Department of Chemical Engineering

TKP4580 - Chemical Process Technology, Specialization Project

Evaluation of Strategies for Solving Mathematical Programs with Complementarity Constraints

Author:
Ann Iren Fossøy

Supervisor:
Johannes Jäschke

Co-supervisor:
Caroline Satye Nakama

NTNU
Norwegian University of Science and Technology

December 16, 2022

Abstract

The aim of this project was to evaluate different approaches for solving mathematical programs with complementarity constraints, MPCCs. This complementarity formulation has a wide range of applications and is found particularly beneficial in representing discrete events, as binary numbers can be avoided. Due to this, the complementarity formulations tend to be utilised rapidly in several fields, such as science, economics and engineering, especially chemical engineering. MPCCs are consequently evolving to be an active field of interest both academically and industrially, and it is therefore important to solve them precisely. However, MPCCs can be particularly challenging to solve with standard nonlinear application tools, due to some inherent properties. Further modifications of MPCCs are therefore crucial. To address this, four different solution strategies were investigated. Two of the approaches relied on relaxing the complementarity constraints, whereas the others considered a penalisation reformulation. In the evaluation process, properties such as performance, accuracy, complementarity satisfaction and number of iterations were considered. The result of the study shows that the most appropriate approach is highly dependent on the purpose, as the different methods offer different advantages. The dynamic approach was shown to be promising due to its level of accuracy and the automatic adjustment of the tuning parameter, but had the disadvantage of needing many iterations. The relaxation approaches were found to be deficient, as the complementarity constraints were not satisfied to the degree of accuracy that was set. However, the relaxation parameters were found to be easier to tune compared to the penalisation parameters. Further research should draw attention to the penalisation reformulation, as this approach showed to present better cost-benefits in general. However, it is necessary to take a closer look at this method to make a more definite assessment of its performance.

Contents

1 Introduction **1**

2 Background and Theory **3**

2.1 Constrained Optimisation 3

2.1.1 Discrete and Continuous Optimisation 3

2.1.2 Convexity Property 4

2.2 Constraint Qualifications 4

2.3 Optimality Conditions 5

2.4 The Penalty and Relaxation Approaches 6

2.5 Mathematical Program with Complementarity Constraints 7

2.6 Interior Point Optimiser 8

3 Methodology **10**

3.1 The Reformulations of the MPCCs 10

3.1.1 The Relaxation Approaches 10

3.1.2 The Exact Penalisation Approach 11

3.1.3 The Practical Interior-penalty Method 12

3.2 Bound Relax Factor for IPOPT 13

3.3 The MacMPEC Collection 14

3.4 Implementation and Data Processing 14

3.4.1 The Performance Profiles 14

3.4.2 Percentage Error 15

3.4.3 Complementarity Satisfaction 15

3.4.4 Initial Guesses 15

4 Results and Discussions **16**

4.1 Comparison of Performance 16

4.1.1 Feasible Solutions for the MPCC 18

4.1.2 Changing the Penalisation Parameter 19

4.1.3 Changing the Relaxation Parameter 20
4.2 Satisfaction of the Complementarity Constraints 21
4.3 Comparison of Accuracy 22
4.4 Comparison of Iterations 24
4.5 Changing the Initial Values 25

5 Final remarks **27**

Appendices **31**

A Problems from MacMPEC Collection **32**

B The Approaches Implemented in Julia **35**
B.1 The Relaxation Methods 35
B.2 The Static Penalisation Method 40

List of Figures

4.1 The performance profile of the different reformulations. Convergence to feasible solutions of the reformulations are accounted as solutions 17
4.2 The performance profile of the different reformulations. Only feasible solutions of the MPCCs were accounted as solved. 18
4.3 The performance profile of Reg(ϵ) with different relaxation values. 20
4.4 Satisfaction of the complementarity constraints. All feasible solutions of the reformulations are included. 21
4.5 The percentage of problems solved within a certain error range. 22
4.6 The percentage of problems solved within a certain number of iterations for each approach. 24

Nomenclature

Acronyms

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

Chapter 1

Introduction

Mathematical programs with complementarity constraints, MPCCs, have evolved to become an active subject in the literature during the past decades. An explanation for this increased attention is the great variety of applications for this particular mathematical formulation [1]. MPCC is a constrained optimisation problem containing complementarity constraints, which relates two or more variables by forcing at least one of them to be at its bound. This property makes it possible to model disjunctions without relying on binary variables [2]. The complementarity formulation has been found beneficial to areas within science, economic and engineering [3]. A broad range of applications is especially found in chemical engineering-related fields, such as safety valve operation, flow reversal and phase disappearance. Conventionally, problems containing disjunctions have been solved with mixed integer programming or disjunctive programming techniques. However, these methods have proven to be time-consuming, especially for cases with several discrete decisions. The complementarity formulation becomes beneficial as certain modifications allow for standard nonlinear programming tools to be applied, which may improve solution time significantly [2]. However, despite the large field of applications, existing solution strategies for the MPCCs need to be improved as they are not fully developed. Solving MPCCs may be demanding for classical nonlinear optimisation tools, due to some inherent properties of MPCCs. To accommodate these challenges, several approaches have been proposed, such as relaxation, smoothing, penalisation and lifting [4].

In this project, mainly four different solution strategies are studied and compared. Two of the approaches are relaxation approaches which reformulate the MPCC by relaxing the complementarity condition by a relaxation parameter. The two other reformulations are penalisation approaches where the complementarity relationship is replaced by a penalty term in the objective function. For one of those methods, the penalisation parameter is manually selected and remains constant during the optimisation process. In the other approach, the penalisation parameter is dynamically updated as the optimisation proceeds. The solution strategies are being tested on several different MPCCs implemented in Julia by using the modelling language JuMP [5]. The result is further utilised to evaluate and compare the approaches in terms of performance, complementarity satisfaction, num-

ber of iterations and accuracy.

The aim of this project is to evaluate and compare the different solutions strategies. In this report, relevant background information, such as general optimisation knowledge, MPCC properties and familiar solving strategies, are found in Chapter 2. Chapter 3 covers methodology and provides information about the implementation, and describes how the project was completed. Chapter 4 presents the achieved results and discusses them. Lastly, Chapter 5 provides recommendations for further research, as well as the overall conclusion.

Chapter 2

Background and Theory

This chapter presents relevant background information related to constrained optimisation, as the main focus of this project will be on a special class within this field, MPCC. However, MPCCs are known to be challenging to solve due to some inherent properties contradicting some fundamental key concepts related to optimisation. In this chapter, these key concepts will be elaborated on, as well as how these challenges can be addressed.

2.1 Constrained Optimisation

A constrained optimisation problem consists of minimising an objective function with respect to certain constraints on the variables. The general formulation is given as follows,

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

where f is the objective function and $c_i,\ i \in \mathcal{E}$ and $c_i,\ i \in \mathcal{I}$ are the equality and inequality constraints, respectively. The set of points that satisfy the constraints is defined as the feasible set, and can be mathematically expressed as,

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

However, if the feasible set is empty due to constraints mutually contradicting each other, the problem is defined as infeasible, and no solution exists [6].

2.1.1 Discrete and Continuous Optimisation

In the field of optimisation, a distinction is made between discrete and continuous optimisation. Discrete optimisation is characterised by having decision variables which belong to a finite set and

can be either integers or binary numbers. On the contrary, the decision variables in continuous optimisation are drawn from an infinite set. Due to this assumption, continuous optimisation tends to be easier to solve as the gradient of the objective and constraint functions, as long as they exist, can give precious information about the behaviour of the function close to a specific point. An optimisation algorithm can take advantage of this smoothness in the prediction of an efficient search direction. However, this is not the case for discrete optimisation as the behaviour of the function may change significantly from one feasible point to another [6].

2.1.2 Convexity Property

Convexity is a desired property regarding continuous optimisation, as the problems become remarkably easier to solve. An optimisation problem is convex if both the objective function and the feasible set are convex. The feasible set will be convex if the straight line connecting any points within the set, lies completely inside this set. This is the case only if the constraint functions are linear, and the inequality constraint functions are concave. An objective function is convex if the following property is satisfied for any arbitrary numbers of x and y,

$$f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha)f(y), \tag{2.3}$$

where α is any number between 0 and 1. The convexity property guarantees that there is only one optimum, which is the global solution. A global solution is the solution that leads to the smallest possible value of the objective function. The convex property facilitates the convergence of these solutions. Even for large-scale problems, convex optimisation tends to be remarkably efficient [6].

A nonconvex problem, contrary to convex problems, may have a widely varying curvature, several saddle points and multiple feasible regions containing multiple local solutions. As follows, global solutions are significantly more challenging to locate compared to local solutions, which are optimum within a restricted region of possible solutions. Additionally, the optimisation algorithm may require an initial guess. The choice of initial guess is found crucial, as it has a great impact on the local solution obtained. Even-though convexity is desired, many practical problems are nonconvex which may make them difficult to solve accurately within a reasonable amount of time [7].

2.2 Constraint Qualifications

In order to solve optimisation problems, many optimisation algorithms are designed in such a way that they must determine whether a descent step from a given feasible point is possible. For nonlinear programs, the objective function and the constraint functions need to be linearized by Taylor series expansions. However, if the linearized approximation resembles the geometric feature of the feasible set poorly, the information obtained from the approximation will be insufficient. To prevent this, certain assumptions need to be taken into account. These assumptions are defined as

constraint qualifications and ensure that the constraint set, Ω, and the linearized approximation behave similarly close to the feasible point. Constraint qualifications introduce certain restrictions on the constraint functions, eliminating irregularities at the boundary of the feasible set. These qualifications are fundamental assumptions used in the design phase of optimisation algorithms and are therefore crucial for the algorithm to work properly [6].

There exist several different constraint qualifications, but the most frequently used is the linear independence constraint qualification, LICQ. LICQ states that the gradient of the active constraints needs to be linearly independent of each other. Mangasarian-Formovitz constraint qualification, MFCQ, is another, but it is a weaker constraint qualification. MFCQ requires the gradients of the equality constraints to be linear independent, and the existence of a search direction d that fulfils the following requirements,

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

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

where $\mathcal{A}(x^*)$ is the active set containing all the equality constraints and the inequality constraints which are active at their bound. The first requirement gives the direction of the vector d. The linearized active inequality constraints form a region, which d must point to the interior of. The last requirement indicates that d must be in the null space of the equality constraint gradients [2].

2.3 Optimality Conditions

In the characterisation of solutions, a distinction is made between necessary and sufficient conditions. The necessary conditions are required to be satisfied for every possible solution, but can not be used to confirm that the true optimum has been found. Nevertheless, the necessary conditions are helpful to identify candidates which qualify as possible optimum. Sufficient conditions, on the other hand, can be used to declare that a certain point is indeed the optimal solution.

The first-order necessary conditions for a constrained optimisation problem are known as the Karush-Kuhn-Tucker conditions, often referred to as the KKT conditions. For the KKT conditions to hold, LICQ has to be satisfied at any local solution. Additionally, both the objective and the constraint functions need to be continuously differentiable. The KKT conditions state that for any local solution, x^*, there is a Lagrange multiplier vector, λ^*, such that the following equations are satisfied,

$$\nabla_x \mathcal{L}(x^*, \lambda^*) = 0, \tag{2.4a}$$
$$c_i(x^*) = 0, \quad \text{for all } i \in \mathcal{E}, \tag{2.4b}$$
$$c_i(x^*) \geq 0, \quad \text{for all } i \in \mathcal{I}, \tag{2.4c}$$
$$\lambda_i^* \geq 0, \quad \text{for all } i \in \mathcal{I}, \tag{2.4d}$$
$$\lambda_i^* c_i(x^*) = 0, \quad \text{for all } i \in \mathcal{I} \cup \mathcal{E}. \tag{2.4e}$$

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

$$\mathcal{L}(x, \lambda) = f(x) + \sum_{i \in \mathcal{E}} \lambda_i c_i(x) + \sum_{i \in \mathcal{I}} \lambda_i c_i(x). \tag{2.5}$$

However, the KKT condition alone is not sufficient to characterise an optimum. In addition to satisfying the KKT condition, as well as the constraint qualifications, it is also required that the curvature in the directions of the constraints is positive. This can be mathematically expressed as,

$$d^T \nabla_{xx}^2 \mathcal{L}(x^*, \lambda^*) d \geq 0, \tag{2.6}$$

where $d \neq 0$ is a constraint direction which fulfils the following requirements,

$$\nabla c_i(x^*)^T d = 0, \text{ for all } i \in \mathcal{E}, \tag{2.7a}$$
$$\nabla c_i(x^*)^T d = 0, \text{ for all } i \in \mathcal{A}(x^*) \cap \mathcal{I} \text{ where } \lambda_i^* > 0, \tag{2.7b}$$
$$\nabla c_i(x^*)^T d \geq 0, \text{ for all } i \in \mathcal{A}(x^*) \cap \mathcal{I} \text{ where } \lambda_i^* = 0. \tag{2.7c}$$

The second-order sufficient conditions state that if a feasible point, x^*, satisfies the first-order necessary conditions, i.e. the KKT conditions given in equation (2.4), and the second-order necessary conditions, i.e. the curvature requirement given in equation (2.6), it can be concluded that x^* is indeed a solution for the constrained optimisation problem [6].

2.4 The Penalty and Relaxation Approaches

The penalty method is a class of algorithms used to solve constrained optimisation problems. In the penalisation approach, the constrained optimisation problems are fully or partly replaced by an unconstrained problem. This is done by adding penalisation terms to the objective function which discourages violations of the previous constraints. The constraints in a general optimisation problem, see equation (2.1), can be incorporated into the objective function by using an exact penalty objective function, given as follows

$$\min_{x,y} \quad f(x, y) + \rho \sum_{i \in \mathcal{E}} |c_i(x, y)| + \rho \sum_{i \in \mathcal{I}} [c_i(x, y)]^-, \tag{2.8}$$

where $[y]^- := \max(-y, 0)$, and $\rho > 0$ is the penalty parameter [6]. The modified unconstrained optimisation problem can converge to the exact solution of the original constrained program for

particular values of the penalty parameter. The value of the penalty parameter is therefore crucial for the performance of this method [2].

Another optimisation technique is the relaxation approach. In this approach, the equality constraints are relaxed by a positive relaxation parameter, ϵ. The general constrained optimisation problem, see equation (2.1), can then be rewritten as,

$$\begin{aligned} \min_{x,y} \quad & f(x,y) \\ \text{s.t.} \quad & c_i(x,y) \geq \epsilon \;\; i \in \mathcal{E} \\ & c_i(x,y) \geq 0 \;\; i \in \mathcal{I}. \end{aligned} \tag{2.9}$$

This optimisation problem will be significantly easier to solve as the hard constraints are relaxed [6].

2.5 Mathematical Program with Complementarity Constraints

A mathematical program with complementarity constraints, MPCC, is a type of optimisation problem containing complementarity constraints. These constraints can be mathematically expressed as, $0 \leq x \perp y \geq 0$, such that the overall MPCC formulation becomes,

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

where $\perp$ is the complementarity operator. A complementarity constraint describes a relationship between two variables, or expressions, where at least one must be at its bound. This relationship can mathematically be expressed as,

$$\begin{aligned} & x_i = 0 \vee y_i = 0 \;\; \forall i \\ & x \geq 0, \;\; y \geq 0, \end{aligned} \tag{2.11}$$

where the logical "or" operator can be characterized as inclusive as both variables can be at their bound [2]. It is appealing to utilise standard nonlinear programming, NLP, solvers to tackle MPCCs, as they are already well-developed and ensure fast convergence [8]. To be able to do so, other equivalent analytical forms of the complementarity relationship must be taken into account,

$$x^T y = 0, \;\; x \geq 0, y \geq 0, \tag{2.12a}$$

$$x_i y_i = 0 \;\forall i, \;\; x \geq 0, y \geq 0, \tag{2.12b}$$

$$x_i y_i \leq 0 \;\; \forall i, \;\; x \geq 0, y \geq 0. \tag{2.12c}$$

However, further adjustments are crucial as NLP solvers tend to fail when these complementarity reformulations are directly used. MPCC has an inherent nonconvexity, due to the complementarity constraints. As a result, even locating local solutions can be found challenging. Without any further modifications, MPCCs fail to satisfy important constraint qualifications such as LICQ and MFCQ at any feasible point. This is problematic as the NLP tools are built under the assumption of those constraint qualifications.

There exist different concepts of stationarity for MPCCs, as many of the NLP methods can give convergence to stationary points that are not necessarily optimum. Bouligand stationarity, commonly known as B-stationarity, is the stationarity which characterises optimality for MPCCs. A point is characterised as B-stationary if it is feasible and if the solution to the following linear program, obtained by linearizing the objective function and constrain functions,

$$\begin{aligned} \min_{d} \quad & \nabla f(x^*, y^*, z^*)^T d \\ \text{s.t.} \quad & c_i(x^*, y^*, z^*) + \nabla c_i(x^*, y^*, z^*)^T d \geq 0, \quad i \in \mathcal{I} \\ & c_i(x^*, y^*, z^*) + \nabla c_i(x^*, y^*, z^*)^T d = 0, \quad i \in \mathcal{E} \\ & 0 \leq x^* + d_x \perp y^* + d_y \geq 0, \end{aligned} \tag{2.13}$$

is $d = 0$. However, B-stationarity may be computationally demanding to detect. A more practical approach is strong stationarity which may imply B-stationarity if a special kind of constraint qualification, MPEC-LICQ, holds. The MPEC-LICQ demands that the set of gradient vectors are linearly independent of each other. If MPEC-LICQ is satisfied at the solution to the MPCC, it can be further proven that the point will be strongly stationary [2].

2.6 Interior Point Optimiser

The interior point optimiser, more commonly known as IPOPT, is a software package to solve continuous optimisation problems. IPOPT serves as a general nonlinear programming, NLP, solver, capable of solving large-scale nonlinear and nonconvex constrained optimisation problems. IPOPT is based on a barrier approach, which belongs to the interior point methods. IPOPT handles the optimisation problems by reformulating it into general form, presented as follows,

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & c(x) = 0 \\ & x \geq 0. \end{aligned} \tag{2.14}$$

In this formulation $c(x)$ is a matrix containing both the inequality and the equality constraints. The inequality constraints are converted into equality constraints by the use of slack variables such that the inequality constraints become, $c_i(x) - s = 0$. The slack variables will then be incorporated into the vector x. By considering the auxiliary barrier formulation the optimisation problem will

be further expressed as,

$$\begin{aligned} \min_{x,y} \quad & f(x,y) - \mu \sum_{i}^{n} \ln(x_i) \\ \text{s.t.} \quad & c(x) = 0, \end{aligned} \tag{2.15}$$

where μ defines the barrier parameter. In terms of this formulation, the optimisation problem will approach the optimal solution, x^*, when μ approaches zero. IPOPT starts by solving the optimisation problem for a given μ to a relaxed accuracy. The algorithm will then decrease μ and find a solution with improved accuracy, using the previous solution as basis. The objective function decreases for each iteration by computing steps based on the newton method with line search. This approach is repeated until the first-order optimality condition is satisfied within a certain tolerance [9].

Chapter 3

Methodology

In this project, several MPCCs are reformulated into different nonlinear optimisation problems. A total of four different reformulations were scrutinized and will be presented further in this chapter, as well as how the results were obtained.

Throughout this project, some important assumptions and definitions have been made. Solving the problem will further be referred to as converging to a local minimum, as ensuring global convergence is too demanding. Additionally, a distinction was made between solutions to the original problem, the MPCC, and solutions to the reformulations. In most cases, they will be equivalent, as a solution of the reformulation is also a solution of the MPCC. However, there may arise exceptions where the solution of the reformulation violates the MPCC constraints and hence is deemed infeasible. This difference will be specified and taken into account in the Results and Discussion Chapter. It may also occur that IPOPT solves the problem with a numerical error, or solves it "almost locally", meaning that the problem is solved with respect to a reduced tolerance. In this project, both these cases will be seen as equivalent to the approach being incapable of solving the problem. Lastly, the initial values for a specific problem remain the same independently of the solution strategy being applied.

3.1 The Reformulations of the MPCCs

The different approaches implemented and evaluated in this project are based on the penalisation approach or the relaxation approach, which were presented in section 2.4.

3.1.1 The Relaxation Approaches

In relaxation approaches, the complementarity constraints are relaxed with a positive relaxation parameter, ϵ. Two different reformulations were investigated, where the first reformulation is as

follows,

$$\begin{aligned} \text{Reg}(\epsilon): \quad \min_{x,y,z} \quad & f(x,y,z) \\ \text{s.t.} \quad & c_i(x,y,z) = 0,\ i \in \mathcal{E} \\ & c_i(x,y,z) \geq 0,\ i \in \mathcal{I} \\ & x, y \geq 0 \\ & x_i y_i \leq \epsilon, \quad \forall i. \end{aligned} \tag{3.1}$$

In this reformulation, each individual complementarity pair, (x_i, y_i), is relaxed with the same specified relaxation parameter, ϵ. The other reformulation investigated is

$$\begin{aligned} \text{RegComp}(\epsilon): \quad \min_{x,y,z} \quad & f(x,y,z) \\ \text{s.t.} \quad & c_i(x,y,z) = 0,\ i \in \mathcal{E} \\ & c_i(x,y,z) \geq 0,\ i \in \mathcal{I} \\ & x, y \geq 0 \\ & x^T y \leq \epsilon. \end{aligned} \tag{3.2}$$

In this method, instead of relaxing each complementarity pair individually, the inner product of the two complementarity vectors, x and y, is relaxed. The solution of $\text{Reg}(\epsilon)$ and $\text{RegComp}(\epsilon)$ will converge to the solution of the MPCC when ϵ is driven to zero. However, there may be numerical difficulties if the parameter is selected to be significantly small [2].

In this project, the value of the relaxation parameter, ϵ was selected manually and was chosen to be $\epsilon = 10^{-6}$ when the different approaches were compared. However, attempts were made to change the parameter to observe how it affected the results.

3.1.2 The Exact Penalisation Approach

In the exact penalisation approach, the MPCC is rewritten into this particular form,

$$\begin{aligned} \text{PF}(\rho): \quad \min_{x,y,z} \quad & f(x,y,z) + \rho x^T y \\ \text{s.t.} \quad & c_i(x,y,z) = 0,\ i \in \mathcal{E} \\ & c_i(x,y,z) \geq 0,\ i \in \mathcal{I} \\ & x, y \geq 0, \end{aligned} \tag{3.3}$$

where ρ is a positive penalisation parameter. In the presented reformulation, the complementarity pairs are added to the objective function and removed as constraints. Usually, according to the general penalisation reformulation, an inequality constraint such as $c(x) = x^T y \leq 0$ should be written in the following way, $\max(0, x^T y)$, before added to the objective function. In this case, however, as both variables are positive, $x, y \geq 0$, the formulation will be equivalent to writing $x^T y$. This problem will reflect the true solution of the MPCC for a sufficiently large value of ρ. However, the

appropriate value of ρ is dependent on the scaling and the extent of the problem. The penalisation formulation presented ensures the smoothness of the problem, which gives allowance for standard nonlinear programming tools to be applied.

In this project, the penalisation parameter was selected to be $\rho = 10$ when the different approaches were compared. However, there are some disadvantages of keeping the penalisation parameter constant throughout the optimisation. One specific value of ρ may cause convergence to a local solution nearby, which does not yield the smallest value of the objective function possible [10].

3.1.3 The Practical Interior-penalty Method

A way to improve performance and achieve greater numerical efficiency of the penalisation approach is to dynamically update the penalty parameter during the optimisation. In this project, a dynamic interior-penalty method for MPCCs, see Algorithm 1 [10], has been implemented [11]. The KKT conditions for the barrier formulation of the penalisation approach, given in equation (3.3), is

$$\begin{aligned} \nabla f(x) - \nabla c_{\mathcal{E}}(x)^T \lambda_{\mathcal{E}} - \nabla c_{\mathcal{I}}(x)^T \lambda_{\mathcal{I}} - \begin{pmatrix} 0 \\ \mu X^{-1} e - \pi y \\ \mu Y^{-1} e - \pi x \end{pmatrix} = 0, \\ s_i \lambda_i - \mu = 0, \ i \in \mathcal{I}, \\ c_i(x) = 0, \ i \in \mathcal{E}, \\ c_i(x) - s_i 0 = 0, i \in \mathcal{I}, \end{aligned} \tag{3.4}$$

where X and Y are diagonal matrices with x_i and y_i on the diagonal, respectively. Further, $c_{\mathcal{E}}(x)$ is all the equality constraints combined in a vector, while $c_{\mathcal{I}}(x)$ contains all the inequality constraints. The KKT can be further expressed in this form,

$$\nabla_x \mathcal{L}_{\mu,\pi}(x, s, \lambda) = 0, \tag{3.5a}$$

$$S\lambda_{\mathcal{I}} - \lambda e = 0, \tag{3.5b}$$

$$c_{\mathcal{E}}(x) = 0, \tag{3.5c}$$

$$c_{\mathcal{I}}(x) - s = 0. \tag{3.5d}$$

The implemented algorithm changes the penalty parameter if the conditions presented in (3.6) are satisfied, but the complementarity value, $||\min\{x^j, y^j\}||_\infty$, is still not equal to or below the defined tolerance. After updating the penalisation parameter, the algorithm will return to step 2 and compute a new primal-dual step based on the KKT conditions. However, if the complementarity value has been satisfied for the defined tolerance, but the stopping test fails, the algorithm will start a new iteration in the barrier loop. In each barrier iteration, the barrier parameter will decrease such that both ϵ^k_{comp} and ϵ^k_{pen} decrease as well. With this in mind, the algorithm as a whole can be written as shown below.

Algorithm 1 : A Practical Interior-Penalty Method for MPCCs [10].

Initialization: The initial primal and dual variables are, $z^0 = (x^0, s^0, \lambda^0)$. Choose, π^0 which is the initial penalty and $\gamma \in (0, 1)$, Set $j = 0, k = 1$

Repeat(barrier loop)

1. Choose the barrier parameter, μ^k and a stopping tolerance 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 $z^c = z^{j-1}$
   ii. Compute a primal-dual step d^j with $\mu = \mu^k$, $\pi = \pi^k$ and $z = z^c$ based on the KKT conditions given in equation (3.4).
   iii. Let $z^j = z^c + d^j$

   until the conditions as follows,

$$||\nabla_x \mathcal{L}_{\mu,\pi}(x, s, \lambda)|| \leq \epsilon^k_{pen}, \tag{3.6a}$$
$$||S\lambda_{\mathcal{I}} - \lambda e|| \leq \epsilon^k_{pen}, \tag{3.6b}$$
$$||c_{\mathcal{E}}(x)|| \leq \epsilon^k_{pen}, \tag{3.6c}$$
$$||c_{\mathcal{I}}(x) - s|| \leq \epsilon^k_{pen}, \tag{3.6d}$$

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

until a stopping test for the MPCC is satisfied

In this algorithm, j denotes the inner iteration, while k is the barrier loop iteration.

3.2 Bound Relax Factor for IPOPT

During the comparison of the different approaches, it was discovered that the complementarity variables could become negative, and were accordingly slightly violating the complementarity constraints. This is due to the bound relax factor of IPOPT. IPOPT relaxes the bounds by a default value of 10^{-8}, which permits negative values close to zero [12]. This is found particularly inconvenient for the RegComp(ϵ) approach, as it takes the inner product of the two complementarity vectors. To avoid negative values, the relax factor was reduced to 10^{-16}, which is used as a basis for the rest of the results throughout the report. This can, however, make the problems more computationally demanding.

3.3 The MacMPEC Collection

The MacMPEC collection is a library which contains several MPCC test problems written in AMPEL by Sven Leyffer [13]. The library also provides the optimal objective value, or the best solution found for each of the problems. In this particular project, these MPCCs worked as benchmarks for testing the performance of the different approaches. The number of variables, constraint functions and complementarity pairs varied for each problem, as well as the extent of non-linearity and convexity. The selection was done in such a way that a great diversity of problems was obtained to test not only the performance but also the robustness of each reformulation. A total of 64 problems were chosen for this project and are listed in Appendix A. It is relevant to note that most of the problems implemented can be categorised as small-scale problems.

3.4 Implementation and Data Processing

An algorithm was written for each of the approaches which reformulated the MPCCs into the desired form. The algorithm for PF(ρ) dynamic, see Algorithm 1, was used based on an existing implementation [11]. The other algorithms were based on this implementation as well and can be found in Appendix B. Each of the individual problems was manually implemented by translating the problems from the library into the modelling language JuMP [5], which is compatible with the solver IPOPT [9] in Julia. These problems were further used in the process of evaluating the different approaches. Results such as the number of iterations, CPU time, complementarity satisfaction and accuracy, were stored for each individual problem for each approach. These results were further visualised by creating figures with the PyPlot package. How each of the results was processed will be further presented.

3.4.1 The Performance Profiles

To compare the performance of the different reformulation strategies, performance profiles were created based on a metric from an existing paper [2]. Firstly, the performance ratio for each solver on each particular problem was calculated as follows,

$$\eta_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} \ : \ 1 \leq s \leq n_s\}}, \tag{3.7}$$

where p is the problem and s is the solver. $t_{p,s}$ is the time the solver s used to solve the particular problem p. If the solver was unable to solve the problem, $t_{p,s}$ was set to a large number representing infinity. This was also done if the approach was only able to solve the problem with reduced accuracy, which is the case if the problem is "almost locally solved" or if the solution is returned with a numerical error. Further, the performance profiles were created by plotting the following equation,

$$p_s(\tau) = \frac{1}{n_p}\text{size}\{p \in P \ : \ \eta_{p,s} \leq \tau\}, \tag{3.8}$$

where P is the set containing all the problems and $p_s(\tau)$ gives the percentage of problems that were solved in less than a time factor, τ, times the best performance [2].

3.4.2 Percentage Error

As the magnitude of each problem varies, percentage error was used to compare the accuracy of each method. Percentage error, δ, is given by,

$$\delta = \left| \frac{\upsilon - \upsilon_E}{\upsilon_E} \right| \cdot 100, \tag{3.9}$$

where υ is the objective function value found by the optimiser and υ_E is the true accurate function value provided by the MacMPEC library. However, this formula is not defined when $\upsilon_E = 0$, for this particular exception, the formula was adjusted as follows,

$$\delta = \left| \frac{\upsilon - \upsilon_E}{1 + \upsilon_E} \right| \cdot 100. \tag{3.10}$$

The result was further used to make a bar plot, which shows in which error range the majority of the problems lie.

3.4.3 Complementarity Satisfaction

In the examination of the satisfaction of the complementarity constraints, the infinity norm,

$$||x||_\infty = \max(|x_1|, |x_2|, ..., |x_m|) = \max_{i=1}^{m} |x_i|, \tag{3.11}$$

has been used. In this expression, $x \in \mathbb{C}^m$ is a vector, and x_i is an element with index i. The complementarity variable that obtained the lowest value in each pair was stored in a shared vector for this particular problem. Further, the infinity norm was used on this specific vector, and simply returned the magnitude of the largest entry in the vector. This can be used to measure the size of the vector, and to further make conclusions about the satisfaction of the complementarity constraints [6]. Using this norm, rather than calculating $x^T y$, is beneficial as the scaling and the number of variables have less impact on the measurement [10].

3.4.4 Initial Guesses

IPOPT sets the default starting value for all variables as zero, or the nearest value from zero if zero is not in the feasible set. However, zero is not always found to be the best initial guess. For instance, this can be problematic for non-linearities that are not defined for zero, like $1/x$. As earlier mentioned, in the evaluation process of the different solution strategies, the initial guesses were either the values provided by the library or chosen to be zero. Consequently, the values did not vary between the approaches when they were compared. However, this was studied separately in the project by using a trial-and-error approach to see how it affected the results.

Chapter 4

Results and Discussions

This chapter covers the results from the comparison of the solutions strategies, as well as remarks on these results. Properties such as performance, robustness, the satisfaction of complementarity constraints, accuracy and the number of iterations are taken into account. Furthermore, this chapter addresses the effect of changing the tuning parameters, the relaxation or the penalisation parameter. Lastly, the influence of initial guesses is looked into as well.

4.1 Comparison of Performance

The different approaches were compared in terms of performance by generating a performance plot, see Figure 4.1. This was accomplished by following the procedure presented in Section 3.4.1.
However, some additional assumptions were taken in the creation of this plot. As the MPCCs have an inherent nonconvexity, multiple local solutions were possible. However, among the selected problems, only a few of them yielded different objective function values for the different approaches. This was the case for the *Bilevel1* problem. The penalty approaches managed to converge to a better solution, which was also the solution listed in the MacMPEC library, than the relaxation approaches. However, for the *Ex.9.2.3* problem none of the approaches were able to reach the optimal solution, but a feasible local solution was found by all of them. Lastly, the maximisation problem *Bilin* gave different solutions for each of the different approaches, but only RegComp($\epsilon = 10^{-6}$) was able to find the same solution provided by the library. As these problems were found to have a minor impact on the overall result of the performance, they were included in the creation of the plot given in Figure 4.1. It is additionally important to emphasise that in this particular figure, a solution was defined as the solution of the reformulation, even though it may be infeasible for the original MPCC. This type of issue will, however, be further discussed later. Another thing to keep in mind is that the CPU time fluctuates significantly for each run. This will result in small variations in the performance. However, the overall result and the conclusions drawn from it will be the same.

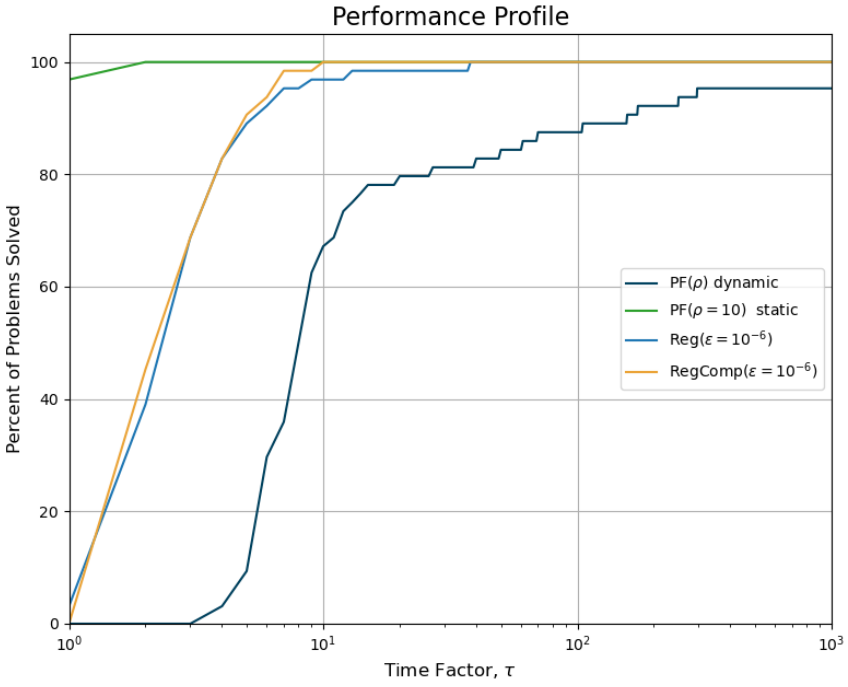

Figure 4.1: The performance profile of the different reformulations. Convergence to feasible solutions of the reformulations are accounted as solutions

Figure 4.1 compared the relative solving time and the percent of problems solved for each of the different approaches, represented by different colours. The dark blue line represents the PF(ρ) dynamic approach and is clearly the slowest among the reformulations. Additionally, this method was unable to solve all the problems but managed to solve 95.3 percent of them. This was expected as the algorithm needs to be provided with suitable initial values for the parameters, as presented in Algorithm 1 in Section 3.1.3. Due to the variety in magnitude and complexity of the different problems, it may be challenging to locate standard initial values for the parameters that are suitable in every case. The other penalty approach, illustrated in green, was clearly shown to be the fastest among the approaches. The approach was also able to solve all of the problems. Lastly, the relaxation approaches, Reg($\epsilon = 10^{-6}$) and RegComp($\epsilon = 10^{-6}$), performed quite similarly but were significantly slower than the penalisation approaches. However, likewise as the static penalisation approach, the methods were able to solve all problems. Figure 4.1 gives the impression that all the methods are robust, as they were able to solve the majority of the problems. However, most of the problems tested were quite small considering the number of variables, complementarity constraints and constraints in general. A greater difference between the approaches might have arisen if more complex problems were implemented.

4.1.1 Feasible Solutions for the MPCC

In figure 4.1, a problem was defined as solved if the solution lay within the feasible region of the reformulation. However, this does not imply that the solution is feasible for the original problem as well. IPOPT is not able to detect if the complementarity constraints are violated, as the constraints of the reformulation can be fulfilled regardless. This is taken into account in Figure 4.2, as valid solutions are those that are feasible for both the reformulation and the original problem as well. However, it can be challenging to interpret if a solution is defined as feasible or not. For problem *Df1*, for example, the dynamic penalisation approach converged to a solution where both the expressions in the complementarity pair were equal to 10^{-5}. In some cases, this may be accurate enough to conclude that both are zero, and the complementarity constraint is consequently satisfied. However, in other cases, this may be insufficient. A specific threshold should be chosen, where the order of magnitude depends on the field of application for the MPCC. In this particular project, the threshold was chosen to be 10^{-6}, which makes the solution of the dynamic reformulation of *Df1* infeasible.

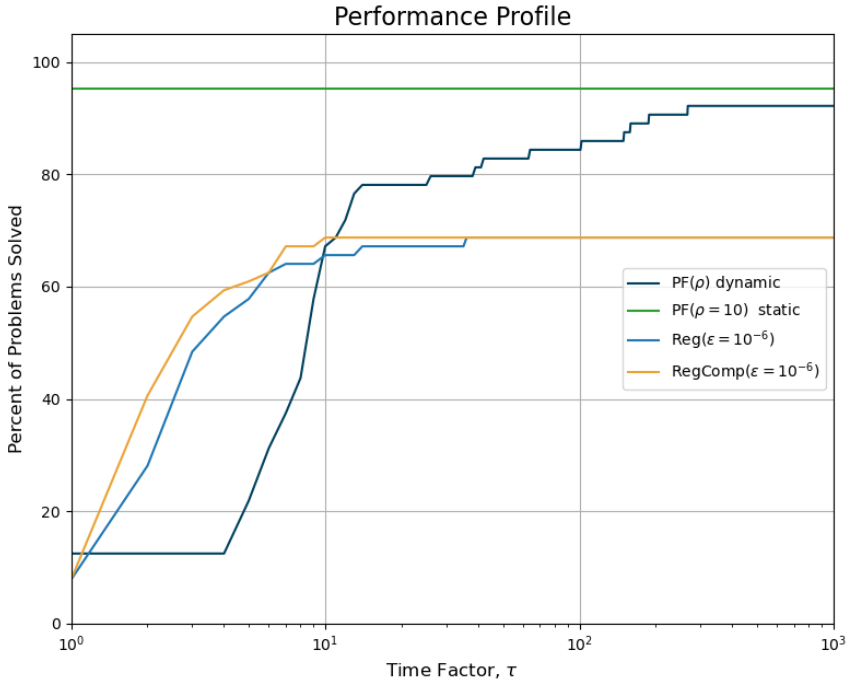

Figure 4.2: The performance profile of the different reformulations. Only feasible solutions of the MPCCs were accounted as solved.

Figure 4.2 was compared with Figure 4.1 in order to evaluate the approaches' ability to find feasible solutions for the original problems, the MPCCs. This change significantly reduced the number of problems solved by the relaxation approaches. Only 68.8 percent of the problems were solved, such that the complementarity constraints were sufficiently satisfied. The static PF($\rho = 10$) approach was on the contrary, able to solve approximately 95.3 per cent. As this is only a minor decrease, only

a few problems violated the complementarity constraints. Likewise for the dynamic penalization approach, the percentage of problems solved decreased slightly to 92.2 percent. With these observations in mind, it is clear that the relaxation approaches are more at risk of giving convergence to infeasible solutions for the MPCCs than the penalisation approaches. However, if the threshold was decreased, and consequently more solutions were accounted as feasible, the performance of the relaxation approaches could be improved.

4.1.2 Changing the Penalisation Parameter

The penalisation parameter in the PF(ρ) static approach, was selected to $\rho = 10$ for each problem. However, this penalisation value was not consistently the ideal tuning value in practice. This was the case for the minimisation problem *Ex.9.1.6*, which has the optimum solution resulting in $f(x) = -49.0$. The penalisation reformulation detected a local solution, such that $f(x) = -15$ with $\rho = 10$, but by increasing the penalisation parameter significantly, to $\rho = 1000$, the algorithm managed to converge to a solution corresponding to $f(x) = -49.0$. A wrongly tuned penalty parameter can also lead to the original problem becoming infeasible. This was the case for the problem *Scale5*. By using $\rho = 10$ the problem converged to a point that resulted in an objective value equal to 0.45, which is in fact better than the expected value provided by the library. However, the solution was found to be infeasible as the complementarity variables were $x = 0.95$ and $y = 0.95$ and therefore violated the complementarity constraint. This problem could, however, reach convergence to a feasible solution by increasing the penalisation parameter to $\rho = 1000$. This specific result was also found for the *Scale1* problem, as the problem became infeasible for a $\rho = 10$.

In most cases, it is difficult to predict the ideal penalisation problem as it depends on the scaling of the problem. In some cases, even $\rho = 1$ can be proven to be the best-suited value. As demonstrated, finding a suitable tuning parameter can be challenging in the static penalisation method, and can thus be seen as a disadvantage of this particular approach.

4.1.3 Changing the Relaxation Parameter

Additionally, an experiment was performed where the relaxation parameter in the Reg(ϵ) approach was varied, as illustrated in Figure 4.3, following the same procedure as in the comparison of the overall performance. In this particular comparison all the solutions, also the ones violating the complementarity constraints, are included.

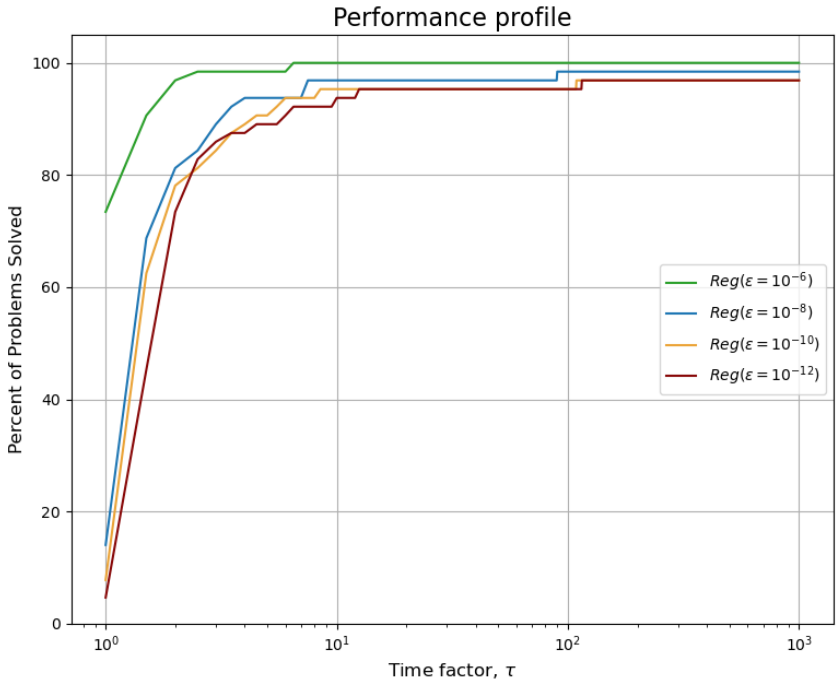

Figure 4.3: The performance profile of Reg(ϵ) with different relaxation values.

The performance profile shows that by decreasing the relaxation values, thus tightening the constraints, the solving time increases as well. The reformulation with the largest $\epsilon = 10^{-6}$, represented by the green line, required the shortest solving time and had the ability to solve all the problems. However, by decreasing the relaxation parameter, the solving time increased. The approach with $\epsilon = 10^{-12}$, corresponding to the red line, was the slowest among the approaches compared. This is expected as a small relaxation value, tightens the complementarity constraints and therefore reduces the feasible regions. Both Reg($\epsilon = 10^{-12}$) and Reg($\epsilon = 10^{-10}$) were able to solve 96.7 percent of the problems, which is only slightly less than the Reg($\epsilon = 10^{-6}$). Lastly, Reg($\epsilon = 10^{-8}$) solved 98.4 percent of the problems and was faster than both Reg($\epsilon = 10^{-10}$) and Reg($\epsilon = 10^{-12}$), but slower than Reg($\epsilon = 10^{-6}$). According to the result, the tuning of the relaxation parameters affects performance. However, the magnitude of the effect was minor. This is reasonable as the problems implemented were relatively simple. It might appear that for large-scale problems containing several complementarity constraints, this would have had a greater impact.

4.2 Satisfaction of the Complementarity Constraints

Complementarity satisfaction has been calculated and compared using the infinity norm, as described in Section 3.4.3. The result is presented in Figure 4.4. The only problems included are the ones IPOPT was able to solve. In this particular comparison, all the solutions are included regardless if they are feasible solutions of the original MPCCs.

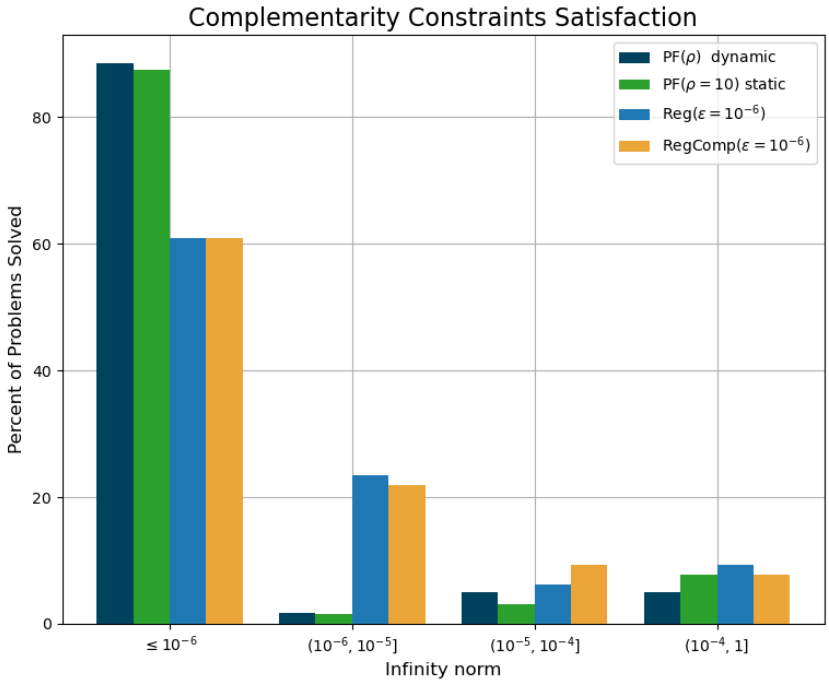

Figure 4.4: Satisfaction of the complementarity constraints. All feasible solutions of the reformulations are included.

Figure 4.4 shows the percent of problems solved within a certain complementarity constraint satisfaction. Based on the figure, it can be stated that penalisation approaches were the ones satisfying the complementarity constraints to the greatest extent. The dynamic penalisation approach, PF(ρ), managed to solve 88.5 percent of the solved problem with $||\min\{x, y\}||_\infty \leq 10^{-6}$, where x and y denote the complementarity expressions. The static penalisation approach, PF($\rho = 10$), was similar, with 87.5 percent of problems solved with $||\min\{x, y\}||_\infty \leq 10^{-6}$. The relaxation approaches, however, perform relatively poorly when it comes to complementarity satisfaction. Even though the relaxation value was tuned for $\epsilon = 10^{-6}$, only 60.9 percent of them were able to achieve an infinity norm less or equal to 10^{-6}. The 39.1 percent above this limit is caused by the relaxation constraint only demanding that the product of the complementarity variables to be below 10^{-6}. For instance, in problem $Ex9.2.6$ solved by the Reg($\epsilon = 10^{-6}$) approach, both the expressions in one of the complementarity pairs achieved the value 0.00707. This satisfied the constraint of the relaxation

formulation as, $0.00707 \cdot 0.00707 \leq 10^{-6}$, but violated the complementarity constraint as non of the expressions are truly zero. The solution was feasible for the reformulation but was found infeasible for the original problem. This was particularly found as a problem for the relaxation formulations and is accordingly a disadvantage of the methods. This issue also occurred with the penalisation methods, but as the figure illustrates, to a significantly less extent.

4.3 Comparison of Accuracy

The performance plots, presented in the previous sections, did not present information about the accuracy of the solutions. This information is crucial as the overall performance will be poor if the accuracy is insufficient, regardless of the speed of the approach. This property was investigated and compared by calculating the percentage error, as explained in Section 3.4.2. Based on the calculations, Figure 4.5 was created, which presents the percentage of problems solved within a certain error range. In this case, the problems included are the ones the approach had the ability to solve, regardless of whether it was found feasible for the MPCC or not.

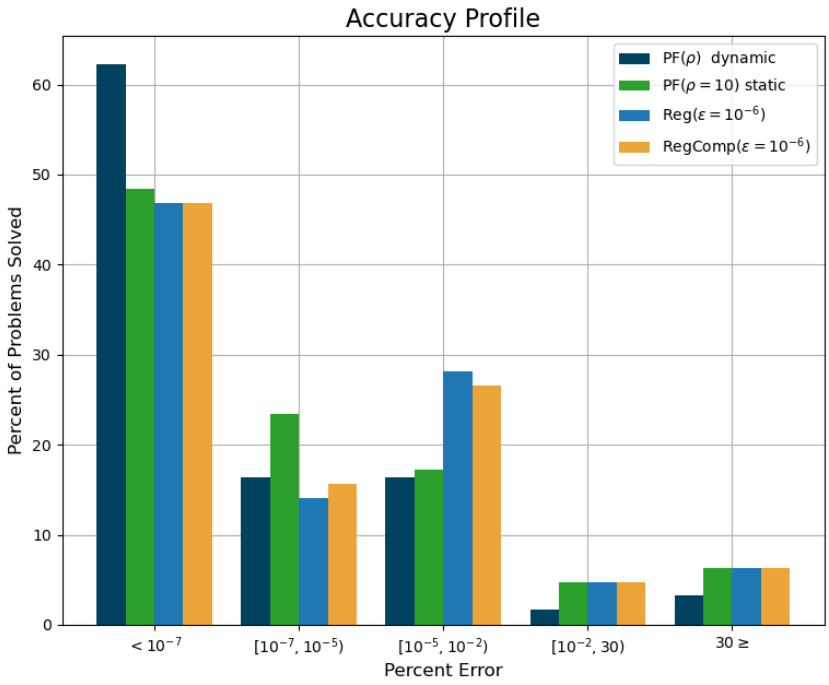

Figure 4.5: The percentage of problems solved within a certain error range.

According to Figure 4.5, the dynamic PF(ρ) approach, as given in dark blue, was the most accurate, where 62.2 percent of the problems had an error below 10^{-7} percent. Additionally, 95 percent of the problems were solved with an error less than 10^{-2} percent. The problems laying on the left of the figure, with the highest error percentage, are mostly the problems that were unable to converge

to the solution provided by the library, but instead detected local solutions. The static PF($\rho = 10$) was only able to solve 48.4 percent of the problems with an error less than 10^{-7} percent, which is significantly less than the dynamic penalisation approach. However, this approach was able to solve 89.1 percent of the problems below an error of 10^{-2} percent, which is not that far from the dynamic approach. Both the relaxation approaches, seem to be quite similar to the static penalisation approach when it comes to accuracy, with 46.9 percent of the problems solved with an error less than 10^{-7} percent, and 89.1 percent solved with an error below 10^{-2} percent.

It needs to be emphasised that improved accuracy for the relaxation approaches, and the static approach as well, could be achieved if the approaches were differently tuned. It has been confirmed that the value of ρ has a great effect on the result. Adjusting the value of ρ for each problem might have resulted in increased accuracy. The same applies to the relaxation approaches. However, manually tuning the parameters has the disadvantage of being quite time-consuming. The importance of the grade of accuracy is again dependent on the application of the MPCC. However, all the approaches seem to be relatively accurate, but the dynamic penalisation approach certainly stands out as the most accurate among the approaches.

4.4 Comparison of Iterations

The number of iterations was compared by creating Figure 4.6, which shows the percentage of problems solved within a certain number of iterations.

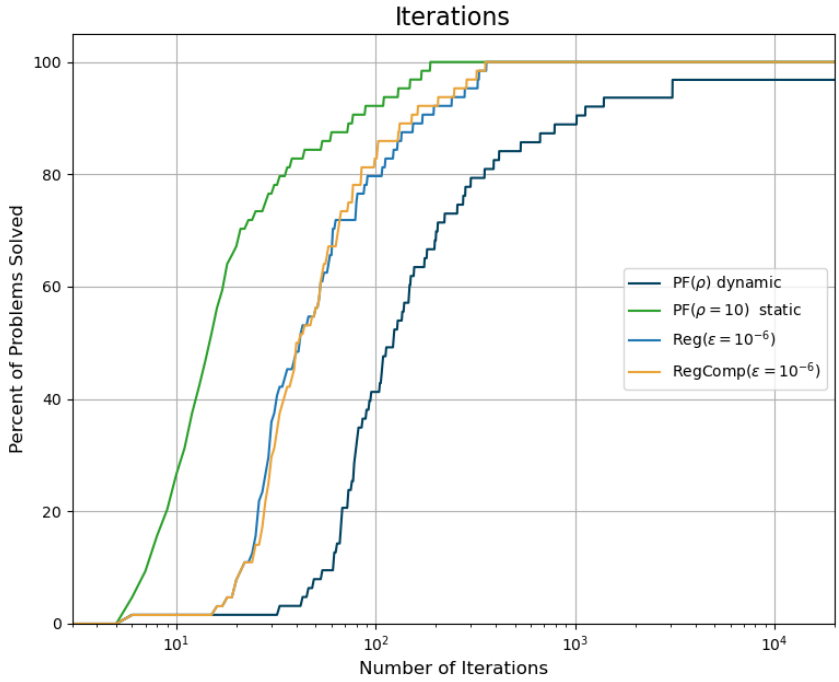

Figure 4.6: The percentage of problems solved within a certain number of iterations for each approach.

The dynamic penalisation approach, represented by the blue dark line, required significantly more iterations than the other approaches. This was expected and is caused by the outer barrier loop, see Section 3.1.3. In contrast to other approaches, that only require one optimisation, IPOPT solves the problem in every barrier iteration. IPOPT does not accept initialisation of the multipliers and is therefore not utilising the result from the previous iterations. This explains why IPOPT required more iterations than expected to solve the problems. As anticipated, this seems to have a large impact on the solving time, as the dynamic penalisation approach was the slowest of them all. The relaxation approaches performed quite similarly when it comes to iterations, and required approximately the same numbers. The approach which requires the least iterations is the static PF($\rho = 10$) approach, represented in green. This also corresponds to the expectations, as this approach was the fastest. It was, therefore, as anticipated, a correlation between the number of iterations and the solving time, such that more iterations resulted in longer solving time.

4.5 Changing the Initial Values

In the comparison process of the approaches, the initial values were provided by the MacMPEC library or set to the default value of zero. In reality, these values were not consistently the most suitable initial guesses. Based on the experience gained from this study, the appropriate initial values were found to depend on both the specific problem and the applied solution strategy as well.

The minimisation problem *Design-cent-1*, for instance, was provided with initial values for all 12 variables. These initial guesses resulted in convergence to the optimum for all the approaches. Despite that, an attempt was made to use the default initial values instead to investigate how this would affect the result. This change led to the limit of iterations being reached, such that none of the approaches were able to converge. In this particular problem, the importance of a good initial guess was demonstrated, as a poor guess might lead to algorithmic failure.

The maximisation problem *Bilin* was found sensitive to the choice of initial values as well. For this problem, however, the algorithm managed to converge to a solution despite changes in the initial guesses. *Bilin* was provided with the starting values $x_i^0 = 1$ and $y_i^0 = 1$, where i is in the index for variables in vectors x and y. However, with these initial guesses, only RegComp($\epsilon = 10^{-6}$) was able to converge to the optimum provided by the library, such that the objective function was $f(x, y) = 18.4$. The other approaches detected inferior local solutions instead. For this specific initial guess, the static penalisation approach, PF($\rho = 10$) converged to a solution resulting in $f(x, y) = 13$. By increasing the initial guesses to $x_i^0 = 10$ and $y_i^0 = 10$, the algorithm managed to find the optimum corresponding to $f(x, y) = 18.4$. This initial guess, however, was not the most beneficial guess for Reg($\epsilon = 10^{-6}$), as the objective function value was $f(x, y) = 16$, which is slightly below the optimum. Several initial values were further tested. Among those was the default starting value of IPOPT, $x_i^0, y_i^0 = 0$. Unfortunately, this resulted in an even lower objective function value, $f(x, y) = 5.6$. Hence, it can be confirmed that the default value is not always the best-fitted guess. After several trials, the most appropriate initial guess for this solving strategy on this particular problem was found to be $x_i^0 = 0.1$ and $y_i^0 = 1$. On the contrary, this guess was found unsuited for the PF($\rho = 10$) static approach. Based on these observations, it can be concluded that it might be challenging to detect suitable initial guesses for MPCCs, as it depends on the problem formulation and the solution strategy as well.

It was also observed that a good initial guess could reduce the number of iterations required. This was the case for *Bilevel2m* with Reg($\epsilon = 10^{-6}$). By improving the initial guess, the number of iterations was reduced from 58 to 50 iterations. However, this change in initial guess was found less appropriate for the PF($\rho = 10$) approach as the number of iterations increased from 25 to 38. Again, this reinforces the claim that the suitable initial guesses vary from one approach to another.

The result above shows that one specific initial value can be appropriate for one approach, but unsuitable for another. As follows the initial guesses given in this project could have affected the performance, robustness and complementarity satisfaction of the different methods. By changing the initial guesses from one approach to another, the result could have differed. Additionally, as the problem *Bilin* emphasised, the initial guess is highly crucial in which local solution is being detected. This is a known challenge related to non-convexity, as different initial guesses may give different local solutions.

Chapter 5

Final remarks

In this project, both the relaxation and the penalisation approaches were able to solve the MPCCs to a certain extent. However, a closer examination and comparison revealed that each method had its advantages and disadvantages.

Both the static and dynamic penalisation approaches were remarkably promising in this study. In terms of performance, the static penalisation approach was shown to be the most efficient, as the solving time was the shortest and the number of iterations required was the least. It was further confirmed that the majority of the detected solutions for this approach, fulfilled the complementarity constraint to an acceptable level. In other words, the solutions located by this approach were indeed feasible for the original MPCC as well. The dynamic penalisation approach was, similar to the static one, able to solve the magnitude of the problems such that the complementarity constraints were satisfied. However, it required significantly longer solving time and a great number of iterations. Despite this, the method was found to be particularly accurate and hence outperformed the static approach in this area. This method also has the advantage of adjusting the penalisation parameter automatically. The static penalisation method, on the contrary, required the penalisation parameter to be manually tuned, which was shown to be challenging.

The relaxation approaches performed relatively poorly, apart from being more efficient than the dynamic penalisation approach. Even though they were able to solve all the problems, many solutions were accounted as infeasible due to violation of the complementarity constraints. The approaches were also relatively slow compared to the static penalisation approach, but the level of accuracy was comparable. However, the relaxation parameter was significantly easier to tune than the penalisation parameter, since there was a clear correlation between the magnitude of the parameter and the performance.

To conclude, the most appropriate approach will depend on the purpose and the field of application, as different approaches offer different advantages. Some situations call for a fast method with simply tuned parameters, making the relaxation method the proper choice. However, if accuracy is crucial, regardless of time, the dynamic penalisation method is found favourable. A common feature of all the methods was their high sensitivity to the choice of initial values. Poor initial guesses were shown to give convergence to other local solutions, or in the worst case, result in algorithmic failure.

As for further research, the penalisation approaches should be the main focus. A better implementation of the dynamic penalisation method is needed in order to make a more definite assessment of its performance. If the method is still found to be deficient, it may be necessary to look at other strategies for updating the penalisation parameter. It may also be expedient to try other solvers, such as SQP which is based on an active set method rather than an interior point method. Furthermore, a greater number of problems, as well as more complex large-scale problems should be implemented, as it is desired to draw conclusions from a large variety of problems. Lastly, the choice of initial values should be emphasised, as it was shown to have a major impact on the overall performance.

Bibliography

[1] Jean-Pierre Dussault, Mounir Haddou, Abdeslam Kadrani, and Tangi Migot. How to compute a m-stationary point of the mpcc. 2017.

[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] Roberto Andreani, Joaquim João Júdice, José Mario Martínez, and Tiara Martini. Feasibility problems with complementarity constraints. *European Journal of Operational Research*, 249 (1):41–54, 2016.

[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] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017. doi: 10.1137/15M1020575.

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

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

[8] Roger Fletcher* and Sven Leyffer. Solving mathematical programs with complementarity constraints as nonlinear programs. *Optimization Methods and Software*, 19(1):15–40, 2004.

[9] 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.

[10] 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.

[11] Caroline Nakama. Julia MPCC library. https://github.com/carolinesnakama/MPCCLibrary, 2020. Accessed: 12.08.22.

[12] Yoshiaki Kawajir, François Margot, Stefan Vigerske, and Andreas Wächter. Ipopt documentation. URL `https://coin-or.github.io/Ipopt/index.html`.

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

Appendices

Appendix A

Problems from MacMPEC Collection

Problem	Optimal Solution
Bard1	17.0000
Bard2	17.0000
Bard3	6598
Bard3m	-12.6787
Bilevel1	0.0.
Bilevel2	-6600.00
Bilevel3	-6600.00
Bilin	-12.6787
Design-cent-1	1.86065
Design-cent-2	3.48382
Design-cent-3	3.72337
Design-cent-4	3.0792
Design-cent-21	3.48382
Design-cent-31	3.72337
Desilva	-1.0
Df1	0.0
Ex9.1.1	-13.00
Ex9.1.3	29.2

Ex9.1.4	-37.0
Ex9.1.5	-1.0
Ex9.1.6	-49.0
Ex9.1.7	-26.0
Ex9.1.9	3.11111
Ex9.2.2	100.0
Ex9.2.3	-55
Ex9.2.4	0.5
Ex9.2.6	-1.0
Ex9.2.7	17.0
Ex9.2.8	1.5
Ex9.2.9	2.0
Flp2	0.0
Gauvin	20.0
Gnash10	-230.823
Gnash11	-129.823
Gnash12	-36.9331
Gnash13	-7.06178
Gnash14	-0.179046
Gnash15	-354.699
Gnash16	-241.442
Gnash17	-90.7591
Gnash18	-25.6982
Gnash19	-6.11571
Gnashm10	-230.823
Gnashm11	-129.823
Gnashm12	-36.9331
Gnashm13	-7.06178
Gnashm14	-0.179046

Jr1	0.5
Jr2	0.5
Kth1	0
Kth2	0
Kth3	0.5
Outrata31	3.20077
Outrata32	3.4494
Outrata33	4.60425
Outrata34	6.59268
Ralph2	0.0
Scale1	1.0
Scale5	100.0
Scholtes1	2.0
Scholtes3	0.5
Scholtes4	$-3.07446 \cdot 10^{-7}$

Table 1: The problems selected for this project from the MacMPEC Collection. [13]

Appendix B

The Approaches Implemented in Julia

B.1 The Relaxation Methods

```
#
# Implementation of the Relaxation Approach
# For Mathematical Programs with Complementarity Constraints
#
# Ann Iren Fossoey and Caroline S. M. Nakama
# Autumn 2022
#
# Last update: 15.12.2022
#

"""
        nonLinear(vec)

Returns ture if the array contains a nonlinear or quadratic expression,
such that the expression becomes nonlinear;
`vec` is a array containing the expressions;
"""

function nonLinear(vec)
    return any(x -> isa(x, Union{QuadExpr, NonlinearExpression}),vec);
end

"""
        complementaritySatisfaction(nc,nvc,cval)

Takes the infity norm of min_val which is a vector containing
the smalles value in each complementarity pair.
`nc` is a integer containing the length of the compl. vector.
`nvc` is an array containing the length of each individual pair.
"""

function complementaritySatisfaction(nc,nvc,cvar)
    min_val = zeros(sum(nvc),1);
```

```
    cval = Vector{Tuple{Any,Any}}(undef, nc);
    prodCval = []
    complStatus = [true];
    for i in 1:nc
        # return the current value stored in the nonlinear parameter.
        cval[i] = (value.(cvar[i][1]), value.(cvar[i][2]));
        # the one in the pair which has the smallest value.
        min_val[nvc[i]:nvc[i+1]-1] .= min.(cval[i][1], cval[i][2]);
        len = length(cval[i][1])
        if len == 1 && !isa(cvar[i][1], Vector)
                push!(prodCval,cval[i][1] * cval[i][2]);
        else
            for j in 1:len
                push!(prodCval,cval[i][1][j]*cval[i][2][j]);
            end
         end
    end
    compl = norm(min_val, Inf);
    for i in prodCval
        if i > 10^(-6) + 10^(-16)
            complStatus[1] = false;
        end
    end
    return compl,cval,prodCval,complStatus[1];
end

"""
        reg_solve!(model, comp, epsilon, objective = :obj)

Solves an MPCC problem defined in JuMP using the Ipopt solver.
Adds each complementarity pair as an inividual constraint.

`model` is a JuMP model initialized Ipopt Optimizer
and containing the MPCC problem;
`comp` is an array of pairs with the complementarity variables;
`epsilon` is the relaxation parameter;
`objective` is a symbol corresponding to the name of the
JuMP expression that defines the objective function;

"""
function reg_solve!(model, comp, epsilon, objective = :obj)

    nc = length(comp);
    cvar = Vector{Tuple{Any,Any}}(undef, nc);
    clen = Array{Int64,1}(undef, nc);
    nvc = Array{Int64,1}(undef, nc + 1);
    nvc[1] = 1;

    # Check if one of the element in the pair is a matrix, if so flatten it
    for i in 1:nc
        if isa(comp[i][1],Matrix)||any(x -> isa(x, NonlinearExpression),comp[i])
            cvar[i] = (collect(Iterators.flatten(comp[i][1])),
            collect(Iterators.flatten(comp[i][2])));
        else
            cvar[i] = (comp[i][1], comp[i][2]);
        end
```

```
        if isa(cvar[i][1], VariableRef)
            nvar = 1;
        else
            nvar = length(cvar[i][1]);
        end

        clen[i] = nvar;
        nvc[i + 1] = nvar + nvc[i];
    end

    for i in 1:nc
        if any(x -> isa(x, Union{QuadExpr, NonlinearExpression}), cvar[i])
                @NLconstraint(model, cvar[i][1] * cvar[i][2] <= epsilon);
        elseif any(x -> isa(x, Union{VariableRef, AffExpr}), cvar[i])
                @constraint(model, cvar[i][1] * cvar[i][2]  <= epsilon);
        else
            for j in 1:clen[i]
                tmp = [cvar[i][1][j], cvar[i][2][j]];
                if nonLinear(tmp)
                    @NLconstraint(model, tmp[1] * tmp[2] <= epsilon);
                else
                    @constraint(model, tmp[1] * tmp[2] <= epsilon);
                end
            end
        end
    end

    obj = getindex(model, objective);
    if isa(obj, NonlinearExpression)
        @NLobjective(model, Min, obj);
    else
        @objective(model, Min, obj);
    end

    # Get number of iterations
    iters = [0]
    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)
        iters[1] = iter_count
        return true
    end
    MOI.set(model, Ipopt.CallbackFunction(), my_callback);

    # Solving the optimization problem
    optimize!(model);
    setIter_til_utskrift(iters[1]);
```

```
    # Check of statisfied the complementarities are
    compl,cval,prodCval, complStatus = complementaritySatisfaction(nc,nvc,cvar);
    setComp_til_utskrift(compl);
    SetCompSatisfaction(complStatus);

    println("status = ", termination_status(model));
    println("objective value = ", value(obj));
    println("Solve time = ", solve_time(model));
    println("MPCC complementarity = ", compl);
    println("MPCC satisfied = ",complStatus);
    println("IPOPT iterations = ", getIter_til_utskrift());




"""
        regComp_solve!(model, comp, epsilon, objective = :obj)

Solves an MPCC problem defined in JuMP using the Ipopt solver.
Elementswise multiplication of complementarity constraint as one constraint.

`model` is a JuMP model initialized Ipopt Optimizer
and containing the MPCC problem;
`comp` is an array of pairs with the complementarity variables;
`epsilon` is the relaxation parameter;
`objective` is a symbol corresponding to the name of the JuMP
 expression that defines the objective function.

"""

function regComp_solve!(model, comp, epsilon, objective = :obj)
    # getting model variables
    nc = length(comp);
    cvar = Vector{Tuple{Any,Any}}(undef, nc);  # Vector containing the comp.cons
    clen = Array{Int64,1}(undef, nc);
    nvc = Array{Int64,1}(undef, nc + 1);
    nvc[1] = 1;

    for i in 1:nc
        if isa(comp[i][1],Matrix)||any(x -> isa(x, NonlinearExpression),comp[i])
            cvar[i] = (collect(Iterators.flatten(comp[i][1])),
            collect(Iterators.flatten(comp[i][2])));
        else
            cvar[i] = (comp[i][1], comp[i][2]);
        end

        if isa(cvar[i][1], VariableRef)
            nvar = 1;
        else
            nvar = length(cvar[i][1]);
        end

        clen[i] = nvar;
        nvc[i + 1] = nvar + nvc[i];
    end

    # Check if vector or singel variable/expression
```

```
    for i in 1:nc
        if any(x -> isa(x, Union{QuadExpr, NonlinearExpression}), cvar[i])
                @NLconstraint(model, cvar[i][1] * cvar[i][2] <= epsilon);
        elseif any(x -> isa(x, Union{VariableRef, AffExpr}), cvar[i])
                @constraint(model, cvar[i][1] * cvar[i][2] <= epsilon);
        else
            tmp1 = cvar[i][1];
            tmp2 = cvar[i][2];
            if any(x -> isa(x, Union{Vector{QuadExpr},
                Vector{NonlinearExpression}}),cvar[i])
                @NLconstraint(model, sum(tmp1[j]*tmp2[j]
                    for j in 1:clen[i]) <= epsilon);
            else
                @constraint(model, sum(tmp1[j]*tmp2[j]
                    for j in 1:clen[i]) <= epsilon);
            end
        end
    end

    obj = getindex(model, objective);
    if isa(obj,NonlinearExpression)
        @NLobjective(model, Min, obj);
    else
        @objective(model, Min, obj);
    end

    # Get number of iterations
    iters = [0]
    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)
        iters[1] = iter_count
        return true
    end
    MOI.set(model, Ipopt.CallbackFunction(), my_callback);


    # solving the optimization problem
    optimize!(model);
    setIter_til_utskrift(iters[1]);

    # check of statisfied the complementarities are
    compl,cval, prodCval, complStatus= complementaritySatisfaction(nc,nvc,cvar);
    setComp_til_utskrift(compl);
    SetCompSatisfaction(complStatus);

    println("status = ", termination_status(model));
    println("objective value = ", value(obj));
```

```
    println("Solve time = ", solve_time(model));
    println("MPCC complementarity = ", compl);
    println("MPCC satisfied = ",complStatus);
    println("IPOPT iterations = ", getIter_til_utskrift());
end
```

B.2 The Static Penalisation Method

```
#
# Implementation of the Penalisation Approach
# For Mathematical Programs with Complementarity Constraints
#
# Ann Iren Fossoey and Caroline S. M. Nakama
# Autumn 2022
#
# Last update: 15.12.2022
#

"""
        PF_solve! (model, comp, rho, objective = :obj)

Solves an MPCC problem defined in JuMP using the Ipopt solver.

`model` is a JuMP model initialized Ipopt Optimizer and
containing the MPCC problem;
`comp` is an array of pairs with the complementarity variables;
`rho` is the penalization parameter;
`objective` is a symbol corresponding to the name of the JuMP expression
that defines the objective function;


"""
function PF_solve!(model, comp, rho, objective = :obj)
    nc = length(comp);
    cvar = Vector{Tuple{Any,Any}}(undef, nc);
    cval = copy(cvar);
    clen = Array{Int64,1}(undef, nc);
    nvc = Array{Int64,1}(undef, nc + 1);
    nvc[1] = 1;

    # Check if one of the element in the pair is a matrix, if so flatten it
    for i in 1:nc
        if isa(comp[i][1], Matrix)||any(x -> isa(x, NonlinearExpression),comp[i])
            cvar[i] = (collect(Iterators.flatten(comp[i][1])),
            collect(Iterators.flatten(comp[i][2])));
        else
            cvar[i] = (comp[i][1], comp[i][2]);
        end

        if isa(cvar[i][1], VariableRef)
            nvar = 1;
```

```
        else
            nvar = length(cvar[i][1]);
        end

        clen[i] = nvar
        nvc[i + 1] =  nvar + nvc[i];
    end

    pen = Array{Any, 1}(undef, nc)

    for i in 1:nc
        if any(x -> isa(x, Union{QuadExpr, NonlinearExpression}), cvar[i])
            pen[i] = @NLexpression(model, cvar[i][1] * cvar[i][2]);
        elseif any(x -> isa(x, Union{VariableRef, AffExpr}), cvar[i])
            pen[i] = @expression(model, cvar[i][1] * cvar[i][2]);
        else
            if any(x -> isa(x, Union{Vector{QuadExpr},
                Vector{NonlinearExpression}}), cvar[i])
                pen[i] = @NLexpression(model, sum(cvar[i][1][j] * cvar[i][2][j]
                for j in 1:clen[i]));
            else
                pen[i] = @expression(model, sum(cvar[i][1][j] * cvar[i][2][j]
                 for j in 1:clen[i]));
            end
        end
    end


    obj = getindex(model, objective);

    # Adding the penalization term to the objective function
    if isa(obj,NonlinearExpression)||any(x -> typeof(x)==NonlinearExpression,pen)
        @NLobjective(model, Min, obj + rho * sum(pen[i] for i in 1:nc));
    else
        @objective(model, Min, obj + rho * sum(pen[i] for i in 1:nc));
    end

    # Get number of iterations
    iters = [0]
    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)
        iters[1] = iter_count
        return true
    end
    MOI.set(model, Ipopt.CallbackFunction(), my_callback);

    # solving the optimization problem
```

```
    optimize!(model);
    setIter_til_utskrift(iters[1]);

    # check of statisfied the complementarities are
    compl,cval,prodCval,complStatus = complementaritySatisfaction(nc,nvc,cvar);
    setComp_til_utskrift(compl);

    println("status = ", termination_status(model));
    println("objective value = ", value(obj));
    println("Solve time = ", solve_time(model));
    println("MPCC complementarity = ", compl);
    println("MPCC satisfied = ",complStatus);
    println("IPOPT iterations = ", getIter_til_utskrift());
end
```