Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

Ann Iren Fossøy

Implementation and Performance Analysis of Interior-Point Methods for Solving Mathematical Programs with Complementarity Constraints

Master's thesis in Chemical Engineering and Biotechnology
Supervisor: Johannes Jäschke
Co-supervisor: Caroline Satye Nakama
June 2023

NTNU
Norwegian University of
Science and Technology

Ann Iren Fossøy

Implementation and Performance Analysis of Interior-Point Methods for Solving Mathematical Programs with Complementarity Constraints

Master's thesis in Chemical Engineering and Biotechnology
Supervisor: Johannes Jäschke
Co-supervisor: Caroline Satye Nakama
June 2023

Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

NTNU
Norwegian University of
Science and Technology

Abstract

The aim of this thesis is to implement and evaluate interior-point methods for mathematical programs with complementarity constraints, MPCC. MPCC has become a powerful framework for modelling systems with complementarity relationships. These relationships are present in many practical problems, such that effective solving strategies are crucial. However, complementarity constraints present specific difficulties due to their inherent nonconvex nature and violation of essential constraint qualifications. In order to tackle these challenges, two potential solution strategies have been proposed in the literature, the Classic and Dynamic approaches. These approaches are based on a penalty reformulation and were, in this thesis, implemented as extensions of existing interior-point-based algorithms for nonlinear optimisation. However, the Dynamic approach differs from the Classic by allowing greater flexibility in adjusting the penalty parameter. In order to evaluate their performance, an in-house developed solver was used as the foundation for their implementations. After conducting a small-scale problem investigation, it was observed that both algorithms performed well overall, but the Dynamic algorithm outperformed the Classic when unbounded penalty problems were confronted. To further investigate the methods, IPOPT was used as the basis as a more advanced solver was required for this purpose. However, it was challenging to incorporate the additional features as this solver prohibits access to specific steps. Despite this, the integration of the Classic implementation was deemed sufficient and applied to a multicomponent flash tank where phase changes were modelled through complementarity constraints. The steady-state simulations were successfully conducted, while the dynamic simulation presented a significant challenge and needed multiple adjustments. Despite this, a successful simulation which agreed with the physical expectations was eventually obtained. Further research should, however, focus on improving the integration with the existing solvers to obtain a better implementation of the algorithms. Additional model modifications may also be necessary, as poorly posed and non-unique model formulations may be the blame for the problems encountered. Despite the difficulties faced, the Classic and Dynamic methods appeared promising, but further investigations are necessary before a definitive conclusion can be drawn.

Samandrag

Denne avhandlinga tar for seg implementering og evaluering av løysningsmetodar for matematiske program med komplementaritetsavgrensingar, også kjent som MPCC. Desse løysningsmetodane baserer seg på ein bestemt optimaliseringsmetode nærmare kjent som indrepunktmetoden. MPCC er eit kraftig rammeverk for å modellere system der komplementaritetsrelasjonar er til stades. Desse relasjonane er funne i kombinasjon med mange reelle problem, slik at effektive løysningsstrategiar er avgjerande. MPCC er svært utfordrande å løyse ettersom dei er ikkje-konvekse optimaliseringsproblem, som bryt med fundamentale konsept innanfor optimalisering. For å handtere desse problema, har det blitt foreslått to ulike løysningsstrategiar i litteraturen, nærmare kjent som den klassiske og dynamiske metoden. Implementeringa av desse metodane har i denne avhandlinga basert seg på eksisterande optimaliseringsløysara som brukar indrepunktmetoden. Den dynamiske framgangsmåten skil seg frå den klassiske ved at den tilet hyppigare justeringar av straffeparameteren. Denne parameteren er i kostnadsfunksjonen og bestemmer vektinga av tilfredsstillinga til komplementaritetsavgrensingane. For å samanlikne metodane blei ein internt utvikla løysar brukt som grunnlag for implementeringa, og metodane blei anvendt på mindre komplekse problem. Resultat frå denne undersøkinga demonstrerte at begge metodane presterte generelt bra, men at den dynamiske metoden overgjekk den klassiske metoden i spesifikke tilfelle. IPOPT blei vidare brukt som grunnlag for implementeringane, ettersom ein velutvikla løyser var nødvendig for å undersøke metodane ytterlegare. Det var i midlertidig utfordrande å inkorporera algoritmane med denne løysaren, ettersom løysaren og grensesnittet forbaud brukaren tilgang til visse trinn i løysaren. Til tross for dette blei implementeringa av den klassiske algoritmen tilstrekkeleg og vidare brukt for å simulera ein multikomponent flashtank med faseendringar. Den stasjonære simuleringa blei vellykka gjennomført, medan den dynamiske simuleringa bydde på fleire utfordringar. Fleire justeringar var nødvendig for å oppnå ei simulering som var i tråd med dei fysiske forventingane. Vidare forsking bør i midlertidig fokusera på å forbetra integrasjonen med dei eksisterande løysarane, slik at ei betre implementering av algoritmane kan bli oppnådd. Det er også forbetringspotensial i flashtankmodellen, da problema som oppstod kan skyldast ei dårleg, ikkje-unik modellformulering. Til tross for problema som oppstod undervegs, viser den klassiske og den dynamiske algoritmen eit lovande potensial. Ytterlegare eksperimentering er uansett nødvendig for å trekke ein endeleg konklusjon om metodane.

Preface

This master's thesis was completed during the spring semester of 2023 as the final part of my five years master's degree program in Chemical Engineering at the Norwegian University of Science and Technology, NTNU. My thesis was written at the Department of Chemical Engineering, in the Process Systems Engineering group.

I would like to thank my supervisor Johannes Jäschke for his excellent guidance, expertise, and support throughout the project. Furthermore, I would like to express my gratitude to Caroline Satye Nakama for introducing me to the fascinating subject of MPCC, her enthusiasm, and our interesting discussions. I would also like to thank Evren Mert Turan for all his excellent coding assistance.

Lastly, I would like to thank all my friends here in Trondheim, as well as my family, for their major support throughout my studies. I could never have completed this study without them!

Declaration of Compliance:

I hereby declare that this is an independent work according to the exam regulations of the Norwegian University of Science and Technology, NTNU.

Trondheim, June 2023
Ann Iren Fossøy

Contents

List of Tables

List of Figures

Nomenclature

Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
IPOPT	Interior Point Optimiser
KKT	Karush-Kuhn-Tucker
LICQ	Linear Independence Constraint Qualifications
MFCQ	Mangasarian-Formovitz Constraint Qualifications
MPCC	Mathematical Programs with Complementarity Constraints
MPEC	Mathematical Programs with Equilibrium Constraints
NLP	Nonlinear Programming

Chapter 1

Introduction

In recent decades, there has been an increased interest in a particular type of constrained optimisation problem, mathematical programs with complementarity constraints, shortened as MPCC. This modelling framework expands the traditional mathematical programming paradigm by including complementarity constraints [1]. A mutually dependent relationship between two variables is established in these constraints, forcing at least one variable to be at its bound [2]. The main reason for this growing interest in MPCCs is their ability to model a wide range of real-world scenarios. This modelling framework can represent certain discrete decisions and non-smooth phenomena by integrating them seamlessly with continuous optimisation variables. As a result, MPCCs apply to a wide range of disciplines, including engineering, economics, and ecology, as well as many others [3]. Specifically, in chemical engineering, MPCCs play a significant role and can be used to capture phenomena such as flow reversal, phase disappearance and safety valve operations. Problems with disjunctions have conventionally been solved with binary variables through mixed integer programming or disjunctive programming techniques. These methods are non-smooth approaches, renowned for being computationally demanding and time-consuming. MPCC, on the other hand, offers a more precise and effective approach which enhances system performance [2].

Unfortunately, solving MPCCs has proven to be challenging due to their highly nonconvex nature, as well as the linear dependency of the constraints. These programs might resemble standard nonlinear optimisation programs, but the presence of the complementarity conditions introduces both theoretical and numerical difficulties [2]. There have been proposed various approaches to address MPCCs in the literature, including penalisation, relaxation, smoothing, and lifting [4]. An attractive way of solving MPCC is to reformulate it to an equivalent nonlinear program, NLP such that state-of-the-art nonlinear optimisation tools for smooth optimisation can be applied [5]. However, this poses certain challenges, as MPCC tends to violate constraint qualifications, which are fundamental concepts crucial for the NLP solvers to work properly [3]. As a result, further adjustments are required to make these approaches adequate.

In the chemical industry, MPCCs can be used in combination with flash tanks where phase changes are present. The appearance and disappearance of equilibrium phases require adeptness in the model equations of the process. The flash tank has in previous master thesis [6] been simulated using a non-smooth approach with mid-functions [7]. However, Biegler posed a continuous method which involved complementarity constraints. In this method, the vapour-liquid equilibrium condition is relaxed by a parameter if the system happens to be in the single-phase regime. To ensure continuity in the model, the mole fractions are extended continuously into these regimes, even though they might not be defined there if the corresponding phase is absent [8]. However, this specific formulation of a flash tank system will not solve adequately without a sufficient method for solving MPCC.

1.1 Scope of Work

In this thesis, the possibility of adjusting NLP solvers for solving MPCC is further investigated. Two interior-point algorithms for solving MPCC have been proposed in the literature, the Dynamic and the Classic approach [9]. These approaches involve reformulating the problem by using a penalty technique, which was according to the specialisation project found to be promising [10]. The main difference between the approaches is how frequently the penalty parameter can be updated. In this thesis, the approaches will be integrated into two already existing interior-point solvers, where the degree of accessibility and complexity vary. The interior-point solvers selected for this study are the in-house developed solver, Int Point Solver [11] and IPOPT [12]. A small-scale study is performed to distinguish between the methods, as well as a larger case study of a flash tank. The main objective of this thesis is thus to look into different implementation aspects, evaluate the two approaches in terms of performance and study their ability to simulate a flash tank where multiple phases are present.

1.2 Outline

In this thesis, relevant background information about optimisation, interior-point methods and MPCC, in general, is provided in Chapter 2. This chapter also presents the Classic and Dynamic algorithms and explains the concept of orthogonal collocation, a numerical technique for solving differential equations. Further, a discussion of important implementation aspects relevant to this thesis can be found in Chapter 3, as well as the results of the small-scale problem investigation. In Chapter 4, important concepts within thermodynamics are explained, such that the flash tank model can be derived. Chapter 5 presents the result of the case study of the flash tank involving both a stationary and a dynamic version. Lastly, Chapter 6 provides a deeper discussion and gives the conclusion reached based on the result, as well as the recommendations for further research.

Chapter 2

Background and Theory

A brief introduction to constraint optimisation is presented in the first part of this chapter. This is essential as the primary focus will be on a particular class within this domain, MPCC. The introduction of complementarity in optimisation raises some challenges. To explain how these challenges are overcome, key concepts within optimisation need to be clarified. Further in the chapter, the penalty approach and the interior-point method are introduced. These techniques are then combined and adapted to handle complementarity constraints, leading to the Classic and Dynamic approaches. Lastly, a short introduction to a numerical method for solving differential equations, orthogonal collocation, is given to cover the dynamic part of the flash tank model.

2.1 Constrained Optimisation

Constrained optimisation involves selecting the values of the variables which minimises the objective function with respect to certain restrictions. This can be mathematically expressed as,

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & c_i(x) = 0, i \in \mathcal{E}, \\ & c_i(x) \geq 0, i \in \mathcal{I}, \end{aligned} \tag{2.1}$$

where $f(x)$ is the objective function and $c_i(x), i \in \mathcal{E}$ and $c_i(x), i \in \mathcal{I}$ denote the equality and the inequality constraints, respectively. All points satisfying these constraints belong to the feasible set, defined as follows,

$$\Omega = \{x \,|c_i(x) = 0,\ i \in \mathcal{E};\ c_i(x) \geq 0,\ i \in \mathcal{I}\}. \tag{2.2}$$

However, the constraints may mutually contradict each other, leading to an empty feasible set. In this case, the problem in question is deemed infeasible and no solution exists [13]

2.1.1 Continuous Optimisation

Optimisation can also be divided into discrete and continuous forms. In discrete optimisation, the variables are drawn from a finite set, such that the variables are of integer or binary type. However, this thesis deals with continuous optimisation. In continuous optimisation, the discrete variables originate from an uncountable infinite set. These optimisation problems tend to be easier to solve due to the smoothness of the functions facilitating the prediction of an effective search direction. If the gradient of the objective function and constraint functions exist, they can provide valuable information about the behaviour of the function in the neighbourhood of a specific point [13].

2.1.2 Local and Global Solutions

In optimisation, a distinction is also made between local and global solutions. A Local solution has to satisfy the following equation,

$$f(x^*) \leq f(x) \text{ for } x \in \mathcal{N} \cap \Omega,$$

where $\mathcal{N} = \|x - x^*\| < \varepsilon$ with $\varepsilon > 0$ denotes the neighbourhood of the solution x^*. In other words, the local solution is the point resulting in the lowest function value within a restricted neighbourhood of feasible points. On the contrary, the global solution is the point giving the lowest possible function value, taking all the feasible points into consideration. Global solutions are generally more computationally demanding to find than local solutions, except for the case of convex optimisation [8].

2.1.3 Convex Optimisation

Convexity is a property found beneficial in combination with continuous optimisation as it reduces the complexity of the problem. A constrained optimisation problem, as given in equation(2.1), is convex only when both the objective function and the associated feasible set are convex. The objective function, $f(x)$ $x \in \mathbb{R}^n$, is convex if and only if the following property,

$$f(\alpha x_1 + (1-\alpha)x_2) \leq \alpha f(x_1) + (1-\alpha) f(x_2), \tag{2.3}$$

holds for all points $x_1, x_2 \in \mathbb{R}^n$ and any arbitrary number of $\alpha \in (0, 1)$. The special case of strict convexity occurs if the inequality is found to be strict. Further, a feasible set, Ω, will be convex if and only if,

$$\alpha x_1 + (1-\alpha)x_2 \in \Omega, \tag{2.4}$$

hold for all points $x_1, x_2 \in \Omega$ and $\alpha \in (0, 1)$. Thus, any straight line connecting two points within the set must also exist entirely within the set. This will be the case only if the equality constraint functions are linear, and the inequality constraint functions are concave [8].

The convexity property is advantageous as it ensures that every minimum is a global minimum. Additionally, in the case of strict convexity, there will be only one minimum, referred to as the unique global solution. The convex property also facilitates the convergence of these solutions and can make even large-scale problems remarkably efficient to solve. However, even though the convexity property is found desirable, many practical problems are found to be non-convex [13]. In the case of non-convexity, global solutions are regarded as challenging to discover within a reasonable amount of time. This is due to the widely varying curvature, several saddle points and the potential of multiple solutions and local optima. In addition, unlike convex problems, the initial guess plays a decisive factor in which local points the algorithm detects [14].

2.2 Constraint Qualifications

Constraint qualifications are additional regularity conditions crucial in the detection of candidates for solutions. An essential step in most optimisation algorithms is determining whether a feasible descent step away from the current point is possible. Gradients of objective functions and constraint functions are usually useful for this purpose. However, in the case of nonlinear programming, these functions need to be linearized by Taylor series expansions. For this information to be sufficient, the geometric feature of the feasible set needs to be constructively captured by the linearized approximation around the point in question. The constraint qualifications will ensure that this similarity is adequate by introducing certain restrictions on the constraint functions. These restrictions eliminate undesired irregularities at the boundary of the feasible set, such that the linearized approximation resembles the constraint set sufficiently. As constraint qualifications play a pivotal role in the characterization of optimum, they are found essential for optimisation algorithms to perform successfully [13].

There are several different constraint qualifications, where the linear independence constraint qualification, LICQ, tends to appear rapidly in the design of optimisation algorithms. This qualification requires that the active constraint gradients are linearly independent of each other. Another, but weaker, constraint qualification, is the Mangasarian-Formovitz constraint qualification, MFCQ. This regularity condition is fulfilled if the gradients of the equality constraint are linearly independent, and if there additionally exists a search direction d such that the following requirements,

$$\nabla c_i(x^*)^T d > 0, \text{ for all } i \in \mathcal{A}(x^*) \cap \mathcal{I},$$

$$\nabla c_i(x^*)^T d = 0, \text{ for all } i \in \mathcal{E},$$

are fulfilled. $\mathcal{A}(x^*)$ is here referred to as the active set consisting of both the equality constraint and the active inequality constraints. According to the requirements, the direction d should point into the interior of a specific region formed by the linearized active inequality constraints. In addition, d has to exist inside the null space of the gradient of the equality constraints for MFCQ to be fully satisfied [2].

2.3 Optimality Conditions

After a feasible step has been applied, the optimisation algorithm has to evaluate whether the algorithm has succeeded in detecting a possible optimum. The optimality conditions play an essential role in this matter, as they are mathematical expressions established to recognise and certify solutions. The optimality conditions can be divided into two groups, necessary and sufficient conditions. The necessary conditions are requirements needed to be fulfilled for each solution and are therefore found advantageous in the detection of solution candidates. In spite of this, the necessary conditions cannot be used to verify whether a candidate is indeed the optimal solution. For this purpose, sufficient conditions are essential, as they can declare if a given solution is in fact the correct solution. However, sufficient conditions are not characterised as necessary, and may therefore fail for certain strict local minimisers [13].

In the case of constrained optimisation problems, the first-order necessary conditions are referred to as the Karush-Kuhn-Tucker conditions, the KKT conditions. However, in order for these conditions to be necessary for an optimum, the LICQ has to be satisfied with any local solutions, and the objective function, as well as the constraint functions, need to be continuously differentiable. The KKT conditions express that there exists a Lagrange multiplier vector, λ^* for any local solution, x^*, such that the following requirements are satisfied,

$$\nabla_x \mathcal{L}(x^*, \lambda^*) = 0, \tag{2.5a}$$
$$c_i(x^*) = 0, \quad \text{for all } i \in \mathcal{E}, \tag{2.5b}$$
$$c_i(x^*) \geq 0, \quad \text{for all } i \in \mathcal{I}, \tag{2.5c}$$
$$\lambda_i^* \geq 0, \quad \text{for all } i \in \mathcal{I}, \tag{2.5d}$$
$$\lambda_i^* c_i(x^*) = 0, \quad \text{for all } i \in \mathcal{I} \cup \mathcal{E}, \tag{2.5e}$$

where $\mathcal{L}(x, \lambda)$ is the Lagrangian function given as follows,

$$\mathcal{L}(x, \lambda) = f(x) + \sum_{i \in \mathcal{E}} \lambda_i c_i(x) + \sum_{i \in \mathcal{I}} \lambda_i c_i(x). \tag{2.6}$$

However, as KKT conditions are unable to confirm the optimality of a point, the curvature of the constraints needs to be examined to make further conclusions. The second-order necessary conditions require positive curvature in the direction of the constraints at the solution, provided that both KKT conditions and LICQ hold. Mathematically speaking, this means that

$$d^T \nabla^2_{xx} \mathcal{L}(x^*, \lambda^*) d \geq 0, \tag{2.7}$$

must be verified for all solutions. In this context, d denotes the constraint direction, which has to be non-zero and fulfil the following requirements,

$$\nabla c_i(x^*)^T d = 0, \text{ for all } i \in \mathcal{E}, \tag{2.8a}$$

$$\nabla c_i(x^*)^T d = 0, \text{ for all } i \in \mathcal{A}(x^*) \cap \mathcal{I} \text{ where } \lambda_i^* > 0, \tag{2.8b}$$

$$\nabla c_i(x^*)^T d \geq 0, \text{ for all } i \in \mathcal{A}(x^*) \cap \mathcal{I} \text{ where } \lambda_i^* = 0. \tag{2.8c}$$

Finally, the second-order sufficient conditions declare that if a feasible point, x^*, satisfies the KKT conditions, i.e equation (2.5), and the second-order necessary conditions as well, i.e. the curvature requirement given in equation (2.7), the x^* is in fact the solution of the constrained optimisation problem [13].

2.4 Penalty Approaches

Penalty approaches are methods of solving constrained optimisation problems by reconstructing them into partly or fully unconstrained problems. The objective function is extended by terms created from the constraints, allowing the constraints to be removed without being violated. There exist several penalty functions. Among those, the exact penalty function is found practical, reformulating a general constrained optimisation problem, as given in equation (2.1), in the following way

$$\min_x \quad f(x) + \pi \sum_{i \in \mathcal{E}} |c_i(x)| + \pi \sum_{i \in \mathcal{I}} [c_i(x)]^-, \tag{2.9}$$

where $[y]^- := \max(y, 0)$, and π is the penalty parameter. The penalty parameter is a positive scalar that specifies the weighting of constraint satisfaction compared to the objective function minimization. The exact penalty functions are desirable as a certain value of the penalty parameter can return the exact solution of the original optimisation problem. However, the exact penalty function has the drawback of being non-differentiable for certain points, due to the presence of the absolute value [13].

2.5 Interior-Point Method

The interior-point methods consist of algorithms utilising a special technique for solving optimisation problems. They are characterised by requiring strict satisfaction of the inequality constraints, such that the boundary of the feasible set is being avoided. Therefore, the algorithm search for solutions in the interior of the feasible region rather than exploring the boundary [15]. Within the interior-point methods, primal-dual methods are regarded as the most successful and practical approaches, proceeding by searching for primal and dual variables which fulfil the KKT conditions. These methods have proven to be successful in combination with both linear and nonlinear problems [13].

2.5.1 The Barrier and Continuation Interpretations

There exist two interpretations of interior-point methods, barriers and continuation, where both are deemed useful in the derivation of the technique. The barrier approach requires constrained optimisation problems, as in equation (2.1), to be rewritten in the following way

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & c_i(x) = 0, \ i \in \mathcal{E}, \\ & c_i(x) - s = 0, \ i \in \mathcal{I}, \\ & s \geq 0, \end{aligned} \tag{2.10}$$

where the inequality constraints are transformed into equalities by the use of slack variables, $s \geq 0$. By keeping this definition in mind, the equivalent barrier problem becomes as follows,

$$\begin{aligned} \min_{x} \quad & f(x) - \mu \sum_{i=1}^{n} \ln(s_i) \\ \text{s.t.} \quad & c_i(x) = 0, \ i \in \mathcal{E}, \\ & c_i(x) - s_i = 0, \ i \in \mathcal{I}, \end{aligned} \tag{2.11}$$

where $\mu > 0$ is the barrier parameter. In this formulation, $s \geq 0$ is removed as a constraint, as the logarithm prohibits s from becoming too close to zero. This is due to $-\ln(s)$ approaching infinity as s approaches zero. The barrier approach involves solving a sequence of barrier problems with decreasing barrier parameters, μ. Eventually, when μ approaches zero, the barrier problem approaches the optimal solution, x^*.

In the other approach, the continuation interpretation, the KKT conditions are applied to problem (2.10), such that the following system is obtained,

$$\nabla f(x) - \nabla c_{\mathcal{E}}(x)\lambda - \nabla c_{\mathcal{I}}(x) z = 0 \tag{2.12a}$$

$$c_{\mathcal{E}}(x) = 0 \tag{2.12b}$$

$$c_{\mathcal{I}}(x) - s = 0 \tag{2.12c}$$

$$Sz - \mu e = 0, \tag{2.12d}$$

where $\mu = 0$ and $s, z \geq 0$. In this set of equations, λ and z are the Lagrangian multipliers of the equality and inequality constraints, respectively. Further, $c_{\mathcal{E}}$ is the vector holding the equality constraints, while $c_{\mathcal{I}}$ contains the inequalities. In the continuous interpretation, the perturbed KKT system, given in equation (2.12), is solved for a sequence of positive μ approaching zero. However, $s, z > 0$ must be satisfied for each solution, in order to be accounted acceptable. Eventually, when μ is adequately close to zero, a point that satisfies the KKT conditions is achieved [13].

2.5.2 The Interior-Point Algorithm

A basic interior point method computes the step direction by applying Newton's method to the nonlinear system in equation (2.12), resulting in the following primal-dual system,

$$\begin{bmatrix} \nabla \mathcal{L}_{xx}^2 & 0 & -\nabla c_{\mathcal{E}}^T(x) & -\nabla c_{\mathcal{I}}^T(x) \\ 0 & Z & 0 & S \\ \nabla c_{\mathcal{E}}(x) & 0 & 0 & 0 \\ \nabla c_{\mathcal{I}}(x) & -I & 0 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_s \\ p_\lambda \\ p_z \end{bmatrix} = - \begin{bmatrix} \nabla f(x) - \nabla c_{\mathcal{E}}^T(x)\lambda - \nabla c_{\mathcal{I}}^T(x) z \\ Sz - \mu e \\ c_{\mathcal{E}}(x) \\ c_{\mathcal{I}}(x) - s \end{bmatrix}. \tag{2.13}$$

Once the step directions, $p = (p_x, p_s, p_y, p_z)$, have been computed, each step needs to be appropriately sized. A line backtracking search strategy can be used to compute this length. In this technique, the step size is initialised to a relatively large value before it is gradually reduced until a sufficient decrease in the objective function is obtained [13]. When the step sizes are computed, the new iterates can be calculated through,

$$x^+ = x + \alpha_s p_x, \quad s^+ = s + \alpha_s p_s, \tag{2.14a}$$

$$\lambda^+ = y + \alpha_z p_\lambda, \quad z^+ = z + \alpha_z p_z, \tag{2.14b}$$

where α_s and α_z are the step sizes. The algorithm makes progress towards a solution by computing steps based on the derived primal-dual system given in equation (2.13). However, the algorithm must receive an indication of when the optimal solution has been reached in order to terminate. From the perturbed KKT-system in equation (2.12), the optimality error of the barrier problem can be defined as,

$$E_\mu(x, s, \lambda, z) := \max \left\{ \frac{||\nabla f(x) + \nabla c_{\mathcal{E}}(x)\lambda - \nabla c_{\mathcal{I}}(x) z||_\infty}{s_d}, ||c(x)||_\infty, \frac{||Sz - \mu e||_\infty}{s_c} \right\}, \tag{2.15}$$

where s_d and s_c are the scaling parameters, and $c(x)$ consist of both $c_{\mathcal{I}}(x) - s$ and $c_{\mathcal{E}}(x)$. The first term in the maximum operator is referred to as dual infeasibility, the second term is primal infeasibility, and the last term is complementarity infeasibility. The overall optimal solution requires that the optimality error (2.15) is significantly small for an adequately small μ. In other words,

$$E_\mu(x^*, s^*, \lambda^*, z^*) \leq \epsilon_{tol}, \tag{2.16}$$

needs to be satisfied, for $\mu \approx 0$ and a specified overall tolerance $\epsilon_{tol} > 0$. However, in order to proceed from one barrier problem to the next barrier problem, only an approximated solution is required. The tolerance for the optimality error is then loosened, such that the following condition needs to be satisfied,

$$E_\mu(x, s, \lambda, z,) \leq \kappa_\epsilon \mu, \tag{2.17}$$

where κ_ϵ is a constant such that $\kappa_\epsilon \mu \geq \epsilon_{tol}$ [12]. The overall algorithm can then be written as in Algorithm 1.

Algorithm 1 Interior-Point algorithm [13]

Initialization: Select x^0 and $s^0 > 0$, λ^0 and $z^0 > 0$. Choose an initial barrier parameter $\mu^0 > 0$, the overall tolerance ϵ_{tol} and the parameter κ_ϵ
Set k $\leftarrow 0$
Repeat until $E_{\mu^k}(x^k, \lambda^k, z^k) \leq \epsilon_{tol}$
    Repeat until $E_{\mu^k}(x^k, s^k, \lambda^k, z^k) \leq \kappa^\epsilon \mu^k$
        Obtain $p = (p_x, p_s, p_\lambda, p_z)$ by solving the system in equation (2.13)
        Compute α^s, α^z with line search strategy.
        Compute $w^{k+1} = (x^{k+1}, s^{k+1}, \lambda^{k+1}, z^{k+1})$ by using equation (2.14)
        Set $\mu^{k+1} \leftarrow \mu^k$ and $k \leftarrow k + 1$
    end
    Decrease μ^k
end

In the presented method, the approximated solution of the previous barrier problem will be used in the initialisation of the current problem. In this algorithm, there is an outer barrier loop and an inner loop, where the barrier parameter is decreased for each outer iteration. There exist several ways of decreasing the barrier parameter. However, to prevent the barrier parameter from decreasing too drastically in comparison to the overall tolerance, the following rule can be used,

$$\mu^{k+1} = \max\left\{\frac{\epsilon_{\text{tol}}}{10}, \min\left\{\kappa_\mu \mu^k, (\mu^k)^{\theta_\mu}\right\}\right\}, \tag{2.18}$$

where $0 < \kappa_\mu < 1$ and $1 < \theta_\mu < 2$ are given constants [12].

2.5.3 Filter

In interior point methods, a filter or a merit function is essential for the algorithm to evaluate whether a step computed from the primal-dual system should be accepted. A merit function bases its recommendation on a combination of both improvements in objective function and constraint violation. Filter methods, on the contrary, accept trial points if they improve either the objective function, $\varphi_{\mu^k}(x)$ or the constraint violation $\theta(x)$. It is, however, the latter quantity that has the highest priority. If the current constraint violation is above a certain minimum, $\theta(x^k) > \theta^{\min}$, a trial step is accepted if either improvement in the objective function or constraint violation is obtained. In the opposite case, when $\theta(x^k) \leq \theta^{\min}$, a greater emphasis is placed on improving the objective function.

More specifically, the filter is a set consisting of combinations of constraint violations and objective function values that result in ineffective trial points. At each new outer iteration k, the filter is initialised to,

$$\mathcal{F}_k := \{(\theta, \varphi_\mu) \in \mathbb{R}^2 : \theta \geq \theta^{\max}\}. \tag{2.19}$$

The filter will during the inner iterations be augmented with newly discovered failed combinations. The purpose of the filter is to prevent the algorithm from returning to the neighbourhood of already rejected points, such that cycling can be avoided. If a trial point gives a combination of the objective function value and constraint violation such that,

$$(\theta(x^k(\alpha)), \varphi(x^k(\alpha)) \in \mathcal{F}_k,$$

the trial point will be unacceptable to the current filter and therefore rejected by the algorithm [12].

2.6 Mathematical Program with Complementarity Constraints

Mathematical programs with complementarity constraints are a special type of optimisation programs characterized by containing complementarity constraints on the form, $0 \leq x_1 \perp x_2 \geq 0$. This complementarity relationship restricts at least one of the two variables to be on their bound. The overall definition of an MPCC then becomes,

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & c_i(x) = 0, \ i \in \mathcal{E}, \\ & c_i(x) \geq 0, \ i \in \mathcal{I}, \\ & 0 \leq x_1 \perp x_2 \geq 0, \end{aligned} \tag{2.20}$$

where x_1 and x_2 are vectors, and $x = (x_1, x_2)$. The complementarity relationship implies the following,

$$\begin{aligned} & x_{1i} = 0 \vee x_{2i} = 0 \ \ \forall i \\ & x_1 \geq 0, \ x_2 \geq 0, \end{aligned} \tag{2.21}$$

where the logical "or" operator is defined as inclusive allowing both variables to be active at their bound [9]. It is desirable to embed this relationship within an NLP formulation such that well-developed tools can be applied to ensure fast convergence. However, MPCCs in their original form, as given in equation (2.20), are singular optimization problems which make direct usage of NLP solvers impossible. Thus, other analytical reformulations of the complementarity relationship have been studied,

$$x_1^T x_2 = 0, \ x_1, x_2 \geq 0, \tag{2.22a}$$

$$x_{1i} x_{2i} = 0 \ \forall i, \ x_{1i}, x_{2i} \geq 0, \tag{2.22b}$$

$$x_{1i} x_{2i} \leq 0 \ \ \forall i, \ x_{1i}, x_{2i} \geq 0. \tag{2.22c}$$

However, even further adjustments of these formulations are necessary as they tend to fail in combination with NLP solvers. The complementarity relationship introduces an inherent nonconvexity,

which makes even local solutions hard to detect. Moreover, MPCCs also violate important constraint qualifications such as LICQ and MPCQ, which are essential for well-posed nonlinear programs [2].

Due to the violation of the constraint qualifications other concepts of stationary need to be taken into consideration in order to classify solutions of MPCC. The NLP methods can give convergence to stationarity points which are not surly optimum. There exist different concepts of stationarity for MPCCs, but Bouligand stationarity, known as B-stationarity, is the type of stationarity which characterises optimality for MPCCs. By linearizing the objective function and the constraint functions, the following system is obtained,

$$\begin{aligned} \min_{d} \quad & \nabla f(x^*)^T d \\ \text{s.t.} \quad & c_i(x^*) + \nabla c_i(x^*)^T d \geq 0, \quad i \in \mathcal{I} \\ & c_i(x^*) + \nabla c_i(x^*)^T d = 0, \quad i \in \mathcal{E} \\ & 0 \leq x_1^* + d_{x_1} \perp x_2^* + d_{x_2} \geq 0, \end{aligned} \tag{2.23}$$

where $x^* = (x_1^*, x_2^*)$. A point will be defined as B-stationary if it is feasible and if $d = 0$ is the solution to the linearized program in equation 2.23. However, B-stationary is found to be impractical and challenging to handle. Strong stationarity is a stronger and more tractable stationarity condition implying B-stationarity if MPEC-LICQ, a special kind of constraint qualification, holds. It has been proven that if a certain point is a solution to the MPCC, and the MPEC-LICQ holds, then the point will be strongly stationary indeed. The assumption of strong stationarity then enables the use of equivalent, well-posted NLP reformulation to find the solutions of MPCCs [8]. However, a detailed description of MPCC stationarity concepts is outside the scope of this thesis. For more information, interested readers are encouraged to look into the monogram [16].

2.7 The Penalty Reformulation of the MPCC

To avoid the difficulties which arise with the standard reformulation of the MPCC, the penalty technique is applied such that the optimisation problem can be expressed as follows,

$$\begin{aligned} \min_{x} \quad & f(x) + \pi x_1^T x_2 \\ \text{s.t.} \quad & c_i(x) = 0, \ i \in \mathcal{E} \\ & c_i(x) \geq 0, \ i \in \mathcal{I} \\ & x_1, x_2 \geq 0, \end{aligned} \tag{2.24}$$

where $\pi > 0$ is the penalty parameter. In this reformulation, the complementarity pairs are removed as constraints and added as a new term in the objective function. The penalty term $x_1^T x_2$ should according to the general penalty approach be written as $[x_1^T x_2]^- = \max(0, x_1^T x_2)$. However, since $x_1, x_2 \geq 0$ the two expressions become identical, making $x_1^T x_2$ sufficient. This particular reformulation is smooth and therefore permits standard nonlinear programming tools to be applied

successfully [13]. However, a challenge associated with penalty technique is determining a suitable penalty parameter value, π. Generally, the magnitude of π is unknown in advance and depends greatly on the problem at hand.

2.8 Interior-Penalty Method for MPCC

By taking the reformulation of the MPCC in equation (2.24) into account, the equivalent barrier problem becomes as follows,

$$\begin{aligned} \min_{x} \quad & f(x) + \pi x_1^T x_2 - \mu \sum_{i \in \mathcal{I}} \log s_i - \mu \sum_{i=1}^{p} \log x_{1i} - \mu \sum_{i=1}^{p} \log x_{2i} \\ \text{s.t.} \quad & c_i(x) = 0, \ i \in \mathcal{E}, \\ & c_i(x) - s_i = 0, \ i \in \mathcal{I}, \end{aligned} \tag{2.25}$$

where p is the number of elements in the vectors x_i, $\mu > 0$ is the barrier parameter and $s_i > 0$, $i \in \mathcal{I}$ are the slack variables transforming the inequality constraints into equality constraints. The associated Lagrangian can be further expressed as,

$$\begin{aligned} \mathcal{L}_{\mu,\pi}(x, s, \lambda, z) = & f(x) + \pi x_1^T x_2 - \mu \sum_{i \in \mathcal{I}} \log s_i - \mu \sum_{i=1}^{p} \log x_{1i} - \mu \sum_{i=1}^{p} \log x_{2i} \\ & - \sum_{i \in \mathcal{E}} \lambda_i c_i(x) - \sum_{i \in \mathcal{I}} z_i (c_i(x) - s_i), \end{aligned} \tag{2.26}$$

such that the Karush-Kuhn-Tucker conditions for the barrier problem, also known as the first-order necessary conditions, will be,

$$\begin{aligned} \nabla f(x) - \nabla c_{\mathcal{E}}(x)^T \lambda - \nabla c_{\mathcal{I}}(x)^T z - \begin{pmatrix} 0 \\ \mu X_1^{-1} e - \pi x_2 \\ \mu X_2^{-1} e - \pi x_1 \end{pmatrix} &= 0, \\ s_i z_i - \mu = 0, \ i \in \mathcal{I}, \\ c_i(x) = 0, \ i \in \mathcal{E}, \\ c_i(x) - s_i = 0, \ i \in \mathcal{I}. \end{aligned} \tag{2.27}$$

In these equations, $\nabla c_{\mathcal{E}}$ and $\nabla c_{\mathcal{I}}$ are vectors containing the gradients of each individual equality and inequality constraint, respectively. Further, X_i are diagonal matrices with x_i on the diagonal. By using the same convention for s, the KKT conditions can be expressed expressed in a more concise manner,

$$\nabla_x \mathcal{L}_{\mu,\pi}(x, s, \lambda, z) = 0, \tag{2.28a}$$

$$Sz - \mu e = 0, \tag{2.28b}$$

$$c(x, s) = 0, \tag{2.28c}$$

where

$$c(x,s) = \begin{pmatrix} c_{\mathcal{E}}(x) \\ c_{\mathcal{I}}(x) - s \end{pmatrix}. \tag{2.29}$$

To successfully solve the MPCC, these optimality conditions, as well as the complementarity relationships, need to be sufficiently satisfied. In order to determine the level of complementarity satisfaction the following mathematical expression is used,

$$||\min\{x_1^k, x_2^k\}||_\infty \tag{2.30}$$

This formulation is found advantageous as it is independent of the number of variables and the scaling of the problem only has a minor effect. By following the interior-point method principles in Chapter 2.5, a general algorithm for solving MPCC is presented in Algorithm 2. In this algorithm the barrier parameter μ^k, as well as the stopping tolerances, ϵ_{pen}^k, ϵ_{comp}^k are decreasing for each iteration k and will eventually converge to 0 as $k \to \infty$. In order to determine if this algorithm has successfully solved the problem a stopping test needs to be conducted. In this thesis, the test involves verifying if the optimality error, given in equation (2.15), is sufficiently low and ensuring that the barrier parameter and corresponding complementarity error are within a certain tolerance.

Algorithm 2 Interior-Penalty Method for MPCCs [9].

Initialization: Choose initial primal and dual variables, $w^0 = (x^0, s^0, \lambda^0, z^0)$. Set k = 1.

Repeat

1. Choose barrier parameter μ^k, and the stopping tolerances and ϵ_{comp}^k, ϵ_{pen}^k.
2. Get π^k. Find an approximate solution w^k to the barrier problem, that satisfies $x_1^k, x_2^k, s^k, z^k > 0$ and,

$$||\nabla_x \mathcal{L}_{\mu,\pi}(x,s,\lambda,y)|| \leq \epsilon_{pen}^k, \tag{2.31a}$$
$$||Sz - \mu e|| \leq \epsilon_{pen}^k, \tag{2.31b}$$
$$||c(x,s)|| \leq \epsilon_{pen}^k, \tag{2.31c}$$

and,

$$||\min\{x_1^k, x_2^k\}||_\infty \leq \epsilon_{comp}^k \tag{2.32}$$

3. Let $k \leftarrow k + 1$

Until stopping test satisfied.

There exist various approaches to adjusting the penalty parameter, π. The parameter can be kept constant or increased dynamically throughout the optimisation of the barrier problem, leading to the Classic and the Dynamic approach, respectively [9].

2.8.1 The Classic Approach

In the Classic practical interior-point method the penalty parameter is kept fixed in the inner iteration loop until the barrier problem is solved within a given accuracy. The Classic approach is fully represented in Algorithm 3. When the conditions presented in equation (2.33) are satisfied, the values of the complementarity relationships are assessed in step 3. If the norm of the complementarity pairs is above the present tolerance, the penalty parameter in the objective function is increased. The algorithm will then return to the inner loop, such that a new primal-dual step is computed based on the KKT system. On the contrary, if the complementarity condition is adequately satisfied the penalty parameter remains unchanged, and the stopping test is conducted. A successful computes test indicates that the solution has been found. However, in the event of a failed test, the barrier parameter will be decreased, and consequently, the tolerances as well, and a new iteration in the barrier loop will be carried out [9]. In Algorithm 3, j denotes the inner iteration, while k is the barrier loop iteration.

Algorithm 3 : The Classic interior-Penalty Method for MPCCs [9].

Initialization: Choose initial primal and dual variables, $w^0 = (x^0, s^0, \lambda^0, z^0)$, an initial penalty parameter, π^0, and lastly the parameter $\gamma \in (0, 1)$.

Set $j = 0,\ k = 1$

Repeat (barrier loop)

1. Find a barrier parameter μ^k
   Compute ϵ^k_{pen}
   Let $\epsilon^k_{comp} = (\mu^k)^\gamma$.
   Let $\pi^k = \pi^{k-1}$.
2. **Repeat** (inner loop)
   i. set $j + 1 \leftarrow j$ and the current point to be w^{j-1}
   ii. Compute a primal-dual step $p^j = (p^j_x, p^j_s, p^j_\lambda, p^j_z)$ with $\mu = \mu^k$, $\pi = \pi^k$ and w^{j-1} based on the KKT conditions given in equation (2.27).
   iii. Let $w^j = w^{j-1} + \alpha^j p^j$

   until the conditions as follows,

$$||\nabla_x \mathcal{L}_{\mu,\pi}(x, s, \lambda)|| \leq \epsilon^k_{pen}, \tag{2.33a}$$

$$||Sz - \mu e|| \leq \epsilon^k_{pen}, \tag{2.33b}$$

$$||c(x)|| \leq \epsilon^k_{pen}, \tag{2.33c}$$

   are satisfied.
3. **If** $||\min\{x^j_1, x^j_2\}||_\infty \leq \epsilon^k_{comp}$
   let $w^k = w^j$, $k \leftarrow k + 1$
   else set $\pi^k \leftarrow 10\pi^k$ and go to step 2.

until a stopping test is satisfied

2.8.2 The Dynamic Approach

There are some disadvantages related to fixing the penalty parameter throughout the optimisation of the barrier problem. The Dynamic interior-penalty method, presented in Algorithm 4, allows the penalty parameter to be adjusted in each inner iteration loop, making it more flexible than the Classic. In the inner loop, the complementarity satisfaction is checked before the barrier problem is successfully solved. Additionally, the largest complementarity value from the m previous iterations is multiplied with a fraction η to be compared with the current complementarity value, see step iv. in Algorithm 4. If the norm of the complementarity pairs is above the present tolerance and equation 2.34 is true as well, the penalty parameter is increased. However, in the event that the opposite is true, the algorithm will proceed to check if the conditions in equation 2.35 are satisfied without increasing the penalty parameter first. The Dynamic and Classic approaches contain several parameters. The values of the parameters are collected from Leyffers paper and are listed in Table 2.1 [9].

Table 2.1: The chosen values for parameters in the Classic and Dynamic algorithms [9].

Parameter	**Description**	**Value**
π^0	The initial penalty value	1
λ	In the complementarity tolerance	0.4
η	The fraction used in equation (2.34)	0.9
m	The integer used in equation (2.34)	3

Algorithm 4 : The dynamic interior-Penalty Method for MPCCs [9].

Initialization: Choose initial primal and dual variables, $w^0 = (x^0, s^0, \lambda^0, z^0)$, an initial penalty parameter, π^0, the parameters $\gamma \in (0,1)$, and $\eta \in (0,1)$, and lastly an integer $m \geq 0$.
Set $j = 0,\ k = 1$
Repeat (barrier loop)

1. Find a barrier parameter μ^k
   Compute ϵ^k_{pen}
   Let $\epsilon^k_{comp} = (\mu^k)^\gamma$.
2. **Repeat** (inner loop)
   i. set $j+1 \leftarrow j$ and the current point to be w^{j-1}. Let $\pi^j = \pi^{j-1}$
   ii. Compute a primal-dual step $p^j = (p^j_x, p^j_s, p^j_\lambda, p^j_z)$ with $\mu = \mu^k$, $\pi = \pi^k$ and w^{j-1} based on the KKT conditions given in equation (2.27).
   iii. Let $w^j = w^{j-1} + \alpha^j p^j$
   iv. If $||\min\{x^j_1, x^j_2\}||_\infty \leq \epsilon^k_{comp}$ and

$$x_1^{jT} x_2^j > \eta \max\{x_1^{jT} x_2^j, .., x_1^{(j-m+1)T} x_2^{(j-m+1)T}\}, \tag{2.34}$$

   set $\pi^j \leftarrow 10\pi^k$ and go to step 2

   until the conditions as follows,

$$||\nabla_x \mathcal{L}_{\mu,\pi}(x,s,\lambda)|| \leq \epsilon^k_{pen}, \tag{2.35a}$$
$$||Sz - \mu e|| \leq \epsilon^k_{pen}, \tag{2.35b}$$
$$||c(x)|| \leq \epsilon^k_{pen}, \tag{2.35c}$$

   are satisfied.
3. **If** $||\min\{x^j_1, x^j_2\}||_\infty \leq \epsilon^k_{comp}$
   let $w^k = w^j$, $k \leftarrow k+1$
   else set $\pi^k \leftarrow 10\pi^k$ and go to step 2.

until a stopping test is satisfied.

2.8.2.1 Remarks on Implementation

According to the description of the Dynamic approach, the penalty parameter can be increased after each inner iteration, see Algorithm 4 step iv.. In spite of this, forcing the solver to complete two iterations before possibly adjusting the penalty parameter was found to yield beneficial results. Additionally, after the penalty parameter is changed, the algorithm should complete two iterations before permitting further changes. These adjustments contributed to preventing the penalty value to be increased unnecessarily. Based on the original implementation, a poor initial guess could encourage the algorithm to increase the penalty value immediately, as

$$x_1^{jT} x_2^j > \eta \max\{x_1^{jT} x_2^j\}$$

always will be true. However, the algorithm might discover that an increase in penalty value is unnecessary if a few iterations are conducted before permitting the adjustment. In this thesis, the proposed suggestions are used in the implementation of the Dynamic approach.

2.9 Orthogonal Collocation

When solving dynamic optimisation problems numerically, it is often convenient to discrete the system of differential equations to obtain an optimisation problem of the form (2.1). In this thesis, this is obtained by using orthogonal collocation. Orthogonal collocation is a numerical technique for solving differential equations on the form,

$$\frac{dz}{dt} = f(z(t), t), \; z(0) = z_0. \tag{2.36}$$

The fundamental idea is that the solution of the differential equation can be represented through polynomials. The partitioned domain consists of finite elements, where each element, $t \in [t_{i-1}, t_i]$, can be represented through different polynomials of order $K+1$,

$$z_i^K(t) = \alpha_0 + \alpha_1^t + ... + \alpha_K t^K. \tag{2.37}$$

A polynomial can be represented in several equivalent ways, including the Lagrange interpolation representation. They are a linear combination of scaled basis polynomials which result in the lowest degree polynomial that passes through a set of data points. The interpolation requires in total $K+1$ interpolation points, τ, in each interval. The Lagrange interpolation polynomial for interval i becomes,

$$z_i^K(t) = \sum_{j=0}^{K} \ell_j(\tau) z_{i,j}, \quad t \in [t_{i-1}, t_i], \tag{2.38}$$

where

$$t = t_{i-1} + h_i \tau, \tag{2.39}$$

and the basis polynomials are,

$$\ell_j(\tau) = \prod_{k=0, k\neq j}^{K} \frac{\tau - \tau_k}{\tau_j - \tau_k}. \tag{2.40}$$

In this representation, h_i denotes the length of element i, and τ are the interpolation points satisfying,

$$\tau \in [0,1],\ \tau_0 = 0,\ \tau_j < \tau_{j+1}, \tag{2.41}$$

where $j = 0, \ldots, K-1$. The basis polynomials are, in fact, delta functions such that,

$$\ell_j(\tau_i) = \delta_{ji} = \begin{cases} 1, \text{if } j = i \\ 0, \text{if } j \neq i. \end{cases} \tag{2.42}$$

This polynomial representation is desirable as it ensures that the interpolation polynomials pass through all the given interpolation points, resulting in $z^K(t_{i,j}) = z_{i,j}$. This is illustrated in Figure 2.1. The collocation points can, in theory, be chosen arbitrarily. However, there are certain choices leading to a better approximation, such as shifted Gauss Legendre roots and the Radau Roots, presented in Table 2.2.

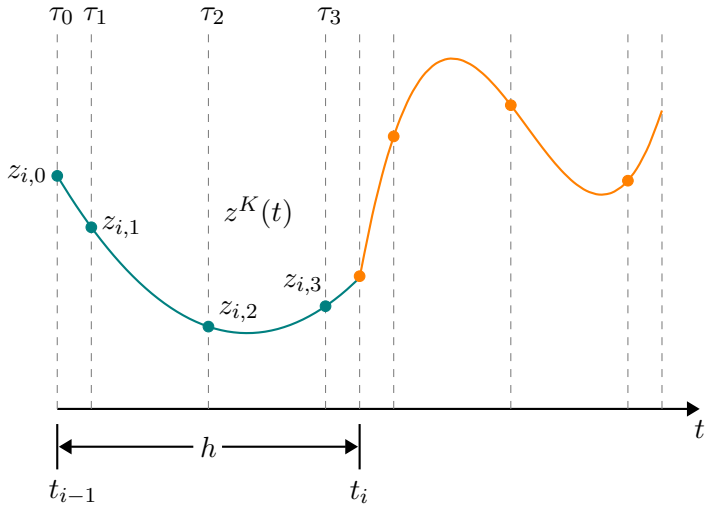


Figure 2.1: Lagrange polynomial representation of solution across a finite element of length h. τ is there the collocation points.

As stated before, the solution can be represented through polynomials, as given in equation (2.38). However, for this representation to be valid the collocation equation needs to be satisfied. The collocation equation is obtained by substituting the Lagrange polynomial representation into equation (2.36),

$$\frac{dz^K}{dt}(t_{i,k}) = f(z^K(t_{i,k}), t_{i,k}), \quad k = 1, \ldots, K. \tag{2.43}$$

Table 2.2: The collocation point as the shifted Gauss-Legendre and Radau roots [8].

Degree K	**Legrendre Roots**	**Radau Roots**
1	0.500000	1.000000
2	0.211325	0.333333
	0.788675	1.000000
3	0.112702	0.155051
	0.500000	0.644949
	0.887298	1.000000
4	0.069432	0.088588
	0.330009	0.409467
	0.669991	0.787659
	0.930568	1.000000
5	0.046910	0.057104
	0.230765	0.276843
	0.500000	0.583590
	0.769235	0.860240
	0.953090	1.000000

This ensures that the time derivatives of the polynomial approximation are identical to the original differential equation at all the collocation points. By taking the polynomial structure into account and additionally substituting t with τ, the collocation equation can be reformulated as,

$$\sum_{j=0}^{K=0} z_{i,j} \frac{d\ell_j(\tau_k)}{d\tau} = h_i f(z_{i,k}, t_{i,k}), \quad k = 1, \ldots, K. \tag{2.44}$$

In this equation, the terms $a_{j,k} = \frac{d\ell_j(\tau_k)}{d\tau}$ are constants and can therefore be pre-computed based on the choice of the fixed collocation points, and represented in form of a squared matrix,

$$a = \begin{bmatrix} \frac{d\ell_0(\tau_0)}{d\tau} & \cdots & \frac{d\ell_0(\tau_k)}{d\tau} \\ \vdots & \ddots & \vdots \\ \frac{d\ell_k(\tau_0)}{d\tau} & \cdots & \frac{d\ell_k(\tau_k)}{d\tau} \end{bmatrix}. \tag{2.45}$$

In the case of three collocation points, the collocation equation becomes,

$$\begin{aligned} &z_{i,0}(-30\tau_k^2 + 36\tau_k - 9) + z_{i,1}(46.7423\tau_k^2 - 51.2592\tau_k + 10.0488) \\ &+ z_{i,2}(-26.7423\tau_k^2 + 20.5925\tau_k - 1.38214) + z_{i,3}(10\tau_k^2 - \frac{16}{3}\tau_k + \frac{1}{3}) \\ &= h_i f(z_{i,k}, t_{i-1} + \tau_k), \; k = 1, .., 3. \end{aligned} \tag{2.46}$$

By inserting the specific type of collocation points, the constant terms in the brackets can be computed and represented using the squared matrix given in equation (2.45). In addition to the collocation constraints obtained in equation (2.44), continuity constraints are crucial in order to ensure continuity between the finite elements as the system has been discretized. The continuity constraints are dependent on the implementation approach, but with Lagrange interpolation profiles, they can be expressed as,

$$z_{i+1,0} = \sum_{j=0}^{K} \ell_j(\tau = 1) z_{i,j}, \quad i = 1, ..., N-1 \tag{2.47a}$$

$$z_f = \sum_{j=0}^{K} \ell_j(\tau = 1) z_{N,j}, \quad z_{1,0} = z_0. \tag{2.47b}$$

However, in some cases, it may be sufficient to enforce the endpoint of one interval to be equal to the first collocation point in the next interval [8],

$$z_{i-1,K} - z_{i,0} = 0. \tag{2.48}$$

Chapter 3

Implementation and Small-Scale Problem Investigation

The following section discusses the implementation of the Classic and Dynamic interior-penalty methods, along with some of the challenges encountered during the implementation. The Classic and the Dynamic algorithm were presented in Section 2.8 in Algorithm 3 and 4, respectively. The algorithms were implemented as extensions to already existing interior point solvers. In this thesis, a solver developed in-house, hereafter referred to as the Int Point Solver, as well as IPOPT, were taken into consideration. Toward the end of this chapter, the result from a small-scale problem investigation is presented to compare the approaches in terms of performance.

The programming language used for the implementations was Julia [17]. The algorithms were here implemented as functions with the optimisation problem and the complementarity pairs passed as arguments. In the initialisation state, the penalty technique was applied, such that the complementarity pairs were included in the objective function and multiplied by the initial penalty value. As mentioned in the previous section, the initial value was set to $\pi^0 = 1$ as this was recommended in literature [9]. The complete implementations of the Classic and Dynamic Algorithms with IPOPT can be found in Appendix F.1 and F.2, respectively.

3.1 Int Point Solver

The Int Point Solver was developed purely in Julia by Dr Nakama [11], following the same principles as the IPOPT implementation [12]. As the solver was developed in-house, the distinct stages within the algorithm were easily accessible. In this manner, the additional features associated with the complementarity constraints in the approaches were completely incorporated into the interior-point solver. These features consist of computing the complementarity values and the complementarity tolerances and, if necessary, adjusting the penalty parameter accordingly. The solver is responsible for reducing the barrier parameter, step computation, and expanding the filter as the optimisation

process proceeds.

Despite meeting the basic requirements of an interior-point solver, the solver is still under development and certain concepts have not yet been implemented, such as the feasibility restoration phase, second-order correction and the ability to handle sparse systems efficiently. The feasibility restoration phase aims to restore feasibility when an infeasible region has been reached, while the second-order correction provides additional information to expedite convergence if a trial point has been rejected. These are concepts which contribute to improving the overall performance [12]. Due to these limitations, the solver is currently most suitable for solving low-complexity problems and may need to be provided with good initial guesses to converge.

3.2 Interior Point OPTimizer

IPOPT, short for Interior POint OPTimizer, is an open-source numerical software package for solving large-scale nonlinear optimisation problems [12]. This solver was implemented originally in Fortran and C. The primary concept was to incorporate the additional features described in the algorithms into a state-of-the-art interior-point-based solver. Essentially, the solver should be responsible for decreasing the barrier parameter, step computation and filter updating, as for the Int Point Solver. However, it was found challenging to adapt IPOPT for these additional features in a satisfactory manner, as the solver prevents the user from accessing and applying changes at certain stages in the optimisation process. Hence, to facilitate the process, the barrier parameter was manually decreased and passed to the solver as an optimiser attribute.

In the implementation, IPOPT handles mainly the inner iterations, whereas the barrier loop is manually implemented due to access restrictions. In the barrier loop, the current threshold for the optimality error and the complementarity tolerance were computed based on the present barrier parameter. The threshold, as well as the barrier parameter, were then passed to IPOPT as optimiser attributes. In the Classic approach, the complementarity values are examined after each barrier problem is successfully solved within the current tolerance. The Dynamic approach, however, also assessed the complementarity satisfaction for each inner iteration. To access information during the optimisation process, a callback function was required. This callback function is invoked by the solver such that custom information can be accessed at a specific point in the optimisation process. However, the callback function could only provide information about the complementarity values and was not capable of performing any modifications to the problem.

There are many advantages of using IPOPT as the basis for the implementations. As a well-developed, effective, and robust solver, IPOPT is capable of handling a wide range of optimisation problems of high complexity. This solver is, on the contrary to the Int Point Solver, provided with support for sparse structures, feasibility restoration and second-order correction. Despite this, a

complex solver which is not primarily coded in Julia has its limitations. As pointed out, the barrier parameter was supposed to be decreased by IPOPT in the implementation. However, the solver and the JuMP interface prohibit modification of the objective function during the optimisation process. Hence, IPOPT had to be terminated for the penalty parameter to be increased. These interruptions seemed to confuse the solver. Even though warm-starts for the primal and dual solutions were used, the solver was not provided with all the information from the previous runs. The filter, for instance, could not be initialised with the filter from the previous barrier problem. These confusions may encourage the algorithm to increase the penalty parameter unnecessarily, leading to convergence failure. In the implementation, the barrier parameter was therefore adjusted manually to avoid unnecessary interruptions. This is, however, not a perfect implementation as IPOPT is presumably not designed to allow manual adjustment of the barrier parameter. In this thesis, although not entirely optimal, the barrier parameter was adjusted manually as the algorithms were still able to solve the problems sufficiently. Despite this adjustment, there were still problems associated with the Dynamic implementation. This approach requires the possibility of modifying the problems during the inner interaction, such that interruptions are unavoidable. Due to the lack of a sufficient implementation, this thesis will not explore the Dynamic approach with IPOPT any further.

3.2.1 The JuMP Interface

As mentioned, JuMP was used as the high-level interface for the modelling [18]. This software package has recently added the ability to set warm-starts for the primal and dual solutions, a feature that improves performance when solving sequences of related problems. This option was found beneficial in the implementation of the Classic and Dynamic algorithms, as the barrier parameter was decreased manually in the algorithm. The primal and dual warm starts permit the present barrier problem to be initiated with the optimal solution of the previous barrier problem. During the specialisation project, the specified feature was unpublished and was described as a drawback of the approach [10]. However, in this thesis, this feature was included and shown to have a positive impact on performance, as the number of iterations generally decreased.

3.3 A small-scale Problem Investigation

The Classic and Dynamic approaches were implemented with both interior-point solvers. To compare the approaches in terms of performance the Int Point Solver was used, as the complementarity features were sufficiently integrated into the solver in this implementation. However, this implementation does not support efficient sparse structure handling, so the comparison is based on rather small-scaled problems.

To perform the comparison, several MPCC problems were retrieved from the MacMPEC collection, a library containing various MPCC test problems provided by Sven Leyffer [19]. The result from the comparison is presented in Table 3.1, where the penalty parameter value, the number of

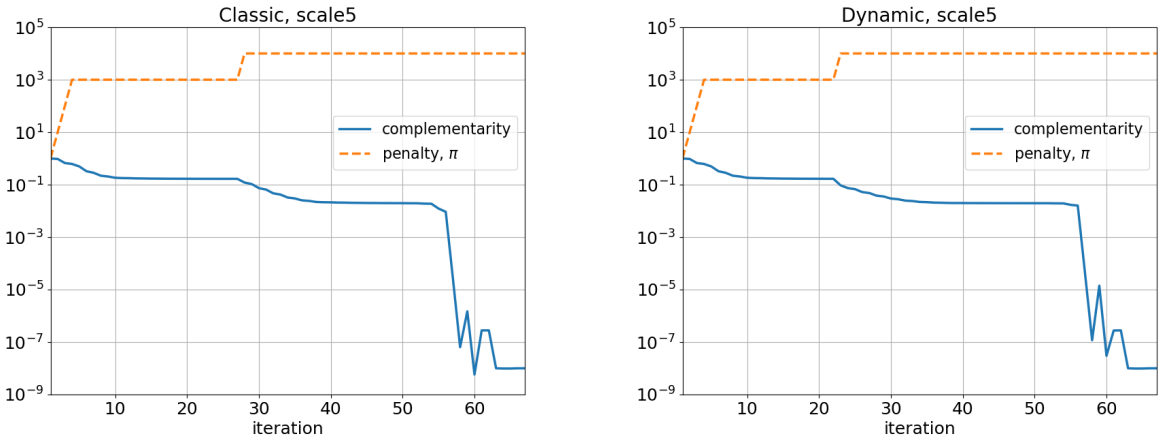
iterations, as well as some additional comments are given. The restoration phase and second-order correction notes indicate that the algorithm failed to converge and additional features, which are yet not implemented, might be needed to restore feasibility. The problems which showed a difference in performance are highlighted in orange. The first noticeable difference is that the Classic implementation used fewer iterations to solve the *bilevel1* problem than the Dynamic. The Dynamic algorithm increased the penalty parameter to $\pi = 100$, while the Classic deemed the initial penalty value of $\pi = 1$ to be sufficient. The unnecessarily high penalty value associated with the Dynamic approach may be responsible for the extra iterations encountered. Despite this, both approaches were able to detect the optimal solution. The table further shows that problem *bilevel2* was only solved with the Classic implementation. However, except for these two problems, the Dynamic approach performed better or equally well as the Classic approach. The table additionally reveals that the majority of the problems were effectively solved by both implementations. This similarity in performance can however be explained by the collection consisting of mostly well-scaled problems of low complexity.

Table 3.1: Results of small-scale problem investigation with π values and the number of iterations for both approaches included. The problems highlighted in orange are the problems where a difference between the approaches was observed. According to the table, both methods performed well in this study. The problems were retrieved from the MacMPEC library [19].

Problem	**Classic implementation**			**Dynamic implementation**		
	π	**it**	**comment**	π	**it**	**comment**
bard1	1	21	-	1	21	-
bard3	1	32	-	1	32	-
bard3m	10	33	-	10	33	-
bilevel1	1	24	-	100	32	-
bilevel2	1	45	-	-	-	restoration phase
bilin	-	-	restoration phase	-	-	restoration phase
design-cent-1	1	-	second order correction	10	26	-
design-cent-2	-	-	diverging	10	32	-
desilva	1	21	-	1	21	-
df1	1	26	-	1	26	-
ex9.1.1	1	22	-	1	22	-
ex9.1.4	-	-	restoration phase	-	-	restoration phase
ex9.1.7	-	-	restoration phase	-	-	restoration phase
ex9.2.4	1	15	-	1	15	-
ex9.2.8	1	16	-	1	16	-
flp2	10	25	-	10	25	-
gauvin	1	19	-	1	19	-
jr2	10	17	-	10	17	-
kth1	1	18	-	1	18	-
kth3	10	20	-	10	20	-
outrata31	-	-	second order correction	-	-	second order correction
outrata32	1	22	-	1	22	-
outrata34	10	29	-	10	29	-
ralph2	-	-	unbounded problem	10	50	-
scale1	1000	16	-	1000	16	-
scale5	10000	67	-	10000	67	-
scholtes1	1	17	-	1	17	-
scholtes4	10000	19	-	10000	19	-

3.3.1 Similarity in Performance

The resemblance in performance can be demonstrated through problem *scale5*, presented in Figure 3.1. In this figure, the orange dashed line gives the penalty parameter value, and the blue line represents the complementarity values at each iteration. As shown in the figure, both approaches were capable of increasing the penalty parameter to $\pi = 1000$ after just a few iterations had been conducted. Despite this, there was still a need for an additional increase in the penalty value for the formulation to be sufficient. The Dynamic algorithm identified this need in fewer iterations than the Classic. Nevertheless, this did not seem to have a remarkable impact on the performance as the approaches converged to the optimal solution in the same amount of iterations. Based on this specific problem, it appeared that both approaches performed equally well despite the small difference.



(a) Classic

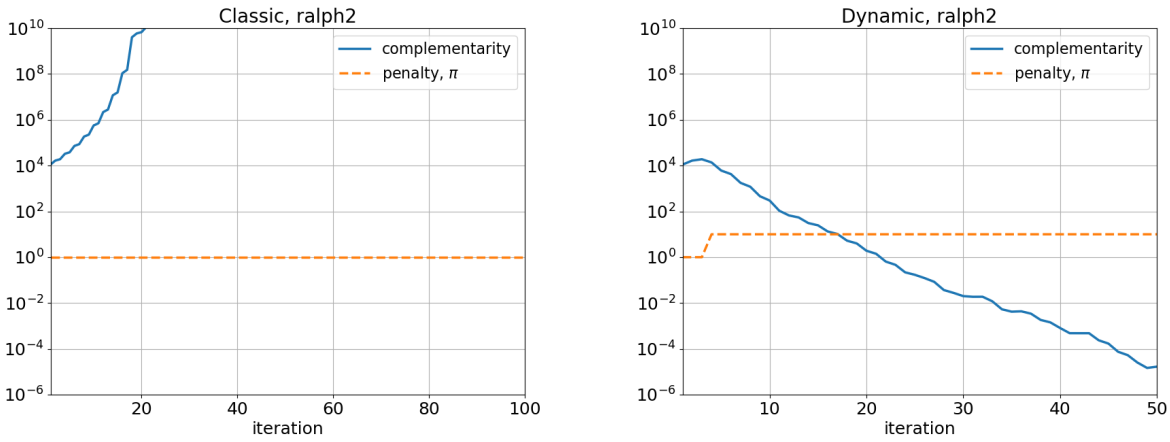
(b) Dynamic

Figure 3.1: The penalty and complementarity value, $||\min\{x_1^k, x_2^k\}||_\infty$, at each inner iteration for the *scale5* problem. The Classic and Dynamic approaches performed essentially the same.

3.3.2 Unbounded Penalty Problem

The *ralph2* problem captured a significant difference between the two approaches. The behaviours of the algorithms for this particular problem are presented in Figure 3.2. The Classic implementation, in Figure3.2a, shows a drastic increase in the complementarity value since the very first iteration. Although this increase continued, the penalty parameter remained unchanged. This is due to the initial value of the penalty parameter resulting in an unbounded barrier problem. The Classic implementation only allows the penalty parameter to be modified once the barrier problem has been solved successfully. The barrier problem, however, is not solvable with the initial penalty value, causing the iterations to diverge. The Classic implementation is, therefore, incapable of solving the problem. It is noteworthy that the complementarity value after twenty iterations is hidden due to the figure's scaling. This scaling is, however, intentional as it makes the figures comparable. Despite this, it is evident that the complementarity values continue increasing toward infinity until the maximum of iterations is reached.

The Dynamic approach, on the other hand, is well suited for such problems, as the complementarity values are evaluated frequently, such that the need for a larger penalty value can be detected early. Figure 3.2b shows that only three iterations were required before the penalty value was increased. After this modification, a remarkable reduction in the complementarity value was obtained, and the optimal solution was eventually detected. According to these results, the Dynamic approach is unequivocally preferred when dealing with unbounded penalty problems. This result is also consistent with the result obtained in the paper which the implementations are based on [5].



(a) Classic **(b)** Dynamic

Figure 3.2: The penalty and complementarity value, $||\min\{x_1^k, x_2^k\}||_\infty$, at each inner iteration for the *Ralph2* problem. The complementarity values diverge with the Classic approach as it is incapable of increasing the penalty problem. The Dynamic approach manages to increase the penalty parameter and solves the problem.

Chapter 4

Flash Tank Model

The purpose of this chapter is to describe the multiphase flash tank model used in this thesis, which is based on the master's thesis of Reed [6] which follows a non-smooth approach [7]. To begin with, an introduction to flash calculations is given, followed by a description of a few basic thermodynamic expressions associated with the flash tank model, including enthalpy calculations and phase equilibrium. Additionally, a proposal of how phase changes can be accounted for in the model in terms of complementarity constraints is provided. This section is then followed by an introduction of model assumptions and parameters, and the derivation of the differential and the algebraic equations for the flash tank model. The model equations in steady state form, as well as in dynamic form, are also listed in Appendix C. The code is provided in Appendix G.

4.1 Flash Calculations

A flash is a single-equilibrium-stage distillation, with the purpose of separating a mixture, such that the more volatile components will appear in the vapour phase, while the less volatile components will emerge as a liquid. In the case of a liquid feed, heat will be added such that the most volatile components will be vaporised. Vapour feed, on the other hand, should be partially condensed by removing heat. In flash calculations, the vapour-liquid-equilibrium equations are combined with the component's material and energy balances. There exist several types of flash calculations depending upon which variables are known in advance [20]. The approach used in this thesis will be further elaborated on.

4.2 Enthalpy Calculations

Enthalpy is a thermodynamic property, defined as

$$H = U + P \cdot V, \tag{4.1}$$

where P is the pressure of the system, V is the volume and U is the internal energy. Enthalpy is a state function such that the value is only dependent on the initial and final state of the system, regardless of the path taken. There exist different ways of calculating the enthalpy change in the system depending on the available data. The enthalpy value at specified states can, however, only be determined relative to a reference state. By assuming an ideal mixture, the enthalpy, of both liquids and gases, can be calculated by taking the sum of each component's contribution,

$$h = \sum_{i}^{nc} z_i \cdot h_i, \tag{4.2}$$

where h_i is the ideal molar enthalpy and z_i is the molar fraction of component i [21]. The enthalpy change, dh, of a system can be calculated through intensive properties. By expressing the change as a function of temperature, T and pressure, P, the total differential of the enthalpy becomes,

$$dh = \left(\frac{\partial h}{\partial T}\right)_p dT + \left(\frac{\partial h}{\partial P}\right)_T dP. \tag{4.3}$$

Specific heat capacity,

$$c_p = \left(\frac{\partial h}{\partial T}\right)_p, \tag{4.4}$$

is a thermodynamic property which describes the amount of energy needed to increase the temperature of a substance by one unit if the pressure remains constant. The specific heat capacity is specific for each substance and independent of the type of process. By inserting this definition into equation (4.3), the enthalpy change becomes,

$$dh = c_p\, dT + \left(\frac{\partial h}{\partial P}\right)_T dP. \tag{4.5}$$

In order to further transform the equation, another fundamental representation of enthalpy is required,

$$dh = Tds + VdP, \tag{4.6}$$

where s is the entropy of the system. The change of the entropy can, as enthalpy, be expressed in terms of temperature and pressure, such that the following expression is obtained,

$$ds = \left(\frac{\partial s}{\partial T}\right)_p dT + \left(\frac{\partial s}{\partial P}\right)_T dP. \tag{4.7}$$

Equation (4.7) can then be substituted into equation (4.6), which result in,

$$dh = T\left(\frac{\partial s}{\partial T}\right)_p dT + \left[V + T\left(\frac{\partial s}{\partial P}\right)_T\right] dP. \tag{4.8}$$

By further comparing the coefficients of dP and dT in equation (4.5) and (4.8), the relations

$$\left(\frac{\partial h}{\partial P}\right)_T = V + T\left(\frac{\partial s}{\partial P}\right)_T, \tag{4.9}$$

$$\left(\frac{\partial s}{\partial T}\right)_P = \frac{c_p}{T}, \tag{4.10}$$

are obtained. By using one of the Maxwell relations, see Appendix E, equation (4.9) can then be transformed to,

$$\left(\frac{\partial h}{\partial P}\right)_T = V - \left(\frac{\partial V}{\partial T}\right)_p. \tag{4.11}$$

Lastly, by substituting this relation into equation (4.5) the expression for the enthalpy change becomes [22],

$$dh = c_p\, dT + \left[V - \left(\frac{\partial V}{\partial T}\right)_p\right] dP. \tag{4.12}$$

This equation, however, can be further simplified by assuming ideal gas,

$$dh = c_p\, dT. \tag{4.13}$$

This relation is also found valid for liquids as the enthalpy of a liquid is, as for an ideal gas, assumed to be independent of the pressure [21].

The expression of the enthalpy in the vapour and liquid phase in the flash tank model can then be obtained. By assuming constant c_p, and choosing the reference condition to be a specific temperature in the liquid phase, and an enthalpy contribution as in equation (4.2) the enthalpies can be calculated as follows,

$$h_L = \sum_i^{nc} x_i \cdot c_{p,L,i} \cdot (T - T_{ref}), \tag{4.14}$$

$$h_V = \sum_i^{nc} y_i \ \left(\Delta h_{vap,i} + c_{p,V,i} \cdot (T - T_{ref})\right). \tag{4.15}$$

In this context, $\Delta h_{vap,i}$ denotes the vaporisation enthalpy of component i, while $c_{p,L,i}$ and $c_{p,V,i}$ is the heat capacity of component i in liquid and gas phase, respectively.

4.3 Phase Equilibrium

Phase equilibrium is a concept which describes the distribution and balance of different phases within a system. For phase equilibrium to be established between two different phases, α and β, the global Gibbs free energy, G needs to be minimised, such that

$$dG = dG^\alpha + dG^\beta = 0. \tag{4.16}$$

Under the assumption of constant temperature and pressure, Gibbs free energy in two-phase systems can be expressed as follows,

$$(dG)_{T,P} = \sum_{i=1}^{nc} \mu_i^\alpha dn_i^\alpha + \sum_{i=1}^{nc} \mu_i^\beta dn_i^\beta = 0, \tag{4.17}$$

where nc is the number of components, μ_i is the chemical potential and n_i is the molar mass of component i. According to the law of material balance, the number of moles leaving phase α needs to be equal to the moles appearing in phase β, such that $dn_i^\beta = -dn_i^\alpha \ \forall i \in [1, nc]$ [23]. By taking this into account, equation (4.17) can be rewritten as follow,

$$(dG)_{T,P} = \sum_{i}^{nc} (\mu_i^\alpha - \mu_i^\beta) dn_i^\alpha = 0. \tag{4.18}$$

The molar mass change of a specific component in phase α, dn_i^α, can further be assumed to be independent close to the equilibrium point, such that the equilibrium condition becomes,

$$\mu_i^\alpha = \mu_i^\beta, \forall i \in [1, nc]. \tag{4.19}$$

According to this condition, a component needs to have the same chemical potential in all the existing phases for phase equilibrium to be established. The last criterion of phase equilibrium is that the temperature and the pressure are uniform throughout the system, such that [24]

$$T^\alpha = T^\beta \ , \ P^\alpha = P^\beta. \tag{4.20}$$

4.3.1 Vapour-liquid Equilibrium Calculations

In a system with established vapour-liquid equilibrium, the concentrations can be approximated through certain theories. Henry's law constant, also known as the vapour-liquid distribution ratio, describes how a chemical component distributes itself between the liquid and vapour phases,

$$K_i = \frac{y_i}{x_i}, \tag{4.21}$$

where y_i and x_i are the mole fraction of vapour and liquid phase, respectively. A high K value indicates that the component prefers to be in the vapour phase, while a low value favours the liquid phase. The value of K is characteristic of each chemical component and is dependent on temperature, pressure and phase compositions. However, the phase composition dependency can in most cases be neglected. The K value can be further calculated through Raoult's law with the assumption of an ideal mixture,

$$K_i = \frac{p_i^{sat}(T)}{p}, \tag{4.22}$$

where p_i^{sat}, the saturation pressure, is the vapour pressure of pure component i at temperature T. This vapour pressure can further be found through empirical relationships such as Antoine's equation,

$$\log_{10}(p_i^{sat}) = A_i - \frac{B_i}{T + C_i}, \tag{4.23}$$

where A_i, B_i and C_i are constants unique for each chemical compound i [21].

4.4 Phase Changes

Flash separators are usually assumed to operate at vapour-liquid equilibrium, between the dew and bubble point conditions, in the two-phase region. An illustration of the phase diagram for a flash separation is given in Figure 4.1. In this figure, the dew point curve indicates when a pure vapour mixture of two components starts condensing into the liquid phase. The bubble point curve, on the other hand, shows when a corresponding liquid mixture starts vaporising.

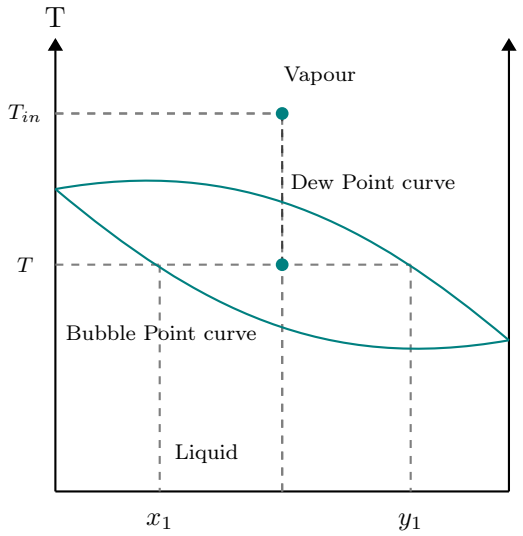


Figure 4.1: A phase diagram for a flash distillation at constant pressure, where the inlet flow is assumed to be pure vapour at T_{in}. The dew point curve, as well as the bubble point curve, are presented in green in the figure.

In the case of phase disappearance, the mixture will appear outside the two-phase region, and the vapour-liquid equilibrium will no longer be present. These phase transitions need to be accounted for in the calculations. Biegler posed an approach which involves using a relaxation parameter and incorporating complementarity relationships, such as the following equations are obtained [8],

$$y_i = \beta K_i x_i, \tag{4.24a}$$
$$\beta - 1 - s_V + s_L = 0, \tag{4.24b}$$
$$0 \leq F_L \perp s_L \geq 0, \tag{4.24c}$$
$$0 \leq F_V \perp s_V \geq 0, \tag{4.24d}$$

where β is the relaxation parameter, F_V and F_L are the vapour and liquid outlet flows, and s_V and s_L are the corresponding slack variables. This formulation ensures continuity in the model by extending the phase mole fractions into the single-phase regime, even when the corresponding phase is absent. If both phases are present, the vapour-liquid equilibrium will hold and $\beta = 1$. The complementarity constraints will in this scenario require that both slack variables are zero as

$F_L > 0$ and $F_V > 0$. However, if either the liquid phase or the vapour phase is absent, the associated slack variable becomes positive and $\beta \neq 1$. The phase equilibrium condition is hence being relaxed such that a feasible solution can be obtained. The complementarity system is necessary in order to capture the cases of phase disappearance and phase appearance, but will however make the model more computationally demanding to solve [8].

4.5 Assumptions and Model Parameters

The flash will in this thesis be modelled as a UV-flash, such that the internal energy, U, the volume, V, and the component holdups, M_i are specified. Hence, the temperature, T, the pressure P, the vapour fraction, f, and the composition in both the vapour and liquid phase, (x_i, y_i), can be found through flash calculations [25],

$$(U, V, Mc_i) \rightarrow (T, P, f, x_i, y_i). \tag{4.25}$$

An illustration of the flash tank is given in Figure 4.2, where the molar inlet flow, its composition and its temperature are specified.

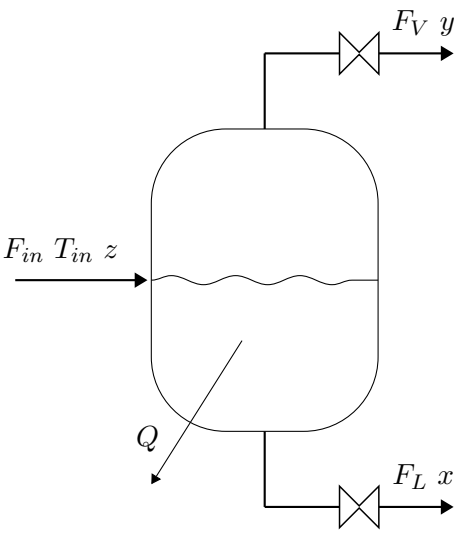


Figure 4.2: An illustration of the flash tank. F_{in} is the inlet flow with the inlet temperature T_{in} and the composition z. The liquid and vapour flow is also shown as F_L and F_V, respectively, with their corresponding composition. Q illustrate the heat being removed from the tank.

As earlier stated, the definition of a reference condition is necessary for performing the enthalpy calculations over the system. In this thesis, the reference condition is defined as a pure component in the liquid phase at $T_{\text{ref}} = 298.15$ K. The phases are further assumed to be homogeneous, which indicates that the phases have uniform properties throughout, such that the temperature and the composition are the same in every part of the phase. Additionally, ideal gas is assumed to be a sufficient approximation for the vapour, and the liquid is assumed to be incompressible compared to

the vapour. The feed in the simulation is determined to be an equimolar mixture between methanol and water in pure vapour. The thermodynamic data for both components is presented in Table 4.1.

Table 4.1: The thermodynamic data of water and methanol. The heat of vaporisation is collected from Skogestad [21], heat capacities from Cengel [22] and Antoine equation parameters from NIST [26]

Parameter	Water	Methanol	Unit
$c_{p,L}$	75.351	81.08	kJ/(kmol K)
$c_{p,V}$	33.58	44.06	kJ/(kmol K)
h_{vap}	40.660	35.21	kJ/mol
ρ	55.33	24.72	kmol/m^3
A	4.6543	5.15853	-
B	1435.264	1569.613	-
C	−64.848	−34.846	-

4.6 Material Balances

The model has a dynamic material and energy balance which can be expressed through differential equations. The general balance of an extensive quantity, φ, is given as,

$$\frac{d\varphi}{dt} = \varphi_{in} - \varphi_{out} + \varphi_{gen} - \varphi_{loss}, \tag{4.26}$$

where φ_{gen} and φ_{loss} are the amount generated and lost by the system, respectively. By assuming that no reaction takes place, the generation and the loss term can be neglected such that the molar balance for the system's components can be simplified to,

$$\frac{dM_i}{dt} = M_{i,in} - M_{i,out} \quad i = 1, \ldots, nc. \tag{4.27}$$

Expressing the molar component holdup in terms of molar flows and compositions, the balance becomes,

$$\frac{dM_i}{dt} = F_{in} \cdot z_i - F_L \cdot x_i - F_V \cdot y_i \quad i = 1, \ldots, nc, \tag{4.28}$$

where F_{in} is the total inlet flow, F_L and F_V is the liquid and vapour flow, respectively, and z_i, x_i and y_i denote the molar fraction of component i in the different flows. For the system to be fully defined, additional equations for the molar holdups are required,

$$M_i = M_L \cdot x_i + M_V \cdot y_i \quad i = 1, \ldots, nc \tag{4.29}$$

$$\sum_{i}^{nc} M_i = M_L + M_V, \tag{4.30}$$

where M_L and M_V are the molar holdups in the liquid and vapour phases, respectively. Additionally, as the phase mole fractions are extended into the single-phase regime, the following balance is still valid and is therefore included in the model as well,

$$\sum_{i}^{nc} y_i - \sum_{i}^{nc} x_i = 0. \tag{4.31}$$

4.7 Energy Balances

The dynamic equation for the energy balance can be derived by following a similar procedure as for the material balance. The kinetic and potential energy is assumed to be negligible, resulting in a balance that revolves around internal energy. From equation (4.26), the energy balance becomes,

$$\frac{dU}{dt} = U_{in} - U_{out} + Q + W, \tag{4.32}$$

where Q denotes the heat transferred to the system and W is the supplied work. There exist several types of work. However, by the assumption of constant volume and no shaft or mechanical work, there is only flow work present in the system, $W_{flow} = pV$. By applying the definition of enthalpy $H = U + PV$, the balance for the system can then be given as [21],

$$\frac{dU}{dt} = H_{in} - H_{out} + Q, \tag{4.33}$$

and in terms of molar enthalpy and molar flows, this results in,

$$\frac{dU}{dt} = F_{in}h_{in} - F_L h_L - F_V h_v + Q, \tag{4.34}$$

where h_{in} is the molar enthalpy of the inlet flow, while h_L and h_V are the molar enthalpy of the liquid and vapour flow, respectively. The expressions for h_L and h_V were previously derived in this chapter but are restated below,

$$h_L = \sum_{i}^{nc} x_i \cdot c_{p,L,i} \cdot (T - T_{ref}), \tag{4.35}$$

$$h_V = \sum_{i}^{nc} y_i \ \left(\Delta h_{vap,i} + c_{p,V,i} \cdot (T - T_{ref})\right). \tag{4.36}$$

By using these expressions above, the total enthalpy holdup in the flash tank can be related to the molar holdups as follows,

$$H = M_L h_L + M_V h_V, \tag{4.37}$$

Lastly, to combine the enthalpy with the internal energy in the system, the definition of enthalpy is included as an equation as well,

$$H = U + PV. \tag{4.38}$$

4.8 Phase Distribution

In the derivation of the energy balance, the volume was assumed to be constant. It is further assumed that the entire volume is occupied at each time, such that the following equation is obtained,

$$V_{tot} = V_V + V_L, \tag{4.39}$$

where V_V is the volume in the vapour phase, while V_L is the volume in the liquid phase. The volume of the vapour phase can be found through ideal gas low,

$$pV_V = M_V TR, \tag{4.40}$$

where R is the gas constant. The liquid volume, however, is calculated by,

$$V_L = \frac{M_L}{\rho_L}, \tag{4.41}$$

where ρ_L is the density of the liquid in the tank computed by taking the sum of each component's contributions,

$$\frac{1}{\rho_L} = \sum_i^{nc} \frac{x_i}{\rho_i}, \tag{4.42}$$

where ρ_i is density of pure component i.

4.9 Valve Equations

The outflows, both the liquid and the vapour flow, can be determined through valve equations. The valve equations proposed are,

$$F_V = c_V \cdot \frac{V_V}{V} \cdot \frac{p - p^0}{\sqrt{|p - p^0| + \epsilon}}, \tag{4.43}$$

$$F_L = c_L \cdot \frac{V_L}{V} \cdot \frac{p - p^0}{\sqrt{|p - p^0| + \epsilon}}, \tag{4.44}$$

where p^0 is the outlet pressure, c_V and c_L are valve coefficients for the vapour and liquid flow, respectively, and $\epsilon > 0$ is a small value crucial for the function to be Lipschitz continuous at $p = p^0$. However, in the case of reverse flow, the pressure difference along the valve becomes negative, and the expressions above become undefined. To prevent this, check valves are included in the expressions through a max operator. Thus, the modified valve equations become as follows,

$$F_V = c_V \cdot \frac{V_V}{V} \cdot \max\left(0, \frac{p - p^0}{\sqrt{|p - p^0| + \epsilon}}\right), \tag{4.45}$$

$$F_L = c_L \cdot \frac{V_L}{V} \cdot \max\left(0, \frac{p - p^0}{\sqrt{|p - p^0| + \epsilon}}\right). \tag{4.46}$$

The check valves will turn off the outlet flows when the pressure difference along the valve becomes negative [7]. However, these expressions are non-smooth due to both the absolute value operator and the maximum operator. Equation (4.45) and (4.46) can, however, be reformulated through complementarity relationships. The absolute value $z := |p - p^0|$ can be expressed in the following way,

$$z = s_{avm} + s_{avp} \tag{4.47a}$$

$$p - p^0 = s_{avm} - s_{avp} \tag{4.47b}$$

$$0 \leq s_{avm} \perp s_{avp} \geq 0, \tag{4.47c}$$

where s_{avm} and s_{avp} are the complementarity variables. Furthermore, the max expression,

$$w := \max\left(0, \frac{p - p^0}{\sqrt{z + \epsilon}}\right),$$

can be equivalently expressed as

$$w = 0 + s_{wm} \tag{4.48a}$$

$$p^0 - p = (z + \epsilon)^{0.5}(s_{wp} - s_{wm}) \tag{4.48b}$$

$$0 \leq s_{wm} \perp s_{wp} \geq 0, \tag{4.48c}$$

where s_{wm} and s_{wp} is another pair of complementarity variables [8]. By including equation (4.47) and (4.48), the vapour and liquid flow can be given as,

$$F_V = c_V \cdot \frac{V_V}{V} \cdot w, \tag{4.49}$$

$$F_L = c_L \cdot \frac{V_L}{V} \cdot w. \tag{4.50}$$

A more detailed derivation of the absolute and max operator expressed in terms of complementarity relationships is given in Appendix B.

Chapter 5

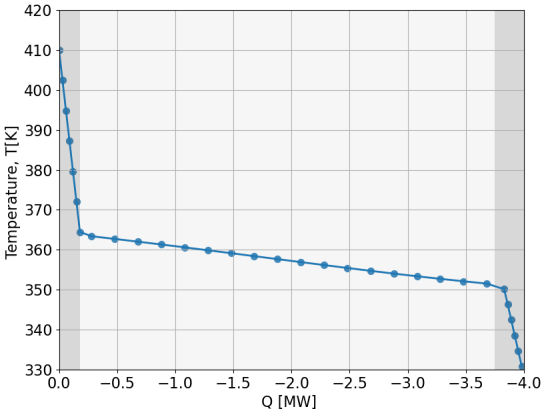
Case Study of Flash Tank

To further assess the Classic approach's ability to address MPCCs, a study was conducted on a multi-phase flash tank. The implementation based on IPOPT was chosen for this study due to its capability of solving complex problems. The section starts by presenting the results from the simulation of the steady-state flash tank, before proceeding to the dynamic version.

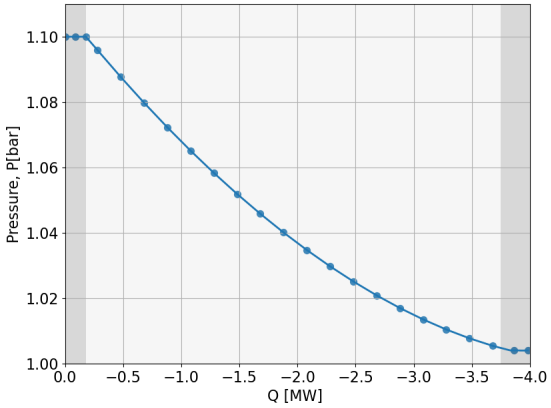
5.1 Stationary Flash Tank

In the simulations, pure vapour entered the tank at $T_{in} = 410$ K, with a flow rate of $F_{in} = 0.1$ kmol/s. Both water and methanol were present in the feed in an equimolar ratio, $z = [0.5\ 0.5]$. In the simulations, the heat removal, Q was varied to observe how the steady-state values changed accordingly. Throughout this chapter, the dark grey will indicate single-phase regions, whereas the two-phase region will be shaded in light grey.

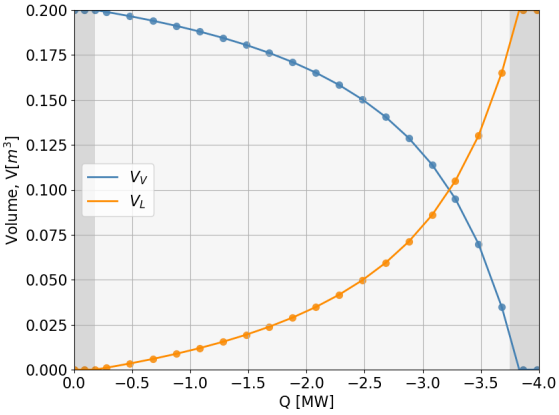
The plots of the temperature, the pressure and the volume at steady-state with respect to heat removal are presented in Figure 5.1. As long as the heat removal rate was significantly low, the system operated in the vapour-only regime. The temperature profile, presented in Figure 5.1a, shows a fast temperature decrease in this region until the dew point was reached, at $Q \approx -0.15$ MW. The system then entered the two-phase region, where the latent heat of vaporisation was released to the surrounding, resulting in a less drastic temperature decrease. As the heat removal increased, more liquid occupied the tank volume, as shown in Figure 5.1c. This change in phase distribution affected the pressure in the tank. As an ideal gas behaviour was assumed, the pressure will decrease when less vapour is present as the number of gas molecules decreases. The liquid, however, will not have a significant impact on the pressure as its compressibility is assumed to be negligible. Thus, the pressure decrease observed in Figure 5.1b is consistent with the expectations. Eventually, the heat removal was sufficient for no vapour to remain, at $Q \approx -3.75$ MW, such that the temperature decrease returned to a stepper rate, and the volume was entirely occupied by the liquid.



(a) Steady-state temperature [K]



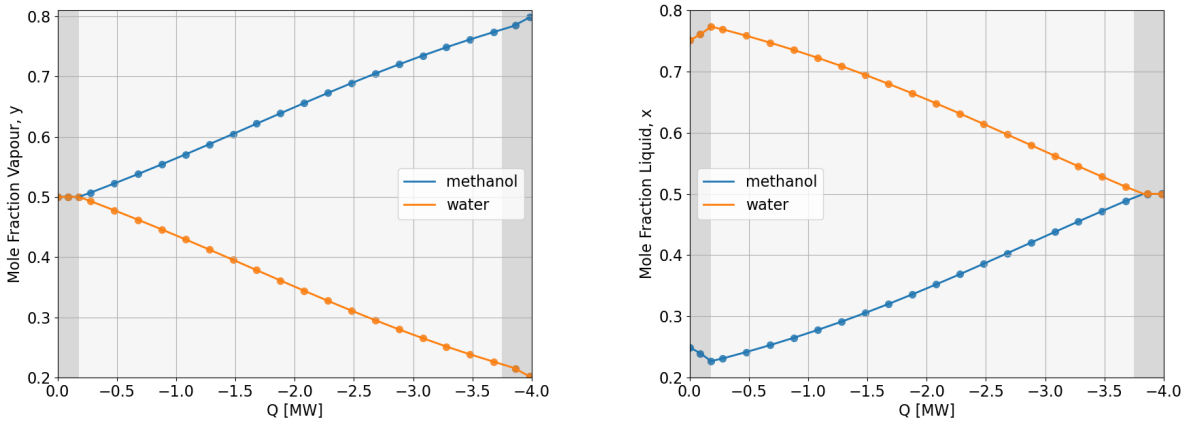
(b) Steady-state pressure [bar]



(c) Steady-state volume [m^3]

Figure 5.1: The temperature, pressure and volume in the flash tank at steady-state in relation to heat removal. The areas shaded in dark grey represent one-phase regions.

According to Figure 5.2a, the steady-state mole fraction of water in the vapour decreased as the heat removal increased. This is reasonable as water will start condensing before methanol, due to its higher saturation temperature. Water, therefore, had a higher mole fraction than methanol in the liquid, until the liquid-only regime was reached and the outflow became equimolar, see Figure 5.2b. It is worth mentioning that the mole fractions given in the one-phase region, the dark grey areas, apply only when the corresponding phase is present. The liquid phase was absent until $Q \approx -0.18$ MW, such that the fractions prior to this, do not provide any useful physical information. Similarly, when the heat removal was above $Q \approx -3.75$ MW the vapour phase disappeared and the corresponding mole fraction became meaningless.

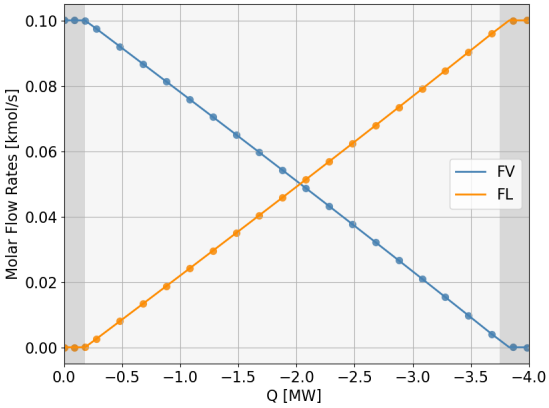


(a) Steady-state vapour mole fraction, y

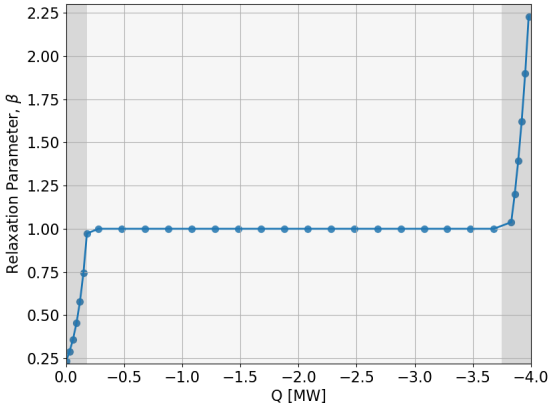
(b) Steady-state liquid mole fraction, x

Figure 5.2: The liquid and vapour mole fractions at steady-state with respect to heat removal. Dark grey areas represent one-phase regions.

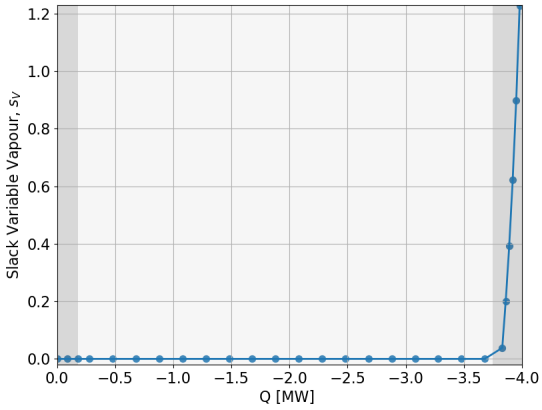
In order to account for phase disappearance, a relaxation parameter, β was included along with two slack variables, s_L and s_V. Figure 5.3b shows how the steady-state value of β changed as the heat removal increased. Initially, the flash tank operated in the vapour-only regime, such that $\beta < 1$. However, when both phases became present, in the light grey area, β increased to 1, and remained constant until the vapour phase disappeared completely. The flash tank then entered the liquid-only regime, and β was adjusted accordingly. s_L and F_L, and s_V and F_V are the complementarity pairs accounting for the phase changes. Their steady-state values with respect to heat removal are presented in Figure 5.3 as well. According to the figure, the presence of vapour flow caused the corresponding slack variable to become zero, $s_V = 0$, which is consistent with the complementarity constraint. s_V then remained at zero until Q reached a significant level causing the vapour flow to disappear. The other complementarity pair, consisting of F_L and s_L, were also in line with the complementarity restrictions. When $F_L > 0$, then $s_L = 0$ and vice versa.



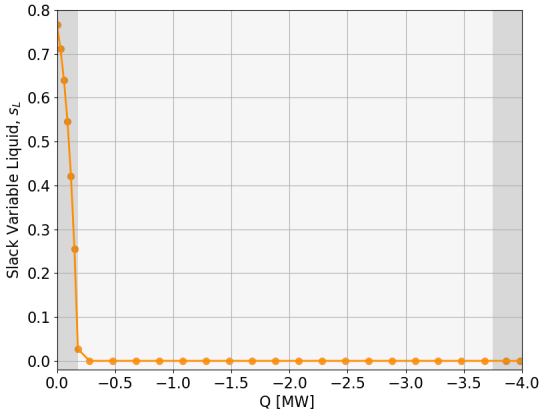
(a) Steady-state molar flows [kmol/s]



(b) Steady-state relaxation parameter, β.



(c) Slack variable for the vapour phase, s_V



(d) Slack variable for the liquid phase, s_L

Figure 5.3: The relaxation parameter, β the slack variables s_L and s_V, and the molar flow rates in the flash tank at steady state with respect to heat removal. The single-phase regions are represented in dark grey.

The result for the steady-state simulation were obtained without any significant challenges, and matched the theoretical and physical expectations as previously elaborated. This findings, therefore, strongly support the use of complementarity relationship to account for the phase changes in steady-state flash tank simulations. The obtained result also demonstrates the Classic approach's ability to tackle a stationarity flash tank system efficiently and precisely.

5.2 Dynamic flash tanks

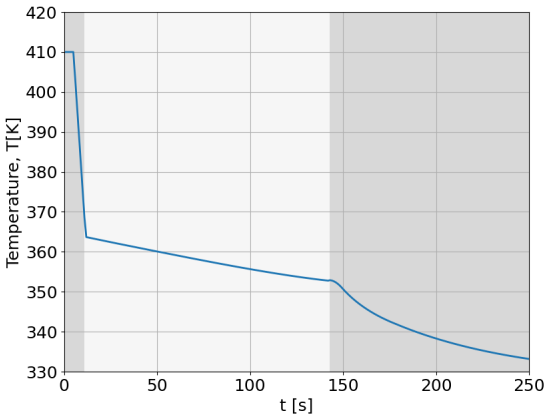
A dynamic version of the flash tank was created by including the differential equations and the continuity constraint associated with the orthogonal collocation method. The simulations were conducted with three collocation points of shifted Gauss-Legendre roots in each final element. The initial condition of the model was chosen to be the values obtained from the stationary flash tank with zero heat removal. The heat removal in the dynamic flash tank was gradually decreased until $t = 150$ s, as described in Algorithm 5. Subsequently, the heat removal remained consistently at -4 MW.

Algorithm 5 Decreasing the heat removal

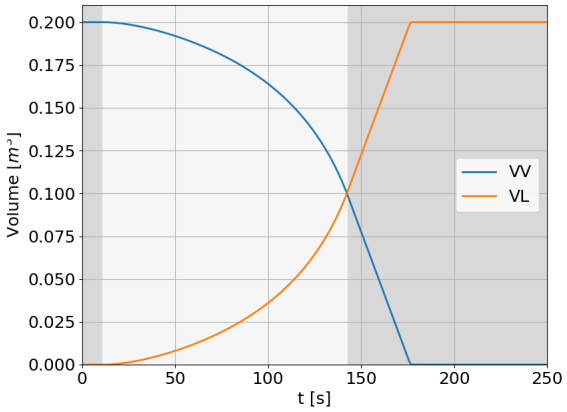
$t \leftarrow 1$
while $t \leq 250$ **do**
  if $t < 5$ **then**
    $Q \leftarrow 0$
  else if $5 \leq t < 150$ **then**
    $Q \leftarrow \frac{-4}{150-5} \cdot (t - 5)$
  else
    $Q \leftarrow -4$
  end if
  $t \leftarrow t + 1$
end while

The temperature, the volume and the heat removal profiles are presented in Figure 5.4. As in the previous section, the dark grey areas represent the single-phase regions, while the light grey shows the two-phase regime. Initially, the temperature decrease was large as no phase transition was present. However, this decrease declined as vapour condensed due to the released latent heat of vaporisation counteracting the temperature decrease. As the system entered the two-phase region, the volume of the tank was gradually more occupied by the liquid, which can be observed in Figure 5.4b. This result is consistent with the results obtained from the steady-state investigation, as both the volume and the temperature profiles follow the same pattern. Moreover, it is noteworthy that the system has not reached steady-state at $t = 250$ s as the temperature was still decreasing at this

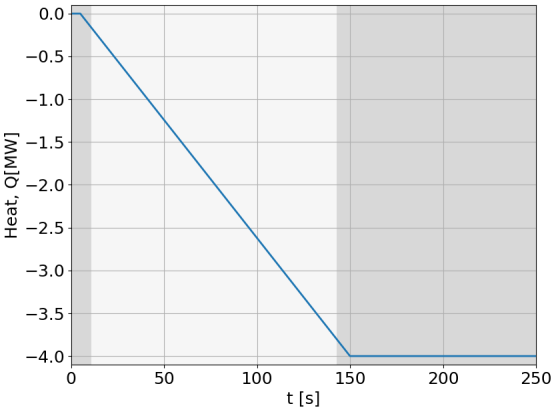
time. A longer time horizon is, however, possible, but this horizon was considered appropriate and sufficient to illustrate the dynamic flash tank simulation.



(a) Temperature [K]



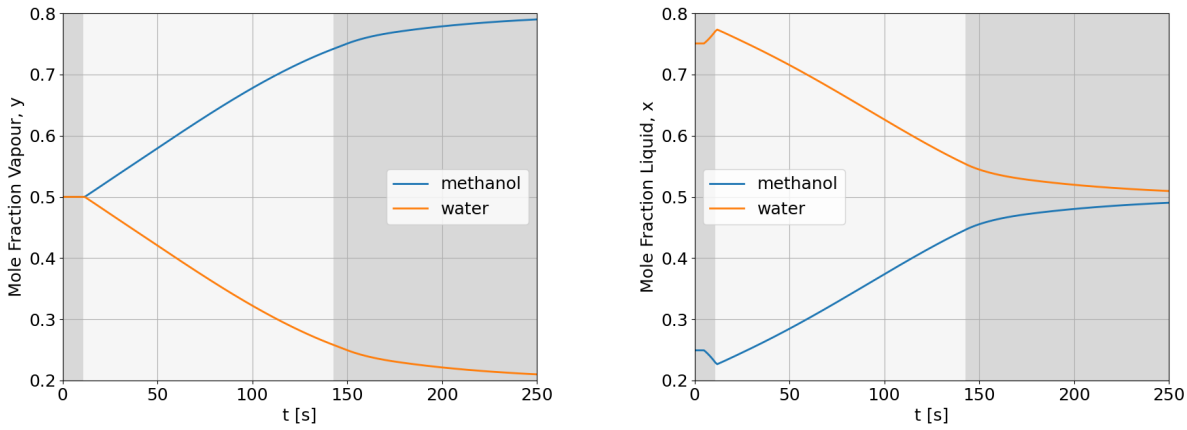
(b) Volume [m^3]



(c) Heat removal [MW]

Figure 5.4: The temperature, volume and heat removal change with time, t. Dark grey areas represent one-phase regions.

The component composition in both the vapour and the liquid phase is presented in Figure 5.5. These graphs will not be discussed in detail, as they follow the same pattern as the previously discussed steady-state observations. However, to briefly recap, methanol dominated the vapour phase, while water occupied the liquid as expected due to the difference in saturation temperature. As previously stated, the mole fractions are only useful when the corresponding phase exists, as the values are assigned only to maintain continuity in the model equations. Overall, the composition profiles correspond to the expectation based on the physical properties of the two components, and the result from the steady-state simulation as well.

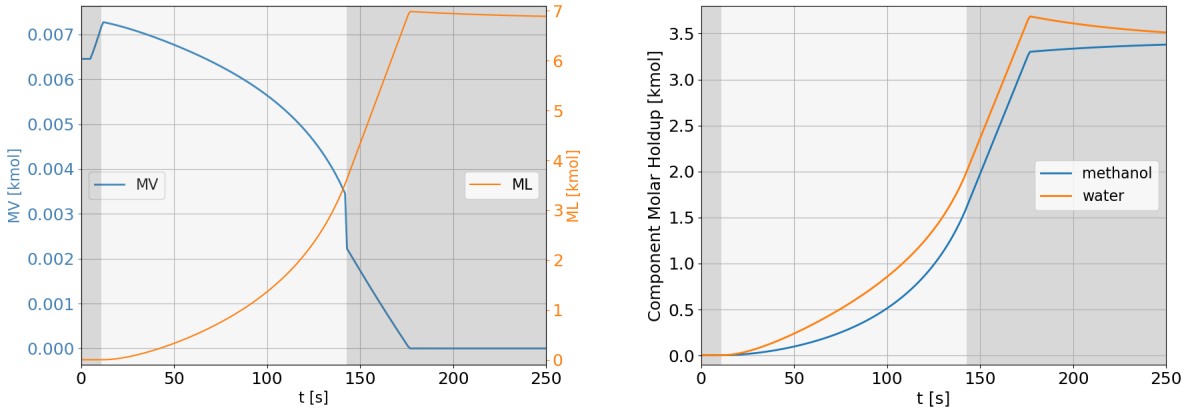


(a) Mole fraction vapour, y

(b) Mole fraction liquid, x

Figure 5.5: The liquid and vapour composition in the flash tank with respect to time. The dark grey areas represent single-phase regions. The liquid is absent until $t \approx 10$ s such that the mole fraction of liquid before this time is meaningless. The same applies for the mole fraction of vapour above $t \approx 175$ s, as the vapour flow is zero subsequent to this.

The molar component holdups and the molar phase holdups are presented in Figure 5.6. The liquid and vapour holdups results, presented in Figure 5.6a, correspond to the previous observations. As the amount of heat removed increased, the vapour holdup decreased while the liquid holdup increased. Further, it can be observed through Figure 5.6b that both components' holdup increased with time. This increase is expected due to the density difference between the liquid and vapour phases. A more significant component holdup was necessary when the liquid amount increased to ensure that the entire volume was occupied at all times. A second observation is that the molar holdup of water was constantly slightly above the molar holdup of methanol. As water constituted most of the liquid, this outcome was predictable.



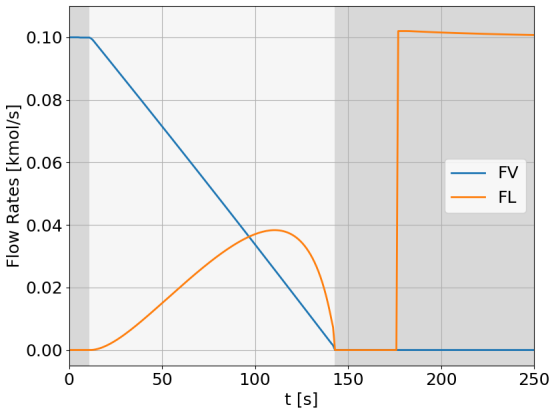
(a) Vapour and liquid molar holdup [kmol].

(b) Component molar holdup [kmol]

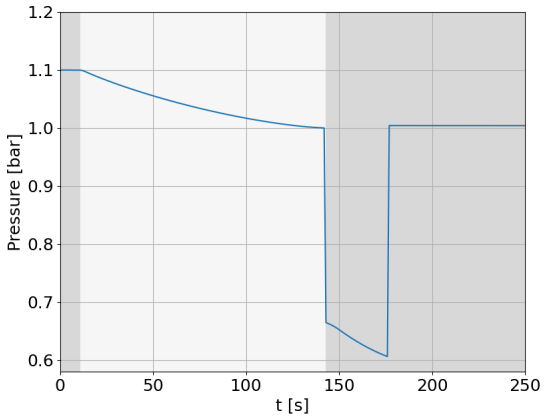
Figure 5.6: The liquid and vapour holdups and the component molar holdups with respect to time. The dark grey areas represent the single-phase regions.

As previously mentioned, the phase transitions were represented through complementarity pairs. Figure 5.7 shows how the molar flows, the relaxation variable and the slack variables changed over time. In the initial stage, the tank was in the vapour-only regime. The relaxation parameter was then below one, while the slack variables were $s_V = 0$ and $s_L \neq 0$. However, as soon as the tank entered the two-phase region, at $t \approx 10$ s, β increased to one and remained at this value as long as the tank existed within this region. In addition, when the liquid started flowing out of the tank, the associated slack variable, s_L, decreased to zero, which agrees with the complementarity constraints. According to Figure 5.7a, the flow rates were both zero at $t \approx 144$ s. This is a result of the pressure of the flash tank decreasing below the outlet pressure, which can be observed in Figure 5.7b. At this moment, the tank was mostly filled with liquid, which is less dense than vapour. As a result, the pressure dropped to maintain the tank's total volume. As both $F_V = 0$ and $F_L = 0$ the corresponding slack variables became non-zero, and $\beta \neq 1$. However, at $t \approx 175$ s the liquid flow started increasing again such that s_L returned to zero, and $\beta > 1$, as expected according to the relaxed liquid-vapour equilibrium equation.

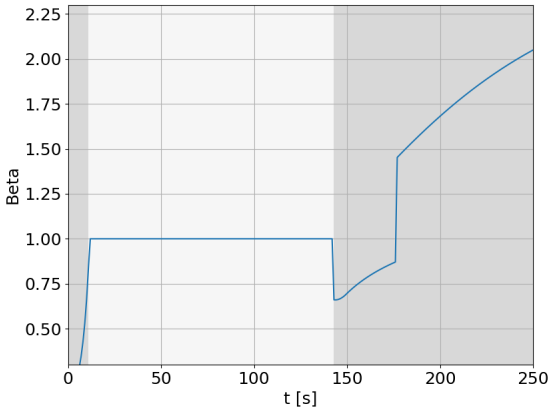
In the valve equations for the flow rates, the max operator and the absolute value are expressed in terms of complementarity relationships as well. Figure 5.8a and 5.8b provide the values of the complementarity variables for the max operator, s_{wp} and s_{wm}, respectively. When the pressure in the tank was above the outlet pressure, $s_{wm} > 0$ while $s_{wp} = 0$. This corresponds to the first expression in the valve equations, equation (4.46) and (4.45), to be dominant. However, when the drastic pressure drop inside the tank appeared, at $t \approx 144$ s, both outflows became zero as the second term in the valve equations became active. This can be observed in the slack variables as $s_{wm} = 0$ and $s_{wp} > 0$. Eventually, the pressure returned to normal, which caused s_{wp} to increase



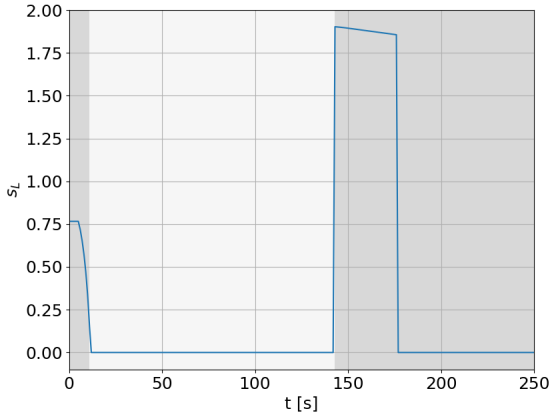
(a) The molar flows [kmol/s]



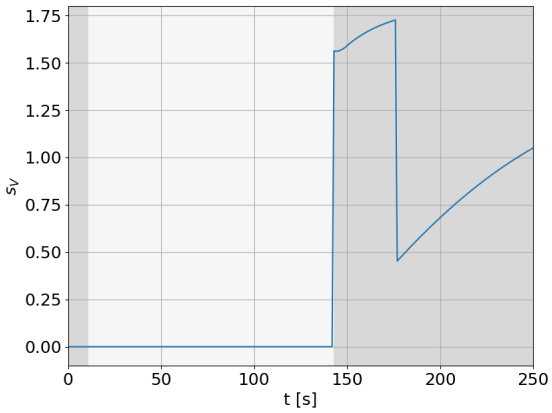
(b) Pressure [bar]



(c) The relaxation parameter, β.



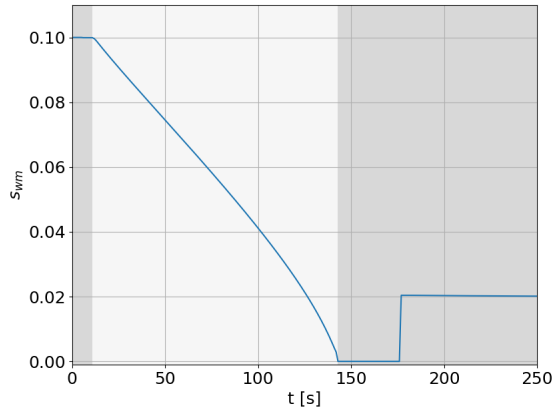
(d) Slack variable for the liquid phase, s_L.



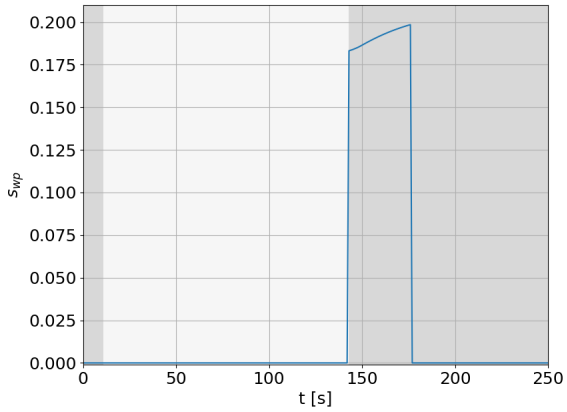
(e) Slack variable for the vapour phase, s_V.

Figure 5.7: The values of the molar flow rates, the pressure, the relaxation parameter, β, and both the slack variables for each of the two phases with respect to time, t. The areas shaded in dark grey represent one-phase regions.

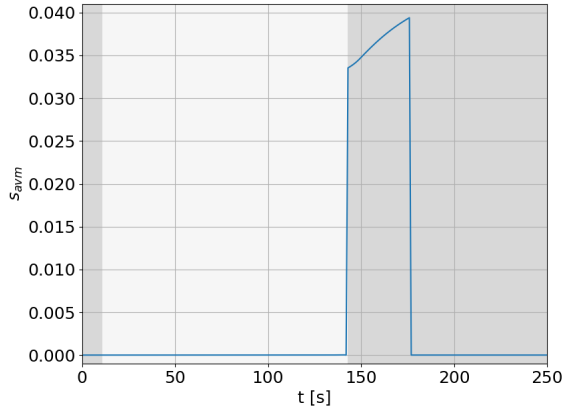
and s_{wp} to return to zero. Similar patterns can be seen in the complementarity pair representing the absolute value, s_{avm} and s_{avp}. Initially, when the pressure is above the outlet pressure, $s_{avm} = 0$ and $s_{avp} > 0$. Eventually, when the pressure dropped, s_{avm} increased while s_{avp} became zero. At $t \approx 175\ s$, they returned to their original state as the pressure in the flash tank increased. According to this simulation, the complementarity pairs were able to represent both the max operator and the absolute operator correctly.



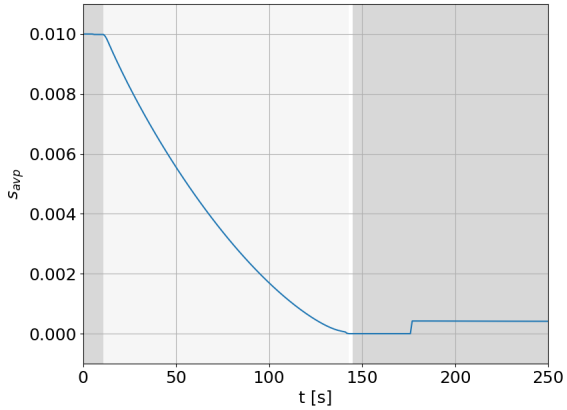
(a) Complementarity variable for max operator, s_{wm}.



(b) Complementarity variable for max operator, s_{wp}.



(c) Complementarity variable for absolute value, s_{avm}



(d) Complementarity variable for absolute value, s_{avp}.

Figure 5.8: The values of the complementarity variables for the max operator, s_{wm} and s_{wp}, and the absolute expression, s_{avm} and s_{avp} in the valve equations, see equation (4.46) and (4.45), with respect to time. The areas shaded in dark grey represent one-phase regions.

The obtained result agrees with the anticipated physical behaviour, reinforcing the reliability of the Classic approach. The result is also consistent with the result presented in Reed's master thesis, which is based on a non-smooth model approach rather than using complementarity constraints as in this thesis. This similarity further strengthens the belief that the model is correctly implemented.

It should be pointed out that the flash tank model is an oversimplified representation of a flash tank system and does not reflect reality entirely. A real flash tank system would remove heat at a slower pace and allow for flow reversal, preventing the sudden pressure drop observed in the simulations. However, the obtained result is in line with the model presented in this thesis.

It was quite challenging to make the algorithm converge to a solution that made sense from a physical perspective. To make the problem easily converge, the time horizon was extended compared to Reed's proposal, such that heat could be removed at a slower rate. However, further adjustments were necessary. At first, the entire time horizon, along with all final elements, was given to the algorithm, but the solver was unable to converge. To reduce the size of the optimisation problem, the algorithm was provided with one or a few final elements at a time. To ensure continuity in the simulation, each final element's initial condition was set equal to the previous element's result. Splitting the optimisation problem in this manner would normally be forbidden but since the objective function is set to a constant value of 1, rather than minimizing costs or maximising profits, this can be done. Trial-and-error testing was used to determine the number of elements that should be provided at once. At first, it was found beneficial to provide the solver with one final element at a time until t = 90. Then, the entire time horizon $t \in [90, 200]$ was provided at once. The last final elements, however, had to be provided one at a time as initially. This splitting is also presented in Algorithm 6. However, there may be other combinations which may yield convergence as well. To further help the solver in the direction of the optimal solution, the temperature in one element was constrained to be below or equal to the temperature of the previous element. This is a reasonable constraint as heat is removed in the model such that a decrease in temperature is to be expected. Additionally, constraints on the flow rates and the pressure were included as well. This seemed to make the algorithm successfully converge to a meaningful solution, which corresponds with the expectations as well as Reed's findings. The implemented code for this simulation can be found in Appendix G.3.

Algorithm 6 : Splitting of time horizon in dynamic flash tank.
One final element, nfe, corresponds to one second.

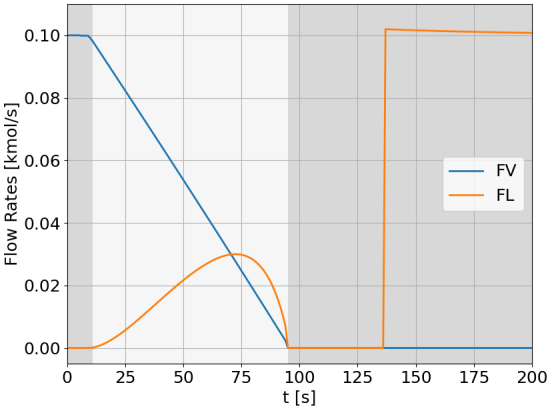
```
t ← 1
while t ≤ 250 do
    if t > 90 then
        nfe ← 1
        Classic solve MPCC
        t ← t + 1
    else if t = 90 then
        nfe ← 111
        Classic solve MPCC
        t ← 201
    else
        nfe ← 1
        Classic solve MPCC
        t ← t + 1
    end if
end while
```

5.2.1 Simulation Problems

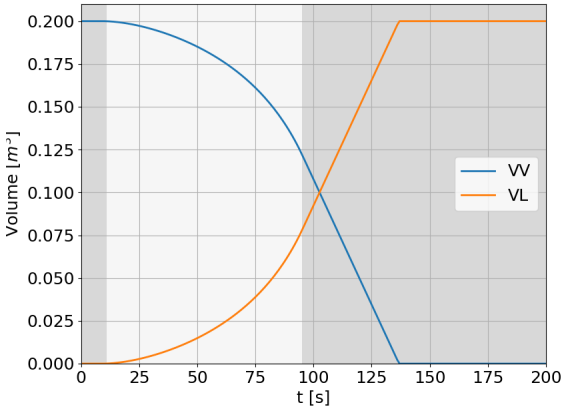
Several attempts were carried out to achieve convergence to a meaningful solution. In order to explain the issues encountered during the simulations, an examination of a specific simulation which did not achieve the intended results will be further conducted. One final element at a time was provided to the solver in this specific attempt. Additionally, the rate of heat removal was chosen to be greater than in the successful simulations, consequently leading to a shorter time horizon of 200 s. Except for these changes, the constraints and model equations were identical to the successful simulation.

The flow rate, temperature, volume, and component molar holdup profiles are presented in Figure 5.10. According to these results, the simulation appears to be accurate. These profiles also match the physical predictions and are identical to the result obtained in the previous section. However, the slack variables, the relaxation parameter and the pressure profiles presented in Figure 5.9 reveal the inaccuracies in the simulation. At $t \approx 95$ s the pressure inside the tank goes below the outlet pressure causing the outlet flow valves to close. This allows the pressure to vary to a greater extent without violating any constraints, causing the oscillations observed in Figure 5.10a. The oscillatory behaviour of the pressure reveals the lack of process insights within the simulation, which emphasise the limitations of the model formulation. An oscillatory behaviour can also be observed in Figure 5.10c and 5.10d showing the s_L and s_V profiles, respectively. When the valves closed, the outlet flows became zero, $F_L = 0$ and $F_V = 0$. Consequently, this allowed the corresponding

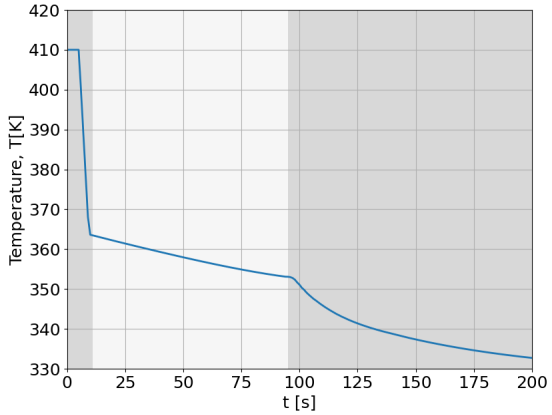
slack variables s_L and s_V to be either zero or above zero. However, the results show that the slack variables, and the relaxation parameter as well, are not uniquely defined, indicating that multiple solutions exist. The rest of the results from this specific simulation can be found in Appendix D.



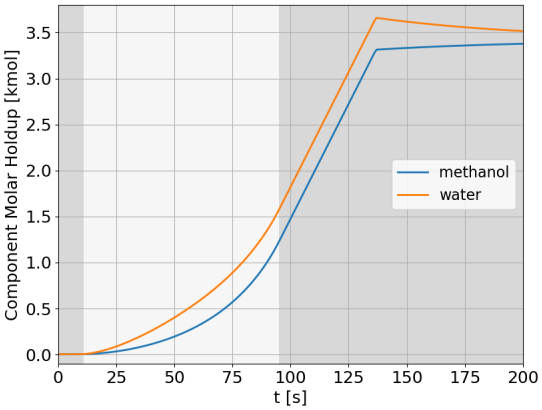
(a) The molar flow rates [kmol/s]



(b) The volume [m^3]

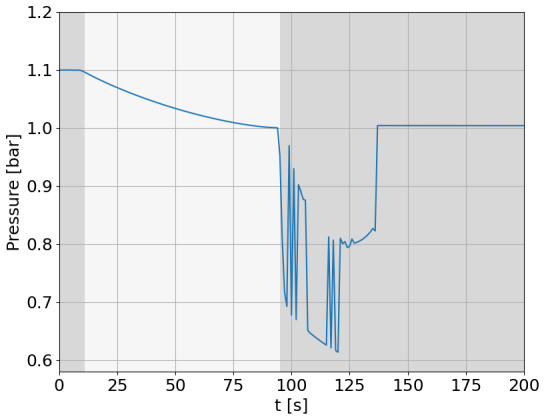


(c) The temperature [K]

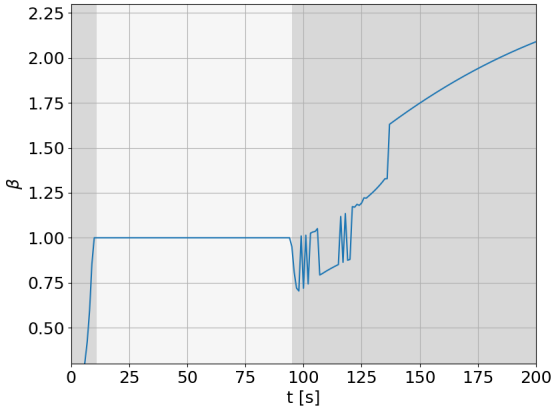


(d) The component molar holdup [kmol]

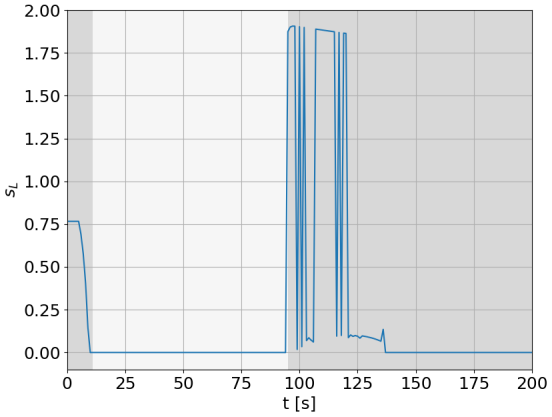
Figure 5.9: The result of a failed simulation. The molar flow rates, the volume, the temperature and the molar holdup with respect to time. The dark grey areas shows the single-phase regions.



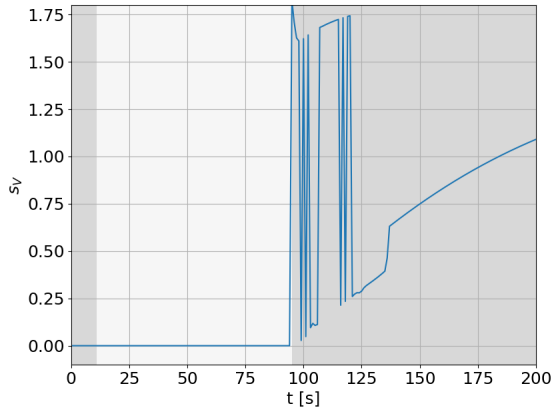
(a) The pressure [bar]



(b) The relaxation parameter β



(c) The slack variable for the liquid, s_L



(d) The slack variable for the vapour, s_V

Figure 5.10: The result from a failed simulation showing oscillations. The pressure, the slack variables and the relaxation parameter, β with respect to time.

This particular simulation reveals that the model formulation, and perhaps the Classic implementation, is not as robust and well-defined as desired after all. As previously mentioned, the simulations lack process insights and are consequently highly dependent on the system being well-posed and uniquely defined. In the situation where none of the phases is present, the equations accounting for the phase equilibrium and the phase transitions may appear inadequate. The results apply that convergence of the flash tank was best achieved when only a single phase was present. In the case of phase transition, and especially when $F_L, F_V = 0$, it is likely that the MPCC solutions do not satisfy strong stationarity or B stationarity, resulting in the convergence issues encountered. If it was possible to ensure convergence to strong stationarity points only, the results presented in this section might be avoidable. However, this is beyond the scope of this thesis and will not be further discussed.

Chapter 6

Final Remarks

As previously stated, MPCCs are generally hard to solve due to some inherent properties. This thesis explored two interior-penalty algorithms, the Classic and Dynamic, for solving such problems. Important implementation aspects were discussed, and a performance evaluation was conducted. These approaches were implemented as extensions of already developed interior point-solvers. The in-house developed Int Point Solver was easily adapted for the additional features required by the algorithms, as the various parts of the solver were readily accessible. As curtain concepts related to performance had yet to be included, this solver was deemed appropriate for more minor problems. This implementation demonstrated that the Dynamic algorithm outperformed the Classic algorithm when faced with unbounded penalty problems. However, except for such cases, the Classic algorithm was sufficient for the majority of the problems. In order to explore the methods more thoroughly, a case study was performed on a flash tank. IPOPT was chosen as the implementation basis for this study due to its remarkable proficiency with sparse systems. However, achieving a successful integration between IPOPT and the additional features turned out to be challenging. In order to overcome the problems encountered, the barrier parameter was adjusted manually to prevent certain interruptions. Unfortunately, problems were still associated with the Dynamic algorithm, such that the case study was conducted with the Classic algorithm.

The results from the steady-state flash tank simulations were promising. The Classic approach was able to adjust the penalty parameter effectively, resulting in a solution that made sense from a physical perspective. The associated complementarity constraints and the relaxation parameter β, were assigned values which corresponded with the expectations. Despite the encouraging result, several challenges were encountered with the dynamic version. Adjustments were required for the algorithm to return a valid solution according to the physical assumptions. The solver had to be provided with fewer elements than anticipated, the time horizon needed to be extended and additional constraints were required. The oscillating behaviour could result from the problem formulations not being unique. Several combinations of the variables were possible, such that the solver had to be guided in the right direction. The deficiencies in the implementation could also contribute to

some of the convergence issues encountered. A successful simulation was eventually obtained, which demonstrated that the Classic implementation can, with certain modifications, address a dynamic flash tank.

To conclude, both the Dynamic and the Classic approach appeared promising in the problem investigation. However, the study was limited to rather small-scale problems, such that a more comprehensive study is needed to make an overall conclusion. The dynamic simulation revealed that the flash tank model need to be modified, as it was impossible to achieve an accurate simulation using the current model without providing guidance. Thus, this case study demonstrated that MPCCs are in fact demanding to solve. The simulation issues could, however, also arise from an inaccurate algorithm implementation as well. Overall, the findings in this master thesis, provide valuable insights into the potential of the methods but also emphasise the need for further research on this subject.

6.1 Recommendations for Further Work

In this thesis, the implementation was carried out with the programming language Julia. However, IPOPT is primarily coded in another coding language, such that a conversion is done along the way. As a result, it is difficult to access all the steps within the algorithm. Further research could therefore be to use a C API. API is short for Application Programming Interface and C API specifically refers to an interface designed with the C programming language. For the purpose of this thesis, the C API may be more suited than the JuMP interface, as more precise control of the algorithm is allowed. Another suggestion is using a solver primarily coded in Julia, with efficient sparse handling included. Such a solver could be MadNLP [27], which is a relatively newer optimization solver compared to IPOPT.

As pointed out, an improvement in the model formulation may be necessary as well to obtain a well-posed, uniquely defined system. A suggestion could be to include the slack variables in the objective function. By doing so, it may be possible to improve convergence. However, a deep process insight is required to make such adjustments work properly in the simulations. There may be other improvements in the simulations that can be carried out as well. As previously stated, the flash tank model is a significantly simplified representation of a flash tank system. In the flash tank model, the rate of heat removal is significantly faster than what would typically occur in a real flash tank system. Additionally, in a real flash tank system, the presence of check valves is not employed, such that reverse flows are allowed. The behaviour of the molar flows and the pressure in the figures are correct according to assumptions but are not representative of a real flash tank system. Removing the check valves such that reverse flows are possible and reducing the rate of heat removal may therefore be suggestions for improvements. However, this may also increase the computational demand of the simulations.

Bibliography

[1] Alberto Seeger. *Recent Advances in Optimization*. Springer-Verlag Berlin Heidelberg, 2006.

[2] BT Baumrucker, Jeffrey G Renfro, and Lorenz T Biegler. Mpec problem formulations and solution strategies with chemical engineering applications. *Computers & Chemical Engineering*, 32(12):2903–2913, 2008.

[3] M Teresa T Monteiro and José Filipe P Meira. A penalty method and a regularization strategy to solve mpcc. *International Journal of Computer Mathematics*, 88(1):145–149, 2011.

[4] Tim Hoheisel, Christian Kanzow, and Alexandra Schwartz. Theoretical and numerical comparison of relaxation methods for mathematical programs with complementarity constraints. *Mathematical Programming*, 137(1):257–288, 2013.

[5] Sven Leyffer. Mathematical programs with complementarity constraints. *SIAG/OPT Views-and-News*, 14(1):15–18, 2003.

[6] Marius Reed. Nonsmooth modelling of multiphase multicomponent heat exchangers with phase changes. Master's thesis, NTNU, 2018.

[7] Ali M Sahlodin, Harry AJ Watson, and Paul I Barton. Nonsmooth model for dynamic simulation of phase changes. *AIChE Journal*, 62(9):3334–3351, 2016.

[8] Lorenz T Biegler. *Nonlinear programming: concepts, algorithms, and applications to chemical processes*. SIAM, 2010.

[9] Sven Leyffer, Gabriel López-Calva, and Jorge Nocedal. Interior methods for mathematical programs with complementarity constraints. *SIAM Journal on Optimization*, 17(1):52–77, 2006.

[10] Ann Iren Fossøy. Evaluation of strategies for solving mathematical programs with complementarity constraints, 2023. Specialization Project, NTNU.

[11] Caroline S.M Nakama. Interior point solver algorithm. *Internal report*, January 2023. Unpublished.

[12] Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106(1): 25–57, 2006.

[13] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999.

[14] Lorenz T. Biegler. *Nonlinear Programming*. Society for Industrial and Applied Mathematics, 2010.

[15] Nikolaos Ploskas, Nikolaos Samaras, et al. *Linear programming using MATLAB®*, volume 127. Springer, 2017.

[16] Sven Leyffer and Todd S Munson. A globally convergent filter method for mpecs. *Preprint ANL/MCS-P1457-0907, Argonne National Laboratory, Mathematics and Computer Science Division*, 2007.

[17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1), 2017. URL https://doi.org/10.1137/141000671.

[18] Miles Lubin, Oscar Dowson, Joaquim Dias Garcia, Joey Huchette, Benoît Legat, and Juan Pablo Vielma. Jump 1.0: Recent improvements to a modeling language for mathematical optimization. *Mathematical Programming Computation*, 2023. In press.

[19] Sven Leyffer. MacMPEC. https://wiki.mcs.anl.gov/leyffer/index.php/MacMPEC, 2015. Accessed: 12.08.22.

[20] Junior D Seader, Ernest J Henley, and D Keith Roper. *Separation process principles: With applications using process simulators*. John Wiley & Sons, 2016.

[21] Sigurd Skogestad. *Chemical and energy process engineering*. CRC press, 2008.

[22] Yunus A Cengel, Michael A Boles, and Mehmet Kanoğlu. *Thermodynamics: an engineering approach*, volume 5. McGraw-hill New York, 2011.

[23] Morten Helbæk, Signe Kjelstrup, and Morten Helb. *Fysikalsk kjemi*. Fagbokforl., 2006.

[24] Tore Haug-Warberg. Den termodynamiske arbeidsboken. *Kolofon forlag AS*, 2006.

[25] P Flatby, S Skogestad, and P Lundström. Rigorous dynamic simulation of distillation columns based on uv-flash. In *Advanced Control of Chemical Processes 1994*, pages 261–266. Elsevier, 1994.

[26] Eds. P.J. Linstrom and W.G. Mallard. *NIST Chemistry WebBook*. NIST Standard Reference Database Number 69.

[27] Sungho Shin, Carleton Coffrin, Kaarthik Sundar, and Victor M Zavala. Graph-based modeling and decomposition of energy infrastructures. *arXiv preprint arXiv:2010.02404*, 2020.

Appendices

Appendix A

Units Used in Julia

Table A.1: The units of the parameters used in the simulations in Julia.

Parameters	**Description**	**Unit**
ρ	The density	$\mathrm{Mmol/m^3}$
$c_{p,L}$	The specific heat capacity, liquid	$\mathrm{MJ/(kmol\,kK)}$
$c_{p,V}$	The specific heat capacity, vapour	$\mathrm{MJ/(kmol\,kK)}$
h_{vap}	The heat of vaporisation	MJ/kmol
T_{ref}	The reference temperature	kK

Table A.2: The units of the variables used in the simulations in Julia.

Variables	**Description**	**Unit**
U	The internal energy	MJ
H	The enthalpy	MJ
M_i	The component molar holdup	kmol
M_L	The liquid molar holdup	kmol
M_V	The vapour molar holdup	kmol
F_L	The liquid outflow	kmol/s
F_V	The vapour outflow	kmol/s
ρ_L	The liquid density	Mmol/m^3
h_L	The molar liquid enthalpy	MJ/kmol
h_V	The molar vapour enthalpy	MJ/kmol
T	The temperature	kK
p	The pressure	MPa
V_V	The vapour volume	m^3
V_L	The liquid volume	m^3
$p_{sat,i}$	The saturation pressure	MPa
x_i	The liquid molar fraction	-
y_i	The vapour molar fraction	-
k_i	The equilibrium constant	-

Appendix B

Complementarity Reformulations

Complementarity formulation can be used to represent non-smooth functions such as the absolute value and the max operator. The absolute value operator $w = |f(x)|$ can be equivalently expressed as,

$$w = f(x)y \tag{B.1}$$

$$y = \arg\left\{\min_{\hat{y}} -f(x)\hat{y} \quad \text{s.t} -1 \leq \hat{y} \leq 1\right\} \tag{B.2}$$

The optimality condition to the optimisation problem can be formulated as,

$$f(x) = s_{avm} - s_{avp}, \tag{B.3a}$$

$$0 \leq s_{avm} \perp (1 - y) \geq 0, \tag{B.3b}$$

$$0 \leq s_{avp} \perp (y + 1) \geq 0. \tag{B.3c}$$

It is then possible to eliminate y by manipulating equation (B.3b) and (B.3c). Lastly, by inserting equation (B.3a) into equation (B.1) the final representation of the absolute value operator by utilising complementarity formulations becomes,

$$z = s_{avm} + s_{avp}, \tag{B.4a}$$

$$f(x) = s_{avm} - s_{avp}, \tag{B.4b}$$

$$0 \leq s_{avm} \perp s_{avp} \geq 0. \tag{B.4c}$$

A similar approach can be followed to rewrite the max operator $w = \max\{f(x), u\}$. The max operator can be expressed as follow,

$$w = f(x) + (u - f(x))y, \tag{B.5}$$

$$y = \arg\left\{\min_{\hat{y}}(f(x) - u)\hat{y} \quad \text{s.t} \;\; 0 \leq \hat{y} \leq 1\right\}. \tag{B.6}$$

The optimality condition for this problem becomes,

$$w = f(x) + (u - f(x))y \tag{B.7a}$$
$$f(x) - u = s_{wp} - s_{wm}, \tag{B.7b}$$
$$0 \leq s_{wm} \perp (1 - y) \geq 0 \tag{B.7c}$$
$$0 \leq s_{wp} \perp y \geq 0 \tag{B.7d}$$

By manipulating these equations such that y is eliminated, the final expression for the max operator in terms of complementarity formulations becomes [8],

$$w = f(x) + s_{wm}, \tag{B.8a}$$
$$f(x) - u = s_{wp} - s_{wm}, \tag{B.8b}$$
$$0 \leq s_{wp} \perp s_{wm} \geq 0. \tag{B.8c}$$

Appendix C

Flash Tank Model Equations

C.1 Stationary Flash Tank

The phase equilibrium equations

$$\log_{10}(p_{\text{sat},i}) = A_i - \frac{B_i}{T + C_i} \tag{C.1}$$

$$K_i = \frac{p_{\text{sat},i}}{p} \tag{C.2}$$

$$\beta \cdot K_i \cdot x_i = y_i \tag{C.3}$$

$$\sum_{i=1}^{nc} y_i - \sum_{i=1}^{nc} x_i = 0 \tag{C.4}$$

$$\beta - 1 - s_V + s_L = 0 \tag{C.5}$$

The material balances

$$F \cdot z_i = F_V \cdot y_i + F_L \cdot x_i \tag{C.6}$$

$$M_i = M_L \cdot x_i + M_V \cdot y_i \tag{C.7}$$

$$\sum_{i=1}^{nc} M_i = M_L + M_V \tag{C.8}$$

The energy balances

$$F \cdot h_i = F_L \cdot h_L + F_V \cdot h_V \tag{C.9}$$

$$h_L = \sum_i^{nc} x_i \cdot c_{p,L,i} \cdot (T - T_{ref}), \tag{C.10}$$

$$h_V = \sum_i^{nc} y_i \ \left[h_{vap,i} + c_{p,V,i} \cdot (T - T_{ref})\right], \tag{C.11}$$

$$H = M_L \cdot h_L + M_V \cdot h_V \tag{C.12}$$

$$H = U + p \cdot V \tag{C.13}$$

Phase distribution

$$V_V + V_L = V \tag{C.14}$$

$$V_V \cdot p = M_V \cdot R \cdot T \tag{C.15}$$

$$\frac{1}{\rho_L} = \sum_{i=1}^{nc} \frac{x_i}{\rho_i} \tag{C.16}$$

The valve equations

$$z = s_{avm} + s_{avp} \tag{C.17}$$

$$p - p^0 = s_{avm} - s_{avp} \tag{C.18}$$

$$0 \leq s_{avm} \perp s_{avp} \geq 0 \tag{C.19}$$

$$w = 0 + s_{wm} \tag{C.20}$$

$$p^0 - p = (z + \epsilon)^{0.5}(s_{wp} - s_{wm}) \tag{C.21}$$

$$0 \leq s_{wm} \perp s_{wp} \geq 0 \tag{C.22}$$

$$F_V = c_V \cdot \frac{V_V}{V} \cdot w, \tag{C.23}$$

$$F_L = c_L \cdot \frac{V_L}{V} \cdot w. \tag{C.24}$$

C.2 Dynamic Flash Tank

The phase equilibrium equations

$$\log_{10}(p_{\text{sat},i}(t)) = A_i - \frac{B_i}{T(t) + C_i} \tag{C.25}$$

$$k_i(t) = \frac{p_{\text{sat},i}(t)}{p(t)} \tag{C.26}$$

$$\beta(t) \cdot k_i(t) \cdot x_i(t) = y_i(t) \tag{C.27}$$

$$\sum_{i=1}^{nc} y_i(t) - \sum_{i=1}^{nc} x_i(t) = 0 \tag{C.28}$$

$$\beta(t) - 1 - s_V(t) + s_L(t) = 0 \tag{C.29}$$

The material balances

$$\frac{dM_i(t)}{dt} = F_{in}(t) \cdot z_i - F_L(t) \cdot x_i(t) - F_V(t) \cdot y_i(t) + Q(t) \tag{C.30}$$

$$M_i(t) = M_L(t) \cdot x_i(t) + M_V(t) \cdot y_i(t) \tag{C.31}$$

$$\sum_{i=1}^{nc} M_i(t) = M_L(t) + M_V(t) \tag{C.32}$$

The energy balances

$$\frac{dU_i(t)}{dt} = F_{in} \cdot h_{in} - F_L(t) \cdot h_L(t) - F_V(t) \cdot h_V(t) \tag{C.33}$$

$$h_L(t) = \sum_i^{nc} x_i(t) \cdot c_{p,L,i} \cdot (T(t) - T_{ref}), \tag{C.34}$$

$$h_V(t) = \sum_i^{nc} y_i(t) \ \left[h_{vap,i} + c_{p,V,i} \cdot (T(t) - T_{ref})\right], \tag{C.35}$$

$$H(t) = M_L(t) \cdot h_L(t) + M_V(t) \cdot h_V(t) \tag{C.36}$$

$$H(t) = U(t) + p(t) \cdot V \tag{C.37}$$

Phase distribution

$$V_V(t) + V_L(t) = V \tag{C.38}$$

$$V_V(t) \cdot p(t) = M_V(t) \cdot R \cdot T(t) \tag{C.39}$$

$$\frac{1}{\rho_L(t)} = \sum_{i=1}^{nc} \frac{x_i(t)}{\rho_i} \tag{C.40}$$

The valve equations

$$z(t) = s_{avm}(t) + s_{avp}(t) \tag{C.41}$$

$$p(t) - p^0 = s_{avm}(t) - s_{avp}(t) \tag{C.42}$$

$$0 \leq s_{avm}(t) \perp s_{avp}(t) \geq 0 \tag{C.43}$$

$$w(t) = 0 + s_{wm}(t) \tag{C.44}$$

$$p^0 - p(t) = (z(t) + \epsilon)^{0.5}(s_{wp}(t) - s_{wm}(t)) \tag{C.45}$$

$$0 \leq s_{wm}(t) \perp s_{wp}(t) \geq 0 \tag{C.46}$$

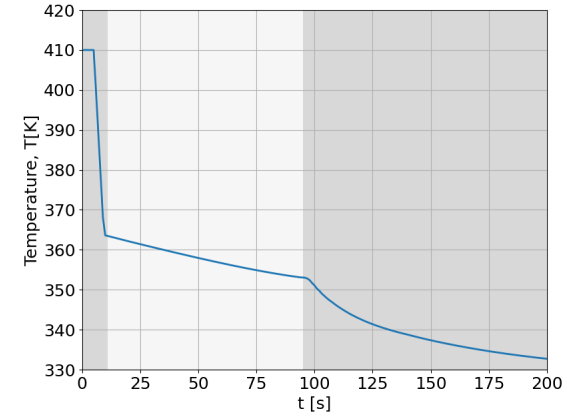
$$F_V(t) = c_V \cdot \frac{V_V(t)}{V} \cdot w(t), \tag{C.47}$$

$$F_L(t) = c_L \cdot \frac{V_L(t)}{V} \cdot w(t). \tag{C.48}$$

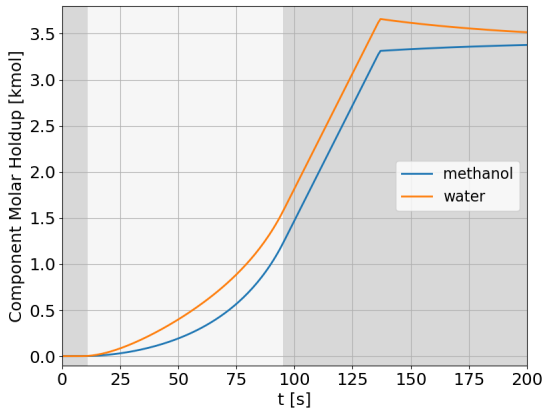
Appendix D

Flash Tank Simulation Failure

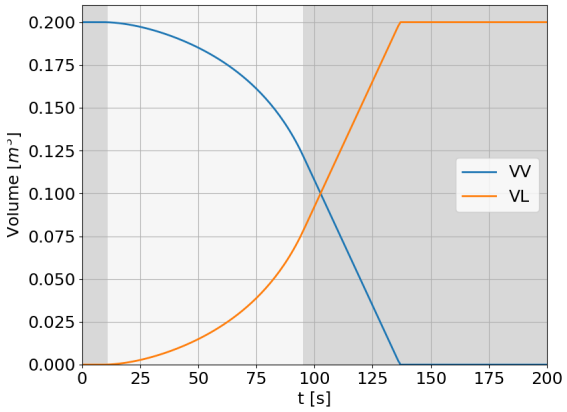
The following figures show the results from a failed simulation of a dynamic flash tank where the solver received one final element at a time, and the time horizon was 200 s.



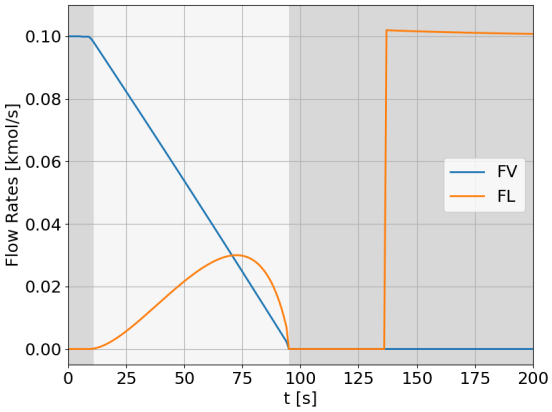
(a) The temperature [K].



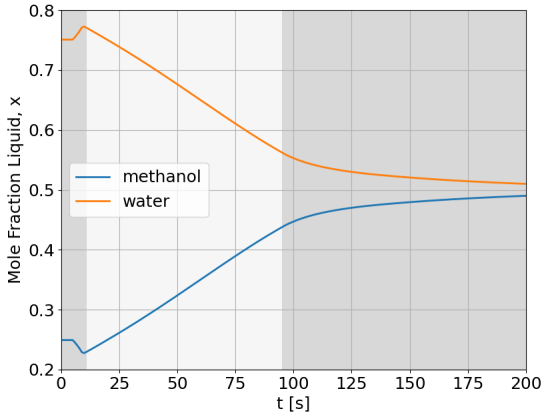
(b) Component molar holdup [kmol]



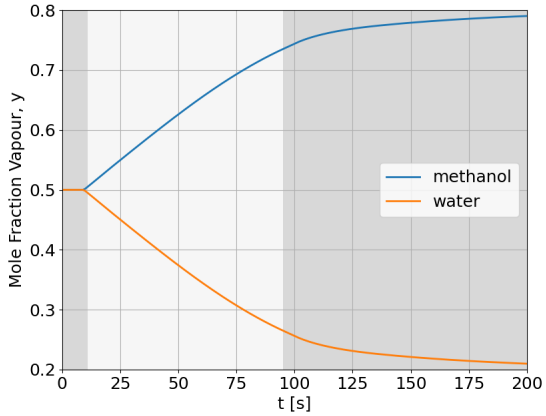
(c) The volume [m^3]



(d) The molar flows [$\mathrm{kmol\,s^{-1}}$]

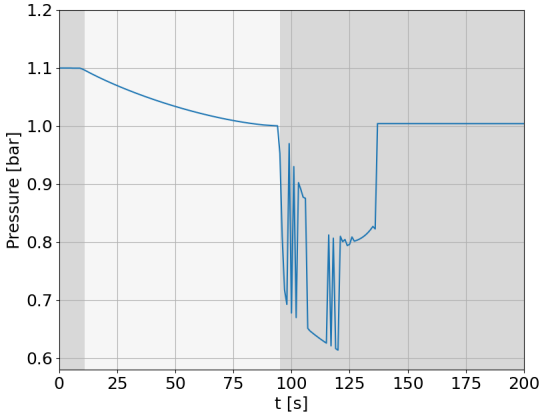


(e) The liquid mole fraction, x

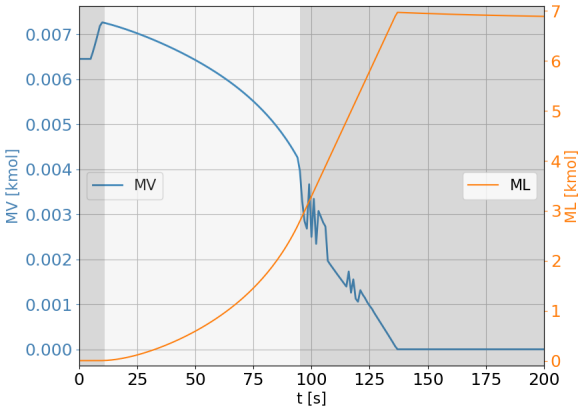


(f) The vapour mole fraction,y.

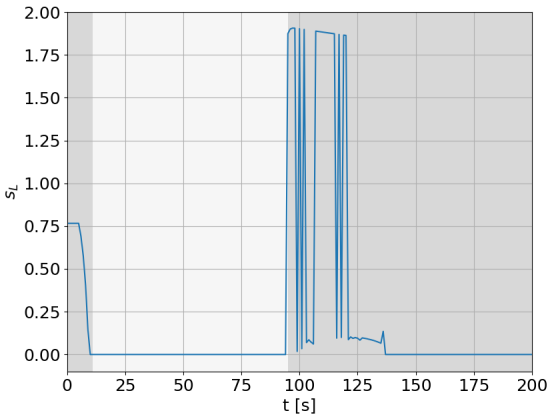
Figure D.1: The result of a failed simulation. The values of temperature, volume, component holdup, the molar flows, and the mole fractions with respect to time. The dark grey areas show the single-phase regions



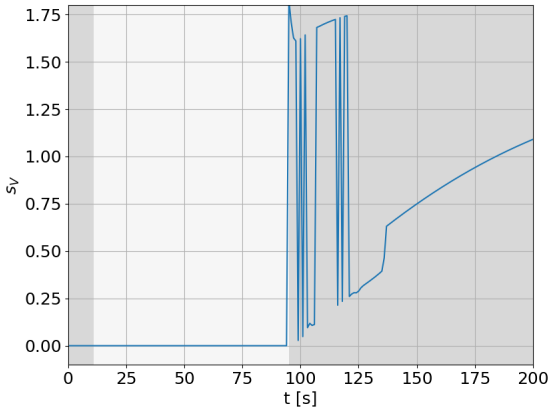
(a) Pressure [bar].



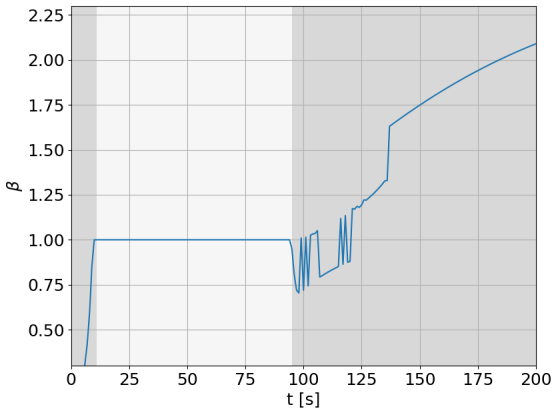
(b) Liquid and vapour molar holdup [kmol]



(c) The slack variable for liquid phase s_L

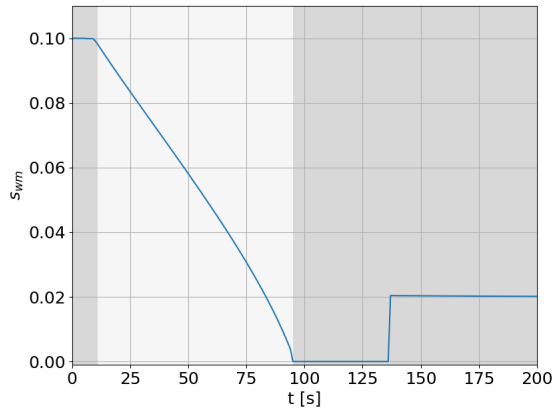


(d) The slack variable for vapour phase s_V

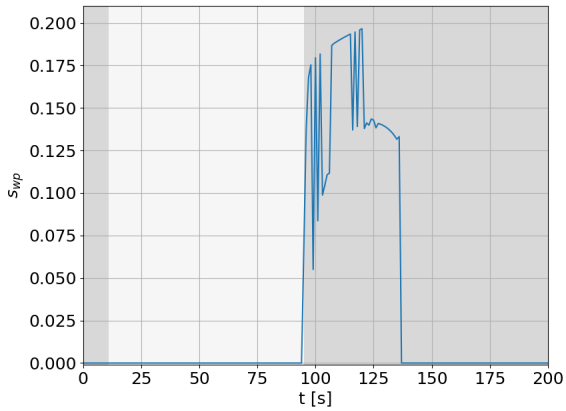


(e) The relaxation parameter, β.

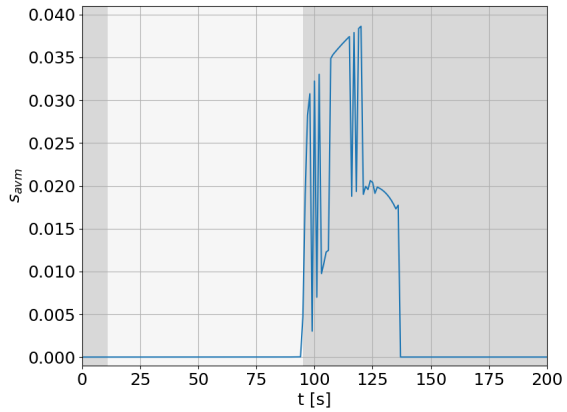
Figure D.2: The result of a failed simulation. The values of the slack variables s_L and s_V, the relaxation parameter β, the pressure and the molar holdups. The single-phase regions are represented in dark grey.



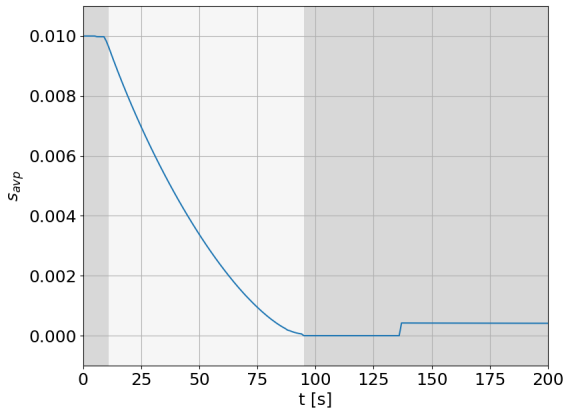
(a) The complementarity value for max operator, s_{wm}



(b) The complementarity value for max operator, s_{wp}



(c) The complementarity value for absolute value, s_{avm}



(d) The complementarity value for absolute value, s_{avp}

Figure D.3: The result of a failed simulation. The values of the complementarity variables for the max operator and the absolute expression in the valve equations with respect to time. The dark grey area indicates the two-phase region.

Appendix E

The Maxwell Relations

Entropy, s is a thermodynamic property which can not be measured directly. For this property to be computed, it must be related to other measurable properties. The Maxwell relations are equations which relate partial derivatives of the properties pressure P volume V, temperature T and entropy with each other. It is important to note that these relations only apply to simple compressible systems. The Maxwell relations are derived from the Gibbs equations,

$$du = Tds - PdV \tag{E.1a}$$

$$dh = Tds + VdP \tag{E.1b}$$

$$da = -sdT + VdP \tag{E.1c}$$

$$dg = -sdT + VdP, \tag{E.1d}$$

where u is internal energy and a and g are the Helmholtz and Gibbs energy, respectively. By examining these relations, they appear to be on the form,

$$dz = Mdx + Ndy, \tag{E.2}$$

with,

$$\left(\frac{\partial M}{\partial y}\right)_x = \left(\frac{\partial N}{\partial x}\right)_y. \tag{E.3}$$

Equation (E.3) can be applied to all of the properties u,h,a and g as they have exact differentials. The Maxwell relations are then derived [22],

$$\left(\frac{\partial T}{\partial V}\right)_s = -\left(\frac{\partial P}{\partial s}\right)_V \tag{E.4a}$$

$$\left(\frac{\partial T}{\partial P}\right)_s = \left(\frac{\partial V}{\partial s}\right)_p \tag{E.4b}$$

$$\left(\frac{\partial s}{\partial V}\right)_T = \left(\frac{\partial P}{\partial T}\right)_V \tag{E.4c}$$

$$\left(\frac{\partial s}{\partial P}\right)_T = -\left(\frac{\partial v}{\partial T}\right)_p \tag{E.4d}$$

Appendix F

The Approaches Implemented in Julia

The following codes are the implementations of the Classic and Dynamic approaches using IPOPT as the implementation basis.

F.1 The Classic Implementation

```
#
# Implementation of the Classic algorithm.
#
# Based on the paper:
# "Interior Methods for Mathematical Programs with
# Complementarity Constraints" by Sven Leyffer
#
# Ann Iren Fossøy
# Spring 2023
#
# Last update: 04.06.23
#

# Custom union types
ProdL = Union{VariableRef, AffExpr}
ProdNL = Union{QuadExpr, NonlinearExpression,ProdL}

TypeL = Union{QuadExpr, VariableRef, AffExpr}
TypeNL = Union{NonlinearExpression,TypeL}

"""
    classic_solve_mpcc!(model, comp, objective = :obj; kwargs...)

Solves an MPCC problem defined in JuMP using the Classic interior-penalty approach
with IPOPT as basis.

`model` is a JuMP model containing the MPCC problem;
`comp` is an array of pairs with the complementarity variables;
`objective` is a symbol for the JuMP expression definng the objective function;
```

```
`kwargs` are parameters used by the MPCC algorithm;
"""

function classic_solve_mpcc!(model, comp, objective = :obj ; kwargs...)
    # Defining parameters
    γ =    get(kwargs, :gamma, 0.4);
    π =    get(kwargs, :pi, 1);           # the penalty parameter starting value
    μ =    get(kwargs, :mu, 0.1);         # the barrier parameter starting value
    κϵ =   get(kwargs, :κϵ, 10);
    ϵtol = get(kwargs, :ϵtol, 1e-8);
    κμ =   get(kwargs, :κμ, 0.2)
    θμ =   get(kwargs, :θμ, 1.5)

    nc = length(comp);
    cvar = Vector{Tuple{Any,Any}}(undef, nc);   # the vector with the comp. pairs
    lenComp = Array{Int64, 1}(undef, nc)        # the length of each comp. pairs
    pen = Array{Any, 1}(undef, nc)              # the vector with penalty terms
    obj = getindex(model, objective)

    # Intalization of iteration
    k = 1;
    j = 0;

    lastObjVal = 0;
    objVal = 0;
    itCounter = 0;

    for i in 1:nc
        add_comp_pair!(cvar, comp[i][1],comp[i][2],i)
        add_length!(lenComp,cvar[i][1],i)

        if lenComp[i] == 1
            add_pen!(pen, model, i, [cvar[i][1]; cvar[i][2]]...)
        else
            add_pen!(pen, model, i,lenComp[i], [cvar[i][1]; cvar[i][2]]...)
        end
    end


    # Adding the penalty terms to the objective function
    update_objective!(model,obj,π,nc,pen...);

    # Number of variables
    n = num_variables(model);
    m = num_constraints(model,count_variable_in_set_constraints = false);

    while k <= 100

        ϵcomp = μ^γ
        ϵpen =  μ * κϵ
        set_optimizer_attributes(model, "mu_target" => μ, "mu_init" => μ,
                                        "dual_inf_tol" =>  ϵpen,
                                        "constr_viol_tol" => ϵpen,
                                        "compl_inf_tol" => ϵpen,
                                        "warm_start_init_point" => "yes"
                                       );
```

```
iters = []              # number of iterations
dual_inf = []           # the primal infeasbility
primal_inf = []         # the dual infeasbility
compl_inf = []          # the complementarity infeasbility
val_obj = []            # the objective function value
val_var = []            # the values of the variables
x_it = []
y_it = []

function my_callback(
    alg_mod::Cint,
    iter_count::Cint,
    obj_value::Float64,
    inf_pr::Float64,
    inf_du::Float64,
    mu::Float64,
    d_norm::Float64,
    regularization_size::Float64,
    alpha_du::Float64,
    alpha_pr::Float64,
    ls_trials::Cint)

    append!(iters, iter_count)
    append!(dual_inf,inf_du)
    append!(primal_inf, inf_pr)
    append!(val_obj, obj_value)

    # To acesses the complementarity infeasibility in each iteraion
    x, z_L, z_U = zeros(n), zeros(n), zeros(n)
    g, lambda = zeros(m), zeros(m)
    scaled = true
    prob = unsafe_backend(model).inner
    Ipopt.GetIpoptCurrentIterate(prob,scaled,n,x,z_L,z_U,m,g,lambda)
    x_L_violation, x_U_violation = zeros(n), zeros(n)
    compl_x_L, compl_x_U, grad_lag_x = zeros(n), zeros(n), zeros(n)
    nlp_constraint_violation, compl_g = zeros(m), zeros(m)
    Ipopt.GetIpoptCurrentViolations(
        prob,
        scaled,
        n,
        x_L_violation,
        x_U_violation,
        compl_x_L,
        compl_x_U,
        grad_lag_x,
        m,
        nlp_constraint_violation,
        compl_g,
    )
    inf_compl = maximum([compl_x_L;compl_x_U])
    append!(compl_inf,inf_compl)
    append!(val_var,[x])
    append!(x_it,x[1])
    append!(y_it,x[2])

    return true
end
```

```
        MOI.set(model, Ipopt.CallbackFunction(), my_callback)

        # Optimizing the model
        optimize!(model);

        # Get number of inner iterations
        j += last(iters)

        # Get the complementarity satisfaction
        comp_satisfaction = complementaritycheck(cvar,nc)

        if comp_satisfaction > ϵcomp  # if complementarity not satisfied
            if π < 1e14
                π = π * 10

                update_objective!(model,obj,π,nc,pen...)

            else
                 error("Couldn't find a suitable value for pi")
            end

        else                          # if complementarity satisfied satisfied
            print_iteration(model,k,objVal,μ,π)

            lastObjVal =  objVal
            objVal = objective_value(model)

            # Checking stopping test for MPCC
            if (μ <= ϵtol && optimality(primal_inf,dual_inf,compl_inf) <= ϵtol
                && abs(objVal - lastObjVal) <= ϵtol)

                print_result(model,k,objVal,j)
                return 1
                break

            else
                k += 1;
                μ = decrease_μ(μ, ϵtol, κμ, θμ)

                i = 1
                while i < 5

                    if last(compl_inf) < κϵ*μ
                        μ = decrease_μ(μ, ϵtol, κμ, θμ)
                    else
                        break
                    end
                    i += 1
                end
                set_optimal_start_values(model)
            end
        end
        if k > 100
            println("Maximum number of iterations reached.")
        end
    end
    return 0
```

```
end


"""
    add_comp_pair!(cvar,comp_1,comp_2,i)

Adds the complementarity pair into a vector cvar

`comp_1` is the first expression in the complementarity pair;
`comp_2` is the second expression in the complementarity pair;
`cvar` is the vector containing the complementairty pairs;
`i` is the number of the current complementarity pairs;

"""

function add_comp_pair!(cvar, comp_1::Matrix{T},
    comp_2::Matrix{T}, i)where T <: TypeNL

    cvar[i] = (collect(Iterators.flatten(comp_1)),
    collect(Iterators.flatten(comp_2)));
end

function add_comp_pair!(cvar,comp_1::T,
    comp_2::T ,i) where T<:JuMP.Containers.DenseAxisArray

    cvar[i] = (collect(Iterators.flatten(comp_1)),
    collect(Iterators.flatten(comp_2)));
end

function add_comp_pair!(cvar, comp_1::Union{VariableRef, AffExpr, QuadExpr},
    comp_2::NonlinearExpression,i)

    cvar[i] = (collect(Iterators.flatten(comp_1)),
    collect(Iterators.flatten(comp_2)));
end

function add_comp_pair!(cvar, comp_1::NonlinearExpression,
    comp_2::Union{VariableRef, AffExpr, QuadExpr},i)

    cvar[i] = (collect(Iterators.flatten(comp_1)),
    collect(Iterators.flatten(comp_2)));
end

function add_comp_pair!(cvar, comp_1::NonlinearExpression,
     comp_2::NonlinearExpression, i)

    cvar[i] = (collect(Iterators.flatten(comp_1)),
    collect(Iterators.flatten(comp_2)));
end

function add_comp_pair!(cvar, comp_1, comp_2, i)

    cvar[i] = (comp_1, comp_2)
end


"""
```

```
add_length!(compi, lenComp, i)

Find the length of the current complementairty pair and add to vector.

`compi` is one of the complementarity expressions in the pair;
`lenComp` is the vector storing the lengths;
`i` is the number of the current complementarity pair;
"""


function add_length!(lenComp,compi::VariableRef,i)

    lenComp[i] = 1;
end

function add_length!(lenComp,compi,i)

    lenComp[i] = length(compi)
end



"""
    add_pen!(pen, model, i, cvar...)

Multuiply the complementairty pairs, creating the penalty terms, and adding
it to the vector 'pen'

`pen` is the vector containing the penalty expressions;
`model` is the JuMP model;
`i` is the number of the current complementarity pair;
`cvar` containts the complementarity pair;
"""

function add_pen!(pen, model::JuMP.Model,i, cvar::ProdNL...)

    pen[i] = @NLexpression(model, cvar[1] * cvar[2]);
end


function add_pen!(pen, model::JuMP.Model,i, cvar::ProdL...)

    pen[i] = @expression(model, cvar[1] * cvar[2]);
end


function add_pen!(pen, model::JuMP.Model, i, len, cvar::ProdL...)
    cvar_1 = cvar[1:len]
    cvar_2 = cvar[len+1:end]
    pen[i] = @expression(model,
    sum(cvar_1[j] * cvar_2[j] for j in eachindex(cvar_1)));
end

function add_pen!(pen, model::JuMP.Model, i, len, cvar::ProdNL...)
    cvar_1 = cvar[1:len]
    cvar_2 = cvar[len+1:end]
    pen[i] = @NLexpression(model,
```

```
    sum(cvar_1[j] * cvar_2[j] for j in eachindex(cvar_1)));
end

"""
    update_objective!(model, obj, π, nc, pen...)

Multiplying the complementairty pairs, creating the penalty terms,
and adding it to the vector 'pen'

`model` is the JuMP model;
`obj` is the objective function;
`π` is the current value for π;
`nc` is the length of the cvar vector;
`pen` is the vector containing the penalty expressions;

"""


function update_objective!(model::JuMP.Model, obj::NonlinearExpression,
    π, nc, pen...)

    @NLobjective(model, Min, obj + π * sum(pen[i] for i in 1:nc));
end

function update_objective!(model::JuMP.Model,obj,π, nc, pen::TypeNL...)

    @NLobjective(model, Min, obj + π * sum(pen[i] for i in 1:nc));
end

function update_objective!(model::JuMP.Model,obj::NonlinearExpression,
    π, nc, pen::TypeNL...)

    @NLobjective(model, Min, obj + π * sum(pen[i] for i in 1:nc));
end

function update_objective!(model::JuMP.Model,obj,π,nc,pen::TypeL...)

    @objective(model, Min, obj + π * sum(pen[i] for i in 1:nc));
end



"""
    complementarycheck(cvar,nc)

Cheaking the complementarity satisfation by taking the infity norm of
the smallest element in each complementarity pair

`cvar` the vector containg the complementarity pairs;
`nc` is the length of the cvar vector;

"""

function complementarycheck(cvar,nc)
    compVal = Vector{Tuple{Any,Any}}(undef,nc);
    minVal = [];
    for i in 1:nc
```

```
        compVal[i] = (value.(cvar[i][1]), value.(cvar[i][2]))
        append!(minVal,min.(compVal[i][1], compVal[i][2]))
    end
    norm(minVal, Inf)
end



"""
    print_iteration(model, k, ovjVal, μ, π)

Print each iteration

`model` is the JuMP model;
`k` is the current outer iteration;
`objVal` is the objective function value;
`μ` is the current barrier parameter;
`π` is the current penalty parameter;

"""

function print_iteration(model,k,objVal,μ, π)
    @printf("it: %2i, obj = %.3f, Termination = %11s,log(mu) = %3.1f,
    pi = %1i \n", k, objVal, termination_status(model) ,log10(μ),π)
end

"""
    print_result(model, k, objVal,j)

Print the result.

`model` is the JuMP model;
`k` is the numer of outer iteration;
`objVal` is the objective function value found;
`j`  is the total number inner iterations;

"""

function print_result(model,k,objVal,j)
    println("\n Outer iterations = ", k,
            "\n Objective value = ", objVal,
            "\n Inner iterations = ", j);
end


"""
    set_optimal_start_values(model)
Set primal and dudal warm start from JuMP
Taken from: https://jump.dev/JuMP.jl/stable/tutorials/conic/start_values/

`model` is the JuMP model
"""

function set_optimal_start_values(model::Model)
    # Store a mapping of the variable primal solution
    variable_primal = Dict(x => value(x) for x in all_variables(model))
    # In the following, we loop through every constraint and store a mapping
```

```
    # from the constraint index to a tuple containing the primal and dual
    # solutions.
    constraint_solution = Dict()
    nlp_dual_start = nonlinear_dual_start_value(model)
    for (F, S) in list_of_constraint_types(model)
        # We add a try-catch here because some constraint types might not
        # support getting the primal or dual solution.
        try
            for ci in all_constraints(model, F, S)
                constraint_solution[ci] = (value(ci), dual(ci))
            end
        catch
            @info("Something went wrong getting $F-in-$S. Skipping")
        end
    end
    # Now we can loop through our cached solutions and set the starting values.
    for (x, primal_start) in variable_primal
        set_start_value(x, primal_start)
    end
    for (ci, (primal_start, dual_start)) in constraint_solution
        # set_start_value(ci, primal_start)
        set_dual_start_value(ci, dual_start)
    end
    set_nonlinear_dual_start_value(model, nlp_dual_start)
    return
end

"""
    optimality(primal_inf,dual_inf, compl_inf)
Returns the largest value of the primal, dual and complementarity infeasbility

`primal_inf` vector containing the primal infeasbility
`primal_dual` vector containing the dual infeasbility.
`compl_inf` vector containing the complementarity infeasbility

"""

function optimality(primal_inf,dual_inf,compl_inf)
    max(last(primal_inf),last(dual_inf),last(compl_inf))
end


"""
    decrease_μ(μ0,ϵtol,κμ,θμ)
Decreasing μ accroding to a spesific rule.
This way, the barrier parameter does not become smaller than necessary
given the overall tolerance ϵtol of the problem.

`μ0` the previous μ
`ϵtol` the overall tolerance
`κμ` spesific parameter satisfying 0 < κμ < 1
`θμ` spesific parameter satisfying 1 < θμ < 2

"""

function decrease_μ(μ0,ϵtol,κμ,θμ)
    μ  = max(ϵtol*0.1, min(κμ*μ0, μ0^θμ))
```

```
end
```

F.2 The Dynamic Implementation

```
#
# Implementation of the Dynamic algorithm.
#
# Based on the paper: "Interior Methods for Mathematical Programs
# with Complementarity Constraints" by Sven Leyffer
#
# Ann Iren Fossøy
# Spring 2023
#
# Last update: 04.06.23
#


"""
    dynamic_solve_mpcc!(model, comp, objective = :obj; kwargs...)

Solves an MPCC problem defined in JuMP using the Dynamic interior-penalty approach
with IPOPT as basis.

`model` is a JuMP model  containing the MPCC problem;
`comp` is an array of pairs with the complementarity variables;
`objective` is a symbol for the JuMP expression definng the objective function;
`kwargs` are parameters used by the MPCC algorithm;
"""

function dynamic_solve_mpcc!(model, comp, objective = :obj ; kwargs...)
    # defining parameters
    γ =    get(kwargs, :gamma, 0.4);
    π =    get(kwargs, :pi, 1);          # the penalty parameter starting value
    μ =    get(kwargs, :mu, 0.1);        # the barrier parameter starting value
    κϵ =   get(kwargs, :κϵ, 10);
    ϵtol = get(kwargs, :ϵtol, 1e-8);
    κμ =   get(kwargs, :κμ, 0.2)
    θμ =   get(kwargs, :θμ, 1.5)
    η =    get(kwargs, :η, 0.9)

    nc = length(comp);
    cvar = Vector{Tuple{Any,Any}}(undef, nc); # the vector with the comp. pairs
    lenComp = Array{Int64, 1}(undef, nc)      # the length of each comp. pair
    pen = Array{Any, 1}(undef, nc)            # the vector with the penalty terms
    obj = getindex(model, objective)

    # Intalization of iteration
    k = 1;
    j = 0;

    m_val = 3
```

```
val = ones(m_val)    # values attained in the m last iterations

lastObjVal = 0;
objVal = 0;

for i in 1:nc
    add_comp_pair!(cvar, comp[i][1],comp[i][2],i)
    add_length!(lenComp,cvar[i][1],i)

    if lenComp[i] == 1
        add_pen!(pen, model, i, [cvar[i][1]; cvar[i][2]]...)
    else
        add_pen!(pen, model, i, lenComp[i], [cvar[i][1]; cvar[i][2]]...)
    end

end

# Adding the penalty terms to the objective function
update_objective!(model,obj,π,nc,pen...);

# Number of variables
n = num_variables(model);
m = num_constraints(model,count_variable_in_set_constraints = false);

p = 0;                      # the current indice
itCounter = 0;              # total number of inner iteraitons
while k <= 100

    ϵcomp = μ^γ
    ϵpen = μ * κϵ


    set_optimizer_attributes(model, "mu_target" => μ, "mu_init" => μ,
                                    "dual_inf_tol" =>  ϵpen,
                                    "constr_viol_tol" => ϵpen,
                                    "compl_inf_tol" => ϵpen)



    iters = []              # Number of iterations
    dual_inf = []           # The primal infeasbility
    primal_inf = []         # The dual infeasbility
    compl_inf = []          # The complementarity infeasbility
    val_obj = []            # The objective function value
    val_var = []            # The values of the variables
    x_it = []
    y_it = []

    compVal = []
    piVal = []


    current_val_val = [0.]
    η_max_val_val = [0.]
    condition = true;
```

```
        function my_callback(
            alg_mod::Cint,
            iter_count::Cint,
            obj_value::Float64,
            inf_pr::Float64,
            inf_du::Float64,
            mu::Float64,
            d_norm::Float64,
            regularization_size::Float64,
            alpha_du::Float64,
            alpha_pr::Float64,
            ls_trials::Cint)


            append!(iters, iter_count)
            append!(dual_inf,inf_du)
            append!(primal_inf, inf_pr)
            append!(val_obj, obj_value)

            # To acesses the complementarity infeasibility in each iteraion
            x, z_L, z_U = zeros(n), zeros(n), zeros(n)
            g, lambda = zeros(m), zeros(m)
            scaled = true
            prob = unsafe_backend(model).inner
            Ipopt.GetIpoptCurrentIterate(prob,scaled,n,x,z_L,z_U,m,g,lambda)
            x_L_violation, x_U_violation = zeros(n), zeros(n)
            compl_x_L, compl_x_U, grad_lag_x = zeros(n), zeros(n), zeros(n)
            nlp_constraint_violation, compl_g = zeros(m), zeros(m)
            Ipopt.GetIpoptCurrentViolations(
                prob,
                scaled,
                n,
                x_L_violation,
                x_U_violation,
                compl_x_L,
                compl_x_U,
                grad_lag_x,
                m,
                nlp_constraint_violation,
                compl_g,
            )
            inf_compl = maximum([compl_x_L; compl_x_U])
            append!(compl_inf,inf_compl)
            append!(val_var,[x])
            append!(x_it,x[1])
            append!(y_it,x[2])


            if iter_count !=0

                append!(compVal,complCheck(model,cvar,nc))
                append!(piVal, π)

                p = (itCounter % m_val != 0) ? p + 1 : 1
                update_val!(val,model,cvar,nc,p,lenComp)

                if iter_count >= 3
```

```
                max_val =  maximum(val)
                current_val = val[p]
                compSat = complCheck(model,cvar,nc)
                condition = !(compSat > ϵcomp && current_val > η*max_val)

                # For debugging
                current_val_val[1] = current_val
                η_max_val_val[1] = η*max_val
            else
                condition = true
            end

            itCounter += 1
        else
            condition = true
        end

        return condition
    end
    MOI.set(model, Ipopt.CallbackFunction(), my_callback)

    # Optimizing the model
    optimize!(model);

    # Get number of inner iterations
    j += last(iters)

    # If current_val > η*max_val
    if MOI.get(model, MOI.TerminationStatus()) == MOI.INTERRUPTED
        if π < 1e8
            π = π * 10
            update_objective!(model,obj,π,nc,pen...)

        else
            error("Couldn't find a suitable value for pi")
        end

    else
        # Get the complementarity satisfaction
        comp_satisfaction = complementaritycheck(cvar,nc)

        if comp_satisfaction >= ϵcomp   # if complementarity not satisfied
            if π < 1e8
                π = π * 10
                update_objective!(model,obj,π,nc,pen...)


            else
                error("Couldn't find a suitable value for pi")
            end

        else                            # if complementarity satisfied
            print_iteration(model,k,objVal,μ,π)

            lastObjVal =  objVal
            objVal = objective_value(model)
```

```
                # Checking stopping test for MPCC
                if (μ <= ϵtol && optimality(primal_inf,dual_inf,compl_inf) <= ϵtol
                    && abs(objVal - lastObjVal) < ϵtol)

                    print_result(model,k,objVal,j)
                    return 1
                    break

                else
                    k += 1;
                    μ = decrease_μ(μ, ϵtol, κμ, θμ)

                    i = 1
                    while i < 5

                        if last(compl_inf) < κϵ*μ
                            μ = decrease_μ(μ, ϵtol, κμ, θμ)
                        else
                            break
                        end
                        i += 1
                    end
                    set_optimal_start_values(model)
                    set_optimizer_attributes(model,
                    "warm_start_init_point" => "yes")
                end
            end
        end
        if k > 100
            println("Maximum number of iterations reached.")
        end
    end
    return  0
end


"""
    update_val!(val,model,cvar,,nc,p,lencomp)
Updating the val vectors after each inner iterations.
The val vector containts the product of the complementarity pairs in the
m last iterations

`val` the vector beining updated
`model` the JuMP model
`cvar` the vector holding the comp. pairs
`p` the current indice to be updated
`lenComp` the vector containg the length of the comp. pairs

"""

function update_val!(val,model,cvar,nc,p,lenComp)
    compVal = Vector{Tuple{Any,Any}}(undef,nc);
    compProd = Array{Float64, 1}(undef, nc)

    function func(xi)
        return callback_value(model,xi)
    end
```

```
    for i in 1:nc
        compVal[i] = (value.(func,cvar[i][1]), value.(func,cvar[i][2]))
        compProd[i] = (lenComp[i] == 1) ? compVal[i][1] * compVal[i][2]
        : sum(compVal[i][1][j] * compVal[i][2][j] for j in lenComp[i])
    end
    val[p] = sum(compProd[i] for i in nc)
end

"""
   complCheck(model,cvar,nc)
Check the complementarity satisfaction after each inner iterations

`model` the JuMP model
`cvar` the vector holding the comp. pairs

returns the infity norm
"""


function complCheck(model,cvar,nc)
    compVal = Vector{Tuple{Any,Any}}(undef,nc);
    minVal = [];

    function funci(xi)
        return callback_value(model,xi)
    end

    for i in 1:nc
        compVal[i] = (value.(funci,cvar[i][1]), value.(funci,cvar[i][2]))
        append!(minVal,min.(compVal[i][1], compVal[i][2]))
    end
    norm(minVal, Inf)
end
```

Appendix G

The Flash Thank implementation

G.1 The Flash Tank Functions

```
#
# Functions for composing the flash model
#

using MPCCLibrary;

R = 8.314                               # MJ/(kmol · Kk)

struct Dynamic
    nfe::Int16                          # number of final elements
    ncp::Int8                           # number of collocation points
    tspan::Tuple{Float64, Float64}      # the time horizon
end

struct Substance
    name::String
    rho::Float64                        # density                   Mmol/m^3
    CpL::Float64                        # Heat capacity liquid      MJ/(kmol · Kk)
    CpV::Float64                        # Vapor capacity liquid     MJ/(kmol · Kk)
    Hvap::Float64                       # Heat of vaporization      MJ/(kmol)
    Antoine_par::Array{Float64,1}       # [A, B, C]                 bar & K
    Tref::Float64                       # Referance temperature     Kk
end

mutable struct Feed
    Comp::Array{Substance,1}
    z::Array{Float64,1}                 # Mole fraction
    F::Float64                          # Feed stream               Kmol/s
    T::Float64                          # Temperature of feed       Kk
    p::Float64                          # Preesure of feed          MPa
    Feed() = new()
end
```

```
# -------------------------- Singel flash tank ------------------------------

"""
    flash_ss!(model, Fin, kwargs...)
A representation of a steady state model of flash tank;
Adds constraints to model based on the inlet feed.
Returns the complementarity pairs as a vector

`model` is a JuMP model containing the MPCC problem;
`Fin` is the feed given as a struct;
`kwargs` are custom choice for parameters;
"""


function flash_ss!(model::Model, Fin::Feed ;kwargs...)

    # parameters
    Q =  get(kwargs, :Q,  0.);              # MW
    V =  get(kwargs, :V,  0.2);             # m^3
    p0 = get(kwargs, :p0, 0.1);             # MPa
    ϵ =  get(kwargs, :ϵ,  1e-10);
    cL = get(kwargs, :cL, 5);               # kmol/(s·√(MPa))
    cV = get(kwargs, :cV, 1);               # kmol/(s·√(MPa))
    C0 = 3*145.0377e-6;

    # feed spesifications
    z = Fin.z;
    F = Fin.F;                              # kmol/s
    Tin = Fin.T;                            # Kk
    pin = Fin.p;                            # MPa
    nc = length(z);                         # number of components

    # components spesifications
    A = Array{Float64,1}(undef, nc);
    B = Array{Float64,1}(undef, nc);
    C = Array{Float64,1}(undef, nc);

    rho = Array{Float64,1}(undef, nc);      # Mmol/m^3
    CpL = Array{Float64,1}(undef, nc);      # MJ/(kmol · Kk)
    CpV = Array{Float64,1}(undef, nc);      # MJ/(kmol · Kk)
    h_vap = Array{Float64,1}(undef, nc);    # MJ/(kmol)

    Tref = Array{Float64,1}(undef, nc);     # Kk

    for i in 1:nc
        A[i] = Fin.Comp[i].Antoine_par[1];
        B[i] = Fin.Comp[i].Antoine_par[2];
        C[i] = Fin.Comp[i].Antoine_par[3];
        rho[i] = Fin.Comp[i].rho;
        CpL[i] = Fin.Comp[i].CpL;
        CpV[i] = Fin.Comp[i].CpV;
        h_vap[i] = Fin.Comp[i].Hvap;
        Tref[i] = Fin.Comp[i].Tref;
    end

    hin  = sum(z .* (h_vap .+ CpV .* (Tin .- Tref)))            # MJ/kmol
```

```
n = pin*V/(R*Tin)                                                    # kmol

# model variables
@variable(model, 0 <= x[1:nc] <= 1);
@variable(model, 0 <= y[1:nc] <= 1);
@variable(model, FLr >= 0 , start = 0);                              # kmol
@variable(model, FVr >= 0 , start = 0.2);                            # kmol
@variable(model, K[1:nc] >= 0);
@variable(model, psat[1:nc] >= 0);                                   # MPa
@variable(model, hL, start = 0);                                     # MJ/kmol
@variable(model, hV, start = hin);                                   # MJ/kmol
@variable(model, T >= 0, start = Tin);                               # Kk
@variable(model, p >= 0, start = pin);                               # MPa
@variable(model, VV >= 0, start = 0.2);                              # m^3
@variable(model, VL >= 0, start = 0);                                # m^3
@variable(model, Mc[1:nc] >= 0, start = n);                          # kmol
@variable(model, ML >= 0, start = 0);                                # kmol
@variable(model, MV >= 0, start = n);                                # kmol
@variable(model, U);                                                 # MJ
@variable(model, H);                                                 # MJ
@variable(model, rhoL >= 0);                                         # MJ/Mmol

# auxiliary variables
@variable(model, beta, start = 1);
@variable(model, sv >= 0);       # slack for vapor fraction
@variable(model, sl >= 0);       # slack for liquid fraction
@variable(model, ap >= 0);       # absolute value for p - p0
@variable(model, avp >= 0);      # auxiliary value for absolute expression
@variable(model, avm >= 0);      # auxiliary value for absolute expression
@variable(model, am >= 0);       # maximum value in the valve calculation
@variable(model, wp >= 0);       # auxiliary value for maximum expression
@variable(model, wm >= 0);       # auxiliary value for maximum expression


## constraints
# phase equilibrium
@NLconstraint(model, ant[i in 1:nc],
    psat[i] == 0.1 * 10^(A[i] -  B[i]/(T*1e3 + C[i])))
@constraint(model, eqp[i in 1:nc], K[i] * p == psat[i]);
@NLconstraint(model, eqc[i in 1:nc], beta * K[i] * x[i] == y[i]);
@constraint(model, equil,
    sum(y[i] for i in 1:nc) - sum(x[i] for i in 1:nc) == 0);
@constraint(model, aux, beta - 1 - sv + sl  == 0);

# material balance
@constraint(model, bal_c[i in 1:nc], F*z[i] - FVr*y[i] - FLr*x[i] == 0);
@constraint(model, tot_c[i in 1:nc], Mc[i] == ML*x[i] + MV*y[i]);
@constraint(model, bal_t, sum(Mc[i] for i in 1:nc) == ML + MV);

# energy balance
@constraint(model, bal_e, F*hin - FLr*hL - FVr*hV + Q  == 0);
@constraint(model, ent_l,
    hL == sum(x[i] * CpL[i] * (T - Tref[i]) for i in 1:nc));
@constraint(model, ent_v,
    hV == sum(y[i] * (h_vap[i] + CpV[i] * (T - Tref[i])) for i in 1:nc));
@constraint(model, ent_t, H == ML * hL + MV * hV);
@constraint(model, ent,   H == U + p * V);
```

```
    # volume calculation
    @constraint(model, vol,   VV + VL == V);
    @constraint(model, v_vap, VV * p == MV * R * T);
    @constraint(model, v_liq, VL * rhoL * 1e3  == ML);
    @NLconstraint(model, densL, 1/rhoL == sum(x[i]/rho[i] for i in 1:nc ))

    # valves (outlet)
    @constraint(model, abs1, p - p0 == avp - avm);
    @constraint(model, abs2, ap == avp + avm);
    @NLconstraint(model, max1, p0 - p == (ap^0.5 + ϵ)*(wp - wm));
    @constraint(model, max2, am == wm);
    @constraint(model, valL, FLr  == cL * (VL/V) * am);
    @constraint(model, valV, FVr  == cV * (VV/V) * am);


    ## objective function defined as an expression
    return [(sv, FVr); (sl, FLr); (avp, avm); (wp, wm)];
end

"""
    flash_dynamic!(model, Fin, Dyn, kwargs...)
A representation of a dynamic model of flash tank;
Adds constraints to model based on the inlet feed.
Returns the complementarity pairs as a vector

`model` is a JuMP model containing the MPCC problem;
`Dyn` is a struct containg  nfe, ncp and tspan
`Fin` is the feed given as a struct;
`kwargs` are custom choice for parameters;
"""

function flash_dynamic!(model::Model, Fin::Feed, Dyn::Dynamic,;kwargs...)

    # dynamic parameters
    nfe = Dyn.nfe;
    ncp = Dyn.ncp;
    tspan = Dyn.tspan;
    adot = collocation_matrix(ncp, "Radau");
    dt = (tspan[2]-tspan[1])/nfe;

    # parameters
    Q =  get(kwargs, :Q,  zeros(nfe));
    V =  get(kwargs, :V,  0.2);
    p0 = get(kwargs, :p0, 0.1);
    ϵ =  get(kwargs, :ϵ,  1e-10);
    cL = get(kwargs, :cL, 5);
    cV = get(kwargs, :cV, 1);
    C0 = 3*145.0377e-6;

    # feed spesifications
    z = Fin.z;
    F = ones(nfe)*Fin.F;
    Tin = Fin.T;
    pin = Fin.p;
    nc = length(z);
```

```
    A = Array{Float64,1}(undef, nc);
    B = Array{Float64,1}(undef, nc);
    C = Array{Float64,1}(undef, nc);
    rho = Array{Float64,1}(undef, nc);
    CpL = Array{Float64,1}(undef, nc);
    CpV = Array{Float64,1}(undef, nc);
    h_vap = Array{Float64,1}(undef, nc);
    Tref = Array{Float64,1}(undef, nc);

    for i in 1:nc
        A[i] = Fin.Comp[i].Antoine_par[1];
        B[i] = Fin.Comp[i].Antoine_par[2];
        C[i] = Fin.Comp[i].Antoine_par[3];
        rho[i] = Fin.Comp[i].rho;
        CpL[i] = Fin.Comp[i].CpL;
        CpV[i] = Fin.Comp[i].CpV;
        h_vap[i] = Fin.Comp[i].Hvap;
        Tref[i] = Fin.Comp[i].Tref;
    end

    hin = ones(nfe) * sum(z .* (h_vap .+ CpV .* (Tin .- Tref)))
    n = pin*V/(R*Tin)

    # model variables
    @variable(model, 0 <= x[1:nc,1:nfe,2:ncp+1] <= 1);
    @variable(model, 0 <= y[1:nc,1:nfe,2:ncp+1] <= 1);
    @variable(model, 0 <= FLr[1:nfe,2:ncp+1] <= 0.102, start = 0);
    @variable(model, 0 <= FVr[1:nfe,2:ncp+1] <= 0.102, start = 0);
    @variable(model, K[1:nc,1:nfe,2:ncp+1] >= 0);
    @variable(model, psat[1:nc,1:nfe,2:ncp+1] >= 0);
    @variable(model, hL[1:nfe,2:ncp+1], start = 0);
    @variable(model, hV[1:nfe,2:ncp+1], start = hin[1]);
    @variable(model, 0 <= T[1:nfe,2:ncp+1] <= Tin, start = Tin);
    @variable(model, 0.05 <= p[1:nfe,2:ncp+1] , start = pin);
    @variable(model, 0 <= VV[1:nfe,2:ncp+1] <= V, start = 0.2);
    @variable(model, 0 <= VL[1:nfe,2:ncp+1] <= V, start = 0);
    @variable(model, Mc[1:nc,1:nfe,1:ncp+1] >= 0, start = n);
    @variable(model, ML[1:nfe,2:ncp+1] >= 0, start = 0);
    @variable(model, MV[1:nfe,2:ncp+1] >= 0, start = n);
    @variable(model, U[1:nfe,1:ncp+1]);
    @variable(model, H[1:nfe,2:ncp+1]);
    @variable(model, rhoL[1:nfe,2:ncp+1] >= 0);

    # auxiliary variables
    @variable(model, 0 <= beta[1:nfe,2:ncp+1], start = 1);
    @variable(model, 0 <= sv[1:nfe, 2:ncp+1] <= 2);  # slack for vapor fraction
    @variable(model, 0 <= sl[1:nfe, 2:ncp+1] <= 2);  # slack for liquid fraction
    @variable(model, 0 <= ap[1:nfe, 2:ncp+1] <= 2);  # absolute value for p - p0
    @variable(model, 0 <= avp[1:nfe, 2:ncp+1] <= 2); # auxiliary value for abs.
    @variable(model, 0 <= avm[1:nfe, 2:ncp+1] <= 2); # auxiliary value for abs.
    @variable(model, 0 <= am[1:nfe, 2:ncp+1] <= 2);  # maximum value in the valve eq.
    @variable(model, 0 <= wp[1:nfe, 2:ncp+1] <= 2);  # auxiliary value for max expr.
    @variable(model, 0 <= wm[1:nfe, 2:ncp+1] <= 2);  # auxiliary value for max expr.


    ## constraints
    # phase equilibrium
```

```
    @NLconstraint(model, ant[i in 1:nc, j in 1:nfe, k in 2:ncp+1],
        psat[i,j,k] == 0.1 * 10^(A[i] -  B[i]/(T[j,k]*1e3 + C[i])))

    @constraint(model, eqp[i in 1:nc, j in 1:nfe, k in 2:ncp+1],
        K[i,j,k] * p[j,k] == psat[i,j,k]);

    @NLconstraint(model, eqc[i in 1:nc, j in 1:nfe, k in 2:ncp+1],
        beta[j,k] * K[i,j,k] * x[i,j,k] == y[i,j,k]);

    @constraint(model, equil[j in 1:nfe, k in 2:ncp+1],
        sum(y[i,j,k] for i in 1:nc) - sum(x[i,j,k] for i in 1:nc) == 0);

    @constraint(model, aux[j in 1:nfe, k in 2:ncp+1],
        beta[j,k] - 1 - sv[j,k] + sl[j,k]  == 0);

    # material balance
    @constraint(model, bal_c[i in 1:nc, j in 1:nfe, k in 2:ncp+1],
        sum(Mc[i,j,l] * adot[l,k] for l in 1:ncp+1) ==
        dt*(F[j]*z[i] - FVr[j,k]*y[i,j,k] - FLr[j,k]*x[i,j,k]));

    @constraint(model, cont_m[i in 1:nc, j in 1:nfe-1],
        Mc[i,j,end] - Mc[i,j+1,1] == 0);

    @constraint(model, tot_c[i in 1:nc, j in 1:nfe, k in 2:ncp+1],
        Mc[i,j,k] ==  ML[j,k]*x[i,j,k] + MV[j,k]*y[i,j,k]);

    @constraint(model, bal_t[j in 1:nfe, k in 2:ncp+1],
        sum(Mc[i,j,k] for i in 1:nc) == ML[j,k] + MV[j,k]);

    # energy balance
    @constraint(model, bal_e[j in 1:nfe, k in 2:ncp+1],
        sum(U[j,l] * adot[l,k] for l in 1:ncp+1) ==
        dt*(F[j]*hin[j] - FLr[j,k]*hL[j,k] - FVr[j,k]*hV[j,k] + Q[j]));

    @constraint(model, cont_u[j in 1:nfe-1], U[j,end] - U[j+1,1] == 0);

    @constraint(model, ent_l[j in 1:nfe, k in 2:ncp+1],
        hL[j,k] == sum(x[i,j,k] * CpL[i] * (T[j,k] - Tref[i]) for i in 1:nc));

    @constraint(model, ent_v[j in 1:nfe, k in 2:ncp+1],
        hV[j,k] == sum(y[i,j,k] * (h_vap[i] + CpV[i]*(T[j,k] - Tref[i]))
        for i in 1:nc));

    @constraint(model, ent_t[j in 1:nfe, k in 2:ncp+1],
        H[j,k] == ML[j,k] * hL[j,k] + MV[j,k] * hV[j,k]);
    @constraint(model, ent[j in 1:nfe, k in 2:ncp+1],
        H[j,k] == U[j,k] + p[j,k] * V);

    # volume calculation
    @constraint(model, vol[j in 1:nfe, k in 2:ncp+1],
        VV[j,k] + VL[j,k] == V);

    @constraint(model, v_vap[j in 1:nfe, k in 2:ncp+1],
        VV[j,k] * p[j,k] == MV[j,k] * R * T[j,k]);

    @constraint(model, v_liq[j in 1:nfe, k in 2:ncp+1],
        VL[j,k] * rhoL[j,k] * 1e3 == ML[j,k]);
```

```
    @NLconstraint(model, densL[j in 1:nfe, k in 2:ncp+1],
        1/rhoL[j,k] == sum(x[i,j,k]/(rho[i]) for i in 1:nc))

    # valves (outlet)
    @constraint(model, abs1[j in 1:nfe, k in 2:ncp+1],
        p[j,k] - p0 == avp[j,k] - avm[j,k]);

    @constraint(model, abs2[j in 1:nfe, k in 2:ncp+1],
        ap[j,k] == avp[j,k] + avm[j,k]);

    @NLconstraint(model, max1[j in 1:nfe, k in 2:ncp+1],
        p0 - p[j,k] == (ap[j,k]^0.5 + ϵ)*(wp[j,k] - wm[j,k]));

    @constraint(model, max2[j in 1:nfe, k in 2:ncp+1],
        am[j,k] == wm[j,k]);

    @constraint(model, valL[j in 1:nfe, k in 2:ncp+1],
        FLr[j,k] == cL * (VL[j,k]/V) * am[j,k]);

    @constraint(model, valV[j in 1:nfe, k in 2:ncp+1],
        FVr[j,k] == cV * (VV[j,k]/V) * am[j,k]);


    ## objective function defined as an expression
    return [(sv, FVr); (sl, FLr); (avp, avm); (wp, wm)], T;

end

"""
    set_initial_condition!(model, icmodel)
Set the result from the from the simulation of the icmodel as initial condition
for the current model.
`model` is a JuMP model containing the MPCC problem;
`icmodel` the JuMP model used as basis for the initial condition
"""

function set_initial_condition!(model::Model,icmodel::Model)

    Mc0 = value.(icmodel[:Mc])
    U0 = value.(icmodel[:U])

    # initializing dynamic variables
    Mc = model[:Mc]
    U = model[:U]

    fix.(Mc[:,1,1], Mc0, force = true);
    fix.(U[1,1], U0, force = true);
end

function set_initial_condition!(model::Model,value)

    # initializing dynamic variables
    Mc = model[:Mc]
    U = model[:U]

    Mc0 = value[:Mc]
```

```
    U0 = value[:U]

    fix.(Mc[:,1,1], Mc0, force = true);
    fix.(U[1,1], U0, force = true);
end
"""
    get_ss_values!(ssmodel)

Returns a vector which containts the steady state values.

`ssmodel` is a JuMP model containing the MPCC problem;
"""

function get_ss_values(ssmodel::Model)
    ssvars = [:x; :y; :FLr; :FVr; :K; :psat; :hL; :hV;
              :T; :p; :VV; :VL; :Mc; :ML; :MV; :U; :H;
              :rhoL; :beta; :sv; :sl; :ap; :avp; :avm; :am; :wp; :wm;]

    ssvalues = Dict{Symbol,Any}();
    for v in ssvars
        ssvalues[v] = value.(ssmodel[v]);
    end
    return ssvalues
end

"""
    get_last_values!(ssmodel)

Returns a vector which containts the values of the variables for the last
nfe and ncp

`ssmodel` is a JuMP model containing the MPCC problem;
"""

function get_last_values(ssmodel::Model)
    ssvars = [:x; :y; :FLr; :FVr; :K; :psat; :hL; :hV; :T;
              :p; :VV; :VL; :Mc; :ML; :MV; :U; :H; :rhoL; :beta;
              :sv; :sl; :ap; :avp; :avm; :am; :wp; :wm;]

    ssvalues = Dict{Symbol,Any}();
    for v in ssvars
        if ndims(value.(ssmodel[v])) == 3
            ssvalues[v] = value.(ssmodel[v][:,end,end])
        else
            ssvalues[v] = value.(ssmodel[v][end,end])
        end
    end
    return ssvalues
end
"""
    get_last_values_each_element!(ssmodel,fe)

Returns a vector which containts the value from the last ncp for each
final element.

`ssmodel` is the JuMP model;
"""
```

```
function get_last_values_each_element(ssmodel::Model,fe)
    ssvars = [:x; :y; :FLr; :FVr; :K; :psat; :hL; :hV;
              :T; :p; :VV; :VL; :Mc; :ML; :MV; :U; :H;
              :rhoL; :beta; :sv; :sl; :ap; :avp; :avm; :am; :wp; :wm;]

    ssvalues = Dict{Symbol,Any}();
    for v in ssvars
        if ndims(value.(ssmodel[v])) == 3
            ssvalues[v] = value.(ssmodel[v][:,fe,end])
        else
            ssvalues[v] = value.(ssmodel[v][fe,end])
        end
    end
    return ssvalues
end

"""
    set_initial_guess_from_ss!(model, ssguess,feed)

set the inital guess for a model from a steady state model

`model` is the JuMP model to get initial guesses;
`ssguess` a vector containg the result from a steady state sim.
"""


function set_initial_guess_from_ss!(model::Model, ssguess::Dict{Symbol,Any})
    nc = length(ssguess[:x]);
    guesses = copy(ssguess)

    for (key, value) in guesses
        if length(value) == nc
            for j in 1:nc
                set_start_value.(model[key][j,:,:], value[j])
            end
        else
            set_start_value.(model[key][:,:], value)
        end
    end
end
```

G.2 The Stationary Flash Tank Simulation

```
#
# Single flash tank
#

import PyPlot;
using Revise
using MPCCLibrary
using LaTeXStrings
using Serialization

include("flash_functions.jl");



# Defining the substances
water =    Substance("water",    1/18.02,     75,    35   , 40.660,
[4.6543;  1435.264; -64.848],  0.29815);

methanol = Substance("methanol", 0.792/32.04, 81.08, 44.06, 35.210,
[5.15853; 1569.613; -34.846],  0.29815);


# Creating feed stream
feed = Feed();
feed.Comp = [methanol; water];
feed.F = 0.1;                                              # kmol/s
feed.z = [0.5; 0.5];                                       # composition
feed.T = 0.410;                                            # Kk
feed.p = 0.12;


ssmodel = create_model(linear_solver = "ma97", print_level = 0);
sscomp = flash_ss!(ssmodel, feed, Q = -0.19);
@expression(ssmodel, obj, 1.);
classic_solve_mpcc!(ssmodel, sscomp);
#values = get_ss_values(ssmodel);
print_ss_values(ssmodel,true,-0.19)

ssmodel = create_model(linear_solver = "ma97", print_level = 0);
sscomp = flash_ss!(ssmodel, feed, Q = -0.20);
@expression(ssmodel, obj, 1.);
classic_solve_mpcc!(ssmodel, sscomp);
values = get_ss_values(ssmodel);
print_ss_values(values,true,-0.20)


Q1 = collect(0:-0.03:-0.18)
Q2 = collect(-0.28:-0.2:-3.70)
Q3 = collect(-3.83:-0.03:-4.0)
Q = vcat(Q1, Q2, Q3)
tot_t = length(Q)
```

```
T = Vector{Float64}(undef,tot_t)
p = Vector{Float64}(undef,tot_t)
beta = Vector{Float64}(undef,tot_t)
x = Vector{Array{Float64,1}}(undef, tot_t)
y = Vector{Array{Float64,1}}(undef, tot_t)
Mc = Vector{Array{Float64,1}}(undef, tot_t)
FLr = Vector{Float64}(undef,tot_t)
FVr = Vector{Float64}(undef,tot_t)
VV =  Vector{Float64}(undef,tot_t)
VL = Vector{Float64}(undef,tot_t)
ML = Vector{Float64}(undef,tot_t)
MV =  Vector{Float64}(undef,tot_t)
sv = Vector{Float64}(undef,tot_t)
sl = Vector{Float64}(undef,tot_t)


for t in 1:tot_t

println("i = ", t)
println("Q = ", Q[t])

    ssmodel = create_model(linear_solver = "ma97", print_level = 0);
    sscomp = flash_ss!(ssmodel, feed, Q = Q[t]);
    @expression(ssmodel, obj, 1.);
    classic_solve_mpcc!(ssmodel, sscomp);
    values = get_ss_values(ssmodel);
    print_ss_values(values,true,Q[t])

    T[t] = values[:T]
    FLr[t] = values[:FLr]
    FVr[t] = values[:FVr]
    VV[t] = values[:VV]
    VL[t] = values[:VL]
    x[t] = values[:x]
    y[t] = values[:y]
    Mc[t] = values[:Mc]
    p[t] = values[:p]*10
    beta[t] = values[:beta]
    MV[t] = values[:MV]
    ML[t] = values[:ML]
    sv[t] = values[:sv]
    sl[t] = values[:sl]
end

output_file = open("ss_flash.jls", "w")
serialize(output_file, T)
serialize(output_file, FLr)
serialize(output_file, FVr)
serialize(output_file, VV)
serialize(output_file, VL)
serialize(output_file, x)
serialize(output_file, y)
serialize(output_file, beta)
serialize(output_file, p)
serialize(output_file, MV)
serialize(output_file, ML)
serialize(output_file, sv)
```

```
serialize(output_file, sl)
serialize(output_file, Mc)
close(output_file)
```

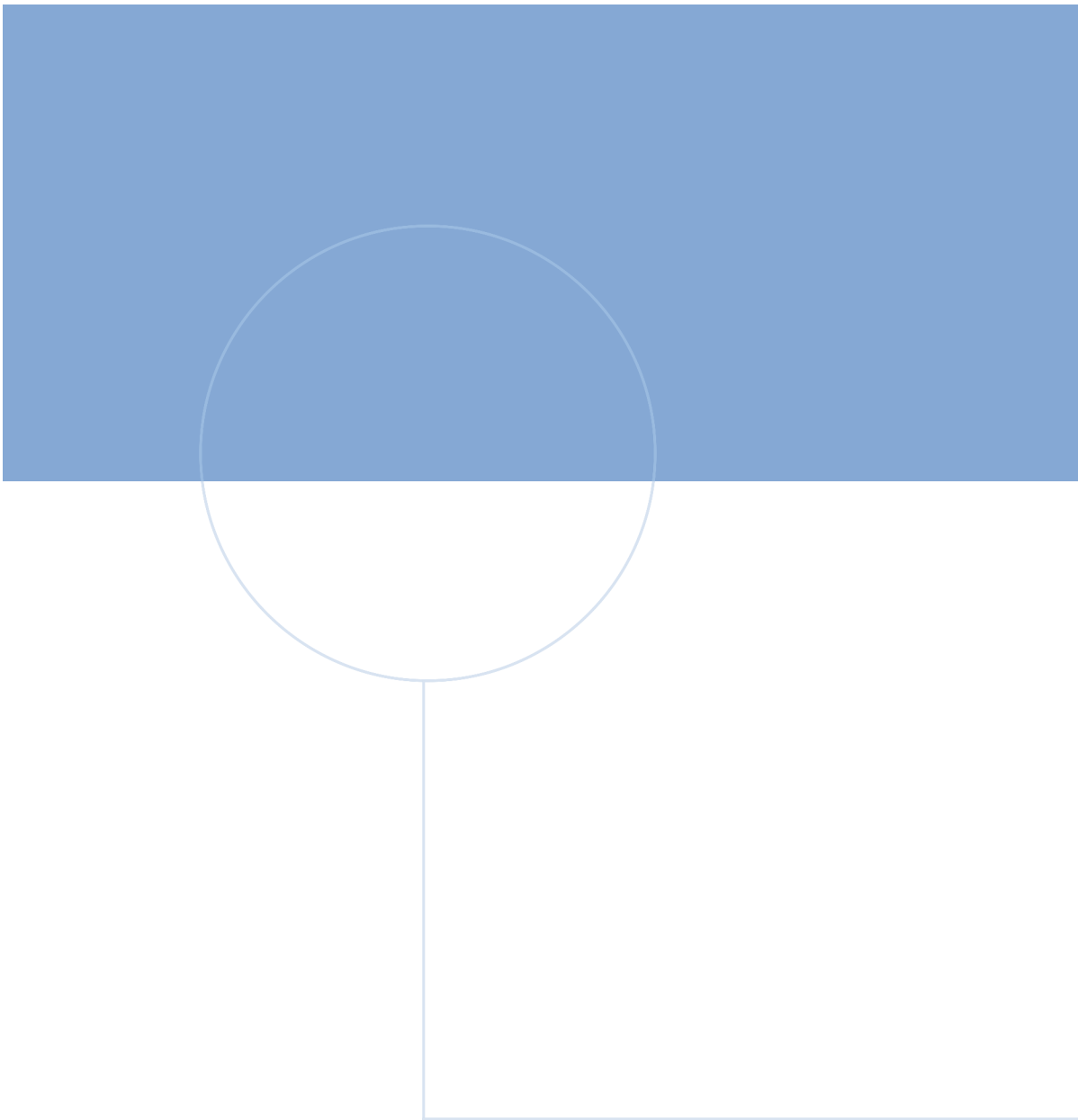
G.3 The Dynamic Flash Tank Simulation

```
import PyPlot;
using Revise
using MPCCLibrary
using LaTeXStrings
using Serialization

include("flash_functions.jl");

# Defining the substances
water =    Substance("water",    1/18.02,     75,    35   , 40.660,
[4.6543;  1435.264; -64.848],  0.29815);

methanol = Substance("methanol", 0.792/32.04, 81.08, 44.06, 35.210,
[5.15853; 1569.613; -34.846],  0.29815);


# Creating feed stream
feed = Feed();
feed.Comp = [methanol; water];
feed.F = 0.1;                                # kmol/s
feed.z = [0.5; 0.5];                         # composition
feed.T = 0.410;                              # kK
feed.p = 0.12;                               # MPa

Q = Vector{Float64}()

for t in 1:250
    if t < 5
        append!(Q,0)
    elseif 5 <= t <= 150
        temp = -4/(150-5) * (t-5)
        append!(Q,temp)
    else
        append!(Q,-4)
    end
end

# Dynamic parameters
nfe = 1
ncp = 3;
tspan = (0.,1);
dyn_par = Dynamic(nfe, ncp, tspan);
dt = (tspan[2]-tspan[1])/nfe
t = tspan[1]:dt:tspan[2]
tot_t = length(Q)


# Singel study state flash tank for inital condition
icmodel = create_model(linear_solver = "ma97", print_level = 0);
iccomp = flash_ss!(icmodel, feed, Q = 0.0);
@expression(icmodel, obj, 1.);
classic_solve_mpcc!(icmodel, iccomp);
```

```
ic = get_ss_values(icmodel);

T = Vector{Float64}(undef,tot_t)
p = Vector{Float64}(undef,tot_t)
beta = Vector{Float64}(undef,tot_t)
x = Vector{Array{Float64,1}}(undef, tot_t)
y = Vector{Array{Float64,1}}(undef, tot_t)
Mc = Vector{Array{Float64,1}}(undef, tot_t)
FLr = Vector{Float64}(undef,tot_t)
FVr = Vector{Float64}(undef,tot_t)
VV =  Vector{Float64}(undef,tot_t)
VL = Vector{Float64}(undef,tot_t)
ML = Vector{Float64}(undef,tot_t)
MV =  Vector{Float64}(undef,tot_t)
sv = Vector{Float64}(undef,tot_t)
sl =  Vector{Float64}(undef,tot_t)
avp = Vector{Float64}(undef,tot_t)
avm =  Vector{Float64}(undef,tot_t)
wp = Vector{Float64}(undef,tot_t)
wm =  Vector{Float64}(undef,tot_t)

values = Dict()
global t = 1;

while t <= 250
    Qt = Q[t]
    println("------------------------------")
    println("t = ", t)
    println("Q = ",Qt)
    if t == 1
        model = create_model(linear_solver = "ma97", print_level = 0)
        comp,Tm = flash_dynamic!(model,feed,dyn_par, Q = ones(nfe)*Qt)
        set_initial_condition!(model, icmodel);
        @expression(model, obj, 1.)
        classic_solve_mpcc!(model,comp, :obj)
        global values = get_last_values(model)

        T[t] = values[:T]
        FLr[t] = values[:FLr]
        FVr[t] = values[:FVr]
        VV[t] = values[:VV]
        VL[t] = values[:VL]
        x[t] = values[:x]
        y[t] = values[:y]
        Mc[t] = values[:Mc]
        p[t] = values[:p]
        beta[t] = values[:beta]
        MV[t] = values[:MV]
        ML[t] = values[:ML]
        sv[t] = values[:sv]
        sl[t] = values[:sl]
        avp[t] = values[:avp]
        avm[t] = values[:avm]
        wp[t] = values[:wp]
        wm[t] = values[:wm]

        global t += 1
```

```
elseif t == 90
    model = create_model(linear_solver = "ma97", print_level = 0)
    comp,Tm = flash_dynamic!(model,feed,Dynamic(111,3,(0.,111)),Q = Q[90:200])
    set_initial_condition!(model,values)
    set_initial_guess_from_ss!(model, values)
    @expression(model, obj, 1.)
    classic_solve_mpcc!(model,comp, :obj)

    for i in 1:111
        t = 89 + i
        println(t)
        values = get_last_values_each_element(model,i)
        T[t] = values[:T]
        FLr[t] = values[:FLr]
        FVr[t] = values[:FVr]
        VV[t] = values[:VV]
        VL[t] = values[:VL]
        x[t] = values[:x]
        y[t] = values[:y]
        Mc[t] = values[:Mc]
        p[t] = values[:p]
        beta[t] = values[:beta]
        MV[t] = values[:MV]
        ML[t] = values[:ML]
        sv[t] = values[:sv]
        sl[t] = values[:sl]
        avp[t] = values[:avp]
        avm[t] = values[:avm]
        wp[t] = values[:wp]
        wm[t] = values[:wm]

        println(T[t])
        println(beta[t])
    end
    global t = 201
    global values = get_last_values_each_element(model,111)

else
    model = create_model(linear_solver = "ma97", print_level = 0)
    comp,Tm = flash_dynamic!(model,feed,dyn_par, Q = ones(nfe)*Qt)
    @constraint(model,Tcon[ j in 1:nfe, k in 2:ncp+1], Tm[j,k] <= values[:T])
    set_initial_condition!(model,values)
    set_initial_guess_from_ss!(model, values)
    @expression(model, obj, 1.)
    classic_solve_mpcc!(model,comp, :obj)
    global values = get_last_values(model)

    T[t] = values[:T]
    FLr[t] = values[:FLr]
    FVr[t] = values[:FVr]
    VV[t] = values[:VV]
    VL[t] = values[:VL]
    x[t] = values[:x]
    y[t] = values[:y]
    Mc[t] = values[:Mc]
    p[t] = values[:p]
```

```
        beta[t] = values[:beta]
        MV[t] = values[:MV]
        ML[t] = values[:ML]
        sv[t] = values[:sv]
        sl[t] = values[:sl]
        avp[t] = values[:avp]
        avm[t] = values[:avm]
        wp[t] = values[:wp]
        wm[t] = values[:wm]

        global t = t + 1;

    end
end

output_file = open("singelFlash.jls", "w")
serialize(output_file, T)
serialize(output_file, FLr)
serialize(output_file, FVr)
serialize(output_file, VV)
serialize(output_file, VL)
serialize(output_file, x)
serialize(output_file, y)
serialize(output_file, beta)
serialize(output_file, p)
serialize(output_file, MV)
serialize(output_file, ML)
serialize(output_file, sv)
serialize(output_file, sl)
serialize(output_file, avp)
serialize(output_file, avm)
serialize(output_file, wp)
serialize(output_file, wm)
serialize(output_file, Mc)
close(output_file)
```



NTNU

Norwegian University of
Science and Technology