Solution Methods for Optimal Power Flow

Sara Vicoso Moreno (s1512504/B087606)
Teodora Stevanovska (s1610724/B101266)
Johanna Wiesflecker (s1505265/B085194)
Victor Xie (s1521631/B087234)

Supervisor: Dr. Andreas Grothey

Year 4 Project
School of Mathematics
University of Edinburgh
March 19, 2019

Abstract

Optimal Power Flow problems aim to minimize the power generating cost while satisfying network constraints. Here we develop our own Sequential Linear Programming and Sequential Quadratic Programming solvers from first principles in MATLAB. We use a MATLAB-AMPL interface to obtain the AC OPF model data from AMPL. These solvers will aim to converge to optimal solutions for different sized Optimal Power Flow problems. We find SQP to be more reliable than SLP. However SLP converges faster for cases where it obtains a solution. We conclude that the solutions provided by the SLP solver are not optimal and our SQP solver only finds optimal solutions for two cases. We suggest further alterations to the algorithms we have written to improve optimality of the solutions.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Sara Vicoso Moreno, Teodora Stevanovska, Johanna Wiesflecker & Victor Xie*)

Contents

Chapter 1

Introduction

In modern society, technology is on the rise. As of 2016, 87% of the world's population had access to electricity compared to 71% in 1990 [1]. So satisfying the rising demand for power, whilst keeping the cost of generating power as low as possible, is becoming an even more important job than ever. However, modelling power systems and minimizing their operational costs is not as straightforward as it seems. It is predicted that we see an increase in power demand by more than half in the next 20 years [2] and we need to find a solution capable of handling the needs.

The Optimal Power Flow problem was first introduced by Carpentier in 1962 [3] [4, p. 5] due to an alarming increase in power requirement. This caused concern about how much power the current systems could hold. Carpentier suggested a change from modelling power flow through real power, to an updated model that utilised real and reactive power. As a result the setup changed from a simple network optimization problem to a much more complex and non convex problem [4].

The system considered by the Optimal Power Flow problem consists of buses, lines and generators. Buses are the locations of power demand as well as power generation and generation can only occur if one or more generators are located at the bus. Lines represent the connections between the buses, through which power can be sent to satisfy demand at buses without generators. In addition, there are general constraints added to the system that cannot be exceeded as part of modelling the solution. These constraints are:

- Generation capabilities at each of the generators.
- Voltage level limits at each of the buses.
- Thermal line limits (which restrict the amount of power that can be sent through a certain line).
- Kirchhoff's Laws

[5, p. 12-13].

Carpentier formulated the Optimal Power Flow problem as an extension of the Economic Dispatch problem, which is the power system optimization problem that was used before, with the fundamental difference being the addition of Kirchhoff's Laws [6]. The Economic Dispatch problem had many limitations, such as the generating units and loads not being connected to the same bus or results leading to unacceptable flows and voltage magnitudes in the network. These issues could have been resolved by adding inequality constraints for every violation. However, doing so for each issue was not practical, which is why a more generalized approach needed to be found. Carpentier added Kirchhoff's Laws to the problem formulation to rectify this, as these laws define the proper flow of power around the network [7].

The goal of the Optimal Power Flow problem is to minimize the cost of generation, whilst maintaining the optimal system settings and satisfying all the given constraints.

The power system diagrams represent the system as a network flow from generators to loads, e.g. Figure 1.1 demonstrates the simplest possible circuit, with a single source and a single load. While the direction of the instantaneous current oscillates, the flow of real power is always from the source to the load [5, p. 7], which is the location of power demand.

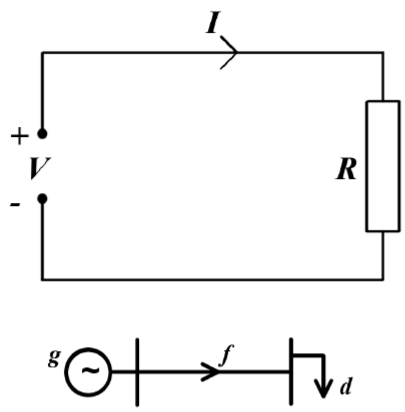

Figure 1.1: A simple AC circuit [5, p. 8]

In theory, we model the network as a graph, in which each of the nodes represents a bus (some buses with generators, some without) and every edge represents a line connecting the corresponding buses. At each generator-less bus, we are given a numerical value for the power demand at that bus. For all buses with generators, we are given the power demand at this specific bus, as well as the maximum power that can be produced there.

How much power flows along a certain line is governed by Kirchhoff's Current Laws, which state that at any node in the network, the input and output of current must equal the same [8, p. 63].

Flow limits act as another restriction in the problem. These correspond to the power lost, as heat, whilst the current travels along the line. Since the increase in temperature causes the lines to sag, due to expansion, there is a limit imposed on how much power can be transferred before the lines sag too much.[1] From this point onward, flow limits will be referred to as thermal line limits.

Kirchoff's Voltage Laws and the thermal line limits are the only non-linear constraints, as we will see from the problem's mathematical formulation. These constraints, as well as the bounds on voltage magnitude, add to the non-convexity of the problem [3] [9]. Over the years many solution approaches have been suggested such as Newton-based solution of optimality conditions, linear programming, hybrids of linear and quadratic programming, interior point methods or nonlinear programming [10] [11].

In this project, we set out to develop our own nonlinear programming solver for the Alternating Current Optimal Power Flow model from first-principles. We will aim to implement it first based on Sequential Linear Programming (SLP) and then Sequential Quadratic Programming (SQP). This report will give a brief background on all methods that we will use for developing our own Optimal Power Flow solvers as well as an overview of the problems we encountered. We will also present some of the results we obtained that are relevant for our understanding of the accuracy and efficiency of our solvers. It will start with setting up the AC model in the AMPL environment for use of the MATLAB-AMPL interface. This model will first be tested with the MATPOWER test network cases and the results will be used as a reference for our numerical solvers.

The paper is split into 5 chapters excluding the Introduction and Conclusion.

[1] The flow limits are also known as thermal line limits.

Chapter 2

Problem set up

In this chapter, we present the mathematical formulation of the Optimal Power Flow problem. Brief descriptions for each of the constraints are provided, but we do not concern ourselves with the particular physics of the problem as it is not relevant to our solution. A simple description of the network topology as well as of important concepts is included.

2.1 Network Topology

As mentioned in the Introduction, the Optimal Power Flow problem concerns a power system network that we interpret as a graph, where the buses are the nodes of the graph and the lines are represented by the edges. The generators are added to the graph as an extra connection at their location buses. A bus is allowed to have any number of generators. Power is generated at the generators and travels through the network to meet demands at the loads.

Call the set of buses B, the set of lines L and the set of generators G. Then we label each of the nodes as b_i, for $i = 1, 2, \ldots, |B|$, and each of the edges as $l_{b_i b_j}$, where b_i corresponds to its start node and b_j to its end node. The generators we label g_i, for $i = 1, 2, \ldots, |G|$. Note that even though we add direction to the lines by defining their start and end nodes, power can flow across them in both directions, i.e. from the end node to the start node as well. This flow is, however, regulated by laws of physics.

Power flow in an AC network has two components: real and reactive power. Real power is the power absorbed by the resistive component of the load [12, p. 16]. Reactive power is the amplitude of the power oscillating into and out of the load [12, p. 17]. What this means is that real power is the actual power consumption in the circuit, whereas reactive power does not produce any useful power output. Because of this, real power flow is necessarily positive, but reactive power flow is not. Reactive power flow occurs because of the phase difference between the voltage and the current waves. It is positive when the phase angle between voltage and current is positive but negative otherwise [12, p. 20].

As hinted above, voltage and current do not peak at the same time. The lag in between the periods of the two waves is called the phase difference and can be expressed as an angle, generally in degrees. This phase angle is what we refer to as the voltage phase, since the convention is to take the angle by which the voltage wave leads the current wave.[1] Then, if the voltage is leading the current, the voltage phase will be positive, if not, it will be negative [13]. In our model, however, we compute the voltage phases at the buses relative to the voltage phase at a reference bus, i.e. as a difference from the phase angle at the reference bus.

Using our graph and our mathematical model (described in detail in Section 2.3), and knowing the specific properties and demands of our network, we wish to find how much power is being generated at the generators, but also at each node, so that we are able to determine the power flowing in and out of the nodes (to meet demands). The power generation at a node is the sum of the power generated by each generator located at said node.

2.2 Nomenclature

For the purpose of this project we use the following notation:

Sets:

- set of Buses indexed by b, B
- set of Lines indexed by l, L
- set of Generators indexed by g, G

Variables:

- Real Power Generation at generator g, p_g
- Reactive Power Generation at generator g, q_g
- Real Power Generation at bus b, p_b
- Reactive Power Generation at bus b, q_b
- Voltage Level at bus b, V_b
- Voltage Phase at bus b, ϕ_b
- Real Power Flow from bus b_0 to bus b_1, $\rho_{b_0 b_1}$
- Reactive Power Flow from bus b_0 to bus b_1, $\psi_{b_0 b_1}$

Parameters:

- Cost coefficients c_1, c_2 and c_3

[1] Note that voltage and current are sinusoidal waves.

- Real Power Generation Limits at generator g, p_g^L and p_g^U
- Reactive Power Generation Limits at generator g, q_g^L and q_g^U
- Location of generator g, α_g
- Susceptance of Reactive Power at bus b, s_b^Q
- Voltage Limits at bus b, v_b^L and v_b^U
- Reference bus, β_0
- Real and Reactive Power Demand at bus b, d_b^P and d_b^Q
- Thermal Line Limit for line l, from bus b_0 to bus b_1, $t_{b_0b_1}$
- Line Conductance for line l, from bus b_0 to bus b_1, $c_{b_0b_1}$
- Line Susceptance for line l, from bus b_0 to bus b_1, $s_{b_0b_1}$
- Line Resistance for line l, from bus b_0 to bus b_1, $r_{b_0b_1}$
- Shunt Susceptance for line l, from bus b_0 to bus b_1, $s_{b_0b_1}^S$

2.3 AC OPF Formulation

Optimal Power Flow problems are optimization problems with the aim of minimizing the energy generating cost. The objective function for such a problem is represented as follows:

$$\text{minimize:} \qquad \sum_{g \in G} c_g(p_g)$$

where $c_g(p_g)$ is the cost function of a generator g [5, p. 12]. We define our cost function similarly for every generator[2] as

$$c_g(p_g) = c_1 p_g^2 + c_2 p_g + c_3$$

[15, p. 80].

Then, our objective function becomes:

$$\text{minimize:} \qquad \sum_{g \in G} c_1 p_g^2 + c_2 p_g + c_3 \tag{2.1}$$

[2] In accordance with the network data [14] that we are using in the modelling stage.

The variables representing real power generation in this equation are subject to upper and lower limits. Hence for each generator in the model we have:

$$p_g^L \leq p_g \leq p_g^U. \tag{2.2}$$

Likewise, we must have limits on the reactive power generation at each of the generators:

$$q_g^L \leq q_g \leq q_g^U. \tag{2.3}$$

Note that while p_g^L, p_g^U and q_g^U must be non-negative, q_g^L can be negative, as seen in Section 2.1. This is because reactive power flow is not a representation of net power consumption, like real power, but rather of the energy flow between components of the circuit. The sign of reactive power q_g is defined so that the direction of flow for both real and reactive power is the same [5, p. 6].

Unlike the power generation at each of the generation buses[3], the power flowing along lines cannot be regulated, as line flow is directed by physics, namely by Kirchhoff's Current Law (KCL) and by Kirchhoff's Voltage Law (KVL). These laws apply both to real power and reactive power flows, determining how much power flows along the lines and in which direction, while taking into account the power loss through heat. The direction is represented by positive and negative flow — positive flow denotes power flowing from the start bus to the end bus, whereas negative flow denotes power flowing from the end bus to the start bus. In addition to determining line flow, Kirchhoff's Laws set the voltage phases and voltage levels at the buses. These laws are translated as the following constraints:

Kirchhoff's Current Laws:

$$\sum_{g|\alpha_g=b} p_g = d_b^P + \sum_{b'} \rho_{bb'} \tag{2.4}$$

$$\sum_{g|\alpha_g=b} q_g = d_b^Q + \sum_{b'} \psi_{bb'} + s_b^Q V_b^2 \tag{2.5}$$

where b' represents any bus connected to bus b by a line.

Kirchhoff's Current Laws, as defined above, express "the conservation of flow at buses" [5, p. 12]. For reactive power there is an additional term in the expression that relates to "sources and sinks of reactive power [...] which are installed at the bus" [5, p. 12].

Kirchhoff's Voltage Laws:

$$\begin{aligned} \rho_{b_0 b_1} = &+ c_{b_0 b_1} V_{b_0}^2 \\ &- V_{b_0} V_{b_1} (c_{b_0 b_1} \cos(\phi_{b_0} - \phi_{b_1}) + s_{b_0 b_1} \sin(\phi_{b_0} - \phi_{b_1})) \end{aligned} \tag{2.6}$$

[3]Buses connected to one or more generators. As opposed to load buses, connected to loads where demand is determined.

$$\begin{aligned}\psi_{b_0b_1} = &- (s_{b_0b_1} + s^S_{b_0b_1}/2)V_{b_0}^2 \\ &- V_{b_0}V_{b_1}(c_{b_0b_1}\sin(\phi_{b_0} - \phi_{b_1}) - s_{b_0b_1}\cos(\phi_{b_0} - \phi_{b_1}))\end{aligned} \tag{2.7}$$

Kirchhoff's Voltage Laws have to be applied in both directions for each of the lines, since due to line loss, power flow is not conserved along the lines [5, p. 13].

To remove degeneracy in the voltage phases, we arbitrarily choose a reference bus where we fix the voltage phase to zero.[4] Notice that the KVL only concerns relative phase angles. Then:

$$\phi_{\beta_0} = 0. \tag{2.8}$$

The voltage level at the buses is variable. Similarly to the generation limits, voltage level constraints must be applied at each bus:

$$V_b^L \leq V_b \leq V_b^U. \tag{2.9}$$

Voltages are scaled so that the nominal voltage is 1.0, when it would otherwise be 380kV [16]. Then, $V_b \approx 1.0$ and it is kept to $1.0 \pm 10\%$ unless specified otherwise.

Finally, we consider the thermal line limits, which control how much flow can cross a line without it sagging excessively. For each line, we then have:

$$t_{b_0b_1}^2 \geq \rho_{b_0b_1}^2 + \psi_{b_0b_1}^2 \tag{2.10}$$

Because of the power loss across a line, the injections of power onto, or out of, the line at each end are not equal. So, in the same way that we are required to enforce KVL at both ends of each line, must we also enforce the thermal line limits.

This formulation is based on the one given in [5, p. 11-13]. We have extended it by taking into account shunt susceptance, by suggestion of our supervisor, as well as by defining the cost function of the generators.

2.3.1 Summary

To conclude, the AC Optimal Power Flow problem can be represented by the following model with 8 constraints:

minimize:

$$\sum_{g \in G} c_1 p_g^2 + c_2 p_g + c_3$$

[4] We do this, as opposed to using a slack bus [5], because we are not including security constraints in our AC model.

subject to:

$$
\begin{aligned}
p_g^L \leq & p_g \leq p_g^U && \forall g \in G \\
q_g^L \leq & q_g \leq q_g^U && \forall g \in G \\
V_b^L \leq & V_b \leq V_b^U && \forall b \in B \\
t_{b_0 b_1}^2 \geq & \rho_{b_0 b_1}^2 + \psi_{b_0 b_1}^2 && \forall l, -l \in L \\
\sum_{g|\alpha_g = b} p_g = & d_b^P + \sum_{b'} \rho_{bb'} && \forall b \in B \\
\sum_{g|\alpha_g = b} q_g = & d_b^Q + \sum_{b'} \psi_{bb'} + s_b^Q V_b^2 && \forall b \in B
\end{aligned}
$$

$$
\begin{aligned}
\rho_{b_0 b_1} = & + c_{b_0 b_1} V_{b_0}^2 && \\
& - V_{b_0} V_{b_1} (c_{b_0 b_1} \cos(\phi_{b_0} - \phi_{b_1}) + s_{b_0 b_1} \sin(\phi_{b_0} - \phi_{b_1})) && \forall l, -l \in L \\
\psi_{b_0 b_1} = & - (s_{b_0 b_1} + s_{b_0 b_1}^S / 2) V_{b_0}^2 && \\
& - V_{b_0} V_{b_1} (c_{b_0 b_1} \sin(\phi_{b_0} - \phi_{b_1}) - s_{b_0 b_1} \cos(\phi_{b_0} - \phi_{b_1})) && \forall l, -l \in L
\end{aligned}
$$

$$\phi_{\beta_0} = 0$$

Chapter 3

AMPL Implementation and Results

In this chapter we describe our implementation of the Optimal Power Flow problem in AMPL [17] and we present the results we obtained with AMPL for the different test networks in our data. This chapter also includes an explanation of how we obtained our modelling data and where we extracted it from.

3.1 AMPL Model

3.1.1 Implementation of the problem

We begin by setting up the model in AMPL, a modelling software designed to solve linear and nonlinear optimization problems with built-in solvers. It takes an objective function, the required constraints, a data set and returns a solution to the problem by using one of the solvers. Hence, the implementation of the problem in AMPL is very faithful to the theoretical representation in terms of readability. We have followed the description of the AC model as in Section 2.3 to build our code. Our AMPL implementation, as well as a correspondence table between the variables[1] described in Section 2.2 and the AMPL model variable names, are both included in Appendix A at the end of this paper. These results were used as a guideline for building our own SLP solver.

The objective function of our AMPL model (A.2, l.50) is the same as equation (2.1) in the previous section, representing the operating costs of generating the required power output in the given network. More precisely, the goal – or objective – of our model is to minimize these costs while ensuring all of the constraints are satisfied.

The generation constraints, equations (2.2) and (2.3) above, are quite straightforward to implement in AMPL (A.2, l.37-38) as they can be directly applied to the program. The same applies to the voltage level constraints, represented in

[1]Note that the word 'variables' here is used to encompass all variables, sets and parameters.

equation (2.9) (A.2, l.35).

In our problem formulation, we have mentioned the importance of both KVL and thermal limits being enforced at both ends of the lines. To that effect, we create new parameters `FromBus` and `ToBus`. `FromBus` is an array over the set `Lines` that indicates the start bus for each line. Similarly, `ToBus` indicates the end bus for each line. Using these new arrays we can implement the KVL constraints in both directions for each line by first defining equations (2.6) and (2.7) with b_0 in `FromBus` and b_1 in `ToBus`. We can then define them a second time with b_0 in `ToBus` and b_1 in `FromBus` (A.2, l.71-81).[2]

We model KCL, equations (2.4) and (2.5), in a similar fashion with the help of the arrays `ToBus` and `FromBus` (A.2, l.57-67). So we will have two sums, one for line injections onto the lines starting at the generation bus and one for line injections onto the lines ending at the generation bus, as opposed to the sum over all buses connected to the generation bus by a line.

Making use of the line injection variables at each end of the lines defined in the KCL constrains, we set thermal line limits at both ends of the lines (A.2, l.85-89).

Finally, we choose to set the reference bus as the first bus (A.2, l.92-93). We do this without loss of generality — as we have mentioned in Section 2.3, this is an arbitrary choice. All our data cases contain `Bus 1`, hence why we choose it.

We use the nonlinear solver MINOS 5.51, which is AMPL's default built-in solver, to obtain the results for our model.

3.1.2 Modelling Data

Our modelling data was extracted from the MATPOWER files [14]. MATPOWER provides data for networks with different sizes and configurations in the form of MATLAB [18] scripts. These scripts contain bus, branch, cost and generator data matrices for the network together with the MVA base[3] used.

We converted the matrices into an AMPL readable format by creating the MATLAB script `DataExtract.m` (A.5) which takes the MATPOWER data, looks for the relevant information, processes it and then writes it to a `.dat` file[4] in the format required by AMPL.

After extracting the matrices and MVA base, our script processes it in the following way:

[2] We do this instead of working with a set of ordered pairs, since this is uncomplicated to create from the MATPOWER data as opposed to creating the pairs.

[3] Scalar value used to convert power into per unit quantities.

[4] An example of this file, for a case with 9 buses, can be found in A.6.

The set Buses is given by column 1 in the bus matrix. The Lines and Generators sets are defined by the number of rows of the respective matrices.

Most of the parameters in our model can be found directly from the matrices and extracted in the same way that the set of Buses was extracted — each column in a matrix has values for a different parameter, defined in [14]. Exceptions are: power values and line limits that have to be divided by the MVA base, cost values which must be multiplied by the MVA base to the appropriate degree and line susceptance and conductance which are calculated from reactance and resistance as follows:

$$B = -\frac{X}{R^2 + X^2} \tag{3.1}$$

and

$$G = \frac{R}{R^2 + X^2}, \tag{3.2}$$

where B is susceptance, G is conductance, X is reactance and R is resistance — derivation found in (A.3). A table of precise data locations can be found in (A.4).

Finally, we make use of the functions AMPLcomment.m and AMPLvectorint.m (found in [19]) in addition to AMPLvectorExt.m (A.5) to write all the relevant extracted data to the .dat file.

The function AMPLvectorExt.m is an extended version of AMPLvector.m (also in [19]). Instead of simply indexing the data by the continuous set of natural numbers, it indexes it by a given set called indexdata, instead. The reason for this alteration is that, for some of the cases, the bus numbering is not continuous, i.e. it has gaps between the indices of two consecutive buses. In these cases we must index the bus data with the jumps included, for proper use in AMPL.

3.1.3 Output

To run the model for an OPF network with X number of buses we use the caseX.run file. This produces a resultsX.txt file which includes: the results for each of the variables found by AMPL's built-in solver MINOS 5.51; a feasibility message — or any other message that the solver gives; the objective value if the case is feasible; the number of iterations; and the time it takes the solver to run the model with the given data.[5] The .run and results files for each of the cases are included in Appendices A.7 and A.8, respectively.

3.2 Results

In our analysis we use data for the test networks with 9, 14, 24, 30, 39, 57, 118 and 300 buses. From now on, we are going to refer to these networks as case

[5]The latter is useful for the analysis of results in Chapter 6.

9, 14, 24... and so on for the purpose of simplicity. In this section, we present the results for case 9 and case 14, which yield feasible solutions. We do not present other results as they follow a similar analysis, but we include Table 3.3 with the objective values and feasibility for the other cases as well, as a comparison to MATPOWER results.

3.2.1 Case 9

We first consider case 9. This network consists of 9 buses and 3 generators, which are positioned at buses 1, 2 and 3. Figure 3.1 shows the exact topology of the network [14]: the buses are the 'nodes' represented by bars, which are connected to each other with 'edges' that represent the lines; the generators are represented by the blue symbols attached at b_1, b_2 and b_3; the arrows seen at some of the buses are the loads — the locations of demand of the network. There are 3 loads in this network.

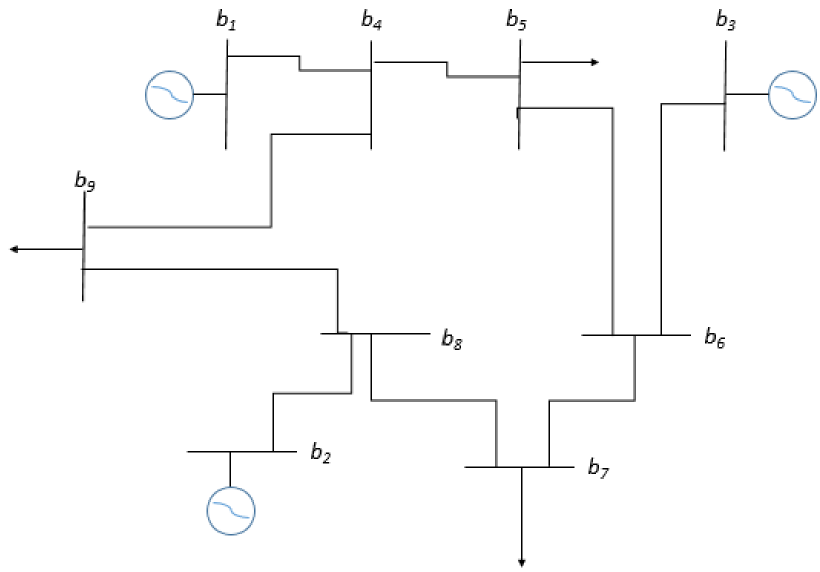

Figure 3.1: Single line diagram for IEEE 9-Bus system [14]

The solution produced by AMPL for this case is found in `results9.txt` (A.8) and is summarised in Table 3.1.[6] The power generation from each generator can be read off from the table since there is a single generator at each of the generation buses. This solution yields an objective value of $5296.69 per hour.

[6]Tables 3.1 and 3.2 show results to 3 decimal places. Real power generation is written in MW and reactive power in MVAr, obtained by multiplying the values in the results files by the MVA base.

Bus Index	V_b (p.u.)	ϕ_b (rad)	p_b (MW)	q_b (MVAr)
1	1.100	0	89.799	12.966
2	1.097	0.085	134.321	0.032
3	1.087	0.057	94.187	-22.634
4	1.094	-0.043	0	0
5	1.084	-0.069	0	0
6	1.100	0.011	0	0
7	1.089	-0.021	0	0
8	1.100	0.016	0	0
9	1.072	-0.081	0	0

Table 3.1: AMPL solution to the 9-bus case

Results for power flows along the lines are shown in the `results9.txt` file. The solution was obtained in 43 iterations.

We observe a total power generation of 318.307 MW, with a consequent reactive power total of 35.632 MVAr. The voltage magnitude for the buses varies in the interval $[1.072, 1.100]$, and the voltage phase in the interval $[-0.081, 0.085]$ radians relative to the reference bus, bus 1.

3.2.2 Case 14

In this network we have 14 buses and 5 generators located at buses 1, 2, 3, 6 and 8. The network topology is shown in Figure 3.2, interpreted similarly to Figure 3.1 above. Again, we find a single generator at each generation bus. Unlike case 9, this case does not have line limits so these are set to high values. The lines connecting bus 4 to bus 7, bus 4 to bus 9 and bus 5 to bus 6 have zero resistance, as seen in the diagram from the circles on these lines. There are 11 loads in this network, not depicted in Figure 3.2.

Table 3.2 summarizes the results produced by AMPL that can be found in `results14.txt` (A.8). This solution yields an objective value of \$8092.56 per hour, obtained in 104 iterations.

As before, we can read off each generator's power output from the table. We notice that generator 4, located at bus 6, does not produce any real power, and that generator 1, located at bus 1, does not produce any reactive power. In total, 268.527 MW of real power and 109.313 MVAr of reactive power are produced. The line flows can be found in `results14.txt`.

We see that, in this case, voltage magnitudes are kept in the interval $[0.946, 1.060]$ and voltage phases in the interval $[-0.261, -0.070]$ radians relative to the voltage phase at the reference bus, bus 1.

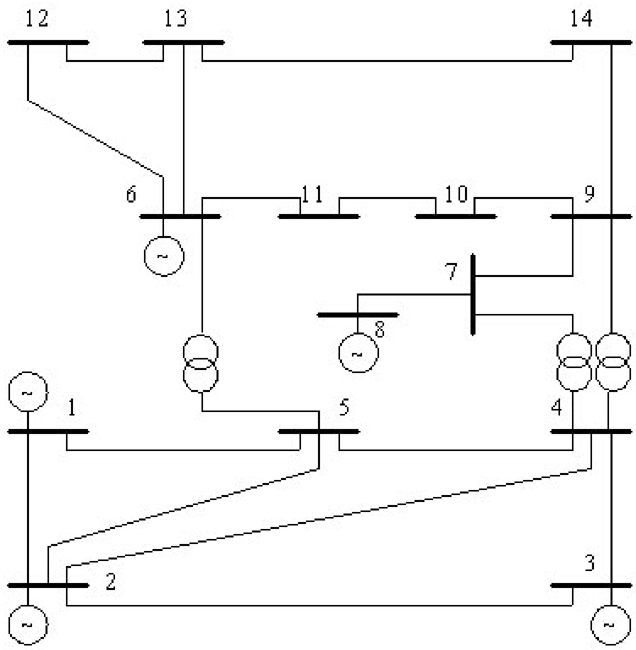

Figure 3.2: Single line diagram for IEEE 14-Bus system [20]

Bus Index	V_b (p.u.)	ϕ_b (rad)	p_b (MW)	q_b (MVAr)
1	1.060	0	193.945	0
2	1.042	-0.070	36.676	31.545
3	1.017	-0.174	28.658	29.768
4	1.007	-0.149	0	0
5	1.013	-0.129	0	0
6	0.996	-0.234	0	24.000
7	0.993	-0.196	0	0
8	1.034	-0.180	9.248	24.000
9	0.961	-0.232	0	0
10	0.959	-0.238	0	0
11	0.974	-0.239	0	0
12	0.978	-0.251	0	0
13	0.972	-0.251	0	0
14	0.946	-0.261	0	0

Table 3.2: AMPL solution to the 14-bus case

3.2.3 Table of Results

Table 3.3 gives the objective solution for the feasible cases and indicates which cases are infeasible.

Case	Objective value ($ per hour)
9 Buses	5296.68
14 Buses	8092.56
24 Buses	infeasible
30 Buses	576.89
39 Buses	41869.05
57 Buses	infeasible
118 Buses	129713.51*
300 Buses	infeasible

*118-bus case gives message "the superbasics limit (50) is too small".[7]

Table 3.3: Table of AMPL results for different OPF systems

3.2.4 Comparison with MATPOWER

MATPOWER provides the function `opf.m` [21] which gives the AC power flow solution to a given network. The results obtained with `opf.m` for the cases we are using are found in Table 3.4.

Case	Objective value ($ per hour)
9 Buses	5296.69
14 Buses	8081.53
24 Buses	63352.21
30 Buses	576.89
39 Buses	41864.18
57 Buses	41737.79
118 Buses	129660.69
300 Buses	719725.08

Table 3.4: Table of MATPOWER results for different OPF systems

Comparing Tables 3.3 and 3.4, we observe that for the AMPL feasible cases the objective values are very similar if not identical to the MATPOWER objective values. MATPOWER's `opf.m` uses the MATPOWER built-in linear solver which is an Interior Point solver [21], whereas AMPL uses MINOS 5.51 which is a nonlinear solver. This could account for the small differences in objective

[7]We could not find the correct syntax to use in AMPL to change the superbasics limit to a higher value.

values from one set of results to the other, as well as for the fact that under the MATPOWER solver all cases have feasible solutions.

On further inspection of the results in the `opf.m` output for each case, we see that the variable values are indeed very similar to the AMPL ones we have obtained. Most voltage magnitudes and voltage phases are accurate to 3 decimal places and the power generations follows similar patterns.

3.3 Future Development

As we have seen above, our implementation of the AC OPF in AMPL is very accurate — it follows the problem's mathematical formulation and mostly agrees with the MATPOWER test results. However, we would like to try to use different AMPL solvers to see if we could obtain more accurate results or make all cases feasible as in MATPOWER.

MINOS 5.51 is a nonlinear solver, in the sense that "its methods are especially effective for nonlinear objectives subject to linear and near-linear constraints" [22]. However, the KVL constraints are not near-linear, they are nonlinear. We would aim to find a solver that is effective in finding a solution for nonlinear objectives subject to linear and nonlinear constraints. AMPL has a large list of solvers available, one of which might produce better results.

Chapter 4

Sequential Linear Programming

4.1 Background

Sequential Linear Programming (SLP), also known as Successive Linear Programming,[1] consists of linearizing the objective and constraints in a region around an initial solution by using their Taylor series expansions [24, p. 1]. The resulting linear programming problem is then solved by standard methods such as the simplex algorithm [25, p. 358]. In the 1970s Sequential Linear Programming has been used widely in the energy industry as a means of solving the Optimal Power Flow problem [24, p. 1107-1108]. It is still used in practice due to its simplicity and because of how easily accessible simplex algorithm solvers are. However, methods such as Sequential Quadratic Programming and Interior Point Methods may be more reliable at solving the OPF problem [25, p. 355-356].

This chapter first introduces the theory behind Sequential Linear Programming and how it is applied to the Optimal Power Flow problem. We then introduce trust region linear programming. Finally, we aim to implement this theory into a MATLAB script, in order to create our own SLP solver. Sections 4.3 onward focus on the general implementation of this theory and the changes we need to make in order to find an optimal solution to the Optimal Power Flow problem.

For ease of explanation, the AC OPF problem can be restated as:

$$\begin{aligned} \min_{x} \quad & f(x) \\ s.t. \quad & g_{\min} \leq g(x) \leq g_{\max} \\ & x_{\min} \leq x \leq x_{\max}, \end{aligned} \tag{4.1}$$

where $f(x)$ represents the objective function (2.1) and $g(x)$ represents all the constraints of the problem. This is also the format in which the MATLAB-AMPL interface presents the problem.

[1] The words 'sequential' or 'successive' are used interchangeably in the context of linear programming approximation schemes. This work prefers to use the phrase 'sequential linear programming' [23, p. 135].

In the equation above we have represented the equality sets as inequality constraints. We can do this by duplicating the equality constraints and then replacing the two equalities with the two inequality signs:

$$h(x) = h_{bound} \tag{4.2}$$

becomes

$$\begin{aligned} h(x) &\leq h_{bound} \\ h(x) &\geq h_{bound}. \end{aligned} \tag{4.3}$$

Writing these two constraints as one gives:

$$h_{bound} \leq h(x) \leq h_{bound}, \tag{4.4}$$

which is the same format as the one used for the constraints, $g(x)$, in equation (4.1).

4.2 Applying SLP to OPF

As the name 'sequential' suggests, SLP is an iterative method. At each iteration i, f and g in equation 4.1 are approximated in a neighbourhood of the current solution estimate $x^{(i)}$ by using their first order Taylor series [24, p. 1]. So the objective and the constraints can be written in the following way:

$$f(x) = f(x^{(i)} + d) \approx f(x^{(i)}) + \nabla f(x^{(i)})^{\mathrm{T}} d \tag{4.5}$$

and

$$g(x) = g(x^{(i)} + d) \approx g(x^{(i)}) + \nabla g(x^{(i)})^{\mathrm{T}} d. \tag{4.6}$$

Here we define $d = x - x^{(i)}$ and p as the number of constraints. If we apply the Taylor approximations in (4.5) and (4.6) to the OPF-model in (4.1), our problem can now be rewritten as:

$$\begin{aligned} \min_{d} \quad & f(x^{(i)}) + \nabla f(x^{(i)})^{\mathrm{T}} d \\ s.t. \quad & g_{min} \leq g(x^{(i)}) + \nabla g(x^{(i)})^{\mathrm{T}} d \leq g_{max}, \\ & x_{min} - x^{(i)} \leq d \leq x_{max} - x^{(i)}. \end{aligned} \tag{4.7}$$

Note that here the variable is d and $x^{(i)}$ is fixed.

Now, when considering the actual OPF problem, we can see that $x^{(i)}$ and d are vectors. The derivative $\nabla f(x^{(i)})$ is therefore a vector of partial derivatives of the objective function, whilst $\nabla g(x^{(i)})$ represents the Jacobian matrix of the constraints. Hence we can write:

$$\nabla f(x^{(i)}) = \begin{bmatrix} \frac{\partial f(x^{(i)})}{\partial x_1} \\ \frac{\partial f(x^{(i)})}{\partial x_2} \\ \vdots \\ \frac{\partial f(x^{(i)})}{\partial x_n} \end{bmatrix} \tag{4.8}$$

and

$$\nabla g(x^{(i)}) = \begin{bmatrix} \frac{\partial g_1(x^{(i)})}{\partial x_1} & \frac{\partial g_1(x^{(i)})}{\partial x_2} & \cdots & \frac{\partial g_1(x^{(i)})}{\partial x_n} \\ \frac{\partial g_2(x^{(i)})}{\partial x_1} & \frac{\partial g_2(x^{(i)})}{\partial x_2} & \cdots & \frac{\partial g_2(x^{(i)})}{\partial x_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial g_p(x^{(i)})}{\partial x_1} & \frac{\partial g_p(x^{(i)})}{\partial x_2} & \cdots & \frac{\partial g_p(x^{(i)})}{\partial x_n} \end{bmatrix}, \tag{4.9}$$

where x_i is the i^{th} component of x in regard to how we differentiate the functions $f(x^{(i)})$ and $g(x^{(i)})$. The element $x^{(i)}$ represents the point at which we evaluate the Jacobian matrix and the vector of partial derivatives. This element is the solution from the previous iteration of SLP.

We now have a linear programming approximation of our original OPF problem at each iteration i. However, this may be a very poor approximation if the change in x (i.e. $|x^{(i)} - x^{(i+1)}|$) is not sufficiently small. In order to work around this problem we introduce a trust region to the step d [26]. This way we can ensure that we have a globally convergent solver, which means that the solver converges to a local solution of the problem from any starting point $x^{(0)}$ [27, p. 1].

4.2.1 Constraint Linear Programming

When applying the trust region, ρ, to step d, we then have two bounds for d:

$$\begin{aligned} x_{min} - x^{(i)} \leq &d \leq x_{max} - x^{(i)} \\ -\rho \leq &d \leq \rho. \end{aligned} \tag{4.10}$$

Rather than leaving these two bounds for d we can instead express them as one bound by taking the minimum of the magnitude of each of the bounds [28, p. 29-31][26]. This then gives the following new bound for d:

$$-\min(|\ x_{min} - x^{(i)}\ |, \rho) \leq d \leq \min(|\ x_{max} - x^{(i)}\ |, \rho). \tag{4.11}$$

Notice that we set the lower bound to the negative of the minimum. This is because we know that $x_{min} - x$ is a negative bound, so if the magnitude of the trust region is smaller than the bound, we introduce the trust region as a negative lower bound. We can therefore rewrite our problem with the new trust region constraints:

$$
\begin{aligned}
\min \quad & f(x^{(i)}) + \nabla f(x^{(i)})^{\mathrm{T}} d \\
s.t. \quad & g(x^{(i)}) + \nabla g(x^{(i)})^{\mathrm{T}} d \leq g_{max} \\
& -g(x^{(i)}) - \nabla g(x^{(i)})^{\mathrm{T}} d \leq -g_{min} \\
& -\min(\mid x_{min} - x \mid, \rho) \leq d \\
& -\min(\mid x_{max} - x \mid, \rho) \leq -d,
\end{aligned}
\tag{4.12}
$$

where the bounds of d, equation (4.11), and the constraints, equation (4.7), each have been split into two separate constraints.

The aim of the trust region is to define a region where the constraint's Taylor approximations are good, i.e. close to the actual constraints. Its size then depends on the problem formulation, for this defines the shape of the constraints, as well as on the point $x^{(i)}$ at which we are iterating. A good trust region would theoretically be a small one, because Taylor approximations are good near the point at which they are defined. However, its actual size (whatever its order of magnitude) must be dynamically dependent on how successful the previous step was, since we do not want the solution to get trapped. We want to allow the algorithm to make a bigger step if that means the objective value will be improved.

In practice, if any of the first two inequality constraints in equation (4.12) does not hold and the trust region radius is too small, the problem may be infeasible because there is no room to take a step to a new $x^{(i)}$ that will give an actual feasible solution. In this case the solution is trapped. This could be resolved with a Phase I method which we will discuss in Chapter 6.

4.3 Implementation into MATLAB

As seen in the previous section, applying SLP to the Optimal Power Flow problem leads to a linearization of the equations. Therefore, this linearization can be treated as a linear programming problem, as opposed to a nonlinear programming problem, and methods such as the Simplex method and Interior Point method can be used to solve this problem [25, p. 355-356]. Taking this into consideration, we use MATLAB to perform the iterations and analyze the results of applying this method to the OPF problem.

To implement the SLP method on our AC model, we make use of the model we created in AMPL and build[2] a MATLAB-AMPL interface to transfer the necessary data from the AMPL model. In order to successfully transfer the AMPL code, `.nl` files had to be created. Since we encounter a few issues with the readability and functionality of the AMPL code when transferred to MATLAB,

[2]We use 'build' in the sense that we have installed the necessary functions from an existing MATLAB-AMPL interface. More on this later.

a few modifications had to be made. MATLAB itself has a built-in linear programming solver, named `linprog` [29, p. 15-219], therefore, we use it to solve the linearized equations and constraints of the model. The `amplfunc` [30, p. 22] function is fundamental in building the SLP solver, because it is used to extract the necessary data from the AMPL model, namely the lower and upper bounds of the constraints and variables, the primal solution and the dual solution — at first use — , as well as $f(x^{(i)})$, $\nabla f(x^{(i)})$, $g(x^{(i)})$, $\nabla g(x^{(i)})$, $W(x^{(i)})$ at the input point $x^{(i)}$ — for each iteration. To do this we run the model on AMPL with the data we want to test, to produce a **.nl** file, which `amplfunc` then reads and uses to provide the required information[3].

4.3.1 Nomenclature

For the purpose of the MATLAB code, we have used the following notation and variables:

- `x` - primal solution output from `amplfunc`
- `bl` - lower bounds on the primal variables from `amplfunc`
- `bu` - upper bounds on the primal variables from `amplfunc`
- `v` - dual solution output from `amplfunc`
- `cl` - lower bounds on the constraints from `amplfunc`
- `cu` - upper bounds on the constraints from `amplfunc`
- `x_i` - primal initial guess
- `d` - primal solution of the LP (obtained by `linprog`)
- `x_i_temp` - trial point for next step
- `trust` - size of trust region
- `lbound` - lower bound for `d`
- `ubound` - upper bound for `d`
- `f_i` - value of the objective at `x_i`
- `predicted_obj` - value of the objective at `x_i_temp`
- `g_i` - value of the constraints at `x_i`
- `nabla_f_i` - gradient of the objective at `x_i`
- `nabla_g_i` - Jacobian matrix of constraints at `x_i`

[3] A more precise explanation of how `amplfunc` is used in the script is given in Section 4.3.2.

- `A` - coefficient matrix for constraints of `d`
- `b` - vector of bounds for constraints of `d`
- `A2` - coefficient matrix for non-infinity constraints of `d`
- `b2` - vector of bounds for non- infinity constraints of `d`
- `cv_old` - constraint violation at `x_i`
- `cv_new` - constraint violation at `x_i_temp`
- `constraint_ratio` - ratio of old and new constraint violations
- `objective_ratio` - ratio of old, new and predicted objective values
- `fval` - from `linprog`
- `exitflag` - from `linprog`
- `output` - from `linprog`
- `lambda` - lagrangian multipliers from `linprog`

4.3.2 Building the SLP Solver and Issues

The MATLAB code mentioned in this section can be found in Appendix B[4].

Before we can start with implementing the code that uses SLP to solve our Optimal Power Flow problem, we need to use the MATLAB-AMPL interface, in the form of the function `amplfunc`, to load the required information into MATLAB. There are three different ways of using `amplfunc` that we use for building our solver:

- `[x,bl,bu,v,cl,cu] = amplfunc('case9.nl')`
- `[f_i, g_i] = amplfunc(x_i,0)`
- `[nabla_f_i, nabla_g_i] = amplfunc(x_i,1)`

The first example gives all the bounds for the variables and constraints (`bl` & `bu` and `cl` & `cu`, respectively), while the second and third give the objective and constraint values evaluated at $x^{(i)}$ and their partial derivatives. Whether the function or its partial derivative is the output is defined by the second input into `amplfunc`: 0 gives the objective and constraint values and 1 gives their partial derivatives. `x` and `v` correspond to the primal and dual variables of the problem. If these variables are predefined in the AMPL model then the values will be exported to these two variables, if they are not predefined then `x` and `v`

[4]For the purpose of explaining all the steps and running through the examples, we have added the code for solving the OPF with 9 buses, which is the smallest example we have worked with as part of this project.

will be zero vectors.

After these values have been obtained from AMPL, we then start with the setup of the SLP algorithm.

Algorithm 1 Basic SLP Algorithm

procedure input: *'case9.nl'*
    Extract bl, bu, cl, cu from AMPL model
    Define starting point $x^{(i)}$
    Define size of trust region
    Define lower and upper bounds of variables $d = x - x^{(i)}$
    Set the initial d to a number greater than the exit condition.[5]
    Set iteration counter i to 1
    While $\text{norm}_{inf}(d) > \epsilon$
        Adjust bounds for d
        Perform one iteration of SLP using `SLP.m`
        Perform `progress_test.m` to see if the new step is more accurate
    End
procedure output: Objective value f, solution x
Perform dual feasibility test to see if the solution is optimal

The pseudo-code in Algorithm 1 gives an overview to the implementation for SLP. We start by loading all the necessary constraints from the AMPL model using `amplfunc` (B.1, l.4). We then need to define an initial point $x^{(i)}$ from which we will start the algorithm (B.1, l.7-8). For our solver we decide to use the so called "flat start" which involves setting all the voltage levels to 1 and all the remaining variables to 0 [9, p. 4785].

We then define an arbitrary trust region that we will start working with (B.1, l.11). In this case we choose a trust region size of 5, which we found through trial and error to work really well for this set up. Before starting the first iteration we also need to define a counter that will be used to track the number of iterations (B.1, l.14). This is necessary for the outputs and the table we will create later on.

Once all constraints are loaded and defined, we can start with the first iteration of SLP. At first we need to define the bounds of d for this iteration (B.1, l.22-23). We will redefine these bounds in every iteration before solving the LP because the trust region size will be changing in a lot of the iterations, which means that the bounds will have to be redefined to account for the new trust region. In the MATLAB code, found in Appendix B.1, we use the function `SLP.m` to solve the LP approximation of our problem near the current solution `x_i` (B.1, l.26), the code for which is given in Appendix B.2 [6]. Each iteration of SLP gives

[5] MATLAB does not offer a do-while loop, so we must define the initial value for d a priori to run the while loop.

[6] The `SLP.m` function will be explained in more detail further down this section.

a new trial point called `x_i_temp`.

Before taking the step from `x_i` to `x_i_temp`, we first need to evaluate if this new point actually gives a better solution than the previous one. This is because we use linear approximations to solve for a new point rather than solving the actual problem. So while the solution might satisfy all constraints of the linearization, there might still be constraint violations in the original model.

To evaluate whether the new point `x_i_temp` gives a better solution or not, we use `progress_test_basic.m` [7] [8](B.1, l.29). In this script the constraint violations and the objective function evaluated at the old point `x_i` and the new point `x_i_temp` are compared by using ratios of change. If the two ratios are greater than the constraints defined in the `progress_test` script we take the step and if they are not, we restrict the trust region and try to find a better point. Our script then prints the important values of the current iteration — current, predicted and actual new objective values, old and new constraint violations and progress ratios — to the console before increasing the iteration counter and starting the next iteration.

To stop the algorithm we use a while loop. The stopping condition of this while loop is based on the step size d at the current point $x^{(i)}$. We have set this value to be 1e-5 (B.1, l.19).

After the SLP algorithm has stopped we then use `linprog` one more time to evaluate the Lagrangian multipliers, so we can check if the KKT conditions[9] hold (B.1, l.41-64)[10].

The `SLP.m` function

Algorithm 2 shows the pseudo-code for the `SLP.m` function. This function[11] is used in each iteration to find a new trial point `x_i_temp`. It takes our current values `x_i`, `cu`, `cl`, `lbound` and `ubound` (B.2, l.1). The function then evaluates the constraints and objective value as well as their partial derivatives at the trial point `x_i` by using `amplfunc` (B.2, l.5-6).

In order to solve the linearization of our model, we use the built-in MATLAB function `linprog`. However, this function can only solve optimization problems

[7] For the MATLAB code of this function see Appendix B.3

[8] Throughout this chapter we discuss the scripts `progress_test_basic.m`, `progress_test_TR.m` and `progress_test_TR_CV.m`. These are all variations of the initial intended function `progress_test.m` that account for modifications made during development of the code. As such we use '`progress_test.m`' or simply '`progress_test`' interchangeably with the other names. The `progress_test.m` script will be explained in more detail further down this section.

[9] See Section 4.4 for further explanations.

[10] The steps taken before calling `linprog` are the the same steps as in `SLP.m` and will be explained in more detail in the corresponding subsection of this chapter.

[11] The MATLAB code for `SLP.m` can be found in Appendix B.2.

Algorithm 2 `SLP.m` function

procedure input: $x^{(i)}$, *cu, cl, lbound, ubound*
    Evaluate $f^{(i)}$, and $g^{(i)}$ at $x^{(i)}$
    Evaluate $\nabla f^{(i)}$ and $\nabla g^{(i)}$ at $x^{(i)}$
    Define coefficient matrix A of linearized optimization problem
    Define upper bounds b for constraints of linearized optimization problem
    Remove infinity values from constraints
    Use `linprog` to solve optimization problem for d
    Define the predicted objective value $f^{(i)}_{predicted}$
    Define temporary $x^{(i)}_{temporary}$ value
procedure output: $x^{(i)}_{temporary}$, $g^{(i)}$, $f^{(i)}$, $f^{(i)}_{predicted}$, d

of the following format:

$$\begin{aligned} \min \quad & f^T d \\ s.t. \quad & A_1 d \leq b_1 \\ & A_2 d = b_2 \\ \text{lower bound} \leq & d \leq \text{upper bound}. \end{aligned} \tag{4.13}$$

The `amplfunc` we use provides us with both upper and lower bounds for all of the constraints. From this output we cannot tell which of the constraints are equalities and which are inequalities. But, as shown at the beginning of this chapter, equalities can also be expressed as inequalities. So, all constraints will be considered as inequality constraints and therefore we will work with an optimization problem of the following form:

$$\begin{aligned} \min \quad & f^T d \\ s.t. \quad & Ad \leq b \\ \text{lower bound} \leq & d \leq \text{upper bound}. \end{aligned} \tag{4.14}$$

We allocate the values of the constraints and their lower and upper bounds to one coefficient matrix A and one upper bound vector b (B.2, l.7-8).

Another problem we encounter while using `linprog` is the fact, that this function cannot deal with the constraint vector b having `inf` entries. We therefore have to add some lines of code that deal with the removal of `inf` from the problem (B.2, l.11-13). After removing all `inf` values we can then use `linprog` to solve the LP, which will give us a new solution d (B.2, l.19).

Finally, we can define a predicted objective value `predicted_obj` and a temporary value `x_i_temp`, that will be used for progress testing in the script `progress_test.m` (B.2, l.22 and 25).

Algorithm 3 `progress_test.m` script

procedure input: *temporary* $x^{(i)}$, $f^{(i)}$, $g^{(i)}$
  Evaluate $f^{(i)}$ and $g^{(i)}$ at $x^{(i)}_{temporary}$
  Evaluate the constraint violation for the old $x^{(i)}$
  Evaluate the constraint violation for the new $x^{(i)}_{temporary}$
  Define the ratio of the actual change in constraint violation to predicted constraint violation
  Define the ratio of the actual change in objective to predicted change in objective
  if constraint ratio $\geq$ -0.1 & objective ratio ≥ 0
    Set $x^{(i)}$ equal to the temporary $x^{(i)}_{temporary}$
  else
    Reduce trust region
    Redefine lower and upper bound of the variables
  end
procedure output: either a new $x^{(i)}$ or a new trust region and bounds for the variables

The `progress_test.m` script

The `progress_test.m` script is the most important part of our SLP model. It is used to decide whether our new trial point is good enough be be taken as the next step in our iteration or not. In the next section we develop this script further in order to improve the solution that we can find with the SLP script. In this basic version of `progress_test`, shown by the pseudo-code in Algorithm 3, we use the temporary $x^{(i)}_{temporary}$ value `x_i_temp`, and the values of $f^{(i)}$ and $g^{(i)}$ evaluated at the previous point `x_i` as well as the predicted objective $f^{(i)}_{predicted}$ from the SLP function[12]. These three variables are referenced as `f_i`, `g_i` and `predicted_obj` in this script.

In order to evaluate the progress made between `x_i` and `x_i_temp`, we evaluate the ratio change in constraint violations at each of these points, as well as the ratio change in objective function (B.3, l.10 and 13).

We define

$$\text{constraint\_ratio} = \frac{\text{cv}_{x^{(i)}} - \text{cv}_{x^{(i)}_{temporary}}}{\text{cv}_{x^{(i)}}}$$
$$\text{objective\_ratio} = \frac{f^{(i)} - f^{(i)}_{temporary}}{f^{(i)} - f^{(i)}_{predicted}},$$

where cv refers to the constraint violations at the given point $x^{(i)}$ or $x^{(i)}_{temporary}$ and $f^{(i)}$ represents the objective value at $x^{(i)}$, $f^{(i)}_{temporary}$ represents the objective

[12] $f^{(i)}$ refers to $f(x^{(i)})$. Similar notation is used for $g(x^{(i)})$, and other variations of f and g.

value at $x^{(i)}_{temporary}$ the two points that we are comparing. $f^{(i)}_{predicted}$ represents the objective value found by using `linprog`.

There are many different ways of defining progress at this point. In this basic version of `progress_test` we define 'progress' in the objective as the new objective being greater than the old one (we have a ratio constraint of 0). The 'progress' for the constraint violation on the other hand is defined in such a way that we also allow steps that make the violation up to 10% worse than the predicted violation of 0 (by `linprog`). If both of these conditions are satisfied we take the step and set `x_i = x_i_temp` (B.3, l.18) for the next iteration. If there is no improvement according to at least one of the two conditions, we restrict the trust region (B.3, l.21). We then perform another iteration of SLP by using `SLP.test` with the old `x_i`.

4.4 Optimality Conditions

Once the SLP script converges to a solution, we have to check if this solution is optimal. The conditions we use to test this arise from the "Karush-Kuhn-Tucker conditions" [25, p. 358].

4.4.1 Karush-Kuhn-Tucker Conditions

Given an optimization problem of the following form:

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & g(x) \leq 0 \end{aligned}$$

it has a corresponding Lagrangian function of:

$$\mathcal{L}(x, \lambda) = f(x) + \lambda^T g(x).$$

where $\lambda \geq 0$ is the vector of Lagrangian multipliers for constraints $g(x)$. Then if x^* is an optimal local solution of the problem, the following conditions, also known as Karush-Kuhn-Tucker conditions, hold:

$$\begin{aligned} \nabla_x \mathcal{L}(x^*, \lambda^*) &= 0 \\ g(x^*) &\leq 0 \\ \lambda^* &\geq 0 \\ \lambda^{*T} g(x^*) &= 0 \end{aligned} \tag{4.15}$$

[25, p. 321]. The last two conditions in this set mean that for component i of x either the constraint is active ($g(x_i^*) = 0$) or $\lambda_i^* = 0$ [25, p. 321]. This is the main condition we will use when analysing the optimality of the solution given by our SLP script.

4.4.2 Optimality Condition for the SLP Problem

We now express the Karush-Kuhn-Tucker condtions for our problem. As mentioned in the section above the problem that we are solving with `linprog` is of the following form:

$$\begin{aligned} \min_d \quad & f(d) \\ \text{s.t.} \quad & Ad \leq b \\ \text{lower bound} \leq & d \leq \text{upper bound}. \end{aligned}$$

We can however express this in more detail, using the expressions from the script:

$$\begin{aligned} \min_d \quad & f^{(i)} + \nabla f^{(i)^T} d \\ \text{s.t.} \quad & A2d \leq b2 \\ \text{lbound} \leq & d \leq \text{ubound}, \end{aligned} \tag{4.16}$$

where $f^{(i)} = f(x^{(i)})$.

We can now split the bounds on d into two separate bounds:

$$\begin{aligned} \min_d \quad & f^{(i)} + \nabla f^{(i)^T} d \\ \text{s.t.} \quad & A2d \leq b2 \\ & \text{lbound} \leq d \\ & -\text{ubound} \leq -d. \end{aligned} \tag{4.17}$$

We then introduce three Lagrangian multipliers $s \geq 0$, $l \geq 0$ and $k \geq 0$ and write the Lagrangian:

$$\mathcal{L}(d, s, l, k) = f^{(i)} + \nabla f^{(i)^T} d + s^T(A2d - b2) + l^T(d - \text{ubound}) + k^T(-d + \text{lbound}).$$

The corresponding derivative with regard to d is:

$$\nabla_d \mathcal{L}(d, s, l, k) = \nabla f^{(i)} + A2s + l - k. \tag{4.18}$$

The Lagrangian multipliers used in equation (4.18) can be obtained by using `linprog` with the following output array:

```
[ d,fval,exitflag,output,lambda] =  linprog(f_i +
    nabla_f_i.',A2,b2,[],[],lbound,ubound).
```

The output `lambda` is a structure with four fields `lower, upper, eqlin, ineqlin`. These fields can be called by using `'lambda.*'`[13] and they correspond to the Lagrangian multipliers in 4.18. Since we do not have any equality constraints in our problem, `lambda.eqlin` is equal to zero and the other three fields correspond to the multipliers as follows:

[13]The * represents any of the four fields.

- `lambda.lower` $= k$
- `lambda.upper` $= l$
- `lambda.ineqlin` $= s$.

We have more than one Lagrange multiplier in our problem, therefore the KKT[14] conditions need to hold for all Lagrange multipliers we have introduced to the problem. From the final KKT condition we know that for every component of an optimal solution we either need the Lagrangian multiplier to be zero or the solution component must lie at the boundary of the corresponding constraint. Now, we are actually working with the linearization of the problem, which has bounds that are reinforced by the trust region. So if the bounds of a component of the solution are given by the trust region, then this component does not lie on the boundary of the initial problem. Therefore, for this solution to be optimal the Lagrange multiplier corresponding to that constraint has to be zero, so that the KKT conditions are still satisfied.

4.4.3 Basic SLP and case9.nl

When we run the basic SLP code introduced in this section we find a solution with objective value \$5296.69. The solution of this script would be optimal if all the KKT conditions hold. However, from Table B.1 we can see that there are some entries with a nonzero Lagrangian multiplier that have a bound given by the trust region (B.6, Table B.1, Entries 31 & 62). We can see this from the table because the entries for $bu - x^{(i)}$ (the actual constraints for the variables) and `ubound` (the LP constraint for the variables) are not equal for those entries. So the KKT conditions are not satisfied for this solution and the solution is not optimal.

This could be due to the fact that the trust region is too restrictive. To improve the objective value we must move toward the optimal solution and when we linearize the constraint at our $x^{(i)}$ we must make a step along the linearized constraint in order to move in the direction of objective value improvement. By moving away from the current point we are then stepping further away from the actual constraint, as the linearization is only a good approximation near $x^{(i)}$. This could make the constraint violation worse. This is why we have introduced the trust region into our problem, as it prevents the constraint violation from becoming too large. However, a very small trust region can become restrictive in some cases, because it may trap the solution at a non-optimal point, as we have mentioned in Section 4.2.1.

The following section introduces changes to the script that can rectify both these problems and explain them in more detail.

[14]Karush-Kuhn-Tucker

4.5 Script Development

We know that the trust region in the basic script is too restrictive, which prevents the algorithm from making progress as seen in Section 4.4.3, and therefore we cannot find an optimal solution to the problem. Based on this, we can implement changes in our progress making test that allow for the trust region to increase even if that means increasing constraint violations in such a case where progress becomes impossible otherwise. This way we will be able to move our solution in the direction of improvement even if it means violating the actual constraint. We will also allow the constraint violation to move freely if it is close enough to zero that we may disregard it.

4.5.1 Increasing the Trust Region

The idea we want to implement is that if we make a good enough step in terms of the solution, we will then increase the trust region in order to allow for bigger, possibly better, steps that decrease the objective and constraint values significantly. In our script `progress_test_TR.m`, we define the condition for an increase in the trust region as having both ratios greater than 0.75 (B.4, l.19). If this is the case we accept the step and then we also double the trust region (B.4, l.20-21).

In this script we now also have a condition for rejecting a step. This happens if either of the ratios is below 0.05, so we want an increase of at least 5% in both the objective value and the constraint ratio in order to accept a step as making progress. If a step is rejected we decrease the trust region by halving it and we do not update `x_i` (B.4, l.25-26). If neither of the two conditions above occurs, we just update `x_i` without changing the trust region (B.4, l.30-31).

With this script the objective value is $5296.69 again. However, we still have nonzero Lagrangian multipliers for components of the solution that have their corresponding bound given by the trust region (B.6, Table B.2, Entries 28 & 62). So the KKT conditions are not satisfied and we therefore do not have an optimal solution. What is interesting at this point, is the fact that we are given the same objective value at the final solution produced by the SLP code. But when we look at the actual solutions produced by the two scripts we can see, that they are different (C.4, Table C.4, c.1 & 2).

4.5.2 Ignoring Change in Constraint Violation

If the constraint violation is small enough that we assume it to be zero, we may allow it to worsen, while staying close to zero, if it means that we will improve the objective value[15].

[15] This may also be a consequence of allowing the trust region size to increase. However, we tried to implement it implicitly to see if it would help the algorithm satisfy the optimality conditions, as we were not able to make the trust region size adjustment completely dynamic.

Then another change we can make to the `progress_test` script is to add a condition that for sufficiently small constraint violations, we will ignore the changes on the constraint violations when testing for progress in a new step. We do this by adding a new if-statement: if the constraint violation is greater than 1e-6, which is a value we found to be working well in this case by trial and error, the script runs the same way it ran in the previous section (B.5, l.17-33); if the constraint violation is smaller, we repeat the same conditions as before, only this time we disregard the constraint violation ratio in all of the if statements and only test the change in objective value for the new value `x_i_temp` (B.5, l.34-51).

With this script we find once again an objective value of $5296.69. But we also find that the KKT conditions don't hold for this script either. There is still one nonzero Lagrangian multiplier for the upper bound of the variables that corresponds to an upper bound that is given by the trust region (B.6, Table B.3, Entry 25). So we still get a solution that is not optimal. When looking at the solutions that are given by the three scripts we can see, that for all the scripts we get a slightly different, non-optimal solution (C.4, Table C.4, c.1, 2 & 3). Since we do not seem to be able to find an optimal solution from just using our SLP solver, we will try to implement a Sequential Quadratic Programming (SQP) solver in the next chapter.

4.6 SLP Results

The aim of Chapter 4 was to implement an SLP solver for the Optimal Power Flow problem. We only used case 9 to build and test this solver throughout the chapter. In this final section we now look at the solutions produced for the other network cases we are using from MATPOWER as well as some of the changes that had to be made to the script in order to get the SLP solver to work for the other cases.

Overall, we found that the solver converged to a solution for four cases, namely cases 9, 14, 39 and 118. For case 30 it produced 9 iterations before `linprog` failed to find a feasible solution for the linearization about the current solution `x_i`. For the remaining cases (24,57 and 300) SLP failed to produce any form of output. A table with the final objective values, number of iterations and the running time for the cases where the algorithm converged to a solution, can be found in the final analysis chapter (Chapter 6, Table 6.1).

In order to get some of the cases to run, we had to change certain conditions in the SLP script. To get cases 14, 39 and 118 to converge to a solution the initial trust region had to be set to 10. Otherwise these cases would not run or converge to a solution. In addition, we also increased the stopping condition for some of the cases (1e-4 for case 39 and 1e-3 for case 118). Even though with this new stopping condition the algorithm finished with a larger final step d the constraint violations are still small for these cases (8.05e-8 for case 39 and 1.84e-5), which suggests that the solutions are feasible. We can therefore conclude that the SLP solver finds feasible solutions, for those cases that run and converge to a solution. However, as we have shown in previous sections of this chapter, those solutions are not optimal since the KKT condtions do not hold.[16] We have therefore not succeeded at building an SLP solver that finds the optimal solution for Optimal Power Flow problems.

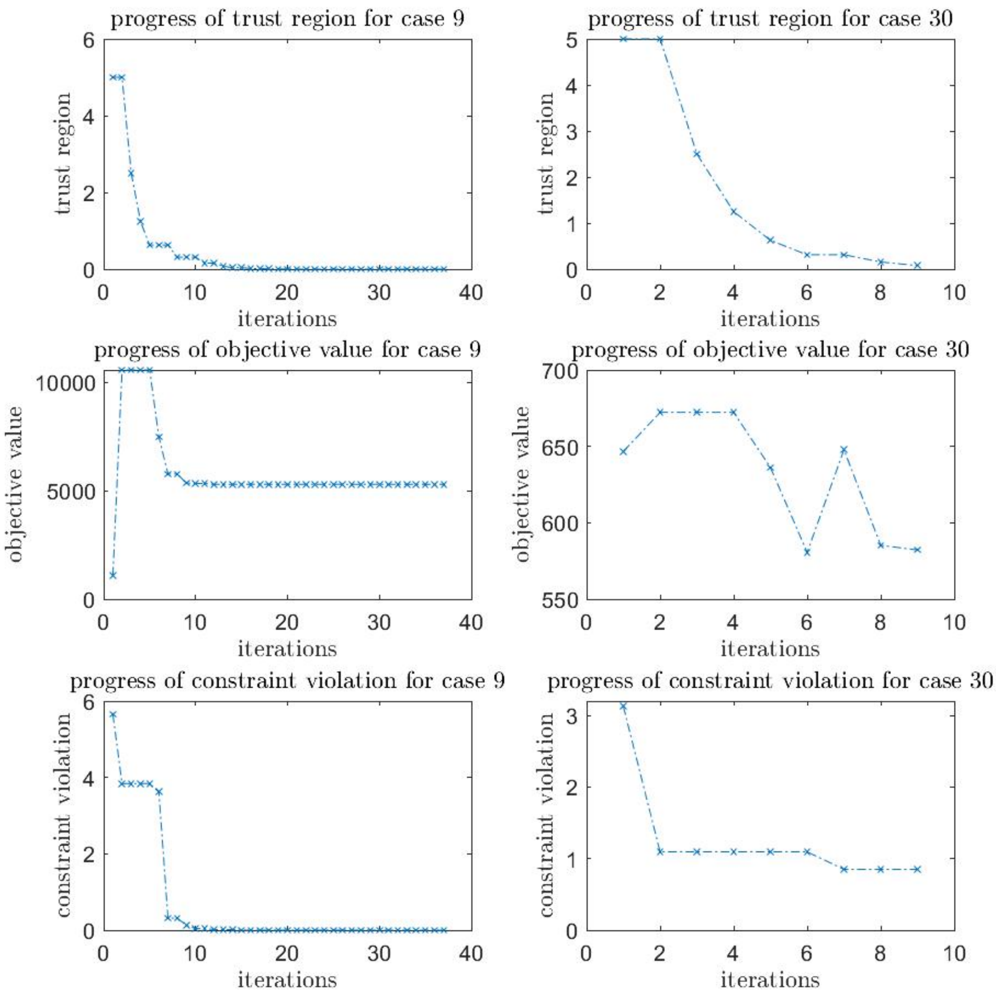

Figure 4.1: Plots that show the progress of the trust region, objective value and constraint violation for cases 9 and 30, when the SLP solver is applied.

Figure 4.1 shows the change in trust region, objective value and constraint violation throughout the iterations for cases 9 and 30. For case 9, which is a case that converges to a feasible solution, we can see from the graphs that the objective value starts to converge after around 10 iterations. Similarly, the constraint violations approach 0 after 10 iterations. What we can also see is that the trust region is very close to 0. This might be a reason why the solutions we find are not optimal, because with such a restrictive trust region all constraints are bound

[16]This can be seen from the output tables in our script.

to be enforced by the trust region which makes it unlikely that the final KKT condition holds, as none of the variables lie on the actual problem constraints. For case 30 on the other hand, the constraint violations are very clearly not 0 and the objective value doesn't converge to a final value either. We can also see that once again the trust region is very small, which suggests that here the trust region is too restrictive as well and therefore `linprog` cannot find a new feasible solution for the linearization around the current solution approximation `x_i`.

Overall, the SLP method works and could be used to produce feasible solutions in certain cases even though the solutions are not optimal. There are two ways in which we can try to improve the solver. We can either implement a 'Phase I', which would deal with the infeasible solutions at the start and would potentially provide us with a feasible starting point and a better direction of convergence [31, p. 44], or switch to using Sequential Quadratic Programming, which we know is more accurate for nonlinear programming problems [25, p. 355-356]. Due to limited time at this stage in our project we opted for implementing the SQP solver as it only required a few changes to our current SLP solver to work. Chapter 5 focuses on the implementation and results analysis of the SQP solver.

Chapter 5

Sequential Quadratic Programming

5.1 Background

Sequential Quadratic Programming was first introduced in 1963 [31, p. 4]. Since the 1970s, when it was brought to the wider attention of the optimization research audience, it has been widely researched [31, p. 4]. The main idea behind this method is that we have an approximate solution, $x^{(i)}$, which can be used to find a quadratic approximation to the problem at this point. This quadratic optimization problem can be solved using standard methods and its solution, $x^{(i+1)}$, is used as the approximate solution to the Nonlinear Porgamming (NLP) problem for the next iteration step [31, p. 7].

To derive our SQP method we first use the same notation as in equation (4.1) to formulate the NLP problem:

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & g_{\min} \leq g(x) \leq g_{\max} \\ & x_{\min} \leq x \leq x_{\max}, \end{aligned} \tag{5.1}$$

where $f(x)$ and $g(x)$ are the objective function and the constraints respectively. As in Chapter 4, we have expressed the equality constraints as inequalities, which is the format that `amplfunc` uses to provide the model setup.

5.2 SQP Formulation

Since the idea of SQP is to solve a quadratic approximation of the problem at some point $x^{(i)}$, we first need to express our NLP problem as a quadratic optimization problem, which has the following form:

$$\begin{aligned} \min_{x} \quad & x^T Q x + c \\ \text{s.t.} \quad & Ax \le b \\ x_{min} \le & x \le x_{max}. \end{aligned} \tag{5.2}$$

Similar to SLP, we use Taylor series approximations of the objective and constraints around the approximate solution $x^{(i)}$. What changes here is that we need to find the second order Taylor series approximation of the objective function. We can therefore write:

$$f(x) = f(x^{(i)} + d) \approx f(x^{(i)}) + \nabla f(x^{(i)})^T d + \frac{1}{2} d^T Hf(x^{(i)}) d \tag{5.3}$$

and

$$g(x) = g(x^{(i)} + d) \approx g(x^{(i)}) + \nabla g(x^{(i)}) d, \tag{5.4}$$

where $d = x - x^{(i)}$ and $f(x^{(i)})$ and $\nabla g(x^{(i)})$ are the vector of partial derivatives of $f(x^{(i)})$ and the Jacobian matrix of the constraints $g(x^{(i)})$, respectively. $Hf(x^{(i)})$ is the Hessian matrix of the objective function $f(x^{(i)})$, which can be written as follows:

$$Hf(x^{(i)}) = \begin{bmatrix} \frac{\partial^2 f(x^{(i)})}{\partial x_1 \partial x_1} & \frac{\partial^2 f(x^{(i)})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x^{(i)})}{\partial x_n \partial x_n} \\ \frac{\partial^2 f(x^{(i)})}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x^{(i)})}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f(x^{(i)})}{\partial x_2 \partial x_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 f(x^{(i)})}{\partial x_p \partial x_1} & \frac{\partial^2 f(x^{(i)})}{\partial x_p \partial x_2} & \cdots & \frac{\partial^2 f(x^{(i)})}{\partial x_p \partial x_n} \end{bmatrix}. \tag{5.5}$$

Subbing equations (5.3) and (5.4) back into (5.1) and letting $d = x - x^{(i)}$ gives:

$$\begin{aligned} \min_{d} \quad & f(x^{(i)}) + \nabla f(x^{(i)})^T d + \frac{1}{2} d^T Hf(x^{(i)}) d \\ \text{s.t.} \quad & g_{\min} \le g(x^{(i)}) + \nabla g(x^{(i)})^T d \le g_{\max} \\ & x_{\min} - x \le d \le x_{\max} - x. \end{aligned} \tag{5.6}$$

Note that d is now the variable and $x^{(i)}$ is fixed. Now, we can add the trust region constraints to the problem in the same way as in Section 4.2.1 and split each of the constraints into two bounds to get:

$$\begin{aligned} \min_{d} \quad & f(x^{(i)}) + \nabla f(x^{(i)})^T d + \frac{1}{2} d^T Hf(x^{(i)}) d \\ \text{s.t.} \quad & g(x^{(i)}) + \nabla g(x^{(i)})^T d \le g_{\max} \\ & -g(x^{(i)}) - \nabla g(x^{(i)})^T d \le -g_{\min} \\ & -\min(\mid x_{\min} - x \mid, \rho) \le d \\ & -\min(\mid x_{\max} - x \mid, \rho) \le -d, \end{aligned} \tag{5.7}$$

where ρ is the size of the trust region.

5.3 Optimality Conditions

The optimality conditions that are used are the same as the ones stated in Section 4.4.2. However, for this solver we will be using `quadprog` to obtain the Lagrange multipliers for the final solution.

5.4 MATLAB Implementation

The MATLAB implementation for SQP is very similar to the one for SLP. We use the final script used for SLP and change it to solve the quadratic Taylor approximations of the OPF problem instead. The scripts for SQP can be found in Appendix C. In this section we will only focus on the changes made to the final SLP solver in order to implement a SQP solver. All parts of the script that are not talked about in this section, work in the same way as in the previous chapter. We will therefore be using the same nomenclature as in Chapter 4 with the addition of the following variable:

- `lam` - dual variables of the previous iteration (obtained from `quadprog`).

The `SQP_solver.m` script

The most important change we make to the main solver script, is the switch from calling the `SLP.m` function to calling the `SQP.m` function (C.1, l.30). In order to call SQP we have to introduce a new variable `lam` to the main part of the script. This new variable is used to keep track of the dual variables at each iteration.

In addition, we also call on `quadprog` [29, p. 15-413] rather than `linprog` to test and check if the Lagrangian Duality condition holds (C.1, l.60).

The `SQP.m` function

The most significant changes we make to implement the SQP function are the ones required to implement the switch from using `linprog` to using `quadprog`.

In order to be able to use `quadprog` we need the Hessian matrix of the objective function, since the functions needs the gradient of the objective and its second derivative to solve the given quadratic optimization problem. We can obtain the Hessian by using the dual variables as the input to `amplfunc` (C.2, l.7). The dual variables `-v` are obtained from the entries of `lambda.inequlin`, which we mentioned previously for the SLP code.

Once the Hessian and all other variables are called from `amplfunc`, we have to remove the `inf` values again, since `quadprog` cannot deal with these entries either (C.2, l.14-16).

When the quadratic optimization problem is solved for `d` (C.2, l.23), we have to redefine the dual variables with the infinity values that were previously removed (C.2, l.28-32). The remaining part of the function is identical to the `SLP.m` function.

The `progress_test.m` script

The only change we make in this script is the definition of how small the constraint violations have to be in order to disregard the change in the constraint violation during the progress test. Since, we assume SQP to be more accurate we decrease the size of the constraint violation to 1e-12. (C.3, l.17).

5.4.1 SQP and case9.nl

Running the SQP for case 9 gives an objective value of $5296.69 and all dual test entries are so small that we can consider them to be zero (C.4, Table C.1). So the first condition of the KKT conditions is satisfied. Condition 2 is satisfied since we know that the constraint violations are so close to zero, that we can assume the solution is feasible and the third condition is fulfilled by the assumption that the Lagrangian multipliers are greater or equal to zero.

In our SQP problem we have 3 Lagrangian multipliers, one for the constraint inequalities and two for the upper and lower bounds of the variables (one for each). From the table printed into the console (C.2, l.23), we can see that the fourth KKT condition is satisfied for the Lagrange multiplier for the constraint inequalities. For the remaining two multipliers we check if the fourth condition holds by doing the same comparison we did in Chapter 4 for the SLP scripts. From Tables C.3 and C.2 (C.4) we can see that all non-zero entries for the Lagrangian multipliers (l and k) have corresponding bounds that are not given by the trust region. Therefore all KKT conditions are satisfied and we have an optimal soltuon for case 9. We can also see in Table C.4 (C.4) that the first 10 entries of the solution given by SQP are not equal to the ones given by SLP.

5.5 SQP Results

So far in Chapter 5 we have built the SQP solver for OPF problems based on case 9. In this section we focus on the results obtained by running the SQP solver for all network cases from MATPOWER that we decided to work with, as well as additional changes that had to be implemented to get the solver to converge to a solution for some of the cases.

The SQP solver worked as expected and gave a solution for the cases with 9 and 14 buses. In both cases the KKT condtions hold, which means that the solver found optimal solutions for those cases. For cases 39 and 118 the solver converged to feasible solutions, but the final solutions did not satisfy the final KKT condition (as can be seen from the output provided by C.1, l.76-78) and are therefore not optimal.

For the cases with 24, 30 and 57 buses the scripts managed to run, but stopped after a few iterations and did not converge to a feasible solution. Lastly, we concluded that the large network with 300 buses needs to be treated as a special case, since the computational effort required to run for this case was so great that MATLAB could not perform a single iteration on our computer.

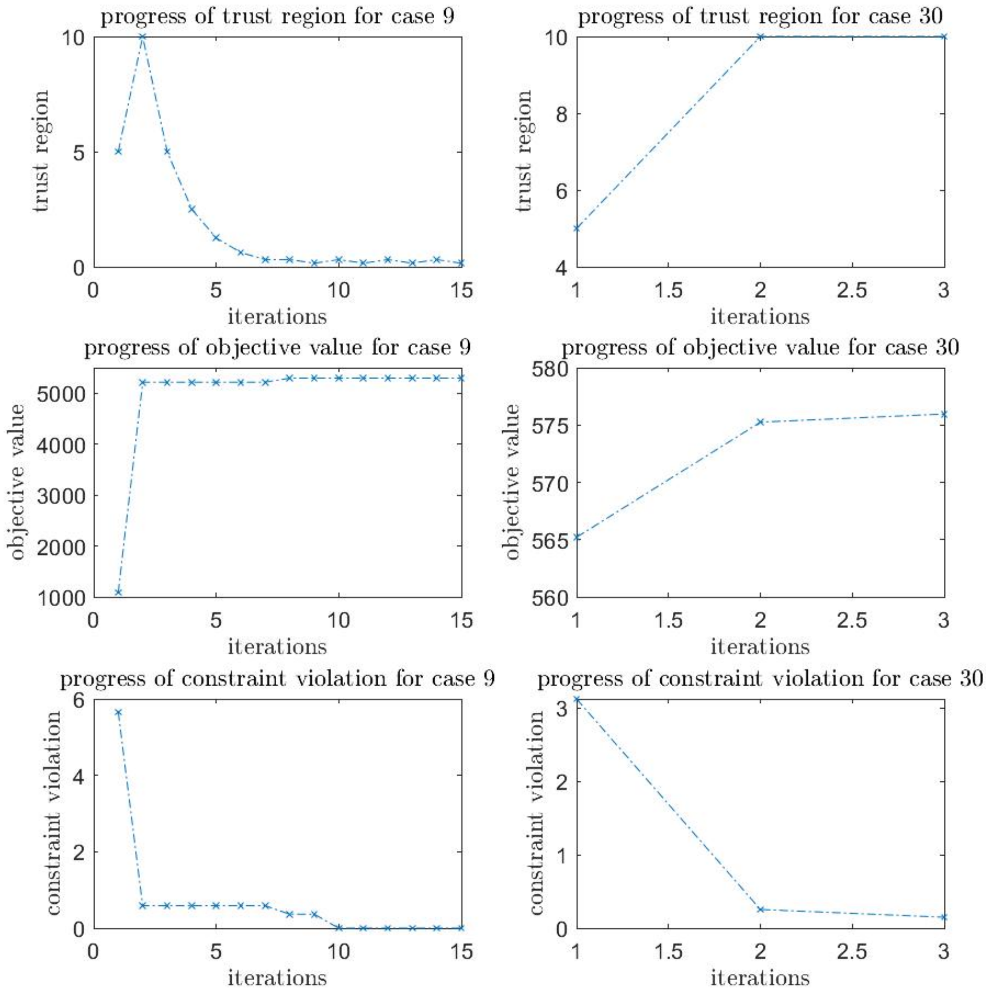

Figure 5.1: Plots that show the progress of the trust region, objective value and constraint violation for cases 9 and 30, when the SQP solver is applied.

To show how the results and the performance of the solver differ, we have plotted the trust region size, the objective value and the constraint violation against the iterations for case 9 and case 30. These plots are shown in Figure 5.1. We chose these cases in particular because the two represent a different behaviour of the solver. As mentioned, case 9 ran and gave an optimal solution, whereas case

30 ran but stopped after 3 iterations.

We can see from the plots that for case 9 the objective value converges to the expected value of \$5296.69 while the constraint violation converges to 0. What can also be seen is that the trust region for this case does not converge to 0. This is also a good indication that the solution found might be optimal as a larger trust region means that more constraints of the linearization are given by the actual constraints of the problem rather than the trust region. Therefore, fewer of the Lagrangian multipliers are required to be 0 for the KKT conditions to hold.

For case 30 on the other hand something really interesting happens. As we can see from the plots the trust region increases for the first step and then it stays at 10 for the next step. At the same time the objective approaches \$576.89, which is the objective value provided by MATPOWER, while the constraint violation decreases. This suggests that the first few solution approximations found by `quadprog` are making really good progress. However after the third iteration `quadprog` fails to find an optimal solution to the quadratic approximation of the problem. This suggests that even though the solver seems to be making good progress, it approaches an infeasible solution since it fails to produce new feasible solution approximations. One method that could help to avoid such a behaviour of the solver is to introduce a 'Phase I' to the solver. We have not focused on this method for the present project.

As described in the previous sections of the chapter, the standard initial trust region size was set to be 5 and the stopping condition size was set to 1e-6. When running all the cases the stopping condtion had to be increase for two cases (1e-4 for case 39 and 1e-5 for case 118). The initial trust region size stayed equal to 5 for all cases.

In summary, we found that for the small cases the KKT conditions hold, but this ceases to be true as the size of the cases increases, namely for case 39 and 118. The KKT will hold only where we observe $\lambda = 0$ or where the bound on the approximated constraint is the same as the bound on the actual constraint, i.e. we must have either

$$\lambda = 0 \tag{5.8}$$

or

$$g(x^{(i)}) = 0. \tag{5.9}$$

We can read these results off the tables in the output of the SQP script as explained in section 5.4.1.

Chapter 6

Results Analysis

In this chapter we compare the results obtained by solving the Optimal Power Flow problem using the three different solvers we have presented in the previous chapters. These are AMPL's built-in solver, our Sequential Linear Programming solver and our Sequential Quadratic Programming solver.

The results for different sized networks are presented in Table 6.1. Together with the results obtained from our solvers, we include a column that presents the objective values and computation times from the MATPOWER directory and a column with the values obtained from solving the AC model with AMPL. AMPL has many different solvers for problems like these, but for these results we have used the default, MINOS 5.51.

Note that the '/' symbol stands for the empty values, i.e. the values that we have not obtained a result for. The fields that have a '/' symbol at the place of the computation time are the cases for which the solvers ran, but stopped after a few iterations and did not give a solution. If there are three '/' symbols in the field, the solvers could not run for the case that the field stands for.

We use these values for reference and comparison of the efficiency. Starting with a very small network, with only 9 buses, to the one with 300 buses we have tested our solvers to end up with the following conclusions:

- The MATPOWER directory gives an optimal solution for all of the networks, whereas the other solvers do not.
- All of the solvers gave similar objective values for the cases that are treated as feasible.
- The AMPL solver has small computation times, but a greater number of iterations than the other solvers.
- The Sequential Linear Programming solver ran and gave a feasible solution for cases 9, 14, 39 and 118. It ran, but stopped for case 30 and did not run at all for cases 24, 57 and 300.

- The Sequential Quadratic Programming solver ran and gave an optimal solution for cases 9 and 14. The solver ran and gave a feasible solution for case 39 and 118, however the solutions were not optimal. The solver could run, but stopped for cases 24, 30 and 57 and did not run at all for case 300.

Case		MATPOWER[1]	AMPL	SLP	SQP
9 Buses	Objective value	5296.69	5296.68	5296.69	5296.69
	Iterations		43	37	15
	Computation time	3.24s	4.281s	1.39s	0.69s
14 Buses	Objective value	8081.53	8092.55	8092.56	8092.56
	Iterations		104	64	30
	Computation time	0.71s	3.859s	1.65s	2.66s
24 Buses	Objective value	63352.21	infeasible	/	68698.73
	Iterations		1781	/	6
	Computation time	0.40s	4.921s	/	/
30 Buses	Objective value	576.89	576.89	580.67	575.25
	Iterations		240	9	3
	Computation time	0.39s	4.015s	/	/
39 Buses	Objective value	41864.18	41869.05	41869.05	41858.90
	Iterations		241	36	55
	Computation time	0.48s	12s	1.61s	34.27s
57 Buses	Objective value	41737.79	infeasible	/	42390.27
	Iterations		1981	/	18
	Computation time	0.26s	4.704s	/	/
118 Buses	Objective value	129660.69	129713.51 [2]	132102.31	129712.00
	Iterations		1113	36	51
	Computation time	0.37s	8.265s	5.13s	1973.97s
300 Buses	Objective value	719725.08	infeasible	/	/
	Iterations		6722	/	/
	Computation time	0.83s	7.453s	/	/

Table 6.1: Table presenting the results for the objective value found by MATPOWER, AMPL's built-in solver and our SLP and SQP solvers together with the number of iterations and the computation time.

Firstly, we used MATPOWER to check the results we have obtained and make sure that our AMPL model and SLP and SQP solvers work as expected. MATPOWER uses an Interior Point solver, therefore the results can be slightly different. The objective values for each of the cases that run are very similar regardless of the solver. Differences in objective value range between 0% and 5%.

A remarkable observation is that the AMPL solver and our SLP solver both manage to find a feasible solution for the same cases. Nonetheless while AMPL's solutions were optimal, SLP's were not. The issue that both of the solvers have

[1] We do not have number of iterations from MATPOWER.
[2] 118-bus case gave the message "the superbasics limit (50) is too small".

to deal with, is moving away from a local minimum of the feasibility problem. So, whenever the trial point reaches a local minimum of the feasible region, these solvers cannot progress to a new trial point, since they consider the current point to be the best solution that can be found. An exception is case 30, for which AMPL managed to find an optimal solution, but for which SLP stopped working after 9 iterations. Despite that, the objective value that was obtained after these 9 iterations was \$580.67 per hour, which is close to the one found by AMPL.

To improve on the solutions produced by the SLP solver, we used our SQP solver. Case 57 was previously infeasible when using AMPL and did not converge when using SLP. Notably, this was improved as the SQP solution converged in only 18 iterations to a feasible solution with an objective value of \$42390.27 per hour. Note that this is not an optimal solution.

In Section 4.3.2 we explained that the size of the initial trust region is 5 as a consequence of a few trials and errors. However, this did not work for all of the cases. Namely, the trust region of this size was too tight for the case 39, so an initial feasible point could not be found. Our solvers could not start the iterations so the size of the trust region was increased to 10 in SLP. A surprising result is that the SLP solver the initial trust region size to double before starting to converge for case 118. Changing the size of the initial trust region was of great use when attempting to run some of the cases. On the other hand, the SQP solver worked without any issues with the initial size of 5.

Another approach that would potentially deal with the issue of infeasibility is implementing the 'Phase I' procedure into our solvers. The procedure would detect infeasibility and fail if the constraints are inconsistent. Moreover, it would produce a feasible initial point to start with and give a better direction for the trust region [31, p. 44].

To analyze the efficiency of our SLP and SQP solvers we have measured the computation times that MATLAB needs to find the optimal solution, using the built-in `tic-toc` function. These times were longer than the ones for MATPOWER and AMPL, since the approach and the solver used are different. As can be seen from Table 6.1, the computation times for the SLP and SQP solvers increased as the network of buses became bigger. This is a predictable result since our solvers needed to run the scripts explained in Chapters 4 and 5 for a bigger amount of data.

Having long computation times reduces the efficiency of the solvers. So, we looked into a future development that could possibly deal with this issue. One idea would be creating a hybrid of the Sequential Linear and Sequential Quadratic Programming solvers. This would be a solver that uses the computationally cheaper[3] SLP method until it stops working or reaches a non-optimal solution. It would then continue by implementing the SQP method instead, since this method is

[3]SLP is computationally cheap because it is easier to solve a linearization.

more accurate at finding an optimal solution. We would expect this solver to have smaller computation times than SQP and to be able to produce an optimal solution where SLP alone is not.

The reasoning behind this is that the SLP method would make the solver faster since it would only need to make a linear approximation up to the first term and compute the Jacobian matrix. After failing, the SQP method would take over, converging more efficiently. In summary, with the SLP solver the iteration point will approach the optimal solution quickly and SQP will help it converge.

Finally, by comparing the results we observe that the number of iterations needed to obtain a feasible solution is much lower when using our SLP and SQP solvers than the number of iterations when using AMPL. An interesting example is the network with 118 buses, where the solution converges in 36 and 51 iterations, instead of 1113 with the AMPL solver.

Chapter 7

Conclusion

The AC OPF problem can be approached through a number of different mathematical methods, as we have discussed. In this project we encountered Interior Point solvers when implementing the problem in AMPL and comparing it with MATPOWER, and we successfully implemented both Sequential Linear Programming as well as Sequential Quadratic Programming solvers, as we set out to do.

The OPF problem formulation is a nonlinear optimization problem. The SLP method, as we explained in Section 4.2.1, linearizes the problem around an approximate solution point, so that we may treat the problem as a linear programming problem at each iteration of the method. This simplifies the problem as it allows for the solution to be obtained with simplex algorithm solvers.

After going through several modifications to rectify the linearization and convergence algorithm as described through Section 4.5, the SLP solver in its final stage still could not converge to an optimal solution. The optimality condition failed to hold in the SLP solver because the solution to which it converged presented nonzero values for λ when the bound of the variable was given by the trust region. This made for a non-optimal solution because it violated the KKT conditions.

To overcome this problem, we had two options: to try to implement the 'Phase I' method in our solver, or to develop a SQP solver from the SLP. Under time constraints, we chose to progress with the second option, as it made for simple changes to our existing algorithm.

SQP is an iterative method, as is SLP, with the difference being that instead of linearizing it, we use a quadratic approximation of the objective around the approximated solution at each iteration. To implement this change we switched to using MATLAB's quadratic programming function.

The SQP solver presented an improvement to our SLP solver as it was able to find optimal solutions for two cases that SLP was not able to optimize. In addition, it was able to produce a few iterations for every case (except case 300 since we did not have enough computational capacity to run it).

We found that while the solvers are both accurate in the objective values they present[1], SQP gave some optimal solutions to the cases where SLP did not, so we may still conclude that SQP is more reliable than SLP.

Future development is needed if we are to successfully implement a solver that converges to optimal solutions for all seven network cases we have used in our project (excluding case 300). To do this we could implement 'Phase I' as mentioned before. By implementing 'Phase I', we would be able to potentially fix cases that stop running as it can detect infeasibility immediately and give a feasible starting point for our algorithm. We would first implement this change in the SLP solver since this algorithm provides cheaper computations, and we would further develop the SQP solver similarly, in the event that there were still cases that did not converge. To improve efficiency and computation time we could also use a combination of both SLP and SQP methods, as opposed to solely one of the two. Something else we might want to consider is finding a better way of dynamically adjusting the trust region size depending on the previous step and current approximate solution.

[1]Compared to AMPL or MATPOWER.

Bibliography

[1] The World Bank. Access to electricity (% of population). https://data.worldbank.org/indicator/EG.ELC.ACCS.ZS. Accessed:06-02-2019.

[2] World energy needs and nuclear power. http://www.world-nuclear.org/information-library/current-and-future-generation/world-energy-needs-and-nuclear-power.aspx. Accessed:06-03-2019.

[3] Mohammad Rasoul Narimani, Daniel K Molzahn, Dan Wu, and Mariesa L Crow. Empirical investigation of non-convexities in optimal power flow problems. In *2018 Annual American Control Conference (ACC)*, pages 3847–3854. IEEE, 2018.

[4] J. Carpentier. Optimal power flows. *International Journal of Electrical Power & Energy Systems*, 1(1):3–15, 1979.

[5] C. Dent. The AC and DC optimal power flow models. Technical report, The University of Edinburgh, available from author.

[6] Stephen Frank and Steffen Rebennack. An introduction to optimal power flow: Theory, formulation, and examples. *IIE Transactions*, 48(12):1172–1197, 2016.

[7] Mary B Cain, Richard P O'neill, and Anya Castillo. History of optimal power flow and formulations. *Federal Energy Regulatory Commission*, pages 1–36, 2012.

[8] Anshi Chen. New way to use kirchhoff's current law simplifies circuit analysis. *Electronic Design*, 56(10):63–64.

[9] Waqquas A. Bukhsh, Andreas Grothey, Ken I. M. Mckinnon, and Paul A. Trodden. Local solutions of the optimal power flow problem. *IEEE Transactions on Power Systems*, 28(4):4780–4788, 2013.

[10] Ramababu Adapa, M.E. El-Hawary, and James A. Momoh. A review of selected optimal power flow literature to 1993. i. nonlinear and quadratic programming approaches. *IEEE Transactions on Power Systems*, 14(1):96–104, 1999.

[11] Ramababu Adapa, M.E. El-Hawary, and James A. Momoh. A review of selected optimal power flow literature to 1993. ii. newton, linear programming and interior point methods. *IEEE Transactions on Power Systems*, 14(1):105–111, 1999.

[12] Hadi Saadat. *Power system analysis*. WCB/McGraw-Hill, Boston ; London, 1999.

[13] Andrew McHutchon. Ac power: A worked example. April 22, 2013.

[14] C. E. Murillo-Sanchez, R. J. Thomas, and Ray D. Zimmerman. “MATPOWER: Steady-State Operations, Planning and Analysis Tools for Power Systems Research and Education”. *IEEE Transactions on Power Systems*, 26(1):12–19, 2011.

[15] Carlos E. Murillo-Sánchez and Ray D. Zimmerman. Matpower user’s manual (version 7.0b1). `http://www.pserc.cornell.edu/matpower/MATPOWER-manual.pdf`, 2018. Accessed:04-03-2019.

[16] Jürgen Schlabbach and Karl-Heinz Rofalski. *Power system engineering: planning, design, and operation of power systems and equipment*. John Wiley & Sons, 2014.

[17] AMPL. *student license*. AMPL Optimization Inc., Sierra Place, Albuquerque, 2019.

[18] MATLAB. *version 7.10.0 (R2018b)*. The MathWorks Inc., Natick, Massachusetts, 2018.

[19] Arthur Richards. Ampl data file toolbox. `https://uk.mathworks.com/matlabcentral/fileexchange/39795-ampl-data-file-toolbox`. Accessed:06-03-2019.

[20] Power Systems and Evolutionary Algorithms. 14-bus system (ieee test case). `https://al-roomi.org/power-flow/14-bus-system`. Accessed: 23-02-2019.

[21] Carlos E. Murillo-Sánchez and Ray D. Zimmerman. Description of opf. `http://www.pserc.cornell.edu/matpower/docs/ref/matpower5.0/opf.html`, 2015. Accessed:04-03-2019.

[22] AMPL Optimization. Minos for ampl. `https://ampl.com/products/solvers/solvers-we-sell/minos/`. Accessed: 04-02-2019.

[23] LPMI Sampath, Bhagyesh V Patil, HB Gooi, Jan Marian Maciejowski, and KV Ling. A trust-region based sequential linear programming approach for ac optimal power flow problems. *Electric Power Systems Research*, 165:134–143, 2018.

[24] F. Palacios-Gomez, M. Engquist, and L. Lasdon. Nonlinear optimization by successive linear programming. *Management science.*, 28(10), 1982.

[25] Jorge Nocedal. *Numerical optimization*. Springer series in operations research. Springer, New York, second edition.. edition, 2006.

[26] Elisa Riccietti, Stefania Bellavia, and Stefano Sello. Sequential linear programming and particle swarm optimization for the optimization of energy districts. *Engineering Optimization*, 51(1):84–100, 2019.

[27] Diana Mărginean Petrovai. The global convergence of the algorithms described by multifunctions. *Procedia Engineering*, 181:924–927, 2017.

[28] Richard H. Byrd, Nicholas I.M. Gould, Jorge Nocedal, and Richard A. Waltz. An algorithm for nonlinear optimization using linear programming and equality constrained subproblems. *Mathematical Programming*, 100(1):27–48, 2003.

[29] *MATLAB: optimization toolbox: user's guide (R2018b)*. The MathWorks Inc., 2018.

[30] David M Gay. Hooking your solver to ampl. Technical report, Citeseer, 1997.

[31] Paul T Boggs and Jon W Tolle. Sequential quadratic programming. *Acta numerica*, 4:1–51, 1995.

[32] Jonathan Law and Richard Rennie. *A Dictionary of Physics*. Oxford University Press, 7 edition, 2015.

Appendix A

AMPL Model

In this appendix we include the AMPL code we used as well as tables presenting our choice of variables. We also include the MATLAB script used to extract the data from the MATPOWER data files to an AMPL readable format.

A.1 Variable Names

The following table contains the correspondence between the variable, parameter and set names used in the problem set up and the names in our AMPL model:

Variable description	**Variable notation**	**Variable name in AMPL**
Set of Buses	B	`Buses`
Set of Generators	G	`Generators`
Set of Lines	L	`Lines`
Real Power Generation at Generators	p_g	`RealPowerGeneration`
Reactive Power Generation at Generators	q_g	`ReactivePowerGeneration`
Real Power Generation at Buses	p_b	`RealPowerGenerationAtBuses`
Reactive Power Generation at Buses	q_b	`ReactivePowerGenerationAtBuses`
Voltage Level at Buses	V_b	`VoltageLevel`
Voltage Phase at Buses	ϕ_b	`VoltagePhase`
Real Power flow between Buses	ρ_{b_0,b_1}	`RealPowerInjectionFromBus` and `RealPowerInjectionToBus`
Reactive Power flow between Buses	ψ_{b_0,b_1}	`ReactivePowerInjectionFromBus` and `ReactivePowerInjectionToBus`
Cost coefficients	c_1, c_2, c_3	`Cost1`, `Cost2` and `Cost3`
Real Power Generation Limits	p_g^L, p_g^U	`RealPowerGenLimitLower` and `RealPowerGenLimitUpper`

Reactive Power Generation Limits	q_g^L, q_g^U	`ReactivePowerGenLimitLower` and `ReactivePowerGenLimitUpper`
Generator location	α_g	`GenLocation`
Susceptance of Reactive Power	sq_b	`SusceptanceOfReactivePower`
Voltage Limits	v_b^L, v_b^U	`VoltageLimitLower` and `VoltageLimitUpper`
Real Power Demands	d_b^p	`RealPowerBusDemand`
Reactive Power Demands	d_b^q	`ReactivePowerBusDemand`
Thermal Line Limit	t_l	`ThermalLineLimit`
Line Conductance	c_{b_0,b_1}	`LineConductance`
Line Susceptance	s_{b_0,b_1}	`LineSusceptance`
Line Resistance	r_{b_0,b_1}	`LineResistance`
Shunt Susceptance	sh_{b_0,b_1}	`ShuntSusceptance`
Line start bus		`FromBus`
Line end bus		`ToBus`

The following table contains an explanation of the type and format of each variable, parameter and set in our AMPL model:

Variable name in AMPL	**Description of variable**
`Buses`	Set of integers, indexed by `b`
`Generators`	Set of integers, indexed by `g`
`Lines`	Set of integers, indexed by `l`
`RealPowerGeneration`	Array of variables, over `g` in `Generators`
`ReactivePowerGeneration`	Array of variables, over `g` in `Generators`
`RealPowerGenerationAtBuses`	Array of variables, over `b` in `Buses`
`ReactivePowerGenerationAtBuses`	Array of variables, over `b` in `Buses`
`VoltageLevel`	Array of variables, over `b` in `Buses`
`VoltagePhase`	Array of variables, over `b` in `Buses`
`RealPowerInjectionFromBus`	Array of variables, over `l` in `Lines`
`RealPowerInjectionToBus`	Array of variables, over `l` in `Lines`
`ReactivePowerInjectionFromBus`	Array of variables, over `l` in `Lines`
`ReactivePowerInjectionToBus`	Array of variables, over `l` in `Lines`
`Cost1`, `Cost2` and `Cost3`	Real numbers
`RealPowerGenLimitLower`	Array of real, over `g` in `Generators`
`RealPowerGenLimitUpper`	Array of real, over `g` in `Generators`
`ReactivePowerGenLimitLower`	Array of real, over `g` in `Generators`
`ReactivePowerGenLimitUpper`	Array of real, over `g` in `Generators`
`GenLocation`	Array of integers (corresponding to bus each generator is located at), over `g` in `Generators`
`SusceptanceOfReactivePower`	Array of real, over `b` in `Buses`

VoltageLimitLower	Array of real, over b in Buses
VoltageLimitUpper	Array of real, over b in Buses
RealPowerBusDemand	Array of real, over b in Buses
ReactivePowerBusDemand	Array of real, over b in Buses
ThermalLineLimit	Array of real, over l in Lines
LineConductance	Array of real, over l in Lines
LineSusceptance	Array of real, over l in Lines
LineResistance	Array of real, over l in Lines
ShuntSusceptance	Array of real, over l in Lines
FromBus	Array of integers (corresponding to the start bus for each line), over l in Lines
ToBus	Array of integers (corresponding to the end bus for each line), over l in Lines

A.2 AMPL Code

Here we include our AMPL code that models the AC formulation of the Optimal Power Flow problem:

```
set Generators;     #indexed by g
set Buses;          #indexed by b
set Lines;          #indexed by l

param Cost1{g in Generators};
param Cost2{g in Generators};
param Cost3{g in Generators};
param RealPowerGenLimitLower{g in Generators};
param RealPowerGenLimitUpper{g in Generators};
param ReactivePowerGenLimitLower{g in Generators};
param ReactivePowerGenLimitUpper{g in Generators};
param GenLocation{g in Generators};

param SusceptanceOfReactivePower{b in Buses};
param VoltageLimitUpper{b in Buses};
param VoltageLimitLower{b in Buses};
param GeneratorIndex{b in Buses};

param RealPowerBusDemand{b in Buses};
param ReactivePowerBusDemand{b in Buses};

param ThermalLineLimit{l in Lines};

param LineConductance{l in Lines};
param LineSusceptance{l in Lines};
param LineResistance{l in Lines};
param ShuntSusceptance{l in Lines};

param FromBus{l in Lines};
param ToBus{l in Lines};

# Upper an lower limits defined here in the declaration for
```

```
# the Voltage Level, for each bus, and for Real and Reactive
# Power Generation, for each generator
var VoltageLevel{b in Buses} >= VoltageLimitLower[b], <= VoltageLimitUpper[b];
var VoltagePhase{b in Buses};
var RealPowerGeneration{g in Generators} <= RealPowerGenLimitUpper[g],>= RealPowerGenLimitLower[g];
var ReactivePowerGeneration{g in Generators} <= ReactivePowerGenLimitUpper[g],>= ReactivePowerGenLimitLower[g];
var RealPowerGenerationAtBuses{b in Buses};
var ReactivePowerGenerationAtBuses{b in Buses};
var RealPowerInjectionFromBus{l in Lines};
var RealPowerInjectionToBus{l in Lines};
var ReactivePowerInjectionFromBus{l in Lines};
var ReactivePowerInjectionToBus{l in Lines};


##############################

# Objective function
minimize OperatingCost: sum{g in Generators}( Cost1[g]*RealPowerGeneration[g]^2 + Cost2[g]*RealPowerGeneration[g] + Cost3[g]);

##############################


# Kirchhoff Current Law constraints
# 'AtBuses' constraints are just sums  to make the KCL constraints more readable
subject to RealPowerGenerationAtBusesCS{b in Buses}:
  RealPowerGenerationAtBuses[b] = sum{g in Generators: GenLocation[g]==b} RealPowerGeneration[g];

subject to ReactivePowerGenerationAtBusesCS{b in Buses}:
  ReactivePowerGenerationAtBuses[b] = sum{g in Generators: GenLocation[g]==b} ReactivePowerGeneration[g];

subject to RealCurrentCS{b in Buses}:
  RealPowerGenerationAtBuses[b] = RealPowerBusDemand[b] + sum{l in Lines: FromBus[l]==b} RealPowerInjectionFromBus[l] + sum{l in Lines: ToBus[l] == b} RealPowerInjectionToBus[l];

subject to ReactiveCurrentCS{b in Buses}:
  ReactivePowerGenerationAtBuses[b] = ReactivePowerBusDemand[b]  + sum{l in Lines: FromBus[l]==b} ReactivePowerInjectionFromBus[l] + sum{l in Lines: ToBus[l] == b} ReactivePowerInjectionToBus[l] + SusceptanceOfReactivePower[b]*VoltageLevel[b]^2;

# Kirchoff Voltage Law constraints
# Defined at both ends of the line
subject to RealVoltageCS1{l in Lines}:
   RealPowerInjectionFromBus[l] = LineConductance[l]*(VoltageLevel[FromBus[l]])^2-VoltageLevel[FromBus[l]]*VoltageLevel[ToBus[l]]*(LineConductance[l]*cos(VoltagePhase[FromBus[l]]-VoltagePhase[ToBus[l]])+LineSusceptance[l]*sin(VoltagePhase[FromBus[l]]-
```

```
    VoltagePhase[ToBus[l]]));

subject to RealVoltageCS2{l in Lines}:
   RealPowerInjectionToBus[l] = LineConductance[l]*(VoltageLevel[
    ToBus[l]])^2-VoltageLevel[ToBus[l]]*VoltageLevel[FromBus[l]]*(
    LineConductance[l]*cos(VoltagePhase[ToBus[l]]-VoltagePhase[
    FromBus[l]])+LineSusceptance[l]*sin(VoltagePhase[ToBus[l]]-
    VoltagePhase[FromBus[l]]));

subject to ReactiveVoltageCS1{l in Lines}:
  ReactivePowerInjectionFromBus[l] = - (LineSusceptance[l]+
    ShuntSusceptance[l]/2)*(VoltageLevel[FromBus[l]])^2-VoltageLevel[
    FromBus[l]]*VoltageLevel[ToBus[l]]*(LineConductance[l]*sin(
    VoltagePhase[FromBus[l]]-VoltagePhase[ToBus[l]])-LineSusceptance[
    l]*cos(VoltagePhase[FromBus[l]]-VoltagePhase[ToBus[l]]));

subject to ReactiveVoltageCS2{l in Lines}:
  ReactivePowerInjectionToBus[l] = - (LineSusceptance[l]+
    ShuntSusceptance[l]/2)*(VoltageLevel[ToBus[l]])^2-VoltageLevel[
    ToBus[l]]*VoltageLevel[FromBus[l]]*(LineConductance[l]*sin(
    VoltagePhase[ToBus[l]]-VoltagePhase[FromBus[l]])-LineSusceptance[
    l]*cos(VoltagePhase[ToBus[l]]-VoltagePhase[FromBus[l]]));

# Thermal line limits constraints
# Defined at both ends of the lines
subject to ThermalLimitsCS1{l in Lines}:
  (ThermalLineLimit[l])^2 >= (RealPowerInjectionFromBus[l])^2 + (
    ReactivePowerInjectionFromBus[l])^2;

subject to ThermalLimitsCS2{l in Lines}:
  (ThermalLineLimit[l])^2 >= (RealPowerInjectionToBus[l])^2 + (
    ReactivePowerInjectionToBus[l])^2;

# Set voltage phase at a reference bus equal to 0
subject to VoltagephaseCS{b in Buses: b == 1}:
  VoltagePhase[b] == 0;
```

A.3 Derivation of Susceptance and Conductance

Susceptance, B, and conductance, G, can be calculated from impedance, Z, as follows:

Impedance is given as

$$Z = R + iX,$$

where R denotes resistance and X denotes reactance [32, Impedance].

The reciprocal of impedance is admittance, Y, [32, Admittance]. Furthermore, susceptance is given as the reciprocal of reactance, i.e. the imaginary part of admittance [32, Susceptance], and conductance as "the ratio of the resistance to the square of the impedance in an alternating-current circuit" [32, Conductance].

We have,

$$Y = \frac{1}{Z} = \frac{1}{R + iX} = (\frac{R}{R^2 + X^2}) + i(\frac{-X}{R^2 + X^2}).$$

Now, note that $Z^2 = R^2 + X^2$ [32, Impedance]. Then conductance is the real part of admittance. Hence,

$$B = -\frac{X}{R^2 + X^2},$$

$$G = \frac{R}{R^2 + X^2}.$$

A.4 Data Location in MATPOWER Files

Here we include a table of correspondences between the sets and parameters we require in our model and their location in the MATPOWER data.

Parameter	**Location in `caseX.m`**
`Buses`	Column 1 of `buses` matrix
`Generators`	Number of rows of `gen` matrix (made into an integer array)
`Lines`	Number of rows of `branch` matrix (made into an integer array)
`Cost1`, `Cost2` and `Cost3`	Columns 5, 6 and 7 of `cost` matrix (multiplied by MVA^2, MVA and 1, respectively)
`RealPowerGenLimitLower`	Column 10 of `gen` matrix (divided by MVA base)
`RealPowerGenLimitUpper`	Column 9 of `gen` matrix (divided by MVA base)
`ReactivePowerGenLimitLower`	Column 5 of `gen` matrix (divided by MVA base)
`ReactivePowerGenLimitUpper`	Column 4 of `gen` matrix (divided by MVA base)
`GenLocation`	Column 1 of `gen` matrix
`SusceptanceOfReactivePower`	Column 6 of `buses` matrix
`VoltageLimitLower`	Column 13 of `buses` matrix
`VoltageLimitUpper`	Column 12 of `buses` matrix
`RealPowerBusDemand`	Column 3 of `buses` matrix (divided by MVA base)
`ReactivePowerBusDemand`	Column 4 of `buses` matrix (divided by MVA base)
`ThermalLineLimit`	Column 6 of `branch` matrix (divided by MVA base)
`LineConductance`	Calculated from resistance and reactance, columns 3 and 4 of `branch` matrix

`LineSusceptance`	Calculated from resistance and reactance, columns 3 and 4 of `branch` matrix
`LineResistance`	Column 3 of `branch` matrix
`ShuntSusceptance`	Column 5 of `branch` matrix
`FromBus`	Column 1 of `branch` matrix
`ToBus`	Column 2 of `branch` matrix

A.5 Data Extraction Script

Here we include the MATLAB script `DataExtractIndex.m` we used to extract the network data from the MATPOWER files and transform it as necessary into an AMPL readable format, as well as the function `AMPLvectorExt.m`.

```
clear all;
clc;

data=case9;
bus= data.bus;
line = data.branch;
generator = data.gen;
cost = data.gencost;
base = data.baseMVA;

fid = fopen('case9.dat','w');
% different sets:
b = size(bus,1);
l = size(line,1);
g = size(generator,1);
fprintf(fid,'\n');
%set of Buses
fprintf(fid,'set Buses :=');
for i= 1:b-1
    fprintf(fid,'%6.0f', bus(i,1));
end
fprintf(fid,'%6.0f;\n', bus(b,1));
%set of lines
fprintf(fid,'set Lines :=');
for i= 1:l-1
    fprintf(fid,'%4.0f', i);
end
fprintf(fid,'%4.0f;\n', l);
%set of generators
fprintf(fid,'set Generators :=');
for i= 1:g-1
    fprintf(fid,'%4.0f', i);
end
fprintf(fid,'%4.0f;\n', g);


% Generator Data:
genIndex = 1:g;

fprintf(fid,'\n');
```

```
fprintf(fid,'\n');
AMPLcomment(fid,'Generator data:');
fprintf(fid,'\n');
AMPLvectorExt(fid,'Cost1',base^2*cost(:,5),genIndex);
AMPLvectorExt(fid,'Cost2',base*cost(:,6),genIndex);
AMPLvectorExt(fid,'Cost3',cost(:,7),genIndex);
AMPLvectorExt(fid,'RealPowerGenLimitLower',generator(:,10)/base,
    genIndex);
AMPLvectorExt(fid,'RealPowerGenLimitUpper',generator(:,9)/base,
    genIndex);
AMPLvectorExt(fid,'ReactivePowerGenLimitLower',generator(:,5)/base,
    genIndex);
AMPLvectorExt(fid,'ReactivePowerGenLimitUpper',generator(:,4)/base,
    genIndex);
AMPLvectorint(fid,'GenLocation',generator(:,1));


%Bus Data:
busIndex = bus(:,1);

fprintf(fid,'\n');
fprintf(fid,'\n');
AMPLcomment(fid,'Bus data:');
fprintf(fid,'\n');
AMPLvectorExt(fid,'SusceptanceOfReactivePower',bus(:,6)/base,
    busIndex);
AMPLvectorExt(fid,'VoltageLimitLower',bus(:,13),busIndex);
AMPLvectorExt(fid,'VoltageLimitUpper',bus(:,12),busIndex);
AMPLvectorExt(fid,'RealPowerBusDemand',bus(:,3)/base,busIndex);
AMPLvectorExt(fid,'ReactivePowerBusDemand',bus(:,4)/base,busIndex);


%Line Data:
lineIndex = 1:l;

fprintf(fid,'\n');
fprintf(fid,'\n');
AMPLcomment(fid,'Line data:');
fprintf(fid,'\n');
AMPLvectorExt(fid,'ThermalLineLimit',line(:,6)/base,lineIndex);
conductance = line(:,3)./(line(:,3).^2 + line(:,4).^2);
susceptance = -line(:,4)./(line(:,3).^2+line(:,4).^2);
AMPLvectorExt(fid,'LineConductance',conductance,lineIndex);
AMPLvectorExt(fid,'LineSusceptance',susceptance,lineIndex);
AMPLvectorExt(fid,'ShuntSusceptance',line(:,5),lineIndex);
AMPLvectorExt(fid,'LineResistance',line(:,3),lineIndex);
AMPLvectorint(fid,'FromBus',line(:,1));
AMPLvectorint(fid,'ToBus',line(:,2));


fclose(fid);
```

```
function count=AMPLvectorExt(fid,pname,p,indexdata)
%
%    function count=AMPLvector(fid,pname,p)
%
```

```
% Write vector of floats to AMPL data file
%
%   fid    : file handle of the data file from 'fopen'
%   pname : name to be given to the parameter in the file (string)
%   p      : the value of the parameter (vector)
%
%   count : number of bytes written
%
% Copyright A. Richards, MIT, 2002
%
%        Extension:
%            indexdata: indices for the value of the parameter
%
index = indexdata;
s=size(p);
c=0;
if min(size(p))==1
   % vector
   l=max(s);
   c = c + fprintf(fid,['param ' pname ' := ']);
   for i=[1:(l-1)]
      c = c + fprintf(fid,'%4.0f %20.15f,',index(i),p(i));
   end
   c = c + fprintf(fid,'%4.0f %20.15f;\n',index(l),p(l));
else
   error('Not vector')
end
count=c;
```

A.6 Data Files

`case9.dat` :

```

set Buses :=      1      2      3      4      5      6      7      8      9;
set Lines :=    1    2    3    4    5    6    7    8    9;
set Generators :=    1    2    3;


# Generator data:

param Cost1 :=     1 1100.000000000000000,    2   850.000000000000110,
      3  1225.000000000000000;
param Cost2 :=     1  500.000000000000000,    2   120.000000000000000,
      3  100.000000000000000;
param Cost3 :=     1  150.000000000000000,    2   600.000000000000000,
      3  335.000000000000000;
param RealPowerGenLimitLower :=     1     0.100000000000000,    2
    0.100000000000000,    3     0.100000000000000;
param RealPowerGenLimitUpper :=     1     2.500000000000000,    2
    3.000000000000000,    3     2.700000000000000;
param ReactivePowerGenLimitLower :=     1    -3.000000000000000,    2
     -3.000000000000000,    3    -3.000000000000000;
```

```
param ReactivePowerGenLimitUpper :=    1    3.000000000000000,    2
      3.000000000000000,   3    3.000000000000000;
param GenLocation :=    1             1,   2             2,   3
        3;


# Bus data:

param SusceptanceOfReactivePower :=    1    0.000000000000000,    2
      0.000000000000000,   3    0.000000000000000,   4
    0.000000000000000,   5    0.000000000000000,   6
    0.000000000000000,   7    0.000000000000000,   8
    0.000000000000000,   9    0.000000000000000;
param VoltageLimitLower :=    1    0.900000000000000,    2
    0.900000000000000,   3    0.900000000000000,   4
    0.900000000000000,   5    0.900000000000000,   6
    0.900000000000000,   7    0.900000000000000,   8
    0.900000000000000,   9    0.900000000000000;
param VoltageLimitUpper :=    1    1.100000000000000,    2
    1.100000000000000,   3    1.100000000000000,   4
    1.100000000000000,   5    1.100000000000000,   6
    1.100000000000000,   7    1.100000000000000,   8
    1.100000000000000,   9    1.100000000000000;
param RealPowerBusDemand :=    1    0.000000000000000,    2
    0.000000000000000,   3    0.000000000000000,   4
    0.000000000000000,   5    0.900000000000000,   6
    0.000000000000000,   7    1.000000000000000,   8
    0.000000000000000,   9    1.250000000000000;
param ReactivePowerBusDemand :=    1    0.000000000000000,    2
    0.000000000000000,   3    0.000000000000000,   4
    0.000000000000000,   5    0.300000000000000,   6
    0.000000000000000,   7    0.350000000000000,   8
    0.000000000000000,   9    0.500000000000000;


# Line data:

param ThermalLineLimit :=    1    2.500000000000000,    2
    2.500000000000000,   3    1.500000000000000,   4
    3.000000000000000,   5    1.500000000000000,   6
    2.500000000000000,   7    2.500000000000000,   8
    2.500000000000000,   9    2.500000000000000;
param LineConductance :=    1    0.000000000000000,    2
    1.942191248714727,   3    1.282009138424115,   4
    0.000000000000000,   5    1.155087480890097,   6
    1.617122473246136,   7    0.000000000000000,   8
    1.187604379291149,   9    1.365187713310580;
param LineSusceptance :=    1   -17.361111111111111,    2
    -10.510682051867933,   3   -5.588244962361526,   4
    -17.064846416382252,   5   -9.784270426363172,   6
    -13.697978596908442,   7  -16.000000000000000,   8
    -5.975134533308591,   9  -11.604095563139930;
param ShuntSusceptance :=    1    0.000000000000000,    2
    0.158000000000000,   3    0.358000000000000,   4
    0.000000000000000,   5    0.209000000000000,   6
    0.149000000000000,   7    0.000000000000000,   8
```

```
0.306000000000000,   9   0.176000000000000;
param LineResistance :=   1   0.000000000000000,   2   0.017000000000000,   3   0.039000000000000,   4   0.000000000000000,   5   0.011900000000000,   6   0.008500000000000,   7   0.000000000000000,   8   0.032000000000000,   9   0.010000000000000;
param FromBus :=   1      1,   2      4,   3      5,   4      3,   5      6,   6      7,   7      8,   8      8,   9      9;
param ToBus :=   1      4,   2      5,   3      6,   4      6,   5      7,   6      8,   7      2,   8      9,   9      4;
```

case14.dat :

```

set Buses :=   1   2   3   4   5   6   7   8   9   10   11   12   13   14;
set Lines :=   1   2   3   4   5   6   7   8   9   10   11   12   13   14   15   16   17   18   19   20;
set Generators :=   1   2   3   4   5;


# Generator data:

param Cost1 :=   1  430.293000000000010,   2 2500.000000000000000,   3  100.000000000000000,   4  100.000000000000000,   5  100.000000000000000;
param Cost2 :=   1 2000.000000000000000,   2 2000.000000000000000,   3 4000.000000000000000,   4 4000.000000000000000,   5  4000.000000000000000;
param Cost3 :=   1    0.000000000000000,   2    0.000000000000000,   3    0.000000000000000,   4    0.000000000000000,   5  0.000000000000000;
param RealPowerGenLimitLower :=   1   0.000000000000000,   2   0.000000000000000,   3   0.000000000000000,   4   0.000000000000000,   5   0.000000000000000;
param RealPowerGenLimitUpper :=   1   3.324000000000000,   2   1.400000000000000,   3   1.000000000000000,   4   1.000000000000000,   5   1.000000000000000;
param ReactivePowerGenLimitLower :=   1   0.000000000000000,   2   -0.400000000000000,   3   0.000000000000000,   4   -0.060000000000000,   5   -0.060000000000000;
param ReactivePowerGenLimitUpper :=   1   0.100000000000000,   2   0.500000000000000,   3   0.400000000000000,   4   0.240000000000000,   5   0.240000000000000;
param GenLocation :=   1      1,   2      2,   3      3,   4      6,   5      8;


# Bus data:

param SusceptanceOfReactivePower :=   1   0.000000000000000,   2   0.000000000000000,   3   0.000000000000000,   4   0.000000000000000,   5   0.000000000000000,   6   0.000000000000000,   7   0.000000000000000,   8
```

```
    0.000000000000000,   9    0.190000000000000,  10
    0.000000000000000,  11    0.000000000000000,  12
    0.000000000000000,  13    0.000000000000000,  14
    0.000000000000000;
param VoltageLimitLower :=    1    0.940000000000000,    2
    0.940000000000000,   3    0.940000000000000,   4
    0.940000000000000,   5    0.940000000000000,   6
    0.940000000000000,   7    0.940000000000000,   8
    0.940000000000000,   9    0.940000000000000,  10
    0.940000000000000,  11    0.940000000000000,  12
    0.940000000000000,  13    0.940000000000000,  14
    0.940000000000000;
param VoltageLimitUpper :=    1    1.060000000000000,    2
    1.060000000000000,   3    1.060000000000000,   4
    1.060000000000000,   5    1.060000000000000,   6
    1.060000000000000,   7    1.060000000000000,   8
    1.060000000000000,   9    1.060000000000000,  10
    1.060000000000000,  11    1.060000000000000,  12
    1.060000000000000,  13    1.060000000000000,  14
    1.060000000000000;
param RealPowerBusDemand :=    1    0.000000000000000,    2
    0.217000000000000,   3    0.942000000000000,   4
    0.478000000000000,   5    0.076000000000000,   6
    0.112000000000000,   7    0.000000000000000,   8
    0.000000000000000,   9    0.295000000000000,  10
    0.090000000000000,  11    0.035000000000000,  12
    0.061000000000000,  13    0.135000000000000,  14
    0.149000000000000;
param ReactivePowerBusDemand :=    1    0.000000000000000,    2
    0.127000000000000,   3    0.190000000000000,   4
    -0.039000000000000,   5    0.016000000000000,   6
    0.075000000000000,   7    0.000000000000000,   8
    0.000000000000000,   9    0.166000000000000,  10
    0.058000000000000,  11    0.018000000000000,  12
    0.016000000000000,  13    0.058000000000000,  14
    0.050000000000000;


# Line data:

param ThermalLineLimit :=    1   99.000000000000000,    2
    99.000000000000000,   3   99.000000000000000,   4
    99.000000000000000,   5   99.000000000000000,   6
    99.000000000000000,   7   99.000000000000000,   8
    99.000000000000000,   9   99.000000000000000,  10
    99.000000000000000,  11   99.000000000000000,  12
    99.000000000000000,  13   99.000000000000000,  14
    99.000000000000000,  15   99.000000000000000,  16
    99.000000000000000,  17   99.000000000000000,  18
    99.000000000000000,  19   99.000000000000000,  20
    99.000000000000000;
param LineConductance :=    1    4.999131600798035,    2
    1.025897454970189,   3    1.135019192307396,   4
    1.686033150614943,   5    1.701139667094405,   6
    1.985975709925561,   7    6.840980661495671,   8
    0.000000000000000,   9    0.000000000000000,  10
```

```
    0.000000000000000,  11    1.955028563177261,  12
    1.525967440450974,  13    3.098927403837987,  14
    0.000000000000000,  15    0.000000000000000,  16
    3.902049552447428,  17    1.424005487019931,  18
    1.880884753700400,  19    2.489024586821919,  20
    1.136994157806327;
param LineSusceptance :=    1   -15.263086523179551,    2
    -4.234983682334831,   3   -4.781863151757718,   4
    -5.115838325872082,   5   -5.193927397969713,   6
    -5.068816977593921,   7  -21.578553981691588,   8
    -4.781943381790359,   9   -1.797979071523608,  10
    -3.967939052456154,  11   -4.094074344240442,  12
    -3.175963965029400,  13   -6.102755448193115,  14
    -5.676979846721544,  15   -9.090082719752749,  16
    -10.365394127060915,  17    -3.029050456930603,  18
    -4.402943749460521,  19   -2.251974626172212,  20
    -2.314963475105352;
param ShuntSusceptance :=    1     0.052800000000000,    2
    0.049200000000000,    3    0.043800000000000,    4
    0.034000000000000,    5    0.034600000000000,    6
    0.012800000000000,    7    0.000000000000000,    8
    0.000000000000000,    9    0.000000000000000,   10
    0.000000000000000,   11    0.000000000000000,   12
    0.000000000000000,   13    0.000000000000000,   14
    0.000000000000000,   15    0.000000000000000,   16
    0.000000000000000,   17    0.000000000000000,   18
    0.000000000000000,   19    0.000000000000000,   20
    0.000000000000000;
param LineResistance :=    1     0.019380000000000,    2
    0.054030000000000,    3    0.046990000000000,    4
    0.058110000000000,    5    0.056950000000000,    6
    0.067010000000000,    7    0.013350000000000,    8
    0.000000000000000,    9    0.000000000000000,   10
    0.000000000000000,   11    0.094980000000000,   12
    0.122910000000000,   13    0.066150000000000,   14
    0.000000000000000,   15    0.000000000000000,   16
    0.031810000000000,   17    0.127110000000000,   18
    0.082050000000000,   19    0.220920000000000,   20
    0.170930000000000;
param FromBus :=    1              1,    2              1,    3
     2,    4              2,    5              2,    6              3,    7
            4,    8              4,    9              4,   10              5,
     11             6,   12              6,   13              6,   14
         7,   15              7,   16              9,   17              9,   18
              10,   19             12,   20             13;
param ToBus :=    1              2,    2              5,    3
    3,    4              4,    5              5,    6              4,    7
           5,    8              7,    9              9,   10              6,
    11             11,   12             12,   13             13,   14
        8,   15              9,   16             10,   17             14,   18
             11,   19             13,   20             14;
```

A.7 Run Files

`case9.run` :

```
model AC_Model_new.mod;
data case9.dat;
option auxfiles rc;
option presolve 0;
write bcase9;
option log_file 'results9.txt';
solve;                              #Gives objective value and iterations
display RealPowerGeneration > 'results9.txt';        #Real power generated at each generator
display ReactivePowerGeneration > 'results9.txt';     #Reactive power generated at each generator
display RealPowerGenerationAtBuses > 'results9.txt';    #Real power generated at each bus
display ReactivePowerGenerationAtBuses > 'results9.txt';  #Reactive power generated at each bus
display VoltageLevel > 'results9.txt';              #Voltage level at each bus
display VoltagePhase > 'results9.txt';              #Voltage phase at each bus, given as a difference from the reference bus
display RealPowerInjectionFromBus > 'results9.txt';     #Real power injection onto line l, from it's start bus, i.e. pairs (b1,b2)
display RealPowerInjectionToBus > 'results9.txt';     #Real power injection onto line l, from it's end bus, i.e. pairs (b2,b1)
display ReactivePowerInjectionFromBus > 'results9.txt';   #Reactive power injection onto line l, from it's start bus, i.e. pairs (b1,b2)
display ReactivePowerInjectionToBus > 'results9.txt';   #Reactive power injection onto line l, from it's end bus, i.e. pairs (b2,b1)
display _ampl_elapsed_time > 'results9.txt';          #Measures the computation time.
option log_file '';
```

`case14.run` :

```
model AC_Model_new.mod;
data case14.dat;
option auxfiles rc;
option presolve 0;
write bcase14;
option log_file 'results14.txt';
solve;                              #Gives objective value and iterations
display RealPowerGeneration > 'results14.txt';        #Real power generated at each generator
display ReactivePowerGeneration > 'results14.txt';      #Reactive power generated at each generator
display RealPowerGenerationAtBuses > 'results14.txt';   #Real power generated at each bus
```

```
display ReactivePowerGenerationAtBuses > 'results14.txt'; #Reactive power generated at each bus
display VoltageLevel > 'results14.txt';          #Voltage level at each bus
display VoltagePhase > 'results14.txt';          #Voltage phase at each bus, given as a difference from the reference bus
display RealPowerInjectionFromBus > 'results14.txt';    #Real power injection onto line l, from it's start bus, i.e. pairs (b1,b2)
display RealPowerInjectionToBus > 'results14.txt';      #Real power injection onto line l, from it's end bus, i.e. pairs (b2,b1)
display ReactivePowerInjectionFromBus > 'results14.txt';    #Reactive power injection onto line l, from it's start bus, i.e. pairs (b1,b2)
display ReactivePowerInjectionToBus > 'results14.txt';    #Reactive power injection onto line l, from it's end bus, i.e. pairs (b2,b1)
display _ampl_elapsed_time > 'results14.txt';         #Measures the computation time.
option log_file '';
```

A.8 Results Files

`results9.txt` :

```
MINOS 5.51: optimal solution found.
43 iterations, objective 5296.686204
Nonlin evals: obj = 32, grad = 31, constrs = 32, Jac = 31.
RealPowerGeneration [*] :=
1  0.897987
2  1.34321
3  0.941874
;

ReactivePowerGeneration [*] :=
1   0.129656
2   0.000318443
3  -0.226342
;

RealPowerGenerationAtBuses [*] :=
1  0.897987
2  1.34321
3  0.941874
4  0
5  0
6  0
7  0
8  0
9  0
;

ReactivePowerGenerationAtBuses [*] :=
1   0.129656
```

```
2    0.000318443
3   -0.226342
4    0
5    0
6    0
7    0
8    0
9    0
;

VoltageLevel [*] :=
1   1.1
2   1.09735
3   1.08662
4   1.09422
5   1.08445
6   1.1
7   1.08949
8   1.1
9   1.07176
;

VoltagePhase [*] :=
1    0
2    0.0854102
3    0.0567152
4   -0.0429861
5   -0.0694987
6    0.0105224
7   -0.0208789
8    0.0158063
9   -0.0805511
;

RealPowerInjectionFromBus [*] :=
1    0.897987
2    0.352212
3   -0.549593
4    0.941874
5    0.382183
6   -0.619309
7   -1.34321
8    0.72111
9   -0.542827
;

RealPowerInjectionToBus [*] :=
1   -0.897987
2   -0.350407
3    0.559691
4   -0.941874
5   -0.380691
6    0.622096
7    1.34321
8   -0.707173
9    0.545775
```

```
;

ReactivePowerInjectionFromBus [*] :=
1    0.129656
2   -0.0389007
3   -0.161176
4   -0.226342
5   -0.0510046
6   -0.163162
7    0.0933237
8   -0.101513
9   -0.310759
;

ReactivePowerInjectionToBus [*] :=
1   -0.0904698
2   -0.138824
3   -0.221908
4    0.272912
5   -0.186838
6    0.00818962
7    0.000318443
8   -0.189241
9    0.12937
;

_ampl_elapsed_time = 4.281
```

`results14.txt` :

```
MINOS 5.51: optimal solution found.
104 iterations, objective 8092.559278
Nonlin evals: obj = 62, grad = 61, constrs = 62, Jac = 61.
RealPowerGeneration [*] :=
1   1.93945
2   0.366761
3   0.286581
4   0
5   0.092482
;

ReactivePowerGeneration [*] :=
1   0
2   0.315454
3   0.297676
4   0.24
5   0.24
;

RealPowerGenerationAtBuses [*] :=
 1   1.93945
 2   0.366761
 3   0.286581
 4   0
 5   0
 6   1.11022e-16
```

```
 7   0
 8   0.092482
 9   0
10   0
11   0
12   0
13   0
14   0
;

ReactivePowerGenerationAtBuses [*] :=
 1   0
 2   0.315454
 3   0.297676
 4   0
 5   0
 6   0.24
 7   0
 8   0.24
 9   0
10   0
11   0
12   0
13   0
14   0
;

VoltageLevel [*] :=
 1   1.06
 2   1.04158
 3   1.01731
 4   1.00746
 5   1.01333
 6   0.996018
 7   0.992991
 8   1.03376
 9   0.961219
10   0.959337
11   0.973744
12   0.978439
13   0.972038
14   0.946228
;

VoltagePhase [*] :=
 1    0
 2   −0.0703814
 3   −0.173958
 4   −0.148837
 5   −0.128577
 6   −0.234347
 7   −0.195705
 8   −0.179835
 9   −0.232205
10   −0.238339
11   −0.238728
```

```
12   -0.25077
13   -0.250905
14   -0.261167
;

RealPowerInjectionFromBus [*] :=
 1    1.29633
 2    0.643123
 3    0.559018
 4    0.486101
 5    0.371943
 6   -0.10996
 7   -0.485301
 8    0.22413
 9    0.144988
10    0.422799
11    0.0607899
12    0.0777502
13    0.172259
14   -0.092482
15    0.316612
16    0.0657563
17    0.100844
18   -0.0243929
19    0.0158775
20    0.0504302
;

RealPowerInjectionToBus [*] :=
 1   -1.2673
 2   -0.622697
 3   -0.545459
 4   -0.473254
 5   -0.364506
 6    0.111437
 7    0.488403
 8   -0.22413
 9   -0.144988
10   -0.422799
11   -0.0597835
12   -0.0768775
13   -0.169655
14    0.092482
15   -0.316612
16   -0.0656071
17   -0.0994374
18    0.0247835
19   -0.0157755
20   -0.0495626
;

ReactivePowerInjectionFromBus [*] :=
 1   -0.0781002
 2    0.0781002
 3   -5.95301e-05
 4    0.0412031
```

```
5    0.0388963
6    0.0969129
7    0.0183892
8    0.0749778
9    0.0898151
10   0.0920064
11   0.0825584
12   0.0316023
13   0.096895
14  -0.229096
15   0.292566
16  -0.00314018
17   0.00729863
18  -0.0615365
19   0.013786
20   0.0474593
;

ReactivePowerInjectionToBus [*] :=
1    0.108415
2   -0.0466786
3    0.0107631
4   -0.0379196
5   -0.0527239
6   -0.106263
7   -0.00860395
8   -0.0634697
9   -0.0738756
10  -0.0460557
11  -0.080451
12  -0.029786
13  -0.0917656
14   0.24
15  -0.271832
16   0.00353653
17  -0.00430707
18   0.062451
19  -0.0136937
20  -0.0456929
;

_ampl_elapsed_time = 3.859
```

Appendix B

MATLAB SLP code

In this Appendix we include the MATLAB code for implementing the Sequential Linear Programming method to the case with 9 buses.

B.1 SLP Script

```
clear all;

%load data from AMPL Model
[x,bl,bu,v,cl,cu] = amplfunc('case9.nl');

% define inital point for SLP:
x_i = zeros(length(bl),1);
x_i(1:9)= ones(9,1);

% define inital size of trustregion
trust = 5;

% set iteration counter to 1
i = 1;

% set d to be larger than the given exit condition
d = 1;

while norm(d,inf) > 1e-5

     % define the bounds for d before solving the next LP
    lbound = -min(trust, abs(bl-x_i));
    ubound = min(trust, abs(bu-x_i));

    % find the next trial point x_i_temp
    [x_i_temp, g_i, f_i,predicted_obj,d] = SLP(x_i,cu,cl,lbound,
   ubound);

    % test progress of temporary x_i value
    progress_test_basic;

    % display the important values at the current iteration
    disp(sprintf('%4d %8.5g %8.5g %8.5g %8.5g %8.5g %8.5g %8.5g %8.5
   g\n', ...
```

```
        i, trust, f_i, predicted_obj, f_i_new, cv_old, cv_new, ...
        constraint_ratio, objective_ratio));

    % increase the iteration number by one
    i = i + 1;
end

% evaluate the objective and the constraints at the solution found by SLP
[f_i, g_i] = amplfunc(x_i,0);
[nabla_f_i, nabla_g_i] = amplfunc(x_i,1);

% evaluate and define the Coefficient matrix of the constraints and the
% upper bound vector b
A = [ nabla_g_i;-nabla_g_i];
b = [cu-g_i;-cl+g_i];

%remove infinity values
infx = (b<inf);
A2 = A(infx,:);
b2 = b(infx);

% evaluate the Lagrangian multipliers
[d,fval,exitflag,output,lambda] = linprog(nabla_f_i.',A2,b2,[],[],lbound, ubound);

% evaluate the testing condition for Lagrangian duality
dual_test = (nabla_f_i) + (lambda.ineqlin' *A2)' - lambda.lower + lambda.upper;

%print a table with the testing condtion, the lagrangian multipliers and
%the bounds of the problem.
table(dual_test, lambda.lower,lambda.upper, x_i-bl, bu-x_i, lbound, ubound,...
    'VariableNames',{'dual_test','lambdalower','lambdaupper',...
    'x_iminusbl', 'buminusx_i','lbound','ubound'})
```

B.2 SLP Function

```
function [x_i_temp, g_i, f_i, predicted_obj,d] = SLP(x_i,cu,cl, lbound, ubound)
% This function performs one iteration of SLP.

% define coefficicent matrix A and bound vector b for constraints
[f_i, g_i] = amplfunc(x_i,0);
[nabla_f_i, nabla_g_i] = amplfunc(x_i,1);
A = [ nabla_g_i;-nabla_g_i];
b = [cu-g_i;-cl+g_i];

% remove constraints with infinity bounds
infx = (b<inf);
A2 = A(infx,:);
```

```
b2 = b(infx);

% stop outputs from linprog
options = optimset('linprog');
options.Display = 'off';
% solve the LP problem using linprog
d = linprog(nabla_f_i.',A2,b2,[],[],lbound, ubound, options);

% define the objective value predicted by linprog
predicted_obj = f_i+nabla_f_i'*d;

% define temporary x_i value to test progress before taking step
x_i_temp = d + x_i;
end
```

B.3 Progress Test

```
% evaluate the objective function and the constraints at the new
    trial
% point x_i_temp
[f_i_new, g_i_new] = amplfunc(x_i_temp,0);

% evaluate constraint violations
cv_old = sum(abs(max(g_i-cu,0))) + sum(abs(min(g_i-cl,0)));
cv_new = sum(abs(max(g_i_new-cu,0))) + sum(abs(min(g_i_new-cl,0)));

% define the constraint ratio
constraint_ratio = (cv_old - cv_new)/cv_old;

% define the objective ratio
objective_ratio = (f_i - f_i_new)./(f_i-predicted_obj);

% test if the new trial point x_i_temp is better than the current
    point x_i
if constraint_ratio > -0.1 & objective_ratio > 0
    % if the new point is better, take the step and redefine x_i
    x_i = x_i_temp;
else
    % if the old point is better decrease the trust region.
    trust = trust/2;
end
```

B.4 Progress Test with Condition to Increase the Trust Region

```
% evaluate the objective function and the constraints at the new
    trial
% point x_i_temp
[f_i_new, g_i_new] = amplfunc(x_i_temp,0);

```

```
% evaluate constraint violations
cv_old = sum(abs(max(g_i-cu,0))) + sum(abs(min(g_i-cl,0)));
cv_new = sum(abs(max(g_i_new-cu,0))) + sum(abs(min(g_i_new-cl,0)));

% define the constraint ratio
constraint_ratio = (cv_old - cv_new)/cv_old;

% define the objective ratio
objective_ratio = (f_i - f_i_new)./(f_i-predicted_obj);

% start testing conditions

% if both ratios are greater than 0.75 we take the step and increase the
% trust region
if min(constraint_ratio, objective_ratio) > 0.75
    trust = trust*2;
    x_i = x_i_temp;

% if either of the ratios is lower than 0.05 we reduce the trust region and
% reject the new trial point
elseif min(constraint_ratio, objective_ratio) < 0.05
    trust = trust/2;

% if both constraints are within the range of 0.05 to 0.75 we take the step
% without cahnging the trust region.
else
    x_i = x_i_temp;
end
```

B.5 Progress Test with Condition to Ignore Changes in Constraint Violations

```
% evaluate the objective function and the constraints at the new trial
% point x_i_temp
[f_i_new, g_i_new] = amplfunc(x_i_temp,0);

% evaluate constraint violations
cv_old = sum(abs(max(g_i-cu,0))) + sum(abs(min(g_i-cl,0)));
cv_new = sum(abs(max(g_i_new-cu,0))) + sum(abs(min(g_i_new-cl,0)));

% define the constraint ratio
constraint_ratio = (cv_old - cv_new)/cv_old;

% define the objective ratio
objective_ratio = (f_i - f_i_new)./(f_i-predicted_obj);

% start testing conditions

if cv_old > 1e-6
```

```
    % if both ratios are greater than 0.75 we take the step and
    increase the
    % trust region
    if min(constraint_ratio, objective_ratio) > 0.75
        trust = trust*2;
        x_i = x_i_temp;

        % if either of the ratios is lower than 0.05 we reduce the
    trust region and
        % reject the new trial point
    elseif min(constraint_ratio, objective_ratio) < 0.05
        trust = trust/2;

        % if both constraints are within the range of 0.05 to 0.75
    we take the step
        % without cahnging the trust region.
    else
        x_i = x_i_temp;
    end
else
    % if both ratios are greater than 0.75 we take the step and
    increase the
    % trust region
    if objective_ratio > 0.75
        trust = trust*2;
        x_i = x_i_temp;

        % if either of the ratios is lower than 0.05 we reduce the
    trust region and
        % reject the new trial point
    elseif objective_ratio < 0.05
        trust = trust/2;

        % if both constraints are within the range of 0.05 to 0.75
    we take the step
        % without cahnging the trust region.
    else
        x_i = x_i_temp;
    end
end
```

B.6 Optimality Conditions

Entry	l	$bu - x_i$	ubound	Entry	l	$bu - x_i$	ubound
1	8.22e+00	0.00e+00	0.00e+00	40	0.00e+00	Inf	4.77e-06
2	0.00e+00	2.65e-03	4.77e-06	41	0.00e+00	Inf	4.77e-06
3	0.00e+00	1.34e-02	4.77e-06	42	0.00e+00	Inf	4.77e-06
4	0.00e+00	5.78e-03	4.77e-06	43	0.00e+00	Inf	4.77e-06
5	0.00e+00	1.56e-02	4.77e-06	44	0.00e+00	Inf	4.77e-06
6	7.54e+01	0.00e+00	0.00e+00	45	0.00e+00	Inf	4.77e-06
7	0.00e+00	1.05e-02	4.77e-06	46	0.00e+00	Inf	4.77e-06
8	7.76e+01	0.00e+00	0.00e+00	47	0.00e+00	Inf	4.77e-06
9	0.00e+00	2.82e-02	4.77e-06	48	0.00e+00	Inf	4.77e-06
10	0.00e+00	Inf	4.77e-06	49	0.00e+00	Inf	4.77e-06
11	0.00e+00	Inf	4.77e-06	50	0.00e+00	Inf	4.77e-06
12	0.00e+00	Inf	4.77e-06	51	0.00e+00	Inf	4.77e-06
13	0.00e+00	Inf	4.77e-06	52	0.00e+00	Inf	4.77e-06
14	0.00e+00	Inf	4.77e-06	53	0.00e+00	Inf	4.77e-06
15	0.00e+00	Inf	4.77e-06	54	0.00e+00	Inf	4.77e-06
16	0.00e+00	Inf	4.77e-06	55	0.00e+00	1.60e+00	4.77e-06
17	0.00e+00	Inf	4.77e-06	56	0.00e+00	1.66e+00	4.77e-06
18	0.00e+00	Inf	4.77e-06	57	0.00e+00	1.76e+00	4.77e-06
19	0.00e+00	Inf	4.77e-06	58	0.00e+00	2.87e+00	4.77e-06
20	0.00e+00	Inf	4.77e-06	59	0.00e+00	3.00e+00	4.77e-06
21	0.00e+00	Inf	4.77e-06	60	0.00e+00	3.23e+00	4.77e-06
22	0.00e+00	Inf	4.77e-06	61	0.00e+00	Inf	4.77e-06
23	0.00e+00	Inf	4.77e-06	62	5.41e-03	Inf	4.77e-06
24	0.00e+00	Inf	4.77e-06	63	0.00e+00	Inf	4.77e-06
25	0.00e+00	Inf	4.77e-06	64	0.00e+00	Inf	4.77e-06
26	0.00e+00	Inf	4.77e-06	65	0.00e+00	Inf	4.77e-06
27	0.00e+00	Inf	4.77e-06	66	0.00e+00	Inf	4.77e-06
28	0.00e+00	Inf	4.77e-06	67	0.00e+00	Inf	4.77e-06
29	0.00e+00	Inf	4.77e-06	68	0.00e+00	Inf	4.77e-06
30	0.00e+00	Inf	4.77e-06	69	0.00e+00	Inf	4.77e-06
31	8.43e-03	Inf	4.77e-06	70	0.00e+00	Inf	4.77e-06
32	0.00e+00	Inf	4.77e-06	71	0.00e+00	Inf	4.77e-06
33	0.00e+00	Inf	4.77e-06	72	0.00e+00	Inf	4.77e-06
34	0.00e+00	Inf	4.77e-06	73	0.00e+00	Inf	4.77e-06
35	0.00e+00	Inf	4.77e-06	74	0.00e+00	Inf	4.77e-06
36	0.00e+00	Inf	4.77e-06	75	0.00e+00	Inf	4.77e-06
37	0.00e+00	Inf	4.77e-06	76	0.00e+00	Inf	4.77e-06
38	0.00e+00	Inf	4.77e-06	77	0.00e+00	Inf	4.77e-06
39	0.00e+00	Inf	4.77e-06	78	0.00e+00	Inf	4.77e-06

Table B.1: Entries of the upper bound variables for the basic SLP script.

Entry	l	$bu - x_i$	ubound	Entry	l	$bu - x_i$	ubound
1	8.22e+00	0.00e+00	0.00e+00	40	0.00e+00	Inf	9.54e-06
2	0.00e+00	2.65e-03	9.54e-06	41	0.00e+00	Inf	9.54e-06
3	0.00e+00	1.34e-02	9.54e-06	42	0.00e+00	Inf	9.54e-06
4	0.00e+00	5.78e-03	9.54e-06	43	0.00e+00	Inf	9.54e-06
5	0.00e+00	1.56e-02	9.54e-06	44	0.00e+00	Inf	9.54e-06
6	7.54e+01	0.00e+00	0.00e+00	45	0.00e+00	Inf	9.54e-06
7	0.00e+00	1.05e-02	9.54e-06	46	0.00e+00	Inf	9.54e-06
8	7.76e+01	0.00e+00	0.00e+00	47	0.00e+00	Inf	9.54e-06
9	0.00e+00	2.82e-02	9.54e-06	48	0.00e+00	Inf	9.54e-06
10	0.00e+00	Inf	9.54e-06	49	0.00e+00	Inf	9.54e-06
11	0.00e+00	Inf	9.54e-06	50	0.00e+00	Inf	9.54e-06
12	0.00e+00	Inf	9.54e-06	51	0.00e+00	Inf	9.54e-06
13	0.00e+00	Inf	9.54e-06	52	0.00e+00	Inf	9.54e-06
14	0.00e+00	Inf	9.54e-06	53	0.00e+00	Inf	9.54e-06
15	0.00e+00	Inf	9.54e-06	54	0.00e+00	Inf	9.54e-06
16	0.00e+00	Inf	9.54e-06	55	0.00e+00	1.60e+00	9.54e-06
17	0.00e+00	Inf	9.54e-06	56	0.00e+00	1.66e+00	9.54e-06
18	0.00e+00	Inf	9.54e-06	57	0.00e+00	1.76e+00	9.54e-06
19	0.00e+00	Inf	9.54e-06	58	0.00e+00	2.87e+00	9.54e-06
20	0.00e+00	Inf	9.54e-06	59	0.00e+00	3.00e+00	9.54e-06
21	0.00e+00	Inf	9.54e-06	60	0.00e+00	3.23e+00	9.54e-06
22	0.00e+00	Inf	9.54e-06	61	0.00e+00	Inf	9.54e-06
23	0.00e+00	Inf	9.54e-06	62	1.09e-02	Inf	9.54e-06
24	0.00e+00	Inf	9.54e-06	63	0.00e+00	Inf	9.54e-06
25	0.00e+00	Inf	9.54e-06	64	0.00e+00	Inf	9.54e-06
26	0.00e+00	Inf	9.54e-06	65	0.00e+00	Inf	9.54e-06
27	0.00e+00	Inf	9.54e-06	66	0.00e+00	Inf	9.54e-06
28	3.21e-02	Inf	9.54e-06	67	0.00e+00	Inf	9.54e-06
29	0.00e+00	Inf	9.54e-06	68	0.00e+00	Inf	9.54e-06
30	0.00e+00	Inf	9.54e-06	69	0.00e+00	Inf	9.54e-06
31	0.00e+00	Inf	9.54e-06	70	0.00e+00	Inf	9.54e-06
32	0.00e+00	Inf	9.54e-06	71	0.00e+00	Inf	9.54e-06
33	0.00e+00	Inf	9.54e-06	72	0.00e+00	Inf	9.54e-06
34	0.00e+00	Inf	9.54e-06	73	0.00e+00	Inf	9.54e-06
35	0.00e+00	Inf	9.54e-06	74	0.00e+00	Inf	9.54e-06
36	0.00e+00	Inf	9.54e-06	75	0.00e+00	Inf	9.54e-06
37	0.00e+00	Inf	9.54e-06	76	0.00e+00	Inf	9.54e-06
38	0.00e+00	Inf	9.54e-06	77	0.00e+00	Inf	9.54e-06
39	0.00e+00	Inf	9.54e-06	78	0.00e+00	Inf	9.54e-06

Table B.2: Entries for the upper bound variables for the SLP script with varying Trust region size.

Entry	l	$bu - x_i$	ubound	Entry	l	$bu - x_i$	ubound
1	8.22e+00	0.00e+00	0.00e+00	40	0.00e+00	Inf	9.54e-06
2	0.00e+00	2.65e-03	9.54e-06	41	0.00e+00	Inf	9.54e-06
3	0.00e+00	1.34e-02	9.54e-06	42	0.00e+00	Inf	9.54e-06
4	0.00e+00	5.78e-03	9.54e-06	43	0.00e+00	Inf	9.54e-06
5	0.00e+00	1.56e-02	9.54e-06	44	0.00e+00	Inf	9.54e-06
6	7.54e+01	0.00e+00	0.00e+00	45	0.00e+00	Inf	9.54e-06
7	0.00e+00	1.05e-02	9.54e-06	46	0.00e+00	Inf	9.54e-06
8	7.76e+01	0.00e+00	0.00e+00	47	0.00e+00	Inf	9.54e-06
9	0.00e+00	2.82e-02	9.54e-06	48	0.00e+00	Inf	9.54e-06
10	0.00e+00	Inf	9.54e-06	49	0.00e+00	Inf	9.54e-06
11	0.00e+00	Inf	9.54e-06	50	0.00e+00	Inf	9.54e-06
12	0.00e+00	Inf	9.54e-06	51	0.00e+00	Inf	9.54e-06
13	0.00e+00	Inf	9.54e-06	52	0.00e+00	Inf	9.54e-06
14	0.00e+00	Inf	9.54e-06	53	0.00e+00	Inf	9.54e-06
15	0.00e+00	Inf	9.54e-06	54	0.00e+00	Inf	9.54e-06
16	0.00e+00	Inf	9.54e-06	55	0.00e+00	1.60e+00	9.54e-06
17	0.00e+00	Inf	9.54e-06	56	0.00e+00	1.66e+00	9.54e-06
18	0.00e+00	Inf	9.54e-06	57	0.00e+00	1.76e+00	9.54e-06
19	0.00e+00	Inf	9.54e-06	58	0.00e+00	2.87e+00	9.54e-06
20	0.00e+00	Inf	9.54e-06	59	0.00e+00	3.00e+00	9.54e-06
21	0.00e+00	Inf	9.54e-06	60	0.00e+00	3.23e+00	9.54e-06
22	0.00e+00	Inf	9.54e-06	61	0.00e+00	Inf	9.54e-06
23	0.00e+00	Inf	9.54e-06	62	0.00e+00	Inf	9.54e-06
24	0.00e+00	Inf	9.54e-06	63	0.00e+00	Inf	9.54e-06
25	3.95e-03	Inf	9.54e-06	64	0.00e+00	Inf	9.54e-06
26	0.00e+00	Inf	9.54e-06	65	0.00e+00	Inf	9.54e-06
27	0.00e+00	Inf	9.54e-06	66	0.00e+00	Inf	9.54e-06
28	0.00e+00	Inf	9.54e-06	67	0.00e+00	Inf	9.54e-06
29	0.00e+00	Inf	9.54e-06	68	0.00e+00	Inf	9.54e-06
30	0.00e+00	Inf	9.54e-06	69	0.00e+00	Inf	9.54e-06
31	0.00e+00	Inf	9.54e-06	70	0.00e+00	Inf	9.54e-06
32	0.00e+00	Inf	9.54e-06	71	0.00e+00	Inf	9.54e-06
33	0.00e+00	Inf	9.54e-06	72	0.00e+00	Inf	9.54e-06
34	0.00e+00	Inf	9.54e-06	73	0.00e+00	Inf	9.54e-06
35	0.00e+00	Inf	9.54e-06	74	0.00e+00	Inf	9.54e-06
36	0.00e+00	Inf	9.54e-06	75	0.00e+00	Inf	9.54e-06
37	0.00e+00	Inf	9.54e-06	76	0.00e+00	Inf	9.54e-06
38	0.00e+00	Inf	9.54e-06	77	0.00e+00	Inf	9.54e-06
39	0.00e+00	Inf	9.54e-06	78	0.00e+00	Inf	9.54e-06

Table B.3: Entries for the upper bound variables for the SLP script that ignores the changes in constraint ratios.

Appendix C

MATLAB SQP Code

In this Appendix we include the MATLAB code for implementing the Sequential Quadratic Programming method to the 9 bus case.

C.1 SQP Script

```
clear all;

%load data from AMPL Model
[x,bl,bu,v,cl,cu] = amplfunc('case9.nl');

% define inital point for SLP:
x_i = zeros(length(bl),1);
x_i(1:9)= ones(9,1);

% define inital size of trustregion
trust = 5;

% define an inital large current constraint violation
d = 1;
lam = zeros(size(cl));

% set iteration counter to 1
i = 1;
fid = fopen('iteration.txt','w');
fprintf(fid,'Iteration & Trust region &Current Objective & predicted new objective & actual new objective &Current Constraint violation & new constraint violation & Constraint ratio & objective ratio \\\\ \n');

while norm(d,inf) > 1e-5

    % adjust the trust region to fit the potential changes made to the
    % trust region.
    lbound = -min(trust, abs(bl-x_i));
    ubound = min(trust, abs(bu-x_i));

    % find the next trial point x_i_temp
```

```
    [x_i_temp, g_i, f_i,predicted_obj,d,lam] = SQP(x_i,cu,cl,lbound,
    ubound,lam);

    trust_old = trust;
    % test progress of temporary x_i value
    progress_test_SQP;

    % display the important values at the current iteration
    disp(sprintf('%4d %8.5g %8.5g %8.5g %8.5g %8.5g %8.5g %8.5g %8.5
    g\n', ...
        i, trust_old, f_i, predicted_obj, f_i_new, cv_old, cv_new,
    ...
        constraint_ratio, objective_ratio));

    % increase the iteration number by one
    i = i + 1;

end

% evaluate the objective and the constraints at the solution found
    by SLP
[f_i, g_i] = amplfunc(x_i,0);
[nabla_f_i, nabla_g_i] = amplfunc(x_i,1);

% evaluate and define the Coefficient matrix of the constraints and
    the
% upper bound vector b
A = [ nabla_g_i;-nabla_g_i];
b = [cu-g_i;-cl+g_i];
W = amplfunc(-v);
%remove infinity values
infx = (b<inf);
A2 = A(infx,:);
b2 = b(infx);
% evaluate the Lagrangian multipliers
[d,fval,exitflag,output,lambda] = quadprog(W, nabla_f_i.',A2,b2
    ,[],[],lbound,...
    ubound, zeros(size(x_i)));

% reinsert the infinity constraints:
lam = zeros(size(b));
m = length(b);
lamtmp(infx) = lambda.ineqlin';
lam = lamtmp(1:m/2)+lamtmp(m/2+1:m);
lam = lam';


% evaluate the testing condition for Lagrangian duality
dual_test = (nabla_f_i) + (lambda.ineqlin' *A2)' - lambda.lower +
    lambda.upper;

%print a table with the testing condtion, the lagrangian multipliers
    and
%the bounds of the problem.
table(dual_test, lambda.lower,lambda.upper, x_i-bl, bu-x_i, lbound,
    ubound,...
```

```
    'VariableNames',{'dual_test','lambdalower','lambdaupper',...
    'x_iminusbl', 'buminusx_i','lbound','ubound'})


% print a table to test KKT condition for lagrange multiplier that is
% used for the constraint inequalities:
table(lam,g_i,cl,cu, 'VariableNames',{'LagrangeMultiplier', 'ConstraintsAtSolution',...
    'LowerBound','UpperBound'})
```

C.2 SQP Function

```
function [x_i_temp, g_i, f_i, predicted_obj,d,lam] = SQP(x_i,cu,cl, lbound, ubound, v)
% This function performs one iteration of SLP.

% evaluate the objective and the constraints at the current trial point
[f_i, g_i] = amplfunc(x_i,0);
[nabla_f_i, nabla_g_i] = amplfunc(x_i,1);
W = amplfunc(-v);
% evaluate and define the Coefficient matrix of the constraints and the
% upper bound vector b
A = [ nabla_g_i;-nabla_g_i];
b = [cu-g_i;-cl+g_i];

%remove infinity values
infx = (b<inf);
A2 = A(infx,:);
b2 = b(infx);

%change settings of quadprog so no outputs are diplayed
options = optimset('quadprog');
options.Display = 'off';

% solve the SQP
[d, fval, exitflag, output, lambda] = quadprog(W, nabla_f_i.',A2,b2,[],[],lbound,...
    ubound, zeros(size(x_i)), options);

% define the new dual variables at d, that we will need to evaluate the
% Hessian W in the next iteration.
lam = zeros(size(b));
m = length(b);
lamtmp(infx) = lambda.ineqlin';
lam = lamtmp(1:m/2)+lamtmp(m/2+1:m);
lam = lam';

% define the predicted objective at x_i_temp
predicted_obj = f_i+nabla_f_i'*d + 0.5*d'*W*d;

```

```
% define temporary x_i value to test progress before taking step
x_i_temp = d + x_i;
end
```

C.3 Progress Test

```
% evaluate the objective function and the constraints at the new trial
% point x_i_temp
[f_i_new, g_i_new] = amplfunc(x_i_temp,0);

% evaluate constraint violations
cv_old = sum(abs(max(g_i-cu,0))) + sum(abs(min(g_i-cl,0)));
cv_new = sum(abs(max(g_i_new-cu,0))) + sum(abs(min(g_i_new-cl,0)));

% define the constraint ratio
constraint_ratio = (cv_old - cv_new)/cv_old;

% define the objective ratio
objective_ratio = (f_i - f_i_new)./(f_i-predicted_obj);

% start testing conditions

if cv_old > 1e-12
    % if both ratios are greater than 0.75 we take the step and increase the
    % trust region
     if min(constraint_ratio, objective_ratio) > 0.75
         trust = trust*2;
         x_i = x_i_temp;

         % if either of the ratios is lower than 0.05 we reduce the trust region and
         % reject the new trial point
     elseif min(constraint_ratio, objective_ratio) < 0.05
         trust = trust/2;

         % if both constraints are within the range of 0.05 to 0.75 we take the step
         % without cahnging the trust region.
     else
         x_i = x_i_temp;
    end
else
    % if both ratios are greater than 0.75 we take the step and increase the
    % trust region
     if objective_ratio > 0.75
         trust = trust*2;
         x_i = x_i_temp;

         % if either of the ratios is lower than 0.05 we reduce the trust region and
         % reject the new trial point
```

```
    elseif objective_ratio < 0.05
        trust = trust/2;

        % if both constraints are within the range of 0.05 to 0.75
    we take the step
        % without cahnging the trust region.
    else
        x_i = x_i_temp;
    end
end
```

C.4 Duality Condition

Entry	Dual Test	Entry	Dual Test	Entry	Dual Test
1	3.997e-13	27	-1.137e-13	53	-4.041e-19
2	4.550e-13	28	4.069e-19	54	-4.004e-19
3	6.822e-13	29	-3.894e-19	55	6.395e-05
4	9.095e-13	30	4.547e-13	56	-3.042e-05
5	2.728e-12	31	4.547e-13	57	-2.945e-05
6	-1.734e-12	32	8.527e-14	58	-4.580e-19
7	-9.095e-13	33	-4.547e-13	59	-3.798e-19
8	1.464e-12	34	-3.891e-19	60	4.707e-19
9	9.095e-13	35	4.547e-13	61	-2.416e-13
10	0.000e+00	36	3.916e-19	62	-3.895e-19
11	7.276e-12	37	-4.578e-19	63	6.395e-14
12	-9.095e-13	38	-1.421e-14	64	0.000e+00
13	1.455e-11	39	-4.267e-19	65	1.563e-13
14	-3.638e-12	40	4.706e-19	66	0.000e+00
15	1.455e-11	41	4.111e-19	67	1.705e-13
16	1.455e-11	42	4.092e-19	68	0.000e+00
17	-7.276e-12	43	3.806e-19	69	-1.990e-13
18	7.276e-12	44	-2.842e-14	70	-1.421e-14
19	1.066e-13	45	4.038e-19	71	-3.798e-19
20	3.885e-19	46	4.598e-19	72	1.421e-14
21	3.892e-19	47	4.265e-19	73	0.000e+00
22	-3.904e-19	48	1.421e-14	74	0.000e+00
23	3.876e-19	49	-4.749e-19	75	0.000e+00
24	2.203e-13	50	-4.092e-19	76	0.000e+00
25	3.897e-19	51	-1.421e-14	77	0.000e+00
26	3.482e-13	52	-3.796e-19	78	0.000e+00

Table C.1: Entries for the `dual_test` variables for the SQP script.

Entry	k	$x_i - bl$	lbound	Entry	k	$x_i - bl$	lbound
1	0.00e+00	2.00e-01	-2.00e-01	40	0.00e+00	Inf	-6.25e-01
2	0.00e+00	1.97e-01	-1.97e-01	41	0.00e+00	Inf	-6.25e-01
3	0.00e+00	1.87e-01	-1.87e-01	42	0.00e+00	Inf	-6.25e-01
4	0.00e+00	1.94e-01	-1.94e-01	43	0.00e+00	Inf	-6.25e-01
5	0.00e+00	1.84e-01	-1.84e-01	44	0.00e+00	Inf	-6.25e-01
6	0.00e+00	2.00e-01	-2.00e-01	45	0.00e+00	Inf	-6.25e-01
7	0.00e+00	1.89e-01	-1.89e-01	46	0.00e+00	Inf	-6.25e-01
8	0.00e+00	2.00e-01	-2.00e-01	47	0.00e+00	Inf	-6.25e-01
9	0.00e+00	1.72e-01	-1.72e-01	48	0.00e+00	Inf	-6.25e-01
10	0.00e+00	Inf	-6.25e-01	49	4.75e-19	Inf	-6.25e-01
11	3.88e-19	Inf	-6.25e-01	50	4.09e-19	Inf	-6.25e-01
12	3.89e-19	Inf	-6.25e-01	51	4.08e-19	Inf	-6.25e-01
13	3.90e-19	Inf	-6.25e-01	52	3.80e-19	Inf	-6.25e-01
14	3.90e-19	Inf	-6.25e-01	53	4.04e-19	Inf	-6.25e-01
15	3.90e-19	Inf	-6.25e-01	54	4.00e-19	Inf	-6.25e-01
16	3.90e-19	Inf	-6.25e-01	55	0.00e+00	7.98e-01	-6.25e-01
17	3.89e-19	Inf	-6.25e-01	56	7.53e-19	1.24e+00	-6.25e-01
18	3.89e-19	Inf	-6.25e-01	57	4.51e-19	8.42e-01	-6.25e-01
19	0.00e+00	Inf	-6.25e-01	58	4.58e-19	3.13e+00	-6.25e-01
20	0.00e+00	Inf	-6.25e-01	59	3.80e-19	3.00e+00	-6.25e-01
21	0.00e+00	Inf	-6.25e-01	60	0.00e+00	2.77e+00	-6.25e-01
22	3.90e-19	Inf	-6.25e-01	61	1.41e-18	Inf	-6.25e-01
23	0.00e+00	Inf	-6.25e-01	62	3.90e-19	Inf	-6.25e-01
24	3.88e-19	Inf	-6.25e-01	63	3.94e-19	Inf	-6.25e-01
25	0.00e+00	Inf	-6.25e-01	64	0.00e+00	Inf	-6.25e-01
26	3.92e-19	Inf	-6.25e-01	65	0.00e+00	Inf	-6.25e-01
27	3.93e-19	Inf	-6.25e-01	66	0.00e+00	Inf	-6.25e-01
28	0.00e+00	Inf	-6.25e-01	67	0.00e+00	Inf	-6.25e-01
29	3.89e-19	Inf	-6.25e-01	68	0.00e+00	Inf	-6.25e-01
30	3.90e-19	Inf	-6.25e-01	69	0.00e+00	Inf	-6.25e-01
31	0.00e+00	Inf	-6.25e-01	70	4.58e-19	Inf	-6.25e-01
32	0.00e+00	Inf	-6.25e-01	71	3.80e-19	Inf	-6.25e-01
33	0.00e+00	Inf	-6.25e-01	72	0.00e+00	Inf	-6.25e-01
34	3.89e-19	Inf	-6.25e-01	73	0.00e+00	Inf	-6.25e-01
35	0.00e+00	Inf	-6.25e-01	74	0.00e+00	Inf	-6.25e-01
36	0.00e+00	Inf	-6.25e-01	75	0.00e+00	Inf	-6.25e-01
37	4.58e-19	Inf	-6.25e-01	76	0.00e+00	Inf	-6.25e-01
38	4.22e-19	Inf	-6.25e-01	77	0.00e+00	Inf	-6.25e-01
39	4.27e-19	Inf	-6.25e-01	78	0.00e+00	Inf	-6.25e-01

Table C.2: Entries for the lower bound variables for the SQP script.

Entry	l	$bu - x_i$	ubound	Entry	l	$bu - x_i$	ubound
1	8.22e+00	4.44e-16	4.44e-16	40	4.71e-19	Inf	6.25e-01
2	2.35e-16	2.65e-03	2.65e-03	41	4.11e-19	Inf	6.25e-01
3	4.44e-17	1.34e-02	1.34e-02	42	4.09e-19	Inf	6.25e-01
4	5.20e-17	5.78e-03	5.78e-03	43	3.81e-19	Inf	6.25e-01
5	1.18e-17	1.56e-02	1.56e-02	44	4.08e-19	Inf	6.25e-01
6	7.54e+01	0.00e+00	0.00e+00	45	4.04e-19	Inf	6.25e-01
7	1.62e-17	1.05e-02	1.05e-02	46	4.60e-19	Inf	6.25e-01
8	7.76e+01	0.00e+00	0.00e+00	47	4.26e-19	Inf	6.25e-01
9	6.92e-18	2.82e-02	2.82e-02	48	4.29e-19	Inf	6.25e-01
10	0.00e+00	Inf	6.25e-01	49	0.00e+00	Inf	6.25e-01
11	0.00e+00	Inf	6.25e-01	50	0.00e+00	Inf	6.25e-01
12	0.00e+00	Inf	6.25e-01	51	0.00e+00	Inf	6.25e-01
13	0.00e+00	Inf	6.25e-01	52	0.00e+00	Inf	6.25e-01
14	0.00e+00	Inf	6.25e-01	53	0.00e+00	Inf	6.25e-01
15	0.00e+00	Inf	6.25e-01	54	0.00e+00	Inf	6.25e-01
16	0.00e+00	Inf	6.25e-01	55	3.90e-19	1.60e+00	6.25e-01
17	0.00e+00	Inf	6.25e-01	56	0.00e+00	1.66e+00	6.25e-01
18	0.00e+00	Inf	6.25e-01	57	0.00e+00	1.76e+00	6.25e-01
19	3.93e-19	Inf	6.25e-01	58	0.00e+00	2.87e+00	6.25e-01
20	3.88e-19	Inf	6.25e-01	59	0.00e+00	3.00e+00	6.25e-01
21	3.89e-19	Inf	6.25e-01	60	4.71e-19	3.23e+00	6.25e-01
22	0.00e+00	Inf	6.25e-01	61	0.00e+00	Inf	6.25e-01
23	3.88e-19	Inf	6.25e-01	62	0.00e+00	Inf	6.25e-01
24	0.00e+00	Inf	6.25e-01	63	0.00e+00	Inf	6.25e-01
25	3.90e-19	Inf	6.25e-01	64	0.00e+00	Inf	6.25e-01
26	0.00e+00	Inf	6.25e-01	65	0.00e+00	Inf	6.25e-01
27	0.00e+00	Inf	6.25e-01	66	0.00e+00	Inf	6.25e-01
28	4.07e-19	Inf	6.25e-01	67	0.00e+00	Inf	6.25e-01
29	0.00e+00	Inf	6.25e-01	68	0.00e+00	Inf	6.25e-01
30	0.00e+00	Inf	6.25e-01	69	0.00e+00	Inf	6.25e-01
31	3.89e-19	Inf	6.25e-01	70	0.00e+00	Inf	6.25e-01
32	3.88e-19	Inf	6.25e-01	71	0.00e+00	Inf	6.25e-01
33	3.88e-19	Inf	6.25e-01	72	4.71e-19	Inf	6.25e-01
34	0.00e+00	Inf	6.25e-01	73	0.00e+00	Inf	6.25e-01
35	3.92e-19	Inf	6.25e-01	74	0.00e+00	Inf	6.25e-01
36	3.92e-19	Inf	6.25e-01	75	0.00e+00	Inf	6.25e-01
37	0.00e+00	Inf	6.25e-01	76	0.00e+00	Inf	6.25e-01
38	0.00e+00	Inf	6.25e-01	77	0.00e+00	Inf	6.25e-01
39	0.00e+00	Inf	6.25e-01	78	0.00e+00	Inf	6.25e-01

Table C.3: Entries for the upper bound variables for the SQP script.

Basic script	TR variation	Ignore CV	SQP
1.1000000000	1.1000000000	1.1000000000	1.1000000000
1.0973546211	1.0973546200	1.0973546200	1.0973546244
1.0866202973	1.0866203063	1.0866203166	1.0866203031
1.0942215245	1.0942215274	1.0942215275	1.0942215156
1.0844485017	1.0844485095	1.0844485147	1.0844484947
1.1000000000	1.1000000000	1.1000000000	1.1000000000
1.0894894845	1.0894894839	1.0894894832	1.0894894841
1.1000000000	1.1000000000	1.1000000000	1.1000000000
1.0717554655	1.0717554614	1.0717554522	1.0717554475
0.0000000000	0.0000000000	0.0000000000	0.0000000000
⋮	⋮	⋮	⋮

Table C.4: Table giving the first 10 entries of the 9 bus case for the 4 different solvers we have implemented in this report.