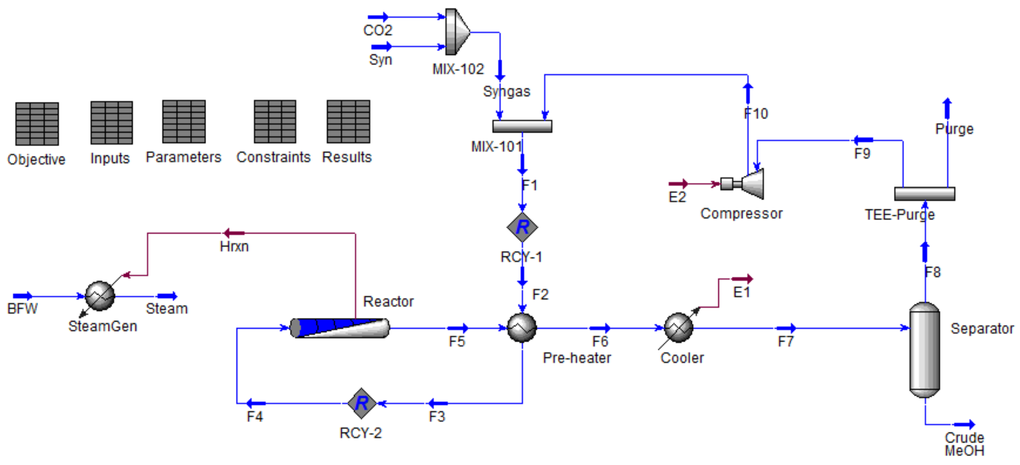


Figure 4.5: `UniSim` process flow diagram.

Typical operating temperature and pressure reported by Løvik (2001) and Arthur (2010) for Lurgi reactors are shown in Table 4.1. Fresh syngas make-up flow was set to 6000 t/d. The nominal composition of fresh syngas is shown in Table 4.2. Methane is considered to be inert.

Table 4.1: Typical operating conditions.

Parameter	Value
Reactor temperature	250 °C
Reactor pressure	80 bar
Separator temperature	45 °C

4.2.2 Reactor

In order to consider the effect of the size of the reactor on the operation a PFR model was used. Heterogeneous catalytic reactions for the hydrogenation of carbon dioxide and for the reverse water-gas-shift reaction were used to model the reactor.

Table 4.2: Nominal composition of syngas.

Component	mol fraction
Hydrogen	0.63
Carbon monoxide	0.31
Carbon dioxide	0.05
Methane	0.01

The kinetic model is the one proposed by Vanden Bussche and Froment (1996) as implemented by Arthur (2010). Besides the kinetic constants, Vanden Bussche and Froment (1996) give the information for the $Cu/ZnO/Al_2O_3$ catalyst particle, while Arthur (2010) develops the model to introduce it to the process simulator and provides the sizing of the reactor. The pressure drop through the reactor is calculated as per the Ergun equation.

Temperature control and steam generation

Temperature control is achieved by means of the production of steam, by controlling the steam pressure. For the purpose of the simulation, the temperature of the reactor is set for the outlet of the reactor (flow F_5 in Figure 4.5). The temperature of the boiler feed water (BFW) is set 10°C below the reactor temperature, to have a ΔT of 10°C in the steam generator. The outlet of the steam generator is set to be steam at the saturation point. The pressure and mass flow of steam are calculated given temperature, saturation (saturation = 1), and the available energy, which corresponds to the heat that is removed from the reactor.

4.2.3 Heat exchangers

Disregarding the heat exchange in the reactor, there are two heat exchangers in the process: the pre-heater and the cooler. The pre-heater uses part of the energy in the reactor outlet stream to pre-heat the inlet stream and, at the same time, as a first cooling of the outlet of the reactor. The cooler adjusts the temperature of the outlet to allow the separation of crude methanol from the gases. The cooler uses cooling water (CWS) as cooling medium.

Given that the intention the plantwide control procedure is to design a control structure, the heat exchange area should be fixed and the heat exchange capacity should be limited within a certain range. The value of *UA* of the pre-heater at the nominal point was calculated and then set as constant in the *Specs* tab of the unit in the simulation.

In the specialization project it was noted that the most difficult issue for convergence of this heat exchanger was the possibility of temperature crossing on the hot side (F_3 - F_5). For the specialization project, this situation was solved by adding an additional slack variable to represent the temperature difference and adding a constraint to keep it feasible.

For the thesis, this situation was solved differently, without the requirement of additional variables and constraints. In order to facilitate convergence, the temperature difference on the hot side was added as an additional *Spec* for the heat exchanger. By adding an "estimated" negative value, the calculations start approaching far from the temperature crossing and eventually converge. A screenshot of this set-up is found in figure A.4, in appendix **??**.

The Cooler is modeled as a simple cooler, without information of the cooling media in the simulation. However, the "real" heat exchanger has a maximum cooling capacity given by the maximum flow of cooling water and a minimum capacity. These two bounds are implemented as inequality constraints.

4.2.4 Separator

The gas-liquid separator is modeled as a two-phase separator without pressure drop or change of temperature. The cooling is done in the cooler described in section 4.2.3. The separator has one inlet, a liquid outlet and a vapor outlet. The vapor phase carries most of the non-condensable gas, while the liquid phase is crude methanol and contains most of the methanol and the produced water.

4.2.5 Purge

The purge is separated through a purge TEE. The recycle ratio is set as the recycled flow to the purged flow; when the recycle ratio increases, the flow in the recirculation increases. As the purge has some value, it is considered that it can be "sold" to produce some energy. This will be further explained in Chapter 5.

The calculation of the price [USD/kg] of the purge is made in the "Objective" spreadsheet, considering the cost, density, and LHV of the fresh syngas and the density and LHV of the purge. As the composition and LHV of the purge vary among simulations, this value is calculated for each simulation as per equation 4.3.

$$\mathrm{Cost_{purge}}\ [\$/\mathrm{kg}] = \mathrm{Cost_{syngas}}\ [\$/\mathrm{kg}]\frac{\rho_{syngas}[\mathrm{kg/m^3}]LHV_{purge}[\mathrm{MJ/m^3}]}{\rho_{purge}[\mathrm{kg/m^3}]LHV_{syngas}[\mathrm{MJ/m^3}]} \tag{4.3}$$

LHV and density values are all at 15°C. The cost of the syngas is constant.

4.2.6 Circulation compressor

For the sake of simplicity in the simulation, the recirculation compressor is considered to be a unit that increases pressure and no IGVs or curves were implemented. It is considered that it is driven by electricity.

In order to introduce consistency in the costs, a cost for the electricity is calculated, assuming that there is a power generation plant with 50% efficiency that uses a fuel with the same characteristics of the purge. The cost of the electricity that drives the compressor is calculated as per equation 4.4.

$$\text{Cost}_{\text{electricity}}\ [\$/\text{kWh}] = \text{Cost}_{\text{purge}}[\$/\text{kg}]\frac{\rho_{purge}[\text{kg/m}^3]3600[\text{s/h}]}{0.5LHV_{purge}[\text{MJ/m}^3]1000[\text{kJ/MJ}]} \tag{4.4}$$

As the cost of the purge is variable, the cost of the electricity is also variable.

Chapter 5 Optimization Problem

The problem can be seen as an optimization problem, formulated as:

$$\begin{aligned} \min_{u} \quad & J(u,x,d) \\ \text{s.t.} \quad & l_b \le u \le u_b \\ & g(u,x,d) \le 0 \\ & c(u,x,d) = 0 \end{aligned} \tag{5.1}$$

Where u refers to inputs or decision variables, d are the disturbances, and x refers to internal variables such as the thermodynamic model or the kinetic model.

When not using simulators, model equations are set as constraints; however, when using process simulators, model equations are not required to be defined explicitly. Then, the constraints are operational or quality constraints. The operational mode is *Mode I*; the throughput is given and variations of the feed rate are considered to be disturbances.

5.1 Objective function

The operational objective to be maximized is given by the profit, defined in equation 5.2. As the optimization problem is set as a minimization problem, the objective function considers as positive the cost of cooling water, electricity, syngas, while the cost of crude methanol (product), the sold purge, and produced steam are set as negative.

$$J = C_{syn}\dot{m}_{syn} + C_{CWS}\dot{H}_{CWS} + C_e\dot{H}_e - C_{steam}\dot{m}_{steam} - C_{purge}\dot{m}_{purge} - C_{MeOH}\dot{m}_{MeOH} \tag{5.2}$$

Costs for the objective function

The cost of raw material and the price of crude methanol used for the optimization are shown in Table 5.1. Pellegrini et al. (2011) report the price of crude methanol, which, as expected, is lower than the price of high purity methanol reported by Zhang et al. (2013) and Methanex (2013).

Table 5.1: Costs of methanol and syngas. Source: (Pellegrini et al., 2011; Noureldin et al., 2013)

Symbol	Variable	Cost	Unit
C_{MeOH}	Crude methanol	0.204	\$/kg
C_{syngas}	Syngas (2:1)	0.150	\$/kg

The purge is required to keep the inert methane concentration within acceptable limits, considering that the size of the equipment is given. As the purge has some energy content, the cost of the purge is calculated in the simulation, as explained in section 4.2.5, using equation 4.3, which considers the $\mathrm{LHV}_{15^\circ\mathrm{C}}$ of the purge relative to the $\mathrm{LHV}_{15^\circ\mathrm{C}}$ of the syngas and the price of syngas. At nominal operating conditions, the price of the purge is 0.10 \$/kg[1].

The cost for cooling water and steam is shown in Table 5.2. Cooling water is considered to be an inexpensive service[2]. Steam is produced using the heat of reaction. The price of steam is reported by Noureldin et al. (2013) as 0.008 \$/kg.

Table 5.2: Costs of cooling water and steam. Sources:(Pellegrini et al., 2011; Zhang et al., 2013; Noureldin et al., 2013)

Symbol	Variable	Cost	Unit
C_{CWS}	Cooling Water (CWS)	0.1000	\$/GJ
C_{steam}	Steam	0.0008	\$/kg

The cost of electric energy is calculated in the simulation, as explained in section 4.2.6, using equation 4.4. At nominal conditions, the price is 0.054 \$/kWh [3].

[1] Noureldin et al. (2013) used a similar method to calculate the cost of (1:1) syngas, which has a similar composition as the purge, and reported 0.080 \$/kg.

[2] The cost of using demineralized water (DMW) instead of cooling water supply (CWS) would be in the range of 0.5 \$/GJ, considering 0.021 \$/treported by Pellegrini et al. (2011) and a ΔT of 10°C.

[3] The cost for the Norwegian industry is in the range of 0.05 \$/kWh (Statistics Norway, 2013). Pellegrini et al. (2011) report a cost of 0.032 \$/kWh for Saudi Arabia.

5.2 Constraints

Constraints were set accordingly to a physical plant. `UniSim` is a design program. If physical constraints are not set, every simulation it will size the equipment accordingly to mass and energy flows. The idea of this work is not to design an optimal plant in terms of sizing, but to analyze the behavior of a given plant when designing a plantwide control structure. Therefore, there are two main sources for the constraints: those coming from operating values typical for the (methanol) process and those coming from the capacity and limitations of the specific plant.

To exemplify this, any methanol reactor has a similar maximum operating temperature, which is between 240-260°C according to Aasberg Petersen et al. (2011). This is because every methanol reactor uses a similar type of catalyst that will sinter above a certain temperature (about 280-300 °C). However, the areas of the heat exchangers of each methanol plant are not necessarily the same among plants and will limit the operating window of each specific plant.

The sizing of the equipment physical characteristics of the plant as a source for constraints can be observed in:

- **Compressor and recirculation loop capacity:** Constraints have to be set to define lower and upper bounds on the compressing capacity; either Δp, or flow. These should be treated as hard constraints because they must be always satisfied in order to assure convergence on the process simulator. Also related to the recirculating capacity and the size of the pipeline, valves, and compressor, there is a minimum and a maximum recirculation ratio and therefore, the purge flow is also bounded.

- **Cooler:** It is simulated as a simple heat exchanger that withdraws energy, and the cooling water is not explicitly defined. The outlet temperature is varied within a lower and an upper bound. The range of variation is small and should be possible with a real heat exchanger. Additionally, a range for the retired energy was set as a soft constraint. The result is physically feasible because given a Δ T in the cooling water and assuming a constant *UA*, there is a maximum and a minimum flow of water through the pipeline. Therefore, there is a maximum and minimum heat removal capacity.

- **Pre-cooler heat exchanger:** Considering *UA* as constant, and given the heat capacities, inlet temperature, and the flows of both streams, there are no actual degrees of freedom. *UA* could be set as an equality constraint. For this thesis it was set as constant in the simulation, as explained in section 4.2.3.

In the case of the reactor, the size is given and no additional explicit constraints are required to be defined for the optimization model. Implicitly, the kinetic model and the sizing are set, as explained in section 4.2.2.

Supposing that the design and sizing of a plant is given, the operating variables can be seen as inputs (decision variables) for the optimization. These can be varied within certain bounds, given in Table 5.3. These are constraints in the form $l_b \leq u \leq u_b$.

Table 5.3: Lower and upper bounds for inputs.

Input	Unit	Lower bound	Upper bound
Reactor temperature	°C	240	260
Reactor pressure	bar	75	81
Separator temperature	°C	40	50
Recirculation ratio	-	0.90	0.99

There are four degrees of freedom.

For some inputs, such as the temperature of the reactor, the lower and upper bounds can be easily defined based on process knowledge. However, as the behavior of the process is not linear, it can be the case that the bounds for some inputs are not so evident. If the input bounds are not defined correctly, some input combinations might violate physical principles.

The optimization algorithms that were used vary the inputs only within the lower and upper bounds. Therefore, if there is a hard constraint, in the sense that it requires to be satisfied to assure the convergence of the process simulator, it must be included as an input along with those in Table 5.3.

Once that combinations that would lead to an error in the process simulator have been handled as inputs, there are solutions that will not stop the process simulator from converging but would be unfeasible in a real plant or undesired. The "non-desired" solutions are discarded by introducing inequality constraints in the form $g(u, x, d) \leq 0$. From the point of view of the set-up, these are treated as soft constraints, because they are outputs and the optimization algorithm might violate them at some point. If the optimization problem is well posed and the optimization algorithm works correctly, the solution will not violate these constraints.

For example, `UniSim` allows valves to increase pressure and coolers to increase temperature; a constraint on Δp can be set. In the same line, another source of soft constraints is related to the quality of the product. Having a low quality will not avoid the process simulator to converge. In this case, a lower limit on the concentration of methanol in the product is set.

The inequality constraints ($g(u, x, d) \leq 0$) for the problem are:

$$0.80 - x_{MeOH} \leq 0 \tag{5.3a}$$

$$x_{CH_4} - 0.10 \leq 0 \tag{5.3b}$$

$$0.75 * \dot{H}^*_{cooler} - \dot{H}_{cooler} \leq 0 \tag{5.3c}$$

$$-1.25 * \dot{H}^*_{cooler} + \dot{H}_{cooler} \leq 0 \tag{5.3d}$$

Where:

- x_{MeOH}: mol fraction of methanol in crude methanol stream.
- x_{CH_4}: mol fraction of methane (inert) in recycle stream.
- $\dot{H}^*_{cooler}$: nominal heat flow removed from the cooler.
- $\dot{H}_{cooler}$: actual heat flow removed from the cooler.

It should be difficult to violate constraints (5.3c) and (5.3d), due to the bounds on the outlet temperature of the cooler.

There are no explicit equality constraints ($c(u, x, d) = 0$) in the model. Implicitly, pre-cooler's UA is set constant, but this is done in the simulation.

Chapter 6
Optimization of the Model

Once that the simulation (model) was set in `UniSim` and the optimization problem was defined, the objective function and the constraints were introduced in spreadsheets so that the values could be read and modified by an external program by means of the Component Object Model (COM) interface. Additionally, inputs and disturbances were exposed also through spreadsheets. This way, only the external program handles the optimization while the process is completely simulated by `UniSim`.

The Component Object Model (COM) interface allows the use of a code in an automation client to interact with `UniSim`. The code can access exposed objects, making possible to send and receive information to and from `UniSim` (Oli Systems, 2007). It is important to outline that not every `UniSim` property is exposed through the COM interface. However, this was overcome using `UniSim` spreadsheets to access the values. This way, even properties that are not available through the COM interface become accessible as they become "values" in the spreadsheets. This gives valuable flexibility for the definition of a robust optimization problem and allows a cleaner communication.

It is important to stress that the way that the model and solvers were set, they are formulated independently. `UniSim` and the external program communicate in a very clean way. For the solver, the simulation variables are mere numeric values. On the other hand, if the process simulation is seen as the NLP problem, the inputs are only modified at specific pre-defined locations of the spreadsheets.

6.1 Setting `UniSim` for optimization

The values for the optimization problem described in chapter 5 were added to four dedicated spreadsheets. The values that are required for the optimization problem, such as the objective function, the inputs, the constraints, the parameters (disturbances), and the measurements are written in specific cells, which are defined in the external code. The layout of the spreadsheets can be consulted in Appendix A.

The spreadsheets are:

- **Objective:** contains the objective function in cell A2, which is the only cell in this spreadsheet that interacts with the external program. The rest of the spreadsheet is used to get the values from the simulation, introduce the costs and perform any other calculation that is required to to calculate the objective function.
- **Inputs:** columns A-E are read by `Matlab`. Decision variables set as constraints in the form $l_b \leq u \leq u_b$ are set here. Columns A through E contain: the actual value of the input, the lower bound, the upper bound, the initial value for the optimization, and the units of measurement. Column F is not read or modified by the external program, but is used to describe the decision variable.
- **Constraints:** inequality constraints are specified in column A. Equality constraints are specified in column B. The rest of the columns are used to get the values and perform the required calculations.
- **Parameters:** the disturbances are exposed in column A. This allows the external program to modify their values and test the model response to disturbances.
- **Measurements:** measurements candidate to be controlled variables are exposed in this spreadsheet, giving flexibility to save the results and perform the analysis.

Posing the optimization problem in a way that assures convergence is a very important step before using the external program to optimize the problem with and without disturbances. This has been described in section 4.2. In order to assure that the optimization result is consistent and constant among runs, the recycle sensitivities were increased[1]. The maximum number of iterations was also increased to assure convergence.

6.2 Gradient-based solver

NLP methods have been used previously for the methanol process with Lurgi reactors to analyze the effect of varying operating conditions on a process plant. Kralj and Glavič (2009) used a gradient method for a methanol process because, despite a global optimal solution is not guaranteed, it gives good results for complex processes.

[1] As explained in `UniSim` manual, the entered sensitivities values serve as a multiplier for `UniSim` internal convergence tolerances (Unisim, 2007).

For the specialization project and this thesis, the gradient-based optimization was done using the NLP solver `fmincon` in `Matlab`. The interior-point algorithm was used and gradients are estimated using finite differences. The interior point algorithm assures that the input constraints are always satisfied, reducing the possibility of unwittingly evaluating unfeasible points.

Process variables such as compositions, temperatures, and flows are in different orders of magnitude. To facilitate convergence, two scaling functions were written. One function scales from zero to one the input values read from `UniSim` and these scaled values are used in `fmincom`; the other function "de-scales" the variables to send them back to `Unisim`.

A feature of the `Matlab` code (appendix B.2.1) is that it reads the number of bounded inputs, equality and inequality constraints. With this information, the NLP code (Appendix B.2.2) is generated. This way, there is no need to re-write the code for optimization.

6.3 Gradient-free solver

In the specialization project it was noted that the gradient-based optimization algorithm required many iterations to find the solution. This was explained with the fact that the model is not explicit, not linear, and noisy. For this reason, it was decided to explore the possibility of using a gradient-free optimization algorithm to perform the optimization.

The gradient-free algorithm that was implemented is BOBYQA (Bound Optimization BY Quadratic Approximation), developed by Powell (2009). A very simplified version of the building interpolation step was developed initially, but afterwards it was decided to use the code available in the NLopt open-source library for nonlinear optimization (Johnson, 2008). NLopt is written in C but is callable from other programming languages. For this thesis, the interface was Python 2.7.

Similar to gradient-based optimization case, a code was written to interact with the simulation and the optimization algorithm. This program reads the number of inputs and their bounds, normalizes them, and varies their values to evaluate the objective function at different points. The code is included in appendix B.1.1 and B.1.2. If required it can also read and set the values of the disturbances, as shown in Figure 6.1.

An important difference between the gradient-based and the gradient-free optimization programs is that while `fmincon` is set to handle input and output constraints, BOBYQA only handles input constraints. In order to handle output

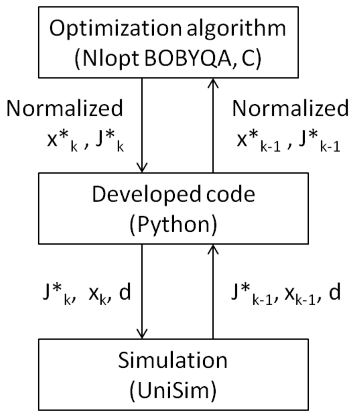

Optimization algorithm (Nlopt BOBYQA, C)
Normalized x^*_k , J^*_k
Normalized x^*_{k-1} , J^*_{k-1}
Developed code (Python)
J^*_k, x_k, d
J^*_{k-1}, x_{k-1}, d
Simulation (UniSim)

Figure 6.1: Interactions between optimization algorithm and simulation.

constraints, the objective function was modified in the simulation by adding a penalty function, as described in section 3.4.

The values for the penalties were established according to the order of magnitude of the variable and the order of magnitude of the objective function. The absolute value of the maximum mole fraction of methane in the recirculation and the minimum mole fraction of methanol in the product, will always be lower than 1. Therefore, in order to be relevant, the penalty was in the order of 10^6- 10^9. In the other hand, the heat removed from the cooler is in the order of magnitude of 10^8. Therefore, the penalty was set in the order of 10^3. These values worked correctly for this particular problem and the solutions that the optimizer found did not violate the constraints.

The ℓ_1 penalty function (equation 3.5) and the quadratic penalty function (equation 3.4) were tested. Both worked fine for the purpose of the problem, as long as the problem was initially feasible. If the optimization had been started in a feasible point and the optimization algorithm evaluated an unfeasible point in which the output constraints were violated, the algorithm returned to the feasible region in the next iteration. However, if the optimization started in an infeasible point, the algorithm had problems finding the feasible region.

Chapter 7

Design of control structure

In this section the first three steps of the plantwide control procedure are performed systematically using the tools developed in the previous chapters.

The goal of this analysis is to establish the basis to use a process simulator such as `UniSim` to systematically apply the plantwide control procedure. The first three steps of the plantwide procedure presented in Skogestad (2012) were applied: the definition of the operational objectives, the steady state optimization, and the identification of candidate measurements. In order to perform the steady state optimization in Step 2, a reliable steady state model must be available. The simulation described in section 4.2 and the code and modifications described in chapter 6 were used to solve the optimization problem in chapter 5.

7.1 Step 1: Define operational objectives

Chapter 5 describes the definition of the optimization problem. Equation 5.2 defines the objective function, equation set 5.3 defines the inequality constraints for the outputs and Table 5.3 details the lower and upper bounds for the inputs. The goal is to minimize the cost (maximize profit) satisfying operational constraints.

7.2 Step 2: Determine steady state optimal operation

This step is usually very time consuming (Skogestad, 2012), and this work is done with the intention of helping to automatize it.

7.2.1 Identification of steady-state degrees of freedom

There are five physical degrees of freedom (valves), as shown in Figure 7.1; but there are only four steady-state degrees of freedom.

In Figure 7.1 there is a valve to control the level in the separator. This adds a physical degree of freedom, but would not have any steady state effect. Therefore, there are four steady-state degrees of freedom. The valve to control the level in the separator was not included in the simulation (Figure 4.5).

In this case, the "evident" physical degrees of freedom in the simulation correspond to the steady state degrees of freedom. This could be seen as a result of the fact that the simulation was set for a steady-state analysis.

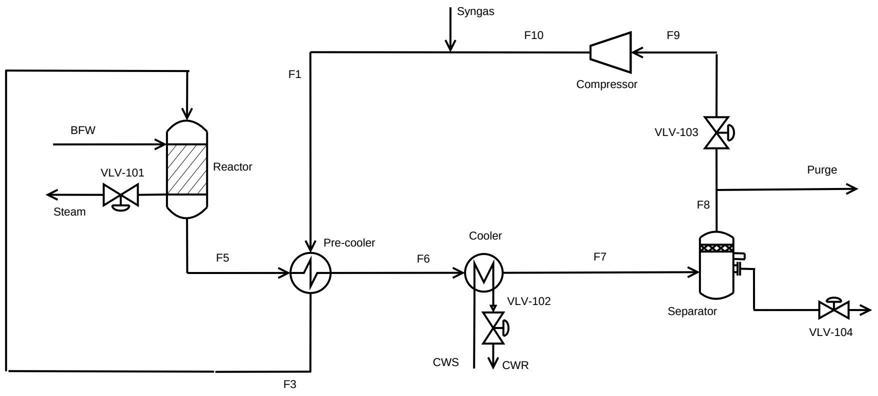


Figure 7.1: Process flow diagram.

Using the potential steady-state degrees of freedom method (Skogestad, 2012), we obtain the same result:

- splitter: 1 (TEE-Purge; *n=2; n-1=1*)
- heat exchanger: 2 (Cooler and the generation of steam in the reactor)
- pressure: 1 (Compressor)

Figure 7.1 shows how the actual manipulation of the process parameters would be done (there is a valve/compressor for each degree of freedom). The reactor is in gas phase, and it is not a degree of freedom. The pre-cooler does not add a degree of freedom because both flows and the size are given and there is no bypass.

7.2.2 Identification of important disturbances

Two important disturbances are in the feed: the flow and the composition of fresh syngas. It was defined that the H_2:CO:CO_2 ratio in the syngas would be kept constant and that the mole fraction of the inert component (methane) would be the disturbance.

$$d = [\dot{m}_{syn}, x_{CH_4}] \tag{7.1}$$

7.2.3 Optimization

The process was optimized using the NLP solver and simulation as explained in chapter 6. The nominal syngas make-up flow is 24 000 kmol/h (about 6000 t/dor 250 000 kg/h) and the nominal composition is the one in Table 4.2. It was defined that syngas flow could vary ±10%. Keeping the H_2:CO:CO_2 ratio constant. The composition of CH_4 was varied ±10%. The optimum values are shown in Figure 7.2.

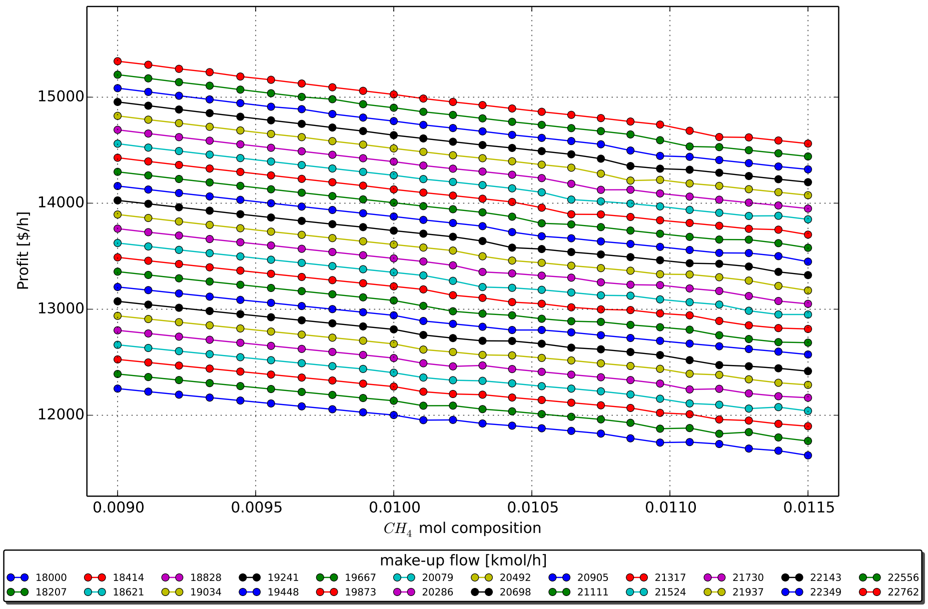


Figure 7.2: Effect of disturbances (x_{CH4} and fresh syngas make-up) on solution.

From the contour plot in figure 7.3 and the 3D plot in figure 7.4 the same trends as described above can be observed.

7.2.4 Operating regions

Constraints are defined in section 5.2. Input constraints are identified in Table 7.1, while output constraints are identified in Table 7.2. Optimization was performed

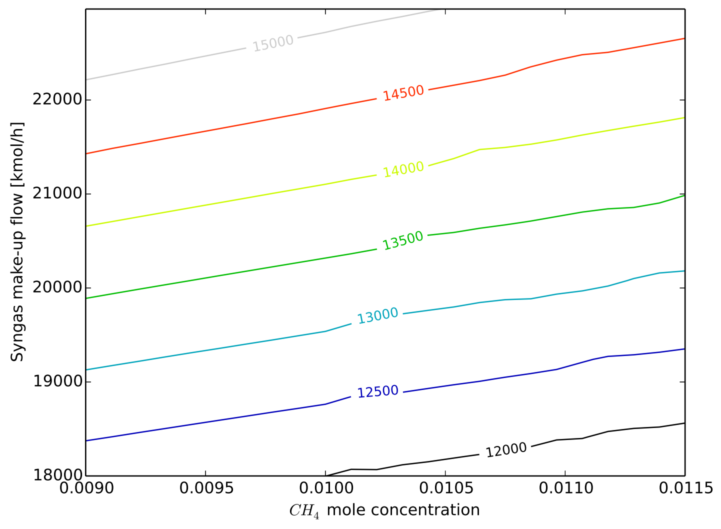


Figure 7.3: Contour plot for the different optimum values as function of disturbances.

varying 1% from -10% to +10% of nominal flow and methane composition. It can be pinpointed that along these optimization procedures, the simulation worked fine.

Table 7.1: Input inequality constraints identification

Constraint	Variable	Description
c1	F5 temperature	Reactor temperature
c2	F4 pressure	Reactor pressure
c3	F7 temperature	Cooler outlet and separator temperature
c4	TEE-Purge ratio	Recycle ratio (relative to purge)

Table 7.3 shows the active constraints in each of the identified regions depicted in Figure 7.5. For input constraints, that have a lower and upper bound, the subscript indicates whether it is the upper or the lower bound that is active.

Some observations regarding the constraint regions are:

- $c5$ is active all over the area.
- $c2$ (operating pressure) activates on its upper bound frequently.
- $c3$ (cooler temperature) activates on its lower bound frequently.

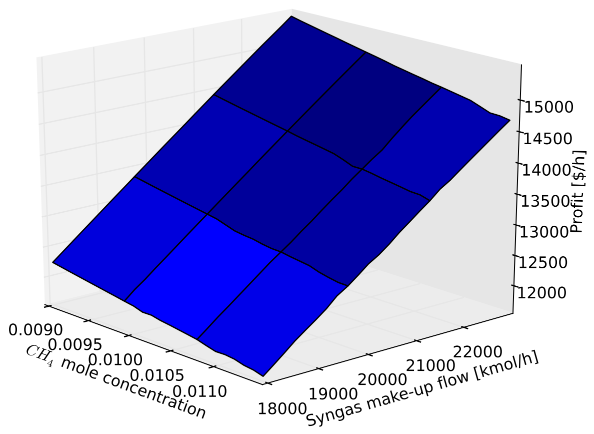


Figure 7.4: Effect of disturbances on optimum.

Table 7.2: Output inequality constraints identification

Constraint	**Variable**
c5	MeOH minimum mole concentration in crude methanol
c6	Methane maximum mole concentration in recycle
c7	Cooler low capacity
c8	Cooler high capacity

Table 7.3: Identification of active constraint regions.

I	c5
II	$c3_{lower}$, c5
III	$c2_{upper}$,c5
IV	$c2_{upper}$, $c3_{lower}$, c5

– In region I, where only $c5$ is active, the values of $c2$ and $c3$ are actually very close to their bounds.

As it will be discussed in chapter 9, this constraint area map was made over a rather small disturbance span. However, it serves as a "close-up" for the performance of the gradient-free optimizer and the results that can be obtained. In the case of

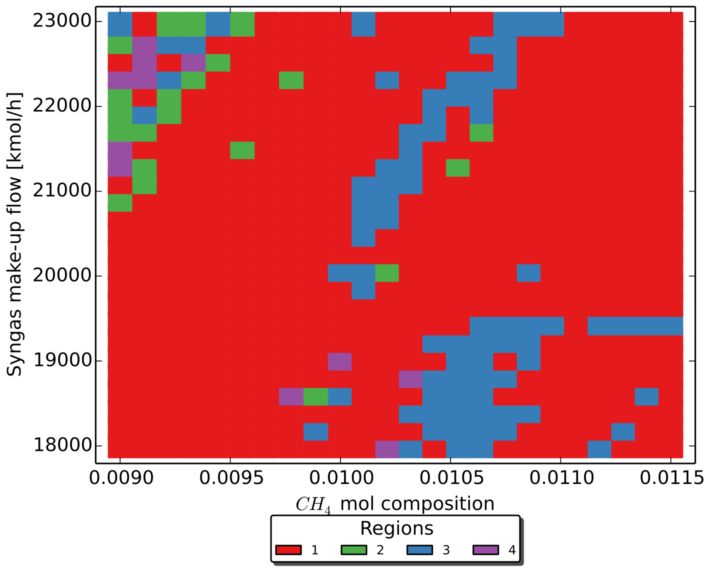


Figure 7.5: Active constraint regions.

this particular problem, figures 7.3 and 7.4 show that in this area, despite the model is not linear, the optimum was rather flat. Given the constraints, the optimum all over the analyzed span was close to the bounds of three of the four inputs (reactor temperature upper bound, operating pressure upper bound, cooler temperature lower bound. This situation made the optimization harder for the algorithm, as it was constantly limited. It is fair to mention that for this active constraint map, 600 optimization procedures and 49887 function evaluations were performed.

7.3 Step 3: Select primary (economic) controlled variables

It is noteworthy to remember the requirements that Skogestad (2000) outlines to select the controlled variables. The optimal value of c should be insensitive to disturbances:

– The optimal value of c should be insensitive to disturbances.

– c should be easy to measure and control (so that the implementation error is acceptable).

– The value of c should be sensitive to changes in the manipulated variables.

– For cases with more than one unconstrained degrees of freedom, the selected controlled variables should be independent.

Original input variables are:
$u^T = [u_1, u_2, u_3, u_4] = [T_{reactor}, P_{reactor}, T_{cooler}, \text{recirculation ratio}]$

The control structure is designed for region III, with one output and one input constraint active. Active constraints are:

– Reactor pressure is bounded at its upper value (*input constraint*).

– The concentration of methane in the recirculation is at its maximum value (*output constraint*).

7.3.1 Pairing of constrained variables

As mentioned in section 2.2.1, active input constraints would mean to fully close or open a valve. Active output constraints would require a controller.

As the reactor pressure is an active constraint at the upper bound and an input, it means that the compressor should be operated at its maximum capacity from the point of view of pressure. It should be noted that for this decision, no considerations regarding the possible time delay between the reactor pressure and the compressor or pressure losses in the pipeline were taken into account.

The concentration of methane in the recirculation, which is an output constraint, requires a controller, that could be a feedback controller. Based on the "pair close" rule, we chose to use the recirculation ratio to control the concentration of methane. In this analysis, we are not considering the implementation error or noise. If that was the case, *back-off* would be required as a safety margin. The resulting process flow diagram is shown in figure 7.6.

7.3.2 Selection of self-optimizing controlled variables for the remaining degrees of freedom

At this point, from the initial four degrees of freedom, there are two remaining degrees of freedom, represented by $T_{reactor}$ and T_{cooler}. It is noteworthy to say that in the analyzed region, the optimum value of the temperature of the reactor is very close to the upper bound (260°C), and that the optimum value of the temperature of the cooler is also close to the lower bound (30°C).

During the optimization in step 2, several measurements (y) were saved and were available to calculate Δy^{opt} and $\mathbf{F}$. For the purpose of this analysis, it is

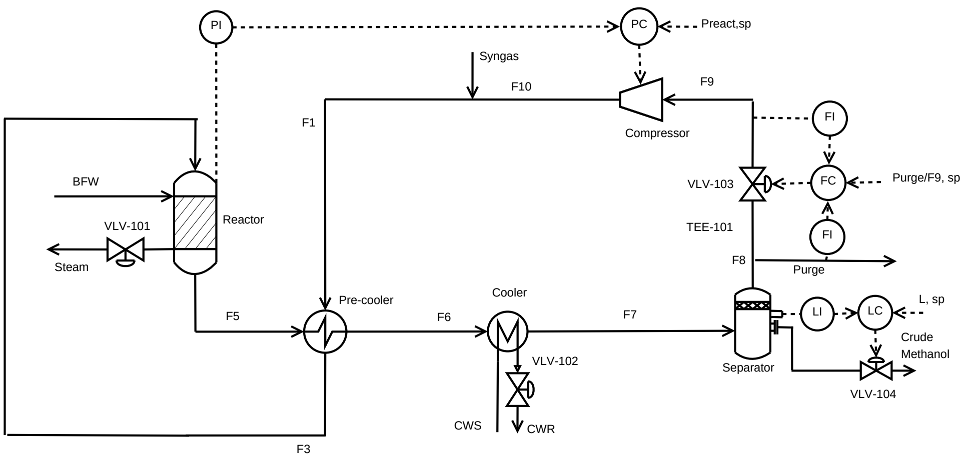


Figure 7.6: Process flow diagram with pairings for constrained variables.

considered that there is no noise. Alstad and Skogestad (2007) mention that "in the measurement vector y, input vector u is generally included, including the inputs that have been selected to the control active constraints. They add that the measurements of the active constraints should not be included in y, because they are constant and, provide no information about the operation".

As we require $n_y \geq n_u + n_d$ to be satisfied we need at least $2+2$ measured variables. We have $11 > 2+2$. Then, we can take decision 1 of the plantwide control procedure, which is to select the primary controlled variables $\boldsymbol{CV_1} = \boldsymbol{H}y$. This requires finding **H**, which can be full or not. The null space method, as described in section 2.2.1, is used in this case. Selected candidate measurements are:

$$y^T_{candidate} = [P_{steam}(\text{bar}), T_{cooler}(^\circ\text{C}), Flow_{purge}(\text{t/h}), F_{reactor}(\text{t/h})] \tag{7.2}$$

The dimension of **F** is $n_y \times n_d$= 4 x 2, and defined as:

$$F = \begin{pmatrix} \frac{\partial y_1^{opt}}{\partial d_1} & \frac{\partial y_1^{opt}}{\partial d_2} \\ \frac{\partial y_2^{opt}}{\partial d_1} & \frac{\partial y_2^{opt}}{\partial d_2} \\ \frac{\partial y_3^{opt}}{\partial d_1} & \frac{\partial y_3^{opt}}{\partial d_2} \\ \frac{\partial y_4^{opt}}{\partial d_1} & \frac{\partial y_4^{opt}}{\partial d_2} \end{pmatrix}$$

Commonly, the elements in **F** are approximated linearly as in equation 7.3, where h

is the step size.

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{7.3}$$

As the measurements are available, instead of using one direction for the disturbance, the elements in **F** will be calculated using central differences centered formula of order $O(h^4)$, as in equation 7.4.

$$f'(x) \approx \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h} \tag{7.4}$$

The point $d = [18414\text{kmol/h}, 0.0105]$ is established as the initial (nominal) operating condition for this analysis. The other evaluation points that will be used are based on $\Delta d = [\pm 0.207\text{tmol/h}, \pm 0.01\%\text{mol}]$ and $\Delta d = [\pm 0.414\text{tmol/h}, \pm 0.02\%\text{mol}]$. The units of measurement were adjusted to have both in the same order of magnitude. It is understood that this is a rather small disturbance; however, this analysis is done as a mere demonstration. With this information the numerical values of **F** are:

$$\boldsymbol{F} = \begin{pmatrix} -2.8094 & -89.5144 \\ 0.9365 & -17.5066 \\ 1.2831 & 20.4067 \\ 15.1951 & -135.1277 \end{pmatrix}$$

The dimension of **H** is $n_u \times n_y$= 2 x 4. It the null space of **F** and numerically is[1]:

$$\boldsymbol{H} = \begin{pmatrix} 0.299 & -0.3061 & 0.9038 & -0.0022 \\ 0.2721 & 0.9349 & 0.2265 & -0.0264 \end{pmatrix}$$

With this information, the self-optimizing control was implemented in `UniSim`, by adding an "Adjust" operation to "control" the active constraint and another two "Adjust" operations to use the two remaining degrees of freedom to control the self-optimizing control variables. This is shown in figures A.7 and A.8 in appendix A.

Figure 7.7 depicts the optimized profit (objective function) and the profit resulting of the self-optimizing control, both as function of the magnitude of the total disturbance ($\sqrt{d_1^2 + d_2^2}$). This could be seen as a "worst-case scenario" because it would mean that both disturbances occur at the same time. The figure does not show the section in which the self-optimizing control and the re-optimized profit are equal, but it shows that the self-optimizing control-structure follows closely the re-optimized result. Figure 7.8 depicts the Loss, which, as expected, is relatively small in terms of the magnitude of the optimum values.

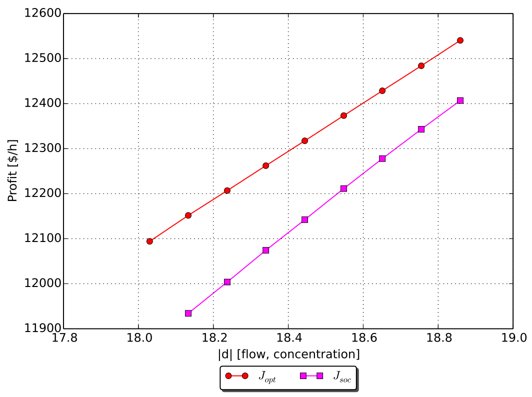


Figure 7.7: Optimized profit and self-optimizing control profit

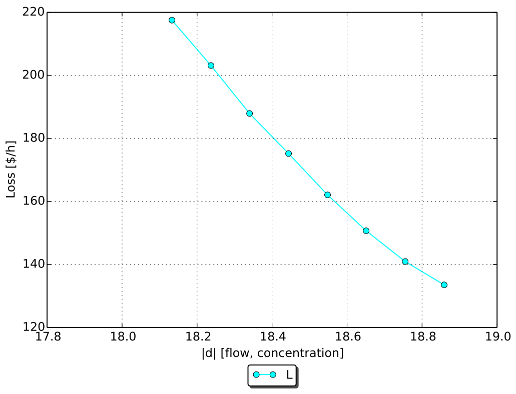


Figure 7.8: Loss from self-optimizing control

An observation that needs to be mentioned is that, given the set-up of this control structure, when using the self optimizing control, at the highest disturbance the self-optimizing control structure became unfeasible, as both degrees of freedom (reactor temperature and cooler temperature) reached their bounds without finding a solution. This could be effect of the fact that even if the inputs where not strictly constrained, their values at the solution were very close to their upper or lower bounds.

[1]H was calculated in Matlab using H=null(F.')'

Chapter 8 Performance of the Gradient-Free Solver

In this thesis the use gradient-free solvers to perform the optimization of models created in process simulators is analyzed. In order to get a picture of the performance of this type of solver, some basic statistics are presented. Additionally, it is compared with the gradient-based solver used for the specialization project.

8.1 Gradient-free solver overall performance

Standard convergence analysis methods are not commonly applied to gradient-free optimizers. Therefore, the performance of the used solver will be analyzed from the point of view of number of function evaluations, time to solve the optimization, and ability to find the solution.

8.1.1 Number of function evaluations and time to solve optimization

In general function evaluations depend on the type of processor and, as can be seen in figure 8.1, on the tolerance. Initially, the simulations were run with tolerance 10^{-6}. However, it was observed that results were not consistent because there were several "constraint areas" with only one or two elements. Afterwards, it was decided to decrease the tolerance to 10^{-8} and use the results of these simulations for the design of the control structure.

Just as a point of comparison, the time to solve the optimization is shown in figure 8.2. As it has already been mentioned, it depends on the processor, other processes carried out with the computer at the same time, and even programming. However, it is included to give a reference of the time that it was required to solve this particular problem and the time that would be required to solve other problems. There were three outliers that took more than 10000 seconds to find a solution that are not depicted. Similar to the number of evaluations, it is observed that it is not a normal distribution, as it is skewed to the right. Moreover, there is a bound on the

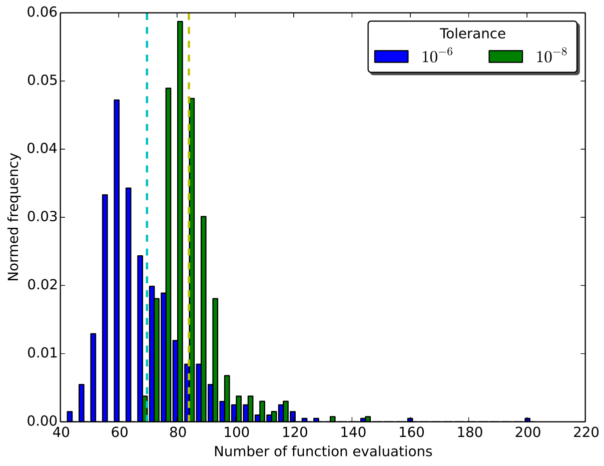


Figure 8.1: Number of function evaluations, with tolerance 10^{-6} and 10^{-8}.

minimum, meaning that, even for simple optimizations or those that start close to the optimum, the number of evaluations and time for optimization do not decrease substantially.

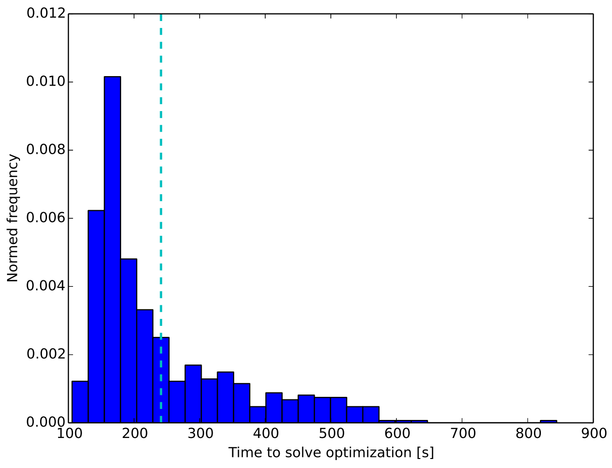


Figure 8.2: Time [s] to solve the optimization with tolerance 1×10^{-8}.

8.2 Finding the solution

As mentioned earlier, initially the scan for the constraint areas was done using a tolerance of 1×10^{-6}. However, this resulted in several regions with very dispersed elements and very few elements in each region, as can be observed in figure 8.3, in which there are two elements in "region VII". Moreover, the contour plot, shown in figure 8.4, was rather uneven. For this reason, it was decided to re-run the optimizations with a lower tolerance (1×10^{-8}).

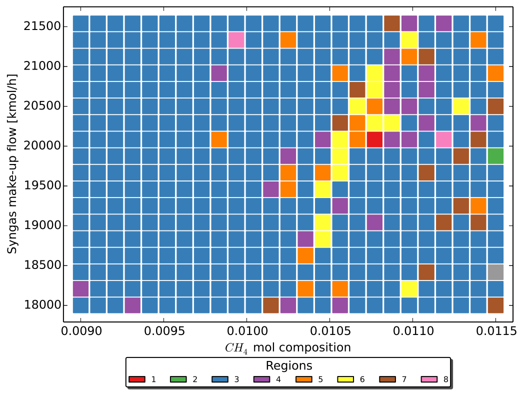


Figure 8.3: Constraint areas found with tolerance 10^{-6}. These areas were not used for the design of the control structure.

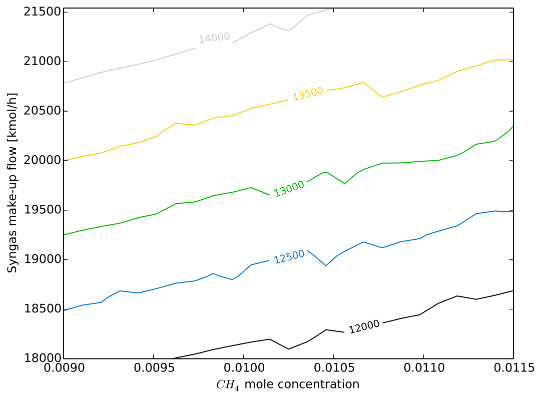


Figure 8.4: Contour plot found with tolerance 10^{-6}.

8.3 Comparison of optimization strategies

In order to evaluate the convenience of using a derivative-free algorithm against using a derivative-based algorithm, both algorithms (BOBYQA and `fmincon`) were tested. Both algorithms were tested with the same simulation. The only difference was that, `fmincon` has the option of reading and considering inequality constraints ($g(u, x, d) \leq 0$) and the gradient-free algorithm does not. Therefore, when the optimization was performed by `fmincon`, the optimization function was not modified and equation 5.2 was used. When the gradient-free algorithm was used, a penalty term was added to the objective function, as described in section 6.3.

It should be reminded that:

- both algorithms are local optimization algorithms and that none claims to be a global optimizer;
- for each iteration step, the derivative based algorithm requires $2n + 1$ function evaluations.

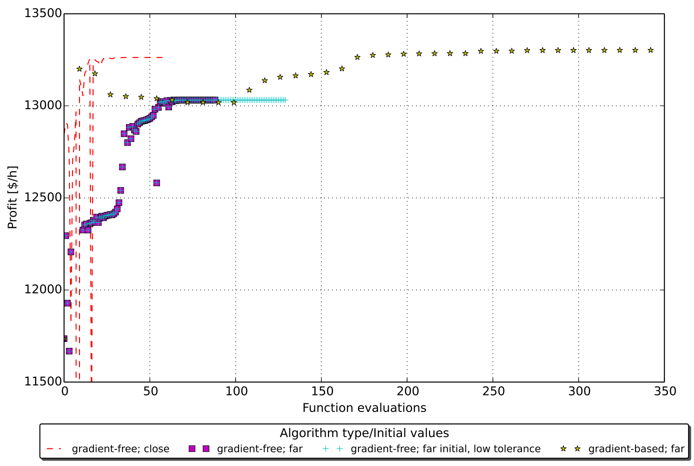


Figure 8.5: Comparison of performance of gradient-free and gradient-based algorithms at nominal conditions, with different initial points.

Figure 8.5 shows that if it is initialized far from the solution, the derivative-based method has a better initial step than the derivative-based because the second value is very close to the optimum. It then moves away remains almost 50 evaluations in the same values (around 13000$/h). As it has the information of better results, it keeps iterating and does many more iterations before going back to values close to the ones it had found initially, and find the optimum at around 13200$/h

On the other hand, the derivative-based algorithm advances more slowly towards the optimum stops at fewer evaluations, and never "gets the information" about the existence of a "better" optimum. Therefore, after "falling into" the valley around 3000 $/hit stops at that local optimum, after 89 iterations.

The test described above was done with a tolerance of 10^{-6}. To analyze if this had happened because the tolerance was not low enough, the tolerance was reduced to 10^{-8}. It can be observed that for this particular case, the algorithm followed the same path and stayed at the same result, only with more iterations, as shown in Table 8.1.

If initialized closer to the optimum, the derivative based algorithm does not "loose track" and finds the "better" optimum in 61 iterations, fewer than the derivative-based algorithm. In the figure it can also be observed that the algorithm evaluated values that were unfeasible, and the penalty function penalized those results. It can also be observed that when this happened, the algorithm returned to the feasible region.

Table 8.1: Performance of derivative-based and derivative-free algorithms on the analyzed problem.

Type	Initial	Tolerance	# steps	# eval	Time [s]	Time [s]/ eval
Derivative-based	far	1.0E-08	39	351	2260	6.4
Derivative-free	far	1.0E-08	111	111	467	4.2
Derivative-free	far	1.0E-06	89	89	396	4.4
Derivative-free	close	1.0E-06	61	61	200	3.3

Chapter 9
Discussion

Together with obtaining the model, the optimization step is often the most time consuming in the plantwide control procedure (Skogestad, 2012). The work presented in this report seeks to facilitate the integration of process simulators in the automatic design of the economic plantwide control procedure. Some insight about what is required from the process model for optimization, mainly applicable to the first three steps of the procedure, was obtained.

9.1 On the setting of the optimization model

The optimization model was initially developed for the specialization project; it was developed to be used for optimization with Matlab `fmincon` NLP solver. Initially during the specialization project, it was intended to handle operating constraints solely by adding inequality constraints (equation set 5.3). However, NLP solvers evaluate and vary input parameters in the process simulator and then verifies whether the output constraints are satisfied or not. It was then noted that when some constraints are violated, the process simulator still converges. For example, when the inequality constraint that assures that the methanol quality is satisfied, the simulation still works; the "only" issue is that the solution does not serve the purpose of the analysis. However, in the case of other variables, if an unfeasible value is set in the simulator it will not converge and fail. The problems should be formulated such that variables are be set as inputs.

Knowledge and insight of the process is a very powerful tool and well selected boundaries are very important to assure convergence of the simulator. From the point of view of the definition of the optimization problem, it is important to assure that the boundaries of the inputs give results that will converge. For this reason, it is recommended to set the hard constraints as inputs u or decision variables (Inputs spreadsheet), because they need to be satisfied at every moment to assure convergence in the simulator. Then, the soft constraints, which are not absolutely necessary for the convergence of the simulator should be set as equality and inequality constraints

(Constraints spreadsheet). This way, these will help to discern between the solutions that satisfy requirements such as product quality or some equipment sizing.

From the procedure point of view, the number of degrees of freedom is what matters, and which variables we include in u is not really important Skogestad (2012). However, from the simulation point of view, an appropriate selection of variables, significantly improves robustness. In the end, the manipulated variables in control might be different from the inputs required in the process simulator. Using spreadsheets as means of communication with the optimization solver facilitates the clear distinction of inputs and outputs and eliminates possible errors.

9.1.1 Unit operations in the simulation

The reactor is a plug-flow reactor with a defined size, giving implicit limitations to the model. In general, it converged easily and behaved as expected. When the model was optimized the temperature of the reactor was in the high range and close to the upper bound.

The pre-cooler does not give a degree of freedom because it does not have a bypass. For this same reason, it is a unit operation that is difficult to converge. In the specialization project, a decision variable was used to adjust the temperatures so that the heat transfer area (UA) was kept constant. Despite setting a constraint for LMDT that would assure a feasible heat exchanger, the optimization procedure did not avoid the process simulator to "try" solutions with temperature crossings. In those cases, the process simulator failed to converge and stopped. This was solved by using the temperature difference between the hot inlet and the cold outlet instead of the temperature of the hot variable as a decision variable to find the temperatures in the pre-cooler. This was implemented by calculating the cold outlet temperature based on a given ΔT and the temperature of the hot inlet in an additional spreadsheet.

As the addition of variables implies a bigger optimization problem, an alternative solution was explored for the thesis. It was found that by setting the value of UA in the simulation and giving an appropriate estimate for the initial temperature approach, the simulation was also robust enough for the purposes of simulation, and the requirement of extra variables was eliminated. If the number of variables is reduced, the number of function evaluations that the optimization problem requires is reduced. However, a matter of further analysis would be if the time for the simulation does not increase with this modification.

On the other hand, the cooler was modeled as a simple heat exchanger, and the size is limited with output constraints in terms of cooling capacity. As the cooling water circuit is not included in the main simulation, this approach is quite efficient and converges easily, making it a very appropriate option.

9.2 On the simulation results

The objective of this analysis was to explore the use of a commercial process simulator such as `UniSim` to design the control structure for a chemical plant. This procedure involves optimization and most of the identified challenges are related to the stability of the simulation when varying the inputs and disturbing the process. In the end, the objective was reached because the simulation ran smoothly enough when disturbed and optimized. During the optimization phase of the plantwide control procedure, more than 1000 optimization results were obtained. Considering that the average number of function evaluations was between 60 and 80 (depending on the tolerance), the simulation was run and converged more than 60 000 times.

The level of detail of the simulation was set trying to approximate the limitations of a real plant. As discussed earlier, in general, the behavior of the equipment and results were accordingly to the expected. If the simulation was used to design a plant, most probably the simulation results would be different. For a deeper analysis, the level of detail would require to be increased. For example, despite the kinetic model that was implemented, proposed by Vanden Bussche and Froment (1996), behaved as expected in the sense that the optimum temperature and pressure of the reactor were consistently either at the upper bound or close to it, it was developed for pressures between 15 and 51 bar(as many other models), while the simulation is run at pressures in the range of 80 bar. Other aspects that could be modeled with more detail are: heat exchangers' pressure drops (which were set constant at 10 mbar) and heat transfer coefficients, as well as the compressor curves.

An effort was put on setting the costs for the objective function at real values. For this reason, the costs for purge and electricity are updated at each simulation, considering the LHV (lower heating value) and energy ratio with respect to natural gas. This way, it can be assured that the costs are within a reasonable magnitude. Additionally, these costs were compared to commercial values to assure that the value was reasonable.

9.3 On the optimization methods

By using the process simulator, there are no explicit equations that model the behavior of the process. This simplifies the modeling step of the optimization procedure. However, in a standard optimization problem, when the equations for the behavior of the parameters of the problem are given, before starting to actually solve the problem it should be defined whether it is convex or not in the area of interest. Chemical processes are mostly non-linear and it is possible that operating conditions are in a non-convex area.

It has to be mentioned that both optimization methods are local optimization methods, meaning that there is no complete scan of the optimization function. Therefore, there is no obvious way to find out if the result is a global minimum; it can only be assured that it is a local minimum. Moreover, it is possible that the solution is highly dependent on the initial inputs.

In the specialization project and this thesis, (`fmincon`) was the gradient-based NLP solver used to optimize the problem. The results were good in the sense that convergence was reached and the optimization results were consistent. However, it has to be reminded that for each evaluation the gradient has to be calculated. To do this, the simulation needs to be run $2n + 1$ times. As the simulation does not converge immediately in some regions, this makes the algorithm quite slow. The time to converge for `fmincon` varies between 5-15 minutes (at least about 300 seconds and up to 900 seconds). It has to be mentioned that tolerances were small, to assure that the results were consistent given the high non-linearity of the process. The time to converge should not be a significant problem if a limited number of conditions is to be tested and if the analysis is to be done offline. However, in the specialization project this area of opportunity was found.

For the reasons mentioned above, the approach to use derivative-free optimization methods was analyzed during this thesis. These methods have the advantage that are designed to solve problems in which the derivatives are not available or costly to calculate, which is actually the case when using a process simulator. In the other hand, many current algorithms are effective only for relatively small problems and the effective handling of constraints is still under investigation, as explained in section 3.3. As the process simulator convergence is affected by the appropriate handling of the constraints, especially the bounds for the inputs, this was an issue to be solved during this thesis.

The gradient-free algorithm BOBYQA (Powell, 2009), as implemented by NLopt open-source library for nonlinear optimization (Johnson, 2008), was used. As mentioned in chapter 8, this gradient-free algorithm performed well. However, as mentioned in the same chapter, the repetibility of the results is still a matter that requires attention. In some cases, the optimization had to be re-run to obtain a consistent result. Moreover, it was required to set a very low tolerance to get relatively good results. However, even with the used tolerance, the appearance of the active constraint area is not very uniform and could still be a matter of the capacity of the optimizer. This is consistent with the results of the analysis made by Ríos and Sahinidis (2013). They mention that despite BOBYQA and its non bounded version NEWUOA are among the local derivative-free algorithms that perform better, sometimes they do not find the optimum and do not have an excellent refinement ability.

In general, the gradient-free solver required less evaluations than the gradient-based solver, confirming the potential of this type of solvers to speed-up the optimization phase of the plantwide control procedure. While the gradient-based solver frequently required more than 100 evaluations (in the specialization project), and it was not rare that it required more than 400 evaluations, as shown in section 8.3, the gradient-free solver rarely required more than 150 evaluations, even with low tolerances.

Despite this potential, big area of opportunity for this optimization algorithm would be a "warm-start" routine. It was noted that, despite starting at the optimum value, the optimization algorithm required more than $2n + 1$ function evaluations to "realize" that it was at the optimum. The lowest numbers of function evaluations were in the range of 30, while in the case of the analyzed problem $2n + 1$ would be 9 evaluations.

However, considering the issues on the reliability of the results, there is still work to be done. The results must be reliable because important decisions for the design of the control structure are based on them.

9.4 On the optimization, active constraints, and self optimizing control

The constraint that limits the concentration of the inert in the recirculation (not of the product) remains active in all the evaluated regions. This might seem as "not-optimal" because it is a constraint fixed by the user and the appropriate correct number could be a matter of discussion. However, as this procedure is intended to be applied in an existing plant, it is feasible that physical limitations are in place. For example, that some mechanical elements would not stand a maximum concentration of one component. Another explanation would be that the used models, for example the kinetic models, loose reliability at certain conditions. It is accepted that it could be a matter of analysis to release this constraint and study the effect of this action on the active constraint regions.

On the other hand, three out of the four input constraints, namely the temperature of the reactor, operating pressure, and temperature of the cooler, were consistently close to the optimum. Reactor pressure reached its bound frequently and was the second most common constrained situation.

The optimization phase gave enough information for the design of the control structure, based on self-optimizing variables selected by using the null-space method. The step-wise procedure was followed and the control structure was designed and tested in one constraint region. The test was done in the "worst-case scenario", meaning that both disturbances occurred at the same time (both d_1 and d_2 decreased or increased). The designed self-optimizing control was good in the sense that the

Loss was small and more or less constant and that J_{soc} followed J_{opt} without requiring additional optimization, and, thus, allowing an acceptable operation of the plant, even with disturbances.

An improvement that was made regarding the standard way to perform the null space method was the estimation of the elements in $\mathbf{F}$. During the optimization step it was cheap to obtain information of the measurements. Therefore, afterwards, all this information was available to estimate the sensibility matrix. As the values of the optimized measurements at several steps were available, instead of a linear approximation, the central differences centered formula of order $O(h^4)$ was used to estimate $\partial y^{opt}/\partial d_i$.

It has to be noted that, given the shape of the constraint areas, the designed control structure only works for a very small range of disturbances, which might not be practical in many industrial cases. This example justifies the requirement to continue working on the design of control structures that work in wider areas.

Chapter 10 Conclusion

The use of commercial process simulators could be an important step to develop an automated procedure for designing control structures that achieve safe and close to optimal economic performance. Hiding unnecessary complexities would ease the acceptance of the plantwide control procedure by process engineers in the industry. However, the use of process simulators to generate the model still has unresolved issues. The plantwide control procedure intends to design the control structure of a given chemical plant, not to design the plant. Commercial simulators are set up in "design mode" and often work poorly in "operation mode". Also, there is no standardized way to set up the simulators for the optimization in Step 2 - and to be used for the rest of the steps.

In this thesis, the use of `UniSim`, one of the most popular commercial simulators, was explored. The analyzed process was a methanol plant, that includes the basic features of a typical chemical plant: a reactor, a separator, and a recycle stream with purge. This way, relevant issues of this configuration were analyzed. The model included sizing of equipment such as the reactor and heat exchangers. This way, the simulation could be run in "operation mode".

As capacity constraints were introduced, special attention was put on the convergence of an integrated heat exchanger. Quality constraints such as minimum methanol concentration in the product or maximum inert concentration were also introduced. In the end, a robust simulation was obtained. With the simulation and optimization algorithm, active constraint regions were identified.

It became evident that a clear definition of input and output constraints is important to achieve a robust simulation. The simulation needs to be robust to avoid it from crashing during the optimization procedure. The optimization algorithm varies the inputs, and cannot "predict" the outputs before running the simulation. Therefore, the choice of input and output constraints becomes important for the convergence of the simulator.

The way that the simulation and the optimization were set up allows the NLP problem to be generated independently of the NLP solver and vice versa. The problem (model) is managed by the process simulator while the solver was managed by Python or Matlab. For the specialization project it was done using only Matlab; for this thesis it was mainly done using a gradient-free solver in NLOpt in Python. In the future, the optimization could be done with a different algorithm or program, just by reading and modifying the input values in the simulation spreadsheets via COM interface.

The active constraint regions were found using the gradient-free solver. It was confirmed that it requires less function evaluations than the gradient-based solver. However, there is still some work to be done regarding the quality of the results, which were not completely consistent when performing the optimizations with a tolerance of 1×10^{-8} compared to the results when performing the optimizations with a tolerance of 1×10^{-6} . The control structure was designed using the results obtained with the lower tolerance.

As the gradient-free solver does not handle output constraints, a penalty function had to be implemented. The magnitude of the penalties was defined according to the magnitude of the constraint variables with respect to the penalty functions. However, there is still an area of opportunity in the systematic definition of the penalties. This problem is not strictly related to the plantwide control procedure, but it would help to ease the use of gradient-free solvers.

Finally, a control structure using self-optimizing variables was designed. As expected, the self-optimizing profit followed closely the optimized profit and the loss was consistently small. This supports the use of self-optimizing control as a very good alternative to continuous optimization.

10.1 Further Work

In order to apply the remaining steps of the plantwide control procedure, some work that can be done from the simulation point of view:

– Generate a dynamic simulation and analyze the use of the process simulator to perform the controllability analysis for the bottom-up design. This should also be done in a consistent and systematic way.

– Make a more detailed analysis of the operating regions (even a finer mesh).

– Develop a more detailed simulation:

  ◦ Compare effect on different types of reactor. The reactor model has been analyzed previously (Løvik, 2001; Kim et al., 2013).

    - Include the power plant in the simulation.
    - Add variable speed curves or IGVs to the compressor.
- Explore the use of "user variables" in `UniSim` to make more flexible the possible inputs for the simulation.

From the coding point of view, some improvements could be:

- Implement a "warm start" strategy for the gradient-free algorithm to reduce the number of function evaluations when the initial values are close to the solution.
- In the loop that is used to generate the operating regions and optimize with different disturbances:
    - If there is an "exception", close and re-open the simulation without saving the last simulation. Otherwise, if the exception was caused by an error in the simulation, the optimization algorithm cannot be re-started.
    - Use the solution of the previous optimization as the initial values for the next optimization, as a "warm start".

These actions would allow not only to generate a finer mesh for the operating regions more efficiently but also to evaluate the effects of inputs and disturbances on measurements also in a more efficient manner.

Bibliography

Aasberg Petersen, K., Nielsen, C. S., Dybkj_r, I., and Perregaard, J. (2011). Large Scale Methanol Production from Natural Gas. Technical report, Haldor Topsoe.

Ahmed, T. H. (1989). *Hydrocarbon phase behavior.* Gulf Pub. Co.

Alstad, V. and Skogestad, S. (2007). Null Space Method for Selecting Optimal Measurement Combinations as Controlled Variables. *Industrial & Engineering Chemistry Research*, 46(3):846–853.

Arouxét, M. B., Echebest, N., and Pilotta, E. A. (2011). Active-set strategy in Powell's method for optimization without derivatives. *Computational & Applied Mathematics*, 30(1):171–196.

Arthur, T. (2010). *Control Structure Design for Methanol Process.* Master thesis, Norwegian University of Science and Technology.

Asadollahi, M., Næ vdal, G., Dadashpour, M., and Kleppe, J. (2014). Production optimization using derivative free methods applied to Brugge field case. *Journal of Petroleum Science and Engineering*, 114:22–37.

Aske, E. M. B. and Skogestad, S. (2009). Dynamic Degrees of Freedom for Tighter Bottleneck Control. In de Brito Alves, R. M., do Nascimento, C. A. O., and Biscaia, E. C., editors, *Computer Aided Chemical Engineering*, volume 27, pages 1275–1280. Elsevier.

Atlas Copco (2011). Driving Centrifugal Compressor Technology.

Birgin, E. G. and Martínez, J. M. (2007). Improving ultimate convergence of an Augmented Lagrangian method. *Optimization Methods and Software*, 23(2):177–195.

Bloch, H. P. (2006). Understanding Centrifugal Process Gas Compressors. In *Compressors and Modern Process Applications.* John Wiley & Sons, Inc.

Brandao de Araujo, A. C. (2007). *Studies on Plantwide Control.* Doctoral thesis for the degree of philosophiae doctor, Norwegian University of Science and Technology.

Braunschweig, B., Joulia, X., Herder, P. M., Stikkelman, R. M., Dijkema, G. P., and Correljé, A. F. (2008). Design of a syngas infrastructure. *Computer Aided Chemical Engineering*, 25:223–228.

Breman, B. B. and Beenackers, A. A. (1996). Thermodynamic Models To Predict Gas Liquid Solubilities in the Methanol Synthesis, the MethanolHigher Alcohol Synthesis, and the Fischer Tropsch Synthesis via Gas Slurry Processes. *Industrial & Engineering Chemistry Research*, 35(10):3763–3775.

Chang, C. J., Chiu, K.-L., and Day, C.-Y. (1998). A new apparatus for the determination of P xy diagrams and Henry's constants in high pressure alcohols with critical carbon dioxide. *The Journal of Supercritical Fluids*, 12(3):223–237.

Chang, C. J., Day, C.-Y., Ko, C.-M., and Chiu, K.-L. (1997). Densities and P-x-y diagrams for carbon dioxide dissolution in methanol, ethanol, and acetone mixtures. *Fluid Phase Equilibria*, 131(1):243–258.

Chang, T. and Rousseau, R. W. (1985). Solubilities of carbon dioxide in methanol and methanol-water at high pressures: experimental data and modeling. *Fluid Phase Equilibria*, 23(2):243–258.

Conn, A. R., Scheinberg, K., and Vicente, L. N. (2009). *Introduction to Derivative-Free Optimization.* Society for Industrial and Applied Mathematics and the Mathematical Programming Society, Philadelphia, USA.

Downs, J. J. and Skogestad, S. (2011). An industrial and academic perspective on plantwide control. *Annual Reviews in Control*, 35(1):99–110.

Fasano, G., Morales, J. L., and Nocedal, J. (2009). On the geometry phase in model-based algorithms for derivative-free optimization. *Optimization Methods and Software*, 24(1):145–154.

Fiedler, E., Grossman, G., Kersebohm, B., Weiss, G., and Witte, C. (2005). Methanol. In *Ullmann's Encyclopedia of Industrial Chemistry.* Wiley-VCH Verlag GmbH & Co.

Graaf, G., Sijtsema, P., Stamhuis, E., and Joosten, G. (1986). Chemical equilibria in methanol synthesis. *Chemical Engineering Science*, 41(11):2883–2890.

Graaf, G., Stamhuis, E., and Beenackers, A. (1988). Kinetics of low-pressure methanol synthesis. *Chemical Engineering Science*, 43(12):3185–3195.

Hamelinck, C. N. and Faaij, A. P. (2002). Future prospects for production of methanol and hydrogen from biomass. *Journal of Power Sources*, 111(1):1–22.

Hitachi Plant Systems (2012). Accomplishment of Centrifugal Compressor with Inlet Guide Vane for Methanol Synthesis Plant.

Jacobsen, M. G. (2011). *Identifying active constrain regions for optimal operation of process plants.* PhD thesis, Norwegian University of Science and Technology.

Johnson, S. G. (2008). The NLopt nonlinear-optimization package.

Joung, S. N., Yoo, C. W., Shin, H. Y., Kim, S. Y., Yoo, K.-P., Lee, C. S., and Huh, W. S. (2001). Measurements and correlation of high-pressure VLE of binary CO2 alcohol systems (methanol, ethanol, 2-methoxyethanol and 2-ethoxyethanol). *Fluid Phase Equilibria*, 185(1):219–230.

Kim, W. S., Yang, D. R., Moon, D. J., and Ahn, B. S. (2013). The process design and simulation for the methanol production on the FPSO (floating production, storage and off-loading) system. *Chemical Engineering Research and Design*.

Klier, K., Chatikavanij, V., Herman, R., and Simmons, G. (1982). Catalytic synthesis of methanol from CO/H2 , The effects of carbon dioxide. *Journal of Catalysis*, 74(2):343–360.

Kralj, A. K. and Glavič, P. (2009). Multi-criteria optimization in a methanol process. *Applied Thermal Engineering*, 29(5):1043–1049.

Lange, J.-P. (2001). Methanol synthesis: a short review of technology improvements. *Catalysis Today*, 64(1):3–8.

Lim, H.-W., Park, M.-J., Kang, S.-H., Chae, H.-J., Bae, J. W., and Jun, K.-W. (2009). Modeling of the Kinetics for Methanol Synthesis using Cu/ZnO/Al 2 O 3 /ZrO 2 Catalyst: Influence of Carbon Dioxide during Hydrogenation. *Industrial & Engineering Chemistry Research*, 48(23):10448–10455.

Løvik, I. (2001). *Modelling, Estimation and Optimization of the Methanol Synthesis with Catalyst Deactivation*. Doctoral thesis for the degree of philosophiae doctor, Norwegian University of Science and Technology.

Lüdtke, K. H. (2004). *Process Centrifugal Compressors: Basics, Function, Operation, Design, Application (Google eBook)*. Springer.

Methanex (2013). Methanex - Methanol Price.

Minasidis, V., Jäschke, J., and Skogestad, S. (2013). Economic plantwide control: Automated controlled variable selection for a reactor-separator-recycle process. In *10th International Symposium on Dynamics and Control of Process Systems (submitted)*, Mumbai, India.

Moulijn, J. A., Makkee, M., and van Diepen, A. E. (2013). *Chemical Process Technology*. Wiley, 2nd editio edition.

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer, 2nd edition edition.

Noureldin, M. M. B., Elbashir, N. O., and El-Halwagi, M. M. (2013). Optimization and Selection of Reforming Approaches for Syngas Generation from Natural/Shale Gas. *Industrial & Engineering Chemistry Research*.

Ohsaki, H., Horiba, J., Hashizume, K., and Masutani, J. (2004). High Efficiency Mitsubishi Centrifugal Compressors and Steam Turbines for Large Methanol and DME Plants. *Mitsubishi Heavy Industries, Ltd.*

Oli Systems (2007). Oli Pro Customization Guide.

Panahi, M. (2011). *Plantwide Control for Economically Optimal Operation of Chemical Plants.* Doctoral thesis for the degree of philosophiae doctor, Norwegian University of Science and Technology.

Pellegrini, L. A., Soave, G., Gamba, S., and Langè, S. (2011). Economic analysis of a combined energy methanol production plant. *Applied Energy*, 88(12):4891–4897.

Powell, M. (2003). On trust region methods for unconstrained minimization without derivatives. *Mathematical Programming*, 97(3):605–623.

Powell, M. (2004). The NEWUOA software for unconstrained optimization without derivatives. Technical report, University of Cambridge, Cambridge, UK.

Powell, M. (2007a). A view of algorithms for optimization without derivatives. Technical report, University of Cambridge. Department of Applied Mathematics and Theoretical Physics., Cambridge, UK.

Powell, M. (2007b). Developments of NEWUOA for minimization without derivatives. Technical report, University of Cambridge, Cambridge, UK.

Powell, M. (2009). The BOBYQA algorithm for bound constrained optimization without derivatives. Technical report, University of Cambridge, Cambridge, UK.

Powell, M. J. D. (2008). Developments of NEWUOA for minimization without derivatives. *IMA Journal of Numerical Analysis*, 28(4):649–664.

Riaz, A., Zahedi, G., and Klemeš, J. J. (2013). A review of cleaner production methods for the manufacture of methanol. *Journal of Cleaner Production*, 57:19–37.

Ríos, L. M. and Sahinidis, N. V. (2013). Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293.

Rostrup-Nielsen, J. and Christiansen, L. J. (2011). *Concepts in Syngas Manufacture.* World Scientific.

Shahrokhi, M. and Baghmisheh, G. (2005). Modeling, simulation and control of a methanol synthesis fixed-bed reactor. *Chemical Engineering Science*, 60(15):4275–4286.

Skogestad, S. (2000). Plantwide control: the search for the self-optimizing control structure. *Journal of Process Control*, 10(5):487–507.

Skogestad, S. (2004). Control structure design for complete chemical plants. *Computers & Chemical Engineering*, 28(1-2):219–234.

Skogestad, S. (2012). Economic Plantwide Control. In Kariwala, Vinay Rangaiah, G. P., editor, *Plantwide Control*, chapter 11, pages 229–251. John Wiley & Sons, Ltd.

Speight, J. G. (2002). *Chemical and Process Design Handbook*. McGraw Hill.

Statistics Norway (2013). Electricity prices, Q2 2013.

Steimel, J., Harrmann, M., Schembecker, G., and Engell, S. (2013). Model-based conceptual design and optimization tool support for the early stage development of chemical processes under uncertainty. *Computers & Chemical Engineering*, 59:63–73.

Unisim (2007). Unisim Operations Guide.

Van-Dal, E. S. o. and Bouallou, C. (2013). Design and simulation of a methanol production plant from CO2 hydrogenation. *Journal of Cleaner Production*, 57:38–45.

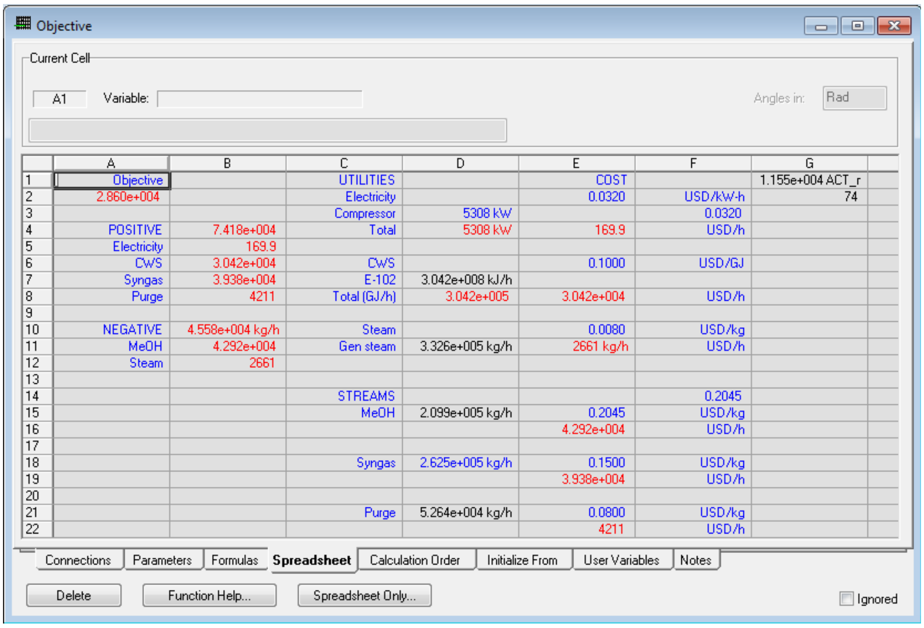
Vanden Bussche, K. and Froment, G. (1996). A Steady-State Kinetic Model for Methanol Synthesis and the Water Gas Shift Reaction on a Commercial Cu/ZnO/Al2O3Catalyst. *Journal of Catalysis*, 161(1):1–10.

Yoon, J. H., Lee, H. S., and Lee, H. (1993). High-pressure vapor-liquid equilibria for carbon dioxide + methanol, carbon dioxide + ethanol, and carbon dioxide + methanol + ethanol. *Journal of Chemical & Engineering Data*, 38(1):53–55.

Zhang, Y., Cruz, J., Zhang, S., Lou, H. H., and Benson, T. J. (2013). Process simulation and optimization of methanol production coupled to tri-reforming process. *International Journal of Hydrogen Energy*, 38(31):13617–13630.

Appendix A
UniSim Setting

The layout of the spreadsheets used to modify values in the simulation is shown in this Appendix. COM works in the Objective, Input, Parameters, Measurements, and Constraints spreadsheets.



Objective

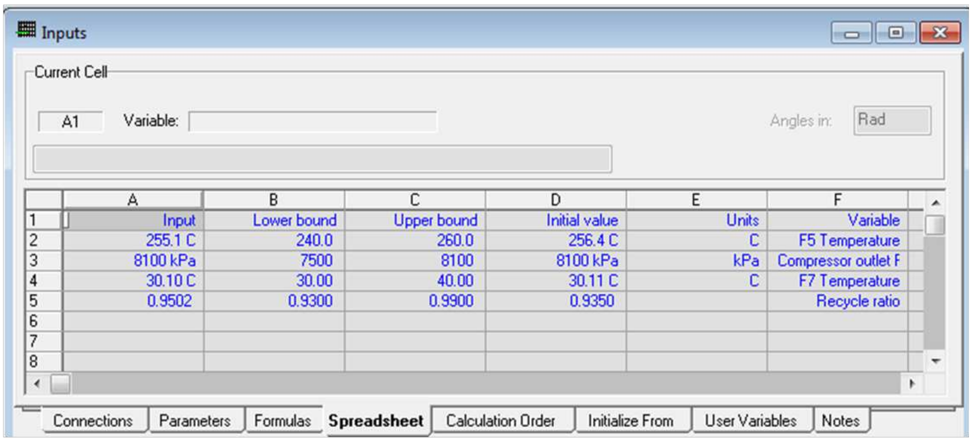
Current Cell

A1 Variable:

Angles in: Rad

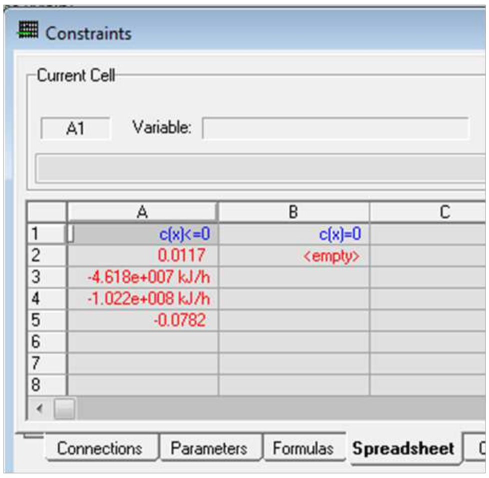
	A	B	C	D	E	F	G
1	Objective		UTILITIES		COST		1.155e+004 ACT_r
2	2.860e+004		Electricity		0.0320	USD/kW-h	74
3			Compressor	5308 kW		0.0320	
4	POSITIVE	7.418e+004	Total	5308 kW	169.9	USD/h	
5	Electricity	169.9					
6	CWS	3.042e+004	CWS		0.1000	USD/GJ	
7	Syngas	3.938e+004	E-102	3.042e+008 kJ/h			
8	Purge	4211	Total (GJ/h)	3.042e+005	3.042e+004	USD/h	
9							
10	NEGATIVE	4.558e+004 kg/h	Steam		0.0080	USD/kg	
11	MeOH	4.292e+004	Gen steam	3.326e+005 kg/h	2661 kg/h	USD/h	
12	Steam	2661					
13							
14			STREAMS			0.2045	
15			MeOH	2.099e+005 kg/h	0.2045	USD/kg	
16					4.292e+004	USD/h	
17							
18			Syngas	2.625e+005 kg/h	0.1500	USD/kg	
19					3.938e+004	USD/h	
20							
21			Purge	5.264e+004 kg/h	0.0800	USD/kg	
22					4211	USD/h	

Figure A.1: Layout of Objective spreadsheet.



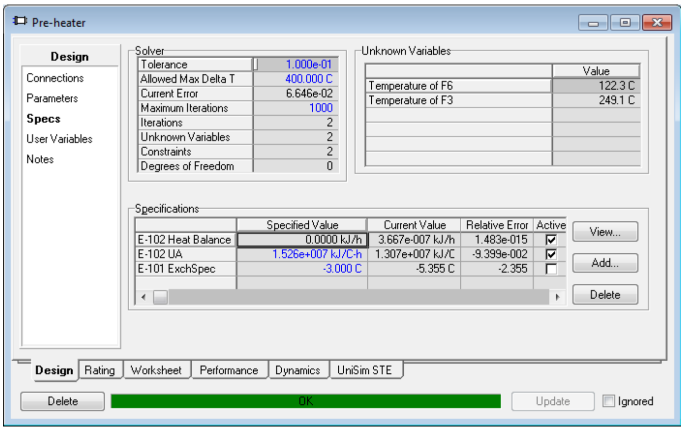
	A	B	C	D	E	F
1	Input	Lower bound	Upper bound	Initial value	Units	Variable
2	255.1 C	240.0	260.0	256.4 C	C	F5 Temperature
3	8100 kPa	7500	8100	8100 kPa	kPa	Compressor outlet F
4	30.10 C	30.00	40.00	30.11 C	C	F7 Temperature
5	0.9502	0.9300	0.9900	0.9350		Recycle ratio
6						
7						
8						

Figure A.2: Layout of Inputs spreadsheet.



	A	B	C
1	c(x)<=0	c(x)=0	
2	0.0117	<empty>	
3	-4.618e+007 kJ/h		
4	-1.022e+008 kJ/h		
5	-0.0782		
6			
7			
8			

Figure A.3: Layout of Constraints spreadsheet.



Solver

Tolerance	1.000e-01
Allowed Max Delta T	400.000 C
Current Error	6.646e-02
Maximum Iterations	1000
Iterations	2
Unknown Variables	2
Constraints	2
Degrees of Freedom	0

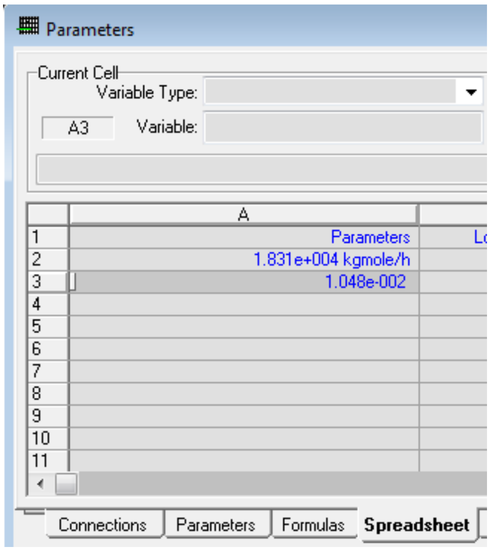
Unknown Variables

	Value
Temperature of F6	122.3 C
Temperature of F3	249.1 C

Specifications

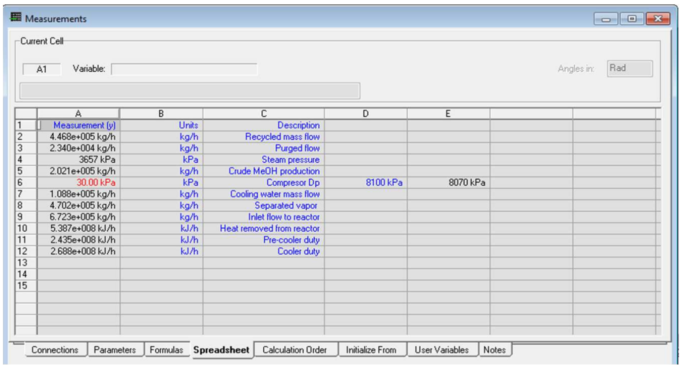
	Specified Value	Current Value	Relative Error	Active
E-102 Heat Balance	0.0000 kJ/h	3.667e-007 kJ/h	1.483e-015	✓
E-102 UA	1.526e+007 kJ/C-h	1.307e+007 kJ/C	-9.399e-002	✓
E-101 ExchSpec	-3.000 C	-5.355 C	-2.355	☐

Figure A.4: Set-up of pre-cooler to adjust UA.



	A
1	Parameters
2	1.831e+004 kgmole/h
3	1.048e-002
4	
5	
6	
7	
8	
9	
10	
11	

Figure A.5: Layout of spreadsheet for setting and reading parameters (disturbances).



	A	B	C	D	E		
1	Measurement (y)	Units	Description				
2	4.468e+005 kg/h	kg/h	Recycled mass flow				
3	2.340e+004 kg/h	kg/h	Purged flow				
4	3657 kPa	kPa	Steam pressure				
5	2.021e+005 kg/h	kg/h	Crude MeOH production				
6	30.00 kPa	kPa	Compresor Dp	8100 kPa	8070 kPa		
7	1.088e+005 kg/h	kg/h	Cooling water mass flow				
8	4.702e+005 kg/h	kg/h	Separated vapor				
9	6.723e+005 kg/h	kg/h	Inlet flow to reactor				
10	5.387e+008 kJ/h	kJ/h	Heat removed from reactor				
11	2.435e+008 kJ/h	kJ/h	Pre-cooler duty				
12	2.688e+008 kJ/h	kJ/h	Cooler duty				
13							
14							
15							

Figure A.6: Layout of spreadsheet for reading measurements.

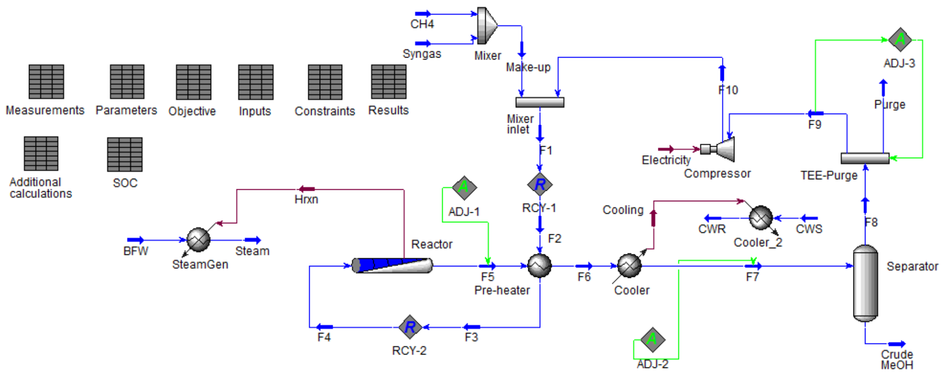
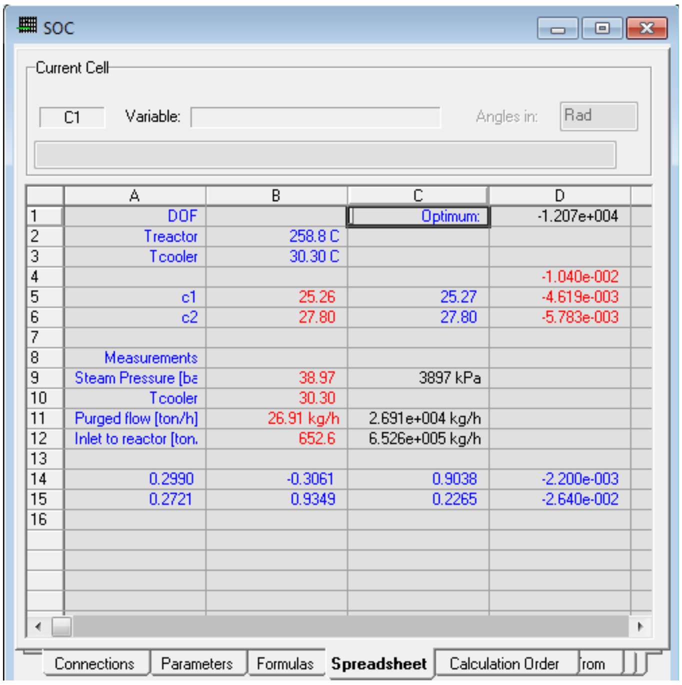


Figure A.7: Layout of simulation for self-optimizing-control. Active constraint is adjusted with "Adjust". The two remaining degrees of freedom are used through two "adjust", to keep c_1 and c_2 constant.



	A	B	C	D
1	DOF		Optimum:	-1.207e+004
2	Treactor	258.8 C		
3	Tcooler	30.30 C		
4				-1.040e-002
5	c1	25.26	25.27	-4.619e-003
6	c2	27.80	27.80	-5.783e-003
7				
8	Measurements			
9	Steam Pressure [ba	38.97	3897 kPa	
10	Tcooler	30.30		
11	Purged flow [ton/h]	26.91 kg/h	2.691e+004 kg/h	
12	Inlet to reactor [ton.	652.6	6.526e+005 kg/h	
13				
14	0.2990	-0.3061	0.9038	-2.200e-003
15	0.2721	0.9349	0.2265	-2.640e-002
16				

Figure A.8: Spreadsheet that was added to calculate and adjust c_1 and c_2.

Appendix B
Code

B.1 Python Code

B.1.1 Optimize Unisim Loop

Using Unisim1 (section B.1.2) interacts with UniSim spreadsheet to disturb the process, evaluate, optimize, and save results.

```python
'''
Created on Apr 23, 2014

@author: vladimim
'''
#Modified by Adriana Reyes to:
#Read and set parameters - disturbances (using Unisim2py)
#Read measurements
#Iterate over parameters
#SOME FUNCTIONS THAT ARE ON THE ORIGINAL CODE BY VLADIMIM
#HAVE BEEN REMOVED BECAUSE THEY WERE NOT USED FOR THE THESIS

import nlopt
import time
import numpy as np
from Unisim1 import Unisim2py
import csv

def objFunc(x, grad):
    global counter
    global beginIter

    Jterms = unisim1.getObjectiveFunctionTerms
    J = lambda x: sum(Jterms(x))
    if grad.size > 0:
        getGradient(J, x, grad)
    counter += 1

    return J(x)


def optimizationBOBYQA():

   # import information from the Unisim
    lb = np.array(unisim1.getLowerBounds(scaled=True))
    print "scaled lb:", lb
    ub = np.array(unisim1.getUpperBounds(scaled=True))
    print "scaled ub:",ub
    x0 = np.array(unisim1.getInitialInputValues(scaled=True))
    print "x0", x0


    # Construct nlopt opt object
    opt = nlopt.opt(nlopt.LN_BOBYQA, unisim1.numberOfInputs)
    # Set the bounds
    opt.set_lower_bounds(lb)
    opt.set_upper_bounds(ub)
    # Specify the objective function
    opt.set_min_objective(objFunc)
    # Maximum number of evaluations
    opt.set_maxeval(2000)
    # Tolerance
    opt.set_xtol_rel(1e-8)

    start = time.time()

    x = opt.optimize(x0)

    end = time.time()

    print "The assignment took", end-start, "seconds."

    print "Algorithm: ", opt.get_algorithm_name()
    print "Optimum at ", x
    print "Output inequality constraints:", unisim1.getInequalityConstraintValues()
    print "Output equality constraints:", unisim1.getEqualityConstraintValues()
    print "Disturbances: ", unisim1.getParametersValues()
    print "Minimum value = ", opt.last_optimum_value()
    print "Result code = ", opt.last_optimize_result()
    if opt.last_optimize_result()>0:
        print "nlopt Successful termination"
    else:
        print "Check: http://ab-initio.mit.edu/wiki/index.php/NLopt_Reference#Return_values"
    print "Function evaluations = ", counter
    print "\n"

    soln.append(opt.last_optimum_value())
    runTime.append(end-start)
    resultCode.append(opt.last_optimize_result())
    evaluations.append(counter)
    inputConstr.append(x)
    outputConstr.append(unisim1.getInequalityConstraintValues())


if __name__ == '__main__':

    unisim1 = Unisim2py()

    global soln
    global runTime
    global resultCode
    global evaluations
    global inputConstr
    global outputConstr

    #global solni
    #global runTimei
    #global resultCodei
    #global evaluationsi
    #global inputConstri
    #global outputConstri

    global writer
    global i
    global j

    #Definition of number of steps
    step=0.01
    #For i (flow)
    resinit=0.983333333
    resfin=1.2
    #for j (composition)
    res1init=0.9
    res1fin=1.0

    #if you want to read the values for iteration from a file
    flow = genfromtxt('Flows.csv', delimiter=',')
    res=np.linspace(resinit,resfin,(resfin-resinit)/step+1)
    res1=np.linspace(res1init,res1fin,(res1fin-res1init)/step+1)

    soln=[]
    runTime=[]
    resultCode=[]
    evaluations=[]
    inputConstr=[]
    outputConstr=[]

    point= 0

    f = open('Results_lowtol_lowconc.csv', 'wb')
    fieldnames = ('Flow_j', 'Comp_j','Flow', 'Comp', 'Soln', 'Time','Code',
        'Evaluations','Input1','Input2','Input3','Input4','Input1sc','
        Input2sc','Input3sc','Input4sc','Output1','Output2','Output3','
        Output4','Meas1','Meas2','Meas3','Meas4','Meas5','Meas6','Meas7','
        Meas8','Meas9','Meas10','Meas11')
    writer = csv.DictWriter(f,fieldnames=fieldnames, restval='missing')

    for i in flow:
        for j in res1:

            counter = 0

            lb = np.array(unisim1.getLowerBounds(scaled=True))
            print lb
            ub = np.array(unisim1.getUpperBounds(scaled=True))
            print ub
            x0 = np.array(unisim1.getInitialInputValues(scaled=True))
            print x0

            #set disturbances
            #dnom=np.array([2.e4, 0.01])
            dnom=np.array([1, 0.01])
            var=np.array([i, j])
            d=var*dnom
            print "d=",d
            unisim1.setParametersValues(d)

            print unisim1.hyFlowsheet.Operations.Item("Objective").Cell("A
                {0}".format(3)).CellVariable.Value
            print unisim1.hyFlowsheet.Operations.Item("Objective").Cell("A
                {0}".format(3)).CellVariable.IsKnown
            print unisim1.hyFlowsheet.Operations.Item("Objective").Cell("A
                {0}".format(3)).CellVariable.IsValid

            unisim1.isObjectiveMultiTerm = False
            unisim1.printNLP()

            try:
                optimizationBOBYQA()

                inputConstr2=np.asarray(inputConstr)
                outputConstr2=np.asarray(outputConstr)

                x=unisim1.getInputsValues(scaled=False)
                print "Inputs", x

                y= unisim1.getMeasurementsValues()

                writer.writerow({ 'Flow_j':i,
                                  'Comp_j':j,
                                  'Flow':d[0],
                                     'Comp':d[1],
                                     'Soln':soln[point],
                                     'Time':runTime[point],
                                     'Code':resultCode[point],
                                     'Evaluations':evaluations[point],
                                     'Input1':x[0],
                                     'Input2':x[1],
                                     'Input3':x[2],
                                     'Input4':x[3],
                                     'Input1sc':inputConstr2.item((point,
                                         0)),
                                     'Input2sc':inputConstr2.item((point,
                                         1)),
                                     'Input3sc':inputConstr2.item((point,
                                         2)),
                                     'Input4sc':inputConstr2.item((point,
                                         3)),
                                     'Output1':outputConstr2.item((point,
                                         0)),
                                     'Output2':outputConstr2.item((point,
                                         1)),
                                     'Output3':outputConstr2.item((point,
                                         2)),
                                     'Output4':outputConstr2.item((point,
                                         3)),
                                     'Meas1':y[0],
                                     'Meas2':y[1],
                                     'Meas3':y[2],
                                     'Meas4':y[3],
                                     'Meas5':y[4],
                                     'Meas6':y[5],
                                     'Meas7':y[6],
                                     'Meas8':y[7],
                                     'Meas9':y[8],
                                     'Meas10':y[9],
                                     'Meas11':y[10],
                                     })


            except nlopt.roundoff_limited as e:
                print e
            point += 1
```

B.1.2 Optimize Unisim

Interacts directly with UniSim spreadsheets.

```python
'''
Created on Apr 7, 2014

@author: vladimim
'''
#Modified by Adriana Reyes to read measurements from spreadsheet
#import comtypes.client as comClient
#import comtypes.gen.UniSimDesign as UD
import sys
import win32com.client as w32c
from win32com.client import gencache
gencache.EnsureModule('{707BF17F-353C-40B2-A5D2-26B20CE7DCFF}', 0, 1, 1)
# import UnisimLib as UD
# import numpy as np
import logging


class Hysys2py:
    def __init__(self, cell):
        self.currentValue = cell

class Unisim2py:
    """A class that load/connects to an open Hysys document through COM and
         attempts to convert it to NLP format
                            min f(x) s.t. lb<=x<=ub h(x)=0 g(x)<=0 """
    def __init__(self):
        try:
            # Start a logger
            logging.basicConfig(level=logging.DEBUG)
            self.logger = logging.getLogger(type(self).__name__)
            self.lastFeasibleObjectiveTerms = []
            self.lastFeasibleInputs = []
            self.isObjectiveMultiTerm = False
            # Attempt to load/connect to Hysys/Unisim
            # gencache.EnsureModule('{707BF17F-353C-40B2-A5D2-26B20CE7DCFF
                }', 0, 1, 1)
            self.hyApp = w32c.Dispatch("UnisimDesign.Application")
            self.logger.debug("...Object initialized "+ repr(self.hyApp))
            self.hyCase = self.hyApp.ActiveDocument
            self.hyCase.Solver.CanSolve = True
            self.logger.debug("...Simulation loaded and running" + repr(
                self.hyCase))
            self.hyFlowsheet = self.hyCase.Flowsheet
            self.logger.debug("...Flowsheet loaded " + repr(self.
                hyFlowsheet))
            self._loadInputsAnsUnits()
            self.logger.debug("...Inputs references and their corresponding measurement units loaded. ")
            self._loadInitialValues()
            self.logger.debug("...References for the initial input values loaded. ")
            self._loadInputsBounds()
            self.logger.debug("...Inputs bounds references loaded. ")
            self._loadInequalityConstraints()
            self.logger.debug("...Inequality constraints references loaded")
            self._loadEqualityConstraints()
            self.logger.debug("...Equality constraints references loaded")
            self._loadObjectiveFunctionTerms()
            self.logger.debug("...Objective function reference loaded")
            self._loadParametersAndUnits()
            self.logger.debug("...Parameter references loaded")
            self._loadMeasurementsAndUnits()
            self.logger.debug("...Measurements references loaded")
            print "MAKE SURE THAT THE FLOWSHEET HAS CONVERGED BEFORE RUNNING THE PROGRAM!"
            print "PRINT THE NLP FORMULATION AND CHECK THAT EVERYTHING IS CORRECT"
        except Exception:
            self.logger.exception("Failed to initialize the Hysys2Py ", exc_info=True)
            sys.exit(1)

    def getLoadedSimulationName(self):
        return self.hyCase.FullName

    def _testFunction(self):
        print repr(self.hyFlowsheet.Operations)
        print repr(self.hyFlowsheet.Operations.Item("Inputs"))
        print self.hyFlowsheet.Operations.Item("Inputs").Cell("A2")
        print self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").AttachmentType
        print self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").Units
        print self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").VariableName
        print self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").VariableType
        print repr(self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").CellVariable)
        print self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").CellVariable
        print self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").CellValue
        print self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").CellText
        print self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").AttachedObjectName
        test1 = self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").CellValue
        print test1
        self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").CellVariable.Value = 258.5
        test1 = 295
        print test1, self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").CellValue

        inputDict = {'input1': self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").CellVariable}

        print inputDict["input1"]
        inputDict["input1"].Value = 260
        print inputDict["input1"]
        self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").CellVariable.Value = 259
        print inputDict["input1"]
        inputDict["input1"].Value = 261
        print inputDict["input1"]
        test2 = self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").CellVariable
        print test2.Value
        test2.Value = 262
        print self.hyFlowsheet.Operations.Item("Inputs").Cell("A2").CellVariable, inputDict["input1"]

    def _loadInputsAnsUnits(self):
        self.inputsList = list()
        self.unitsList = list()
        index = 1
        # Store references to Input cells
        while self.hyFlowsheet.Operations.Item("Inputs").Cell("A{0}".format(index+1)).CellVariable.IsKnown:
            # Create lists
            self.inputsList.append(self.hyFlowsheet.Operations.Item("Inputs").Cell("A{0}".format(index+1)).CellVariable)
            self.unitsList.append(str(self.hyFlowsheet.Operations.Item("Inputs").Cell("E{0}".format(index+1)).CellText))
            index += 1
        # Save the number of inputs
        self.numberOfInputs = index-1
        if self.numberOfInputs<1:
            self.logger.debug("Can't find any inputs !")
            raise

    def _loadParametersAndUnits(self):
        self.parameterList = list()
        self.parameterUnitsList = list()
        index = 1
        # Store references to Input cells
```

```python
        while self.hyFlowsheet.Operations.Item("Parameters").Cell("A{0}".format(index+1)).CellVariable.IsKnown:
            # Create lists
            self.parameterList.append(self.hyFlowsheet.Operations.Item("Parameters").Cell("A{0}".format(index+1)).CellVariable)
            self.parameterUnitsList.append(str(self.hyFlowsheet.Operations.Item("Parameters").Cell("E{0}".format(index+1)).CellText))
            index += 1
        # Save the number of inputs
        self.numberOfParameters = index-1

    def _loadInputsBounds(self):
        self.upperBoundsList = list()
        self.lowerBoundsList = list()
        index = 1
        # Store references to Input cells
        while self.hyFlowsheet.Operations.Item("Inputs").Cell("A{0}".format(index+1)).CellVariable.IsKnown:
            # Create lists
            self.lowerBoundsList.append(self.hyFlowsheet.Operations.Item("Inputs").Cell("B{0}".format(index+1)).CellVariable)
            self.upperBoundsList.append(self.hyFlowsheet.Operations.Item("Inputs").Cell("C{0}".format(index+1)).CellVariable)
            index += 1

    def _loadInitialValues(self):
        self.initialValuesList  = list()
        index = 1
        # Store references to Input cells
        while self.hyFlowsheet.Operations.Item("Inputs").Cell("A{0}".format(index+1)).CellVariable.IsKnown:
            self.initialValuesList.append(self.hyFlowsheet.Operations.Item("Inputs").Cell("D{0}".format(index+1)).CellVariable)
            index += 1

    def _loadObjectiveFunctionTerms(self):
        self.objectiveFunctionTerms = list()
        index = 1
        try:
            # Load Objective
            while self.hyFlowsheet.Operations.Item("Objective").Cell("A{0}".format(index+1)).CellVariable.IsKnown:
                self.objectiveFunctionTerms.append(self.hyFlowsheet.Operations.Item("Objective").Cell("A{0}".format(index+1)).CellVariable)
                index += 1
            self.numberOfObjectiveTerms = index-1
            if self.numberOfObjectiveTerm == 0:
                print "NO KNOWN OBJECTIVE FUNCTION TERMS !"
                raise
        except Exception:
            self.logger.exception("Failed to load objective function terms"
                , exc_info=True)

    def _loadInequalityConstraints(self):
        self.inequalityConstraintsList = list()
        index = 1
        # Inequality Constraints
        try:
            while self.hyFlowsheet.Operations.Item("Constraints").Cell("A
                {0}".format(index+1)).CellVariable.IsKnown:
                self.inequalityConstraintsList.append(self.hyFlowsheet.
                    Operations.Item("Constraints").Cell("A{0}".format(index
                    +1)).CellVariable)
                index += 1
            self.numberOfInequalityConstraints = index - 1
        except Exception:
            self.logger.exception("Failed to load the inequality
                constraints", exc_info=True)


    def _loadEqualityConstraints(self):
        self.equalityConstraintsList = list()
        index = 1
        # Inequality Constraints
        try:
            while self.hyFlowsheet.Operations.Item("Constraints").Cell("B
                {0}".format(index+1)).CellVariable.IsKnown:
                self.equalityConstraintsList.append(self.hyFlowsheet.
                    Operations.Item("Constraints").Cell("B{0}".format(index
                    +1)).CellVariable)
                index += 1
            self.numberOfEqualityConstraints = index - 1
        except Exception:
            self.logger.exception("Failed to load the equality constraints"
                , exc_info=True)


    def _loadMeasurementsAndUnits(self):
        self.measurementsList = list()
        self.measurementUnitsList = list()
        index = 1
        # Store references to Input cells
        while self.hyFlowsheet.Operations.Item("Measurements").Cell("A{0}".
            format(index+1)).CellVariable.IsKnown:
            # Create lists
            self.measurementsList.append(self.hyFlowsheet.Operations.Item("
                Measurements").Cell("A{0}".format(index+1)).CellVariable)
            self.measurementUnitsList.append(str(self.hyFlowsheet.
                Operations.Item("Measurements").Cell("B{0}".format(index+1)
                ).CellText))
            index += 1
        # Save the number of inputs
```

```python
        self.numberOfMeasurements = index-1

    def getInputsValues(self, scaled=True):
        x = list()
        index = 0
        if not scaled:
            for cellVariable in self.inputsList:
                x.append(cellVariable.GetValue(self.unitsList[index]))
                index += 1
        else:
            for cellVariable in self.inputsList:
                x.append(self.scaleInputs(cellVariable.GetValue(self.
                    unitsList[index]),index))
                index += 1
        return x
        for cellVariable in self.inputsList:
            x.append(cellVariable.GetValue(self.unitsList[index]))
            index += 1
        return x

    def setInputListValues(self,x):
        index = 0
        self.hyCase.Solver.CanSolve=False
        for cellVariable in self.inputsList:
            if (x[index]>=0 and x[index]<=1):
                cellVariable.SetValue(self.descaleInputs(x[index], index),
                    self.unitsList[index])
            else:
                print "...Invalid inputs:", x
                sys.exit(1)
            index += 1
        self.hyCase.Solver.CanSolve=True
        print "x = ", x

    def setParametersValues(self,x):
        index = 0
        self.hyCase.Solver.CanSolve=False
        for cellVariable in self.parameterList:
            cellVariable.SetValue(x[index],self.parameterUnitsList[index])
            index += 1
        self.hyCase.Solver.CanSolve=True

    def getMeasurementsValues(self):
        x = list()
        index = 0
        for cellVariable in self.measurementsList:
            x.append(cellVariable.GetValue(self.measurementUnitsList[index
                ]))
            index += 1
        return x

    def getParametersValues(self):
        x = list()
        index = 0
        for cellVariable in self.parameterList:
            x.append(cellVariable.GetValue(self.parameterUnitsList[index]))
            index += 1
        return x

    def getLowerBounds(self, scaled=True):
        x = list()
        index = 0
        if not scaled:
            for cellVariable in self.lowerBoundsList:
                x.append(cellVariable.GetValue())
                index += 1
        else:
            for cellVariable in self.lowerBoundsList:
                x.append(0)
                index += 1
        return x

    def getUpperBounds(self, scaled=True):
        x = list()
        index = 0
        if not scaled:
            for cellVariable in self.upperBoundsList:
                x.append(cellVariable.GetValue())
                index += 1
        else:
            for cellVariable in self.upperBoundsList:
                x.append(1)
                index += 1
        return x

    def getInitialInputValues(self, scaled=True):
        x = list()
        index = 0
        if not scaled:
            for cellVariable in self.initialValuesList:
                x.append(cellVariable.GetValue(self.unitsList[index]))
                index += 1
        else:
            for cellVariable in self.initialValuesList:
                x.append(self.scaleInputs(cellVariable.GetValue(self.
                    unitsList[index]),index))
                index += 1
        return x

    def getInequalityConstraintValues(self):
        x = list()
        index = 0
        for cellVariable in self.inequalityConstraintsList:
            x.append(cellVariable.GetValue())
            index += 1
        return x

    def getEqualityConstraintValues(self):
        x = list()
        index = 0
        for cellVariable in self.equalityConstraintsList:
            x.append(cellVariable.GetValue())
            index += 1
        return x

    def getObjectiveFunctionTerms(self, inputs, isMultiTerm=False):
        self.setInputListValues(inputs)
        Jterms=list()
        if self.isObjectiveMultiTerm:
            feasibleSolution = True
            for index in range(self.numberOfObjectiveTerms):
                if self.objectiveFunctionTerms[index].IsKnown:
                    Jterms.append(self.objectiveFunctionTerms[index].Value)
                    print "Known term", self.objectiveFunctionTerms[index].
                        Value
                else:
                    Jf = self.lastFeasibleObjectiveTerms[index]
                    xf = self.lastFeasibleInputs
                    Jc = Jf**(1+sum(abs(abs(xf)-abs(inputs))))
                    Jterms.append(Jc)
                    print "J{0} term of the objective function is unknown (
                        probably simulation didn't converge)!".format(index
                        )
                    print "Adding a penalty J{0}**(1+sum||xc|-|xf||)"
                    print "last known value of J{0} = {1}, current value of
                         J{0} = {2} )".format(index,Jf,Jc)
                    feasibleSolution = False
            if feasibleSolution:
                self.lastFeasibleObjectiveTerms=Jterms
                self.lastFeasibleInputs = inputs
        else:
            Jterms.append(self.objectiveFunctionTerms[0].Value)
        print "J = sum({0}) = {1}".format(Jterms,sum(Jterms))
        return Jterms

    def scaleInputs(self, inputValue, inputIndex):
        return (inputValue - self.lowerBoundsList[inputIndex].Value)/(self.
            upperBoundsList[inputIndex].Value-self.lowerBoundsList[
            inputIndex].Value)

    def descaleInputsAll(self, inputValues):
        u_descaled = list()
        for inputIndex in xrange(0,len(inputValues)):
            u_descaled.append(self.lowerBoundsList[inputIndex].Value +
                inputValues[inputIndex] * (self.upperBoundsList[inputIndex
                ].Value-self.lowerBoundsList[inputIndex].Value))
        return u_descaled

    def descaleInputs(self, inputValue, inputIndex):
        return self.lowerBoundsList[inputIndex].Value + inputValue * (self.
            upperBoundsList[inputIndex].Value-self.lowerBoundsList[
            inputIndex].Value)

    def printNLP(self):

        x = self.getInputsValues(scaled=False)
        x_scaled = self.getInputsValues()
        x0 = self.getInitialInputValues(scaled=False)
        x0_scaled = self.getInitialInputValues(scaled=True)
        lb = self.getLowerBounds(scaled=False)
        lb_scaled = self.getLowerBounds()
        ub = self.getUpperBounds(scaled=False)
        ub_scaled = self.getUpperBounds()
        param = self.getParametersValues()
        Jterms = self.getObjectiveFunctionTerms(x0_scaled)
        J = sum(Jterms)
        hx = self.getEqualityConstraintValues()
        gx = self.getInequalityConstraintValues()

        print "\npNLP format:\n"
        print "min f(x,p) starting from x0,p"
        print "x belong to R^{0}".format(self.numberOfInputs)
        print "subject to: lb <= x <= ub"
        print "            h(x,p) = 0"
        print "            g(x,p) <= 0\n"
        print "current values:"
        print "f(x)          = sum({0}) = {1}".format(Jterms,J)
        print "x0 (scaled)   = {0}".format(x0_scaled)
        print "x0            = {0}".format(x0)
        print "x (scaled)    = {0}".format(x_scaled)
        print "x             = {0}\n".format(x)
        print "parameter values"
        print "p = {0}\n".format(param)
        print "lb <= x <= ub   (scaled) {0} <= x_scaled <= {1}\n".format(
            lb_scaled[0],ub_scaled[0])

        for index in xrange(0,self.numberOfInputs):
            print "{index}: {lb} <= {x}[{unit}] <= {ub}".format(index=index
                , lb=lb[index], x=x[index], unit=self.unitsList[index], ub=
                ub[index])

        print "h(x) = {0}".format(hx)
        print "g(x) = {0}".format(gx)

def main():
    unisimDesignSimulation = Unisim2py()
    print "Loaded ! " +  unisimDesignSimulation.getLoadedSimulationName()
    unisimDesignSimulation.printNLP()
```

```python

if __name__ == '__main__':
    main()
```

B.1.3 Analyze Results and Plotting

Reads data from results *.cvs file, evaluates number of constraint areas, plots constraint area map, calculates number of function evaluations, creates plots, saves plots.

```
'''
Created on 09/06/2014

@author: Adriana
'''

import matplotlib.pyplot as plt
import numpy as np
import sys
import csv
from numpy import genfromtxt
import os
import math
import scipy as sp
from scipy import interpolate
from scipy.interpolate import griddata
from matplotlib import cm


class getOptimumValues():
    def handy(self,results):
        #This creates a dictionary in which the keys are the flows
        #and the data related to each key is an array
        #that contains all the results (optimum values) obtained with that
            flow
        #with different methane compositions
        d1= np.unique(results[:,2])
        d2= np.unique(results[:,3])
        #print "d1", d1
        num1=np.size(d1)
        num2=np.size(d2)
        print "num1, size d1- flow", num1
        print "num2, size d2- composition", num2
        for i in range (num1):
            if np.isnan(d1[i]):
                np.delete(d1,i)

        d1Optimum = { "%.0f" % d1[i] : self.do_something(i) for i in range
             (0,int(num1)) }
        keys = d1Optimum.keys()
        keys.sort()
       # for x in keys:
       #     print x, '=', d1Optimum[x]

        return d1Optimum

    def do_something(self, i):

        d1= np.unique(results[:,2])
        rowsResults=len(results)
        #print "rowsResults", rowsResults
        optimum=[]
        for j in range (0,rowsResults):
            #if the flow is the flow that we want to get
            if  results[j, 2]== d1[i]:
                optimum= np.hstack((optimum,results[j, 4]))

        return optimum

    def handy_d2(self,results):
        #This creates a dictionary in which the keys are the flows
        #and the data related to each key is an array
        #that contains all the results (optimum values) obtained with that
            flow
        #with different methane compositions
        d1= np.unique(results[:,2])
        d2= np.unique(results[:,3])
        #print "d1", d1
        num1=np.size(d1)
        num2=np.size(d2)
        print "num1, size d1- flow", num1
        print "num2, size d2- composition", num2
        for i in range (num1):
            if np.isnan(d1[i]):
                np.delete(d1,i)

        d2_2plot = { "%.0f" % d1[i] : self.do_something_d2(i) for i in
            range(0,int(num1)) }
        keys2 =  d2_2plot.keys()
        keys2.sort()
        for x in keys2:
            print x, '=', d2_2plot[x]

        return d2_2plot

    def do_something_d2(self, i):

        d1= np.unique(results[:,2])
        rowsResults=len(results)
        #print "rowsResults", rowsResults
        d2plot=[]
        for j in range (0,rowsResults):
            #if the flow is the flow that we want to get
            if  results[j, 2]== d1[i]:
                d2plot= np.hstack((d2plot,results[j, 3]))

        return d2plot
```

```

def unique_rows(data):
    uniq = np.unique(data.view(data.dtype.descr * data.shape[1]))
    return uniq.view(data.dtype).reshape(-1, data.shape[1])

if __name__ == '__main__':

    results = genfromtxt('aResultsforconstraintareas.csv', delimiter=',')
    results2 = genfromtxt('Results463manipaddedmodified.csv', delimiter=','
        )
    results3 = genfromtxt('Results_re_run2nd_part.csv', delimiter=',')

    for i in range (len(results)):
        if np.isnan(results[i,0]):
            np.delete(results,i,0)

    d1init=0.9
    d1fin=1.05
    d2init=0.9
    d2fin=1.1

    d1= np.unique(results[:,2])
    d2= np.unique(results[:,3])
    print "d2.size", np.size(d2)


    sizeResults=np.size(results)
    rowsResults=np.size(results,0)
    columnsResults=np.size(results,1)

    num=np.size(d1)

    d1Optimum=getOptimumValues().handy(results)
    d2PlotOptimum=getOptimumValues().handy_d2(results)

    d1OptimumKeys=sorted(d1Optimum)
    d2PlotOptimumKeys=sorted(d2PlotOptimum)

    plt.figure(1)
    ax = plt.subplot(111)

    for i in xrange(num-1):
        ax.plot(d2PlotOptimum[d2PlotOptimumKeys[i]], -1*d1Optimum[
            d1OptimumKeys[i]], label=d1OptimumKeys[i], marker='o')

    plt.xlabel('$CH_4$ mol composition')
    plt.ylabel('Profit [$/h]')
    plt.axis([np.min(d2)-(d2[1]-d2[0]), np.max(d2)+(d2[1]-d2[0]),-1*(np.max
        (results[:,4])+0.1*(np.max(results[:,4])-np.min(results[:,4]))),
        -1*(np.min(results[:,4])-0.1*(np.max(results[:,4])-np.min(results
        [:,4])))])
    plt.grid(True)

    box = ax.get_position()
    ax.set_position([box.x0, box.y0 + box.height * 0.1,
                 box.width, box.height * 0.9])
    # Put a legend below current axis
    leg=ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.1),
          fancybox=True, shadow=True, ncol=12,title="make-up flow [kmol/h]",
               fontsize=8)

    leg.get_frame().set_alpha(0.5)
    plt.savefig('optimumvsdisturbances.eps', format='eps', dpi=1000,
        transparent=True)


    #Identify active constraint regions
    boundedMax=0.999
    boundedMin=0.001
    boundIneq=-0.001
    activeConstr=np.zeros((int(rowsResults),12))

    for i in range (0,int(rowsResults)):
        if results[i,12]>boundedMax:
            activeConstr[i,0]=1
        elif results[i,12]<boundedMin:
            activeConstr[i,1]=1
        if results[i,13]>boundedMax:
            activeConstr[i,2]=1
        elif results[i,13]<boundedMin:
            activeConstr[i,3]=1
        if results[i,14]>boundedMax:
            activeConstr[i,4]=1
        elif results[i,14]<boundedMin:
            activeConstr[i,5]=1
        if results[i,15]>boundedMax:
            activeConstr[i,6]=1
        elif results[i,15]<boundedMin:
            activeConstr[i,7]=1
        if results[i,16]> boundIneq:
            activeConstr[i,8]=1
        if results[i,17]> boundIneq:
            activeConstr[i,9]=1
        if results[i,18]> boundIneq:
            activeConstr[i,10]=1
        if results[i,19]> boundIneq:
            activeConstr[i,11]=1


    #print"activeConstr", activeConstr
    constrRegions = unique_rows(activeConstr)
    print "constraint regions=", constrRegions

    #number of constraint regions
    numConstrRegions=np.size(constrRegions,0)
    print "number of constraint regions", numConstrRegions

    #Construct an array with three columns: flow, composition, active constraint
    plotConstrRegions=np.zeros((int(rowsResults),3))
    for i in range (0,int(rowsResults)):
        plotConstrRegions[i,0]=results[i,2]
        plotConstrRegions[i,1]=results[i,3]
        for j in range (0,int(numConstrRegions)):
            if np.array_equal(activeConstr[i,:],constrRegions[j,:]):
                plotConstrRegions[i,2]=j+1

    #Associate regions with colors; five constraint regions
    #Color palette taken from http://colorbrewer2.org/#


    rgbColors2=list()
    #This variable should be called hexColors; hexadecimal color notation was used :)
    #Possible improvement for code: create array with color coding;
    #something like colors=['#e41a1c', '#4daf4a',...]
    #for easier manipulation; and to eliminate requirement of re-writing colors.

    elementsRegions=np.zeros(numConstrRegions)

    region1=[]
    region2=[]
    region3=[]
    region4=[]
    region5=[]
    region6=[]
    region7=[]
    region8=[]

    for i in range (0,int(rowsResults)):
        if plotConstrRegions[i,2]==1:
            rgbColors2.append('#e41a1c')
            elementsRegions[0]+=1
            region1= np.hstack((region1,i))
        elif plotConstrRegions[i,2]==2:
            rgbColors2.append('#4daf4a')
            elementsRegions[1]+=1
            region2= np.hstack((region2,i))
        elif plotConstrRegions[i,2]==3:
            rgbColors2.append('#377eb8')
            elementsRegions[2]+=1
            region3= np.hstack((region3,i))
        elif plotConstrRegions[i,2]==4:
            rgbColors2.append('#984ea3')
            elementsRegions[3]+=1
            region4= np.hstack((region4,i))
        elif plotConstrRegions[i,2]==5:
            rgbColors2.append('#ff7f00')
            elementsRegions[4]+=1
            region5= np.hstack((region5,i))
        elif plotConstrRegions[i,2]==6:
            rgbColors2.append('#ffff33')
            elementsRegions[5]+=1
            region6= np.hstack((region6,i))
        elif plotConstrRegions[i,2]==7:
            rgbColors2.append('#a65628')
            elementsRegions[6]+=1
            region7= np.hstack((region7,i))
        elif plotConstrRegions[i,2]==8:
            rgbColors2.append('#f781bf')
            elementsRegions[7]+=1
            region8= np.hstack((region8,i))
        elif plotConstrRegions[i,2]==9:
            rgbColors2.append('#999999')
            elementsRegions[8]+=1

    print "elementsRegions", elementsRegions
    print "region1:", region1
    print "region2:", region2
    print "region3:", region3
    print "region4:", region4
    print "region5:", region5
    print "region6:", region6
    print "region7:", region7
    print "region8:", region8

    regionNames= []
    for i in range(numConstrRegions):
        regionNames=np.hstack((regionNames,str(i+1)))

    p1 = plt.Rectangle((0, 0), 1, 1, fc="#e41a1c")
    p2 = plt.Rectangle((0, 0), 1, 1, fc="#4daf4a")
    p3 = plt.Rectangle((0, 0), 1, 1, fc="#377eb8")
    p4 = plt.Rectangle((0, 0), 1, 1, fc="#984ea3")
    p5 = plt.Rectangle((0, 0), 1, 1, fc="#ff7f00")
    p6 = plt.Rectangle((0, 0), 1, 1, fc="#ffff33")
    p7 = plt.Rectangle((0, 0), 1, 1, fc="#a65628")
    p8 = plt.Rectangle((0, 0), 1, 1, fc="#f781bf")

    p=[p1,p2,p3,p4,p5,p6,p7,p8]

    plt.figure(2)
    ax = plt.subplot(111)
    plt.scatter(plotConstrRegions[:,1], plotConstrRegions[:,0], s=20*np.pi*
        np.ones(rowsResults), color=rgbColors2, alpha=0.8, marker="s")
    plt.xlabel('$CH_4$ mol composition')
    plt.ylabel('Syngas make-up flow [kmol/h]')
    plt.axis([np.min(d2)-(d2[1]-d2[0]), np.max(d2)+(d2[1]-d2[0]),np.min(d1)
        -(d1[1]-d1[0]), np.max(d1)+(d1[1]-d1[0])])

    box = ax.get_position()
    ax.set_position([box.x0, box.y0 + box.height * 0.1,
                 box.width, box.height * 0.9])

    # Put a legend below current axis
    leg=ax.legend(p,regionNames, loc='upper center', bbox_to_anchor=(0.5,
        -0.1),
          fancybox=True, shadow=True, ncol=10,title="Regions", fontsize=8)

    leg.get_frame().set_alpha(0.5)

    plt.savefig('constraintRegions.eps', format='eps', dpi=1000,
        transparent=True)


    plt.figure(3)
    #time for optimization
    timeRun= results[:,5]
    plt.hist(timeRun, bins=20)
    plt.xlabel("Time to solve optimization [s]")
    plt.ylabel("Frequency")
    plt.savefig('runningTime.eps', format='eps', dpi=1000, transparent=True
        )

    plt.figure(4)
    #function evaluations
    fnEvaluations= results[:,7]
    plt.hist(fnEvaluations, bins=15, normed=False)
    plt.xlabel("Number of function evaluations")
    plt.ylabel("Frequency")
    plt.savefig('numEvaluations.eps', format='eps', dpi=1000, transparent=
        True)

    plt.figure(5)
    #needs two results to compare
    fneval1=results2[:,7]
    fneval2=results3[:,7]
    plt.hist([fneval1, fneval2], bins=40, normed=False, label=['$10^{-6}$',
         '$10^{-8}$'])
    plt.axvline(fneval1.mean(), color='c', linestyle='dashed', linewidth=2)
    plt.axvline(fneval2.mean(), color='y', linestyle='dashed', linewidth=2)
    plt.legend(fancybox=True, shadow=True, ncol=10,title="Tolerance")
    plt.xlabel("Number of function evaluations")
    plt.ylabel("Frequency")

    plt.show()
```

B.1.4 Contour Plots

Creates contour plots from results and saves figures.

```
'''
Created on 13/06/2014

@author: Adriana
'''

if __name__ == '__main__':
    from mpl_toolkits.mplot3d import axes3d
    import matplotlib.pyplot as plt
    import numpy as np

    from matplotlib import cm
    from numpy import genfromtxt

    #read file
    results = genfromtxt('Resultsforconstraintareas.csv', delimiter=',')
    #read flows and extract unique values
    d1= np.unique(results[:,2])
    #read concentrations and extract unique values
    d2= np.unique(results[:,3])

    #re arrange data
    lats = results[:,2]
    lons = results[:,3]
    values = -1*results[:,4]
    lat_uniq = list(set(lats.tolist()))
    nlats = len(lat_uniq)
    lon_uniq = list(set(lons.tolist()))
    nlons = len(lon_uniq)
    color_map = cm.spectral
    print lats.shape, nlats, nlons
    yre = lats.reshape(nlats,nlons)
    xre = lons.reshape(nlats,nlons)
    zre = values.reshape(nlats,nlons)

    #Generate 2D contour plot
    plt.figure(1)
    CS = plt.contour(xre, yre, zre, cmap=color_map)
    plt.clabel(CS, inline=1, fontsize=10, fmt='%1.0f')
    plt.xlabel("$CH_4$ mole concentration")
    plt.ylabel("Syngas make-up flow [kmol/h]")
    plt.savefig('contourPlot.eps', format='eps', dpi=1000, transparent=True
        )

    #Generate 3D plot
    fig=plt.figure(2)
    ax = fig.gca(projection='3d')

    ax.plot_surface(xre, yre, zre, rstride=8, cstride=8, alpha=0.3)
    cset = ax.contourf(xre, yre, zre, zdir='z', offset=-100, cmap=plt.cm.
        prism)
    cset = ax.contourf(xre, yre, zre, zdir='x', offset=-40, cmap=plt.cm.
        prism)
    cset = ax.contourf(xre, yre, zre, zdir='y', offset=40, cmap=plt.cm.
        prism)
    ax.set_xlabel('$CH_4$ mole concentration')
    ax.set_ylabel('Syngas make-up flow [kmol/h]')
    ax.set_zlabel('Profit [$/h]')
    plt.savefig('contourPlot3D.eps', format='eps', dpi=1000, transparent=
        True)

    plt.show()
```

B.1.5 Compare Optimization Approaches

Compares number of evaluations and running time of gradient-free and gradient-based optimization codes.

```python
'''
Created on 11/06/2014

@author: Adriana
'''

import matplotlib.pyplot as plt
import numpy as np
import sys
import csv
from numpy import genfromtxt
import os
import math

def extractData(data):

    extracted=[]

    for i in range(1,len(data)):
        # if it is a not even position
        #save it
        #values were saved in even positions
        if i%2!=0:
            extracted= np.hstack((extracted,data[i]))
    return extracted

if __name__ == '__main__':
    pass
    matlabData = genfromtxt('MatlabNominalSteps.csv', delimiter=',')
    dataGoodStart = genfromtxt('CH4nominal_goodStart.csv', delimiter=',')
    dataFarStart = genfromtxt('CH4nominal_farStart.csv', delimiter=',')
    dataFarStartLowTol = genfromtxt('CH4nominal_farStart_lowTol.csv',
        delimiter=',')

    dataGoodStart1=extractData(dataGoodStart)
    dataFarStart1=extractData(dataFarStart)
    dataFarStartLowTol1=extractData(dataFarStartLowTol)

    plt.figure(1)
    ax = plt.subplot(111)
    ax.plot(np.arange(len(dataGoodStart1)), -1*dataGoodStart1, 'r--', np.
        arange(len(dataFarStart1)), -1*dataFarStart1, 'ms', np.arange(len(
        dataFarStartLowTol1)), -1*dataFarStartLowTol1, 'c+', 9*np.arange(
        len(matlabData)), -1*matlabData, 'y*')
    plt.axis([0,350,11500,13500])
    plt.grid(True)

    box = ax.get_position()
    ax.set_position([box.x0, box.y0 + box.height * 0.1,
                     box.width, box.height * 0.9])
    leg=ax.legend(('gradient-free; close', 'gradient-free; far', 'gradient-
        free; far initial, low tolerance', 'gradient-based; far'),loc='
        upper center', bbox_to_anchor=(0.5, -0.1),
          fancybox=True, shadow=True, ncol=4,title="Algorithm type/Initial
              values", fontsize=10)

    leg.get_frame().set_alpha(0.5)

    plt.xlabel('Function evaluations')
    plt.ylabel('Profit [$/h]')
    plt.show()
```

B.1.6 Plot Self Optimizing Control Results

Plots results of self optimizing control and compares them to the optimum when re-optimizing.

```python
'''
Created on 16/06/2014

@author: Adriana
'''
#Plot J vs J* and Loss
import matplotlib.pyplot as plt
import numpy as np
#import sys
#import csv
#from numpy import genfromtxt
#import os
#import math
#import scipy as sp
#from scipy import interpolate
#from scipy.interpolate import griddata
#from matplotlib import cm

if __name__ == '__main__':

    d_opt=[18.859,18.755,18.651,18.548,18.444,18.340,18.237,18.133,18.030]
    d_soc=[18.859,18.755,18.651,18.548,18.444,18.340,18.237,18.133]

    J_soc
        =[12406.8746,12343.06349,12277.85196,12211.31132,12142.23406,12074.26847,12003.753

    J_opt
        =[12540.4105,12483.99881,12428.54605,12373.3966,12317.42806,12262.18033,12206.8774


    L
        =[133.5358971,140.9353166,150.6940922,162.0852808,175.1939981,187.9118596,203.1235


    plt.figure(1)
    ax = plt.subplot(111)
    ax.plot(d_opt, J_opt, 'ro', d_soc, J_soc, 'ms')
    plt.axis([17.8,19,11900,12600])
    plt.grid(True)
    #plt.label=['free;close initial', 'free,far initial', 'free,far initial
        , low tol', 'gradient,far initial']
    box = ax.get_position()
    ax.set_position([box.x0, box.y0 + box.height * 0.1,
                 box.width, box.height * 0.9])
    leg=ax.legend(('J*', 'self-optimizing control'),loc='upper center',
        bbox_to_anchor=(0.5, -0.1),
         fancybox=True, shadow=True, ncol=2, fontsize=12)

    leg.get_frame().set_alpha(0.5)
    #plt.legend()
    plt.xlabel('|d|[flow, concentration]')
    plt.ylabel('Profit [$/h]')
    plt.show()

    plt.figure(1)
    ax = plt.subplot(111)
    ax.plot( d_soc, L, color='cyan',marker='o')
    plt.axis([17.8,19,120,220])
    plt.grid(True)
    #plt.label=['free;close initial', 'free,far initial', 'free,far initial
        , low tol', 'gradient,far initial']
    box = ax.get_position()
    ax.set_position([box.x0, box.y0 + box.height * 0.1,
                 box.width, box.height * 0.9])
    leg=ax.legend(('Loss'),loc='upper center', bbox_to_anchor=(0.5, -0.1),
         fancybox=True, shadow=True, ncol=1, fontsize=12)

    leg.get_frame().set_alpha(0.5)
    #plt.legend()
    plt.xlabel('|d|[flow, concentration]')
    plt.ylabel('Profit [$/h]')
    plt.show()
```

B.2 Matlab Code

B.2.1 Matlab NLP-Generator

```
clc
clear all
close all

%Declare global variables
global h;
global hyCase;
global f;

h = actxserver('UnisimDesign.Application');
hyCase = h.Activedocument;
f = hyCase.Flowsheet;

%%
% Count the inputs
i=0;
while f.Operations.Item('Inputs').Cell(strcat(['A' int2str(i+2)])).
    CellVariable.IsKnown
    i=i+1;
end

numberOfInputs=i;

if numberOfInputs<1
    error('Warning, no inputs');
end

inputIndices= [ 2:1:numberOfInputs+1;
                2:1:numberOfInputs+1];
inputIndices2= [ 1:1:numberOfInputs;
                2:1:numberOfInputs+1];
inputIndices3=[ 2:1:numberOfInputs+1;
                1:1:numberOfInputs;
                2:1:numberOfInputs+1];
inputIndices4= [ 1:1:numberOfInputs;
                2:1:numberOfInputs+1;
                1:1:numberOfInputs];

% Read the upper and lower bounds plus the initial values
inputLB=zeros(numberOfInputs,1);
inputUB=zeros(numberOfInputs,1);
initialUs=zeros(numberOfInputs,1);

parameter.inputUnits=cell(numberOfInputs,1);

for i=1:1:numberOfInputs
```

```
    inputLB(i)=f.Operations.Item('Inputs').Cell(strcat(['B' int2str(i+1)]))
        .CellVariable.Value;
    inputUB(i)=f.Operations.Item('Inputs').Cell(strcat(['C' int2str(i+1)]))
        .CellVariable.Value;
    initialUs(i)=f.Operations.Item('Inputs').Cell(strcat(['D' int2str(i+1)
        ])).CellVariable.Value;
    parameter.inputUnits(i,1)=cellstr(f.Operations.Item('Inputs').Cell(
        strcat(['E' int2str(i+1)])).CellText);
end

lb=strcat('lb = [',num2str(inputLB'),']'';\n');
ub=strcat('ub = [',num2str(inputUB'),']'';\n');
u0=strcat('u0 = [',num2str(initialUs'),']'';\n');
%%
fName = 'NLP4MATLAB.m';          %# A file name
fid = fopen(fName,'w');          %# Open the file

%Create the opt
if fid ~= -1
    fprintf(fid,...
        strcat(...
        'function [u_opt,fval,exitflag] = NLP4MATLAB()\n\n',...
        'clc\n',...
        'clear all\n',...
        'close all\n\n',...
        'global h;\n',...
        'global f;\n',...
        'global hyCase;\n',...
        'h = actxserver(''UnisimDesign.Application'');\n',...
        'hyCase = h.Activedocument;\n',...
        'f = hyCase.Flowsheet;\n\n',...
        'par=[];\n'));
%     fprintf(fid,'lb(%d) = f.Operations.Item(''Inputs'').Cell(''B%d'').
    CellVariable.Value;\n',inputIndices2);
%     fprintf(fid,'ub(%d) = f.Operations.Item(''Inputs'').Cell(''C%d'').
    CellVariable.Value;\n',inputIndices2);
    fprintf(fid,'lb=zeros(1,%d);\nub=ones(1,%d);\n',numberOfInputs,
        numberOfInputs);
    fprintf(fid,'u0(%d) = scaleInputs(f.Operations.Item(''Inputs'').Cell(''
        D%d'').CellVariable.Value,lb,ub,1);\n',inputIndices2);
    fprintf(fid,...
        strcat(...
        'options = optimset(''TolFun'',10e-8,''TolCon'',1e-4,''Display'',''
            iter'',''Algorithm'',''interior-point'',''Diagnostics'',''on'',
             ''FinDiffType'',''central'',''ScaleProblem'',''obj-and-constr'
            ',''FinDiffRelStep'',1e-2);\n',...
        'tic\n',...
        '[u_opt,fval,exitflag]=fmincon(@(u)objFun(u,par),u0,[],[],[],[],lb,
            ub,@(u)nonLinConFun(u,par),options);\n',...
        'toc\n',...
        'end\n\n'...
        ));
```

```
end
%%
if fid ~= -1
    fprintf(fid,...
        strcat(...
        'function y = objFun(u,par)\n',...
        'global f;\n',...
        'global hyCase;\n',...
        'hyCase.Solver.CanSolve=0;\n'...
        ));
    fprintf(fid,'lb(%d) = f.Operations.Item(''Inputs'').Cell(''B%d'').
        CellVariable.Value;\n',inputIndices2);
    fprintf(fid,'ub(%d) = f.Operations.Item(''Inputs'').Cell(''C%d'').
        CellVariable.Value;\n',inputIndices2);
    fprintf(fid,'input%dUnits = f.Operations.Item(''Inputs'').Cell(''E%d'')
        .CellText;\n',inputIndices);
    fprintf(fid,'f.Operations.Item(''Inputs'').Cell(''A%d'').CellVariable.
        SetValue(deScaleInputs(u,lb,ub,%d),input%dUnits);\n',inputIndices3)
        ;

    fprintf(fid,...
        strcat(...
        'hyCase.Solver.CanSolve=1;\n',...
        'y = f.Operations.Item(''Objective'').Cell(''A2'').CellValue;\n'
            ,...
        'end\n\n'...
        ));
end
%%
% Count the constraints
i=0;
while f.Operations.Item('Constraints').Cell(strcat(['A' int2str(i+2)])).
    CellVariable.IsKnown
    i=i+1;
end
numberOfInequalityConstraints=i;

i=0;
while f.Operations.Item('Constraints').Cell(strcat(['B' int2str(i+2)])).
    CellVariable.IsKnown
    i=i+1;
end
numberOfEqualityConstraints=i;

ineqConIndices=[2:1:numberOfInequalityConstraints+1;2:1:
    numberOfInequalityConstraints+1];
eqConIndices=[2:1:numberOfEqualityConstraints+1;2:1:
    numberOfEqualityConstraints+1];

if fid ~= -1
    fprintf(fid,...
        strcat(...
        'function [c,ceq] = nonLinConFun(u,par)\n',...
        'global f;\n',...
        'global hyCase;\n',...
        'hyCase.Solver.CanSolve=0;\n'...
        ));
    fprintf(fid,'lb(%d) = f.Operations.Item(''Inputs'').Cell(''B%d'').
        CellVariable.Value;\n',inputIndices2);
    fprintf(fid,'ub(%d) = f.Operations.Item(''Inputs'').Cell(''C%d'').
        CellVariable.Value;\n',inputIndices2);
    fprintf(fid,'input%dUnits = f.Operations.Item(''Inputs'').Cell(''E%d'')
        .CellText;\n',inputIndices);
    fprintf(fid,'f.Operations.Item(''Inputs'').Cell(''A%d'').CellVariable.
        SetValue(deScaleInputs(u,lb,ub,%d),input%dUnits);\n',inputIndices3)
        ;

    fprintf(fid,'hyCase.Solver.CanSolve=1;\n');

    if numberOfInequalityConstraints>0
        fprintf(fid,'c(%d)=f.Operations.Item(''Constraints'').Cell(''A%d'')
            .CellValue;\n',ineqConIndices);
    else
        fprintf(fid,'c=[];\n');
    end
    if numberOfEqualityConstraints>0
        fprintf(fid,'ceq(%d)=f.Operations.Item(''Constraints'').Cell(''B%d'
            ').CellValue;\n',eqConIndices);
    else
        fprintf(fid,'ceq=[];\n');
    end
    fprintf(fid,'end\n\n');
end
%%
if fid ~= -1
    fprintf(fid,...
        strcat(...
        'function y = scaleInputs(u,lb,ub,index)\n',...
        'y=(u(index)-lb(index))/(ub(index)-lb(index));\n',...
        'end\n\n'...
        ));
end

%%
if fid ~= -1
    fprintf(fid,...
        strcat(...
        'function y = deScaleInputs(u,lb,ub,index)\n',...
        'y=lb(index)+u(index)*(ub(index)-lb(index));\n',...
        'end\n\n'...
        ));
end

fclose(fid);
```

B.2.2 NLP Example code

```
function [u_opt,fval,exitflag] = NLP4MATLAB()

clc
clear all
close all

global h;
global f;
global hyCase;
h = actxserver('UnisimDesign.Application');
hyCase = h.Activedocument;
f = hyCase.Flowsheet;

par=[];
lb=zeros(1,4);
ub=ones(1,4);
u0(1) = scaleInputs(f.Operations.Item('Inputs').Cell('D2').CellVariable.
    Value,lb,ub,1);
u0(2) = scaleInputs(f.Operations.Item('Inputs').Cell('D3').CellVariable.
    Value,lb,ub,1);
u0(3) = scaleInputs(f.Operations.Item('Inputs').Cell('D4').CellVariable.
    Value,lb,ub,1);
u0(4) = scaleInputs(f.Operations.Item('Inputs').Cell('D5').CellVariable.
    Value,lb,ub,1);
options = optimset('TolFun',10e-8,'TolCon',1e-4,'Display','iter','Algorithm
    ','interior-point','Diagnostics','on', 'FinDiffType','central','
    ScaleProblem','obj-and-constr','FinDiffRelStep',1e-2);
tic
[u_opt,fval,exitflag]=fmincon(@(u)objFun(u,par),u0,[],[],[],[],lb,ub,@(u)
    nonLinConFun(u,par),options);
toc
end

function y = objFun(u,par)
global f;
global hyCase;
hyCase.Solver.CanSolve=0;
lb(1) = f.Operations.Item('Inputs').Cell('B2').CellVariable.Value;
lb(2) = f.Operations.Item('Inputs').Cell('B3').CellVariable.Value;
lb(3) = f.Operations.Item('Inputs').Cell('B4').CellVariable.Value;
lb(4) = f.Operations.Item('Inputs').Cell('B5').CellVariable.Value;
ub(1) = f.Operations.Item('Inputs').Cell('C2').CellVariable.Value;
ub(2) = f.Operations.Item('Inputs').Cell('C3').CellVariable.Value;
ub(3) = f.Operations.Item('Inputs').Cell('C4').CellVariable.Value;
ub(4) = f.Operations.Item('Inputs').Cell('C5').CellVariable.Value;
input2Units = f.Operations.Item('Inputs').Cell('E2').CellText;
input3Units = f.Operations.Item('Inputs').Cell('E3').CellText;
input4Units = f.Operations.Item('Inputs').Cell('E4').CellText;
input5Units = f.Operations.Item('Inputs').Cell('E5').CellText;
```

```
f.Operations.Item('Inputs').Cell('A2').CellVariable.SetValue(deScaleInputs(
    u,lb,ub,1),input2Units);
f.Operations.Item('Inputs').Cell('A3').CellVariable.SetValue(deScaleInputs(
    u,lb,ub,2),input3Units);
f.Operations.Item('Inputs').Cell('A4').CellVariable.SetValue(deScaleInputs(
    u,lb,ub,3),input4Units);
f.Operations.Item('Inputs').Cell('A5').CellVariable.SetValue(deScaleInputs(
    u,lb,ub,4),input5Units);
hyCase.Solver.CanSolve=1;
y = f.Operations.Item('Objective').Cell('A2').CellValue;
end

function [c,ceq] = nonLinConFun(u,par)
global f;
global hyCase;
hyCase.Solver.CanSolve=0;
lb(1) = f.Operations.Item('Inputs').Cell('B2').CellVariable.Value;
lb(2) = f.Operations.Item('Inputs').Cell('B3').CellVariable.Value;
lb(3) = f.Operations.Item('Inputs').Cell('B4').CellVariable.Value;
lb(4) = f.Operations.Item('Inputs').Cell('B5').CellVariable.Value;
ub(1) = f.Operations.Item('Inputs').Cell('C2').CellVariable.Value;
ub(2) = f.Operations.Item('Inputs').Cell('C3').CellVariable.Value;
ub(3) = f.Operations.Item('Inputs').Cell('C4').CellVariable.Value;
ub(4) = f.Operations.Item('Inputs').Cell('C5').CellVariable.Value;
input2Units = f.Operations.Item('Inputs').Cell('E2').CellText;
input3Units = f.Operations.Item('Inputs').Cell('E3').CellText;
input4Units = f.Operations.Item('Inputs').Cell('E4').CellText;
input5Units = f.Operations.Item('Inputs').Cell('E5').CellText;
f.Operations.Item('Inputs').Cell('A2').CellVariable.SetValue(deScaleInputs(
    u,lb,ub,1),input2Units);
f.Operations.Item('Inputs').Cell('A3').CellVariable.SetValue(deScaleInputs(
    u,lb,ub,2),input3Units);
f.Operations.Item('Inputs').Cell('A4').CellVariable.SetValue(deScaleInputs(
    u,lb,ub,3),input4Units);
f.Operations.Item('Inputs').Cell('A5').CellVariable.SetValue(deScaleInputs(
    u,lb,ub,4),input5Units);
hyCase.Solver.CanSolve=1;
c(2)=f.Operations.Item('Constraints').Cell('A2').CellValue;
c(3)=f.Operations.Item('Constraints').Cell('A3').CellValue;
c(4)=f.Operations.Item('Constraints').Cell('A4').CellValue;
c(5)=f.Operations.Item('Constraints').Cell('A5').CellValue;
ceq=[];
end

function y = scaleInputs(u,lb,ub,index)
y=(u(index)-lb(index))/(ub(index)-lb(index));
end

function y = deScaleInputs(u,lb,ub,index)
y=lb(index)+u(index)*(ub(index)-lb(index));
end
```