Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

January 14, 2018

Security constrained optimal power flow problems

A study of different optimization techniques

Samuel A. Cruz Alegría

Abstract

The electrical power grid is a critical infrastructure, and in addition to economic dispatch, the grid operation should be resilient to failures of its components. Increased penetration of renewable energy sources is placing greater stress on the grid, shifting operation of the power grid equipment towards their operational limits. Thus, any unexpected contingency could be critical to the overall operation. Consequently, it is essential to operate the grid with a focus on the security measures. *Security constrained optimal power flow* imposes additional security constraints to the optimal power flow problem. It aims for minimum adjustments in the base precontingency operating state, such that in the event of any contingency, the postcontingency states will remain secure and within operating limits. For a realistic power network, however, with numerous contingencies considered, the overall problem size becomes intractable for single-core optimization tools in short time frames for real-time industrial operations, such as rapid resolution of new optimal operating conditions over changing network demands, or real-time electricity market responses to electricity prices. Optimization software becomes a major bottleneck for the energy system modellers. Given that optimization software becomes a major bottleneck, this thesis aims to explore different optimization frameworks; specifically, it will cover the IPOPT and Optizelle optimization frameworks.

Advisor
Prof. Olaf Schenk
Assistants
Juraj Kardoš, Dr. Drosos Kourounis

Advisor's approval (Prof. Olaf Schenk): Date:

Acknowledgements

I would like to thank my advisors Prof. Dr. Olaf Schenk, Juraj Kardoš, and Dr. Drosos Kourounis, this work could not have been accomplished without all of their help and support. In particular, I would like to thank Juraj Kardoš for his constant support and guidance.

Contents

1 Introduction

1.1 Project description

This thesis aims to compare two optimization frameworks: *IPOPT* [14] and *Optizelle* [16]. These two frameworks were designed for *nonlinear optimization*. In particular, we note that the MATLAB proramming language [9] was used to deliver the results discussed in this thesis. We begin by exploring some relevant concepts from optimization theory, namely: unconstrained optimization, constrained optimization, and the primal-dual interior point method. These concepts are discussed as they are the fundamental mathematical formulations behind the IPOPT and Optizelle frameworks. We stress that the primal-dual interior point method is the most relevant here, as this is what is used by the two frameworks. Furthermore, it is important to us as it is one of the methods that can solve our SCOPF problem, which is a case of *nonlinear programming*; we will discuss these concepts in further detail in section 4. For simplicity, we will henceforth refer to IPOPT and Optizelle as *the frameworks*.

In order to test the capabilities of the frameworks, we must choose a problem that pushes them to the limit. For this reason, we chose to explore the security constrained power flow (SCOPF) problem. This problem is a nonlinear and nonconvex problem, given that the objective function is a quadratic function — therefore nonlinear and convex — with constraint functions that are nonlinear and nonconvex. In essence, SCOPF is the optimal power flow (OPF) problem with added security constraints. Given a realistic power network, the size of the problem can be quite large. Furthermore, the problem grows linearly with the number of contingencies; thus, the more security constraints that are imposed, the harder the problem becomes.

The standard formulation of the SCOPF problem is as follows [3]:

$$\begin{aligned}
&\underset{\mathbf{x}\in\mathscr{X}}{\text{minimize}}\ f(\mathbf{x}) && \text{(1a)}\\
&\text{subject to} && \text{(1b)}\\
&\qquad \mathbf{g}(\mathbf{x}) = \mathbf{0}, && \text{(1c)}\\
&\qquad \mathbf{h}(\mathbf{x}) \geq \mathbf{0}, && \text{(1d)}\\
&\qquad \mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max}. && \text{(1e)}
\end{aligned}$$

where:

- $f : \mathscr{X} \to \mathbb{R}$ is the *objective function* we wish to minimize;
- $g : \mathscr{X} \to \mathscr{Y}$ are the *equality constraints*;
- $h : \mathscr{X} \to \mathscr{Z}$ are the *inequality constraints*,

where X, Y, and Z denote *vector spaces*, more specifically, Hilbert spaces [17].

This standard formulation (1) can be modified so that we have only equality constraints, using *slack variables*, a notion we will discuss further in section 3:

$$\begin{aligned}
&\underset{\mathbf{x}\in\mathscr{X}}{\text{minimize}}\ f(\mathbf{x}) && \text{(2a)}\\
&\text{subject to} && \text{(2b)}\\
&\qquad \begin{bmatrix} \mathbf{g}(\mathbf{x}) \\ \mathbf{h}(\mathbf{x}) - \mathbf{s} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}, && \text{(2c)}\\
&\qquad \mathbf{s} \geq \mathbf{0}, && \text{(2d)}\\
&\qquad \mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max}. && \text{(2e)}
\end{aligned}$$

To be precise, the solution, $\mathbf{x}^*$, lies within the *feasible set* Ω, defined as

$$\Omega := \{\mathbf{x} \mid \mathbf{g}(\mathbf{x}) = 0 \ \wedge\ \mathbf{h}(\mathbf{x}) \geq 0 \ \wedge\ \mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max}\} \tag{3}$$

for the standard formulation (1), and

$$\Omega := \{\mathbf{x}, \mathbf{s} \mid \mathbf{g}(\mathbf{x}) = \mathbf{0} \ \wedge\ \mathbf{h}(\mathbf{x}) - \mathbf{s} = \mathbf{0} \ \wedge\ \mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max} \ \wedge\ \mathbf{s} \geq 0\} \tag{4}$$

for the formulation with slack variables (2).

Moreover, we repeat the equality and inequality constraints for all the power grid network configurations, reflecting the changed topology caused by the occurence of contingencies in the standard formulation (1); thus, once more, we highlight that the problem grows linearly with the number of contingencies. The number of equality and inequality constraints in the standard formulation (1) are $2 \times \text{nb}$ and $2 \times \text{nl}$, respectively, where nb is the number of buses and nl is the number of branches. We henceforth refer to the number of equality and inequality constraints as NEQ and NINEQ, respectively. We further note that the optimization variable, $\mathbf{x}$, contains variables for all scenarios, as well as global variables required for the optimization. Furthermore, we note the following cardinalities: $|Y| = \text{ns} \times \text{NEQ}$, $|Z| = \text{ns} \times \text{NINEQ}$, where ns is the number of scenarios, i.e., the cardinality of the contingency set. Henceforth, let us denote the contingency set by $\mathscr{G}$. We will not delve deeper into the formulation of the SCOPF problem as it is not relevant to the comparison of the frameworks, which is the intent of this thesis.

Given the formulation of the problem and considering the fact that, for a real power network, the problem size can be significant, we can appreciate the need for an efficient framework, i.e., one that will allow us to solve the SCOPF problem in a reasonable time and with reasonable precision.

Later in the thesis, we briefly discuss how to interface with the frameworks through code, i.e., how to formulate the problem in a way that is understandable by the frameworks.

IPOPT was designed to solve *large-scale* nonlinear optimization problems [14], whereas Optizelle was designed to solve *general purpose* nonlinear optimization problems [16]. Therefore, we designed tests to assess how the frameworks compare when solving an optimization problem on power grids provided by MATPOWER [3], against a growing number of contingencies. A *contingency* is a consideration of a possible failure in the power grid. Here, we shall refer to a *dynamic contingency set* as one consisting of different possible points of failure, i.e., of different contingencies.

In order to compare the frameworks, we analyze the behaviour with respect to three different numerical aspects:

1. Time to solution vs. number of contingencies.
2. Number of iterations vs. number of contingencies.
3. Time to solution, per iteration vs. number of contingencies.

Furthermore, we explore the interface, availability, and support tools offered by these two frameworks. Specifically, we discuss how straightforward it is to set up the SCOPF problem with these frameworks and the relative difficulty of interfacing the problem with them. We leave the results used for the comparison for section 7. For now, we argue the relevance of the three numerical aspects under scrutiny in the following paragraphs.

The experiments were performed on a single compute node and on a single core, so no deliberate parallelism was used to speed up the computation. Also, as detailed in the abstract, for a realistic power network, with numerous contingencies considered, the overall problem size becomes intractable for single-core optimization tools in short timeframes. Therefore, given a growing number of contingencies, we wish to minimize the time taken by a framework to solve the problem, as it is of paramount importance in real-time industrial operations.

Additionally, both frameworks rely on the primal-dual interior point method. Thus, in order to properly compare the frameworks, we must analyze the number of iterations with respect to the number of contingencies, as both methods should, in theory, perform similarly as far as the mathematical formulation goes. Of course, the numerical implementation of the method by each framework may lead to a different performance.

In order to conclude the thesis, we discuss the relative efficiency of the frameworks with respect to the three numerical aspects under investigation. Finally, we come to a verdict on our quest to find the better optimization framework with respect to our given test power grid.

1.2 Linearity and nonlinearity

A *linear function* (or map) $f(x)$ is one that satisfies both of the following properties [6]:

1. Additivity: $f(x+y) = f(x) + f(y)$.
2. Homegeneity: $f(\alpha x) = \alpha f(x)$.

In terms of a geometrical representation, the graph of a linear function is a line for a function of a single variable, a plane for a function of two variables, and a hyperplane for a function of numerous variables. It can be represented as

$$f(x) = ax^k + b, \tag{5}$$

where a is a constant and $k \in \{0, 1\}$ for a real variable x, or

$$A\mathbf{x} = \mathbf{b}, \tag{6}$$

for $\mathbf{x}$ in a real vector space, where we can express a coefficient matrix A as an n-by-n matrix.

An example of a linear function is $f(x) = x$, as can be seen in Figure 1.

A *nonlinear function* is one that does not satisfy the aforementioned properties or cannot be thought of in this manner. An example of a nonlinear function is $f(x) = \sin(x)$, as can be seen in Figure 2.

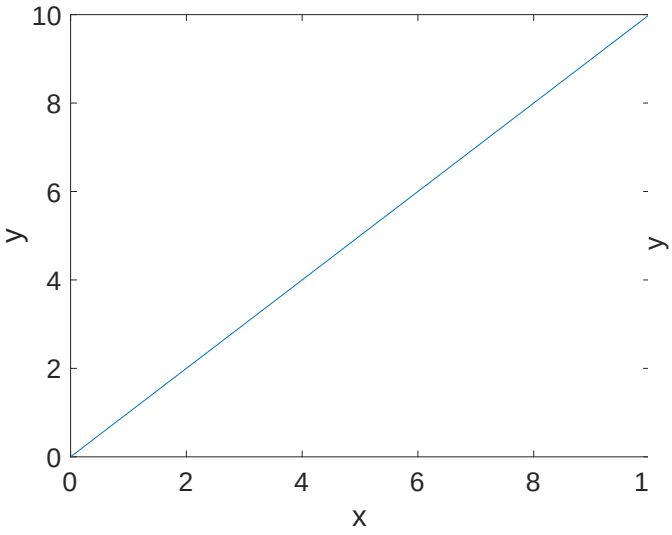

Figure 1. A linear function, $f(x) = x$.

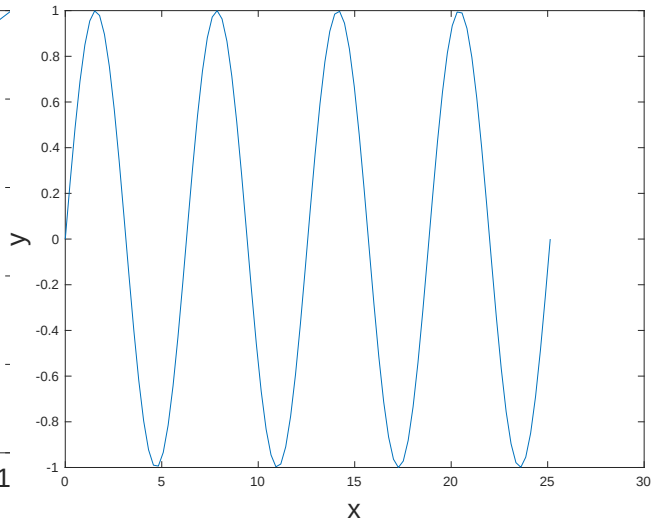

Figure 2. A nonlinear function, $f(x) = \sin(x)$.

1.3 Convex functions

Let $\mathscr{X}$ be a convex set in a real vector space, and let $f : \mathscr{X} \to \mathbb{R}$ be a function.

- f is called *convex* if

$$\forall x_1, x_2 \in \mathscr{X}, \forall t \in [0, 1] : f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2). \tag{7}$$

In other words, if we take a line segment from a point $(x_1, f(x_1))$ to $(x_2, f(x_2))$, and f is convex, then f must lie below this line segment. Similarly, in n dimensions, the function f must lie "below" the hyperplane. Evidently, a *nonconvex function* is one that does not satisfy this property.

An example of a convex function is $f(x) = x^2$, as can be seen in Figure 3. An example of a nonconvex function is $f(\mathbf{x}) = \sin(x_1) + \cos(x_2)$, as can be seen in Figure 4.

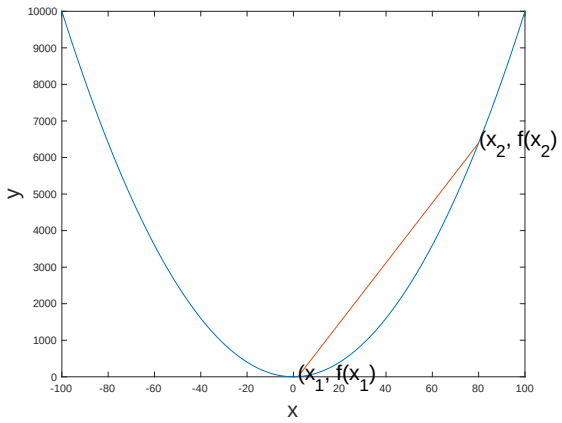

Figure 3. A convex function, $f(x) = x^2$.

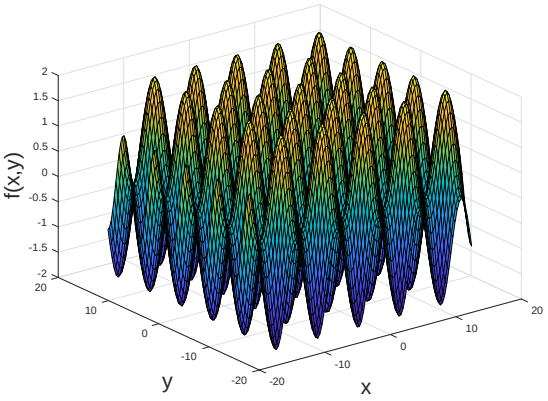

Figure 4. A nonconvex function, $f(\mathbf{x}) = \sin(x_1) + \cos(x_2)$.

1.4 Nonlinear and nonconvex nature of the problem

Notably, the SCOPF problem is a case of *nonlinear programming*, i.e., solving an optimization problem defined by a system of equalities and inequalities, collectively termed *constraints*, over a set of *unknown* variables, along with an objective function to be minimized, where some of the constraints or the objective function are nonlinear [13]. The SCOPF problem is both *nonlinear and nonconvex*.

Given the nonlinear nature of the problem, there may be many local optima – an example where we can see that a nonlinear problem has many local optima is the aforementioned function $f(x) = \sin(x)$, as seen in Figure 2.

In convex optimization, there is a unique optimal solution, which is globally optimal; otherwise, we can prove that there is no feasible solution to the problem. Thus, given that the problem is in part nonconvex, we may have more than one optimal solution, which may not necessarily be globally optimal. Moreover, due to the nonconvex nature of the problem, we need to correct properties of the Hessian matrix in order to ensure that the search direction yields a decrease in the objective function.

2 Unconstrained Optimization

2.1 Summary

This summary is based on the text published in chapter 9.2 of the book *A First Course in Numerical Methods* (Ascher, Greif) [7].

In unconstrained optimization, we look at our familiar *objective function*, $f(\mathbf{x})$, which we wish to minimize, i.e., we have the following problem:

$$\underset{\mathbf{x}\in\mathscr{X}}{\text{minimize}}\ f(\mathbf{x}), \tag{8}$$

where we require that the objective function $f : \mathscr{X} \to \mathscr{R}$ be a sufficiently smooth function (in several variables), without constraints.

A *necessary condition* for a minimum, $\mathbf{x}^*$, of (8) is

$$\nabla f(\mathbf{x}^*) = \mathbf{0}, \tag{9}$$

i.e., we require a *vanishing gradient* at the point $\mathbf{x}^*$. Generally, a point where the gradient vanishes is called a *critical point*.

The gradient vector, $\nabla f(\mathbf{x})$, is defined as follows:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}.$$

Furthermore, a *sufficient condition* for a critical point to be a minimum is that the *Hessian* matrix $\nabla^2 f(\mathbf{x}^*)$ be symmetric positive definite. We recall that a matrix A is *symmetric* if it is a square matrix, i.e., $A \in \mathbb{R}^{n \times n}$, and $A = A^T$. Furthermore, we note that a matrix is *positive definite* if $\mathbf{x}^T A \mathbf{x} > 0 \ \forall \mathbf{x} \neq 0$.

The Hessian matrix, $\nabla^2 f(\mathbf{x})$, is defined as follows:

$$H(\mathbf{x}) = \nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

We note that in order to satisfy the necessary condition for a minimum, we require that $f \in C^1$, i.e., that it is sufficiently smooth and differentiable, at least to first order. Moreover, in order to satisfy the sufficient condition for a minimum, we require that $f \in C^2$, i.e., that it is sufficiently smooth and differentiable at least to second order.

In general, solving (8) yields n nonlinear equations in n unknowns. For realistic problems, we often cannot find minimum points by inspection. Usually, it is unclear how many local minima a function $f(\mathbf{x})$ has, and if it has more than one local minimum, how to efficiently find the *global* minimum, which has the smallest value for $f(\mathbf{x})$.

Furthermore, sometimes there is no finite argument at which a function attains a minimum or a maximum. For example, consider $f(\mathbf{x}) = x_1^2 + x_2^2$ (Figure 5), which has a unique minimum at the origin (0,0), and no maximum. Now, consider $f(\mathbf{x}) = -(x_1^2 + x_2^2)$ (Figure 6), which has a unique maximum at the origin (0,0), and no minimum.

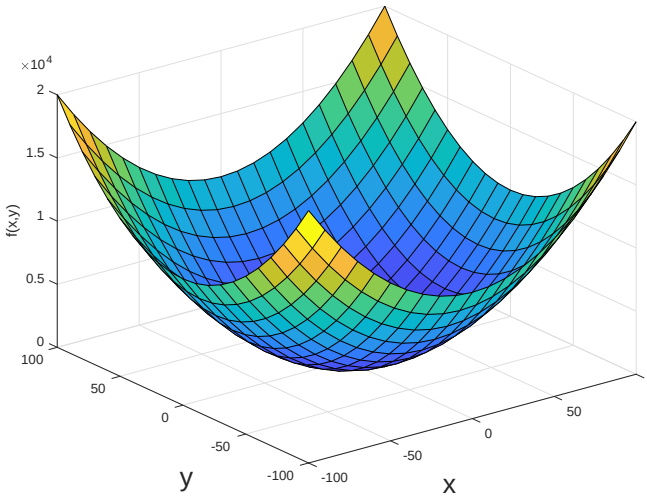

Figure 5. $f(\mathbf{x}) = x_1^2 + x_2^2$.

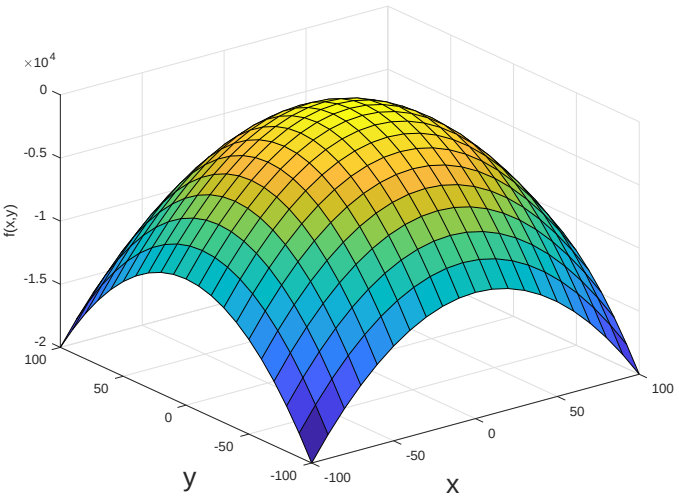

Figure 6. $f(\mathbf{x}) = -(x_1^2 + x_2^2)$.

As a final note, we recognize that, in general, finding a maximum for $f(\mathbf{x})$ is equivalent to finding the minimum for $-f(\mathbf{x})$.

3 Constrained Optimization

3.1 Summary

This summary is based on the text published in chapter 9.3 of the book *A First Course in Numerical Methods* (Ascher, Greif) [8].

In essence, we are looking at our familiar *objective function*, $f(\mathbf{x})$, which we wish to minimize, subject to certain equality and/or inequality constraints, i.e., we have the following problem:

$$\underset{\mathbf{x}\in\mathscr{X}}{\text{minimize}}\ \mathbf{f}(\mathbf{x}) \tag{10a}$$

$$\text{subject to } \mathbf{g}(\mathbf{x}) = \mathbf{0}, \tag{10b}$$

$$\mathbf{h}(\mathbf{x}) \geq \mathbf{0}, \tag{10c}$$

$$\mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max}. \tag{10d}$$

We recall that $g : \mathscr{X} \to \mathscr{Y}$ and $h : \mathscr{X} \to \mathscr{Z}$, meaning that g and h may be constructed from various functions g_i and h_i, respectively. Furthermore, we now group g and h into a single function c, defined as follows:

$$\mathbf{c}(\mathbf{x}) = \begin{bmatrix} \mathbf{g}(\mathbf{x}) \\ \mathbf{h}(\mathbf{x}) \end{bmatrix}, \tag{11}$$

which comprises all equality and inequality constraints — please note that this is not the same as the method of slack variables which we will discuss later in subsection 3.2. Therefore, we expand the definition of Ω from (3) as follows [8]:

$$\Omega := \{\mathbf{x} \in \mathscr{X} \mid c_i(\mathbf{x}) = 0,\ i \in \mathscr{E},\ c_i(\mathbf{x}) \geq 0,\ i \in \mathscr{I}\}. \tag{12}$$

Hence, $\mathscr{E}$ and $\mathscr{I}$ are the set of indices corresponding to equality and inequality constraints, respectively. We assume that $\forall i\ c_i(\mathbf{x}) \in C^1$. Moreover, we refer to any point $\mathbf{x} \in \Omega$ as a *feasible solution*, which we note is not necessarily an *optimal solution*.

If the unconstrained minimum of f is inside Ω, then we have our basic unconstrained minimization problem, as in section 2. Here, we consider the case where the unconstrained minimizer of f is not inside Ω.

A *level set* consists of all points $\mathbf{x}$ for which $f(\mathbf{x})$ has the same value. Morever, the gradient of f at some point $\mathbf{x}$ is orthogonal to the level set that passes through that point.

We define the *active set* for each $\mathbf{x} \in \mathscr{X}$ as follows:

$$\mathscr{A}(\mathbf{x}) := \mathscr{E} \cup \{i \in \mathscr{I} \mid c_i(\mathbf{x}) = 0\}. \tag{13}$$

Hence, we look for solutions where $\mathscr{A}(\mathbf{x}^*)$ is nonempty [8].

As in section 2, we have first order necessary and second order necessary and sufficient conditions for a local minimum. We will only desribe the first order necessary conditions for simplicity. We begin by assuming *constraint qualification* [8].

Definition 3.1 *Constraint qualification Let* $\mathbf{x}^*$ *be a local critical point and denote by* $\nabla c_i(\mathbf{x})$ *the gradients of the constraints. Furthermore, let* A_*^T *be the matrix whose columns are the gradients* $\nabla c_i(\mathbf{x}^*)$ *of all active constraints, i.e., those belonging to* $\mathscr{A}(\mathbf{x}^*)$*. The* constraint qualitication *assumption is that* A_*^T *has full column rank.*

Now, let us define the *Lagrangian*

$$\mathscr{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) - \sum_{i \in \mathscr{E} \cup \mathscr{I}} \lambda_i c_i(\mathbf{x}). \tag{14}$$

Then, we have the following necessary first order conditions for a minimum, collectively known as the *Karush–Kuhn–Tucker (KKT) conditions* [8]:

Theorem 3.1 *Constrained Minimization Conditions. Assume that* $f(\mathbf{x})$ *and* $c_i(\mathbf{x})$ *are smooth enough near a critical point* $\mathbf{x}^*$ *and that constraint qualification holds. Then there is a vector of* Lagrange multipliers $\boldsymbol{\lambda}^*$ *such that*

$$\nabla_{\mathbf{x}} \mathscr{L}(\mathbf{x}^*, \boldsymbol{\lambda}^*) = \mathbf{0}, \tag{15a}$$

$$c_i(\mathbf{x}^*) = \mathbf{0} \qquad \forall i \in \mathscr{E}, \tag{15b}$$

$$c_i(\mathbf{x}^*) \geq \mathbf{0} \qquad \forall i \in \mathscr{I}, \tag{15c}$$

$$\lambda_i^* \geq \mathbf{0} \qquad \forall i \in \mathscr{I}, \tag{15d}$$

$$\lambda_i^* c_i(\mathbf{x}^*) = \mathbf{0} \qquad \forall i \in \mathscr{E} \cup \mathscr{I}. \tag{15e}$$

Moreover, we have

$$\nabla L(\mathbf{x},\boldsymbol{\lambda}) = \begin{bmatrix} \nabla_{\mathbf{x}} L(\mathbf{x},\boldsymbol{\lambda}) \\ \nabla_{\boldsymbol{\lambda}} L(\mathbf{x},\boldsymbol{\lambda}) \end{bmatrix} = \begin{bmatrix} \nabla f(\mathbf{x}) - \nabla \mathbf{c}^{\mathrm{T}}(\mathbf{x})\boldsymbol{\lambda} \\ -\mathbf{c}(\mathbf{x}) \end{bmatrix}, \tag{16a}$$

$$\nabla^2 L(\mathbf{x},\boldsymbol{\lambda}) = \begin{bmatrix} \nabla_{\mathbf{xx}} L(\mathbf{x},\boldsymbol{\lambda}) & \nabla_{\mathbf{x}\boldsymbol{\lambda}} L(\mathbf{x},\boldsymbol{\lambda}) \\ \nabla_{\boldsymbol{\lambda}\mathbf{x}} L(\mathbf{x},\boldsymbol{\lambda}) & \nabla_{\boldsymbol{\lambda}\boldsymbol{\lambda}} L(\mathbf{x},\boldsymbol{\lambda}) \end{bmatrix} = \begin{bmatrix} \nabla^2 f(\mathbf{x}) - \nabla^2 \mathbf{c}(\mathbf{x})\boldsymbol{\lambda} & -\nabla \mathbf{c}^{\mathrm{T}}(\mathbf{x}) \\ -\nabla \mathbf{c}(\mathbf{x}) & \mathbf{0} \end{bmatrix}, \tag{16b}$$

where we refer to $H(\mathbf{x}) = \nabla^2 f(\mathbf{x})$ as the *Hessian* of $f(\mathbf{x})$, and $J(\mathbf{x}) = \nabla \mathbf{c}(\mathbf{x})$ as the *Jacobian* of the constraints.

The Hessian has the following structure:

$$H(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}. \tag{17}$$

The Jacobian has the following structure:

$$J(\mathbf{x}) = \begin{bmatrix} \frac{\partial c_1}{\partial x_1} & \frac{\partial c_1}{\partial x_2} & \cdots & \frac{\partial c_1}{\partial x_n} \\ \frac{\partial c_2}{\partial x_1} & \frac{\partial c_2}{\partial x_2} & \cdots & \frac{\partial c_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial c_m}{\partial x_1} & \frac{\partial c_m}{\partial x_2} & \cdots & \frac{\partial c_m}{\partial x_n} \end{bmatrix}. \tag{18}$$

We revisit our examples of unconstrained minimization from Figure 5 and Figure 6, but with the addition of constraints on the variables x_1 and x_2. Thus, we have the following examples for constrained optimization

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{minimize}}\ x_1^2 + x_2, \tag{19a}$$

$$\text{subject to } x_1^2 + x_2^2 \leq r^2. \tag{19b}$$

$$\underset{\mathbf{x}\in\mathbb{R}^2}{\text{maximize}}\ -(x_1^2 + x_2), \tag{20a}$$

$$\text{subject to } x_1^2 + x_2^2 \leq r^2. \tag{20b}$$

We set $r = 100$, with Figure 7 corresponding to (19), and with Figure 8 corresponding to (20).

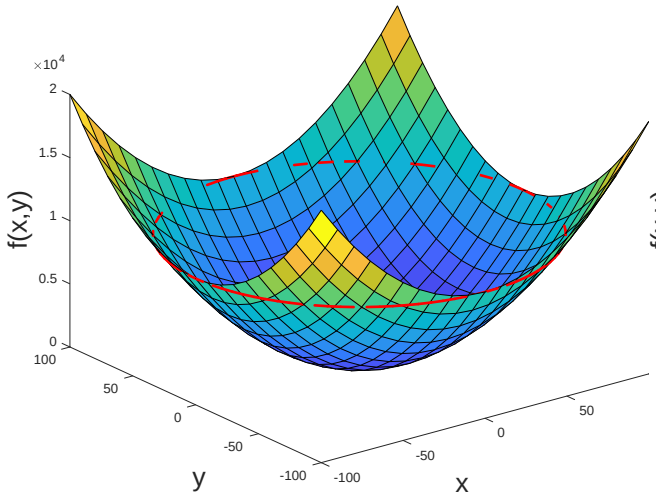

Figure 7. $f(\mathbf{x}) = x_1^2 + x_2^2$.

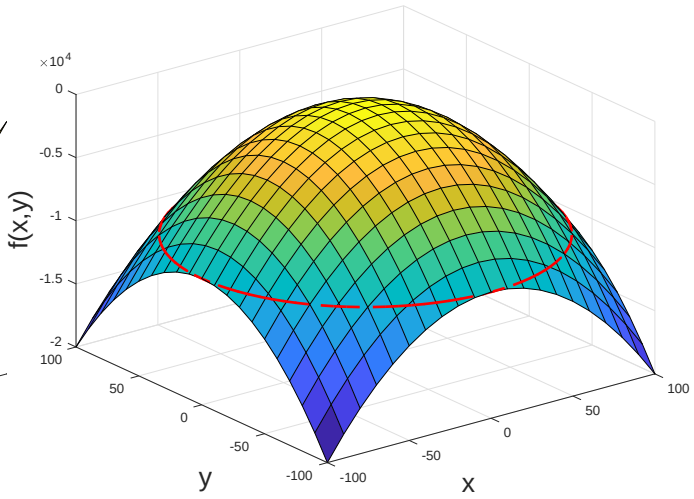

Figure 8. $f(\mathbf{x}) = -(x_1^2 + x_2^2)$.

3.2 Slack variables

We can convert inequality constraints to equality constraints by introducing *slack variables*, i.e.,

$$c_i(\mathbf{x}) = 0 \qquad \text{for } i \in \mathscr{E}, \tag{21a}$$

$$c_i(\mathbf{x}) - \mathbf{s} = 0 \qquad \text{for } i \in \mathscr{I}, \tag{21b}$$

$$\mathbf{s} \geq \mathbf{0}, \qquad \mathbf{s} \in \mathbb{R}^m, \tag{21c}$$

$$\mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max}, \qquad \mathbf{x}_{\min}, \mathbf{x}_{\max} \in \mathbb{R}^n. \tag{21d}$$

Hence, we define

$$\mathbf{x}_S = \begin{bmatrix} \mathbf{x} \\ \mathbf{s} \end{bmatrix}, \tag{22a}$$

$$g(\mathbf{x}) := c_i(\mathbf{x}), \text{ for } i \in \mathscr{E}, \tag{22b}$$

$$h(\mathbf{x}, \mathbf{s}) := c_i(\mathbf{x}) - \mathbf{s}, \text{ for } i \in \mathscr{I}. \tag{22c}$$

and solve for the following:

$$\underset{\mathbf{x}_S \in \mathscr{X}}{\text{minimize}}\ \mathbf{f}(\mathbf{x_S}) \tag{23a}$$

$$\text{subject to } \begin{bmatrix} \mathbf{g}(\mathbf{x}) \\ \mathbf{h}(\mathbf{x}, \mathbf{s}) \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \tag{23b}$$

$$\mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max}, \tag{23c}$$

$$\mathbf{s} \geq \mathbf{0}. \tag{23d}$$

4 Primal-Dual Interior Point Method

This summary is based on the text published in thesis [15].

4.1 Summary

The system of equations formed in (1) yield a problem that is nonlinear and nonconvex. Such a problem leads us to a *nonlinear program* (NLP), i.e., the process of solving an optimization problem defined by a system of equalities and inequalities — collectively termed *constraints* — over a set of unknown variables, along with an objective function to be maximized or minimized, where some of the constraints of the objective function are nonlinear [13].

For simplicity, let us consider the following NLP:

$$\underset{\mathbf{x} \in \mathscr{X}}{\text{minimize}}\ \mathbf{f}(\mathbf{x}) \tag{24a}$$

$$\text{subject to } \mathbf{c}(\mathbf{x}) = \mathbf{0} \tag{24b}$$

$$x^{(i)} \geq 0, \tag{24c}$$

where $x^{(i)}$ denotes the ith element of $\mathbf{x}$.

A popular class of algorithms used to solve NLPs are called *sequential quadratic programming (SQP)* methods. Unfortunately, at the beginning of the optimization process these type of methods may require a considerable amount of time to identify the active set $\mathscr{A}$, a problem that is known to be an NP-hard combinatorial problem. Thus, in the worst case, the solution time increases exponentially with the size of the problem.

4.2 Barrier methods

There exist methods that can get around the problem of identifying the active bound constraints mentioned in subsection 4.1. Such a class of methods are *barrier methods*; they circumvent the problem by replacing the bound constraints with a logarithmic barrier term which is added to the objective function f. For now, we assume that there are no inequality constraints other than the bounds (24c). We can make this assumption given that, using slack variables, we convert inequality constraints to equality constraints and are left only with the bounds on the variables $\mathbf{x}$ and $\mathbf{s}$, as illustrated in (23). Thus, we have the following problem:

$$\underset{\mathbf{x} \in \mathscr{X}}{\text{minimize}}\ \phi_\mu(\mathbf{x}) := f(\mathbf{x}) - \mu \sum_{i \in \mathscr{I}} \ln\left(x^{(i)}\right), \tag{25a}$$

$$\text{subject to } \mathbf{c}(\mathbf{x}) = \mathbf{0}, \tag{25b}$$

where μ is the barrier parameter, subject to $\mu > 0$. We recall that the $\ln(x)$ function is not defined at $x = 0$.

Hence, the objective function of our *barrier problem* (25) becomes increasingly large as $\mathbf{x}$ approaches the boundary of the region, defined by the lower bounds (24c). Therefore, the solution $\mathbf{x}_*^\mu$ must lie in the interior of this set, i.e., $(x_*^\mu)^{(i)} > 0$ for $i \in \mathscr{I}$.

The degree of influence of the barrier term "$-\mu \sum_{i\in\mathscr{I}}$" is determined by the size of μ and, under certain conditions, $\mathbf{x}_*^\mu$ converges to a local solution $\mathbf{x}^*$ of the original problem (24) as $\mu \to 0$.

The KKT conditions for our system (25) are thus:

$$\nabla \phi_\mu(\mathbf{x}) + \nabla \mathbf{c}(\mathbf{x})^T \boldsymbol{\lambda} = \mathbf{0}, \tag{26a}$$
$$\mathbf{c}(\mathbf{x}) = \mathbf{0}. \tag{26b}$$

Solving this system of equations directly with a Newton-type method, as in a straightforward application of an SQP method to (25), leads to a method we refer to as a *primal method*, which deals only with the *primal variables* $\mathbf{x}$ and possibly the equality multipliers $\boldsymbol{\lambda}$ as iterates. However, the term "$\nabla \phi_\mu(\mathbf{x})$" has components including $\frac{\mu}{x^{(i)}}$, thus, the system is not defined at a solution $\mathbf{x}^*$ of our base NLP (24) with the active bound $x^{(i)} = 0$, and the radius of convergence of Newton's method applied to (26) converges to 0 as $\mu \to 0$, i.e., there are increasingly fewer starting points for which the method converges.

For this reason, it is sometimes more suitable to use *primal-dual* methods rather than following a primal approach. Here, we introduce the *dual variables* $\mathbf{v}$, defined as:

$$v^{(i)} := \frac{\mu}{x^{(i)}}. \tag{27}$$

With this definition, the KKT conditions (26) are quivalent to the following perturbed KKT conditions or *primal-dual equations*:

$$\nabla \phi_\mu(\mathbf{x}) + \nabla \mathbf{c}(\mathbf{x})^T \boldsymbol{\lambda} - \mathbf{v} = \mathbf{0}, \tag{28a}$$
$$\mathbf{c}(\mathbf{x}) = \mathbf{0}, \tag{28b}$$
$$x^{(i)} v^{(i)} - \mu = 0 \qquad \text{for } i \in \mathscr{I}. \tag{28c}$$

We notice that for $\mu = 0$, the conditions (28) together with the inequalities

$$x^{(i)} \geq 0 \text{ and } v^i \geq 0 \text{ for } i \in \mathscr{I} \tag{29}$$

are actually the KKT conditions (26) for the barrier problem (25), where the dual variables $\mathbf{v}$ then correspond to the multipliers for the bound constraints (24c).

Primal-dual methods solve system (28) using a Newton-type approach, maintaining iterates for both $\mathbf{x}_k$ and $\mathbf{v}_k$, and possibly $\boldsymbol{\lambda}_k$. Since the inequalities (29) hold strictly at the optimal solution of the barrier problem (25) for $\mu > 0$, $\mathbf{x}_k$ and $\mathbf{v}_k$ are always required to strictly satisfy the inequalities, i.e.,

$$x_k^{(i)} > 0 \text{ and } v_k^{(i)} > 0 \text{ for } i \in \mathscr{I} \tag{30}$$

for all k, and can approach 0 only asymptotically as $\mu \to 0$. Given that the bound 0 is never actually reached, these methods are often called *interior point methods*.

An example of how such a method works in $\mathbb{R}^2$ can be seen in Figure 9 — blue lines show the constraints, red shows each iteration of the algorithm [5].

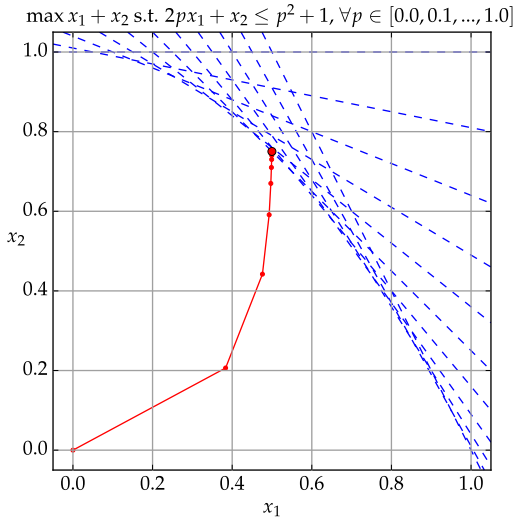

Figure 9. Example interior point method solution.

5 Matpower

5.1 Summary

This summary is based on the text published in paper [3].

The primary functionality of MATPOWER is to solve power flow and optimal power flow problems. This involves the following:

1. Preparing the input data defining all of the relevant power system parameters.
2. Invoking the function to run the simulation.
3. Viewing and accessing the results that are printed to the screen and/or saved in output and data structures or files.

MATPOWER is a modular framework which facilicates the extension of the standard OPF model by additional security constraints, yielding the SCOPF problem. The implementation of the SCOPF problem and its IPOPT interface were created before starting this project. The contribution of this work is the implementation of the SCOPF interface to the Optizelle solver and the analysis of its computational complexity.

5.2 Preparing case input data

The input data for the case to be simulated are specified in a set of data matrices packaged as the fields of a MATLAB struct, referred to as “MATPOWER case" struct and conventionally denoted by the variable `mpc`. This struct is typically defined in a case file, either a function M-file whose return value is the `mpc` struct or a MAT-file that defines a variable named `mpc` when loaded. The main simulation routines, whose names begin with “`run`,” accept either a file name or a MATPOWER case struct as input. We can use the `loadcase` function to load the data from a case file into a struct if we wish to make modifications to the data before passing it to the simulation.

```
mpc = loadcase(casefilename);
```

5.3 Solving the case

The solver is invoked by calling one of the main simulation functions. In our case, we used our own file, `runscopf.m`, passing in our `mpc` struct along with two other structs:

```
[RESULTS, SUCCESS, info] = runscopf(mpc, contingencies, mpopt);
```

Where `mpc` is as aforementined, `contingencies` is our contingency set, and `mpopt` is a struct containing MATPOWER options.

Finally, in order to extract the results, we can simply access the `RESULTS` struct. Moreover, if we're looking for information, such as the time taken to execute the algorithm, we can extract it from the `info` struct. In essence, we can choose which information we want to deliver back to the user from our solvers; in fact, this flexibility was exploited and it is how we put the results together. We note that in case of failure, these results are not delivered back to the user, as an exception is thrown.

6 Optimization Software

In section 5, we gave a high-level description of how MATPOWER can be set up and used to solve the OPF or the SCOPF problem. However, we have yet to explore how we set up the IPOPT and Optizelle solvers that were used by MATPOWER to solve the problem.

6.1 IPOPT

This summary is based on the text published in paper [1], and the text published in the website for IPOPT [14].

6.1.1 Introduction to IPOPT

IPOPT (**I**nterior **P**oint **OPT**imizer, pronounced eye-pea-Opt) is a software package for large-scale *nonlinear optimization*. It is designed to find (local) solutions of mathematical optimization problems of the following form:

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f(\mathbf{x}) \tag{31a}$$

$$\text{subject to } \mathbf{c}_{\min} \leq \mathbf{c}(\mathbf{x}) \leq \mathbf{c}_{\max}, \tag{31b}$$

$$\mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max}, \tag{31c}$$

where $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ is the objective function, and $\mathbf{c}(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$ are the constraint functions, which comprise both equality and inequality constraints. The vectors $\mathbf{c}_{\min}$ and $\mathbf{c}_{\max}$ denote the lower and upper bounds on the constraints, and the vectors $\mathbf{x}_{\min}$ and $\mathbf{x}_{\max}$ are the bounds on the variables $\mathbf{x}$. The functions $f(\mathbf{x})$ and $\mathbf{c}(\mathbf{x})$ can be nonlinear *and* nonconvex, but should be twice continuously differentiable (C^2) [14]. Indeed, we require C^2 in order to satisfy the sufficient condition for a minimum. We note that, where we wish to express equality constraints, we would set the corresponding components in $\mathbf{c}_{\min}$ and $\mathbf{c}_{\max}$ to the same value.

IPOPT is written in C++ and is released as *open source code* under the Eclipse Public License (EPL). It is *available* from the COIN-OR initiative [4].

The IPOPT distribution can be used to generate a library linked to one's own C++, C, Fortran, or Java code, as well as a solver executable for the AMPL [11] modeling environment. The package includes interfaces to CUTEr [12] optimization testing environment, as well as the MATLAB and R programming environments. IPOPT can be used on Linux/UNIX, Mac OS X, and Windows platforms.

Some special features of IPOPT are [14] the following:

- Derivative checker.
- Hessian approximation.
- Warm starts via AMPL.
- Ipopt: Optimal Sensitivity Based on AMPL/Ipopt.
- Employing second derivative information, if available, or otherwise approximating it by means of a limited-memory quasi-Newton approach (BFGS and SR1) [2].
- Global convergence of the method is ensured by a line search procedure, based on a filter method [2].

One drawback of IPOPT is that it is not thread-safe; it currently uses smart pointers and a tagging mechanism [2].

6.1.2 Interfacing with IPOPT through code

IPOPT requires more than the problem definition, namely, the following:

1. Problem dimensions.
   - Number of variables.
   - Number of constraints.
2. Problem bounds.
   - Variable bounds.
   - Constraint bounds.
3. Initial starting point.
   - Initial values for the *primal* $\mathbf{x}$ variables.
   - Initial values for the multipliers (only required for warm-start option).
4. Problem structure.
   - Number of nonzeros in the Jacobian of the constraints.
   - Number of nonzeros in the Hessian of the Lagrangian function.
   - Sparsity structure of the Jacobian of the constraints.
   - Sparsity structure of the Hessian of the Lagrangian function.
5. Evaluation of problem functions — information evaluated using a given point $(\mathbf{x}, \boldsymbol{\lambda}, \sigma_f)$, coming from IPOPT.
   - Objective function $f(\mathbf{x})$.
   - Gradient of the objective $\nabla f(\mathbf{x})$.
   - Constraint function values $\mathbf{c}(\mathbf{x})$.
   - Jacobian $\nabla \mathbf{c}(\mathbf{x})^T$ of the constraints.
   - Hessian of the Lagrangian function, $L(\mathbf{x}, \boldsymbol{\lambda})$, where,

$$L(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \langle \mathbf{c}(\mathbf{x}), \boldsymbol{\lambda} \rangle, \tag{32}$$

$$H_{\mathbf{xx}}(\mathbf{x}, \boldsymbol{\lambda}, \sigma_f) = \sigma_f \nabla^2 f(\mathbf{x}) + \langle \nabla^2 \mathbf{c}(\mathbf{x}), \boldsymbol{\lambda} \rangle, \tag{33}$$

     where we introduce a factor σ_f in front of the objective term so that IPOPT can ask for the Hessian of the objective function or the constraints independently, if required. We could choose only the objective term by setting the constraint multipliers λ_i to 0 and σ_f to 1. Evidently, to choose only the constraints, we would set σ_f to 0.

Moreover, we note that IPOPT is invoked as follows:

```
[x, info] = ipopt(x0,funcs,options);
```

Here:

- `x0` is our initial guess.
- `funcs` is a struct that containts the definitions of the following:
  - The objective function: $f(\mathbf{x})$.
  - The gradient of the objective function: $\nabla f(\mathbf{x})$.
  - The evaluation of the constraints: $\mathbf{c}(\mathbf{x})$.
  - The Jacobian of the constraints: $\nabla \mathbf{c}(\mathbf{x})$.
  - The Hessian of the Lagrangian function (see (33)).
  - The structure of the Jacobian of the constraints.

- The structure of the Hessian of the Lagrangian function.

- `options` is a struct that contains the following:
  - (Possibly) auxiliary data.
  - Options for the interior point method.
  - Lower and upper bounds for the primal variables $\mathbf{x}$.
  - Lower and upper bounds for the constraints $c_i(\mathbf{x})$.
  - (Optional, for warm start) Set initial values for the Lagrange multipliers for the lower and upper bounds on the variables and for the constraints. A warm start is enabled by setting the option `warm_start_init_point = 'yes'`.

In the remainder of this section, we go deeper into the requirements from IPOPT and show some code excerpts where appropriate.

6.1.3 Problem dimension

In the context of our problem, what is meant by the problem dimension is the number of variables x_i that make up the vector of unknown variables $\mathbf{x}$, as well as the number of constraints in our problem, i.e., the number of constraints $c_i(\mathbf{x})$ together with the bounds $\mathbf{x}_{\min} < \mathbf{x} < \mathbf{x}_{\max}$ discussed in (10). We note that there is no need to formulate the problem using slack variables as presented in subsection 3.2 in IPOPT, as the problem is automatically formulated as (2) [15]. Moreover, the number of variables and the number of constraints are derived automatically by IPOPT.

6.1.4 Problem bounds

The problem bounds in IPOPT consist of the bounds on the variable $\mathbf{x}$ and the bounds on the equality and inequality constraints. We note that the equality and inequality constraints are grouped together in $\mathbf{c}(\mathbf{x})$. Hence, we are referring specifically to the following:

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f(\mathbf{x}) \tag{34a}$$

$$\text{subject to } \mathbf{c}_{\min} \leq \mathbf{c}(\mathbf{x}) \leq \mathbf{c}_{\max}, \tag{34b}$$

$$\mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max}. \tag{34c}$$

As we know from our standard problem formulation (1), we don't have upper bounds on the equality and inequality constraints, we only have lower bounds. In IPOPT, the concept of being unbounded from above is, by definition, $+\infty$. Hence, we have the following:

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}}\ f(\mathbf{x}) \tag{35a}$$

$$\text{subject to } \mathbf{c}_{\min} \leq \mathbf{c}(\mathbf{x}) \leq +\boldsymbol{\infty}, \tag{35b}$$

$$\mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max}. \tag{35c}$$

In the code, we need to specify the bounds individually for each x_i and each c_i, respectively; however, we can group the individual bounds in an array. Furthermore, we need to specify the bounds separately for the variables and the constraints. Here is an excerpt from the code that illustrates how this was done:

```
options.lb = xmin;
options.ub = xmax;
options.cl = [repmat([zeros(2*nb, 1);  -Inf(2*nl2, 1)], [ns, 1]); l];
options.cu = [repmat([zeros(2*nb, 1); zeros(2*nl2, 1)], [ns, 1]); u+1e10];
```

Here `options.lb` and `options.ub` denote the lower and upper bounds on $\mathbf{x}$, respectively, and `options.cl` and `options.cu` denote the lower and upper bounds on $\mathbf{c}(\mathbf{x})$, respectively.

As was mentioned in section 1, `nb` represents the number of buses, and though not previously mentioned, `nl2` denotes the number of constrained lines. The MATLAB function `repmat` allows us to repeat the lower and upper bounds on the constraints for each scenario. This can be seen as the bounds are repeated `ns` times, where `ns` represents the number of scenarios.

As also previously mentioned in section 1, we recall that the bounds on $\mathbf{x}$ already account for all scenarios, since

x contains variables for all scenarios as well as global variables required for the optimization.

Furthermore, we defined `xmin` and `xmax` in the previous code excerpt using the MATPOWER package and later made some modifications in the code so that the bounds aligned with our SCOPF problem. This was done as follows:

```
%% bounds on optimization vars xmin <= x <= xmax
[x0, xmin, xmax] = getv(om); %returns standard OPF form [Va Vm Pg Qg]

% add small pertubation to UB so that we prevent ipopt removing variables
% for which LB=UB, except the Va of the reference bus
tmp = xmax(REFbus_idx);
xmax = xmax + 1e-10;
xmax(REFbus_idx) = tmp;

% replicate bounds for all scenarios and append global limits
xl = xmin([VAopf VMopf(nPVbus_idx) QGopf PGopf(REFgen_idx)]); %local variables
xg = xmin([VMopf(PVbus_idx) PGopf(nREFgen_idx)]); %global variables
xmin = [repmat(xl, [ns, 1]); xg];

xl = xmax([VAopf VMopf(nPVbus_idx) QGopf PGopf(REFgen_idx)]); %local variables
xg = xmax([VMopf(PVbus_idx) PGopf(nREFgen_idx)]); %global variables
xmax = [repmat(xl, [ns, 1]); xg];
```

6.1.5 Initial starting point

Given that we did not use the warm-start option for IPOPT, we only needed to specify the initial values for the *primal* variables **x**, i.e., as described in 6.1.2, we only need to specify our initial guess `x0`. We note that in the code, `x0` was first defined using the MATPOWER package, as shown in the previous code excerpt, but was later modified in an attempt to select an interior initial point based on the bounds, as follows:

```
    ll = xmin; uu = xmax;
    ll(xmin == -Inf) = -1e10;               %% replace Inf with numerical proxies
    uu(xmax ==  Inf) =  1e10;
    x0 = (ll + uu) / 2;                     %% set x0 mid-way between bounds
    k = find(xmin == -Inf & xmax < Inf);    %% if only bounded above
    x0(k) = xmax(k) - 1;                    %% set just below upper bound
    k = find(xmin > -Inf & xmax == Inf);    %% if only bounded below
    x0(k) = xmin(k) + 1;                    %% set just above lower bound

    % adjust voltage angles to match reference bus
    Varefs = bus(REFbus_idx, VA)   (pi/180);
    for i = 0:ns-1
        idx = model.index.getGlobalIndices(mpc, ns, i);
        x0(idx(VAscopf)) = Varefs(1);
    end
```

6.1.6 Problem structure

As we have discussed before, IPOPT needs to know the number of nonzeros in both the Jacobian of the constraints, $\nabla \mathbf{c}(\mathbf{x})$, and the Hessian of the Lagrangian function. This, however, can be derived by IPOPT as soon as we define the sparsity structure of both.

In order to specify the sparsity structure of the Jacobian of the constraints and the Hessian of the Lagrangian function for our SCOPF problem, a relatively large portion of code was used. For the interested reader, you can find this portion of code in Appendix A.

6.1.7 Evaluation of problem functions

As mentioned before, in order to call IPOPT we require a `funcs` structure. This structure contains all of the functions that we need in order to evaluate the problem functions using a given point $(\mathbf{x}, \boldsymbol{\lambda}, \sigma_f)$, kept by IPOPT. In the code, we defined the `funcs` structure as follows:

```
  %% assign function handles
  funcs.objective         = @objective;
  funcs.gradient          = @gradient;
  funcs.constraints       = @constraints;
```

```
funcs.jacobian          = @jacobian;
funcs.hessian           = @hessian;
funcs.jacobianstructure = @(d) Js;
funcs.hessianstructure  = @(d) Hs;
```

Here the function handles correspond to the definitions of the functions mentioned in 6.1.2. For the interested reader, you can find the portions of the code to define these functions in Appendix A.

6.1.8 Pros and cons

Pros:

- The problem dimensions and bounds come solely from the problem definition.
- It is a nonlinear programming solver designed for solving sparse large-scale problems.
- It can be customized for a variety of matrix formats (for instance, *triplet format*) [14].

Cons:

- It needs to know the number of nonzero elements and the sparsity structure (row and column indices of each of the nonzero entries) of the constraint Jacobian and the Hessian of the Lagrangian function.
  - Once defined, this nonzero structure must remain constant for the entire optimization procedure, meaning that the structure needs to include entries for any element that could ever be nonzero, not only those that are nonzero at the starting point.

6.2 Optizelle

This summary is based on the text published in the manual of Optizelle (version 1.2.0) [17], and the text published in the website for Optizelle [16].

6.2.1 Introduction to Optizelle

Optizelle is an *open source* software library designed to solve general purpose nonlinear optimization problems of the following form(s):

- Unconstrained

$$\underset{\mathbf{x}\in\mathcal{X}}{\text{minimize}}\ f(\mathbf{x}). \tag{36a}$$

- Equality constrained

$$\underset{\mathbf{x}\in\mathcal{X}}{\text{minimize}}\ f(\mathbf{x}) \tag{37a}$$
$$\text{subject to } \mathbf{g}(\mathbf{x}) = \mathbf{0}. \tag{37b}$$

- Inequality constrained

$$\underset{\mathbf{x}\in\mathcal{X}}{\text{minimize}}\ f(\mathbf{x}) \tag{38a}$$
$$\text{subject to } \mathbf{h}(\mathbf{x}) \geq \mathbf{0}. \tag{38b}$$

- Constrained

$$\underset{\mathbf{x}\in\mathcal{X}}{\text{minimize}}\ f(\mathbf{x}) \tag{39a}$$
$$\text{subject to } \mathbf{g}(\mathbf{x}) = \mathbf{0}, \tag{39b}$$
$$\mathbf{h}(\mathbf{x}) \geq \mathbf{0}. \tag{39c}$$

It features the following:

- State of the art algorithms:

  - Unconstrained — steepest descent, preconditioner nonlinear-CG (Fletcher–Reeves, Polak–Ribiere, Hestenes–Stiefel), BFGS, Newton-CG, SR1, trust-region Newton, Barzilai–Borwein two point approximation.
  - Equality constrained — inexact composite-step SQP.
  - Inequality constrained — *primal-dual interior point method* for cone constraints (linear, second-order cone, and semidefinite), log-barrier method for cone constraints.
  - Constrained — any combination of the above.
- Open source:
  - Released under the 2-Clause BSD License.
  - Free and ready to use with both open and closed source commercial codes.
- Multilanguage support:
  - Interfaces to C++, MATLAB/Octave, and Python.
- Robust combinations and repeatability:
  - Can stop, archive, and restart the computation from any optimization iteration.
  - Combined with multilanguage support, the optimization can be started in one language and migrated to another.
- User-defined parallelism:
  - Fully compatible with OpenMP, MPI, and GPUs.
- Extensible linear algebra:
  - Supports user-defined vector algebra and preconditioners.
  - Enables sparse, dense, and matrix-free computations.
  - Defines custom inner products and compatibility with preconditioners such as algebraic multigrid which makes Optizelle well-suited for PDE constrained optimization.
- Sophisticated control of the optimization algorithms:
  - Allows the user to insert arbitrary code into the optimization algorithm, which enables custom heuristics to be embedded without modifying the source.

Furthermore, there is a *community forum* for Optizelle.

In our experience, one drawback for Optizelle is that the documentation [17] stated that the inequality constraints could be formulated as follows:

$$\mathbf{h}(\mathbf{x}) \geq \mathbf{0};$$

however, we observed that strict inequalities $\mathbf{h}(\mathbf{x}) > \mathbf{0}$ were required; otherwise, problems would arise in solving the SCOPF problem. Our observation seems to have been confirmed in the forum post found in [18].

6.2.2 Interfacing with Optizelle through code

As was mentioned before in section 1, we have

- $f : \mathcal{X} \to \mathbb{R}$ is the *objective function* we wish to minimize,
- $g : \mathcal{X} \to \mathcal{Y}$ are the *equality constraints*,
- $h : \mathcal{X} \to \mathcal{Z}$ are the *inequality constraints*,

where X, Y, and Z denote *vector spaces*, more specifically, Hilbert spaces [17]. For our problem, these vectors spaces are in $\mathbb{R}^m$, but Optizelle allows for the vector spaces to be spaces of functions such as $L^2(\Omega)$ or matrices such as $\mathbb{R}^{m\times n}$ as long as one provides the necessary operations required to compute on a vector of such a vector space [17].

Other than this, in order solve our *NLP*, Optizelle requires for us to define the following:

- The objective function $f(\mathbf{x})$, the gradient of the objective function $\nabla f(\mathbf{x})$, and the Hessian of the objective function $\nabla^2 f(\mathbf{x})$.
- The equality constraints, which are encoded into a function $\mathbf{g}(\mathbf{x})$, the Jacobian of the equality constraints $\nabla \mathbf{g}(\mathbf{x})$, and the Hessian of the equality constraints $\nabla^2 \mathbf{g}(\mathbf{x})$.
- The inequality constraints, which are encoded into a function $\mathbf{h}(\mathbf{x})$, the Jacobian of the inequality constraints $\nabla \mathbf{h}(\mathbf{x})$, and the Hessian of the inequality constraints $\nabla^2 \mathbf{h}(\mathbf{x})$.

Indeed, the Hessian definitions are necessary if we wish to use second-order algorithms.

It is important to note that Optizelle allows for the equality constraints to be nonlinear, but it requires that the inequality constraints be *affine* [17]. We recall an affine function is one where $\forall \alpha \in \mathbb{R}, h(\alpha x+(1-\alpha)x) = \alpha h(x)+(1-\alpha)h(x)$ or, equivalently, $h''(x) = 0$. This is required in order to simplify the line search that maintains the nonnegativity of the inequality constraints. In our SCOPF problem, we have a basis consisting of nonlinear equality and inequality constraints. For this reason, our original problem formulation (1) (repeated here for clarity):

$$\begin{aligned} &\underset{\mathbf{x}\in\mathcal{X}}{\text{minimize}}\ f(\mathbf{x}) && \text{(40a)}\\ &\text{subject to } \mathbf{g}(\mathbf{x}) = \mathbf{0}, && \text{(40b)}\\ &\qquad\qquad \mathbf{h}(\mathbf{x}) \geq \mathbf{0}, && \text{(40c)}\\ &\qquad\qquad \mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max}, && \text{(40d)} \end{aligned}$$

was reformulated as follows, using *slack variables*:

$$\begin{aligned} &\underset{\mathbf{x}\in\mathcal{X}}{\text{minimize}}\ f(\mathbf{x}) && \text{(41a)}\\ &\text{subject to } \begin{bmatrix} \mathbf{g}(\mathbf{x}) \\ \mathbf{h}(\mathbf{x})-\mathbf{s} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}, && \text{(41b)}\\ &\qquad\qquad \mathbf{s} \geq \mathbf{0}, && \text{(41c)}\\ &\qquad\qquad \mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max}. && \text{(41d)} \end{aligned}$$

In order to optimize effectively, Optizelle requires the evaluation $\mathbf{c}_\epsilon(\mathbf{x},\mathbf{s}) := \begin{bmatrix} \mathbf{g}(\mathbf{x}) \\ \mathbf{h}(\mathbf{x})-\mathbf{s} \end{bmatrix}$, the Fréchet (total) derivative applied to a vector, $\mathbf{c}'_\epsilon(\mathbf{x},\mathbf{s})\boldsymbol{\delta}\mathbf{x}$, and the adjoint of the Fréchet derivative applied to the vector of Lagrange multipliers for the equality constraints, $\mathbf{c}'_\epsilon(\mathbf{x},\mathbf{s})\boldsymbol{\delta}\mathbf{y}$. In order to use second-order algorithms, we also require the second derivative operation $(\mathbf{c}''_\epsilon(\mathbf{x},\mathbf{s})\boldsymbol{\delta}\mathbf{x})\boldsymbol{\delta}\mathbf{y}$.

Let us define the reformulated inequality constraints as

$$\mathbf{c}_\iota(\mathbf{x},\mathbf{s}) := \begin{bmatrix} \mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max} \\ \mathbf{s} \geq \mathbf{0} \end{bmatrix}. \tag{42}$$

As was the case for the (reformulated) equality constraints $\mathbf{c}_\epsilon(\mathbf{x},\mathbf{s})$, we require the same operations from $\mathbf{c}_\iota(\mathbf{x},\mathbf{s})$ but since $\mathbf{c}_\iota(\mathbf{x},\mathbf{s})$ is affine, $(\mathbf{c}''_\iota(\mathbf{x},\mathbf{s})\boldsymbol{\delta}\mathbf{x})\boldsymbol{\delta}\mathbf{z} = 0$.

6.2.3 Setting up Optizelle

To start, we need to create an *optimization state*, which contains an entire description of the current state of the optimization algorithm. This is unique to the particular optimization problem, but all algorithms in a particular formulation share the same state [17]. Most algorithms do not require information about all of these pieces, but they are present to make it easier to switch from one algorithm to another. In order to define an optimization state, we instantiate the state class with the particular class of formulation we need. In our case, we have a constrained optimization problem, so we define the optimization state by

```
1    state = Optizelle.Constrained.State.t(X,Y,Z,x,y,z);
```

Here, `X`, `Y`, and `Z` are the vector spaces corresponding to the codomain of our functions f, g, and h, respectively, from our standard formulation (1). In our case, $X, Y, Z \in \mathbb{R}^m$. We note that in the remainder of this section, $\mathbf{x}$ corresponds to $\mathbf{x}_S$ (compare (22a)), with the slack variables initialized to $\mathbf{s} = \mathbf{0}+\boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon}$ represents a small perturbation which was required based on our observation that the inequality constraints c_ι needed to be strictly satisfied. In the code

excerpt above, $\mathbf{x} \equiv x_S$, $\mathbf{y}$, and $\mathbf{z}$ denote the initial guesses for our unknown *primal* variables $\mathbf{x}$, the equality multipliers $\mathbf{y}$, and the inequality multipliers $\mathbf{z}$, respectively.

In order to pass the functions used in the optimization to Optizelle, we accumulate each of them into a *bundle of functions*. These bundles are simple structures that contain the appropriate function. Given that we have a constrained problem, we define this as follows:

```
fns = Optizelle.Constrained.Functions.t;
```

Then, we can define the information related to our objective function $f(\mathbf{x})$, our equality constraints $\mathbf{g}(\mathbf{x})$, and our inequality constraints $\mathbf{h}(\mathbf{x})$ as follows:

```
fns.f = MyObj(myauxdata);
fns.g = MyEq(myauxdata);
fns.h = MyIneq(myauxdata);
```

where `myauxdata` contains information that is used locally in each of the function evaluations.

In the following code excerpts, we show how we defined the functions used by Optizelle at a high level:

```
%% Define objective function.
function self = MyObj(myauxdata)
% Evaluation
self.eval = @(x) objective(x, myauxdata);

% Gradient
self.grad = @(x) gradient(x, myauxdata);

% Hessian-vector product
self.hessvec = @(x, dx) hessvec(x, dx, myauxdata);
...
end
```

In order to optimize the objective function $f(\mathbf{x})$, Optizelle requires for us to provide the evaluation of the objective function, and the gradient of the objective function. In order to use second-order algorithms, we also need to provide the product of the Hessian of the objective function with a vector δx. In the code excerpt above, we return a *struct* containing all of these necessary components, i.e., the returned struct `self`. Above, `self` is composed of the following:

- `self.eval` is defined as the *function handle* to our function which evaluates $f(\mathbf{x})$.
- `self.grad` is defined as the function handle to our function which evaluates $\nabla f(\mathbf{x})$.
- `self.hessvec` is defined as the function handle to our function which evaluates $\nabla^2 f(\mathbf{x})\delta\mathbf{x}$.

As Optizelle optimizes the objective function $f(\mathbf{x})$, these function handles are called with actual arguments. We will spare the definition of our functions pointed to by the function handles as it is not necessary in order to understand the code at a high level. For the interested reader, the definition of these functions can be found in Appendix B.

```
%% Define equality constraints.
function self = MyEq(myauxdata)

% y=g(x)
self.eval = @(x) constraints(x, myauxdata);

% y=g'(x)dx
self.p = @(x,dx) jacobvec(x, dx, myauxdata);

% xhat=g'(x)*dy
self.ps = @(x,dy) jacobvec(x, dy, myauxdata);

% xhat=(g''(x)dx)*dy
self.pps = @(x,dx,dy) hessian(x, myauxdata, dy)   dx;
...
end
```

Similarly to the code excerpt describing the necessary information for the objective function, the code excerpt above details the necessary information for the equality constraints $\mathbf{g}(\mathbf{x})$. Here, the returned struct `self` is composed of the following:

- `self.eval` is defined as the function handle to our function which evaluates $\mathbf{g}(\mathbf{x})$.
- `self.p` is defined as the function handle to our function which evaluates $\nabla\mathbf{g}(\mathbf{x})\delta x$.
- `self.ps` is defined as the function handle to our function which evaluates $\nabla\mathbf{g}(\mathbf{x})\cdot\delta y$.
- `self.pps` is defined as the function handle to our function which evaluates $(\nabla^2\mathbf{g}(\mathbf{x})\delta x)\cdot\delta y$.

```
%% Define inequality constraints.
function self = MyIneq(myauxdata)
% z=h(x)
self.eval = @(x) constraints(x, myauxdata);

% z=h'(x)dx
self.p = @(x,dx) jacobvec(x, dx, myauxdata);

% xhat=h'(x)*dz
self.ps = @(x,dz) jacobvec(x, dz, myauxdata);

% xhat=(h''(x)dx)*dz
self.pps = @(x,dx,dz) sparse(length(x),length(x));
...
end
```

Finally, the code excerpt above details the necessary information for the inequality constraints $\mathbf{h}(\mathbf{x})$. Here, the returned struct `self` is composed of the following:

- `self.eval` is defined as the function handle to our function which evaluates $\mathbf{h}(x)$.
- `self.p` is defined as the function handle to our function which evaluates $\nabla\mathbf{h}(\mathbf{x})\delta x$.
- `self.ps` is defined as the function handle to our function which evaluates $\nabla\mathbf{h}(\mathbf{x})\cdot\delta z$.
- `self.pps` is defined as a sparse zero-matrix, since the inequality constraints are affine.

The last step is for us to *call the optimization solver*. We note that this can only be done once the state, (optionally) parameters, and functions are set. Given that we have a constrained problem, we invoke Optizelle's optimization algorithm as follows:

```
state = Optizelle.Constrained.Algorithms.getMin( ...
      Optizelle.Rm,Optizelle.Rm,Optizelle.Rm,Optizelle.Messaging.stdout, ...
      fns,state);
```

After the optimization solver concludes, the solution lives in the optimization state in a variable called `x`, and the reason the optimization stopped resides in a variable called `opt_stop`. In the code, we extracted both as follows:

```
% Print out the reason for convergence
fprintf('The_algorithm_converged_due_to:_%s\n', ...
   Optizelle.OptimizationStop.to_string(state.opt_stop));

results = struct('x', state.x);
```

6.2.4 Pros and cons

Pros:

- Does not need sparsity structure of Jacobian of constraints or Hessian of the Lagrangian function.
- The Lagrangian function does not need to be specified by the user. This means that the user can specify all related data to the objective function $f(\mathbf{x})$, the equality constraints $\mathbf{g}(\mathbf{x})$, and the inequality constraints $\mathbf{h}(\mathbf{x})$, separately; indeed, it is *required* for the formulation to be done as such. This feature could be viewed favourably with respect to modularity.

Cons:

- Requires that the inequality constraints function h be *affine*. Since the equality and inequality constraints in the SCOPF problem are nonlinear, we are *required* to reformulate the standard formulation (1) using slack variables, yielding the formulation (2).
- Being required to separately specify all related data for the objective function $f(\mathbf{x})$, the equality constraints $\mathbf{g}(\mathbf{x})$, and the inequality constraints $\mathbf{h}(\mathbf{x})$, means that we have more possible points of failure.

7 Experiments

7.1 Methodology

We tested the relative performance of IPOPT and Optizelle by having them solve a SCOPF problem with a growing dynamic contingency set. This means we used a test power grid provided by MATPOWER and identified a feasible dynamic contingency set $\mathscr{G}$. The test power grid we used was *case9*. We began by testing the frameworks with an empty contingency set — i.e., $|\mathscr{G}| = 0$ — sequentially incrementing the number of contingencies, reaching a maximum of six contingencies — i.e., $|\mathscr{G}| = 6$.

The experiment measured the convergence behaviour of IPOPT and Optizelle by examining the value of the objective function, $f(\mathbf{x})$, the norm of the gradient, $\|\nabla f(\mathbf{x})\|$, and the value of the power flow equations, $\mathbf{g}(\mathbf{x})$. Furthermore, the average time per iteration was measured for both frameworks.

When performing these measurements, no limit on the number of iterations was placed on IPOPT, whereas we limited the number of iterations in Optizelle. This was done since IPOPT always managed to converge to a solution in less than fifteen iterations, while Optizelle never managed to converge to a solution. In order to measure the average number of iterations for both solvers, the average time and the average number of iterations to reach a stopping point was measured — in IPOPT's case due to convergence to a solution, in Optizelle's case due to a stopping condition being met. These averages were taken over the results from five repetitions of the experiment.

7.2 The experiment

These were the settings we used for IPOPT in the experiment:

```
print_level                  5
print_timing_statistics      yes
linear_solver                pardiso
pardiso_matching_strategy complete2x2
dual_inf_tol                 1e-6
compl_inf_tol                1e-6
constr_viol_tol              1e-6
mu_strategy                  monotone
```

These were the settings we used for Optizelle in the experiment:

```
{"Optizelle" : {
      "msg_level" : 1,
      "H_type" : "UserDefined",
      "iter_max" : 5000,
      "delta" : 100,
      "eps_trunc" : 1e-6,
      "eps_grad" : 1e-6,
      "eps_kind" : "Absolute",
      "eps_dx" : 1e-6},
   "Naturals" : {
      "iter" : 14
   }
}
```

although, when measuring the average number of iterations, we set the maximum number of iterations to fifty.

We note that all *tolerance values* were set to 10^{-6}, for both IPOPT and Optizelle. In particular, we set the tolerance of the power flow equations to 10^{-2}.

On one hand, in all cases, Optizelle *did not converge* to a solution that satisfied the power flow equations. On the other hand, in all cases, IPOPT *converged* to a solution that satisfied the power flow equations; moreover, IPOPT took at most fourteen iterations to converge to a valid solution. From this, we can conclude that IPOPT is both faster and more accurate than Optizelle. Indeed, this seems to indicate that the implementation of the primal-dual interior

point method by IPOPT is more performant and robust than that of Optizelle.

Table 1 lists important details we noticed when trying to solve case9 with different contingency sets (`cont`, $\mathscr{G}$), using *Optizelle* (we note that PF stands for *power flow*):

Contingency set	*Maximum PF error*	*Reason for convergence*
`cont = no contingencies`	4.618512×10^{-1}	MaxItersExceeded
`cont = 2`	4.013213	MaxItersExceeded
`cont = (2,3)`	1.549788	StepSmall
`cont = (2,3,5)`	1.549788	StepSmall
`cont = (2,3,5,6)`	1.549788	StepSmall
`cont = (2,3,5,6,8)`	1.549787	StepSmall
`cont = (2,3,5,6,8,9)`	1.549786	StepSmall

Table 1. Optizelle power flow equation evaluation.

where "StepSmall" indicates that Optizelle converged to a solution that satisfied our setting $\epsilon_{\text{dx}} = 10^{-6}$. We note that in the context of our SCOPF problem, the violation of the power flow equations was significant.

For comparison, we list the maximum PF error from IPOPT in Table 2 below:

Contingency set	*Maximum PF error*	*Reason for convergence*
`cont = no contingencies`	$2.9370950116458516 \times 10^{-13}$	Optimal Solution Found
`cont = 2`	$5.6760982025672035 \times 10^{-10}$	Optimal Solution Found
`cont = (2,3)`	$5.7509247364251337 \times 10^{-10}$	Optimal Solution Found
`cont = (2,3,5)`	$2.7187258000438419 \times 10^{-10}$	Optimal Solution Found
`cont = (2,3,5,6)`	$5.6067400722170646 \times 10^{-10}$	Optimal Solution Found
`cont = (2,3,5,6,8)`	$2.4691740763138625 \times 10^{-7}$	Optimal Solution Found
`cont = (2,3,5,6,8,9)`	$9.6350508127507339 \times 10^{-10}$	Optimal Solution Found

Table 2. IPOPT power flow equation evaluation.

Comparing the results from Table 1 and Table 2, we have further proof that IPOPT not only performed better than Optizelle, but it performed better than we required it to. Indeed, the highest evaluation of the constraints by IPOPT was $2.4691740763138625 \times 10^{-7}$, which is five orders of magnitude more accurate than required by our power flow equations tolerance value of 10^{-2}.

In Figures 10-16 below, $f(\mathbf{x})$ denotes the objective function, and $\mathbf{g}(\mathbf{x})$ denotes the constraint functions, i.e., the power flow equations. Each figure contains two plots corresponding to a contingency set with cardinality $|\mathscr{G}| = i$, for $i = 0, ..., 6$. On the left plot, we plot $f(\mathbf{x})$; on the right plot, we plot $\|\nabla f(\mathbf{x})\|$ and $\|\mathbf{g}(\mathbf{x})\|$; both plots are made against the number of iterations. We note that there are two y-axes in the plots on the right-hand side of each figure, with the data corresponding to the colour of the y-axis labels. We represent the data from IPOPT with a continuous line, and the data from Optizelle with a loosely dashed line.

After these figures, we present the average time per iteration in Figure 17.

We will analyze Figures 10-17 after having presented them.

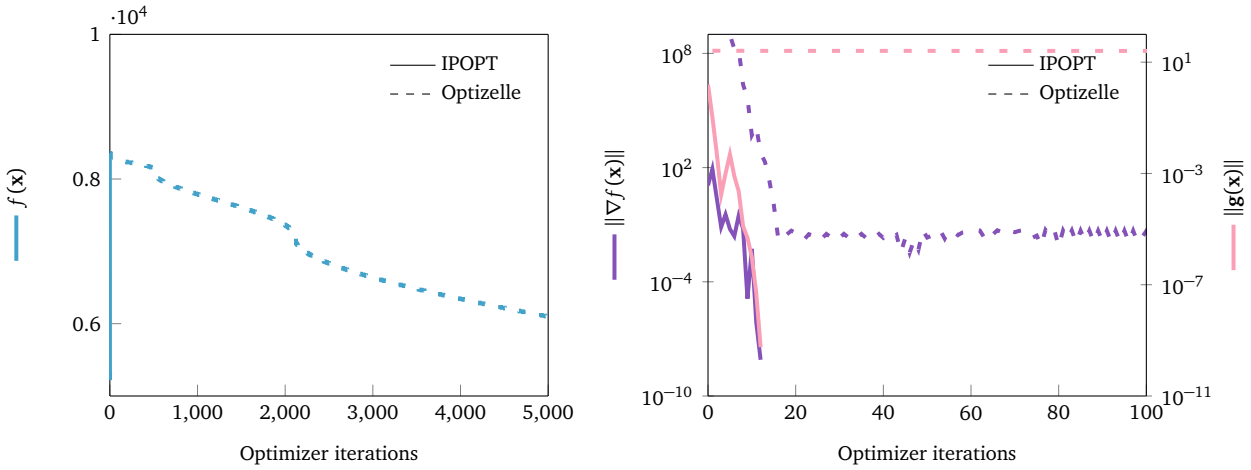

Figure 10. cont = no contingencies, $f(\mathbf{x})$, $\|\nabla f(\mathbf{x})\|$, $\|\mathbf{g}(\mathbf{x})\|$.

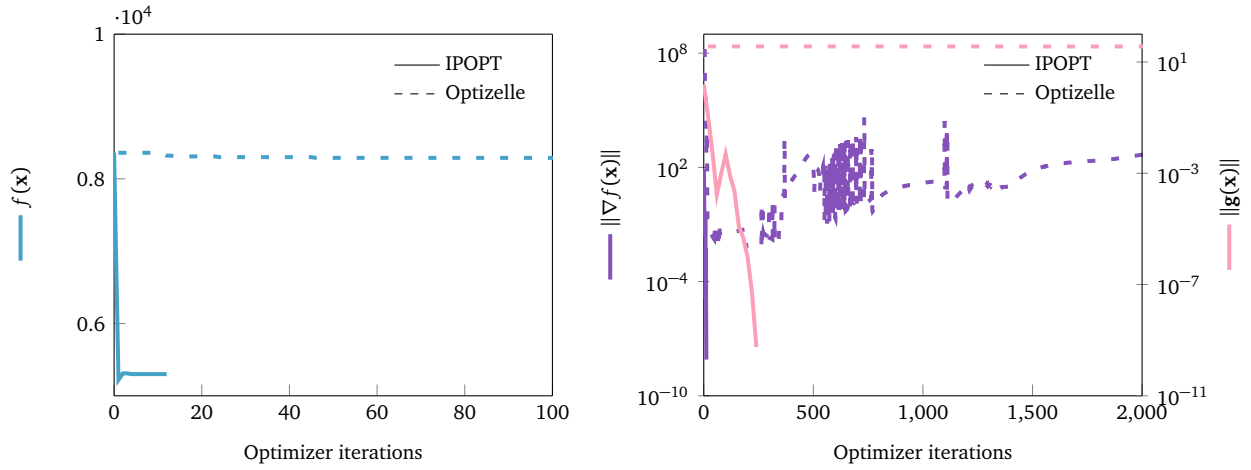

Figure 11. cont = 2, $f(\mathbf{x})$, $\|\nabla f(\mathbf{x})\|$, $\|\mathbf{g}(\mathbf{x})\|$.

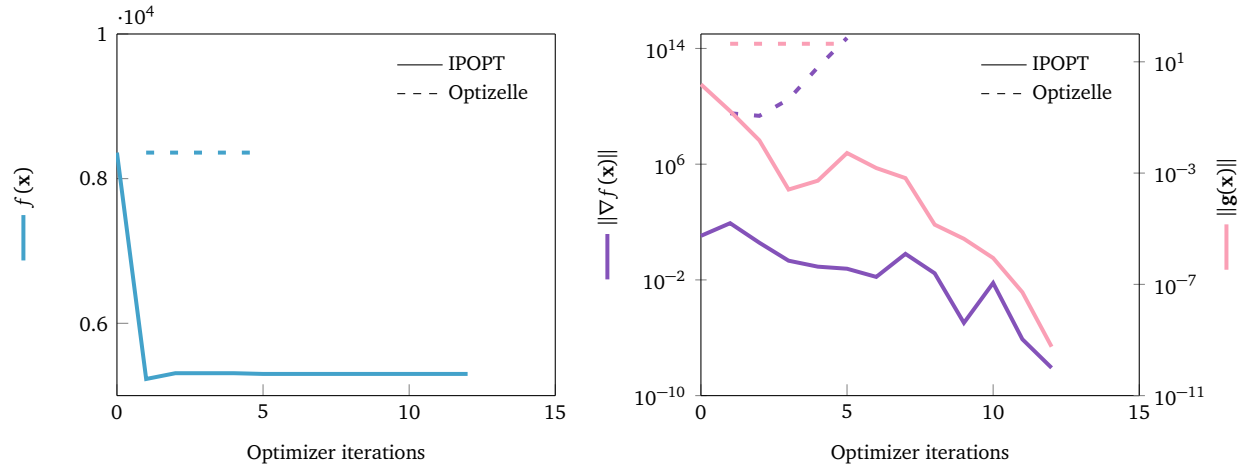

Figure 12. cont = (2, 3), $f(\mathbf{x})$, $\|\nabla f(\mathbf{x})\|$, $\|\mathbf{g}(\mathbf{x})\|$.

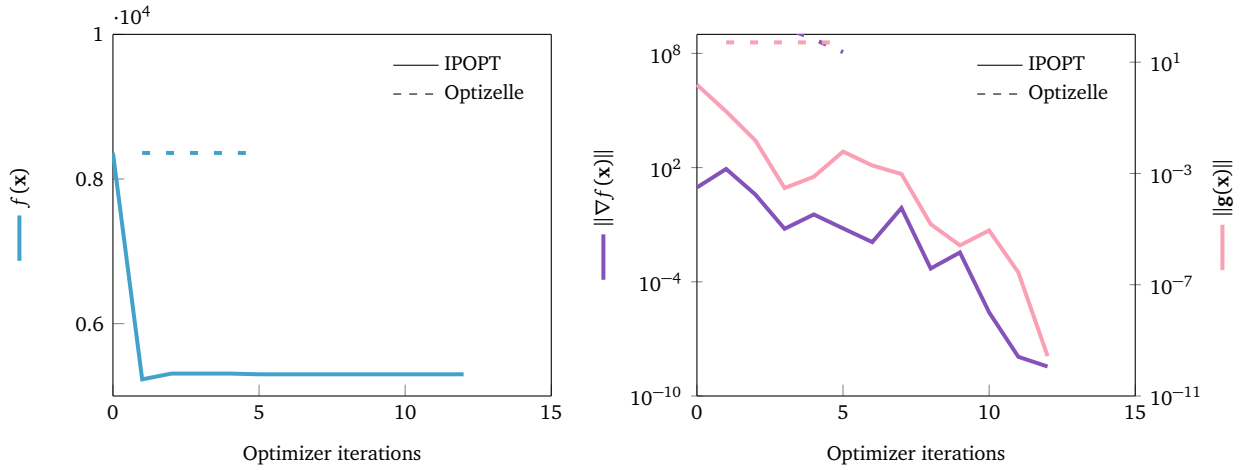

Figure 13. cont $= (2,3,5)$, $f(\mathbf{x})$, $\|\nabla f(\mathbf{x})\|$, $\|\mathbf{g}(\mathbf{x})\|$.

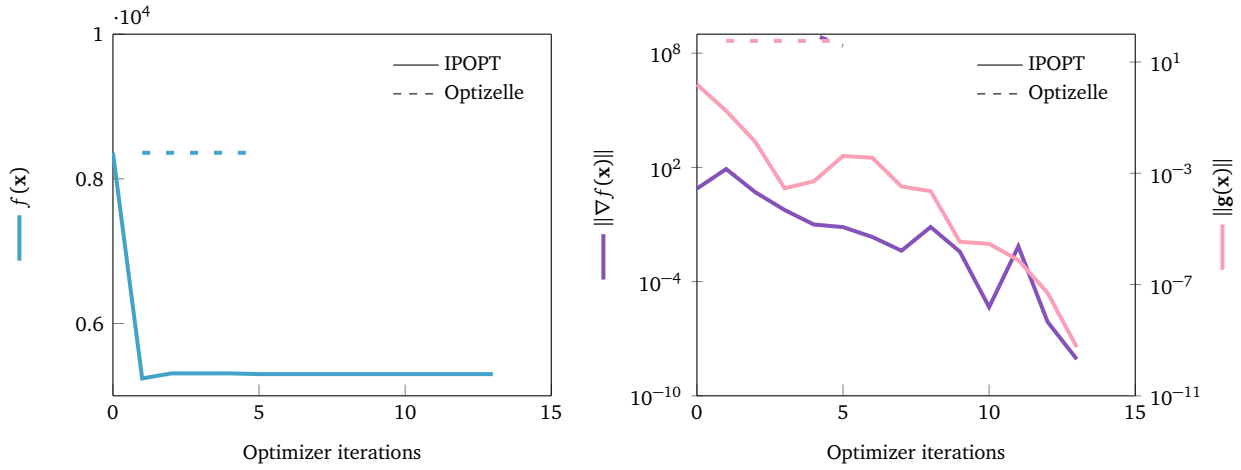

Figure 14. cont $= (2,3,5,6)$, $f(\mathbf{x})$, $\|\nabla f(\mathbf{x})\|$, $\|\mathbf{g}(\mathbf{x})\|$.

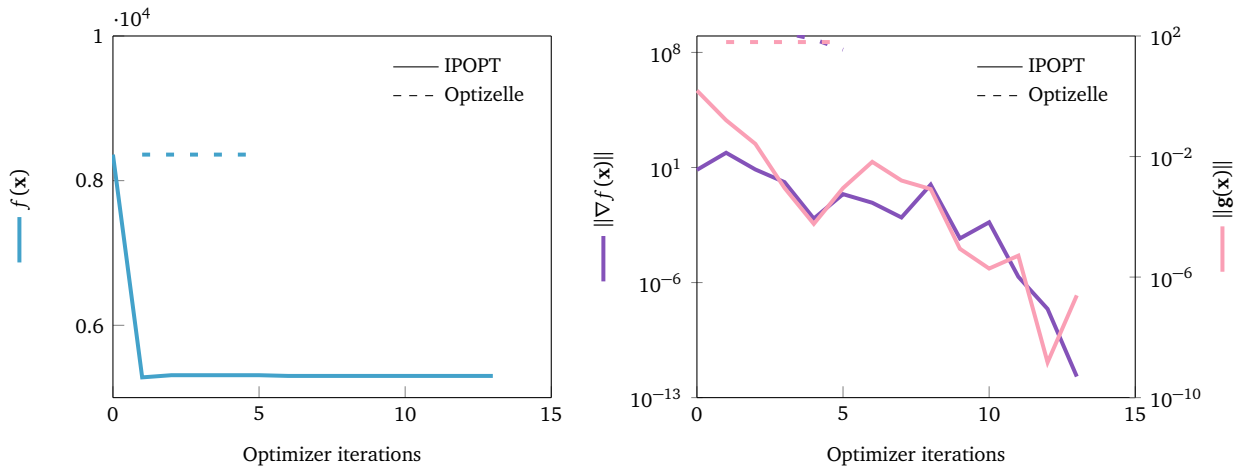

Figure 15. cont $= (2,3,5,6,8)$, $f(\mathbf{x})$, $\|\nabla f(\mathbf{x})\|$, $\|\mathbf{g}(\mathbf{x})\|$.

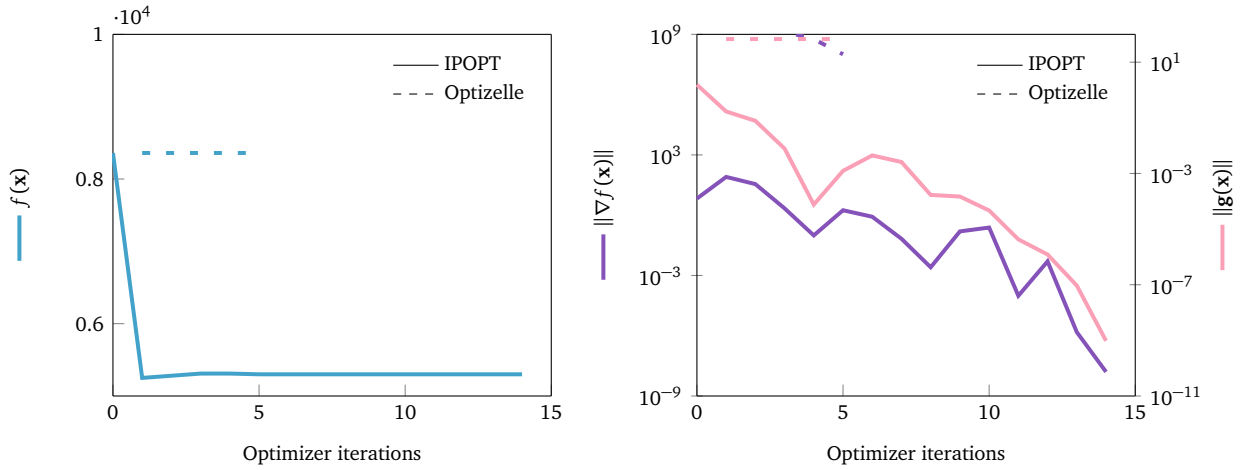

Figure 16. cont $= (2, 3, 5, 6, 8, 9)$, $f(\mathbf{x})$, $\|\nabla f(\mathbf{x})\|$, $\|\mathbf{g}(\mathbf{x})\|$.

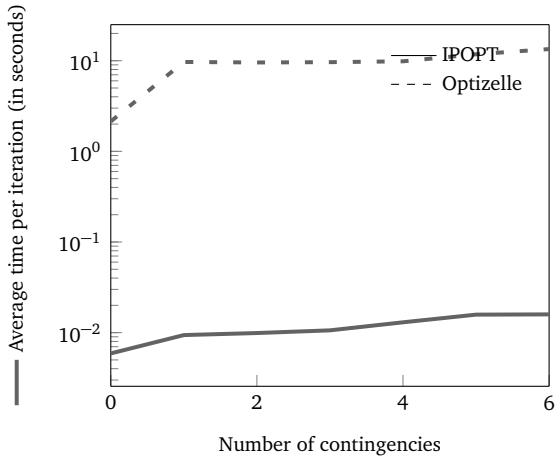

Figure 17. Average time per iteration.

7.3 Analysis

7.3.1 Convergence behaviour

Here, we are shall refer strictly to Figures 10-16.

In all of the figures, we notice that Optizelle did not come close to the minimum objective value found by IPOPT, as can be seen in the plots of $f(\mathbf{x})$. In fact, Optizelle only made significant progress with the objective function in the case of an empty contingency set, as can be seen in Figure 10.

In Figure 10, it appears that Optizelle did not find an acceptable solution given that the gradient $\nabla f(\mathbf{x})$ was relatively small. This suggests that Optizelle may have found a local minimum in this case albeit not an admissible one, given that the power flow equations were violated. In particular, we also note that the gradient $\nabla f(\mathbf{x})$ actually increased in Figure 12, as far as the order of 10^{14}. Moreover, in Figure 11, the gradient is reduced first and then increases, meaning that Optizelle actually moved away from the local minimum it found in that case.

In the cases where Optizelle managed to make progress with the objective function $f(\mathbf{x})$, namely in the cases with no contingencies and with one contingency, it did not manage to reduce the value of the constraints $\mathbf{g}(\mathbf{x})$ to an acceptable level, as can be seen in Figures 10 and 11. In all other cases, Optizelle converged due to a small step size $\|\mathrm{dx}\|$, which suggests it found a solution it qualified as satisfactory given the constraint $\epsilon_{\mathrm{dx}} = 10^{-6}$; unfortunately, we stress once again that this solution was not admissible.

Evidently, IPOPT performed very well. In all cases, it converged to a solution that satisfied the power flow equations very quickly. Moreover, it took IPOPT only one iteration to reach a neighborhood of the optimal solution.

Furthermore, as can be seen in all of the figures, the value of $\|\nabla f(\mathbf{x})\|$ and $\|\mathbf{g}(\mathbf{x})\|$ was within an admissible region, i.e., close to 0.

7.3.2 Average timer per iteration

Referring to Figure 17, it is clear that IPOPT once more performed better than Optizelle, given that the average time per iteration from IPOPT was approximately two orders of magnitude lesser than that of Optizelle's. Given that we used a $\log_{10}$ scale on the y axis, this means that IPOPT was at least one hundred times faster than Optizelle. Moreover, we highlight the fact that IPOPT actually converged to a solution during this time, whereas Optizelle did not. Furthermore, we can see that in Optizelle's case the average time per iteration increased approximately six-fold when comparing the time taken with an empty contingency set and the time taken with a contingency set consisting of six contingencies. In contrast, the average time per iteration increased approximately three-fold in IPOPT's case when comparing the time taken with an empty contingency set and the time taken with a contingency set of consisting of six contingencies.

These results suggest that the average time per iteration with respect to a growing contingency set is relatively constant in IPOPT's case but not in Optizelle's case, which leads to the belief that IPOPT is more scalable to bigger problems than Optizelle.

All in all, based on the convergence behaviour and the average time per iteration of both frameworks, the results further highlight that IPOPT appears to be more robust and efficient than Optizelle.

8 Conclusion

In this thesis, the *IPOPT* and *Optizelle* frameworks have been explored in the context of solving the security constarined power flow (SCOPF) problem. We went through a brief description of the SCOPF problem and its nature, as well as exploring the concepts of unconstrained optimization, constrained optimization, and the primal-dual interior point method — the primal-dual interior point method being the main actor behind IPOPT's and Optizelle's optimization algorithms for constrained problems. We also briefly explored the software used to deliver the results in this thesis, namely MATPOWER, IPOPT, and Optizelle. We then presented the methodology of our experiments as well as the results arising from them. A clear difference in performance between IPOPT and Optizelle manifested. On one hand, Optizelle proved to be very slow at solving the problem, and failed to converge to an admissible solution in all cases. Moreover, it only reduced the value of the objective function in a significant way in one case, without ever reducing the value of the power flow equations to a tolerable level. On the other hand, IPOPT performed very well — it converged to the solution within a reasonable time and number of iterations, and delivered results that were far more precise than required by our tolerance values. Moreover, the average time per iteration observed for IPOPT and Optizelle suggest that IPOPT is scalable to bigger problems whereas Optizelle does not appear to be so. The reason for this is that the test power grid we used, *case 9*, is relatively small. Furthermore, even in this small power grid, Optizelle failed to provide admissible results and the average time per iteration did not appear to be relatively constant given a growing problem resulting from adding further contingencies. In the context of the SCOPF problem and the results acquired in this thesis, we can conclude that IPOPT is more performant at solving the SCOPF problem than Optizelle.

9 Appendix A

Where necessary, lines are wrapped in the code excerpts displayed in this section. A red arrow is used to mark the beginning of a broken line to emphasize the line break [10].

The following illustrates how we built the sparsity structure of the *Jacobian of the constraints* $\nabla c(x)$ required by IPOPT, as mentioned in 6.1.6 (Problem structure).

```
%% Jacobian of constraints
Js = sparse(0,0);

for i = 0:ns-1
    %update Cb to reflect the bus connectivity caused by contingency
    Cb = Cb_nominal;
    if (model.cont(i+1) > 0)
        c = model.cont(i+1);
```

```
        f = branch(c, F_BUS);                           %% "from" bus
        t = branch(c, T_BUS);                           %% "to" bus
        Cb(f,t) = 0;
        Cb(t,f) = 0;
    end

    %update Cl to reflect the contingency
    Cl2 = Cl2_nominal;
    if (model.cont(i+1) > 0)
        c = model.cont(i+1);
        Cl2(c, :) = 0;
    end

    % Jacobian wrt local variables
    %     dVa  dVm(nPV)   dQg   dPg(REF)   <- local variables for each scenario
    %    | Cb     Cb'      0     Cg' | ('one' at row of REF bus, otherwise zeros)
    %    |                           |
    %    | Cb     Cb'      Cg     0  |
    %    |                           |
    %    | Cl     Cl'      0      0  |
    %    |                           |
    %    | Cl     Cl'      0      0  |
    Js_local = [
        Cb      Cb(:, nPVbus_idx)    sparse(nb, ng)   Cg(:, REFgen_idx);
        Cb      Cb(:, nPVbus_idx)     Cg              sparse(nb, 1);
        Cl2     Cl2(:, nPVbus_idx)   sparse(nl2, ng+1);
        Cl2     Cl2(:, nPVbus_idx)   sparse(nl2, ng+1);
    ];
    % Jacobian wrt global variables
    %     dVm(PV) dPg(nREF)   <- global variables for all scenarios
    %    | Cb'        Cg'  | ('one' at row of REF bus, otherwise zeros)
    %    |                 |
    %    | Cb'         0   |
    %    |                 |
    %    | Cl'         0   |
    %    |                 |
    %    | Cl'         0   |
    Js_global = [
     Cb(:, PVbus_idx)  Cg(:, nREFgen_idx);
     Cb(:, PVbus_idx)  sparse(nb, ng-1);
     Cl2(:, PVbus_idx) sparse(nl2, ng-1);
     Cl2(:, PVbus_idx) sparse(nl2, ng-1);
    ];

    Js = [Js;
          sparse(size(Js_local,1), i size(Js_local,2)) Js_local sparse(size(Js_local,1), (ns-1-i) size(
              ↪ Js_local,2)) Js_global];

%     Js = kron(eye(ns), Js_local); %replicate jac. w.r.t local variables
%     Js = [Js kron(ones(ns,1), Js_global)]; % replicate and append jac w.r.t global variables
end
Js = [Js; A]; %append linear constraints
```

The following illustrates how we built the sparsity structure of the *Hessian of the Lagrangian function* required by IPOPT, as mentioned in 6.1.6 (Problem structure).

```
%% Hessian of lagrangian Hs = f(x)_dxx + c(x)_dxx + h(x)_dxx
Hs = sparse(0,0);
Hs_gl = sparse(0,0);

for i = 0:ns-1
    %update Cb to reflect the bus connectivity caused by contingency
    Cb = Cb_nominal;
    if (model.cont(i+1) > 0)
        c = model.cont(i+1);
        f = branch(c, F_BUS);                           %% "from" bus
        t = branch(c, T_BUS);                           %% "to" bus
        Cb(f,t) = 0;
        Cb(t,f) = 0;
    end

    %update Cl to reflect the contingency
    Cl2 = Cl2_nominal;
```

```
    if (model.cont(i+1) > 0)
        c = model.cont(i+1);
        Cl2(c, :) = 0;
    end

    %--- hessian wrt. scenario local variables ---

    %           dVa  dVm(nPV)  dQg dPg(REF)
    % dVa      | Cb     Cb'     0     0  |
    %          |                         |
    % dVm(nPV)| Cb'    Cb'     0     0  |
    %          |                         |
    % dQg      |  0      0      0     0  |
    %          |                         |
    % dPg(REF)|  0      0      0    Cg' | (only nominal case has Cg', because it is used in cost
        ↪ function)

    Hs_ll =[
        Cb                Cb(:, nPVbus_idx)          sparse(nb, ng+1);%assuming 1 REF gen
        Cb(nPVbus_idx,:)  Cb(nPVbus_idx, nPVbus_idx) sparse(length(nPVbus_idx), ng+1);
                   sparse(ng+1, nb+length(nPVbus_idx)+ng+1);
    ];
    %replicate hess. w.r.t local variables
    %Hs = kron(eye(ns), Hs_ll);

    %set d2Pg(REF) to 1 in nominal case
    if (i==0)
        Hs_ll(nb+length(nPVbus_idx)+ng+1, nb+length(nPVbus_idx)+ng+1) = 1;
    end

    %--- hessian w.r.t local-global variables ---

    %           dVm(PV)  dPg(nREF)
    % dVa      | Cb'     0    |
    %          |              |
    % dVm(nPV)| Cb'     0    |
    %          |              |
    % dQg      |  0      0    |
    %          |              |
    % dPg(REF)|  0      0    |
    Hs_lg  = [
       Cb(:, PVbus_idx)           sparse(nb, ng-1);
       Cb(nPVbus_idx, PVbus_idx)  sparse(length(nPVbus_idx), ng-1);
       sparse(ng+length(REFgen_idx), length(PVbus_idx)+ng-1)
    ];
    %Hs_lg = kron(ones(ns,1), Hs_lg);

    Hs = [Hs;
          sparse(size(Hs_ll,1), i size(Hs_ll,2)) Hs_ll sparse(size(Hs_ll,1), (ns-1-i) size(Hs_ll,2))
              ↪ Hs_lg];
    Hs_gl = [Hs_gl Hs_lg'];

end

% --- hessian w.r.t global variables ---

%        dVm(PV)  dPg(nREF)
% dVm(PV)  | Cb'  0   |
%          |          |
% dPg(nREF)| 0  f_xx' |
Hs_gg =[
    Cb_nominal(PVbus_idx, PVbus_idx)          sparse(length(PVbus_idx), ng-1);
    sparse(ng-1, length(PVbus_idx))               eye(ng-1);
];

% --- Put together local and global hessian ---
% local hessians sits at (1,1) block
% hessian w.r.t global variables is appended to lower right corner (2,2)
% and hessian w.r.t local/global variables to the (1,2) and (2,1) blocks
%        (l)      (g)
% (l) | Hs_ll    Hs_lg |
%     |                |
% (g) | Hs_gl    Hs_gg |
```

```
Hs = [Hs;
      Hs_gl   Hs_gg];


Hs = tril(Hs);
```

The following displays the definition of the functions required by IPOPT, as mentioned in 6.1.7 (Evaluation of problem functions).

```
  function f = objective(x, d)
  mpc = get_mpc(d.om);
  ns = size(d.cont, 1);           %% number of scenarios (nominal + ncont)

  % use nominal case to evaluate cost fcn (only pg/qg are relevant)
  idx_nom = d.index.getGlobalIndices(mpc, ns, 0);
  [VAscopf, VMscopf, PGscopf, QGscopf] = d.index.getLocalIndicesSCOPF(mpc);

  f = opf_costfcn(x(idx_nom([VAscopf VMscopf PGscopf QGscopf])), d.om);
```

```
  function grad = gradient(x, d)
  mpc = get_mpc(d.om);
  ns = size(d.cont, 1);           %% number of scenarios (nominal + ncont)

  %evaluate grad of nominal case
  idx_nom = d.index.getGlobalIndices(mpc, ns, 0);
  [VAscopf, VMscopf, PGscopf, QGscopf] = d.index.getLocalIndicesSCOPF(mpc);
  [VAopf, VMopf, PGopf, QGopf] = d.index.getLocalIndicesOPF(mpc);

  [f, df, d2f] = opf_costfcn(x(idx_nom([VAscopf VMscopf PGscopf QGscopf])), d.om);

  grad = zeros(size(x,1),1);
  grad(idx_nom(PGscopf)) = df(PGopf); %nonzero only nominal case Pg
```

```
function constr = constraints(x, d)
mpc = get_mpc(d.om);
nb = size(mpc.bus, 1);          %% number of buses
ng = size(mpc.gen, 1);          %% number of gens
nl = size(mpc.branch, 1);       %% number of branches
ns = size(d.cont, 1);           %% number of scenarios (nominal + ncont)
NCONSTR = 2 nb + 2 nl;

constr = zeros(ns (NCONSTR), 1);

[VAscopf, VMscopf, PGscopf, QGscopf] = d.index.getLocalIndicesSCOPF(mpc);
[VAopf, VMopf, PGopf, QGopf] = d.index.getLocalIndicesOPF(mpc);

for i = 0:ns-1
    cont = d.cont(i+1);
    idx = d.index.getGlobalIndices(mpc, ns, i);
    [Ybus, Yf, Yt] = makeYbus(mpc.baseMVA, mpc.bus, mpc.branch, cont);
    [hn_local, gn_local] = opf_consfcn(x(idx([VAscopf VMscopf PGscopf QGscopf])), d.om, Ybus, Yf, Yt, d.
        ↪ mpopt, d.il);
    constr(i (NCONSTR) + (1:NCONSTR)) = [gn_local; hn_local];
end

if ~isempty(d.A)
    constr = [constr; d.A x]; %append linear constraints
end
```

```
function J = jacobian(x, d)
mpc = get_mpc(d.om);
nb = size(mpc.bus, 1);          %% number of buses
nl = size(mpc.branch, 1);       %% number of branches
ns = size(d.cont, 1);           %% number of scenarios (nominal + ncont)
NCONSTR = 2 nb + 2 nl;          %% number of constraints (eq + ineq)

J = sparse(ns (NCONSTR), size(x,1));

% get indices of REF gen and PV bus
[REFgen_idx, nREFgen_idx] = d.index.getREFgens(mpc);
```

```
[PVbus_idx, nPVbus_idx] = d.index.getXbuses(mpc,2);%2==PV

[VAscopf, VMscopf, PGscopf, QGscopf] = d.index.getLocalIndicesSCOPF(mpc);
[VAopf, VMopf, PGopf, QGopf] = d.index.getLocalIndicesOPF(mpc);

for i = 0:ns-1
    %compute local indices
    idx = d.index.getGlobalIndices(mpc, ns, i);

    cont = d.cont(i+1);
    [Ybus, Yf, Yt] = makeYbus(mpc.baseMVA, mpc.bus, mpc.branch, cont);
    [hn, gn, dhn, dgn] = opf_consfcn(x(idx([VAscopf VMscopf PGscopf QGscopf])), d.om, Ybus, Yf, Yt, d.
        ↪ mpopt, d.il);
    dgn = dgn';
    dhn = dhn';

    %jacobian wrt local variables
    J(i NCONSTR + (1:NCONSTR), idx([VAscopf VMscopf(nPVbus_idx) QGscopf PGscopf(REFgen_idx)])) ...
       = [dgn(:,[VAopf VMopf(nPVbus_idx) QGopf PGopf(REFgen_idx)]);...
          dhn(:,[VAopf VMopf(nPVbus_idx) QGopf PGopf(REFgen_idx)])];
    %jacobian wrt global variables
    J(i NCONSTR + (1:NCONSTR), idx([VMscopf(PVbus_idx) PGscopf(nREFgen_idx)])) ...
       = [dgn(:, [VMopf(PVbus_idx) PGopf(nREFgen_idx)]);...
          dhn(:, [VMopf(PVbus_idx) PGopf(nREFgen_idx)])];
end
J = [J; d.A]; %append Jacobian of linear constraints
```

```
  function H = hessian(x, sigma, lambda, d)
  mpc = get_mpc(d.om);
  nb = size(mpc.bus, 1);          %% number of buses
  ng = size(mpc.gen, 1);          %% number of gens
  nl = size(mpc.branch, 1);       %% number of branches
  ns = size(d.cont, 1);           %% number of scenarios (nominal + ncont)
  NCONSTR = 2 nb + 2 nl;

  H = sparse(size(x,1), size(x,1));

  % get indices of REF gen and PV bus
  [REFgen_idx, nREFgen_idx] = d.index.getREFgens(mpc);
  [PVbus_idx, nPVbus_idx] = d.index.getXbuses(mpc,2);%2==PV

  [VAscopf, VMscopf, PGscopf, QGscopf] = d.index.getLocalIndicesSCOPF(mpc);
  [VAopf, VMopf, PGopf, QGopf] = d.index.getLocalIndicesOPF(mpc);

  for i = 0:ns-1
      %compute local indices and its parts
      idx = d.index.getGlobalIndices(mpc, ns, i);

      cont = d.cont(i+1);
      [Ybus, Yf, Yt] = makeYbus(mpc.baseMVA, mpc.bus, mpc.branch, cont);

      lam.eqnonlin   = lambda(i NCONSTR + (1:2 nb));
      lam.ineqnonlin = lambda(i NCONSTR + 2 nb + (1:2 nl));
      H_local = opf_hessfcn(x(idx([VAscopf VMscopf PGscopf QGscopf])), lam, sigma, d.om, Ybus, Yf, Yt, d
          ↪ .mpopt, d.il);

      % H_ll (PG_ref relevant only in nominal case, added to global part)
       H(idx([VAscopf VMscopf(nPVbus_idx) QGscopf]), idx([VAscopf VMscopf(nPVbus_idx) QGscopf])) =...
              H_local([VAopf VMopf(nPVbus_idx) QGopf], [VAopf VMopf(nPVbus_idx) QGopf]);

      % H_lg and H_gl (PG parts are implicitly zero, could leave them out)
      H(idx([VAscopf VMscopf(nPVbus_idx) QGscopf PGscopf(REFgen_idx)]), idx([VMscopf(PVbus_idx) PGscopf(
          ↪ nREFgen_idx)])) = ...
              H_local([VAopf VMopf(nPVbus_idx) QGopf PGopf(REFgen_idx)], [VMopf(PVbus_idx) PGopf(
                  ↪ nREFgen_idx)]);
      H(idx([VMscopf(PVbus_idx) PGscopf(nREFgen_idx)]), idx([VAscopf VMscopf(nPVbus_idx) QGscopf PGscopf
          ↪ (REFgen_idx)])) = ...
              H_local([VMopf(PVbus_idx) PGopf(nREFgen_idx)], [VAopf VMopf(nPVbus_idx) QGopf PGopf(
                  ↪ REFgen_idx)]);

      % H_gg hessian w.r.t global variables (and PG_ref_0)
      if i == 0
```

```
        % H_pg at non-reference gens, these are global variables
        H(idx([PGscopf(nREFgen_idx)]), idx([PGscopf(nREFgen_idx)])) = ...
            H_local([PGopf(nREFgen_idx)], [PGopf(nREFgen_idx)]);

        % H_pgref is local variable for nominal scenario, but used in f()
        H(idx([PGscopf(REFgen_idx)]), idx([PGscopf(REFgen_idx)])) = ...
            H_local([PGopf(REFgen_idx)], [PGopf(REFgen_idx)]);
    end

    %each scenario contributes to hessian w.r.t global VM variables at PV buses
    H(idx([VMscopf(PVbus_idx)]), idx([VMscopf(PVbus_idx)])) = ...
        H(idx([VMscopf(PVbus_idx)]), idx([VMscopf(PVbus_idx)])) + ...
        H_local([VMopf(PVbus_idx)], [VMopf(PVbus_idx)]);
end

H = tril(H);
```

10 Appendix B

Where necessary, lines are wrapped in the code excerpts displayed in this section. A red arrow is used to mark the beginning of a broken line to emphasize the line break [10].

The following illustrates the definition of the functions required by Optizelle, as mentioned in 6.2.3 (Setting up Optizelle). We note that we included certain checks and print statements in the code in order to verify that the inequality constraints were strictly satisfied.

```
%% Define objective function.
function self = MyObj(myauxdata)
% Evaluation
self.eval = @(x) objective(x, myauxdata);

% Gradient
self.grad = @(x) gradient(x, myauxdata);

% Hessian-vector product
self.hessvec = @(x, dx) hessvec(x, dx, myauxdata);

% Helper functions.
   function f = objective(x, myauxdata)
      % Extract data.
      idx_nom = myauxdata.idx_nom;
      model = myauxdata.model;
      om = myauxdata.om;
      mpc = myauxdata.mpc;

      % Indices of local OPF solution vector x = [VA VM PG QG].
      [VAscopf, VMscopf, PGscopf, QGscopf] = model.index.getLocalIndicesSCOPF(mpc);

      f = opf_costfcn(x(idx_nom([VAscopf VMscopf PGscopf QGscopf])), om);

      if find(isnan(f), 1)
         disp('MyObj: f is NaN')
      end
   end

   function grad = gradient(x, myauxdata)
      % Extract data.
      idx_nom = myauxdata.idx_nom;
      model = myauxdata.model;
      om = myauxdata.om;
      mpc = myauxdata.mpc;

      % Indices of local OPF solution vector x = [VA VM PG QG].
      [VAscopf, VMscopf, PGscopf, QGscopf] = model.index.getLocalIndicesSCOPF(mpc);
      [VAopf, VMopf, PGopf, QGopf] = model.index.getLocalIndicesOPF(mpc);

      [f, df, d2f] = opf_costfcn(x(idx_nom([VAscopf VMscopf PGscopf QGscopf])), om);

      % Nonzero only nominal case Pg.
```

```
        grad = zeros(size(x,1),1);
        grad(idx_nom(PGscopf)) = df(PGopf);

        if find(isnan(grad), 1)
           disp('MyObj:_NaN_found_in_grad')
        end

    end

    function res = hessvec(x, dx, myauxdata)
        % Extract data.
        idx_nom = myauxdata.idx_nom;
        model = myauxdata.model;
        om = myauxdata.om;
        mpc = myauxdata.mpc;

        % Indices of local OPF solution vector x = [VA VM PG QG].
        [VAscopf, VMscopf, PGscopf, QGscopf] = model.index.getLocalIndicesSCOPF(mpc);
        [VAopf, VMopf, PGopf, QGopf] = model.index.getLocalIndicesOPF(mpc);

        [f, df, d2f] = opf_costfcn(x(idx_nom([VAscopf VMscopf PGscopf QGscopf])), om);

        % Nonzero only nominal case Pg.
        H = sparse(size(x,1),size(x,1));
        H(idx_nom(PGscopf), idx_nom(PGscopf)) = d2f(PGopf, PGopf);

        if find(isnan(dx), 1)
           disp('MyObj:_NaN_found_in_dx')
        end

        res = H   dx;

        if find(isnan(res), 1)
           disp('MyObj:_NaN_found_in_hessvec_result')
        end
    end
end
```

```
%% Define equality constraints.
function self = MyEq(myauxdata)

% y=g(x)
self.eval = @(x) constraints(x, myauxdata);

% y=g'(x)dx
self.p = @(x,dx) jacobvec(x, dx, myauxdata);

% xhat=g'(x)*dy
self.ps = @(x,dy) jacobvec(x, dy, myauxdata);

% xhat=(g''(x)dx)*dy
self.pps = @(x,dx,dy) hessian(x, myauxdata, dy)   dx;

% Helper functions.
    function constr = constraints(x, myauxdata)
        % Extract data.
        om = myauxdata.om;
        mpc = myauxdata.mpc;
        model = myauxdata.model;
        mpopt = myauxdata.mpopt;
        il = myauxdata.il;
        lenx_no_s = myauxdata.lenx_no_s; % length of x without slack variables.
        NEQ = myauxdata.NEQ; % number of equality constraints.
        withSlacks = myauxdata.withSlacks;

        nb = size(mpc.bus, 1);     %% number of buses
        nl = size(mpc.branch, 1);  %% number of branches
        ns = size(model.cont, 1);  %% number of scenarios (nominal + ncont)

        constr = zeros(ns (NEQ), 1);

        [VAscopf, VMscopf, PGscopf, QGscopf] = model.index.getLocalIndicesSCOPF(mpc);
```

```
35
36          for i = 0:ns-1
37             cont = model.cont(i+1);
38             idx = model.index.getGlobalIndices(mpc, ns, i);
39
40             [Ybus, Yf, Yt] = makeYbus(mpc.baseMVA, mpc.bus, mpc.branch, cont);
41             [hn_local, gn_local] = opf_consfcn(x(idx([VAscopf VMscopf PGscopf QGscopf])), om, Ybus, Yf,
                   ↪ Yt, mpopt, il);
42
43             if withSlacks
44                % Extract slack variable(s) s from x.
45                % s = [s0, s1, ... , sNS]
46                s = x(lenx_no_s + i 2 nl + (1:2 nl));
47
48                % Since h(x) <= 0 in Matpower, -h(x) - s = 0, s >= 0
49                % as required by Optizelle.
50                constr(i (NEQ) + (1:NEQ)) = [gn_local; -hn_local - s];
51             else
52                constr(i (NEQ) + (1:NEQ)) = gn_local;
53             end
54          end
55
56
57          %% Test for Inf, -Inf, and NaN in constr.
58          if find(constr == Inf, 1)
59             disp('MyEq: Inf found in constr')
60          end
61
62          if find(constr == -Inf, 1)
63             disp('MyEq: -Inf found in constr')
64          end
65
66          if find(isnan(constr), 1)
67             disp('MyEq: NaN found in constr')
68          end
69       end
70
71       function jvec = jacobvec(x, d, myauxdata)
72          % Extract data.
73          om = myauxdata.om;
74          mpc = myauxdata.mpc;
75          model = myauxdata.model;
76          mpopt = myauxdata.mpopt;
77          il = myauxdata.il;
78          lenx_no_s = myauxdata.lenx_no_s; % length of x without slack variables.
79          NEQ = myauxdata.NEQ; % number of equality constraints.
80          withSlacks = myauxdata.withSlacks;
81
82
83          nb = size(mpc.bus, 1);     %% number of buses
84          nl = size(mpc.branch, 1);  %% number of branches
85          ns = size(model.cont, 1);     %% number of scenarios (nominal + ncont)
86
87          % get indices of REF gen and PV bus
88          [REFgen_idx, nREFgen_idx] = model.index.getREFgens(mpc);
89          [PVbus_idx, nPVbus_idx] = model.index.getXbuses(mpc,2);%2==PV
90
91          [VAscopf, VMscopf, PGscopf, QGscopf] = model.index.getLocalIndicesSCOPF(mpc);
92          [VAopf, VMopf, PGopf, QGopf] = model.index.getLocalIndicesOPF(mpc);
93
94          % -Id. Placed in the lower-right "corner" of each scenario's
95          % Jacobian matrix.
96          neg_identity = -sparse(eye(2 nl, 2 nl));
97
98          J = sparse(ns NEQ, length(x));
99
100         for i = 0:ns-1
101            idx = model.index.getGlobalIndices(mpc, ns, i);
102
103            cont = model.cont(i+1);
104            [Ybus, Yf, Yt] = makeYbus(mpc.baseMVA, mpc.bus, mpc.branch, cont);
105            [hn, gn, dhn, dgn] = opf_consfcn(x(idx([VAscopf VMscopf PGscopf QGscopf])), om, Ybus, Yf, Yt,
                   ↪  mpopt, il);
```

```
            % Transpose since opf_consfcn transposed solutions.
            % Take negative since Matpower requires h(x) <= 0 but
            % Optizelle requires h(x) >= 0.
            dhn = -dhn';
            dgn = dgn';

            if withSlacks
                %jacobian wrt local variables
                J(i NEQ + (1:NEQ), idx([VAscopf VMscopf(nPVbus_idx) QGscopf PGscopf(REFgen_idx)])) = [dgn
                    ↪ (:,[VAopf VMopf(nPVbus_idx) QGopf PGopf(REFgen_idx)]);...
                   dhn(:,[VAopf VMopf(nPVbus_idx) QGopf PGopf(REFgen_idx)])];
                %jacobian wrt global variables
                J(i NEQ + (1:NEQ), idx([VMscopf(PVbus_idx) PGscopf(nREFgen_idx)])) = [dgn(:, [VMopf(
                    ↪ PVbus_idx) PGopf(nREFgen_idx)]);...
                   dhn(:, [VMopf(PVbus_idx) PGopf(nREFgen_idx)])];

            else
                %jacobian wrt local variables
                J(i NEQ + (1:NEQ), idx([VAscopf VMscopf(nPVbus_idx) QGscopf PGscopf(REFgen_idx)])) = dgn
                    ↪ (:,[VAopf VMopf(nPVbus_idx) QGopf PGopf(REFgen_idx)]);

                %jacobian wrt global variables
                J(i NEQ + (1:NEQ), idx([VMscopf(PVbus_idx) PGscopf(nREFgen_idx)])) = dgn(:, [VMopf(
                    ↪ PVbus_idx) PGopf(nREFgen_idx)]);
            end

            % Set corner of Jacobian of this scenario to -Id.
            % Number of rows in dgn is 2*nb; in dhn it is 2*nl.
            % Number of columns in dgn, and dhn, is lenx_no_s.
            % Structure of Jacobian in scenario i is:
            % J = [dgn, 0; dhn, -Id]. Hence why we use following row and
            % column indices.
            % Note: -Id has dimensions 2*nl by 2*nl.
            %
            % Considering the system of slack variables, we have
            % [g(x); h(x) - s] for the equality constraints.
            % As we can see, we have 0s in the upper right corner of J since the
            % partial derivative of the constraints g(x) w.r.t. the slack variables
            % is 0, since g(x) does not at all depend on s.
            % Similarly, we have -Id in the specified (lower-right) corner
            % given the constraints w.r.t. the slack variables are
            % h(x) - s: taking the partial derivative of h(x) - s w.r.t. the slack
            % variables yields the -Id matrix.
            if withSlacks
                J(i NEQ + 2 nb + (1:2 nl), lenx_no_s + i 2 nl + (1:2 nl)) = neg_identity;
            end
        end

        dType = 0;
        if (size(d, 1) == size(x, 1))  % case: d == dx
            dType = 'dx';
            jvec = J   d;
        elseif (size(d, 1) == ns NEQ) % case: d == dy
            jvec = J'   d;
            dType = 'dy';
        end

        %% Test for Inf, -Inf, and NaN in d.
        if find(d == Inf)
            fprintf('MyEq:_Inf_found_in_%s\n', dType);
        end

        if find(d == -Inf)
            fprintf('MyEq:_-Inf_found_in_%s\n', dType);
        end

        if find(isnan(d))
            fprintf('MyEq:_NaN_found_in_%s\n', dType);
        end

        %% Test for Inf, -Inf, and NaN in jvec.
        if find(jvec == Inf, 1)
```

```
            disp('MyEq: Inf found in jvec')
        end

        if find(jvec == -Inf, 1)
            disp('MyEq: -Inf found in jvec')
        end

        if find(isnan(jvec), 1)
            disp('MyEq: NaN found in jvec')
        end
    end

    function H = hessian(x, myauxdata, dy)
        % Extract data.
        il = myauxdata.il;
        om = myauxdata.om;
        mpc = myauxdata.mpc;
        model = myauxdata.model;
        mpopt = myauxdata.mpopt;
        NEQ = myauxdata.NEQ; % number of equality constraints.
        NINEQ = myauxdata.NINEQ;
        withSlacks = myauxdata.withSlacks;

        nb = size(mpc.bus, 1);          %% number of buses
        nl = size(mpc.branch, 1);       %% number of branches
        ns = size(model.cont, 1);           %% number of scenarios (nominal + ncont)

        H = sparse(size(x,1), size(x,1));

        % get indices of REF gen and PV bus
        [REFgen_idx, nREFgen_idx] = model.index.getREFgens(mpc);
        [PVbus_idx, nPVbus_idx] = model.index.getXbuses(mpc,2);%2==PV

        [VAscopf, VMscopf, PGscopf, QGscopf] = model.index.getLocalIndicesSCOPF(mpc);
        [VAopf, VMopf, PGopf, QGopf] = model.index.getLocalIndicesOPF(mpc);

        sigma = 0;

        for i = 0:ns-1
            %compute local indices and its parts
            idx = model.index.getGlobalIndices(mpc, ns, i);

            cont = model.cont(i+1);
            [Ybus, Yf, Yt] = makeYbus(mpc.baseMVA, mpc.bus, mpc.branch, cont);

            % c(x) = [gn0; hn0 - s0; gn1; hn1 - s1; ... ; gnNs; hnNs - sNs]
            % (Ns is the number of scenarios)
            % where gn corresponds to equality constraints, hn corresponds to
            % inequality constraints.
            %
            % The lagrange multipliers in the dy will be composed accordingly:
            % dy = [lamEq0; lamIneq0; lamEq1; lamIneq1; ... ; lamEqNs; lamIneqNs]
            % Hence why we need to extract the lagrange multipliers as
            % follows.
            lam.eqnonlin = dy(i NEQ + (1:2 nb), 1);

            if withSlacks
                lam.ineqnonlin = dy(i NEQ + 2 nb + (1:2 nl), 1);
            else
                lam.ineqnonlin = zeros(NINEQ, 1);
            end

            % Take negative since Matpower requires h(x) <= 0 but
            % Optizelle requires h(x) >= 0.
            lam.ineqnonlin = -lam.ineqnonlin;

            H_local = opf_hessfcn(x(idx([VAscopf VMscopf PGscopf QGscopf])), lam, sigma, om, Ybus, Yf, Yt
                ↪ , mpopt, il);

            % H_ll (PG_ref relevant only in nominal case, added to global part)
            H(idx([VAscopf VMscopf(nPVbus_idx) QGscopf]), idx([VAscopf VMscopf(nPVbus_idx) QGscopf]))
                ↪ =...
                H_local([VAopf VMopf(nPVbus_idx) QGopf], [VAopf VMopf(nPVbus_idx) QGopf]);
```

```
            % H_lg and H_gl (PG parts are implicitly zero, could leave them out)
            H(idx([VAscopf VMscopf(nPVbus_idx) QGscopf PGscopf(REFgen_idx)]), idx([VMscopf(PVbus_idx)
                ↪ PGscopf(nREFgen_idx)])) = ...
               H_local([VAopf VMopf(nPVbus_idx) QGopf PGopf(REFgen_idx)], [VMopf(PVbus_idx) PGopf(
                   ↪ nREFgen_idx)]);
            H(idx([VMscopf(PVbus_idx) PGscopf(nREFgen_idx)]), idx([VAscopf VMscopf(nPVbus_idx) QGscopf
                ↪ PGscopf(REFgen_idx)])) = ...
               H_local([VMopf(PVbus_idx) PGopf(nREFgen_idx)], [VAopf VMopf(nPVbus_idx) QGopf PGopf(
                   ↪ REFgen_idx)]);

            % H_gg hessian w.r.t global variables (and PG_ref_0)
            if i == 0
               % H_pg at non-reference gens, these are global variables
               H(idx([PGscopf(nREFgen_idx)]), idx([PGscopf(nREFgen_idx)])) = ...
                  H_local([PGopf(nREFgen_idx)], [PGopf(nREFgen_idx)]);

               % H_pgref is local variable for nominal scenario, but used in f()
               H(idx([PGscopf(REFgen_idx)]), idx([PGscopf(REFgen_idx)])) = ...
                  H_local([PGopf(REFgen_idx)], [PGopf(REFgen_idx)]);
            end

            %each scenario contributes to hessian w.r.t global VM variables at PV buses
            H(idx([VMscopf(PVbus_idx)]), idx([VMscopf(PVbus_idx)])) = ...
               H(idx([VMscopf(PVbus_idx)]), idx([VMscopf(PVbus_idx)])) + ...
               H_local([VMopf(PVbus_idx)], [VMopf(PVbus_idx)]);
         end

         %% Test for Inf, -Inf, and NaN in H.
         if find(H == Inf, 1)
            disp('MyEq:_Inf_found_in_H')
         end

         if find(H == -Inf, 1)
            disp('MyEq:_-Inf_found_in_H')
         end

         if find(isnan(H), 1)
            disp('MyEq:_NaN_found_in_H')
         end
      end
   end
```

```
   %% Define inequality constraints.
   function self = MyIneq(myauxdata)
   % z=h(x)
   self.eval = @(x) constraints(x, myauxdata);

   % z=h'(x)dx
   self.p = @(x,dx) jacobvec(x, dx, myauxdata);

   % xhat=h'(x)*dz
   self.ps = @(x,dz) jacobvec(x, dz, myauxdata);

   % xhat=(h''(x)dx)*dz
   self.pps = @(x,dx,dz) sparse(length(x),length(x));

   % Helper functions.
      function constr = constraints(x, myauxdata)
         % Append constraints for xmin <= x <= xmax.
         xmin = myauxdata.xmin;
         xmax = myauxdata.xmax;

         constr = [x - xmin;
            xmax - x(1:myauxdata.lenx_no_s)];

         %% Test for Inf, -Inf, and NaN in constr.
         if find(constr == Inf, 1)
            disp('MyIneq:_Inf_found_in_constr')
         end

         if find(constr == -Inf, 1)
```

```
            disp('MyIneq: -Inf found in constr')
        end

        if find(isnan(constr), 1)
            disp('MyIneq: NaN found in constr')
        end
    end

    function jvec = jacobvec(x, d, myauxdata)
        lenx_no_s = myauxdata.lenx_no_s;

        % Append Jacobian for x - xmin >=0.
        J = speye(length(x));

        Jmax = sparse(lenx_no_s, length(x));
        Jmax(1:lenx_no_s, 1:lenx_no_s) = -eye(lenx_no_s);

        % Append Jacobian for xmax - x (with no slacks) >= 0.
        J = [J; Jmax];

        dType = 0;
        if (size(d, 1) == size(x, 1))  % case: d == dx
            jvec = J   d;
            dType = 'dx';
        elseif (size(d, 1) == myauxdata.NINEQ) % case: d == dz
            dType = 'dz';
            jvec = J'   d;
        end

        %% Test for Inf, -Inf, and NaN in d.
        if find(d == Inf)
            fprintf('MyIneq: Inf found in %s\n', dType);
        end

        if find(d == -Inf)
            fprintf('MyIneq: -Inf found in %s\n', dType);
        end

        if find(isnan(d))
            fprintf('MyIneq: NaN found in %s\n', dType);
        end

        %% Test for Inf, -Inf, and NaN in jvec.
        if find(jvec == Inf, 1)
            disp('MyIneq: Inf found in jvec')
        end

        if find(jvec == -Inf, 1)
            disp('MyIneq: -Inf found in jvec')
        end

        if find(isnan(jvec), 1)
            disp('MyIneq: NaN found in jvec')
        end
    end
end
```

References

[1] J. Asprion. The matlab interface for ipopt. https://www.ethz.ch/content/dam/ethz/special-interest/mavt/dynamic-systems-n-control/idsc-dam/Research_Onder/Downloads/IPOPT/IPOPT_MatlabInterface_V0p1.pdf, 2013.

[2] COIN-OR. Ipopt frequently asked questions. https://projects.coin-or.org/Ipopt/wiki/FAQ, 2010.

[3] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas. Matpower: Steadystate operations, planning and analysis tools for power systems research and education. *Power Systems, IEEE Transactions on*, 26(1):12–19, February 2011. http://ieeexplore.ieee.org/document/5491276/.

[4] C.-O. Foundation. Coin-or. https://www.coin-or.org/, 2016.

[5] Gjacquenot. Solution of example lp in karmarkar's algorithm. blue lines show the constraints, red shows each iteration of the algorithm. https://commons.wikimedia.org/wiki/File:Karmarkar.svg, 2015.

[6] D. Heeger. Linear systems theory. http://www.cns.nyu.edu/~david/handouts/linear-systems/linear-systems.html, 2003.

[7] U. M. Ascher and C. Greif. *A First Course in Numerical Methods*. Society for Industrial and Applied Mathematics Philadelphia, 2011. Chapter 9. Section 2: Unconstrained optimization.

[8] U. M. Ascher and C. Greif. *A First Course in Numerical Methods*. Society for Industrial and Applied Mathematics Philadelphia, 2011. Chapter 9. Section 3: Constrained optimization.

[9] MathWorks. Matlab. https://ch.mathworks.com/products/matlab.html, 2017.

[10] H. Menke. lstlisting line wrapping. https://tex.stackexchange.com/questions/116534/lstlisting-line-wrapping, 2013.

[11] A. Optimization inc. and B. Web Design. Ampl. http://ampl.com/, 2013.

[12] D. Orban. Cuter. http://www.cuter.rl.ac.uk//, 2011.

[13] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 2 edition, 1999.

[14] A. Wächter and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. https://projects.coin-or.org/Ipopt, 2006.

[15] A. Wächter and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

[16] J. Young. Optizelle. http://www.optimojoe.com/products/optizelle/, 2016.

[17] J. Young. *Optizelle v1.2.0*. Optimojoe, October 2016. http://www.optimojoe.com/uploads/reports/Optizelle-1.2.0-a4.pdf.

[18] J. Young. Infeasible initial guess. http://forum.optimojoe.com/t/infeasible-initial-guess/46, 2017.