Tutorial 2: Three-Phase Optimal Power Flow Formulation and Its Pyomo/Python Implementation[1]

Pedro P. Vergara

Intelligent Electrical Power Grids (IEPG) Group, Delft University of Technology, the Netherlands.

Email: p.p.vergarabarrios@tudelft.nl

Notation

Sets:

$\mathcal{F}$	Set of phases {A, B, C},
$\mathcal{L}$	Set of lines,
$\mathcal{N}$	Set of nodes of the distribution network,
$\mathcal{T}$	Set of time-periods.

Indexes:

ϕ, ψ	Phases $\phi \in \mathcal{F}$ and $\psi \in \mathcal{F}$,
mn	Line $mn \in \mathcal{L}$,
n, m	Nodes $n \in \mathcal{N}$ and $m \in \mathcal{N}$,
t	Time step $t \in \mathcal{T}$.

Parameters:

$\overline{I}_{mn}$	Maximum lines current limit,
$P^D_{m,\phi,t}$	Active power demand consumption,
$Q^D_{m,\phi,t}$	Reactive power demand consumption,
V_0	Nominal voltage magnitude,
$Z_{mn,\phi,\psi}$	Line impedance,
$Z'_{mn,\phi,\psi}$	Transformed line impedance, defined as $Z'_{mn,\phi,\psi} = Z_{mn,\phi,\psi}\underline{/\theta_\psi - \theta_\phi}$.

Continuous Variables:

$P_{mn,\phi}$	Active power flow in the lines,
$P^L_{mn,\phi}$	Active power losses in the lines,
$Q_{mn,\phi}$	Reactive power flow in the lines,
$Q^L_{mn,\phi}$	Active power losses in the lines,
$S_{mn,\phi}$	Apparent power flow in lines,
$S^L_{mn,\phi}$	Apparent power losses in lines,
$V_{m,\phi,t}$	Voltage magnitude.

1. Introduction

This tutorial provides an introduction to the three-phase optimal power flow formulation, used to define the state (i.e. node voltage magnitude, power flowing in the lines, etc.) of the electrical grid in distribution

[1]This document is based on the publication: P. P. Vergara, J. C. López, M. J. Rider and L. C. P. da Silva, "Optimal Operation of Unbalanced Three-Phase Islanded Droop-Based Microgrids," in IEEE Transactions on Smart Grid, vol. 10, no. 1, pp. 928-940, Jan. 2019, doi: 10.1109/TSG.2017.2756021. If you are using this document, I kindly request to cite this reference in your work.

systems. The power flow formulation is the first step to develop and test new dispatch and control algorithms in power systems [1]. The formulation developed in this document is a function of the real (active power) and imaginary (reactive power) of the complex power flowing in the lines of a distribution system, leading to a mathematical formulation that falls within the nonlinear programming (NLP) area. Other formulations, based for instance in the real and imaginary parts of the voltage and current, are also available [2]. Initially, we present the mathematical derivation to obtain the full formulation, composed of four main groups of expressions: (i) the active and reactive power losses in lines, (ii) the active and reactive node's power balance, (iii) the voltage drop in lines, and (iv) the current magnitude limits flowing in the lines. After presenting the mathematical three-phase optimal power flow (OPF) formulation, we will implement it in Pyomo/Python. As this formulation is classified as an NLP formulation, we will need to use a solver suitable for such type of problems.

2. Three-Phase Optimal Power Flow Formulation

Before stating the objective function (in this case the active power losses) of a three-phase power flow formulation, first, we need to derive a general expression for the power losses in a line. To do this, consider the model of the three-phase line that goes from node m to node n presented in Fig. 1. We can derive the active power losses as the real part of the apparent i.e., $P^L_{mn,\phi} = \Re\{S^L_{mn,\phi}\}$. Thus, to derive the complex (or apparent) power losses, consider that $S^L_{mn,\phi,t}$ can be defined as,

$$S^L_{mn,\phi} = \Delta V_{mn,\phi} I^*_{mn,\phi}, \tag{1}$$

where the three-phase voltage magnitude drop ($\Delta V_{mn,\phi}$) in line mn is given by

$$\Delta V_{mn,\phi} = V_{m,\phi} - V_{n,\phi} = \sum_{\psi \in \mathcal{F}} Z_{mn,\phi,\psi} I_{mn,\psi}, \tag{2}$$

and the three-phase current ($I_{mn,\psi}$) in line mn is,

$$I_{mn,\psi} = \frac{S^*_{mn,\psi}}{V^*_{m,\psi}}. \tag{3}$$

Notice that the expression in (2) models the voltage drop between line mn in phase ϕ as a function of the *self-* and the *mutual-impedance* (see Fig. 1) of the same phase and the current flowing in the line, while expression in (3) is the definition of complex current (as a function of the complex power and the voltage) that you learn in an introductory electrical engineering course[2].

If we replace the expression in (2) into the expression (1), we obtain

$$S^L_{mn,\phi,t} = (\sum_{\psi \in \mathcal{F}} Z_{mn,\phi,\psi} I_{mn,\psi,t}) I^*_{mn,\phi,t}. \tag{4}$$

Notice that by replacing (2) into (1), the resultant expression in only a function of the line impedance and the line current. As we are aiming to define a power flow formulation that is a function of the active ($P_{mn,\phi}$)

[2]See, for examples, [3].

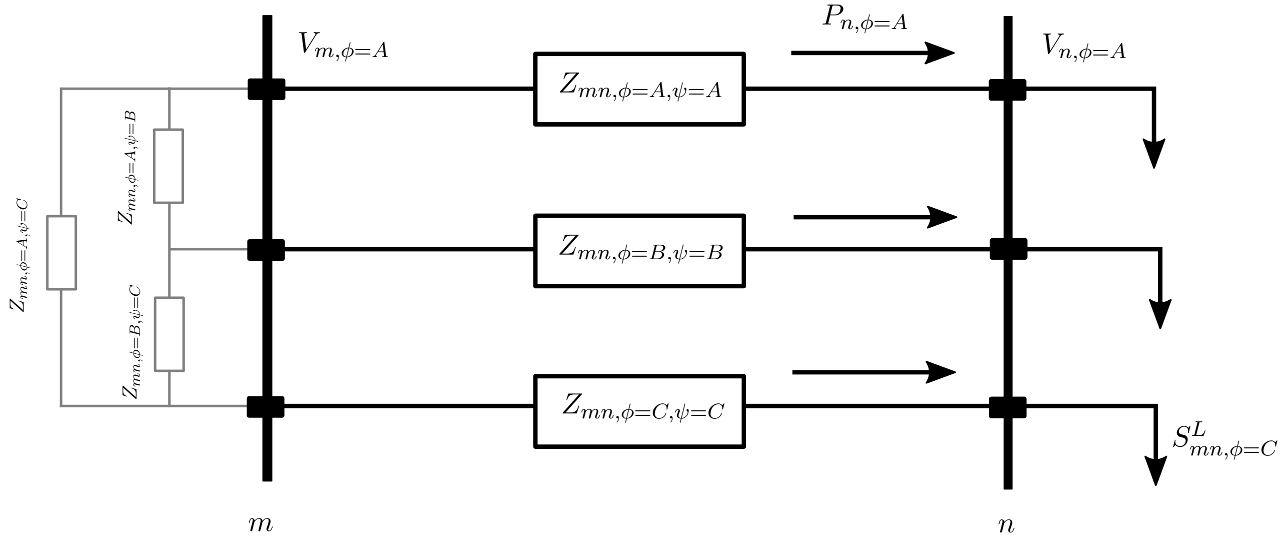

Figure 1: Representation of the three-phase (phases $\phi, \psi \in \mathcal{F} = \{A, B, C\}$ line between the nodes m and n. $Z_{mn,\phi,\psi}$ represents the line impedance of the line that goes from node m to n. If $\phi = \psi$ (e. g. $Z_{mn,\phi=A,\psi=A}$) is known as *self-impedance*, otherwise, if $\phi \neq \psi$, (e. g. $Z_{mn,\phi=A,\psi=B}$) is known as *mutual-impedance* between phase ϕ and ψ.

and reactive ($Q_{mn,\phi}$) and the nodes voltage ($V_{m,\phi}, V_{n,\phi}$), we can replace the expression in (3) into (4)[3], and obtain

$$S^L_{mn,\phi} = \left(\sum_{\psi \in \mathcal{F}} Z_{mn,\phi,\psi} I_{mn,\psi} \right) \frac{S_{mn,\phi,t}}{V_{m,\phi}}. \tag{5}$$

Applying the same procedure in the expression (4) i.e., using (3) to define the line current $I_{mn,\psi}$, we get the next expression

$$S^L_{mn,\phi} = \left(\sum_{\psi \in \mathcal{F}} Z_{mn,\phi,\psi} \left(\frac{S^*_{mn,\psi}}{V^*_{m,\psi}} \right) \right) \frac{S_{mn,\phi}}{V_{m,\phi}}. \tag{6}$$

The terms outside the round brackets are defined for phase ϕ, while the sum is defined for phase ψ (in which $\psi \in \mathcal{F}$). Based on this, we can move the voltage term to inside the round brackets without modifying this expression, obtaining

$$S^L_{mn,\phi,t} = \left(\sum_{\psi \in \mathcal{F}} \frac{Z_{mn,\phi,\psi}}{V^*_{m,\psi} V_{m,\phi}} S^*_{mn,\psi} \right) S_{mn,\phi}. \tag{7}$$

To simplify the expression in (7), we will write the node voltages in terms of their magnitude and angle, i. e. $V_{m,\psi} = |V_{m,\psi,t}| \underline{/ \theta_\psi}$, $V_{m,\phi} = |V_{m,\phi,t}| \underline{/ \theta_\phi}$, obtaining

$$S^L_{mn,\phi,t} = \left(\sum_{\psi \in \mathcal{F}} \frac{Z_{mn,\phi,\psi}}{|V_{m,\psi}| \underline{/ -\theta_\psi} |V_{m,\phi}| \underline{/ \theta_\phi}} S^*_{mn,\psi} \right) S_{mn,\phi}. \tag{8}$$

To multiply two complex number when they are expressed in term of their magnitude and angle, we simple multiply their magnitude and sum their angle (notice the negative in front of θ_ψ due to the conjugate in the

[3] Notice that $I^*_{mn,\phi} = S_{mn,\phi}/V_{m,\phi}$.

expression (7)), obtaining

$$S^L_{mn,\phi,t} = \left(\sum_{\psi \in \mathcal{F}} \frac{Z_{mn,\phi,\psi}}{|V_{m,\psi}||V_{m,\phi}|\underline{/\theta_\phi - \theta_\psi}} S^*_{mn,\psi} \right) S_{mn,\phi}. \tag{9}$$

We can reduce expression (9) even further if we define a new line impedance $Z_{mn,\phi,\psi}$ as $Z^{'}_{mn,\phi,\psi} = Z_{mn,\phi,\psi}\underline{/\theta_\psi - \theta_\phi}$. After some rearrange, expression in (9) can be written as,

$$S^L_{mn,\phi} = \left(\sum_{\psi \in \mathcal{F}} \frac{Z^{'}_{mn,\phi,\psi}}{|V_{m,\psi}||V_{m,\phi}|} S^*_{mn,\psi} \right) S_{mn,\phi}. \tag{10}$$

which is defined for all lines $mn \in \mathcal{L}$ and for all phases $\phi \in \mathcal{F}$.

The expression in (10) is still a complex expression ($S^L_{mn,\phi}$ is a complex power). We can expand this expression and define it in terms of its real (active power losses, $P^L_{mn,\phi}$) and imaginary (reactive power losses, $Q^L_{mn,\phi}$) part, as $S^L_{mn,\phi} = P^L_{mn,\phi} + jQ^L_{mn,\phi}$. To do this, we will need to expand all other complex expressions (i.e. $Z^{'}_{mn,\phi,\psi}$ and $S^*_{mn,\psi}$) in terms of their real and imaginary part, obtaining the next expression,

$$S^L_{mn,\phi} = \left(\sum_{\psi \in \mathcal{F}} \frac{R^{'}_{mn,\phi,\psi} + jX^{'}_{mn,\phi,\psi}}{|V_{m,\psi}||V_{m,\phi}|} P_{mn,\psi} - jQ_{mn,\psi} \right) S_{mn,\phi}. \tag{11}$$

To reduce expression (11) to a more simple form will require a little bit of more mathematical handling. We leave it for you to strengthen your skills and obtain the expressions that we present next. The expressions for the active and reactive power losses are the ones presented in (12) and (13), respectively. Notice that these expressions are nonlinear due to the product terms of the power flow variables.

$$\begin{aligned} P^L_{mn,\phi} = \sum_{\psi \in \mathcal{F}} \frac{1}{|V_{m,\psi}||V_{m,\phi}|} \Big(& R^{'}_{mn,\phi,\psi} P_{mn,\phi} P_{mn,\psi} + R^{'}_{mn,\phi,\psi} Q_{mn,\phi} Q_{mn,\psi} \\ & + X^{'}_{mn,\phi,\psi} P_{mn,\phi} Q_{mn,\psi} - X^{'}_{mn,\phi,\psi} Q_{mn,\phi} P_{mn,\psi} \Big), \end{aligned} \tag{12}$$

$$\begin{aligned} Q^L_{mn,\phi} = \sum_{\psi \in \mathcal{F}} \frac{1}{|V_{m,\psi}||V_{m,\phi}|} \Big(& -R^{'}_{mn,\phi,\psi} P_{mn,\phi} Q_{mn,\psi} + R^{'}_{mn,\phi,\psi} Q_{mn,\phi} P_{mn,\psi} \\ & + X^{'}_{mn,\phi,\psi} P_{mn,\phi} P_{mn,\psi} + X^{'}_{mn,\phi,\psi} Q_{mn,\phi} Q_{mn,\psi} \Big). \end{aligned} \tag{13}$$

The expression in (12) (i.e. the active power losses) is usually regarded as the *objective function* in a basic OPF formulation. We will use this expression later when we code the three-phase OPF and solve it in Pyomo/Python.

To derive the second group of the mathematical expression for the three-phase OPF formulation, i.e., the active and reactive node's power balance, consider first the power balance shown in Fig. 2. For each node m, the total amount of power *entering* that node, coming from all lines that connect nodes k with node m, must be equal to the total amount of power *leaving* that node, including the power through the lines that connect node m with other nodes n and the power supplied to the demand connected to that node ($P^D_{m,\phi}$). To

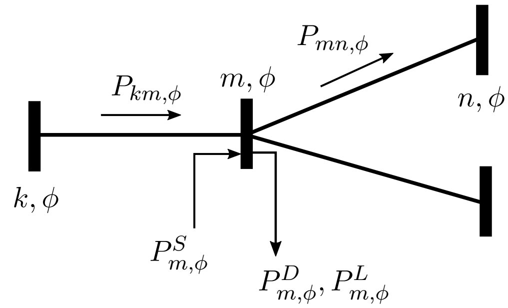

Figure 2: Active power balance in the node m and phase ϕ. The total amount of power *entering* that node must be equal to the total amount of power *leaving* that node.

facilitate implementation in Pyomo/Python, we can assume that a substation is connected to each node m in our distribution network. Of course, the total amount of power that this substation supply to all the nodes in the distribution network must be equal to zero, except for the actual reference node (usually regarded as $m = 1$), which corresponds to the node in which the distribution network is connected to the medium-voltage (MV) network through an MV/LV distribution transformer. Following this reasoning, the expressions for the active and reactive power balance in a distribution network can be modeled as in the expressions in (14) and (15), respectively. Notice that these expressions are linear expressions of the power flow variables and will be added to the OPF formulation as *constraints*.

$$\sum_{km\in\mathcal{L}} P_{km,\phi} - \sum_{mn\in\mathcal{L}} \left(P_{mn,\phi} + P^L_{mn,\phi}\right) + P^S_{m,\phi} = P^D_{m,\phi}, \tag{14}$$

$$\sum_{km\in\mathcal{L}} Q_{km,\phi} - \sum_{mn\in\mathcal{L}} \left(Q_{mn,\phi} + Q^L_{mn,\phi}\right) + Q^S_{m,\phi} = Q^D_{m,\phi}. \tag{15}$$

To model the voltage drop in line mn, we will start with the voltage drop definition presented in expression (2). Replacing expression (3) into (2), we obtain

$$V_{m,\phi} - V_{n,\phi} = \sum_{\psi\in\mathcal{F}} Z_{mn,\phi,\psi} \frac{S^*_{mn,\psi}}{V^*_{m,\psi}}. \tag{16}$$

Expression (16) is already a function of the voltage and power however it is still a complex expression. We will work a little bit more with expression (16) to express it in terms of its real and imaginary parts. To do this, we will start by multiplying both sides by $V^*_{m,\phi,t}$, obtaining

$$V^*_{m,\phi}(V_{m,\phi} - V_{n,\phi}) = \sum_{\psi\in\mathcal{F}} Z_{mn,\phi,\psi} V^*_{m,\phi} \frac{S^*_{mn,\psi}}{V^*_{m,\psi}}. \tag{17}$$

Considering the above-presented definition of $Z'_{mn,\phi,\psi}$ and based on the assumption that the voltage magnitude in the same nodes for different phases is approximately equal, i.e. $|V_{m,\psi,t}| \approx |V_{m,\phi,t}|$ [4][4], we can obtain the next expression,

$$V^*_{m,\phi}(V_{m,\phi} - V_{n,\phi}) = \sum_{\psi\in\mathcal{F}} Z'_{mn,\phi,\psi} S^*_{mn,\psi}. \tag{18}$$

[4]This assumption is only valid as distribution networks are radial.

For the left-hand side of (18), we will expand the expression recalling that the product of a complex number by its conjugate is equal to a real number with a value equal to the square complex number's magnitude. On the right-hand side, we will replace all complex terms in terms of their real and imaginary part. After doing this, we obtain

$$|V_{m,\phi}|^2 - V^*_{m,\phi}V_{n,\phi} = \sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi} + jX^{'}_{mn,\phi,\psi})(P_{mn,\psi} - jQ_{mn,\psi}). \tag{19}$$

We can use the Euler's formula (i.e. $e^{j\theta} = \sin\theta + j\cos\theta$) to expand even further the left-hand side of expression (19), obtaining

$$|V_{m,\phi}|^2 - [|V_{m,\phi}||V_{n,\phi}|\cos\theta_{mn,\phi} + j|V_{m,\phi}||V_{n,\phi}|\sin\theta_{mn,\phi}] = \\ \sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi} + jX^{'}_{mn,\phi,\psi})(P_{mn,\psi} - jQ_{mn,\psi}). \tag{20}$$

In expression (20), we have defined $\theta_{mn,\phi}$ as $\theta_{mn,\phi} = \theta_{n,\phi} - \theta_{m,\phi}$. Notice that we can now split expression (20) into its real and imaginary part in both sides, given us the next two expressions

$$|V_{m,\phi}||V_{n,\phi}|\cos\theta_{mn,\phi} = |V_{m,\phi}|^2 - \sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}P_{mn,\psi} + X^{'}_{mn,\phi,\psi}Q_{mn,\psi}), \tag{21}$$

$$|V_{m,\phi}||V_{n,\phi}|\sin\theta_{mn,\phi} = \sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}Q_{mn,\psi} - X^{'}_{mn,\phi,\psi}P_{mn,\psi}). \tag{22}$$

We can combine once again these two expressions (but now in a real expression, not complex) taking advantage of the fact that $\cos^2\theta_{mn,\phi} + \sin^2\theta_{mn,\phi} = 1$. By doing this, we obtain

$$\left(|V_{m,\phi}|^2 - \sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}P_{mn,\psi} + X^{'}_{mn,\phi,\psi}Q_{mn,\psi})\right)^2 + \left(\sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}Q_{mn,\psi} - X^{'}_{mn,\phi,\psi}P_{mn,\psi})\right)^2 = (|V_{m,\phi}||V_{n,\phi}|)^2. \tag{23}$$

We can expand the first term of (23) obtaining

$$|V_{m,\phi}|^4 - 2|V_{m,\phi}|^2\sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}P_{mn,\psi} + X^{'}_{mn,\phi,\psi}Q_{mn,\psi}) + \left(\sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}P_{mn,\psi} + X^{'}_{mn,\phi,\psi}Q_{mn,\psi})\right)^2 \\ + \left(\sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}Q_{mn,\psi} - X^{'}_{mn,\phi,\psi}P_{mn,\psi})\right)^2 = (|V_{m,\phi}||V_{n,\phi}|)^2. \tag{24}$$

We can multiply both sides of the expression by $1/|V_{m,\phi}|^2$, obtaining

$$|V_{m,\phi}|^2 - 2\sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}P_{mn,\psi} + X^{'}_{mn,\phi,\psi}Q_{mn,\psi}) + \frac{1}{|V_{m,\phi}|^2}\left(\sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}P_{mn,\psi} + X^{'}_{mn,\phi,\psi}Q_{mn,\psi})\right)^2 \\ + \frac{1}{|V_{m,\phi}|^2}\left(\sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}Q_{mn,\psi} - X^{'}_{mn,\phi,\psi}P_{mn,\psi})\right)^2 = |V_{n,\phi}|^2. \tag{25}$$

Re-arranging expression (25), we obtain

$$|V_{m,\phi}|^2-|V_{n,\phi}|^2 = 2\sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}P_{mn,\psi}+X^{'}_{mn,\phi,\psi}Q_{mn,\psi})+\frac{1}{|V_{m,\phi}|^2}\left(\sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}P_{mn,\psi}+X^{'}_{mn,\phi,\psi}Q_{mn,\psi})\right)^2$$
$$+\frac{1}{|V_{m,\phi}|^2}\left(\sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}Q_{mn,\psi}-X^{'}_{mn,\phi,\psi}P_{mn,\psi})\right)^2. \quad (26)$$

Expression (26) correspond to the voltage magnitude drop in lines. We will add this expression in the OPF formulation as an additional *constraint*. Notice that (26) is a nonlinear expression which can be written mathematically more elegant if we recognize that $Z^{'*}_{mn,\phi,\psi}S_{mn\phi} = (R^{'}_{mn,\phi,\psi}+jX^{'}_{mn,\phi,\psi})(P_{mn,\psi}+jQ_{mn,\psi}) = (R^{'}_{mn,\phi,\psi}P_{mn,\psi}+X^{'}_{mn,\phi,\psi}Q_{mn,\psi})+j(R^{'}_{mn,\phi,\psi}Q_{mn,\psi}-X^{'}_{mn,\phi,\psi}P_{mn,\psi})$. Thus, (26) can be re-written as

$$|V_{m,\phi,t}|^2-|V_{n,\phi,t}|^2 = 2\sum_{\psi\in\mathcal{F}}\left(R^{'}_{mn,\phi,\psi}P_{mn,\psi,t}+X^{'}_{mn,\phi,\psi}Q_{mn,\psi,t}\right)-\frac{1}{|V_{m,\phi,t}|^2}\left|\sum_{\psi\in\mathcal{F}}Z^{'*}_{mn,\phi,\psi}S_{mn,\phi,t}\right|^2. \quad (27)$$

To define the current magnitude limit, our last expression for the three-phase OPF formulation, we can limit the current magnitude using the expression (3), obtaining

$$|I_{mn,\phi,t}|^2 \le \overline{I}^2_{mn}. \quad (28)$$

Notice that the current magnitude limit is the same for all phases in the distribution network. We can expand expression (28) based on the definition of apparent power $|I_{mn,\phi}| = |S_{mn,\phi}|/|V_{m,\phi}|$, obtaining

$$\frac{|S_{mn,\psi}|^2}{|V^{*}_{m,\psi}|^2} \le \overline{I}^2_{mn}, \quad (29)$$

which can also be expressed as,

$$(P^2_{mn,\phi}+Q^2_{mn,\phi})/|V_{m,\phi}|^2 \le \overline{I}^2_{mn}. \quad (30)$$

Expression (30) will be added to the OPF formulation also as a *constraint*. Notice that this expression in a linear expression, as $\overline{I}^2_{mn}$ is a parameter of the model.

To summarize, the three-phase OPF formulation is given as the next optimization problem:

$$\min\left\{\sum_{mn\in\mathcal{L},\psi\in\mathcal{F}}P^L_{mn,\psi}\right\}. \quad (31)$$

Subject to the next set of constraints:

$$P^L_{mn,\phi} = \sum_{\psi\in\mathcal{F}}\frac{1}{|V_{m,\psi}||V_{m,\phi}|}\Big(R^{'}_{mn,\phi,\psi}P_{mn,\phi}P_{mn,\psi}+R^{'}_{mn,\phi,\psi}Q_{mn,\phi}Q_{mn,\psi}$$
$$+X^{'}_{mn,\phi,\psi}P_{mn,\phi}Q_{mn,\psi}-X^{'}_{mn,\phi,\psi}Q_{mn,\phi}P_{mn,\psi}\Big), \forall mn\in\mathcal{L},\forall\phi\in\mathcal{F}; \quad (32)$$

$$Q^L_{mn,\phi} = \sum_{\psi \in \mathcal{F}} \frac{1}{|V_{m,\psi}||V_{m,\phi}|} \Big(-R'_{mn,\phi,\psi} P_{mn,\phi} Q_{mn,\psi} + R'_{mn,\phi,\psi} Q_{mn,\phi} P_{mn,\psi}$$
$$+ X'_{mn,\phi,\psi} P_{mn,\phi} P_{mn,\psi} + X'_{mn,\phi,\psi} Q_{mn,\phi} Q_{mn,\psi} \Big), \forall mn \in \mathcal{L}, \forall \phi \in \mathcal{F}; \quad (33)$$

$$\sum_{km \in \mathcal{L}} P_{km,\phi} - \sum_{mn \in \mathcal{L}} \left(P_{mn,\phi} + P^L_{mn,\phi} \right) + P^S_{m,\phi} = P^D_{m,\phi}, \forall m \in \mathcal{N}, \forall \phi \in \mathcal{F}; \quad (34)$$

$$\sum_{km \in \mathcal{L}} Q_{km,\phi} - \sum_{mn \in \mathcal{L}} \left(Q_{mn,\phi} + Q^L_{mn,\phi} \right) + Q^S_{m,\phi} = Q^D_{m,\phi}, \forall m \in \mathcal{N}, \forall \phi \in \mathcal{F}; \quad (35)$$

$$|V_{m,\phi}|^2 - |V_{n,\phi}|^2 = 2 \sum_{\psi \in \mathcal{F}} (R'_{mn,\phi,\psi} P_{mn,\psi} + X'_{mn,\phi,\psi} Q_{mn,\psi}) + \frac{1}{|V_{m,\phi}|^2} \left(\sum_{\psi \in \mathcal{F}} (R'_{mn,\phi,\psi} P_{mn,\psi} + X'_{mn,\phi,\psi} Q_{mn,\psi}) \right)^2$$
$$+ \frac{1}{|V_{m,\phi}|^2} \left(\sum_{\psi \in \mathcal{F}} (R'_{mn,\phi,\psi} Q_{mn,\psi} - X'_{mn,\phi,\psi} P_{mn,\psi}) \right)^2, \forall mn \in \mathcal{L}, \forall \phi \in \mathcal{F}; \quad (36)$$

$$(P^2_{mn,\phi} + Q^2_{mn,\phi}) / |V_{m,\phi}|^2 \leq \overline{I}^2_{mn}, \forall mn \in \mathcal{L}, \forall \phi \in \mathcal{F}. \quad (37)$$

Besides these expressions, in order to solve this OPF model, we need to additionally set $P^S_{m,\phi} = 0$, $\forall m \in \mathcal{N} \neq \{1\}$, and $|V_{m,\phi}| = V_0$, $\forall m \in \mathcal{N} = \{1\}$ for $\phi = \{A\}$, $|V_{m,\phi}| = V_0$, $\forall m \in \mathcal{N} = \{1\}$ for $\phi = \{B\}$, and $|V_{m,\phi}| = V_0$, $\forall m \in \mathcal{N} = \{1\}$ for $\phi = \{C\}$, where V_0 is the nominal voltage magnitude value (i.e., 1 p.u.).

3. Implementation in Pyomo/Python

The implementation of the three-phase OPF formulation derived in Sec. 2 will follow the procedure presented in Fig. 3. In this section, first, we will code a general main python file that will contain the four main functions shown in Fig. 3. These functions are

- `pre_processing_system_data()`,
- `system_data()_for_pyomo()`,
- `three_phase_opf_model()`, and
- `print_results()`.

Then, we will go through each of these functions to learn how to code them. Finally, we will execute the main python file (`main_file.py`) and check the output results in the Python console. I recommend to you to not copy the whole Python code that I will be providing next, but instead code it yourself after understanding what each line actually do and how they are related to the mathematical formulation in (31)–(37). This will help you when you have to advance this code to deploy your own developments. The data related to active and reactive power demand, the distribution network topology and line's impedance (as well as the Python code for each of these functions) are available in my GitHub (`https://github.com/pedropa1/three_phase_opf_formulation`), open to the general public. The distribution network test corresponds to a three-phase 25 nodes system, taken from [5]. Finally, I also recommend to you to check my **Tutorial 1: Implementing an Optimization Model in Pyomo/Python**, so you can have a better understanding of how Pyomo works.

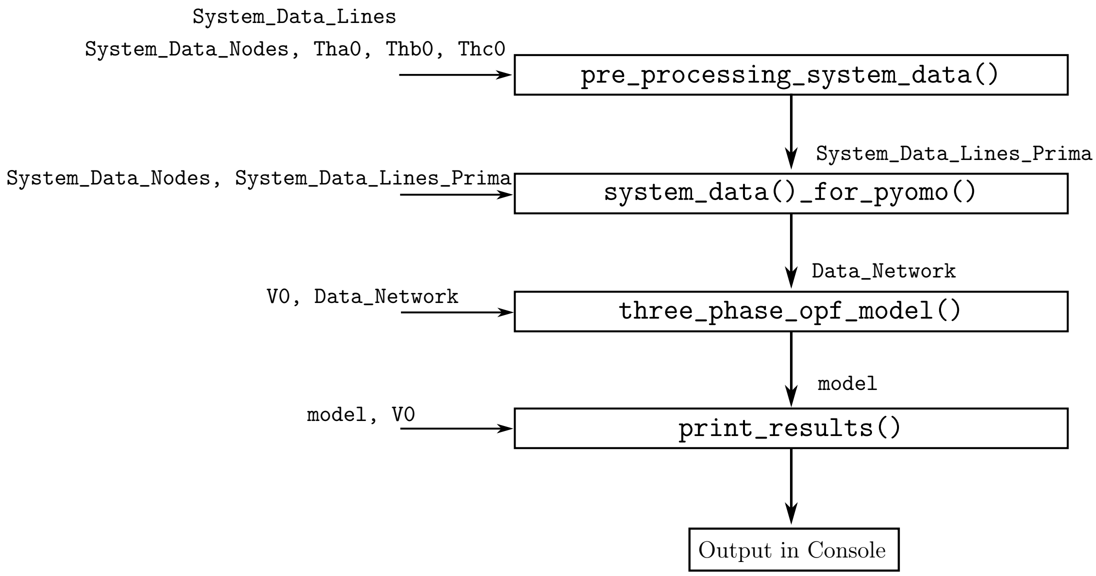

Figure 3: Function structure of the `main_file.py` files composed of four main functions, highlighting the inputs and outputs.

The implementation of the `main_file.py` can be seen in Listing 1. From line 2 to line 9, all the important Python library need it are imported, including the Pyomo environment[5]. From line 13 to 17, initial model parameters that will be used are defined, including the nominal voltage (V_0 as `V0`), and the angular frequency reference for the substation bus ($m = 1$) of each phase $\theta_{\phi=\{A\}} = 0$ as `Tha0 = 0`, $\theta_{\phi=\{B\}} = -2\pi/3$ as `Thb0 = -2.0944` and $\theta_{\phi=\{C\}} = 2\pi/3$ as `Thc0 = 2.0944`. In line 20 and 21, two DataFrames[6] are created using Pandas, one containing all the nodes information (active and reactive power demand, see the file `Nodes_25_3F.xlsx`) and the other all the lines information (node from, node to, and all reactance and resistance per phase, see `Lines_25_3F.xlsx`) of the distribution network. Before continuing I recommend to you to go to my GitHub and check these files quickly. Check for instance the header of the columns in both files (What do you see?).

Functions `pre_processing_system_data()`, `system_data()_for_pyomo()`, and `three_phase_opf_model()` are implemented in lines 24, 27 and 30, respectively. Each of these functions will be discussed in the next sections. Once the Pyomo model has been created (as in line 30), in line 31, we define the optimization solver. As in this case the three-phase OPF formulation falls within a NLP formulation, we can use open-access IPOPT solver[7]. Other solvers such as CPLEX (for linear programming) and GUROBI (for linear and quadratic programming), can also be used. In line 32, we solve our model. If we set the option `tee` for `solver` as `True` we will be able to see how the iterative optimization process is developing and the type of solution that the solver found (more about this in Sec. 3.5). Finally, in line 35, function `print_results()` is implemented, which allow us to check the results after solving the OPF model.

```
1
2 from pyomo.environ import *
3 import system_data_for_pyomo as sd
4 import three_phase_opf_model as pf
5 import pre_processing_system_data as psd
6 import print_results as prnt
7 import math
```

[5] More information available at `http://www.pyomo.org/`
[6] You might need to also have installed the package `xlrd` to extract data from an Excel spreadsheet file.
[7] More information available at `https://coin-or.github.io/Ipopt/`

```python
import pandas as pd
import numpy as np

if __name__=="__main__":

    # General Parameters
    V0 = 4.16/math.sqrt(3)
    Tha0 = 0
    Thb0 = -2.0944
    Thc0 = 2.0944

    # Distribution System Data
    System_Data_Nodes = pd.read_excel('Nodes_25_3F.xlsx')
    System_Data_Lines = pd.read_excel('Lines_25_3F.xlsx')

# ---------  Pre-processing System Data --------------
    System_Data_Lines_Prima = psd.pre_processing_system_data(System_Data_Lines,
   System_Data_Nodes, Tha0, Thb0, Thc0)

# ---------  System Data for Pyomo --------------
    Data_Network = sd.system_data_for_pyomo(System_Data_Nodes, System_Data_Lines_Prima)

# --------- Create the Model --------------
    model =  pf.three_phase_opf_model(V0, Data_Network)
    solver = SolverFactory('ipopt')
    results = solver.solve(model, tee=True)

# --------- Print Results --------------
    prnt.print_results(model,V0)
```

Listing 1: Main Python file to implement the three-phase OPF formulation.

3.1. *Pre-Processing Function*

Listing 2 shows the implementation of the `pre_processing_system_data()` function. This function require as inputs the parameters `Tha0,Thb0, Thc0`, and the DataFrames `System_Data_lines` and `System_Data_Nodes`; and provides as output a DataFrame named as `System_Data_Lines_Prima` as shown in line 53. This DataFrame contains all the $R^{'}_{mn,\phi,\psi}$ and $X^{'}_{mn,\phi,\psi}$ line parameters of the distribution network, recalling that $Z^{'}_{mn,\phi,\psi} = R^{'}_{mn,\phi,\psi} + jX^{'}_{mn,\phi,\psi} = Z_{mn,\phi,\psi}\underline{/\theta_\psi - \theta_\phi}$. To estimate $R^{'}_{mn,\phi,\psi}$ and $X^{'}_{mn,\phi,\psi}$, we will need first to estimate $Z_{mn,\phi,\psi}$ since `System_Data_Lines` only provides $R_{mn,\phi,\psi}$ and $X_{mn,\phi,\psi}$. This procedure is done for all lines in the loop in line 15. As an example, in line 16, the self-impedance of phase A (`Zaa`) is calculated and stored in column 2 of the DataFrame `System_Data_Lines_Impedances`. The remaining self- and mutual-impedance for all phases as well as their angle are calculated in lines 17 to 27.

Once the estimation of $Z_{mn,\phi,\psi}$ is finished and stored in the DataFrame `System_Data_Lines_Impedances`, in the loop in line 33, the DataFrame `System_Data_Lines_Prima` will be filled with the corresponding estimation of $R^{'}_{mn,\phi,\psi}$ and $X^{'}_{mn,\phi,\psi}$ for each line. You can check which parameter is saved in each column if you check first how the DataFrame is created in line 29. For instance, in line 34, the parameter `Raa_p`, that represents $R^{'}_{mn,\phi=A,\psi=A}$, is estimated as $R^{'}_{mn,\phi=A,\psi=A} = Z_{mn,\phi=A,\psi=A}\cos(\theta_{\psi=A} - \theta_{\phi=A})$ and stored in the DataFrames column 2. A similar procedure is done for the remaining parameters in lines 31 to 51. I encourage you to derive the expressions used in lines 31 to 51 so you can understand where they come from. Finally, in line 53, the function returns as output DataFrame `System_Data_Lines_Impedances`.

```
1
2 import pandas as pd
3 import math
4
5 def pre_processing_system_data(System_Data_Lines, System_Data_Nodes, Tha0, Thb0, Thc0):
6
7     # From Ohm to mOhms as everythin is in kW
8     for col in ['Raa', 'Xaa', 'Rbb', 'Xbb', 'Rcc', 'Xcc', 'Rab', 'Xab', 'Rac', 'Xac', 'Rbc', 'Xbc']:
9         System_Data_Lines[col] = System_Data_Lines[col]/1000
10
11    # Process System_Data_Lines to obtain the Impedance Prima
12    System_Data_Lines_Impedances = pd.DataFrame(0.0, index = System_Data_Lines.index, columns = ['FROM', 'TO', 'Zaa', 'Thaa', 'Zbb', 'Thbb', 'Zcc', 'Thcc', 'Zab', 'Thab', 'Zac', 'Thac', 'Zbc', 'Thbc'])
13    System_Data_Lines_Impedances['FROM'] =     System_Data_Lines['FROM']
14    System_Data_Lines_Impedances['TO'] =     System_Data_Lines['TO']
15    for i in System_Data_Lines_Impedances.index:
16        System_Data_Lines_Impedances.iloc[i,2] = math.sqrt(System_Data_Lines.iloc[i,2]**2 + System_Data_Lines.iloc[i,3]**2) # Zaa
17        System_Data_Lines_Impedances.iloc[i,3] = math.atan(System_Data_Lines.iloc[i,3]/System_Data_Lines.iloc[i,2])         # Thaa
18        System_Data_Lines_Impedances.iloc[i,4] = math.sqrt(System_Data_Lines.iloc[i,4]**2 + System_Data_Lines.iloc[i,5]**2) # Zbb
19        System_Data_Lines_Impedances.iloc[i,5] = math.atan(System_Data_Lines.iloc[i,5]/System_Data_Lines.iloc[i,4])         # Thbb
20        System_Data_Lines_Impedances.iloc[i,6] = math.sqrt(System_Data_Lines.iloc[i,6]**2 + System_Data_Lines.iloc[i,7]**2) # Zcc
21        System_Data_Lines_Impedances.iloc[i,7] = math.atan(System_Data_Lines.iloc[i,7]/System_Data_Lines.iloc[i,6])         # Thcc
22        System_Data_Lines_Impedances.iloc[i,8] = math.sqrt(System_Data_Lines.iloc[i,8]**2 + System_Data_Lines.iloc[i,9]**2) # Zab
23        System_Data_Lines_Impedances.iloc[i,9] = math.atan(System_Data_Lines.iloc[i,9]/System_Data_Lines.iloc[i,8])         # Thab
24        System_Data_Lines_Impedances.iloc[i,10] = math.sqrt(System_Data_Lines.iloc[i,10]**2 + System_Data_Lines.iloc[i,11]**2) # Zac
25        System_Data_Lines_Impedances.iloc[i,11] = math.atan(System_Data_Lines.iloc[i,11]/System_Data_Lines.iloc[i,10])         # Thac
26        System_Data_Lines_Impedances.iloc[i,12] = math.sqrt(System_Data_Lines.iloc[i,12]**2 + System_Data_Lines.iloc[i,13]**2) # Zbc
27        System_Data_Lines_Impedances.iloc[i,13] = math.atan(System_Data_Lines.iloc[i,13]/System_Data_Lines.iloc[i,12])         # Thbc
28
29    System_Data_Lines_Prima = pd.DataFrame(0.0, index = System_Data_Lines.index, columns = ['FROM', 'TO', 'Raa_p', 'Xaa_p', 'Rbb_p', 'Xbb_p', 'Rcc_p', 'Xcc_p', 'Rab_p', 'Xab_p', 'Rac_p', 'Xac_p', 'Rbc_p', 'Xbc_p', 'Rba_p', 'Xba_p', 'Rca_p', 'Xca_p', 'Rcb_p' ,'Xcb_p', 'Imax'])
30    System_Data_Lines_Prima['FROM'] =  System_Data_Lines['FROM']
31    System_Data_Lines_Prima['TO'] =  System_Data_Lines['TO']
```

```python
    System_Data_Lines_Prima['Imax'] =  System_Data_Lines['Imax']
    for i in System_Data_Lines_Prima.index:
        System_Data_Lines_Prima.iloc[i,2] = System_Data_Lines_Impedances.iloc[i,2]*math.cos(System_Data_Lines_Impedances.iloc[i,3] + Tha0 - Tha0) # Raa_p
        System_Data_Lines_Prima.iloc[i,3] = System_Data_Lines_Impedances.iloc[i,2]*math.sin(System_Data_Lines_Impedances.iloc[i,3] + Tha0 - Tha0) # Xaa_p
        System_Data_Lines_Prima.iloc[i,4] = System_Data_Lines_Impedances.iloc[i,4]*math.cos(System_Data_Lines_Impedances.iloc[i,5] + Thb0 - Thb0) # Rbb_p
        System_Data_Lines_Prima.iloc[i,5] = System_Data_Lines_Impedances.iloc[i,4]*math.sin(System_Data_Lines_Impedances.iloc[i,5] + Thb0 - Thb0) # Xbb_p
        System_Data_Lines_Prima.iloc[i,6] = System_Data_Lines_Impedances.iloc[i,6]*math.cos(System_Data_Lines_Impedances.iloc[i,7] + Thc0 - Thc0) # Rcc_p
        System_Data_Lines_Prima.iloc[i,7] = System_Data_Lines_Impedances.iloc[i,6]*math.sin(System_Data_Lines_Impedances.iloc[i,7] + Thc0 - Thc0) # Xcc_p
        System_Data_Lines_Prima.iloc[i,8] = System_Data_Lines_Impedances.iloc[i,8]*math.cos(System_Data_Lines_Impedances.iloc[i,9] + Thb0 - Tha0) # Rab_p
        System_Data_Lines_Prima.iloc[i,9] = System_Data_Lines_Impedances.iloc[i,8]*math.sin(System_Data_Lines_Impedances.iloc[i,9] + Thb0 - Tha0) # Xab_p
        System_Data_Lines_Prima.iloc[i,10] = System_Data_Lines_Impedances.iloc[i,10]*math.cos(System_Data_Lines_Impedances.iloc[i,11] + Thc0 - Tha0) # Rac_p
        System_Data_Lines_Prima.iloc[i,11] = System_Data_Lines_Impedances.iloc[i,10]*math.sin(System_Data_Lines_Impedances.iloc[i,11] + Thc0 - Tha0) # Xac_p
        System_Data_Lines_Prima.iloc[i,12] = System_Data_Lines_Impedances.iloc[i,12]*math.cos(System_Data_Lines_Impedances.iloc[i,13] + Thc0 - Thb0) # Rbc_p
        System_Data_Lines_Prima.iloc[i,13] = System_Data_Lines_Impedances.iloc[i,12]*math.sin(System_Data_Lines_Impedances.iloc[i,13] + Thc0 - Thb0) # Xbc_p
        System_Data_Lines_Prima.iloc[i,14] = System_Data_Lines_Impedances.iloc[i,8]*math.cos(System_Data_Lines_Impedances.iloc[i,9] + Tha0 - Thb0) # Rba_p
        System_Data_Lines_Prima.iloc[i,15] = System_Data_Lines_Impedances.iloc[i,8]*math.sin(System_Data_Lines_Impedances.iloc[i,9] + Tha0 - Thb0) # Xba_p
        System_Data_Lines_Prima.iloc[i,16] = System_Data_Lines_Impedances.iloc[i,10]*math.cos(System_Data_Lines_Impedances.iloc[i,11] + Tha0 - Thc0) # Rca_p
        System_Data_Lines_Prima.iloc[i,17] = System_Data_Lines_Impedances.iloc[i,10]*math.sin(System_Data_Lines_Impedances.iloc[i,11] + Tha0 - Thc0) # Xca_p
        System_Data_Lines_Prima.iloc[i,18] = System_Data_Lines_Impedances.iloc[i,12]*math.cos(System_Data_Lines_Impedances.iloc[i,13] + Thb0 - Thc0) # Rcb_p
        System_Data_Lines_Prima.iloc[i,19] = System_Data_Lines_Impedances.iloc[i,12]*math.sin(System_Data_Lines_Impedances.iloc[i,13] + Thb0 - Thc0) # Xcb_p

    return System_Data_Lines_Prima
```

Listing 2: Python implementation for the `pre_processing_system_data()` function.

3.2. *System Data Function*

Listing 3 shows the implementation of the `three_phase_opf_model()` function. This function takes as inputs the DataFrames `System_Data_Nodes` and `System_Data_Lines_Prima`, and gives as output the Python List `Data_Network`, containing the set of nodes of the distribution network (`N`, i.e. $\mathcal{N}$ in the mathematical formulation), the set of lines (`L`, i.e. $\mathcal{L}$), the active (`PDa, PDb, PDc`, i.e., $P^D_{m,\phi}$) and reactive (`QDa, QDb, QDc`, i.e., $Q^D_{m,\phi}$) power demand of each node, and the *prima* self- and mutual-impedance ($R^{'}_{mn,\phi,\psi}$ and $X^{'}_{mn,\phi,\psi}$) for all lines. `Data_Network` is a Python List containing all these information in the form of Dictionaries (except by `N` which is also a List, as can be seen in line 4). We will see next how to obtain these Dictionaries.

The first Dictionary in `Data_Network` correspond to `Tn`, which it does not have equivalent in the mathematical formulation. We introduce `Tn` in line 5 to represent the type of node (`Tn` comes from "Type of node"). As you can see in second column of the file `Nodes_25_3F.xlsx`, all nodes have `Tn=0` and only node 1 has `Tn=1`. We use this representation to denote that node 1 is the reference node (or swing bus) or the boundary node between the MV and LV distribution network. You will see later how this representation help us to simply our Pyomo model. In line 5, we can see that `Tn` is a function of the nodes of the distribution network (`N[i]`) and once created has the next form: `{1:1, 2:0, 3:0,..., 25:0}`. This Dictionary representation is the way we need to describe input information (and variables) for Pyomo and can be read as: For node `1`, `Tn` has a value of `1`, for node `2`, `Tn` has a value of `0`, up to node 25, `Tn` has a value of `0`. If it is still not clear, let us check another example.

In line 6, the Dictionary for the active power demand of phase A is created (`PDa`). We know from the mathematical model that we need to define an active power demand value for each node, represented as $P^D_{m,\phi=A}$. This way, we need then to create a Dictionary that looks like this: {(Node 1, Demand Phase A Node 1), (Node 2, Demand Phase A Node 2),..., (Node 25, Demand Phase A Node 25)}. This is indeed how Dictionary (`PDa`) in line 6 looks like: `{1:0.0, 2:0.0, 3:35.0, 4:50.0,..., 25:60.0}`. Go and compare this node-demand Dictionary with the one listed in the file `Nodes_25_3F.xlsx` for Phase A.

The remain of lines, from line 7 to line 31, creates the remain Dictionaries, check each line in the code presented in Listing 3 and compare it with the parameters in the mathematical formulation in (31)–(37).

```
1
2  def system_data_for_pyomo(System_Data_Nodes, System_Data_Lines_Prima):
3      # Network Data
4      N = [System_Data_Nodes.loc[i,'N'] for i in System_Data_Nodes.index]
5      Tn = {N[i]: System_Data_Nodes.loc[i,'Tn'] for i in System_Data_Nodes.index}
6      PDa = {N[i]: System_Data_Nodes.loc[i,'PDa'] for i in System_Data_Nodes.index}
7      QDa = {N[i]: System_Data_Nodes.loc[i,'QDa'] for i in System_Data_Nodes.index}
8      PDb = {N[i]: System_Data_Nodes.loc[i,'PDb'] for i in System_Data_Nodes.index}
9      QDb = {N[i]: System_Data_Nodes.loc[i,'QDb'] for i in System_Data_Nodes.index}
10     PDc = {N[i]: System_Data_Nodes.loc[i,'PDc'] for i in System_Data_Nodes.index}
11     QDc = {N[i]: System_Data_Nodes.loc[i,'QDc'] for i in System_Data_Nodes.index}
12     L = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']) for i
       in System_Data_Lines_Prima.index}
13     Raa_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']):
       System_Data_Lines_Prima.loc[i,'Raa_p'] for i in System_Data_Lines_Prima.index}
14     Xaa_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']):
       System_Data_Lines_Prima.loc[i,'Xaa_p'] for i in System_Data_Lines_Prima.index}
15     Rbb_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']):
       System_Data_Lines_Prima.loc[i,'Rbb_p'] for i in System_Data_Lines_Prima.index}
16     Xbb_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']):
       System_Data_Lines_Prima.loc[i,'Xbb_p'] for i in System_Data_Lines_Prima.index}
17     Rcc_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']):
       System_Data_Lines_Prima.loc[i,'Rcc_p'] for i in System_Data_Lines_Prima.index}
18     Xcc_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']):
       System_Data_Lines_Prima.loc[i,'Xcc_p'] for i in System_Data_Lines_Prima.index}
19     Rab_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']):
       System_Data_Lines_Prima.loc[i,'Rab_p'] for i in System_Data_Lines_Prima.index}
20     Xab_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']):
       System_Data_Lines_Prima.loc[i,'Xab_p'] for i in System_Data_Lines_Prima.index}
```

```python
21     Rac_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']): System_Data_Lines_Prima.loc[i,'Rac_p'] for i in System_Data_Lines_Prima.index}
22     Xac_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']): System_Data_Lines_Prima.loc[i,'Xac_p'] for i in System_Data_Lines_Prima.index}
23     Rbc_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']): System_Data_Lines_Prima.loc[i,'Rbc_p'] for i in System_Data_Lines_Prima.index}
24     Xbc_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']): System_Data_Lines_Prima.loc[i,'Xbc_p'] for i in System_Data_Lines_Prima.index}
25     Rba_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']): System_Data_Lines_Prima.loc[i,'Rba_p'] for i in System_Data_Lines_Prima.index}
26     Xba_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']): System_Data_Lines_Prima.loc[i,'Xba_p'] for i in System_Data_Lines_Prima.index}
27     Rca_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']): System_Data_Lines_Prima.loc[i,'Rca_p'] for i in System_Data_Lines_Prima.index}
28     Xca_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']): System_Data_Lines_Prima.loc[i,'Xca_p'] for i in System_Data_Lines_Prima.index}
29     Rcb_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']): System_Data_Lines_Prima.loc[i,'Rcb_p'] for i in System_Data_Lines_Prima.index}
30     Xcb_p = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']): System_Data_Lines_Prima.loc[i,'Xcb_p'] for i in System_Data_Lines_Prima.index}
31     Imax = {(System_Data_Lines_Prima.loc[i,'FROM'],System_Data_Lines_Prima.loc[i,'TO']): System_Data_Lines_Prima.loc[i,'Imax'] for i in System_Data_Lines_Prima.index}
32 
33     Data_Network = [N, L, Tn, PDa, PDb, PDc, QDa, QDb, QDc, Raa_p, Xaa_p, Rbb_p, Xbb_p, Rcc_p, Xcc_p, Rab_p, Xab_p, Rac_p, Xac_p, Rbc_p, Xbc_p, Rba_p, Xba_p, Rca_p, Xca_p, Rcb_p, Xcb_p, Imax]
34 
35     return Data_Network
```

Listing 3: Python implementation of the `system_data()_for_pyomo()` function.

3.3. OPF Formulation Function in Pyomo

Listing 4 shows the implementation of the function `three_phase_opf_model()`, which contains the OPF formulation in Pyomo language. This function takes as inputs the nominal voltage `V0` and the DataFrame `Data_Network`, and it returns as output our model, saved in `model` as shown in line 332. In the first part of Listing 4, from line 6 to 38, we extracted the model's parameters from `Data_Network` and saved them into Pyomo parameters. To do this, it is important to match the parameter definition with the same column location as defined in line 33 in Listing 3. In line 41, we define the type of Pyomo model that we will build. In this case, we will build a `ConcreteModel()`. More information about other types of models can be found in Pyomo documentation.

In lines 43 and 44, we define the two sets for our mathematical formulation. These corresponds to the set of nodes (`N`) and the set of lines (`L`). Notice the Pyomo structure to define such sets: `Name_Model.Name_Set = Set(initialize=Parameter_with_Set_Data)`. In our case, the name of our model is just `model` (line 41), the name set is `N`, and the Pyomo parameter containing the set information is also named `N` (see line 7). We follow a similar Pyomo structure to define the parameters `Tn` and `V0` in lines 47 and 48, with the difference that first, we indicate that `Tn` is a parameter that takes value for each element of the set `N`; and second, we set `mutable` as `True` to indicate that these parameters must be update it if the function `three_phase_opf_model()` is called more than once time[8]. A similar procedure is done to define the remain parameters in line 50 to line 77.

The second part of this function, from line 83 to 192, the variables of our model are defined. To define variable `Va`, which represent the voltage magnitude of the phase A for all the nodes in the distribution network, observe in line 83 that we first define a rule (called `Va_init_rule`) that will fix the value of `Va` to `V0` by setting `fix` as `True`, if `Tn == 1` (see line 85 and 86); otherwise, `fix` is set to `False`, indicating that `Va` is not fixed to be `V0`. In this case, as `Va` is not fixed, Pyomo understand line 88 as if we were given an initial value to `Va` as `V0`. In power systems, this is a common practice called *warm start*, which helps improving the convergence of power flows algorithms. Finally, in line 91, we define `Va` as a variable that is defined for each element of the set `N`, and that must be initialize following the rule `Va_init_rule`. The definition of the remain voltage variables, `Vb` and `Vc` follows a similar procedure as shown in lines 93 and 103, respectively.

To define the active power of the substation, defined in our mathematical formulation as $P^S_{m,\phi}$ and in the Pyomo model as `PSa, PSb, PSc`; we follow a similar structure as used to define `Va`, but in this case we must fix `PSa` to `0` if `Tn == 0`[9]. This is done in the rule defined in lines 118 to 125. The remain code to the same procedure to define `PSb` and `PSc`.

In the final lines of this second part, other variables are defined, such as the active power flowing in lines (`Pa, Pb, Pc`), reactive power flowing in lines (`Qa, Qb, Qc`), active power losses in lines (`Plss_a, Plss_b, Plss_c`)[10] and reactive power in lines (`Qlss_a, Qlss_b, Qlss_c`), as shown in lines 179 to 192, all initialized as `0`.

In the third, and last, part of this function, we define the objective function in lines 196 to 198, and all the model constraints in lines 205 to 330. The Pyomo structure to define the objective function and the constraints follow a similar structure that the one used to define the variables and the sets, i.e., define a rule and then use it to define the constraint (or objective function). Check, for instance, how the active power balance constraint in expression (34) is defined in line 268. I encourage you to compare the way the mathematical expression in (34) is written and how similar it is coded in line 268. This is one of the main advantages of algebraic modeling languages such as Pyomo (and GAMS, AMPL, etc), the similar way in which optimization problems are coded with regards to their mathematical formulation.

```
1
2  from pyomo.environ import *
3
4  def three_phase_opf_model(V0, Data_Network):
5
6      #%% Data Processing
7      N = Data_Network[0]
8      L = Data_Network[1]
9      Tb = Data_Network[2]
10
11     # Power
12     PDa = Data_Network[3]
13     PDb = Data_Network[4]
```

[8]Think as an example that you want to solve the same OPF formulation in a loop with different demand values. By setting `mutable` as `True`, the model will update all these parameters with the updated values in the DataFrame `Data_Network`

[9]Remember that we defined that the substation is connected to all nodes in our model. This active power that the substation provides must be 0 if the node is not the substation node, identified with `Tn == 1`.

[10]In the mathematical formulation the active power losses were expressed as $P^L_{mn,\phi}$. In Pyomo, we have written this as (`Plss_a, Plss_b, Plss_c`) to help us to follow better the code.

```python
    PDc = Data_Network[5]
    QDa = Data_Network[6]
    QDb = Data_Network[7]
    QDc = Data_Network[8]

    # Impedances Prima
    Raa_p = Data_Network[9]
    Xaa_p = Data_Network[10]
    Rbb_p = Data_Network[11]
    Xbb_p = Data_Network[12]
    Rcc_p = Data_Network[13]
    Xcc_p = Data_Network[14]
    Rab_p = Data_Network[15]
    Xab_p = Data_Network[16]
    Rac_p = Data_Network[17]
    Xac_p = Data_Network[18]
    Rbc_p = Data_Network[19]
    Xbc_p = Data_Network[20]
    Rba_p = Data_Network[21]
    Xba_p = Data_Network[22]
    Rca_p = Data_Network[23]
    Xca_p = Data_Network[24]
    Rcb_p = Data_Network[25]
    Xcb_p = Data_Network[26]
    Imax = Data_Network[27]

    #%% Type of Model
    model = ConcreteModel()
    #%% Define Sets
    model.N = Set(initialize=N)    # Set of Nodes
    model.L = Set(initialize=L)    # Set of lines

    # Define Parameters
    model.Tb = Param(model.N, initialize=Tb, mutable=True)    # Type of node. SS node == 1
    model.V0 = Param(initialize=V0, mutable=True)

    # Network Parameters
    model.Raa_p = Param(model.L, initialize=Raa_p, mutable=True)
    model.Xaa_p = Param(model.L, initialize=Xaa_p, mutable=True)
    model.Rbb_p = Param(model.L, initialize=Rbb_p, mutable=True)
    model.Xbb_p = Param(model.L, initialize=Xbb_p, mutable=True)
    model.Rcc_p = Param(model.L, initialize=Rcc_p, mutable=True)
    model.Xcc_p = Param(model.L, initialize=Xcc_p, mutable=True)
    model.Rab_p = Param(model.L, initialize=Rab_p, mutable=True)
    model.Xab_p = Param(model.L, initialize=Xab_p, mutable=True)
    model.Rac_p = Param(model.L, initialize=Rac_p, mutable=True)
    model.Xac_p = Param(model.L, initialize=Xac_p, mutable=True)
    model.Rbc_p = Param(model.L, initialize=Rbc_p, mutable=True)
    model.Xbc_p = Param(model.L, initialize=Xbc_p, mutable=True)
    model.Rba_p = Param(model.L, initialize=Rba_p, mutable=True)
    model.Xba_p = Param(model.L, initialize=Xba_p, mutable=True)
    model.Rca_p = Param(model.L, initialize=Rca_p, mutable=True)
```

```python
    model.Xca_p = Param(model.L, initialize=Xca_p, mutable=True)
    model.Rcb_p = Param(model.L, initialize=Rcb_p, mutable=True)
    model.Xcb_p = Param(model.L, initialize=Xcb_p, mutable=True)
    model.Imax = Param(model.L, initialize=Imax, mutable=True)

    # Active and Reactive Demand Power
    model.PDa = Param(model.N, initialize=PDa, mutable=True)
    model.PDb = Param(model.N, initialize=PDb, mutable=True)
    model.PDc = Param(model.N, initialize=PDc, mutable=True)
    model.QDa = Param(model.N, initialize=QDa, mutable=True)
    model.QDb = Param(model.N, initialize=QDb, mutable=True)
    model.QDc = Param(model.N, initialize=QDc, mutable=True)

    # Define Variables
    # -------------------------------------------
    # Voltages
    # -------------------------------------------
    def Va_init_rule(model, i):
        if model.Tb[i] == 1:
            temp = model.V0
            model.Va[i].fixed = True
        else:
            temp = model.V0
            model.Va[i].fixed = False
        return temp
    model.Va = Var(model.N, initialize = Va_init_rule)  # Voltage Phase A

    def Vb_init_rule(model, i):
        if model.Tb[i] == 1:
            temp = model.V0
            model.Vb[i].fixed = True
        else:
            temp = model.V0
            model.Vb[i].fixed = False
        return temp
    model.Vb = Var(model.N, initialize = Vb_init_rule)  # Voltage Phase B

    def Vc_init_rule(model, i):
        if model.Tb[i] == 1:
            temp = model.V0
            model.Vc[i].fixed = True
        else:
            temp = model.V0
            model.Vc[i].fixed = False
        return temp
    model.Vc = Var(model.N, initialize = Vc_init_rule)  # Voltage Phase C

    # -------------------------------------------
    # Power
    # -------------------------------------------

    # Active and Reactive Power of the Substation
```

```python
    def PSa_init_rule(model, i):
        if model.Tb[i] == 0:
            temp = 0
            model.PSa[i].fixed = True
        else:
            temp = 0
            model.PSa[i].fixed = False
        return temp
    model.PSa = Var(model.N, initialize = PSa_init_rule)  # Active Power SS Phase A

    def PSb_init_rule(model, i):
        if model.Tb[i] == 0:
            temp = 0
            model.PSb[i].fixed = True
        else:
            temp = 0
            model.PSb[i].fixed = False
        return temp
    model.PSb = Var(model.N, initialize = PSb_init_rule)  # Active Power SS Phase B

    def PSc_init_rule(model, i):
        if model.Tb[i] == 0:
            temp = 0
            model.PSc[i].fixed = True
        else:
            temp = 0
            model.PSc[i].fixed = False
        return temp
    model.PSc = Var(model.N, initialize = PSc_init_rule)  # Active Power SS Phase C

    def QSa_init_rule(model, i):
        if model.Tb[i] == 0:
            temp = 0
            model.QSa[i].fixed = True
        else:
            temp = 0
            model.QSa[i].fixed = False
        return temp
    model.QSa = Var(model.N, initialize = QSa_init_rule)  # Reactive Power SS Phase A

    def QSb_init_rule(model, i):
        if model.Tb[i] == 0:
            temp = 0
            model.QSb[i].fixed = True
        else:
            temp = 0
            model.QSb[i].fixed = False
        return temp
    model.QSb = Var(model.N, initialize = QSb_init_rule)  # Reactive Power SS Phase A

    def QSc_init_rule(model, i):
        if model.Tb[i] == 0:
```

```
        temp = 0
        model.QSc[i].fixed = True
    else:
        temp = 0
        model.QSc[i].fixed = False
    return temp
model.QSc = Var(model.N, initialize = QSc_init_rule)  # Reactive Power SS Phase A

# Active and Reactive Power of Lines
model.Pa = Var(model.L, initialize=0)
model.Pb = Var(model.L, initialize=0)
model.Pc = Var(model.L, initialize=0)
model.Qa = Var(model.L, initialize=0)
model.Qb = Var(model.L, initialize=0)
model.Qc = Var(model.L, initialize=0)

# Active and Reactive Power Losses of Lines
model.Plss_a = Var(model.L, initialize=0)
model.Plss_b = Var(model.L, initialize=0)
model.Plss_c = Var(model.L, initialize=0)
model.Qlss_a = Var(model.L, initialize=0)
model.Qlss_b = Var(model.L, initialize=0)
model.Qlss_c = Var(model.L, initialize=0)


#%% Define the Objective Fuction
def Total_Active_Losses(model):
    return (sum(model.Plss_a[i,j] + model.Plss_b[i,j] + model.Plss_c[i,j] for i,j in model.L))
model.obj = Objective(rule=Total_Active_Losses)

#%% Define the Operational Constraints

# ---------------------------------------------------------------
# Define Active Power Losses
# --------------------------------------------------------------
def active_power_losses_phase_A_rule(model, i,j):
    return (model.Plss_a[i,j] ==   (1/(model.Va[i]*model.Va[i]))*(model.Raa_p[i,j]*model.Pa[i,j]*model.Pa[i,j] + model.Raa_p[i,j]*model.Qa[i,j]*model.Qa[i,j] +\
                                                                  model.Xaa_p[i,j]*model.Pa[i,j]*model.Qa[i,j] - model.Xaa_p[i,j]*model.Qa[i,j]*model.Pa[i,j]) +\
                                   (1/(model.Va[i]*model.Vb[i]))*(model.Rab_p[i,j]*model.Pb[i,j]*model.Pa[i,j] + model.Rab_p[i,j]*model.Qb[i,j]*model.Qa[i,j] +\
                                                                  model.Xab_p[i,j]*model.Pb[i,j]*model.Qa[i,j] - model.Xab_p[i,j]*model.Qb[i,j]*model.Pa[i,j]) +\
                                   (1/(model.Va[i]*model.Vc[i]))*(model.Rac_p[i,j]*model.Pc[i,j]*model.Pa[i,j] + model.Rac_p[i,j]*model.Qc[i,j]*model.Qa[i,j] +\
                                                                  model.Xac_p[i,j]*model.Pc[i,j]*model.Qa[i,j] - model.Xac_p[i,j]*model.Qc[i,j]*model.Pa[i,j]))
model.active_power_losses_phase_A = Constraint(model.L, rule=active_power_losses_phase_A_rule)

```

```python
    def active_power_losses_phase_B_rule(model, i,j):
        return(model.Plss_b[i,j] == (1/(model.Vb[i]*model.Va[i]))*(model.Rba_p[i,j]*model.Pa[i,j]*model.Pb[i,j] + model.Rba_p[i,j]*model.Qa[i,j]*model.Qb[i,j] +\
                                                              model.Xba_p[i,j]*model.Pa[i,j]*model.Qb[i,j] - model.Xba_p[i,j]*model.Qa[i,j]*model.Pb[i,j]) +\
                                    (1/(model.Vb[i]*model.Vb[i]))*(model.Rbb_p[i,j]*model.Pb[i,j]*model.Pb[i,j] + model.Rbb_p[i,j]*model.Qb[i,j]*model.Qb[i,j] +\
                                                              model.Xbb_p[i,j]*model.Pb[i,j]*model.Qb[i,j] - model.Xbb_p[i,j]*model.Qb[i,j]*model.Pb[i,j]) +\
                                    (1/(model.Vb[i]*model.Vc[i]))*(model.Rbc_p[i,j]*model.Pc[i,j]*model.Pb[i,j] + model.Rbc_p[i,j]*model.Qc[i,j]*model.Qb[i,j] +\
                                                              model.Xbc_p[i,j]*model.Pc[i,j]*model.Qb[i,j] - model.Xbc_p[i,j]*model.Qc[i,j]*model.Pb[i,j]))
    model.active_power_losses_phase_B = Constraint(model.L, rule=active_power_losses_phase_B_rule)

    def active_power_losses_phase_C_rule(model, i,j):
        return(model.Plss_c[i,j] == (1/(model.Vc[i]*model.Va[i]))*(model.Rca_p[i,j]*model.Pa[i,j]*model.Pc[i,j] + model.Rca_p[i,j]*model.Qa[i,j]*model.Qc[i,j] +\
                                                              model.Xca_p[i,j]*model.Pa[i,j]*model.Qc[i,j] - model.Xca_p[i,j]*model.Qa[i,j]*model.Pc[i,j]) +\
                                    (1/(model.Vc[i]*model.Vb[i]))*(model.Rcb_p[i,j]*model.Pb[i,j]*model.Pc[i,j] + model.Rcb_p[i,j]*model.Qb[i,j]*model.Qc[i,j] +\
                                                              model.Xcb_p[i,j]*model.Pb[i,j]*model.Qc[i,j] - model.Xcb_p[i,j]*model.Qb[i,j]*model.Pc[i,j]) +\
                                    (1/(model.Vc[i]*model.Vc[i]))*(model.Rcc_p[i,j]*model.Pc[i,j]*model.Pc[i,j] + model.Rcc_p[i,j]*model.Qc[i,j]*model.Qc[i,j] +\
                                                              model.Xcc_p[i,j]*model.Pc[i,j]*model.Qc[i,j] - model.Xcc_p[i,j]*model.Qc[i,j]*model.Pc[i,j]))
    model.active_power_losses_phase_C = Constraint(model.L, rule=active_power_losses_phase_C_rule)

    # ---------------------------------------------------------------
    # Define Reactive Power Losses
    # ---------------------------------------------------------------

    def reactive_power_losses_phase_A_rule(model, i,j):
        return(model.Qlss_a[i,j] == (1/(model.Va[i]*model.Va[i]))*(-model.Raa_p[i,j]*model.Pa[i,j]*model.Qa[i,j] + model.Raa_p[i,j]*model.Qa[i,j]*model.Pa[i,j] +\
                                                               model.Xaa_p[i,j]*model.Pa[i,j]*model.Pa[i,j] + model.Xaa_p[i,j]*model.Qa[i,j]*model.Qa[i,j]) +\
                                    (1/(model.Va[i]*model.Vb[i]))*(-model.Rab_p[i,j]*model.Pb[i,j]*model.Qa[i,j] + model.Rab_p[i,j]*model.Qb[i,j]*model.Pa[i,j] +\
                                                               model.Xab_p[i,j]*model.Pb[i,j]*model.Pa[i,j] + model.Xab_p[i,j]*model.Qb[i,j]*model.Qa[i,j]) +\
                                    (1/(model.Va[i]*model.Vc[i]))*(-model.Rac_p[i,j]*model.Pc[i,j]*model.Qa[i,j] + model.Rac_p[i,j]*model.Qc[i,j]*model.Pa[i,j] +\
                                                               model.Xac_p[i,j]*model.Pc[i,j]*model.Pa[i,j] + model.Xac_p[i,j]*model.Qc[i,j]*model.Qa[i,j]))
    model.reactive_power_losses_phase_A = Constraint(model.L, rule=reactive_power_losses_phase_A_rule)

```

```
    def reactive_power_losses_phase_B_rule(model, i,j):
        return(model.Qlss_b[i,j] == (1/(model.Vb[i]*model.Va[i]))*(-model.Rba_p[i,j]*model.Pa[i,j]*model.Qb[i,j] + model.Rba_p[i,j]*model.Qa[i,j]*model.Pb[i,j] +\
                                                                model.Xba_p[i,j]*model.Pa[i,j]*model.Pb[i,j] + model.Xba_p[i,j]*model.Qa[i,j]*model.Qb[i,j]) +\
                                    (1/(model.Vb[i]*model.Vb[i]))*(-model.Rbb_p[i,j]*model.Pb[i,j]*model.Qb[i,j] + model.Rbb_p[i,j]*model.Qb[i,j]*model.Pb[i,j] +\
                                                                model.Xbb_p[i,j]*model.Pb[i,j]*model.Pb[i,j] + model.Xbb_p[i,j]*model.Qb[i,j]*model.Qb[i,j]) +\
                                    (1/(model.Vb[i]*model.Vc[i]))*(-model.Rbc_p[i,j]*model.Pc[i,j]*model.Qb[i,j] + model.Rbc_p[i,j]*model.Qc[i,j]*model.Pb[i,j] +\
                                                                model.Xbc_p[i,j]*model.Pc[i,j]*model.Pb[i,j] + model.Xbc_p[i,j]*model.Qc[i,j]*model.Qb[i,j]))
    model.reactive_power_losses_phase_B = Constraint(model.L, rule=reactive_power_losses_phase_B_rule)

    def reactive_power_losses_phase_C_rule(model, i,j):
        return(model.Qlss_c[i,j] == (1/(model.Vc[i]*model.Va[i]))*(-model.Rca_p[i,j]*model.Pa[i,j]*model.Qc[i,j] + model.Rca_p[i,j]*model.Qa[i,j]*model.Pc[i,j] +\
                                                                model.Xca_p[i,j]*model.Pa[i,j]*model.Pc[i,j] + model.Xca_p[i,j]*model.Qa[i,j]*model.Qc[i,j]) +\
                                    (1/(model.Vc[i]*model.Vb[i]))*(-model.Rcb_p[i,j]*model.Pb[i,j]*model.Qc[i,j] + model.Rcb_p[i,j]*model.Qb[i,j]*model.Pc[i,j] +\
                                                                model.Xcb_p[i,j]*model.Pb[i,j]*model.Pc[i,j] + model.Xcb_p[i,j]*model.Qb[i,j]*model.Qc[i,j]) +\
                                    (1/(model.Vc[i]*model.Vc[i]))*(-model.Rcc_p[i,j]*model.Pc[i,j]*model.Qc[i,j] + model.Rcc_p[i,j]*model.Qc[i,j]*model.Pc[i,j] +\
                                                                model.Xcc_p[i,j]*model.Pc[i,j]*model.Pc[i,j] + model.Xcc_p[i,j]*model.Qc[i,j]*model.Qc[i,j]))
    model.reactive_power_losses_phase_C = Constraint(model.L, rule=reactive_power_losses_phase_C_rule)

    # --------------------------------------------------------------
    # Active Power Balance
    # --------------------------------------------------------------

    def active_power_balance_phase_A_rule(model, k):
        return (sum(model.Pa[j,i] for j,i in model.L if i == k) - sum(model.Pa[i,j] + model.Plss_a[i,j] for i,j in model.L if i == k) + model.PSa[k] == model.PDa[k])
    model.active_power_balance_phase_A = Constraint(model.N, rule=active_power_balance_phase_A_rule)

    def active_power_balance_phase_B_rule(model, k):
        return (sum(model.Pb[j,i] for j,i in model.L if i == k) - sum(model.Pb[i,j] + model.Plss_b[i,j] for i,j in model.L if i == k) + model.PSb[k] == model.PDb[k])
    model.active_power_balance_phase_B = Constraint(model.N, rule=active_power_balance_phase_B_rule)

    def active_power_balance_phase_C_rule(model, k):
        return (sum(model.Pc[j,i] for j,i in model.L if i == k) - sum(model.Pc[i,j] + model.Plss_c[i,j] for i,j in model.L if i == k) + model.PSc[k] == model.PDc[k])
    model.active_power_balance_phase_C = Constraint(model.N, rule=
```

```
active_power_balance_phase_C_rule)

    # ---------------------------------------------------------------
    # Reactive Power Balance
    # ---------------------------------------------------------------

    def reactive_power_balance_phase_A_rule(model, k):
        return (sum(model.Qa[j,i] for j,i in model.L if i == k) - sum(model.Qa[i,j] + model.Qlss_a[i,j] for i,j in model.L if i == k) + model.QSa[k] == model.QDa[k])
    model.reactive_power_balance_phase_A = Constraint(model.N, rule=reactive_power_balance_phase_A_rule)

    def reactive_power_balance_phase_B_rule(model, k):
        return (sum(model.Qb[j,i] for j,i in model.L if i == k) - sum(model.Qb[i,j] + model.Qlss_b[i,j] for i,j in model.L if i == k) + model.QSb[k] == model.QDb[k])
    model.reactive_power_balance_phase_B = Constraint(model.N, rule=reactive_power_balance_phase_B_rule)

    def reactive_power_balance_phase_C_rule(model, k):
        return (sum(model.Qc[j,i] for j,i in model.L if i == k) - sum(model.Qc[i,j] + model.Qlss_c[i,j] for i,j in model.L if i == k) + model.QSc[k] == model.QDc[k])
    model.reactive_power_balance_phase_C = Constraint(model.N, rule=reactive_power_balance_phase_C_rule)

    # ---------------------------------------------------------------
    # Voltage Drop in Lines
    # ---------------------------------------------------------------

    def voltage_drop_phase_A_rule(model, i,j):
        return(model.Va[i]**2 - model.Va[j]**2 == 2*(model.Raa_p[i,j]*model.Pa[i,j] + model.Xaa_p[i,j]*model.Qa[i,j]) + 2*(model.Rab_p[i,j]*model.Pb[i,j] + model.Xab_p[i,j]*model.Qb[i,j]) +\
                                                2*(model.Rac_p[i,j]*model.Pc[i,j] + model.Xac_p[i,j]*model.Qc[i,j]) -\
                                              (1/(model.Va[i]**2))*((model.Raa_p[i,j]*model.Pa[i,j] + model.Xaa_p[i,j]*model.Qa[i,j] +\
                                                                      model.Rab_p[i,j]*model.Pb[i,j] + model.Xab_p[i,j]*model.Qb[i,j] +\
                                                                      model.Rac_p[i,j]*model.Pc[i,j] + model.Xac_p[i,j]*model.Qc[i,j])**2) -\
                                              (1/(model.Va[i]**2))*((model.Raa_p[i,j]*model.Qa[i,j] - model.Xaa_p[i,j]*model.Pa[i,j] +\
                                                                      model.Rab_p[i,j]*model.Qb[i,j] - model.Xab_p[i,j]*model.Pb[i,j] +\
                                                                      model.Rac_p[i,j]*model.Qc[i,j] - model.Xac_p[i,j]*model.Pc[i,j])**2))
    model.voltage_drop_phase_A = Constraint(model.L, rule=voltage_drop_phase_A_rule)

    def voltage_drop_phase_B_rule(model, i,j):
        return(model.Vb[i]**2 - model.Vb[j]**2 == 2*(model.Rab_p[i,j]*model.Pa[i,j] + model.Xab_p[i,j]*model.Qa[i,j]) + 2*(model.Rbb_p[i,j]*model.Pb[i,j] + model.Xbb_p[i,j]*model.Qb[i,j])   +\
```

```python
                                                    2*(model.Rbc_p[i,j]*model.Pc[i,j] + model.Xbc_p[i,j]*model.Qc[i,j]) -\
                                                  (1/(model.Vb[i]**2))*((model.Rba_p[i,j]*model.Pa[i,j] + model.Xba_p[i,j]*model.Qa[i,j] +\
                                                                         model.Rbb_p[i,j]*model.Pb[i,j] + model.Xbb_p[i,j]*model.Qb[i,j] +\
                                                                         model.Rbc_p[i,j]*model.Pc[i,j] + model.Xbc_p[i,j]*model.Qc[i,j])**2) -\
                                                  (1/(model.Vb[i]**2))*((model.Rba_p[i,j]*model.Qa[i,j] - model.Xba_p[i,j]*model.Pa[i,j] +\
                                                                         model.Rbb_p[i,j]*model.Qb[i,j] - model.Xbb_p[i,j]*model.Pb[i,j] +\
                                                                         model.Rbc_p[i,j]*model.Qc[i,j] - model.Xbc_p[i,j]*model.Pc[i,j])**2))
    model.voltage_drop_phase_B = Constraint(model.L, rule=voltage_drop_phase_B_rule)

    def voltage_drop_phase_C_rule(model, i,j):
        return(model.Vc[i]**2 - model.Vc[j]**2 == 2*(model.Rac_p[i,j]*model.Pa[i,j] + model.Xac_p[i,j]*model.Qa[i,j]) + 2*(model.Rbc_p[i,j]*model.Pa[i,j] + model.Xbc_p[i,j]*model.Qa[i,j]) +\
                                                    2*(model.Rcc_p[i,j]*model.Pc[i,j] + model.Xcc_p[i,j]*model.Qc[i,j]) -\
                                                  (1/(model.Vc[i]**2))*((model.Rca_p[i,j]*model.Pa[i,j] + model.Xca_p[i,j]*model.Qa[i,j] +\
                                                                         model.Rcb_p[i,j]*model.Pb[i,j] + model.Xcb_p[i,j]*model.Qb[i,j] +\
                                                                         model.Rcc_p[i,j]*model.Pc[i,j] + model.Xcc_p[i,j]*model.Qc[i,j])**2) -\
                                                  (1/(model.Vc[i]**2))*((model.Rca_p[i,j]*model.Qa[i,j] - model.Xca_p[i,j]*model.Pa[i,j] +\
                                                                         model.Rcb_p[i,j]*model.Qb[i,j] - model.Xcb_p[i,j]*model.Pb[i,j] +\
                                                                         model.Rcc_p[i,j]*model.Qc[i,j] - model.Xcc_p[i,j]*model.Pc[i,j])**2))
    model.voltage_drop_phase_C = Constraint(model.L, rule=voltage_drop_phase_C_rule)

    return model
```

Listing 4: Python implementation of the `three_phase_opf_model()` function.

3.4. Print Results Function

Listing 5 shows the implementation of the `print_results()` function. This function takes as input the Pyomo model (`model`) and the nominal voltage magnitude (`V0`) and print out useful information that can be used to assess the quality of the solution provided by IPOPT. You can customize these types of print functions to be able to check any result that you might want to validate. In line 6, the `print_results()` function print the total active power demand per phase (`PDa`, `PDb`, `PDc`) and the total phase demand (`PDa + PDb + PDc`). Notice in line 6 that in order to read a value from a model in Pyomo the command `value` must be used. As an example, to read the active demand of phase A in node 1, you must call it as `value.PDa[1]`, which must return `0.0`. In line 7, `print_results()` function print the active power supplied by the substation per phase (`PSa`, `PSb`, `PSc`) and total (`PSa + PSb + PSc`), while in line 8, the active power losses are printed by phase

(`Plssa`, `Plssb`, `Plssc`) and total (`Plssa + Plssb + Plssc`). A similar implementation is done to print the same results but for the reactive power in lines 10 to 12. Finally, in line 17, the `for` loop prints the voltage magnitude for all the nodes of the distribution network in p.u.

```python
1
2 def print_results(model, V0):
3     print('---------------------------------------------------------------------')
4     print('\t\tPhA\t\tPhB\t\tPhC\t\tTotal')
5     print('---------------------------------------------------------------------')
6     print("PD\t\t%0.2f\t\t%0.2f\t\t%0.2f\t\t%0.2f"%(sum(model.PDa[i].value for i in model.N),sum(model.PDb[i].value for i in model.N),sum(model.PDc[i].value for i in model.N),sum(model.PDa[i].value + model.PDb[i].value+ model.PDc[i].value  for i in model.N)))
7     print("PS\t\t%0.2f\t\t%0.2f\t\t%0.2f\t\t%0.2f"%(sum(model.PSa[i].value for i in model.N),sum(model.PSb[i].value for i in model.N),sum(model.PSc[i].value for i in model.N),sum(model.PSa[i].value + model.PSb[i].value+ model.PSc[i].value  for i in model.N)))
8     print("Plss\t\t%0.2f\t\t%0.2f\t\t%0.2f\t\t%0.2f"%(sum(model.Plss_a[i,j].value for i,j in model.L),sum(model.Plss_b[i,j].value for i,j in model.L),sum(model.Plss_c[i,j].value for i,j in model.L),sum(model.Plss_a[i,j].value + model.Plss_b[i,j].value + model.Plss_c[i,j].value  for i,j in model.L)))
9     print('---------------------------------------------------------------------')
10    print("QD\t\t%0.2f\t\t%0.2f\t\t%0.2f\t\t%0.2f"%(sum(model.QDa[i].value for i in model.N),sum(model.QDb[i].value for i in model.N),sum(model.QDc[i].value for i in model.N),sum(model.QDa[i].value + model.QDb[i].value+ model.QDc[i].value  for i in model.N)))
11    print("QS\t\t%0.2f\t\t%0.2f\t\t%0.2f\t\t%0.2f"%(sum(model.QSa[i].value for i in model.N),sum(model.QSb[i].value for i in model.N),sum(model.QSc[i].value for i in model.N),sum(model.QSa[i].value + model.QSb[i].value+ model.QSc[i].value  for i in model.N)))
12    print("Qlss\t\t%0.2f\t\t%0.2f\t\t%0.2f\t\t%0.2f"%(sum(model.Qlss_a[i,j].value for i,j in model.L),sum(model.Qlss_b[i,j].value for i,j in model.L),sum(model.Qlss_c[i,j].value for i,j in model.L),sum(model.Qlss_a[i,j].value + model.Qlss_b[i,j].value + model.Qlss_c[i,j].value  for i,j in model.L)))
13    print('---------------------------------------------------------------------')
14    print('---------------------------------------------------------------------')
15    print('i\tVa\t\tVb\t\tVc')
16    print('---------------------------------------------------------------------')
17    for i in model.N:
18        print("%i\t%0.5f\t\t%0.5f\t\t%0.5f"%(i,model.Va[i].value/V0,model.Vb[i].value/V0,model.Vc[i].value/V0))
```

Listing 5: Python implementation of the `print_results()` function.

3.5. Executing the Main Python File

Before executing the main file, be sure that all functions (in the Python files) are located in the same folder that the main function described in the Listing 1. The output results after executing the main file are shown in Listing 6. Some important information can be extracted from the output information give by the solver IPOPT during the optimization process. For instance, in line 22, we can observe that the EXIT flag of IPOPT is indicating that the optimal solution to our model was found. We can also observe information such as the total number of objective function evaluations (during the optimization process), the number of equality, inequality constraints, etc., in lines 12 to 20. The output of the `print_results()` can be seen from line 23 to line 61. For instance, in line 37, we can observe that the voltage magnitude of node 1 is equal to 1 p.u., as we were expecting. We can also observe that as there is not any distributed generation unit within the distribution network, the voltage magnitude decreases as we move towards the end of the feeders. This type of analysis are important as it will help you to conclude that the solution provided by IPOPT matches with what it was expected.

As a final piece of advice, do not trust 100% the output of IPOPT solver only because the EXIT flag indicates `Optimal Solution Found`. A solution is optimal if it meets certain error tolerance conditions when the quality and inequality constraints are checked. This is important as the way you design a constraint might not be the way it works (or understood by the model), and this produce results that you might not be expecting. Thus, always check your results and compared them with your model: are all my constraints met? All results follow my understanding of the problem?, etc.

```

Number of Iterations....: 3

                                   (scaled)                 (unscaled)
Objective...............:  1.4543161730643087e+002   1.4543161730643087e+002
Dual infeasibility......:  2.7543095528026319e-014   2.7543095528026319e-014
Constraint violation....:  1.4973338124946167e-009   1.4973338124946167e-009
Complementarity.........:  0.0000000000000000e+000   0.0000000000000000e+000
Overall NLP error.......:  1.4973338124946167e-009   1.4973338124946167e-009


Number of objective function evaluations             = 4
Number of objective gradient evaluations             = 4
Number of equality constraint evaluations            = 4
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 4
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 3
Total CPU secs in IPOPT (w/o function evaluations)   =      0.006
Total CPU secs in NLP function evaluations           =      0.001

EXIT: Optimal Solution Found.
--------------------------------------------------------------------
        PhA       PhB       PhC       Total
--------------------------------------------------------------------
PD        1073.30   1083.30   1083.30   3239.90
PS        1123.94   1137.37   1124.02   3385.33
Plss    50.64   54.07   40.72   145.43
--------------------------------------------------------------------
QD        792.00    801.00    800.00    2393.00
QS        847.41    852.59    854.08    2554.08
Qlss    55.41   51.59   54.08   161.08
--------------------------------------------------------------------
--------------------------------------------------------------------
i     Va        Vb        Vc
--------------------------------------------------------------------
1     1.00000   1.00000   1.00000
2     0.97125   0.96293   0.96999
3     0.96435   0.95415   0.96294
4     0.96093   0.94994   0.95979
5     0.95987   0.94871   0.95872
```

```
6       0.95631    0.94505    0.95439
7       0.94346    0.92965    0.94072
8       0.95420    0.94257    0.95224
9       0.93750    0.92235    0.93455
10        0.93315     0.91666     0.92968
11        0.93107    0.91401    0.92749
12        0.93008    0.91269    0.92644
13        0.93037    0.91304    0.92667
14        0.93755    0.92268    0.93420
15        0.93540    0.92014    0.93200
16        0.94239    0.92839    0.93963
17        0.93635    0.92154    0.93285
18        0.95850    0.94724    0.95674
19        0.95361    0.94238    0.95215
20        0.95601    0.94447    0.95417
21        0.95499    0.94320    0.95294
22        0.95305    0.94064    0.95089
23        0.95764    0.94641    0.95686
24        0.95562    0.94417    0.95495
25        0.95321    0.94216    0.95303
```

Listing 6: Python output after executing the main function file in Listing 1.

4. Multi-Period Three-Phase OPF Formulation

The OPF formulation previously presented in Sec. 2 can be easily extended to a multi-period OPF formulation, as shown next:

$$\min \left\{ \sum_{mn\in\mathcal{L},\psi\in\mathcal{F},t\in\mathcal{T}} P^L_{mn,\psi,t} \right\} \tag{38}$$

Subject to the next set of constraints

$$\begin{aligned} P^L_{mn,\phi,t} = \sum_{\psi\in\mathcal{F}} \frac{1}{|V_{m,\psi,t}||V_{m,\phi,t}|} \Big(& R'_{mn,\phi,\psi} P_{mn,\phi,t} P_{mn,\psi,t} + R'_{mn,\phi,\psi} Q_{mn,\phi,t} Q_{mn,\psi,t} \\ & + X'_{mn,\phi,\psi} P_{mn,\phi,t} Q_{mn,\psi,t} - X'_{mn,\phi,\psi} Q_{mn,\phi,t} P_{mn,\psi,t} \Big), \forall mn \in \mathcal{L}, \forall \phi \in \mathcal{F}, \forall t \in \mathcal{T}; \end{aligned} \tag{39}$$

$$\begin{aligned} Q^L_{mn,\phi,t} = \sum_{\psi\in\mathcal{F}} \frac{1}{|V_{m,\psi,t}||V_{m,\phi,t}|} \Big(& -R'_{mn,\phi,\psi} P_{mn,\phi,t} Q_{mn,\psi,t} + R'_{mn,\phi,\psi} Q_{mn,\phi,t} P_{mn,\psi,t} \\ & + X'_{mn,\phi,\psi} P_{mn,\phi,t} P_{mn,\psi,t} + X'_{mn,\phi,\psi} Q_{mn,\phi,t} Q_{mn,\psi,t} \Big), \forall mn \in \mathcal{L}, \forall \phi \in \mathcal{F}, \forall t \in \mathcal{T}; \end{aligned} \tag{40}$$

$$\sum_{km\in\mathcal{L}} P_{km,\phi,t} - \sum_{mn\in\mathcal{L}} \left(P_{mn,\phi,t} + P^L_{mn,\phi,t} \right) + P^S_{m,\phi,t} = P^D_{m,\phi,t}, \forall m \in \mathcal{N}, \forall \phi \in \mathcal{F}, \forall t \in \mathcal{T}; \tag{41}$$

$$\sum_{km\in\mathcal{L}} Q_{km,\phi,t} - \sum_{mn\in\mathcal{L}} \left(Q_{mn,\phi,t} + Q^L_{mn,\phi,t} \right) + Q^S_{m,\phi,t} = Q^D_{m,\phi,t}, \forall m \in \mathcal{N}, \forall \phi \in \mathcal{F}, \forall t \in \mathcal{T}; \tag{42}$$

$$|V_{m,\phi,t}|^2 - |V_{n,\phi,t}|^2 = 2\sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}P_{mn,\psi,t} + X^{'}_{mn,\phi,\psi}Q_{mn,\psi,t}) +$$

$$\frac{1}{|V_{m,\phi,t}|^2}\left(\sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}P_{mn,\psi,t} + X^{'}_{mn,\phi,\psi}Q_{mn,\psi,t})\right)^2 + \frac{1}{|V_{m,\phi,t}|^2}\left(\sum_{\psi\in\mathcal{F}}(R^{'}_{mn,\phi,\psi}Q_{mn,\psi,t} - X^{'}_{mn,\phi,\psi}P_{mn,\psi,t})\right)^2,$$

$$\forall mn \in \mathcal{L}, \forall \phi \in \mathcal{F}, \forall t \in \mathcal{T}; \quad (43)$$

$$(P^2_{mn,\phi,t} + Q^2_{mn,\phi,t})/|V_{m,\phi,t}|^2 \leq \overline{I}^2_{mn}, \forall mn \in \mathcal{L}, \forall \phi \in \mathcal{F}. \quad (44)$$

Besides these expressions, in order to solve this OPF model, we need to additionally set $P^S_{m,\phi,t} = 0$, $\forall m \in \mathcal{N} \neq \{1\}, \forall t \in \mathcal{T}$, and $|V_{m,\phi,t}| = V_0$, $\forall m \in \mathcal{N} = \{1\}, \forall t \in \mathcal{T}$ for $\phi = \{A\}$, $|V_{m,\phi,t}| = V_0$, $\forall m \in \mathcal{N} = \{1\}, \forall t \in \mathcal{T}$ for $\phi = \{B\}$, and $|V_{m,\phi,t}| = V_0$, $\forall m \in \mathcal{N} = \{1\}, \forall t \in \mathcal{T}$ for $\phi = \{C\}$, where V_0 is the nominal voltage magnitude value (i.e., 1 p.u.).

5. Exercises

1. Based on the presented Pyomo/Python implementation provided in this chapter, implement the multi-period OPF formulation presented in Sec. 4 considering the same load for all nodes for three different load levels i.e., for $t = 1$ the load is 100% (as in Sec. 3), for $t = 2$ the load is 80% and for $t = 3$ the load is 75%.

References

[1] A. Keane, L. F. Ochoa, C. L. T. Borges, G. W. Ault, A. D. Alarcon-Rodriguez, R. A. F. Currie, F. Pilo, C. Dent, G. P. Harrison, State-of-the-art techniques and challenges ahead for distributed generation planning and optimization, IEEE Trans. Power Systems 28 (2013) 1493–1502. doi:10.1109/TPWRS.2012.2214406.

[2] J. S. Giraldo, P. P. Vergara, J. C. López, P. H. Nguyen, N. G. Paterakis, A novel linear optimal power flow model for three-phase electrical distribution systems, in: 2020 International Conference on Smart Energy Systems and Technologies (SEST), 2020, pp. 1–6. doi:10.1109/SEST48500.2020.9203557.

[3] P. Schavemaker, L. van der Sluis, Electrical Power System Essentials, Wiley, 2017. URL: https://books.google.nl/books?id=Kz_CDgAAQBAJ.

[4] P. P. Vergara, J. C. Lopez, M. J. Rider, L. C. P. da Silva, Optimal operation of unbalanced three-phase islanded droop-based microgrids, IEEE Trans. Smart Grid 10 (2019) 928–940. doi:10.1109/TSG.2017.2756021.

[5] G. K. V. Raju, P. R. Bijwe, Efficient reconfiguration of balanced and unbalanced distribution systems for loss minimisation, IET Gen. Trans. Distr. 2 (2008) 7–12. doi:10.1049/iet-gtd:20070216.